Stuart Russell
Peter Norvig

Künstliche Intelligenz

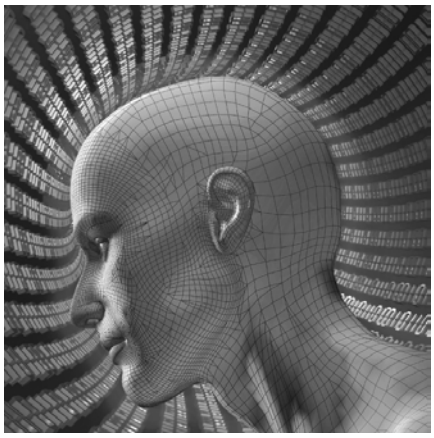
Ein moderner Ansatz

3., aktualisierte Auflage

ALWAYS LEARNING

PEARSON

Künstliche Intelligenz



Stuart Russell
Peter Norvig

Künstliche Intelligenz

Ein moderner Ansatz

3., aktualisierte Auflage

Fachliche Betreuung:
Prof. Dr. Frank Kirchner

PEARSON

Higher Education
München • Harlow • Amsterdam • Madrid • Boston
San Francisco • Don Mills • Mexico City • Sydney
a part of Pearson plc worldwide

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.dnb.de* abrufbar.

Bild AILA, DFKI GmbH, Robotics Innovation Center, Bremen

10 9 8 7 6 5 4 3 2 1

14 13 12

ISBN 978-3-86894-098-5 (Print); 978-3-86326-504-5 (PDF); 978-3-86326-067-5 (EPub)

www.pearson.de
A part of Pearson plc worldwide
Programmleitung: Birger Peil, bpeil@pearson.de
Development: Alice Kachnij, akachnij@pearson.de
Fachlektorat: Prof. Dr. Frank Kirchner, Bremen
Korrektorat: Petra Kienle, Fürstenfeldbruck
Übersetzung: Frank Langenau, Chemnitz
Einbandgestaltung: Thomas Arlt, tarlt@adesso21.net
Titelbild: Shutterstock
Herstellung: Monika Weiher, mweiher@pearson.de
Satz: mediaService, Siegen (www.media-service.tv)
Druck und Verarbeitung: Drukarnia Dimograf, Bielsko-Biala

Printed in Poland

Inhaltsübersicht

Inhaltsverzeichnis

Teil VI Kommunizieren, Wahrnehmen und Handeln

Teil VII Schlussfolgerungen

Vorwort

Künstliche Intelligenz (KI) ist ein umfangreiches Gebiet und dies ist ein umfangreiches Buch. Wir haben versucht, das volle Spektrum des Gebietes abzudecken, das Logik, Wahrscheinlichkeit und stetige Mathematik, Wahrnehmung, Schließen, Lernen und Aktion sowie alles von Geräten aus der Mikroelektronik bis hin zu unbemannten Fahrzeugen zur Planetenerkundung umfasst. Das Buch ist auch deshalb so dick, weil wir stark in die Tiefe gehen.

Der Untertitel dieses Buches lautet „Ein moderner Ansatz". Diese eher leere Phrase soll Ihnen sagen, dass wir versucht haben, alle heute bekannten Fakten in ein gemeinsames Gerüst einzubauen, anstatt zu versuchen, jeden Teilbereich der KI in seinem eigenen historischen Kontext zu erklären. Wir entschuldigen uns bei denjenigen, deren Teilbereiche dadurch weniger deutlich erkennbar sind.

Neu in dieser Auflage

Diese Auflage erfasst die Änderungen in der KI, die seit der letzten Ausgabe in 2003 stattgefunden haben. Hierzu gehören wichtige Anwendungen der KI-Technologie wie beispielsweise der weitverbreitete Einsatz von Spracherkennung, Maschinenübersetzung, autonomen Fahrzeugen und Haushaltsrobotern. Es gibt auch algorithmische Meilensteine, wie etwa die Lösung des Damespieles. Und es lassen sich viele Fortschritte in der Theorie verzeichnen, insbesondere in Bereichen wie probabilistisches Schließen, maschinelles Lernen und Computervision. Der aus unserer Sicht wichtigste Punkt ist die stetige Weiterentwicklung in Bezug auf die Denkweise über das Gebiet. Dies hat sich auch auf die Organisation des Buches ausgewirkt. Die folgende Übersicht nennt die wichtigsten Änderungen:

- Wir gehen verstärkt auf partiell beobachtbare und nichtdeterministische Umgebungen ein, insbesondere in den nicht probabilistischen Szenarios von Suche und Planung. Die Konzepte von *Belief State* (eine Menge möglicher Welten) und *Zustandsabschätzung* (Verwaltung des Belief State) werden in diesen Szenarios eingeführt; später im Buch fügen wir Wahrscheinlichkeiten hinzu.
- Außer den Arten von Umgebungen und Agenten behandeln wir jetzt auch ausführlicher die Arten der *Darstellungen*, die ein Agent verwenden kann. Dabei unterscheiden wir zwischen *atomaren* Darstellungen (in denen jeder Zustand der Welt als Blackbox behandelt wird), *faktorisierten* Darstellungen (in denen ein Zustand eine Menge von Attribut-Wert-Paaren ist) und *strukturierten* Darstellungen (in denen die Welt aus Objekten und Beziehungen zwischen ihnen besteht).
- Zum Thema Planung gehen wir jetzt tiefer auf Kontingenzplanung in partiell beobachtbaren Umgebungen ein und geben einen neuen Ansatz für die hierarchische Planung an.
- Die Ausführungen zu probabilistischen Modellen erster Ordnung wurden erweitert und beschreiben jetzt auch Modelle des offenen Universums für Fälle, wo hinsichtlich der existierenden Objekte Unsicherheit besteht.

- Das einführende Kapitel zum maschinellen Lernen wurde vollständig neu geschrieben. Insbesondere geben wir eine breitere Vielfalt und größere Anzahl moderner Lernalgorithmen an und stellen sie auf ein festeres theoretisches Fundament.
- Websuche und Informationsextraktion sowie Techniken für das Lernen aus sehr großen Datenmengen nehmen jetzt einen größeren Raum ein.
- 20% der Quellenangaben in dieser Ausgabe beziehen sich auf Arbeiten, die nach 2003 veröffentlicht wurden.
- Schätzungsweise 20% des Stoffes sind brandneu. Die restlichen 80% spiegeln ältere Arbeiten wider, wurden aber in großen Teilen neu gefasst, um das Fachgebiet in einem einheitlicheren Bild zu präsentieren.

Überblick über das Buch

Das große Thema, unter dem alles zusammengefasst wird, ist das Konzept eines **intelligenten Agenten**. Wir definieren die KI als die Lehre von Agenten, die Wahrnehmungen aus der Umgebung erhalten und Aktionen ausführen. Jeder dieser Agenten implementiert eine Funktion, die Wahrnehmungsfolgen auf Aktionen abbildet, und wir beschreiben unterschiedliche Möglichkeiten, diese Funktionen darzustellen, wie zum Beispiel reaktive Agenten, Echtzeitplaner, neuronale Netze und entscheidungstheoretische Systeme. Wir erklären die Rolle des Lernens als Erweiterung der Reichweite des Entwicklers in unbekannte Umgebungen und zeigen, wie diese Rolle den Agentenentwurf einschränkt, wobei wir uns besonders auf die explizite Wissensrepräsentation und das Schließen konzentrieren. Wir beschreiben Robotik und Vision nicht als unabhängig definierte Probleme, sondern als Hilfsmittel, ein Ziel zu erreichen. Wir betonen dabei, wie wichtig die Aufgabenumgebung ist, um den zweckmäßigsten Agentenentwurf zu ermitteln.

Unser wichtigstes Ziel ist, die *Konzepte* vorzustellen, die sich in den letzten fünfzig Jahren der KI-Forschung entwickelt haben – ebenso wie verwandte Arbeiten aus den letzten zwei Jahrtausenden. Wir haben versucht, bei der Vorstellung dieser Konzepte übermäßige Formalitäten zu vermeiden und dabei eine ausreichende Genauigkeit beizubehalten. Wo immer es möglich war, haben wir Pseudocode-Algorithmen aufgenommen, um die Konzepte zu konkretisieren. Der dabei verwendete Pseudocode ist in Anhang B beschrieben.

Dieses Buch ist hauptsächlich für die Verwendung im Grundstudium oder in einer Vorlesungsfolge vorgesehen. Das Buch hat 27 Kapitel, die jeweils einer Vorlesungswoche entsprechen; für das gesamte Buch brauchen Sie also zwei Semester. Für eine Vorlesung, die sich nur über ein Semester erstreckt, können ausgewählte Kapitel verwendet werden, die den Interessen des Dozenten und der Studenten entsprechen. Außerdem kann das Buch in einer Vorlesung im Hauptstudium verwendet werden (vielleicht unter zusätzlicher Verwendung einer der in den bibliografischen Hinweisen aufgeführten Quellen). Vorschläge finden Sie auf der Website des Buches unter *aima.cs.berkeley.edu*. Die einzige Voraussetzung ist eine Vertrautheit mit grundlegenden Konzepten der Informatik (Algorithmen, Datenstrukturen, Komplexität) auf einer philosophischen Ebene. Für einige Themen sind mathematische Kenntnisse auf Abiturniveau und lineare Algebra hilfreich; der erforderliche mathematische Hintergrund wird in Anhang A dargestellt.

Am Ende jedes Kapitels sind Übungen angegeben. Übungen, die einigen Programmieraufwand erfordern, sind mit einem **Dreiecksymbol** gekennzeichnet. Am besten lassen sich diese Übungen lösen, wenn Sie das Codearchiv unter *aima.cs.berkeley.edu* nutzen. Einige davon sind umfangreich genug, um als Teamprojekte vergeben zu werden.

Im gesamten Buch sind wichtige Punkte durch ein *Hinweissymbol* gekennzeichnet. Es gibt einen umfassenden Index, mit dessen Hilfe Sie sich im Buch zurechtfinden können. Neu eingeführte Begriffe sind **fett** ausgezeichnet.

Die Website

Link

Die Website für das Buch, *aima.cs.berkeley.edu*, enthält:

- Implementierungen der Algorithmen im Buch in mehreren Programmiersprachen
- Eine Liste von über 1000 Lehrstätten, in denen dieses Buch eingesetzt wird, viele mit Links zu Online-Kursunterlagen und Lehrplänen
- Eine kommentierte Liste mit über 800 Links zu Sites im gesamten Web, wo wichtige KI-Informationen zur Verfügung stehen
- Eine Liste mit ergänzendem Material und Links für die einzelnen Kapitel
- Anleitungen, wie Sie einer Diskussionsgruppe zu dem Buch beitreten können
- Anleitungen, wie Sie den Autoren Fragen oder Kommentare zukommen lassen können
- Anleitungen, wie Sie auf Fehler im Buch aufmerksam machen können (falls es sie gibt)
- Folien und andere Lehrmaterialien für Dozenten

Danksagungen

Dieses Buch wäre nicht möglich gewesen ohne die vielen Mitwirkenden, deren Namen es nicht auf die Titelseite geschafft haben. Jitendra Malik und David Forsyth haben *Kapitel 24* (Computervision) geschrieben und Sebastian Thrun *Kapitel 25* (Robotik). Von Vibhu Mittal stammt ein Teil von *Kapitel 22* (Natürliche Sprache). Nick Hay, Mehran Sahami und Ernest Davis haben einige der Übungen verfasst. Zoran Duric (George Mason), Thomas C. Henderson (Utah), Leon Reznik (RIT), Michael Gourley (Central Oklahoma) und Ernest Davis (NYU) haben das Manuskript durchgesehen und hilfreiche Vorschläge unterbreitet. Wir danken insbesondere Ernie Davis für seine unermüdliche Gabe, mehrere Entwürfe zu lesen und das Buch zu verbessern. Nick Hay brachte die Bibliografie in die richtige Form und verbrachte zum Abgabetermin die Nacht bis um 05:30 Uhr mit Codeschreiben, um das Buch besser zu machen. Jon Barron formatierte und verbesserte die Diagramme in dieser Ausgabe, während Tim Huang, Mark Paskin und Cynthia Bruyns bei den Diagrammen und Algorithmen in vorherigen Ausgaben halfen. Ravi Mohan und Ciaran O'Reilly schreiben und warten die Java-Codebeispiele auf der Website. John Canny hat das Robotik-Kapitel für die erste Ausgabe geschrieben und Douglas Edwards recherchierte die historischen Anmerkungen. Tracy Dunkelberger, Allison Michael, Scott Disanno und Jane Bonnell bei Pearson haben ihr Bestes versucht, um uns im Zeitplan zu halten, und lieferten viele hilfreiche Vorschläge. Am hilfreichsten von allen ist Julie Sussman, P. P. A. gewesen, die jedes Kapitel gelesen und umfangreiche Verbesserungsvorschläge abgegeben hat. Die Korrekturleser in den vorherigen Ausgaben haben uns auf fehlende Kommas hingewiesen und gegebenenfalls alternative Wortvorschläge unterbreitet; Julie wies uns auf fehlende Minuszeichen hin und erkannte x_i, wenn es tatsächlich x_j heißen sollte. Auf jeden im Buch verbliebenen Rechtschreibfehler oder jede verwirrende Erklärung kommen garantiert fünf, die Julie korrigiert hat. Selbst wenn ein Stromausfall sie zwang, bei Kerzenlicht statt LCD-Beleuchtung zu arbeiten, hat sie beharrlich weitergemacht.

Stuart möchte seinen Eltern für ihre stetige Unterstützung und Ermutigung danken, ebenso wie seiner Frau, Loy Sheflott, für ihre endlose Geduld und ihre grenzenlose Weisheit. Er hofft, Gordon, Lucy, George und Isaac werden dies bald lesen, nachdem sie ihm vergeben haben, dass er so lange daran gearbeitet hat. RUGS (Russell's Unusual Group of Students) war wie immer ungewöhnlich hilfreich.

Peter möchte seinen Eltern (Torsten und Gerda) danken, die ihm besonders am Anfang hilfreich zur Seite standen, ebenso wie seiner Frau (Kris), seinen Kindern (Bella und Juliet), Kollegen und Freunden, die ihm Mut gemacht und ihm die langen Stunden des Schreibens und Überarbeitens verziehen haben.

Beide Autoren danken den Bibliothekaren in Berkeley, Stanford, und bei der NASA ebenso wie den Entwicklern von CiteSeer, Wikipedia und Google, die die Art und Weise unserer Nachforschungen revolutioniert haben. Es ist unmöglich, allen Menschen zu danken, die das Buch verwendet und Vorschläge eingebracht haben, aber besonderer Dank für die hilfreichen Hinweise gebührt Gagan Aggarwal, Eyal Amir, Ion Androutsopoulos, Krzysztof Apt, Warren Haley Armstrong, Ellery Aziel, Jeff Van Baalen, Darius Bacon, Brian Baker, Shumeet Baluja, Don Barker, Tony Barrett, James Newton Bass, Don Beal, Howard Beck, Wolfgang Bibel, John Binder, Larry Bookman, David R. Boxall, Ronen Brafman, John Bresina, Gerhard Brewka, Selmer Bringsjord, Carla Brodley, Chris Brown, Emma Brunskill, Wilhelm Burger, Lauren Burka, Carlos Bustamante, Joao Cachopo, Murray Campbell, Norman Carver, Emmanuel Castro, Anil Chakravarthy, Dan

Chisarick, Berthe Choueiry, Roberto Cipolla, David Cohen, James Coleman, Julie Ann Comparini, Corinna Cortes, Gary Cottrell, Ernest Davis, Tom Dean, Rina Dechter, Tom Dietterich, Peter Drake, Chuck Dyer, Doug Edwards, Robert Egginton, Asma'a El-Budrawy, Barbara Engelhardt, Kutluhan Erol, Oren Etzioni, Hana Filip, Douglas Fisher, Jeffrey Forbes, Ken Ford, Eric Fosler-Lussier, John Fosler, Jeremy Frank, Alex Franz, Bob Futrelle, Marek Galecki, Stefan Gerberding, Stuart Gill, Sabine Glesner, Seth Golub, Gosta Grahne, Russ Greiner, Eric Grimson, Barbara Grosz, Larry Hall, Steve Hanks, Othar Hansson, Ernst Heinz, Jim Hendler, Christoph Herrmann, Paul Hilfinger, Robert Holte, Vasant Honavar, Tim Huang, Seth Hutchinson, Joost Jacob, Mark Jelasity, Magnus Johansson, Istvan Jonyer, Dan Jurafsky, Leslie Kaelbling, Keiji Kanazawa, Surekha Kasibhatla, Simon Kasif, Henry Kautz, Gernot Kerschbaumer, Max Khesin, Richard Kirby, Dan Klein, Kevin Knight, Roland Koenig, Sven Koenig, Daphne Koller, Rich Korf, Benjamin Kuipers, James Kurien, John Lafferty, John Laird, Gus Larsson, John Lazzaro, Jon LeBlanc, Jason Leatherman, Frank Lee, Jon Lehto, Edward Lim, Phil Long, Pierre Louveaux, Don Loveland, Sridhar Mahadevan, Tony Mancill, Jim Martin, Andy Mayer, John McCarthy, David McGrane, Jay Mendelsohn, Risto Miikkulanien, Brian Milch, Steve Minton, Vibhu Mittal, Mehryar Mohri, Leora Morgenstern, Stephen Muggleton, Kevin Murphy, Ron Musick, Sung Myaeng, Eric Nadeau, Lee Naish, Pandu Nayak, Bernhard Nebel, Stuart Nelson, XuanLong Nguyen, Nils Nilsson, Illah Nourbakhsh, Ali Nouri, Arthur Nunes-Harwitt, Steve Omohundro, David Page, David Palmer, David Parkes, Ron Parr, Mark Paskin, Tony Passera, Amit Patel, Michael Pazzani, Fernando Pereira, Joseph Perla, Wim Pijls, Ira Pohl, Martha Pollack, David Poole, Bruce Porter, Malcolm Pradhan, Bill Pringle, Lorraine Prior, Greg Provan, William Rapaport, Deepak Ravichandran, Ioannis Refanidis, Philip Resnik, Francesca Rossi, Sam Roweis, Richard Russell, Jonathan Schaeffer, Richard Scherl, Hinrich Schuetze, Lars Schuster, Bart Selman, Soheil Shams, Stuart Shapiro, Jude Shavlik, Yoram Singer, Satinder Singh, Daniel Sleator, David Smith, Bryan So, Robert Sproull, Lynn Stein, Larry Stephens, Andreas Stolcke, Paul Stradling, Devika Subramanian, Marek Suchenek, Rich Sutton, Jonathan Tash, Austin Tate, Bas Terwijn, Olivier Teytaud, Michael Thielscher, William Thompson, Sebastian Thrun, Eric Tiedemann, Mark Torrance, Randall Upham, Paul Utgoff, Peter van Beek, Hal Varian, Paulina Varshavskaya, Sunil Vemuri, Vandi Verma, Ubbo Visser, Jim Waldo, Toby Walsh, Bonnie Webber, Dan Weld, Michael Wellman, Kamin Whitehouse, Michael Dean White, Brian Williams, David Wolfe, Jason Wolfe, Bill Woods, Alden Wright, Jay Yagnik, Mark Yasuda, Richard Yen, Eliezer Yudkowsky, Weixiong Zhang, Ming Zhao, Shlomo Zilberstein und unserem geschätzten Kollegen, dem anonymen Kritiker.

Die Autoren

Stuart Russell wurde 1962 in Portsmouth in England geboren. Er erhielt seinen B.A. mit First-Class Honours in Physik an der Oxford University 1982 und seinen Doktortitel in Informatik in Stanford 1986. Er trat der Fakultät der Universität of California in Berkeley bei, wo er Professor für Informatik, Direktor des Center for Intelligent Systems und Inhaber des Smith-Zadeh Chair in Engineering ist. 1990 erhielt er den Presidential Young Investigator Award of the National Science Foundation und 1995 war er Mitgewinner des Computers and Thought Award. Er war 1996 Miller Professor der University of California und wurde 2000 zum Chancellor Professor berufen. 1998 gab er die Forsythe Memorial Lectures an der Stanford University. Er ist Fellow und ehemaliges Vorstandsmitglied der American Association for Artificial Intelligence. Er hat über 100 Artikel zu den unterschiedlichsten Themen der künstlichen Intelligenz veröffentlicht. Unter anderem hat er auch die Bücher *The Use of Knowledge in Analogy and Induction* und (zusammen mit Eric Wefald) *Do the Right Thing: Studies in Limited Rationality* geschrieben.

Peter Norvig ist Forschungsleiter bei Google, Inc. und war von 2002 bis 2005 der verantwortliche Leiter für die Kernalgorithmen zur Websuche. Er ist Fellow der American Association for Artificial Intelligence und der Association for Computing Machinery. Früher war er Leiter der Abteilung für Computerwissenschaften im Ames Research Center der NASA, wo er für die Forschung und Entwicklung in den Bereichen der künstlichen Intelligenz und Robotik der NASA verantwortlich war. Zuvor war er leitender Wissenschaftler bei Junglee, wo er half, einen der ersten Internet-Informationsextrahierungsdienste zu entwickeln. Er hat seinen B.S. in angewandter Mathematik von der Brown University sowie einen Doktortitel in Informatik von der Universität of California in Berkeley. Er wurde mit den Distinguished Alumni and Engineering Innovation Awards von Berkeley und der Exceptional Achievement Medal von der NASA geehrt. Er war Professor an der University of Southern California und Forschungsfakultätsmitglied in Berkeley. Zu seinen Buchveröffentlichungen gehören *Paradigms of AI Programming: Case Studies in Common Lisp*, *Verbmobil: A Translation System for Face-to-Face Dialog* und *Intelligent Help Systems for UNIX*.

TEIL I

Künstliche Intelligenz

I

Einführung

1

In diesem Kapitel wollen wir erklären, warum wir die künstliche Intelligenz für so ein interessantes Gebiet halten, das sich lohnt zu erforschen. Wir wollen versuchen zu definieren, worum es sich dabei genau handelt – eine sinnvolle Maßnahme, bevor wir wirklich anfangen.

Wir bezeichnen uns selbst als *Homo sapiens* – den weisen Menschen –, weil unsere **Intelligenz** für uns so wichtig ist. Tausende von Jahren haben wir versucht zu verstehen, *wie wir denken*; d.h., wie eine Handvoll Materie eine Welt, die weit komplizierter als sie selbst ist, wahrnehmen, verstehen, vorhersagen und manipulieren kann. Das Gebiet der **künstlichen Intelligenz**, KI, geht noch weiter: Es versucht nicht nur, Intelligenz zu verstehen, sondern den Beweis für das Verständnis zu führen, indem es intelligente, technische Systeme erschafft.

KI gehört zu den neueren Gebieten in Wissenschaft und Technik. Ernsthafte Arbeiten begannen unmittelbar nach dem Zweiten Weltkrieg und ihr Name wurde 1956 geprägt. Zusammen mit der Molekularbiologie wird KI von Wissenschaftlern in anderen Disziplinen regelmäßig als das Gebiet bezeichnet, „in dem ich am liebsten arbeiten würde". Ein Student der Physik mag das berechtigte Gefühl haben, dass alle guten Ideen bereits von Galileo, Newton, Einstein und anderen vor ihm gedacht wurden. KI dagegen hält noch viele offene Türen für mehrere Vollzeit-Einsteins und -Edisons bereit.

Die KI beinhaltet momentan eine große Vielzahl von Unterbereichen, von allgemeinen Bereichen (wie beispielsweise Lernen und Wahrnehmung) bis hin zu speziellen Bereichen wie beispielsweise Schach spielen, das Beweisen mathematischer Theoreme, das Schreiben von Gedichten, das Führen eines Fahrzeuges auf einer belebten Straße oder die Krankheitsdiagnose. KI ist für jede intellektuelle Aufgabe relevant. Es handelt sich um ein wirklich universelles Gebiet.

1.1 Was ist KI?

Wir haben behauptet, KI sei interessant, aber wir haben nicht gesagt, *worum* es sich dabei handelt. ▶ Abbildung 1.1 zeigt acht Definitionen künstlicher Intelligenz in einem zweidimensionalen Layout. Die oberen Definitionen haben mit *Denkprozessen* und *logischem Schließen* zu tun, während sich die unteren mit dem *Verhalten* beschäftigen. Die Definitionen auf der linken Seite messen den Erfolg in Bezug auf die Wiedergabetreue *menschlicher* Leistung, während sich die Definitionen auf der rechten Seite auf eine *ideale* Leistungsgröße – die sogenannte **Rationalität** – beziehen. Ein System ist rational, wenn es das seinen Kenntnissen entsprechende „Richtige" macht.

In der Vergangenheit wurden alle vier Ansätze der KI verfolgt, jeweils von verschiedenen Leuten mit unterschiedlichen Methoden. Ein um den Menschen zentrierter Ansatz muss teilweise eine empirische Wissenschaft sein, die sich mit Beobachtungen und Hypothesen zu menschlichem Verhalten löst. Ein rationalistischer[1] Ansatz verwendet eine Kombination aus Mathematik und Ingenieurwissenschaft. Die einzelnen

1 Die Unterscheidung zwischen *menschlichem* und *rationalem* Verhalten bedeutet nicht, dass Menschen unbedingt „irrational" im Sinne von „emotional instabil" oder „geistesgestört" sind. Man muss nur bedenken, dass wir nicht perfekt sind: Nicht alle Schachspieler sind Großmeister und leider erhält nicht jeder die beste Note in seinem Examen. Einige systematische Fehler in der menschlichen Logik werden von Kahnemann et al. (1982) aufgelistet.

Ansätze haben sich gegenseitig sowohl geschadet als auch unterstützt. Wir wollen die vier Ansätze jetzt genauer betrachten.

Menschliches Denken	Rationales Denken
„Das spannende, neuartige Unterfangen, Computern das Denken beizubringen, ... Maschinen mit Verstand im wahrsten Sinne des Wortes." (Haugeland, 1985) „[Die Automatisierung von] Aktivitäten, die wir dem menschlichen Denken zuordnen, Aktivitäten wie beispielsweise Entscheidungsfindung, Problemlösung, Lernen ..." (Bellman, 1978)	„Die Studie mentaler Fähigkeiten durch die Nutzung programmiertechnischer Modelle." (Charniak und McDermott, 1985) „Das Studium derjenigen mathematischen Formalismen, die es ermöglichen, wahrzunehmen, logisch zu schließen und zu agieren." (Winston, 1992)
Menschliches Handeln	**Rationales Handeln**
„Die Kunst, Maschinen zu schaffen, die Funktionen erfüllen, die, werden sie von Menschen ausgeführt, der Intelligenz bedürfen." (Kurzweil, 1990) „Das Studium des Problems, Computer dazu zu bringen, Dinge zu tun, bei denen ihnen momentan der Mensch noch überlegen ist." (Rich und Knight, 1991)	„Computerintelligenz ist die Studie des Entwurfs intelligenter Agenten." (Poole et al., 1998) „KI ... beschäftigt sich mit intelligentem Verhalten in künstlichen Maschinen." (Nilsson, 1998)

Abbildung 1.1: Einige Definitionen künstlicher Intelligenz, angeordnet in vier Kategorien.

1.1.1 Menschliches Handeln: der Ansatz mit dem Turing-Test

Der **Turing-Test**, entwickelt von Alan Turing (1950), war darauf ausgelegt, eine zufriedenstellende operative Definition von Intelligenz anzugeben. Ein Computer besteht den Test, wenn ein menschlicher Fragesteller, der einige schriftliche Fragen stellt, nicht erkennen kann, ob die schriftlichen Antworten von einem Menschen stammen oder nicht. *Kapitel 26* beschreibt diesen Test ausführlich und geht auch der Frage nach, ob ein Computer wirklich intelligent ist, wenn er diesen Test besteht. Es sei hier darauf hingewiesen, dass es eine Fülle von Aufgaben zu bearbeiten gibt, wenn man einen Computer programmieren will, der den Turing-Test im strengen Sinne bestehen muss. Der Computer muss die folgenden Eigenschaften besitzen:

- **Verarbeitung natürlicher Sprache**, was es ihm ermöglicht, erfolgreich in Englisch zu kommunizieren
- **Wissensrepräsentation**, damit er speichern kann, was er weiß oder hört
- **Automatisches logisches Schließen**, um anhand der gespeicherten Informationen Fragen zu beantworten und neue Schlüsse zu ziehen
- **Maschinenlernen**, um sich an neue Umstände anzupassen sowie Muster zu erkennen und zu extrapolieren

Der Test von Turing vermied bewusst eine direkte physische Interaktion zwischen dem Fragesteller und dem Computer, weil die *physische* Simulation einer Person für die Intelligenz nicht erforderlich ist. Der sogenannte **totale Turing-Test** verwendet jedoch ein Videosignal, sodass der Fragesteller die wahrnehmungsorientierten Fähigkeiten des Prüflings testen kann. Außerdem dient es als Möglichkeit für den Fragesteller, phy-

sische Objekte „weiterzugeben“. Um den vollen Turing-Test zu bestehen, braucht der Computer

- **Computervision**, um Objekte wahrzunehmen, und
- **Roboter**, um Objekte zu manipulieren und zu bewegen.

Diese sechs Disziplinen bilden einen Großteil der KI und Turing verdient Lob für den Entwurf eines Tests, der noch 60 Jahre später relevant ist. Bisher haben die KI-Forscher allerdings wenig Aufwand betrieben, den Turing-Test zu bestehen, weil sie glaubten, es sei wichtiger, die zugrunde liegenden Prinzipien der Intelligenz zu verstehen, als ein System zu bauen. Die Sehnsucht nach „künstlichem Fliegen“ wurde erfüllt, als die Gebrüder Wright und andere endlich aufhörten, Vögel zu imitieren, und begannen, Windkanäle einzusetzen und sich mit Aerodynamik zu beschäftigten. Lehrbücher der Luft- und Raumfahrtkunde definieren das Ziel ihres Forschungsgebietes eben nicht als den „Bau von Maschinen, die genau wie Tauben fliegen, sodass sie sogar andere Tauben täuschen können“.

1.1.2 Menschliches Denken: der Ansatz der kognitiven Modellierung

Wenn wir sagen, ein bestimmtes Programm denkt wie ein Mensch, müssen wir irgendwie festlegen können, wie Menschen denken. Wir müssen *in* die Arbeitsweise des menschlichen Gehirns eindringen. Dafür gibt es drei Möglichkeiten: durch Introspektion (wobei man versucht, eigene Gedanken zu analysieren, während sie entstehen), durch psychologische Experimente (durch Beobachtung handelnder Personen) sowie über Hirntomografie (wobei das aktive Gehirn beobachtet wird). Sobald wir eine ausreichend genaue Theorie des Verstandes besitzen, wird es möglich, diese Theorie als Computerprogramm auszudrücken. Wenn die Eingaben und Ausgaben des Programms menschlichem Verhalten entsprechen, ist dies ein Indiz dafür, dass einige der Mechanismen des Programms auch im Menschen funktionieren können. Beispielsweise waren Allen Newell und Herbert Simon, die GPS – den „General Problem Solver“ – entwickelten (Newell und Simon, 1961), nicht einfach damit zufrieden, dass ihr Programm Probleme korrekt löste. Sie beschäftigten sich mehr damit, seine Schritte beim logischen Schließen mit denen von Menschen zu vergleichen, die dieselben Probleme lösen. Das interdisziplinäre Gebiet der **Kognitionswissenschaft** bringt die Computermodelle aus der KI sowie die experimentellen Techniken aus der Psychologie zusammen, um exakte und überprüfbare Theorien zur Arbeitsweise des menschlichen Verstandes zu konstruieren.

Die Kognitionswissenschaft ist an sich ein faszinierendes Gebiet, das mehrere Lehrbücher und wenigstens eine Enzyklopädie verdiente (Wilson und Keil, 1999). Gelegentlich werden wir auf Ähnlichkeiten oder Unterschiede zwischen KI-Techniken und der menschlichen Kognition hinweisen. Die eigentliche Kognitionswissenschaft im Sinne des Studiums biologischer, kognitiver Prozesse basiert jedoch notwendigerweise auf einer experimentellen Untersuchung von Menschen oder Tieren. Dies überlassen wir aber anderen Büchern und gehen davon aus, dass der Leser für seine Experimente nur Zugriff auf einen Computer hat.

In den ersten Jahren der KI gab es häufig Verwechslungen zwischen den Ansätzen: Ein Autor behauptete, dass ein Algorithmus für eine Aufgabe gut geeignet sei und dass er *deshalb* ein gutes Modell der menschlichen Vorgehensweise darstelle, oder umgekehrt. Moderne Autoren unterscheiden zwischen den beiden Ansätzen – durch diese

Unterscheidung war es möglich, dass sich sowohl die KI als auch die Kognitionswissenschaft entwickeln konnten. Jedoch befruchten sich die beiden Gebiete weiterhin gegenseitig. Dies gilt zurzeit besonders auf dem Gebiet der Computervision, das verstärkt neurophysiologische Ergebnisse in Berechnungsmodelle einfließen lässt.

1.1.3 Rationales Denken: der Ansatz der „Denkregeln"

Der griechische Philosoph Aristoteles war einer der ersten, der versuchte, „Denken" zu formalisieren, d.h. unwiderlegbare Prozesse für logisches Schließen festzulegen. Sein **Syllogismus** stellte Muster für Argumentstrukturen bereit, die immer zu korrekten Schlüssen führten, wenn ihnen korrekte Prämissen übergeben wurden – zum Beispiel: „Sokrates ist ein Mensch; alle Menschen sind sterblich; deshalb ist auch Sokrates sterblich." Diese Regeln sollten die Vorgehensweise des Verstandes abbilden; ihre Untersuchung eröffnete ein Gebiet, das als **Logik** bezeichnet wird.

Die Logiker des 19. Jahrhunderts entwickelten eine exakte Notation für Aussagen zu allen Arten von Dingen in der Welt sowie die Beziehungen zwischen ihnen. (Im Gegensatz dazu steht die gewöhnliche arithmetische Notation, die nur Aussagen über Zahlen liefert.) Bis 1965 existierten Programme, die im Prinzip *jedes* lösbare Problem lösen konnten, das in logischer Notation beschrieben war (obwohl das Programm möglicherweise in eine Endlosschleife gerät, wenn keine Lösung existiert). Die sogenannte **logistische** Tradition innerhalb der künstlichen Intelligenz hofft, auf solchen Programmen aufbauen zu können, um intelligente Systeme zu erzeugen.

Diesem Ansatz stehen zwei große Hindernisse gegenüber. Erstens ist es nicht einfach, formloses Wissen aufzunehmen und es mithilfe formaler Begriffe so darzustellen, wie es die Logik erfordert. Dies gilt insbesondere dann, wenn das Wissen nicht sicher ist. Zweitens gibt es einen großen Unterschied zwischen der Fähigkeit, ein Problem „im Prinzip" zu lösen und dies auch in der Praxis zu tun. Selbst Probleme, die mit nur einigen hundert Fakten abgebildet werden können, können die Rechenleistung eines Computers überfordern, es sei denn, es gibt gewisse Richtlinien, beispielsweise welche Schlussfolgerungen als Erstes ausprobiert werden sollen. Obwohl diese beiden Probleme prinzipiell für *jeden* Versuch gelten, logisches Schließen in Computern zu realisieren, werden sie zuerst in der Logik formuliert.

1.1.4 Rationales Handeln: der Ansatz der rationalen Agenten

Ein **Agent** ist einfach etwas, das agiert (*Agent* kommt vom lateinischen *agere* – tun, handeln, machen). Natürlich tun alle Computerprogramme etwas, doch von Computeragenten erwartet man, dass sie mehr tun: autonom operieren, ihre Umgebung wahrnehmen, über einen längeren Zeitraum beständig sein, sich an Änderungen anpassen sowie Ziele erzeugen und verfolgen. Ein **rationaler Agent** ist ein Agent, der sich so verhält, dass er das beste Ergebnis erzielt, oder, falls es Unsicherheiten gibt, das beste erwartete Ergebnis.

Im KI-Ansatz der „Denkregeln" lag die Betonung auf korrekten logischen Schlussfolgerungen. Das Ziehen korrekter Schlüsse ist manchmal *Teil* des rationalen Agenten, denn eine Möglichkeit, rational zu handeln, besteht gerade darin, sein Handeln an logischem Schließen auszurichten. Andererseits stellen korrekte Schlussfolgerungen nicht die *ganze* Rationalität dar, weil es bestimmte Situationen gibt, in denen man das beweisbar Richtige nicht tut. Darüber hinaus gibt es Möglichkeiten, rational zu agieren, ohne Schlussfolge-

rungen zu ziehen. Beispielsweise ist das Zurückschrecken vor einer heißen Herdplatte eine Reflexaktion, die normalerweise erfolgreicher ist, als erst lange zu planen.

Alle Fertigkeiten, die für den Turing-Test benötigt werden, erlauben es auch einem Agenten, rational zu handeln. Durch Wissensrepräsentation und Schlussfolgern können Agenten gute Entscheidungen treffen. Wir müssen in der Lage sein, verständliche Sätze in natürlicher Sprache zu bilden, um in einer komplexen Gesellschaft bestehen zu können. Wir lernen nicht nur des Lernens wegen, sondern weil sich dadurch unsere Fähigkeiten verbessern, effektives Verhalten zu generieren.

Tipp

Gegenüber den anderen Ansätzen hat das Konzept des rationalen Agenten zwei Vorteile. Erstens ist es allgemeiner als der Ansatz der „Denkregeln", weil korrektes Schlussfolgern nur einen Mechanismus von vielen zur Erzielung der Rationalität darstellt. Zweitens ist es eher geeignet, den Fortschritt zu befördern als Ansätze, die sich menschliches Handeln zum Vorbild nehmen oder auf menschlichen Gedanken basieren. Rationalität ist mathematisch klar definiert und allgemein gültig und daher besser geeignet, überprüfbare Agentenentwürfe zu erzeugen. Das menschliche Verhalten dagegen ist an eine bestimmte Umgebung gut angepasst und praktisch durch die Summe der Dinge definiert, die Menschen tun. *Dieses Buch konzentriert sich deshalb auf allgemeine Prinzipien rationaler Agenten sowie auf Komponenten, aus denen sie erzeugt werden können*. Wir werden sehen, dass trotz der offensichtlichen Einfachheit, mit der sich das Problem ausdrücken lässt, eine enorme Vielfalt an Problemen entstehen kann, wenn wir versuchen, es zu lösen. *Kapitel 2* beschäftigt sich mit einigen dieser Probleme detaillierter.

Ein wichtiger Aspekt ist dabei stets zu beachten: Wir werden schnell erkennen, dass die perfekte Rationalität – also immer das Richtige zu tun – in komplexen Umgebungen nicht erreichbar ist. Die Anforderungen an die Computerleistung sind einfach zu hoch. Für einen Großteil dieses Buches wollen wir jedoch die Arbeitshypothese einführen, dass eine perfekte Rationalität ein guter Ausgangspunkt für die Analyse ist. Damit wird das Problem vereinfacht und eine geeignete Einstellung für die meisten Grundlagen auf diesem Gebiet bereitgestellt. *Kapitel 5* und *17* beschäftigen sich explizit mit dem Problem der **begrenzten Rationalität** – wie man angemessen handelt, wenn nicht genügend Zeit bleibt, alle wünschenswerten Berechnungen vorzunehmen.

1.2 Die Grundlagen der künstlichen Intelligenz

In diesem Abschnitt geben wir einen kurzen Abriss der Disziplinen, der Ideen, Anschauungen und Techniken, die zur KI beigetragen haben. Wie bei jedem Abriss müssen wir uns auf eine kleine Anzahl von Menschen, Ereignissen und Gedanken konzentrieren und andere ignorieren, die möglicherweise ebenfalls wichtig waren. Wir stellen einige zentrale Fragen in den Mittelpunkt der Diskussion.[2]

2 Wir wollen den Eindruck vermeiden, diese Fragen seien die einzigen, um die es in diesen Disziplinen geht, oder dass die Disziplin die KI als ultimatives Ziel anstrebt.

1.2.1 Philosophie

- Können formale Regeln verwendet werden, um gültige Schlüsse zu ziehen?
- Wie entsteht aus einem physischen Gehirn der mentale Verstand?
- Woher stammt Wissen?
- Wie führt Wissen zu einer Aktion?

Aristoteles (384–322 v.Chr.) formulierte als Erster eine exakte Menge von Gesetzen, die die rationale Komponente des Verstandes definierten. Er entwickelte ein formloses System von Syllogismen für ein korrektes logisches Schließen, das es ihm im Prinzip erlaubte, mechanisch Schlüsse zu ziehen, wenn bestimmte Prämissen als Ausgangspunkt vorhanden waren. Viel später hatte Raimundus Lullus (auch Ramon Llull, 1232–1316) die Idee, dass ein sinnvolles logisches Schließen durch eine mechanische Maschine durchgeführt werden könnte. Thomas Hobbes (1588–1679) schlug vor, logisches Schließen mit einer numerischen Berechnung zu vergleichen, bei der wir „unsere stillen Gedanken addieren und subtrahieren". Die Automatisierung der eigentlichen Berechnung war zu dieser Zeit bereits im vollen Gange. Um 1500 entwickelte Leonardo da Vinci (1452–1519) eine mechanische Rechenmaschine, setzte sie aber nicht in die Praxis um; Rekonstruktionen in neuer Zeit haben bewiesen, dass sein Entwurf funktioniert hätte. Die erste bekannte Rechenmaschine wurde etwa 1623 von dem deutschen Wissenschaftler Wilhelm Schickard (1592–1635) entwickelt. Berühmter ist aber die 1642 von Blaise Pascal (1623–1662) gebaute Pascaline. Pascal schrieb: „Die arithmetische Maschine erzeugt Ergebnisse, die dem Verstand näher zu kommen scheinen als alles, was Tiere zu leisten im Stande sind." Gottfried Wilhelm Leibniz (1646–1716) baute ein mechanisches Gerät, das Operationen in Bezug auf Konzepte statt auf Zahlen ausführen sollte, wobei aber der Anwendungsbereich dieses Gerätes recht begrenzt war. Leibniz übertraf Pascal, indem er eine Rechenmaschine baute, die addieren, multiplizieren und Wurzel ziehen konnte, während die Pascaline nur in der Lage war, zu addieren und zu subtrahieren. Es gab auch Spekulationen, dass Maschinen nicht nur Berechnungen ausführen, sondern auch denken und eigenständig handeln könnten. In seinem Buch *Leviathan* von 1651 beschreibt Thomas Hobbes die Idee eines „künstlichen Tieres" und sagt dazu: „Denn was ist das Herz anderes als eine Feder, was sind die Nerven anderes als lauter Stränge und die Gelenke anderes als lauter Räder?"

Es ist eine Sache zu sagen, dass der Verstand – zumindest teilweise – entsprechend logischer Regeln arbeitet, und technische Systeme aufbaut, die einige dieser Regeln simulieren; eine andere ist es, den Verstand selbst als ein derartiges technisches System zu betrachten. René Descartes (1596–1650) präsentierte die erste klare Diskussion zur Unterscheidung zwischen Verstand und Materie sowie der daraus entstehenden Probleme. Ein Problem mit einer ausschließlich technischen Konzeption des Verstandes ist, dass damit scheinbar wenig Raum für den freien Willen bleibt: Wenn der Verstand völlig von logischen Regeln gesteuert wird, hat er auch nicht mehr freien Willen als ein Stein, der „entscheidet", zum Mittelpunkt der Erde zu fallen. Descartes war ein starker Verfechter der Leistung logischer Schlüsse beim Verstehen der Welt, einer Philosophie, die man heute **Rationalismus** nennt, und einer, zu deren Verfechter Aristoteles und Leibniz zählen. Doch Descartes war auch ein Befürworter des **Dualismus**. Er war der Ansicht, dass es einen Teil des menschlichen Verstandes (oder der Seele oder des Geistes) gibt, der außerhalb der Natur liegt und nicht den physikalischen Gesetzen unterliegt. Tiere dagegen besäßen diese dualen Qualitäten nicht; sie könnten als

Maschinen behandelt werden. Eine Alternative zum Dualismus ist der **Materialismus**, der besagt, dass die Arbeitsweise des Gehirns gemäß den Gesetzen der Physik den Verstand *bildet*. Der freie Wille ist einfach die Art und Weise, wie die wählende Entität die verfügbaren Alternativen wahrnimmt.

Betrachtet man ein Gehirn, das Wissen verarbeitet, ist das nächste Problem die Einrichtung der Wissensquelle. Die **Empirismus**-Bewegung, beginnend mit *Novum Organum*[3] von Francis Bacon (1561–1626), wird durch eine Aussage von John Locke (1630–1704) charakterisiert: „Wir verstehen nicht, was wir nicht zuvor gedacht haben." David Hume (1711–1776) schlug in seinem *A Treatise of Human Nature* (Hume, 1739) vor, was man heute als das Prinzip der **Induktion** bezeichnet: Allgemeine Regeln werden erworben, indem man sich wiederholten Assoziationen zwischen ihren Elementen aussetzt. Aufbauend auf der Arbeit von Ludwig Wittgenstein (1889–1951) und Bertrand Russel (1872–1970) entwickelte der berühmte Wiener Zirkel unter der Leitung von Rudolf Carnap (1891–1970) die Doktrin des **logischen Positivismus**. Diese Doktrin besagt, dass das gesamte Wissen durch logische Theorien charakterisiert werden kann, die letztlich mit **Beobachtungssätzen** verbunden sind, die sensorischen Eingaben entsprechen; logischer Positivismus kombiniert somit Rationalismus und Empirismus.[4] Die **Bestätigungstheorie** von Carnap und Carl Hempel (1905–1997) versuchte zu analysieren, wie Wissen aus Erfahrung erworben wird. Carnaps Buch, *Der logische Aufbau der Welt* (1928), definierte eine explizite rechentechnische Prozedur für das Extrahieren von Wissen aus elementaren Erfahrungen. Es handelte sich dabei wahrscheinlich um die erste Theorie des Verstandes als Rechenprozess.

Das letzte Element in der philosophischen Betrachtung des Verstandes ist die Verknüpfung zwischen Wissen und Handeln. Diese Frage ist für die KI zentral, weil für Intelligenz sowohl Aktion als auch logisches Schließen erforderlich sind. Außerdem können wir nur durch ein Verständnis dafür, wie Aktionen gerechtfertigt werden, verstehen, wie wir einen Agenten erstellen können, dessen Aktionen zu rechtfertigen (d.h. rational) sind. Aristoteles argumentierte (in *De Motu Animalium*), dass Aktionen gerechtfertigt seien, wenn es eine logische Verknüpfung zwischen Zielen und Wissen über das Ergebnis der Aktion gibt:

> *Aber wie kann es sein, dass das Denken manchmal von Aktion begleitet wird und manchmal nicht, manchmal von Bewegung und manchmal nicht? Es scheint, als ob fast dasselbe passiert wie beim logischen Schließen, wobei Schlussfolgerungen über sich nicht verändernde Objekte gezogen werden. Aber in diesem Fall ist das Ziel eine Spekulation ... während hier der Schluss, der aus zwei Prämissen stammt, eine Aktion ist. ... ich brauche Kleidung; ein Mantel ist Kleidung. Ich brauche einen Mantel. Was ich brauche, muss ich machen; ich brauche einen Mantel. Ich muss einen Mantel machen. Und die Schlussfolgerung „Ich muss einen Mantel machen" ist eine Aktion.*

3 Das *Novum Organum* ist eine Aktualisierung des *Organon* von Aristoteles oder Gedankeninstrument. Somit kann Aristoteles sowohl als Empiriker als auch als Rationalist gelten.

4 Bei dieser Analogie können alle sinnvollen Aussagen bestätigt oder widerlegt werden, indem man entweder Experimente durchführt oder die Bedeutung der Wörter analysiert. Weil dies, wie beabsichtigt, einen Großteil der Metaphysik ausschließt, war der logische Positivismus in manchen Kreisen nicht sehr beliebt.

In der *Nikomachischen Ethik* (Buch III. 3, 1112b) beschäftigt sich Aristoteles weiter mit diesem Thema und schlägt einen Algorithmus vor:

> *Das Hin und Her unserer Überlegung richtet sich nicht auf das Ziel, sondern auf die Wege zum Ziel. Ein Arzt überlegt sich nicht, ob er heilen, ein Redner nicht, ob er überzeugen, ein Staatsmann nicht, ob er einen wohl geordneten Staat schaffen soll, und auch sonst gibt es kein Schwanken über das Ziel. Sondern: Das Ziel wird aufgestellt, und dann setzt das Überlegen ein, wie und auf welchen Wegen es erreicht werden kann. Und wenn sich mehrere Wege bieten, dann sucht man den leichtesten und besten zu erkennen. Gibt es nur einen einzigen Weg zur Verwirklichung, so wird überlegt, wie es auf diesem möglich sei und auf welch weiterem Weg eben dieser eine hinwiederum erreicht werden könnte – so lange, bis man zur ersten Ursache gelangt, die in der Reihenfolge des Findens das Letzte ist. Denn wer überlegt, der scheint, in der geschilderten Weise, zu suchen und analytisch zu verfahren, wie es bei der Lösung geometrischer Konstruktionsaufgaben üblich ist. Freilich ist offenbar nicht jegliches Suchen ein Überlegen, z.B. nicht das Suchen nach mathematischen Lösungen, dagegen ist jedes Überlegen ein Suchen – und was beim Aufstellen der Analyse das Letzte ist, scheint bei der Verwirklichung das Erste zu sein. Und wenn der Mensch auf etwas Unmögliches stößt, nimmt er von seinem Vorhaben Abstand, z.B. wenn er dazu Geld braucht, dieses aber unmöglich herbeigeschafft werden kann. Erscheint aber das Vorgehen als möglich, so nimmt er die Sache in die Hand.*

Der Algorithmus von Aristoteles wurde 2300 Jahre später von Newell und Simon in ihrem GPS-Programm implementiert. Wir bezeichnen es jetzt als Regressionsplanungssystem (siehe *Kapitel 10*).

Die zielbasierte Analyse ist praktisch, sagt aber nichts darüber aus, was das Ziel ist, wenn mehrere Aktionen zu einem Ziel führen könnten oder wenn keine Aktion es vollständig realisiert. Antoine Arnauld (1612–1694) beschrieb korrekt eine quantitative Formel für die Entscheidung, welche Aktion in solchen Fällen ergriffen werden sollte (siehe *Kapitel 16*). Das Buch von John Stuart Mill (1806–1873), *Utilitarianism* (Mill, 1863), unterstützte die Idee der rationalen Entscheidung in allen Bereichen menschlicher Aktivitäten. Die formalere Entscheidungstheorie wird im folgenden Abschnitt beschrieben.

1.2.2 Mathematik

- Wie lauten die formalen Regeln, um gültige Schlüsse zu ziehen?
- Was kann berechnet werden?
- Wie argumentieren wir mit unsicheren Informationen?

Die Philosophen umrissen bereits einige grundlegende Ideen der KI, aber für den Schritt zu einer formalen Wissenschaft war eine gewisse mathematische Formalisierung in fundamentalen Bereichen erforderlich: Logik, Berechenbarkeit, Algorithmik und Wahrscheinlichkeit.

Die Idee einer formalen Logik lässt sich bis zu den alten griechischen Philosophen zurückverfolgen, doch begann ihre mathematische Entwicklung praktisch erst mit der Arbeit von George Boole (1815–1864), der die Details der Aussagen- oder Booleschen Logik ausarbeitete (Boole, 1847). Gottlob Frege (1848–1925) erweiterte 1879 die Logik von Boole, um Objekte und Relationen einzubinden. Daraus entstand die heute ver-

wendete Logik erster Stufe (First-Order-Logik).[5] Alfred Tarski (1902–1983) führte eine Referenztheorie ein, die zeigt, wie man Objekte in einer Logik mit Objekten der realen Welt verknüpfen kann.

Im nächsten Schritt waren die Grenzen zu erforschen, was mit Logik und Algorithmik möglich war. Als erster nicht trivialer **Algorithmus** gilt der Euklidische Algorithmus für die Berechnung des größten gemeinsamen Teilers. Das Wort *Algorithmus* (und die Idee zum Studium von Algorithmen) geht bis zu al-Khowarazmi zurück, einem persischen Mathematiker des 9. Jahrhunderts, dessen Arbeiten auch die arabischen Ziffern und die Algebra nach Europa brachten. Boole und andere diskutierten Algorithmen für logisches Schließen und zum Ende des 19. Jahrhunderts wurden Anstrengungen unternommen, allgemeine mathematische Beweise als logische Deduktion zu formalisieren. Kurt Gödel (1906–1978) zeigte 1930, dass es eine effektive Prozedur gibt, um jede richtige Aussage in der Logik erster Stufe von Frege und Russell zu beweisen, doch war diese Logik erster Ordnung nicht in der Lage, das Prinzip der mathematischen Induktion zu erfassen, das notwendig ist, um die natürlichen Zahlen zu charakterisieren. Im Jahre 1931 zeigte Gödel, dass Einschränkungen in Bezug auf Deduktion existieren. Sein **Unvollständigkeitstheorem** zeigte, dass es in jeder formalen Theorie, die so streng wie Peano-Arithmetik (die elementare Theorie natürlicher Zahlen) ist, wahre Aussagen gibt, die in dem Sinne unvorhersagbar sind, dass sie innerhalb der Theorie keinen Beweis haben.

Dieses grundlegende Ergebnis lässt sich auch so interpretieren, dass einige Funktionen auf ganzen Zahlen nicht durch einen Algorithmus dargestellt – d.h. nicht berechnet – werden können. Dies motivierte Alan Turing (1912–1954), zu versuchen, genau zu charakterisieren, welche Funktionen **berechenbar** sind – d.h. berechnet werden können. Dieses Konzept ist immer noch etwas problematisch, weil die Idee einer Berechnung oder effektiven Prozedur schwer formal definiert werden kann. Allerdings wird die Church-Turing-These, die aussagt, dass die Turing-Maschine (Turing, 1936) in der Lage ist, jede berechenbare Funktion zu berechnen, im Allgemeinen als hinreichende Definition akzeptiert. Turing zeigte außerdem, dass es Funktionen gab, die keine Turing-Maschine berechnen kann. Beispielsweise kann keine Maschine *allgemein* sagen, ob ein bestimmtes Programm bei einer vorgegebenen Eingabe terminiert oder unendlich ausgeführt wird.

Obwohl Entscheidbarkeit und Berechenbarkeit wichtig für ein Verständnis der Algorithmik sind, hatte das Konzept der **Handhabbarkeit** einen sehr viel größeren Einfluss. Vereinfacht ausgedrückt, wird ein Problem als nicht handhabbar bezeichnet, wenn die Zeit für die Lösung von Instanzen des Problems mit der Größe der Instanzen exponentiell wächst. Die Unterscheidung zwischen polynomiellem und exponentiellem Wachstum im Hinblick auf die Komplexität wurde zuerst Mitte der 1960er Jahre genauer betrachtet (Cobham, 1964; Edmonds, 1965). Sie ist wichtig, weil ein exponentielles Wachstum bedeutet, dass selbst mittelgroße Instanzen nicht in einer angemessenen Zeit gelöst werden können. Man sollte also versuchen, das Gesamtproblem – intelligentes Verhalten zu erzeugen – in handhabbare statt in nicht handhabbare Teilprobleme zu zerlegen.

Wie lässt sich nun ein nicht handhabbares Problem erkennen? Eine Methode hierfür bietet die Theorie der **NP-Vollständigkeit**, die auf Steven Cook (1971) und Richard Karp

5 Die von Frege vorgeschlagene Notation für die First-Order-Logik hat sich nie durchgesetzt, wie aus einfachen Beispielen leicht ersichtlich ist.

(1972) zurückgeht. Cook und Karp zeigten die Existenz großer Klassen kanonischer kombinatorischer Probleme beim Suchen und Schlussfolgern, die NP-vollständig sind. Jede Problemklasse, auf die die Klasse der NP-vollständigen Probleme reduziert werden kann, ist wahrscheinlich nicht handhabbar. (Auch wenn es noch keinen Beweis gibt, dass NP-vollständige Probleme zwangsläufig nicht handhabbar sind, sind die meisten Theoretiker davon überzeugt.) Diese Ergebnisse stehen im Gegensatz zu dem Optimismus, mit dem die Boulevardpresse die ersten Computer begrüßt hatte – „elektronische Superhirne", die „Schneller als Einstein!" waren. Trotz der steigenden Geschwindigkeit von Computern zeichnen sich intelligente Systeme durch sorgfältige Nutzung von Ressourcen aus. Salopp ausgedrückt: Die Welt ist eine extrem große Probleminstanz! Arbeiten zur KI haben bei der Erklärung geholfen, warum bestimmte Instanzen NP-vollständige Probleme schwierig, andere dagegen einfach sind (Cheeseman et al., 1991).

Neben der Logik und der Berechnung ist der dritte große Beitrag der Mathematik zur KI die Theorie der **Wahrscheinlichkeit**. Der Italiener Gerolamo Cardano (1501–1576) formulierte als Erster die Idee der Wahrscheinlichkeit und beschrieb sie im Hinblick auf mögliche Ergebnisse von Spielereignissen. Blaise Pascal (1623–1662) zeigte 1654 in einem Brief an Pierre Fermat (1601–1665), wie man den Ausgang eines abgebrochenen Glücksspieles vorhersagen und den Spielern durchschnittliche Auszahlungen zuweisen kann.

Wahrscheinlichkeit wurde schnell zu einem unverzichtbaren Bestandteil aller quantitativen Wissenschaften und half, mit unsicheren Messungen und unvollständigen Theorien umzugehen. James Bernoulli (1654–1705), Pierre Laplace (1749–1827) und andere entwickelten die Theorie weiter und führten neue statistische Methoden ein. Thomas Bayes (1702–1761) schlug eine Regel für das Aktualisieren von Wahrscheinlichkeiten unter Berücksichtigung neuer Beweise vor. Auf der Regel von Bayes beruhen die meisten modernen Ansätze für unbestimmte Schlussfolgerungen in KI-Systemen.

1.2.3 Wirtschaftswissenschaft

- Wie sollen wir Entscheidungen treffen, um eine maximale Rendite zu erzielen?
- Wie sollen wir das tun, was andere nicht tun?
- Wie sollen wir vorgehen, wenn die Rendite möglicherweise weit in der Zukunft liegt?

Die Wirtschaftswissenschaft hatte ihre Geburt 1776, als der schottische Philosoph Adam Smith (1723–1790) sein Werk *An Inquiry into the Nature and Causes of the Wealth of Nations* veröffentlichte. Während die alten Griechen und andere zu wirtschaftlichen Überlegungen beitrugen, war Smith der Erste, der die Wirtschaft als Wissenschaft behandelte, wobei er die Idee verfolgte, dass die Wirtschaftssysteme als aus einzelnen Agenten bestehend betrachtet werden können, die ihr eigenes wirtschaftliches Wohlergehen maximieren. Die meisten Menschen stellen sich unter der Wirtschaftswissenschaft nur Geld vor, aber die Wirtschaftswissenschaftler sagen, dass sie eigentlich untersuchen, wie Menschen Entscheidungen treffen, die zu bevorzugten Ergebnissen führen. Wenn McDonalds einen Hamburger für einen Euro anbietet, behauptet das Unternehmen, dass es den Euro bevorzugt und hofft, dass die Kunden den Hamburger bevorzugen. Die mathematische Behandlung der „bevorzugten Ergebnisse" oder des **Nutzens** (der „Nützlichkeit") wurde als Erstes von Léon Walras (1834–1910) formalisiert und von Frank Ramsey (1931) und später von John von Neumann und Oskar Morgenstern in ihrem Buch *The Theory of Games and Economic Behaviour* (1944) verbessert.

Die **Entscheidungstheorie**, die die Wahrscheinlichkeitstheorie mit der Nützlichkeitstheorie kombiniert, stellt ein formales und vollständiges Gerüst für (wirtschaftliche und andere) Entscheidungen bereit, die unter unsicheren Bedingungen getroffen werden – d.h., in Fällen, in denen *probabilistische* Beschreibungen die Umgebung des Entscheidungsfinders hinreichend beschreiben. Dies ist geeignet für „große" Wirtschaftssysteme, wo kein Agent Aktionen der anderen Agenten als Individuen berücksichtigen muss. Für „kleine" Wirtschaftssysteme ist die Situation sehr viel mehr mit einem **Spiel** vergleichbar: Die Aktionen eines Spielers können den Nutzen eines anderen wesentlich beeinflussen (entweder positiv oder negativ). Die von von Neumann und Morgenstern entwickelte **Spieltheorie** (siehe auch Luce und Raiffa, 1957) beinhaltete das überraschende Ergebnis, dass bei bestimmten Spielen ein rationaler Agent nach Richtlinien handeln sollte, die zufallsorientiert sind (oder zumindest zu sein scheinen). Im Unterschied zur Entscheidungstheorie bietet die Spieltheorie keine eindeutige Beschreibung für das Auswählen von Aktionen.

Größtenteils kümmerten sich Wirtschaftswissenschaftler nicht um die dritte der oben aufgelisteten Fragen, wie nämlich rationale Entscheidungen getroffen werden, wenn die Rendite aus den Aktionen nicht unmittelbar ist, sondern stattdessen aus mehreren in Folge getroffenen Aktionen entsteht. Dieses Thema wurde im Gebiet der **Operationsforschung** verfolgt, die während des Zweiten Weltkrieges aus den Bemühungen Großbritanniens entstand, Radarinstallationen zu optimieren. Später fand sie zivile Anwendung in komplexen Managemententscheidungen. Die Arbeit von Richard Bellmann (1957) formalisiert eine Klasse sequenzieller Entscheidungsprobleme, die sogenannten **Markov'schen Entscheidungsprozesse**, auf die wir in den *Kapiteln 17* und *21* genauer eingehen werden.

Die Arbeit auf den Gebieten der Wirtschaftswissenschaft und Operationsforschung hat einen Großteil zu unserem Konzept rationaler Agenten beigetragen und dennoch arbeiteten die KI-Forscher jahrelang auf vollkommen getrennten Wegen. Ein Grund dafür war die offensichtliche Komplexität bei der Findung rationaler Entscheidungen. Herbert Simon (1916–2001), der Pionier auf dem Gebiet der KI-Forschung, gewann 1978 den Nobelpreis in Wirtschaftswissenschaften für sein frühes Werk. Darin zeigte er, dass Modelle, die auf **Anspruchserfüllung** basierten – wobei Entscheidungen getroffen werden, die „gut genug" sind, anstatt in aufwändigen Berechnungen eine optimale Entscheidung zu finden –, eine bessere Beschreibung des tatsächlichen menschlichen Verhaltens darstellen (Simon, 1947). In den 1990er Jahren gab es ein neu erwachtes Interesse an entscheidungstheoretischen Techniken für Agentensysteme (Wellman, 1995).

1.2.4 Neurowissenschaft

■ Wie verarbeitet das Gehirn Informationen?

Die **Neurowissenschaft** ist die Erforschung des Nervensystems, insbesondere des Gehirns. Obwohl die genaue Art und Weise, wie das Gehirn Denken ermöglicht, eines der größten Rätsel der Wissenschaft ist, wird die Tatsache, dass es Denken erlaubt, seit Tausenden von Jahren anerkannt, weil bewiesen ist, dass schwere Kopfverletzungen zu mentalen Ausfällen führen können. Man weiß auch schon lange, dass sich menschliche Gehirne irgendwie unterscheiden; etwa 335 v. Chr. schrieb Aristoteles: „Von allen Tieren hat der Mensch proportional zu seiner Größe das größte Gehirn."[6] Den-

6 In der Zwischenzeit wurde erkannt, dass das Spitzhörnchen (Scandentia) ein größeres Verhältnis von Gehirn- zu Körpermasse hat.

noch erkannte man erst in der Mitte des 18. Jahrhunderts, dass das Gehirn der Sitz des Bewusstseins ist. Zuvor stellte man sich unter anderem vor, das Bewusstsein befinde sich im Herzen oder in der Milz.

Die Untersuchungen von Paul Broca (1824–1880) im Hinblick auf die Aphasie (Sprachdefizit) bei hirngeschädigten Patienten im Jahr 1861 demonstrierten die Existenz lokalisierter Bereiche im Gehirn, die für bestimmte kognitive Funktionen verantwortlich sind. Insbesondere zeigte er, dass sich das Sprachzentrum in einem Teil der linken Gehirnhälfte befindet, der heute als Broca-Bereich bezeichnet wird.[7] Damals war bekannt, dass das Gehirn aus Nervenzellen besteht, sogenannten **Neuronen**, aber erst 1873 entwickelte Camillo Golgi (1843–1926) eine Färbungstechnik, die die Beobachtung einzelner Neuronen im Gehirn erlaubte (siehe ▶ Abbildung 1.2). Diese Technik wurde von Santiago Ramon y Cajal (1852–1934) in seinen frühen Studien der neuronalen Strukturen des Gehirns verwendet.[8] Nicolas Rashevsky (1936, 1938) war der Erste, der mathematische Modelle auf die Untersuchung des Nervensystems anwandte.

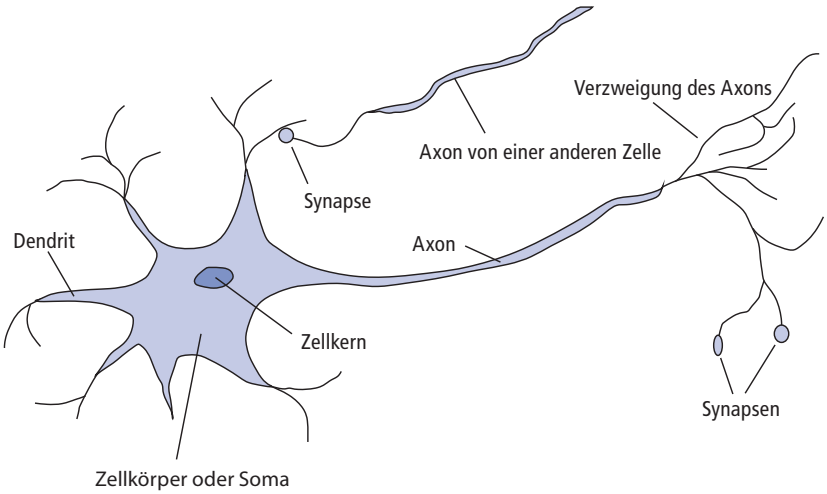


Abbildung 1.2: Die Bestandteile einer Nervenzelle, eines sogenannten Neurons. Jedes Neuron besteht aus einem Zellkörper, dem sogenannten Soma, der einen Zellkern enthält. Aus dem Zellkörper heraus verzweigen sich mehrere Fasern, die sogenannten Dendriten, und eine einzelne lange Faser, das Axon. Das Axon erstreckt sich über eine lange Distanz, die sehr viel größer sein kann, als der Maßstab in dieser Zeichnung darstellt. Normalerweise ist ein Axon einen Zentimeter lang (das ist das Hundertfache des Durchmessers des Zellkörpers), es kann aber auch bis zu einem Meter lang sein. Ein Neuron knüpft Verbindungen mit 10 bis 100.000 anderen Neuronen, und zwar an Verbindungspunkten, die als Synapsen bezeichnet werden. Signale werden mithilfe einer komplizierten elektrochemischen Reaktion von Neuron zu Neuron weitergegeben. Die Signale steuern auf kurze Sicht die Aktivität des Gehirns und ermöglichen auch Änderungen auf lange Sicht in der Konnektivität der Neuronen. Man nimmt an, dass diese Mechanismen die Grundlage für das Lernen im Gehirn sind. Ein Großteil der Informationsverarbeitung erfolgt im cerebralen Cortex, der Hirnrinde, die die äußere Schicht des Gehirns bildet. Die grundlegende organisatorische Einheit scheint ein Gewebestrang zu sein, der mit rund 0,5 Millimeter Durchmesser etwa 20.000 Neuronen enthält und sich bis zur vollen Tiefe der Gehirnrinde erstreckt, die beim Menschen etwa vier Millimeter beträgt.

7 Häufig wird auch Alexander Hood (1824) als mögliche frühere Quelle zitiert.

8 Golgi blieb bei seinem Glauben, dass die Funktionen des Gehirns hauptsächlich in einem kontinuierlichen Medium ausgeführt wurden, in das die Neuronen eingebettet waren, während Cajal die „neuronale Doktrin“ veröffentlichte. Beide Forscher erhielten 1906 den Nobelpreis gemeinsam, doch ihre Dankesreden waren eher gegensätzlich.

Heute haben wir Daten über die Zuordnung zwischen bestimmten Bereichen des Gehirns und den Körperteilen, die sie steuern oder von denen sie sensorische Eingaben erhalten. Solche Zuordnungen können sich innerhalb weniger Wochen radikal ändern und einige Tiere scheinen mehrere Zuordnungen zu besitzen. Darüber hinaus wissen wir noch nicht wirklich, wie andere Bereiche Funktionen übernehmen können, wenn ein Bereich beschädigt wird. Es gibt fast keine Theorie darüber, wie das Gedächtnis eines Menschen gespeichert wird.

Die Messung gesunder Gehirnaktivität begann 1929 mit der Erfindung des Elektroencephalographen (EEG) durch Hans Berger. Die neueste Entwicklung funktionaler Magnetresonanzbilder (fMRI, functional Magnetic Resonance Imaging) (Ogawa et al., 1990) bietet den Neurowissenschaftlern völlig neue detaillierte Bilder der Gehirnaktivität und ermöglicht damit Messungen, die auf interessante Weise fortlaufenden kognitiven Prozessen entsprechen. Sie werden unterstützt durch Fortschritte in der Einzelzell-Aufzeichnung der Neuronenaktivitäten. Einzelne Neuronen lassen sich elektrisch, chemisch oder sogar optisch (Han und Boyden, 2007) stimulieren, wodurch neuronale Ein-/Ausgabe-Beziehungen abgebildet werden können. Trotz dieser Fortschritte sind wir jedoch noch weit davon entfernt, zu verstehen, wie kognitive Prozesse tatsächlich funktionieren.

	Supercomputer	Personalcomputer	Menschliches Gehirn
Recheneinheiten	10^4 CPUs, 10^{12} Transistoren	4 CPUs, 10^9 Transistoren	10^{11} Neuronen
Speichereinheiten	10^{14} Bit RAM	10^{11} Bit RAM	10^{11} Neuronen
	10^{15} Bit Festplatte	10^{13} Bit Festplatte	10^{14} Synapsen
Zykluszeit	10^{-9} s	10^{-9} s	10^{-3} s
Operationen/s	10^{15}	10^{10}	10^{17}
Speicheraktualisierungen/s	10^{14}	10^{10}	10^{14}

Abbildung 1.3: Ein ungefährer Vergleich der reinen Rechenressourcen, die einem IBM BLUE GENE-Supercomputer, einem typischen Personalcomputer in 2008 und einem menschlichen Gehirn zur Verfügung stehen. Die Werte des Gehirns sind praktisch feststehend, während die Zahlen für Supercomputer etwa alle fünf Jahre um den Faktor 10 steigen und sich somit dem Gehirn nähern konnten. Personalcomputer hinken bei allen Kennzahlen bis auf die Zykluszeit hinterher.

Tipp

Der wirklich faszinierende Schluss ist, *dass eine Ansammlung einfacher Zellen zu Verstand, Aktion und Bewusstsein führen kann* oder mit den markigen Worten von John Searle (1992), dass Gehirne Geist verursachen. Die einzige wirkliche alternative Theorie ist die Mystik: dass es irgendeinen mystischen Bereich gibt, in dem Gedanken arbeiten und der jenseits der physischen Wissenschaften liegt.

Gehirne und digitale Computer weisen unterschiedliche Eigenschaften auf. Wie aus den Angaben zur Zykluszeit in ▶ Abbildung 1.3 hervorgeht, arbeiten Computer eine Million Mal schneller als ein Gehirn. Das Gehirn gleicht dies mit weit mehr Speicherkapazität und Zwischenverbindungen aus, als selbst ein High-End-Personalcomputer besitzt, wobei aber die größten Supercomputer mit einer Kapazität aufwarten, die der

des Gehirns ähnlich ist. (Allerdings ist zu beachten, dass das Gehirn scheinbar nicht alle seine Neuronen gleichzeitig verwendet.) Futuristen verweisen gern auf diese Zahlen, wobei sie auf eine nahende **Singularität** hinweisen, bei der Computer eine übermenschliche Leistungsebene erreichen (Vinge, 1993; Kurzweil, 2005). Die Zahlenvergleiche allein sind jedoch nicht besonders informativ. Selbst mit einem Computer, der über eine scheinbar unbegrenzte Kapazität verfügt, würden wir immer noch nicht wissen, wie das Intelligenzniveau des Gehirns erreicht werden kann.

1.2.5 Psychologie

■ Wie denken und handeln Menschen und Tiere?

Die Ursprünge der wissenschaftlichen Psychologie werden häufig auf die Arbeit des deutschen Physikers Hermann von Helmholtz (1821–1894) und seines Studenten Wilhelm Wundt (1832–1920) zurückgeführt. Helmholtz wendete die wissenschaftliche Methode auf die Untersuchung der menschlichen visuellen Wahrnehmung an und sein *Handbuch der physiologischen Optik* wird heute als „die wichtigste Abhandlung über die Physiologie der menschlichen visuellen Wahrnehmung“ bezeichnet (Nalwa, 1993, Seite 15). 1879 eröffnete Wundt das erste Labor experimenteller Psychologie an der Universität Leipzig. Wundt bestand auf sorgfältig kontrollierten Experimenten, in denen seine Mitarbeiter eine wahrnehmende oder introspektive Aufgabe ausführten, während sie ihre Denkprozesse analysieren. Die sorgfältige Kontrolle führte irgendwann dazu, dass die Psychologie zu einer Wissenschaft wurde, aber die subjektive Natur der Daten machte es unwahrscheinlich, dass bei einem Experiment der Durchführende seine eigenen Theorien je widerlegen würde. Dagegen verfügten Biologen, die das Verhalten von Tieren untersuchten, nicht über introspektive Daten und entwickelten eine objektive Methodik, wie von H. S. Jennings (1906) in seiner einflussreichen Arbeit *Behavior of the Lower Organisms* beschrieben. Die Anwendung dieser Perspektive auf Menschen, die **Behaviorismus**-Bewegung, angeführt von John Watson (1878–1958), wies *jede* Theorie zurück, an der mentale Prozesse beteiligt waren, weil diese Introspektion keinen zuverlässigen Beweis darstellen konnte. Die Behavioristen beharrten darauf, nur objektive Messungen der Wahrnehmungen (oder *Stimulus*) zu betrachten, die einem Tier bereitgestellt wurden, sowie die entsprechenden Aktionen (oder die *Antwort*). Behavioristen fanden zwar eine Menge über Ratten und Tauben heraus, hatten aber weniger Erfolg beim Verstehen von Menschen.

Kognitive Psychologie, die das Gehirn als informationsverarbeitendes Gerät betrachtet, kann mindestens bis zu den Arbeiten von William James (1842–1910) zurückverfolgt werden. Helmholtz bestand außerdem darauf, dass die Wahrnehmung eine Art unbewusste logische Schlussfolgerung beinhalte. Die kognitive Perspektive wurde weitgehend vom Behaviorismus in Amerika überdeckt, aber am Lehrstuhl für angewandte Psychologie unter der Leitung von Frederic Bartlett (1886–1969) erlebte die kognitive Modellierung ihre Blütezeit. Das Werk *The Nature of Explanation* von Bartletts Schüler und Nachfolger, Kenneth Craik (1943), etablierte die Legitimität solcher „mentalen“ Begriffe wie Glauben und Ziele wieder kraftvoll, mit der Begründung, sie seien genauso wissenschaftlich wie etwa die Verwendung von Druck und Temperatur für die Beschreibung von Gasen, obwohl diese aus Molekülen bestehen, die keine dieser Eigenschaften besitzen. Craik spezifizierte die drei wichtigsten Schritte eines wissensbasierten Agenten: (1) Der Stimulus muss in eine interne Darstellung übersetzt werden, (2) die Darstellung wird von kognitiven Prozessen manipuliert, um neue interne Darstellungen abzu-

leiten, und (3) diese wiederum werden zurück in eine Aktion übersetzt. Er erklärte deutlich, warum dies ein sinnvolles Design für einen Agenten ist:

> *Wenn der Organismus ein „kleineres Modell" externer Realität und seiner eigenen möglichen Aktionen im Kopf trägt, ist er in der Lage, verschiedene Alternativen auszuprobieren, die beste davon zu erkennen, auf zukünftige Situationen zu reagieren, bevor diese auftreten, das Wissen über vergangene Ereignisse im Umgang mit der Gegenwart und der Zukunft zu nutzen und in jedem Fall auf viel vollständigere, sicherere und kompetentere Weise auf die Anforderungen zu reagieren, denen er gegenübersteht. (Craik, 1943)*

Nachdem Craik 1945 bei einem Fahrradunfall ums Leben gekommen war, wurde seine Arbeit von Donald Broadbent fortgesetzt, dessen Buch *Perception and Communication* (1958) zu den ersten Arbeiten zählt, die psychologische Phänomene als Informationsverarbeitung modellieren. In der Zwischenzeit führte in Amerika die Entwicklung der Computermodellierung zur Entstehung des Gebietes der **Kognitionswissenschaft**. Man kann sagen, dieses Gebiet sei mit einem Workshop im September 1956 am MIT entstanden. (Wir werden noch sehen, dass dies gerade zwei Monate nach der Konferenz war, bei der die KI selbst „geboren" wurde.) Im Workshop stellte George Miller *The Magic Number Seven* vor, Noam Chomsky präsentierte *Three Models of Language* und Allen Newell und Herbert Simon stellten *The Logic Theory Machine* vor. Diese drei einflussreichen Arbeiten zeigten, wie Computermodelle verwendet werden konnten, um damit die Psychologie des Gedächtnisses, der Sprache und des logischen Denkens zu betrachten. Unter Psychologen gilt heute die übliche (wenn auch bei weitem nicht allgemeine) Ansicht, dass „eine kognitive Theorie mit einem Computerprogramm vergleichbar sein sollte" (Anderson, 1980), d.h., sie sollte einen detaillierten informationsverarbeitenden Mechanismus beschreiben, mit dem eine kognitive Funktion implementiert werden könnte.

1.2.6 Technische Informatik

■ Wie können wir einen effizienten Computer bauen?

Für den Erfolg der künstlichen Intelligenz brauchen wir zwei Dinge: Intelligenz und ein Werkzeug. Der Computer ist das Werkzeug der Wahl. Der moderne digitale elektronische Computer wurde unabhängig und fast gleichzeitig von Wissenschaftlern in drei Ländern entwickelt, die am Zweiten Weltkrieg beteiligt waren. Der erste *funktionsfähige* Computer war der elektromechanische Heath Robinson,[9] 1940 von dem Team um Alan Turing für einen einzigen Zweck erstellt: die Dechiffrierung deutscher Nachrichten. 1943 entwickelte dieselbe Gruppe den Colossus, eine leistungsfähige Universalmaschine, die auf Vakuumröhren basierte.[10] Der erste funktionsfähige *programmierbare* Computer war der Z-3, eine Erfindung von Konrad Zuse in Deutschland, 1941. Zuse erfand außerdem Gleitkommazahlen und die erste höhere Programmiersprache, Plankalkül. Der erste *elektronische* Computer, der ABC, wurde von John Atanasof und seinem Schüler Clifford Berry

9 Heath Robinson war ein Karikaturist, der für seine Darstellung skurriler und absurd komplizierter Vorrichtungen für alltägliche Aufgaben wie beispielsweise das Streichen eines Butterbrotes bekannt war.

10 In der Zeit nach dem Krieg wollte Turing diesen Computer für die KI-Forschung verwenden – beispielsweise für eines der ersten Schachprogramme (Turing et al., 1953). Seine Bemühungen stießen in der britischen Regierung auf Ablehnung.

zwischen 1940 und 1942 an der Iowa State University zusammengebaut. Die Forschungen von Atanasof erfuhren nur wenig Unterstützung und Anerkennung; es war der ENIAC, der als Teil eines geheimen Militärprojektes an der Universität Pennsylvania von einem Team entwickelt wurde, dem auch John Mauchly und John Eckert angehörten, und der sich als der einflussreichste Vorläufer moderner Computer erwies.

Seit damals hat jede Generation von Computerhardware eine Steigerung im Hinblick auf Geschwindigkeit und Kapazität sowie eine Verringerung in Bezug auf den Preis mit sich gebracht. Die Leistung hat sich etwa alle 18 Monate verdoppelt. Dies gilt bis etwa in das Jahr 2005, als die Hersteller aufgrund von Problemen mit der Verlustleistung dazu übergingen, die Anzahl der CPU-Kerne und nicht mehr die Taktgeschwindigkeit zu erhöhen. Derzeit geht man davon aus, dass zukünftige Leistungssteigerungen durch massive Parallelisierung realisiert werden – eine merkwürdige Konvergenz zu den Eigenschaften des Gehirns.

Natürlich gab es auch vor dem elektronischen Computer schon Rechenmaschinen. Die frühesten automatisierten Maschinen, die aus dem 17. Jahrhundert stammen, wurden bereits in *Abschnitt 1.2* angesprochen. Die erste programmierbare Maschine war ein Webstuhl, der 1805 von Joseph Marie Jacquard (1752–1834) entwickelt wurde und Lochkarten verwendete, um die Anweisungen für die zu webenden Muster zu speichern. Mitte des 19. Jahrhunderts entwarf Charles Babbage (1792–1871) zwei Maschinen, die er beide nicht fertigstellte. Die „Differenzmaschine“ sollte mathematische Tabellen für den Maschinenbau und wissenschaftliche Projekte berechnen. Schließlich hat man sie 1991 im Wissenschaftsmuseum von London fertig gebaut und ihre Funktionsfähigkeit demonstriert (Swade, 2000). Die „analytische Maschine“ von Babbage war noch viel ambitionierter: Sie beinhaltete adressierbaren Speicher, gespeicherte Programme und bedingte Sprünge und war die erste Maschine, die allgemeine Berechnungen durchführen konnte. Die Kollegin von Babbage, Ada Lovelace, Tochter des Dichters Lord Byron, war vielleicht der erste Programmierer der Welt. (Die Programmiersprache Ada ist nach ihr benannt.) Sie schrieb Programme für die nicht fertiggestellte analytische Maschine und dachte sogar darüber nach, ob diese Maschine nicht auch Schach spielen oder Musik komponieren könnte.

Die KI verdankt natürlich auch vieles der Softwareseite der Informatik, die die Betriebssysteme, Programmiersprachen und Werkzeuge für die Entwicklung moderner Programme auf diesem Feld bereitgestellt hat (und die Veröffentlichungen dazu). Hier haben sich die Mühen gelohnt: Die Arbeiten für die KI brachten viele Ideen hervor, die wiederum in die etablierte Informatik eingeflossen sind, unter anderem Time Sharing, interaktive Interpreter, PCs mit Fenstern und Mäusen, schnelle Entwicklungsumgebungen, den Datentyp der verketteten Liste, automatische Speicherverwaltung sowie Schlüsselkonzepte symbolischer, funktionaler, deklarativer und objektorientierter Programmierung.

1.2.7 Regelungstheorie und Kybernetik

■ Wie können künstliche Maschinen unter eigener Steuerung arbeiten?

Ktesibios von Alexandria (ca. 250 v. Chr.) baute die erste sich selbst steuernde Maschine: eine Wasseruhr mit einem Regulator, der einen konstanten Durchfluss gewährleistete. Diese Erfindung änderte die Definition dessen, was eine künstliche Maschine machen konnte. Zuvor konnten nur lebendige Dinge ihr Verhalten als Reaktion auf Umgebungsänderungen anpassen. Weitere Beispiele für selbst einstellende rückgekoppelte Rege-

lungssysteme sind unter anderem der von James Watt (1736–1819) entwickelte Dampfmaschinenregler sowie der Thermostat, erfunden von Cornelis Drebbel (1572–1633), der auch das U-Boot erfand. Die mathematische Theorie stabiler Regelkreise wurde im 19. Jahrhundert entwickelt.

Die zentrale Figur bei der Entwicklung der heute sogenannten **Regelungstheorie** war Norbert Wiener (1894–1964). Wiener war ein brillanter Mathematiker, der unter anderem mit Bertrand Russell zusammenarbeitete, bevor er begann, Interesse an biologischen und mechanischen Kontrollsystemen und ihrer Verbindung zur Kognition zu entwickeln. Wie Craik (der Regelungssysteme auch als psychologische Modelle benutzte) forderten Wiener und seine Kollegen Arturo Rosenblueth und Julian Bigelow die behavioristische Orthodoxie heraus (Rosenblueth et al., 1943). Sie betrachteten ein zweckbestimmtes Verhalten, das aus einem regulativen Mechanismus heraus entsteht, der versucht, den „Fehler" – den Unterschied zwischen dem aktuellen Zustand und dem Zielzustand – zu minimieren. Ende der 40er Jahre organisierte Wiener zusammen mit Warren McCulloch, Walter Pitts und John von Neumann eine Reihe wichtiger Konferenzen, die die neuen mathematischen und programmiertechnischen Modelle der Kognition erklärten. Wieners Buch, *Kybernetik* (1948), wurde zu einem Bestseller und machte die Öffentlichkeit auf die Möglichkeit von Maschinen mit künstlicher Intelligenz aufmerksam. In der Zwischenzeit bereitete Ross Ashby (Ashby, 1940) in Großbritannien den Weg für ähnliche Ideen. Ashby, Alan Turing, Grey Walter und andere bildeten den *Ratio Club* für „diejenigen, die die Ideen von Wiener hatten, bevor Wieners Buch erschienen ist". In seinem Werk *Design for a Brain* (1948, 1952) führte Ashby seine Idee näher aus, dass Intelligenz durch Geräte – **Homöostat** genannt – geschaffen werden könne, die mit geeigneten Rückkopplungsschleifen ein stabiles adaptives Verhalten erreichen.

Die moderne Regelungstheorie, insbesondere der Zweig, den man als stochastische Optimierung bezeichnet, hat den Entwurf von Systemen zum Ziel, die über die Zeit eine **Zielfunktion** maximieren. Das entspricht in etwa unserer Betrachtung der KI als dem Entwurf von Systemen, die sich optimal verhalten. Warum aber sind dann KI und die Regelungstheorie zwei unterschiedliche Gebiete, insbesondere wenn man die engen Verbindungen ihrer Gründer betrachtet? Die Antwort liegt in der engen Kopplung zwischen den mathematischen Techniken, mit denen die Teilnehmer vertraut waren, und der zugehörigen Problemmenge, die in der jeweiligen Betrachtung der Welt enthalten war. Kalkül und Matrixalgebra, die Werkzeuge der Regelungstheorie, führten von sich aus zu Systemen, die sich durch feste Mengen stetiger Variablen beschreiben lassen, während KI zum Teil ins Leben gerufen wurde, um diesen erkannten Beschränkungen zu entfliehen. Die Werkzeuge der logischen Schlussfolgerungen und der Programmierung erlaubten es den KI-Forschern, Probleme wie Sprache, Sehen und Planung zu betrachten, die völlig außerhalb des Zuständigkeitsbereiches der Regelungstheoretiker lagen.

1.2.8 Linguistik

■ Wie hängen Sprache und Denken zusammen?

1957 veröffentlichte B.F. Skinner das Buch *Verbal Behavior*. Dabei handelt es sich um eine umfassende, detaillierte Beschreibung des behavioristischen Ansatzes zur Spracherlernung, geschrieben von dem wichtigsten Experten auf diesem Gebiet. Seltsamerweise

wurde eine Besprechung dieses Buches genauso bekannt wie das eigentliche Buch und führte dazu, das Interesse am Behaviorismus so gut wie verschwinden zu lassen. Der Autor dieser Besprechung war der Linguist Noam Chomsky, der gerade ein Buch über seine eigene Theorie veröffentlicht hatte, *Syntactic Structures*. Chomsky zeigte, dass die behavioristische Theorie das Konzept der Kreativität in der Sprache nicht berücksichtigen konnte – sie erklärte nicht, wie ein Kind Sätze verstehen und erzeugen konnte, die es nie zuvor gehört hatte. Die Theorie von Chomsky – die auf syntaktischen Modellen basierte, welche auf den indischen Linguistiker Panini zurückgehen (etwa 350 v. Chr.) – konnte dies erklären und anders als vorhergehende Theorien war sie formal genug, dass sie im Prinzip programmiert werden konnte.

Die moderne Linguistik und die KI wurden also etwa gleichzeitig „geboren“ und wuchsen zusammen auf, mit einer Schnittmenge in einem hybriden Gebiet, der sogenannten **Computerlinguistik** oder **natürlichen Sprachverarbeitung**. Das Problem, Sprache zu verstehen, stellte sich bald als deutlich komplizierter heraus, als es 1957 erschien. Sprachverstehen bedingt, dass man sowohl das Thema als auch den Kontext versteht und nicht nur die Struktur von Sätzen. Das scheint offensichtlich zu sein, wurde aber erst in den 60er Jahren allgemein anerkannt. Ein Großteil der früheren Arbeiten im Hinblick auf die Wissensrepräsentation (wie man Wissen in einer Form darstellt, dass ein Computer damit schlussfolgern kann) war an die Sprache gebunden und wurde von Linguistikforschern erkundet, was wiederum mit Jahrzehnten von Arbeiten im Hinblick auf die philosophische Analyse der Sprache verbunden war.

1.3 Die Geschichte der künstlichen Intelligenz

Nachdem wir nun die Hintergründe beleuchtet haben, wollen wir die eigentliche Entwicklung der KI betrachten.

1.3.1 Der Reifungsprozess der künstlichen Intelligenz (1943–1955)

Die erste Arbeit, die heute allgemein als KI anerkannt wird, stammt von Warren McCulloch und Walter Pitts (1943). Sie basiert auf drei Quellen: der Kenntnis des grundlegenden Aufbaus und der Funktion der Neuronen im Gehirn, einer formalen Analyse der Aussagenlogik durch Russell und Whitehead und der Programmiertheorie von Turing. Sie schlugen ein Modell künstlicher Neuronen vor, wobei jedes Neuron durch den Zustand „an“ oder „aus“ charakterisiert ist und ein Wechsel nach „an“ erfolgt, wenn es von einer ausreichenden Anzahl benachbarter Neuronen stimuliert wird. Der Zustand eines Neurons wurde bezeichnet als „faktisch äquivalent zu einer logischen Aussage, die den entsprechenden Stimulus beschreibt“. Sie zeigten beispielsweise, dass jede berechenbare Funktion von einem Netz aus verbundenen Neuronen berechnet und dass alle logischen Verknüpfungen (UND, ODER, NICHT usw.) durch einfache Netzstrukturen implementiert werden konnten. McCulloch und Pitts behaupteten außerdem, dass geeignet definierte Netze lernfähig seien. Donald Hebb (1949) demonstrierte eine einfache Aktualisierungsregel für die Veränderung der Verbindungsstärken zwischen den Neuronen. Seine Regel, heute auch als **Hebb'sches Lernen** bezeichnet, ist bis heute ein einflussreiches Modell geblieben.

Zwei Studenten in Harvard, Marvin Minsky und Dean Edmonds, bauten 1950 den ersten neuronalen Netzcomputer. Der sogenannte SNARC arbeitete mit 3000 Vakuumröhren und einem nicht mehr benötigten Autopilot-Mechanismus aus einem B-24-Bomber, um ein Netz aus 40 Neuronen zu simulieren. Später in Princeton studierte Minsky universelle Berechnungen in neuronalen Netzen.

Der Promotionsausschuss von Minsky war skeptisch, ob diese Art von Arbeit noch als Mathematik betrachtet werden sollte, aber von Neumann soll gesagt haben: „Wenn es heute noch nicht der Fall ist, dann wird es irgendwann der Fall sein." Minsky bewies später einflussreiche Theoreme, die die Grenzen der Forschung zu neuronalen Netzen aufzeigten.

Es gab viele frühe Beispiele für Arbeiten, die sich als KI charakterisieren lassen, doch vielleicht am einflussreichsten war die Vision von Alan Turing. Bereits 1947 hielt er Vorträge zum Thema an der London Mathematical Society und formulierte eine überzeugende Agenda in seinem 1950 erschienenen Artikel „Computing Machinery and Intelligence". Dort führte er den Turing-Test, Maschinenlernen, genetische Algorithmen und Reinforcement-Lernen (verstärkendes Lernen) ein. Er schlug das Konzept des *Kindprogramms* vor und erklärte: „Anstatt zu versuchen, mit einem Programm die Intelligenz eines Erwachsenen nachzubilden, sollte man nicht eher versuchen, ein Programm zu erzeugen, das die Intelligenz eines Kindes nachbildet?"

1.3.2 Die Geburt der künstlichen Intelligenz (1956)

Princeton beherbergte noch eine weitere wichtige Persönlichkeit auf dem Gebiet der KI, John McCarthy. Nachdem er dort 1951 promoviert und noch zwei Jahre als Lehrer gearbeitet hatte, ging McCarthy nach Stanford und dann an das Dartmouth College, das zum offiziellen Geburtsort der KI wurde. McCarthy konnte Minsky, Claude Shannon und Nathaniel Rochester gewinnen, ihm zu helfen, Forscher aus ganz Amerika zusammenzubringen, die an der Automatentheorie, neuronalen Netzen und der Erforschung von Intelligenz interessiert waren. Sie organisierten im Sommer 1956 einen zwei Monate dauernden Workshop in Dartmouth. Im Aufruf dazu heißt es:[11]

> *Wir schlagen vor, im Sommer 1956 am Dartmouth College in Hanover, New Hampshire, mit einer zehnköpfigen Gruppe eine zweimonatige Untersuchung zur künstlichen Intelligenz durchzuführen. Diese Untersuchung ist auf der Grundlage der Vermutung fortzuführen, dass alle Aspekte des Lernens oder andere Merkmale der Intelligenz im Prinzip so genau beschrieben werden können, dass sich eine Maschine bauen lässt, um sie zu simulieren. Dabei soll herausgefunden werden, wie man Maschinen dazu bringt, Sprache zu verwenden, Abstraktionen und Konzepte zu bilden, Probleme zu lösen, die derzeit dem Menschen vorbehalten sind, und sich selbst zu verbessern. Wir glauben, dass sich ein signifikanter Fortschritt bei einem oder mehreren dieser Probleme erzielen lässt, wenn eine sorgfältig ausgewählte Gruppe von Wissenschaftlern für einen Sommer lang an diesem Thema zusammenarbeitet.*

11 Dies war die erste offizielle Verwendung des von McCarthy geprägten Begriffs *künstliche Intelligenz*. Vielleicht wäre „berechnende Rationalität" genauer und weniger bedrohlich gewesen, doch „KI" setzte sich durch. Zum 50. Jahrestag der Dartmouth-Konferenz merkte McCarthy an, dass er Begriffe wie „Computer" oder „berechnend" aus Rücksicht auf Norbert Wiener, der analoge kybernetische Geräte und keine digitalen Computer förderte, vermieden hatte.

Insgesamt gab es zehn Teilnehmer, unter anderem Trenchard More aus Princeton, Arthur Samuel von IBM und Ray Solomonoff und Oliver Selfridge vom MIT.

Zwei Forscher von Carnegie Tech,[12] Allen Newell und Herbert Simon, stahlen allen die Show. Während die anderen Ideen hatten und manche auch Programme für bestimmte Anwendungen, wie beispielsweise Schachspiele, hatten Newell und Simon bereits ein Schlussfolgerungsprogramm, den Logic Theorist (LT), von dem Simon behauptete: „Wir haben ein Computerprogramm erstellt, das in der Lage ist, nichtnumerisch zu denken, und damit das alte Problem Bewusstsein/Körper gelöst.“[13] Kurze Zeit nach dem Workshop war das Programm in der Lage, die meisten der Theoreme aus *Kapitel 2* von *Principa Mathematica* von Whitehead zu beweisen. Man sagt, Russell war hocherfreut, als ihm Simon zeigte, wie das Programm einen Beweis für ein Theorem erarbeitet hatte, der kürzer als der in *Principa* war. Die Herausgeber des *Journal of Symbolic Logic* waren weniger beeindruckt; sie wiesen einen Aufsatz ab, der von Newell, Simon und dem Logic Theorist geschrieben war.

Der Workshop in Dartmouth brachte keine neuen Durchbrüche, aber er sorgte dafür, dass sich die wichtigsten Personen kennenlernten. In den nächsten 20 Jahren wurde das Gebiet von diesen Leuten und ihren Studenten und Kollegen am MIT, an der CMU, in Stanford und bei IBM beherrscht.

Betrachtet man den Vorschlag des Workshops in Dartmouth (McCarthy et al., 1955), erkennt man, warum es erforderlich war, dass die künstliche Intelligenz zu einem separaten Gebiet wird. Warum konnte nicht die gesamte Arbeit, die im Gebiet der künstlichen Intelligenz stattfand, unter dem Namen der Regelungstheorie laufen oder als Operations Research oder Entscheidungstheorie, die schließlich alle ähnliche Ziele wie die künstliche Intelligenz hatten? Oder warum ist die KI nicht einfach ein Zweig der Mathematik? Die erste Antwort ist, dass die KI von Anfang an das Konzept hatte, menschliche Fähigkeiten wie Kreativität, selbstständige Verbesserung und Sprachnutzung nachzubilden. Keines der anderen Gebiete beschäftigte sich mit diesen Aspekten. Die zweite Antwort ist die Methodik. Die KI ist das einzige dieser Gebiete, das offensichtlich ein Zweig der Informatik ist (obwohl die Operations Research ähnliches Gewicht auf Computersimulationen legt), und die KI ist das einzige Gebiet, das versucht, Maschinen zu erstellen, die in komplexen, sich ändernden Umgebungen autonom arbeiten.

1.3.3 Früher Enthusiasmus, große Erwartungen (1952–1969)

In gewissen Grenzen waren die frühen Jahre der KI äußerst erfolgreich. Betrachtet man die primitiven Computer und Programmierwerkzeuge von damals und die Tatsache, dass die Computer nur wenige Jahre zuvor als Geräte betrachtet wurden, die rechnen konnten und nichts sonst, war es erstaunlich, dass ein Computer überhaupt etwas machte, was im entferntesten als intelligent zu betrachten war. Die intellektuelle Schicht glaubte im Großen und Ganzen lieber, dass „eine Maschine niemals *X* tun könne“. (In *Kapitel 26* finden Sie eine lange Liste der von Turing zusammengetragenen *X*-e.) Die Antwort der KI-Forscher war natürlich, ein *X* nach dem anderen zu

12 Heute Carnegie Mellon University (CMU).

13 Newell und Simon erfanden auch eine Sprache zur Listenverarbeitung, IPL, mit der der LT geschrieben wurde. Sie hatten keinen Compiler und übersetzten das Programm manuell in Maschinensprache. Um Fehler zu vermeiden, arbeiteten sie parallel und riefen sich Binärzahlen zu, wenn sie Anweisungen schrieben, um sicherzustellen, dass sie übereinstimmten.

demonstrieren. John McCarthy bezeichnete diesen Zeitraum auch als „Look, Ma, no hands!"-Ära („Schau Mama, ich kann freihändig Rad fahren!").

Den frühen Erfolgen von Newell und Simon folgte der General Problem Solver, GPS. Anders als der Logic Theorist war dieses Programm von Anfang an darauf ausgelegt, die menschlichen Protokolle zur Problemlösung zu imitieren. Innerhalb der begrenzten Klasse logischer Rätsel, die er bearbeiten konnte, stellte sich heraus, dass die Reihenfolge, in der das Programm Teilziele und mögliche Aktionen in Betracht zog, mit den Ansätzen eines Menschen bei denselben Problemstellungen vergleichbar war. Damit war GPS wahrscheinlich das erste Programm, das den Ansatz eines „menschlichen Denkens" verkörperte. Der Erfolg von GPS und nachfolgender Programme als Modelle der Kognition führte dazu, dass Newell und Simon ihre berühmte Hypothese des **physischen Symbolsystems** formulierten, das besagt, dass „ein physisches Symbolsystem die nötigen und hinreichenden Mittel für allgemein intelligentes Handeln besitzt". Sie meinten damit, dass jedes System (Mensch oder Maschine), das Intelligenz aufweist, arbeiten muss, indem es Datenstrukturen manipuliert, die sich aus Symbolen zusammensetzen. Wir werden später noch sehen, dass diese Hypothese aus vielen Richtungen angegriffen wurde.

Bei IBM erstellten Nathaniel Rochester und seine Kollegen einige der ersten KI-Programme. Herbert Gelernter (1959) konstruierte den Geometry Theorem Prover, der in der Lage war, Theoreme zu beweisen, die für viele Mathematikschüler zu kompliziert gewesen wären. Ab 1952 schrieb Arthur Samuel mehrere Programme für das Damespiel, die irgendwann lernten, auf hohem Amateurniveau zu spielen. Dabei widerlegte er die Idee, dass Computer nur das tun können, was man ihnen sagt: Sein Programm lernte schnell, besser zu spielen als sein Entwickler. Das Programm wurde im Februar 1956 im Fernsehen vorgestellt und machte großen Eindruck. Wie Turing hatte auch Samuel große Probleme, Rechenzeit aufzutreiben. Er arbeitete nachts und verwendete Maschinen, die sich in der Produktion von IBM noch in der Testphase befanden. In *Kapitel 5* geht es um Spiele und *Kapitel 21* erläutert die Lerntechniken, die Samuel angewandt hat.

John McCarthy ging von Dartmouth ans MIT und entwickelte dort drei wichtige Beiträge in einem bedeutsamen Jahr: 1958. Im MIT AI Lab Memo No. 1 definierte McCarthy die höhere Programmiersprache **Lisp**, die für die nächsten 30 Jahre zur vorherrschenden Programmiersprache der KI wurde. Mit Lisp hatte McCarthy das Werkzeug, das er brauchte, aber der Zugriff auf die wenigen und teuren Rechenressourcen stellte immer noch ein ernsthaftes Problem dar. Als Lösung hierfür erfanden er und andere Forscher des MIT das Time Sharing. Ebenfalls 1958 veröffentlichte McCarthy den Aufsatz *Programs with Common Sense*, in dem er den Advice Taker beschrieb, ein hypothetisches Programm, das als erstes vollständiges KI-System betrachtet werden kann. Wie der Logic Theorist und der Geometry Theorem Prover war das Programm von McCarthy darauf ausgelegt, Wissen zu nutzen, um nach Problemlösungen zu suchen. Aber anders als die anderen sollte es das allgemeine Weltwissen nutzen. Beispielsweise zeigte er, wie einige einfache Axiome es dem Programm ermöglichen, einen Plan zu entwickeln, um zum Flughafen zu fahren. Das Programm war außerdem darauf ausgelegt, im normalen Ablauf neue Axiome zu akzeptieren, sodass es auch Kompetenz auf neuen Gebieten erlangen konnte, *ohne neu programmiert zu werden*. Der Advice Taker verkörperte damit die zentralen Grundlagen der Wissensrepräsentation und des Schließens: dass es sinnvoll ist, eine formale, explizite Darstellung der Welt und ihrer Funktionsweise zu haben sowie in der Lage zu sein, diese Darstellung durch deduktive Prozesse zu verändern. Bemerkenswert ist, wie viel von diesem Artikel von 1958 heute noch relevant ist.

1958 war auch das Jahr, in dem Marvin Minsky ans MIT kam. Seine anfängliche Zusammenarbeit mit McCarthy hielt jedoch nicht lange an. McCarthy konzentrierte sich auf Darstellung und Schließen in formaler Logik, während Minsky mehr daran interessiert war, Programme zum Laufen zu bringen, und irgendwann eine antilogische Weltanschauung entwickelte. 1963 gründete McCarthy in Stanford das AI Lab. Sein Plan, mithilfe von Logik den ultimativen Advice Taker zu entwickeln, wurde 1965 von J.A. Robinsons Entdeckung der Resolutionsmethode vorangetrieben. (Dabei handelt es sich um einen vollständigen Algorithmus für den Beweis von Theoremen der Logik erster Stufe, wie in *Kapitel 9* noch beschrieben wird.) Die Arbeit in Stanford konzentrierte sich auf allgemeine Methoden für logisches Schließen. Anwendungen der Logik waren unter anderem die Frage/Antwort- und Planungssysteme von Cordell Green (Green, 1969b) sowie das Roboter-Projekt von Shakey am SRI (Stanford Research Institute). Dieses Projekt, das wir in *Kapitel 25* ausführlich vorstellen, war das erste, das eine vollständige Integration von logischem Schließen und physischem Handeln erreichen konnte.

Minsky betreute mehrere Studenten, die sich mit begrenzten Problemkreisen beschäftigten, für die scheinbar Intelligenz zur Lösung benötigt wurde. Diese begrenzten Domänen wurden auch als **Mikrowelten** bezeichnet. Das Programm SAINT von James Slagle (1963) war in der Lage, geschlossene Integrationsaufgaben zu lösen, wie sie im ersten Jahr des Grundstudiums häufig vorkommen. Das Programm ANALOGY von Tom Evans (1968) löste Probleme der geometrischen Analogie, wie man sie häufig in IQ-Tests findet. Das Programm STUDENT von Daniel Bobrow (1967) löste Algebra-Textaufgaben, wie beispielsweise die folgende: Die Anzahl der Kunden von Tom ist gleich dem Doppelten des Quadrats von 20 Prozent der Anzahl der Anzeigen, die er aufgegeben hat. Er hat 45 Anzeigen aufgegeben. Wie viele Kunden hat Tom?

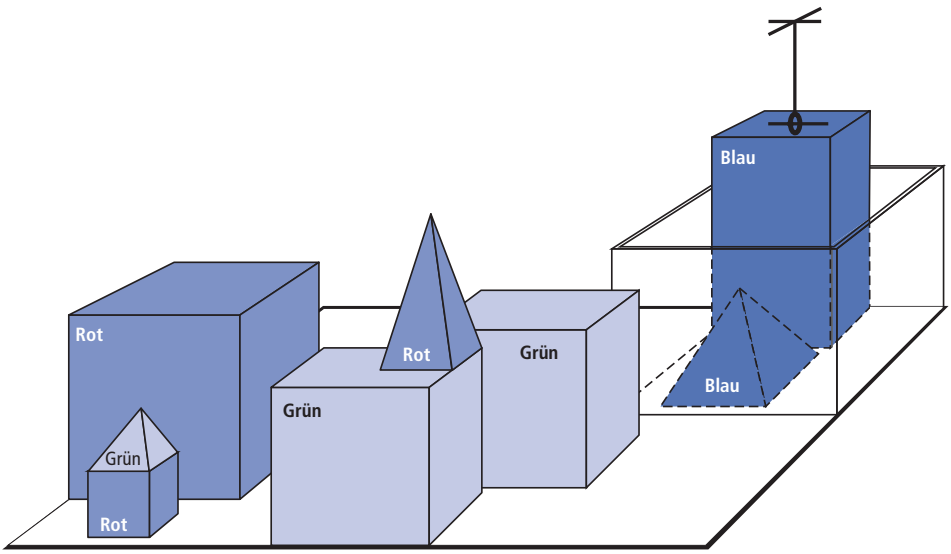


Abbildung 1.4: Eine Szene aus der Block-Welt. SHRDLU (Winograd, 1972) hat soeben den Befehl „Finde einen Block, der größer als derjenige ist, den du gerade hältst, und lege ihn in den Kasten“ verarbeitet.

Die berühmteste Mikrowelt war die Blockwelt, die aus mehreren festen Blöcken bestand, die auf einem Tisch (oder häufiger auf der Simulation eines Tisches) platziert waren, wie in ▶ Abbildung 1.4 gezeigt. Eine typische Aufgabe in dieser Welt war, die Blöcke auf eine

gewisse Weise umzuordnen, wozu eine Roboterhand verwendet wird, die jeweils einen Block aufgreifen kann. Die Blockwelt war Ausgangspunkt für das Visionsprojekt von David Huffman (1971), der Visions- und Constraint-Propagation-Arbeit von David Waltz (1975), der Lerntheorie von Patrick Winston (1970), dem Programm für das Verständnis der natürlichen Sprache von Terry Winograd (1972) und dem Planer von Scott Fahlman (1974).

Es entstanden zahlreiche frühe Arbeiten, die auf den neuronalen Netzen von McCulloch und Pitts aufbauten. Die Arbeit von Winograd und Cowan (1963) zeigte, wie eine große Anzahl an Elementen zusammen ein individuelles Konzept darstellen konnte, mit einer entsprechenden Steigerung von Robustheit und Parallelität. Erweitert wurden die Lernmethoden von Hebb durch Bernie Widrow (Widrow und Hoff, 1960; Widrow, 1962), der seine Netze als **Adalines** bezeichnete, sowie durch Frank Rosenblatt (1962) mit seinen **Perceptrons**. Das **Perceptron-Konvergenztheorem** (Block et al., 1962) besagt, dass der Lernalgorithmus die Verbindungsstärken eines Perceptrons anpassen kann, um Übereinstimmungen mit beliebigen Eingabedaten herzustellen, sofern es entsprechende Übereinstimmungen gibt. Diese Themen werden in *Kapitel 20* beschrieben.

1.3.4 Ein bisschen Realität (1966–1973)

Die KI-Forscher waren von Anfang an nicht zimperlich mit ihren Prognosen bevorstehender Erfolge. Die folgende Aussage von Herbert Simon von 1957 wird häufig zitiert:

> *Ich will Sie nicht überraschen oder schockieren – aber am einfachsten kann ich zusammenfassend berichten, dass es heute Maschinen gibt, die denken, die lernen und die kreativ sind. Darüber hinaus wächst ihre Fähigkeit, diese Dinge zu tun, zügig weiter, bis – in absehbarer Zukunft – der Bereich der Aufgabenstellungen, mit denen sie zurechtkommen, genau so groß ist wie der Bereich, den der menschliche Verstand bewältigen kann.*

Begriffe wie „absehbare Zukunft“ können unterschiedlich interpretiert werden, aber Simon traf eine noch genauere Vorhersage: dass innerhalb von zehn Jahren ein Computer Schachweltmeister sein und dass ein wichtiges mathematisches Theorem von einer Maschine bewiesen werden würde. Diese Vorhersagen verwirklichten sich (beziehungsweise fast) nicht in zehn, sondern in 40 Jahren. Die übersteigerte Zuversicht von Simon gründete auf der vielversprechenden Leistung früher KI-Systeme im Hinblick auf einfache Beispiele. In fast allen Fällen zeigte sich jedoch, dass diese früheren Systeme scheiterten, wenn man versuchte, die Probleme zu erweitern oder schwierigere Aufgabenstellungen zu wählen.

Die erste Schwierigkeit entstand, weil die meisten frühen Programme nichts von ihrem Aufgabenbereich wussten; ihre Erfolge beruhten auf einfachen syntaktischen Manipulationen. Eine typische Geschichte findet man in frühen Bemühungen maschineller Übersetzung, die 1957 in der Aufregung über den Start von Sputnik vom U.S. National Research Council unternommen wurden, um die Übersetzung wissenschaftlicher russischer Arbeiten zu beschleunigen. Anfänglich glaubte man, einfache syntaktische Übertragungen, die auf der russischen und der englischen Grammatik basierten, und der Austausch von Wörtern unter Verwendung eines elektronischen Wörterbuches seien ausreichend, um die genaue Bedeutung der Sätze zu bewahren. Tatsache ist aber, dass für eine Übersetzung ein Hintergrundwissen erforderlich ist, um Mehrdeutigkeiten zu vermeiden und den Inhalt der Sätze korrekt wiedergeben zu können. Die berühmte

Rückübersetzung von „Der Geist ist willig, aber das Fleisch ist schwach“ zu „Der Wodka ist gut, aber das Fleisch ist verfault“ zeigt, welchen Schwierigkeiten man gegenüberstand. 1966 wurde in einem Bericht von einem Gutachterausschuss festgestellt, dass „es keine maschinelle Übersetzung allgemeiner wissenschaftlicher Texte gibt und auch keine unmittelbare Aussicht darauf besteht“. Alle Förderung akademischer Übersetzungsprojekte durch die US-Regierung wurde abgebrochen. Heute ist die maschinelle Übersetzung ein nicht perfektes, aber weit verbreitetes Werkzeug für technische, wirtschaftliche, Regierungs- und Internetdokumente.

Die zweite Schwierigkeit war die Komplexität vieler Probleme, die die KI lösen wollte. Die meisten der früheren KI-Programme lösten Probleme, indem sie verschiedene Schrittkombinationen ausprobierten, bis die Lösung gefunden war. Diese Strategie funktionierte anfänglich, weil Mikrowelten nur sehr wenige Objekte enthielten und damit sehr wenige mögliche Aktionen und sehr kurze Lösungsfolgen. Bevor die Komplexitätstheorie entwickelt wurde, war man allgemein der Auffassung, dass eine „Skalierung“ auf größere Probleme einfach eine Frage schnellerer Hardware und größerer Speicher war. Der Optimismus, der beispielsweise die Entwicklung des Resolutionstheorem-Beweises begleitete, wurde schnell gedämpft, als die Forscher keine Theoreme mit mehr als etwa ein paar Dutzend Fakten beweisen konnten. *Die Tatsache, dass ein Programm im Prinzip eine Lösung finden kann, bedeutet nicht, dass das Programm irgendwelche Mechanismen enthält, die man benötigt, um diese Lösung auch in der Praxis zu finden.*

Die Illusion unbeschränkter Rechenleistung war nicht auf Problemlösungsprogramme beschränkt. Experimente in der **Maschinenevolution** (heute auch als **genetische Algorithmen** bezeichnet) (Friedberg, 1958; Friedberg et al., 1959) basierten auf dem zweifellos korrekten Glauben, dass man durch eine geeignete Folge kleiner Mutationen im Maschinencode ein effizientes Programm für jede konkrete Aufgabe erzeugen kann. Die Idee war dann, beliebige Mutationen auszuprobieren, wobei ein Auswahlprozess die Mutationen beibehielt, die sinnvoll erschienen. Trotz Tausender Stunden CPU-Zeit konnte fast kein Fortschritt nachgewiesen werden. Moderne genetische Algorithmen verwenden bessere Repräsentationen und haben mehr Erfolg gezeigt.

Das mangelnde Verständnis für die „kombinatorische Explosion“ war einer der wichtigsten Kritikpunkte an der KI, wie im Lighthill-Bericht beschrieben (Lighthill, 1973), der die Grundlage für die Entscheidung der britischen Regierung darstellte, die Unterstützung der KI-Forschung in allen außer in zwei Universitäten einzustellen. (Die mündliche Überlieferung zeigt ein etwas anderes und farbigeres Bild, mit politischen Ambitionen und persönlichen Animositäten, die hier nicht beschrieben werden können.)

Eine dritte Schwierigkeit entstand durch die allgemeinen Einschränkungen der grundlegenden Strukturen, die verwendet wurden, um intelligentes Verhalten zu erzeugen. Beispielsweise bewies das Buch *Perceptron* (1969) von Minsky und Papert, dass Perceptronen (eine einfache Form eines neuronalen Netzes) nachweislich alles lernen konnten, was sie auch darstellen konnten, aber sie konnten nur sehr wenig darstellen. Insbesondere konnte ein Perceptron mit zwei Eingängen (eingeschränkt, um einfacher zu sein als die Form, die Rosenblatt ursprünglich untersucht hat) nicht trainiert werden zu erkennen, wann sich seine beiden Eingänge unterschieden. Obwohl ihre Ergebnisse nicht für komplexere, mehrschichtige Netze galten, wurde die Forschung neuronaler Netze sehr schnell fast überhaupt nicht mehr unterstützt. Ironischerweise wurden gerade die neuen Backpropagation-Lernalgorithmen für mehrschichtige Netze, die Ende der 1980er Jahre eine enorme Wiederbelebung der Forschung auf dem Gebiet neuronaler Netze bewirkten, tatsächlich bereits 1969 entdeckt (Bryson und Ho, 1969).

1.3.5 Wissensbasierte Systeme: der Schlüssel zum Erfolg? (1969–1979)

Das in den ersten zehn Jahren der KI-Forschung entstandene Bild des Problemlösens war ein allgemeiner Suchmechanismus, der versuchte, elementare Schritte zu verknüpfen, um vollständige Lösungen zu finden. Solche Ansätze werden auch als **schwache Methoden** bezeichnet, weil sie zwar allgemein sind, aber nicht auf größere oder schwierigere Probleminstanzen skaliert werden können. Die Alternative zu schwachen Methoden ist die Verwendung eines leistungsfähigeren, bereichsspezifischeren Wissens, das logische Ableitungsschritte zulässt und einfache typische Fälle in begrenzten Erfahrungsbereichen verarbeiten kann. Man könnte sagen, dass man die Antwort fast kennen muss, wenn ein schwieriges Problem zu lösen ist.

Das Programm DENDRAL (Buchanan et al., 1969) war ein frühes Beispiel für diesen Ansatz. Es wurde in Stanford entwickelt, wo Ed Feigenbaum (ein früherer Student von Herbert Simon), Bruce Buchanan (ein Philosoph, der zum Informatiker geworden war) und Joshua Lederberg (ein mit dem Nobelpreis ausgezeichneter Genetiker) sich zusammenschlossen, um das Problem der Analyse molekularer Strukturen aus der Information eines Massenspektrometers zu lösen. Die Eingabe für das Programm besteht aus der grundlegenden Formel des Moleküls (z.B. $C_6H_{13}NO_2$) und dem Massenspektrum, das die Massen der verschiedenen Bestandteile des Moleküls angibt, die erzeugt werden, wenn es von einem Elektronenstrahl beschossen wird. Beispielsweise könnte das Massenspektrum bei $m = 15$ eine Spitze aufweisen, was der Masse einer Methylgruppe (CH_3) entspricht.

Die naive Version des Programms erzeugte alle möglichen Strukturen, die mit der Formel konsistent waren, sagte dann voraus, welches Massenspektrum für sie beobachtet werden konnte, und verglich dies mit dem tatsächlichen Spektrum. Wie man erwarten konnte, ist dies selbst für Moleküle mittlerer Größe schwer zu handhaben. Die DENDRAL-Forscher konsultierten analytische Chemiker und stellten fest, dass diese nach bekannten Mustern von Spitzen in dem Spektrum suchten, die auf gemeinsame Unterstrukturen im Molekül hinwiesen. Beispielsweise wird die folgende Regel für die Erkennung einer Keton-Untergruppe (C=O) (die 28 wiegt) verwendet:

```
falls es zwei Spitzen bei x1 und x2 gibt, sodass gilt
(a) x1 + x2 = M + 28 (M ist die Masse des gesamten Moleküls),
(b) x1 – 28 ist eine hohe Spitze,
(c) x2 – 28 ist eine hohe Spitze,
(d) mindestens eines von x1 und x2 ist hoch,
dann handelt es sich um eine Keton-Untergruppe.
```

Die Erkenntnis, dass das Molekül eine bestimmte Unterstruktur enthält, reduziert die Anzahl der möglichen Kandidaten wesentlich. DENDRAL war leistungsfähig, weil ...

> *Das gesamte relevante theoretische Wissen für die Lösung dieser Probleme wurde aus seiner allgemeinen Form in den [Spektrum Vorhersage Komponente] („ersten Prinzipien“) auf effiziente spezielle Formen („Kochrezepte“) abgebildet. (Feigenbaum et al., 1971)*

Die Bedeutung von DENDRAL war, dass es sich dabei um das erste erfolgreiche *wissensintensive* System handelte: Seine Expertise beruhte auf einer großen Anzahl spezieller Regeln. Spätere Systeme beinhalteten auch das Hauptthema des Ansatzes von McCarthys Advice Taker – die klare Trennung des Wissens (in Form von Regeln) von der Schlussfolgerungskomponente.

Mit dieser Erfahrung im Hinterkopf begannen Feigenbaum und andere in Stanford, am HPP (Heuristic Programming Project) zu arbeiten, um zu erforschen, in welchem Ausmaß die neue Methodik von **Expertensystemen** auf andere Bereiche menschlicher Erfahrung angewendet werden konnte. Die nächste große Anstrengung war auf dem Gebiet der medizinischen Diagnostik zu verzeichnen. Feigenbaum, Buchanan und Dr. Edward Shortliffe entwickelten MYCIN für die Diagnose von Blutinfektionen. Mit etwa 450 Regeln war MYCIN in der Lage, mit einigen Experten gleichzuziehen – und schnitt wesentlich besser als Jungmediziner ab. Außerdem wies es zwei wichtige Unterschiede zu DENDRAL auf. Erstens gab es im Unterschied zu DENDRAL-Regeln kein allgemeines theoretisches Modell, aus dem die MYCIN-Regeln abgeleitet werden konnten. Sie mussten aus umfassenden Befragungen von Experten erworben werden, die sie sich wiederum aus Lehrbüchern, von anderen Experten und aus direkten Fallstudien angeeignet hatten. Zweitens mussten die Regeln die Unsicherheit widerspiegeln, die medizinischem Wissen anhaftet. MYCIN beinhaltete eine Unsicherheitsberechnung mit sogenannten **Sicherheitsfaktoren** (siehe *Kapitel 14*), wovon man (damals) den Eindruck hatte, dass es die Art, wie Ärzte den Einfluss von Anzeichen auf die Diagnose einschätzten, gut widerspiegelte.

Die Bedeutung von Fachwissen wurde auch auf dem Gebiet des Verstehens natürlicher Sprache offensichtlich. Obwohl das SHRDLU-System von Winograd für das Verstehen natürlicher Sprache so großes Aufsehen erregt hatte, verursachte seine Abhängigkeit von einer syntaktischen Analyse zum Teil dieselben Probleme, wie sie in früheren Arbeiten zum maschinellen Übersetzen aufgetreten waren. Es konnte Mehrdeutigkeiten überwinden und Pronomenbezüge verstehen, aber hauptsächlich deshalb, weil es speziell für einen Bereich ausgelegt worden war – die Blockwelt. Mehrere Forscher, unter anderem Eugene Charniak, wie Winograd ebenfalls Doktorand am MIT, wiesen darauf hin, dass robuste Spracherkennung ein allgemeines Wissen über die Welt sowie eine allgemeine Methode für die Anwendung dieses Wissens benötigen würde.

In Yale konzentrierte sich der vom Linguisten zum KI-Forscher gewandelte Roger Schank auf diesen Aspekt und behauptete: „Es gibt keine Syntax“, was sehr viele Linguisten aufbrachte, aber eine hilfreiche Diskussion anregte. Schank und seine Studenten erstellten mehrere Programme (Schank und Abelson, 1977; Wilensky, 1978; Schank und Riesbeck, 1981; Dyer, 1983), die alle die Aufgabe hatten, natürliche Sprache zu verstehen. Die Betonung lag dabei jedoch nicht auf der Sprache *per se*, sondern mehr auf den Problemen der Darstellung und des Schließens mit dem Wissen, das für die Spracherkennung erforderlich war. Die Probleme waren unter anderem die Darstellung stereotyper Situationen (Cullingford, 1981), die Beschreibung der Organisation des menschlichen Gedächtnisses (Rieger, 1976; Kolodner, 1983) sowie das Verstehen von Plänen und Zielen (Wilensky, 1983).

Die allgemeine Zunahme von Anwendungen auf realistische Probleme bewirkte einen gleichzeitigen Anstieg des Bedarfs nach funktionierenden Schemata zur Wissensrepräsentation. Es wurde eine große Zahl verschiedener Darstellungs- und Schlussfolgerungssprachen entwickelt. Einige davon basierten auf Logik – beispielsweise wurde die Sprache PROLOG in Europa beliebt und die PLANNER-Familie in den USA. Andere, die ebenfalls Minskys Konzept der **Frames** (**Rahmen**) verfolgten (1975), wendeten einen strukturierteren Ansatz an, indem sie Tatsachen über bestimmte Objekt- und Ereignistypen zusammensetzten und die Typen in einer großen taxonomischen Hierarchie anordneten, vergleichbar mit einer biologischen Klassifizierung.

1.3.6 KI wird zu einem Industriezweig (1980 bis heute)

Das erste erfolgreiche kommerzielle Expertensystem, R1, wurde zunächst bei Digital Equipment Corporation (McDermott, 1982) eingesetzt. Das Programm half, Aufträge für neue Computersysteme zu konfigurieren; 1986 sparte es etwa 40 Millionen Dollar jährlich für das Unternehmen ein. 1988 setzte die AI-Gruppe von DEC 40 Expertensysteme ein und es wurden ständig mehr. Du Pont hatte 100 Expertensysteme in Verwendung und 500 in Entwicklung, was schätzungsweise zehn Millionen Dollar pro Jahr sparte. Fast jedes größere Unternehmen in den USA hatte eine eigene KI-Gruppe und verwendete oder erforschte Expertensysteme.

1981 kündigte Japan das Projekt „5. Generation" an – einen Zehnjahresplan für den Aufbau intelligenter Computer unter PROLOG. Als Reaktion darauf bildeten die USA die MCC (Microelectronics and Computer Technology Corporation) als Forschungskonsortium für die Sicherstellung nationaler Wettbewerbsfähigkeit. In beiden Fällen war die KI Teil eines umfassenderen Engagements, unter anderem beim Chip-Design und bei Forschungen zur Benutzerschnittstelle. In Großbritannien sorgte der Alvey-Bericht für die Wiederaufnahme der Unterstützung, die durch den Lighthill-Bericht aufgehoben worden war.[14]

Insgesamt gab es in der KI-Industrie einen Anstieg von einigen Millionen Dollar im Jahr 1980 auf mehrere Milliarden Dollar im Jahr 1988. Hunderte von Firmen erstellten Expertensysteme, Erkennungssysteme und Roboter sowie Soft- und Hardware, die für diese Zwecke spezialisiert war. Bald danach begann eine Periode, die auch als „KI-Winter" bezeichnet wird, in der viele Unternehmen verschwanden, weil sie ihre außergewöhnlichen Versprechungen nicht halten konnten.[15]

1.3.7 Die Rückkehr der neuronalen Netze (1986 bis heute)

Mitte der 1980er kamen mindestens vier verschiedene Gruppen wieder auf den **Backpropagation**-Lernalgorithmus zurück, der 1969 von Bryson und Ho zum ersten Mal angesprochen worden war. Der Algorithmus wurde auf viele Lernprobleme in der Informatik und Psychologie angewendet und die allgemeine Veröffentlichung der Ergebnisse im Sammelband *Parallel Distributed Processing* (Rumelhart und McClelland, 1986) erregte großes Aufsehen.

Die sogenannten **konnektionistischen** Modelle intelligenter Systeme wurden von einigen Gruppen als direkte Konkurrenz sowohl für die symbolischen Modelle von Newell und Simon als auch für den logistischen Ansatz von McCarthy und anderen betrachtet (Smolensky, 1988). Es scheint offensichtlich, dass Menschen auf irgendeiner Ebene Symbole manipulieren – tatsächlich behauptet das Buch von Terrence Deacon, *The symbolic Species* (1997), dass dies das definierende Merkmal des Menschen sei, aber die glühendsten Konnektionisten fragten, ob die Symbolmanipulation eine wirklich erklärende Rolle bei detaillierten Modellen der Kognition einnehme. Diese Frage bleibt unbeantwortet, aber es herrscht gegenwärtig die Ansicht, dass sich konnektionistische und symbolische Ansätze

14 In Deutschland wurde das Deutsche Forschungszentrum für Künstliche Intelligenz gegründet, das bis heute besteht und das größte Institut für KI weltweit ist.

15 Um sich nicht in Verlegenheit zu bringen, wurde ein neuer Bereich namens IKBS (Intelligent Knowledge-Based Systems, intelligente wissensbasierte Systeme) eingeführt, weil die künstliche Intelligenz offiziell gestoppt worden war.

ergänzen und nicht miteinander konkurrieren. Wie es bei der Trennung von KI und kognitiver Wissenschaft eingetreten war, hat sich die moderne Forschung der neuronalen Netze in zwei Gebiete aufgeteilt – in einen Zweig, der sich mit dem Erstellen effektiver Netzarchitekturen und Algorithmen sowie dem Verständnis ihrer mathematischen Eigenschaften befasste, und einen anderen, der mit der sorgfältigen Modellierung der empirischen Eigenschaften von realen Neuronen und Ensembles von Neuronen zu tun hatte.

1.3.8 KI wird zu einer Wissenschaft (1987 bis heute)

In den letzten Jahren konnten wir eine Revolution sowohl in Bezug auf den Inhalt als auch auf die Methodik der Arbeit in der künstlichen Intelligenz beobachten.[16] Heute ist es üblicher, auf existierenden Theorien aufzubauen, statt völlig neue vorzuschlagen, Behauptungen auf strengen Theoremen oder strengen experimentellen Beweisen basieren zu lassen statt auf Intuition und die Bedeutung anhand von realistischen Anwendungen zu zeigen statt anhand von Spielzeugbeispielen.

KI wurde zum Teil als Rebellion gegen die Beschränkungen existierender Gebiete gegründet, wie beispielsweise der Regelungstheorie und der Statistik, beinhaltet jedoch jetzt auch diese Gebiete. David McAllester (1998) hat es so ausgedrückt:

> *In der früheren Zeit der KI schien es wahrscheinlich, dass neue Formen symbolischer Berechnung, zum Beispiel Frames und semantische Netze, einen Großteil der klassischen Theorie überflüssig machen würden. Das führte zu einer Art von Isolationismus, wobei die KI größtenteils von der restlichen Informatik abgetrennt wurde. Dieser Isolationismus wird momentan aufgegeben. Man hat erkannt, dass das maschinelle Lernen nicht von der Informationstheorie isoliert werden soll, dass unbestimmtes Schließen nicht von der stochastischen Modellierung isoliert sein soll, dass Suche nicht von klassischer Optimierung und Steuerung isoliert sein soll und dass das automatische Schließen nicht von formalen Methoden und statischer Analyse isoliert sein soll.*

In Bezug auf die Methodik ist die KI endlich fest unter die wissenschaftliche Methode gerückt. Hypothesen sind strengen empirischen Experimenten zu unterziehen, um akzeptiert zu werden, und die Ergebnisse müssen statistisch auf ihre Bedeutung hin analysiert werden (Cohen, 1995). Mithilfe gemeinsamer Bibliotheken von Testdaten und Code ist es jetzt möglich, Experimente zu replizieren.

Ein Beispiel hierfür ist der Bereich der Spracherkennung. In den 1970er Jahren wurden eine Vielzahl verschiedener Architekturen und Ansätze ausprobiert. Viele davon waren eher *ad hoc* und instabil und wurden nur für ein paar wenige, speziell ausgewählte Beispiele demonstriert. In den letzten Jahren beherrschten Ansätze, die auf **Hidden-Markov-Modellen** (HMMs) basieren, das Gebiet. Zwei Aspekte von HMMs sind relevant. Erstens basieren sie auf einer strengen mathematischen Theorie. Das hat es den Sprachforschern erlaubt, auf mehreren Jahrzehnten mathematischer Ergebnisse

16 Einige charakterisierten diese Änderung als Sieg der Saubermänner – die denken, KI-Theorien sollten auf mathematischer Strenge gründen – über die Schmuddler – die stattdessen lieber viele Ideen ausprobieren, einige Programme schreiben und dann abschätzen wollen, was scheinbar funktioniert. Beide Ansätze sind wichtig. Eine Verschiebung in Richtung der Sauberkeit bedeutet, dass das Gebiet eine gewisse Stabilität und Reife erreicht hat. Ob diese Stabilität durch eine neue Schmuddelidee gestört wird, ist eine andere Frage.

aufzubauen, die auf anderen Gebieten entwickelt worden waren. Zweitens werden sie durch einen Prozess des Trainings mit einer große Menge realer Sprachdaten erzeugt. Damit wird sichergestellt, dass das Verhalten robust ist, und in strengen Blindtests haben die HMMs ihre Trefferquoten ständig verbessert. Die Sprachtechnologie und der verwandte Bereich der Handschrifterkennung vollziehen bereits den Übergang zu industriellen und privaten Anwendungen auf breiter Front. Dabei gibt es keinen wissenschaftlichen Anspruch, dass Menschen HMMs zur Spracherkennung einsetzen; vielmehr bieten HMMs ein mathematisches Gerüst für das Verstehen des Problems und unterstützen den technischen Anspruch, dass sie in der Praxis gut funktionieren.

Maschinelle Übersetzung absolvierte den gleichen Werdegang wie Spracherkennung. In den 1950er Jahren begeisterte man sich zunächst für ein Konzept, das auf Wortsequenzen beruhte, wobei die Modelle entsprechend den Prinzipien der Informationstheorie lernten. Dieses Konzept geriet in den 1960er Jahren in Vergessenheit, rückte aber in den späten 1990er Jahren wieder in den Vordergrund und dominiert jetzt das Gebiet.

Auch die neuronalen Netze passen in diesen Trend. Viele Arbeiten zu neuronalen Netzen in den 1980er Jahren versuchten herauszufinden, was möglich war, und zu lernen, wie sich neuronale Netze von „traditionellen“ Techniken unterscheiden. Mit einer verbesserten Methodik und theoretischen Grundgerüsten ist das Gebiet bei einem Verständnis angelangt, wo neuronale Netze jetzt mit entsprechenden Techniken aus der Statistik, Mustererkennung und dem Maschinenlernen verglichen werden können, und für jede Anwendung kann die aussichtsreichste Technik ausgewählt werden. Als Ergebnis dieser Entwicklungen hat die sogenannte **Data-Mining**-Technologie einen starken neuen Industriezweig hervorgebracht.

Das Werk *Probabilistic Reasoning in Intelligent Systems* von Judea Pearl (1988) führte zu einer neuen Akzeptanz der Wahrscheinlichkeits- und Entscheidungstheorie in der KI, nämlich in Folge eines wiederbelebten Interesses für den Artikel „In Defense of Probability“ von Peter Cheeseman (1985). Der Formalismus des **Bayesschen Netzes** wurde eingeführt, um eine effiziente Darstellung von und strenges Schließen mit unsicherem Wissen zu erlauben. Dieser Ansatz löst viele Probleme wahrscheinlichkeitstheoretischer Schlussfolgerungssysteme der 1960er und 1970er Jahre; jetzt ist er vorherrschend in der KI-Forschung zu unsicherem Schließen und Expertensystemen. Der Ansatz erlaubt es, aus Erfahrungen zu lernen, und er kombiniert das Beste aus der klassischen KI und neuronalen Netzen. Die Arbeiten von Judea Pearl (1982a) und von Eric Horvitz und David Heckerman (Horvitz und Heckerman, 1986; Horvitz et al., 1986) propagierten das Konzept normativer Expertensysteme: Expertensysteme, die rational gemäß der Gesetze der Entscheidungstheorie agieren und nicht versuchen, die Gedankenschritte menschlicher Experten zu imitieren. Das Betriebssystem Windows™ beinhaltet mehrere normative Diagnose-Expertensysteme für die Fehlerkorrektur. Weitere Informationen zu diesem Thema finden Sie in den *Kapiteln 13* und *16*.

Ähnliche sanfte Revolutionen sind im Bereich der Robotik, Computervision und Wissensrepräsentation aufgetreten. Ein besseres Verständnis für die Probleme und ihre Komplexitätseigenschaften hat zusammen mit gewachsener mathematischer Raffinesse zu funktionierenden Forschungsplänen und robusten Methoden geführt. Obwohl Gebiete wie Vision und Robotik durch den höheren Grad an Formalisierung und Spezialisierung in den 1990er Jahren etwas von der „Mainstream-KI“ isoliert wurden, hat sich dieser Trend in den letzten Jahren umgekehrt, da sich Werkzeuge insbesondere vom maschinel-

len Lernen für viele Probleme als effektiv erwiesen haben. Der Prozess der Reintegration lässt bereits signifikante Vorteile erkennen.

1.3.9 Das Entstehen intelligenter Agenten (1995 bis heute)

Möglicherweise ermutigt durch den Fortschritt beim Lösen von Teilproblemen der KI begannen die Forscher auch, das Problem des „vollständigen Agenten" erneut zu betrachten. Die Arbeit von Allen Newell, John Laird und Paul Rosenbloom an SOAR (Newell, 1990; Laird et al., 1987) ist das bekannteste Beispiel einer vollständigen Agentenarchitektur. Eine der wichtigsten Umgebungen für intelligente Agenten ist das Internet. KI-Systeme sind in webbasierten Anwendungen so üblich geworden, dass das Suffix „-bot" in die Alltagssprache übergegangen ist. Darüber hinaus liegen vielen Internetwerkzeugen wie beispielsweise Suchmaschinen, Empfehlungssystemen und Website-Aggregatoren KI-Technologien zugrunde.

Eine Folge des Versuches, vollständige Agenten zu erstellen, ist die Erkenntnis, dass die zuvor isolierten Teilbereiche der KI möglicherweise neu organisiert werden müssen, wenn ihre Ergebnisse verknüpft werden sollen. Insbesondere wird heute allgemein anerkannt, dass sensorische Systeme (Vision, Sonar, Spracherkennung usw.) keine perfekt zuverlässigen Informationen über die Umgebung liefern können. Aus diesem Grund müssen Schluss- und Planungssysteme in der Lage sein, Unsicherheiten zu kompensieren. Eine zweite große Folge der Agentenperspektive ist, dass KI in viel engeren Kontakt zu anderen Gebieten gelangt ist, wie beispielsweise Regelungstheorie und Wirtschaft, die sich ebenfalls mit Agenten beschäftigen. Jüngste Fortschritte bei der Regelung von Roboterfahrzeugen leiten sich aus einer Mischung von Konzeptionen ab, die von besseren Sensoren, über regelungstheoretische Integration von Fühlern bis zu Lokalisierung und Zuordnung sowie einem auf höherer Ebene angesiedelten Planungsniveau reichen.

Trotz dieser Erfolge haben einige einflussreiche Gründer der künstlichen Intelligenz, einschließlich John McCarthy (2007), Marvin Minsky (2007), Nils Nilsson (1995, 2005) und Patrick Winston (Beal und Winston, 2009), ihre Unzufriedenheit mit dem Fortschritt der KI ausgedrückt. Ihrer Meinung nach sollte KI weniger darauf aus sein, immer bessere Versionen von Anwendungen zu schaffen, die bei einer bestimmten Aufgabe wie zum Beispiel Führen eines Fahrzeuges, Schachspielen oder Spracherkennung gut funktionieren. Stattdessen glauben sie, dass die KI zu ihren Wurzeln zurückkehren sollte, nämlich sich – mit den Worten von Simon – auf „Maschinen, die denken, die lernen und die kreativ sein können" zu konzentrieren. Dieses Bestreben bezeichnen sie als **KI auf menschlicher Ebene** (Human-Level AI, HLAI); ihr erstes Symposium fand 2004 statt (Minsky et al., 2004). Für die Realisierung dieses Ansatzes sind sehr große Wissensdatenbanken erforderlich. Hendler et al. (1995) diskutieren, wie diese Wissensdatenbanken entstehen können.

Eine verwandte Idee ist das Teilgebiet der **Artificial General Intelligence** oder AGI (Goertzel und Pennachin, 2007), deren erste Konferenz das *Journal of Artificial General Intelligence* im Jahr 2008 abgehalten und organisiert hat. AGI sucht nach einem universellen Algorithmus für das Lernen und Handeln in einer beliebigen Umgebung. Die Wurzeln dieses Konzepts liegen in der Arbeit von Ray Solomonoff (1964), einem der Teilnehmer der ursprünglichen Dartmouth-Konferenz von 1956. Außerdem ist es ein Anliegen zu garantieren, dass es sich um wirklich **freundliche KI** (**Friendly AI**) handelt (Yudkowsky, 2008; Omohundro, 2008), worauf wir in *Kapitel 26* zurückkommen.

1.3.10 Die Verfügbarkeit sehr großer Datenmengen (2001 bis heute)

In der 60-jährigen Geschichte der Informatik lag die Betonung auf dem Algorithmus als Hauptuntersuchungsobjekt. Neuere Arbeiten in der KI legen aber nahe, dass es für viele Probleme sinnvoller ist, sich mehr mit den Daten zu befassen und weniger wählerisch zu sein, welcher Algorithmus anzuwenden ist. Das gilt aufgrund der zunehmenden Verfügbarkeit sehr großer Datenquellen: zum Beispiel Billionen englischer Wörter und Bilder aus dem Web (Kilgarriff und Grefenstette, 2006) oder Milliarden Basenpaare genomischer Sequenzen (Collins et al., 2003).

Ein einflussreicher Artikel in dieser Richtung ist die Arbeit von Yarowsky (1995) zur Mehrdeutigkeit des Wortsinnes: Bezieht sich das Wort „plant“ in einem Satz auf die Flora (flora) oder eine Fabrik (factory)? Vorherige Ansätze für das Problem hatten sich auf manuell kommentierte Beispiele in Verbindung mit maschinellen Lernalgorithmen gestützt. Yarowsky zeigte, dass sich die Aufgabe mit einer Genauigkeit von über 96% lösen ließ, ohne dass überhaupt irgendwelche kommentierten Beispiele erforderlich sind. Stattdessen kann man in einer sehr großen Sammlung von unkommentiertem Text und lediglich mit den Wörterbuchdefinitionen der beiden Lesarten – „works, industrial plant“ und „flora, plant life“ – einige Beispiele in der Sammlung kommentieren und von da an per **Bootstrapping** neue Muster lernen lassen, die dabei helfen, neue Beispiele zu kommentieren. Banko und Brill (2001) zeigen, dass derartige Techniken noch besser funktionieren, wenn die Menge des verfügbaren Textes von einer Million Wörter auf eine Milliarde ansteigt und dass die Leistungsverbesserung durch die Verwendung von weiteren Daten jegliche Unterschiede in der Algorithmenauswahl übersteigt. Ein mittelmäßiger Algorithmus mit 100 Millionen Wörtern von nicht kommentierten Trainingsdaten übertrifft den besten bekannten Algorithmus mit 1 Million Wörtern.

Als weiteres Beispiel diskutieren Hays und Efros (2007) das Problem, Leerstellen in einer Fotografie zu füllen. Angenommen, Sie möchten mit Photoshop einen Exfreund aus einem Gruppenfoto ausmaskieren und müssen nun den maskierten Bereich mit etwas ausfüllen, das dem Hintergrund entspricht. Hays und Efros definierten einen Algorithmus, der eine Sammlung von Fotos nach etwas Passendem durchsucht. Dabei zeigte sich, dass ihr Algorithmus mit einer Sammlung von nur zehntausend Fotos recht schlecht abschnitt, jedoch eine ausgezeichnete Leistung zeigte, als sie die Sammlung auf zwei Millionen Fotos aufstockten.

Derartige Arbeiten deuten darauf hin, dass der „Wissensengpass“ in KI – das Problem, wie das gesamte Wissen, das ein System benötigt, auszudrücken ist – in vielen Anwendungen durch Lernmethoden statt durch manuell kodierte Wissensverarbeitung gelöst werden kann, sofern den Lernalgorithmen genügend Daten zur Verfügung stehen (Halevy et al., 2009). Journalisten beobachteten die sprunghafte Zunahme von neuen Anwendungen und schrieben, dass der „KI-Winter“ einen neuen Frühling bescheren kann (Havenstein, 2005). Wie Kurzweil (2005) schreibt, sind „heute viele Tausende von KI-Anwendungen tief in die Infrastruktur jedes Industriezweigs eingebettet“.

1.4 Die aktuelle Situation

Was kann die KI heute? Es ist schwierig, eine präzise Antwort auf diese Frage zu geben, weil es in so vielen Teilbereichen Aktivitäten gibt. Hier stellen wir ein paar Beispielanwendungen vor; andere lernen Sie im Laufe des Buches kennen.

Roboterfahrzeuge: Ein führerloses Roboterauto namens STANLEY „raste“ mit 35 km/h durch das raue Gelände der Mojave-Wüste. Es bewältigte als Erstes den 132 Meilen (212 km) langen Kurs und gewann damit die DARPA Grand Challenge 2005. STANLEY ist ein VW-Touareg, ausgerüstet mit Kameras, Radar und Laser-Entfernungsmesser, um die Umgebung abzutasten, sowie Onboard-Software, um Lenkung, Bremsen und Beschleunigung zu steuern (Thrun, 2006). Im folgenden Jahr gewann BOSS von CMU die Urban Challenge, wobei sich das Fahrzeug sicher durch den Verkehr auf den Straßen einer geschlossenen Air-Force-Basis bewegte, dabei die Verkehrsregeln beachtete sowie Zusammenstöße mit Fußgängern und anderen Fahrzeugen vermied.

Spracherkennung: Ein Reisender, der bei einer Fluggesellschaft anruft, um einen Flug zu buchen, kann während der gesamten Konversation durch eine automatisierte Spracherkennung und ein Dialogverwaltungssystem geführt werden.

Autonomes Planen und Zeitplanen: 100 Millionen Meilen von der Erde entfernt wurde das Remote-Agent-Programm der NASA zum ersten automatischen Planungsprogramm an Bord, das die Zeitplanung von Operationen für ein Raumschiff kontrollierte (Jonsson et al., 2000). *Remote Agent* erzeugte Pläne anhand von komplexen Zielen, die von der Erde aus spezifiziert wurden, und es überwachte die Operationen des Raumschiffes während der Ausführung der Pläne – wobei einige Probleme beim Auftreten erkannt, diagnostiziert und behoben wurden. Das Nachfolgeprogramm MAPGEN (Al-Chang et al., 2004) plant die täglichen Operationen für die Mars Exploration Rover der NASA und MEXAR2 (Cesta et al., 2007) und übernahm die Missionsplanung – sowohl die logistische als auch die wissenschaftliche Planung – für die Mars-Express-Mission 2008 der ESA (European Space Agency).

Spiele: DEEP BLUE von IBM war das erste Computerprogramm, das einen Weltmeister, Garry Kasparov, in einem Schachspiel besiegte – mit einem Stand von 3,5 zu 2,5 in einem öffentlichen Spiel (Goodman und Keene, 1997). Kasparov beschrieb sein Gefühl, dass ihm am Brett eine „neuartige Intelligenz“ gegenübersitze. Das Newsweek-Magazin beschreibt das Spiel als „den neuesten Stand des Gehirns“. Der Wert der IBM-Aktien stieg um 18 Milliarden Dollar. Schachmeister analysierten die Niederlage von Kasparov und erzielten in den folgenden Jahren öfter ein Remis, doch die jüngsten Mensch-Computer-Partien gewann überzeugend der Computer.

Spam-Bekämpfung: Jeden Tag klassifizieren Lernalgorithmen über eine Milliarde E-Mails als Spam und ersparen so den Empfängern das zeitaufwändige und mühsame Aussortieren unerwünschter Nachrichten, die bei vielen Benutzern 80% oder 90% des Posteinganges ausmachen könnten, wenn Algorithmen nicht vorher eingreifen würden. Da die Spam-Versender ständig ihre Taktik anpassen, ist es für ein statisch programmiertes Verfahren äußerst schwierig, Schritt zu halten. Lernende Algorithmen funktionieren hier am besten (Sahami et al., 1998; Goodman und Heckerman, 2004).

Logistische Planung: Während der Krise am Persischen Golf im Jahre 1991 setzten die US-Streitkräfte DART (Dynamic Analysis and Replanning Tool) ein (Cross und Walker, 1994), um eine automatisierte logistische Planung durchzuführen und Zeitpläne für Transportaufgaben zu erstellen. Daran beteiligt waren bis zu 50.000 Fahrzeuge, Ladung und Menschen gleichzeitig, und es mussten Ausgangspunkte, Ziele, Routen und Konfliktauflösung zwischen allen Parametern berücksichtigt werden. Die KI-Planungstechnik erlaubte es, innerhalb von Stunden einen Plan zu erstellen, für den man mit den älteren Methoden mehrere Wochen gebraucht hätte. Die DARPA (Defense Advanced Research Project Agency) berichtet, dass sich allein durch diese eine Anwendung die 30-jährigen Investitionen in die KI mehr als ausgezahlt hätten.

Robotik: Die Firma *iRobot* hat über zwei Millionen Roomba-Staubsaugerroboter für den Privathaushalt verkauft. Außerdem vertreibt die Firma den robusteren PackBot nach Irak und Afghanistan, wo er eingesetzt wird, um mit gefährlichen Stoffen umzugehen, Sprengstoffe zu beseitigen und Scharfschützen aufzuspüren.

Maschinelle Übersetzung: Ein Computerprogramm, das automatisch von Arabisch nach Englisch übersetzt und es einem Englisch Sprechenden ermöglicht, die Schlagzeile „Ardogan Confirms That Turkey Would Not Accept Any Pressure, Urging Them to Recognize Cyprus" zu übersetzen (Erdogan bestätigt, dass sich die Türkei nicht unter Druck setzen lässt, um Zypern anzuerkennen). Das Programm verwendet ein statistisches Modell, das aus Beispielen von Arabisch-Englisch-Übersetzungen und aus Beispielen von englischem Text mit insgesamt zwei Billionen Wörtern aufgebaut ist (Brants et al., 2007). Keiner der Informatiker im Team spricht Arabisch, doch sie beherrschen statistische Verfahren und maschinelle Lernalgorithmen.

Bibliografische und historische Hinweise

Der methodologische Stand der künstlichen Intelligenz wird in *The Sciences of the Artificial* von Herb Simon (1981) betrachtet, wo die Forschungsgebiete beschrieben werden, die sich mit komplexen Artefakten beschäftigen. Er erklärt, wie man die KI sowohl als Wissenschaft als auch als Mathematik betrachten kann. Cohen (1995) bietet einen Überblick über eine experimentelle Methodologie in KI.

Shieber (1994) diskutiert den Turing-Test (Turing, 1950) und prangert seine Nützlichkeit im Wettbewerb für den Loebner-Preis an. Ford und Hayes (1995) behaupten, dass der Test an sich für die KI nicht hilfreich ist. Bringsjord (2008) gibt einen Rat für einen Turing-Test-Schiedsrichter. Shieber (2004) und Epstein et al. (2008) sammeln eine Reihe von Abhandlungen zum Turing-Test. *Artificial Intelligence: The Very Idea* von John Haugeland (1985) bietet einen lesenswerten Bericht über die philosophischen und praktischen Probleme der KI. Eine Anthologie zu bedeutenden frühen Artikeln in der KI ist in den Sammelbänden von Webber und Nilsson (1981) und von Luger (1995) zu finden. Die *Encyclopedia of AI* (Shapiro, 1992) enthält ähnlich wie bei Wikipedia Übersichtsartikel zu fast jedem KI-Thema. Diese Artikel bieten in der Regel einen guten Ausgangspunkt für die Forschungsliteratur zu den einzelnen Themen. Eine aufschlussreiche und umfassende Geschichte der KI wird von Nils Nilsson (2009) dargestellt, einem der Pioniere auf diesem Gebiet.

Die neuesten Arbeiten erscheinen in den Tagungsbänden der wichtigsten KI-Konferenzen: der alle zwei Jahre stattfindenden IJCAI (International Joint Conference on AI), der jährlichen ECAI (European Conference on AI) und der National Conference on AI, nach der Sponsororganisation häufig auch als AAAI bezeichnet. Die wichtigsten Zeitschriften für allgemeine KI sind *Artificial Intelligence*, *Computational Intelligence*, die *IEEE Transactions on Pattern Analyses and Machine Intelligence*, *IEE Intelligent Systems* sowie die elektronische Zeitschrift *Journal of Artificial Intelligence Research*. Außerdem gibt es viele Konferenzen und Zeitschriften, die sich mit speziellen Bereichen beschäftigen und die wir in den entsprechenden Kapiteln vorstellen. Die wichtigsten professionellen Gesellschaften für KI sind die AAAI (American Association for Artificial Intelligence), die ACM SIGART (Special Interest Group in Artificial Intelligence) und die Society for Artificial Intelligence and Simulation of Behaviour (AISB). Das *AI Magazine* von AAAI enthält viele themenbezogene und lehrreiche Artikel. Darüber hinaus bietet die Webseite *aaai.org* Neuigkeiten und Hintergrundinformationen.

Zusammenfassung

Dieses Kapitel hat die KI definiert und den kulturellen Hintergrund erläutert, vor dem die Entwicklung stattgefunden hat. Die folgenden Punkte fassen einige der wichtigsten Aspekte zusammen:

- Unterschiedliche Menschen nähern sich der KI mit unterschiedlichen Zielen. Dabei müssen zwei wichtige Fragen gestellt werden: Geht es ihnen um Denken oder Verhalten? Wollen sie Menschen modellieren oder von einem Idealstandard ausgehen?
- In diesem Buch verfolgen wir die Perspektive, dass Intelligenz hauptsächlich mit **rationalem Handeln** zu tun hat. Im Idealfall ergreift ein **intelligenter Agent** die bestmögliche Aktion in einer Situation. Wir werden das Problem der Erstellung von Agenten, die in dieser Hinsicht intelligent sind, noch betrachten.
- Philosophen (zurück bis 400 v. Chr.) machten die KI wahrnehmbar, indem sie Theorien aufstellten, dass sich der Verstand manchmal wie eine Maschine verhält, dass er mit Wissen arbeitet, das in irgendeiner internen Sprache kodiert ist, und dass das Bewusstsein genutzt werden kann, um die auszuführenden Aktionen auszuwählen.
- Mathematiker stellten die Werkzeuge bereit um logisch sichere Aussagen sowie unsichere probabilistische Aussagen zu manipulieren. Außerdem entwickelten sie die Grundlagen für das Verständnis von Berechenbarkeit und das Schließen über Algorithmen.
- Wirtschaftswissenschaftler haben das Problem formalisiert, Entscheidungen zu treffen, die das erwartete Ergebnis für den Entscheidungstreffer maximieren.
- Neurowissenschaftler entdeckten einige Fakten, wie das Gehirn arbeitet und in welcher Hinsicht es Computern ähnelt bzw. sich davon unterscheidet.
- Psychologen übernahmen die Idee, dass Menschen und Tiere als Maschinen der Informationsverarbeitung betrachtet werden können. Die Linguistiker zeigten, dass die Sprachnutzung ebenfalls in dieses Modell passt.
- Computertechniker haben die immer leistungsfähigeren Maschinen bereitgestellt, die KI-Anwendungen erst möglich machen.
- Die Regelungstheorie beschäftigt sich mit dem Entwurf von Geräten, die auf Grundlage eines Feedbacks aus der Umgebung optimal agieren. Anfänglich unterschieden sich die mathematischen Werkzeuge der Regelungstheorie wesentlich von der KI, aber die beiden Gebiete rücken immer näher zusammen.
- Die Geschichte der KI zeigt Erfolge, falschen Optimismus und daraus resultierende Rückschritte im Hinblick auf Begeisterung und Finanzierung. Daneben gab es Zeiten, in denen neue kreative Ansätze eingeführt und die besten davon systematisch verbessert wurden.
- Die KI ist in den letzten zehn Jahren schneller vorangekommen, weil mehr wissenschaftliche Methoden bei den Experimenten mit und dem Vergleich von Ansätzen angewendet wurden.
- Jüngere Fortschritte im Verständnis für die theoretische Basis der Intelligenz gingen Hand in Hand mit Verbesserungen der Fähigkeiten realer Systeme. Die Teilbereiche der KI wurden stärker integriert und die KI fand eine gemeinsame Grundlage mit anderen Disziplinen.

Übungen zu Kapitel 1

Lösungshinweise

Diese Übungen sollen zur Diskussion anregen. Einige davon können als Semesterarbeiten dienen. Alternativ lassen sich jetzt erste Versuche durchführen und nach der vollständigen Durcharbeitung dieses Buches noch einmal betrachten.

1 Definieren Sie mit Ihren eigenen Worten: (a) Intelligenz, (b) künstliche Intelligenz, (c) Agent, (d) Rationalität, (e) logisches Schließen.

2 Jedes Jahr wird der Loebner-Preis dem Programm verliehen, das am besten eine Version des Turing-Tests besteht. Finden Sie heraus, wer im letzten Jahr der Gewinner des Loebner-Preises war, und berichten Sie darüber. Welche Techniken verwendet das Programm? Wie konnte es den aktuellen Stand der KI vorantreiben?

3 Sind Reflexhandlungen (wie das Zurückzucken von einem heißen Ofen) rational? Sind sie intelligent?

4 Es gibt bekannte Problemklassen, die für Computer schwierig handhabbar sind, und andere, die nachweisbar unentscheidbar sind. Bedeutet das, dass eine KI nicht möglich ist?

5 Die neuronale Struktur der Nacktschnecke *Aplysia* wurde eingehend untersucht (zuerst vom Nobelpreisträger Eric Kandel), da sie lediglich 20.000 Neuronen besitzt, von denen die meisten recht groß sind und sich leicht manipulieren lassen. Wie lässt sich die Rechenleistung in Bezug auf Speicheraktualisierungen pro Sekunde mit dem in Abbildung 1.3 beschriebenen High-End-Computer vergleichen, wenn man annimmt, dass die Zykluszeit für ein *Aplysia*-Neuron in der Größenordnung eines menschlichen Neurons liegt?

6 Wie könnte die Introspektion – eine Aufnahme der inneren Gedanken – ungenau sein? Könnte ich mich in Bezug darauf, was ich denke, irren? Diskutieren Sie die Antwort.

7 In welchem Umfang sind die folgenden Computersysteme Instanzen künstlicher Intelligenz:

- Barcode-Scanner im Supermarkt
- Sprachgesteuerte Telefonmenüs
- Funktionen der Rechtschreib- und Grammatikprüfung in Microsoft Word
- Algorithmen für das Internet-Routing, die dynamisch auf den Netzzustand reagieren

8 Viele der vorgeschlagenen Berechnungsmodelle für kognitive Aktivitäten enthalten recht komplexe mathematische Operationen, beispielsweise ein Bild mit einer Gauß-Funktion falten (d.h. weichzeichnen) oder das Minimum der Entropiefunktion finden. Die meisten Menschen (und zweifellos alle Tiere) lernen derartige mathematische Operationen niemals kennen, fast niemand hat damit vor einer Hochschulausbildung zu tun und praktisch niemand kann die Faltung einer Funktion mit einer Gauß-Funktion im Kopf berechnen. Ist es sinnvoll zu sagen, dass das „Bildverarbeitungssystem" derartige mathematische Berechnungen ausführt, während die Person an sich keine Ahnung davon hat, wie dies zu bewerkstelligen ist?

9 Laut einiger Autoren bilden Wahrnehmung und motorische Fertigkeiten den wichtigsten Teil der Intelligenz und „höhere" Fertigkeiten müssten zwangsläufig „parasitär" sein – einfache Ergänzungen dieser zugrunde liegenden Fertigkeiten. Sicherlich wurde ein Großteil der Evolution und ein Großteil des Gehirns der Wahrnehmung und den motorischen Fertigkeiten gewidmet, wobei die KI jedoch herausgefunden hat, dass Aufgaben wie beispielsweise Spiele und logische Schlussfolgerungen in vielen Fällen einfacher sind als Wahrnehmen und Handeln in der realen Welt. Glauben Sie, die traditionelle Konzentration der KI auf höhere kognitive Fähigkeiten sei fehl am Platz?

10 Handelt es sich bei künstlicher Intelligenz um Wissenschaft oder um Technik? Oder verkörpert sie beides? Erläutern Sie Ihre Antworten.

11 „Natürlich können Computer nicht intelligent sein – sie tun genau das, was ihnen ihre Programmierer befehlen." Trifft die zweite Aussage zu und impliziert sie gleichzeitig die erste?

12 „Natürlich können Tiere nicht intelligent sein – sie tun nur, was ihnen ihre Gene vorgeben." Trifft die zweite Aussage zu und impliziert sie gleichzeitig die erste?

13 „Natürlich können Tiere, Menschen und Computer nicht intelligent sein – sie tun nur, was ihnen ihre Atome nach den Gesetzen der Physik befehlen." Trifft die zweite Aussage zu und impliziert sie gleichzeitig die erste?

14 Informieren Sie sich in der KI-Literatur, ob die folgenden Aufgaben heute schon von Computern gelöst werden können:

a. Ein akzeptables Tischtennisspiel absolvieren
b. Mit dem Auto im Zentrum von Kairo fahren
c. In Victorville, Kalifornien, fahren
d. Den Wochenbedarf an Lebensmitteln im Kaufhaus einkaufen
e. Den Wochenbedarf an Lebensmitteln über das Internet einkaufen
f. Eine annehmbare Partie Bridge auf Wettkampfniveau spielen
g. Neue mathematische Theoreme entdecken und beweisen
h. Eine bewusst lustige Geschichte schreiben
i. Einen kompetenten rechtlichen Rat in einem speziellen juristischen Bereich erteilen
j. Gesprochenes Englisch in Echtzeit in gesprochenes Schwedisch übersetzen
k. Eine komplizierte chirurgische Operation ausführen

Versuchen Sie, für die derzeit noch nicht realisierbaren Aufgaben herauszufinden, welche Schwierigkeiten es gibt, und vorherzusagen, wann – falls überhaupt – diese überwunden sein werden.

15 In verschiedenen Teilgebieten der KI finden Wettbewerbe statt, indem man eine Standardaufgabe definiert und Forscher dazu einlädt, ihr Bestes zu geben. Beispiele hierfür sind die DARPA Grand Challenge für Roboterautos, der Internationale Planungswettbewerb (International Planning Competition, IPC), die Robocup-Roboterfußballliga, die TREC-Konferenz zum Abrufen von Informationen sowie Wettbewerbe in maschineller Übersetzung und Spracherkennung. Recherchieren Sie fünf dieser Wettbewerbe und beschreiben Sie die Fortschritte, die im Lauf der Jahre gemacht wurden. In welchem Umfang haben die Wettbewerbe den Stand der Technik in der KI vorangebracht? In welchem Maße haben sie dem Gebiet geschadet, da sie neuen Ideen die Energie entzogen haben?

Intelligente Agenten

2

ÜBERBLICK

In diesem Kapitel beschreiben wir das Wesen von Agenten (perfekt oder nicht), die Verschiedenartigkeit von Umgebungen sowie die resultierende Vielfalt der Agententypen.

In *Kapitel 1* wurde das Konzept der **rationalen Agenten** als zentral für unseren Zugang zur künstlichen Intelligenz bezeichnet. In diesem Kapitel wollen wir dieses Konzept konkretisieren. Wir werden sehen, dass sich das Konzept der Rationalität auf eine Vielzahl von Agenten anwenden lässt, die in jeder vorstellbaren Umgebung arbeiten. Wir wollen in diesem Buch anhand dieses Konzeptes eine kleine Menge von Entwurfsgrundlagen für das Erstellen erfolgreicher Agenten entwickeln – Systeme, die wirklich als **intelligent** bezeichnet werden können.

Wir beginnen mit der Betrachtung von Agenten, Umgebungen und ihrer Beziehung zueinander. Die Beobachtung, dass sich einige Agenten besser verhalten als andere, führt von selbst zu der Idee eines rationalen Agenten – das ist ein Agent, der sich so gut wie möglich verhält. Wie gut sich ein Agent verhalten kann, ist von der Art der Umgebung abhängig; einige Umgebungen sind schwieriger als andere. Wir werden hier eine vereinfachte Kategorisierung der Umgebungen skizzieren und darüber hinaus zeigen, wie die Eigenschaften einer Umgebung den Entwurf geeigneter Agenten für diese Umgebung beeinflussen. Wir beschreiben mehrere grundlegende „Gerüste“ für den Entwurf von Agenten, die im restlichen Buch weiter ausgebaut werden.

2.1 Agenten und Umgebungen

Ein **Agent** besitzt **Sensoren,** mit denen er seine Umgebung wahrnehmen kann, und **Aktuatoren,** durch die er handelt. Dieses einfache Konzept ist in ▶ Abbildung 2.1 dargestellt. Ein menschlicher Agent hat Augen, Ohren und andere Organe als Sensoren sowie Hände, Füße, Mund und andere Körperteile als Aktuatoren. Ein Roboter-Agent könnte Kameras und Infrarotentfernungsmesser als Sensoren verwenden und verschiedene Motoren als Aktuatoren. Ein Software-Agent übernimmt Tastenbetätigungen, Dateiinhalte und Netzwerkpakete als sensorische Eingaben und wirkt auf die Umgebung, indem er etwas auf dem Bildschirm anzeigt, Dateien schreibt oder Netzwerkpakete versendet.

Tipp

Wir verwenden den Begriff **Wahrnehmung**, um die wahrgenommenen Eingaben des Agenten zu jedem beliebigen Zeitpunkt zu beschreiben. Die **Wahrnehmungsfolge** eines Agenten ist der vollständige Verlauf von allem, was der Agent je wahrgenommen hat. Im Allgemeinen *kann die Auswahl einer Aktion durch den Agenten zu jedem bestimmten Zeitpunkt von der gesamten bisherigen Wahrnehmungsfolge abhängig sein, die bis dahin beobachtet wurde, doch nicht von etwas, was er noch nicht wahrgenommen hat.* Indem wir die Auswahl der Aktion des Agenten für jede mögliche Wahrnehmungsfolge spezifizieren, haben wir mehr oder weniger alles festgelegt, was es über den Agenten zu sagen gibt. Mathematisch ausgedrückt sagen wir, dass das Verhalten eines Agenten durch die **Agentenfunktion** beschrieben wird, die jede beliebige Wahrnehmungsfolge auf eine Aktion abbildet.

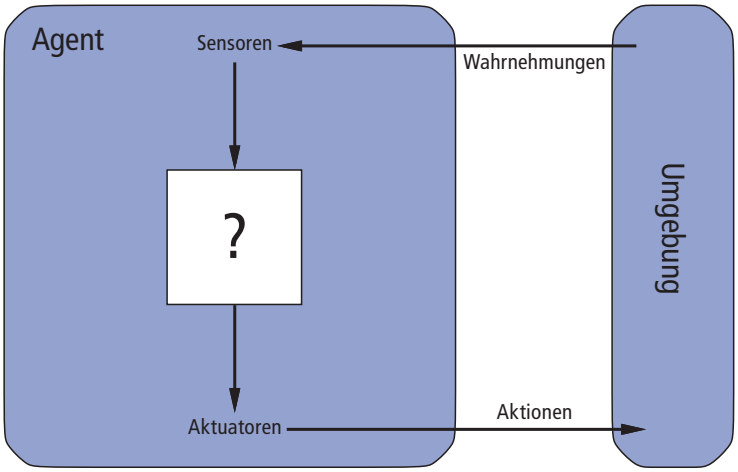


Abbildung 2.1: Agenten kommunizieren über Sensoren und Aktuatoren mit ihrer Umgebung.

Wir können uns vorstellen, die Agentenfunktion, die einen beliebigen Agenten beschreibt, *tabellarisch anzuordnen*. Für die meisten Agenten würde daraus eine sehr große Tabelle entstehen – eigentlich unendlich groß, es sei denn, wir begrenzen die Länge der Wahrnehmungsfolgen, die wir berücksichtigen wollen. Für einen Agenten, mit dem wir experimentieren wollen, können wir diese Tabelle erstellen, indem wir alle möglichen Wahrnehmungsfolgen ausprobieren und dann aufzeichnen, welche Aktionen der Agent als Reaktion darauf ausführt.[1] Die Tabelle ist natürlich eine externe Charakterisierung des Agenten. Intern wird die Agentenfunktion für einen künstlichen Agenten durch ein **Agentenprogramm** implementiert. Beachten Sie, dass diese beiden Konzepte unterschieden werden müssen. Die Agentenfunktion ist eine abstrakte mathematische Beschreibung; das Agentenprogramm ist eine konkrete Implementierung, die innerhalb eines physischen Systems ausgeführt wird.

Um diese Konzepte zu verdeutlichen, verwenden wir ein ganz einfaches Beispiel – die in ▶ Abbildung 2.2 gezeigte Staubsaugerwelt. Diese Welt ist so einfach, dass wir alles beschreiben können, was passieren kann; es handelt sich dabei um eine künstliche Welt, sodass wir verschiedene Varianten davon erfinden können. Diese konkrete Welt hat nur zwei Positionen: die Quadrate *A* und *B*. Der Staubsauger-Agent nimmt wahr, in welchem Quadrat er sich befindet und ob Schmutz in dem Quadrat liegt. Er kann entscheiden, sich nach links oder rechts zu bewegen, den Schmutz aufzusaugen oder nichts zu tun. Eine sehr einfache Agentenfunktion ist die folgende: Wenn das aktuelle Quadrat schmutzig ist, soll gesaugt werden, andernfalls soll eine Bewegung zum anderen Quadrat erfolgen. ▶ Abbildung 2.3 zeigt einen Teil der tabellarischen Darstellung dieser Agentenfunktion. Ein Agentenprogramm, das diese Funktion implementiert, finden Sie später in diesem Kapitel in ▶ Abbildung 2.8.

1 Wenn der Agent seine Aktionen einer gewissen Zufälligkeit nach ausführt, müssten wir jede Folge mehrfach ausprobieren, um die Wahrscheinlichkeit jeder Aktion festzustellen. Man könnte annehmen, ein zufälliges Verhalten sei dumm, aber später in diesem Kapitel zeigen wir, dass es sehr intelligent sein kann.

Tipp

Wie aus Abbildung 2.3 hervorgeht, können wir verschiedene Staubsaugerwelt-Agenten definieren, indem wir einfach die rechte Spalte unterschiedlich ausfüllen. Die offensichtliche Frage lautet also: Wie wird die Tabelle richtig ausgefüllt? Mit anderen Worten: Was macht einen Agenten gut oder schlecht, intelligent oder dumm? Wir beantworten diese Fragen im nächsten Abschnitt.

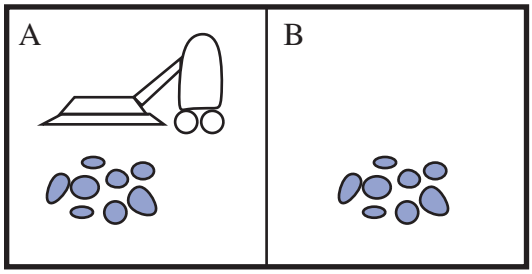


Abbildung 2.2: Eine Staubsaugerwelt mit nur zwei Positionen.

Wahrnehmungsfolge	Aktion
[*A, Sauber*]	*Rechts*
[*A, Schmutzig*]	*Saugen*
[*B, Sauber*]	*Links*
[*B, Schmutzig*]	*Saugen*
[*A, Sauber*], [*A, Sauber*]	*Rechts*
[*A, Sauber*], [*A, Schmutzig*]	*Saugen*
...	...
[*A, Sauber*], [*A, Sauber*], [*A, Sauber*]	*Rechts*
[*A, Sauber*], [*A, Sauber*], [*A, Schmutzig*]	*Saugen*
...	...

Abbildung 2.3: Ausschnitt aus einer tabellarischen Darstellung einer einfachen Agentenfunktion für die Staubsaugerwelt aus Abbildung 2.2.

Bevor wir diesen Abschnitt abschließen, wollen wir anmerken, dass das Konzept eines Agenten als Werkzeug für die Analyse von Systemen vorgesehen ist und nicht als absolute Charakterisierung, die die Welt in Agenten und Nichtagenten unterteilt. Man könnte selbst einen Taschenrechner als Agenten betrachten, der die Aktion wählt, „4“ anzuzeigen, wenn er die Wahrnehmungsfolge „2 + 2 =“ erhält, aber eine solche Analyse würde kaum zu unserem Verständnis für den Taschenrechner beitragen. In gewissem Sinne lassen sich sämtliche Bereiche der Technik als das Entwerfen von Artefakten ansehen, die mit der Welt kommunizieren. KI arbeitet am interessantesten Ende des Spektrums (so wie es die Autoren sehen), wo die Artefakte über beträchtliche Rechenressourcen verfügen und die Aufgabenumgebung voraussetzt, nichttriviale Entscheidungen zu treffen.

2.2 Gutes Verhalten: das Konzept der Rationalität

Ein **rationaler Agent** ist ein Agent, der das Richtige tut – einfacher ausgedrückt, jeder Eintrag in der Tabelle für die Agentenfunktion ist korrekt ausgefüllt. Offensichtlich ist es besser, das Richtige zu tun, als das Falsche zu tun, aber was bedeutet es, das Richtige zu tun?

Wir beantworten diese uralte Frage in uralter Manier: indem wir die Konsequenzen des Agentenverhaltens betrachten. Wenn ein Agent in eine Umgebung eingesetzt wird, generiert er eine Sequenz von Aktionen entsprechend den Wahrnehmungen, die er empfängt. Diese Aktionssequenz veranlasst die Umgebung, eine Folge von Zuständen zu durchlaufen. Entspricht die Folge unseren Vorstellungen, hat sich der Agent gut verhalten. Dieses Konzept des angestrebten Ergebnisses wird durch ein **Leistungsmaß** erfasst, das jede gegebene Sequenz von Umgebungszuständen auswertet.

Wir haben *Umgebungszustände* gesagt, nicht *Agentenzustände*. Wenn wir Erfolg als Meinung des Agenten zu seiner eigenen Leistung definieren, könnte ein Agent perfekte Rationalität erreichen, indem er einfach von sich selbst behauptet, seine Leistung sei perfekt. Insbesondere sind menschliche Agenten für den „Saure-Trauben-Effekt“ berüchtigt – vorzugeben, dass sie etwas eigentlich gar nicht wollen (z.B. einen Nobel-Preis), nachdem sie es nicht erhalten konnten.

Tipp

Offenbar gibt es kein feststehendes Leistungsmaß für alle Aufgaben und Agenten; in der Regel wird ein Konstrukteur ein den Umständen angepasstes Maß empfehlen. Das ist nicht so einfach, wie es sich anhört. Betrachten Sie zum Beispiel den Staubsauger-Agenten aus dem vorigen Abschnitt. Wir könnten vorschlagen, die Leistung zu messen, indem wir untersuchen, wie viel Schmutz in einer Achtstundenschicht entfernt wurde. Bei einem rationalen Agenten ist natürlich das, was Sie gefordert haben, das, was Sie erhalten. Ein rationaler Agent kann diese Leistungsbewertung maximieren, indem er den Schmutz aufsaugt, ihn dann wieder auf den Boden wirft, ihn dann wieder aufsaugt usw. Eine geeignetere Leistungsbewertung wäre, den Agenten dafür zu belohnen, einen sauberen Boden zu hinterlassen. Beispielsweise könnte je ein Punkt für jedes saubere Quadrat pro Zeitschritt erteilt werden (und vielleicht Strafpunkte für verbrauchte Elektrizität und erzeugten Lärm). *Als Faustregel kann formuliert werden: Es ist besser, Leistungsbewertungen nicht danach zu entwickeln, was man tatsächlich in der Umgebung haben will, sondern danach, wie man glaubt, dass sich der Agent verhalten soll.*

Selbst wenn die offensichtlichen Fallstricke umgangen werden, bleiben noch einige komplizierte Knoten zu entwirren. Beispielsweise basiert das Konzept des „sauberen Bodens“ im vorigen Abschnitt auf einer durchschnittlichen Sauberkeit im Laufe der Zeit. Dennoch kann dieselbe durchschnittliche Sauberkeit von zwei ganz unterschiedlichen Agenten erzielt werden, wobei der eine immer einen mittelmäßigen Job leistet, während der andere tatkräftig saugt, aber lange Pausen macht. Was davon zu bevorzugen ist, mag der Hausmeisterwissenschaft unterliegen, ist aber letztlich eine schwierige philosophische Frage mit weitreichenden Implikationen. Was ist besser – ein unbekümmertes Leben mit allen Höhen und Tiefen oder eine sichere, aber langweilige Existenz? Was ist besser – ein Wirtschaftssystem, in dem jeder in mittlerer Armut lebt, oder eines, in dem einige wenige im Überfluss leben, während andere sehr arm sind? Wir überlassen diese Fragen dem interessierten Leser als Übung.

2.2.1 Rationalität

Was jeweils rational ist, hängt von vier Dingen ab:

- der Leistungsbewertung, die das Erfolgskriterium definiert,
- dem Vorwissen des Agenten über die Umgebung,
- den Aktionen, die der Agent ausführen kann,
- der bisherigen Wahrnehmungsfolge des Agenten.

Tipp

Das führt zur **Definition eines rationalen Agenten**:

> *Ein rationaler Agent soll für jede mögliche Wahrnehmungsfolge eine Aktion auswählen, von der erwartet werden kann, dass sie seine Leistungsbewertung maximiert, wenn man seine Wahrnehmungsfolge sowie vorhandenes Wissen, über das er verfügt, in Betracht zieht.*

Betrachten wir den einfachen Staubsauger-Agenten, der ein Quadrat säubert, falls es schmutzig ist, und andernfalls zum nächsten Quadrat weitergeht – dies ist die in Abbildung 2.3 tabellarisch dargestellte Agentenfunktion. Handelt es sich dabei um einen rationalen Agenten? Es kommt darauf an! Zunächst müssen wir angeben, wie die Leistungsbewertung aussieht, was über die Umgebung bekannt ist und über welche Sensoren und Aktuatoren der Agent verfügt. Wir wollen Folgendes annehmen:

- Die Leistung wird mit einem Punkt für jedes saubere Quadrat in jedem Zeitschritt über eine „Lebensdauer“ von 1000 Zeitschritten bewertet.
- Die „Geografie“ der Umgebung ist *a priori* bekannt (Abbildung 2.2), aber die Schmutzverteilung und die Ausgangsposition des Agenten nicht. Saubere Quadrate bleiben sauber und durch Saugen wird das aktuelle Quadrat gereinigt. Die Aktionen *Links* und *Rechts* bewegen den Agenten nach links und rechts, außer wenn der Agent dadurch die Umgebung verlassen würde; in diesem Fall bleibt er dort, wo er ist.
- Die einzigen verfügbaren Aktionen sind *Links*, *Rechts* und *Saugen*.
- Der Agent nimmt seine Position korrekt wahr und erkennt, ob sie schmutzig ist.

Wir behaupten, dass der Agent *unter diesen Umständen* wirklich rational ist; seine erwartete Leistung ist mindestens so hoch wie die jedes anderen Agenten. In Übung 2.1 werden Sie dies beweisen.

Es ist leicht erkennbar, dass derselbe Agent unter anderen Umständen irrational wäre. Beispielsweise pendelt er, nachdem der gesamte Schmutz beseitigt ist, unnötig vor und zurück. Wenn die Leistungsbewertung eine Strafe von einem Punkt für jede Bewegung nach links oder rechts vergibt, schneidet der Agent relativ schlecht ab. Ein besserer Agent für diesen Fall würde nichts mehr tun, nachdem sicher ist, dass alle Quadrate sauber sind. Würden saubere Quadrate wieder schmutzig, sollte der Agent in bestimmten Zeitabständen eine Überprüfung vornehmen und sie gegebenenfalls wieder säubern. Wenn die Geografie der Umgebung unbekannt ist, muss der Agent sie erkunden, statt auf den Quadraten *A* und *B* zu verbleiben. In Übung 2.1 werden Sie Agenten für solche Fälle entwerfen.

2.2.2 Allwissenheit, Lernen und Autonomie

Wir müssen sorgfältig zwischen Rationalität und **Allwissenheit** unterscheiden. Ein allwissender Agent kennt das tatsächliche Ergebnis seiner Aktionen und kann entsprechend handeln; in der Realität ist Allwissenheit jedoch unmöglich. Betrachten Sie das folgende Beispiel: Eines Tages gehe ich an den Champs Elysées spazieren und erkenne einen alten Freund auf der anderen Straßenseite. Es ist kein Auto zu sehen und ich habe auch sonst keine Verabredung; deshalb ist es rational, auf die andere Straßenseite zu gehen. In der Zwischenzeit fällt in 33.000 Fuß Höhe eine Ladungstür aus einem Passagierflugzeug.[2] Und bevor ich es auf die andere Seite schaffe, werde ich platt gemacht. War es irrational, die Straße zu überqueren? Es ist sehr unwahrscheinlich, dass in meiner Todesanzeige steht: „Der Dummkopf versuchte, die Straße zu überqueren."

Dieses Beispiel zeigt, dass Rationalität nicht dasselbe wie Perfektion ist. Die Rationalität maximiert die *erwartete* Leistung, während die Perfektion die *tatsächliche* Leistung maximiert. Das Abrücken von der Forderung nach Perfektion ist nicht nur eine Frage der Fairness gegenüber Agenten. Entscheidend ist Folgendes: Wenn wir erwarten, dass ein Agent das tut, was sich im Nachhinein als beste Aktion herausstellt, ist es unmöglich, einen Agenten zu entwerfen, der diese Spezifikation erfüllt – es sei denn, wir verbessern die Leistung von Kristallkugeln oder Zeitmaschinen.

Für unsere Definition der Rationalität brauchen wir keine Allwissenheit, weil die rationale Auswahl nur von der *bisherigen* Wahrnehmungsfolge abhängig ist. Außerdem müssen wir sicherstellen, dass wir nicht versehentlich dem Agenten erlaubt haben, ausgesprochen schwachsinnige Aktivitäten auszuführen. Wenn ein Agent beispielsweise nicht nach beiden Seiten sieht, bevor er eine stark befahrene Straße überquert, kann ihm seine Wahrnehmungsfolge nicht vermitteln, dass sich gerade ein großer Lastwagen mit hoher Geschwindigkeit nähert. Besagt unsere Definition der Rationalität, dass es jetzt in Ordnung ist, die Straße zu überqueren? Mitnichten! Erstens wäre es unvernünftig, die Straße auf der Basis dieser nicht informativen Wahrnehmungsfolge zu überqueren: Das Risiko eines Unfalls durch ein Überqueren ohne sich umzuschauen ist zu groß. Zweitens sollte ein rationaler Agent die Aktion „Schauen" auswählen, bevor er auf die Straße tritt, weil das Schauen hilft, die erwartete Leistung zu maximieren.

Aktionen auszuführen, um *zukünftige Wahrnehmungen zu verändern* – manchmal auch als **Informationssammlung** bezeichnet –, ist ein wichtiger Teil der Rationalität und wird in *Kapitel 16* genauer beschrieben. Ein zweites Beispiel für das Sammeln von Informationen ist die **Exploration**, die von einem Staubsauger-Agenten in einer zunächst unbekannten Umgebung unternommen werden muss.

Unserer Definition nach muss ein rationaler Agent nicht nur Informationen sammeln, sondern auch so viel wie möglich aus seiner Wahrnehmung **lernen**. Die anfängliche Konfiguration des Agenten könnte ein gewisses Vorwissen über die Umgebung reflektieren, aber sobald der Agent Erfahrungen sammelt, können diese verändert und erweitert werden. Es gibt Extremfälle, wo die Umgebung *a priori* vollständig bekannt ist. In diesen Fällen muss der Agent nichts wahrnehmen oder lernen; er handelt einfach korrekt. Solche Agenten sind natürlich anfällig. Betrachten Sie den gemeinen Mistkäfer. Nachdem er sein Nest gegraben und seine Eier gelegt hat, holt er eine Mistkugel von einem nahe gelegenen Haufen, um den Eingang zu versperren. Nimmt man ihm die Mistkugel *unterwegs*

2 Siehe N. Henderson, „New door latches urged for Boeing 747 jumbo jets", Washington Post, 24. August 1989.

weg, setzt er seine Aufgabe einfach fort und spielt beim Versperren des Nestes mit der nicht existierenden Mistkugel, wobei er nicht einmal bemerkt, dass er sie nicht mehr besitzt. Die Evolution hat eine Annahme in das Verhalten des Käfers eingebaut und wenn diese verletzt wird, entsteht ein erfolgloses Verhalten. Etwas intelligenter ist die Heuschrecken-Grabwespe. Die weibliche Wespe gräbt eine Höhle, kommt heraus und beißt eine Raupe, zieht diese zur Höhle, geht wieder in die Höhle und überprüft, ob alles in Ordnung ist, zieht die Raupe in die Höhle und legt ihre Eier. Die Raupe dient als Nahrungsquelle für die aus den Eiern schlüpfenden Larven. So weit, so gut. Schiebt jedoch ein Entomologe die Raupe ein paar Zentimeter von ihrer ursprünglichen Position weg, während die Wespe ihre Überprüfung ausführt, gelangt sie wieder zum „Raupe-vor-die-Höhle-ziehen"-Schritt ihres Planes und setzt diesen unverändert fort, auch nach Dutzenden von Unterbrechungen durch ein Verschieben der Raupe. Die Wespe ist nicht in der Lage zu lernen, dass ihr ursprünglicher Plan fehlschlägt, und ändert ihn deshalb nicht.

Soweit sich der Agent auf das Vorwissen seines Entwicklers verlässt statt auf seine eigenen Wahrnehmungen, sprechen wir von fehlender **Autonomie** des Agenten. Ein rationaler Agent sollte autonom sein – er sollte lernen, was immer möglich ist, um unvollständiges oder fehlerhaftes Vorwissen zu kompensieren. Beispielsweise verhält sich ein Staubsauger-Agent, der lernt vorherzusehen, wo und wann zusätzlicher Schmutz erscheint, besser als ein Agent, der dies nicht kann. In der Praxis fordert man selten vollständige Autonomie von Beginn an: Wenn der Agent wenig oder keine Erfahrung hat, müsste er zufällig agieren, es sei denn, sein Entwickler gibt ihm eine Hilfestellung. So wie die Evolution Tiere mit ausreichend vielen eingebauten Reflexen versorgt, um lange genug zu überleben und selbst zu lernen, ist es sinnvoll, einen KI-Agenten ebenso mit einem Vorabwissen auszustatten sowie einer Fähigkeit zu lernen. Nach ausreichender Erfahrung mit seiner Umgebung kann das Verhalten eines rationalen Agenten letztlich unabhängig von seinem Vorwissen werden. Damit erlaubt es die Berücksichtigung des Lernens, einen einzigen rationalen Agenten zu entwickeln, der in unterschiedlichsten Umgebungen erfolgreich sein kann.

2.3 Die Natur der Umgebungen

Nachdem wir eine Definition der Rationalität besitzen, können wir fast schon darüber nachdenken, rationale Agenten zu entwickeln. Zunächst müssen wir uns jedoch Gedanken über **Aufgabenumgebungen** machen, die im Wesentlichen die „Probleme" darstellen, deren „Lösungen" die rationalen Agenten sind. Zunächst zeigen wir, wie eine Aufgabenumgebung spezifiziert wird, und verdeutlichen diesen Prozess anhand von zahlreichen Beispielen. Anschließend zeigen wir, dass es Aufgabenumgebungen in den verschiedensten Varianten gibt. Die Variante der Aufgabenumgebung wirkt sich direkt darauf aus, welches Design für das Agentenprogramm geeignet ist.

2.3.1 Spezifizieren der Aufgabenumgebung

In unserer Beschreibung zur Rationalität des einfachen Staubsauger-Agenten mussten wir eine Leistungsbewertung, die Umgebung sowie die Aktuatoren und Sensoren des Agenten spezifizieren. Wir werden diese Komponenten nun unter der Überschrift der **Aufgabenumgebung** gruppieren. Für diejenigen unter uns, die gerne Abkürzungen verwenden, sprechen wir auch von der **PEAS**-Beschreibung (**P**erformance, **E**nviron-

ment, **A**ctuators, **S**ensors – Leistung, Umgebung, Aktuatoren, Sensoren). Bei der Entwicklung eines Agenten muss der erste Schritt immer darin bestehen, die Aufgabenumgebung so vollständig wie möglich zu spezifizieren.

Die Staubsaugerwelt war ein einfaches Beispiel. Jetzt wollen wir ein komplexeres Problem betrachten: einen automatisierten Taxifahrer. Bevor sich der Leser größere Sorgen macht, wollen wir jedoch darauf hinweisen, dass ein vollständig automatisiertes Taxi den Fähigkeiten der existierenden Technologie noch weit voraus ist. (*Abschnitt 1.4* beschreibt ein existierendes Roboterfahrzeug.) Die vollständige Aufgabe des Fahrens ist äußerst *offen*. Es gibt keine Begrenzung im Hinblick auf neue Kombinationen der Umstände, die auftreten können – ein weiterer Grund dafür, warum wir uns in unserer Diskussion darauf konzentrieren wollen. ▶ Abbildung 2.4 bietet einen Überblick über die PEAS-Beschreibung für die Aufgabenumgebung des Taxis. Wir stellen die einzelnen Elemente in den nachfolgenden Abschnitten genauer vor.

Agententyp	Leistungsbewertung	Umgebung	Aktuatoren	Sensoren
Taxifahrer	sicher, schnell, der Straßenverkehrsordnung gehorchend, angenehme Fahrweise, maximale Gewinne	Straßen, anderer Verkehr, Fußgänger, Fahrgäste	Steuerrad, Gas, Bremse, Hupe, Blinker, Anzeige	Kameras, Sonar, Tachometer, GPS, Kilometerzähler, Motorsensoren, Tastatur

Abbildung 2.4: PEAS-Beschreibung der Aufgabenumgebung für ein automatisiertes Taxi.

Erstens, welche **Leistungsbewertung** streben wir für unseren automatisierten Fahrer an? Wünschenswerte Qualitäten sind unter anderem, zum korrekten Ziel zu gelangen, möglichst wenig Treibstoff zu verbrauchen, verschleißarm zu fahren, die Fahrzeit und/oder die Kosten gering zu halten, Verletzungen von Verkehrsregeln ebenso wie Behinderungen anderer Fahrer zu vermeiden; maximale Sicherheit und Komfort für den Fahrgast zu bieten und maximale Gewinne zu erzielen. Offensichtlich stehen einige dieser Ziele im Widerspruch zueinander; deshalb werden Kompromisse erforderlich sein.

Zweitens, welche **Fahrumgebung** findet das Taxi vor? Jeder Taxifahrer muss mit einer Vielzahl von Straßen zurechtkommen, von Landstraßen und Wohnstraßen in der Stadt bis hin zu zwölfspurigen Autobahnen. Auf den Straßen ist mit anderen Verkehrsteilnehmern, Fußgängern, streunenden Tieren, Baustellen, Polizeiautos, Pfützen und Schlaglöchern zu rechnen. Außerdem muss das Taxi mit potenziellen und tatsächlichen Fahrgästen kommunizieren. Darüber hinaus gibt es einige optionale Auswahlmöglichkeiten. Das Taxi könnte in Südkalifornien eingesetzt werden, wo es kaum Schnee gibt, aber auch in Alaska, wo fast immer Schnee liegt. Es könnte immer rechts fahren, aber wir könnten auch verlangen, dass es flexibel genug ist, um links zu fahren, wenn es in Großbritannien oder Japan eingesetzt werden soll. Offensichtlich wird das Entwurfsproblem umso einfacher, je eingeschränkter die Umgebung ist.

Die einem automatisierten Taxi zur Verfügung stehenden **Aktuatoren** sind mehr oder weniger dieselben wie die eines menschlichen Fahrers: Bedienung des Motors über das Gaspedal und Steuerung über Lenkung und Bremse. Darüber hinaus muss er einen Anzeigebildschirm oder einen Sprachsynthesizer zur Verfügung stellen, um mit den Fahrgästen zu sprechen, und vielleicht eine Möglichkeit finden, mit anderen Fahrzeugen zu kommunizieren – freundlich oder auch anders.

Zu den grundlegenden **Sensoren** für das Taxi gehören eine oder mehrere steuerbare Videokameras, damit das Taxi die Straße sehen kann. Dies lässt sich ergänzen durch Infrarot- oder Ultraschallsensoren, die die Abstände zu anderen Fahrzeugen und Hindernissen erkennen. Um Blitzfotos wegen Geschwindigkeitsüberschreitungen zu vermeiden, sollte das Taxi mit einem Tachometer ausgerüstet sein. Zudem empfiehlt sich für ein gepflegtes Fahrverhalten – speziell in Kurven – ein Beschleunigungsmesser. Damit sich der mechanische Zustand des Fahrzeuges feststellen lässt, benötigt es die üblichen Sensoren für Motor, Kraftstoff und Elektrik. Wie für menschliche Fahrer ist zudem ein GPS-Gerät (Global Positioning System) von Vorteil, damit sich das Fahrzeug nicht verirrt. Schließlich braucht es eine Tastatur oder ein Mikrofon, damit die Fahrgäste das gewünschte Ziel mitteilen können.

In ▶ Abbildung 2.5 skizzieren wir die grundlegenden PEAS-Elemente für einige zusätzliche Agententypen. Weitere Beispiele finden Sie in Übung 2.4. Einige Leser sind vielleicht überrascht, dass unsere Liste der Agententypen einige Programme enthält, die in einer völlig künstlichen Umgebung arbeiten – definiert durch Tastatureingaben und Zeichenausgaben auf einem Bildschirm. „Natürlich", könnte man sagen, „immerhin ist es ja keine echte Umgebung." Letztlich spielt jedoch nicht die Unterscheidung zwischen „realer" und „künstlicher" Umgebung eine Rolle, sondern die Komplexität der Beziehungen zwischen dem Verhalten des Agenten, der durch die Umgebung erzeugten Wahrnehmungsfolge und der Leistungsbewertung. Einige „reale" Umgebungen sind tatsächlich ganz einfach. Beispielsweise kann ein Roboter, der für die Qualitätskontrolle von Teilen auf einem Fließband verantwortlich ist, mehrere vereinfachende Annahmen treffen: dass die Beleuchtung immer gleich ist, dass auf dem Fließband nur Teile erscheinen, die er kennt, und dass nur zwei Aktionen (Annehmen oder Verwerfen) möglich sind.

Agententyp	Leistungs-bewertung	Umgebung	Aktuatoren	Sensoren
Medizinisches Diagnose-system	Gesunder Patient, verringerte Kosten	Patient, Krankenhaus, Team	Anzeige von Fragen, Untersuchungen, Diagnosen, Behand-lungen, Empfehlungen	Tastatureingabe der Symptome, Befunde, Antwor-ten des Patienten
Satellitenbild-Analysesystem	Korrekte Bild-kategorisierung	Downlink vom Satelliten in der Umlaufbahn	Anzeige der Szenen-kategorisierung	Farbpixel-Arrays
Packroboter für Teile	Prozentsatz der Teile in korrekten Behältern	Fließband mit Teilen; Behälter	Arm und Greifhand	Kamera, Winkelsensoren
Raffinerie-Controller	Reinheit, Ausbeute, Sicherheit	Raffinerie, Arbeiter	Ventile, Pumpen, Heizungen, Anzeigen	Temperatur, Druck, chemische Sensoren
Interaktiver Englischlehrer	Punktzahl des Schülers bei einem Test	Schülergruppe, Prüfungs-kommission	Anzeige von Übun-gen, Vorschlägen, Korrekturen	Tastatureingabe

Abbildung 2.5: Beispiele für Agententypen und ihre PEAS-Beschreibungen.

Im Gegensatz dazu existieren einige **Software-Agenten** (auch als Software-Roboter oder **Softbots** bezeichnet) in umfangreichen, unbegrenzten Bereichen. Stellen Sie sich einen Softbot-Website-Operator vor, der Nachrichtenquellen im Internet durchsuchen und dem Benutzer die interessanten Artikel anzeigen soll, während er Werbeflächen verkauft, um Einnahmen zu erzielen. Um optimal zu arbeiten, braucht er Verarbeitungsfähigkeiten für natürliche Sprache, er muss lernen, woran jeder Benutzer und Inserent interessiert ist, und er muss seine Pläne dynamisch ändern können – wenn beispielsweise die Verbindung zu einer Nachrichtenquelle ausfällt oder wenn eine neue Quelle online geht. Das Internet ist eine Umgebung, deren Komplexität der physischen Welt gleichkommt und zu deren Einwohnern viele künstliche und menschliche Agenten gehören.

2.3.2 Eigenschaften von Aufgabenumgebungen

Der Bereich an Aufgabenumgebungen, die in der KI auftreten können, ist offensichtlich riesig. Wir können jedoch eine relativ kleine Anzahl an Dimensionen identifizieren, nach denen sich die Aufgabenumgebungen kategorisieren lassen. Diese Dimensionen legen weitgehend das zweckmäßige Design für einen Agenten fest ebenso wie die Anwendbarkeit der grundlegenden Technikfamilien für die Implementierung von Agenten. Zunächst listen wir die Dimensionen auf und analysieren dann mehrere Aufgabenumgebungen, um die Konzepte zu verdeutlichen. Die Definitionen sind hier formlos dargestellt; spätere Kapitel enthalten exaktere Aussagen und Beispiele für die einzelnen Umgebungen.

■ **Vollständig beobachtbar** im Vergleich zu **teilweise beobachtbar**

Wenn die Sensoren eines Agenten ihm zu jedem beliebigen Zeitpunkt Zugriff auf den vollständigen Zustand der Umgebung bieten, sagen wir, die Aufgabenumgebung ist vollständig beobachtbar.[3] Eine Aufgabenumgebung ist dann vollständig beobachtbar, wenn die Sensoren alle für die Auswahl einer Aktion relevanten Aspekte erkennen; die Relevanz wiederum ist von der Leistungsbewertung abhängig. Vollständig beobachtbare Umgebungen sind praktisch, weil der Agent keinen internen Zustand verwalten muss, um die Welt zu beobachten. Eine Umgebung kann teilweise beobachtbar sein, wenn Störungen oder ungenaue Sensoren vorliegen oder wenn Teile des Zustandes einfach nicht in den Sensordaten enthalten sind – beispielsweise kann ein Staubsauger-Agent mit nur einem lokalen Schmutzsensor nicht erkennen, ob Schmutz in anderen Quadraten vorliegt, und ein automatisiertes Taxi kann nicht sehen, was andere Fahrer denken. Besitzt der Agent überhaupt keine Sensoren, ist die Umgebung **nicht beobachtbar**. Man könnte annehmen, dass die Lage des Agenten in derartigen Fällen hoffnungslos ist, doch wie *Kapitel 4* noch zeigt, können die Ziele des Agenten dennoch erreichbar sein, manchmal mit Bestimmtheit.

■ **Einzelagent** im Vergleich zum **Multiagenten**

Die Unterscheidung zwischen Einzelagenten- und Multiagentenumgebungen scheint einfach zu sein. Beispielsweise ist ein Agent, der eigenständig ein Kreuzworträtsel löst, offensichtlich in einer Einzelagentenumgebung angesiedelt, während sich ein Agent, der Schach spielt, in einer Zweiagentenumgebung befindet. Es gibt jedoch einige subtile

3 Die erste Auflage dieses Buches verwendete die Begriffe **zugreifbar** und **nicht zugreifbar** an Stelle von **vollständig** und **teilweise beobachtbar**; **nichtdeterministisch** an Stelle von **stochastisch** und **nicht episodisch** an Stelle von **sequenziell**. Die neue Terminologie ist konsistenter zum üblichen Sprachgebrauch.

Feinheiten. Erstens haben wir beschrieben, wie eine Entität als Agent betrachtet werden *kann*, aber wir haben nicht erklärt, welche Entitäten als Agenten betrachtet werden *müssen*. Muss ein Agent A (beispielsweise der Taxifahrer) ein Objekt B (ein anderes Fahrzeug) als Agenten behandeln oder kann er es einfach als ein Objekt ansehen, das sich entsprechend der physikalischen Gesetze verhält, vergleichbar mit Wellen am Strand oder Blättern im Wind? Die Schlüsselunterscheidung ist, ob das Verhalten von B am besten damit beschrieben wird, dass es eine Leistungsbewertung maximiert, deren Wert vom Verhalten von Agent A abhängig ist. Im Schach beispielsweise versucht die gegnerische Entität, ihre Leistungsbewertung zu maximieren, was den Schachregeln entsprechend die Leistungsbewertung von Agent A minimiert. Schach ist also eine **konkurrierende** Multiagentenumgebung. In der Taxifahrerumgebung dagegen maximiert das Vermeiden von Zusammenstößen die Leistungsbewertung aller Agenten; es handelt sich also um eine partiell **kooperative** Multiagentenumgebung. Zum Teil ist sie auch konkurrierend, weil beispielsweise jeweils nur ein Auto einen Parkplatz belegen kann. Die Probleme beim Entwurf von Agenten in Multiagentenumgebungen unterscheiden sich häufig stark von denen in Einzelagentenumgebungen; beispielsweise erweist sich die **Kommunikation** in Multiagentenumgebungen häufig als rationales Verhalten; in einigen konkurrierenden Umgebungen ist ein **Zufallsverhalten** rational, weil es die Fallstricke der Vorhersehbarkeit vermeidet.

■ Deterministisch im Vergleich zu stochastisch

Wenn der nächste Zustand der Umgebung vollständig durch den aktuellen Zustand und die durch den Agenten ausgeführten Aktionen festgelegt wird, sagen wir, die Umgebung ist **deterministisch**; andernfalls ist sie **stochastisch**. Im Prinzip muss sich ein Agent keine Sorgen über Unsicherheiten in einer vollständig beobachtbaren, deterministischen Umgebung machen. (In unserer Definition ignorieren wir Unbestimmtheit, die ausschließlich von den Aktionen anderer Agenten in einer Multiagentenumgebung herrührt; somit kann ein Spiel deterministisch sein, selbst wenn der Agent eventuell nicht in der Lage ist, die Aktionen der anderen vorherzusagen.) Ist die Umgebung teilweise beobachtbar, könnte sie allerdings als stochastisch *erscheinen*. Die meisten realen Situationen sind so komplex, dass es unmöglich ist, alle nicht beobachteten Aspekte zu verfolgen; aus praktischen Gründen müssen sie als stochastisch behandelt werden. Das Taxifahren ist in dieser Hinsicht zweifellos stochastisch, weil man das Verhalten des Verkehrs nie genau vorhersehen kann; darüber hinaus können die Reifen platzen oder der Motor fällt aus, ohne dass es eine Vorwarnung gegeben hat. Die Staubsaugerwelt, die wir beschrieben haben, ist deterministisch, aber Varianten davon können stochastische Elemente beinhalten, wie beispielsweise zufällig erscheinenden Schmutz oder einen unzuverlässigen Saugmechanismus (Übung 2.13). Wir bezeichnen eine Umgebung als **unbestimmt**, wenn sie nicht vollständig beobachtbar oder nicht deterministisch ist. Schließlich sei angemerkt: Unsere Verwendung des Wortes „stochastisch“ impliziert im Allgemeinen, dass Unbestimmtheit hinsichtlich der Ergebnisse in Form von Wahrscheinlichkeiten ausgedrückt wird; als **nichtdeterministisch** gilt eine Umgebung, in der Aktionen durch ihre möglichen Ergebnisse charakterisiert werden, ihnen aber keine Wahrscheinlichkeiten zugeordnet sind. Nichtdeterministische Umgebungsbeschreibungen sind üblicherweise mit Leistungsbewertungen verknüpft, die voraussetzen, dass der Agent für alle möglichen Ergebnisse seiner Aktionen erfolgreich ist.

■ **Episodisch** im Vergleich zu **sequenziell**

In einer episodischen Aufgabenumgebung ist die Erfahrung des Agenten in atomare Episoden unterteilt. In jeder Episode empfängt der Agent eine Wahrnehmung und führt dann eine einzelne Aktion aus. Entscheidend ist, dass die nächste Episode nicht von in früheren Episoden ausgeführten Aktionen abhängt. Viele Klassifizierungsaufgaben sind episodisch. Beispielsweise basiert jede Entscheidung eines Agenten, der defekte Teile auf einem Fließband erkennen soll, auf dem aktuell betrachteten Teil, unabhängig von früheren Entscheidungen; darüber hinaus hat die aktuelle Entscheidung keinen Einfluss darauf, ob das nächste Teil defekt ist. In sequenziellen Umgebungen dagegen könnte die aktuelle Entscheidung alle zukünftigen Entscheidungen beeinflussen.[4] Schach und Taxifahren sind sequenziell: In beiden Fällen haben Kurzzeitaktionen Langzeitwirkungen. Episodische Umgebungen sind viel einfacher als sequenzielle Umgebungen, weil der Agent nicht vorausdenken muss.

■ **Statisch** im Vergleich zu **dynamisch**

Kann sich die Umgebung ändern, während ein Agent eine Entscheidung trifft, sprechen wir von einer dynamischen Umgebung für diesen Agenten; andernfalls ist sie statisch. Der Umgang mit statischen Umgebungen ist einfach, weil der Agent die Welt nicht beobachten muss, während er eine Aktion beschließt, und er muss sich auch keine Gedanken über den Zeitablauf machen. Dynamische Umgebungen dagegen fragen den Agenten ständig, was er tun will; wenn er noch nicht entschieden hat, ist dies gleichbedeutend damit, als wolle er nichts tun. Ändert sich die Umgebung selbst im Laufe der Zeit nicht, wohl aber die Leistungsbewertung des Agenten, sagen wir, die Umgebung ist **semidynamisch**. Taxifahren ist offensichtlich dynamisch: Die anderen Autos und das Taxi selbst bewegen sich, während der Fahralgorithmus berechnet, was als Nächstes zu tun ist. Schach ist, wenn es mit einer Uhr gespielt wird, semidynamisch. Kreuzworträtseln ist statisch.

■ **Diskret** im Vergleich zu **stetig**

Die Unterscheidung zwischen diskret und stetig bezieht sich auf den *Zustand* der Umgebung, auf die Behandlung der *Zeit* sowie auf die *Wahrnehmungen* und *Aktionen* des Agenten. Beispielsweise hat die Schachumgebung eine endliche Anzahl unterschiedlicher Zustände (die Uhr ausgenommen). Außerdem ist Schach durch eine diskrete Menge von Wahrnehmungen und Aktionen gekennzeichnet. Taxifahren ist eine Aufgabe mit stetigem Zustand und stetiger Zeit: Die Geschwindigkeit und die Position des Taxis und der anderen Fahrzeuge durchlaufen ständig einen Bereich stetiger Werte, und zwar ohne sprunghafte Veränderungen über die Zeit. Die Aktionen für das Taxifahren sind ebenfalls stetig (Lenkwinkel usw.). Die Eingabedaten von digitalen Kameras sind streng betrachtet diskret, werden aber typischerweise als Darstellung stetig variierender Intensität und Positionen betrachtet.

■ **Bekannt** im Vergleich zu **unbekannt**

Streng genommen bezieht sich diese Unterscheidung nicht auf die Umgebung an sich, sondern auf den Wissenstand von Agenten (oder Designern) über die „physikalischen Gesetze“ der Umgebung. In einer bekannten Umgebung sind die Ergebnisse (oder in einer stochastischen Umgebung die Ergebniswahrscheinlichkeiten) für alle Aktionen

4 Das Wort „sequenziell“ wird in der Informatik auch als das Gegenteil von „parallel“ verwendet. Die beiden Bedeutungen haben aber im Wesentlichen nichts miteinander zu tun.

gegeben. Wenn die Umgebung unbekannt ist, muss der Agent offensichtlich lernen, wie sie funktioniert, um gute Entscheidungen treffen zu können. Die Unterscheidung zwischen bekannten und unbekannten Umgebungen ist nicht die gleiche wie zwischen vollständig und teilweise beobachtbaren Umgebungen. Eine *bekannte* Umgebung kann durchaus *teilweise* beobachtbar sein – zum Beispiel kenne ich bei Solitär-Kartenspielen zwar die Regeln, kann aber nicht die Karten sehen, die noch nicht umgedreht wurden. Umgekehrt kann eine *unbekannte* Umgebung *vollständig* beobachtbar sein – in einem neuen Videospiel kann der Bildschirm das gesamte Spiel zeigen, doch ich weiß erst, was die Schalter bewirken, wenn ich sie ausprobiere.

Aufgabenumgebung	Beobachtbar	Agenten	Deterministisch	Episodisch	Statisch	Diskret
Kreuzworträtsel	vollständig	Einzel	deterministisch	sequenziell	statisch	diskret
Schach mit Uhr	vollständig	Multi	strategisch	sequenziell	semi-dynamisch	diskret
Poker	teilweise	Multi	strategisch	sequenziell	statisch	diskret
Backgammon	vollständig	Multi	stochastisch	sequenziell	statisch	diskret
Taxifahren	teilweise	Multi	stochastisch	sequenziell	dynamisch	stetig
Medizinische Diagnose	teilweise	Einzel	stochastisch	sequenziell	dynamisch	stetig
Bildanalyse	vollständig	Einzel	deterministisch	episodisch	semi-dynamisch	stetig
Teile packender Roboter	teilweise	Einzel	stochastisch	episodisch	dynamisch	stetig
Raffinerie-Controller	teilweise	Einzel	stochastisch	sequenziell	dynamisch	stetig
Interaktiver Englischlehrer	teilweise	Multi	stochastisch	sequenziell	dynamisch	diskret

Abbildung 2.6: Beispiele für Aufgabenumgebungen und ihre Eigenschaften.

Wie zu erwarten, ist der schwierigste Fall *teilweise beobachtbar*, *Multiagent*, *stochastisch*, *sequenziell*, *dynamisch*, *stetig* und *unbekannt*. Taxifahren ist unter allen diesen Aspekten schwierig, außer dass die Umgebung des Fahrers zum größten Teil bekannt ist. Das Führen eines Mietwagens in einem neuen Land mit nicht vertrauten geografischen Gegebenheiten und Verkehrsregeln ist noch etwas aufregender.

▶ Abbildung 2.6 listet die Eigenschaften mehrerer bekannter Umgebungen auf. Beachten Sie, dass die Antworten nicht immer ganz eindeutig sind. Beispielsweise beschreiben wir den Teile packenden Roboter als episodisch, da er normalerweise jedes Teil isoliert von anderen betrachtet. Doch wenn einmal ein großes Los defekter Teile auftaucht, sollte der Roboter aus mehreren Beobachtungen lernen, dass sich die Verteilung von Defekten geändert hat, und sein Verhalten für darauffolgende Teile entsprechend anpassen. Wir haben keine Spalte „bekannt/unbekannt“ aufgenommen, da dies wie bereits weiter vorn erklärt streng genommen keine Eigenschaft der Umgebung ist. Für bestimmte Umgebungen wie zum Beispiel Schach und Poker ist es zwar relativ leicht, den Agenten mit voll-

ständigen Regelkenntnissen auszustatten, doch ist es gleichwohl interessant zu betrachten, wie ein Agent lernt, diese Spiele ohne derartige Vorkenntnisse zu meistern.

Einige Antworten in der Tabelle hängen davon ab, wie die Aufgabenumgebung definiert ist. Wir haben die Aufgabe der medizinischen Diagnose als Einzelagent eingeordnet, weil der Krankheitsprozess eines Patienten nicht profitabel als Agent modelliert wird; ein System zur medizinischen Diagnose könnte jedoch auch mit widerspenstigen Patienten und einem skeptischen Team zu tun haben, sodass die Umgebung auch einen Multiagentenaspekt haben könnte. Darüber hinaus ist die medizinische Diagnose episodisch, wenn man die Aufgabe als Auswahl einer Diagnose anhand einer gegebenen Liste von Symptomen betrachtet; das Problem ist sequenziell, wenn die Aufgabe den Vorschlag einer Testreihe, die Auswertung von Fortschritten im Laufe der Behandlung usw. beinhalten kann. Außerdem sind viele Umgebungen episodisch auf höherer Ebene als die einzelnen Aktionen des Agenten. Beispielsweise besteht ein Schachturnier aus einer Folge von Spielen; jedes Spiel ist eine Episode, weil (im Großen und Ganzen) der Beitrag der Züge in einem Spiel zur Gesamtleistung des Agenten nicht durch die Züge seines vorherigen Spiels beeinflusst wird. Andererseits ist die Entscheidungsfindung innerhalb eines einzelnen Spiels natürlich sequenziell.

Der zu diesem Buch bereitgestellte Code (*aima.cs.berkeley.edu*) beinhaltet Implementierungen für eine Reihe von Umgebungen zusammen mit einem allgemeinen Umgebungssimulator, der einen oder mehrere Agenten in einer simulierten Umgebung platziert, ihr Verhalten über die Zeit beobachtet und sie gemäß einer vorgegebenen Leistungsbewertung bemisst. Solche Experimente werden häufig nicht für eine einzelne Umgebung, sondern für viele Umgebungen, die aus einer **Umgebungsklasse** ausgewählt werden, ausgeführt. Um beispielsweise einen Taxifahrer im simulierten Verkehr auszuwerten, müssten wir viele Simulationen mit unterschiedlichen Verkehrsaufkommen, Lichtverhältnissen und Wetterbedingungen durchführen. Würden wir den Agenten nur für ein einzelnes Szenario entwerfen, könnten wir bestimmte Eigenschaften dieser spezifischen Situation nutzen, was jedoch kein guter Entwurf für das Fahren im Allgemeinen wäre. Aus diesem Grund wird in dem bereitgestellten Code auch ein **Umgebungsgenerator** für jede Umgebungsklasse angeboten, der einzelne Umgebungen (mit bestimmten Wahrscheinlichkeiten) auswählt, in denen der Agent ausgeführt werden soll. Beispielsweise initialisiert der Umgebungsgenerator für den Staubsauger das Schmutzmuster sowie die Position des Agenten zufällig. Wir sind an der durchschnittlichen Leistung des Agenten in der Umgebungsklasse interessiert. Ein rationaler Agent für eine bestimmte Umgebungsklasse maximiert diese durchschnittliche Leistung. In den Übungen 2.9 bis 2.13 durchlaufen Sie den Prozess, eine Umgebungsklasse zu entwickeln und dort verschiedene Agenten zu bewerten.

2.4 Die Struktur von Agenten

Bei unserer bisherigen Betrachtung der Agenten haben wir ihr *Verhalten* beschrieben – die Aktion, die nach einer bestimmten Wahrnehmungsfolge ausgeführt wurde. Nun müssen wir in den sauren Apfel beißen und darüber sprechen, wie ein Agent intern arbeitet. Die Aufgabe der künstlichen Intelligenz ist es, das **Agentenprogramm** zu entwerfen, das die Agentenfunktion implementiert – die Zuordnung von Wahrnehmungen zu Aktionen. Wir gehen davon aus, dass dieses Programm auf einem Computer

mit physischen Sensoren und Aktuatoren ausgeführt wird – was wir auch als **Architektur** bezeichnen wollen:

$$Agent = Architektur + Programm$$

Offensichtlich muss das Programm, das wir wählen, für die Architektur geeignet sein. Wenn das Programm Aktionen wie beispielsweise *Gehen* empfiehlt, sollte die Architektur also Füße haben. Bei der Architektur kann es sich um einen gewöhnlichen PC handeln oder aber um ein Roboterauto mit mehreren Computern, Kameras und anderen Sensoren an Bord. Im Allgemeinen ist die Architektur dafür verantwortlich, die Wahrnehmungen von den dem Programm verfügbaren Sensoren entgegenzunehmen, das Programm auszuführen und die Aktionsauswahl des Programms nach ihrer Erzeugung an die Aktuatoren weiterzugeben. Ein Großteil dieses Buches beschreibt den Entwurf von Agentenprogrammen, während sich die *Kapitel 24* und *25* direkt mit Sensoren und Aktuatoren beschäftigen.

2.4.1 Agentenprogramme

Die Agentenprogramme, die wir in diesem Buch entwerfen, haben alle denselben Aufbau: Sie nehmen die aktuelle Wahrnehmung als Eingabe von den Sensoren entgegen und geben eine Aktion an die Aktuatoren zurück.[5] Beachten Sie den Unterschied zwischen dem Agentenprogramm, das die aktuelle Wahrnehmung als Eingabe entgegennimmt, und der Agentenfunktion, die den gesamten Wahrnehmungsverlauf entgegennimmt. Das Agentenprogramm übernimmt nur die aktuelle Wahrnehmung als Eingabe, weil aus der Umgebung nichts weiter zur Verfügung gestellt wird. Müssen aber die Aktionen des Agenten von der gesamten Wahrnehmungsfolge abhängen, muss sich der Agent die Wahrnehmungen merken.

Wir beschreiben die Agentenprogramme in der einfachen Pseudocode-Sprache, die in Anhang B erklärt ist. (Der online bereitgestellte Code enthält Implementierungen in echten Programmiersprachen.) Zum Beispiel zeigt ▶ Abbildung 2.7 ein relativ triviales Agentenprogramm, das eine Wahrnehmungsfolge verfolgt und sie dann als Index auf eine Tabelle mit Aktionen verwendet, um zu entscheiden, was getan werden soll. Die Tabelle – ein Beispiel ist in Abbildung 2.3 für die Staubsaugerwelt angegeben – repräsentiert explizit die Agentenfunktion, die das Agentenprogramm verkörpert. Um auf diese Weise einen rationalen Agenten zu erstellen, müssen wir als Entwickler eine Tabelle aufbauen, die für jede mögliche Wahrnehmungsfolge eine geeignete Aktion enthält.

```
function TABLE-DRIVEN-AGENT(percept) returns eine Aktion
    persistent: percepts, eine Sequenz, anfangs leer
                table, eine Tabelle mit Aktionen, durch Wahrnehmungs-
                folgen indiziert, anfangs vollständig spezifiziert

    percept an das Ende von percepts anfügen
    action ← LOOKUP(percepts, table)
return action
```

Abbildung 2.7: Das Programm TABLE-DRIVEN-AGENT wird für jede neue Wahrnehmung aufgerufen und gibt jeweils eine Aktion zurück. Es verwaltet die vollständige Wahrnehmungsfolge im Hauptspeicher.

5 Es gibt auch andere Möglichkeiten, Agentenprogramme aufzubauen; beispielsweise könnten wir die Agentenprogramme als Coroutinen gestalten, die asynchron zur Umgebung ausgeführt werden. Jede diese Coroutinen hat einen Eingangs- und Ausgangs-Port und besteht aus einer Schleife, die Wahrnehmungen vom Eingabe-Port liest und Aktionen auf den Ausgabe-Port schreibt.

Die Überlegung, warum der tabellengesteuerte Ansatz für die Entwicklung von Agenten fehlschlagen muss, ist sehr aufschlussreich. Es sei P die Menge möglicher Wahrnehmungen und T die Lebensdauer des Agenten (die Gesamtzahl der Wahrnehmungen, die er entgegennimmt). Die Nachschlagetabelle enthält

$$\sum_{t=1}^{T} |P|^t$$

Einträge. Betrachten Sie das automatisierte Taxi: Die visuelle Eingabe von einer einzelnen Kamera trifft mit einer Geschwindigkeit von etwa 27 MB pro Sekunde (30 Bilder pro Sekunde, 640 x 480 Pixel mit 24 Bit Farbinformation) ein. Damit erhalten wir eine Nachschlagetabelle mit mehr als $10^{250.000.000.000}$ Einträgen für eine Stunde Fahrt. Selbst die Nachschlagetabelle für Schach – ein winziges, gutmütiges Fragment der realen Welt – hätte mindestens 10^{150} Einträge. Die erschreckende Größe dieser Tabellen (die Anzahl der Atome im beobachtbaren Universum beträgt weniger als 10^{80}) hat zur Folge, dass (a) kein physischer Agent dieser Welt über den Platz verfügt, die Tabelle zu speichern, (b) die Entwickler nicht die Zeit hätten, die Tabelle zu erstellen, (c) kein Agent je alle wichtigen Tabelleneinträge aus seiner Erfahrung lernen könnte und (d) selbst wenn die Umgebung einfach genug ist, um eine sinnvolle Tabellengröße zu erzeugen, der Entwickler immer noch keine Anhaltspunkte darüber hat, wie er die Tabelleneinträge füllen soll.

Trotzdem erledigt das Programm TABLE-DRIVEN-AGENT *genau das*, was wir wollen: Es implementiert die gewünschte Agentenfunktion. Die wichtigste Herausforderung der KI ist es, herauszufinden, wie man Programme schreibt, die rationales Verhalten so weit wie möglich aus einem vergleichsweise kleinen Programm statt aus einer riesigen Tabelle erzeugen. Wie zahlreiche Beispiele zeigen, lässt sich dies in anderen Bereichen erfolgreich realisieren: So wurden die riesigen Quadratwurzeltabellen, mit denen Ingenieure und Schulkinder vor 1970 gearbeitet haben, jetzt durch ein fünfzeiliges Programm ersetzt, das die Methode von Newton implementiert und auf Taschenrechnern ausgeführt wird. Die Frage lautet: Kann die KI für allgemeines intelligentes Verhalten das leisten, was Newton für die Quadratwurzel getan hat? Wir glauben daran.

```
function REFLEX-VACUUM-AGENT([location, status]) returns eine Aktion

    if status = Schmutzig then return Saugen
    else if location = A then return Rechts
    else if location = B then return Links
```

Abbildung 2.8: Das Agentenprogramm für einen einfachen Reflexagenten in der Staubsaugerumgebung mit zwei Zuständen. Dieses Programm implementiert die in Abbildung 2.3 tabellarisch dargestellte Agentenfunktion.

Im restlichen Abschnitt skizzieren wir vier grundlegende Agentenprogramme, die die Prinzipien verkörpern, auf denen die meisten intelligenten Systeme aufbauen:

- Einfache Reflexagenten
- Modellbasierte Reflexagenten
- Zielbasierte Agenten
- Nutzenbasierte Agenten

Jede Art von Agentenprogramm kombiniert spezielle Komponenten in besonderer Art und Weise, um Aktionen zu generieren. *Abschnitt 2.4.6* erläutert in allgemeinen Begriffen, wie sich alle diese Agenten in lernende Agenten umwandeln lassen, die die Leistung ihrer Komponenten verbessern können, um bessere Aktionen zu generieren.

2.4.2 Einfache Reflexagenten

Der einfachste Agent ist der **einfache Reflexagent**. Diese Agenten wählen Aktionen auf Grundlage der *aktuellen* Wahrnehmung aus und ignorieren den restlichen Wahrnehmungsverlauf. Beispielsweise handelt es sich bei dem Staubsauger-Agenten, dessen Agentenfunktion in Abbildung 2.3 tabellarisch dargestellt ist, um einen einfachen Reflexagenten, weil er seine Entscheidung nur auf der aktuellen Position und darauf, ob diese Schmutz enthält, gründet. ▶ Abbildung 2.8 zeigt ein Agentenprogramm für einen derartigen Assistenten.

Beachten Sie, dass das Staubsauger-Agentenprogramm tatsächlich sehr klein im Vergleich zur entsprechenden Tabelle ist. Die Reduzierung ergibt sich vor allem daraus, dass der Wahrnehmungsverlauf ignoriert wird, wodurch die Anzahl der Möglichkeiten von 4^T auf lediglich 4 zurückgeht. Eine weitere, kleinere Reduzierung entsteht aus der Tatsache, dass die Aktion nicht von der Position abhängt, wenn das aktuelle Quadrat schmutzig ist.

Einfaches Reflexverhalten tritt sogar in komplexeren Umgebungen auf. Stellen Sie sich vor, Sie seien der Fahrer des automatisierten Taxis. Wenn das Auto vor Ihnen bremst und seine Bremslichter aufleuchten, sollten Sie dies erkennen und ebenfalls bremsen. Mit anderen Worten erfolgt auf die visuelle Eingabe hin eine bestimmte Verarbeitung, um die Bedingung einzurichten, die wir als „das Auto vor uns bremst“ bezeichnen. Dies löst dann eine bestimmte eingerichtete Verbindung im Agentenprogramm zur Aktion „bremsen beginnen“ aus. Eine derartige Verbindung heißt Bedingungs-Aktions-Regel,[6] dargestellt als

if *auto-vor-uns-bremst* ***then*** *bremsen-beginnen*

Menschen haben viele solcher Verbindungen, wobei es sich zum Teil um erlernte Antworten (beispielsweise beim Fahren), zum Teil um angeborene Reflexe (wie etwa das Blinzeln, wenn sich etwas den Augen nähert) handelt. In diesem Buch werden Sie verschiedene Möglichkeiten kennenlernen, wie solche Verbindungen erlernt und implementiert werden können.

Das Programm in Abbildung 2.8 ist spezifisch für eine bestimmte Staubsaugerumgebung. In einem unabhängigeren und flexibleren Ansatz erstellt man zuerst einen universellen Interpreter für Bedingungs-Aktions-Regeln und legt dann Regelmengen für spezifische Aufgabenumgebungen fest. ▶ Abbildung 2.9 zeigt die Struktur dieses allgemeinen Programms in schematischer Form und wie die Bedingungs-Aktions-Regeln es dem Agenten erlauben, die Verbindung von der Wahrnehmung zur Aktion vorzunehmen. (Stoßen Sie sich nicht daran, wenn dies trivial erscheint; es wird gleich interessanter.) Wir verwenden Rechtecke, um den aktuellen internen Zustand des Entscheidungsprozesses im Agenten zu kennzeichnen, und Ovale, um die Hintergrundinformationen darzustellen, die in diesem Prozess verwendet werden. ▶ Abbildung 2.10 zeigt das Agen-

6 Auch als **Situations-Aktions-Regeln**, **Produktionen** oder **if-then**-Regeln bezeichnet.

tenprogramm, das ebenfalls sehr einfach ist. Die Funktion INTERPRET-INPUT erzeugt eine abstrahierte Beschreibung des aktuellen Zustandes aus der Wahrnehmung und die Funktion RULE-MATCH gibt die erste Regel in der Regelmenge zurück, die mit der vorgegebenen Zustandsbeschreibung übereinstimmt. Beachten Sie, dass die Beschreibung im Hinblick auf „Regeln“ und „Übereinstimmung“ rein konzeptuell ist; echte Implementierungen können so einfach sein wie eine Menge von Logikgattern, die einen booleschen Schaltkreis implementieren.

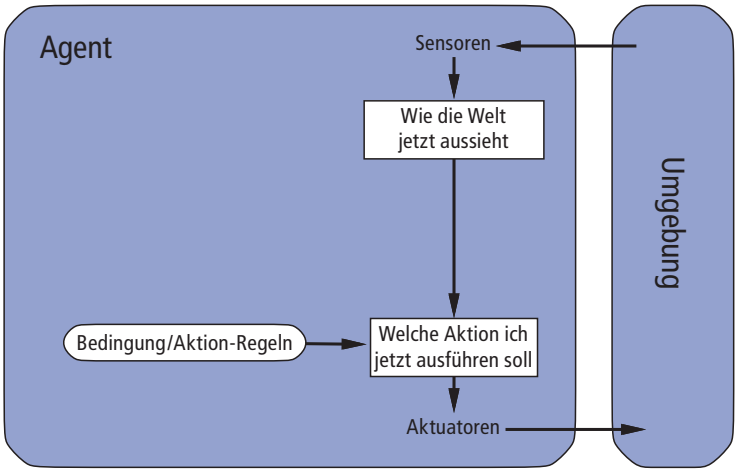


Abbildung 2.9: Schematische Darstellung eines einfachen Reflexagenten.

```
function SIMPLE_REFLEX-AGENT(percept) returns eine Aktion
    persistent: rules, eine Menge von Bedingungs-Aktions-Regeln

     state ← INTERPRET-INPUT(percept)
     rule ← RULE-MATCH(state, rules)
     action ← rule.ACTION
     return action
```

Abbildung 2.10: Ein einfacher Reflexagent. Er handelt entsprechend einer Regel, deren Bedingungen mit dem durch die Wahrnehmung definierten aktuellen Zustand übereinstimmen.

Tipp

Einfache Reflexagenten haben die vortreffliche Eigenschaft, dass sie einfach sind. Allerdings sind sie auch nur sehr begrenzt intelligent. Der Agent in Abbildung 2.10 *funktioniert nur dann, wenn die korrekte Entscheidung auf der Grundlage nur der aktuellen Wahrnehmung getroffen werden kann – d.h., nur wenn die Umgebung vollständig beobachtbar ist.* Selbst ein bisschen Unbeobachtbarkeit kann ernsthafte Probleme verursachen. Beispielsweise setzt die zuvor beschriebene Bremsregel voraus, dass die Bedingung *Auto-vor-uns-bremst* aus der aktuellen Wahrnehmung – einem einzigen Videobild – erkannt werden kann. Das funktioniert, wenn das vorausfahrende Auto ein mittig platziertes Bremslicht besitzt. Leider haben ältere Automodelle andere Anordnungen der Rücklichter, Bremslichter und Blinker und es ist nicht immer möglich, anhand eines einzigen Bildes zu entscheiden, ob das Auto bremst. Ein einfacher Reflexagent, der hinter einem solchen Auto herfährt, würde entweder ständig bremsen, unnötigerweise bremsen oder im ungünstigsten Fall überhaupt nicht bremsen.

Ein ähnliches Problem kann in der Staubsaugerwelt auftauchen. Angenommen, ein einfacher Staubsauger-Reflexagent verliert seinen Positionssensor und verfügt nur noch über einen Schmutzsensor. Ein solcher Agent hat nur zwei mögliche Wahrnehmungen: [*Schmutzig*] und [*Sauber*]. Er kann *saugen* als Reaktion auf [*Schmutzig*]; aber was soll er für [*Sauber*] tun? Eine Bewegung nach *Links* schlägt (immer) fehl, wenn er in Quadrat *A* anfängt, und eine Bewegung nach *Rechts* schlägt (immer) fehl, wenn er in Quadrat *B* anfängt. Für einfache Reflexagenten in teilweise beobachtbaren Umgebungen sind Endlosschleifen häufig unvermeidbar.

Ein Entkommen aus der Endlosschleife ist möglich, wenn der Agent seine Aktionen **zufällig auswählen** kann. Nimmt der Staubsauger-Agent beispielsweise [*Sauber*] wahr, könnte er eine Münze werfen, um zwischen *Links* und *Rechts* auszuwählen. Man kann leicht zeigen, dass der Agent das andere Quadrat in durchschnittlich zwei Schritten erreicht. Ist dieses Quadrat schmutzig, säubert er es und die Säuberungsarbeit ist abgeschlossen. Ein einfacher Reflexagent mit Zufallsauswahl könnte also einem einfachen deterministischen Reflexagenten überlegen sein.

In *Abschnitt 2.3* haben wir erwähnt, dass zufälliges Verhalten der richtigen Art in einigen Multiagentenumgebungen rational sein kann. In Einzelagentenumgebungen ist die Zufälligkeit normalerweise nicht rational. Es ist ein praktischer Trick, der einem einfachen Reflexagenten in einigen Situationen hilft, aber größtenteils sind wir besser beraten, wenn wir komplexere deterministische Agenten verwenden.

2.4.3 Modellbasierte Reflexagenten

Ein Agent kommt mit einer teilweisen Beobachtbarkeit am besten zurecht, wenn er den für ihn momentan nicht sichtbaren Teil der Welt verfolgt. Das heißt, der Agent sollte eine Art **internen Zustand** führen, der vom Wahrnehmungsverlauf abhängig ist und damit zumindest einen Teil der nicht beobachteten Aspekte des aktuellen Zustandes widerspiegelt. Für das Bremsproblem ist der interne Zustand nicht allzu umfangreich – man braucht nur das vorhergehende Bild aus der Kamera, sodass der Agent erkennen kann, wenn zwei rote Lichter am hinteren Ende des Fahrzeuges gleichzeitig an- oder ausgehen. Für andere Fahraufgaben, wie beispielsweise einen Spurwechsel, muss sich der Agent merken, wo sich andere Autos befinden, wenn er sie nicht alle gleichzeitig sehen kann. Und damit das Fahren überhaupt möglich ist, muss der Agent verfolgen, wo seine Schlüssel sind.

Die Aktualisierung dieser internen Zustandsinformationen über die Zeit erfordert es, zwei Arten von Wissen im Agentenprogramm zu kodieren. Erstens brauchen wir Informationen darüber, wie sich die Welt unabhängig vom Agenten weiterentwickelt – beispielsweise, dass ein überholendes Auto im Allgemeinen näher ist, als es noch vor einem Moment war. Zweitens brauchen wir Informationen darüber, wie sich die eigenen Aktionen des Agenten auf die Welt auswirken – wenn beispielsweise der Agent das Lenkrad im Uhrzeigersinn dreht, fährt das Auto nach rechts, oder wenn man fünf Minuten lang mit 120 km/h auf der Autobahn nach Norden gefahren ist, befindet man sich normalerweise zehn Kilometer nördlich von dem Punkt, an dem man noch vor fünf Minuten war. Dieses Wissen darüber, „wie die Welt funktioniert“ – egal, ob in einfachen booleschen Schaltungen oder in vollständigen wissenschaftlichen Theorien implementiert –, wird als **Modell** der Welt bezeichnet. Ein Agent, der ein solches Modell verwendet, heißt **modellbasierter Agent**.

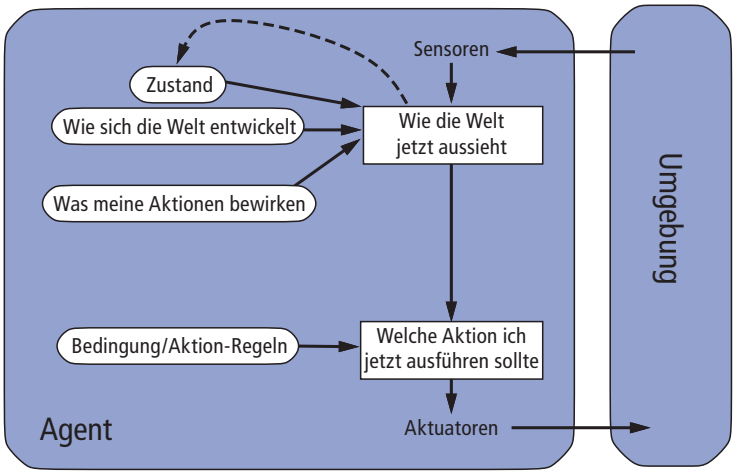


Abbildung 2.11: Ein modellbasierter Reflexagent.

▶ Abbildung 2.11 zeigt die Struktur des Reflexagenten mit internem Zustand und verdeutlicht, wie die aktuelle Wahrnehmung mit dem früheren internen Zustand verknüpft wird, um die aktualisierte Beschreibung des aktuellen Zustandes basierend auf dem Modell des Agenten über die Funktionsweise der Welt zu erzeugen. ▶ Abbildung 2.12 zeigt das Agentenprogramm. Der interessante Teil ist die Funktion UPDATE-STATE, die dafür verantwortlich ist, die neue interne Zustandsbeschreibung zu erstellen. Wie Modelle und Zustände im Detail dargestellt werden, hängt von der Art der Umgebung und der konkreten, im Agentenentwurf verwendeten Technologie ab. Detaillierte Beispiele für Modelle und Aktualisierungsalgorithmen finden Sie in den *Kapiteln 4*, *12*, *11*, *15*, *17* und *25*.

```
function MODEL-BASED-REFLEX-AGENT(percept) returns eine Aktion
  persistent: state, die aktuelle Vorstellung des Agenten vom Zustand der Welt
              model , eine Beschreibung, wie der nächste Zustand vom
              aktuellen Zustand und der Aktion abhängt
              rules, eine Menge von Bedingungs-Aktions-Regeln
              action, die vorherige Aktion, anfangs keine

  state ← UPDATE-STATE(state, action, percept, model)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

Abbildung 2.12: Ein modellbasierter Reflexagent. Er verwaltet den aktuellen Zustand der Welt mithilfe eines internen Modells. Anschließend wählt er eine Aktion auf dieselbe Weise wie der Reflexagent aus.

Unabhängig von der Art der verwendeten Darstellung kann der Agent nur selten den aktuellen Zustand einer teilweise beobachtbaren Umgebung *genau* bestimmen. Stattdessen repräsentiert der Kasten mit der Beschriftung „Wie die Welt jetzt aussieht“ (siehe Abbildung 2.11) die „beste Annahme“ (oder manchmal beste Annahmen) des Agenten. Zum Beispiel ist ein automatisiertes Taxi eventuell nicht in der Lage, um den vor ihm haltenden großen Truck herumzublicken, und kann nur vermuten, was den Stau verursacht. Somit muss der Agent trotz einer unvermeidbaren Unbestimmtheit im aktuellen Zustand eine Entscheidung treffen.

Vielleicht weniger offensichtlich in Bezug auf den internen „Zustand“, der von einem modellbasierten Agenten verwaltet wird, ist es, dass der Agent nicht im wörtlichen Sinn beschreiben muss, „wie die Welt jetzt aussieht“. Wenn zum Beispiel das Taxi nach Hause fährt, könnte eine Regel verlangen, dass getankt wird, sofern der Tank nicht wenigstens halb voll ist. Obwohl „nach Hause fahren“ ein Aspekt des Weltzustandes zu sein scheint, ist das Ziel des Taxis tatsächlich ein Aspekt des internen Zustandes des Agenten. Wenn Sie sich darüber wundern, betrachten Sie einfach den Fall, dass sich das Taxi an genau demselben Platz zur selben Zeit befindet, aber ein anderes Ziel ansteuern möchte.

2.4.4 Zielbasierte Agenten

Es genügt nicht immer, etwas über den aktuellen Zustand der Umgebung zu wissen, um zu entscheiden, was zu tun ist. Beispielsweise kann das Taxi an einer Straßenkreuzung links abbiegen, rechts abbiegen oder geradeaus weiterfahren. Die richtige Entscheidung ist davon abhängig, wohin das Taxi fahren will. Mit anderen Worten benötigt der Agent neben der aktuellen Zustandsbeschreibung auch eine Art **Ziel**information, die wünschenswerte Situationen beschreibt – beispielsweise am Fahrtziel des Fahrgastes anzukommen. Das Agentenprogramm kann dies mit dem Modell (den gleichen Informationen, die im modellbasierten Reflexagenten verwendet wurden) kombinieren, um Aktionen auszuwählen, die das Ziel erreichen. ▶ Abbildung 2.13 zeigt die Struktur eines zielbasierten Agenten.

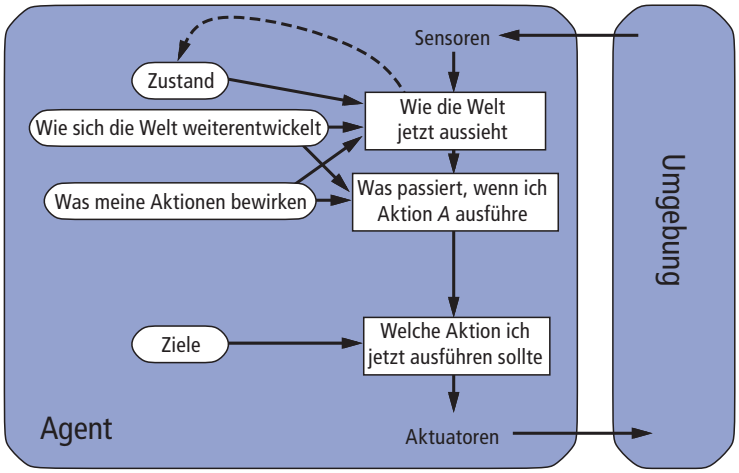


Abbildung 2.13: Ein modellbasierter, zielbasierter Agent. Er verwaltet den Zustand der Welt sowie eine Menge von Zielen, die er zu erreichen versucht, und wählt eine Aktion aus, die (im Endeffekt) zur Erreichung dieser Ziele führt.

Manchmal ist die zielbasierte Aktionsauswahl einfach – etwa wenn die Zielerreichung direkt aus einer einzigen Aktion resultiert. Manchmal ist sie komplizierter – zum Beispiel wenn der Agent lange Folgen von Drehungen und Wendungen betrachten muss, um das Ziel zu erreichen. **Suchen** (*Kapitel 3* bis *5*) und **Planen** (*Kapitel 10* und *11*) sind Teilbereiche der KI, die der Aufgabe gewidmet sind, Aktionsfolgen zu finden, die die Ziele des Agenten erreichen.

Beachten Sie, dass sich diese Art der Entscheidungsfindung grundsätzlich von den zuvor beschriebenen Bedingungs-Aktions-Regeln unterscheidet, weil dabei die Zukunft berück-

sichtigt wird – sowohl „Was passiert, wenn ich dieses oder jenes mache?“ als auch „Macht mich das glücklich?“. Im Entwurf von Reflexagenten wird diese Information nicht explizit dargestellt, weil die eingebauten Regeln direkt von Wahrnehmungen auf Aktionen abbilden. Der Reflexagent bremst, wenn er Bremslichter sieht. Ein zielbasierter Agent könnte im Prinzip schlussfolgern, dass das Auto vor ihm langsamer wird, wenn er seine Bremslichter sieht. Berücksichtigt man, wie sich die Welt normalerweise entwickelt, ist Bremsen die einzige Aktion, die das Ziel erreicht, nicht auf das andere Auto aufzufahren.

Obwohl der zielbasierte Agent ineffizienter erscheint, ist er sehr viel flexibler, weil das Wissen, das seine Entscheidungen unterstützt, explizit dargestellt wird und verändert werden kann. Wenn es anfängt zu regnen, kann der Agent sein Wissen darüber aktualisieren, wie effektiv seine Bremsen wirken; das bewirkt automatisch, dass alle relevanten Verhaltensmuster angepasst werden, sodass sie den neuen Bedingungen entsprechen. Für den Reflexagenten dagegen müssten wir viele Bedingungs-Aktions-Regeln neu formulieren. Das Verhalten des zielbasierten Agenten kann einfach geändert werden, sodass er eine andere Position ansteuert, indem diese Position als das Ziel spezifiziert wird. Die Regeln des Reflexagenten, wann er abbiegen und wann er geradeaus weiterfahren soll, funktionieren nur für ein einziges Ziel; sie müssen alle ersetzt werden, wenn er woanders hin fahren soll.

2.4.5 Nutzenbasierte Agenten

Ziele allein sind nicht ausreichend, um in den meisten Umgebungen ein hochqualitatives Verhalten zu erzeugen. Beispielsweise gibt es viele Aktionsfolgen, die das Taxi an sein Fahrtziel bringen (und damit das Ziel erreichen), aber einige sind schneller, sicherer, zuverlässiger oder billiger als andere. Ziele stellen nur eine grobe binäre Unterscheidung zwischen den Zuständen „glücklich“ und „unglücklich“ dar. Eine allgemeinere Leistungsbewertung sollte einen Vergleich verschiedener Zustände der Welt erlauben, und zwar genau danach, wie glücklich sie den Agenten machen würden. Da sich „glücklich“ nicht sehr wissenschaftlich anhört, verwenden Wirtschaftswissenschaftler und Informatiker stattdessen den Begriff **Nutzen**.

Wie bereits erläutert, weist eine Leistungsbewertung einer gegebenen Sequenz von Umgebungszuständen eine Punktzahl zu, sodass sich leicht unterscheiden lässt zwischen mehr oder weniger wünschenswerten Wegen, zum Ziel des Taxis zu gelangen. Die **Nutzenfunktion** eines Agenten ist praktisch eine Internalisierung der Leistungsbewertung. Wenn die interne Nutzenfunktion und die externe Leistungsbewertung übereinstimmen, ist ein Agent, der Aktionen auswählt, um seinen Nutzen zu maximieren, rational entsprechend der externen Leistungsbewertung.

Es sei noch einmal betont, dass dies nicht die *einzige* Möglichkeit ist, rational zu sein – Sie haben bereits ein rationales Agentenprogramm für die Staubsaugerwelt (Abbildung 2.8) kennengelernt, das keine Vorstellung von seiner Nutzenfunktion hat –, doch wie bei zielbasierten Agenten besitzt ein nutzenbasierter Agent viele Vorteile hinsichtlich Flexibilität und Lernen. Darüber hinaus sind Ziele in zwei Arten von Fällen unzulänglich, wohingegen ein nutzenbasierter Agent trotzdem rationale Entscheidungen treffen kann. Erstens spezifiziert die Nutzenfunktion einen passenden Kompromiss, wenn es zueinander in Konflikt stehende Ziele gibt, die nur zum Teil erreicht werden können (zum Beispiel Geschwindigkeit und Sicherheit). Zweitens bietet der Nutzen eine Möglichkeit, die Wahrscheinlichkeit eines Erfolges gegen die Wichtigkeit der Ziele abzuschätzen.

In der realen Welt sind partielle Beobachtbarkeit und stochastisches Verhalten allgegenwärtig. Das Treffen von Entscheidungen geschieht somit unter Unbestimmtheit. Aus technischer Sicht wählt ein rationaler nutzenbasierter Agent die Aktion aus, die den **erwarteten Nutzen** der Aktionsergebnisse maximiert – d.h. den Nutzen, den sich der Agent bei den gegebenen Wahrscheinlichkeiten und Nutzenwerten jedes Ergebnisses im Durchschnitt verspricht. (Anhang A definiert Erwartung genauer.) In *Kapitel 16* zeigen wir, dass sich jeder rationale Agent so verhalten muss, als besäße er eine Nutzenfunktion, deren erwarteten Wert er zu maximieren versucht. Ein Agent, der eine explizite Nutzenfunktion besitzt, kann mit einem universellen Algorithmus, der nicht von der spezifischen zu maximierenden Nutzenfunktion abhängig ist, rationale Entscheidungen treffen. Auf diese Weise wird die „globale" Definition der Rationalität – in der die Agentenfunktionen mit der höchsten Leistung als rational bezeichnet werden – in eine „lokale" Einschränkung für den Entwurf rationaler Agenten umgewandelt, die in einem einfachen Programm ausgedrückt werden kann.

▶ Abbildung 2.14 zeigt die Struktur eines nutzenbasierten Agenten. Nutzenbasierte Agentenprogramme werden in Teil 4 behandelt, wo wir Entscheidungen treffende Agenten entwerfen, die mit der Unsicherheit zurechtkommen, die stochastischen oder teilweise beobachtbaren Umgebungen eigen ist.

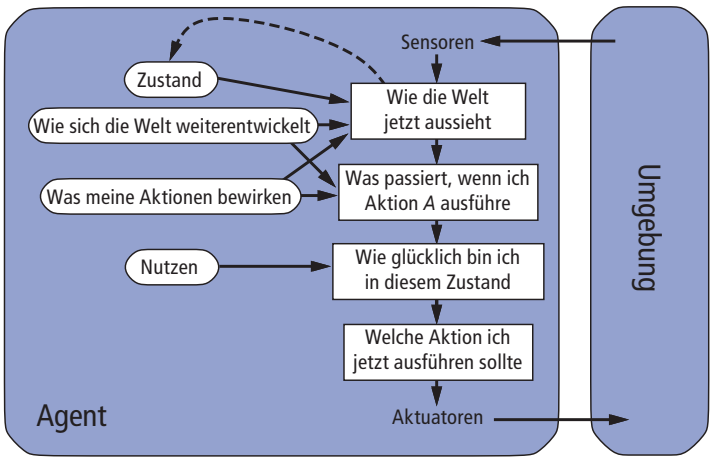


Abbildung 2.14: Ein modellbasierter, nutzenbasierter Agent. Er verwendet ein Modell der Welt zusammen mit einer Nutzenfunktion, die seine Vorlieben in Hinblick auf die verschiedenen Zustände der Welt bewertet. Anschließend wählt er die Aktion aus, die zum besten erwarteten Nutzen führt, wobei der erwartete Nutzen ermittelt wird, indem ein Durchschnitt aller möglichen Ergebniszustände berechnet wird, gewichtet nach der Wahrscheinlichkeit ihres Eintreffens.

Vielleicht fragen Sie sich jetzt: „Ist das wirklich so simpel? Wir erstellen einfach Agenten, die erwarteten Nutzen maximieren, und schon sind wir fertig?" Es stimmt, dass derartige Agenten intelligent sind, doch ist das nicht so einfach. Ein nutzenbasierter Agent muss seine Umgebung modellieren und verfolgen. Diese Aufgaben setzen umfangreiche Forschungen zum Wahrnehmen, Darstellen, Schlussfolgern und Lernen voraus. Die Ergebnisse dieser Forschungen füllen viele Kapitel dieses Buches. Eine ebenso schwierige Aufgabe ist es, das den Nutzen maximierende Vorgehen auszuwählen, was ausgeklügelte Algorithmen voraussetzt, die mehrere weitere Kapitel füllen. Wie *Kapitel 1* erwähnt hat, ist selbst mit diesen Algorithmen aufgrund der Berechnungskomplexität in der Praxis keine perfekte Rationalität erreichbar.

2.4.6 Lernende Agenten

Wir haben Agentenprogramme beschrieben, die Aktionen nach verschiedenen Methoden auswählen. Allerdings haben wir bisher noch nicht erklärt, wie die Agentenprogramme *entstehen*. In seinem berühmten frühen Aufsatz betrachtet Turing (1950) die Idee, seine intelligenten Maschinen tatsächlich manuell zu programmieren. Er schätzt, wie viel Arbeit das bedeuten würde, und kommt zu dem Schluss: „Eine schnellere Methode scheint wünschenswert". Als Methode schlägt er vor, lernende Maschinen zu erstellen und diese dann zu lehren. In vielen Bereichen der KI ist dies heute die bevorzugte Art, Systeme auf dem neuesten Stand der Technik zu erstellen. Wie bereits weiter vorn erwähnt, hat das Lernen noch einen weiteren Vorteil: Es erlaubt dem Agenten, in zunächst unbekannten Umgebungen zu arbeiten und kompetenter zu werden, als sein Ausgangswissen es erlauben würde. In diesem Abschnitt führen wir kurz die wichtigsten Konzepte lernender Agenten ein. In fast jedem Kapitel des Buches werden wir etwas zu Chancen und Methoden für das Lernen für bestimmte Arten von Agenten sagen. Teil 5 beschäftigt sich ausführlicher mit den verschiedenen Lernalgorithmen an sich.

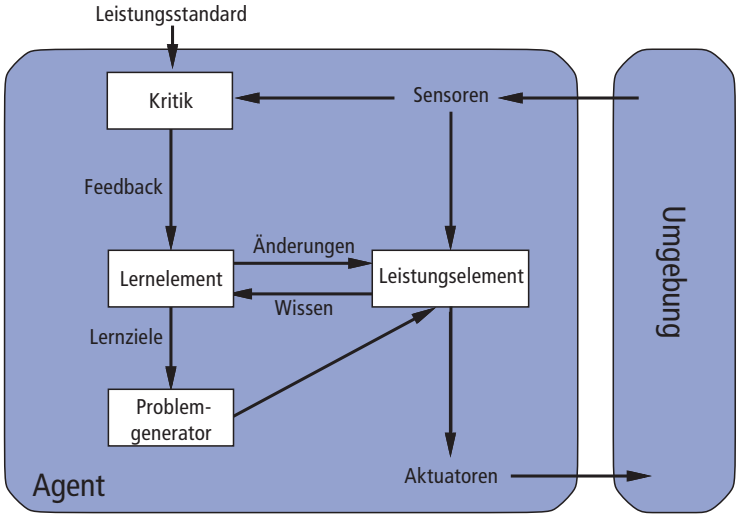


Abbildung 2.15: Ein allgemeiner lernender Agent.

Ein lernender Agent kann in vier konzeptuelle Komponenten unterteilt werden, wie in ▶ Abbildung 2.15 gezeigt. Die wichtigste Unterscheidung besteht zwischen dem **Lernelement**, das dafür verantwortlich ist, Verbesserungen zu erzielen, und dem **Leistungselement**, das für die Auswahl externer Aktionen verantwortlich ist. Das Leistungselement ist das, was wir zuvor als den ganzen Agenten betrachtet haben: Es nimmt Wahrnehmungen auf und entscheidet, welche Aktionen ausgeführt werden. Das Lernelement verwendet das Feedback aus der **Kritik** darüber, welche Ergebnisse der Agent erzielt, und entscheidet, ob das Leistungselement abgeändert werden soll, um in Zukunft bessere Ergebnisse zu erzielen.

Der Entwurf des Lernelements ist stark von dem Entwurf des Leistungselements abhängig. Wenn wir versuchen, einen Agenten zu entwerfen, der eine bestimmte Fertigkeit erlernt, ist die erste Frage nicht: „Wie schaffe ich es, dies zu lernen?", sondern: „Welche Art Leistungselement braucht mein Agent, um dies zu tun, nachdem er gelernt hat?" Für

einen Agentenentwurf können Lernmechanismen konstruiert werden, um jeden Teil des Agenten zu verbessern.

Die Kritik teilt dem Lernelement mit, wie gut sich der Agent im Hinblick auf einen festgelegten Leistungsstandard verhält. Die Kritik ist notwendig, weil die Wahrnehmungen selbst keinen Hinweis auf den Erfolg des Agenten bieten. Beispielsweise könnte ein Schachprogramm eine Wahrnehmung erhalten, die darauf hinweist, dass es seinen Gegner schachmatt gesetzt hat, aber es ist ein Leistungsstandard erforderlich, um zu wissen, ob das gut ist; die Wahrnehmungswelt sagt nichts darüber aus. Es ist wichtig, dass der Leistungsstandard festgelegt ist. Vom Konzept her kann man ihn sich als völlig außerhalb des Agenten befindlich vorstellen, weil der Agent ihn nicht verändern muss, um sein eigenes Verhalten anzupassen.

Die letzte Komponente des lernenden Agenten ist der **Problemgenerator**. Er ist dafür verantwortlich, Aktionen vorzuschlagen, die zu neuen und informativen Erfahrungen führen. Könnte das Leistungselement allein entscheiden, würde es immer die Aktionen ausführen, die seinem Wissen nach am besten sind. Wenn der Agent jedoch ein wenig experimentierfreudig wäre und einige vielleicht auf kurze Sicht suboptimale Aktionen ausführen würde, könnte er eventuell auf lange Sicht wesentlich bessere Aktionen entdecken. Die Aufgabe des Problemgenerators ist es, diese sondierenden Aktionen vorzuschlagen. So verhalten sich Wissenschaftler, wenn sie experimentieren. Galilei dachte nicht, dass es an sich wertvoll sei, Steine von einem Turm in Pisa zu werfen. Er versuchte nicht, die Steine zu zersplittern, und auch nicht, die Gehirne zufällig anwesender Passanten zu deformieren. Er zielte darauf ab, seinen eigenen Verstand zu verändern, indem er eine bessere Theorie für die Bewegung von Objekten ausarbeitete.

Um den Gesamtentwurf zu konkretisieren, wollen wir zum Beispiel des automatisierten Taxis zurückkommen. Das Leistungselement besteht aus einer Sammlung von Wissen und Prozeduren, die dem Taxi zur Auswahl seiner Fahrtaktionen zur Verfügung stehen. Das Taxi gelangt auf die Straße und fährt unter Verwendung dieses Leistungselementes. Die Kritik beobachtet die Welt und gibt Informationen an das Lernelement weiter. Nachdem das Taxi beispielsweise schnell über drei Spuren hinweg links abgebogen ist, beobachtet die Kritik die Beschimpfungen durch andere Fahrer. Aus dieser Erfahrung kann das Lernelement eine Regel formulieren, die besagt, dass dies eine schlechte Aktion war, und das Leistungselement wird verändert, indem die neue Regel installiert wird. Der Problemgenerator könnte bestimmte Verhaltensbereiche identifizieren, die verbessert werden müssen, und Experimente vorschlagen, wie beispielsweise das Ausprobieren von Bremsen auf unterschiedlichen Straßenbelägen unter unterschiedlichen Bedingungen.

Das Lernelement kann Änderungen an jeder der in den Agentenskizzen (Abbildung 2.9, Abbildung 2.11, Abbildung 2.13 und Abbildung 2.14) gezeigten „Wissens“-Komponenten vornehmen. In den einfachsten Fällen wird direkt aus der Wahrnehmungsfolge gelernt. Die Beobachtung von Paaren aufeinanderfolgender Zustände der Umgebung könnte es dem Agenten erlauben zu lernen, „wie sich die Welt entwickelt“, und die Beobachtung der Ergebnisse seiner Aktionen könnte es dem Agenten erlauben zu lernen, „was seine Aktionen bewirken“. Führt das Taxi beispielsweise einen bestimmten Bremsdruck aus, während es auf einer nassen Straße fährt, wird es schnell herausfinden, wie viel Bremswirkung tatsächlich erzielt wird. Offensichtlich sind diese beiden Lernaufgaben schwieriger, wenn die Umgebung nur teilweise beobachtbar ist.

Die im obigen Abschnitt beschriebenen Formen des Lernens müssen nicht auf den externen Leistungsstandard zugreifen – in gewisser Hinsicht ist dies der universelle Standard, um Voraussagen zu treffen, die mit dem Experiment übereinstimmen. Für einen nutzenbasierten Agenten, der Nutzeninformationen lernen will, ist die Situation etwas komplizierter. Stellen Sie sich beispielsweise vor, der Taxifahrer-Agent erhält keinerlei Trinkgeld von den Fahrgästen, die während der Fahrt völlig durchgeschüttelt wurden. Der externe Leistungsstandard muss den Agenten darüber informieren, dass der Verlust von Trinkgeldern ein negativer Beitrag zu seiner Gesamtleistung ist. Auf diese Weise könnte der Agent lernen, dass aggressive Manöver nicht zu seinem eigenen Nutzen beitragen. In gewisser Weise identifiziert der Leistungsstandard einen Teil der eintreffenden Wahrnehmung als **Belohnung** (oder **Strafe**), die ein direktes Feedback zur Qualität des Verhaltens des Agenten bereitstellt. Fest „verdrahtete" Leistungsstandards, wie beispielsweise Angst und Hunger bei Tieren, können auf diese Weise verstanden werden. Dieses Thema wird in *Kapitel 21* weiter vertieft.

Insgesamt gesehen bestehen Agenten aus einer Vielzahl von Komponenten und diese Komponenten können innerhalb des eigenen Programms auf vielerlei Arten dargestellt werden, sodass es eine große Vielfalt an Lernmethoden zu geben scheint. Allerdings ist ein übergreifender Aspekt vorhanden. Das Lernen in intelligenten Agenten lässt sich als Prozess der Veränderung jeder einzelnen Komponente des Agenten zusammenfassen, um die Komponenten besser mit der verfügbaren Feedback-Information abzustimmen und damit die Gesamtleistung des Agenten zu verbessern.

2.4.7 Wie die Komponenten von Agentenprogrammen funktionieren

Entsprechend der bisher (von recht hoher Warte) gegebenen Beschreibung bestehen Agentenprogramme aus verschiedenen Komponenten, deren Funktion es ist, Fragen zu stellen wie zum Beispiel: „Wie sieht die Welt jetzt aus?", „Welche Aktion soll ich jetzt ausführen?" und „Was bewirken meine Aktionen?". Für einen KI-Studenten lautet die nächste Frage: „Wie in aller Welt funktionieren diese Komponenten?" Es sind noch rund tausend Seiten notwendig, um diese Frage ordnungsgemäß zu beantworten. Zunächst aber wollen wir die Aufmerksamkeit des Lesers auf einige grundlegende Unterschiede zwischen den verschiedenen Möglichkeiten lenken, wie die Komponenten die vom Agenten bewohnte Umgebung darstellen können.

Grob gesagt können wir die Darstellungen entlang einer Achse zunehmender Komplexität und Ausdrucksfähigkeit anordnen – **atomar**, **faktorisiert** und **strukturiert**. Diese Konzepte lassen sich am besten anhand einer bestimmten Agentenkomponente veranschaulichen, beispielsweise einer Komponente, die sich mit „Was meine Aktionen bewirken" befasst. Diese Komponente beschreibt die Änderungen, die in der Umgebung als Ergebnis einer ausgeführten Aktion auftreten können. ▶ Abbildung 2.16 zeigt schematische Darstellungen, wie sich diese Übergänge darstellen lassen.

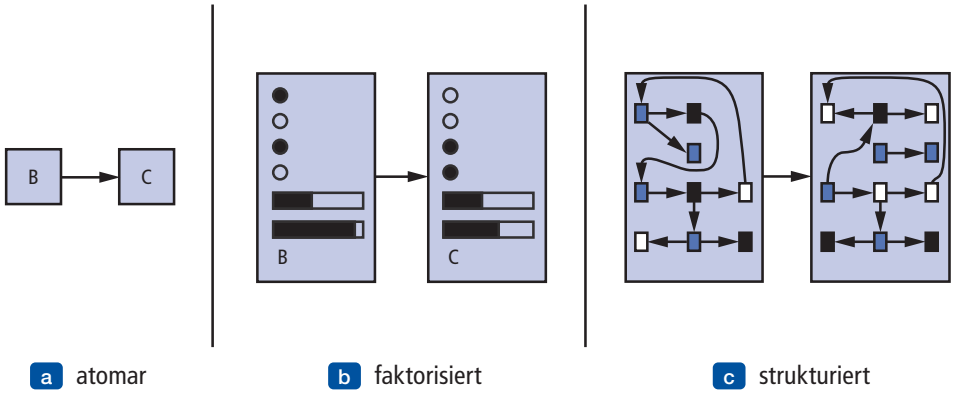


Abbildung 2.16: Drei Möglichkeiten, um Zustände und die Übergänge zwischen ihnen darzustellen. (a) Atomare Darstellung: Ein Zustand (wie zum Beispiel B oder C) ist eine Blackbox ohne innere Struktur; (b) Faktorisierte Darstellung: Ein Zustand besteht aus einem Vektor von Attributwerten – booleschen Werten, Gleitkommazahlen oder einem festen Satz von Symbolen; (c) Strukturierte Darstellung: Ein Zustand umfasst Objekte, die jeweils über eigene Attribute sowie Beziehungen zu anderen Objekten verfügen können.

In einer **atomaren Darstellung** ist jeder Zustand der Welt unteilbar – er besitzt keine interne Struktur. Sehen Sie sich das Problem an, eine Strecke von einem Ende des Landes über eine Folge von Städten zum anderen Ende zu fahren (auf dieses Problem kommen wir in Abbildung 3.2 zurück). Um dieses Problem zu lösen, kann es genügen, den Zustand der Welt auf lediglich den Namen der Stadt, in der wir uns befinden, zu reduzieren – ein einzelnes Wissensatom; eine „Blackbox", deren einzige erkennbare Eigenschaft darin besteht, zu einer anderen Blackbox identisch zu sein oder sich davon zu unterscheiden. Die Algorithmen, die der **Suche** und **Spielen** zugrunde liegen (*Kapitel 3* bis *5*), **Hidden-Markov-Modelle** (*Kapitel 15*) und **Markov-Entscheidungsprozesse** (*Kapitel 17*) arbeiten sämtlich mit atomaren Darstellungen – oder sie behandeln die Darstellungen zumindest so, als wären sie atomar.

Nehmen wir nun eine detailgetreuere Beschreibung für dasselbe Problem, wobei wir uns mit mehr als lediglich der atomaren Position in der einen oder einer anderen Stadt befassen müssen. Unter Umständen haben wir darauf zu achten, wie viel Kraftstoff im Tank ist, wie die aktuellen GPS-Koordinaten lauten, ob die Ölwarnlampe betriebsbereit ist, wie viel Kleingeld wir für Mautstraßen haben, welche Station im Autoradio eingestellt ist usw. Eine **faktorisierte Darstellung** teilt jeden Zustand in einen festen Satz von **Variablen** oder **Attributen** auf, die jeweils einen **Wert** haben können. Während zwei verschiedene atomare Zustände nichts gemeinsam haben – sie sind einfach unterschiedliche Blackboxes –, können zwei verschiedene faktorisierte Zustände bestimmte Attribute gemeinsam verwenden (beispielsweise an einer bestimmten GPS-Position sein) und andere nicht (beispielsweise genügend oder keinen Kraftstoff haben); damit lässt sich wesentlich einfacher herausarbeiten, wie ein Zustand in einen anderen überführt werden kann. Mit faktorisierten Darstellungen ist es auch möglich, Unbestimmtheit darzustellen – um zum Beispiel Unwissenheit bezüglich der Kraftstoffmenge im Tank auszudrücken, lässt man das Attribut leer. Viele wichtige Bereiche der KI basieren auf faktorisierten Darstellungen, einschließlich der Algorithmen für **Rand- und Nebenbedingungen** (*Kapitel 6*), **Aussagenlogik** (*Kapitel 7*), **Planung** (*Kapitel 10* und *11*), **Bayessche Netze** (*Kapitel 13* bis *16*) und die Algorithmen zum **maschinellen Lernen** in den *Kapiteln 18*, *20* und *21*.

Für viele Aufgaben müssen wir die Welt so verstehen, dass sie *Dinge* enthält, die miteinander in Beziehung stehen, und nicht einfach Variablen mit Werten. Zum Beispiel könnten wir feststellen, dass ein großer Truck vor uns rückwärts in die Einfahrt eines Milchbetriebes stößt, wo sich aber eine Kuh losgemacht hat und den Weg des Trucks versperrt. Eine faktorisierte Darstellung dürfte kaum mit dem Attribut *TruckAheadBackingIntoDairyFarmDrivewayBlockedByLooseCow* ausgestattet sein, das die Werte *true* oder *false* annehmen kann. Stattdessen brauchen wir eine **strukturierte Darstellung**, in der sich Objekte wie Kühe und LKWs sowie ihre verschiedenartigen und sich ändernden Beziehungen explizit beschreiben lassen (siehe Abbildung 2.16(c)). Strukturierte Darstellungen bilden die Grundlage von **relationalen Datenbanken** und **Logik erster Stufe** (*Kapitel 8*, *9* und *12*), **Wahrscheinlichkeitsmodellen erster Stufe** (*Kapitel 14*), **wissensbasiertem Lernen** (*Kapitel 19*) und einem großen Teil der **Spracherkennung** (*Kapitel 22* und *23*). Praktisch betrifft fast alles, was Menschen in natürlicher Sprache ausdrücken, Objekte und deren Beziehungen.

Wie bereits erwähnt, handelt es sich bei der Achse, auf der atomare, faktorisierte und strukturierte Darstellungen liegen, um die Achse zunehmender **Ausdrucksfähigkeit**. Grob gesagt kann eine ausdrucksfähigere Darstellung alles – zumindest so genau – erfassen wie eine weniger ausdrucksfähigere Darstellung und noch einiges mehr. Oftmals ist die ausdrucksfähigere Sprache *wesentlich* prägnanter; zum Beispiel lassen sich die Schachregeln in einer Sprache mit strukturierter Darstellung wie zum Beispiel Logik erster Ordnung auf einer oder zwei Seiten niederschreiben, während in einer Sprache mit faktorisierter Darstellung wie der Aussagenlogik Tausende von Seiten notwendig wären. Andererseits werden Schließen und Lernen komplizierter, wenn die Ausdruckskraft der Darstellung zunimmt. Um von den Vorzügen ausdrucksstarker Darstellungen zu profitieren und gleichzeitig deren Nachteile zu vermeiden, müssen intelligente Systeme für die reale Welt gegebenenfalls an allen Punkten entlang der Achse gleichzeitig arbeiten.

Bibliografische und historische Hinweise

Die zentrale Rolle der Aktionen in der Intelligenz – das Konzept praktischen Schließens – geht mindestens bis zur *Nikomachischen Ethik* von Aristoteles zurück. Praktisches Schließen war außerdem das Thema des einflussreichen Aufsatzes „Programs with Common Sense“ von McCarthy (1958). Die Gebiete der Robotik und Regelungstheorie sind der Natur nach grundsätzlich mit dem Aufbau physischer Agenten beschäftigt. Das Konzept eines **Controllers** in der Steuerungstheorie ist identisch mit dem eines Agenten in der KI. Es mag vielleicht überraschend sein, dass sich die KI während eines Großteils ihrer Geschichte mehr auf isolierte Komponenten von Agenten – Frage-Antwort-Systeme, Theorem-Beweiser, Visionssysteme usw. – als auf ganze Agenten konzentriert hat. Die Diskussion von Agenten im Buch von Genesereth und Nilsson (1987) war eine bemerkenswerte Ausnahme. Die ganzheitliche Sicht auf den Agenten ist heute in dem Bereich allgemein akzeptiert und zentrales Thema in neueren Büchern (Poole et al., 1998; Nilsson, 1998; Padgham und Winikoff, 2004; Jones, 2007).

Kapitel 1 hat die Wurzeln des Rationalitätskonzeptes in der Philosophie und Wirtschaft verfolgt. In der künstlichen Intelligenz war dieses Konzept bis Mitte der 1980er Jahre nur von peripherem Interesse, bis unzählige Diskussionen über die richtigen technischen Grundlagen des Bereiches entstanden. Eine Arbeit von Jon Doyle (1983) sagte voraus, dass der Entwurf rationaler Agenten zu einer der wichtigsten Aufgaben

in der KI werden würde, während andere populäre Themen sich abspalten und neue Disziplinen bilden würden.

Eine sorgfältige Berücksichtigung der Umgebungseigenschaften und ihrer Konsequenzen für den Entwurf rationaler Agenten ist in der Tradition der Regelungstheorie am offensichtlichsten – beispielsweise beschäftigen sich klassische Steuerungssysteme (Dorf und Bishop, 2004; Kirk, 2004) mit vollständig beobachtbaren, deterministischen Umgebungen; die stochastisch optimale Steuerung (Kumar und Varaiya, 1986; Bertsekas und Shreve, 2007) befasst sich mit teilweise beobachtbaren, stochastischen Umgebungen; und die hybride Steuerung (Henzinger und Sastry, 1998; Cassandras und Lygeros, 2006) beschreibt Umgebungen, die sowohl diskrete als auch stetige Elemente enthalten. Die Unterscheidung zwischen vollständig und teilweise beobachtbaren Umgebungen ist ebenfalls zentral in der Literatur zur **dynamischen Programmierung**, die im Bereich der Operationsforschung entwickelt wurde (Puterman, 1994) und auf die wir in *Kapitel 17* genauer eingehen werden.

Reflexagenten waren das primäre Modell für psychologische Behavioristen, wie beispielsweise Skinner (1953), der versuchte, die Psychologie von Organismen streng auf Eingaben-Ausgaben- oder Stimulanz-Reaktions-Abbildungen zu reduzieren. Der Übergang vom Behaviorismus zum Funktionalismus in der Psychologie, der zumindest teilweise durch die Anwendung der Computermetapher auf Agenten vorangetrieben wurde (Putnam, 1960; Lewis, 1966), führte den internen Zustand des Agenten in das Bild ein. Die meisten Arbeiten in der KI betrachten das Konzept reiner Reflexagenten mit Zustand als zu einfach, um eine große Unterstützung darzustellen, aber die Arbeiten von Rosenschein (1985) und Brooks (1986) stellten diese Annahme in Frage (siehe *Kapitel 25*). In den letzten Jahren wurde viel Arbeit darauf verwendet, effiziente Algorithmen zu finden, um komplexe Umgebungen zu verfolgen (Hamscher et al., 1992; Simon, 2006). Das Programm Remote Agent, das das Raumschiff Deep Space One (siehe *Abschnitt 1.4*) steuerte, ist ein besonders beeindruckendes Beispiel (Muscettola et al., 1998; Jonsson et al., 2000).

Zielbasierte Agenten werden überall vorausgesetzt, von Aristoteles' Anschauung des praktischen Schließens bis hin zu den früheren Arbeiten von McCarthy über die logische KI. Shakey, der Roboter (Fikes und Nilsson, 1971; Nilsson, 1984), war die erste robotische Verkörperung eines logischen, zielbasierten Agenten. Eine vollständige logische Analyse zielbasierter Agenten erschien in Genesereth und Nilsson (1987) und Shoham (1993) entwickelte eine zielbasierte Programmiermethodologie, die sogenannte agentenorientierte Programmierung. Der agentenbasierte Ansatz ist heute in der Softwaretechnik äußerst populär (Ciancarini und Wooldridge, 2001). Er ist auch in das Gebiet der Betriebssysteme eingedrungen, wo sich **autonome Datenverarbeitung** auf Computersysteme und Netzwerke bezieht, die sich mit einer Wahrnehmen-Handeln-Schleife und Methoden maschinellen Lernens selbst überwachen und steuern (Kephart und Chess, 2003). Eine Sammlung von Agentenprogrammen, die dafür konzipiert sind, in einer echten Multiagentenumgebung reibungslos zusammenzuarbei-

ten, weist zwangsläufig Modularität auf – die Programme nutzen keinen internen Status gemeinsam und kommunizieren nur über die Umgebung miteinander. Im Bereich der **Multiagentensysteme** ist es üblich, das Agentenprogramm eines einzelnen Agenten als Sammlung von autonomen Subagenten zu entwerfen. In manchen Fällen kann man sogar beweisen, dass das resultierende System die gleichen optimalen Lösungen wie ein monolithisches Design liefert.

Die zielbasierte Sicht dominiert außerdem die Tradition der kognitiven Psychologie im Bereich des Problemlösens. Sie begann mit dem enorm einflussreichen *Human Problem Solving* (Newell und Simon, 1972) und durchsetzte alle späteren Arbeiten von Newell (Newell, 1990). Ziele, weiter analysiert als Wünsche (allgemein) und Absichten (aktuell verfolgt), sind zentral für die von Bratman (1987) entwickelte Agententheorie. Diese Theorie war einflussreich sowohl für das Verständnis natürlicher Sprache als auch für Multiagentensysteme.

Horvitz et al. (1988) schlagen insbesondere die Verwendung der Rationalität als Maximierung erwarteten Nutzens als Grundlage für die KI vor. Der Text von Pearl (1988) war der erste in der KI, der Wahrscheinlichkeit und Nützlichkeitstheorie im Detail betrachtete. Seine Darlegungen praktischer Methoden für das Schließen und Entscheiden unter Unsicherheit war wahrscheinlich der größte Faktor in der schnelleren Entwicklung hin zu nutzenbasierten Agenten in den 1990er Jahren (siehe Teil 4).

Der in Abbildung 2.15 gezeigte allgemeine Entwurf für lernende Agenten ist klassisch in der Literatur zum maschinellen Lernen (Buchanan et al., 1978; Mitchell, 1997). Beispiele für den Entwurf, wie er in Programmen verkörpert wird, gehen mindestens bis zum Lernprogramm für ein Damespiel von Arthur Samuel (1959, 1967) zurück. Lernende Agenten werden in Teil 5 genauer beschrieben.

Das Interesse an Agenten und deren Entwurf ist in den letzten Jahren rapide gestiegen, zum Teil auch aufgrund des Wachstums des Internets und des erkannten Bedarfes an automatisierten und mobilen **Softbots** (Etzioni und Weld, 1994). Relevante Artikel sind in *Readings in Agents* (Huhn und Singh, 1998) und *Foundations of Rational Agency* (Wooldridge und Rao, 1999) zusammengefasst. Artikel zu Multiagentensystemen bieten in der Regel eine gute Einführung zu vielen Aspekten des Agentenentwurfes (Weiss, 2000a; Wooldridge, 2002). In den 1990er Jahren begannen mehrere Konferenzreihen zum Thema Agenten, unter anderem der *International Workshop on Agent Theories, Architectures and Languages* (ATAL), die *International Conference on Autonomous Agents* (AGENTS) und die *International Conference on Multi-Agent Systems* (ICMAS). Im Jahr 2002 wurden diese drei Veranstaltungen vereint, um die *International Joint Conference on Autonomous Agents and Multi-Agent Systems* (AAMAS) zu bilden. Die Zeitschrift *Autonomous Agents and Multi-Agent Systems* wurde 1998 gegründet. Schließlich bietet *Dung Beetle Ecology* (Hanski und Cambefort, 1991) eine Fülle von interessanten Informationen über das Verhalten von Mistkäfern. Auf YouTube finden sich inspirierende Videoaufnahmen ihrer Aktivitäten.

Zusammenfassung

Dieses Kapitel war ein Schnelldurchgang durch die KI, hier als die Wissenschaft des Agentenentwurfes betrachtet. Die wichtigsten Aspekte können wie folgt zusammengefasst werden:

- Ein **Agent** ist etwas, das Wahrnehmungen in einer Umgebung aufnimmt und dort handelt. Die **Agentenfunktion** für einen Agenten spezifiziert die Aktion, die der Agent als Reaktion auf eine Wahrnehmungsfolge unternimmt.
- Die **Leistungsbewertung** gewichtet das Verhalten des Agenten in einer Umgebung. Ein **rationaler Agent** agiert so, dass er unter Berücksichtigung der bisher gesehenen Wahrnehmungsfolge den erwarteten Wert der Leistungsbewertung maximiert.
- Die Spezifikation einer **Aufgabenumgebung** beinhaltet die Leistungsbewertung, die externe Umgebung, die Aktuatoren und die Sensoren. Beim Entwurf eines Agenten muss der erste Schritt immer sein, die Aufgabenumgebung so vollständig wie möglich zu spezifizieren.
- Aufgabenumgebungen variieren entlang mehrerer wesentlicher Dimensionen. Sie können vollständig oder teilweise beobachtbar, Einzel- oder Multiagentenumgebungen, deterministisch oder stochastisch, episodisch oder sequenziell, statisch oder dynamisch, diskret oder stetig und bekannt oder unbekannt sein.
- Das **Agentenprogramm** implementiert die Agentenfunktion. Es gibt eine Vielzahl grundlegender Entwürfe für Agentenprogramme, die die Art der Information reflektieren, die explizit angegeben ist und für den Entscheidungsprozess verwendet wird. Die Entwürfe unterscheiden sich im Hinblick auf Effizienz, Kompaktheit und Flexibilität. Welcher Entwurf für das Agentenprogramm geeignet ist, hängt von der Art der Umgebung ab.
- **Einfache Reflexagenten** reagieren direkt auf Wahrnehmungen, während **modellbasierte Reflexagenten** einen internen Zustand verwalten, um Aspekte der Welt zu verfolgen, die in der aktuellen Wahrnehmung nicht offensichtlich sind. **Zielbasierte Agenten** agieren, um ihre Ziele zu erreichen, und **nutzenbasierte Agenten** versuchen, ihre eigene erwartete „Zufriedenheit" zu maximieren.
- Alle Agenten können ihre Leistung durch **Lernen** verbessern.

Übungen zu Kapitel 2

Lösungshinweise

1 Untersuchen Sie die Rationalität verschiedener Staubsauger-Agentenfunktionen.

a. Zeigen Sie, dass die in Abbildung 2.3 beschriebene einfache Staubsauger-Agentenfunktion unter den im darauffolgenden *Abschnitt „Rationalität"* aufgelisteten Voraussetzungen tatsächlich rational ist.

b. Beschreiben Sie eine rationale Agentenfunktion für den Fall, in dem jede Bewegung einen Punkt kostet. Braucht das zugehörige Agentenprogramm einen internen Zustand?

c. Diskutieren Sie mögliche Agentenentwürfe für die Fälle, in denen saubere Quadrate wieder schmutzig werden können und die Geografie der Umgebung unbekannt ist. Ist es in diesen Fällen sinnvoll, dass der Agent aus seiner Erfahrung lernt? Wenn dies der Fall ist, was sollte er lernen?

2 Schreiben Sie eine Abhandlung zur Beziehung zwischen Evolution und Autonomie, Intelligenz und/oder Lernen.

3 Geben Sie für die folgenden Aussagen an, ob sie wahr oder falsch sind, und untermauern Sie Ihre Antwort mit Beispielen oder Gegenbeispielen, wo es zweckmäßig ist.

a. Ein Agent, der nur teilweise Informationen über den Zustand erfasst, kann nicht perfekt rational sein.

b. Es existieren Aufgabenumgebungen, in denen sich kein reiner Reflexagent rational verhalten kann.

c. Es existiert eine Aufgabenumgebung, in der jeder Agent rational ist.

d. Die Eingabe für ein Agentenprogramm ist dieselbe wie die Eingabe für die Agentenfunktion.

e. Jede Agentenfunktion lässt sich durch eine bestimmte Programm-/Maschinenkombination implementieren.

f. Angenommen, ein Agent wählt seine Aktion gleich verteilt zufällig aus dem Satz der möglichen Aktionen aus. Es existiert eine deterministische Aufgabenumgebung, in der der Agent rational ist.

g. Es ist möglich, dass ein bestimmter Agent in zwei unterschiedlichen Aufgabenumgebungen perfekt rational ist.

h. Jeder Agent ist in einer nicht beobachtbaren Umgebung rational.

i. Ein perfekter rationaler Agent für das Pokerspiel verliert niemals.

4 Geben Sie für die folgenden Aktivitäten eine PEAS-Beschreibung der Aufgabenumgebung an und charakterisieren Sie sie in Bezug auf die in *Abschnitt 2.3.2* aufgelisteten Eigenschaften:

a. Eine Bodenturnübung durchführen

b. Auf dem Meeresgrund nach Titan forschen

c. Fußball spielen

d. Im Internet gebrauchte Bücher zur KI kaufen

e. Tennis gegen eine Ballwand spielen

f. Einen Hochsprung ausführen

g. Auf einer Auktion für einen Artikel bieten

5 Definieren Sie mit eigenen Worten die folgenden Begriffe: Agent, Agentenfunktion, Agentenprogramm, Rationalität, Autonomie, Reflexagent, modellbasierter Agent, zielbasierter Agent, nutzenbasierter Agent, lernender Agent.

6 Diese Übung betrachtet die Unterschiede zwischen Agentenfunktionen und Agentenprogrammen.

a. Kann es mehrere Agentenprogramme geben, die eine bestimmte Agentenfunktion implementieren? Nennen Sie ein Beispiel oder zeigen Sie, warum es nicht möglich ist.

b. Gibt es Agentenfunktionen, die von keinem Agentenprogramm implementiert werden können?

c. Implementiert jedes Agentenprogramm für eine vorgegebene feste Maschinenarchitektur genau eine Agentenfunktion?

d. Wie viele verschiedene mögliche Agentenprogramme gibt es für eine bestimmte Architektur mit n Bit Speicher?

e. Angenommen, wir lassen das Agentenprogramm unverändert, erhöhen aber die Geschwindigkeit des Computers auf das Doppelte. Ändert sich dadurch die Agentenfunktion?

7 Schreiben Sie als Pseudocode Agentenprogramme für die zielbasierten und nutzenbasierten Agenten.

8 Ein einfacher Thermostat soll einen Ofen einschalten, wenn die Temperatur mindestens 3 Grad unterhalb der Einstellung liegt, und den Ofen ausschalten, wenn die Temperatur mindestens 3 Grad oberhalb der Einstellung liegt. Ist ein Thermostat eine Instanz eines einfachen Reflexagenten, eines modellbasierten Agenten oder eines zielbasierten Agenten?

Die folgenden Übungen beschäftigen sich alle mit der Implementierung von Umgebungen und Agenten für die Staubsaugerwelt.

9 Implementieren Sie einen Leistungsbewertungs-Umgebungssimulator für die in Abbildung 2.2 gezeigte und im *Abschnitt 2.2.1* spezifizierte Staubsaugerwelt. Ihre Implementierung sollte modular sein, sodass die Eigenschaften von Sensoren, Aktuatoren und Umgebung (Größe, Form, Schmutzplatzierung usw.) einfach geändert werden können. (*Hinweis:* Für einige Programmiersprachen und Betriebssysteme gibt es bereits Implementierungen in dem online zur Verfügung gestellten Code.)

10 Betrachten Sie eine veränderte Version der Staubsaugerumgebung aus Übung 2.9, in der der Agent für jede Bewegung mit einem Punktabzug bestraft wird.

a. Kann ein einfacher Reflexagent perfekt rational für diese Umgebung sein? Erläutern Sie Ihre Antwort.

b. Wie verhält es sich bei einem Reflexagenten mit Zuständen? Entwerfen Sie einen solchen Agenten.

c. Wie ändern sich Ihre Antworten für a und b, wenn die Wahrnehmungen dem Agenten den Sauber/Schmutzig-Zustand jedes Quadrates in der Umgebung mitteilen?

11 Betrachten Sie eine veränderte Version der Staubsaugerumgebung aus Übung 2.9, wobei die Geografie der Umgebung – Größe, Begrenzung und Hindernisse – ebenso wie die Anfangskonfiguration des Schmutzes unbekannt sind. (Der Agent kann sich neben *Links* und *Rechts* auch *Vor* und *Zurück* bewegen.)

a. Kann ein einfacher Reflexagent perfekt rational für diese Umgebung sein? Erläutern Sie Ihre Antwort.

b. Kann ein einfacher Reflexagent mit *zufallsgesteuerter* Agentenfunktion besser abschneiden als ein einfacher Reflexagent? Entwerfen Sie einen solchen Agenten und messen Sie seine Leistung für mehrere Umgebungen.

c. Können Sie eine Umgebung entwerfen, in der sich Ihr zufallsgesteuerter Agent sehr schlecht verhält? Zeigen Sie Ihre Ergebnisse.

d. Kann ein Reflexagent mit Zustand besser sein als ein einfacher Reflexagent? Entwerfen Sie einen derartigen Agenten und messen Sie seine Leistung für mehrere Umgebungen. Können Sie einen rationalen Agenten dieses Typs entwerfen?

12 Wiederholen Sie Übung 2.11 für den Fall, in dem der Positionssensor durch einen Kollisionssensor ersetzt wird, der erkennt, wenn der Agent auf ein Hindernis stößt oder die Grenzen der Umgebung überschreitet. Wie sollte sich der Agent verhalten, wenn der Kollisionssensor ausfällt?

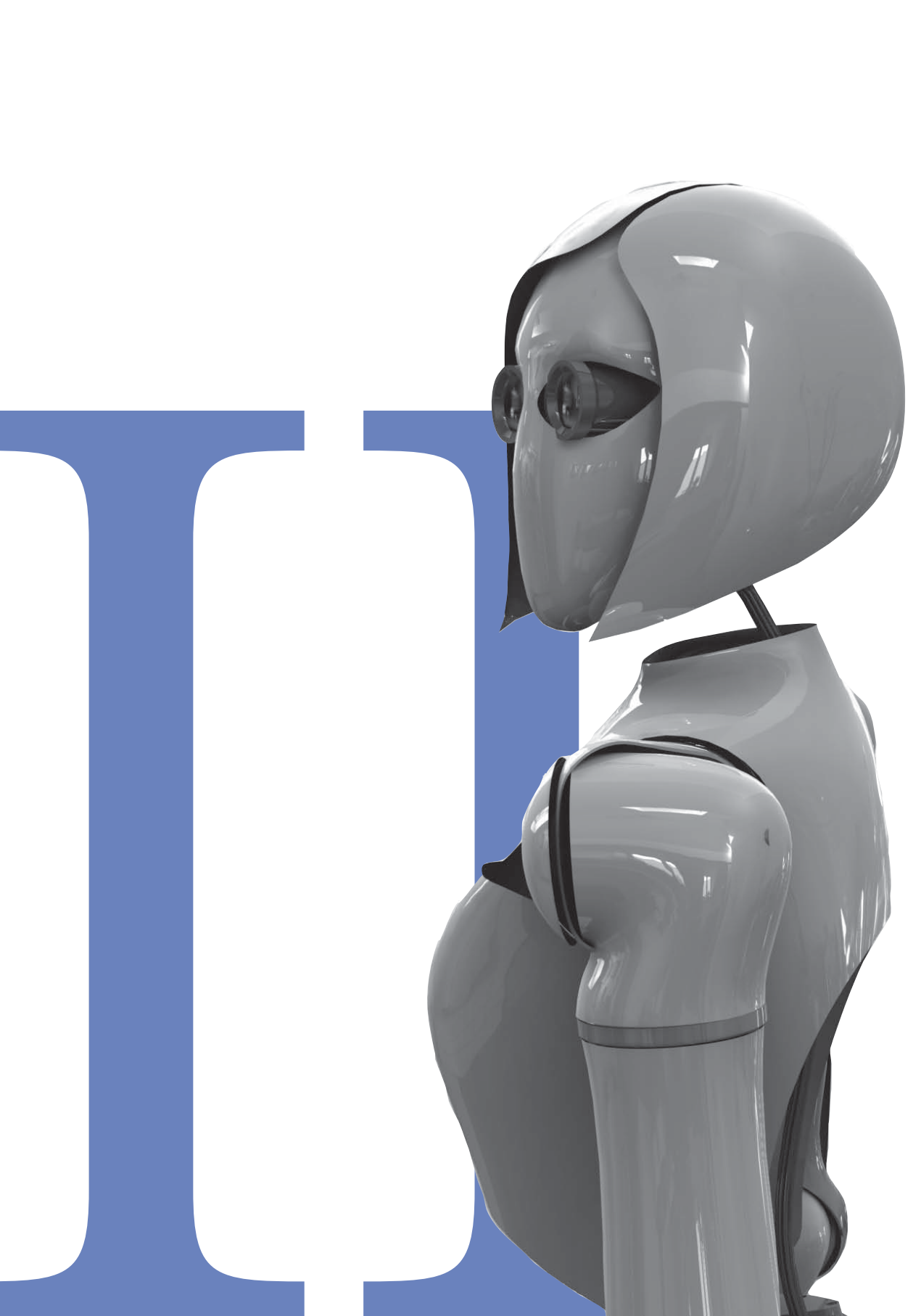
13 Die Staubsaugerumgebungen in den vorherigen Übungen waren alle deterministisch. Diskutieren Sie mögliche Agentenprogramme für jede der folgenden stochastischen Versionen:

a. Murphys Gesetz: In 25 Prozent aller Fälle kann die Aktion *Saugen* den Boden nicht säubern, wenn er schmutzig ist, und wirft Schmutz auf den Boden, wenn er sauber ist. Wie wird Ihr Agentenprogramm beeinflusst, wenn der Schmutzsensor in zehn Prozent aller Fälle die falsche Antwort gibt?

b. Kleinkinder: Nach jedem Zeitintervall besteht für jedes saubere Quadrat eine zehnprozentige Wahrscheinlichkeit, dass es wieder schmutzig wird. Können Sie einen rationalen Agenten für diesen Fall entwerfen?

TEIL II

Problemlösen



Problemlösung durch Suchen

3

ÜBERBLICK

In diesem Kapitel erfahren Sie, wie ein Agent eine Aktionsfolge ermittelt, mit der er an seine Ziele gelangt, wenn eine einzelne Aktion dafür nicht ausreichend ist.

Die einfachsten in *Kapitel 2* beschriebenen Agenten waren die Reflexagenten, die ihre Aktionen auf einer direkten Zuordnung von Zuständen zu Aktionen aufbauen. Derartige Agenten verhalten sich nicht zufriedenstellend in Umgebungen, für die diese Zuordnung zu groß ist, um gespeichert zu werden, und für die es zu lange dauern würde, um sie zu erlernen. Zielbasierte Agenten dagegen berücksichtigen zukünftige Aktionen und ziehen dabei in Betracht, wie wünschenswert ihre Ergebnisse sind.

Dieses Kapitel beschreibt eine Variante eines zielbasierten Agenten, den sogenannten **problemlösenden Agenten**. Problemlösende Agenten verwenden **atomare** Darstellungen, wie sie *Abschnitt 2.4.7* beschrieben hat – d.h., Zustände der Welt werden als Ganzes betrachtet, wobei für die problemlösenden Algorithmen keine interne Struktur sichtbar ist. Bei zielbasierten Agenten, die erweiterte **faktorisierte** oder **strukturierte** Darstellungen verwenden, spricht man normalerweise von **planenden Agenten**, mit denen sich die *Kapitel 7* und *10* beschäftigen.

Wir beginnen unsere Diskussion zum Problemlösen mit genauen Definitionen von **Problemen** und ihren **Lösungen** und zeigen mehrere Beispiele, um diese Definitionen zu veranschaulichen. Anschließend beschreiben wir mehrere allgemeine Suchalgorithmen, die für die Lösung dieser Probleme verwendet werden können. Es werden verschiedene **nicht informierte** Suchalgorithmen vorgestellt – Algorithmen, die keine Informationen über das Problem außer seiner Definition erhalten. Obwohl einige dieser Algorithmen jedes lösbare Problem lösen können, ist keiner davon in der Lage, dies effizient zu tun. Besser verhalten sich dagegen **informierte** Suchalgorithmen, die bestimmte Hinweise erhalten, wo nach Lösungen zu suchen ist.

In diesem Kapitel beschränken wir uns auf die einfachste Art von Aufgabenumgebung, bei der die Lösung für ein Problem immer eine feste Sequenz von Aktionen ist. Der allgemeine Fall – in dem zukünftige Aktionen des Agenten abhängig von zukünftigen Wahrnehmungen variieren können – wird in *Kapitel 4* behandelt.

Dieses Kapitel verwendet Konzepte der asymptotischen Komplexität (d.h. der $O()$-Notation) und NP-Vollständigkeit. Leser, die damit nicht vertraut sind, sollten in Anhang A nachlesen.

3.1 Problemlösende Agenten

Intelligente Agenten sollten versuchen, ihr Leistungsmaß zu maximieren. Wie wir in *Kapitel 2* erwähnt haben, wird dies manchmal einfacher, wenn der Agent ein **Ziel** annehmen kann und versucht, es zu erreichen. Betrachten wir zuerst, wie und warum ein Agent dies tun sollte.

Stellen Sie sich vor, ein Agent befindet sich in der Innenstadt von Arad in Rumänien und genießt eine Städtereise. Das Leistungsmaß des Agenten beinhaltet viele Faktoren: Er möchte seine Sonnenbräune vertiefen, sein Rumänisch verbessern, sich alle Sehenswürdigkeiten ansehen, das Nachtleben genießen (falls es ein solches gibt), einen Kater vermeiden usw. Das Entscheidungsproblem ist komplex und bedingt viele Kompromisse und ein sorgfältiges Studium von Touristenführern. Nehmen wir nun an, der Agent hat ein nicht umtauschbares Flugticket ab Bukarest für den nächsten Tag. In diesem Fall ist es sinnvoll, dass der Agent das **Ziel** verfolgt, nach Bukarest zu gelangen.

Aktionsfolgen, die dazu führen, dass Bukarest nicht rechtzeitig erreicht werden kann, können ohne weitere Überlegung zurückgewiesen werden und das Entscheidungsproblem des Agenten wird wesentlich vereinfacht. Ziele helfen, das Verhalten zu organisieren, indem die Etappenziele begrenzt werden, die der Agent zu erreichen versucht, und folglich die Aktionen, die er berücksichtigen muss. Die **Zielformulierung**, die auf der aktuellen Situation und dem Leistungsmaß des Agenten basiert, ist der erste Schritt zur Problemlösung.

Wir betrachten ein Ziel als eine Menge von Zuständen der Welt – und zwar jener Zustände, in denen das Ziel erfüllt wird. Der Agent hat nun die Aufgabe herauszufinden, wie er jetzt und in Zukunft zu handeln hat, damit er einen Zielzustand erreicht. Bevor er das tun kann, muss er entscheiden (oder wir müssen in seinem Namen entscheiden), welche Aktionen und Zustände berücksichtigt werden sollen. Wenn er versucht, Aktionen auf dem Niveau von „Bewege deinen linken Fuß um einen Zentimeter nach vorne“ oder „Dreh das Lenkrad um ein Grad nach links“ zu berücksichtigen, wird er womöglich nie aus der Parklücke herauskommen, ganz zu schweigen von Bukarest, weil auf dieser Detailebene zu viel Unsicherheit besteht und er zu viele Schritte für eine Lösung braucht. Die **Problemformulierung** ist der Prozess, zu entscheiden, welche Aktionen und Zustände für ein gegebenes Ziel berücksichtigt werden sollen. Wir werden diesen Prozess später genauer betrachten. Hier wollen wir annehmen, dass der Agent Aktionen auf der Ebene berücksichtigt, von einer Stadt in eine andere zu fahren. Jeder Zustand entspricht demzufolge dem Aufenthalt in einer bestimmten Stadt.

Unser Agent hat nun das Ziel angenommen, nach Bukarest zu fahren, und überlegt, wie er von Arad aus dorthin gelangt. Drei Straßen führen aus Arad heraus, eine nach Sibiu, eine andere nach Timisoara und eine nach Zerind. Keine davon erreicht das Ziel. Wenn der Agent also nicht sehr vertraut mit der rumänischen Geografie ist, weiß er nicht, welcher Straße er folgen soll.[1] Mit anderen Worten, der Agent weiß nicht, welche seiner möglichen Aktionen die beste ist, weil er noch nicht genug über den Zustand weiß, der aus der Ausführung der einzelnen Aktionen resultiert. Wenn der Agent über kein zusätzliches Wissen verfügt – d.h., wenn die Umgebung in dem Sinne **unbekannt** ist, wie es in *Abschnitt 2.3* definiert wurde –, hat er keine andere Wahl, als eine der Aktionen zufällig auszuprobieren. *Kapitel 4* beschäftigt sich mit dieser bedauerlichen Situation.

Tipp

Nehmen wir jedoch an, der Agent besitzt eine Straßenkarte von Rumänien. Diese Straßenkarte versorgt den Agenten mit Informationen über die Zustände, in die er geraten könnte, sowie über die Aktionen, die er ausführen kann. Der Agent kann diese Informationen nutzen, um zukünftige Phasen einer hypothetischen Reise durch jede der drei Städte in Betracht zu ziehen, um eine Route zu finden, die schließlich nach Bukarest führt. Nachdem er auf der Karte einen Pfad von Arad nach Bukarest gefunden hat, kann er sein Ziel erreichen, indem er die Fahraktionen ausführt, die den Reiseetappen entsprechen. Im Allgemeinen *kann ein Agent mit mehreren direkten Optionen unbekannten Wertes entscheiden, was zu tun ist, indem er zuerst* zukünftige *Aktionen untersucht, die schließlich zu Zuständen mit bekanntem Wert führen.*

1 Wir gehen davon aus, dass es den meisten Lesern nicht anders geht, und Sie können sich selbst vorstellen, wie ratlos unser Agent ist. Wir entschuldigen uns bei den rumänischen Lesern, die diesem pädagogischen Trick nicht folgen können.

Um konkreter zu erläutern, was wir unter „zukünftige Aktionen untersuchen“ verstehen, müssen wir genauer auf die Eigenschaften der Umgebung eingehen, wie sie *Abschnitt 2.3* definiert hat. Fürs Erste nehmen wir an, dass die Umgebung **beobachtbar** ist, sodass der Agent immer den aktuellen Zustand kennt. Für den Agenten, der in Rumänien fährt, ist es sinnvoll anzunehmen, dass sich jede Stadt auf der Landkarte mit einem Zeichen für ankommende Fahrer zu erkennen gibt. Außerdem nehmen wir die Umgebung als **diskret** an, sodass bei jedem Zustand nur endlich viele Aktionen zur Auswahl stehen. Für die Navigation in Rumänien ist dies zutreffend, da jede Stadt mit einigen anderen Städten verbunden ist. Die Umgebung setzen wir als **bekannt** voraus, sodass der Agent weiß, welche Zustände durch jede Aktion erreicht werden. (Eine genaue Karte genügt, um diese Bedingung für Navigationsprobleme zu erfüllen.) Schließlich nehmen wir an, dass die Umgebung **deterministisch** ist, sodass jede Aktion genau ein Ergebnis liefert. Unter idealen Bedingungen trifft dies für den Agenten in Rumänien zu – wenn er sich also entscheidet, von Arad nach Sibiu zu fahren, kommt er letztlich auch in Sibiu an. Wie *Kapitel 4* zeigt, sind die Bedingungen natürlich nicht immer ideal.

Tipp

Unter diesen Annahmen ist die Lösung für ein beliebiges Problem eine feste Sequenz von Aktionen. „Selbstverständlich!“, mag man sagen, „Wie sollte es sonst sein?“. Im Allgemeinen könnte es eine Verzweigungsstrategie sein, die abhängig von eintreffenden Wahrnehmungen unterschiedliche Aktionen in der Zukunft empfiehlt. Zum Beispiel könnte der Agent unter nicht ganz idealen Bedingungen zwar planen, von Arad nach Sibiu und dann nach Rimnicu Vilcea zu fahren, er müsste aber auch einen Ausweichplan bereithalten, falls er versehentlich in Zerind statt in Sibiu ankommt. Wenn der Agent in der glücklichen Lage ist, den Anfangszustand zu kennen, und die Umgebung bekannt und deterministisch ist, weiß er genau, wo er nach der ersten Aktion sein wird und was er wahrnehmen wird. Da nach der ersten Aktion nur genau eine Wahrnehmung möglich ist, kann die Lösung nur eine mögliche zweite Aktion spezifizieren usw.

Das Ermitteln einer Folge von Aktionen, die das Ziel erreicht, wird als **Suche** bezeichnet. Ein Suchalgorithmus nimmt ein Problem als Eingabe entgegen und gibt eine **Lösung** in Form einer Aktionsfolge zurück. Nachdem eine Lösung gefunden wurde, können die Aktionen ausgeführt werden, die sie empfiehlt. Man spricht hierbei von der **Ausführungsphase**. Wir haben also einen einfachen „Formulieren-Suchen-Ausführen“-Entwurf für den Agenten, wie in ▶ Abbildung 3.1 gezeigt. Nach der Formulierung eines Zieles und eines zu lösenden Problems ruft der Agent eine Suchprozedur auf, um es zu lösen. Anschließend verwendet er die Lösung, um seine Aktionen auszuführen und als Nächstes das zu tun, was die Lösung empfiehlt – in der Regel die erste Aktion der Folge –, und diesen Schritt dann aus der Folge zu entfernen. Nachdem die Lösung ausgeführt wurde, formuliert der Agent ein neues Ziel.

Während der Agent die Lösungsfolge ausführt, *ignoriert er seine Wahrnehmungen*, wenn er eine Aktion auswählt, da er im Voraus weiß, wie sie aussehen. Ein Agent, der seine Pläne bei geschlossenen Augen ausführt, muss sich schon sehr sicher sein, was passiert. Regelungstheoretiker bezeichnen dies als **Open-Loop**-System, weil das Ignorieren der Wahrnehmungen die Schleife zwischen dem Agenten und der Umgebung unterbricht.

Wir beschreiben zuerst den Prozess der Problemformulierung und widmen dann einen Großteil des Kapitels verschiedenen Algorithmen für die SEARCH-Funktion. Auf die Arbeitsweise der Funktionen UPDATE-STATE und FORMULATE-GOAL gehen wir in diesem Kapitel nicht weiter ein.

```
function SIMPLE-PROBLEM-SOLVING-AGENT (percept) returns eine Aktion
  persistent:  seq, eine Aktionsfolge, anfangs leer
               state, eine Beschreibung des aktuellen Weltzustands
               goal, ein Ziel, anfangs null
               problem, eine Problemformulierung

  state ← UPDATE-STATE(state, percept)
  if seq ist leer then
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
    if seq = failure then return eine Nullaktion
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Abbildung 3.1: Ein einfacher problemlösender Agent. Er formuliert zuerst ein Ziel und ein Problem, sucht nach einer Aktionsfolge, die das Problem lösen würde, und führt dann die Aktionen nacheinander aus. Nachdem er damit fertig ist, formuliert er ein weiteres Ziel und beginnt von Neuem.

3.1.1 Wohldefinierte Probleme und Lösungen

Ein **Problem** kann formal durch vier Komponenten definiert werden:

- dem **Ausgangszustand**, in dem der Agent beginnt. Zum Beispiel könnte der Ausgangszustand für unseren Agenten in Rumänien als *In*(*Arad*) beschrieben werden.
- einer Beschreibung der möglichen **Aktionen**, die dem Agenten zur Verfügung stehen. Für einen gegebenen Zustand s gibt die Funktion ACTIONS(s) die Menge der Aktionen zurück, die sich in s ausführen lassen. Wir sagen, dass jede dieser Aktionen in s anwendbar ist. Zum Beispiel sind vom Zustand *In*(*Arad*) aus die Aktionen {*Go*(*Sibiu*), *Go*(*Timisoara*), *Go*(*Zerind*)} **anwendbar**.
- einer Beschreibung, was jede Aktion bewirkt, formal als **Überführungsmodell** bezeichnet und durch eine Funktion RESULT(s, a) spezifiziert, die den Zustand zurückgibt, der von der Ausführung der Aktion a im Zustand s resultiert. Außerdem bezeichnen wir mit **Nachfolger** einen Zustand, der von einem gegebenen Zustand durch eine einzelne Aktion erreichbar ist.[2] Zum Beispiel haben wir:

  RESULT(*In*(*Arad*), *Go*(*Zerind*)) = *In*(*Zerind*).

  Zusammen definieren Ausgangszustand, Aktionen und Überführungsmodell implizit den **Zustandsraum** des Problems – die Menge aller Zustände, die von diesem Ausgangszustand durch eine Aktionsfolge erreichbar sind. Der Zustandsraum bildet ein gerichtetes Netz oder einen **Graphen**, in dem die Knoten Zustände darstellen und die Verknüpfungen zwischen den Knoten die Aktionen sind. (Die in ▶ Abbildung 3.2 gezeigte Landkarte von Rumänien kann als Graph des Zustandsraumes interpretiert

2 Viele Abhandlungen zur Problemlösung, einschließlich der vorherigen Ausgaben dieses Buches, verwenden anstelle der separaten Funktionen ACTIONS und RESULT eine **Nachfolgerfunktion**, die die Menge aller Nachfolger zurückgibt. Mit der Nachfolgerfunktion ist es recht schwierig, einen Agenten zu beschreiben, der weiß, welche Aktionen er probieren kann, aber nicht, was sie erreichen. Außerdem verwenden manche Autoren RESULT(a, s) anstelle von RESULT(s, a) und andere wieder DO anstelle von RESULT.

werden, wenn wir annehmen, dass jede Straße für zwei Fahraktionen steht – eine in jede Richtung.) Ein **Pfad** im Zustandsraum ist eine Abfolge von Zuständen, die durch eine Aktionsfolge verbunden sind.

- dem **Zieltest**, der entscheidet, ob ein bestimmter Zustand ein Zielzustand ist. Manchmal gibt es eine explizite Menge möglicher Zielzustände und der Test überprüft einfach, ob der gegebene Zustand einer davon ist. Das Ziel des Agenten in Rumänien ist die aus einem Element bestehende Menge {*In*(*Bukarest*)}. Manchmal wird das Ziel als abstrakte Eigenschaft und nicht als explizite Auflistung einer Menge von Zuständen spezifiziert. Im Schach beispielsweise ist das Ziel, einen Zustand namens „Schachmatt" zu erreichen, in dem der gegnerische König angegriffen wird und nicht mehr entkommen kann.
- einer **Pfadkosten**funktion, die jedem Pfad einen numerischen Kostenwert zuweist. Der problemlösende Agent wählt eine Kostenfunktion, die sein eigenes Leistungsmaß reflektiert. Da für den Agenten, der nach Bukarest zu gelangen versucht, hauptsächlich die Zeit eine Rolle spielt, könnten die Kosten eines Pfades durch seine Länge in Kilometern ausgedrückt werden. In diesem Kapitel gehen wir davon aus, dass die Kosten eines Pfades als die *Summe* der Kosten der einzelnen Aktionen entlang des Pfades beschrieben werden können.[3] Die **Schrittkosten** für die Durchführung einer Aktion a, um vom Zustand s in den Zustand s' zu gelangen, werden als $c(s, a, s')$ ausgedrückt. Die Schrittkosten für Rumänien sind in Abbildung 3.2 als Entfernungen dargestellt. Wir gehen davon aus, dass Schrittkosten nicht negativ sind.[4]

Die oben beschriebenen Elemente definieren ein Problem und können in einer einzelnen Datenstruktur zusammengefasst werden, die einem Problemlösungsalgorithmus als Eingabe übergeben wird. Eine **Lösung** für ein Problem ist eine Aktionsfolge, die vom Ausgangszustand zu einem Zielzustand führt. Die Lösungsqualität wird anhand der Pfadkostenfunktion gemessen und eine **optimale Lösung** hat die geringsten Pfadkosten aller Lösungen.

3.1.2 Probleme formulieren

Im obigen Abschnitt haben wir eine Formulierung für das Problem, nach Bukarest zu gelangen, in Form von Ausgangszustand, Aktionen, Übergangsmodell, Zieltest und Pfadkosten vorgeschlagen. Diese Formulierung erscheint vernünftig, ist aber immer noch ein *Modell* – eine abstrakte mathematische Beschreibung – und nicht die Wirklichkeit. Vergleichen Sie die hier gewählte einfache Zustandsbeschreibung *In*(*Arad*) mit einer tatsächlichen Reise durch das Land, wobei der Zustand der Welt so viele weitere Kleinigkeiten beinhaltet: die Reisebegleiter, die Musik im Radio, die Bilder, die man durch das Fenster sieht, die etwaige Anwesenheit von Verkehrspolizisten, die Entfernung zum nächsten Rastplatz, den Straßenzustand, das Wetter usw. All diese Betrachtungen werden in unseren Zustandsbeschreibungen nicht berücksichtigt, weil sie irrelevant für das Problem sind, einen Weg nach Bukarest zu finden. Der Prozess, Details aus einer Repräsentation zu entfernen, wird als **Abstraktion** bezeichnet.

3 Diese Annahme ist algorithmisch komfortabel, aber auch theoretisch zu rechtfertigen – siehe dazu *Abschnitt 17.2*.

4 Die Auswirkungen negativer Kosten werden in Übung 3.8 genauer betrachtet.

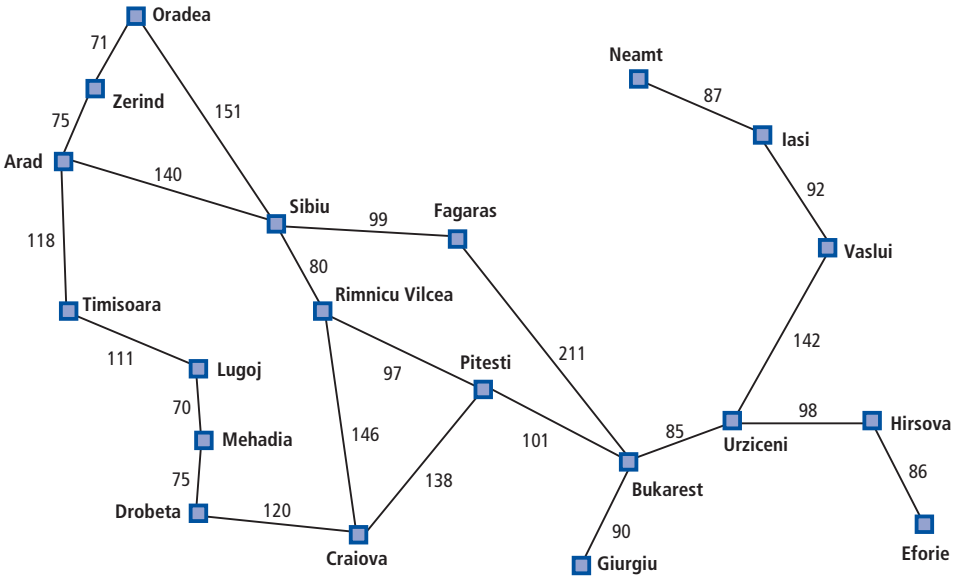


Abbildung 3.2: Eine vereinfachte Straßenkarte eines Teiles von Rumänien.

Neben der Abstrahierung der Zustandsbeschreibung müssen wir auch die eigentlichen Aktionen abstrahieren. Eine Fahraktion hat viele Wirkungen. Sie ändert nicht nur die Position des Fahrzeuges und seiner Insassen, sondern benötigt Zeit, verbraucht Treibstoff, verursacht Luftverschmutzung und verändert den Agenten (man sagt ja auch, Reisen bildet). In unserer Formulierung berücksichtigen wir nur die Positionsänderung. Außerdem gibt es viele Aktionen, die wir völlig weglassen: das Einschalten des Radios, das Aus-dem-Fenster-Sehen, das Langsamer-fahren-weil-eine-Polizeistreife-hinter-uns-ist usw. Und natürlich geben wir keine Aktionen an, die sich auf dem Niveau „Lenkrad um 3 Grad nach links drehen“ befinden.

Können wir bei der Definition des geeigneten Abstraktionsgrades präziser sein? Stellen Sie sich die abstrakten Zustände und Aktionen vor, die wir als Entsprechung zu großen Mengen detaillierter Weltzustände und detaillierter Aktionsfolgen gewählt haben. Betrachten wir jetzt eine Lösung für das abstrakte Problem: beispielsweise den Pfad von Arad über Sibiu nach Rimnicu Vilcea und Pitesti nach Bukarest. Diese abstrakte Lösung entspricht einer großen Menge detaillierter Pfade. Beispielsweise könnten wir auf der Fahrt zwischen Sibiu und Rimnicu Vilcea das Radio eingeschaltet haben und es für die restliche Reise ausschalten. Die Abstraktion ist *gültig*, wenn wir jede abstrakte Lösung zu einer Lösung der detaillierteren Welt expandieren können; eine hinreichende Bedingung ist, dass es für jeden detaillierten Zustand, zum Beispiel „in Arad“, einen detaillierten Pfad zu einem Zustand gibt, zum Beispiel „in Sibiu“ usw.[5] Die Abstraktion ist *nützlich*, wenn die Ausführung jeder der Aktionen in der Lösung einfacher als das ursprüngliche Problem ist; in diesem Fall sind sie einfach

5 In *Abschnitt 11.2* finden Sie einen vollständigeren Satz von Definitionen und Algorithmen.

genug, dass sie ohne weitere Suche oder Planung durch einen durchschnittlichen Fahragenten ausgeführt werden können. Die Auswahl einer guten Abstraktion bedingt also, dass so viele Details wie möglich entfernt werden, dennoch die Gültigkeit erhalten bleibt und sichergestellt ist, dass die abstrakten Aktionen einfach auszuführen sind. Ginge es nicht um die Fähigkeit, nützlichere Abstraktionen zu erzeugen, würden intelligente Agenten von der realen Welt völlig verdrängt.

3.2 Beispielprobleme

Der problemlösende Ansatz wurde auf einen immensen Bereich von Aufgabenumgebungen angewendet. Wir werden hier einige der bekanntesten Beispiele auflisten und dabei zwischen *Spielproblemen* und *Problemen der realen Welt* unterscheiden. Anhand von **Spielproblemen** sollen verschiedene Methoden zur Problemlösung demonstriert werden. Sie können in einer präzisen, exakten Beschreibung angegeben werden. Das bedeutet, sie lassen sich ganz einfach von den Forschern nutzen, um die Leistung von Algorithmen zu vergleichen. Bei einem **Problem aus der realen Welt** machen sich die Leute über seine Lösung wirklich Gedanken. Derartige Probleme haben normalerweise nicht eine einzige Beschreibung, auf die sich alle geeinigt haben, aber wir können das allgemeine Aussehen ihrer Formulierungen angeben.

3.2.1 Spielprobleme

Das erste Beispiel, das wir hier betrachten, ist die in *Kapitel 2* eingeführte **Staubsaugerwelt** (siehe Abbildung 2.2). Diese kann wie folgt als Problem formuliert werden:

- **Zustände:** Der Zustand wird sowohl durch die Position des Agenten als auch die verschmutzten Orte bestimmt. Der Agent befindet sich an einer von zwei Positionen, die jeweils schmutzig sein können, es aber nicht sein müssen. Es gibt also $2 \times 2^2 = 8$ mögliche Weltzustände. Eine größere Umgebung mit n Positionen umfasst $n \cdot 2^n$ Zustände.
- **Ausgangszustand:** Jeder beliebige Zustand kann als Ausgangszustand festgelegt werden.
- **Aktionen:** In dieser einfachen Umgebung besitzt jeder Zustand lediglich drei Aktionen: *Links*, *Rechts* und *Saugen*.
- **Überführungsmodell:** Die Aktionen haben ihre erwarteten Wirkungen, außer dass die *Links*-Bewegung im äußersten linken Quadrat, die *Rechts*-Bewegung im äußersten rechten Quadrat und *Saugen* auf einem sauberen Quadrat wirkungslos sind. ▶ Abbildung 3.3 zeigt den vollständigen Zustandsraum.
- **Zieltest:** Überprüft, ob alle Quadrate sauber sind.
- **Pfadkosten:** Jeder Schritt kostet 1, die gesamten Pfadkosten entsprechen also der Anzahl der Schritte im Pfad.

Verglichen mit der realen Welt hat dieses Spielproblem diskrete Positionen, diskreten Schmutz, zuverlässige Reinigung – und es wird nie wieder schmutzig, nachdem geputzt wurde. *Kapitel 4* lockert einige dieser Vorgaben.

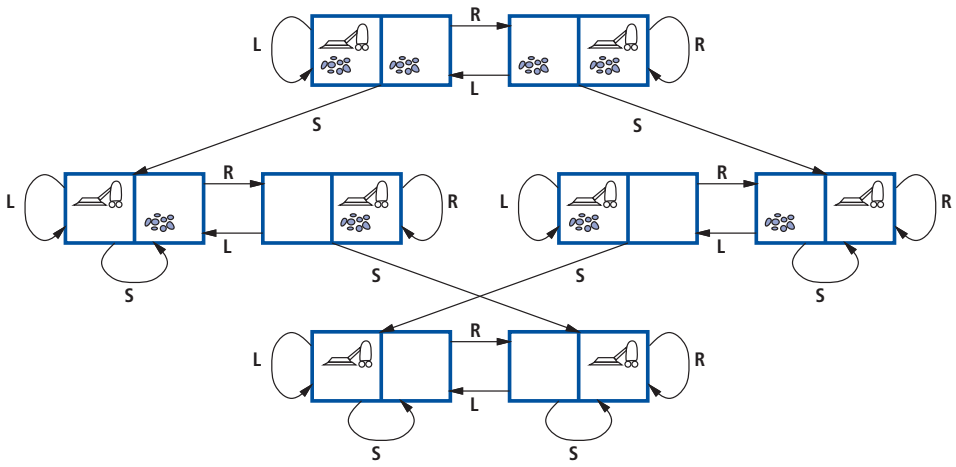


Abbildung 3.3: Der Zustandsraum für die Staubsaugerwelt. Pfeile stehen für Aktionen: L = Links, R = Rechts, S = Saugen.

Das **8-Puzzle**, für das Sie in ▶ Abbildung 3.4 ein Beispiel sehen, besteht aus einem 3×3-Spielbrett mit acht nummerierten Feldern und einem leeren Raum. Ein Feld, das sich neben dem leeren Feld befindet, kann dorthin verschoben werden. Das Ziel dabei ist, einen vorgegebenen Zielzustand wie beispielsweise im rechten Teil der Abbildung gezeigt zu erreichen. Die Standardformulierung sieht wie folgt aus:

- **Zustände:** Eine Zustandsbeschreibung gibt die Position jedes der acht Felder und des Leerfeldes in einem der neun Quadrate an.
- **Ausgangszustand:** Jeder Zustand kann als Ausgangszustand festgelegt werden. Beachten Sie, dass jedes beliebige Ziel von genau der Hälfte der möglichen Ausgangszustände erreicht werden kann (Übung 3.5).
- **Aktionen:** Die einfachste Formulierung definiert die Aktionen als Züge des Leerfeldes *Links*, *Rechts*, *Oben* oder *Unten*. Unterschiedliche Teilmengen dieser Aktionen sind möglich, abhängig davon, wo sich das Leerfeld befindet.
- **Übergangsmodell:** Gibt für einen bestimmten Zustand und eine Aktion den resultierenden Zustand zurück. Wenn wir zum Beispiel *Links* auf den in Abbildung 3.4 dargestellten Ausgangszustand anwenden, hat der resultierende Zustand die 5 und das Leerfeld hat gewechselt.
- **Zieltest:** Überprüft, ob der Zustand mit der in Abbildung 3.4 gezeigten Zielkonfiguration übereinstimmt. (Es sind auch andere Zielkonfigurationen möglich.)
- **Pfadkosten:** Jeder Schritt kostet 1, die Gesamtpfadkosten entsprechen also der Anzahl der Schritte im Pfad.

Welche Abstraktionen haben wir hier angewendet? Die Aktionen werden auf ihre Anfangs- und Endzustände abstrahiert, wobei die Zwischenpositionen ignoriert werden, während das Feld verschoben wird. Die Abstraktion schließt Aktionen aus, wie zum Beispiel das Schütteln des Brettes, wenn sich Felder verklemmt haben, oder das gewaltsame Ausbauen von Feldern mit einem Messer und das erneute Zusammensetzen des Spieles. Übrig bleibt eine Beschreibung der Puzzle-Regeln, wobei wir alle Details der physischen Manipulationen weggelassen haben.

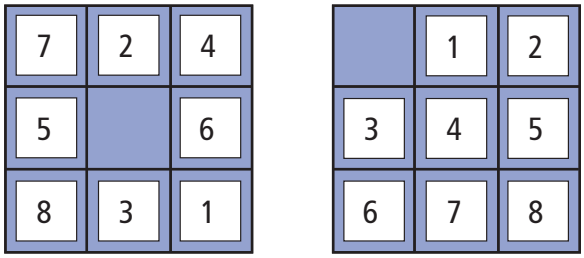


Abbildung 3.4: Ein typisches Beispiel für ein 8-Puzzle.

Das 8-Puzzle gehört zur Familie der **Schiebeblock-Puzzles**, die häufig als Testprobleme für neue Suchalgorithmen in der künstlichen Intelligenz verwendet werden. Diese allgemeine Klasse ist NP-vollständig; man erwartet also nicht, Methoden zu finden, die im ungünstigsten Fall wesentlich besser sind als die in diesem und dem nächsten Kapitel beschriebenen Suchalgorithmen. Das 8-Puzzle hat $9!/2 = 181.440$ erreichbare Zustände und ist einfach zu lösen. Das 15-Puzzle (auf einem Spielbrett mit 4×4) hat etwa 1,3 Billionen Zustände und zufällige Instanzen können durch die besten Suchalgorithmen in ein paar Millisekunden optimal gelöst werden. Das 24-Puzzle (auf einem Spielbrett mit 5×5) hat etwa 10^{25} Zustände und zufällige Instanzen brauchen mehrere Stunden, um es optimal zu lösen.

Das Ziel des **8-Damen-Problems** ist es, acht Damen so auf einem Schachbrett zu platzieren, dass keine der Damen eine andere angreift. (Eine Dame greift jede Figur in derselben Zeile, Spalte oder Diagonalen an.) ▶ Abbildung 3.5 zeigt einen Lösungsversuch, der fehlschlägt: Die Dame in der ganz rechten Spalte wird durch die Dame ganz oben links angegriffen.

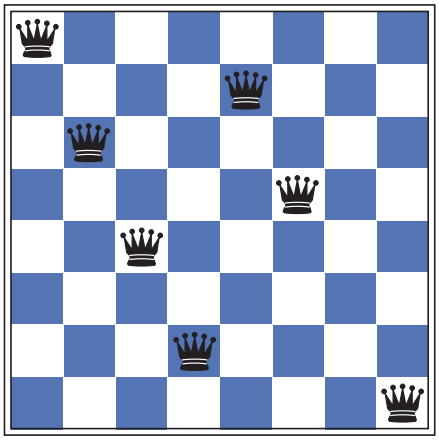


Abbildung 3.5: Versuch einer Lösung für das 8-Damen-Problem. Er schlägt fehl. Die Lösung bleibt Ihnen als Übung überlassen.

Obwohl es für dieses Problem wie auch für die gesamte *n*-Damen-Familie effiziente Spezialalgorithmen gibt, bleibt es ein interessantes Testproblem für Suchalgorithmen. Es gibt hauptsächlich zwei Arten der Formulierung: Eine **inkrementelle Formulierung**

verwendet Operatoren, die die Zustandsbeschreibung erweitern, beginnend mit einem leeren Zustand. Für das 8-Damen-Problem bedeutet das, dass jede Aktion dem Zustand eine Dame hinzufügt. Eine **vollständige Zustandsformulierung** beginnt mit allen acht Damen auf dem Brett und verschiebt sie dort. In jedem Fall sind die Pfadkosten nicht von Interesse, weil nur der endgültige Zustand zählt. Die erste inkrementelle Formulierung, die man ausprobieren könnte, sieht wie folgt aus:

- **Zustände:** Eine beliebige Anordnung von 0 bis 8 Damen auf dem Brett ist ein Zustand.
- **Ausgangszustand:** keine Dame auf dem Brett
- **Aktion:** einem beliebigen Quadrat eine Dame hinzufügen
- **Überführungsmodell:** gibt das Brett mit einer auf das spezifizierte Quadrat hinzugefügten Dame zurück
- **Zieltest:** Es befinden sich acht Damen auf dem Brett und keine davon wird angegriffen.

Bei dieser Formulierung müssen wir $64 \times 63 \times ... \times 57 \approx 1{,}8 \times 10^{14}$ mögliche Sequenzen auswerten. Eine bessere Formulierung würde es verbieten, eine Dame auf einem bereits angegriffenen Quadrat zu platzieren:

- **Zustände:** alle möglichen Anordnungen von n Damen ($0 \leq n \leq 8$), eine pro Spalte in den am weitesten links liegenden n Spalten, wobei keine Dame eine andere angreift.
- **Aktionen:** einem beliebigen Quadrat in der am weitesten links liegenden leeren Spalte eine Dame hinzufügen, sodass diese nicht durch eine andere Dame angegriffen wird.

Diese Formulierung reduziert den 8-Damen-Zustandsraum von $1{,}8 \times 10^{14}$ auf nur noch 2057 und Lösungen lassen sich leicht finden. Für 100 Damen dagegen reduziert sich der Zustandsraum von 10^{400} Zuständen auf etwa 10^{52} Zustände (Übung 3.6) – eine zwar erhebliche Verbesserung, aber immer noch nicht ausreichend, um das Problem handhabbar zu machen. *Abschnitt 4.1* beschreibt die vollständige Zustandsformulierung und *Kapitel 6* stellt einen einfachen Algorithmus vor, der selbst das Millionen-Damen-Problem mit Leichtigkeit löst.

Unser letztes Spielproblem wurde von Donald Knuth (1964) entwickelt. Es veranschaulicht, wie unendliche Zustandsräume entstehen können. Knuth vermutete, dass sich beginnend mit der Zahl 4 durch eine Sequenz von Fakultäts-, Quadratwurzel- und Floor-Operationen (auch Gaußklammer oder Abrundungsfunktion genannt) jede gewünschte positive Ganzzahl erreichen lässt. Zum Beispiel kann 5 aus 4 wie folgt erreicht werden:

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5.$$

Die Problemdefinition ist sehr einfach:

- **Zustände:** positive Zahlen
- **Ausgangszustand:** 4
- **Aktionen:** Fakultäts-, Quadratwurzel- oder Floor-Operation anwenden (Fakultät nur für Ganzzahlen)
- **Überführungsmodell:** wie durch die mathematischen Definitionen der Operationen gegeben
- **Zieltest:** Zustand ist die gewünschte positive Ganzzahl.

Unseres Wissens gibt es keine Schranke hinsichtlich der Größe der Zahl, die in diesem Prozess konstruiert werden kann, um ein gegebenes Ziel zu erreichen – zum Beispiel wird im Ausdruck für 5 die Zahl 620.448.401.733.239.439.360.000 generiert –, sodass der Zustandsraum für dieses Problem unendlich ist. Derartige Zustandsräume treten häufig in Aufgaben auf, die mathematische Ausdrücke, Schaltkreise, Beweise, Programme und andere rekursiv definierte Objekte generieren.

3.2.2 Probleme aus der realen Welt

Wir haben bereits gesehen, wie das **Routensuche-Problem** im Hinblick auf die vorgegebenen Positionen und die Übergänge entlang von Verbindungen zwischen ihnen definiert ist. Algorithmen zur Routensuche werden in zahlreichen Anwendungen eingesetzt. Einige, wie zum Beispiel Websites und Bordsysteme von Kraftfahrzeugen, die die Fahrtrichtungen liefern, sind relativ einfache Erweiterungen des Rumänien-Beispiels. Andere, wie zum Beispiel Routing von Videostreams in Computernetzen, militärische Operationsplanung und Planungssysteme für den Flugreiseverkehr, beinhalten erheblich komplexere Spezifikationen. Sehen Sie sich als Beispiel die Flugverkehrsprobleme an, die von einer Website zur Reiseplanung gelöst werden müssen:

- **Zustände:** Jeder Zustand umfasst offensichtlich eine Position (z.B. einen Flughafen) und die aktuelle Zeit. Da zudem die Kosten einer Aktion (eines Flugabschnittes) von vorherigen Abschnitten, ihren Flugtarifen und ihrem Status als Inlands- oder Auslandsflüge abhängen können, muss der Zustand zusätzliche Informationen über diese „historischen" Aspekte aufzeichnen.
- **Ausgangszustand:** wird durch die Abfrage des Benutzers spezifiziert
- **Aktionen:** Nimm einen beliebigen Flug vom aktuellen Ort in irgendeiner Klasse, der nach der aktuellen Zeit startet und bei Bedarf noch genügend Zeit für den Transfer innerhalb des Flughafens lässt.
- **Übergangsmodell:** Der Zustand nach Absolvieren des Fluges enthält das Ziel des Fluges als aktuelle Position und die Ankunftszeit des Fluges als aktuelle Zeit.
- **Zieltest:** Gelangen wir innerhalb einer vom Benutzer vorgegebenen Zeit ans Ziel?
- **Pfadkosten:** Diese sind vom Flugpreis, der Wartezeit, der Flugzeit, Zoll- und Einwanderungsprozeduren, Sitzqualität, Tageszeit, Flugzeugtyp, Vielflieger-Meilenbonus usw. abhängig.

Kommerzielle Flugbuchungssysteme verwenden eine Problemformulierung dieser Art – mit vielen zusätzlichen Komplikationen, um mit den schwer durchschaubaren Tarifstrukturen klarzukommen, die die Fluggesellschaften eingeführt haben. Jeder erfahrene Reisende weiß jedoch, dass nicht alle Flüge planmäßig verlaufen. Ein wirklich gutes System sollte Alternativpläne berücksichtigen – wie beispielsweise Sicherungsreservierungen auf alternativen Flügen –, und zwar in einem Maße, dass diese entsprechend den Kosten und der Ausfallwahrscheinlichkeit des ursprünglichen Fluges zu rechtfertigen sind.

Touring-Probleme sind eng verwandt mit Problemen der Routensuche, allerdings mit einem wichtigen Unterschied. Betrachten Sie beispielsweise das Problem „Besuche jede in Abbildung 3.2 gezeigte Stadt mindestens einmal, wobei Bukarest sowohl Ausgangs- als auch Endpunkt sein soll". Wie bei der Routensuche entsprechen die Aktionen den Reisen zwischen den benachbarten Städten. Der Zustandsraum unterscheidet

sich jedoch erheblich. Jeder Zustand muss nicht nur die aktuelle Position beinhalten, sondern auch *die Menge der Städte, die der Agent besucht hat*. Der Ausgangszustand wäre also *In*(*Bukarest*), *Besucht*({*Bukarest*}) und ein typischer Zwischenzustand wäre *In*(*Vaslui*), *Besucht*({*Bukarest*, *Urziceni*, *Vaslui*}) und der Zieltest würde überprüfen, ob sich der Agent in Bukarest befindet und alle 20 Städte besucht wurden.

Das **Problem des Handelsreisenden** (**Traveling Salesman Problem**, TSP) ist ein Touring-Problem, wobei jede Stadt genau einmal besucht werden muss. Das Ziel dabei ist, die *kürzeste* Tour zu finden. Das Problem ist als NP-hart bekannt, aber es wurde ein enormer Aufwand getrieben, die Fähigkeiten von TSP-Algorithmen zu verbessern. Neben der Reiseplanung für Handelsreisende wurden diese Algorithmen auch für andere Aufgaben verwendet, wie beispielsweise zur Bewegungsplanung für automatische Leiterplattenbohreinrichtungen und zum Optimieren der Fahrwege von Regalbediengeräten in Lagerhäusern.

Ein **VLSI-Layout** bedingt die Positionierung von Millionen von Bauelementen und Verbindungen auf einem Chip, um Fläche, Schaltverzögerungen und Streukapazitäten zu minimieren und die Produktionsausbeute zu maximieren. Das Layoutproblem kommt nach der logischen Entwurfsphase und ist normalerweise in zwei Teile gegliedert: **Zellen-Layout** und **Kanal-Routing**. Bei einem Zellen-Layout werden die elementaren Komponenten der Schaltung in Zellen gruppiert, die jeweils eine bestimmte Funktion ausführen. Jede Zelle hat einen feststehenden Platzbedarf (Größe und Umriss) und benötigt eine bestimmte Anzahl an Verbindungen zu allen anderen Zellen. Das Ziel dabei ist, die Zellen so auf dem Chip zu platzieren, dass sie sich nicht überlappen und dass ausreichend viel Platz für die verbindenden Drähte ist, die zwischen den Zellen verlaufen sollen. Das Kanal-Routing ermittelt eine bestimmte Route für jeden Draht durch die Lücken zwischen den Zellen. Diese Suchprobleme sind äußerst komplex, doch zweifellos wert, sie zu lösen. Später in diesem Kapitel lernen Sie einige Algorithmen kennen, die in der Lage sind, eine Lösung zu liefern.

Die **Roboternavigation** ist eine Verallgemeinerung des Problems der Routensuche, das wir zuvor beschrieben haben. Statt einer diskreten Menge von Routen kann sich ein Roboter im stetigen Raum mit einer (im Prinzip) unendlichen Menge möglicher Aktionen und Zustände bewegen. Für einen rollenden Roboter, der sich auf einer flachen Oberfläche bewegt, ist der Raum im Wesentlichen zweidimensional. Verfügt der Roboter über Arme und Beine oder Räder, die ebenfalls zu steuern sind, wird der Suchraum mehrdimensional. Man benötigt erweiterte Techniken, um den Suchraum endlich zu machen. Wir werden einige dieser Methoden in *Kapitel 25* betrachten. Neben der Komplexität des Problems müssen reale Roboter auch mit Fehlern der Sensoreneingabe und der Motorsteuerung zurechtkommen.

Die **automatische Montageablaufsteuerung** komplexer Objekte durch einen Roboter wurde zuerst von FREDDY (Michie, 1972) demonstriert. Seither gab es nur wenig Fortschritte, aber es gibt sie, nämlich im Hinblick darauf, wann der Zusammenbau komplizierter Objekte wie beispielsweise elektrischer Motoren wirtschaftlich durchführbar ist. Bei Montageproblemen besteht das Ziel darin, eine Reihenfolge zu finden, in der die Einzelteile eines bestimmten Objektes zusammengebaut werden. Wird die falsche Reihenfolge gewählt, gibt es später in der Sequenz keine Möglichkeit, ein weiteres Teil einzubauen, ohne bereits erledigte Arbeiten noch einmal rückgängig zu machen. Die Überprüfung eines Schrittes in der Sequenz hinsichtlich Realisierbarkeit ist ein schwieriges geometrisches Suchproblem, das eng mit der Roboternavigation verknüpft ist. Die

Ermittlung zulässiger Aktionen ist also der aufwändigste Teil der Montageablaufsteuerung. Jeder praktische Algorithmus muss es vermeiden, den gesamten Zustandsraum auszuwerten, und darf nur einen kleinen Teil davon betrachten. Ein weiteres wichtiges Montageproblem ist der **Proteinentwurf**. Hier wird das Ziel verfolgt, eine Sequenz von Aminosäuren zu finden, die sich zu einem dreidimensionalen Protein falten, das die richtigen Eigenschaften besitzt, um eine Krankheit zu heilen.

3.3 Die Suche nach Lösungen

Nachdem wir einige Probleme formuliert haben, müssen wir sie jetzt lösen. Da eine Lösung eine Aktionssequenz darstellt, betrachten Suchalgorithmen verschiedene mögliche Aktionssequenzen. Die möglichen Aktionssequenzen, die beim Ausgangszustand beginnen, bilden einen **Suchbaum** mit dem Ausgangszustand als Wurzel. Die Zweige sind Aktionen und die **Knoten** entsprechen den Zuständen im Zustandsraum des Problems. ▶ Abbildung 3.6 zeigt die ersten Schritte des wachsenden Suchbaumes, wenn eine Route von Arad nach Bukarest gesucht wird.

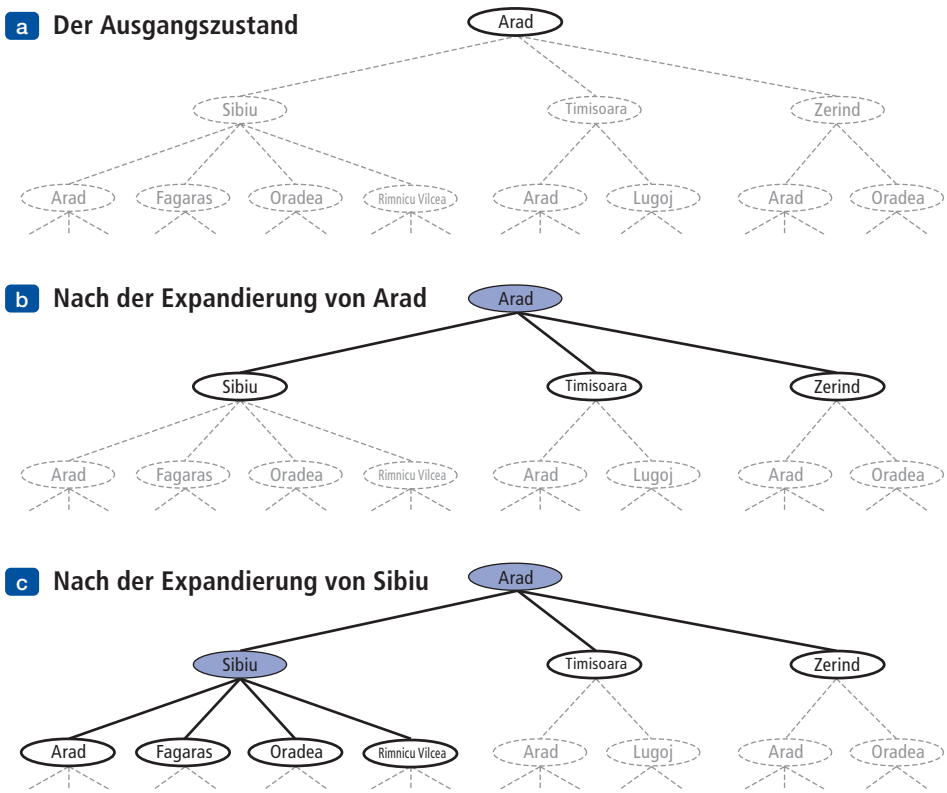


Abbildung 3.6: Partielle Suchbäume für die Ermittlung einer Route von Arad nach Bukarest. Knoten, die expandiert wurden, sind grau schattiert dargestellt; Knoten, die erzeugt, aber noch nicht expandiert wurden, sind fett ausgezeichnet; Knoten, die noch nicht erzeugt wurden, sind mit hellen gestrichelten Linien dargestellt.

Der Wurzelknoten des Baumes entspricht dem Anfangszustand *In*(*Arad*). Im ersten Schritt wird getestet, ob dies ein Zielzustand ist. (Offensichtlich ist das nicht der Fall, aber diese Überprüfung ist wichtig, damit wir Scheinprobleme lösen können, wie beispielsweise „Starte in Arad, gehe nach Arad".) Dann müssen wir ins Auge fassen, verschiedene andere Aktionen zu unternehmen. Dazu **expandieren** wir den aktuellen Zustand. Das heißt, wir wenden jede zulässige Aktion auf den aktuellen Zustand an und **erzeugen** damit eine neue Menge von Zuständen. In diesem Fall fügen wir drei Zweige vom **übergeordneten Knoten** *In*(*Arad*) hinzu, die zu drei neuen **untergeordneten Knoten** führen: *In*(*Arad*), *In*(*Timisoara*) und *In*(*Zerind*). Jetzt müssen wir auswählen, welche dieser drei Möglichkeiten weiter betrachtet werden soll.

Dies ist das Wesen der Suche – es wird jetzt eine Option weiterverfolgt, die anderen werden beiseitegelegt und später betrachtet, falls die erste Suche nicht zu einer Lösung führt. Angenommen, wir wählen zuerst Sibiu aus. Wir prüfen, ob es sich dabei um einen Zielzustand handelt (das ist nicht der Fall), und expandieren den Baum, um *In*(*Arad*), *In*(*Fagaras*), *In*(*Oradea*) und *In*(*RimnicuVilcea*) zu erhalten. Wir können dann einen beliebigen dieser vier Zustände auswählen oder zurückgehen und Timisoara oder Zerind auswählen. Jeder dieser sechs Knoten ist ein **Blattknoten**, d.h. ein Knoten, der keine untergeordneten Knoten im Baum aufweist. Die Menge aller zur Erweiterung an einem bestimmten Punkt verfügbaren Blattknoten wird als **Grenzknoten** bezeichnet. (Viele Autoren sprechen hier von einer **offenen Liste**, die sowohl geografisch nicht ganz so treffend als auch ungenauer ist, da andere Datenstrukturen besser geeignet sind als eine Liste.) In Abbildung 3.6 besteht die Grenze jedes Baumes aus den Knoten mit fetten Umrisslinien.

Das Expandieren der Knoten an der Grenze setzt sich fort, bis entweder eine Lösung gefunden wird oder es keine weiteren Zustände mehr zu expandieren gibt. ▶ Abbildung 3.7 zeigt eine formlose Darstellung des allgemeinen TREE-SEARCH-Algorithmus. Diese Basisstruktur ist allen Suchalgorithmen gemein. Hauptsächlich unterscheiden sich die Algorithmen darin, wie sie auswählen, welcher Zustand als Nächstes zu erweitern ist – das ist die sogenannte **Suchstrategie**.

```
function TREE-SEARCH(problem) returns eine Lösung oder einen Fehler
  Die Grenzknoten mit dem Ausgangszustand von problem initialisieren
  loop do
    if die Grenze ist leer then return Fehler
    Wähle einen Blattknoten aus und entferne ihn aus den Grenzknoten
    if Knoten enthält einen Zielzustand then return die entsprechende Lösung
    Knoten expandieren und der Grenze die resultierenden Knoten hinzufügen


function GRAPH-SEARCH (problem) returns eine Lösung oder einen Fehler
  Die Grenzknoten mit dem Ausgangszustand von problem initialisieren
  Die untersuchte Menge als leer initialisieren
  loop do
    if die Grenze ist leer then return Fehler
    Wähle einen Blattknoten aus und entferne ihn aus den Grenzknoten
    if Knoten enthält einen Zielzustand then return die entsprechende Lösung
    Füge den Knoten zur untersuchten Menge hinzu
    Knoten expandieren und der Grenze die resultierenden Knoten hinzufügen
      Nur, wenn nicht in der Grenze der untersuchten Menge
```

Abbildung 3.7: Eine formlose Beschreibung der allgemeinen Algorithmen für Baum- und Graphensuche. Die in GRAPH-SEARCH fett kursiv markierten Teile sind die erforderlichen Ergänzungen, um wiederholte Zustände zu behandeln.

Bei dem in Abbildung 3.6 gezeigten Suchbaum wird dem aufmerksamen Leser eine Merkwürdigkeit aufgefallen sein: Er schließt den Pfad von Arad nach Sibiu und wieder zurück nach Arad ein! Wir sagen, dass *In*(*Arad*) ein **wiederholter Zustand** im Suchbaum ist, der in diesem Fall durch einen **schleifenförmigen Pfad** generiert wird. Die Betrachtung eines derartigen schleifenförmigen Pfades bedeutet, dass der vollständige Suchbaum für Rumänien *unendlich* ist, da es keine Schranke gibt, wie oft man eine Schleife durchlaufen kann. Andererseits besitzt der Suchraum – die in Abbildung 3.2 gezeigte Karte – nur 20 Zustände. Wie wir in *Abschnitt 3.4* erläutern, können Schleifen dazu führen, dass bestimmte Algorithmen scheitern, und sonst lösbare Probleme unlösbar machen. Erfreulicherweise ist es gar nicht notwendig, schleifenförmige Pfade zu betrachten. Darauf können wir uns nicht nur aus der Intuition heraus verlassen: Da Pfadkosten additiv und Schrittkosten nichtnegativ sind, ist ein schleifenförmiger Pfad zu einem beliebigen Zustand niemals besser als der gleiche Pfad mit entfernter Schleife.

Schleifenpfade sind ein Spezialfall des allgemeinen Konzeptes **redundanter Pfade**, die existieren, wenn es mehr als einen Weg gibt, um von einem Zustand zu einem anderen zu gelangen. Sehen Sie sich die Pfade Arad–Sibiu (140 km lang) und Arad–Zerind–Oradea–Sibiu (297 km lang) an. Offenkundig ist der zweite Pfad redundant – es ist lediglich ein umständlicherer Weg, um zum selben Zustand zu gelangen. Wenn es darum geht, das Ziel zu erreichen, gibt es keinerlei Grund, mehr als einen Pfad zu einem bestimmten Zustand zu verwalten, da jeder Zielzustand, der durch Erweitern des einen Pfades erreichbar ist, auch durch Erweitern des anderen Pfades erreicht werden kann.

In manchen Fällen ist es möglich, das Problem von vornherein so zu definieren, dass sich redundante Pfade vermeiden lassen. Wenn wir zum Beispiel das 8-Damen-Problem (*Abschnitt 3.2.1*) so definieren, dass eine Dame in eine beliebige Spalte gesetzt werden kann, lässt sich jeder Zustand mit n Damen auf $n!$ unterschiedlichen Pfaden erreichen. Formuliert man dagegen das Problem so, dass jede neue Dame in der jeweils am weitesten links befindlichen freien Spalte platziert wird, ist jeder Zustand nur noch über einen einzigen Pfad erreichbar.

In anderen Fällen sind redundante Pfade unvermeidbar. Dazu gehören alle Probleme, in denen die Aktionen reversibel sind, beispielsweise Probleme zur Routensuche und Schiebeblock-Puzzles. Die Routensuche in einem **rechteckigen Gitter** (wie es beispielsweise später für ▶ Abbildung 3.9 verwendet wird) ist ein besonders wichtiges Beispiel in Computerspielen. In einem derartigen Gitter hat jeder Zustand vier Nachfolger. Ein Suchbaum der Tiefe d, der wiederholte Zustände einschließt, hat also 4^d Blätter, wobei es jedoch nur etwa $2d^2$ verschiedene Zustände innerhalb von d Schritten in jedem beliebigen Zustand gibt. Für d=20 ergeben sich dafür etwa eine Billion Knoten, aber nur etwa 800 verschiedene Zustände. Somit kann das Verfolgen redundanter Pfade ein handhabbares Problem zu einem unlösbaren Problem machen. Das gilt selbst für Algorithmen, die in der Lage sind, unendliche Schleifen zu vermeiden.

Tipp

Wie heißt es doch: *Algorithmen, die ihren Verlauf vergessen, sind dazu verdammt, ihn zu wiederholen.* Wenn man sich merkt, wo man war, kann man die Untersuchung redundanter Pfade vermeiden. Dazu erweitern wir den TREE-SEARCH-Algorithmus mit einer als **untersuchte Menge** (oder auch **geschlossene Liste**) bezeichneten Datenstruk-

tur, die sich jeden erweiterten Knoten merkt. Neu generierte Knoten, die mit bereits vorher erzeugten Knoten – in der untersuchten Menge oder der Grenze – übereinstimmen, können verworfen werden, anstatt sie zu den Grenzknoten hinzuzufügen. Abbildung 3.7 zeigt den neuen Algorithmus namens GRAPH-SEARCH. Die speziellen Algorithmen in diesem Kapitel bauen auf diesem allgemeinen Konzept auf.

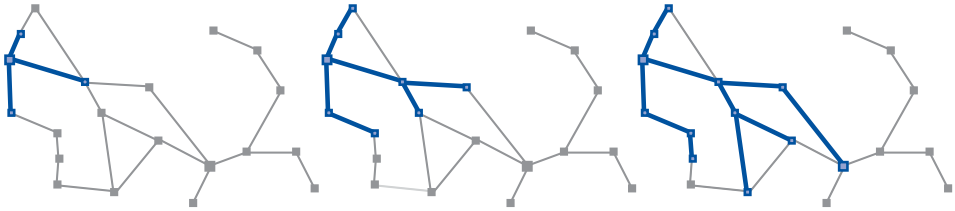


Abbildung 3.8: Eine Sequenz von Suchbäumen, die von einer Graphensuche für das Rumänien-Problem von Abbildung 3.2 generiert wurde. Auf jeder Stufe haben wir jeden Pfad um einen Schritt erweitert. Auf der dritten Stufe ist die nördlichste Stadt (Oradea) zu einer Sackgasse geworden: Ihre beiden Nachfolger wurden bereits über andere Pfade untersucht.

Offenkundig enthält der vom GRAPH-SEARCH-Algorithmus konstruierte Suchbaum höchstens ein Exemplar jedes Zustandes – der Baum wächst also gewissermaßen direkt auf dem Graphen des Zustandsraumes wie in ▶ Abbildung 3.8 gezeigt. Der Algorithmus besitzt eine andere angenehme Eigenschaft: Die Grenze bildet einen **Separator**, d.h., sie trennt den Zustandsgraphen in den untersuchten und den nicht untersuchten Bereich, sodass jeder Pfad vom Ausgangszustand zu einem nicht untersuchten Zustand durch einen Zustand in der Grenze laufen muss. (Falls Ihnen das vollkommen klar erscheint, sollten Sie sich jetzt an Übung 3.14 versuchen.) Abbildung 3.9 veranschaulicht diese Eigenschaft. Da jeder Schritt einen Zustand von der Grenze in den untersuchten Bereich verschiebt, während einige Zustände aus dem nicht untersuchten Bereich in die Grenze wandern, ist zu erkennen, dass der Algorithmus systematisch die Zustände im Zustandsraum einen nach dem anderen untersucht, bis er eine Lösung findet.

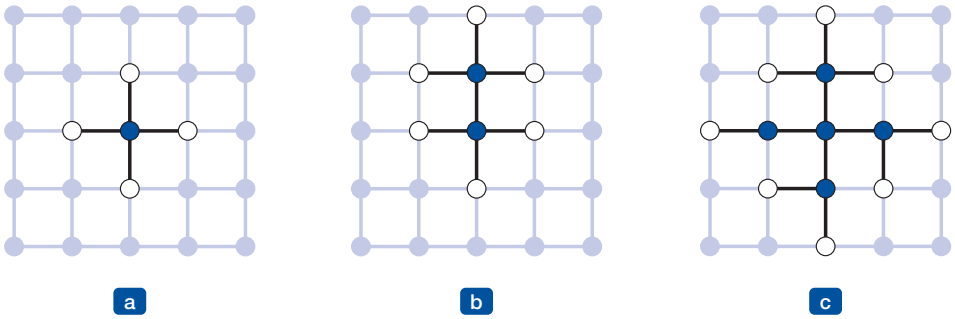


Abbildung 3.9: Die Trennungseigenschaft von GRAPH-SEARCH, veranschaulicht an einem Problem mit rechteckigem Gitternetz. Die Grenze (weiße Knoten) trennt immer den untersuchten Bereich des Zustandsraumes (schwarze Knoten) vom nicht untersuchten Bereich (graue Knoten). In (a) wurde lediglich die Wurzel erweitert. In (b) wurde ein Blattknoten erweitert. In (c) wurden die restlichen Nachfolger des Wurzelknotens im Uhrzeigersinn erweitert.

3.3.1 Infrastruktur für Suchalgorithmen

Suchalgorithmen erfordern eine Datenstruktur, um den zu konstruierenden Suchbaum zu verfolgen. Für jeden Knoten n des Baumes haben wir eine Struktur mit vier Komponenten:

- n.STATE: der Zustand im Zustandsraum, dem der Knoten entspricht;
- n.PARENT: der Knoten im Suchbaum, der diesen Knoten generiert hat;
- n.ACTION: die Aktion, die auf den übergeordneten Knoten angewandt wurde, um den Knoten zu generieren;
- n.PATH-COST: die traditionell mit $g(n)$ bezeichneten Kosten des Pfades vom Ausgangszustand zum Knoten, wie durch die Zeiger des übergeordneten Knotens angegeben ist.

Angesichts der Komponenten für einen übergeordneten Knoten ist es leicht zu sehen, wie die notwendigen Komponenten für einen untergeordneten Knoten zu berechnen sind. Die Funktion CHILD-NODE übernimmt einen übergeordneten Knoten und eine Aktion und gibt den resultierenden untergeordneten Knoten zurück:

```
function CHILD-NODE(problem, parent, action) returns ein Knoten
  return einen Knoten mit
    STATE = problem.RESULT(parent.STATE, action),
    PARENT = parent, ACTION = action,
    PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

▶ Abbildung 3.10 zeigt die Knotendatenstruktur. Die PARENT-Zeiger verketten die Knoten zu einer Baumstruktur. Diese Zeiger erlauben es auch, den Lösungspfad zu erweitern, wenn ein Zielknoten gefunden ist. Die Funktion SOLUTION liefert die Sequenz der Aktionen, die erhalten wird, indem den übergeordneten Zeigern zurück zur Wurzel gefolgt wird.

Bislang haben wir zwischen Knoten und Zuständen nicht sorgfältig unterschieden, doch beim Schreiben von detaillierten Algorithmen ist diese Unterscheidung wichtig. Ein Knoten ist eine Verwaltungsstruktur, um den Suchbaum darzustellen. Ein Zustand entspricht einer Konfiguration der Welt. Knoten befinden sich damit auf bestimmten Pfaden, wie sie durch PARENT-Zeiger definiert werden, während das bei Zuständen nicht der Fall ist.

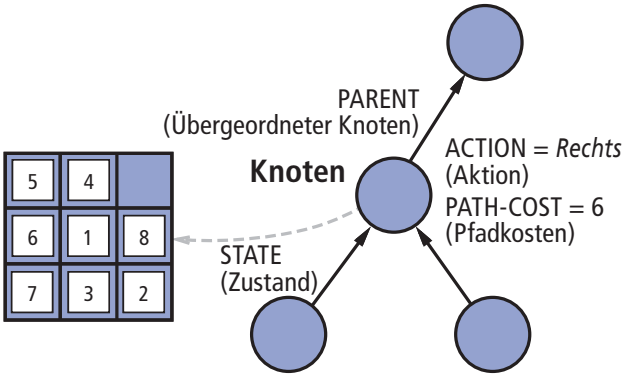


Abbildung 3.10: Knoten sind die Datenstrukturen, aus denen sich der Suchbaum zusammensetzt. Jeder Knoten hat einen übergeordneten Knoten, einen Zustand sowie verschiedene Felder für die Verwaltung. Pfeile verweisen vom untergeordneten zum übergeordneten Knoten.

Darüber hinaus können zwei unterschiedliche Knoten den gleichen Weltzustand enthalten, wenn dieser Zustand über zwei unterschiedliche Suchpfade erzeugt wird.

Nachdem wir nun über Knoten verfügen, brauchen wir einen Platz, wo wir sie unterbringen. Die Grenze muss in einer Form gespeichert werden, dass der Suchalgorithmus leicht den nächsten Knoten auswählen kann, der entsprechend seiner bevorzugten Strategie zu erweitern ist. Die geeignete Datenstruktur hierfür ist eine **Warteschlange** (**Queue**). Die Operationen auf einer Warteschlange sehen folgendermaßen aus:

- EMPTY?(*queue*) gibt nur dann *true* zurück, wenn die Warteschlange keine Elemente mehr enthält.
- POP(*queue*) entfernt das erste Element aus der Warteschlange und gibt es zurück.
- INSERT(*element, queue*) fügt ein Element ein und gibt die resultierende Warteschlange zurück.

Charakterisiert werden Warteschlangen durch die Reihenfolge, in der sie die eingefügten Knoten speichern. Drei gebräuchliche Varianten sind die **FIFO** (First-In, First-Out)-**Warteschlange**, die das älteste Element der Warteschlange zurückgibt, die **LIFO** (Last-In, First-Out)-**Warteschlange** (auch als **Stack** bezeichnet) und die **Prioritätswarteschlange**, die das Element der Warteschlange mit der höchsten Priorität entsprechend einer festgelegten Reihenfolgefunktion zurückgibt.

Die untersuchte Menge lässt sich mit einer Hashtabelle implementieren, um effiziente Prüfungen auf wiederholte Zustände zu ermöglichen. Mit einer guten Implementierung können Einfügen und Nachschlagen in ungefähr konstanter Zeit ausgeführt werden, unabhängig davon, wie viele Zustände gespeichert sind. Bei der Implementierung der Hashtabelle ist auf das richtige Konzept für die Gleichheit zwischen den Zuständen zu achten. Zum Beispiel muss die Hashtabelle im Problem des Handelsreisenden (*Abschnitt 3.2.2*) wissen, dass die Menge der besuchten Städte {*Bukarest*, *Urziceni*, *Vaslui*} gleichbedeutend mit {*Urziceni*, *Vaslui*, *Bukarest*} ist. Manchmal lässt sich dies höchst einfach erreichen, indem darauf bestanden wird, die Datenstrukturen für Zustände in einer bestimmten **kanonischen Form** einzurichten – d.h., logisch äquivalente Zustände sollten auf die gleiche Datenstruktur abgebildet werden. Im Fall von Zuständen, die durch Mengen beschrieben werden, wären zum Beispiel eine Bit-Vektor-Darstellung oder eine sortierte Liste kanonisch, eine unsortierte Liste dagegen nicht.

3.3.2 Leistungsbewertung für die Problemlösung

Bevor wir uns dem Entwurf spezifischer Suchalgorithmen zuwenden, müssen wir zunächst die Kriterien betrachten, die sich zur Auswahl eines zweckmäßigen Algorithmus heranziehen lassen. Die Leistung eines Algorithmus kann auf viererlei Arten bewertet werden:

- **Vollständigkeit:** Findet der Algorithmus garantiert eine Lösung, wenn es eine gibt?
- **Optimalität:** Findet die Strategie die optimale Lösung, wie in *Abschnitt 3.1.1* definiert?
- **Zeitkomplexität:** Wie lange dauert es, eine Lösung zu finden?
- **Speicherkomplexität:** Wie viel Speicher wird für die Suche benötigt?

Zeit- und Speicherkomplexität werden immer im Hinblick auf ein bestimmtes Schwierigkeitsmaß des Problems betrachtet. In der theoretischen Informatik ist das typische Maß die Größe des Zustandsraumgraphen $|V| + |E|$, wobei V die Menge der Ecken (Knoten)

des Graphen und E die Menge der Kanten (Verknüpfungen) ist. Dies ist zweckmäßig, wenn der Graph eine explizite Datenstruktur ist, die als Eingabe an das Suchprogramm übergeben wird. (Ein Beispiel dafür ist die Straßenkarte von Rumänien.) In der künstlichen Intelligenz wird der Graph oftmals implizit durch den Ausgangszustand, die Aktionen und das Übergangsmodell beschrieben und er ist häufig unendlich. Aus diesen Gründen wird die Komplexität in Form von drei Größen ausgedrückt: b, der **Verzweigungsfaktor** (**Branching Factor**) oder die maximale Anzahl der Nachfolger jedes Knotens; d, die **Tiefe** (**depth**) des flachsten Knotens (d.h. die Anzahl der Schritte entlang des Pfades von der Wurzel aus); und m, die maximale Länge eines beliebigen Pfades im Zustandsraum. Zeit wird häufig als Anzahl der während der Suche erzeugten Knoten gemessen, und der Speicherplatz als maximale Anzahl der im Hauptspeicher abgelegten Knoten. Meistens beschreiben wir Zeit- und Platzkomplexität für die Suche in einem Baum. Für einen Graphen hängt die Antwort davon ab, wie „redundant" die Pfade im Zustandsraum sind.

Um die Effektivität eines Suchalgorithmus zu beurteilen, können wir einfach nur die **Suchkosten** betrachten – die in der Regel von der Zeitkomplexität abhängig sind, aber auch einen Term für die Speicherbenutzung beinhalten können – oder wir können die **Gesamtkosten** verwenden, die die Suchkosten und die Pfadkosten der gefundenen Lösung kombinieren. Für das Problem, eine Route von Arad nach Bukarest zu finden, sind die Suchkosten die Zeitdauer, die die Suche benötigt, und die Lösungskosten die Gesamtlänge des Pfades in Kilometern. Um also die Gesamtkosten zu berechnen, müssen wir Kilometer und Millisekunden addieren. Es gibt zwar keinen offiziellen „Umrechnungskurs" zwischen den beiden, aber es könnte in diesem Fall sinnvoll sein, Kilometer in Millisekunden umzuwandeln, indem man eine Schätzung der Durchschnittsgeschwindigkeit des Autos verwendet (weil sich der Agent hauptsächlich um die Zeit kümmert). Das ermöglicht es dem Agenten, eine optimale Abwägung zu finden, ab wann eine weitere Berechnung zur Ermittlung eines kürzeren Pfades kontraproduktiv wird. *Kapitel 16* greift das allgemeinere Problem auf, Kompromisse zwischen verschiedenen Gütern zu finden.

3.4 Uninformierte Suchstrategien

Dieser Abschnitt beschäftigt sich mit mehreren Suchstrategien, die wir unter der Überschrift der **uninformierten Suche** (auch als **blinde Suche** bezeichnet) finden. Der Begriff bedeutet, dass die Strategien keine zusätzlichen Informationen über Zustände besitzen außer den in der Problemdefinition vorgegebenen. Alles, was sie tun können, ist, Nachfolger zu erzeugen und einen Zielzustand von einem Nichtzielzustand zu unterscheiden. Die einzelnen Strategien unterscheidet man nach der *Reihenfolge*, in der sie die Knoten erweitern. Strategien, die wissen, ob ein Nichtzielzustand „vielversprechender" als ein anderer ist, werden auch als **informierte Suche** oder **heuristische Suche** bezeichnet. *Abschnitt 3.5* beschäftigt sich ausführlich mit ihnen.

3.4.1 Breitensuche (Breadth-first)

Die **Breitensuche** ist eine einfache Strategie, wobei der Wurzelknoten als erster expandiert wird, dann alle Nachfolger des Wurzelknotens und anschließend *deren* Nachfolger usw. Im Allgemeinen werden zuerst alle Knoten einer bestimmten Tiefe im Suchbaum expandiert, bevor Knoten in der nächsten Ebene expandiert werden.

```
function BREADTH-FIRST-SEARCH (problem) returns eine Lösung oder einen Fehler

  node ← ein Knoten mit STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← eine FIFO-Warteschlange mit node als einzigem Element
  explored ← eine leere Menge
  loop do
    if EMPTY?(frontier) then return Fehler
    node ← POP(frontier) /* Wählt flachsten Knoten in frontier aus */
    node.STATE zu explored hinzufügen
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE ist nicht in explored oder frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Abbildung 3.11: Breitensuche auf einem Graphen.

Die Breitensuche ist eine Instanz des allgemeinen GRAPH-SEARCH-Algorithmus (Abbildung 3.7), in dem der *flachste* nicht erweiterte Knoten zum Expandieren ausgewählt wird. Dies lässt sich sehr einfach mithilfe einer FIFO-Warteschlange für die Grenzknoten erreichen. Somit rücken neue Knoten (die immer tiefer als ihre übergeordneten Knoten liegen) in der Warteschlange nach hinten und ältere Knoten, die flacher als die neuen Knoten sind, werden zuerst erweitert. Gegenüber dem allgemeinen GRAPH-SEARCH-Algorithmus gibt es eine geringfügige Optimierung: Der Algorithmus wendet den Zieltest auf jeden Knoten an, wenn er den Knoten *erzeugt*, und nicht, wenn er den Knoten zur Erweiterung auswählt. Mehr zu dieser Entscheidung folgt später in diesem Kapitel, wenn es um die Zeitkomplexität geht. Außerdem verwirft der Algorithmus gemäß der allgemeinen Vorlage für die Graphensuche jeden neuen Pfad zu einem Zustand, der bereits in der Menge der Grenzknoten oder untersuchten Knoten enthalten ist. Man kann leicht erkennen, dass jeder derartige Pfad mindestens so tief wie ein bereits gefundener sein muss. Somit hat die Breitensuche immer den flachsten Pfad zu jedem Knoten in den Grenzknoten.

▶ Abbildung 3.11 gibt den Pseudocode an und ▶ Abbildung 3.12 zeigt den Fortschritt einer Suche in einem einfachen Binärbaum.

Wie lässt sich die Breitensuche gemäß den vier Kriterien aus dem vorherigen Abschnitt beurteilen? Es ist leicht zu sehen, dass sie *vollständig* ist – wenn sich der flachste Zielknoten auf einer endlichen Tiefe d befindet, findet ihn die Breitensuche letztendlich, nachdem sie alle flacheren Knoten erzeugt hat (vorausgesetzt, der Verzweigungsfaktor b ist endlich). Sobald ein Zielknoten erzeugt wird, wissen wir, dass es sich um den flachsten Zielknoten handelt, weil alle flacheren Knoten bereits generiert sein und den Zieltest nicht bestanden haben müssen. Der *flachste* Zielknoten ist nicht unbedingt der *optimale*; technisch betrachtet ist die Breitensuche optimal, wenn die Pfadkosten eine nichtfallende Funktion der Knotentiefe darstellen. Im häufigsten derartigen Szenario haben alle Aktionen die gleichen Kosten.

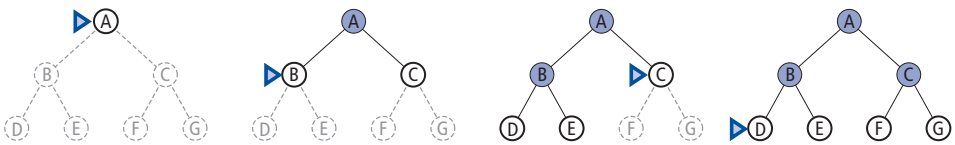


Abbildung 3.12: Breitensuche für einen einfachen Binärbaum. In jeder Phase ist der nächste zu expandierende Knoten durch einen Pfeil gekennzeichnet.

Bisher waren alle Informationen über die Breitensuche positiv. Hinsichtlich Zeit und Platz trifft das allerdings nicht ganz zu. Stellen Sie sich die Suche in einem einheitlichen Baum vor, in dem jeder Zustand b Nachfolger hat. Die Wurzel des Suchbaumes erzeugt b Knoten auf der ersten Ebene, die jeweils b weitere Knoten erzeugen, was auf der zweiten Ebene insgesamt b^2 Knoten ergibt. Jeder *dieser* Knoten erzeugt b weitere Knoten, womit wir auf der dritten Ebene b^3 Knoten erhalten usw. Nehmen wir nun an, die Lösung befindet sich in der Tiefe d. Im ungünstigsten Fall handelt es sich um den auf dieser Ebene zuletzt generierten Knoten. Die Gesamtzahl der erzeugten Knoten beträgt damit:

$$b + b^2 + b^3 + \ldots + b^d = O(b^{d+1})$$

(Würde der Algorithmus den Zieltest auf Knoten anwenden, wenn er sie erweitert, und nicht, wenn er sie generiert, wäre bereits die gesamte Ebene der Knoten bei Tiefe d erweitert, bevor das Ziel erkannt ist, und die Zeitkomplexität würde $O(b^{d+1})$ sein.)

Zur Platzkomplexität: Für jede Art von Graphensuche, die jeden erweiterten Knoten in der untersuchten Menge speichert, liegt die Platzkomplexität immer innerhalb eines Faktors b der Zeitkomplexität. Speziell gilt für die Breitensuche, dass jeder generierte Knoten im Hauptspeicher verbleibt. Mit $O(b^{d-1})$ Knoten in der *untersuchten* Menge und $O(b^d)$ Knoten im Grenzbereich ergibt sich eine Platzkomplexität von $O(b^d)$, d.h., sie wird von der Größe der Grenzknoten dominiert. Der Übergang zu einer Baumsuche würde nicht viel Platz sparen und in einem Zustandsraum mit vielen redundanten Pfaden könnte der Wechsel viel Zeit kosten. Eine exponentielle Komplexitätsschranke wie etwa $O(b^d)$ ist erschreckend. ▶ Abbildung 3.13 zeigt, warum das so ist. Sie listet für verschiedene Werte der Lösungstiefe d den Zeit- und Speicherbedarf einer Breitensuche mit einem Verzweigungsfaktor $b = 10$ auf. Für die Werte in der Tabelle wird angenommen, dass sich 1 Million Knoten pro Sekunde erzeugen lassen und dass ein Knoten 1000 Byte Speicherplatz benötigt. Diese Annahmen treffen auf viele Suchprobleme ungefähr zu (eventuell um einen Faktor 100 mehr oder weniger), wenn sie auf einem modernen PC ausgeführt werden.

Tiefe	Knoten	Zeit	Speicher
2	1100	0,11 Millisekunden	107 Kilobyte
4	111100	11 Millisekunden	10,6 Megabyte
6	10^7	1,1 Sekunden	1 Gigabyte
8	10^9	2 Minuten	103 Gigabyte
10	10^{11}	3 Stunden	10 Terabyte
12	10^{13}	13 Tag	1 Petabyte
14	10^{15}	3,5 Jahre	99 Petabyte
16	10^{16}	350 Jahre	10 Exabyte

Abbildung 3.13: Zeit- und Speicheranforderungen für die Breitensuche. Die hier gezeigten Zahlen setzen einen Verzweigungsfaktor von b = 10 voraus; 10.000 Knoten/Sekunde; 1000 Byte/Knoten.

Aus Abbildung 3.13 können wir zwei Dinge lernen: Erstens *sind die Speicheranforderungen für die Breitensuche ein größeres Problem als die Ausführungszeit*. Bei einem wichtigen Problem mit der Suchtiefe 12 kann man schon einmal 13 Tage Wartezeit in Kauf nehmen, doch bringt kein Personalcomputer die benötigen Petabyte Hauptspeicher mit. Zum Glück gibt es andere Strategien, die weniger Speicher benötigen.

Tipp

Die zweite Lektion ist, dass die Zeitanforderungen immer noch einen wesentlichen Faktor darstellen. Wenn Ihr Problem eine Lösung bei Tiefe 16 hat, dauert es (unseren Annahmen nach) rund 350 Jahre, bis die Breitensuche (oder eine andere uninformierte Suche) sie gefunden hat. Im Allgemeinen *können Suchprobleme mit exponentieller Komplexität außer für die kleinsten Instanzen nicht von uninformierten Methoden gelöst werden*.

Tipp

3.4.2 Suche mit einheitlichen Kosten (Uniform-cost-Suche)

Die Breitensuche ist optimal, wenn alle Schrittkosten gleich sind, weil sie immer den *flachsten* nicht expandierten Knoten expandiert. Mit einer einfachen Erweiterung finden wir einen Algorithmus, der optimal für jede Schrittkostenfunktion ist. Anstatt den flachsten Knoten zu expandieren, expandiert die **Suche mit einheitlichen Kosten** den Knoten n mit den *geringsten Pfadkosten* $g(n)$. Dazu werden die Grenzknoten, wie in ▶ Abbildung 3.14 gezeigt wird, als Prioritätswarteschlange sortiert nach g gespeichert.

Außer der Sortierung der Warteschlange nach den Pfadkosten gibt es zwei andere signifikante Unterschiede zur Breitensuche. Erstens wendet der Algorithmus den Zieltest auf einen Knoten an, wenn er ihn *zum Expandieren auswählt* (wie im generischen GRAPH-SEARCH-Algorithmus, den Abbildung 3.7 zeigt), und nicht, wenn er den Knoten *erzeugt*. Das geschieht deshalb, weil der erste Zielknoten, der generiert wird, auf einem suboptimalen Pfad liegen kann. Der zweite Unterschied betrifft einen zusätzlichen Test für den Fall, dass ein besserer Pfad zu einem momentan im Grenzbereich befindlichen Knoten gefunden wird.

```
function UNIFORM-COST-SEARCH(problem) gibt eine Lösung oder einen Fehler zurück

  node ← ein Knoten mit STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← eine Prioritätswarteschlange, sortiert nach PATH-COST mit node
  als einzigem Element
  explored ← eine leere Menge
  loop do
    if EMPTY?(frontier) then return Fehler
    node ← POP(frontier) /* wählt Knoten mit geringsten Kosten in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    node.STATE zu explored hinzufügen
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE ist nicht in explored oder frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE ist in frontier mit höheren PATH-COST then
        diesen frontier-Knoten durch child ersetzen
```

Abbildung 3.14: Suche mit einheitlichen Kosten auf einem Graphen. Der Algorithmus ist identisch mit dem allgemeinen Graphensuchalgorithmus in Abbildung 3.7, verwendet aber eine Prioritätswarteschlange und führt zusätzlich einen Test aus, falls ein kürzerer Pfad zu einem Grenzknotenzustand entdeckt wird. Die Datenstruktur für *frontier* muss einen effizienten Test auf Mitgliedschaft unterstützen, sodass sie die Fähigkeiten einer Prioritätswarteschlange und einer Hashtabelle kombinieren sollte.

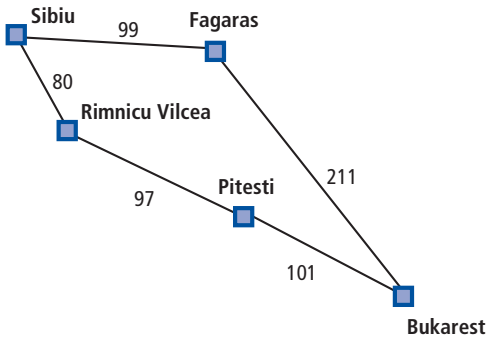


Abbildung 3.15: Ausgewählter Teil des Rumänien-Zustandsraumes, um die Suche mit einheitlichen Kosten zu veranschaulichen.

Im Beispiel von ▶ Abbildung 3.15 besteht das Problem darin, von Sibiu nach Bukarest zu gelangen. Hier kommen die beiden Modifikationen ins Spiel. Die Nachfolger von Sibiu sind Rimnicu Vilcea und Fagaras mit Kosten von 80 bzw. 99. Der Knoten mit den niedrigsten Kosten ist Rimnicu Vilcea. Er wird als Nächstes erweitert, wobei Pitesti hinzukommt und die Kosten 80 + 97 = 177 betragen. Damit ist nun Fagaras der Knoten mit den niedrigsten Kosten. Er wird also erweitert, wodurch Bukarest hinzukommt und sich die Kosten nun auf 99 + 211 = 310 belaufen. Zwar ist jetzt ein Zielknoten generiert worden, doch fährt die Suche mit einheitlichen Kosten fort, wählt Pitesti zur Erweiterung aus und fügt einen zweiten Pfad nach Bukarest mit den Kosten 80 + 97 + 101 = 278 hinzu. Der Algorithmus prüft nun, ob dieser neue Pfad besser als der alte ist. Trifft das zu, wird der alte Pfad verworfen. Die Suche wählt also Bukarest – mit g-Kosten von 278 – zum Expandieren aus und gibt die Lösung zurück.

Tipp

Es ist leicht zu sehen, dass die Suche mit einheitlichen Kosten im Allgemeinen optimal ist. Zuerst ist festzustellen, dass der optimale Pfad zu einem Knoten n gefunden wurde, wenn die Suche mit einheitlichen Kosten diesen Knoten zur Erweiterung ausgewählt hat. (Wäre dies nicht der Fall, müsste es gemäß der Trennungseigenschaft der Graphensuche (siehe Abbildung 3.9) einen anderen Grenzknoten n' auf dem optimalen Pfad vom Startknoten zu n geben. Der Definition nach hätte n' geringere g-Kosten als n und wäre demnach zuerst ausgewählt worden.) Und da Schrittkosten nichtnegativ sind, werden Pfade niemals kürzer, wenn Knoten hinzukommen. Aus diesen beiden Tatsachen zusammen lässt sich ableiten, dass die Suche mit einheitlichen Kosten die Knoten in der Reihenfolge ihrer optimalen Pfadkosten erweitert. Folglich muss der erste Zielknoten, der zur Erweiterung ausgewählt ist, die optimale Lösung darstellen.

Bei der Suche mit einheitlichen Kosten geht es nicht um die *Anzahl* der Schritte, die ein Pfad aufweist, sondern nur um deren Gesamtkosten. Aus diesem Grund gerät die Suche in eine Endlosschleife, wenn es einen Pfad mit einer unendlichen Sequenz von 0-Kosten-Aktionen gibt – zum Beispiel eine *NoOp*-Aktion.[6] Vollständigkeit ist garantiert, solange die Kosten jedes Schrittes den Wert einer kleinen positiven Konstante ϵ überschreiten.

Da die Suche mit einheitlichen Kosten durch Pfadkosten und nicht durch Tiefe gesteuert wird, lässt sich ihre Komplexität nicht einfach im Hinblick auf b und d charakterisieren.

6 *NoOp* oder „No Operation“ ist eine Anweisung in Assemblersprache, die nichts bewirkt.

Stattdessen bezeichnen wir mit C^* die Kosten der optimalen Lösung[7] und nehmen an, dass jede Aktion mindestens ε kostet. Dann ist die schlechteste Zeit- und Speicherkomplexität des Algorithmus gleich $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$, was sehr viel größer als b^d sein kann. Das liegt daran, dass die Suche mit einheitlichen Kosten große Bäume kleiner Schritte untersuchen kann, bevor sie Pfade untersucht, die große und vielleicht nützliche Schritte beinhalten. Wenn alle Schrittkosten gleich sind, ist $b^{1+\lfloor C^*/\varepsilon \rfloor}$ einfach gleich b^{d+1}. In diesem Fall ist die Suche mit einheitlichen Kosten vergleichbar mit der Breitensuche, außer dass die Breitensuche anhält, sobald sie ein Ziel generiert, während die Suche mit einheitlichen Kosten alle Knoten an der Tiefe des Zieles daraufhin überprüft, ob einer der Knoten geringere Kosten aufweist. Somit hat die Suche mit einheitlichen Kosten grundsätzlich mehr zu tun, indem sie Knoten an Tiefe d unnötigerweise erweitert.

3.4.3 Tiefensuche (Depth-first)

Die **Tiefensuche** (**Depth-first**) expandiert immer den *tiefsten* Knoten im aktuellen Grenzbereich des Suchbaumes. ▶ Abbildung 3.16 zeigt den Fortschritt dieser Suche. Die Suche geht unmittelbar auf die tiefste Ebene des Suchbaumes, wo die Knoten keinen Nachfolger haben. Wenn diese Knoten expandiert werden, werden sie aus dem Grenzbereich entfernt, sodass sich die Suche dann zum nächst tieferen Knoten „rettet“, der noch nicht erkundete Nachfolger aufweist.

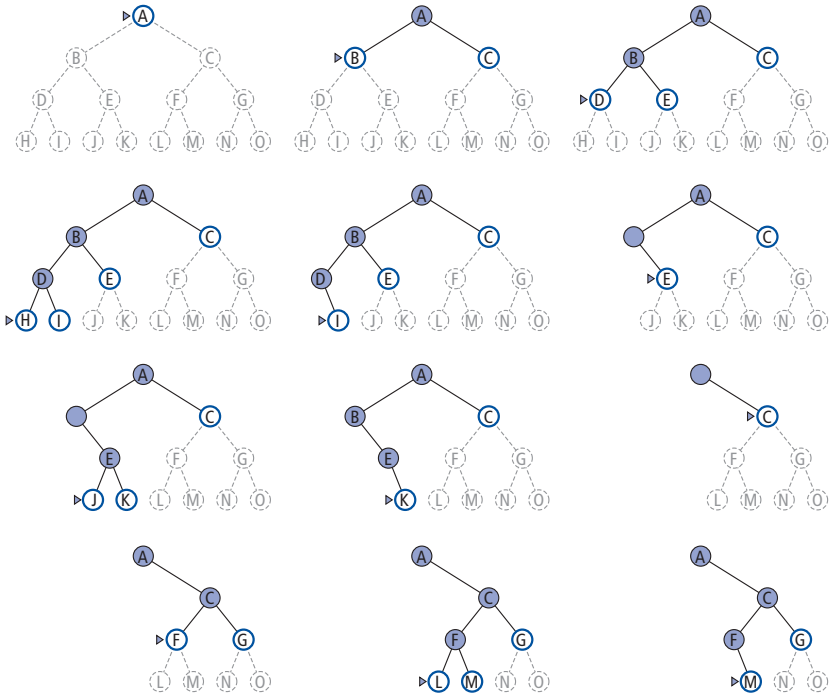


Abbildung 3.16: Tiefensuche in einem Binärbaum. Der nicht untersuchte Bereich ist hellgrau dargestellt. Untersuchte Knoten ohne Nachkommen im Grenzbereich werden aus dem Speicher entfernt. Knoten auf der Tiefe 3 haben keine Nachfolger und *M* ist der einzige Zielknoten.

7 Hier und im gesamten Buch bezeichnet das Sternchen in C^* einen optimalen Wert für C.

Die Tiefensuche ist eine Instanz des Graphensuchalgorithmus gemäß Abbildung 3.7. Während die Breitensuche eine FIFO-Warteschlange verwendet, arbeitet die Tiefensuche mit einer LIFO-Warteschlange. Eine LIFO-Warteschlange bedeutet, dass der jeweils zuletzt generierte Knoten zum Erweitern ausgewählt wird. Dies muss der tiefste nicht erweiterte Knoten sein, da er um eine Ebene tiefer als sein übergeordneter Knoten liegt – der seinerseits der tiefste nicht erweiterte Knoten war, als er ausgewählt wurde.

Als Alternative zur Implementierung im Stil von GRAPH-SEARCH ist es üblich, die Tiefensuche mit einer rekursiven Funktion zu implementieren, die sich für jeden ihrer untergeordneten Knoten der Reihe nach selbst aufruft. (▶ Abbildung 3.17 zeigt einen rekursiven Depth-first-Algorithmus, der eine Begrenzung für die Tiefe aufweist.)

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns eine Lösung oder Fehler/
Cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns eine Lösung oder Fehler/
Cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
    cutoff occurred? ← false
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      result ← RECURSIVE-DLS(child, problem, limit - 1)
      if result = cutoff then cutoff occurred? ← true
      else if result ≠ failure then return result
    if cutoff occurred? then return cutoff else return failure
```

Abbildung 3.17: Eine rekursive Implementierung der tiefenbeschränkten Suche.

Die Eigenschaften der Tiefensuche hängen stark davon ab, ob die Version der Graphensuche oder der Baumsuche verwendet wird. Die Version der Graphensuche, die wiederholte Zustände und redundante Pfade vermeidet, ist vollständig in endlichen Zustandsräumen, da sie letztlich jeden Knoten erweitert. Dagegen ist die Version der Baumsuche nicht vollständig – zum Beispiel folgt der Algorithmus in Abbildung 3.6 unaufhörlich der Schleife Arad–Sibiu–Arad–Sibiu. Die Tiefensuche lässt sich ohne zusätzliche Speicherkosten so modifizieren, dass der Algorithmus neue Zustände gegen diejenigen im Pfad von der Wurzel aus zum aktuellen Knoten überprüft. Dies vermeidet Endlosschleifen in endlichen Zustandsräumen, unterdrückt jedoch nicht die Vermehrung von redundanten Pfaden. In unendlichen Zustandsräumen scheitern beide Versionen, wenn ein unendlicher Nichtzielpfad angetroffen wird. Zum Beispiel würde die Tiefensuche im 4-Problem von Knuth den Fakultätsoperator endlos anwenden.

Aus ähnlichen Gründen sind beide Versionen nicht optimal. Zum Beispiel untersucht die Tiefensuche in Abbildung 3.16 den linken Teilbaum selbst dann, wenn Knoten *C* ein Zielknoten ist. Wäre Knoten *J* ebenfalls ein Zielknoten, würde die Tiefensuche ihn als Lösung anstelle von *C* zurückgeben, was eine bessere Lösung wäre. Somit ist die Tiefensuche nicht optimal.

Die Zeitkomplexität der Tiefensuche bei Graphen ist durch die Größe des Zustandsraumes begrenzt (die natürlich unendlich sein kann). Dagegen kann eine Tiefensuche bei Bäumen sämtliche $O(b^m)$ Knoten im Suchbaum erzeugen, wobei m die maximale Tiefe eines beliebigen Knotens ist. Dieser Wert kann wesentlich größer als die Größe

des Zustandsraumes sein. Beachten Sie, dass m selbst sehr viel größer als d (die Tiefe der flachsten Lösung) sein kann und unendlich ist, wenn der Baum unbeschränkt ist.

Bislang scheint die Tiefensuche keinen klaren Vorteil gegenüber der Breitensuche zu haben. Weshalb also beschäftigen wir uns hier damit? Der Grund ist die Platzkomplexität. Für eine Graphensuche gibt es keinen Vorteil, doch eine Tiefensuche als Baumsuche braucht nur einen einzigen Pfad von der Wurzel zu einem Blattknoten zusammen mit den übrigen nicht erweiterten gleich geordneten Knoten für jeden Knoten auf dem Pfad zu speichern. Nachdem ein Knoten expandiert wurde, kann er aus dem Speicher entfernt werden, sobald alle seine Nachkommen vollständig erkundet wurden (siehe Abbildung 3.16). Für einen Zustandsraum mit Verzweigungsfaktor b und einer maximalen Tiefe m benötigt die Tiefensuche nur $O(bm)$ Knoten. Mit den gleichen Annahmen wie für Abbildung 3.13 und vorausgesetzt, dass Knoten mit derselben Tiefe wie der Zielknoten keinen Nachfolger haben, zeigt sich, dass für die Tiefensuche lediglich 156 KB statt 10 Exabyte bei einer Tiefe von $d = 16$ erforderlich sind – ein 7 Billionen Mal geringerer Speicherplatz. Dies hat zur Akzeptanz der Tiefensuche für Bäume als Hauptinstrument in vielen Bereichen der KI geführt, einschließlich Problemen unter Rand- und Nebenbedingungen (*Kapitel 6*), Erfüllbarkeitsproblemen in der Aussagenlogik (*Kapitel 7*) und logischer Programmierung (*Kapitel 9*). Im Rest dieses Abschnittes konzentrieren wir uns hauptsächlich auf die Tiefensuche für Bäume.

Eine Variante der Tiefensuche ist die sogenannte **Backtracking-Suche**, die noch weniger Speicher benötigt. (Siehe *Kapitel 6* für weitere Details.) Beim Backtracking wird jeweils nur ein einziger Nachfolger erzeugt und nicht alle Nachfolger auf einmal; jeder partiell expandierte Knoten merkt sich, welcher Nachfolger als Nächstes zu erzeugen ist. Auf diese Weise wird nur $O(m)$ Speicher statt $O(bm)$ Speicher benötigt. Die Backtracking-Suche ermöglicht einen weiteren Trick zur Speicherersparnis (und Zeitersparnis): die Idee, einen Nachfolger zu erzeugen, indem man die aktuelle Zustandsbeschreibung direkt *modifiziert*, anstatt sie zuerst zu kopieren. Damit reduzieren sich die Speicheranforderungen auf nur eine einzige Zustandsbeschreibung und $O(m)$ Aktionen. Damit dies funktioniert, müssen wir in der Lage sein, Veränderungen rückgängig zu machen, wenn wir zurückgehen, um den nächsten Nachfolger zu erzeugen. Für Probleme mit großen Zustandsbeschreibungen, wie beispielsweise bei Robotermontage, sind diese Techniken entscheidend für den Erfolg.

3.4.4 Tiefenbeschränkte Suche (Depth-limited)

Das prekäre Scheitern der Tiefensuche in unendlichen Zustandsräumen lässt sich überwinden, indem man eine Tiefensuche mit vorgegebener Tiefenbegrenzung ℓ bereitstellt. Das bedeutet, dass die Knoten der Tiefe ℓ behandelt werden, als hätten sie keine Nachfolger. Dieser Ansatz wird auch als **tiefenbeschränkte Suche** (**Depth-limited-Suche**) bezeichnet. Die Tiefenbegrenzung löst das Problem unendlich langer Pfade. Leider führt sie auch eine zusätzliche Quelle der Unvollständigkeit ein, wenn wir $\ell < d$ wählen, d.h., das flachste Ziel liegt unterhalb der Tiefenbegrenzung. (Das ist wahrscheinlich, wenn d unbekannt ist.) Die tiefenbeschränkte Suche ist außerdem nicht optimal, wenn wir $\ell > d$ wählen. Ihre Zeitkomplexität ist $O(b^\ell)$, die Speicherkomplexität ist $O(bl)$. Die Tiefensuche kann als Sonderfall der tiefenbeschränkten Suche mit $\ell = \infty$ betrachtet werden.

Manchmal lassen sich Tiefenbegrenzungen aus Kenntnissen über das Problem ableiten. Zum Beispiel sind auf der Landkarte von Rumänien 20 Städte verzeichnet. Wenn es also eine Lösung gibt, wissen wir, dass sie im ungünstigsten Fall die Länge 19 auf-

weist; demnach ist $\ell = 19$ eine mögliche Wahl. Doch wenn wir die Landkarte wirklich sorgfältig studieren, zeigt sich, dass jede Stadt von jeder anderen Stadt in höchstens neun Schritten erreicht werden kann. Diese Zahl, auch als **Durchmesser** des Zustandsraumes bezeichnet, bietet eine bessere Tiefenbegrenzung, die zu einer effizienteren tiefenbeschränkten Suche führt. Für die meisten Probleme kennen wir jedoch keine gute Tiefenbegrenzung, bevor wir das Problem nicht gelöst haben.

Die tiefenbeschränkte Suche kann als einfache Variante der allgemeinen Baum- oder Graphensuchalgorithmen implementiert werden. Alternativ lässt sie sich als einfacher rekursiver Algorithmus wie in Abbildung 3.17 gezeigt implementieren. Beachten Sie, dass die tiefenbeschränkte Suche mit zwei Fehlertypen terminieren kann: Der Standardfehlerwert (*failure*) zeigt an, dass es keine Lösung gibt; der *cutoff*-Wert zeigt an, dass es keine Lösung innerhalb der Tiefenbegrenzung gibt.

3.4.5 Iterativ vertiefende Tiefensuche (iterative deepening depth-first search)

Die **iterativ vertiefende Suche** (oder iterativ vertiefende Tiefensuche) ist eine allgemeine Strategie, die häufig in Kombination mit der Tiefensuche für Bäume verwendet wird, die die beste Tiefenbegrenzung ermittelt. Dazu erhöht sie schrittweise die Begrenzung – zuerst 0, dann 1, dann 2 usw. –, bis ein Ziel gefunden ist. Das passiert, wenn die Tiefenbegrenzung d erreicht, die Tiefe des flachsten Zielknotens. ▶ Abbildung 3.18 zeigt den zugehörigen Algorithmus. Die iterative Vertiefung kombiniert die Vorteile der Tiefen- und der Breitensuche. Wie bei der Tiefensuche sind ihre Speicheranforderungen recht bescheiden, genauer gesagt $O(bd)$. Wie die Breitensuche ist sie vollständig, wenn der Verzweigungsfaktor endlich ist, und optimal, wenn die Pfadkosten eine nicht fallende Funktion der Knotentiefe sind. ▶ Abbildung 3.19 zeigt vier Iterationen von ITERATIVE-DEEPENING-SEARCH für einen binären Suchbaum, wobei die Lösung in der vierten Iteration gefunden wird.

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns eine Lösung oder Fehler
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

Abbildung 3.18: Der Algorithmus für die iterativ vertiefende Suche, der wiederholt eine tiefenbeschränkte Suche mit steigenden Begrenzungen anwendet. Er terminiert, sobald eine Lösung gefunden ist oder wenn die tiefenbeschränkte Suche einen Fehler zurückgibt und damit anzeigt, dass keine Lösung existiert.

Die Suche durch iterative Vertiefung mag wie Verschwendung aussehen, weil die Zustände mehrfach erzeugt werden. Es stellt sich jedoch heraus, dass dies nicht sehr aufwändig ist. Der Grund dafür ist, dass sich bei einem Suchbaum mit demselben (oder fast demselben) Verzweigungsfaktor auf jeder Ebene die meisten Knoten auf der untersten Ebene befinden, sodass es kaum eine Rolle spielt, wenn die oberen Ebenen mehrfach erzeugt werden. Bei einer iterativ vertiefenden Suche werden die Knoten auf der untersten Ebene (Tiefe d) einmal erzeugt, die auf der zweiten Ebene von unten zweimal usw., bis zu den der Wurzel untergeordneten Knoten, die d-mal erzeugt werden. Somit berechnet sich die Gesamtzahl der im ungünstigsten Fall erzeugten Knoten zu N(iterativ vertiefende Suche) $= d(b) + (d-1)b^2 + \ldots + (1)b^d$.

Das bedeutet eine Zeitkomplexität von $O(b^d)$ – asymptotisch das Gleiche wie bei der Breitensuche. Zusätzliche – wenn auch nicht sehr hohe – Kosten entstehen für das mehrfache Erzeugen der oberen Ebenen. Zum Beispiel ergeben sich für $b = 10$ und $d = 5$ die Zahlen:

$$N(\text{iterativ vertiefende Suche}) = 0 + 400 + 3000 + 20000 + 100000 = 123450$$

$$N(\text{Breitensuche}) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100$$

Tipp

Wenn Sie wirklich über das Wiederholen der Wiederholung besorgt sind, können Sie als Hybridvariante eine Breitensuche ausführen, bis fast der gesamte Speicher aufgebraucht ist, und dann mit iterativ vertiefender Suche von allen Grenzknoten aus arbeiten. *Im Allgemeinen ist die iterative Vertiefung die bevorzugte uninformierte Suchmethode, wenn es einen großen Suchraum gibt und die Tiefe der Lösung nicht bekannt ist.*

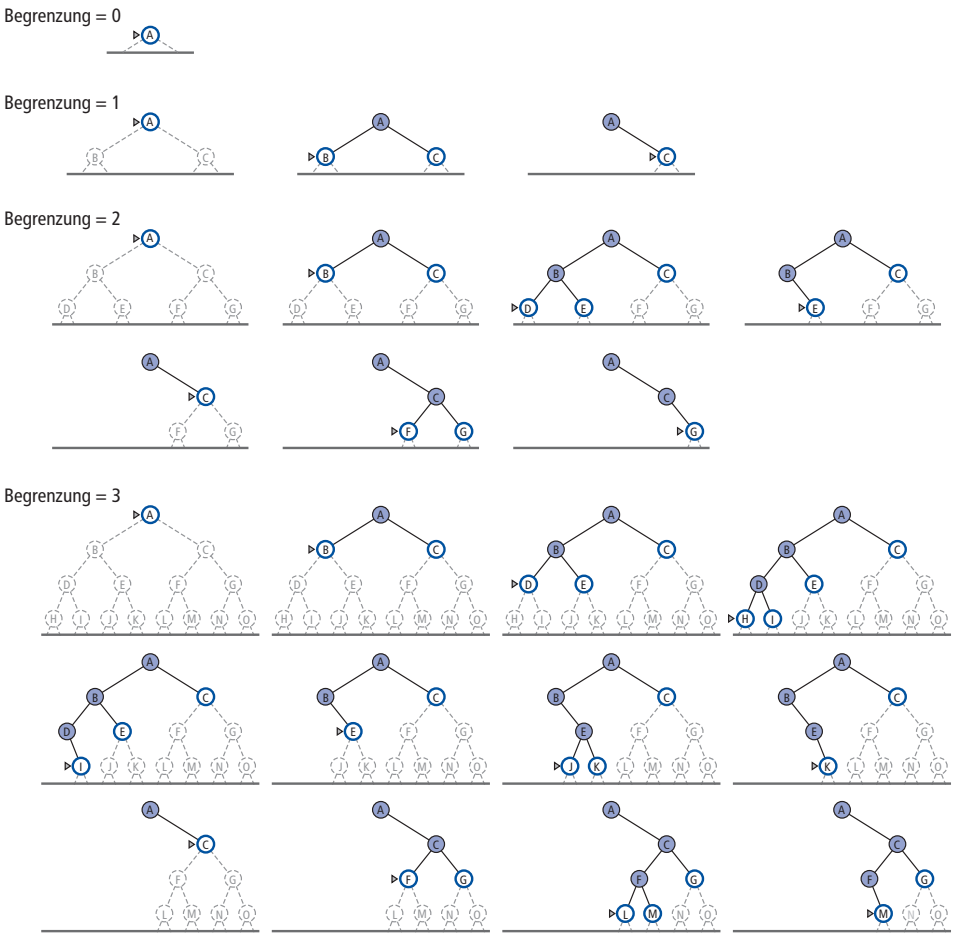


Abbildung 3.19: Vier Iterationen der iterativ vertiefenden Suche für einen Binärbaum.

Die iterativ vertiefende Suche ist analog zur Breitensuche, weil sie in jedem Iterationsschritt eine vollständige Ebene neuer Knoten auswertet, bevor sie zur nächsten Ebene weitergeht. Es scheint sinnvoll zu sein, ein iteratives Analogon zur Suche mit einheitlichen Kosten zu entwickeln, wobei die Optimalitätsgarantien des letztgenannten Algorithmus geerbt und gleichzeitig seine Speicheranforderungen vermieden werden. Die Idee dabei ist, statt steigender Tiefenbegrenzungen steigende Pfadkostenbegrenzungen zu verwenden. Der resultierende Algorithmus, auch als **iterativ verlängernde Suche** (**iterative lengthening**) bezeichnet, wird in Übung 3.18 untersucht. Leider zeigt sich, dass bei iterativer Verlängerung ein wesentlicher Mehraufwand im Vergleich zur Suche mit einheitlichen Kosten entsteht.

3.4.6 Bidirektionale Suche

Der bidirektionalen Suche liegt die Idee zugrunde, zwei gleichzeitige Suchvorgänge auszuführen – einen vorwärts vom Ausgangszustand aus, den anderen rückwärts vom Ziel aus – und darauf zu hoffen, dass sich die beiden Suchprozesse in der Mitte treffen (▶ Abbildung 3.20). Die Motivation dabei ist, dass $b^{d/2} + b^{d/2}$ sehr viel kleiner als b^d ist, oder bezogen auf die Abbildung, dass die Fläche der beiden kleinen Kreise kleiner als die Fläche eines großen Kreises ist, dessen Mittelpunkt am Start liegt und dessen Radius bis zum Ziel verläuft.

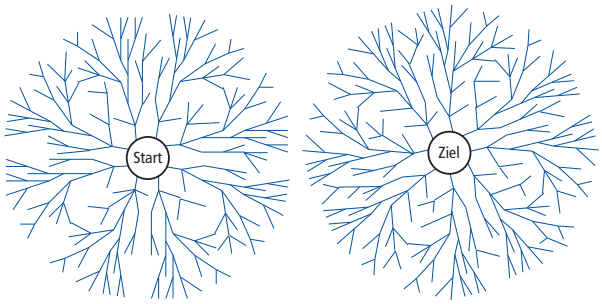


Abbildung 3.20: Schematische Ansicht einer bidirektionalen Suche, die erfolgreich ist, wenn eine Verzweigung vom Startknoten auf eine Verzweigung vom Zielknoten trifft.

Die bidirektionale Suche wird implementiert, indem man den Zieltest durch einen Test ersetzt, ob die Grenzknoten der beiden Suchen eine Schnittmenge bilden; in diesem Fall wurde eine Lösung gefunden. (Die erste derartige gefundene Lösung ist nicht unbedingt optimal, selbst wenn beide Suchvorgänge als Breitensuche laufen. Deshalb ist mit einer zusätzlichen Suche sicherzustellen, dass es keine andere Abkürzung gibt.) Der Test kann stattfinden, wenn jeder Knoten erzeugt oder zur Expandierung ausgewählt wird, und benötigt in Verbindung mit einer Hashtabelle eine konstante Zeit. Hat ein Problem beispielsweise die Lösungstiefe $d = 6$ und in jeder Richtung läuft eine Breitensuche für jeweils einen Knoten, treffen sich die beiden Suchen im ungünstigsten Fall, wenn sie sämtliche Knoten bei Tiefe 3 generiert haben. Für $b = 10$ bedeutet das insgesamt 2.220 erstellte Knoten im Vergleich zu 1.111.110 Knoten für eine Standardbreitensuche. Die Zeitkomplexität der bidirektionalen Suche mit Breitensuchen in beiden Richtungen ist also $O(b^{d/2})$. Die Speicherkomplexität ist ebenfalls $O(b^{d/2})$. Diese Größe lässt sich um rund die Hälfte verringern, wenn eine der beiden Suchen als iterative Tie-

fensuche realisiert wird, doch muss mindestens einer der Grenzbereiche im Hauptspeicher stehen, um den Schnittmengentest durchführen zu können. Diese Speicheranforderung ist die größte Schwäche der bidirektionalen Suche.

Die reduzierte Zeitkomplexität macht die bidirektionale Suche attraktiv, aber wie suchen wir rückwärts? Das ist nicht ganz so einfach, wie es sich anhört. Die **Vorgänger** eines Zustandes x sollen alle diejenigen Zustände sein, die x als Nachfolger haben. Für die bidirektionale Suche ist eine Methode zur Berechnung der Vorgänger erforderlich. Wenn sich sämtliche Aktionen im Zustandsraum umkehren lassen, sind die Vorgänger von x einfach ihre Nachfolger. Für andere Fälle kann wesentlich mehr Raffinesse erforderlich sein.

Was ist nun mit „Ziel“ gemeint, wenn wir „rückwärts vom Ziel aus“ suchen? Für das 8-Puzzle und für die Suche nach einer Route in Rumänien gibt es nur einen Zielzustand, sodass die Rückwärtssuche der Vorwärtssuche recht ähnlich ist. Gibt es mehrere explizit aufgelistete Zielzustände – beispielsweise die beiden schmutzfreien Zielzustände in Abbildung 3.3 –, können wir einen neuen Dummy-Zielzustand konstruieren, dessen unmittelbare Vorgänger alle eigentlichen Zielzustände sind. Wenn aber das Ziel eine abstrakte Beschreibung ist, wie zum Beispiel im n-Damenproblem, dass „keine Dame eine andere Dame attackiert“, ist es schwierig, die bidirektionale Suche zu verwenden.

3.4.7 Vergleich uninformierter Suchstrategien

▶ Abbildung 3.21 vergleicht die Suchstrategien im Hinblick auf die vier Bewertungskriterien, die *Abschnitt 3.3.2* eingeführt hat. Dieser Vergleich gilt für Versionen mit Baumsuche. Die Versionen mit Graphensuche unterscheiden sich vor allem darin, dass die Tiefensuche für endliche Zustandsräume vollständig ist und dass die Platz- und Zeitkomplexitäten durch die Größe des Zustandsraumes beschränkt sind.

Kriterium	Breitensuche	Einheitliche Kosten	Tiefensuche	Beschränkte Tiefensuche	Iterative Vertiefung	Bidirektional (falls möglich)
Vollständig?	Ja[a]	Ja[a, b]	Nein	Nein	Ja[a]	Ja[a, d]
Zeit	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Speicher	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(bm)$	$O(b?)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Ja[c]	Ja	Nein	Nein	Ja[c]	Ja[c, d]

Abbildung 3.21: Bewertung der Suchstrategien. b ist der Verzweigungsfaktor, d die Tiefe der flachsten Lösung, m die maximale Tiefe des Suchbaumes, ℓ die Tiefenbegrenzung. Die hochgestellten Symbole haben folgende Bedeutungen: [a]: vollständig, wenn b endlich ist; [b]: vollständig, wenn die Schrittkosten $\geq \varepsilon$ für jedes positive ε sind; [c]: optimal, wenn alle Schrittkosten identisch sind; [d]: wenn beide Richtungen eine Breitensuche verwenden.

3.5 Informierte (heuristische) Suchstrategien

Dieser Abschnitt zeigt, wie eine **informierte Suche** – die neben der Definition des eigentlichen Problems auch problemspezifisches Wissen nutzt – Lösungen effizienter als eine uninformierte Strategie findet.

Der allgemeine Ansatz, den wir hier betrachten wollen, ist die sogenannte **Bestensuche** (**Best-First**). Die Bestensuche ist eine Instanz des allgemeinen TREE-SEARCH- oder GRAPH-SEARCH-Algorithmus und wählt einen Knoten auf Basis einer **Evaluierungsfunktion** $f(n)$ zur Expandierung aus. Die Evaluierungsfunktion wird als Kostenabschätzung konstruiert, sodass der Knoten mit der *geringsten* Bewertung zuerst erweitert wird. Die Implementierung der Breitensuche als Graphensuche ist identisch zur Suche mit einheitlichen Kosten (Abbildung 3.14), außer dass f anstelle von g verwendet wird, um die Prioritätswarteschlange zu ordnen.

Die Wahl von f bestimmt die Suchstrategie. (Wie zum Beispiel Übung 3.22 zeigt, umfasst die Bestensuche die Tiefensuche als Spezialfall.) Die meisten Algorithmen der Bestensuche beinhalten als Komponente von f eine mit $h(n)$ bezeichnete **Heuristikfunktion**:

$h(n)$ = geschätzte Kosten für billigsten Pfad vom Zustand an Knoten n zu einem Zielzustand

(Zwar übernimmt die Funktion $h(n)$ einen *Knoten* als Eingabe, doch hängt sie im Unterschied zu $g(n)$ nur vom *Zustand* an diesem Knoten ab.) Beispielsweise könnte man in Rumänien die Kosten für den billigsten Pfad von Arad nach Bukarest über die Luftliniendistanz zwischen Arad und Bukarest abschätzen.

Heuristikfunktionen sind die gebräuchlichste Form, dem Suchalgorithmus zusätzliches Wissen über das Problem mitzuteilen. Wir werden in *Abschnitt 3.6* genauer auf Heuristiken eingehen. Hier betrachten wir sie als beliebige, nichtnegative, problemspezifische Funktionen, mit einer Einschränkung: Wenn n ein Zielknoten ist, dann gilt $h(n) = 0$. Der restliche Abschnitt beschreibt zwei Möglichkeiten, heuristische Informationen für die Durchführung der Suche zu verwenden.

3.5.1 „Gierige" Bestensuche (Greedy Best-first)

Die **„gierige" Bestensuche**[8] (**Greedy Best-first**) versucht, den Knoten zu expandieren, der dem Ziel am nächsten liegt, mit der Begründung, dass dies wahrscheinlich schnell zu einer Lösung führt. Damit bewertet der Algorithmus die Knoten nur mithilfe der Heuristikfunktion, d.h. $f(n) = h(n)$.

Sehen Sie sich nun an, wie dies für die Routensuchprobleme in Rumänien funktioniert. Als Heuristik verwenden wir die **Luftliniendistanz** und bezeichnen sie als h_{LLD}. Ist das Ziel Bukarest, brauchen wir die Luftliniendistanzen zu Bukarest, die in ▶ Abbildung 3.22 gezeigt sind. Beispielsweise ist $h_{LLD}(In(Arad)) = 366$. Beachten Sie, dass sich die Werte für h_{LLD} nicht aus der eigentlichen Problembeschreibung berechnen lassen. Darüber hin-

8 In der ersten Auflage haben wir von der „gierigen" Suche gesprochen; andere Autoren bezeichnen sie als Bestensuche. Unsere verallgemeinerte Verwendung des letztgenannten Begriffes folgt *Pearl (1984)*.

aus ist eine gewisse Erfahrung notwendig, um zu wissen, dass h_{LLD} etwas über die tatsächlichen Straßendistanzen aussagt und damit eine sinnvolle Heuristik darstellt.

Arad	366	**Mehadia**	241
Bukarest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Dobreta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

Abbildung 3.22: Werte für h_{LLD} – Luftliniendistanzen nach Bukarest.

▶ Abbildung 3.23 zeigt den Ablauf bei einer „gierigen“ Bestensuche unter Verwendung von h_{LLD}, um einen Pfad von Arad nach Bukarest zu finden. Der erste Knoten, der von Arad aus expandiert wird, ist Sibiu, weil er näher an Bukarest liegt als Zerind oder Timisoara. Der nächste zu expandierende Knoten ist Fagaras, weil er am nächsten liegt. Fagaras wiederum erzeugt Bukarest, das Ziel. Für dieses spezielle Problem findet die „gierige“ Bestensuche unter Verwendung von h_{LLD} eine Lösung, ohne je einen Knoten zu expandieren, der nicht auf dem Lösungspfad liegt; damit sind die Suchkosten minimal. Allerdings ist die Suche nicht optimal: Der Pfad über Sibiu und Fagaras nach Bukarest ist 32 km länger als der Pfad über Rimnicu Vilcea und Pitesti. Daraus ist ersichtlich, warum der Algorithmus „gierig“ heißt – er versucht in jedem Schritt, dem Ziel so nahe wie möglich zu kommen.

Vergleichbar mit der Tiefensuche ist die gierige Bestensuche für Bäume selbst in einem endlichen Zustandsraum unvollständig. Betrachten Sie das Problem, von Iasi nach Fagaras zu gelangen. Die Heuristik schlägt vor, Neamt sollte zuerst expandiert werden, weil es am nächsten bei Fagaras liegt, aber das ist eine Sackgasse. Die Lösung ist, zuerst Vaslui anzusteuern – ein Schritt, der der Heuristik nach eigentlich weiter vom Ziel entfernt liegt – und dann weiter nach Urziceni, Bukarest und Fagaras zu fahren. Der Algorithmus findet diese Lösung allerdings niemals. Denn wenn er Neamt erweitert, kommt Iasi zurück in den Grenzbereich. Iasi ist näher an Fagaras als Vaslui und somit wird Iasi erneut erweitert, was zu einer Endlosschleife führt. (Die Version der Graphensuche *ist* in endlichen Zustandsräumen vollständig, in unendlichen jedoch nicht.) Im ungünstigsten Fall ist die Zeit- und Platzkomplexität für die Version der Baumsuche $O(b^m)$, wobei m die maximale Tiefe des Suchraumes darstellt. Mit einer guten Heuristikfunktion kann die Komplexität jedoch wesentlich reduziert werden. Wie weit diese Reduzierung möglich ist, hängt vom jeweiligen Problem und der Qualität der Heuristik ab.

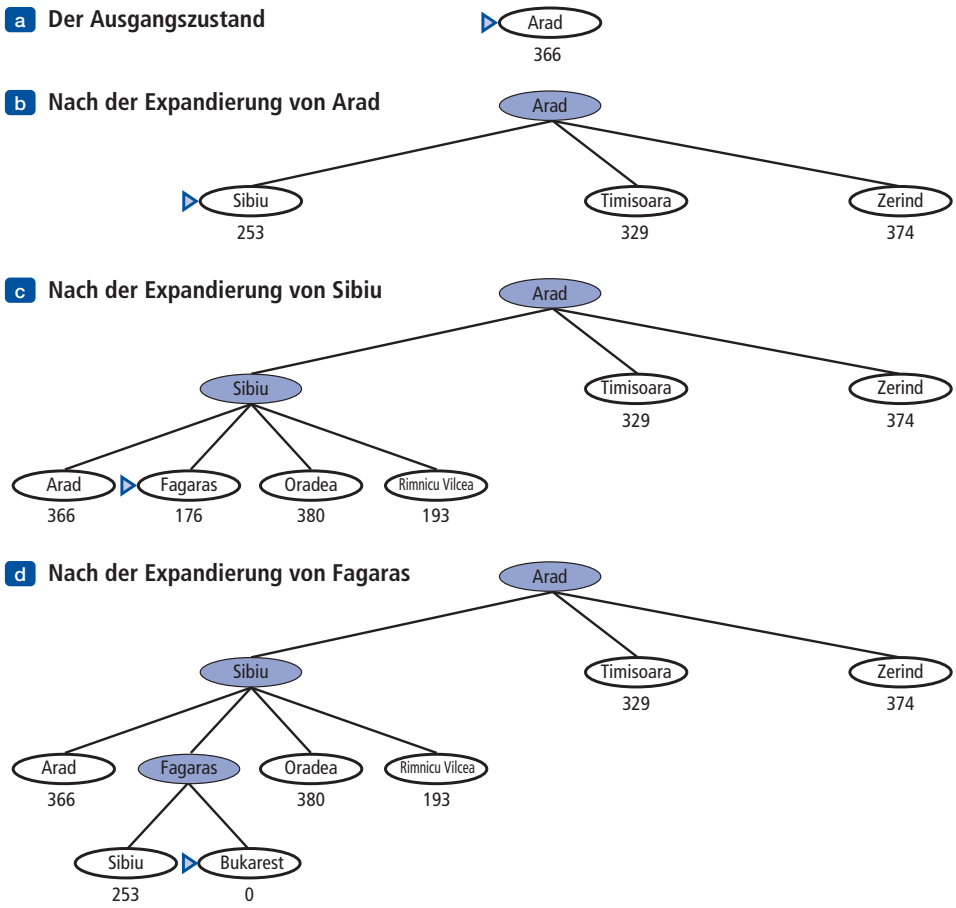


Abbildung 3.23: Phasen der Pfadsuche nach Bukarest in einer „gierigen" Bestensuche, wobei die Luftliniendistanz-Heuristik h_{LLD} angewendet wird. Die Knoten sind mit ihren h-Werten beschriftet.

3.5.2 A*-Suche: Minimierung der geschätzten Gesamtkosten für die Lösung

Die bekannteste Form der Bestensuche ist die sogenannte **A*-Suche**. Sie wertet Knoten aus, indem sie $g(n)$, die Kosten zur Erreichung des Ziels, und $h(n)$, die Kosten, um von dem Knoten zum Ziel zu gelangen, verknüpft:

$$f(n) = g(n) + h(n).$$

Weil $g(n)$ die Pfadkosten vom Ausgangsknoten angibt und $h(n)$ die geschätzten Kosten des billigsten Pfades von n zum Ziel darstellt, erhalten wir:

$$f(n) = \text{geschätzte Kosten für die billigste Lösung durch } n.$$

Wenn wir also versuchen, die billigste Lösung zu finden, ist es sinnvoll, zuerst den Knoten mit dem geringsten Wert von $g(n) + h(n)$ auszuprobieren. Es stellt sich heraus, dass diese Strategie mehr als nur sinnvoll ist: Vorausgesetzt, die Heuristikfunktion $h(n)$ erfüllt bestimmte Bedingungen, ist die A*-Suche sowohl vollständig als auch

optimal. Der Algorithmus ist identisch mit UNIFORM-COST-SEARCH, außer dass A* mit $g + h$ anstelle von g arbeitet.

Bedingungen für Optimalität: Zulässigkeit und Konsistenz

Als erste Bedingung für Optimalität fordern wir, dass $h(n)$ eine **zulässige Heuristik** ist. Dabei handelt es sich um eine Heuristik, die die Kosten, um das Ziel zu erreichen, *niemals überschätzt*. Da $g(n)$ die tatsächlichen Kosten darstellt, um n entlang des aktuellen Pfades zu erreichen, und $f(n) = g(n) + h(n)$ gilt, folgt unmittelbar, dass $f(n)$ die tatsächlichen Kosten einer Lösung entlang des Pfades durch n nie überschätzt.

Zulässige Heuristiken sind von Haus aus optimistisch, da sie für die Lösung des Problems von geringeren Kosten ausgehen, als tatsächlich anfallen. Ein offensichtliches Beispiel für eine zulässige Heuristik ist die Luftliniendistanz h_{LLD}, die wir angewendet haben, um nach Bukarest zu gelangen. Die Luftliniendistanz ist zulässig, weil der kürzeste Pfad zwischen zwei Punkten eine Gerade ist, sodass die Gerade keine Überschätzung sein kann. ▶ Abbildung 3.24 zeigt den Ablauf einer A*-Baumsuche nach Bukarest. Die Werte von g werden aus den Schrittkosten in Abbildung 3.2 berechnet und die Werte für h_{LLD} stammen aus Abbildung 3.22. Beachten Sie insbesondere, dass Bukarest erstmalig in Schritt (e) im Grenzbereich erscheint, aber nicht zur Expandierung ausgewählt wird, weil seine f-Kosten (450) höher als die von Pitesti (417) liegen. Man kann dies auch so ausdrücken, dass es eine Lösung durch Pitesti geben könnte, deren Kosten höchstens 417 betragen, sodass der Algorithmus nicht nach einer Lösung sucht, die 450 kostet.

Eine zweite, etwas strengere Bedingung, die sogenannte **Konsistenz** (manchmal auch als **Monotonie** bezeichnet), ist nur erforderlich für Anwendungen von A* auf die Graphensuche.[9] Eine Heuristik $h(n)$ ist konsistent, wenn für jeden Knoten n und jeden durch eine Aktion a erzeugten Nachfolger n' von n die geschätzten Kosten für das Erreichen des Zieles von n aus nicht größer als die Schrittkosten sind, um nach n' zu gelangen, plus der geschätzten Kosten, das Ziel von n' aus zu erreichen:

$$h(n) \leq c(n, a, n') + h(n').$$

Dies ist eine Form der allgemeinen **Dreiecksungleichung**, die besagt, dass jede Seite eines Dreiecks nicht länger als die Summe der beiden anderen Seiten sein kann. Hier wird das Dreieck durch n, n' und das Ziel G_n, das n am nächsten ist, gebildet. Für eine zulässige Heuristik ist diese Ungleichung unbedingt sinnvoll: Gäbe es eine Route von n nach G_n über n', die billiger als $h(n)$ wäre, würde dies die Eigenschaft verletzen, dass $h(n)$ eine untere Schranke für die Kosten ist, um G_n zu erreichen.

Es lässt sich recht einfach zeigen (Übung 3.32), dass jede konsistente Heuristik auch zulässig ist. Konsistenz ist demzufolge eine strengere Forderung als Zulässigkeit, doch ist es keine triviale Aufgabe, zulässige, aber nicht konsistente Heuristiken zu entwickeln. Alle in diesem Kapitel beschriebenen zulässigen Heuristiken sind auch konsistent. Betrachten Sie beispielsweise h_{LLD}. Wir wissen, dass die allgemeine Dreiecksungleichheit erfüllt ist, wenn jede Seite nach der Luftliniendistanz gemessen wird und die Luftliniendistanz zwischen n und n' nicht größer als $c(n, a, n')$ ist. Damit ist h_{LLD} eine konsistente Heuristik.

9 Mit einer zulässigen, aber inkonsistenten Heuristik erfordert A* zusätzlichen Verwaltungsaufwand, um Optimalität sicherzustellen.

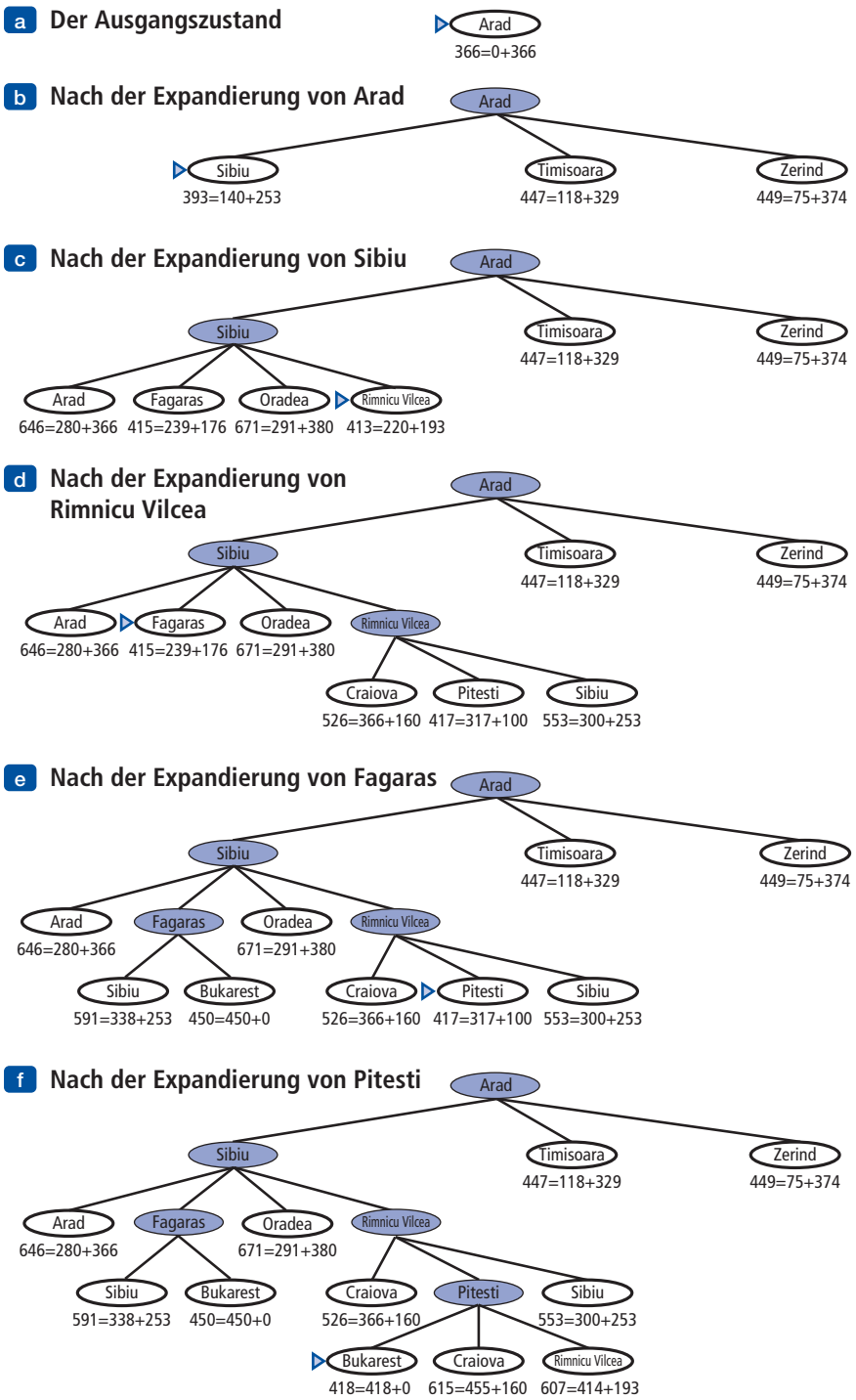


Abbildung 3.24: Phasen in einer A*-Suche nach Bukarest. Die Knoten sind mit $f = g + h$ beschriftet. Die h-Werte stellen die Luftliniendistanzen nach Bukarest aus Abbildung 3.22 dar.

Optimalität von A*

Tipp

Wie bereits erwähnt, besitzt A* die folgenden Eigenschaften: *Die Version der Baumsuche von A* ist optimal, wenn $h(n)$ zulässig ist, während die Version der Graphensuche optimal ist, wenn $h(n)$ konsistent ist.*

Wir zeigen hier die zweite dieser beiden Forderungen, da sie nützlicher ist. Das Argument spiegelt im Wesentlichen das Argument für die Optimalität einer Suche mit einheitlichen Kosten wider, wobei g durch f ersetzt wird – genau wie im A*-Algorithmus selbst.

Tipp

Im ersten Schritt wird Folgendes festgestellt: *Wenn $h(n)$ konsistent ist, dann sind die Werte von $f(n)$ entlang eines beliebigen Pfades nicht fallend.* Der Beweis folgt direkt aus der Definition der Konsistenz. Angenommen, n' ist ein Nachfolger von n; dann gilt für eine Aktion a immer $g(n') = g(n) + c(n, a, n')$ und wir erhalten:

$$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \geq g(n) + h(n) = f(n).$$

Tipp

Im nächsten Schritt ist zu beweisen: *Wenn A* einen Knoten n zum Expandieren auswählt, ist der optimale Pfad zu diesem Knoten gefunden worden.* Wo dies nicht der Fall ist, müsste es gemäß der Trennungseigenschaft der Graphensuche von Abbildung 3.9 einen anderen Grenzknoten n' auf dem optimalen Pfad vom Startknoten zu n geben. Da f entlang eines beliebigen Pfades nicht fallend ist, hätte n' geringere f-Kosten als n und wäre zuerst ausgewählt worden.

Aus den beiden vorherigen Beobachtungen folgt, dass die Knotenfolge, die durch A* unter Verwendung von GRAPH-SEARCH expandiert wurde, in nicht fallender Reihenfolge von $f(n)$ vorliegt. Damit muss der erste für die Expandierung ausgewählte Zielknoten eine optimale Lösung sein, weil f die wahren Kosten für Zielknoten verkörpert (die $h = 0$ haben) und alle späteren Knoten mindestens genauso teuer sind.

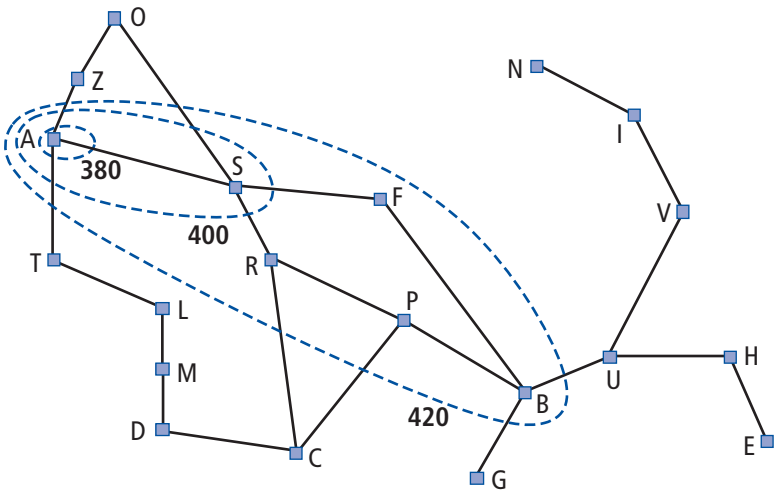


Abbildung 3.25: Karte von Rumänien mit den Konturen für $f = 380$, $f = 400$ und $f = 420$ mit Arad als Ausgangszustand. Knoten innerhalb einer bestimmten Kontur haben f-Kosten kleiner oder gleich dem Konturwert.

Die Tatsache, dass die f-Kosten auf jedem beliebigen Pfad nicht fallend sind, bedeutet auch, dass wir **Konturen** im Zustandsraum zeichnen können, ähnlich den Konturen in einer topografischen Karte. ▶ Abbildung 3.25 zeigt ein Beispiel dafür. Innerhalb der

Kontur mit der Beschriftung 400 haben alle Knoten ein $f(n)$ kleiner oder gleich 400 usw. Weil A* den Grenzknoten mit den kleinsten f-Kosten expandiert, sehen wir, dass sich eine A*-Suche vom Ausgangsknoten aus ausbreitet und Knoten in konzentrischen Bändern steigender f-Kosten hinzufügt.

Bei einer Suche mit einheitlichen Kosten (A*-Suche mit $h(n) = 0$) entstehen die Bänder „kreisförmig“ um den Ausgangszustand herum. Mit einer genaueren Heuristik dehnen sich die Bänder in Richtung Zielzustand aus und konzentrieren sich eng um den optimalen Pfad. Wenn C^* die Kosten des optimalen Lösungspfades darstellt, können wir Folgendes sagen:

- A* expandiert alle Knoten mit $f(n) < C^*$.
- A* könnte dann einige der Knoten rechts von der „Zielkontur“ (mit $f(n) = C^*$) expandieren, bevor ein Zielknoten ausgewählt wird.

Die Vollständigkeit verlangt, dass es nur endlich viele Knoten mit Kosten kleiner oder gleich C^* gibt. Diese Bedingung ist erfüllt, wenn alle Schrittkosten ein bestimmtes endliches ε überschreiten und wenn b endlich ist.

Beachten Sie, dass A* keine Knoten mit $f(n) > C^*$ expandiert – beispielsweise wird in Abbildung 3.24 Timisoara nicht expandiert, obwohl es ein untergeordneter Knoten des Wurzelknotens ist. Wir sagen, der Unterbaum unterhalb von Timisoara wird **gekürzt**; weil h_{LLD} zulässig ist, kann der Algorithmus diesen Unterbaum getrost ignorieren und dennoch Optimalität garantieren. Das Konzept der **Kürzung** (**Pruning**) – das Eliminieren von Möglichkeiten noch vor der Auswertung, ohne sie betrachten zu müssen – ist für viele Bereiche der KI wichtig.

Schließlich ist festzustellen, dass von optimalen Algorithmen dieser Art – Algorithmen, die Suchpfade von der Wurzel aus erweitern und die gleichen heuristischen Informationen verwenden – A* **optimal effizient** für jede gegebene konsistente Heuristik ist. Anders ausgedrückt lässt sich für keinen anderen optimalen Algorithmus garantieren, dass er weniger Knoten als A* expandiert (außer möglicherweise durch Stichwahl unter den Knoten mit $f(n) = C^*$). Das liegt daran, dass jeder Algorithmus, der nicht sämtliche Knoten mit $f(n) < C^*$ expandiert, das Risiko eingeht, die optimale Lösung zu übersehen.

Es ist durchaus zufriedenstellend, dass die A*-Suche unter allen diesen Suchalgorithmen vollständig, optimal und optimal effizient ist. Leider bedeutet das nicht gleichzeitig, dass A* die Antwort auf all unsere Suchanforderungen darstellt. Der Haken dabei ist, dass für die meisten Probleme die Anzahl der Zustände innerhalb des Zielkontur-Suchraums immer noch exponentiell zur Länge der Lösung wächst. Die Einzelheiten der Analyse gingen zwar über den Rahmen dieses Buches hinaus, die grundsätzlichen Ergebnisse seien aber im Folgenden angeführt. Für Probleme mit konstanten Schrittkosten wird die Zunahme der Laufzeit als Funktion der optimalen Lösungstiefe d in Form des **absoluten Fehlers** oder des **relativen Fehlers** der Heuristik analysiert. Der absolute Fehler ist definiert als $\Delta \equiv h^* - h$, wobei h^* die tatsächlichen Kosten darstellt, um von der Wurzel zum Ziel zu gelangen, und der relative Fehler ist mit $\varepsilon \equiv (h^* - h)/h^*$ definiert.

Die Komplexitätsergebnisse hängen stark von den Annahmen ab, die über den Zustandsraum getroffen werden. Das einfachste untersuchte Modell ist ein Zustandsraum, der ein einziges Ziel besitzt und im Wesentlichen einen Baum mit umkehrbaren Aktionen darstellt. (Das 8-Puzzle erfüllt die erste und dritte dieser Annahmen.) In diesem Fall ist die Zeitkomplexität von A* exponentiell im maximalen absoluten Fehler,

d.h. $O(b^{\Delta})$. Bei konstanten Schrittkosten können wir dies als $O(b^{\varepsilon d})$ schreiben, wobei d die Lösungstiefe ist. Für fast alle praktisch verwendeten Heuristiken steigt der absolute Fehler mindestens proportional zu den Pfadkosten h^*, sodass ε konstant bleibt oder wächst und die Zeitkomplexität exponentiell in d ist. Außerdem können wir die Wirkung einer genaueren Heuristik sehen: $O(b^{\varepsilon d}) = O((b^{\varepsilon})^d)$, sodass der effektive Verzweigungsfaktor (den der nächste Abschnitt formaler definiert) b^{ε} ist.

Weist der Zustandsraum viele Zielzustände auf – insbesondere nahezu optimale Zielzustände –, kann der Suchvorgang vom optimalen Pfad abkommen und es entstehen zusätzliche Kosten proportional zur Anzahl der Ziele, deren Kosten innerhalb eines Faktors ε der optimalen Kosten liegen. Schließlich ist die Situation im allgemeinen Fall eines Graphen noch ungünstiger. Es kann exponentiell viele Zustände mit $f(n) < C^*$ geben, selbst wenn der absolute Fehler durch eine Konstante gebunden ist. Sehen Sie sich zum Beispiel eine Version der Staubsaugerwelt an, wo der Agent jedes Quadrat zu Einheitskosten säubern kann, ohne es überhaupt besuchen zu müssen: In diesem Fall spielt es keine Rolle, in welcher Reihenfolge die Quadrate gereinigt werden. Sind zu Beginn N Quadrate schmutzig, gibt es 2^N Zustände, bei denen eine Teilmenge gesäubert worden ist und die alle auf einem optimalen Lösungspfad liegen – wodurch sie $f(n) < C^*$ erfüllen –, selbst wenn die Heuristik einen Fehler von 1 hat.

Durch die Komplexität von A* ist es mit diesem Algorithmus oftmals unpraktisch, auf einer optimalen Lösung zu bestehen. Mit Varianten von A* lassen sich recht schnell suboptimale Lösungen finden und manchmal ist es auch möglich, Heuristiken zu entwerfen, die genauer, aber nicht im strengen Sinne zulässig sind. In jedem Fall bietet die Verwendung einer guten Heuristik erhebliche Einsparungen verglichen mit einer uninformierten Suche. *Abschnitt 3.6* geht der Frage nach, wie sich gute Heuristiken entwerfen lassen.

Die Rechenzeit ist jedoch nicht der größte Nachteil von A*. Weil A* alle erzeugten Knoten im Speicher behält (wie alle GRAPH-SEARCH-Algorithmen), geht dem Algorithmus normalerweise der Speicher aus, lange bevor die Rechenzeit knapp wird. Aus diesem Grund ist A* für viele größere Probleme nicht geeignet. Allerdings gibt es Algorithmen, die das Platzproblem überwinden, ohne Optimalität oder Vollständigkeit zu opfern – auf Kosten einer etwas längeren Ausführungszeit. Darauf gehen wir als Nächstes ein.

3.5.3 Speicherbegrenzte heuristische Suche

Speicheranforderungen für A* lassen sich am einfachsten reduzieren, wenn man das Konzept der iterativen Vertiefung in den heuristischen Suchkontext aufnimmt, woraus der **Iterative-Deepening-A***-Algorithmus (IDA*) – der iterativ vertiefende A*-Algorithmus – entstanden ist. Der größte Unterschied zwischen IDA* und der standardmäßigen iterativen Vertiefung ist, dass die Kürzung an den f-Kosten ($g + h$) vorgenommen wurde und nicht an der Tiefe; bei jeder Iteration ist der Kürzungswert (Cutoff) der kleinste f-Kostenwert jedes Knotens, der die Kürzung der vorhergehenden Iteration überschritten hat. IDA* ist für viele Probleme mit einheitlichen Schrittkosten praktisch und vermeidet den erheblichen Zusatzaufwand, um eine Warteschlange mit Knoten sortiert zu halten. Bedauerlicherweise leidet er unter denselben Schwierigkeiten wie die iterative Version der Suche mit einheitlichen Kosten, wie in Übung 3.18 beschrieben, wenn reellwertige Kosten entstehen. Dieser Abschnitt betrachtet kurz zwei andere speicherbegrenzte Algorithmen namens RBFS und MA*.

Die **rekursive Bestensuche** (Recursive Best-First Search, RBFS) ist ein einfacher rekursiver Algorithmus, der versucht, die Arbeitsweise der Standard-Bestensuche nachzubilden, dabei aber mit linearem Speicher auszukommen. ▶ Abbildung 3.26 zeigt den Algorithmus. Seine Struktur ist mit der einer rekursiven Tiefensuche (Depth-First) vergleichbar, aber anstatt den aktuellen Pfad unendlich weit nach unten zu verfolgen, überwacht er mithilfe der Variablen *f_limit* den *f*-Wert des besten alternativen Pfades, der von jedem Vorläufer des aktuellen Knotens zur Verfügung steht. Überschreitet der aktuelle Knoten diese Obergrenze, geht die Rekursion zurück zum alternativen Pfad. Im Rekursionsschritt ersetzt RBFS den *f*-Wert jedes Knotens auf dem Pfad durch einen **gesicherten Wert** (**backed-up value**) – den besten *f*-Wert seiner untergeordneten Knoten. Auf diese Weise merkt sich RBFS den *f*-Wert des besten Blattes im vergessenen Unterbaum und kann damit entscheiden, ob es sinnvoll ist, den Unterbaum später erneut zu expandieren. ▶ Abbildung 3.27 zeigt, wie RBFS Bukarest erreicht.

```
function RECURSIVE-BEST-FIRST-SEARCH (problem) returns eine Lösung oder Fehler
  return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns eine Lösung oder Fehler und
eine neue f-Kostengreze
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← []
  for each action in problem.ACTIONS (node.STATE) do
    CHILD-NODE(problem, node, action) in successors einfügen
  if successors ist leer then return failure, ∞
  for each s in successors do /* f mit Wert von vorheriger Suche
  aktualisieren, falls vorhanden */
    s.f ← max(s.g + s.h, node.f ))
  loop do
    best ← den kleinsten f-Wert-Knoten in successors
    if best.f > f_limit then return failure, best.f
    alternative ← der zweitkleinste f-Wert unter successors
    result, best.f ← RBFS(problem, best, min(f_limit, alternative))
    if result ≠ failure then return result
```

Abbildung 3.26: Der Algorithmus für die rekursive Bestensuche (Best-First).

RBFS ist etwas effizienter als IDA*, leidet aber dennoch unter einer exzessiven Knotenneubildung. Im Beispiel in Abbildung 3.27 folgt RBFS zuerst dem Pfad über Rimnicu Vilcea, „überlegt es sich dann anders“ und versucht es mit Fagaras, revidiert aber seine Meinung erneut. Die Meinungsänderungen treten auf, weil wahrscheinlich jedes Mal der *f*-Wert des momentan besten Pfades zunimmt, wenn dieser erweitert wird – *h* ist normalerweise weniger optimistisch für Knoten, die näher am Ziel liegen. Wenn das passiert, könnte der zweitbeste Pfad zum besten Pfad werden, sodass die Suche zurückgehen muss, um ihn weiterzuverfolgen. Jede Meinungsänderung entspricht einer Iteration von IDA* und könnte viele erneute Expandierungen bereits vergessener Knoten verursachen, um den besten Pfad wiederherzustellen und ihn um einen Knoten zu erweitern.

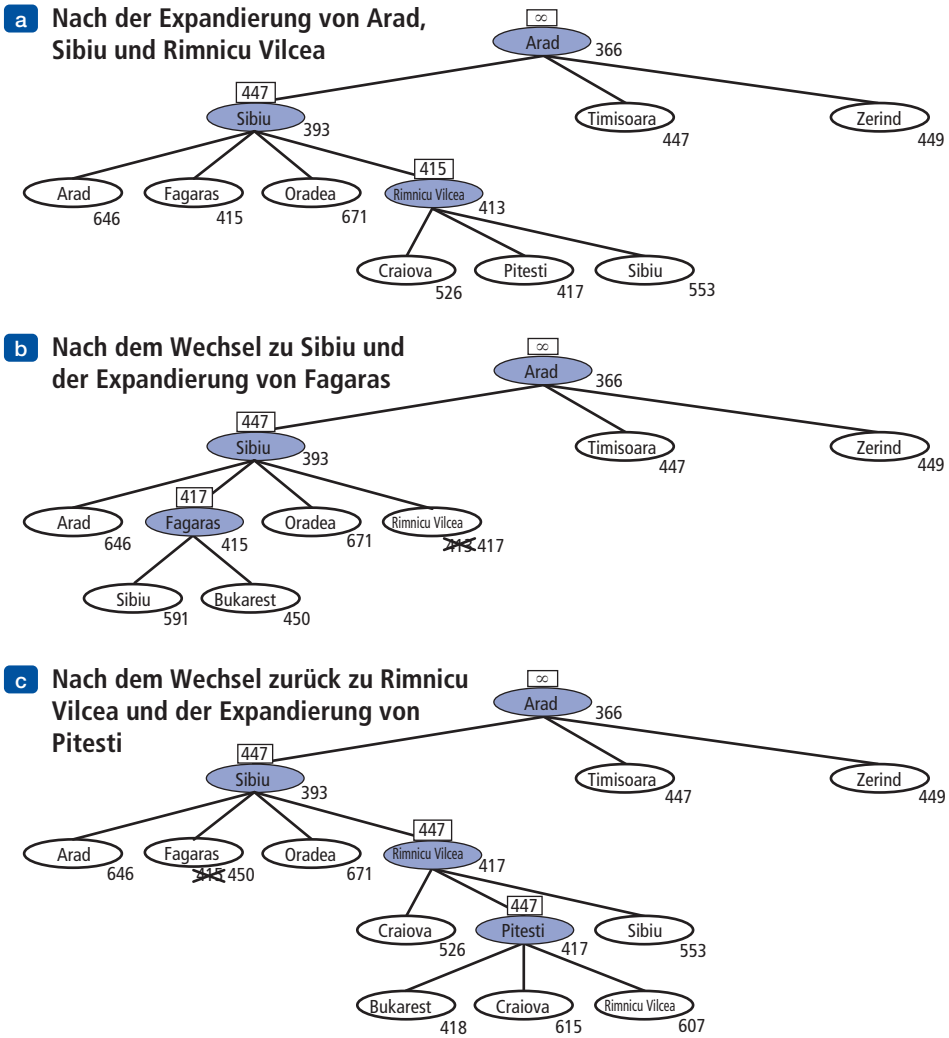


Abbildung 3.27: Phasen in einer RBFS-Suche nach dem kürzesten Weg nach Bukarest. Der f-Grenzwert für jeden rekursiven Aufruf ist jeweils oberhalb der Knoten dargestellt und jeder Knoten ist mit seinen f-Kosten bezeichnet. (a) Der Pfad über Rimnicu Vilcea wird verfolgt, bis das aktuell beste Blatt (Pitesti) einen Wert aufweist, der schlechter ist als der beste alternative Pfad (Fagaras). (b) Im Rekursionsschritt wird der beste Blattwert des vergessenen Unterbaumes (417) auf Rimnicu Vilcea gesetzt; anschließend wird Fagaras expandiert, was einen besten Blattwert von 450 zum Vorschein bringt. (c) Im Rekursionsschritt wird der beste Blattwert des vergessenen Unterbaumes (450) auf Fagaras gesetzt; anschließend wird Rimnicu Vilcea expandiert. Dieses Mal wird die Expandierung nach Bukarest fortgesetzt, weil der alternative Pfad (über Timisoara) mindestens 447 kostet.

Wie A* ist RBFS ein optimaler Algorithmus, wenn die Heuristikfunktion $h(n)$ zulässig ist. Seine Speicherkomplexität ist linear in der Tiefe der tiefsten optimalen Lösung, seine Zeitkomplexität ist jedoch schwierig zu charakterisieren: Sie hängt sowohl von der Genauigkeit der Heuristikfunktion ab als auch davon, wie oft sich der beste Pfad ändert, wenn Knoten expandiert werden.

IDA* und RBFS leiden darunter, *zu wenig* Speicher zu verwenden. Zwischen den Iterationen bewahrt IDA* nur einen einzigen Wert auf: die aktuelle Obergrenze der *f*-Kosten. RBFS hält mehr Information im Speicher, hat aber nur einen linearen Speicherplatzbedarf: Selbst wenn mehr Speicher zur Verfügung steht, hat RBFS keine Möglichkeit, ihn zu nutzen. Da beide Algorithmen das meiste davon vergessen, was sie getan haben, erweitern sie letztlich dieselben Zustände mehrere Male erneut. Darüber hinaus leiden sie an der möglicherweise exponentiellen Zunahme der Komplexität, die mit redundanten Pfaden in Graphen verbunden ist (siehe *Abschnitt 3.3*).

Es erscheint deshalb sinnvoll, den gesamten verfügbaren Speicher zu nutzen. Zwei Algorithmen, die diese Methode anwenden, sind **MA*** (Memory-Bounded A*, speicherbegrenzter A*) und **SMA*** (Simplified MA*, vereinfachter MA*). Wir werden SMA* beschreiben, da dieser Algorithmus im wahrsten Sinne des Wortes „einfacher" ist. SMA* verhält sich genau wie A*, wobei das beste Blatt expandiert wird, bis der Speicher voll ist. An dieser Stelle kann er dem Suchbaum keinen neuen Knoten mehr hinzufügen, ohne einen alten zu entfernen. SMA* verwirft immer den schlechtesten Blattknoten – denjenigen mit dem höchsten *f*-Wert. Wie RBFS sichert SMA* dann den Wert des vergessenen Knotens in seinem übergeordneten Knoten. Auf diese Weise erfährt der Vorgänger eines vergessenen Unterbaumes die Qualität des besten Pfades in diesem Unterbaum. Anhand dieser Informationen stellt SMA* den Unterbaum nur dann wieder her, wenn gezeigt wurde, dass alle anderen Pfade schlechter aussehen als der Pfad, den er bereits vergessen hat. Anders ausgedrückt: Wenn alle Nachfolger eines Knotens *n* vergessen sind, wissen wir nicht, wohin wir von *n* aus gehen sollen, haben aber immer noch eine Idee, wie sinnvoll es ist, von *n* aus irgendwohin zu gehen.

Der vollständige Algorithmus ist zwar zu kompliziert, um ihn hier wiedergeben zu können,[10] doch sollte ein Detail erwähnt werden. Wir haben gesagt, dass SMA* das beste Blatt expandiert und das schlechteste Blatt löscht. Was passiert, wenn *alle* Blattknoten den gleichen *f*-Wert haben? Um zu vermeiden, denselben Knoten sowohl zum Löschen als auch zum Erweitern auszuwählen, expandiert SMA* das *neueste* beste Blatt und löscht das *älteste* schlechteste Blatt. Diese fallen zusammen, wenn es nur ein einziges Blatt gibt, doch muss es sich in diesem Fall beim aktuellen Suchbaum um einen einzigen Pfad von der Wurzel zum Blatt handeln, der den gesamten Speicher füllt. Ist das Blatt kein Zielknoten, ist diese Lösung mit dem verfügbaren Speicher nicht erreichbar, *selbst wenn sie auf einem optimalen Lösungspfad liegt*. Der Knoten kann also genauso verworfen werden, als hätte er keine Nachfolger.

SMA* ist vollständig, wenn es eine erreichbare Lösung gibt – d.h. wenn *d*, die Tiefe des flachsten Zielknotens, kleiner als die Speichergröße (ausgedrückt in Knoten) ist. Er ist optimal, wenn eine optimale Lösung erreichbar ist; andernfalls gibt der Algorithmus die beste erreichbare Lösung zurück. Konkret heißt das, dass SMA* eine recht robuste Variante ist, um optimale Lösungen zu finden, insbesondere wenn der Zustandsraum ein Graph ist, die Schrittkosten nicht einheitlich sind und die Knotenerstellung teuer im Vergleich zu dem Zusatzaufwand ist, die Menge der Grenzknoten und der untersuchten Knoten zu verwalten.

Tipp

Bei sehr schwierigen Problemen kommt es jedoch häufig vor, dass SMA* gezwungen wird, ständig innerhalb einer Menge möglicher Lösungspfade hin- und herzuschalten, von der nur eine kleine Untermenge in den Speicher passt. (Das erinnert an das **Trashing-**

10 In der ersten Auflage dieses Buches gab es eine grobe Skizze.

Problem (Ein-/Auslagerung von Seiten) in Paging-Systemen bei Festplatten.) Wegen der zusätzlichen Zeit, die für die wiederholte Neuerstellung derselben Knoten aufzuwenden ist, sind dann Probleme, die bei unbegrenztem Speicher für A* praktisch lösbar wären, für SMA* nicht handhabbar. Anders ausgedrückt *können Speicherbeschränkungen unter dem Aspekt der Rechenzeit ein Problem nicht handhabbar machen*. Obwohl es derzeit keine Theorie gibt, die den Kompromiss zwischen Zeit und Speicher beschreibt, scheint dies ein Problem zu sein, dem man nicht entrinnen kann. Der einzige Ausweg ist, die Forderung der Optimalität zu verwerfen.

3.5.4 Lernen, besser zu suchen

Wir haben mehrere feste Suchstrategien vorgestellt – Breitensuche, „gierige" Breitensuche usw. –, die von Informatikern entworfen wurden. Könnte ein Agent *lernen*, wie sich besser suchen lässt? Die Antwort lautet „Ja" und die Methode beruht auf einem wichtigen Konzept, dem sogenannten **Zustandsraum auf Metaebene**. Jeder Zustand in einem Zustandsraum auf Metaebene erfasst den internen (berechneten) Zustand eines Programmes, das in einem **Zustandsraum auf Objektebene** sucht, wie beispielsweise Rumänien. Zum Beispiel besteht der interne Zustand des A*-Algorithmus aus dem aktuellen Suchbaum. Jede Aktion in dem Zustandsraum auf Metaebene ist ein Rechenschritt, der den internen Zustand ändert; z.B. expandiert jeder Rechenschritt in A* einen Blattknoten und fügt dem Baum seine Nachfolger hinzu. Somit lässt sich Abbildung 3.24, die eine Folge von größer werdenden Suchbäumen zeigt, als Darstellung eines Pfades im Zustandsraum auf Metaebene ansehen, wo jeder Zustand auf dem Pfad ein Suchbaum auf Objektebene ist.

Der Pfad in Abbildung 3.24 hat nun fünf Schritte, inklusive eines Schrittes für die Expandierung von Fagaras, der nicht besonders hilfreich ist. Für schwierigere Probleme gibt es viele solcher Fehlschritte und ein Algorithmus für **Lernen auf Metaebene** kann aus diesen Erfahrungen lernen, um das Untersuchen aussichtsloser Suchbäume zu vermeiden. *Kapitel 21* beschäftigt sich mit Techniken für derartige Lernverfahren. Ziel des Lernens ist es, die Gesamtkosten für die Problemlösung zu minimieren und dabei einen Kompromiss zwischen Rechenaufwand und Pfadkosten zu schließen.

3.6 Heuristikfunktionen

In diesem Abschnitt betrachten wir Heuristiken für das 8-Puzzle, um die allgemeine Natur von Heuristiken zu verstehen.

Das 8-Puzzle war eines der ersten Probleme für die heuristische Suche. Wie in *Abschnitt 3.2* erwähnt, ist es das Ziel des Puzzles, die Felder horizontal oder vertikal an den leeren Platz zu schieben, bis die Konfiguration der Zielkonfiguration entspricht (▶ Abbildung 3.28).

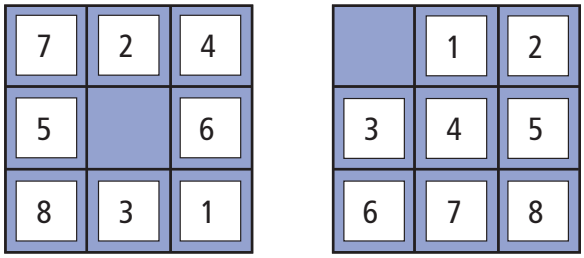


Abbildung 3.28: Ein typisches Beispiel für das 8-Puzzle. Die Lösung umfasst 26 Schritte.

Die durchschnittlichen Lösungskosten für eine zufällig generierte Instanz des 8-Puzzles umfassen etwa 22 Schritte. Der Verzweigungsfaktor liegt bei etwa 3. (Wenn das leere Feld in der Mitte liegt, gibt es vier mögliche Züge; befindet es sich in einer Ecke, gibt es zwei, und befindet es sich an einer Kante, gibt es drei.) Das bedeutet, eine erschöpfende Suche bis zur Tiefe 22 würde etwa $3^{22} \approx 3{,}1 \times 10^{10}$ Zustände betrachten. Eine Graphensuche würde dies um einen Faktor von etwa 170.000 reduzieren, weil es nur $9!/2 = 181.440$ unterschiedliche Zustände gibt, die erreichbar sind (siehe Übung 3.5). Das ist eine überschaubare Zahl, aber die entsprechende Zahl für das 15-Puzzle beträgt etwa 10^{13}. Deshalb besteht die nächste wichtige Aufgabe darin, eine gute Heuristikfunktion zu finden. Wenn wir unter Verwendung von A* die kürzesten Lösungen finden wollen, brauchen wir eine Heuristikfunktion, die nie die Anzahl der Schritte zum Ziel überschätzt. Es gibt zahlreiche Ansatzversuche für eine solche Heuristik zum 15-Puzzle; nachfolgend sind zwei der gebräuchlicheren Ansätze beschrieben:

- h_1 = die Anzahl falsch platzierter Felder. In Abbildung 3.28 befinden sich alle acht Felder an der falschen Position, der Ausgangszustand liegt also bei $h_1 = 8$. h_1 ist eine zulässige Heuristik, weil klar ist, dass jedes Feld, das sich nicht am richtigen Platz befindet, mindestens einmal bewegt werden muss.
- h_2 = die Summe der Distanzen der Felder von ihren Zielpositionen. Weil die Felder sich nicht diagonal bewegen können, berechnen wir die Distanz aus der Summe der horizontalen und vertikalen Distanzen. Man spricht auch von der **City-Block-Distanz** oder **Manhattan-Distanz**. h_2 ist ebenfalls zulässig, weil sich mit einem beliebigen Zug höchstens ein Feld um einen Schritt näher an das Ziel bringen lässt. Die Felder 1 bis 8 im Startzustand ergeben eine Manhattan-Distanz von

$$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18.$$

Wie zu erwarten, ergibt keiner dieser Werte eine Schätzung oberhalb der tatsächlichen Lösungskosten, die bei 26 liegen.

3.6.1 Die Wirkung heuristischer Genauigkeit auf die Leistung

Eine Möglichkeit, die Qualität einer Heuristik zu charakterisieren, ist der **effektive Verzweigungsfaktor** b^*. Ist die von A* erzeugte Gesamtzahl von Knoten für ein bestimmtes Problem gleich N und die Lösungstiefe gleich d, dann ist b^* der Verzweigungsfaktor, den ein einheitlicher Baum der Tiefe d bräuchte, um $N + 1$ Knoten aufzunehmen. Es gilt also:

$$N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d.$$

Findet A* beispielsweise eine Lösung bei der Tiefe 5 unter Verwendung von 52 Knoten, ist der effektive Verzweigungsfaktor gleich 1,92. Der effektive Verzweigungsfaktor kann über verschiedene Probleminstanzen hinweg variieren, normalerweise ist er aber für ausreichend schwierige Probleme ziemlich konstant. (Die Existenz eines effektiven Verzweigungsfaktors folgt aus dem weiter vorn erwähnten Ergebnis, dass die Anzahl der durch A* erweiterten Knoten exponentiell mit der Lösungstiefe zunimmt.) Deshalb können experimentelle Messungen von b^* für eine kleine Menge von Problemen einen guten Anhaltspunkt für die allgemeine Nützlichkeit der Heuristik bieten. Für eine gute Heuristik liegt der Wert von b^* nahe 1, wodurch sich ziemlich große Probleme bei zumutbaren Berechnungskosten lösen lassen.

Um die Heuristikfunktionen h_1 und h_2 zu testen, haben wir 1200 zufällige Probleme mit Lösungslängen von 2 bis 24 (100 für jede gerade Zahl) erzeugt und sie mit einer iterativ vertiefenden Suche gelöst, wobei eine A*-Baumsuche unter Verwendung von h_1 und h_2 angewendet wurde. ▶ Abbildung 3.29 gibt die durchschnittliche Anzahl der von jeder Strategie expandierten Knoten sowie den effektiven Verzweigungsfaktor an. Wie die Ergebnisse zeigen, ist h_2 besser als h_1 und sehr viel besser als die Verwendung einer iterativ vertiefenden Suche. Selbst für kleine Probleme mit $d = 12$ ist A* mit h_2 rund 50.000-mal effizienter als eine uninformierte iterativ vertiefende Suche.

	Suchkosten (erzeugte Knoten)			Effektiver Verzweigungsfaktor		
d	IDS	A* (h_1)	A* (h_2)	IDS	A* (h_1)	A* (h_2)
2	10	6	6	2,45	1,79	1,79
4	112	13	12	2,87	1,48	1,45
6	680	20	18	2,73	1,34	1,30
8	6384	39	25	2,80	1,33	1,24
10	47127	93	39	2,79	1,38	1,22
12	3644035	227	73	2,78	1,42	1,24
14	–	539	113	–	1,44	1,23
16	–	1301	211	–	1,45	1,25
18	–	3056	363	–	1,46	1,26
20	–	7276	676	–	1,47	1,27
22	–	18094	1219	–	1,48	1,28
24	–	39135	1641	–	1,48	1,26

Abbildung 3.29: Vergleich der Suchkosten und der effektiven Verzweigungsfaktoren für die ITERATIVE-DEEPENING-SEARCH- und A*-Algorithmen mit $h_1$und h_2. Die Daten wurden über 100 Instanzen des 8-Puzzles für verschiedene Lösungslängen d gemittelt.

Man stellt sich vielleicht die Frage, ob h_2 *immer* besser als h_1 ist. Die Antwort lautet „Im Grunde ja“. Aus den Definitionen der beiden Heuristiken ist leicht zu erkennen, dass für jeden Knoten n gilt: $h_2(n) \geq h_1(n)$. Wir sagen damit, h_2 **dominiert** h_1. Diese **Dominanz** kann direkt in Effizienz übersetzt werden: A* mit h_2 expandiert nie mehr Knoten als A* mit h_1 (außer möglicherweise für einige Knoten mit $f(n) = C^*$). Das

Argument ist einfach. Aus *Abschnitt 3.5.2* wissen Sie, dass jeder Knoten mit $f(n) < C^*$ sicher expandiert wird. Mit anderen Worten wird jeder Knoten mit $h(n) < C^* - g(n)$ sicher expandiert. Weil jedoch h_2 mindestens so groß wie h_1 für alle Knoten ist, wird jeder Knoten, der von der A*-Suche mit h_2 expandiert wird, sicher auch mit h_1 expandiert, und h_1 könnte auch bewirken, dass andere Knoten ebenfalls expandiert werden. Damit ist es im Allgemeinen besser, eine Heuristikfunktion mit höheren Werten zu verwenden, vorausgesetzt, dass sie konsistent ist und die Berechnung der Heuristik nicht zu lange dauert.

3.6.2 Zulässige Heuristikfunktionen aus gelockerten Problemen generieren

Wir haben gesehen, dass sowohl h_1 (falsch platzierte Felder) als auch h_2 (Manhattan-Distanz) relativ gute Heuristiken für das 8-Puzzle darstellen und dass h_2 besser ist. Wie ist man zu h_2 gekommen? Ist es für einen Computer möglich, eine solche Heuristik mechanisch zu erfinden?

Tipp

h_1 und h_2 sind Schätzungen für die verbleibenden Pfadlängen für das 8-Puzzle, aber auch vollkommen exakte Pfadlängen für *vereinfachte* Versionen des Puzzles. Würden die Regeln für das Puzzle geändert, sodass ein Feld sich an eine beliebige Position bewegen könnte, statt nur auf das benachbarte leere Feld, würde h_1 die genaue Anzahl der Schritte für die kürzeste Lösung liefern. Könnte sich vergleichbar dazu ein Feld um ein Quadrat in jeder beliebigen Richtung bewegen, selbst auf ein belegtes Quadrat, würde h_2 die genaue Anzahl der Schritte in der kürzesten Lösung angeben. Ein Problem mit weniger Beschränkungen im Hinblick auf die Aktionen wird als **gelockertes Problem** bezeichnet. Der Zustandsraumgraph des gelockerten Problems ist ein Supergraph des ursprünglichen Zustandsraumes, da das Entfernen von Beschränkungen zusätzliche Kanten im Graphen erzeugt.

Weil das gelockerte Problem zusätzliche Kanten in den Zustandsraum einbringt, ist jede optimale Lösung im ursprünglichen Problem per Definition auch eine Lösung im gelockerten Problem. Allerdings kann das gelockerte Problem *bessere* Lösungen haben, wenn sich durch die hinzugefügten Kanten Abkürzungen ergeben. Folglich sind *die Kosten einer optimalen Lösung für ein gelockertes Problem eine zulässige Heuristik für das ursprüngliche Problem*. Und da die abgeleitete Heuristik die exakten Kosten für das gelockerte Problem angibt, muss sie der Dreiecksungleichung entsprechen und ist deshalb *konsistent* (siehe *Abschnitt 3.5.2*).

Wird eine Problemdefinition in einer formalen Sprache formuliert, ist es möglich, gelockerte Probleme automatisch zu konstruieren.[11] Werden zum Beispiel die Aktionen des 8-Puzzles als

> Ein Feld kann von Quadrat A nach Quadrat B verschoben werden, wenn
>
> A horizontal oder vertikal neben B liegt **und** B leer ist.

beschrieben, können wir drei gelockerte Probleme erzeugen, indem wir eine oder beide Bedingungen entfernen:

11 In den *Kapiteln 8* und *10* beschreiben wir formale Sprachen, die für diese Aufgabenstellung geeignet sind; mit formalen Beschreibungen, die sich manipulieren lassen, kann die Konstruktion gelockerter Probleme automatisiert werden. Im Moment verwenden wir dafür die natürliche Sprache.

(a) Ein Feld kann von Quadrat A auf Quadrat B verschoben werden, wenn A neben B liegt.

(b) Ein Feld kann von Quadrat A auf Quadrat B verschoben werden, wenn B leer ist.

(c) Ein Feld kann von Quadrat A auf Quadrat B verschoben werden.

Aus (a) können wir h_2 (Manhattan-Distanz) ableiten. Die Begründung ist, dass h_2 die richtige Punktzahl enthält, wenn wir die einzelnen Felder nacheinander auf ihr Ziel schieben. Die von (b) abgeleitete Heuristik wird in Übung 3.34 genauer erörtert. Aus (c) können wir h_1 (falsch platzierte Felder) ableiten, weil es die richtige Punktzahl wäre, wenn die Felder innerhalb eines einzigen Schrittes an ihre vorgesehenen Positionen verschoben werden könnten. Beachten Sie, dass es wichtig ist, dass die durch diese Technik erzeugten gelockerten Probleme im Wesentlichen *ohne Suche* erzeugt werden können, weil es die gelockerten Regelungen erlauben, dass das Problem in acht voneinander unabhängige Teilprobleme zerlegt wird. Wenn das gelockerte Problem schwierig zu lösen ist, sind die Werte der entsprechenden Heuristik nur mit großem Aufwand zu ermitteln.[12]

Ein Programm namens ABSOLVER kann aus Problemdefinitionen automatisch Heuristiken erzeugen, und zwar unter Verwendung der Methode der „gelockerten Probleme“ und verschiedener anderer Techniken (Prieditis, 1993). ABSOLVER hat eine neue Heuristik für das 8-Puzzle erzeugt, die besser als jede zuvor existierende Heuristik ist, und die erste brauchbare Heuristik für den Zauberwürfel von Rubik gefunden.

Bei der Erzeugung neuer Heuristikfunktionen hat man häufig das Problem, keine „eindeutig beste“ Heuristik zu finden. Wenn eine Menge zulässiger Heuristiken $h_1 \ldots h_m$ für ein Problem zur Verfügung steht und keine davon die anderen dominiert, welche sollen wir dann verwenden? Wie sich zeigt, brauchen wir keine Auswahl zu treffen. Wir erhalten die global beste, indem wir definieren:

$$h(n) = max\{ h_1(n), ..., h_m(n)\}.$$

Diese zusammengesetzte Heuristik verwendet die Funktion, die am genauesten für den betreffenden Knoten ist. Weil die Komponenten-Heuristiken zulässig sind, ist h zulässig; man kann auch einfach beweisen, dass h konsistent ist. Darüber hinaus dominiert h alle seine Komponenten-Heuristiken.

3.6.3 Zulässige Heuristiken aus Unterproblemen generieren: Musterdatenbanken

Zulässige Heuristiken können auch von den Lösungskosten eines **Unterproblems** für ein bestimmtes Problem abgeleitet werden. ▶ Abbildung 3.30 beispielsweise zeigt ein Unterproblem der 8-Puzzle-Instanz aus Abbildung 3.28. Das Unterproblem fordert, dass die Felder 1, 2, 3 und 4 an ihre korrekte Position gebracht werden. Offensichtlich sind die Kosten für die optimale Lösung dieses Unterproblems eine Untergrenze für die Kosten des vollständigen Problems. In manchen Fällen ist diese Vorgehensweise genauer als die Manhattan-Distanz.

12 Beachten Sie, dass man zu einer perfekten Heuristik kommen kann, wenn man es erlaubt, dass h „heimlich“ eine vollständige Breitensuche ausführt. Es gibt also einen Kompromiss zwischen Genauigkeit und Rechenzeit für Heuristikfunktionen.

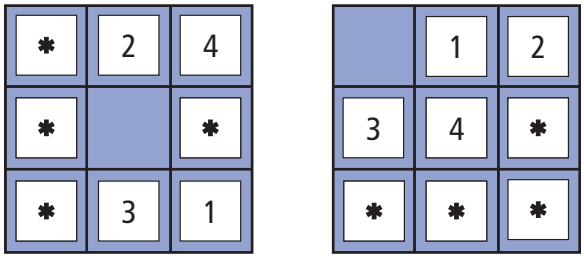


Abbildung 3.30: Ein Unterproblem der 8-Puzzle-Instanz aus Abbildung 3.28. Die Aufgabe besteht darin, die Felder 1, 2, 3 und 4 an ihre korrekten Positionen zu schieben, ohne sich darum zu kümmern, was mit den anderen Feldern passiert.

Der Gedanke hinter den **Musterdatenbanken** ist, diese genauen Lösungskosten für jede mögliche Unterprobleminstanz zu speichern – in unserem Beispiel jede mögliche Konfiguration der vier Felder und des leeren Feldes. (Zwar sind die Positionen der anderen vier Felder für die Lösung des Unterproblems irrelevant, jedoch gehen Züge mit diesen Feldern in die Kosten ein.) Wir berechnen also eine zulässige Heuristik h_{DB} für jeden vollständigen Zustand, den wir bei einer Suche erkannt haben, indem wir einfach die entsprechende Konfiguration des Unterproblems in der Datenbank nachschlagen. Um die eigentliche Datenbank zu konstruieren, suchen wir rückwärts[13] vom Ziel aus und zeichnen die Kosten jedes neu gefundenen Musters auf. Der Aufwand für die Suche amortisiert sich bei vielen aufeinanderfolgenden Probleminstanzen.

Die Auswahl von 1-2-3-4 geschieht relativ willkürlich; wir könnten auch Datenbanken für 5-6-7-8 oder für 2-4-6-8 usw. erzeugen. Jede Datenbank ergibt eine zulässige Heuristik und wie bereits erläutert, ist es möglich, diese Heuristiken zu kombinieren, indem der Maximalwert verwendet wird. Eine kombinierte Heuristik dieser Art ist sehr viel genauer als die Manhattan-Distanz. So lässt sich die Anzahl der Knoten, die für die Lösung zufälliger 15-Puzzles erzeugt werden, um einen Faktor von 1000 reduzieren.

Es stellt sich die Frage, ob die Heuristiken aus der 1-2-3-4-Datenbank und der 5-6-7-8-Datenbank *addiert* werden können, da sich ja die beiden Unterprobleme scheinbar nicht überlappen. Ergäbe sich daraus trotzdem eine zulässige Heuristik? Die Antwort lautet „Nein“, weil die Lösungen des Unterproblems 1-2-3-4 und des Unterproblems 5-6-7-8 für einen bestimmten Zustand mit Sicherheit einige Züge gemeinsam haben – es ist unwahrscheinlich, dass 1-2-3-4 verschoben werden können, ohne 5-6-7-8 zu berühren, und umgekehrt. Wie verhält es sich aber, wenn wir diese Züge nicht zählen? Das heißt, wir zeichnen nicht die Gesamtkosten der Lösung des Unterproblems 1-2-3-4 auf, sondern nur die Anzahl der Züge, die mit 1-2-3-4 zu tun haben. Damit ist leicht zu erkennen, dass die Summe der beiden Kosten immer noch eine Untergrenze für die Kosten zur Lösung des Gesamtproblems darstellt. Dies ist die Idee hinter den **disjunkten Musterdatenbanken**. Mithilfe solcher Datenbanken ist es möglich, beliebige 15-Puzzles in ein paar Millisekunden zu lösen – die Anzahl der dabei generierten Knoten ist im Vergleich zur Manhattan-Distanz um einen Faktor von 10.000 kleiner. Für 24-Puzzles lässt sich eine Beschleunigung von etwa einer Million erhalten.

13 Indem rückwärts vom Ziel aus gearbeitet wird, stehen die genauen Lösungskosten jeder gefundenen Instanz unmittelbar zur Verfügung. Dies ist ein Beispiel für **dynamische Programmierung**, mit der sich *Kapitel 17* ausführlich beschäftigt.

Disjunkte Musterdatenbanken funktionieren bei Schiebeblock-Puzzles, weil sich das Problem so aufteilen lässt, dass jeder Zug nur ein Unterproblem beeinflusst – weil jeweils nur ein Feld auf einmal verschoben wird. Für ein Problem in der Art des Zauberwürfels von Rubik ist eine derartige Aufteilung schwierig, da an jedem Zug 8 oder 9 der 26 Würfel beteiligt sind. Für Rubiks Würfel wurden zwar auch allgemeinere Methoden vorgeschlagen, um additive, zulässige Heuristiken zu definieren (Yang et al., 2008), doch haben sie keine bessere Heuristik geliefert als die beste nichtadditive Heuristik für das Problem.

3.6.4 Heuristiken aus Erfahrung lernen

Eine Heuristikfunktion $h(n)$ soll die Kosten einer Lösung beginnend mit dem Zustand in Knoten n abschätzen. Wie könnte ein Agent eine derartige Funktion konstruieren? Eine Lösung wurde in den vorherigen Abschnitten angegeben – nämlich die Ableitung gelockerter Probleme, für die eine optimale Lösung einfach gefunden werden kann. Eine weitere Lösung ist, aus Erfahrung zu lernen. „Erfahrung" bedeutet hier, dass beispielsweise viele 8-Puzzles gelöst werden. Jede optimale Lösung für ein 8-Puzzle-Problem bietet Beispiele, aus denen $h(n)$ gelernt werden kann. Jedes Beispiel umfasst einen Zustand vom Lösungspfad und die tatsächlichen Kosten der Lösung von diesem Punkt an. Anhand dieser Beispiele lässt sich mit einem Lernalgorithmus eine Funktion $h(n)$ konstruieren, die (mit etwas Glück) Lösungskosten für andere Zustände vorhersagen kann, die während der Suche auftreten. *Kapitel 18* erläutert hierfür entsprechende Techniken, die mithilfe von neuronalen Netzen, Entscheidungsbäumen und anderen Methoden arbeiten. (Ebenfalls anwendbar sind die in *Kapitel 21* beschriebenen Methoden für verstärkendes Lernen.)

Induktive Lernmethoden funktionieren am besten, wenn ihnen nicht einfach die reinen Zustandsbeschreibungen übergeben werden, sondern **Merkmale** eines Zustandes, die relevant sind, um den Wert des Zustandes vorherzusagen. Beispielsweise könnte das Merkmal „Anzahl der falsch platzierten Felder" hilfreich sein, um die tatsächliche Distanz eines Zustandes vom Ziel vorherzusagen. Nennen wir dieses Funktionsmerkmal $x_1(n)$. Wir könnten 100 zufällig erzeugte 8-Puzzle-Konfigurationen nehmen und Statistiken zu ihren tatsächlichen Lösungskosten ermitteln. Wir stellen beispielsweise fest, dass für $x_1(n)$ gleich 5 die durchschnittlichen Lösungskosten bei etwa 14 liegen usw. Mit diesen Daten lässt sich der Wert von x_1 verwenden, um $h(n)$ vorherzusagen. Natürlich können wir mehrere Merkmale heranziehen. So könnte ein zweites Merkmal $x_2(n)$ die „Anzahl der Paare benachbarter Felder, die im Zielzustand nicht benachbart sind" sein. Wie sollten $x_1(n)$ und $x_2(n)$ kombiniert werden, um $h(n)$ vorherzusagen? Ein gebräuchlicher Ansatz ist die Verwendung einer Linearkombination:

$$h(n) = c_1 x_1(n) + c_2 x_2(n).$$

Die Konstanten c_1 und c_2 werden angepasst, um die beste Übereinstimmung mit den tatsächlichen Daten zu Lösungskosten zu erhalten. Man erwartet sowohl für c_1 als auch für c_2 positive Werte, da sich das Problem durch falsch platzierte Felder und nicht korrekt benachbarte Paare schwerer lösen lässt. Diese Heuristik erfüllt die Bedingung, dass $h(n) = 0$ für Zielzustände gilt, ist aber nicht unbedingt zulässig oder konsistent.

Bibliografische und historische Hinweise

Das Thema der Zustandsraumsuche geht mehr oder weniger in seiner aktuellen Form auf die frühen Jahre der KI zurück. Durch die Arbeit von Newell und Simon zum *Logic Theorist* (1957) und GPS (1961) etablierte sich die Problemlösung als kanonische Aufgabe der KI. Arbeiten in der Operationsforschung von Richard Bellman (1957) haben gezeigt, wie wichtig additive Pfadkosten beim Vereinfachen von Optimierungsalgorithmen sind. Der Aufsatz zu *Automated Problem Solving* von Nils Nilsson (1971) hat das Gebiet auf ein solides theoretisches Fundament gestellt.

Die meisten der in diesem Kapitel analysierten Zustandsraum-Suchprobleme haben eine lange Geschichte in der Literatur und sind weniger banal, als sie vielleicht erscheinen mögen. Das in Übung 3.9 verwendete Problem mit den Missionaren und den Kannibalen hat Amarel (1968) ausführlich analysiert. Es wurde aber bereits früher betrachtet – in der KI von Simon und Newell (1961) sowie in der Operationsforschung von Bellman und Dreyfus (1962).

Das 8-Puzzle ist ein kleiner Cousin des 15-Puzzles, dessen Geschichte ausführlich von Slocum und Sonneveld (2006) nacherzählt wurde. Lange Zeit glaubte man, dass es der berühmte amerikanische Spieldesigner Sam Loyd erfunden hat, was auf seine dementsprechenden Behauptungen seit 1891 zurückgeht (Loyd, 1959). Tatsächlich aber wurde es Mitte der 1870er Jahre von Noyes Chapman, einem Postangestellten in Canastota, New York, erfunden. (Chapman konnte seine Erfindung nicht patentieren, da Ernest Kinsey 1878 ein generisches Patent erteilt worden war, das sich auf das Verschieben von Steinen mit Buchstaben, Zahlen oder Bildern bezog.) Das Spiel erregte schnell die Aufmerksamkeit der Öffentlichkeit und von Mathematikern (Johnson und Story 1879; Tait, 1880). Die Herausgeber des *American Journal of Mathematics* berichteten: „Das 15-Puzzle wurde in den letzten Wochen beim amerikanischen Publikum immer bekannter und man kann getrost sagen, dass es die Aufmerksamkeit von neun von zehn Personen beiderlei Geschlechts sowie aller Altersstufen und Schichten auf sich gezogen hat.“ Ratner und Warmuth (1986) zeigten, dass die allgemeine $n \times n$-Version des 15-Puzzles zur Klasse der NP-vollständigen Probleme gehört.

Das 8-Damen-Problem wurde zuerst 1848 anonym im deutschen Schachmagazin *Schach* veröffentlicht, später einem gewissen Max Bezzel zugeordnet. Bei der erneuten Veröffentlichung im Jahre 1850 zog es die Aufmerksamkeit des großen Mathematikers Carl Friedrich Gauß auf sich, der versuchte, alle möglichen Lösungen aufzulisten, anfangs jedoch nur 72 fand, schließlich aber zur korrekten Antwort 92 gelangte. Allerdings hatte zuerst Nauck 1850 alle 92 Lösungen veröffentlicht. Netto (1901) verallgemeinerte das Problem auf n Damen und Abramson und Yung (1989) fanden einen Algorithmus mit $O(n)$.

Alle in diesem Kapitel aufgeführten Suchprobleme aus der realen Welt sind Gegenstand eingehender Forschungen gewesen. Methoden zur Auswahl optimaler Flüge bleiben größtenteils proprietär, aber Carl de Marcken (persönliche Kommunikation) hat gezeigt, dass Preisbildung und Beschränkungen von Flugtickets so kompliziert geworden sind, dass das Problem der Auswahl eines optimalen Fluges formell nicht entscheidbar ist. Das Problem des Handlungsreisenden ist ein kombinatorisches Standardproblem der theoretischen Informatik (Lawler, 1985; Lawler et al., 1992). Karp (1972) bewies, dass das Problem des Handlungsreisenden NP-hart ist, aber es wurden effektive heuristische Näherungsmethoden entwickelt (Lin und Kernighan, 1973). Arora (1998) leitete ein vollständig polynomiales Näherungsschema für Euklidsche TSPs ab. VLSI-Layoutmethoden

werden von Shahookar und Mazumder (1991) erforscht und in VLSI-Zeitschriften erschienen zahlreiche Artikel zur Layoutoptimierung. Roboternavigation und Montageprobleme werden in *Kapitel 25* beschrieben.

Algorithmen der uninformierten Suche für die Problemlösung sind ein zentrales Thema der klassischen Informatik (Horowitz und Sahni, 1978) und Operationsforschung (Dreyfus, 1969). Die Breitensuche wurde von Moore (1959) für die Lösung von Labyrinthen formuliert. Die Methode der **dynamischen Programmierung** (Bellman und Dreyfus, 1962), die systematisch alle Lösungen für alle Unterprobleme steigender Längen aufzeichnet, kann als eine Art von Breitensuche für Graphen betrachtet werden. Der Algorithmus von Dijkstra (1959) zum Berechnen des kürzesten Pfades zwischen zwei Punkten ist der Ursprung der Suche mit einheitlichen Kosten. Diese Arbeiten haben auch das Konzept von untersuchten und Grenzknotenmengen (geschlossene und offene Listen) eingeführt.

Eine Version der iterativ vertiefenden Suche, die darauf ausgelegt war, die Schachuhr effizient zu nutzen, wurde zuerst von Slate und Atkin (1977) im Schachspielprogramm CHESS 4.5 verwendet. Der B-Algorithmus von Martelli (1977) umfasst einen iterativ vertiefenden Aspekt und dominiert auch die ungünstigste Leistung von A* mit zulässigen, jedoch inkonsistenten Heuristiken. Die Technik der iterativen Vertiefung tauchte in der Arbeit von Korf (1985a) auf. Die von Pohl (1971) eingeführte bidirektionale Suche kann in bestimmten Fällen ebenfalls effektiv sein.

Die Verwendung von heuristischen Informationen in der Problemlösung erscheint in einem frühen Artikel von Simon und Newell (1958), doch kamen der Ausdruck „heuristische Suche“ und die Verwendung von heuristischen Funktionen, die die Distanz zum Ziel bewerten, erst etwas später auf (Newell und Ernst, 1965; Lin, 1965). Doran und Michie (1966) führten umfangreiche experimentelle Untersuchungen heuristischer Suchen aus. Obwohl sie Pfadlänge und „Penetranz“ (das Verhältnis von Pfadlänge zur Gesamtanzahl der bisher untersuchten Knoten) analysiert haben, wurden von ihnen die durch die Pfadkosten $g(n)$ gelieferten Informationen scheinbar ignoriert. Der A*-Algorithmus, der die tatsächlichen Pfadkosten in die heuristische Suche einbindet, wurde von Hart, Nilsson und Raphael (1968) entwickelt und hat später einige Korrekturen erfahren (Hart et al., 1972). Dechter und Pearl (1985) demonstrierten die optimale Effizienz von A*.

Der ursprüngliche Artikel zu A* führt die Konsistenzbedingung für heuristische Funktionen ein. Die Monotoniebedingung wurde von Pohl (1977) als einfacherer Ersatz vorgeschlagen, wobei aber Pearl (1984) zeigte, dass beide äquivalent sind.

Pohl (1977) bereitete den Weg zur Untersuchung der Beziehung zwischen dem Fehler in heuristischen Funktionen und der Zeitkomplexität von A*. Grundlegende Ergebnisse wurden für die Baumsuche mit einheitlichen Schrittkosten und einem einzelnen Zielknoten (Pohl, 1977; Gaschnig, 1979; Huyn et al., 1980; Pearl, 1984) sowie mit mehreren Zielknoten (Dinh et al., 2007) erhalten. Der „effektive Verzweigungsfaktor“ wurde von Nilsson (1971) als empirisches Maß der Effizienz vorgeschlagen; er ist äquivalent zur Annahme eines Zeitkostenverlaufes von $O((b^*)^d)$. Für die auf einen Graphen angewandte Baumsuche geben Korf et al. (2001) an, dass sich die Zeitkosten besser als $O(b^{d-k})$ modellieren lassen, wobei k von der heuristischen Genauigkeit abhängt. Allerdings hat diese Analyse einige Kontroversen ausgelöst. Für die Graphensuche merken Helmert und Röger (2008) an, dass mehrere gut bekannte Probleme exponentiell viele Knoten auf optimalen Lösungspfaden enthalten, was selbst mit konstantem absoluten Fehler in h eine exponentielle Zeitkomplexität für A* bedeutet.

Es gibt viele Varianten des A*-Algorithmus. Pohl (1973) schlug eine dynamische Gewichtung vor, die eine gewichtete Summe $f_w(n) = w_g\, g(n) + w_h\, h(n)$ der aktuellen Pfadlänge und die heuristische Funktion als Bewertungsfunktion statt der einfachen Summe $f(n) = g(n) + h(n)$ wie in A* verwendet. Die Gewichte w_g und w_h werden dynamisch mit fortschreitender Suche angepasst. Es lässt sich zeigen, dass der Algorithmus von Pohl ε-zulässig ist – d.h. garantiert Lösungen innerhalb eines Faktors $1 + \varepsilon$ der optimalen Lösung findet, wobei ε ein an den Algorithmus übergebener Parameter ist. Die gleiche Eigenschaft zeigt der A*-Algorithmus (Pearl, 1984), der einen beliebigen Knoten aus dem Grenzbereich auswählen kann, vorausgesetzt, dass seine *f*-Kosten innerhalb eines Faktors $1 + \varepsilon$ des Grenzknotens mit den niedrigsten *f*-Kosten liegen. Die Auswahl lässt sich so ausführen, dass die Suchkosten minimiert werden.

Es wurden auch bidirektionale Versionen von A* untersucht. Mit einer Kombination aus bidirektionalem A* und bekannten Orientierungspunkten wurden auf effiziente Weise Fahrtrouten für den Online-Kartendienst von Microsoft ermittelt (Goldberg et al., 2006). Nachdem eine Menge von Pfaden zwischen Orientierungspunkten erfasst wurde, kann der Algorithmus einen optimalen Pfad zwischen jedem Punktepaar in einem 24 Millionen Punkte umfassenden Graphen der Vereinigten Staaten finden, wobei weniger als 0,1% des Graphen durchsucht werden müssen. Zu anderen Konzepten der bidirektionalen Suche gehört eine Breitensuche, die rückwärts vom Ziel bis zu einer festgelegten Tiefe geht, woran sich eine IDA*-Suche in Vorwärtsrichtung anschließt (Dillenburg und Nelson, 1994; Manzini, 1995).

A* und andere Zustandsraum-Suchalgorithmen sind eng mit den *Branch-and-Bound*-Techniken verwandt, die in der Operationsforschung gebräuchlich sind (Lawler und Wood, 1966). Die Beziehungen zwischen Zustandsraumsuche und Branch-and-Bound wurden eingehend untersucht (Kumar und Kanal, 1983; Nau et al., 1984; Kumar et al., 1988). Martelli und Montanari (1978) demonstrierten eine Verbindung zwischen dynamischer Programmierung (siehe *Kapitel 17*) und bestimmten Arten von Zustandsraumsuche. Kumar und Kanal (1988) versuchten eine „große Vereinheitlichung“ von heuristischer Suche, dynamischer Programmierung und Branch-and-Bound-Techniken unter der Bezeichnung CDP – dem „Composite Decision Process“ (Verbundentscheidungsprozess). Da Computer in den späten 1950er und frühen 1960er Jahren höchstens über einige Tausend Wörter an Hauptspeicher verfügten, war die speicherbegrenzte heuristische Suche ein frühes Forschungsthema. Eines der ersten Suchprogramme, der *Graph Traverser* (Doran und Michie, 1966), bemüht einen Operator nach einer Bestensuche bis zur Speicherbegrenzung. IDA* (Korf, 1985a, 1985b) war der erste verbreitete optimale, speicherbegrenzte heuristische Suchalgorithmus, von dem eine große Anzahl an Varianten entwickelt wurde. Eine Analyse der Effizienz von IDA* und seiner Schwierigkeiten mit reellwertigen Heuristiken erscheint in Patrick et al. (1992).

RBFS (Korf, 1993) ist tatsächlich etwas komplizierter als der in Abbildung 3.26 gezeigte Algorithmus, der einem unabhängig entwickelten Algorithmus namens **iterative Expansion** näherkommt (Russell, 1992). RBFS verwendet eine untere sowie die obere Schranke. Mit zulässigen Heuristiken verhalten sich die beiden Algorithmen identisch, RBFS erweitert aber auch Knoten in der Reihenfolge der Bestensuche sogar mit einer nicht zulässigen Heuristik. Das Konzept, den besten alternativen Pfad zu verfolgen, ist bereits in der eleganten Prolog-Implementierung des A*-Algorithmus von Bratko (1986) und im DTA*-

Algorithmus (Russell und Wefald, 1991) zu finden. Die zuletzt genannte Arbeit beschäftigt sich auch mit Zustandsräumen und Lernen auf Metaebene.

SMA* (Simplified MA*) entstand als Versuch, MA* als Vergleichsalgorithmus für IE zu implementieren (Russell, 1992). Kaindl und Khorsand (1994) haben SMA* angewandt, um einen bidirektionalen Suchalgorithmus zu schaffen, der erheblich schneller als vorherige Algorithmen ist. Korf und Zhang (2000) beschreiben einen Teile-und-herrsche-Ansatz und Zhou und Hansen (2002) führen die speicherbegrenzte A*-Graphensuche und eine Strategie für den Wechsel zur Breitensuche ein, um die Speichereffizienz zu erhöhen (Zhou und Hansen, 2006). Korf (1995) setzt sich mit speicherbegrenzten Suchverfahren auseinander.

Die Idee, zulässige Heuristiken aus gelockerten Problemen ableiten zu können, findet sich im wegweisenden Artikel von Held und Karp (1970), die die MST-Heuristik (Minimum-Spanning-Tree, minimaler Spannbaum) heranzogen, um das Problem des Handelsreisenden (TSP) zu lösen. (Siehe Übung 3.33.)

Die Automatisierung des Lockerungsprozesses wurde erfolgreich von Prieditis (1993) implementiert, der auf früheren Arbeiten mit Mostow (Mostow und Prieditis, 1989) aufbaut. Holte und Hernadvolgyi (2001) beschreiben neuere Vorstöße in Richtung Automatisierung des Prozesses. Der Einsatz von Musterdatenbanken, um zulässige Heuristiken abzuleiten, geht auf Gasser (1995) sowie Culberson und Schaeffer (1996, 1998) zurück; disjunkte Musterdatenbanken werden von Korf und Felner (2002) beschrieben; von Edelkamp (2009) stammt eine ähnliche Methode mit symbolischen Mustern. Felner et al. (2007) zeigt, wie sich Musterdatenbanken komprimieren lassen, um Platz zu sparen. Die wahrscheinlichkeitstheoretische Interpretation von Heuristiken wurde eingehend von Pearl (1984) sowie von Hansson und Mayer (1989) untersucht.

Die bei weitem umfassendste Quelle von Heuristiken und heuristischen Suchalgorithmen ist das Buch *Heuristics* von Pearl (1984). Es bietet eine besonders gute Abdeckung der breiten Vielfalt von Ablegern und Variationen des A*-Algorithmus, einschließlich strenger Beweise ihrer formalen Eigenschaften. Von Kanal und Kumar (1988) stammt eine Anthologie wichtiger Artikel zu heuristischer Suche und Rayward-Smith et al. (1996) behandeln Ansätze aus der Operationsforschung. Artikel zu neuen Suchalgorithmen – die erstaunlicherweise immer noch entdeckt werden – erscheinen in Fachzeitschriften wie zum Beispiel *Artificial Intelligence* und *Journal of the ACM*.

Auf das Thema der parallelen Suchalgorithmen ist dieses Kapitel nicht eingegangen, unter anderem deshalb nicht, weil es eine längere Diskussion zu parallelen Computerarchitekturen voraussetzt. **Parallele Suche** ist in den 1990er Jahren zu einem populären Thema sowohl in der KI als auch in der theoretischen Informatik geworden (Mahanti und Daniels, 1993; Grama und Kumar, 1995; Crauser et al., 1998) und erlebt ein Comeback in der Ära moderner Mehrkernprozessor- und Clusterarchitekturen (Ralphs et al., 2004; Korf und Schultze, 2005). Von zunehmender Bedeutung sind auch die Suchalgorithmen für sehr große Graphen, bei denen eine Festplattenspeicherung notwendig ist (Korf, 2008).

Zusammenfassung

Dieses Kapitel hat Methoden vorgestellt, mit denen ein Agent in deterministischen, beobachtbaren, statischen und vollständig bekannten Umgebungen Aktionen auswählen kann. In derartigen Fällen kann der Agent Aktionsfolgen konstruieren, die seine Ziele erreichen. Dieser Prozess wird als **Suche** bezeichnet.

- Bevor ein Agent nach Lösungen suchen kann, muss er sein **Ziel** identifizieren und anhand des Zieles ein klar abgegrenztes **Problem** formulieren.
- Ein Problem besteht aus fünf Teilen: dem **Ausgangszustand**, einer Menge von **Aktionen**, einem **Übergangsmodell**, das die Ergebnisse dieser Aktionen beschreibt, einer **Zieltest**funktion und einer **Pfadkosten**funktion. Die Umgebung des Problems wird durch einen **Zustandsraum** dargestellt. Ein **Pfad** durch den Zustandsraum vom Ausgangszustand zu einem Zielzustand ist eine **Lösung**.
- Suchalgorithmen behandeln Zustände und Aktionen als **atomar**: Sie betrachten keinerlei interne Strukturen, die sie eventuell besitzen.
- Ein allgemeiner TREE-SEARCH-Algorithmus geht sämtlichen möglichen Pfaden nach, um eine Lösung zu finden, während ein GRAPH-SEARCH-Algorithmus die Betrachtung redundanter Pfade vermeidet.
- Suchalgorithmen werden nach **Vollständigkeit**, **Optimalität**, **Zeit**- und **Speicherkomplexität** beurteilt. Die Komplexität ist von b, dem Verzweigungsfaktor im Zustandsraum, sowie von d, der Tiefe der flachsten Lösung, abhängig.
- Methoden der **uninformierten Suche** haben nur auf die Problemdefinition Zugriff. Man unterscheidet folgende Basisalgorithmen:
  - Die **Breitensuche** erweitert zuerst den flachsten Knoten. Sie ist vollständig, optimal für einheitliche Schrittkosten, weist aber exponentielle Platzkomplexität auf.
  - Die **Suche mit einheitlichen Kosten** expandiert den Knoten mit den geringsten Pfadkosten, $g(n)$, und ist optimal für allgemeine Schrittkosten.
  - Die **Tiefensuche** erweitert zuerst den tiefsten nicht expandierten Knoten. Sie ist weder vollständig noch optimal, zeichnet sich aber durch eine lineare Platzkomplexität aus. Die **tiefenbeschränkte Suche** realisiert zusätzlich eine Tiefenbegrenzung.
  - Die **iterativ vertiefende Suche** ruft die tiefenbeschränkte Suche mit steigenden Tiefenbegrenzungen auf, bis ein Ziel gefunden wird. Sie ist vollständig, optimal für einheitliche Schrittkosten, weist eine mit der Breitensuche vergleichbare Zeitkomplexität auf und ist durch eine lineare Platzkomplexität gekennzeichnet.
  - Die **bidirektionale Suche** kann zwar die Zeitkomplexität erheblich reduzieren, ist aber nicht immer anwendbar und kann zu viel Speicher verbrauchen.

- Methoden der **informierten Suche** haben in der Regel Zugriff auf eine **Heuristikfunktion** $h(n)$, die die Kosten einer Lösung von n abschätzt.
  - Die generische **Bestensuche** (Best-First Search) wählt einen zu expandierenden Knoten entsprechend einer **Evaluierungsfunktion** aus.
  - Die **gierige Bestensuche** (Greedy Best-First Search) erweitert Knoten mit minimalem h(n). Zwar ist sie nicht optimal, aber oftmals effizient.
  - Die **A*-Suche** erweitert Knoten mit minimalem $f(n) = g(n) + h(n)$. A* ist vollständig und optimal, sofern $h(n)$ zulässig (für TREE-SEARCH) oder konsistent (für GRAPH-SEARCH) ist. Die Platzkomplexität von A* ist dennoch hinderlich.
  - **RBFS** (Recursive Best-First Search, rekursive Bestensuche) und **SMA*** (Simplified Memory-Bounded A*, vereinfachter speicherbegrenzter A*) sind robuste, optimale Suchalgorithmen mit beschränktem Speicherbedarf; steht genügend Zeit zur Verfügung, können sie Probleme lösen, wozu A* wegen Speichermangels nicht in der Lage ist.
- Die Leistung von heuristischen Suchalgorithmen hängt von der Qualität der Heuristikfunktion ab. Manchmal lassen sich gute Heuristiken konstruieren, indem man die Problemdefinition lockert, vorberechnete Lösungskosten für Unterprobleme in einer Musterdatenbank speichert oder aus Erfahrungen mit der Problemklasse lernt.

Übungen zu Kapitel 3

Lösungshinweise

1 Erläutern Sie, warum einer Zielformulierung eine Problemformulierung folgen muss.

2 Geben Sie eine vollständige Problemformulierung für jedes der folgenden Probleme an. Wählen Sie eine hinreichend genaue Formulierung, um sie implementieren zu können.

a. In einer Reihe stehen sechs verschlossene Glasbehälter. In den ersten fünf Behältern befindet sich jeweils ein Schlüssel, mit dem sich der nächste Behälter in der Reihe aufschließen lässt. Der letzte Kasten enthält eine Banane. Sie besitzen den Schlüssel zum ersten Kasten und Sie möchten die Banane haben.

b. Sie beginnen mit der Sequenz ABABAECCEC oder ganz allgemein mit jeder beliebigen Sequenz, die aus A, B, C und E besteht. Diese Sequenz können Sie mithilfe der folgenden Äquivalenzen umwandeln: AC = E, AB = BC, BB = E und Ex = x für jedes x. Zum Beispiel lässt sich ABBC in AEC, dann nach AC und schließlich nach E umwandeln. Das Ziel besteht darin, die Sequenz E zu erzeugen.

c. In einem $n \times n$-Raster ist jedes Quadrat anfangs entweder unlackierter Fußboden oder ein bodenloses Loch. Als Ausgangsposition stehen Sie auf einem unlackierten Fußbodenquadrat und können entweder das Quadrat unter sich färben oder zu einem benachbarten unlackierten Quadrat gehen. Der gesamte Fußboden soll lackiert werden.

d. Ein Containerschiff liegt im Hafen und ist mit Containern beladen – 13 Reihen mit jeweils 13 Containern in der Breite und 5 Containern in der Höhe. Sie steuern einen Kran, der sich zu jeder Position über dem Schiff bewegen kann, nehmen einen darunter befindlichen Container auf und transportieren ihn auf das Dock. Das Schiff soll vollständig entladen werden.

3 Die Felder in einem 9×9-Gitter lassen sich rot oder blau färben. Anfangs sind alle Felder des Gitters blau. Die Farbe jedes Feldes lässt sich aber beliebig oft ändern. Nehmen Sie an, dass das Gitter in 9 Teilquadrate zu je 3×3 Felder unterteilt ist. Alle Felder eines Teilquadrates sollen die gleiche Farbe haben, benachbarte Teilquadrate müssen unterschiedliche Farben aufweisen.

a. Formulieren Sie dieses Problem in unkomplizierter Weise. Berechnen Sie die Größe des Zustandsraumes.

b. Ein Feld muss nur einmal gefärbt werden. Formulieren Sie das Problem neu und berechnen Sie die Größe des Zustandsraumes. Würde die Breitensuche für Graphen bei diesem Problem schneller als bei dem in (a) laufen? Wie verhält es sich mit iterativ vertiefender Baumsuche?

c. Entsprechend der Zielstellung müssen wir nur Färbungen berücksichtigen, wo jedes Teilquadrat einheitlich gefärbt ist. Formulieren Sie das Problem neu und berechnen Sie die Größe des Zustandsraumes.

d. Wie viele Lösungen hat dieses Problem?

e. Die Teile (b) und (c) haben das ursprüngliche Problem (a) erfolgreich abstrahiert. Können Sie eine Übersetzung von Lösungen in Problem (c) zu Lösungen in Problem (b) und von Lösungen in Problem (b) zu Lösungen für Problem (a) angeben?

4 Nehmen Sie an, zwei Freunde wohnen in verschiedenen Städten, die auf einer Landkarte wie zum Beispiel der in Abbildung 3.2 gezeigten Karte von Rumänien eingezeichnet sind. Bei jedem Zug können wir gleichzeitig jeden Freund zu einer benachbarten Stadt auf der Karte bewegen. Die Zeitdauer, um von der Stadt i zur Nachbarstadt j zu gelangen, ist gleich der Straßenentfernung $d(i, j)$ zwischen den Städten, wobei aber bei jedem Zug der ankommende Freund warten muss, bis der andere eingetroffen ist (und den ersten auf seinem Mobiltelefon anruft), bevor der nächste Zug beginnen kann. Die beiden Freunde sollen sich möglichst schnell treffen.

a. Schreiben Sie eine detaillierte Formulierung für dieses Suchproblem. (Es dürfte hierzu hilfreich sein, eine formale Notation zu definieren.)

b. Es sei $D(i, j)$ die Luftliniendistanz zwischen den Städten i und j. Welche der folgenden Heuristikfunktionen sind zulässig? (i) $D(i, j)$; (ii) $2 \cdot D(i, j)$; (iii) $D(i, j)/2$.

c. Gibt es vollständig verbundene Karten, für die keine Lösung existiert?

d. Gibt es Karten, in denen es sämtliche Lösungen erfordern, dass ein Freund dieselbe Stadt zweimal besucht?

5 Zeigen Sie, dass die 8-Puzzle-Zustände in zwei disjunkte Mengen unterteilt sind, sodass jeder Zustand von jedem anderen Zustand in derselben Menge erreichbar ist, während kein Zustand von irgendeinem Zustand in der anderen Menge erreichbar ist. (*Hinweis*: Siehe Berlekamp et al. (1982).) Entwickeln Sie eine Prozedur, um zu entscheiden, zu welcher Klasse ein bestimmter Zustand gehört, und erläutern Sie, warum dies für die Erzeugung zufälliger Zustände sinnvoll ist.

6 Betrachten Sie das n-Damen-Problem unter Verwendung der „effizienten" inkrementellen Formulierung, die im Anschluss an Abbildung 3.5 diskutiert wird. Erklären Sie, warum die Größe des Zustandsraumes mindestens $\sqrt[3]{n!}$ beträgt, und schätzen Sie das größte n ab, für das eine umfassende Untersuchung praktikabel ist. (Hinweis: Leiten Sie eine Untergrenze für den Verzweigungsfaktor ab, indem Sie die maximale Anzahl der Quadrate betrachten, die eine Dame in jeder Spalte angreifen kann.)

7 Betrachten Sie das Problem, den kürzesten Pfad zwischen zwei Punkten auf einer Ebene zu finden, die konvexe polygonale Hindernisse aufweist, wie in ▶ Abbildung 3.31 gezeigt. Dies ist eine Idealisierung des Problems, das ein Roboter lösen muss, um in einer überfüllten Umgebung zu navigieren.

a. Nehmen Sie an, dass der Zustandsraum aus allen Positionen (x, y) in der Ebene besteht. Wie viele Zustände gibt es? Wie viele Pfade gibt es zum Ziel?

b. Erläutern Sie kurz, warum der kürzeste Pfad von einem Polygoneckpunkt zu irgendeinem anderen in der Ebene aus geradlinigen Segmenten bestehen muss, die einige der Eckpunkte der Polygone verbinden. Definieren Sie jetzt einen guten Zustandsraum. Wie groß ist dieser Zustandsraum?

c. Definieren Sie die erforderlichen Funktionen, um das Suchproblem zu implementieren, einschließlich einer ACTIONS-Funktion, die einen Eckpunkt als Eingabe übernimmt und eine Menge von Vektoren zurückgibt, die jeweils den aktuellen Eckpunkt einem der Eckpunkte zuordnen, die sich in einer geraden Linie erreichen lassen. (Vergessen Sie nicht die Nachbarn im selben Polygon.) Verwenden Sie die Luftliniendistanz für die Heuristikfunktion.

d. Wenden Sie einen oder mehrere der Algorithmen aus diesem Kapitel an, um Probleme in diesem Bereich zu lösen, und erörtern Sie ihre Leistung.

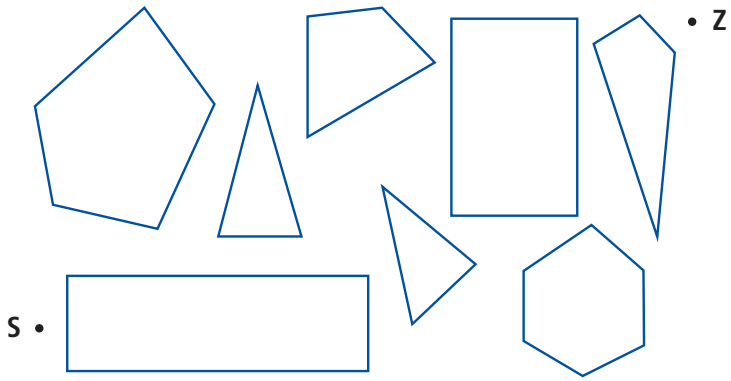


Abbildung 3.31: Ein Szenario mit polygonalen Hindernissen. *S* und *Z* sind die Start- und Zielzustände.

8 In *Abschnitt 3.1.1* haben wir gesagt, dass wir keine Probleme mit negativen Pfadkosten betrachten wollen. In dieser Übung erläutern wir diese Entscheidung ausführlicher.

a. Nehmen Sie an, dass Aktionen beliebig große negative Kosten haben können. Erklären Sie, warum diese Möglichkeit jeden optimalen Algorithmus zwingen würde, den gesamten Zustandsraum zu durchsuchen.

b. Hilft es, wenn wir darauf bestehen, dass Schrittkosten größer oder gleich einer negativen Konstanten c sein müssen? Betrachten Sie sowohl Bäume als auch Graphen.

c. Nehmen Sie an, dass eine Menge von Aktionen eine Schleife im Zustandsraum bildet, sodass die Ausführung der Menge in einer bestimmten Reihenfolge zu keiner Änderung am Zustand führt. Was bedeutet es für das optimale Verhalten eines Agenten in einer solchen Umgebung, wenn alle diese Aktionen negative Kosten haben?

d. Man kann sich leicht Aktionen mit hohen negativen Kosten vorstellen, sogar in Bereichen wie der Routensuche. Zum Beispiel könnten einige Straßenzüge landschaftlich so reizvoll sein, dass dies die normalen Kosten im Hinblick auf Zeit und Treibstoff bei weitem übertrifft. Erklären Sie präzise innerhalb des Kontextes der Zustandsraumsuche, warum Menschen nicht unendlich lange in malerischen Gebieten kreisen, und erklären Sie, wie Sie den Zustandsraum und die Aktionen für die Routensuche definieren, sodass auch künstliche Agenten Schleifen vermeiden können.

e. Können Sie sich einen realen Bereich vorstellen, in dem Schrittkosten Schleifen verursachen?

9 Das Problem der **Missionare und Kannibalen** wird im Allgemeinen wie folgt formuliert: Drei Missionare und drei Kannibalen befinden sich auf einer Seite eines Flusses und sie haben ein Boot, das ein oder zwei Menschen aufnehmen kann. Suchen Sie eine Möglichkeit, alle auf die andere Seite zu bringen, ohne je eine Gruppe von Missionaren auf einer Seite zu lassen, wo ihnen die Kannibalen zahlenmäßig überlegen sind. Dieses Problem ist in der KI sehr bekannt, weil es Thema des ersten Artikels war, der zur Problemformulierung aus analytischer Sicht erschien (Amarel, 1968).

a. Formulieren Sie das Problem präzise und treffen Sie nur die Unterscheidungen, die für die Sicherstellung einer gültigen Lösung erforderlich sind. Zeichnen Sie eine Skizze des vollständigen Zustandsraumes.

b. Implementieren und lösen Sie das Problem optimal unter Verwendung eines geeigneten Suchalgorithmus. Ist es sinnvoll, auf wiederholte Zustände zu prüfen?

c. Warum ist es Ihrer Meinung nach für Menschen schwierig, dieses Rätsel zu lösen, wenn man überlegt, dass der Zustandsraum so einfach ist?

10 Definieren Sie in eigenen Worten die folgenden Begriffe: Zustand, Zustandsraum, Suchbaum, Suchknoten, Ziel, Aktion, Übergangsmodell und Verzweigungsfaktor.

11 Wie unterscheiden sich Weltzustand, Zustandsbeschreibung und Suchknoten? Weshalb ist diese Unterscheidung nützlich?

12 Eine Aktion wie zum Beispiel $Go(Sibiu)$ besteht eigentlich aus einer langen Sequenz feinkörnigerer Aktionen: Wagen starten, Handbremse lösen, beschleunigen usw. Mit zusammengesetzten derartigen Aktionen verringert sich die Anzahl der Schritte in einer Lösungssequenz und damit auch die Suchzeit. Dies kann man nun ins logische Extrem treiben, indem man Superzusammensetzungen von Aktionen aus jeder möglichen Sequenz von Go-Aktionen macht. Dann wird jede Probleminstanz durch eine einzige Superzusammensetzungsaktion wie zum Beispiel $Go(Sibiu)Go(Rimnicu\ Vilcea)Go(Pitesti)Go(Bukarest)$ gelöst. Erläutern Sie, wie die Suche in dieser Formulierung funktionieren würde. Ist dieses Konzept zweckmäßig, um die Problemlösung zu beschleunigen?

13 Führt ein endlicher Zustandsraum immer zu einem endlichen Suchbaum? Wie verhält es sich bei einem endlichen Zustandsraum, der ein Baum ist? Können Sie Genaueres darüber sagen, welche Arten von Zustandsräumen immer zu endlichen Suchbäumen führen? (Übernommen aus Bender, 1996.)

14 Beweisen Sie, dass der GRAPH-SEARCH-Algorithmus die in Abbildung 3.9 dargestellte Trennungseigenschaft der Graphensuche erfüllt. (*Hinweis*: Zeigen Sie zunächst, dass die Eigenschaft am Ausgangspunkt gilt. Weisen Sie dann nach, dass die Eigenschaft auch nach der Iteration des Algorithmus erfüllt ist, wenn sie davor gilt.) Beschreiben Sie einen Suchalgorithmus, der die Eigenschaft verletzt.

15 Welche der folgenden Aussagen sind richtig und welche falsch? Erläutern Sie Ihre Antworten.

a. Die Tiefensuche erweitert immer mindestens so viele Knoten wie eine A*-Suche mit einer zulässigen Heuristik.

b. Für das 8-Puzzle ist $h(n) = 0$ eine zulässige Heuristik.

c. A* ist für die Robotik nicht geeignet, da Wahrnehmungen, Zustände und Aktionen stetig sind.

d. Die Breitensuche ist vollständig, selbst wenn Schrittkosten von 0 zugelassen werden.

e. Ein Turm kann auf einem Schachbrett in gerader Linie senkrecht oder waagerecht eine beliebige Anzahl Felder ziehen, jedoch nicht über andere Figuren springen. Die Manhattan-Distanz ist eine zulässige Heuristik für das Problem, den Turm von Feld A zu Feld B mit der kleinsten Anzahl von Zügen zu bewegen.

16 Das Starterset einer Holzeisenbahn enthält die in ▶ Abbildung 3.32 gezeigten Teile. Die Aufgabe besteht darin, diese Gleisstücke zu einer Strecke ohne überlappende Streckenteile und ohne offene Stumpfgleise, von wo aus der Zug auf den Fußboden rollen könnte, zusammenzufügen.

a. Nehmen Sie an, dass die Gleisstücke exakt ohne Spiel zusammenpassen. Geben Sie eine genaue Formulierung der Aufgabe als Suchproblem an.

b. Identifizieren Sie einen geeigneten uninformierten Suchalgorithmus für diese Aufgabe und erläutern Sie Ihre Auswahl.

c. Erläutern Sie, warum das Problem unlösbar wird, wenn eine der Weichen fehlt.

d. Geben Sie eine obere Grenze für die Gesamtgröße des Zustandsraumes an, der durch Ihre Formulierung definiert wird. (*Hinweis*: Überlegen Sie sich den maximalen Verzweigungsfaktor für den Konstruktionsvorgang und die maximale Tiefe, wobei Sie das Problem der überlappenden und offenen Streckenteile außer Acht lassen. Gehen Sie zu Beginn davon aus, dass jedes Teil nur einmal vorhanden ist.)

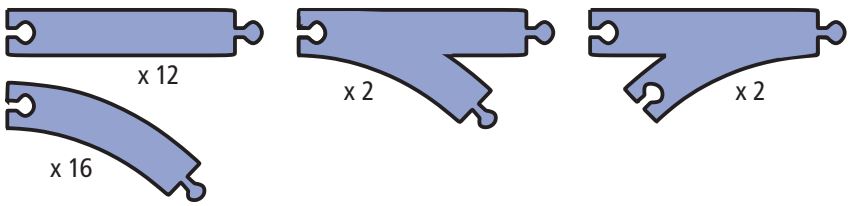


Abbildung 3.32: Die Gleisstücke, die im Starter-Set einer Holzeisenbahn enthalten sind. An jedem Teil steht die Anzahl, wie oft es im Set vorhanden ist. Bogenstücke und Weichen lassen sich drehen, sodass Kurven in jeder Richtung möglich sind. Die Gleisbögen erstrecken sich jeweils über einen Winkel von 45°.

17 Implementieren Sie zwei Versionen der Funktion RESULT(s, a) für das 8-Puzzle: eine, die die Datenstruktur für den übergeordneten Knoten s kopiert und bearbeitet, und eine, die den übergeordneten Zustand direkt modifiziert (und Modifikationen bei Bedarf rückgängig macht). Schreiben Sie Versionen der iterativ vertiefenden Tiefensuche, die diese Funktionen verwendet, und vergleichen Sie ihre Leistung.

18 In *Abschnitt 3.4.5* haben wir die **iterativ verlängernde Suche** erwähnt, ein iteratives Analogon zur Suche mit einheitlichen Kosten. Die Idee dabei ist, steigende Obergrenzen für Pfadkosten zu verwenden. Wird ein Knoten erzeugt, dessen Pfadkosten die aktuelle Obergrenze überschreiten, wird er sofort verworfen. Für jede neue Iteration wird die Obergrenze auf die niedrigsten Pfadkosten der in der vorherigen Iteration verworfenen Knoten gesetzt.

a. Zeigen Sie, dass dieser Algorithmus optimal für allgemeine Pfadkosten ist.

b. Betrachten Sie einen einheitlichen Baum mit Verzweigungsfaktor b, einer Lösungstiefe d und einheitlichen Schrittkosten. Wie viele Iterationen sind für die iterative Verlängerung erforderlich?

c. Betrachten Sie jetzt die Schrittkosten, die aus dem stetigen Bereich $[0, 1]$ entstehen, wobei $0 < \varepsilon < 1$ ist. Wie viele Iterationen sind im ungünstigsten Fall erforderlich?

d. Implementieren Sie den Algorithmus und wenden Sie ihn auf Instanzen des 8-Puzzles und des Handlungsreisenden-Problems an. Vergleichen Sie die Leistung des Algorithmus mit der der Suche mit einheitlichen Kosten und kommentieren Sie Ihre Ergebnisse.

19 Beschreiben Sie einen Zustandsraum, in dem sich eine iterativ vertiefende Suche sehr viel schlechter verhält als eine Breitensuche (z.B. $O(n^2)$ im Vergleich zu $O(n)$).

20 Schreiben Sie ein Programm, das als Eingabe die URLs von zwei Webseiten übernimmt und einen Pfad der Links von der einen zur anderen ermittelt. Welche Suchstrategie ist am besten geeignet? Ist eine Verwendung der bidirektionalen Suche sinnvoll? Könnte eine Suchmaschine verwendet werden, um eine Vorgängerfunktion zu implementieren?

21 Betrachten Sie das Staubsaugerwelt-Problem, das in Abbildung 2.2 definiert wurde.

a. Welcher der in diesem Kapitel definierten Algorithmen ist für dieses Problem geeignet? Sollte der Algorithmus mit Baumsuche oder Graphensuche arbeiten?

b. Wenden Sie den von Ihnen gewählten Algorithmus an, um eine optimale Aktionsfolge für eine 3×3-Welt zu berechnen, in deren Ausgangszustand Schmutz in den drei oberen Quadraten liegt und der Agent sich in der Mitte befindet.

c. Konstruieren Sie einen Suchagenten für die Staubsaugerwelt und bewerten Sie seine Leistung in einer Menge von 3×3-Welten mit einer Wahrscheinlichkeit von 0,2 von Schmutz in jedem Quadrat. Berücksichtigen Sie die Suchkosten sowie die Pfadkosten bei der Leistungsbewertung und verwenden Sie einen vernünftigen Umrechnungssatz.

d. Vergleichen Sie Ihren besten Suchagenten mit einem einfachen zufälligen Reflexagenten, der saugt, wenn Schmutz vorhanden ist, und sich andernfalls zufällig bewegt.

e. Betrachten Sie, was passiert, wenn die Welt auf $n \times n$ vergrößert wird. Inwieweit variiert die Leistung des Suchagenten und des Reflexagenten mit der Größe von n?

22 Beweisen Sie die folgenden Aussagen oder geben Sie ein Gegenbeispiel an:

a. Die Breitensuche ist ein Spezialfall der Suche mit einheitlichen Kosten.

b. Die Tiefensuche ist ein Spezialfall der Bestensuche für Bäume.

c. Die Suche mit einheitlichen Kosten ist ein Spezialfall der A*-Suche.

23 Vergleichen Sie die Leistung von A* und RBFS für eine Menge von zufällig erzeugten Problemen in den Bereichen 8-Puzzle (mit Manhattan-Distanz) und TSP (mit MST – siehe Übung 3.33). Erörtern Sie Ihre Ergebnisse. Wie wirkt es sich auf die Leistung von RBFS aus, wenn zu den heuristischen Werten im 8-Puzzle-Bereich eine kleine Zufallszahl addiert wird?

24 Verfolgen Sie den Ablauf der A*-Suche bei der Lösung des Problems, von Lugoj nach Bukarest zu gelangen, wobei als Heuristik die Luftliniendistanz verwendet wird. Mit anderen Worten: Zeigen Sie die Sequenz der Knoten, die der Algorithmus betrachtet, und geben Sie die f-, g- und h-Punktzahl für jeden Knoten an.

25 Manchmal existiert für ein Problem keine gute Bewertungsfunktion, doch es gibt eine gute Vergleichsmethode: ein Weg, um festzustellen, ob der eine Knoten besser als ein anderer ist, ohne beiden numerische Werte zuzuweisen. Zeigen Sie, dass dies für eine Bestensuche genügt. Gibt es ein Analogon von A* für diese Einrichtung?

26 Entwerfen Sie einen Zustandsraum, in dem A* mithilfe von GRAPH-SEARCH eine suboptimale Lösung mit einer $h(n)$-Funktion zurückgibt, die zulässig, aber inkonsistent ist.

27 Genaue Heuristiken verringern nicht unbedingt die Suchzeit im ungünstigsten Fall. Definieren Sie für eine gegebene Tiefe d ein Suchproblem mit einem Zielknoten bei Tiefe d und schreiben Sie eine Heuristikfunktion, sodass gilt $|h(n) - h^*(n)| \leq O(\log h^*(n))$, aber A* alle Knoten mit einer Tiefe kleiner als d erweitert.

28 Der **heuristische Pfadalgorithmus** (Pohl, 1977) ist eine Bestensuche mit der Bewertungsfunktion $f(n) = (2 - w)g(n) + wh(n)$. Für welche Werte von w ist die Suche vollständig? Für welche Werte ist sie optimal, vorausgesetzt, dass h zulässig ist? Welche Art der Suche wird für $w = 0$, $w = 1$ und $w = 2$ ausgeführt?

29 Betrachten Sie die unbegrenzte Version des regulären 2D-Rasters in Abbildung 3.9. Der Ausgangszustand befindet sich am Ursprung $(0, 0)$ und der Zielzustand bei (x, y).

a. Wie groß ist der Verzweigungsfaktor b in diesem Zustandsraum?

b. Wie viele unterschiedliche Zustände gibt es bei Tiefe k (für $k > 0$)?

c. Wie groß ist die maximale Anzahl der Knoten, die die Breitensuche für Bäume erweitert?

d. Wie groß ist die maximale Anzahl der Knoten, die die Breitensuche für Graphen erweitert?

e. Ist $h = |u - x| + |v - y|$ eine zulässige Heuristik für einen Zustand bei (u, v)? Erläutern Sie Ihre Antwort.

f. Wie viele Knoten werden durch die A*-Graphensuche mithilfe von h erweitert?

g. Bleibt h zulässig, wenn Verbindungen entfernt werden?

h. Bleibt h zulässig, wenn Verbindungen zwischen nicht benachbarten Zuständen hinzugefügt werden?

30 Betrachten Sie das Problem für das Ziehen von k Springern von k Ausgangsquadraten $s_1, ..., s_k$ zu k Zielquadraten $g_1, ..., g_k$ auf einem nicht begrenzten Schachbrett, wobei die Regel gilt, dass zwei Springer nicht zur selben Zeit auf demselben Feld landen dürfen. Jede Aktion besteht aus dem Ziehen von bis zu k Springern gleichzeitig. Das Manöver soll mit der kleinsten Anzahl von Aktionen abgeschlossen werden.

a. Wie groß ist der maximale Verzweigungsfaktor in diesem Zustandsraum, ausgedrückt als Funktion von k?

b. Nehmen Sie h_i als zulässige Heuristik für das Problem an, Springer i allein zu Ziel g_i zu ziehen. Welche der folgenden Heuristiken ist für das k-Springerproblem zulässig? Welche ist die beste dieser Heuristiken?

(i) $\min\{h_1, \ldots, h_k\}$.

(ii) $\max\{h_1, \ldots, h_k\}$.

(iii) $\sum_{i=1}^{k} h_i$.

c. Wiederholen Sie (b) für den Fall, in dem nur ein Springer auf einmal gezogen werden darf.

31 *Abschnitt 3.5.1* hat gezeigt, dass die als Heuristik verwendete Luftliniendistanz die gierige Bestensuche vom Problem abbringen kann, von Iasi nach Fagaras zu gelangen. Allerdings ist die Heuristik perfekt für das entgegengesetzte Problem geeignet, nämlich den Weg von Fagaras nach Iasi zu suchen. Gibt es Probleme, für die die Heuristik in beiden Richtungen irreführend ist?

32 Beweisen Sie, dass eine Heuristik zulässig sein muss, wenn sie konsistent ist. Konstruieren Sie eine zulässige Heuristik, die nicht konsistent ist.

33 Das Problem des Handelsreisenden (Traveling Salesman Problem, TSP) lässt sich mit der MST-Heuristik (Minimum-Spanning-Tree, minimaler Spannbaum) lösen, die die Kosten für die Vervollständigung einer Tour bewertet, wenn eine partielle Tour bereits konstruiert worden ist. Die MST-Kosten einer Menge von Städten ist die kleinste Summe der Verbindungskosten eines beliebigen Baumes, der sämtliche Städte verbindet.

a. Zeigen Sie, wie sich diese Heuristik aus einer gelockerten TSP-Version ableiten lässt.

b. Zeigen Sie, dass die MST-Heuristik die Luftliniendistanz dominiert.

c. Schreiben Sie einen Problemgenerator für Instanzen des TSP, in denen Städte durch zufällige Punkte im Einheitsquadrat dargestellt werden.

d. Suchen Sie in der Literatur einen effizienten Algorithmus für die Konstruktion der MST-Heuristik und verwenden Sie ihn mit der A*-Graphensuche, um Instanzen des TSP zu lösen.

34 *Abschnitt 3.6.2* hat die Lockerung des 8-Puzzles definiert, wobei ein Feld von Quadrat A nach Quadrat B verschoben werden kann, wenn B leer ist. Die genaue Lösung dieses Problems definiert die **Heuristik von Gaschnig** (Gaschnig, 1979). Erläutern Sie, warum Gaschnigs Heuristik mindestens ebenso genau wie h_1 (falsch platzierte Felder) ist, und zeigen Sie Fälle, in denen sie genauer als h_1 und h_2 (Manhattan-Distanz) ist. Erläutern Sie, wie sich Gaschnigs Heuristik effizient berechnen lässt.

35 Für das 8-Puzzle haben wir zwei einfache Heuristiken angegeben: Manhattan-Distanz und falsch platzierte Felder. In der Literatur findet man mehrere Heuristiken, die behaupten, für dieses Problem besser geeignet zu sein – siehe zum Beispiel Nilsson (1971), Mostow und Prieditis (1989) sowie Hansson et al. (1992). Überprüfen Sie diese Behauptungen, indem Sie die Heuristiken implementieren und die Leistung der resultierenden Algorithmen vergleichen.

Über die klassische Suche hinaus

4

ÜBERBLICK

In diesem Kapitel lockern wir die vereinfachenden Annahmen des vorhergehenden Kapitels und kommen dabei der realen Welt näher.

Kapitel 3 hat sich mit einer einzigen Kategorie von Problemen beschäftigt: beobachtbare, deterministische, bekannte Umgebungen, wo die Lösung eine Sequenz von Aktionen ist. Dieses Kapitel zeigt nun, was passiert, wenn diese Annahmen gelockert werden. Wir beginnen mit einem recht einfachen Fall: Die *Abschnitte 4.1* und *4.2* stellen Algorithmen vor, die eine rein **lokale Suche** im Zustandsraum durchführen und einen oder mehrere aktuelle Zustände auswerten und abändern, anstatt systematisch Pfade von einem Ausgangszustand aus zu erkunden. Diese Algorithmen sind geeignet für Probleme, bei denen es nur auf den Lösungszustand ankommt und nicht auf die Pfadkosten, um ihn zu erreichen. Die Algorithmenfamilie für lokale Suchen beinhaltet Methoden, die durch die statistische Physik (**Simulated Annealing** (**simulierte Abkühlung**)) sowie durch die Evolutionsbiologie (**genetische Algorithmen**) inspiriert wurden.

In den *Abschnitten 4.3* und *4.4* untersuchen wir dann, was passiert, wenn wir die Annahmen für Determinismus und Beobachtbarkeit lockern. Im Kern geht es darum, dass ein Agent unter jeder **Kontingenz**, die ihm seine Wahrnehmungen gegebenenfalls offenbaren, berücksichtigen muss, was zu tun ist, wenn er nicht genau vorhersagen kann, welche Wahrnehmungen er empfangen wird. Bei teilweiser Beobachtbarkeit muss der Agent auch die Zustände verfolgen, in denen er sich befinden könnte.

Schließlich betrachtet *Abschnitt 4.5* die **Online-Suche**, wobei der Agent anfangs mit einem völlig unbekannten Zustandsraum konfrontiert wird und diesen erkunden muss.

4.1 Lokale Suchalgorithmen und Optimierungsprobleme

Die bisher vorgestellten Suchalgorithmen waren darauf ausgelegt, die Suchräume systematisch zu erkunden. Diese Systematik wurde erzielt, indem man einen oder mehrere Pfade im Speicher hielt und aufzeichnete, welche Alternativen an jedem Punkt entlang des Pfades erkundet wurden. Sobald ein Ziel gefunden ist, bildet der *Pfad* zu diesem Ziel auch eine *Lösung* für das Problem. Bei vielen Problemen ist jedoch der Pfad zu den Zielen nicht von Bedeutung. Im 8-Damen-Problem beispielsweise (siehe *Abschnitt 3.2.1*) spielt nur die endgültige Konfiguration der Damen eine Rolle und nicht die Reihenfolge, in der sie auf das Brett gestellt werden. Die gleiche generelle Eigenschaft gilt für viele wichtige Anwendungen, wie zum Beispiel den Entwurf von integrierten Schaltungen, das Layout von Fertigungshallen, die Terminplanung von Produktionsaufträgen, die automatische Programmierung, die Netzoptimierung in der Telekommunikation, die Tourenplanung sowie die Portfolioverwaltung.

Wenn der Pfad zum Ziel keine Rolle spielt, könnten wir eine andere Klasse von Algorithmen in Betracht ziehen, die sich überhaupt nicht um die Pfade kümmern. Algorithmen für die **lokale Suche** arbeiten mit einem einzelnen **aktuellen Knoten** (anstelle von mehreren Pfaden) und bewegen sich im Allgemeinen nur zu Nachbarn dieses Knotens. Typischerweise werden die Pfade, die bei der Suche verfolgt werden, nicht beibehalten. Obwohl lokale Suchalgorithmen nicht systematisch sind, haben sie zwei wesentliche Vorteile: (1) Sie brauchen sehr wenig Speicher – normalerweise eine konstante Menge – und (2) sie können häufig sinnvolle Lösungen in großen oder unendlichen (stetigen) Zustandsräumen finden, für die systematische Algorithmen nicht geeignet sind.

Neben der Ermittlung von Zielen sind lokale Suchalgorithmen auch praktisch für die Lösungen reiner **Optimierungsprobleme**, bei denen das Ziel darin besteht, den besten Zustand gemäß einer **Zielfunktion** zu finden. Viele Optimierungsprobleme passen nicht in das „Standard"-Suchmodell, das *Kapitel 3* eingeführt hat. Beispielsweise gibt es in der Natur eine Zielfunktion – die reproduktive Fitness –, für die man die Darwinsche Evolution als Optimierungsversuch ansehen könnte, wobei es aber weder einen „Zieltest" noch „Pfadkosten" für dieses Problem gibt.

Um die lokale Suche zu verstehen, halten wir es für sinnvoll, **Zustandsraum-Landschaften** zu betrachten (wie in ▶ Abbildung 4.1 gezeigt). Eine Landschaft hat sowohl eine „Position" (durch den Zustand definiert) als auch eine „Erhebung" (durch den Wert der heuristischen Kostenfunktion oder Zielfunktion definiert). Wenn die Erhebung den Kosten entspricht, besteht das Ziel darin, das tiefste Tal zu finden – ein **globales Minimum**; entspricht die Erhebung einer Zielfunktion, dann geht es darum, den höchsten Gipfel zu finden – ein **globales Maximum**. (Beide Varianten lassen sich in die jeweils andere umwandeln, indem Sie einfach ein Minuszeichen einfügen.) Lokale Suchalgorithmen erkunden diese Landschaft. Ein **vollständiger** lokaler Suchalgorithmus findet immer ein Ziel, wenn es ein solches gibt; ein **optimaler** Algorithmus findet immer ein globales Minimum/Maximum.

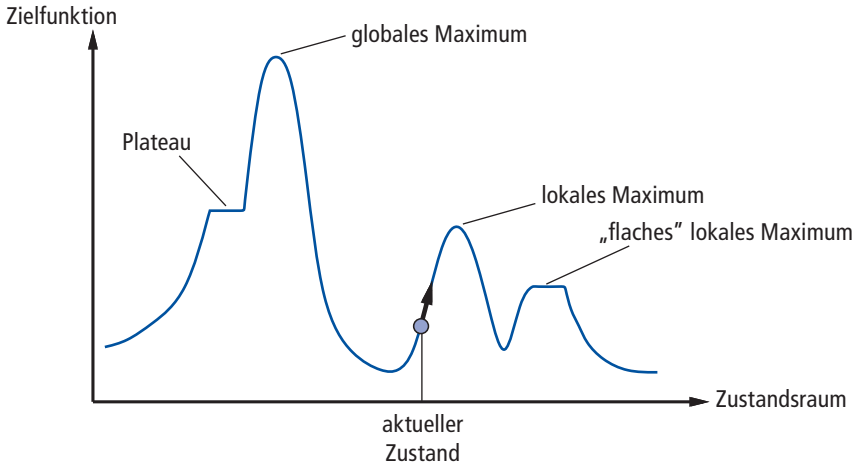


Abbildung 4.1: Eine eindimensionale Zustandsraum-Landschaft, wobei die Erhebung der Zielfunktion entspricht. Die Aufgabe besteht darin, das globale Maximum zu finden. Eine Hillclimbing-Suche („Bergsteigen") modifiziert den aktuellen Zustand und versucht ihn zu verbessern, wie durch den Pfeil gezeigt. Die verschiedenen topografischen Funktionsmerkmale werden im Text definiert.

4.1.1 Hillclimbing-Suche (Bergsteigeralgorithmus)

Der **Hillclimbing**-Suchalgorithmus (Version des **steilsten Anstieges**) ist in ▶ Abbildung 4.2 dargestellt. Es handelt sich dabei einfach um eine Schleife, die ständig in die Richtung des steigenden Wertes verläuft, d.h. bergauf. Sie terminiert, wenn sie einen „Gipfel" erreicht, wo kein Nachbar einen höheren Wert hat. Der Algorithmus verwaltet keinen Suchbaum, sodass die Datenstruktur für den aktuellen Knoten nur den Zustand und den Wert der Zielfunktion aufzeichnen muss. Dieses Bergsteigen sieht nicht über die unmittelbaren Nachbarn des aktuellen Zustandes hinaus. Das erinnert an einen Ver-

such, die Spitze des Mount Everest in dickem Nebel zu finden, während man gleichzeitig an Gedächtnisverlust leidet.

```
function HILL-CLIMBING(problem) returns einen Zustand, der ein lokales Maximum ist
  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
    neighbor ← ein current-Nachfolger mit höchstem Wert
    if neighbor.VALUE ≤ current.VALUE then return current.STATE
    current ← neighbor
```

Abbildung 4.2: Der Bergsteiger-Suchalgorithmus, der die grundlegendste lokale Suchtechnik darstellt. Bei jedem Schritt wird der aktuelle Knoten durch den besten Nachbarn ersetzt. In dieser Version ist das der Nachbar mit dem höchsten Wert (VALUE). Verwendet man dagegen eine heuristische Kostenschätzung h, wird der Nachbar mit dem kleinsten h gesucht.

Um das Bergsteigen zu verdeutlichen, verwenden wir wieder das **8-Damen-Problem**, das in *Abschnitt 3.2.1* vorgestellt wurde. Lokale Suchalgorithmen verwenden normalerweise eine **vollständige Zustandsformulierung**, wobei jeder Zustand acht Damen auf dem Brett hat – eine pro Spalte. Bei den Nachfolgern eines Zustandes handelt es sich um alle möglichen Zustände, die durch Ziehen einer einzelnen Dame auf ein anderes Quadrat in derselben Spalte entstehen (sodass jeder Zustand $8 \times 7 = 56$ Nachfolger hat). Die heuristische Kostenfunktion h ist die Anzahl der Damenpaare, die einander direkt oder indirekt angreifen. Das globale Minimum dieser Funktion ist null, was nur bei perfekten Lösungen auftritt. ▶ Abbildung 4.3(a) zeigt einen Zustand mit $h = 17$. Außerdem sind die Werte aller seiner Nachfolger angegeben, wobei die besten Nachfolger den Wert $h = 12$ haben. Bergsteigeralgorithmen wählen normalerweise zufällig aus der Menge der besten Nachfolger aus, wenn es nicht nur einen besten Nachfolger gibt.

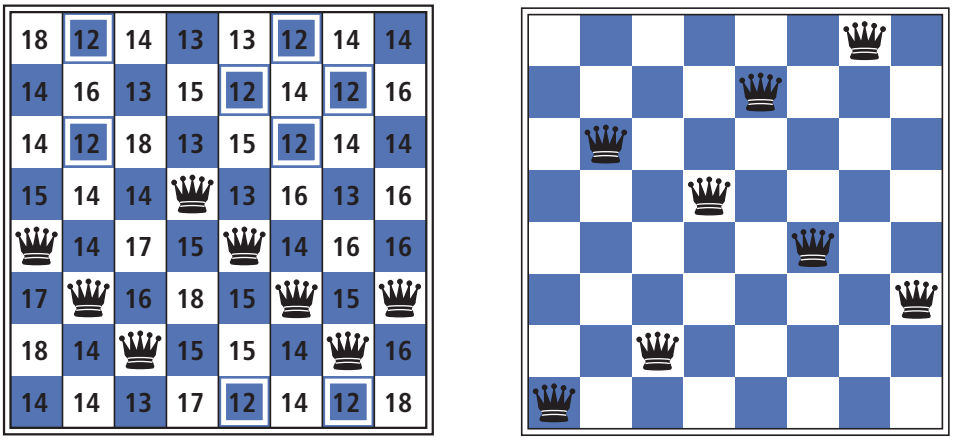


Abbildung 4.3: (a) Ein 8-Damen-Zustand mit heuristischer Kostenschätzung $h = 17$, die den Wert von h für jeden möglichen Nachfolger zeigt, den man erhält, wenn man eine Dame innerhalb ihrer Spalte verschiebt. Die besten Züge sind markiert. (b) Ein lokales Minimum im 8-Damen-Zustandsraum; der Wert des Zustandes ist $h = 1$, doch sind die Kosten bei jedem Nachfolger höher.

Bergsteigen wird manchmal auch als **„gierige" lokale Suche** (greedy local search) bezeichnet, weil dieser Algorithmus einen guten Nachbarzustand auswählt, ohne darüber

nachzudenken, wo es weitergehen soll. Obwohl Gier als eine der sieben Todsünden gilt, stellt sich heraus, dass „gierige“ Algorithmen häufig gute Leistungen aufweisen. Das Bergsteigen kommt oftmals schnell zu einer Lösung, weil es normalerweise relativ einfach ist, einen schlechten Zustand zu verbessern. Zum Beispiel sind vom Zustand in Abbildung 4.3(a) lediglich fünf Schritte erforderlich, um den Zustand in ▶ Abbildung 4.3(b) zu erreichen, der den Wert $h = 1$ hat und einer Lösung sehr nahe kommt. Leider bleibt das Bergsteigen aus den folgenden Gründen häufig stecken:

- **Lokale Maxima:** Ein lokales Maximum ist eine Spitze, die höher als jeder ihrer Nachbarzustände, aber niedriger als das globale Maximum ist. Bergsteigeralgorithmen, die in die Nähe eines lokalen Maximums gelangen, werden nach oben in Richtung Spitze gezogen, bleiben dann aber dort stecken und haben keine weiteren Möglichkeiten mehr. Abbildung 4.1 veranschaulicht das Problem schematisch. Konkreter gesagt, ist der Zustand in Abbildung 4.3(b) tatsächlich ein lokales Maximum (d.h. ein lokales Minimum für die Kosten h); jeder Zug einer einzelnen Dame führt zu einer ungünstigeren Situation.
- **Kammlinien:** ▶ Abbildung 4.4 zeigt eine Kammlinie. Kammlinien führen zu einer Folge lokaler Maxima, in denen es „gierigen“ Algorithmen sehr schwer fällt zu navigieren.
- **Plateaus:** Ein Plateau ist ein flacher Bereich der Zustandsraum-Landschaft. Es kann sich dabei um ein flaches lokales Maximum handeln, von dem aus es keinen Ausgang nach oben gibt, oder um einen **Bergrücken**, von dem aus weitere Fortschritte möglich sind (siehe Abbildung 4.1). Eine Bergsteigen-Suche verirrt sich möglicherweise auf dem Plateau.

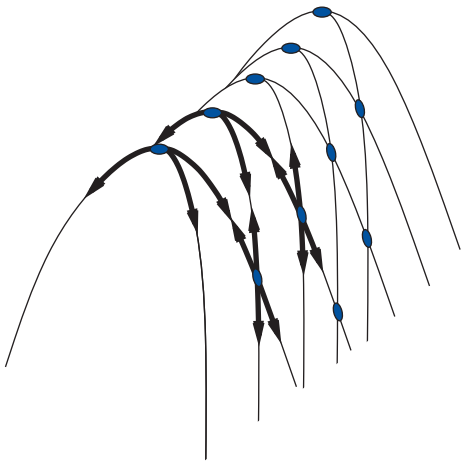


Abbildung 4.4: Hier erkennen Sie, warum Kammlinien beim Bergsteigen Schwierigkeiten verursachen. Das Zustandsraster (dunkle Kreise) wird von einer Kammlinie überlagert, die von links nach rechts verläuft und eine Folge lokaler Maxima erzeugt, die nicht direkt miteinander verbunden sind. Von jedem lokalen Maximum aus führen sämtliche möglichen Aktionen bergab.

In jedem Fall erreicht der Algorithmus einen Punkt, an dem kein Fortschritt mehr erzielt wird. Beginnend bei einem zufällig erzeugten 8-Damen-Zustand bleibt das Bergsteigen nach der Methode des steilsten Anstieges in 86% aller Fälle stecken und löst nur 14% der Probleminstanzen. Das Verfahren arbeitet schnell und benötigt nur

durchschnittlich vier Schritte, wenn es erfolgreich ist, und drei, wenn es stecken bleibt – nicht schlecht für einen Zustandsraum mit $8^8 \approx 17$ Millionen Zuständen.

Der in Abbildung 4.2 angegebene Algorithmus hält an, wenn er ein Plateau erreicht, wo der beste Nachfolger denselben Wert wie der aktuelle Zustand hat. Wäre es nicht sinnvoll weiterzugehen – um einen **Seitenweg** zu erlauben, in der Hoffnung, dass es sich bei dem Plateau eigentlich um einen Bergrücken handelt, wie in Abbildung 4.1 gezeigt? Die Antwort ist normalerweise „Ja“, doch ist Vorsicht geboten. Wenn wir immer Seitwärtsbewegungen erlauben, wo es keine Aufwärtsbewegungen mehr gibt, tritt eine Endlosschleife auf, wenn der Algorithmus ein flaches lokales Maximum erreicht, bei dem es sich nicht um einen Bergrücken handelt. Als allgemeine Lösung führt man eine Begrenzung der Anzahl aufeinanderfolgender Seitwärtsbewegungen ein. Beispielsweise könnten wir sagen, dass wir 100 aufeinanderfolgende Seitwärtsbewegungen im 8-Damen-Problem erlauben. Das erhöht den Prozentsatz der Probleminstanzen, die durch Bergsteigen gelöst werden, von 14% auf 94%. Der Erfolg hat jedoch auch seinen Preis: Der Algorithmus braucht durchschnittlich etwa 21 Schritte für jede erfolgreiche Instanz und 64 Schritte für jeden Versager.

Es wurden viele Varianten des Bergsteigens erfunden. Das **stochastische Bergsteigen** wählt zufällig aus den Aufwärtsbewegungen aus; die Wahrscheinlichkeit der Auswahl kann mit der Steilheit der Aufwärtsbewegung variieren. Diese Variante konvergiert normalerweise langsamer als die Methode der steilsten Steigung, findet aber in einigen Zustandslandschaften bessere Lösungen. Das **Bergsteigen mit erster Auswahl** implementiert ein stochastisches Bergsteigen, in dem zufällig Nachfolger erzeugt werden, bis einer erzeugt wurde, der besser als der aktuelle Zustand ist. Das ist eine gute Strategie, wenn ein Zustand mehrere (zum Beispiel Tausende) Nachfolger hat.

Die bisher beschriebenen Bergsteigeralgorithmen sind unvollständig – häufig finden sie kein Ziel, wenn eines existiert, weil sie an lokalen Maxima stecken bleiben können. Das **Bergsteigen mit zufälligem Neustart** übernimmt die alte Redensart: „Wenn Sie beim ersten Mal keinen Erfolg haben, probieren Sie es einfach noch einmal.“ Es führt eine Folge von Bergsteigen-Suchen von zufällig erzeugten Ausgangszuständen aus durch,[1] bis ein Ziel gefunden ist. Es ist vollständig mit einer Wahrscheinlichkeit nahe bei 1 – aus dem einfachen Grund, dass es irgendwann einen Zielzustand als den Ausgangszustand erzeugt. Wenn jede Bergsteigen-Suche eine Wahrscheinlichkeit p für den Erfolg aufweist, ist die erwartete Anzahl erforderlicher Neustarts gleich $1/p$. Für 8-Damen-Instanzen ohne erlaubte Züge zur Seite beträgt $p \approx 0{,}14$, sodass wir etwa sieben Iterationen brauchen, um ein Ziel zu finden (sechs Fehlschläge und einen Erfolg). Die erwartete Anzahl der Schritte sind die Kosten für eine erfolgreiche Iteration zuzüglich $(1-p)/p$-mal die Kosten der Fehlschläge oder etwa 22 Schritte. Wenn wir Seitwärtszüge erlauben, sind durchschnittlich $1/0{,}94 \approx 1{,}06$ Iterationen erforderlich und $(1 \times 21) + (0{,}06/0{,}94) \times 64 \approx 25$ Schritte. Für das 8-Damen-Problem ist dann das Bergsteigen mit zufälligem Neustart tatsächlich sehr effektiv. Selbst für drei Millionen Damen findet der Ansatz Lösungen in einer Zeit unter einer Minute.[2]

1 Das Erzeugen eines *zufälligen* Zustandes aus einem implizit spezifizierten Zustandsraum kann an sich ein schwieriges Problem sein.

2 Luby *et al.* (1993) beweisen, dass es in einigen Fällen am besten ist, zufällige Suchalgorithmen nach einer bestimmten feststehenden Zeit neu zu starten, und dass dies *wesentlich* effizienter sein kann, als jede Suche unbestimmt lange fortzusetzen. Ein Beispiel dafür ist, Seitenzüge zu verbieten oder ihre Anzahl zu begrenzen.

Der Erfolg des Bergsteigens hängt sehr von der Gestalt der Zustandsraum-Landschaft ab: Wenn es wenige lokale Maxima und Plateaus gibt, findet das Bergsteigen mit zufälligem Neustart sehr schnell eine gute Lösung. Andererseits sieht die Landschaft bei vielen realen Problemen wie eine Familie fast kahler Stachelschweine auf einem flachen Boden aus, wobei winzige Stachelschweine auf den Spitzen jedes Stachels eines anderen Stachelschweins liegen – *ad infinitum*. NP-harte Probleme habe normalerweise eine exponentielle Anzahl lokaler Maxima, an denen sie hängen bleiben können. Trotz dieser Tatsache kann häufig nach einer kleinen Anzahl von Neustarts ein ausreichend gutes lokales Maximum gefunden werden.

4.1.2 Simulated-Annealing – simuliertes Abkühlen

Ein Bergsteigeralgorithmus, der *niemals* „Abwärtsbewegungen" in Richtung von Zuständen mit geringerem Wert (oder höheren Kosten) macht, ist garantiert unvollständig, weil er an einem lokalen Maximum hängen bleiben kann. Im Gegensatz dazu ist ein rein zufälliges Weitergehen (Random Walk) – d.h. die Bewegung zu einem Nachfolger, der gleichverteilt zufällig aus der Menge der Nachfolger ausgewählt wird – vollständig, aber äußerst ineffizient. Aus diesem Grund scheint es sinnvoll, zu versuchen, das Bergsteigen mit einem zufälligen Weitergehen zu kombinieren, sodass man sowohl Effizienz als auch Vollständigkeit erhält. Das **Simulated Annealing** ist ein solcher Algorithmus. In der Metallurgie ist das **Glühen** (Annealing) der Prozess, Metalle oder Glas zu tempern oder zu härten, indem man sie auf eine hohe Temperatur aufheizt und dann allmählich abkühlt, sodass das Material in einen kristallinen Zustand mit geringer Energie übergehen kann. Um das Simulated Annealing zu erläutern, wollen wir unsere Perspektive vom Bergsteigen zum **Gradientenabstieg** (d.h. Kostenminimierung) wechseln und uns vorstellen, wir müssten einen Tischtennisball in die tiefste Spalte einer buckligen Oberfläche bringen. Wenn wir den Ball einfach rollen lassen, bleibt er an einem lokalen Minimum liegen. Wenn wir die Oberfläche schütteln, können wir den Ball aus dem lokalen Minimum herausspringen lassen. Der Trick dabei ist, gerade so viel zu schütteln, dass der Ball das lokale Minimum verlässt, aber nicht so stark, dass er sich ganz von dem globalen Minimum entfernt. Beim Simulated Annealing besteht die Lösung darin, zuerst stark zu schütteln (d.h. bei einer hohen Temperatur) und dann die Intensität des Schüttelns allmählich zu verringern (d.h. die Temperatur zu verringern).

Die innere Schleife des Algorithmus für das Simulated Annealing (▶ Abbildung 4.5) ist dem Bergsteigen recht ähnlich. Anstatt jedoch die *beste* Bewegung auszuwählen, greift der Algorithmus eine *zufällige* Bewegung heraus. Verbessert die Bewegung die Situation, wird sie immer akzeptiert. Andernfalls akzeptiert der Algorithmus die Bewegung mit einer Wahrscheinlichkeit kleiner 1. Die Wahrscheinlichkeit sinkt exponentiell mit der „Schlechtigkeit" der Bewegung – um den Betrag ΔE, um den die Auswertung verschlechtert wird. Die Wahrscheinlichkeit verringert sich ebenfalls, wenn die „Temperatur" T sinkt: „Schlechte" Bewegungen treten zu Beginn wahrscheinlicher auf, wenn T hoch ist, und werden immer unwahrscheinlicher, wenn T sinkt. Wenn der *Zeitplan* den Wert von T langsam genug sinken lässt, findet der Algorithmus ein globales Optimum mit einer Wahrscheinlichkeit nahe 1.

```
function SIMULATED-ANNEALING(problem, schedule) returns einen Lösungszustand
  inputs: problem, ein Problem
          schedule, eine Zuordnung von Zeit zu "Temperatur"
  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
    T ← schedule(t)
    if T = 0 then return current
    next ← ein zufällig ausgewählter Nachfolger von current
    ΔE ← next.VALUE - current.VALUE
    if ΔE > 0 then current ← next
    else current ← next nur mit Wahrscheinlichkeit e^(ΔE/T)
```

Abbildung 4.5: Der Simulated Annealing-Algorithmus ist eine Version des stochastischen Bergsteigens, wobei einige Abwärtsbewegungen erlaubt sind. Abwärtsbewegungen werden früh im Annealing-Zeitplan häufiger akzeptiert und dann mit fortschreitender Zeit weniger oft. Der Eingabewert *schedule* legt den Wert von T als Funktion der Zeit fest.

Das Simulated Annealing wurde in den frühen 80er Jahren ausgiebig verwendet, um VLSI-Layoutprobleme zu lösen. Es wurde häufig für die Herstellungsplanung und andere große Optimierungsaufgaben angewendet. In Übung 4.3 werden Sie seine Leistung mit der des Bergsteigens mit zufälligem Neustart für das 8-Damen-Problem vergleichen.

4.1.3 Lokale Strahlsuche (Local Beam Search)

Wird nur ein einziger Knoten im Speicher gehalten, mutet das wie eine Extremreaktion auf das Problem der Speicherbegrenzung an. Der **Local Beam Search**-Algorithmus (**lokale Strahlsuche**)[3] verwaltet k Zustände statt nur eines einzigen. Er beginnt mit k zufällig erzeugten Zuständen. Bei jedem Schritt werden alle Nachfolger aller k Zustände erzeugt. Ist einer davon ein Ziel, wird der Algorithmus beendet. Andernfalls wählt er die besten k Nachfolger aus der vollständigen Liste aus und wiederholt den Ablauf.

Tipp

Auf den ersten Blick erscheint eine lokale Strahlsuche mit k Zuständen nichts anderes zu sein als die parallele statt sequenzielle Ausführung von k zufälligen Neustarts. Tatsächlich sind die beiden Algorithmen jedoch sehr unterschiedlich. Bei einer Suche mit zufälligem Neustart wird jeder Suchprozess unabhängig von den anderen ausgeführt. *Bei einer lokalen Strahlsuche werden wichtige Informationen zwischen den parallelen Such-Threads übergeben.* Praktisch teilen die Zustände, die die besten Nachfolger erzeugen, den anderen mit: „Kommt doch zu mir, hier ist das Gras grüner!“ Der Algorithmus bricht fruchtlose Suchen schnell ab und konzentriert seine Ressourcen dort, wo der größte Fortschritt erzielt wird.

In ihrer einfachsten Form kann die lokale Strahlsuche unter einem Mangel an Vielfalt zwischen den k Zuständen leiden – sie können sich schnell in einem kleinen Bereich des Zustandsraumes konzentrieren und machen die Suche zu wenig mehr als einer teureren Version des Bergsteigens. Mit einer Variante, die analog zum stochastischen Bergsteigen auch als **stochastische Strahlsuche** bezeichnet wird, lässt sich dieses Problem umgehen. Anstatt die besten k aus dem Pool der möglichen Nachfolger auszuwählen, wählt die stochastische Strahlsuche k Nachfolger zufällig aus – wobei die Wahrscheinlichkeit, einen bestimmten Nachfolger auszuwählen, eine wachsende Funktion ihres Wertes ist. Die stochastische Strahlsuche erinnert etwas an den Prozess der natürlichen

3 Local Beam Search (lokale Strahlsuche) ist eine Adaption der **Beam Search**, die einen pfadbasierten Algorithmus darstellt.

Auswahl, wobei die „Nachfolger“ (Nachkommen) eines „Zustandes“ (Organismus) die nächste Generation gemäß ihrem „Wert“ (Fitness) bevölkern.

4.1.4 Genetische Algorithmen

Ein **genetischer Algorithmus** (oder **GA**) ist eine Variante der stochastischen Strahlsuche, bei der Nachfolgerzustände erzeugt werden, indem *zwei* übergeordnete (Eltern-)Zustände kombiniert werden, anstatt einen einzelnen Zustand zu verändern. Die Analogie zur natürlichen Auswahl ist dieselbe wie bei der stochastischen Strahlsuche, außer dass wir es jetzt mit sexueller statt mit asexueller Reproduktion zu tun haben.

Wie die Strahlsuchen beginnen auch genetische Algorithmen mit einer Menge von k zufällig erzeugten Zuständen, die als **Population** bezeichnet werden. Jeder Zustand oder jedes **Individuum** wird durch eine Zeichenfolge aus einem endlichen Alphabet dargestellt – am gebräuchlichsten sind dabei Zeichenfolgen, die aus 0 und 1 bestehen. Zum Beispiel muss ein 8-Damen-Zustand die Positionen von acht Damen darstellen, die jeweils in einer Spalte mit acht Quadraten stehen, und erfordert deshalb $8 \times \log_2 8 = 24$ Bit. Alternativ könnte der Zustand auch mit acht Ziffern dargestellt werden, die im Bereich zwischen 1 und 8 liegen. (Später werden wir sehen, dass sich die beiden Kodierungen unterschiedlich verhalten.) ▶ Abbildung 4.6(a) zeigt eine Population von vier Zeichenfolgen mit je acht Ziffern, die 8-Damen-Zustände darstellen.

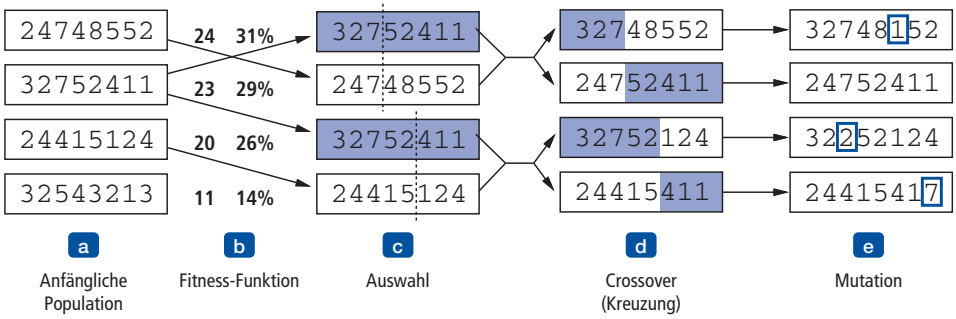


Abbildung 4.6: Der genetische Algorithmus, hier veranschaulicht an Zeichenfolgen aus Ziffern, die 8-Damen-Zustände darstellen. Die Anfangspopulation in (a) wird nach der Fitness-Funktion in (b) geordnet, woraus sich Fortpflanzungspaare in (c) ergeben. Sie produzieren eine Nachkommenschaft in (d), die in (e) einer Mutation unterzogen wird.

Die Erstellung der nächsten Generation von Zuständen wird in ▶ Abbildung 4.6(b) bis (e) gezeigt. In (b) wird jeder Zustand durch die Zielfunktion oder (in der Terminologie der genetischen Algorithmen) die **Fitness-Funktion** bewertet. Eine Fitness-Funktion sollte für bessere Zustände höhere Werte zurückgeben; für das 8-Damen-Problem verwenden wir deshalb die Anzahl der *nicht angreifenden* Damenpaare, die einen Wert von 28 für eine Lösung aufweist. Die Werte der vier Zustände sind 24, 23, 20 und 11. In dieser besonderen Variante des genetischen Algorithmus ist die Wahrscheinlichkeit, zur Reproduktion ausgewählt zu werden, direkt proportional zur Fitness-Punktzahl. Die Prozentzahlen sind neben den eigentlichen Punktzahlen angegeben.

In (c) werden zwei Paare zufällig entsprechend den Wahrscheinlichkeiten in (b) zur Reproduktion ausgewählt. Beachten Sie, dass ein Individuum zweimal ausgewählt

wird und eines überhaupt nicht.[4] Für jedes zu verbindende Paar wird ein Kreuzungspunkt zufällig aus den Positionen in der Zeichenfolge ausgewählt (**Crossover**). In Abbildung 4.6 befinden sich die Kreuzungspunkte nach der dritten Ziffer im ersten Paar und nach der fünften Ziffer im zweiten Paar[5].

In (d) werden die eigentlichen Nachkommen erzeugt, indem die Eltern-Zeichenfolgen am Kreuzungspunkt gekreuzt werden. Zum Beispiel erhält das erste Kind des ersten Paares die ersten drei Ziffern vom ersten Elternteil und die restlichen Ziffern vom zweiten Elternteil, während das zweite Kind die ersten drei Ziffern vom zweiten Elternteil und die übrigen Ziffern vom ersten Elternteil erhält. ▶ Abbildung 4.7 zeigt die 8-Damen-Zustände, die an diesem Reproduktionsschritt beteiligt sind. Wie aus dem Beispiel hervorgeht, kann die Kreuzung bei recht verschiedenen Elternteilen einen Zustand hervorbringen, der sich erheblich von beiden Elternzuständen unterscheidet. Häufig ist die Population früh im Prozess sehr vielfältig, sodass die Kreuzung (wie das Simulated Annealing) oft früh im Suchprozess große Schritte im Zustandsraum unternimmt und später kleinere Schritte, wenn sich die meisten Individuen schon recht ähnlich sind.

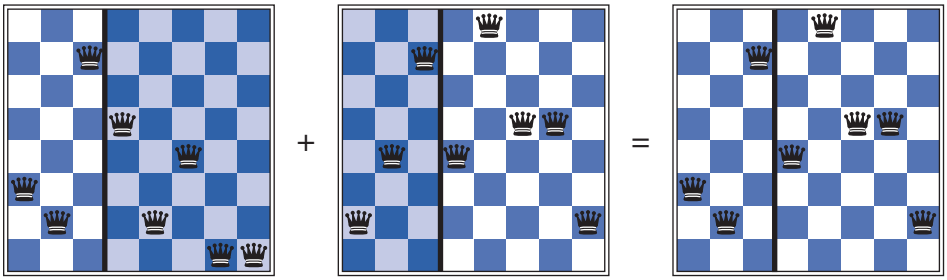


Abbildung 4.7: Die 8-Damen-Zustände, die den beiden ersten Eltern in ▶ Abbildung 4.6(c) entsprechen sowie dem ersten Nachkommen in Abbildung 4.6(d). Die grau schattierten Spalten gehen bei der Kreuzung verloren. Die nicht schattiert dargestellten Spalten werden beibehalten.

In (e) schließlich unterliegt jede Position einer zufälligen **Mutation** mit einer kleinen unabhängigen Wahrscheinlichkeit. Eine Ziffer wurde im ersten, dritten und vierten Nachkommen mutiert. Im 8-Damen-Problem entspricht dies der zufälligen Auswahl einer Dame, die dann auf ein zufälliges Quadrat in ihrer Spalte verschoben wird. ▶ Abbildung 4.8 beschreibt einen Algorithmus, der all diese Schritte implementiert.

Wie die stochastische Strahlsuche kombinieren genetische Algorithmen eine Aufwärtstendenz mit einer zufälligen Erkundung und dem Austausch von Informationen zwischen parallelen Such-Threads. Der vielleicht wichtigste Vorteil von genetischen Algorithmen ergibt sich aus der Kreuzungsoperation. Es lässt sich auch mathematisch zeigen, dass Crossover keinen Vorteil bringt, wenn die Positionen des genetischen Codes anfangs in einer zufälligen Reihenfolge permutiert werden. Intuitiv betrachtet, stammt der Vorteil aus der Möglichkeit des Crossover, große Buchstabenblöcke zu kombinieren, die sich unabhängig voneinander entwickelt haben, um sinnvolle Funktionen auszuführen und damit den Grad der Granularität der Suche zu erhöhen. Wer-

4 Es gibt viele Varianten dieser Auswahlregel. Es lässt sich zeigen, dass die Methode der **Auslese**, bei der alle Individuen unter einem vorgegebenen Schwellenwert verworfen werden, schneller als die Zufallsversion konvergiert (Baum *et al.*, 1995).

5 Hier spielt die Kodierung eine Rolle. Wenn statt acht Ziffern eine 24-Bit-Kodierung verwendet wird, hat der Kreuzungspunkt eine Zweidrittelchance, in der Mitte einer Ziffer zu liegen, was zu einer im Grunde willkürlichen Mutation dieser Ziffer führt.

den zum Beispiel die ersten drei Damen auf den Positionen 2, 4 und 6 platziert (wo sie sich gegenseitig nicht angreifen), könnte dies einen sinnvollen Block bilden, der sich mit anderen Blöcken kombinieren lässt, um eine Lösung zu konstruieren.

```
function GENETIC-ALGORITHM(population, FITNESS-FN) returns ein Individuum
  inputs: population, eine Menge von Individuen
          FITNESS-FN, eine Funktion, die die Fitness eines Individuums misst

  repeat
    new_population ← leere Menge
    for i = 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (kleine Zufallswahrscheinlichkeit) then child ← MUTATE(child)
        child zu new_population hinzufügen
    population ← new_population
  until ein Individuum ist fit genug oder es ist genügend Zeit verstrichen
  return das beste Individuum in population, gemäß FITNESS-FN
```

```
function REPRODUCE(x, y) returns ein Individuum
  inputs: x, y, Elternindividiuen

  n ← LENGTH(x); c ← Zufallszahl von 1 bis n
  return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

Abbildung 4.8: Ein genetischer Algorithmus. Dieser Algorithmus ist der gleiche wie der in Abbildung 4.6 gezeigte, mit einer Abweichung: In dieser gebräuchlicheren Version erzeugt jede Kombination von zwei Elternteilen nur einen Nachkommen und nicht zwei.

Wie dies funktioniert, erklärt die Theorie der genetischen Algorithmen anhand eines **Schemas**, das eine Teilzeichenfolge darstellt, in der einige der Positionen unspezifiziert bleiben können. Beispielsweise beschreibt das Schema 246***** alle 8-Damen-Zustände, bei denen sich die ersten drei Damen an den Positionen 2, 4 und 6 befinden. Zeichenfolgen, die mit diesem Schema übereinstimmen (wie z.B. 24613578), werden als **Instanzen** des Schemas bezeichnet. Es lässt sich zeigen, dass die Anzahl der Instanzen des Schemas innerhalb der Population mit der Zeit zunimmt, wenn die durchschnittliche Fitness der Instanzen eines Schemas über dem Mittelwert liegt. Es liegt auf der Hand, dass sich dieser Effekt kaum bemerkbar macht, wenn benachbarte Bits nichts miteinander zu tun haben, da es dann wenige zusammenhängende Blöcke gibt, die einen eindeutigen Nutzen bieten. Genetische Algorithmen funktionieren am besten, wenn die Schemas sinnvollen Komponenten einer Lösung entsprechen. Ist die Zeichenfolge beispielsweise die Darstellung einer Antenne, könnten die Schemas Komponenten der Antenne darstellen, wie etwa Reflektoren und Deflektoren. Eine gute Komponente ist wahrscheinlich in einer Vielfalt unterschiedlicher Designs gut geeignet. Das legt nahe, dass die erfolgreiche Verwendung genetischer Algorithmen eine sorgfältige Auswahl der Darstellung erforderlich macht.

In der Praxis hatten genetische Algorithmen weitreichenden Einfluss auf Optimierungsprobleme, wie beispielsweise das Layout von Schaltkreisen oder die Produktionsablaufplanung. Momentan ist nicht klar, ob die Attraktivität genetischer Algorithmen auf ihre Leistung zurückzuführen ist oder auf die ästhetisch ansprechenden Ursprünge in der Evolutionstheorie. Es bleibt noch viel zu tun, um die Bedingungen herauszuarbeiten, unter denen genetische Algorithmen die beste Leistung erbringen.

4.2 Lokale Suche in stetigen Räumen

In *Kapitel 2* haben wir die Unterscheidung zwischen diskreten und stetigen Umgebungen erklärt und darauf hingewiesen, dass die meisten Umgebungen der realen Welt stetig sind. Keiner der bisher beschriebenen Algorithmen (außer Bergsteigen mit erster Auswahl und Simulated Annealing) kann stetige Zustände und Aktionsräume verarbeiten, da sie unendliche Verzweigungsfaktoren haben. Dieser Abschnitt bietet eine sehr kurze Einführung in einige lokale Suchtechniken, um optimale Lösungen in stetigen Räumen zu finden. Das Literaturangebot zu diesem Thema ist riesig; viele der grundlegenden Techniken stammen aus dem 17. Jahrhundert, nach der Entwicklung der Analysis durch Newton und Leibniz.[6] Wir werden an mehreren Stellen dieses Buches auf Verwendungszwecke für diese Techniken hinweisen, unter anderem in den Kapiteln zum Lernen, zur Vision und zur Robotik.

Evolution und Suche

Die Theorie der **Evolution** wurde von Charles Darwin in *On the Origin of Species by Means of Natural Selection* (1859) und unabhängig von Alfred Russel Wallace (1858) entwickelt. Das zentrale Konzept ist einfach: Bei der Reproduktion treten Variationen (als Mutationen bezeichnet) auf und werden in nachfolgenden Generationen annähernd proportional zur ihrer Auswirkung auf die reproduktive Fitness beibehalten.

Darwin entwickelte seine Theorie, ohne dass er wusste, wie die Merkmale von Organismen vererbt und modifiziert werden können. Die Wahrscheinlichkeitsgesetze, denen diese Prozesse unterliegen, wurden zuerst von Gregor Mendel (1866) identifiziert, einem Mönch, der mit Erbsen experimentierte. Viel später identifizierten Watson und Crick (1953) die Struktur des DNA-Moleküls und sein Alphabet, AGTC (Adenin, Guanin, Thymin, Cytosin). In Standardmodellen treten Variationen sowohl durch Punktmutationen in der Buchstabenfolge als auch durch „Kreuzung" auf (wobei die DNA eines Nachkommen erzeugt wird, indem lange Abschnitte der DNA beider Elternteile kombiniert werden).

Die Analogien zu lokalen Suchalgorithmen wurden bereits beschrieben; der hauptsächliche Unterschied zwischen einer stochastischen Strahlsuche und der Evolution ist die Verwendung der sexuellen Reproduktion, wobei *Nachfolger* aus mehreren Organismen und nicht nur aus einem einzigen erzeugt werden. Die eigentlichen Mechanismen der Evolution sind jedoch weitaus umfangreicher, als die meisten genetischen Algorithmen es darstellen können. Beispielsweise können Mutationen auch Umkehrungen, Verdoppelungen oder das Verschieben großer DNA-Abschnitte beinhalten; manche Viren borgen sich DNA von einem Organismus und fügen sie in einen anderen ein; und es gibt transponierende Viren, die nichts weiter tun, als sich selbst viele Tausend Mal innerhalb des Genoms zu kopieren. Es gibt sogar Gene, die Zellen aus potenziellen Partnern vergiften, die das Gen nicht tragen, und auf diese Weise die Wahrscheinlichkeit ihrer Replikation erhöhen. Noch viel wichtiger ist *die Tatsache, dass die Gene selbst die Mechanismen kodieren*, wie das Genom reproduziert und in einen Organismus übersetzt wird. Bei genetischen Algorithmen sind diese Mechanismen in einem separaten Programm realisiert, das nicht innerhalb der zu manipulierenden Zeichenfolgen dargestellt wird.

6 Grundlegende Kenntnisse im Bereich der multivariaten Analysis sowie der Vektorarithmetik ist beim Lesen dieses Abschnittes sehr hilfreich.

Die Darwinsche Evolution mag ineffizient erscheinen, da sie blindlings etwa 10^{45} Organismen erzeugt hat, ohne ihre Suchheuristiken auch nur um einen Deut zu verbessern. Allerdings hat bereits 50 Jahre vor Darwin der bekannte französische Naturforscher Jean Lamarck (1809) eine Evolutionstheorie vorgeschlagen, wobei *Merkmale, die während der Lebenszeit eines Organismus durch Anpassung angeeignet wurden, an die Nachkommen weitergegeben* werden. Ein solcher Prozess wäre effektiv, scheint aber in der Natur nicht aufzutreten. Viel später schlug James Baldwin (1896) eine ähnlich erscheinende Theorie vor: Das Verhalten, das während der Lebensdauer eines Organismus erlernt wird, könnte die Evolutionsgeschwindigkeit beschleunigen. Anders als die Theorie von Lamarck ist die Theorie von Baldwin vollständig konsistent mit der Darwinschen Theorie, weil sie auf dem Auswahldruck basiert, der auf Individuen wirkt, die lokale Optima in der Menge möglicher – durch ihre genetische Konstitution erlaubter – Verhaltensweisen gefunden haben. Moderne Computersimulationen bestätigen, dass es den „Baldwin-Effekt" gibt, nachdem die „normale" Evolution Organismen erzeugt hat, deren internes Leistungsmaß mit der tatsächlichen Fitness korreliert.

Beginnen wir mit einem Beispiel. Angenommen, wir wollen drei neue Flughäfen irgendwo in Rumänien bauen, sodass die Summe der Quadratdistanzen von jeder Stadt auf der Landkarte (Abbildung 4.2) zum nächsten Flughafen so gering wie möglich ist. Der Zustandsraum wird dann durch die Koordinaten der Flughäfen definiert: (x_1, y_1), (x_2, y_2) und (x_3, y_3). Dabei handelt es sich um einen *sechsdimensionalen* Raum; wir sagen auch, dass die Zustände durch sechs **Variablen** definiert sind. (Im Allgemeinen sind Zustände durch einen n-dimensionalen Vektor von Variablen definiert, $\mathbf{x}$.) Bewegt man sich in diesem Raum, entspricht das dem Verschieben eines oder mehrerer der Flughäfen auf der Landkarte. Die Zielfunktion $f(x_1, y_1, x_2, y_2, x_3, y_3)$ ist für jeden einzelnen Zustand relativ einfach zu berechnen, nachdem wir die nächstliegenden Städte berechnet haben. Mit C_i bezeichnen wir die Menge der Städte, deren nächster Flughafen (im aktuellen Zustand) der Flughafen i ist. Dann haben wir *in der Nachbarschaft des aktuellen Zustandes*, wo die C_i konstant bleiben:

$$f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^{3} \sum_{c \in C_i} (x_1 - x_c)^2 + (y_i - y_c)^2. \tag{4.1}$$

Dieser Ausdruck ist *lokal* korrekt, jedoch nicht global, weil die Mengen C_i (unstetige) Funktionen des Zustandes sind.

Stetigkeitsprobleme lassen sich zum Beispiel vermeiden, indem einfach die Nachbarschaft jedes Zustandes **diskretisiert** wird. Beispielsweise können wir jeweils nur einen Flughafen auf einmal entweder in x- oder y-Richtung um einen festen Betrag $\pm\delta$ verschieben. Bei sechs Variablen erhalten wir damit zwölf mögliche Nachfolger für jeden Zustand. Dann können wir einen beliebigen der zuvor beschriebenen lokalen Suchalgorithmen anwenden. Man könnte auch stochastisches Bergsteigen oder Simulated Annealing direkt anwenden, ohne den Raum zu diskretisieren. Diese Algorithmen wählen Nachfolger zufällig aus, was sich bewerkstelligen lässt, indem zufällige Vektoren der Länge δ erzeugt werden.

Viele Methoden versuchen, mithilfe des **Gradienten** der Landschaft ein Maximum zu finden. Der Gradient der Zielfunktion ist ein Vektor ∇f, der die Größe und die Richtung der steilsten Steigung angibt. Für unser Problem erhalten wir:

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right).$$

In einigen Fällen finden wir ein Maximum, indem wir die Gleichung $\nabla f = 0$ lösen. (Das ließe sich beispielsweise realisieren, wenn wir lediglich einen Flughafen platzieren; die Lösung ist der arithmetische Mittelwert der Koordinaten aller Städte.) Häufig kann diese Gleichung jedoch nicht in geschlossener Form gelöst werden. Zum Beispiel ist der Ausdruck für den Gradienten bei drei Flughäfen davon abhängig, welche Städte im aktuellen Zustand am nächsten zu jedem Flughafen liegen. Das bedeutet, wir können den Gradienten *lokal*, aber nicht *global* berechnen; zum Beispiel:

$$\frac{\partial f}{\partial x_1} = 2 \sum_{c \in C_1} (x_i - x_c). \tag{4.2}$$

Mit einem lokal korrekten Ausdruck für den Gradient können wir Bergsteigen nach der Methode des steilsten Anstieges ausführen, indem wir den aktuellen Zustand entsprechend der folgenden Formel aktualisieren:

$$\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x}).$$

Dabei ist α eine kleine Konstante, die oftmals als **Schrittweite** bezeichnet wird. In anderen Fällen ist die Zielfunktion möglicherweise überhaupt nicht in einer differenzierbaren Form verfügbar – beispielsweise kann der Wert einer bestimmten Menge von Flughafenpositionen durch Einsatz eines großen Wirtschafts-Simulationspaketes ermittelt werden. In diesen Fällen lässt sich ein sogenannter **empirischer Gradient** ermitteln, indem die Reaktionen auf kleine Inkrement- und Dekrementschritte in jeder Koordinate ausgewertet werden. Die Suche mit empirischem Gradienten ist dieselbe wie beim Bergsteigen mit steilstem Anstieg in einer diskretisierten Version des Zustandsraumes.

Hinter der Aussage „α ist eine kleine Konstante“ verbirgt sich eine riesige Vielfalt an Methoden zur Anpassung von α. Das grundlegende Problem dabei ist, dass bei einem zu kleinen α zu viele Schritte erforderlich sind; ist α zu groß, könnte die Suche über das Maximum hinausschießen. Die Technik der **Zeilensuche** versucht, dieses Dilemma zu umgehen, indem sie die aktuelle Gradientenrichtung erweitert – normalerweise durch wiederholte Verdopplung von α –, bis f wieder sinkt. Der Punkt, an dem dies auftritt, wird zum neuen aktuellen Zustand. Es gibt mehrere Denkschulen, wie die neue Richtung an dieser Stelle ausgewählt werden soll.

Für viele Probleme ist der effektivste Algorithmus die althergebrachte Methode von **Newton-Raphson** (Newton, 1671; Raphson, 1690). Dies ist eine allgemeine Technik zur Ermittlung der Nullstellen von Funktionen – d.h. die Lösung von Gleichungen der Form $g(x) = 0$. Man berechnet dazu eine neue Abschätzung der Nullstelle x gemäß der Formel von Newton:

$$x \leftarrow x - g(x)/g'(x).$$

Um ein Maximum oder ein Minimum von f zu finden, müssen wir ein $\mathbf{x}$ finden, sodass der *Gradient* (d.h. $\nabla f(\mathbf{x}) = \mathbf{0}$) gleich null ist. Somit wird $g(x)$ in der Formel von Newton zu $\nabla f(\mathbf{x})$ und die aktualisierte Gleichung kann in Matrix-Vektor-Form wie folgt geschrieben werden:

$$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x}) \nabla f(\mathbf{x}).$$

Dabei ist $\mathbf{H}_f(\mathbf{x})$ die **Hesse-Matrix** der zweiten Ableitungen, deren Elemente H_{ij} durch $\partial^2 f / \partial x_i \partial x_j$ gegeben sind. Für unser Flughafenbeispiel ergibt sich aus Gleichung (4.2), dass $\mathbf{H}_f(\mathbf{x})$ besonders einfach ist: Die nicht auf der Diagonalen liegenden Elemente

sind 0 und die Diagonalelemente für Flughafen i stellen einfach die doppelte Anzahl der Städte in C_i dar. Die Berechnung eines Moments zeigt, dass ein Schritt der Aktualisierung den Flughafen i direkt zum Schwerpunkt von C_i verschiebt, d.h. zum Minimum des lokalen Ausdruckes für f aus Gleichung (4.1).[7] Da die Hesse-Matrix n^2 Einträge enthält, wird es bei mehrdimensionalen Problemen zu teuer, sie zu berechnen und zu invertieren, und es wurden zahlreiche Näherungsversionen der Newton-Raphson-Methode entwickelt.

Lokale Suchmethoden leiden unter lokalen Maxima, Kämmen und Plateaus in stetigen Zustandsräumen genauso wie in diskreten Räumen. Zufälliges Neustarten und Simulated Annealing können verwendet werden und sind häufig recht hilfreich. Mehrdimensionale stetige Räume erweisen sich jedoch als große Orte, in denen man sich leicht verirren kann.

Ein letztes Thema, von dem man zumindest einmal gehört haben sollte, ist die **beschränkte Optimierung**. Ein Optimierungsproblem ist beschränkt, wenn die Lösungen bestimmte strenge Beschränkungen im Hinblick auf die Werte der Variablen erfüllen müssen. In unserem Problem der Flughafenstandorte könnten wir die möglichen Positionen darauf beschränken, dass sie sich innerhalb von Rumänien und auf dem Festland (und nicht in der Mitte irgendwelcher Seen) befinden müssen. Die Schwierigkeit bei beschränkten Optimierungsproblemen ist von der Art der Beschränkungen und der Zielfunktion abhängig. Die bekannteste Kategorie bilden die Probleme der **linearen Programmierung**, wobei die Beschränkungen lineare Ungleichheiten sein müssen, die eine **konvexe Menge**[8] bilden, und die Zielfunktion ebenfalls linear ist. Probleme der linearen Programmierung können in einer Zeit gelöst werden, die polynomiell zur Anzahl der Variablen ist.

Lineare Programmierung gehört wahrscheinlich zu der am intensivsten untersuchten und vielseitigsten Klasse von Optimierungsproblemen. Sie stellt einen Spezialfall des allgemeinen Problems der **konvexen Optimierung** dar, bei der der eingeschränkte Bereich ein beliebig konvexer Bereich und die Zielfunktion jede beliebige Funktion sein kann, die innerhalb des eingeschränkten Bereiches konvex ist. Unter bestimmten Bedingungen sind konvexe Optimierungsprobleme auch polynomiell lösbar und lassen sich in der Praxis durchaus mit Tausenden von Variablen durchführen. Verschiedene wichtige Probleme im maschinellen Lernen und in der Steuerungstheorie können als Probleme der konvexen Optimierung formuliert werden (siehe *Kapitel 20*).

4.3 Suchen mit nichtdeterministischen Aktionen

In *Kapitel 3* nahmen wir an, dass die Umgebung vollständig beobachtbar und deterministisch ist und die Agenten wissen, welche Wirkungen die einzelnen Aktionen auslösen. Der Agent kann demzufolge genau berechnen, welcher Zustand aus einer beliebigen

7 Im Allgemeinen lässt sich das Newton-Raphson-Verfahren so betrachten, dass es eine quadratische Oberfläche an f bei x anpasst und sie dann direkt zum Minimum dieser Oberfläche verschiebt – die auch das Minimum von f ist, wenn f quadratisch ist.

8 Eine Menge von Punkten S ist konvex, wenn die Gerade, die zwei beliebige Punkte in S verbindet, auch in S beschränkt ist. Eine **konvexe Funktion** ist eine Funktion, für die der Raum „über“ ihr eine konvexe Menge bildet; per Definition besitzen konvexe Funktionen keine lokalen (im Unterschied zu globalen) Minima.

Sequenz von Aktionen resultiert, und weiß immer, in welchem Zustand er sich befindet. Seine Wahrnehmungen liefern keine neuen Informationen nach jeder Aktion, obwohl sie natürlich dem Agenten den Ausgangszustand mitteilen.

Wenn die Umgebung teilweise beobachtbar und/oder nichtdeterministisch ist, werden Wahrnehmungen nützlich. In einer teilweise beobachtbaren Umgebung hilft jede Wahrnehmung, die Menge der möglichen Zustände, die der Agent einnehmen kann, einzuengen und es so dem Agenten zu erleichtern, seine Ziele zu erreichen. Ist die Umgebung nichtdeterministisch, kann der Agent anhand der Wahrnehmungen feststellen, welches der möglichen Ergebnisse seiner Aktionen tatsächlich aufgetreten ist. In beiden Fällen lassen sich die zukünftigen Wahrnehmungen nicht im Voraus bestimmen und die zukünftigen Aktionen des Agenten hängen von diesen zukünftigen Wahrnehmungen ab.

Die Lösung eines Problems ist also keine Sequenz, sondern ein **Kontingenzplan** (auch als **Strategie** bezeichnet), der spezifiziert, was in Abhängigkeit von den empfangenen Wahrnehmungen zu tun ist. In diesem Abschnitt geht es zunächst um Nichtdeterminismus und *Abschnitt 4.4* setzt sich dann mit teilweiser Beobachtbarkeit auseinander.

4.3.1 Die erratische Staubsaugerwelt

Als Beispiel verwenden wir die Staubsaugerwelt, die *Kapitel 2* eingeführt hat und die in *Abschnitt 3.2.1* als Suchproblem definiert wurde. Der Zustandsraum umfasst bekanntlich die acht Zustände, die in ▶ Abbildung 4.9 dargestellt sind. Es gibt drei Aktionen – *Links*, *Rechts* und *Saugen* – und das Ziel besteht darin, sämtlichen Schmutz zu beseitigen (Zustände 7 und 8). Wenn die Umgebung beobachtbar, deterministisch und vollständig bekannt ist, lässt sich das Problem trivial mit jedem der in *Kapitel 3* beschriebenen Algorithmen lösen. Als Lösung ergibt sich eine Aktionsfolge. Ist zum Beispiel der Ausgangszustand 1, erreicht die Aktionsfolge [*Saugen*, *Rechts*, *Links*] einen Zielzustand 8.

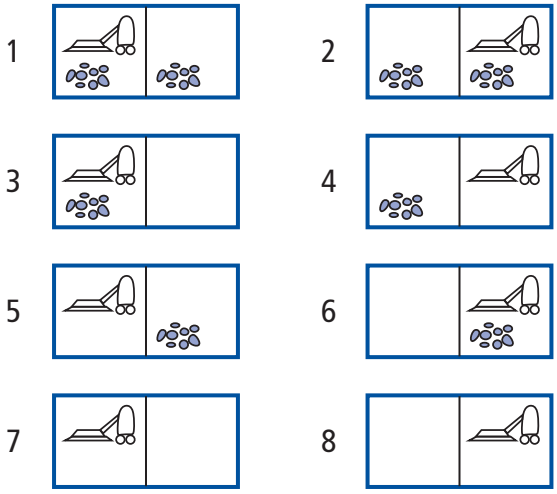


Abbildung 4.9: Die acht möglichen Zustände der Staubsaugerwelt. Die Zustände 7 und 8 sind Zielzustände.

Nehmen wir nun an, dass wir Nichtdeterminismus in Form eines leistungsfähigen, jedoch erratischen Staubsaugers einführen. In der **erratischen Staubsaugerwelt** funktioniert die Aktion *Saugen* wie folgt:

- Wird die Aktion auf ein schmutziges Quadrat angewendet, säubert sie das Quadrat und nimmt manchmal auch Schmutz in einem benachbarten Quadrat auf.
- Wird die Aktion auf ein sauberes Quadrat angewendet, lässt die Aktion gelegentlich Schmutz auf den Teppich fallen.[9]

Um eine genaue Formulierung dieses Problems angeben zu können, müssen wir das Konzept eines **Übergangsmodells** aus *Kapitel 3* verallgemeinern. Anstatt das Übergangsmodell durch eine RESULT-Funktion zu definieren, die einen einzelnen Status zurückgibt, verwenden wir eine RESULTS-Funktion, die eine Menge möglicher Ergebniszustände liefert. Zum Beispiel führt in der erratischen Staubsaugerwelt die Aktion *Saugen* im Zustand 1 zu einem Zustand in der Menge {5, 7} – der Schmutz im rechten Quadrat kann abgesaugt werden oder auch nicht.

Außerdem müssen wir das Konzept einer **Lösung** für das Problem verallgemeinern. Wenn wir zum Beispiel in Zustand 1 beginnen, gibt es keine einzelne *Folge* von Aktionen, die das Problem löst. Stattdessen brauchen wir einen Kontingenzplan wie den folgenden:

$$[\mathit{Saugen}, \textbf{if } \mathit{State} = 5 \textbf{ then } [\mathit{Rechts}, \mathit{Saugen}] \textbf{ else } [\,]]. \tag{4.3}$$

Somit können Lösungen für nichtdeterministische Probleme verschachtelte **if-then-else**-Anweisungen enthalten; d.h., es handelt sich um Bäume statt um Sequenzen. Das erlaubt die Auswahl von Aktionen basierend auf Kontingenzen, die während der Ausführung entstehen. Viele Probleme in der realen, physischen Welt sind Kontingenzprobleme, da eine genaue Vorhersage unmöglich ist. Deshalb halten viele Menschen ihre Augen offen, wenn sie durch die Gegend laufen oder fahren.

4.3.2 AND-OR-Suchbäume

Als Nächstes stellt sich die Frage, wie Kontingenzlösungen für nichtdeterministische Probleme gefunden werden können. Wie in *Kapitel 3* konstruieren wir zunächst Suchbäume, doch haben die Bäume hier einen anderen Charakter.

In einer deterministischen Umgebung wird die einzige Verzweigung durch die eigenen Entscheidungen des Agenten in jedem Zustand eingeführt. Diese Knoten bezeichnen wir als **OR-Knoten**. In der Staubsaugerwelt beispielsweise wählt der Agent an einem OR-Knoten *Links*, *Rechts* oder *Saugen*. In einer nichtdeterministischen Umgebung werden Verzweigungen auch durch die Auswahl des Ergebnisses für jede Aktion durch die *Umgebung* eingeführt. Diese Knoten bezeichnen wir als **AND-Knoten**. Zum Beispiel führt die *Saugen*-Aktion in Zustand 1 zu einem Zustand in der Menge {5, 7}, sodass der Agent einen Plan für Zustand 5 *und* für Zustand 7 finden müsste. Diese beiden Arten von Knoten wechseln sich ab, was zu einem **AND-OR-Baum** wie in ▶ Abbildung 4.10 dargestellt führt.

9 Wir gehen davon aus, dass die meisten Leser schon mit ähnlichen Problemen zu tun hatten und sich in die Lage unseres Agenten versetzen können. Gleichzeitig entschuldigen wir uns bei den Besitzern moderner, effizienter Haushaltsgeräte, die von diesem pädagogischen Gerät nicht profitieren können.

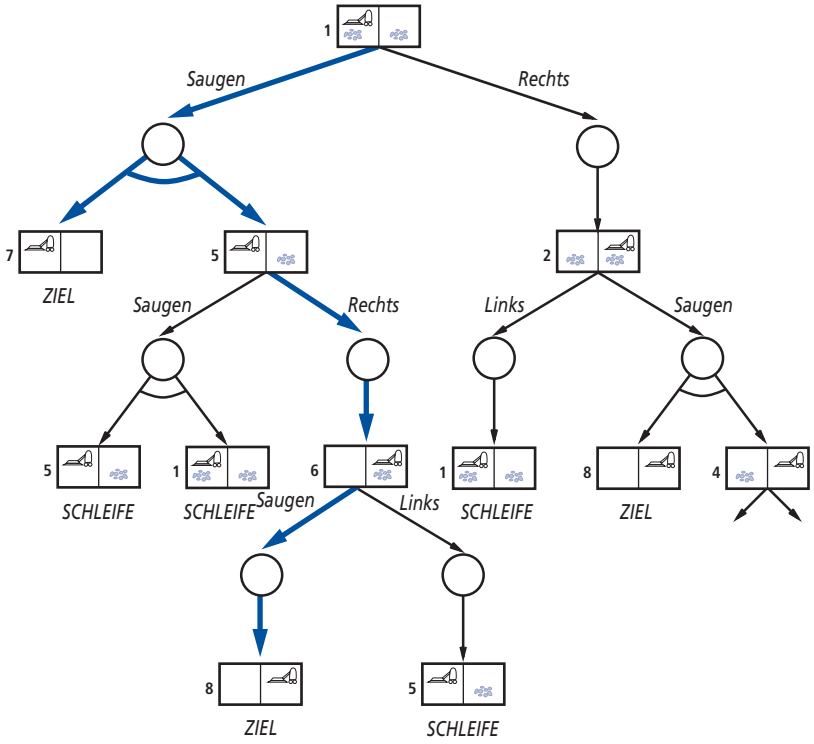


Abbildung 4.10: Die beiden ersten Ebenen des Suchbaumes für die erratische Staubsaugerwelt. Zustandsknoten sind OR-Knoten, wo eine bestimmte Aktion ausgewählt werden muss. Bei den als Kreise dargestellten AND-Knoten muss jedes Ergebnis behandelt werden, wie es der Bogen anzeigt, der die abgehenden Zweige verbindet. Die gefundene Lösung ist mit fetten Linien gekennzeichnet.

Eine Lösung für ein AND-OR-Suchproblem ist ein Teilbaum, der (1) einen Zielknoten an jedem Blatt besitzt, (2) eine Aktion an jedem seiner OR-Knoten spezifiziert und (3) jeden Ergebniszweig an jedem seiner AND-Knoten einbindet. Abbildung 4.10 zeigt die Lösung mit fetten Linien. Die Lösung entspricht dem Plan, der in Gleichung (4.3) angegeben ist. (Der Plan verwendet die **if-then-else**-Notation für die AND-Zweige. Wenn ein Knoten mehr als zwei Zweige aufweist, ist eine **case**-Konstruktion eventuell besser geeignet.) Es ist recht unkompliziert, den in Abbildung 4.1 gezeigten grundlegenden Agenten zur Problemlösung anzupassen, um derartige Kontingenzlösungen auszuführen. Es käme auch ein etwas abweichendes Agentendesign infrage, bei dem der Agent handeln kann, bevor er einen garantierten Plan gefunden hat und sich mit bestimmten Kontingenzen nur befasst, wenn sie während der Ausführung auftreten. Eine derartige **Verschachtelung** (**Interleaving**) von Suche und Ausführung ist auch nützlich für das Untersuchen von Problemen (siehe *Abschnitt 4.5*) und für Spiele (siehe *Kapitel 5*).

▶ Abbildung 4.11 gibt einen rekursiven Algorithmus mit Tiefensuche für AND-OR-Graphensuche an. Ein entscheidender Aspekt des Algorithmus ist die Art und Weise, in der er Zyklen behandelt, die oftmals in nichtdeterministischen Problemen auftreten (z.B. wenn eine Aktion manchmal keine Wirkung hat oder wenn ein nicht beabsichtigter Effekt korrigiert werden kann). Ist der aktuelle Zustand identisch mit einem Zustand auf dem Pfad von der Wurzel aus, endet der Algorithmus und gibt einen Fehler zurück. Das heißt

nicht, dass es keine Lösung vom aktuellen Zustand aus gibt; es bedeutet einfach, dass er bei Vorhandensein einer nichtzyklischen Lösung von der früheren Verkörperung des aktuellen Zustandes erreichbar sein muss, sodass die neue Verkörperung verworfen werden kann. Mit diesem Test können wir sicherstellen, dass der Algorithmus in jedem endlichen Zustandsraum terminiert, da jeder Pfad ein Ziel, eine Sackgasse oder einen wiederholten Zustand erreichen muss. Der Algorithmus prüft allerdings nicht, ob es sich beim aktuellen Zustand um die Wiederholung eines Zustandes auf irgendeinem anderen Pfad von der Wurzel handelt, was für die Effizienz wichtig ist. Übung 4.4 geht dieser Frage nach.

```
function AND-OR-GRAPH-SEARCH(problem) returns einen Bedingungsplan oder Fehler
  OR-SEARCH(problem.INITIAL-STATE, problem, [ ])

function OR-SEARCH(state, problem, path) returns einen Bedingungsplan oder
Fehler
  if problem.GOAL-TEST(state) then return den leeren Plan
  if state ist auf path then return failure
  for each action in problem.ACTIONS(state) do
    plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
    if plan = failure then return [action | plan]
  return failure
```

```
function AND-SEARCH(states, problem, path) returns einen Bedingungsplan oder
Fehler
  for each s_i in states do
    plan_i ← OR-SEARCH(s_i , problem, path)
    if plan_i = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1}
  else plan_n ]
```

Abbildung 4.11: Ein Algorithmus für die Suche in AND-OR-Graphen, die von nichtdeterministischen Umgebungen erzeugt werden. Er gibt einen bedingten Plan zurück, der unter allen Umständen einen Zielzustand erreicht. (Die Notation $[x \mid l]$ verweist auf die Liste, die gebildet wird, indem Objekt x vorn an die Liste l angefügt wird.)

AND-OR-Graphen lassen sich auch durch Methoden der Breitensuche oder Bestensuche erkunden. Das Konzept einer Heuristikfunktion muss modifiziert werden, um die Kosten einer Kontingenzlösung statt einer Sequenz abzuschätzen, doch ist das Konzept der Zulässigkeit weiterhin gültig und es gibt ein Analogon zum A*-Algorithmus für das Suchen optimaler Lösungen. In den bibliografischen Hinweisen am Ende des Kapitels sind weiterführende Quellen angegeben.

4.3.3 Immer wieder versuchen

Sehen Sie sich die unsichere Staubsaugerwelt an, die mit der gewöhnlichen (nicht erratischen) Staubsaugerwelt identisch ist, außer dass Bewegungsaktionen manchmal scheitern, wobei der Agent auf derselben Stelle bleibt. Zum Beispiel führt die Bewegung *Rechts* im Zustand 1 zur Zustandsmenge {1, 2}. ▶ Abbildung 4.12 zeigt Teile der Graphensuche. Offenbar gibt es vom Zustand 1 aus keinerlei azyklische Lösungen mehr und AND-OR-GRAPH-SEARCH würde mit Fehler zurückkehren. Allerdings gibt es eine **zyklische Lösung**, die ständig *Rechts* probiert, bis es funktioniert. Um diese Lösung auszudrücken, können Sie mit einem neuen **Bezeichner** einen Teil des Planes kennzeichnen und später diesen Bezeichner anstelle des Planes selbst verwenden.

Somit sieht unsere zyklische Lösung wie folgt aus:

$$[Saugen, L_1 : Rechts, \textbf{\textit{if}}\ State = 5\ \textbf{\textit{then}}\ L_1\ \textbf{\textit{else}}\ Saugen].$$

(Eine bessere Syntax für den Schleifenteil dieses Planes wäre „***while*** *State* = 5 ***do*** *Rechts*“.) Im Allgemeinen lässt sich ein zyklischer Plan als Lösung ansehen, sofern jeder Blattknoten einen Zielzustand verkörpert und ein Blatt von jedem Punkt im Plan aus erreichbar ist. Übung 4.5 behandelt die für AND-OR-GRAPH-SEARCH nötigen Modifikationen. Entscheidend ist, dass eine Schleife im Zustandsraum zurück zu einem Zustand L übersetzt wird in eine Schleife im Plan zurück zu dem Punkt, wo der Teilplan für Zustand L ausgeführt wird.

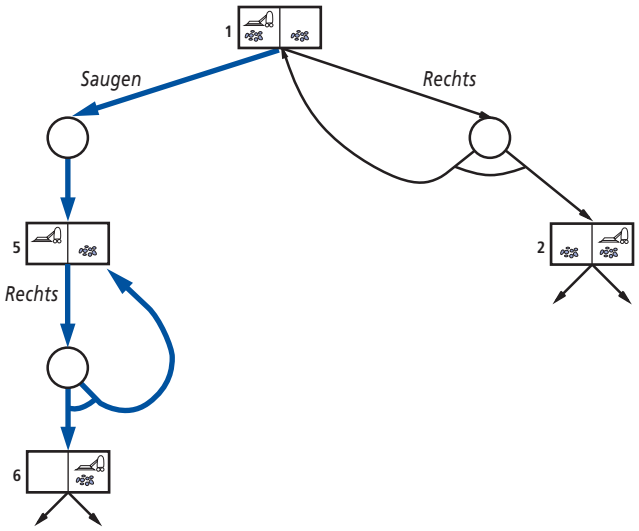


Abbildung 4.12: Teil des Suchgraphen für die unsichere Staubsaugerwelt mit (einigen) explizit dargestellten Zyklen. Alle Lösungen für dieses Problem sind zyklische Pläne, da es keine Möglichkeit gibt, zuverlässig zu ziehen.

Gemäß der Definition einer zyklischen Lösung erreicht ein Agent, der eine derartige Lösung ausführt, letztlich das Ziel, sofern das Ergebnis einer nichtdeterministischen Aktion schließlich auftritt. Ist diese Bedingung sinnvoll? Das hängt von der Ursache für den Nichtdeterminismus ab. Wenn die Aktion einen Würfel wirft, ist es sinnvoll anzunehmen, dass irgendwann eine Sechs fällt. Dagegen kann eine Aktion, die im Hotel eine Schlüsselkarte in das Türschloss einführt, beim ersten Mal nicht funktionieren, irgendwann aber doch klappen, außer wenn der Gast den falschen Schlüssel besitzt (oder sich im Zimmer geirrt hat!). Nach sieben oder acht Versuchen nehmen die meisten Leute an, dass etwas mit dem Schlüssel nicht stimmt, und wenden sich an die Rezeption, um sich einen neuen Schlüssel geben zu lassen. Um diese Entscheidung zu verstehen, kann man sagen, dass die anfängliche Problemformulierung (beobachtbar, nichtdeterministisch) zugunsten einer anderen Formulierung (teilweise beobachtbar, deterministisch) aufgegeben wird, wo der Fehler auf eine nicht beobachtbare Eigenschaft des Schlüssels zurückzuführen ist. *Kapitel 13* geht ausführlicher auf dieses Thema ein.

4.4 Mit partiellen Beobachtungen suchen

Wir kommen nun zum Problem der partiellen Beobachtbarkeit, wo die Wahrnehmungen des Agenten nicht ausreichen, um den genauen Zustand festzumachen. Wie bereits zu Beginn des vorherigen Abschnittes erwähnt, kann eine Aktion – selbst in einer deterministischen Umgebung – zu einem von mehreren möglichen Ergebnissen führen, wenn sich der Agent in einem von mehreren möglichen Zuständen befindet. Das erforderliche Schlüsselkonzept für das Lösen partiell beobachtbarer Probleme ist der **Belief State**.[10] Er repräsentiert die aktuellen Annahmen des Agenten über die möglichen physischen Zustände, in denen er sich befinden kann, wobei die Folge der Aktionen und Wahrnehmungen bis zu diesem Zeitpunkt herangezogen werden. Wir beginnen mit dem einfachsten Szenario für die Untersuchung der Belief States, in dem der Agent überhaupt keine Sensoren besitzt, und bringen dann partielles Abtasten sowie nichtdeterministische Aktionen ein.

4.4.1 Suchen ohne Beobachtung

Wenn die Wahrnehmungen des Agenten *überhaupt keine Informationen* liefern, haben wir es mit einem sogenannten **sensorlosen** Problem oder auch **konformen** Problem zu tun. Auf den ersten Blick scheint für den sensorlosen Agenten keine Hoffnung zu bestehen, ein Problem zu lösen, wenn er keine Ahnung hat, in welchem Zustand er sich befindet. Praktisch sind sensorlose Probleme recht oft lösbar. Darüber hinaus können sensorlose Agenten überraschend nützlich sein, hauptsächlich deshalb, weil sie sich nicht auf eine korrekte Arbeitsweise von Sensoren verlassen. Zum Beispiel wurden in Fertigungssystemen viele ausgeklügelte Methoden entwickelt, um Teile aus einer unbekannten Ausgangsposition mithilfe einer Folge von Aktionen ohne irgendwelche Sensorik korrekt zu orientieren. Die hohen Kosten für die Sensorik stellen einen weiteren Grund dar, um es zu vermeiden: Zum Beispiel verschreiben Ärzte oftmals ein Breitband-Antibiotikum – anstatt gemäß dem Kontingenzplan einen teuren Bluttest anzuordnen, auf die Rückmeldung der Ergebnisse zu warten, um dann ein spezifischeres Antibiotikum und vielleicht einen Klinikaufenthalt zu verschreiben, weil die Infektion inzwischen zu weit fortgeschritten ist.

Die Staubsaugerwelt können wir zu einer sensorlosen Version machen. Angenommen, der Agent kennt die Geografie seiner Welt, weiß aber nicht, wo er sich befindet oder wie die Schmutzverteilung aussieht. In diesem Fall kommt als Ausgangszustand jedes Element der Menge {1, 2, 3, 4, 5, 6, 7, 8} infrage. Wie sieht es nun aus, wenn er die Aktion *Rechts* ausprobiert? Dadurch gelangt er in einen der Zustände {2, 4, 6, 8} – der Agent besitzt nun mehr Informationen! Zudem endet die Aktionsfolge [*Rechts*, *Saugen*] immer in einem der Zustände {4, 8}. Schließlich erreicht die Sequenz [*Rechts*, *Saugen*, *Links*, *Saugen*] unabhängig vom Ausgangszustand garantiert den Zielzustand 7. Wir sagen, dass der Agent die Welt in den Zustand 7 **zwingen** kann.

10 Ein „Belief State“ (Glaubenszustand) ist eine hypothetischer Zustand, der durch eine Wahrscheinlichkeitsverteilung über eine Teilmenge aller Zustände definiert wird.

Um sensorlose Probleme zu lösen, suchen wir im Raum der Belief States und nicht der physischen Zustände.[11] Im Belief-States-Raum ist das Problem *vollständig beobachtbar*, weil der Agent immer seinen eigenen Belief State kennt. Darüber hinaus ist die Lösung (falls vorhanden) immer eine Aktionsfolge. Das hängt wie in den gewöhnlichen Problemen von *Kapitel 3* damit zusammen, dass die nach jeder Aktion empfangenen Wahrnehmungen vollständig vorhersagbar sind – sie sind immer leer! Deshalb sind auch keine Kontingenzen einzukalkulieren. Das gilt selbst dann, *wenn die Umgebung nichtdeterministisch ist.*

Es ist lehrreich zu sehen, wie das Belief-State-Suchproblem konstruiert wird. Das zugrunde liegende Problem P sei durch ACTIONS_P, RESULT_P, GOAL-TEST_P und STEP-COST_P definiert. Wir können dann das entsprechende sensorlose Problem wie folgt definieren:

- **Belief States:** Der gesamte Belief-States-Raum enthält jede mögliche Menge von physischen Zuständen. Wenn P über N Zustände verfügt, hat das sensorlose Problem bis zu 2^N Zustände, obwohl viele vom Anfangszustand aus unerreichbar sein können.
- **Anfangszustand:** typischerweise die Menge aller Zustände in P, obwohl der Agent in manchen Fällen mehr Kenntnisse als diese besitzt.
- **Aktionen:** Dies ist etwas kompliziert. Angenommen, der Agent befindet sich im Belief State $b = \{s_1, s_2\}$, doch ist $\text{ACTIONS}_P(s_1) \neq \text{ACTIONS}_P(s_2)$. Der Agent ist dann unsicher, welche Aktionen legal sind. Wenn wir davon ausgehen, dass illegale Aktionen keine Wirkung auf die Umgebung haben, ist es sicher, die *Vereinigung* aller Aktionen in jedem der physischen Zustände im aktuellen Belief State b zu bilden:

$$ACTIONS(b) = \bigcup_{s \in b} ACTIONS_P(s).$$

  Wenn dagegen eine unzulässige Aktion das Ende der Welt bedeuten könnte, ist es sicherer, nur die Schnittmenge (den Durchschnitt) zuzulassen, d.h. die Menge der Aktionen, die in allen Zuständen zulässig sind. Für die Staubsaugerwelt hat jeder Zustand die gleichen zulässigen Aktionen, sodass beide Methoden das gleiche Ergebnis liefern.
- **Übergangsmodell:** Der Agent weiß nicht, welcher Zustand im Belief State der richtige ist. Er kann also entsprechend seinem Kenntnisstand zu jedem der Zustände gelangen, die aus der Anwendung der Aktion auf einen der physischen Zustände im Belief State resultieren. Für deterministische Aktionen kann die Menge

$$b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ und } s \in b\} \tag{4.4}$$

  erreicht werden. Bei deterministischen Aktionen ist b' niemals größer als b. Mit Nichtdeterminismus haben wir

$$b' = \text{RESULT}(b, a) = \{s' : s' \in \text{RESULTS}_P(s, a) \text{ und } s \in b\}$$

$$= \bigcup_{s \in b} RESULTS_P(s,a)\,,$$

  was größer als b sein kann, wie ▶ Abbildung 4.13 zeigt. Das Generieren des neuen Belief State nach der Aktion ist der sogenannte **Vorhersageschritt**; die Notation $b' = \text{PREDICT}_P(b, a)$ erweist sich hier als praktisch.

11 In einer vollständig beobachtbaren Umgebung enthält jeder Glaubenszustand genau einen physischen Zustand. Somit können wir die Algorithmen in *Kapitel 3* als Suchen in einem Glaubenszustandsraum von einzelnen Glaubenszuständen ansehen.

- **Zieltest:** Der Agent möchte einen Plan, der sicher funktioniert, d.h., dass ein Belief State nur dann zum Ziel führt, wenn alle physischen Zustände in ihm GOAL-TEST $_P$ erfüllen. Der Agent kann das Ziel zwar *zufällig* früher erreichen, doch *weiß* er das nicht.
- **Pfadkosten:** Dies ist ebenfalls kompliziert. Wenn für dieselbe Aktion in unterschiedlichen Zuständen verschiedene Kosten möglich sind, könnten die Kosten für die Ausführung einer Aktion in einem bestimmten Belief State einen von mehreren Werten annehmen. (Das führt zu einer neuen Klasse von Problemen, die wir in Übung 4.8 untersuchen.) Momentan nehmen wir an, dass die Kosten einer Aktion in allen Zuständen gleich sind und sich somit direkt aus dem zugrunde liegenden physischen Problem übertragen lassen.

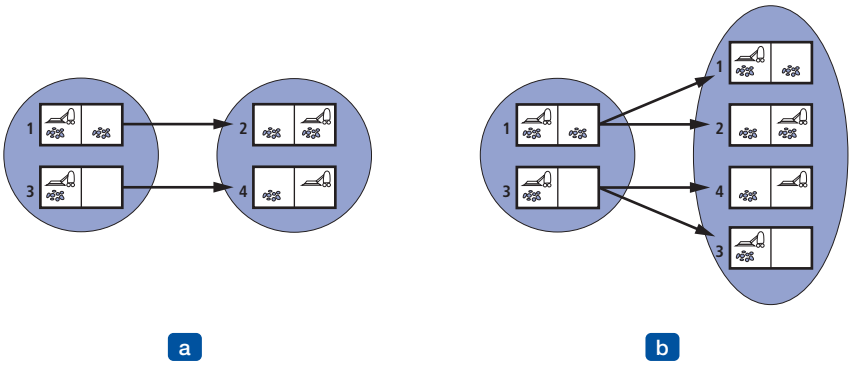


Abbildung 4.13: (a) Vorhersage des nächsten Belief State für die sensorlose Staubsaugerwelt mit einer deterministischen Aktion, *Rechts*. (b) Vorhersage für denselben Belief State und dieselbe Aktion in der unsicheren Version der sensorlosen Staubsaugerwelt.

▶ Abbildung 4.14 zeigt den erreichbaren Belief-States-Raum für die deterministische, sensorlose Staubsaugerwelt. Unter den $2^8 = 256$ möglichen Belief States gibt es nur zwölf erreichbare Belief States.

Die obigen Definitionen erlauben es, die Belief-State-Problemformulierung aus der Definition des zugrunde liegenden physischen Problems automatisch zu konstruieren. Anschließend können wir einen der Suchalgorithmen von *Kapitel 3* anwenden. Praktisch ist sogar noch etwas mehr als dies möglich. In der „normalen" Graphensuche werden neu generierte Zustände daraufhin überprüft, ob sie mit vorhandenen Zuständen identisch sind. Dies funktioniert auch für Belief States. Zum Beispiel erreicht in Abbildung 4.14 die Aktionsfolge [*Saugen*, *Links*, *Saugen*], die beim Ausgangszustand beginnt, denselben Belief State wie [*Rechts*, *Links*, *Saugen*], nämlich {5, 7}. Sehen Sie sich nun den Belief State an, der durch [*Links*] erreicht wird, nämlich {1, 3, 5, 7}. Offenbar ist dies nicht mit {5, 7} identisch, es stellt aber eine *Obermenge* dar. Wenn eine Aktionsfolge eine Lösung für einen Belief State *b* ergibt, ist sie auch eine Lösung für jede Teilmenge von *b*, wie sich leicht beweisen lässt (siehe Übung 4.7). Folglich können wir einen Pfad, der {1, 3, 5, 7} erreicht, verwerfen, wenn {5, 7} bereits erzeugt worden ist. Wurde umgekehrt bereits {1, 3, 5, 7} generiert und als lösbar befunden, dann ist jede *Teilmenge* wie zum Beispiel {5, 7} garantiert lösbar. Diese zusätzliche Kürzungsebene kann die Effizienz der sensorlosen Problemlösung erheblich verbessern.

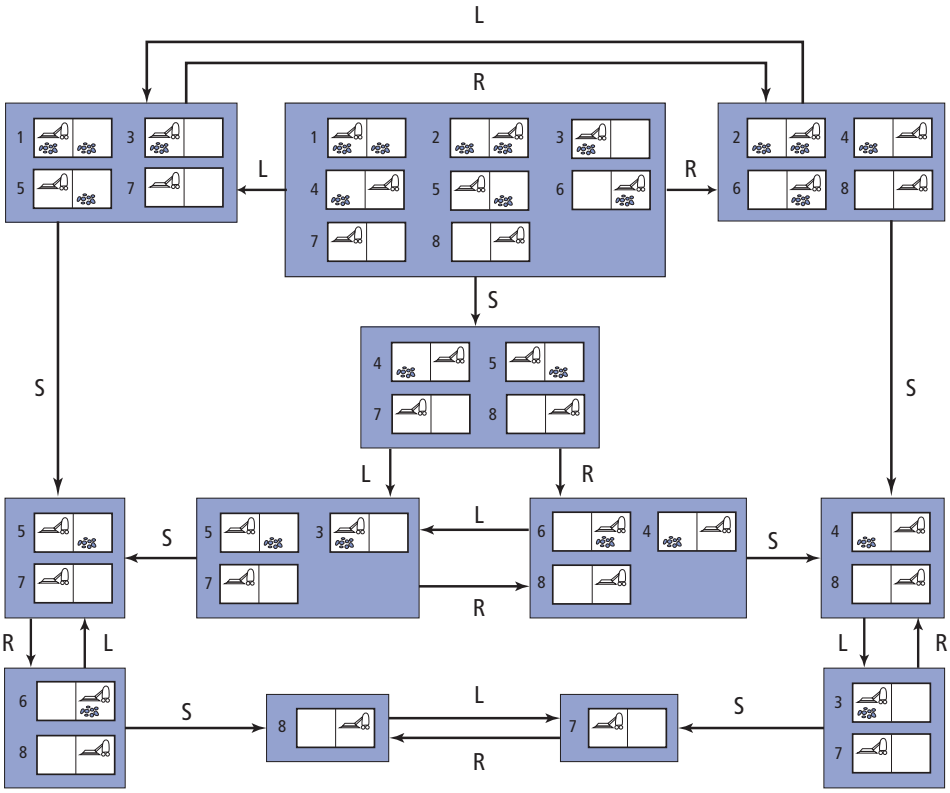


Abbildung 4.14: Der erreichbare Teil des Belief-States-Raumes für die deterministische, sensorlose Staubsaugerwelt. Jedes grau unterlegte Feld entspricht einem einzelnen Belief State. Der Agent befindet sich zu jedem beliebigen Zeitpunkt in einem bestimmten Belief State, weiß aber nicht, in welchem physischen Zustand er sich befindet. Der anfängliche Belief State (völliges Unwissen) ist das obere Feld in der Mitte. Aktionen werden durch beschriftete Pfeile dargestellt. Schleifen wurden der Übersichtlichkeit halber weggelassen.

Selbst mit dieser Verbesserung ist die sensorlose Problemlösung in der hier beschriebenen Form nur selten praktisch realisierbar. Die Schwierigkeit ergibt sich nicht so sehr aus der Mächtigkeit des Belief-States-Raumes – selbst wenn er exponentiell größer als der zugrunde liegende physische Zustandsraum ist; in den meisten Fällen sind Verzweigungsfaktor und Lösungslänge im Belief-States-Raum und physischen Zustandsraum ohnehin nicht sehr verschieden. Die eigentliche Schwierigkeit liegt in der Größe jedes Belief State. Zum Beispiel enthält der anfängliche Belief State für die 10×10-Staubsaugerwelt 100×2^{100} oder rund 10^{32} physische Zustände – bei weitem zu viel, wenn wir die atomare Darstellung verwenden, die eine explizite Liste der Zustände verkörpert.

Eine Lösung besteht darin, den Belief State durch eine kompaktere Beschreibung darzustellen. Zum Beispiel könnte man in Deutsch sagen, dass der Agent im Ausgangszustand „nichts“ weiß, und nachdem er die Aktion *Links* ausgeführt hat, sagen: „Nicht in der

ganz rechten Spalte“ usw. *Kapitel 7* erläutert, wie sich dies in einem formalen Darstellungsschema realisieren lässt. Ein anderer Ansatz ist es, die standardmäßigen Suchalgorithmen zu vermeiden, die Belief States als Blackboxes genau wie jeden anderen Problemzustand behandeln. Stattdessen können wir einen Blick *in* die Belief States werfen und Algorithmen der **inkrementellen Belief-States-Suche** entwickeln, die die Lösung mit jeweils einem physischen Zustand auf einmal aufbauen. Zum Beispiel ist der anfängliche Belief State in der sensorlosen Staubsaugerwelt {1, 2, 3, 4, 5, 6, 7, 8} und wir müssen eine Aktionsfolge finden, die in allen acht Zuständen funktioniert. Dazu können wir folgendermaßen vorgehen: Zuerst suchen wir eine Lösung, die für Zustand 1 funktioniert, und überprüfen dann, ob sie auch für Zustand 2 funktioniert. Trifft das nicht zu, gehen wir zurück und suchen nach einer anderen Lösung für Zustand 1 usw. Genau wie eine AND-OR-Suche eine Lösung für jeden Zweig bei einem AND-Knoten zu finden hat, muss dieser Algorithmus eine Lösung für jeden Zustand im Belief State finden. Der Unterschied besteht darin, dass eine AND-OR-Suche eine andere Lösung für jeden Zweig finden kann, während eine inkrementelle Belief-States-Suche *eine* Lösung finden muss, die für *alle* Zustände funktioniert. Der inkrementelle Ansatz hat vor allem den Vorteil, ein Scheitern schnell erkennen zu können – ist ein Belief State nicht lösbar, wird normalerweise auch eine kleine Teilmenge des Belief State, die aus den ersten untersuchten Zuständen besteht, ebenfalls nicht lösbar sein. In manchen Fällen führt dies zu einer Beschleunigung, die proportional zur Größe der Belief States ist, die ihrerseits so groß wie der physische Zustandsraum selbst sein können.

Selbst der effizienteste Lösungsalgorithmus ist praktisch wertlos, wenn keine Lösungen existieren. Viele Dinge lassen sich ohne Sensorik nicht ausführen. Das trifft zum Beispiel auf das sensorlose 8-Puzzle zu. Andererseits kommt man mit ein wenig Sensorik schon ein gutes Stück weiter. So lässt sich jede 8-Puzzle-Instanz lösen, wenn lediglich ein Quadrat sichtbar ist – um die Lösung zu erhalten, verschiebt man nacheinander jedes Feld in das sichtbare Quadrat und verfolgt dann seine Position.

4.4.2 Suchen mit Beobachtungen

Für ein im Allgemeinen partiell beobachtbares Problem müssen wir spezifizieren, wie die Umgebung Wahrnehmungen für den Agenten generiert. Zum Beispiel könnten wir die Staubsaugerwelt mit lokaler Abtastung als eine Umgebung definieren, in der der Agent einen Positionssensor und einen Sensor für den unmittelbar vor Ort vorhandenen Schmutz besitzt, aber keinen Sensor, mit dem sich Schmutz in anderen Quadraten erkennen lässt. Die formale Problemspezifikation umfasst eine Funktion PERCEPT(s), die die in einem bestimmten Zustand erhaltene Wahrnehmung zurückgibt. (Wenn die Erfassung nichtdeterministisch ist, verwenden wir eine PERCEPTS-Funktion, die mögliche Wahrnehmungen als Menge zurückgibt.) Zum Beispiel liefert die Funktion PERCEPT im Zustand 1 der Staubsaugerwelt mit lokaler Abtastung das Ergebnis [*A*, *Schmutzig*]. Vollständig beobachtbare Probleme sind ein Spezialfall, in dem PERCEPT(s) = s für jeden Zustand s gilt, während sensorlose Probleme einen Spezialfall darstellen, in dem PERCEPT(s) = *null* ist.

Bei partiellen Beobachtungen ist es üblicherweise so, dass eine bestimmte Wahrnehmung von mehreren Zuständen hervorgerufen worden sein kann. Zum Beispiel wird die Wahrnehmung [*A*, *Schmutzig*] sowohl durch Zustand 3 als auch durch Zustand 1 erzeugt. Wenn dies die anfängliche Wahrnehmung ist, wird folglich der anfängliche Belief State für die Staubsaugerwelt mit lokaler Abtastung {1, 3} sein. Die Funktionen ACTIONS, STEP-COST und GOAL-TEST werden aus dem zugrunde liegenden physischen Problem genauso wie für sensorlose Probleme konstruiert, das Übergangsmodell ist jedoch ein wenig komplizierter. Wie ▶ Abbildung 4.15 zeigt, kann man sich die Übergänge von einem Belief State zum nächsten für eine bestimmte Aktion als Ablauf in drei Stufen vorstellen:

- Die Stufe der **Vorhersage** ist die gleiche wie für sensorlose Probleme; für eine bestimmte Aktion a im Belief State b ist der vorhergesagte Belief State $\hat{b}$ = PREDICT (b,a).[12]
- Die Stufe der **Beobachtungsvorhersage** bestimmt die Menge der Wahrnehmungen o, die im vorhergesagten Belief State beobachtet werden könnten:

$$\text{POSSIBLE-PERCEPTS}(\hat{b}) = \{o : o = \text{PERCEPT}(s) \text{ und } s \in \hat{b}\}.$$

- Die Stufe der Aktualisierung bestimmt für jede mögliche Wahrnehmung den Belief State, der aus der Wahrnehmung resultieren würde. Der neue Belief State b_o ist einfach die Menge der Zustände in $\hat{b}$, die die Wahrnehmung erzeugt haben könnten:

$$b_o = \text{UPDATE}(\hat{b}, o) = \{s : o = \text{PERCEPT}(s) \text{ und } s \in \hat{b}\}.$$

  Jeder aktualisierte Belief State b_o kann nicht größer als der vorhergesagte Belief State $\hat{b}$ sein; Beobachtungen können nur helfen, Unbestimmtheit im Vergleich zum sensorlosen Fall zu verringern. Beim deterministischen Abtasten sind darüber hinaus die Belief States für die verschiedenen möglichen Wahrnehmungen getrennt, wobei sie eine *Partition* des ursprünglich vorhergesagten Belief State bilden.

Bringt man diese drei Stufen zusammen, erhalten wir die möglichen Belief States, die aus einer bestimmten Aktion resultieren, und die darauffolgenden möglichen Wahrnehmungen:

$$\text{RESULTS}(b, a) = \{b_o : b_o = \text{UPDATE}(\text{PREDICT}(b, a), o) \text{ und}$$
$$o \in \text{POSSIBLE-PERCEPTS}(\text{PREDICT}(b, a))\}. \quad (4.5)$$

Auch hier stammt der Nichtdeterminismus im partiell beobachtbaren Problem von der Unfähigkeit, genau vorherzusagen, welche Wahrnehmung erhalten wird, nachdem die Aktion stattgefunden hat; zugrunde liegender Nichtdeterminismus in der physischen Umgebung kann zu dieser Unfähigkeit beitragen, indem der Belief State auf der Vorhersagestufe vergrößert wird, was zu mehr Wahrnehmungen auf der Beobachtungsstufe führt.

12 Hier und das ganze Buch hindurch bedeutet der „Hut“ auf dem b einen geschätzten oder vorhergesagten Wert für b.

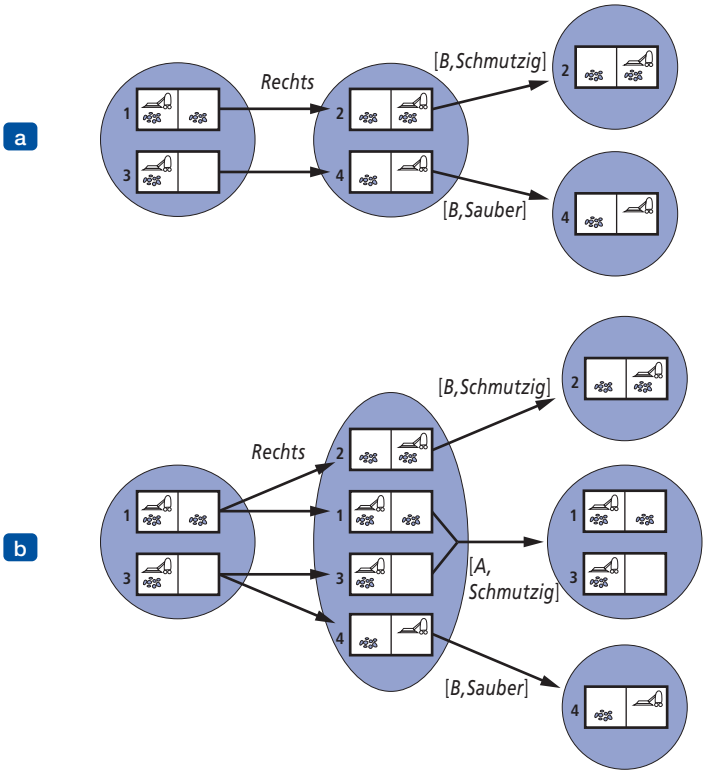


Abbildung 4.15: Zwei Beispiele für Übergänge in Staubsaugerwelten mit lokalem Abtasten. (a) In der deterministischen Welt wird *Rechts* im anfänglichen Belief State angewandt, was in einem neuen Belief State mit zwei möglichen physischen Zuständen resultiert; für diese Zustände sind die möglichen Wahrnehmungen [*B*, *Schmutzig*] und [*B*, *Sauber*], was zu zwei Belief States führt, die jeweils einelementige Mengen darstellen. (b) In der unsicheren Welt wird *Rechts* im anfänglichen Belief State angewandt, was zu einem neuen Belief State mit vier physischen Zuständen führt; für diese Zustände sind die möglichen Wahrnehmungen [*A*, *Schmutzig*], [*B*, *Schmutzig*] und [*B*, *Sauber*], was die drei gezeigten Belief States ergibt.

4.4.3 Partiell beobachtbare Probleme lösen

Der vorherige Abschnitt hat gezeigt, wie sich die Funktion RESULTS für ein nichtdeterministisches Belief-States-Problem aus einem zugrunde liegenden physischen Problem und der Funktion PERCEPT ableiten lässt. Anhand einer derartigen Formulierung kann der AND-OR-Suchalgorithmus von Abbildung 4.11 direkt angewandt werden, um eine Lösung abzuleiten. ▶ Abbildung 4.16 zeigt einen Teil des Suchbaumes für die Staubsaugerwelt mit lokaler Abtastung, wobei eine anfängliche Wahrnehmung [*A*, *Schmutzig*] angenommen wird. Die Lösung ist der Bedingungsplan

[*Saugen*, *Rechts*, ***if*** *Bstate* = {6} ***then*** *Saugen* ***else*** [ ]].

Da wir dem AND-OR-Suchalgorithmus ein Belief-States-Problem angeboten haben, gibt der Algorithmus einen Bedingungsplan zurück, der den Belief State und nicht den tatsächlichen Zustand testet. Genauso sollte es sein: In einer partiell beobachtbaren Umgebung ist der Agent nicht in der Lage, eine Lösung auszuführen, die das Testen des tatsächlichen Zustandes verlangt.

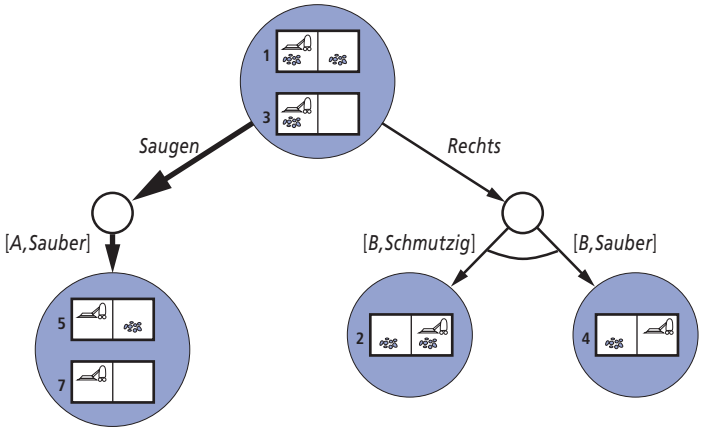


Abbildung 4.16: Die erste Ebene des AND-OR-Suchbaumes für ein Problem in der Staubsaugerwelt mit lokaler Abtastung; *Saugen* ist der erste Schritt der Lösung.

Wie im Fall der Standardsuchalgorithmen, die auf sensorlose Probleme angewendet werden, behandelt der AND-OR-Suchalgorithmus Belief States wie Blackboxes, genau wie alle anderen Zustände. Dies lässt sich verbessern, indem man – analog zu sensorlosen Problemen – auf vorher generierte Belief States prüft, die Teil- oder Obermengen des aktuellen Zustandes sind. Analog zu den für sensorlose Probleme beschriebenen Algorithmen lassen sich auch inkrementelle Suchalgorithmen ableiten, die gegenüber dem Blackbox-Konzept eine erhebliche Beschleunigung bieten.

4.4.4 Ein Agent für partiell beobachtbare Umgebungen

Der Entwurf eines problemlösenden Agenten für partiell beobachtbare Umgebungen ist recht ähnlich dem einfachen problemlösenden Agenten in Der Agent formuliert ein Problem, ruft einen Suchalgorithmus auf (wie zum Beispiel AND-OR-GRAPH-SEARCH), um es zu lösen, und führt die Lösung aus. Dabei gibt es vor allem zwei Unterschiede. Erstens stellt die Lösung für ein Problem einen Bedingungsplan und keine Sequenz dar; wenn der erste Schritt ein **if**-**then**-**else**-Ausdruck ist, muss der Agent die Bedingung im **if**-Zweig testen und den **then**-Zweig bzw. den **else**-Zweig entsprechend ausführen. Zweitens muss der Agent seinen Belief State verwalten, wenn er Aktionen durchführt und Wahrnehmungen empfängt. Dieser Vorgang erinnert an den Prozess der Vorhersage-Beobachtung-Aktualisierung in Gleichung (4.5.), ist jedoch tatsächlich einfacher, weil die Wahrnehmung durch die Umgebung geliefert und nicht durch Agenten berechnet wird. Mit einem anfänglichen Belief State b, einer Aktion a und einer Wahrnehmung o lautet der neue Belief State:

$$b' = \text{UPDATE}(\text{PREDICT}(b, a), o). \tag{4.6}$$

▶ Abbildung 4.17 zeigt den Belief State, wie er in der Kindergarten-Staubsaugerwelt mit lokaler Abtastung verwaltet wird, wobei jedes Quadrat jederzeit schmutzig werden kann, sofern der Agent es nicht momentan aktiv reinigt.[13]

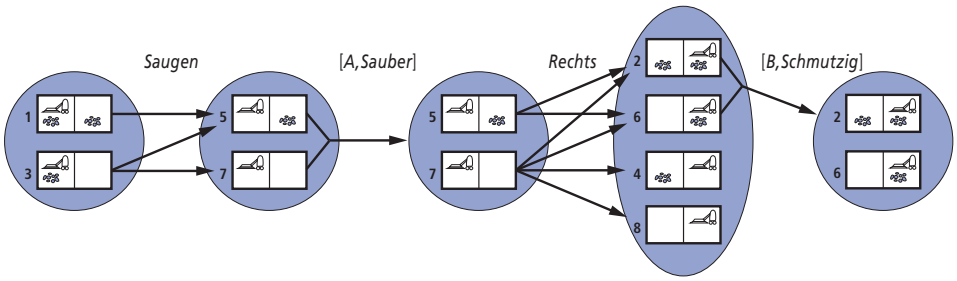
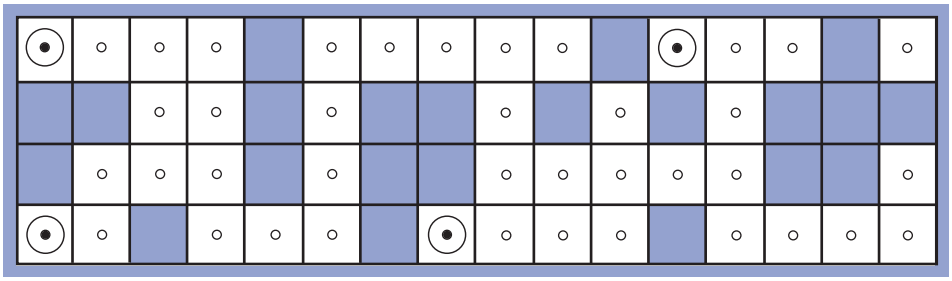


Abbildung 4.17: Zwei Vorhersage-Aktualisierungs-Zyklen der Belief-States-Verwaltung in der Kindergarten-Staubsaugerwelt mit lokaler Abtastung.

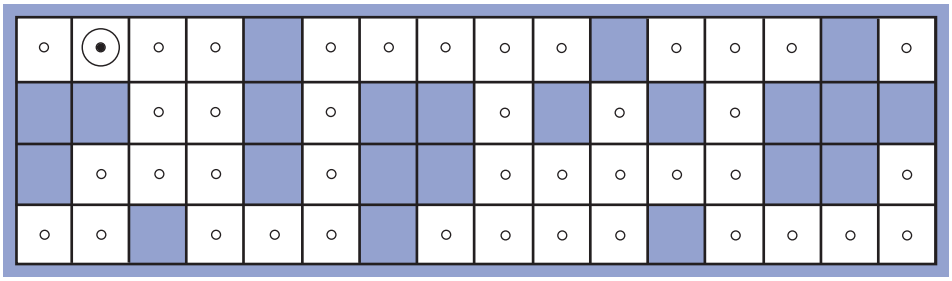
In teilweise beobachtbaren Umgebungen – zu denen die meisten Umgebungen in der realen Welt zählen – ist es eine Kernfunktion jedes intelligenten Systems, den Belief State zu verwalten. Diese Funktion läuft unter verschiedenen Namen, unter anderem **Überwachung**, **Filterung** und **Zustandsschätzung**. Gleichung (4.6) wird als **rekursiver** Zustandsschätzer bezeichnet, weil er den neuen Belief State aus dem vorherigen berechnet, anstatt die gesamte Wahrnehmungsfolge zu untersuchen. Damit der Agent nicht „zurückfällt", muss die Berechnung genauso schnell erfolgen, wie die Wahrnehmungen eintreffen. In komplexeren Umgebungen wird die Aktualisierungsberechnung undurchführbar – der Agent muss einen Belief State näherungsweise berechnen und sich dabei vielleicht auf die Implikationen der Wahrnehmung für die Aspekte der Umgebung konzentrieren, die momentan von Interesse sind. Dieses Problem wurde vor allem für stochastische, stetige Umgebungen mit den Werkzeugen der Wahrscheinlichkeitstheorie untersucht, wie *Kapitel 15* noch näher erläutert. Hier zeigen wir ein Beispiel in einer diskreten Umgebung mit deterministischen Sensoren und nichtdeterministischen Aktionen.

Im Beispiel geht es um einen Roboter, der eine Aufgabe der **Positionierung** zu lösen hat: Er muss anhand einer Weltkarte und einer Sequenz von Wahrnehmungen und Aktionen feststellen, wo er sich befindet. Unser Roboter wird in der labyrinthartigen Umgebung gemäß ▶ Abbildung 4.18 platziert. Als Ausstattung erhält er vier Sonarsensoren, die ihm für die vier Himmelsrichtungen sagen, ob es ein Hindernis gibt – in der Abbildung die Außenwand oder ein schwarzes Quadrat. Wir nehmen an, dass die Sensoren absolut korrekte Daten liefern und dass der Roboter eine richtige Karte der Umgebung besitzt. Doch leider ist das Navigationssystem des Roboters defekt. Führt er also eine *Bewegen*-Aktion aus, bewegt er sich zufällig auf eines der benachbarten Quadrate. Die Aufgabe des Roboters besteht nun darin, seine aktuelle Position zu bestimmen.

13 Hier entschuldigen wir uns wieder bei den Lesern, die mit den Wirkungen kleiner Kinder auf die Umgebung nicht vertraut sind.



a Mögliche Positionen des Roboters nach E_1 = NSW



b Mögliche Positionen des Roboters nach E_1 = NSW, E_2 = NS

Abbildung 4.18: Mögliche Positionen des Roboters, (*), (a) nach einer Beobachtung $E_1 = NSW$ und (b) nach einer zweiten Beobachtung $E_2 = NS$. Setzt man rauschfreie Sensoren und ein genaues Übergangsmodell voraus, gibt es keine anderen möglichen Positionen für den Roboter, die dieser Sequenz der beiden Beobachtungen entsprechen.

Angenommen, der Roboter hat bereits gewechselt und weiß demnach nicht, wo er sich befindet. Der anfängliche Belief State b besteht also aus der Menge aller Positionen. Der Roboter empfängt dann die Wahrnehmung *NSW* – d.h., die Hindernisse befinden sich im Norden, Westen und Süden – und führt eine Aktualisierung mithilfe der Gleichung b_o = UPDATE(b) durch, was die vier in Abbildung 4.18(a) gezeigten Positionen liefert. Anhand des Labyrinths können Sie sich davon überzeugen, dass es nur vier Positionen gibt, die die Wahrnehmung *NSW* liefern.

Als Nächstes führt der Roboter eine *Bewegen*-Aktion aus, doch ist das Ergebnis nichtdeterministisch. Der neue Belief State, b_a = PREDICT(b_o, *Bewegen*), enthält alle Positionen, die einen Schritt von den Positionen in b_o entfernt sind. Trifft die zweite Wahrnehmung, *NS*, ein, führt der Roboter UPDATE(b_a, *NS*) aus und stellt fest, dass der Belief State auf die einzelne Position zusammengeschrumpft ist, die in Abbildung 4.18(b) dargestellt ist. Das ist die einzige Position, die das Ergebnis von

$$\text{UPDATE}(\text{PREDICT}(\text{UPDATE}(b, NSW), \textit{Bewegen}), NS)$$

sein kann.

Mit nichtdeterministischen Aktionen vergrößert der PREDICT-Schritt den Belief State, doch der UPDATE-Schritt lässt ihn wieder schrumpfen – solange die Wahrnehmungen einige nützliche identifizierende Informationen liefern. Manchmal allerdings helfen die Wahrnehmungen kaum für die Positionierung: Wenn es einen oder mehrere lange Ost-West-Korridore gäbe, könnte ein Roboter zwar eine lange Folge von *NS*-Wahrnehmungen empfangen, aber niemals wissen, wo er sich im Korridor befindet.

4.5 Online-Suchagenten und unbekannte Umgebungen

Bisher haben wir uns auf Agenten konzentriert, die **Offline-Suche** verwenden. Sie berechnen eine vollständige Lösung, bevor sie einen Fuß in die reale Welt setzen, und führen dann die Lösung aus. Im Gegensatz dazu arbeitet ein Agent für **Online-Suche**,[14] indem er Berechnungen und Aktionen **verzahnt** (**Interleaving**): Er führt zuerst eine Aktion aus, beobachtet dann die Umgebung und berechnet die nächste Aktion. Die Online-Suche ist ein sinnvolles Konzept in dynamischen oder halbdynamischen Bereichen – das sind Bereiche, wo es einen Punktabzug gibt, wenn man zu lange mit Berechnungen zubringt. Die Online-Suche ist auch für nichtdeterministische Gebiete hilfreich, weil sie es dem Agenten erlaubt, seinen Berechnungsaufwand auf die Kontingenzen zu konzentrieren, die tatsächlich auftreten, statt auf diejenigen, die möglicherweise passieren, vielleicht aber auch nicht. Natürlich gibt es einen Kompromiss: Je mehr ein Agent im Voraus plant, desto seltener wird er in Schwierigkeiten geraten.

Die Online-Suche ist ein *notwendiges* Konzept für unbekannte Umgebungen, wo der Agent nicht weiß, welche Zustände existieren und was seine Aktionen bewirken. In diesem Zustand der Unwissenheit hat es der Agent mit einem **Explorationsproblem** zu tun und er muss seine Aktionen als Experimente nutzen, um genügend zu lernen und lohnenswerte Überlegungen anzustellen.

Das typische Beispiel für eine Online-Suche ist ein Roboter, der in einem neuen Haus platziert wird und es erkunden soll, um eine Karte zu schaffen, die er nutzen kann, um vom Punkt A zum Punkt B zu gelangen. Methoden zum Entfliehen aus Labyrinthen – Allgemeinwissen für Heldenanwärter des Altertums – sind ebenfalls Beispiele für Online-Suchalgorithmen. Die Erkundung von Räumen ist jedoch nicht die einzige Form der Exploration. Betrachten Sie ein neugeborenes Kind: Es ist zu vielen möglichen Aktionen fähig, kennt aber für keine davon das Ergebnis und hat nur ein paar der möglichen Zustände erfahren, die es erreichen kann. Die schrittweise Entdeckung der Welt ist für das Baby zum Teil ein Online-Suchprozess.

4.5.1 Online-Suchprobleme

Ein Online-Suchproblem kann nur von einem Agenten gelöst werden, der Aktionen ausführt, und nicht von einem reinen Rechenprozess. Wir nehmen zwar eine deterministische und vollständig beobachtbare Umgebung an (*Kapitel 17* lockert diese Annahmen), schreiben aber vor, dass der Agent nur Folgendes kennt:

- die Funktion ACTIONS(s), die eine Liste der im Zustand s erlaubten Aktionen zurückgibt,
- die Schrittkostenfunktion $c(s, a, s')$, die der Agent allerdings erst verwenden kann, wenn er weiß, dass s' das Ergebnis ist, und
- die Funktion GOAL-TEST(s).

Insbesondere ist wichtig, dass der Agent RESULT(s,a) *nur dann* bestimmen kann, wenn er sich in s befindet und a ausführt. Zum Beispiel weiß der Agent in dem in ▶ Abbildung 4.19 gezeigten Problem mit dem Irrgarten nicht, dass die Aktion *Auf* von (1,1) nach (1,2)

14 Der Begriff „online“ wird in der Informatik häufig gebraucht, um auf Algorithmen zu verweisen, die Eingabedaten verarbeiten müssen, sobald diese ankommen, anstatt darauf zu warten, bis die vollständige Eingabedatenmenge zur Verfügung steht.

führt; er weiß auch nicht, dass die Aktion *Ab* ihn zurück nach (1,1) bringt. Dieser Grad der Unwissenheit lässt sich in bestimmten Anwendungen reduzieren – beispielsweise könnte ein Erkundungsroboter wissen, wie seine Bewegungsaktionen funktionieren, und nur im Hinblick auf die Positionen von Hindernissen unwissend sein.

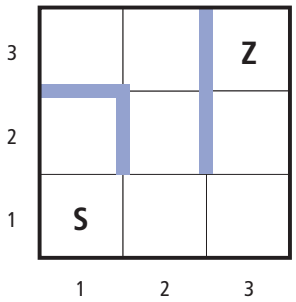


Abbildung 4.19: Ein einfaches Irrgartenproblem. Der Agent beginnt bei S und soll Z erreichen, weiß aber nichts über die Umgebung.

Schließlich könnte der Agent Zugriff auf eine zulässige Heuristikfunktion $h(s)$ haben, die den Abstand vom aktuellen Zustand zu einem Zielzustand abschätzt. In Abbildung 4.19 beispielsweise könnte der Agent die Position des Ziels kennen und dann in der Lage sein, die Manhattan-Distanzheuristik anzuwenden.

Normalerweise ist es das Ziel des Agenten, einen Zielzustand zu erreichen und dabei minimale Kosten zu verursachen. (Ein weiteres mögliches Ziel wäre es, einfach die gesamte Umgebung zu erkunden.) Die Kosten sind die gesamten Pfadkosten für den Pfad, den der Agent tatsächlich zurücklegt. Häufig werden diese Kosten mit den Fahrtkosten des Pfades verglichen, dem der Agent folgen würde, *wenn ihm der Suchraum im Voraus bekannt wäre* – d.h., mit dem kürzesten Pfad (oder der kürzesten vollständigen Erkundung). In der Terminologie der Online-Algorithmen spricht man auch vom **kompetitiven Faktor** (**Competitive Ratio**), der so klein wie möglich sein sollte.

Tipp

Diese Forderung hört sich zwar vernünftig an, aber man erkennt schnell, dass der beste erzielbare kompetitive Faktor in einigen Fällen unendlich ist. Sind beispielsweise einige Aktionen **nicht umkehrbar** (**irreversibel**) – d.h., sie führen zu einem Zustand, aus dem keine Aktion zum vorherigen Zustand zurückführt –, könnte die Online-Suche versehentlich in eine **Sackgasse** geraten, von der aus kein Zielzustand erreichbar ist. Vielleicht sind Sie der Meinung, dass der Begriff „versehentlich“ hier unpassend ist – schließlich könnte es einen Algorithmus geben, der den Sackgassenpfad nicht annimmt, wenn er ihn erkundet. Unsere Behauptung ist, um genau zu sein, dass *kein Algorithmus Sackgassen in allen Zustandsräumen vermeiden kann*. Betrachten Sie die beiden Sackgassen-Zustandsräume in ▶ Abbildung 4.20(a). Für einen Online-Suchalgorithmus, der die Zustände S und A besucht hat, sehen die beiden Zustandsräume identisch aus, sodass er in beiden Fällen dieselbe Entscheidung treffen muss. Aus diesem Grund versagt er in einem von ihnen. Dies ist ein Beispiel für ein **Widersacherargument** – wir können uns einen Widersacher vorstellen, der den Zustandsraum konstruiert, während der Agent diesen erkundet, und die Ziele und die Sackgassen nach Belieben platziert.

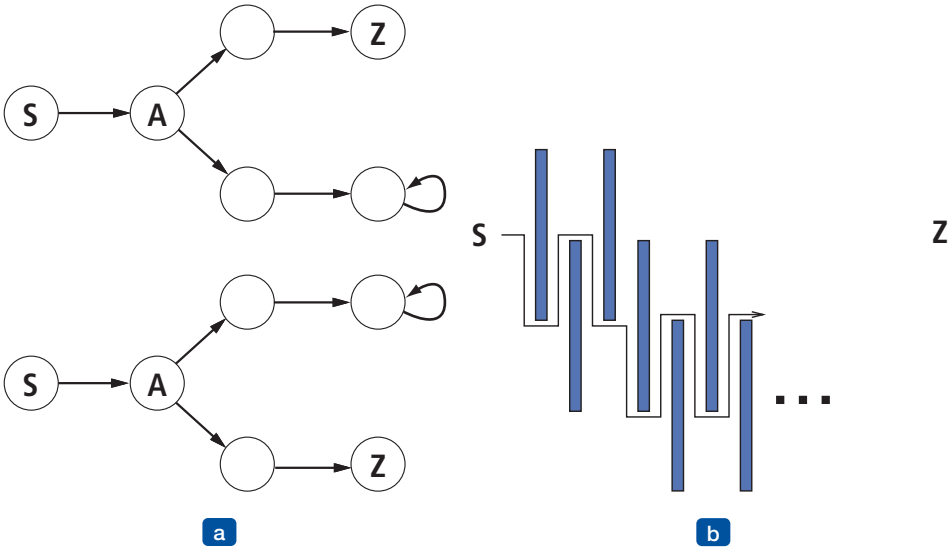


Abbildung 4.20: (a) Zwei Zustandsräume, die einen Online-Suchagenten in eine Sackgasse führen können. Ein Agent schlägt in mindestens einem dieser Räume fehl. (b) Eine zweidimensionale Umgebung, die einen Online-Suchagenten veranlassen kann, eine beliebige ineffiziente Route zu dem Ziel zu verfolgen. Egal, welche Auswahl der Agent trifft, der Widersacher blockiert diese Route mit einer anderen langen, dünnen Wand, sodass der verfolgte Pfad sehr viel länger als der bestmögliche Pfad ist.

Sackgassen sind ein wirklich schwieriges Problem für die Robotererkundung – Treppen, Rampen, Klippen, Einbahnstraßen und jede Art natürlichen Geländes bieten Gelegenheiten für nicht umkehrbare Aktionen. Um Fortschritte zu machen, nehmen wir einfach an, dass der Zustandsraum **sicher erkundbar** – d.h. von jedem erreichbaren Zustand aus ein Zielzustand erreichbar – ist. Zustandsräume mit umkehrbaren Aktionen, wie beispielsweise Irrgärten oder 8-Puzzles, können als nicht gerichtete Graphen betrachtet werden und sind offensichtlich sicher erkundbar.

Selbst in sicher erkundbaren Umgebungen kann kein begrenzter kompetitiver Faktor garantiert werden, wenn die Pfade unbegrenzte Kosten haben. Das kann einfach in Umgebungen mit nicht umkehrbaren Aktionen gezeigt werden, bleibt jedoch auch für den umkehrbaren Fall gültig, wie in ▶ Abbildung 4.20(b) gezeigt. Aus diesem Grund wird die Leistung von Online-Suchalgorithmen häufig im Hinblick auf die Größe des gesamten Zustandsraumes und nicht nur auf die Tiefe des flachsten Ziels beschrieben.

4.5.2 Online-Suchagenten

Nach jeder Aktion empfängt ein Online-Agent eine Wahrnehmung, die ihm mitteilt, welchen Zustand er erreicht hat; aus diesen Informationen kann er seine Karte der Umgebung erweitern. Anhand der aktuellen Karte kann er entscheiden, wohin er als Nächstes gehen soll. Diese Verzahnung von Planung und Aktion bedeutet, dass Online-Suchalgorithmen sich deutlich von Offline-Suchalgorithmen unterscheiden, die wir bereits kennengelernt haben. Beispielsweise haben Offline-Algorithmen wie etwa A* die Fähigkeit, einen Knoten in einem Teil und unmittelbar danach einen Knoten in einem anderen Teil des Raumes zu expandieren, weil bei der Knotenexpandierung simulierte und keine rea-

len Aktionen stattfinden. Ein Online-Algorithmus dagegen kann Nachfolger nur für einen Knoten entdecken, den er physisch belegt. Um zu vermeiden, dass der gesamte Weg durch den Baum zurückgelegt werden muss, um den nächsten Knoten zu expandieren, scheint es sinnvoller zu sein, Knoten in einer *lokalen Reihenfolge* zu expandieren. Die Tiefensuche hat genau diese Eigenschaft, weil (außer beim Backtracking) der nächste expandierte Knoten ein Nachfahre des zuvor expandierten Knotens ist.

▶ Abbildung 4.21 zeigt einen Online-Tiefensuche-Agent. Dieser Agent speichert seine Karte in einer Tabelle, *result*[*s*, *a*], die den Zustand aufzeichnet, der aus der Ausführung von Aktion *a* in Zustand *s* resultiert. Wenn eine Aktion aus dem aktuellen Zustand nicht erkundet wurde, probiert der Agent diese Aktion aus. Die Schwierigkeit entsteht, wenn der Agent alle Aktionen in einem Zustand ausprobiert hat. Bei der Offline-Tiefensuche wird der Zustand einfach aus der Warteschlange entfernt; bei einer Online-Suche muss der Agent sich physisch rückwärts bewegen. Bei der Tiefensuche bedeutet das, dass er zu dem Zustand zurückgeht, aus dem er in den aktuellen Zustand gelangt ist. Dazu verwaltet er eine Tabelle, die für jeden Zustand die Vorgängerzustände auflistet, zu denen der Agent noch nicht zurückgegangen ist. Gibt es keine weiteren Zustände mehr, zu denen der Agent zurückgehen kann, ist die Suche abgeschlossen.

```
function ONLINE-DFS-AGENT(s') returns eine Aktion
  inputs: s', eine Wahrnehmung, die den aktuellen Zustand kennzeichnet
  persistent: result, eine durch state und action indizierte Tabelle,
              anfangs leer
              untried, eine Tabelle, die für jeden Zustand die noch nicht
              ausprobierten Aktionen auflistet
              unbacktracked, eine Tabelle, die für jeden Zustand die noch nicht
              ausprobierten Rückwärtsschritte auflistet
              s, a, vorheriger Zustand und vorherige Aktion, anfangs null

  if GOAL-TEST(s') then return stop
  if s' ist ein neuer Zustand (nicht in untried) then untried[s'] ← ACTIONS(s')
  if s ungleich null then
      result[s, a] ← s'
      s am Anfang von unbacktracked[s'] einfügen
  if untried[s'] leer then
    if unbacktracked[s'] leer then return stop
    else a ← eine Aktion b, sodass result[s', b] = POP(unbacktracked[s'])
  else a ← POP(untried[s'])
  s ← s'
  return a
```

Abbildung 4.21: Ein Online-Suchagent, der eine Tiefensuche verwendet. Der Agent kann nur in Zustandsräumen angewendet werden, in denen sich jede Aktion durch eine bestimmte andere Aktion „rückgängig machen" lässt.

Wir empfehlen dem Leser, den Fortschritt von ONLINE-DFS-AGENT zu verfolgen, der auf das Labyrinth in Abbildung 4.19 angewendet wird. Man erkennt leicht, dass der Agent im schlimmsten Fall jeden Link im Zustandsraum zweimal durchläuft. Für die Erkundung ist das optimal; für die Suche nach einem Ziel dagegen könnte der kompetitive Faktor beliebig schlecht werden, wenn er auf eine lange Exkursion geht, während sich ein Ziel direkt neben dem Ausgangszustand befindet. Eine Online-Variante der iterativen Vertiefung löst dieses Problem; in einer Umgebung, bei der es sich um einen einheitlichen Baum handelt, ist der kompetitive Faktor eines solchen Agenten eine kleine Konstante.

Aufgrund seiner Methode des Backtrackings funktioniert der ONLINE-DFS-AGENT nur in Zustandsräumen, in denen die Aktionen umkehrbar sind. Es gibt etwas komplexere Algorithmen, die auch in allgemeinen Zustandsräumen funktionieren, aber keiner dieser Algorithmen hat einen begrenzten kompetitiven Faktor.

4.5.3 Lokale Online-Suche

Wie die Tiefensuche hat die **Bergsteigen-Suche** die Eigenschaft der Lokalität in ihren Knotenexpandierungen. Weil nur ein aktueller Zustand im Speicher gehalten wird, handelt es sich bei der Bergsteigen-Suche eigentlich um einen Online-Suchalgorithmus! Leider ist er in seiner einfachsten Form nicht sehr praktisch, weil der Agent dabei auf lokalen Maxima sitzen bleibt, ohne weitergehen zu können. Darüber hinaus können keine zufälligen Neustarts verwendet werden, weil sich der Agent nicht selbst in einen neuen Zustand bringen kann.

Statt zufälliger Neustarts könnte man auch einen **zufälligen Spaziergang** (**Random Walk**) in Betracht ziehen, um die Umgebung zu erkunden. Ein derartiger Algorithmus wählt zufällig eine der verfügbaren Aktionen aus dem aktuellen Zustand aus; noch nicht ausprobierte Aktionen können bevorzugt ausgewählt werden. Es lässt sich leicht beweisen, dass ein zufälliges Weitergehen *irgendwann* ein Ziel findet oder seine Erkundung abschließt, vorausgesetzt, der Raum ist endlich.[15] Andererseits kann der Prozess sehr langsam sein. ▶ Abbildung 4.22 zeigt eine Umgebung, in der ein zufälliger Spaziergang exponentiell viele Schritte benötigt, um das Ziel zu finden, weil bei jedem Schritt ein Rückwärtsgehen doppelt so wahrscheinlich wie ein Vorwärtsschritt ist. Das Beispiel ist natürlich etwas konstruiert, aber es gibt auch viele Zustandsräume der realen Welt, deren Topologie diese Art von „Fallen“ für zufällige Spaziergänge verursacht.

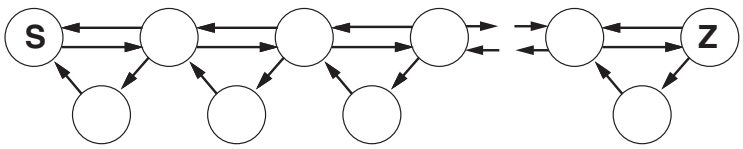


Abbildung 4.22: Eine Umgebung, in der ein zufälliger Spaziergang exponentiell viele Schritte benötigt, um das Ziel zu erreichen.

Die Erweiterung des Bergsteigens mit *Speicher* statt mit Zufälligkeit ist ein effektiverer Ansatz. Das grundlegende Konzept dabei ist, eine „aktuell beste Schätzung“ $H(s)$ der Kosten zu speichern, um das Ziel von jedem Zustand aus zu erreichen, der besucht wurde. $H(s)$ beginnt mit der heuristischen Schätzung $h(s)$ und wird aktualisiert, sobald der Agent Erfahrungen im Zustandsraum gesammelt hat. ▶ Abbildung 4.23 zeigt ein einfaches Beispiel in einem eindimensionalen Zustandsraum. In (a) scheint der Agent auf einem flachen lokalen Minimum in dem schattiert dargestellten Zustand stecken geblieben zu sein. Anstatt zu bleiben, wo er ist, sollte der Agent dem scheinbar besten Pfad zum Ziel folgen, basierend auf den aktuellen Kostenschätzun-

15 Zufällige Spaziergänge sind vollständig für unendliche eindimensionale und zweidimensionale Raster. In einem dreidimensionalen Raster liegt die Wahrscheinlichkeit, dass der Spaziergang irgendwann zum Ausgangspunkt zurückkehrt, nur bei etwa 0,3405 (Hughes, 1995.)

gen für seine Nachbarn. Die geschätzten Kosten zur Erreichung des Ziels über einen Nachbarn s' sind die Kosten, um zu s' zu gelangen, plus die geschätzten Kosten, um von dort aus zu einem Ziel zu gelangen – d.h. $c(s, a, s') + H(s')$. In dem Beispiel gibt es zwei Aktionen mit den geschätzten Kosten 1+9 und 1+2, sodass es am sinnvollsten erscheint, sich nach rechts zu bewegen. Jetzt ist deutlich, dass die Kostenschätzung von 2 für den grau schattierten Zustand allzu optimistisch war. Weil der beste Zug 1 kostet und zu einem Zustand geführt hat, der mindestens zwei Schritte von einem Ziel entfernt ist, muss der grau schattierte Zustand mindestens drei Schritte von einem Ziel entfernt sein; deshalb sollte sein H entsprechend aktualisiert werden, wie in Abbildung 4.23(b) gezeigt. Setzt man diesen Prozess fort, geht der Agent doppelt so oft vorwärts und rückwärts, aktualisiert dabei jedes Mal H und „bügelt" damit das lokale Minimum, bis er nach rechts entflieht.

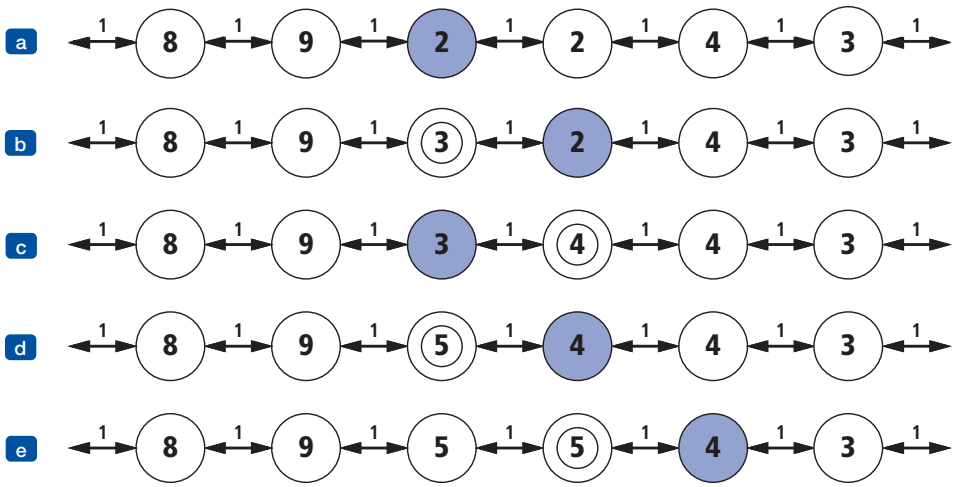


Abbildung 4.23: Fünf Iterationen von LRTA* für einen eindimensionalen Zustandsraum. Jeder Zustand ist mit $H(s)$ beschriftet, der aktuellen Kostenschätzung für die Erreichung eines Ziels, und an jeder Verbindung sind die jeweiligen Schrittkosten angegeben. Die schattiert dargestellten Zustände zeigen die Position des Agenten, die aktualisierten Werte jeder Iteration sind mit einem Kreis gekennzeichnet.

Ein Agent, der dieses Schema implementiert, das auch als **LRTA*** (Learning Real-Time A*, lernender Echtzeit-A*) bezeichnet wird, ist in ▶ Abbildung 4.24 gezeigt. Wie ONLINE-DFS-AGENT erzeugt er eine Karte der Umgebung unter Verwendung der Tabelle *result*. Der Agent aktualisiert die Kostenschätzung für den Zustand, den er gerade verlassen hat, und wählt dann den „scheinbar besten" Zug gemäß seinen aktuellen Kostenschätzungen aus. Ein wichtiges Detail ist, dass für Aktionen, die in einem Zustand s noch nicht ausprobiert wurden, immer angenommen wird, dass sie unmittelbar zum Ziel führen, und zwar mit den geringsten möglichen Kosten, d.h. $h(s)$. Dieser **Optimismus unter Unsicherheit** ermutigt den Agenten, neue, möglicherweise aussichtsreiche Pfade zu erkunden.

```
function LRTA*-AGENT(s') returns eine Aktion
  inputs: s', eine Wahrnehmung, die den aktuellen Zustand identifiziert
  persistent: result, eine durch Zustand und Aktion indizierte Tabelle,
              anfangs leer
              H, eine per Zustand indizierte Tabelle mit Kostenabschätzungen,
              anfangs leer
              s, a, vorhergehender Zustand und Aktion, anfangs null

  if GOAL-TEST(s') then return stop
  if s' neuer Zustand (nicht in H) then H[s'] ← h(s')
  if s ungleich null
      result[s, a] ← s'
      H[s] ← min_{b∈ACTION(s)} LRTA*-COST(s, b, result[s, b], H)
   a ← eine Aktion b in ACTIONS(s'), die
                        LRTA*-COST(s', b, result[s', b], H) minimiert
   s ← s'
   return a

function LRTA*-COST(s, a, s', H) returns eine Kostenschätzung
  if s' nicht definiert then return h(s)
  else return c(s, a, s') + H[s']
```

Abbildung 4.24: LRTA*-Agent wählt eine Aktion gemäß den Werten benachbarter Zustände aus, die aktualisiert werden, wenn der Agent in den Zustandsraum eintritt.

Ein LRTA*-Agent findet ein Ziel garantiert in einer endlichen, sicher erkundbaren Umgebung. Anders als A* ist er jedoch nicht vollständig für unendliche Zustandsräume – es gibt Fälle, in denen er auf unendliche Abwege geleitet wird. Er kann eine Umgebung von n Zuständen im ungünstigsten Fall in $O(n^2)$ Schritten erkunden, weist jedoch sehr viel häufiger eine bessere Leistung auf. Der LRTA*-Agent ist Mitglied einer großen Familie von Online-Agenten, die definiert werden können, indem die Regeln für die Auswahl der Aktion und die Aktualisierungsregeln auf unterschiedliche Weise spezifiziert werden. Wir werden diese Familie, die ursprünglich für stochastische Umgebungen entwickelt wurde, in *Kapitel 21* genauer betrachten.

4.5.4 Lernen bei der Online-Suche

Die anfängliche Unwissenheit von Online-Suchagenten bietet mehrere Möglichkeiten zum Lernen. Erstens lernen Agenten eine „Karte“ der Umgebung – genauer gesagt, die Wirkungen jeder Aktion in jedem Zustand –, indem sie einfach ihre Erfahrungen aufzeichnen. (Beachten Sie, dass die Annahme deterministischer Umgebungen bedeutet, dass eine Erfahrung für jede Aktion ausreichend ist.) Zweitens beschaffen sich lokale Suchagenten genauere Kostenschätzungen für jeden Zustand mithilfe lokaler Aktualisierungsregeln, wie in LRTA*. *Kapitel 21* zeigt, dass diese Aktualisierungen irgendwann für jeden Zustand zu *exakten* Werten konvergieren, vorausgesetzt, der Agent erkundet den Zustandsraum auf die richtige Weise. Nachdem exakte Werte bekannt sind, lassen sich optimale Entscheidungen treffen, indem einfach zum Nachfolger mit den niedrigsten Kosten gegangen wird – d.h., reines Bergsteigen ist dann eine optimale Strategie.

Wenn Sie unserem Vorschlag gefolgt sind, das Verhalten von ONLINE-DFS-AGENT in der Umgebung von Abbildung 4.19 nachzuvollziehen, haben Sie erkannt, dass der Agent nicht besonders clever ist. Nachdem der Agent beispielsweise erfahren hat, dass die

Aktion *Auf* von (1,1) nach (1,2) geht, hat er noch immer keine Ahnung, dass die Aktion *Ab* zurück zu (1,1) geht oder die Aktion *Auf* auch von (2,1) nach (2,2), von (2,2) nach (2,3) usw. Prinzipiell soll der Agent lernen, dass *Auf* die y-Koordinate erhöht, es sei denn, es steht eine Mauer im Weg, und dass *Ab* sie verringert usw. Damit dies passiert, sind zwei Dinge erforderlich: Erstens brauchen wir eine formale und explizit manipulierbare Darstellung für derartig allgemeine Regeln; bisher haben wir die Informationen innerhalb der Blackbox verborgen, die wir als die RESULT-Funktion bezeichnet haben. Teil III ist diesem Thema gewidmet. Zweitens brauchen wir Algorithmen, die geeignete allgemeine Regeln aus den spezifischen Beobachtungen des Agenten konstruieren können. Darum geht es in *Kapitel 18*.

Bibliografische und historische Hinweise

Lokale Suchtechniken haben eine lange Geschichte in der Mathematik und Informatik. Tatsächlich lässt sich die Newton-Raphsonsche Methode (Newton, 1671; Raphson, 1690) als eine sehr effiziente lokale Suchmethode für stetige Räume ansehen, in denen Gradientendaten verfügbar sind. Brent (1973) ist eine klassische Referenz für Optimierungsalgorithmen, die keine derartigen Informationen verlangen. Die Strahlsuche, die wir als lokalen Suchalgorithmus vorgestellt haben, war ursprünglich eine Variante mit begrenzter Breite der dynamischen Programmierung zur Spracherkennung im HARPY-System (Lowerre, 1976). Ein verwandter Algorithmus wird umfassend von Pearl (1984, *Kapitel 5*) analysiert.

Das Thema der lokalen Suche wurde Anfang der 1990er Jahre durch überraschend gute Ergebnisse für die Lösung großer Probleme unter Neben- oder Randbedingungen (Constraint-Satisfaction Problems) wie zum Beispiel *n*-Damen (Minton et al., 1992) und logisches Schließen (Selman et al., 1992) sowie durch die Integration der Zufälligkeit, mehrerer simultaner Suchen und anderer Verbesserungen wiederbelebt. Diese Renaissance dessen, was Christos Papadimitriou als die „New Age"-Algorithmen bezeichnete, führte auch zu einem gesteigerten Interesse in der theoretischen Informatik (Koutsoupias und Papadimitriou, 1992; Aldous und Vazirani, 1994). Im Bereich der Operationsforschung hat eine Variante des Bergsteigens, die sogenannte **Tabusuche**, große Popularität erzielt (Glover und Laguna, 1997). Dieser Algorithmus verwaltet eine Tabuliste von *k* zuvor besuchten Zuständen, die nicht erneut besucht werden können. Neben einer verbesserten Effizienz beim Durchsuchen von Graphen kann diese Liste dem Algorithmus erlauben, aus einem lokalen Minimum zu entkommen. Eine weitere sinnvolle Verbesserung des Bergsteigens ist der sogenannte STAGE-Algorithmus (Boyan und Moore, 1998). Die Idee dabei ist, die beim Bergsteigen mit zufälligem Neustart gefundenen lokalen Maxima zu verwenden, um sich ein Bild von der allgemeinen Form der Landschaft zu machen. Der Algorithmus passt eine glatte Oberfläche in die Menge der lokalen Maxima an und berechnet dann die globalen Maxima dieser Oberfläche analytisch. Dies wird zum neuen Neustartpunkt. Es wurde gezeigt, dass der Algorithmus in der Praxis auch für schwierige Probleme funktioniert. (Gomes et al., 1998) haben gezeigt, dass die Laufzeiten von systematischen Backtracking-Algorithmen häufig eine sogenannte **Heavy-tailed Verteilung** (**endlastige Verteilung**) aufweisen, d.h., dass die Wahrscheinlichkeit einer sehr langen Laufzeit höher ist, als man erwarten würde, wenn die Laufzeiten exponentiell verteilt wären. Wenn die Laufzeitverteilung endlastig ist, wird eine Lösung durch zufällige Neustarts im Mittel schneller gefunden als durch einen einzigen Lauf bis zur Fertigstellung.

Simulated Annealing wurde zuerst von Kirkpatrick et al. (1983) beschrieben, die direkt Anleihen beim **Metropolis-Algorithmus** aufnahmen (der für die Simulation komplexer Systeme in der Physik verwendet wird (Metropolis et al., 1953) und angeblich bei einer Dinnerparty in Los Alamos erfunden wurde). Heute ist Simulated Annealing ein eigenes Gebiet und jedes Jahr werden Hunderte von Aufsätzen dazu veröffentlicht.

Die Ermittlung optimaler Lösungen in stetigen Räumen ist Thema in mehreren Bereichen, unter anderem in der **Optimierungstheorie**, in der **Theorie der optimalen Steuerung** und in der **Variationsrechnung**. Die grundlegenden Techniken werden gut von Bishop (1995) erläutert; Press et al. (2007) behandeln ein breites Spektrum von Algorithmen und geben funktionsfähige Software an.

Wie Andrew Moore zeigt, haben sich Forscher für Such- und Optimierungsalgorithmen von den unterschiedlichsten Gebieten inspirieren lassen: Metallurgie (Simulated Annealing), Biologie (genetische Algorithmen), Wirtschaft (marktbasierte Algorithmen), Entomologie (Optimierung von Ameisenkolonien), Neurologie (neuronale Netze), Tierverhalten (Reinforcement Learning), Bergsteigen (Hillclimbing) und andere.

Die **lineare Programmierung** (LP) wurde erstmals vom russischen Mathematiker Leonid Kantorovich (1939) systematisch untersucht und war eine der ersten Anwendungen von Computern; der **Simplex-Algorithmus** (Dantzig, 1949) wird trotz einer im ungünstigsten Fall exponentiellen Komplexität immer noch verwendet. Karmarkar (1984) entwickelte die weit effizientere Familie von **Innere-Punkt-Verfahren**. Nesterov und Nemirovski (1994) haben gezeigt, dass sie polynomiale Komplexität für die allgemeinere Klasse konvexer Optimierungsprobleme besitzt. Ausgezeichnete Einführungen zur konvexen Optimierung werden von Ben-Tal und Nemirovski (2001) sowie von Boyd und Vandenberghe (2004) gegeben.

Die Arbeit von Sewell Wright (1931) zum Konzept einer **Fitness-Landschaft** war ein wichtiger Vorgänger der Entwicklung genetischer Algorithmen. In den 1950er Jahren verwendeten mehrere Statistiker, unter anderem Box (1957) und Friedman (1959) Evolutionstechniken für Optimierungsprobleme, aber erst Rechenberg (1965) führte **Evolutionsstrategien** ein, um Optimierungsprobleme bei Tragflächen zu lösen, wodurch der Ansatz an Popularität gewann. In den 1960er und 1970er Jahren pries John Holland (1975) genetische Algorithmen sowohl als praktisches Werkzeug als auch als Methode zur Erweiterung unseres Verständnisses zur (biologischen oder anderweitigen) Adaption an (Holland, 1995). Die Bewegung **Artificial Life** (Langton, 1995) führte diese Idee noch einen Schritt weiter und betrachtete die Produkte genetischer Algorithmen als *Organismen* und nicht mehr als Lösungen für Probleme. Arbeiten zu diesem Thema von Hinton und Nowlan (1987) sowie von Ackley und Littman (1991) haben viel zur Erklärung der Bedeutung des Baldwin-Effektes beigetragen. Für allgemeine Hintergrundinformationen zur Evolution können wir Smith und Szathmáry (1999), Ridley (2004) und Carroll (2007) empfehlen.

Die meisten Vergleiche genetischer Algorithmen mit anderen Ansätzen (insbesondere stochastischen Bergsteigens) haben gezeigt, dass die genetischen Algorithmen langsamer konvergieren (O'Reilly und Oppacher, 1994; Mitchell et al. 1996; Juels und Wattenberg, 1996; Baluja, 1997). Diese Feststellungen waren bei den Anhängern genetischer Algorithmen nicht allgemein beliebt, aber neuere Versuche dieser Gruppe, die populationsbasierte Suche als Annäherungsform des Bayes'schen Lernens zu verstehen (siehe *Kapitel 20*), könnten helfen, die Lücke zwischen dem Fachgebiet und seinen Kritikern zu schließen (Pelikan et al., 1999). Die Theorie **quadratischer dynamischer Systeme** könnte auch die Leistung von genetischen Algorithmen erklären (Rabani et al., 1998). Beispiele für genetische Algorithmen finden Sie als Anwendung für den Entwurf von

Antennen bei Lohn et al. (2001) sowie als Anwendung für computerunterstützten Entwurf bei Renner und Ekart (2003).

Der Bereich der **genetischen Programmierung** ist eng mit den genetischen Algorithmen verwandt. Der prinzipielle Unterschied besteht darin, dass die Darstellungen, die mutiert und kombiniert werden, Programme und keine Bit-Zeichenfolgen sind. Die Programme werden in Form von Ausdrucksbäumen dargestellt; die Ausdrücke können in einer Standardsprache wie Lisp formuliert oder speziell für die Darstellung von Schaltungen, Roboter-Controllern usw. konzipiert sein. Beim Crossover werden Unterbäume statt Unterzeichenfolgen kombiniert. Diese Form der Mutation garantiert, dass die Nachkommen wohlgeformte Ausdrücke sind, was nicht der Fall wäre, würden die Programme wie Zeichenfolgen manipuliert.

Das Interesse an der genetischen Programmierung wurde durch die Arbeiten von John Koza (Koza, 1992, 1994) beflügelt, geht aber mindestens auf frühere Experimente mit Maschinencode von Friedberg (1958) und endlicher Automaten von Fogel et al. (1966) zurück. Wie bei den genetischen Algorithmen gibt es Debatten im Hinblick auf die Effektivität der Technik. Koza et al. (1999) beschreiben Experimente zum automatisierten Schaltungsentwurf mithilfe genetischer Programmierung.

Die Magazine *Evolutionary Computation* und *IEEE Transactions on Evolutionary Computation* beschreiben genetische Algorithmen und genetische Programmierung; man findet auch Artikel zu diesem Thema in *Complex Systems*, *Adaptive Bahavior* und *Artificial Life*. Als Hauptkonferenz gilt die *Genetic and Evolutionary Computation Conference* (GECCO). Gute Überblicksartikel zu genetischen Algorithmen sind bei Mitchell (1996), Fogel (2000), Langdon und Poli (2002) sowie im kostenlosen Online-Buch von Poli et al. (2008) zu finden.

Die Unvorhersehbarkeit und teilweise Beobachtbarkeit von realen Umgebungen hat man frühzeitig in Roboter-Projekten erkannt, die mit Planungstechniken arbeiteten. Dazu gehören Shakey (Fikes et al., 1972) und FREDDY (Michie, 1974). Mehr Aufmerksamkeit wurde diesen Problemen nach der Veröffentlichung des einflussreichen Artikels *Planning and Acting* von McDermott (1978a) gewidmet.

Die erste Arbeit, die sich explizit auf AND-OR-Bäume stützt, scheint das in *Kapitel 1* erwähnte Programm SAINT von Slagle für die symbolische Integration zu sein. Amarel (1967) hat die Idee auf das Beweisen von Theoremen der Aussagenlogik angewandt (mehr zu diesem Thema in *Kapitel 7*) und einen Suchalgorithmus ähnlich dem AND-OR-GRAPH-SEARCH eingeführt. Nilsson (1971) hat den Algorithmus weiterentwickelt und formalisiert. Außerdem beschrieb er den Algorithmus AO*, der – wie aus dem Namen hervorgeht – optimale Lösungen bei einer gegebenen zulässigen Heuristik sucht. AO* wurde von Martelli und Montanari (1973) analysiert und verbessert. Bei AO* handelt es sich um einen Top-down-Algorithmus; eine Bottom-up-Verallgemeinerung von A* ist A*LD (Felzenszwalb und McAllester, 2007), was für A* Lightest Derivation (Hellste Abweichung) steht. Das Interesse für die AND-OR-Suche ist in den letzten Jahren neu erwacht, und zwar mit neuen Algorithmen für das Suchen zyklischer Lösungen (Jimenez und Torras, 2000; Hansen und Zilberstein, 2001) und neuen Techniken, die durch die dynamische Programmierung inspiriert wurden (Bonet und Geffner, 2005).

Die Idee, teilweise beobachtbare Probleme in Belief-States-Probleme zu überführen, stammt von Astrom (1965) für den wesentlich komplexeren Fall probabilistischer Unsicherheit (siehe *Kapitel 17*). Erdmann und Mason (1988) untersuchten das Problem der sensorlosen Robotermanipulation mithilfe einer stetigen Form der Belief-States-Suche.

Sie zeigten, dass es mit einer gut abgestimmten Folge von Schwenkaktionen möglich ist, ein Teil auf einem Tisch aus einer willkürlichen Anfangsposition heraus zu positionieren. Praktischere Methoden, die auf einer Folge von genau ausgerichteten diagonalen Barrieren auf einem Förderband beruhen, verwenden die gleichen algorithmischen Erkenntnisse (Wiegley et al., 1996).

Das Belief-States-Konzept wurde im Kontext der sensorlosen und teilweise beobachtbaren Suchprobleme von Genesereth und Nourbakhsh (1993) neu erfunden. In der Gemeinde der auf Logik basierenden Planung (Goldman und Boddy, 1996; Smith und Weld, 1998) wurden zusätzliche Arbeiten zu sensorlosen Problemen durchgeführt. Diese Arbeiten unterstreichen die prägnanten Darstellungen für Belief States, wie sie in *Kapitel 11* erläutert werden. Bonet und Geffner (2000) haben die ersten wirksamen Heuristiken für die Belief-States-Suche eingeführt; verfeinert wurden sie von Bryce et al. (2006). Der inkrementelle Ansatz für die Belief-States-Suche, bei dem Lösungen für Teilmengen von Zuständen innerhalb jedes Belief State inkrementell konstruiert werden, wurde in der Planungsliteratur von Kurien et al. (2002) untersucht. Von Russell und Wolfe (2005) wurden mehrere neue inkrementelle Algorithmen für nichtdeterministische, partiell beobachtbare Probleme eingeführt. *Kapitel 17* nennt weiterführende Quellen für die Planung in stochastischen, partiell beobachtbaren Umgebungen.

Seit mehreren Jahrhunderten besteht großes Interesse an den Algorithmen zur Erkundung unbekannter Zustandsräume. Die Tiefensuche in einem Labyrinth kann implementiert werden, indem man die linke Hand an der Wand belässt; Schleifen können vermieden werden, indem man jede Kreuzung markiert. Die Tiefensuche schlägt fehl, wenn nicht umkehrbare Aktionen vorliegen; das allgemeinere Problem der Erkundung von **Eulerschen Graphen** (d.h. Graphen, bei denen jeder Knoten eine gleiche Anzahl eingehender und ausgehender Kanten aufweist) wurde durch einen von Hierholzer (1873) entwickelten Algorithmus gelöst. Die erste eingehende algorithmische Untersuchung des Explorationsproblems für beliebige Graphen nahmen Deng und Papadimitriou (1990) vor. Sie entwickelten einen völlig allgemeinen Algorithmus, zeigten aber, dass für die Erkundung eines allgemeinen Graphen kein begrenzter kompetitiver Faktor möglich ist. Papadimitriou und Yannakakis (1991) untersuchten die Frage, Pfade zu einem Ziel in geometrischen Pfadplanungsumgebungen zu finden (wobei alle Aktionen umkehrbar sind). Sie zeigten, dass ein kleiner kompetitiver Faktor für quadratische Hindernisse möglich ist, dass jedoch mit allgemeinen rechteckigen Hindernissen kein begrenzter Faktor erzielt werden kann (siehe Abbildung 4.20).

Der LRTA*-Algorithmus wurde von Korf (1990) als Teil einer Untersuchung zur **Echtzeitsuche** für Umgebungen betrachtet, in denen der Agent agieren muss, nachdem er eine vorgegebene Zeit lang gesucht hat (eine gebräuchliche Situation in Spielen zwischen zwei Mitspielern). LRTA* ist tatsächlich ein Sonderfall von Algorithmen zu verstärktem Lernen für stochastische Umgebungen (Barto et al., 1995). Seine Strategie des Optimismus unter Unsicherheit – immer den nächstliegenden nicht besuchten Zustand ansteuern – kann in einem Explorationsmuster resultieren, das im uninformierten Fall weniger effizient ist als eine einfache Tiefensuche (Koenig, 2000). Dasgupta et al. (1994) zeigen, dass eine iterativ vertiefende Online-Suche optimal effizient für die Suche nach einem Ziel in einem einheitlichen Baum ohne heuristische Informationen ist. Mehrere informierte Varianten des LRTA*-Themas wurden mit anderen Such- und Aktualisierungsmethoden innerhalb des bekannten Teils des Graphen entwickelt (Pemberton und Korf, 1992). Momentan weiß man noch nicht zufriedenstellend, wie man Ziele mit optimaler Effizienz findet, wenn man heuristische Informationen benutzt.

Zusammenfassung

Dieses Kapitel hat sich mit Suchalgorithmen für Probleme beschäftigt, die über den „klassischen" Fall einer Suche nach dem kürzesten Pfad zu einem Ziel in einer beobachtbaren, deterministischen, diskreten Umgebung hinausgehen.

- Methoden der *lokalen Suche*, wie beispielsweise das **Bergsteigen**, arbeiten mit vollständigen Zustandsformulierungen und behalten nur einige wenige Knoten im Speicher. Es wurden mehrere stochastische Algorithmen entwickelt, unter anderem das **Simulated Annealing**, das optimale Lösungen zurückgibt, wenn es einen geeigneten Zeitplan für das „Abkühlen" erhält.
- Viele lokale Suchmethoden lassen sich auch auf Probleme in stetigen Räumen anwenden. Probleme der **linearen Programmierung** und **konvexen Optimierung** gehorchen bestimmten Einschränkungen hinsichtlich der Gestalt des Zustandsraumes und dem Wesen der Zielfunktion. Zudem erlauben sie Algorithmen mit polynomieller Zeit, die in der Praxis oftmals äußerst effizient arbeiten.
- Ein **genetischer Algorithmus** ist eine stochastische Bergsteigen-Suche, wobei eine große Population an Zuständen verwaltet wird. Neue Zustände werden durch **Mutation** und **Kreuzung** (**Crossover**) erzeugt, wobei Zustandspaare aus der Population kombiniert werden.
- In **nichtdeterministischen Umgebungen** können Agenten eine AND-OR-Suche anwenden, um Kontingenzpläne zu erzeugen, die das Ziel unabhängig davon erreichen, welche Ergebnisse während der Ausführung auftreten.
- Ist eine Umgebung partiell beobachtbar, repräsentiert der Belief State die Menge der möglichen Zustände, die der Agent einnehmen kann.
- Standardsuchalgorithmen können direkt auf den Belief-States-Raum angewandt werden, um **sensorlose Probleme** zu lösen, und die Belief-States-AND-OR-Suche kann im Allgemeinen partiell beobachtbare Probleme lösen. Inkrementelle Algorithmen, die Lösungen zustandsweise innerhalb eines Belief State konstruieren, arbeiten oftmals effizienter.
- **Explorationsprobleme** treten auf, wenn der Agent keine Ahnung von den Zuständen und Aktionen seiner Umgebung hat. Für sicher erkundbare Umgebungen können **Online-Suchagenten** eine Karte erstellen und ein Ziel finden, falls ein solches existiert. Die Aktualisierung von heuristischen Schätzungen aus der Erfahrung bietet eine effektive Methode, um lokalen Minima zu entgehen.

Übungen zu Kapitel 4

Lösungshinweise

1 Nennen Sie den Namen des Algorithmus, der aus den folgenden Sonderfällen resultiert:

a. Lokale Strahlsuche mit $k = 1$

b. Lokale Strahlsuche mit genau einem Anfangszustand und ohne Beschränkung der Anzahl beibehaltener Zustände

c. Simulated Annealing mit $T = 0$ zu jedem beliebigen Zeitpunkt (und Weglassen des Abschlusstests)

d. Simulated Annealing mit $T = \infty$ zu jedem beliebigen Zeitpunkt

e. Genetischer Algorithmus mit Populationsgröße $N = 1$

2 Übung 3.16 betrachtet das Problem, Eisenbahnstrecken zu bauen, unter der Annahme, dass die Teile ohne Spiel zusammenpassen. Nehmen Sie nun das reale Problem, in dem die Gleisstücke nicht genau passen, aber bis zu 10 Grad zu jeder Seite der „korrekten" Ausrichtung gedreht werden dürfen. Erläutern Sie, wie das Problem formuliert werden kann, damit es sich durch Simulated Annealing lösen lässt.

3 Erzeugen Sie eine große Anzahl Instanzen der 8-Puzzle- und 8-Damen-Probleme und lösen Sie sie (falls möglich) durch Bergsteigen (mit den Varianten des steilsten Anstieges und der Bestensuche), Bergsteigen mit zufälligem Neustart und Simulated Annealing. Messen Sie die Suchkosten sowie den Prozentsatz gelöster Probleme und stellen Sie sie in einem Diagramm über den optimalen Lösungskosten dar. Kommentieren Sie Ihre Ergebnisse.

4 Der AND-OR-GRAPH-SEARCH-Algorithmus in Abbildung 4.11 prüft nur im Pfad von der Wurzel zum aktuellen Zustand auf wiederholte Zustände. Nehmen Sie zusätzlich an, dass der Algorithmus jeden besuchten Zustand speichert und gegen diese Liste überprüft. (Ein Beispiel finden Sie bei BREADTH-FIRST-SEARCH in Abbildung 3.11.) Ermitteln Sie, welche Informationen zu speichern sind und wie der Algorithmus auf diese Informationen zugreifen sollte, wenn er einen wiederholten Zustand findet. (*Hinweis:* Sie müssen mindestens zwischen Zuständen, für die vorher ein erfolgreicher Teilplan konstruiert wurde, und Zuständen, für die kein Teilplan gefunden werden konnte, unterscheiden.) Erläutern Sie, wie Sie Bezeichner gemäß *Abschnitt 4.3.3* verwenden, um mehrere Kopien von Teilplänen zu vermeiden.

5 Erläutern Sie genau, wie der AND-OR-GRAPH-SEARCH-Algorithmus zu modifizieren ist, um einen zyklischen Plan zu erzeugen, wenn kein azyklischer Plan existiert. Hier haben Sie es mit drei Fragen zu tun: Bezeichnen der Planschritte, sodass ein zyklischer Plan auf einen früheren Teil des Planes zurückverweisen kann, Modifizieren von OR-SEARCH, damit der Algorithmus weiterhin nach azyklischen Plänen sucht, nachdem er einen zyklischen Plan gefunden hat, und Erweitern der Plandarstellung, um anzugeben, ob ein Plan zyklisch ist. Zeigen Sie, wie Ihr Algorithmus in (a) der unsicheren Staubsaugerwelt und (b) der unsicheren, erratischen Staubsaugerwelt funktioniert. Es empfiehlt sich, dass Sie Ihre Ergebnisse mit einer Computerimplementierung überprüfen.

6 In *Abschnitt 4.4.1* haben wir Belief States eingeführt, um sensorlose Suchprobleme zu lösen. Eine Aktionsfolge löst ein sensorloses Problem, wenn sie jeden physischen Zustand im anfänglichen Belief State b auf einen Zielzustand abbildet. Nehmen Sie an, der Agent kennt mit $h*(s)$ die wahren optimalen Kosten für das Lösen des physischen Zustandes s im vollständig beobachtbaren Problem für jeden Zustand s in b. Suchen Sie eine zulässige Heuristik $h(b)$ für das sensorlose Problem in Form dieser Kosten und beweisen Sie ihre Zulässigkeit. Erörtern Sie die Genauigkeit dieser Heuristik für das sensorlose Staubsaugerproblem von Abbildung 4.14. Wie gut verhält sich der Algorithmus A*?

7 Diese Übung untersucht Teilmengen-Obermengen-Beziehungen zwischen Belief States in sensorlosen oder teilweise beobachtbaren Umgebungen.

a. Beweisen Sie: Wenn eine Aktionsfolge eine Lösung für einen Belief State b ist, stellt sie auch eine Lösung für jede beliebige Teilmenge von b dar. Lässt sich etwas zu Obermengen von b sagen?

b. Erläutern Sie im Detail, wie Sie die Graphensuche für sensorlose Probleme modifizieren, um von Ihren Antworten in (a) zu profitieren.

c. Erläutern Sie im Detail, wie Sie die AND-OR-Suche für teilweise beobachtbare Probleme über die in (b) beschriebenen Änderungen hinaus modifizieren.

8 In *Abschnitt 4.4.1* wurde angenommen, dass eine bestimmte Aktion die gleichen Kosten erzeugt, wenn sie in einem beliebigen physischen Zustand innerhalb eines bestimmten Belief State ausgeführt wird. (Dies führt zu einem Belief-States-Problem mit wohldefinierten Schrittkosten.) Was passiert, wenn die Annahme nicht gilt? Ist das Konzept der Optimalität in diesem Kontext immer noch sinnvoll oder verlangt es nach einer Modifikation? Betrachten Sie auch die verschiedenen möglichen Definitionen für die „Kosten" der Ausführung einer Aktion in einem Belief State – zum Beispiel könnten wir das *Minimum* der physischen Kosten nehmen, das *Maximum* oder auch ein Kosten*intervall*, bei dem die untere Grenze die minimalen Kosten und die obere Grenze die maximalen Kosten darstellt, oder man nimmt einfach die Menge aller möglichen Kosten für diese Aktion. Erläutern Sie jeweils, ob A* (bei Bedarf mit Modifikationen) optimale Lösungen zurückgeben kann.

9 Betrachten Sie die sensorlose Version der erratischen Staubsaugerwelt. Zeichnen Sie den Belief-States-Bereich, der vom anfänglichen Belief State $\{1, 3, 5, 7\}$ erreichbar ist, und erläutern Sie, warum das Problem nicht lösbar ist.

10 Das Navigationsproblem in Übung 3.7 lässt sich wie folgt in eine Umgebung überführen:

– Die Wahrnehmung wird zu einer Liste der Positionen der sichtbaren Eckpunkte *relativ zum Agenten*. Die Position des Roboters ist *nicht* in die Wahrnehmung eingeschlossen! Der Roboter muss seine eigene Position aus der Karte lernen; momentan können Sie annehmen, dass jede Position eine andere „Ansicht" besitzt.

– Jede Aktion wird zu einem Vektor, der einen geraden, zu folgenden Pfad beschreibt. Ist der Pfad hindernisfrei, verläuft die Aktion erfolgreich; andernfalls hält der Roboter an dem Punkt an, wo sein Pfad zuerst ein Hindernis kreuzt. Wenn der Agent einen Bewegungsvektor von null zurückgibt und sich beim Ziel befindet (das feststeht und bekannt ist), teleportiert die Umgebung den Agenten zu einer zufälligen Position (nicht innerhalb eines Hindernisses).

– Die Leistungsbewertung belastet den Agenten mit 1 Punkt für jede zurückgelegte Distanzeinheit und belohnt ihn mit jeweils 1000 Punkten, wenn das Ziel erreicht ist.

a. Implementieren Sie diese Umgebung und einen dazugehörenden Problemlösungsagenten. Nach jeder Teleportation muss der Agent ein neues Problem formulieren, wozu die Entdeckung seiner aktuellen Position gehört.

b. Dokumentieren Sie die Leistung Ihres Agenten (indem Sie den Agenten bei seinen Bewegungen zweckmäßige Kommentare generieren lassen) und erstellen Sie einen Leistungsbericht über 100 Episoden.

c. Modifizieren Sie die Umgebung, sodass der Agent in 30% der Zeit bei einem nicht beabsichtigten Zielort landet (der zufällig aus den anderen sichtbaren Eckpunkten – falls vorhanden – gewählt wird; andernfalls findet keinerlei Bewegung statt). Dies ist ein grobes Modell der Bewegungsfehler eines realen Roboters. Modifizieren Sie den Agenten, damit er bei Erkennen eines derartigen Fehlers ermittelt, wo er sich befindet, und dann einen Plan konstruiert, um dorthin zurückzugehen, wo er war, und den alten Plan wieder aufnimmt. Denken Sie daran, dass es manchmal ebenfalls schief gehen kann, wenn der Agent an die alte Position zurückgeht! Zeigen Sie ein Beispiel, wie der Agent erfolgreich zwei aufeinanderfolgende Bewegungsfehler überwindet und trotzdem das Ziel erreicht.

d. Probieren Sie nun zwei andere Schemas für die Wiederherstellung nach einem Fehler aus: (1) Wenden zum nächsten Eckpunkt auf der ursprünglichen Route und (2) Neuplanen einer Route zum Ziel von der neuen Position aus. Vergleichen Sie die Leistung der drei Wiederherstellungsschemas. Würde sich die Einbindung der Suchkosten auf den Vergleich auswirken?

e. Nehmen Sie nun an, dass es Positionen gibt, von denen aus die Ansicht identisch ist. (Zum Beispiel könnte man annehmen, dass die Welt ein Raster mit quadratischen Hindernissen ist.) Mit welchem Problem wird der Agent nun konfrontiert? Wie sehen Lösungen aus?

11 Ein Agent befinde sich in einer 3×3 großen Labyrinthumgebung, wie in Abbildung 4.19 gezeigt. Der Agent kennt seine Ausgangsposition mit (3, 3), das Ziel mit (1, 1) sowie die vier Aktionen *Auf*, *Ab*, *Links*, *Rechts*, die ihre übliche Wirkung haben, es sei denn, sie werden durch eine Mauer blockiert. Der Agent weiß *nicht*, wo sich die internen Wände befinden. In jedem beliebigen Zustand nimmt der Agent die Menge der erlaubten Aktionen wahr; er kann auch erkennen, ob der Zustand ein bereits besuchter Zustand oder ein neuer Zustand ist.

a. Erklären Sie, wie dieses Online-Suchproblem als Offline-Suche im Belief-States-Raum betrachtet werden kann, wobei der anfängliche Belief State alle möglichen Umgebungskonfigurationen beinhaltet. Wie groß ist der anfängliche Belief State? Wie groß ist der Raum der Belief States?

b. Wie viele verschiedene Wahrnehmungen sind im Ausgangszustand möglich?

c. Beschreiben Sie die ersten Verzweigungen eines Kontingenzplanes für dieses Problem. Wie groß ist der vollständige Plan in etwa?

d. Beachten Sie, dass dieser Kontingenzplan eine Lösung für *jede mögliche Umgebung* ist, die der vorgegebenen Beschreibung entspricht. Eine Verzahnung von Suche und Ausführung ist deshalb nicht unbedingt erforderlich, selbst wenn es sich um unbekannte Umgebungen handelt.

12 In dieser Übung betrachten wir das Bergsteigen im Kontext der Roboternavigation und der Verwendung der in Abbildung 3.31 gezeigten Umgebung als Beispiel.

a. Wiederholen Sie Übung 4.10 unter Verwendung des Bergsteigens. Bleibt Ihr Agent an einem lokalen Minimum hängen? Ist es *möglich*, dass er aufgrund konvexer Hindernisse hängen bleibt?

b. Konstruieren Sie eine nicht konvexe polygonale Umgebung, in der der Agent steckenbleibt.

c. Ändern Sie den Bergsteigeralgorithmus so ab, dass er statt einer Suche mit der Tiefe 1 eine Suche mit der Tiefe k ausführt, um zu entscheiden, wo er als Nächstes hingehen soll. Er soll den besten Pfad mit k Schritten ermitteln, einen Schritt ausführen und dann den Prozess wiederholen.

d. Gibt es ein k, für das der neue Algorithmus garantiert aus einem lokalen Minimum entfliehen kann?

e. Erklären Sie, wie es LRTA* einem Agenten ermöglicht, in diesem Fall aus einem lokalen Minimum zu entfliehen.

13 Vergleichen Sie die Zeitkomplexität von LRTA* mit seiner Speicherkomplexität.

Adversariale Suche

5

ÜBERBLICK

In diesem Kapitel betrachten wir die Probleme, die auftreten, wenn wir versuchen, in einer Welt vorauszuplanen, in der andere Agenten gegen uns planen.

5.1 Spiele

Kapitel 2 hat **Multiagenten-Umgebungen** vorgestellt, in denen jeder Agent die Aktionen anderer Agenten und ihren Einfluss auf das eigene Wohlergehen berücksichtigen muss. In *Kapitel 4* wurde erläutert, dass die Unvorhersehbarkeit dieser anderen Agenten viele mögliche **Eventualitäten** in den Problemlösungsprozess des Agenten einbringen kann. In diesem Kapitel geht es nun um **konkurrierende** (kompetitive) Umgebungen, in denen die Ziele der Agenten in Konflikt zueinander stehen, was zu **adversarialen Suchproblemen** – häufig auch als **Spiele** bezeichnet – führt.

Die mathematische **Spieltheorie**, ein Zweig der Wirtschaftswissenschaften, betrachtet alle Multiagenten-Umgebungen als Spiel, sofern der Einfluss jedes Agenten auf die anderen Agenten „signifikant" ist, und zwar unabhängig davon, ob die Agenten kooperativ oder konkurrierend handeln.[1] In der künstlichen Intelligenz haben die gebräuchlichsten Spiele in der Regel eine spezielle Natur – die Spieltheoretiker sprechen von deterministischen Zwei-Personen-**Nullsummenspielen** mit **vollständiger Information**, bei denen zwei Spieler abwechselnd agieren (wie zum Beispiel Schach). In unserer Terminologie sind das deterministische, vollständig beobachtbare Umgebungen, in denen zwei Agenten abwechselnd handeln und in denen die Nutzenwerte am Ende des Spieles immer gleich groß und entgegengesetzt sind. Gewinnt ein Spieler beispielsweise ein Schachspiel (+1), muss der andere Spieler es notwendigerweise verlieren (-1). Genau dieser Gegensatz zwischen den Nutzenfunktionen der Agenten macht die Situation adversarial.

Seit man von der menschlichen Kultur sprechen kann, haben Spiele die intellektuellen Fähigkeiten des Menschen herausgefordert – manchmal in einem alarmierenden Ausmaß. Die abstrakte Natur macht Spiele zu einem interessanten Studienobjekt für KI-Forscher. Der Zustand eines Spieles ist einfach darzustellen und Agenten sind normalerweise auf eine kleine Zahl von Aktionen begrenzt, deren Ergebnisse durch genaue Regeln definiert sind. Physische Spiele, wie beispielsweise Kricket oder Eishockey, haben sehr viel kompliziertere Beschreibungen, einen sehr viel größeren Bereich möglicher Aktionen und recht ungenaue Regeln, die definieren, welche Aktionen erlaubt sind. Mit Ausnahme des Roboterfußballs konnten diese physischen Spiele in der KI-Gemeinde kein größeres Interesse erregen.

Spiele sind im Unterschied zu den meisten anderen in *Kapitel 3* betrachteten Spielzeugproblemen interessant, *weil* sie schwierig zu lösen sind. Schach beispielsweise hat einen durchschnittlichen Verzweigungsfaktor von 35 und Spiele benötigen oft 50 Züge pro Spieler, sodass der Suchbaum etwa 35^{100} oder 10^{154} Knoten hat (obwohl der Suchgraph „nur" etwa 10^{40} verschiedene Knoten aufweist). Spiele fordern also wie die reale Welt die Fähigkeit, *irgendeine* Entscheidung zu treffen, selbst wenn die Berechnung einer *optimalen* Entscheidung nicht möglich ist. Außerdem bestrafen Spiele Ineffizienzen schwer. Während eine Implementierung einer halb so effizienten A*-Suche für eine vollständige Ausführung einfach doppelt so viel kostet, wird ein Schachprogramm, das in seiner verfügbaren Zeit nur halb so effizient ist, unter sonst gleichen Bedingungen

1 Umgebungen mit sehr vielen Agenten werden am besten als **Wirtschaftssysteme** statt als Spiele betrachtet.

wahrscheinlich in Grund und Boden gespielt. Die Spielforschung hat deshalb zu einer Anzahl interessanter Ideen geführt, wie man die Zeit bestmöglich nutzt.

Wir beginnen mit einer Definition des optimalen Zuges und einem Algorithmus, mit dem dieser zu finden ist. Anschließend betrachten wir Techniken für die Auswahl eines guten Zuges, wenn die Zeit begrenzt ist. Durch die **Kürzung** können wir Teile des Suchbaumes ignorieren, die für die endgültige Auswahl keine Rolle spielen. Heuristische Auswertungsfunktionen erlauben es uns, den echten Nutzen eines Zustandes abzuschätzen, ohne eine vollständige Suche durchführen zu müssen. In *Abschnitt 5.5* werden Spiele wie etwa Backgammon beschrieben, die eine Zufallskomponente enthalten. Außerdem beschreiben wir Bridge, das Elemente einer **unvollständigen Information** beinhaltet, weil für keinen Spieler alle Karten einsehbar sind. Schließlich betrachten wir noch, wie moderne Spielprogramme im Verhältnis zum menschlichen Gegner abschneiden, und zeigen die Richtung für zukünftige Entwicklungen auf.

Zuerst betrachten wir Spiele mit zwei Spielern, MAX und MIN genannt – aus Gründen, die gleich offensichtlich werden. MAX zieht zuerst und dann wechseln sich die Spieler ab, bis das Spiel zu Ende ist. Am Spielende erhält der gewinnende Spieler Punkte, der Verlierer erhält eine Bestrafung. Ein Spiel kann formal als eine Art Suchproblem mit den folgenden Komponenten definiert werden:

- S_0: der **Ausgangszustand**, der angibt, wie das Spiel beim Start eingerichtet ist
- PLAYER(s): definiert, welcher Spieler in einem Zustand am Zug ist
- ACTIONS(s): gibt die Menge der zulässigen Züge in einem Zustand zurück
- RESULT(s, a): das **Übergangsmodell**, das das Ergebnis eines Zuges definiert
- TERMINAL-TEST(s): ein **Endetest**, der *true* liefert, wenn das Spiel vorüber ist, und andernfalls *false* zurückgibt. Die Zustände, in denen das Spiel geendet hat, werden als **Endzustände** bezeichnet.
- UTILITY(s, p): Eine **Nutzenfunktion** (auch Zielfunktion oder Auszahlungsfunktion genannt) definiert den endgültigen numerischen Wert für ein Spiel, das in einem Endzustand s für einen Spieler p endet. Beim Schach ist das Ergebnis ein Gewinn, ein Verlust oder ein Remis mit den Werten +1, 0 bzw. 1/2. Bei manchen Spielen sind die möglichen Ergebnisse breiter gefächert. So reichen die möglichen Auszahlungen beim Backgammon von 0 bis +192. Ein **Nullsummenspiel** ist (verwirrenderweise) als ein Spiel definiert, bei dem die gesamte Auszahlung an alle Spieler gleich der für jede Instanz des Spieles ist. Schach ist ein Nullsummenspiel, weil jedes Spiel eine Auszahlung von 0 + 1, 1 + 0 oder 1/2 + 1/2 hat. Treffender wäre der Begriff „konstante Summe“ gewesen, doch ist Nullsumme ein eingeführter Begriff und zudem sinnvoll, wenn man annimmt, dass jeder Spieler ein „Startgeld“ von 1/2 zahlen muss.

Der Ausgangszustand und die Funktionen ACTIONS und RESULT definieren den **Spielbaum** für das Spiel – ein Baum, in dem die Knoten Spielzustände und die Kanten Züge darstellen. ▶ Abbildung 5.1 zeigt einen Teil des Spielbaumes für Tic Tac Toe (Kreise und Kreuze). Im Ausgangszustand hat MAX neun mögliche Züge. Im Spiel setzen abwechselnd MAX und MIN ein X oder ein O, bis wir Blattknoten erreichen, die Endzuständen entsprechen, sodass ein Spieler drei seiner Zeilen in einer Reihe hat oder alle Quadrate gefüllt sind. Die Zahl in jedem Blattknoten gibt den Ergebniswert aus der Perspektive von MAX an; hohe Werte sind gut für MAX und schlecht für MIN (daher haben die Spieler auch ihre Namen).

Der Suchbaum für Tic Tac Toe ist relativ klein – weniger als 9! = 362.880 Endknoten. Beim Schach sind es dagegen über 10^{40} Knoten, sodass man sich den Suchbaum am besten als theoretisches Konstrukt vorstellt, das sich in der Praxis nicht umsetzen lässt. Doch unabhängig von der Größe des Suchbaumes hat MAX die Aufgabe, einen guten Zug zu suchen. Wir verwenden hier den Begriff **Suchbaum** für einen Baum, der dem vollständigen Spielbaum überlagert ist, und untersuchen genügend Knoten, damit ein Spieler ermitteln kann, welchen Zug er ausführen soll.

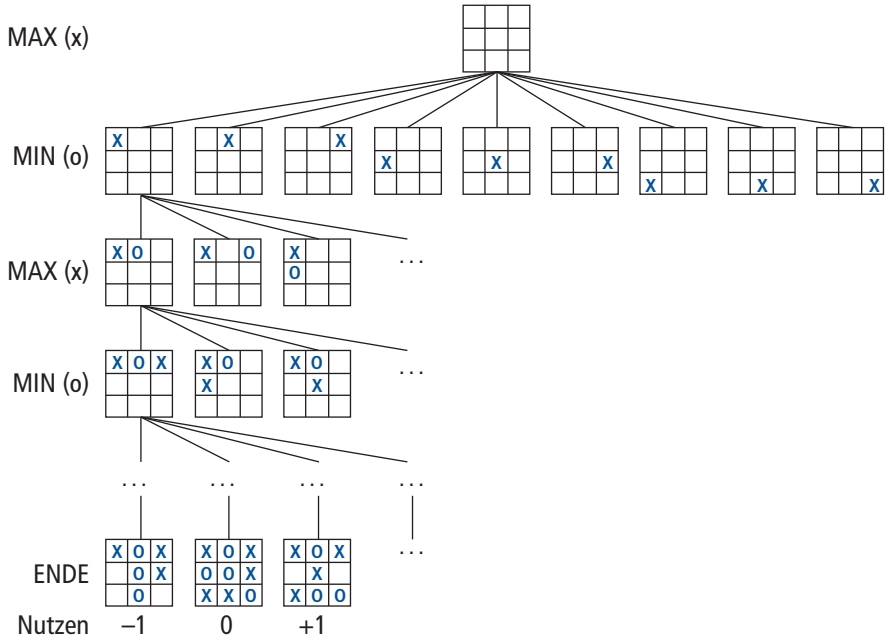


Abbildung 5.1: Ein (partieller) Suchbaum für das Spiel Tic Tac Toe. Der oberste Knoten ist der Ausgangszustand und MAX zieht zuerst und platziert dabei ein X in ein leeres Quadrat. Wir zeigen einen Teil des Suchbaumes, der alternative Züge von MIN (O) und MAX (X) darstellt, bis wir irgendwann Endzustände erreichen, denen gemäß den Spielregeln Werte zugewiesen werden können.

5.2 Optimale Entscheidungen in Spielen

In einem normalen Suchproblem wäre die optimale Lösung eine Folge von Aktionen, die zu einem Zielzustand führen – einem Endzustand, bei dem es sich um einen Gewinn handelt. In einer adversarialen Suche dagegen hat MIN auch noch etwas dazu zu sagen. MAX muss also eine mögliche **Strategie** finden, die den Zug von MAX ab dem Ausgangszustand angibt und dann die Züge von MAX in den Zuständen, die aus den einzelnen Gegenzügen von MIN auf *diese* Züge resultieren usw. Dies ist genau analog zum AND-OR-Suchalgorithmus (Abbildung 4.11), wobei MAX die Rolle von OR spielt und MIN gleichbedeutend mit AND ist. Grob gesagt, führt eine optimale Strategie zu Ergebnissen, die mindestens so gut wie bei jeder anderen Strategie sind, wenn man gegen einen unfehlbaren Gegner spielt. Wir zeigen zunächst, wie man diese optimale Strategie findet.

Selbst ein einfaches Spiel wie Tic Tac Toe ist zu kompliziert, als dass wir den gesamten Spielbaum zeichnen könnten; deshalb wechseln wir zu dem trivialen Spiel in ▶ Abbildung 5.2. Die möglichen Züge für MAX am Wurzelknoten sind als a_1, a_2 und a_3 bezeichnet. Die möglichen Antworten auf a_1 für MIN sind b_1, b_2, b_3 usw. Dieses besondere Spiel endet nach je einem Zug von MAX und MIN. (In der Spielterminologie sagen wir, dass dieser Baum einen Zug tief ist und aus zwei Halbzügen, die auch als **Schicht** bezeichnet werden, besteht.) Die Nutzenwerte der Endzustände in diesem Spiel befinden sich in einem Bereich zwischen 2 und 14.

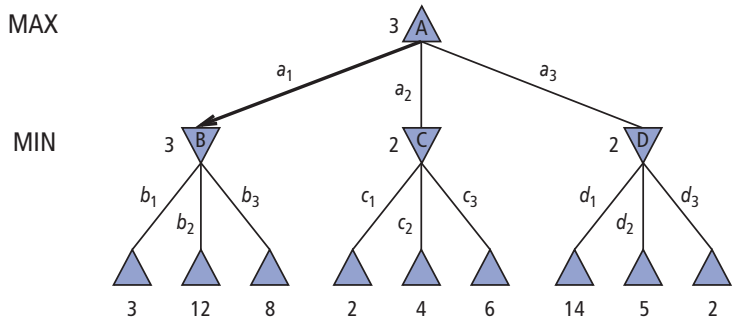


Abbildung 5.2: Ein Zwei-Schichten-Spielbaum. Die Δ-Knoten sind „MAX-Knoten", wobei MAX am Zug ist, und die ∇-Knoten sind „MIN-Knoten". Die Endknoten zeigen die Nutzenwerte für MAX; die anderen Knoten sind mit ihren Minimax-Werten beschriftet. Der beste Zug von MAX an der Wurzel ist a_1, weil er zu dem Nachfolger mit dem höchsten Minimax-Wert führt, und der beste Gegenzug von MIN ist b_1, weil er zu dem Nachfolger mit dem niedrigsten Minimax-Wert führt.

Für einen bestimmten Spielbaum lässt sich die optimale Strategie aus dem **Minimax-Wert** jedes Knotens ermitteln, was wir als MINIMAX(n) schreiben. Der Minimax-Wert eines Knotens ist der Nutzen (für MAX), sich im korrespondierenden Zustand zu befinden, *vorausgesetzt, beide Spieler spielen optimal* von hier aus bis zum Spielende. Offensichtlich ist der Minimax-Wert eines Endzustandes genau sein Nutzen. Darüber hinaus bevorzugt MAX bei einer Auswahlmöglichkeit den Zug zu einem Zustand mit maximalem Wert, während MIN einen Zustand mit minimalem Wert bevorzugt. Wir haben also:

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \textit{wenn } \text{TERMINAL-TEST}(s) \\ \max_{s \in Action(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \textit{wenn } \text{PLAYER}(s) = \text{MAX} \\ \min_{s \in Action(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \textit{wenn } \text{PLAYER}(s) = \text{MIN} \end{cases}$$

Jetzt wenden wir diese Definitionen auf den Spielbaum in Abbildung 5.2 an. Die Endknoten auf der unteren Ebene sind bereits mit ihren Nutzenwerten von der UTILITY-Funktion des Spieles beschriftet. Der erste MIN-Knoten, *B*, hat drei Nachfolgerzustände mit den Werten 3, 12 und 8, sein Minimax-Wert ist also 3. Analog dazu haben die beiden anderen MIN-Knoten den Minimax-Wert 2. Der Wurzelknoten ist ein MAX-Knoten; seine Nachfolgerzustände haben die Minimax-Werte 3, 2 und 2; er hat also einen Minimax-Wert von 3. Wir können auch die **Minimax-Entscheidung** an der Wurzel identifizieren: Die Aktion a_1 ist die optimale Auswahl für MAX, weil sie zu dem Nachfolger mit dem höchsten Minimax-Wert führt.

Diese Definition optimalen Spielens für MAX setzt voraus, dass auch MIN optimal spielt – das Ergebnis des *ungünstigsten Falles* für MAX wird maximiert. Was passiert, wenn MIN nicht optimal spielt? Dann lässt sich leicht zeigen (Übung 5.7), dass MAX noch besser abschneidet. Es kann andere Strategien gegen suboptimale Gegner geben, die besser als die Minimax-Strategie sind; aber diese Strategien sind im Allgemeinen schlechter gegen optimale Gegner.

5.2.1 Der Minimax-Algorithmus

Der **Minimax-Algorithmus** (▶ Abbildung 5.3) berechnet die Minimax-Entscheidung aus dem aktuellen Zustand. Er verwendet eine einfache rekursive Berechnung der Minimax-Werte jedes Nachfolgerzustandes und implementiert direkt die definierenden Gleichungen. Die Rekursion verläuft bis an die Blätter des Baumes und dann werden die Minimax-Werte durch den Baum hindurch **gespeichert**, wenn die Rekursion abgewickelt wird. In Abbildung 5.2 beispielsweise läuft der Algorithmus zuerst nach unten zu den drei Knoten unten links und wendet dann die UTILITY-Funktion darauf an, um festzustellen, dass ihre Werte 3, 12 bzw. 8 sind. Anschließend ermittelt er das Minimum dieser Werte (3) und gibt es als den gespeicherten Wert von Knoten *B* zurück. Ein ähnlicher Prozess erzeugt die neuen Werte 2 für *C* und 2 für *D*. Schließlich nehmen wir das Maximum von 3, 2 und 2, um den gespeicherten Wert von 3 für den Wurzelknoten zu erhalten.

```
function MINIMAX-DECISION(state) returns eine Aktion
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns ein Nutzenwert
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← -∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns ein Nutzenwert
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

Abbildung 5.3: Ein Algorithmus für die Berechnung von Minimax-Entscheidungen. Er gibt die Aktion zurück, die dem bestmöglichen Zug entspricht, d.h. dem Zug, der zu dem Ergebnis mit der besten Nützlichkeit führt – unter der Annahme, dass der Gegner so spielt, dass er die Nützlichkeit minimieren will. Die Funktionen MAX-VALUE und MIN-VALUE durchlaufen den gesamten Spielbaum bis hin zu den Blättern, um den gespeicherten Wert eines Zustandes zu ermitteln. Die Notation $\arg\max_{a \in S} f(a)$ berechnet das Element a der Menge S, das den maximalen Wert von $f(a)$ hat.

Der Minimax-Algorithmus untersucht den Spielbaum mit einer vollständigen Tiefensuche. Ist die maximale Tiefe des Baumes gleich m und gibt es b erlaubte Züge an jedem Punkt, ist die Zeitkomplexität des Minimax-Algorithmus gleich $O(b^m)$. Die Speicherkomplexität ist $O(bm)$ für einen Algorithmus, der alle Aktionen gleichzeitig erzeugt, und $O(m)$ für einen Algorithmus, der die Aktionen nacheinander erzeugt

(siehe *Abschnitt 3.4.3*). Für reale Spiele sind die Zeitkosten natürlich völlig inakzeptabel, aber dieser Algorithmus dient als Grundlage für die mathematische Analyse von Spielen und für realistischere Algorithmen.

5.2.2 Optimale Entscheidungen in Mehrspieler-Spielen

Viele bekannte Spiele sind für mehr als zwei Spieler ausgelegt. Betrachten wir also jetzt, wie wir die Minimax-Idee auf Spiele mit mehreren Spielern erweitern können. Das ist aus der technischen Perspektive ganz einfach, wirft aber einige interessante neue konzeptionelle Fragen auf.

Zuerst müssen wir den einzelnen Wert für jeden Knoten durch einen *Vektor* mit Werten ersetzen. Zum Beispiel wird in einem Drei-Spieler-Spiel mit den Spielern *A*, *B* und *C* jedem Knoten ein Vektor $\langle v_A, v_B, v_C \rangle$ zugeordnet. Für Endzustände zeigt dieser Vektor den Nutzen des Zustandes aus der Perspektive des jeweiligen Spielers an. (In Zwei-Spieler-Nullsummenspielen kann der zweielementige Vektor auf einen einzigen Wert reduziert werden, weil die Werte immer entgegengesetzt sind.) Am einfachsten implementiert man das, indem man die UTILITY-Funktion einen Vektor mit Nutzenwerten zurückgeben lässt.

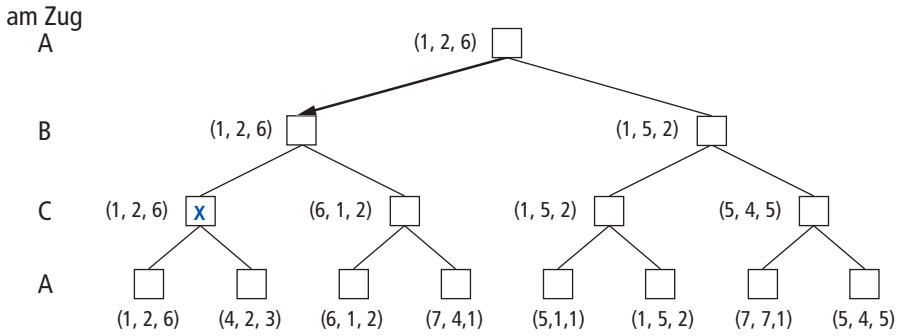


Abbildung 5.4: Die ersten drei Schichten eines Spielbaumes mit drei Spielern (A, B, C). Jeder Knoten ist mit den Werten aus der Perspektive der einzelnen Spieler beschriftet. Der beste Zug ist an der Wurzel markiert.

Jetzt müssen wir die Nichtendzustände betrachten. Sehen Sie sich den mit X markierten Knoten im Spielbaum in ▶ Abbildung 5.4 an. In diesem Zustand wählt Spieler *C*, was zu tun ist. Die beiden Auswahlmöglichkeiten führen zu Endzuständen mit den Nutzenvektoren $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$ und $\langle v_A = 4, v_B = 2, v_C = 3 \rangle$. Weil 6 größer als 3 ist, sollte *C* den ersten Zug wählen. Das bedeutet, wenn Zustand *X* erreicht ist, führt ein nachfolgendes Spiel zu einem Endzustand mit den Nutzenwerten $\langle v_A = 1, v_B = 2, v_C = 6 \rangle$. Damit ist der gespeicherte Wert von *X* dieser Vektor. Der gespeicherte Wert eines Knotens *n* ist immer der Nutzenvektor des Nachfolgerzustandes mit dem höchsten Wert, den der Spieler bei *n* wählen kann. Jeder, der Mehrspieler-Spiele spielt, wie beispielsweise Diplomacy, erkennt schnell, dass dort sehr viel mehr passiert als in Zwei-Spieler-Spielen. Mehrspieler-Spiele beinhalten häufig **Zusammenschlüsse** (**Allianzen**) unter den Spielern – egal ob formell oder formlos. Allianzen werden im Laufe des Spieles geschlossen und aufgelöst. Wie verstehen wir ein solches Verhalten? Sind Allianzen eine natürliche Konsequenz optimaler Strategien für alle Spieler in einem Mehrspieler-Spiel? Es stellt sich heraus, dass das durchaus sein kann. Angenommen,

A und B befinden sich in schwachen Positionen, während C eine stärkere Position innehat. Dann ist es für A und B optimal, C anzugreifen, anstatt sich gegenseitig anzugreifen und auf diese Weise zuzulassen, dass C sie einzeln zerstört. Somit entwickelt sich aus rein egoistischem Verhalten eine Zusammenarbeit. Sobald natürlich C unter dem gemeinsamen Beschuss schwächer geworden ist, verliert die Allianz ihre Bedeutung und A oder B könnten das Abkommen verletzen. In einigen Fällen machen explizite Allianzen nur konkret, was ohnehin passiert wäre. In anderen Fällen bedeutet es ein soziales Stigma, eine Allianz zu brechen; deshalb müssen die Spieler abwägen zwischen dem Verlassen einer Allianz und dem langfristigen Nachteil, nicht als vertrauenswürdig betrachtet zu werden. Weitere Informationen über diese Komplikationen finden Sie in *Abschnitt 17.5*.

Handelt es sich nicht um ein Nullsummenspiel, kann die Zusammenarbeit auch zwischen nur zwei Spielern auftreten. Nehmen Sie beispielsweise an, dass es einen Endzustand mit den Nutzenwerten $\langle v_A = 1000, v_B = 1000 \rangle$ gibt und dass 1000 der höchstmögliche Nutzen für jeden Spieler ist. Die optimale Strategie ist dann für beide Spieler, alles Mögliche zu unternehmen, um diesen Zustand zu erreichen – d.h., die Spieler arbeiten automatisch zusammen, um ein wechselseitig wünschenswertes Ziel zu erreichen.

5.3 Alpha-Beta-Kürzung

Das Problem bei der Minimax-Suche ist, dass die Anzahl der auszuwertenden Spielzustände exponentiell zur Anzahl der Züge zunimmt. Leider lässt sich der Exponent nicht beseitigen, es zeigt sich aber, dass man ihn praktisch halbieren kann. Es ist nämlich möglich, die korrekte Minimax-Entscheidung zu berechnen, ohne jeden Knoten im Spielbaum betrachten zu müssen. Das bedeutet, wir können uns das Konzept der **Kürzung** (Pruning) aus *Kapitel 3* ausborgen, um große Teile des Baumes aus der Analyse auszuschließen. Die entsprechende Technik, die wir hier untersuchen wollen, ist die **Alpha-Beta-Kürzung**. Wendet man das Verfahren auf einen standardmäßigen Minimax-Baum an, gibt es den gleichen Zug wie die Minimax-Methode zurück, beschneidet aber die Verzweigungen, die die endgültige Entscheidung wahrscheinlich nicht beeinflussen können.

Sehen Sie sich noch einmal den Zwei-Schichten-Spielbaum von Abbildung 5.2 an. Wir gehen die Berechnung der optionalen Entscheidung erneut durch, achten aber jetzt besonders darauf, was wir an jedem Punkt in dem Prozess wissen. Die Schritte sind in ▶ Abbildung 5.5 erklärt. Das Ergebnis ist, dass wir zur Minimax-Entscheidung gelangen, ohne zwei der Blattknoten überhaupt auswerten zu müssen.

Eine weitere Betrachtungsweise ist eine Vereinfachung der Formel für MINIMAX. Für die beiden nicht ausgewerteten Nachfolger des Knotens C in Abbildung 5.5 seien die Werte x und y angenommen. Der Wert des Wurzelknotens ist dann gegeben durch:

$$\begin{aligned}
\text{MINIMAX}(root) &= \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2)) \\
&= \max(3, \min(2, x, y), 2) \\
&= \max(3, z, 2) \quad \text{wobei } z = \min(2, x, y) \leq 2 \\
&= 3
\end{aligned}$$

Mit anderen Worten sind der Wert der Wurzel und damit die Minimax-Entscheidung *unabhängig* von den Werten der gekürzten Blätter x und y.

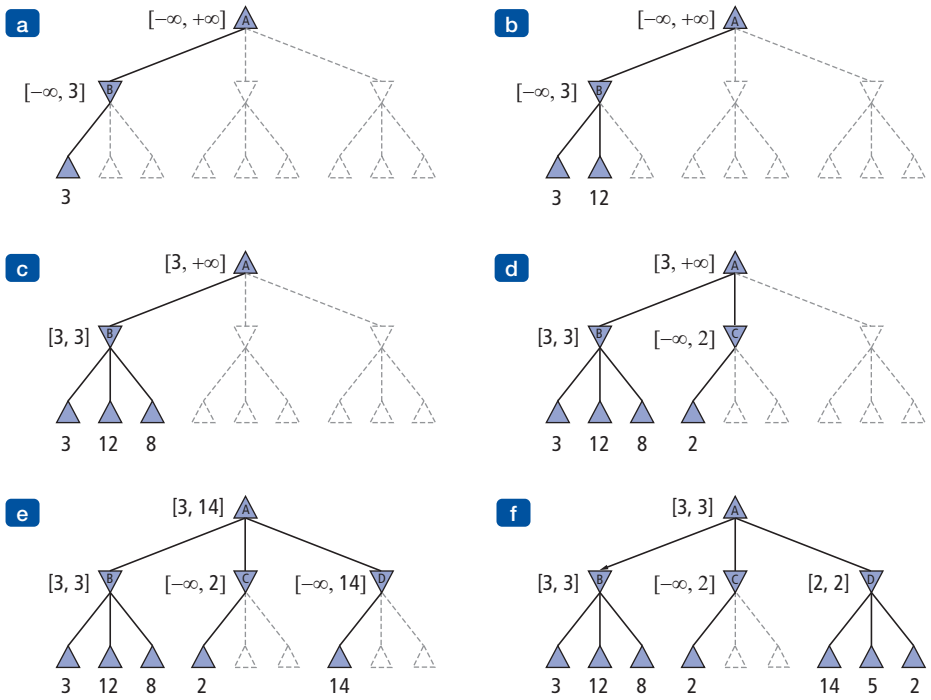


Abbildung 5.5: Phasen in der Berechnung der optimalen Entscheidung für den Spielbaum in Abbildung 5.2. Wir zeigen an jedem Punkt den Bereich möglicher Werte für jeden Knoten. (a) Das erste Blatt unter B hat den Wert 3. Somit hat B, ein MIN-Knoten, einen Wert von *höchstens* 3. (b) Das zweite Blatt unter B hat den Wert 12. MIN würde diesen Zug vermeiden; der Wert von B ist also immer noch höchstens 3. (c) Das dritte Blatt unter B hat den Wert 8. Wir haben alle Nachfolger von B gesehen; der Wert von B ist also genau 3. Jetzt können wir ableiten, dass der Wert der Wurzel *mindestens* 3 ist, weil MAX eine Auswahlmöglichkeit von 3 an der Wurzel hat. (d) Das erste Blatt unter C hat den Wert 2. Damit hat C – MIN-Knoten – einen Wert von *höchstens* 2. Wir wissen jedoch, dass B den Wert 3 hat und MAX deshalb nie C wählen würde. Es ist also nicht sinnvoll, die weiteren Nachfolger von C zu betrachten. Das ist ein Beispiel für Alpha-Beta-Kürzung. (e) Das erste Blatt unter D hat den Wert 14, sodass D *höchstens* 14 ist. Das ist immer noch höher als die beste Alternative von MAX (d.h. 3), deshalb müssen wir die Nachfolgerzustände von D weiter betrachten. Beachten Sie auch, dass wir jetzt Grenzen für alle Nachfolger der Wurzel haben; der Wert der Wurzel also höchstens 14 ist. (f) Der zweite Nachfolger von D ist 5. Wir müssen also weiterhin auswerten. Der dritte Nachfolger ist 2; damit ist D genau gleich 2. Die Entscheidung von MAX an der Wurzel ist, nach B zu ziehen, woraus sich ein Wert von 3 ergibt.

Tipp

Die Alpha-Beta-Kürzung kann auf Bäume beliebiger Tiefe angewendet werden und häufig ist es möglich, nicht nur einzelne Blätter, sondern ganze Unterbäume zu kürzen. Das allgemeine Prinzip sieht wie folgt aus: Man betrachtet einen Knoten n irgendwo im Baum (siehe ▶ Abbildung 5.6), sodass der Spieler die Wahl hat, zu diesem Knoten zu ziehen. Hat der Spieler eine bessere Wahl m entweder am übergeordneten Knoten von n oder an einem anderen Auswahlpunkt weiter oben, wird *n im tatsächlichen Spiel niemals erreicht*. Sobald wir also genug über n herausgefunden haben (indem wir einige seiner Nachfolger betrachten), um zu diesem Schluss zu kommen, können wir es kürzen.

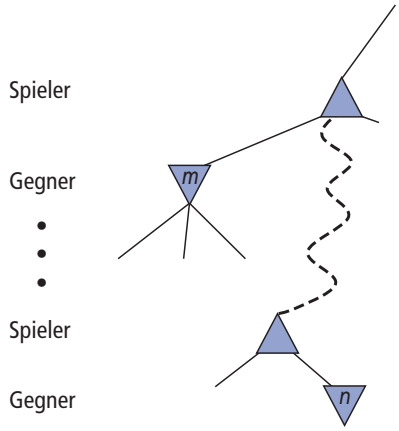


Abbildung 5.6: Alpha-Beta-Kürzung: der allgemeine Fall. Ist m für den Spieler besser als n, gelangen wir im Spiel nie zu n.

Wie bereits erwähnt, ist die Minimax-Suche eine Tiefensuche. Wir müssen also zu jedem Zeitpunkt lediglich die Knoten entlang eines einzigen Pfades im Baum betrachten. Die Alpha-Beta-Kürzung hat ihren Namen aus den beiden folgenden Parametern erhalten, die die Grenzen für die aktualisierten Werte beschreiben, die irgendwo entlang des Pfades auftreten:

α = der Wert der besten Wahl (d.h. die mit dem höchsten Wert), die wir bisher an jedem Auswahlpunkt entlang des Pfades für MAX ermittelt haben

β = der Wert der besten Wahl (d.h. die mit dem niedrigsten Wert), die wir bisher an jedem Auswahlpunkt entlang des Pfades für MIN ermittelt haben

Die **Alpha-Beta-Suche** aktualisiert die Werte von α und β, während sie ausgeführt wird, und kürzt die verbleibenden Verzweigungen an einem Knoten (d.h. terminiert den rekursiven Aufruf), sobald für den Wert des aktuellen Knotens bekannt ist, dass er schlechter als der aktuelle α- oder β-Wert für MAX bzw. MIN ist. ▶ Abbildung 5.7 zeigt den vollständigen Algorithmus. Es empfiehlt sich, sein Verhalten bei der Anwendung auf den Baum aus Abbildung 5.5 nachzuvollziehen.

```
function ALPHA-BETA-SEARCH(state) returns eine Aktion
  v ← MAX-VALUE(state, -∞, +∞)
  return die action in ACTIONS(state) mit Wert v

function MAX-VALUE(state, α, β) returns einen Nutzenwert
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← -∞
  for each a in ACTIONS(state) do
    v ← MAX(v , MIN-VALUE(RESULT(s, a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

```
function MIN-VALUE(state, α, β) returns einen Nutzenwert
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
```

```
    v ← MIN(v, MAX-VALUE(RESULT(s, a), α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Abbildung 5.7: Der Alpha-Beta-Suchalgorithmus. Beachten Sie, dass es sich hierbei um die gleichen Routinen handelt wie bei den MINIMAX-Funktionen in Abbildung 5.3 bis auf die beiden Zeilen in MIN-VALUE und MAX-VALUE, die α und β verwalten (und die Mechanismen, die für die Übergabe dieser Parameter zuständig sind).

5.3.1 Zugreihenfolge

Die Effektivität des Alpha-Beta-Kürzens ist wesentlich von der Reihenfolge abhängig, in der die Zustände untersucht werden. Zum Beispiel könnten wir in Abbildung 5.5(e) und (f) überhaupt keine Nachfolger von D kürzen, weil die schlechteren Nachfolger (aus Perspektive von MIN) zuerst erzeugt wurden. Wäre der dritte Nachfolger von D zuerst erzeugt worden, hätten wir die beiden anderen kürzen können. Das legt nahe, dass es sinnvoll sein könnte, zuerst die Nachfolger auszuprobieren, die wahrscheinlich die besten Werte erbringen.

Wenn dies möglich ist,[2] zeigt sich, dass Alpha-Beta nur $O(b^{m/2})$ Knoten auswerten muss, um den besten Zug zu ermitteln, und nicht $O(b^m)$ wie Minimax. Das bedeutet, dass der effektive Verzweigungsfaktor zu $\sqrt{b}$ statt b wird – für Schach also 6 statt 35. Anders ausgedrückt, kann Alpha-Beta innerhalb derselben Zeit einen etwa doppelt so tiefen Baum gegenüber Minimax lösen. Werden die Nachfolger in zufälliger Reihenfolge statt nach einer Bestensuche ausgewertet, ergibt sich eine Gesamtzahl der Knoten zu etwa $O(b^{3m/4})$ für moderate b. Beim Schach bringt Sie eine relativ einfache Sortierfunktion (z.B. zuerst Schlagen, dann Bedrohen, dann Vorwärtszüge und dann Rückwärtszüge zu probieren) in einen Faktorbereich von 2 des Ergebnisses für den besten Fall $O(b^{m/2})$.

Wenn wir dynamische Zugfolgeschemata einführen – um beispielsweise zuerst die Züge auszuprobieren, die in der Vergangenheit die besten waren –, gelangen wir noch näher an das theoretische Limit heran. Bei der Vergangenheit kann es sich um den vorherigen Zug handeln – oftmals bleibt dieselbe Bedrohung erhalten – oder von der vorherigen Untersuchung des aktuellen Zuges stammen. Informationen vom aktuellen Zug lassen sich unter anderem mit iterativer Tiefensuche gewinnen. Die Suche erstreckt sich zuerst bis in eine Tiefe von 1 Schicht und zeichnet den besten Zugpfad auf. Dann geht die Suche 1 Schicht tiefer, verwendet aber den aufgezeichneten Pfad, um über die Zugreihenfolge zu informieren. Wie *Kapitel 3* erläutert hat, schlägt sich die iterative Tiefensuche bei einem exponentiellen Spielbaum nur um einen konstanten Bruchteil in der Gesamtsuchzeit nieder, was mehr sein kann, als sich mit einer besseren Zugreihenfolge erreichen lässt. Die besten Züge werden oftmals auch als **Killerzüge** bezeichnet. Und wenn sie zuerst ausprobiert werden, spricht man auch von einer Killerzugheuristik.

In *Kapitel 3* haben wir bemerkt, dass wiederholte Zustände im Suchbaum eine exponentielle Zunahme der Kosten verursachen können. In Spielen treten aufgrund von **Transpositionen** häufig wiederholte Zustände auf – verschiedene Permutationen der Zugfolge, die zur selben Position führen. Hat Weiß beispielsweise einen Zug a_1, der

2 Natürlich ist es nicht perfekt möglich; andernfalls könnte die Sortierfunktion genutzt werden, um ein perfektes Spiel zu spielen!

von Schwarz mit b_1 beantwortet werden kann, und ein anderer, nicht damit zusammenhängender Zug a_2 auf der anderen Seite des Brettes kann mit b_2 beantwortet werden, dann führen die Folgen $[a_1, b_1, a_2, b_2]$ und $[a_1, b_2, a_2, b_1]$ beide zur selben Position. Es lohnt sich, die Auswertung der resultierenden Position bei ihrem ersten Auftreten in einer Hash-Tabelle zu speichern, sodass wir sie bei nachfolgenden Vorkommen nicht erneut berechnen müssen. Die Hash-Tabelle von bereits zuvor gesehenen Positionen wird traditionell als **Transpositionstabelle** bezeichnet; sie ist im Wesentlichen identisch mit der *untersuchten* Liste in GRAPH-SEARCH (*Abschnitt 3.3*). Die Verwendung einer Transpositionstabelle kann drastische Auswirkungen haben – beim Schach manchmal bis zu einer Verdopplung der erreichbaren Suchtiefe. Wenn wir andererseits eine Million Knoten pro Sekunde auswerten, ist es nicht sinnvoll, sie *alle* in der Transpositionstabelle abzulegen. Es wurden verschiedene Strategien verwendet, um diejenigen Knoten auszuwählen, die beizubehalten bzw. zu verwerfen sind.

5.4 Unvollständige Echtzeitentscheidungen

Der Minimax-Algorithmus erzeugt den gesamten Spielsuchraum, während der Alpha-Beta-Algorithmus es uns erlaubt, große Teile davon zu kürzen. Alpha-Beta muss dennoch den gesamten Weg bis zu Endzuständen für zumindest einen Teil des Suchraumes durchsuchen. Diese Tiefe ist häufig nicht praktisch, weil Züge innerhalb einer sinnvollen Zeit erfolgen müssen – normalerweise höchstens innerhalb von ein paar Minuten. Shannon schlug in *Programming a Computer for Playing Chess* (1950) vor, dass die Programme die Suche eher abbrechen und eine heuristische **Bewertungsfunktion** für Zustände in der Suche anwenden sollen, was letztlich Nichtendknoten in Endblätter umwandelte. Mit anderen Worten besteht sein Vorschlag darin, Minimax oder Alpha-Beta auf zwei Arten abzuändern: Die Nutzenfunktion wird durch eine heuristische Bewertungsfunktion EVAL ersetzt, die eine Abschätzung für den Nutzen der Position ermittelt, und der Endetest wird durch einen **Cutoff-Test** (**Abbruchtest**) ersetzt, der entscheidet, wann EVAL angewendet werden soll. Damit erhalten wir die folgenden Ausdrücke für heuristisches Minimax für den Zustand s und maximale Tiefe d:

$$\text{H-MINIMAX}(s, d)$$

$$\begin{cases} \text{EVAL}(s) & \textit{wenn } \text{CUTOFF-TEST}(s, d) \\ \max_{a \in \textit{Action}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d+1) & \textit{wenn } \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in \textit{Action}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d+1) & \textit{wenn } \text{PLAYER}(s) = \text{MIN} \end{cases}$$

5.4.1 Bewertungsfunktionen

Eine Bewertungsfunktion gibt eine *Schätzung* des erwarteten Nutzens des Spieles von einer bestimmten Position ab zurück, so wie die Heuristikfunktionen aus *Kapitel 3* eine Schätzung der Distanz zum Ziel liefern. Die Idee eines Schätzers war nicht neu, als Shannon sie vorschlug. Jahrhunderte lang haben Schachspieler (und Anhänger anderer Spiele) Möglichkeiten entwickelt, den Wert einer Position einzuschätzen, weil Menschen noch beschränkter in ihren Suchmöglichkeiten als Computerprogramme sind. Es dürfte klar sein, dass die Leistung eines Spieleprogramms von der Qualität seiner Bewer-

tungsfunktion abhängig ist. Eine ungenaue Bewertungsfunktion leitet einen Agenten an Positionen, die sich als Verluste herausstellen. Aber wie genau können wir gute Bewertungsfunktionen entwerfen?

Erstens sollte die Bewertungsfunktion die *Endzustände* auf dieselbe Weise sortieren wie die echte Nutzenfunktion: Zustände, die Gewinne darstellen, müssen ein besseres Ergebnis liefern als Remis, die ihrerseits besser sein müssen als Verluste. Andernfalls könnte ein Agent, der die Bewertungsfunktion verwendet, sich irren, selbst wenn er das Spiel von Anfang bis Ende überblicken kann. Zweitens darf die Berechnung nicht zu lange dauern! (Im Prinzip dreht sich alles darum, schneller zu suchen.) Drittens sollte die Bewertungsfunktion für andere als Endzustände stark mit den tatsächlichen Gewinnchancen korreliert sein.

Der Begriff „Gewinnchancen“ könnte Fragen aufwerfen. Schließlich ist Schach kein Glücksspiel: Wir kennen den aktuellen Zustand mit Bestimmtheit und es wird auch nicht gewürfelt. Doch wenn die Suche bei Nichtendzuständen abgebrochen werden muss, ist der Algorithmus zwangsläufig *unsicher* im Hinblick auf das endgültige Ergebnis dieser Zustände. Diese Art der Unsicherheit wird durch Beschränkungen der Rechenressourcen, nicht durch Informationsbeschränkungen verursacht. Betrachtet man die beschränkte Menge an Rechenressourcen, die der Bewertungsfunktion für einen bestimmten Zustand zur Verfügung stehen, lässt sich das Endergebnis bestenfalls raten.

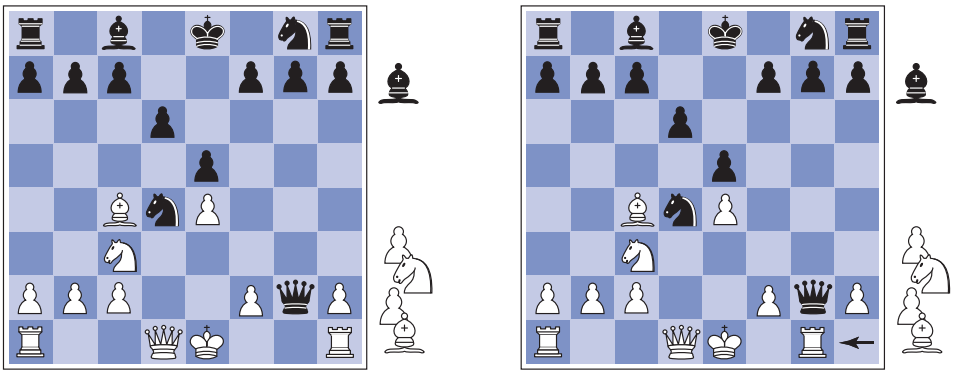
Formulieren wir dieses Konzept konkreter. Die meisten Bewertungsfunktionen beruhen darauf, verschiedene **Merkmale** des Zustandes zu berechnen – zum Beispiel wären das beim Schach Merkmale für die Anzahl der weißen Bauern, der schwarzen Bauern, der weißen Damen, der schwarzen Damen usw. Die Merkmale definieren in ihrer Gesamtheit verschiedene *Kategorien* oder *Äquivalenzklassen* von Zuständen: Die Zustände in jeder Kategorie haben für alle Merkmale dieselben Werte. Zum Beispiel enthält eine Kategorie alle Endspiele mit zwei Bauern gegen einen Bauern. Allgemein ausgedrückt enthält jede Kategorie bestimmte Zustände, die zum Gewinn führen, einige, die ein Remis ergeben, und einige, die Verlust bedeuten. Die Bewertungsfunktion kann nicht wissen, welche Zustände welche sind, aber sie kann einen einzelnen Wert zurückgeben, der die *Proportion* der Zustände zu jedem Ergebnis widerspiegelt. Nehmen wir zum Beispiel an, dass laut unserer Erfahrung 72% der Zustände in der Kategorie „zwei Bauern gegen einen Bauern“ zu einem Gewinn führen (Nutzen +1); 20% zu einem Verlust (0) und 8% zu einem Remis (1/2). Eine sinnvolle Bewertung für Zustände in der Kategorie ist dann der **Erwartungswert**: $(0{,}72 \times +1) + (0{,}20 \times 0) + (0{,}08 \times 1/2) = 0{,}76$. Im Prinzip kann der Erwartungswert für jede Kategorie ermittelt werden, was zu einer Bewertungsfunktion führt, die für jeden Zustand funktioniert. Wie bei Endzuständen muss die Bewertungsfunktion keine wirklichen Erwartungswerte zurückgeben, solange die *Reihenfolge* der Zustände dieselbe ist.

In der Praxis sind für diese Art der Analyse zu viele Kategorien und damit zu viel Erfahrung erforderlich, um alle Wahrscheinlichkeiten eines Gewinns abzuschätzen. Stattdessen berechnen die meisten Bewertungsfunktionen separate numerische Beiträge aus jedem Merkmal und *kombinieren* diese, um den Gesamtwert zu ermitteln. Zum Beispiel geben Schachbücher für Anfänger für jede Figur geschätzte **Materialwerte** an: Ein Bauer hat den Wert 1, ein Springer oder ein Läufer 3, ein Turm 5 und die Dame 9. Andere Merkmale wie etwa „gute Bauernstruktur“ oder „Sicherheit des Königs“ könnten z.B. eine halbe Bauerneinheit wert sein. Diese Merkmalswerte werden dann einfach addiert, um die Bewertung der Position zu erhalten.

Ein Sicherheitsvorteil, der äquivalent mit einem Bauern ist, bedeutet eine hohe Gewinnwahrscheinlichkeit, und ein Sicherheitsvorteil, der äquivalent mit drei Bauern ist, sollte einen so gut wie sicheren Gewinn bedeuten, wie in ▶ Abbildung 5.8(a) gezeigt. Mathematisch wird diese Art der Bewertungsfunktion als **gewichtete Linearfunktion** bezeichnet, weil sie als

$$\text{EVAL}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s),$$

ausgedrückt werden kann, wobei jedes w_i eine Gewichtung und jedes f_i ein Merkmal der Position ist. Beim Schach könnten die f_i die Anzahlen der einzelnen Figurenarten auf dem Brett angeben und die w_i die Werte für die Figuren sein (1 für Bauern, 3 für Läufer usw.).



a Weiß ist am Zug

b Weiß ist am Zug

Abbildung 5.8: Zwei Schachstellungen, die sich nur in der Position des Turmes unten rechts unterscheiden. In (a) hat Schwarz einen Vorteil von einem Springer und zwei Bauern, was für einen Spielgewinn genügen dürfte. In (b) schlägt Weiß die Dame und erlangt dadurch einen Vorteil, der für einen Gewinn stark genug sein sollte.

Es scheint zwar sinnvoll zu sein, die Werte der Merkmale zu addieren, doch letztlich beruht dies auf der sehr strengen Annahme, dass der Beitrag jedes Merkmals *unabhängig* von den Werten der anderen Merkmale ist. Weist man beispielsweise einem Läufer den Wert 3 zu, wird damit die Tatsache ignoriert, dass Läufer im Endspiel leistungsfähiger sind, wo sie sehr viel Platz für ihre Manöver haben. Aus diesem Grund verwenden die heutigen Programme für Schach und andere Spiele auch *nichtlineare* Kombinationen von Funktionsmerkmalen. Ein Läuferpaar könnte beispielsweise etwas mehr wert sein als einfach der doppelte Wert eines einzelnen Läufers und ein Läufer ist im Endspiel mehr wert als am Anfang (d.h., wenn das Merkmal *Zugzahl* hoch oder das Merkmal *Anzahl der verbleibenden Figuren* niedrig ist).

Der aufmerksame Leser hat zweifellos bemerkt, dass die Merkmale und Gewichtungen *nicht* Teil der Schachregeln sind! Sie stammen aus Jahrhunderten an Erfahrung, die die Menschen beim Schachspielen gesammelt haben. In Spielen, wo diese Art der Erfahrung nicht zur Verfügung steht, lassen sich die Gewichtungen der Bewertungsfunktion mit Techniken des maschinellen Lernens (siehe *Kapitel 18*) abschätzen. Seien Sie aber versichert, dass die Anwendung dieser Techniken auf Schach bestätigt hat, dass ein Läufer tatsächlich etwa drei Bauern wert ist.

5.4.2 Abbrechen der Suche

Im nächsten Schritt ist die Funktion ALPHA-BETA-SEARCH so abzuändern, dass sie die Funktion EVAL aufruft, sobald es sinnvoll ist, die Suche abzubrechen. Für die Implementierung ersetzen wir die beiden Zeilen in Abbildung 5.7, die TERMINAL-TEST enthalten, durch die folgende Zeile:

```
if CUTOFF-TEST(state, depth) then return EVAL(state)
```

Wir müssen noch ein wenig Verwaltungsaufwand betreiben, sodass die aktuelle Tiefe *depth* bei jedem rekursiven Aufruf inkrementiert wird. Der Umfang der Suche lässt sich am einfachsten steuern, wenn eine feste Tiefe vorgegeben wird, sodass CUTOFF-TEST(*state*, *depth*) für alle Werte von *depth* größer einer festen Tiefe *d* den Wert *true* zurückgibt. (Die Funktion muss auch für alle Endzustände *true* zurückgeben, so wie es bei TERMINAL-TEST der Fall war.) Die Tiefe *d* legt man so fest, dass der Zug innerhalb der zugestandenen Zeit ausgewählt wird. Ein robusterer Ansatz wendet die iterative Vertiefung an (siehe *Kapitel 3*). Wenn die Zeit abläuft, gibt das Programm den Zug zurück, der durch die tiefste vollständige Suche ausgewählt wurde. Als Bonus hilft die iterative Tiefensuche auch bei der Zugreihenfolge.

Diese einfachen Ansätze können jedoch zu Fehlern führen, weil die Bewertungsfunktion nur Schätzungen vornimmt. Sehen Sie sich noch einmal die einfache Bewertungsfunktion für Schach an, die auf einem Materialvorteil basiert. Angenommen, das Programm sucht bis zur Tiefenbeschränkung und erreicht die in ▶ Abbildung 5.8(b) gezeigte Position, in der Schwarz um einen Springer und zwei Bauern im Vorteil ist. Es würde dies als heuristischen Wert des Zustandes melden und dabei aussagen, dass Schwarz in diesem Zustand wahrscheinlich gewinnt. Im nächsten Zug schlägt Weiß jedoch die Dame von Schwarz, ohne dass es einen Ausgleich dafür gibt. Damit war die Position letztlich ein Gewinn für Weiß, aber das ist nur ersichtlich, wenn um mehr als eine Schicht vorausgesehen wird.

Offensichtlich braucht man einen komplexeren Abbruchtest. Die Bewertungsfunktion sollte nur auf Positionen angewendet werden, die **ruhend** sind – d.h. die in der nahen Zukunft keine wilden Wertumschwünge aufweisen werden. Im Schach beispielsweise sind Positionen, in denen günstige Figuren geschlagen werden können, nicht ruhend für eine Bewertungsfunktion, die nur Material zählt. Nichtruhende Positionen können weiter expandiert werden, bis ruhende Positionen erreicht sind. Diese zusätzliche Suche wird als **Ruhesuche** bezeichnet; manchmal ist sie darauf beschränkt, nur bestimmte Arten von Zügen zu berücksichtigen, wie etwa Züge, mit denen Figuren geschlagen werden und die damit schnell die Unsicherheiten in der Position auflösen.

Der **Horizonteffekt** ist schwieriger zu eliminieren. Er tritt auf, wenn das Programm mit einem Zug des Gegners konfrontiert wird, der ernsthafte Schäden verursacht und letztlich unausweichlich ist, sich jedoch vorübergehend durch Verzögerungstaktiken verhindern lässt. Sehen Sie sich dazu das Schachspiel in ▶ Abbildung 5.9 an. Es ist klar, dass der schwarze Läufer keine Chance hat zu entkommen. Zum Beispiel kann der weiße Turm ihn schlagen, indem er nach h1, dann nach a1 und schließlich nach a2 geht – ein Schlagen bei einer Schichttiefe von 6. Doch Schwarz hat eine Zugfolge, die das Schlagen des Läufers „über den Horizont hinaus“ vereitelt. Nehmen wir an, dass Schwarz bis zur Schichttiefe 8 sucht. Die meisten Züge von Schwarz führen letztlich dazu, dass Weiß den Läufer schlägt, und werden deshalb als „schlechte“ Züge markiert. Doch Schwarz kann auch dem weißen König mit dem Bauern auf e4 Schach bieten. Daraufhin

schlägt der König den Bauern. Jetzt bietet Schwarz erneut Schach, und zwar mit dem Bauern auf f5, woraufhin ein weiterer Bauer geschlagen wird. Hierfür sind 4 Schichten nötig. Und von da an genügen die verbleibenden 4 Schichten nicht, um den Läufer zu schlagen. Schwarz glaubt zwar, dass der Spielverlauf den Läufer zum Preis von zwei Bauern rettet, hat aber tatsächlich nichts weiter getan, als das unvermeidliche Schlagen des Läufers über den Horizont zu verschieben, den Schwarz sehen kann.

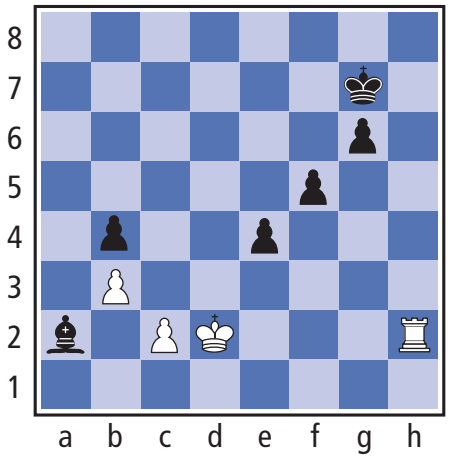


Abbildung 5.9: Der Horizonteffekt. Mit Schwarz am Zug ist der schwarze Läufer sicher verloren. Doch Schwarz kann dieses Ereignis vereiteln, indem es mit seinen Bauern dem weißen König Schach bietet und den König zwingt, die Bauern zu schlagen. Das drückt den unvermeidlichen Verlust des Läufers über den Horizont und der Algorithmus betrachtet die Bauernopfer als gute statt als schlechte Züge.

Der Horizonteffekt lässt sich unter anderem durch die **singuläre Erweiterung** mildern, d.h. mit einem Zug, der „deutlich besser" als andere Züge in einer bestimmten Position ist. Entdeckt der Algorithmus im Verlauf der Suche irgendwo im Baum einen singulären Zug, merkt er ihn sich. Wenn die Suche die normale Tiefenbeschränkung erreicht, prüft der Algorithmus, ob die singuläre Erweiterung einen gültigen Zug darstellt. Ist das der Fall, berücksichtigt der Algorithmus den Zug. Der Baum wird dadurch zwar tiefer, doch da es nur wenige singuläre Erweiterungen geben dürfte, kommen nicht zu viele vollständige Knoten zum Baum hinzu.

5.4.3 Vorabkürzung

Bisher haben wir davon gesprochen, die Suche in einer bestimmten Tiefe abzubrechen und eine Alpha-Beta-Kürzung vorzunehmen, die nachweisbar keinen Effekt auf das Ergebnis hat (zumindest in Bezug auf die Werte der heuristischen Auswertung). Es ist außerdem möglich, eine **Vorabkürzung** vorzunehmen, d.h. bestimmte Züge an einem bestimmten Knoten sofort zu kürzen, ohne sie weiter in Betracht zu ziehen. Zweifellos betrachten die meisten Schachspieler ohnehin nur wenige Züge von jeder Position aus (zumindest bewusst). Ein Ansatz für die Vorabkürzung ist die **Strahlsuche**: Auf jeder Schicht betrachtet man nur einen „Strahl" der n besten Züge (entsprechend der Bewertungsfunktion), anstatt sämtliche möglichen Züge zu berücksichtigen. Leider ist dieser Ansatz recht gefährlich, weil es keine Garantie gibt, dass nicht gerade der beste Zug gekürzt wird.

Der PROBCUT-Algorithmus (für Probabilistic Cut, wahrscheinlichkeitsorientiertes Abschneiden) (Buro, 1995) ist eine Vorabkürzungsversion der Alpha-Beta-Suche. Er greift auf statistische Daten zurück, die aus vorherigen Erfahrungen gewonnen wurden, um die Gefahr zu verringern, den besten Zug abzuschneiden. Die Alpha-Beta-Suche kürzt jeden Knoten, der *nachweislich* außerhalb des aktuellen (α, β)-Fensters liegt. PROBCUT beschneidet auch Knoten, die *wahrscheinlich* außerhalb des Fensters liegen. Der Algorithmus ermittelt diese Wahrscheinlichkeit mithilfe einer flachen Suche, um den gespeicherten Wert v eines Knotens zu berechnen und dann anhand der vergangenen Erfahrung abzuschätzen, wie wahrscheinlich es ist, dass ein Punktestand von v bei Tiefe d im Baum außerhalb von (α, β) liegt. Buro wandte diese Technik auf sein Othello-Programm LOGISTELLO an und stellte fest, dass eine Version seines Programms mit PROBCUT die normale Version in 64% der Zeit schlug, selbst wenn der normalen Version doppelt so viel Zeit eingeräumt wurde.

Kombiniert man alle hier beschriebenen Techniken, erhalten wir ein Programm, das bemerkenswert Schach (oder andere Spiele) spielt. Angenommen, wir haben eine Bewertungsfunktion für Schach, einen sinnvollen Abbruchtest mit Ruhesuche und eine große Transpositionstabelle implementiert. Nehmen wir weiterhin an, dass wir nach Monaten mühevoller Bitschiebereien auf einem neuen PC etwa eine Million Knoten pro Sekunde erzeugen und bewerten können, sodass wir etwa 200 Millionen Knoten pro Zug nach Standardzeitregeln (drei Minuten pro Zug) durchsuchen können. Der Verzweigungsfaktor für Schach beträgt durchschnittlich etwa 35, und 35^5 ist etwa 50 Millionen. Wenn wir also eine Minimax-Suche anwenden, könnten wir nur fünf Schichten voraussehen. Ein solches Programm ist zwar nicht inkompetent, könnte aber leicht von einem durchschnittlichen menschlichen Spieler geschlagen werden, der in der Regel sechs oder acht Schichten vorausplanen kann. Mit einer Alpha-Beta-Suche schaffen wir etwa zehn Schichten, was zu einem Profispielerniveau führt. *Abschnitt 5.8* beschreibt weitere Kürzungstechniken, die die effektive Suchtiefe auf etwa 14 Schichten erweitern können. Um einen Großmeisterstatus zu erreichen, bräuchten wir eine stark verbesserte Bewertungsfunktion und eine große Datenbank mit Eröffnungen und Endspielzügen.

5.4.4 Suche und Nachschlagen

Irgendwie scheint es wie ein Overkill für ein Schachprogramm zu sein, ein Spiel damit zu beginnen, einen Baum mit einer Milliarde Spielzuständen zu betrachten, nur um zum Schluss zu kommen, dass der erste Zug „Bauer e4“ lautet. Schachbücher, die gute Eröffnungs- und Endspiele beschreiben, gibt es bereits seit einem Jahrhundert (Tattersall, 1911). Es überrascht demnach nicht, dass viele Spieleprogramme für Eröffnungen und Endspiele mit Nachschlagetabellen statt per Suche arbeiten.

Für die Eröffnungen stützt sich der Computer vor allem auf das menschliche Know-how. Die beste Empfehlung menschlicher Experten zum Ablauf von Eröffnungen wird aus Büchern kopiert und in Tabellen eingegeben, mit denen der Computer dann arbeitet. Allerdings können Computer auch statistische Daten aus einer Datenbank mit vorher gespielten Partien übernehmen, um festzustellen, welche Eröffnungssequenzen am häufigsten zu einem Gewinn geführt haben. In der Anfangsphase des Spieles gibt es hinsichtlich der Züge nur wenige Auswahlmöglichkeiten, dafür aber umfangreiche Expertenkommentare und Erfahrungen aus vergangenen Spielen. Nach üblicherweise

zehn Zügen taucht dann schon eine seltenere Position auf und das Programm muss vom Nachschlagen zur Suche übergehen.

Gegen Ende des Spieles gibt es weniger mögliche Positionen und somit mehr Gelegenheit, per Nachschlagen zu arbeiten. Doch hier ist es der Computer, der das Know-how besitzt: Die Computeranalyse von Endspielen geht weit über das hinaus, was durch den Menschen erreicht wurde. Ein Mensch kann die allgemeine Strategie für ein Endspiel König und Turm gegen König (KTK) angeben: die Beweglichkeit des gegnerischen Königs einschränken, indem man ihn gegen den Rand des Brettes drückt und mit dem eigenen König am Entkommen aus dieser Enge hindert. Andere Endspiele wie zum Beispiel König, Läufer und Springer gegen König (KLSK) sind schwer zu beherrschen und haben keine prägnante Strategiebeschreibung. Ein Computer kann dagegen das Endspiel vollständig lösen, indem er eine **Richtlinie** erzeugt, die eine Zuordnung von jedem möglichen Zustand zum besten Zug in diesem Zustand darstellt. Dann können wir einfach den besten Zug nachschlagen, anstatt ihn erneut berechnen zu müssen. Wie groß wird die Nachschlagetabelle für KLSK-Endspiele sein? Es gibt 462 Möglichkeiten, die beiden Könige auf dem Brett zu platzieren, ohne dass sie direkt benachbart sind. Nachdem die Könige aufgestellt sind, gibt es 62 leere Felder für den Läufer, 61 für den Springer und zwei mögliche Spieler, um den nächsten Zug auszuführen, sodass sich $462 \times 62 \times 61 \times 2 = 3.494.568$ mögliche Positionen ergeben. Einige davon sind Schachmattstellungen, die Sie als solche in einer Tabelle markieren. Anschließend praktizieren Sie eine **retrograde** Minimax-Suche: Kehren Sie die Schachregeln um und führen Sie Rückgängig-Züge statt regulärer Züge aus. Jeder Zug von Weiß, der unabhängig vom schwarzen Antwortzug in einer als Gewinn markierten Position endet, muss auch ein Gewinn sein. Diese Suche setzen Sie fort, bis sämtliche 3.494.568 Positionen als Gewinn, Verlust oder Remis aufgelöst sind, und schon haben Sie eine unfehlbare Nachschlagetabelle für alle KLSK-Endspiele.

Mithilfe dieser Technik und einem *Meisterstück* von Optimierungskniffen haben Ken Thompson (1986, 1996) und Lewis Stiller (1992, 1996) sämtliche Schachendspiele mit bis zu fünf Steinen und einige mit sechs Steinen gelöst und sie im Internet bereitgestellt. Stiller entdeckte einen Fall, bei dem ein zwingendes Matt existierte, aber 262 Züge erforderlich waren. Das verursachte einige Betroffenheit, da entsprechend der Schachregeln innerhalb von 50 Zügen ein Stein geschlagen oder ein Bauer gezogen werden muss. Eine spätere Arbeit von Marc Bourzutschky und Yakov Konoval (Bourzutschky, 2006) löste alle bauernlosen Endspiele mit sechs Steinen und einige mit sieben Steinen. Es gibt auch ein KDSKTLS-Endspiel, das bei bestem Spiel 517 Züge bis zu einem Schlagen benötigt, was dann zu einem Matt führt.

Könnte man die Tabellen für Schachendspiele von 6 auf 32 Steine erweitern, würde Weiß beim Eröffnungszug wissen, ob das Spiel mit Gewinn, Verlust oder Remis endet. Für Schach konnte dies bisher noch nicht realisiert werden, doch für das Damespiel ist dies bereits passiert, wie der Abschnitt mit den historischen Anmerkungen erläutert.

5.5 Stochastische Spiele

Im realen Leben können uns viele unvorhersagbare Ereignisse in unvorhergesehene Situationen bringen. Viele Spiele reflektieren diese Unvorhersehbarkeit durch die Einführung eines Zufallselements, wie beispielsweise mithilfe eines Würfels. Wir sprechen deshalb von **stochastischen Spielen**. Backgammon ist ein typisches Beispiel für ein Spiel, das Zufall und Geschicklichkeit kombiniert. Vor dem Zug eines Spielers wird gewürfelt, um die erlaubten Züge festzulegen. Zum Beispiel hat Weiß in der in ▶ Abbildung 5.10 gezeigten Backgammon-Position 6-5 gewürfelt und es gibt vier mögliche Züge.

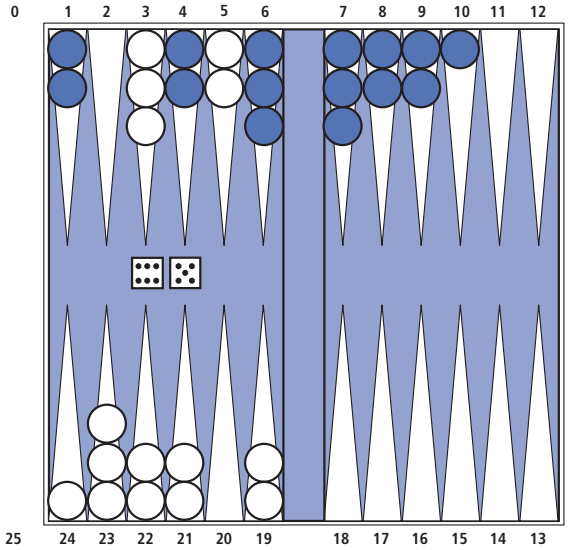


Abbildung 5.10: Eine typische Backgammon-Position. Ziel des Spieles ist es, alle Steine vom Brett zu entfernen. Weiß zieht im Uhrzeigersinn nach 25, Schwarz zieht gegen den Uhrzeigersinn nach 0. Eine Figur kann sich an jede beliebige Position bewegen, es sei denn, es befinden sich mehrere Steine des Gegners dort; befindet sich ein Stein des Gegners dort, wird dieser geschlagen und muss von Neuem beginnen. In der gezeigten Position hat Weiß 6-5 gewürfelt und kann unter vier erlaubten Zügen wählen: (5-10, 5-11), (5-11, 19-24), (5-10, 10-16) und (5-11, 11-16), wobei die Notation (5-11, 11-16) bedeutet, dass ein Stein von Position 5 nach 11 und dann ein Stein von 11 nach 16 bewegt wird.

Obwohl Weiß seine eigenen erlaubten Züge kennt, weiß es nicht, was Schwarz würfeln wird, und damit auch nicht, welche erlaubten Züge ihm zur Verfügung stehen. Das bedeutet, Weiß kann keinen Standardspielbaum der Art erstellen, wie wir ihn beim Schach und beim Tic Tac Toe gesehen haben. Ein Spielbaum im Backgammon muss neben den MAX- und MIN-Knoten auch **Zufallsknoten** enthalten. Zufallsknoten sind in ▶ Abbildung 5.11 als Kreise dargestellt. Die Verzweigungen, die von jedem Zufallsknoten aus abgehen, kennzeichnen die möglichen Würfelaugen und jede davon ist mit dem Würfelauge und der Wahrscheinlichkeit, dass sie auftritt, beschriftet. Es gibt 36 Möglichkeiten, zwei Würfel zu werfen, die alle gleich wahrscheinlich sind; weil jedoch 6-5 dasselbe ist wie 5-6, gibt es nur 21 verschiedene Würfe. Die sechs Pasche (1-1 bis 6-6) haben die Wahrscheinlichkeit von 1/36, bei den anderen 15 Würfen beträgt die Wahrscheinlichkeit 1/18.

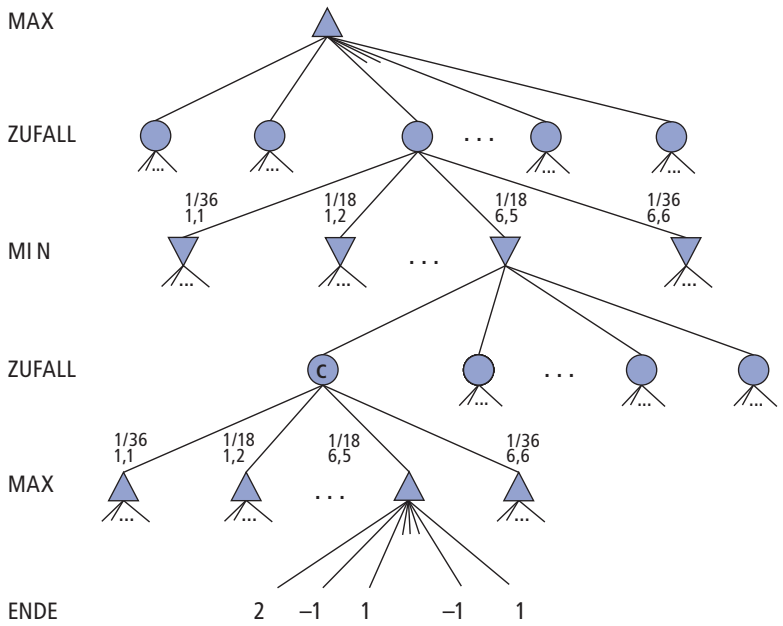


Abbildung 5.11: Skizze eines Spielbaumes für eine Backgammon-Position.

Der nächste Schritt ist, zu verstehen, wie man korrekte Entscheidungen trifft. Offensichtlich wollen wir immer noch den Zug auswählen, der zur besten Position führt. Allerdings haben Positionen keine definitiven Minimax-Werte. Stattdessen können wir nur den **Erwartungswert** einer Position berechnen: den Mittelwert über alle möglichen Ergebnisse der Zufallsknoten.

Das führt dazu, dass wir den Minimax-Wert für deterministische Spiele zu einem **erwarteten Minimax-Wert** (**Expectiminimax-Wert**) für Spiele mit Zufallsknoten verallgemeinern. Endknoten sowie MAX- und MIN-Knoten (für die die Würfelzahlen bekannt sind) arbeiten genau wie zuvor. Für Zufallsknoten berechnen wir den Erwartungswert, der die Summe des Wertes über alle Ergebnisse, gewichtet nach der Wahrscheinlichkeit jeder Zufallsaktion, darstellt:

$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \textit{wenn } \text{TERMINAL-TEST}(s) \\ \max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \textit{wenn } \text{PLAYER}(s)=\text{MAX} \\ \min_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \textit{wenn } \text{PLAYER}(s)=\text{MIN} \\ \sum_r P(r)\text{EXPECTIMINIMAX}(\text{RESULT}(s,r)) & \textit{wenn } \text{PLAYER}(s)=\text{CHANCE} \end{cases}$$

Hierin stellt r ein mögliches Würfelergebnis (oder ein anderes Zufallsereignis) dar und RESULT(s, r) liefert den gleichen Zustand wie s, wobei zusätzlich gilt, dass r das Würfelergebnis ist.

5.5.1 Bewertungsfunktionen für Zufallsspiele

Wie bei MINIMAX ist die offensichtliche Abschätzung bei EXPECTIMINIMAX, die Suche an irgendeinem Punkt abzubrechen und auf jedes Blatt eine Bewertungsfunktion anzuwenden. Man möchte meinen, dass sich Bewertungsfunktionen für Spiele wie etwa Backgammon nicht von Bewertungsfunktionen für Schach unterscheiden – sie müssen besseren Positionen einfach nur höhere Punkte zuweisen. Tatsächlich bedeutet das Vorliegen von Zufallsknoten jedoch, dass man noch sorgfältiger darauf achten muss, was die Werte einer Bewertungsfunktion bedeuten. ▶ Abbildung 5.12 zeigt, was passiert: Bei einer Bewertungsfunktion, die den Blättern die Werte [1, 2, 3, 4] zuweist, ist der Zug a_1 der beste; bei den Werten [1, 20, 30, 400] ist der Zug a_2 der beste. Das Programm verhält sich also völlig anders, wenn wir die Skalierung einiger Bewertungswerte ändern! Um diese Empfindlichkeit zu vermeiden, muss die Bewertungsfunktion eine positive lineare Transformation der Wahrscheinlichkeit, von einer Position aus zu gewinnen, sein (oder allgemeiner gesagt, von dem erwarteten Nutzen der Position). Das ist eine wichtige und allgemeine Eigenschaft von Situationen mit einer Unsicherheitskomponente, wie *Kapitel 16* noch ausführlicher erläutert.

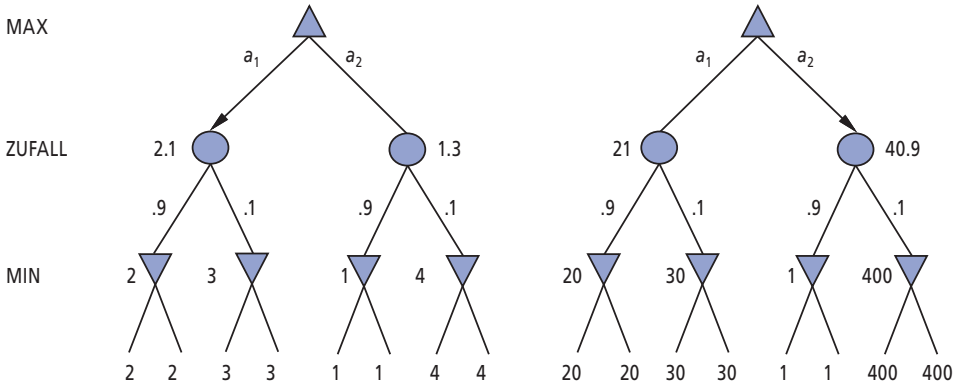


Abbildung 5.12: Eine die Reihenfolge beibehaltende Transformation der Blattwerte ändert den besten Wert.

Würde das Programm alle Würfelergebnisse, die im restlichen Spiel auftreten, im Voraus kennen, wäre die Lösung eines Spieles mit Würfel gleich dem Lösen eines Spieles ohne Würfel, was Minimax in einer Zeit von $O(b^m)$ erledigt, wobei b der Verzweigungsfaktor und m die maximale Tiefe des Spielbaumes sind. Weil EXPECTIMINIMAX auch alle möglichen Würfelfolgen berücksichtigt, benötigt es die Zeit $O(b^m n^m)$, wobei n die Anzahl der verschiedenen Würfe ist.

Selbst wenn die Suchtiefe auf einen kleinen Wert d beschränkt bleibt, ist es bei Glücksspielen aufgrund der zusätzlichen Kosten im Vergleich zu denen für Minimax unrealistisch, sehr weit vorauszusehen. Beim Backgammon ist n gleich 21 und b liegt normalerweise bei etwa 20, kann aber in einigen Situationen – für Paschwürfe – auf 4000 steigen. Drei Schichten sind das höchste, was wir bewältigen können.

Es gibt noch eine Möglichkeit, sich das Problem vorzustellen: Der Vorteil von Alpha-Beta ist, dass dabei weitere Entwicklungen ignoriert werden, die beim bestmöglichen Spiel nicht passieren werden. Es konzentriert sich also nur auf das, was wahrscheinlich auftritt. In Spielen mit Würfeln gibt es *keine* wahrscheinliche Zugfolge, denn damit diese Züge stattfinden, müssen die Würfel zuerst die richtigen Augen zeigen,

damit sie erlaubt sind. Das ist ein allgemeines Problem, wenn Unsicherheit mit ins Spiel kommt: Die Möglichkeiten werden extrem multipliziert und die detaillierte Planung von Aktionen wird sinnlos, weil die Welt dabei vielleicht nicht mitspielt.

Zweifellos ist dem Leser aufgefallen, dass vielleicht etwas wie Alpha-Beta-Kürzung auf Spielbäume mit Zufallsknoten angewendet werden könnte. Und das erweist sich als richtig. Die Analyse für MIN- und MAX-Knoten bleibt unverändert, aber mit ein bisschen Einfallsreichtum können wir auch Zufallsknoten kürzen. Betrachten Sie den Zufallsknoten C in Abbildung 5.11 und was mit seinem Wert passiert, während wir seine untergeordneten Knoten untersuchen und bewerten. Ist es möglich, eine Obergrenze für den Wert von C zu finden, bevor wir alle seine untergeordneten Knoten betrachtet haben? (Bekanntlich braucht Alpha-Beta genau das, um einen Knoten und seinen Unterbaum zu kürzen.) Auf den ersten Blick scheint es vielleicht unmöglich, weil der Wert von C der *Mittelwert* der Werte seiner untergeordneten Knoten ist, und um den Mittelwert einer Menge von Zahlen zu berechnen, müssen wir sämtliche Zahlen betrachten. Doch wenn wir Grenzen für die möglichen Werte der Nutzenfunktion festlegen, können wir auch zu Grenzen für den Mittelwert gelangen, ohne jede Zahl ansehen zu müssen. Wenn wir beispielsweise sagen, dass alle Nutzenwerte zwischen -2 und $+2$ liegen, ist der Wert von Blattknoten begrenzt und wir *können* eine Obergrenze für den Wert eines Zufallsknotens angeben, ohne alle seine untergeordneten Knoten zu betrachten.

Alternativ ließe sich mit der **Monte-Carlo-Simulation** eine Position auswerten. Dabei beginnt man mit einem Alpha-Beta- (oder einem anderen) Suchalgorithmus. Von einer Ausgangsposition an lässt man den Algorithmus Tausende von Spielen gegen sich selbst spielen, wobei die Würfelergebnisse zufällig erzeugt werden. Beim Backgammon zeigt sich, dass der prozentuale Gewinnanteil einen guten Näherungswert für den Wert der Position darstellt, selbst wenn der Algorithmus mit einer nicht perfekten Heuristik arbeitet und nur einige Schichten durchsucht (Tesauro, 1995). Für Spiele mit Würfeln bezeichnet man eine derartige Simulation als **Rollout**.

5.6 Teilweise beobachtbare Spiele

Schach wurde oftmals als Krieg en miniature beschrieben, doch fehlt diesem Spiel mindestens ein Wesensmerkmal realer Kriege – die **teilweise Beobachtbarkeit**. Im „Nebel des Krieges“ (Fog of War, FoW) sind die Existenz und die Aufstellung der feindlichen Einheiten oftmals unbekannt, bis sie durch direkten Kontakt aufgedeckt werden. Deshalb gehören zu einer Kriegführung auch Aufklärer und Spione, um Informationen zu sammeln, sowie Geheimhaltung und Täuschung, um den Feind zu verwirren. Teilweise beobachtbare Spiele weisen diese Charakteristika auf und unterscheiden sich somit qualitativ von den Spielen, die die vorherigen Abschnitte beschrieben haben.

5.6.1 Kriegspiel: teilweise beobachtbares Schach

In *deterministisch* teilweise beobachtbaren Spielen rührt die Unsicherheit über den Zustand des Brettes ausschließlich vom fehlenden Zugriff auf die vom Gegner getroffenen Entscheidungen her. Zu dieser Klasse gehören einfache Spiele wie *Schiffe versenken* (bei dem die Schiffe jedes Spielers an Positionen platziert werden, die dem Gegner verborgen bleiben, sich aber auch nicht ändern) und *Stratego* (bei dem Stein-

positionen bekannt sind, die Steinarten jedoch verborgen bleiben). Wir untersuchen hier das Spiel **Kriegspiel**, eine teilweise beobachtbare Variante von Schach, bei der Steine ziehen können, für den Gegner jedoch vollkommen unsichtbar sind.

Die Regeln von Kriegspiel sehen folgendermaßen aus: Weiß und Schwarz sehen jeweils ein Brett, das nur ihre eigenen Steine enthält. Ein Schiedsrichter, der alle Steine sehen kann, trifft Spielentscheidungen und gibt regelmäßig Kommentare ab, die von beiden Spielern gehört werden. Weiß am Zug schlägt dem Schiedsrichter einen Zug vor, der zulässig wäre, wenn es keine schwarzen Steine gäbe. Ist der Zug tatsächlich (wegen der schwarzen Steine) nicht zulässig, kommentiert das der Schiedsrichter als „illegal“. In diesem Fall kann Weiß weitere Züge vorschlagen, bis ein zulässiger Zug gefunden wurde – dabei lernt Weiß mehr über die Positionen der schwarzen Steine. Sobald ein zulässiger Zug vorgeschlagen wurde, gibt der Schiedsrichter einen oder mehrere der folgenden Kommentare: „Schlagen auf Feld *X*“, wenn eine Figur geschlagen wurde, und „Schach durch *D*“, wenn der schwarze König im Schach steht, wobei *D* die Richtung des Schachgebotes ist und durch „Springer“, „Rang“, „Linie“, „lange Diagonale“ oder „kurze Diagonale“ ersetzt werden kann. (Bei einem Abzugsschach kann der Schiedsrichter zwei „Schach“-Meldungen ansagen.) Wenn Schwarz schachmatt gesetzt oder zugunfähig ist, teilt das der Schiedsrichter entsprechend mit. Andernfalls ist Schwarz am Zug.

Kriegspiel scheint entsetzlich kompliziert zu sein, doch Menschen beherrschen es recht gut und Computerprogramme holen bereits auf. Es hilft dabei, das Konzept eines **Belief State** zu wiederholen, wie er in *Abschnitt 4.4* definiert und in Abbildung 4.14 veranschaulicht wurde – die Menge aller *logisch möglichen* Brettzustände bei gegebenem vollständigen Verlauf der Wahrnehmungen bis zu diesem Zeitpunkt. Anfangs ist der Belief State von Weiß ein Singleton, da die schwarzen Steine noch nicht bewegt wurden. Nachdem Weiß gezogen und Schwarz geantwortet hat, enthält der Belief State von Weiß 20 Positionen, da Schwarz auf jeden weißen Zug 20 Antworten hat. Das Verfolgen des Belief State im weiteren Spielverlauf ist exakt das Problem der **Zustandsabschätzung**, für die *Gleichung (4.6)* den Aktualisierungsschritt angibt. Die Kriegspiel-Zustandsabschätzung können wir direkt auf das teilweise beobachtbare, nichtdeterministische Gerüst von *Abschnitt 4.4* abbilden, wenn wir den Gegenspieler als Quelle des Nichtdeterminismus betrachten; d.h., RESULTS des Zuges von Weiß wird aus dem (vorhersagbaren) Ergebnis des eigenen Zuges von Weiß und des nichtvorhersagbaren Ergebnisses aus der Antwort von Schwarz zusammengesetzt.[3]

Für einen aktuellen Belief State könnte Weiß fragen: „Kann ich das Spiel gewinnen?“ Für ein teilweise beobachtbares Spiel wird das Konzept einer **Strategie** geändert. Anstatt einen auszuführenden Zug zu spezifizieren für jeden möglichen Zug, den der Gegenspieler unternehmen könnte, brauchen wir einen Zug für jede mögliche Wahrnehmungsfolge, die empfangen werden könnte. Für Kriegspiel ist eine gewinnende Strategie oder **garantiertes Schachmatt** eine Strategie, die für jede mögliche Wahrnehmungsfolge zu einem tatsächlichen Schachmatt für jeden möglichen Brettzustand im aktuellen Belief State führt, unabhängig davon, wie der Gegner zieht. Mit dieser Definition ist der Belief State des Gegners ohne Bedeutung – die Strategie muss funktionieren, selbst wenn der Gegner alle Steine sehen kann. Dadurch vereinfacht sich die Berechnung erheblich. ▶ Abbildung 5.13 zeigt einen Teil für ein garantiertes Schachmatt für das

3 Manchmal wird der Glaubenszustand zu groß, um lediglich als Liste von Brettzuständen dargestellt zu werden. Dieses Problem interessiert uns momentan jedoch nicht. *Kapitel 7* und *8* schlagen Methoden vor, mit denen sich Glaubenszustände kompakt darstellen lassen.

KTK-Endspiel (König und Turm gegen König). In diesem Fall besitzt Schwarz lediglich einen Stein (den König), sodass sich ein Belief State für Weiß auf einem einzigen Brett darstellen lässt, indem jede mögliche Position des schwarzen Königs markiert wird.

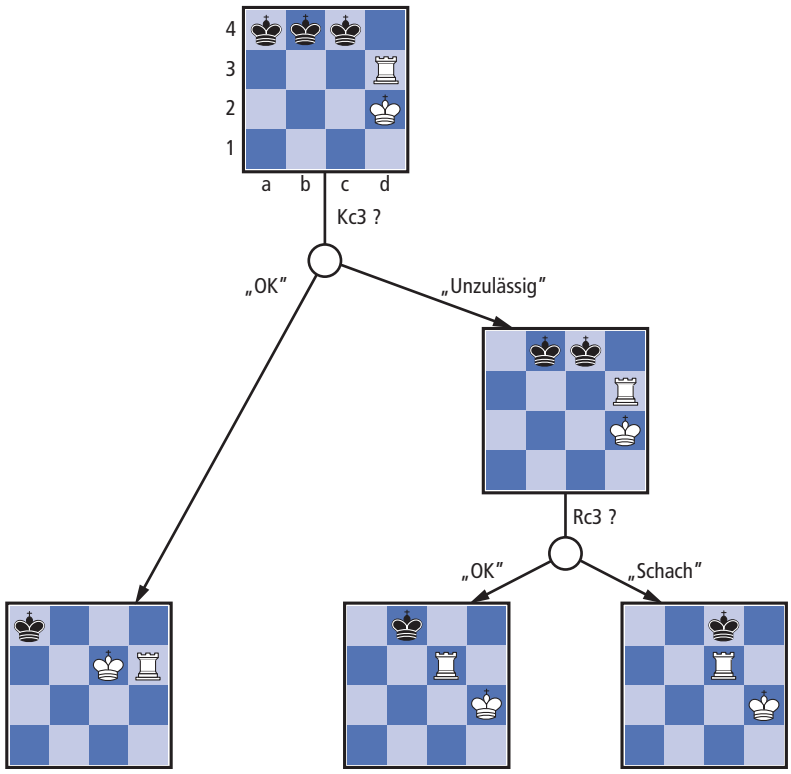


Abbildung 5.13: Auf einem verkleinerten Brett dargestellter Teil eines garantierten Schachmatt im KTK-Endspiel. Im anfänglichen Belief State befindet sich der schwarze König auf einer von drei möglichen Positionen. Durch eine Kombination von Sondierungszügen engt die Strategie dies auf eine Möglichkeit ein. Die Züge bis zum Schachmatt lassen wir dem Leser als Übung.

Der allgemeine AND-OR-Suchalgorithmus lässt sich auf den Belief-States-Raum anwenden, um garantierte Schachmatts zu finden, genau wie es *Abschnitt 4.4* beschreibt. Der in diesem Abschnitt erwähnte inkrementelle Belief-States-Algorithmus findet oftmals Mattfolgen im Mittelspiel bis zur Tiefe 9 – was wahrscheinlich weit über die Fähigkeiten menschlicher Spieler hinausgeht.

Neben garantierten Mattfolgen erlaubt Kriegspiel ein vollkommen neues Konzept, das in vollständig beobachtbaren Spielen keinen Sinn macht: **probabilistisches Schachmatt**. Derartige Mattfolgen müssen dennoch in jedem Brettzustand im Belief State funktionieren; sie sind probabilistisch in Bezug auf die Randomisierung der Züge des gewinnenden Spielers. Das Grundkonzept sei am Problem verdeutlicht, einen alleinstehenden schwarzen König nur mithilfe des weißen Königs ausfindig zu machen. Durch zufällige Züge wird der weiße König *irgendwann* auf den schwarzen König stoßen, selbst wenn dieser versucht, seinem Schicksal zu entgehen, da Schwarz nicht unendlich lange in der Lage ist, die richtigen Ausweichzüge zu erraten. Entsprechend der Terminologie in der

Wahrscheinlichkeitstheorie tritt die Entdeckung *mit der Wahrscheinlichkeit* 1 auf. Das KLSK-Endspiel – König, Läufer und Springer gegen König – wird in diesem Sinne gewonnen. Weiß konfrontiert Schwarz mit einer unendlichen Folge zufälliger Entscheidungen. Eine dieser Entscheidungen wird Schwarz falsch erraten und die eigene Position aufdecken, was zum Matt führt. Das KLLK-Endspiel wird dagegen mit der Wahrscheinlichkeit $1 - \varepsilon$ gewonnen. Weiß kann einen Gewinn nur erzwingen, wenn es seine Läufer für einen Zug ungeschützt lässt. Ist Schwarz in diesem Moment zur Stelle und schlägt den Läufer (ein Zug, der zum Verlust führt, wenn die Läufer gedeckt sind), endet das Spiel remis. Weiß kann sich für den gefährlichen Zug zu einem zufällig gewählten Zeitpunkt mitten in einer sehr langen Sequenz entscheiden und somit ε auf eine beliebig kleine Konstante – jedoch nicht bis auf 0 – verringern.

Es kommt recht selten vor, dass sich ein garantiertes oder probabilistisches Matt innerhalb einer vernünftigen Tiefe finden lässt, vom Endspiel einmal abgesehen. Manchmal funktioniert eine Mattstrategie für *einige* der Brettzustände im aktuellen Belief State, jedoch nicht in anderen. Es kann erfolgreich sein, eine derartige Strategie auszuprobieren, was zu einem **versehentlichen Schachmatt** führt – versehentlich in dem Sinne, dass Weiß nicht *wissen* kann, dass es Matt setzt – wenn sich die schwarzen Steine gerade an den richtigen Stellen befinden. (Die meisten Mattfolgen in Spielen zwischen Menschen zeichnen sich durch diesen zufälligen Charakter aus.) Diese Idee führt natürlich zu der Frage, wie wahrscheinlich es ist, dass eine bestimmte Strategie gewinnt, woraus sich wiederum die Frage ergibt, mit welcher Wahrscheinlichkeit jeder Brettzustand im aktuellen Belief State der wahre Brettzustand ist.

Tipp

Man könnte zunächst geneigt sein vorzuschlagen, dass alle Brettzustände im aktuellen Belief State gleich wahrscheinlich sind – doch dies kann nicht richtig sein. Betrachten Sie zum Beispiel den Belief State von Weiß nach dem ersten Spielzug. Gemäß Definition (und der Annahme, dass Schwarz optimal spielt) muss Schwarz einen optimalen Zug gespielt haben, sodass allen Brettzuständen, die aus suboptimalen Zügen resultieren, eine Wahrscheinlichkeit von null zugewiesen sein sollte. Dieses Argument ist aber auch nicht ganz richtig, da *das Ziel jedes Spielers nicht einfach darin besteht, Steine auf die richtigen Felder zu schieben, sondern auch, die Informationen zu minimieren, die der Gegner über die eigene Stellung erhält*. Das Spielen jeder *vorhersagbaren* „optimalen" Strategie liefert dem Gegner Informationen. Folglich erfordert optimales Spielen in teilweise beobachtbaren Spielen die Bereitschaft, ein wenig zufällig zu spielen. (Aus diesem Grund führen Hygieneinspektoren von Gaststätten ihre Besuche zufällig durch.) Das heißt, gelegentlich Züge auszuwählen, die „an sich" schwach sind – die aber wegen ihrer Unvorhersagbarkeit an Stärke gewinnen, da der Gegner höchstwahrscheinlich keine Verteidigung gegen sie vorbereitet hat.

Aus diesen Betrachtungen ergibt sich scheinbar, dass die den Brettzuständen im aktuellen Belief State zugeordneten Wahrscheinlichkeiten nur anhand einer optimal randomisierten Strategie berechnet werden können; andererseits scheint es für die Berechnung dieser Strategie erforderlich zu sein, die Wahrscheinlichkeiten der verschiedenen Zustände zu kennen, in denen sich das Brett befinden kann. Dieses Rätsel lässt sich auflösen, wenn man das spieltheoretische Konzept einer **Gleichgewichtslösung** akzeptiert, dem wir in *Kapitel 17* weiter nachgehen. Ein Gleichgewicht spezifiziert eine für jeden Spieler optimal randomisierte Strategie. Allerdings ist es selbst für kleine Spiele unerschwinglich teuer, Gleichgewichte zu berechnen, und für Kriegspiel indiskutabel. Das Design effektiver Algorithmen für allgemeines Kriegspiel ist derzeit noch ein offenes

Forschungsthema. Die meisten Systeme führen tiefenbegrenzte Nachschlagemethoden in ihrem eigenen Belief-States-Raum aus und ignorieren den Belief State des Gegners. Die Bewertungsfunktionen ähneln denen für das beobachtbare Spiel, schließen aber eine Komponente für die Größe des Belief State ein – kleiner ist besser!

5.6.2 Kartenspiele

Kartenspiele bieten viele Beispiele für *stochastisch* teilweise Beobachtbarkeit, wobei die fehlenden Informationen zufällig erzeugt werden. So werden in vielen Spielen zu Beginn die Karten zufällig ausgegeben, wobei jeder Spieler ein Blatt (d.h. eine je nach Spiel festgelegte Anzahl Karten) erhält, das für die anderen Spieler nicht einsehbar ist. Zu diesen Spielen gehören Bridge, Whist, Hearts sowie einige Formen von Poker.

Auf den ersten Blick scheinen Kartenspiele den Würfelspielen zu gleichen: Die Karten werden zufällig verteilt und bestimmen die Züge, die jeder Spieler machen kann, doch werden alle „Würfel“ bereits ganz am Anfang geworfen! Auch wenn sich diese Analogie als inkorrekt erweist, legt sie einen wirksamen Algorithmus nahe: alle möglichen Verteilungen der unsichtbaren Karten betrachten, jede auflösen, als würde es sich um ein vollständig beobachtbares Spiel handeln, und dann den Zug auswählen, der im Mittel über alle Verteilungen das beste Ergebnis liefert. Wir nehmen an, dass jede Verteilung s mit der Wahrscheinlichkeit $P(s)$ auftritt. Der gewünschte Zug ist dann

$$\arg\max_a \sum_s P(s) MINIMAX(RESULT(s,a)). \tag{5.1}$$

Hier führen wir exakt MINIMAX aus, wenn es rechentechnisch machbar ist, sonst H-MINIMAX.

In den meisten Kartenspielen ist nun die Anzahl der möglichen Verteilungen ziemlich groß. Zum Beispiel sieht beim Bridge jeder Spieler lediglich zwei von vier Blättern. Mit zwei nicht einsehbaren Blättern von jeweils 13 Karten gibt es also

$$\binom{26}{13} = 10.400.600.$$

Verteilungen. Da es allein schon schwierig ist, eine Verteilung aufzulösen, steht das Lösen von 10 Millionen Verteilungen außer Frage. Stattdessen greifen wir auf eine Monte-Carlo-Näherung zurück: Anstatt *alle* Verteilungen zu summieren, nehmen wir eine *zufällige Stichprobe* von N Verteilungen, wobei die Wahrscheinlichkeit der Verteilung s, die in der Stichprobe erscheint, proportional zu $P(s)$ ist:

$$\arg\max_a \frac{1}{N} \sum_{i=1}^{N} MINIMAX(RESULT(s_i,a)). \tag{5.2}$$

(Im Summenausdruck erscheint $P(s)$ nicht explizit, da die Proben bereits entsprechend $P(s)$ entnommen wurden.) Bei großen N strebt die Summe über der zufälligen Probe dem genauen Wert zu, doch selbst für relativ kleine N – etwa 100 bis 1000 – liefert die Methode eine gute Näherung. Mit einer vernünftigen Abschätzung von $P(s)$ lässt sich die Methode auch auf deterministische Spiele wie Kriegspiel anwenden.

Für Spiele in der Art von Whist und Hearts, bei denen es keine Gebots- oder Reizphase gibt, bevor das eigentliche Spiel beginnt, ist jede Verteilung gleichwahrscheinlich und somit sind die Werte von $P(s)$ alle gleich. Bei Bridge beginnt das Spiel mit einer Reizphase, in der jede Seite anzeigt, wie viele Stiche sie für einen Gewinn erwartet. Da die Spieler auf der Grundlage ihrer Handkarten bieten, erfahren die anderen Spieler mehr über die Wahrscheinlichkeit jeder Verteilung. Allerdings ist es aus den Gründen, wie sie bei Kriegspiel erläutert wurden, nicht trivial, die Reizgebote in die Entscheidung für die Spielweise des Blattes einzubeziehen: Spieler können so bieten, dass die Gegner möglichst wenige Informationen erhalten. Selbst dann ist dieser Ansatz für Bridge recht wirksam, wie *Abschnitt 5.7* noch zeigt.

Die in den Gleichungen 5.1 und 5.2 beschriebene Strategie wird manchmal auch als „Mittelwertbildung über Hellsehen“ bezeichnet, da sie davon ausgeht, dass das Spiel unmittelbar nach dem ersten Zug für beide Spieler beobachtbar wird. Trotz ihrer intuitiven Attraktivität kann die Strategie irreleiten. Folgende Geschichte soll das verdeutlichen:

> Tag 1: Straße *A* führt zu einem Goldhaufen; Straße *B* führt zu einer Gabelung. Zweigen Sie an der Gabelung nach links ab, finden Sie einen größeren Goldhaufen, wenn Sie aber die rechte Abzweigung nehmen, werden Sie von einem Bus überfahren.
>
> Tag 2: Straße *A* führt zu einem Goldhaufen; Straße *B* führt zu einer Gabelung. Zweigen Sie an der Gabelung nach rechts ab, finden Sie einen größeren Goldhaufen, wenn Sie aber die linke Abzweigung nehmen, werden Sie von einem Bus überfahren.
>
> Tag 3: Straße *A* führt zu einem Goldhaufen; Straße *B* führt zu einer Gabelung. Ein Abzweig der Gabelung führt zu einem größeren Goldhaufen, doch wenn Sie den falschen Abzweig wählen, werden Sie von einem Bus überfahren.
>
> Leider wissen Sie nicht, um welche Gabelung es sich handelt.

Die Mittelwertbildung über Hellsehen führt zur folgenden Argumentation: Am Tag 1 ist *B* die richtige Wahl; am Tag 2 ist *B* die richtige Wahl; am Tag 3 ist die Situation die gleiche wie am Tag 1 oder am Tag 2, sodass *B* immer noch die richtige Wahl sein muss.

Jetzt zeigt sich, wie Mittelwertbildung über Hellsehen scheitert: Es berücksichtigt nicht den *Belief State*, in dem sich der Agent befinden wird, nachdem er gehandelt hat. Ein Belief State völliger Unkenntnis ist nicht wünschenswert, insbesondere wenn eine Möglichkeit den sicheren Tod bedeutet. Da das Konzept davon ausgeht, dass jeder zukünftige Zustand automatisch ein Zustand mit perfektem Wissen sein wird, wählt es weder Aktionen aus, die *Informationen sammeln* (wie der erste Zug in Abbildung 5.13), noch Aktionen, die Informationen vor dem Gegner verbergen oder einem Partner Informationen liefern, weil es annimmt, dass diese Parteien bereits die Informationen kennen. Und es wird auch niemals beim Poker **bluffen**[4], weil es davon ausgeht, dass die eigenen Karten auch vom Gegner gesehen werden können. In *Kapitel 17* zeigen wir, wie sich Algorithmen konstruieren lassen, die alle diese Dinge bewerkstelligen, indem sie das echte teilweise beobachtbare Entscheidungsproblem lösen.

4 Bluffen – Wetten, als ob das Blatt gut wäre, selbst wenn das nicht zutrifft – ist ein Kernbestandteil der Pokerstrategie.

5.7 Hochklassige Spielprogramme

Der russische Mathematiker Alexander Kronrod hat 1965 Schach als „Drosophila der künstlichen Intelligenz“ bezeichnet. John McCarthy widerspricht: Während die Genetiker mithilfe von Fruchtfliegen Entdeckungen machen, die sich auf die Biologie im breiteren Maßstab anwenden lassen, hat die KI mithilfe von Schach etwas unternommen, was dem Züchten von sehr schnellen Fruchtfliegen äquivalent ist. Als vielleicht bessere Analogie könnte man sagen, dass Schach für die künstliche Intelligenz das ist, was Grand-Prix-Rennen für die Autoindustrie sind: Spieleprogramme auf dem neuesten Stand der Technik arbeiten bestechend schnell auf stark optimierten Maschinen, die sehr komplexe Technik enthalten, aber sie sind nicht besonders nützlich, um Einkäufe damit zu erledigen oder im Gelände zu fahren. Ungeachtet dessen sind Rennen und Spielen spannend und erzeugen einen steten Strom von Innovationen, die von der breiteren Gemeinschaft übernommen wurden. Dieser Abschnitt zeigt, worauf es ankommt, um in den verschiedenen Spielen überlegen zu sein.

Schach: Das mittlerweile pensionierte Schachprogramm DEEP BLUE ist bekannt dafür, den Weltmeister Garry Kasparov in einem spektakulären Schaukampf besiegt zu haben. DEEP BLUE lief auf einem Parallelcomputer mit 30 RS/6000-Prozessoren von IBM, die eine Alpha-Beta-Suche ausführten, und zeichnete sich durch eine Konfiguration von 480 speziellen VLSI-Schachprozessoren aus, die die Züge und die Zugreihenfolge für die letzten Ebenen des Baumes erzeugten und die Blattknoten bewerteten. Die Suchgeschwindigkeit von DEEP BLUE erreichte bis zu 30 Milliarden Positionen pro Zug, wobei routinemäßig die Tiefe 14 erreicht wurde. Der Schlüssel zum Erfolg scheint seine Fähigkeit zu sein, einzelne Erweiterungen über die Tiefenbeschränkung hinaus vorzunehmen, um ausreichend interessante Folgen zwingender/erzwungener Züge zu bilden. In einigen Fällen erreichte die Suche eine Tiefe von 40 Schichten. Die Bewertungsfunktion hatte über 8000 Funktionsmerkmale, die zu einem großen Teil sehr spezifische Figurenmuster beschrieben. Ein „Eröffnungsbuch“ mit etwa 4000 Positionen wurde ebenso verwendet wie eine Datenbank mit 700.000 Großmeisterspielen, aus denen Konsensempfehlungen extrahiert werden konnten. Darüber hinaus verwendete das System eine große Endspieldatenbank mit gelösten Positionen, die alle Positionen mit fünf Figuren und viele mit sechs Figuren enthielt. Durch diese Datenbank ließ sich die effektive Suchtiefe wesentlich erweitern, sodass DEEP BLUE in einigen Fällen perfekt spielte, auch wenn er noch viele Züge vom Schachmatt entfernt war.

Der Erfolg von DEEP BLUE verstärkte den allgemein herrschenden Glauben, dass der Erfolg von Spiele spielenden Computern hauptsächlich der immer leistungsfähigeren Hardware zu verdanken sei – eine Ansicht, die von IBM bestärkt wurde. Doch haben letztlich die algorithmischen Verbesserungen dazu geführt, dass sich die Welt-Computerschachmeisterschaften mit Programmen gewinnen lassen, die auf Standard-PCs laufen. Mithilfe vielfältiger Kürzungsheuristiken ist es möglich, den effektiven Verzweigungsfaktor auf weniger als 3 zu reduzieren (verglichen mit dem tatsächlichen Verzweigungsfaktor von etwa 35). Die wichtigste davon ist die **Null-Zug-Heuristik**, die eine gute Untergrenze für den Wert einer Position ermittelt und dabei eine flache Suche verwendet, bei der der Gegner am Anfang zweimal zieht. Diese Untergrenze erlaubt häufig eine Alpha-Beta-Kürzung ohne den Aufwand einer Suche mit voller Tiefe. Ebenso wichtig ist die **Nutzlosigkeitskürzung**, mit der sich im Voraus entscheiden lässt, welche Züge zu einem Beta-Abbruch in den Nachfolgerknoten führen.

Als Nachfolger von DEEP BLUE kann HYDRA angesehen werden. Dieses Programm läuft auf einem Cluster aus 64 Prozessoren mit 1 GB Arbeitsspeicher je Prozessor und mit spezieller Hardware in Form von FPGA (Field Programmable Gate Array)-Chips. Mit rund 200 Millionen Bewertungen je Sekunde liegt HYDRA in der Größenordnung von DEEP BLUE, erreicht aber eine Tiefe von 18 Schichten gegenüber von nur 14 bei DEEP BLUE, was auf die aggressive Verwendung der Null-Zug-Heuristik und Vorwärtskürzung zurückzuführen ist.

Der Gewinner der Welt-Computerschachweltmeisterschaften von 2008 und 2009 heißt RYBKA und gilt als der derzeit stärkste Computerspieler. RYBKA verwendet einen standardmäßigen 8-Kern-Xeon-Prozessor mit 3,2 GHz von Intel, wobei aber über das Konzept des Programms nur wenig bekannt ist. Der wichtigste Vorteil von RYBKA scheint in seiner Bewertungsfunktion zu liegen, die von ihrem Hauptentwickler, dem internationalen Meister Vasik Rajlich und mindestens drei anderen Großmeistern optimiert wurde.

Wie die jüngsten Turniere zeigen, sind die Spitzencomputerschachprogramme inzwischen an allen menschlichen Rivalen vorbeigezogen. (Details hierzu finden Sie unter den historischen Anmerkungen.)

Dame: Jonathan Schaeffer und seine Kollegen entwickelten CHINOOK, das auf normalen PCs läuft und mit Alpha-Beta-Suche arbeitet. CHINOOK schlug 1990 den langjährigen menschlichen Weltmeister in einem abgekürzten Spiel und ist seit 2007 in der Lage, mithilfe der Alpha-Beta-Suche in Kombination mit einer Datenbank von 39 Billionen Endspielpositionen perfekt zu spielen.

Othello, auch als **Reversi** bezeichnet, ist wahrscheinlich als Computerspiel bekannter denn als Brettspiel. Es hat einen kleineren Suchraum als Schach und in der Regel 5 bis 15 erlaubte Züge, wobei aber die Bewertungserfahrung von Grund auf neu entwickelt werden musste. 1997 besiegte das Programm LOGISTELLO (Buro, 2002) den menschlichen Weltmeister, Takeshi Murakami, mit sechs zu null Spielen. Allgemein ist klar, dass ein Mensch bei Othello keine Chance gegen den Computer hat.

Backgammon: *Abschnitt 5.5* hat erklärt, warum die Berücksichtigung der Unsicherheit durch Würfeln eine tiefe Suche zu einem teuren Luxus macht. Die meisten Arbeiten zu Backgammon haben versucht, die Bewertungsfunktion zu verbessern. Gerry Tesauro (1992) kombinierte die verstärkende Lernmethode von Samuel mit Techniken neuronaler Netze, um eine bemerkenswert exakte Bewertung zu entwickeln, die für eine Suche bis zur Tiefe 2 oder 3 verwendet wird. Nach mehr als einer Million Trainingsspielen ist das Programm TD-GAMMON von Tesauro den besten Spielern der Welt ebenbürtig. Die Meinungen des Programms zu einigen der Eröffnungszüge für das Spiel haben in einigen Fällen das althergebrachte Wissen radikal umgekrempelt.

Go ist das beliebteste Brettspiel in Asien. Weil das Brett 19 x 19 Felder hat und Züge in (nahezu) jedes leere Feld erlaubt sind, beginnt der Verzweigungsfaktor bei 361, was für reguläre Alpha-Beta-Suchmethoden zu viel ist. Außerdem ist es schwierig, eine Bewertungsfunktion zu schreiben, da die Kontrolle von Gebieten bis zum Endspiel oftmals unvorhersagbar ist. Demzufolge verzichten die Spitzenprogramme wie zum Beispiel MOGO auf die Alpha-Beta-Suche und verwenden stattdessen Monte Carlo-Rollouts. Der Trick besteht darin zu entscheiden, welche Züge im Verlauf des Rollouts zu machen sind. Es gibt keine aggressive Kürzung; alle Züge sind möglich. Die Methode UCT (Upper Confidence bounds on Trees, Obere Konfidenzgrenzen bei Bäumen) wählt zufällige Züge in den ersten Iterationen aus und steuert mit der Zeit die Probenauswahl, um Züge zu bevorzugen, die in vorherigen Proben zu Gewinnen geführt haben. Dazu kom-

men noch einige Feinheiten einschließlich wissensbasierter Regeln, die besondere Züge vorschlagen, wenn ein bestimmtes Muster erkannt wird, und eingeschränkte lokale Suche, um taktische Fragen zu entscheiden. Darüber hinaus binden manche Programme spezielle Techniken aus der **kombinatorischen Spieltheorie** ein, um Endspiele zu analysieren. Diese Techniken zerlegen eine Position in Unterpositionen, die sich separat analysieren und dann kombinieren lassen (Berlekamp und Wolfe, 1994; Müller, 2003). Die auf diese Weise erhaltenen optimalen Lösungen haben viele Go-Berufsspieler überrascht, die bis dahin der Meinung waren, optimal gespielt zu haben. Derzeitige Go-Programme spielen auf einem reduzierten 9×9-Brett mit Meisterniveau, bringen es aber auf einem ausgewachsenen Brett nur bis zu einem gehobenen Amateurniveau.

Bridge ist ein Kartenspiel mit unvollständiger Information: Die Karten eines Spielers können von den anderen Spielern nicht eingesehen werden. Außerdem ist Bridge ein *Mehrspielerspiel* mit vier Spielern statt zwei, obwohl die Spieler in zwei Teams eingeteilt sind. Wie wir in *Abschnitt 5.6* gesehen haben, kann optimales Spiel in teilweise beobachtbaren Spielen wie Bridge Elemente der Informationsermittlung, der Kommunikation und einer sorgfältigen Abwägung von Wahrscheinlichkeiten beinhalten. Viele dieser Techniken werden im Programm Bridge Baron™ (Smith et al., 1998) verwendet, das 1997 die Computerbridge-Meisterschaft gewann. Bridge Baron spielt zwar nicht optimal, ist aber eines der wenigen erfolgreichen Spielsysteme, das komplexe hierarchische Pläne verwendet (siehe *Kapitel 11*). Dabei stützt es sich auf Konzepte wie etwa **Finessing** oder **Squeezing**, die allen Bridge-Spielern vertraut sind.

Das Programm GIB (Ginsberg, 1999) gewann im Jahr 2000 die Weltmeisterschaft ganz entscheidend mithilfe der Monte-Carlo-Methode. Seitdem sind andere Gewinnerprogramme der Führung von GIB gefolgt. Die wesentliche Innovation von GIB ist die Verwendung einer **erklärungsbasierten Verallgemeinerung**, um allgemeine Regeln für optimales Spiel in verschiedenen Klassen von Standardsituationen zu berechnen und zwischenzuspeichern, anstatt jede Situation individuell zu bewerten. Zum Beispiel gibt es in einer Situation, in der der eine Spieler die Karten A-K-Q-J-4-3-2 einer Farbe und ein anderer Spieler die Karten 10-9-8-7-6-5 besitzt, $7 \times 6 = 42$ Möglichkeiten, dass der erste Spieler eine Karte dieser Farbe ausspielen und der zweite Spieler bedienen kann. GIB behandelt diese Situationen aber als lediglich zwei: Der erste Spieler kann entweder eine hohe Karte oder eine niedrige Karte anspielen; die genauen Kartenwerte sind nicht relevant. Mit dieser Optimierung (neben einigen anderen) ist GIB in der Lage, eine vollständig beobachtbare Kartenverteilung mit 52 Karten in rund einer Sekunde *exakt* zu lösen. Die taktische Genauigkeit von GIB wiegt die Unfähigkeit des Programms auf, anhand von Informationen zu schließen. Es wurde in der Weltmeisterschaft von 1998 in einem Feld von 35 Teilnehmern im Paarturnier (wobei nur Hand gespielt und nicht gereizt wird) Zwölfter, was die Erwartungen vieler Experten bei weitem übertraf.

Es gibt mehrere Gründe, warum GIB auf der Expertenebene im Unterschied zu Kriegspiel-Programmen mit Monte-Carlo-Simulation spielt. Erstens ist die GIB-Bewertung der vollständig beobachtbaren Spielversion exakt, wobei der gesamte Spielbaum durchsucht wird, während sich Kriegspiel-Programme auf ungenaue Heuristiken stützen. Weit wichtiger ist aber die Tatsache, dass beim Bridge der größte Teil der Unbestimmtheit in den partiell beobachtbaren Informationen aus der Zufälligkeit der Kartenverteilung und nicht vom adversarialen Spiel des Gegners stammt. Monte-Carlo-Simulation kann gut mit Zufälligkeit umgehen, ist aber nicht immer für Strategie geeignet, speziell wenn die Strategie den Wert von Informationen einbezieht.

Scrabble: Die meisten Leute glauben, dass das Schwierige an Scrabble die Bildung guter Wörter ist, doch bei gegebenem offiziellen Wörterbuch zeigt sich, dass es ziemlich einfach ist, einen Zuggenerator zu programmieren, um den Zug mit der höchsten Punktzahl zu suchen (Gordon, 1994b). Das heißt jedoch nicht, dass das Spiel gelöst ist: Nimmt man lediglich den Zug mit der höchsten Punktzahl, resultiert jede Partie in einem guten Spiel, nicht jedoch in einem Expertenspiel. Das Problem besteht dabei darin, dass Scrabble sowohl teilweise beobachtbar als auch stochastisch ist: Sie wissen nicht, welche Buchstaben die anderen Spieler besitzen oder welche Buchstaben Sie als Nächstes ziehen. Scrabble vereint demnach die Schwierigkeiten von Backgammon und Bridge. Dennoch hat das Programm QUACKLE im Jahr 2006 den früheren Weltmeister David Boys 3 zu 2 geschlagen.

5.8 Alternative Ansätze

Weil die Berechnung optimaler Entscheidungen in Spielen größtenteils nicht handhabbar ist, müssen alle Algorithmen bestimmte Annahmen und Abschätzungen treffen. Der Standardansatz, der auf Minimax, Bewertungsfunktionen und Alpha-Beta basiert, ist nur eine Möglichkeit, dies zu realisieren. Da am Standardansatz bereits schon so lange gearbeitet wird, dominiert er möglicherweise alle anderen Methoden beim Turnierspiel. Es besteht die Ansicht, dass sich dadurch die Spieltheorie vom Mainstream der KI-Forschung abgeschieden hat: Der Standardansatz bietet nicht mehr genügend Raum für neue Einblicke in die allgemeinen Fragen der Entscheidungsfindung. In diesem Abschnitt geht es um die Alternativen.

Betrachten wir zuerst die Heuristik Minimax. Sie wählt einen optimalen Zug in einem vorgegebenen Suchbaum aus, sofern *die Blattknotenbewertungen völlig korrekt sind*. In der Praxis sind die Bewertungen üblicherweise grobe Schätzungen des Wertes einer Position und man kann davon ausgehen, dass sie mit großen Fehlern behaftet sind. ▶ Abbildung 5.14 zeigt einen Zwei-Schichten-Spielbaum, für den Minimax empfiehlt, der rechten Verzweigung zu folgen, da 100 > 99 ist. Das ist der korrekte Zug, wenn die Bewertungen alle korrekt sind. Selbstverständlich aber liefert die Bewertungsfunktion nur einen Näherungswert. Es sei angenommen, dass die Bewertung jedes Knotens einen Fehler aufweist, der unabhängig von anderen Knoten ist und einer zufälligen Verteilung mit dem Mittelwert 0 und der Standardabweichung σ entspricht. Bei $\sigma = 5$ ist dann die linke Verzweigung in 71% der Fälle besser und bei $\sigma = 2$ in 58% der Fälle. Dahinter steht die Intuition, dass die rechte Verzweigung vier Knoten besitzt, die nahe an 99 reichen. Wenn durch einen Fehler in der Bewertung eines der vier Knoten die rechte Verzweigung unter 99 rutscht, ist die linke Verzweigung besser.

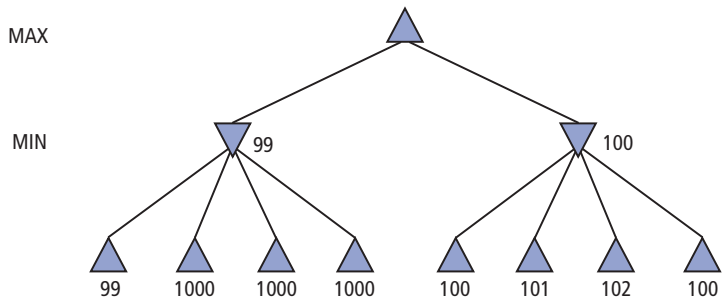


Abbildung 5.14: Ein Zwei-Schichten-Spielbaum, für den die Minimax-Heuristik einen Fehler liefern kann.

In der Praxis sehen die Dinge noch schlechter aus, da der Fehler in der Bewertungsfunktion nicht unabhängig ist. Wenn wir einen falschen Knoten erhalten, ist die Wahrscheinlichkeit hoch, dass in der Nähe liegende Knoten im Baum ebenfalls falsch sind. Aus dem Umstand, dass der Knoten mit der Bezeichnung 99 über gleichgeordnete (Geschwister-) Knoten mit der Bezeichnung 1000 verfügt, kann man schließen, dass er praktisch einen höheren wahren Wert haben könnte. Es ließe sich zwar eine Bewertungsfunktion, die eine Wahrscheinlichkeitsverteilung über mögliche Werte zurückgibt, verwenden, doch ist es schwierig, diese Verteilungen ordnungsgemäß zusammenzufassen, denn es fehlt ein gutes Modell der sehr starken Abhängigkeiten, die zwischen den Werten gleichgeordneter Knoten existieren.

Nun betrachten wir den Suchalgorithmus, der den Baum erzeugt. Das Ziel eines Algorithmenentwicklers ist es, eine Berechnung anzugeben, die schnell läuft und einen guten Zug ermittelt. Das offensichtlichste Problem des Alpha-Beta-Algorithmus ist, dass er darauf ausgelegt ist, nicht nur einen guten Zug auszuwählen, sondern auch die Grenzen für die Werte aller erlaubten Züge zu berechnen. Um zu sehen, warum diese zusätzliche Information unnötig ist, betrachten Sie eine Position, wo es nur einen erlaubten Zug gibt. Die Alpha-Beta-Suche erzeugt immer noch einen großen und völlig nutzlosen Baum und wertet diesen aus. Letztlich erfahren wir dadurch nur, dass der einzige Zug der beste Zug ist, und weisen ihm einen Wert zu. Doch da wir den Zug ohnehin ausführen müssen, ist es überflüssig, den Wert des Zuges zu kennen. Und gibt es nur einen offensichtlich guten Zug und mehrere Züge, die zwar zulässig sind, aber zu einem schnellen Verlust führen, möchten wir mit Alpha-Beta keine Zeit verschwenden, um einen genauen Wert für den einzigen guten Zug zu ermitteln. Es ist besser, den Zug schnell auszuführen und die Zeit für später zu sparen. Damit kommen wir zum Konzept des *Nutzens einer Knotenexpandierung*. Ein guter Suchalgorithmus sollte Knotenexpandierungen hohen Nutzens auswählen – d.h. solche, die wahrscheinlich zur Entdeckung eines sehr viel besseren Zuges führen. Wenn es keine Knotenexpandierungen gibt, deren Nutzen größer als ihre Kosten sind (gemessen nach der Zeit), sollte der Algorithmus die Suche abbrechen und einen Zug machen. Beachten Sie, dass das nicht nur in Situationen mit klarem Favoriten funktioniert, sondern auch für den Fall *symmetrischer* Züge, für die keine noch so umfangreiche Suche zeigen kann, dass ein Zug besser als ein anderer ist.

Diese Art des Schließens darüber, was Berechnungen leisten, wird auch als **Metaschließen** bezeichnet (Schließen über das Schließen). Es kann nicht nur auf das Spielen angewendet werden, sondern auf jede Art des Schließens. Alle Berechnungen werden zu dem Zweck angestellt, bessere Entscheidungen zu erzielen, alle haben Kosten und alle haben eine bestimmte Wahrscheinlichkeit, eine bestimmte Verbesserung der Entscheidungsqualität zu erbringen. Alpha-Beta verwendet die einfachste Art des Metaschließens, nämlich ein Theorem mit dem Effekt, dass bestimmte Zweige des Baumes ohne Verlust ignoriert werden können. Dafür sind wesentliche Verbesserungen möglich. In *Kapitel 16* werden wir sehen, wie diese Konzepte präzisiert und implementierbar gemacht werden können.

Schließlich wollen wir noch auf das Wesen der eigentlichen Suche eingehen. Algorithmen für die heuristische Suche und für das Spielen arbeiten, indem sie beginnend beim Ausgangszustand Folgen konkreter Zustände erzeugen und dann eine Bewertungsfunktion darauf anwenden. Offensichtlich unterscheidet sich das von der Vorgehensweise, wie Menschen Spiele spielen. Im Schach hat man häufig ein bestimmtes Ziel im Sinn – beispielsweise, die Dame des Gegners zu schlagen – und kann dieses Ziel nutzen, um *selektiv* plausible Pläne zu erzeugen, um es zu erreichen. Diese Art des zielgerichteten

Schließens oder Planens macht manchmal die kombinatorische Suche völlig überflüssig. David Wilkins' (1980) PARADISE ist das einzige Programm, das erfolgreich ein zielgerichtetes Schließen im Schach angewendet hat: Es war in der Lage, einige Schachprobleme zu lösen, für die Kombinationen aus 18 Zügen erforderlich waren. Momentan gibt es jedoch noch kein ausreichendes Wissen, wie man diese beiden Arten von Algorithmen zu einem robusten und effizienten System *kombinieren* könnte, obwohl Bridge Baron ein Schritt in die richtige Richtung sein könnte. Ein vollständig integriertes System wäre eine wesentliche Errungenschaft nicht nur für die Spieleforschung, sondern auch für die allgemeine KI-Forschung, weil es eine gute Grundlage für einen allgemeinen intelligenten Agenten darstellen könnte.

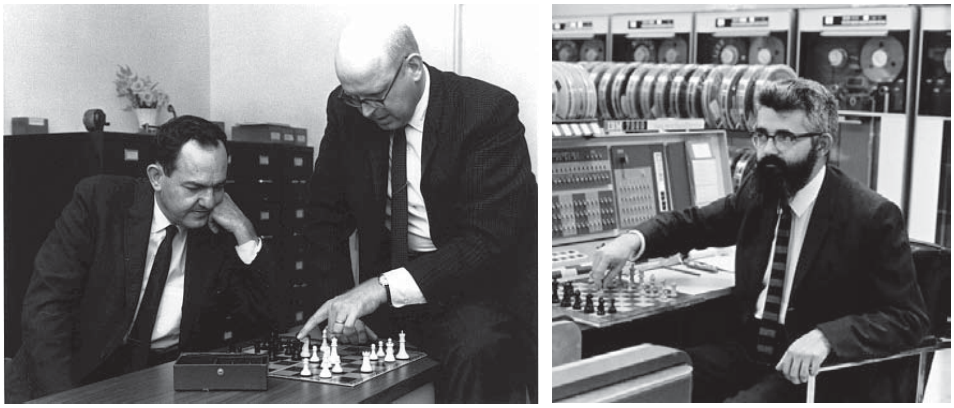
Bibliografische und historische Hinweise

Die frühe Geschichte mechanischen Spielens wurde von zahlreichen Schwindlern geprägt. Das dreisteste Beispiel unter ihnen war Baron Wolfgang von Kempelens (1734-1804) „Türke", ein angeblicher Schachspielautomat, der Napoleon besiegte, bevor schließlich herauskam, dass es sich dabei um einen Zauberkasten handelte, in dem ein menschlicher Schachspieler versteckt war (siehe Levitt, 2000). Er spielte von 1769 bis 1854. Im Jahr 1846 hat Charles Babbage (der von dem Türken fasziniert war) die erste ernsthafte Diskussion der Machbarkeit von Schach und Dame spielenden Computern angefacht (Morrison und Morrison, 1961). Babbage verstand allerdings nicht die exponentielle Komplexität von Suchbäumen und behauptete, dass „die in der Analytical Engine realisierten Kombinationen jede erforderliche Anzahl – selbst die für das Schachspiel – enorm überträfen." Er entwarf auch eine spezielle Maschine, die Tic Tac Toe spielen sollte (baute sie aber nicht). Die erste wirklich spielende Maschine wurde um 1890 von dem spanischen Ingenieur Leonardo Torres y Quevedo gebaut. Er spezialisierte sich auf das Schachendspiel mit König und Turm gegen König und garantierte einen Gewinn von König und Turm von jeder beliebigen Position aus.

Der Minimax-Algorithmus wird auf eine 1912 von Ernst Zermelo, dem Entwickler der modernen Mengentheorie, veröffentlichte Arbeit zurückgeführt. Leider enthielt diese Arbeit mehrere Fehler und hat Minimax nicht korrekt beschrieben. Andererseits wurden damit die grundlegenden Ideen der retrograden Analyse dargelegt und vorgeschlagen (jedoch nicht bewiesen), was als Zermelo-Theorem bekannt wurde: dass Schach bestimmt ist – entweder ist Weiß in der Lage, einen Gewinn zu erzwingen, oder Schwarz kann dies tun, oder aber es wird ein Remis; wir wissen nur nicht, was eintritt. Zermelo sagt, dass wir es schließlich wissen müssen: „Schach würde natürlich den Charakter eines Spieles überhaupt verlieren." Eine solide Grundlage für die Spieltheorie wurde in der Grundlagenarbeit *Theory of Games and Economic Behavior* (von Neumann und Morgenstern, 1944) entwickelt, die unter anderem in einer Analyse zeigte, dass für einige Spiele Strategien erforderlich sind, die dem *Zufall* unterliegen (oder anderweitig unvorhersagbar sind). Weitere Informationen finden Sie in *Kapitel 17*.

John McCarthy griff die Idee der Alpha-Beta-Suche 1956 auf, veröffentlichte sie aber nicht. Das Schachprogramm NSS (Newell et al., 1958) verwendete eine vereinfachte Alpha-Beta-Version; es war das erste Schachprogramm mit dieser Vorgehensweise. Hart und Edwards (1961) sowie Hart et al. (1972) beschrieben eine Alpha-Beta-Kürzung. Alpha-Beta wurde auch vom Schachprogramm „Kotok-McCarthy" verwendet, das von einem Studenten von John McCarthy entwickelt worden war (Kotok, 1962). Knuth und Moore (1975) bewiesen die Korrektheit von Alpha-Beta und analysierten dessen Zeit-

komplexität. Pearl (1982b) zeigt Alpha-Beta als asymptotisch optimal unter allen Spielbaum-Suchalgorithmen fester Tiefe.



a b

Abbildung 5.15: Pioniere im Computerschach: (a) Herbert Simon und Allen Newell, Entwickler des Programms NSS (1958); (b) John McCarthy und das Programm KOTOK-MCCARTHY auf einem IBM 7090 (1967).

Es wurden mehrere Versuche unternommen, die in *Abschnitt 5.8* beschriebenen Probleme mit dem „Standardansatz“ zu lösen. Der erste noch unvollständige Suchalgorithmus mit einigen theoretischen Grundlagen war vielleicht B* (Berliner, 1979), der versucht, Intervallgrenzen für den möglichen Wert eines Knotens im Spielbaum zu verwalten, anstatt eine Schätzung eines einzigen Wertes abzugeben. Blattknoten werden zur Expandierung ausgewählt, um die obersten Grenzen zu verfeinern, bis ein Zug der „offensichtlich beste Zug“ ist. Palay (1985) erweitert das B*-Konzept durch Verwendung von Wahrscheinlichkeitsverteilungen für Werte anstelle von Intervallen. David McAllesters (1988) konspirative Zahlensuche expandiert Blattknoten, die durch Änderung ihrer Werte das Programm veranlassen könnten, an der Wurzel einem neuen Zug den Vorrang zu geben. MGSS* (Russell und Wefald, 1989) verwendet die Techniken der Entscheidungstheorie aus *Kapitel 16*, um den Wert der Erweiterung jedes Blattes in Bezug auf die erwartete Verbesserung in der Entscheidungsqualität an der Wurzel abzuschätzen. Er besiegte einen Alpha-Beta-Algorithmus für Othello, obwohl die Suche nur mit einer um eine Größenordnung geringeren Anzahl von Knoten stattfand. Der MGSS*-Ansatz ist im Prinzip für die Steuerung jeder Form von Überlegung anwendbar.

Die Alpha-Beta-Suche ist in vielerlei Hinsicht das Zwei-Spieler-Analogon zur Tiefensuche mit Verzweigen und Begrenzen (Branch-and-Bound), die im Fall des Einzelagenten von A* dominiert wird. Der SSS*-Algorithmus (Stockman, 1979) kann als Zweispieler-A* betrachtet werden und erweitert niemals mehr Knoten als Alpha-Beta, um zur gleichen Entscheidung zu gelangen. Die Speicheranforderungen und der zusätzliche Verwaltungsaufwand für die Warteschlange machen SSS* in seiner ursprünglichen Form unpraktisch, aber aus dem RBFS-Algorithmus (Korf und Chickering, 1996) wurde eine Version mit linearem Speicheraufwand entwickelt. Plaat et al. (1996) haben eine neue Sicht von SSS* als Kombination von Alpha-Beta und Transpositionstabellen entwickelt, die zeigte, wie die Nachteile des ursprünglichen Algorithmus überwunden werden können. Sie entwickelten eine neue Variante, MTD(*f*), die von zahlreichen anderen wichtigen Programmen übernommen wurde.

D.F. Beal (1980) und Dana Nau (1980, 1983) untersuchten die Schwächen von Minimax für Schätzungsbewertungen. Sie zeigten, dass Minimax unter bestimmten Annahmen zur Verteilung der Blattwerte im Baum Werte an der Wurzel ergeben kann, die tatsächlich weniger *zuverlässig* sind als die direkte Verwendung der eigentlichen Bewertungsfunktion. Das Buch *Heuristics* (1984) von Pearl erklärt zum Teil dieses scheinbare Paradoxon und analysiert viele Spielealgorithmen. Baum und Smith (1997) schlagen einen auf Wahrscheinlichkeiten basierenden Ersatz für Minimax vor und zeigen dabei, dass er für viele Spiele bessere Auswahlen erzeugt. Der Expectiminimax-Algorithmus wurde von Donald Michie (1996) vorgeschlagen. Bruce Ballard (1983) erweiterte die Alpha-Beta-Kürzung auf Bäume mit Zufallsknoten und Hauk (2004) überprüft diese Arbeit und gibt empirische Ergebnisse an.

Koller und Pfeffer (1997) beschreiben ein System, das partiell beobachtbare Spiele vollständig lösen kann. Das System ist ziemlich allgemein konzipiert und eignet sich für Spiele, deren optimale Strategie Züge mit Zufallskomponente voraussetzen, und Spiele, die komplexer sind als solche, mit denen bisherige Systeme umgehen konnten. Dennoch ist es nicht in der Lage, komplexe Spiele wie Poker, Bridge und Kriegspiel zu behandeln. Frank et al. (1998) beschreiben mehrere Varianten der Monte-Carlo-Suche, unter anderem auch eine, bei der MIN vollständige Informationen besitzt, MAX dagegen nicht. Unter den deterministischen, partiell beobachtbaren Spielen hat Kriegspiel die meiste Aufmerksamkeit erfahren. Ferguson demonstrierte von Hand abgeleitete Strategien mit Zufallskomponente, die den Gewinn von Kriegspiel mit einem Läufer und einem Springer (1992) oder zwei Läufern (1995) gegen einen König ermöglichten. Die ersten Kriegspiel-Programme haben sich darauf konzentriert, Endspiele im Schach zu suchen, und AND-OR-Suche im Belief-States-Raum ausgeführt (Sakuta und Iida, 2002; Bolognesi und Ciancarini, 2003). Mithilfe inkrementeller Belief-States-Algorithmen ließen sich im Schach wesentlich komplexere Mittelspiele finden (Russell und Wolfe, 2005; Wolfe und Russell, 2007), doch bleibt eine effiziente Zustandseinschätzung das Haupthindernis für effektives Allgemeinspiel (Parker et al., 2005).

Schach gehörte zu den ersten Aufgaben, denen sich KI annahm, wobei frühe Anstrengungen durch viele Pioniere der Computertechnik zu verzeichnen sind. Hier sind Konrad Zuse (1945), Norbert Wiener mit seinem Buch *Kybernetik* (1948) und Alan Turing 1950 (siehe Turing et al., 1953) zu nennen. Doch es war der Artikel *Programming a Computer for Playing Chess* (1950) von Claude Shannon, der mit dem vollständigsten Satz von Ideen aufwarten konnte und eine Darstellung für Brettpositionen, eine Bewertungsfunktion, das Prinzip der Ruhesuche und einige Konzepte für die selektive (nicht erschöpfende) Spielbaumsuche beschrieb. Slater (1950) und die Kommentatoren seines Artikels untersuchten ebenfalls die Möglichkeiten, einen Computer Schach spielen zu lassen.

D.G. Prinz (1952) hatte ein Programm fertiggestellt, das die Endspielprobleme im Schach löste, aber kein ganzes Spiel absolvieren konnte. Stan Ulam und eine Gruppe am Los Alamos National Lab schufen ein Programm, das Schach auf einem 6×6-Brett ohne Läufer spielte (Kister et al., 1957). Es konnte eine Suche mit einer Tiefe von 4 Schichten in 12 Minuten ausführen. Alex Bernstein schrieb das erste dokumentierte Programm, das ein vollständiges Spiel Standardschach spielen konnte (Bernstein und Roberts, 1958).[5]

5 Das russische Programm BESM wurde möglicherweise vor Bernsteins Programm geschrieben.

Das erste Schachspiel zwischen Computern fand zwischen dem Programm Kotok-McCarthy vom MIT (Kotok, 1962) und dem Programm ITEP, das Mitte der 1960er Jahre in Moskau am Institut für theoretische und experimentelle Physik (ITEP) geschrieben wurde (Adelson-Velsky et al., 1970), statt. Dieses Interkontinental-Match wurde per Telegrafie gespielt. Es endete 1967 mit einem 3:1-Sieg für das Programm ITEP. Das erste Schachprogramm, das erfolgreich gegen Menschen spielte, war MACHACK-6 vom MIT (Greenblatt et al., 1967). Seine Elo-Bewertung von etwa 1400 lag weit über dem Anfängerniveau 1000.

Der 1980 eingerichtete Fredkin Prize bot Preise für fortschrittliche Meilensteine im Schachspielen. Der mit $5.000 dotierte Preis für das erste Programm, das eine Meisterbewertung erreicht, ging an BELLE mit einer Bewertung von 2250 Punkten (Condon und Thompson, 1982). Der $10.000-Preis für das erste Programm mit einer USCF-Bewertung (United States Chess Federation) von 2.500 Punkten (nahe dem Großmeisterniveau) ging 1989 an DEEP THOUGHT (Hsu et al., 1990). Der Großpreis von $100.000 wurde DEEP BLUE (Campbell et al., 2002; Hsu, 2004) für seinen Meilenstein-Sieg über den Weltmeister Garry Kasparov in einem Schaukampf 1997 verliehen. Kasparov schrieb:

> *Entscheidend in dem Kampf war Spiel 2, das eine Kerbe in meinem Gedächtnis hinterlassen hat ... wir haben etwas gesehen, das unsere wildesten Erwartungen weit übertraf, wie gut ein Computer langfristige Konsequenzen seiner Positionsentscheidungen vorhersehen kann. Die Maschine weigerte sich, an eine Position zu ziehen, die einen Kurzzeitvorteil erbringt – und zeigte ein sehr menschliches Gespür für Gefahr. (Kasparov, 1997)*

Die vielleicht vollständigste Beschreibung eines modernen Schachprogramms finden Sie bei Ernst Heinz (2000), dessen Programm DARKTHOUGHT das am höchsten bewertete nicht kommerzielle PC-Programm der Weltmeisterschaften von 1999 war.

In den letzten Jahren sind Schachprogramme selbst an den besten Menschen der Welt vorbeigezogen. HYDRA schlug 2004–2005 den Großmeister Evgeny Vladimirov mit 3,5–0,5, den Weltmeister Ruslan Ponomariov 2–0 und den auf Platz 7 der Weltrangliste stehenden Michael Adams mit 5,5–0,5. Das Programm DEEP FRITZ schlug 2006 den Weltmeister Vladimir Kramnik 4–2 und 2007 hat RYBKA mehrere Großmeister besiegt, wobei das Programm den menschlichen Gegnern Vorteile (beispielsweise einen Bauern) einräumte. Mit dem Stand von 2009 ist die höchste je verzeichnete Elo-Bewertung die von Kasparov mit 2851. HYDRA (Donninger und Lorenz, 2004) wird mit 2850 bis 3000 bewertet, was vorwiegend auf seinen überraschend hohen Wettkampfsieg über Michael Adams beruht. Das Programm RYBKA wird zwischen 2900 und 3100 eingestuft, wobei diese Angaben auf einer kleineren Anzahl von Spielen basieren und nicht als zuverlässig gelten. Ross (2004) zeigt, wie menschliche Spieler gelernt haben, sich auf einige der Schwächen von Computerprogrammen einzustellen.

Dame war das erste klassische Spiel, das vollständig von einem Computer gespielt wurde. Christopher Strachey (1952) schrieb das erste funktionsfähige Programm für das Damespiel. Ab 1952 entwickelte Arthur Samuel von IBM in seiner Freizeit ein Dameprogramm, das seine eigene Bewertungsfunktion lernte, indem es Tausende Male gegen sich selbst spielte (Samuel, 1959, 1967). Wir beschreiben dieses Konzept in *Kapitel 21* im Detail. Das Programm von Samuel begann als Anfänger, war aber nach nur ein paar Tagen Spiel mit sich selbst besser als Samuel selbst geworden. 1962 schlug es Robert Nealy, einen Meister im „Blind-Dame“, durch einen Fehler, den dieser gemacht hatte. Berücksichtigt man Samuels Rechnerausrüstung (ein IBM 704) mit 10.000 Worten

Arbeitsspeicher, einem Magnetband als Langzeitspeicher und einem 0,000001 GHz-Prozessor, bleibt dieser Sieg eine wirklich großartige Leistung.

Die Herausforderung, der sich Samuel gestellt hatte, wurde von Jonathan Schaeffer von der Universität Alberta aufgegriffen. Sein Programm CHINOOK wurde Zweiter bei den U.S. Open von 1990 und erlangte damit das Recht, den Weltmeister herauszufordern. Anschließend stand er vor einem Problem – verkörpert durch Marion Tinsley. Dr. Tinsley hatte die Weltmeisterschaft 40 Jahre lang gewonnen und dabei insgesamt nur drei Spiele verloren. Im ersten Spiel gegen CHINOOK erlitt Tinsley seinen vierten und fünften Verlust, gewann den Kampf aber mit 20,5–18,5. Die Weltmeisterschaft im August 1994 zwischen Tinsley und Chinook endete vorzeitig, weil Tinsley sich aus gesundheitlichen Gründen zurückziehen musste. Chinook wurde der offizielle Weltmeister. Schaeffer baute seine Datenbank der Endspiele weiter aus und „löste" 2007 das Damespiel (Schaeffer et al., 2007; Schaeffer, 2008). Dies wurde von Richard Bellman (1965) vorausgesagt. Im Artikel, der das Konzept der dynamischen Programmierung für retrograde Analyse einführte, schrieb er: „Beim Damespiel ist die Anzahl der möglichen Züge in einer bestimmten Situation so klein, dass wir eine vollständige Lösung für das Problem des optimalen Spielens in diesem Spiel durch einen Digitalrechner mit Sicherheit erwarten dürfen." Allerdings hatte Bellman die Größe des Spielbaumes für das Damespiel nicht ganz erfasst. Es gibt rund 500 Billiarden Positionen. Nach 18 Jahren Berechnung auf einem Cluster aus 50 oder mehr Computern hat das Team um Jonathan Schaeffer eine Endspieltabelle für alle Damespielpositionen mit 10 oder weniger Steinen fertiggestellt: über 39 Billionen Einträge. Von da aus waren sie in der Lage, mit einer Vorwärts-Alpha-Beta-Suche eine Strategie abzuleiten, die beweist, dass Dame bei perfektem Spiel durch beide Seiten im Unentschieden endet. Beachten Sie, dass dies eine Anwendung der bidirektionalen Suche ist (*Abschnitt 3.4.6*). Das Erstellen einer Endspieltabelle für das gesamte Damespiel wäre unpraktisch: Der Speicherbedarf für die Tabelle würde etwa eine Milliarde Gigabyte betragen. Ebenso unpraktisch wäre eine Suche ohne jede Tabelle: Der Suchbaum weist rund 8^{47} Positionen auf und die Suche würde mit der heutigen Technologie Tausende Jahre dauern. Nur mit einer Kombination aus intelligenter Suche, Endspieldaten und einem Preisverfall von Prozessoren und Speicher lässt sich das Damespiel lösen. Somit reiht sich Dame in Qubic (Patashnik, 1980), Connect Four (Allis, 1988) und Nine-Men's Morris (Gasser, 1998) als eines der Spiele ein, die durch Computeranalyse gelöst wurden.

Backgammon ist ein Strategiespiel mit Zufallskomponente und wurde mathematisch von Gerolamo Cardano (1663) analysiert, aber erst Ende der 1970er Jahre als Computerspiel aufgegriffen. Das erste Backgammon-Programm war BKG (Berliner, 1980b). Es verwendete eine komplexe, manuell konstruierte Bewertungsfunktion und suchte nur bis zur Tiefe 1. Es war das erste Programm, das einen menschlichen Weltmeister in einem großen klassischen Spiel besiegte (Berliner, 1980a). Berliner räumte bereitwillig ein, dass BKG Glück beim Würfeln hatte. Dagegen spielte das Programm TD-GAMMON von Gerry Tesauro (1995) beständig auf Weltmeisterebene. Das Programm BGBLITZ war der Gewinner der Computerolympiade 2008.

Go ist ein deterministisches Spiel, stellt aber durch den großen Verzweigungsfaktor eine echte Herausforderung dar. Die wichtigsten Fragen und frühe Literatur im Computer-Go wurden von Bouzy und Cazenave (2001) sowie Müller (2002) zusammengefasst. Bis 1997 gab es keine leistungsfähigen Go-Programme. Heute spielen die besten Programme die *meisten* ihrer Züge auf der Meisterebene; das einzige Problem besteht darin, dass sie im Verlauf des Spieles mindestens einen schweren Schnitzer machen,

der einem starken Gegner das Gewinnen ermöglicht. Während die Alpha-Beta-Suche in den meisten Spielen vorherrschend ist, haben viele jüngere Go-Programme Monte-Carlo-Methoden basierend auf dem UCT (Upper Confidence bounds on Trees, Obere Konfidenzgrenzen bei Bäumen)-Schema übernommen (Kocsis and Szepesvari, 2006). Das bis 2009 stärkste Go-Programm ist MoGo von Gelly und Silver (Wang und Gelly, 2007; Gelly und Silver, 2008). Im August 2008 hatte MoGo einen überraschenden Gewinn gegen den Top-Berufsspieler Myungwan Kim zu verzeichnen, obwohl MoGo mit einem Handicap von neun Steinen spielen musste (was etwa dem Handicap einer Dame im Schach entspricht). Kim schätzte die Stärke von MoGo mit 2 bis 3 Dan, dem unteren Ende fortgeschrittener Amateurspieler. Für dieses Match wurde MoGo auf einem Supercomputer mit 800 Prozessoren und einer Leistung von 15 Teraflops ausgeführt (dem 1000-Fachen von Deep Blue). Einige Wochen später gewann MoGo mit einem Handicap von nur fünf Steinen gegen einen Meisterspieler mit dem 6. Dan. In der 3 × 3-Form von Go erreicht MoGo ungefähr den 1. Dan eines Meisterspielers. Schnelle Fortschritte sind wahrscheinlich, da Experimente mit neuen Formen der Monte-Carlo-Suche fortgeführt werden. Der von der Computer Go Association veröffentlichte *Computer Go Newsletter* beschreibt aktuelle Entwicklungen.

Bridge: Smith et al. (1998) berichten, wie ihr planungsbasiertes Programm die Computer-Bridge-Meisterschaft 1998 gewann, und Ginsberg (2001) beschreibt, wie sein GIB-Programm, das auf Monte-Carlo-Simulation basiert, die folgende Computermeisterschaft gewann und überraschend gut gegen menschliche Spieler sowie Problemmengen aus Standardwerken abschnitt. Von 2001 bis 2007 wurde die Computer-Bridge-Meisterschaft fünfmal durch Jack und zweimal durch Wbridge5 gewonnen. Zu keinem Programm gibt es akademische Aufsätze, die ihre Struktur beschreiben, doch nutzen sie angeblich die Monte-Carlo-Technik, die erstmals durch Levy (1989) für Bridge vorgeschlagen wurde.

Scrabble: Eine gute Beschreibung des Spitzenprogramms Maven gibt sein Autor Brian Sheppard (2002). Das Generieren des Zuges mit dem höchsten Punktewert wird von Gordon (1994) beschrieben. Richards und Amir (2007) behandeln das Modellieren von Gegnern.

Soccer (Kitano et al., 1997b; Visser et al., 2008) und **Billiards** (Lam und Greenspan, 2008; Archibald et al., 2009) sowie andere stochastische Spiele mit einem kontinuierlichen Aktionsraum ziehen vermehrt die Aufmerksamkeit in der KI auf sich, und zwar sowohl in der Simulation als auch mit physischen Roboterspielern.

Computerspiel-Meisterschaften finden jährlich statt und Artikel erscheinen an den verschiedensten Stellen. Der eher irreführend benannte Tagungsband *Heuristic Programming in Artificial Intelligence* berichtet von den Computerolympiaden, die eine Vielzahl von Spielen umfassen. Die General Game Competition (Love et al., 2006) testet Programme, die allein anhand einer logischen Beschreibung der Spielregeln lernen müssen, ein unbekanntes Spiel zu spielen. Darüber hinaus gibt es mehrere bearbeitete Sammlungen zu wichtigen Arbeiten der Computerspiel-Forschung (Levy, 1988a, 1988b; Marsland und Schaeffer, 1990). Die 1997 gegründete ICCA (International Computer Chess Association) veröffentlicht vierteljährlich das *ICGA Journal* (zuvor *ICCA Journal*). Wichtige Arbeiten wurden in der Serienanthologie *Advances in Computer Chess* veröffentlicht, beginnend mit Clarke (1977). Band 134 des Journals *Artificial Intelligence* (2002) enthält Beschreibungen aktueller Programme für Schach, Othello, Hex, Shogi, Go, Backgammon, Poker, Scrabble und anderer Spiele. Seit 1998 wird eine zweijährliche Konferenz *Computers and Games* abgehalten.

Zusammenfassung

In diesem Kapitel haben wir mehrere Spiele betrachtet, um zu verstehen, was unter „optimalem Spielen" zu verstehen ist und wie man in der Praxis gut spielt. Die wichtigsten Konzepte sind:

- Ein Spiel kann definiert werden durch den **Ausgangszustand** (wie das Brett aufgestellt ist), die erlaubten **Aktionen** in jedem Zustand, das **Ergebnis** jeder Aktion sowie einen **Endetest** (der besagt, wann das Spiel vorbei ist) und eine **Nutzenfunktion**, die auf die Endzustände anzuwenden ist.
- In **Zwei-Spieler-Nullsummenspielen** mit **perfekter Information** kann der **Minimax**-Algorithmus unter Verwendung einer Tiefensuchauflistung des Spielbaumes optimale Züge auswählen.
- Der **Alpha-Beta**-Suchalgorithmus berechnet denselben optimalen Zug wie Minimax, erzielt aber eine größere Effizienz durch Eliminieren von Unterbäumen, die wahrscheinlich irrelevant sind.
- Normalerweise ist es nicht machbar, den gesamten Suchbaum zu betrachten (nicht einmal mit Alpha-Beta); deshalb müssen wir die Suche an irgendeiner Stelle abbrechen und eine **Bewertungsfunktion** anwenden, die eine Abschätzung für den Nutzen eines Zustandes ergibt.
- Viele Spieleprogramme berechnen im Voraus Tabellen für die besten Züge im Eröffnungs- und Endspiel, sodass sie einen Zug nachschlagen können, anstatt ihn suchen zu müssen.
- Spiele mit Zufallskomponente können mithilfe einer Erweiterung des Minimax-Algorithmus bearbeitet werden, der einen **Zufallsknoten** auswertet, indem er den durchschnittlichen Nutzen aller seiner untergeordneten Knoten ermittelt, gewichtet nach der Wahrscheinlichkeit jedes dieser untergeordneten Knoten.
- Ein optimales Spielen in Spielen mit **unvollständiger Information** wie beispielsweise Kriegspiel und Bridge bedingt ein Schließen zu den aktuellen und zukünftigen **Belief States** jedes Spielers. Eine einfache Annäherung kann erzielt werden, indem der Wert einer Aktion über jede mögliche Konfiguration der fehlenden Informationen gemittelt wird.
- Programme haben sogar menschliche Meister in Spielen wie Schach, Dame und Othello besiegt. In verschiedenen Spielen mit unvollständiger Information wie zum Beispiel Poker, Bridge und Kriegspiel sowie Spielen mit sehr großen Verzweigungsfaktoren und wenig gutem Heuristikwissen wie etwa Go ist der Mensch allerdings noch überlegen.

Übungen zu Kapitel 5

Lösungshinweise

1 Nehmen Sie ein Orakel *OM(s)* an, das den gegnerischen Zug in jedem beliebigen Zustand korrekt vorhersagen kann. Formulieren Sie damit die Definition eines Spieles als (Einzelagent-) Suchproblem. Beschreiben Sie einen Algorithmus, um den optimalen Zug zu suchen.

2 Betrachten Sie das Problem, zwei 8-Puzzles zu lösen.

a. Geben Sie eine vollständige Problemformulierung in der Art von *Kapitel 3* an.

b. Wie groß ist der erreichbare Zustandsraum? Geben Sie einen genauen numerischen Ausdruck an.

c. Nehmen Sie an, dass das Problem wie folgt adversarial gemacht wird: Die beiden Spieler führen abwechselnd Züge aus. Mit einem Münzwurf wird das Puzzle ermittelt, für das ein Zug in dieser Runde zu erfolgen hat. Der Gewinner ist derjenige, der zuerst ein Puzzle gelöst hat. Mit welchem Algorithmus lässt sich bei dieser Vorgabe ein Zug auswählen?

d. Geben Sie einen formlosen Beweis an, dass schließlich jemand gewinnt, wenn beide Spieler perfekt spielen.

3 Nehmen Sie an, dass in Übung 3.4 einer der Freunde dem anderen ausweichen möchte. Das Problem wird dann zu einem Zweispieler-**Verfolgungs-Ausweichen**-Spiel. Wir nehmen nun an, dass die Spieler abwechselnd ziehen. Das Spiel endet erst, wenn die Spieler auf demselben Knoten ankommen; von der Endauszahlung für den Verfolger wird die benötigte Gesamtzeit abgezogen. (Der Flüchtende „gewinnt", indem er niemals verliert.) ▶ Abbildung 5.16 zeigt ein Beispiel.

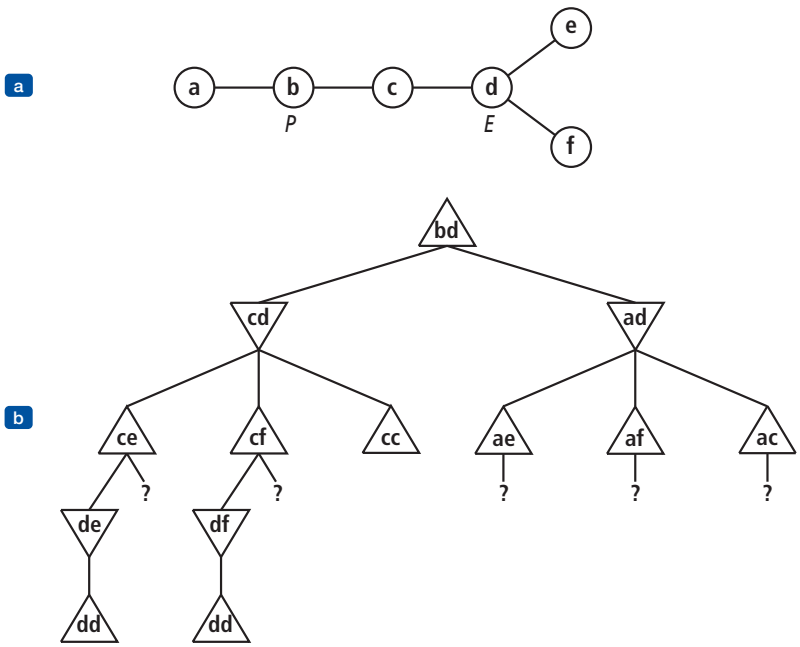


Abbildung 5.16: (a) Eine Karte, bei der die Kosten jeder Kante gleich 1 sind. Anfangs befindet sich der Flüchtende *P* am Knoten **b** und der Verfolger *E* am Knoten **d**. (b) Ein Teilspielbaum für diese Karte. Jeder Knoten ist mit den Positionen für P und E bezeichnet. P-Züge sind zuerst angegeben. Mit einem Fragezeichen markierte Verzweigungen müssen noch untersucht werden.

a. Kopieren Sie den Spielbaum und markieren Sie die Werte der Terminalknoten.

b. Schreiben Sie neben jeden internen Knoten die stärkste Tatsache, die Sie über ihren Wert ableiten können (eine Zahl, eine oder mehrere Ungleichungen wie zum Beispiel „≥ 14" oder ein „?").

c. Schreiben Sie unter jedes Fragezeichen den Namen des Knotens, der durch diese Verzweigung erreicht wird.

d. Erläutern Sie, wie eine Begrenzung des Knotenwertes in (c) von der Betrachtung der kürzesten Pfadlängen auf der Karte abgeleitet werden kann, und leiten Sie derartige Grenzen für die betreffenden Knoten ab. Berücksichtigen Sie dabei die Kosten, um zu jedem Blattknoten zu gelangen, sowie die Kosten, um ihn aufzulösen.

e. Nehmen Sie nun an, dass der gegebene Baum mit den Blattgrenzen von (d) von links nach rechts ausgewertet wird. Kreisen Sie diejenigen Fragezeichenknoten ein, die weiter expandiert werden müssten, unter Berücksichtigung der Grenzen von Teil (d), und streichen Sie diejenigen durch, die überhaupt nicht betrachtet werden müssen.

f. Können Sie irgendetwas allgemein beweisen über denjenigen, der das Spiel gewinnt, auf einer Karte, die ein Baum ist?

4 Beschreiben oder implementieren Sie Zustandsbeschreibungen, Zuggeneratoren, Endetests, Nutzenfunktionen und Bewertungsfunktionen für eines oder mehrere der folgenden Spiele: Monopoly, Scrabble, Bridge (mit einem bestimmten Kontrakt) und Poker (Texas Hold'em).

5 Beschreiben und implementieren Sie eine *Echtzeit-Mehrspieler*-Spielumgebung, wobei die Zeit Teil des Umgebungszustandes ist und die Spieler feststehende Zeitabschnitte erhalten.

6 Erörtern Sie, wie gut der Standardansatz zum Spielen auf Spiele wie etwa Tennis, Pool oder Kricket angewendet werden kann, die in einem stetigen physischen Zustandsraum stattfinden.

7 Beweisen Sie die folgende Behauptung: Für jeden Spielbaum ist der von MAX unter Verwendung von Minimax-Entscheidungen gegen ein suboptimales MIN erhaltene Nutzen nie kleiner als der Nutzen, der erzielt wird, wenn gegen ein optimales MIN gespielt wird. Können Sie einen Spielbaum entwerfen, wobei MAX unter Verwendung einer *suboptimalen* Strategie gegen ein suboptimales MIN noch besser ist?

8 Betrachten Sie das in ▶ Abbildung 5.17 gezeigte Zweispieler-Spiel:

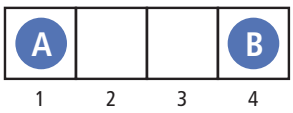


Abbildung 5.17: Die Ausgangsposition eines einfachen Spieles. Spieler *A* zieht als Erster. Die beiden Spieler ziehen abwechselnd und jeder Spieler muss seinen Spielstein in ein freies benachbartes Feld in beliebiger Richtung schieben. Wenn der Gegner ein benachbartes Feld belegt, kann ein Spieler über den Gegner auf ein freies Quadrat springen, falls ein solches vorhanden ist. (Ist beispielsweise *A* auf 3 und *B* auf 2, dann kann *A* zurück auf 1 springen.) Das Spiel endet, wenn ein Spieler das entgegengesetzte Brettende erreicht hat. Erreicht Spieler *A* zuerst das Feld 4, ist der Spielwert für *A* gleich +1; erreicht der Spieler *B* das Feld 1 zuerst, ist der Spielwert für *A* gleich -1.

a. Zeichnen Sie den gesamten Spielbaum unter Verwendung der folgenden Konventionen:
   - Schreiben Sie jeden Status als (s_A, s_B), wobei s_A und s_B die Token-Positionen bezeichnen.
   - Stellen Sie jeden Endzustand in einem Quadrat dar und schreiben Sie seinen Spielwert in einen Kreis.
   - Schreiben Sie Schleifenzustände (Zustände, die bereits auf dem Pfad zur Wurzel erschienen sind) in doppelte Quadrate. Weil nicht klar ist, wie Schleifenzuständen Werte zugewiesen werden, kennzeichnen Sie sie mit einem Fragezeichen (?) in einem Kreis.

b. Markieren Sie jeden Knoten mit seinem aktualisierten Minimax-Wert (auch in einem Kreis). Erklären Sie, wie Sie die „?"-Werte behandelt haben und warum Sie das so gemacht haben.

c. Erklären Sie, warum der Minimax-Standardalgorithmus für diesen Spielbaum fehlschlagen würde, und skizzieren Sie kurz, wie Sie ihn reparieren könnten. Ziehen Sie dazu Ihre Antwort auf (b) heran. Bietet Ihr veränderter Algorithmus optimale Entscheidungen für alle Spiele mit Schleifen?

d. Dieses Spiel auf vier Quadraten kann auf n Quadrate für jedes $n > 2$ verallgemeinert werden. Beweisen Sie, dass A gewinnt, wenn n gerade ist, und dass es verliert, wenn n ungerade ist.

9 Dieses Problem verdeutlicht die grundlegenden Konzepte beim Spielen anhand von Tic Tac Toe. Wir definieren X_n als die Anzahl der Zeilen, Spalten oder Diagonalen mit genau n X und keinem O. Analog dazu ist O_n die Anzahl der Zeilen, Spalten oder Diagonalen mit genau n O. Die Nutzenfunktion ordnet jeder Position mit $X_3 = 1$ den Wert $+1$ zu, jeder Position mit $O_3 = 1$ den Wert -1. Alle anderen Endpositionen haben den Nutzen 0. Für andere als Endpositionen verwenden wir eine lineare Bewertungsfunktion, die definiert ist als $Eval(s) = 3X_2(s) + X_1(s) - (3O_2(s) + O_1(s))$.

a. Wie viele mögliche Tic Tac Toe-Spiele gibt es schätzungsweise?

b. Zeigen Sie den gesamten Spielbaum beginnend bei einem leeren Brett bis zur Tiefe 2 (d.h. mit einem X und einem O auf dem Brett) und berücksichtigen Sie dabei die Symmetrie.

c. Markieren Sie in Ihrem Baum die Bewertungen aller Positionen an der Tiefe 2.

d. Markieren Sie unter Verwendung des Minimax-Algorithmus in Ihrem Baum die aktualisierten Werte für die Positionen an den Tiefen 1 und 0 und verwenden Sie diese Werte, um den besten Anfangszug zu wählen.

e. Kreisen Sie die Knoten der Tiefe 2 ein, die *nicht* bewertet werden, wenn eine Alpha-Beta-Kürzung angewendet wird, vorausgesetzt, die Knoten werden in der optimalen Reihenfolge für die Alpha-Beta-Kürzung erzeugt.

10 Betrachten Sie die Familie von verallgemeinerten Tic Tac Toe-Spielen, die wie folgt definiert ist: Jedes konkrete Spiel wird durch eine Menge S von *Quadraten* und eine Auflistung W von *Gewinnpositionen* spezifiziert. Jede Gewinnposition ist eine Teilmenge von S. Zum Beispiel ist im Standard-Tic Tac Toe S eine Menge von 9 Quadraten und W eine Auflistung von 8 Teilmengen von S – die drei Zeilen, die drei Spalten und die beiden Diagonalen. Im Übrigen ist das Spiel identisch mit dem standardmäßigen Tic Tac Toe. Beginnend mit einem leeren Spielfeld setzen die Spieler abwechselnd ihre Zeichen auf ein leeres Quadrat. Ein Spieler, der jedes Quadrat in einer Gewinnposition markiert hat, gewinnt das Spiel. Es kommt zu einem Unentschieden, wenn alle Felder markiert sind und keiner der Spieler gewonnen hat.

a. Es sei $N = |S|$, die Anzahl der Quadrate. Geben Sie eine obere Grenze für die Anzahl der Knoten im vollständigen Spielbaum für das verallgemeinerte Tic Tac Toe als Funktion von N an.

b. Geben Sie eine untere Grenze für die Größe des Spielbaumes für den ungünstigsten Fall an, bei dem $W = \{\}$.

c. Schlagen Sie eine plausible Bewertungsfunktion vor, die sich für jede Instanz des verallgemeinerten Tic Tac Toe verwenden lässt. Die Funktion kann von S und W abhängen.

d. Nehmen Sie an, dass es möglich ist, in $100N$ Maschinenanweisungen ein neues Spielfeld zu erstellen und zu überprüfen, ob es sich um eine Gewinnposition handelt. Der Prozessor werde mit 2 GHz getaktet. Lassen Sie Speicherbeschränkungen außer Acht. Wie groß darf entsprechend Ihrer Abschätzung von (a) ein Spielbaum sein, um ihn vollständig durch Alpha-Beta in einer Sekunde/einer Minute/einer Stunde CPU-Zeit zu lösen?

11 Entwickeln Sie ein allgemeines Spieleprogramm, das verschiedene Spiele beherrscht.

a. Implementieren Sie Zuggeneratoren und Bewertungsfunktionen für eines oder mehrere der folgenden Spiele: Kalah, Othello, Dame und Schach.

b. Konstruieren Sie einen allgemeinen Alpha-Beta-Spielagenten, der Ihre Implementierung benutzt.

c. Vergleichen Sie die Auswirkungen, wenn Sie die Suchtiefe erhöhen oder die Zugreihenfolge oder Bewertungsfunktion verbessern. Wie nahe kommt Ihr effektiver Verzweigungsfaktor dem Idealfall einer perfekten Zugreihenfolge?

d. Implementieren Sie einen selektiven Suchalgorithmus wie zum Beispiel B* (Berliner, 1979), konspirative Zahlensuche (McAllester, 1988) oder MGSS* (Russell und Wefald, 1989) und vergleichen Sie dessen Leistung mit A*.

12 Beschreiben Sie, wie sich die Minimax- und Alpha-Beta-Algorithmen für Zweispieler-Nichtnullsummen-Spiele ändern, in denen jeder Spieler über seine eigene Nutzenfunktion verfügt und Nutzenfunktionen beiden Spielern bekannt sind. Kann ein Knoten durch Alpha-Beta gekürzt werden, wenn es keine Beschränkungen hinsichtlich der beiden Endnutzenwerte gibt? Wie sieht es aus, wenn sich die Nutzenfunktionen der Spieler in jedem Zustand zu einer Zahl zwischen den Konstanten $-k$ und k summieren, wodurch das Spiel nahezu zu einem Nullsummenspiel wird?

13 Entwickeln Sie einen formalen Korrektheitsbeweis für Alpha-Beta-Kürzung. Betrachten Sie dazu die Situation in ▶ Abbildung 5.18. Die Frage ist, ob der Knoten n_j gekürzt werden soll, bei dem es sich um einen MAX-Knoten und einen Nachfolger des Knotens n_1 handelt. Die grundlegende Idee ist, ihn ausschließlich dann zu kürzen, wenn gezeigt werden kann, dass der Minimax-Wert von n_1 unabhängig vom Wert n_j ist.

a. Modus n_1 nimmt den Minimalwert seiner untergeordneten Werte an:

$$n_1 = \min\left(n_2, n_{21}, \ldots, n_{2b_2}\right).$$

Finden Sie einen vergleichbaren Ausdruck für n_2 und damit einen Ausdruck für n_1 in Hinblick auf n_j.

b. Sei l_i der minimale (oder maximale) Wert der Knoten *links* von Knoten n_i in der Tiefe i, dessen Minimax-Wert bereits bekannt ist. Analog dazu sei r_i der minimale (oder maximale) Wert der noch nicht untersuchten Knoten rechts von n_i in der Tiefe i. Schreiben Sie Ihren Ausdruck für n_1 so um, dass die Werte von l_i und r_i verwendet werden.

c. Formulieren Sie nun den Ausdruck so, dass Sie zeigen können, dass um n_1 zu beeinflussen, n_j eine bestimmte Grenze, die sich von den l_i-Werten ableitet, nicht überschreiten darf.

d. Wiederholen Sie den Prozess für den Fall, in dem es sich bei n_j um einen MIN-Knoten handelt.

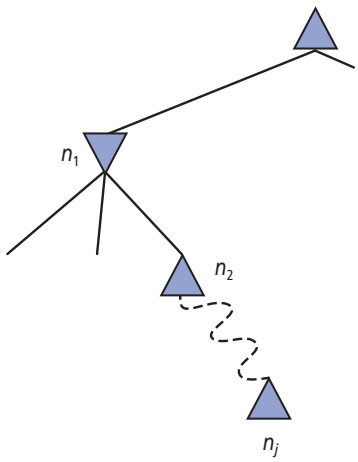


Abbildung 5.18: Situation, in der überprüft werden soll, ob Knoten n_j gekürzt wird.

14 Beweisen Sie, dass der Zeitbedarf für Alpha-Beta-Kürzung $O(b^{m/2})$ bei optimaler Zugreihenfolge beträgt, wobei m die maximale Tiefe des Spielbaumes ist.

15 Angenommen, Sie haben ein Schachprogramm, das 5 Millionen Knoten pro Sekunde auswerten kann. Finden Sie eine kompakte Darstellung eines Spielzustandes für die Speicherung in einer Transpositionstabelle. Wie viele Einträge passen in etwa in eine 1 GB große, in den Arbeitsspeicher geladene Tabelle? Ist das ausreichend für 3 Minuten Suche für einen Zug? Wie viele Tabellensuchen können Sie in der Zeit erledigen, die eine Bewertung dauert? Nehmen Sie nun an, dass die Transpositionstabelle auf Festplatte gespeichert wird. Wie viele Bewertungen könnten Sie in der Zeit vornehmen, die es dauert, eine Festplattensuche unter Verwendung standardmäßiger Festplatten auszuführen?

16 In dieser Frage geht es um das Kürzen in Spielen mit Zufallsknoten. ▶ Abbildung 5.19 zeigt den vollständigen Spielbaum für ein triviales Spiel. Nehmen Sie an, dass die Blattknoten von links nach rechts ausgewertet werden sollen und dass vor der Auswertung eines Blattknotens nichts über seinen Wert bekannt ist – der Bereich der möglichen Werte erstreckt sich von $-\infty$ bis ∞.

a. Kopieren Sie die Abbildung, markieren Sie den Wert aller internen Knoten und kennzeichnen Sie den besten Zug an der Wurzel mit einem Pfeil.

b. Ist es erforderlich, die Werte der siebenten und achten Blätter zu ermitteln, wenn die Werte der ersten sechs Blätter bekannt sind? Muss das achte Blatt ausgewertet werden, wenn die Werte der ersten sieben Blätter bekannt sind? Erläutern Sie Ihre Antworten.

c. Angenommen, die Werte der Blattknoten sind bekannt und liegen zwischen -2 und 2 (jeweils inklusive). Wie groß ist der Wertebereich für den linken Zufallsknoten, nachdem die beiden ersten Blätter ausgewertet worden sind?

d. Kreisen Sie alle Blätter ein, die unter der Annahme in (c) nicht ausgewertet werden müssen.

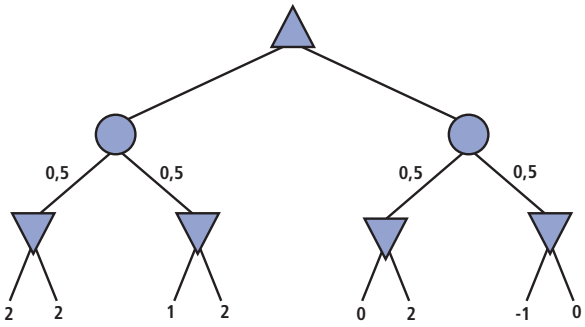


Abbildung 5.19: Der vollständige Spielbaum für ein triviales Spiel mit Zufallsknoten.

17 Implementieren Sie für das Kürzen von Spielbäumen mit Zufallsknoten den Expectiminimax-Algorithmus und den *-Alpha-Beta-Algorithmus, der von Ballard (1983) beschrieben wurde. Probieren Sie sie für ein Spiel wie etwa Backgammon aus und beurteilen Sie die Effektivität von *-Alpha-Beta.

18 Beweisen Sie, dass bei einer positiven linearen Transformation von Blattwerten (d.h. bei der Transformation eines Wertes von x in $ax + b$, wobei $a > 0$ ist) die Auswahl des Zuges in einem Spielbaum unverändert bleibt, selbst wenn Zufallsknoten vorhanden sind.

19 Betrachten Sie die folgende Vorgehensweise für die Auswahl von Zügen in Spielen mit Zufallsknoten:

- Erzeugen Sie Würfelfolgen (etwa 50) bis zu einer geeigneten Tiefe (etwa 8).
- Mit bekannten Würfen wird der Spielbaum deterministisch. Lösen Sie für jede Würfelfolge den resultierenden deterministischen Spielbaum unter Verwendung von Alpha-Beta.
- Verwenden Sie die Ergebnisse, um den Wert jedes Zuges abzuschätzen und den besten auszuwählen.

Funktioniert diese Vorgehensweise? Warum (nicht)?

20 Im Folgenden besteht ein „Max"-Baum nur aus Max-Knoten, während ein „Expectimax"-Baum aus einem Max-Knoten an der Wurzel mit abwechselnden Ebenen von Zufalls- und Max-Knoten besteht. Bei Zufallsknoten sind sämtliche Ergebniswahrscheinlichkeiten ungleich null. Das Ziel besteht darin, *den Wert der Wurzel* durch eine Suche mit Tiefenbeschränkung zu *finden*.

a. Nehmen Sie Blattwerte als endlich, aber unbeschränkt an. Ist dann Kürzung (wie in Alpha-Beta) in einem Max-Baum überhaupt möglich? Geben Sie ein Beispiel an oder erläutern Sie, warum nicht.

b. Ist Kürzung in einem Expectimax-Baum unter den gleichen Bedingungen überhaupt möglich? Geben Sie ein Beispiel an oder erläutern Sie, warum nicht.

c. Ist Kürzung in einem Max-Baum möglich, wenn Blattwerte auf den Bereich [0, 1] beschränkt werden? Geben Sie ein Beispiel an oder erläutern Sie, warum nicht.

d. Ist Kürzung in einem Expectimax-Baum möglich, wenn Blattwerte auf den Bereich [0, 1] beschränkt werden? Geben Sie ein Beispiel an (das sich qualitativ von Ihrem Beispiel in (e) unterscheidet, falls zutreffend) oder erläutern Sie, warum nicht.

e. Ist Kürzung in einem Max-Baum möglich, wenn Blattwerte nur nicht-negativ sein dürfen? Geben Sie ein Beispiel an oder erläutern Sie, warum nicht.

f. Ist Kürzung in einem Expectimax-Baum möglich, wenn Blattwerte nur nicht-negativ sein dürfen? Geben Sie ein Beispiel an oder erläutern Sie, warum nicht.

g. Betrachten Sie die Ergebnisse eines Zufallsknotens in einem Expectimax-Baum. Durch welche der folgenden Auswertungsreihenfolgen ergeben sich höchstwahrscheinlich Kürzungsmöglichkeiten: (i) kleinste Wahrscheinlichkeit zuerst, (ii) höchste Wahrscheinlichkeit zuerst, (iii) macht keinen Unterschied?

21 Welche der folgenden Aussagen sind wahr und welche falsch? Geben Sie kurze Erläuterungen an.

a. In einem vollständig beobachtbaren, Nullsummenspiel zwischen zwei perfekt rationalen Spielern, die abwechselnd am Zug sind, hilft es nicht, dass der erste Spieler die Strategie kennt, die der zweite Spieler wählt – d.h., welchen Zug der zweite Spieler nach dem Zug des ersten Spielers ausführt.

b. In einem partiell beobachtbaren, Nullsummenspiel zwischen zwei perfekt rationalen Spielern, die abwechselnd am Zug sind, hilft es nicht, wenn der erste Spieler weiß, welchen Zug der zweite Spieler auf den ihm bekannten Zug des ersten Spielers hin ausführt.

c. Ein perfekt rationaler Backgammon-Agent verliert nie.

22 Betrachten Sie sorgfältig das Zusammenspiel zwischen Zufallsereignissen und partiellen Informationen in jedem der in Übung 5.4 betrachteten Spiele.

a. Für welches Spiel ist das Expectiminimax-Standardmodell am besten geeignet? Implementieren Sie den Algorithmus und führen Sie ihn in Ihrem Spieleagenten aus. Wenden Sie dafür geeignete Abänderungen der Spieleumgebung an.

b. Für welches wäre das in Übung 6.8 beschriebene Schema geeignet?

c. Diskutieren Sie, wie Sie mit der Tatsache zurechtkommen könnten, dass die Spieler in einigen der Spiele nicht dasselbe Wissen über den aktuellen Zustand besitzen.

Probleme unter Rand- oder Nebenbedingungen

6

ÜBERBLICK

In diesem Kapitel erfahren Sie, dass man zu zahlreichen leistungsfähigen neuen Suchmethoden und einem besseren Verständnis für die Struktur und die Komplexität eines Problems gelangt, wenn man die Zustände nicht nur als kleine Blackboxes behandelt.

In den *Kapiteln 3* und *4* ging es um das Konzept, dass Probleme gelöst werden können, indem man in einem **Zustandsraum** sucht. Diese Zustände können durch domänenspezifische Heuristiken ausgewertet und auf ihre Eignung als Zielzustand hin getestet werden. Aus der Perspektive des Suchalgorithmus ist jedoch jeder Zustand atomar oder unteilbar – eine Blackbox ohne erkennbare interne Struktur.

Dieses Kapitel beschreibt einen Weg, um ein breites Spektrum von Problemen effizienter zu lösen. Wir verwenden eine **faktorisierte Darstellung** für jeden Zustand: einen Satz von Variablen, die jeweils einen Wert besitzen. Ein Problem ist gelöst, wenn jede Variable einen Wert hat, der sämtliche Einschränkungen hinsichtlich der Variablen erfüllt. Ein auf diese Weise beschriebenes Problem ist ein sogenanntes **Problem unter Rand- und Nebenbedingungen** (**Constraint Satisfaction Problem**, **CSP**).

CSP-Suchalgorithmen nutzen die Struktur von Zuständen und verwenden *universelle* statt *problemspezifischen* Heuristiken, um die Lösung komplexer Probleme zu erlauben. Der Kerngedanke besteht darin, große Teile des Suchraumes auf einmal zu eliminieren, indem Variablen-/Wert-Kombinationen ermittelt werden, die die Bedingungen verletzen.

6.1 Probleme unter Rand- und Nebenbedingungen – Definition

Ein Problem unter Rand- und Nebenbedingungen besteht aus den drei Komponenten X, D und C:

X ist eine Menge von Variablen (X_1, X_2, ..., X_n).

D ist eine Menge von Domänen, (D_1, ..., D_n), eine für jede Variable.

C ist eine Menge von Randbedingungen (Constraints), die zulässige Wertekombinationen spezifizieren.

Jede Domäne D_i besteht aus einer Menge zulässiger Werte $\{v_1, ..., v_k\}$ für Variable X_i. Jede Randbedingung C_i besteht aus einem Paar (*scope*, *rel*), in dem *scope* ein Tupel von Variablen darstellt, die an der Randbedingung beteiligt sind, und die Relation *rel* die Werte definiert, die diese Variablen annehmen können. Eine Relation lässt sich als explizite Liste von Werten ausdrücken, die die Bedingung erfüllen, oder als abstrakte Relation, die zwei Operationen unterstützt: Testen, ob ein Tupel ein Mitglied der Relation ist, und Enumerieren (Aufzählen) der Mitglieder der Relation. Haben zum Beispiel sowohl X_1 als auch X_2 die Domäne $\{A, B\}$, kann man die Randbedingung, die besagt, dass die beiden Variablen unterschiedliche Werte haben müssen, als $\langle(X_1 , X_2), [(A, B), (B, A)]\rangle$ oder als (X_1, X_2), $X_1 \neq X_2$ formulieren.

Um ein CSP zu lösen, müssen wir einen Zustandsraum und das Konzept einer Lösung definieren. Jeder Zustand in einem CSP wird durch eine Zuweisung von Werten zu einigen oder allen Variablen definiert. Ein Zustand des Problems wird durch eine **Zuweisung** von Werten für einige oder alle Variablen definiert, $\{X_i = v_i, X_j = v_j, ...\}$. Eine Zuweisung, die keine Randbedingungen verletzt, wird auch als **konsistente** oder erlaubte Zuweisung bezeichnet. Bei einer **vollständigen Zuweisung** sind sämtlichen

Variablen Werte zugewiesen und eine **Lösung** für ein CSP ist eine konsistente, vollständige Zuweisung. Eine **partielle Zuweisung** weist Werte nur einigen Variablen zu.

6.1.1 Beispielproblem: Färben von Karten

Angenommen, es gefällt uns nicht mehr in Rumänien und wir betrachten eine Karte von Australien, auf der alle Bundesstaaten und Territorien angezeigt werden (siehe ▶ Abbildung 6.1(a)). Wir haben die Aufgabe, jedes Gebiet rot, grün oder blau so einzufärben, dass keine zwei benachbarten Gebiete dieselbe Farbe haben. Um das Ganze als CSP zu formulieren, definieren wir die Variablen entsprechend der Bundesstaaten:

$$X = \{WA, NT, Q, NSW, V, SA, T\}.$$

Die Domäne jeder Variablen sei die Menge $D_i = \{rot, gruen, blau\}$. Die Randbedingungen fordern, dass benachbarte Gebiete unterschiedliche Farben haben. Da zwischen den Gebieten neun Grenzen existieren, gibt es neun Randbedingungen:

$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V,\\ WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}.$$

Hier verwenden wir Kurzschreibweisen. So steht $SA \neq WA$ für $\langle(SA, WA), SA \neq WA)\rangle$, wobei sich $SA \neq WA$ wiederum wie folgt vollständig aufzählen lässt:

$$\{(rot, gruen), (rot, blau), (gruen, rot), (gruen, blau), (blau, rot), (blau, gruen)\}.$$

Es gibt viele mögliche Lösungen für dieses Problem, wie zum Beispiel:

$$\{WA = rot, NT = gruen, Q = rot, NSW = gruen, V = rot, SA = blau, T = rot\}.$$

Hilfreich ist es, sich ein CSP als **Constraint-Graphen** wie in ▶ Abbildung 6.1(b) gezeigt vorzustellen. Die Knoten dieses Graphen entsprechen den Variablen des Problems und eine Verknüpfung (oder: Kante) verbindet je zwei Variablen, die zu einer Beschränkung gehören.

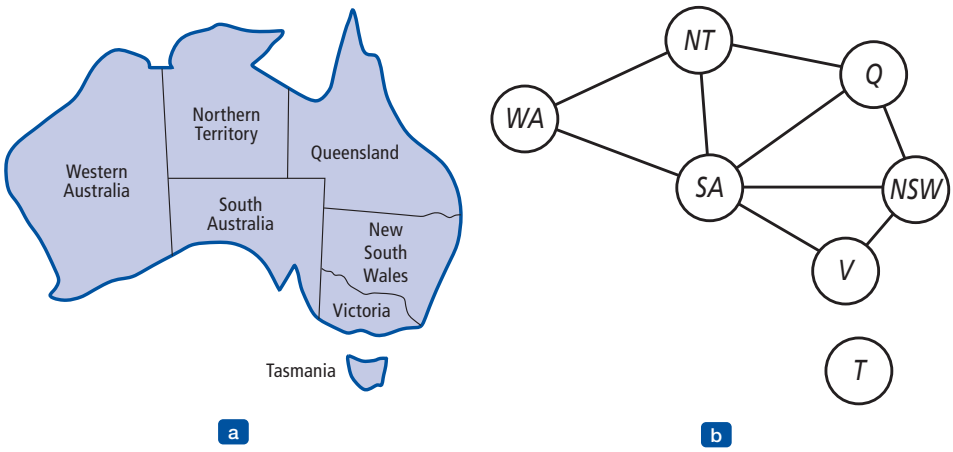


Abbildung 6.1: (a) Die wichtigsten Bundesstaaten und Territorien von Australien. Die Einfärbung der Karte kann als Problem unter Rand- und Nebenbedingungen (CSP) betrachtet werden. Das Ziel dabei ist, jedem Gebiet eine Farbe zuzuordnen, sodass keine benachbarten Gebiete dieselbe Farbe haben. (b) Das Problem der Karteneinfärbung als Constraint-Graph.

Warum formulieren wir ein Problem als CSP? Zum einen liefern CSPs eine natürliche Darstellung für die vielfältigsten Probleme. Wenn Sie bereits über ein System zur Lösung von CSPs verfügen, ist es oftmals einfacher, damit ein Problem zu lösen, als eine spezielle Lösung zu entwerfen, die ein anderes Suchverfahren verwendet. Außerdem können CSP-Lösungsmodule schneller als Zustandsraum-Suchmodule arbeiten, da das CSP-Lösungsmodul große Abschnitte des Suchraumes in kurzer Zeit eliminieren kann. Haben wir zum Beispiel im Problem Australien $\{SA = blau\}$ gewählt, können wir schließen, dass keine der fünf Nachbarvariablen den Wert *blau* annehmen kann. Ohne von der Beschränkungsweitergabe (Constraint Propagation) zu propagieren, müsste ein Suchverfahren $3^5 = 243$ Zuweisungen für die fünf Nachbarvariablen betrachten; mit Beschränkungsweitergabe brauchen wir *blau* als Wert überhaupt nicht zu berücksichtigen, sodass nur $2^5 = 32$ Zuweisungen einzubeziehen sind – eine Verringerung von 87%.

Bei normaler Zustandsraumsuche können wir nur fragen: Ist dieser konkrete Zustand ein Ziel? Nein? Wie sieht es mit diesem Zustand aus? Haben wir dagegen bei CSPs herausgefunden, dass eine partielle Zuweisung keine Lösung ist, können wir sofort weitere Verfeinerungen der partiellen Zuweisung verwerfen. Darüber hinaus ist zu sehen, warum die Zuweisung keine Lösung ist – wir sehen, welche Variablen eine Bedingung verletzen, und können unsere Aufmerksamkeit auf die relevanten Variablen konzentrieren. Im Ergebnis lassen sich viele Probleme, die für normale Zustandsraumsuche nicht behandelbar sind, schnell lösen, wenn sie als CSP formuliert werden.

6.1.2 Beispielproblem: Fertigungsablaufplanung

Fabriken haben das Problem, die an einem Tag anfallenden Arbeitsaufträge zu planen, wobei verschiedene Bedingungen einzuhalten sind. In der Praxis werden viele dieser Probleme mit CSP-Techniken gelöst. Nehmen wir als Beispiel das Problem, den Montageablauf eines Autos zu planen. Der gesamte Job setzt sich aus einzelnen Aufgaben (Tasks) zusammen und jede Aufgabe lässt sich als Variable modellieren, wobei jede Variable den Beginn der Aufgabe als ganze Anzahl von Minuten angibt. Randbedingungen können verlangen, dass eine Aufgabe vor einer anderen stattfinden muss – zum Beispiel muss ein Rad montiert sein, bevor sich die Radkappe aufsetzen lässt – und dass nur eine entsprechende Anzahl von Aufgaben auf einmal ablaufen kann. Außerdem können Randbedingungen spezifizieren, dass eine Aufgabe eine bestimmte Zeitspanne bis zur Fertigstellung benötigt.

Wir betrachten hier einen kleinen Abschnitt der Automontage, der aus 15 Aufgaben besteht: Montieren der Achsen (vorn und hinten), Befestigen aller vier Räder (rechts und links, vorn und hinten), Anziehen der Radmuttern an jedem Rad, Befestigen der Radkappen und Inspektion der Endmontage. Diese Aufgaben können wir mit 15 Variablen darstellen:

$$X = \{Achse_{V}, Achse_{H}, Rad_{RV}, Rad_{LV}, Rad_{RH}, Rad_{LH}, Muttern_{RV}, Muttern_{LV}, Muttern_{RH}, Muttern_{LH}, Kappe_{RV}, Kappe_{LV}, Kappe_{RH}, Kappe_{LH}, Inspektion\}.$$

Der Wert jeder Variablen ist die Zeit, zu der die Aufgabe beginnt. Als Nächstes stellen wir **Reihenfolgebeschränkungen** zwischen einzelnen Aufgaben dar. Immer wenn eine Aufgabe T_1 vor Aufgabe T_2 stattfinden muss und Aufgabe T_1 eine Zeitdauer d_1 bis zum Abschluss benötigt, fügen wir eine arithmetische Randbedingung in der Form $T_1 + d_1 \leq T_2$ hinzu.

In unserem Beispiel müssen die Achsen montiert sein, bevor sich die Räder befestigen lassen, und das Montieren einer Achse dauert 10 Minuten. Wir schreiben also:

$$Achse_V + 10 \leq Rad_{RV}\,;\; Achse_V + 10 \leq Rad_{LV}$$

$$Achse_H + 10 \leq Rad_{RH}\,;\; Achse_H + 10 \leq Rad_{LH}.$$

Als Nächstes stellen wir für jedes Rad fest, dass das Rad montiert werden muss (was 1 Minute dauert), dann die Radmuttern festzuziehen sind (2 Minuten) und schließlich die Radkappen aufzusetzen sind (1 Minute, doch bisher noch nicht dargestellt):

$$Rad_{RV} + 1 \leq Muttern_{RV}\,;\; Muttern_{RV} + 2 \leq Kappe_{RV}$$

$$Rad_{LV} + 1 \leq Muttern_{LV}\,;\; Muttern_{LV} + 2 \leq Kappe_{LV}$$

$$Rad_{RH} + 1 \leq Muttern_{RH}\,;\; Muttern_{RH} + 2 \leq Kappe_{RH}$$

$$Rad_{LH} + 1 \leq Muttern_{LH}\,;\; Muttern_{LH} + 2 \leq Kappe_{LH}$$

Es sei angenommen, dass für die Montage der Räder vier Arbeiter vorhanden sind, die jedoch ein Werkzeug zum Positionieren der Achse gemeinsam nutzen müssen. Wir brauchen hier eine **disjunktive Beschränkung**, um auszudrücken, dass sich die Bearbeitungszeiten an $Achse_V$ und $Achse_H$ nicht überlappen dürfen; entweder kommt die eine zuerst an die Reihe oder die andere:

$$(Achse_V + 10 \leq Achse_H) \quad \text{oder} \quad (Achse_H + 10 \leq Achse_V).$$

Diese Einschränkung sieht komplizierter aus, da sie Arithmetik und Logik kombiniert. Doch sie lässt sich noch auf eine Menge von Wertepaaren reduzieren, die $Achse_V$ und $Achse_H$ annehmen können.

Außerdem müssen wir feststellen, dass die Inspektion als Letztes kommt und 3 Minuten dauert. Für jede Variable außer *Inspektion* fügen wir eine Einschränkung der Form $X + d_X \leq Inspektion$ hinzu. Schließlich sei als Forderung angenommen, dass die gesamte Montage in 30 Minuten zu erledigen ist. Dies lässt sich erreichen, indem die Domäne aller Variablen begrenzt wird:

$$D_i = \{1, 2, 3, \ldots, 27\}.$$

Das hier dargestellte Problem ist trivial zu lösen, doch wurden CSPs auch auf derartige Arbeitsablaufplanungsprobleme mit Tausenden von Variablen angewandt. In manchen Fällen gibt es komplizierte Einschränkungen, die im CSP-Formalismus schwierig zu spezifizieren sind. Dann kommen fortgeschrittenere Planungsverfahren zum Einsatz, wie *Kapitel 11* erläutert.

6.1.3 Variationen des CSP-Formalismus

Die einfachste Art von CSP verwendet Variablen, die **diskret** sind und **endliche Domänen** aufweisen. Probleme der Karteneinfärbung und Planungen mit Zeitbeschränkungen gehören zu dieser Art. Das in *Kapitel 3* beschriebene 8-Damen-Problem kann ebenfalls als CSP mit endlicher Domäne betrachtet werden, wobei die Variablen Q_1, ..., Q_8 die Positionen der acht Damen in den Spalten 1, ..., 8 darstellen und jede Variable die Domäne $D_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$ hat.

Eine diskrete Domäne kann **unendlich** sein, wie zum Beispiel die Menge der Ganzzahlen oder die Menge der Zeichenfolgen. (Wenn wir beim Problem der Fertigungsablaufpla-

nung keinen Endtermin festgelegt hätten, gäbe es eine unendliche Anzahl von Startzeiten für jede Variable.) Bei unendlichen Domänen ist es nicht mehr möglich, Beschränkungen zu beschreiben, indem man alle erlaubten Wertekombinationen aufzählt. Stattdessen muss eine **Beschränkungssprache** verwendet werden, die direkt Einschränkungen wie zum Beispiel $T_1 + d_1 \leq T_2$ versteht, ohne die Menge der Paare zulässiger Werte für (T_1, T_2) aufzuzählen. Es gibt spezielle Lösungsalgorithmen (die wir hier nicht besprechen werden) für **lineare Beschränkungen** bei ganzzahligen Variablen – d.h. Beschränkungen wie die oben gezeigte, bei denen jede Variable nur in linearer Form erscheint. Es kann gezeigt werden, dass es keinen Algorithmus für die Lösung allgemeiner **nichtlinearer Beschränkungen** bei ganzzahligen Variablen gibt.

Probleme unter Rand- und Nebenbedingungen mit **stetigen Domänen** treten in der Praxis sehr häufig auf und werden im Bereich der Operationsforschung ausführlich betrachtet. Beispielsweise ist für die Ablaufplanung der Experimente des Weltraumteleskops Hubble ein sehr präzises Timing der Beobachtungen erforderlich; der Anfang und das Ende jeder Beobachtung und jedes Manövers sind stetigwertige Variablen, die einer Vielzahl von astronomischen, prioritätsgesteuerten und leistungsbezogenen Beschränkungen gehorchen müssen. Die am besten erforschte Kategorie von CSPs mit stetiger Domäne sind die Probleme der **linearen Programmierung**, wobei Beschränkungen lineare Gleichungen oder Ungleichungen sein müssen. Probleme der linearen Programmierung können in polynomialer Zeit in Bezug auf die Anzahl der Variablen gelöst werden. Probleme mit anderen Arten von Beschränkungen und Zielfunktionen wurden ebenfalls untersucht – quadratische Programmierung, konische Programmierung zweiter Ordnung usw.

Neben der Betrachtung der Variablentypen, die in CSPs erscheinen können, ist es sinnvoll, die Beschränkungstypen zu untersuchen. Der einfachste Typ ist die **unäre Beschränkung**, die den Wert einer einzelnen Variablen kontrolliert. Beispielsweise könnte es im Karteneinfärbungsproblem der Fall sein, dass die Südaustralier eine unüberwindbare Abneigung gegen die Farbe Grün haben. Dies lässt sich mit der unären Beschränkung $\langle(SA), SA \neq gruen\rangle$ ausdrücken.

Eine **binäre Beschränkung** bezieht sich auf zwei Variablen. Beispielsweise ist $SA \neq NSW$ eine binäre Beschränkung. Ein binäres CSP ist ein Problem mit ausschließlich binären Beschränkungen; es kann wie in Abbildung 6.1(b) gezeigt als Constraint-Graph dargestellt werden.

Es lassen sich auch Beschränkungen höherer Ordnung beschreiben. So schreibt zum Beispiel die ternäre Beschränkung *Between*(X, Y, Z) vor, dass der Wert von Y zwischen X und Z liegen soll.

Bei einer **globalen Beschränkung** handelt es sich um eine Beschränkung, die eine beliebige Anzahl von Variablen beinhalten kann. (Der Name hat sich eingebürgert, ist aber etwas irreführend, da die Beschränkung nicht *alle* Variablen in einem Problem einschließen muss.) Eine der gebräuchlichsten globalen Beschränkungen ist *Alldiff*, die besagt, dass alle in der Beschränkung beteiligten Variablen unterschiedliche Werte haben müssen. In Sudoku-Problemen (siehe *Abschnitt 6.2.6*) müssen alle Variablen in einer Zeile oder Spalte einer *Alldiff*-Beschränkung genügen. Ein anderes Beispiel sind **kryptoarithmetische** Rätsel (siehe ▶ Abbildung 6.2(a)). Jeder Buchstabe in einem kryptoarithmetischen Rätsel repräsentiert eine andere Ziffer. Für den Fall in Abbildung 6.2(a) wird dies als globale Beschränkung *Alldiff* (F,T,U,W,R,O) ausgedrückt. Die Additionsbeschränkungen für die vier Spalten des Rätsels lassen sich wie folgt als n-stellige Beschränkungen schreiben:

$$O + O = R + 10 \cdot C_{10}$$
$$C_{10} + W + W = U + 10 \cdot C_{100}$$
$$C_{100} + T + T = O + 10 \cdot C_{1000}$$
$$C_{1000} = F.$$

Dabei sind C_{10}, C_{100} und C_{1000} Hilfsvariablen, die die Übertragsziffer in die Zehner-, Hunderter- bzw. Tausender-Spalten darstellen. Diese können in einem **Beschränkungshypergraphen** dargestellt werden, wie in ▶ Abbildung 6.2(b) gezeigt. Ein Hypergraph besteht aus normalen Knoten (den Kreisen in der Abbildung) und Hyperknoten (den Quadraten), die n-stellige Beschränkungen darstellen.

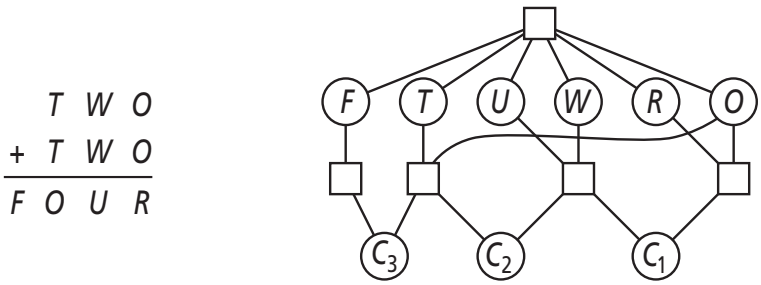


Abbildung 6.2: (a) Ein kryptoarithmetisches Problem. Jeder Buchstabe steht für eine andere Ziffer; das Ziel dabei ist, eine Ersetzung der Buchstaben durch Ziffern zu finden, sodass die resultierende Summe arithmetisch korrekt ist – mit der zusätzlichen Einschränkung, dass keine führenden Nullen erlaubt sind. (b) Der Beschränkungshypergraph für das kryptoarithmetische Problem, der die *Alldiff*-Beschränkung (Quadrat an der Spitze) sowie die Beschränkungen für die Spaltenaddition (die vier Quadrate in der Mitte) zeigt. Die Variablen C_1, C_2 und C_3 verkörpern die Übertragsziffern für die drei Spalten.

Wie Sie in Übung 6.5 beweisen sollen, lässt sich alternativ jede Beschränkung mit höherer Ordnung und endlicher Domäne zu einer Menge binärer Beschränkungen reduzieren, wenn genügend Hilfsvariablen eingeführt werden. Wir könnten also jedes CSP in ein Problem überführen, das nur aus binären Beschränkungen besteht. Das vereinfacht die Algorithmen. Eine andere Möglichkeit, ein n-stelliges CSP in ein binäres Problem zu konvertieren, ist die **duale Graphen**-Transformation: Erstellen Sie einen neuen Graphen, in dem es genau eine Variable für jede Beschränkung im ursprünglichen Graphen und genau eine binäre Beschränkung für jedes Beschränkungspaar im ursprünglichen Graphen mit gemeinsamen Variablen gibt. Umfasst zum Beispiel der ursprüngliche Graph die Variablen $\{X, Y, Z\}$ und die Beschränkungen $\langle(X, Y, Z), C_1\rangle$ und $\langle(X, Y), R_1\rangle$, besteht der duale Graph aus den Variablen $\{C_1, C_2\}$ mit der binären Beschränkung $<(C_1, C_2), R_1>$, wobei R_1 eine neue Relation darstellt, die die Beschränkung zwischen C_1 und C_2 auf der Basis der gemeinsamen Variablen (X, Y, Z) definiert.

Es gibt allerdings zwei Gründe, warum eine globale Beschränkung wie zum Beispiel *Alldiff* einer Menge von binären Beschränkungen vorzuziehen ist. Erstens ist es leichter und weniger fehleranfällig, die Problembeschreibung mithilfe von *Alldiff* zu formulieren. Zweitens ist es möglich, spezielle Inferenzalgorithmen für globale Beschränkungen zu entwerfen, die für eine Menge primitiverer Beschränkungen nicht verfügbar sind. *Abschnitt 6.2.5* beschäftigt sich mit diesen Inferenzalgorithmen.

Die bisher beschriebenen Beschränkungen waren alle absolute Beschränkungen, deren Verletzung eine mögliche Lösung ausschließt. Viele CSPs aus der realen Welt beinhalten **Prioritätsbeschränkungen**, die angeben, welche Lösungen bevorzugt sind. In einer Aufgabenstellung beispielsweise, wo ein Stundenplan für eine Universität erstellt werden soll, gibt es absolute Beschränkungen, dass kein Professor zur gleichen Zeit zwei Vorlesungen halten kann. Doch wir können auch Prioritätsbeschränkungen erlauben: Prof. R möchte vielleicht lieber morgens unterrichten, während Prof. N den Abendunterricht vorzieht. Ein Stundenplan, nach dem Professor R um 14:00 unterrichten müsste, wäre immer noch eine zulässige Lösung (es sei denn, Professor R ist der Dekan), allerdings ist sie nicht optimal. Prioritätsbeschränkungen können oft als Kosten für einzelne Variablenzuweisungen kodiert werden – wenn man beispielsweise einem Nachmittagseintrag für Prof. R Kosten von 2 Punkten im Hinblick auf die allgemeine Zielfunktion zuweist, während Vormittagseinträge nur 1 Punkt kosten. Mit dieser Formulierung können CSPs mit Prioritäten unter Verwendung von optimierenden Suchmethoden gelöst werden, egal ob pfadbasiert oder lokal. Wir sprechen hierbei von einem **Beschränkungsoptimierungsproblem** (**Constraint Optimization Problem**, COP). Probleme der linearen Programmierung haben mit derartigen Optimierungen zu tun.

6.2 Beschränkungsweitergabe: Inferenz in CSPs

In der normalen Zustandsraumsuche kann ein Algorithmus nur eines tun: Suchen. In CSPs gibt es eine Wahlmöglichkeit: Ein Algorithmus kann suchen (eine neue Variablenzuweisung aus mehreren Möglichkeiten auswählen) oder einen speziellen Typ von **Inferenz** – eine sogenannte **Beschränkungsweitergabe** – ausführen, d.h. mithilfe der Beschränkungen die Anzahl der erlaubten Werte für eine Variable reduzieren, was wiederum die erlaubten Werte für eine andere Variable verringern kann, usw. Beschränkungsweitergabe lässt sich mit der Suche verflechten oder als Vorverarbeitungsschritt durchführen, bevor die Suche beginnt. Manchmal kann diese Vorverarbeitung bereits das gesamte Problem lösen, sodass überhaupt keine Suche erforderlich ist.

Der Kerngedanke ist **lokale Konsistenz**. Wenn wir jede Variable als Knoten in einem Graphen behandeln (siehe Abbildung 6.1(b)) und jede binäre Beschränkung als Kante, bewirkt das Erzwingen lokaler Konsistenz in jedem Teil des Graphen, dass inkonsistente Werte im Graphen durchweg zu eliminieren sind. Es gibt verschiedene Arten der lokalen Konsistenz, die wir nun nacheinander behandeln.

6.2.1 Knotenkonsistenz

Eine einzelne Variable (entsprechend einem Knoten im CSP-Netz) ist **knotenkonsistent**, wenn alle Werte in der Domäne der Variablen die unären Beschränkungen der Variablen erfüllen. Zum Beispiel beginnt im Problem der Karteneinfärbung für Australien (Abbildung 6.1), bei dem die Südaustralier kein Grün mögen, die Variable *SA* zunächst mit der Domäne {*rot*, *gruen*, *blau*} und wir können sie knotenkonsistent machen, indem wir *gruen* eliminieren, wodurch *SA* mit der reduzierten Domäne {*rot*, *blau*} verbleibt. Ein Netz ist knotenkonsistent, wenn jede Variable im Netz knotenkonsistent ist.

Es ist immer möglich, alle unären Beschränkungen in einem CSP zu eliminieren, indem Knotenkonsistenz ausgeführt wird. Ebenso ist es möglich, alle n-stelligen Beschränkungen in binäre Beschränkungen zu transformieren (siehe Übung 6.5). Deswegen ist es

üblich, CSP-Lösungsmodule zu definieren, die ausschließlich mit binären Beschränkungen arbeiten. Sofern nichts anderes erwähnt wird, gilt diese Annahme für den Rest dieses Kapitels.

6.2.2 Kantenkonsistenz

Eine Variable in einem CSP ist **kantenkonsistent**, wenn jeder Wert in ihrer Domäne die binären Beschränkungen der Variablen erfüllt. Formeller ausgedrückt ist X_i kantenkonsistent in Bezug auf eine andere Variable X_j, wenn es für jeden Wert in der aktuellen Domäne D_i einen Wert in der Domäne D_j gibt, der die binäre Beschränkung für die Kante (X_i, X_j) erfüllt. Ein Netz ist kantenkonsistent, wenn jede Variable mit jeder anderen Variablen kantenkonsistent ist. Nehmen Sie zum Beispiel die Beschränkung $Y = X^2$, bei der die Domäne sowohl von X als auch von Y die Menge der Ziffern ist. Diese Beschränkung lässt sich explizit als

$$\langle (X, Y), \{(0, 0), (1, 1), (2, 4), (3, 9))\}\rangle$$

schreiben.

Um X kantenkonsistent in Bezug auf Y zu machen, reduzieren wir die Domäne von X auf $\{0, 1, 2, 3\}$. Wenn wir auch Y kantenkonsistent in Bezug auf X machen, wird die Domäne von Y zu $\{0, 1, 4, 9\}$ und das gesamte CSP ist kantenkonsistent.

Andererseits kann Kantenkonsistenz nichts für das Problem der Karteneinfärbung von Australien tun. Sehen Sie sich dazu die folgende Ungleichungsbeschränkung zu (*SA*, *WA*) an:

$$\{(rot, gruen), (rot, blau), (gruen, rot), (gruen, blau), (blau, rot), (blau, gruen)\}.$$

Unabhängig davon, welchen Wert Sie für *SA* (oder für *WA*) wählen, gibt es einen gültigen Wert für die andere Variable. Die Anwendung der Kantenkonsistenz hat also keine Wirkung auf die Domänen der beiden Variablen.

Der bekannteste Algorithmus für Kantenkonsistenz heißt AC-3 (siehe ▶ Abbildung 6.3). Um jede Variable kantenkonsistent zu machen, verwaltet der AC-3-Algorithmus eine Warteschlange aller zu betrachtenden Kanten. (Tatsächlich ist die Reihenfolge der Bearbeitung nicht wichtig, sodass die Datenstruktur praktisch eine Menge ist, doch hat sich die Bezeichnung Warteschlange eingebürgert.) Anfangs enthält die Warteschlange sämtliche Kanten im CSP. AC-3 ruft dann eine beliebige Kante (X_i, X_j) aus der Warteschlange ab und macht X_i kantenkonsistent in Bezug auf X_j. Bleibt dabei D_i unverändert, geht der Algorithmus einfach zur nächsten Kante weiter. Doch wenn dies D_i bereinigt (die Domäne verkleinert), fügen wir der Warteschlange sämtliche Kanten (X_k, X_i) hinzu, wobei X_k ein Nachbar von X_i ist. Dies ist erforderlich, weil die Änderung in D_i weitere Verringerungen in den Domänen von D_k ermöglichen kann, selbst wenn wir vorher X_k betrachtet haben. Wird D_i auf nichts reduziert, besitzt das gesamte CSP keine konsistente Lösung und AC-3 kann sofort eine Fehlermeldung zurückgeben. Andernfalls prüfen wir weiter und versuchen, Werte aus den Domänen von Variablen zu entfernen, bis keine weiteren Kanten mehr in der Warteschlange stehen. Zu diesem Zeitpunkt bleiben wir mit einem CSP zurück, das dem ursprünglichen CSP äquivalent ist – beide besitzen die gleichen Lösungen –, doch das kantenkonsistente CSP wird in den meisten Fällen schneller zu durchsuchen sein, weil die Domänen seiner Variablen kleiner sind.

```
function AC-3(csp) returns false, wenn eine Inkonsistenz gefunden wird,
                           andernfalls true
  inputs: csp, ein binäres CSP mit Komponenten (X, D, C)
  local variables: queue, eine Warteschlange mit Kanten,
                           anfangs alle Kanten in csp

  while queue nicht leer do
    (Xi, Xj) ← REMOVE-FIRST(queue)
    if REVISE(csp, Xi, Xj) then
      if Größe von Di = 0 then return false
      for each Xk in Xi.NEIGHBORS - {Xj} do
        (Xk, Xi) zu queue hinzufügen
  return true
```

```
function REVISE(csp, Xi, Xj) returns true, wenn wir die Domäne von Xi bereinigen
  revised ← false
  for each x in Di do
    if kein Wert y in Dj erlaubt (x, y), die Beschränkung zwischen
                                  Xi und Xj zu erfüllen then
      x aus Di löschen
      revised ← true
  return revised
```

Abbildung 6.3: Der Kantenkonsistenz-Algorithmus AC-3. Nach Anwendung von AC-3 ist entweder jede Kante kantenkonsistent oder eine Variable hat eine leere Domäne, d.h., das CSP kann nicht gelöst werden. Der Name „AC-3" wurde vom Erfinder des Algorithmus verwendet (Mackworth, 1977), weil es die dritte Version war, die er in seinem Buch entwickelt hat.

Die Komplexität von AC-3 lässt sich wie folgt analysieren. Wir nehmen ein CSP mit n Variablen, die jeweils eine maximale Domänengröße von d haben, und mit binären Beschränkungen (Kanten) an. Jede Kante (X_k, X_i) kann in die Warteschlange nur d-Mal eingefügt werden, weil X_i höchstens d Werte zum Löschen besitzt. Das Überprüfen der Konsistenz einer Kante kann in einer Zeit von $O(d^2)$ erfolgen, sodass wir im ungünstigsten Fall eine Gesamtzeit von $O(cd^3)$ bekommen.[1]

Das Konzept der Kantenkonsistenz lässt sich erweitern, um n-stellige und nicht nur binäre Beschränkungen zu behandeln. Man spricht dabei von verallgemeinerter Kantenkonsistenz oder manchmal auch – je nach Autor – von Hyperkantenkonsistenz. Eine Variable X_i wird in Bezug auf eine n-stellige Beschränkung **verallgemeinert kantenkonsistent**, wenn für jeden Wert v in der Domäne von X_i ein Tupel von Werten existiert, das Mitglied der Beschränkung ist, alle seine Werte aus den Domänen der korrespondierenden Variablen stammen und seine X_i-Komponente gleich v ist. Als Beispiel sei angenommen, dass alle Variablen die Domäne $\{0, 1, 2, 3\}$ haben. Um dann die Variable X konsistent zur Beschränkung $X < Y < Z$ zu machen, müssten wir 2 und 3 aus der Domäne von X entfernen, da die Beschränkung nicht erfüllt werden kann, wenn X den Wert 2 oder 3 hat.

1 Der AC-4-Algorithmus nach Mohr und Henderson (1986) läuft im ungünstigsten Fall mit einer Zeit von $O(cd^2)$, kann aber in durchschnittlichen Fällen langsamer als AC-3 sein – siehe Übung 6.12.

6.2.3 Pfadkonsistenz

Kantenkonsistenz kann viel dazu beitragen, die Domänen von Variablen zu reduzieren. Dabei wird manchmal eine Lösung gefunden (indem jede Domäne auf die Größe 1 verringert wird) und manchmal festgestellt, dass sich das CSP nicht lösen lässt (wenn irgendeine Domäne zur Größe 0 reduziert wird). Für andere Netze ist Kantenkonsistenz jedoch nicht in der Lage, genügend Schlüsse zu ziehen. Betrachten Sie das Problem der Karteneinfärbung von Australien, wobei aber nur zwei Farben – Rot und Blau – erlaubt sind. Hier kann Kantenkonsistenz nichts tun, weil jede Variable bereits kantenkonsistent ist: Jede kann rot mit blau am anderen Ende der Kante (oder umgekehrt) sein. Zweifellos gibt es aber keine Lösung für das Problem: Da Western Australia, Northern Territory und South Australia zusammenstoßen, brauchen wir allein dafür mindestens drei Farben.

Kantenkonsistenz engt die Domänen (unäre Beschränkungen) mithilfe der Kanten (binäre Beschränkungen) ein. Um bei Problemen wie der Karteneinfärbung weiterzukommen, brauchen wir ein stärkeres Konsistenzkonzept. **Pfadkonsistenz** strafft die binären Beschränkungen mithilfe impliziter Beschränkungen, die anhand von Variablentripeln hergeleitet werden.

Eine Menge aus zwei Variablen $\{X_i, X_j\}$ ist pfadkonsistent in Bezug auf eine dritte Variable X_m, wenn es für jede Zuweisung $\{X_i = a, X_j = b\}$, die mit den Beschränkungen auf $\{X_i, X_j\}$ konsistent ist, eine Zuweisung zu X_m gibt, die die Beschränkungen auf $\{X_i, X_m\}$ und $\{X_m, X_j\}$ erfüllt. Man spricht hier von Pfadkonsistenz, weil es sich so darstellt, als würde man auf einen Pfad von X_i nach X_j mit X_m in der Mitte blicken.

Sehen wir uns nun an, wie sich Pfadkonsistenz beim Einfärben der Australienkarte mit zwei Farben behauptet. Wir machen die Menge $\{WA, SA\}$ pfadkonsistent in Bezug auf *NT*. Zunächst zählen wir die konsistenten Zuweisungen zur Menge auf. In diesem Fall gibt es nur zwei: $\{WA = rot, SA = blau\}$ und $\{WA = blau, SA = rot\}$. Es ist zu erkennen, dass mit beiden dieser Zuweisungen *NT* weder rot noch blau sein kann (weil dies entweder mit *WA* oder mit *SA* in Konflikt käme). Da es keine gültige Auswahl für *NT* gibt, eliminieren wir beide Zuweisungen und haben letztlich keine gültigen Zuweisungen mehr für $\{WA, SA\}$. Somit wissen wir, dass es für dieses Problem keine Lösung geben kann. Der Algorithmus PC-2 (Mackworth, 1977) kommt zu Pfadkonsistenz in fast der gleichen Weise wie AC-3 zu Kantenkonsistenz. Aufgrund dieser Ähnlichkeit zeigen wir diesen Algorithmus hier nicht.

6.2.4 *k*-Konsistenz

Strengere Formen der Weitergabe können mit dem Konzept der sogenannten k-**Konsistenz** definiert werden. Ein CSP ist k-konsistent, wenn für jede Menge von $k-1$ Variablen und für jede konsistente Zuweisung zu diesen Variablen der k-ten Variablen immer ein konsistenter Wert zugewiesen werden kann. Die 1-Konsistenz besagt, dass sich bei gegebener leerer Menge jede Menge aus einer Variablen konsistent machen lässt: Dies bezeichnen wir als Knotenkonsistenz. 2-Konsistenz ist dasselbe wie Kantenkonsistenz. Für binär beschränkte Netze ist 3-Konsistenz dasselbe wie Pfadkonsistenz.

Ein CSP ist **streng** k-**konsistent**, wenn er k-konsistent ist und auch $(k-1)$-konsistent, $(k-2)$-konsistent, ... bis hin zu 1-konsistent. Angenommen, wir haben ein CSP-Problem mit n Knoten und machen es streng n-konsistent (d.h. streng k-konsistent für $k = n$). Das Problem können wir wie folgt lösen: Zuerst wählen wir einen konsistenten Wert für X_1. Dann sind wir garantiert in der Lage, einen Wert für X_2 zu wählen, weil der Graph 2-konsistent ist, für X_3, weil er 3-konsistent ist, usw. Für jede Variable X_i

müssen wir nur die d Werte in der Domäne durchsuchen, um einen Wert zu finden, der konsistent mit $X_1, ..., X_{i-1}$ ist. Eine Lösung finden wir garantiert in der Zeit $O(n^2 d)$. Natürlich gibt es nichts umsonst: Jeder Algorithmus für die Einrichtung der n-Konsistenz braucht im ungünstigsten Fall eine in n exponentielle Zeit. Darüber hinaus erfordert n-Konsistenz Speicherplatz, der exponentiell mit n zunimmt. Das Speicherproblem ist sogar noch schwerwiegender als die notwendige Zeit. In der Praxis ist die Ermittlung einer geeigneten Konsistenzebene häufig eine empirische Wissenschaft. Praktiker berechnen vor allem 2-Konsistenz und weniger häufig 3-Konsistenz.

6.2.5 Globale Beschränkungen

Wie bereits erwähnt, umfasst eine **globale Beschränkung** eine beliebige Anzahl von Variablen (aber nicht unbedingt alle Variablen). Globale Beschränkungen tauchen häufig in realen Problemen auf und können unter Verwendung spezieller Algorithmen abgehandelt werden, die effizienter sind als die bisher beschriebenen universellen Methoden. Die *Alldiff*-Beschränkung beispielsweise besagt, dass alle beteiligten Variablen unterschiedliche Werte haben müssen (wie bei dem oben vorgestellten kryptoarithmetischen Problem und den später noch gezeigten Sudoku-Rätseln). Eine einfache Form der Inkonsistenzüberprüfung für *Alldiff* funktioniert wie folgt: Wenn an der Beschränkung m Variablen beteiligt sind, sie insgesamt n mögliche unterschiedliche Werte haben und $m > n$ gilt, dann kann die Beschränkung nicht erfüllt werden. Das führt zum folgenden einfachen Algorithmus: Man entfernt zunächst eine Variable aus der Beschränkung, deren Domäne nur einen einzigen Wert enthält, und löscht den Wert dieser Variablen aus den Domänen der restlichen Variablen. Das wiederholt man, solange es Variablen mit einzelnen Werten gibt. Wird irgendwann eine leere Domäne erzeugt oder es sind mehr Variablen als Domänenwerte übrig, wurde eine Inkonsistenz erkannt.

Diese Methode kann die Inkonsistenz in der Zuweisung $\{WA = rot, NSW = rot\}$ für Abbildung 6.1 erkennen. Beachten Sie, dass die Variablen SA, NT und Q durch eine *Alldiff*-Beschränkung miteinander verbunden sind, weil jedes Paar zwei verschiedene Farben haben muss. Nach Anwendung von AC-3 mit der partiellen Zuweisung wird die Domäne jeder Variablen auf {*gruen*, *blau*} reduziert. Wir haben also drei Variablen und nur zwei Farben, womit die *Alldiff*-Beschränkung verletzt ist. Eine einfache Konsistenzprozedur für eine Beschränkung höherer Ordnung ist also manchmal effektiver als die Anwendung der Kantenkonsistenz auf eine äquivalente Menge binärer Beschränkungen. Für *Alldiff* gibt es noch komplexere Inferenzalgorithmen (siehe van Hoeve und Katriel, 2006), die mehr Beschränkungen weiterleiten, aber rechentechnisch wesentlich aufwändiger sind.

Eine andere wichtige Beschränkung höherer Ordnung ist die **Ressourcen-Beschränkung**, manchmal auch als *atmost*-Beschränkung bezeichnet. Als Beispiel sollen in einem Planungsproblem $P_1, ..., P_4$ die Anzahl der Personen bezeichnen, die vier bestimmten Aufgaben zugeordnet sind. Die Beschränkung, dass nicht mehr als zehn Personen insgesamt zugeordnet werden dürfen, wird als *atmost*(10, P_1, P_2, P_3, P_4,) geschrieben. Eine Inkonsistenz kann einfach erkannt werden, indem die Summe der Minimumwerte der aktuellen Domänen überprüft wird. Hat beispielsweise jede Variable die Domäne {3, 4, 5, 6}, kann die *atmost*-Beschränkung nicht erfüllt werden. Wir können die Konsistenz auch erzwingen, indem wir den Maximalwert aus jeder Domäne entfernen, wenn dieser nicht konsistent mit den Minimumwerten der anderen Domänen ist. Hat also jede Variable in unserem Beispiel die Domäne {2, 3, 4, 5, 6}, können die Werte 5 und 6 aus jeder Domäne entfernt werden.

Für große ressourcenbeschränkte Probleme mit ganzzahligen Werten – wie etwa logistische Probleme, wobei der Transport von Tausenden von Menschen in Hunderten von Fahrzeugen geplant werden soll – ist es normalerweise nicht möglich, die Domäne jeder Variablen als große Menge ganzzahliger Zahlen darzustellen und diese Menge durch Anwendung konsistenzüberprüfender Methoden schrittweise zu reduzieren. Stattdessen werden Domänen durch Ober- und Untergrenzen dargestellt und durch **Grenzenweitergabe** verwaltet. Angenommen, es gibt in einem Flugplanungsproblem zwei Flüge F_1 und F_2, deren Flugzeuge die Kapazitäten 165 bzw. 385 haben. Die anfänglichen Domänen für die Anzahl der Passagiere auf jedem Flug sind dann:

$$D_1 = [0, 165] \text{ und } D_2 = [0, 385].$$

Nehmen wir nun als zusätzliche Beschränkung an, dass die beiden Flüge insgesamt 420 Menschen transportieren müssen: $F_1 + F_2 = 420$. Geben wir die Grenzenbeschränkungen weiter, reduzieren wir die Domänen auf:

$$D_1 = [35, 165] \text{ und } D_2 = [255, 385].$$

Wir sagen, ein CSP ist **grenzenkonsistent**, wenn es für jede Variable X und sowohl für die nach unten als auch nach oben begrenzten Werte von X einen Wert von Y gibt, der die Beschränkung zwischen X und Y für jede Variable Y erfüllt. Diese Art der Grenzenweitergabe wird in praktischen Beschränkungsproblemen häufig angewendet.

6.2.6 Sudoku-Beispiel

Das beliebte **Sudoku**-Rätsel hat Millionen Leute in Probleme unter Rand- und Nebenbedingungen eingeführt, auch wenn sie es vielleicht nicht erkennen. Ein Sudoku-Spielfeld besteht aus 81 Feldern, von denen einige mit den Ziffern von 1 bis 9 gefüllt sind. Ziel des Rätsels ist es, alle übrigen Felder so zu füllen, dass keine Ziffer zweimal in einer Reihe, einer Spalte oder in einem Block aus 3 × 3 Feldern erscheint (siehe ▶ Abbildung 6.4). Eine Reihe, eine Spalte oder ein Block wird als **Einheit** bezeichnet.

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

a

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

b

Abbildung 6.4: Ein Sudoku-Rätsel (a) und dessen Lösung (b).

Die in Zeitschriften und Rätselbüchern abgedruckten Sudoku-Rätsel haben die Eigenschaft, dass es genau eine Lösung gibt. Obwohl manche Rätsel schwierig von Hand zu lösen sind und mehrere 10 Minuten in Anspruch nehmen können, benötigt ein CSP-Lösungsprogramm selbst für die schwersten Sudoku-Probleme weniger als 0,1 Sekunde.

Ein Sudoku-Rätsel lässt sich als CSP mit 81 Variablen – eine für jedes Feld – betrachten. Wir verwenden hier die Variablennamen $A1$ bis $A9$ für die oberste Reihe (von links nach rechts) und fahren mit den Buchstaben nach unten fort, sodass die letzte Reihe mit den Variablen $I1$ bis $I9$ bezeichnet wird. Die Domäne der leeren Felder lautet {1, 2, 3, 4, 5, 6, 7, 8, 9}, während die Domäne der bereits ausgefüllten Felder aus einem einzelnen Wert besteht. Außerdem gibt es 27 verschiedene *Alldiff*-Beschränkungen: jeweils eine für jede Reihe, jede Spalte und jeden Block aus 9 Feldern:

Alldiff ($A1$, $A2$, $A3$, $A4$, $A5$, $A6$, $A7$, $A8$, $A9$)
Alldiff ($B1$, $B2$, $B3$, $B4$, $B5$, $B6$, $B7$, $B8$, $B9$)
...
Alldiff ($A1$, $B1$, $C1$, $D1$, $E1$, $F1$, $G1$, $H1$, $I1$)
Alldiff ($A2$, $B2$, $C2$, $D2$, $E2$, $F2$, $G2$, $H2$, $I2$)
...
Alldiff ($A1$, $A2$, $A3$, $B1$, $B2$, $B3$, $C1$, $C2$, $C3$)
Alldiff ($A4$, $A5$, $A6$, $B4$, $B5$, $B6$, $C4$, $C5$, $C6$)
...

Sehen wir uns nun an, wie weit Kantenkonsistenz uns bringt. Wir nehmen an, dass die *Alldiff*-Beschränkungen zu binären Beschränkungen (wie zum Beispiel $A1 \neq A2$) erweitert wurden, sodass wir den AC-3-Algorithmus direkt anwenden können. Sehen Sie sich Variable $E6$ in Abbildung 6.4(a) an – das leere Feld zwischen der 2 und der 8 im mittleren Block. Von den Beschränkungen im Block können wir nicht nur 2 und 8, sondern auch 1 und 7 aus der Domäne von E6 entfernen. Von den Beschränkungen in ihrer Spalte können wir 5, 6, 2, 8, 9 und 3 eliminieren. Dadurch bleibt $E6$ mit der Domäne {4} zurück oder anders ausgedrückt: Wir kennen die Antwort für $E6$. Als Nächstes sehen wir uns die Variable $I6$ an – das Feld im unteren mittleren Block, das von 1, 3 und 3 umgeben ist. Durch Anwenden der Kantenkonsistenz auf die Spalte der Variablen eliminieren wir 5, 6, 2, 4 (da wir inzwischen wissen, dass $E6$ den Wert 4 haben muss), 8, 9 und 3. Durch Kantenkonsistenz mit $I5$ eliminieren wir 1 und in der Domäne von $I6$ bleibt nur noch der Wert 7 zurück. Jetzt gibt es acht bekannte Werte in Spalte 6, sodass sich durch Kantenkonsistenz herleiten lässt, dass $A6$ den Wert 1 haben muss. Die Inferenz wird entsprechend weiter fortgesetzt und schließlich kann AC-3 das gesamte Rätsel lösen – die Domänen aller Variablen ließen sich auf einen einzelnen Wert reduzieren, wie Abbildung 6.4(b) zeigt.

Natürlich würde Sudoku bald seinen Reiz verlieren, wenn sich jedes Rätsel durch eine mechanische Anwendung von AC-3 lösen ließe, und in der Tat funktioniert AC-3 nur für die leichtesten Sudoku-Rätsel. PC-2 ist in der Lage, etwas schwerere Rätsel zu lösen, doch ist dies mit höheren Berechnungskosten verbunden: In einem Sudoku-Rätsel sind 255.960 verschiedene Pfadbeschränkungen zu berücksichtigen. Um die schwersten Rätsel effizient zu lösen, müssen wir intelligenter vorgehen.

In der Tat liegt die Attraktivität von Sudoku-Rätseln für den menschlichen Rätsellöser darin, einfallsreich in der Anwendung komplexerer Inferenzstrategien zu sein. Begeisterte Anhänger geben ihnen wohlklingende Namen wie zum Beispiel „nackte Tripels“. Diese Strategie funktioniert folgendermaßen: Suche in einer beliebigen Einheit (Reihe,

Spalte oder Block) drei Felder, deren Domänen dieselben drei Zahlen oder eine Teilmenge dieser Zahlen enthalten. Zum Beispiel könnten die drei Domänen {1, 8}, {3, 8} und {1, 3, 8} sein. Hieraus ist nicht zu erkennen, welches Feld 1, 3 oder 8 enthält, doch wir wissen, dass die drei Zahlen auf die drei Felder verteilt sein müssen. Demzufolge können wir 1, 3 und 8 aus den Domänen jedes anderen Feldes in der Einheit entfernen.

Interessant ist, wie weit wir gehen können, ohne viel Spezifisches für Sudoku feststellen zu müssen. Natürlich müssen wir sagen, dass es 81 Variablen mit Domänen, die die Ziffern 1 bis 9 enthalten, und 27 *Alldiff*-Beschränkungen gibt. Darüber hinaus jedoch gelten alle Strategien – Kantenkonsistenz, Pfadkonsistenz usw. – allgemein für alle CSPs und nicht nur für Sudoku-Probleme. Selbst „nackte Tripels“ ist eigentlich eine Strategie, um Konsistenz von *Alldiff*-Beschränkungen durchzusetzen, und das hat mit Sudoku an sich nichts zu. Hier zeigt sich die Leistungsfähigkeit des CSP-Formalismus: Für jeden neuen Problembereich brauchen wir nur das Problem in Form von Beschränkungen zu definieren; dann kann der allgemeine Mechanismus zum Auflösen der Beschränkungen eingreifen.

6.3 Backtracking-Suche für CSPs

Sudoku-Probleme sind dafür ausgelegt, durch Inferenz über Randbedingungen gelöst zu werden. Viele andere CSPs lassen sich jedoch durch Inferenz allein nicht lösen; irgendwann müssen wir nach einer Lösung suchen. Dieser Abschnitt beschäftigt sich mit Backtracking-Suchalgorithmen, die auf partiellen Zuweisungen arbeiten; im nächsten Abschnitt sehen wir uns dann lokale Suchalgorithmen über vollständigen Zuweisungen an.

Wir könnten eine Standardsuche mit Tiefenbeschränkung (aus *Kapitel 3*) anwenden. Ein Zustand wäre eine partielle Zuweisung und eine Aktion wäre das Addieren von *var* = *value* zur Zuweisung. Doch für ein CSP mit n Variablen der Domänengröße d erkennen wir schnell etwas ganz Schreckliches: Der Verzweigungsfaktor auf der obersten Ebene ist gleich nd, weil jeder der n Variablen jeder von d Werten zugewiesen werden kann. Auf der nächsten Ebene ist der Verzweigungsfaktor gleich $(n - 1)d$ usw. für n Ebenen. Wir erzeugen einen Baum mit $n! \cdot d^n$ Blättern, obwohl es nur d^n mögliche vollständige Zuweisungen gibt!

Unsere scheinbar sinnvolle, aber naive Problemformulierung hat eine wesentliche Eigenschaft ignoriert, die alle CSPs aufweisen: **Kommutativität**. Ein Problem ist kommutativ, wenn die Anwendungsreihenfolge einer vorgegebenen Aktionsmenge keinen Einfluss auf das Ergebnis hat. Das ist für CSPs der Fall, denn wenn wir Variablen Werte zuweisen, erzielen wir dieselbe partielle Zuweisung – unabhängig von der Reihenfolge. Demzufolge brauchen wir nur eine *einzelne* Variable an jedem Knoten im Suchbaum zu betrachten. Im Wurzelknoten eines Suchbaumes für die Einfärbung der Karte von Australien beispielsweise hätten wir die Wahl zwischen *SA* = *rot*, *SA* = *gruen* und *SA* = *blau*, aber wir würden nie zwischen *SA* = *rot* und *WA* = *blau* wählen. Mit dieser Beschränkung ist die Anzahl der Blätter gleich d^n, wie wir hoffen wollen.

Der Begriff **Backtracking-Suche** wird für eine Tiefensuche verwendet, die Werte jeweils für eine Variable auswählt und zurückgeht, wenn einer Variablen keine weiteren erlaubten Werte zugewiesen werden können. ▶ Abbildung 6.5 zeigt den Algorithmus. Er wählt wiederholt eine nicht zugewiesene Variable aus und probiert dann nacheinander alle Werte in der Domäne dieser Variablen, um eine Lösung zu finden. Wird eine Inkonsis-

tenz erkannt, gibt BACKTRACK einen Fehler zurück. Dies veranlasst den vorhergehenden Aufruf, einen anderen Wert zu probieren.

```
function BACKTRACKING-SEARCH(csp) returns eine Lösung oder einen Fehler
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns eine Lösung oder einen Fehler
  if assignment ist vollständig then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value ist konsistent mit assignment then
      füge {var = value} zu assignment hinzu
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        füge inferences zu assignment hinzu
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    entferne {var = value} und inferences aus assignment
  return failure
```

Abbildung 6.5: Ein einfacher Backtracking-Algorithmus für Probleme unter Rand- und Nebenbedingungen (CSPs). Der Algorithmus baut auf der rekursiven Tiefensuche aus *Kapitel 3* auf. Durch Variieren der Funktionen SELECT-UNASSIGNED-VARIABLE und ORDER-DOMAIN-VALUES können wir die im Text beschriebene universelle Heuristik implementieren. Die Funktion INFERENCE lässt sich optional verwenden, um wie gewünscht Kanten-, Pfad- oder k-Konsistenz zu realisieren. Führt eine Werteauswahl zu einem Fehler (den entweder INFERENCE oder BACKTRACK bemerkt), werden Wertzuweisungen (einschließlich der durch INFERENCE vorgenommenen) aus der aktuellen Zuweisung entfernt und es wird ein neuer Wert ausprobiert.

Ein Teil des Suchbaumes für das Australienproblem ist in ▶ Abbildung 6.6 gezeigt, wo wir Variablen in der Reihenfolge *WA*, *NT*, *Q*, ... zugewiesen haben. Da die Darstellung von CSPs standardisiert ist, ist es nicht erforderlich, BACKTRACKING-SEARCH einen domänenspezifischen Ausgangszustand, eine Aktionsfunktion, ein Übergangsmodell oder einen Zieltest bereitzustellen

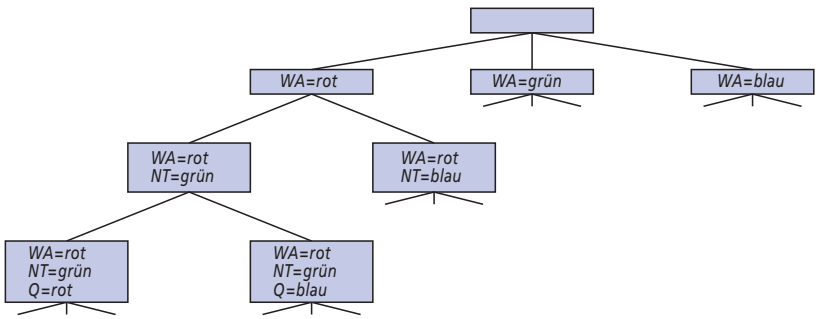


Abbildung 6.6: Teil des Suchbaumes für das Karteneinfärbeproblem in Abbildung 6.1.

BACKTRACKING-SEARCH speichert nur eine einzelne Darstellung eines Zustandes und ändert diese Darstellung, anstatt neue Darstellungen zu erzeugen, wie es in *Abschnitt 3.4.3* beschrieben wird.

In *Kapitel 3* haben wir die schlechte Leistung uninformierter Suchalgorithmen verbessert, indem wir ihnen domänenspezifische Heuristikfunktionen bereitgestellt haben, die wir aus unserem Wissen über das Problem ableiteten. Es stellt sich heraus, dass wir CSPs

effizient *ohne* solches domänenspezifisches Wissen lösen können. Stattdessen können wir die in Abbildung 6.5 nicht spezifizierten Funktionen mit einer gewissen Intelligenz ausstatten, um sie für die Beantwortung folgender Fragen heranzuziehen:

1. Welche Variable sollte als nächste zugewiesen (SELECT-UNASSIGNED-VARIABLE) und in welcher Reihenfolge sollten die Werte ausprobiert (ORDER-DOMAIN-VALUES) werden?
2. Welche Inferenz sollte nach jedem Schritt in der Suche durchgeführt werden (INFERENCE)?
3. Wenn die Suche an einer Zuweisung ankommt, die eine Beschränkung verletzt, kann die Suche eine Wiederholung dieses Fehlers vermeiden?

Die folgenden Unterabschnitte beantworten diese Fragen.

6.3.1 Reihenfolge von Variablen und Werten

Der Backtracking-Algorithmus enthält die Zeile

```
var ← SELECT-UNASSIGNED-VARIABLE(csp)
```

Die einfachste Strategie für SELECT-UNASSIGNED-VARIABLE ist es, die nächste nicht zugewiesene Variable in der Reihenfolge $\{X_1, X_2, ...\}$ auszuwählen. Diese statische Variablenreihenfolge ergibt selten die effizienteste Suche. Beispielsweise gibt es nach den Zuweisungen für $WA = rot$ und $NT = gruen$ in Abbildung 6.6 nur noch einen möglichen Wert für SA; deshalb ist es sinnvoll, als Nächstes $SA = blau$ zuzuweisen, anstatt Q einen Wert zuzuweisen. Nachdem SA zugewiesen ist, stehen die Auswahlen für Q, NSW und V sogar zwingend fest. Dieses intuitive Konzept – die Auswahl der Variablen mit den wenigsten „erlaubten“ Werten – wird auch als **MRV-Heuristik** (Minimum Remaining Values, Minimum an verbleibenden Werten) bezeichnet. Es wurde auch als die „am meisten beschränkte Variable“- oder „Erster-Fehler“-Heuristik bezeichnet. Der letztgenannte Name kommt daher, weil die MRV-Heuristik eine Variable auswählt, die am wahrscheinlichsten bald einen Fehler erzeugt, wodurch der Suchbaum gekürzt wird. Gibt es eine Variable X, für die keine weiteren erlaubten Werte vorhanden sind, wählt die MRV-Heuristik X aus und der Fehler wird sofort erkannt – das vermeidet sinnloses Durchsuchen von anderen Variablen. Die MRV-Heuristik arbeitet normalerweise besser als mit einer zufälligen oder statischen Reihenfolge, manchmal um einen Faktor von 1.000 oder mehr, auch wenn die Ergebnisse je nach Problem breit gefächert sind.

Die MRV-Heuristik hilft uns überhaupt nicht dabei, welches Gebiet in Australien als Erstes eingefärbt werden soll, weil es zunächst für jedes Gebiet drei erlaubte Farben gibt. In diesem Fall ist die **Gradheuristik** praktisch. Sie versucht, den Verzweigungsgrad zukünftiger Auswahlen zu reduzieren, indem sie die Variable auswählt, die in den meisten Beschränkungen für andere, nicht zugewiesene Variablen enthalten ist. In Abbildung 6.1 ist SA die Variable mit dem höchsten Grad, 5; die anderen Variablen haben den Grad 2 oder 3, außer T mit dem Grad 0. Nachdem SA ausgewählt wurde, löst die Anwendung der Gradheuristik das Problem ohne jeden falschen Schritt – Sie können zu jedem Auswahlpunkt *jede* konsistente Farbe auswählen und gelangen dennoch ohne Backtracking zu einer Lösung. Die MRV-Heuristik ist im Allgemeinen eine leistungsfähigere Hilfe, aber die Gradheuristik kann praktisch sein, um zwischen „gleichwertigen“ Alternativen zu entscheiden.

Nachdem eine Variable ausgewählt wurde, muss der Algorithmus die Reihenfolge festlegen, in der er ihre Werte betrachtet. Dazu kann in einigen Fällen eine **Heuristik des am wenigsten einschränkenden Wertes** (Least-Constraining-Value) effektiv sein. Sie bevorzugt den Wert, der die wenigsten Auswahlen für die benachbarten Variablen im Constraint-Graphen ausschließt. Angenommen, in Abbildung 6.1 haben wir die partielle Zuweisung mit $WA = rot$ und $NT = gruen$ vorgenommen und unsere nächste Auswahl erfolgt für Q. Blau wäre eine schlechte Wahl, weil sie den letzten erlaubten Wert für den Nachbarn von Q eliminiert, SA. Die Heuristik des am wenigsten einschränkenden Wertes zieht deshalb Rot Blau vor. Im Allgemeinen versucht die Heuristik, die höchste Flexibilität für nachfolgende Variablenzuweisungen beizubehalten. Wenn wir versuchen, alle Lösungen für ein Problem zu finden und nicht nur die erste, dann spielt die Reihenfolge natürlich keine Rolle, weil wir ohnehin jeden Wert betrachten müssen. Dasselbe gilt, wenn es für das Problem keine Lösungen gibt.

Warum sollte die Variablenauswahl zuerst, die Werteauswahl dagegen zuletzt scheitern? Für ein breites Problemspektrum ist festzustellen, dass eine Variablenreihenfolge, bei der eine Variable mit der minimalen Anzahl von verbleibenden Werten ausgewählt wird, dabei hilft, die Anzahl der Knoten im Suchbaum zu minimieren, indem größere Teile des Baumes früher gekürzt werden. Für die Reihenfolge der Werte besteht der Trick darin, dass wir nur eine Lösung brauchen; deshalb ist es sinnvoll, zuerst nach den wahrscheinlichsten Werten zu suchen. Wenn wir alle Lösungen aufzählen und nicht nur eine finden wollen, ist die Wertereihenfolge irrelevant.

6.3.2 Suche und Inferenz verknüpfen

Bisher haben wir gezeigt, wie AC-3 und andere Algorithmen Reduzierungen in der Domäne der Variablen erschließen können, *bevor* die Suche beginnt. Inferenz kann aber im Verlauf einer Suche noch leistungsfähiger sein: Immer wenn wir einen Wert für eine Variable auswählen, haben wir eine gänzlich neue Gelegenheit, auf neue Domänenreduzierungen auf den benachbarten Variablen zu schließen.

Eine der einfachsten Formen der Inferenz ist die sogenannte **Vorabüberprüfung**. Immer wenn eine Variable X zugewiesen wird, richtet der Prozess der Vorabüberprüfung eine Kantenkonsistenz für sie ein: Für jede nicht zugewiesene Variable Y, die durch eine Beschränkung mit X verbunden ist, wird jeder Wert aus der Domäne von Y gelöscht, der mit dem für X gewählten Wert inkonsistent ist. Da eine Vorabüberprüfung nur in Bezug auf Kantenkonsistenz schlussfolgert, gibt es keinen Grund, eine Vorabüberprüfung auszuführen, wenn in einem vorhergehenden Schritt bereits eine Kantenkonsistenz realisiert wurde.

▶ Abbildung 6.7 zeigt den Ablauf der Backtracking-Suche für das Australien-CSP mit Vorabüberprüfung. Für dieses Beispiel sind zwei wichtige Aspekte zu berücksichtigen. Beachten Sie zunächst, dass nach der Zuweisung von $WA = rot$ und $Q = gruen$ die Domänen von NT und SA auf einen einzigen Wert reduziert wurden; wir haben die von diesen Variablen abhängige Verzweigung vollständig eliminiert, indem wir Informationen von WA und Q weitergegeben haben. Ein zweiter wichtiger Punkt ist, dass nach Zuweisung von $V = blau$ die Domäne von SA leer ist. Damit hat die Vorabüberprüfung erkannt, dass die partielle Zuweisung $\{WA = rot, Q = gruen, V = blau\}$ mit den Beschränkungen des Problems inkonsistent ist, und der Algorithmus geht deshalb sofort zurück.

Für viele Probleme ist die Suche effizienter, wenn die MRV-Heuristik mit Vorabüberprüfung kombiniert wird. Sehen Sie sich Abbildung 6.7 nach der Zuweisung von $\{WA = rot\}$ an. Intuitiv scheint es, dass diese Zuweisung ihre Nachbarn *NT* und *SA* beschränkt, sodass wir diese Variablen als Nächstes behandeln könnten und sich dann alle anderen Variablen automatisch ergeben. Genau dies passiert mit MRV: *NT* und *SA* haben zwei Werte; also wird einer von ihnen zuerst ausgewählt, dann der andere, dann nacheinander *Q*, *NSW* und *V*. Schließlich umfasst *T* immer noch drei Werte und jeder davon funktioniert. Vorabüberprüfung lässt sich als effizienter Weg ansehen, um die Informationen, die die MRV-Heuristik für ihre Aufgabe benötigt, inkrementell zu berechnen.

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Anfängliche Domänen	R G B	R G B	R G B	R G B	R G B	R G B	R G B
Nach *WA*=*rot*	(R)	G B	R G B	R G B	R G B	G B	R G B
Nach *Q*=*gruen*	(R)	B	(G)	R B	R G B	B	R G B
Nach *V*=*blau*	(R)	B	(G)	R	(B)		R G B

Abbildung 6.7: Fortschreitende Suche für die Einfärbung einer Landkarte mit Vorabüberprüfung. Als Erstes wird $WA = rot$ zugewiesen; die Vorabüberprüfung entfernt dann *rot* aus den Domänen der benachbarten Variablen *NT* und *SA*. Nachdem $Q = gruen$ zugewiesen wurde, wird *gruen* aus den Domänen von *NT*, *SA* und *NSW* entfernt. Nachdem $V = blau$ zugewiesen wurde, wird *blau* aus den Domänen von *NSW* und *SA* entfernt, sodass *SA* keine erlaubten Werte mehr besitzt.

Die Vorabüberprüfung erkennt viele Inkonsistenzen, aber nicht alle. Das Problem besteht darin, dass sie die aktuelle Variable kantenkonsistent macht, aber nicht vorausschauend arbeitet und auch alle anderen Variablen kantenkonsistent macht. Betrachten Sie beispielsweise die dritte Zeile von Abbildung 6.7. Sie zeigt, dass, wenn *WA* gleich *rot* und *Q* gleich *gruen* ist, *NT* und *SA* zwingend *blau* sein müssen. Sie sind jedoch benachbart und dürfen deshalb nicht denselben Wert haben. Die Vorabüberprüfung erkennt dies nicht als Inkonsistenz, weil sie nicht weit genug in die Zukunft sieht.

Der als **MAC** (für **Maintaining Arc Consistency**, Aufrechterhaltung der Kantenkonsistenz) bezeichnete Algorithmus erkennt diese Inkonsistenz. Nachdem einer Variablen X_i ein Wert zugewiesen wurde, ruft die Prozedur INFERENCE den Algorithmus AC-3 auf, doch statt einer Warteschlange aller Kanten im CSP beginnen wir nur mit den Kanten (X_j, X_i) für alle nicht zugewiesenen Variablen X_j, die Nachbarn von X_i sind. Von hier aus führt AC-3 die Beschränkungsweiterleitung in der üblichen Weise aus. Sobald die Domäne irgendeiner Variablen auf die leere Menge geschrumpft ist, scheitert der Aufruf von AC-3. Dann wissen wir, dass wir unverzüglich zurückgehen müssen. Es ist zu erkennen, dass MAC unbedingt leistungsfähiger ist als Vorabüberprüfung, da Vorabüberprüfung das Gleiche wie MAC für die anfänglichen Kanten in der Warteschlange von MAC ausführt; doch im Gegensatz zu MAC leitet die Vorabüberprüfung die Beschränkungen nicht rekursiv weiter, wenn die Domänen der Variablen geändert wurden.

6.3.3 Intelligentes Backtracking: rückwärts schauen

Der Algorithmus BACKTRACKING-SEARCH in Abbildung 6.5 hat eine sehr einfache Strategie, was zu tun ist, wenn eine Verzweigung der Suche fehlschlägt: Er geht zurück zur vorhergehenden Variablen und probiert es mit einem anderen Wert für sie. Man spricht auch vom **chronologischen Backtracking**, weil der *zuletzt verwendete* Ent-

scheidungspunkt erneut betrachtet wird. In diesem Unterabschnitt zeigen wir, dass es sehr viel bessere Vorgehensweisen gibt.

Überlegen Sie, was passiert, wenn wir in Abbildung 6.1 ein einfaches Backtracking mit einer festen Variablenreihenfolge *Q*, *NSW*, *V*, *T*, *SA*, *WA*, *NT* anwenden. Angenommen, wir haben die partielle Zuweisung {*Q* = *rot*, *NSW* = *gruen*, *V* = *blau*, *T* = *rot*} erzeugt. Wenn wir die nächste Variable (*SA*) probieren, erkennen wir, dass jeder Wert eine Beschränkung verletzt. Wir gehen zurück zu *T* und probieren es mit einer neuen Farbe für Tasmanien! Offensichtlich ist das Unsinn – eine Umfärbung von Tasmanien kann das Problem mit Südaustralien nicht lösen.

Ein intelligenterer Ansatz für das Backtracking ist, zu einer Variablen zurückzugehen, die das Problem lösen könnte – zu einer Variablen, die dafür verantwortlich war, dass einer der möglichen Werte von *SA* unmöglich gemacht wurde. Dazu verfolgen wir eine Menge von Zuweisungen, die mit einem bestimmten Wert für *SA* in Konflikt stehen. Die Menge (in diesem Fall {*Q* = *rot*, *NSW* = *gruen*, *V* = *blau*}) wird als **Konfliktmenge** für *SA* bezeichnet. Die Methode **Backjumping** („Zurückspringen“) führt ein Backtracking zu der *zuletzt verwendeten* Zuweisung in der Konfliktmenge durch; in diesem Fall würde das Backjumping Tasmanien überspringen und es mit einem neuen Wert für *V* probieren. Dieses Vorgehen lässt sich einfach implementieren, indem man BACKTRACK so modifiziert, dass der Algorithmus die Konfliktmenge akkumuliert, während er nach einem erlaubten Wert für die Zuweisung sucht. Wird kein erlaubter Wert gefunden, sollte der Algorithmus das zuletzt verwendete Element der Konfliktmenge sowie den Fehlerhinweis zurückgeben.

Der aufmerksame Leser wird bemerkt haben, dass Vorabüberprüfung die Konfliktmenge ohne zusätzlichen Aufwand bereitstellen kann: Immer wenn die Vorabüberprüfung basierend auf einer Zuweisung von $X = x$ einen Wert aus der Domäne von Y löscht, sollte sie $X = x$ der Konfliktmenge von Y hinzufügen. Wenn der letzte Wert aus der Domäne von Y entfernt wird, werden die Zuweisungen in der Konfliktmenge von Y der Konfliktmenge von X hinzugefügt. Wenn wir dann zu Y gelangen, wissen wir sofort, wohin wir im Bedarfsfall zurückspringen müssen.

Zudem wird der aufmerksame Leser etwas Seltsames bemerkt haben: Das Backjumping tritt auf, wenn jeder Wert in einer Domäne in Konflikt mit der aktuellen Zuweisung steht; jedoch erkennt die Vorabüberprüfung dieses Ereignis und verhindert, dass die Suche je einen solchen Knoten erreicht! Tatsächlich kann gezeigt werden, dass *jede* Verzweigung, die durch das Backjumping gekürzt wird, auch durch die Vorabüberprüfung gekürzt wird. Damit ist ein einfaches Backjumping in einer Suche mit Vorabüberprüfung und tatsächlich auch in einer Suche, die eine strengere Konsistenzüberprüfung verwendet (wie etwa MAC), redundant.

Trotz der Beobachtungen aus dem vorigen Abschnitt bleibt der Gedanke hinter dem Backjumping sinnvoll: abhängig von den Gründen für den Fehler zurückzuspringen. Das Backjumping erkennt einen Fehler, wenn die Domäne einer Variablen leer wird, aber häufig wird eine Verzweigung verworfen, lange bevor dies passiert. Betrachten Sie erneut die partielle Zuweisung {*WA* = *rot*, *NSW* = *rot*} (die, wie wir aus der vorherigen Diskussion wissen, inkonsistent ist). Angenommen, wir probieren es als Nächstes mit *T* = *rot* und weisen dann *NT*, *Q*, *V*, *SA* zu. Wir wissen, dass für diese vier letzten Variablen keine Zuweisung funktionieren kann; deshalb gehen uns schließlich die Werte aus, die wir für *NT* probieren können. Jetzt ist die Frage, wohin wir zurückspringen sollen. Das Backjumping kann nicht funktionieren, weil es für *NT* Werte *gibt*,

die konsistent mit den zuvor zugewiesenen Variablen sind – *NT* hat keine vollständige Konfliktmenge vorhergehender Variablen, die den Fehler verursacht haben. Wir wissen jedoch, dass die vier Variablen *NT*, *Q*, *V* und *SA insgesamt* bedingt durch eine Menge vorhergehender Variablen zu einem Fehler geführt haben, und das müssen die Variablen sein, die direkt mit den vier Variablen in Konflikt stehen. Das führt zu einer tieferen Bedeutung der Konfliktmenge für eine Variable wie *NT*: Es ist diese Menge vorhergehender Variablen, die verursacht hat, dass *NT zusammen mit allen nachfolgenden Variablen* keine konsistente Lösung hat. In diesem Fall umfasst die Menge *WA* und *NSW*, sodass der Algorithmus zu *NSW* zurückgehen und Tasmanien überspringen sollte. Ein Backjumping-Algorithmus, der auf diese Weise definierte Konfliktmengen verwendet, wird auch als **konfliktgesteuertes Backjumping** bezeichnet.

Jetzt müssen wir erklären, wie diese neuen Konfliktmengen berechnet werden. Die Methode ist wirklich ganz einfach. Der „endgültige" Fehler einer Verzweigung der Suche tritt immer auf, wenn die Domäne einer Variablen leer wird. Diese Variable hat eine Standardkonfliktmenge. In unserem Beispiel schlägt *SA* fehl und ihre Konfliktmenge ist (angenommen) {*WA*, *NT*, *Q*}. Wir springen zu *Q* zurück und *Q absorbiert* die Konfliktmenge von *SA* (natürlich minus *Q* selbst) in seine eigene Konfliktmenge, die {*NT*, *NSW*} ist; die neue Konfliktmenge ist {*WA*, *NT*, *NSW*}. Das bedeutet, es gibt von *Q* an keine Lösung, betrachtet man die vorherige Zuweisung zu {*WA*, *NT*, *NSW*}. Wir gehen also zu *NT* zurück, der zuletzt von diesen betrachteten Variablen. *NT* absorbiert {*WA*, *NT*, *NSW*} – {*NT*} in seine eigene direkte Konfliktmenge {*WA*}, womit wir {*WA*, *NSW*} erhalten (wie im vorigen Absatz beschrieben). Jetzt springt der Algorithmus zurück zu *NSW*, wie wir gehofft haben. Zusammenfassend können wir sagen: Sei X_j die aktuelle Variable und $conf(X_j)$ ihre Konfliktmenge. Wenn jeder mögliche Wert für X_j fehlschlägt, springen wir zurück zur zuletzt verwendeten Variablen X_i in $conf(X_j)$ und setzen:

$$conf(X_i) \leftarrow conf(X_i) \cup conf(X_j) - \{X_i\}.$$

Wenn wir einen Widerspruch erreichen, kann Backjumping uns sagen, wie weit zurückzuspringen ist, sodass wir keine Zeit verschwenden, Variablen zu ändern, die das Problem nicht beseitigen. Doch wir möchten auch vermeiden, immer wieder in das gleiche Problem zu geraten. Wenn die Suche bei einem Widerspruch ankommt, wissen wir, dass eine bestimmte Teilmenge der Konfliktmenge für das Problem verantwortlich ist. Das **Lernen von Beschränkungen** (**Constraint Learning**) ist ein Konzept, eine Minimalmenge von Variablen aus der Konfliktmenge zu ermitteln, die das Problem verursacht. Diese Menge von Variablen wird zusammen mit ihren korrespondierenden Werten als **No-good** (svw. nutzlos, untauglich) bezeichnet. Dann erfassen wir das No-good, indem wir entweder eine neue Beschränkung zum CSP hinzufügen oder einen separaten Cache von No-goods führen.

Betrachten Sie zum Beispiel den Zustand {*WA* = *rot*, *NT* = *gruen*, *Q* = *blau*} in der untersten Reihe von Abbildung 6.6. Durch Vorabüberprüfung erfahren wir, dass dieser Zustand ein No-good ist, weil es keine gültige Zuweisung zu *SA* gibt. Im konkreten Fall würde die Aufzeichnung des No-goods aber nichts bringen, denn nachdem wir diesen Zweig aus dem Suchbaum gekürzt haben, treffen wir niemals mehr auf diese Kombination. Nehmen Sie nun an, dass der Suchbaum in Abbildung 6.6 tatsächlich Teil eines größeren Suchbaumes ist, bei dem zuerst Werte für *V* und *T* zugewiesen wurden. Dann würde es sich lohnen, {*WA* = *rot*, *NT* = *gruen*, *Q* = *blau*} als No-good

aufzuzeichnen, da wir für jede mögliche Menge von Zuweisungen zu V und T wieder zum selben Problem gelangen.

No-goods können effizient durch Vorabüberprüfung oder durch Backjumping genutzt werden. Das Lernen von Beschränkungen ist eine der wichtigsten Techniken, auf die moderne CSP-Lösungsmodule zurückgreifen, um auch komplexe Probleme effizient zu lösen.

6.4 Lokale Suche für Probleme unter Rand- und Nebenbedingungen

Lokale Suchalgorithmen (siehe *Abschnitt 4.1*) erweisen sich als sehr effektiv für die Lösung vieler CSPs. Sie verwenden eine vollständige Zustandsformulierung: Der Ausgangszustand weist jeder Variablen einen Wert zu und die Suche ändert den Wert von jeweils einer Variablen. Zum Beispiel könnte im 8-Damen-Problem (siehe Abbildung 4.3) der Ausgangszustand eine zufällige Konfiguration von acht Damen in acht Spalten sein und jeder Schritt verschiebt eine einzelne Dame an eine neue Position in ihrer Spalte. Typischerweise verletzt die anfängliche Annahme mehrere Beschränkungen. Der Kern der lokalen Suche besteht darin, die verletzten Beschränkungen zu eliminieren.[2]

Bei der Auswahl eines neuen Wertes für eine Variable ist die offensichtlichste Heuristik, den Wert auszuwählen, der die wenigsten Konflikte mit anderen Variablen verursacht – die **Min-Conflicts**-Heuristik. Der Algorithmus ist in ▶ Abbildung 6.8 angegeben und seine Anwendung auf ein 8-Damen-Problem ist in ▶ Abbildung 6.9 zu sehen.

```
function MIN-CONFLICTS(csp, max_steps) returns eine Lösung oder einen Fehler
  inputs: csp, ein CSP
    max_steps, Anzahl der Schritte, die erlaubt sind, bevor aufgegeben wird

  current ← eine anfangs vollständige Zuweisung für csp
  for i = 1 to max_steps do
    if current ist eine Lösung für csp then return current
    var ← eine zufällig aus csp.VARIABLES gewählte konfliktverursachende Variable
    value ← der Wert v für var, der CONFLICTS(var, v, current, csp) minimiert
    setze var = value in current
  return failure
```

Abbildung 6.8: Der MIN-CONFLICTS-Algorithmus für die Lösung von CSPs durch lokale Suche. Der Ausgangszustand kann zufällig gewählt werden oder durch einen „gierigen" Zuweisungsprozess, der einen minimal konfliktverursachenden Wert für jede Variable auswählt. Die Funktion CONFLICTS zählt die Anzahl der Beschränkungen, die ein bestimmter Wert für die restliche aktuelle Zuweisung verletzt.

2 Die lokale Suche lässt sich leicht auf Optimierungsprobleme unter Rand- und Nebenbedingungen (Constraint Optimization Problems, COPs) erweitern. In diesem Fall können alle Techniken für Hill Climbing und Simulated Annealing angewendet werden, um die Zielfunktion zu optimieren.

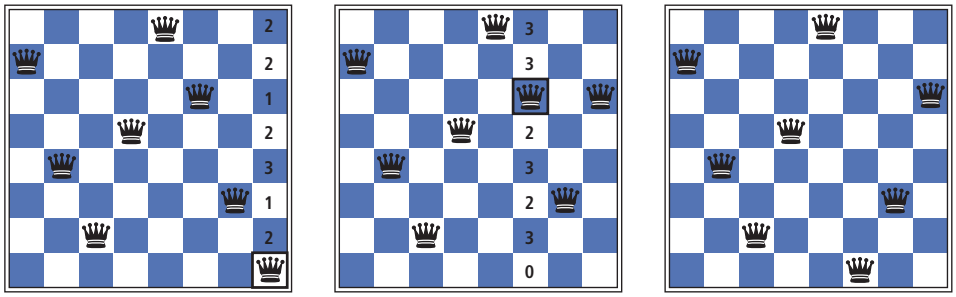


Abbildung 6.9: Eine Lösung in zwei Schritten für ein 8-Damen-Problem unter Verwendung von MIN-CONFLICTS. In jeder Phase wird eine Dame ausgewählt, um sie in ihrer Spalte neu anzuordnen. In den verschiedenen Quadraten wird jeweils die Anzahl der Konflikte (in diesem Fall die Anzahl der angreifenden Damen) gezeigt. Der Algorithmus verschiebt die Dame auf das Quadrat mit den wenigsten Konflikten, wobei zufällig zwischen gleichwertigen Quadraten entschieden wird.

MIN-CONFLICTS ist für viele CSPs überraschend effektiv. Im n-Damen-Problem ist es erstaunlich, dass ohne Berücksichtigung der anfänglichen Platzierung der Damen die Laufzeit von MIN-CONFLICTS so gut wie *unabhängig von der Problemgröße* ist. Der Algorithmus löst sogar das *Millionen*-Damen-Problem in durchschnittlich 50 Schritten (nach der anfänglichen Zuweisung). Diese bemerkenswerte Beobachtung war der Anstoß, der in den 1990er Jahren zu zahlreichen Forschungen zur lokalen Suche und zur Unterscheidung zwischen einfachen und schwierigen Problemen geführt hat, die wir in *Kapitel 7* genauer betrachten. Man könnte sagen, n-Damen ist einfach für die lokale Suche, weil die Lösungen im Zustandsraum dicht verteilt sind. MIN-CONFLICTS funktioniert auch für schwierige Probleme gut. Beispielsweise wurde es verwendet, um die Beobachtungen für das Weltraumteleskop Hubble einzuplanen, wodurch die Zeit zur Planung von Beobachtungen für eine Woche von drei Wochen (!) auf etwa zehn Minuten reduziert wurde.

Alle lokalen Suchverfahren aus *Abschnitt 4.1* kommen für die Anwendung auf CSPs infrage und einige davon haben sich als besonders effektiv erwiesen. Die Landschaft eines CSP unter der MinConflicts-Heuristik weist eine Reihe von Plateaus auf. Es kann Millionen von Variablenzuweisungen geben, die lediglich einen Konflikt weit von einer Lösung entfernt sind. Plateau-Suche – die Seitwärtsbewegungen zu einem anderen Zustand mit der gleichen Punktebewertung erlaubt – kann hilfreich sein, damit die lokale Suche einen Ausweg von diesem Plateau findet. Dieses Wandern auf dem Plateau lässt sich mit **Tabu-Suche** steuern: Es wird eine kleine Liste der zuletzt besuchten Zustände geführt und dem Algorithmus verboten, zu diesen Zuständen zurückzukehren. Um von Plateaus zu entkommen, lässt sich auch Simulated Annealing einsetzen.

Durch eine andere Technik, die sogenannte **Beschränkungsgewichtung** (**Constraint Weighting**), lässt sich die Suche auf die wichtigen Beschränkungen konzentrieren. Dabei wird jeder Beschränkung ein numerisches Gewicht W_i zugeordnet, wobei anfangs alle Gewichte den Wert 1 erhalten. Bei jedem Suchschritt wählt der Algorithmus ein zu änderndes Variablen-/Werte-Paar, das im niedrigsten Gesamtgewicht aller verletzten Beschränkungen resultiert. Die Gewichte werden dann angepasst, indem man das Gewicht jeder Beschränkung, die durch die aktuelle Zuweisung verletzt wird, inkrementiert. Daraus ergeben sich zwei Vorteile: Die Plateaus erhalten eine Topografie, die es ermöglicht, eine Verbesserung vom aktuellen Zustand aus zu erreichen, und die Gewichte der Beschränkungen, die sich als schwer lösbar erweisen, erhöhen sich mit der Zeit.

Ein weiterer Vorteil der lokalen Suche ist, dass sie in einer Online-Einstellung verwendet werden kann, wenn sich das Problem ändert. Das ist insbesondere bei Planungsaufgaben sehr wichtig. Der Flugplan für eine Woche kann Tausende von Flügen und Zehntausende von Personalzuordnungen beinhalten, aber schlechtes Wetter auf einem Flughafen kann den gesamten Plan zunichte machen. Wir wollen den Plan mit einem Minimum an Änderungen korrigieren. Das ist ganz einfach mit einem lokalen Suchalgorithmus, der von dem aktuellen Plan ausgeht. Eine Backtracking-Suche mit der neuen Menge an Beschränkungen ist in der Regel sehr viel zeitaufwändiger und könnte eine Lösung mit vielen Änderungen gegenüber dem aktuellen Plan finden.

6.5 Die Struktur von Problemen

In diesem Abschnitt betrachten wir Methoden, wie die *Struktur* des Problems, so wie sie durch den Constraint-Graphen repräsentiert wird, genutzt werden kann, um schnell Lösungen zu finden. Die meisten der hier gezeigten Ansätze sind sehr allgemein und auch auf andere Probleme als CSPs anzuwenden, wie etwa probabilistisches Schließen. Immerhin ist die einzige Möglichkeit, auf die wir hoffen können, mit der realen Welt zurechtzukommen, ihre Zerlegung in viele Unterprobleme. Beim Constraint-Graphen für Australien (Abbildung 6.1(b), erneut in ▶ Abbildung 6.12(a) wiedergegeben) fällt eine Tatsache auf: Tasmanien ist nicht mit dem Festland verbunden.[3] Intuitiv ist offensichtlich, dass die Einfärbung von Tasmanien und die Einfärbung des Festlandes voneinander **unabhängige Unterprobleme** sind – eine beliebige Lösung für das Festland kombiniert mit einer beliebigen Lösung für Tasmanien ergibt eine Lösung für die gesamte Karte. Unabhängigkeit kann einfach ermittelt werden, indem man nach **verbundenen Komponenten** des Constraint-Graphen sucht. Jede Komponente entspricht einem Unterproblem CSP_i. Wenn die Zuweisung S_i eine Lösung von CSP_i ist, dann ist $\cup_i S_i$ eine Lösung von $\cup_i CSP_i$. Warum ist das wichtig? Betrachten Sie Folgendes: Angenommen, jedes CSP_i hat c Variablen aus insgesamt n Variablen, wobei c eine Konstante ist. Dann gibt es n/c Unterprobleme, die jeweils höchstens d^c Arbeit für ihre Lösung bedingen. Die Gesamtarbeit beträgt also $O(d^c n/c)$, was *linear* in n ist; ohne die Zerlegung ist die Gesamtarbeit gleich $O(d^n)$, was *exponentiell* in n ist. Wollen wir es konkreter ausdrücken: Unterteilt man ein boolesches CSP mit 80 Variablen in vier Unterprobleme, reduziert man die Lösungszeit für den ungünstigsten Fall von der Lebensdauer des Universums auf weniger als eine Sekunde.

Tipp

Völlig unabhängige Unterprobleme sind wunderbar – aber selten. Erfreulicherweise lassen sich einige andere Graph-Strukturen ebenfalls einfach lösen. Zum Beispiel ist ein Constraint-Graph ein **Baum**, wenn zwei beliebige Variablen durch höchstens einen Pfad miteinander verbunden sind. Wir zeigen, *dass jedes baumförmig strukturierte CSP in einer Zeit gelöst werden kann, die linear zur Anzahl der Variablen ist.*[4] Der Schlüssel dazu ist ein neues Konsistenzkonzept, die sogenannte **gerichtete Kantenkonsistenz** (Directed Arc Consistency, DAC). Als gerichtet kantenkonsistent definiert ist ein CSP unter einer Reihenfolge der Variablen $X_1, X_2, ..., X_n$ genau dann, wenn jede X_i kantenkonsistent mit jeder X_j für $j > i$ ist.

3 Ein sorgfältiger Kartograph oder ein tasmanischer Patriot kann argumentieren, dass Tasmanien nicht in derselben Farbe wie sein nächstgelegener Nachbar auf dem Festland eingefärbt werden sollte, um den Eindruck zu vermeiden, dass es Teil dieses Bundesstaates sein *könnte*.

4 Leider haben nur sehr wenige Regionen der Erde, mit der möglichen Ausnahme von Sulawesi, baumförmige Landkarten.

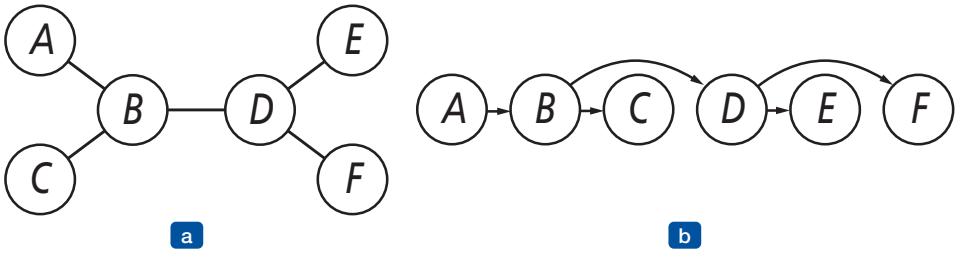


Abbildung 6.10: (a) Der Constraint-Graph des baumstrukturierten CSP. (b) Eine lineare Reihenfolge der Variablen, die konsistent mit dem Baum mit A als Wurzel ist. Dies wird als **topologische Sortierung** der Variablen bezeichnet.

Um ein baumstrukturiertes CSP zu lösen, legt man zuerst eine Variable als Wurzel des Baumes fest und wählt dann eine Reihenfolge der Variablen, sodass jede Variable nach ihrem übergeordneten Element im Baum erscheint. Eine derartige Reihenfolge wird **topologische Sortierung** genannt. ▶ Abbildung 6.10(a) zeigt einen Beispielbaum und (b) eine mögliche Reihenfolge. Jeder Baum mit n Knoten besitzt $n-1$ Kanten, sodass wir diesen Graphen in $O(n)$ Schritten gerichtet kantenorientiert machen können, wobei jeder Schritt bis zu d mögliche Domänenwerte für zwei Variablen vergleichen muss, was eine Gesamtzeit von $O(nd^2)$ ergibt. Nachdem wir über einen gerichteten kantenorientierten Graphen verfügen, können wir einfach die Liste der Variablen nach unten hin durchgehen und einen beliebigen verbleibenden Wert auswählen. Da jede Verknüpfung von einem übergeordneten zu seinem untergeordneten Element kantenkonsistent ist, wissen wir, dass für jeden Wert, den wir für das übergeordnete Element auswählen, ein gültiger auszuwählender Wert für das untergeordnete Element bleibt. Das bedeutet, dass wir nicht zurückspringen müssen, sondern die Variablen linear durchlaufen können. ▶ Abbildung 6.11 zeigt den vollständigen Algorithmus.

```
function TREE-CSP-SOLVER(csp) returns eine Lösung oder einen Fehler
  inputs: csp, ein CSP mit den Komponenten X, D, C

  n ← Anzahl der Variablen in X
  assignment ← eine leere Zuweisung
  root ← eine beliebige Variable in X
  X ← TOPOLOGICALSORT(X, root)
  for j = n down to 2 do
    MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
    if lässt sich nicht konsistent machen then return failure
  for i = 1 to n do
    assignment[X_i] ← beliebiger konsistenter Wert von D_i
    if kein konsistenter Wert vorhanden then return failure
  return assignment
```

Abbildung 6.11: Der TREE-CSP-SOLVER-Algorithmus zum Lösen von baumstrukturierten CSPs. Wenn das CSP eine Lösung besitzt, finden wir sie in linearer Zeit, andernfalls entdecken wir einen Widerspruch.

Nachdem wir nun einen effizienten Algorithmus für Bäume haben, können wir überlegen, ob allgemeinere Constraint-Graphen irgendwie auf Bäume *reduziert* werden können. Es gibt zwei primäre Möglichkeiten dafür, wobei die eine auf dem Löschen von Knoten basiert, die andere auf der Zusammenlegung von Knoten.

Der erste Ansatz weist einigen Variablen Werte zu, sodass die restlichen Variablen einen Baum bilden. Betrachten Sie den Constraint-Graphen für Australien, der in Abbildung 6.12(a) noch einmal gezeigt ist. Könnten wir Südaustralien löschen, würde der Graph zu einem Baum, wie in (b) gezeigt. Glücklicherweise ist das möglich (im Graph, nicht auf dem Kontinent!), indem wir einen Wert für *SA* festlegen und aus den Domänen der anderen Variablen alle Werte entfernen, die inkonsistent mit dem für *SA* gewählten Wert sind.

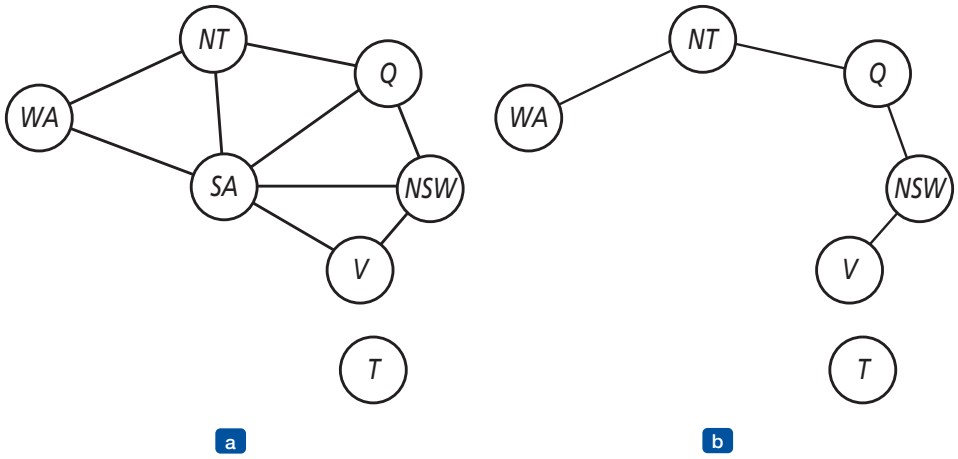


Abbildung 6.12: (a) Der ursprüngliche Constraint-Graph aus Abbildung 6.1. (b) Der Constraint-Graph nach Entfernen von *SA*.

Damit sind alle Lösungen für das CSP, nachdem *SA* und seine Randbedingungen entfernt wurden, mit dem für *SA* gewählten Wert konsistent. (Das funktioniert für binäre CSPs; bei Randbedingungen höherer Ordnung ist die Situation komplexer.) Jetzt können wir den verbleibenden Baum mit dem oben gezeigten Algorithmus lösen und damit das gesamte Problem. Im allgemeinen Fall (im Gegensatz zur Karteneinfärbung) könnte der für *SA* gewählte Wert natürlich falsch sein, sodass wir jeden möglichen Wert ausprobieren müssten. Der allgemeine Algorithmus sieht wie folgt aus:

1. Wähle eine Untermenge S aus den Variablen des CSP, sodass der Constraint-Graph zu einem Baum wird, nachdem S entfernt wurde. S wird auch als **zyklische Schnittmenge** bezeichnet.

2. Für alle möglichen Zuweisungen an die Variablen in S, die alle Randbedingungen für S erfüllen,

   entferne aus den Domänen der restlichen Variablen alle Werte, die inkonsistent mit der Zuweisung von S sind, und

   wenn es für das verbleibende CSP eine Lösung gibt, gib sie zusammen mit der Zuweisung für S zurück.

Wenn die zyklische Schnittmenge die Größe c hat, dann ist die Gesamtlaufzeit gleich $O(d^c \cdot (n - c)d^2)$: Wir müssen jede der d^c Kombinationen von Werten für die Variablen in S ausprobieren und für jede Kombination müssen wir ein Baumproblem der Größe $n - c$ lösen. Ist der Graph „fast ein Baum“, wird c klein sein und unsere Einsparungen gegenüber dem einfachen Backtracking sind enorm. Im ungünstigsten Fall kann c jedoch $(n - 2)$ groß sein. Die Ermittlung der *kleinsten* zyklischen Schnittmenge ist NP-hart,

aber es gibt mehrere effiziente Annäherungsalgorithmen für diese Aufgabe. Der gesamte algorithmische Ansatz wird auch als **Schnittmengenkonditionierung** bezeichnet; wir werden ihr in *Kapitel 14* wieder begegnen, wo sie für das Schließen im Hinblick auf Wahrscheinlichkeiten verwendet wird.

Der zweite Ansatz basiert auf der Konstruktion einer **Baumzerlegung** des Constraint-Graphen in eine Menge verbundener Unterprobleme. Jedes Unterproblem wird unabhängig gelöst und die resultierenden Lösungen werden anschließend kombiniert. Wie die meisten Teilen&Herrschen-Algorithmen funktioniert das gut, wenn keines der Unterprobleme zu groß ist. ▶ Abbildung 6.13 zeigt eine Baumzerlegung des Karteneinfärbeproblems in fünf Unterprobleme. Eine Baumzerlegung muss die folgenden drei Bedingungen erfüllen:

- Jede Variable im ursprünglichen Problem erscheint in mindestens einem der Unterprobleme.
- Sind zwei Variablen im ursprünglichen Problem durch eine Randbedingung verknüpft, müssen sie zusammen (und mit der Randbedingung) in mindestens einem der Unterprobleme erscheinen.
- Wenn eine Variable in zwei Unterproblemen im Baum erscheint, muss sie in jedem Unterproblem entlang des Pfades erscheinen, der diese Unterprobleme verbindet.

Die beiden ersten Bedingungen stellen sicher, dass alle Variablen und Randbedingungen in der Zerlegung repräsentiert werden. Die dritte Bedingung erscheint recht technisch, reflektiert aber einfach die Randbedingung, dass jede beliebige Variable in jedem Unterproblem, in dem sie auftritt, denselben Wert haben muss; diese Randbedingung wird durch die Verbindungen erzwungen, die die Unterprobleme im Baum verknüpfen. Beispielsweise erscheint *SA* in allen vier verbundenen Unterproblemen wie in Abbildung 6.13 gezeigt. In Abbildung 6.12 können Sie sich davon überzeugen, dass diese Zerlegung sinnvoll ist.

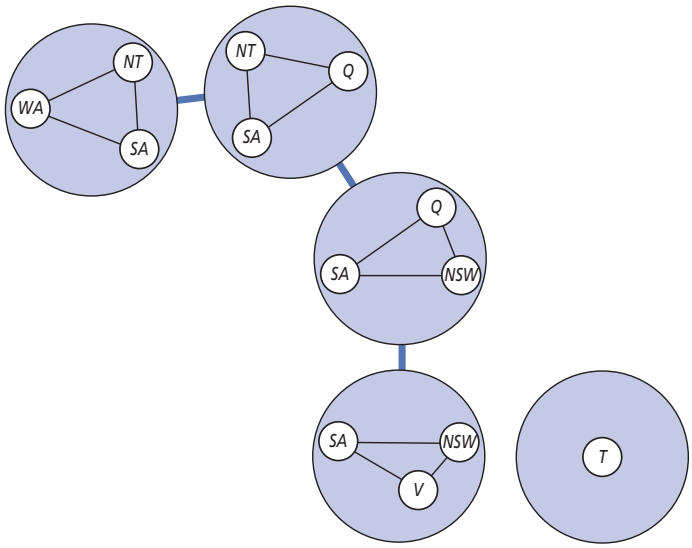


Abbildung 6.13: Eine Baumzerlegung für den Constraint-Graphen in Abbildung 6.12(a).

Wir lösen die einzelnen Unterprobleme unabhängig voneinander. Gibt es für eines davon keine Lösung, wissen wir, dass es für das gesamte Problem keine Lösung gibt. Können wir alle Unterprobleme lösen, versuchen wir, wie folgt eine globale Lösung zu konstruieren. Als Erstes betrachten wir jedes Unterproblem als „Megavariable“, deren Domäne die Menge aller Lösungen für das Unterproblem ist. Beispielsweise ist das ganz linke Unterproblem in Abbildung 6.13 ein Karteneinfärbungsproblem mit drei Variablen. Es hat damit sechs Lösungen – eine davon ist $\{WA = rot, SA = blau, NT = gruen\}$. Anschließend lösen wir die Randbedingungen, die die Unterprobleme verbinden. Dazu verwenden wir den oben gezeigten effizienten Algorithmus für Bäume. Die Randbedingungen zwischen Unterproblemen erzwingen einfach, dass die Lösungen der Unterprobleme dieselben Werte für ihre Variablen verwenden. Hat man beispielsweise die Lösung $\{WA = rot, SA = blau, NT = gruen\}$ für das erste Unterproblem, ist die einzige konsistente Lösung für das nächste Unterproblem gleich $\{SA = blau, NT = gruen, Q = rot\}$.

Tipp

Ein Constraint-Graph erlaubt viele Baumzerlegungen; bei der Auswahl einer Zerlegung ist es das Ziel, die Unterprobleme so klein wie möglich zu machen. Die **Baumbreite** einer Baumzerlegung eines Graphen ist um 1 kleiner als die Größe des größten Unterproblems; die Baumbreite des eigentlichen Graphen ist definiert als die minimale Baumbreite unter allen möglichen Baumzerlegungen. Hat ein Graph die Baumbreite w und wir haben die entsprechende Baumzerlegung, dann kann das Problem in der Zeit $O(nd^{w+1})$ gelöst werden. *Damit sind CSPs mit Constraint-Graphen begrenzter Baumbreite in polynomialer Zeit lösbar.* Leider ist die Ermittlung der Zerlegung mit minimaler Baumbreite NP-hart, aber es gibt heuristische Methoden, die in der Praxis ausreichend gut funktionieren.

Bislang haben wir uns auf die Struktur des Contraint-Graphen konzentriert. Es kann aber auch eine wichtige Struktur in den *Werten* der Variablen geben. Sehen Sie sich das Karteneinfärbeproblem mit n Farben an. Für jede konsistente Lösung gibt es eigentlich eine Menge von $n!$ Lösungen, die durch Permutieren der Farbennamen entstehen. Zum Beispiel wissen wir bei der Karte von Australien, dass WA, NT und SA unterschiedliche Farben haben müssen, doch gibt es $3! = 6$ Möglichkeiten, die drei Farben diesen drei Gebieten zuzuweisen. Das ist die sogenannte **Wertsymmetrie**. Wenn wir die Symmetrie brechen, lässt sich der Suchraum um einen Faktor von $n!$ reduzieren. Dazu führen wir eine **Symmetrie brechende Beschränkung** ein. Für unser Beispiel könnten wir mit einer willkürlichen Reihenfolgebeschränkung $NT < SA < WA$ fordern, dass die drei Werte in alphabetischer Reihenfolge erscheinen müssen. Diese Randbedingung stellt sicher, dass nur eine der $n!$ Lösungen möglich ist: $\{NT = blau, SA = gruen, WA = rot\}$.

Für die Karteneinfärbung war es leicht, eine Beschränkung zu finden, die die Symmetrie beseitigt, und im Allgemeinen lassen sich Beschränkungen finden, die alle bis auf eine symmetrische Lösung in polynomialer Zeit ermitteln. Es ist aber NP-hart, sämtliche Symmetrie unter den Zwischenmengen der Werte während der Suche zu eliminieren. In der Praxis hat sich das Brechen der Wertsymmetrie als wichtig und effektiv für eine breite Palette von Problemen erwiesen.

Bibliografische und historische Hinweise

Die früheste Arbeit im Hinblick auf die Berücksichtigung von Randbedingungen beschäftigte sich größtenteils mit numerischen Randbedingungen. Gleichungsrandbedingungen mit ganzzahligen Domänen wurden im 7. Jahrhundert von dem indischen Mathematiker Brahmagupta untersucht; sie werden häufig auch als **Diophantische Gleichungen** bezeichnet, benannt nach dem griechischen Mathematiker Diophantus (ca. 200–284), der die Domäne positiver rationaler Zahlen betrachtete. Systematische Methoden für die Lösung linearer Gleichungen durch Variableneliminierung wurden von Gauss (1829) untersucht; die Lösung linearer Ungleichheitsrandbedingungen gehen zurück auf Fourier (1827).

CSPs mit endlichen Domänen haben ebenfalls eine lange Geschichte. Beispielsweise ist die **Grapheneinfärbung** (mit dem Spezialfall der Karteneinfärbung) ein altes Problem in der Mathematik. Die 4-Farben-Behauptung (dass jeder ebene Graph mit vier oder weniger Farben eingefärbt werden kann) wurde 1852 zuerst von Francis Guthrie aufgestellt, einem Studenten von De Morgan. Es gab keine Lösung – trotz mehrerer veröffentlichter Behauptungen des Gegenteils –, bis Appel und Haken (1977) einen Beweis angaben (siehe das Buch *Four Colors Suffice* (Wilson, 2004)). Puristen waren enttäuscht, dass sich ein Teil des Beweises auf einen Computer stützte. Georges Gonthier (2008) leitete mithilfe des COQ-Theorembeweisers einen formalen Beweis her, dass der Beweis von Appel und Haken korrekt war.

In der gesamten Geschichte der Informatik treten immer wieder spezielle Klassen von CSPs auf. Eines der einflussreichsten frühen Beispiele war das SKETCHPAD-System (Sutherland, 1963), das geometrische Randbedingungen in Diagrammen löste und der Vorläufer moderner Zeichenprogramme und CAD-Werkzeuge war. Die Identifizierung von CSPs als *allgemeine* Klasse geht auf Ugo Montanari (1974) zurück. Die Reduzierung höherklassiger CSPs auf reine binäre CSPs mit Hilfsvariablen (siehe Übung 6.5) stammt ursprünglich vom Logistiker Charles Sanders Peirce im 19. Jahrhundert. Sie wurde in der CSP-Literatur von Dechter (1990b) erstmals erwähnt und von Bacchus und Beek (1998) ausgearbeitet. CSPs mit Prioritäten für die Lösungen werden in der Optimierungsliteratur umfassend untersucht; eine Verallgemeinerung des CSP-Netzwerkes, sodass es Prioritäten zulässt, ist in Bistarelli et al. (1997) beschrieben. Der Bucket-Eliminierungs-Algorithmus (Dechter, 1999) kann auch auf Optimierungsprobleme angewendet werden.

Methoden für die Weitergabe von Randbedingungen wurden populär nach dem Erfolg von Waltz (1975) für polyhedrale Zeilenbeschriftungsaufgaben der Computervision. Waltz zeigte, dass durch die Weitergabe bei vielen Problemen ein Backtracking völlig überflüssig wurde. Montanari (1974) führte das Konzept der Constraint-Netze und die Weitergabe durch Pfadkonsistenz ein. Alan Mackworth (1977) schlug den AC-3-Algorithmus für die Erzwingung der Kantenkonsistenz vor sowie die allgemeine Idee, das Backtracking mit einem gewissen Maß an Konsistenzerzwingung zu kombinieren. AC-4, ein effizienterer Kantenkonsistenz-Algorithmus, wurde von Mohr und Henderson (1986) entwickelt. Bald nach Erscheinen der Texte von Mackworth begannen die Forscher, mit der Abwägung zwischen den Kosten für die Konsistenzerzwingung und den Vorteilen in Hinblick auf die reduzierte Suche zu experimentieren. Haralick und Elliot (1980) favorisierten den von McGregor (1979) beschriebenen Algorithmus der minimalen Vorabüberprüfung, während Gaschnig (1979) eine vollständige Kantenkonsistenz-Überprüfung nach jeder Variablenzuweisung vorschlug – ein Algorithmus,

der später von Sabin und Freuder (1994) als MAC bezeichnet wurde. Der letztgenannte Text bietet den recht überzeugenden Beweis, dass sich die vollständige Überprüfung der Kantenkonsistenz bei schwierigeren CSPs auszahlt. Freuder (1978, 1982) untersuchte das Konzept der *k*-Konsistenz und seine Beziehung zur Lösungskomplexität von CSPs. Apt (1999) beschreibt ein generisches Algorithmenframework, innerhalb dessen Algorithmen zur Konsistenzweitergabe analysiert werden können. Bessière (2006) gibt einen aktuellen Überblick.

Erstmalig im Kontext der **Constraint-Logikprogrammierung (Constraint Logic Programming)** wurden spezielle Methoden für die Verarbeitung von Randbedingungen höherer Ordnung entwickelt. Marriott und Stuckey (1998) bieten eine ausgezeichnete Beschreibung der Forschung auf diesem Gebiet. Die *Alldiff*-Randbedingung wurde von Regin (1994), Stergiou und Walsh (1999) sowie von van Hoeve (2001) untersucht. Grenzenrandbedingungen wurden von Van Hentenryck et al. (1998) in die Constraint-Logikprogrammierung eingebunden. Ein Überblick der globalen Beschränkungen wird von van Hoeve und Katriel (2006) angegeben.

Sudoku ist zum bekanntesten CSP geworden und wurde als solches von Simonis (2005) beschrieben. Agerbeck und Hansen (2008) beschreiben einige der Strategien und zeigen, dass Sudoku auf einem $n^2 \times n^2$-Brett in die Klasse der NP-harten Probleme fällt. Reeson et al. (2007) stellen ein interaktives Lösungsmodul vor, das auf CSP-Techniken basiert.

Die Backtracking-Suche geht auf Golomb und Baumert (1965) zurück und ihre Anwendung auf CSP ist Bitner und Reingold (1975) zuzuschreiben, obwohl sie den grundlegenden Algorithmus bis ins 19. Jahrhundert zurückverfolgt haben. Bitner und Reingold haben auch die MRV-Heuristik eingeführt, die sie als Heuristik der am *meisten beschränkten Variablen* bezeichnet haben. Brelaz (1979) verwendete die Gradheuristik als Abbruchkriterium nach Anwendung der MRV-Heuristik. Der resultierende Algorithmus ist trotz seiner Einfachheit immer noch die beste Methode für die *k*-Einfärbung zufälliger Graphen. Haralick und Elliot (1980) schlugen die Heuristik mit dem *am wenigsten einschränkenden Wert* vor.

Die grundlegende Backjumping-Methode geht auf John Gaschnig (1977, 1979) zurück. Kondrak und van Beek (1997) zeigten, dass dieser Algorithmus letztlich durch die Vorabüberprüfung subsumiert wird. Konfliktgesteuertes Backjumping wurde von Prosser (1993) entwickelt. Die allgemeinste und leistungsfähigste Form intelligenten Backtrackings wurde schon sehr früh von Stallman und Sussman (1977) entwickelt. Ihre Technik des **abhängigkeitsgesteuerten Backtrackings** führte zur Entwicklung von **Wahrheitsverwaltungssystemen (Truth Maintenance Systems)** (Doyle, 1979), die wir in *Abschnitt 12.6.2* beschreiben. Die Verknüpfung zwischen den beiden Bereichen wurde von de Kleer (1989) analysiert.

Die Arbeit von Stallman und Sussman führte auch zum Konzept **Lernen von Beschränkungen** (**Contraint Learning**), bei dem die durch die Suche ermittelten partiellen Ergebnisse gespeichert und später in der Suche wiederverwendet werden können. Das Konzept wurde von Dechter (1990a) formal in die Backtracking-Suche aufgenommen. **Backmarking** (Gaschnig, 1979) ist eine besonders einfache Methode, wobei konsistente und inkonsistente paarweise Zuweisungen gespeichert werden, um die erneute Überprüfung von Randbedingungen zu vermeiden. Das Backmarking kann mit dem konfliktgesteuerten Backjumping kombiniert werden; Kondrak und van Beek (1997) zeigen

einen hybriden Algorithmus, der nachweisbar jede der beiden Methoden separat realisiert. Die Methode des **dynamischen Backtrackings** (Ginsberg, 1993) behält erfolgreiche partielle Zuweisungen von späteren Untermengen von Variablen zurück, wenn er das Backtracking für eine frühere Auswahl vornimmt, die den späteren Erfolg nicht ungültig macht.

Empirische Untersuchungen verschiedener Backtracking-Methoden mit Zufallskomponente wurden von Gomes et al. (2000) sowie Gomes und Selman (2001) durchgeführt. Van Beek (2006) gibt einen Überblick zum Backtracking.

Die lokale Suche in CSPs wurde durch die Arbeit von Kirkpatrick et al. (1983) zum Simulated Annealing (siehe *Kapitel 4*) bekannt, das häufig für Planungsaufgaben benutzt wird. Die MIN-CONFLICTS-Heuristik wurde von Gu (1989) eingeführt und unabhängig dazu von Minton et al. (1992) entwickelt. Sosic und Gu (1994) zeigten, wie sie angewendet werden kann, um das 3.000.000-Damen-Problem in weniger als einer Minute zu lösen. Der erstaunliche Erfolg der lokalen Suche unter Verwendung von MIN-CONFLICTS für das n-Damen-Problem führte zum Wiederaufleben der Natur und der allgemeinen Gültigkeit von „einfachen“ und „schwierigen“ Problemen. Peter Cheeseman et al. (1991) untersuchten die Schwierigkeit zufällig erzeugter CSPs und erkannten, dass fast alle diese Probleme entweder leicht zu lösen sind oder keine Lösung aufweisen. Nur wenn die Parameter des Problemgenerators in einem bestimmten engen Bereich gesetzt werden, in dem etwa die Hälfte der Probleme lösbar sind, finden wir „schwierige“ Probleminstanzen. Wir werden in *Kapitel 7* weiter über dieses Phänomen sprechen. Konolige (1994) zeigte, dass lokale Suche der Backtracking-Suche bei Problemen mit einem gewissen Grad an lokaler Struktur überlegen ist; dies führte zur Arbeit, die lokale Suche und Inferenz kombinierte, wie zum Beispiel die von Pinkas und Dechter (1995). Hoos und Tsang (2006) geben einen Überblick über lokale Suchverfahren.

Arbeiten zur Struktur und Komplexität von CSPs stammen ursprünglich von Freuder (1985), der zeigte, dass Suchen für kantenkonsistente Bäume ohne jedes Backtracking funktioniert. Ein ähnliches Ergebnis mit Erweiterungen auf azyklische Hypergraphen wurde in der Datenbankgemeinde (Beeri et al., 1983) entwickelt. Bayardo und Miranker (1994) stellen einen Algorithmus für baumstrukturierte CSPs vor, der ohne jegliche Vorverarbeitung in linearer Zeit läuft.

Seit Veröffentlichung dieser Untersuchungen gab es große Fortschritte bei der Entwicklung allgemeinerer Ergebnisse, die die Lösungskomplexität eines CSP mit der Struktur seines Constraint-Graphen miteinander in Beziehung brachten. Das Konzept der Baumbreite wurde von den Graphentheoretikern Robertson und Seymour (1986) eingeführt. Dechter und Pearl (1987, 1989) bauten auf der Arbeit von Freuder auf; sie wendeten dasselbe Konzept (das sie als **induzierte Breite** bezeichneten) auf CSPs an und entwickelten den in *Abschnitt 6.5* beschriebenen Ansatz zur Baumzerlegung. Aufbauend auf dieser Arbeit und Ergebnissen aus der Datenbanktheorie entwickelten Gottlob et al. (1999a, 1999b) ein Konzept, die **Hyperbaumbreite**, die auf der Einordnung des CSP als Hypergraph basiert. Neben der Darstellung, dass jedes CSP mit der Hyperbaumbreite w in der Zeit $O(n^{w+1} \log n)$ gelöst werden kann, zeigten sie auch, dass die Hyperbaumbreite alle zuvor definierten Maße der „Breite“ subsumieren, nämlich in der Hinsicht, dass es Fälle gibt, wo die Hyperbaumbreite begrenzt ist, die anderen Maße dagegen unbegrenzt sind.

Das Interesse an zurückschauenden Ansätzen zum Backtracking erhielt neue Impulse durch die Arbeit von Bayardo und Schrag (1997), deren RELSAT-Algorithmus das Constraint-Learning und Backjumping kombinierte. Außerdem zeigten sie, dass dieser Algorithmus vielen anderen Algorithmen zu dieser Zeit überlegen war. Dies führte zu AND/OR-Suchalgorithmen, die sich sowohl auf CSPs als auch probabilistisches Schließen anwenden ließen (Dechter und Mateescu, 2007). Brown et al. (1988) führten das Konzept der Symmetriebrechung in CSPs ein und Gent et al. (2006) geben einen neueren Überblick dazu.

Das Gebiet der **verteilten CSPs** beschäftigt sich mit der Lösung von CSPs, wenn es eine Sammlung von Agenten gibt, die jeweils eine Teilmenge der Beschränkungsvariablen steuern. Seit 2000 wurden zu diesem Problem mehrere jährliche Workshops abgehalten und es gibt gute Abhandlungen an anderen Stellen (Collin et al., 1999; Pearce et al., 2008; Shoham und Leyton-Brown, 2009).

Das Vergleichen von CSP-Algorithmen ist vor allem eine empirische Wissenschaft: Nur wenige theoretische Ergebnisse zeigen, dass der eine Algorithmus einen anderen bei allen Problemen dominiert; stattdessen sind Sie auf eigene Experimente angewiesen, um festzustellen, welche Algorithmen bei typischen Probleminstanzen besser als andere abschneiden. Wie Hooker (1995) betont, müssen wir sorgfältig unterscheiden zwischen Testen in Konkurrenzsituationen – wie es in Wettbewerben zwischen Algorithmen basierend auf der Laufzeit auftritt – und wissenschaftlichem Testen, dessen Ziel es ist, die Eigenschaften eines Algorithmus, die seine Wirksamkeit in einer Problemklasse bestimmen, zu erkennen.

Die neuesten Lehrbücher von Apt (2003) und Dechter (2003) sowie die Sammlung von Rossi et al. (2006) sind ausgezeichnete Quellen zur Verarbeitung von Randbedingungen. Es gibt auch mehrere gute Überblicke älteren Datums, unter anderem von Kumar (1992), Dechter und Frost (2002) sowie Batrak (2001) ebenso wie die Enzyklopädieeinträge von Dechter (1992) und Mackworth (1992). Pearson und Jeavons (1997) bieten einen Überblick über lenkbare CSP-Klassen, wobei sowohl strukturelle Zerlegungsmethoden als auch Methoden, die sich auf die Eigenschaften der eigentlichen Domänen oder Randbedingungen beziehen, betrachtet werden. Kondrak und van Beek (1997) bieten einen analytischen Überblick über Backtracking-Suchalgorithmen, und Bacchus und van Run (1995) präsentieren einen etwas empirischeren Überblick. Die Programmierung mit Beschränkungen wird in den Büchern von Apt (2003) sowie Fruhwirth und Abdennadher (2003) behandelt. Mehrere interessante Anwendungen sind in der Sammlung von Freuder und Mackworth (1994) beschrieben. Artikel zu CSPs erscheinen regelmäßig in *Artificial Intelligence* sowie in einem Journal für Spezialisten, *Constraints*. Die wichtigste Konferenz zu diesem Thema ist die *Conference on Principles and Practice of Constraint Programming*, häufig auch als *CP* abgekürzt.

Zusammenfassung

- **Probleme unter Rand- oder Nebenbedingungen** (**Constraint Satisfaction Problems**, **CSPs**) stellen einen Zustand mit einer Menge von Variablen-/Wert-Paaren dar und repräsentieren die Bedingungen für eine Lösung durch eine Menge von Beschränkungen auf den Variablen. Viele wichtige Probleme aus der realen Welt können als CSPs beschrieben werden. Die Struktur eines CSP kann durch seinen **Constraint-Graphen** dargestellt werden.
- Eine Reihe von Inferenzverfahren verwenden die Beschränkungen, um zu schlussfolgern, welche Variablen-/Wert-Paare konsistent sind und welche nicht. Hierzu gehören Knoten-, Kanten-, Pfad- und k-Konsistenz.
- Die **Backtracking-Suche,** eine Form der Tiefensuche, wird häufig für die Lösung von CSPs verwendet. Inferenz lässt sich mit Suche verflechten.
- Die Heuristik mit **minimalen verbleibenden Werten** sowie die **Gradheuristik** sind domänenunabhängige Methoden für die Entscheidung, welche Variable in einer Backtracking-Suche als Nächstes auszuwählen ist. Die Heuristik des **am wenigsten einschränkenden Wertes** hilft bei der Entscheidung, welcher Wert zuerst für eine bestimmte Variable auszuprobieren ist. Backtracking tritt auf, wenn für eine Variable keine erlaubten Zuweisungen mehr gefunden werden. **Konfliktgesteuertes Backjumping** führt ein Backtracking direkt zur Ursache des Problems durch.
- Die lokale Suche unter Verwendung der MIN-CONFLICTS-**Heuristik** wurde mit großem Erfolg auf CSPs angewendet.
- Die Lösungskomplexität eines CSP ist eng mit der Struktur seines Constraint-Graphen verknüpft. Baumstrukturierte Probleme können in linearer Zeit gelöst werden. **Schnittmengenkonditionierung** kann ein allgemeines CSP zu einem baumstrukturierten CSP reduzieren und ist sehr effizient, wenn eine kleine Schnittmenge gefunden werden kann. Techniken der **Baumzerlegung** wandeln das CSP in einen Baum aus Unterproblemen um und sind effizient, wenn die **Baumbreite** des Constraint-Graphen klein ist.

Übungen zu Kapitel 6

Lösungshinweise

1 Wie viele Lösungen gibt es für das Karteneinfärbungsproblem in Abbildung 6.1? Wie viele Lösungen ergeben sich, wenn vier Farben erlaubt sind? Wie lautet die Antwort für zwei Farben?

2 Betrachten Sie das Problem, Kreuzworträtsel zu erstellen (nicht zu lösen),[5] indem Sie Wörter in ein rechteckiges Raster einpassen. Das Raster, das als Teil des Problems vorgegeben ist, gibt an, welche Quadrate leer und welche schattiert sind. Nehmen Sie an, dass eine Liste mit Wörtern (z.B. ein Wörterbuch) bereitgestellt wird und Sie die Aufgabe haben, unter Verwendung einer beliebigen Untermenge der Liste die leeren Quadrate einzutragen. Formulieren Sie dieses Problem auf genau zwei Arten:

a. Als allgemeines Suchproblem: Wählen Sie einen geeigneten Suchalgorithmus und geben Sie eine Heuristikfunktion an, wenn Sie denken, es ist eine dafür erforderlich. Ist es besser, die leeren Quadrate mit jeweils einem Buchstaben oder mit jeweils einem Wort gleichzeitig auszufüllen?

b. Als CSP: Sollen die Variablen dabei Wörter oder Buchstaben sein?

Welche Formulierung halten Sie für besser? Warum?

3 Geben Sie exakte Formulierungen für die folgenden Probleme als CSPs an:

a. Geradlinig ausgerichtete **Teppichbodenverlegung**: Finden Sie nicht überlappende Positionen in einem großen Rechteck für mehrere kleinere Rechtecke.

b. **Stundenplan:** Es gibt eine feststehende Anzahl von Lehrern und Klassenzimmern, eine Liste der anzubietenden Kurse sowie eine Liste möglicher Zeitpunkte für Kurse. Jeder Lehrer kann mehrere verschiedene Kurse halten.

c. **Hamilton-Tour:** Gegeben ist ein Netz von Städten, die durch Straßen miteinander verbunden sind. Wählen Sie eine Reihenfolge aus, um alle Städte in einem Land genau einmal zu besuchen.

4 Lösen Sie das kryptoarithmetische Problem aus Abbildung 6.2 manuell und verwenden Sie dafür Backtracking mit Vorabüberprüfung und die MRV-Heuristik sowie die Heuristik mit dem am wenigsten beschränkenden Wert.

5 Zeigen Sie, wie eine einzelne ternäre Randbedingung wie zum Beispiel „A + B = C" in drei binäre Randbedingungen umgewandelt werden kann, indem eine Hilfsvariable verwendet wird. Sie können von endlichen Domänen ausgehen. (*Hinweis:* Verwenden Sie eine neue Variable, die Werte annimmt, bei denen es sich um Paare anderer Werte handelt, und verwenden Sie Randbedingungen wie „X ist das erste Element des Paares Y".) Anschließend zeigen Sie, wie Randbedingungen mit mehr als drei Variablen ähnlich behandelt werden können. Zeigen Sie schließlich, wie sich unäre Randbedingungen eliminieren lassen, indem die Domänen von Variablen verändert werden. Damit ist die Demonstration abgerundet, dass jedes CSP in ein CSP mit nur binären Randbedingungen umgewandelt werden kann.

5 Ginsberg *et al.* (1990) beschreiben mehrere Methoden für die Konstruktion von Kreuzworträtseln. Littman *et al.* (1999) widmen sich dem schwierigeren Problem, sie zu lösen.

6 Betrachten Sie das folgende Logikrätsel: In fünf Häusern, die in unterschiedlichen Farben angestrichen sind, wohnen fünf Personen unterschiedlicher Nationalitäten, die jeweils unterschiedliche Zigarettenmarken rauchen, ein jeweils unterschiedliches Getränk bevorzugen und jeweils unterschiedliche Haustiere haben. Anhand der folgenden Tatsachen soll die Frage „Wo lebt das Zebra und in welchem Haus wird Wasser getrunken?" beantwortet werden:

Der Engländer lebt im roten Haus.
Dem Spanier gehört der Hund.
Der Norweger lebt in dem ersten Haus links.
Das grüne Haus befindet sich unmittelbar rechts des elfenbeinfarbenen Hauses.
Der Mann, der Hershey-Stangen isst, lebt in dem Haus rechts neben dem Mann mit dem Fuchs.
Im gelben Haus werden Kit Kats gegessen.
Der Norweger lebt neben dem blauen Haus.
Der Smarties-Esser züchtet Schnecken.
Der Snickers-Esser trinkt Orangensaft.
Der Ukrainer trinkt Tee.
Der Japaner isst Milky Ways.
Im Haus neben dem Haus mit dem Pferd wird Kit Kats gegessen.
Im grünen Haus wird Kaffee getrunken.
Im mittleren Haus wird Milch getrunken.

Erörtern Sie verschiedene Darstellungen dieses Problems als CSP. Warum sollte eine dieser Darstellungen favorisiert werden?

7 Betrachten Sie den Graphen mit acht Knoten A_1, A_2, A_3, A_4, H, T, F_1, F_2. Hierbei ist A_i für alle i mit $A_i{+}1$ verbunden, jeder A_i ist mit H verbunden, H ist mit T verbunden und T ist mit jedem F_i verbunden. Suchen Sie eine Einfärbung mit drei Farben mithilfe der folgenden Strategie: Backtracking mit konfliktgesteuertem Backjumping, der Variablenreihenfolge A_1, H, A_4, F_1, A_2, F_2, A_3, T und der Wertereihenfolge R, G, B.

8 Erläutern Sie, warum in einer CSP-Suche die Heuristik zweckmäßig ist, die Variable zu wählen, die am *meisten* beschränkt ist, aber den Wert, der am *wenigsten* beschränkend ist.

9 Generieren Sie wie folgt zufällige Instanzen des Karteneinfärbungsproblems: Verstreuen von n Punkten auf dem Einheitsquadrat; Auswählen eines Punktes X per Zufall, Verbinden von X durch eine Gerade mit dem nächsten Punkt Y, sodass X nicht bereits mit Y verbunden ist und die Gerade keine andere Linie kreuzt; wiederholen Sie den vorherigen Schritt, bis keine Verbindungen mehr möglich sind. Die Punkte stellen Gebiete auf der Karte dar und die Linien verbinden Nachbarn. Versuchen Sie nun k-Einfärbungen für jede Karte zu finden, sowohl für $k = 3$ als auch $k = 4$. Verwenden Sie dabei MIN-CONFLICTS, Backtracking, Backtracking mit Vorabüberprüfung und Backtracking mit MAC. Konstruieren Sie eine Tabelle der mittleren Zeiten für jeden Algorithmus für Werte von n bis zum größten Wert, den Sie handhaben können. Erörtern Sie Ihre Ergebnisse.

10 Zeigen Sie mithilfe des AC-3-Algorithmus, dass die Kantenkonsistenz die Inkonsistenz der partiellen Zuweisung $\{WA = rot, V = blau\}$ für das in Abbildung 6.1 gezeigte Problem erkennt.

11 Was ist die Komplexität für den ungünstigsten Fall bei der Ausführung von AC-3 für ein baumstrukturiertes CSP?

12 AC-3 stellt *jede* Kante (X_k, X_i) in die Schlange zurück, wenn ein *beliebiger* Wert aus der Domäne von X_i entfernt wird, selbst wenn jeder Wert von X_k konsistent mit mehreren verbleibenden Werten von X_i ist. Angenommen, wir beobachten für jede Kante (X_k, X_i) die Anzahl der verbleibenden Werte von X_i, die mit jedem Wert von X_k konsistent sind. Erläutern Sie, wie diese Zahlen effizient zu aktualisieren sind, und zeigen Sie, dass die Kantenkonsistenz in der Gesamtzeit $O(n^2 d^2)$ erzwungen werden kann.

13 Der Algorithmus TREE-CSP-SOLVER (Abbildung 6.10) macht Kanten konsistent. Er beginnt bei den Blättern und arbeitet sich rückwärts bis zur Wurzel vor. Warum? Was passiert, wenn er die entgegengesetzte Richtung gehen würde?

14 Wir haben Sudoku als CSP eingeführt, das durch Suche über partiellen Zuweisungen gelöst werden soll, weil dies der Weg ist, den die meisten Rätsellöser für Sudoku im Allgemeinen einschlagen. Es ist natürlich auch möglich, diese Probleme mit lokaler Suche über vollständigen Zuweisungen anzugehen. Wie gut würde ein lokaler Problemlöser mithilfe der MIN-CONFLICTS-Heuristic bei Sudoku-Problemen abschneiden?

15 Definieren Sie in eigenen Worten die Begriffe „Beschränkung" (bzw. „Randbedingung"), „Kommutativität", „Kantenkonsistenz", „Backjumping", „MIN-CONFLICTS" und „zyklische Schnittmenge".

16 Angenommen, man weiß, dass ein Graph eine zyklische Schnittmenge von nicht mehr als k Knoten aufweist. Beschreiben Sie einen einfachen Algorithmus, der eine minimale zyklische Schnittmenge findet, deren Laufzeit nicht länger als $O(n^k)$ für ein CSP mit n Variablen ist. Suchen Sie in der Literatur nach Methoden, um annähernd minimale Schnittmengen in einer Zeit zu finden, die polynomial zur Größe der Schnittmenge ist. Ist trotz der Existenz solcher Algorithmen die Methode der zyklischen Schnittmengen noch zweckmäßig?

17 Betrachten Sie das Problem, eine Oberfläche (vollständig und mit exakter Abdeckung) durch n Dominosteine (2×1-Rechtecke) zu kacheln. Die Oberfläche ist eine beliebige Auflistung von $2n$ 3×3-Quadraten, die durch Kanten verbunden sind (d.h. an einer Kante, nicht einer Ecke, anliegen) (z.B. ein Schachbrett, ein Schachbrett, bei dem einige Felder fehlen, eine 10×1-Zeile von Quadraten etc.).

a. Formulieren Sie dieses Problem genau als CSP, bei dem die Dominosteine die Variablen sind.

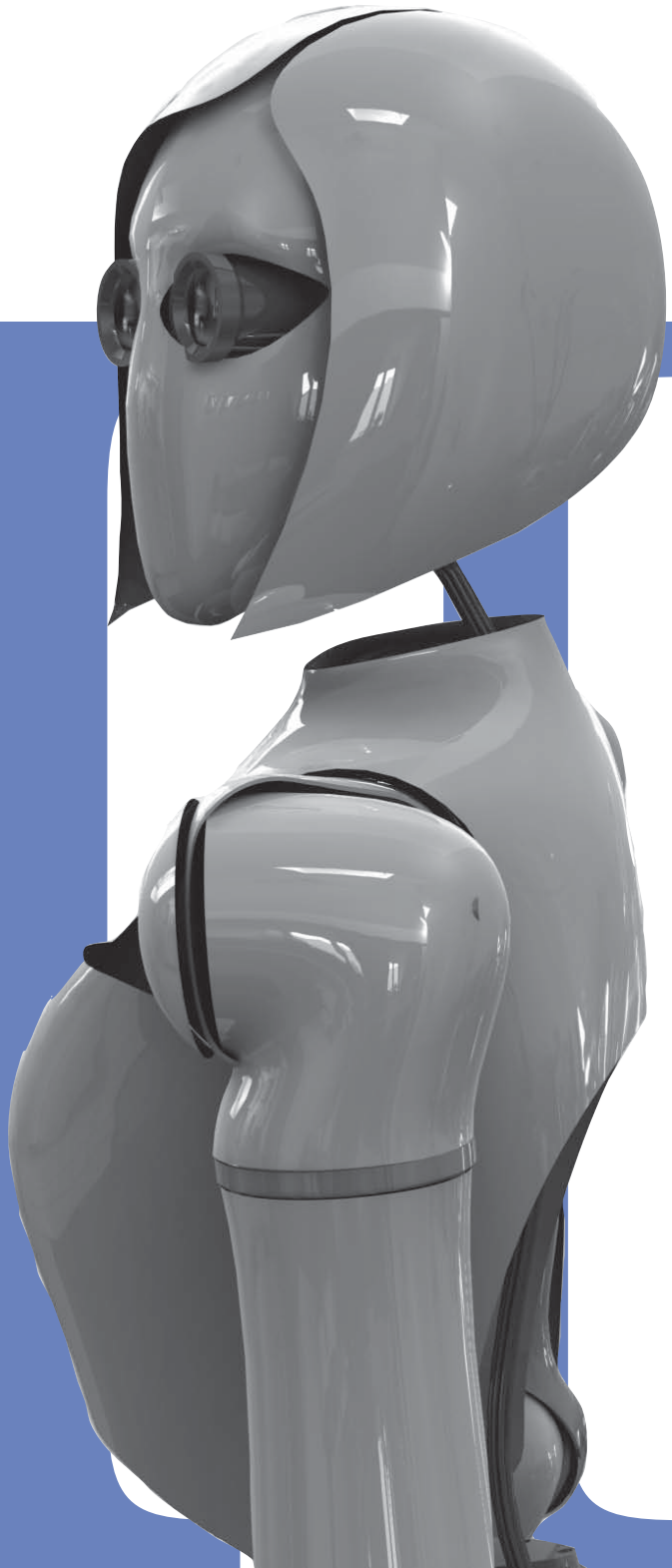
b. Formulieren Sie dieses Problem genau als CSP, bei dem die Quadrate die Variablen sind, wobei der Zustandsraum so klein wie möglich zu halten ist. (Hinweis: Spielt es eine Rolle, welche konkreten Dominosteine auf einem bestimmten Paar von Quadraten zu liegen kommen?)

c. Konstruieren Sie eine Oberfläche, die aus sechs Feldern besteht, sodass Ihre CSP-Formulierung von Teil (b) einen *baumstrukturierten* Graphen hat.

d. Beschreiben Sie genau die Menge der lösbaren Instanzen, die einen baumstrukturierten Graphen besitzen.

TEIL III

Wissen, Schließen und Planen



Logische Agenten

7

ÜBERBLICK

In diesem Kapitel entwerfen wir Agenten, die Darstellungen einer komplexen Welt bilden können, durch einen Schlussfolgerungsprozess neue Darstellungen über die Welt ableiten und anhand dieser neuen Darstellungen schließen, was zu tun ist.

Menschen wissen Dinge – so scheint es. Und was sie wissen, hilft ihnen, Dinge zu tun. Das sind keine leeren Behauptungen. Es handelt sich um gewichtige Aussagen darüber, wie Intelligenz von Menschen erreicht wird – nicht durch reine Reflexmechanismen, sondern durch **Schlussfolgerungsprozesse**, die auf internen Darstellungen von Wissen arbeiten. In der künstlichen Intelligenz wird dieses Konzept durch **wissensbasierte Agenten** verkörpert.

Die in den *Kapiteln 3* und *4* beschriebenen problemlösenden Agenten wissen Dinge, jedoch nur in einem sehr begrenzten, unflexiblen Sinn. Zum Beispiel ist das Übergangsmodell für das 8-Puzzle – Wissen, welche Aktionen auszuführen sind – innerhalb des domänenspezifischen Codes der RESULT-Funktion verborgen. Es lässt sich verwenden, um das Ergebnis von Aktionen vorherzusagen, aber nicht, um zu schließen, dass zwei Kacheln nicht denselben Platz einnehmen können oder dass Zustände mit ungerader Parität nicht von Zuständen mit gerader Parität erreicht werden können. Die von problemlösenden Agenten verwendeten atomaren Darstellungen sind ebenfalls recht einschränkend. Wenn ein Agent in einer partiell beobachtbaren Umgebung darstellen möchte, was er über den aktuellen Zustand weiß, kann er das nur so realisieren, dass er alle möglichen konkreten Zustände auflistet – ein hoffnungsloses Unterfangen in großen Umgebungen.

Kapitel 6 führte das Konzept ein, Zustände als Zuweisungen von Werten zu Variablen darzustellen. Dies ist ein Schritt in die richtige Richtung. Bestimmte Teile des Agenten können somit in domänenunabhängiger Weise arbeiten und es sind effizientere Algorithmen möglich. In diesem und den folgenden Kapiteln führen wir diesen Schritt sozusagen zu seinem logischen Abschluss – wir entwickeln **Logik** als allgemeine Klasse von Darstellungen, um wissensbasierte Agenten zu unterstützen. Derartige Agenten können Informationen kombinieren und erneut zusammenstellen, um die unterschiedlichsten Aufgaben zu erledigen. Häufig ist dieser Prozess jedoch relativ weit von den momentanen Anforderungen entfernt – etwa damit zu vergleichen, wenn ein Mathematiker ein Theorem beweist oder ein Astronom die Lebenserwartung der Erde berechnet. Wissensbasierte Agenten können neue Aufgaben in Form explizit beschriebener Ziele entgegennehmen, durch Unterweisung oder Erlernen neuen Wissens über die Umgebung schnell Kompetenz erlangen und sich an Änderungen in der Umgebung anpassen, indem sie das relevante Wissen aktualisieren.

Wir beginnen in *Abschnitt 7.1* mit dem allgemeinen Agentenentwurf. *Abschnitt 7.2* stellt eine einfache neue Umgebung vor – die Wumpus-Welt – und veranschaulicht die Arbeitsweise eines wissensbasierten Agenten, ohne zu sehr ins technische Detail zu gehen. Dann erläutern wir in *Abschnitt 7.3* die allgemeinen Konzepte der **Logik** und in *Abschnitt 7.4* die Eigenheiten der **Aussagenlogik**. Auch wenn die Aussagenlogik weniger ausdrucksstark als die **Logik erster Stufe** (*Kapitel 8*) ist, hilft sie, die grundlegenden Konzepte der Logik zu verdeutlichen. Außerdem gibt es umfassend entwickelte Inferenztechnologien, die wir in den *Abschnitten 7.5* und *7.6* beschreiben. *Abschnitt 7.7* schließlich kombiniert das Konzept wissensbasierter Agenten mit der Technologie der Aussagenlogik, um einige einfache Agenten für die Wumpus-Welt zu erzeugen.

7.1 Wissensbasierte Agenten

Die zentrale Komponente eines wissensbasierten Agenten ist seine **Wissensbasis** (**Knowledge Base**, **KB**). Eine Wissensbasis ist eine Menge von **Sätzen**. (Hier ist „Sätze“ als technischer Begriff zu verstehen. Er hat mit dem Konzept der Sätze in der natürlichen Sprache zu tun, ist aber nicht identisch damit.) Jeder Satz wird in einer Sprache ausgedrückt, die als **Wissensrepräsentationssprache** bezeichnet wird und bestimmte Annahmen über die Welt darstellt. Manchmal würdigen wir einen Satz mit dem Namen **Axiom**, wenn der Satz als gegeben hinzunehmen ist, ohne von anderen Sätzen abgeleitet zu sein.

Es muss eine Möglichkeit geben, der Wissensbasis neue Sätze hinzuzufügen, und eine Möglichkeit, Bekanntes abzufragen. Die Standardnamen für diese Aufgaben sind TELL (mitteilen) und ASK (fragen). Beide Aufgaben können eine **Inferenz** (Schlussfolgerung) beinhalten – d.h. die Ableitung neuer Sätze von alten Sätzen. Inferenz muss folgender Anforderung genügen: Wenn man einer Wissensbasis eine Frage stellt (ASK), muss die Antwort aus dem folgen, was der Wissensbasis zuvor mitgeteilt (TELL) wurde. Später in diesem Kapitel werden wir genauer auf die wichtige Bedeutung des Schlüsselwortes „folgen“ eingehen. Wir nehmen zunächst einfach an, dass der Schlussfolgerungsprozess nicht einfach Dinge erfinden soll.

▶ Abbildung 7.1 zeigt das Gerüst eines wissensbasierten Agentenprogramms. Wie alle unsere Agenten nimmt es Wahrnehmungen als Eingabe entgegen und liefert eine Aktion zurück. Der Agent verwaltet eine Wissensbasis, *KB*, die anfänglich ein bestimmtes **Hintergrundwissen** enthalten kann.

```
function KB-AGENT(percept) returns eine action
  persistent: KB, eine Wissensbasis
              t, ein Zähler, anfangs 0, gibt die Zeit an
  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

Abbildung 7.1: Ein generischer wissensbasierter Agent. Der Agent fügt seiner Wissensbasis eine gegebene Wahrnehmung hinzu, fragt die Wissensbasis nach der besten Aktion ab und teilt der Wissensbasis mit, dass er die Aktion tatsächlich ausgeführt hat.

Immer wenn das Agentenprogramm aufgerufen wird, erledigt es zwei Dinge: Erstens teilt es der Wissensbasis mit (TELL), was es wahrnimmt. Zweitens fragt es die Wissensbasis ab (ASK), welche Aktionen es ausführen soll. Bei der Beantwortung dieser Frage kann ein umfassendes Schließen zum aktuellen Zustand der Welt erfolgen, was die Ergebnisse möglicher Aktionsfolgen betrifft, usw. Drittens teilt das Agentenprogramm der Wissensbasis mit (TELL), welche Aktion es ausgewählt hat, und der Agent bewertet die Aktion.

Die Details der Darstellungssprache sind in drei Funktionen verborgen, die die Schnittstelle zwischen den Sensoren und den Aktuatoren einerseits und der Kerndarstellung und dem Schlussfolgerungssystem andererseits implementieren. Die Funktion MAKE-PERCEPT-SENTENCE konstruiert einen Satz unter der Annahme, dass der Agent die angegebene Wahrnehmung zur angegebenen Zeit empfangen hat. Die Funktion MAKE-ACTION-QUERY konstruiert einen Satz, der fragt, welche Aktion zur aktuellen Zeit ausgeführt werden soll. Schließlich konstruiert die Funktion MAKE-ACTION-SENTENCE einen Satz, der versichert, dass die gewählte Aktion ausgeführt wurde. Die Details des Inferenzmechanismus sind in TELL und ASK verborgen. Spätere Abschnitte werden diese Details offenlegen.

Der Agent in Abbildung 7.1 scheint den Agenten mit dem in *Kapitel 2* beschriebenen internen Zustand ganz ähnlich zu sein. Aufgrund der Definitionen von TELL und ASK jedoch ist der wissensbasierte Agent kein willkürliches Programm für die Berechnung von Aktionen. Er ist vergleichbar mit einer Beschreibung auf der **Wissensebene**, wo wir nur spezifizieren müssen, was der Agent weiß und welche Ziele er hat, um sein Verhalten festzulegen. Ein automatisiertes Taxi beispielsweise könnte das Ziel haben, einen Fahrgast nach Marin County zu bringen, und wissen, dass sich dies in San Francisco befindet und die Golden Gate Bridge die einzige Verbindung zwischen den beiden Positionen ist. Wir können dann erwarten, dass es die Golden Gate Bridge überquert, *weil es weiß, dass es auf diese Weise sein Ziel erreicht*. Beachten Sie, dass diese Analyse unabhängig davon ist, wie das Taxi auf **Implementierungsebene** funktioniert. Es spielt keine Rolle, ob sein geografisches Wissen als verkettete Liste oder als Pixelabbildung implementiert ist oder ob es schließt, indem es Symbol-Strings manipuliert, die in Registern abgelegt sind, oder rauschende Signale in einem Netzwerk aus Neuronen weitergibt. Dies ist der sogenannte **deklarative** Ansatz der Systemerstellung. Im Gegensatz dazu kodiert der **prozedurale** Ansatz die gewünschten Verhaltensweisen direkt als Programmcode. In den 1970er und 1980er Jahren führten Verfechter beider Ansätze heiße Debatten. Heute verstehen wir, dass ein erfolgreicher Agent oftmals sowohl deklarative als auch prozedurale Elemente in seinem Entwurf kombinieren muss und dass sich deklaratives Wissen häufig in effizienteren prozeduralen Code kompilieren lässt.

Einen wissensbasierten Agenten können wir auch mit einem Mechanismus ausstatten, der ihn in die Lage versetzt, selbst zu lernen. Diese Mechanismen, die in *Kapitel 18* genauer beschrieben werden, erzeugen ein allgemeines Wissen zur Umgebung aus einer Folge von Wahrnehmungen. Ein lernender Agent kann völlig autonom sein.

7.2 Die Wumpus-Welt

In diesem Abschnitt beschreiben wir eine Umgebung, in der wissensbasierte Agenten ihre Qualität beweisen können. Die **Wumpus-Welt** besteht aus einer Höhle mit Räumen, die über Durchgänge miteinander verbunden sind. Irgendwo in der Höhle lauert der Wumpus, eine Bestie, die jeden frisst, der seinen Raum betritt. Der Wumpus kann von einem Agenten erschossen werden, doch besitzt der Agent nur einen einzigen Pfeil. Einige Räume enthalten Falltüren, durch die jeder fällt, der diese Räume betritt (außer dem Wumpus, der zu groß ist, um durchzufallen). Die einzige Verlockung, sich in dieser Umgebung zu bewegen, ist die Aussicht, einen Berg Gold zu finden. Obwohl die Wumpus-Welt angesichts der modernen Computerspiele recht archaisch anmuten mag, veranschaulicht sie einige wichtige Aspekte der Intelligenz.

▶ Abbildung 7.2 zeigt ein Beispiel für die Wumpus-Welt. Die genaue Definition der Aufgabenumgebung ist, wie in *Abschnitt 2.3* gezeigt, durch die folgende PEAS-Beschreibung (Performance, Environment, Actuators, Sensors) gegeben:

- **Leistungsmaß (Performance):** +1000 für Entkommen aus der Höhle zusammen mit dem Gold, −1000, wenn man durch eine Falltür fällt oder vom Wumpus gefressen wird, −1 für jede unternommene Aktion und −10 für die Verwendung des Pfeiles.
- **Umgebung (Environment):** Ein 4 × 4-Raster von Räumen. Der Agent beginnt seinen Weg immer im Feld mit der Bezeichnung [1,1] und sieht nach rechts. Die Positionen des Goldes und des Wumpus werden zufällig nach einer Gleichverteilung gewählt, befinden

sich jedoch nicht auf dem Ausgangsfeld des Agenten. Darüber hinaus kann jedes Feld außer dem Anfangsfeld mit einer Wahrscheinlichkeit von 0,2 eine Falltür haben.

- **Aktuatoren (Actuators):** Der Agent kann sich nach vorne bewegen (*Forward*), sich um 90 Grad nach links (*TurnLeft*) oder um 90 Grad nach rechts (*TurnRight*) drehen. Der Agent stirbt einen schrecklichen Tod, wenn er auf ein Feld tritt, das eine Falltür hat, oder wenn er einen lebenden Wumpus entdeckt. (Es ist sicher, wenn auch nicht ganz geruchsneutral, auf ein Feld mit einem toten Wumpus zu steigen.) Eine Vorwärtsbewegung hat keine Wirkung, wenn der Agent vor einer Mauer steht. Die Aktion *Grab* (Greifen) kann verwendet werden, um ein Objekt aufzunehmen, das sich auf demselben Feld wie der Agent befindet. Mit der Aktion *Shoot* (Schießen) lässt sich ein Pfeil geradlinig in die Richtung schießen, in die der Agent sieht. Der Pfeil fliegt, bis er den Wumpus trifft (und damit tötet) oder an eine Mauer prallt. Der Agent hat nur einen einzigen Pfeil, deshalb hat auch nur die erste *Shoot*-Aktion eine Wirkung. Schließlich ist es mit einer Aktion *Climb* möglich, aus der Höhle zu entkommen, allerdings nur von Feld [1,1].
- **Sensoren (Sensors):** Der Agent ist mit fünf Sensoren ausgestattet, die ihm jeweils unterschiedliche Informationen bereitstellen:
  - In dem Feld, das den Wumpus enthält, sowie in allen direkt (nicht diagonal) benachbarten Feldern nimmt der Agent einen üblen Gestank (*Stench*) wahr.
  - In Feldern, die direkt einem Feld mit Falltür benachbart sind, nimmt der Agent einen Luftzug (*Breeze*) wahr.
  - In dem Feld, auf dem sich das Gold befindet, nimmt der Agent ein Glitzern (*Glitter*) wahr.
  - Läuft ein Agent gegen eine Wand, nimmt er einen Stoß (*Bump*) wahr.
  - Wenn der Wumpus getötet wird, stößt er einen schrecklichen Schrei (*Scream*) aus, den jeder in der Höhle hören kann.

  Die Wahrnehmungen werden dem Agentenprogramm in Form einer Liste mit fünf Symbolen übergeben. Gibt es zum Beispiel Gestank und Luftzug, aber kein Glitzern, Stoßen oder Schreien, erhält der Agent die Wahrnehmung [*Stench*, *Breeze*, *None*, *None*, *None*].

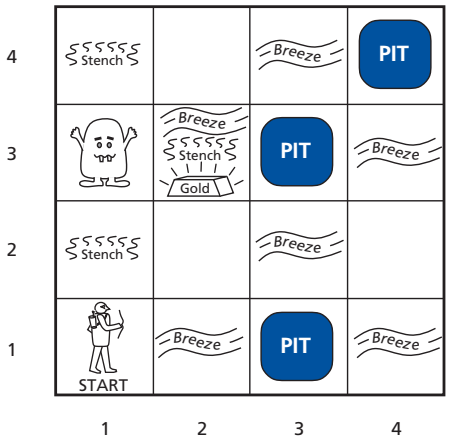


Abbildung 7.2: Eine typische Wumpus-Welt. Der Agent befindet sich unten links und sieht nach rechts.

Die Wumpus-Umgebung können wir dann zusammen mit den verschiedenen in *Kapitel 2* angegebenen Dimensionen charakterisieren. Zweifellos handelt es sich um einen diskreten, statischen und einzelnen Agenten. (Der Wumpus bewegt sich zum Glück nicht.) Er ist sequentiell, da Belohnungen erst eintreffen können, nachdem viele Aktionen unternommen wurden. Er ist partiell beobachtbar, da einige Aspekte des Zustandes nicht direkt wahrnehmbar sind: die Position des Agenten, der Gesundheitszustand des Wumpus und die Verfügbarkeit eines Pfeiles. Hinsichtlich der Positionen von Falltüren und Wumpus gilt: Wir können sie als nicht beobachtete Teile des Zustandes ansehen, der unveränderlich ist – in diesem Fall ist das Übergangsmodell für die Umgebung vollständig bekannt. Wir könnten auch sagen, dass das Übergangsmodell selbst unbekannt ist, da der Agent nicht weiß, welche *Forward*-Aktion fatal ist – in diesem Fall wird das Wissen im Übergangsmodell des Agenten durch das Entdecken der Felder mit Falltüren und dem Wumpus vervollständigt.

Die größte Schwierigkeit für den Agenten ist sein anfängliches Unwissen über die Umgebungskonfiguration. Für die Beseitigung dieses Unwissens scheint man ein logisches Schließen zu benötigen. In den meisten Instanzen der Wumpus-Welt ist es für den Agenten möglich, das Gold unverletzt zu finden. Manchmal muss sich der Agent entscheiden, ob er mit leeren Händen heimgeht oder aber den Tod riskiert, um das Gold zu finden. Etwa 21 Prozent der Umgebungen sind völlig unfair, weil sich das Gold über einer Falltür befindet oder von Falltüren umgeben ist.

Jetzt wollen wir beobachten, wie ein wissensbasierter Wumpus-Agent die in Abbildung 7.2 gezeigte Umgebung erkundet. Wir verwenden eine formlose Wissensrepräsentationssprache, die daraus besteht, die Symbole in einem Raster zu notieren (wie in ▶ Abbildung 7.3 und ▶ Abbildung 7.4 zu sehen).

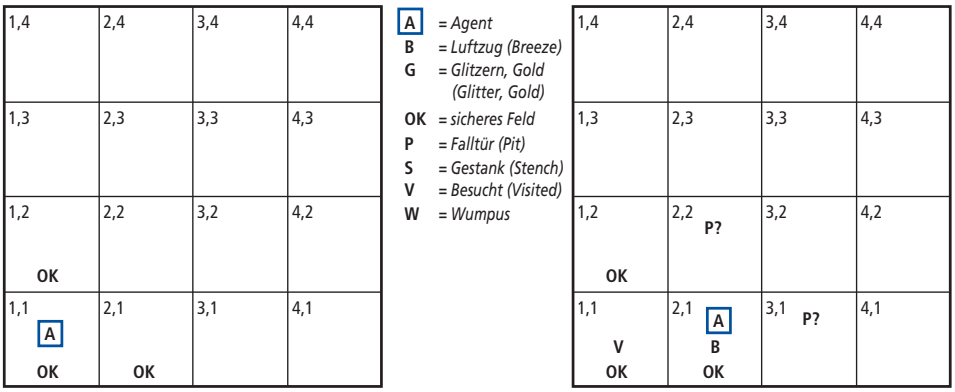


Abbildung 7.3: Der erste Schritt, den der Agent in der Wumpus-Welt unternimmt. (a) Die Ausgangssituation nach der Wahrnehmung [*None, None, None, None, None*]. (b) Nach einem Zug mit der Wahrnehmung [*None, Breeze, None, None, None*].

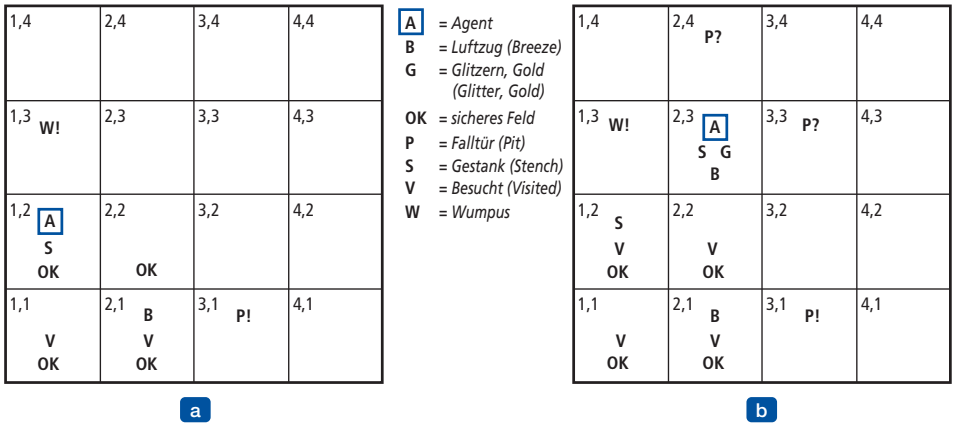


Abbildung 7.4: Zwei spätere Phasen im Fortschritt des Agenten. (a) Nach dem dritten Zug mit der Wahrnehmung [*Stench, None, None, None, None*]. (b) Nach dem fünften Zug mit der Wahrnehmung [*Stench, Breeze, Glitter, None, None*].

Die Wissensbasis des Agenten enthält anfänglich die Regeln der Umgebung, wie oben bereits beschrieben; insbesondere weiß er, dass er sich auf Feld [1,1] befindet und dass [1,1] ein sicheres Feld ist. Wir kennzeichnen dies mit einem „A“ bzw. „OK“ im Feld [1,1].

Die erste Wahrnehmung ist [*None, None, None, None, None*], aus der der Agent schließen kann, dass seine benachbarten Felder, [1,2] und [2,1], gefahrlos betreten werden können – sie sind OK. Abbildung 7.3(a) zeigt den Wissenszustand des Agenten an diesem Punkt.

Ein vorsichtiger Agent bewegt sich nur auf ein Feld, von dem er weiß, dass es *OK* ist. Angenommen, der Agent beschließt, auf das Feld [2,1] zu gehen. Der Agent nimmt auf [2,1] einen Luftzug wahr (mit „B“ für Breeze bezeichnet); es muss sich also eine Falltür in einem benachbarten Feld befinden. Da es gemäß den Spielregeln in [1,1] keine Falltür geben kann, muss sie sich in [2,2] oder [3,1] befinden. Die Notation „P?“ in Abbildung 7.3(b) weist auf eine mögliche Falltür in diesen Feldern hin. Zu diesem Zeitpunkt gibt es nur ein einziges Feld, von dem bekannt ist, dass es OK ist, und das noch nicht besucht wurde. Der vorsichtige Agent dreht sich also um, geht zurück nach [1,1] und setzt dann mit [1,2] fort.

In [1,2] nimmt der Agent einen Gestank wahr (*Stench*), was zu dem in Abbildung 7.4(a) gezeigten Wissenszustand führt. Der Gestank in [1,2] bedeutet, dass irgendwo in der Nähe ein Wumpus sein muss. Der Wumpus kann sich gemäß den Spielregeln jedoch nicht in [1,1] befinden und auch nicht in [2,2] (sonst hätte der Agent in [2,1] einen Gestank wahrgenommen). Daraus kann der Agent schließen, dass sich der Wumpus in [1,3] aufhält. Diese Schlussfolgerung wird mit der Notation „W!“ gekennzeichnet. Darüber hinaus bedeutet das Fehlen eines Luftzuges in [1,2], dass es in [2,2] keine Falltür gibt. Da der Agent bereits geschlossen hat, dass es in [2,2] oder [3,1] eine Falltür geben muss, bedeutet das, dass sie sich in [3,1] befindet. Das ist ein relativ schwieriger Schluss, weil er Wissen kombiniert, das zu unterschiedlichen Zeiten an unterschiedlichen Stellen ermittelt wurde und das auf dem Fehlen einer Wahrnehmung basiert, um einen wichtigen Schritt vorzunehmen.

Der Agent hat sich jetzt selbst bewiesen, dass es in [2,2] weder eine Falltür noch einen Wumpus gibt, es ist also OK, sich dorthin zu bewegen. Wir werden den Wissenszustand des Agenten an der Stelle [2,2] nicht zeigen; wir gehen einfach davon aus, dass

sich der Agent umdreht und nach [2,3] geht, womit wir Abbildung 7.4(b) erhalten. In [2,3] erkennt der Agent das Glitzern; er sollte sich also das Gold greifen und dann nach Hause zurückkehren.

In jedem Fall, wo der Agent einen Schluss aus verfügbarer Information zieht, ist dieser Schluss *garantiert* korrekt, wenn die verfügbare Information korrekt ist. Das ist eine grundlegende Eigenschaft logischen Schließens. Im Rest des Kapitels beschreiben wir, wie man logische Agenten erstellt, die die notwendigen Informationen repräsentieren und die Schlüsse ziehen können, die in den vorigen Absätzen beschrieben wurden.

7.3 Logik

Dieser Abschnitt bietet einen Überblick über alle grundlegenden Konzepte für logische Repräsentation und logisches Schließen. Die faszinierenden Konzepte sind unabhängig von den konkreten Formen der Logik. Wir verschieben deshalb die Beschreibung technischer Details der einzelnen Logikformen auf den nächsten Abschnitt und verwenden stattdessen bekannte Beispiele normaler Arithmetik.

In *Abschnitt 7.1* haben wir gesagt, dass Wissensbasen aus Sätzen bestehen. Diese Sätze werden gemäß der **Syntax** der Repräsentationssprache ausgedrückt, die alle Sätze darstellen kann, welche korrekt gebildet werden. Das Konzept der Syntax ist in der gewöhnlichen Arithmetik ausreichend klar: Bei „$x + y = 4$“ handelt es sich um einen korrekt gebildeten Satz, bei „$x4y+ =$ “ dagegen nicht.

Eine Logik muss auch die **Semantik** oder Bedeutung der Sätze definieren. Die Semantik der Sprache definiert die **Wahrheit** jedes Satzes im Hinblick auf jede **mögliche Welt**. Beispielsweise spezifiziert die Semantik für die Arithmetik, dass der Satz „$x + y = 4$“ wahr ist in einer Welt, in der x gleich 2 und y gleich 2 ist, aber falsch in einer Welt, in der x gleich 1 und y gleich 1 ist.[1] In der Standardlogik muss jeder Satz in jeder möglichen Welt entweder wahr oder falsch sein – es gibt kein „dazwischen“.

Wenn wir präzise sein müssen, verwenden wir den Begriff **Modell** anstelle von „mögliche Welt“. Während man sich mögliche Welten als (potenziell) reale Umgebungen vorstellen kann, in denen sich der Agent befindet oder nicht befindet, sind Modelle mathematische Abstraktionen, die einfach festlegen, ob ein relevanter Satz wahr oder falsch ist. Formlos können wir uns beispielsweise x und y als die Anzahl der Männer und Frauen vorstellen, die sich an einen Tisch setzen und Bridge spielen, und der Satz $x + y = 4$ ist wahr, wenn es insgesamt vier sind; formal ausgedrückt sind die möglichen Modelle einfach alle möglichen Zuweisungen von Zahlen an die Variablen x und y. Jede dieser Zuweisungen legt die Wahrheit jedes Satzes der Arithmetik fest, deren Variablen x und y sind. Wenn ein Satz α in Modell m wahr ist, sagen wir, m **erfüllt** α oder manchmal auch m **ist ein Modell von** α. Mit der Notation $M(\alpha)$ meinen wir die Menge aller Modelle von α.

Nachdem wir ein Konzept der Wahrheit gefunden haben, können wir über logisches Schließen sprechen. Das beinhaltet auch die **logische Konsequenz** (**Entailment**) zwischen den Sätzen – die Idee, dass ein Satz *logisch* aus einem anderen Satz *folgt*. In mathematischer Notation schreiben wir:

$$\alpha \models \beta.$$

1 Die in *Kapitel 14* beschriebene **Fuzzy-Logik** erlaubt Abstufungen der Wahrheit.

Das bedeutet, dass der Satz β eine logische Konsequenz von Satz α ist. Die formale Definition der logischen Konsequenz lautet: $\alpha \models \beta$ gilt genau dann, wenn in jedem Modell, in dem α wahr ist, auch β wahr ist. Mit dem eben eingeführten Konzept können wir schreiben:

$$\alpha \models \beta \text{ genau dann, wenn } M(\alpha) \subseteq M(\beta).$$

(Beachten Sie hier die Richtung des Zeichens $\subseteq$: Wenn $\alpha \models \beta$, dann ist α eine strengere Behauptung als β: Es schließt *mehr* mögliche Wörter aus.) Die Relation der logischen Konsequenz ist aus der Arithmetik bekannt; wir sind zufrieden mit dem Gedanken, dass der Satz $x = 0$ den Satz $xy = 0$ nach sich zieht. Offensichtlich ist es in jedem Modell, wo x null ist, der Fall, dass xy null ist (unabhängig vom Wert von y).

Dieselbe Analyse können wir auf das Schlussfolgerungsbeispiel aus der Wumpus-Welt des vorigen Abschnittes anwenden. Betrachten Sie die Situation in Abbildung 7.3(b): Der Agent hat in [1,1] nichts erkannt, aber einen Luftzug in [2,1] wahrgenommen. Diese Wahrnehmungen bilden zusammen mit dem Wissen des Agenten über die Regeln der Wumpus-Welt die Wissensbasis. Der Agent ist (unter anderem) daran interessiert, ob die benachbarten Felder [1,2], [2,2] und [3,1] Falltüren enthalten. Jedes der drei Felder kann eine Falltür enthalten; deshalb gibt es (für dieses Beispiel) $2^3 = 8$ mögliche Modelle. Sie sind in ▶ Abbildung 7.5 dargestellt.[2]

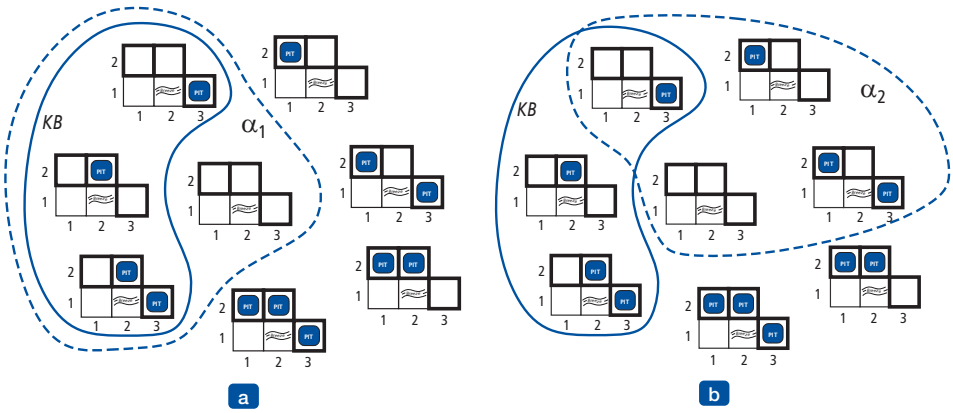


Abbildung 7.5: Mögliche Modelle für das Vorhandensein von Falltüren (Pit) in den Feldern [1,2], [2,2] und [3,1]. Die Wissensbasis (KB) entsprechend den Beobachtungen von Nichts in [1,1] und einem Luftzug in [2,1] wird von der durchgehenden Linie markiert. (a) Gestrichelte Linie zeigt Modelle von α_1 (keine Falltür in [1,2]). (b) Die gestrichelte Linie zeigt Modelle von α_2 (keine Falltür in [2,2]).

Die Wissensbasis kann man sich als Menge von Sätzen oder als einzelnen Satz vorstellen, der sämtliche individuellen Sätze behauptet. Die Wissensbasis ist falsch in Modellen, die dem widersprechen, was der Agent weiß – beispielsweise ist die Wissensbasis falsch in jedem Modell, in dem [1,2] eine Falltür enthält; weil es keinen Luftzug in [1,1] gibt. Praktisch gibt es drei Modelle, in denen die Wissensbasis wahr ist, diese sind in

2 Obwohl die Abbildung die Modelle als partielle Wumpus-Welten darstellt, handelt es sich dabei eigentlich um nichts weiter als Zuweisungen von *true* und *false* für die Sätze „Es gibt in [1,2] eine Falltür“ usw. Modelle im mathematischen Sinn brauchen diese grauenvollen behaarten Wumpus-Gestalten nicht.

Abbildung 7.5 von einer durchgehenden Linie umgeben. Jetzt betrachten wir zwei mögliche Schlussfolgerungen:

α_1 = „Es gibt keine Falltür in [1,2].“

α_2 = „Es gibt keine Falltür in [2,2].“

Wir haben die Modelle von α_1 und α_2 in Abbildung 7.5(a) und Abbildung 7.5(b) mit gestrichelten Linien markiert. Eine genaue Betrachtung zeigt:

In jedem Modell, in dem die Wissensbasis (*KB*) wahr ist, ist auch α_1 war.

Folglich gilt $KB \models \alpha_1$: Es gibt keine Falltür in [1,2]. Wir sehen auch:

In einigen Modellen, in denen die Wissensbasis (*KB*) wahr ist, ist α_2 falsch.

Folglich gilt KB $\not\models \alpha_2$: Der Agent *kann nicht* schließen, dass es in [2,2] keine Falltüren gibt. (Er kann aber auch nicht schließen, dass es in [2,2] eine Falltür gibt.)[3]

Das obige Beispiel verdeutlicht nicht nur die logische Konsequenz, sondern zeigt auch, wie ihre Definition angewendet werden kann, um Schlüsse abzuleiten – d.h. **logische Inferenzen** auszuführen. Der in Abbildung 7.5 gezeigte **Inferenzalgorithmus** wird auch als **Model Checking** (Modellprüfung) bezeichnet, weil er alle möglichen Modelle auflistet, um zu überprüfen, ob α in allen Modellen wahr ist, in denen *KB* wahr ist, d.h., dass $M(KB) \subseteq M(\alpha)$ gilt.

Um logische Konsequenz und Inferenz zu verstehen, kann es hilfreich sein, sich die Menge aller Konsequenzen von *KB* als Heuhaufen und α als Nadel vorzustellen. Logische Konsequenz ist vergleichbar mit der Tatsache, dass sich die Nadel im Heuhaufen befindet; die Inferenz ist wie der Prozess, sie zu finden. Diese Unterscheidung ist in einer formalen Notation verkörpert: Wenn ein Inferenzalgorithmus i in der Lage ist, α von *KB* abzuleiten, schreiben wir:

$KB \vdash_i \alpha$,

was gesprochen wird als „α wird durch i von *KB* abgeleitet“ oder „i leitet α von *KB* ab“.

Ein Inferenzalgorithmus, der nur folgerbare Sätze ableitet, wird als **zuverlässig** oder **wahrheitserhaltend** bezeichnet. Zuverlässigkeit ist eine äußerst wünschenswerte Eigenschaft. Eine unzuverlässige Inferenzprozedur erfindet bei ihrer Ausführung einfach irgendwelche Dinge – sie kündigt die Entdeckung nicht vorhandener Nadeln an. Man erkennt schnell, dass Model-Checking, dort, wo es angewendet werden kann,[4] ein gutes Verfahren ist.

Auch die Eigenschaft der **Vollständigkeit** ist wünschenswert: Ein Inferenzalgorithmus ist vollständig, wenn er jeden folgerbaren Satz ableiten kann. Für reale Heuhaufen, die eine endliche Größe aufweisen, scheint es offensichtlich, dass eine systematische Untersuchung immer feststellen kann, ob sich eine Nadel darin befindet. Für viele Wissensbasen ist jedoch der Heuhaufen der Konsequenzen unendlich groß und die Vollständigkeit wird zu einem wichtigen Aspekt.[5] Glücklicherweise gibt es vollständige Inferenzprozeduren für Logiken, die hinreichend ausdrucksstark sind, um viele Wissensbasen abzudecken.

3 Der Agent kann die *Wahrscheinlichkeit* berechnen, dass es in [2,2] eine Falltür gibt; *Kapitel 13* zeigt, wie das geht.

4 Das Model-Checking funktioniert, wenn der Modellraum endlich ist – beispielsweise in Wumpus-Welten fester Größe. Für die Arithmetik dagegen ist der Modellraum unendlich: Selbst wenn wir uns auf die ganzen Zahlen beschränken, gibt es unendlich viele Wertepaare für x und y im Satz $x + y = 4$.

5 Vergleichen Sie dies mit dem Fall der unendlichen Suchräume in *Kapitel 3*, wo die Tiefensuche nicht vollständig ist.

Tipp

Wir haben einen Schlussfolgerungsprozess beschrieben, dessen Schlussfolgerungen in allen Welten, in denen die Vorgaben wahr sind, garantiert wahr sind; insbesondere gilt: *Wenn KB in der realen Welt wahr ist, dann ist auch jeder durch eine stichhaltige Inferenzprozedur von KB abgeleitete Satz α in der realen Welt wahr.* Während also ein Inferenzprozess mit der „Syntax" arbeitet – mit internen physischen Konfigurationen, wie etwa Bits in Registern oder Mustern elektrischer Blips in Gehirnen –, *entspricht* der Prozess der Beziehung in der realen Welt, dass ein Aspekt der realen Welt aufgrund anderer Aspekte der realen Welt der Fall ist.[6] Diese Entsprechung zwischen der Welt und ihrer Repräsentation ist in ▶ Abbildung 7.6 gezeigt.

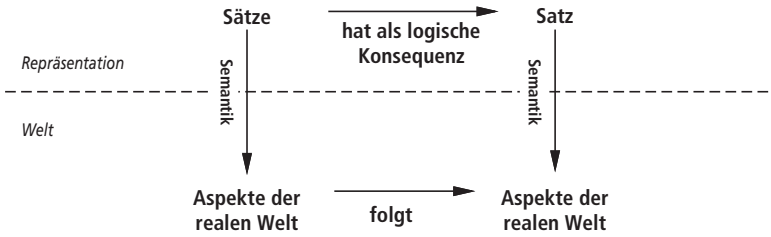


Abbildung 7.6: Sätze sind physische Konfigurationen des Agenten. Das Schließen ist ein Prozess, neue physische Konfigurationen aus alten zu erzeugen. Logisches Schließen sollte sicherstellen, dass die neuen Konfigurationen Aspekte der Welt repräsentieren, die aus den Aspekten folgen, welche von den alten Konfigurationen repräsentiert werden.

Tipp

Der letzte Aspekt, der von einem logischen Agenten berücksichtigt werden muss, ist die **Fundierung** – die Verbindung zwischen logischen Schlussfolgerungsprozessen und der realen Umgebung, in der der Agent existiert. Insbesondere: *Wie wissen wir, dass KB in der realen Welt wahr ist*? (Schließlich ist *KB* nur „Syntax" im Kopf des Agenten.) Das ist eine philosophische Frage, über die sehr viele Bücher geschrieben wurden (siehe *Kapitel 26*). Eine einfache Antwort ist, dass die Sensoren des Agenten die Verbindung herstellen. Zum Beispiel hat unser Agent in der Wumpus-Welt einen Geruchssensor. Das Agentenprogramm erzeugt einen geeigneten Satz, sobald ein Geruch wahrzunehmen ist. Wenn sich dann dieser Satz in der Wissensbasis befindet, ist er in der realen Welt wahr. Somit sind Bedeutung und Wahrheit von Wahrnehmungssätzen durch die Erfassungsprozesse und durch die Satzkonstruktion, die sie erzeugt, definiert. Was ist mit dem restlichen Wissen des Agenten, wie etwa seinem Glauben, dass Wumpi in benachbarten Quadraten Gerüche ausströmen? Das ist keine direkte Repräsentation einer einzelnen Wahrnehmung, sondern eine allgemeine Regel – abgeleitet vielleicht aus Erfahrung durch Wahrnehmung, aber nicht identisch mit einer Aussage zu dieser Erfahrung. Allgemeine Regeln wie diese werden durch einen Satzkonstruktionsprozess erzeugt, den wir als **Lernen** bezeichnen – Thema von Teil V. Lernen ist fehlbar. Es könnte sein, dass Wumpi stinken, *außer am 29. Februar in Schaltjahren*, nämlich an ihrem Badetag. Somit ist *KB* möglicherweise in der realen Welt nicht wahr, aber mit einer guten Lernprozedur kann man durchaus optimistisch sein.

6 Wie Wittgenstein (1922) in seinem berühmten *Tractatus logico-philosophicus* sagt: „Die Welt ist alles, was der Fall ist."

7.4 Aussagenlogik: eine sehr einfache Logik

Jetzt stellen wir eine sehr einfache Logik vor, die sogenannte **Aussagenlogik**. Wir beschäftigen uns mit der Syntax der Aussagenlogik und mit ihrer Semantik – wie die Wahrheit von Sätzen ermittelt wird. Anschließend betrachten wir die **logische Konsequenz** – die Beziehung zwischen einem Satz und einem anderen Satz, der daraus folgt – und beobachten, wie dies zu einem einfachen Algorithmus für logische Inferenz führt. Das alles findet natürlich in der Wumpus-Welt statt.

7.4.1 Syntax

Die **Syntax** der Aussagenlogik definiert die erlaubten Sätze. Die **atomaren Sätze** bestehen aus einem einzigen **Aussagensymbol**. Jedes dieser Symbole steht für eine Aussage, die wahr oder falsch sein kann. Wir verwenden Symbole, die mit einem Großbuchstaben beginnen und weitere Buchstaben oder Indizes enthalten können, zum Beispiel P, Q, R, $W_{1,3}$ und *North*. Die Namen sind willkürlich gewählt, können aber manchmal einen mnemonischen Hinweis für den Leser enthalten – so schreiben wir $W_{1,3}$ für die Aussage, dass sich der Wumpus in [1,3] befindet. (Beachten Sie, dass Symbole wie etwa $W_{1,3}$ *atomar* sind, d.h., W, 1 und 3 haben als Bestandteile des Symbols keine Bedeutung.) Es gibt zwei Aussagensymbole mit fester Bedeutung: *True* ist die immer wahre Aussage und *False* ist die immer falsche Aussage.

Komplexe Sätze werden unter Verwendung **logischer Verknüpfungen** aus einfacheren Sätzen aufgebaut. Es gibt fünf gebräuchliche Verknüpfungen:

- $\neg$ (Nicht): Ein Satz wie etwa $\neg W_{1,3}$ wird als **Negation** von $W_{1,3}$ bezeichnet. Ein **Literal** ist entweder ein atomarer Satz (ein **positives Literal**) oder ein negierter atomarer Satz (ein **negatives Literal**).
- $\wedge$ (Und): Ein Satz, dessen Hauptverknüpfung $\wedge$ ist (wie etwa $W_{1,3} \wedge P_{3,1}$), wird als **Konjunktion** bezeichnet; seine Bestandteile sind **Konjunkte**.
- $\vee$ (Oder): Ein Satz, der $\vee$ verwendet (wie etwa $(W_{1,3} \wedge P_{3,1}) \vee W_{2,2}$), ist eine **Disjunktion** bestehend aus den **Disjunkten** $(W_{1,3} \wedge P_{3,1})$ und $W_{2,2}$. (Historisch kommt das $\vee$ aus dem Lateinischen, „vel“, das bedeutet „oder“. Die meisten Menschen merken es sich lieber als umgekehrtes $\wedge$.)
- $\Rightarrow$ (impliziert): Ein Satz wie etwa $(W_{1,3} \wedge P_{3,1}) \Rightarrow \neg W_{2,z2}$ wird als **Implikation** (oder Konditional) bezeichnet. Seine **Prämisse** oder **Antezedenz** ist $(W_{1,3} \wedge P_{3,1})$ und seine **Schlussfolgerung** oder **Konsequenz** ist $\neg W_{2,2}$. Implikationen werden auch als **Regeln** oder **if-then-Aussagen** (Wenn-Dann-Aussagen) bezeichnet. Das Symbol für die Implikation wird in anderen Büchern manchmal als $\supset$ oder $\rightarrow$ dargestellt.
- $\Leftrightarrow$ (Genau dann, wenn): Der Satz $W_{1,3} \Leftrightarrow \neg W_{2,2}$ ist ein **Bikonditional** (oder eine Bi-Implikation). In anderen Büchern wird auch das Symbol $\equiv$ verwendet.

▶ Abbildung 7.7 gibt eine formale Grammatik der Aussagenlogik an; falls Sie nicht mit der BNF-Notation vertraut sind, können Sie sich in *Anhang B* informieren. Die BNF-Grammatik ist an sich mehrdeutig; ein Satz mit mehreren Operatoren lässt sich durch die Grammatik auf mehreren Wegen analysieren (parsen). Um die Mehrdeutigkeit zu beseitigen, definieren wir für jeden Operator eine Rangfolge. Der *Nicht*-Operator ($\neg$) hat den höchsten Vorrang. Das heißt, das im Satz $\neg A \wedge B$ das Negationszeichen $\neg$ am engsten bindet, was uns das Äquivalent von $(\neg A) \wedge B$ und nicht von $\neg(A \wedge B)$ liefert. (Das Konzept bei gewöhnlicher Arithmetik ist das gleiche: $-2 + 4$ ist 2 und nicht -6.) Im

Zweifelsfall sollten Sie mithilfe von Klammern die richtige Interpretation sicherstellen. Eckige Klammern bedeuten das Gleiche wie runde Klammern; der jeweils gewählte Klammerntyp soll allein eine möglichst gute Lesbarkeit des Satzes gewährleisten.

$$
\begin{aligned}
\mathit{Satz} &\rightarrow \mathit{AtomarerSatz} \mid \mathit{KomplexerSatz} \\
\mathit{AtomarerSatz} &\rightarrow \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \ \ldots \\
\mathit{KomplexerSatz} &\rightarrow (\ \mathit{Satz}\ ) \mid [\ \mathit{Satz}\ ] \\
&\mid \neg\ \mathit{Satz} \\
&\mid (\mathit{Satz}\ \wedge\ \mathit{Satz}) \\
&\mid (\mathit{Satz}\ \vee\ \mathit{Satz}) \\
&\mid (\mathit{Satz}\ \Rightarrow\ \mathit{Satz}) \\
&\mid (\mathit{Satz}\ \Leftrightarrow\ \mathit{Satz})
\end{aligned}
$$

Operatorrangfolge: $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$

Abbildung 7.7: Eine BNF-Grammatik (Backus-Naur-Form) von Sätzen der Aussagenlogik. Außerdem ist die Operatorrangfolge vom höchsten zum niedrigsten angegeben.

7.4.2 Semantik

Nachdem wir im letzten Abschnitt die Syntax der Aussagenlogik beschrieben haben, geht es nun um ihre Semantik. Die Semantik definiert die Regeln, wie die Wahrheit eines Satzes im Hinblick auf ein bestimmtes Modell ermittelt wird. In der Aussagenlogik legt ein Modell einfach die **Wahrheitswerte** – *true* oder *false* – für jedes Aussagensymbol fest. Verwenden die Sätze in der Wissensbasis beispielsweise die Aussagensymbole $P_{1,2}$, $P_{2,2}$ und $P_{3,1}$, dann ist ein mögliches Modell:

$$m_1 = \{\ P_{1,2} = \mathit{false}, P_{2,2} = \mathit{false}, P_{3,1} = \mathit{true}\ \}.$$

Bei drei Aussagensymbolen gibt es $2^3 = 8$ mögliche Modelle – genau die in Abbildung 7.5 dargestellten. Beachten Sie aber, dass die Modelle reine mathematische Objekte ohne zwingende Verbindung zu Wumpus-Welten sind. $P_{1,2}$ ist nur ein Symbol; es könnte bedeuten: „Es befindet sich eine Falltür auf Feld [1,2]“, aber auch: „Ich bin heute und morgen in Paris.“

Die Semantik für die Aussagenlogik muss angeben, wie der Wahrheitswert *beliebiger* Sätze für ein vorgegebenes Modell berechnet werden kann. Das erfolgt rekursiv. Alle Sätze werden aus atomaren Sätzen und den fünf Verknüpfungen gebildet; aus diesem Grund müssen wir angeben, wie die Wahrheit atomarer Sätze und wie die Wahrheit von Sätzen, die mit den verschiedenen Verknüpfungen erzeugt werden, berechnet wird. Für atomare Sätze ist es einfach:

- *true* ist in jedem Modell wahr und *false* ist in jedem Modell falsch.
- Der Wahrheitswert jedes anderen Aussagensymbols muss direkt im Modell angegeben werden. Im zuvor angegebenen Modell m_1 beispielsweise ist $P_{1,2}$ falsch.

Für komplexe Sätze haben wir fünf Regeln, die für beliebige Teilsätze P und Q in jedem Modell m gelten:

- $\neg P$ ist true genau dann, wenn P in m false ist.
- $P \wedge Q$ ist true genau dann, wenn P und Q in m true sind.
- $P \vee Q$ ist true genau dann, wenn entweder P oder Q in m true ist.
- $P \Rightarrow Q$ ist true, außer wenn P true und Q false in m ist.
- $P \Leftrightarrow Q$ ist true genau dann, wenn P und Q in m beide true oder beide false sind.

Die Regeln lassen sich auch mit **Wahrheitstabellen** ausdrücken, die den Wahrheitswert eines komplexen Satzes für jede mögliche Zuweisung von Wahrheitswerten für seine Komponenten angeben. ▶ Abbildung 7.8 zeigt Wahrheitstabellen für die fünf logischen Verknüpfungen. Mithilfe dieser Tabellen kann der Wahrheitswert jedes Satzes im Hinblick auf ein Modell m durch eine einfache rekursive Auswertung berechnet werden. Beispielsweise ergibt der Satz $\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1})$ ausgewertet in m_1 den Wahrheitswert $true \wedge (false \vee true) = true \wedge true = true$. In Übung 7.3 werden Sie den Algorithmus PL-TRUE?(s,m) schreiben, der den Wahrheitswert eines aussagenlogischen Satzes s in einem Modell m berechnet.

P	Q	$\neg$P	P $\wedge$ Q	P $\vee$ Q	P $\Rightarrow$ Q	P $\Leftrightarrow$ Q
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Abbildung 7.8: Wahrheitstabellen für die fünf logischen Verknüpfungen. Um anhand der Tabelle zum Beispiel den Wert von $P \vee Q$ zu berechnen, wenn P true und Q false ist, sehen Sie zuerst ganz links in den beiden ersten Spalten nach, wo P true und Q false ist (es ist die dritte Zeile). Anschließend sehen Sie in der Zeile unter der Spalte $P \vee Q$ nach, um das Ergebnis zu finden: true.

Die Wahrheitstabellen für „und“, „oder“ und „nicht“ sind eng mit unserem Verständnis der natürlichen Sprache verwandt. Eine mögliche Verwechslung besteht darin, dass $P \vee Q$ das Ergebnis *true* liefert, wenn P oder Q oder beide *true* sind. Es gibt noch eine andere Verknüpfung, „exklusiv oder“ (kurz „xor“), die *false* ergibt, wenn beide Disjunkte *true* sind.[7] Es gibt keinen Konsens im Hinblick auf das Symbol für xor; so verwendet man $\dot{\vee}$ oder $\neq$ und $\oplus$.

Die Wahrheitstabelle für $\Rightarrow$ erscheint Ihnen auf den ersten Blick vielleicht etwas rätselhaft, weil sie nicht in unser intuitives Verständnis von „P impliziert Q“ oder „wenn P, dann Q“ passt. Zum einen fordert die Aussagenlogik keinerlei Relation von *Kausalität* oder *Relevanz* zwischen P und Q. Der Satz „5 ist ungerade impliziert Tokio ist die Hauptstadt von Japan“ ist ein wahrer Satz der Aussagenlogik (bei normaler Interpretation), selbst wenn sich das für unser normales Sprachverständnis seltsam anhört. Eine weitere Quelle der Verwirrung ist, dass jede Implikation wahr ist, wenn ihre Antezedenz falsch ist. Beispielsweise ist „5 ist gerade impliziert Sam ist intelligent“ wahr, unabhängig davon, ob Sam intelligent ist oder nicht. Das erscheint abstrus, aber es wird sinnvoll, wenn Sie sich vorstellen, „$P \Rightarrow Q$“ als „Wenn P wahr ist, behaupte ich, dass Q wahr ist. Andernfalls mache ich keine Behauptung“ auszudrücken. Die einzige Möglichkeit, wie dieser Satz *false* werden kann, ist, wenn P *true*, aber Q *false* ist.

Die Wahrheitstabelle für ein Bikonditional, $P \Leftrightarrow Q$, zeigt, dass es *true* ist, wenn sowohl $P \Rightarrow Q$ als auch $Q \Rightarrow P$ *true* sind. In natürlicher Sprache sagt man dafür häufig: „P nur genau dann, wenn Q“ (oder im Englischen „P if and only if Q“, abgekürzt „P iff Q“). Die meisten Regeln der Wumpus-Welt werden am besten mithilfe von $\Leftrightarrow$ angegeben. Ein Feld weist beispielsweise dann einen Luftzug auf, *wenn* ein benachbartes Feld eine Falltür besitzt, und ein Feld hat *nur dann* einen Luftzug, *wenn* ein benachbartes Feld eine Falltür besitzt. Wir brauchen also ein Bikonditional wie etwa:

7 Im Lateinischen gibt es ein separates Wort für „exklusiv oder“, *aut*.

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}),$$

wobei $B_{1,1}$ bedeutet, dass auf [1,1] ein Luftzug vorhanden ist.

7.4.3 Eine einfache Wissensbasis

Nachdem wir die Semantik für die Aussagenlogik definiert haben, können wir eine Wissensbasis für die Wumpus-Welt konstruieren. Wir konzentrieren uns hier auf die *unveränderlichen* Aspekte der Wumpus-Welt und verschieben die veränderlichen Aspekte auf einen späteren Abschnitt. Momentan benötigen wir die folgenden Symbole für jede [x, y]-Position:

- $P_{x,y}$ ist true, wenn sich in [x, y] eine Falltür befindet.
- $W_{x,y}$ ist true, wenn sich in [x, y] ein Wumpus (tot oder lebend) befindet.
- $B_{x,y}$ ist true, wenn der Agent in [x, y] einen Luftzug wahrnimmt.
- $S_{x,y}$ ist true, wenn der Agent in [x, y] einen Gestank wahrnimmt.

Wir schreiben Sätze, die genügen, um $\neg P_{1,2}$ abzuleiten (es gibt keine Falltür in [1, 2]), wie es in *Abschnitt 7.3* formlos angegeben wurde. Die Sätze bezeichnen wir mit R_i, damit wir auf sie verweisen können:

- Es gibt keine Falltür in [1,1]:

  R_1: $\neg P_{1,1}$.

- Ein Feld hat genau dann einen Luftzug, wenn sich auf einem benachbarten Feld eine Falltür befindet. Das muss für jedes Feld angegeben werden; fürs Erste binden wir nur die relevanten Felder ein:

  R_2: $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$.

  R_3: $B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,1} \vee P_{3,1})$.

- Die obigen Sätze sind in allen Wumpus-Welten wahr. Jetzt nehmen wir die Luftzugwahrnehmungen für die beiden ersten in der Welt des Agenten besuchten Felder auf, was zu der in Abbildung 7.3(b) gezeigten Situation führt:

  R_4: $\neg B_{1,1}$.

  R_5: $B_{2,1}$.

7.4.4 Eine einfache Inferenzprozedur

Unser Ziel besteht nun darin, zu entscheiden, ob $KB \models \alpha$ für einen bestimmten Satz α gilt. Ist beispielsweise $\neg P_{2,2}$ eine Konsequenz, die sich aus unserer *KB* ergibt? Unser erster Algorithmus für Inferenz ist ein Modellüberprüfungsansatz, der eine direkte Implementierung der Definition logischer Konsequenz ist: Er listet die Modelle auf und überprüft, ob α in jedem Modell wahr ist, in dem *KB* wahr ist. Bei der Aussagenlogik sind Modelle Zuweisungen von *true* oder *false* für jedes Aussagensymbol. Betrachten wir wieder unser Beispiel der Wumpus-Welt. Die relevanten Aussagensymbole sind hier $B_{1,1}$, $B_{2,1}$, $P_{1,1}$, $P_{1,2}$, $P_{2,1}$, $P_{2,2}$ und $P_{3,1}$. Bei sieben Symbolen gibt es $2^7 = 128$ mögliche Modelle; in drei davon ist *KB* wahr (▶ Abbildung 7.9). In diesen drei Modellen ist $\neg P_{1,2}$ wahr, weil sich in [1,2] keine Falltür befindet. Andererseits ist $P_{2,2}$ in zwei von drei Modellen wahr und in einem falsch; wir können also nicht erkennen, ob sich in [2,2] eine Falltür befindet.

Abbildung 7.9 zeigt eine exaktere Form des Schließens, das in Abbildung 7.5 illustriert wurde.

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*false*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

Abbildung 7.9: Eine Wahrheitstabelle für die im Text beschriebene Wissensbasis. KB ist wahr, wenn R_1 bis R_5 wahr sind, was in nur drei der 128 Zeilen der Fall ist (unterstrichene Werte in der rechten Spalte). In allen drei Zeilen ist $P_{1,2}$ falsch, es befindet sich also keine Falltür in [1,2]. Andererseits könnte es eine Falltür in [2,2] geben, was aber nicht der Fall sein muss.

▶ Abbildung 7.10 zeigt einen allgemeinen Algorithmus, der über aussagenlogische Konsequenz entscheidet. Wie der BACKTRACKING-SEARCH-Algorithmus in *Abschnitt 6.3* erzeugt TT-ENTAILS? eine rekursive Auflistung eines endlichen Zuweisungsraumes für Variablen. Der Algorithmus ist **korrekt**, weil er direkt die Definition der logischen Konsequenz implementiert, und **vollständig**, weil er für jedes *KB* und jedes α funktioniert und immer terminiert – es gibt nur endlich viele Modelle, die ausgewertet werden müssen.

```
function TT-Entails?(KB, α) returns true oder false
   inputs: KB, die Wissensbasis, ein Satz in Aussagenlogik
           α, die Abfrage, ein Satz in Aussagenlogik

   symbols ← eine Liste der Aussagensymbole in KB und α
   return TT-Check-All(KB, α, symbols, {})
```

```
function TT-Check-All(KB, α, symbols, model) returns true oder false
   if Empty?(symbols) then
      if PL-True?(KB, model) then return PL-True?(α, model)
      else return true // wenn KB false ist, immer true zurückgeben
   else do
      P ← First(symbols)
      rest ← Rest(symbols)
      return (TT-Check-All(KB, α, rest, model ∪ {P = true})
              and
              TT-Check-All(KB, α, rest, model ∪ {P = false}))
```

Abbildung 7.10: Ein Auflistungsalgorithmus für die Wahrheitstabelle, um über aussagenlogische Konsequenz zu entscheiden. TT steht für Truth Table (Wahrheitstabelle). PL-TRUE? gibt *true* zurück, wenn ein Satz innerhalb eines Modells gilt. Die Variable *model* stellt ein partielles Modell dar – eine Zuweisung an nur einige der Variablen. Das Schlüsselwort *and* wird hier als logische Operation mit den beiden Argumenten verwendet und gibt entweder *true* oder *false* zurück.

Tipp

Natürlich bedeutet „endlich viele“ nicht immer dasselbe wie „wenige“. Wenn *KB* und α insgesamt n Symbole enthalten, gibt es 2^n Modelle. Die Zeitkomplexität des Algorithmus ist also $O(2^n)$. (Die Speicherkomplexität ist nur $O(n)$, weil die Aufzählung eine Tiefensuche verwendet.) Später in diesem Kapitel sehen wir Algorithmen, die in der Praxis sehr viel effizienter sind. Leider ist jede logische Konsequenz der Aussagenlogik co-NO-vollständig (d.h. wahrscheinlich nicht einfacher als NP-vollständig – siehe Anhang A), *sodass jeder bekannte Inferenzalgorithmus für die Aussagenlogik im ungünstigsten Fall eine Komplexität aufweist, die exponentiell zur Eingabegröße ist.*

$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$	Kommutativität von $\wedge$
$(\alpha \vee \beta) \equiv (\beta \vee \alpha)$	Kommutativität von $\vee$
$((\alpha \wedge \beta) \vee \gamma \equiv (\alpha \wedge (\beta \vee \gamma))$	Assoziativität von $\wedge$
$((\alpha \vee \beta) \vee \gamma \equiv (\alpha \vee (\beta \vee \gamma))$	Assoziativität von $\vee$
$\neg(\neg\alpha) \equiv \alpha$	Eliminierung der doppelten Negation
$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha)$	Kontraposition
$(\alpha \Rightarrow \beta) \equiv (\neg \alpha \vee \beta)$	Implikationseliminierung
$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$	Bikonditionalelimierung
$\neg(\alpha \wedge \beta) \equiv (\neg \alpha \vee \neg \beta)$	de Morgan
$\neg(\alpha \vee \beta) \equiv (\neg \alpha \wedge \neg \beta)$	de Morgan
$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	Distributivität von $\wedge$ über $\vee$
$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	Distributivität von $\vee$ über $\wedge$

Abbildung 7.11: Logische **Äquivalenzen**. Die Symbole α, β und γ stehen für beliebige Sätze der Aussagenlogik.

7.5 Theoreme der Aussagenlogik beweisen

Bislang haben wir gezeigt, wie logische Konsequenz durch *Modellprüfung* zu ermitteln ist: Modelle aufzählen und zeigen, dass der Satz in allen Modellen gelten muss. In diesem Abschnitt zeigen wir nun, wie sich logische Konsequenz durch **Theorembeweisen** realisieren lässt – Regeln der Inferenz direkt auf Sätze in unserer Wissensbasis anwenden, um einen Beweis des gewünschten Satzes zu konstruieren, ohne Modelle zu konsultieren. Ist die Anzahl der Modelle groß, die Länge des Beweises jedoch kurz, kann Theorembeweisen effizienter als Modellüberprüfung sein.

Bevor wir uns mit den Details von Algorithmen zum Theorembeweisen beschäftigen, brauchen wir neben der logischen Konsequenz noch ein paar zusätzliche Konzepte. Das erste Konzept ist die **logische Äquivalenz**: Zwei Sätze α und β sind logisch äquivalent, wenn sie in derselben Modellmenge wahr sind. Wir schreiben dies als $\alpha \equiv \beta$. Beispielsweise kann man (unter Verwendung von Wahrheitstabellen) einfach zeigen, dass $P \wedge Q$ und $Q \wedge P$ logisch äquivalent sind; andere Äquivalenzen sind in ▶ Abbildung 7.11 gezeigt. Sie spielen in der Logik etwa dieselbe Rolle wie arithmetische Identitäten in der normalen Mathematik. Eine alternative Definition der Äquivalenz lautet wie folgt: Zwei beliebige Sätze α und β sind äquivalent genau dann, wenn jeder eine logische Konsequenz des anderen ist:

$$\alpha \equiv \beta \text{ genau dann, wenn } \alpha \models \beta \text{ und } \beta \models \alpha.$$

Das zweite Konzept, das wir brauchen, ist die **Gültigkeit**. Ein Satz ist gültig, wenn er in *allen* Modellen wahr ist. Beispielsweise ist der Satz $P \vee \neg P$ aussagenlogisch gültig. Gültige Sätze werden auch als **Tautologien** bezeichnet – sie sind *unbedingt* wahr. Weil der Satz *True* in allen Modellen wahr ist, ist jeder gültige Satz logisch äquivalent mit *True*. Wozu braucht man gültige Sätze? Aus unserer Definition der logischen Konsequenz können wir das **Deduktionstheorem** ableiten, das schon den alten Griechen bekannt war:

Tipp

Für alle Sätze α und β gilt $\alpha \models \beta$ genau dann, wenn der Satz $(\alpha \Rightarrow \beta)$ gültig ist.

(In Übung 7.5 werden Sie dies beweisen.) Folglich können wir entscheiden, ob $\alpha \models \beta$ ist, indem wir überprüfen, ob $(\alpha \Rightarrow \beta)$ in jedem Modell gültig ist – was letztlich der Inferenzalgorithmus in Abbildung 7.10 realisiert –, oder indem wir beweisen, dass $(\alpha \Rightarrow \beta)$ äquivalent mit *True* ist. Umgekehrt besagt das Deduktionstheorem, dass jeder gültige Implikationssatz eine legitime Inferenz beschreibt.

Das letzte Konzept, das wir brauchen, ist die **Erfüllbarkeit**. Ein Satz ist erfüllbar, wenn er in *irgendeinem* Modell wahr ist. Beispielsweise ist die zuvor gezeigte Wissensbasis $(R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5)$ erfüllbar, weil es drei Modelle gibt, in denen sie wahr ist, wie in Abbildung 7.9 gezeigt. Die Erfüllbarkeit kann überprüft werden, indem die möglichen Modelle aufgelistet werden, bis eines gefunden ist, das den Satz erfüllt. Der Test der Erfüllbarkeit von Sätzen in der Aussagenlogik – das sogenannte **SAT**-Problem (Satisfiability) – war das erste Problem, das als NP-vollständig nachgewiesen wurde. Beispielsweise fragen alle CSPs in *Kapitel 6* letztlich, ob die Randbedingungen durch irgendeine Zuweisung erfüllbar sind.

Gültigkeit und Erfüllbarkeit sind natürlich verwandt: α ist gültig genau dann, wenn $\neg\alpha$ nicht erfüllbar ist. Im Umkehrschluss ist α genau dann erfüllbar, wenn $\neg\alpha$ nicht gültig ist. Wir haben auch das folgende praktische Ergebnis:

Tipp

$\alpha \models \beta$ genau dann, wenn der Satz $(\alpha \wedge \neg\beta)$ nicht erfüllbar ist.

Beweist man β aus α, indem man die Nichterfüllbarkeit von $(\alpha \wedge \neg\beta)$ überprüft, entspricht das genau der mathematischen Standardbeweistechnik der *reductio ad absurdum* (zu Deutsch „Reduzierung auf etwas Absurdes“). Man spricht auch vom Beweis durch **Widerlegung** oder Beweis durch **Widerspruch**. Man geht davon aus, dass ein Satz β falsch ist, und zeigt, dass das zu einem Widerspruch mit bekannten Axiomen α führt. Dieser Widerspruch entspricht genau der Aussage, dass der Satz $(\alpha \wedge \neg\beta)$ nicht erfüllbar ist.

7.5.1 Inferenz und Beweise

Dieser Abschnitt beschreibt **Inferenzregeln**, die sich anwenden lassen, um einen Beweis herzuleiten – eine Kette von Schlussfolgerungen, die zum gewünschten Ziel führt. Die bekannteste Regel ist der sogenannte **Modus ponens** und wird wie folgt geschrieben:

$$\frac{\alpha \Rightarrow \beta, \alpha}{\beta}.$$

Die Notation bedeutet, dass auf den Satz β geschlossen werden kann, wenn Sätze der Form $\alpha \Rightarrow \beta$ und α gegeben sind. Sind beispielsweise $(\mathit{WumpusAhead} \wedge \mathit{WumpusAlive}) \Rightarrow \mathit{Shoot}$ und $(\mathit{WumpusAhead} \wedge \mathit{WumpusAlive})$ gegeben, kann *Shoot* geschlossen werden.

Eine weitere praktische Inferenzregel ist die **Und-Eliminierung**, die besagt, dass aus einer Konjunktion jedes der Konjunkte geschlossen werden kann:

$$\frac{\alpha \wedge \beta}{\alpha}.$$

Aus (*WumpusAhead* $\wedge$ *WumpusAlive*) kann *WumpusAlive* geschlossen werden.

Durch Betrachtung der möglichen Wahrheitswerte von α und β kann man einfach zeigen, dass Modus ponens und Und-Eliminierung ein für alle Mal korrekt sind. Diese Regeln können dann in jeder beliebigen Instanz verwendet werden und korrekte Inferenzen erzeugen, ohne Modelle auflisten zu müssen.

Alle logischen Äquivalenzen aus Abbildung 7.11 können als Inferenzregeln verwendet werden. Beispielsweise führt die Äquivalenz für die Bikonditional-Eliminierung zu den beiden folgenden Inferenzregeln:

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad \text{und} \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}.$$

Nicht alle Inferenzregeln funktionieren in beide Richtungen wie diese. Beispielsweise können wir Modus ponens nicht in der umgekehrten Richtung ausführen, um $\alpha \Rightarrow \beta$ und α aus β zu erhalten.

Jetzt wollen wir zeigen, wie diese Inferenzregeln und Äquivalenzen in der Wumpus-Welt verwendet werden können. Wir beginnen mit der Wissensbasis, die R_1 bis R_5 enthält, und zeigen, wie wir $\neg P_{1,2}$ beweisen, d.h., dass sich keine Falltür in [1,2] befindet. Zuerst wenden wir die Bikonditional-Eliminierung auf R_2 an und erhalten:

$$R_6\colon (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}).$$

Dann wenden wir die Und-Eliminierung auf R_6 an und erhalten:

$$R_7\colon ((P_{1,2} \vee P_{2,1})) \Rightarrow B_{1,1}).$$

Die logische Äquivalenz für Kontraposition ergibt:

$$R_8\colon (\neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1})).$$

Jetzt können wir Modus ponens auf R_8 und die Wahrnehmung R_4 (d.h. $\neg B_{1,1}$) anwenden und erhalten:

$$R_9\colon \neg(P_{1,2} \vee P_{2,1}).$$

Schließlich wenden wir noch die Regel von de Morgan an und erhalten den Schluss:

$$R_{10}\colon \neg P_{1,2} \wedge \neg P_{2,1}.$$

Das bedeutet, weder [1,2] noch [2,1] enthalten eine Falltür.

Diesen Beweis haben wir manuell gefunden, doch wir können jeden der Suchalgorithmen von *Kapitel 3* anwenden, um eine Schrittfolge zu suchen, die einen Beweis bildet. Wir müssen lediglich ein Beweisproblem wie folgt definieren:

- **Anfangszustand:** die anfängliche Wissensbasis.
- **Aktionen:** Die Menge der Aktionen besteht aus allen Inferenzregeln, angewandt auf alle Sätze, die mit der oberen Hälfte der Inferenzregel übereinstimmen.
- **Ergebnis:** Das Ergebnis einer Aktion besteht darin, den Satz in die untere Hälfte der Inferenzregel hinzuzufügen.
- **Ziel:** Das Ziel ist ein Zustand, der den Satz enthält, den wir versuchen zu beweisen.

Tipp

Die Suche nach Beweisen ist eine Alternative zum Auflisten von Modellen. In vielen praktischen Fällen *kann die Suche eines Beweises effizienter sein, weil sie irrelevante Aussagensymbole ignorieren kann, und zwar unabhängig davon, wie viele es davon gibt.* Zum Beispiel erwähnt der oben gezeigte Beweis, der zu $\neg P_{1,2} \wedge \neg P_{2,1}$ führte, nicht die Aussagensymbole $B_{2,1}$, $P_{1,1}$, $P_{2,2}$ oder $P_{3,1}$. Sie können ignoriert werden, weil die Zielaussage, $P_{1,2}$, nur in Satz R_2 erscheint; die anderen Aussagensymbole in R_2 erscheinen nur in R_4 und R_2; somit haben R_1, R_3 und R_5 keine Bedeutung für den Beweis. Dasselbe würde auch gelten, wenn wir der Wissensbasis eine Million weiterer Sätze hinzufügten; der einfache Wahrheitstabellenalgorithmus dagegen würde durch das exponentielle Ansteigen der Anzahl an Modellen völlig überfordert.

Eine letzte Eigenschaft logischer Systeme ist die **Monotonie**, die besagt, dass die Menge der folgerbaren Sätze nur *wachsen* kann, wenn der Wissensbasis Informationen hinzugefügt werden.[8] Für alle Sätze α und β gilt:

$$\text{wenn} \quad KB \models \alpha \quad \text{dann} \quad KB \wedge \beta \models \alpha.$$

Nehmen wir zum Beispiel an, die Wissensbasis enthält die zusätzliche Behauptung β, die aussagt, dass es in der Welt genau acht Falltüren gibt. Dieses Wissen könnte dem Agenten helfen, *zusätzliche* Schlüsse zu ziehen, aber er kann keinen Schluss revidieren, den α bereits abgeleitet hat – etwa dass es in [1,2] keine Falltür gibt. Die Monotonie bedeutet, dass Inferenzregeln angewendet werden können, wo geeignete Prämissen in der Wissensbasis gefunden werden – der Schluss aus der Regel muss *unabhängig von dem anderen Inhalt der Wissensbasis* erfolgen.

7.5.2 Beweis durch Resolution

Wir haben behauptet, dass die bisher gezeigten Inferenzregeln *zuverlässig* sind, haben aber noch nicht die Frage der *Vollständigkeit* der Inferenzalgorithmen, die sie verwenden, betrachtet. Suchalgorithmen wie etwa die iterativ vertiefende Suche (*Abschnitt 3.4.5*) sind vollständig in dem Sinne, dass sie jedes erreichbare Ziel finden, aber wenn die verfügbaren Inferenzregeln ungeeignet sind, ist das Ziel nicht erreichbar – es gibt keinen Beweis, der nur diese Inferenzregeln verwendet. Hätten wir beispielsweise die Regel der Bikonditional-Eliminierung weggelassen, wäre der Beweis aus dem vorigen Abschnitt nicht erfolgreich gewesen. Dieser Abschnitt stellt eine einzige Inferenzregel vor, die **Resolution**, die einen vollständigen Inferenzalgorithmus ergibt, wenn sie mit einem beliebigen vollständigen Suchalgorithmus kombiniert wird.

Wir beginnen mit der Anwendung einer sehr einfachen Version der Resolutionsregel in der Wumpus-Welt. Betrachten wir die Schritte, die zu Abbildung 7.4(a) geführt haben: der Agent geht von [2,1] auf [1,1] zurück und dann weiter nach [1,2], wo er einen Gestank, aber keinen Luftzug wahrnimmt. Wir fügen der Wissensbasis die folgenden Tatsachen hinzu:

$$R_{11}: \neg B_{1,2}$$
$$R_{12}: B_{1,2} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{1,3}).$$

8 Die **nichtmonotonen** Logiken, die die Eigenschaft der Monotonie verletzen, bilden eine typische menschliche Eigenschaft beim Schließen nach: es sich anders zu überlegen. Sie werden in *Abschnitt 12.7* beschrieben.

Nach derselben Vorgehensweise, die zuvor zu R_{10} geführt hat, können wir jetzt das Fehlen von Falltüren in [2,2] und [1,3] ableiten (beachten Sie, dass bereits bekannt ist, dass es in [1,1] keine Falltür gibt):

$$R_{13}: \neg P_{2,2}$$
$$R_{14}: \neg P_{1,3}.$$

Außerdem wenden wir die Bikonditional-Eliminierung auf R_3 an, gefolgt von Modus ponens für R_5. Damit erhalten wir die Tatsache, dass es eine Falltür in [1,1], [2,2] oder [3,1] gibt:

$$R_{15}: P_{1,1} \vee P_{2,2} \vee P_{3,1}.$$

Jetzt kommt die erste Anwendung der Resolutionsregel: Das Literal $\neg P_{2,2}$ in R_{13} wird mit dem Literal $P_{2,2}$ in R_{15} *aufgelöst*, um die **Resolvente**

$$R_{16}: P_{1,1} \vee P_{3,1}$$

zu erhalten. Im Klartext: Wenn es eine Falltür in einem der Felder [1,1], [2,2] oder [3,1] gibt und sie nicht in [2,2] liegt, dann befindet sie sich in [1,1] oder [3,1]. Analog dazu wird das Literal $\neg P_{1,1}$ in R_1 mit dem Literal $P_{1,1}$ in R_{16} aufgelöst, sodass sich ergibt:

$$R_{17}: P_{3,1}.$$

Im Klartext: Wenn es eine Falltür in einem der Felder [1,1] oder [3,1] gibt und sie nicht in [1,1] liegt, dann befindet sie sich in [3,1]. Diese beiden letzten Inferenzschritte sind Beispiele für die Inferenzregel der **Einheitsresolution**:

$$\frac{l_1 \vee \ldots \vee l_k, m}{l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k}.$$

wobei jedes l ein Literal und l_i und m **komplementäre Literale** (d.h., das eine ist eine Negation des anderen) sind. Die Einheitsresolutionsregel verwendet also eine **Klausel** – eine Disjunktion von Literalen – und ein Literal und erzeugt daraus eine neue Klausel. Beachten Sie, dass ein einzelnes Literal als Disjunktion eines Literals betrachtet werden kann und daher auch als **Einheitsklausel** bezeichnet wird.

Die Einheitsresolutionsregel kann auch zur vollständigen **Resolutionsregel** verallgemeinert werden:

$$\frac{l_1 \vee \ldots \vee l_k, m_1 \vee \ldots \vee m_n}{l_1 \vee \ldots \vee l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k \vee m_1 \vee \ldots m_{j-1} \vee m_{j+1} \vee \ldots \vee m_n},$$

wobei l_i und m_j komplementäre Literale sind. Das bedeutet, die Resolution nimmt zwei Klauseln und erzeugt eine neue Klausel, die alle Literale aus den beiden ursprünglichen Klauseln enthält, *außer* den beiden komplementären Literalen. Wir haben beispielsweise:

$$\frac{P_{1,1} \vee P_{3,1}, \neg P_{1,1} \vee \neg P_{2,2}}{P_{3,1} \vee \neg P_{2,2}}.$$

Es gibt einen eher technischen Aspekt der Resolutionsregel: Die resultierende Klausel sollte nur eine Kopie jedes Literals enthalten.[9] Das Entfernen von Mehrfachkopien von

9 Wird eine Klausel als *Menge* von Literalen betrachtet, wird diese Beschränkung automatisch berücksichtigt. Verwendet man die Mengennotation für die Klauseln, wird die Auflösungsregel sehr viel verständlicher, allerdings auf Kosten einer zusätzlich einzuführenden Notation.

Literalen wird auch als **Faktorisierung** bezeichnet. Wenn wir beispielsweise $(A \vee B)$ mit $(A \vee \neg B)$ auflösen, erhalten wir $(A \vee A)$, was einfach auf A reduziert wird.

Die *Korrektheit* der Resolutionsregel ist leicht ersichtlich, wenn man das Literal l_i betrachtet. Ist l_i wahr, dann ist m_j falsch und damit muss $m_1 \vee ... \vee m_{j-1} \vee m_{j+1} \vee ... \vee m_n$ wahr sein, weil $m_1 \vee ... \vee m_n$ gegeben ist. Ist l_i falsch, dann muss $l_1 \vee ... \vee l_{i-1} \vee l_{i+1} \vee ... \vee l_k$ wahr sein, weil $l_1 \vee ... \vee l_k$ gegeben ist. Nun ist l_i entweder wahr oder falsch, sodass die eine oder die andere dieser Schlussfolgerungen gilt – genau wie die Resolutionsregel sagt.

Tipp

Was für die Resolutionsregel mehr überrascht, ist, dass sie die Grundlage für eine Familie *vollständiger* Inferenzprozeduren bildet. *Ein resolutionsbasierter Theorembeweiser kann für beliebige Sätze α und β in Aussagenlogik entscheiden, ob $\alpha \models \beta$ ist.* Die beiden nächsten Unterabschnitte erklären, wie Resolution dies bewerkstelligt.

Konjunktive Normalform

Tipp

Die Resolutionsregel ist nur auf Disjunktionen von Literalen anwendbar, scheint also nur für Wissensbasen und Abfragen relevant zu sein, die aus Klauseln bestehen. Wie kann sie dann zu einer vollständigen Inferenzprozedur für die gesamte Aussagenlogik führen? Die Antwort ist, *dass jeder Satz der Aussagenlogik logisch äquivalent mit einer Konjunktion von Klauseln ist.* Ein Satz, der als Konjunktion von Klauseln ausgedrückt wird, befindet sich in der sogenannten **konjunktiven Normalform** (**KNF**, siehe Abbildung 7.15). Wir beschreiben nun eine einfache Umwandlungsprozedur in KNF. Wir veranschaulichen diese Prozedur, indem wir den Satz $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$, in KNF umwandeln. Das geschieht in folgenden Schritten:

1 Wir eliminieren $\Leftrightarrow$ und ersetzen $\alpha \Leftrightarrow \beta$ durch $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$:

$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$.

2 Wir eliminieren $\Rightarrow$, indem wir $\alpha \Rightarrow \beta$ durch $\neg\alpha \vee \beta$ ersetzen:

$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$.

3 KNF setzt voraus, dass $\neg$ nur in Literalen erscheint, wir „verschieben also $\neg$ nach innen“, indem wir wiederholt die folgenden Äquivalenzen aus Abbildung 7.11 anwenden:

$\neg(\neg\alpha) \equiv \alpha$	Eliminierung der doppelten Negation
$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta)$	de Morgan
$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$	de Morgan

Im Beispiel brauchen wir lediglich eine Anwendung der letzten Regel:

$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$.

4 Jetzt haben wir einen Satz mit verschachtelten $\wedge$- und $\vee$-Operatoren, die auf Literale angewendet werden. Wir wenden das Distributivgesetz aus Abbildung 7.11 an und verteilen $\vee$ über $\wedge$, wo immer das möglich ist.

$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$

Der ursprüngliche Satz befindet sich nun in KNF – als Konjunktion von drei Klauseln. Er ist jetzt sehr viel schwieriger zu lesen, kann aber als Eingabe für eine Resolutionsprozedur verwendet werden.

Ein Resolutionsalgorithmus

Auf der Resolution basierende Inferenzprozeduren arbeiten unter Verwendung des Beweisprinzips durch Widerspruch, wie in *Abschnitt 7.5* beschrieben. Um also zu zeigen, dass $KB \models \alpha$ ist, zeigen wir, dass $(KB \wedge \neg\alpha)$ nicht erfüllbar ist. Dazu benutzen wir einen Widerspruchsbeweis.

▶ Abbildung 7.12 zeigt einen Resolutionsalgorithmus. Zuerst wird $(KB \wedge \neg\alpha)$ in KNF umgewandelt. Anschließend wird die Resolutionsregel auf die resultierenden Klauseln angewendet. Jedes Paar, das komplementäre Literale enthält, wird aufgelöst, um eine neue Klausel zu erzeugen, die der Menge hinzugefügt wird, falls sie nicht schon darin enthalten ist. Der Prozess wird fortgesetzt, bis eines von zwei Dingen passiert:

- Es gibt keine neuen Klauseln, die hinzugefügt werden können, dann folgt α nicht aus *KB*, oder:
- Zwei Klauseln werden aufgelöst, um die *leere* Klausel zu liefern, in welchem Fall α aus *KB* folgt.

```
function PL-RESOLUTION(KB, α) returns true oder false
  inputs: KB, die Wissensbasis, ein Satz in Aussagenlogik
          α, die Abfrage, ein Satz in Aussagenlogik

  clauses ← die Menge der Klauseln in der KNF-Darstellung von KB ∧ ¬α
  new ← { }
  loop do
    for each Paar von Klauseln Ci, Cj in clauses do
      resolvents ← PL-RESOLVE(Ci, Cj)
      if resolvents enthält die leere Klausel then return true
      new ← new ∪ resolvents
    if new ⊆ clauses then return false
    clauses ← clauses ∪ new
```

Abbildung 7.12: Ein einfacher Resolutionsalgorithmus für die Aussagenlogik. Die Funktion PL-RESOLVE gibt die Menge aller möglichen Klauseln zurück, die durch Resolution ihrer beiden Eingaben erhalten werden.

Die leere Klausel – eine Disjunktion ohne Disjunkte – ist äquivalent mit *False*, weil eine Disjunktion nur dann wahr ist, wenn mindestens eine ihrer Disjunkte wahr ist. Eine andere Möglichkeit zu erkennen, ob eine leere Klausel einen Widerspruch darstellt, ist die Beobachtung, dass sie nur aus der Resolution von zwei komplementären Einheitsklauseln wie etwa P und $\neg P$ entsteht.

Wir können die Resolutionsprozedur auf eine sehr einfache Inferenz in der Wumpus-Welt anwenden. Wenn sich der Agent auf [1,1] befindet, spürt er keinen Luftzug; es kann also keine Falltür auf benachbarten Feldern geben. Die relevante Wissensbasis ist:

$$KB = R_2 \wedge R_4 = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}.$$

Wir wollen α beweisen, das zum Beispiel $\neg P_{1,2}$ ist. Wenn wir $(KB \wedge \neg\alpha)$ in KNF umwandeln, erhalten wir die oben in ▶ Abbildung 7.13 gezeigten Klauseln. Die zweite Zeile in der Abbildung zeigt Klauseln, die durch Resolution von Klauselpaaren aus der ersten Zeile entstehen. Wird dann $P_{1,2}$ mit $\neg P_{1,2}$ aufgelöst, erhalten wir die leere Klausel, dargestellt als kleines Quadrat. Betrachtet man Abbildung 7.13 genauer, zeigt sich, dass viele Resolutionsschritte sinnlos sind. Beispielsweise ist die Klausel

$B_{1,1} \vee \neg B_{1,1} \vee P_{1,2}$ äquivalent mit $True \vee P_{1,2}$, was äquivalent mit $True$ ist. Die Ableitung, dass $True$ wahr ist, ist nicht sehr hilfreich. Aus diesem Grund können Klauseln, in denen zwei komplementäre Literale auftreten, verworfen werden.

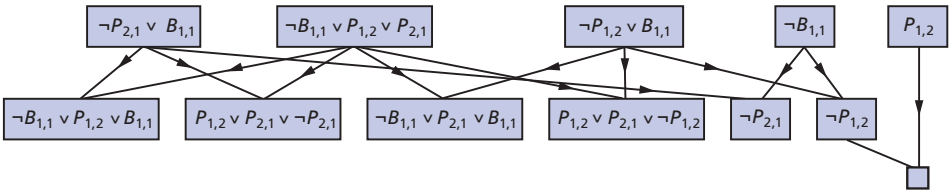


Abbildung 7.13: Partielle Anwendung von PL-RESOLUTION auf eine einfache Inferenz in der Wumpus-Welt. Es wird gezeigt, dass $\neg P_{1,2}$ aus den ersten vier Klauseln in der obersten Zeile folgt.

Vollständigkeit der Resolution

Um unsere Diskussion der Resolution abzuschließen, zeigen wir jetzt, warum PL-RESOLUTION vollständig ist. Dazu führen wir den **Resolutionsabschluss** $RC(S)$ einer Menge von Klauseln S ein, d.h. die Menge aller durch wiederholte Anwendung der Resolutionsregel auf Klauseln in S oder ihre Ableitungen ableitbaren Klauseln. Der Resolutionsschluss ist, was PL-RESOLUTION als den Endwert der Variablen *clauses* berechnet. Es ist leicht erkennbar, dass $RC(S)$ endlich sein muss, weil es nur endlich viele verschiedene Klauseln gibt, die aus den Symbolen $P_1, ..., P_k$, die in S erscheinen, konstruiert werden können. (Beachten Sie, dass dies ohne den Faktorisierungsschritt, der Mehrfachkopien von Literalen entfernt, nicht gilt!) Damit terminiert RL-RESOLUTION immer.

Das Vollständigkeitstheorem für die Resolution in der Aussagenlogik wird als **(Grund-) Resolutionstheorem** bezeichnet:

> *Wenn eine Menge von Klauseln nicht erfüllbar ist, enthält der Resolutionsabschluss dieser Klauseln die leere Klausel.*

Wir beweisen dieses Theorem, indem wir sein Gegenteil zeigen: Wenn der Schluss $RC(S)$ die leere Klausel *nicht* enthält, ist S erfüllbar. In der Tat können wir ein Modell für S mit geeigneten Wahrheitswerten $P_1, ..., P_k$ erstellen. Die Konstruktionsprozedur sieht folgendermaßen aus:

Für i von 1 bis k:

- Wenn eine Klausel in $RC(S)$ das Literal $\neg P_i$ enthält und alle ihre anderen Literale unter der für $P_i, ..., P_{i-1}$ gewählten Zuweisung falsch sind, wird *false* an P_i zugewiesen.
- Andernfalls wird *true* an P_i zugewiesen.

Diese Zuweisung an $P_1, ..., P_k$ ist ein Modell von S. Um dies zu zeigen, nehmen wir das Gegenteil an – d.h., dass auf einer bestimmten Stufe i in der Sequenz die Zuweisung des Symbols P_i bewirkt, dass eine Klausel C falsch wird. Damit dies passiert, müssen bereits alle *anderen* Literale in C durch Zuweisungen an $P_1, ..., P_{i-1}$ falsifiziert worden sein. Somit muss C jetzt entweder wie ($false \vee false \vee ... false \vee P_i$) oder wie ($false \vee false \vee ...false \vee \neg P_i$) aussehen. Wenn nur eine dieser beiden Klauseln in $RC(S)$ enthalten ist, weist der Algorithmus den entsprechenden Wahrheitswert an P_i zu, um C wahr zu machen, sodass C nur falsifiziert werden kann, wenn *beide* Klauseln in $RC(S)$

vorkommen. Da nun $RC(S)$ unter Resolution abgeschlossen wird, enthält es die Resolvente dieser beiden Klauseln und diese Resolvente hat bereits alle ihre Literale durch die Zuweisungen an P_1, ..., P_{i-1} falsifizieren lassen. Dies widerspricht unserer Annahme, dass die erste falsifizierte Klausel auf der Stufe i erscheint. Folglich haben wir nachgewiesen, dass die Konstruktion niemals eine Klausel in $RC(S)$ falsifiziert; d.h., sie erzeugt ein Modell von $RC(S)$ und somit ein Modell von S selbst (da S in $RC(S)$ enthalten ist).

7.5.3 Horn-Klauseln und definitive Klauseln

Die Vollständigkeit der Resolution macht sie zu einer sehr wichtigen Inferenzmethode. In vielen praktischen Situationen braucht man jedoch nicht die ganze Leistungsfähigkeit der Resolution. Manche Wissensbasen erfüllen in der Praxis nur bestimmte Einschränkungen in der Form der Sätze, die sie enthalten, und können so einen restriktiveren und effizienteren Inferenzalgorithmus verwenden.

Eine derartige eingeschränkte Form ist die **definitive Klausel**, eine Disjunktion von Literalen, von denen *genau eines positiv ist*. Beispielsweise ist die Klausel $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$ eine definitive Klausel, $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1})$ dagegen nicht.

Etwas allgemeiner gefasst ist die **Horn-Klausel** (▶ Abbildung 7.14) eine Disjunktion von Literalen, von denen *höchstens eine positiv ist*. Horn-Klauseln werden unter Resolution abgeschlossen: Wenn Sie zwei Horn-Klauseln auflösen, erhalten Sie eine Horn-Klausel zurück.

Wissensbasen, die ausschließlich definitive Klauseln enthalten, sind aus folgenden drei Gründen interessant:

1 Jede definitive Klausel kann als Implikation geschrieben werden, deren Prämisse eine Konjunktion positiver Literale und deren Schluss ein einziges positives Literal ist (siehe Übung 7.13). Zum Beispiel lässt sich die definite Klausel $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$ als die Implikation $(L_{1,1} \wedge Breeze) \Rightarrow B_{1,1}$ schreiben. In der zuletzt gezeigten Form ist der Satz verständlicher und sagt Folgendes aus: Wenn sich der Agent auf dem Feld [1,1] befindet und einen Luftzug spürt, dann gibt es in [1,1] einen Luftzug.

In der Horn-Form wird die Prämisse als **Rumpf** und der Schluss als **Kopf** bezeichnet. Ein Satz, der aus einem einzigen positiven Literal wie zum Beispiel $L_{1,1}$ besteht, heißt **Fakt**. Er lässt sich ebenfalls in der Implikationsform als $True \Rightarrow L_{1,1}$ schreiben, doch ist es einfacher, lediglich $L_{1,1}$ zu schreiben.

2 Inferenz mit Horn-Klauseln kann mithilfe von Algorithmen zur **Vorwärts-** oder **Rückwärtsverkettung** erfolgen, die nachfolgend erklärt werden. Beide Algorithmen sind sehr natürlich, weil die Inferenzschritte offensichtlich und für den Menschen leicht nachzuvollziehen sind. Diese Art der Inferenz ist die Grundlage für **logische Programmierung**, mit der sich *Kapitel 9* beschäftigt.

3 Die logische Konsequenz mit Horn-Klauseln kann in einer Zeit ermittelt werden, die *linear* zur Größe der Wissensbasis ist – eine angenehme Überraschung.

$$
\begin{aligned}
\mathit{KNFSentence} &\rightarrow \mathit{Klausel}_1 \wedge \ldots \wedge \mathit{Klausel}_n \\
\mathit{Clause} &\rightarrow \mathit{Literal}_1 \vee \ldots \vee \mathit{Literal}_m \\
\mathit{Literal} &\rightarrow \mathit{Symbol} \mid \neg \mathit{Symbol} \\
\mathit{Symbol} &\rightarrow P \mid Q \mid R \mid \ldots \\
\mathit{HornClauseForm} &\rightarrow \mathit{DefiniteClauseForm} \mid \mathit{GoalClauseForm} \\
\mathit{DefiniteClauseForm} &\rightarrow (\mathit{Symbol}_1 \wedge \ldots \wedge \mathit{Symboll}) \Rightarrow \mathit{Symbol} \\
\mathit{GoalClauseForm} &\rightarrow (\mathit{Symbol}_1 \wedge \ldots \wedge \mathit{Symboll}) \Rightarrow \mathit{False}
\end{aligned}
$$

Abbildung 7.14: Eine Grammatik für die konjunktive Normalform (KNF), Horn-Klauseln und definitive Klauseln. Eine Klausel wie zum Beispiel $A \wedge B \Rightarrow C$ ist immer noch eine definitive Klausel, wenn sie als $\neg A \vee \neg B \vee C$ geschrieben wird. Allerdings gilt nur die erste Schreibweise als kanonische Form für definitive Klauseln. Eine weitere Klasse ist der *k*-KNF-Satz, ein KNF-Satz, bei dem jede Klausel höchstens *k* Literale enthält.

7.5.4 Vorwärts- und Rückwärtsverkettung

Der Algorithmus zur Vorwärtsverkettung, PL-FC-ENTAILS?(*KB*, *q*), stellt fest, ob ein einzelnes Aussagensymbol *q* – die Abfrage – durch eine Wissensbasis mit definitiven Klauseln folgerbar ist. Er beginnt mit bekannten Tatsachen (positiven Literalen) aus der Wissensbasis. Wenn alle Prämissen einer Implikation bekannt sind, wird ihr Schluss der Menge bekannter Fakten hinzugefügt. Sind beispielsweise $L_{1,1}$ und *Breeze* bekannt und $(L_{1,1} \wedge \mathit{Breeze}) \Rightarrow B_{1,1}$ ist die Wissensbasis, kann $B_{1,1}$ hinzugefügt werden. Dieser Prozess wird fortgesetzt, bis die Abfrage *q* hinzugefügt wird oder keine weiteren Inferenzen mehr stattfinden können. ▶ Abbildung 7.15 zeigt den detaillierten Algorithmus; der wichtigste Aspekt dabei ist, dass er in linearer Zeit ausgeführt werden kann.

```
function PL-FC-ENTAILS?(KB, q) returns true oder false
  inputs: KB, die Wissensbasis, eine Menge von definitiven Klauseln
                der Aussagenlogik
          q, die Abfrage, ein Aussagensymbol
  count ← eine Tabelle, wobei count[c] die Anzahl der Symbole in der
          Prämisse von c ist
  inferred ← eine Tabelle, wobei inferred[s] anfangs false für
             alle Symbole ist
  agenda ← eine Warteschlange mit Symbolen, anfangs die Symbole, die in
           KB als wahr bekannt sind

  while agenda ist nicht leer do
    p ← POP(agenda)
    if p = q then return true
    if inferred[p] = false then
      inferred[p] ← true
      for each Klausel c in KB, bei der p in c.PREMISE ist do
        dekrementiere count[c]
        if count[c] = 0 then füge c.CONCLUSION zu agenda hinzu
  return false
```

Abbildung 7.15: Der Algorithmus für die Vorwärtsverkettung in der Aussagenlogik. Die *agenda* verwaltet die als wahr bekannten, aber noch nicht „verarbeiteten" Symbole. Die Tabelle *count* verwaltet, wie viele Prämissen jeder Implikation noch unbekannt sind. Immer wenn ein neues Symbol *p* aus der Agenda verarbeitet wird, wird der Zähler für jede Implikation, in deren Prämisse *p* erscheint, um 1 verringert (mit geeigneter Indizierung leicht in konstanter Zeit zu ermitteln). Erreicht ein Zähler den Wert 0, sind alle Prämissen der Implikation bekannt, sodass der Schluss der Agenda hinzugefügt werden kann. Schließlich müssen wir noch verwalten, welche Symbole verarbeitet wurden; ein abgeleitetes Symbol muss der Agenda nicht hinzugefügt werden, wenn es zuvor verarbeitet wurde. Damit vermeidet man wiederholte Arbeit und verhindert Endlosschleifen, die durch Implikationen wie $P \Rightarrow Q$ und $Q \Rightarrow P$ entstehen.

Der Algorithmus lässt sich am besten anhand eines Beispiels und eines Bildes verstehen. ▶ Abbildung 7.16(a) zeigt eine einfache Wissensbasis aus Horn-Klauseln mit A und B als bekannten Fakten. ▶ Abbildung 7.16(b) zeigt dieselbe Wissensbasis als **AND-OR-Graphen** (siehe *Kapitel 4*). In AND-OR-Graphen weisen mehrere Verknüpfungen durch eine Kante auf eine Konjunktion hin – jede Verknüpfung muss bewiesen sein –, während mehrere Verknüpfungen ohne Kante eine Disjunktion anzeigen – jede beliebige Verknüpfung kann bewiesen werden. Es ist leicht zu sehen, wie die Vorwärtsverkettung im Graphen funktioniert. Die bekannten Blätter (hier A und B) sind festgelegt und die Inferenz leitet den Graphen so weit wie möglich weiter. Immer wenn eine Konjunktion erscheint, wartet die Weiterleitung, bis alle Konjunkte bekannt sind, bevor fortgefahren wird. Der Leser sollte das Beispiel im Detail nachvollziehen.

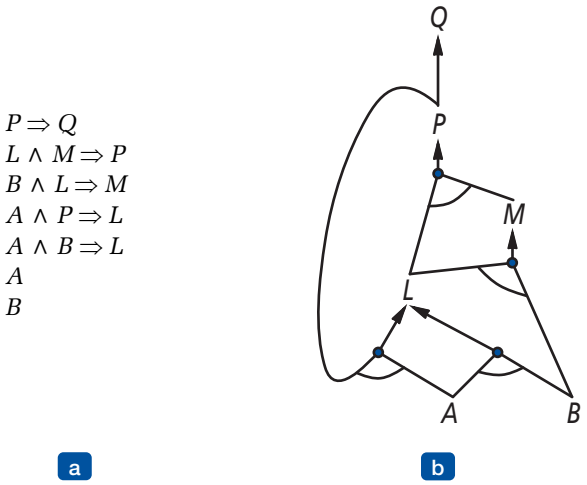


Abbildung 7.16: (a) eine einfache Wissensbasis aus Horn-Klauseln. (b) Der entsprechende AND-OR-Graph.

Tipp

Es ist leicht zu erkennen, dass die Vorwärtsverkettung **korrekt** ist: Jede Inferenz ist letztlich eine Anwendung des Modus ponens. Darüber hinaus ist die Vorwärtsverkettung **vollständig**: Jeder folgerbare atomare Satz wird abgeleitet. Am einfachsten lässt sich dies erkennen, wenn man den endgültigen Zustand der *abgeleiteten* Tabelle betrachtet (nachdem der Algorithmus einen **Fixpunkt** erreicht hat, von wo aus keine neuen Inferenzen mehr möglich sind.) Die Tabelle enthält *true* für jedes während des Prozesses abgeleitete Symbol und *false* für alle anderen Symbole. Wir können die Tabelle als logisches Modell betrachten; darüber hinaus ist *jede definitive Klausel in der ursprünglichen KB in diesem Modell wahr*. Um dies zu erkennen, nehmen wir das Gegenteil an, nämlich dass eine Klausel $a_1 \wedge \ldots \wedge a_k \Rightarrow b$ im Modell falsch ist. Dann muss $a_1 \wedge \ldots \wedge a_k$ im Modell wahr sein und b muss im Modell falsch sein. Das widerspricht jedoch unserer Annahme, dass der Algorithmus einen Fixpunkt erreicht hat! Wir können deshalb schließen, dass die Menge der atomaren Sätze, die an dem Fixpunkt abgeleitet wurden, ein Modell der ursprünglichen Wissensbasis definiert. Darüber hinaus muss jeder atomare Satz q, der aus der Wissensbasis gefolgert wird, in allen ihren Modellen und speziell in diesem Modell wahr sein. Also muss jede logische Konsequenz q durch den Algorithmus abgeleitet werden.

Die Vorwärtsverkettung ist ein Beispiel für das allgemeine Konzept des **datengesteuerten Schließens** – d.h. des Schließens, wobei zunächst die bekannten Daten berücksichtigt werden. Es kann innerhalb eines Agenten verwendet werden, um Schlüsse aus eingehenden Wahrnehmungen zu ziehen – häufig ohne spezifische Abfrage im Sinn. Beispielsweise könnte der Wumpus-Agent seine Wahrnehmungen mittels TELL an die Wissensbasis weitergeben – unter Verwendung eines inkrementellen Vorwärtsverkettungsalgorithmus, wobei der Agenda neue Fakten hinzugefügt werden können, um neue Inferenzen zu initiieren. In Menschen findet ein gewisses datengesteuertes Schließen statt, wenn neue Information eintrifft. Wenn ich beispielsweise in einem Raum bin und höre, dass es anfängt zu regnen, könnte mir bewusst werden, dass das Picknick abgesagt wird. Es fällt mir dagegen wahrscheinlich nicht ein, dass das siebzehnte Blütenblatt der größten Rose im Nachbargarten nass wird; Menschen verwenden die Vorwärtsverkettung unter sorgfältiger Kontrolle, sodass sie nicht mit irrelevanten Konsequenzen überschwemmt werden.

Der Algorithmus für die Rückwärtsverkettung arbeitet, wie der Name schon sagt, von der Abfrage aus rückwärts. Wenn man weiß, dass die Abfrage q wahr ist, ist keine Arbeit erforderlich. Andernfalls sucht der Algorithmus diejenigen Implikationen in der Wissensbasis, deren Konklusion q ist. Können alle Prämissen von einer dieser Implikationen als wahr bewiesen werden (durch Rückwärtsverkettung), ist q wahr. Wird dieser Algorithmus auf die Abfrage Q in Abbildung 7.16 angewendet, arbeitet er sich im Graphen zurück, bis er eine Menge bekannter Fakten A und B erreicht hat, die die Grundlage für einen Beweis bildet. Der Algorithmus ist im Wesentlichen mit dem AND-OR-GRAPH-SEARCH-Algorithmus in Abbildung 4.11 identisch. Wie bei der Vorwärtsverkettung wird eine effiziente Implementierung in linearer Zeit ausgeführt.

Die Rückwärtsverkettung ist eine Art **zielgerichteten Schließens**. Sie ist praktisch, um bestimmte Fragen zu beantworten, wie etwa „Was soll ich jetzt tun?“ oder „Wo sind meine Schlüssel?“ Häufig sind die Kosten für die Rückwärtsverkettung *sehr viel geringer* als linear zur Größe der Wissensbasis, weil der Prozess nur die relevanten Fakten berücksichtigt.

7.6 Effektive aussagenlogische Inferenz

In diesem Abschnitt beschreiben wir zwei Familien effizienter Algorithmen für aussagenlogische Inferenz, die auf Model-Checking basieren. Ein Ansatz basiert auf der Backtracking-Suche, ein anderer auf einer Hill-Climbing-Suche. Diese Algorithmen sind Teil der „Technologie“ der Aussagenlogik. Dieser Abschnitt kann beim ersten Durchlesen des Kapitels kurz überflogen werden.

Die hier beschriebenen Algorithmen überprüfen die Erfüllbarkeit. (Wie bereits erwähnt, lässt sich die logische Konsequenz $\alpha \models \beta$ testen, indem man die *Nicht*erfüllbarkeit von $\alpha \wedge \neg\beta$ testet.) Wir haben bereits auf den Zusammenhang hingewiesen, ein zufriedenstellendes Modell für einen logischen Satz und die Lösung für ein Problem unter Rand- und Nebenbedingungen zu finden; deshalb überrascht es vielleicht nicht, dass die beiden Algorithmenfamilien so stark an die Backtracking-Algorithmen in *Abschnitt 6.3* und die lokalen Suchalgorithmen aus *Abschnitt 6.4* erinnern. Sie sind jedoch auch unabhängig davon sehr wichtig, weil viele kombinatorische Probleme aus der Informatik auf eine Überprüfung der Erfüllbarkeit eines aussagenlogischen Satzes reduziert werden können. Jede Verbesserung von Erfüllbarkeitsalgorithmen hat enorme Konsequenzen für unsere Fähigkeit, allgemein mit Komplexität umzugehen.

7.6.1 Ein vollständiger Backtracking-Algorithmus

Der erste Algorithmus, den wir hier betrachten, wird häufig auch als **Davis-Putman-Algorithmus** bezeichnet, nach der bahnbrechenden Arbeit von Martin Davis und Hilary Putman (1960). Praktisch geht es hier um die Version des Algorithmus, die von Davis, Logemann und Loveland (1962) beschrieben wurde; deshalb werden wir ihn nach den Initialen der vier Autoren als DPLL bezeichnen. DPLL nimmt einen Satz in konjunktiver Normalform als Eingabe entgegen – eine Menge von Klauseln. Wie bei BACKTRACKING-SEARCH und TT-ENTAILS? handelt es sich dabei letztlich um eine rekursive Tiefensuchauflistung möglicher Modelle. Er verkörpert drei Verbesserungen gegenüber dem einfachen Schema von TT-ENTAILS?:

- *Frühe Terminierung*: Der Algorithmus erkennt, ob der Satz wahr oder falsch sein muss, selbst bei einem partiell vollständigen Modell. Eine Klausel ist wahr, wenn *ein beliebiges* Literal wahr ist, selbst wenn die anderen Literale noch keine Wahrheitswerte haben; damit könnte der Satz als Ganzes als wahr erachtet werden, noch bevor das Modell vollständig ist. Beispielsweise ist der Satz $(A \vee B) \wedge (A \vee C)$ wahr, wenn A wahr ist, unabhängig von den Werten von B und C. Analog dazu ist ein Satz falsch, wenn irgendeine Klausel falsch ist, was vorkommt, wenn jedes seiner Literale falsch ist. Auch das kann auftreten, lange bevor das Modell vollständig ist. Die frühe Terminierung vermeidet die Auswertung ganzer Unterbäume im Suchraum.
- *Reines-Symbol-Heuristik*: Ein **reines Symbol** ist ein Symbol, das in allen Klauseln immer mit demselben „Vorzeichen“ erscheint. In den drei Klauseln $(A \vee \neg B)$, $(\neg B \vee \neg C)$ und $(C \vee A)$ ist das Symbol A rein, weil nur das positive Literal erscheint, B ist rein, weil nur das negative Literal erscheint, und C ist nicht rein. Man erkennt leicht: Hat ein Satz ein Modell, dann hat er ein Modell, bei dem die reinen Symbole so zugewiesen sind, dass ihre Literale *true* ergeben, weil damit eine Klausel niemals falsch werden kann. Beachten Sie, dass der Algorithmus bei der Ermittlung der Reinheit eines Symbols Klauseln ignorieren kann, von denen man bereits weiß, dass sie im bisher konstruierten Modell wahr sind. Enthält das Modell beispielsweise $B = \mathit{false}$, dann ist die Klausel $(\neg B \vee \neg C)$ bereits wahr und in den übrigen Klauseln erscheint C nur als positives Literal; demzufolge wird C rein.
- *Einheitsklausel*-Heuristik: Eine **Einheitsklausel** wurde weiter oben als Klausel mit nur einem Literal definiert. Im Kontext von DPLL steht sie auch für Klauseln, in denen allen Literalen außer einem bereits *false* vom Modell zugewiesen wurde. Enthält das Modell beispielsweise $B = \mathit{true}$, vereinfacht sich $(\neg B \vee \neg C)$ zu $\neg C$, einer Einheitsklausel. Wenn diese Klausel wahr sein soll, muss C offensichtlich auf *false* gesetzt werden. Die Einheitsklausel-Heuristik weist alle diese Symbole zu, bevor sie zum Rest verzweigt. Eine wichtige Konsequenz dieser Heuristik ist, dass jeder Versuch, ein Literal (durch Widerlegung) zu beweisen, das sich bereits in der Wissensbasis befindet, sofort erfolgreich ist (Übung 7.22). Beachten Sie auch, dass die Zuweisung an eine Einheitsklausel eine weitere Einheitsklausel erzeugen kann – z.B. wenn C auf *false* gesetzt ist, wird $(C \vee A)$ zu einer Einheitsklausel und bewirkt damit, dass *true* an A zugewiesen wird. Diese „Kaskade“ erzwungener Zuweisungen wird als **Einheitspropagierung** bezeichnet. Sie erinnert an den Prozess der Vorwärtsverkettung mit definiten Klauseln und wenn der KNF-Ausdruck nur definitive Klauseln enthält, ist DPLL im Wesentlichen eine Nachbildung der Vorwärtsverkettung (siehe Übung 7.23).

```
function DPLL(clauses, symbols, model) returns true oder false

  if jede Klausel in clauses ist wahr in model then return true
  if eine Klausel in clauses ist falsch in model then return false
  P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
  if P ist nicht null then return DPLL(clauses, symbols - P, model ∪ {P = value})
  P, value ← FIND-UNIT-CLAUSE(clauses, model)
  if P ist nicht null then return DPLL(clauses, symbols - P, model ∪ {P = value})
  P ← FIRST(symbols); rest ← REST(symbols)
  return DPLL(clauses, rest, model ∪ {P = true}) or
         DPLL(clauses, rest, model ∪ {P = false}))
```

Abbildung 7.17: Der DPLL-Algorithmus für die Überprüfung der Erfüllbarkeit eines Satzes in der Aussagenlogik. FIND-PURE-SYMBOL und FIND-UNIT-CLAUSE sind im Text erklärt; beide geben ein Symbol (oder null) zurück sowie den Wahrheitswert, der diesem Symbol zugewiesen werden soll. Wie TT-ENTAILS? arbeitet er über partielle Modelle.

▶ Abbildung 7.17 zeigt den DPLL-Algorithmus und stellt damit im Wesentlichen das Gerüst des Suchprozesses dar.

Dagegen zeigt Abbildung 7.17 nicht die Tricks, mit denen sich SAT-Solver auf große Probleme skalieren lassen. Interessant ist, dass die meisten dieser Tricks in der Tat sehr allgemein sind und bereits in anderer Gestalt zu sehen waren:

1. **Komponentenanalyse** (wie bei Tasmanien in CSPs): Wenn DPLL Wahrheitswerte an Variablen zuweist, kann die Menge der Klauseln in zwei disjunkte Teilmengen – die sogenannten **Komponenten** – getrennt werden, die keine nicht zugewiesenen Variablen gemeinsam haben. Lässt sich mit einem effizienten Verfahren erkennen, wann dies auftritt, kann ein Solver (Lösungsprogramm) beträchtliche Geschwindigkeitsgewinne erzielen, indem er auf jeder Komponente separat arbeitet.

2. **Variablen- und Wertereihenfolge** (wie in *Abschnitt 6.3.1* für CSPs erläutert): Unsere einfache Implementierung von DPLL verwendet eine willkürliche Variablenreihenfolge und probiert immer den Wert *true* vor *false* aus. Gemäß der Gradheuristik (siehe *Abschnitt 6.3.1*) ist die Variable auszuwählen, die am häufigsten über allen restlichen Klauseln erscheint.

3. **Intelligentes Backtracking** (wie in *Abschnitt 5.3* für CSPs erläutert): Viele Probleme, die sich mit chronologischem Backtracking nicht in Stunden der Laufzeit lösen lassen, benötigen nur Sekunden mit intelligentem Backtracking, das den gesamten Weg zurück zum Konfliktpunkt sichert. Sämtliche SAT-Solver verwenden in irgendeiner Form das **Lernen von Konfliktklauseln**, um Konflikte aufzuzeichnen, damit sie später in der Suche nicht wiederholt werden. Normalerweise speichern die Programme nur eine begrenzte Menge von Konflikten und verwerfen selten genutzte Klauseln.

4. **Zufällige Neustarts** (wie in *Abschnitt 4.1.1* für das Hill-Climbing gezeigt): Manchmal scheint ein Lauf keinen Fortschritt zu machen. In diesem Fall können wir von der Spitze des Suchbaumes aus neu starten, anstatt zu versuchen, die aktuelle Ausführung fortzusetzen. Nach dem Neustart werden andere Zufallsauswahlen (sowohl bei Variablen als auch Werten) vorgenommen. Die im ersten Lauf erlernten Klauseln bleiben auch nach einem Neustart erhalten und können dabei helfen, den Suchraum zu verkleinern. Zwar garantiert Neustarten nicht, dass eine Lösung schneller gefunden wird, reduziert aber die zeitlichen Schwankungen bis zu einer Lösungsfindung.

5 Geschicktes Indizieren (wie in vielen Algorithmen zu sehen): Die in DPLL selbst verwendeten Beschleunigungsmethoden sowie die in modernen Solvern eingesetzten Tricks setzen schnelles Indizieren von Elementen wie zum Beispiel „die Menge der Klauseln, in denen Variable X_i als positives Literal erscheint“ voraus. Komplizierter wird diese Aufgabe durch die Tatsache, dass die Algorithmen nur an den Klauseln interessiert sind, die durch vorhergehende Zuweisungen an Variablen noch nicht erfüllt sind, sodass die Indizierungsstrukturen dynamisch zu aktualisieren sind, wenn die Berechnung voranschreitet.

Diese Erweiterungen versetzen moderne Solver in die Lage, Probleme mit Millionen von Variablen zu lösen. Sie haben Gebiete wie Hardwareverifikation und Sicherheitsprotokollverifikation revolutioniert, die vorher umständliche, handgeführte Beweise erforderten.

7.6.2 Lokale Suchalgorithmen

Bisher haben wir in diesem Buch mehrere lokale Suchalgorithmen vorgestellt, unter anderem HILL-CLIMBING (*Abschnitt 4.1.1*) und SIMULATED-ANNEALING. Diese Algorithmen können direkt auf Erfüllbarkeitsprobleme angewendet werden, vorausgesetzt, wir wählen die richtige Bewertungsfunktion. Weil es das Ziel ist, eine Zuweisung zu finden, die jede Klausel erfüllt, ist eine Bewertungsfunktion geeignet, die die Anzahl der nicht erfüllten Klauseln zählt. Genau dieses Maß wird vom MIN-CONFLICTS-Algorithmus für CSPs (*Abschnitt 6.4*) verwendet. Alle diese Algorithmen unternehmen Schritte im Raum vollständiger Zuweisungen und wechseln jeweils den Wahrheitswert eines einzigen Symbols. Der Raum enthält normalerweise viele lokale Minima, sodass man verschiedene Formen der Zufälligkeit braucht, um aus diesen zu entkommen. In den vergangenen Jahren wurde sehr viel experimentiert, um einen sinnvollen Ausgleich zwischen „Gier“ und Zufälligkeit zu finden.

Einer der einfachsten und effektivsten Algorithmen, der aus all dieser Arbeit hervorgegangen ist, heißt WALKSAT (▶ Abbildung 7.18). Der Algorithmus wählt bei jedem Durchlauf eine nicht erfüllte Klausel aus und ein Symbol in der Klausel, das gewechselt werden soll. Er wählt zufällig zwischen zwei Methoden, wie das zu wechselnde Symbol ausgewählt wird: (1) über einen „min-conflicts“-Schritt, der die Anzahl nicht erfüllter Klauseln in dem neuen Zustand minimiert, und (2) über einen „zufälligen“ Schritt, der das Symbol zufällig auswählt.

```
function WALKSAT(clauses, p, max_flips)
                 returns ein zufriedenstellendes Modell oder einen Fehler
  inputs: clauses, eine Menge von Klauseln in Aussagenlogik
          p, Wahrscheinlichkeit für eine "Random Walk"-Bewegung,
          normalerweise bei 0,5
          max_flips, Anzahl der erlaubten Wechsel bis zum Aufgeben
  model ← eine zufällige Zuweisung von true/false an die Symbole in clauses
  for i = 1 to max_flips do
    if model erfüllt clauses then return model
    clause ← zufällig ausgewählte Klausel aus clauses, die in model falsch ist
    with probability p wechsle den Wert in model eines
                      zufällig aus clause ausgewählten Symbols
    else wechsle den Wert eines Symbols, welches in clause die Anzahl
         der erfüllten Klauseln maximiert
  return failure
```

Abbildung 7.18: Der WALKSAT-Algorithmus für die Überprüfung der Erfüllbarkeit durch zufälliges Wechseln der Variablenwerte. Es gibt viele Versionen dieses Algorithmus.

Wenn WalkSAT ein Modell zurückgibt, ist der Eingabesatz zwar erfüllbar, doch wenn der Algorithmus einen Fehler (*failure*) liefert, kann das zwei mögliche Ursachen haben: Entweder ist der Satz nicht erfüllbar oder wir müssen dem Algorithmus mehr Zeit zugestehen. Wenn wir $max\_flips = \infty$ und $p > 0$ setzen, gibt WalkSAT irgendwann ein Modell zurück (falls ein solches existiert), weil die Random-Walk-Schritte schließlich auf eine Lösung treffen. Leider terminiert der Algorithmus nie, wenn *max_flips* unendlich und der Satz nicht erfüllbar ist!

Aus diesem Grund ist WalkSAT am sinnvollsten, wenn zu erwarten ist, dass eine Lösung existiert – beispielsweise haben die in den *Kapiteln 3* und *6* beschriebenen Probleme normalerweise Lösungen. Andererseits kann WalkSAT nicht immer erkennen, ob ein Problem *unerfüllbar* ist, was erforderlich ist, um die logische Konsequenz zu erkennen. Zum Beispiel kann ein Agent mit WalkSAT nicht *zuverlässig* beweisen, dass ein Feld in der Wumpus-Welt sicher ist. Stattdessen kann er sagen: „Ich habe eine Stunde darüber nachgedacht und kann mir keine Welt vorstellen, in der das Feld *nicht* sicher wäre." Das kann ein guter empirischer Indikator sein, dass das Feld sicher ist, doch es ist zweifellos kein Beweis.

7.6.3 Die Landschaft zufälliger SAT-Probleme

Manche SAT-Probleme sind schwieriger als andere. *Einfache* Probleme lassen sich von jedem alten Algorithmus lösen, doch da wir wissen, dass SAT-Probleme NP-vollständig sind, müssen wenigstens einige Probleminstanzen eine exponentielle Laufzeit erfordern. In *Kapitel 6* haben wir einige interessante Entdeckungen zu bestimmten Arten von Problemen gemacht. Zum Beispiel hat sich das *n*-Damen-Problem – von dem man dachte, es sei sehr kompliziert für Backtracking-Suchalgorithmen – als überaus einfach für lokale Suchmethoden wie z.B. Min-Conflicts herausgestellt. Das liegt daran, dass die Lösungen im Zuweisungsraum sehr dicht verteilt sind und es für jede anfängliche Zuweisung garantiert irgendwo in der Nähe eine Lösung gibt. Damit ist das *n*-Damen-Problem einfach, weil es **unterbeschränkt** ist.

Wenn wir Erfüllbarkeitsprobleme in konjunktiver Normalform betrachten, ist ein unterbeschränktes Problem ein Problem mit relativ *wenigen* Klauseln, die die Variablen beschränken. Zum Beispiel zeigt der folgende Ausdruck einen zufällig erzeugten 3-KNF-Satz mit fünf Symbolen und fünf Klauseln:

$$(\neg D \vee \neg B \vee C) \wedge (B \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee E)$$
$$\wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C).$$

Von den 32 möglichen Zuweisungen handelt es sich bei 16 um Modelle dieses Satzes; es wären also lediglich zwei zufällige Auswahlen notwendig, um ein Modell zu finden. Wie die meisten unterbeschränkten Probleme ist dies ein einfaches Erfüllbarkeitsproblem. Andererseits weist ein *überbeschränktes* Problem viele Klauseln bezogen auf die Anzahl der Variablen auf und hat möglicherweise keine Lösungen.

Um über diese grundlegenden Vermutungen hinauszugehen, müssen wir genau definieren, wie zufällige Sätze erzeugt werden. Die Notation $CNF_k(m, n)$ kennzeichnet einen *k*-KNF-Satz mit *m* Klauseln und *n* Symbolen, wobei die Klauseln gleichmäßig, unabhängig und ersatzlos unter allen Klauseln mit *k* verschiedenen – positiven oder negativen – Literalen zufällig ausgewählt werden. (Ein Symbol darf nicht zweimal in einer Klausel erscheinen, noch darf eine Klausel zweimal in einem Satz auftreten.)

Für eine gegebene Quelle zufälliger Sätze können wir die Wahrscheinlichkeit der Erfüllbarkeit messen. ▶ Abbildung 7.19(a) stellt die Wahrscheinlichkeit für $CNF_3(m, 50)$, d.h. Sätze mit 50 Variablen und 3 Literalen pro Klausel, als Funktion des Klausel/Symbol-Verhältnisses m/n dar. Wie zu erwarten, liegt die Wahrscheinlichkeit der Erfüllbarkeit für kleine m/n nahe 1, für große m/n ist sie fast 0. Die Wahrscheinlichkeit fällt bei $m/n = 4{,}3$ steil ab. Empirisch finden wir, dass die „Klippe" ungefähr an der gleichen Stelle bleibt (für $k = 3$) und mit zunehmendem n steiler wird. Theoretisch besagt das **Schwellenwert-Phänomen der Erfüllbarkeit**, dass es für jedes $k \geq 3$ ein Schwellenverhältnis r_k gibt, sodass für n unendlich die Wahrscheinlichkeit, dass $CNFk(n, rn)$ erfüllbar ist, für alle Werte von r unterhalb des Schwellenwertes zu 1 und für alle Werte darüber zu 0 wird. Das Phänomen bleibt unbewiesen.

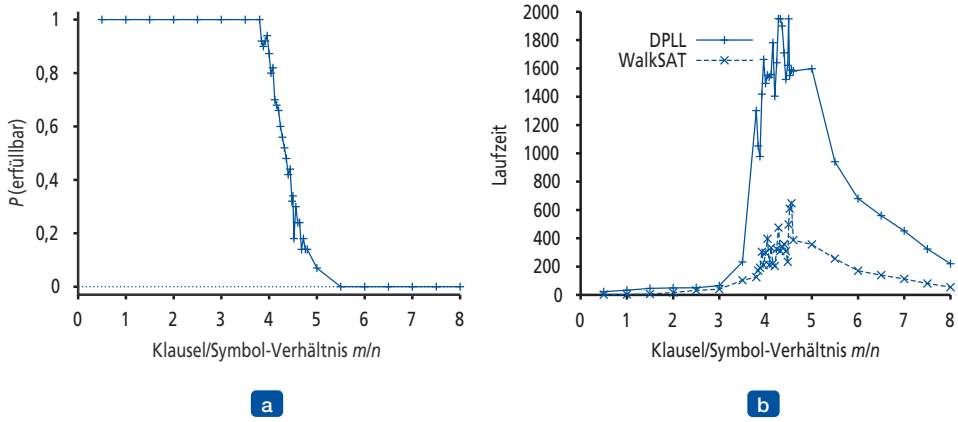


Abbildung 7.19: (a) Graph, der die Wahrscheinlichkeit zeigt, dass ein zufälliger 3-KNF-Satz mit $n = 50$ Symbolen erfüllbar ist, dargestellt als Funktion des Klausel/Symbol-Verhältnisses m/n. (b) Graph der mittleren Laufzeit (gemessen als Anzahl der rekursiven Aufrufe von DPLL, eine gute Näherung) für zufällige 3-KNF-Sätze. Die schwierigsten Probleme weisen ein Klausel/Symbol-Verhältnis von etwa 4/3 auf.

Nachdem wir eine gute Vorstellung davon haben, was erfüllbare und nicht erfüllbare Probleme sind, stellt sich als Nächstes die Frage, wie schwierige Probleme aussehen. Es zeigt sich, dass sie oftmals ebenfalls beim Schwellwert liegen. ▶ Abbildung 7.19(b) zeigt, dass Probleme mit 50 Symbolen beim Schwellwert von 4,3 rund 20-mal schwieriger zu lösen sind als diejenigen mit einem Verhältnis von 3,3. Die unterbeschränkten Probleme sind am einfachsten zu lösen (da es so leicht ist, eine Lösung zu raten); die überbeschränkten Probleme sind nicht so leicht wie die unterbeschränkten, aber immer noch wesentlich einfacher als diejenigen, die unmittelbar beim Schwellwert angesiedelt sind.

7.7 Agenten auf der Basis von Aussagenlogik

In diesem Abschnitt bringen wir das zusammen, was wir bisher gelernt haben, um für die Wumpus-Welt Agenten zu konstruieren, die mit Aussagenlogik arbeiten. Im ersten Schritt versetzen wir den Agenten in die Lage, den Zustand der Welt anhand seiner bisherigen Wahrnehmungen – bis zum möglichen Umfang – abzuleiten. Dazu ist es erforderlich, ein vollständiges logisches Modell der Wirkungen von Aktionen zu schreiben. Außerdem zeigen wir, wie der Agent die Welt effizient verfolgen kann, ohne für jede Infe-

renz in der Wahrnehmungsgeschichte zurückgehen zu müssen. Schließlich zeigen wir, wie der Agent mit logischer Inferenz Pläne konstruieren kann, mit denen er seine Ziele garantiert erreicht.

7.7.1 Der aktuelle Zustand der Welt

Wie bereits zu Beginn des Kapitels festgestellt, operiert ein logischer Agent, indem er aus einer Wissensbasis von Sätzen über die Welt herleitet, was zu tun ist. Diese Wissensbasis besteht aus Axiomen – generellem Wissen darüber, wie die Welt funktioniert – und Wahrnehmungssätzen, die aus der Erfahrung des Agenten in einer bestimmten Welt erhalten werden. In diesem Abschnitt konzentrieren wir uns auf das Problem, den aktuellen Zustand der Wumpus-Welt herzuleiten – wo er sich befindet, ob das Feld sicher ist usw.

In *Abschnitt 7.4.3* haben wir begonnen, Axiome zu sammeln. Der Agent weiß, dass das Ausgangsfeld keine Falle ($\neg P_{1,1}$) und keinen Wumpus ($\neg W_{1,1}$) enthält. Darüber hinaus ist ihm für jedes Feld bekannt, dass es nur dann windig ist, wenn es auf einem benachbarten Feld eine Falle gibt, und auf einem Feld nur dann ein Gestank wahrnehmbar ist, wenn sich auf einem benachbarten Feld ein Wumpus befindet. Somit können wir eine große Sammlung von Sätzen der folgenden Form einbinden:

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$
$$S_{1,1} \Leftrightarrow (W_{1,2} \vee W_{2,1})$$
$$\ldots$$

Außerdem weiß der Agent, dass genau ein Wumpus existiert. Dies wird in zwei Teilen ausgedrückt. Zuerst müssen wir sagen, dass es *mindestens einen* Wumpus gibt:

$$W_{1,1} \vee W_{1,2} \vee \ldots \vee W_{4,3} \vee W_{4,4}.$$

Dann müssen wir sagen, dass es *höchstens einen* Wumpus gibt. Für jedes Feldpaar fügen wir einen Satz hinzu, der besagt, dass mindestens eines von ihnen Wumpus-frei sein muss:

$$\neg W_{1,1} \vee \neg W_{1,2}$$
$$\neg W_{1,1} \vee \neg W_{1,3}$$
$$\ldots$$
$$\neg W_{4,3} \vee \neg W_{4,4}.$$

So weit, so gut. Sehen wir uns nun die Wahrnehmungen des Agenten an. Wenn es momentan einen Gestank gibt, könnte man annehmen, dass der Wissensbasis eine Aussage *Stench* hinzuzufügen ist. Das ist aber nicht ganz richtig: War im vorherigen Zeitschritt kein Gestank wahrzunehmen, würde $\neg$*Stench* bereits zugesichert sein und die neue Behauptung würde einfach in einem Widerspruch resultieren. Das Problem lässt sich lösen, wenn wir erkennen, dass eine Wahrnehmung *nur etwas über die aktuelle Zeit* behauptet. Wenn also der Zeitschritt (wie an MAKE-PERCEPT-SENTENCE in Abbildung 7.1 übergeben) 4 ist, fügen wir der Wissensbasis $Stench^4$ und nicht *Stench* hinzu – um geschickt jeglichen Widerspruch mit $\neg Stench^3$ zu vermeiden. Das Gleiche gilt für die Wahrnehmungen *Breeze*, *Bump*, *Glitter* und *Scream*.

Das Konzept, Aussagen mit Zeitschritten zu verbinden, erstreckt sich auf jeden Aspekt der Welt, der sich mit der Zeit ändert. Zum Beispiel umfasst die anfängliche Wissens-

basis $L^0_{1,1}$ – der Agent befindet sich im Feld [1, 1] zur Zeit 0 – sowie $FacingEast^0$, $HaveArrow^0$ und $WumpusAlive^0$. Einen Aspekt der Welt, der sich ändert, bezeichnen wir als **Fluent** (vom Lateinischen *fluens*, fließend). Analog zur Diskussion über faktorisierte Darstellungen in *Abschnitt 2.4.7* ist „Fluent" ein Synonym für „Zustandsvariable". Symbole, die permanenten Aspekten der Welt zugeordnet sind, brauchen keinen hochgestellten Zeitindex und werden auch als **zeitlose Variablen** bezeichnet.

Die Wahrnehmungen von Gestank und Luftzug können wir wie folgt mit den Eigenschaften der Felder, wo sie auftauchen, direkt über den örtlichen Fluenten verbinden.[10] Für jeden Zeitschritt t und jedes Feld $[x, y]$ behaupten wir:

$$L^t_{x,y} \Rightarrow (Breeze^t \Leftrightarrow B_{x,y})$$
$$L^t_{x,y} \Rightarrow (Stench^t \Leftrightarrow S_{x,y}).$$

Natürlich brauchen wir jetzt Axiome, mit denen der Agent Fluenten wie zum Beispiel $L^t_{x,y}$ verfolgen kann. Diese Fluenten ändern sich als Ergebnis von Aktionen, die der Agent unternimmt. Deshalb müssen wir – in der Terminologie von *Kapitel 3* – das **Übergangsmodell** der Wumpus-Welt als Menge von logischen Sätzen formulieren.

Zuerst brauchen wir Aussagensymbole für die auftretenden Aktionen. Wie bei Wahrnehmungen werden diese Symbole mit einem Zeitindex versehen; somit bedeutet $Forward^0$, dass der Agent die Aktion *Forward* zur Zeit 0 ausführt. Per Konvention tritt die Wahrnehmung für einen gegebenen Zeitschritt zuerst auf, gefolgt von der Aktion für diesen Zeitschritt, woran sich ein Übergang zum nächsten Zeitschritt anschließt.

Um zu beschreiben, wie sich die Welt ändert, können wir versuchen, **Wirkungsaxiome** zu schreiben, die das Ergebnis einer Aktion beim nächsten Zeitschritt spezifizieren. Wenn sich zum Beispiel der Agent zur Zeit 0 an Position [1, 1] befindet, nach Osten blickt und vorwärts geht (Aktion *Forward*), ist das Ergebnis, dass der Agent auf Feld [2, 1] steht und sich nicht mehr in [1, 1] aufhält:

$$L^0_{1,1} \wedge FacingEast^0 \wedge Forward^0 \Rightarrow (L^1_{2,1} \wedge \neg L^1_{1,1}). \tag{7.1}$$

Ein derartiger Satz ist für jeden möglichen Zeitschritt, für jedes der 16 Felder und für jede der vier Richtungen erforderlich. Außerdem brauchen wir ähnliche Sätze für die anderen Aktionen: *Grab*, *Shoot*, *Climb*, *TurnLeft* und *TurnRight*.

Angenommen, der Agent entscheidet sich, zur Zeit 0 vorwärts zu gehen, und behauptet diesen Fakt in seiner Wissensbasis. Indem der Agent das Wirkungsaxiom in Gleichung (7.1) mit den anfänglichen Zusicherungen über den Zustand zur Zeit 0 kombiniert, kann er jetzt schließen, dass er sich in [2, 1] befindet. Das heißt, ASK(KB, $L^1_{2,1}$) = *true*. So weit, so gut. Leider gibt es von anderen Stellen keine so guten Neuigkeiten zu vermelden: Auf ASK(KB, $HaveArrow^1$) hin erhalten wir die Antwort *false*, d.h., der Agent kann weder beweisen, dass er den Pfeil noch besitzt, noch dass er ihn *nicht* besitzt! Die Information ist verlorengegangen, weil das Wirkungsaxiom nicht in der Lage ist festzustellen, was als Ergebnis einer Aktion *unverändert* bleibt. Die Notwendigkeit, dies zu tun, führt zum **Rahmenproblem**.[11] Als mögliche Lösung für das Rahmenproblem könnte

10 *Abschnitt 7.4.3* hat diese Anforderung praktischerweise übergangen.

11 Der Name „Rahmenproblem" (engl. Frame Problem) kommt von „Frame of Reference" (Bezugssystem/Referenzumgebung) in der Physik – dem angenommenen stationären Hintergrund, bezüglich dessen Bewegungen gemessen werden. Außerdem gibt es eine Analogie zu den Frames (Einzelbildern) eines Filmes, wobei normalerweise der größte Teil des Hintergrundes konstant bleibt, während Änderungen im Vordergrund stattfinden.

man **Rahmenaxiome** hinzufügen, die explizit sämtliche Aussagen zusichern, die gleich bleiben. Zum Beispiel würden wir für jede Zeit t die folgenden Sätze haben:

$$Forward^t \Rightarrow (HaveArrow^t \Leftrightarrow HaveArrow^{t+1})$$
$$Forward^t \Rightarrow (WumpusAlive^t \Leftrightarrow WumpusAlive^{t+1})$$
$$\ldots$$

Hier erwähnen wir explizit jede Aussage, die von Zeit t zu Zeit $t+1$ unter der Aktion *Forward* unverändert bleibt. Obwohl der Agent jetzt weiß, dass er immer noch den Pfeil besitzt, nachdem er vorwärts gegangen ist, und dass der Wumpus nicht getötet wurde oder wiederauferstanden ist, scheint die ausufernde Anzahl von Rahmenaxiomen bemerkenswert ineffizient. In einer Welt mit m unterschiedlichen Aktionen und n Fluenten hat die Menge der Rahmenaxiome eine Größe von $O(mn)$. Diese spezifische Manifestation des Rahmenproblems wird auch als **repräsentationelles Rahmenproblem** bezeichnet. Aus historischer Sicht war das Problem für KI-Forscher von erheblicher Bedeutung; wir erläutern es näher in den Hinweisen am Ende des Kapitels.

Das repräsentationelle Rahmenproblem ist deshalb wichtig, weil die reale Welt gelinde gesagt sehr viele Fluenten aufweist. Erfreulich für uns Menschen ist, dass jede Aktion in der Regel nicht mehr als eine kleine Anzahl k dieser Fluenten ändert – die Welt trägt **Lokalität** zur Schau. Um das repräsentationelle Rahmenproblem zu lösen, ist es erforderlich, das Übergangsmodell mit einer Menge von Axiomen der Größe $O(mk)$ statt $O(mn)$ zu definieren. Außerdem gibt es ein **inferentielles Rahmenproblem**: das Problem, die Ergebnisse eines t-Zeitschrittplanes der Aktion in der Zeit $O(kt)$ statt $O(nt)$ vorwärts zu projizieren.

Um das Problem zu lösen, konzentriert man sich nicht mehr darauf, Axiome über Aktionen zu schreiben, sondern Axiome über Fluenten zu formulieren. Somit haben wir für jeden Fluent F ein Axiom, das den Wahrheitswert von F^{t+1} als Fluenten (einschließlich des Fluenten F selbst) zur Zeit t definiert und die Aktionen, die zur Zeit t aufgetreten sein können. Jetzt lässt sich der Wahrheitswert von F^{t+1} nach zwei Verfahren festlegen: Entweder bewirkt die Aktion zur Zeit t, dass F zu $t+1$ gleich *true* ist, oder F war bereits *true* zur Zeit t und die Aktion zur Zeit t bewirkt nicht, dass es *false* wird. Ein Axiom in dieser Form wird als **Nachfolgerzustands-Axiom** bezeichnet und hat folgendes Schema:

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t).$$

Eines der einfachsten Nachfolgerzustands-Axiome ist das für *HaveArrow*. Da es keine Aktion für Neuladen gibt, verschwindet der Teil $ActionCausesF^t$ und es bleibt

$$HaveArrow^{t+1} \Leftrightarrow (HaveArrow^t \wedge \neg Shoot^t). \tag{7.2}$$

Für die Position des Agenten sehen die Nachfolgerzustands-Axiome wesentlich umfangreicher aus. Zum Beispiel ist ${L^{t+1}}_{1,1}$ gleich *true*, wenn der Agent entweder (a) von [1, 2] vorwärts geht, wenn er nach Süden sieht, oder von [2, 1], wenn er nach Westen blickt; oder (b) ${L^t}_{1,1}$ war bereits *true* und die Aktion hat nicht zu einer Bewegung geführt (entweder weil die Aktion keine *Forward*-Aktion war oder weil die Aktion zu einem Stoß an eine Wand geführt hat). In Aussagenlogik formuliert, wird dies zu

$$\begin{aligned} {L^{t+1}}_{1,1} \Leftrightarrow\ & ({L^t}_{1,1} \wedge (\neg Forward^t \vee Bump^{t+1})) \\ & \vee ({L^t}_{1,2} \wedge (South^t \wedge Forward^t)) \\ & \vee ({L^t}_{2,1} \wedge (West^t \wedge Forward^t)). \end{aligned} \tag{7.3}$$

In Übung 7.26 besteht Ihre Aufgabe darin, Axiome für die Fluenten der Wumpus-Welt zu schreiben.

Für einen gegebenen vollständigen Satz von Nachfolgerzustands-Axiomen und die anderen Axiome, die am Anfang dieses Abschnittes aufgelistet wurden, ist der Agent in der Lage, eine Frage zu stellen (ASK) und jede beantwortbare Frage über den aktuellen Zustand der Welt zu beantworten. Zum Beispiel sieht die Anfangssequenz der Wahrnehmungen und Aktionen in *Abschnitt 7.2* wie folgt aus:

$$\begin{aligned}
&\neg Stench^0 \wedge \neg Breeze^0 \wedge \neg Glitter^0 \wedge \neg Bump^0 \wedge \neg Scream^0 ; Forward^0 \\
&\neg Stench^1 \wedge Breeze^1 \wedge \neg Glitter^1 \wedge \neg Bump^1 \wedge \neg Scream^1 ; TurnRight^1 \\
&\neg Stench^2 \wedge Breeze^2 \wedge \neg Glitter^2 \wedge \neg Bump^2 \wedge \neg Scream^2 ; TurnRight^2 \\
&\neg Stench^3 \wedge Breeze^3 \wedge \neg Glitter^3 \wedge \neg Bump^3 \wedge \neg Scream^3 ; Forward^3 \\
&\neg Stench^4 \wedge \neg Breeze^4 \wedge \neg Glitter^4 \wedge \neg Bump^4 \wedge \neg Scream^4 ; TurnRight^4 \\
&\neg Stench^5 \wedge \neg Breeze^5 \wedge \neg Glitter^5 \wedge \neg Bump^5 \wedge \neg Scream^5 ; Forward^5 \\
&Stench^6 \wedge \neg Breeze^6 \wedge \neg Glitter^6 \wedge \neg Bump^6 \wedge \neg Scream^6
\end{aligned}$$

An diesem Punkt liefert ASK(KB, ${L^6}_{1,2}$) = *true*, sodass der Agent weiß, wo er sich befindet. Darüber hinaus ergeben ASK(KB, $W_{1,3}$) = *true* und ASK(KB, $P_{3,1}$) = *true*, sodass der Agent den Wumpus und eine der Falltüren gefunden hat. Die wichtigste Frage für den Agenten ist, ob er sich gefahrlos auf ein Feld begeben kann, d.h. das Feld weder eine Falltür noch einen lebenden Wumpus enthält. Es ist zweckmäßig, hierfür Axiome hinzuzufügen, die folgende Form haben:

$$OK^t_{x,y} \Leftrightarrow \neg P_{x,y} \wedge \neg(W_{x,y} \wedge WumpusAlive^t).$$

Schließlich liefert ASK(KB, ${OK^6}_{2,2}$) = *true*, sodass sich der Agent gefahrlos auf das Feld [2, 2] bewegen kann. Mit einem korrekten und vollständigen Inferenzalgorithmus wie zum Beispiel DPLL kann der Agent in der Tat jede beantwortbare Frage, welche Felder OK sind, beantworten – und das in nur wenigen Millisekunden für kleine bis mittlere Wumpus-Welten.

Die Lösung der repräsentationellen und inferentiellen Rahmenprobleme ist ein großer Schritt vorwärts, doch bleibt ein übles Problem: Wir müssen bestätigen, dass für eine Aktion *alle* notwendigen Vorbedingungen gelten, damit sie die beabsichtigte Wirkung hat. Wir haben festgestellt, dass die *Forward*-Aktion den Agenten nach vorn bringt, sofern keine Wand im Wege steht. Doch es gibt viele andere ungewöhnliche Ausnahmen, die zum Scheitern der Aktion führen: Der Agent kann unterwegs stürzen, einen Herzinfarkt erleiden, von Riesenfledermäusen fortgetragen werden usw. Es handelt sich um ein sogenanntes **Qualifizierungsproblem**, alle diese Ausnahmen zu spezifizieren. Es gibt keine vollständige Lösung innerhalb der Logik; Systemdesigner brauchen Fingerspitzengefühl, um zu entscheiden, wie detailliert sie beim Spezifizieren ihres Modells sein möchten und welche Details sie auslassen möchten. *Kapitel 13* zeigt, dass es mithilfe der Wahrscheinlichkeitstheorie möglich ist, alle Ausnahmen zusammenzufassen, ohne sie explizit zu benennen.

7.7.2 Ein hybrider Agent

Die Fähigkeit, verschiedene Aspekte des Weltzustandes herzuleiten, lässt sich relativ einfach mit Bedingungs-Aktions-Regeln und mit problemlösenden Algorithmen aus den *Kapiteln 3* und *4* kombinieren, um einen **hybriden Agenten** für die Wumpus-Welt zu erzeugen. ▶ Abbildung 7.20 zeigt eine mögliche Vorgehensweise.

```
function HYBRID-WUMPUS-AGENT(percept) returns eine Aktion
  inputs: percept, eine Liste, [stench,breeze,glitter,bump,scream]
  persistent: KB, eine Wissensbasis, anfangs die zeitlose "Wumpus-Physik"
              t, ein Zähler, anfangs 0, als Zeitanzeige
              plan, eine Aktionsfolge, anfangs leer

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  TELL der KB die zeitlichen "Physik"-Sätze für Zeit t
  safe ← {[x, y] : ASK(KB, OK^t_x,y) = true}
  if ASK(KB, Glitter^t) = true then
     plan ← [Grab] + PLAN-ROUTE(current, {[1,1]}, safe) + [Climb]
  if plan ist leer then
     unvisited ← {[x, y] : ASK(KB, L^t'_x,y) = false für alle t' ≤ t}
     plan ← PLAN-ROUTE(current, unvisited ∩ safe, safe)
  if plan ist leer und ASK(KB, HaveArrow^t) = true then
     possible_wumpus ← {[x, y] : ASK(KB, ¬W_x,y) = false}
     plan ← PLAN-SHOT(current, possible_wumpus, safe)
  if plan ist leer then // keine Wahl, außer ein Risiko einzugehen
     not_unsafe ← {[x, y] : ASK(KB, ¬OK^t_x,y) = false}
     plan ← PLAN-ROUTE(current, unvisited ∩ not_unsafe, safe)
  if plan is empty then
     plan ← PLAN-ROUTE(current, {[1, 1]}, safe) + [Climb]
  action ← POP(plan)
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action

---

function PLAN-ROUTE(current,goals,allowed) returns eine Aktionsfolge
  inputs: current, die aktuelle Position des Agenten
          goals, eine Menge von Feldern; es wird versucht, eine Route zu
                 einem von ihnen zu planen
          allowed, eine Menge von Feldern, die einen Teil der Route bilden können
  problem ← ROUTE-PROBLEM(current, goals, allowed)
  return A*-GRAPH-SEARCH(problem)
```

Abbildung 7.20: Programm eines hybriden Agenten für die Wumpus-Welt. Es verwendet eine aussagenlogische Wissensbasis, um auf den Zustand der Welt zu schließen, und entscheidet mithilfe einer Kombination von problemlösender Suche und domänenspezifischem Code, welche Aktionen auszuführen sind.

Das Agentenprogramm verwaltet und aktualisiert eine Wissensbasis sowie einen aktuellen Plan. Die anfängliche Wissensbasis enthält die *zeitlosen* Axiome – diejenigen, die nicht von t abhängen, wie zum Beispiel das Axiom, das einen Luftzug auf einem Feld mit der Anwesenheit einer Falltür verknüpft. Bei jedem Zeitschritt wird der neue Wahrnehmungssatz hinzugefügt, und zwar zusammen mit allen Axiomen, die von t abhängen, wie zum Beispiel die Nachfolgerzustands-Axiome. (Der nächste Abschnitt erläutert, warum der Agent für *zukünftige* Zeitschritte keine Axiome benötigt.) Dann

verwendet der Agent logische Inferenz, d.h., er fragt per ASK die Wissensbasis ab, um herauszufinden, welche Felder sicher sind und welche noch besucht werden müssen.

Der Hauptteil des Agentenprogramms konstruiert einen Plan basierend auf einer fallenden Priorität von Zielen. Gibt es ein Glitzern, konstruiert das Programm einen Plan, um das Gold aufzunehmen, einer Route zurück zum Startfeld zu folgen und aus der Höhle auszusteigen. Wenn es andernfalls keinen aktuellen Plan gibt, plant das Programm eine Route zum nächsten sicheren Feld, das noch nicht besucht wurde, wobei gewährleistet wird, dass die Route nur durch sichere Felder führt. Die Routenplanung erfolgt mit A*-Suche und nicht mit ASK. Gibt es keine sicheren Quadrate zu erkunden, versucht der Agent – falls er noch einen Pfeil besitzt – im nächsten Schritt, ein sicheres Quadrat zu schaffen, indem er auf eine der möglichen Wumpus-Positionen schießt. Diese werden durch Abfragen ermittelt, wo ASK(KB, $\neg W_{x,y}$) falsch ist – d.h., wo *nicht* bekannt ist, ob sich dort *kein* Wumpus befindet. Die (hier nicht gezeigte) Funktion PLAN-SHOT plant mithilfe von PLAN-ROUTE eine Aktionssequenz, die diesen Schuss realisiert. Scheitert dies, sucht das Programm nach einem zu untersuchenden Feld, das nicht nachweisbar unsicher ist – d.h. ein Feld, für das ASK(KB, $\neg {OK^t}_{x,y}$) falsch zurückgibt. Wenn es kein derartiges Feld gibt, ist die Mission unmöglich. Der Agent zieht sich dann auf [1, 1] zurück und steigt aus der Höhle.

7.7.3 Logische Zustandsschätzung

Das in Abbildung 7.20 gezeigte Agentenprogramm funktioniert prinzipiell gut, weist aber eine entscheidende Schwäche auf: Mit ablaufender Zeit steigt der Berechnungsaufwand, der mit den Aufrufen von ASK verbunden ist, immer weiter. Das passiert hauptsächlich deshalb, weil die erforderlichen Inferenzen zeitlich immer weiter zurückgehen und immer mehr Aussagensymbole einbeziehen müssen. Offenbar ist dies nicht haltbar – wir können keinen Agenten gebrauchen, dessen Verarbeitungszeit für jede Wahrnehmung proportional zur Länge seines Lebens wächst! Was wir wirklich brauchen, ist eine *konstante* Aktualisierungszeit – d.h. Unabhängigkeit von t. Die offensichtliche Antwort ist es, die Ergebnisse der Inferenz zu speichern oder in einem **Cache** (Zwischenspeicher) abzulegen, sodass der Inferenzprozess beim nächsten Zeitschritt auf den Ergebnissen früherer Zeitschritte aufbauen kann, anstatt wieder von Grund auf neu beginnen zu müssen.

Wie *Abschnitt 4.4* gezeigt hat, können die Vorgeschichte der Wahrnehmungen und ihre sämtlichen Konsequenzen durch den **Belief State** ersetzt werden – d.h. durch eine Darstellung der Menge aller möglichen Zustände der Welt.[12] Die Aktualisierung des Belief State beim Eintreffen neuer Wahrnehmungen wird als **Zustandsschätzung** bezeichnet. Während es sich beim Belief State in *Abschnitt 4.4* um eine explizite Liste von Zuständen gehandelt hat, können wir hier einen logischen Satz mit den Aussagensymbolen, die mit dem aktuellen Zeitschritt verbunden sind, sowie die zeitlosen Symbole verwenden. Zum Beispiel stellt der logische Satz

$$WumpusAlive^1 \wedge {L^1}_{2,1} \wedge B_{2,1} \wedge (P_{3,1} \vee P_{2,2}) \tag{7.4}$$

12 Die Wahrnehmungsgeschichte selbst können wir uns als eine Darstellung des Belief State vorstellen, allerdings eines Belief State, der Inferenz zunehmend teurer macht, je länger die Geschichte wird.

die Menge aller Zustände zur Zeit 1 dar, zu der der Wumpus lebt, der Agent auf dem Feld [2, 1] steht, das Feld windig ist und es in [3, 1] und/oder [2, 2] eine Falltür gibt.

Es ist allerdings nicht ganz einfach, einen genauen Belief State als logische Formel aufrechtzuerhalten. Bei n Fluent-Symbolen für die Zeit t gibt es 2^n mögliche Zustände – d.h. Zuweisungen von Wahrheitswerten zu diesen Symbolen. Die Menge der Belief States ist nun die Potenzmenge (Menge aller Teilmengen) der Menge der physischen Zustände. Da 2^n physische Zustände existieren, gibt es 2^{2^n} Belief States. Selbst wenn wir die kompakteste Kodierung logischer Formeln verwenden würden, bei der jeder Belief State durch eine eindeutige Binärzahl darstellt wird, wären Zahlen mit $\log_2(2^{2^n}) = 2^n$ Bit erforderlich, um den aktuellen Belief State zu bezeichnen. Eine genaue Zustandsschätzung erfordert also logische Formeln, deren Größe exponentiell mit der Anzahl der Symbole zunimmt.

Ein sehr gebräuchliches und natürliches Schema für die Zustandsschätzung ist es, Belief States als Konjunktionen von Literalen darzustellen, d.h. durch 1-KNF-Formeln. Dazu versucht das Agentenprogramm einfach, X^t und $\neg X^t$ für jedes Symbol X^t (sowie jedes zeitlose Symbol, dessen Wahrheitswert noch nicht bekannt ist) für den gegebenen Belief State bei $t-1$ zu beweisen. Die Konjunktion der n beweisbaren Literale wird zum neuen Belief State und der vorherige Belief State wird verworfen.

Tipp

Wichtig ist zu verstehen, dass dieses Schema mit der Zeit einige Informationen verlieren kann. Wenn zum Beispiel der Satz in Gleichung (7.4) der wahre Belief State wäre, würden weder $P_{3,1}$ noch $P_{2,2}$ einzeln beweisbar sein und keiner würde im 1-KNF-Belief-State erscheinen. (Übung 7.27 untersucht eine mögliche Lösung für dieses Problem.) Da andererseits jedes Literal im 1-KNF-Belief-State aus dem vorherigen Belief State bewiesen worden ist und der anfängliche Belief State eine wahre Behauptung ist, wissen wir, dass der gesamte 1-KNF-Belief-State wahr sein muss. Somit *umfasst die Menge der möglichen Zustände, die durch den 1-KNF-Belief-State dargestellt werden, sämtliche Zustände, die bei gegebener vollständiger Wahrnehmungsgeschichte überhaupt möglich sind.* Wie ▶ Abbildung 7.21 zeigt, fungiert der 1-KNF-Belief-State als einfache Außenhülle – oder **konservative Näherung** – um den exakten Belief State. Diesem Konzept der konservativen Näherungen für komplizierte Mengen begegnen wir in vielen Bereichen der KI.

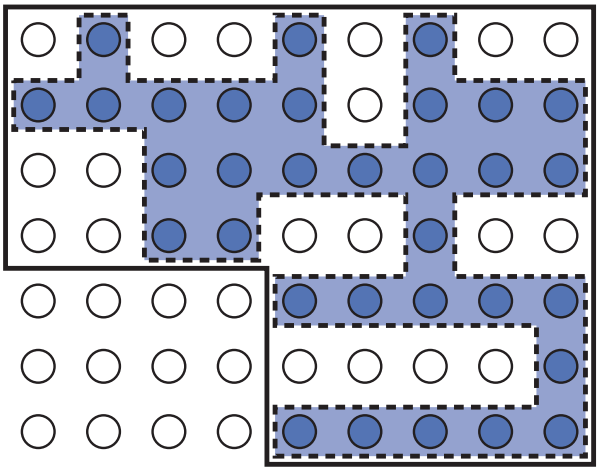


Abbildung 7.21: Darstellung eines 1-KNF-Belief-State (fette Umrisslinie) als einfach darstellbare, konservative Näherung zum exakten (unstetigen) Belief State (schattierter Bereich mit gestrichelter Umrisslinie). Jede mögliche Welt wird als Kreis gezeichnet; die grauen Kreise sind mit allen Wahrnehmungen konsistent.

7.7.4 Pläne durch aussagenlogische Inferenz erstellen

Der Agent in Abbildung 7.20 bestimmt mithilfe logischer Inferenz, welche Felder sicher sind, verwendet aber die A*-Suche, um Pläne zu erstellen. In diesem Abschnitt zeigen wir, wie sich Pläne durch logische Inferenz erzeugen lassen.

Die Grundidee ist recht einfach:

1 Konstruiere einen Satz, der folgende Elemente einbindet:

a. $Init^0$, eine Sammlung von Zusicherungen über den Ausgangszustand;

b. $Transition^1$, ..., $Transition^t$, die Nachfolgerzustands-Axiome für alle möglichen Aktionen zu jedem Zeitpunkt bis zur maximalen Zeit t;

c. die Zusicherung, dass das Ziel zur Zeit t erreicht wird: $HaveGold^t \wedge ClimbedOut^t$.

2 Präsentiere den gesamten Satz einem SAT-Solver. Wenn der Solver ein zufriedenstellendes Modell findet, ist das Ziel erreichbar; ist der Satz unbefriedigend, lässt sich das Problem nicht planen.

3 Wenn ein Modell gefunden wurde, extrahiere aus dem Modell diejenigen Variablen, die Aktionen darstellen und denen *true* zugewiesen wurde. Zusammen verkörpern sie einen Plan, um die Ziele zu erreichen.

▶ Abbildung 7.22 zeigt SATPLAN, eine aussagenlogische Planungsprozedur. Sie implementiert die eben angegebenen Grundideen mit einem Kniff. Da der Agent nicht weiß, wie viele Schritte er benötigt, um das Ziel zu erreichen, probiert der Algorithmus jede mögliche Anzahl von Schritten t bis zu einer denkbar größten Planlänge T_{max} aus. Somit findet er garantiert den kürzesten Plan, falls ein solcher existiert. Durch die Art und Weise, wie SATPLAN nach einer Lösung sucht, lässt sich dieser Ansatz in einer partiell beobachtbaren Umgebung einsetzen; SATPLAN würde lediglich die nicht beobachtbaren Variablen auf die Werte setzen, die der Algorithmus benötigt, um eine Lösung zu erzeugen.

```
function SATPlan(init, transition, goal, T_max) returns Lösung oder Fehler
  inputs: init, transition, goal, bilden eine Beschreibung des Problems
          T_max, eine Obergrenze für die Planlänge
  for t = 0 to T_max do
  cnf ← Translate-To-Sat(init, transition, goal, t)
  model ← Sat-Solver (cnf)
  if model ist nicht null then
    return Extract-Solution(model)
  return failure
```

Abbildung 7.22: Der SATPLAN-Algorithmus. Das Planungsproblem wird in einen KNF-Satz übersetzt, in dem zugesichert wird, dass das Ziel zu einem festen Zeitschritt t erreicht ist, und für jeden Zeitschritt bis t werden Axiome eingebunden. Wenn der Erfüllbarkeitsalgorithmus ein Modell findet, wird ein Plan extrahiert, indem er diejenigen Aussagensymbole betrachtet, die sich auf Aktionen beziehen und denen im Modell *true* zugewiesen wurde. Existiert kein Modell, wiederholt sich der Vorgang, wobei das Ziel auf einen Schritt später verschoben wird.

Der entscheidende Schritt bei der Verwendung von SATPLAN ist die Konstruktion der Wissensbasis. Bei flüchtiger Betrachtung mag es scheinen, dass die Axiome der Wumpus-Welt in *Abschnitt 7.7.1* für die obigen Schritte 1(a) und 1(b) nicht ausreichen. Allerdings gibt es einen beträchtlichen Unterschied zwischen den Anforderungen an logische Konsequenz (wie durch ASK getestet) und denjenigen für Erfüllbarkeit. Betrachten Sie zum Beispiel die anfängliche Position des Agenten [1, 1] und nehmen Sie an, dass

das anspruchslose Ziel des Agenten darin besteht, zur Zeit 1 in [2, 1] zu sein. Die anfängliche Wissensbasis enthält $L^0_{1,1}$ und das Ziel ist $L^1_{2,1}$. Mithilfe von ASK können wir $L^1_{2,1}$ beweisen, wenn $Forward^0$ zugesichert wird, und ohne Weiteres können wir $L^1_{2,1}$ nicht beweisen, wenn stattdessen zum Beispiel $Shoot^0$ zugesichert wird. Jetzt wird SATPLAN den Plan [$Forward^0$] finden; so weit, so gut. Leider findet SATPLAN auch den Plan [$Shoot^0$]. Wie konnte das passieren? Um das herauszufinden, untersuchen wir das Modell, das SATPLAN konstruiert: Das Programm bindet die Zuweisung $L^0_{2,1}$ ein, d.h., der Agent kann sich zur Zeit 1 in [2, 1] befinden, indem er sich dort zur Zeit 0 aufhält und schießt. Es stellt sich die Frage: „Haben wir nicht gesagt, dass sich der Agent zur Zeit 0 in [1, 1] befindet?“ Ja, haben wir. Doch wir haben dem Agenten nicht mitgeteilt, dass er sich nicht an zwei Plätzen auf einmal aufhalten kann! Für die logische Konsequenz ist $L^0_{2,1}$ unbekannt und kann demzufolge nicht in einem Beweis verwendet werden; für die Erfüllbarkeit andererseits ist $L^0_{2,1}$ unbekannt und kann demzufolge auf einen beliebigen Wert gesetzt werden, der dabei hilft, das Ziel wahr zu machen. Aus diesem Grund eignet sich SATPLAN als gutes Debugging-Werkzeug für Wissensbasen, da es Plätze offenlegt, an denen Wissen fehlt. Im konkreten Fall können wir die Wissensbasis korrigieren, indem wir zusichern, dass sich der Agent bei jedem Zeitschritt an genau einem Ort befindet. Dazu verwenden wir eine Auflistung von Sätzen ähnlich denen, mit denen wir die Existenz von genau einem Wumpus zusichern. Alternativ können wir $\neg L^0_{x,y}$ für alle Orte außer [1, 1] zusichern; das Nachfolgerzustandsaxiom für den Ort kümmert sich um die darauffolgenden Zeitschritte. Die gleichen Korrekturen funktionieren auch, um zu gewährleisten, dass der Agent nur genau eine Richtung hat.

SATPLAN hat aber noch mehr Überraschungen auf Lager. Erstens findet der Algorithmus Modelle mit unmöglichen Aktionen, beispielsweise Schießen ohne Pfeil. Um die Ursachen zu verstehen, müssen wir uns genauer ansehen, was die Nachfolgerzustands-Axiome (wie zum Beispiel Gleichung (7.3)) über Aktionen aussagen, deren Vorbedingungen nicht erfüllt sind. Die Axiome *sagen zwar korrekt voraus*, dass nichts passiert, wenn eine derartige Aktion ausgeführt wird (siehe Übung 10.14), doch sie *besagen nicht*, dass sich die Aktion nicht ausführen lässt! Um zu vermeiden, dass Pläne mit unzulässigen Aktionen generiert werden, müssen wir **Vorbedingungs-Axiome** hinzufügen, die vorschreiben, dass für das Auftreten einer Aktion die entsprechenden Vorbedingungen erfüllt sein müssen.[13] Zum Beispiel müssen wir für jede Zeit t sagen, dass

$$Shoot^t \Rightarrow HaveArrow^t.$$

Wenn nun ein Plan zu irgendeinem Zeitpunkt die *Shoot*-Aktion auswählt, muss der Agent zu dieser Zeit einen Pfeil besitzen.

Die zweite Überraschung von SATPLAN ist das Erstellen von Plänen mit mehreren gleichzeitigen Aktionen. Zum Beispiel kann der Algorithmus ein Modell hervorbringen, in dem sowohl $Forward^0$ als auch $Shoot^0$ wahr sind, was nicht erlaubt ist. Um dieses Problem zu beseitigen, führen wir **Aktionsausschluss-Axiome** ein: Für jedes Paar von Aktionen A^t_i und A^t_j fügen wir das Axiom

$$\neg A^t_i \vee \neg A^t_j$$

hinzu. Es lässt sich auch feststellen, dass es nicht allzu schwer ist, vorwärts zu gehen und gleichzeitig zu schießen, während es beispielsweise ziemlich umständlich ist,

13 Das Hinzufügen der Vorbedingungs-Axiome bedeutet, dass wir keine Vorbedingungen für Aktionen in den Nachfolgerzustands-Axiomen einbinden müssen.

gleichzeitig zu schießen und das Gold aufzunehmen. Indem man Aktionsausschluss-Axiome nur für diejenigen Aktionspaare aufstellt, die sich wirklich gegenseitig behindern, lassen sich Pläne berücksichtigen, die mehrere simultane Aktionen einschließen – und da SATPLAN den kürzesten zulässigen Plan findet, können wir sicher sein, dass der Algorithmus von dieser Fähigkeit Gebrauch macht.

Alles in allem findet SATPLAN Modelle für einen Satz, der den Anfangszustand, das Ziel, die Nachfolgerzustands-Axiome, die Vorbedingungs-Axiome und die Aktionsausschluss-Axiome enthält. Es lässt sich zeigen, dass diese Sammlung von Axiomen hinreichend ist, und zwar in dem Sinne, dass es keinerlei „Pseudolösungen" mehr gibt. Jedes Modell, das den aussagenlogischen Satz erfüllt, stellt einen gültigen Plan für das ursprüngliche Problem dar. Die moderne SAT-Lösungstechnologie macht den Ansatz recht praktikabel. Zum Beispiel hat ein Solver im Stil von DPLL keine Schwierigkeiten, die aus elf Schritten bestehende Lösung für die in Abbildung 7.2 gezeigte Instanz der Wumpus-Welt zu generieren.

Dieser Abschnitt hat einen deklarativen Ansatz für die Agentenkonstruktion beschrieben: Der Agent kombiniert Zusicherungssätze in der Wissensbasis mit logischer Inferenz. Dieser Ansatz weist allerdings einige Schwächen auf, die sich in Formulierungen wie zum Beispiel „für jede Zeit t" und „für jedes Feld $[x, y]$" verbergen. Für jeden praktischen Agenten sind diese Formulierungen durch Code zu implementieren, der automatisch Instanzen des allgemeinen Satzschemas generiert und in die Wissensbasis einfügt. Für eine Wumpus-Welt vernünftiger Größe – die etwa mit einem kleineren Computerspiel vergleichbar ist – brauchen wir vielleicht ein Brett von 3×3 Feldern und 1000 Zeitschritte, was zu Wissensbasen mit einigen zehn oder hundert Millionen Sätzen führt. Das wird nicht nur recht unpraktisch, sondern veranschaulicht ein tieferes Problem: Wir wissen etwas über die Wumpus-Welt – nämlich, dass die „Physik" in der gleichen Weise über alle Felder und Zeitschritte funktioniert –, was wir nicht direkt in der Sprache der Aussagenlogik ausdrücken können. Um dieses Problem zu lösen, brauchen wir eine ausdrucksstärkere Sprache, und zwar eine, in der sich Formulierungen wie „für jede Zeit t" und „für jedes Feld $[x, y]$" in einer natürlichen Form ausdrücken lassen. Die in *Kapitel 8* beschriebene Logik erster Stufe ist eine derartige Sprache; in Logik erster Stufe lässt sich eine Wumpus-Welt beliebiger Größe und Dauer in rund zehn Sätzen beschreiben statt in zehn Millionen oder zehn Billionen.

Bibliografie und historische Hinweise

Die Arbeit von John McCarthy, „Programs with Common Sense" (McCarthy, 1958, 1968) verbreitete das Konzept der Agenten, die logisches Schließen verwenden, um eine Verbindung zwischen Wahrnehmungen und Aktionen herzustellen. Außerdem brach er eine Lanze für den Deklarativismus, indem er zeigte, dass es ein sehr eleganter Softwareentwurf ist, einem Agenten mitzuteilen, was er tun soll. Der Artikel von Allen Newell (1982), „The Knowledge Level", zeigt, dass rationale Agenten auf einer abstrakten Ebene beschrieben und analysiert werden können, die durch das Wissen definiert ist, das sie selbst besitzen, und nicht durch die Programme, die sie ausführen. Die deklarativen und prozeduralen Ansätze der KI werden durch Boden (1977) verglichen. Die Diskussion wurde unter anderem von Brooks (1991) und Nilsson (1991) wieder aufgenommen und dauert auch heute noch an (Shaparau et al., 2008). Inzwischen hat sich der deklarative Ansatz auch in anderen Bereichen der Informatik wie zum Beispiel der Netzwerktechnik etabliert (Loo et al., 2006).

Die eigentliche Logik hatte ihre Ursprünge in der antiken griechischen Philosophie und Mathematik. In den Arbeiten von Plato findet man verschiedene logische Konzepte – Konzepte, die die syntaktische Struktur von Sätzen mit ihrem Wahrheitsgehalt, ihrer Bedeutung oder mit der Gültigkeit ihrer Argumente in Verbindung brachten. Die erste bekannte systematische Studie der Logik wurde von Aristoteles durchgeführt, dessen Arbeiten von seinen Studenten im Jahr 322 v. Chr. zusammengetragen wurden – zu einem Werk, das unter dem Namen *Organon* bekannt ist. Die **Syllogismen** von Aristoteles waren das, was wir heute als Inferenzregeln bezeichnen würden. Obwohl die Syllogismen Elemente sowohl der Aussagenlogik als auch der Logik erster Stufe enthielten, fehlten dem System als Ganzes die kompositionalen Eigenschaften, die erforderlich sind, um Sätze beliebiger Komplexität zu verarbeiten.

Die eng damit verwandten Schulen von Megara und Stoa (die ihre Ursprünge im fünften Jahrhundert v. Chr. haben und mehrere Jahrhunderte n. Chr. weiterwirkten) begannen mit der systematischen Untersuchung der grundlegenden logischen Verknüpfungen. Die Verwendung von Wahrheitstabellen für die Definition logischer Verknüpfungen geht auf Philo von Megara zurück. Die Stoiker erachteten fünf grundlegende Inferenzregeln ohne Beweis als gültig, unter anderem die Regel, die wir heute als Modus ponens bezeichnen. Sie leiteten mehrere andere Regeln von diesen fünf Regeln ab, wofür sie unter anderem Konzepte wie das Ableitungstheorem (*Abschnitt 7.5*) verwendeten und dabei einen viel klareren Beweisbegriff als Aristoteles hatten. Einen guten Überblick über die Geschichte der Logik von Megara und Stoa finden Sie bei Benson Mates (1953).

Das Konzept, logische Inferenz auf einen rein mechanischen Prozess zu reduzieren, der auf eine formale Sprache angewendet wird, geht auf Wilhelm Leibniz (1646–1716) zurück, auch wenn er nur bescheidene Erfolge verbuchen kann, was die Umsetzung seiner Ideen angeht. George Boole (1847) stellte das erste umfassende und funktionierende System formaler Logik in seinem Buch *The Mathematical Analysis of Logic* vor. Die Logik von Boole hielt sich in ihrer Modellierung eng an die gewöhnliche Algebra mit realen Zahlen und verwendete die Substitution logisch äquivalenter Ausdrücke als primäre Inferenzmethode. Obwohl das System von Boole immer noch nicht so gut wie die vollständige Aussagenlogik war, kam er ihr so nahe, dass andere Mathematiker die Lücken schnell füllen konnten. Schröder (1877) beschrieb die konjunktive Normalform, während die Horn-Form sehr viel später von Alfred Horn (1951) eingeführt wurde. Die erste umfassende Darlegung moderner Aussagenlogik (und Logik erster Stufe) findet man in der *Begriffsschrift* von Gottlob Frege (1879).

Das erste mechanische Gerät, das logische Inferenzen durchführte, wurde vom dritten Earl von Stanhope (1753–1816) konstruiert. Der Stanhope Demonstrator konnte Syllogismen und bestimmte Inferenzen in der Wahrscheinlichkeitstheorie verarbeiten. William Stanley Jevons, einer der Leute, die auf der Arbeit von Boole aufbauten und diese verbesserten, konstruierte 1869 sein „logisches Klavier“, das Inferenzen in boolescher Logik ausführte. Eine unterhaltsame und lehrreiche Geschichte dieser und anderer früher mechanischer Geräte für das Schließen finden Sie bei Martin Gardner (1968). Das erste veröffentlichte Computerprogramm für die logische Inferenz war der Logic Theorist von Newell, Shaw und Simon (1957). Dieses Programm sollte den menschlichen Denkprozess nachbilden. Martin Davis (1957) hatte bereits ein Programm entworfen, das 1954 einen Beweis erbrachte, aber die Ergebnisse des Logic Theorist wurden etwas früher veröffentlicht.

Wahrheitstabellen als Methode, die Gültigkeit oder Unerfüllbarkeit von Sätzen in der Sprache der Aussagenlogik zu überprüfen, wurden unabhängig voneinander von Ludwig Wittgenstein (1922) und Emil Post (1921) eingeführt. In den 1930er Jahren wurde ein wesentlicher Fortschritt im Hinblick auf Inferenzmethoden für die Logik erster Stufe erzielt. Insbesondere zeigte Gödel (1930), dass eine vollständige Prozedur für die Inferenz in der Logik erster Stufe durch eine Reduzierung auf die Aussagenlogik unter Verwendung des Theorems von Herbrand (Herbrand, 1930) zu erhalten war. Wir werden diese Geschichte in *Kapitel 9* wieder aufgreifen. Wichtig dabei ist, dass die Entwicklung effizienter aussagenlogischer Algorithmen in den 1960er Jahren größtenteils durch das Interesse der Mathematiker an effizienten Theorembeweisern für die Logik erster Stufe entstand. Der Davis-Putnam-Algorithmus (Davis und Putnam, 1960) war der erste effektive Algorithmus für die aussagenlogische Resolution, war aber in den meisten Fällen weniger effizient als der DPLL-Backtracking-Algorithmus, der zwei Jahre später vorgestellt wurde (1962). Die vollständige Resolutionsregel und ein Beweis für ihre Vollständigkeit erschienen in einer wegweisenden Arbeit von J.A. Robinson (1965), der auch zeigte, wie Schließen in der Logik erster Stufe ohne Ausweichen auf aussagenlogische Techniken erfolgen konnte.

Stephen Cook (1971) zeigte, dass ein Test der Erfüllbarkeit eines Satzes in der Aussagenlogik (das SAT-Problem) NP-vollständig ist. Weil ein Test auf logische Konsequenz äquivalent zum Test der Unerfüllbarkeit ist, ist dieser co-NP-vollständig. Viele Untermengen der Aussagenlogik sind bekannt, für die das Erfüllbarkeitsproblem polynomial lösbar ist; eine solche Untermenge bilden die Horn-Klauseln. Der Vorwärtsverkettungsalgorithmus in linearer Zeit für Horn-Klauseln geht auf Dowling und Gallier (1984) zurück, die ihren Algorithmus als Datenflussprozess beschreiben, vergleichbar mit der Signalweiterleitung in einer Schaltung.

Frühe theoretische Untersuchungen haben gezeigt, dass DPLL eine polynomiale Komplexität für den durchschnittlichen Fall bestimmter natürlicher Verteilungen von Problemen besitzt. Diese sehr interessante Tatsache wurde weniger attraktiv, als Franco und Paul (1983) zeigten, dass dieselben Probleme in konstanter Zeit gelöst werden konnten, indem man einfach beliebige Zuweisungen erriet. Die in diesem Kapitel beschriebene Methode der zufälligen Erzeugung bringt sehr viel schwierigere Probleme mit sich. Motiviert durch den empirischen Erfolg der lokalen Suche für diese Probleme zeigten Koutsoupias und Papadimitriou (1992), dass ein einfacher Bergsteigeralgorithmus *fast alle* Instanzen von Erfüllbarkeitsproblemen sehr schnell lösen kann, und legten dar, dass schwierige Probleme sehr selten sind. Darüber hinaus zeigte Schöning (1999) eine zufallsbasierte Variante des Hill-Climbing-Algorithmus, dessen Laufzeit für den *ungünstigsten* Fall für 3-SAT-Probleme (d.h. Erfüllbarkeit von 3-KNF-Sätzen) bei $O(1{,}333^n)$ liegt – immer noch exponentiell, aber wesentlich schneller als bei vorherigen Schranken für den ungünstigsten Fall. Der aktuelle Rekord beträgt $O(1{,}324^n)$ (Iwama und Tamaki, 2004). Achlioptas et al. (2004) und Alekhnovich et al. (2005) legen Familien von 3-SAT-Instanzen vor, für die sämtliche bekannten DPLL-artigen Algorithmen eine exponentielle Laufzeit benötigen.

Auf der praktischen Seite sind Effizienzgewinne bei Lösungsprogrammen (Solvern) der Aussagenlogik zu verzeichnen. Für 10 Minuten Rechenzeit konnte der ursprüngliche DPLL-Algorithmus im Jahr 1962 nur Probleme mit nicht mehr als 10 oder 15 Variablen lösen. Demgegenüber konnte der SATZ-Solver (Li und Anbulagan, 1997) mit 1.000 Variablen umgehen, dank der optimierten Datenstrukturen für indizierte Variablen. Zwei entscheidende Beiträge waren die Indizierungstechnik **überwachtes Literal** von Zhang und Stickel (1996), die Einheitspropagierung sehr effizient machte, und die Einführung der Technik zum Lernen von Klauseln (d.h. Beschränkungen) aus der CSP-Gemeinde durch Bayardo und Schrag (1997). Aufbauend auf diesen Ideen und angespornt durch die Aussicht auf die Lösung von Schaltkreisverifizierungsproblemen im industriellen Maßstab, haben Moskewicz et al. (2001) den CHAFF-Solver entwickelt, der Probleme mit Millionen von Variablen verarbeiten kann. Seit 2002 werden regelmäßig SAT-Wettbewerbe abgehalten. Die meisten Siegerbeiträge sind entweder Abkömmlinge von CHAFF oder verwenden den gleichen allgemeinen Ansatz. In die zweite Kategorie fällt RSAT (Pipatsrisawat und Darwiche, 2007), der Gewinner von 2007. Außerdem ist MINISAT (Een und Sörensson, 2003) erwähnenswert, eine OpenSource-Implementierung, die unter *http://minisat.se* zur Verfügung steht und darauf ausgelegt ist, leicht modifiziert und verbessert zu werden. Die aktuelle Landschaft von Solvern wird von Gomes et al. überblicksmäßig dargestellt.

Lokale Suchalgorithmen für die Erfüllbarkeit wurden in den 1980er Jahren von verschiedenen Autoren untersucht. Alle diese Algorithmen basierten auf der Idee, die Anzahl der nicht erfüllten Klauseln zu minimieren (Hansen und Jaumard, 1990). Ein besonders effektiver Algorithmus wurde von Gu (1989) und unabhängig davon von Selman et al. (1992) entwickelt, der ihn als GSAT bezeichnete und zeigte, dass er in der Lage war, einen großen Bereich sehr schwieriger Probleme sehr schnell zu lösen. Der in diesem Kapitel beschriebene WALKSAT-Algorithmus geht auf Selman et al. (1996) zurück.

Der „Phasenübergang“ der Erfüllbarkeit zufälliger k-SAT-Probleme wurde zuerst von Simon und Dubois (1989) beobachtet und gab den Anstoß zu einer ganzen Reihe theoretischer und empirischer Forschungen. Das hängt zum Teil auch mit der naheliegenden Verbindung zum Phänomen des Phasenüberganges in der statistischen Physik zusammen. Cheeseman et al. (1991) beobachteten Phasenübergänge in mehreren CPS und vermuteten, dass alle NP-harten Probleme einen Phasenübergang aufweisen. Crawford und Auton (1993) lokalisierten den 3-SAT-Übergang bei einem Klausel-/Variablen-Verhältnis von etwa 4,26 und wiesen darauf hin, dass dies mit einem scharfen Anstieg der Laufzeit ihrer SAT-Solver zusammenfällt. Cook und Mitchell (1997) bieten einen ausgezeichneten Überblick über die frühe Literatur zu diesem Problem.

Der gegenwärtige Status des theoretischen Verständnisses wird von Achlioptas (2009) zusammengefasst. Das **Schwellenwert-Phänomen der Erfüllbarkeit** besagt, dass es für jedes k eine scharfe Erfüllbarkeitsschwelle r_k gibt, sodass für eine Variablenanzahl von $n \to \infty$ Instanzen unterhalb des Schwellenwertes mit der Wahrscheinlichkeit 1 erfüllbar sind, während diejenigen oberhalb des Schwellenwertes mit der Wahrscheinlichkeit 1 nicht erfüllbar sind. Die Vermutung wurde von Friedgut (1999) nicht völlig bewiesen: Es existiert zwar eine scharfe Schwelle, doch kann ihre Position von n sogar bei $n \to \infty$ abhängen. Trotz signifikanter Fortschritte in der asymptotischen Analyse

der Schwellwertposition für große k (Achlioptas und Peres, 2004; Achlioptas et al., 2007) lässt sich für k = 3 lediglich beweisen, dass die Schwelle im Bereich [3,52; 4,51] liegt. Gemäß der aktuellen Theorie hängt eine Spitze in der Laufzeit eines SAT-Solvers nicht unbedingt mit der Erfüllbarkeitsschwelle zusammen, sondern stattdessen mit einem Phasenübergang in der Lösungsverteilung und Struktur von SAT-Instanzen. Empirische Ergebnisse von Coarfa et al. (2003) unterstützen diese Sichtweise. Tatsächlich nutzen Algorithmen wie zum Beispiel **Survey-Propagation** (Parisi und Zecchina, 2002; Maneva et al., 2007) die besonderen Eigenschaften von zufälligen SAT-Instanzen nahe der Erfüllbarkeitsschwelle und übertreffen damit erheblich die Leistung von allgemeinen SAT-Solvern auf derartigen Instanzen.

Die besten Informationsquellen zur Erfüllbarkeit sind sowohl in theoretischer als auch praktischer Hinsicht das *Handbook of Satisfiability* (Biere et al., 2009) und die regelmäßigen *International Conferences on Theory and Applications of Satisfiability Testing* (als SAT bekannt).

Die Idee, Agenten mit Aussagenlogik zu erstellen, lässt sich auf die wegweisende Arbeit von McCulloch und Pitts (1943) zurückführen, die Pionierarbeit auf dem Gebiet der neuronalen Netze geleistet haben. Im Gegensatz zur gängigen Ansicht beschäftigte sich die Arbeit mit der Implementierung eines booleschen schaltungsbasierten Agentenentwurfes im Gehirn. Schaltungsbasierte Agenten haben in der KI jedoch wenig Aufmerksamkeit erfahren, abgesehen davon, dass sie Algorithmen in Allzweckcomputern ausführen. Die bemerkenswerteste Ausnahme ist die Arbeit von Stan Rosenschein (Rosenschein, 1985; Kaelbling und Rosenschein, 1990), der Möglichkeiten entwickelte, schaltungsbasierte Agenten aus deklarativen Beschreibungen der Aufgabenumgebung zu kompilieren. (Der Ansatz von Rosenschein wird eingehend in der zweiten Ausgabe dieses Buches beschrieben.) Die Arbeit von Rod Brooks (1986, 1989) demonstriert die Effektivität des schaltungsbasierten Entwurfes für Robotersteuerungen – ein Thema, das wir in *Kapitel 25* wieder aufgreifen werden. Brooks (1991) behauptet, dass man *ausschließlich* schaltungsbasierte Entwürfe für die KI brauche – dass Repräsentation und Schließen umständlich, teuer und unnötig seien. Unserer Meinung nach ist kein Ansatz für sich allein ausreichend. Williams et al. (2003) zeigen, wie ein hybrider Agentenentwurf, der sich nicht allzu sehr von unserem Wumpus-Agenten unterscheidet, eingesetzt wurde, um NASA-Raumflugkörper zu steuern, Aktionsfolgen zu planen sowie Fehler zu diagnostizieren und zu korrigieren.

Das allgemeine Problem, eine partiell beobachtbare Umgebung zu verfolgen, wurde für zustandsbasierte Darstellungen in *Kapitel 4* eingeführt. Eine Instanziierung für aussagenlogische Darstellungen wurde von Amir und Russell (2003) untersucht; sie identifizierten mehrere Klassen von Umgebungen, die effiziente Zustandsschätzungsalgorithmen ermöglichen, und zeigten, dass für mehrere andere Klassen das Problem nicht lösbar ist. Das Problem der **zeitlichen Projektion**, bei dem ermittelt wird, welche Aussagen nach einer ausgeführten Aktionsfolge noch wahr sind, lässt sich als Sonderfall der Zustandsschätzung mit leeren Wahrnehmungen auffassen. Viele Autoren haben dieses Problem wegen seiner Bedeutung in der Planung untersucht; manche wichtigen Ergebnisse zur Härte wurden von Liberatore (1997) aufgestellt. Das Konzept der Darstellung eines Belief State mit Aussagen kann auf Wittgenstein (1922) zurückgeführt werden.

Die logische Zustandsschätzung setzt natürlich eine logische Darstellung der Wirkungen von Aktionen voraus – ein Kernproblem in der KI seit Ende der 1950er Jahre. Der vorherrschende Ansatz ist der Formalismus des **Situationskalküls** (McCarthy, 1963) gewesen, der in Logik erster Stufe ausgedrückt wird. Mit dem Situationskalkül und verschiedenen Erweiterungen und Alternativen beschäftigen sich die *Kapitel 10* und *12*. Der in diesem Kapitel gewählte Ansatz – die Verwendung zeitlicher Indizes an aussagenlogischen Variablen – ist restriktiver, hat aber den Vorzug der Einfachheit. Das allgemeine Konzept, das im Algorithmus SATPLAN verankert ist, wurde von Kautz und Selman (1992) vorgeschlagen. Spätere Generationen von SATPLAN konnten von den Fortschritten bei SAT-Solvern (wie weiter vorn beschrieben) profitieren und gehören weiterhin zu den effektivsten Verfahren, schwierige Probleme zu lösen (Kautz, 2006).

Das **Rahmenproblem** wurde zuerst von McCarthy und Hayes (1969) erkannt. Viele Forscher haben das Problem in Logik erster Stufe als unlösbar betrachtet und es hat eine umfangreiche Untersuchung in nichtmonotonen Logiken angespornt. Philosophen von Dreyfus (1972) bis Crockett (1994) führen das Rahmenproblem als ein Symptom für das zwangsläufige Scheitern des gesamten KI-Unternehmens an. Die Lösung des Rahmenproblems mit Nachfolgerzustands-Axiomen ist Ray Reiter (1991) zu verdanken. Thielscher (1999) identifiziert das inferentielle Rahmenproblem als separates Konzept und gibt eine Lösung an. Rückschauend ist festzustellen, dass die Agenten von Rosenschein (1985) Schaltungen nutzen, die Nachfolgerzustands-Axiome implementieren, doch Rosenschein hat nicht erkannt, dass dabei das Rahmenproblem größtenteils gelöst war. Foo (2001) erläutert, warum sich die Modelle der Steuerungstheorie für diskrete Ereignisse, wie sie der Techniker normalerweise verwendet, mit dem Rahmenproblem nicht explizit befassen: weil sie mit Vorhersage und Steuerung zu tun haben und nicht mit Erklärungen und Nachdenken über kontrafaktische Situationen.

Moderne aussagenlogische Solver haben auf breiter Front Einzug in industrielle Anwendungen gehalten. Die Anwendung der aussagenlogischen Inferenz bei der Synthese von Computerhardware gehört heute zur Standardtechnik, die viele Installationen im großen Maßstab aufweist (Nowick et al., 1993). Mit dem Erfüllbarkeitsprüfer SATMC wurde eine bisher unbekannte Schwachstelle beim Anmeldeprotokoll eines Benutzers in einem Webbrowser aufgedeckt (Armando et al., 2008).

Die Wumpus-Welt wurde von Gregory Yob (1975) erfunden. Ironischerweise entwickelte Yob sie, weil es ihm langweilig war, Spiele in einem Raster zu spielen: Die Topologie seiner ursprünglichen Wumpus-Welt war ein Dodekaeder und wir haben sie zurück in das gute alte Raster geführt. Michael Genesereth war der Erste, der vorschlug, die Wumpus-Welt als Testumgebung für Agenten zu verwenden.

Zusammenfassung

Wir haben wissensbasierte Agenten vorgestellt und gezeigt, wie man eine Logik definiert, die diese Agenten nutzen können, um Schlüsse über die Welt zu ziehen. Die wichtigsten Aspekte dabei waren:

- Intelligente Agenten brauchen Wissen über die Welt, um sinnvolle Entscheidungen treffen zu können.
- Wissen ist in den Agenten in Form von **Sätzen** enthalten, die in einer Wissensrepräsentationssprache formuliert und in einer **Wissensbasis** gespeichert sind.
- Ein wissensbasierter Agent setzt sich aus einer Wissensbasis und einem Inferenzmechanismus zusammen. Er arbeitet, indem er Sätze über die Welt in seiner Wissensbasis speichert und den Inferenzmechanismus einsetzt, um neue Sätze abzuleiten und anhand dieser Sätze zu entscheiden, welche Aktion ausgeführt werden soll.
- Eine Repräsentationssprache wird durch ihre **Syntax** definiert, die die Struktur der Sätze spezifiziert, sowie durch ihre **Semantik**, die die **Wahrheit** jedes Satzes in jeder möglichen **Welt** oder in jedem möglichen **Modell** definiert.
- Die Beziehung der **logischen Konsequenz** zwischen Sätzen ist wesentlich für unser Verständnis des Schließens. Ein Satz β ist die logische Konsequenz eines anderen Satzes α, wenn β in allen Welten wahr ist, wo auch α wahr ist. Äquivalente Definitionen beinhalten die **Gültigkeit** des Satzes $\alpha \Rightarrow \beta$ und die **Unerfüllbarkeit** des Satzes $\alpha \wedge \neg\beta$.
- Inferenz ist der Prozess, neue Sätze aus alten Sätzen abzuleiten. **Korrekte** Inferenzalgorithmen leiten nur Sätze ab, die logische Konsequenzen sind; **vollständige** Algorithmen leiten alle logischen Konsequenzen ab.
- Die **Aussagenlogik** ist eine sehr einfache Sprache, die aus **Aussagensymbolen** und **logischen Verknüpfungen** besteht. Sie kann Aussagensymbole verarbeiten, die als wahr oder falsch bekannt oder völlig unbekannt sind.
- Die Menge der möglichen Modelle für ein feststehendes aussagenlogisches Vokabular ist endlich, sodass die logische Konsequenz mithilfe der Auflistung von Modellen überprüft werden kann. Effiziente **Model-Checking-**Inferenzalgorithmen für die Aussagenlogik sind unter anderem Backtracking und lokale Suchmethoden, die große Probleme oftmals sehr schnell lösen können.
- **Inferenzregeln** sind Muster korrekter Inferenzen, die verwendet werden können, um Beweise zu finden. Die **Resolutionsregel** ergibt einen vollständigen Inferenzalgorithmus für Wissensbasen, die in **konjunktiver Normalform** (**KNF**) ausgedrückt sind. **Vorwärtsverkettung** und **Rückwärtsverkettung** sind sehr natürliche Schlussalgorithmen für Wissensbasen in der **Horn-Form**.
- Methoden der **lokalen Suche** wie zum Beispiel WalkSAT können verwendet werden, um Lösungen zu finden. Derartige Algorithmen sind korrekt, aber nicht vollständig.
- Bei der logischen **Zustandsschätzung** wird ein logischer Satz verwaltet, der die Menge der möglichen Zustände beschreibt, die konsistent mit dem Beobachtungsverlauf sind. Jeder Aktualisierungsschritt erfordert Inferenz mithilfe des Übergangsmodells der Umgebung. Es wird erstellt aus Nachfolgerzustands-Axiomen, die spezifizieren, wie sich jeder **Fluent** ändert.
- Entscheidungen innerhalb eines logischen Agenten können durch SAT Solving erfolgen: mögliche Modelle finden, die zukünftige Aktionsfolgen spezifizieren, die das Ziel erreichen. Dieser Ansatz funktioniert nur für vollständig beobachtbare oder sensorlose Umgebungen.
- Aussagenlogik lässt sich nicht auf Umgebungen mit unbeschränkter Größe skalieren, weil ihr die Ausdruckskraft fehlt, sich prägnant mit Zeit, Raum und universellen Mustern von Beziehungen zwischen Objekten zu befassen.

Übungen zu Kapitel 7

Lösungshinweise

1 Angenommen, der Agent hat es bis zu dem in Abbildung 7.4(a) gezeigten Punkt geschafft und dabei in [1,1] nichts, in [2,1] einen Luftzug und in [1,2] einen Gestank wahrgenommen. Jetzt will er den Inhalt von [1,3], [2,2] und [3,1] herausfinden. Jedes dieser Felder kann eine Falltür und höchstens einen Wumpus enthalten. Konstruieren Sie gemäß dem Beispiel aus Abbildung 7.5 die Menge möglicher Welten. (Dabei sollten Sie 32 von ihnen finden.) Markieren Sie die Welten, in denen die KB wahr ist, und diejenigen, in denen jeder der folgenden Sätze wahr ist:

α_2 = „Es gibt keine Falltür in [2,2]."

α_3 = „Der Wumpus befindet sich in [1,3]."

Zeigen Sie also, dass $KB \models \alpha_2$ und $KB \models \alpha_3$.

2 (Übernommen aus Barwise und Etchemendy, 1993) Können Sie unter den folgenden Voraussetzungen beweisen, dass das Einhorn mythisch ist? Ist es auch magisch? Gehörnt?

Wenn das Einhorn mythisch ist, ist es unsterblich, aber wenn es nicht mythisch ist, ist es ein sterbliches Säugetier. Ist das Einhorn entweder unsterblich oder ein Säugetier, ist es gehörnt. Das Einhorn ist magisch, wenn es gehörnt ist.

3 Betrachten Sie das Problem, zu entscheiden, ob ein aussagenlogischer Satz in einem bestimmten Modell wahr ist.

a. Schreiben Sie einen rekursiven Algorithmus PL-TRUE?(s, m), der genau dann *true* zurückgibt, wenn der Satz s im Modell m wahr ist (wobei m einen Wahrheitswert für jedes Symbol in s zuweist). Der Algorithmus soll innerhalb einer Zeit ausgeführt werden, die linear zur Größe des Satzes ist. (Alternativ verwenden Sie eine Version dieser Funktion aus dem online bereitgestellten Code.)

b. Nennen Sie drei Beispiele für Sätze, die in einem *partiellen* Modell, das für einige der Symbole keinen Wahrheitswert angibt, als wahr oder falsch ermittelt werden können.

c. Zeigen Sie, dass der Wahrheitswert (falls vorhanden) eines Satzes in einem partiellen Modell im Allgemeinen nicht effizient ermittelt werden kann.

d. Ändern Sie Ihren Algorithmus PL-TRUE? so ab, dass er manchmal die Wahrheit aus partiellen Modellen beurteilen kann, während er seine rekursive Struktur und lineare Laufzeit beibehält. Nennen Sie drei Beispiele für Sätze, wo Ihr Algorithmus die Wahrheit in einem partiellen Modell *nicht* erkennt.

e. Untersuchen Sie, ob der modifizierte Algorithmus TT-ENTAILS? effizienter macht.

4 Welche der folgenden Aussagen sind korrekt?

a. $False \models \alpha$.

b. $True \models False$.

c. $(A \wedge B) \models (A \Leftrightarrow B)$.

d. $A \Leftrightarrow B \models A \vee B$.

e. $A \Leftrightarrow B \models \neg A \vee B$.

f. $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B \vee C) \wedge (B \wedge C \wedge D \Rightarrow E)$.

g. $(A \vee B) \wedge (\neg C \vee \neg D \vee E) \models (A \vee B) \wedge (\neg D \vee E)$.

h. $(A \vee B) \wedge \neg(A \Rightarrow B)$ ist erfüllbar.

i. $(A \wedge B) \Rightarrow C \models (A \Rightarrow C) \vee (B \Rightarrow C)$.

j. $(C \vee (\neg A \wedge \neg B)) \equiv ((A \Rightarrow C) \wedge (B \Rightarrow C))$.

k. $(A \Leftrightarrow B) \wedge (\neg A \vee B)$ ist erfüllbar.

l. $(A \Leftrightarrow B) \Leftrightarrow C$ hat die gleiche Anzahl von Modellen wie $(A \Leftrightarrow B)$ für jede feste Menge von Aussagensymbolen, die A, B, C einschließen.

5 Beweisen Sie jede der folgenden Behauptungen:

a. α ist genau dann gültig, wenn $True \models \alpha$.

b. Für jedes α gilt $False \models \alpha$.

c. $\alpha \models \beta$ genau dann, wenn der Satz $(\alpha \Rightarrow \beta)$ gültig ist.

d. $\alpha \equiv \beta$ genau dann, wenn der Satz $(\alpha \Leftrightarrow \beta)$ gültig ist.

e. $\alpha \models \beta$ genau dann, wenn der Satz $(\alpha \wedge \neg\beta)$ unerfüllbar ist.

6 Beweisen Sie die folgenden Behauptungen oder geben Sie ein Gegenbeispiel an:

a. Wenn $\alpha \models \gamma$ und/oder $\beta \models \gamma$, dann gilt $(\alpha \wedge \beta) \models \gamma$.

b. Wenn $(\alpha \wedge \beta) \models \gamma$, dann gilt $\alpha \models \gamma$ und/oder $\beta \models \gamma$.

c. Wenn $\alpha \models (\beta \vee \gamma)$, dann gilt $\alpha \models \beta$ und/oder $\alpha \models \gamma$.

7 Betrachten Sie ein Vokabular mit nur vier Aussagensymbolen, A, B, C und D. Wie viele Modelle gibt es für die folgenden Sätze?

a. $B \vee C$

b. $\neg A \vee \neg B \vee \neg C \vee \neg D$

c. $(A \Rightarrow B) \wedge A \wedge \neg B \wedge C \wedge D$

8 Wir haben vier verschiedene binäre logische Verknüpfungen definiert.

a. Gibt es noch weitere, die praktisch sein könnten?

b. Wie viele binäre Verknüpfungen kann es geben?

c. Warum sind einige davon nicht sehr praktisch?

9 Überprüfen Sie unter Verwendung einer Methode Ihrer Wahl die Äquivalenzen aus Abbildung 7.11.

10 Entscheiden Sie, ob die folgenden Sätze gültig, unerfüllbar oder keines von beiden sind. Überprüfen Sie Ihre Entscheidungen anhand von Wahrheitstabellen oder der Äquivalenzregeln aus Abbildung 7.11.

a. $Smoke \Rightarrow Smoke$

b. $Smoke \Rightarrow Fire$

c. $(Smoke \Rightarrow Fire) \Rightarrow (\neg Smoke \Rightarrow \neg Fire)$

d. $Smoke \vee Fire \vee \neg Fire$

e. $((Smoke \wedge Heat) \Rightarrow Fire) \Leftrightarrow ((Smoke \Rightarrow Fire) \vee (Heat \Rightarrow Fire)$

f. $Big \vee Dumb \vee (Big \Rightarrow Dumb)$

g. $(Big \wedge Dumb) \vee \neg Dumb$

11 Jeder Satz der Aussagenlogik ist logisch äquivalent mit der Behauptung, dass es keine mögliche Welt gibt, in der er falsch wäre. Beweisen Sie anhand dieser Beobachtung, dass jeder Satz in KNF formuliert werden kann.

12 Beweisen Sie mittels Resolution den Satz $\neg A \vee \neg B$ aus den Klauseln in Übung 7.19.

13 Diese Übung betrachtet die Beziehung zwischen Klauseln und Implikationssätzen.

a. Zeigen Sie, dass die Klausel $(\neg P_1 \vee \ldots \vee \neg P_m \vee Q)$ logisch äquivalent mit dem Implikationssatz $(P_1 \wedge \ldots \wedge P_m) \Rightarrow Q$ ist.

b. Zeigen Sie, dass jede Klausel (unabhängig von der Anzahl positiver Literale) in der Form $(P_1 \wedge \ldots \wedge P_m) \Rightarrow (Q_1 \vee \ldots \vee Q_n)$ geschrieben werden kann ist, wobei die P und Q Aussagensymbole sind. Eine Wissensbasis, die aus solchen Symbolen besteht, befindet sich in **implikativer Normalform** oder **Kowalski-Form** (Kowalski, 1979).

c. Schreiben Sie die vollständige Resolutionsregel für Sätze in implikativer Normalform.

14 Gemäß mancher Politexperten ist eine Person, die radikal (R) ist, wählbar (W), wenn sie konservativ (K) ist. Andernfalls ist sie nicht wählbar.

a. Welche der folgenden Sätze sind korrekte Darstellungen dieser Behauptung?

(i) $(R \wedge E) \Leftrightarrow C$

(ii) $R \Rightarrow (E \Leftrightarrow C)$

(iii) $R \Leftrightarrow ((C \Rightarrow E) \vee \neg E)$

b. Welche der Sätze in (a) lassen sich in Horn-Form ausdrücken?

15 In dieser Frage geht es um die Darstellung von Erfüllbarkeitsproblemen (SAT-Problemen) als CSP.

a. Zeichnen Sie den Constraint-Graphen, der dem SAT-Problem

$$(\neg X_1 \vee X_2) \wedge (\neg X_2 \vee X_3) \wedge \ldots \wedge (\neg X_{n-1} \vee X_n)$$

entspricht, für den konkreten Fall $n = 4$.

b. Wie viele Lösungen gibt es für dieses allgemeine SAT-Problem als Funktion von n?

c. Angenommen, wir wenden BACKTRACKING-SEARCH (*Abschnitt 6.3*) an, um *alle* Lösungen für ein SAT-CSP des in (a) angegebenen Typs zu finden. (Um alle Lösungen für ein CSP zu finden, modifizieren wir einfach den Basisalgorithmus, sodass er nach jeder gefundenen Lösung weitersucht.) Nehmen Sie an, dass die Reihenfolge der Variablen $X_1, \ldots, X_n$ lautet und $false$ vor $true$ eingeordnet ist. Wie viel Zeit benötigt der Algorithmus bis zur Fertigstellung? (Schreiben Sie einen $O()$-Ausdruck als Funktion von n.)

d. Wir wissen, dass sich SAT-Probleme in Horn-Form in linearer Zeit durch Vorwärtsverkettung (Einheitspropagierung) lösen lassen. Außerdem ist bekannt, dass jedes baumstrukturierte binäre CSP mit diskreten, endlichen Domänen in linearer Zeit zur Anzahl der Variablen gelöst werden kann (*Abschnitt 6.5*). Sind diese beiden Fakten miteinander verknüpft? Erörtern Sie Ihre Antwort.

16 Beweisen Sie jede der folgenden Behauptungen:

a. Jedes Paar aussagenlogischer Klauseln besitzt entweder keine Resolventen oder alle ihre Resolventen sind logisch äquivalent.

b. Es gibt keine Klausel, die mit sich selbst aufgelöst die Klausel $(\neg P \vee \neg Q)$ (nach der Faktorisierung) liefert.

c. Wenn eine aussagenlogische Klausel C mit einer Kopie von sich selbst aufgelöst werden kann, muss sie logisch äquivalent zu $True$ sein.

17 Sehen Sie sich den folgenden Satz an:

$$[(Food \Rightarrow Party) \vee (Drinks \Rightarrow Party)] \Rightarrow [(Food \wedge Drinks) \Rightarrow Party]$$

a. Bestimmen Sie mithilfe der Aufzählung, ob dieser Satz gültig, erfüllbar (aber nicht gültig) oder nicht erfüllbar ist.

b. Konvertieren Sie die linken und rechten Seiten der Hauptimplikation nach CNF, zeigen Sie dabei jeden Schritt und erläutern Sie, wie die Ergebnisse Ihre Antwort auf (a) bestätigen.

c. Beweisen Sie Ihre Antwort auf (a) mithilfe der Resolution.

18 Ein Satz ist in **disjunktiver Normalform** (DNF), wenn er eine Disjunktion von Konjunktionen von Literalen ist. Zum Beispiel ist der Satz $(A \wedge B \wedge \neg C) \vee (\neg A \wedge C) \vee (B \wedge \neg C)$ in DNF.

a. Jeder aussagenlogische Satz ist logisch äquivalent mit der Behauptung, dass es tatsächlich eine mögliche Welt gibt, in der er wahr ist. Beweisen Sie anhand dieser Beobachtung, dass jeder Satz in DNF formuliert werden kann.

b. Konstruieren Sie einen Algorithmus, der jeden Satz in Aussagenlogik nach DNF konvertiert. (*Hinweis:* Der Algorithmus ähnelt dem Algorithmus zur Umwandlung in KNF, der in *Abschnitt 7.5.2 angegeben* ist.)

c. Konstruieren Sie einen einfachen Algorithmus, der als Eingabe einen Satz in DNF übernimmt und eine zufriedenstellende Zuweisung (sofern vorhanden) zurückgibt oder meldet, dass keine zufriedenstellende Zuweisung existiert.

d. Wenden Sie die Algorithmen in (b) und (c) auf die folgenden Sätze an:

$A \Rightarrow B$

$B \Rightarrow C$

$C \Rightarrow \neg A$.

e. Da der Algorithmus in (b) dem Algorithmus zur Konvertierung in KNF sehr ähnelt und da der Algorithmus in (c) wesentlich einfacher ist als jeder Algorithmus zum Lösen einer Menge von Sätzen in KNF, stellt sich die Frage: Warum wird diese Technik nicht beim automatischen Schließen eingesetzt?

19 Konvertieren Sie die folgende Menge von Sätzen in das Klauselformat:

S1: $A \Leftrightarrow (C \vee E)$

S2: $E \Rightarrow D$

S3: $B \wedge F \Rightarrow \neg C$

S4: $E \Rightarrow C$

S5: $C \Rightarrow F$

S6: $C \Rightarrow B$

Geben Sie eine Ablaufverfolgung für die Ausführung von DPLL auf der Konjunktion dieser Klauseln an.

20 Ist ein zufällig erzeugter 4-KNF-Satz mit n Symbolen und m Klauseln wahrscheinlicher oder weniger wahrscheinlich lösbar als ein zufällig erzeugter 3-KNF-Satz mit n Symbolen und m Klauseln? Erläutern Sie Ihre Antwort.

21 Minesweeper, das bekannte Computerspiel, ist eng mit der Wumpus-Welt verwandt. Eine Minesweeper-Welt ist ein rechteckiges Raster aus N Quadraten, auf denen M unsichtbare Minen verteilt sind. Die Quadrate können vom Agenten getestet werden; trifft er auf eine Mine, wird er sofort getötet. Minesweeper zeigt das Vorhandensein von Minen an, indem es für jedes bereits offengelegte Quadrat die *Anzahl* der Minen angibt, die direkt oder diagonal benachbart sind. Das Ziel dabei ist, jedes nicht mit einer Mine versehene Quadrat offenzulegen.

a. Sei $X_{i,j}$ wahr, wenn das Quadrat $[i,j]$ eine Mine enthält. Schreiben Sie die Behauptung, dass sich neben [1,1] genau zwei Minen befinden, als Satz auf, der irgendeine logische Kombination von $X_{i,j}$-Aussagensymbolen enthält.

b. Verallgemeinern Sie Ihre Behauptung aus (a), indem Sie erklären, wie ein KNF-Satz konstruiert wird, der behauptet, dass k von n benachbarten Feldern Minen enthalten.

c. Erklären Sie genau, wie ein Agent mithilfe von DPLL beweisen kann, dass ein bestimmtes Quadrat eine Mine enthält (oder nicht), und ignorieren Sie dabei die globale Randbedingung, dass es insgesamt M Minen gibt.

d. Angenommen, die globale Beschränkung wird unter Verwendung Ihrer Methode aus Teil (b) konstruiert. Wie hängt die Anzahl der Klauseln von M und N ab? Schlagen Sie eine Möglichkeit vor, DPLL so abzuändern, dass die globale Beschränkung nicht explizit dargestellt werden muss.

e. Werden Schlüsse, die von der Methode in Teil (c) abgeleitet sind, ungültig, wenn die globale Randbedingung berücksichtigt wird?

f. Nennen Sie Beispiele für Konfigurationen von Werten, die *weitläufige Abhängigkeiten* enthalten, sodass der Inhalt eines gegebenen nicht offengelegten Quadrates Informationen über den Inhalt eines weit entfernten Quadrates liefert. [*Hinweis*: Verwenden Sie ein $N \times 1$-Spielfeld.]

22 Wie lange dauert es, mithilfe von DPLL zu beweisen, dass $KB \models \alpha$ ist, wenn das Literal α *bereits in KB enthalten* ist? Erläutern Sie Ihre Antwort.

23 Verfolgen Sie das Verhalten von DPLL für die Wissensbasis in Abbildung 7.16, wenn versucht wird, Q zu beweisen, und vergleichen Sie dieses Verhalten mit dem des Vorwärtsverkettungsalgorithmus.

24 Erörtern Sie, was mit *optimalem* Verhalten in der Wumpus-Welt gemeint ist. Zeigen Sie, dass der HYBRID-WUMPUS-AGENT nicht optimal ist, und schlagen Sie Verfahren vor, ihn zu verbessern.

25 Nehmen Sie an, ein Agent bewohnt eine Welt mit zwei Zuständen, S und $\neg S$, und kann genau eine von zwei Aktionen, a und b, ausführen. Aktion a bewirkt nichts und Aktion b wechselt von einem Zustand in den anderen. Es sei S^t die Aussage, dass sich der Agent im Zustand S zur Zeit t befindet, und a^t die Aussage, dass der Agent die Aktion a zur Zeit t ausführt (analog für b^t).

a. Schreiben Sie ein Nachfolgerzustands-Axiom für S^{t+1}.

b. Konvertieren Sie den Satz in (a) nach KNF.

c. Führen Sie einen Widerspruchsbeweis (Resolution), dass wenn sich der Agent in $\neg S$ zur Zeit t befindet und a ausführt, er sich immer noch in $\neg S$ zur Zeit $t+1$ befindet.

26 *Abschnitt 7.7.1* gibt einige der Nachfolgerzustands-Axiome an, die für die Wumpus-Welt erforderlich sind. Formulieren Sie Axiome für alle übrigen Fluent-Symbole.

27 Modifizieren Sie den HYBRID-WUMPUS-AGENT, um die Methode der 1-KNF logischen Zustandsschätzung zu verwenden, die in *Abschnitt 7.7.3* beschrieben wurde. Wir haben dort angemerkt, dass ein derartiger Agent keine komplexeren Glaubenszustände als die Disjunktion $P_{3,1} \vee P_{2,2}$ lernen, verwalten und verwenden kann. Schlagen Sie eine Methode vor, um dieses Problem aus dem Weg zu räumen, indem Sie zusätzliche Aussagensymbole definieren, und probieren Sie sie in der Wumpus-Welt aus. Verbessert sich dadurch die Leistung des Agenten?

Logik erster Stufe – First-Order-Logik

8

ÜBERBLICK

In diesem Kapitel erkennen wir, dass die Welt übersät ist mit vielen Objekten, die zum Teil in einer Beziehung zu anderen Objekten stehen. Wir bemühen uns, Schlüsse über sie zu ziehen.

In *Kapitel 7* haben wir gezeigt, dass ein wissensbasierter Agent die Welt, in der er arbeitet, repräsentieren und seine Aktionen erschließen kann. Wir haben die Aussagenlogik als unsere Repräsentationssprache verwendet, weil sie ausreichend ist, um die grundlegenden Konzepte der Logik und wissensbasierter Agenten zu verdeutlichen. Leider ist die Sprache der Aussagenlogik zu schwach, um Wissen aus komplexen Umgebungen auf präzise Weise zu repräsentieren. In diesem Kapitel betrachten wir die **Logik erster Stufe** (**First-Order-Logik**),[1] die ausdrucksstark genug ist, um einen Großteil unseres Allgemeinwissens zu repräsentieren. Außerdem beinhaltet sie viele andere Repräsentationssprachen bzw. bildet die Grundlage dafür und ist jahrzehntelang intensiv erforscht worden. Wir beginnen in *Abschnitt 8.1* mit einer allgemeinen Diskussion von Repräsentationssprachen; *Abschnitt 8.2* beschreibt die Syntax und die Semantik der Logik erster Stufe; die *Abschnitte 8.3* und *8.4* zeigen die Verwendung der Logik erster Stufe für einfache Repräsentationen.

8.1 Wiederholung der Repräsentation

In diesem Abschnitt beschreiben wir die Natur der Repräsentationssprachen. Unsere Diskussion motiviert die Entwicklung der Logik erster Stufe, einer sehr viel ausdrucksstärkeren Sprache als die in *Kapitel 7* vorgestellte Aussagenlogik. Wir betrachten die Aussagenlogik und andere Arten von Sprachen, um zu verstehen, was funktioniert und was scheitert. Unsere Diskussion ist relativ oberflächlich und fasst Denkprozesse, Versuche – und ihr Scheitern – aus mehreren Jahrhunderten in nur ein paar wenigen Absätzen zusammen.

Programmiersprachen (wie z.B. C++, Java oder Lisp) bilden die bei weitem umfangreichste Klasse formaler Sprachen, die ganz allgemein eingesetzt werden. Die Programme selbst stellen im direkten Sinn nur einen Rechenprozess dar. Datenstrukturen in Programmen können Fakten repräsentieren; z.B. könnte ein Programm ein 4×4-Array verwenden, um den Inhalt der Wumpus-Welt darzustellen. Die Programmieranweisung $World[2,2] \leftarrow Pit$ ist also eine recht natürliche Weise, zu behaupten, dass es auf dem Quadrat [2,2] eine Falltür gibt. (Derartige Repräsentationen können als *ad hoc* betrachtet werden; Datenbanksysteme wurden entwickelt, um eine allgemeinere, bereichsunabhängigere Möglichkeit zu schaffen, Fakten zu speichern und wieder zu laden.) Den Programmiersprachen fehlt jedoch ein allgemeiner Mechanismus, um Fakten aus anderen Fakten abzuleiten; jede Aktualisierung einer Datenstruktur erfolgt durch eine bereichsspezifische Prozedur, deren Details vom Programmierer aus seinem eigenen Wissen über den Bereich abgeleitet wurden. Dieser **prozedurale** Ansatz steht im Gegensatz zu der **deklarativen** Natur der Aussagenlogik, wo Wissen und Inferenz voneinander getrennt sind und die Inferenz völlig bereichsunabhängig ist.

Ein zweiter Nachteil von Datenstrukturen in Programmen (und im Übrigen auch von Datenbanken) ist das Fehlen einer einfachen Möglichkeit, beispielsweise zu sagen: „Es befindet sich eine Falltür in [2,2] oder [3,1]“ oder: „Wenn sich der Wumpus in [1,1]

1 Die Logik erster Stufe (First Order Logic, FOL) wird auch als **Prädikatenkalkül erster Stufe** (First Order Predicate Calculus, FOPC) oder einfach als **Prädikatenlogik** bezeichnet.

befindet, befindet er sich nicht in [2,2].“ Programme können für jede Variable nur einen einzigen Wert speichern. Einige Systeme erlauben es, dass dieser Wert „unbekannt“ ist, aber sie besitzen nicht die Ausdruckskraft, um partielle Informationen zu verarbeiten.

Die Aussagenlogik ist eine deklarative Sprache, weil ihre Semantik auf einer Wahrheitsrelation zwischen Sätzen und möglichen Welten basiert. Außerdem hat sie genügend Ausdruckskraft, um mit partiellen Informationen zurechtzukommen – mithilfe von Disjunktion und Negation. Die Aussagenlogik hat eine dritte Eigenschaft, die in Repräsentationssprachen wünschenswert ist, nämlich die **Kompositionalität**. In einer kompositionalen Sprache ist die Bedeutung eines Satzes eine Funktion der Bedeutung seiner Bestandteile. Beispielsweise ist die Bedeutung von „$S_{1,4} \wedge S_{1,2}$“ mit den Bedeutungen von „$S_{1,4}$“ und „$S_{1,2}$“ verwandt. Es wäre sehr seltsam, wenn „$S_{1,4}$“ bedeuten würde, dass in Quadrat [1,4] ein Gestank wahrzunehmen ist, und „$S_{1,2}$“ bedeutete, dass in [1,2] ein Gestank wahrgenommen wird, während „$S_{1,4} \wedge S_{1,2}$“ hieße, dass Frankreich und Polen im Eishockey-Qualifikationsspiel der letzten Woche 1:1 gespielt haben. Offensichtlich macht die Nichtkompositionalität das Leben für das Schlusssystem sehr viel schwieriger.

Wie wir in *Kapitel 7* gesehen haben, fehlt der Aussagenlogik die Ausdruckskraft, um eine Umgebung mit vielen Objekten *prägnant* zu beschreiben. Beispielsweise waren wir gezwungen, für Luftzüge und Falltüren für jedes Quadrat separate Regeln zu schreiben, wie etwa:

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}).$$

In unserer natürlichen Sprache scheint es dagegen ganz einfach, ein für alle Mal zu sagen: „Quadrate, die an eine Falltür grenzen, sind windig!“ Die Syntax und Semantik unserer natürlichen Sprache machen es möglich, die Umgebung auf prägnante Weise zu beschreiben.

8.1.1 Die Sprache des Denkens

Natürliche Sprachen (wie Deutsch oder Englisch) sind wirklich sehr ausdrucksstark. Wir haben fast dieses gesamte Buch in natürlicher Sprache geschrieben, mit ein paar gelegentlichen Abweichungen in andere Sprachen (unter anderem Logik, Mathematik und die Bildersprache). Es gibt eine lange Tradition in der Linguistik und der Sprachphilosophie, die die natürliche Sprache im Wesentlichen als deklarative Wissensrepräsentationssprache betrachtet. Wenn wir in der Lage wären, die Regeln für natürliche Sprache aufzudecken, könnten wir sie in Repräsentations- und Inferenzsystemen verwenden und von den Milliarden Seiten profitieren, die in natürlicher Sprache geschrieben wurden.

Die moderne Sicht der natürlichen Sprache ist, dass sie einem ganz anderen Zweck dient, nämlich als Medium für die **Kommunikation** und nicht nur für die reine Repräsentation. Wenn ein Sprecher mit dem Finger irgendwohin zeigt und „Schau!“ sagt, dann weiß der Zuhörer z.B., dass Superman endlich über den Dächern aufgetaucht ist. Wir würden jedoch nicht sagen, dass der Satz „Schau!“ genau diesen Fakt repräsentiert. Stattdessen ist die Bedeutung des Satzes sowohl vom Satz selbst als auch vom **Kontext**, in dem er gesprochen wurde, abhängig. Man könnte also offensichtlich einen Satz wie „Schau!“ nicht in einer Wissensbasis speichern und erwarten, dass seine Bedeutung wiederhergestellt werden kann, ohne nicht auch eine Repräsentation des Kontextes zu speichern – womit sich die Frage stellt, wie der Kontext dargestellt werden kann. Darüber hinaus leiden natürliche Sprachen unter der **Mehrdeutigkeit**, ein

Problem für eine Repräsentationssprache. Wie Pinker (1995) es ausdrückt: „Wenn Leute von einer *Feder* reden, dann gibt es selten Missverständnisse darüber, ob sie an Vogelfedern, Schreibgeräte oder ihr Auto denken – und auch wenn ein Wort zwei Gedanken entsprechen kann, können Gedanken keine Wörter sein."

Die berühmte **Sapir-Whorf-Hypothese** behauptet, dass unser Verständnis der Welt durch die Sprache, die wir sprechen, stark beeinflusst wird. Whorf (1956) schrieb: „Wir zerteilen die Natur, organisieren sie in Konzepten und schreiben ihr dabei Bedeutungen zu, hauptsächlich weil wir Teilnehmer an einer Übereinkunft sind, sie in dieser Weise zu organisieren – eine Übereinkunft, die für unsere Sprachgemeinschaft gilt und in den Mustern unserer Sprache kodifiziert ist." Es stimmt sicherlich, dass unterschiedliche Sprachgemeinschaften die Welt unterschiedlich gliedern. Der Franzose hat zwei Wörter „chaise" und „fauteuil" für ein Konzept, das englische Sprecher mit dem einen Wort „chair" abdecken (im Deutschen wiederum „Stuhl" und „Sessel"). Der englisch Sprechende kann aber die Kategorie „fauteuil" leicht erkennen und ihr einen Namen geben – ungefähr „open-arm chair". Macht also die Sprache wirklich einen Unterschied? Whorf stützte sich hauptsächlich auf Intuition und Spekulation, doch in der Zwischenzeit verfügen wir über reale Daten aus anthropologischen, psychologischen und neurologischen Untersuchungen.

Können Sie sich zum Beispiel erinnern, welcher der beiden folgenden Sätze die Einleitung zu *Abschnitt 8.1* gebildet hat?

> „In diesem Abschnitt beschreiben wir die Natur der Repräsentationssprachen ..."
>
> „Dieser Abschnitt befasst sich mit dem Thema der Wissensrepräsentationssprachen ..."

Wanner (1974) führte ein ähnliches Experiment durch und stellte fest, dass die Testpersonen die richtige Auswahl auf gut Glück – in etwa 50% der Fälle – trafen, sich aber an den Inhalt dessen, was sie gelesen hatten, mit einer Genauigkeit besser als 90% erinnern konnten. Dies legt nahe, dass Menschen die Wörter verarbeiten, um eine Art nonverbale Darstellung zu bilden.

Interessanter ist der Fall, in dem ein Konzept in einer Sprache vollkommen fehlt. Sprecher der australischen Aboriginie-Sprache Guugu Yimithirr haben keine Wörter für relative Richtungen wie zum Beispiel vorn, hinten, rechts oder links. Stattdessen verwenden sie absolute Richtungen und sagen beispielsweise im übertragenen Sinn: „Ich fühle einen Schmerz in meinem Nordarm." Dieser Unterschied in der Sprache macht einen Unterschied im Verhalten aus: Guugu Yimithirr-Sprecher schneiden beim Navigieren im offenen Gelände besser ab, während Englischsprechende besser darin sind, die Gabel rechts vom Teller zu platzieren.

Die Sprache scheint das Denken auch durch scheinbar willkürliche grammatische Merkmale wie etwa das Geschlecht von Substantiven zu beeinflussen. Zum Beispiel ist „Brücke" im Spanischen männlich und im Deutschen weiblich. Boroditsky (2003) forderte Testpersonen auf, englische Adjektive auszuwählen, um eine Fotografie einer bestimmten Brücke zu beschreiben. Spanische Sprecher wählten *big*, *dangerous*, *strong* und *towering*, während sich deutsche Sprecher für *beautiful*, *elegant*, *fragile* und *slender* entschieden. Wörter können als Ankerpunkte dienen, die beeinflussen, wie wir die Welt wahrnehmen. Loftus und Palmer (1974) zeigten Teilnehmern an einem Experiment einen Film eines Autounfalls. Probanden, die gefragt wurden: „How fast were the cars going when they contacted each other?" („Wie schnell waren die Autos, als sie sich

berührten?“) gaben im Durchschnitt 32 mph (52 km/h) an, während Probanden, denen die Frage mit dem Wort „smashed“ (zerschmettert) anstelle von „contacted“ gestellt wurde, 41 mph (66 km/h) für dieselben Autos im selben Film nannten.

In einem Schlusssystem mit Logik erster Stufe, das KNF verwendet, ist zu sehen, dass die linguistischen Formen „$\neg(A \vee B)$“ und „$\neg A \wedge \neg B$“ gleich sind, weil wir einen Blick in das System werfen und sehen können, dass die beiden Sätze in derselben kanonischen KNF-Form gespeichert sind. Können wir das mit dem menschlichen Gehirn auch tun? Bis vor Kurzem war die Antwort „Nein“, doch mittlerweile lautet sie „Eventuell“. Mitchell *et al.* (2008) haben Probanden in einen fMRT-Apparat (funktionelle Magnetresonanztomographie) gesteckt, ihnen Wörter wie zum Beispiel „Sellerie“ gezeigt und ihre Hirnaktivitäten aufgenommen. Die Forscher waren dann in der Lage, ein Computerprogramm zu trainieren, um aus einem Hirnbild vorherzusagen, welches Wort dem Probanden präsentiert worden ist. Bei zwei Auswahlmöglichkeiten (z.B. „Sellerie“ oder „Flugzeug“) sagt das System in 77% der Fälle korrekt voraus. Das System kann sogar überdurchschnittlich gut Wörter vorhersagen, von denen es vorher noch kein fMRT-Bild gesehen hat (indem es die Bilder von verwandten Wörtern betrachtet), und für Leute, die es noch nie gesehen haben (was beweist, dass fMRT eine gewisse Ebene allgemeiner Darstellung bei verschiedenen Menschen offenbart). Derartige Arbeiten stecken zwar noch in den Kinderschuhen, doch versprechen fMRT und andere Bildtechniken wie zum Beispiel intrakranielle Elektrophysiologie (Sahin *et al.*, 2009), uns konkretere Vorstellungen davon zu vermitteln, wie menschliche Wissensdarstellungen funktionieren.

Vom Standpunkt der formalen Logik aus gesehen macht die Darstellung desselben Wissens auf zwei unterschiedliche Arten absolut keinen Unterschied; aus jeder Darstellung lassen sich die gleichen Fakten ableiten. In der Praxis aber kann die eine Darstellung weniger Schritte erfordern, um einen Schluss abzuleiten. Ein Schlussfolgerungsmodul (Reasoner) mit begrenzten Ressourcen gelangt also möglicherweise mithilfe der einen Darstellung, nicht aber mit der anderen zum Schluss. Für nichtdeduktive Aufgaben wie zum Beispiel Lernen aus Erfahrung hängen die Ergebnisse zwangsläufig von der Form der verwendeten Darstellungen ab. Betrachtet ein lernendes Programm zwei mögliche Theorien der Welt, die beide mit allen Daten konsistent sind, ist es am zweckmäßigsten, sich für die prägnanteste Theorie zu entscheiden – was wiederum von der verwendeten Sprache abhängt, um Theorien darzustellen (mehr dazu in *Kapitel 18*). Somit ist der Einfluss der Sprache auf das Denken für jeden lernenden Agenten unvermeidbar.

8.1.2 Die Vorzüge der formalen und natürlichen Sprachen vereinen

Wir können die Grundlagen der Aussagenlogik übernehmen – eine deklarative, kompositionale Semantik, die kontextunabhängig und eindeutig ist – und darauf eine ausdruckskräftigere Logik aufbauen, wobei wir uns die Ideen zur Repräsentation von der natürlichen Sprache ausleihen und gleichzeitig ihre Nachteile vermeiden. Wenn wir die Syntax der natürlichen Sprache betrachten, sind die offensichtlichsten Elemente die Substantive und die Substantivausdrücke, die auf **Objekte** verweisen (Felder, Falltüren, Wumpi), sowie die Verben und Verbausdrücke, die auf **Relationen** zwischen den Objekten verweisen (ist windig, ist benachbart, schießt). Einige dieser Relationen sind **Funktionen** – Relationen, in denen es nur einen „Wert“ für eine bestimmte „Eingabe“ gibt. Man kann ganz einfach Beispiele für Objekte, Relationen und Funktionen auflisten:

- Objekte: Menschen, Häuser, Zahlen, Theorien, Ronald McDonald, Farben, Baseballspiele, Kriege, Jahrhunderte, ...
- Relationen: Dabei kann es sich um unäre Relationen oder **Eigenschaften** handeln, wie etwa rot, rund, falsch, weiblich, mehrgeschichtlich, ..., oder um allgemeinere *n*-äre Relationen, wie etwa Bruder von, größer als, innerhalb, Teil von, hat die Farbe, tritt auf nach, kommt zwischen, ...
- Funktionen: Vater von, bester Freund, Dritte von rechts, eins mehr als, Anfang von, ...

Tatsächlich kann man sich fast jede Behauptung so vorstellen, dass sie auf Objekte und Eigenschaften oder Relationen verweist. Einige Beispiele:

- „Eins plus zwei gleich drei."

  Objekte: Eins, zwei, drei, eins plus zwei; Relationen: gleich; Funktion: plus. („Eins plus zwei" ist ein Name für das Objekt, das man erhält, wenn man die Funktion „plus" auf die Objekte „eins" und „zwei" anwendet. Ein anderer Name für dieses Objekt ist „drei.")

- „Felder, die neben dem Wumpus liegen, stinken."

  Objekte: Wumpus, Felder; Eigenschaft: stinken; Relation: neben.

- „Der böse König John regierte England 1200."

  Objekte: John, England, 1200; Relation: regierte; Eigenschaften: böse; König.

Die Sprache der **Logik erster Stufe** (**First-Order-Logik**), deren Syntax und Semantik wir im nächsten Abschnitt definieren werden, ist um Objekte und Relationen herum aufgebaut. Sie war so wichtig für Mathematiker, Philosophen und die künstliche Intelligenz, weil man es gerade in diesen Bereichen – wie auch im täglichen Leben – mit Objekten und den Beziehungen zwischen ihnen zu tun hat. Die Logik erster Stufe kann Fakten über *einige* oder *alle* Objekte im Universum ausdrücken. Damit ist man in der Lage, allgemeine Gesetze oder Regeln darzustellen, wie etwa die Aussage: „Felder, die neben dem Wumpus liegen, stinken."

Der wichtigste Unterschied zwischen Aussagenlogik und Logik erster Stufe liegt in den durch jede Sprache getroffenen **ontologischen Bindungen** – d.h. was sie über die Natur der *Realität* voraussetzt. Mathematisch wird diese Bindung über die Natur der formalen **Modelle**, in denen die Wahrheit der Sätze definiert ist, ausgedrückt. Beispielsweise geht die Aussagenlogik davon aus, dass Fakten in der Welt gelten oder nicht gelten. Jeder Fakt kann sich in einem von zwei Zuständen befinden: wahr oder falsch, und jedes Modell weist jedem Aussagensymbol *true* oder *false* zu (siehe *Abschnitt 7.4.2*).[2] Die Logik erster Stufe setzt mehr voraus; nämlich dass die Welt aus Objekten besteht, die in bestimmten Beziehungen zueinander stehen, die gelten oder nicht gelten. Die formalen Modelle sind dementsprechend komplizierter als die für Aussagenlogik. Spezielle Logiken können im Hinblick auf die ontologischen Bindungen noch weiter gehen; die **temporale Logik** beispielsweise geht davon aus, dass Fakten zu bestimmten *Zeiten* gelten und diese Zeiten (wobei es sich um Zeitpunkte oder Zeitintervalle handeln kann) sortiert sind. Spezielle Logiken erteilen also bestimmten

2 Im Gegensatz dazu haben Fakten bei der **Fuzzy-Logik** einen **Wahrheitsgrad** zwischen 0 und 1. Zum Beispiel könnte der Satz „Wien ist eine große Stadt" in unserer Welt nur bis zu einem Grad von 0,6 in Fuzzy Logik wahr sein.

Objektarten (und den Axiomen über sie) einen „Erste-Klasse“-Status innerhalb der Logik, anstatt sie einfach innerhalb der Wissensbasis zu definieren. Die **Logik höherer Stufe** (**Higher-Order Logic**) betrachtet die Relationen und Funktionen, auf die die Logik erster Stufe verweist, als Objekte an sich. Damit ist es möglich, Zusicherungen im Hinblick auf *alle* Relationen zu treffen – z.B. könnte man definieren wollen, was es für eine Relation bedeutet, transitiv zu sein. Anders als die meisten speziellen Logiken ist die Logik höherer Stufe wirklich ausdrucksstärker als die Logik erster Stufe, weil einige Sätze der Logik höherer Stufe nicht durch eine endliche Anzahl von Sätzen der Logik erster Stufe ausgedrückt werden können.

Eine Logik kann auch durch ihre **epistemologischen Bindungen** charakterisiert werden – die möglichen Wissenszustände, die sie im Hinblick auf die verschiedenen Fakten erlaubt. Sowohl in der Aussagenlogik als auch in der Logik erster Stufe repräsentiert ein Satz einen Fakt und der Agent glaubt entweder, dass der Satz wahr ist, dass der Satz falsch ist, oder er hat keine Meinung. Diese Logiken haben also drei mögliche Wissenszustände zu jedem Satz. Systeme dagegen, die die **Wahrscheinlichkeitstheorie** anwenden, können einen *Glaubensgrad* haben, der von 0 (völliger Unglaube) bis 1 (völliger Glaube) reicht.[3] Beispielsweise könnte ein probabilistischer Agent aus der Wumpus-Welt glauben, dass sich der Wumpus mit einer Wahrscheinlichkeit von 0,75 in [1,3] befindet. Einen Überblick über die ontologischen und epistemologischen Bindungen fünf verschiedener Logiken finden Sie in ▶ Abbildung 8.1.

Sprache	Ontologische Bindung (was in der Welt existiert)	Epistemologische Bindung (was ein Agent im Hinblick auf Fakten glaubt)
Aussagenlogik	Fakten	Wahr/falsch/unbekannt
Logik erster Stufe	Fakten, Objekte, Relationen	Wahr/falsch/unbekannt
Temporale Logik	Fakten, Objekte, Relationen, Zeiten	Wahr/falsch/unbekannt
Wahrscheinlichkeitstheorie	Fakten	Glaubensgrad $\in$ [0,1]
Fuzzy Logik	Fakten mit einem Wahrheitsgrad $\in$ [0,1]	Bekannter Intervallwert

Abbildung 8.1: Formale Sprachen und ihre ontologischen und epistemologischen Bindungen.

Im nächsten Abschnitt werden wir die Details der Logik erster Stufe genauer betrachten. So wie ein Physikstudent ein gewisses mathematisches Grundwissen benötigt, muss ein Student der KI ein Talent dafür entwickeln, mit logischen Notationen zu arbeiten. Andererseits ist es aber auch wichtig, sich *nicht* zu sehr auf die *Besonderheiten* der logischen Notation zu konzentrieren – schließlich gibt es Dutzende verschiedener Versionen. Wichtig ist vor allem, wie die Sprache prägnante Repräsentationen unterstützt und wie ihre Semantik zu folgerichtigen Schlussprozeduren führt.

3 Verwechseln Sie nicht den Glaubensgrad der Wahrscheinlichkeitstheorie mit dem Wahrheitsgrad der Fuzzy Logik. Einige Fuzzy-Systeme unterstützen sogar eine Unsicherheit (Glaubensgrad) im Hinblick auf Wahrheitsgrade.

8.2 Syntax und Semantik der Logik erster Stufe

Wir beginnen diesen Abschnitt mit einer genaueren Beschreibung, wie die möglichen Welten der Logik erster Stufe die ontologische Bindung zu Objekten und Relationen widerspiegeln. Anschließend stellen wir die verschiedenen Elemente der Sprache vor und erklären dabei ihre Semantik.

8.2.1 Modelle für die Logik erster Stufe

Aus *Kapitel 7* wissen Sie, dass die Modelle einer logischen Sprache formale Strukturen sind, die die möglichen in Betracht gezogenen Welten bilden. Jedes Modell verknüpft das Vokabular der logischen Sätze mit Elementen der möglichen Welt, sodass sich die Wahrheit jedes Satzes ermitteln lässt. Modelle für die Aussagenlogik verknüpfen also Aussagensymbole mit vordefinierten Wahrheitswerten. Modelle für die Logik erster Stufe sind wesentlich interessanter. Erstens beinhalten sie Objekte! Die **Domäne** eines Modells ist die Menge der Objekte oder **Domänenelemente**, die es enthält. Die Domäne muss *nichtleer* sein – jede mögliche Welt muss mindestens ein Objekt enthalten (siehe Übung 8.7 für eine Diskussion von leeren Welten). Im mathematischen Sinne spielt es keine Rolle, *was* diese Objekte sind – es geht nur darum, *wie viele* es in jedem speziellen Modell gibt. Doch aus pädagogischen Gründen verwenden wir ein konkretes Beispiel. ▶ Abbildung 8.2 zeigt ein Modell mit fünf Objekten: Richard Löwenherz, König von England von 1189 bis 1199; seinen jüngeren Bruder, den bösartigen König John, der von 1199 bis 1215 regierte; die linken Beine von Richard und John und eine Krone.

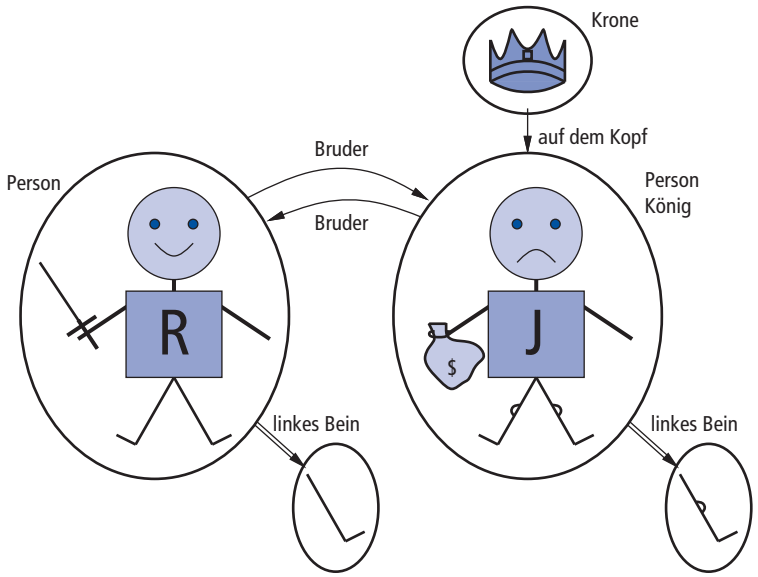


Abbildung 8.2: Ein Modell mit fünf Objekten, zwei binären Relationen, drei unären Relationen (durch Beschriftungen an den Objekten angezeigt) und einer unären Funktion, linkes Bein.

Die Objekte in dem Modell können auf verschiedene Arten miteinander in Beziehung stehen. In der Abbildung sind Richard und John Brüder. Formal ausgedrückt, ist eine Relation einfach die Menge der **Tupel** der Objekte, die in einer Beziehung zueinander

stehen. (Ein Tupel ist eine Sammlung von Objekten, die in fester Reihenfolge angeordnet sind, und wird mit spitzen Klammern um die Objekte geschrieben.) Somit ist die Relation *Brüder* in diesem Modell die Menge

$$\{\langle \text{Richard Löwenherz, König John}\rangle, \langle \text{König John, Richard Löwenherz}\rangle\} \tag{8.1}$$

(Wir haben hier den Objekten mithilfe der natürlichen Sprache Namen gegeben, Sie können die Namen gedanklich jedoch auch durch Bilder ersetzen.) Die Krone befindet sich auf dem Kopf von König John; die Relation „auf dem Kopf" enthält also nur ein Tupel, ⟨die Krone, König John⟩. Die Relationen „Bruder" und „auf dem Kopf" sind binäre Relationen – d.h., sie verknüpfen Objektpaare. Darüber hinaus enthält das Modell unäre Relationen oder Eigenschaften: Die Eigenschaft „Person" ist sowohl für Richard als auch für John wahr; die Eigenschaft „König" ist nur für John wahr (mutmaßlich, weil Richard zu diesem Zeitpunkt tot ist); und die Eigenschaft „Krone" ist nur für die Krone wahr.

Bestimmte Arten von Beziehungen werden am besten als Funktionen betrachtet, wobei ein bestimmtes Objekt genau zu einem Objekt in dieser Beziehung stehen muss. Beispielsweise hat jede Person ein linkes Bein; deshalb hat das Modell die unäre Funktion „linkes Bein", die die folgenden Zuordnungen beinhaltet:

$$\begin{aligned}&\langle \text{Richard Löwenherz}\rangle \rightarrow \text{Richards linkes Bein}\\ &\langle \text{König John}\rangle \rightarrow \text{Johns linkes Bein.}\end{aligned} \tag{8.2}$$

Streng betrachtet bedingen Modelle in der Logik erster Stufe **totale Funktionen**, d.h., es muss für jedes Eingabe-Tupel einen Wert geben. Die Krone muss also ein linkes Bein haben ebenso wie jedes der linken Beine. Es gibt eine technische Lösung für dieses unangenehme Problem, wobei ein zusätzliches „unsichtbares" Objekt eingeführt wird, das das linke Bein von allem darstellt, das selbst kein linkes Bein hat, einschließlich von sich selbst. Glücklicherweise haben diese technischen Angelegenheiten keine Bedeutung, solange man keine Bedingungen für linke Beine von Dingen angibt, die keine linken Beine haben.

Bislang haben wir die Elemente beschrieben, die Modelle für Logik erster Stufe füllen. Zu den anderen wichtigen Teilen eines Modells gehört die Verknüpfung zwischen diesen Elementen und dem Vokabular der logischen Sätze, was wir als Nächstes erläutern.

8.2.2 Symbole und Interpretationen

Wir kommen nun zur Syntax der Logik erster Stufe. Der ungeduldige Leser findet in ▶ Abbildung 8.3 einen vollständigen Überblick über die formale Grammatik.

Die grundlegenden syntaktischen Elemente der Logik erster Stufe sind die Symbole, die für Objekte, Relationen und Funktionen stehen. Es gibt also drei Arten von Symbolen: **Konstantensymbole**, die für Objekte stehen, **Prädikatssymbole**, die für Relationen stehen, und **Funktionssymbole**, die für Funktionen stehen. Wir übernehmen die Konvention, dass diese Symbole mit Großbuchstaben beginnen. Beispielsweise könnten wir die Konstantensymbole *Richard* und *John* verwenden, die Prädikatssymbole *Bruder*, *AufDemKopf*, *Person*, *König* und *Krone* und das Funktionssymbol *LinkesBein*. Wie bei den aussagenlogischen Symbolen bleibt die Namenswahl völlig dem Benutzer überlassen. Jedes Prädikats- und Funktionssymbol hat eine **Stelligkeit** oder **Arität**, die die Anzahl der Argumente angibt.

$$
\begin{aligned}
\mathit{Sentence} &\rightarrow \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} &\rightarrow \mathit{Predicate} \mid \mathit{Predicate}(\mathit{Term}, \ldots) \mid \mathit{Term} = \mathit{Term} \\
\mathit{ComplexSentence} &\rightarrow (\ \mathit{Sentence}\ ) \mid [\ \mathit{Sentence}\ ] \\
&\quad \mid \neg \mathit{Sentence} \\
&\quad \mid \mathit{Sentence} \wedge \mathit{Sentence} \\
&\quad \mid \mathit{Sentence} \vee \mathit{Sentence} \\
&\quad \mid \mathit{Sentence} \Rightarrow \mathit{Sentence} \\
&\quad \mid \mathit{Sentence} \Leftrightarrow \mathit{Sentence} \\
&\quad \mid \mathit{Quantifier}\ \mathit{Variable}, \ldots\ \mathit{Sentence} \\
\mathit{Term} &\rightarrow \mathit{Function}(\mathit{Term}, \ldots) \\
&\quad \mid \mathit{Constant} \\
&\quad \mid \mathit{Variable} \\
\mathit{Quantifier} &\rightarrow \forall \mid \exists \\
\mathit{Constant} &\rightarrow A \mid X_1 \mid \mathit{John} \mid \ldots \\
\mathit{Variable} &\rightarrow a \mid x \mid s \mid \ldots \\
\mathit{Predicate} &\rightarrow \mathit{True} \mid \mathit{False} \mid \mathit{After} \mid \mathit{Loves} \mid \mathit{Raining} \mid \ldots \\
\mathit{Function} &\rightarrow \mathit{Mother} \mid \mathit{LeftLeg} \mid \ldots
\end{aligned}
$$

OPERATOR PRECEDENCE : $\neg$, $=$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$

Abbildung 8.3: Die Syntax der Logik erster Stufe mit Gleichheit, angegeben in Backus-Naur-Form. (Weitere Informationen finden Sie in Anhang B, falls Sie mit dieser Notation nicht vertraut sind.) Die angegebenen Operatorrangfolgen geben die Reihenfolge von der höchsten zur niedrigsten Priorität an. Bei Quantoren gilt die Rangfolge, dass ein Quantor über allem gilt, was rechts von ihm steht.

Wie in der Aussagenlogik muss jedes Modell die erforderlichen Informationen bereitstellen, um zu ermitteln, ob ein gegebener Satz wahr oder falsch ist. Somit umfasst jedes Modell neben seinen Objekten, Relationen und Funktionen eine **Interpretation**, die genau angibt, auf welche Objekte, Relationen und Funktionen die Konstanten-, Prädikats- und Funktionssymbole verweisen. Eine mögliche Interpretation für unser Beispiel – die wir als **beabsichtigte Interpretation** bezeichnen wollen – lautet wie folgt:

- *Richard* bezieht sich auf Richard Löwenherz; *John* bezieht sich auf den bösen König John.
- *Bruder* bezieht sich auf die Bruderbeziehung, d.h. die Menge der Objekt-Tupel, die in Gleichung (8.1) angegeben sind; *AufDemKopf* bezieht sich auf die Relation „auf dem Kopf“, die zwischen der Krone und König John gilt; *Person*, *König* und *Krone* beziehen sich auf die Objektmengen, die Personen, Könige und Kronen sind.
- *LinkesBein* bezieht sich auf die Funktion „linkes Bein“, d.h. die in Gleichung (8.2) angegebene Zuordnung.

Es gibt viele andere mögliche Interpretationen für diese Symbole für das hier gezeigte Modell. Eine Interpretation bildet beispielsweise *Richard* auf die Krone ab und *John* auf das linke Bein von König John. Es gibt fünf Objekte in dem Modell und damit bereits 25 mögliche Interpretationen allein für die Konstantensymbole *Richard* und *John*. Beachten Sie, dass nicht alle Objekte einen Namen haben müssen – die beabsichtigte Interpretation gibt beispielsweise der Krone oder den Beinen keine Namen. Außerdem ist es möglich, dass ein Objekt mehrere Namen hat; es gibt eine Interpretation, unter der sowohl *Richard* als auch *John* auf die Krone verweisen.[4] Wenn Sie diese Möglichkeit

4 Später untersuchen wir in *Abschnitt 8.2.8* eine Semantik, in der jedes Objekt genau einen Namen besitzt.

verwirrt, überlegen Sie einfach, dass es in der Aussagenlogik durchaus möglich ist, ein Modell zu konstruieren, in dem sowohl *Wolkig* als auch *Sonnig* wahr sind; es ist die Aufgabe der Wissensbasis, Modelle auszuschließen, die inkonsistent zu unserem Wissen sind.

Alles in allem besteht ein Modell in Logik erster Stufe aus einer Menge von Objekten und einer Interpretation, die Konstantensymbole zu Objekten, Prädikatssymbole zu Relationen auf diesen Objekten und Funktionssymbole zu Funktionen auf diesen Objekten zuordnet. Genau wie bei Aussagenlogik sind logische Konsequenz, Gültigkeit usw. in Form *aller möglichen Modelle* definiert. ▶ Abbildung 8.4 vermittelt eine Vorstellung davon, wie die Menge aller möglichen Modelle aussieht. Die Abbildung zeigt, dass Modelle hinsichtlich der Anzahl der enthaltenen Objekte – von eins bis unendlich – und in der Art und Weise, wie Konstantensymbole zu Objekten zugeordnet werden, variieren. Gibt es zwei Konstantensymbole und ein Objekt, müssen beide Symbole auf dasselbe Objekt verweisen; dies kann aber auch bei mehreren Objekten passieren. Wenn es mehr Objekte als Konstantensymbole gibt, besitzen einige der Objekte keine Namen. Aufgrund der unbegrenzten Anzahl möglicher Modelle ist die Überprüfung der logischen Konsequenz durch Aufzählung aller möglichen Modelle für Logik erster Stufe (im Unterschied zur Aussagenlogik) nicht mehr praktikabel. Selbst wenn die Anzahl der Objekte begrenzt ist, kann die Anzahl der Kombinationen sehr groß sein (siehe Übung 8.5). Für das Beispiel in Abbildung 8.4 gibt es 137.506.194.466 Modelle mit sechs oder weniger Objekten.

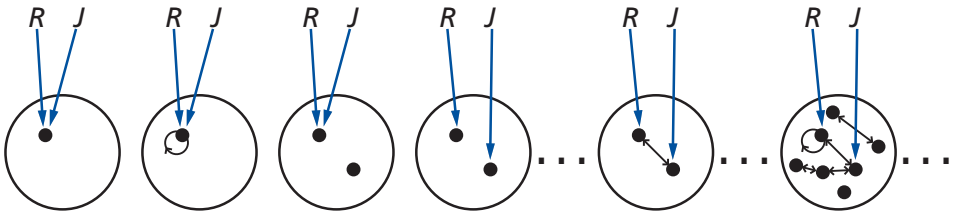


Abbildung 8.4: Einige Mitglieder der Menge aller Modelle für eine Sprache mit zwei Konstantensymbolen, *R* und *J*, und einem binären Relationssymbol. Die Interpretation jedes Konstantensymbols wird durch einen grauen Pfeil gezeigt. Innerhalb jedes Modells sind die in Beziehung stehenden Objekte durch Pfeile verbunden.

8.2.3 Terme

Ein **Term** ist ein logischer Ausdruck, der sich auf ein Objekt bezieht. Konstantensymbole sind also Terme, aber es ist nicht immer praktisch, für die Angabe jedes Objektes ein separates Symbol zu verwenden. In unserer natürlichen Sprache könnten wir etwa den Ausdruck „König Johns linkes Bein“ verwenden, anstatt seinem Bein einen Namen geben zu müssen. Und genau dafür sind Funktionssymbole vorgesehen: Statt eines Konstantensymbols verwenden wir *LinkesBein*(*John*). Im allgemeinen Fall wird ein komplexer Term durch ein Funktionssymbol gefolgt von einer in Klammern eingeschlossenen Liste mit Termen als Argumente für das Funktionssymbol gebildet. Beachten Sie, dass ein komplexer Term einfach nur eine komplizierte Art von Name ist. Es ist kein „Subroutinenaufruf“, der „einen Wert zurückgibt“. Es gibt keine Subroutine *LinkesBein*, die eine Person als Eingabe entgegennimmt und ein Bein zurückgibt. Wir können über linke Beine schließen (z.B. mit der allgemeinen Regel, dass jeder eines hat, und daraus geschlossen werden kann, dass auch John eines haben muss), ohne je

eine Definition von *LinkesBein* bereitzustellen. Mit Subroutinen in Programmiersprachen ist das nicht möglich.[5]

Die formale Semantik von Termen ist ganz einfach. Betrachten Sie einen Term $f(t_1, ..., t_n)$. Das Funktionssymbol f verweist auf irgendeine Funktion im Modell (nennen wir es F), die Argumentterme verweisen auf Objekte in der Domäne (nennen wir sie $d_1, ..., d_n$) und der Term als Ganzes verweist auf das Objekt, d.h. den Wert der Funktion F, angewendet auf $d_1, ..., d_n$. Angenommen, das Funktionssymbol *LinkesBein* verweist auf die in Gleichung (8.2) gezeigte Funktion und *John* verweist auf König John, dann verweist *LinkesBein*(*John*) auf das linke Bein von König John. Auf diese Weise legt die Interpretation jeden Verweis jedes Terms fest.

8.2.4 Atomare Sätze

Nachdem wir sowohl Terme für die Verweise auf Objekte als auch Prädikatssymbole für die Verweise auf Relationen haben, können wir sie kombinieren, um **atomare Sätze** zu konstruieren, die Fakten ausdrücken. Ein atomarer Satz (oder kurz **Atom**) wird aus einem Prädikatssymbol optional gefolgt von einer in Klammern eingeschlossenen Termliste gebildet, wie zum Beispiel:

Bruder(*Richard*, *John*).

Damit wird unter der zuvor formulierten beabsichtigten Interpretation ausgesagt, dass Richard Löwenherz der Bruder von König John ist.[6] Atomare Sätze können komplexe Terme als Argumente verwenden. Der Satz

Verheiratet(*Vater*(*Richard*), *Mutter*(*John*))

besagt also, dass der Vater von Richard Löwenherz mit der Mutter von König John verheiratet ist (ebenfalls unter einer geeigneten Interpretation).

Tipp

Ein atomarer Satz ist in einem bestimmten Modell unter einer bestimmten Interpretation ***wahr****, wenn die durch das Prädikatssymbol angegebene Relation für die von den Argumenten angegebenen Objekte gilt.*

8.2.5 Komplexe Sätze

Mithilfe **logischer Verknüpfungen** können wir komplexere Sätze mit der gleichen Syntax und Semantik wie im aussagenlogischen Kalkül konstruieren. So sind die folgenden vier Sätze im Modell gemäß Abbildung 8.2 unter unserer beabsichtigten Interpretation wahr:

5 λ-**Ausdrücke** sind eine praktische Notation, wobei neue Funktionssymbole „dynamisch“ erzeugt werden. Beispielsweise kann die Funktion, die ihr Argument quadriert, als $(\lambda x\, x \times x)$ geschrieben und wie jedes andere Funktionssymbol auf Argumente angewendet werden. Ein λ-Ausdruck kann auch als Prädikatssymbol definiert und verwendet werden (siehe *Kapitel 22*). Der Operator `lambda` in Lisp spielt genau dieselbe Rolle. Beachten Sie, dass die Verwendung von λ auf diese Weise *nicht* die formale Ausdruckskraft der Logik erster Stufe erhöht, weil jeder Satz, der einen λ-Ausdruck enthält, so umgeschrieben werden kann, dass seine Argumente „eingebaut“ werden, um einen äquivalenten Satz zu bilden.

6 Wir folgen normalerweise der Konvention für die Argumentreihenfolge, dass $P(x,y)$ als „x ist ein P von y“ gelesen wird.

$\neg Bruder(LinkesBein(Richard, John)$
$Bruder(Richard, John) \wedge Bruder(John, Richard)$
$König(Richard) \vee König(John)$
$\neg König(Richard) \Rightarrow König(John)$

8.2.6 Quantoren

Nachdem wir eine Logik haben, die Objekte zulässt, ist es nur natürlich, dass wir Eigenschaften ganzer Objektsammlungen ausdrücken wollen, anstatt die Objekte immer dem Namen nach aufzulisten. Dafür verwenden wir **Quantifizierer** (oder kurz **Quantoren**). Die Logik erster Stufe enthält zwei Standardquantoren, *universell* und *existentiell* (Allquantor und Existenzquantor).

Allquantor ($\forall$)

In *Kapitel 7* hatten wir die Schwierigkeit, allgemeine Regeln in der Aussagenlogik auszudrücken. Regeln wie etwa „Felder, die an das Feld mit dem Wumpus angrenzen, stinken“ oder „Alle Könige sind Personen“ sind Alltäglichkeiten in der Logik erster Stufe. Wir werden in *Abschnitt 8.3* auf die erste Regel zurückkommen. Die zweite Regel, „Alle Könige sind Personen“ wird in der Logik erster Stufe geschrieben als:

$$\forall\, x\ König(x) \Rightarrow Person(x).$$

$\forall$ wird normalerweise ausgesprochen als „Für alle...“. (Beachten Sie, dass das umgekehrte A für „Alle“ steht.) Der Satz besagt also: „Für alle x gilt, wenn x ein König ist, dann ist x eine Person.“ Das Symbol x wird als **Variable** bezeichnet. Per Konvention schreibt man Variablen in Kleinbuchstaben. Eine Variable ist an sich ein Term und kann als solcher Argument einer Funktion sein – z.B. $LinkesBein(x)$. Ein Term ohne Variablen wird als **Grundterm** bezeichnet.

Intuitiv besagt der Satz $\forall x\ P$, wobei P ein beliebiger logischer Ausdruck ist, dass P für jedes Objekt x wahr ist. Genauer gesagt, ist $\forall x\ P$ in einem bestimmten Modell wahr, wenn P in allen möglichen **erweiterten Interpretationen** wahr ist, die aus der gegebenen Interpretation konstruiert werden können, wobei jede erweiterte Interpretation ein Domänenelement spezifiziert, auf das x verweist.

Das hört sich kompliziert an, ist aber einfach nur eine sorgfältige Formulierung der intuitiven Bedeutung des Allquantors. Betrachten Sie das in Abbildung 8.2 gezeigte Modell und die ihm zugeordnete beabsichtigte Interpretation. Wir können die Interpretation auf fünf Arten erweitern:

$x \rightarrow$ Richard Löwenherz,
$x \rightarrow$ König John,
$x \rightarrow$ Richards linkes Bein,
$x \rightarrow$ Johns linkes Bein,
$x \rightarrow$ die Krone.

Der allquantifizierte Satz $\forall x\ König(x) \Rightarrow Person(x)$ ist unter der ursprünglichen Interpretation wahr, wenn der Satz $König(x) \Rightarrow Person(x)$ in jeder der fünf erweiterten Interpretationen wahr ist. Das bedeutet, der universell quantifizierte Satz ist äquivalent mit der Zusicherung der folgenden fünf Sätze:

Richard Löwenherz ist ein König ⇒ Richard Löwenherz ist eine Person.
König John ist ein König ⇒ König John ist eine Person.
Richards linkes Bein ist ein König ⇒ Richards linkes Bein ist eine Person.
Johns linkes Bein ist ein König ⇒ Johns linkes Bein ist eine Person.
Die Krone ist ein König ⇒ Die Krone ist eine Person.

Wir wollen diese Behauptungen sorgfältig betrachten. Weil König John in unserem Modell der einzige König ist, behauptet der zweite Satz, dass er eine Person ist, was wir hoffen wollen. Aber was ist mit den anderen vier Sätzen, die Behauptungen über Beine und Kronen treffen? Ist das Teil der Bedeutung von „Alle Könige sind Personen"? Die anderen vier Behauptungen sind im Modell wahr, aber treffen keine Aussagen darüber, ob sich Beine, Kronen oder sogar Richard als Personen qualifizieren. Das liegt daran, dass keines dieser Objekte ein König ist. Betrachten Sie die Wahrheitstabelle für ⇒ (siehe Abbildung 7.8). Hier sehen Sie, dass die Implikation wahr ist, wenn ihre Prämisse falsch ist – *unabhängig* von der Wahrheit der Schlussfolgerung. Durch Behauptung des allquantifizierten Satzes, der äquivalent mit der Behauptung einer ganzen Liste einzelner Implikationen ist, behaupten wir also die Schlussfolgerung der Regel nur für die Objekte, für die die Prämisse wahr ist, und sagen nichts über alle anderen Individuen aus, für die die Prämisse falsch ist. Die Wahrheitstabelleneinträge für ⇒ sind also hervorragend geeignet für die Formulierung allgemeiner Regeln mit Allquantoren.

Ein häufiger Fehler, der selbst von fleißigen Lesern gemacht wird, die diesen Abschnitt mehrfach gelesen haben, ist die Verwendung der Konjunktion statt der Implikation. Der Satz

$$\forall x\ König(x) \wedge Person(x)$$

wäre äquivalent mit der Behauptung

Richard Löwenherz ist ein König ∧ Richard Löwenherz ist eine Person
König John ist ein König ∧ König John ist eine Person
Richards linkes Bein ist ein König ∧ Richards linkes Bein ist eine Person

usw. Das trifft offensichtlich nicht das, was wir ausdrücken wollen.

Existenzquantor (∃)

Der Allquantor trifft Aussagen über jedes Objekt. Analog können wir eine Aussage zu einem *bestimmten* Objekt im Universum machen, ohne es per Namen anzugeben, indem wir einen Existenzquantor verwenden. Um beispielsweise zu sagen, dass König John eine Krone auf dem Kopf hat, schreiben wir:

$$\exists x\ Krone(x) \wedge AufDemKopf(x, John).$$

$\exists x$ wird ausgesprochen als: „Es gibt ein x, sodass gilt..." oder „Für ein x...".

Intuitiv besagt der Satz $\exists x\ P$, dass P für mindestens ein Objekt wahr ist. Genauer gesagt, ist $\exists x\ P$ in einem bestimmten Modell unter einer bestimmten Interpretation wahr, wenn P in *mindestens einer* erweiterten Interpretation wahr ist, die x einem Domänenelement zuweist. Für unser Beispiel bedeutet das, dass mindestens eine der folgenden Aussagen wahr sein muss:

Richard Löwenherz ist eine Krone ∧ Richard Löwenherz befindet sich auf dem Kopf von John.

König John ist eine Krone ∧ König John befindet sich auf dem Kopf von John.

Richards linkes Bein ist eine Krone ∧ Richards linkes Bein befindet sich auf dem Kopf von John.

Johns linkes Bein ist eine Krone ∧ Johns linkes Bein befindet sich auf dem Kopf von John.

Die Krone ist eine Krone ∧ Die Krone befindet sich auf dem Kopf von John.

Die fünfte Behauptung ist im Modell wahr; deshalb ist der ursprünglich existenzquantifizierte Satz im Modell wahr. Beachten Sie, dass der Satz unserer Definition gemäß auch in einem Modell wahr wäre, wo König John zwei Kronen trägt. Das ist vollständig konsistent mit dem ursprünglichen Satz: „König John hat eine Krone auf seinem Kopf."[7]

So wie ⇒ die natürliche Verknüpfung für die Verwendung in Kombination mit ∀ zu sein scheint, ist ∧ die natürliche Verknüpfung für die Verwendung in Kombination mit ∃. Verwendet man ∧ als Hauptverknüpfung in Kombination mit ∀, würde das zu einer überstarken Aussage im Beispiel des vorigen Abschnittes führen; die Verwendung von ⇒ in Kombination mit ∃ führt zu einer sehr schwachen Aussage. Betrachten Sie den folgenden Satz:

$$\exists x\ Krone(x) \Rightarrow AufDemKopf(x, John).$$

Oberflächlich betrachtet, sieht das wie eine vernünftige Wiedergabe unseres Satzes aus. Wendet man die Semantik an, erkennen wir, dass der Satz besagt, mindestens eine der folgenden Behauptungen sei wahr:

Richard Löwenherz ist eine Krone ⇒ Richard Löwenherz befindet sich auf dem Kopf von John;

König John ist eine Krone ⇒ König John befindet sich auf dem Kopf von John;

Richards linkes Bein ist eine Krone ⇒ Richards linkes Bein befindet sich auf dem Kopf von John;

usw. Jetzt ist eine Implikation wahr, wenn sowohl Prämisse als auch Schlussfolgerung wahr sind *oder wenn ihre Prämisse falsch ist*. Ist also Richard Löwenherz keine Krone, ist die erste Behauptung wahr und der Existenzquantor ist erfüllt. Ein existenzquantifizierter Implikationssatz ist also wahr, wenn ein *beliebiges* Objekt die Prämisse nicht erfüllen kann; solche Sätze haben also recht wenig Aussagekraft.

Verschachtelte Quantoren

Häufig wollen wir komplexere Sätze unter Verwendung mehrerer Quantoren ausdrücken. Im einfachsten Fall haben alle Quantoren denselben Typ. Beispielsweise kann der Satz „Brüder sind Geschwister" geschrieben werden als:

$$\forall x\ \forall y\ Bruder(x, y) \Rightarrow Geschwister(x, y).$$

7 Es gibt eine Variante des Existenzquantors, die häufig als $\exists^1$ oder ∃! dargestellt wird, was bedeutet: „Es gibt genau ein ..." Dieselbe Bedeutung kann unter Verwendung von Gleichheitsaussagen ausgedrückt werden.

Aufeinanderfolgende Quantoren desselben Typs können als Quantor mit mehreren Variablen geschrieben werden. Um beispielsweise auszudrücken, dass das Geschwistersein eine symmetrische Eigenschaft ist, können wir schreiben:

$$\forall x, y\ Geschwister(x, y) \Rightarrow Geschwister(y, x).$$

Es gibt aber auch gemischte Fälle. „Jeder liebt jemanden“ bedeutet, dass es für jede Person jemanden gibt, den diese Person liebt:

$$\forall x\ \exists y\ Liebt(x, y).$$

Um dagegen zu sagen: „Es gibt jemanden, der von jedem geliebt wird“, schreiben wir

$$\exists y\ \forall x\ Liebt(x, y).$$

Die Reihenfolge der Quantoren spielt hier eine große Rolle. Das wird deutlicher, wenn wir Klammern einfügen. $\forall x\ (\exists y\ Liebt(x, y))$ besagt, dass *jeder* eine bestimmte Eigenschaft hat, nämlich die Eigenschaft, dass jemand ihn liebt. Dagegen drückt $\exists x\ (\forall y\ Liebt(x, y))$ aus, dass *irgendjemand* auf der Welt eine bestimmte Eigenschaft hat, nämlich dass er von jedem geliebt wird.

Einige Verwirrung kann entstehen, wenn zwei Quantoren mit demselben Variablennamen verwendet werden. Betrachten Sie folgenden Satz:

$$\forall x\ [Krone(x) \vee (\exists x\ Bruder(Richard, x))].$$

Das x in $Bruder(Richard, x)$ ist hier *existenzquantifiziert*. Die Regel ist, dass die Variable zu dem innersten Quantor gehört, der sie verwendet; hierbei unterliegt sie keiner weiteren Quantifizierung. Man könnte sich das Ganze auch wie folgt vorstellen: $\exists x\ Bruder(Richard, x)$ ist ein Satz über Richard (dass er einen Bruder hat), nicht über x; setzt man also ein $\forall x$ außerhalb, hat das keine Wirkung. Man hätte genauso gut $\exists z\ Bruder(Richard, z)$ schreiben können. Weil das sehr verwirrend sein kann, verwenden wir bei verschachtelten Quantoren immer unterschiedliche Variablennamen.

Verbindungen zwischen ∀ und ∃

Die beiden Quantoren sind durch Negation eng miteinander verwandt. Die Behauptung, dass jeder Pastinak hasst, ist dasselbe wie die Behauptung, dass es niemanden gibt, der ihn mag, und umgekehrt:

$$\forall x\ \neg Mag(x, Pastinak) \quad \text{ist äquivalent mit} \quad \neg\exists x\ Mag(x, Pastinak).$$

Wir können noch einen Schritt weiter gehen. „Jeder mag Eis“ bedeutet, dass es niemanden gibt, der Eis nicht mag:

$$\forall x\ Mag(x, Eis) \quad \text{ist äquivalent mit} \quad \neg\exists x\ \neg Mag(x, Eis).$$

Weil ∀ eigentlich eine Konjunktion über das Universum der Objekte ist und ∃ eine Disjunktion, sollte es nicht überraschen, dass sie den De Morganschen Regeln gehorchen. Die De Morganschen Regeln für quantifizierte und nicht quantifizierte Sätze lauten:

$$\begin{array}{lcl} \forall x\ \neg P & \equiv & \neg\exists x\ P \\ \neg\forall x\ P & \equiv & \exists x\ \neg P \\ \forall x\ P & \equiv & \neg\exists x\ \neg P \\ \exists x\ P & \equiv & \neg\forall x\ \neg P \end{array} \qquad \begin{array}{lcl} \neg(P \vee \neg Q) & \equiv & \neg P \wedge Q \\ \neg(P \wedge Q) & \equiv & \neg P \vee \neg Q \\ P \wedge Q & \equiv & \neg(\neg P \vee \neg Q) \\ P \vee Q & \equiv & \neg(\neg P \wedge \neg Q) \end{array}$$

Wir brauchen also eigentlich nicht sowohl $\forall$ als auch $\exists$, ebenso wie wir eigentlich nicht sowohl $\wedge$ als auch $\vee$ brauchen. Die Lesbarkeit ist jedoch immer noch wichtiger als die Sparsamkeit; deshalb behalten wir beide Quantoren bei.

8.2.7 Gleichheit

Die Logik erster Stufe beinhaltet neben der Verwendung eines Prädikats und mehrerer Terme, wie zuvor beschrieben, eine weitere Möglichkeit, atomare Sätze zu erstellen. Wir können das **Gleichheitssymbol** verwenden, um Aussagen zu treffen, dass sich zwei Terme auf dasselbe Objekt beziehen. Zum Beispiel besagt

$$Vater(John) = Henry,$$

dass das Objekt, auf das *Vater*(*John*) verweist, und das Objekt, auf das *Henry* verweist, ein und dasselbe sind. Weil eine Interpretation die Verweise jedes Terms festlegt, ist die Feststellung der Wahrheit eines Gleichheitssatzes einfach eine Überprüfung, ob die Verweise der beiden Terme dasselbe Objekt bezeichnen.

Das Gleichheitssymbol kann verwendet werden, um Fakten über eine bestimmte Funktion auszusagen, wie wir es eben für das *Vater*-Symbol gemacht haben. Es kann auch in Verbindung mit einer Negierung verwendet werden, um zu besagen, dass zwei Terme nicht dasselbe Objekt darstellen. Um zu sagen, dass Richard mindestens zwei Brüder hat, schreiben wir:

$$\exists x, y \;\; Bruder(x, Richard) \wedge Bruder(y, Richard) \wedge \neg(x = y).$$

Der Satz:

$$\exists x, y \;\; Bruder(x, Richard) \wedge Bruder(y, Richard)$$

hat nicht die beabsichtigte Bedeutung. Insbesondere ist er wahr im Modell von Abbildung 8.2, wo Richard nur einen Bruder hat. Um dies zu erkennen, betrachten Sie die erweiterte Interpretation, wo sowohl x als auch y an *König John* zugewiesen werden. Durch das Hinzufügen von $\neg(x = y)$ werden solche Modelle ausgeschlossen. Manchmal wird als Abkürzung für $\neg(x = y)$ die Notation $x \neq y$ verwendet.

8.2.8 Eine alternative Semantik?

Wir setzen das Beispiel aus dem vorherigen Abschnitt fort und nehmen an, dass wir glauben, dass Richard zwei Brüder, John und Geoffrey, hat.[8] Können wir diesen Tatbestand erfassen, indem wir aussagen

$$Bruder(John, Richard) \wedge Bruder(Geoffrey, Richard). \tag{8.3}$$

Nicht ganz. Erstens ist diese Behauptung wahr in einem Modell, in dem Richard nur einen Bruder hat – wir müssen $John \neq Geoffrey$ hinzufügen. Zweitens filtert der Satz keine Modelle aus, in denen Richard viele andere Brüder außer John und Geoffrey hat. Somit lautet die korrekte Übersetzung von „Die Brüder von Richard sind John und Geoffrey“ wie folgt:

$$Bruder(John, Richard) \wedge Bruder(Geoffrey, Richard) \wedge John \neq Geoffrey \\ \wedge \forall x \; Bruder(x, Richard) \Rightarrow (x = John \vee x = Geoffrey).$$

8 Tatsächlich hatte er vier; die beiden anderen heißen William und Henry.

Für viele Zwecke scheint dies erheblich umständlicher als der entsprechende Ausdruck in natürlicher Sprache. Deshalb können Menschen Fehler machen, wenn sie ihr Wissen in Logik erster Stufe übersetzen, was zu nicht intuitiven Verhaltensweisen aus logischen Schlusssystemen führt, die das Wissen verwenden. Lässt sich eine Semantik entwerfen, die einen übersichtlicheren logischen Ausdruck ermöglicht?

Ein Ansatz, den man häufig in Datenbanksystemen findet, funktioniert folgendermaßen: Erstens bestehen wir darauf, dass jedes Konstantensymbol auf ein anderes Objekt verweist – das ist die sogenannte **Annahme eindeutiger Namen**. Zweitens gehen wir davon aus, dass atomare Sätze, die nicht als wahr bekannt sind, tatsächlich falsch sind – die **Annahme zur Weltabgeschlossenheit**. Schließlich berufen wir uns auf **Domänenabgeschlossenheit**. Das bedeutet, dass jedes Modell außer den durch Konstantensymbole benannten Elementen keine weiteren Domänenelemente enthält. Gemäß der resultierenden Semantik, die wir als **Datenbanksemantik** bezeichnen, um sie von der Standardsemantik der Logik erster Stufe zu unterscheiden, stellt der Satz in Gleichung (8.3) tatsächlich fest, dass John und Geoffrey die beiden Brüder von Richard sind. Datenbanksemantik wird auch in Logikprogrammiersystemen verwendet, wie es *Abschnitt 9.4.5* noch erläutert.

Es ist recht lehrreich, die Menge aller möglichen Modelle für den gleichen Fall wie in Abbildung 8.4 gezeigt unter Datenbanksemantik zu betrachten. ▶ Abbildung 8.5 zeigt einige Modelle, die vom Modell, bei dem kein Tupel die Relation erfüllt, bis zum Modell, bei dem alle Tupel die Relation erfüllen, reichen. Mit zwei Objekten lassen sich vier mögliche Tupel aus zwei Elementen bilden, sodass es $2^4 = 16$ unterschiedliche Teilmengen von Tupeln gibt, die die Relation erfüllen können. Somit gibt es insgesamt 16 mögliche Modelle – erheblich weniger als die unendliche Anzahl von Modellen für die Standardsemantik der Logik erster Stufe. Andererseits setzt die Datenbanksemantik definitives Wissen darüber voraus, was die Welt enthält.

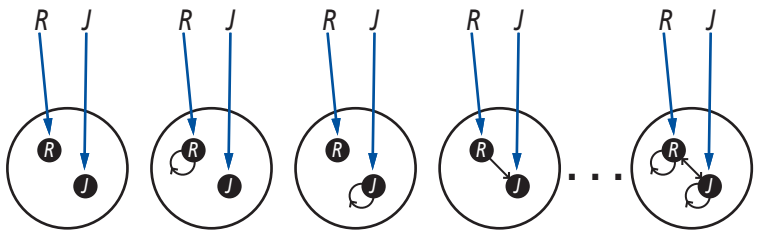


Abbildung 8.5: Einige Mitglieder der Menge aller Modelle für eine Sprache mit zwei Konstantensymbolen, R und J, und einem binären Relationssymbol unter Datenbanksemantik. Die Interpretation der Konstantensymbole ist feststehend und für jedes Konstantensymbol gibt es ein eindeutiges Objekt.

Dieses Beispiel macht einen wichtigen Punkt deutlich: Es gibt nicht nur genau eine „korrekte“ Semantik für Logik. Die Nützlichkeit jeder vorgeschlagenen Semantik hängt davon ab, wie prägnant und intuitiv sich das Wissen ausdrücken lässt, das wir formulieren wollen, und wie einfach und natürlich sie ist, um die entsprechenden Inferenzregeln zu entwickeln. Datenbanksemantik ist am praktischsten, wenn wir über die Identität aller Objekte, die in der Wissensbasis beschrieben sind, sicher sind und wenn wir alle Fakten zur Hand haben; in anderen Fällen ist sie ziemlich umständlich. Für den Rest dieses Kapitels nehmen wir die Standardsemantik an und weisen auf Instanzen hin, in denen diese Wahl zu umständlichen Ausdrücken führt.

8.3 Anwendung der Logik erster Stufe

Nachdem wir eine ausdrucksstarke Logiksprache definiert haben, wollen wir betrachten, wie sie angewendet wird. Das erfolgt am besten anhand von Beispielen. Wir haben einige einfache Sätze gesehen, die die verschiedenen Aspekte logischer Syntax demonstriert haben. In diesem Abschnitt zeigen wir weitere systematische Repräsentationen einfacher **Domänen**. In der Wissensrepräsentation ist eine Domäne einfach ein Teil der Welt, über die wir ein bestimmtes Wissen ausdrücken wollen.

Wir beginnen mit einer kurzen Beschreibung der TELL/ASK-Schnittstelle für Wissensbasen erster Stufe. Anschließend betrachten wir die Domänen von Verwandtschaftsbeziehungen, Zahlen, Mengen und Listen sowie der Wumpus-Welt. Der nächste Abschnitt enthält ein umfangreicheres Beispiel (elektronische Schaltkreise) und in *Kapitel 12* geht es um das gesamte Universum.

8.3.1 Zusicherungen und Abfragen in der Logik erster Stufe

Sätze werden einer Wissensbasis mithilfe von TELL hinzugefügt, genau wie bei der Aussagenlogik. Solche Sätze werden als **Zusicherungen** bezeichnet. Beispielsweise können wir zusichern, dass John ein König ist und dass Könige Personen sind:

$$\text{TELL}(KB, König(John))$$

$$\text{TELL}(KB, Person(Richard))$$

$$\text{TELL}(KB, \forall x \;\; König(x) \Rightarrow Person(x)).$$

Mit ASK können wir der Wissensbasis Fragen stellen, z.B.:

$$\text{ASK}(KB, König(John)).$$

Diese Abfrage ergibt *true*. Bei Fragen, die mit ASK gestellt werden, spricht man von **Abfragen** oder **Zielen** (nicht zu verwechseln mit den Zielen, die zur Beschreibung des gewünschten Zustandes eines Agenten verwendet werden). Allgemein ausgedrückt, sollte jede Abfrage, die aus der Wissensbasis logisch folgt, bejahend beantwortet werden. Für die beiden Zusicherungen aus dem vorigen Abschnitt sollte beispielsweise die Abfrage

$$\text{ASK}(KB, Person(John))$$

ebenfalls *true* ergeben. Wir können auch quantifizierte Abfragen stellen, wie etwa:

$$\text{ASK}(KB, \exists x \; Person(x)).$$

Die Antwort auf diese Abfrage könnte *true* sein, doch ist dies möglicherweise nicht so hilfreich, wie wir es gern hätten. Es ist vielmehr vergleichbar damit, wie auf die Frage „Können Sie mir die Uhrzeit sagen?“ mit „Ja“ geantwortet wird. Möchten wir wissen, mit welchem Wert von x der Satz wahr wird, brauchen wir eine andere Funktion, ASKVARS, die wir mit

$$\text{ASKVARS}(KB, Person(x))$$

aufrufen und die einen Strom von Antworten liefert. In diesem Fall gibt es zwei Antworten: $\{x/John\}$ und $\{x/Richard\}$. Eine derartige Antwort heißt **Substitution** oder **Bindungsliste**. Die Funktion ASKVARS ist normalerweise für Wissensbasen reserviert, die ausschließlich aus Horn-Klauseln bestehen, weil in derartigen Wissensbasen jeder Weg, die Abfrage wahr zu machen, die Variablen an spezifische Werte bindet. Bei Logik erster Stufe

ist das nicht der Fall; wenn *KB* mitgeteilt worden ist, dass $König(John) \vee König(Richard)$ gilt, dann gibt es keine Bindung an x für die Abfrage $\exists x\ König(x)$, selbst wenn die Abfrage wahr ist.

8.3.2 Die Verwandtschaftsdomäne

Als erstes Beispiel betrachten wir die Domäne der Familienbeziehungen oder Verwandtschaften. Diese Domäne beinhaltet Fakten wie etwa „Elisabeth ist die Mutter von Charles“ und „Charles ist der Vater von William“ sowie Regeln wie „Die eigene Großmutter ist die Mutter eines eigenen Elternteils“.

Offensichtlich sind die Objekte unserer Domäne Menschen. Wir haben zwei unäre Prädikate, *Männlich* und *Weiblich*. Verwandtschaftsverhältnisse – Eltern, Brüder, verheiratet usw. – werden durch binäre Prädikate dargestellt: *Eltern, Geschwister, Bruder, Schwester, Kind, Tochter, Sohn, Gatte, Ehemann, Ehefrau, Großeltern, Enkel, Cousin, Tante* und *Onkel*. Wir verwenden die Funktionen für *Mutter* und *Vater*, weil jeder Mensch genau eines davon hat (zumindest hat die Natur es so geplant).

Wir können die Funktionen und Prädikate durchlaufen und aufschreiben, was wir hinsichtlich der anderen Symbole wissen. Beispielsweise ist die Mutter der weibliche Elternteil:

$$\forall m, c\ \ Mutter(c) = m \Leftrightarrow Weiblich(m) \wedge Eltern(m, c)$$

Der Ehemann ist der männliche Gatte:

$$\forall w, h\ \ Ehemann(h, w) \Leftrightarrow Männlich(m) \wedge Gatte(h, w)$$

Männlich und weiblich sind disjunkte Kategorien:

$$\forall x\ \ Männlich(x) \Leftrightarrow \neg Weiblich(x)$$

Eltern und Kind sind inverse Relationen:

$$\forall p, c\ \ Eltern(p, c) \Leftrightarrow Kind(c, p)$$

Ein Großelternteil ist ein Elternteil eines eigenen Elternteils:

$$\forall g, c\ \ Großeltern(g, c) \Leftrightarrow \exists\ p\ Eltern(g, p) \wedge Eltern(p, c)$$

Ein Geschwister ist ein weiteres Kind der eigenen Eltern:

$$\forall x, y\ \ Geschwister(x, y) \Leftrightarrow x \neq y\ \wedge\ \exists p\ Eltern(p, x) \wedge Eltern(p, y)$$

Das könnten wir über mehrere Seiten fortsetzen und in Übung 8.11 werden Sie genau das tun.

Jeder dieser Sätze kann als **Axiom** der Verwandtschaftsdomäne betrachtet werden, wie *Abschnitt 7.1* erläutert hat. Axiome sind im Allgemeinen rein mathematischen Domänen zugeordnet – wir werden später noch Axiome für Zahlen zeigen –, aber sie werden in allen Domänen benötigt. Sie bieten die grundlegende faktische Information, aus der sinnvolle Schlüsse gezogen werden können. Unsere Verwandtschaftsaxiome sind auch **Definitionen**; sie haben die Form $\forall x, y\ \ P(x, y) \Leftrightarrow \ldots$ Die Axiome definieren die Funktion *Mutter* und die Prädikate *Ehemann, Männlich, Eltern, Großeltern* und *Geschwister* mithilfe anderer Prädikate. Unsere Definitionen bilden eine grundlegende Menge von Prädikaten (*Kind, Gatte* und *Weiblich*), mit denen die anderen definiert werden. Das ist eine sehr natürliche Vorgehensweise, die Repräsentation einer Domäne zu erstellen,

und sie ist vergleichbar mit der Vorgehensweise, wie Softwarepakete durch aufeinanderfolgende Definitionen von Subroutinen aus primitiven Bibliotheksfunktionen aufgebaut werden. Beachten Sie, dass es nicht unbedingt eine einzige Menge primitiver Prädikate geben muss; wir hätten auch genauso gut *Eltern*, *Gatte* und *Männlich* verwenden können. Wir werden noch sehen, dass es in einigen Domänen keine deutlich erkennbare Grundmenge gibt.

Nicht alle logischen Sätze über eine Domäne sind Axiome. Einige davon sind **Theoreme** – d.h., sie ergeben sich aus den Axiomen. Betrachten Sie beispielsweise die Zusicherung, dass das Geschwistersein symmetrisch ist:

$$\forall x, y \;\; Geschwister(x, y) \Leftrightarrow Geschwister(y, x).$$

Ist dies ein Axiom oder ein Theorem? Es ist ein Theorem, das logisch aus dem Axiom folgt, welches das Geschwistersein definiert. Wenn wir die Wissensbasis nach diesem Satz mit ASK abfragen, sollte sie *true* zurückgeben.

Aus rein logischer Perspektive muss eine Wissensbasis nur Axiome und keine Theoreme enthalten, weil die Theoreme die Menge der Schlussfolgerungen nicht erhöhen, die aus der Wissensbasis folgen. Aus praktischer Perspektive sind Theoreme jedoch wichtig, um den Rechenaufwand bei der Ableitung neuer Sätze gering zu halten. Ohne sie muss ein Schlusssystem bei jeder Abfrage ganz von vorne anfangen, so als müsste ein Mathematiker für jedes neue Problem die Rechenregeln neu erfinden.

Nicht alle Axiome sind Definitionen. Einige enthalten allgemeinere Informationen über bestimmte Prädikate, ohne eine Definition darzustellen. Einige Prädikate haben keine vollständige Definition, weil wir nicht genügend wissen, um sie vollständig zu charakterisieren. Beispielsweise gibt es keine offensichtliche Möglichkeit, den folgenden Satz zu vervollständigen:

$$\forall x \;\; Person(x) \Leftrightarrow ...$$

Glücklicherweise erlaubt uns die Logik erster Stufe, das Prädikat *Person* zu verwenden, ohne es vollständig zu definieren. Stattdessen können wir partielle Spezifikationen von Eigenschaften schreiben, die jede Person besitzt und die ein Objekt zu einer Person machen:

$$\forall x \;\; Person(x) \Rightarrow ...$$
$$\forall x \;\; ... \Rightarrow Person(x).$$

Axiome können auch „einfache Tatsachen“ sein, wie etwa *Männlich*(*Jim*) und *Gatte*(*Jim*, *Laura*). Solche Fakten bilden die Beschreibungen spezifischer Probleminstanzen, wodurch es möglich wird, dass spezifische Fragen beantwortet werden. Die Antworten auf diese Fragen sind dann Theoreme, die aus den Axiomen folgen. Häufig stellt man fest, dass die erwarteten Antworten nicht auftreten – zum Beispiel würde man von *Gatte*(*Jim*, *Laura*) (unter Beachtung der Gesetzeslage in vielen Ländern) erwarten, $\neg$*Gatte*(*George*, *Laura*) ableiten zu können; doch folgt das nicht aus den zuvor angegebenen Axiomen – selbst nachdem wir *Jim* $\neq$ *George* wie in *Abschnitt 8.2.8* vorgeschlagen hinzufügen. Das ist ein Zeichen dafür, dass ein Axiom fehlt. In Übung 8.8 werden Sie es ergänzen.

8.3.3 Zahlen, Mengen und Listen

Zahlen sind vielleicht das beste Beispiel dafür, wie aus einem winzigen Kern von Axiomen eine große Theorie aufgebaut werden kann. Wir beschreiben hier die Theorie der **natürlichen Zahlen**, der nicht negativen ganzen Zahlen. Wir brauchen ein Prädikat *NatNum*, das für natürliche Zahlen wahr ist; wir brauchen ein Konstantensymbol, 0, und wir brauchen ein Funktionssymbol, S (Successor, Nachfolger). Die **Peano-Axiome** definieren natürliche Zahlen und die Addition.[9] Natürliche Zahlen sind rekursiv definiert:

$$NatNum(0)$$
$$\forall n \;\; NatNum(n) \Rightarrow NatNum(S(n)).$$

Das bedeutet, 0 ist eine natürliche Zahl, und für jedes Objekt n gilt: Wenn n eine natürliche Zahl ist, dann ist $S(n)$ eine natürliche Zahl. Die natürlichen Zahlen sind also 0, $S(0)$, $S(S(0))$ usw. (Nachdem Sie *Abschnitt 8.2.8* gelesen haben, werden Sie feststellen, dass diese Axiome auch andere natürliche Zahlen außer den üblichen zulassen; siehe Übung 8.13.) Wir brauchen auch Axiome, die die Nachfolgerfunktion beschränken:

$$\forall n \;\; 0 \neq S(n)$$
$$\forall m, n \;\; m \neq n \Rightarrow S(m) \neq S(n).$$

Jetzt können wir die Addition mithilfe der Nachfolgerfunktion definieren:

$$\forall m \;\; NatNum(m) \Rightarrow +(m, 0) = m$$
$$\forall m, n \;\; NatNum(m) \wedge NatNum(n) \Rightarrow +(S(m), n) = S(+(m, n)).$$

Das erste dieser Axiome besagt, dass das Addieren von 0 zu einer natürlichen Zahl m wieder m ergibt. Beachten Sie die Verwendung des binären Funktionssymbols „+“ im Term $+(m, 0)$; in der normalen Mathematik würde der Term unter Verwendung der **Infixnotation** als m + 0 geschrieben. (Die Notation, die wir für die Logik erster Stufe verwendet haben, heißt **Präfixnotation**.) Um unsere Sätze über Zahlen leichter lesbar zu machen, erlauben wir die Verwendung der Infixnotation. Wir können $S(n)$ auch als $n + 1$ schreiben, sodass das zweite Axiom wie folgt aussieht:

$$\forall m, n \;\; NatNum(m) \wedge NatNum(n) \Rightarrow (m + 1) + n = (m + n) + 1.$$

Dieses Axiom reduziert die Addition auf die wiederholte Anwendung der Nachfolgerfunktion.

Die Verwendung der Infixnotation ist ein Beispiel für **syntaktischen Zucker**, d.h. eine Erweiterung oder Abkürzung der Standardsyntax, welche die Semantik nicht verändert. Jeder Satz, der diesen Zucker verwendet, kann „entzuckert“ werden, um einen äquivalenten Satz in normaler Logik erster Stufe zu erzeugen.

Nachdem wir die Addition haben, ist es ganz einfach, die Multiplikation als wiederholte Addition, die Potenzierung als wiederholte Multiplikation, die ganzzahlige Division mit Rest, die Primzahlen usw. zu definieren. Die gesamte Zahlentheorie (einschließlich Kryptografie) lässt sich also aus einer Konstanten, einer Funktion, einem Prädikat und vier Axiomen aufbauen.

9 Die Peano-Axiome enthalten auch das Induktionsprinzip. Dies ist ein Satz der Logik zweiter Stufe und nicht der Logik erster Stufe. Die Bedeutung dieses Unterschiedes wird in *Kapitel 9* erklärt.

Die Domäne der **Mengen** ist grundlegend für die Mathematik und auch für das Schließen mit gesundem Menschenverstand. (Es ist sogar möglich, die Zahlentheorie in Form der Mengentheorie zu definieren.) Wir wollen in der Lage sein, einzelne Mengen darzustellen, einschließlich der leeren Menge. Wir brauchen eine Möglichkeit, Mengen aufzubauen, indem wir ein Element hinzufügen oder die Vereinigung oder Schnittmenge von zwei Mengen bilden. Wir wollen wissen, ob ein Element zu einer Menge gehört, und wir wollen Mengen unterscheiden können von Objekten, die keine Mengen sind.

Wir verwenden das normale Vokabular der Mengentheorie als syntaktischen Zucker. Die leere Menge ist eine Konstante, die man als {} schreibt. Es gibt ein unäres Prädikat, *Set*, das für Mengen wahr ist. Die binären Prädikate sind $x \in s$ (x ist Element von s) und $s_1 \subseteq s_2$ (Menge s_1 ist eine, nicht unbedingt echte, Untermenge von s_2). Die binären Funktionen sind $s_1 \cap s_2$ (die Schnittmenge von zwei Mengen), $s_1 \cup s_2$ (die Vereinigungsmenge von zwei Mengen) und $\{x|s\}$ (die Menge, die durch das Hinzufügen des Elements x zur Menge s entsteht). Eine mögliche Menge von Axiomen sieht wie folgt aus:

1 Die einzigen Mengen sind die leere Menge und Mengen, die entstehen, indem man einer Menge etwas hinzufügt:

$$\forall s \;\; Set(s) \Leftrightarrow (s = \{\}) \vee (\exists x, s_2 \;\; Set(s_2) \wedge s = \{x \mid s_2\}).$$

2 Der leeren Menge wurden keine Elemente hinzugefügt. Mit anderen Worten gibt es keine Möglichkeit, {} in eine kleinere Menge und ein Element zu zerlegen:

$$\neg \exists x, s \;\; \{x|s\} = \{\}.$$

3 Das Hinzufügen eines Elements, das sich bereits in der Menge befindet, hat keine Wirkung:

$$\forall x, s \;\; x \in s \Leftrightarrow s = \{x \mid s\}.$$

4 Die einzigen Elemente einer Menge sind die Elemente, die ihr hinzugefügt wurden. Wir drücken dies rekursiv aus und sagen, dass x genau dann ein Element von s ist, wenn s gleich einer Menge s_2 ist, der ein Element y hinzugefügt wurde, wobei entweder y dasselbe wie x oder x ein Element von s_2 ist:

$$\forall x, s \;\; x \in s \Leftrightarrow \exists y, s_2 \;\; (s = \{y \mid s_2\} \wedge (x = y \vee x \in s_2)).$$

5 Eine Menge ist genau dann eine Untermenge einer anderen Menge, wenn alle Elemente der ersten Menge Elemente der zweiten Menge sind:

$$\forall s_1, s_2 \;\; s_1 \subseteq s_2 \Leftrightarrow (\forall x \;\; x \in s_1 \;\Rightarrow\; x \in s_2).$$

6 Zwei Mengen sind genau dann gleich, wenn jede eine Untermenge der anderen ist:

$$\forall s_1, s_2 \;\; (s_1 = s_2) \Leftrightarrow (s_1 \subseteq s_2 \wedge s_2 \subseteq s_1).$$

7 Ein Objekt befindet sich genau dann in der Schnittmenge von zwei Mengen, wenn es Element beider Mengen ist:

$$\forall x, s_1, s_2 \;\; x \in (s_1 \cap s_2) \Leftrightarrow (x \in s_1 \wedge x \in s_2).$$

8 Ein Objekt befindet sich genau dann in der Vereinigungsmenge von zwei Mengen, wenn es Element beider Mengen ist:

$$\forall x, s_1, s_2 \;\; x \in (s_1 \cup s_2) \Leftrightarrow (x \in s_1 \vee x \in s_2).$$

Listen sind mit Mengen vergleichbar. Der Unterschied ist, dass Listen sortiert sind und in einer Liste dasselbe Element mehrfach erscheinen kann. Wir können für Listen das Lisp-Vokabular verwenden: *Nil* ist die konstante Liste, die keine Elemente enthält; *Cons*, *Append*, *First* und *Rest* sind Funktionen; *Find* ist das Prädikat, das für Listen das macht, was *Member* für Mengen macht. *List?* ist ein Prädikat, das nur für Listen wahr ist. Wie bei Mengen ist es üblich, syntaktischen Zucker in logischen Sätzen über Listen zu verwenden. Die leere Liste ist []. Der Term $Cons(x, y)$, wobei y eine nicht leere Liste ist, wird als $[x \mid y]$ geschrieben. Der Term $Cons(x, Nil)$ (d.h. die Liste mit dem Element x) wird als $[x]$ geschrieben. Eine Liste mit mehreren Elementen wie z.B. $[A, B, C]$ entspricht dem verschachtelten Term $Cons(A, Cons(B, Cons(C, Nil)))$. In Übung 8.17 werden Sie die Axiome für Listen formulieren.

8.3.4 Die Wumpus-Welt

In *Kapitel 7* wurden einige aussagenlogische Axiome für die Wumpus-Welt formuliert. Die Axiome der Logik erster Stufe in diesem Abschnitt sind sehr viel prägnanter und drücken auf sehr natürliche Weise genau das aus, was wir sagen wollen.

Wie Sie wissen, erhält der Wumpus-Agent einen Wahrnehmungsvektor mit fünf Elementen. Der entsprechende Satz in der Wissensbasis der Logik erster Stufe muss sowohl die Wahrnehmung als auch die Zeit, zu der sie aufgetreten ist, enthalten; andernfalls wird der Agent verwirrt darüber, was er wann wahrgenommen hat. Wir verwenden für die Zeitschritte ganze Zahlen. Ein typischer Wahrnehmungssatz würde wie folgt aussehen:

$$Percept([Stench, Breeze, Glitter, None, None], 5).$$

Hier ist *Percept* ein binäres Prädikat und *Stench* usw. sind Konstanten, die in einer Liste abgelegt sind. Die Aktionen in der Wumpus-Welt können durch logische Terme repräsentiert werden:

$$Turn(Right), Turn(Left), Forward, Shoot, Grab, Release, Climb.$$

Um zu ermitteln, welche Aktion die beste ist, erstellt das Agentenprogramm die Abfrage

$$\text{AskVars}(\exists a \;\; BestAction(a, 5)),$$

die eine Bindungsliste wie zum Beispiel $\{a / Grab\}$ zurückgibt. Das Agentenprogramm kann dann *Grab* als die zu ergreifende Aktion zurückgeben. Die reinen Wahrnehmungsdaten implizieren bestimmte Fakten über den aktuellen Zustand, zum Beispiel:

$$\forall t, s, g, m, c \;\; Percept([s, Breeze, g, m, c], t) \Rightarrow Breeze(t)$$
$$\forall t, s, b, m, c \;\; Percept([s, b, Glitter, m, c], t) \Rightarrow Glitter(t)$$

usw. Diese Regeln weisen eine triviale Form des Schlussprozesses auf, auch als **Perzeption** bezeichnet, worauf wir in *Kapitel 24* noch genauer eingehen. Beachten Sie die Quantifizierung über die Zeit t. In der Aussagenlogik müssten wir für jeden Zeitschritt Kopien jedes Satzes anlegen.

Durch quantifizierte Implikationssätze kann auch ein einfaches „Reflexverhalten“ implementiert werden. Beispielsweise haben wir:

$$\forall t \;\; Glitter(t) \;\Rightarrow\; BestAction(Grab, t).$$

Mit der Wahrnehmung und den Regeln aus den vorigen Absätzen würde dies zu dem gewünschten Schluss $BestAction(Grab, 5)$ führen – d.h., *Grab* ist die richtige Aktion.

Wir haben die Eingaben und Ausgaben des Agenten dargestellt; jetzt wollen wir die eigentliche Umgebung darstellen. Wir beginnen mit den Objekten. Offensichtliche Kandidaten sind Felder, Falltüren und der Wumpus. Wir könnten jedem Feld einen Namen geben – $Feld_{1,2}$ usw. –, aber dann wäre die Tatsache, dass $Feld_{1,2}$ und $Feld_{1,3}$ benachbart sind, ein zusätzlicher „Fakt“ und wir bräuchten für jedes Feldpaar einen solchen Fakt. Besser ist die Verwendung eines komplexen Terms, in dem Zeile und Spalte als ganze Zahlen erscheinen; wir können z.B. einfach den Listenterm [1,2] verwenden. Die Nachbarschaft von zwei Feldern lässt sich als

$$\forall x, y, a, b \;\; Adjacent([x, y], [a, b]) \Leftrightarrow$$
$$(x = a \wedge (y = b - 1 \vee y = b + 1)) \vee (y = b \wedge (x = a - 1 \vee x = a + 1))$$

definieren. Wir könnten auch jede Falltür mit einem Namen versehen, was aus einem anderen Grund nicht sinnvoll wäre: Es gibt keinen Grund, zwischen den Falltüren zu unterscheiden.[10] Es ist einfacher, ein unäres Prädikat *Pit* zu verwenden, das für Felder wahr ist, die Falltüren enthalten. Weil es nur genau einen Wumpus gibt, ist eine Konstante *Wumpus* so gut wie ein unäres Prädikat (und aus der Perspektive des Wumpus möglicherweise auch würdiger).

Die Position des Agenten ändert sich mit der Zeit; deshalb schreiben wir $At(Agent, s, t)$, was ausdrückt, dass sich der Agent zur Zeit t auf dem Feld s befindet. Die Position des Wumpus können wir mit $\forall t \;\; At(Wumpus, [2, 2], t)$ festlegen. Dann lässt sich sagen, dass Objekte zu einem bestimmten Zeitpunkt nur an einer einzigen Position sein können:

$$\forall x, s_1, s_2, t \; At(x, s_1, t) \wedge At(x, s_2, t) \Rightarrow s_1 = s_2.$$

Aus seiner aktuellen Position kann der Agent Eigenschaften des Feldes von Eigenschaften seiner aktuellen Wahrnehmung ableiten. Befindet sich der Agent beispielsweise auf einem Feld und nimmt einen Luftzug wahr, ist dieses Feld windig (breezy):

$$\forall s, t \;\; At(Agent, s, t) \wedge Breeze(t) \Rightarrow Breezy(s).$$

Es ist sinnvoll zu wissen, ob ein *Feld* windig ist, weil wir wissen, dass sich die Falltüren nicht bewegen können. Beachten Sie, dass es für *Breezy* kein Zeitargument gibt.

Nachdem wir erkannt haben, welche Orte windig sind (oder stinken) und – was sehr wichtig ist – welche *nicht* windig sind (oder *nicht* stinken), kann der Agent ableiten, wo sich die Falltüren befinden (und wo sich der Wumpus aufhält). Während die Aussagenlogik ein separates Axiom für jedes Feld notwendig macht (siehe R_2 und R_3 in *Abschnitt 7.4.3*) und eine unterschiedliche Menge von Axiomen für jedes geografische Layout der Welt benötigen würde, kommt die Logik erster Stufe mit einem Axiom aus:

$$\forall s \;\; Breezy(s) \Leftrightarrow \exists r \;\; Adjacent(r, s) \wedge Pit(r). \tag{8.4}$$

Analog dazu können wir in Logik erster Stufe über der Zeit quantifizieren, sodass wir lediglich ein Nachfolgerzustands-Axiom für jedes Prädikat benötigen und nicht ein anderes Exemplar für jeden Zeitschritt. Zum Beispiel wird das Axiom für den Pfeil in Gleichung (7.2) in *Abschnitt 7.7.1* zu:

$$\forall t \;\; HaveArrow(t + 1) \Leftrightarrow (HaveArrow(t) \wedge \neg Action(Shoot, t)).$$

10 So wie die meisten von uns nicht jedem Vogel einen Namen geben, der im Herbst über unsere Köpfe fliegt, um im Süden zu überwintern. Ein Ornithologe, der Wanderungsmuster, Überlebenszahlen usw. untersuchen will, *gibt* jedem Vogel einen Namen, indem er ihm einen Ring anlegt, weil einzelne Vögel beobachtet werden müssen.

Aus diesen beiden Beispielsätzen lässt sich erkennen, dass die Formulierung in Logik erster Stufe weniger prägnant ist als die ursprüngliche Beschreibung in natürlicher Sprache, die *Kapitel 7* angegeben hat. Der Leser sei eingeladen, analoge Axiome für die Position und Richtung des Agenten zu konstruieren; in diesen Fällen quantifizieren die Axiome sowohl über Raum als auch Zeit. Wie im Fall der aussagenlogischen Zustandsschätzung kann ein Agent logische Inferenz mit Axiomen dieser Art verwenden, um Aspekte der Welt, die nicht direkt beobachtet werden, zu verfolgen. *Kapitel 10* beschäftigt sich eingehender mit dem Thema der Nachfolgerzustands-Axiome erster Stufe und wie sich damit Pläne konstruieren lassen.

8.4 Wissensmodellierung in Logik erster Stufe

Der vorige Abschnitt hat gezeigt, wie man die Logik erster Stufe verwendet, um Wissen in drei einfachen Domänen zu repräsentieren. Dieser Abschnitt beschreibt den allgemeinen Prozess des Aufbaus einer Wissensbasis – auch als **Wissensmodellierung** (**Knowledge Engineering**) bezeichnet. Ein Wissensingenieur ist jemand, der eine bestimmte Domäne erforscht, in Erfahrung bringt, welche Konzepte in dieser Domäne wichtig sind, und eine formale Repräsentation der Objekte und Relationen in der Domäne erzeugt. Wir demonstrieren den Prozess der Wissensmodellierung in einer Domäne elektronischer Schaltkreise, die Ihnen bereits vertraut sein sollte, sodass wir uns auf die dabei erforderlichen Repräsentationsaspekte konzentrieren können. Der Ansatz, den wir hier verfolgen, ist geeignet, um *spezielle* Wissensbasen zu entwickeln, deren Domäne sorgfältig begrenzt ist und deren Abfragebereich im Voraus bekannt ist. *Allgemeine* Wissensbasen, die Abfragen über den gesamten Bereich menschlichen Wissens unterstützen, werden in *Kapitel 12* beschrieben.

8.4.1 Der Prozess der Wissensmodellierung

Projekte der Wissensmodellierung unterscheiden sich stark in Inhalt, Gültigkeitsbereich und Schwierigkeitsgrad, aber alle diese Projekte beinhalten die folgenden Schritte:

1. *Die Aufgabe identifizieren.* Der Wissensingenieur muss den Bereich der Fragen festlegen, die die Wissensbasis unterstützt, ebenso wie die Fakten, die für jede spezifische Probleminstanz zur Verfügung stehen. Muss zum Beispiel die Wumpus-Wissensbasis in der Lage sein, Aktionen auszuwählen, oder soll sie nur Fragen über den Inhalt der Umgebung beantworten? Beinhalten die Sensorfakten die aktuelle Position? Anhand der Aufgabe kann entschieden werden, welches Wissen repräsentiert werden muss, um Probleminstanzen mit Antworten zu verbinden. Dieser Schritt ist analog zum PEAS-Prozess für den Entwurf von Agenten in *Kapitel 2*.

2. *Relevantes Wissen sammeln.* Der Wissensingenieur ist möglicherweise schon Experte in der Domäne oder arbeitet mit realen Experten zusammen, um in Erfahrung zu bringen, was sie wissen – ein Prozess, der auch als **Wissensakquisition** bezeichnet wird. In dieser Phase wird das Wissen nicht formal repräsentiert. Die Idee dabei ist, den Gültigkeitsbereich der Wissensbasis zu verstehen, den die Aufgabe bedingt, und außerdem zu verstehen, wie die Domäne funktioniert.

   Für die Wumpus-Welt, die durch eine künstliche Regelmenge definiert ist, lässt sich das relevante Wissen einfach identifizieren. (Beachten Sie jedoch, dass die Definition der Nachbarschaft in den Regeln der Wumpus-Welt nicht explizit bereitgestellt

wurde.) Für reale Domänen kann der Aspekt der Relevanz relativ schwierig sein – zum Beispiel kann es für ein System zur Simulation von VLSI-Entwürfen erforderlich sein, unter Umständen Kriechströme oder Skin-Effekte zu berücksichtigen.

3 *Ein Vokabular aus Prädikaten, Funktionen und Konstanten festlegen.* Das bedeutet, die wichtigen Konzepte auf Domänenebene müssen in Namen auf Logikebene übersetzt werden. Das beinhaltet viele Fragen zum *Stil* der Wissensmodellierung. Wie der Programmierstil kann auch dieser Stil wichtigen Einfluss auf den Erfolg des Projektes haben. Sollen beispielsweise Falltüren durch Objekte oder durch unäre Prädikate für Felder repräsentiert werden? Soll die Richtung des Agenten eine Funktion oder ein Prädikat sein? Soll die Position des Wumpus von der Zeit abhängen? Nachdem diese Entscheidungen getroffen sind, erhält man ein Vokabular, das auch als **Ontologie** der Domäne bezeichnet wird. Das Wort *Ontologie* steht für eine bestimmte Theorie des Daseins. Die Ontologie bestimmt, welche Arten von Dingen existieren, aber sie legt nicht ihre bestimmten Eigenschaften und Beziehungen untereinander fest.

4 *Allgemeines Wissen über die Domäne kodieren.* Der Wissensingenieur schreibt alle Axiome für alle Vokabularbegriffe auf. Damit wird die Bedeutung der Terme (im Rahmen der Möglichkeiten) festgelegt, sodass der Experte den Inhalt überprüfen kann. Häufig zeigt dieser Schritt fehlerhafte Konzepte oder Lücken im Vokabular auf, die korrigiert werden müssen. Dazu kehrt man zu Schritt 3 zurück und durchläuft den Prozess erneut.

5 *Kodierung einer Beschreibung der spezifischen Probleminstanz.* Wenn die Ontologie ausreichend gut durchdacht ist, ist dieser Schritt einfach. Er beinhaltet das Niederschreiben einfacher atomarer Sätze über Instanzen von Konzepten, die bereits Teil der Ontologie sind. Für einen logischen Agenten werden die Probleminstanzen durch die Sensoren bereitgestellt, während eine „entkörperlichte“ Wissensbasis mit zusätzlichen Sätzen versorgt wird, ähnlich wie die traditionellen Programme mit Eingabedaten versorgt werden.

6 *Abfragen an die Inferenzprozedur richten und Antworten erhalten.* Hier kommt die Belohnung für den Aufwand: Wir können es der Inferenzprozedur überlassen, mit Axiomen und problemspezifischen Fakten zu arbeiten, um die Fakten abzuleiten, an denen wir interessiert sind. Somit vermeiden wir es, einen anwendungsspezifischen Lösungsalgorithmus schreiben zu müssen.

7 *Fehler in der Wissensbasis eliminieren.* Leider sind die Antworten auf Abfragen selten im ersten Anlauf korrekt. Genauer gesagt, die Antworten sind für die Wissensbasis *wie sie geschrieben wurde* korrekt, vorausgesetzt, die Inferenzprozedur ist korrekt, aber das sind häufig nicht die Antworten, die der Benutzer erwartet. Fehlt beispielsweise ein Axiom, können einige Abfragen aus der Wissensbasis nicht beantwortet werden. Das kann einen beachtlichen Debugging-Aufwand verursachen. Fehlende oder zu schwache Axiome können schnell identifiziert werden, indem man Stellen beobachtet, wo die Schlusskette unerwartet unterbrochen wird. Beinhaltet die Wissensbasis beispielsweise eine diagnostische Regel (siehe Übung 8.14), um den Wumpus zu finden

$$\forall s \;\; Smelly(s) \Rightarrow Adjacent(Home(Wumpus), s)$$

anstelle des Bikonditionals, kann der Agent die *Abwesenheit* von Wumpi nie beweisen. Fehlerhafte Axiome können erkannt werden, weil sie falsche Aussagen über die Welt darstellen. So ist beispielsweise der Satz

$$\forall x \; AnzahlDerBeine(x, 4) \Rightarrow Säugetier(x)$$

Tipp

für Reptilien, Amphibien und insbesondere Tische falsch. Dass dieser Satz falsch ist, *kann unabhängig von der restlichen Wissensbasis festgestellt werden*. Im Gegensatz dazu sieht ein typischer Fehler in einem Programm wie folgt aus:

```
offset = position + 1
```

Es kann nicht entschieden werden, ob diese Aussage korrekt ist, ohne nicht das restliche Programm daraufhin zu betrachten, ob `offset` verwendet wird, um auf die aktuelle Position zu verweisen oder auf die Position nach der aktuellen Position, oder ob der Wert von `position` durch eine andere Anweisung geändert wird und damit auch `offset` wieder geändert werden sollte.

Um diesen siebenstufigen Prozess besser zu verstehen, wenden wir ihn jetzt auf ein größeres Beispiel an – die Domäne der elektronischen Schaltkreise.

8.4.2 Die Domäne der elektronischen Schaltkreise

In diesem Abschnitt werden wir eine Ontologie und Wissensbasis entwickeln, die es uns erlaubt, über digitale Schaltungen wie in ▶ Abbildung 8.6 gezeigt zu schließen. Wir folgen dem siebenstufigen Prozess für die Wissensmodellierung.

Die Aufgabe identifizieren

Es gibt viele Aufgaben, die mit dem Schließen im Hinblick auf digitale Schaltkreise zu tun haben. Auf der höchsten Ebene wird dabei die Funktionalität des Schaltkreises analysiert. Addiert beispielsweise der in Abbildung 8.6 gezeigte Schaltkreis korrekt? Wenn alle Eingänge auf High (Hoch-Pegel) liegen, welche Ausgabe erzeugt dann Gatter A2? Auch Fragen zum Aufbau des Schaltkreises sind interessant. Beispielsweise, welche Gatter sind an den ersten Eingang angeschlossen? Enthält der Schaltkreis Rückkopplungsschleifen? Das ist unsere Aufgabe in diesem Abschnitt. Es gibt detailliertere Analyseebenen, wie etwa im Hinblick auf Timing-Verzögerungen, Schaltkreisfläche, Stromverbrauch, Produktionskosten usw. Für jede dieser Ebenen braucht man zusätzliches Wissen.

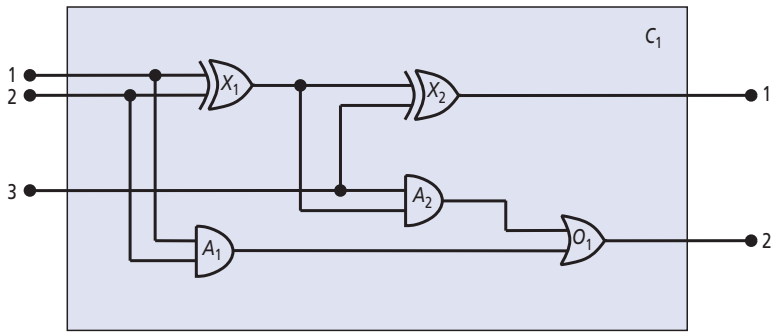


Abbildung 8.6: Ein digitaler Schaltkreis C1, der vorgibt, ein Ein-Bit-Volladdierer zu sein. Die beiden ersten Eingänge sind die beiden zu addierenden Bits, der dritte Eingang ist ein Übertrags-Bit. Der erste Ausgang ist die Summe, der zweite Ausgang das Übertrags-Bit für den nächsten Addierer. Der Schaltkreis enthält zwei XOR-Gatter, zwei AND-Gatter und ein OR-Gatter.

Relevantes Wissen sammeln

Was wissen wir über digitale Schaltkreise? Für unsere Zwecke bestehen sie aus Leitungen und Gattern. Das Signal verläuft entlang der Leitungen zu den Eingangsanschlüssen der Gatter und jedes Gatter erzeugt ein Signal auf dem Ausgang, wo es auf einer weiteren Leitung weitergeführt wird. Um festzustellen, welche Signale das sind, müssen wir wissen, wie die Gatter ihre Eingangssignale verarbeiten. Es gibt vier Arten von Gattern: AND-, OR- und XOR-Gatter haben zwei Eingänge, NOT-Gatter haben einen Eingang. Alle Gatter haben einen Ausgang. Schaltungen haben ebenso wie Gatter Ein- und Ausgänge.

Um Schlüsse zur Funktionalität und Konnektivität ziehen zu können, müssen wir nicht über die Leitungen als solche sprechen und auch nicht darüber, wie die Leitungen verlegt oder wie zwei Leitungen verbunden sind. Hier interessiert uns nur die Verbindung zwischen den Anschlüssen – wir können sagen, dass ein Ausgangsanschluss mit einem anderen Eingangsanschluss verbunden ist, ohne eine Aussage darüber treffen zu müssen, was diese Anschlüsse überhaupt verbindet. Es gibt viele andere Faktoren der Domäne, die für unsere Analyse irrelevant sind, wie beispielsweise die Größe, die Form, die Farbe oder die Kosten der verschiedenen Komponenten.

Wenn wir irgendetwas anderes analysieren müssten als den Entwurf auf Gatterebene, hätten wir eine andere Ontologie. Wären wir beispielsweise am Debugging fehlerhafter Schaltkreise interessiert, so wäre es sinnvoll, die Leitungen in die Ontologie aufzunehmen, weil eine defekte Leitung den darauf verlaufenden Signalfluss stören kann. Wären wir am Entwurf eines Produktes interessiert, das möglichst viel Gewinn bringt, so wären die Kosten für den Schaltkreis und seine Geschwindigkeit im Vergleich zu anderen Produkten auf dem Markt von Bedeutung.

Ein Vokabular festlegen

Wir wissen, dass wir über Schaltkreise, Anschlüsse, Signale und Gatter sprechen wollen. Im nächsten Schritt wählen wir Funktionen, Prädikate und Konstanten für ihre Darstellung. Als Erstes müssen wir ein Gatter von anderen Gattern und von anderen Objekten unterscheiden können. Jedes Gatter wird als Objektname durch eine Konstante dargestellt, über die wir zusichern können, dass es sich beispielsweise um $Gate(X_1)$ handelt. Das Verhalten jedes Gatters wird durch seinen Typ bestimmt: eine der Konstanten *AND*, *OR*, *XOR* und *NOT*. Da ein Gatter genau einen Typ hat, bietet sich eine Funktion an: $Type(X_1) = XOR$. Schaltkreise werden genauso wie Gatter durch ein Prädikat gekennzeichnet: $Circuit(C_1)$.

Als Nächstes betrachten wir Anschlüsse, die durch das Prädikat $Terminal(x)$ gekennzeichnet werden. Ein Gatter oder eine Schaltung kann einen oder mehrere Eingangsanschlüsse (kurz: Eingänge) und einen oder mehrere Ausgangsanschlüsse (kurz: Ausgänge) haben. Wir verwenden die *Funktion* $In(1, X_1)$, um den ersten Eingang für Gatter X_1 zu bezeichnen. Eine ähnliche Funktion *Out* wird für Ausgänge verwendet. Die Funktion $Arity(c, i, j)$ besagt, dass Schaltkreis c über i Eingänge und j Ausgänge verfügt. Die Verbindung zwischen Gattern kann durch das Prädikat *Connected* dargestellt werden, das zwei Ein- oder Ausgänge als Argumente übernimmt, wie in $Connected(Out(1, X_1), In(1, X_2))$.

Schließlich müssen wir noch wissen, ob ein Signal aktiviert („Ein“) oder deaktiviert („Aus“) ist. Eine Möglichkeit wäre die Verwendung eines unären Prädikats, nämlich *On*, das wahr ist, wenn das Signal an einem Ein- oder Ausgang aktiv ist. Das macht es jedoch schwer, Fragen wie etwa: „Welche möglichen Werte haben die Signale an den

Ausgängen von Schaltung C_1?“ zu stellen. Deshalb werden wir zwei „Signalwerte“ als Objekte einführen, 1 und 0, sowie eine Funktion *Signal*(*t*), die den Signalwert für den Anschluss *t* bezeichnet.

Allgemeines Wissen über die Domäne kodieren

Ein Anzeichen für eine gute Ontologie ist die Tatsache, dass nur wenige allgemeine Regeln spezifiziert werden müssen, die sich deutlich und prägnant formulieren lassen. Wir benötigen lediglich folgende Axiome:

1 Wenn zwei Anschlüsse verbunden sind, haben sie dasselbe Signal:

$$\forall t_1, t_2 \quad Terminal(t_1) \wedge Terminal(t_2) \wedge Connected(t_1, t_2) \Rightarrow Signal(t_1) = Signal(t_2).$$

2 Das Signal an jedem Anschluss ist entweder 1 oder 0:

$$\forall t \quad Terminal(t) = \Rightarrow Signal(t) = 1 \vee Signal(t) = 0.$$

3 *Connected* ist ein kommutatives Prädikat:

$$\forall t_1, t_2 \quad Connected(t_1, t_2) \Leftrightarrow Connected(t_2, t_1).$$

4 Es gibt vier Arten von Gattern:

$$\forall g \; Gate(g) \wedge k = Type(g) \Rightarrow k = AND \vee k = OR \vee k = XOR \vee k = NOT.$$

5 Der Ausgang eines AND-Gatters ist genau dann 0, wenn einer seiner Eingänge 0 ist:

$$\forall g \; Gate(g) \wedge \text{Type(g)} = \text{AND} \Rightarrow Signal(Out(1, g)) = 0 \Leftrightarrow \exists n \; Signal(In(n, g)) = 0.$$

6 Der Ausgang eines OR-Gatters ist genau dann 1, wenn einer seiner Eingänge 1 ist:

$$\forall g \; Gate(g) \wedge Type(g) = OR \Rightarrow Signal(Out(1, g)) = 1 \Leftrightarrow \exists n \; Signal\,(In(n, g)) = 1.$$

7 Der Ausgang eines XOR-Gatters ist genau dann 1, wenn seine Eingänge unterschiedlich sind:

$$\forall g \; Gate(g) \wedge Type(g) = XOR \Rightarrow Signal(Out(1, g)) = 1 \Leftrightarrow Signal\,(In(1, g)) = Signal\,(In(2, g)).$$

8 Der Ausgang eines NOT-Gatters unterscheidet sich von seinem Eingang:

$$\forall g \; Gate(g) \wedge (Type(g) = NOT\,) \Rightarrow Signal(Out(1, g)) = Signal(In(1, g)).$$

9 Die Gatter haben (mit Ausnahme von NOT) zwei Eingänge und einen Ausgang.

$$\forall g \; Gate(g) \wedge Type(g) = NOT \Rightarrow Arity(g, 1, 1)$$
$$\forall g \; Gate(g) \wedge k = Type(g) \wedge (k = AND \vee k = OR \vee k = XOR) \Rightarrow Arity(g, 2, 1).$$

10 Ein Schaltkreis besitzt Anschlüsse bis zu seiner Eingangs- und Ausgangsstelligkeit und nichts über seine Stelligkeit hinaus:

$$\forall c, i, j \; Circuit(c) \wedge Arity(c, i, j) \Rightarrow$$
$$\forall n \; (n \leq i \Rightarrow Terminal(In(c, n))) \wedge (n > i \Rightarrow In(c, n) = Nothing) \wedge$$
$$\forall n \; (n \leq j \Rightarrow Terminal(Out(c, n))) \wedge (n > j \Rightarrow Out(c, n) = Nothing).$$

11 Gatter, Anschlüsse, Signale, Gattertypen und *Nothing* sind voneinander verschieden.

$$\forall g, t \;\; Gate(g) \wedge Terminal(t) \Rightarrow g \neq t \neq 1 \neq 0 \neq OR \neq AND \neq XOR \neq NOT \neq Nothing$$

12 Gatter sind Schaltkreise:

$$\forall g \;\; Gate(g) \Rightarrow Circuit(g).$$

Die spezifische Probleminstanz kodieren

Der in Abbildung 8.6 gezeigte Schaltkreis ist als C_1 mit der folgenden Beschreibung kodiert. Zuerst kategorisieren wir die Gatter:

$$Circuit(C_1) \wedge Arity(C_1, 3, 2)$$
$$Gate(X_1) \wedge Type(X_1) = XOR$$
$$Gate(X_2) \wedge Type(X_2) = XOR$$
$$Gate(A_1) \wedge Type(A_1) = AND$$
$$Gate(A_2) \wedge Type(A_2) = AND$$
$$Gate(O_1) \wedge Type(O_1) = OR.$$

Dann zeigen wir die Verknüpfung zwischen ihnen:

$$Connected(Out(1, X_1), In(1, X_2)) \qquad Connected(In(1, C_1), In(1, X_1))$$
$$Connected(Out(1, X_1), In(2, A_2)) \qquad Connected(In(1, C_1), In(1, A_1))$$
$$Connected(Out(1, A_2), In(1, O_1)) \qquad Connected(In(2, C_1), In(2, X_1))$$
$$Connected(Out(1, A_1), In(2, O_2)) \qquad Connected(In(2, C_1), In(2, A_1))$$
$$Connected(Out(1, X_2), Out(1, C_1)) \qquad Connected(In(3, C_1), In(2, X_2))$$
$$Connected(Out(1, O_1), Out(2, C_1)) \qquad Connected(In(3, C_1), In(1, A_2)).$$

Abfragen an die Inferenzprozedur richten

Welche Eingabekombinationen bewirken, dass der erste Ausgang von C_1 (das Summen-Bit) gleich 0 und der zweite Ausgang von C_1 (das Übertrags-Bit) gleich 1 sind?

$$\exists i_1, i_2, i_3 \;\; Signal(In(1, C_1)) = i_1 \;\wedge\; Signal(In(2, C_1)) = i_2 \;\wedge\; Signal(In(3, C_1)) = i_3 \;\wedge\; Signal(Out(1, C_1)) = 0 \;\wedge\; Signal(Out(2, C_1)) = 1$$

Die Antworten sind Substitutionen für die Variablen i_1, i_2 und i_3, sodass der resultierende Satz durch die Wissensbasis folgerbar ist. ASKVARS liefert drei derartige Substitutionen:

$$\{i_1/1, i_2/1, i_3/0\} \;\; \{i_1/1, i_2/0, i_3/1\} \;\; \{i_1/0, i_2/1, i_3/1\}.$$

Welche möglichen Wertemengen gibt es für alle Anschlüsse des Addiererschaltkreises?

$$\exists i_1, i_2, i_3, o_1, o_2 \;\; Signal(In(1, C_1)) = i_1 \;\wedge\; Signal(In(2, C_1)) = i_2 \;\wedge\; Signal(In(3, C_1)) = i_3 \;\wedge\; Signal(Out(1, C_1)) = o_1 \;\wedge\; Signal(Out(2, C_1)) = o_2.$$

Diese letzte Abfrage gibt eine vollständige Eingangs-Ausgangs-Tabelle für das Bauelement zurück, anhand derer überprüft werden kann, dass es die Eingaben tatsächlich korrekt addiert. Dies ist ein einfaches Beispiel für eine **Schaltungsüberprüfung**. Wir können die Definition für den Schaltkreis auch nutzen, um größere digitale Systeme zu erstellen, für die die gleiche Überprüfung durchgeführt werden kann (siehe Übung 8.17). Viele Domänen weisen eine ähnlich strukturierte Entwicklung ihrer Wissensbasen auf, wobei komplexere Konzepte auf einfacheren Konzepten aufbauend definiert werden.

Debugging der Wissensbasis

Wir können die Wissensbasis auf verschiedene Arten stören, um zu beobachten, welche fehlerhaften Verhaltensweisen daraus entstehen. Angenommen, Sie lesen *Abschnitt 8.2.8* nicht und vergessen folglich die Zusicherung $1 \neq 0$. Plötzlich kann das System keine Ausgaben für die Schaltung mehr beweisen, außer für die Eingaben 000 und 110. Wir können das Problem eingrenzen, indem wir die Ausgaben jedes Gatters abfragen. Beispielsweise könnten wir fragen:

$$\exists i_1, i_2, o \;\; Signal(In(1, C_1)) = i_1 \;\wedge\; Signal(In(2, C_1)) = i_2 \;\wedge\; Signal(Out(1, X_1))$$

wodurch deutlich wird, dass an X_1 für die Eingangsbelegungen 10 und 01 keine Ausgänge bekannt sind. Wir betrachten dann das Axiom für XOR-Gatter, das auf X_1 angewendet wird:

$$Signal(Out(1, X_1)) = 1 \;\Leftrightarrow\; Signal(In(1, X_1)) \neq Signal(In(2, X_1)).$$

Wenn die Eingänge beispielsweise 1 und 0 sind, wird dies reduziert zu:

$$Signal(Out(1, X_1)) = 1 \;\Leftrightarrow\; 1 \neq 0.$$

Jetzt ist das Problem offensichtlich: Das System kann nicht ableiten, dass $Signal(Out(1, X_1)) = 1$ ist, deshalb müssen wir ihm mitteilen, dass $1 \neq 0$ gilt.

Bibliografische und historische Hinweise

Obwohl schon die Logik von Aristoteles Verallgemeinerungen über Objekte berücksichtigt, ist sie weit von der Ausdrucksstärke der Logik erster Stufe entfernt. Eine erhebliche Hürde für ihre weitere Entwicklung war ihre Konzentration auf einstellige Prädikate zum Ausschluss von mehrstelligen relationalen Prädikaten. Die erste systematische Abhandlung kommt von Augustus De Morgan (1864), der anhand des folgenden Beispiels die Arten der Inferenzen zeigt, die mit der Logik von Aristoteles nicht zu verarbeiten waren: „Alle Pferde sind Tiere; demzufolge ist der Kopf eines Pferds der Kopf eines Tiers.“ Diese Inferenz ist Aristoteles nicht zugänglich, weil jede Regel, die diese Inferenz unterstützt, zuerst den Satz mithilfe des zweistelligen Prädikats „x ist der Kopf von y“ analysieren muss. Die Logik der Relationen wurde eingehend von Charles Sanders Peirce (1870) untersucht.

Echte Logik erster Stufe geht auf die Einführung von Quantoren in der *Begriffsschrift* von Gottlob Frege (1879) zurück. Peirce (1883) hat ebenfalls und unabhängig von Frege die Logik erster Stufe entwickelt, wenn auch etwas später. Die Fähigkeit von Frege, Quantoren zu verschachteln, war ein großer Schritt vorwärts, doch hat er eine umständliche Notation verwendet. Die heute gebräuchliche Notation für die Logik erster Stufe wurde hauptsächlich von Giuseppe Peano (1889) entwickelt, aber die Semantik ist fast identisch mit der von Frege. Seltsamerweise gehen die Axiome von Peano größtenteils auf Grassmann (1861) und Dedekind (1888) zurück.

Leopold Löwenheim (1915) stellte 1915 eine systematische Behandlung der Modelltheorie für die Logik erster Stufe vor. Diese Arbeit beschäftigte sich auch mit dem Gleichheitssymbol als integralem Bestandteil der Logik. Die Ergebnisse von Löwenheim wurden von Thoralf Skolem (1920) erweitert. Alfred Tarski (1935, 1956) formulierte eine explizite Definition der Wahrheit und modelltheoretischen Erfüllung in der Logik erster Stufe unter Verwendung der Mengentheorie.

McCarthy (1958) war hauptverantwortlich für die Einführung der Logik erster Stufe als Werkzeug für den Aufbau von KI-Systemen. Die Aussichten für die logikbasierte KI wurden durch die Entwicklung der Auflösung durch Robinson (1965) wesentlich vorwärts getrieben. Eine vollständige Prozedur für die Inferenz bei der Logik erster Stufe ist in *Kapitel 9* beschrieben. Der logizistische Ansatz wurzelte in Stanford. Cordell Green (1969a, 1969b) entwickelte ein First-Order-Schlusssystem, QA3, das zu ersten Versuchen führte, einen logischen Roboter in SRI zu entwickeln (Fikes und Nilsson, 1971). Die Logik erster Stufe wurde von Zohar Manna und Richard Waldinger (1971) für Schlüsse über Programme und später von Michael Genesereth (1984) für Schlüsse über Schaltkreise angewendet. In Europa wurde die Logikprogrammierung (eine beschränkte Form des Schließens mithilfe von Logik erster Stufe) für die linguistische Analyse entwickelt (Colmerauer *et al.*, 1973), ebenso wie für allgemeine deklarative Systeme (Kowalski, 1974). Die Programmierlogik wurde in Edinburgh durch das LCF-Projekt (Logic for Computable Functions) (Gordon *et al.*, 1979) weitergeführt. Diese Entwicklungen sind in den *Kapiteln 9* und *12* genauer beschrieben.

Zu den praktischen Anwendungen, die mit Logik erster Stufe erstellt wurden, gehören ein System, mit dem sich die Fertigungsanforderungen für elektronische Produkte (Mannion, 2002) auswerten lassen, ein System zum Schließen über Richtlinien für den Dateizugriff und digitale Rechteverwaltung (Halpern und Weissman, 2008) und ein System für die automatisierte Zusammenstellung von Webdiensten (McIlraith und Zeng, 2001).

Reaktionen auf die Whorf-Hypothese (Whorf, 1956) und das Problem von Sprache und Denken im Allgemeinen sind in mehreren Büchern zu finden (Gumperz und Levinson, 1996; Bowerman und Levinson, 2001; Pinker, 2003; Gentner und Goldin-Meadow, 2003). Die „Theorie"-Theorie (Gopnik und Glymour, 2002; Tenenbaum *et al.*, 2007) betrachtet das kindliche Lernen über die Welt als Analogon zur Konstruktion von wissenschaftlichen Theorien. Genau wie Voraussagen bei einem maschinellen Lernalgorithmus stark vom realisierten Vokabular abhängen, hängt die kindliche Formulierung von Theorien von der linguistischen Umgebung ab, in der das Lernen stattfindet.

Es gibt eine Reihe guter Einführungen zur Logik erster Stufe, unter anderem auch von führenden Persönlichkeiten in der Geschichte der Logik: Alfred Tarski (1941), Alonzo Church (1956) und W.V. Quine (1982) (der am lesenswertesten ist). Enderton (1972) bietet eine mehr mathematisch orientierte Sichtweise. Eine hochgradig formale Behandlung der Logik erster Stufe ist zusammen mit vielen erweiterten Logikthemen bei Bell und Machover (1977) zu finden. Manna und Waldinger (1985) geben eine verständliche Einführung in die Logik aus der Perspektive der Informatik, ebenso wie Huth und Ryan (2004), die sich auf Programmverifikation konzentrieren. Barwise und Etchemendy (2002) verfolgen einen Ansatz ähnlich dem hier verwendeten. Smullyan (1995) legt prägnante Ergebnisse in Tabellenform vor. Gallier (1986) bietet eine äußerst strenge mathematische Darstellung der Logik erster Stufe zusammen mit umfangreichen Informationen zu ihrer Verwendung im automatisierten Schließen. Das Buch *Logical Foundations of Artificial Intelligence* (Genesereth und Nilsson, 1987) ist sowohl eine fundierte Einführung in die Logik als auch die erste systematische Abhandlung logischer Agenten mit Wahrnehmungen und Aktionen. Zudem gibt es zwei gute Handbücher: van Bentham und ter Meulen (1997) sowie Robinson und Voronkov (2001). Die führende Fachzeitschrift für das Gebiet der reinen mathematischen Logik ist das *Journal of Symbolic Logic*, während sich das *Journal of Applied Logic* mit Themen befasst, die enger mit künstlicher Intelligenz zu tun haben.

Zusammenfassung

Dieses Kapitel hat die **Logik erster Stufe** (**First-Order-Logik**) vorgestellt, eine Repräsentationssprache, die sehr viel leistungsfähiger als die Aussagenlogik ist. Wichtig sind insbesondere die folgenden Aspekte:

- Wissensrepräsentationssprachen sollten deklarativ, kompositional, ausdrucksstark, kontextunabhängig und eindeutig sein.
- Logiken unterscheiden sich in ihren **ontologischen** und **epistemologischen Bindungen**. Während die Aussagenlogik nur auf der Existenz von Fakten aufbaut, berücksichtigt die Logik erster Stufe die Existenz von Objekten und Relationen und gewinnt damit an Ausdruckskraft.
- Die Syntax der Logik erster Stufe baut auf der Syntax der Aussagenlogik auf. Sie fügt Terme hinzu, um Objekte darzustellen, und verfügt über universelle und existentielle Quantoren, um Zusicherungen über alle oder einige der möglichen Werte der quantifizierten Variablen zu konstruieren.
- Eine **mögliche Welt** oder ein **Modell** für die Logik erster Stufe umfasst eine Menge von Objekten und eine Interpretation, die Konstantensymbole zu Objekten, Prädikatssymbole zu Beziehungen zwischen Objekten und Funktionssymbole zu Funktionen auf Objekten zuordnet.
- Ein atomarer Satz ist genau dann wahr, wenn die durch das Prädikat benannte Relation zwischen den durch die Terme benannten Objekten gilt. **Erweiterte Interpretationen**, die Quantifizierungsvariablen zu Objekten im Modell zuordnen, definieren die Wahrheit von quantifizierten Sätzen.
- Die Entwicklung einer Wissensbasis in Logik erster Stufe bedingt eine sorgfältige Analyse der Domäne, die Auswahl eines Vokabulars und die Kodierung von Axiomen für die Unterstützung der gewünschten Inferenzen.

Übungen zu Kapitel 8

Lösungshinweise

1 Eine logische Wissensbasis repräsentiert die Welt unter Verwendung einer Menge von Sätzen ohne explizite Struktur. Eine **analoge** Repräsentation dagegen hat eine physische Struktur, die direkt der Struktur des dargestellten Objektes entspricht. Betrachten Sie eine Straßenkarte Ihres Landes als analoge Repräsentation von Fakten über das Land. Die zweidimensionale Struktur der Karte entspricht der zweidimensionalen Oberfläche des Gebietes.

a. Nennen Sie fünf Beispiele für *Symbole* in der Kartensprache.

b. Ein *expliziter* Satz ist ein Satz, den der Ersteller der Repräsentation aufschreibt. Ein *impliziter* Satz ist ein Satz, der aufgrund der Eigenschaften der analogen Repräsentation aus expliziten Sätzen folgt. Nennen Sie drei Beispiele für *implizite* und *explizite* Sätze in der Kartensprache.

c. Nennen Sie drei Beispiele für Fakten zur physischen Struktur Ihres Landes, die nicht in der Kartensprache repräsentiert werden können.

d. Nennen Sie zwei Beispiele für Fakten, die in der Kartensprache einfacher auszudrücken sind als in der Logik erster Stufe.

e. Nennen Sie zwei weitere Beispiele für sinnvolle analoge Repräsentationen. Welche Vor- und Nachteile haben diese Sprachen?

2 Betrachten Sie eine Wissensbasis, die nur zwei Sätze enthält: *P(a)* und *P(b)*. Ist in dieser Wissensbasis $\forall x\ P(x)$ folgerbar? Erklären Sie Ihre Antwort anhand von Modellen.

3 Ist der Satz $\exists x, y\ \ x = y$ gültig? Erläutern Sie Ihre Antwort.

4 Schreiben Sie einen logischen Satz auf, sodass jede Welt, in der er wahr ist, genau ein Objekt enthält.

5 Stellen Sie sich ein Symbolvokabular vor, das c Konstantensymbole, p_k Prädikatssymbole für jede k-Stelligkeit und f_k Funktionssymbole für jede k-Stelligkeit enthält, mit $1 \leq k \leq A$. Die Domäne hat die fixe Größe D. Für jede beliebige Interpretation/Modell-Kombination wird jedes Prädikats- oder Funktionssymbol auf eine Relation bzw. Funktion derselben Stelligkeit abgebildet. Sie können voraussetzen, dass die Funktionen in dem Modell Eingabe-Tupel erlauben, die keinen Wert für die Funktion enthalten (d.h., der Wert ist das unsichtbare Objekt). Leiten Sie eine Formel für die Anzahl möglicher Interpretation/Modell-Kombinationen für eine Domäne mit D Elementen ab. Machen Sie sich dabei keine Gedanken über die Eliminierung redundanter Kombinationen.

6 Welche der folgenden Sätze sind gültig (zwangsläufig wahr)?

a. $(\exists x\ x = x) \Rightarrow (\forall y\ \exists z\ y = z)$

b. $\forall x\ \ P(x) \vee \neg P(x)$

c. $\forall x\ \ Smart\,(x) \vee (x = x)$

7 Betrachten Sie eine Version der Semantik für Logik erster Stufe, in der Modelle mit leeren Domänen erlaubt sind. Geben Sie mindestens zwei Beispiele für Sätze an, die entsprechend der Standardsemantik gültig sind, nicht jedoch entsprechend der neuen Semantik. Erörtern Sie, welche Ergebnisse für Ihre Beispiele intuitiv sinnvoller erscheinen.

8 Folgt der Fakt $\neg Gatte(George, Laura)$ aus den Fakten $Jim \neq George$ und $Gatte(Jim, Laura)$? Geben Sie einen Beweis an, falls dies der Fall sein sollte. Stellen Sie andernfalls zusätzliche Axiome je nach Bedarf bereit.

9 Betrachten Sie ein Vokabular mit den folgenden Symbolen:

$Beruf(p, b)$: Prädikat. Person p hat den Beruf b.

$Kunde(p1, p2)$: Prädikat. Person $p1$ ist ein Kunde von Person $p2$.

$Chef(p1, p2)$: Prädikat. Person $p1$ ist Chef von Person $p2$.

$Doktor, Chirurg, Anwalt, Schauspieler$: Konstanten, die Berufe bezeichnen.

$Emily, Joe$: Konstanten, die Personen bezeichnen.

Verwenden Sie diese Symbole, um die folgenden Zusicherungen in Logik erster Stufe zu formulieren:

a. Emily ist weder Chirurg noch Anwalt.

b. Joe ist ein Schauspieler, übt aber auch noch einen anderen Beruf aus.

c. Alle Chirurgen sind Doktoren.

d. Joe hat keinen Anwalt (d.h., er ist kein Kunde irgendeines Anwalts).

e. Emily hat einen Chef, der Anwalt ist.

f. Es gibt einen Anwalt, dessen Kunden alle Doktoren sind.

g. Jeder Chirurg hat einen Anwalt.

10 In jedem der folgenden Sätze geben wir einen Satz in natürlicher Sprache und eine Zahl der infrage kommenden logischen Ausdrücke an. Erklären Sie für jeden der logischen Ausdrücke, ob er (1) den natürlichsprachigen Satz korrekt ausdrückt, (2) syntaktisch ungültig und demzufolge bedeutungslos ist oder (3) syntaktisch gültig ist, aber die Bedeutung des natürlichsprachigen Satzes nicht ausdrückt.

a. Jede Katze liebt ihre Mutter oder ihren Vater.

(i) $\forall\, x\; Katze(x) \Rightarrow Liebt(x, Mutter\,(x) \vee Vater\,(x))$.

(ii) $\forall\, x \leftarrow Katze(x) \vee Liebt(x, Mutter\,(x)) \vee Liebt(x, Vater\,(x))$.

(iii) $\forall\, x\; Katze(x) \wedge (Liebt(x, Mutter(x)) \vee Liebt(x, Vater(x)))$.

b. Jeder Hund, der einen seiner Brüder liebt, ist glücklich.

(i) $\forall x\; Hund(x) \wedge (\exists y\; Bruder(y, x) \wedge Liebt(x, y)) \Rightarrow Glücklich(x)$.

(ii) $\forall x, y\; Hund(x) \wedge Bruder(y, x) \wedge Liebt(x, y) \Rightarrow Glücklich(x)$.

(iii) $\forall x\; Hund(x) \wedge [\forall\, y\; Bruder(y, x) \Leftrightarrow Liebt(x, y)] \Rightarrow Glücklich(x)$.

c. Kein Hund beißt ein Kind seines Besitzers.

(i) $\forall x\; Hund(x) \Rightarrow \neg Beißt(x, Kind(Besitzer(x)))$.

(ii) $\neg\exists x, y\; Hund(x) \wedge Kind(y, Besitzer(x)) \wedge Beißt(x, y)$.

(iii) $\forall\, x\; Hund(x) \Rightarrow (\forall y\; Kind(y, Besitzer(x)) \Rightarrow \neg Beißt(x, y))$.

(iv) $\neg\exists x\; Hund(x) \Rightarrow (\exists y\; Kind(y, Besitzer(x)) \wedge Beißt(x, y))$.

d. Die Postleitzahlen innerhalb eines Landes beginnen für alle mit der gleichen Ziffer.

(i) $\forall x, s, z1\ [Land(s) \wedge WohntIn(x, s) \wedge Zip(x) = z1\ ] \Rightarrow$
$[\forall y, z2\ WohntIn(y, s) \wedge Zip(y) = z2 \Rightarrow Ziffer(1, z1) = Ziffer(1, z2)]$.

(ii) $\forall x, s\ [Land(s) \wedge WohntIn(x, s) \wedge \exists z1\ Zip(x) = z1\ ] \Rightarrow$
$[\forall y, z2\ WohntIn(y, s) \wedge Zip(y) = z2 \wedge Ziffer(1, z1\ ) = Ziffer(1, z2)]$.

(iii) $\forall x, y, s\ Land(s) \wedge WohntIn(x, s) \wedge WohntIn(y, s) \Rightarrow$
$Ziffer(1, Zip(x) = Zip(y))$.

(iv) $\forall x, y, s\ Land(s) \wedge WohntIn(x, s) \wedge WohntIn(y, s) \Rightarrow$
$Ziffer(1, Zip(x)) = Ziffer(1, Zip(y))$.

11 Vervollständigen Sie die folgenden Übungen über logische Sätze:

a. Übersetzen Sie in gutes, natürliches Deutsch (keine x oder y!):
$\forall x, y, l\ SprichtSprache(x, l) \wedge SprichtSprache(y, l)$
$\Rightarrow Versteht(x, y) \wedge Versteht(y, x)$.

b. Erläutern Sie, warum dieser Satz durch den angegebenen Satz folgerbar ist:
$\forall x, y, l\ SprichtSprache\ (x, l\ ) \wedge SprichtSprache\ (y, l\ )$
$\Rightarrow Versteht\ (x, y)$.

c. Übersetzen Sie die folgenden Sätze in Logik erster Stufe:

(i) Verstehen führt zu Freundschaft.

(ii) Freundschaft ist transitiv.

Denken Sie daran, alle verwendeten Prädikate, Funktionen und Konstanten zu definieren.

12 Wahr oder falsch? Erläutern Sie Ihre Antworten.

a. $\exists x\ x = Rumpelstilzchen$ ist ein gültiger (zwangsläufig wahrer) Satz in Logik erster Stufe.

b. Jeder existentiell quantifizierte Satz in Logik erster Stufe ist in jedem Modell wahr, das genau ein Objekt enthält.

c. $\forall x, y\ x = y$ ist erfüllbar.

13 Schreiben Sie die ersten beiden Peano-Axiome in *Abschnitt 8.3.3* als einzelnes Axiom neu, das $NatNum(x)$ definiert, um mögliche natürliche Zahlen auszuschließen, außer denen, die durch die Nachfolgerfunktion generiert werden.

14 Gleichung (8.4) in *Abschnitt 8.3.4* definiert die Bedingungen, unter denen ein Feld windig ist. Hier betrachten wir zwei andere Arten, um diesen Aspekt der Wumpus-Welt zu beschreiben:

a. Wir können **Diagnoseregeln** schreiben, die von beobachteten Effekten zu verborgenen Ursachen führen. Um Falltüren zu finden, muss entsprechend der offensichtlichen Diagnoseregeln ein benachbartes Feld eine Falltür enthalten, wenn ein Feld windig ist; und wenn ein Feld nicht windig ist, enthält kein angrenzendes Feld eine Falltür. Schreiben Sie diese beiden Regeln in Logik erster Ordnung und zeigen Sie, dass ihre Konjunktion logisch mit Gleichung (8.4) äquivalent ist.

b. Wir können **kausale Regeln** schreiben, die von der Ursache zur Wirkung führen. Eine offensichtliche kausale Regel ist, dass eine Falltür bewirkt, dass alle benachbarten Felder windig sind. Schreiben Sie diese Regel in Logik erster Ordnung, erläutern Sie, warum sie im Vergleich zu Gleichung (8.4) unvollständig ist, und geben Sie das fehlende Axiom an.

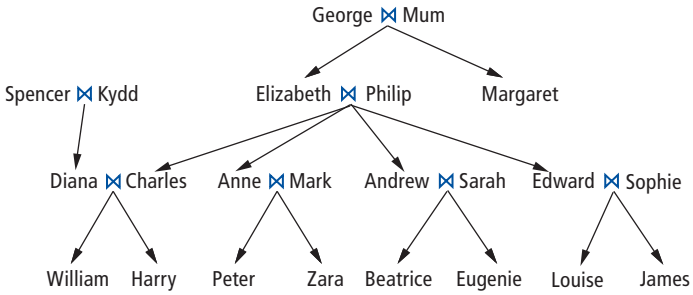


Abbildung 8.7: Ein typischer Stammbaum. Das Symbol ⋈ verbindet Ehegatten und Pfeile verweisen auf Kinder.

15 Schreiben Sie Axiome, die die Prädikate *Enkel*, *Urgroßeltern*, *Vorfahr*, *Bruder*, *Schwester*, *Tochter*, *Sohn*, *CousinErstenGrades*, *Schwager*, *Schwägerin*, *Tante* und *Onkel* beschreiben. Finden Sie die richtige Definition für die m-te Cousine mit n-facher Entfernung und schreiben Sie die Definition in Logik erster Stufe. Schreiben Sie jetzt die grundlegenden Fakten auf, die im Stammbaum in ▶ Abbildung 8.7 gezeigt sind. Verwenden Sie ein geeignetes logisches Schlusssystem, teilen Sie ihm mit TELL alle Sätze mit, die Sie aufgeschrieben haben, und fragen Sie mit ASK, wer die Enkelkinder von Elizabeth, die Schwager von Diana und die Großeltern von Zara sind.

16 Schreiben Sie einen Satz, der zusichert, dass $+$ eine kommutative Funktion ist. Folgt Ihr Satz aus den Peano-Axiomen? Wenn ja, erläutern Sie warum. Geben Sie andernfalls ein Modell an, in dem die Axiome wahr sind und Ihr Satz falsch ist.

17 Verwenden Sie die Mengenaxiome als Beispiele und schreiben Sie Axiome für die Listendomäne, einschließlich aller Konstanten, Funktionen und Prädikate, die in diesem Kapitel angesprochen wurden.

18 Erklären Sie, was an der folgenden vorgeschlagenen Definition für benachbarte Felder in der Wumpus-Welt falsch ist:

$$\forall x, y \quad Adjacent([x, y], [x+1, y]) \wedge Adjacent([x, y], [x, y+1]).$$

19 Schreiben Sie die erforderlichen Axiome auf, um Schlüsse zur Position des Wumpus zu ziehen. Verwenden Sie dazu das Konstantensymbol $Wumpus$ und ein binäres Prädikat $At(Wumpus, Location)$. Beachten Sie, dass es nur einen Wumpus gibt!

20 Nehmen Sie die Prädikate $Parent(p, q)$ und $Female(p)$ sowie die Konstanten $Joan$ und $Kevin$ an (deren Bedeutungen selbst erklärend sein dürften). Drücken Sie jeden der folgenden Sätze in Logik erster Stufe aus. (Verwenden Sie die Abkürzung $\exists$, um „es gibt genau ein" auszudrücken.)

a. Joan hat eine Tochter (möglicherweise auch mehrere und möglicherweise auch Söhne).

b. Joan hat genau eine Tochter (kann aber auch Söhne haben).

c. Joan hat genau ein Kind, eine Tochter.

d. Joan und Kevin haben genau ein Kind zusammen.

e. Joan hat mindestens ein Kind mit Kevin und keine Kinder mit sonst jemandem.

21 Arithmetische Behauptungen lassen sich in Logik erster Stufe mit dem Prädikatssymbol $<$, den Funktionssymbolen $+$ und $\times$ sowie den Konstantensymbolen 0 und 1 schreiben. Zusätzliche Prädikate können ebenfalls mit Bikonditionalen definiert werden.

a. Stellen Sie die Eigenschaft „x ist eine gerade Zahl" dar.

b. Stellen Sie die Eigenschaft „x ist eine Primzahl" dar.

c. Die Goldbachsche Vermutung ist die (immer noch unbewiesene) Behauptung, dass jede gerade Zahl gleich der Summe von zwei Primzahlen ist. Stellen Sie diese Behauptung als logischen Satz dar.

22 In *Kapitel 6* haben wir mithilfe von Gleichheit die Relation zwischen einer Variablen und ihrem Wert angegeben. Zum Beispiel haben wir mit $WA = red$ ausgedrückt, dass Westaustralien rot gefärbt wird. Um dies in Logik erster Stufe darzustellen, müssen wir etwas weitschweifiger $ColorOf\,(WA) = red$ formulieren. Welche inkorrekte Inferenz ließe sich ziehen, wenn wir Sätze wie $WA = red$ direkt als logische Behauptungen schreiben würden?

23 Schreiben Sie in Logik erster Stufe die Behauptung, dass jeder Schlüssel (Key) und mindestens eine Socke ($Sock$) von jedem Paar ($Pair$) Socken für immer verloren geht ($Lost$), wenn nur das folgende Vokabular verwendet wird: $Key(x)$, x ist ein Schlüssel; $Sock(x)$, x ist eine Socke; $Pair(x, y)$, x und y sind ein Paar; *Now*, die aktuelle Zeit; $Before(t_1\,, t_2)$, Zeit t_1 kommt vor Zeit t_2; *Lost*(x, t), Objekt x geht zur Zeit t verloren.

24 Übersetzen Sie in Logik erster Stufe den Satz: „Die DNA von jedermann ist eindeutig und wird von der DNA ihrer Eltern abgeleitet." Sie müssen die genaue beabsichtigte Bedeutung Ihrer Vokabularterme spezifizieren. (*Hinweis:* Verwenden Sie nicht das Prädikat $Unique(x)$, da Eindeutigkeit keine wirkliche Eigenschaft eines Objektes an sich ist!)

25 Entscheiden Sie für jeden der folgenden deutschen Sätze, ob der begleitende Satz in Logik erster Ordnung eine gute Übersetzung ist. Erläutern Sie andernfalls, warum nicht, und korrigieren Sie ihn.

a. Für jedes Apartment in London ist die Miete geringer als für Apartments in Paris.

$$\forall x\,[Apt(x) \wedge In(x, London)] \Rightarrow \exists y\,([Apt(y) \wedge In(y, Paris)] \Rightarrow (Rent\,(x) < Rent(y))).$$

b. Es gibt genau ein Apartment in Paris mit einer Miete unter $1000.

$$\exists x\; Apt(x) \wedge In(x, Paris) \wedge \forall y\,[Apt(y) \quad In(y, Paris) \wedge (Rent\,(y) < Dollars(1000))] \Rightarrow (y = x).$$

c. Wenn ein Apartment teurer als alle Apartments in London ist, muss es sich in Moskau befinden.

$$\forall x\; Apt(x) \wedge [\forall y\; Apt(y) \wedge In(y, London\,) \wedge (Rent\,(x) > Rent(y))] \Rightarrow In(x, Moscow\,).$$

26 Repräsentieren Sie die folgenden Sätze in Logik erster Stufe und verwenden Sie dafür ein prägnantes Vokabular (das Sie definieren müssen):

a. Einige Studenten haben im Frühjahr 2009 Französisch gewählt.

b. Jeder Student, der Französisch gewählt hat, hat es bestanden.

c. Nur ein Student hat im Frühjahr 2009 Griechisch gewählt.

d. Die beste Note in Griechisch ist immer höher als die beste Note in Französisch.

e. Jede Person, die eine Versicherung abschließt, ist intelligent.

f. Es gibt einen Agenten, der nur für nicht versicherte Menschen Versicherungen abschließt.

g. Es gibt einen Frisör, der alle Männer der Stadt rasiert, die dies nicht selbst machen.

h. Eine Person, die in Großbritannien geboren ist und deren beide Elternteile Bürger oder Einwohner von Großbritannien sind, ist von Geburt an Staatsbürger von Großbritannien.

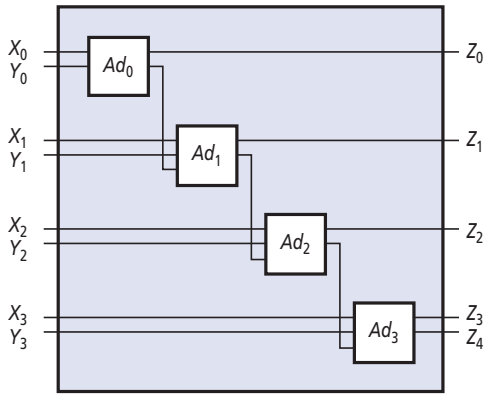
i. Eine Person, die außerhalb von Großbritannien geboren wird und für die ein Elternteil von Geburt an Staatsbürger von Großbritannien ist, ist von der Abstammung her Staatsbürger von Großbritannien.

j. Politiker können einige Menschen immer täuschen und sie können alle Menschen manchmal täuschen, aber sie können nicht alle Menschen immer täuschen.

k. Alle Griechen sprechen die gleiche Sprache. (Verwenden Sie $Speaks(x, l)$ in der Bedeutung, dass Person x die Sprache l spricht.)

27 Schreiben Sie eine allgemeine Menge von Fakten und Axiomen, um die Zusicherung „Wellington hörte von Napoleons Tod" zu repräsentieren und die Frage „Hat Napoleon von Wellingtons Tod gehört?" korrekt zu beantworten.

28 Erweitern Sie das Vokabular aus *Abschnitt 8.4*, um eine Addition für binäre Zahlen mit *n* Bit zu definieren. Anschließend kodieren Sie die Beschreibung des 4-Bit-Addierers aus ▶ Abbildung 8.8 und stellen die Fragen, anhand derer Sie erkennen können, ob der Addierer korrekt arbeitet.



$$\begin{array}{cccccc} & X_3 & X_2 & X_1 & X_0 \\ + & Y_3 & Y_2 & Y_1 & Y_0 \\ \hline Z_4 & Z_3 & Z_2 & Z_1 & Z_0 \end{array}$$

Abbildung 8.8: Ein 4-Bit-Addierer. Jedes Ad_i ist ein 1-Bit-Addierer, wie in Abbildung 8.6 gezeigt.

29 Die Schaltkreisrepräsentation in diesem Kapitel ist detaillierter als nötig, wenn wir nur die Funktionalität des Schaltkreises betrachten wollen. Eine einfachere Formulierung beschreibt ein Gatter oder einen Schaltkreis mit *m* Eingängen und *n* Ausgängen unter Verwendung eines Prädikats mit *m+n* Argumenten, sodass das Prädikat genau dann wahr ist, wenn die Eingänge und Ausgänge konsistent sind. Zum Beispiel werden *NOT*-Gatter durch das binäre Prädikat *NOT(i, o)* beschrieben, für das $NOT(0, 1)$ und $NOT(1, 0)$ bekannt sind. Zusammengesetzte Gatter werden mithilfe von Konjunktionen von Gatterprädikaten definiert, in denen gemeinsam genutzte Variablen direkte Verbindungen darstellen. Beispielsweise lässt sich eine *NAND*-Schaltung aus *AND* und *NOT* zusammensetzen:

$$\forall i_1, i_2, o_a, o \;\; AND(i_1, i_2, o_a) \wedge NOT(o_a, o) \Rightarrow NAND(i_1, i_2, o).$$

Definieren Sie anhand dieser Repräsentation den 1-Bit-Addierer aus Abbildung 8.6 und den 4-Bit-Addierer aus Abbildung 8.8 und erklären Sie, mit welchen Abfragen Sie den Entwurf überprüfen würden. Welche Arten von Abfragen werden von dieser Repräsentation *nicht* unterstützt, die von der Repräsentation in *Abschnitt 8.4* unterstützt werden?

30 Besorgen Sie sich einen Passantrag für Ihr Land und identifizieren Sie die Regeln, anhand derer ermittelt wird, ob ein Antragsteller einen Pass erhalten soll. Übersetzen Sie diese in Logik erster Stufe und folgen Sie dabei den in *Abschnitt 8.4* skizzierten Schritten.

31 Betrachten Sie eine Wissensbasis in Logik erster Stufe, die Welten mit Menschen, Songs, Alben (z.B. „Meet the Beatles") und Datenträgern (d.h. konkreten physischen Instanzen von CDs) beschreibt. Das Vokabular enthält die folgenden Symbole:

$CopyOf(d, a)$: Prädikat. Datenträger d ist eine Kopie von Album a.

$Owns(p, d)$: Prädikat. Person p besitzt Datenträger d.

$Sings(p, s, a)$: Album a umfasst eine Aufnahme des Songs s, gesungen von Person p.

$Wrote(p, s)$: Person p schreibt Song s.

$McCartney$, $Gershwin$, $BHoliday$, Joe, $EleanorRigby$, $ThecManILove$, $Revolver$: Konstanten mit den Bedeutungen, wie sie zweifellos aus der Bezeichnung zu erkennen sind.

Drücken Sie die folgenden Aussagen in Logik erster Stufe aus:

a. Gershwin hat „The Man I Love" geschrieben.

b. Gershwin hat nicht „Eleanor Rigby" geschrieben.

c. Entweder Gershwin oder McCartney hat „The Man I Love" geschrieben.

d. Joe hat mindestens einen Song geschrieben.

e. Joe besitzt ein Exemplar von *Revolver*.

f. Jeder Song, den McCartney auf *Revolver* singt, wurde von McCartney geschrieben.

g. Gershwin hat überhaupt keine Songs auf *Revolver* geschrieben.

h. Jeder Song, den Gershwin geschrieben hat, wurde auf irgendeinem Album aufgenommen. (Möglicherweise wurden verschiedene Songs auf verschiedenen Alben aufgenommen.)

i. Es gibt ein einziges Album, das jeden Song enthält, den Joe geschrieben hat.

j. Joe besitzt ein Exemplar eines Albums, auf dem Billie Holiday „The Man I Love" singt.

k. Joe besitzt ein Exemplar jedes Albums, auf dem ein Song von McCartney gesungen enthalten ist. (Natürlich wird jedes verschiedene Album auf einer anderen physischen CD instanziiert.)

l. Joe besitzt ein Exemplar jedes Albums, auf dem alle Songs von Billie Holiday gesungen werden.

Inferenz in der Logik erster Stufe

9

ÜBERBLICK

In diesem Kapitel definieren wir effektive Prozeduren für die Beantwortung von Fragen, die in Logik erster Stufe gestellt werden.

Kapitel 7 hat gezeigt, wie sich korrekte und vollständige Inferenz für Aussagenlogik erreichen lässt. In diesem Kapitel erweitern wir diese Ergebnisse, um Algorithmen zu erhalten, die alle in Logik erster Stufe ausgedrückten und beantwortbaren Fragen beantworten können. *Abschnitt 9.1* stellt die Inferenzregeln für Quantoren vor und zeigt, wie man die Inferenz der Logik erster Stufe auf eine aussagenlogische Inferenz reduziert – wenn auch mit großem Aufwand. *Abschnitt 9.2* beschreibt das Konzept der **Unifikation** und zeigt, wie es verwendet werden kann, um Inferenzregeln zu konstruieren, die direkt auf Sätze der Logik erster Stufe angewendet werden können. Anschließend diskutieren wir drei wichtige Familien der Inferenzalgorithmen der Logik erster Stufe: Die **Vorwärtsverkettung** und ihre Anwendung auf **deduktive Datenbanken** und **Produktionssysteme** sind in *Abschnitt 9.3* beschrieben; die **Rückwärtsverkettung** und **Logikprogrammiersysteme** werden in *Abschnitt 9.4* entwickelt. Vorwärts- und Rückwärtsverkettung können sehr effizient sein, lassen sich aber nur auf Wissensbasen anwenden, die als Mengen von Horn-Klauseln ausgedrückt werden können. Allgemeine Sätze in Logik erster Stufe erfordern auf Resolution basierende **Theorem-Beweissysteme**, mit denen sich *Abschnitt 9.5* beschäftigt.

9.1 Aussagen- und prädikatenlogische Inferenz

Dieser und der nächste Abschnitt beschreiben die Konzepte, die den modernen logischen Inferenzsystemen zugrunde liegen. Wir beginnen mit einigen einfachen Inferenzregeln, die auf Sätze mit Quantoren angewendet werden können, um Sätze ohne Quantoren zu erhalten. Diese Regeln führen natürlich zu dem Schluss, dass die Inferenz *in der Logik erster Stufe* erfolgen kann, indem man die Wissensbasis in eine *Aussagenlogik* umwandelt und die *aussagenlogische* Inferenz verwendet, die wir bereits kennen. Der nächste Abschnitt weist auf ein offensichtliches Kurzverfahren hin, das zu Inferenzmethoden führt, die Sätze aus der Logik erster Stufe direkt manipulieren.

9.1.1 Inferenzregeln für Quantoren

Wir beginnen mit Allquantoren. Angenommen, unsere Wissensbasis enthält als Axiom die gängige Ansicht, dass alle gierigen Könige böse sind:

$$\forall x \quad König(x) \wedge Gierig(x) \Rightarrow Böse(x).$$

Dann scheint es durchaus erlaubt zu sein, einen der folgenden Sätze abzuleiten:

$$König(John) \wedge Gierig(John) \Rightarrow Böse(John)$$
$$König(Richard) \wedge Gierig(Richard) \Rightarrow Böse(Richard)$$
$$König(Vater(John)) \wedge Gierig(Vater(John)) \Rightarrow Böse(Vater(John))$$
$$\vdots$$

Die Regel der **universellen Instanziierung** (kurz **UI**) besagt, dass wir jeden Satz ableiten können, den wir erhalten, indem wir einen **Grundterm** (einen Term ohne Variab-

len) für die Variable einsetzen.[1] Um die Inferenzregel formal zu schreiben, verwenden wir das in *Abschnitt 8.3* eingeführte Konzept der **Substitutionen**. Mit SUBST(θ, α) werde das Ergebnis der Anwendung der Substitution θ auf den Satz α bezeichnet. Dann wird die Regel geschrieben als

$$\frac{\forall v \alpha}{SUBST(\{v / g\}, \alpha)} .$$

für jede Variable v und den Grundterm g. Die drei oben gezeigten Sätze beispielsweise erhält man mit den Substitutionen $\{x / John\}$, $\{x / Richard\}$ und $\{x / Vater(John)\}$.

In der Regel für **Existentielle Instanziierung** wird die Variable durch ein einzelnes *neues Konstantensymbol* ersetzt. Die formale Anweisung lautet wie folgt: Für jeden Satz α, jede Variable v und jedes Konstantensymbol k, das nicht an anderer Stelle in der Wissensbasis auftritt, ist

$$\frac{\exists v \alpha}{SUBST(\{v / k\}, \alpha)} .$$

Zum Beispiel können wir aus dem Satz

$$\exists x \quad Krone(x) \wedge AufDemKopf(x, John)$$

den Satz

$$Krone(C_1) \wedge AufDemKopf(C_1, John)$$

ableiten, solange C_1 an keiner anderen Stelle in der Wissensbasis vorkommt. Grundsätzlich besagt der existentielle Satz, dass es irgendein Objekt gibt, das eine Bedingung erfüllt. Die Anwendung der existentiellen Instanziierungsregel gibt diesem Objekt einfach einen Namen. Natürlich darf dieser Name noch nicht zu einem anderen Objekt gehören. In der Mathematik finden wir ein schönes Beispiel: Angenommen, wir entdecken, dass es eine Zahl gibt, die ein wenig größer als 2.71828 ist und die die Gleichung $d(x^y)/dy = x^y$ für x erfüllt. Wir können dieser Zahl einen Namen geben, wie etwa e, aber es wäre ein Fehler, ihr den Namen eines bereits existierenden Objektes zu geben, wie etwa π. In der Logik wird der neue Name als **Skolem-Konstante** bezeichnet. Die Existentielle Instanziierung ist ein Sonderfall eines allgemeineren Prozesses, der sogenannten **Skolemisierung**, die wir in *Abschnitt 9.5* beschreiben.

Während die Universelle Instanziierung mehrfach angewendet werden kann, um viele verschiedene Konsequenzen zu produzieren, lässt sich die Existentielle Instanziierung nur einmal anwenden; dann kann der existenzquantifizierte Satz verworfen werden. Nachdem wir beispielsweise den Satz *Töte*(*Mörder*, *Opfer*) hinzugefügt haben, brauchen wir den Satz $\exists x$ *Töte*(x, *Opfer*) nicht mehr. Streng gesagt, ist die neue Wissensbasis nicht logisch äquivalent der alten, aber es kann gezeigt werden, dass sie **inferentiell äquivalent** in dem Sinne ist, dass sie genau dann erfüllbar ist, wenn die ursprüngliche Wissensbasis erfüllbar ist.

1 Verwechseln Sie diese Substitutionen nicht mit den erweiterten Interpretationen, die wir für die Definition der Semantik von Quantoren verwendet haben. Die Substitution ersetzt eine Variable durch einen Term (ein Stück Syntax), um einen neuen Satz zu erzeugen, während eine Interpretation eine Variable auf ein Objekt in der Domäne abbildet.

9.1.2 Reduzierung auf aussagenlogische Inferenz

Nachdem wir Regeln für die Ableitung nicht quantifizierter Sätze von quantifizierten Sätzen haben, wird es möglich, die Inferenz der Logik erster Stufe auf eine aussagenlogische Inferenz zu reduzieren. In diesem Abschnitt beschreiben wir die wichtigsten Ideen dafür; die Details finden Sie in *Abschnitt 9.5*.

Die erste Idee ist, dass sich ein allquantifizierter Satz durch die Menge *aller möglichen* Instanziierungen ersetzen lässt, genau wie ein existenzquantifizierter Satz durch eine Instanziierung ersetzt werden kann. Angenommen, unsere Wissensbasis enthält nur die folgenden Sätze:

$$\begin{aligned} &\forall x \quad \mathit{König}(x) \wedge \mathit{Gierig}(x) \Rightarrow \mathit{Böse}(x) \\ &\mathit{König}(\mathit{John}) \\ &\mathit{Gierig}(\mathit{John}) \\ &\mathit{Bruder}(\mathit{Richard}, \mathit{John}). \end{aligned} \tag{9.1}$$

Dann wenden wir UI auf den ersten Satz an und verwenden dafür alle möglichen Grundterm-Substitutionen aus dem Vokabular der Wissensbasis – in diesem Fall $\{x / \mathit{John}\}$ und $\{x / \mathit{Richard}\}$. Wir erhalten damit:

$$\begin{aligned} &\mathit{König}(\mathit{John}) \wedge \mathit{Gierig}(\mathit{John}) \Rightarrow \mathit{Böse}(\mathit{John}) \\ &\mathit{König}(\mathit{Richard}) \wedge \mathit{Gierig}(\mathit{Richard}) \Rightarrow \mathit{Gierig}(\mathit{Richard}) \end{aligned}$$

und verwerfen den allquantifizierten Satz. Jetzt ist die Wissensbasis im Wesentlichen aussagenlogisch, wenn wir die grundlegenden atomaren Sätze – *König*(*John*), *Gierig*(*John*) usw. – als Aussagensymbole betrachten. Aus diesem Grund können wir jeden der vollständigen aussagenlogischen Algorithmen aus *Kapitel 7* anwenden, um Schlüsse wie z.B. *Böse*(*John*) zu erhalten.

Wie *Abschnitt 9.5* noch zeigt, lässt sich diese Technik der **Umwandlung in die Aussagenlogik** völlig verallgemeinern. Das bedeutet, dass jede Wissensbasis und Abfrage der Logik erster Stufe so in die Aussagenlogik überführt werden kann, dass die logische Konsequenz beibehalten wird. Damit haben wir eine vollständige Entscheidungsprozedur für die logische Konsequenz ..., vielleicht aber auch nicht. Es gibt ein Problem: Wenn die Wissensbasis ein Funktionssymbol enthält, ist die Menge möglicher Grundterm-Substitutionen unendlich! Erwähnt beispielsweise die Wissensbasis das Symbol *Vater*, können unendlich viele verschachtelte Terme wie etwa *Vater*(*Vater*(*Vater*(*John*))) konstruiert werden. Unsere aussagenlogischen Algorithmen haben Schwierigkeiten mit einer unendlich großen Menge von Sätzen.

Glücklicherweise gibt es ein berühmtes Theorem, das auf Jacques Herbrand (1930) zurückgeht. Es besagt, dass, wenn ein Satz aus der ursprünglichen Wissensbasis erster Stufe logisch folgt, es auch einen Beweis gibt, der mit einer *endlichen* Untermenge der in Aussagenlogik umgewandelten Wissensbasis zu führen ist. Weil jede dieser Untermengen eine maximale Tiefe der Verschachtelung ihrer Grundterme aufweist, finden wir die Untermenge, indem wir zuerst alle Instanziierungen mit Konstantensymbolen (*Richard* und *John*) erzeugen, dann alle Terme der Tiefe 1 (*Vater*(*Richard*) und *Vater*(*John*)), dann alle Terme der Tiefe 2 usw., bis wir in der Lage sind, einen aussagenlogischen Beweis aller logisch konsequenten Sätze zu konstruieren.

Tipp

Wir haben einen Ansatz für die Inferenz der Logik erster Stufe aufgezeigt (durch Umwandlung in Aussagenlogik), der **vollständig** ist – d.h., jeder logisch konsequente Satz kann bewiesen werden. Damit wurde ein großes Ziel erreicht, wenn man bedenkt,

dass der Raum möglicher Modelle unendlich ist. Andererseits wissen wir erst, nachdem der Beweis erfolgt ist, dass der Satz tatsächlich eine logische Konsequenz der Wissensbasis *ist*. Was passiert, wenn der Satz *nicht* logisch folgt? Können wir das feststellen? Für die Logik erster Stufe können wir das nicht. Unsere Beweisprozedur kann immer weiterlaufen und immer tiefer verschachtelte Terme erzeugen, aber wir wissen nicht, ob sie in einer hoffnungslosen Schleife festhängt oder ob der Beweis im nächsten Moment herauskommt. Das ist genau das Problem, das wir von Turing-Maschinen kennen. Alan Turing (1936) und Alonzo Church (1936) haben beide auf unterschiedliche Art die Unvermeidbarkeit dieses Sachverhaltes bewiesen. *Die Frage der logischen Konsequenz in der Logik erster Stufe ist* **semientscheidbar** *– d.h., es gibt Algorithmen, die jede logische Konsequenz akzeptieren, aber es gibt keinen Algorithmus, der jeden nicht logisch folgenden Satz identifiziert.*

9.2 Unifikation und Lifting

Der vorige Abschnitt hat das Verständnis der Inferenz in der Logik erster Stufe beschrieben, wie es bis Anfang der 1960er Jahre existierte. Der aufmerksame Leser (und sicherlich auch jeder Logiker der frühen 1960er Jahre) hat bemerkt, dass dieser Ansatz der Umwandlung in die Aussagenlogik recht ineffizient ist. Hat man beispielsweise die Abfrage $Böse(x)$ und die Wissensbasis aus Gleichung (9.1), scheint es unsinnig zu sein, Sätze wie etwa $König(Richard) \wedge Gierig(Richard) \Rightarrow Böse(Richard)$ zu erzeugen. Die Inferenz von $Böse(John)$ aus Sätzen wie

$$\forall x \quad König(x) \wedge Gierig(x) \Rightarrow Böse(x)$$
$$König(John)$$
$$Gierig(John)$$

scheint für einen Menschen völlig offensichtlich. Wir zeigen jetzt, wie wir sie für einen Computer offensichtlich machen.

9.2.1 Eine Inferenzregel der Logik erster Stufe

Die Inferenz, dass John böse ist – d.h. dass $\{x / John\}$ die Abfrage $Böse(x)$ löst –, funktioniert wie folgt: Man verwendet die Regel, dass gierige Könige böse sind, findet ein x, sodass x ein König und x gierig ist, und leitet dann ab, dass x böse ist. Allgemeiner gesagt, wenn es eine Substitution θ gibt, die jede der Konjunkten der Prämisse der Implikation identisch mit Sätzen macht, die sich bereits in der Wissensbasis befinden, können wir den Schluss der Implikation behaupten, nachdem wir θ angewendet haben. In diesem Fall erreicht $\theta = \{x / John\}$ das Ziel.

Wir können dem Inferenzschritt sogar noch mehr Arbeit überlassen. Anstatt beispielsweise zu wissen, dass $Gierig(John)$ gilt, nehmen wir an zu wissen, dass *jeder* gierig ist:

$$\forall y \quad Gierig(y). \tag{9.2}$$

Wir wollen dann immer noch schließen können, dass $Böse(John)$ gilt, weil wir wissen, dass John ein König ist (das ist gegeben) und dass John gierig ist (weil jeder gierig ist). Damit das funktioniert, müssen wir eine Substitution sowohl für die Variablen im Implikationssatz als auch für die Variablen in dem Satz, für den eine Übereinstimmung gesucht wird, finden. In diesem Fall hat die Anwendung der Substitution $\{x / John, y / John\}$

auf die Implikation die Prämisse *König*(x) und *Gierig*(x) und die Wissensbasissätze *König*(*John*) und *Gierig*(y) machen sie identisch. Wir können also den Schluss der Implikation ableiten.

Dieser Inferenzprozess kann als einzige Inferenzregel formuliert werden, die wir als **Verallgemeinerten Modus ponens** bezeichnen:[2] Gibt es für atomare Sätze p_i, p_i' und q eine Substitution θ mit SUBST(θ,p_i') = SUBST(θ,p_i) für alle i, dann gilt:

$$\frac{p_1', p_2', \ldots, p_n', (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{SUBST(\theta, q)} .$$

Es gibt n+1 Prämissen für diese Regel: die n atomaren Sätze p_i' und die eine Implikation. Der Schluss ist das Ergebnis der Anwendung der Substitution θ auf die Konsequenz q. Für unser Beispiel ist das:

p_1' ist *König*(*John*)	p_1 ist *König*(x)
p_2' ist *Gierig*(y)	p_2 ist *Gierig*(x)
θ ist {x / *John*, y / *John* }	q ist *Böse*(x)
SUBST(θ, q) ist *Böse*(*John*)	

Es ist einfach zu zeigen, dass der Verallgemeinerte Modus ponens eine korrekte Inferenzregel ist. Zuerst beobachten wir, dass für jeden Satz p (dessen Variablen allquantifiziert sind) und für jede Substitution θ durch universelle Instanziierung

$$p \models \text{SUBST}(\theta, p)$$

gilt. Es gilt insbesondere für ein θ, das die Bedingungen für den Verallgemeinerten Modus ponens erfüllt. Aus p_1', ... p_n' können wir also ableiten:

$$\text{SUBST}(\theta, p_1') \wedge \ldots \wedge Subst(\theta, p_n').$$

Und aus der Implikation $p_1 \wedge \ldots \wedge p_n \Rightarrow q$ können wir ableiten:

$$\text{SUBST}(\theta, p_1) \wedge \ldots \wedge Subst(\theta, p_n) \Rightarrow Subst(\theta, q).$$

Nun ist θ im Verallgemeinerten Modus ponens so definiert, dass SUBST(θ, p_i') = SUBST(θ,p_i) für alle i gilt; deshalb stimmt der erste dieser beiden Sätze genau mit der Prämisse des zweiten überein. Damit folgt SUBST(θ, q) durch den Modus ponens.

Der Verallgemeinerte Modus ponens ist eine **geliftete** (**angehobene**) Version des Modus ponens – er erhebt den Modus ponens von der (variablenfreien) Grundaussagenlogik in die Logik erster Stufe. Wir werden im restlichen Kapitel sehen, dass wir geliftete Versionen der Vorwärtsverkettung, der Rückwärtsverkettung und der in *Kapitel 7* eingeführten Resolutionsalgorithmen entwickeln können. Der wichtigste Vorteil bei gelifteten Inferenzregeln gegenüber der Umwandlung in die Aussagenlogik ist, dass sie nur die Substitutionen vornehmen, die erforderlich sind, um die Ausführung bestimmter Inferenzen zu ermöglichen.

2 Der verallgemeinerte Modus ponens ist allgemeiner als der Modus ponens (*Abschnitt 7.5.1*), und zwar in dem Sinne, dass die bekannten Fakten und die Prämisse der Implikation nur bis zu einer Substitution und nicht exakt übereinstimmen müssen. Andererseits erlaubt Modus ponens jeden Satz α als die Prämisse und nicht nur eine Konjunktion von atomaren Sätzen.

9.2.2 Unifikation

Für geliftete Inferenzregeln müssen Substitutionen gefunden werden, die bewirken, dass unterschiedliche logische Ausdrücke identisch aussehen. Dieser Prozess wird auch als **Unifikation** (Vereinheitlichung) bezeichnet und ist eine Schlüsselkomponente aller Inferenzalgorithmen der Logik erster Stufe. Der Algorithmus UNIFY nimmt zwei Sätze entgegen und gibt einen **Unifikator** für sie zurück, falls es einen solchen gibt:

$$\text{UNIFY}(p, q) = \theta \text{ wobei } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q).$$

Wir betrachten jetzt einige Beispiele dafür, wie sich UNIFY verhalten soll. Angenommen, wir haben die Abfrage ASKVARS(*Kennt*(*John*, *x*)): Wen kennt John? Einige Antworten auf diese Abfrage findet man, indem man alle Sätze in der Wissensbasis sucht, die sich mit *Kennt*(*John*, *x*) unifizieren lassen. Nachfolgend sehen Sie die Ergebnisse einer Unifikation mit vier verschiedenen Sätzen, die sich in der Wissensbasis befinden könnten.

$$\text{UNIFY}(\mathit{Kennt}(\mathit{John}, x), \mathit{Kennt}(\mathit{John}, \mathit{Jane})) = \{x / \mathit{Jane}\}$$

$$\text{UNIFY}(\mathit{Kennt}(\mathit{John}, x), \mathit{Kennt}(y, \mathit{Bill})) = \{x / \mathit{Bill}, y / \mathit{John}\}$$

$$\text{UNIFY}(\mathit{Kennt}(\mathit{John}, x), \mathit{Kennt}(y, \mathit{Mutter}(y))) = \{y / \mathit{John}, x / \mathit{Mutter}(\mathit{John})\}$$

$$\text{UNIFY}(\mathit{Kennt}(\mathit{John}, x), \mathit{Kennt}(x, \mathit{Elizabeth})) = \mathit{fail}$$

Die letzte Unifikation schlägt fehlt, weil *x* nicht gleichzeitig die Werte *John* und *Elizabeth* annehmen kann. Nachdem *Kennt*(*x*, *Elizabeth*) bedeutet: „Jeder kennt Elizabeth", *sollten* wir auch ableiten können, dass John Elizabeth kennt. Das Problem tritt nur auf, weil die beiden Sätze denselben Variablennamen *x* verwenden. Es lässt sich vermeiden, indem einer der beiden zu unifizierenden Sätze **standardisiert** (bereinigt) wird, d.h., seine Variablen werden umbenannt, um Namenskonflikte zu vermeiden. Beispielsweise können wir das *x* aus *Kennt*(*x*, *Elizabeth*) in x_{17} umbenennen (das ist ein neuer Variablenname), ohne dass sich die Bedeutung ändert. Jetzt funktioniert die Unifikation:

$$\text{UNIFY}(\mathit{Kennt}(\mathit{John}, x), \mathit{Kennt}(x_{17}, \mathit{Elizabeth})) = \{x / \mathit{Elizabeth}, x_{17} / \mathit{John}\}.$$

In Übung 9.13 werden Sie ein weiteres Beispiel kennenlernen, wo eine Standardisierung erforderlich ist.

Es gibt noch eine Komplikation: Wir haben gesagt, dass UNIFY eine Substitution zurückgeben soll, die bewirkt, dass die beiden Argumente gleich aussehen. Es könnte jedoch mehrere solcher Unifikatoren geben. Beispielsweise könnte UNIFY(*Kennt*(*John*, *x*), *Kennt*(*y*, *z*)) die Substitutionen {*y* / *John*, *x* / *z*} oder {*y* / *John*, *x* / *John*, *z* / *John* } zurückgeben. Der erste Unifikator gibt *Kennt*(*John*, *z*) als Ergebnis der Unifikation zurück, während der zweite *Kennt*(*John*, *John*) liefert. Das zweite Ergebnis könnte man aus dem ersten durch eine weitere Substitution { *z* / *John* } erhalten; wir sagen, der erste Unifikator ist *allgemeiner* als der zweite, weil er weniger Randbedingungen für die Werte der Variablen festlegt. Es stellt sich heraus, dass es für jedes unifizierbare Paar von Ausdrücken einen **kleinsten gemeinsamen** oder **allgemeinsten Unifikator** gibt (MGU, Most General Unifier), der eindeutig bis zur Umbenennung und Substitution von Variablen ist. (Zum Beispiel sind {*x*/*John*} und {*y*/*John*} als äquivalent zu betrachten, wie auch {*x*/*John*, *y*/*John*} und {*x*/*John*, *y*/*x*}.) In unserem Beispiel ist es {*y* / *John*, *x* / *z*}.

▶ Abbildung 9.1 zeigt einen Algorithmus für die Berechnung der allgemeinsten Unifikatoren. Der Prozess ist ganz einfach: Die beiden Ausdrücke werden simultan „Seite an Seite“ rekursiv ausgewertet, wobei ein Unifikator aufgebaut wird, was aber scheitert, wenn zwei einander entsprechende Punkte in den Strukturen nicht übereinstimmen. Es gibt nur einen aufwändigen Schritt: Wenn eine Variable mit einem komplexen Term verglichen wird, muss man prüfen, ob die Variable selbst in dem Term vorkommt; ist dies der Fall, schlägt der Vergleich fehl, weil kein konsistenter Unifikator konstruiert werden kann. Zum Beispiel lässt sich $S(x)$ nicht mit $S(S(x))$ vereinheitlichen. Dieser sogenannte **Occur Check** (**Vorkommensprüfung**) bewirkt, dass die Komplexität des gesamten Algorithmus quadratisch in der Größe der zu unifizierenden Ausdrücke wird. Einige Systeme, einschließlich aller Logikprogrammiersysteme, lassen den Occur Check einfach weg und führen deshalb manchmal nicht korrekte Inferenzen aus; andere Systeme verwenden komplexere Algorithmen mit linearer Zeitkomplexität.

```
function UNIFY(x, y, θ) returns eine Substitution,
                                um x und y identisch zu machen
  inputs: x, eine Variable, Konstante, Liste oder ein Verbundausdruck
          y, eine Variable, Konstante, Liste oder ein Verbundausdruck
          θ, die bisher aufgebaute Substitution (optional, standardmäßig leer)

  if θ = Fehler then return Fehler
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
    return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) and LIST?(y) then
    return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return Fehler
```

```
function UNIFY-VAR(var, x, θ) returns eine Substitution

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return Fehler
  else return füge {var/x} zu θ hinzu
```

Abbildung 9.1: Der Unifikationsalgorithmus. Der Algorithmus vergleicht elementweise die Struktur der Eingaben. Dabei wird die Substitution θ, die das Argument von UNIFY darstellt, aufgebaut und verwendet, um sicherzustellen, dass spätere Vergleiche mit früher eingerichteten Bindungen konsistent sind. In einem zusammengesetzten Ausdruck, wie etwa $F(A, B)$, ermittelt die Funktion OP das Funktionssymbol F und die Funktion ARGS ermittelt die Argumentliste (A, B).

9.2.3 Speichern und Abrufen

Den Funktionen TELL und ASK für Aufbau und Abfrage einer Wissensbasis liegen die elementareren Funktionen STORE und FETCH zugrunde. STORE(s) speichert einen Satz s in einer Wissensbasis, während FETCH(q) alle Unifikatoren zurückgibt, sodass die Abfrage q mit einem der Sätze in der Wissensbasis vereinheitlicht wird. Das Problem, das wir für die Demonstration der Unifikation verwendet haben – die Ermittlung von Fakten, die mit *Kennt*(*John*, x) unifizieren –, ist ein Beispiel für die Anwendung von FETCH.

Am einfachsten lassen sich STORE und FETCH implementieren, wenn man alle Fakten in einer langen Liste verwaltet und jede Abfrage gegen jedes Element der Liste unifiziert. Diese Vorgehensweise ist zwar ineffizient, funktioniert aber, und Sie brauchen nicht mehr, um das restliche Kapitel zu verstehen. Der restliche Abschnitt zeigt Möglichkeiten auf, das Abrufen effizienter zu machen, und kann beim ersten Durchlesen übersprungen werden.

Wir können FETCH effizienter machen, indem wir sicherstellen, dass Unifikationen nur für solche Sätze angewendet werden, für die eine *gewisse* Wahrscheinlichkeit besteht, dass sie unifiziert werden. Beispielsweise ist es sinnlos, *Kennt*(*John*, *x*) mit *Bruder*(*Richard*, *John*) zu vereinheitlichen. Derartige Unifikationen können wir vermeiden, indem wir die Fakten in der Wissensbasis **indizieren**. Ein einfaches Schema, die **Prädikatenindizierung**, legt alle *Kennt*-Fakten in einem Behälter ab, alle *Bruder*-Fakten in einem anderen. Die Behälter können in einer Hash-Tabelle abgelegt werden, um einen effizienten Zugriff zu ermöglichen.

Die Prädikatenindizierung ist sinnvoll, wenn es mehrere Prädikatssymbole, aber nur wenige Klauseln für jedes Symbol gibt. In einigen Anwendungen gibt es jedoch viele Klauseln für ein bestimmtes Prädikatssymbol. Angenommen, die Steuerbehörden wollen überwachen, wer wen beschäftigt, und verwenden dazu das Prädikat *Beschäftigt*(*x*, *y*). Das wäre ein sehr großer Behälter mit wahrscheinlich Millionen von Arbeitgebern und noch mehr Arbeitnehmern. Die Beantwortung einer Abfrage wie *Beschäftigt*(*x*, *Richard*) mithilfe der Prädikatenindizierung bedingt das Durchsuchen des gesamten Behälters.

Für diese Abfrage wäre es hilfreich, wenn die Fakten sowohl nach dem Prädikat als auch nach dem zweiten Argument indiziert wären, vielleicht unter Verwendung eines kombinierten Hashtabellen-Schlüssels. Dann könnten wir den Schlüssel einfach aus der Abfrage konstruieren und genau die Fakten finden, die mit der Abfrage unifizieren. Für andere Abfragen, wie etwa *Beschäftigt*(*IBM*, *y*), müssten wir die Fakten indizieren, indem wir das Prädikat mit dem ersten Argument kombinieren. Dazu können Fakten unter mehreren Indexschlüsseln gespeichert werden, sodass unterschiedliche Abfragen, mit denen sie vereinheitlicht werden könnten, unmittelbar zugänglich sind.

Für einen Satz, der gespeichert werden soll, ist es möglich, Indizes für *alle möglichen* Abfragen zu konstruieren, die sich mit ihm vereinigen. Für den Fakt *Beschäftigt*(*IBM*, *Richard*) lauten die Abfragen:

Beschäftigt(*IBM*, *Richard*)	Beschäftigt IBM Richard?
Beschäftigt(*x*, *Richard*)	Wer beschäftigt Richard?
Beschäftigt(*IBM*, *y*)	Wen beschäftigt IBM?
Beschäftigt(*x*, *y*)	Wer beschäftigt wen?

Diese Abfragen bilden einen **Subsumtionsverband**, wie in ▶ Abbildung 9.2(a) gezeigt. Der Verband hat einige interessante Eigenschaften. Beispielsweise wird das Kind jedes Knotens im Verband mithilfe einer einzigen Subsumtion aus seinen Eltern ermittelt; der „höchste“ gemeinsame Nachfahre von zwei Knoten ist das Ergebnis der Anwendung ihres allgemeinsten Unifikators. Der Teil des Verbandes über einem beliebigen Grundfakt kann systematisch konstruiert werden (Übung 9.5). Ein Satz mit wiederholten Konstanten hat einen etwas anderen Verband, wie in ▶ Abbildung 9.2(b) gezeigt. Funktionssymbole und Variablen in den zu speichernden Sätzen führen zu noch interessanteren Verbandsstrukturen.

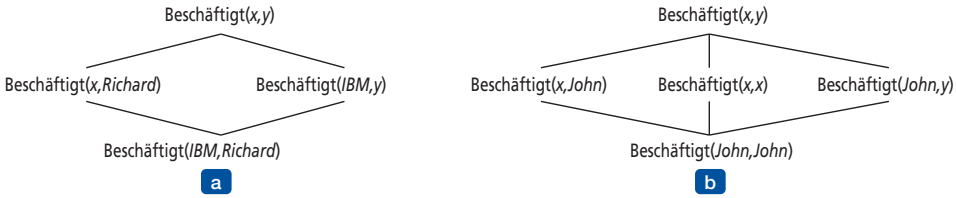


Abbildung 9.2: (a) Der Subsumtionsverband, dessen niedrigster Knoten der Satz *Beschäftigt*(*IBM*, *Richard*) ist. (b) Der Subsumtionsverband für den Satz *Beschäftigt*(*John*, *John*).

Das bisher beschriebene Schema funktioniert gut, wenn der Verband eine kleine Anzahl an Knoten enthält. Für ein Prädikat mit n Argumenten enthält der Verband $O(2^n)$ Knoten. Wenn Funktionssymbole erlaubt sind, ist die Anzahl der Knoten außerdem exponentiell in der Größe der Terme in dem zu speichernden Satz. Das kann zu einer riesigen Anzahl an Indizes führen. An einem bestimmten Punkt werden die Vorteile der Indizierung durch die Kosten für das Speichern und die Verwaltung all dieser Indizes aufgehoben. Wir können reagieren, indem wir eine feststehende Strategie anwenden, beispielsweise nur solche Schlüssel verwalten, die aus einem Prädikat plus allen Argumenten bestehen, oder eine adaptive Strategie verwenden, die Indizes erzeugt, um den Anforderungen der gestellten Abfragearten gerecht zu werden. Für die meisten KI-Systeme ist die Anzahl der zu speichernden Fakten klein genug, dass eine effiziente Indizierung als gelöstes Problem betrachtet werden kann. Für kommerzielle Datenbanken, wo die Anzahl der Fakten in die Milliarden geht, ist das Problem Gegenstand intensiver Untersuchungen und technologischer Entwicklungen gewesen.

9.3 Vorwärtsverkettung

In *Abschnitt 7.5* wurde ein Algorithmus für die Vorwärtsverkettung für aussagenlogische definite Klauseln vorgestellt. Die Idee dabei ist einfach: Man beginnt mit den atomaren Sätzen in der Wissensbasis, wendet den Modus ponens in Vorwärtsrichtung an und fügt neue atomare Sätze ein, bis keine weiteren Inferenzen gemacht werden können. Hier erklären wir, wie der Algorithmus auf definite Klauseln erster Stufe angewendet wird und wie er effizient implementiert werden kann. Definite Klauseln wie etwa *Situation* $\Rightarrow$ *Antwort* sind besonders praktisch für Systeme, die Inferenzen machen, um auf neu eingetroffene Information zu reagieren. Viele Systeme können auf diese Weise definiert werden und das Schließen mit Vorwärtsverkettung lässt sich sehr effizient implementieren.

9.3.1 Definite Klauseln erster Stufe

Die definiten Klauseln erster Stufe sind den aussagenlogischen definiten Klauseln sehr ähnlich (*Abschnitt 7.5.3*): Es handelt sich dabei um Disjunktionen von Literalen, *von denen genau eines positiv ist*. Eine definite Klausel ist entweder atomar oder sie ist eine Implikation, deren Antezedenz eine Konjunktion positiver Literale und deren Konsequenz ein einziges positives Literal ist. Nachfolgend sehen Sie definite Klauseln erster Stufe:

$$König(x) \wedge Gierig(x) \Rightarrow Böse(x)$$
$$König(John)$$
$$Gierig(y).$$

Anders als aussagenlogische Literale können Literale in der Logik erster Stufe Variablen enthalten, wobei man annimmt, dass diese Variablen allquantifiziert sind. (Normalerweise lassen wir Allquantoren weg, wenn wir definite Klauseln schreiben.) Nicht jede Wissensbasis kann in eine Menge definiter Klauseln umgewandelt werden, weil es die Einschränkung mit dem einzelnen positiven Literal gibt – aber viele können. Betrachten Sie das folgende Problem:

> Laut Gesetz ist es für einen Amerikaner ein Verbrechen, Waffen an feindliche Nationen zu verkaufen. Das Land Nono, ein Feind von Amerika, besitzt einige Raketen und alle seine Raketen wurden ihm von Colonel West verkauft, der Amerikaner ist.

Wir werden beweisen, dass West ein Krimineller ist. Zuerst werden wir diese Fakten als definite Klauseln erster Stufe repräsentieren. Der nächste Abschnitt zeigt, wie der Algorithmus für die Vorwärtsverkettung das Problem löst.

„... ist es für einen Amerikaner ein Verbrechen, Waffen an feindliche Nationen zu verkaufen“:

$$\mathit{Amerikaner}(x) \wedge \mathit{Waffe}(y) \wedge \mathit{Verkauft}(x, y, z) \wedge \mathit{Feindlich}(z) \Rightarrow \mathit{Kriminell}(x). \quad (9.3)$$

„Nono ... besitzt einige Raketen.“ Der Satz $\exists x\ \mathit{Besitzt}(\mathit{Nono}, x) \wedge \mathit{Rakete}(x)$ wird durch die Existentielle Instantiierung in zwei definite Klauseln umgewandelt, wobei eine neue Konstante M_1 eingeführt wird:

$$\mathit{Besitzt}(\mathit{Nono}, M_1) \quad (9.4)$$
$$\mathit{Rakete}(M_1) \quad (9.5)$$

„Alle seine Raketen wurden ihm von Colonel West verkauft“:

$$\mathit{Rakete}(x) \wedge \mathit{Besitzt}(\mathit{Nono}, x) \Rightarrow \mathit{Verkauft}(\mathit{West}, x, \mathit{Nono}). \quad (9.6)$$

Außerdem müssen wir wissen, dass Raketen Waffen sind:

$$\mathit{Rakete}(x) \Rightarrow \mathit{Waffe}(x). \quad (9.7)$$

Und wir müssen wissen, dass ein Feind Amerikas als „feindlich“ gilt:

$$\mathit{Feind}(x, \mathit{Amerika}) \Rightarrow \mathit{Feindlich}(x). \quad (9.8)$$

„West, der Amerikaner ist ...“:

$$\mathit{Amerikaner}(\mathit{West}). \quad (9.9)$$

„Das Land Nono, ein Feind Amerikas ...“:

$$\mathit{Feind}(\mathit{Nono}, \mathit{Amerika}). \quad (9.10)$$

Diese Wissensbasis enthält keine Funktionssymbole und ist deshalb eine Instanz der Klasse der **Datalog**-Datenbanken. Datalog ist eine Sprache, die auf definite Klauseln erster Stufe ohne Funktionssymbole beschränkt ist. Ihr Name ergibt sich daraus, dass sie die Art der Anweisungen darstellen kann, die in relationalen Datenbanken üblich sind. Wir werden sehen, dass das Fehlen von Funktionssymbolen die Inferenz sehr viel einfacher macht.

9.3.2 Ein einfacher Algorithmus für die Vorwärtsverkettung

Der erste Algorithmus für die Vorwärtsverkettung, den wir hier betrachten wollen, ist sehr einfach, wie in ▶ Abbildung 9.3 gezeigt. Beginnend bei den bekannten Fakten führt er alle Regeln aus, deren Prämissen erfüllt sind, und fügt ihre Schlüsse den bekannten Fakten hinzu. Der Prozess wird wiederholt, bis die Abfrage beantwortet ist (vorausgesetzt, es ist nur eine Antwort erforderlich) oder bis keine neuen Fakten mehr hinzugefügt werden. Beachten Sie, dass ein Fakt nicht „neu" ist, wenn er nur eine **Umbenennung** eines bekannten Fakts darstellt. Ein Satz ist eine Umbenennung eines anderen, wenn die Sätze bis auf die Namen der Variablen identisch sind. Beispielsweise sind *Mag*(*x*, *Eis*) und *Mag*(*y*, *Eis*) Umbenennungen voneinander, weil sie sich nur in der Wahl von *x* oder *y* unterscheiden; ihre Bedeutungen sind identisch: Jeder mag Eis.

```
function FOL-FC-Ask(KB, α) returns eine Substitution oder false
  inputs: KB, die Wissensbasis, eine Menge definiter Klauseln erster Stufe
          α, die Abfrage, ein atomarer Satz
  local variables: new, die neuen Sätze, die bei jeder Iteration abgeleitet
                   werden

  repeat until new ist leer
    new ← { }
    for each rule in KB do
      (p1 ∧ … ∧ pn ⇒ q) ← Standardize-Variables(rule)
      for each θ derart, dass Subst(θ, p1 ∧ … ∧ pn) = Subst(θ, p1' ∧ … ∧ pn')
                  für einige p1', …, pn' in KB
        q' ← Subst(θ, q)
        if q' wird nicht unifiziert mit einem in KB enthaltenen Satz oder new then
          füge q' zu new hinzu
          ϕ ← Unify(q', α)
          if ϕ ist nicht fail then return ϕ
    füge new zu KB hinzu
  return false
```

Abbildung 9.3: Ein konzeptionell einfacher, aber sehr ineffizienter Algorithmus für die Vorwärtsverkettung. Bei jeder Iteration fügt er zu *KB* die atomaren Sätze hinzu, die in einem Schritt von den Implikationssätzen und den bereits in *KB* enthaltenen atomaren Sätzen abgeleitet werden können. Die Funktion Standardize-Variables ersetzt alle Variablen in ihren Argumenten durch die neuen, die vorher noch nicht verwendet wurden.

Wir wenden wieder unser Problem aus der Welt der Verbrecher an, um zu zeigen, wie FOL-FC-Ask funktioniert. Die Implikationssätze sind (9.3), (9.6), (9.7) und (9.8). Es sind zwei Iterationen erforderlich:

- In der ersten Iteration hat Regel (9.3) nicht erfüllte Prämissen.

  Regel (9.6) ist erfüllt mit $\{x / M_1\}$ und *Verkauft*(*West*, M_1, *Nono*) wird hinzugefügt.

  Regel (9.7) ist erfüllt mit $\{x / M_1\}$ und *Waffe*(M_1) wird hinzugefügt.

  Regel (9.8) ist erfüllt mit {*x* / *Nono*} und *Feindlich*(*Nono*) wird hinzugefügt.

- In der zweiten Iteration ist Regel (9.3) mit $\{x / West, y / M_1, z / Nono\}$ erfüllt und *Kriminell*(*West*) wird hinzugefügt.

▶ Abbildung 9.4 zeigt den erzeugten Beweisbaum. Beachten Sie, dass an dieser Stelle keine neuen Inferenzen mehr möglich sind, weil jeder Satz, der durch die Vorwärtsverkettung geschlossen werden konnte, bereits explizit in der Wissensbasis enthalten ist. Eine solche Wissensbasis wird auch als **Fixpunkt** des Inferenzprozesses bezeichnet. Fix-

punkte, die durch Vorwärtsverkettung mit definiten Klauseln erster Stufe erzielt werden, sind denen für die aussagenlogische Vorwärtsverkettung (*Abschnitt 7.5.4*) sehr ähnlich; der wichtigste Unterschied ist, dass ein Fixpunkt erster Stufe allquantifizierte atomare Sätze enthalten kann.

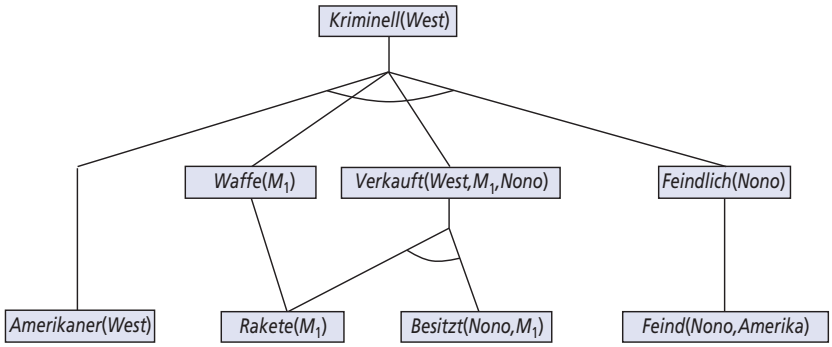


Abbildung 9.4: Der Beweisbaum, der durch die Vorwärtsverkettung für das Verbrecherbeispiel erzeugt wird. Die anfänglichen Fakten erscheinen auf der untersten Ebene, Fakten, die während der ersten Iteration abgeleitet wurden, in der mittleren Ebene und Fakten, die während der zweiten Iteration abgeleitet wurden, auf der obersten Ebene.

FOL-FC-Ask ist einfach zu analysieren. Erstens ist die Funktion **korrekt**, weil jede Inferenz einfach nur eine Anwendung des Verallgemeinerten Modus ponens ist, der korrekt ist. Zweitens ist sie **vollständig** für Wissensbasen mit definiten Klauseln; das bedeutet, sie beantwortet jede Abfrage, deren Antworten logische Konsequenzen einer Wissensbasis mit definiten Klauseln sind. Für Datalog-Wissensbasen, die keine Funktionssymbole enthalten, ist der Beweis der Vollständigkeit relativ einfach. Wir beginnen damit, die Anzahl möglicher Fakten zu zählen, die hinzugefügt werden können, was die Anzahl der Iterationen bestimmt. Es sei k die maximale **Stelligkeit** (Anzahl der Argumente) eines beliebigen Prädikats, p die Anzahl der Prädikate und n die Anzahl der Konstantensymbole. Offensichtlich kann es nicht mehr als pn^k unterschiedliche Grundfakten geben; der Algorithmus muss also nach so vielen Iterationen einen Fixpunkt erreichen. Dann können wir ein Argument erzeugen, das ganz ähnlich dem Beweis der Vollständigkeit für die aussagenlogische Vorwärtsverkettung ist (*Abschnitt 7.5.4*). Die Details, wie der Übergang von der aussagenlogischen Vollständigkeit zur Vollständigkeit in der Logik erster Stufe erfolgen kann, sind für den Resolutionsalgorithmus in *Abschnitt 9.5* beschrieben.

Für allgemeine definite Klauseln mit Funktionssymbolen kann FOL-FC-Ask unendlich viele neue Fakten erzeugen; deshalb müssen wir hier sorgfältiger sein. Für den Fall, in dem eine Antwort auf den Abfragesatz q eine logische Konsequenz der Wissensbasis ist, müssen wir den Satz von Herbrand bemühen, um zu behaupten, dass der Algorithmus einen Beweis findet. (Weitere Informationen für den Resolutionsfall finden Sie in *Abschnitt 9.5*.) Gibt es für die Abfrage keine Antwort, kann es sein, dass der Algorithmus in einigen Fällen nicht terminiert. Beinhaltet die Wissensbasis beispielsweise die Peano-Axiome

$$NatNum(0)$$
$$\forall n \quad NatNum(n) \Rightarrow NatNum(S(n))$$

fügt die Vorwärtsverkettung $NatNum(S(0))$, $NatNum(S(S(0)))$, $NatNum(S(S(S(0))))$ usw. ein. Dieses Problem ist im Allgemeinen nicht zu vermeiden. Wie bei der allgemeinen Logik erster Stufe ist die logische Konsequenz bei definiten Klauseln semientscheidbar.

9.3.3 Effiziente Vorwärtsverkettung

Der in Abbildung 9.3 gezeigte Algorithmus für die Vorwärtsverkettung soll Ihnen helfen, das Konzept zu verstehen, und ist nicht auf die Effizienz der Operation ausgelegt. Es gibt drei mögliche Ursachen für die Komplexität. Erstens, die „innere Schleife“ des Algorithmus sucht alle möglichen Unifikatoren, sodass die Prämisse einer Regel mit einer geeigneten Menge von Fakten in der Wissensbasis unifiziert. Man spricht häufig von einem **Mustervergleich** (**Pattern Matching**), was sehr aufwändig sein kann. Zweitens, der Algorithmus überprüft jede Regel bei jeder Iteration erneut, um festzustellen, ob ihre Prämissen erfüllt sind, selbst wenn der Wissensbasis bei jeder Iteration nur sehr wenige Einträge hinzugefügt werden. Und drittens kann der Algorithmus viele Fakten erzeugen, die für das Ziel nicht relevant sind. Wir werden jede dieser Ursachen nacheinander betrachten.

Regeln mit bekannten Fakten vergleichen

Das Problem, die Prämisse einer Regel mit den Fakten in der Wissensbasis zu vergleichen, scheint recht einfach zu sein. Angenommen, wir wollen die folgende Regel anwenden:

$$Rakete(x) \Rightarrow Waffe(x).$$

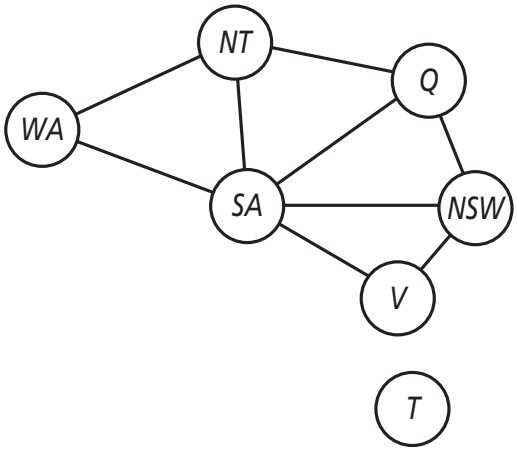
Wir müssen alle Fakten finden, die mit $Rakete(x)$ unifizieren; in einer sinnvoll indizierten Wissensbasis kann dies in konstanter Zeit pro Fakt erfolgen. Betrachten Sie jetzt die folgende Regel:

$$Rakete(x) \wedge Besitzt(Nono, x) \Rightarrow Verkauft(West, x, Nono).$$

Wieder können wir alle Objekte, die Nono besitzt, in konstanter Zeit pro Objekt finden; anschließend könnten wir für jedes Objekt überprüfen, ob es sich um eine Rakete handelt. Wenn die Wissensbasis viele Objekte enthält, die Nono besitzt, aber sehr wenige Raketen, wäre es jedoch besser, zuerst alle Raketen zu suchen und dann zu überprüfen, ob sie Nono gehören. Dies ist das Problem der **Reihenfolge von Konjunkten**: eine Reihenfolge zu finden, um die Konjunkte der Regelprämisse zu lösen, sodass die Gesamtkosten minimiert werden. Es stellt sich heraus, dass die Ermittlung einer optimalen Reihenfolge NP-hart ist, aber es gibt gute Heuristiken. Beispielsweise würde die **MRV-Heuristik** (Minimum Remaining Values, Minimum an verbleibenden Werten), die in *Kapitel 6* für CSPs verwendet wurde, vorschlagen, die Konjunkte so zu sortieren, dass zuerst nach Raketen gesucht wird, wenn es weniger Raketen als von Nono besessene Objekte gibt.

Tipp

Die Verbindung zwischen Pattern Matching und CSP ist sehr eng. Wir können jedes Konjunkt als Beschränkung der darin enthaltenen Variablen betrachten – z.B. ist $Rakete(x)$ eine unäre Beschränkung für x. Wenn wir diese Idee fortführen, *können wir jedes CSP mit endlicher Domäne als einzelne definite Klausel zusammen mit einigen ihr zugeordneten Grundfakten ausdrücken*. Betrachten Sie das Problem der Karteneinfärbung aus Abbildung 6.1, das in ▶ Abbildung 9.5(a) noch einmal gezeigt ist. ▶ Abbildung 9.5(b) zeigt eine äquivalente Formulierung als einzelne definite Klausel. Offensichtlich kann der Schluss *Colorable*() nur dann abgeleitet werden, wenn das CSP eine Lösung hat. Weil CSPs im Allgemeinen 3-SAT-Probleme als Sonderfälle beinhalten, können wir schließen, *dass der Vergleich einer definiten Klausel mit einer Menge von Fakten NP-hart ist*.



a

$$Diff(wa, nt) \wedge Diff(wa, sa) \wedge Diff(nt, q) \wedge Diff(nt, sa) \wedge Diff(q, nsw) \wedge Diff(q, sa) \wedge Diff(nsw, v) \wedge Diff(nsw, sa) \wedge Diff(v, sa) \Rightarrow Colorable()$$

$$Diff(Rot, Blau) Diff(Rot, Grün)$$
$$Diff(Grün, Rot) Diff(Grün, Blau)$$
$$Diff(Blau, Rot) Diff(Blau, Grün)$$

b

Abbildung 9.5: (a) Constraint-Graph für die Einfärbung der Landkarte von Australien. (b) Das CSP für das Karteneinfärbungsproblem, ausgedrückt als einzelne definite Klausel. Jeder Kartenbereich wird als Variable dargestellt, deren Wert eine der Konstanten *Rot*, *Grün* oder *Blau* sein kann.

Es mag etwas deprimierend erscheinen, dass die Vorwärtsverkettung ein NP-hartes Vergleichsproblem in seiner inneren Schleife aufweist. Es gibt jedoch drei Möglichkeiten, die uns aufmuntern könnten:

- Wir können uns daran erinnern, dass die meisten Regeln in echten Wissensbasen klein und einfach sind (wie die Regeln in unserem Verbrecherbeispiel) und nicht groß und kompliziert (wie die CSP-Formulierung in Abbildung 9.5). In der Datenbankwelt wird häufig vorausgesetzt, dass sowohl die Größe der Regeln als auch die Stelligkeit der Prädikate durch eine Konstante begrenzt ist und dass man sich nur über die **Datenkomplexität** Gedanken machen muss – d.h. die Komplexität der Inferenz als Funktion der Anzahl der Grundfakten in der Datenbank.
- Wir können Subklassen von Regeln einrichten, für die der Vergleich effizient ist. Im Wesentlichen kann jede Datalog-Klausel als Definition eines CSP betrachtet werden; der Vergleich ist also dann behandelbar, wenn das entsprechende CSP behandelbar ist. *Kapitel 6* beschreibt mehrere behandelbare Familien von CPSs. Wenn zum Beispiel der Constraint-Graph (der Graph, dessen Knoten Variablen sind und dessen Kanten die Beschränkungen darstellen) einen Baum bildet, kann das CSP in linearer Zeit gelöst werden. Genau das Gleiche gilt für den Regelvergleich. Wenn wir beispielsweise Südaustralien aus der Karte in Abbildung 9.5 entfernen, erhalten wir die resultierende Klausel

  $$Diff(wa, nt) \wedge Diff(nt, q) \wedge Diff(q, nsw) \wedge Diff(nsw, v) \Rightarrow Colorable().$$

  Das entspricht dem in Abbildung 6.12 gezeigten reduzierten CSP. Algorithmen für die Lösung baumstrukturierter CSPs können direkt auf das Problem des Regelvergleiches angewendet werden.
- Wir können probieren, redundante Regelvergleichsversuche im Algorithmus für die Vorwärtsverkettung zu eliminieren, was Thema des nächsten Abschnittes sein wird.

Inkrementelle Vorwärtsverkettung

Als wir die Vorwärtsverkettung anhand des Verbrecherbeispiels gezeigt haben, haben wir geschummelt: Insbesondere ließen wir Regelvergleiche weg, die der in Abbildung 9.3 gezeigte Algorithmus ausführt. In der zweiten Iteration etwa vergleicht die Regel

$$Rakete(x) \Rightarrow Waffe(x)$$

Tipp

(erneut) mit $Rakete(M_1)$ und natürlich ist der Schluss $Waffe(M_1)$ bereits bekannt; deshalb passiert nichts. Solche redundanten Regelvergleiche können vermieden werden, wenn wir die folgende Beobachtung berücksichtigen: *Jeder neue Fakt, der während der Iteration t abgeleitet wird, muss von mindestens einem neuen Fakt abgeleitet sein, der während der Iteration* $t - 1$ *abgeleitet wurde*. Das trifft zu, weil jede Inferenz, die keinen neuen Fakt aus Iteration $t - 1$ zieht, dies bereits in Iteration $t - 1$ getan haben könnte.

Diese Beobachtung führt zu einem Algorithmus für eine inkrementelle Vorwärtsverkettung, wobei wir während der Iteration t eine Regel nur dann überprüfen, wenn ihre Prämisse ein Konjunkt p_i enthält, das mit einem Fakt p_i' unifiziert, der während der Iteration $t - 1$ neu abgeleitet wurde. Der Schritt des Regelvergleiches verändert dann p_i, sodass es mit p_i' übereinstimmt, erlaubt es aber, dass die anderen Konjunkte der Regel mit Fakten aus beliebigen vorhergehenden Iterationen übereinstimmen. Dieser Algorithmus erzeugt genau dieselben Fakten bei jeder Iteration wie der in Abbildung 9.3 gezeigte, ist aber effizienter.

Mit einer geeigneten Indizierung ist es ganz einfach, alle Regeln zu identifizieren, die von jedem beliebigen Fakt ausgelöst werden können, und viele reale Systeme arbeiten tatsächlich in einem „Aktualisierungsmodus“, wo bei jedem neuen (mit TELL veranlassten) Eintrag eines Fakts im System eine Vorwärtsverkettung stattfindet. Die Inferenz durchläuft kaskadenförmig die Regelmenge, bis der Fixpunkt erreicht ist. Anschließend verarbeitet der Prozess den nächsten neuen Fakt.

Normalerweise wird nur ein kleiner Teil der Regeln in der Wissensbasis ausgelöst, wenn ein bestimmter Fakt hinzugefügt wird. Das bedeutet, es erfolgt viel redundante Arbeit beim wiederholten Aufbau partieller Vergleiche, die nicht erfüllte Prämissen aufweisen. Unser Verbrecherbeispiel ist zu klein, um dies zu verdeutlichen, aber beachten Sie, dass während der ersten Iteration eine partielle Übereinstimmung zwischen der Regel

$$Amerikaner(x) \wedge Waffe(y) \wedge Verkauft(x, y, z) \wedge Feindlich(z) \Rightarrow Kriminell(x)$$

und dem Fakt $Amerikaner(West)$ konstruiert wird. Diese partielle Übereinstimmung wird dann verworfen und in der zweiten Iteration (wenn die Regel erfolgreich war) wieder aufgebaut. Es wäre besser, die partiellen Übereinstimmungen aufzubewahren und schrittweise zu vervollständigen, wenn neue Fakten eintreffen, anstatt sie immer zu verwerfen.

Der **Rete**-Algorithmus[3] war der erste Algorithmus, der ernsthaft versuchte, dieses Problem zu lösen. Der Algorithmus leistet eine Vorverarbeitung der Regelmenge in der Wissensbasis, um eine Art Datenflussnetz zu konstruieren, in dem jeder Knoten ein Literal aus einer Regelprämisse ist. Variablenbindungen durchfließen das Netz und werden ausgefiltert, wenn sie nicht mit einem Literal übereinstimmen. Verwenden zwei Literale in einer Regel dieselbe Variable – z.B. $Verkauft(x, y, z) \wedge Feindlich(z)$ im Verbrecherbeispiel –, werden die Bindungen aus jedem Literal durch einen Gleichheitsknoten gefiltert.

3 Rete ist das lateinische Wort für Netz.

Eine Variablenbindung, die einen Knoten für ein n-stelliges Literal erreicht, wie etwa *Verkauft*(x, y, z), muss möglicherweise warten, bis Bindungen für die anderen Variablen eingerichtet sind, bevor der Prozess fortgesetzt werden kann. Der Zustand eines solchen Rete-Netzes berücksichtigt zu jedem Zeitpunkt alle partiellen Übereinstimmungen der Regeln und vermeidet damit viele wiederholte Berechnungen.

Rete-Netzwerke und verschiedene Verbesserungen daran waren eine der Schlüsselkomponenten der sogenannten **Produktionssysteme**, die zu den frühesten Vorwärtsverkettungssystemen gehörten, die ganz allgemein eingesetzt wurden.[4] Das XCON-System (ursprünglich als R1 bezeichnet, McDermott, *1982*) wurde unter Verwendung einer Produktionssystemarchitektur erstellt. XCON enthielt mehrere Tausend Regeln für den Entwurf verschiedener Konfigurationen aus Computerkomponenten der Digital Equipment Corporation. Es war einer der ersten klaren kommerziellen Erfolge auf dem sich entwickelnden Gebiet der Expertensysteme. Viele andere, ähnliche Systeme wurden unter Verwendung derselben zugrunde liegenden Technologie entwickelt, die in der allgemeinen Sprache OPS-5 implementiert wurde.

Produktionssysteme sind auch bekannt in **kognitiven Architekturen** – d.h. Modellen menschlichen Schließens –, wie z.B. ACT (Anderson, 1983) oder SOAR (Laird et al., 1987). In solchen Systemen bildet der „Arbeitsspeicher“ des Modells das Kurzzeitgedächtnis des Menschen nach und die Produktionen sind Teil des Langzeitgedächtnisses. Bei jedem Operationszyklus werden Produktionen mit dem Arbeitsspeicher verglichen, in dem die Fakten enthalten sind. Eine Produktion, deren Bedingungen erfüllt sind, kann Fakten im Arbeitsspeicher hinzufügen oder daraus entfernen. Im Gegensatz zur typischen Situation in Datenbanken haben Produktionssysteme häufig viele Regeln und relativ wenige Fakten. Mit einer geeignet optimierten Vergleichstechnologie können einige moderne Systeme mit über mehreren zehn Millionen Regeln in Echtzeit arbeiten.

Irrelevante Fakten

Die letzte Ursache für Ineffizienzen bei der Vorwärtsverkettung scheint charakteristisch für den Ansatz zu sein und tritt auch im aussagenlogischen Kontext auf. Die Vorwärtsverkettung trifft alle erlaubten Inferenzen basierend auf den bekannten Fakten, *selbst wenn diese irrelevant für das vorliegende Ziel sind*. In unserem Verbrecherbeispiel gab es keine Regeln, die irrelevante Schlüsse ziehen konnten, das Fehlen der Richtungsorientierung war also kein Problem. In anderen Fällen (d.h. wenn viele Regeln die Essgewohnheiten von Amerikanern und die Preise von Raketen beschreiben) erzeugt FOL-FC-ASK viele irrelevante Schlüsse.

Eine Möglichkeit, irrelevante Schlüsse zu vermeiden, ist die Verwendung der Rückwärtsverkettung, die in *Abschnitt 9.4* beschrieben ist. Eine weitere Lösung wäre, die Vorwärtsverkettung auf eine ausgewählte Untermenge an Regeln zu begrenzen, wie es in PL-FC-ENTAILS? (*Abschnitt 7.5.4*) der Fall ist. Ein dritter Ansatz hat sich im Bereich der **deduktiven Datenbanken** herausgebildet. Hier handelt es sich um sehr große Datenbanken, ähnlich relationalen Datenbanken, aber mit Verwendung der Vorwärtsverkettung als Standardwerkzeug für Inferenz anstelle von SQL-Abfragen. Die Idee dabei ist, die Regelmenge mithilfe der Informationen vom Ziel neu zu schreiben, sodass nur relevante Variablenbindungen – diejenigen, die zu einer sogenannten **magischen Menge**

4 Das Wort **Produktion** in **Produktionssystem** bezeichnet eine Bedingung/Aktion-Regel.

gehören – bei der Vorwärtsinferenz berücksichtigt werden. Ist das Ziel beispielsweise *Kriminell*(*West*), wird die Regel, die *Kriminell*(*x*) schließt, so umgeschrieben, dass sie ein zusätzliches Konjunkt enthält, das den Wert von *x* beschränkt:

$$Magisch(x) \wedge Amerikaner(x) \wedge Waffe(y) \wedge Verkauft(x, y, z) \wedge Feindlich(z) \Rightarrow Kriminell(x).$$

Der Fakt *Magisch*(*West*) wird auch der Wissensbasis hinzugefügt. Auf diese Weise wird Colonel West beim Vorwärts-Inferenzprozess erkannt, selbst wenn die Wissensbasis Fakten über Millionen von Amerikanern enthält. Der vollständige Prozess zur Definition magischer Mengen und das Umschreiben der Wissensbasis ist zu kompliziert, als dass er hier genauer beschrieben werden könnte, aber die grundlegende Idee dabei ist, eine Art „generischer" Rückwärtsinferenz vom Ziel aus durchzuführen, um festzustellen, welche Variablenbindungen beschränkt werden müssen. Der Ansatz mit der magischen Menge kann auch als eine Art Hybridform zwischen der Vorwärtsinferenz und der Rückwärtsvorverarbeitung betrachtet werden.

9.4 Rückwärtsverkettung

Die zweite große Familie logischer Inferenzalgorithmen verwendet den Ansatz der **Rückwärtsverkettung**, der in *Abschnitt 7.5* vorgestellt wurde. Diese Algorithmen arbeiten rückwärts vom Ziel aus und verketten die Regeln, um bekannte Fakten zu finden, die den Beweis unterstützen. Wir beschreiben den grundlegenden Algorithmus und dann, wie er in der **Logikprogrammierung**, eine der gebräuchlichsten Formen automatisierten Schließens, verwendet wird. Außerdem sehen wir, dass die Rückwärtsverkettung einige Nachteile im Vergleich zur Vorwärtsverkettung hat, und wir suchen nach Möglichkeiten, sie zu kompensieren. Schließlich betrachten wir noch die enge Verbindung zwischen Logikprogrammierung und CSPs.

9.4.1 Ein Algorithmus für die Rückwärtsverkettung

▶ Abbildung 9.6 zeigt einen Algorithmus für definite Klauseln. FOL-BC-Ask(*KB*, *goal*) wird bewiesen, wenn die Wissensbasis eine Klausel der Form *lhs* ⇒ *goal* enthält, wobei *lhs* (für left-hand side, linksseitig) eine Liste von Konjunkten ist. Ein atomarer Fakt wie zum Beispiel *Amerikaner*(*West*) wird als Klausel betrachtet, deren *lhs* die leere Liste ist. Jetzt ließe sich eine Abfrage, die Variablen enthält, auf mehrere Arten beweisen. Zum Beispiel könnte die Abfrage *Person*(*x*) mit der Substitution {*x*/*John*} wie auch mit {*x*/*Richard*} bewiesen werden. Wir implementieren also FOL-BC-Ask als **Generator** – eine Funktion, die mehrfach zurückkehrt und dabei jeweils ein mögliches Ergebnis liefert.

```
function FOL-BC-Ask(KB, query) returns einen Generator von Substitutionen
  return FOL-BC-Or(KB, query, { })

---

generator FOL-BC-Or(KB, goal, θ) yields eine Substitution
  for each Regel (lhs ⇒ rhs) in Fetch-Rules-For-Goal(KB, goal) do
    (lhs, rhs) ← Standardize-Variables((lhs, rhs))
    for each θ' in FOL-BC-And(KB, lhs, Unify(rhs, goal, θ)) do
      yield θ'

---
```

```
generator FOL-BC-AND(KB, goals, θ) yields eine Substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else do
    first, rest ← FIRST(goals), REST(goals)
    for each θ' in FOL-BC-OR(KB, SUBST(θ, first), θ) do
      for each θ'' in FOL-BC-AND(KB, rest, θ') do
        yield θ''
```

Abbildung 9.6: Ein einfacher Algorithmus zur Rückwärtsverkettung für Wissensbasen erster Stufe.

Rückwärtsverkettung ist eine Art AND-OR-Suche – der OR-Teil, weil die Zielabfrage durch eine beliebige Regel in der Wissensbasis bewiesen werden kann, und der AND-Teil, weil alle Konjunkten in *lhs* einer Klausel bewiesen werden müssen. FOL-BC-OR ruft alle Klauseln ab, die mit dem Ziel unifiziert werden können. Dabei werden die Variablen in der Klausel auf vollkommen neue Variablen standardisiert und wenn sich dann *rhs* der Klausel tatsächlich mit dem Ziel unifizieren lässt, wird jedes Konjunkt in *lhs* mithilfe von FOL-BC-AND bewiesen. Diese Funktion beweist wiederum nacheinander jedes der Konjunkte und verfolgt dabei die akkumulierte Substitution. ▶ Abbildung 9.7 zeigt den Beweisbaum für die Herleitung von *Kriminell*(*West*) aus den Sätzen (9.3) bis (9.10).

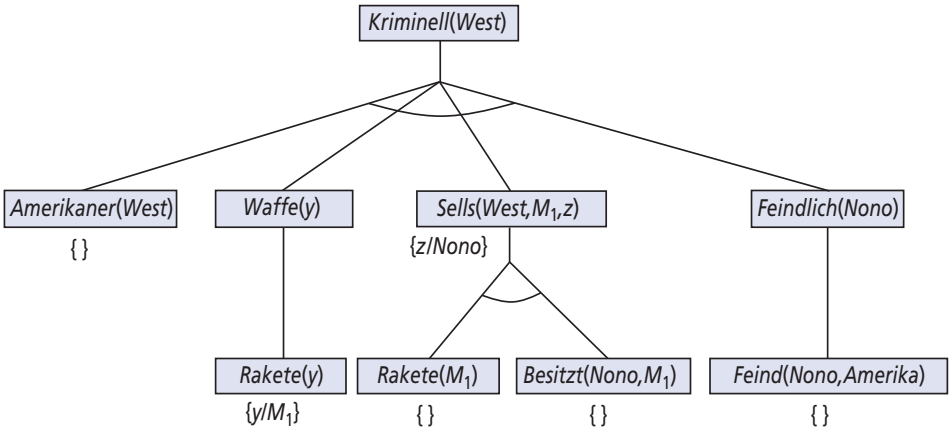


Abbildung 9.7: Durch Rückwärtsverkettung erzeugter Beweisbaum, um zu beweisen, dass West ein Krimineller ist. Der Baum sollte entsprechend einer Tiefensuche von links nach rechts gelesen werden. Um *Kriminell(West)* beweisen zu können, müssen wir die vier Konjunkten darunter beweisen. Einige davon befinden sich in der Wissensbasis, andere müssen weiter rückwärts verkettet werden. Neben dem entsprechenden Unterziel sind die Bindungen für jede erfolgreiche Unifikation dargestellt. Nachdem ein Unterziel in einer Konjunktion erfolgreich war, wird seine Substitution auf die nachfolgenden Unterziele angewendet. Wenn also FOL-BC-ASK zum letzten Konjunkt gelangt, ist im ursprünglichen *Feindlich(z)* bereits *z* an *Nono* gebunden.

Die Rückwärtsverkettung, wie wir sie geschrieben haben, ist offensichtlich ein Tiefensuchalgorithmus. Das bedeutet, seine Speicheranforderungen sind linear zur Größe des Beweises (wobei wir hier den Speicher für die Sammlung der Lösungen vernachlässigen wollen). Außerdem bedeutet es, dass die Rückwärtsverkettung (anders als die Vorwärtsverkettung) unter Problemen mit wiederholten Zuständen und Unvollständigkeit leidet. Wir werden auf diese Probleme und mögliche Lösungen noch genauer eingehen, aber zuerst betrachten wir, wie die Rückwärtsverkettung in Logikprogrammiersystemen verwendet wird.

9.4.2 Logikprogrammierung

Logikprogrammierung ist eine Technologie, die der Verkörperung des in *Kapitel 7* beschriebenen deklarativen Ideals recht nahe kommt: dass Systeme aufgebaut werden sollen, indem man Wissen in einer formalen Sprache ausdrückt, und dass Probleme gelöst werden sollen, indem man Inferenzprozesse für dieses Wissen ausführt. Das Ideal ist in der Gleichung von Robert Kowalski zusammengefasst:

Algorithmus = *Logik* + *Steuerung*

Prolog ist die bei weitem gebräuchlichste Logikprogrammiersprache. Sie wird hauptsächlich als Sprache für die schnelle Entwicklung von Prototypen verwendet sowie für Aufgaben der Symbolmanipulation, wie etwa die Entwicklung von Compilern (Van Roy, 1990) und dem Parsen natürlicher Sprache (Pereira und Warren, 1980). Viele Expertensysteme für rechtliche, medizinische, finanztechnische oder andere Domänen wurden in Prolog geschrieben.

Prolog-Programme sind Mengen definiter Klauseln, welche in einer Notation geschrieben werden, die sich etwas von der standardmäßigen Logik erster Stufe unterscheidet. Prolog verwendet Großbuchstaben für Variablen und Kleinbuchstaben für Konstanten – entgegengesetzt zu unserer Konvention für Logik. Kommas trennen Konjunkte in einer Klausel und die Klausel wird „rückwärts“ in Bezug auf die bisher verwendete Schreibweise notiert; anstatt $A \wedge B \Rightarrow C$ zu schreiben, wird in Prolog `C :- A, B` notiert. Ein typisches Beispiel sieht so aus:

```
Kriminell(X) :- amerikaner(X), waffe(y), verkauft(X, Y, Z), feindlich(Z)
```

Die Notation `[E|L]` kennzeichnet eine Liste, deren erstes Element `E` und deren Rest `L` ist. Das folgende Beispiel zeigt ein Prolog-Programm für `append(X, Y, Z)`, das erfolgreich ist, wenn sich die Liste `Z` durch Anfügen der Listen `X` und `Y` ergibt:

```
append([], Y, Y).
append([A|X], Y, [A|Z]) :- append(X, Y, Z)
```

In natürlicher Sprache können wir diese Klauseln lesen als (1) Verketten einer leeren Liste mit einer Liste `Y` erzeugt dieselbe Liste `Y` und (2) `[A|Z]` ist das Ergebnis, wenn man `[A|X]` mit `Y` verkettet, vorausgesetzt, `Z` ist das Ergebnis aus dem Anfügen von `X` zu `Y`. In den meisten höheren Programmiersprachen lässt sich eine ähnliche rekursive Funktion schreiben, die beschreibt, wie zwei Listen verkettet werden. Die Prolog-Definition ist aber tatsächlich sehr viel leistungsfähiger, da sie eine *Relation* beschreibt, die zwischen ihren Argumenten besteht, und nicht nur eine *Funktion*, die aus zwei Argumenten berechnet wird. Beispielsweise können wir die Abfrage `append(X, Y, [1,2])` stellen: Welche beiden Listen können angefügt werden, damit sich `[1, 2]` ergibt? Wir erhalten die folgenden Lösungen:

```
X=[]            Y=[1, 2];
X=[1]           Y=[2];
X=[1,2]         Y=[]
```

Die Ausführung von Prolog-Programmen erfolgt über eine Tiefensuche-Rückwärtsverkettung, wobei Klauseln in der Reihenfolge ausprobiert werden, in der sie in die Wissensbasis geschrieben wurden. Einige Aspekte von Prolog liegen außerhalb der standardmäßigen logischen Inferenz:

- Prolog verwendet die Datenbanksemantik gemäß *Abschnitt 8.2.8* und nicht die Semantik der Logik erster Stufe. Dies wird deutlich in der Behandlung von Gleichheit und Negation (siehe Abschnitt 9.4.5).
- Es gibt mehrere eingebaute Funktionen für Arithmetik. Literale, die diese Funktionssymbole verwenden, werden „bewiesen“, indem Code ausgeführt wird, anstatt weitere Inferenzen vorzunehmen. Zum Beispiel ist das Ziel „`X is 4+3`“ erfolgreich und bindet `X` an 7. Andererseits schlägt das Ziel „`5 is X+Y`“ fehl, weil die eingebauten Funktionen keine beliebigen Gleichungen lösen können.[5]
- Es gibt eingebaute Prädikate, die bei der Ausführung Nebeneffekte aufweisen. Unter anderem handelt es sich dabei um Eingabe/Ausgabe-Prädikate sowie die Prädikate `assert/retract` für die Bearbeitung der Wissensbasis. Für diese Prädikate gibt es kein Gegenstück in der Logik und sie können verwirrende Effekte verursachen – wenn beispielsweise Fakten in einem Zweig des Beweisbaumes behauptet werden, der irgendwann fehlschlägt.
- Der **Occur Check** wird im Unifikationsalgorithmus von Prolog ausgelassen. Somit können auch einige inkorrekte Inferenzen gezogen werden. In der Praxis ist das aber fast nie ein Problem.
- Prolog verwendet Tiefensuche-Rückwärtsverkettung ohne Überprüfungen auf unendliche Rekursion. Dadurch ist die Ausführung sehr schnell, wenn die richtige Menge von Axiomen gegeben ist, aber unvollständig, wenn es sich um die falschen Axiome handelt.

Das Design von Prolog verkörpert einen Kompromiss zwischen Deklarativität und Ausführungseffizienz – unter Berücksichtigung dessen, was man zu dem Zeitpunkt, als Prolog entwickelt wurde, unter Effizienz verstanden hat.

9.4.3 Effiziente Implementierung von Logikprogrammen

Die Ausführung eines Prolog-Programms kann in zwei Modi erfolgen: interpretiert und kompiliert. Bei der Interpretation wird im Wesentlichen der FOL-BC-Ask-Algorithmus gemäß Abbildung 9.6 ausgeführt, wobei das Programm als Wissensbasis dient. Wir sagen „im Wesentlichen“, weil Prolog-Interpreter eine Vielzahl von Verbesserungen beinhalten, um eine maximale Geschwindigkeit zu erzielen. Hier betrachten wir nur zwei davon.

Erstens musste unsere Implementierung die Iteration über möglichen Ergebnissen, die von den einzelnen Subfunktionen generiert wurden, explizit verwalten. Prolog-Interpreter verfügen über eine globale Datenstruktur, einen Stack von **Auswahlpunkten**, um die verschiedenen Möglichkeiten zu verfolgen, die wir in FOL-BC-Or betrachtet haben. Dieser globale Stack ist effizienter und er erleichtert das Debugging, da der Debugger im Stack auf- und abgehen kann.

Zweitens verbringt unsere einfache Implementierung von FOL-BC-Ask viel Zeit damit, Substitutionen zu erzeugen. Anstatt Substitutionen explizit zu konstruieren, besitzt Prolog Variablen, die sich ihre aktuelle Bindung merken. Zu jedem Zeitpunkt ist jede Variable im Programm entweder ungebunden oder an einen bestimmten Wert gebunden. Insgesamt definieren diese Variablen und Werte implizit die Substitution für den aktuellen

5 Wenn Peano-Axiome bereitgestellt werden, lassen sich derartige Ziele mithilfe von Inferenz innerhalb eines Prolog-Programms lösen.

Beweiszweig. Eine Erweiterung des Pfades kann nur neue Variablenbindungen hinzufügen, denn ein Versuch, eine andere Bindung für eine bereits gebundene Variable hinzuzufügen, führt zu einem Fehler der Unifikation. Wenn ein Pfad in der Suche fehlschlägt, kehrt Prolog zu einem früheren Auswahlpunkt zurück und muss dann möglicherweise die Bindungen einiger Variablen aufheben. Dazu sammelt es alle bereits gebundenen Variablen in einem Stack, der auch als **Trail** bezeichnet wird. Wenn UNIFY-VAR eine neue Variable bindet, wird die Variable auf dem Trail abgelegt. Schlägt ein Ziel fehl und ist eine Rückkehr zu einem früheren Auswahlpunkt erforderlich, wird die Bindung der betreffenden Variablen aufgehoben, wenn sie vom Trail entfernt werden.

Selbst die effizientesten Prolog-Interpreter brauchen wegen Indexsuche, Unifikation und Aufbau des rekursiven Aufrufstacks mehrere Tausend Maschinenbefehle pro Inferenzschritt. Der Interpreter verhält sich dabei immer so, als hätte er das Programm nie zuvor gesehen; er muss beispielsweise Klauseln *suchen*, die mit dem Ziel übereinstimmen. Ein kompiliertes Prolog-Programm dagegen ist eine Inferenzprozedur für eine spezifische Menge an Klauseln; deshalb *weiß* es, welche Klauseln mit dem Ziel übereinstimmen. Prolog erzeugt im Grunde genommen einen Miniatur-Theorembeweiser für jedes einzelne Prädikat und eliminiert dabei einen Großteil des Zusatzaufwandes der Interpretation. Außerdem ist es möglich, die Unifikationsroutine für jeden einzelnen Aufruf **offen zu kodieren** und damit eine explizite Analyse der Termstruktur zu vermeiden. (Weitere Informationen über die offen kodierte Unifikation finden Sie bei Warren et al., 1977).

Die Befehlssätze heutiger Computer weisen eine unzulängliche Übereinstimmung mit der Semantik von Prolog auf. Deshalb kompilieren die meisten Prolog-Compiler in eine Zwischensprache und nicht direkt in die Maschinensprache. Die bekannteste Zwischensprache ist WAM (Warren Abstract Machine), benannt nach David H.D. Warren, der mit zur Implementierung des ersten Prolog-Compilers beigetragen hat. Die WAM ist ein abstrakter Befehlssatz, der für Prolog geeignet ist und entweder interpretiert oder in Maschinensprache übersetzt werden kann. Andere Compiler übersetzen Prolog in eine höhere Sprache, wie etwa Lisp oder C, und verwenden dann den Compiler für diese Sprache, um das Ganze in Maschinencode zu übersetzen. Zum Beispiel lässt sich die Definition des Prädikates `Append` in den in ▶ Abbildung 9.8 gezeigten Code kompilieren. Es gibt dabei mehrere interessante Aspekte:

- Anstatt die Wissensbasis nach `Append`-Klauseln durchsuchen zu müssen, werden die Klauseln zu einer Prozedur. Um die Inferenzen auszuführen, wird einfach die Prozedur aufgerufen.
- Wie bereits beschrieben, werden die aktuellen Variablenbindungen auf einem Trail abgelegt. Der erste Schritt der Prozedur speichert den aktuellen Zustand des Trails, sodass er durch RESET-TRAIL wiederhergestellt werden kann, falls die erste Klausel fehlschlägt. Damit werden alle Bindungen aufgehoben, die der erste Aufruf von UNIFY erzeugt hat.
- Der komplizierteste Teil ist die Verwendung von **Fortsetzungen** (*continuations*), um Auswahlpunkte zu implementieren. Sie können sich unter einer solchen Fortsetzung vorstellen, dass eine Prozedur und eine Argumentliste zusammengepackt werden, die in ihrer Kombination definieren, was als Nächstes passieren soll, wenn das aktuelle Ziel erfolgreich ist. Es ist nicht ausreichend, von einer Prozedur wie APPEND zurückzukehren, wenn das Ziel erfolgreich ist, weil es auf unterschiedliche Arten erfolgreich sein kann und jede davon ausgewertet werden muss. Das Fortset-

zungsargument löst dieses Problem, weil es aufgerufen werden kann, wenn das Ziel erfolgreich war. Wenn im APPEND-Code das erste Argument leer ist und das zweite Argument mit dem dritten unifiziert wird, war das APPEND-Prädikat erfolgreich. Anschließend rufen wir die Fortsetzung mit den geeigneten Bindungen auf dem Trail auf, um zu tun, was auch immer als Nächstes zu tun ist. Befindet sich beispielsweise der Aufruf von APPEND auf der obersten Ebene, gibt die Fortsetzung alle Bindungen der Variablen aus.

```
procedure APPEND(ax, y, az, continuation)

  trail ← GLOBAL-TRAIL-POINTER()
  if ax = [ ] und UNIFY(y, az) then CALL(continuation)
  RESET-TRAIL(trail)
  a, x, z ← NEW-VARIABLE(), NEW-VARIABLE(), NEW-VARIABLE()
  if UNIFY(ax, [a|x]) und UNIFY(az, [a|z]) then APPEND(x, y, z, continuation)
```

Abbildung 9.8: Pseudocode, der das Ergebnis der Kompilierung des `Append`-Prädikates darstellt. Die Funktion NEW-VARIABLE gibt eine neue Variable zurück, die sich von allen anderen bisher verwendeten Variablen unterscheidet. Die Prozedur CALL*(continuation)* setzt die Ausführung fort, und zwar mit der angegebenen Fortsetzung.

Vor Warrens Arbeiten zur Kompilierung der Inferenz in Prolog war die Logikprogrammierung zu langsam für den allgemeinen Einsatz. Compiler von Warren und anderen erlaubten es, dass der Prolog-Code Geschwindigkeiten erreichte, die ihn im Hinblick auf zahlreiche Standard-Benchmarks konkurrenzfähig mit C machten (Van Roy, 1990). Natürlich wird es durch die Tatsache, dass man einen Planer oder einen Parser für die natürliche Sprache mit nur ein paar Dutzend Prolog-Zeilen schreiben kann, etwas attraktiver für das Prototyping der meisten kleinen KI-Forschungsprojekte als C.

Auch die Parallelisierung kann wesentliche Beschleunigungen erzielen. Es gibt vor allem zwei Quellen für Parallelität. Die erste davon, die sogenannte **OR-Parallelität**, stammt aus der Möglichkeit, dass ein Ziel mit vielen verschiedenen Klauseln in der Wissensbasis unifiziert wird. Jedes erzeugt einen unabhängigen Zweig im Suchraum, der zu einer potenziellen Lösung führen kann, und all diese Zweige können parallel gelöst werden. Die zweite, die sogenannte **AND-Parallelität**, stammt aus der Möglichkeit, jedes Konjunkt im Rumpf einer Implikation parallel zu lösen. Die AND-Parallelität ist schwieriger zu erzielen, weil Lösungen für die gesamte Konjunktion konsistente Bindungen für alle Variablen erforderlich machen. Jeder konjunktive Zweig muss mit allen anderen Zweigen kommunizieren, um eine globale Lösung sicherzustellen.

9.4.4 Redundante Inferenz und Endlosschleifen

Jetzt wenden wir uns der Achilles-Sehne von Prolog zu: der fehlenden Übereinstimmung zwischen Tiefensuche und Suchbäumen, die wiederholte Zustände und Endlospfade enthalten. Betrachten Sie das folgende Logikprogramm, das feststellt, ob es in einem gerichteten Graphen einen Pfad zwischen zwei Punkten gibt:

```
path(X, Z) :- link(X, Z).
path(X, Z) :- path(X, Y), link(Y, Z).
```

▶ Abbildung 9.9(a) zeigt einen einfachen Dreiknotengraphen, der durch die Fakten `link(a, b)` und `link(b, c)` beschrieben wird. Mit diesem Programm erzeugt die Abfrage

`path(a, c)` den in ▶ Abbildung 9.10(a) gezeigten Beweisbaum. Wenn wir die beiden Klauseln dagegen in der folgenden Reihenfolge angeben:

```
path(X, Z) :- path(X, Y), link(Y, Z).
path(X, Z) :- link(X, Z).
```

folgt Prolog dem in ▶ Abbildung 9.10(b) gezeigten Endlospfad. Prolog ist also **unvollständig** als Theorembeweiser für definite Klauseln – selbst für Datalog-Programme, wie dieses Beispiel zeigt –, weil es für einige Wissensbasen logische Konsequenzen nicht beweisen kann. Beachten Sie, dass die Vorwärtsverkettung nicht unter diesem Problem leidet: Nachdem `path(a, b)`, `path(b, c)` und `path(a, c)` abgeleitet sind, hält die Vorwärtsverkettung an.

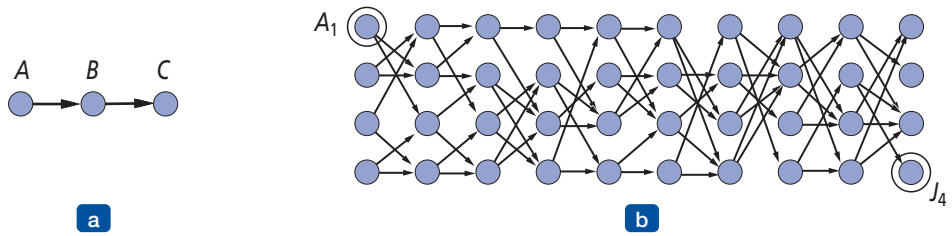


Abbildung 9.9: (a) Die Suche nach einem Pfad von *A* nach *C* kann Prolog in eine Endlosschleife führen. (b) Ein Graph, in dem jeder Knoten mit zwei beliebigen Nachfolgern in der nächsten Ebene verbunden ist. Die Suche nach einem Pfad von A_1 nach J_4 benötigt 877 Inferenzen.

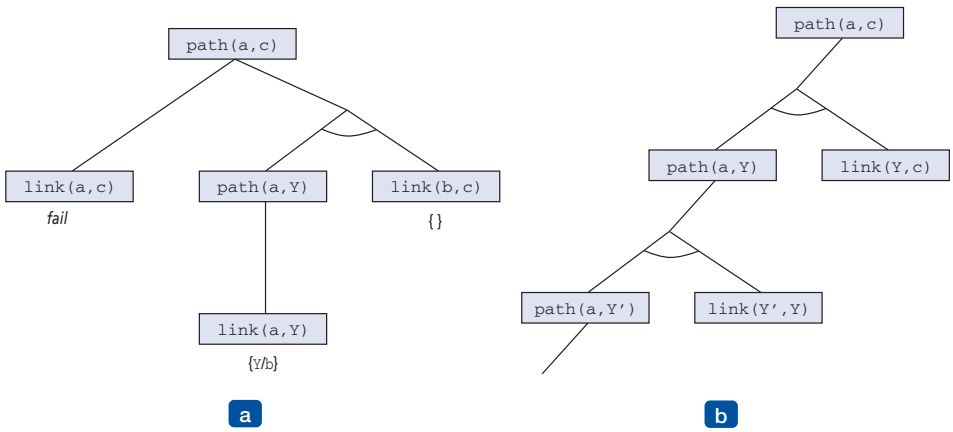


Abbildung 9.10: (a) Beweis, dass es einen Pfad von *A* nach *C* gibt. (b) Endloser Beweisbaum, der erzeugt wird, wenn sich die Klauseln in der „falschen" Reihenfolge befinden.

Die Tiefensuche-Rückwärtsverkettung hat auch Probleme mit redundanten Berechnungen. Sucht man in ▶ Abbildung 9.9(b) beispielsweise einen Pfad von A_1 nach J_4, führt Prolog 877 Inferenzen aus, wobei die meisten alle möglichen Pfade zu Knoten finden, von denen aus das Ziel nicht zu erreichen ist. Das ist vergleichbar mit dem Problem wiederholter Zustände, das in *Kapitel 3* angesprochen wurde. Die Gesamtzahl an Inferenzen kann exponentiell zur Anzahl der erzeugten Grundfakten sein. Wenn wir stattdessen eine Vorwärtsverkettung anwenden, können höchstens n^2`path(X, Y)` Fakten erzeugt werden, die n Knoten verknüpfen. Für das in Abbildung 9.9(b) gezeigte Problem sind nur 62 Inferenzen erforderlich.

Die Vorwärtsverkettung bei Graphensuchproblemen ist ein Beispiel für die **dynamische Programmierung**, wobei die Lösungen von Unterproblemen inkrementell aus den Lösungen für kleinere Unterprobleme konstruiert und zur Vermeidung von wiederholten Berechnungen zwischengespeichert werden. Einen ähnlichen Effekt können wir in einem Rückwärtsverkettungssystem mithilfe der **Memoisation** verwenden – d.h. das Zwischenspeichern von Lösungen für Unterziele, sobald sie gefunden werden, und die Wiederverwendung dieser Lösungen, wenn das Unterziel erneut auftritt, anstatt die vorherige Berechnung zu wiederholen. Dies ist auch der Ansatz, den **tabulierte Logikprogrammiersysteme** verfolgen, die effiziente Speicher- und Lademechanismen nutzen, um die Memoisation zu realisieren. Dabei wird die Zielgerichtetheit der Rückwärtsverkettung mit der dynamischen Programmiereffizienz der Vorwärtsverkettung kombiniert. Außerdem ist die tabulierte Logikprogrammierung vollständig für Datalog-Programme, was bedeutet, dass sich der Programmierer weniger um Endlosschleifen kümmern muss. (Es ist dennoch möglich, mit Prädikaten wie `Vater(X, Y)`, die auf eine möglicherweise unbeschränkte Anzahl von Objekten verweisen, in eine Endlosschleife zu geraten.)

9.4.5 Datenbanksemantik von Prolog

Prolog verwendet Datenbanksemantik, wie sie *Abschnitt 8.2.8* erläutert hat. Die Zusicherung von eindeutigen Namen besagt, dass jede Prolog-Konstante und jeder Grundterm auf ein anderes Objekt verweisen, und die Zusicherung der geschlossenen Welt heißt, dass es sich bei den einzigen Sätzen, die wahr sind, um diejenigen handelt, die durch die Wissensbasis gefolgert werden können. In Prolog gibt es keine Möglichkeit zu behaupten, dass ein Satz falsch ist. Dadurch ist Prolog weniger ausdrucksstark als Logik erster Stufe, doch gehört dies dazu, was Prolog effizienter und prägnanter macht. Sehen Sie sich die folgenden Prolog-Behauptungen über einige Kursangebote an:

$$Course(CS, 101), Course(CS, 102), Course(CS, 106), Course(EE, 101). \qquad (9.11)$$

Unter Annahme eindeutiger Namen sind *CS* und *EE* unterschiedlich (genau wie 101, 102 und 106) – und demnach gibt es vier unterschiedliche Kurse. Wird die Weltabgeschlossenheit angenommen, existieren keine anderen Kurse, sodass es genau vier Kurse gibt. Doch wenn diese Behauptungen in Logik erster Stufe (First Order Logic, FOL) statt in Prolog erfolgen, würden wir lediglich aussagen, dass die Anzahl der Kurse zwischen 1 und ∞ liegt. Das hängt damit zusammen, dass die Zusicherungen (in FOL) weder die Möglichkeit leugnen, dass andere nicht erwähnte Kurse ebenso angeboten werden, noch besagen sie, dass sich die erwähnten Kurse voneinander unterscheiden. Möchten wir Gleichung (9.11) in Logik erster Stufe übersetzen, würden wir dies erhalten:

$$\begin{aligned} &Course(d, n) \Leftrightarrow (d = CS \wedge n = 101) \vee (d = CS \wedge n = 102) \\ &\vee (d = CS \wedge n = 106) \vee (d = EE \wedge n = 101). \end{aligned} \qquad (9.12)$$

Dies wird als **Abschluss** von Gleichung (9.11) bezeichnet. Er drückt in Logik erster Stufe das Konzept aus, dass es höchstens vier Kurse gibt. Um in Logik erster Stufe den Gedanken auszudrücken, dass es wenigstens vier Kurse gibt, müssen wir den Abschluss des Gleichheitsprädikates schreiben:

$$\begin{aligned} &x{=}y \Leftrightarrow (x = CS \wedge y = CS) \vee (x = EE \wedge y = EE) \vee (x = 101 \wedge y = 101) \\ &\vee (x = 102 \wedge y = 102) \vee (x = 106 \wedge y = 106). \end{aligned}$$

Der Abschluss ist nützlich, um die Datenbanksemantik zu verstehen, doch wenn sich Ihr Problem mit Datenbanksemantik beschreiben lässt, ist es in der Praxis effizienter, mit Prolog oder einem anderen Datenbanksemantiksystem zu schlussfolgern, anstatt nach Logik erster Stufe zu übersetzen und mit einem vollständigen Theorembeweiser in Logik erster Stufe zu schließen.

9.4.6 Logikprogrammierung mit Randbedingungen

In unserer Beschreibung der Vorwärtsverkettung (*Abschnitt 9.3*) haben wir gezeigt, wie Probleme unter Rand- und Nebenbedingungen (CSPs) als definite Klauseln kodiert werden können. Standard-Prolog löst solche Probleme auf genau dieselbe Weise wie der Backtracking-Algorithmus in Abbildung 6.5.

Weil das Backtracking die Domänen der Variablen auflistet, funktioniert es nur für CSPs mit **endlichen Domänen**. In Prolog-Denkweise ausgedrückt, muss es für jedes Ziel mit ungebundenen Variablen eine endliche Anzahl von Lösungen geben. (Zum Beispiel hat das Ziel `diff(Q, SA)`, das besagt, dass Queensland und Südaustralien unterschiedliche Farben haben müssen, sechs Lösungen, wenn drei Farben unterstützt werden.) CSPs mit unendlichen Domänen – z.B. mit ganzzahligen oder reellwertigen Variablen – bedingen ganz andere Algorithmen, wie beispielsweise die Propagierung von Grenzen oder eine lineare Programmierung.

Sehen Sie sich das folgende Beispiel an. Wir definieren `triangle(X, Y, Z)` als Prädikat, das gilt, wenn die drei Argumente Zahlen sind, die die Dreiecksungleichung erfüllen:

```
triangle(X, Y, Z) :-
    X>0, Y>0, Z>0, X+Y>=Z, Y+Z>=X, X+Z>=Y
```

Wenn wir an Prolog die Abfrage `triangle(3, 4, 5)` richten, funktioniert das problemlos. Fragen wir dagegen `triangle(3, 4, Z)`, wird keine Lösung gefunden, weil das Unterziel `Z>=0` von Prolog nicht verarbeitet werden kann; wir können nicht einen ungebundenen Wert mit 0 vergleichen.

Die **Constraint-Logikprogrammierung** (CLP) erlaubt es, dass Variablen *beschränkt* statt *gebunden* werden. Eine Lösung für ein Constraint-Logikprogramm ist die spezifischste Menge der Bedingungen für die Abfragevariablen, die aus der Wissensbasis abgeleitet werden können. Die Lösung für die Abfrage `triangle(3, 4, Z)` beispielsweise ist die Bedingung `7 >= Z >= 1`. Standard-Logikprogramme sind nur ein Sonderfall von CLP, wobei die Lösungsbedingungen Gleichheitsbedingungen sein müssen – d.h. Bindungen.

CLP-Systeme beinhalten zahlreiche bedingungslösende Algorithmen für die in der Sprache erlaubten Bedingungen. Ein System beispielsweise, das lineare Ungleichheiten für reellwertige Variablen erlaubt, könnte einen linearen Programmieralgorithmus für die Lösung dieser Bedingungen enthalten. CLP-Systeme übernehmen auch einen viel flexibleren Ansatz für die Lösung von Standardabfragen der Logikprogrammierung. Anstatt etwa eine Tiefensuche und Backtracking von links nach rechts durchzuführen, könnten sie einen der effizienteren Algorithmen aus *Kapitel 6* verwenden, wie etwa die heuristische Konjunktsortierung, Backjumping, Schnittmengenkonditionie-

rung usw. CLP-Systeme kombinieren deshalb Elemente von Algorithmen für Rand- und Nebenbedingungen, Logikprogrammierung und deduktive Datenbanken.

Es wurden mehrere Systeme definiert, die dem Programmierer mehr Kontrolle über die Suchreihenfolge für die Inferenz erlauben. So ermöglicht die Sprache MRS (Genesereth und Smith, 1981; Russell, 1985) dem Programmierer, **Metaregeln** zu schreiben, um festzustellen, welche Konjunkte als Erstes ausprobiert werden sollen. Der Benutzer könnte eine Regel schreiben, die besagt, dass das Ziel mit den wenigsten Variablen zuerst ausprobiert werden soll, oder er könnte domänenspezifische Regeln für bestimmte Prädikate schreiben.

9.5 Resolution

Die letzte unserer drei Familien von Logiksystemen basiert auf der **Resolution**. In *Abschnitt 7.5* haben wir gesehen, dass die aussagenlogische Resolution mithilfe der Widerlegung eine vollständige Inferenzprozedur für die Aussagenlogik ist. In diesem Abschnitt beschreiben wir, wie die Resolution auf die Logik erster Stufe erweitert werden kann.

9.5.1 Konjunktive Normalform für die Logik erster Stufe

Wie im aussagenlogischen Fall verlangt die Resolution für die Logik erster Stufe, dass sich die Sätze in **konjunktiver Normalform** (KNF) befinden – d.h. einer Konjunktion von Klauseln, wobei jede Klausel eine Disjunktion von Literalen ist.[6] Literale können Variablen enthalten, für die man voraussetzt, dass sie allquantifiziert sind. Beispielsweise wird der Satz

$$\forall x \quad Amerikaner(x) \wedge Waffe(y) \wedge Verkauft(x, y, z) \wedge Feindlich(z) \Rightarrow Kriminell(x)$$

in KNF zu

$$\neg Amerikaner(x) \vee \neg Waffe(y) \vee \neg Verkauft(x, y, z) \vee \neg Feindlich(z) \vee Kriminell(x)$$

Tipp

Jeder Satz der Logik erster Stufe kann in einen im Hinblick auf die Inferenz äquivalenten KNF-Satz umgewandelt werden. Insbesondere ist der KNF-Satz nur dann nicht erfüllbar, wenn der ursprüngliche Satz nicht erfüllbar ist. Wir haben also eine Grundlage für die Ausführung von Beweisen durch Widerspruch für die KNF-Sätze.

Die Prozedur für die Umwandlung in KNF ist derjenigen für den aussagenlogischen Fall, den Sie in *Abschnitt 7.5.2* kennengelernt haben, sehr ähnlich. Der wichtigste Unterschied entsteht aus der Notwendigkeit, Existenzquantoren zu eliminieren. Wir zeigen diese Prozedur, indem wir den Satz „Jeder, der alle Tiere liebt, wird von irgendjemandem geliebt“ umwandeln:

$$\forall x \quad [\forall y \quad Tier(y) \Rightarrow Liebt(x, y)] \quad \Rightarrow \quad [\exists y \quad Liebt(y, x)].$$

6 Eine Klausel kann auch als Implikation mit einer Konjunktion von Atomen in der Prämisse und einer Disjunktion von Atomen in der Schlussfolgerung dargestellt werden (Übung 7.13). Diese Form wird als **implikative Normalform** oder **Kowalski-Form** bezeichnet (insbesondere, wenn sie mit einem Symbol für die Linksimplikation geschrieben wird (Kowalski, 1979b)) und ist häufig viel leichter lesbar.

Die Schritte sehen wie folgt aus:

- **Implikationen eliminieren:**

$$\forall x \ [\neg \forall y \ \neg Tier(y) \ \vee \ Liebt(x, y)] \ \vee \ [\exists y \ Liebt(y, x)].$$

- **$\neg$ nach innen verschieben:** Neben den üblichen Regeln für negierte Verknüpfungen brauchen wir Regeln für negierte Quantoren. Wir haben also:

$\neg \forall x \, p$	wird zu	$\exists x \, \neg p$
$\neg \exists x \, p$	wird zu	$\forall x \, \neg p$

  Unser Satz durchläuft also die folgenden Umwandlungen:

$$\forall x \ [\exists y \ \neg(\neg Tier(y) \ \vee \ Liebt(x, y))] \ \vee \ [\exists y \ Liebt(y, x)]$$
$$\forall x \ [\exists y \ \neg\neg Tier(y) \ \wedge \ \neg Liebt(x, y)] \ \vee \ [\exists y \ Liebt(y, x)]$$
$$\forall x \ [\exists y \ Tier(y) \ \wedge \ \neg Liebt(x, y)] \ \vee \ [\exists y \ Liebt(y, x)].$$

  Beachten Sie, wie ein Allquantor ($\forall y$) in der Prämisse der Implikation zu einem Existenzquantor geworden ist. Der Satz liest sich jetzt wie „Entweder gibt es ein Tier, das x nicht liebt, oder (wenn das nicht der Fall ist) jemand liebt x.“ Offensichtlich wurde die Bedeutung des ursprünglichen Satzes beibehalten.

- **Variablen standardisieren:** Für Sätze wie $(\exists x \ P(x)) \vee (\exists x \ Q(x))$, die denselben Variablennamen zweimal verwenden, wird der Name einer dieser Variablen geändert. Damit vermeidet man spätere Verwirrungen, wenn die Quantoren wegfallen. Wir erhalten also:

$$\forall x \ [\exists y \ Tier(y) \ \wedge \ \neg Liebt(x, y)] \ \vee \ [\exists z \ Liebt(z, x)].$$

- **Skolemisieren:** Die **Skolemisierung** ist der Prozess, Existenzquantoren durch einfache Eliminierung zu entfernen. Im einfachen Fall entspricht das der Regel der Existentiellen Instanziierung, wie in *Abschnitt 9.1* gezeigt: Man übersetzt $\exists x \ P(x)$ in $P(A)$, wobei A eine neue Konstante ist. Allerdings können wir die Existenzielle Instanziierung nicht auf unseren obigen Satz anwenden, weil er dem Muster $\exists v \ \alpha$ nicht entspricht; nur Teile des Satzes stimmen mit dem Muster überein. Wenn wir die Regel blind auf die beiden übereinstimmenden Teile anwenden, erhalten wir:

$$\forall x \ [Tier(A) \ \wedge \ \neg Liebt(x, A)] \ \vee \ Liebt(B, x)].$$

  Das hat die völlig falsche Bedeutung: Es besagt, dass jeder entweder ein bestimmtes Tier A nicht lieben kann oder von einer bestimmten Entität B geliebt wird. Unser ursprünglicher Satz erlaubt es allen Personen, unterschiedliche Tiere nicht zu lieben oder von einer unterschiedlichen Person geliebt zu werden. Wir wollen also, dass die Skolem-Entitäten von x und z abhängig sind:

$$\forall x \ [Tier(F(x)) \ \wedge \ \neg Liebt(x, F(x))] \ \vee \ Liebt(G(x), x).$$

  Hier sind F und G **Skolem-Funktionen**. Die allgemeine Regel besagt, dass die Argumente der Skolem-Funktion allquantifizierte Variablen sind, in deren Gültigkeitsbereich der Existenzquantor erscheint. Wie bei der Existentiellen Instanziierung ist der skolemisierte Satz genau dann erfüllbar, wenn der ursprüngliche Satz erfüllbar ist.

- **Allquantoren entfernen:** An dieser Stelle müssen alle restlichen Variablen allquantifiziert sein. Darüber hinaus ist der Satz äquivalent mit einem Satz, in dem alle Allquantoren auf die linke Seite verschoben wurden. Deshalb können wir die Allquantoren entfernen:

$$[Tier(F(x)) \ \wedge \ \neg Liebt(x, F(x))] \ \vee \ Liebt(G(x), x).$$

- ∨ **über** ∧ **verteilen**:

$$[Tier(F(x)) \vee Liebt(G(x), x)] \wedge [\neg Liebt(x, F(x)) \vee Liebt(G(x), x)].$$

  In diesem Schritt kann es auch sein, dass verschachtelte Konjunktionen und Disjunktionen aufgelöst werden müssen.

Der Satz befindet sich jetzt in KNF und besteht aus zwei Klauseln. Er ist relativ unlesbar. (Es kann hilfreich sein zu erklären, dass die Skolem-Funktion $F(x)$ auf das Tier verweist, das möglicherweise von x nicht geliebt wird, während $G(x)$ sich auf jemanden bezieht, der möglicherweise x liebt.) Glücklicherweise haben es Menschen selten mit KNF-Sätzen zu tun – der Übersetzungsprozess lässt sich leicht automatisieren.

9.5.2 Die Resolutions-Inferenzregel

Die Resolutionsregel für Klauseln erster Stufe ist einfach eine geliftete Version der aussagenlogischen Resolutionsregel, die Sie in *Abschnitt 7.5.2* kennengelernt haben. Zwei Klauseln, von denen man annimmt, dass sie standardisiert sind, sodass sie keine gemeinsamen Variablen haben, können aufgelöst werden, wenn sie komplementäre Literale enthalten. Aussagenlogische Literale sind komplementär, wenn eines die Negation des anderen ist. Literale erster Stufe sind komplementär, wenn eines mit der Negation des anderen unifiziert. Wir haben also:

$$\frac{l_1 \vee \ldots \vee l_k, m_1 \vee \ldots m_n}{SUBST(\theta, l_1 \vee \ldots l_{i-1} \vee l_{i+1} \vee \ldots \vee l_k \vee m_1 \vee \ldots \vee m_{j-1} \vee m_{j+1} \vee \ldots \vee m_n)},$$

wobei $\text{UNIFY}(l_i, \neg m_j) = \theta$ gilt. Beispielsweise können wir die beiden Klauseln

$$[Tier(F(x)) \vee Liebt(G(x), x)] \text{ und } [\neg Liebt(u, v) \vee \neg Tötet(u, v)]$$

auflösen, indem wir die komplementären Literale $Liebt(G(x), x)$ und $\neg Liebt(u, v)$ mit dem Unifikator $\theta = \{u / G(x), v / x\}$ eliminieren, um die **resolvierte** Klausel

$$[Tier(F(x)) \vee \neg Tötet(G(x), x)]$$

zu erhalten.

Diese Regel ist die **binäre Resolutionsregel**, weil sie genau zwei Literale auflöst. Die binäre Resolutionsregel selbst ergibt keine vollständige Inferenzprozedur. Die vollständige Resolutionsregel löst Untermengen von Literalen in jeder Klausel auf, die unifizierbar sind. Ein alternativer Ansatz ist die Erweiterung der **Faktorisierung** – das Entfernen redundanter Literale – auf die Logik erster Stufe. Die aussagenlogische Faktorisierung reduziert zwei Literale auf eines, wenn sie *identisch* sind; die Faktorisierung erster Stufe reduziert zwei Literale auf eines, wenn sie *unifizierbar* sind. Der Unifikator muss auf die gesamte Klausel angewendet werden. Die Kombination der binären Resolution und der Faktorisierung ist vollständig.

9.5.3 Beispielbeweise

Die Resolution beweist, dass $KB \models \alpha$ gilt, indem sie beweist, dass $KB \wedge \neg\alpha$ unerfüllbar ist, d.h. durch Ableitung der leeren Klausel. Der algorithmische Ansatz ist identisch mit dem aussagenlogischen Fall, der in Abbildung 7.12 beschrieben wurde; deshalb wollen wir ihn hier nicht wiederholen. Stattdessen zeigen wir zwei Beispielbeweise. Der erste ist das Verbrecherbeispiel aus *Abschnitt 9.3*. Die Sätze in KNF sind:

$\neg Amerikaner(x) \vee \neg Waffe(y) \vee \neg Verkauft(x, y, z) \vee \neg Feindlich(z) \vee Kriminell(x)$
$\neg Rakete(x) \vee \neg Besitzt(Nono, x) \vee Verkauft(West, x, Nono)$
$\neg Feind(x, Amerika) \vee Feindlich(x)$
$\neg Rakete(x) \vee Waffe(x)$
$Besitzt(Nono, M_1)$ $\quad$ $Rakete(M_1)$
$Amerikaner(West)$ $\quad$ $Feind(Nono, Amerika)$

Außerdem binden wir das negierte Ziel $\neg Kriminell(West)$ ein. ▶ Abbildung 9.11 zeigt den Resolutionsbeweis. Beachten Sie die Struktur: eine einzige „Wirbelsäule“, beginnend mit der Zielklausel, die gegen die Klauseln aus der Wissensbasis aufgelöst wird, bis die leere Klausel erzeugt wird. Das ist charakteristisch für die Resolution in Horn-Klausel-Wissensbasen. Die Klauseln entlang der Hauptsäule entsprechen *genau* den konsekutiven Werten der Variablen *goals* in dem in Abbildung 9.6 gezeigten Algorithmus für die Rückwärtsverkettung. Das liegt daran, dass wir immer mit einer Klausel auflösen, deren positives Literal mit dem ganz linken Literal der „aktuellen“ Klausel auf der Säule unifiziert; das ist genau das, was bei der Rückwärtsverkettung passiert. Somit ist die Rückwärtsverkettung eigentlich ein Sonderfall der Resolution mit einer besonderen Kontrollstrategie, die entscheidet, was die Resolution als Nächstes ausführen soll.

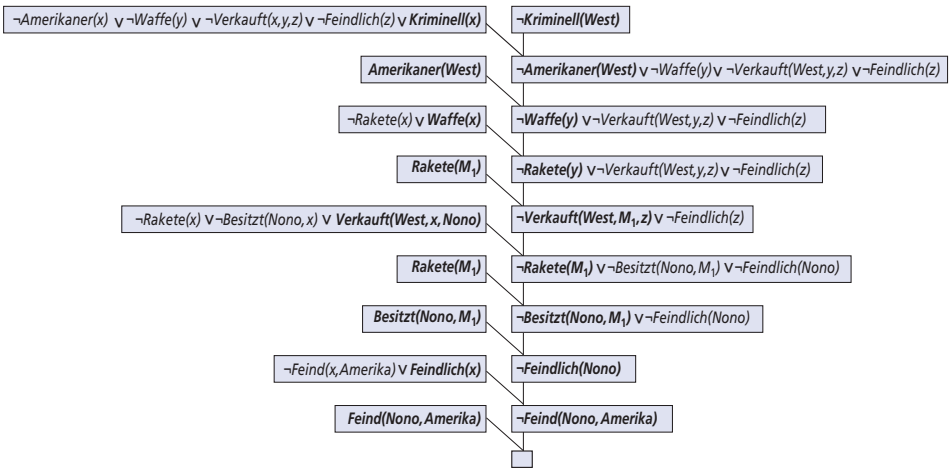


Abbildung 9.11: Ein Resolutionsbeweis, dass West ein Krimineller ist. Bei jedem Schritt sind die Literale, die unifizieren, fett dargestellt.

Unser zweites Beispiel verwendet die Skolemisierung und beinhaltet Klauseln, bei denen es sich nicht um definite Klauseln handelt. Das führt zu einer etwas komplexeren Beweisstruktur. In natürlicher Sprache stellt sich das Problem wie folgt dar:

Jeder, der alle Tiere liebt, wird von irgendjemandem geliebt.

Jemand, der ein Tier tötet, wird von niemandem geliebt.

Jack liebt alle Tiere.

Jack oder Neugier haben die Katze getötet, deren Name Tuna ist.

Hat Neugier die Katze getötet?

Zuerst drücken wir die ursprünglichen Sätze, ein wenig Hintergrundwissen und das negierte Ziel G in Logik erster Stufe aus:

A. $\forall x\,[\forall y \quad Tier(y) \Rightarrow Liebt(x, y)] \Rightarrow [\exists y\; Liebt(y, x)]$
B. $\forall x\,[\exists y \quad Tier(z) \wedge Tötet(x, z)] \Rightarrow [\forall z\; \neg Liebt(y, x)]$
C. $\forall x\; Tier(x) \Rightarrow Liebt(Jack, x)$
D. $Tötet(Jack, Tuna) \vee Tötet(Neugier, Tuna)$
E. $Katze(Tuna)$
F. $\forall x\; Katze(x) \Rightarrow Tier(x)$
¬G. $\neg\; Tötet(Neugier, Tuna)$.

Jetzt wenden wir die Konvertierungsprozedur an, um jeden Satz in KNF umzuwandeln:

A1. $Tier(F(x)) \vee Liebt(G(x), x)$
A2. $\neg\; Liebt(x, F(x)) \vee Liebt(G(x), x)$
B. $\neg Liebt(y, x)] \vee \neg Tier(z) \vee \neg Tötet(x, z)$
C. $\neg Tier(x) \vee Liebt(Jack, x)$
D. $Tötet(Jack, Tuna) \vee Tötet(Neugier, Tuna)$
E. $Katze(Tuna)$
F. $\neg Katze(x) \vee Tier(x)$
¬G. $\neg\; Tötet(Neugier, Tuna)$.

Den Resolutionsbeweis, dass Neugier die Katze getötet hat, sehen Sie in ▶ Abbildung 9.12. In natürlicher Sprache könnte der Beweis wie folgt geführt werden:

> Angenommen, Neugier hat Tuna nicht getötet. Wir wissen, dass entweder Jack oder Neugier es getan haben; also müsste es Jack gewesen sein. Tuna ist jedoch eine Katze und Katzen sind Tiere. Weil jeder, der ein Tier tötet, von niemandem geliebt wird, wissen wir, dass niemand Jack liebt. Andererseits liebt Jack alle Tiere, deshalb liebt ihn jemand; damit haben wir einen Widerspruch. Folglich hat Neugier die Katze getötet.

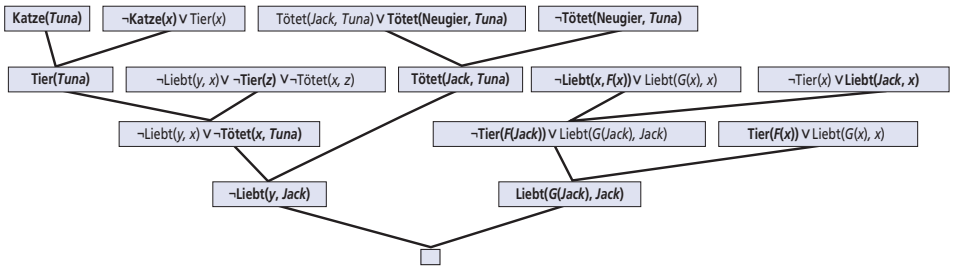


Abbildung 9.12: Ein Resolutionsbeweis, dass Neugier die Katze getötet hat. Beachten Sie die Verwendung der Faktorisierung in der Ableitung der Klausel $Liebt(G(Jack), Jack)$. Außerdem ist oben rechts zu beachten, dass die Unifikation von $Liebt(x, F(x))$ und $Liebt(Jack, x)$ nur erfolgreich sein kann, nachdem die Variablen standardisiert wurden.

Der Beweis beantwortet die Frage: „Hat Neugier die Katze getötet?“, aber häufig wollen wir allgemeinere Fragen stellen, wie etwa: „Wer hat die Katze getötet?“ Die Resolution ist dazu in der Lage, aber für die Antwort fällt etwas mehr Arbeit an. Das Ziel ist $\exists w\; Tötet(w, Tuna)$, das durch Negation zu $\neg Tötet(w, Tuna)$ in KNF wird. Wiederholt man den Beweis aus Abbildung 9.12 mit dem neuen negierten Ziel, erhält man einen einfacheren Beweisbaum, allerdings mit der Substitution $\{w / Neugier\}$ in einem der Schritte. In diesem Fall ist die Aufgabe, herauszufinden, wer die Katze getötet hat, also nur eine Frage der Verwaltung aller Bindungen für die Abfragevariablen im Beweis.

Leider kann die Resolution **nichtkonstruktive Beweise** für existentiell quantifizierte Ziele erzeugen. Beispielsweise wird $\neg Tötet(w, Tuna)$ mit $Tötet(Jack, Tuna) \vee Tötet(Neugier, Tuna)$ aufgelöst und ergibt $Tötet(Jack, Tuna)$, was erneut mit $\neg Tötet(w, Tuna)$ aufgelöst wird und die leere Klausel ergibt. Beachten Sie, dass w in diesem Beweis zwei unterschiedliche Bindungen hat; die Resolution teilt uns mit, dass irgendjemand Tuna getötet hat – entweder Jack oder Neugier. Das ist keine besondere Überraschung! Eine Lösung ist es, die erlaubten Resolutionsschritte zu beschränken, sodass die Abfragevariablen in jedem Beweis nur ein einziges Mal gebunden werden können; in diesem Fall müssen wir in der Lage sein, ein Backtracking über die möglichen Bindungen durchzuführen. Eine weitere Lösung ist, dem negierten Ziel ein spezielles **Antwortliteral** hinzuzufügen, was $\neg Tötet(w, Tuna) \vee Antwort(w)$ ergibt. Jetzt erzeugt der Resolutionsprozess eine Antwort, wenn eine Klausel erzeugt wird, die nur *ein einziges* Antwortliteral enthält. Für den Beweis in Abbildung 9.12 ist das $Antwort(Neugier)$. Der nichtkonstruktive Beweis würde die Klausel $Antwort(Neugier) \vee Antwort(Jack)$ erzeugen, die keine Antwort darstellt.

9.5.4 Vollständigkeit der Resolution

Dieser Abschnitt zeigt einen Vollständigkeitsbeweis für die Resolution. Er kann von allen übersprungen werden, die ihn als gegeben voraussetzen wollen.

Wir werden zeigen, dass die Resolution **widerspruchsvollständig** ist, d.h., *wenn* eine Menge von Sätzen unerfüllbar ist, kann die Resolution immer einen Widerspruch ableiten. Die Resolution kommt nicht infrage, um alle logischen Konsequenzen einer Menge von Sätzen zu erzeugen, aber sie kann verwendet werden, um festzustellen, ob ein bestimmter Satz logische Konsequenz einer Menge von Sätzen ist. Damit eignet sie sich, um alle Antworten auf eine bestimmte Frage $Q(x)$ zu finden, indem bewiesen wird, dass $KB \wedge \neg Q(x)$ nicht erfüllbar ist.

Tipp

Wir werden es als gegeben betrachten, dass alle Sätze erster Stufe (ohne Gleichheit) als Satz von Klauseln in KNF umgeschrieben werden können. Das lässt sich durch Induktion für die Form des Satzes beweisen, wobei atomare Sätze als Basisfall betrachtet werden (Davis und Putnam, 1960). Unser Ziel ist es deshalb, das Folgende zu beweisen: *Wenn S eine unerfüllbare Menge von Klauseln ist, dann führt die Anwendung einer endlichen Anzahl von Resolutionsschritten auf S zu einem Widerspruch.*

Unsere Beweisskizze folgt dem ursprünglichen Beweis nach Robinson, wobei einige Vereinfachungen von Genesereth und Nilsson (1987) übernommen wurden. Die Grundstruktur des Beweises (▶ Abbildung 9.13) sieht folgendermaßen aus:

1. Erstens beobachten wir, dass es, wenn S nicht erfüllbar ist, eine bestimmte Menge von *Grundinstanzen* der Klauseln von S gibt, sodass diese Menge ebenfalls nicht erfüllbar ist (Satz von Herbrand).

2. Anschließend beziehen wir uns auf das **Grundresolutionstheorem** aus *Kapitel 7*, das besagt, dass die aussagenlogische Resolution für Grundsätze vollständig ist.

3. Dann verwenden wir ein **Lifting-Lemma**, um zu zeigen, dass es für jeden aussagenlogischen Resolutionsbeweis, welcher die Menge der Grundsätze verwendet, einen entsprechenden Resolutionsbeweis erster Stufe gibt, der die Sätze erster Stufe verwendet, aus denen die Grundsätze ermittelt wurden.

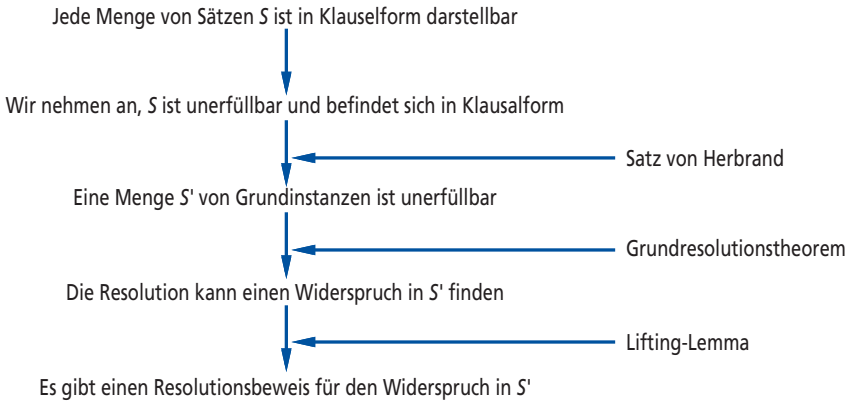


Abbildung 9.13: Struktur eines Vollständigkeitsbeweises für die Resolution.

Für die Ausführung des ersten Schrittes brauchen wir drei neue Konzepte:

- **Herbrand-Universum:** Wenn S eine Menge von Klauseln ist, dann ist H_S, das Herbrand-Universum von S, die Menge aller Grundterme, die aus dem Folgenden konstruierbar sind:
  a. den Funktionssymbolen in S (sofern vorhanden)
  b. den Konstantensymbolen in S (sofern vorhanden); falls es keine gibt, aus dem Konstantensymbol A

  Enthält S beispielsweise nur die Klausel $\neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B)$, dann ist H_S die folgende unendliche Menge von Grundtermen:

  $$\{ A, B, F(A, A), F(A, B), F(B, A), F(B, B), F(A, F(A, A)), ...\}.$$

- **Sättigung:** Wenn S eine Menge von Klauseln und P eine Menge von Grundtermen ist, dann ist $P(S)$, die Sättigung von S in Bezug auf P, die Menge aller Grundklauseln, die man durch Anwendung aller möglichen konsistenten Substitutionen von Grundtermen in P mit Variablen in S erhält.

- **Herbrand-Basis:** Die Sättigung einer Menge S von Klauseln im Hinblick auf ihr Herbrand-Universum wird als Herbrand-Basis von S bezeichnet, dargestellt als $H_S(S)$. Wenn S beispielsweise nur die oben gezeigte Klausel enthält, ist $H_S(S)$ die unendliche Menge von Klauseln:

  $$\{\neg P(A, F(A, A)) \vee \neg Q(A, A) \vee R(A, B),$$
  $$\neg P(B, F(B, A)) \vee \neg Q(B, A) \vee R(B, B),$$
  $$\neg P(F(A, A), F(F(A, A), A)) \vee \neg Q(F(A, A), A) \vee R(F(A, A), B),$$
  $$\neg P(F(A, B), F(F(A, B), A)) \vee \neg Q(F(A, B), A) \vee R(F(A, B), B), ...\}.$$

Diese Definitionen erlauben es uns, eine Form des **Satzes von Herbrand** zu formulieren (Herbrand, 1930):

> Wenn eine Menge S von Klauseln nicht erfüllbar ist, gibt es eine endliche Untermenge $H_S(S)$, die ebenfalls nicht erfüllbar ist.

Es sei S' diese endliche Menge von Grundsätzen. Jetzt können wir uns auf das Grundresolutionstheorem (*Abschnitt 7.5.2*) beziehen, um zu zeigen, dass der **Resolutionsabschluss** $RC(S')$ die leere Klausel enthält. Das bedeutet, die Ausführung der aussagenlogischen Resolution zur Vervollständigung von S' leitet einen Widerspruch ab.

Nachdem wir festgestellt haben, dass es immer einen Resolutionsbeweis gibt, der eine endliche Untermenge der Herbrand-Basis von S enthält, ist der nächste Schritt, zu zeigen, dass es einen Resolutionsbeweis unter Verwendung der Klauseln von S selbst gibt, bei denen es sich nicht unbedingt um Grundklauseln handeln muss.

Gödels Unvollständigkeitstheorem

Durch eine kleine Erweiterung der Sprache der Logik erster Stufe, die das **mathematische Induktionsschema** in der Arithmetik ermöglichte, konnte Kurt Gödel in seinem **Unvollständigkeitstheorem** zeigen, dass es wahre arithmetische Sätze gibt, die nicht bewiesen werden können.

Der Beweis des Unvollständigkeitstheorems kann im Rahmen dieses Buches nicht besprochen werden, weil er mindestens 30 Seiten füllt, aber wir können Ihnen hier einen Hinweis geben. Wir beginnen mit der logischen Zahlentheorie. In dieser Theorie gibt es eine einzige Konstante, 0, und eine einzige Funktion, S (die Nachfolgerfunktion). In dem vorgesehenen Modell bezeichnet $S(0)$ die 1, $S(S(0))$ die 2 usw. Auf diese Weise hat die Sprache Namen für alle natürlichen Zahlen. Das Vokabular beinhaltet auch die Funktionssymbole $+$, $\times$ und $Expt$ (Exponentiation, Potenzierung) sowie die übliche Menge an logischen Verknüpfungen und Quantoren. Der erste Schritt ist zu bemerken, dass die Menge der Sätze, die wir in dieser Sprache aufschreiben können, nummeriert werden kann. (Stellen Sie sich vor, es wird eine alphabetische Reihenfolge für die Symbole definiert und dann wird jede der Mengen von Sätzen der Länge 1, 2 usw. nach alphabetischer Reihenfolge angeordnet.) Dann können wir jeden Satz α mit einer eindeutigen natürlichen Zahl $\#\alpha$ versehen (der **Gödel-Nummer**). Das ist entscheidend: Die Zahlentheorie enthält einen Namen für jeden ihrer eigenen Sätze. Analog dazu können wir jeden möglichen Beweis P mit einer Gödel-Nummer $G(P)$ versehen, weil ein Beweis einfach eine endliche Folge von Sätzen ist.

Nehmen wir jetzt an, wir haben eine rekursiv aufzählbare Menge A von wahren Aussagen über die natürlichen Zahlen. Wie Sie wissen, lässt sich A durch eine gegebene Menge von ganzen Zahlen benennen; dann können wir uns auch vorstellen, in unserer Sprache einen Satz $\alpha(j, A)$ der folgenden Art zu schreiben:

> $\forall\, i \quad i$ ist nicht die Gödel-Nummer eines Beweises des Satzes, dessen Gödel-Nummer j ist, wobei der Beweis nur Prämissen in A verwendet.

Dann sei σ der Satz $\alpha(\#\sigma, A)$, d.h. ein Satz, der seine eigene Unbeweisbarkeit aus A ausdrückt. (Dass dieser Satz existiert, ist immer wahr, aber nicht ganz offensichtlich.)

Jetzt bringen wir das folgende geniale Argument: Angenommen, σ ist aus A beweisbar; dann ist σ falsch (weil σ besagt, dass es nicht bewiesen werden kann). Dann haben wir jedoch einen falschen Satz, der aus A bewiesen werden kann: A kann also nicht nur aus wahren Sätzen bestehen – eine Verletzung unserer Prämisse. Aus diesem Grund ist σ nicht aus A beweisbar. Aber das ist genau das, was σ selbst behauptet; damit ist σ ein wahrer Satz.

Auf diese Weise haben wir (unter Einsparung von 29½ Seiten) gezeigt, dass es für jede Menge wahrer Sätze der Zahlentheorie, und insbesondere jede Menge grundlegender Axiome, andere wahre Sätze gibt, die aus diesen Axiomen *nicht* bewiesen werden können. Damit wird unter anderem festgestellt, dass wir nie alle Theoreme der Mathematik *innerhalb eines vorgegebenen Axiomsystems* beweisen können. Offensichtlich war das eine wichtige Entdeckung für die Mathematik. Ihre Bedeutung für die KI wurde allgemein diskutiert, beginnend mit Spekulationen von Gödel selbst. Wir werden diese Diskussion in *Kapitel 26* aufgreifen.

Wir beginnen mit der Betrachtung einer einzelnen Anwendung der Resolutionsregel. Robinson gibt folgendes Lemma an:

Es seien C_1 und C_2 zwei Klauseln ohne gemeinsame Variablen sowie C_1' und C_2' Grundinstanzen von C_1 und C_2. Wenn C' eine Resolvente von C_1' und C_2' ist, dann gibt es eine Klausel C, sodass gilt: (1) C ist eine Resolvente von C_1 und C_2 und (2) C' ist eine Grundinstanz von C.

Dies bezeichnet man auch als **Lifting-Lemma**, weil es einen Beweisschritt von Grundklauseln in allgemeine Klauseln erster Stufe anhebt („liftet“). Um sein grundlegendes Lifting-Lemma zu beweisen, musste Robinson die Unifikation erfinden und alle Eigenschaften der meisten allgemeinen Unifikatoren ableiten. Anstatt den Beweis hier zu wiederholen, veranschaulich wir einfach das Lemma:

$$
\begin{aligned}
C_1 &= \neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B) \\
C_2 &= \neg N(G(y), z) \vee P(H(y), z) \\
C_1' &= \neg P(H(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\
C_2' &= \neg N(G(B), F(H(B), A)) \vee P(H(B), F(H(B), A)) \\
C' &= \neg N(G(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B) \\
C &= \neg N(G(y), F(H(y), A)) \vee \neg Q(H(y), A) \vee R(H(y), B).
\end{aligned}
$$

Wir sehen, dass C' tatsächlich eine Grundinstanz von C ist. Damit C_1' und C_2' Resolventen haben, müssen sie konstruiert werden, indem zuerst auf C_1 und C_2 der allgemeinste Unifikator eines Paares komplementärer Literale in C_1 und C_2 angewendet wird. Aus dem Lifting-Lemma kann einfach eine ähnliche Aussage über beliebige Folgen von Anwendungen der Resolutionsregel abgeleitet werden:

> Für jede Klausel C' im Resolutionsabschluss von S' gibt es eine Klausel C im Resolutionsabschluss von S, sodass C' eine Grundinstanz von C ist und die Ableitung von C dieselbe Länge wie die Ableitung von C' hat.

Aus diesem Fakt folgt, dass die leere Klausel auch im Resolutionsabschluss von S erscheinen muss, wenn sie im Resolutionsabschluss von S' erscheint. Das liegt daran, dass die leere Klausel keine Grundinstanz einer anderen Klausel sein kann. Zusammenfassend können wir sagen: Wir haben gezeigt, dass es eine endliche Ableitung der leeren Klausel unter Verwendung der Resolutionsregel gibt, wenn S nicht erfüllbar ist.

Das Lifting des Theorembeweisens von Grundklauseln auf Klauseln erster Stufe stellt eine riesige Leistungssteigerung dar. Diese Steigerung basiert auf der Tatsache, dass der Beweis in der Logik erster Stufe die Variablen nur dann instanziieren muss, wenn sie für den Beweis benötigt werden, während die Grundklauselmethoden eine große Anzahl zufälliger Instanziierungen auswerten mussten.

9.5.5 Gleichheit

Keine der bisher in diesem Kapitel beschriebenen Inferenzmethoden behandelt eine Behauptung der Form $x = y$. Es gibt drei verschiedene Ansätze dafür. Der erste Ansatz ist, die Gleichheit zu axiomisieren – Sätze über die Gleichheitsrelation in der Wissensbasis abzulegen. Wir müssen sagen, dass die Gleichheit reflexiv, symmetrisch und transitiv ist. Außerdem müssen wir feststellen, dass wir Gleiches für Gleiches in jedem Prädikat und

jeder Funktion ersetzen können. Wir brauchen also drei grundlegende Axiome und dann je eines für jedes Prädikat und jede Funktion:

$$\begin{aligned}
&\forall x \quad x = x\\
&\forall x, y \quad x = y \Rightarrow y = x\\
&\forall x, y, z \quad x = y \wedge y = z \Rightarrow x = z\\
&\\
&\forall x, y \quad x = y \Rightarrow (P_1(x) \Leftrightarrow P_1(y))\\
&\forall x, y \quad x = y \Rightarrow (P_2(x) \Leftrightarrow P_2(y))\\
&\quad\vdots\\
&\forall w, x, y, z \quad w = y \wedge x = z \Rightarrow (F_1(w, x) = F_1(y, z))\\
&\forall w, x, y, z \quad w = y \wedge x = z \Rightarrow (F_2(w, x) = F_2(y, z))\\
&\quad\vdots
\end{aligned}$$

Mit diesen Sätzen kann eine Standard-Inferenzprozedur wie etwa die Resolution Aufgaben ausführen, für die Schlüsse über Gleichheit erforderlich sind, wie beispielsweise die Lösung mathematischer Gleichungen. Allerdings generieren diese Axiome sehr viele Schlüsse, von denen die meisten für einen Beweis nicht hilfreich sind. Deshalb hat man nach effizienteren Wegen für die Behandlung der Gleichheit gesucht. Eine Alternative ist es, Inferenzregeln statt Axiome hinzuzufügen. Die einfachste Regel, die **Demodulation**, übernimmt eine Einheitsklausel $x = y$ und eine Klausel α, die den Term x enthält, und bildet eine neue Klausel, indem x in α durch y ersetzt wird. Das funktioniert, wenn sich der Term in α mit x unifizieren lässt; er muss nicht genau gleich zu x sein. Die Demodulation ist richtungsabhängig; für ein gegebenes x = y wird das x immer durch y ersetzt und niemals umgekehrt. Das heißt, dass sich die Demodulation eignet, um Ausdrücke mithilfe von Demodulatoren wie zum Beispiel $x + 0 = x$ oder $x^1 = x$ zu vereinfachen. Bei einem anderen Beispiel, das mit

$$\begin{aligned}
&\mathit{Vater}(\mathit{Vater}(x)) = \mathit{VaterseiteGrossvater}(x)\\
&\mathit{Geburtstag}(\mathit{Vater}(\mathit{Vater}(\mathit{Bella})), 1926)
\end{aligned}$$

gegeben ist, können wir durch Demodulation schließen:

$$\mathit{Geburtstag}(\mathit{VaterseiteGrossvater}(\mathit{Bella}), 1926)$$

Formaler ausgedrückt, haben wir

- **Demodulation:** Für alle Terme x, y und z, wobei z irgendwo im Literal m_i erscheint und wo $\textsc{Unify}(x, z) = \theta$ ist, gilt:

$$\frac{x = y, m_1 \vee \ldots \vee m_n}{SUB(SUBST(\theta, x), SUBST(\theta, y) m_1 \vee \ldots \vee m_n)}.$$

  Hier ist $\textsc{Subst}$ die übliche Substitution einer Bindungsliste und $\textsc{Sub}(x, y, m)$ bedeutet, x durch y überall dort zu ersetzen, wo x in m erscheint.

Die Regel kann auch erweitert werden, sodass Nichteinheitsklauseln berücksichtigt werden, in denen ein Gleichheitsliteral auftritt:

- **Paramodulation:** Für alle Terme x, y und z, wobei z irgendwo in Literal m_i erscheint und wo $\textsc{Unify}(x , z) = \theta$ ist, gilt:

$$\frac{l_1 \vee \ldots \vee l_k \vee x = y, m_1 \vee \ldots \vee m_n}{SUB(SUBST(\theta, x), SUBST(\theta, y), SUBST(\theta, l_1 \vee \ldots \vee l_k \vee m_1 \vee \ldots \vee m_n))}.$$

Ein Beispiel: Aus

$$P(F(x, B), x) \vee Q(x) \quad \text{und} \quad F(A, y) = y \vee R(y)$$

erhalten wir θ = UNIFY(F(A, y), F(x, B)) = {x/A, y/B} und können durch Paramodulation schließen, dass der Satz

$$P(B, A) \vee Q(A) \vee R(B)$$

gilt. Paramodulation liefert eine vollständige Inferenzprozedur für Logik erster Stufe mit Gleichheit.

Ein dritter Ansatz verarbeitet das Schließen mit Gleichheit komplett innerhalb eines erweiterten Unifikationsalgorithmus. Das bedeutet, Terme sind unifizierbar, wenn sie *beweisbar* gleich unter einer bestimmten Substitution sind, wobei „beweisbar“ ein gewisses Schließen zur Gleichheit erlaubt. Beispielsweise sind die Terme 1+2 und 2+1 normalerweise nicht unifizierbar, aber ein Unifikationsalgorithmus, der weiß, dass $x + y = y + x$ ist, könnte sie mithilfe der leeren Substitution unifizieren. Die **Gleichheitsunifikation** dieser Art kann mithilfe effizienter Algorithmen erfolgen, die für die jeweils verwendeten Axiome entworfen wurden (Kommutativität, Assoziativität usw.) statt durch explizite Inferenz mit diesen Axiomen. Theorembeweiser, die diese Technik verwenden, sind eng mit den in *Abschnitt 9.4* beschriebenen Constraint-Logikprogrammiersystemen verwandt.

9.5.6 Resolutionsstrategien

Wir wissen, dass die wiederholte Anwendung der Resolutions-Inferenzregel irgendwann einen Beweis findet, falls ein solcher existiert. In diesem Unterabschnitt betrachten wir Strategien, die helfen, *effizient* Beweise zu finden.

Einheitspriorität: Diese Strategie bevorzugt Resolutionen, bei denen einer der Sätze ein einzelnes Literal darstellt (auch als **Einheitsklausel** bezeichnet). Die Idee hinter der Strategie ist, dass wir versuchen, eine leere Klausel zu produzieren, sodass es sinnvoll sein kann, Inferenzen zu bevorzugen, die kürzere Klauseln erzeugen. Das Auflösen eines Einheitssatzes (wie etwa P) mit einem anderen Satz (wie etwa $\neg P \vee \neg Q \vee R$) ergibt immer eine Klausel (in diesem Fall $\neg Q \vee R$), die kürzer als die andere Klausel ist. Als diese Strategie der Einheitspriorität 1964 zum ersten Mal für die aussagenlogische Inferenz ausprobiert wurde, führte sie zu einer drastischen Beschleunigung und ermöglichte den Beweis von Theoremen, die ohne diese Prioritäten nicht hätten verarbeitet werden können. Die **Einheitsresolution** ist eine eingeschränkte Form der Resolution, wobei jeder Resolutionsschritt eine Einheitsklausel beinhalten muss. Die Einheitsresolution ist im Allgemeinen unvollständig, für Horn-Wissensbasen ist sie jedoch vollständig. Einheitsresolutionsbeweise für Horn-Wissensbasen erinnern an die Vorwärtsverkettung.

Der OTTER-Theorembeweiser (Organized Techniques for Theorem-proving and Effective Research, McCune, 1992) verwendet eine Form der Bestensuche. Seine Heuristikfunktion misst das „Gewicht“ jeder Klausel, wobei leichtere Klauseln bevorzugt werden. Die genaue Wahl der Heuristik liegt beim Benutzer, doch sollte im Allgemeinen das Gewicht einer Klausel mit ihrer Größe oder Schwierigkeit korreliert sein. Einheitsklauseln werden als leicht behandelt; die Suche lässt sich somit als Verallgemeinerung der Einheitsbevorzugungsstrategie ansehen.

Stützmenge (Set of Support): Prioritäten, die bestimmte Resolutionen bevorzugt ausprobieren, sind hilfreich, aber im Allgemeinen ist es effektiver, zu versuchen, einige potenzielle Resolutionen völlig zu eliminieren. Zum Beispiel können wir darauf bestehen, dass jeder Resolutionsschritt mindestens ein Element einer speziellen Klauselmenge – die *Stützmenge* – einbezieht. Die Resolvente wird dann der Stützmenge hinzugefügt. Wenn die Stützmenge klein im Vergleich zur gesamten Wissensbasis ist, wird der Suchraum wesentlich reduziert.

Wir müssen mit diesem Ansatz vorsichtig umgehen, weil eine ungeeignete Auswahl für die Stützmenge den Algorithmus unvollständig macht. Wenn wir jedoch die Stützmenge S so wählen, dass die restlichen Sätze gemeinsam erfüllbar sind, ist die Resolution mit Stützmenge vollständig. Zum Beispiel kann man die negierte Abfrage als Stützmenge verwenden, mit der Annahme, dass die ursprüngliche Wissensbasis konsistent ist. (Wenn sie schließlich nicht konsistent ist, ist die Tatsache, dass die Abfrage daraus folgt, müßig.) Die Strategie der Stützmenge hat den zusätzlichen Vorteil, dass sie zielgerichtete Beweisbäume erzeugt, die für Menschen oftmals leicht verständlich sind.

Eingaberesolution: In dieser Strategie kombiniert jede Resolution einen der Eingabesätze (aus der Wissensbasis oder der Abfrage) mit einem anderen Satz. Der Beweis in Abbildung 9.11 verwendet nur Eingaberesolutionen und hat die charakteristische Form einer einzelnen „Wirbelsäule“, wobei einzelne Sätze entlang dieser Wirbelsäule kombiniert werden. Offensichtlich ist der Speicherplatz für Beweisbäume dieser Art kleiner als der Speicherplatz für Beweisgraphen. In Horn-Wissensbasen ist der Modus ponens eine Art Eingaberesolutionsstrategie, weil er eine Implikation aus der ursprünglichen Wissensbasis mit einigen anderen Sätzen kombiniert. Es ist also nicht überraschend, dass die Eingaberesolution für Wissensbasen in Horn-Form vollständig ist, im allgemeinen Fall dagegen unvollständig. Die Strategie der **linearen Resolution** ist eine leichte Verallgemeinerung, die es erlaubt, dass P und Q zusammen aufgelöst werden, wenn sich P entweder in der ursprünglichen Wissensbasis befindet oder P im Beweisbaum ein Vorfahre von Q ist. Die lineare Resolution ist vollständig.

Subsumtion: Die Methode der Subsumtion eliminiert alle Sätze, die durch einen existierenden Satz in der Wissensbasis subsumiert werden, also spezifischer sind als dieser. Befindet sich beispielsweise $P(x)$ in der Wissensbasis, ist es nicht sinnvoll, $P(A)$ hinzuzufügen, und noch viel weniger sinnvoll, $P(A) \vee Q(B)$ hinzuzufügen. Die Subsumtion hilft, die Wissensbasis und damit den Suchraum klein zu halten.

9.5.7 Praktische Verwendung von Resolutions-Theorembeweisern

Theorembeweiser lassen sich auf die Probleme anwenden, die mit der **Synthese** und **Verifizierung** von Hard- und Software zu tun haben. Deshalb wird die Forschung zu Theorem-Beweismethoden auf den Gebieten Hardwaredesign, Programmiersprachen und Softwaretechnik – d.h. nicht nur in der KI – durchgeführt.

Im Fall von Hardware beschreiben Axiome die Interaktionen zwischen Signalen und Schaltungselementen. (Ein Beispiel dazu finden Sie in *Abschnitt 8.4.2*.) Speziell für die Verifizierung konzipierte logische Schlussfolgerungsmodule sind in der Lage, vollständige CPUs einschließlich ihrer Timing-Eigenschaften zu verifizieren (Srivas und Bickford, 1990). Mit dem Theorembeweiser AURA ließen sich Schaltkreise entwerfen, deren Design kompakter als bei allen vorherigen Versionen war (Wojciechowski und Wojcik, 1983).

Im Fall von Software ist das Schlussfolgern über Programme vergleichbar mit dem Schlussfolgern über Aktionen, wie es *Kapitel 7* erläutert hat: Axiome beschreiben die Vorbedingungen und Wirkungen jeder Anweisung. Die formale Synthese von Algorithmen war eine der ersten Anwendungen von Theorembeweisern, wie sie Cordell Green (1969a) skizziert hat, der wiederum auf früheren Ideen von Herbert Simon (1963) aufbaut. Das Konzept besteht darin, ein Theorem konstruktiv dahingehend zu beweisen, dass „es ein Programm *p* gibt, das eine bestimmte Spezifikation erfüllt." Obwohl eine sogenannte vollständig automatisierte **deduktive Synthese** für die Allzweckprogrammierung bislang noch in den Kinderschuhen steckt, kann die handgeführte deduktive Synthese beim Entwurf mehrerer neuer und komplizierter Algorithmen bereits Erfolge verbuchen. Auch im Bereich der Synthese von Spezialprogrammen wie zum Beispiel Code für wissenschaftliche Berechnungen wird aktiv geforscht.

Ähnliche Verfahren werden heute für die Softwareverifikation durch Systeme wie zum Beispiel den SPIN-Modellchecker (Holzmann, 1997) eingesetzt. Zum Beispiel wurde das Raumfahrzeug-Steuerungsprogramm Remote Agent vor und nach dem Flug verifiziert (Havelund et al., 2000). Der RSA-Algorithmus für die Public-Key-Verschlüsselung und der Boyer-Moore-Algorithmus zum Zeichenfolgenvergleich sind ebenfalls auf diese Weise verifiziert worden (Boyer und Moore, 1984).

Bibliografische und historische Hinweise

Gottlob Frege, der 1879 die erste vollständige Logik erster Stufe entwickelte, basierte sein Inferenzsystem auf einer großen Sammlung logisch gültiger Schemas sowie einer einzelnen Inferenzregel, Modus ponens. Whitehead und Russell (1910) erklärten die sogenannten *Übergangsregeln* (der Begriff stammt eigentlich von Herbrand (1930)), mit denen Quantoren an den Anfang von Formeln verschoben werden. Skolem-Konstanten und Skolem-Funktionen gehen auf Thoralf Skolem (1920) zurück. Seltsamerweise hat Skolem auch das Herbrand-Universum eingeführt (Skolem, 1928).

Der Satz von Herbrand (Herbrand, 1930) spielte eine entscheidende Rolle bei der Entwicklung automatisierter Schlussmethoden. Herbrand ist auch der Erfinder der Unifikation. Gödel (1930) baute auf den Ideen von Skolem und Herbrand auf und zeigte, dass die Logik erster Stufe eine vollständige Beweisprozedur hat. Alan Turing (1936) und Alonzo Church (1936) zeigten gleichzeitig unter Verwendung sehr unterschiedlicher Beweise, dass die Gültigkeit in der Logik erster Stufe nicht entscheidbar war. Das ausgezeichnete Buch von Enderton (1972) erklärt all diese Ergebnisse in einer strengen, aber dennoch gut verständlichen Weise.

Abraham Robinson schlug vor, dass ein automatisiertes Schlussfolgerungsmodul mithilfe der Überführung in Aussagenlogik und dem Satz von Herbrand aufgebaut werden könnte, und Paul Gilmore (1960) schrieb das erste Programm. Davis und Putnam (1960) haben die Methode der Überführung in Aussagenlogik eingeführt, die *Abschnitt 9.1* erläutert hat. Prawitz (1960) entwickelte das wichtige Konzept, den Suchprozess durch die Forderung nach aussagenlogischer Inkonsistenz zu steuern und Terme aus dem Herbrand-Universum nur dann zu erzeugen, wenn es erforderlich war, um eine aussagenlogische Inkonsistenz zu erzeugen. Nach weiteren Entwicklungen durch andere Forscher führte diese Idee J.A. Robinson (nicht verwandt mit A. Robinson) zu der Idee, die Resolutionsmethode zu entwickeln (Robinson, 1965).

In der KI wurde die Resolution von Cordell Green und Bertram Raphael (1968) für Frage/Antwort-Systeme übernommen. Frühe KI-Implementierungen konzentrieren sich auf Datenstrukturen, die das effiziente Abrufen von Fakten erlauben; diese Arbeit wird in Büchern über die KI-Programmierung beschrieben (Charniak et al., 1987; Norvig, 1992; Forbus und de Kleer, 1993). Anfang der 1970er Jahre hatte sich die **Vorwärtsverkettung** in der KI als leicht verständliche Alternative zur Resolution etabliert. Da KI-Anwendungen normalerweise eine große Anzahl an Regeln beinhalten, war es wichtig, effiziente Techniken für den Regelvergleich zu entwickeln, insbesondere für inkrementelle Updates. Die Technologie für **Produktionssysteme** wurde entwickelt, um solche Anwendungen zu unterstützen. Die Produktionssystemsprache OPS-5 (Forgy, 1981; Brownston et al., 1985), die den **Rete**-Übereinstimmungsprozess (Forgy, 1982) einbindet, wurde für Anwendungen wie zum Beispiel das Expertensystem R1 zur Minicomputerkonfiguration eingesetzt (McDermott, 1982).

Die kognitive Architektur SOAR (Laird et al., 1987) wurde für die Bearbeitung sehr großer Regelmengen konzipiert – bis zu einer Million Regeln (Doorenbos, 1994). Beispielanwendungen von SOAR umfassen die Steuerung eines Simulators für Kampfflugzeuge (Jones et al., 1998), die Verwaltung des Luftraumes (Taylor et al., 2007), KI-Figuren für Computerspiele (Wintermute et al., 2007) und Ausbildungswerkzeuge für Soldaten (Wray und Jones, 2005).

Die Forschung auf dem Gebiet der **deduktiven Datenbanken** begann mit einem Workshop in Toulouse, 1977, der Experten für logische Inferenz und Datenbanksysteme zusammenbrachte (Gallaire und Minker, 1978). Die einflussreichen Arbeiten von Chandra und Harel (1980) sowie von Ullman (1985) führten zur Akzeptanz von Datalog als Standardsprache für deduktive Datenbanken. Durch die von Bancilhon et al. (1986) entwickelte Technik der **magischen Mengen** für das Neuschreiben von Regeln konnte die Vorwärtsverkettung von der Zielgerichtetheit der Rückwärtsverkettung profitieren. Derzeit arbeitet man unter anderem daran, mehrere Datenbanken in einen einheitlichen Datenraum zu integrieren (Halevy, 2007).

Die **Rückwärtsverkettung** für logische Inferenz erschien erstmals in der Sprache PLANNER von Hewitt (1969). Inzwischen hatte Alain Colmerauer 1972 Prolog entwickelt und implementiert, um natürliche Sprache zu parsen – die Klauseln von Prolog waren anfangs als kontextfreie Grammatikregeln vorgesehen (Roussel, 1975; Colmerauer et al., 1973). Ein großer Teil des theoretischen Hintergrundes für die Logikprogrammierung wurde von Robert Kowalski, der mit Colmerauer zusammenarbeitete, entwickelt. Einen historischen Überblick finden Sie bei Kowalski (1988) sowie Colmerauer und Roussel (1993). Effiziente Prolog-Compiler basieren im Allgemeinen auf dem WAM (Warren Abstract Machine)-Berechnungsmodell, das von David H. D. Warren (1983) entwickelt wurde. Van Roy (1990) zeigte, dass Prolog-Programme hinsichtlich der Geschwindigkeit durchaus mit C-Programmen konkurrieren können.

Methoden zur Vermeidung unnötiger Schleifen in rekursiven Logikprogrammen wurden unabhängig von Smith et al. (1986) sowie Tamaki und Sato (1986) entwickelt. In der letztgenannten Arbeit wird auch die Memoisation für Logikprogramme beschrieben, eine Methode, die von David S. Warren ausführlich als **tabulierte Logikprogrammierung** entwickelt wurde. Swift und Warren (1994) zeigen, wie die WAM erweitert werden kann, um Tabulierung zu verarbeiten, sodass Datalog-Programme um eine Größenordnung schneller ausgeführt werden können als deduktive Datenbanksysteme mit Vorwärtsverkettung.

Frühe theoretische Arbeiten zur Constraint-Logikprogrammierung stammen von Jaffar und Lassez (1987). Jaffar et al. (1992) entwickelten das CLP(R)-System für die Verarbeitung reellwertiger Bedingungen. Heute gibt es kommerzielle Produkte, mit denen sich Konfigurations- und Optimierungsprobleme im großen Maßstab lösen lassen; zu den bekanntesten gehört ILOG (Junker, 2003). Die Antwortmengenprogrammierung (Gelfond, 2008) erweitert Prolog und erlaubt Disjunktion und Negation.

Es gibt zahlreiche Lehrbücher zur Logikprogrammierung und zu Prolog, unter anderem von Shoham (1994), Bratko (2001), Clocksin (2003) und Clocksin und Mellish (2003). Bis zur Einstellung im Jahr 2000 war das *Journal of Logic Programming* das wichtigste Magazin; es wurde jetzt durch *Theory and Practice of Logic Programming* ersetzt. Es gibt verschiedene Konferenzen zur Logikprogrammierung, wie etwa die *ICLP* (*International Conference on Logic Programming*) und das *ILPS* (*International Logic Programming Symposium*).

Die Forschung zum **mathematischen Theorembeweisen** begann noch vor der Entwicklung der ersten vollständigen Logiksysteme erster Stufe. Der Geometry Theorem Prover von Herbert Gelernter (Gelernter, 1959) verwendete heuristische Suchmethoden kombiniert mit Diagrammen zur Kürzung falscher Unterziele und war in der Lage, einige sehr komplizierte Ergebnisse der Euklidschen Geometrie zu beweisen. Die Demodulations- und Paramodulationsregeln für das Schließen mit Gleichheit wurden von Wos et al. (1967) bzw. Wos und Robinson (1968) eingeführt. Diese Regeln wurden auch unabhängig im Kontext von Systemen zur Umformulierung von Termen entwickelt (Knuth und Bendix, 1970). Die Einbindung des Schließens mit Gleichheit in den Unifikationsalgorithmus geht auf Gordon Plotkin (1972) zurück. Jouannaud und Kirchner (1991) bieten einen Überblick über die Gleichheitsunifikation unter dem Aspekt der Umformulierung von Termen. Ein Überblick über die Unifikation wird von Baader und Snyder (2001) angegeben.

Es wurden mehrere Steuerungsstrategien für die Resolution vorgeschlagen, beginnend mit der Einheitsprioritätsstrategie (Wos et al., 1964). Die Strategie der Stützmengen wurde von Wos et al. (1965) eingeführt, um die Resolution zielgerichteter zu machen. Die lineare Resolution wurde zum ersten Mal von Loveland (1970) erwähnt. Genesereth und Nilsson (1987, *Kapitel 5*) zeigen eine kurze, aber gründliche Analyse einer Vielzahl von Steuerungsstrategien.

A Computational Logic (Boyer und Moore, 1979) ist das grundlegende Nachschlagewerk zum Boyer-Moore-Theorembeweiser. Stickel (1992) beschreibt den PTTP (Prolog Technology Theorem Prover), der die Vorteile der Prolog-Kompilierung mit der Vollständigkeit der Modelleliminierung kombiniert. SETHEO (Letz et al., 1992) ist ein weiterer allgemein eingesetzter Theorembeweiser, der auf diesem Ansatz aufbaut. LEANTAP (Beckert und Posegga, 1995) ist ein effizienter Theorembeweiser, der in nur 25 Zeilen Prolog implementiert ist. Weidenbach (2001) beschreibt SPASS, einen der leistungsfähigsten aktuellen Theorembeweiser. Als erfolgreichster Theorembeweiser hat in den letzten Jahreswettbewerben VAMPIRE (Riazanov and Voronkov, 2002) abgeschnitten. Das COQ-System (Bertot et al., 2004) und der E-Gleichungslöser (Schulz, 2004) haben sich ebenfalls als wertvolle Werkzeuge für das Beweisen von Korrektheit erwiesen. Theorembeweiser sind eingesetzt worden, um Software für die Raumfahrzeugsteuerung automatisch zu synthetisieren und zu verifizieren (Denney et al., 2006). Dazu gehört auch die neue Orion-Kapsel der NASA (Lowry, 2008). Der Entwurf des 32-Bit-Mikroprozessors FM9001 wurde durch das NQTHM-System (Hunt und Brock, 1992) als korrekt

bewiesen. Die *Conference on Automated Deduction* (CADE) führt einen jährlichen Wettbewerb für automatisierte Theorembeweiser durch. Von 2002 bis 2008 ist VAMPIRE (Riazanov und Voronkov, 2002) das erfolgreichste System gewesen. Wiedijk (2003) vergleicht die Stärke von 15 mathematischen Beweisern. TPTP (Thousands of Problems for Theorem Provers) ist eine Bibliothek für Probleme zum Beweisen von Theoremen und geeignet, um die Leistung von Systemen zu vergleichen (Sutcliffe und Suttner, 1998; Sutcliffe et al., 2006).

Theorembeweiser haben neue mathematische Ergebnisse hervorgebracht, die menschliche Mathematiker seit Jahrzehnten umgangen haben, wie es das Buch *Automated Reasoning and the Discovery of Missing Elegant Proofs* (Wos und Pieper, 2003) ausführlich darstellt. Das Programm SAM (Semi-Automated Mathematics) war das erste, das ein Lemma in der Verbandtheorie bewies (Guard et al., 1969). Das Programm AURA hat ebenfalls offene Fragen in mehreren Bereichen der Mathematik beantwortet (Wos und Winker, 1983). Der Boyer-Moore-Theorembeweiser (Boyer und Moore, 1979) wurde von Natarajan Shankar eingesetzt, um den ersten streng formalen Beweis des Gödel-Unvollständigkeitstheorems zu erbringen (Shankar, 1986). Mit dem NUPRL-System ließen sich das Paradoxon von Girard (Howe, 1987) und das Lemma von Higman (Murthy und Russell, 1990) beweisen. Robbins schlug 1933 eine einfache Axiommenge – die **Robbins-Algebra** – vor, die scheinbar die Boolesche Algebra definiert, wofür jedoch kein Beweis gefunden werden konnte (trotz ernsthafter Arbeit von Alfred Tarski und anderen). Am 10. Oktober 1996, nach acht Tagen Rechenzeit, fand EQP (eine Variante von OTTER) einen Beweis (McCune, 1997).

Viele der frühen Artikel zur mathematischen Logik finden sich in *From Frege to Gödel: A Source Book in Mathematical Logic* (van Heijenoort, 1967). Zu den Lehrbüchern für automatisierte Deduktion gehören der Klassiker *Symbolic Logic and Mechanical Theorem Proving* (Chang und Lee, 1973) sowie die jüngeren Werke von Duffy (1991), Wos et al. (1992), Bibel (1993) und Kaufmann et al. (2000). Die führende Fachzeitschrift für das Theorembeweisen ist das *Journal of Automated Reasoning*; die Hauptkonferenzen sind die jährliche *Conference on Automated Deduction* (*CADE*) und die *International Joint Conference on Automated Reasoning* (*IJCAR*). Das *Handbook of Automated Reasoning* (Robinson und Voronkov, 2001) stellt eine Sammlung von Artikeln auf diesem Gebiet dar. Das Buch *Mechanizing Proof* (2004) von MacKenzie behandelt Geschichte und Technik von Theorembeweisen in populärwissenschaftlicher Form.

Zusammenfassung

Wir haben eine Analyse der logischen Inferenz in der Logik erster Stufe sowie mehrere Algorithmen dafür vorgestellt.

- Ein erster Ansatz verwendet Inferenzregeln (**universelle Instanziierung** und **existenzielle Instanziierung**), um das Inferenzproblem in die **Aussagenlogik zu überführen**. Dieser Ansatz ist normalerweise sehr langsam.
- Die Verwendung der **Unifikation** für die Ermittlung geeigneter Substitutionen für Variablen eliminiert den Instanziierungsschritt in Beweisen erster Stufe, sodass der Prozess in vielen Fällen effizienter wird.
- Eine geliftete Version des **Modus ponens** verwendet die Unifikation, um eine natürliche und leistungsfähige Inferenzregel bereitzustellen, den **verallgemeinerten Modus ponens**. Die Algorithmen für **Vorwärtsverkettung** und **Rückwärtsverkettung** wenden diese Regel auf Mengen definiter Klauseln an.
- Der verallgemeinerte Modus ponens ist vollständig für definite Klauseln, obwohl das Problem der logischen Konsequenz **semientscheidbar** ist. Für **Datalog**-Wissensbasen, die aus funktionsfreien definiten Klauseln bestehen, ist logische Konsequenz entscheidbar.
- Die Vorwärtsverkettung wird in **deduktiven Datenbanken** verwendet, wo sie mit Operationen von relationalen Datenbanken kombiniert werden kann. Außerdem wird sie in **Produktionssystemen** eingesetzt, die effiziente Aktualisierungen mit sehr großen Regelmengen durchführen. Die Vorwärtsverkettung ist vollständig für Datalog-Programme und wird in polynomialer Zeit ausgeführt.
- Die Rückwärtsverkettung wird in **Logikprogrammiersystemen** eingesetzt, die mit einer komplexen Compiler-Technologie arbeiten, um eine sehr schnelle Inferenz zu gewährleisten. Die Rückwärtsverkettung leidet unter redundanten Inferenzen und Endlosschleifen; sie können durch die **Memoisation** abgeschwächt werden.
- Prolog verwendet im Unterschied zur Logik erster Stufe eine geschlossene Welt mit der Annahme eindeutiger Namen und Negation als Fehler. Das macht Prolog zu einer praktischeren Programmiersprache, bringt sie aber weiter von reiner Logik weg.
- Die verallgemeinerte **Resolutions**-Inferenzregel unterstützt ein vollständiges Beweissystem für die Logik erster Stufe und verwendet dazu Wissensbasen in konjunktiver Normalform.
- Es gibt mehrere Strategien für die Reduzierung des Suchraumes eines Resolutionssystems, ohne die Vollständigkeit zu gefährden. Zu den wichtigsten Fragen gehört die Behandlung der Gleichheit; wir haben gezeigt, wie **Demodulation** und **Paramodulation** verwendet werden können.
- Effiziente resolutionsbasierte Theorembeweiser wurden verwendet, um interessante mathematische Theoreme zu beweisen sowie Software und Hardware zu verifizieren und zu synthetisieren.

Übungen zu Kapitel 9

Lösungshinweise

1 Beweisen Sie, dass die Universelle Instanziierung korrekt ist und dass die Existentielle Instanziierung eine inferentiell äquivalente Wissensbasis erzeugt.

2 Aus $Mag(Jerry, Eis)$ scheint sinnvoll abzuleiten zu sein: $\exists x \;\; Mag(x, Eis)$. Schreiben Sie eine allgemeine Inferenzregel, **Existentielle Einführung**, die diese Inferenz bestätigt. Formulieren Sie sorgfältig die Bedingungen, die von den beteiligten Variablen und Termen erfüllt sein müssen.

3 Angenommen, eine Wissensbasis enthält nur einen Satz, $\exists x \;\; SoHochWie(x, Everest)$. Welche der folgenden Ergebnisse sind legitim für die Anwendung der Existentiellen Instanziierung?

a. $SoHochWie(Everest, Everest)$
b. $SoHochWie(Kilimandscharo, Everest)$
c. $SoHochWie(Kilimandscharo, Everest) \wedge SoHochWie(BenNevis, Everest)$ (nach zwei Anwendungen)

4 Geben Sie für jedes Paar atomarer Sätze den allgemeinsten Unifikator an (falls existent):

a. $P(A, A, B)$, $P(x, y, z)$
b. $Q(y, G(A, B))$, $Q(G(x, x), y)$
c. $Älter(Vater(y), y)$, $Älter(Vater(x), Jerry)$.
d. $Kennt(Vater(y), y)$, $Kennt(x, x)$

5 Betrachten Sie den in Abbildung 9.2 gezeigten Subsumtionsverband.

a. Konstruieren Sie den Verband für den Satz $Beschäftigt(Mutter(John), Vater(Richard))$.
b. Konstruieren Sie den Verband für den Satz $Beschäftigt(IBM, y)$ („Jeder arbeitet für IBM"). Achten Sie darauf, alle Abfragearten zu berücksichtigen, die mit dem Satz unifizieren.
c. Gehen Sie davon aus, dass STORE jeden Satz unter jedem Knoten in seinem Subsumtionsverband indiziert. Erklären Sie, wie FETCH arbeiten sollte, wenn einige dieser Sätze Variablen enthalten; verwenden Sie als Beispiele die Sätze in (a) und (b) sowie die Abfrage $Beschäftigt(x, Vater(x))$.

6 Schreiben Sie die logischen Repräsentationen für die folgenden Sätze auf, sodass sie für die Verwendung mit dem Verallgemeinerten Modus ponens geeignet sind:

a. Pferde, Kühe und Schweine sind Säugetiere.
b. Ein Nachkomme eines Pferds ist ein Pferd.
c. Blaubart ist ein Pferd.
d. Blaubart ist der Vater von Charly.
e. Nachkomme und Eltern sind inverse Relationen.
f. Jedes Säugetier hat Eltern.

7 Diese Fragen betreffen die Themen Substitution und Skolemisierung.

a. Für die gegebene Prämisse $\forall x \exists y \, P(x, y)$ ist es nicht gültig zu schließen, dass $\exists q \, P(q, q)$. Geben Sie ein Beispiel für ein Prädikat P an, bei dem die erste Prämisse wahr, die zweite aber falsch ist.

b. Nehmen Sie an, dass eine Inferenzmaschine falsch geschrieben ist und der Occurs-Check fehlt, sodass sie zulässt, dass ein Literal wie $P(x, F(x))$ mit $P(q, q)$ unifiziert wird. (Wie erwähnt lassen das die meisten Standardimplementierungen von Prolog tatsächlich zu.) Zeigen Sie, dass es eine derartige Inferenzmaschine erlaubt, den Schluss $\exists\, y\, P(q, q)$ aus der Prämisse $\forall\, x\, \exists\, y\, P(x, y)$ zu ziehen.

c. Nehmen Sie an, dass eine Prozedur, die Logik erster Stufe in die Klauselform konvertiert, $\forall\, x\, \exists\, y\, P(x, y)$ falsch in $P(x, Sk0)$ skolemisiert – d.h., sie ersetzt y durch eine Skolem-Konstante statt durch eine Skolem-Funktion von x. Zeigen Sie, dass eine Inferenzmaschine, die eine derartige Prozedur verwendet, es gleichermaßen zulässt, $\exists\, q\, P(q, q)$ aus der Prämisse $\forall\, x\, \exists\, y\, P(x, y)$ abzuleiten.

d. Ein häufiger Fehler unter Studenten ist es, anzunehmen, dass man in der Unifikation einen Term für eine Skolem-Konstante statt für eine Variable einsetzen darf. Zum Beispiel würden sie sagen, dass die Formeln $P(Sk1)$ und $P(A)$ unter der Substitution $\{Sk1/A\}$ unifiziert werden können. Geben Sie ein Beispiel an, wo dies zu einer ungültigen Inferenz führt.

8 In dieser Frage geht es um Horn-Wissensbasen wie zum Beispiel die folgende:

$P(F(x)) \Rightarrow P(x)$
$Q(x) \Rightarrow P(F(x))$
$P(A)$
$Q(B)$

Es sei FC ein Algorithmus mit Tiefensuche-Vorwärtsverkettung, der wiederholt sämtliche Konsequenzen der momentan erfüllten Regeln hinzufügt. BC sei ein Algorithmus mit Tiefensuche-Rückwärtsverkettung (von links nach rechts arbeitend), der Klauseln in der Reihenfolge probiert, wie sie in der Wissensbasis angegeben sind. Welche der folgenden Aussagen sind wahr?

a. FC leitet das Literal $Q(A)$ ab.

b. FC leitet das Literal $P(B)$ ab.

c. Wenn FC ein bestimmtes Literal nicht ableiten konnte, ergibt es sich nicht als logische Konsequenz aus der Wissensbasis.

d. Für die Abfrage $P(B)$ gibt BC das Ergebnis $true$ zurück.

e. Wenn BC für ein bestimmtes Abfrageliteral nicht $true$ liefert, ergibt es sich nicht als logische Konsequenz aus der Wissensbasis.

9 Erläutern Sie, wie sich ein gegebenes 3-SAT-Problem beliebiger Größe mithilfe einer einzelnen definiten Klausel erster Stufe und nicht mehr als 30 Grundfakten schreiben lässt.

10 Nehmen Sie an, dass die folgenden Axiome gegeben sind:

1. $0 \leq 4$
2. $5 \leq 9$
3. $\forall\, x\; x \leq x$
4. $\forall\, x\; x \leq x + 0$
5. $\forall\, x\; x + 0 \leq x$
6. $\forall\, x, y\; x + y \leq y + x$
7. $\forall\, w, x, y, z\; w \leq y \wedge x \leq z \Rightarrow w + x \leq y + z$

8. $\forall\, x, y, z\ x \leq y \wedge y \leq z \Rightarrow x \leq z$
   a. Beweisen Sie den Satz $5 \leq 4 + 9$ mit Rückwärtsverkettung. (Achten Sie darauf, nur die hier angegebenen Axiome zu verwenden und nicht alles andere, was Sie über Arithmetik wissen.) Zeigen Sie nur die Schritte, die zum Erfolg führen, nicht die irrelevanten Schritte.
   b. Beweisen Sie den Satz $5 \leq 4 + 9$ mit Vorwärtsverkettung. Zeigen Sie auch hier nur die Schritte, die zum Erfolg führen.

11 Ein bekanntes Kinderrätsel lautet: „Ich habe weder Brüder noch Schwestern, aber der Vater dieses Mannes ist der Sohn meines Vaters." Verwenden Sie die Regeln aus der Verwandtschaftsdomäne (*Abschnitt 8.3.2*), um zu zeigen, wer dieser Mann ist. Sie können jede der in diesem Kapitel beschriebenen Inferenzmethoden anwenden. Warum glauben Sie, ist dieses Rätsel schwierig?

12 Angenommen, wir legen in einer logischen Wissensbasis einen Ausschnitt der amerikanischen Zensusdaten ab, die das Alter, den Wohnort, das Geburtsdatum sowie die Mutter einer Person angeben, und verwenden die Sozialversicherungsnummer als identifizierende Konstante für jede Person. Das Alter von George ist also gegeben mit $Alter(443\text{-}65\text{-}1282, 56)$. Welches der folgenden Indexschemas S1–S5 erlaubt eine effiziente Lösung für welche der Abfragen Q1–Q4 (wobei eine normale Rückwärtsverkettung vorausgesetzt wird)?

- **S1**: Ein Index für jedes Atom an jeder Position
- **S2**: Ein Index für jedes erste Argument
- **S3**: Ein Index für jedes Prädikatatom
- **S4**: Ein Index für jede *Kombination* aus Prädikat und erstem Argument
- **S5**: Ein Index für jede *Kombination* aus Prädikat und zweitem Argument und ein Index für jedes erste Argument
- **Q1**: $Alter(443\text{-}65\text{-}1282, x)$
- **Q2**: $WohntIn(x, Houston)$
- **Q3**: $Mutter(x, y)$
- **Q4**: $Alter(x, 34) \wedge WohntIn(x, KleinesDorfInAmerika)$

13 Angenommen, wir können das Problem von Variablenkonflikten in der Unifikation bei der Rückwärtsverkettung ein für alle Mal vermeiden, indem man alle Sätze in der Wissensbasis standardisiert. Zeigen Sie, dass dieser Ansatz für einige Sätze nicht funktioniert. (*Hinweis*: Betrachten Sie einen Satz, bei dem ein Teil mit einem anderen Teil unifiziert.)

14 In dieser Übung verwenden wir die Sätze, die Sie in Übung 9.6 geschrieben haben, um Fragen mithilfe eines Rückwärtsverkettungs-Algorithmus zu beantworten.

a. Zeichnen Sie den Beweisbaum, der durch einen erschöpfenden Rückwärtsverkettungs-Algorithmus für die Abfrage $\exists h \;\; Pferd(h)$ erzeugt wird, wobei die Klauseln in der angegebenen Reihenfolge verglichen werden.
b. Was können Sie zu dieser Domäne sagen?
c. Wie viele Lösungen für h folgen aus Ihren Sätzen?
d. Können Sie sich eine Methode vorstellen, sie alle zu finden? (*Hinweis*: Lesen Sie dazu Smith et al. (1986).)

15 Verfolgen Sie die Ausführung des Algorithmus für die Rückwärtsverkettung in Abbildung 9.6, um das Verbrecherproblem zu lösen. Zeigen Sie die Wertefolge, die die Variable *goals* annimmt, und ordnen Sie sie in einem Baum an.

16 Der folgende Prolog-Code definiert ein Prädikat `P`. (Denken Sie daran, dass Terme in Großbuchstaben in Prolog Variablen und keine Konstanten sind.)

```
P(X, [X|Y])
P(X, [Y|Z]) :- P(X, Z).
```

a. Zeigen Sie Beweisbäume und Lösungen für die Abfragen `P(A, [1,2,3])` und `P(2, [1,A,3])`.

b. Welche Standardlistenoperation stellt `P` dar?

17 In dieser Übung betrachten wir das Sortieren in Prolog.

a. Schreiben Sie Prolog-Klauseln, die das Prädikat `sorted(L)` definieren, das genau dann wahr ist, wenn die Liste `L` in aufsteigender Reihenfolge sortiert ist.

b. Schreiben Sie eine Prolog-Definition für das Prädikat `perm(L,M)`, das genau dann wahr ist, wenn `L` eine Permutation von `M` ist.

c. Definieren Sie `sort(L,M)` (`M` ist eine sortierte Version von `L`) unter Verwendung von `perm` und `sorted`.

d. Führen Sie `sort` für immer längere Listen aus, bis Sie die Geduld verlieren. Welche Zeitkomplexität weist Ihr Programm auf?

e. Schreiben Sie in Prolog einen schnelleren Sortieralgorithmus wie etwa Insertionsort oder Quicksort.

18 In dieser Übung betrachten wir die rekursive Anwendung der Rewrite-Regeln unter Verwendung der Logikprogrammierung. Eine Rewrite-Regel (in OTTER-Terminologie **Demodulator**) ist eine Gleichung mit einer angegebenen Richtung. Beispielsweise gibt die Regel $x+0 \rightarrow x$ vor, dass ein Ausdruck, der mit $x+0$ übereinstimmt, durch den Ausdruck x ersetzt werden soll. Die Anwendung von Rewrite-Regeln ist ein zentraler Teil von Gleichungsschlusssystemen. Wir verwenden das Prädikat `rewrite(X,Y)`, um die Rewrite-Regeln zu repräsentieren. Beispielsweise wurde die frühere Rewrite-Regel geschrieben als `rewrite(X+0,X)`. Einige Terme sind *elementar* und können nicht weiter vereinfacht werden. Wir schreiben also `elementar(0)`, um damit auszudrücken, dass 0 ein elementarer Term ist.

a. Schreiben Sie eine Definition eines Prädikates `simplify(X,Y)`, das wahr ist, wenn `Y` eine vereinfachte Version von `X` ist – d.h. wenn keine weiteren Rewrite-Regeln auf Unterausdrücke von `Y` angewendet werden können.

b. Schreiben Sie mehrere Regeln für die Vereinfachung von Ausdrücken, die arithmetische Operatoren beinhalten, und wenden Sie Ihren Vereinfachungsalgorithmus auf ein paar Beispielausdrücke an.

c. Schreiben Sie mehrere Rewrite-Regeln der symbolischen Differenzierung und verwenden Sie sie zusammen mit Ihren Vereinfachungsregeln, um Ausdrücke mit arithmetischen Ausdrücken (einschließlich Potenzierung) zu differenzieren und zu vereinfachen.

19 In dieser Übung betrachten wir die Implementierung von Suchalgorithmen in Prolog. Angenommen, `successor(X,Y)` ist wahr, wenn der Zustand `Y` ein Nachfolger von Zustand `X` ist, und `goal(X)` ist wahr, wenn `X` ein Zielzustand ist. Schreiben Sie eine Definition für `solve(X,P)`, was bedeutet, dass `P` ein Pfad (Liste von Zuständen) ist, der mit `X` beginnt, in einem Zielzustand endet und aus einer Folge erlaubter Schritte gemäß der Definition von `successor` besteht. Sie werden feststellen, dass sich das mit der Tiefensuche am einfachsten bewerkstelligen lässt. Wie einfach wäre es, eine heuristische Suchsteuerung einzufügen?

20 Es sei L die Sprache erster Stufe mit einem einzelnen Prädikat $S(p, q)$ in der Bedeutung „p rasiert q". Nehmen Sie eine Domäne von Personen an.

a. Betrachten Sie den Satz „Es gibt eine Person P, die jeden rasiert, der sich nicht selbst rasiert, und nur Personen, die sich nicht selbst rasieren." Drücken Sie dies in L aus.

b. Konvertieren Sie den Satz von (a) in eine Klausel-Form.

c. Konstruieren Sie einen Resolutionsbeweis, um zu zeigen, dass die Klauseln in (b) inhärent inkonsistent sind. (*Hinweis:* Sie benötigen keine zusätzlichen Axiome.)

21 Wie kann die Resolution genutzt werden, um zu zeigen, dass ein Satz gültig ist? Wie kann sie genutzt werden, um zu zeigen, dass ein Satz nicht erfüllbar ist?

22 Konstruieren Sie ein Beispiel für zwei Klauseln, die zusammen in zwei unterschiedlichen Weisen aufgelöst werden können, was zwei unterschiedliche Ergebnisse liefert.

23 Aus dem Satz „Schafe sind Tiere" folgt: „Der Kopf eines Schafes ist der Kopf eines Tieres." Zeigen Sie, dass diese Inferenz gültig ist, indem Sie die folgenden Schritte ausführen:

a. Übersetzen Sie die Prämisse und den Schluss in die Sprache der Logik erster Stufe. Verwenden Sie drei Prädikate: $KopfVon(h, x)$ (d.h. „h ist der Kopf von x"), $Schaf(x)$ und $Tier(x)$.

b. Negieren Sie den Schluss und wandeln Sie die Prämisse und den negierten Schluss in eine konjunktive Normalform um.

c. Verwenden Sie die Resolution, um zu zeigen, dass der Schluss aus der Prämisse folgt.

24 Nachfolgend sehen Sie zwei Sätze in der Sprache der Logik erster Stufe:

$$(\mathbf{A}): \forall x \ \exists y \ \ (x \geq y)$$

$$(\mathbf{B}): \exists y \ \forall x \ \ (x \geq y).$$

a. Angenommen, die Variablen erstrecken sich über alle natürlichen Zahlen 0, 1, 2, ..., ∞ und das Prädikat „$\geq$" bedeutet „ist größer oder gleich als". Übersetzen Sie (A) und (B) mit dieser Interpretation in die natürliche Sprache.

b. Ist (A) bei dieser Interpretation wahr?

c. Ist (B) bei dieser Interpretation wahr?

d. Ist (B) eine logische Konsequenz von (A)?

e. Ist (A) eine logische Konsequenz von (B)?

f. Versuchen Sie unter Verwendung der Resolution zu beweisen, dass (A) aus (B) folgt. Machen Sie das auch dann, wenn Sie glauben, dass (A) keine logische Konsequenz von (B) ist. Setzen Sie dies fort, bis der Beweis unterbrochen wird und Sie ihn nicht fortsetzen können (falls er unterbrochen wird). Zeigen Sie die unifizierende Substitution für jeden Resolutionsschritt. Erklären Sie, falls der Beweis fehlschlägt, wo, wie und warum er fehlgeschlagen ist.

g. Versuchen Sie nun zu beweisen, dass (B) aus (A) folgt.

25 Die Resolution kann nichtkonstruktive Beweise für Abfragen mit Variablen erzeugen, sodass wir einen speziellen Mechanismus einführen mussten, um definite Antworten zu extrahieren. Erklären Sie, warum dieses Problem nicht auftritt, wenn Wissensbasen nur definite Klauseln enthalten.

26 Wir haben in diesem Kapitel gesagt, dass die Resolution nicht für die Erzeugung aller logischen Konsequenzen einer Menge von Sätzen verwendet werden kann. Ist irgendein anderer Algorithmus dazu in der Lage?

Klassisches Planen

10

ÜBERBLICK

In diesem Kapitel zeigen wir, wie ein Agent die Struktur eines Problems nutzen kann, um komplexe Aktionspläne zu konstruieren.

Wir haben KI als das Studium rationaler Aktionen definiert, was bedeutet, dass **Planen** (oder **Planung**) – das Entwickeln eines Aktionsplanes, um ein Ziel zu erreichen – ein entscheidender Teil der KI ist. Wir haben bisher zwei Beispiele für planende Agenten gesehen: den auf einer Suche basierenden problemlösenden Agenten aus *Kapitel 3* und den hybriden logischen Agenten aus *Kapitel 7*. In diesem Kapitel führen wir eine Darstellung für Planungsprobleme ein, die sich auf Probleme ausdehnen lassen, für die die bisher gezeigten Ansätze nicht geeignet sind.

Abschnitt 10.1 entwickelt eine ausdrucksstarke und doch sorgfältig beschränkte Sprache für die Repräsentation von Planungsproblemen. *Abschnitt 10.2* zeigt, wie Vorwärts- und Rückwärtssuchalgorithmen diese Repräsentation nutzen können – primär durch exakte Heuristiken, die automatisch von der Struktur der Repräsentation abgeleitet werden können. (Das entspricht der Vorgehensweise, wie in *Kapitel 6* effektive domänenunabhängige Heuristiken für Probleme unter Rand- und Nebenbedingungen konstruiert wurden.) *Abschnitt 10.3* zeigt, wie eine Datenstruktur namens Planungsgraph die Suche nach einem Plan effizienter gestalten kann. Wir beschreiben dann einige der anderen Ansätze für Planung und vergleichen am Ende die verschiedenen Ansätze.

Dieses Kapitel behandelt vollständig beobachtbare, deterministische, statische Umgebungen mit einzelnen Agenten. Die *Kapitel 11* und *17* beschäftigen sich mit partiell beobachtbaren, stochastischen, dynamischen Umgebungen mit mehreren Agenten.

10.1 Definition der klassischen Planung

Der problemlösende Agent von *Kapitel 3* findet Aktionssequenzen, die in einem Zielzustand resultieren. Doch er hat mit atomaren Darstellungen von Zuständen zu tun und benötigt deshalb gute domänenspezifische Heuristiken, um eine gute Leistung zu erzielen. Der hybride aussagenlogische Agent von *Kapitel 7* findet Pläne ohne domänenspezifische Heuristiken, da er domänenunabhängige Heuristiken basierend auf der logischen Struktur des Problems nutzt. Allerdings stützt er sich auf grundlegende (variablenfreie) aussagenlogische Inferenz, was bedeutet, dass er bei vielen Aktionen und Zuständen überfordert sein kann. Zum Beispiel musste in der Wumpus-Welt die einfache Aktion, einen Schritt vorwärts zu gehen, für alle vier Agentenrichtungen, T Zeitschritte und n^2 aktuelle Positionen wiederholt werden.

Als Reaktion darauf haben sich Planungsforscher auf eine **faktorisierte Darstellung** festgelegt – eine Darstellung, in der ein Zustand der Welt durch eine Auflistung von Variablen repräsentiert wird. Wir verwenden eine Sprache, die als **PDDL** (Planning Domain Definition Language) bezeichnet wird und es uns erlaubt, alle $4Tn^2$ Aktionen mit nur einem Aktionsschema auszudrücken. Es gibt mehrere Versionen von PDDL; wir wählen eine einfache Version aus und ändern ihre Syntax, um mit dem Rest des Buches konsistent zu sein.[1] Wir zeigen nun, wie PDDL die vier Elemente beschreibt, die wir brauchen, um ein Suchproblem zu definieren: den Ausgangszustand, die in einem Zustand verfügbaren Aktionen, das Ergebnis bei Anwendung einer Aktion und den Zieltest.

1 PDDL wurde von der ursprünglichen Planungssprache STRIPS (Fikes und Nilsson, 1971) abgeleitet, die etwas restriktiver als PDDL ist: Die Vorbedingungen und Ziele dürfen in STRIPS keine negativen Literale enthalten.

Jeder Zustand wird als Konjunktion von Fluenten dargestellt, die grundlegende, funktionslose Atome sind. (Ein Fluent ist eine Funktion oder Relation, die sich von einer Situation zur nächsten ändern kann.) Zum Beispiel könnte $Arm \wedge Unbekannt$ den Zustand eines unglücklichen Agenten darstellen und ein Zustand in einem Paketauslieferungsproblem könnte lauten $Bei(Truck_1, Melbourne) \wedge Bei(Truck_2, Sydney)$. Es wird **Datenbanksemantik** verwendet: Die Annahme der geschlossenen Welt bedeutet, dass alle nicht erwähnten Fluenten falsch sind, und die Annahme der eindeutigen Namen heißt, dass $Truck_1$ und $Truck_2$ verschieden sind. Die folgenden Fluenten sind in einem Zustand nicht erlaubt: $Bei(x, y)$ (da er nicht grundlegend ist), $\neg Arm$ (weil er eine Negation ist) und $Bei(Vater(Fred), Sydney)$ (weil er ein Funktionssymbol verwendet). Die Darstellung von Zuständen ist sorgfältig zu konzipieren, damit man einen Zustand entweder als Konjunktion von Fluenten, die sich durch logische Inferenz manipulieren lassen, oder als Menge von Fluenten, die sich mit Mengenoperationen manipulieren lassen, behandeln kann. Der Umgang mit der **Mengensemantik** ist manchmal einfacher.

Aktionen werden durch eine Menge von Aktionsschemas beschrieben, die implizit die Funktionen ACTIONS(s) und RESULT(s, a) definieren, die für eine problemlösende Suche erforderlich sind. Wie *Kapitel 7* gezeigt hat, muss jedes System zur Aktionsbeschreibung das Rahmenproblem lösen – um zu sagen, was sich als Ergebnis der Aktion geändert hat oder was gleich geblieben ist. Klassische Planung konzentriert sich auf Probleme, bei denen die meisten Aktionen die meisten Dinge unverändert lassen. Stellen Sie sich eine Welt vor, die aus mehreren Objekten auf einer flachen Oberfläche besteht. Die Aktion, ein Objekt anzustoßen, bewirkt dann, dass das Objekt seine Position um einen Vektor Δ ändert. Eine prägnante Beschreibung der Aktion sollte lediglich Δ erwähnen und nicht alle Objekte aufführen, die an Ort und Stelle bleiben. Deshalb spezifiziert PDDL das Ergebnis einer Aktion in Bezug auf das, was sich ändert; alles, was gleich bleibt, wird nicht erwähnt.

Ein Satz von (variablenfreien) Grundaktionen lässt sich durch ein einzelnes **Aktionsschema** darstellen. Das Schema ist eine **geliftete** Darstellung – es hebt die Schlussebene von der Aussagenlogik auf eine eingeschränkte Teilmenge der Logik erster Stufe an. Das folgende Beispiel zeigt ein Aktionsschema für das Fliegen eines Flugzeuges von einem Ort zu einem anderen:

$Aktion(Fliegen(p, von, nach),$
PRECOND: $Bei(p, von) \wedge Flugzeug(p) \wedge Flughafen(von) \wedge Flughafen(nach)$
EFFECT: $\neg Bei(p, von) \wedge Bei(p, nach))$.

Das Schema besteht aus dem Aktionsnamen, einer Liste aller im Schema verwendeten Namen, einer **Vorbedingung** und einem **Effekt**. Obwohl wir noch nichts darüber gesagt haben, wie das Aktionsschema in logische Sätze konvertiert, sollten Sie sich die Variablen als allquantifiziert vorstellen. Wir können frei wählen, mit welchen Werten wir die Variablen instanziieren möchten. So zeigt das folgende Beispiel eine Grundaktion, die aus dem Substituieren von Werten für alle Variablen resultiert:

$Action(Fliegen(P1, SFO, JFK),$
PRECOND: $Bei(P1, SFO) \wedge Flugzeug(P1) \wedge Flughafen\,(SFO) \wedge Flughafen\,(JFK)$
EFFECT: $\neg Bei(P1, SFO) \wedge Bei(P1, JFK))$.

Die Vorbedingung und der Effekt einer Aktion sind jeweils Konjunktionen von Literalen (positive oder negierte atomare Sätze). Die Vorbedingung definiert die Zustände, in denen die Aktion ausgeführt werden kann, und die Wirkung definiert das Ergebnis

einer ausgeführten Aktion. Eine Aktion kann in Zustand s ausgeführt werden, wenn s eine logische Konsequenz von a ist. Die logische Konsequenz lässt sich ebenfalls mit der Mengensemantik ausdrücken: $s \models q$ gilt genau dann, wenn jedes positive Literal in q in s enthalten und jedes negierte Literal in q nicht enthalten ist. In formaler Notation sagen wir:

$$(a \in \text{ACTIONS}(s)) \Leftrightarrow s \models \text{PRECOND}(a),$$

wobei alle Variablen in a allquantifiziert sind. Zum Beispiel

$$\forall p, von, nach\ (Fliegen(p, von, nach) \in \text{ACTIONS}(s)) \Leftrightarrow \\ s \models (Bei(p, von) \wedge Flugzeug(p) \wedge Flughafen(von) \wedge Flughafen(nach)).$$

Wir sagen, dass Aktion a in Zustand s **anwendbar** ist, wenn die Vorbedingungen durch s erfüllt werden. Wenn ein Aktionsschema a Variablen enthält, kann es mehrere anwendbare Instanziierungen haben. Zum Beispiel kann mit dem in ▶ Abbildung 10.1 definierten Ausgangszustand die Aktion *Fliegen* als $Fliegen(P_1, SFO, JFK)$ oder als $Fliegen(P_2, JFK, SFO)$ instanziiert werden. Beide sind im Ausgangszustand anwendbar. Verfügt eine Aktion a über v Variablen, benötigt sie in einer Domäne mit k eindeutigen Objektnamen im ungünstigsten Fall eine Zeit von $O(V^k)$, um die anwendbaren Grundaktionen zu finden.

$Init\ (Bei(C_1, SFO) \wedge Bei(C_2, JFK) \wedge Bei(P_1, SFO) \wedge Bei(P_2, JFK)$
$\quad \wedge Fracht(C_1) \wedge Fracht(C_2) \wedge Flugzeug(P_1) \wedge Flugzeug(P_2)$
$\quad \wedge Flughafen(JFK) \wedge Flughafen(SFO))$
$Goal\ (Bei(C_1, JFK) \wedge Bei(C_2, SFO))$
$Action(Load(c, p, a),$
$\quad$ PRECOND: $Bei(c, a) \wedge Bei(p, a) \wedge Fracht(c) \wedge Flugzeug(p) \wedge Flughafen(a)$
$\quad$ EFFECT: $\neg Bei(c, a) \wedge In(c, p))$
$Action(Unload\ (c, p, a),$
$\quad$ PRECOND: $In(c, p) \wedge Bei(p, a) \wedge Fracht(c) \wedge Flugzeug(p) \wedge Flughafen(a)$
$\quad$ EFFECT: $Bei(c, a) \wedge \neg In(c, p))$
$Action(Fliegen\ (p, von, nach),$
$\quad$ PRECOND: $Bei(p, von) \wedge Flugzeug(p) \wedge Flughafen(von) \wedge Flughafen(nach)$
$\quad$ EFFECT: $\neg Bei(p, von) \wedge Bei(p, nach))$

Abbildung 10.1: Eine PDDL-Beschreibung für ein Luftfrachtplanungsproblem.

Manchmal möchten wir ein PDDL-Problem **in Aussagenlogik überführen** – jedes Aktionsschema durch eine Menge von Grundaktionen ersetzen und dann einen aussagenlogischen Solver wie zum Beispiel SATPLAN verwenden, um eine Lösung zu suchen. Allerdings ist dies unpraktisch, wenn v und k groß sind.

Das **Ergebnis** der Ausführung von Aktion a in Zustand s wird als Zustand s' definiert, dargestellt durch die Menge der Fluenten, die gebildet wird, indem wir mit s beginnend die als negative Literale in den Effekten der Aktion erscheinenden Fluenten entfernen (was wir als **Löschliste** oder DEL(a) bezeichnen) und die als positive Literale in den Effekten der Aktion erscheinenden Fluenten hinzufügen (was wir als **Hinzufügeliste** oder ADD(a) bezeichnen):

$$Result(s, a) = (s - \text{DEL}(a)) \cup \text{ADD}(a). \tag{10.1}$$

Mit der Aktion $Fliegen(P_1, SFO, JFK)$ beispielsweise würden wir $Bei(P_1, SFO)$ entfernen und $Bei(P_1, JFK)$ hinzufügen. Es ist ein Erfordernis von Aktionsschemas, dass jede

Variable im Effekt auch in der Vorbedingung erscheinen muss. Auf diese Weise werden beim Vergleich der Vorbedingung mit dem Zustand s alle Variablen gebunden und demzufolge hat RESULT(s, a) nur Grundatome. Mit anderen Worten werden Grundzustände unter der RESULT-Operation geschlossen.

Beachten Sie auch, dass die Fluenten die Zeit nicht explizit referenzieren, wie es in *Kapitel 7* geschehen ist. Dort waren hochgestellte Indizes für die Zeit erforderlich und Nachfolgerzustandsaxiome hatten die Form:

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t).$$

In PDDL sind die Zeiten und Zustände implizit in den Aktionsschemas enthalten: Die Vorbedingung verweist immer auf die Zeit t und der Effekt auf die Zeit $t + 1$.

Eine Menge von Aktionsschemas dient als Definition einer Planungs*domäne*. Um ein spezifisches *Problem* innerhalb der Domäne zu definieren, werden ein Ausgangszustand und ein Ziel hinzugefügt. Der **Ausgangszustand** ist eine Konjunktion von Grundatomen. (Wie bei allen Zuständen wird die Annahme der geschlossenen Welt verwendet. Das bedeutet, dass alle nicht erwähnten Atome falsch sind.) Das **Ziel** ist genau wie eine Vorbedingung: eine Konjunktion von (positiven oder negativen) Literalen, die Variablen enthalten können, wie zum Beispiel $Bei(p, SFO) \wedge Flugzeug(p)$. Da alle Variablen als existenzquantifiziert behandelt werden, besteht dieses Ziel darin, irgendein Flugzeug bei SFO zu haben. Das Problem ist gelöst, wenn wir eine Aktionssequenz finden können, die in einem Zustand s endet, der das Ziel zur Folge hat. Zum Beispiel hat der Zustand $Reich \wedge Berühmt \wedge Unglücklich$ das Ziel $Reich \wedge Berühmt$ zur Folge und der Zustand $Flugzeug(Flugzeug_1) \wedge Bei(Flugzeug_1, SFO)$ zieht das Ziel $Bei(p, SFO) \wedge Flugzeug(p)$ nach sich.

Jetzt haben wir Planung als Suchproblem definiert: Wir haben einen Ausgangszustand, eine Funktion ACTIONS, eine Funktion RESULT und einen Zieltest. Bevor wir effiziente Suchalgorithmen eingehend untersuchen, sehen wir uns zunächst einige Beispielprobleme an.

10.1.1 Beispiel: Luftfrachttransport

Abbildung 10.1 zeigt ein Luftfrachttransportproblem, das das Beladen und Entladen von Fracht (Cargo) und die Luftbeförderung von Ort zu Ort beinhaltet. Das Problem lässt sich mit drei Aktionen definieren: *Laden*, *Entladen* und *Fliegen*. Die Aktionen beeinflussen zwei Prädikate: $In(c, p)$ bedeutet, dass sich die Fracht c im Flugzeug p befindet, und $Bei(x, a)$ heißt, dass Objekt x (entweder Flugzeug oder Fracht) auf dem Flughafen a zu finden ist. Es ist sicherzustellen, dass die *Bei*-Prädikate ordnungsgemäß verwaltet werden. Wenn ein Flugzeug von einem Flughafen zu einem anderen fliegt, wird die gesamte Fracht im Flugzeug mitgenommen. In Logik erster Stufe wäre es leicht, über alle Objekte, die sich im Flugzeug befinden, zu quantifizieren. Doch die grundlegende PDDL besitzt keinen Allquantor, sodass wir eine andere Lösung brauchen. Wir sagen, dass ein Stück Luftfracht aufhört, *Bei* irgendwo zu sein, wenn es sich *In* einem Flugzeug befindet; die Fracht wird nur *Bei* dem neuen Flughafen sein, wenn sie entladen wird. Somit bedeutet *Bei* eigentlich: „Steht an einem bestimmten Ort zur Verfügung.“ Der folgende Plan ist eine Lösung für das Problem:

$$[Laden(C_1, P_1, SFO), Fliegen(P_1, SFO, JFK), Entladen(C_1, P_1, JFK),$$
$$Laden(C_2, P_2, JFK), Fliegen(P_2, JFK, SFO), Entladen(C_2, P_2, SFO)].$$

Schließlich gibt es noch das Problem der falschen Aktionen wie zum Beispiel *Fliegen*(P_1, *JFK*, *JFK*), die eine No-Op sein sollte, jedoch widersprüchliche Effekte hat (entsprechend der Definition würde der Effekt *Bei*(P_1, *JFK*) ∧ ¬*Bei*(P_1, *JFK*) einschließen). Es ist üblich, derartige Probleme zu ignorieren, da sie nur selten fehlerhafte Pläne verursachen. Der korrekte Ansatz ist es, Ungleichheitsvorbedingungen hinzuzufügen, die besagen, dass die *von*- und *nach*-Flughäfen unterschiedlich sein müssen.

10.1.2 Beispiel: das Ersatzreifenproblem

Betrachten Sie das Problem, einen platten Reifen zu wechseln (▶ Abbildung 10.2). Das Ziel besteht darin, einen funktionierenden Ersatzreifen richtig an der Achse des Autos zu befestigen, wobei der Ausgangszustand einen platten Reifen an der Achse und einen funktionierenden Ersatzreifen im Kofferraum hat. Um das Ganze einfach zu halten, ist unsere Version des Problems sehr abstrakt und es gibt keine festsitzenden Radmuttern und andere Komplikationen. Es gibt nur vier Aktionen: den Ersatzreifen aus dem Kofferraum holen, den platten Reifen von der Achse entfernen, den Ersatzreifen auf der Achse befestigen und das Auto über Nacht unbeaufsichtigt lassen. Wir gehen davon aus, dass sich das Auto in einer sehr kriminellen Umgebung befindet, sodass der Effekt dieser fehlenden Beaufsichtigung ist, dass die Reifen verschwinden. Eine Lösung für das Problem ist [*Entfernen*(*Platt*, *Achse*), *Entfernen*(*Ersatzreifen*, *Kofferraum*), *Befestigen*(*Ersatzreifen*, *Achse*)].

Init(*Reifen*(*Platt*) ∧ *Reifen*(*Ersatzreifen*) ∧ *Bei*(*Platt*, *Achse*)
∧ *Bei*(*Ersatzreifen*, *Kofferraum*))
Goal(*Bei*(*Ersatzreifen*, *Achse*))
Action(*Entfernen*(*obj*, *loc*),
PRECOND: *Bei*(*obj*, *loc*)
EFFECT: ¬*Bei*(*obj*, *loc*) ∧ *Bei*(*obj*, *Boden*))
Action(*Befestigen*(*t*, *Achse*),
PRECOND: *Reifen*(*t*) ∧ *Bei*(*t*, *Boden*) ∧ ¬*Bei*(*Platt*, *Achse*)
EFFECT: ¬*Bei*(*t*, *Boden*) Ù *Bei*(*t*, *Achse*))
Action(*ÜberNachtStehenlassen*,
PRECOND:
EFFECT: ¬*Bei*(*Ersatzreifen*, *Boden*) ∧ ¬*Bei*(*Ersatzreifen*, *Achse*) ∧
¬*Bei*(*Ersatzreifen*, *Kofferraum*)
∧ ¬*Bei*(*Platt*, *Boden*) ∧ ¬*Bei*(*Platt*, *Achse*) ∧ ¬*Bei*(*Platt*, *Kofferraum*))

Abbildung 10.2: Das einfache Ersatzreifenproblem.

10.1.3 Beispiel: die Blockwelt

Eine der bekanntesten Planungsdomänen ist die sogenannte **Blockwelt**. Diese Domäne besteht aus einer Menge würfelförmiger Blöcke, die auf einem Tisch stehen.[2] Die Blöcke können gestapelt werden, doch es passt immer nur ein Block direkt auf einen anderen. Ein Roboterarm kann einen Block aufheben und ihn an eine andere Position bewegen, beispielsweise auf den Tisch oder auf einen anderen Block. Der Arm kann jeweils nur

2 Die in der Planungsforschung verwendete Blockwelt ist viel einfacher als die Version von SHRDLU, die in *Abschnitt 1.3.3* gezeigt wurde.

einen Block gleichzeitig aufheben; es ist also nicht möglich, einen Block aufzuheben, auf dem sich ein anderer Block befindet. Das Ziel ist immer, einen oder mehrere Stapel aus Blöcken zu bauen, wobei vorgegeben ist, welche Blöcke sich auf welchen anderen Blöcken befinden sollen. Ein Ziel könnte beispielsweise sein, den Block A auf Block B und Block C auf Block D zu stellen (▶ Abbildung 10.4).

Wir verwenden $Auf(b, x)$, um anzuzeigen, dass sich Block b auf x befindet, wobei x entweder ein anderer Block oder der Tisch ist. Die Aktion für das Bewegen von Block b von x auf y ist $Bewegen(b, x, y)$. Eine der Vorbedingungen für das Bewegen von b ist, dass sich kein anderer Block darauf befindet. In der Logik erster Stufe wäre das $\neg\exists x\ Auf(x, b)$ oder alternativ $\forall x\ \neg Auf(x, b)$. Die grundlegende PDDL erlaubt keine Quantoren, sodass wir stattdessen ein Prädikat $Frei(x)$ einführen, das wahr ist, wenn sich auf x nichts befindet. (▶ Abbildung 10.3 zeigt die vollständige Problembeschreibung.)

$$
\begin{aligned}
&Init(Auf(A, Tisch) \wedge Auf(B, Tisch) \wedge Auf(C, A)\\
&\quad \wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Frei(B) \wedge Frei(C))\\
&Goal(Auf(A, B) \wedge Auf(B, C))\\
&Action(Bewegen(b, x, y),\\
&\quad \text{PRECOND: } Auf(b, x) \wedge Frei(b) \wedge Frei(y) \wedge Block(b) \wedge Block(y) \wedge\\
&\qquad (b{=}x) \wedge (b{=}y) \wedge (x{=}y),\\
&\quad \text{EFFECT: } Auf(b, y) \wedge Frei(x) \wedge \neg Auf(b, x) \wedge \neg Frei(y))\\
&Action(AufTischStellen(b, x),\\
&\quad \text{PRECOND: } Auf(b, x) \wedge Frei(b) \wedge Block(b) \wedge (b{=}x),\\
&\quad \text{EFFECT: } Auf(b, Tisch) \wedge Frei(x) \wedge \neg Auf(b, x))
\end{aligned}
$$

Abbildung 10.3: Ein Planungsproblem in der Blockwelt: einen Turm aus drei Blöcken bauen.
Eine Lösung ist die Sequenz [$AufTischStellen(C, A)$, $Bewegen(B, Tisch, C)$, $Bewegen(A, Tisch, B)$].

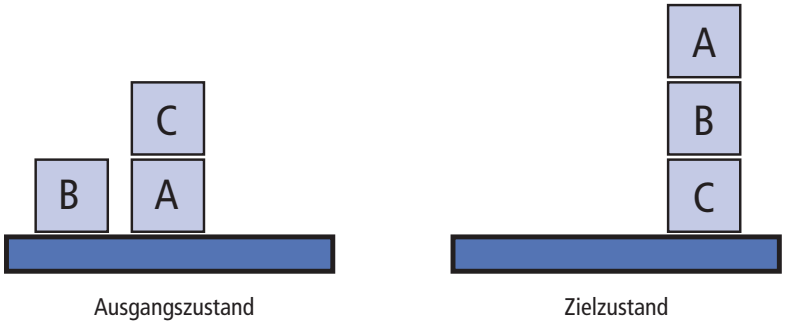


Abbildung 10.4: Diagramm des Blockweltproblems in Abbildung 10.3.

Die Aktion *Bewegen* bewegt einen Block b von x nach y, wenn sowohl b als auch y frei sind. Nach dem Bewegen ist x frei, nicht aber y. Ein erster Versuch beim *Bewegen*-Schema lautet:

$$
\begin{aligned}
&Aktion(Bewegen(b, x, y),\\
&\quad \text{PRECOND: } Auf(b, x) \wedge Frei(b) \wedge Frei(y)\\
&\quad \text{EFFECT: } Auf(b, y) \wedge Frei(x) \wedge \neg Auf(b, x) \wedge \neg Frei(y))
\end{aligned}
$$

Leider verwaltet diese Aktion *Frei* nicht korrekt, wenn x oder y der Tisch ist. Wenn $x = Tisch$, dann hat diese Aktion den Effekt $Frei(Tisch)$, aber der Tisch sollte nicht frei werden, und wenn $y = Tisch$, dann hat sie die Vorbedingung $Frei(Tisch)$, aber der

Tisch muss nicht frei sein, um einen Block darauf bewegen zu können. Um das zu korrigieren, machen wir zwei Dinge. Erstens führen wir eine weitere Aktion ein, um einen Block b von x auf den Tisch zu bewegen:

$Aktion(AufTischStellen(b, x),$
PRECOND: $Auf(b, x) \wedge Frei(b)$
EFFECT: $Auf(b, Tisch) \wedge Frei(x) \wedge \neg Auf(b, x))$

Zweitens verwenden wir die Interpretation von $Frei(x)$ als: „Es ist freier Platz auf x, um einen Block aufzunehmen." Bei dieser Interpretation ist $Frei(Tisch)$ immer wahr. Das einzige Problem ist, dass nichts den Planer daran hindert, $Bewegen(b, x, Tisch)$ statt $AufTischStellen(b, x)$ zu verwenden. Wir könnten mit diesem Problem leben – der Suchraum dafür wird größer als nötig sein, aber es entstehen keine fehlerhaften Antworten –, wir könnten aber auch das Prädikat *Block* einführen und $Block(b) \wedge Block(y)$ der Vorbedingung von *Bewegen* hinzufügen.

10.1.4 Die Komplexität der klassischen Planung

In diesem Unterabschnitt betrachten wir die theoretische Komplexität der Planung und halten zwei Entscheidungsprobleme auseinander. **PlanSAT** ist die Frage, ob es irgendeinen Plan gibt, der ein Planungsproblem löst. **Gebundenes PlanSAT** fragt, ob es eine Lösung der Länge k oder kürzer gibt; dies lässt sich heranziehen, um einen optimalen Plan zu finden.

Das erste Ergebnis ist, dass beide Entscheidungsprobleme für klassische Planung entscheidbar sind. Der Beweis folgt aus der Tatsache, dass die Anzahl der Zustände endlich ist. Doch wenn wir der Sprache Funktionssymbole hinzufügen, wird die Anzahl der Zustände unendlich und PlanSAT wird nur noch semientscheidbar: Ein Algorithmus existiert, der mit der korrekten Antwort für ein beliebiges lösbares Problem terminiert, aber bei unlösbaren Problemen eventuell nicht terminiert. Das gebundene PlanSAT-Problem bleibt selbst in der Anwesenheit von Funktionssymbolen entscheidbar. Beweise für die Behauptungen in diesem Abschnitt finden Sie in Ghallab et al. (2004).

Sowohl PlanSAT als auch gebundenes PlanSAT gehören zur Komplexitätsklasse PSCPACE; eine Klasse, die größer (und folglich schwieriger) als NP ist und auf Probleme verweist, die sich durch eine deterministische Turing-Maschine mit einem polynomiellen Platzbedarf lösen lassen. Selbst wenn wir einige recht strenge Einschränkungen machen, bleiben die Probleme ziemlich schwierig. Wenn wir zum Beispiel keine negativen Effekte zulassen, sind beide Probleme immer noch NP-hart. Erlauben wir jedoch auch keine negativen Vorbedingungen, reduziert sich PlanSAT auf die Klasse P.

Diese Ergebnisse für den ungünstigsten Fall scheinen entmutigend zu sein. Wir können uns mit der Tatsache trösten, dass Agenten üblicherweise nicht gefragt werden, Pläne für willkürliche Probleminstanzen im ungünstigsten Fall zu finden, sondern eher nach Plänen in spezifischen Domänen gefragt werden (wie zum Beispiel Blockweltprobleme mit n Blöcken), was wesentlich leichter als der theoretisch ungünstigste Fall sein kann. Für viele Domänen (einschließlich der Blockwelt und der Luftfrachtwelt) ist gebundenes PlanSAT NP-vollständig, während PlanSAT in P ist; mit anderen Worten ist optimale Planung gewöhnlich schwer, suboptimale Planung dagegen manchmal leicht. Um in Problemen, die einfacher als die Probleme im ungünstigsten Fall sind, gut abzuschneiden, brauchen wir gute Suchheuristiken. Das ist der wahre Vorteil des klassischen Planungs-

formalismus: Er hat die Entwicklung von sehr genauen domänenunabhängigen Heuristiken unterstützt, während Systemen, die auf Nachfolgerzustandsaxiomen in Logik erster Stufe basieren, weniger Erfolg beschieden war, mit guten Heuristiken aufzuwarten.

10.2 Planen mit Zustandsraumsuche

Jetzt wenden wir uns den Planungsalgorithmen zu. Wir haben gesehen, wie die Beschreibung eines Planungsproblems ein Suchproblem definiert: Wir können vom Ausgangszustand aus durch den Zustandsraum suchen und nach einem Ziel Ausschau halten. Zu den angenehmen Vorteilen der deklarativen Darstellung des Aktionsschemas gehört, dass wir auch vom Ziel aus rückwärts suchen und nach dem Ausgangszustand Ausschau halten können. ▶ Abbildung 10.5 vergleicht Vorwärts- und Rückwärtssuchen.

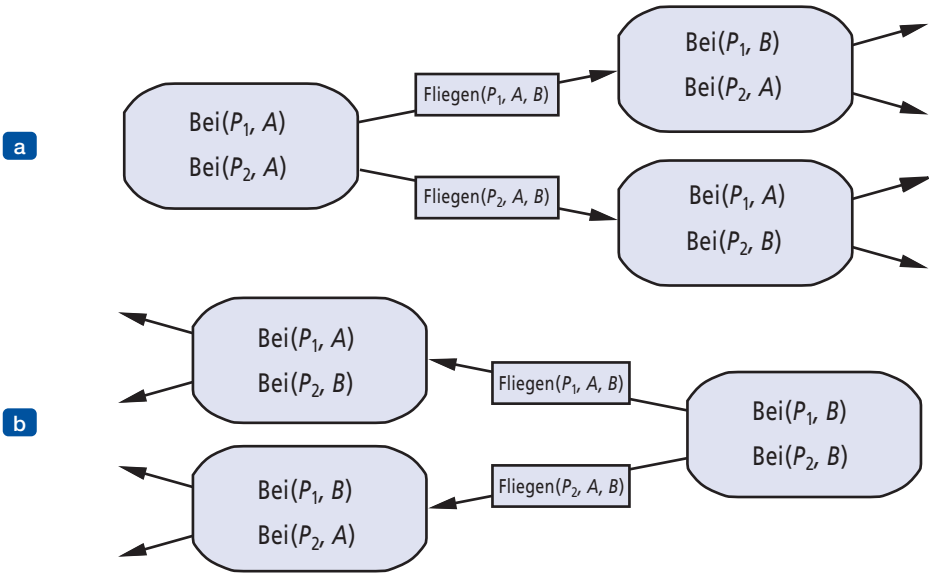


Abbildung 10.5: Zwei Ansätze für die Suche nach einem Plan. (a) Vorwärts-(Progressions-)Suche durch den Zustandsraum, die im Ausgangszustand beginnt und die Aktionen des Problems verwendet, um vorwärts nach einem Element der Menge von Zielzuständen zu suchen. (b) Rückwärts-(Regressions-)Suche durch Mengen von relevanten Zuständen, die bei der Menge von Zuständen, die das Ziel repräsentieren, beginnt und die Umkehrung der Aktionen verwendet, um rückwärts nach dem Ausgangszustand zu suchen.

10.2.1 Vorwärts-(Progressions-)Zustandsraumsuche

Nachdem wir gezeigt haben, wie sich ein Planungsproblem auf ein Suchproblem abbilden lässt, können wir Planungsprobleme mit jedem der heuristischen Suchalgorithmen von *Kapitel 3* oder einem lokalen Suchalgorithmus von *Kapitel 4* lösen (sofern wir die Aktionen, die zum Erreichen des Zieles verwendet wurden, verfolgen). Aus den frühesten Tagen der Planungsforschung (etwa 1961) bis etwa 1998 nahm man an, die Vorwärts-Zustandsraumsuche sei zu ineffizient für die praktische Anwendung. Es ist nicht schwer, Gründe dafür zu nennen.

Erstens neigt die Vorwärtssuche dazu, irrelevante Aktionen zu untersuchen. Betrachten Sie die ehrenvolle Aufgabe, ein Exemplar des Buches *KI: Ein moderner Ansatz* in einer Online-Buchhandlung zu kaufen. Nehmen Sie an, dass es ein Aktionsschema *Kaufen*(*isbn*) mit dem Effekt *Haben*(*isbn*) gibt. Bei den früher üblichen zehnstelligen ISBN-Kennzeichnungen repräsentiert dieses Aktionsschema 10 Milliarden Aktionen, um eine zu finden, die zum Ziel führt.

Zweitens haben Planungsprobleme oftmals große Zustandsräume. Betrachten Sie ein Luftfrachtproblem mit 10 Flughäfen, wo jeder Flughafen 5 Flugzeuge und 20 Frachtstücke hat. Das Ziel ist, die gesamte Fracht vom Flughafen *A* zum Flughafen *B* zu schaffen. Es gibt eine einfache Lösung für das Problem: Man lädt die 20 Frachtstücke in eines der Flugzeuge am Flughafen *A*, fliegt das Flugzeug nach *B* und entlädt die Fracht. Aber die Ermittlung der Lösung kann schwierig sein, weil der mittlere Verzweigungsfaktor riesig ist: Jedes der 50 Flugzeuge kann zu 9 anderen Flughäfen fliegen und jedes der 200 Frachtstücke kann entladen werden (falls es geladen war) oder in ein beliebiges Flugzeug auf seinem Flughafen geladen werden (falls es noch nicht geladen ist). Demnach gibt es in jedem Zustand mindestens 450 Aktionen (wenn sich alle Pakete an Flughäfen ohne Flugzeuge befinden) und maximal 10.450 (wenn alle Pakete und Flugzeuge am selben Flughafen versammelt sind). Im Durchschnitt gibt es etwa 2000 mögliche Aktionen pro Zustand, sodass der Suchgraph bis zur Tiefe der offensichtlichen Lösung etwa 2000^{41} Knoten umfasst.

Es liegt auf der Hand, dass selbst diese relativ kleine Probleminstanz ohne eine genaue Heuristik hoffnungslos ist. Obwohl sich viele Planungsanwendungen der realen Welt auf domänenspezifische Heuristiken verlassen haben, stellt sich heraus (wie *Abschnitt 10.2.3* noch zeigt), dass strenge domänenunabhängige Heuristiken automatisch hergeleitet werden können; und das ist es, was Vorwärtsverkettung praktikabel macht.

10.2.2 Rückwärts-Zustandsraumsuche

In der Rückwärtssuche beginnen wir beim Ziel und wenden Aktionen rückwärts an, bis wir eine Schrittfolge finden, die den Ausgangszustand erreicht. Diese Suche wird als Suche **relevanter Zustände** bezeichnet, da wir nur Aktionen betrachten, die für das Ziel (oder den aktuellen Zustand) relevant sind. Wie in der Belief-States-Suche (*Abschnitt 4.4*) gibt es eine Menge relevanter Zustände, die in jedem Schritt zu betrachten sind, und nicht nur einen einzelnen Zustand.

Wir beginnen mit dem Ziel, das eine Konjunktion von Literalen darstellt, die eine Beschreibung von Zustandsmengen bilden – zum Beispiel beschreibt das Ziel $\neg Arm \wedge Berühmt$ diejenigen Zustände, in denen *Arm* falsch und *Berühmt* wahr ist und jeder andere Fluent einen beliebigen Wert haben kann. Wenn eine Domäne n Grundfluenten enthält, gibt es 2^n Grundzustände (jeder Fluent kann wahr oder falsch sein), jedoch 3^n Beschreibungen von Mengen der Zielzustände (jeder Fluent kann positiv, negativ oder nicht erwähnt sein).

Im Allgemeinen funktioniert die Rückwärtssuche nur, wenn wir wissen, wie wir von einer Zustandsbeschreibung zur Beschreibung des Vorgängerzustandes zurückkommen. Zum Beispiel ist es schwer, rückwärts nach einer Lösung für das n-Damen-Problem zu suchen, weil es keine einfache Möglichkeit gibt, die Zustände zu beschreiben, die einen Zug vom Ziel entfernt sind. Erfreulicherweise wurde die PDDL-Darstellung dafür konzipiert, Aktionen zurückzugehen – wenn sich eine Domäne in PDDL ausdrücken lässt,

können wir darauf die Rückwärtssuche ausführen. Für eine gegebene Grundzielbeschreibung g und eine Grundaktion a liefert uns die Regression von g über a eine Zustandsbeschreibung g', die wie folgt definiert ist:

$$g' = (g - \text{ADD}(a)) \cup \mathit{Precond}(a)$$

Das heißt, die Effekte, die durch die Aktion hinzugefügt wurden, brauchen vorher nicht wahr gewesen zu sein und auch die Vorbedingungen müssen vorher gegolten haben, sonst hätte die Aktion nicht ausgeführt werden können. DEL(a) erscheint nicht in der Formel. Denn während wir wissen, dass die Fluenten in DEL(a) nach der Aktion nicht mehr wahr sind, wissen wir nicht, ob sie vorher wahr gewesen sind. Deshalb ist über sie nichts zu sagen.

Um alle Vorteile der Rückwärtssuche nutzen zu können, müssen wir uns mit partiell nicht instanziierten Aktionen und Zuständen und nicht nur mit den grundlegenden befassen. Nehmen wir zum Beispiel an, dass das Ziel darin besteht, ein bestimmtes Frachtstück nach SFO zu liefern: $\mathit{Bei}(C_2, \mathit{SFO})$. Dies legt die Aktion $\mathit{Entladen}(C_2, p', \mathit{SFO})$ nahe:

$\mathit{Action}(\mathit{Entladen}(C_2, p', \mathit{SFO}),$
PRECOND: $\mathit{In}(C_2, p') \wedge \mathit{Bei}(p', \mathit{SFO}) \wedge \mathit{Fracht}(C_2) \wedge \mathit{Flugzeug}(p') \wedge \mathit{Flughafen}(\mathit{SFO})$
EFFECT: $\mathit{Bei}(C_2, \mathit{SFO}) \wedge \neg \mathit{In}(C_2, p')$

(Beachten Sie die **standardisierten** Variablennamen (wobei in diesem Fall p in p' umbenannt wird), damit es keine Verwechslung von Variablennamen gibt, falls wir dasselbe Aktionsschema zweimal in einem Plan verwenden. Der gleiche Ansatz wurde in *Kapitel 9* für logische Inferenz erster Stufe verwendet.) Dies repräsentiert das Entladen der Fracht von einem nicht spezifizierten Flugzeug auf SFO; dafür kommt jedes Flugzeug infrage, doch müssen wir nicht sagen, welches von ihnen jetzt. Wir können von der Leistung der Darstellungen erster Stufe profitieren: Eine einzige Beschreibung fasst die Möglichkeit zusammen, beliebige Flugzeuge zu verwenden, indem implizit über p' quantifiziert wird. Die per Regression erhaltene Zustandsbeschreibung lautet:

$$g' = \mathit{In}(C_2, p') \wedge \mathit{Bei}(p', \mathit{SFO}) \wedge \mathit{Fracht}(C_2) \wedge \mathit{Flugzeug}(p') \wedge \mathit{Flughafen}(\mathit{SFO}).$$

Schließlich ist zu entscheiden, welche Aktionen für die Regression infrage kommen. In der Vorwärtsrichtung wählen wir Aktionen aus, die **anwendbar** sind – diejenigen Aktionen, die im Plan der nächste Schritt sein könnten. Bei der Rückwärtssuche brauchen wir Aktionen, die **relevant** sind – diejenigen Aktionen, die in einem Plan der letzte Schritt sein könnten, der zum aktuellen Zielzustand führt.

Damit eine Aktion für ein Ziel relevant sein kann, muss sie offensichtlich zum Ziel beitragen: Wenigstens einer der Effekte der Aktion (entweder positiv oder negativ) muss mit einem Element des Zieles unifizieren. Weniger klar ersichtlich ist, dass die Aktion keinen (positiven oder negativen) Effekt haben darf, der ein Element des Zieles negiert. Wenn nun das Ziel $A \wedge B \wedge C$ ist und eine Aktion den Effekt $A \wedge B \wedge \neg C$ hat, lässt sich umgangssprachlich feststellen, dass diese Aktion für das Ziel sehr relevant ist – sie bringt uns zwei Drittel des Weges dorthin. Doch im hier definierten technischen Sinn ist sie nicht relevant, weil diese Aktion nicht der letzte Schritt einer Lösung sein kann – wir brauchen immer wenigstens einen Schritt mehr, um C zu erreichen.

Bei gegebenem Ziel *Bei*(C_2, *SFO*) sind mehrere Instanziierungen von *Entladen* relevant: Wir können jedes konkrete Flugzeug zum Entladen auswählen oder wir lassen das Flugzeug unspezifiziert, indem wir die Aktion *Entladen*(C_2, *p′*, *SFO*) verwenden. Der Verzweigungsfaktor lässt sich reduzieren, ohne irgendwelche Lösungen auszuschließen, indem immer die Aktion verwendet wird, die durch Substituieren des allgemeinsten Unifizierers in das (standardisierte) Aktionsschema entsteht.

Als weiteres Beispiel betrachten wir das Ziel *Haben*(0136042597), wobei ein Ausgangszustand mit 10 Milliarden ISBNs und das einzelne Aktionsschema

$$A = \mathit{Action}(\mathit{Kaufen}(i), \text{PRECOND: } \mathit{ISBN}(i), \text{EFFECT: } \mathit{Own}(i))$$

gegeben sind.

Wie bereits erwähnt, müsste die Vorwärtssuche ohne Heuristik zunächst die 10 Milliarden *Kaufen*-Grundaktionen auflisten. Mit Rückwärtssuche würden wir aber das Ziel *Haben*(0136042597) mit dem (standardisierten) Effekt *Haben*(*i′*) unifizieren, was die Substitution $\theta = \{i'/0136042597\}$ liefert. Dann würden wir über die Aktion *Subst*(θ, *A′*) zurückgehen, um die Zustandsbeschreibung des Vorgängers *ISBN*(0136042597) zu liefern. Dies ist Teil und somit logische Konsequenz des Ausgangszustandes, sodass wir fertig sind.

Wir können dies formeller ausdrücken. Nehmen Sie eine Zielbeschreibung *g* an, die ein Zielliteral g_i enthält, und ein Aktionsschema *A*, das zu *A′* standardisiert wird. Wenn *A′* ein Effektliteral e_j' hat, wo UNIFY(g_i, e_j') = θ gilt und wo wir a' = SUBST(θ, *A′*) definieren, und wenn es keinen Effekt in a' gibt, der die Negation eines Literals in *g* darstellt, dann ist a' eine relevante Aktion gegen *g*.

Die Rückwärtssuche hält den Verzweigungsfaktor für die meisten Problemdomänen kleiner als die Vorwärtssuche. Da aber die Rückwärtssuche mit Zustandsmengen statt mit einzelnen Zuständen operiert, ist es schwerer, gute Heuristiken zu entwickeln. Das ist der Hauptgrund, warum die Mehrheit der aktuellen Systeme die Vorwärtssuche favorisiert.

10.2.3 Heuristiken für die Planung

Weder Vorwärts- noch Rückwärtssuche arbeiten ohne gute Heuristikfunktion effizient. Wie *Kapitel 3* erläutert hat, schätzt eine Heuristikfunktion $h(s)$ die Entfernung von einem Zustand *s* zum Ziel. Und wenn wir eine **zulässige** Heuristik für diese Distanz ableiten können – eine, die keine Überschätzungen vornimmt –, lassen sich mithilfe der A*-Suche optimale Lösungen finden. Um eine zulässige Heuristik abzuleiten, kann man ein **gelockertes Problem** definieren, das leichter zu lösen ist. Die genauen Kosten einer Lösung für dieses einfachere Problem werden dann zur Heuristik für das ursprüngliche Problem.

Per Definition gibt es keine Möglichkeit, einen atomaren Zustand zu analysieren, und somit bedarf es einiger Findigkeit seitens des menschlichen Analytikers, um gute domänenspezifische Heuristiken für das Suchproblem mit atomaren Zuständen zu definieren. Planung verwendet eine faktorisierte Darstellung für Zustände und Aktionsschemas. Dadurch ist es möglich, gute domänenunabhängige Heuristiken zu definieren, und Programme sind in der Lage, eine gute domänenunabhängige Heuristik für ein gegebenes Problem automatisch anzuwenden.

Stellen Sie sich ein Suchproblem als Graph vor, bei dem die Knoten Zustände und die Kanten Aktionen sind. Das Problem besteht nun darin, einen Pfad zu finden, der den Ausgangszustand mit einem Zielzustand verbindet. Es gibt zwei Wege, wie wir dieses Problem lockern können, um es einfacher zu machen: indem wir dem Graphen mehr Kanten hinzufügen, wodurch es grundsätzlich einfacher wird, einen Pfad zu finden, oder indem wir mehrere Knoten zusammen gruppieren und damit eine Abstraktion des Zustandsraumes bilden, die weniger Zustände hat und somit leichter zu durchsuchen ist.

Zuerst sehen wir uns Heuristiken an, die dem Graphen Kanten hinzufügen. Zum Beispiel entfernt die Heuristik **Vorbedingungen ignorieren** alle Vorbedingungen aus Aktionen. Jede Aktion wird in jedem Zustand anwendbar und jeder einzelne Zielfluent kann in nur einem Schritt erreicht werden (falls es eine anwendbare Aktion gibt – andernfalls ist das Problem unmöglich). Dies impliziert fast, dass die Anzahl der erforderlichen Schritte, um das gelockerte Problem zu lösen, die Anzahl der nicht erfüllten Ziele ist – fast, aber nicht ganz, weil (1) irgendeine Aktion mehrere Ziele erreichen kann und (2) manche Aktionen die Wirkungen anderer rückgängig machen können. Für viele Probleme erhält man eine genaue Heuristik, indem man (1) berücksichtigt und (2) ignoriert. Zuerst lockern wir die Aktionen, indem wir alle Vorbedingungen und alle Effekte entfernen, außer denjenigen, die Literale im Ziel sind. Dann bestimmen wir die minimale Anzahl von erforderlichen Aktionen, sodass die Vereinigungsmenge der Effekte dieser Aktionen das Ziel erfüllt. Dies ist eine Instanz des **Mengenüberdeckungsproblems**. Es gibt allerdings einen Wermutstropfen: Das Mengenüberdeckungsproblem ist NP-hart. Zum Glück gibt ein einfacher gieriger Algorithmus eine Mengenabdeckung zurück, deren Größe innerhalb eines Faktors von log n der wahren minimalen Abdeckung liegt, wobei n die Anzahl der Literale im Ziel ist. Leider verliert der gierige Algorithmus die Garantie der Zulässigkeit.

Es ist auch möglich, nur ausgewählte Vorbedingungen von Aktionen zu ignorieren. Sehen Sie sich das Schiebeblock-Puzzle (8-Puzzle oder 15-Puzzle) aus *Abschnitt 3.2* an. Wir könnten dies als Planungsproblem kodieren, das Felder mit einem einzigen Schema *Slide* umfasst:

$$\begin{aligned}&\mathit{Action}(\mathit{Slide}(t, s_1, s_2),\\ &\quad \text{PRECOND: } \mathit{Auf}(t, s_1) \wedge \mathit{Feld}(t) \wedge \mathit{Leer}(s_2) \wedge \mathit{Benachbart}(s_1, s_2)\\ &\quad \text{EFFECT: } \mathit{Auf}(t, s_2) \wedge \mathit{Leer}(s_1) \wedge \neg \mathit{Auf}(t, s_1) \wedge \neg \mathit{Leer}(s_2)).\end{aligned}$$

Wenn wir die Vorbedingungen $\mathit{Leer}(s_2) \wedge \mathit{Benachbart}(s_1, s_2)$ entfernen, kann sich jedes Feld – wie *Abschnitt 3.6* gezeigt hat – in nur einer Aktion zu einem beliebigen Leerfeld bewegen und wir erhalten die Heuristik „Anzahl falsch platzierter Felder". Entfernen wir $\mathit{Leer}(s_2)$, bekommen wir die Heuristik „Manhattan-Distanz". Es ist ohne Weiteres zu sehen, wie sich diese Heuristiken automatisch aus der Aktionsschemabeschreibung ableiten lassen. Die Leichtigkeit, mit der die Schemas manipuliert werden können, ist der große Vorteil der faktorisierten Darstellung von Planungsproblemen im Vergleich zur atomaren Darstellung von Suchproblemen.

Eine andere Möglichkeit ist die Heuristik „**Löschlisten ignorieren**". Nehmen wir für einen Moment an, dass alle Ziele und Vorbedingungen nur positive Literale enthalten.[3] Wir möchten eine gelockerte Version des ursprünglichen Problems erzeugen, die

3 Viele Probleme sind mit dieser Konvention geschrieben. Für Probleme, bei denen das nicht der Fall ist, ersetzen Sie jedes negative Literal $\neg P$ in einem Ziel oder einer Vorbedingung durch ein neues positives Literal P'.

leichter zu lösen ist und wo die Länge der Lösung als gute Heuristik dient. Dies lässt sich bewerkstelligen, indem die Löschlisten aus allen Aktionen (d.h. alle negativen Literale von Effekten) entfernt werden. Dadurch ist es möglich, monoton auf das Ziel zuzusteuern – keine Aktion wird jemals den von einer anderen Aktion gemachten Fortschritt rückgängig machen. Es zeigt sich, dass es immer noch NP-hart ist, die optimale Lösung für dieses gelockerte Problem zu finden, doch eine angenäherte Lösung lässt sich durch Hill-Climbing in polynomialer Zeit ermitteln. ▶ Abbildung 10.6 zeigt einen Teil des Zustandsraumes für zwei Planungsprobleme mithilfe der Heuristik „Löschlisten ignorieren". Die Punkte stellen Zustände und die Kanten Aktionen dar. Die Höhe jedes Punktes über der Grundebene repräsentiert den heuristischen Wert. Zustände auf der Grundebene sind Lösungen. In diesen beiden Problemen gibt es einen breiten Pfad zum Ziel. Es sind keine Sackgassen vorhanden, sodass sich kein Backtracking erforderlich macht; eine einfache Hill-Climbing-Suche findet leicht eine Lösung für diese Probleme (auch wenn es nicht unbedingt eine optimale Lösung ist).

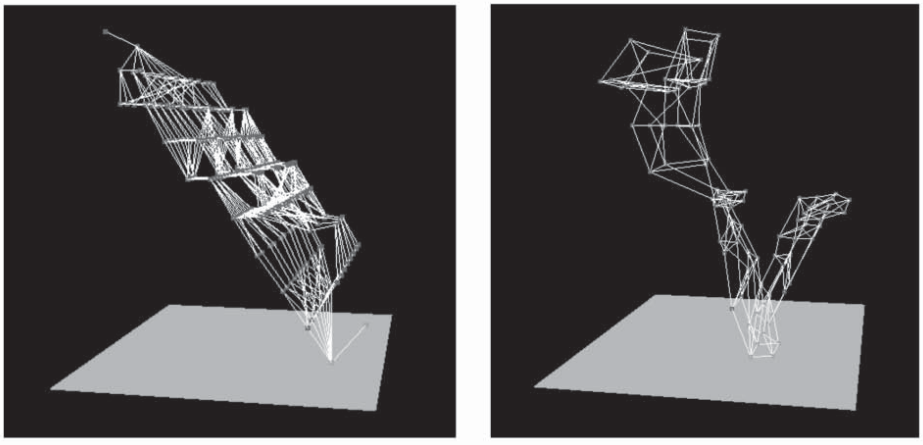


Abbildung 10.6: Die beiden Zustandsräume für Planungsprobleme mit der Heuristik „Löschlisten ignorieren". Die Höhe über der Grundebene ist der heuristische Punktewert eines Zustandes; Zustände auf der Grundebene sind Ziele. Es gibt keine lokalen Minima, sodass die Suche nach dem Ziel direkt verläuft. Nach Hoffmann (2005).

Die gelockerten Probleme hinterlassen uns ein vereinfachtes – aber trotzdem teures – Planungsproblem, nur um den Wert der Heuristikfunktion zu berechnen. Viele Planungsprobleme besitzen 10^{100} oder mehr Zustände und das Lockern der Aktionen tut nichts, um die Anzahl der Zustände zu verringern. Deshalb sehen wir uns nun Lockerungen an, die die Anzahl der Zustände verringern, indem sie eine **Zustandsabstraktion** bilden – eine n:1-Abbildung von Zuständen in der Grunddarstellung des Problems auf die abstrakte Darstellung.

Bei der einfachsten Form der Zustandsabstraktion werden einige Fluenten ignoriert. Betrachten Sie zum Beispiel ein Luftfrachtproblem mit 10 Flughäfen, 50 Flugzeugen und 200 Frachtstücken. Jedes Flugzeug kann sich auf einem der 10 Flughäfen befinden und jedes Paket kann entweder in einem der Flugzeuge verstaut oder auf einem der Flughäfen entladen sein. Es gibt also $50^{10} \times 200^{50+10} \approx 10^{155}$ Zustände. Sehen Sie sich nun ein spezielles Problem in dieser Domäne an, bei dem gerade alle Pakete auf nur 5 der Flughäfen versammelt sind und alle Pakete in einem bestimmten Flugzeug denselben Zielflughafen

haben. Eine zweckmäßige Abstraktion des Problems besteht dann darin, alle *Bei*-Fluenten zu löschen, außer denjenigen, die mit einem Flugzeug und einem Paket an jedem der 5 Flughäfen zu tun haben. Jetzt sind es nur noch $5^{10} \times 5^{5+10} \approx 10^{17}$ Zustände. Eine Lösung in diesem abstrakten Zustandsraum wird kürzer sein als eine Lösung im ursprünglichen Raum (und stellt somit eine zulässige Heuristik dar). Zudem lässt sich die abstrakte Lösung leicht zu einer Lösung für das ursprüngliche Problem erweitern (indem zusätzliche *Laden*- und *Entladen*-Aktionen hinzugefügt werden).

Ein wesentliches Konzept beim Definieren von Heuristiken ist die **Zerlegung** (**Dekomposition**): Dabei trennt man jedes Problem in einzelne Teile, löst jeden Teil unabhängig und führt dann die Teile zusammen. Die Annahme der **Teilziel-Unabhängigkeit** besagt, dass die Kosten für das Lösen einer Konjunktion von Teilzielen durch die Summe der Kosten für das unabhängige Lösen jedes Teilzieles angenähert werden. Die Annahme der Teilziel-Unabhängigkeit kann optimistisch oder pessimistisch sein. Sie ist optimistisch, wenn es negative Interaktionen zwischen den Teilplänen für jedes Teilziel gibt – zum Beispiel wenn eine Aktion in dem einen Teilplan ein durch einen anderen Teilplan erreichtes Ziel löscht. Sie ist pessimistisch und demzufolge unzulässig, wenn Teilpläne redundante Aktionen enthalten – zum Beispiel zwei Aktionen, die sich im zusammengeführten Plan durch eine einzelne Aktion ersetzen lassen.

Angenommen, das Ziel ist eine Menge von Fluenten G, die wir in disjunkte Teilmengen G_1, ..., G_n aufteilen. Wir finden dann Pläne P_1, ..., P_n, die die jeweiligen Teilziele lösen. Wie sieht eine Kostenschätzung des Planes für das Erreichen aller G aus? Die Einzelkosten $Kosten(P_i)$ können wir uns als heuristische Schätzung vorstellen. Außerdem wissen wir, dass wir immer eine zulässige Heuristik erhalten, wenn wir die Schätzungen zusammenfassen, indem wir jeweils ihre Maximalwerte heranziehen. Somit ist $\max_i \textsc{Cost}(P_i)$ zulässig und manchmal sogar genau richtig: Es könnte sein, dass P_1 zufälligerweise alle G_i erreicht. Doch in den meisten Fällen fällt die Schätzung in der Praxis zu niedrig aus. Könnten wir stattdessen die Kosten summieren? Für viele Probleme ergibt dies eine vernünftige Schätzung, doch ist sie nicht zulässig. Der beste Fall liegt vor, wenn sich ermitteln lässt, dass G_i und G_j unabhängig sind. Wenn die Effekte von P_i alle Vorbedingungen und Ziele von P_j unverändert lassen, ist die Schätzung $\textsc{Cost}(P_i) + \textsc{Cost}(P_j)$ zulässig und genauer als die max-Schätzung. In *Abschnitt 10.3.1* zeigen wir, dass Planungsgraphen helfen können, bessere heuristische Schätzungen zu liefern.

Es liegt auf der Hand, dass sich der Suchraum durch Bilden von Abstraktionen erheblich reduzieren lässt. Die Kunst besteht darin, die richtigen Abstraktionen auszuwählen und sie so einzusetzen, dass die Gesamtkosten – eine Abstraktion definieren, eine abstrakte Suche ausführen und die Abstraktion zurück auf das ursprüngliche Problem abbilden – geringer sind als die Kosten für die Lösung des ursprünglichen Problems. Die Techniken der **Musterdatenbanken** aus *Abschnitt 3.6.3* können hier nützlich sein, weil sich die Kosten für das Erstellen der Musterdatenbank über mehrere Probleminstanzen hinweg möglicherweise amortisieren.

Ein Beispiel für ein System, das effektive Heuristiken verwendet, ist FASTFORWARD, kurz FF (Hoffmann, 2005), ein den Zustandsraum vorwärts durchsuchendes Programm, das die Heuristik „Löschlisten ignorieren“ verwendet und die Heuristik mithilfe eines Planungsgraphen (siehe *Abschnitt 10.3*) abschätzt. FF verwendet dann die Heuristik im Rahmen einer Bergsteigersuche (modifiziert, damit sich der Plan verfolgen lässt), um eine Lösung zu suchen. Trifft sie auf ein Plateau oder ein lokales Maximum – wenn

keine Aktion mehr zu einem Zustand mit besserem heuristischen Punktewert führt –, wechselt FF zu einer iterativ vertiefenden Suche, bis das Programm einen besseren Zustand findet, oder gibt auf und beginnt erneut mit der Bergsteigersuche.

10.3 Planungsgraphen

Alle Heuristiken, die wir für das vollständig ordnende und das partiell ordnende Planen vorgestellt haben, können unter Ungenauigkeiten leiden. Dieser Abschnitt zeigt, wie eine spezielle Datenstruktur, der sogenannte **Planungsgraph**, verwendet werden kann, um bessere heuristische Abschätzungen zu realisieren. Diese Heuristiken können auf jede der bisher gezeigten Suchtechniken angewendet werden. Alternativ können wir mithilfe des sogenannten GRAPHPLAN-Algorithmus nach einer Lösung über dem Raum suchen, der durch den Planungsgraphen gebildet wird.

Ein Planungsproblem fragt, ob wir einen Zielzustand vom Ausgangszustand aus erreichen können. Wir nehmen an, dass ein Baum aller möglichen Aktionen vom Ausgangszustand zu den Nachfolgerzuständen und deren Nachfolgern usw. gegeben ist. Wenn wir diesen Baum zweckmäßig indizieren, lässt sich die Planungsfrage „können wir Zustand G von Zustand S_0 aus erreichen?“ unmittelbar durch einen einfachen Blick auf den Baum beantworten. Natürlich weist der Baum eine exponentielle Größe auf, sodass dieser Ansatz unpraktisch ist. Ein Planungsgraph ist für diesen Baum eine Näherung mit polynomialer Größe, der sich schnell konstruieren lässt. Zwar kann der Planungsgraph nicht definitiv beantworten, ob G von S_0 aus erreichbar ist, doch lässt sich damit abschätzen, wie viele Schritte erforderlich sind, um G zu erreichen. Die Schätzung ist immer korrekt, wenn sie meldet, dass das Ziel nicht erreichbar ist, und sie überschätzt niemals die Anzahl der Schritte. Es handelt sich also um eine zulässige Heuristik.

Ein Planungsgraph ist ein gerichteter Graph, der in Ebenen organisiert ist: Die erste Ebene S_0 für den Ausgangszustand besteht aus Knoten, die jeden Fluent darstellen, der in S_0 gültig ist; dann kommt eine Ebene A_0 mit Knoten für jede Grundaktion, die in S_0 anwendbar sein könnte; daran schließen sich abwechselnd Ebenen S_i gefolgt von A_i an, bis wir eine Endebedingung erreichen (mehr dazu später).

Grob gesagt enthält S_i alle Literale, die zur Zeit i gültig sein *könnten*, je nach den Aktionen, die in vorherigen Zeitschritten ausgeführt wurden. Ist es möglich, dass entweder P oder $\neg P$ gültig ist, werden beide in S_i dargestellt. Ebenfalls grob gesagt enthält A_i alle Aktionen, deren Vorbedingungen zur Zeit i erfüllt sein *könnten*. Hier bedeutet „grob gesagt“, dass der Planungsgraph nur eine begrenzte Untermenge der möglichen negativen Interaktionen zwischen den Aktionen aufzeichnet. Demzufolge könnte ein Literal auf Ebene S_j erscheinen, obwohl es eigentlich erst auf einer späteren Ebene – wenn überhaupt – wahr werden kann. (Ein Literal taucht niemals zu spät auf.) Trotz des möglichen Fehlers ist die Ebene j, auf der ein Literal zuerst erscheint, eine gute Schätzung, wie schwierig es ist, das Literal vom Ausgangszustand aus zu erreichen.

Planungsgraphen funktionieren nur für aussagenlogische Planungsprobleme – diejenigen, die keine Variablen enthalten. Wie bereits in *Abschnitt 10.1* erwähnt, ist es recht unkompliziert, eine Menge von Aktionsschemas in Aussagenlogik zu überführen. Trotz des resultierenden Größenzuwachses der Problembeschreibung, haben sich Planungsgraphen als effektive Werkzeuge für die Lösung schwieriger Planungsprobleme erwiesen.

▶ Abbildung 10.7 zeigt ein einfaches Planungsproblem und ▶ Abbildung 10.8 den dazu gehörenden Planungsgraphen. Jede Aktion auf Ebene A_i ist mit ihren Vorbedingungen auf S_i und ihren Effekten auf S_{i+1} verbunden. Ein Literal erscheint also aufgrund einer Aktion. Doch wir wollen auch sagen, dass ein Literal bestehen kann, wenn keine Aktion es negiert. Dies wird durch eine **Persistenzaktion** (manchmal auch *No-Op* genannt) dargestellt. Für jedes Literal C fügen wir dem Problem eine Persistenzaktion mit der Vorbedingung C und dem Effekt C hinzu. In Abbildung 10.8 zeigt Ebene A_0 eine „reale" Aktion, *Essen*(*Kuchen*), sowie zwei Persistenzaktionen, die als kleine Quadrate dargestellt sind.

Init(*Haben*(*Kuchen*))
Ziel(*Haben*(*Kuchen*) ∧ *Gegessen*(*Kuchen*))
Action(*Essen*(*Kuchen*)
  PRECOND: *Haben*(*Kuchen*)
  EFFECT: ¬*Haben*(*Kuchen*) ∧ *Gegessen*(*Kuchen*))
Action(*Backen*(*Kuchen*)
  PRECOND: ¬*Haben*(*Kuchen*)
  EFFECT: *Haben*(*Kuchen*))

Abbildung 10.7: Das Problem „Kuchen haben und Kuchen auch essen".

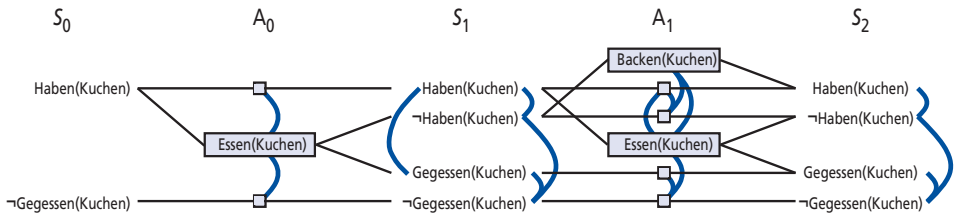


Abbildung 10.8: Der Planungsgraph für das Problem „Kuchen haben und Kuchen auch essen" bis zur Ebene S_2. Rechtecke kennzeichnen Aktionen (kleine Quadrate stehen für Persistenzaktionen) und gerade Linien kennzeichnen Vorbedingungen und Effekte. Mutex-Verknüpfungen sind als graue Bogenlinien dargestellt. Damit der Graph nicht unübersichtlich wird, zeigt die Abbildung nicht alle Mutex-Verknüpfungen. Allgemein gilt: Wenn sich zwei Literale auf S_i gegenseitig ausschließen, schließen sich die Persistenzaktionen für diese Literale auf A_i gegenseitig aus und wir brauchen diese Mutex-Verknüpfung nicht zu ziehen.

Ebene A_0 enthält alle Aktionen, die im Zustand S_0 auftreten *könnten*, zeichnet aber – was genauso wichtig ist – alle Konflikte zwischen den Aktionen auf, die verhindern, dass sie zusammen auftreten. Die grauen Linien in Abbildung 10.8 zeigen **sich gegenseitig ausschließende** (oder **Mutex**-)Verknüpfungen an. Zum Beispiel schließt sich *Essen*(*Kuchen*) gegenseitig mit der Persistenz von *Haben*(*Kuchen*) oder ¬*Gegessen*(*Kuchen*) aus. Wir werden gleich sehen, wie Mutex-Verknüpfungen berechnet werden.

Ebene S_1 enthält alle Literale, die aus der Auswahl einer beliebigen Untermenge der Aktionen in A_0 resultieren könnten. Außerdem enthält sie Mutex-Verknüpfungen (graue Linien), die Literale kennzeichnen, die nicht zusammen auftreten können, unabhängig von den ausgewählten Aktionen. Beispielsweise schließen sich *Haben*(*Kuchen*) und *Gegessen*(*Kuchen*) gegenseitig aus: Abhängig von der Auswahl der Aktionen in A_0 kann das eine oder das andere das Ergebnis sein, aber nicht beide. Mit anderen Worten repräsentiert S_1 einen Belief State: eine Menge möglicher Zustände. Die Elemente dieser Menge sind alles Teilmengen der Literale derart, dass es keine Mutex-Verknüpfung zwischen irgendwelchen Elementen der Teilmenge gibt.

Wir fahren auf diese Weise fort und wechseln zwischen der Zustandsebene S_i und der Aktionsebene A_i, bis wir einen Punkt erreichen, wo zwei aufeinanderfolgende Ebenen identisch sind. An dieser Stelle sagen wir, der Graph ist **ausgeglichen**. Der Graph in Abbildung 10.8 ist bei S_2 ausgeglichen.

Wir erhalten schließlich eine Struktur, wo jede Ebene A_i alle Aktionen enthält, die in S_i anwendbar sind, ebenso wie Bedingungen, die angeben, welche Aktionspaare nicht auf derselben Ebene zusammen ausgeführt werden können. Jede Ebene S_i enthält alle Literale, die aus jeder möglichen Auswahl von Aktionen in A_{i-1} resultieren könnten, ebenso wie die Bedingungen, die angeben, welche Literalpaare nicht möglich sind. Beachten Sie, dass es für den Aufbauprozess des Planungsgraphen *nicht erforderlich* ist, zwischen Aktionen zu wählen, was eine kombinatorische Suche zur Folge hätte. Stattdessen wird dabei nur die Unmöglichkeit bestimmter Auswahlen und Verwendung von Mutex-Verknüpfungen aufgezeichnet.

Wir definieren jetzt Mutex-Verknüpfungen sowohl für Aktionen als auch für Literale. Eine Mutex-Relation gilt zwischen zwei *Aktionen* auf einer bestimmten Ebene, wenn die folgenden drei Bedingungen gelten:

- *Inkonsistente Effekte*: Eine Aktion negiert einen Effekt einer anderen Aktion. Beispielsweise haben *Essen*(*Kuchen*) und die Persistenz von *Haben*(*Kuchen*) inkonsistente Effekte, weil sie sich nicht über den Effekt *Haben*(*Kuchen*) einig sind.
- *Interferenz*: Einer der Effekte einer Aktion ist die Negation einer Vorbedingung der anderen. Beispielsweise stört sich *Essen*(*Kuchen*) mit der Persistenz von *Haben* (*Kuchen*), weil sie ihre Vorbedingung negiert.
- *Konkurrierende Bedürfnisse*: Eine der Vorbedingungen einer Aktion schließt sich gegenseitig mit einer Vorbedingung der anderen aus. Beispielsweise schließen sich *Backen*(*Kuchen*) und *Essen*(*Kuchen*) gegenseitig aus, weil sie um den Wert der Vorbedingung *Haben*(*Kuchen*) konkurrieren.

Eine Mutex-Relation gilt zwischen zwei *Literalen* auf derselben Ebene, wenn die eine die Negation der anderen ist oder wenn jedes mögliche Paar von Aktionen, das die beiden Literale erreichen könnte, sich gegenseitig ausschließt. Diese Bedingung wird auch als *inkonsistente Unterstützung* bezeichnet. Beispielsweise schließen sich *Haben*(*Kuchen*) und *Essen*(*Kuchen*) in S_1 gegenseitig aus, weil die einzige Möglichkeit, *Haben*(*Kuchen*), also die Persistenzaktion zu erreichen, sich mit der einzigen Möglichkeit ausschließt, *Gegessen*(*Kuchen*) zu erreichen, nämlich *Essen*(*Kuchen*). In S_2 schließen sich die beiden Literale nicht gegenseitig aus, weil es neue Möglichkeiten gibt, sie zu erreichen, wie z.B. *Backen*(*Kuchen*) und die Persistenz von *Gegessen*(*Kuchen*), die sich nicht gegenseitig ausschließen.

Ein Planungsgraph ist polynomial in der Größe des Planungsproblems. Für ein Planungsproblem mit l Literalen und a Aktionen hat jede S_i nicht mehr als l Knoten und l^2 Mutex-Verknüpfungen und jede Ebene A_i nicht mehr als $a + l$ Knoten (einschließlich der No-Ops), $(a + l)^2$ Mutex-Verknüpfungen und $2(al + l)$ Vorbedingungs- und Effektverknüpfungen. Somit hat ein vollständiger Graph mit n Ebenen eine Größe von $O(n(a + l)^2)$. Die Zeit, um den Graphen zu erstellen, hat die gleiche Komplexität.

10.3.1 Planungsgraphen für heuristische Schätzung

Nachdem ein Planungsgraph erstellt wurde, ist er eine reichhaltige Informationsquelle für das Problem. Erstens ist das Problem unlösbar, wenn es irgendein Zielliteral nicht schafft, in der letzten Ebene des Graphen zu erscheinen. Zweitens können wir die Kosten, irgendein Zielliteral g_i vom Zustand s aus zu erreichen, als die Ebene, auf der g_i zuerst im Planungsgraphen erscheint, der aus dem Ausgangszustand s konstruiert wurde, abschätzen. Wir nennen dies die **Ebenenkosten** von g_i. In Abbildung 10.8 hat *Haben*(*Kuchen*) die Ebenenkosten 0 und *Gegessen*(*Kuchen*) hat die Ebenenkosten 1. Es ist einfach zu zeigen (Übung 10.10), dass diese Schätzungen für die einzelnen Ziele zulässig sind. Die Schätzung ist jedoch möglicherweise nicht sehr gut, weil Planungsgraphen auf jeder Ebene mehrere Aktionen erlauben, während die Heuristik nur die Ebene und nicht die Anzahl der Aktionen zählt. Aus diesem Grund ist es üblich, einen **seriellen Planungsgraphen** für die Berechnung der Heuristik zu verwenden. Ein serieller Graph erzwingt, dass zu jedem bestimmten Zeitschritt nur eine Aktion stattfinden kann; dazu werden sich gegenseitig ausschließende Verknüpfungen zwischen jedem Paar von Aktionen (mit Ausnahme von Persistenzaktionen) eingefügt. Die aus seriellen Graphen extrahierten Ebenenkosten sind häufig sinnvolle Schätzungen der tatsächlichen Kosten.

Um die Kosten einer Konjunktion von Zielen zu schätzen, gibt es drei einfache Ansätze. Die Heuristik der **maximalen Ebene** verwendet einfach die maximalen Ebenenkosten aller Ziele; das ist zulässig, aber nicht unbedingt exakt.

Die Heuristik der **Ebenensumme**, die der Annahme der Unabhängigkeit von Unterzielen folgt, gibt die Summe der Ebenenkosten der Ziele zurück; das ist eventuell nicht zulässig, funktioniert aber in der Praxis sehr gut für Probleme, die weitgehend zerlegbar sind. Sie ist sehr viel genauer als die Heuristik der Anzahl nicht erfüllter Ziele aus *Abschnitt 10.2*. Für unser Problem ist die heuristische Schätzung für das konjunktive Ziel *Haben*(*Kuchen*) $\wedge$ *Gegessen*(*Kuchen*) gleich $0+1 = 1$, während die korrekte Antwort 2 ist, die durch den Plan [*Essen*(*Kuchen*), *Backen*(*Kuchen*)] erreicht wird. Das sieht gar nicht so schlecht aus. Schwerer wiegt der Fehler, wenn *Backen*(*Kuchen*) nicht in der Aktionsmenge enthalten ist und die Schätzung 1 ergibt, wenn eigentlich das konjunktive Ziel unmöglich wäre.

Schließlich findet die Heuristik der **Mengenebenen** die Ebene, auf der alle Literale im konjunktiven Ziel im Planungsgraphen erscheinen, ohne dass sich irgendein Paar gegenseitig ausschließt. Diese Heuristik gibt die korrekten Werte von 2 für unser ursprüngliches Problem und ∞ für das Problem ohne *Backen*(*Kuchen*) an. Sie ist zulässig, dominiert die Heuristik der maximalen Ebene und funktioniert äußerst gut bei Aufgaben, in denen es jede Menge von Interaktionen zwischen Teilplänen gibt. Natürlich ist sie nicht perfekt; zum Beispiel ignoriert sie Interaktionen zwischen drei oder mehr Literalen.

Als Werkzeug für die Erzeugung exakter Heuristiken können wir den Planungsgraphen als gelockertes Problem betrachten, das effizient lösbar ist. Um die Natur des gelockerten Problems zu verstehen, müssen wir genau verstehen, was es bedeutet, wenn ein Literal g auf der Ebene S_i im Planungsgraphen erscheint. Im Idealfall wünschen wir uns eine Garantie, dass es einen Plan mit i Aktionsebenen gibt, der g erreicht, und außerdem, dass es keinen solchen Plan gibt, wenn g nicht erscheint. Leider ist es schwierig, eine solche Garantie zu geben – genauso schwierig wie die Lösung des ursprünglichen Planungsproblems. Der Planungsgraph bildet die zweite Hälfte der Garantie (wenn g nicht erscheint, gibt es keinen Plan), aber wenn g erscheint, versprechen alle Planungsgraphen, dass es einen Plan gibt, der *möglicherweise* g erzielt und keine „offensichtli-

chen“ Fehler hat. Ein offensichtlicher Fehler ist definiert als ein Fehler, der sich erkennen lässt, indem man zwei Aktionen oder zwei Literale gleichzeitig betrachtet – mit anderen Worten, indem man die Mutex-Relationen betrachtet. Es könnte subtilere Fehler geben, an denen drei, vier oder mehr Aktionen beteiligt sind, aber die Erfahrung hat gezeigt, dass es den Rechenaufwand nicht wert ist, diese möglichen Fehler zu suchen. Das ähnelt der Lektion, die wir für die CSPs gelernt haben, nämlich dass es häufig sinnvoll ist, die 2-Konsistenz zu berechnen, bevor man nach einer Lösung sucht, dass es jedoch selten den Aufwand wert ist, die 3-Konsistenz oder eine höhere Konsistenz zu berechnen (siehe *Abschnitt 6.2.4*).

Ein Beispiel für ein unlösbares Problem, das sich als solches durch einen Planungsgraphen nicht erkennen lässt, ist das Problem der Blockwelt, wo das Ziel darin besteht, Block A auf B, B auf C und C auf A zu legen. Dies ist ein unmögliches Ziel; ein Turm, bei dem der Fuß auf der Spitze der Spitze liegt. Doch ein Planungsgraph kann die Unmöglichkeit nicht erkennen, weil zwei der drei Teilziele erreichbar sind. Die einzelnen Literalpaare schließen sich nicht gegenseitig aus, das ist nur für die drei als Ganzes der Fall. Um zu erkennen, dass dieses Problem unmöglich ist, müssten wir den Planungsgraphen durchsuchen.

10.3.2 Der Graphplan-Algorithmus

Dieser Unterabschnitt zeigt, wie man einen Plan direkt aus dem Planungsgraphen ableitet, anstatt den Graphen nur zu verwenden, um eine Heuristik zu ermitteln. Der GRAPHPLAN-Algorithmus (▶ Abbildung 10.9) fügt wiederholt mit EXPAND-GRAPH eine Ebene in den Planungsgraphen ein. Nachdem alle Ziele als sich nicht gegenseitig ausschließend im Graphen erscheinen, ruft GRAPHPLAN die Funktion EXTRACT-SOLUTION auf, um nach einem Plan zu suchen, der das Problem löst. Falls dies scheitert, erweitert der Algorithmus eine weitere Ebene und versucht es erneut, wobei mit einem Fehler geschlossen wird, wenn es keinen Grund mehr für eine Fortsetzung gibt.

```
function GRAPHPLAN(problem) returns Lösung oder Fehler

  graph ← INITIAL-PLANNING-GRAPH(problem)
  goals ← CONJUNCTS(problem.GOAL)
  nogoods ← eine leere Hashtabelle
  for tl = 0 to ∞ do
    if goals alle nicht sich gegenseitig ausschließend in S_tl von graph then
       solution ← EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
      if solution ≠ failure then return solution
    if graph und nogoods beide ausgeglichen sind then return failure
    graph ← EXPAND-GRAPH(graph, problem)
```

Abbildung 10.9: Der GRAPHPLAN-Algorithmus ruft EXPAND-GRAPH auf, um eine Ebene hinzuzufügen, bis entweder durch EXTRACT-SOLUTION eine Lösung gefunden wird oder keine Lösung möglich ist.

Jetzt betrachten wir die Anwendung von GRAPHPLAN auf das Ersatzreifenproblem aus *Abschnitt 10.1.2*. ▶ Abbildung 10.10 zeigt den vollständigen Graphen. Die erste Zeile von GRAPHPLAN initialisiert den Planungsgraphen mit einem Graphen, der eine Ebene umfasst (S_0) und den Ausgangszustand verkörpert. Die positiven Fluenten aus dem Ausgangszustand der Problembeschreibung werden genauso gezeigt wie die relevanten negativen Fluenten. Nicht dargestellt sind die unverändert bleibenden positiven Literale (wie zum Beispiel *Reifen*(*Ersatzreifen*)) und die irrelevanten negativen Lite-

rale. Das Ziel *Bei*(*Ersatzreifen*, *Achse*) ist in S_0 nicht vorhanden, sodass wir EXTRACT-SOLUTION nicht aufrufen müssen – wir sind sicher, dass es noch keine Lösung gibt. Stattdessen fügt EXPAND-GRAPH in A_0 die drei Aktionen hinzu, deren Vorbedingungen auf Ebene S_0 existieren (d.h. alle Aktionen außer *Befestigen*(*Ersatzreifen*, *Achse*)), ebenso wie die Persistenzaktionen für alle Literale in S_0. Die Effekte der Aktionen werden in Ebene S_1 hinzugefügt. EXPAND-GRAPH überprüft dann auf Mutex-Relationen und fügt sie dem Graphen hinzu.

Da *Bei*(*Ersatzreifen*, *Achse*) immer noch nicht in S_1 vorhanden ist, rufen wir EXTRACT-SOLUTION wieder nicht auf. Wir rufen erneut EXPAND-GRAPH auf und fügen A_1 und S_1 hinzu, sodass wir zu dem in Abbildung 10.10 dargestellten Planungsgraphen kommen. Jetzt haben wir alle Aktionen vorliegen und es lohnt sich, einige Beispiele von Mutex-Beziehungen und ihre Ursachen zu betrachten:

- *Inkonsistente Effekte*: *Entfernen*(*Ersatzreifen*, *Kofferraum*) schließt sich mit *ÜberNachtStehenlassen* aus, weil die eine Aktion den Effekt *Bei*(*Ersatzreifen*, *Boden*) hat und die andere seine Negation.
- *Interferenz*: *Entfernen*(*Platt*, *Achse*) schließt sich mit *ÜberNachtStehenlassen* aus, weil die eine Aktion die Vorbedingung *Bei*(*Platt*, *Achse*) und die andere ihre Negation als Effekt hat.
- *Konkurrierende Bedürfnisse*: *Befestigen*(*Ersatzreifen*, *Achse*) schließt sich mit *Entfernen*(*Platt*, *Achse*) aus, weil die eine Aktion die Vorbedingung *Bei*(*Platt*, *Achse*) und die andere ihre Negation hat.
- *Inkonsistente Unterstützung*: *Bei*(*Ersatzreifen*, *Achse*) schließt sich mit *Bei*(*Platt*, *Achse*) in S_2 aus, weil die einzige Möglichkeit, *Bei*(*Ersatzreifen*, *Achse*) zu erreichen, *Befestigen*(*Ersatzreifen*, *Achse*) ist, was sich mit der Persistenzaktion ausschließt, die die einzige Möglichkeit darstellt, *Bei*(*Platt*, *Achse*) zu erreichen. Damit erkennen die Mutex-Relationen den unmittelbaren Konflikt, der aus dem Versuch entsteht, zwei Objekte gleichzeitig am selben Platz unterzubringen.

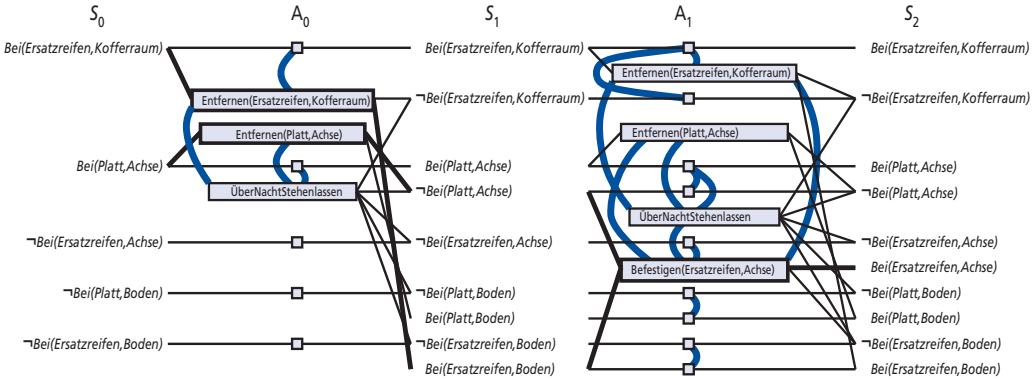


Abbildung 10.10: Der Planungsgraph für das Ersatzreifenproblem nach der Expansion auf Ebene S_2. Mutex-Verknüpfungen sind als graue Linien dargestellt. Es sind nicht alle Verknüpfungen eingezeichnet, weil der Graph sonst zu unübersichtlich wird. Die Lösung ist durch fette Linien und Rahmen gekennzeichnet.

Wenn wir jetzt an den Anfang der Schleife zurückgehen, sind alle Literale vom Ziel in S_2 vorhanden und keines davon schließt sich mit einem anderen aus. Das bedeutet, es könnte eine Lösung existieren und EXTRACT-SOLUTION versucht, sie zu finden. Wir kön-

nen EXTRACT-SOLUTION als boolesches CSP formulieren, dessen Variablen die Aktionen auf jeder Ebene sind, die Werte für jede Variable befinden sich *innerhalb* oder *außerhalb* des Planes und die Beschränkungen werden durch die Mutexe und das Bedürfnis, jedes Ziel und jede Vorbedingung zu erfüllen, verkörpert.

Alternativ können wir EXTRACT-SOLUTION als Rückwärtssuchproblem definieren, wobei jeder Zustand in der Suche einen Zeiger auf eine Ebene im Planungsgraphen und eine Menge nicht erfüllter Ziele enthält. Wir definieren dieses Suchproblem wie folgt:

- Der Ausgangszustand ist die letzte Ebene des Planungsgraphen S_n, zusammen mit der Menge der Ziele aus dem Planungsproblem.
- Die in einem Zustand auf Ebene S_i verfügbaren Aktionen sollen eine konfliktfreie Untermenge der Aktionen in A_{i-1} auswählen, deren Effekte die Ziele in dem Zustand abdecken. Der resultierende Zustand hat die Ebene S_{i-1} und als Menge der Ziele die Vorbedingungen für die ausgewählte Aktionsmenge. Mit „konfliktfrei" meinen wir eine Menge von Aktionen, worin sich jeweils zwei der Aktionen nicht gegenseitig ausschließen und worin sich auch nicht zwei ihrer Vorbedingungen gegenseitig ausschließen.
- Das Ziel ist, einen Zustand auf der Ebene S_0 zu erreichen, sodass alle Ziele erfüllt sind.
- Die Kosten für jede Aktion betragen 1.

Für dieses spezielle Problem beginnen wir in S_2 mit dem Ziel *Bei*(*Ersatzreifen*, *Achse*). Die einzige Wahl, die wir zur Erreichung der Zielmenge haben, ist *Befestigen* (*Ersatzreifen*, *Achse*). Damit gelangen wir zu einem Suchzustand in S_1 mit den Zielen *Bei*(*Ersatzreifen*, *Boden*) und ¬*Bei*(*Platt*, *Achse*). Das erste Ziel kann nur durch *Entfernen*(*Ersatzreifen*, *Kofferraum*) erzielt werden, das zweite durch *Entfernen*(*Platt*, *Achse*) oder durch *ÜberNachtStehenlassen*. Allerdings schließt sich *ÜberNachtStehenlassen* mit *Entfernen*(*Ersatzreifen*, *Kofferraum*) aus, sodass die einzige Lösung *Entfernen*(*Ersatzreifen*, *Kofferraum*) und *Entfernen*(*Platt*, *Achse*) ist. Das bringt uns zu einem Suchzustand S_0 mit den Zielen *Bei*(*Ersatzreifen*, *Kofferraum*) und *Bei*(*Platt*, *Achse*). Beide Aktionen sind in diesem Zustand vorhanden, sodass wir eine Lösung haben: die Aktionen *Entfernen*(*Ersatzreifen*, *Kofferraum*) und *Entfernen*(*Platt*, *Achse*) in Ebene A_0, gefolgt von *Befestigen*(*Ersatzreifen*, *Achse*) in A_1.

Falls EXTRACT-SOLUTION scheitert, eine Lösung für eine Menge von Zielen auf einer bestimmten Ebene zu finden, zeichnen wir das (*level*, *goals*)-Paar als **No-good** auf, wie wir es beim Lernen von Beschränkungen für CSPs getan haben (*Abschnitt 6.3.3*). Wird EXTRACT-SOLUTION erneut mit derselben Ebene und denselben Zielen aufgerufen, können wir das aufgezeichnete No-good finden und sofort einen Fehler zurückgeben, anstatt erneut zu suchen. In Kürze zeigen wir, dass No-goods auch im Endetest verwendet werden.

Wir wissen, dass das Planen PSPACE-vollständig ist und der Aufbau des Planungsgraphen eine polynomiale Zeit dauert. Die Lösungsextrahierung wird also im ungünstigsten Fall nicht handhabbar sein. Aus diesem Grund brauchen wir heuristische Anhaltspunkte, um während der Rückwärtssuche zwischen den Aktionen zu wählen. Ein Ansatz, der in der Praxis sehr gut funktioniert, ist ein „gieriger" Algorithmus, der auf den Ebenenkosten für die Literale basiert. Wir gehen für jede Zielmenge in der folgenden Reihenfolge vor:

1. Wähle das erste Literal mit den höchsten Ebenenkosten aus.
2. Um dieses Literal zu erreichen, bevorzuge Aktionen mit einfacheren Vorbedingungen. Das heißt, wähle eine Aktion aus, sodass die Summe (oder das Maximum) der Ebenenkosten ihrer Vorbedingungen am kleinsten ist.

10.3.3 Terminierung von Graphplan

Bisher haben wir die Frage der Terminierung ausgeklammert. Hier zeigen wir, dass GRAPHPLAN tatsächlich terminiert und einen Fehler zurückgibt, wenn es keine Lösung gibt.

Zunächst ist zu klären, warum wir die Erweiterung des Graphen nicht anhalten können, sobald er ausgeglichen ist. Betrachten Sie eine Luftfrachtdomäne mit einem Flugzeug und n Frachtstücken auf Flughafen A, die alle Flughafen B als Ziel haben. In dieser Version des Problems passt nur jeweils ein Frachtstück in das Flugzeug. Der Graph wird bei Ebene 4 ausgeglichen sein, was die Tatsache widerspiegelt, dass wir jedes Frachtstück in drei Schritten beladen, mit dem Flugzeug transportieren und am Ziel entladen können. Das bedeutet aber nicht, dass sich eine Lösung aus dem Graphen auf Ebene 4 extrahieren lässt; in der Tat erfordert eine Lösung $4n - 1$ Schritte: Für jedes Frachtstück müssen wir laden, fliegen und entladen und für alle außer dem letzten Stück müssen wir zurück zu Flughafen A fliegen, um das nächste Stück zu holen.

Wie lange müssen wir mit dem Erweitern fortfahren, nachdem der Graph ausgeglichen ist? Wenn die Funktion EXTRACT-SOLUTION keine Lösung findet, muss es zumindest eine Menge von Zielen geben, die nicht erreichbar waren und als No-good markiert wurden. Wenn es also möglich ist, dass es weniger No-goods in der nächsten Ebene gibt, sollten wir fortfahren. Sobald der Graph selbst und die No-goods ausgeglichen sind und keine Lösung gefunden wurde, können wir mit Fehler terminieren, weil es keine Möglichkeit einer darauffolgenden Änderung gibt, die eine Lösung beisteuern könnte.

Jetzt müssen wir lediglich noch beweisen, dass der Graph und die No-goods immer ausgeglichen werden. Der Schlüssel zu diesem Beweis ist, dass bestimmte Eigenschaften von Planungsgraphen monoton steigen oder fallen. Hierbei bedeutet „X steigt monoton“, dass die Menge der X auf Ebene $i + 1$ eine Obermenge (nicht unbedingt eine echte Obermenge) der Menge auf der Ebene i ist. Die Eigenschaften sehen wie folgt aus:

- *Literale steigen monoton*: Nachdem ein Literal auf einer bestimmten Ebene erschienen ist, erscheint es auf allen folgenden Ebenen. Das liegt an den Persistenzaktionen; nachdem ein Literal aufgetreten ist, bewirken die Persistenzaktionen, dass es für immer bleibt.
- *Aktionen steigen monoton*: Nachdem eine Aktion auf einer bestimmten Ebene erschienen ist, erscheint sie auch auf allen nachfolgenden Ebenen. Das liegt am Steigen der Literale; wenn die Vorbedingungen einer Aktion auf einer Ebene erscheinen, erscheinen sie auch auf allen folgenden Ebenen und ebenso die Aktion.
- *Mutexe fallen monoton*: Wenn zwei Aktionen sich auf einer bestimmten Ebene A_i gegenseitig ausschließen, dann schließen sie sich auch für alle *vorhergehenden* Ebenen aus, auf denen sie beide auftreten. Das Gleiche gilt für Mutexe zwischen Literalen. In den Abbildungen ist das nicht immer zu sehen, da sie vereinfacht sind: Sie zeigen keine Literale, die nicht auf der Ebene S_i gelten können, und auch keine Aktionen, die

nicht auf der Ebene A_i ausgeführt werden können. Es ist zu sehen, dass „Mutexe fallen monoton“ wahr ist, wenn man berücksichtigt, dass diese unsichtbaren Literale und Aktionen sich mit allem ausschließen.

Der Beweis lässt sich durch Fälle behandeln: Wenn sich die Aktionen A und B auf der Ebene A_i gegenseitig ausschließen, muss einer von drei möglichen Mutex-Typen dafür verantwortlich sein. Die beiden ersten, inkonsistente Effekte und Interferenz, sind Eigenschaften der eigentlichen Aktionen. Wenn sich die Aktionen also auf A_i gegenseitig ausschließen, schließen sie sich auf jeder Ebene gegenseitig aus. Der dritte Fall, konkurrierende Bedürfnisse, ist von Bedingungen auf der Ebene S_i abhängig: Diese Ebene muss eine Vorbedingung von A enthalten, die sich mit einer Vorbedingung von B ausschließt. Diese beiden Vorbedingungen können sich gegenseitig ausschließen, wenn sie Negationen voneinander sind (dann würden sie sich auf jeder Ebene gegenseitig ausschließen) oder wenn alle Aktionen für das Erreichen der einen Ebene alle Aktionen für das Erreichen der anderen Ebene ausschließen. Wir wissen jedoch bereits, dass die verfügbaren Aktionen steigend monoton sind; deshalb müssen aufgrund der Induktion Mutexe fallen.

- *No-goods fallen monoton*: Wenn eine Menge von Zielen auf einer bestimmten Ebene nicht erreichbar ist, sind die Ziele in keiner *vorherigen* Ebene erreichbar. Der Beweis ergibt sich durch Widerspruch: Wenn sie auf irgendeiner vorherigen Ebene erreichbar wären, könnten wir einfach Persistenzaktionen hinzufügen, um sie auf einer nachfolgenden Ebene erreichbar zu machen.

Da die Aktionen und Literale monoton steigen und da es nur eine endliche Anzahl von Aktionen und Literalen gibt, muss eine Ebene kommen, die die gleiche Anzahl von Mutexen und No-goods wie die vorherige Ebene hat. Da Mutexe und No-goods fallen und da es niemals weniger als null Mutexe oder No-goods geben kann, muss eine Ebene kommen, die die gleiche Anzahl von Mutexen und No-goods wie die vorherige Ebene hat. Nachdem ein Graph diesen Zustand erreicht hat und dann eines der Ziele fehlt oder sich gegenseitig mit einem anderen Ziel ausschließt, können wir den GRAPHPLAN-Algorithmus beenden und einen Fehler zurückgeben. Damit ist der Beweis kurz skizziert; für weitere Details sei auf Ghallab et al. (2004) verwiesen.

10.4 Andere klassische Planungskonzepte

Zu den derzeit bekanntesten und effizientesten Konzepten für vollständig automatisierte Planung gehören:

- Übersetzen eines booleschen Erfüllbarkeitsproblems (SAT)
- Vorwärtszustandsraumsuche mit sorgfältig erarbeiteten Heuristiken (*Abschnitt 10.2*)
- Suche mithilfe eines Planungsgraphen (*Abschnitt 10.3*)

Diese drei Konzepte sind nicht die einzigen, die in der 40-jährigen Geschichte der automatisierten Planung ausprobiert wurden. ▶ Abbildung 10.11 zeigt einige der Spitzensysteme in den Internationalen Planungswettbewerben, die seit 1998 alle zwei Jahre abgehalten werden. In diesem Abschnitt beschreiben wir zuerst die Übersetzung eines Erfüllbarkeitsproblems und gehen dann auf drei andere einflussreiche Konzepte ein: Planen als logische Deduktion erster Stufe, als Bedingungserfüllung (Constraint Satisfaction) und als Planverfeinerung.

Jahr	Track	Gewinnende Systeme (Ansätze)
2008	Optimal	GAMER (Model Checking, bidirektionale Suche)
2008	Akzeptabel	LAMA (schnelle Abwärtssuche mit FF-Heuristik)
2006	Optimal	SATPLAN, MAXPLAN (boolesche Erfüllbarkeit)
2006	Akzeptabel	SGPLAN (Vorwärtssuche; partitioniert in unabhängige Subprobleme)
2004	Optimal	SATPLAN (boolesche Erfüllbarkeit)
2004	Akzeptabel	FAST DIAGONALLY DOWNWARD (Vorwärtssuche mit kausalen Graphen)
2002	Automatisiert	LPG (lokale Suche, in CSPs konvertierte Planungsgraphen)
2002	Handkodiert	TLPLAN (zeitliche Aktionslogik mit Steuerungsregeln für Vorwärtssuche)
2000	Automatisiert	FF (Vorwärtssuche)
2000	Handkodiert	TALPLANNER (zeitliche Aktionslogik mit Steuerungsregeln für Vorwärtssuche)
1998	Automatisiert	IPP (Planungsgraphen); HSP (Vorwärtssuche)

Abbildung 10.11: Einige Spitzensysteme im Internationalen Planungswettbewerb (International Planning Competition). Jedes Jahr gibt es verschiedene Tracks: „Optimal" bedeutet, dass die Planer den kürzest möglichen Plan erzeugen müssen, während „Akzeptabel" heißt, dass nichtoptimale Lösungen akzeptiert werden. Mit „Handkodiert" ist gemeint, dass domänenspezifische Heuristiken erlaubt sind; bei „Automatisiert" sind diese nicht zugelassen.

10.4.1 Klassisches Planen als boolesche Erfüllbarkeit

Abschnitt 7.7.4 hat gezeigt, wie SATPLAN Planungsprobleme löst, die in Aussagenlogik ausgedrückt sind. Jetzt zeigen wir, wie sich eine PDDL-Beschreibung in eine Form übersetzen lässt, die durch SATPLAN verarbeitet werden kann. Die Übersetzung ist eine Folge unkomplizierter Schritte:

- *Die Aktionen in Aussagenlogik überführen:* Ersetze jedes Aktionsschema durch eine Menge von Grundaktionen, indem Konstanten für die einzelnen Variablen substituiert werden. Diese Grundaktionen sind nicht Teil der Übersetzung, werden aber in darauffolgenden Schritten verwendet.
- *Den Ausgangszustand definieren:* Behaupte F^O für jeden Fluenten F im Ausgangszustand des Problems und $\neg F^O$ für jeden Fluenten, der im Ausgangszustand nicht erwähnt ist.
- *Das Ziel in Aussagenlogik überführen:* Ersetze für jede Variable im Ziel die Literale, die die Variable enthalten, durch eine Disjunktion über Konstanten. Zum Beispiel würde das Ziel, einen Block A auf einem anderen Block zu haben, $Auf(A, x) \wedge Block(x)$ in einer Welt mit den Objekten A, B und C durch das Ziel

  $$(Auf(A, A) \wedge Block(A)) \vee (Auf(A, B) \wedge Block(B)) \vee (Auf(A, C) \wedge Block(C))$$

  ersetzt.
- *Nachfolgerzustandsaxiome hinzufügen:* Füge für jeden Fluenten F ein Axiom der Form

  $$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t)$$

  hinzu, wobei *ActionCausesF* eine Disjunktion aller Grundaktionen ist, die F in ihrer Hinzufügeliste haben, und *ActionCausesNotF* eine Disjunktion aller Grundaktionen ist, die F in ihrer Löschliste haben.

- *Vorbedingungsaxiome hinzufügen:* Füge für jede Grundaktion A das Axiom $Bei \Rightarrow \text{PRE}(A)^t$ hinzu, d.h. wenn eine Aktion zur Zeit t unternommen wird, müssen die Vorbedingungen wahr gewesen sein.
- *Aktionsausschlussaxiome hinzufügen:* Stelle fest, dass jede Aktion von jeder anderen Aktion verschieden ist.

Die resultierende Übersetzung liegt in der Form vor, die wir an SATPLAN übergeben können, um eine Lösung zu suchen.

10.4.2 Planen als logische Deduktion erster Stufe: Situationskalkül

PDDL ist eine Sprache, die sorgfältig die Ausdrucksstärke der Sprache mit der Komplexität der darauf operierenden Algorithmen abstimmt. Dennoch bleiben einige Probleme, die in PDDL nur schwer auszudrücken sind. Zum Beispiel lässt sich das Ziel „bewege sämtliche Fracht von A nach B unabhängig davon, wie viele Frachtstücke es gibt“ in PDDL nicht ausdrücken, doch in Logik erster Stufe ist dies mithilfe eines Allquantors möglich. In ähnlicher Weise kann Logik erster Stufe globale Beschränkungen wie zum Beispiel „nicht mehr als vier Roboter können sich am selben Ort zur selben Zeit befinden“ prägnant ausdrücken. PDDL kann dies nur mit wiederholten Vorbedingungen auf jeder möglichen Aktion sagen, die mit einer Bewegung zu tun hat.

Die aussagenlogische Darstellung von Planungsproblemen hat auch Grenzen, wie zum Beispiel die Tatsache, dass das Konzept der Zeit direkt an Fluenten gebunden ist. Zum Beispiel bedeutet $Süd^2$, dass „der Agent zur Zeit 2 nach Süden blickt“. Mit dieser Darstellung gibt es keine Möglichkeit zu sagen: „Der Agent würde zur Zeit 2 nach Süden blicken, wenn er zur Zeit 1 eine Rechtsdrehung ausgeführt hat; andernfalls würde er nach Osten blicken.“ Mit Logik erster Stufe können wir diese Einschränkung umgehen, indem wir das Konzept der linearen Zeit durch ein Konzept von Verzweigungssituationen ersetzen, das eine als **Situationskalkül** bezeichnete Darstellung verwendet und wie folgt arbeitet:

- Der Ausgangszustand wird als **Situation** bezeichnet. Wenn s eine Situation und a eine Aktion darstellen, ist $\text{RESULT}(s, a)$ ebenfalls eine Situation. Es gibt keine anderen Situationen. Somit entspricht eine Situation einer Sequenz (oder einem Verlauf) von Aktionen. Man kann sich eine Situation auch als Ergebnis der angewendeten Aktionen vorstellen. Allerdings ist zu beachten, dass zwei Situationen nur einander gleich sind, wenn ihr Beginn und ihre Aktionen gleich sind: $(\text{RESULT}(s, a) = \text{RESULT}(s', a')) \Leftrightarrow (s = s' \wedge a = a')$. ▶ Abbildung 10.12 zeigt einige Beispiele für Aktionen und Situationen.
- Eine Funktion oder Relation, die sich von einer Situation zur nächsten ändern kann, ist ein **Fluent**. Per Konvention ist die Situation s immer das letzte Argument des Fluenten. Zum Beispiel ist $Bei(x, l, s)$ ein relationaler Fluent, der wahr ist, wenn sich Objekt x am Ort l in Situation s befindet, und Ort ist ein funktionaler Fluent, sodass $Ort(x, s) = l$ in der gleichen Situation wie $Bei(x, l, s)$ gilt.

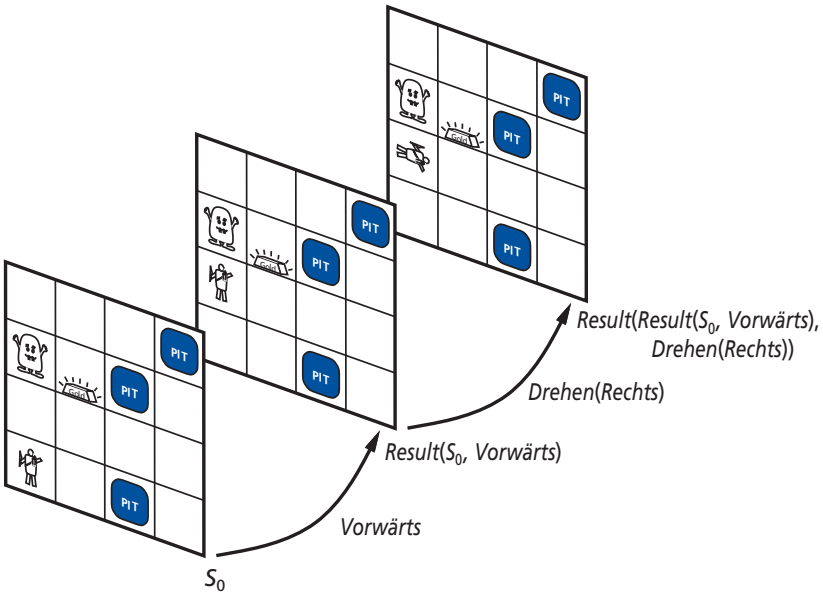


Abbildung 10.12: Situationen als Ergebnisse (RESULT) von Aktionen in der Wumpus-Welt.

- Die Vorbedingungen jeder Aktion werden mit einem **Möglichkeitsaxiom** (Possibility Axiom) beschrieben, das angibt, wann die Aktion unternommen werden kann. Es hat die Form $\Phi(s) \Rightarrow Poss(a, s)$, wobei $\Phi(s)$ eine Formel ist, die mithilfe von s die Vorbedingungen beschreibt. Das folgende Beispiel aus der Wumpus-Welt besagt, dass es möglich ist zu schießen, wenn der Agent lebt und einen Pfeil besitzt:

$$Lebt(Agent, s) \wedge Haben(Agent, Pfeil, s) \Rightarrow Poss(Schiessen, s).$$

- Jeder Fluent wird mit einem **Nachfolgerzustandsaxiom** beschrieben, das aussagt, was dem Fluenten passiert, abhängig von der unternommenen Aktion. Dies ähnelt dem Ansatz, den wir für die Aussagenlogik genommen haben. Das Axiom hat die Form

Aktion ist möglich $\Rightarrow$
(*Fluent ist im Ergebniszustand wahr* $\Leftrightarrow$ *Der Effekt der Aktion hat ihn wahr gemacht*
$\vee$ *Er war vorher wahr und die Aktion hat ihn nicht verändert.*

Zum Beispiel besagt das Axiom für den relationalen Fluenten *Halten*, dass der Agent etwas Gold g hält, nachdem er eine mögliche Aktion genau dann ausgeführt hat, wenn die Aktion ein *Greifen* von g war oder wenn der Agent bereits g hält und die Aktion es nicht losgelassen hat:

$$Poss(a, s) \Rightarrow (Halten(Agent, g, Result(a, s)) \Leftrightarrow a = Greifen(g) \vee (Halten(Agent, g, s) \wedge a \neq Loslassen(g))).$$

- Wir brauchen **eindeutige Aktionsaxiome**, sodass der Agent zum Beispiel $a \neq Loslassen(g)$ schließen kann. Für jedes verschiedene Paar von Aktionsnamen A_i und A_j haben wir ein Axiom, das besagt, dass die Aktionen unterschiedlich sind:

$$A_i(x, ...) \neq A_j(y, ...).$$

  Und für jeden Aktionsnamen A_i haben wir ein Axiom, das besagt, dass zwei Verwendungen dieses Aktionsnamens genau dann gleich sind, wenn alle ihre Argumente gleich sind:

$$A_i(x_1, ..., x_n) = A_i(y_1, ..., y_n) \Leftrightarrow x_1 = y_1 \wedge ... \wedge x_n = y_n.$$

- Eine Lösung ist eine Situation (und folglich eine Sequenz von Aktionen), die das Ziel erfüllt.

Die Arbeiten zum Situationskalkül haben viel dazu beigetragen, die formale Semantik der Planung zu definieren und neue Forschungsbereiche zu erschließen. Bislang gibt es aber keine praktisch ausgeführten Planungsprogramme im großen Maßstab, die auf logischer Deduktion über dem Situationskalkül basieren. Das hängt zum Teil mit der Schwierigkeit zusammen, effiziente Inferenz in Logik erster Stufe zu realisieren, beruht aber hauptsächlich darauf, dass das Gebiet noch keine effektiven Heuristiken für das Planen mit dem Situationskalkül entwickelt hat.

10.4.3 Planen als Problem unter Rand- und Nebenbedingungen

Wir haben gesehen, dass Planen als Problem unter Rand- und Nebenbedingungen (Constraint Satisfaction Problem, CSP) viel mit boolescher Erfüllbarkeit gemein hat, und wir haben auch gezeigt, dass CSP-Techniken für Terminplanungsprobleme effektiv sind. Deshalb dürfte es nicht überraschen, dass sich ein gebundenes Planungsproblem (d.h. das Problem, einen Plan der Länge k zu finden) als CSP kodieren lässt. Die Kodierung ähnelt derjenigen eines SAT-Problems (*Abschnitt 10.4.1*), weist aber eine wichtige Vereinfachung auf: Bei jedem Zeitschritt brauchen wir nur eine einzelne Variable, $Action^t$, deren Domäne die Menge möglicher Aktionen ist. Es ist nicht mehr eine Variable für jede Aktion erforderlich und wir brauchen auch keine Aktionsausschlussaxiome. Außerdem ist es möglich, einen Planungsgraphen zu einem CSP zu kodieren. Diesen Ansatz verwendet GP-CSP (Do und Kambhampati, 2003).

10.4.4 Planen als Verfeinerung von partiell geordneten Plänen

Alle bisher gezeigten Ansätze konstruieren *vollständig geordnete* Pläne, die aus streng linearen Aktionssequenzen bestehen. Diese Darstellung ignoriert die Tatsache, dass viele Teilprobleme unabhängig sind. Eine Lösung für ein Luftfrachtproblem besteht aus einer vollkommen geordneten Aktionsfolge, doch wenn 30 Pakete in das eine Flugzeug auf dem einen Flughafen und 50 Pakete in ein anderes Flugzeug auf einem anderen Flughafen geladen werden, scheint es sinnlos zu sein, mit einer streng linearen Reihenfolge von 80 Ladeaktionen aufzuwarten; die beiden Teilmengen der Aktionen sollten als unabhängig betrachtet werden.

Eine Alternative besteht darin, Pläne als *partiell geordnete* Strukturen darzustellen: Ein Plan ist eine Menge von Aktionen und eine Menge von Beschränkungen der Form $Vor(a_i, a_j)$, die besagt, dass die eine Aktion vor einer anderen auftritt. Der untere Teil

von ▶ Abbildung 10.13 zeigt einen partiell geordneten Plan, der eine Lösung für das Ersatzreifenproblem ist. Aktionen sind als Kästen und Reihenfolgeeinschränkungen als Pfeile dargestellt. Beachten Sie, dass die beiden Aktionen *Entfernen*(*Ersatzreifen*, *Kofferraum*) und *Entfernen*(*Platt*, *Achse*) in beliebiger Reihenfolge stattfinden können, solange sie beide vor der Aktion *Befestigen*(*Ersatzreifen*, *Achse*) abgeschlossen sind.

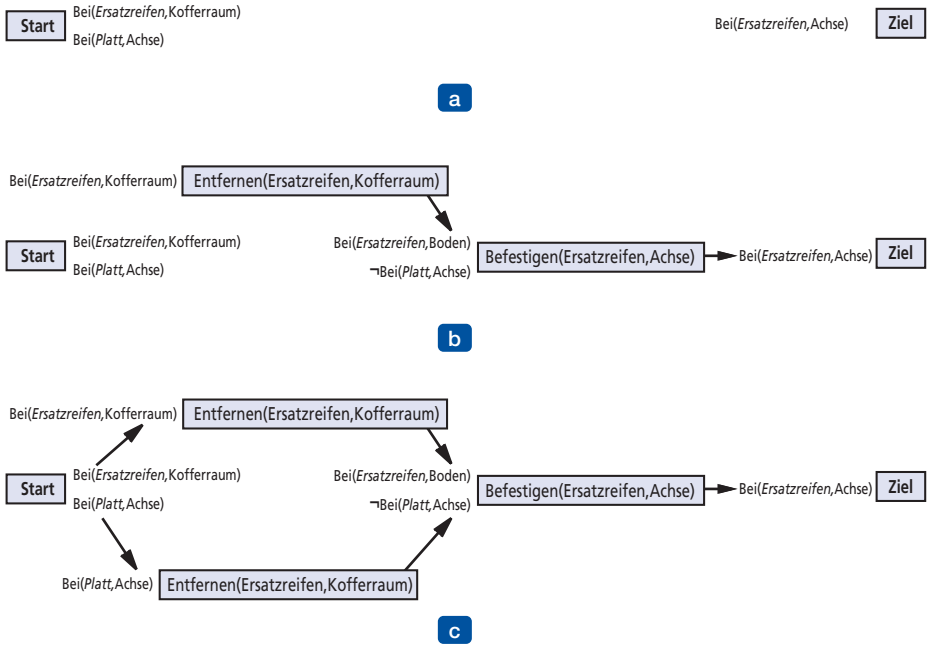


Abbildung 10.13: (a) Das Reifenproblem ausgedrückt als ein leerer Plan. (b) Ein unvollständiger partiell geordneter Plan für das Reifenproblem. Kästen stellen Aktionen dar und Pfeile zeigen an, dass die eine Aktion vor einer anderen auftreten muss. (c) Eine vollständige partiell geordnete Lösung.

Partiell geordnete Pläne werden über eine Suche durch Planraum statt durch den Zustandsraum erzeugt. Wir beginnen mit einem leeren Plan, der lediglich aus dem Ausgangszustand und dem Ziel ohne dazwischen liegende Aktionen besteht, wie es im oberen Teil von Abbildung 10.13 zu sehen ist. Die Suchprozedur sucht dann nach einem **Mangel** im Plan und ergänzt den Plan, um den Mangel zu korrigieren (oder geht in der Suche zurück, um etwas anderes zu probieren, wenn keine Korrektur möglich ist). Ein Mangel ist etwas, das einen partiellen Plan davon abhält, eine Lösung zu sein. Zum Beispiel ist es im leeren Plan ein Mangel, dass keine Aktion *Bei*(*Ersatzreifen*, *Achse*) erreicht. Eine Möglichkeit, den Mangel zu korrigieren, besteht darin, die Aktion *Befestigen*(*Ersatzreifen*, *Achse*) in den Plan einzufügen. Natürlich führt das einige neue Mängel ein: Die Vorbedingungen der neuen Aktion werden nicht erreicht. Die Suche nimmt weitere Ergänzungen am Plan vor (falls erforderlich mit Backtracking), bis alle Mängel aufgelöst sind, wie es der untere Teil von Abbildung 10.13 zeigt. Bei jedem Schritt machen wir die **geringst mögliche Zusage**, um den Mangel zu beseitigen. Zum Beispiel müssen wir beim Hinzufügen der Aktion *Entfernen*(*Ersatzreifen*, *Kofferraum*) zusagen, dass sie vor *Befestigen*(*Ersatzreifen*, *Achse*) auftritt, doch machen wir keine andere Zusage, die sie vor oder nach anderen Aktionen platziert. Wenn es im Aktionsschema eine Variable gäbe, die man ungebunden lassen könnte, würden wir das tun.

In den 1980er und 1990er Jahren sah man die Planung mit partieller Ordnung als beste Möglichkeit an, Planungsprobleme mit unabhängigen Teilproblemen zu behandeln – immerhin war es der einzige Ansatz, der explizit unabhängige Zweige eines Plans darstellt. Andererseits hat dieses Verfahren den Nachteil, keine explizite Darstellung von Zuständen im Zustandsübergangsmodell zu haben. Dadurch gestalten sich manche Berechnungen recht umständlich. Bis 2000 hatten Planer mit Vorwärtssuche ausgezeichnete Heuristiken entwickelt, die sie in die Lage versetzten, effizient die unabhängigen Teilprobleme zu entdecken, für die eigentlich die Planung mit partieller Ordnung konzipiert war. Im Ergebnis sind Planer mit partieller Ordnung bei vollständig automatisierten klassischen Planungsproblemen nicht konkurrenzfähig.

Allerdings bleibt die Planung mit partieller Ordnung ein wichtiges Teilgebiet. Für manche besonderen Aufgaben wie zum Beispiel die Arbeitsvorbereitung stellt die Planung mit partieller Ordnung mit domänenspezifischen Heuristiken die Technologie der Wahl. Viele dieser Systeme verwenden Bibliotheken von Plänen auf höherer Ebene, wie sie *Abschnitt 11.2* beschreibt. Planen mit partieller Ordnung wird ebenfalls häufig in Bereichen eingesetzt, wo es für den Menschen wichtig ist, die Pläne zu verstehen. Operative Pläne für Raumfahrzeuge und Mars-Rover werden durch Planer mit partieller Ordnung generiert und dann durch menschliche Experten überprüft, bevor sie zur Ausführung auf die Fahrzeuge hochgeladen werden. Das Konzept der Planverfeinerung erleichtert es dem Menschen, die Arbeitsweise der Planungsalgorithmen zu verstehen und zu verifizieren, ob sie korrekt sind.

10.5 Analyse von Planungsansätzen

Planung kombiniert die beiden wichtigsten Gebiete der KI, die wir bisher angesprochen haben: *Suche* und *Logik*. Einen Planer kann man entweder als Programm betrachten, das nach einer Lösung sucht, oder als eines, das (konstruktiv) die Existenz einer Lösung beweist. Die wechselseitige Inspiration aus beiden Bereichen hat innerhalb der letzten zehn Jahre zu Leistungsverbesserungen in mehreren Größenordnungen geführt, ebenso wie zu einem verstärkten Einsatz von Planern in industriellen Anwendungen. Leider wissen wir noch nicht genau, welche Techniken am besten für welche Probleme geeignet sind. Es ist durchaus möglich, dass sich neue Techniken entwickeln, die die existierenden Methoden verdrängen.

Planen dient hauptsächlich der Kontrolle einer kombinatorischen Explosion. Wenn es n Aussagen in einer Domäne gibt, dann gibt es 2^n Zustände. Wie wir gesehen haben, ist Planung PSPACE-hart. Gegen einen derartigen Pessimismus kann die Identifizierung von unabhängigen Suchproblemen eine wirksame Waffe sein. Im besten Fall – vollständige Zerlegbarkeit des Problems vorausgesetzt – erhalten wir eine exponentielle Beschleunigung. Die Zerlegbarkeit wird jedoch durch negative Interaktionen zwischen Aktionen verhindert. GRAPHPLAN zeichnet Mutexe auf, um herauszufinden, wo die schwierigen Interaktionen liegen. SATPLAN repräsentiert einen ähnlichen Bereich von Mutex-Beziehungen, verwendet hierzu aber die allgemeine CNF-Form statt einer spezifischen Datenstruktur. Die Vorwärtssuche geht das Problem heuristisch an und versucht, Muster (Teilmengen von Aussagen) zu finden, die unabhängige Teilprobleme abdecken. Da dieser Ansatz heuristisch ist, kann er sogar funktionieren, wenn die Teilprobleme nicht vollständig unabhängig sind.

Manchmal kann man ein Problem effizient lösen, wenn man erkennt, dass negative Interaktionen ausgeschlossen werden müssen. Wir sagen, ein Problem hat **serialisierbare Unterziele**, wenn es eine Reihenfolge von Unterzielen gibt, sodass der Planer sie in dieser Reihenfolge erreichen kann, ohne eines der zuvor erreichten Unterziele rückgängig machen zu müssen. Ist es beispielsweise in der Blockwelt ein Ziel, einen Turm zu bauen (z.B. Block *A* auf *B*, der seinerseits auf *C* liegt, der sich wiederum auf dem *Tisch* befindet, wie Abbildung 10.4 zeigt), dann sind die Unterziele von unten nach oben serialisierbar: Wenn wir zuerst „*C* auf dem *Tisch*“ realisieren, dann müssen wir dies nie rückgängig machen, während wir versuchen, die anderen Unterziele zu erreichen. Ein Planer, der den Von-unten-nach-oben-Trick verwendet, kann jedes Problem der Blockweltdomäne lösen, ohne ein Backtracking durchführen zu müssen (obwohl er dabei vielleicht nicht immer den kürzesten Plan findet).

Betrachten wir ein komplexeres Beispiel. Für den Planer Remote Agent, der das Raumschiff Deep Space One der NASA steuerte, wurde festgestellt, dass die Aussagen für die Steuerung eines Raumschiffes serialisierbar sind. Das ist vielleicht nicht allzu überraschend, weil die Entwickler eines Raumschiffes darauf achten, dass seine Steuerung so einfach wie möglich ist (abhängig von anderen Bedingungen). Unter Ausnutzung der serialisierten Reihenfolge der Ziele konnte der Remote Agent einen Großteil der Suche eliminieren. Das bedeutete, er war schnell genug, um das Raumschiff in Echtzeit zu steuern – etwas, was man zuvor für unmöglich gehalten hatte.

Planer wie GRAPHPLAN, SATPLAN und FF haben den Bereich der Planung vorangebracht, indem sie das Leistungsniveau von Planungssystemen angehoben, die darstellerischen und kombinatorischen Aspekte verdeutlicht und nützliche Heuristiken entwickelt haben. Allerdings stellt sich die Frage, wie weit sich diese Techniken skalieren lassen. Es ist wahrscheinlich, dass sich weiterer Fortschritt bei größeren Problemen nicht auf faktorisierte und aussagenlogische Darstellungen verlassen kann und eine bestimmte Synthese von Logik erster Stufe und hierarchischen Darstellungen mit den derzeit verwendeten effizienten Heuristiken erfordert.

Bibliografische und historische Hinweise

Die KI-Planung entstand aus Arbeiten in den Bereichen der Zustandsraumsuche, des Theorembeweisens und der Steuerungstheorie sowie aus praktischen Anforderungen der Robotik, Zeitplanung und anderen Domänen. STRIPS (Fikes und Nilsson, 1971), das erste große Planungssystem, zeigt die Interaktionen dieser Einflüsse auf. STRIPS wurde am SRI als Planungskomponente der Software für das Roboterprojekt Shakey entworfen. Seine allgemeine Steuerungsstruktur wurde nach der des GPS, des General Problem Solver (Newell und Simon, 1961), modelliert. GPS ist ein Zustandsraum-Suchsystem, das eine Means-End-Analyse verwendete. Bylander (1992) zeigt, dass die einfache STRIPS-Planung PSPACE-vollständig ist. Fikes und Nilsson (1993) geben einen historischen Rückblick auf das STRIPS-Projekt und seine Beziehung zu neueren Arbeiten auf dem Gebiet der Planung.

Die von STRIPS verwendete Repräsentationssprache war sehr viel einflussreicher als sein algorithmischer Ansatz; was wir die „klassische“ Sprache nennen, kommt der nahe, die STRIPS verwendet hat. ADL, die Action Description Language (Pednault, 1986) lockerte einige der Beschränkungen der Sprache STRIPS und ermöglichte es, realistischere Probleme zu kodieren. Nebel (2000) untersucht Schemas für die Kompilierung von ADL nach STRIPS. PDDL, die Problem Domain Description Language (Ghallab et al.,

1998), wurde als vom Computer analysierbare, standardisierte Syntax für die Repräsentation von Planungsproblemen eingeführt und wird seit 1998 als Standardsprache für den Internationalen Planungswettbewerb eingesetzt. Es gab mehrere Erweiterungen und die neueste Version PDDL 3.0 schließt Planbeschränkungen und -präferenzen ein (Gerevini und Long, 2005).

Planer Anfang der 1970er Jahre betrachteten im Allgemeinen vollständig geordnete Aktionsfolgen. Eine Problemzerlegung erreichte man, indem für jedes Unterziel ein Unterplan berechnet wurde und diese Unterpläne dann in irgendeiner Reihenfolge wieder verkettet wurden. Man erkannte schnell, dass dieser Ansatz, von Sacerdoti (1975) auch als **lineare Planung** bezeichnet, unvollständig war. Er kann einige sehr einfache Probleme nicht lösen, so beispielsweise die Sussman-Anomalie (siehe Übung 10.7), wie Allen Brown während seiner Experimente mit dem System HACKER feststellte (Sussman, 1975). Ein vollständiger Planer muss eine **Verzahnung** (**Interleaving**) der Aktionen aus verschiedenen Unterplänen innerhalb einer einzigen Folge erlauben. Das Konzept der serialisierbaren Unterziele (Korf, 1987) entspricht genau der Problemmenge, für die nichtverzahnbare Planer vollständig sind.

Eine Lösung für das Interleaving-Problem war die Zielregressionsplanung, eine Technik, bei der Schritte in einem vollständig geordneten Plan neu angeordnet werden, um Konflikte zwischen Unterzielen zu vermeiden. Dies wurde von Waldinger (1975) vorgestellt und auch in WARPLAN von Arren (1974) verwendet. WARPLAN ist auch deshalb bemerkenswert, weil es der erste Planer war, der in einer logischen Programmiersprache geschrieben war (Prolog). Er gehört zu den besten Beispielen, welch große Einsparungen durch die Verwendung der Logikprogrammierung erzielt werden können: WARPLAN besteht aus nur 100 Zeilen Code, einem kleinen Bruchteil der Größe vergleichbarer Planer der damaligen Zeit.

Dem partiell ordnenden Planen liegen unter anderem die Konflikterkennung (Tate, 1975a) und der Schutz erreichter Bedingungen gegenüber Störungen (Sussman, 1975) zugrunde. Der Aufbau partiell geordneter Pläne (auch als **Aufgabennetze** bezeichnet) erfolgte zuerst im NOAH-Planer (Sacerdoti, 1975, 1977) und im NONLIN-System von Tate (1975b, 1977).

Das partiell ordnende Planen beherrschte die nächsten 20 Forschungsjahre, doch die erste klare formale Darstellung war TWEAK (Chapman, 1987), ein Planer, der ausreichend einfach war, um Beweise der Vollständigkeit und Unhandhabbarkeit (NP-Härte und Unentscheidbarkeit) verschiedener Planungsprobleme zu erlauben. Die Arbeit von Chapman führte zu einer unkomplizierten Beschreibung eines vollständigen partiell ordnenden Planers (McAllester und Rosenblitt, 1991) und dann zu den weit verbreiteten Implementierungen SNLP (Soderland und Weld, 1991) und UCPOP (Penberthy und Weld, 1992). Partiell ordnendes Planen fiel gegen Ende der 1990er Jahre in Ungnade, als schnellere Methoden entstanden. Nguyen und Kambhampati (2001) sagen, eine Rehabilitation sei mehr als verdient: Mit exakten Heuristiken, die von einem Planungsgraphen abgeleitet wurden, skaliert ihr REPOP-Planer sehr viel besser als GRAPHPLAN in parallelisierbaren Domänen und kann mit den schnellsten Zustandsraumplanern konkurrieren.

Das wiederauflebende Interesse an der Zustandsraumplanung wurde angeführt von Drew McDermotts UNPOP-Programm (1996), das als Erstes eine Heuristik „Löschlisten ignorieren“ vorschlug. Der Name UNPOP war eine Reaktion auf die überwältigende Konzentration auf das partiell ordnende Planen zu dieser Zeit; McDermott vermutete, dass andere Ansätze nicht die Aufmerksamkeit erhielten, die sie verdienten. Der Heu-

ristic Search Planner (HSP) von Bonet und Geffner und seine späteren Ableitungen (Bonet und Geffner, 1999; Haslum et al., 2005; Haslum, 2006) waren die ersten Programme, die eine Zustandsraumsuche für große Planungsprobleme praktikabel machten. HSP sucht in der Vorwärtsrichtung, während HSPR (Bonet und Geffner, 1999) rückwärts sucht. Der heute erfolgreichste Zustandsraumsucher ist FF (Hoffmann 2001; Hoffmann und Nebel, 2001; Hoffmann, 2005), Gewinner des Planungswettbewerbes der AIPS 2000. FASTDOWNWARD (Helmert, 2006) ist ein Planer mit Zustandsraumsuche in Vorwärtsrichtung, der die Aktionsschemas zu einer alternativen Darstellung vorverarbeitet, was einige der Beschränkungen expliziter macht. FASTDOWNWARD (Helmert und Richter, 2004; Helmert, 2006) hat 2004 den Planungswettbewerb gewonnen und LAMA (Richter und Westphal, 2008), ein auf FASTDOWNWARD basierender Planer mit verbesserten Heuristiken, gewann den Wettbewerb von 2008.

Bylander (1994) und Ghallab et al. (2004) diskutieren die Berechnungskomplexität verschiedener Varianten des Planungsproblems. Helmert (2003) beweist Komplexitätsgrenzen für viele der Standard-Benchmark-Probleme und Hoffmann (2005) analysiert den Suchraum der Heuristik „Löschlisten ignorieren“. Heuristiken für das Mengenabdeckungsproblem werden von Caprara et al. (1995) im Rahmen von Zeitplanungsoperationen der italienischen Eisenbahn diskutiert. Edelkamp (2009) und Haslum et al. (2007) beschreiben, wie sich Musterdatenbanken für Planungsheuristiken konstruieren lassen. Wie bereits in *Kapitel 3* erwähnt, zeigen Felner et al. (2004) ermutigende Ergebnisse unter Verwendung von Musterdatenbanken für Schiebeblock-Puzzles, die sich als Planungsdomäne vorstellen lassen, doch Hoffmann et al. (2006) zeigen einige Grenzen der Abstraktion für klassische Planungsprobleme auf.

Avrim Blum und Merrick Furst (1995, 1997) gelang mit ihrem GRAPHPLAN-System, das um Größenordnungen schneller war als die partiell ordnenden Planer dieser Zeit, eine Wiederbelebung des Gebietes der Planung. Es folgten schnell andere Graphenplanungssysteme, wie etwa IPP (Koehler et al., 1997), STAN (Fox und Long, 1998) und SGP (Weld et al., 1998). Kurz zuvor war von Ghallab und Laruelle (1994) eine Datenstruktur entwickelt worden, die eng an den Planungsgraphen erinnerte. Ihr partiell ordnender Planer IXTET verwendete ihn, um exakte Heuristiken abzuleiten, um Suchen durchzuführen. Nguyen et al. (2001) bieten eine gründliche Analyse der von Planungsgraphen abgeleiteten Heuristiken. Unsere Diskussion der Planungsgraphen basiert teilweise auf dieser Arbeit sowie auf Vorlesungsskripten und Artikeln von Subbarao Kambhampati. Wie im Kapitel bereits erwähnt, kann ein Planungsgraph auf viele unterschiedliche Arten verwendet werden, um die Suche nach einer Lösung zu leiten. Der Gewinner des Planungswettbewerbes der AIPS von 2002, LPG (Gerevini und Serina, 2002, 2003), durchsuchte Planungsgraphen unter Verwendung einer lokalen Suchtechnik, die durch WALKSAT inspiriert war.

Das Konzept des Situationskalküls für die Planung wurde durch John McCarthy (1963) eingeführt. Die Version, die wir hier zeigen, wurde von Ray Reiter (1991, 2001) vorgeschlagen.

Kautz et al. (1996) untersuchten auch verschiedene Möglichkeiten, um Aktionsschemas in Aussagenlogik zu überführen, und stellten fest, dass die kompaktesten Formen nicht unbedingt zu den kürzesten Lösungszeiten führten. Ernst et al. (1997) führten eine systematische Analyse durch. Darüber hinaus entwickelten sie einen automatischen „Compiler“, der aus PDDL-Problemen aussagenlogische Repräsentationen generierte. Der Planer BLACKBOX, der Ideen aus GRAPHPLAN und SATPLAN kombiniert,

wurde von Kautz und Selman (1998) entwickelt. Der auf einem CSP basierende Planer CPLAN wurde von van Beek und Chen (1999) beschrieben.

Vor kurzem entstand neues Interesse an der Repräsentation von Plänen als **binäre Entscheidungsdiagramme**. Dabei handelt es sich um kompakte Datenstrukturen für boolesche Ausdrücke, die in breitem Maße in der Gemeinde der Hardware-Verifizierung untersucht wurden (Clarke und Grumberg, 1987; McMillan, 1993). Es gibt Techniken, um Eigenschaften binärer Entscheidungsdiagramme zu beweisen, unter anderem auch die Eigenschaft, eine Lösung für ein Planungsproblem zu sein. Cimatti et al. (1998) stellen einen Planer vor, der auf diesem Ansatz basiert. Es wurden aber auch andere Repräsentationen verwendet; beispielsweise zeigen Vossen et al. (2001) die Verwendung der Integerprogrammierung für die Planung.

Welcher Ansatz der beste ist, ist noch offen, aber es gibt bereits einige interessante Vergleiche der verschiedenen Ansätze zur Planung. Helmert (2001) analysiert verschiedene Klassen von Planungsproblemen und zeigt, dass auf Bedingungen basierende Ansätze, wie etwa GRAPHPLAN und SATPLAN, am besten für NP-harte Domänen geeignet sind, während auf Suchen basierende Ansätze besser für Domänen geeignet sind, in denen machbare Lösungen ohne Backtracking gefunden werden können. GRAPHPLAN und SATPLAN haben Probleme in Domänen mit vielen Objekten, weil das bedeutet, dass sie viele Aktionen erzeugen müssen. In einigen Fällen kann das Problem verzögert oder vermieden werden, indem man die in die Aussagenlogik umgewandelten Aktionen nur nach Bedarf dynamisch erzeugt, anstatt sie alle vor dem Suchbeginn zu instantiieren.

Eine umfassende Anthologie der frühen Arbeiten auf dem Gebiet ist *Readings in Planning* (Allen et al., 1990. Weld (1994, 1999) bietet zwei ausgezeichnete Überblicke über Planungsalgorithmen der 1990er Jahre. Es ist interessant, die Änderungen in den fünf Jahren zwischen den beiden Überblicken zu beobachten: Der erste konzentriert sich auf partiell ordnendes Planen und der zweite stellt GRAPHPLAN und SATPLAN vor. Ein ausgezeichnetes Lehrbuch zu allen Aspekten der Planung ist *Automated Planning* (Ghallab et al., 2004). Der Artikel *Planning Algorithms* (2006) von LaValle behandelt sowohl klassische als auch stochastische Planung und geht ausführlich auf die Bewegungsplanung von Robotern ein.

Planungsforschung ist von Anfang an ein zentrales Thema der KI gewesen und Artikel zur Planung sind ein wichtiger Bestandteil von etablierten KI-Zeitschriften und -Konferenzen. Außerdem gibt es spezialisierte Konferenzen wie zum Beispiel die *International Conference on AI Planning Systems*, den *International Workshop on Planning and Scheduling for Space* und die *European Conference on Planning*.

Zusammenfassung

In diesem Kapitel haben wir das Problem des Planens in deterministischen, vollständig beobachtbaren, statischen Umgebungen definiert. Wir haben die PDDL-Darstellung für Planungsprobleme und mehrere algorithmische Ansätze für ihre Lösung vorgestellt. Die wichtigsten Aspekte sind:

- Planungssysteme sind problemlösende Algorithmen, die mit expliziten aussagenlogischen oder relationalen Repräsentationen von Zuständen und Aktionen arbeiten. Diese Repräsentationen ermöglichen die Ableitung effektiver Heuristiken sowie die Entwicklung leistungsfähiger und flexibler Algorithmen für die Lösung von Problemen.
- Die Sprache PDDL (Planning Domain Definition Language) beschreibt die Ausgangs- und Zielzustände als Konjunktionen von Literalen und die Aktionen in Form ihrer Vorbedingungen und Effekte.
- Die Zustandsraumsuche kann vorwärts (**Progression**) oder rückwärts (**Regression**) suchen. Wenn man Unabhängigkeit der Unterziele annimmt und verschiedene Lockerungen in das Planungsproblem einführt, lassen sich effektive Heuristiken ableiten.
- Ein **Planungsgraph** kann beginnend beim Ausgangszustand inkrementell aufgebaut werden. Jede Ebene enthält eine Obermenge aller Literale oder Aktionen, die zu diesem Zeitschritt auftreten könnten, und kodiert einen wechselseitigen Ausschluss (Mutex) zwischen Literalen oder Aktionen, die nicht auftreten können. Planungsgraphen ergeben praktische Heuristiken für Zustandsraum- und partiell geordnete Planer und können im GRAPHPLAN-Algorithmus direkt angewendet werden.
- Zu anderen Ansätzen gehören die Deduktion erster Stufe über Axiomen des Situationskalküls, die Kodierung eines Planungsproblems als boolesches Erfüllbarkeitsproblem oder als Problem unter Rand- und Nebenbedingungen (CSP) sowie das explizite Suchen durch den Raum partiell geordneter Pläne.
- Jeder der wichtigen Ansätze für das Planen hat seine Anhänger und es gibt keine Übereinstimmung darüber, welcher der beste ist. Die Konkurrenz und die gegenseitige Inspiration der Ansätze haben zu erheblichen Effizienzgewinnen für Planungssysteme geführt.

Übungen zu Kapitel 10

1 Betrachten Sie einen Roboter, dessen Arbeitsweise durch die folgenden PDDL-Operatoren beschrieben wird:

$Op(\text{ACTION}: Gehe(x, y), \text{PRECOND}: Bei(Robot, x), \text{EFFECT}: \neg Bei(Robot, x) \wedge Bei(Robot, y))$
$Op(\text{ACTION}: Aufnehmen(o), \text{PRECOND}: Bei(Robot, x) \wedge Bei(o, x), \text{EFFECT}: \neg Bei(o, x) \wedge Halten(o))$
$Op(\text{ACTION}: Ablegen(o), \text{PRECOND}: Bei(Robot, x) \wedge Halten(o), \text{EFFECT}: Bei(o, x) \wedge \neg Halten(o))$

a. Die Operatoren erlauben es dem Roboter, mehrere Objekte auf einmal zu halten. Zeigen Sie, wie sie sich durch ein $LeereHand$-Prädikat für einen Roboter, der nur ein Objekt halten kann, modifizieren lassen.

b. Nehmen Sie an, dass dies die einzigen Aktionen in der Welt sind, und schreiben Sie ein Nachfolgerzustandsaxiom für $LeereHand$.

2 Beschreiben Sie Unterschiede und Ähnlichkeiten zwischen Problemlösung und Planung.

3 Es seien die Aktionsschemas und der Ausgangszustand gemäß Abbildung 10.1 gegeben. Welche anwendbaren konkreten Instanzen von $Fliegen(p, von, nach)$ gibt es für den Zustand, der durch

$$Bei(P_1, JFK) \wedge Bei(P_2, SFO) \wedge Flugzeug(P_1) \wedge Flugzeug(P_2) \wedge Flughafen(JFK) \wedge Flughafen(SFO)$$

beschrieben wird?

4 Das Affen-und-Bananen-Problem beschreibt einen Affen in einem Labor, wobei von der Decke außer Reichweite Bananen hängen. Es steht ein Kasten zur Verfügung, der es dem Affen erlaubt, die Bananen zu erreichen, wenn er auf diesen Kasten klettert. Anfänglich befindet sich der Affe an der Position A, die Bananen an der Position B und der Kasten an der Position C. Der Affe und der Kasten haben die Höhe $Gering$, aber wenn der Affe auf den Kasten klettert, hat er die Höhe $Hoch$ – dieselbe wie die Bananen. Dem Affen stehen verschiedene Aktionen zur Verfügung: $Gehen$, um von einer Position an eine andere zu gehen, $Schieben$, um ein Objekt von einer Position an eine andere zu verschieben, $KletternAuf$, um auf ein Objekt zu klettern, $KletternVon$, um von einem Objekt zu klettern, $Greifen$ oder $Loslassen$, um ein Objekt zu greifen oder es loszulassen. Der Affe kann ein Objekt greifen und festhalten, wenn er und das Objekt sich an derselben Position und in derselben Höhe befinden.

a. Schreiben Sie die Beschreibung des Ausgangszustandes.

b. Schreiben Sie sechs Aktionsschemas.

c. Angenommen, der Affe will die Wissenschaftler foppen, die gerade Kaffeepause haben, indem er sich die Bananen greift, aber den Kasten an seiner ursprünglichen Position belässt. Schreiben Sie dies als allgemeines Ziel (d.h. nicht unter der Annahme, dass sich der Kasten notwendigerweise an der Position C befindet) in der Sprache des Situationskalküls auf. Kann dieses Ziel durch ein klassisches Planungssystem gelöst werden?

d. Ihr Axiom für das *Schieben* ist möglicherweise fehlerhaft, denn wenn das Objekt zu schwer ist, bleibt es an seiner ursprünglichen Position, wenn der *Schieben*-Operator angewendet wird. Korrigieren Sie Ihre Problembeschreibung, sodass auch schwere Objekte berücksichtigt werden.

5 Der ursprüngliche STRIPS-Planer war darauf ausgelegt, den Roboter Shakey zu steuern. ▶ Abbildung 10.14 zeigt eine Version der Shakey-Welt, die aus vier Räumen besteht, die entlang eines Korridors angelegt sind und wo jeder Raum eine Tür und einen Lichtschalter hat. Die Aktionen in der Shakey-Welt bestehen darin, von Position zu Position zu gehen, bewegliche Objekte (z.B. Kästen) zu verschieben, auf feste Objekte (z.B. Kästen) zu klettern bzw. von ihnen herunterzusteigen und Lichtschalter an- und auszuschalten. Der Roboter selbst konnte weder auf einen Kasten steigen noch einen Schalter betätigen, aber der Planer war in der Lage, Pläne zu ermitteln und auszudrucken, die über die Fähigkeiten des Roboters hinausgingen. Die sechs Aktionen von Shakey sehen wie folgt aus:

- $Gehen(x, y, r)$, wobei es erforderlich ist, dass sich Shakey an der Position x befindet und dass x und y Positionen im selben Raum r sind. Konventionsgemäß gehört eine Tür zwischen zwei Räumen zu beiden Räumen.
- Einen Kasten b von der Position x an die Position y im selben Raum schieben: $Schieben(b, x, y, r)$. Wir brauchen das Prädikat $Kasten$ sowie Kostanten für die Kästen.
- Auf einen Kasten von Position x aus klettern: $Hochklettern(x, b)$; von einem Kasten zu Position x herunterklettern: $Herunterklettern(b, x)$. Wir brauchen das Prädikat Auf und die Konstante $Boden$.
- Einen Lichtschalter anschalten: $Anschalten(s, b)$; einen Lichtschalter ausschalten: $Ausschalten(s, b)$. Um ein Licht an- oder auszuschalten, muss Shakey auf einem Kasten an der Position des Lichtschalters stehen.

Schreiben Sie PDDL-Sätze für die sechs Aktionen von Shakey und den Ausgangszustand aus Abbildung 10.14. Konstruieren Sie einen Plan, wie Shakey $Kasten_2$ in $Raum_2$ bringen kann.

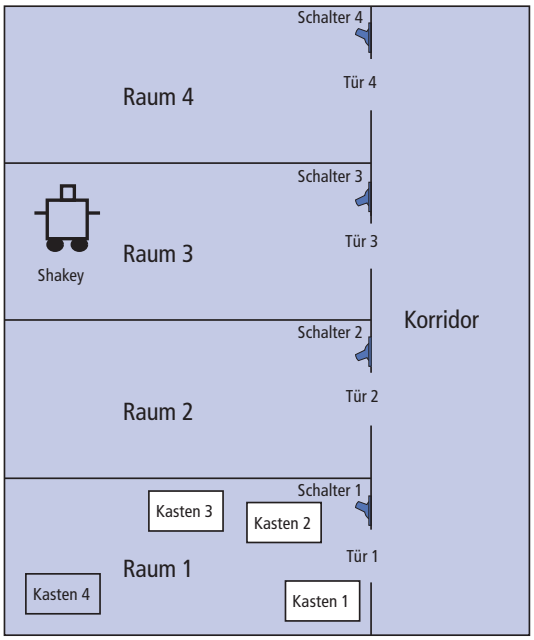


Abbildung 10.14: Shakeys Welt. Shakey kann sich zwischen den Begrenzungen eines Raumes bewegen, durch die Tür zwischen Räumen gehen, auf erklimmbare Objekte steigen, bewegliche Objekte verschieben sowie Lichtschalter an- oder ausschalten.

6 Erklären Sie, warum das Verwerfen negativer Effekte aus jedem Aktionsschema in einem Planungsproblem zu einem gelockerten Problem führt.

7 Abbildung 10.4 zeigt ein Problem aus der Blockwelt, das unter dem Namen **Sussman-Anomalie** bekannt ist. Das Problem wurde als anomal betrachtet, weil die nicht verzahnten Planer der frühen 1970er Jahre es nicht lösen konnten. Schreiben Sie eine Definition des Problems und lösen Sie es – manuell oder mithilfe eines Planungsprogramms. Ein nicht verzahnter (non-interleaved) Planer ist ein Planer, der für zwei Unterziele, G_1 und G_2, entweder einen Plan für G_1, der mit einem Plan für G_2 verkettet ist, erzeugt oder umgekehrt. Erklären Sie, warum ein nicht verzahnter Planer dieses Problem nicht lösen kann.

8 Beweisen Sie, dass Rückwärtssuche mit PDDL-Problemen vollständig ist.

9 Konstruieren Sie die Ebenen 0, 1 und 2 des Planungsgraphen für das Problem in Abbildung 10.1.

10 Beweisen Sie die folgenden Zusicherungen über Planungsgraphen:

a. Ein Literal, das nicht auf der letzten Ebene des Graphen erscheint, kann nicht erreicht werden.

b. Die Ebenenkosten eines Literals in einem seriellen Graphen sind nicht größer als die tatsächlichen Kosten eines optimalen Planes für seine Erreichung.

11 Wir haben gesehen, dass Planungsgraphen nur aussagenlogische Aktionen verarbeiten können. Was passiert, wenn wir Planungsgraphen für ein Problem mit Variablen im Ziel einsetzen wollen, wie beispielsweise $Bei(P_1, x) \wedge Bei(P_2, x)$, wobei für x angenommen wird, dass es durch einen Existenzquantor gebunden ist, der sich über eine endliche Domäne von Positionen erstreckt? Wie könnten Sie ein derartiges Problem kodieren, damit es mithilfe von Planungsgraphen bearbeitet werden kann?

12 Die Heuristik der Mengenebenen (*Abschnitt 10.3.1*) verwendet einen Planungsgraphen, um die Kosten für das Erreichen eines konjunktiven Zieles aus dem aktuellen Zustand heraus abzuschätzen. Für welches gelockerte Problem stellt die Heuristik der Mengenebenen die Lösung dar?

13 Wir haben Vorwärts- und Rückwärts-Zustandsraumsuchen mit partiell ordnenden Planern verglichen und dabei festgestellt, dass es sich bei dem letztgenannten um einen Planraumsucher handelt. Erläutern Sie, wie Vorwärts- und Rückwärts-Zustandsraumsuche ebenfalls als Planraumsucher betrachtet werden können, und geben Sie an, welche Operatoren für die Planverfeinerung verwendet werden.

14 Bisher sind wir davon ausgegangen, dass die von uns erzeugten Pläne immer sicherstellen, dass Vorbedingungen einer Aktion erfüllt sind. Jetzt wollen wir untersuchen, was aussagenlogische Nachfolgerzustandsaxiome wie zum Beispiel $HatPfeil^{t+1} \Leftrightarrow (HatPfeil^t \wedge \neg Schiessen^t)$ über Aktionen aussagen, deren Vorbedingungen nicht erfüllt sind.

a. Zeigen Sie, dass die Axiome vorhersagen, dass nichts passiert, wenn eine Aktion in einem Zustand ausgeführt wird, in dem ihre Vorbedingungen nicht erfüllt sind.

b. Betrachten Sie einen Plan p, der die erforderlichen Aktionen enthält, um ein Ziel zu erreichen, aber auch nicht erlaubte Aktionen einschließt. Gilt

$$Ausgangszustand \wedge Nachfolgerzustandsaxiome \wedge p \models Ziel?$$

c. Kann mit Nachfolgerzustandsaxiomen erster Stufe im Situationskalkül bewiesen werden, dass ein Plan, der nicht erlaubte Aktionen enthält, das Ziel erreicht?

15 Betrachten Sie, wie eine Menge von Aktionsschemas in die Nachfolgerzustandsaxiome des Situationskalküls übersetzt werden können.

a. Betrachten Sie das Schema für $Fliegen(p, von, nach)$. Schreiben Sie eine logische Definition für das Prädikat $Poss(Fliegen(p, von, nach)\ s)$, das wahr ist, wenn die Vorbedingungen für $Fliegen(p, von, nach)$ in einer Situation s erfüllt sind.

b. Angenommen, $Fliegen(p, von, nach)$ ist das einzige Aktionsschema, das dem Agenten zur Verfügung steht. Schreiben Sie ein Nachfolgerzustandsaxiom für $Bei(p, x, s)$, das die gleichen Informationen wie das Aktionsschema erfasst.

c. Nehmen Sie nun an, dass es eine weitere Reisemethode gibt: $Teleportieren(p, von, nach)$. Sie hat die zusätzliche Vorbedingung $\neg Warped(p)$ und den zusätzlichen Effekt $Warped(p)$. Erklären Sie, wie die Situationskalkül-Wissensbasis angepasst werden muss.

d. Entwickeln Sie schließlich eine allgemeine und genau spezifizierte Prozedur, die eine Menge von Aktionsschemas in eine Menge von Nachfolgerzustandsaxiomen übersetzt.

16 Im SATPLAN-Algorithmus in Abbildung 7.22 sichert jeder Aufruf des Erfüllbarkeitsalgorithmus ein Ziel g^T zu, wobei T von 0 bis T_{max} geht. Nehmen Sie stattdessen an, dass der Erfüllbarkeitsalgorithmus nur einmal mit dem Ziel $g^0 \vee g^1 \vee ... \vee g^{T_{max}}$ aufgerufen wird.

a. Wird damit immer ein Plan zurückgegeben, falls ein solcher existiert, der eine Länge kleiner gleich T_{max} hat?

b. Führt dieser Ansatz neue falsche „Lösungen" ein?

c. Erörtern Sie, wie man einen Erfüllbarkeitsalgorithmus wie z.B. WALKSAT abändern könnte, sodass er Lösungen findet (falls solche existieren), wenn ihm ein disjunktives Ziel dieser Form übergeben wird.

Planen und Agieren in der realen Welt

11

ÜBERBLICK

In diesem Kapitel zeigen wir, wie ausdrucksstärkere Repräsentationen und interaktivere Agentenarchitekturen zu Planern führen, die in der realen Welt praktisch eingesetzt werden können.

Das vorherige Kapitel hat die grundlegenden Konzepte, Repräsentationen und Algorithmen für das Planen eingeführt. Planer, die in der realen Welt für das Planen und das Zeitmanagement von Raumfahrzeugen, Fabriken und Militäraktionen eingesetzt werden, sind komplexer; sie erweitern sowohl die Repräsentationssprache als auch die Art und Weise, wie der Planer mit der Umgebung interagiert. Dieses Kapitel beschreibt diese Erweiterungen. *Abschnitt 11.1* erweitert die klassische Sprache für das Planen, um über Aktionen mit Zeit- und Ressourceneinschränkungen zu sprechen. *Abschnitt 11.2* beschreibt Methoden für das Konstruieren von Plänen, die hierarchisch organisiert sind. Dadurch können menschliche Experten dem Planer mitteilen, was sie über das Lösen des Problems wissen. Hierarchie führt praktisch von selbst zu einer effizienten Plankonstruktion, weil der Planer ein Problem auf einer abstrakten Ebene lösen kann, bevor er ins Detail geht. In *Abschnitt 11.3* geht es um Agentenarchitekturen, die mit unsicheren Umgebungen zurechtkommen und Überlegung mit Ausführung verschachteln. Außerdem werden Beispiele für Systeme der realen Welt angegeben. *Abschnitt 11.4* zeigt, wie geplant wird, wenn die Umgebung weitere Agenten enthält.

11.1 Zeit, Zeitpläne und Ressourcen

Die klassische Planungsdarstellung gibt an, was zu tun ist und in welcher Reihenfolge, doch die Darstellung kann nichts über die Zeit aussagen: wie lange eine Aktion dauert und wann sie auftritt. Zum Beispiel könnten die Planer von *Kapitel 10* einen Zeitplan für eine Fluggesellschaft erzeugen, der besagt, welche Pläne welchen Flügen zugewiesen sind, doch müssen wir in der Praxis wissen, wie die Abflugs- und Ankunftszeiten aussehen. Dies ist Thema der **Zeitplanung**. In der realen Welt gibt es auch viele **Ressourcenbeschränkungen**; zum Beispiel hat eine Fluggesellschaft eine begrenzte Anzahl von Mitarbeitern – und Mitarbeiter, die sich auf einem Flug befinden, können nicht zur selben Zeit auf einem anderen sein. Dieser Abschnitt beschäftigt sich mit Methoden zur Darstellung und Lösung von Planungsproblemen, die Zeit- und Ressourcenbeschränkungen umfassen.

Dieser Abschnitt verfolgt den Ansatz „zuerst planen, später zeitplanen“: Das heißt, wir gliedern das Gesamtproblem in eine Planungsphase, in der die Aktionen ausgewählt werden, wobei gewisse Reihenfolgeeinschränkungen zu beachten sind, um den Zielen des Problems zu entsprechen, und in eine spätere Zeitplanungsphase, in der dem Plan zeitliche Informationen hinzugefügt werden, um sicherzustellen, dass er den Ressourcen- und Termineinschränkungen genügt.

Dieser Ansatz ist bei realen Produktions- und Logistikeinrichtungen gebräuchlich, wo die Planungsphase oftmals durch menschliche Experten durchgeführt wird. Die automatisierten Methoden von *Kapitel 10* lassen sich ebenfalls für die Planungsphase einsetzen, sofern sie Pläne erzeugen, die mit den unbedingt notwendigen Reihenfolgebeschränkungen, die für Korrektheit erforderlich sind, auskommen. Hierfür sind GRAPHPLAN (*Abschnitt 10.3*), SATPLAN (*Abschnitt 10.4.1*) und partiell geordnete Planer (*Abschnitt 10.4.4*) geeignet; auf Suche basierende Methoden (*Abschnitt 10.2*) erzeugen vollkommen geordnete Pläne, doch lassen sich diese leicht in Pläne mit minimalen Reihenfolgebeschränkungen konvertieren.

11.1.1 Zeit- und Ressourceneinschränkungen darstellen

Ein typisches **Job-Shop-Scheduling-Problem** (svw. Problem der Fertigungsablaufplanung), wie es in *Abschnitt 6.1.2* eingeführt wurde, umfasst eine Menge von **Jobs**, die ihrerseits eine Sammlung von **Aktionen** verkörpern, zwischen denen Reihenfolgeeinschränkungen bestehen. Jede Aktion hat eine **Dauer** und eine Menge von Ressourceneinschränkungen, die für die Aktion erforderlich sind. Jede Einschränkung spezifiziert einen Ressourcentyp (z.B. Bolzen, Schraubenschlüssel oder Piloten), die erforderliche Anzahl dieser Ressource und ob diese Ressource **konsumierbar** (z.B. stehen Bolzen nach dem Einbau nicht mehr zur Verfügung) oder **wieder verwendbar** (z.B. wird ein Pilot während eines Fluges belegt, steht aber wieder zur Verfügung, wenn der Flug vorüber ist) ist. Ressourcen können auch durch Aktionen mit negativer Konsumption erzeugt werden, einschließlich Fertigungs-, Wachstums- und Nachschubaktionen. Eine Lösung für ein Job-Shop-Scheduling-Problem muss die Anfangszeiten für jede Aktion spezifizieren sowie sämtliche zeitlichen Reihenfolge- und Ressourcenbeschränkungen erfüllen. Wie bei Such- und Planungsproblemen lassen sich Lösungen entsprechend einer Kostenfunktion bewerten; das kann recht kompliziert sein und nichtlineare Ressourcenkosten, zeitabhängige Kosten für Verzögerungen usw. betreffen. Der Einfachheit halber nehmen wir an, dass die Kostenfunktion lediglich die Gesamtdauer des Planes – die sogenannte **Produktionsdauer** – angibt.

▶ Abbildung 11.1 zeigt ein einfaches Beispiel: ein Problem, das die Montage zweier Autos betrifft. Das Problem besteht aus zwei Jobs, die jeweils die Form [*MotorEinbauen*, *RäderMontieren*, *Überprüfen*] haben. Dann deklariert die Anweisung *Resources*, dass es vier Arten von Ressourcen gibt, und nennt die Anzahl jedes beim Start verfügbaren Typs: 1 Motorwinde, 1 Radstation, 2 Inspektoren und 500 Radmuttern. Die Aktionsschemas geben die Dauer und den Ressourcenbedarf für jede Aktion an. Die Radmuttern werden verbraucht, wenn die Räder an das Auto montiert werden, während andere Ressourcen zu Beginn einer Aktion „geborgt“ und am Ende einer Aktion wieder freigegeben werden.

```
Jobs({MotorEinbauen1 ≺ RäderMontieren1 ≺ Überprüfen1 },
     {MotorEinbauen2 ≺ RäderMontieren2 ≺ Überprüfen2 })
Resources(MotorWinden(1), Radstationen(1), Inspektoren(2), Radmuttern(500))
Action(MotorEinbauen1, DAUER:30,
    VERWENDEN: MotorWinden(1))
Action(MotorEinbauen2, DAUER:60,
    VERWENDEN: MotorWinden(1))
Action(RäderMontieren1, DAUER:30,
    VERBRAUCHEN:Radmuttern(20), VERWENDEN: Radstationen(1))
Action(RäderMontieren2, DAUER:15,
    VERBRAUCHEN :Radmuttern(20), VERWENDEN: Radstationen(1))
Action(Überprüfen$_i$, DAUER:10,
    VERWENDEN :Inspektoren(1))
```

Abbildung 11.1: Ein Job-Shop-Scheduling-Problem für die Montage von zwei Autos, mit Ressourcenbeschränkungen. Die Notation $A \prec B$ bedeutet, dass Aktion A vor Aktion B stattfinden muss.

Die Repräsentation von Ressourcen als numerische Größen, wie beispielsweise *Inspektoren*(2), statt als benannte Entitäten, wie etwa *Inspektor*(I_1) und *Inspektor*(I_2), ist ein Beispiel für eine sehr allgemeine Technik, die sogenannte **Aggregation**. Der Aggregation liegt das Prinzip zugrunde, einzelne Objekte zu Quantitäten zu gruppieren, wenn die Objekte im Hinblick auf die vorliegende Aufgabe nicht zu unterscheiden sind. In unserem Montageproblem spielt es keine Rolle, *welcher* Inspektor das Auto überprüft,

sodass es nicht erforderlich ist, die Unterscheidung zu treffen. (Das gleiche Prinzip funktioniert im Missionare&Kannibalen-Problem in *Übung 3.9*.) Die Aggregation ist wichtig, um die Komplexität zu reduzieren. Betrachten Sie, was passiert, wenn ein Zeitplan vorgeschlagen wird, der zehn nebenläufige *Überprüfen*-Aktionen enthält, aber nur neun Inspektoren zur Verfügung stehen. Wenn die Inspektoren als Quantitäten dargestellt werden, wird sofort ein Fehler erkannt und der Algorithmus geht zurück, um einen anderen Zeitplan auszuprobieren. Wenn die Inspektoren als Individuen dargestellt sind, geht der Algorithmus zurück, um alle 10! Möglichkeiten auszuprobieren, den *Überprüfen*-Aktionen Inspektoren zuzuordnen.

11.1.2 Scheduling-Probleme lösen

Wir betrachten zunächst lediglich das zeitliche Scheduling-Problem und ignorieren dabei Ressourcenbeschränkungen. Um die Produktionsdauer (Plandauer) zu minimieren, müssen wir für alle Aktionen die frühesten Startzeiten, die mit den für das Problem angegebenen Reihenfolgeeinschränkungen konsistent sind, finden. Es ist hilfreich, diese Reihenfolgebeschränkungen wie in ▶ Abbildung 11.2 gezeigt als gerichteten Graphen zu betrachten, der die Aktionen in Beziehung setzt. Wir können dann die **Methode des kritischen Pfades** (Critical Path Method, CPM) auf diesen Graphen anwenden, um die möglichen Start- und Endzeiten jeder Aktion zu ermitteln. Ein Pfad durch einen Graphen, der einen partiell geordneten Plan darstellt, ist eine linear geordnete Sequenz von Aktionen, die mit *Start* beginnen und mit *Ende* enden. (Zum Beispiel gibt es im partiell geordneten Plan gemäß Abbildung 11.2 zwei Pfade.)

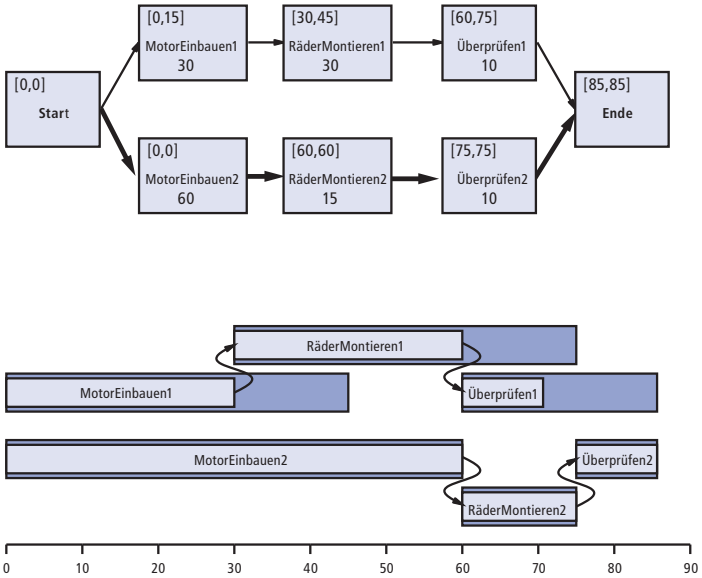


Abbildung 11.2: Oben: eine Darstellung der zeitlichen Beschränkungen für das Job-Shop-Scheduling-Problem gemäß Abbildung 11.1. Die Dauer jeder Aktion ist unten in den jeweiligen Rechtecken angegeben. Beim Lösen des Problems berechnen wir die früheste und späteste Startzeit als [*ES*, *LS*]-Paar, das jeweils oben links angezeigt wird. Die Differenz dieser beiden Zahlen ist der *Spielraum* einer Aktion; Aktionen mit null Spielraum befinden sich auf dem kritischen Pfad, der mit fett ausgezeichneten Pfeilen dargestellt ist. Unten: Hier ist dieselbe Lösung als Zeitleiste zu sehen. Graue Rechtecke stellen die Zeitintervalle dar, während derer eine Aktion ausgeführt werden kann, vorausgesetzt, die Ordnungsbedingungen werden berücksichtigt. Der nicht belegte Teil eines grauen Rechtecks zeigt den Spielraum.

Der **kritische Pfad** ist der Pfad, der die längste Gesamtdauer aufweist; der Pfad ist „kritisch“, weil er die Dauer des gesamten Planes bestimmt – wenn man andere Pfade kürzt, wird damit nicht der gesamte Plan gekürzt, aber eine Verzögerung des Starts einer Aktion auf dem kritischen Pfad verlangsamt den gesamten Plan. Aktionen, die sich nicht auf dem kritischen Pfad befinden, haben einen gewissen Spielraum – ein Zeitfenster, in dem sie ausgeführt werden können. Das Fenster wird abhängig von der frühestmöglichen Startzeit, *ES*, und der letztmöglichen Startzeit, *LS*, spezifiziert. Die Größe $LS - ES$ wird als **Spielraum** einer Aktion bezeichnet. In Abbildung 11.2 sehen wir, dass der gesamte Plan 85 Minuten dauert, dass jede Aktion im obersten Job einen Spielraum von 15 Minuten und jede Aktion auf dem kritischen Pfad (per Definition) 0 Spielraum hat. Zusammen bilden die *ES*- und *LS*-Zeiten für alle Aktionen einen **Zeitplan** (**Schedule**) für das Problem.

Die folgenden Formeln dienen als Definition für *ES* und *LS* sowie als Skizze eines dynamischen Programmieralgorithmus für ihre Berechnung. Die Aktionen sind A und B, wobei A < B bedeutet, dass A vor B kommt:

$$ES(Start) = 0$$
$$ES(B) = max_{A \prec B} ES(A) + Dauer(A)$$
$$LS(Ende) = ES(Ende)$$
$$LS(A) = min_{B \succ A} LS(B) - Dauer(A).$$

Die Idee dabei ist, dass wir zuerst $ES(Start)$ den Wert 0 zuweisen. Sobald wir eine Aktion B haben, sodass allen Aktionen, die unmittelbar vor B kommen, *ES*-Werte zugewiesen sind, setzen wir $ES(B)$ auf das Maximum der frühesten Endzeiten der unmittelbar vorausgehenden Aktionen, wobei die früheste Endzeit einer Aktion definiert ist als die früheste Startzeit plus die Dauer. Dieser Prozess wiederholt sich, bis jeder Aktion ein *ES*-Wert zugewiesen wurde. Die *LS*-Werte werden auf ähnliche Weise berechnet, wobei man von der *Ende*-Aktion aus rückwärts vorgeht.

Die Komplexität des kritischen Pfadalgorithmus ist einfach $O(Nb)$, wobei N die Anzahl der Aktionen und b der maximale Verzweigungsfaktor in eine oder aus einer Aktion ist. (Um das zu erkennen, beachten Sie, dass die Berechnungen von *LS* und *ES* einmal für jede Aktion erfolgen und jede Berechnung über höchstens b andere Aktionen iteriert.) Aus diesem Grund ist das Problem, einen Zeitplan mit minimaler Dauer zu finden, relativ einfach zu lösen, wenn eine partielle Ordnung der Aktionen gegeben ist und keine Ressourceneinschränkungen bestehen.

Im mathematischen Sinne sind Probleme des kritischen Pfades leicht zu lösen, da sie als Konjunktion von linearen Ungleichungen auf den Start- und Endzeiten definiert sind. Wenn wir Ressourcenbeschränkungen einführen, werden die resultierenden Beschränkungen auf Start- und Endzeiten komplizierter. Zum Beispiel erfordern die *MotorEinbauen*-Aktionen, die in Abbildung 11.2 zur selben Zeit beginnen, dieselbe *MotorWinde* und können sich somit nicht überlappen. Die Einschränkung „können sich nicht überlappen“ ist eine Disjunktion von zwei linearen Ungleichungen, eine für jede mögliche Ordnung. Es zeigt sich, dass die Einführung von Disjunktionen das Scheduling mit Ressourcenbeschränkungen NP-hart macht.

▶ Abbildung 11.3 zeigt die Lösung mit der schnellsten Abschlusszeit, 115 Minuten. Das ist 30 Minuten länger als die 85 Minuten für den Zeitplan ohne Ressourcenbeschränkungen. Da es keinen Zeitpunkt gibt, zu dem beide Inspektoren benötigt werden; können wir einen von unseren beiden Inspektoren sofort an eine produktivere Position verschieben.

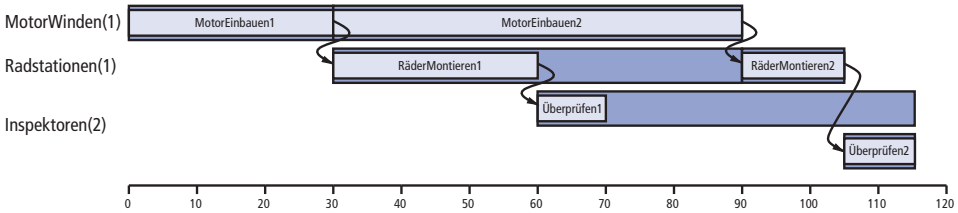


Abbildung 11.3: Eine Lösung für das Job-Shop-Scheduling-Problem von Abbildung 11.1, wobei Ressourcenbeschränkungen berücksichtigt werden. Die linke Spalte listet die drei Ressourcen auf, die Aktionen sind horizontal für die verbrauchten Ressourcen angetragen. Es gibt zwei mögliche Zeitpläne, abhängig davon, welche Montagelinie die Motorwinde zuerst benutzt; wir haben die optimale Lösung gezeigt, die 115 Minuten dauert.

Die Komplexität der Zeitplanung mit Ressourcenbeschränkungen sieht man häufig – sowohl in der Praxis als auch in der Theorie. Ein Problem, das 1963 formuliert wurde – die Suche nach dem optimalen Zeitplan für ein Problem mit nur zehn Maschinen und zehn Jobs mit je 100 Aktionen –, blieb 23 Jahre ungelöst (Lawler et al., 1993). Viele Ansätze wurden ausprobiert, unter anderem Verzweigen&Begrenzen, Simulated Annealing, Tabu-Suche, CSP und andere Techniken aus den *Kapiteln 3* und *4*. Eine einfache, aber verbreitete Heuristik ist der Algorithmus des **minimalen Spielraumes**: Bei jeder Iteration betrachtet er die noch nicht eingeplanten Aktionen, deren Vorgänger alle eingeplant sind, und plant diejenige mit dem geringsten Spielraum für den frühestmöglichen Start ein. Anschließend aktualisiert er die *ES*- und *LS*-Zeiten für jede betroffene Aktion und wiederholt das Ganze. Die Heuristik ähnelt der MRV-Heuristik im Erfüllbarkeitsproblem. In der Praxis funktioniert sie häufig gut, aber für unser Montageproblem ergibt sie eine Lösung von 130 Minuten und nicht die in Abbildung 11.3 gezeigte Lösung mit 115 Minuten.

Bis jetzt haben wir angenommen, dass die Menge der Aktionen und Reihenfolgebeschränkungen feststehend ist. Unter diesen Annahmen lässt sich jedes Zeitplanungsproblem durch eine nicht überlappende Sequenz lösen, die sämtliche Ressourcenkonflikte vermeidet, sofern jede Aktion für sich praktikabel ist. Erweist sich jedoch ein Zeitplanungsproblem als sehr schwierig, empfiehlt es sich möglicherweise nicht, das Problem auf diese Weise zu lösen – es kann besser sein, Aktionen und Beschränkungen erneut zu betrachten, falls dies zu einem wesentlich einfacheren Zeitplanungsproblem führt. Somit ist es sinnvoll, Planen und Zeitplanen zu *integrieren*, indem man Dauer und Überlappung während der Erstellung eines partiell geordneten Planes berücksichtigt. Mehrere der Planungsalgorithmen aus *Kapitel 10* können erweitert werden, um diese Information zu verarbeiten. Beispielsweise können partiell ordnende Planer Verletzungen der Ressourcenbeschränkung erkennen, ähnlich wie sie Konflikte mit kausalen Verknüpfungen erkennen. Heuristiken können abgeändert werden, um die Gesamtabschlusszeit eines Planes zu schätzen und nicht nur die Gesamtkosten der Aktionen. Dies ist momentan ein aktiver Forschungsbereich.

11.2 Hierarchisches Planen

Die Methoden zum Problemlösen und Planen der vorherigen Kapitel arbeiten alle mit einer feststehenden Menge von atomaren Aktionen. Aktionen können zu Sequenzen oder Verzweigungsnetzen zusammengefügt werden; moderne Algorithmen können Lösungen generieren, die Tausende von Aktionen enthalten.

Für Pläne, die durch das menschliche Gehirn ausgeführt werden, sind atomare Aktionen Muskelaktivierungen. Grob gerechnet haben wir 10^3 Muskeln zu aktivieren (je nach Zählweise 639, wobei aber viele mehrere Untereinheiten besitzen); ihre Aktivierung können wir vielleicht zehnmal pro Sekunde anpassen; insgesamt sind wir 10^9 Sekunden munter und lebendig. Somit umfasst ein menschliches Leben rund 10^{13} Aktionen, seien es zwei Größenordnungen mehr oder weniger. Auch wenn wir uns selbst darauf beschränkten, über wesentlich kürzere Zeithorizonte zu planen – zum Beispiel einen zweiwöchigen Urlaub auf Hawaii –, würde ein detaillierter motorischer Plan rund 10^{10} Aktionen enthalten. Das ist eine ganze Menge mehr als 1000.

Um diese Lücke zu überbrücken, müssen KI-Systeme wahrscheinlich das tun, was Menschen zu tun scheinen: auf höheren Abstraktionsebenen planen. Ein vernünftiger Plan für den Hawaii-Urlaub könnte lauten: „Gehe zum Flughafen San Francisco; nimm den Flug 11 der Hawaiian Airlines nach Honolulu; verbringe Urlaubsaktivitäten für zwei Wochen; nimm den Flug 12 der Hawaiian Airlines zurück nach San Francisco; gehe nach Hause.“ Bei einem derartigen Plan lässt sich die Aktion „Gehe zum Flughafen San Francisco“ als eigenständige Planungsaufgabe mit einer Lösung wie zum Beispiel „Fahre zum Dauerparkplatz; parke; nimm den Shuttle zum Terminal“ ansehen. Jede dieser Aktionen lässt sich wiederum weiter zerlegen, bis wir die Ebene der Aktionen erreichen, die ohne Überlegung ausgeführt werden kann, um die erforderlichen motorischen Steuerungsabläufe zu generieren.

In diesem Beispiel sehen wir, dass Planung sowohl vor als auch während der Ausführung des Planes auftreten kann; so würde man wahrscheinlich das Problem einer Routenplanung von einem Stellplatz auf dem Dauerparkplatz zur Haltestelle des Shuttle-Busses verschieben, bis ein bestimmter Stellplatz während der Ausführung gefunden worden ist. Somit bleibt diese konkrete Aktion vor der Ausführungsphase auf einer abstrakten Ebene. Die Diskussion zu diesem Thema verschieben wir bis *Abschnitt 11.3*. Hier konzentrieren wir uns auf den Aspekt der **hierarchischen Zerlegung**. Dieses Prinzip durchzieht fast sämtliche Versuche, die Komplexität in den Griff zu bekommen. Zum Beispiel wird komplexe Software aus einer Hierarchie von Unterprogrammen oder Objektklassen erstellt; Armeen operieren als Hierarchien von Einheiten; Regierungen und Unternehmen haben Hierarchien von Abteilungen, Filialen und Zweigstellen. Hierarchische Strukturen bieten vor allem den Vorteil, dass auf jeder Ebene der Hierarchie eine Berechnungsaufgabe, militärische Mission oder Verwaltungsfunktion auf eine kleine Anzahl von Aktivitäten auf der nächst niedrigeren Ebene reduziert wird, sodass die Berechnungskosten klein sind, um den korrekten Weg zu ermitteln, die Aktivitäten für das aktuelle Problem zu arrangieren. Nichthierarchische Methoden reduzieren dagegen eine Aufgabe auf eine große Anzahl von Einzelaktionen; für Probleme im großen Maßstab ist dies vollkommen untauglich.

11.2.1 Höhere Aktionen

Der grundlegende Formalismus, den wir übernommen haben, um die hierarchische Zerlegung zu verstehen, stammt aus dem Bereich der **hierarchischen Task-Netzwerk-Planung** (Hierarchical Task Network Planning) oder **HTN-Planung**. Wie in der klassischen Planung (*Kapitel 10*) nehmen wir vollständige Beobachtbarkeit und Determinismus sowie die Verfügbarkeit einer Menge von Aktionen – jetzt als **primitive Aktionen** bezeichnet – mit Standard-Vorbedingungs-Effekt-Schemas an. Das Schlüsselkonzept

ist die **höhere Aktion** oder **HLA** (High-Level Action) – zum Beispiel die Aktion „Gehe zum Flughafen San Francisco“ im weiter vorn angegebenen Beispiel. Jede HLA hat eine oder mehrere mögliche **Verfeinerungen**, zu einer Sequenz[1] von Aktionen, die jeweils eine HLA oder eine primitive Aktion (die per Definition keine Verfeinerungen hat) sein können. Zum Beispiel könnte die Aktion „Gehe zum Flughafen San Francisco“, die formal als *Gehen*(*Wohnung*, *SFO*) dargestellt wird, zwei mögliche Verfeinerungen haben, wie sie in ▶ Abbildung 11.4 zu sehen sind. Dieselbe Abbildung zeigt eine rekursive Verfeinerung für die Navigation in der Staubsaugerwelt: Um zu einem Ziel zu gelangen, führe einen Schritt aus und gehe dann zum Ziel.

```
Refinement(Gehen(Wohnung, SFO),
  STEPS: [Fahren(Wohnung, SFODauerparkplatz),
          Shuttle(SFODauerparkplatz, SFO)])
Refinement(Gehen(Wohnung, SFO),
  STEPS: [Taxi(Wohnung, SFO)])
```

```
Refinement(Navigieren([a, b], [x, y]),
  PRECOND: a = x ∧ b = y
  STEPS: [ ] )
Refinement(Navigieren([a, b], [x, y]),
  PRECOND: Verbunden([a, b], [a - 1, b])
  STEPS: [Links, Navigieren([a - 1, b], [x, y])] )
Refinement(Navigieren([a, b], [x, y]),
  PRECOND: Verbunden([a, b], [a + 1, b])
  STEPS: [Rechts, Navigieren([a + 1, b], [x, y])] )
...
```

Abbildung 11.4: Definitionen möglicher Verfeinerungen für zwei höhere Aktionen: zum Flughafen San Francisco gehen und in der Staubsaugerwelt navigieren. Beachten Sie im zweiten Fall die rekursive Natur der Verfeinerungen und die Verwendung von Vorbedingungen.

Diese Beispiele zeigen, dass höhere Aktionen und ihre Verfeinerungen das Wissen verkörpern, wie Dinge zu realisieren sind. Zum Beispiel besagen die Verfeinerungen für *Gehen*(*Wohnung*, *SFO*), dass Sie zum Flughafen gelangen, indem Sie selbst fahren oder sich ein Taxi nehmen; Milch kaufen, hinsetzen und den Springer auf e4 ziehen sind nicht zu betrachten.

Tipp

Bei einer HLA-Verfeinerung, die nur primitive Aktionen enthält, spricht man von einer **Implementierung** der HLA. Zum Beispiel implementieren in der Staubsaugerwelt die beiden Sequenzen [*Rechts*, *Rechts*, *Unten*] und [*Unten*, *Rechts*, *Rechts*] die HLA *Navigieren*([1, 3], [3, 2]). Eine Implementierung eines High-Level-Planes (einer Sequenz von HLAs) ist die Verkettung von Implementierungen jeder HLA in der Sequenz. Mit den Vorbedingungs-Effekt-Definitionen jeder primitiven Aktion ist es unkompliziert zu ermitteln, ob eine bestimmte Implementierung eines High-Level-Planes das Ziel erreicht. Wir können dann sagen, dass *ein High-Level-Plan das Ziel von einem gegebe-*

1 HTN-Planer erlauben oftmals eine Verfeinerung zu partiell geordneten Plänen und sie erlauben die Verfeinerungen von zwei unterschiedlichen HLAs in einem Plan, um Aktionen gemeinsam zu nutzen. Wir lassen diese wichtigen Komplikationen weg, um das Verständnis für die grundlegenden Konzepte der hierarchischen Planung nicht zu erschweren.

nen Zustand aus erreicht, wenn wenigstens eine seiner Implementierungen das Ziel von diesem Zustand aus erreicht. Die Aussage „wenigstens eine“ ist in dieser Definition entscheidend – nicht alle Implementierungen müssen das Ziel erreichen, weil der Agent entscheidet, welche Aktion er ausführt. Somit ist die Menge der möglichen Implementierungen in HTN-Planung – von denen jede ein anderes Ergebnis haben kann – nicht das Gleiche wie die Menge der möglichen Ergebnisse in nichtdeterministischer Planung. Dort haben wir gefordert, dass ein Plan für sämtliche Ergebnisse funktioniert, weil der Agent das Ergebnis nicht auswählen kann; die Natur tut es.

Der einfachste Fall ist eine HLA, die genau eine Implementierung besitzt. In diesem Fall können wir die Vorbedingungen und Effekte der HLA aus denen der Implementierung berechnen (siehe Übung 11.2) und dann die HLA genauso behandeln als wäre sie selbst eine primitive Aktion. Es lässt sich zeigen, dass die richtige Sammlung von HLAs in einer Zeitkomplexität der blinden Suche resultieren kann, die von einer exponentiellen Lösungstiefe zu einer linearen Lösungstiefe abfällt, obwohl es an sich keine triviale Aufgabe sein kann, eine derartige Sammlung von HLAs zu entwickeln. Wenn HLAs mehrere mögliche Implementierungen besitzen, gibt es zwei Optionen: Eine besteht darin, unter den Implementierungen nach einer zu suchen, die funktioniert, wie es *Abschnitt 11.2.2* zeigt, während bei der anderen direkt über die HLAs geschlossen wird – trotz der Vielzahl der Implementierungen –, wie *Abschnitt 11.2.3* erläutert. Die zweite Methode erlaubt die Ableitung von nachweisbar korrekten abstrakten Plänen, ohne dass man ihre Implementierungen betrachten muss.

11.2.2 Primitive Lösungen suchen

HTN-Planung wird oftmals mit einer einzelnen „höheren“ Aktion namens *Act* formuliert, wobei es darum geht, eine Implementierung von *Act* zu finden, die das Ziel erreicht. Dieser Ansatz ist vollkommen allgemein. Zum Beispiel können klassische Planungsprobleme wie folgt definiert werden: Stelle für jede primitive Aktion a_i eine Verfeinerung von *Act* mit Schritten $[a_i, Act]$ bereit. Dies erzeugt eine kursive Definition von *Act*, die es uns erlaubt, Aktionen hinzuzufügen. Allerdings benötigen wir eine Möglichkeit, die Rekursion zu beenden; dazu stellen wir eine weitere Verfeinerung für *Act* bereit, und zwar eine mit einer leeren Liste von Schritten und mit einer Vorbedingung, die gleich dem Ziel des Problems ist. Dies besagt, dass die richtige Implementierung darin besteht, nichts zu tun, wenn das Ziel bereits erreicht ist.

Der Ansatz führt zu einem einfachen Algorithmus: wiederholt eine HLA im aktuellen Plan auswählen und sie durch eine ihrer Verfeinerungen ersetzen, bis der Plan das Ziel erreicht. ▶ Abbildung 11.5 zeigt eine mögliche Implementierung, die auf Breitensuche für Bäume basiert. Pläne werden nach der Verschachtelungstiefe der Verfeinerungen und nicht nach der Anzahl der primitiven Schritte betrachtet. Es ist unkompliziert, eine Version des Algorithmus mit Graphensuche sowie Versionen mit Tiefensuche und iterativ vertiefender Suche zu entwerfen.

```
function HIERARCHICAL-SEARCH(problem, hierarchy) returns eine Lösung oder einen Fehler

  frontier ← eine FIFO-Warteschlange mit [Act] als einzigem ELement
  loop do
    if EMPTY?(frontier) then return failure
    plan ← POP(frontier) /* wählt den flachsten Plan in frontier */
    hla ← die erste HLA in plan oder null, falls keine vorhanden
    prefix, suffix ← die Aktionssubsequenzen vor und nach HLA in plan
    outcome ← RESULT(problem.INITIAL-STATE, prefix)
    if hla ist null then /* plan ist somit primitiv und outcome ist sein Ergebnis */
      if outcome erfüllt problem.GOAL then return plan
    else for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
      frontier ← INSERT(APPEND(prefix, sequence, suffix), frontier)
```

Abbildung 11.5: Eine Implementierung der hierarchischen Vorwärtsplanungssuche mit Breitensuche. Der dem Algorithmus anfangs bereitgestellte Plan ist [*Act*]. Die Funktion REFINEMENTS gibt eine Menge von Aktionssequenzen zurück, eine für jede Verfeinerung der HLA, deren Vorbedingungen durch den angegebenen Zustand *outcome* erfüllt werden.

Im Wesentlichen untersucht diese Form der hierarchischen Suche den Raum der Sequenzen entsprechend dem in der HLA-Bibliothek enthaltenen Wissen, wie Dinge zu realisieren sind. Ein Großteil des Wissens lässt sich kodieren, und zwar nicht einfach in den Aktionssequenzen, die in jeder Verfeinerung spezifiziert sind, sondern auch in den Vorbedingungen für die Verfeinerungen. Für manche Domänen konnten HTN-Planer riesige Pläne mit sehr wenig Suche generieren. Beispielsweise wurde O-PLAN (Bell und Tate, 1985), der HTN-Planen mit Zeitplanen kombiniert, für die Entwicklung von Produktionsplänen für Hitachi eingesetzt. Ein typisches Problem beinhaltet eine Produktlinie von 350 verschiedenen Produkten, 35 Montagemaschinen und über 2000 unterschiedlichen Operationen. Der Planer erzeugt einen 30-Tage-Plan mit drei 8-Stunden-Schichten pro Tag, wobei mehrere zehn Millionen Schritte zu berücksichtigen sind. Ein weiterer wichtiger Aspekt von HTN-Plänen ist, dass sie per Definition hierarchisch strukturiert sind; dies macht sie in der Regel für den Menschen verständlicher.

Die rechentechnischen Vorzüge der hierarchischen Suche werden deutlich, wenn man einen idealisierten Fall untersucht. Es sei angenommen, dass ein Planungsproblem eine Lösung mit d primitiven Aktionen besitzt. Für einen nichthierarchischen Vorwärtszustandsraumplaner mit b zulässigen Aktionen in jedem Zustand betragen die Kosten $O(b^d)$, wie *Kapitel 3* erläutert hat. Für einen HTN-Planer nehmen wir eine sehr regelmäßige Verfeinerungsstruktur: Jede nichtprimitive Aktion besitzt r mögliche Verfeinerungen, jede zu k Aktionen auf der nächst niedrigeren Ebene. Wir möchten wissen, wie viele unterschiedliche Verfeinerungsbäume es mit dieser Struktur gibt. Wenn es nun d Aktionen auf der primitiven Ebene gibt, dann beträgt die Anzahl der Ebenen unterhalb der Wurzel $\log_k d$, sodass sich die Anzahl der internen Verfeinerungsknoten zu

$$1 + k + k^2 + \ldots + k^{\log_k d - 1} = (d-1)/(k-1)$$

berechnet. Jeder interne Knoten hat r mögliche Verfeinerungen, sodass $r^{(d-1)/(k-1)}$ mögliche reguläre Zerlegungsbäume konstruiert werden könnten. Wie diese Formel zeigt, lassen sich für kleine r und große k enorme Einsparungen erzielen: Praktisch berechnen wir die k-te Wurzel der nichthierarchischen Kosten, wenn b und r vergleichbar groß sind. Kleine r und große k bedeuten eine Bibliothek von HLAs mit einer kleinen Anzahl von Verfeinerungen, die jeweils eine lange Aktionssequenz lie-

fern (die es uns nichtsdestotrotz erlaubt, jedes Problem zu lösen). Dies ist nicht immer möglich: Lange Aktionssequenzen, die sich in einem breiten Spektrum von Problemen nutzen lassen, sind äußerst kostbar.

Der Schlüssel für HTN-Planen ist also die Konstruktion einer Planbibliothek, die bekannte Methoden für die Implementierung komplexer höherer Aktionen enthält. Eine Methode für den Aufbau der Bibliothek ist es, die Methoden aus der problemlösenden Erfahrung zu *lernen*. Nach dem mühevollen Aufbau eines Planes von Grund auf kann der Agent den Plan in der Bibliothek speichern – als Methode, die höchste von der Aufgabe (Task) definierte höhere Aktion zu implementieren. Auf diese Weise wird der Agent mit der Zeit immer kompetenter, wenn neue Methoden auf alten Methoden aufgebaut werden. Ein wichtiger Aspekt dieses Lernprozesses ist die Fähigkeit, die konstruierten Methoden zu *verallgemeinern*, indem die für die Probleminstanz spezifischen Details eliminiert (z.B. der Name des Entwicklers oder die Adresse des jeweiligen Ortes) und nur die Schlüsselelemente des Planes beibehalten werden. Methoden für diese Verallgemeinerung sind in *Kapitel 19* beschrieben. Wir können uns nicht vorstellen, dass Menschen so kompetent agieren können, ohne dass sie einen vergleichbaren Mechanismus verwenden.

11.2.3 Abstrakte Lösungen suchen

Der hierarchische Suchalgorithmus des vorherigen Abschnittes verfeinert HLAs durchweg zu primitiven Aktionssequenzen, um zu ermitteln, ob ein Plan funktionsfähig ist. Dies widerspricht dem gesunden Menschenverstand: Es sollte sich feststellen lassen, ob man durch den höheren Plan mit zwei HLAs

[*Fahren*(*Wohnung*, *SFODauerparkplatz*), *Shuttle*(*SFODauerparkplatz*, *SFO*)]

zum Flughafen gelangt, ohne dass es erforderlich ist, eine genaue Route zu bestimmen, einen Parkplatz auszuwählen usw. Die Lösung scheint auf der Hand zu liegen: Wir formulieren Vorbedingungs-Effekt-Beschreibungen der HLAs, so wie wir aufschreiben, was die primitiven Aktionen bewirken. Anhand der Beschreibungen sollte es leicht sein zu beweisen, dass der höhere Plan das Ziel erreicht. Hierbei handelt es sich sozusagen um den Heiligen Gral der hierarchischen Planung. Denn wenn wir einen höheren Plan herleiten, der nachweisbar das Ziel erreicht, wobei er in einem kleinen Suchraum von höheren Aktionen arbeitet, können wir diesem Plan zustimmen und am Problem arbeiten, jeden Schritt des Planes zu verfeinern. Dadurch erhalten wir die gesuchte exponentielle Reduzierung. Damit dies funktioniert, muss jeder höhere Plan, der „behauptet“, das Ziel (aufgrund der Beschreibungen seiner Schritte) zu erreichen, das Ziel in der Tat in dem weiter vorn definierten Sinn erreichen: Er muss mindestens eine Implementierung haben, die das Ziel erreicht. Diese Eigenschaft wird als **Abwärtsverfeinerung** für HLA-Beschreibungen bezeichnet.

Prinzipiell ist es einfach, HLA-Beschreibungen zu formulieren, die die Abwärtsverfeinerungseigenschaft erfüllen: Solange die Beschreibungen *true* sind, muss jeder höhere Plan, der behauptet, das Ziel zu erreichen, dies auch tun – andernfalls treffen die Beschreibungen irgendwelche falschen Behauptungen darüber, was die HLAs tun. Wir haben bereits gesehen, wie sich wahre Beschreibungen für HLAs schreiben lassen, die genau eine Implementierung haben (Übung 11.2); ein Problem taucht auf,

wenn die HLA *mehrere* Implementierungen besitzt. Wie können wir die Effekte einer Aktion beschreiben, die auf viele unterschiedliche Arten implementiert werden kann?

Eine sichere Antwort (zumindest für Probleme, wo alle Vorbedingungen und Ziele positiv sind) besteht darin, nur die positiven Effekte, die durch *jede* Implementierung der HLA erreicht werden, und die negativen Effekte *jeder beliebigen* Implementierung einzuschließen. Dann würde die Abwärtsverfeinerungseigenschaft erfüllt. Leider ist diese Semantik für HLAs viel zu konservativ. Sehen Sie sich noch einmal die HLA *Gehen*(*Wohnung*, *SFO*) an, die zwei Verfeinerungen besitzt, und nehmen Sie zur Veranschaulichung eine einfache Welt an, in der man immer zum Flughafen fahren und parken kann, doch ein Taxi zu nehmen, *Bargeld* als Vorbedingung verlangt. In diesem Fall gelangt man mit *Gehen*(*Wohnung*, *SFO*) nicht immer zum Flughafen. Insbesondere scheitert die Aktion, wenn *Bargeld* falsch ist und wir *Bei*(*Agent*, *SFO*) als Effekt der HLA nicht zusichern können. Allerdings ist dies unsinnig; wenn der Agent kein *Bargeld* hätte, würde er selbst fahren. Vorauszusetzen, dass ein Effekt für jede Implementierung gilt, ist gleichbedeutend mit der Annahme, dass *jemand anderes* – ein Widersacher – die Implementierung auswählt. Er behandelt die Mehrfachergebnisse der HLA genauso, als wenn die HLA wie in *Abschnitt 4.3* eine **nichtdeterministische** Aktion wäre. Für unseren Fall würde der Agent selbst die Implementierung wählen.

Die Programmiersprachengemeinde hat den Begriff **pessimistischer (demonic) Nichtdeterminismus** geprägt, um den Fall zu bezeichnen, in dem ein Gegner die Auswahl trifft. Demgegenüber trifft der Agent beim **optimistischen (angelic) Nichtdeterminismus** die Auswahl selbst. Wir borgen uns diesen Begriff aus, um **optimistische Semantik** für HLA-Beschreibungen zu definieren. Das grundlegende Konzept, das für das Verständnis optimistischer Semantik erforderlich ist, ist die **erreichbare Menge** einer HLA: Für einen gegebenen Zustand s ist die erreichbare Menge für eine HLA h, geschrieben als REACH(s, h), die Menge von Zuständen, die durch beliebige Implementierung der HLA erreichbar sind. Entscheidend dabei ist, dass der Agent wählen kann, zu *welchem* Element der erreichbaren Menge er kommt, wenn er die HLA ausführt; somit ist eine HLA mit mehreren Verfeinerungen „leistungsfähiger“ als dieselbe HLA mit weniger Verfeinerungen. Wir können auch die erreichbare Menge einer Sequenz von HLAs definieren. Zum Beispiel ist die erreichbare Menge einer Sequenz $[h_1, h_2]$ die Vereinigungsmenge aller erreichbaren Mengen, die durch Anwendung von h_2 in jedem Zustand in der erreichbaren Menge von h_1 erhalten wird:

$$REACH(s,[h_1,h_2]) = \bigcup_{s' \in REACH(s,h_1)} REACH(s',h_2).$$

Mit diesen Definitionen erreicht ein High-Level-Plan – eine Sequenz von HLAs – das Ziel, wenn seine erreichbare Menge die Schnittmenge mit der Menge der Zielzustände ist. (Vergleichen Sie dies mit der wesentlich strengeren Bedingung für dämonische Semantik, wo jedes Element der erreichbaren Menge ein Zielzustand sein muss.) Wenn umgekehrt die erreichbare Menge keine Schnittmenge mit dem Ziel darstellt, funktioniert der Plan definitiv nicht. ▶ Abbildung 11.6 veranschaulicht diese Gedanken.

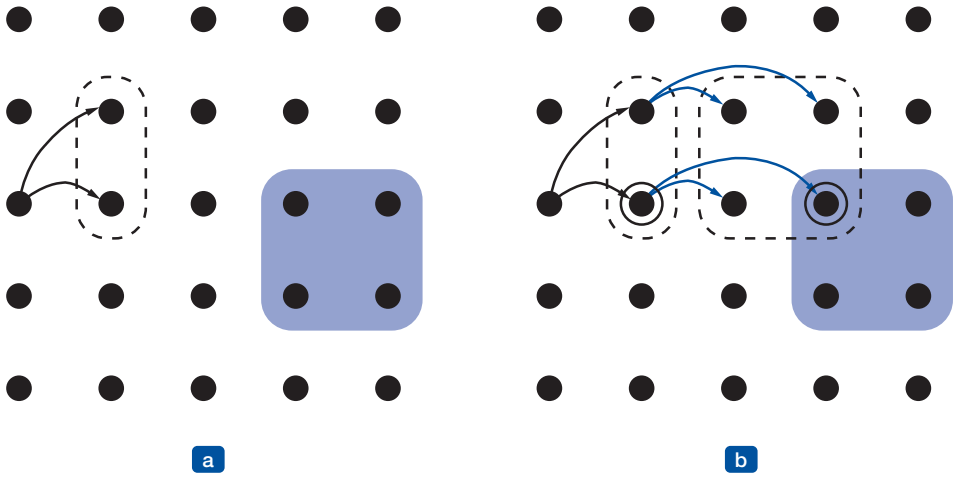


Abbildung 11.6: Schematische Beispiele für erreichbare Mengen. Die Menge der Zielzustände ist grau dargestellt. Schwarze und graue Pfeile zeigen mögliche Implementierungen von h_1 bzw. h_2 an. (a) Die erreichbare Menge von HLA h_1 ist ein Zustand s. (b) Die erreichbare Menge für die Sequenz $[h_1, h_2]$. Da es sich um eine Schnittmenge mit der Zielmenge handelt, erreicht die Sequenz das Ziel.

Das Konzept der erreichbaren Mengen liefert einen unkomplizierten Algorithmus: Suche unter höheren Plänen nach einem, dessen erreichbare Menge die Schnittmenge mit dem Ziel bildet; nachdem dies passiert ist, kann der Algorithmus diesem abstrakten Plan zustimmen (im Wissen, dass er funktioniert) und sich auf die weitere Verfeinerung des Planes konzentrieren. Wir kommen später auf die algorithmischen Fragen zurück; zuerst betrachten wir die Frage, wie die Effekte einer HLA – die erreichbare Menge für jeden möglichen Ausgangszustand – dargestellt werden. Wie bei den klassischen Aktionsschemas von *Kapitel 10* stellen wir die an jedem Fluenten vorgenommenen *Änderungen* dar. Stellen Sie sich einen Fluenten als Zustandsvariable vor. Eine primitive Aktion kann eine Variable hinzufügen, löschen oder auch unverändert lassen. (Bei bedingten Effekten, wie sie *Abschnitt 11.3.1* beschreibt, gibt es eine vierte Möglichkeit: eine Variable in ihr Gegenteil umkehren.) Eine HLA unter optimistischer Semantik kann mehr tun: Sie kann den Wert einer Variablen steuern, indem sie sie je nach gewählter Implementierung auf wahr oder falsch setzt. In der Tat kann eine HLA neun unterschiedliche Effekte auf eine Variable haben: Wenn die Variable anfangs wahr ist, kann sie sie immer auf wahr halten, immer auf falsch setzen oder eine Auswahl treffen; ist die Variable anfangs falsch, kann die Aktion sie immer auf falsch halten, immer auf wahr setzen oder eine Auswahl treffen. Die drei Auswahlen für jeden Fall lassen sich beliebig kombinieren, was neun ergibt. Die schriftliche Darstellung ist etwas anspruchsvoll. Wir verwenden das Symbol $\sim$ in der Bedeutung „möglicherweise, wenn sich der Agent so entscheidet“. Ein Effekt $\tilde{+}A$ bedeutet somit „möglicherweise A hinzufügen“, d.h. entweder A unverändert lassen oder auf wahr setzen. Analog heißt $\tilde{-}A$ „möglicherweise A löschen“ und $\tilde{\pm}A$ bedeutet „möglicherweise A hinzufügen oder löschen“. Die HLA *Gehen*(*Wohnung*, *SFO*) mit den beiden in Abbildung 11.4 gezeigten Verfeinerungen löscht zum Beispiel *Bargeld* (wenn der Agent entscheidet, ein Taxi zu nehmen), sodass sie den Effekt $\tilde{-}$ *Bargeld* haben sollte. Somit sehen wir, dass die Beschreibungen von HLAs im Prinzip aus den Beschreibungen ihrer Verfeinerungen ableitbar sind – in der Tat ist dies erforderlich, wenn wir echte

HLA-Beschreibungen haben möchten, sodass die Abwärtsverfeinerungseigenschaft gilt. Nehmen Sie nun die beiden folgenden Schemas für die HLAs h_1 und h_2 an:

$Action(h_1,$ PRECOND: $\neg A,$ EFFECT: $A \wedge \tilde{-}B)$
$Action(h_2,$ PRECOND: $\neg B,$ EFFECT: $\tilde{+}A \wedge \tilde{+}C)$

Das heißt, h_1 fügt A hinzu und löscht möglicherweise B, während h_2 möglicherweise A hinzufügt und volle Kontrolle über C hat. Wenn nun B im Ausgangszustand wahr ist und das Ziel $A \wedge C$ lautet, dann erreicht die Sequenz $[h_1, h_2]$ das Ziel: Wir wählen eine Implementierung von h_1, die B auf falsch setzt, und wählen dann eine Implementierung von h_2, die A wahr lässt und C auf wahr setzt.

Die obige Diskussion nimmt an, dass die Effekte einer HLA – die erreichbare Menge für jeden gegebenen Ausgangszustand – genau beschrieben werden können, indem der Effekt jeder Variablen beschrieben wird. Es wäre schön, wenn dies immer wahr wäre, doch in vielen Fällen können wir die Effekte nur annähern, weil eine HLA unendlich viele Implementierungen haben und beliebig unstetige erreichbare Mengen produzieren kann – ähnlich wie das unstetige Belief-State-Problem, das Abbildung 7.21 zeigt. Zum Beispiel haben wir gesagt, dass *Gehen*(*Wohnung*, *SFO*) möglicherweise *Bargeld* löscht; die Aktion fügt auch möglicherweise *Bei*(*Auto*, *SFODauerparkplatz*) hinzu; doch sie kann nicht beides tun – letztlich muss sie genau eines tun. Wie bei Belief States müssen wir gegebenenfalls angenäherte Beschreibungen formulieren. Wir verwenden zwei Arten der Annäherung: eine **optimistische Beschreibung** REACH$^+(s, h)$ einer HLA h kann die erreichbare Menge überbewerten, während eine **pessimistische Beschreibung** REACH$^-(s, h)$ die erreichbare Menge unterbewerten kann. Somit haben wir

$$\text{REACH}^-(s, h) \subseteq \text{REACH}(s, h) \subseteq \text{REACH}^+(s, h).$$

Zum Beispiel besagt eine optimistische Beschreibung von *Gehen*(*Wohnung*, *SFO*), dass sie *Bargeld* möglicherweise löscht und *Bei*(*Auto*, *SFODauerparkplatz*) möglicherweise hinzufügt. Ein weiteres gutes Beispiel ergibt sich aus dem 8-Puzzle, bei dem die Hälfte seiner Zustände von einem beliebigen Zustand aus erreichbar ist (siehe *Übung 3.5*): Die optimistische Beschreibung von *Act* könnte gut den gesamten Zustandsraum einschließen, da die genaue erreichbare Menge ziemlich unstetig ist.

Mit angenäherten Beschreibungen muss der Test, ob ein Plan das Ziel erreicht, etwas modifiziert werden. Wenn die optimistisch erreichbare Menge für den Plan keine Schnittmenge mit dem Ziel bildet, funktioniert der Plan nicht; bildet die pessimistisch erreichbare Menge eine Schnittmenge mit dem Ziel, funktioniert der Plan (▶ Abbildung 11.7(a)). Mit genauen Beschreibungen funktioniert ein Plan oder nicht, doch mit angenäherten Beschreibungen gibt es einen Mittelweg: Wenn die optimistische Menge eine Schnittmenge mit dem Ziel bildet, die pessimistische Menge jedoch nicht, können wir nicht sagen, ob der Plan funktioniert (▶ Abbildung 11.7(b)). Treten diese Umstände auf, lässt sich die Unbestimmtheit auflösen, indem der Plan verfeinert wird. Diese Situation kommt beim menschlichen Schließen sehr häufig vor. Zum Beispiel könnte man beim Planen des weiter oben erwähnten zweiwöchigen Hawaii-Urlaubes vorschlagen, zwei Tage auf jeder der sieben Inseln zu verbringen. Die Vorsicht gebietet es aber, dass dieser ehrgeizige Plan verfeinert wird, indem man die Details für den Transport zwischen den Inseln hinzufügt.

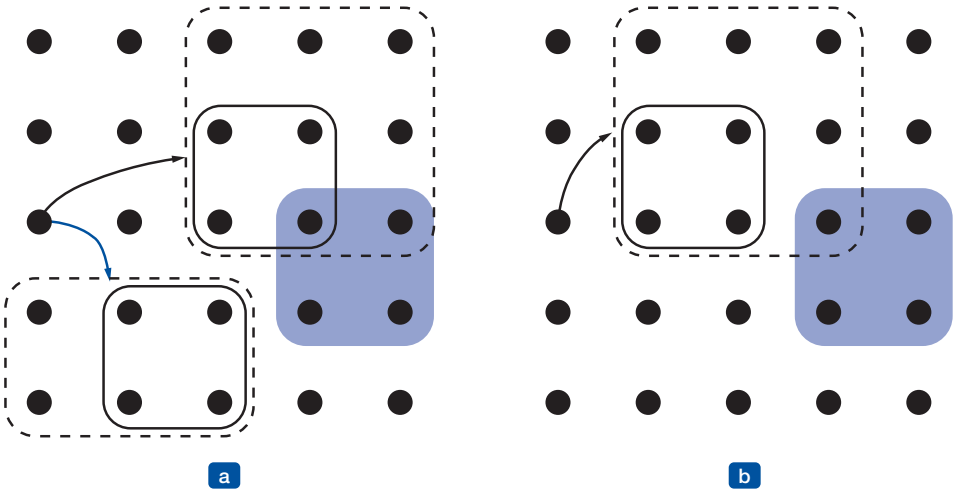


Abbildung 11.7: Zielerreichung für höhere Pläne mit angenäherten Beschreibungen. Die Menge der Zielzustände ist grau dargestellt. Für jeden Plan sind die pessimistisch (durchgehende Linien) und optimistisch (gestrichelte Linien) erreichbaren Mengen angegeben. (a) Der durch den schwarzen Pfeil bezeichnete Plan erreicht definitiv das Ziel, während der durch den grauen Pfeil bezeichnete Plan das Ziel definitiv nicht erreicht. (b) Ein Plan, der weiter verfeinert werden muss, um zu ermitteln, ob er das Ziel wirklich erreicht.

▶ Abbildung 11.8 zeigt einen Algorithmus für hierarchisches Planen mit angenäherten optimistischen Beschreibungen. Der Einfachheit halber haben wir das gleiche Gesamtschema wie vorher in Abbildung 11.5 verwendet, d.h. eine Breitensuche im Raum der Verfeinerungen. Wie eben erläutert, kann der Algorithmus Pläne erkennen, die funktionieren oder nicht, indem er die Schnittmengen aus optimistisch und pessimistisch erreichbaren Mengen mit dem Ziel überprüft. (Übung 11.4 befasst sich mit den Details, wie die erreichbaren Mengen eines Planes bei gegebenen angenäherten Beschreibungen jedes Schrittes berechnet werden.) Wenn ein funktionsfähiger abstrakter Plan gefunden wurde, zerlegt der Algorithmus das ursprüngliche Problem in Teilprobleme, eines für jeden Schritt des Planes. Den Ausgangszustand und das Ziel für jedes Teilproblem erhält man, indem auf einen garantiert erreichbaren Zielzustand über die Aktionsschemas für jeden Schritt des Planes zurückgegriffen wird. (*Abschnitt 10.2.2* hat erläutert, wie Regression arbeitet.) Abbildung 11.6(b) veranschaulicht das Grundprinzip: Der auf der rechten Seite mit einem Kreis versehene Zustand ist der garantiert erreichbare Zielzustand und der auf der linken Seite mit einem Kreis versehene Zustand ist das Zwischenziel, das durch Regression auf das Ziel über die letzte Aktion erhalten wird.

Die Fähigkeit, High-Level-Pläne zu bestätigen oder abzuweisen, kann dem ANGELIC-SEARCH-Algorithmus einen erheblichen rechentechnischen Vorteil gegenüber HIERARCHICAL-SEARCH verschaffen, der seinerseits einen großen Vorteil gegenüber dem guten alten BREADTH-FIRST-SEARCH hat. Nehmen Sie als Beispiel das Säubern einer großen Staubsaugerwelt, bestehend aus rechteckigen Räumen, die durch schmale Korridore miteinander verbunden sind. Es ist zweckmäßig, je eine HLA für *Navigieren* (wie in Abbildung 11.4 gezeigt) und eine für *CleanWholeRoom* zu verwenden. (Das Säubern des Raumes lässt sich mit der wiederholten Anwendung einer anderen HLA implementieren, um jede Reihe zu säubern.) Da es in dieser Domäne fünf Aktionen gibt, wachsen die Kosten für BREADTH-FIRST-SEARCH auf 5^d an, wobei d die Länge der kür-

zesten Lösung ist (ungefähr das Doppelte der Gesamtanzahl von Feldern); der Algorithmus kann nicht einmal Räume der Größe 2 × 2 bewältigen. HIERARCHICAL-SEARCH ist effizienter, leidet aber dennoch am exponentiellen Wachstum, da der Algorithmus sämtliche Reinigungswege ausprobiert, die mit der Hierarchie konsistent sind. Der ANGELIC-SEARCH-Algorithmus skaliert ungefähr linear mit der Anzahl der Felder – er stimmt einer guten High-Level-Sequenz zu und entfernt die anderen Optionen. Beachten Sie, dass das Reinigen einer Menge von Räumen, indem jeder Raum nacheinander gereinigt wird, wirklich nichts Spektakuläres ist: Es ist leicht für den Menschen, genau wegen der hierarchischen Struktur der Aufgabe. Wenn wir betrachten, wie schwer es dem Menschen fällt, kleine Rätselaufgaben wie zum Beispiel das 8-Puzzle zu lösen, scheint es wahrscheinlich, dass die menschliche Fähigkeit, komplexe Probleme zu lösen, zu einem großen Teil auf die Fertigkeit zurückzuführen ist, das Problem zu abstrahieren und zu zerlegen, um Kombinatorik zu eliminieren.

```
function ANGELIC-SEARCH(problem, hierarchy, initialPlan) returns Lösung oder Fehler

  frontier ← eine FIFO-Warteschlange mit initialPlan als einzigem Element
  loop do
    if EMPTY?(frontier) then return fail
    plan ← POP(frontier) /* wählt den flachsten Knoten in frontier */
    if REACH+ (problem.INITIAL-STATE, plan) Schnittmenge mit problem.GOAL then
      if plan ist primitiv then return plan /* REACH+ ist exakt für primitive Pläne */
      guaranteed ← REACH− (problem.INITIAL-STATE, plan) ∩ problem.GOAL
      if guaranteed ≠ { } und MAKING-PROGRESS(plan, initialPlan) then
         finalState ← beliebiges Element von guaranteed
        return DECOMPOSE(hierarchy, problem.INITIAL-STATE, plan, finalState)
      hla ← einige HLA in plan
      prefix, suffix ← die Aktionsteilsequenzen vor und nach hla in plan
      for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
        frontier ← INSERT(APPEND(prefix, sequence, suffix), frontier)

─────────────────────────────────────────────────────────────────────

function DECOMPOSE(hierarchy, s0, plan, sf) returns eine Lösung

  solution ← ein leerer Plan
  while plan ist nicht leer do
    action ← REMOVE-LAST(plan)
    si ← ein Zustand in REACH− (s0, plan) derart dass sf ∈ REACH− (si, action)
    problem ← ein Problem mit INITIAL-STATE = si und GOAL = sf
    solution ← APPEND(ANGELIC-SEARCH(problem, hierarchy, action), solution)
    sf ← si
  return solution
```

Abbildung 11.8: Ein hierarchischer Planungsalgorithmus, der optimistische (angelic) Semantik verwendet, um höhere Pläne, die funktionieren, zu identifizieren und zu bestätigen und höhere Pläne zu vermeiden, die nicht funktionieren. Das Prädikat MAKING-PROGRESS führt einen Test aus, um sicherzustellen, dass wir nicht in einer unendlichen Regression von Verfeinerungen steckenbleiben. Auf der obersten Ebene wird ANGELIC-SEARCH mit [*Act*] als anfänglichem Plan aufgerufen.

Der optimistische Ansatz lässt sich erweitern, um kostenoptimierte Lösungen zu suchen, indem man das Konzept der erreichbaren Menge verallgemeinert. Anstelle eines Zustandes, der erreichbar ist oder nicht, verwendet der Algorithmus einen Kostenwert für den effizientesten Weg, dorthin zu gelangen. (Die Kosten sind ∞ für nicht-erreichbare Zustände.) Die optimistischen und pessimistischen Beschreibungen

begrenzen diese Kosten. Auf diese Weise kann optimistische Suche nachweisbar optimale abstrakte Pläne finden, ohne ihre Implementierungen betrachten zu müssen. Das gleiche Konzept lässt sich verwenden, um effektive **hierarchische Lookahead**-Algorithmen für die Online-Suche im Stil von LRTA* (*Abschnitt 4.5.3*) zu erhalten. In gewisser Hinsicht spiegeln derartige Algorithmen Aspekte menschlicher Überlegung in Aufgaben wie zum Beispiel Planen einer Urlaubsreise nach Hawaii wider: Die Betrachtung von Alternativen geschieht anfangs auf einer abstrakten Ebene über längere Zeiträume hinweg; einige Teile des Planes bleiben bis zur Ausführungszeit ziemlich abstrakt, beispielsweise wie zwei Erholungstage in Molokai zu verbringen sind, während andere Teile bis ins Detail geplant werden, beispielsweise die zu buchenden Flüge und die zu reservierenden Unterkünfte. Ohne derartige Verfeinerungen gibt es keine Garantie, dass der Plan praktikabel ist.

11.3 Planen und Agieren in nicht deterministischen Domänen

In diesem Abschnitt erweitern wir die Planung, um partiell beobachtbare, nichtdeterministische und unbekannte Umgebungen zu behandeln. *Kapitel 4* hat die Suche in ähnlicher Weise erweitert und hier sind die Methoden ebenfalls ähnlich: **sensorloses Planen** (auch als **konformantes Planen** bezeichnet) für Umgebungen ohne Beobachtungen; **Kontingenzplanung** für partiell beobachtbare und nichtdeterministische Umgebungen sowie **Online-Planen** und **Neuplanen** für unbekannte Umgebungen.

Die grundlegenden Konzepte sind die gleichen wie in *Kapitel 4*, doch gibt es auch signifikante Unterschiede. Diese ergeben sich daraus, dass Planer mit faktorisierten und nicht mit atomaren Darstellungen zu tun haben. Dies beeinflusst die Art und Weise, wie wir die Fähigkeit des Agenten hinsichtlich Aktion und Beobachtung darstellen und auf welche Weise wir **Belief States** – die Menge der möglichen physischen Zustände, in denen sich der Agent befinden kann – für unbeobachtbare und partiell beobachtbare Umgebungen repräsentieren. Außerdem können wir von vielen der domänenunabhängigen Methoden profitieren, die *Kapitel 10* für die Berechnung von Suchheuristiken angegeben hat.

Sehen Sie sich folgendes Problem an: Ein Stuhl und ein Tisch sollen einheitlich aussehen – die gleiche Farbe haben. Im Ausgangszustand haben wir zwei Dosen Lack, wobei aber die Farbe und das Möbelstück unbekannt sind. Anfangs ist nur der Tisch im Sichtfeld des Agenten:

$$Init(Object(Tisch) \quad Object(Stuhl) \quad Dose(C_1) \quad Dose(C_2) \quad InSicht(Tisch))$$
$$Goal(Farbe(Stuhl, c) \quad Farbe(Tisch, c)).$$

Es gibt zwei Aktionen: den Deckel von einer Lackdose entfernen und ein Objekt mit dem Lack aus einer offenen Dose streichen. Die Aktionsschemas sind unkompliziert, mit einer Ausnahme: Wir erlauben es jetzt, dass Vorbedingungen und Effekte Variablen enthalten, die nicht zur Variablenliste der Aktion gehören. Das heißt, *Streichen* (x, *dose*) erwähnt nicht die Variable c, die die Farbe des Lacks in der Dose darstellt. Im vollständig beobachtbaren Fall ist dies nicht zulässig – wir müssten die Aktion *Streichen*(x, *dose*, c) benennen. Doch im partiell beobachtbaren Fall wäre uns die Farbe in der Dose vielleicht bekannt, vielleicht aber auch nicht. (Die Variable c ist allquantifiziert, genau wie alle anderen Variablen in einem Aktionsschema.)

Action(*DeckelEntfernen*(*dose*),
PRECOND: *Dose*(*dose*)
EFFECT: *Öffnen*(*dose*))
Action(*Streichen*(*x*, *dose*),
PRECOND: *Object*(*x*) ∧ *Dose*(*dose*) ∧ *Farbe* (*dose*, *c*) ∧ *Öffnen*(*dose*)
EFFECT: *Farbe* (*x*, *c*))

Um ein partiell beobachtbares Problem zu lösen, muss der Agent über die Wahrnehmungen schließen, die er erhält, wenn er den Plan ausführt. Die Wahrnehmung wird durch die Sensoren des Agenten bereitgestellt, wenn er tatsächlich agiert, doch wenn er plant, benötigt er ein Modell seiner Sensoren. In *Kapitel 4* wurde dieses Modell durch die Funktion PERCEPT(*s*) gegeben. Zum Planen erweitern wir PDDL mit einem neuen Schematyp, dem **Wahrnehmungsschema**:

Percept(*Farbe*(*x*, *c*),
PRECOND: *Object*(*x*) ∧ *InSicht*(*x*)
Percept(*Farbe*(*dose*, *c*),
PRECOND: *Dose*(*dose*) ∧ *InSicht* (*dose*) ∧ *Öffnen*(*dose*)

Das erste Schema besagt, dass wann immer ein Objekt in Sicht ist, der Agent die Farbe des Objektes wahrnimmt (das heißt, für das Objekt *x* wird der Agent den Wahrheitswert von *Farbe*(*x*, *c*) für alle *c* lernen.) Das zweite Schema besagt, dass der Agent die Farbe des Lacks in der Dose wahrnimmt, wenn eine offene Dose in Sicht ist. Da es keine exogenen Ereignisse in dieser Welt gibt, bleibt die Farbe eines Objektes gleich, selbst wenn es nicht wahrgenommen wird, bis der Agent eine Aktion ausführt, um die Farbe des Objektes zu ändern. Natürlich benötigt der Agent eine Aktion, die bewirkt, dass Objekte (nacheinander) in Sicht kommen:

Action(*Ansehen*(*x*),
PRECOND: *InSicht*(*y*) ∧ (*x* = *y*)
EFFECT: *InSicht*(*x*) ∧ ¬*InSicht*(*y*))

Für eine vollständig beobachtbare Umgebung würden wir ein *Percept*-Axiom ohne Vorbedingungen für jeden Fluenten einrichten. Ein sensorloser Agent besitzt andererseits überhaupt keine *Percept*-Axiome. Sogar ein sensorloser Agent ist in der Lage, das Färbungsproblem zu lösen. Eine Lösung besteht darin, irgendeine Lackdose zu öffnen und mit dem Lack sowohl Stuhl als auch Tisch anzustreichen, sodass sie **zwingend** dieselbe Farbe erhalten (selbst wenn der Agent nicht weiß, um welche Farbe es sich handelt).

Ein Kontingenzplanagent mit Sensoren kann einen besseren Plan generieren. Zuerst sieht er sich Tisch und Stuhl an, um deren Farben zu ermitteln; wenn sie bereits gleich sind, ist der Plan realisiert. Andernfalls sieht er sich die Lackdosen an; wenn der Lack in einer Dose dieselbe Farbe wie eines der Möbelstücke hat, wendet er diesen Lack auf das andere Stück an. Andernfalls werden beide Stücke mit einer beliebigen Farbe gestrichen.

Schließlich könnte ein Online-Planungsagent einen Kontingenzplan mit zunächst weniger Verzweigungen erzeugen – und vielleicht die Möglichkeit ignorieren, dass keine Dosenfarbe mit der Farbe der Möbelstücke übereinstimmt – und durch Neuplanen die Probleme angehen, wenn sie entstehen. Er könnte sich auch mit einer Unkorrektheit seiner Aktionsschemas befassen. Während ein Kontingenzplaner einfach annimmt, dass die Effekte einer Aktion immer erfolgreich sind – dass es mit dem

Lackieren des Stuhles getan ist – würde ein neuplanender Agent das Ergebnis überprüfen und einen zusätzlichen Plan erstellen, um irgendwelche unerwarteten Fehler zu korrigieren, wenn beispielsweise eine Stelle nicht lackiert ist oder die ursprüngliche Farbe durchscheint.

In der realen Welt kombinieren Agenten die genannten Ansätze. Autohersteller verkaufen Ersatzreifen und Airbags. Das ist der physische Inbegriff von Kontingenzplanverzweigungen, die für Reifenpannen oder Unfälle konzipiert sind. Die meisten Fahrzeugführer betrachten andererseits diese Möglichkeiten niemals; taucht ein Problem auf, reagieren sie wie neuplanende Agenten. Im Allgemeinen planen Agenten nur für Kontingenzen, die wichtige Konsequenzen und eine zu vernachlässigende Wahrscheinlichkeit für ihr Eintreten haben. Somit sollte ein Autofahrer, der über einen Trip durch die Sahara-Wüste nachdenkt, explizite Kontingenzpläne für Störungen planen, während eine Fahrt zum nächsten Supermarkt weniger vorausschauende Planung verlangt. Als Nächstes sehen wir uns die drei Konzepte detaillierter an.

11.3.1 Sensorloses Planen

Abschnitt 4.4.1 hat das grundlegende Konzept eingeführt, in einem Belief State nach einer Lösung für sensorlose Probleme zu suchen. Die Umwandlung eines sensorlosen Planungsproblems zu einem Belief-State-Planungsproblem funktioniert fast in der gleichen Weise wie in *Abschnitt 4.4.1* gezeigt, wobei die Unterschiede vor allem darin bestehen, dass das zugrunde liegende physische Transaktionsmodell durch eine Sammlung von Aktionsschemas dargestellt wird und sich der Belief State durch eine logische Formel statt einer explizit aufgezählten Menge von Zuständen darstellen lässt. Der Einfachheit halber nehmen wir an, dass das zugrunde liegende Planungsproblem deterministisch ist.

Der anfängliche Belief State für das sensorlose Lackierungsproblem kann *InSicht*-Fluenten ignorieren, weil der Agent keine Sensoren besitzt. Darüber hinaus nehmen wir die unveränderten Fakten $Object(Tisch) \wedge Object(Stuhl) \wedge Dose(C1) \wedge Dose(C2)$ als gegeben hin, weil diese in jedem Belief State gelten. Der Agent kennt weder die Farben der Dosen oder der Objekte noch ob die Dosen offen oder verschlossen sind. Er weiß aber, dass die Objekte und Dosen Farben haben: $\forall\, x\ \exists\, c\ Farbe\,(x, c)$. Nach der Skolemisierung (siehe *Abschnitt 9.5*) erhalten wir den anfänglichen Belief State:

$$b_0 = Farbe\,(x, C(x)).$$

In der klassischen Planung, bei der die **Annahme der geschlossenen Welt** getroffen wird, gehen wir davon aus, dass jeder in einem Zustand nicht erwähnte Fluent falsch ist, doch bei sensorloser (und partiell beobachtbarer) Planung müssen wir zu einer **Annahme der offenen Welt** übergehen, in der Zustände sowohl positive als auch negative Fluenten enthalten. Und wenn ein Fluent nicht erscheint, ist sein Wert unbekannt. Somit korrespondiert der Belief State genau mit der Menge der möglichen Welten, die die Formel erfüllen. Für diesen anfänglichen Belief State ist die folgende Aktionssequenz eine Lösung:

$$[DeckelEntfernen(Dose_1), Streichen(Stuhl, Dose_1), Streichen(Tisch, Dose_1)].$$

Wir sehen jetzt, wie der Belief State durch die Aktionssequenz weiterzuleiten ist, um zu zeigen, dass der endgültige Belief State das Ziel erfüllt.

Zuerst ist zu beachten, dass der Agent in einem gegebenen Belief State b jede Aktion betrachten kann, deren Vorbedingungen durch b erfüllt sind. (Die anderen Aktionen können nicht verwendet werden, weil das Übergangsmodell keine Effekte von Aktionen definiert, deren Vorbedingungen eventuell nicht erfüllt sind.) Gemäß *Gleichung (4.4)* lautet die allgemeine Formel für die Aktualisierung des Belief State b für eine gegebene Aktion a in einer deterministischen Welt wie folgt:

$$b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ und } s \in b\},$$

wobei RESULT_P das physische Übergangsmodell definiert. Vorerst nehmen wir an, dass der anfängliche Belief State immer eine Konjunktion von Literalen ist, d.h. eine 1-KNF-Formel. Um den neuen Belief State b' zu konstruieren, müssen wir betrachten, was für jedes Literal ℓ in jedem physischen Zustand s in b passiert, wenn die Aktion angewandt wird. Für Literale, deren Wahrheitswert bereits in b bekannt ist, wird der Wahrheitswert b' aus dem aktuellen Wert sowie den Hinzufügen- und Löschlisten der Aktion berechnet. (Wenn zum Beispiel ℓ in der Löschliste der Aktion steht, wird $\neg\ell$ zu b' hinzugefügt.) Wie sieht es mit einem Literal aus, dessen Wahrheitswert in b unbekannt ist? Hier sind drei Fälle zu unterscheiden:

1. Wenn die Aktion ℓ hinzufügt, wird ℓ in b' wahr sein, unabhängig von seinem Anfangswert.
2. Wenn die Aktion ℓ löscht, wird ℓ in b' falsch sein, unabhängig von seinem Anfangswert.
3. Wenn die Aktion ℓ nicht beeinflusst, behält ℓ seinen Anfangswert (der unbekannt ist) und erscheint nicht in b'.

Folglich sehen wir, dass die Berechnung von b' fast identisch mit dem beobachtbaren Fall ist, der durch *Gleichung (10.1)* spezifiziert wurde:

$$b = \text{RESULT}(b, a) = (b - \text{DEL}(a)) \cup \text{ADD}(a).$$

Die Mengensemantik können wir nicht wirklich verwenden, da wir (1) sicherstellen müssen, dass b' nicht sowohl ℓ als auch $\neg\ell$ enthält, und (2) Atome nicht gebundene Variablen enthalten können. Dennoch wird $\text{RESULT}(b, a)$ berechnet, indem beginnend mit b jedes Atom, das in $\text{DEL}(a)$ erscheint, auf false gesetzt und jedes Atom, das in $\text{ADD}(a)$ erscheint, auf wahr gesetzt wird. Wenn wir zum Beispiel $DeckelEntfernen(Dose_1)$ auf den anfänglichen Belief State b_0 anwenden, erhalten wir:

$$b_1 = Farbe(x, C(x)) \wedge \ddot{O}ffnen(Dose_1).$$

Wenn wir die Aktion $Streichen(Stuhl, Dose_1)$ anwenden, wird die Vorbedingung $Farbe(Dose_1, c)$ durch das bekannte Literal $Farbe(x, C(x))$ mit der Bindung $\{x/Dose_1, c/C(Dose_1)\}$ erfüllt. Der neue Belief State lautet:

$$b_2 = Farbe(x, C(x)) \wedge \ddot{O}ffnen(Dose_1) \wedge Farbe(Stuhl, C(Dose_1)).$$

Schließlich wenden wir die Aktion $Streichen(Tisch, Dose_1)$ an, um

$$b_3 = Farbe(x, C(x)) \wedge \ddot{O}ffnen(Dose_1) \wedge Farbe(Stuhl, C(Dose_1)) \wedge Farbe(Tisch, C(Dose_1))$$

zu erhalten.

Der endgültige Belief State erfüllt das Ziel $Farbe(Tisch, c) \wedge Farbe(Stuhl, c)$, wobei die Variable c an $C(Dose_1)$ gebunden ist.

Tipp

Die obige Analyse der Aktualisierungsregel hat eine sehr wichtige Tatsache gezeigt: *Die als Konjunktionen von Literalen definierte Familie von Belief States wird unter Aktualisierungen, die durch PDDL-Aktionsschemas definiert sind, geschlossen*. Wenn also der Belief State als Konjunktion von Literalen beginnt, liefert jede Aktualisierung eine Konjunktion von Literalen. Das bedeutet, dass sich in einer Welt mit n Fluenten jeder beliebige Belief State als Konjunktion der Größe $O(n)$ darstellen lässt. Angesichts der Tatsache, dass es 2^n Zustände in der Welt gibt, ist dieses Ergebnis sehr ermutigend. Es besagt, dass wir alle Teilmengen dieser 2^n Zustände, die wir überhaupt brauchen, kompakt darstellen können. Darüber hinaus ist der Überprüfungsvorgang nach Belief States, die Teilmengen oder Obermengen von vorher besuchten Zuständen sind, ebenfalls einfach, zumindest im aussagenlogischen Fall.

Der Wermutstropfen bei diesem angenehmen Bild ist, dass es nur für Aktionsschemas funktioniert, die die gleichen Effekte für alle Zustände haben, in denen ihre Vorbedingungen erfüllt sind. Genau diese Eigenschaft macht die Bewahrung der 1-KNF-Belief-State-Darstellung möglich. Sobald der Effekt vom Zustand abhängen kann, werden Abhängigkeiten zwischen Fluenten eingeführt und die 1-KNF-Eigenschaft geht verloren. Betrachten Sie beispielsweise die einfache Staubsaugerwelt, die *Abschnitt 3.2.1* definiert hat.

Angenommen, die Fluenten sind *BeiL* und *BeiR* für die Position des Roboters sowie *SauberL* und *SauberR* für den Zustand der Quadrate. Gemäß der Definition des Problems hat die Aktion *Saugen* keine Vorbedingung – sie kann immer ausgeführt werden. Die Schwierigkeit besteht darin, dass ihre Effekte von der Position des Roboters abhängen: Befindet sich der Roboter bei *BeiL*, ist das Ergebnis *SauberL*, doch wenn er *BeiR* ist, lautet das Ergebnis *SauberR*. Bei derartigen Aktionen benötigen unsere Aktionsschemas etwas Neues: **bedingte Effekte**. Diese haben die Syntax „**when** *condition: effect*“, wobei *condition* eine logische Formel ist, die den resultierenden Zustand beschreibt. Für die Staubsaugerwelt haben wir

Action(*Saugen*,
EFFECT: when *BeiL*: *SauberL* ∧ when *BeiR*: *SauberR*)

Wird dies auf den anfänglichen Belief State *True* angewendet, ist der resultierende Belief State (*BeiL* ∧ *SauberL*) ∨ (*BeiR* ∧ *SauberR*), was keine 1-KNF mehr darstellt. (Dieser Übergang ist in Abbildung 4.14 zu sehen.) Im Allgemeinen können bedingte Effekte zufällige Abhängigkeiten zwischen den Fluenten in einem Belief State induzieren, was im ungünstigsten Fall zu Belief States exponentieller Größe führt.

Es ist wichtig, den Unterschied zwischen Vorbedingungen und bedingten Effekten zu verstehen. *Alle* bedingten Effekte, deren Bedingungen erfüllt sind, lassen ihre Effekte anwenden, um den resultierenden Zustand zu generieren; wenn keine Bedingungen erfüllt sind, bleibt der Ergebniszustand unverändert. Ist andererseits eine *Vorbedingung* nicht erfüllt, ist die Aktion nicht anwendbar und der Ergebniszustand ist undefiniert. Vom Standpunkt der sensorlosen Planung aus ist es besser, bedingte Effekte als eine nicht anwendbare Aktion zu haben. Zum Beispiel könnten wir *Saugen* wie folgt in zwei Aktionen mit unbedingten Effekten aufteilen:

Action(*SaugenL*,
PRECOND: *BeiL*; *EFFECT*: *SauberL*)
Action(*SaugenR*,
PRECOND: *BeiR*; *EFFECT*: *SauberR*)

Jetzt haben wir nur unbedingte Schemas, sodass die Belief States alle in 1-KNF bleiben; leider können wir die Anwendbarkeit von *SaugenL* und *SaugenR* im anfänglichen Belief State nicht ermitteln.

Es scheint dann unumgänglich, dass bei nichttrivialen Problemen unstetige Belief States vorkommen, genau wie wir sie beim Problem der Zustandsschätzung für die Wumpus-Welt kennengelernt haben (siehe Abbildung 7.21). Dort wurde die Lösung vorgeschlagen, eine konservative Annäherung an den exakten Belief State zu verwenden. Zum Beispiel kann der Belief State in 1-KNF verbleiben, wenn er alle Literale enthält, deren Wahrheitswerte ermittelt werden können, und alle anderen Literale als unbekannt behandelt. Auch wenn dieser Ansatz in dem Sinne, dass er niemals einen inkorrekten Plan erzeugt, *einwandfrei* ist, ist er *unvollständig*, weil er möglicherweise keine Lösungen für Probleme finden kann, die zwangsläufig Interaktionen zwischen Literalen beinhalten. Ein triviales Beispiel: Wenn das Ziel für den Roboter darin besteht, auf einem sauberen Quadrat zu sein, ist [*Saugen*] eine Lösung. Ein sensorloser Agent, der auf 1-KNF-Belief-States besteht, wird diese Lösung jedoch nicht finden.

Eine vielleicht bessere Lösung ist es, nach Aktionssequenzen zu suchen, die den Belief State so einfach wie möglich halten. Zum Beispiel generiert in der sensorlosen Vakuumwelt die Aktionssequenz [*Rechts*, *Saugen*, *Links*, *Saugen*] die folgende Sequenz von Belief States:

$$b_0 = True$$
$$b_1 = BeiR$$
$$b_2 = BeiR \wedge SauberR$$
$$b_3 = BeiL \wedge SauberR$$
$$b_4 = BeiL \wedge SauberR \wedge SauberL.$$

Das heißt, der Agent kann das Problem lösen, wobei ein 1-KNF-Belief-State erhalten bleibt, selbst wenn einige Sequenzen (z.B. diejenigen, die mit Saugen beginnen) außerhalb von 1-KNF fallen. Die allgemeine Lektion ist bei Menschen nicht verloren: Wir führen immer kleine Aktionen aus (die Zeit kontrollieren, die Taschen abklopfen, um sich zu vergewissern, die Autoschlüssel dabeizuhaben, Verkehrszeichen entziffern, wenn wir durch eine Stadt fahren), um Unsicherheit zu beseitigen und unseren Belief State überschaubar zu halten.

Es gibt noch einen gänzlich anderen Ansatz für das Problem der nicht handhabbaren unstetigen Belief States: sich überhaupt nicht mit deren Berechnung abzugeben. Angenommen, der anfängliche Belief State ist b_0 und wir würden gern den Belief State wissen, der aus der Aktionssequenz $[a_1, ..., a_m]$ resultiert. Anstatt ihn explizit zu berechnen, stellen wir ihn einfach als „b_0 dann $[a_1, ..., a_m]$“ dar. Diese Darstellung ist bequem, aber unmissverständlich, und zudem recht prägnant – $O(n + m)$, wobei n die Größe des anfänglichen Belief States (in 1-KNF angenommen) und m die maximale Länge einer Aktionssequenz sind. Als Belief-State-Darstellung weist sie allerdings einen Nachteil auf: Unter Umständen ist der Berechnungsaufwand sehr hoch, um zu ermitteln, ob das Ziel erfüllt oder eine Aktion anwendbar ist.

Die Berechnung kann als Test auf logische Konsequenz implementiert werden: Wenn A_m die Sammlung von Nachfolgerzustandsaxiomen darstellt, die erforderlich ist, um Vorkommen der Aktionen $a_1, ..., a_m$ zu definieren – wie es *Abschnitt 10.4.1* für SATPLAN erläutert hat –, und G_m zusichert, dass das Ziel nach m Schritten wahr ist, erreicht der Plan das Ziel, wenn $b_0 \wedge A_m \models G_m$, d.h. wenn $b_0 \wedge A_m \wedge \neg G_m$ nicht erfüllbar ist.

Mit einem modernen SAT-Solver ist es durchaus möglich, dies wesentlich schneller zu bewerkstelligen, als den vollständigen Belief State zu berechnen. Wenn zum Beispiel keine der Aktionen in der Sequenz einen bestimmten Zielfluenten in ihrer Hinzufügenliste hat, entdeckt der Solver dies sofort. Er hilft auch, wenn Teilergebnisse über den Belief State – zum Beispiel Fluenten, die bekanntermaßen wahr oder falsch sind – zwischengespeichert werden, um darauffolgende Berechnungen zu vereinfachen. Das letzte Stück im sensorlosen Planungspuzzle ist eine Heuristikfunktion, um die Suche zu leiten. Die Bedeutung der Heuristikfunktion ist die gleiche wie für die klassische Planung: eine (vielleicht zulässige) Schätzung der Kosten, um das Ziel vom gegebenen Belief State aus zu erreichen. Mit Belief States haben wir einen zusätzlichen Fakt: Das Lösen irgendeiner Teilmenge eines Belief State ist zwangsläufig leichter als das Lösen des Belief State:

$$\textit{if } b_1 \subseteq b_2 \text{ dann } h^*(b_1) \leq h^*(b_2).$$

Folglich ist jede zulässige Heuristik, die für eine Teilmenge berechnet wird, auch für den Belief State selbst zulässig. Als Kandidaten kommen vor allem Singleton-Teilmengen infrage, d.h. einzelne physische Zustände. Wir können jede beliebige Sammlung von Zuständen s_1, ..., s_N nehmen, die sich im Belief State b befinden, eine beliebige zulässige Heuristik h aus *Kapitel 10* anwenden und

$$H(b) = max\{h(s_1), ..., h(s_N)\}$$

als Heuristik für das Lösen von b zurückgeben. Wir könnten auch einen Planungsgraphen direkt auf b selbst verwenden: Wenn es sich um eine Konjunktion von Literalen (1-KNF) handelt, setzt man diese Literale einfach auf die anfängliche Zustandsebene des Graphen. Wenn b nicht in 1-KNF vorliegt, ist es durchaus möglich, Mengen von Literalen zu finden, die zusammen b zur Folge haben. Liegt b beispielsweise in disjunktiver Normalform (DNF) vor, ist jeder Term der DNF-Formel eine Konjunktion von Literalen, die b zur Folge haben, und kann die anfängliche Ebene eines Planungsgraphen bilden. Wie vorher können wir das Maximum der Heuristiken nehmen, die sich aus den einzelnen Mengen von Literalen ergeben. Außerdem können wir unzulässige Heuristiken wie zum Beispiel die Heuristik „Löschlisten ignorieren“ (*Abschnitt 10.2.3*) verwenden, die in der Praxis gut zu funktionieren scheint.

11.3.2 Kontingenzplanen

Wie *Kapitel 4* erläutert hat, eignet sich Kontingenzplanung – das Generieren von Plänen mit bedingten Verzweigungen je nach Wahrnehmung – für Umgebungen mit partieller Beobachtbarkeit und/oder Nichtdeterminismus. Für das partiell beobachtbare Lackierungsproblem mit den weiter vorn angegebenen Wahrnehmungsaxiomen sieht eine mögliche Lösung wie folgt aus:

```
[Ansehen(Tisch), Ansehen(Stuhl),
  if Farbe(Tisch, c) ∧ Farbe(Stuhl, c) then NoOp
    else [DeckelEntfernen(Dose1), Ansehen(Dose1), DeckelEntfernen(Dose2),
Ansehen(Dose2),
      if Farbe(Tisch, c) ∧ Farbe(dose, c) then Streichen(Stuhl, dose)
      else if Farbe(Stuhl, c) ∧ Farbe(dose, c) then Streichen(Tisch, dose)
      else [Streichen(Stuhl, Dose1), Streichen(Tisch, Dose1)]]]
```

Variablen in diesem Plan sollten als existentiell quantifiziert betrachtet werden; die zweite Zeile besagt, dass der Agent nichts zu tun braucht, um das Ziel zu erreichen, wenn eine Farbe c existiert, die die Farbe des Tisches und des Stuhls ist. Wenn dieser Plan ausgeführt wird, kann ein Kontingenzplanagent seinen Belief State als logische

Formel verwalten und jede Verzweigungsbedingung auswerten, indem er ermittelt, ob der Belief State eine logische Konsequenz der Bedingung oder ihrer Negation ist. (Der Kontingenzplanungsalgorithmus muss sicherstellen, dass der Agent niemals in einem Belief State endet, wo der Wahrheitswert der Bedingungsformel unbekannt ist.) Beachten Sie, dass bei Bedingungen erster Stufe die Formel in mehr als einer Weise erfüllt sein kann; zum Beispiel könnte die Bedingung $Farbe(Tisch, c) \wedge Farbe(dose, c)$ durch $\{dose/Dose_1\}$ und durch $\{dose/Dose_2\}$ erfüllt sein, wenn beide Dosen die gleiche Farbe wie der Tisch haben. In diesem Fall kann der Agent eine beliebige Substitution auswählen, um sie auf den Rest des Planes anzuwenden.

Wie *Abschnitt 4.4.2* gezeigt hat, erfolgt die Berechnung des neuen Belief State nach einer Aktion und darauffolgenden Wahrnehmung in zwei Phasen. Die erste Phase berechnet den Belief State nach der Aktion, genau wie für den sensorlosen Agenten:

$$\hat{b} = (b - \text{DEL}(a)) \cup \text{ADD}(a),$$

wobei wir wie zuvor einen Belief State angenommen haben, der als Konjunktion von Literalen dargestellt wird. Die zweite Phase ist etwas komplizierter. Nehmen Sie an, dass Wahrnehmungsliterale $p_1, ..., p_k$ empfangen werden. Man könnte meinen, dass wir diese einfach zum Belief State hinzufügen müssten; tatsächlich können wir auch schließen, dass die Vorbedingungen für sensorische Eingabe erfüllt werden. Wenn nun eine Wahrnehmung p genau ein Wahrnehmungsaxiom $Percept(p, \text{PRECOND}: c)$ hat, wobei c eine Konjunktion von Literalen ist, können diese Literale zusammen mit p in den Belief State gebracht werden. Hat p andererseits mehr als ein Wahrnehmungsaxiom, dessen Vorbedingungen entsprechend dem vorhergesagten Belief State $\hat{b}$ gelten könnten, müssen wir die *Disjunktion* der Vorbedingungen hinzufügen. Offensichtlich bringt dies den Belief State aus 1-KNF heraus und wirft die gleichen Komplikationen wie bedingte Effekte auf, mit fast den gleichen Lösungsklassen.

Mit einem Mechanismus für das Berechnen exakter oder angenäherter Belief States können wir Kontingenzpläne mit einer Erweiterung der AND-OR-Vorwärtssuche über den in *Abschnitt 4.4* verwendeten Belief States generieren. Aktionen mit nichtdeterministischen Effekten – die einfach mithilfe einer Disjunktion im EFFECT des Aktionsschemas definiert werden – lassen sich mit kleineren Änderungen an die Berechnung der Belief-State-Aktualisierung und ohne Änderung an den Suchalgorithmus anpassen.[2] Auf die Heuristikfunktion sind viele der für sensorloses Planen vorgeschlagenen Methoden auch im partiell beobachtbaren, nichtdeterministischen Fall anwendbar.

11.3.3 Online-Neuplanen

Stellen Sie sich einen Punktschweißroboter in einem Autowerk vor. Die schnellen und genauen Bewegungen des Roboters wiederholen sich ständig, wenn die einzelnen Autos die Taktstraße passieren. Obwohl dies technisch eindrucksvoll ist, scheint der Roboter wahrscheinlich nicht allzu *intelligent* zu sein, da es sich bei der Bewegung um eine feststehende, vorprogrammierte Sequenz handelt. Es sieht so aus, dass der Roboter „nicht weiß, was er tut“ in einer sinnvollen Art und Weise. Nehmen Sie nun an, dass eine schlecht befestigte Tür vom Auto abfällt, gerade als der Roboter eine Schweißnaht

2 Wenn zyklische Lösungen für ein nichtdeterministisches Problem erforderlich sind, muss die AND-OR-Suche zu einer mit Schleifen arbeitenden Version wie zum Beispiel LAO* (Hansen und Zilberstein, 2001) verallgemeinert werden.

anbringen will. Der Roboter ersetzt schnell seinen Schweißaktuator durch einen Greifer, nimmt die Tür auf, untersucht sie auf Kratzer, befestigt sie wieder am Auto, sendet dem Bandbetreuer eine E-Mail, wechselt zurück zum Schweißaktuator und setzt seine Arbeit fort. Urplötzlich scheint das Verhalten des Roboters *zielgerichtet* statt rein mechanisch zu sein; wir nehmen an, es resultiert nicht aus einem riesigen, vorberechneten Kontingenzplan, sondern aus einem Online-Neuplanungsvorgang – das bedeutet, dass der Roboter *wissen muss*, was er zu tun versucht.

Neuplanen setzt eine Form von **Ausführungsüberwachung** voraus, um das Erfordernis für einen neuen Plan zu ermitteln. Ein derartiges Erfordernis entsteht, wenn ein Kontingenzplanungsagent es überdrüssig wird, jede winzige Eventualität einzuplanen, beispielsweise ob der Himmel auf seinen Kopf fallen könnte.[3] Manche Zweige eines partiell konstruierten Kontingenzplans können einfach *Neuplanen* (Replan) sagen; wenn ein derartiger Zweig während der Ausführung erreicht wird, kehrt der Agent zum Planungsmodus zurück. Wie bereits erwähnt, sind die Entscheidungen, welcher Teil des Problems im Voraus zu lösen ist und wie viel für das Neuplanen bleiben soll, von Kompromissen zwischen möglichen Ereignissen mit unterschiedlichen Kosten und Eintrittswahrscheinlichkeiten geprägt. Niemand möchte erst bei einer Autopanne mitten in der Sahara-Wüste darüber nachdenken, ob genügend Wasser an Bord ist.

Neuplanen kann auch erforderlich sein, wenn das Modell des Agenten von der Welt falsch ist. Dabei kann es sich um eine **fehlende Vorbedingung** für eine Aktion handeln – vielleicht weiß der Agent nicht, dass sich der Deckel einer Lackbüchse oftmals nur mithilfe eines Schraubendrehers entfernen lässt. Im Modell kann auch ein **Effekt fehlen** – zum Beispiel kommt es vor, dass beim Streichen eines Objekts Lack auf den Boden tropft. Vielleicht haben wir es auch mit einer **fehlenden Zustandsvariablen** im Modell zu tun – so ist etwa im weiter vorn angegebenen Modell nicht berücksichtigt, welche Menge Lack eine Dose enthält, wie die Aktionen diese Menge beeinflussen oder dass die Menge nicht null sein darf. Denkbar ist auch, dass dem Modell Vorkehrungen für **exogene Ereignisse** fehlen, beispielsweise dass jemand die Lackdose umkippt. Zu den exogenen Ereignissen können auch Änderungen im Ziel gehören, beispielsweise eine zusätzliche Forderung, dass der Tisch und der Stuhl nicht schwarz gestrichen sein dürfen. Ohne die Fähigkeit zu überwachen und neu zu planen, ist das Verhalten eines Agenten wahrscheinlich äußerst instabil, wenn er sich auf die absolute Korrektheit seines Modells verlässt.

Der Online-Agent hat zu entscheiden, wie sorgfältig die Umgebung zu überwachen ist. Wir unterscheiden drei Ebenen:

- **Aktionsüberwachung:** Vor dem Ausführen einer Aktion verifiziert der Agent, ob alle Vorbedingungen noch gelten.
- **Planüberwachung:** Vor dem Ausführen einer Aktion verifiziert der Agent, ob der restliche Plan noch erfolgreich sein wird.
- **Zielüberwachung:** Vor dem Ausführen einer Aktion überprüft der Agent, ob es eine bessere Menge von Zielen gibt, die er versuchen könnte zu erreichen.

3 Im Jahr 1954 wurde eine Mrs. Hodges aus Alabama von einem Meteorit getroffen, der durch ihr Dach geschlagen war. 1992 traf ein Stück des Mbale-Meteoriten einen kleinen Jungen auf dem Kopf; zum Glück wurde die Fallgeschwindigkeit durch Bananenblätter gebremst (Jenniskens et al., 1994). Und 2009 behauptete ein deutscher Junge, dass seine Hand von einem erbsengroßen Meteoriten getroffen worden sei. Die erwähnten Vorfälle führten nicht zu schweren Verletzungen, was darauf hindeutet, dass die Notwendigkeit für eine Vorausplanung derartiger Eventualitäten manchmal doch übertrieben ist.

▶ Abbildung 11.9 zeigt eine schematische Darstellung der Aktionsüberwachung. Der Agent verfolgt sowohl seinen ursprünglichen Plan (mit *whole plan* gekennzeichnet) als auch den Teil des Planes, der bislang noch nicht ausgeführt wurde (mit *plan* bezeichnet). Nachdem die ersten Schritte des Planes ausgeführt wurden, erwartet der Agent, im Zustand E zu sein. Der Agent stellt jedoch fest, dass er sich tatsächlich im Zustand O befindet. Er muss dann den Plan reparieren, indem er einen Punkt P im ursprünglichen Plan sucht, zu dem er zurückgehen kann. (Möglicherweise ist P bereits der Zielzustand G.) Der Agent versucht, die Gesamtkosten des Planes zu minimieren: den Reparaturteil (von O bis P) plus die Fortsetzung (von P nach G).

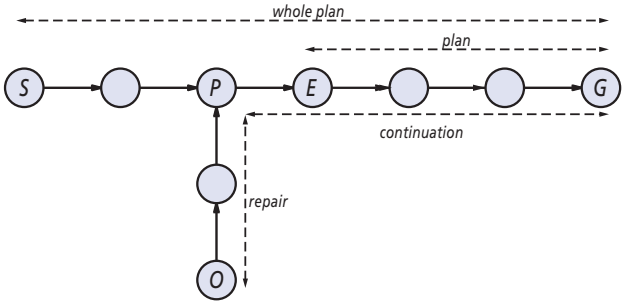


Abbildung 11.9: Vor der Ausführung erzeugt der Planer einen Plan, hier als *whole plan* bezeichnet, um von S nach G zu gelangen. Der Agent führt Schritte des Planes aus, bis er erwartet, im Zustand E zu sein, stellt dann aber fest, dass er sich tatsächlich in O befindet. Der Agent plant dann neu, um mit einer minimalen Reparatur (*repair*) und Fortsetzung (*continuation*) den Zustand G zu erreichen.

Kehren wir nun zum Beispielproblem zurück, einen Stuhl und einen Tisch mit gleicher Farbe zu streichen. Angenommen, der Agent erzeugt folgenden Plan:

```
[Ansehen(Tisch), Ansehen(Stuhl),
  if Farbe(Tisch, c) ∧ Farbe(Stuhl, c) then NoOp
    else [DeckelEntfernen(Dose1), Ansehen(Dose1),
      if Farbe(Tisch, c) ∧ Farbe(Dose1, c) then Streichen(Stuhl, Dose1)
      else REPLAN]]
```

Der Agent ist nun bereit, den Plan auszuführen. Angenommen, der Agent beobachtet, dass der Tisch und die Lackdose weiß sind und der Stuhl schwarz ist. Dann führt er *Streichen*(*Stuhl*, $Dose_1$) aus. Ein klassischer Planer würde an dieser Stelle einen Triumph verkünden; der Plan ist ausgeführt worden. Doch ein Online-Agent mit Ausführungsüberwachung muss die Vorbedingungen des verbleibenden leeren Planes überprüfen – dass der Tisch und der Stuhl die gleiche Farbe haben. Angenommen, der Agent nimmt wahr, dass sie nicht die gleiche Farbe haben – tatsächlich ist der Stuhl jetzt grau gefleckt, weil der schwarze Lack durchscheint. Dann muss der Agent in *whole plan* eine anzustrebende Position und eine Reparaturaktionssequenz finden, um dorthin zu gelangen. Der Agent erkennt, dass der aktuelle Zustand mit der Vorbedingung vor der Aktion *Streichen*(*Stuhl*, $Dose_1$) identisch ist, sodass er die leere Sequenz für *repair* auswählt und seinen *plan* gleich der [*Streichen*]-Sequenz macht, die er eben ausprobiert hat. Mit diesem neuen Plan wird die Ausführungsüberwachung fortgesetzt und die *Streichen*-Aktion erneut probiert. Dieses Verhalten setzt sich in einer Schleife fort, bis der Stuhl als vollständig lackiert wahrgenommen wird. Die Schleife wird durch einen Prozess von Planen-Ausführen-Neuplanen und nicht durch eine explizite Schleife in einem Plan erzeugt. Außerdem ist zu beachten, dass der ursprüngliche Plan nicht jede Eventu-

alität berücksichtigen muss. Wenn der Agent den mit REPLAN markierten Schritt erreicht, kann er einen neuen Plan generieren (und vielleicht $Dose_2$ einbeziehen).

Die Aktionsüberwachung ist eine einfache Methode der Ausführungsüberwachung, führt jedoch manchmal zu einem wenig intelligenten Verhalten. Nehmen wir zum Beispiel an, dass es weder schwarzen noch weißen Lack gibt und der Agent einen Plan konstruiert, um das Lackierungsproblem zu lösen, indem Stuhl und Tisch rot lackiert werden sollen. Angenommen, es ist nur für den Stuhl genügend roter Lack vorhanden. Bei Aktionsüberwachung beginnt der Agent und streicht den Stuhl rot. Dann bemerkt er, dass er keinen Lack mehr hat und den Tisch nicht streichen kann. Jetzt erstellt er einen neuen Plan, eine Reparatur – und streicht eventuell sowohl Stuhl als auch Tisch grün an. Ein Agent mit Planungsüberwachung kann den Fehler erkennen, sobald ein Zustand erreicht ist, von dem aus der restliche Plan nicht mehr funktioniert. Somit verschwendet er keine Zeit damit, den Stuhl rot zu streichen. Die Planungsüberwachung realisiert dies, indem die Vorbedingungen für den Erfolg des gesamten restlichen Planes geprüft werden – d.h. die Vorbedingungen jedes Schrittes im Plan, ausgenommen die Vorbedingungen, die durch einen anderen Schritt im restlichen Plan erreicht werden. Die Planungsüberwachung beendet die Ausführung eines aussichtslosen Planes so früh wie möglich, anstatt fortzufahren, bis der Fehler tatsächlich auftritt.[4] Außerdem erlaubt Planungsüberwachung einen **glücklichen Zufall** – einen zufälligen Erfolg. Wenn jemand des Weges kommt und den Tisch rot anmalt, während der Agent gleichzeitig den Stuhl rot anmalt, sind die Vorbedingungen des endgültigen Planes erfüllt (das Ziel wurde erreicht) und der Agent kann früher nach Hause gehen.

Es ist relativ einfach, einen Planungsalgorithmus so abzuändern, dass jede Aktion im Plan mit den Vorbedingungen der Aktion kommentiert (annotiert) und damit eine Aktionsüberwachung ermöglicht wird. Etwas schwieriger ist es, eine Planungsüberwachung zu aktivieren. Planer mit partieller Ordnung und Planungsgraphen besitzen den Vorteil, dass sie bereits Strukturen aufgebaut haben, die die für Planüberwachung erforderlichen Beziehungen enthalten. Das Erweitern von Zustandsraumplanern mit den notwendigen Annotationen lässt sich durch sorgfältige Buchführung realisieren, wenn die Zielfluenten durch den Plan regressiert werden.

Nachdem wir eine Methode für Überwachung und Neuplanen beschrieben haben, müssen wir fragen: „Funktioniert es?“ Dies ist eine überraschend knifflige Frage. Wenn wir meinen: „Können wir garantieren, dass der Agent immer das Ziel erreicht?“, ist die Antwort „Nein“, weil der Agent versehentlich in eine Sackgasse geraten kann, aus der keine Reparatur herausführt. Zum Beispiel könnte der Staubsaugeragent ein fehlerhaftes Modell von sich selbst haben und nicht wissen, dass sich seine Batterien erschöpfen können. Ist das einmal passiert, kann er keinerlei Pläne mehr reparieren. Wenn wir Sackgassen ausschließen – davon ausgehen, dass ein Plan existiert, um das Ziel aus einem beliebigen Zustand in der Umgebung zu erreichen – und annehmen, dass die Umgebung wirklich nichtdeterministisch ist in dem Sinne, dass ein derartiger Plan immer eine bestimmte Erfolgswahrscheinlichkeit bei einem gegebenen Ausführungsversuch hat, wird der Agent schließlich das Ziel erreichen.

4 Planungsüberwachung bedeutet, dass wir schließlich (nach mehreren Hundert Seiten) einen Agenten haben, der intelligenter als ein Mistkäfer ist (siehe *Abschnitt 2.2.2*). Ein Agent mit Planungsüberwachung würde bemerken, dass er die Mistkugel nicht mehr besitzt, und neu planen, um eine neue Kugel zu formen und sie in das Loch zu stecken.

Schwierigkeiten treten auf, wenn eine Aktion tatsächlich nicht deterministisch ist, sondern vielmehr von einer bestimmten Vorbedingung abhängt, über die der Agent nicht Bescheid weiß. Beispielsweise kann manchmal eine Lackdose leer sein, sodass das Anstreichen aus dieser Dose keinen Effekt hat. Dies lässt sich durch keinerlei Maßnahmen ändern.[5] Eine Lösung besteht darin, zufällig aus der Menge der möglichen Reparaturpläne auszuwählen, anstatt jedes Mal denselben Plan zu probieren. In diesem Fall könnte der Reparaturplan funktionieren, eine andere Dose zu öffnen. Ein besserer Ansatz ist es, ein besseres Modell zu **lernen**. Jeder Voraussagefehler ist eine Gelegenheit zum Lernen; ein Agent sollte in der Lage sein, sein Modell der Welt entsprechend seinen Wahrnehmungen zu modifizieren. Von diesem Zeitpunkt an wird der Neuplaner in der Lage sein, einen Reparaturplan zu erzeugen, der das Problem an der Wurzel packt, anstatt auf gut Glück einen guten Reparaturplan auszuwählen. Die *Kapitel 18* und *19* befassen sich mit derartigen Lernformen.

11.4 Multiagenten-Planen

Bisher haben wir angenommen, dass nur ein Agent erfasst, plant und agiert. Wenn es mehrere Agenten in der Umgebung gibt, sieht sich jeder Agent einem **Multiagenten-Planungsproblem** gegenüber, in dem er seine eigenen Ziele zu erreichen versucht, wobei ihm andere Agenten helfen oder ihn behindern.

Zwischen den reinen Einzelagenten- und echten Multiagenten-Fällen liegt ein breites Spektrum von Problemen, die in unterschiedlichem Maß eine Zerlegung des monolithischen Agenten zeigen. Ein Agent mit mehreren Effektoren, der parallel operieren kann – so ist beispielsweise ein Mensch in der Lage, gleichzeitig zu schreiben und zu sprechen –, muss **Multieffektor-Planen** realisieren, um jeden Effektor zu verwalten und dabei positive und negative Interaktionen unter den Aktoren zu behandeln. Wenn die Effektoren physisch in getrennte Einheiten entkoppelt sind – wie in einer Flotte von Übergaberobotern in einer Fabrik – wird Multieffektor-Planen zum **Mehrkörper-Planen**. Ein Mehrkörperproblem ist dennoch ein „standardmäßiges" Einzelagentenproblem, solange sich die relevanten Sensorinformationen, die jeder Körper sammelt, in einen – entweder zentralen oder lokal zu jedem Körper angeordneten – Pool stellen lassen, um eine gemeinsame Schätzung des Weltzustandes zu bilden, die dann die Ausführung des Gesamtplanes informiert; in diesem Fall agieren die verschiedenen Körper als Einzelkörper. Wenn Kommunikationseinschränkungen dies unmöglich machen, haben wir das, was manchmal als **dezentralisiertes Planungsproblem** bezeichnet wird; dies mag ein unzutreffender Name sein, da die Planungsphase zentralisiert, die Ausführungsphase aber zumindest teilweise entkoppelt ist. In diesem Fall muss der für jeden Körper konstruierte Teilplan gegebenenfalls explizite kommunikative Aktionen mit anderen Körpern einschließen. Zum Beispiel können mehrere Erkundungsroboter, die ein großflächiges Gebiet abdecken, oftmals ohne Funkkontakt zu anderen Robotern sein und sollten ihre Erkundungen während der Zeiten einer stabilen Funkverbindung bekannt machen.

Wenn eine einzelne Entität die Planung realisiert, gibt es wirklich nur genau ein Ziel, das alle Körper zwangsläufig gemeinsam anstreben. Handelt es sich bei den Körpern um getrennte Agenten, die selbstständig planen, können sie dennoch identische Ziele verfol-

5 Vergebliche Wiederholung einer Planreparatur ist genau das Verhalten, das von der Grabwespe an den Tag gelegt wird (*Abschnitt 2.2.2*).

gen; zum Beispiel haben zwei menschliche Tennisspieler, die in einem Doppelteam Tennis spielen, das gemeinsame Ziel, das Match zu gewinnen. Jedoch selbst mit gemeinsamen Zielen sind die Mehrkörper- und Multiagenten-Fälle recht verschieden. In einem Doppelteam aus Mehrkörper-Robotern diktiert ein einziger Plan, welcher Körper auf dem Platz wohin geht und welcher Körper den Ball schlägt. In einem Multiagenten-Doppelteam entscheidet dagegen jeder Agent selbst, was zu tun ist; ohne eine Methode zur **Koordinierung** könnten sich beide Agenten entscheiden, denselben Abschnitt des Platzes abzudecken, und jeder überlässt den Ball dem anderen, damit dieser ihn schlägt.

Der deutlichste Fall eines Multiagenten-Problems liegt natürlich vor, wenn die Agenten unterschiedliche Ziele haben. Im Tennis stehen die Ziele von zwei gegnerischen Teams in direktem Konflikt zueinander, was zur Nullsummensituation von *Kapitel 5* führt. Zuschauer könnte man als Agenten ansehen, falls ihr Beifall oder ihre Missachtung ein signifikanter Faktor ist und durch die Führung des Spielers beeinflusst werden kann; andernfalls lassen sie sich – genau wie das Wetter – als Aspekt der Natur ansehen, der für die Absichten der Spieler als gleichgültig angenommen wird.[6]

Schließlich sind manche Systeme eine Mischung von zentralisierter und Multiagenten-Planung. Zum Beispiel könnte ein Transportunternehmen die Routen seiner LKWs und Flugzeuge jeden Tag zentral und offline planen, manche Aspekte jedoch für autonome Entscheidungen durch Fahrer und Piloten offenlassen, die auf Verkehrs- und Wettersituationen individuell reagieren können. Außerdem werden die Ziele der Firma und ihrer Mitarbeiter in gewissem Umfang durch die Zahlung von Prämien (Gehältern und Bonuszahlungen) zur Deckung gebracht – ein sicheres Zeichen dafür, dass dies ein echtes Multiagenten-System ist.

Die in die Multiagenten-Planung einfließenden Fragen lassen sich grob in zwei Kategorien gliedern. Die erste (in *Abschnitt 11.4.1* behandelte) betrifft Fragen der Darstellung und Planung für mehrere gleichzeitige Aktionen; diese Fragen treten in allen Einrichtungen von Multieffektor- bis Multiagenten-Planung auf. Die zweite (in *Abschnitt 11.4.2* behandelte) bezieht sich auf Fragen von Kooperation, Koordination und Wettbewerb, wie sie in echten Multiagenten-Einrichtungen entstehen.

11.4.1 Planen mit mehreren simultanen Aktionen

Vorerst behandeln wir die Multieffektor-, Mehrkörper- und Multiagenten-Einrichtungen in der gleichen Weise und benennen sie generisch als **Multiaktor**-Einrichtungen, wobei sich der generische Begriff **Aktor** auf Effektoren, Körper und Agenten bezieht. Dieser Abschnitt soll nun herausarbeiten, wie Übergangsmodelle, korrekte Pläne und effiziente Planungsalgorithmen für die Multiaktor-Einrichtung zu definieren sind. Ein korrekter Plan ist ein Plan, der das Ziel erreicht, wenn er durch die Aktoren ausgeführt wird. (In der echten Multiagenten-Einrichtung müssen die Agenten natürlich nicht immer einig sein, einen bestimmten Plan auszuführen, doch wenigstens wissen sie, welche Pläne funktionieren *würden*, wenn sie ihrer Ausführung *zustimmen*. Der Einfachheit halber nehmen wir perfekte **Synchronisierung** an: Jede Aktion benötigt die gleiche Zeit und an jedem Punkt im verknüpften Plan laufen die Aktionen simultan ab.

6 Wir entschuldigen uns bei den Bewohnern von Großbritannien, wo allein der Gedanke an ein Tennisspiel garantiert, dass es regnet.

Wir beginnen mit dem Übergangsmodell; für den deterministischen Fall ist dies die Funktion RESULT(s, a). In der Einzelagenteneinrichtung kann es b unterschiedliche Entscheidungen für die Aktion geben; b kann ziemlich groß sein, insbesondere für Darstellungen der ersten Stufe mit vielen Objekten, auf denen zu agieren ist. Nichtsdestotrotz bieten Aktionsschemas eine prägnante Darstellung. In der Multiaktor-Einrichtung mit n Aktoren wird die Einzelaktion a durch eine **gemeinsame Aktion** $\langle a_1, ..., a_n \rangle$ ersetzt, wobei a_i die vom i-ten Aktor ausgeführte Aktion ist. Sofort sehen wir zwei Probleme: Erstens müssen wir das Übergangsmodell für b^n unterschiedliche gemeinsame Aktionen beschreiben; zweitens haben wir ein Verknüpfungsplanungsproblem mit einem Verzweigungsfaktor von b^n.

Nachdem man die Aktoren zu einem Multiaktor-System mit einem riesigen Verzweigungsfaktor zusammengestellt hatte, konzentrierte sich die Forschung zur Multiaktor-Planung vor allem darauf, die Aktoren bis zum möglichen Ausmaß zu *entkoppeln*, sodass die Komplexität des Problems linear mit n statt exponentiell wächst. Wenn die Aktoren nicht untereinander interagieren – zum Beispiel n Aktoren, die jeweils ein Spiel Solitär spielen –, können wir einfach n separate Probleme lösen. Können wir etwas erreichen, was dieser exponentiellen Verbesserung nahe kommt, wenn die Aktoren **lose gekoppelt** sind? Dies ist natürlich eine zentrale Frage in vielen Bereichen der KI. Explizit kennengelernt haben wir sie im Kontext der CSPs, wobei „baumartige" Constraint-Graphen effiziente Lösungsmethoden geliefert haben (siehe *Abschnitt 6.5*), sowie im Kontext von disjunkten Musterdatenbanken (*Abschnitt 3.6.3*) und additiver Heuristiken für die Planung (*Abschnitt 10.2.3*).

Beim Standardansatz für locker gekoppelte Probleme gibt man vor, die Probleme seien vollkommen entkoppelt, und korrigiert dann die Interaktionen. Für das Übergangsmodell bedeutet dies das Schreiben von Aktionsschemas, als ob die Aktoren unabhängig agieren würden. Sehen wir uns nun an, wie dies für das Doppeltennisproblem funktioniert. Wir nehmen an, dass die Spieler des Teams zu einem bestimmten Zeitpunkt im Spiel das Ziel haben, den Ball zurückzuschlagen, der ihnen zugespielt wurde, und sicherzustellen, dass mindestens einer von ihnen das Netz abdeckt.

▶ Abbildung 11.10 zeigt, wie ein erster Pass bei einer Multiaktor-Definition aussehen könnte.

$$
\begin{aligned}
&\mathit{Aktoren}(A, B)\\
&\mathit{Init}(\mathit{Bei}(A, \mathit{LinkeGrundlinie}) \wedge (B, \mathit{RechtesNetz}) \wedge\\
&\quad \mathit{Erreicht}(\mathit{Ball}, \mathit{RechteGrundlinie})) \wedge \mathit{Partner}(A, B) \wedge \mathit{Partner}(B, A)\\
&\mathit{Goal}(\mathit{Zurückgespielt}(\mathit{Ball}) \wedge (\mathit{Bei}(a, \mathit{RechtesNetz}) \vee \mathit{Bei}(a, \mathit{LinkesNetz}))\\
&\mathit{Action}(\mathit{Schlagen}(\mathit{aktor}, \mathit{Ball}),\\
&\quad \text{PRECOND: } \mathit{Erreicht}(\mathit{Ball}, \mathit{pos}) \wedge \mathit{Bei}(\mathit{aktor}, \mathit{pos})\\
&\quad \text{EFFECT: } \mathit{Zurückgespielt}(\mathit{Ball}))\\
&\mathit{Action}(\mathit{Go}(\mathit{aktor}, \mathit{zu}),\\
&\quad \text{PRECOND: } \mathit{Bei}(\mathit{aktor}, \mathit{pos}) \wedge \mathit{zu} = \mathit{pos},\\
&\quad \text{EFFECT: } \mathit{Bei}(\mathit{aktor}, \mathit{zu}) \wedge \neg \mathit{Bei}(\mathit{aktor}, \mathit{pos}))
\end{aligned}
$$

Abbildung 11.10: Das Doppeltennisproblem. Zwei Aktoren *A* und *B* spielen zusammen und können sich an einer von vier Positionen befinden: *LinkeGrundlinie*, *RechteGrundlinie*, *LinkesNetz* und *RechtesNetz*. Der Ball kann nur zurückgespielt werden, wenn sich ein Spieler an der richtigen Stelle befindet. Jede Aktion muss den Aktor als Argument einschließen.

Mit dieser Definition ist es leicht zu sehen, dass der folgende **Verknüpfungsplan** funktioniert:

PLAN 1:
A : [$Go(A, RechteGrundlinie)$, $Schlagen(A, Ball)$]
B : [$NoOp(B)$, $NoOp(B)$]

Probleme treten allerdings auf, wenn ein Plan beide Agenten den Ball zur selben Zeit schlagen lässt. In der realen Welt funktioniert dies nicht, doch das Aktionsschema für *Schlagen* besagt, dass der Ball erfolgreich zurückgeschlagen wird. Technisch gesehen besteht die Schwierigkeit darin, dass Vorbedingungen den Zustand einschränken, in dem eine Aktion erfolgreich ausgeführt werden kann, aber andere Aktionen, die sie durcheinanderbringen könnten, nicht beschränken.

Um dies zu lösen, erweitern wir Aktionsschemas mit einem neuen Feature: einer **Liste nebenläufiger Aktionen**, die angibt, welche Aktionen parallel auszuführen sind oder nicht parallel ausgeführt werden dürfen. Zum Beispiel ließe sich die Aktion *Schlagen* wie folgt beschreiben:

$Action(Schlagen(a, Ball)$,
CONCURRENT: $b \neq a \Rightarrow \neg Schlagen(b, Ball)$
PRECOND: $Erreicht(Ball, pos) \wedge Bei(a, pos)$
EFFECT: $Zurückgespielt(Ball))$

Mit anderen Worten hat die Aktion *Schlagen* ihren besagten Effekt nur, wenn keine andere *Schlagen*-Aktion durch einen anderen Agenten zur selben Zeit auftritt. (Im SATPLAN-Ansatz würde dies durch ein partielles **Aktionsausschlussaxiom** behandelt.) Für bestimmte Aktionen wird der gewünschte Effekt nur erreicht, wenn eine andere Aktion gleichzeitig auftritt. Zum Beispiel sind zwei Agenten erforderlich, um eine Kühltasche mit Getränken zum Tennisplatz zu tragen:

$Action(Tragen(a, kühltasche, hier, dort)$,
CONCURRENT: $b \neq a \wedge Tragen(b, kühltasche, hier, dort)$
PRECOND: $Bei(a, hier) \wedge Bei(kühltasche, hier) \wedge Kühltasche\ (kühltasche)$
EFFECT: $Bei(a, dort) \wedge Bei(kühltasche, dort) \wedge \neg Bei(a, hier) \wedge$
$\neg Bei(kühltasche, hier))$

Mit derartigen Aktionsschemas lässt sich jeder der in *Kapitel 10* beschriebenen Planungsalgorithmen mit nur geringen Modifikationen anpassen, um Multiaktor-Pläne zu generieren. Soweit die Kopplung unter den Teilplänen lose ist – d.h. dass nebenläufige Einschränkungen während der Plansuche nur selten ins Spiel kommen –, würde man erwarten, dass die verschiedenen Heuristiken, die für die Einzelagentenplanung abgeleitet wurden, ebenso im Multiaktor-Kontext effektiv sind. Wir könnten diesen Ansatz mit den Verfeinerungen der beiden letzten Kapitel erweitern – HTNs, partielle Beobachtbarkeit, Bedingtheit, Ausführungsüberwachung und Neuplanen –, doch würde dies über den Rahmen dieses Buches hinausgehen.

11.4.2 Planen mit mehreren Agenten: Kooperation und Koordination

Wir kommen nun zur echten Multiagenten-Einrichtung, in der jeder Agent seinen eigenen Plan erstellt. Zunächst nehmen wir an, dass die Ziele und die Wissensbasis gemeinsam genutzt werden. Man könnte meinen, dass sich dadurch eine Reduzierung auf den Mehrkörperfall ergibt – jeder Agent berechnet einfach die Verknüpfungslösung und führt seinen eigenen Teil dieser Lösung aus. Doch leider ist das „die“ in „die Verknüpfungslösung“ irreführend.

Für unser Doppelteam existiert mehr als eine Verknüpfungslösung:

PLAN 2:
A : [*Gehe*(*A*, *LinkesNetz*), *NoOp*(*A*)]
B : [*Gehe*(*B*, *RechteGrundlinie*), *Schlagen*(*B*, *Ball*)]

Können sich beide Agenten entweder auf Plan 1 oder auf Plan 2 einigen, wird das Ziel erreicht. Wenn aber *A* den Plan 2 und *B* den Plan 1 wählt, schlägt niemand den Ball zurück. Wenn umgekehrt *A* den Plan 1 und *B* den Plan 2 wählt, versuchen beide, den Ball zu treffen. Die Agenten erkennen dies möglicherweise, doch wie können sie sich koordinieren, um sicherzustellen, dass sie sich auf den Plan einigen?

Eine Option besteht darin, sich über eine **Konvention** zu verständigen, bevor sie sich auf gemeinsame Aktionen einlassen. Eine Konvention ist eine beliebige Bedingung für die Auswahl gemeinsamer Pläne. Zum Beispiel würde die Konvention „bleibe auf deiner Seite des Platzes“ den Plan 1 ausschließen, wodurch die Doppelpartner Plan 2 auswählen. Fahrer auf der Straße sehen sich dem Problem gegenüber, nicht mit anderen Fahrzeugen zu kollidieren; dies wird (partiell) gelöst, indem in den meisten Ländern die Konvention „bleibe auf der rechten Seite der Straße“ akzeptiert wird. Die Alternative „bleibe auf der linken Seite“ funktioniert gleichermaßen, solange sich alle Agenten in einer Umgebung darauf einigen. Ähnliche Betrachtungen gelten für die Entwicklung der menschlichen Sprache, wobei es nicht entscheidend ist, welche Sprache der Einzelne spricht, sondern nur die Tatsache wichtig ist, dass alle in einer Gemeinschaft dieselbe Sprache sprechen. Wenn Konventionen weit verbreitet sind, spricht man von **sozialen Gesetzen**.

Mangels einer Konvention können Agenten **kommunizieren**, um gemeinsames Wissen über einen machbaren gemeinsamen Plan zu erlangen. Zum Beispiel könnte ein Tennisspieler ausrufen: „Meiner!“ oder „Deiner!“, um einen bevorzugten gemeinsamen Plan anzuzeigen. Wir beschreiben in *Kapitel 22* entsprechende Kommunikationsmechanismen ausführlicher, wo wir beobachten, dass für die Kommunikation nicht unbedingt ein verbaler Austausch erforderlich ist. Beispielsweise kann ein Spieler dem anderen einen bevorzugten Plan mitteilen, indem er einfach den ersten Teil dieses Planes ausführt. Wenn Agent *A* zum Netz läuft, muss Agent *B* zurück zur Grundlinie, um den Ball zurückzuschlagen, weil Plan 2 der einzige gemeinsame Plan ist, der damit beginnt, dass *A* zum Netz geht. Dieser auch als **Planerkennung** bezeichnete Ansatz zur Koordination funktioniert, wenn eine einzige Aktion (oder eine kurze Aktionsfolge) ausreichend ist, um einen gemeinsamen Plan eindeutig zu erkennen. Kommunikation kann sowohl mit konkurrierenden als auch mit kooperativen Agenten funktionieren.

Konventionen können auch durch evolutionäre Prozesse entstehen. Zum Beispiel sind Samen fressende Ernteameisen soziale Geschöpfe, die sich aus den weniger sozialen Wespen entwickelt haben. Kolonien von Ameisen führen sehr aufwändige gemeinsame Pläne ohne zentralisierte Kontrolle aus – die Königin ist für die Reproduktion zuständig und nicht für die zentralisierte Planung – und mit sehr begrenzten Berechnungs-, Kommunikations- und Merkfähigkeiten in jeder Ameise (Gordon, 2000, 2007). Die Kolonie hat viele Rollen, unter anderem Arbeiter, Kundschafter und Sammler. Jede Ameise entscheidet sich entsprechend den lokalen Bedingungen, die sie beobachtet, für eine Rolle. Zum Beispiel wandern Sammler vom Nest weg, suchen nach einem Samen und wenn sie einen finden, bringen sie ihn sofort zurück. Somit ist die Rate, mit der Sammler zum Nest zurückkehren, ein Indikator für das heute vorhandene Nahrungsangebot. Wenn die Rate hoch ist, geben andere Ameisen ihre aktuelle Rolle auf und übernehmen die Rolle

der Sammler. Die Ameisen scheinen eine Konvention über die Wichtigkeit von Rollen zu haben – Sammeln ist die wichtigste – und Ameisen wechseln leicht in wichtigere Rollen, jedoch nicht in weniger wichtige. Es gibt einen gewissen Lernmechanismus: Eine Kolonie lernt im Verlauf ihres jahrzehntelangen Lebens, erfolgreichere und umsichtigere Aktionen zu unternehmen, selbst wenn einzelne Ameisen nur etwa ein Jahr leben.

Ein letztes Beispiel für kooperatives Multiagenten-Verhalten ist im Schwarmverhalten von Vögeln zu sehen. Eine sinnvolle Simulation für einen Schwarm lässt sich erhalten, wenn jeder Vogelagent (auch als **Boid** bezeichnet) die Positionen seiner nächsten Nachbarn beobachtet und dann den Kurs und die Beschleunigung wählt, um die gewichtete Summe dieser drei Komponenten zu maximieren:

1. Zusammenhalt: ein positiver Punktwert, wenn sich der Agent der mittleren Position der Nachbarn nähert
2. Trennung: ein negativer Punktwert, wenn der Agent irgendeinem Nachbarn zu nahe kommt
3. Ausrichtung: ein positiver Punktwert, wenn der Agent dem durchschnittlichen Kurs der Nachbarn näher kommt

Wenn alle Boids dieses Regelwerk einhalten, weist der Schwarm das **emergente Verhalten** auf, als Pseudofestkörper mit ungefähr konstanter Dichte zu fliegen, der sich über die Zeit nicht zerstreut und der gelegentlich plötzliche Schwenkbewegungen durchführt. ▶ Abbildung 11.11 zeigt in (a) eine Momentaufnahme der Simulation im Vergleich zu einem realen Vogelschwarm in (b). Wie bei Ameisen ist es nicht erforderlich, dass jeder Agent einen Verknüpfungsplan besitzt, der die Aktionen der anderen Agenten modelliert.

Bei den schwierigsten Multiagenten-Problemen ist sowohl die Kooperation mit Mitgliedern des eigenen Teams als auch der Wettbewerb mit Mitgliedern des gegnerischen Teams zu berücksichtigen und zwar alles ohne zentralisierte Steuerung. Wir sehen dies in Spielen wie zum Beispiel Roboterfußball oder dem in Abbildung 11.11(c) gezeigten Spiel NERO, in dem zwei Teams von Softwareagenten darum kämpfen, die Kontrolltürme zu erobern. Bis jetzt sind Methoden für effiziente Planung in derartigen Umgebungen – zum Beispiel das Profitieren der losen Kopplung – erst in Ansätzen realisiert.

Abbildung 11.11: (a) Mithilfe des Boids-Modells von Reynold simulierter Vogelschwarm. (Bild mit freundlicher Genehmigung von Giuseppe Randazzo, novastructura.net.) (b) Ein realer Starenschwarm. (Bild von Eduardo, pastaboy sleeps auf flickr). (c) Zwei konkurrierende Teams von Agenten, die versuchen, die Türme im Spiel NERO zu erobern. (Bild mit freundlicher Genehmigung von Risto Miikkulainen)

Bibliografische und historische Hinweise

Planen mit Zeitbeschränkungen wurde als Erstes durch den DEVISER realisiert (Vere, 1983). Die Darstellung von Zeit in Plänen wurde von Allen (1984) sowie von Dean et al. (1990) im FORBIN-System aufgegriffen. NONLIN+ (Tate und Whiter, 1984) und SIPE (Wilkins, 1988, 1990) konnten Schlussfolgerungen zur Zuordnung begrenzter Ressourcen für verschiedene Planschritte ziehen. O-PLAN (Bell und Tate, 1985), ein HTN-Planer, verwendete eine einheitliche, allgemeine Repräsentation für Bedingungen im Hinblick auf Zeit und Ressourcen. Neben der im Text bereits erwähnten Hitachi-Anwendung wurde O-PLAN für die Planung der Software-Beschaffung bei Price Waterhouse sowie die Planung für die Hinterachsmontage bei Jaguar eingesetzt.

Die beiden Planer SAPA (Do und Kambhampati, 2001) und T4 (Haslum und Geffner, 2101) verwenden beide eine Vorwärts-Zustandsraumsuche mit komplexen Heuristiken, um Aktionen mit Zeitdauer und Ressourcenaufwand zu verarbeiten. Eine Alternative ist die Verwendung sehr ausdrucksstarker Aktionssprachen, die jedoch durch von Menschen geschriebene domänenspezifische Heuristiken angeleitet werden, wie etwa ASPEN (Fukunaga et al., 1997), HSTS (Jonsson et al., 1000) und IxTeT (Ghallab und Laruelle, 1994).

Es wurden mehrere hybride Planungs-/Ablaufplanungssysteme realisiert: ISIS (Fox et al., 1982; Fox, 1990) wurde für das Job Shop Scheduling bei Westinghouse eingesetzt, GARI (Descotte und Latombe, 1985) planten die Fertigung und Konstruktion mechanischer Teile, FORBIN wurde für die Fertigungskontrolle genutzt und NONLIN+ diente der Planung in der Flottenlogistik. Planung und Zeitplanung stellen wir hier als zwei separate Probleme vor. Cushing et al. (2007) zeigen, dass dies bei bestimmten Problemen zu Unvollständigkeit führen kann. Die Ablaufplanung hat in der Raumfahrt eine lange Geschichte. T-SCHED (Drabble, 1990) wurde verwendet, um den zeitlichen Ablauf von Missionsbefehlsfolgen für den Satelliten UOSAT-II zu planen. OPTIMUM-AIV (Aarup et al., 1994) und PLAN-ERS1 (Fuchs et al., 1990) basieren beide auf O-PLAN und wurden von der ESA (European Space Agency) für die Raketenmontage bzw. die Beobachtungsplanung verwendet. SPIKE (Johnston und Adorf, 1992) wurde von der NASA für die Beobachtungsplanung des Weltraumteleskops Hubble verwendet, während das Space Shuttle Ground Processing Scheduling System (Deale et al., 1994) das Job Shop Scheduling für bis zu 16.000 Arbeitsschichten ausführt. Remote Agent (Muscettola et al., 1998) wurde zum ersten autonomen Planer/Zeitplaner für die Steuerung eines Raumschiffes, als er 1999 an Bord der Sonde Deep Space One flog. Raumfahrtanwendungen haben die Entwicklung von Algorithmen für Ressourcenzuordnungen vorangetrieben; siehe Laborie (2003) und Muscettola (2002). Die Literatur zur Zeitplanung wird in einem klassischen Überblicksartikel (Lawler et al., 1993), einem neueren Buch (Pinedo, 2008) und einem überarbeiteten Handbuch (Blazewicz et al., 2007) dargestellt.

Die Möglichkeit lernender **Macrops** („Macro-Operatoren“, die aus einer Folge primitiver Schritte bestehen) im STRIPS-Programm könnte als erster Mechanismus für hierarchisches Planen betrachtet werden (Fikes et al., 1972). Eine Hierarchie wurde auch im System LAWALY (Siklossy und Dreussi, 1973) verwendet. Das System ABSTRIPS (Sacerdoti, 1974) führte die Idee einer **Abstraktionshierarchie** ein, wobei Planen auf höheren Ebenen erlaubt war, Vorbedingungen für Aktionen auf niedrigeren Ebenen zu ignorieren, um die allgemeine Struktur eines funktionierenden Plans abzuleiten. Austin Tates Doktorarbeit (1975b) und die Arbeit von Earl Sacerdoti (1977) entwickelten die grundlegenden Konzepte des HTN-Planens in ihrer modernen Form. Viele prakti-

sche Planer, einschließlich O-PLAN und SIPE, sind HTN-Planer. Yang (1990) diskutiert Eigenschaften von Aktionen, die HTN-Planen effizient machen. Erol, Hendler und Nau (1994, 1996) stellen einen vollständigen hierarchischen Zerlegungsplaner sowie eine Vielzahl von Komplexitätsergebnissen für reine HTN-Planer vor. Unsere Darstellung von HLAs und optimistischer Semantik geht auf Marthi et al. (2007, 2008) zurück. Kambhampati et al., 1998) haben einen Ansatz vorgestellt, in dem Zerlegungen lediglich eine andere Form der Planverfeinerung sind, ähnlich den Verfeinerungen für nichthierarchisches partiell ordnendes Planen.

Beginnend mit der Arbeit zu Macro-Operatoren in STRIPS hat man mit hierarchischer Planung unter anderem das Ziel verfolgt, vorherige Planungserfahrung in Form verallgemeinerter Pläne wiederzuverwenden. Die Technik des **erklärungsbasierten Lernens**, die *Kapitel 19* ausführlich beschreibt, wurde in mehreren Systemen als Mittel angewendet, vorher berechnete Pläne zu verallgemeinern. Hierzu gehören SOAR (Laird et al., 1986) und PRODIGY (Carbonell et al., 1989). Ein alternativer Ansatz speichert vorher berechnete Pläne in deren ursprünglicher Form und verwendet sie dann wieder, um neue, ähnliche Probleme durch Analogie zum ursprünglichen Problem zu lösen. Dieses Konzept wird in der sogenannten **fallbasierten Planung** eingesetzt (Carbonell, 1983; Alterman, 1988; Hammond, 1989). Kambhampati (1994) führt an, dass fallbasierte Planung als Form der Verfeinerungsplanung analysiert werden sollte und eine formale Grundlage für fallbasiertes partiell ordnendes Planen bietet.

Frühen Planern fehlten Bedingungen und Schleifen, doch waren einige in der Lage, mithilfe von Zwang konformante Pläne zu bilden. NOAH von Sacerdoti verwendete dieses Instrument in seiner Lösung des „Schlüssel und Kästen"-Problems, einem Planungsproblem, in dem der Planer wenig über den Ausgangszustand weiß. Mason (1993) sagte, dass man auf die Sensoreingabe in der Roboterplanung häufig verzichten kann und sollte. Er beschrieb einen sensorlosen Plan, der ein Werkzeug durch eine Sequenz von Kippaktionen an eine bestimmte Position auf einem Tisch bringt – *unabhängig* von der Ausgangsposition.

Goldman und Boddy (1996) haben den Begriff des **konformanten Planens** eingeführt und dabei darauf hingewiesen, dass sensorlose Pläne oftmals effektiv sind, selbst wenn der Agent über Sensoren verfügt. Der erste mittelmäßig effiziente konformante Planer war der Conformant Graphplan oder CGP von Smith und Weld (1998). Ferraris und Giunchiglia (2000) sowie Rintanen (1999) entwickelten unabhängig voneinander auf SATPLAN basierende konformante Planer. Bonet und Geffner (2000) beschreiben einen konformanten Planer, der auf der Heuristiksuche in der Menge der Belief States basiert, wobei sie auf Ideen zurückgreifen, die zuerst in den 1960er Jahren für partiell beobachtbare Markov-Entscheidungsprozesse oder POMDPs entwickelt wurden (siehe *Kapitel 17*).

Derzeit gibt es drei bestimmende Ansätze für die konformante Planung. Die ersten beiden arbeiten mit Heuristiksuche im Belief State: HSCP (Bertoli et al., 2001a) greift auf binäre Entscheidungsdiagramme (BDDs) zurück, um Belief States darzustellen, während Hoffmann und Brafman (2006) den bequemen Ansatz der Berechnung von Vorbedingungs- und Zieltests auf Anforderung mithilfe eines SAT-Solvers aufgreifen. Der dritte Ansatz, der hauptsächlich von Jussi Rintanen (2007) verfochten wird, formuliert das gesamte sensorlose Planungsproblem als quantifizierte boolesche Formel (Quantified Boolean Formula, QBF) und löst sie mithilfe eines Allzweck-QBF-Solvers. Aktuelle konformante Planer sind um fünf Größenordnungen schneller als CGP. Der

Gewinner des Tracks für konformante Planung auf dem Internationalen Planungswettbewerb von 2006 war T_0 (Palacios und Geffner, 2007), der Heuristiksuche im Belief State verwendet und dabei die Belief-State-Darstellung einfach hält, indem er Literale ableitet, die bedingte Effekte abdecken. Bryce und Kambhampati (2007) erörtern, wie ein Planungsgraph verallgemeinert werden kann, um gute Heuristiken für konformante und kontingente Planung zu generieren.

In der Literatur gab es einige Unklarheiten bei der Unterscheidung zwischen den Begriffen „bedingte“ und „kontingente“ Planung. Nach Majercik und Littman (2003) meinen wir mit „bedingt“ einen Plan (oder eine Aktion), der unterschiedliche Effekte abhängig vom tatsächlichen Zustand der Welt hat, und mit „kontingent“ einen Plan, in dem der Agent unterschiedliche Aktionen abhängig von den sensorischen Ergebnissen wählen kann. Das Problem der Kontingenzplanung fand größere Beachtung nach der Veröffentlichung des einflussreichen Artikels *Planning and Acting* von Drew McDermott (1978a).

Der in diesem Kapitel beschriebene Kontingenzplanungsansatz geht auf Hoffmann und Brafman (2005) zurück. Beeinflusst wurde er durch die effizienten Suchalgorithmen für zyklische AND-OR-Graphen, die von Jimenez und Torras (2000) sowie Hansen und Zilberstein (2001) entwickelt wurden. Bertoli et al. (2001b) beschreiben MBP (Model-Based Planner), der binäre Entscheidungsdiagramme verwendet, um konformante und kontingente Planung auszuführen.

In der Rückschau kann man heute erkennen, wie die wichtigen klassischen Planungsalgorithmen zu erweiterten Versionen für unsichere Domänen geführt haben. Die schnelle heuristische Vorwärtssuche durch einen Zustandsraum führte zur Vorwärtssuche im Glaubensraum (Bonet und Geffner, 2000; Hoffmann und Brafman 2005); SATPLAN führte zu stochastischem SATPLAN (Majercik und Littman, 2003) und zum Planen unter Verwendung quantifizierter boolescher Logik (Rintanen, 2007); die partiell ordnende Logik führte zu UWL (Etzioni et al., 1992) und CNLP (Peot und Smith, 1992); GRAPHPLAN führte zu Sensory Graphplan oder SGP (Weld et al., 1998).

Der erste Online-Planer mit Ausführungsüberwachung war PLANEX (Fikes et al., 1972), der mit dem STRIPS-Planer arbeitete, um den Roboter Shakey zu steuern. Der NASL-Planer (McDermott, 1978a) behandelte ein Planungsproblem einfach als Spezifikation für die Ausführung einer komplexen Aktion, sodass Ausführung und Planen vollständig vereinigt wurden. SIPE (System for Interactive Planning and Execution monitoring) (Wilkins, 1988, 1990) war der erste Planer, der sich systematisch mit dem Problem des Neuplanens beschäftigte. Er wurde in Demonstrationsprojekten mehrerer Domänen verwendet, wie etwa für Planungsoperationen auf dem Flugdeck von Flugzeugträgern, für das Job Shop Scheduling einer australischen Brauerei und die Planung von Hochhauskonstruktionen (Kartam und Levitt, 1990).

Mitte der 1980er Jahre führte der Pessimismus bezüglich der schlechten Laufzeiten zum Vorschlag von Reflexagenten, den sogenannten **reaktiven Planungs**systemen (Brooks, 1986; Agre und Chapman, 1987). PENGI (Agre und Chapman, 1987) war in der Lage, ein (vollständig beobachtbares) Videospiel zu spielen – unter Verwendung boolescher Schaltungen in Kombination mit einer „visuellen“ Repräsentation aktueller Ziele und des internen Zustandes des Agenten. „Universal plans“ (Schoppers, 1987, 1989) wurden als Nachschlagetabellen-Methode für reaktives Planen entwickelt, aber es stellte sich heraus, dass es sich dabei um eine Neuentdeckung der Idee der **Strategien** handelte, die in den Entscheidungsprozessen von Markov schon längst

verwendet worden waren (siehe *Kapitel 17*). Ein universeller Plan (oder eine Strategie) enthält eine Abbildung eines beliebigen Zustandes auf die Aktion, die in diesem Zustand ausgeführt werden sollte. Koenig (2001) gibt einen Überblick über Online-Planungstechniken unter dem Namen *Agent-Centered Search*.

Das Multiagenten-Planen hat erst in den letzten Jahren an Popularität gewonnen, obwohl es auf eine lange Geschichte zurückblickt. Konolige (1982) zeigte eine Formalisierung des Multiagenten-Planens in Logik erster Stufe auf, während Pednault (1986) eine Beschreibung im STRIPS-Stil lieferte. Das Konzept der gemeinsamen Absicht, das wesentlich ist, wenn Agenten einen gemeinsamen Plan ausführen sollen, stammt aus Arbeiten zum kommunikativen Handeln (Cohen und Levesque, 1990; Cohen et al., 1990). Boutilier und Brafman (2001) zeigen, wie sich partiell ordnendes Planen auf eine Multiaktor-Einrichtung anpassen lässt. Brafman und Domshlak (2008) formulieren einen Multiaktor-Planungsalgorithmus, dessen Komplexität nur linear mit der Anzahl der Aktoren wächst, vorausgesetzt, dass der Kopplungsgrad (zum Teil durch die **Baumbreite** des Graphen von Interaktionen zwischen Agenten gemessen) begrenzt ist. Petrik und Zilberstein (2009) zeigen, dass ein auf bilinearer Programmierung basierender Ansatz dem in *Kapitel 10* skizzierten Mengenüberdeckungsproblem[7] überlegen ist.

Wir haben kaum die Oberfläche der Arbeiten über Einigungen im Multiagenten-Planen berührt. Durfee und Lesser (1989) diskutieren, wie Aufgaben auf mehrere Agenten durch Verhandlungen verteilt werden können. Kraus et al. (1991) beschreiben ein System für das Spiel *Diplomacy*, ein Brettspiel, wo Einigung, Koalitionsbildung und -auflösung sowie Unehrlichkeit zum Spielverlauf gehören. Stone (2000) zeigt, wie Agenten als Teammitglieder in der konkurrierenden, dynamischen, partiell beobachtbaren Umgebung von Roboterfußball kooperieren können. In einem späteren Artikel analysiert Stone (2003) zwei kompetitive Multiagenten-Umgebungen – RoboCup, ein Wettbewerb für Roboterfußball, und TAC, der auktionsbasierte Trading Agents Competition – und stellt fest, dass die rechnerische Widerspenstigkeit unserer aktuellen theoretischen fundierten Lösungsansätze zu vielen Multiagenten-Systemen geführt hat, die durch Ad-hoc-Methoden entworfen wurden.

In seiner höchst einflussreichen Theorie *Society of Mind* stellt Marvin Minsky (1986, 2007) die These auf, dass menschliche Gehirne aus einem Ensemble von Agenten konstruiert sind. Livnat und Pippenger (2006) beweisen, dass für das Problem der optimalen Pfadermittlung und einer gegebenen Beschränkung des Gesamtumfanges von rechentechnischen Ressourcen die beste Architektur für einen Agenten ein Ensemble von Subagenten ist, von denen jeder versucht, sein eigenes Ziel zu optimieren, und von denen alle in Konflikt mit einem anderen stehen.

Das Vogelschwarmmodell (Boid-Modell) in *Abschnitt 11.4.2* stammt von Reynolds (1987), der einen Oskar für seine Anwendung auf Fledermaus- und Pinguinschwärme im Film *Batman Returns* erhielt. Das Spiel NERO und die Methoden für Lernstrategien werden von Bryant und Miikkulainen (2007) beschrieben.

Zu den neueren Büchern über Multiagenten-Systeme gehören die von Weiss (2000a), Young (2004), Vlassis (2008) sowie Shoham und Leyton-Brown (2009). AAMAS ist eine jährliche Konferenz zu autonomen Agenten und Multiagenten-Systemen.

7 Mengenüberdeckungsproblem = set cover problem

Zusammenfassung

Dieses Kapitel hat sich mit einigen der Komplikationen beim Planen und Handeln in der realen Welt beschäftigt. Die wichtigsten Aspekte sind:

- Viele Aktionen verbrauchen **Ressourcen**, wie beispielsweise Geld, Benzin oder Rohstoffe. Es ist praktisch, diese Ressourcen als numerische Mengen in einem Pool zu behandeln, anstatt beispielsweise zu versuchen, Schlüsse über jede einzelne Münze und jeden einzelnen Geldschein auf der Welt zu ziehen. Aktionen können Ressourcen erzeugen und verbrauchen und normalerweise ist es billig und effektiv, partielle Pläne auf die Erfüllung von Ressourcenbedingungen zu überprüfen, bevor man weitere Verfeinerungen in Betracht zieht.
- Zeit ist eine der wichtigsten Ressourcen. Sie kann von speziellen Zeitplanungsalgorithmen verarbeitet werden, Zeitplanen kann aber auch in das Planen integriert werden.
- HTN-Planen (**Hierarchisches Task-Netzwerk**) erlaubt es dem Agenten, vom Domänenentwickler Rat in Form von **höheren Aktionen** (High-Level Actions, HLAs) entgegenzunehmen. Die Effekte von HLAs lassen sich mit **optimistischer Semantik** definieren, wodurch nachweisbar korrekte höhere Pläne abgeleitet werden können, ohne Implementierungen auf niedrigerer Ebene betrachten zu müssen. HTN-Methoden sind in der Lage, die sehr großen Pläne zu erzeugen, die in vielen Anwendungen der realen Welt erforderlich sind.
- Standardplanungsalgorithmen setzen vollständige und korrekte Informationen sowie deterministische, vollständig beobachtbare Umgebungen voraus. Viele Domänen verletzen diese Annahme.
- **Kontingenzpläne** erlauben es dem Agenten, die Welt während der Ausführung abzutasten, um zu entscheiden, welcher Zweig des Planes zu verfolgen ist. In manchen Fällen lässt sich mit **sensorloser** oder **konformanter Planung** ein Plan konstruieren, der funktioniert, ohne dass Wahrnehmung notwendig ist. Sowohl konformante als auch Kontingenzpläne können durch eine Suche im **Belief State** konstruiert werden. Effiziente Darstellung und Berechnung von Belief States ist ein Schlüsselproblem.
- Ein **Online-Planungsagent** verwendet Ausführungsüberwachung und verzweigt bei Bedarf in Reparaturen, um aus unerwarteten Situationen zurückzukehren, die aufgrund von nichtdeterministischen Aktionen, exogenen Ereignissen oder inkorrekten Modellen der Umgebung entstanden sein können.
- **Multiagenten-Planen** ist erforderlich, wenn es andere Agenten in der Umgebung gibt, mit denen kooperiert oder konkurriert werden soll. Gemeinsame Pläne lassen sich konstruieren, müssen aber um eine Art der Koordination ergänzt werden, wenn sich zwei Agenten einigen sollen, welcher gemeinsame Plan auszuführen ist.
- Dieses Kapitel erweitert die klassische Planung, um nichtdeterministische Umgebungen abzudecken (wo die Ergebnisse von Aktionen unbestimmt sind), doch ist das letzte Wort zur Planung noch nicht gesprochen. *Kapitel 17* beschreibt Techniken für stochastische Umgebungen (in denen die Ergebnisse von Aktionen zugeordnete Wahrscheinlichkeiten haben): Markov-Entscheidungsprozesse, partiell beobachtbare Markov-Entscheidungsprozesse und Spieltheorie. In *Kapitel 21* zeigen wir, dass ein Agent durch verstärkendes Lernen in der Lage ist, aus vergangenen Erfolgen und Fehlern zu lernen, wie er sich verhalten soll.

Übungen zu Kapitel 11

Lösungshinweise

1 Mit einer bestimmten Anzahl von LKW ist eine Menge von Paketen auszuliefern. Der Weg jedes Pakets beginnt an einem bestimmten Ort auf einer Rasterkarte und endet an einem anderen Ort. Jeder LKW wird direkt gesteuert, indem er vorwärts bewegt und gedreht wird. Konstruieren Sie für dieses Problem eine Hierarchie von höheren Aktionen. Welche Kenntnisse über die Lösung kodiert Ihre Hierarchie?

2 Eine High-Level-Aktion habe genau eine Implementierung als Sequenz von primitiven Aktionen. Geben Sie einen Algorithmus an, der die Vorbedingungen und Effekte berechnet, wobei die vollständige Verfeinerungshierarchie und Schemas für die primitiven Aktionen gegeben sind.

3 Nehmen Sie an, dass die optimistisch erreichbare Menge eines höheren Planes eine Obermenge der Zielmenge ist. Lässt sich irgendwie daraus folgern, ob der Plan das Ziel erreicht? Wie sieht es aus, wenn die pessimistisch erreichbare Menge keine Schnittmenge mit der Zielmenge ist? Erläutern Sie Ihre Antwort.

4 Schreiben Sie einen Algorithmus, der einen Ausgangszustand (durch eine Menge von aussagenlogischen Literalen spezifiziert) und eine Sequenz von HLAs (die jeweils durch Vorbedingungen und optimistische Spezifikationen von optimistisch und pessimistisch erreichbaren Mengen definiert sind) übernimmt und optimistische und pessimistische Beschreibungen der erreichten Menge der Sequenz berechnet.

5 Abbildung 11.1 zeigt, wie Aktionen in einem Zeitplanungsproblem mithilfe separater Felder für DAUER, VERWENDEN und VERBRAUCHEN zu beschreiben sind. Nehmen Sie nun an, dass der Zeitplan mit nichtdeterministischer Planung zu kombinieren ist. Dies erfordert nichtdeterministische und bedingte Effekte. Betrachten Sie jedes der drei Felder und erläutern Sie, ob sie separate Felder bleiben oder zu Effekten der Aktion werden sollen. Geben Sie für jedes der drei Felder ein Beispiel an.

6 Einige der Operationen in Standardprogrammiersprachen können als Aktionen modelliert werden, die den Zustand der Welt ändern. Beispielsweise ändert die Zuweisungsoperation den Inhalt einer Speicherstelle, während eine Druckoperation den Zustand des Ausgabestromes ändert. Ein Programm, das aus diesen Operationen besteht, kann auch als Plan betrachtet werden, dessen Ziel durch die Spezifikation des Programms angegeben ist. Aus diesem Grund können Planungsalgorithmen verwendet werden, um Programme zu konstruieren, die eine bestimmte Spezifikation erreichen.

a. Schreiben Sie ein Aktionsschema für den Zuweisungsoperator (der den Wert der einen Variablen an eine andere Variable zuweist). Beachten Sie, dass der ursprüngliche Wert überschrieben wird!

b. Zeigen Sie, wie die Objekterstellung von einem Planer genutzt werden kann, um einen Plan für den Austausch von zwei Variablenwerten unter Verwendung einer temporären Variablen zu erstellen.

7 Betrachten Sie das folgende Argument: In einem Framework, das unsichere Ausgangszustände unterstützt, sind **nichtdeterministische Effekte** lediglich eine notationsbedingte Bequemlichkeit, keine Quelle zusätzlicher Ausdrucksstärke. Für ein Aktionsschema a mit dem nichtdeterministischen Effekt $P \vee Q$ können wir den Effekt immer durch die bedingten Effekte **when** R: $P \wedge$ **when** $\neg R$: Q ersetzen, die sich wiederum zu zwei regulären Aktionen reduzieren lassen. Die Aussage R steht für eine beliebige Aussage, die im Ausgangszustand unbekannt ist und für die es keine Sensoreingabeaktionen gibt. Ist dieses Argument korrekt? Betrachten Sie separat zwei Fälle – einen, bei dem nur eine Instanz des Aktionsschemas a im Plan vorkommt, den anderen, wo es mehrere Instanzen von a gibt.

8 Nehmen Sie an, dass die *Flip*-Aktion immer den Wahrheitswert der Variablen L ändert. Zeigen Sie, wie ihre Effekte zu definieren sind, indem Sie ein Aktionsschema mit bedingten Effekten verwenden. Zeigen Sie, dass trotz der Verwendung von bedingten Effekten eine 1-KNF-Belief-State-Darstellung nach einem *Flip* in 1-KNF verbleibt.

9 In der Blockwelt waren wir gezwungen, zwei Aktionsschemas – *Bewegen* und *AufTischStellen* – einzuführen, um das Prädikat *Frei* korrekt verwalten zu können. Zeigen Sie, wie bedingte Effekte verwendet werden können, um beide Fälle mit einer einzigen Aktion zu repräsentieren.

10 Bedingte Effekte wurden für die Aktion *Saugen* in der Staubsauerwelt demonstriert – welches Quadrat sauber wird, ist davon abhängig, auf welchem Quadrat sich der Roboter befindet. Können Sie sich eine neue Menge aussagenlogischer Variablen vorstellen, um die Zustände der Staubsaugerwelt zu definieren, sodass *Saugen* eine *unbedingte* Beschreibung erhält? Fertigen Sie Beschreibungen von *Saugen*, *Links* und *Rechts* unter Verwendung Ihrer Aussagen an und zeigen Sie, dass sie hinreichend sind, um alle möglichen Zustände der Welt zu beschreiben.

11 Die folgenden Zitate stammen von der Aufschrift von Shampoo-Flaschen. Identifizieren Sie sie als unbedingten, bedingten oder ausführungsüberwachten Plan. (a) „Schäumen. Spülen. Wiederholen." (b) „Massieren Sie das Shampoo in das Haar ein und lassen Sie es mehrere Minuten einwirken. Spülen Sie und wiederholen Sie die Prozedur gegebenenfalls." (c) „Wenden Sie sich an einen Arzt, wenn Probleme auftreten."

12 Betrachten Sie das folgende Problem: Ein Patient erscheint in der Arztpraxis und zeigt Symptome, die entweder durch eine Dehydrierung oder durch die Krankheit D entstanden sein können (aber nicht durch beides). Es gibt zwei mögliche Aktionen: *Trinken*, wodurch die Dehydrierung unbedingt behoben wird, und *Behandeln*, wodurch die Krankheit D geheilt wird, die jedoch eine unerwünschte Nebenwirkung hat, falls der Patient dehydriert ist. Fertigen Sie die Problembeschreibung an und skizzieren Sie einen sensorlosen Plan, der das Problem löst. Nennen Sie dabei alle relevanten möglichen Welten.

Wissensrepräsentation

12

ÜBERBLICK

In diesem Kapitel zeigen wir, wie man die Logik erster Stufe nutzt, um die wichtigsten Aspekte der realen Welt zu repräsentieren, wie etwa Aktionen, Raum, Zeit, mentale Ereignisse und Einkaufen.

Die vorherigen Kapitel haben die Technologie für wissensbasierte Agenten beschrieben: Syntax, Semantik und Beweistheorie der Aussagenlogik sowie der Logik erster Stufe ebenso wie die Implementierung von Agenten, die diese Logiken verwenden. In diesem Kapitel beschäftigen wir uns mit der Frage, welchen *Inhalt* man in der Wissensbasis eines Agenten ablegen sollte – wie man Fakten über die Welt repräsentiert.

Abschnitt 12.1 führt das Konzept einer allgemeinen Ontologie ein, die alles in der Welt in einer Hierarchie von Kategorien organisiert. *Abschnitt 12.2* behandelt die grundlegenden Kategorien von Objekten, Substanzen und Maßen, *Abschnitt 12.3* geht auf Ereignisse ein und in *Abschnitt 12.4* geht es um das Wissen über Glauben. Dann kehren wir zur Technologie für das Schließen mit folgendem Inhalt zurück: *Abschnitt 12.5* erörtert Schlusssysteme, die für effiziente Inferenz mit Kategorien konzipiert sind, *Abschnitt 12.6* diskutiert Schließen mit Standardinformationen. *Abschnitt 12.7* bringt das gesamte Wissen im Kontext einer Internet-Shopping-Umgebung zusammen.

12.1 Ontologisches Engineering

In „Spielzeug"-Domänen ist die Auswahl der Repräsentation nicht so wichtig; viele Optionen funktionieren. Für komplexere Domänen dagegen, wie beispielsweise das Einkaufen im Internet oder das Steuern eines Autos durch den Verkehr, braucht man allgemeinere und flexiblere Repräsentationen. Dieses Kapitel zeigt, wie man diese Repräsentationen erzeugt. Dabei konzentrieren wir uns auf allgemeine Konzepte – z.B. *Ereignisse*, *Zeit*, *physische Objekte* und *Glauben* –, die in vielen unterschiedlichen Domänen auftreten können. Die Repräsentation dieser abstrakten Konzepte wird manchmal auch als **ontologisches Engineering** bezeichnet.

Die Aussicht, *alles* in der Welt repräsentieren zu müssen, ist erschreckend. Natürlich geben wir keine vollständige Beschreibung von allem – das wäre viel zu viel selbst für ein Buch mit 1000 Seiten –, aber wir lassen Platzhalter, wo neues Wissen für andere Domänen eingefügt werden kann. Beispielsweise definieren wir, was es bedeutet, ein physisches Objekt zu sein, und die Details verschiedener Objekttypen – Roboter, Fernseher, Bücher oder was auch immer – können später eingetragen werden. Das entspricht der Art und Weise, wie Designer eines objektorientierten Programmierframeworks (wie zum Beispiel dem grafischen Java-Framework Swing) Konzepte wie *Window* definieren und davon ausgehen, dass Benutzer diese Elemente verwenden, um spezifischere Konzepte wie *SpreadsheetWindow* zu definieren. Das allgemeine Gerüst (Framework) für die Konzepte wird auch als **obere Ontologie** bezeichnet – aufgrund der Konvention, Graphen mit den allgemeineren Konzepten ganz oben und den spezifischeren Konzepten darunter liegend zu zeichnen, wie in ▶ Abbildung 12.1 gezeigt.

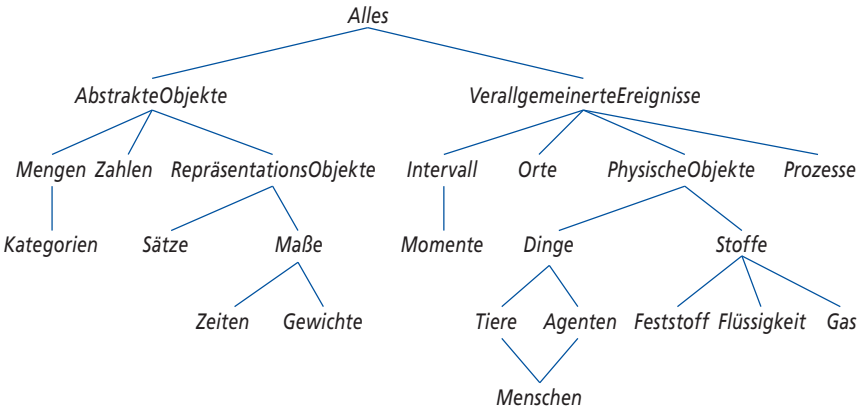


Abbildung 12.1: Die obere Ontologie der Welt, die die Themen aufzeigt, die später im Kapitel behandelt werden. Jede Kante weist darauf hin, dass das darunterliegende Konzept eine Spezialisierung des darüberliegenden Konzeptes ist.

Bevor wir die Ontologie weiter betrachten, sollten wir eine wichtige Warnung aussprechen. Wir haben die Logik erster Stufe gewählt, um Inhalt und Aufbau von Wissen zu beschreiben, obwohl bestimmte Aspekte der realen Welt in Logik erster Stufe schwer auszudrücken sind. Die größte Schwierigkeit ist, dass es für fast alle Verallgemeinerungen Ausnahmen gibt oder dass sie nur bis zu einem gewissen Grad gelten. Obwohl „Tomaten sind rot“ eine sinnvolle Regel ist, sind einige Tomaten grün, gelb oder orange. Ähnliche Ausnahmen findet man zu fast allen allgemeinen Aussagen in diesem Kapitel. Die Fähigkeit, Ausnahmen und Unsicherheit zu verarbeiten, ist extrem wichtig, aber sie ist der Aufgabe nicht zuträglich, die allgemeine Ontologie zu verstehen. Aus diesem Grund verschieben wir die Diskussion der Ausnahmen auf *Abschnitt 12.5* und das allgemeinere Thema zum Schließen mit unsicherer Information auf *Kapitel 13*.

Welchen Nutzen hat eine obere Ontologie? Betrachten Sie noch einmal die Ontologie für Schaltkreise in *Abschnitt 8.4.2*. Sie trifft viele vereinfachende Annahmen: So wird die Zeit überhaupt nicht berücksichtigt, Signale sind feststehend und werden nicht weitergeleitet, die Struktur des Schaltkreises bleibt konstant. Eine allgemeinere Ontologie würde Signale zu bestimmten Zeiten betrachten sowie die Drahtlängen und die Signallaufzeiten berücksichtigen. Damit könnten wir die Timing-Eigenschaften des Schaltkreises simulieren und die Schaltkreisentwickler führen derartige Simulationen in der Tat oftmals durch. Außerdem könnten wir interessantere Gatterklassen einführen, indem wir beispielsweise die Technologie (TTL, CMOS usw.) sowie die Eingabe/Ausgabe-Spezifikation beschreiben. Wenn wir Zuverlässigkeit oder Diagnose diskutieren wollten, würden wir die Möglichkeit berücksichtigen, dass sich die Struktur des Schaltkreises oder die Eigenschaften von Gattern spontan ändern können. Um Streukapazitäten zu berücksichtigen, müssten wir darstellen, wo sich die Leitungen auf der Leiterkarte befinden.

Für die Wumpus-Welt gelten ähnliche Betrachtungen. Wir berücksichtigen dabei zwar die Zeit, aber sie hat eine sehr einfache Struktur: Es passiert nichts, außer wenn der Agent handelt, und alle Änderungen treten unmittelbar auf. Eine allgemeinere Ontologie, die besser für die reale Welt geeignet ist, würde es erlauben, dass sich gleichzeitige Änderungen über eine bestimmte Zeit erstrecken. Wir könnten auch das Prädikat *Pit* verwenden, um auszudrücken, auf welchen Feldern es Falltüren gibt. Wir könnten unterschiedliche Arten von Falltüren unterstützen, indem wir mehrere Individuen einführen, die zur Klasse der Falltüren gehören und jeweils unterschiedliche Eigen-

schaften aufweisen. Analog dazu könnten wir neben den Wumpi noch andere Tiere zulassen. Vielleicht ist es aus den verfügbaren Wahrnehmungen nicht möglich, die genaue Spezies festzulegen; deshalb müssten wir vielleicht eine biologische Klassifizierung der Wumpus-Welt einrichten, um dem Agenten dabei zu helfen, das Verhalten von Höhlenbewohnern aus spärlichen Hinweisen vorherzusagen.

Für jede spezielle Ontologie ist es möglich, Änderungen wie diese vorzunehmen, um dem Allgemeinfall noch näher zu kommen. Dabei stellt sich offensichtlich die Frage: Treffen alle diese Ontologien in einer allgemeinen Ontologie zusammen? Nach jahrhundertelanger philosophischer und rechentechnischer Forschung lautet die Antwort: „Vielleicht." In diesem Abschnitt stellen wir eine universelle Ontologie vor, die die Ideen aus diesen Jahrhunderten in sich vereint. Es gibt zwei wichtige Charakteristika universeller Ontologien, die sie von Sammlungen spezieller Ontologien unterscheiden:

- Eine universelle Ontologie sollte in fast jeder speziellen Domäne angewendet werden können (mit der Ergänzung domänenspezifischer Axiome). Das bedeutet, dass sich kein darstellerischer Aspekt überlisten oder unter den Teppich kehren lässt.
- In jeder hinreichend anspruchsvollen Domäne müssen verschiedene Wissensbereiche *vereinheitlicht* werden, weil am Schließen und am Problemlösen mehrere Bereiche gleichzeitig beteiligt sein könnten. Ein Robotersystem zur Reparatur von Schaltkreisen beispielsweise muss Schlüsse über Schaltungen in Form der elektrischen Verbindungen und des physischen Layouts sowie der Zeit ziehen können – sowohl für die Analyse der Schaltzeiten als auch für die Abschätzung des Arbeitsaufwandes. Die Sätze, die die Zeit beschreiben, müssen sich demnach mit den Sätzen, die das räumliche Layout beschreiben, kombinieren lassen und gleichermaßen gut für Nanosekunden und Minuten wie auch für Ångström und Meter funktionieren.

Wir sollten vorausschicken, dass das Vorhaben einer allgemeinen ontologischen Technik bislang nur beschränkten Erfolg hatte. Keine der KI-Spitzenanwendungen (wie sie *Kapitel 1* aufgelistet hat) macht Gebrauch von einer gemeinsamen Ontologie – alle verarbeiten Spezialwissen. Soziale und politische Betrachtungen können es erschweren, dass sich konkurrierende Parteien auf eine Ontologie einigen. Wie Tom Gruber (2004) sagt: „Jede Ontologie ist ein Abkommen – eine soziale Übereinkunft – zwischen Leuten mit einem gemeinsamen Motiv an einem Austausch." Wenn konkurrierende Absichten die Motivation für eine gemeinsame Nutzung überwiegen, kann es keine gemeinsame Ontologie geben. Die bereits existierenden Ontologien wurden auf vier Wegen erstellt:

1. Durch ein Team von geschulten Ontologen/Logikern, die die Ontologie aufbauen und Axiome schreiben. Das CYC-System wurde hauptsächlich auf diese Weise erstellt (Lenat und Guha, 1990).

2. Durch Importieren von Kategorien, Attributen und Werten aus einer oder mehreren vorhandenen Datenbanken. DBPEDIA wurde erstellt, indem strukturierte Fakten aus Wikipedia importiert wurden (Bizer et al., 2007).

3. Indem Textdokumente geparst und Informationen aus ihnen extrahiert werden. Um TEXTRUNNER zu erstellen, wurde ein umfangreiches Paket von Webseiten gelesen (Banko und Etzioni, 2008).

4. Durch Werben ungelernter Amateure, um Allgemeinwissen einzugeben. Das System OPENMIND wurde von Freiwilligen erstellt, die Fakten in Englisch vorschlugen (Singh et al., 2002; Chklovski und Gil, 2005).

12.2 Kategorien und Objekte

Tipp

Die Unterteilung von Objekten in **Kategorien** ist ein lebenswichtiger Bestandteil der Wissensrepräsentation. Obwohl die Interaktion mit der Welt auf der Ebene einzelner Objekte stattfindet, *wird häufig auf der Ebene von Kategorien geschlossen*. Ein Einkäufer könnte beispielsweise das Ziel haben, einen Basketball zu kaufen und nicht einen *bestimmten* Basketball, wie etwa den BB_9. Kategorien dienen außerdem dazu, Vorhersagen über Objekte zu treffen, nachdem sie klassifiziert wurden. Man leitet das Vorhandensein bestimmter Objekte aus wahrgenommenen Eingaben ab und man leitet die Mitgliedschaft in einer Kategorie von den wahrgenommenen Eigenschaften der Objekte ab und verwendet dann die Kategorieinformation, um Vorhersagen über die Objekte zu treffen. Zum Beispiel kann man aus ihrer grün-gelb gesprenkelten Haut, einer Größe von 30 cm, einer ovalen Gestalt, rotem Fleisch und dem Vorhandensein an der Fruchttheke schließen, dass ein Objekt eine Wassermelone ist; daraus wiederum kann man ableiten, dass sie für einen Obstsalat geeignet ist.

Es gibt zwei Möglichkeiten, Kategorien in Logik erster Stufe zu repräsentieren: Prädikate und Objekte. Wir können also das Prädikat *Basketball*(*b*) verwenden oder die Kategorie als ein Objekt **reifizieren**[1], *Basketbälle*. Dann könnten wir sagen *Element*(*b*, *Basketbälle*) (was wir als $b \in$ *Basketbälle* abkürzen), um damit auszudrücken, dass *b* ein Element der Kategorie der *Basketbälle* ist. Wir sagen *Untermenge*(*Basketbälle*, *Bälle*) (abgekürzt als *Basketbälle* $\subset$ *Bälle*), um damit auszudrücken, dass *Basketbälle* eine **Unterkategorie** oder Untermenge von *Bälle* ist. Die Begriffe Unterkategorie, Unterklasse und Untermenge verwenden wir hier gleichberechtigt.

Kategorien dienen dazu, die Wissensbasis durch **Vererbung** zu organisieren und zu vereinfachen. Wenn wir sagen, dass alle Instanzen der Kategorie *Nahrungsmittel* essbar sind, und behaupten, dass *Obst* eine Unterklasse von *Nahrungsmittel* ist und *Äpfel* eine Unterklasse von *Obst*, können wir herleiten, dass jeder Apfel essbar ist. Wir sagen, die einzelnen Äpfel **erben** die Eigenschaft der Essbarkeit, in diesem Fall aus ihrer Mitgliedschaft in der Kategorie *Nahrungsmittel*.

Unterklassenrelationen organisieren die Kategorien in einer **Taxonomie** oder **Taxonomie-Hierarchie**. Taxonomien wurden jahrhundertelang explizit in technischen Bereichen angewendet. Die größte derartige Taxonomie organisiert rund 10 Millionen lebende und ausgestorbene Spezies, viele davon Käfer,[2] in einer einzigen Hierarchie; das Bibliothekswesen hat eine Taxonomie aller Wissensbereiche entwickelt, kodiert als Dewey-Dezimalsystem; und die Steuerbehörden und andere Regierungsbereiche haben umfassende Taxonomien für Berufe und kommerzielle Produkte entwickelt. Taxonomien sind auch ein wichtiger Aspekt des Allgemeinwissens.

Die Logik erster Stufe macht es einfach, Fakten über Kategorien auszusagen, indem sie entweder die Objekte in Beziehung zu Kategorien setzt oder über ihre Elemente quantifiziert: Die folgenden Punkte geben einige Arten von Fakten zusammen mit einem Beispiel an:

1 Das Überführen einer Aussage in ein Objekt bezeichnet man als **Reifikation**. Dieser Begriff stammt ab vom lateinischen Wort *res*, Sache. John McCarthy hat den Begriff „thingification“ vorgeschlagen, der sich jedoch nie durchgesetzt hat (im Deutschen entsprechend „Vergegenständlichung“).

2 Der berühmte Biologe J. B. S. Haldane leitete „Eine übermäßige Vorliebe für Käfer“ seitens des Schöpfers her.

- Ein Objekt ist ein Element einer Kategorie.

  $BB_9 \in Basketbälle$
- Eine Kategorie ist eine Unterklasse einer anderen Kategorie.

  $Basketbälle \subset Bälle$
- Alle Elemente einer Kategorie haben bestimmte Eigenschaften.

  $(x \in Basketbälle) \Rightarrow Rund(x)$
- Elemente einer Kategorie können anhand bestimmter Eigenschaften erkannt werden.

  $Orange(x) \wedge Rund(x) \wedge Durchmesser(x) = 9{,}5'' \wedge x \in Bälle \Rightarrow x \in Basketbälle$
- Eine Kategorie als Ganzes hat bestimmte Eigenschaften.

  $Hunde \in DomestizierteSpezies$

Weil *Hunde* eine Kategorie und ein Element von *DomestizierteSpezies* ist, muss *DomestizierteSpezies* eine Kategorie von Kategorien sein. Natürlich gibt es Ausnahmen für viele der obigen Regeln (durchstochene Basketbälle sind nicht rund); auf diese Ausnahmen kommen wir später zu sprechen.

Obwohl *Unterklasse* und *Element* die wichtigsten Relationen für Kategorien sind, wollen wir auch in der Lage sein, Relationen zwischen Kategorien auszudrücken, die keine Unterklassen voneinander sind. Wenn wir beispielsweise nur sagen, dass *Männliche* und *Weibliche* Unterklassen von *Tiere* sind, dann haben wir damit noch nicht gesagt, dass männlich nicht gleich weiblich sein kann. Wir sagen, zwei oder mehr Kategorien sind **disjunkt**, wenn sie keine gemeinsamen Elemente haben. Und selbst wenn wir wissen, dass *Männliche* und *Weibliche* disjunkt sind, wissen wir nicht, dass ein Tier, das kein männliches ist, ein weibliches sein muss, es sei denn, wir sagen, *Männliche* und *Weibliche* bilden eine **erschöpfende Zerlegung** der Tiere. Eine disjunkte erschöpfende Zerlegung wird auch als **Partition** bezeichnet. Die folgenden Beispiele verdeutlichen diese drei Konzepte:

$Disjunkt(\{Tiere, Gemüse\})$
$ErschöpfendeZerlegung(\{Amerikaner, Kanadier, Mexikaner\}, NordAmerikaner)$
$Partition(\{Männliche, Weibliche\}, Tiere)$.

(Beachten Sie, dass die *ErschöpfendeZerlegung* von *NordAmerikaner* keine *Partition* ist, weil einige Menschen eine doppelte Staatsbürgerschaft haben.) Die drei Prädikate sind wie folgt definiert:

$Disjunkt(s) \Leftrightarrow (\forall c_1, c_2 \quad c_1 \in s \wedge c_2 \in s \wedge c_1 \neq c_2 \Rightarrow Schnittmenge(c_1, c_2) = \{\})$
$ErschöpfendeZerlegung(s, c) \Leftrightarrow (\forall i \quad i \in c \Leftrightarrow \exists c_2 \quad c_2 \in s \wedge i \in c_2)$
$Partition(s, c) \Leftrightarrow Disjunkt(s) \wedge ErschöpfendeZerlegung(s, c)$.

Kategorien können auch *definiert* werden, indem die notwendigen und hinreichenden Bedingungen für die Mitgliedschaft spezifiziert werden. Ein Junggeselle beispielsweise ist ein nicht verheirateter erwachsener Mann:

$x \in Jungesellen \Leftrightarrow NichtVerheiratet(x) \wedge x \in Erwachsene \wedge x \in Männliche$.

Wie wir weiter unten im Kasten „Natürliche Arten“ diskutieren werden, sind strenge logische Definitionen für Kategorien weder immer möglich noch immer nötig.

12.2.1 Physische Zusammensetzung

Die Idee, dass ein Objekt Teil eines anderen sein kann, ist uns vertraut. Die Nase ist Teil des Kopfes, Rumänien ist Teil von Europa und dieses Kapitel ist Teil dieses Buches. Wir verwenden die allgemeine *TeilVon*-Relation, um auszudrücken, dass ein Ding Teil eines anderen ist. Objekte können in *TeilVon*-Hierarchien gruppiert werden, die an die *Untermengen*-Hierarchie erinnern:

$$\begin{aligned}&\mathit{TeilVon}(\mathit{Bukarest}, \mathit{Rumänien})\\ &\mathit{TeilVon}(\mathit{Rumänien}, \mathit{Osteuropa})\\ &\mathit{TeilVon}(\mathit{Osteuropa}, \mathit{Europa})\\ &\mathit{TeilVon}(\mathit{Europa}, \mathit{Erde}).\end{aligned}$$

Die *TeilVon*-Relation ist transitiv und reflexiv, d.h.

$$\begin{aligned}&\mathit{TeilVon}(x, y) \wedge \mathit{TeilVon}(y, z) \Rightarrow \mathit{TeilVon}(x, z)\\ &\mathit{TeilVon}(x, x).\end{aligned}$$

Aus diesem Grund können wir schließen: $\mathit{TeilVon}(\mathit{Bukarest}, \mathit{Erde})$.

Kategorien **zusammengesetzter Objekte** sind häufig durch strukturelle Relationen zwischen Teilen charakterisiert. Beispielsweise hat ein Zweibeiner zwei Beine, die an einem Rumpf befestigt sind:

$$\begin{aligned}\mathit{Zweibeiner}(a) \Rightarrow \exists l_1, l_2, b \quad &\mathit{Bein}(l_1) \wedge \mathit{Bein}(l_2) \wedge \mathit{Rumpf}(b) \wedge\\ &\mathit{TeilVon}(l_1, a) \wedge \mathit{TeilVon}(l_2, a) \wedge \mathit{TeilVon}(b, a) \wedge\\ &\mathit{Befestigt}(l_1, b) \wedge \mathit{Befestigt}(l_2, b) \wedge\\ &l_1 \neq l_2 \wedge [\forall l_3 \quad \mathit{Bein}(l_3) \wedge \mathit{TeilVon}(l_3, a) \Rightarrow (l_3 = l_1 \vee l_3 = l_2)].\end{aligned}$$

Die Notation für „genau zwei“ ist etwas unansehnlich; wir sind gezwungen zu sagen, dass es zwei Beine gibt, dass sie nicht dieselben sind und dass, wenn jemand ein drittes Bein besitzt, dies dasselbe wie eines der anderen beiden sein muss. In *Abschnitt 12.5.2* beschreiben wir einen Formalismus namens Beschreibungslogik, Bedingungen wie „genau zwei“ einfacher darstellen zu können.

Wir können eine *TeilePartition*-Relation definieren, die analog zur *Partition*-Relation für Kategorien ist (siehe Übung 12.8). Ein Objekt setzt sich aus den Teilen in seiner *TeilePartition* zusammen und man kann sagen, es leitet einige Eigenschaften von diesen Teilen ab. Die Masse eines zusammengesetzten Objektes beispielsweise ist die Summe der Massen aller Teile. Beachten Sie, dass das bei Kategorien, die keine Masse haben, nicht der Fall ist, selbst wenn ihre Elemente eine Masse haben.

Darüber hinaus ist es sinnvoll, zusammengesetzte Objekte mit definiten Teilen, aber ohne bestimmte Struktur zu definieren. Beispielsweise könnten wir sagen: „Die Äpfel in dieser Tasche wiegen zwei Pfund.“ Die Versuchung ist groß, dieses Gewicht der *Menge* der Äpfel in der Tasche zuzuordnen, aber das wäre ein Fehler, weil die Menge ein abstraktes mathematisches Konzept ist, das Elemente enthalten kann, aber kein Gewicht hat. Stattdessen führen wir ein neues Konzept ein, das wir als **Bündel** bezeichnen wollen. Sind die Äpfel beispielsweise Apfel_1, Apfel_2 und Apfel_3, dann bezeichnet

$$\mathit{BündelVon}(\{\mathit{Apfel}_1, \mathit{Apfel}_2, \mathit{Apfel}_3\})$$

das zusammengesetzte Objekt mit drei Äpfeln als Teile (nicht Elemente). Jetzt können wir das Bündel als normales, wenn auch unstrukturiertes Objekt verwenden. Beachten Sie, dass $\mathit{BündelVon}(\{x\}) = x$ ist. Darüber hinaus ist $\mathit{BündelVon}(\mathit{Äpfel})$ das zusammen-

gesetzte Objekt, das aus allen Äpfeln besteht – nicht zu verwechseln mit *Äpfel*, der Kategorie oder Menge aller Äpfel.

Natürliche Arten

Einige Kategorien haben strenge Definitionen: Ein Objekt ist genau dann ein Dreieck, wenn es ein Polygon mit drei Seiten ist. Die meisten Kategorien der realen Welt dagegen haben keine klar abgegrenzte Definition: Es handelt sich dabei um Kategorien **natürlicher Arten**. Zum Beispiel haben Tomaten häufig ein dunkles Scharlachrot, sie sind fast kugelförmig, mit einer Einbuchtung ganz oben, wo sie an den Stamm angewachsen waren, sie haben 5 bis 10 cm Durchmesser und eine dünne, aber feste Haut; im Inneren haben sie Fleisch, Samen und Saft. Es gibt jedoch eine Variante: Einige Tomaten sind orange, unreife Tomaten sind grün, manche sind größer oder kleiner als der Durchschnitt und Kirschtomaten sind alle klein. Anstatt eine vollständige Definition von Tomaten zu geben, haben wir mehrere Funktionsmerkmale, anhand derer Objekte identifiziert werden können, bei denen es sich offensichtlich um typische Tomanten handelt, aber mit denen keine anderen Objekte identifiziert werden können. (Könnte es eine Tomate mit pelziger Haut geben, wie ein Pfirsich?)

Das stellt ein Problem für einen logischen Agenten dar. Der Agent kann nicht sicher sein, dass ein Objekt, das er wahrgenommen hat, eine Tomate ist, und selbst wenn er sicher wäre, könnte er nicht sicher sein, welche der Eigenschaften einer typischen Tomate diese besitzt. Dieses Problem ist eine unvermeidbare Folge der Tatsache, dass sich ein Agent in teilweise beobachtbaren Umgebungen bewegt.

Ein sinnvoller Ansatz ist, das, was für alle Instanzen einer Kategorie wahr ist, von all dem abzugrenzen, was nur für typische Instanzen wahr ist. Zusätzlich zur Kategorie $Tomaten$ haben wir also die Kategorie $Typisch(Tomaten)$. Hier bildet die Funktion $Typisch$ eine Kategorie auf die Unterklasse ab, die nur typische Instanzen enthält:

$$Typisch(c) \subseteq c$$

Das meiste Wissen über natürliche Arten ist das über ihre typischen Instanzen:

$$x \in Typisch(Tomaten) \Rightarrow Rot(x) \wedge Rund(x)$$

Wir können also praktische Fakten über Kategorien ohne exakte Definition formulieren. Die Schwierigkeit, exakte Definitionen für die meisten natürlichen Kategorien bereitzustellen, wird von Wittgenstein (1953) erklärt. Er verwendete das Beispiel von *Spielen*, um zu zeigen, dass die Elemente einer Kategorie „Familienähnlichkeiten" aufweisen statt notwendiger und hinreichender Charakteristiken: Welche strenge Definition schließt Schach, Fangen, Solitär und Völkerball ein?

Der Nutzen des Konzeptes einer strengen Definition wurde auch von Quine (1953) angezweifelt. Er legte dar, dass die Definition von „Junggeselle" als nicht verheirateter erwachsener Mann zweifelhaft ist; man könnte beispielsweise eine Aussage wie „Der Papst ist ein Junggeselle" anzweifeln. Diese Aussage ist zwar streng genommen nicht *falsch*, aber die Formulierung ist eher *unglücklich*, weil sie unbeabsichtigte Inferenzen auf Seiten des Zuhörers nach sich zieht. Diese Spannung ließe sich auflösen, indem man zwischen logischen Definitionen, die für eine interne Wissensrepräsentation geeignet sind, und den abgestufteren Kriterien für eine treffendere linguistische Verwendung unterscheidet. Das Letztere könnte man erreichen, indem man die aus der Ersteren abgeleiteten Behauptungen „filtert". Außerdem ist es möglich, dass Fehler in der linguistischen Verwendung als Feedback für die Abänderung interner Definitionen dienen, sodass eine Filterung überflüssig wird.

Wir können *BündelVon* in Bezug auf die *TeilVon*-Relation definieren. Offensichtlich ist jedes Element s Teil von *BündelVon*(s):

$$\forall x \quad x \in s \Rightarrow \mathit{TeilVon}(x, \mathit{BündelVon}(s)).$$

Darüber hinaus ist *BündelVon*(s) *das kleinste Objekt, das diese Bedingung erfüllt*. Mit anderen Worten muss *BündelVon*(s) Teil jedes Objektes sein, das alle Elemente von s als Teile enthält:

$$\forall y\, [\forall x \quad x \in s \Rightarrow \mathit{TeilVon}(x, y)] \Rightarrow \mathit{Teilvon}(\mathit{BündelVon}(s), y).$$

Diese Axiome sind ein Beispiel für eine allgemeine Technik, die als **logische Minimierung** bezeichnet wird, d.h., es wird ein Objekt als das kleinste definiert, das bestimmte Bedingungen erfüllt.

12.2.2 Maße

Sowohl in den wissenschaftlichen als auch in den ganz gewöhnlichen Theorien dieser Welt haben Objekte eine Höhe, eine Masse, Kosten usw. Die Werte, die wir für diese Eigenschaften zuweisen, werden als **Maße** bezeichnet. Normale quantitative Maße sind recht einfach zu repräsentieren. Wir stellen uns vor, dass das Universum abstrakte „Maßobjekte“ enthält, wie etwa die *Länge*, d.h. die Länge dieses Liniensegments: ⊢——————⊣ Wir bezeichnen diese Länge als 1,5 Zoll oder 3,81 Zentimeter. Dieselbe Länge hat also innerhalb unserer natürlichen Sprache unterschiedliche Namen. Wir stellen die Länge mit einer **Einheitenfunktion** dar, die eine Zahl als Argument übernimmt. (Ein alternatives Schema wird in Übung 12.9 vorgestellt.) Wenn das Liniensegment L_1 heißt, können wir schreiben:

$$\mathit{Länge}(L_1) = \mathit{Zoll}(1{,}5) = \mathit{Zentimeter}(3{,}81).$$

Die Umwandlung zwischen den Einheiten erfolgt, indem man das Vielfache einer Einheit der anderen Einheit gleichsetzt:

$$\mathit{Zentimeter}(2{,}54 \times d) = \mathit{Zoll}(d).$$

Ähnliche Axiome können für Pfund und Kilogramm, Sekunden und Tage sowie Euro und Cent geschrieben werden. Maße können genutzt werden, um Objekte wie folgt zu beschreiben:

$$\mathit{Durchmesser}(\mathit{Basketball}_{12}) = \mathit{Zoll}(9{,}5)$$
$$\mathit{ListenPreis}(\mathit{Basketball}_{12}) = (19)\,€$$
$$d \in \mathit{Tage} \Rightarrow \mathit{Dauer}(d) = \mathit{Stunden}(24).$$

Beachten Sie, dass (1) € keine Euromünze darstellt! Man kann zwei Euromünzen haben, es gibt aber nur ein Objekt namens (1) €. Beachten Sie außerdem, dass sich *Zoll*(0) und *Zentimeter*(0) auf dieselbe Nulllänge beziehen, aber nicht identisch mit anderen Nullmaßen sind, wie etwa *Sekunden*(0).

Einfache quantitative Maße sind leicht darzustellen. Andere Maße sind problematischer, weil sie keine vereinheitlichte Werteskala aufweisen. Übungen sind unterschiedlich schwer, Nachspeisen sind unterschiedlich köstlich und Gedichte sind unterschiedlich schön, aber diesen Qualitäten können keine Zahlen zugeordnet werden. Man könnte in einem Moment reinen Buchhaltungsdenkens solche Eigenschaften als unnütz für das logische Schließen ablehnen oder, was noch schlimmer wäre, den Versuch unternehmen, Schönheit eine numerische Skala zuzuordnen. Das wäre ein schwerer

Fehler, weil es unnötig ist. Der wichtigste Aspekt der Maße ist nicht der jeweilige numerische Wert, sondern die Tatsache, dass Maße in eine bestimmte *Reihenfolge* gebracht werden können.

Auch wenn Maße keine Zahlen sind, können wir sie dennoch vergleichen, indem wir ein Reihenfolgesymbol wie etwa > verwenden. Beispielsweise könnten wir glauben, dass die Übungen von Norvig schwieriger als die von Russell sind und dass man bei schwierigeren Übungen oft weniger Punkte erhält:

$e_1 \in \mathit{Übungen} \land e_2 \in \mathit{Übungen} \land \mathit{Geschrieben}(\mathit{Norvig}, e_1) \land \mathit{Geschrieben}(\mathit{Russell}, e_2) \Rightarrow \mathit{Schwierigkeit}(e_1) > \mathit{Schwierigkeit}(e_2)$

$e_1 \in \mathit{Übungen} \land e_2 \in \mathit{Übungen} \land \mathit{Schwierigkeit}(e_1) > \mathit{Schwierigkeit}(e_2) \Rightarrow \mathit{ErwartetePunktzahl}(e_1) < \mathit{ErwartetePunktzahl}(e_2)$.

Anhand dessen kann man entscheiden, welche Übungen man durchführen will, selbst wenn nie numerische Werte für den Schwierigkeitsgrad vergeben wurden. (Man muss jedoch in der Lage sein, festzustellen, von wem die einzelnen Übungen geschrieben wurden.) Derartige monotone Beziehungen zwischen Maßen bilden die Grundlage für den Bereich der **qualitativen Physik**, einem Teilbereich der KI, der untersucht, wie man für physikalische Systeme schließt, ohne detaillierte Gleichungen und numerische Simulationen vorliegen zu haben. Die qualitative Physik wird im Abschnitt über die historischen Hinweise genauer beschrieben.

12.2.3 Objekte: Dinge und Stoffe

Man kann sagen, die reale Welt besteht aus elementaren Objekten (z.B. atomaren Teilchen) und daraus zusammengesetzten Objekten. Durch das Schließen auf der Ebene großer Objekte, wie beispielsweise Äpfel oder Autos, können wir die Komplexität kompensieren, welche die Verarbeitung riesiger Mengen einzelner elementarer Objekte mit sich bringt. Es gibt jedoch einen wesentlichen Teil der Realität, der scheinbar jeder offensichtlichen **Individuation**[3] (Unterteilung in unterschiedliche Objekte) trotzt. Wir geben diesem Teil den generischen Namen **Stoffe**. Angenommen, ich habe Butter und ein Erdferkel vor mir. Ich kann sagen, ich habe ein Erdferkel, aber es gibt keine offensichtliche Anzahl an „Butter-Objekten“, weil jeder Teil eines Butter-Objektes wiederum ein Butter-Objekt ist, zumindest bis die Teile wirklich sehr klein werden. Das ist der wichtigste Unterschied zwischen *Stoffen* und *Dingen*. Wenn wir ein Erdferkel halbieren, erhalten wir (leider) nicht zwei Erdferkel.

Die natürliche Sprache unterscheidet deutlich zwischen *Stoffen* und *Dingen*. Wir sagen „ein Erdferkel“, aber bis auf einige wenige Ausnahmen kann man nicht sagen „eine Butter“. Linguisten unterscheiden zwischen **zählbaren Substantiven**, wie etwa Erdferkel, Löcher oder Theoreme, und **Massesubstantiven**, wie etwa Butter, Wasser oder Energie. Mehrere konkurrierende Ontologien behaupten, diese Unterscheidung verarbeiten zu können. Wir beschreiben hier nur eine; die anderen sind im Abschnitt über die historischen Hinweise aufgeführt.

Um *Stoffe* korrekt zu repräsentieren, beginnen wir mit dem Offensichtlichsten. Wir brauchen als Objekte in unserer Ontologie zumindest die großen „Stücke“ der *Stoffe*, mit denen wir es zu tun haben. Beispielsweise könnten wir ein Stück Butter als die-

3 Lateinisch individuare = untrennbar/unteilbar machen

selbe Butter erkennen, die am Abend zuvor beim Essen übrig blieb; wir könnten sie nehmen, wiegen, verkaufen oder was auch immer. In dieser Hinsicht ist sie ein Objekt wie das Erdferkel. Wir nennen sie jetzt $Butter_3$. Außerdem definieren wir die Kategorie *Butter*. Informell sind ihre Elemente alle Dinge, von denen man sagen kann, „es ist Butter", einschließlich $Butter_3$. Mit einigen Vorbehalten in Bezug auf sehr kleine Teile, die wir zunächst vernachlässigen, ist jeder Teil eines Butter-Objektes selbst wieder ein Butter-Objekt:

$$b \in Butter \wedge TeilVon(p, b) \Rightarrow p \in Butter.$$

Wir können jetzt sagen, dass Butter bei etwa 30 Grad Celsius schmilzt:

$$b \in Butter \wedge Schmelzpunkt(b, Celsius(30)).$$

Wir könnten weiterhin sagen, dass Butter gelb ist, weniger dicht als Wasser, bei Zimmertemperatur weich, einen hohen Fettanteil hat usw. Andererseits hat Butter keine bestimmte Größe, keine bestimmte Form und kein bestimmtes Gewicht. Wir können spezialisiertere Kategorien von Butter definieren, wie beispielsweise *UngesalzeneButter*, wobei es sich ebenfalls um einen *Stoff* handelt. Die Kategorie *PfundButter* dagegen, die als Elemente alle Butter-Objekte enthält, die ein Pfund wiegen, ist keine Art von *Stoff*! Wenn wir ein Pfund Butter halbieren, erhalten wir leider ebenfalls nicht zwei Pfund Butter.

Letztlich bedeutet es Folgendes: Es gibt Eigenschaften, die **intrinsisch** sind. Sie gehören zu der eigentlichen Substanz des Objektes und nicht zu dem Objekt als Ganzem. Wenn Sie eine Instanz von *Stoff* mittig auseinanderschneiden, behalten die beiden Stücke die intrinsischen Eigenschaften – z.B. Dichte, Siedepunkt, Geschmack, Farbe, Eigentum usw. Dagegen bleiben ihre **extrinsischen** Eigenschaften – Gewicht, Länge, Form usw. – bei einer Unterteilung nicht erhalten. Eine Kategorie von Objekten, die in ihrer Definition nur *intrinsische* Eigenschaften einschließen, ist eine Substanz oder ein Massesubstantiv; eine Klasse, die *irgendwelche* extrinsische Eigenschaften in ihrer Definition enthält, ist ein zählbares Substantiv. Die Kategorie *Stoffe* ist die allgemeinste Substanzkategorie, die keine intrinsischen Eigenschaften spezifiziert. Die Kategorie *Ding* ist die allgemeinste diskrete Objektkategorie, die keine extrinsischen Eigenschaften spezifiziert.

12.3 Ereignisse

Abschnitt 10.4.2 hat gezeigt, wie der Situationskalkül Aktionen und deren Effekte darstellt. Der Situationskalkül ist in seiner Anwendbarkeit beschränkt: Er wurde konzipiert, um eine Welt zu beschreiben, in der Aktionen diskret, unmittelbar und zu jedem Zeitpunkt einzeln auftretend sind. Betrachten Sie eine stetige Aktion wie zum Beispiel das Füllen einer Badewanne. Der Situationskalkül kann sagen, dass die Wanne vor der Aktion leer und nach Abschluss der Aktion voll ist, doch er sagt nichts darüber aus, was während der Aktion passiert. Er ist auch nicht in der Lage, zwei gleichzeitig stattfindende Aktionen zu beschreiben – beispielsweise Zähneputzen, während auf das Volllaufen der Wanne gewartet wird. Um derartige Fälle zu verarbeiten, führen wir einen alternativen Formalismus ein, der als **Ereigniskalkül** bezeichnet wird und auf Zeitpunkten statt auf Situationen basiert.[4]

4 Die Begriffe „Ereignis" und „Aktion" lassen sich austauschbar verwenden. Informell suggeriert „Aktion" einen Agenten, während man mit „Ereignis" die Möglichkeit von agentenlosen Aktionen verbindet.

Der Ereigniskalkül reifiziert[5] Fluenten und Ereignisse. Der Fluent *Bei*(*Shankar*, *Berkeley*) ist ein Objekt, das auf die Tatsache verweist, dass Shankar in Berkeley ist, jedoch an sich nichts darüber aussagt, ob es wahr ist. Um zu behaupten, dass ein Fluent tatsächlich zu einem bestimmten Zeitpunkt wahr ist, verwenden wir das Prädikat T wie in $T(Bei(Shankar, Berkeley), t)$.

Ereignisse werden als Instanzen von Ereigniskategorien beschrieben.[6] Das Ereignis E_1 von Shankar, der von San Francisco nach Washington, D.C. fliegt, wird beschrieben als

$$E_1 \in Flüge \wedge Flieger(E_1, Shankar) \wedge Ausgangspunkt(E_1, SF) \wedge Ziel(E_1, DC).$$

Falls dies zu weitschweifig ist, können wir eine alternative Version der Kategorie von Flugereignissen mit drei Argumenten definieren und sagen

$$E_1 \in Flüge(Shankar, SF, DC).$$

Wir verwenden dann $Passiert(E_1, i)$, um zu sagen, dass das Ereignis E_1 über dem Zeitintervall i stattgefunden hat, und wir sagen das Gleiche in funktionaler Form mit $Extent(E_1) = i$. Zeitintervalle repräsentieren wir durch ein Paar von (Start, Ende)-Zeiten, d.h. $i = (t_1, t_2)$ ist das Zeitintervall, das bei t_1 beginnt und bei t_2 endet. Die vollständige Menge von Prädikaten für die eine Version des Ereigniskalküls lautet:

$T(f, t)$	Fluent f ist zur Zeit t wahr
$Passiert(e, i)$	Ereignis e findet über dem Zeitintervall i statt
$Initiiert(e, f, t)$	Ereignis e bewirkt, dass Fluent f zur Zeit t gültig wird
$Beendet(e, f, t)$	Ereignis e bewirkt, dass Fluent f ab der Zeit t nicht mehr gültig ist
$Abgebrochen(f, i)$	Fluent f wird ab irgendeinem Zeitpunkt während des Intervalls i nicht mehr wahr sein
$Wiederhergestellt(f, i)$	Fluent f wird irgendwann während des Zeitintervalls i wahr

Wir nehmen ein bestimmtes Ereignis, *Start*, an, das den Ausgangszustand beschreibt, indem es sagt, welche Fluenten zur Startzeit initiiert oder terminiert werden. Wir definieren T, indem wir sagen, dass ein Fluent zu einem Zeitpunkt gültig ist, wenn der Fluent durch ein Ereignis zu einer Zeit in der Vergangenheit initiiert und nicht durch ein dazwischenliegendes Ereignis falsch gemacht (abgebrochen) wurde. Ein Fluent ist nicht mehr gültig, wenn er durch ein Ereignis terminiert und nicht durch ein anderes Ereignis wahr gemacht (wiederhergestellt) wurde. Formell lauten die Axiome:

$$Passiert(e, (t_1, t_2)) \wedge Initiiert(e, f, t_1) \wedge \neg Abgebrochen\,(f, (t_1, t)) \wedge t_1 < t \Rightarrow T(f, t)$$
$$Passiert(e, (t_1, t_2)) \wedge Beendet(e, f, t_1) \wedge \neg Wiederhergestellt(f, (t_1, t)) \wedge t_1 < t \Rightarrow \neg T(f, t),$$

wobei *Abgebrochen* und *Wiederhergestellt* durch

$$Abgebrochen\,(f, (t_1, t_2)) \Leftrightarrow \exists\, e, t, t_3\; Passiert(e, (t, t_3)) \wedge t_1 \leq t < t_2 \wedge Beendet(e, f, t)$$
$$Wiederhergestellt(f, (t_1, t_2)) \Leftrightarrow \exists\, e, t, t_3\; Passiert(e, (t, t_3)) \wedge t_1 \leq t < t_2 \wedge Initiiert(e, f, t)$$

definiert sind.

5 Auch Verdinglichung; meint hier das Verbinden von Zeitintervallen und Ereignissen

6 Einige Versionen des Ereigniskalküls unterscheiden Ereigniskategorien nicht von Instanzen der Kategorien.

Es ist komfortabel, T zu erweitern, um über Intervallen sowie Zeitpunkten zu arbeiten; ein Fluent gilt über einem Intervall, wenn er zu jedem Zeitpunkt innerhalb des Intervalls gilt:

$$T(f, (t_1, t_2)) \Leftrightarrow [\forall t\, (t_1 \leq t < t_2) \Rightarrow T(f, t)].$$

Fluenten und Aktionen werden mit domänenspezifischen Axiomen definiert, die ähnlich den Nachfolgerzustandsaxiomen sind. Zum Beispiel können wir sagen, dass ein Agent der Wumpus-Welt nur beim Start die Möglichkeit hat, einen Pfeil zu erhalten, und er den Pfeil nur auf eine Art verwenden kann, indem er ihn schießt:

$$\textit{Initiiert}(e, \textit{HaveArrow}\,(a), t) \Leftrightarrow e = \textit{Start}$$
$$\textit{Beendet}(e, \textit{HaveArrow}\,(a), t) \Leftrightarrow e \in \textit{Shootings}(a).$$

Durch Reifizieren lassen sich Ereignissen beliebige Informationen hinzufügen. Zum Beispiel können wir mit $\textit{Bumpy}(E_1)$ sagen, dass der Flug von Shankar holprig war. In einer Ontologie, wo Ereignisse n-stellige Prädikate sind, gibt es keine Möglichkeit, derartige Zusatzinformationen hinzuzufügen; der Übergang zu einem $(n + 1)$-stelligen Prädikat ist keine skalierbare Lösung.

Den Ereigniskalkül können wir erweitern, um simultane Ereignisse (dass zum Beispiel zum Wippen zwei Personen notwendig sind), exogene Ereignisse (wie zum Beispiel Wind, der bläst und die Position eines Objektes ändert), stetige Ereignisse (wie zum Beispiel einen ständig steigenden Wasserpegel in der Badewanne) und andere Komplikationen darstellen zu können.

12.3.1 Prozesse

Die Ereignisse, die wir bisher gesehen haben, sind sogenannte **diskrete Ereignisse** – sie haben eine definite Struktur. Die Reise von Shankar hat einen Anfang, eine Mitte und ein Ende. Wird sie auf der Hälfte unterbrochen, ist das Ereignis ein anderes – es wäre keine Reise mehr von San Francisco nach Washington, sondern stattdessen eine Reise von San Francisco nach irgendwo über Kanada. Dagegen hat die durch *Flüge* bezeichnete Kategorie von Ereignissen eine andere Qualität. Wenn wir ein kleines Intervall aus dem Flug von Shankar betrachten, etwa den dritten 20-minütigen Abschnitt (während er unruhig auf eine neue Tüte Erdnüsse wartet), ist dieses Ereignis immer noch Element von *Flüge*. Und genau das trifft für jedes Unterintervall zu.

Kategorien von Ereignissen mit dieser Eigenschaft werden als **Prozesskategorien** oder **fließende Ereigniskategorien** bezeichnet. Jeder Prozess e, der über einem Intervall stattfindet, findet auch über jedem Unterintervall statt:

$$(e \in \textit{Processes}) \wedge \textit{Passiert}(e, (t_1, t_4)) \wedge (t_1 < t_2 < t_3 < t_4) \Rightarrow \textit{Passiert}(e, (t_2, t_3)).$$

Die Unterscheidung zwischen fließenden und nicht fließenden Ereignissen entspricht genau dem Unterschied zwischen Substanzen oder *Stoffen* und Einzelobjekten oder *Dingen*. So hat man auch fließende Ereignisse als **temporäre Substanzen** bezeichnet, während Substanzen wie Butter **räumliche Substanzen** sind.

12.3.2 Zeitintervalle

Der Ereigniskalkül bringt uns die Möglichkeit, über die Zeit und Zeitintervalle zu sprechen. Wir betrachten zwei Arten von Zeitintervallen: Momente und ausgedehnte Intervalle. Diese unterscheiden sich lediglich dadurch, dass Momente die Dauer null haben.

$$Partition(\{Momente, AusgedehnteIntervalle\}, Intervalle)$$
$$i \in Momente \Leftrightarrow Dauer(i) = Sekunden(0)$$

Anschließend führen wir eine Zeitskala ein und ordnen Momenten Punkte auf dieser Skala zu, sodass wir absolute Zeiten erhalten. Die Zeitskala ist willkürlich gewählt; wir messen in Sekunden und sagen, dass der Moment um Mitternacht (GMT) am 1. Januar 1900 die Zeit 0 hat. Die Funktionen *Anfang* und *Ende* wählen den ersten und den letzten Moment in einem Intervall aus und die Funktion *Zeit* gibt für einen Moment den Punkt auf der Zeitskala zurück. Die Funktion *Dauer* gibt die Differenz zwischen der Endzeit und der Anfangszeit an.

$$Intervall(i) \Rightarrow Dauer(i) = (Zeit(Ende(i)) - Zeit(Anfang(i)))$$
$$Zeit(Anfang(AD1900)) = Sekunden(0)$$
$$Zeit(Start(AD2001)) = Sekunden(3187324800)$$
$$Zeit(Ende(AD2001)) = Sekunden(3218860800)$$
$$Dauer(AD2001) = Sekunden(31536000)$$

Um diese Zahlen leichter lesbar zu machen, führen wir auch eine Funktion *Datum* ein, die sechs Argumente (Stunden, Minuten, Sekunden, Tag, Monat und Jahr) übernimmt und einen Zeitpunkt zurückgibt:

$$Zeit(Anfang(AD2001)) = Datum(0,0,0,1, Jan, 2001)$$
$$Datum(0,20,21,24,1,1995) = Sekunden(3000000000).$$

Zwei Intervalle *Treffen* sich, wenn die Endzeit des einen gleich der Anfangszeit des zweiten ist. Die vollständige Menge der Intervallbeziehungen, wie sie Allen (1983) vorgeschlagen hat, ist in ▶ Abbildung 12.2 grafisch dargestellt und sieht logisch folgendermaßen aus:

$$\begin{aligned}
Treffen(i, j) &\Leftrightarrow Ende(i) = Anfang(j)\\
Vor(i, j) &\Leftrightarrow Ende(i) < Anfang(j)\\
Nach(j, i) &\Leftrightarrow Vor(i, j)\\
Während(i, j) &\Leftrightarrow Anfang(j) < Anfang(i) < Ende(i) < Ende(j)\\
Überlappen(i, j) &\Leftrightarrow Anfang(i) < Anfang(j) < Ende(i) < Ende(j)\\
Beginnt(i, j) &\Leftrightarrow Anfang(i) = Anfang(j)\\
Endet(i, j) &\Leftrightarrow Ende(i) = Ende(j)\\
IstGleich(i, j) &\Leftrightarrow Anfang(i) = Anfang(j) \wedge Ende(i) = Ende(j).
\end{aligned}$$

Alle Relationen haben ihre intuitiven Bedeutungen, mit Ausnahme von *Überlappen*: Normalerweise stellen wir uns Überlappen als symmetrisch vor (wenn *i* sich mit *j* überlappt, dann überlappt sich *j* mit *i*), doch gilt in dieser Definition *Überlappen*(*i*, *j*) nur, wenn *i* vor *j* beginnt. Um zu sagen, dass die Regentschaft von Elizabeth II unmittelbar auf die von George VI folgte und dass sich die Regentschaft von Elvis mit den 1950er Jahren überlappte, schreiben wir:

$$Treffen(RegiertVon(GeorgeVI), RegiertVo4n(ElizabethII))$$
$$Überlappen(Fünfziger, RegiertVon(Elvis))$$
$$Anfang(Fünfziger) = Anfang(AD1950)$$
$$Ende(Fünfziger) = Ende(AD1959).$$

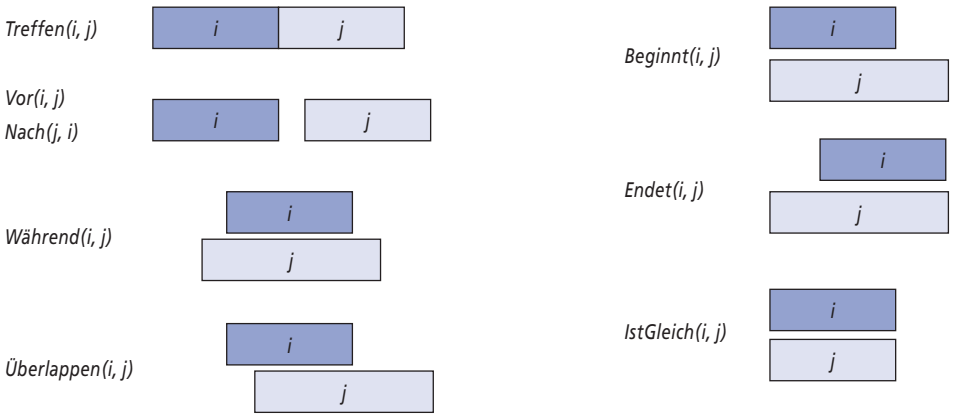


Abbildung 12.2: Prädikate für Zeitintervalle.

12.3.3 Fluenten und Objekte

Wir haben bereits erwähnt, dass man physische Objekte als verallgemeinerte Ereignisse betrachten kann, weil ein physisches Objekt ein Stück Raum/Zeit ist. Zum Beispiel kann man sich *USA* als Ereignis vorstellen, das 1776 als Vereinigung von 13 Staaten begann und sich heute als Vereinigung von 50 Staaten immer noch weiterentwickelt. Wir können die sich ändernden Eigenschaften von *USA* unter Verwendung von Zustandsfluenten beschreiben, zum Beispiel als *Bevölkerung*(*USA*). Eine Eigenschaft der USA, die sich alle vier oder acht Jahre ändert, Attentate ausgenommen, ist ihr Präsident. Man könnte vorschlagen, dass *Präsident*(*USA*) ein logischer Term ist, der zu unterschiedlichen Zeiten ein jeweils unterschiedliches Objekt darstellt. Leider ist das nicht möglich, weil ein Term genau ein Objekt in einer bestimmten Modellstruktur darstellt. (Der Term *Präsident*(*USA*, *t*) kann abhängig von dem Wert von *t* unterschiedliche Objekte darstellen, aber unsere Ontologie trennt Zeitindizes von Fluenten.) Die einzige Möglichkeit ist, dass *Präsident*(*USA*) ein einzelnes Objekt bezeichnet, das zu unterschiedlichen Zeiten aus verschiedenen Menschen besteht. Dieses Objekt ist von 1789 bis 1797 George Washington, von 1797 bis 1801 John Adams usw., wie in ▶ Abbildung 12.3 gezeigt.

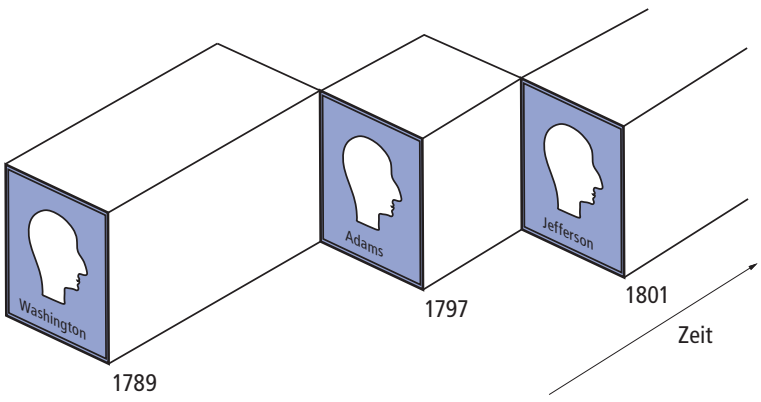


Abbildung 12.3: Eine schematische Darstellung des Objektes Präsident(USA) für die ersten 15 Jahre ihrer Existenz.

Um zu sagen, dass im gesamten Jahr 1790 George Washington der Präsident war, schreiben wir:

$$T(IstGleich(Präsident(USA), GeorgeWashington), AD1790)$$

Wir verwenden das Funktionssymbol *IstGleich* anstelle des logischen Standardprädikates =, weil wir kein Prädikat als Argument an *T* angeben können und weil die Interpretation *nicht* ist, dass *GeorgeWashington* und *Präsident*(*USA*) in 1790 logisch identisch sind; logische Identität kann sich nicht mit der Zeit ändern. Die Identität besteht zwischen den Unterereignissen jedes Objektes, die durch den Zeitraum 1790 definiert sind.

12.4 Mentale Ereignisse und mentale Objekte

Die Agenten, die wir bisher konstruiert haben, besitzen Glauben und können neuen Glauben ableiten. Wissen über das eigene Wissen und die Schlussfolgerungsprozesse sind nützlich, um Inferenz zu steuern. Angenommen, Alice fragt: „Wie lautet die Quadratwurzel von 1764?“, und Bob antwortet: „Ich weiß nicht.“ Wenn Alice darauf besteht, „intensiver nachzudenken“, sollte Bob erkennen, dass sich diese Frage in der Tat mit etwas mehr Nachdenken beantworten lässt. Lautet die Frage dagegen: „Hat sich deine Mutter jetzt hingesetzt?“, sollte Bob erkennen, dass intensiveres Nachdenken hier kaum hilft. Wissen über das Wissen anderer Agenten ist ebenfalls wichtig; Bob sollte erkennen, dass seine Mutter weiß, ob sie sitzt oder nicht, und dass er es herausfinden kann, wenn er sie fragt.

Wir brauchen ein Modell der mentalen Objekte, die sich im Kopf einer Person (oder in einer Wissensbasis) befinden, sowie der mentalen Prozesse, die diese mentalen Objekte manipulieren. Das Modell muss nicht detailliert sein. Wir wollen nicht vorhersagen können, wie viele Millisekunden ein bestimmter Agent braucht, um einen Schluss zu ziehen. Wir sind zufrieden, wenn wir schließen können, dass Mutter weiß, ob sie sitzt oder nicht.

Wir beginnen mit den **aussagenlogischen Standpunkten**, die ein Agent gegenüber mentalen Objekten haben kann: Standpunkte wie zum Beispiel *Glaubt*, *Weiß*, *Will*, *Beabsichtigt* und *Informiert*. Die Schwierigkeit besteht darin, dass sich diese Einstellungen nicht wie „normale“ Prädikate verhalten. Nehmen wir zum Beispiel an, wir behaupten, dass Lois weiß, dass Superman fliegen kann:

$$Weiß(Lois, KannFliegen(Superman)).$$

Dieser Ausdruck hat den kleinen Schönheitsfehler, dass wir normalerweise von *KannFliegen*(*Superman*) als Satz denken, doch hier erscheint er als Term. Dieses Problem lässt sich beheben, indem man *KannFliegen*(*Superman*) reifiziert und damit zu einem Fluenten macht. Schwerwiegender ist folgendes Problem: Wenn es wahr ist, dass Clark Kent Superman ist, müssen wir schließen, dass Lois weiß, dass Clark fliegen kann:

$$(Superman = Clark) \wedge Weiß(Lois, KannFliegen(Superman)) \models Knows(Lois, KannFliegen(Clark)).$$

Dies ist eine Konsequenz aus der Tatsache, dass Gleichheitsschließen in Logik integriert ist. Normalerweise ist das eine gute Sache; wenn unser Agent weiß, dass $2 + 2 = 4$ und $4 < 5$ ist, soll unser Agent auch wissen, dass $2 + 2 < 5$ ist. Diese Eigenschaft wird als **referentielle Transparenz** bezeichnet – es spielt keine Rolle, welchen Term eine Logik verwendet, um auf ein Objekt zu verweisen, es zählt allein das Objekt, das der Term benennt. Doch für aussagenlogische Standpunkte wie *Glaubt* und *Weiß* hätten wir gern referentielle Opazität (Undurchsichtigkeit) – die verwendeten Terme spielen eine Rolle, weil nicht alle Agenten wissen, welche Terme koreferentiell sind.

Diesem Problem lässt sich mit **modaler Logik** beikommen. Normale Logik hat mit einer einzigen Modalität zu tun, der Modalität von Wahrheit, mit der wir „P ist wahr“ ausdrücken können. Modale Logik schließt spezielle modale Operatoren ein, die Sätze (statt Terme) als Argumente übernehmen. Zum Beispiel wird „A weiß P“ durch die Notation $\mathbf{K}_A P$ dargestellt, wobei $\mathbf{K}$ der modale Operator für Wissen ist. Er übernimmt zwei Argumente – einen Agenten (als tiefgestellter Index geschrieben) und einen Satz. Die Syntax modaler Logik ist die gleiche wie Logik erster Stufe, außer dass Sätze auch mit modalen Operatoren gebildet werden können.

Die Semantik modaler Logik ist komplizierter. In Logik erster Stufe enthält ein **Modell** eine Menge von Objekten und eine Interpretation, die jeden Namen dem passenden Objekt, der Relation oder der Funktion zuordnet. In modaler Logik wollen wir in der Lage sein, beide Möglichkeiten zu betrachten – dass Clark die geheime Identität von Superman hat und dass er sie nicht hat. Demzufolge brauchen wir ein Modell, das komplizierter ist und aus einer Sammlung von **möglichen Welten** und nicht nur aus einer einzigen wahren Welt besteht. Die Welten sind durch **Zugänglichkeitsrelationen** (oder: **Erreichbarkeitsrelationen**) in einem Graphen miteinander verknüpft – eine Relation für jeden modalen Operator. Wir sagen, dass Welt w_1 von Welt w_0 aus in Bezug auf den modalen Operator $\mathbf{K}_A$ zugänglich ist, wenn alles in w_1 mit dem konsistent ist, was A in w_0 weiß. Wir schreiben dies als $Acc(\mathbf{K}_A, w_0, w_1)$. In Diagrammen wie in ▶ Abbildung 12.4 stellen wir die Zugänglichkeit als Pfeil zwischen möglichen Welten dar. Zum Beispiel ist in der realen Welt Bukarest die Hauptstadt von Rumänien, doch für einen Agenten, der dies nicht weiß, sind andere mögliche Welten zugänglich, einschließlich derjenigen, wo die Hauptstadt von Rumänien Sibiu oder Sofia ist. Eine Welt, in der $2 + 2 = 5$ ist, wäre vermutlich für keinen Agenten zugänglich.

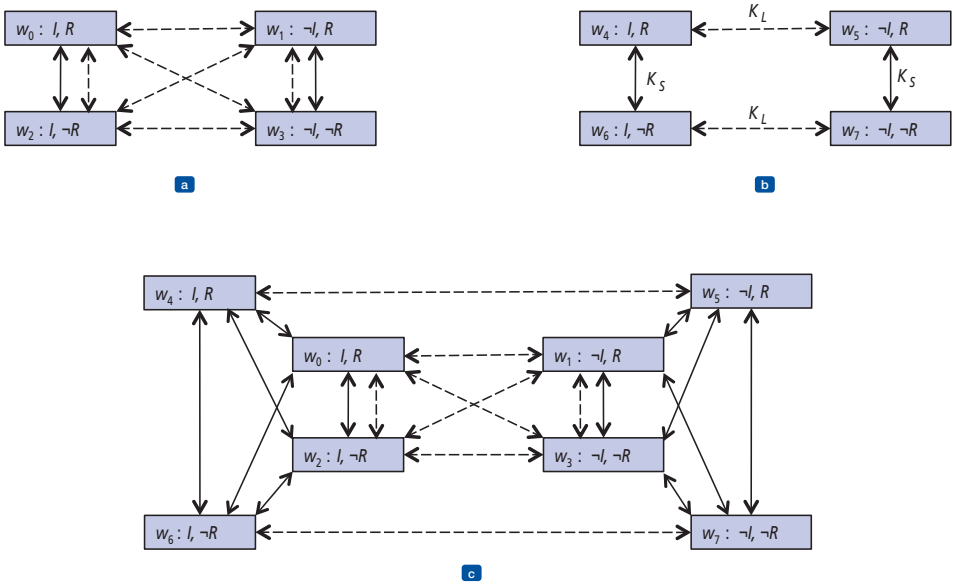


Abbildung 12.4: Mögliche Welten mit Zugänglichkeitsrelationen $\mathbf{K}_{Superman}$ (durchgehende Pfeile) und $\mathbf{K}_{Lois}$ (gestrichelte Pfeile). Die Aussage R bedeutet „Der Wetterbericht sagt für morgen Regen voraus“ und I heißt: „Die geheime Identität von Superman ist Clark Kent“. Alle Welten sind sich selbst zugänglich; die Pfeile von einer Welt zu sich selbst sind nicht dargestellt.

Im Allgemeinen ist ein Wissensatom $\mathbf{K}_A P$ in einer Welt w genau dann wahr, wenn P in jeder Welt, die von w aus zugänglich ist, wahr ist. Die Wahrheit komplexerer Sätze wird durch rekursive Anwendung dieser Regel und der normalen Regeln von Logik erster Stufe abgeleitet. Das heißt, dass modale Logik verwendet werden kann, um über verschachtelte Wissenssätze zu schließen: was der eine Agent über das Wissen eines anderen Agenten weiß. Zum Beispiel können wir sagen, dass selbst wenn Lois nicht weiß, ob die geheime Identität Clark Kent ist, sie weiß, dass Clark es weiß:

$$\mathbf{K}_{Lois}\,[\mathbf{K}_{Clark}\,Identity(Superman, Clark) \vee \mathbf{K}_{Clark}\,\neg Identity(Superman, Clark)].$$

Abbildung 12.4 zeigt einige mögliche Welten für diese Domäne mit Zugänglichkeitsrelationen für Lois und Superman.

Im Diagramm links oben ist es Allgemeingut, dass Superman seine Identität kennt und weder er noch Lois den Wetterbericht gesehen haben. So sind in w_0 die Welten w_0 und w_2 für Superman zugänglich; vielleicht wird Regen vorhergesagt, vielleicht auch nicht. Für Lois sind alle vier Welten untereinander zugänglich; sie weiß nichts über den Bericht oder ob Clark Superman ist. Aber sie weiß, dass Superman weiß, ob er Clark ist, da in jeder Welt, die für Lois zugänglich ist, Superman entweder I oder $\neg I$ weiß. Lois weiß nicht, welcher Fall zutrifft, doch egal wie weiß sie, dass Superman es weiß.

Im Diagramm rechts oben ist es Allgemeingut, dass Lois den Wetterbericht gesehen hat. So weiß sie in w_4, dass Regen vorhergesagt ist, und in w_6 weiß sie, dass kein Regen vorhergesagt wird. Superman kennt den Bericht nicht, weiß aber, dass Lois ihn kennt, weil in jeder Welt, die ihm zugänglich ist, sie entweder R oder $\neg R$ weiß.

Das Diagramm unten stellt das Szenario dar, in dem Allgemeingut ist, dass Superman seine Identität kennt und Lois den Wetterbericht gesehen oder nicht gesehen haben kann. Dies stellen wir durch eine Kombination der beiden oberen Szenarios dar und zeigen mit zusätzlichen Pfeilen, dass Superman nicht weiß, welches Szenario tatsächlich gilt. Lois weiß es nicht, sodass wir für sie keine Pfeile hinzufügen müssen. In w_0 weiß Superman immer noch I, aber nicht R, und jetzt weiß er nicht, ob Lois R weiß. Aus dem, was Superman weiß, kann er in w_0 oder w_2 sein, in welchem Fall Lois nicht weiß, ob R wahr ist, oder er könnte in w_4 sein, in welchem Fall sie R weiß, oder w_6, in welchem Fall sie $\neg R$ weiß.

Da es eine unendliche Anzahl möglicher Welten gibt, führt man einfach diejenigen ein, die man für die Darstellung dessen braucht, was man zu modellieren versucht. Eine neue mögliche Welt wird benötigt, um über unterschiedliche mögliche Fakten zu sprechen (z.B. Regen wird vorhergesagt oder nicht) oder über unterschiedliche Zustände des Wissens zu sprechen (z.B. Lois weiß, dass Regen angekündigt ist). Das bedeutet, dass zwei mögliche Welten wie zum Beispiel w_4 und w_0 in Abbildung 12.4 dieselben Basisfakten über die Welt haben könnten, sich aber in ihren Zugänglichkeitsrelationen unterscheiden und demzufolge in den Fakten über das Wissen.

Modale Logik löst einige verzwickte Fragen im Zusammenspiel von Quantoren und Wissen. So ist der Satz „Bond weiß, dass jemand ein Spion ist“ mehrdeutig. Die erste Lesart ist, dass es einen bestimmten Jemand gibt, von dem Bond weiß, dass er ein Spion ist; dies können wir schreiben als

$$\exists\, x\, \mathbf{K}_{Bond}\, Spion(x)$$

was in modaler Logik bedeutet, dass es ein x gibt, das – in allen zugänglichen Welten – Bond als Spion bekannt ist. Die zweite Lesart ist, dass Bond lediglich weiß, dass es mindestens einen Spion gibt:

$$\mathbf{K}_{Bond} \exists x \; Spion(x).$$

Bei Interpretation in modaler Logik heißt das, dass es in jeder zugänglichen Welt ein x gibt, das ein Spion ist, es aber nicht dasselbe x in jeder Welt sein muss.

Nachdem wir nun einen modalen Operator für Wissen haben, können wir Axiome dafür schreiben. Zuerst lässt sich sagen, dass Agenten in der Lage sind, Schlüsse zu ziehen; wenn ein Agent P kennt und weiß, P impliziert Q, weiß der Agent Q:

$$(\mathbf{K}_a P \wedge \mathbf{K}_a (P \Rightarrow Q)) \Rightarrow \mathbf{K}_a Q.$$

Aus dieser (und einigen anderen Regeln über logische Identitäten) können wir begründen, dass $\mathbf{K}_A(P \vee \neg P)$ eine Tautologie ist; jeder Agent weiß, dass jede Aussage P entweder wahr oder falsch ist. Dagegen ist $(\mathbf{K}_A P) \vee (\mathbf{K}_A \neg P)$ keine Tautologie; im Allgemeinen gibt es viele Aussagen, die dem Agenten weder als wahr noch als falsch bekannt sind.

Es gibt die Auffassung (unter anderem bei Plato), dass Wissen gerechtfertigter wahrer Glaube ist. Das heißt, wenn es wahr ist, wenn du es glaubst und wenn du einen unangreifbaren guten Grund hast, dann weißt du es. Wenn du also etwas weißt, muss es wahr sein. Damit haben wir folgendes Axiom:

$$\mathbf{K}_a P \Rightarrow P.$$

Weiterhin sollten logische Agenten in der Lage sein, das eigene Wissen zu prüfen. Wenn sie etwas wissen, dann wissen sie, dass sie es wissen:

$$\mathbf{K}_a P \Rightarrow \mathbf{K}_a(\mathbf{K}_a P).$$

Ähnliche Axiome können wir für Glaube (oftmals mit B für Belief gekennzeichnet) und andere Modalitäten definieren. Allerdings besteht ein Problem beim Konzept der modalen Logik darin, dass sie **logische Allwissenheit** seitens der Agenten annimmt. Wenn also ein Agent eine Menge von Axiomen kennt, sind ihm auch alle Konsequenzen dieser Axiome bekannt. Selbst für das etwas abstrakte Konzept des Wissens steht das Ganze auf etwas wackeligen Füßen, doch für den Glauben sieht es noch schlechter aus, da man Glaube eher mit einem Verweis auf Dinge, die physisch im Agenten dargestellt werden und nicht nur potenziell ableitbar sind, in Verbindung bringt. Man hat versucht, eine Form von begrenzter Rationalität für Agenten zu definieren; um zu sagen, dass Agenten diejenigen Behauptungen glauben, die sich durch Anwendung von nicht mehr als k Schlussfolgerungsschritten oder nicht mehr als s Sekunden Rechenzeit ableiten lassen. Diese Versuche haben sich allgemein als unzulänglich erwiesen.

12.5 Deduktive Systeme für Kategorien

Kategorien sind die wichtigsten Bausteine für große Wissensrepräsentationsschemas. Dieser Abschnitt beschreibt Systeme, die insbesondere für die Organisation und das Schließen mit Kategorien ausgelegt sind. Es gibt zwei eng verwandte Systemfamilien: **Semantische Netze** bieten grafische Hilfsmittel, um eine Wissensbasis zu visualisieren, und effiziente Algorithmen, um Eigenschaften eines Objektes auf Basis seiner Kategorienzugehörigkeit herzuleiten, und **Beschreibungslogiken** liefern eine formale Sprache, um Kategoriedefinitionen zu konstruieren und zu kombinieren, und effiziente Algorithmen, um Untermengen- und Obermengenbeziehungen zwischen Kategorien zu ermitteln.

12.5.1 Semantische Netze

Charles Peirce schlug 1909 eine grafische Notation für Knoten und Kanten vor, die sogenannten **Existential Graphs** (Existenzgraphen), die er „die Logik der Zukunft“ nannte. Damit begann eine lang andauernde Diskussion zwischen den Fürsprechern der „Logik“ und den Vertretern der „semantischen Netze“. Leider verdeckte diese Diskussion die Tatsache, dass semantische Netze – zumindest solche mit wohl definierter Semantik – eine Art von Logik *sind*. Die Notation, die semantische Netze für bestimmte Arten von Sätzen unterstützen, ist häufig bequemer, aber wenn wir die Aspekte der „Schnittstelle zum Menschen“ über Bord werfen, sind die zugrunde liegenden Konzepte – Objekte, Relationen, Quantifizierung usw. – gleich.

Es gibt viele Varianten semantischer Netze, aber alle sind in der Lage, einzelne Objekte, Objektkategorien und Relationen zwischen Objekten zu repräsentieren. Eine typische grafische Notation zeigt Objekt- oder Kategorienamen in Ovalen oder Kästen an und verknüpft sie mit beschrifteten Kanten. Zum Beispiel hat ▶ Abbildung 12.5 eine *ElementVon*-Verknüpfung zwischen *Mary* und *WeiblichePersonen*, was der logischen Behauptung $Mary \in WeiblichePersonen$ entspricht; analog dazu entspricht die *SchwesterVon*-Verknüpfung zwischen Mary und John der Behauptung *SchwesterVon*(*Mary*, *John*). Wir können Kategorien mithilfe von *UntermengeVon*-Verknüpfungen verbinden usw. Es macht so viel Spaß, Kringel und Pfeile zu zeichnen, dass man sich ganz darin verlieren kann. Wenn wir z.B. wissen, dass Personen weibliche Personen als Mütter haben, können wir dann die *HatMutter*-Verknüpfung zwischen Personen und *WeiblichePersonen* zeichnen? Die Antwort lautet Nein, weil *HatMutter* eine Relation zwischen einer Person und ihrer oder seiner Mutter ist, und Kategorien haben keine Mütter.[7]

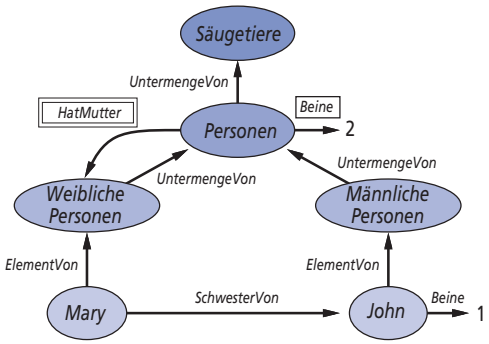


Abbildung 12.5: Ein semantisches Netz mit vier Objekten (John, Mary, 1 und 2) sowie vier Kategorien. Relationen werden durch beschriftete Verknüpfungen dargestellt.

Aus diesem Grund haben wir in Abbildung 12.5 eine spezielle Notation verwendet – die Verknüpfung mit der Beschriftung in einem Kästchen mit doppeltem Rahmen. Diese Verknüpfung behauptet, dass

7 Einige frühere Systeme unterschieden nicht zwischen Eigenschaften von Elementen einer Kategorie und den Eigenschaften der Kategorie als Ganzes. Das kann direkt zu Inkonsistenzen führen, wie von Drew McDermott (1976) in seinem Artikel „Artificial Intelligence Meets Natural Stupidity“ gezeigt. Ein weiteres häufiges Problem war die Verwendung von *IstEin*-Verknüpfungen für Untermengen- und Zugehörigkeitsrelationen in Entsprechung zur Verwendung natürlicher Sprache: „Eine Katze ist ein Säugetier“ und „Fifi ist eine Katze.“ Weitere Informationen zu diesen Aspekten finden Sie in Übung 12.24.

$$\forall x \quad x \in \mathit{Personen} \Rightarrow [\forall y \quad \mathit{HatMutter}(x, y) \Rightarrow y \in \mathit{WeiblichePersonen}].$$

Wir könnten auch behaupten wollen, dass Personen zwei Beine haben, d.h.

$$\forall x \quad x \in \mathit{Personen} \Rightarrow \mathit{Beine}(x, 2).$$

Wie zuvor müssen wir vorsichtig sein, um nicht zu behaupten, dass eine Kategorie Beine hat; die Verknüpfung mit der Beschriftung in einem einfachen Kästchen in Abbildung 12.5 behauptet Eigenschaften jedes Elementes einer Kategorie.

Durch die Notation der semantischen Netze ist es recht komfortabel, das in *Abschnitt 12.2* eingeführte Schließen durch **Vererbung** auszuführen. Weil Mary eine Person ist, erbt sie die Eigenschaft, zwei Beine zu haben. Um also herauszufinden, wie viele Beine Mary hat, folgt der Vererbungsalgorithmus der *ElementVon*-Verknüpfung von Mary zu der Kategorie, der sie angehört, und dann den *UntermengeVon*-Verknüpfungen in der Hierarchie nach oben, bis er eine Kategorie findet, für die es eine Verknüpfung mit in einfach eingerahmten Kästchen befindlicher Beschriftung *Beine* gibt – in diesem Fall die Kategorie *Personen*. Die Einfachheit und Effizienz dieses Inferenzmechanismus im Vergleich zum logischen Theorembeweisen ist einer der wichtigsten Pluspunkte der semantischen Netze.

Die Vererbung wird kompliziert, wenn ein Objekt zu mehr als einer Kategorie gehören oder eine Kategorie eine Untermenge von mehr als einer anderen Kategorie sein kann; man spricht auch von **Mehrfachvererbung**. In diesen Fällen kann der Vererbungsalgorithmus zu zwei oder mehr miteinander in Konflikt stehenden Werten gelangen, die die Abfrage beantworten. Aus diesem Grund wurde die Mehrfachvererbung aus einigen **objektorientierten Programmiersprachen** (OOP) ausgeschlossen, wie z.B. in Java, das die Vererbung als Klassenhierarchie verwendet. In semantischen Netzen ist sie normalerweise zugelassen, aber wir verschieben die zugehörige Diskussion in den *Abschnitt 12.6*.

Der Leser hat vielleicht einen offensichtlichen Nachteil der Notation semantischer Netze im Vergleich zur Logik erster Stufe bemerkt: die Tatsache, dass die Verknüpfungen zwischen den Ovalen nur *binäre* Relationen repräsentieren. Zum Beispiel lässt sich der Satz *Fliegen*(*Shankar*, *NewYork*, *NeuDelhi*, *Gestern*) in einem semantischen Netz nicht direkt ausdrücken. Nichtsdestotrotz *können* wir den Effekt *n*-stelliger Behauptungen erzielen, indem wir die eigentliche Aussage als Ereignis reifizieren, das zu einer geeigneten Ereigniskategorie gehört. ▶ Abbildung 12.6 zeigt die Struktur des semantischen Netzes für dieses spezielle Ereignis. Beachten Sie, dass die Beschränkung auf binäre Relationen die Entwicklung einer umfangreichen Ontologie reifizierter Konzepte erzwingt.

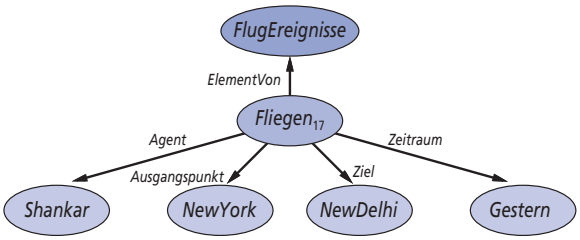


Abbildung 12.6: Ausschnitt aus einem semantischen Netz, der die Repräsentation der logischen Aussage *Fliegen*(*Shankar*, *NewYork*, *NeuDelhi*, *Gestern*) verdeutlicht.

Die Reifikation von Aussagen ermöglicht es, jeden grundlegenden, funktionsfreien atomaren Satz der Logik erster Stufe in der Notation semantischer Netze darzustellen. Bestimmte Arten allquantifizierter Sätze können unter Verwendung inverser Verknüpfungen und Anwendung der einfach und doppelt eingerahmten Kantenbeschriftungen auf Kategorien ausgedrückt werden, aber dennoch sind wir noch weit von der vollständigen Logik erster Stufe entfernt. Negation, Disjunktion, verschachtelte Funktionssymbole und Existenzquantifizierung fehlen. Es ist jedoch *möglich*, die Notation zu erweitern, sodass sie äquivalent zur Logik erster Stufe wird – wie in den existenziellen Graphen von Peirce –, aber damit wird einer der wichtigsten Vorteile der semantischen Netze zunichte gemacht, nämlich die Einfachheit und Transparenz des Inferenzprozesses. Die Entwickler können ein großes Netz aufbauen und behalten dabei stets den Überblick, welche Abfragen effizient sind, weil es (a) einfach ist, die von der Inferenzprozedur durchlaufenen Schritte zu visualisieren, und (b) die Abfragesprache in einigen Fällen so einfach ist, dass überhaupt keine schwierigen Abfragen formuliert werden können. In Fällen, wo die Ausdrucksstärke sich als zu einschränkend erweist, stellen viele Systeme semantischer Netze eine **prozedurale Zuordnung** bereit, um die Lücken zu füllen. Die prozedurale Zuordnung ist eine Technik, wobei eine Abfrage (manchmal auch eine Aussage) über eine bestimmte Relation zu einem Aufruf einer speziellen Prozedur führt, die speziell für diese Relation entwickelt wurde, und nicht zur Ausführung eines allgemeinen Inferenzalgorithmus.

Einer der wichtigsten Aspekte semantischer Netze ist ihre Fähigkeit, **Standardwerte** (Default Values) für Kategorien zu repräsentieren. Ein genauer Blick auf Abbildung 12.5 zeigt, dass John ein Bein hat, obwohl er eine Person ist, und alle Personen zwei Beine haben. In einer streng logischen Wissensbasis wäre das ein Widerspruch, aber in einem semantischen Netz hat die Aussage, dass alle Personen zwei Beine haben, nur einen Standardstatus. Man nimmt also an, eine Person hat zwei Beine, es sei denn, dies wird durch eine spezifischere Information widerlegt. Die Standardsemantik wird durch den Vererbungsalgorithmus in natürlicher Weise erzwungen, weil er vom eigentlichen Objekt aus (in diesem Fall John) die Verknüpfungen nach oben verfolgt und dann abbricht, sobald er einen Wert gefunden hat. Wir sagen, der Standardwert wird durch den spezifischeren Wert **überschrieben**. Beachten Sie, dass wir auch die Standardanzahl der Beine überschreiben könnten, indem wir eine Kategorie *EinbeinigePersonen* einführen, eine Untermenge von *Personen*, der *John* angehört.

Wir können eine streng logische Semantik für das Netz beibehalten, wenn wir sagen, die *Beine*-Behauptung für Personen beinhaltet eine Ausnahme für John:

$$\forall x \quad x \in Personen \wedge x \neq John \Rightarrow Beine(x, 2).$$

Für ein *festes* Netz ist dies semantisch adäquat, aber bei weitem nicht so prägnant wie die Netzwerknotation selbst, wenn es viele Ausnahmen gibt. Für ein Netz, das mit weiteren Behauptungen aktualisiert wird, schlägt dieser Ansatz jedoch fehl – wir wollen eigentlich sagen, dass Personen, die noch unbekannt sind und nur ein Bein haben, ebenfalls Ausnahmen darstellen. *Abschnitt 12.6* beschäftigt sich genauer mit diesem Thema und dem Standardschließen im Allgemeinen.

12.5.2 Beschreibungslogiken

Die Syntax der Logik erster Stufe soll es einfach machen, Dinge über Objekte zu sagen. **Beschreibungslogiken** sind Notationen, die darauf ausgelegt sind, die Beschreibung von Definitionen und Eigenschaften von Kategorien zu erleichtern. Beschreibungslogiksysteme entwickelten sich aus semantischen Netzen. Sie waren eine Antwort auf den Druck, zu formalisieren, was die Netze bedeuten, während die Betonung auf der Taxonomiestruktur als Organisationskonzept beibehalten wurde.

Die wichtigsten Inferenzaufgaben für Beschreibungslogiken sind **Subsumption** (Überprüfung, ob eine Kategorie eine Untermenge einer anderen ist, indem ihre Definitionen verglichen werden) und **Klassifizierung** (Überprüfung, ob ein Objekt zu einer Kategorie gehört). Einige Systeme schließen auch die **Konsistenz** einer Kategoriedefinition ein – ob die Zugehörigkeitskriterien logisch erfüllbar sind.

Die Sprache CLASSIC (Borgida et al., 1989) ist eine typische Beschreibungslogik. Die Syntax von CLASSIC-Beschreibungen ist in ▶ Abbildung 12.7 gezeigt.[8] Um beispielsweise zu sagen, dass Junggesellen nicht verheiratete männliche Erwachsene sind, schreiben wir:

$$Junggeselle = And(NichtVerheiratet, Erwachsen, Männlich).$$

Das Äquivalent in Logik erster Stufe würde wie folgt aussehen:

$$Junggeselle(x) \Leftrightarrow NichtVerheiratet(x) \wedge Erwachsen(x) \wedge Männlich(x).$$

```
Konzept → Thing | KonzeptName
        |  And(Konzept, ...)
        |  All(RollenName, Konzept)
        |  AtLeast(Integer, RollenName)
        |  AtMost(Integer, RollenName)
        |  Fills(RollenName, EinzelName, ...)
        |  SameAs(Pfad, Pfad)
        |  OneOf(EinzelName, ...)
  Pfad  → [RollenName, ...]
```

Abbildung 12.7: Die Syntax von Beschreibungen in einer Untermenge der Sprache CLASSIC.

Beachten Sie, dass zur Beschreibungslogik eine Algebra von Operationen zu Prädikaten gehört, die wir natürlich nicht in Logik erster Stufe ausführen können. Jede Beschreibung in CLASSIC lässt sich auch in einen äquivalenten Satz in Logik erster Stufe übersetzen, doch sind einige Beschreibungen in CLASSIC einfacher. Um beispielsweise die Menge der Männer mit mindestens drei Söhnen zu beschreiben, die alle arbeitslos und mit Doktoren verheiratet sind und höchstens zwei Töchter haben, die Professoren sind und einen Lehrstuhl für Physik oder Mathematik innehaben, schreiben wir:

$$And(Mann, AtLeast(3, Sohn), AtMost(2, Tochter), \\ All(Sohn, And(Arbeitslos, Verheiratet, All(Gattin, Doktor))), \\ All(Tochter, And(Professor, Fills(Lehrstuhl, Physik, Mathematik)))).$$

Wir überlassen es dem Leser als Übung, dies in Logik erster Stufe zu übersetzen.

8 Beachten Sie, dass die Sprache *nicht* erlaubt, einfach auszusagen, dass ein Konzept oder eine Kategorie eine Untermenge eines anderen Konzeptes oder einer anderen Kategorie ist. Diese Vorgehensweise ist beabsichtigt: Die Subsumption zwischen Kategorien muss durch Aspekte der Kategoriebeschreibungen herleitbar sein. Andernfalls fehlt etwas in den Beschreibungen.

Der vielleicht wichtigste Aspekt von Beschreibungslogiken ist ihre Betonung der Behandelbarkeit der Inferenz. Eine Probleminstanz wird gelöst, indem sie zuerst beschrieben und dann gefragt wird, ob sie von einer von mehreren möglichen Lösungskategorien subsumiert wird. In Standardsystemen der Logik erster Stufe ist häufig keine Vorhersage zur Lösungszeit möglich. Oft bleibt es dem Benutzer überlassen, die Repräsentation zu entwerfen, um damit Satzmengen zu umgehen, die aller Voraussicht nach das System mehrere Wochen lang rechnen lassen, bis es zu einer Lösung gelangt. Andererseits zielen Beschreibungslogiken darauf ab, dass sich Subsumptionsüberprüfungen in einer Zeit lösen lassen, die polynomial zur Größe der Beschreibungen ist.[9]

Das klingt grundsätzlich wunderbar – bis man erkennt, dass das nur eine von zwei Konsequenzen haben kann: Entweder können überhaupt keine schwierigen Probleme ausgedrückt werden oder es sind Beschreibungen mit exponentieller Größe dafür erforderlich! Die Handhabbarkeitsresultate bieten jedoch Hinweise darauf, welche Arten von Konstrukten Probleme verursachen könnten, und helfen dem Benutzer damit zu verstehen, wie sich unterschiedliche Repräsentationen verhalten. Beispielsweise fehlen in Beschreibungslogiken üblicherweise *Negation* und *Disjunktion*. Beide zwingen Logiksysteme erster Stufe, eine potenziell exponentielle Fallanalyse zu durchlaufen, um Vollständigkeit sicherzustellen. CLASSIC unterstützt nur eine beschränkte Form der Disjunktion in den Konstrukten *Fills* und *OneOf*, die eine Disjunktion über explizit aufgelistete Individuen erlaubt, aber nicht über Beschreibungen. Bei disjunkten Beschreibungen können verschachtelte Definitionen schnell zu einer exponentiellen Anzahl alternativer Routen führen, durch die eine Kategorie eine andere subsumieren kann.

12.6 Schließen mit Defaultinformation

Im vorigen Abschnitt haben wir ein einfaches Beispiel für eine Behauptung mit Defaultstatus kennengelernt: Menschen haben zwei Beine. Dieser Defaultwert kann durch spezifischere Information überschrieben werden, etwa dass Long John Silver nur ein Bein hat. Wir haben gesehen, dass der Vererbungsmechanismus in semantischen Netzen das Überschreiben von Defaultwerten auf einfache und natürliche Weise implementiert. In diesem Abschnitt betrachten wir die Defaultwerte allgemeiner und achten dabei auf das Verständnis der *Semantik* von Defaultwerten und nicht nur auf die Bereitstellung eines prozeduralen Mechanismus.

12.6.1 Circumscription und Defaultlogik

Wir haben zwei Beispiele vorgestellt, wo offenbar natürliche Inferenzprozesse die Eigenschaft der **Monotonie** der Logik verletzen, die in *Kapitel 7* bewiesen wurde.[10] Dieses Kapitel hat gezeigt, dass eine von allen Elementen einer Kategorie in einem semantischen Netz geerbte Eigenschaft durch spezifischere Informationen für eine Unterkategorie überschrieben werden kann. In *Abschnitt 9.4.5* haben wir gesehen, dass $KB \models \neg\alpha$,

9 CLASSIC unterstützt in der Praxis effiziente Subsumptionstests, aber im ungünstigsten Fall ist die Laufzeit exponentiell.

10 Sie wissen, dass die Monotonie fordert, dass alle logisch konsequenten Sätze logisch konsequent bleiben, nachdem der Wissensbasis neue Sätze hinzugefügt wurden. Das bedeutet, wenn $KB \models \alpha$, dann ist $KB \wedge \beta \models \alpha$.

aber $KB \wedge \alpha \models \alpha$ ist, wenn unter der Annahme einer geschlossenen Welt eine Aussage α in KB nicht erwähnt ist.

Eine einfache Introspektion zeigt, dass diese Fehler der Monotonie im alltäglichen Schließen häufig vorkommen. Es scheint, dass die Menschen häufig zu Schlüssen „springen“. Sieht man beispielsweise, dass ein Auto auf der Straße parkt, glaubt man gerne, dass es vier Räder hat, auch wenn man nur drei sieht. Die Wahrscheinlichkeitstheorie kann natürlich einen Schluss bereitstellen, dass das vierte Rad mit sehr hoher Wahrscheinlichkeit existiert, aber für die meisten Leute kommt die Möglichkeit, dass das Auto nicht vier Räder hat, *gar nicht in Betracht, bis sich irgendein neuer Befund ergibt*. Es scheint also, dass der Vier-Räder-Schluss *standardmäßig* erreicht wird, wenn es keinen Grund gibt, ihn anzuzweifeln. Wenn sich neue Befunde ergeben – wenn man z.B. sieht, dass der Eigentümer des Autos ein Rad trägt und das Auto aufgebockt ist –, kann der Schluss zurückgenommen werden. Diese Art des Schließens weist eine sogenannte **Nichtmonotonie** auf, weil die Menge der Glauben nicht monoton über die Zeit anwächst, wenn sich neue Befunde ergeben. Die **nichtmonotonen Logiken** wurden mit modifizierten Konzepten von Wahrheit und logischer Konsequenz abgeleitet, um ein solches Verhalten zu berücksichtigen. Wir betrachten zwei solcher Logiken, die ausführlich untersucht wurden: Umschreibung (Circumscription) und Defaultlogik.

Die **Circumscription** (Umschreibung) kann man sich auch als leistungsfähigere und genauere Version der Annahme der geschlossenen Welt vorstellen. Der Gedanke dabei ist, bestimmte Prädikate zu spezifizieren, von denen man annimmt, dass sie „so falsch wie möglich“ sind – d.h. falsch für jedes Objekt außer für diejenigen, von denen man weiß, dass sie wahr sind. Angenommen, wir wollen die Defaultregel formulieren, dass Vögel fliegen. Wir würden ein Prädikat $Anormal_1(x)$ einführen und schreiben:

$$Vogel(x) \wedge \neg Anormal1(x) \Rightarrow Fliegt(x).$$

Wenn wir sagen, dass $Anormal_1$ zu **umschreiben** ist, darf ein umschreibender Schließer $\neg Anormal_1(x)$ annehmen, außer wenn $Anormal_1(x)$ als wahr bekannt ist. Damit kann der Schluss $Fliegt(Tweety)$ aus der Prämisse $Vogel(Tweety)$ gezogen werden, doch gilt dieser Schluss nicht mehr, wenn $Anormal_1(Tweety)$ festgestellt wird.

Die Circumscription kann als Beispiel für eine **Modell-Präferenz-Logik** betrachtet werden. In derartigen Logiken ist ein Satz logisch folgerbar (mit Defaultstatus), wenn er in allen *bevorzugten* Modellen der Wissensbasis wahr ist, im Gegensatz zur Forderung der Wahrheit in *allen* Modellen in der klassischen Logik. Für die Circumscription wird ein Modell einem anderen gegenüber bevorzugt, wenn es weniger anormale Objekte enthält.[11] Wir wollen betrachten, wie sich diese Idee im Kontext der Mehrfachvererbung in semantischen Netzen verhält. Als Standardbeispiel, für das die Mehrfachvererbung problematisch ist, wird der sogenannte „Nixon-Diamant“ herangezogen. Er entstand aus der Beobachtung, dass Nixon sowohl ein Quäker (und damit normalerweise Pazifist) als auch Republikaner (und damit normalerweise kein Pazifist) war. Wir können dies wie folgt schreiben:

11 Für die Annahme der geschlossenen Welt wird ein Modell einem anderen gegenüber bevorzugt, wenn es weniger wahre Atome hat – d.h., bevorzugte Modelle sind **minimale** Modelle. Es gibt eine natürliche Verbindung zwischen der Annahme der geschlossenen Welt und Wissensbasen mit definiten Klauseln, weil der durch die Vorwärtsverkettung in solchen Wissensbasen erreichte Fixpunkt das eindeutige minimale Modell ist. (Mehr dazu finden Sie in *Abschnitt 7.5.4*.)

$$Republikaner(Nixon) \wedge Quäker(Nixon).$$
$$Republikaner(x) \wedge \neg Anormal2(x) \Rightarrow \neg Pazifist(x).$$
$$Quäker(x) \wedge \neg Anormal3(x) \Rightarrow Pazifist(x).$$

Wenn wir $Anormal_2$ und $Anormal_3$ umschreiben, gibt es zwei bevorzugte Modelle: eines, in dem $Anormal_2(Nixon)$ und $Pazifist(Nixon)$ gelten, und eines, in dem $Anormal_3(Nixon)$ und $\neg Pazifist(Nixon)$ gelten. Der umschreibende Schließer bleibt also dahingehend agnostisch, ob Nixon ein Pazifist ist. Wenn wir darüber hinaus behaupten wollen, dass religiöser Glauben Priorität gegenüber politischem Glauben besitzt, können wir einen Formalismus verwenden, der auch als **priorisierte Circumscription** bezeichnet wird, um Modelle zu bevorzugen, in denen $Anormal_3$ minimiert ist.

Die **Defaultlogik** ist ein Formalismus, mit dem **Defaultregeln** geschrieben werden können, um bedingte nicht monotone Schlüsse zu ziehen. Eine Defaultregel sieht wie folgt aus:

$$Vogel(x) : Fliegt(x) \,/\, Fliegt(x).$$

Diese Regel bedeutet: Wenn $Vogel(x)$ wahr und $Fliegt(x)$ konsistent mit der Wissensbasis ist, dann kann $Fliegt(x)$ als Default geschlossen werden. Im Allgemeinen hat eine Defaultregel die Form

$$P : J_1, ..., J_n \,/\, C.$$

Dabei ist P die Vorbedingung, C ist der Schluss und J_i sind die Rechtfertigungen – wenn eine davon als falsch bewiesen werden kann, kann der Schluss nicht gezogen werden. Jede Variable, die in J_i oder C erscheint, muss auch in P erscheinen. Das Beispiel des Nixon-Diamanten kann in Defaultlogik mit einem Fakt und zwei Defaultregeln ausgedrückt werden:

$$Republikaner(Nixon) \wedge Quäker(Nixon).$$
$$Republikaner(x) : \neg Pazifist(x) \,/\, \neg Pazifist(x).$$
$$Quäker(x) : \; Pazifist(x) \,/\, Pazifist(x).$$

Um die Bedeutung der Defaultregeln zu interpretieren, definieren wir das Konzept einer **Erweiterung** einer Defaulttheorie als maximale Menge der Konsequenzen der Theorie. Das bedeutet, eine Erweiterung S besteht aus den ursprünglich bekannten Fakten und einer Menge von Schlüssen aus den Defaultregeln, sodass keine zusätzlichen Schlüsse aus S gezogen werden können und die Rechtfertigungen für jeden Defaultschluss in S mit S konsistent sind. Wie es im Fall der bevorzugten Modelle bei der Circumscription ist, haben wir zwei mögliche Erweiterungen für den Nixon-Diamanten: eine, in der er ein Pazifist ist, und eine, in der er das nicht ist. Es gibt Prioritätsschemas, in denen bestimmte Defaultregeln Priorität gegenüber anderen erhalten können und es damit erlauben, bestimmte Mehrdeutigkeiten aufzulösen.

Seit 1980, als die nicht monotonen Logiken zum ersten Mal vorgestellt wurden, sind sehr viele Fortschritte im Hinblick auf das Verständnis ihrer mathematischen Eigenschaften zu verzeichnen. Es gibt jedoch noch immer ungelöste Fragen. Wenn beispielsweise „Autos haben vier Räder“ falsch ist, was bedeutet es dann, diese Aussage in einer Wissensbasis zu haben? Welche Regelmenge ist sinnvoll? Wenn wir nicht für jede Regel separat entscheiden können, ob sie in unsere Wissensbasis gehört, haben wir ein ernsthaftes Problem der Nichtmodularität. Und schließlich: Wie können Glauben, die Defaultzustände haben, für die Entscheidungsfindung genutzt werden? Das ist wahrscheinlich das schwierigste Problem für das Schließen mit Defaultwerten. Entscheidungen beinhalten häufig Kompro-

misse; deshalb muss man die Glaubens*stärken* in den Ergebnissen unterschiedlicher Aktionen und die Kosten für das Treffen einer falschen Entscheidung vergleichen. In Fällen, wo gleichartige Entscheidungen wiederholt getroffen werden, ist es möglich, Defaultregeln als Aussagen für die „Schwellenwahrscheinlichkeit“ zu interpretieren. Zum Beispiel bedeutet die Defaultregel „Meine Bremsen sind immer OK“ in Wirklichkeit: „Die Wahrscheinlichkeit, dass meine Bremsen OK sind, ist, falls es keine weiteren Informationen gibt, ausreichend hoch, sodass es für mich die optimale Entscheidung ist, zu fahren, ohne sie zuvor zu überprüfen.“ Wenn sich der Entscheidungskontext ändert – wenn man beispielsweise einen schwer beladenen Lastwagen eine steile Bergstraße nach unten lenkt –, wird die Defaultregel plötzlich ungeeignet, auch wenn es kein neues Anzeichen gibt, das darauf hinweist, dass die Bremsen defekt sein könnten. Diese Betrachtungen haben dazu geführt, dass einige Forscher überlegt haben, wie sie das Schließen mit Defaultwerten in die Wahrscheinlichkeitstheorie einbetten könnten.

12.6.2 Truth Maintenance Systems (Wahrheitserhaltungssysteme)

Es wurde gezeigt, dass viele der durch ein Wissensrepräsentationssystem gezogenen Inferenzen nur einen Defaultstatus haben und nicht absolut sicher sind. Einige dieser abgeleiteten Fakten erweisen sich unvermeidbar als falsch und müssen angesichts neuer Informationen zurückgezogen werden. Dieser Prozess wird auch als **Glaubensrevision**[12] (**Belief Revision**) bezeichnet. Angenommen, eine Wissensbasis KB enthält einen Satz P – vielleicht ein Defaultschluss, der durch einen Vorwärtsverkettungsalgorithmus aufgezeichnet wurde, oder vielleicht nur eine fehlerhafte Behauptung – und wir wollen TELL(KB, $\neg P$) ausführen. Um zu vermeiden, dass w ir einen Widerspruch erzeugen, müssen wir zuerst RETRACT(KB, P) ausführen. Das hört sich ganz einfach an. Es treten jedoch Probleme auf, wenn *zusätzliche* Sätze von P abgeleitet und in der Wissensbasis behauptet wurden. Beispielsweise hätte die Implikation $P \Rightarrow Q$ verwendet werden können, um Q hinzuzufügen. Die offensichtliche „Lösung“ – alle Sätze zurückzunehmen, die von P abgeleitet wurden – schlägt fehl, weil solche Sätze neben P auch andere Rechtfertigungen haben könnten. Wenn beispielsweise R und $R \Rightarrow Q$ ebenfalls in der Wissensbasis enthalten sind, dann sollte Q überhaupt nicht entfernt werden. **Truth Maintenance Systems** (**TMS**), also „Wahrheitserhaltungssysteme“, sind darauf ausgelegt, genau diese Komplikationen zu kompensieren.

Ein sehr einfacher Ansatz für diese Wahrheitserhaltung ist die Aufzeichnung der Reihenfolge, in der die Sätze der Wissensbasis mitgeteilt wurden, indem man sie von P_1 bis P_n nummeriert. Wenn der Aufruf von RETRACT(KB, P_i) erfolgt, kehrt das System in den Zustand zurück, den es unmittelbar vor dem Hinzufügen von P_i hatte, und entfernt dabei P_i und alle von P_i abgeleiteten Inferenzen. Die Sätze P_{i+1} bis P_n können dann wieder hinzugefügt werden. Das ist einfach und garantiert, dass die Wissensbasis konsistent bleibt, aber das Zurückziehen von P_i macht das Zurückziehen und erneutes Behaupten von $n - i$ Sätzen sowie das Rückgängigmachen aller von diesen Sätzen hergeleiteten Inferenzen erforderlich. Für Systeme, in denen viele Fakten hinzugefügt werden – wie etwa große kommerzielle Datenbanken –, ist das unpraktisch.

12 Ihr wird häufig das **Glaubens-Update** gegenübergestellt, das auftritt, wenn eine Wissensbasis überarbeitet wird, um eine Änderung in der Welt zu reflektieren und nicht neue Informationen über eine feststehende Welt auszudrücken. Das Glaubens-Update kombiniert die Glaubensrevision mit einem Schließen über Zeit und Änderung; außerdem ist es verwandt mit dem Prozess der **Filterung**, der in *Kapitel 15* beschrieben wird.

Ein effizienterer Ansatz ist das **JTMS** (Justification-Based Truth Maintenance System), das auf Rechtfertigungen basiert. In einem JTMS wird jeder Satz der Wissensbasis mit einer **Rechtfertigung** (**Justification**) kommentiert, die die Menge der Sätze enthält, von denen er abgeleitet wurde. Enthält die Wissensbasis beispielsweise bereits $P \Rightarrow Q$, dann bewirkt TELL(P), dass Q mit der Rechtfertigung $\{P, P \Rightarrow Q\}$ eingefügt wird. Im Allgemeinen kann ein Satz beliebig viele Rechtfertigungen haben. Rechtfertigungen machen das Zurückziehen effizient. Für den Aufruf RETRACT(P) löscht das JTMS genau die Sätze, für die P ein Element jeder Rechtfertigung ist. Wenn also ein Satz Q nur die Rechtfertigung $\{P, P \Rightarrow Q\}$ hat, wird er entfernt; selbst mit der zusätzlichen Rechtfertigung $\{P, P \vee R \Rightarrow Q\}$ wird er entfernt; doch wenn er außerdem die Rechtfertigung $\{R, P \vee \Rightarrow Q\}$ hat, bleibt er verschont. Auf diese Weise hängt die erforderliche Zeit für das Zurückziehen von P nur von der Anzahl der von P abgeleiteten Sätze und nicht von der Anzahl anderer Sätze ab, die hinzugekommen sind, seit P in die Wissensbasis eingefügt wurde.

Das JTMS geht davon aus, dass Sätze, die einmal in Betracht gezogen wurden, wahrscheinlich wieder in Betracht gezogen werden. Anstatt also einen Satz völlig aus der Wissensbasis zu löschen, wenn er alle Rechtfertigungen verloren hat, markiert es einfach den Satz, um anzuzeigen, dass er sich *nicht in* der Wissensbasis befindet. Wenn eine nachfolgende Behauptung eine der Rechtfertigungen wiederherstellt, markieren wir den Satz, dass er wieder *in* der Wissensbasis ist. Auf diese Weise behält das JTMS alle Referenzketten bei, die es verwendet, und muss Sätze nicht neu herleiten, wenn eine Rechtfertigung wieder gültig wird.

Neben dem Zurückziehen falscher Informationen können TMS auch verwendet werden, um die Analyse mehrerer hypothetischer Situationen zu beschleunigen. Nehmen wir beispielsweise an, das Rumänische Olympische Komitee suche Städte für die Wettkämpfe in Schwimmen, Leichtathletik und Reiten für die Spiele 2048, die in Rumänien ausgetragen werden. Die erste Hypothese sei *Stadt*(*Schwimmen*, *Pitesti*), *Stadt*(*Leichtathletik*, *Bukarest*) und *Stadt*(*Reiten*, *Arad*). Es muss sehr viel Inferenzarbeit geleistet werden, um die logischen Konsequenzen und damit den Nutzen dieser Auswahl zu ermitteln. Wenn wir stattdessen *Stadt*(*Leichtathletik*, *Sibiu*) in Betracht ziehen wollen, muss das TMS seine Arbeit nicht ganz von vorn beginnen. Stattdessen ziehen wir einfach *Stadt*(*Leichtathletik*, *Bukarest*) zurück und behaupten *Stadt*(*Leichtathletik*, *Sibiu*) – das TMS kümmert sich um die erforderlichen Überarbeitungen. Inferenzketten, die durch die Auswahl von Bukarest erzeugt werden, können für Sibiu wiederverwendet werden, vorausgesetzt, die Schlüsse bleiben dieselben.

Ein auf Annahmen basierendes TMS, ein **ATMS** (Assumption-based Truth Maintenance System), ist darauf ausgelegt, diese Art der Kontextwechsel zwischen hypothetischen Welten besonders effizient zu machen. In einem JTMS erlaubt es die Verwaltung der Rechtfertigungen, schnell von einem Zustand in einen anderen zu wechseln, indem einige Dinge zurückgezogen und einige neue Behauptungen eingetragen werden, wobei aber jederzeit nur ein Zustand repräsentiert wird. Ein ATMS repräsentiert *alle* Zustände, die je in Betracht gezogen wurden, gleichzeitig. Während JTMS einfach jeden Satz markiert, der sich *in* oder *nicht in* der Wissensbasis befindet, notiert ein ATMS für jeden Satz, welche Annahmen dafür sorgen würden, dass der Satz wahr ist. Mit anderen Worten, jeder Satz hat eine Markierung, die aus einer Menge von Annahmemengen besteht. Der Satz gilt, falls alle Annahmen in einer der Annahmemengen gelten.

Wahrheitsverwaltungssysteme unterstützen auch einen Mechanismus für die Erzeugung von **Erklärungen**. Aus technischer Sicht ist eine Erklärung eines Satzes *P* eine Menge von Sätzen *E*, sodass *P* eine logische Konsequenz aus *E* ist. Wenn man bereits weiß, dass die Sätze in *E* wahr sind, stellt *E* einfach eine ausreichende Grundlage für den Beweis zur Verfügung, dass *P* der Fall sein muss. Erklärungen können aber auch **Annahmen** beinhalten – Sätze, von denen man nicht weiß, ob sie wahr sind, die aber für den Beweis von *P* ausreichend wären, wenn sie wahr wären. Beispielsweise hat man vielleicht nicht ausreichend viele Informationen, um zu beweisen, dass das Auto nicht anspringt, aber eine sinnvolle Erklärung könnte die Annahme enthalten, dass die Batterie leer ist. Kombiniert man dies mit dem Wissen, wie Autos funktionieren, erklärt sich das beobachtete Fehlverhalten. In den meisten Fällen bevorzugen wir eine minimale Erklärung E, d.h., es gibt keine echte Untermenge von E, die ebenfalls eine Erklärung darstellt. Ein ATMS kann Erklärungen für das Problem „Auto startet nicht“ erzeugen, indem es Annahmen trifft (wie etwa „Benzin im Auto“ oder „Batterie leer“) – und zwar in jeder von uns gewünschten Reihenfolge, auch wenn einige Annahmen widersprüchlich sind. Anschließend betrachten wir die Beschriftung für den Satz „Auto startet nicht“, um die Mengen der Annahmen auszulesen, die den Satz rechtfertigen würden.

Die genauen Algorithmen für die Implementierung von Wahrheitsverwaltungssystemen sind etwas kompliziert und wir werden hier nicht genauer darauf eingehen. Die Komplexität des Wahrheitserhaltungsproblems ist mindestens so groß wie die der aussagenlogischen Inferenz – d.h. NP-hart. Aus diesem Grund sollten Sie nicht erwarten, dass die Wahrheitserhaltung ein Allheilmittel ist. Bei sorgfältiger Verwendung kann ein TMS jedoch eine wesentliche Verbesserung darstellen, wie ein logisches System mit komplexen Umgebungen und Hypothesen zurechtkommt.

12.7 Die Internet-Shopping-Welt

In diesem letzten Abschnitt führen wir alles zusammen, was wir gelernt haben, um Wissen für einen Shopping-Suchagenten zu kodieren, der einem Käufer hilft, Produktangebote im Internet zu finden. Der Shopping-Agent erhält vom Käufer eine Produktbeschreibung und hat die Aufgabe, eine Liste von Webseiten zu erstellen, auf denen ein solches Produkt zum Verkauf angeboten wird, und ein Ranking nach den besten Angeboten vorzunehmen. In einigen Fällen ist die Produktbeschreibung des Käufers exakt, wie etwa *Canon Rebel XTi Digitalkamera*, und die Aufgabe ist dann, die Geschäfte mit dem besten Angebot zu finden. In anderen Fällen ist die Beschreibung nicht vollständig, wie etwa *Digitalkamera für unter 300* €, und der Agent muss verschiedene Produkte vergleichen.

Die Umgebung des Shopping-Agenten ist das gesamte World Wide Web in seiner vollen Komplexität – nicht eine simulierte Spielzeugumgebung. Die Wahrnehmungen des Agenten sind Webseiten, aber während ein menschlicher Webbenutzer Seiten als Array von Pixeln auf einem Bildschirm sieht, nimmt der Shopping-Agent eine Seite als Zeichenfolge wahr, die aus normalen Wörtern besteht, zwischen denen sich immer wieder Formatierungsbefehle der Seitenbeschreibungssprache HTML befinden. ▶ Abbildung 12.8 zeigt eine Webseite und eine entsprechende HTML-Zeichenfolge. Das Wahrnehmungsproblem für den Shopping-Agenten besteht unter anderem darin, sinnvolle Informationen aus Wahrnehmungen dieser Art zu extrahieren.


```
<h1>Generisches Online-Kaufhaus</h1>
<i>Wählen</i> Sie aus unserer gut sortierten Produktpalette:
<ul>
<li> <a href="http://gen-store.com/compu">Computer</a>
<li> <a href="http://gen-store.com/camer">Kameras</a>
<li> <a href="http://gen-store.com/books">Bücher</a>
<li> <a href="http://gen-store.com/video">Videos</a>
<li> <a href="http://gen-store.com/music">Musik</a>
</ul>
```

Abbildung 12.8: Eine Webseite aus einem generischen Online-Kaufhaus, wie sie von einem menschlichen Benutzer eines Browsers wahrgenommen wird (oben), und die entsprechende HTML-Zeichenfolge, wie sie der Browser oder der Shopping-Agent wahrnehmen (unten). In HTML sind Zeichen zwischen < und > Formatierungsbefehle, die angeben, wie die Seite angezeigt werden soll. Zum Beispiel bedeutet die Zeichenfolge `<i>Wählen</i>`, dass nachfolgend kursive Schrift verwendet, das Wort *Wählen* angezeigt und der kursive Schriftmodus wieder verlassen werden soll. Ein Seitenname wie etwa *http://example.com/books* wird als ***URL*** (***Uniform Resource Locator***) bezeichnet. Die Auszeichnung (das sogenannte Markup) `<a href = "url">Bücher</a>` bedeutet, dass ein Hyperlink zu *url* mit dem ***Ankertext*** *Bücher* erzeugt wird.

Offensichtlich ist die Wahrnehmung auf Webseiten einfacher als beispielsweise die Wahrnehmung bei einer Taxifahrt in Kairo. Nichtsdestotrotz gibt es Komplikationen in dieser Wahrnehmungsaufgabe im Internet. Die in Abbildung 12.8 gezeigte Webseite ist sehr einfach – im Vergleich zu den realen Shopping-Sites, die CSS, Cookies, Java, JavaScript, Flash, Protokolle zum Ausschließen von Bots, falsch formatiertes HTML, Sounddateien und Filme sowie Text, der nur als Teil eines JPEG-Bildes erscheint, enthalten können. Ein Agent, der mit *allen* diesen Dingen im Internet zurechtkommt, ist fast so kompliziert wie ein Roboter, der sich in der realen Welt bewegen kann. Wir konzentrieren uns hier auf einen einfachen Agenten, der einen Großteil dieser Komplikationen ignoriert.

Die erste Aufgabe des Agenten ist es, Produktangebote zu finden, die für eine Abfrage relevant sind. Wenn die Abfrage zum Beispiel „Laptops“ lautet, ist eine Seite mit einer Besprechung des neuesten High-End-Laptops relevant, doch wenn es hier keine Möglichkeit gibt, das Produkt zu kaufen, handelt es sich nicht um ein Angebot. Zunächst lässt sich feststellen, dass eine Seite ein Angebot ist, wenn sie die Wörter „kaufen“, „Preis“ oder „In den Warenkorb“ innerhalb eines HTML-Links oder Formulars auf der Seite enthält. Wenn zum Beispiel auf der Seite eine Zeichenfolge der Form `<a ... In den Warenkorb ... </a>` vorkommt, handelt es sich um ein Angebot. Dies ließe sich zwar in Logik erster Stufe darstellen, doch ist es einfacher, dies in Programmcode zu kodieren. *Abschnitt 22.4* zeigt, wie sich anspruchsvollere Informationen extrahieren lassen.

12.7.1 Links folgen

Die Strategie besteht darin, auf der Homepage eines Online-Stores zu beginnen und alle Seiten zu betrachten, die durch das Verfolgen relevanter Links[13] erreicht werden können. Der Agent kennt mehrere Geschäfte, z.B.:

$$Amazon \in OnlineStores \wedge Homepage(Amazon, \text{"amazon.com"}).$$
$$Ebay \in OnlineStores \wedge Homepage(Ebay, \text{"ebay.com"}).$$
$$ExampleStore \in OnlineStores \wedge Homepage(ExampleStore, \text{"example.com"}).$$

Diese Geschäfte ordnen ihre Waren in Produktkategorien ein und stellen auf ihrer Homepage Links zu den Hauptkategorien bereit. Untergeordnete Kategorien werden erreicht, indem man einer Kette relevanter Links folgt, und irgendwann erreicht man Angebote. Mit anderen Worten ist eine Seite für die Abfrage relevant, wenn sie über eine Kette von null oder mehr relevanten Kategorielinks von der Homepage eines Geschäftes aus erreicht werden kann und dann ein oder mehrere Links zu Produktangeboten führen. Relevanz können wir folgendermaßen definieren:

$$\begin{aligned} &Relevant(seite, abfrage) \Leftrightarrow \\ &\quad \exists geschäft, home \;\; geschäft \in OnlineStores \wedge Homepage(geschäft, home) \\ &\quad \wedge \exists url, url_2 \;\; RelevanteKette(home, url_2, abfrage) \wedge Link(url_2, url) \\ &\qquad \wedge seite = Inhalt(url). \end{aligned}$$

Hier bedeutet das Prädikat $Link(von, nach)$, dass es einen Hyperlink von der *Von*-URL zu der *Zu*-URL gibt. Um zu definieren, was als *RelevanteKette* zählt, müssen wir nicht nur jedem der alten Hyperlinks folgen, sondern auch den Links, deren zugeordneter Ankertext darauf hinweist, dass der Link relevant für die Produktabfrage ist. Dazu verwenden wir $LinkText(von, nach, text)$, d.h., es gibt einen Link zwischen *von* und *nach* mit *text* als Ankertext. Eine Linkkette zwischen zwei URLs, *anfang* und *ende*, ist relevant für eine Beschreibung *d*, wenn der Ankertext eines Links ein relevanter Kategoriename für *d* ist. Die Existenz der eigentlichen Kette wird durch eine rekursive Definition ermittelt, mit der leeren Kette ($anfang = ende$) als Basisfall:

$$\begin{aligned} &RelevanteKette(anfang, ende, abfrage) \Leftrightarrow (anfang = ende) \\ &\quad \vee (\exists u, text \;\; LinkText(anfang, u, text) \wedge RelevanterKategorieName(abfrage, text) \\ &\qquad \wedge RelevanteKette(u, ende, abfrage)) \end{aligned}$$

Jetzt müssen wir definieren, was es bedeutet, dass *text* ein *RelevanterKategorieName* für *abfrage* ist. Zuerst müssen wir Zeichenfolgen mit den von ihnen bezeichneten Kategorien in Verbindung bringen. Dazu verwenden wir das Prädikat $Name(s, c)$, d.h., die Zeichenfolge *s* ist ein Name für die Kategorie *c* – z.B. könnten wir behaupten $Name(\text{„}Laptops\text{“}, LaptopComputer)$. ▶ Abbildung 12.9(b) zeigt einige weitere Beispiele für das Prädikat *Name*. Als Nächstes definieren wir die Relevanz. Angenommen, *abfrage* lautet „Laptops“. Dann ist $RelevanterKategorieName(abfrage, text)$ wahr, wenn eine der folgenden Aussagen gilt:

- Der *text* und die *abfrage* bezeichnen dieselbe Kategorie – z.B. „Notebooks“ und „Laptops“.
- Der *text* bezeichnet eine übergeordnete Kategorie, wie etwa „Computer“.
- Der *text* bezeichnet eine untergeordnete Kategorie, wie etwa „ultraleichte Notebooks“.

13 Eine Alternative zur Strategie der Verfolgung von Links ist die Verwendung einer Internetsuchmaschine; die Technologie hinter der Internetsuche, die Informationsermittlung, ist in *Abschnitt 22.3* genauer beschrieben.

Bücher ⊂ *Produkte*
MusikAufzeichnungen ⊂ *Produkte*
 MusikCDs ⊂ *MusikAufzeichnungen*
Elektronik ⊂ *Produkte*
 DigitalKameras ⊂ *Elektronik*
 StereoAnlage ⊂ *Elektronik*
 Computer ⊂ *Elektronik*
  DesktopComputer ⊂ *Computer*
  LaptopComputer ⊂ *Computer*
  ...

a

Name(*"Bücher"*, *Bücher*)
Name(*"Musik"*, *MusikAufzeichnungen*)
 Name(*"CDs"*, *MusikCDs*)
Name(*"Elektronik"*, *Elektronik*)
 Name(*"digitale Kameras"*, *DigitalKameras*)
 Name(*"Stereos"*, *StereoAnlage*)
 Name(*"Computer"*, *Computer*)
  Name(*"Desktops"*, *DesktopComputer*)
  Name(*"Laptops"*, *LaptopComputer*)
  Name(*"Notebooks"*, *LaptopComputer*)
  ...

b

Abbildung 12.9: (a) Taxonomie von Produktkategorien; (b) Verweiswörter auf diese Kategorien.

Die logische Definition von *RelevanterKategorieName* lautet:

$$RelevanterKategorieName(abfrage, text) \Leftrightarrow \exists c_1, c_2 \quad Name(abfrage, c_1) \wedge Name(text, c_2) \wedge (c_1 \subseteq c_2 \vee c_2 \subseteq c_1). \qquad (12.1)$$

Andernfalls ist der Ankertext irrelevant, weil er eine Kategorie außerhalb dieser Palette anzeigt, wie etwa „Bekleidung“ oder „Rasen & Garten“.

Um den relevanten Links folgen zu können, braucht man eine umfangreiche Hierarchie von Produktkategorien. Der obere Teil dieser Hierarchie könnte wie in ▶ Abbildung 12.9(a) gezeigt aussehen. Es ist nicht machbar, *alle* möglichen Shopping-Kategorien aufzulisten, weil ein Käufer immer wieder mit neuen Bedürfnissen aufwartet und die Hersteller immer wieder neue Produkte auf den Markt bringen, um diese zu erfüllen (zum Beispiel elektrische Kniescheibenwärmer?). Nichtsdestotrotz ist eine Ontologie von etwa tausend Kategorien schon ein sehr praktisches Werkzeug für die meisten Käufer.

Neben der eigentlichen Produkthierarchie brauchen wir auch ein umfangreiches Vokabular mit Namen für Kategorien. Das Leben wäre viel einfacher, gäbe es eine 1:1-Entsprechung zwischen den Kategorien und den Zeichenfolgen, mit denen sie benannt sind. Wir haben das Problem der **Synonymie** bereits kennengelernt – zwei Namen für dieselbe Kategorie, wie z.B. „Laptop Computer“ und „Laptops“. Außerdem gibt es das Problem der **Mehrdeutigkeit** – ein Name für zwei oder mehr unterschiedliche Kategorien. Wenn wir beispielsweise den Satz:

$$Name(\textit{"CDs"}, CertificatesOfDeposit)$$

in die Wissensbasis gemäß Abbildung 12.9(b) einfügen, bezeichnet „CDs“ zwei verschiedene Kategorien.

Aufgrund von Synonymie und Mehrdeutigkeit muss der Agent gegebenenfalls sehr viel mehr Pfade verfolgen und es wird manchmal schwierig, festzustellen, ob eine bestimmte Seite wirklich relevant ist. Schwerwiegender ist das Problem, dass die Bereiche der vom Benutzer eingegebenen Beschreibungen und der von einem Geschäft verwendeten Kategorienamen sehr breit sein können. Beispielsweise könnte der Link „Laptop“ heißen, während in der Wissensbasis nur „Laptops“ eingetragen ist; oder der Benutzer fragt nach „einem Computer, den ich auf dem Klapptisch eines Sitzes der Economy-Klasse in einer Boeing 737 unterbringen kann“. Es ist unmöglich, im Voraus alle Möglichkeiten aufzulisten, wie eine Kategorie bezeichnet werden kann. Der Agent muss deshalb in der Lage sein, in einigen Fällen zusätzliche Schlüsse zu ziehen, um festzustellen, ob die Relation

Name gilt. Im ungünstigsten Fall ist dafür ein vollständiges Verständnis der natürlichen Sprache erforderlich, ein Thema, das wir erst in *Kapitel 22* besprechen werden. In der Praxis sind einige einfache Regeln ausreichend – wenn man beispielsweise erlaubt, dass „Laptop“ auch eine Kategorie „Laptops“ anspricht. In Übung 12.11 werden Sie eine Menge solcher Regeln entwickeln, nachdem Sie sich genauer mit Online-Stores beschäftigt haben.

Sind wir nun mit den logischen Definitionen aus den vorigen Absätzen und geeigneten Wissensbasen mit Produktkategorien und Namenskonventionen in der Lage, einen Inferenzalgorithmus anzuwenden, um eine Menge relevanter Angebote für unsere Abfrage zu ermitteln? Noch nicht ganz! Das fehlende Element ist die Funktion *Inhalt*(*url*), die auf die HTML-Seite an der vorgegebenen URL verweist. Der Agent hat nicht den Seiteninhalt jeder URL in seiner Wissensbasis und er hat auch keine expliziten Regeln für den Schluss, wie diese Inhalte aussehen könnten. Stattdessen können wir dafür sorgen, dass eine geeignete HTTP-Prozedur ausgeführt wird, wenn ein Unterziel die Funktion *Inhalt* beinhaltet. Auf diese Weise erhält der Inferenzmechanismus den Eindruck, als befinde sich das gesamte Web in der Wissensbasis. Dies ist ein Beispiel für eine allgemeine Technik namens **prozedurale Zuordnung**, wobei bestimmte Prädikate und Funktionen von speziellen Methoden verarbeitet werden können.

12.7.2 Angebote vergleichen

Angenommen, die Schlussprozesse des vorigen Abschnittes haben mehrere Angebotsseiten für unsere Abfrage „Laptops“ erbracht. Um diese Angebote zu vergleichen, muss der Agent die relevanten Informationen aus den Angebotsseiten extrahieren – Preis, Geschwindigkeit, Plattengröße, Gewicht usw. Das kann für reale Webseiten eine schwierige Aufgabe sein – aus all den zuvor erwähnten Gründen. Eine allgemeine Methode, mit diesem Problem zurechtzukommen, ist die Verwendung von Programmen, die auch als **Wrapper** (Hüllen) bezeichnet werden, um Informationen von einer Seite zu extrahieren. Die Technologie für diese Art der Informationsextrahierung ist in *Abschnitt 22.4* beschrieben. Hier wollen wir einfach annehmen, dass Wrapper existieren und dass sie Zusicherungen in die Wissensbasis eintragen, wenn man ihnen eine Seite und eine Wissensbasis übergibt. Normalerweise wird eine Hierarchie von Wrappern auf eine Seite angewendet: ein sehr allgemeiner Wrapper, der Daten und Preise extrahiert, ein spezifischerer, der Attribute für computerbezogene Produkte extrahiert, und gegebenenfalls ein Site-spezifischer, der das Format eines bestimmten Geschäftes kennt. Für eine Seite der Site *example.com* mit dem Text

```
IBM     ThinkBook     970.     Unser Preis: $399.00
```

gefolgt von verschiedenen technischen Spezifikationen, wollen wir, dass ein Wrapper Informationen wie die folgenden extrahiert:

$$\begin{aligned}\exists c, angebot \quad & c \in LaptopComputer \wedge angebot \in Produktangebote \wedge \\ & Hersteller(c, IBM) \wedge Modell(c, ThinkBook970) \wedge \\ & Bildschirmgröße(c, Zoll(14)) \wedge Bildschirmtyp(c, FarbLCD) \wedge \\ & Speichergröße(c, Megabyte(512)) \wedge CPUGeschwindigkeit(c, GHz(1{,}2)) \wedge \\ & AngebotenesProdukt(angebot, c) \wedge Geschäft(angebot, GenStore) \wedge \\ & URL(angebot, \texttt{"gen-store.com/compu/34356.html"}) \wedge \\ & Preis(angebot, \$(339)) \wedge Datum(angebot, Heute).\end{aligned}$$

Dieses Beispiel verdeutlicht mehrere Aspekte, die auftauchen, wenn wir die Aufgabe des Wissens-Engineerings für kommerzielle Transaktionen ernst nehmen. Beachten Sie beispielsweise, dass der Preis ein Attribut von *angebot* ist, nicht des eigentlichen Produktes. Das ist wichtig, weil die Angebotspreise in einem bestimmten Geschäft sich täglich ändern können, selbst für denselben Laptop; für einige Kategorien – wie etwa Häuser und Gemälde – kann dasselbe Objekt gleichzeitig von mehreren Händlern zu unterschiedlichen Preisen angeboten werden. Es gibt noch mehr Komplikationen, auf die wir noch nicht eingegangen sind, wie etwa die Möglichkeit, dass der Preis vom Zahlungsmittel abhängig ist oder davon, ob der Käufer bestimmte Rabatte in Anspruch nehmen kann. Die letzte Aufgabe ist, die extrahierten Angebote zu vergleichen. Betrachten Sie beispielsweise die folgenden drei Angebote:

A: 1,4 GHz CPU, 2 GB RAM, 250 GB Festplatte, $299

B: 1,2 GHz CPU, 4 GB RAM, 350 GB Festplatte, $500

C: 1,2 GHz CPU, 2 GB RAM, 250 GB Festplatte, $399

C wird von *A* **dominiert**, d.h., *A* ist billiger und schneller – ansonsten sind sie gleich. Im Allgemeinen wird *Y* von *X* dominiert, wenn *X* einen besseren Wert für mindestens ein Attribut hat und in keinem Attribut schlechter ist. Doch weder *A* noch *B* dominiert den anderen. Um zu entscheiden, was besser ist, müssen wir wissen, wie der Käufer CPU-Geschwindigkeit und Preis gegen Arbeitsspeicher und Festplattengröße aufwiegt. Das allgemeine Thema der Prioritäten zwischen mehreren Attributen wird in *Abschnitt 16.4* beschrieben; hier soll unser Shopping-Agent einfach nur eine Liste aller nicht dominierten Angebote zurückgeben, die der Beschreibung des Käufers entsprechen. In diesem Beispiel sind weder *A* noch *B* dominiert. Beachten Sie, dass dieses Ergebnis auf der Annahme basiert, dass jeder niedrigere Preise, schnellere Prozessoren und mehr Speicherplatz bevorzugt. Einige Attribute, wie beispielsweise die Bildschirmgröße eines Notebooks, sind von den jeweiligen Bedürfnissen des Käufers abhängig (Portabilität gegenüber Anzeigequalität); für diese Aspekte muss der Shopping-Agent einfach beim Benutzer nachfragen.

Der hier beschriebene Shopping-Agent ist ganz einfach; es sind vielerlei Verfeinerungen möglich. Dennoch besitzt er genügend Leistung, um mit dem richtigen domänenspezifischen Wissen nützlich für einen Einkäufer sein zu können. Aufgrund seines deklarativen Aufbaus kann er leicht auf komplexere Anwendungen erweitert werden. Der wichtigste Punkt in diesem Abschnitt war, dass eine Wissensrepräsentation – insbesondere die Produkthierarchie – für einen solchen Agenten erforderlich ist und dass sich der Rest fast von allein ergibt, wenn erst einmal ein bestimmtes Wissen in dieser Form vorliegt.

Bibliografische und historische Hinweise

Briggs (1985) behauptet, dass die Forschung zur formalen Wissensrepräsentation mit den klassischen indischen Theorien zur Shastra-Grammatik (Sanskrit) begann, die in das erste Jahrhundert v. Chr. zurückreicht. In der westlichen Welt kann die Verwendung der Termdefinitionen in der antiken griechischen Mathematik als das erste Vorkommen betrachtet werden: Das Werk *Metaphysik* (wörtlich „Das hinter der Physik“) von Aristoteles ist fast ein Synonym für Ontologie. Letztlich kann die Entwicklung technischer Terminologie in jedem Bereich als Form der Wissensrepräsentation betrachtet werden.

Frühe Diskussionen der Repräsentation in KI tendierten zu einer Konzentration auf „*Problem*repräsentation“ statt „*Wissens*repräsentation“. Lesen Sie dazu beispielsweise die Diskussion des Missionar-und-Kannibalen-Problems von Amarel (1968). In den 1970er Jahren verfolgte die KI die Entwicklung von „Expertensystemen“ (auch als „wissensbasierte Systeme“ bezeichnet), die die Leistung menschlicher Experten für eng abgesteckte Aufgaben erreichten oder überholten, wenn man ihnen ein geeignetes Domänenwissen bereitstellte. Beispielsweise interpretierte das erste Expertensystem, DENDRAL (Feigenbaum et al., 1971; Lindsay et al., 1980) die Ausgaben eines Massenspektrometers (ein Instrument für die Analyse der Struktur organischer chemischer Verbindungen) ebenso genau wie ein erfahrener Chemiker. Obwohl der Erfolg von DENDRAL wichtig war, um die KI-Forschergemeinde davon zu überzeugen, welche Bedeutung der Wissensrepräsentation zukommt, sind die darin verwendeten darstellerischen Formalismen hochspezifisch für die Domäne der Chemie. Mit der Zeit interessierten sich die Forscher für Formalismen und Ontologien der standardisierten Wissensrepräsentation, die den Prozess, neue Expertensysteme zu erstellen, dynamischer gestalten konnten. Dabei betraten sie auch das Terrain, das zuvor von Philosophen und Sprachwissenschaftlern beherrscht wurde. Die Disziplin, die sich in der KI aus der Notwendigkeit ergab, dass die Theorien „funktionieren“ müssen, führte zu einem schnelleren und tieferen Fortschritt als zu der Zeit, als diese Probleme ausschließlich ein Thema der Philosophie waren (auch wenn sie zuweilen zu einer Neuerfindung des Rades geführt hat).

Die Erstellung umfassender Taxonomien oder Klassifizierungen reicht bis in die Antike zurück. Aristoteles (384–322 v. Chr.) beschäftigte sich viel mit Klassifizierungs- und Kategorisierungsschemas. Sein *Organon*, eine Sammlung von Arbeiten zur Logik, die von seinen Studenten nach seinem Tod zusammengetragen wurden, beinhaltet eine Abhandlung namens *Categories*, wo er versucht, das zu konstruieren, was wir als obere Ontologie bezeichnen würden. Außerdem hat er die Konzepte der **Gattung** und der **Spezies** für die Klassifizierung auf unterer Ebene eingeführt. Unser aktuelles System biologischer Klassifizierung einschließlich der „biologischen Nomenklatur“ (Klassifizierung über Gattung und Spezies im technischen Sinn) wurde von dem schwedischen Biologen Carolus Linnaeus (1707–1778, auch Carl von Linne genannt) erfunden. Die Probleme, die mit natürlichen Arten und unscharfen Kategoriegrenzen zu tun haben, wurden unter anderem von Wittgenstein (1953), Quine (1953), Lakoff (1987) und Schwartz (1977) betrachtet.

Das Interesse an größeren Ontologien wächst, wie es das *Handbook on Ontologies* (Staab, 2004) dokumentiert. Das Projekt OPENCYC (Lenat und Guha, 1990; Matuszek et al., 2006) hat eine Ontologie mit 150.000 Begriffen herausgegeben, mit einer oberen Ontologie ähnlich der in Abbildung 12.1 gezeigten sowie spezifischen Begriffen wie „OLED Display“ und „iPhone“, das eine Art „Mobiltelefon“ darstellt und mit „Konsumelektronik“, „Telefon“, „Drahtloskommunikationsgerät“ und anderen Begriffen verbunden ist. Das Projekt DBPEDIA extrahiert strukturierte Daten aus Wikipedia, und zwar insbesondere aus Infoboxen – den Kästen von Attribut/Wert-Paaren, die viele Wikipedia-Artikel begleiten (Wu und Weld, 2008; Bizer et al., 2007). Mitte 2009 enthielt DBPEDIA 2,6 Millionen Datensätze mit rund 100 Fakten pro Datensatz. Die IEEE-Arbeitsgruppe P1600.1 erstellte die Suggested Upper Merged Ontology (SUMO) (Niles und Pease, 2001; Pease und Niles, 2002), die rund 1000 Terme in der oberen Ontologie und Verknüpfungen zu über 20.000 domänenspezifischen Begriffen enthält. Stoffel et al. (1997) beschreibt Algorithmen für effizientes Verwalten einer sehr großen Ontologie. Einen

Überblick zu Techniken, mit denen sich Wissen aus Webseiten extrahieren lässt, geben Etzioni et al. (2008) an.

Im Web bilden sich Darstellungssprachen heraus. RDF (Brickley und Guha, 2004) erlaubt es, Behauptungen in Form von relationalen Tripeln vorzunehmen, und bietet einige Instrumente an, um die Bedeutung von Namen im Lauf der Zeit zu entwickeln. OWL (Smith et al., 2004) ist eine Beschreibungslogik, die Inferenzen über diesen Tripeln unterstützt. Bislang scheint die Nutzung umgekehrt proportional zur Darstellungskomplexität zu sein: Die herkömmlichen HTML- und CSS-Formate machen über 99% des Webinhaltes aus, gefolgt von den einfachsten Darstellungsschemas wie zum Beispiel Mikroformate (Khare, 2006) und RDFa (Adida und Birbeck, 2008), die HTML- und XHTML-Markup verwenden, um literalem Text Attribute hinzuzufügen. Die anspruchsvollen RDF- und OWL-Ontologien werden bislang nur in geringem Umfang genutzt und die Vision des **Semantischen Webs** (Berners-Lee et al., 2001) ist noch nicht realisiert. Die FOIS-Konferenzen (*Formal Ontology in Information Systems*) enthalten zahlreiche interessante Unterlagen sowohl zu allgemeinen als auch zu domänenspezifischen Ontologien.

Die in diesem Kapitel verwendete Taxonomie wurde von den Autoren entwickelt und basiert zum Teil auf ihrer Erfahrung aus dem CYC-Projekt und zum Teil auf der Arbeit von Hwang und Schubert (1993) und Davis (1990). Eine inspirierende Diskussion des allgemeinen Projektes der Repräsentation des Allgemeinwissens erscheint in „The Naive Physics Manifesto“ von Hayes (1978, 1985b). Zu den erfolgreichen tiefen Ontologien auf einem Spezialgebiet gehören das Gene Ontology-Projekt (Consortium, 2008) und CML, die Chemical Markup Language (Murray-Rust et al., 2003).

Doctorow (2001), Gruber (2004) und Halevy et al. (2009) äußern Zweifel an der Realisierbarkeit einer einzigen Ontologie für das gesamte Wissen und Smith (2004) stellt fest, dass „das anfängliche Projekt einer einzigen Ontologie ... weitgehend aufgegeben wurde“.

Der Ereigniskalkül wurde von Kowalski und Sergot (1986) eingeführt, um stetige Zeit zu verarbeiten, und es gibt mehrere Varianten (Sadri und Kowalski, 1995; Shanahan, 1997) und Überblicke (Shanahan, 1999; Mueller, 2006). Van Lambalgen und Hamm (2005) zeigen, wie die Logik von Ereignissen auf die Sprache abgebildet wird, mit der wir über Ereignisse sprechen. Eine Alternative zu den Ereignis- und Situationskalkülen ist der Fluent-Kalkül (Thielscher, 1999). James Allen führte Zeitintervalle für denselben Zweck ein (Allen, 1984). Er sagte, dass Intervalle für das Schließen über ausgedehnte und nebenläufige Ereignisse sehr viel natürlicher als Situationen seien. Peter Ladkin (1986a, 1986b) führte „konkave“ Zeitintervalle ein (Intervalle mit Lücken; im Wesentlichen Vereinigungen gewöhnlicher „konvexer“ Zeitintervalle) und wendete die Techniken der mathematischen abstrakten Algebra auf die Zeitrepräsentation an. Allen (1991) untersucht systematisch die große Vielfalt an Techniken, die für die Zeitrepräsentation zur Verfügung stehen; Beek und Manchak (1996) analysieren Algorithmen für temporales Schließen. Es gibt deutliche Gemeinsamkeiten zwischen der in diesem Kapitel vorgestellten ereignisbasierten Ontologie und einer Analyse von Ereignissen nach dem Philosophen Donald Davidson (1980). Die **Historien** in der Ontologie der Flüssigkeiten von Pat Hayes (1985a) und die Chroniken in der Plantheorie von McDemott (1985) waren ebenfalls wichtige Einflussfaktoren für das Gebiet und dieses Kapitel.

Die Frage des ontologischen Zustandes von Substanzen hat eine lange Geschichte. Plato sagte, Substanzen seien abstrakte Entitäten, die sich völlig von physischen Objekten unterscheiden; er sagte *GemachtAus*(*Butter*$_3$, *Butter*) statt *Butter*$_3 \in$ *Butter*. Das führte zu einer Substanzhierarchie, in der beispielsweise *UngesalzeneButter* eine spezifischere Substanz als *Butter* ist. Die in diesem Kapitel übernommene Position, dass Substanzen Kategorien von Objekten sind, wurde von Richard Montague (1973) vorgestellt. Sie wurde auch im CYC-Projekt eingesetzt. Copeland (1993) führt einen ernsthaften, aber nicht unabwehrbaren Angriff. Der alternative Ansatz, der in diesem Kapitel angesprochen wurde, wobei Butter ein Objekt ist, das aus allen buttrigen Objekten des gesamten Universums besteht, wurde ursprünglich von dem polnischen Logiker Leśniewski (1916) vorgeschlagen. Seine **Mereologie** (der Name leitet sich von dem griechischen Wort für „Teil“ ab) verwendete die Teil/Ganzes-Relation als Ersatz für die mathematische Mengentheorie – mit dem Ziel, abstrakte Entitäten wie Mengen zu eliminieren. Eine besser lesbare Darstellung dieser Konzepte finden Sie bei Leonard und Goodman (1940). *The Structure of Appearance* (1977) von Goodman wendet die Konzepte auf verschiedene Probleme in der Wissensrepräsentation an. Obwohl einige Aspekte des mereologischen Ansatzes schrecklich kompliziert waren – z.B. die Notwendigkeit eines separaten Vererbungsmechanismus, der auf der Teil/Ganzes-Relation basierte –, fand er Unterstützung durch Quine (1960). Harry Bunt (1985) zeigt eine umfassende Analyse seiner Verwendung in der Wissensrepräsentation. Casati und Varzi (1999) behandeln Teile, Ganzes und die räumlichen Positionen.

Mentale Objekte und Zustände waren Thema intensiver Forschung in Philosophie und KI. Man unterscheidet drei Hauptansätze. Der in diesem Kapitel vorgestellte Ansatz, der auf modaler Logik und möglichen Welten basiert, ist der klassische Ansatz aus der Philosophie (Hintikka, 1962; Kripke, 1963; Hughes und Cresswell, 1996). Das Buch *Reasoning about Knowledge* (Fagin et al., 1995) bietet eine gründliche Einführung. Der zweite Ansatz ist eine Theorie erster Stufe, in der mentale Objekte Fluenten sind. Davis (2005) sowie Davis und Morgenstern (2005) beschreiben diesen Ansatz. Er stützt sich auf den Formalismus der möglichen Welten und baut auf der Arbeit von Robert Moore (1980, 1985) auf. Der dritte Ansatz ist eine **syntaktische Theorie**, in der mentale Objekte durch Zeichenfolgen dargestellt werden. Eine Zeichenfolge ist einfach ein komplexer Term, der eine Liste von Symbolen kennzeichnet. So lässt sich *KannFliegen*(*Clark*) durch die Liste der Symbole [*K*, *a*, *n*, *n*, *F*, *l*, *i*, *e*, *g*, *e*, *n*, (, *C*, *l*, *a*, *r*, *k*,)] darstellen. Die syntaktische Theorie der mentalen Objekte wurde zuerst von Kaplan und Montague (1960) eingehend untersucht, die zeigten, dass sie bei mangelnder Sorgfalt zu Paradoxons führt. Ernie Davis (1990) bietet einen ausgezeichneten Vergleich der syntaktischen und modalen Wissenstheorien.

Der griechische Philosoph Porphyr (ca. 234 – 305 v. Chr.), der die *Kategorien* von Aristoteles kommentierte, beschrieb das, was man als das erste semantische Netz bezeichnen könnte. Charles S. Peirce (1909) entwickelte Existenzgraphen als ersten Formalismus für semantische Netze unter Verwendung moderner Logik. Ross Quillian (1961), motiviert durch sein Interesse am menschlichen Gedächtnis und der Spracherkennung, begann mit der Arbeit an semantischen Netzen innerhalb der KI. Eine einflussreiche Arbeit von Marvin Minsky (1975) zeigte eine Version semantischer Netze, die dort als

Frames (Rahmen) bezeichnet wurden; ein Frame war eine Repräsentation eines Objektes oder einer Kategorie, mit Attributen und Relationen zu anderen Objekten oder Kategorien. Schnell stellte sich die Frage der Semantik für die semantischen Netze von Quillian (und die von anderen, die seinem Ansatz folgten) mit ihren allgegenwärtigen und sehr vagen „IS-A-Verknüpfungen". Der bekannte Artikel von Woods (1975) „What's In a Link?" lenkte die Aufmerksamkeit der KI-Forscher auf die Notwendigkeit einer exakten Semantik in den Formalismen zur Wissensrepräsentation. Brachman (1979) arbeitete diesen Aspekt aus und schlug Lösungen vor. In „The Logic of Frames" geht Patrick Hayes (1979) sogar noch weiter und behauptet: „Ein Großteil der ‚Frames' ist letztlich nur eine neue Syntax für Teile der Logik erster Stufe." „Tarskian Semantics, or No Notation without Denotation!" von Drew McDermott (1978b) forderte, dass der modelltheoretische Ansatz der Semantik, wie er in der Logik erster Stufe verwendet wurde, auf alle Formalismen zur Wissensrepräsentation verwendet werden sollte. Das bleibt eine umstrittene Idee. Bemerkenswert ist, dass McDermott selbst seine Position in „A Critique of Pure Reason" (McDermott, 1987) revidiert hat. Selman und Levesque (1993) diskutierten die Komplexität der Vererbung mit Ausnahmen und zeigten, dass sie in den meisten Formulierungen NP-vollständig ist.

Die Entwicklung der Beschreibungslogiken ist die jüngste Phase einer langen Forschungsentwicklung, die versuchte, sinnvolle Untermengen der Logik erster Stufe zu finden, für die die Inferenz rechentechnisch praktikabel ist. Hector Levesque und Ron Brachman (1987) zeigten, dass bestimmte logische Konstrukte – insbesondere bestimmte Verwendungen von Disjunktion und Negation – hauptsächlich für die Nichthandhabbarkeit der logischen Inferenz verantwortlich waren. Aufbauend auf dem System KL-ONE (Schmolze und Lipkis, 1983) entwickelten mehrere Forscher Systeme, die eine theoretische Komplexitätsanalyse einbinden; bemerkenswert sind dabei vor allem KRYPTON (Brachman et al., 1983) und CLASSIC (Borgida et al., 1989). Das Ergebnis waren eine deutliche Erhöhung der Inferenzgeschwindigkeit sowie ein besseres Verständnis für die Interaktion zwischen Komplexität und Ausdruckskraft in Inferenzsystemen. Calvanese et al. (1999) fassen den aktuellen Stand zusammen und Baader et al. (2007) legen ein umfassendes Handbuch der Beschreibungslogik vor. Entgegen diesem Trend sagen Doyle und Patil (1991), dass die Einschränkung der Ausdruckskraft einer Sprache es entweder unmöglich macht, bestimmte Probleme zu lösen, oder den Benutzer ermutigt, die Sprachbeschränkungen durch nichtlogische Mittel zu umgehen.

Die drei wichtigsten Formalismen für den Umgang mit der nichtmonotonen Inferenz – Circumscription (McCarthy, 1980), Defaultlogik (Reiter, 1980) und modale nichtmonotone Logik (McDermott und Doyle, 1980) – wurden in einer Sonderausgabe des AI Journal vorgestellt. Delgrande und Schaub (2003) erörtern die Vorzüge der Varianten, wobei sie auf 25 Jahre zurückblicken. Die Antwortmengenprogrammierung kann man als Erweiterung der Negation durch Fehler oder als Verfeinerung der Circumscription betrachten; die zugrunde liegende Theorie der stabilen Modellsemantik wurde von Gelfond und Lifschitz (1988) vorgestellt, und die führenden Systeme für die Antwortmengenprogrammierung sind DLV (Eiter et al., 1998) und SMODELS (Niemelä et al., 2000). Das Laufwerksbeispiel stammt aus dem Benutzerhandbuch zu SMODELS (Syrjänen, 2000). Lifschitz (2001) diskutiert die Verwendung der Antwortmengenprogrammierung für die Planung. Brewka et al. (1997) bieten einen guten Überblick über die verschiedenen Ansätze zur nichtmonotonen Logik. Clark (1978) beschreibt den Ansatz der Negation durch Fehler für die Logikprogrammierung und Clark-Vervollständigung. Van Emden und Kowalski (1976) zeigen, dass jedes Prolog-Programm ohne Negation ein ein-

deutiges minimales Modell besitzt. In den letzten Jahren gab es ein neues Interesse an Anwendungen der nichtmonotonen Logik auf große Wissensrepräsentationssysteme. Die BENINQ-Systeme für Gewinnauskünfte bei Versicherungen stellen wahrscheinlich die erste kommerziell erfolgreiche Anwendung eines nichtmonotonen Vererbungssystems dar (Morgenstern, 1998). Lifschitz (2001) beschreibt die Anwendung der Antwortmengenprogrammierung auf die Planung. In den Unterlagen zur Konferenz *Logic Programming and Nonmonotonic Reasoning* (LPNMR) sind eine Vielzahl nichtmonotoner Inferenzsysteme beschrieben.

Die Betrachtung von Truth-Maintenance-Systemen begann mit den Systemen TMS (Doyle, 1979) und RUP (McAllester, 1980), bei denen es sich im Wesentlichen um JTMS handelte. Forbus und de Kleer (1993) erläutern ausführlich, wie sich TMS in KI-Anwendungen einsetzen lassen. Nayak und Williams (1997) zeigen, wie es ein effizientes TMS ermöglicht, die Operationen eines NASA-Raumschiffes in Echtzeit zu planen.

Aus offensichtlichen Gründen konnte dieses Kapitel nicht *jeden* Bereich der Wissensrepräsentation detailliert beschreiben. Die drei wichtigsten der hier nicht beschriebenen Themen sind:

Qualitative Physik: Die qualitative Physik ist ein Unterbereich der Wissensrepräsentation, der sich insbesondere mit dem Aufbau einer logischen, nicht numerischen Theorie zu physischen Objekten und Prozessen beschäftigt. Dieser Begriff wurde von Johan de Kleer (1975) geprägt, obwohl man sagen kann, dass das Unternehmen bereits in BUILD von Fahlman (1974) begonnen hatte, einem komplexen Planer für die Konstruktion komplexer Türme aus Blöcken. Fahlman erkannte beim Entwurf, dass der meiste Aufwand (seiner Schätzung nach 80%) für die Modellierung der Physik der Blockwelt anfällt, um die Stabilität verschiedener Baugruppen aus Blöcken zu berechnen, und nicht bei der Planung an sich. Er skizziert einen der naiven Physik ähnlichen Prozess, um zu erklären, warum kleine Kinder BUILD-ähnliche Probleme lösen können, ohne Zugang zu der Hochgeschwindigkeits-Fließkommaarithmetik zu haben, die für die physische Modellierung in BUILD verwendet wurde. Hayes (1985a) verwendet „Historien“ – vierdimensionale Raum/Zeit-Abschnitte ähnlich den Ereignissen von Davidson –, um eine relativ komplexe naive Physik für Flüssigkeiten zu erzeugen. Hayes bewies als Erster, dass eine Wanne mit verschlossenem Ablauf irgendwann überläuft, wenn Wasser weiterhin aus dem offenen Hahn läuft, und dass eine Person, die in einen See fällt, völlig nass wird. Davis (2008) liefert eine Aktualisierung zur Ontologie von Flüssigkeiten, die das Einströmen von Flüssigkeiten in Behälter beschreibt.

De Kleer und Brown (1985), Ken Forbus (1985) und Benjamin Knipers (1985) entwickelten unabhängig voneinander und nahezu gleichzeitig Systeme, die über physikalische Systeme basierend auf der qualitativen Abstraktion der zugrunde liegenden Gleichungen schließen können. Qualitative Physik entwickelte sich bald bis zu einem Punkt, an dem es möglich wurde, eine beeindruckende Vielzahl komplexer physischer Systeme zu analysieren (Yip, 1991). Qualitative Techniken wurden verwendet, um neue Entwürfe für Uhren, Scheibenwischer sowie sechsbeinige Laufmaschinen zu entwickeln (Subramanian und Wang, 1994). Die Sammlung *Readings in Qualitative Reasoning about Physical Systems* (Weld und de Kleer, 1990), ein Enzyklopädie-Artikel von Kuipers (2001) und ein Handbuchartikel von Davis (2007) führen in das Gebiet ein.

Räumliches Schließen: Das Schließen, das erforderlich ist, um sich in der Wumpus-Welt oder in einer Supermarkt-Welt zu bewegen, ist banal im Vergleich zur komplexen räumlichen Struktur der realen Welt. Der erste ernsthafte Versuch, anhand von Allgemeinwissen Schlüsse über den Raum zu ziehen, erscheint in der Arbeit von Ernest Davis (1986, 1990). Der Bereichsverbindungskalkül von Cohn et al. (1997) unterstützt eine Form des qualitativen räumlichen Schließens und hat zu neuen Arten geografischer Informationssysteme geführt; siehe auch Davis (2006). Wie bei qualitativer Physik kommt ein Agent sozusagen lange ohne eine vollständige metrische Darstellung aus. Wenn aber eine derartige Repräsentation erforderlich wird, lassen sich die in der Robotik entwickelten Techniken (*Kapitel 25*) heranziehen.

Psychologisches Schließen: Das psychologische Schließen beinhaltet die Entwicklung einer funktionierenden *Psychologie* für künstliche Agenten, die diese Psychologie verwenden, um Schlüsse über sich selbst und andere Agenten zu ziehen. Das basiert häufig auf einer sogenannten Volkspsychologie, der Theorie, von der man annimmt, dass Menschen sie benutzen, um über sich und andere zu schließen. Wenn die KI-Forscher ihren künstlichen Agenten psychologische Theorien zum Schließen über andere Agenten bereitstellen, basieren diese Theorien häufig auf den Beschreibungen des eigenen Entwurfes ihrer Agenten. Das psychologische Schließen ist momentan am nützlichsten im Kontext der natürlichen Sprachverarbeitung, wo es von zentraler Bedeutung ist, die Absicht des Sprechers abzuleiten.

Minker (2001) sammelt Artikel von führenden Forschern der Wissensdarstellung und fasst somit 40 Jahre Arbeit auf diesem Gebiet zusammen. Die Unterlagen zu der internationalen Konferenz *Priniciples of Knowledge Representation and Reasoning* sind die aktuellsten Quellen für Arbeiten in diesem Bereich. *Readings in Knowledge Representation* (Brachman und Levesque, 1985) und *Formal Theories of the Commonsense World* (Hobbs und Moore, 1985) sind ausgezeichnete Anthologien über die Wissensrepräsentation; das Erstgenannte konzentriert sich auf geschichtlich wichtige Arbeiten über die Repräsentation von Sprachen und Formalismen, das Zweite beschreibt die Sammlung des eigentlichen Wissens. Die Lehrbücher von Davis (1990), Stefik (1995) und Sowa (1999) sind Einführungen zur Wissensrepräsentation, van Harmelen et al. (2007) steuert ein Handbuch bei und eine Sonderausgabe des AI Journal behandelt den jüngsten Fortschritt (Davis und Morgenstern, 2004). Die alle zwei Jahre stattfindende Konferenz zu *Theoretical Aspects of Reasoning About Knowledge* (TARK) befasst sich mit Anwendungen der Wissenstheorie in KI, Wirtschaft und verteilten Systemen.

Zusammenfassung

Wir hoffen, mit der Beschreibung der Details, wie man das unterschiedlichste Wissen repräsentieren kann, dem Leser eine Vorstellung davon, wie sich reale Wissensbasen konstruieren lassen, und ein Gefühl für die interessanten philosophischen Aspekte, die dabei auftauchen, vermittelt zu haben. Als wichtigste Punkte sind zu nennen:

- Für eine umfassende Wissensrepräsentation braucht man eine universelle Ontologie, um die verschiedenen spezifischen Wissensdomänen zu organisieren und miteinander zu verknüpfen.
- Eine allgemeine Ontologie muss eine große Vielfalt an Wissen abdecken und sollte in der Lage sein, im Prinzip mit jeder Domäne zurechtzukommen.
- Der Aufbau einer großen, universellen Ontologie ist eine beträchtliche Herausforderung, die immer noch nicht vollständig realisiert ist, obwohl aktuelle Frameworks recht robust zu sein scheinen.
- Wir haben eine **obere Ontologie** vorgestellt, die auf Kategorien und dem Ereigniskalkül basiert. Es wurden Kategorien, Subkategorien, Teile, strukturierte Objekte, Maße, Substanzen, Ereignisse, Zeit und Raum, Änderungen und Glauben behandelt.
- Zwar lassen sich natürliche Arten nicht vollständig in Logik definieren, doch können Eigenschaften natürlicher Arten dargestellt werden.
- Aktionen, Ereignisse und Zeit können entweder im Situationskalkül oder in ausdrucksstärkeren Repräsentationen wie zum Beispiel im Ereigniskalkül repräsentiert werden. Derartige Repräsentationen ermöglichen es einem Agenten, mithilfe logischer Inferenz Pläne zu konstruieren.
- Wir haben eine detaillierte Analyse der Internet-Shopping-Domäne vorgestellt, die allgemeine Ontologie vorgeführt und gezeigt, wie das Domänenwissen von einem Shopping-Agenten genutzt werden kann.
- Spezielle Repräsentationssysteme wie zum Beispiel **semantische Netze** und **Beschreibungslogiken** wurden eingeführt, um den Aufbau einer Kategoriehierarchie zu unterstützen. **Vererbung** ist eine wichtige Form der Inferenz und erlaubt es, dass die Eigenschaften von Objekten aus ihrer Zugehörigkeit zu Kategorien hergeleitet werden.
- Durch die **Annahme der geschlossenen Welt**, wie sie in Logikprogrammen implementiert ist, lässt es sich auf einfache Weise vermeiden, Unmengen negativer Informationen spezifizieren zu müssen. Am besten wird sie als **Standard** (Default) interpretiert, der durch zusätzliche Informationen überschrieben werden kann.
- **Nichtmonotone Logiken**, wie beispielsweise **Circumscription** und **Defaultlogik**, sollen das Schließen mit Defaultwerten im Allgemeinen abdecken.
- **Truth Maintenance Systems** (Wahrheitsverwaltungssysteme) verarbeiten effizient aktualisiertes und überarbeitetes Wissen.

Übungen zu Kapitel 12

Lösungshinweise

1 Definieren Sie eine Ontologie in Logik erster Stufe für Tic Tac Toe. Die Ontologie sollte Situationen, Aktionen, Quadrate, Spieler, Markierungen (X, O oder leer) und das Konzept für den Spielausgang – gewonnen, verloren, Remis – enthalten. Definieren Sie außerdem das Konzept des erzwungenen Gewinns (oder Remis): eine Position, von der aus ein Spieler einen Gewinn (oder ein Remis) mit der richtigen Sequenz von Aktionen erzwingen kann. Schreiben Sie Axiome für die Domäne. (Hinweis: Die Axiome, die die verschiedenen Quadrate aufzählen und die die Gewinnpositionen charakterisieren, sind ziemlich lang. Diese müssen Sie nicht vollständig ausschreiben, aber deutlich angeben, wie sie aussehen.)

2 Sie sollen ein System erstellen, das Informatikstudenten des Grundstudiums hilft zu entscheiden, welche Kurse sie über einen längeren Zeitraum belegen sollten, um die Studienanforderungen zu erfüllen. (Verwenden Sie die Anforderungen, die an Ihrer Hochschule gelten.) Legen Sie zunächst ein Vokabular fest, mit dem sich alle Informationen darstellen lassen, und repräsentieren Sie sie dann. Formulieren Sie eine Abfrage an das System, die ein zulässiges Studienprogramm als Lösung zurückgibt. Dabei sollten Sie es ermöglichen, dass dieses Programm auf einzelne Studenten genau zugeschnitten wird, indem Ihr System fragt, welche Kurse oder vergleichbaren Kurse der Student bereits belegt hat, und keine Programme zusammenstellen, in denen diese Kurse wiederholt werden.

Schlagen Sie vor, wie Ihr System verbessert werden könnte – indem Sie beispielsweise das Wissen über die Vorlieben, die Arbeitsauslastung, gute und schlechte Dozenten usw. berücksichtigen. Erklären Sie für jede Wissensart, wie sie logisch ausgedrückt werden könnte. Kann Ihr System diese Information einfach einbinden, um das *beste* Studienprogramm für einen Studenten zu ermitteln?

3 Entwickeln Sie ein Darstellungssystem für das Schließen über Fenster in einer fensterbasierten Computerbenutzeroberfläche. Insbesondere sollte Ihre Darstellung in der Lage sein, Folgendes zu beschreiben:

- Den Zustand eines Fensters: minimiert, angezeigt oder nicht vorhanden
- Welches Fenster (falls vorhanden) das aktive Fenster ist
- Die Position jedes Fensters zu einem bestimmten Zeitpunkt
- Die Reihenfolge von überlappenden Fenstern (von vorn nach hinten)
- Die Aktionen, die Fenster erstellen, zerstören, in der Größe ändern und verschieben, den Zustand eines Fensters ändern und ein Fenster nach vorn bringen. Behandeln Sie diese Aktionen als atomar; d.h., befassen Sie sich nicht mit der Frage, wie sich diese Aktionen mit Mausaktionen in Beziehung setzen lassen. Geben Sie Axiome an, die die Wirkungen von Aktionen auf Fluenten beschreiben. Hierzu können Sie entweder den Ereignis- oder den Situationskalkül heranziehen.

Nehmen Sie eine Ontologie an, die $Situationen$, $Aktionen$, $Ganzzahlen$ (für die x- und y-Koordinaten) und $Fenster$ enthält. Definieren Sie eine Sprache über dieser Ontologie; d.h. eine Liste von Konstanten, Funktionssymbolen und Prädikaten mit jeweils einer Beschreibung in natürlicher Sprache. Wenn Sie weitere Kategorien in die Ontologie einfügen müssen (z.B. Pixel), können Sie das tun. Achten Sie aber darauf, diese in Ihrer Rezension zu spezifizieren. Die im Text definierten Symbole können (und sollten) Sie verwenden, Sie müssen aber darauf achten, diese explizit aufzulisten.

4 Drücken Sie die folgenden Aussagen in der Sprache aus, die Sie für die vorherige Übung entwickelt haben:

a. In Situation S_0 liegt Fenster W_1 hinter W_2, ragt aber oben und unten heraus. Geben Sie für diese *keine* genauen Koordinaten an; beschreiben Sie die allgemeine Situation.

b. Wenn ein Fenster angezeigt wird, ist sein oberer Rand höher als sein unterer Rand.

c. Nachdem Sie ein Fenster w erstellt haben, wird es angezeigt.

d. Ein Fenster lässt sich nur minimieren, wenn es angezeigt wird.

5 (Übernommen aus einem Beispiel von Doug Lenat.) Sie haben die Aufgabe, in logischer Form genügend Wissen zu sammeln, um eine Reihe von Fragen über das folgende einfache Szenario zu beantworten:

Gestern ging John in den Supermarkt *Allesfrisch* und kaufte zwei Pfund Tomaten und ein Pfund Rindfleisch.

Versuchen Sie als Erstes, den Inhalt des Satzes als eine Folge von Behauptungen zu repräsentieren. Sie sollten Sätze mit einfacher logischer Struktur schreiben (z.B. Aussagen, dass Objekte bestimmte Eigenschaften haben, dass Objekte in einer bestimmten Beziehung zueinander stehen, dass alle Objekte, die eine Eigenschaft erfüllen, auch eine andere erfüllen usw.) Die folgenden Anmerkungen sollen Ihnen helfen:

– Welche Klassen, Objekte und Relationen brauchen Sie? Was sind ihre Eltern, Geschwister usw.? (Sie brauchen u.a. Ereignisse und eine temporale Reihenfolge.)

– Wo würden sie in einer allgemeineren Hierarchie angeordnet?

– Welche Bedingungen gibt es und welche Beziehungen bestehen zwischen ihnen?

– Wie detailliert müssen Sie die verschiedenen Konzepte ausarbeiten?

Um die unten angegebenen Fragen zu beantworten, muss Ihre Wissensbasis Hintergrundwissen einschließen. Sie müssen sich damit beschäftigen, welche Dinge es in einem Supermarkt gibt, wie man die ausgesuchten Dinge kauft, wofür die Einkäufe verwendet werden usw. Versuchen Sie, Ihre Repräsentation so allgemein wie möglich zu halten. Ein einfaches Beispiel: Sagen Sie nicht: „Menschen kaufen Lebensmittel bei Allesfrisch", weil Ihnen das nicht hilft, wenn andere Menschen in einem anderen Supermarkt einkaufen. Wandeln Sie die Fragen auch nicht in Antworten um; z.B. lautet Frage (c): „Hat John Fleisch gekauft?", und nicht: „Hat John ein Pfund Rindfleisch gekauft?"

Skizzieren Sie die Schlussketten, die diese Fragen beantworten. Verwenden Sie, wenn möglich, ein logisches Schlusssystem, das zeigt, dass Ihre Wissensbasis ausreichend ist. Viele der Dinge, die Sie schreiben, sind in der Realität vielleicht nur annähernd korrekt, aber machen Sie sich nicht zu viele Gedanken; der Sinn dabei ist, das Allgemeinwissen zu extrahieren, das es Ihnen erlaubt, die Fragen überhaupt zu beantworten. Eine wirklich vollständige Antwort auf diese Frage ist *äußerst* schwierig und kann womöglich mit den aktuellen Möglichkeiten der Wissensrepräsentation nicht ermittelt werden. Aber Sie sollten in der Lage sein, eine konsistente Menge an Axiomen für die hier gezeigten (eingeschränkten) Fragen zusammenzustellen.

a. Ist John ein Kind oder ein Erwachsener? [Erwachsener]

b. Hat John jetzt mindestens zwei Tomaten? [Ja]

c. Hat John Fleisch gekauft? [Ja]

d. Wenn Mary gleichzeitig Tomaten gekauft hat, hat John sie dann gesehen? [Ja]
e. Werden die Tomaten im Supermarkt hergestellt? [Nein]
f. Was macht John mit den Tomaten? [Essen]
g. Verkauft *Allesfrisch* auch Deodorant? [Ja]
h. Hat John Geld oder eine Kreditkarte mit in den Supermarkt gebracht? [Ja]
i. Hat John weniger Geld, nachdem er den Supermarkt verlassen hat? [Ja]

6 Nehmen Sie die erforderlichen Änderungen oder Ergänzungen an Ihrer Wissensbasis aus der obigen Übung vor, sodass die folgenden Fragen beantwortet werden können. Nehmen Sie in Ihren Bericht eine Diskussion Ihrer Änderungen auf, woraus hervorgeht, warum sie erforderlich waren, ob sie groß oder klein waren und welche Art von Fragen weitere Änderungen nach sich ziehen würden.

a. Sind noch andere Menschen im Supermarkt, während John einkauft? [Ja – Angestellte!]
b. Ist John Vegetarier? [Nein]
c. Wem gehört das Deodorant bei *Allesfrisch*? [Der Firma *Allesfrisch*]
d. Hat John ein halbes Pfund Rindfleisch? [Ja]
e. Verkauft die Shell-Tankstelle nebenan Benzin? [Ja]
f. Passen die Tomaten in Johns Kofferraum? [Ja]

7 Stellen Sie die folgenden sieben Sätze unter Verwendung und Erweiterung der in diesem Kapitel entwickelten Repräsentationen dar:

a. Wasser ist eine Flüssigkeit zwischen 0 und 100 Grad.
b. Wasser siedet bei 100 Grad.
c. Das Wasser in der Wasserflasche von John ist gefroren.
d. Perrier ist eine Art Wasser.
e. John hat Perrier in seiner Wasserflasche.
f. Alle Flüssigkeiten haben einen Gefrierpunkt.
g. Ein Liter Wasser wiegt mehr als ein Liter Alkohol.

8 Schreiben Sie Definitionen für Folgendes:

a. $ErschöpfendeTeileZerlegung$
b. $TeilePartition$
c. $TeilweiseDisjunkt$

Die Definitionen sollten analog zu den Definitionen für $ErschöpfendeZerlegung$, $Partition$ und $Disjunkt$ sein. Gilt $TeilePartition(s, BündelVon(s))$? Wenn ja, beweisen Sie es; wenn nein, geben Sie ein Gegenbeispiel an und definieren Sie ausreichende Bedingungen, unter denen es gilt.

9 Ein alternatives Schema für die Darstellung von Maßen beinhaltet die Anwendung der Einheitenfunktion auf ein abstraktes Längenobjekt. Bei einem solchen Schema würde man schreiben: $Zoll(Länge(\hat{L1})) = 1{,}5$. Wie verhält sich dieses Schema zu dem in diesem Kapitel gezeigten? Berücksichtigen Sie dabei unter anderem die folgenden Aspekte: Umwandlungsaxiome, Namen für abstrakte Mengen (wie etwa „50 Dollar") und Vergleiche abstrakter Maße in unterschiedlichen Einheiten (50 Zoll ist mehr als 50 Zentimeter).

10 Schreiben Sie eine Menge von Sätzen, die es erlaubt, den Preis einer einzelnen Tomate (oder eines anderen Objektes) zu ermitteln, wenn man den Preis für ein Pfund kennt. Erweitern Sie die Theorie, um den Preis für eine Tüte voll Tomaten zu berechnen.

11 Fügen Sie Sätze hinzu, um die Definition des Prädikates $Name(s, c)$ zu erweitern, sodass eine Zeichenfolge wie etwa „Laptop Computer" mit den geeigneten Kategorienamen aus den verschiedensten Geschäften verglichen werden kann. Versuchen Sie, Ihre Definition zu verallgemeinern. Testen Sie sie, indem Sie zehn Online-Geschäfte betrachten und die Kategorienamen überprüfen, die sie für drei verschiedene Kategorien verwenden. Für die Kategorie der Laptops beispielsweise haben wir die Namen „Notebooks", „Laptops", „Notebook Computers", „Notebook", „Laptops und Notebooks" und „Notebook PCs" gefunden. Einige davon werden durch explizite $Name$-Fakten abgedeckt, während andere von Regeln für die Berücksichtigung des Plurals, von Konjunktionen usw. abgedeckt werden könnten.

12 Schreiben Sie Ereigniskalkülaxiome, um die Aktionen in der Wumpus-Welt zu beschreiben.

13 Formulieren Sie die Intervall-Algebra-Relation, die zwischen jedem Paar der folgenden Ereignisse aus der realen Welt gilt:

LK: Das Leben von Präsident Kennedy

KK: Die Kindheit von Präsident Kennedy

PK: Die Präsidentschaft von Präsident Kennedy

LJ: Das Leben von Präsident Johnson

PJ: Die Präsidentschaft von Präsident Johnson

LO: Das Leben von Präsident Obama

14 In dieser Übung betrachten wir das Problem, eine Route zu planen, über die ein Roboter von einer Stadt in eine andere gelangt. Die grundlegende Aktion des Roboters ist $Gehen(x, y)$, die ihn von der Stadt x zur Stadt y bringt, falls es eine direkte Route von x nach y gibt. $Strasse(x, y)$ ist genau dann wahr, wenn es eine Straße gibt, die die Städte x und y verbindet; ist das der Fall, liefert $Distanz(x, y)$ die Länge der Straße. Die Karte in *Abschnitt 3.1.1* (Abbildung 3.2) zeigt ein Beispiel. Der Roboter beginnt in Arad und muss nach Bukarest gelangen.

a. Schreiben Sie eine geeignete logische Beschreibung der Ausgangssituation des Roboters.

b. Schreiben Sie eine geeignete logische Abfrage, deren Lösungen mögliche Pfade zum Ziel darstellen.

c. Schreiben Sie einen Satz, der die Aktion $Gehen$ beschreibt.

d. Der Roboter habe einen Benzinverbrauch von 5 l auf 100 km. Zu Beginn ist sein Tank mit 75 l gefüllt. Erweitern Sie Ihre Repräsentation, um diese Betrachtungen zu berücksichtigen.

e. Nehmen Sie nun an, dass einige Städte Tankstellen haben, wo der Roboter seinen Tank auffüllen kann. Erweitern Sie Ihre Repräsentation und schreiben Sie alle Regeln, die für die Beschreibung von Tankstellen benötigt werden, einschließlich der Aktion $TankFüllen$.

15 Untersuchen Sie Möglichkeiten, um den Ereigniskalkül zu erweitern und simultane Ereignisse verarbeiten zu können. Lässt sich eine kombinatorische Explosion von Axiomen vermeiden?

16 Konstruieren Sie eine Repräsentation für Wechselkurse zwischen Währungen, die täglichen Schwankungen unterliegen.

17 Definieren Sie das Prädikat $Feststehend$, wobei $Feststehend(Position(x))$ bedeutet, dass die Position des Objektes x über die Zeit feststehend ist.

18 Beschreiben Sie das Ereignis, etwas gegen etwas anderes zu tauschen. Beschreiben Sie das Kaufen als Art Tausch, wobei eines der Objekte gegen einen bestimmten Geldbetrag getauscht wird.

19 Die beiden vorigen Übungen gehen von einem recht primitiven Konzept des Eigentums aus. Beispielsweise *besitzt* der Käufer zunächst die Geldscheine. Dieses Bild bröckelt, wenn sich das Geld etwa auf der Bank befindet, weil es dann keine bestimmten Geldscheine mehr gibt, die er besitzt. Es wird noch komplizierter durch Borgen, Leasen, Mieten und Bürgen. Untersuchen Sie die verschiedenen allgemeingültigen und erlaubten Konzepte des Eigentums und schlagen Sie ein Schema vor, wie sie formal repräsentiert werden können.

20 (Übernommen von Fagin et al. (1995).) Betrachten Sie ein Spiel, das mit einem Stapel von lediglich 8 Karten – 4 Assen und 4 Königen – gespielt wird. Die drei Spieler Alice, Bob und Carlos erhalten beim Geben jeweils zwei Karten. Ohne diese selbst anzusehen, heften sie sich die Karten auf ihre Stirn, sodass andere Spieler sie sehen können. Die Spieler sind dann nacheinander an der Reihe, indem sie entweder ankündigen, dass sie wissen, welche Karten sich auf ihrer eigenen Stirn befinden, wodurch sie das Spiel gewinnen, oder sagen: „Ich weiß nicht." Jeder kennt die Spieler als wahrheitsliebend und perfekt in Bezug auf das Schließen über Glauben.

a. Spiel 1: Alice und Bob haben beide gesagt: „Ich weiß nicht." Carlos sieht, dass Alice zwei Asse (A-A) und Bob zwei Könige (K-K) hat. Was sollte Carlos sagen? (*Hinweis*: Betrachten Sie alle drei möglichen Fälle für Carlos: A-A, K-K, A-K.)

b. Beschreiben Sie jeden Schritt von Spiel 1 mithilfe der Notation modaler Logik.

c. Spiel 2: Carlos, Alice und Bob sagen in ihrer ersten Runde: „Ich weiß nicht." Alice hält die Karten K-K und Bob die Karten A-K. Was sollte Carlos bei seiner zweiten Runde sagen?

d. Spiel 3: Alice, Carlos und Bob sagen in ihrer ersten Runde: „Ich weiß nicht.", Alice auch in ihrer zweiten Runde. Sowohl Alice als auch Bob halten die Karten A-K. Was sollte Carlos sagen?

e. Beweisen Sie, dass es in diesem Spiel immer einen Gewinner gibt.

21 Die in *Abschnitt 12.4* diskutierte Annahme *logischer Allwissenheit* ist für jeden wirklich logisch Schließenden natürlich nicht wahr. Stattdessen handelt es sich um eine von den Anwendungen abhängige mehr oder weniger akzeptable *Idealisierung* des Schlussfolgerungsprozesses. Erörtern Sie die Vernünftigkeit der Annahme für die folgenden Anwendungen des Schließens über Wissen:

a. Schach mit Uhr. Hier kann der Spieler über die Grenzen seines Gegners oder seiner eigenen Fähigkeit, den besten Zug in der verfügbaren Zeit zu finden, schließen wollen. Wenn zum Beispiel Spieler A wesentlich mehr Restzeit zur Verfügung hat als Spieler B, wird A manchmal einen Zug ausführen, der die Situation wesentlich verkompliziert, wobei er darauf hofft, einen Vorteil zu erlangen, da er mehr Zeit hat, eine geeignete Strategie auszuarbeiten.

b. Ein Shopping-Agent in einer Umgebung, in der Kosten für das Zusammentragen von Informationen anfallen

c. Ein automatisiertes Lernprogramm für Mathematik, das darüber schließt, was Studenten verstehen

d. Schließen über Kryptografie mit öffentlichen Schlüsseln, die auf der Nichtbehandelbarkeit bestimmter rechentechnischer Probleme beruht

22 Übersetzen Sie die folgende Beschreibung eines logischen Ausdruckes (aus *Abschnitt 12.5.2*) in Logik erster Stufe und erörtern Sie das Ergebnis:

$$And(Mann, AtLeast(3, Sohn), AtMost(2, Tochter),$$
$$All(Sohn, And(Arbeitslos, Verheiratet, All(Gattin, Doktor)))$$
$$All(Tochter, And(Professor, Fills(Lehrstuhl, Physik, Mathematik)))).$$

23 Wie Sie wissen, lassen sich Vererbungsinformationen in semantischen Netzen durch geeignete Implikationssätze logisch erfassen. In dieser Übung betrachten wir, wie effizient die Verwendung derartiger Sätze für die Vererbung ist.

a. Betrachten Sie die Informationen in einem Gebrauchtwagenkatalog, wie etwa den Schwacke. Dort steht beispielsweise, dass ein Dodge Van von 1973 noch 575 € wert ist (oder vielleicht einmal war). Angenommen, diese gesamten Informationen (für 11.000 Modelle) sind in logischen Sätzen kodiert, wie in diesem Kapitel vorgeschlagen. Schreiben Sie drei solcher Sätze auf, einschließlich der für Dodge Vans Baujahr 1973. Wie würden Sie die Sätze verwenden, um den Wert eines *bestimmten* Autos zu ermitteln, wenn Ihnen ein Rückwärtsverkettungs-Theorembeweiser wie etwa Prolog zur Verfügung stünde?

b. Vergleichen Sie die Zeiteffizienz der Rückwärtsverkettungsmethode für die Lösung dieses Problems mit der Vererbungsmethode in semantischen Netzen.

c. Erklären Sie, warum die Vorwärtsverkettung es ermöglicht, dass ein logikbasiertes System dasselbe Problem effizient löst, vorausgesetzt, die Wissensbasis enthält nur die 11.000 Sätze über die Preise.

d. Beschreiben Sie eine Situation, in der weder Vorwärts- noch Rückwärtsverkettung der Sätze eine effiziente Verarbeitung der Preisabfrage für einen einzelnen Wagen erlaubt.

e. Können Sie eine Lösung vorschlagen, mit der sich eine derartige Abfrage in allen logischen Systemen in allen Fällen effizient ausführen lässt? [*Hinweis*: Beachten Sie, dass zwei Autos desselben Modells aus demselben Jahr denselben Preis haben.]

24 Man könnte sagen, dass die syntaktische Unterscheidung zwischen nicht eingerahmten Verknüpfungen und einfach eingerahmten Verknüpfungen in semantischen Netzen überflüssig ist, weil einfach eingerahmte Verknüpfungen immer Kategorien zugeordnet sind; ein Vererbungsalgorithmus könnte einfach davon ausgehen, dass eine nicht eingerahmte Verknüpfung, die einer Kategorie zugeordnet ist, sich auf alle Elemente dieser Kategorie beziehen soll. Zeigen Sie, dass dieses Argument trügerisch ist, und nennen Sie Beispiele für Fehler, die daraus entstehen würden.

25 Eine vollständige Lösung für das Problem ungenauer Übereinstimmungen der Beschreibung des Käufers beim Einkaufen ist sehr schwierig und setzt die volle Palette von Techniken der Verarbeitung natürlicher Sprache und des Abrufens von Informationen voraus (siehe *Kapitel 22* und *23*). Ein kleiner Schritt ist es, dem Benutzer die Angabe von Minimal- und Maximalwerten für verschiedene Attribute zu erlauben. Der Käufer muss die folgende Grammatik für die Produktbeschreibung verwenden:

Beschreibung→*Kategorie [Verknüpfung Modifikator]**
Verknüpfung→"*mit*" | "*und*" | "*,*"
Modifikator→*Attribut* | *Attribut Operator Wert*
Operator→"=" | ">" | "<"

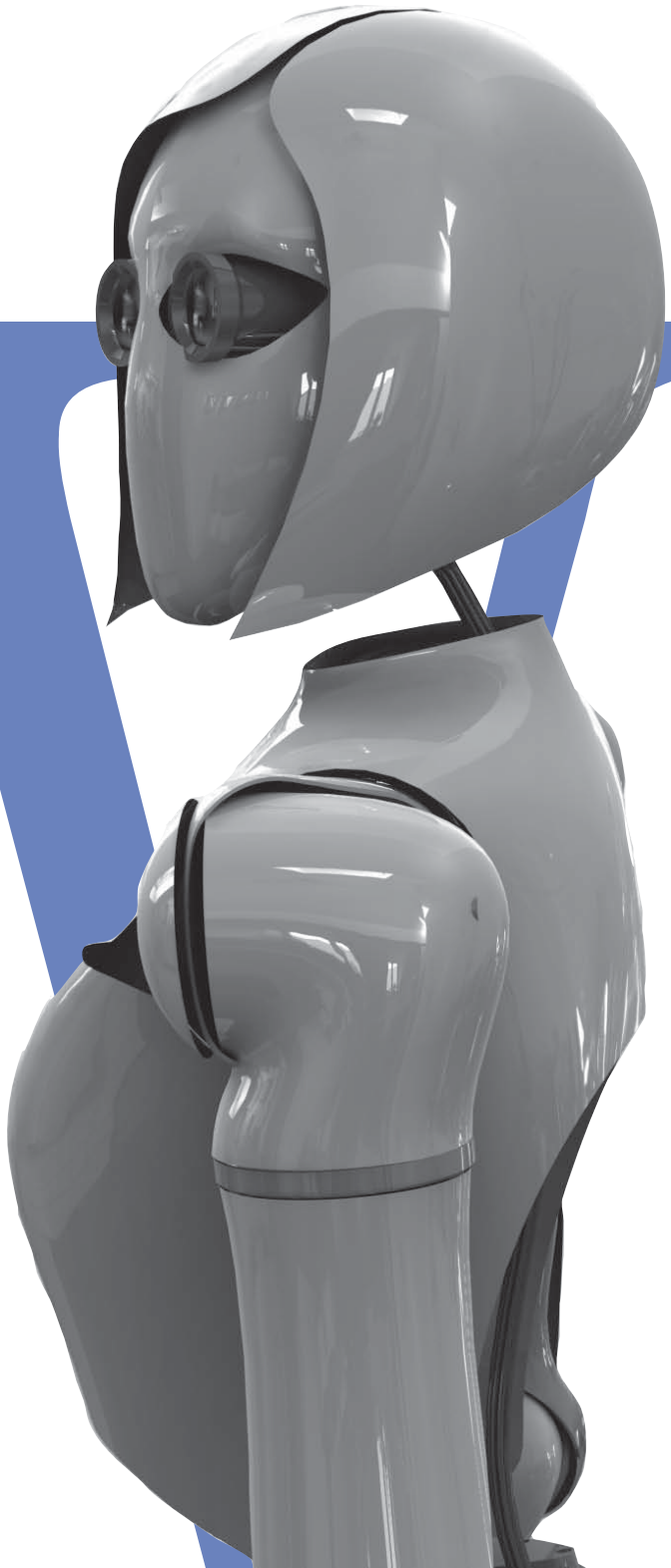
Hier bezeichnet *Kategorie* eine Produktkategorie, *Attribut* ist ein Funktionsmerkmal wie etwa „CPU" oder „Preis" und *Wert* ist der Zielwert für das Attribut. Die Abfrage „Computer mit mindestens 2,5 GHz CPU für unter 500 €" muss also umformuliert werden in „Computer mit CPU > 2,5 GHz und Preis < 500 €". Implementieren Sie einen Shopping-Agenten, der Beschreibungen in dieser Sprache akzeptiert.

26 Unsere Beschreibung des Internet-Shoppings hat den wichtigen Schritt des eigentlichen *Kaufens* des Produkts weggelassen. Entwickeln Sie eine formale logische Beschreibung des Kaufens unter Verwendung des Ereigniskalküls. Definieren Sie die Ereignissequenz, die auftritt, wenn ein Käufer einen Kreditkartenkauf tätigt und irgendwann sein Konto belastet wird und er das Produkt erhält.

TEIL IV

Unsicheres Wissen und Schließen

IV



Unsicherheit quantifizieren

13

ÜBERBLICK

In diesem Kapitel erfahren Sie, wie ein Agent Unsicherheit mithilfe von Glaubensgraden in den Griff bekommt.

13.1 Handeln unter Unsicherheit

Aufgrund von partieller Beobachtbarkeit und/oder Nichtdeterminismus müssen Agenten gegebenenfalls **Unsicherheit** verarbeiten. Ein Agent wird wohl nie bestimmt wissen, in welchem Zustand er sich befindet oder wo er sich nach einer Sequenz von Aktionen befinden wird.

Bisher haben Sie problemlösende Agenten (*Kapitel 4*) und logische Agenten (*Kapitel 7* und *11*) kennengelernt. Ein derartiger Agent verarbeitet Unsicherheit, indem er einen **Belief State** verfolgt – eine Darstellung der Menge aller möglichen Weltzustände, in denen er sich befinden kann – und einen Kontingenzplan erzeugt, der alle möglichen Eventualitäten verarbeitet, die seine Sensoren während der Ausführung melden können. Trotz seiner Vorzüge weist dieser Ansatz allerdings signifikante Nachteile auf, wenn man ihn buchstäblich als Rezept für das Erstellen von Agentenprogrammen übernimmt.

- Ein logischer Agent muss beim Interpretieren von partiellen Sensorinformationen jede logisch mögliche Erklärung für die Beobachtungen betrachten, egal wie unwahrscheinlich sie sind. Dies führt zu unmöglich großen und komplexen Belief-State-Darstellungen.
- Ein korrekter Kontingenzplan, der jede Eventualität verarbeitet, kann beliebig groß werden und muss beliebig unwahrscheinliche Zufälligkeiten berücksichtigen.
- Manchmal gibt es keinen Plan, der das Ziel garantiert erreicht – der Agent muss aber handeln. Es ist also eine Möglichkeit gefragt, um die Vorzüge von Plänen, die nicht garantiert sind, zu vergleichen.

Nehmen Sie zum Beispiel an, dass ein automatisiertes Taxi das Ziel hat, einen Passagier rechtzeitig zum Flughafen zu bringen. Der Agent bildet einen Plan, A_{90}, der besagt, dass die Wohnung 90 Minuten vor dem Abflug zu verlassen ist und das Taxi mit vernünftiger Geschwindigkeit fährt. Selbst wenn der Flughafen nur 15 km entfernt liegt, kann ein logischer Taxiagent nicht mit Bestimmtheit schließen, dass „Plan A_{90} uns pünktlich zum Flughafen bringt“. Stattdessen gelangt er zu der schwächeren Schlussfolgerung: „Plan A_{90} bringt uns rechtzeitig zum Flughafen, sofern das Auto nicht wegen eines Defekts oder Benzinmangels stehenbleibt, ich keinen Unfall habe und es keine Unfälle auf der Brücke gibt, das Flugzeug nicht zu früh startet, das Auto nicht von einem Meteoriten getroffen wird und ...“ Keine dieser Bedingungen kann sicher ermittelt werden, sodass sich der Erfolg des Planes nicht herleiten lässt. Dies ist das **Qualifikationsproblem** (*Abschnitt 7.7.1*), für das wir bisher noch keine wirkliche Lösung kennengelernt haben.

Tipp

Nichtsdestotrotz *ist* A_{90} in gewissem Sinn tatsächlich das Richtige. Was meinen wir damit? Wie in *Kapitel 2* besprochen, wollen wir damit ausdrücken, dass von allen Plänen, die ausgeführt werden könnten, A_{90} erwartungsgemäß das Leistungsmaß des Agenten maximiert (wobei die Erwartung relativ zum Wissen des Agenten über die Umgebung ist). Das Leistungsmaß beinhaltet die Fahrt zum Flughafen, sodass der Flug rechtzeitig erreicht wird, das Vermeiden langer, unproduktiver Wartezeiten auf dem Flughafen sowie das Vermeiden von Strafzetteln für zu schnelles Fahren auf dem Weg zum Flughafen. Das Wissen des Agenten kann keines dieser Ergebnisse für A_{90} garantieren, aber es kann einen bestimmten Glaubensgrad angeben, dass sie erreicht wer-

den. Andere Pläne, wie etwa A_{180}, könnten den Glauben des Agenten verstärken, dass er den Flughafen rechtzeitig erreicht, würden aber auch die Wahrscheinlichkeit einer sehr langen Wartezeit erhöhen. *Das Richtige – die* ***rationale Entscheidung*** *– ist also sowohl von der relativen Wichtigkeit der verschiedenen Ziele als auch von der Wahrscheinlichkeit, dass und in welchem Maß sie erzielt werden, abhängig.* Der restliche Abschnitt verfeinert diese Konzepte in Vorbereitung auf die Entwicklung der allgemeinen Theorien unsicheren Schließens sowie rationaler Entscheidungen, die wir in diesem und den folgenden Kapiteln vorstellen.

13.1.1 Unsicherheit zusammenfassen

Betrachten wir ein Beispiel für unsicheres Schließen: die Diagnose von Zahnschmerzen bei einem Zahnarztpatienten. Eine Diagnose – egal ob für Medizin, Autoreparatur oder was auch immer – beinhaltet fast immer Unsicherheit. Wir wollen versuchen, mithilfe von Aussagenlogik Regeln für eine zahnärztliche Diagnose zu schreiben, sodass wir erkennen, wie der logische Ansatz schrittweise vorgeht. Betrachten Sie die folgende einfache Regel:

$$\textit{Zahnschmerzen} \Rightarrow \textit{Loch}.$$

Das Problem dabei ist, dass diese Regel falsch ist. Nicht alle Patienten mit Zahnschmerzen haben Löcher; einige von ihnen haben Zahnfleischprobleme, einen Abszess oder eines von mehreren anderen Problemen:

$$\textit{Zahnschmerzen} \Rightarrow \textit{Loch} \vee \textit{Zahnfleischprobleme} \vee \textit{Abszess} \ldots$$

Leider müssen wir, um diese Regel wahr zu machen, eine nahezu unbegrenzte Liste möglicher Probleme hinzufügen. Wir könnten versuchen, die Regel in eine kausale Regel umzuwandeln:

$$\textit{Loch} \Rightarrow \textit{Zahnschmerzen}.$$

Diese Regel ist jedoch auch nicht richtig; nicht alle Löcher verursachen Zahnschmerzen. Die einzige Möglichkeit, die Regel zu korrigieren, ist, sie logisch erschöpfend zu formulieren: Die linke Seite muss mit allen Qualifikationen erweitert werden, die gelten müssen, damit ein Loch Zahnschmerzen verursacht. Versucht man, mithilfe von Logik eine Domäne wie die medizinische Diagnose in den Griff zu bekommen, schlägt das hauptsächlich aus drei Gründen fehl:

- **Faulheit:** Es macht zu viel Arbeit, die vollständige Menge der Antezedenzen oder Konsequenzen aufzulisten, die vorliegen müssen, um eine ausnahmslose Regel sicherzustellen, und es ist zu schwierig, solche Regeln anzuwenden.
- **Theoretisches Unwissen:** Die Medizin hat keine vollständige Theorie für die Domäne.
- **Praktisches Unwissen:** Selbst wenn wir alle Regeln kennen, können wir bei einem bestimmten Patienten noch unsicher sein, weil nicht alle erforderlichen Tests ausgeführt wurden oder ausgeführt werden können.

Die Verknüpfung zwischen Zahnschmerzen und Löchern ist keine logische Konsequenz in jede Richtung. Dies ist typisch für die Medizindomäne ebenso wie für alle anderen Domänen, in denen Beurteilungen erforderlich sind: Gesetz, Geschäft, Design, Autoreparatur, Gärtnern, Verabredungen usw. Das Wissen des Agenten kann bestenfalls eine Wahrscheinlichkeit in Bezug auf die relevanten Sätze bereitstellen. Unser wichtigstes

Werkzeug für den Umgang mit Glaubensgraden ist die **Wahrscheinlichkeitstheorie**. In der Terminologie von *Abschnitt 8.1* sind die **ontologischen Bindungen** von Logik und Wahrscheinlichkeitstheorie die gleichen – dass die Welt aus Fakten besteht, die in einem konkreten Fall gelten oder nicht –, die **erkenntnistheoretischen Bindungen** aber verschieden: Ein logischer Agent glaubt, dass jeder Satz wahr oder falsch ist, oder er hat keine Meinung, während ein probabilistischer Agent einen numerischen Glaubensgrad zwischen 0 (für Sätze, die sicher falsch sind) und 1 (sicher wahr) haben kann.

Tipp

*Die Wahrscheinlichkeit bietet eine Möglichkeit, die Unsicherheit **zusammenzufassen**, die aus Faulheit und Unwissen entsteht.* Wir wissen nicht sicher, was einem bestimmten Patienten fehlt, aber wir glauben zu 80% – d.h. mit einer Wahrscheinlichkeit von 0,8 –, dass ein Patient mit Zahnschmerzen ein Loch im Zahn hat. Wir erwarten demnach unter allen Situationen, die anhand unseres Wissens von der aktuellen Situation nicht zu unterscheiden sind, dass der Patient in 80% der Fälle ein Loch hat. Dieser Glaube könnte von statistischen Daten abgeleitet sein – 80% der bisher behandelten Zahnschmerzpatienten hatten Löcher – oder auch von allgemeinem zahnärztlichen Wissen oder aus einer Kombination von Indizien.

Ein verwirrender Punkt ist, dass es zum Zeitpunkt unserer Diagnose keine Unsicherheit in der tatsächlichen Welt gibt: Entweder hat der Patient ein Loch oder nicht. Was bedeutet es also zu sagen, dass die Wahrscheinlichkeit eines Loches 0,8 beträgt? Sollte sie nicht entweder 0 oder 1 sein? Die Antwort lautet, dass Wahrscheinlichkeitsaussagen im Hinblick auf einen Wissenszustand und nicht im Hinblick auf die reale Welt getroffen werden. Wir sagen: „Die Wahrscheinlichkeit, dass der Patient ein Loch hat, beträgt angesichts seiner Zahnschmerzen 0,8." Stellen wir später fest, dass der Patient in der Vergangenheit bereits mit Zahnfleischproblemen zu tun hatte, können wir unsere Aussage ändern: „Die Wahrscheinlichkeit, dass der Patient ein Loch hat, beträgt angesichts seiner Zahnschmerzen und der in der Vergangenheit aufgetretenen Zahnfleischprobleme 0,4." Wenn wir weitere schlüssige Beweise gegen ein Loch zusammentragen, können wir sagen: „Die Wahrscheinlichkeit, dass der Patient ein Loch hat, ist nach allem, was wir wissen, fast 0." Diese Aussagen widersprechen einander nicht; jede ist eine separate Behauptung über einen unterschiedlichen Wissenszustand.

13.1.2 Unsicherheit und rationale Entscheidungen

Betrachten Sie erneut den Plan A_{90}, um zum Flughafen zu gelangen. Angenommen, er gibt uns eine 97%-ige Chance, unseren Flug zu erreichen. Bedeutet das, dass es sich dabei um eine rationale Auswahl handelt? Nicht unbedingt: Es könnte andere Pläne geben, wie etwa A_{180}, mit höheren Wahrscheinlichkeiten. Wenn es lebenswichtig ist, den Flug zu erreichen, dann ist es zu rechtfertigen, auf dem Flughafen länger zu warten. Was ist mit A_{1440}, einem Plan, der vorgibt, 24 Stunden vor dem Abflug loszufahren? In den meisten Situationen ist das keine gute Wahl, denn obwohl der Plan nahezu garantiert, pünktlich zu sein, beinhaltet er auch eine nicht zu vertretende Wartezeit – ganz zu schweigen von einer möglichen unangenehmen Diät der Flughafenküche.

Um solche Entscheidungen zu treffen, muss ein Agent zunächst **Prioritäten** zwischen den verschiedenen möglichen **Ergebnissen** der verschiedenen Pläne setzen. Ein Ergebnis ist ein vollständig spezifizierter Zustand, einschließlich solcher Faktoren wie etwa, ob der Agent rechtzeitig ankommt und wie lange er auf dem Flughafen war-

ten muss. Wir verwenden die **Nutzentheorie**, um Prioritäten darzustellen und mit ihnen zu schließen. Die Nutzentheorie besagt, dass jeder Zustand einen Nutzengrad für einen Agenten hat und dass der Agent Zustände mit höherem Nutzen bevorzugt.

Der Nutzen eines Zustandes ist für einen Agenten relativ: Zum Beispiel ist der Nutzen eines Zustandes, in dem Weiß in einem Schachspiel Schwarz Schachmatt gesetzt hat, offensichtlich hoch für den Agenten, der Weiß spielt, aber gering für den Agenten, der Schwarz spielt. Doch wir können nicht streng nach den Punkten 1, 1/2 und 0 gehen, die durch die Turnierregeln von Schach vorgeschrieben werden – manche Spieler (einschließlich der Autoren) wären schon glücklich, wenn sie ein Remis gegen den Weltmeister erzielen könnten, während andere Spieler (wie etwa der frühere Weltmeister) das vielleicht nicht sind. Es gibt keine Bewertung von Geschmack oder Prioritäten: Sie könnten vielleicht denken, ein Agent, der lieber Schlumpfeis als Schokoladeneis isst, leide unter Geschmacksverirrung, aber Sie können nicht sagen, der Agent sei irrational. Eine Nutzenfunktion kann jede Menge von Prioritäten berücksichtigen – schrullig oder normal, vornehm oder widernatürlich. Und sie kann sogar altruistisches Verhalten berücksichtigen, indem einfach das Wohlergehen der anderen als einer der Faktoren gezählt wird.

Durch den Nutzen ausgedrückte Prioritäten werden in der allgemeinen Theorie der rationalen Entscheidungen – der sogenannten **Entscheidungstheorie** – mit Wahrscheinlichkeiten kombiniert:

Entscheidungstheorie = *Wahrscheinlichkeitstheorie* + *Nutzentheorie*.

Tipp

Das grundlegende Konzept der Entscheidungstheorie ist, *dass ein Agent genau dann rational ist, wenn er die Aktion auswählt, die den höchsten erwarteten Nutzen erbringt, gemittelt über alle möglichen Ergebnisse der Aktion*. Man spricht auch vom Prinzip des **maximalen erwarteten Nutzens** (MEN). Zwar mag „erwartet“ wie ein verschwommener, hypothetischer Term erscheinen, er wird hier aber in einer genauen Bedeutung verwendet: Es ist darunter der „Durchschnitt“ oder das „statistische Mittel“ der Ergebnisse, gewichtet nach der Wahrscheinlichkeit des Ergebnisses, zu verstehen. Wir haben dieses Prinzip bereits in *Kapitel 5* kennengelernt, wo wir kurz über optimale Entscheidungen im Backgammon gesprochen haben. Es handelt sich in der Tat um ein völlig allgemeines Prinzip.

▶ Abbildung 13.1 skizziert die Struktur eines Agenten, der die Entscheidungstheorie einsetzt, um Aktionen auszuwählen. Der Agent ist auf abstrakter Ebene identisch mit den in den *Kapiteln 4* und *7* beschriebenen Agenten, die einen Belief State mit den bisherigen Wahrnehmungen verwalten. Der wichtigste Unterschied ist, dass der Belief State des entscheidungstheoretischen Agenten nicht einfach die *Möglichkeiten* für Weltzustände, sondern auch deren *Wahrscheinlichkeiten* darstellt. Anhand des Belief State kann der Agent probabilistische Vorhersagen für Aktionsergebnisse treffen und damit die Aktion auswählen, die den höchsten erwarteten Nutzen erbringt. Dieses und das nächste Kapitel konzentrieren sich auf die Aufgabe, probabilistische Informationen im Allgemeinen zu repräsentieren und mit ihnen zu rechnen. *Kapitel 15* beschäftigt sich mit Methoden für die speziellen Aufgaben, den Belief State über der Zeit zu repräsentieren und zu aktualisieren sowie die Umgebung vorherzusagen. *Kapitel 16* geht detailliert auf die Nutzentheorie ein und *Kapitel 17* entwickelt Algorithmen für Planungssequenzen von Aktionen in unsicheren Umgebungen.

```
function DT-AGENT(percept) returns eine Aktion
  persistent: belief_state, prob. Glauben über aktuellen Zustand der Welt
              action, die Aktion des Agenten
  aktualisiere belief_state basierend auf action und percept
  berechne Ergebniswahrscheinlichkeiten für Aktionen bei gegebenen
    Aktionsbeschreibungen und aktuellem belief_state
  wähle action mit höchstem erwarteten Nutzen für gegebene
    Wahrscheinlichkeiten von Ergebnissen und Nutzeninformationen aus
  return action
```

Abbildung 13.1: Ein entscheidungstheoretischer Agent, der rationale Aktionen auswählt.

13.2 Grundlegende Notation für die Wahrscheinlichkeit

Damit unser Agent probabilistische Informationen darstellen und verwenden kann, brauchen wir eine formale Sprache. Die Sprache der Wahrscheinlichkeitstheorie ist traditionell formlos gewesen, geschrieben von menschlichen Mathematikern für andere menschliche Mathematiker. Anhang A enthält eine Standardeinführung in elementare Wahrscheinlichkeitstheorie; hier nehmen wir einen Ansatz, der den Anforderungen der KI eher entgegenkommt und mit den Konzepten formaler Logik konsistent ist.

13.2.1 Das Wesen von Wahrscheinlichkeiten

Wie bei logischen Behauptungen dreht es sich bei probabilistischen Behauptungen um mögliche Welten. Während logische Behauptungen aussagen, welche möglichen Welten strikt ausgeschlossen werden (alle diejenigen, in denen die Behauptung falsch ist), sprechen probabilistische Behauptungen darüber, wie beweisbar die verschiedenen Welten sind. In der Wahrscheinlichkeitstheorie bezeichnet man die Menge aller möglichen Welten als **Stichprobenraum**. Die möglichen Welten sind *gegenseitig ausschließend* und *erschöpfend* – zwei mögliche Welten können nicht beide der Fall sein und eine mögliche Welt muss der Fall sein. Wenn wir zum Beispiel zwei (unterscheidbare) Würfel werfen, sind 36 mögliche Welten zu betrachten: (1,1), (1,2), ..., (6,6). Den Stichprobenraum bezeichnet man mit Ω (dem griechischen Großbuchstaben Omega) und verweist mit ω (dem kleinen Omega) auf Elemente des Raumes, d.h. auf konkrete mögliche Welten.

Ein vollständig spezifiziertes **Wahrscheinlichkeitsmodell** weist jeder möglichen Welt eine numerische Wahrscheinlichkeit $P(\omega)$ zu.[1] Die grundlegenden Axiome der Wahrscheinlichkeitstheorie besagen, dass jede mögliche Welt eine Wahrscheinlichkeit zwischen 0 und 1 hat und dass die Gesamtwahrscheinlichkeit der Menge möglicher Welten gleich 1 ist:

$$0 \leq P(\omega) \leq 1 \text{ für jedes } \omega \text{ und } \sum_{\omega \in \Omega} P(\omega) = 1. \tag{13.1}$$

1 Wir nehmen hier eine diskrete, abzählbare Menge von Welten an. Die ordnungsgemäße Behandlung des stetigen Falles bringt bestimmte Komplikationen mit sich, die für die meisten Zwecke in der KI weniger relevant sind.

Unter der Annahme, dass jeder Würfel fair ist und die Würfe sich untereinander nicht stören, hat jede mögliche Welt (1,1), (1,2), ..., (6,6) die Wahrscheinlichkeit 1/36. Wenn dagegen die Würfel konspirieren, um dieselbe Augenzahl zu liefern, können die Welten (1,1), (2,2), (3,3) usw. höhere Wahrscheinlichkeiten haben, wobei die Wahrscheinlichkeiten der übrigen Welten geringer sind.

Probabilistische Behauptungen und Abfragen werden üblicherweise nicht über konkrete mögliche Welten, sondern über Mengen möglicher Welten getroffen. Zum Beispiel könnten wir an Fällen interessiert sein, in denen zwei Würfel zusammen die Augenzahl 11 ergeben, in denen ein Pasch gewürfelt wird usw. In der Wahrscheinlichkeitstheorie bezeichnet man diese Mengen als **Ereignisse** – ein Begriff, der bereits ausgiebig in *Kapitel 12* für ein anderes Konzept verwendet wurde. In der KI beschreibt man die Mengen immer als **Aussagen** in einer formalen Sprache. (*Abschnitt 13.2.2* gibt eine derartige Sprache an.) Für jede Aussage enthält die korrespondierende Menge lediglich diejenigen Welten, in denen die Aussage gilt. Die einer Aussage zugeordnete Wahrscheinlichkeit ist als Summe der Wahrscheinlichkeiten der Welten, in denen die Aussage gilt, definiert:

$$\text{Für jede Aussage } \phi \text{ ist } P(\phi) = \sum_{\omega \in \phi} P(\omega). \tag{13.2}$$

Wenn wir mit zwei fairen Würfeln spielen, haben wir $P(Gesamt = 11) = P((5,6)) + P((6,5)) = 1/36 + 1/36 = 1/18$. Die Wahrscheinlichkeitstheorie verlangt keine vollständigen Kenntnisse der Wahrscheinlichkeiten jeder möglichen Welt. Nehmen wir beispielsweise an, dass die Würfel konspirieren, um dieselbe Augenzahl zu liefern, können wir *behaupten*, dass $P(Pasch) = 1/4$ ist, ohne zu wissen, ob die Würfel einen 6er-Pasch oder einen 2er-Pasch bevorzugen. Genau wie bei logischen Behauptungen *beschränkt* diese Behauptung das zugrunde liegende Wahrscheinlichkeitsmodell, ohne es vollständig zu bestimmen.

Wahrscheinlichkeiten wie zum Beispiel $P(Gesamt = 11)$ und $P(Pasch)$ heißen **unbedingte** oder **A-priori-Wahrscheinlichkeiten**; sie beziehen sich auf Glaubensgrade in Aussagen bei Fehlen irgendwelcher anderer Informationen. Die meiste Zeit allerdings haben wir bestimmte Informationen, sogenannte **Indizien**, die bereits aufgedeckt wurden. Zum Beispiel könnte der erste Würfel bereits eine 5 zeigen und wir warten mit angehaltenem Atem darauf, dass der andere Würfel endlich ausrollt. In diesem Fall sind wir nicht an der unbedingten Wahrscheinlichkeit eines gewürfelten Paschs, sondern an der **bedingten** oder **A-posteriori-Wahrscheinlichkeit** interessiert, dass ein Pasch erscheint, wenn der erste Würfel eine 5 zeigt. Diese Wahrscheinlichkeit wird als $P(Pasch \mid Würfel_1 = 5)$ geschrieben, wobei der senkrechte Strich („|“) als „unter der Voraussetzung“ zu lesen ist. Ähnlich verhält es sich, wenn ich wegen einer regelmäßigen Kontrolluntersuchung zum Zahnarzt gehe. Hier beträgt die Wahrscheinlichkeit $P(Loch) = 0{,}2$. Doch wenn ich den Zahnarzt aufsuche, weil ich Zahnschmerzen habe, kommt eine Wahrscheinlichkeit $P(Loch \mid Zahnschmerzen) = 0{,}6$ ins Spiel. Für den Vorrang von „|“ gilt, dass jeder Ausdruck der Form $P(... \mid ...)$ immer $P((...) \mid (...))$ bedeutet.

Wichtig ist, dass $P(Loch) = 0{,}2$ immer noch *gültig* ist, nachdem *Zahnschmerzen* beobachtet wurden; es ist nur nicht besonders nützlich. Wenn ein Agent Entscheidungen trifft, muss er *alle* Indizien konditionieren, die er beobachtet hat. Außerdem ist der Unterschied zwischen materialer Implikation (Konditional) und logischer Implikation wichtig. Die Behauptung, dass $P(Loch \mid Zahnschmerzen) = 0{,}6$ ist, bedeutet nicht: „Wenn *Zahnschmerzen* wahr ist, immer schlussfolgern, dass *Loch* mit einer Wahrscheinlichkeit von

0,6 wahr ist“, sondern: „Wenn *Zahnschmerzen* wahr ist und *wir keine weiteren Informationen haben*, schlussfolgern, dass *Loch* mit einer Wahrscheinlichkeit von 0,6 wahr ist.“ Diese zusätzliche Bedingung ist wichtig; wenn wir zum Beispiel die ergänzende Information hätten, dass der Zahnarzt keine Löcher gefunden hat, würden wir zweifellos nicht schließen, dass *Loch* mit einer Wahrscheinlichkeit von 0,6 wahr ist, sondern müssen stattdessen $P(Loch \mid Zahnschmerzen \wedge \neg Loch) = 0$ verwenden.

Im mathematischen Sprachgebrauch werden bedingte Wahrscheinlichkeiten wie folgt als unbedingte Wahrscheinlichkeiten definiert: Für beliebige Aussagen a und b haben wir

$$P(a \mid b) = \frac{P(a \wedge b)}{P(b)} . \tag{13.3}$$

Das gilt immer, wenn $P(b) > 0$ ist. Zum Beispiel

$$P(Pasch \mid Würfel_1 = 5) = \frac{P(Pasch \wedge Würfel_1 = 5)}{P(Würfel_1 = 5)} .$$

Die Definition ist sinnvoll, wenn man berücksichtigt, dass die Beobachtung von b alle diejenigen möglichen Welten ausschließt, in denen b falsch ist, wodurch eine Menge zurückbleibt, deren Gesamtwahrscheinlichkeit lediglich $P(b)$ ist. In dieser Menge erfüllen die a-Welten $a \wedge b$ und bilden einen Anteil $P(a \wedge b)/P(b)$.

Die in Gleichung (13.3) angegebene Definition der bedingten Wahrscheinlichkeit lässt sich in anderer Form als sogenannte **Produktregel** schreiben:

$$P(a \wedge b) = P(a \mid b)\, P(b).$$

Die Produktregel lässt sich vielleicht einfacher merken: Sie geht zurück auf die Tatsache, dass b wahr sein muss, damit a und b wahr sind, und bei gegebenem b auch a wahr sein muss.

13.2.2 Die Sprache der Aussagen in Wahrscheinlichkeitsbehauptungen

In diesem und dem nächsten Kapitel werden Aussagen, die Mengen möglicher Welten beschreiben, in einer Notation formuliert, die Elemente der Aussagenlogik und der Notation von Erfüllbarkeitsproblemen kombiniert. In der Terminologie von *Abschnitt 2.4.7* handelt es sich um eine **faktorisierte Darstellung**, in der eine mögliche Welt durch eine Menge von Variablen/Wert-Paaren dargestellt wird.

Bei Variablen in der Wahrscheinlichkeitstheorie spricht man von **Zufallsvariablen** und schreibt ihre Namen mit einem großen Anfangsbuchstaben. Somit handelt es sich bei *Gesamt* und $Würfel_1$ im Würfelbeispiel um Zufallsvariablen. Jede Zufallsvariable besitzt eine **Domäne** – die Menge der möglichen Werte, die sie annehmen kann. Die Domäne von *Gesamt* für die beiden Würfel ist die Menge {2, ..., 12} und die Domäne von $Würfel_1$ ist {1, ..., 6}. Eine boolesche Zufallsvariable hat die Domäne {*true*, *false*} (wobei zu beachten ist, dass die Werte immer klein geschrieben werden); zum Beispiel lässt sich die Aussage, dass ein Pasch gewürfelt wird, als $Pasch = true$ schreiben. Per Konvention kürzt man Aussagen der Form $A = true$ einfach als a ab, während $A = false$ in Kurzform als $\neg a$ geschrieben wird. (Die Verwendungen von *Pasch*, *Loch* und *Zahnschmerzen* im obigen Abschnitt sind Abkürzungen dieser Art.) Wie in CSPs können Domänen Mengen beliebiger Token sein; wir könnten die Domäne von *Alter* mit {*jugendlich*, *teen*, *erwachsen*} festlegen und die Domäne von *Wetter* könnte {*sonnig*, *regnerisch*, *wolkig*, *schnee*} lauten. Ist keine Mehrdeutigkeit möglich, verwendet man üblicher-

weise einen Wert an sich für die Aussage, dass eine bestimmte Variable diesen Wert hat; somit kann *sonnig* für *Wetter* = *sonnig* stehen.

Die obigen Beispiele haben alle endliche Domänen. Variablen können aber auch unendliche Domänen besitzen – entweder diskret (wie bei Ganzzahlen) oder stetig (wie bei Realzahlen). Für eine Variable mit einer geordneten Domäne sind Ungleichungen wie zum Beispiel $AnzahlDerAtomeImUniversum \geq 1070$ ebenfalls erlaubt.

Schließlich können wir diese elementaren Aussagen (einschließlich der abgekürzten Formen für boolesche Variablen) mithilfe der Verknüpfungen der Aussagenlogik kombinieren. Zum Beispiel lässt sich die Feststellung „Die Wahrscheinlichkeit, dass der Patient ein Loch hat, da er ein Teenager ohne Zahnschmerzen ist, beträgt 0,1." wie folgt ausdrücken:

$$P(loch \mid \neg zahnschmerzen \wedge teen) = 0{,}1.$$

Manchmal möchten wir über die Wahrscheinlichkeiten *aller* möglichen Werte einer Zufallsvariablen sprechen. Wir könnten

$$P(Wetter = sonnig) = 0{,}6$$
$$P(Wetter = regen) = 0{,}1$$
$$P(Wetter = wolkig) = 0{,}29$$
$$P(Wetter = schnee\ ) = 0{,}01$$

schreiben, lassen aber als Abkürzung

$$\mathbf{P}(Wetter) = \langle 0{,}6;\ 0{,}1;\ 0{,}29;\ 0{,}01\rangle$$

zu, wobei das fett geschriebene **P** anzeigt, dass das Ergebnis ein Vektor von Zahlen ist, und wo wir eine vordefinierte Reihenfolge ⟨*sonnig*, *regen*, *wolkig*, *schnee*⟩ in der Domäne von Wetter annehmen. Wir sagen, dass die **P**-Anweisung eine **Wahrscheinlichkeitsverteilung** für die Zufallsvariable *Wetter* definiert. Die **P**-Notation wird auch für bedingte Verteilungen verwendet: $\mathbf{P}(X \mid Y)$ liefert die Werte von $P(X = x_i \mid Y = y_j)$ für jedes mögliche *i*, *j*-Paar.

Für stetige Variablen ist es nicht möglich, die gesamte Verteilung als Vektor zu schreiben, da es sich um unendlich viele Werte handelt. Stattdessen können wir die Wahrscheinlichkeit, dass eine Zufallsvariable einen bestimmten Wert x annehmen kann, als parametrisierte Funktion von x definieren. Zum Beispiel drückt der Satz

$$P(MittagsTemp = x) = Gleichförmig_{[18C,26C]}(x)$$

den Glauben aus, dass die Temperaturen zur Mittagszeit zwischen 18 und 26 Grad Celsius gleichförmig verteilt sind. Wir bezeichnen dies als **Wahrscheinlichkeitsdichtefunktion**.

Wahrscheinlichkeitsdichtefunktionen unterscheiden sich in ihrer Bedeutung von diskreten Verteilungen. Wenn man sagt, dass die Wahrscheinlichkeitsdichte im Bereich von $18C$ bis $26C$ gleichförmig ist, heißt das, dass es eine 100%ige Chance gibt, dass die Temperatur in diesen $8C$ breiten Bereich fällt, und eine 50%ige Chance, dass sie in einen beliebigen $4C$ breiten Bereich fällt, usw. Wir schreiben die Wahrscheinlichkeitsdichte für eine stetige Zufallsvariable X am Wert x als $P(X = x)$ oder einfach $P(x)$. Die intuitive Definition von $P(x)$ ist die Wahrscheinlichkeit, dass X in einen beliebig kleinen Bereich fällt, der bei x beginnt und durch die Breite des Bereiches geteilt wird:

$$P(x) = \lim_{dx \to 0} P(x \leq X \leq x + dx) / dx.$$

Für *MittagsTemp* haben wir

$$P(MittagsTemp = x) = Gleichförmig_{[18C,26C]}(x) = \begin{cases} \frac{1}{8C}, \text{ wenn } 18C \le x \le 26C, \\ 0 \text{ andernfalls} \end{cases}$$

wobei hier C für Grad Celsius (nicht für eine Konstante) steht. In $P(MittagsTemp = 20{,}18C) = 1/8C$ ist der Term $1/8C$ keine Wahrscheinlichkeit, sondern eine Wahrscheinlichkeitsdichte. Die Wahrscheinlichkeit, dass *MittagsTemp* genau $20{,}18C$ ist, beträgt 0, da $20{,}18C$ ein Bereich der Breite 0 ist. Einige Autoren verwenden andere Symbole für diskrete Verteilungen und Dichtefunktionen; wir verwenden in beiden Fällen P, weil selten Verwechslungen auftreten und die Gleichungen normalerweise identisch sind. Beachten Sie, dass diese Wahrscheinlichkeiten einheitenlose Zahlen sind, während Dichtefunktionen mit einer Einheit angegeben werden, in diesem Fall reziprokem Grad.

Außer für Verteilungen von einzelnen Variablen brauchen wir eine Notation für Verteilungen auf mehrere Variablen. Hierfür werden Kommas verwendet. Zum Beispiel kennzeichnet $\mathbf{P}(Wetter, Loch)$ die Wahrscheinlichkeiten aller Kombinationen der Werte von *Wetter* und *Loch*. Dies ist eine 4×2-Tabelle mit Wahrscheinlichkeiten, die als **gemeinsame Wahrscheinlichkeitsverteilung** von *Wetter* und *Loch* bezeichnet wird. Wir können auch Variablen mit und ohne Werte mischen; $\mathbf{P}(sonnig, Loch)$ wäre ein Vektor mit zwei Elementen, der die Wahrscheinlichkeiten für einen sonnigen Tag mit einem Loch und einen sonnigen Tag ohne Loch angibt.

Die **P**-Notation erlaubt es, bestimmte Ausdrücke wesentlich prägnanter zu formulieren, als es sonst möglich wäre. Zum Beispiel lassen sich die Produktregeln für alle möglichen Werte von *Wetter* und *Loch* als einzige Gleichung

$$\mathbf{P}(Wetter, Loch) = \mathbf{P}(Wetter \mid Loch)\mathbf{P}(Loch)$$

anstelle der folgenden $4 \times 2 = 8$ Gleichungen (mit den Abkürzungen W und C) schreiben:

$$\begin{aligned}
P(W = sonnig \wedge C = true) &= P(W = sonnig | C = true)\, P(C = true) \\
P(W = regen \wedge C = true) &= P(W = regen | C = true)\, P(C = true) \\
P(W = wolkig \wedge C = true) &= P(W = wolkig | C = true)\, P(C = true) \\
P(W = schnee \wedge C = true) &= P(W = schnee | C = true)\, P(C = true) \\
P(W = sonnig \wedge C = false) &= P(W = sonnig | C = false)\, P(C = false) \\
P(W = regen \wedge C = false) &= P(W = regen | C = false)\, P(C = false) \\
P(W = wolkig \wedge C = false) &= P(W = wolkig | C = false)\, P(C = false) \\
P(W = schnee \wedge C = false) &= P(W = schnee | C = false)\, P(C = false).
\end{aligned}$$

Als degenerierter Fall besitzt $\mathbf{P}(sonnig, loch)$ keine Variablen und ist somit ein einelementiger Vektor, der die Wahrscheinlichkeit eines sonnigen Tages mit einem Loch angibt, was sich auch als $P(sonnig, loch)$ oder $P(sonnig \wedge loch)$ schreiben lässt. Wir verwenden manchmal die **P**-Notation, um Ergebnisse über individuelle P-Werte abzuleiten, und wenn wir sagen „$\mathbf{P}(sonnig) = 0{,}6$“, ist das eigentlich eine Abkürzung für „$\mathbf{P}(sonnig)$ ist der einelementige Vektor $(0{,}6)$, was bedeutet, dass $P(sonnig) = 0{,}6$ ist“.

Tipp

Wir haben nun eine Syntax für Aussagen und Wahrscheinlichkeitsbehauptungen definiert und einen Teil der Semantik angegeben: Gleichung (13.2) definiert die Wahrscheinlichkeit einer Aussage als Summe der Wahrscheinlichkeiten von Welten, in denen sie gilt. Um die Semantik zu vervollständigen, müssen wir angeben, was die Welten sind und wie zu ermitteln ist, ob eine Aussage in einer Welt gilt. Wir borgen

uns diesen Teil direkt von der Semantik der Aussagenlogik aus: *Eine mögliche Welt ist definiert als Zuweisung von Werten zu allen betrachteten Zufallsvariablen.* Es lässt sich leicht zeigen, dass diese Definition die grundlegende Anforderung erfüllt, dass mögliche Welten gegenseitig ausschließend und erschöpfend sind (Übung 13.5). Wenn zum Beispiel die Zufallsvariablen Loch, Zahnschmerzen und Wetter sind, gibt es $2 \times 2 \times 4 = 16$ mögliche Welten. Darüber hinaus lässt sich die Wahrheit jeder gegebenen Position in derartigen Welten mithilfe der gleichen rekursiven Definition von Wahrheit wie bei Formeln in Aussagenlogik leicht ermitteln.

Aus der obigen Definition aller möglichen Welten folgt, dass ein probabilistisches Modell durch die gemeinsame Verteilung für alle Zufallsvariablen bestimmt wird – die sogenannte **vollständige gemeinsame Wahrscheinlichkeitsverteilung**. Zum Beispiel wird die vollständige gemeinsame Wahrscheinlichkeitsverteilung für die Variablen *Loch*, *Zahnschmerzen* und *Wetter* durch $\mathbf{P}(Loch, Zahnschmerzen, Wetter)$ angegeben. Diese gemeinsame Verteilung kann als $2 \times 2 \times 4$-Tabelle mit 16 Einträgen dargestellt werden. Da die Wahrscheinlichkeit jeder Aussage eine Summe über mögliche Welten ist, reicht eine vollständige gemeinsame Verteilung im Prinzip aus, um die Wahrscheinlichkeit jeder beliebigen Aussage zu berechnen.

13.2.3 Warum Wahrscheinlichkeitsaxiome sinnvoll sind

Die grundlegenden Axiome der Wahrscheinlichkeit (Gleichungen (13.1) und (13.2)) implizieren bestimmte Beziehungen unter den Glaubensgraden, die logisch verwandten Aussagen zugesprochen werden können. Zum Beispiel können wir die bekannte Beziehung zwischen der Wahrscheinlichkeit einer Aussage und der Wahrscheinlichkeit ihrer Negation ableiten:

$$\begin{aligned} P(\neg a) &= \Sigma_{\omega \in \neg a} P(\omega) && \text{nach Gleichung (13.2)} \\ &= \Sigma_{\omega \in \neg a} P(\omega) + \Sigma_{\omega \in a} P(\omega) - \Sigma_{\omega \in a} P(\omega) \\ &= \Sigma_{\omega \in \Omega} P(\omega) - \Sigma_{\omega \in a} P(\omega) && \text{Gruppieren der beiden ersten Terme} \\ &= 1 - P(a) && \text{nach (13.1) und (13.2)} \end{aligned}$$

Wir können auch die wohlbekannte Formel für die Wahrscheinlichkeit einer Disjunktion herleiten, die manchmal als **Prinzip von Inklusion und Exklusion** bezeichnet wird:

$$P(a \vee b) = P(a) + P(b) - P(a \wedge b). \tag{13.4}$$

Diese Regel kann man sich leicht merken, wenn man erkennt, dass die Fälle, in denen a gilt, zusammen mit den Fällen, in denen b gilt, mit Sicherheit alle Fälle abdecken, wo $a \vee b$ gilt. Fasst man aber die beiden Fallmengen zusammen, werden ihre Schnittstellen doppelt gezählt; deshalb müssen wir $P(a \wedge b)$ subtrahieren. Der Beweis bleibt dem Leser als Übung überlassen (Übung 13.6).

Die Gleichungen (13.1) und (13.4) werden häufig als **Kolmogorov-Axiome** bezeichnet, nach dem russischen Mathematiker Andrej Kolmogorov, der zeigte, wie sich die restliche Wahrscheinlichkeitstheorie auf dieser einfachen Grundlage aufbauen lässt und wie die Schwierigkeiten zu bewältigen sind, die durch stetige Variablen entstehen.[2]

2 Zu den Schwierigkeiten gehört die Vitali-Menge, eine wohldefinierte Teilmenge des Intervalls [0, 1] ohne wohldefinierte Größe.

Während Gleichung (13.2) den Charakter einer Definition hat, macht Gleichung (13.4) deutlich, dass die Axiome wirklich die Glaubensgrade beschränken, die ein Agent in Bezug auf logisch verwandte Aussagen besitzen kann. Dies ist analog zur Tatsache, dass ein logischer Agent nicht gleichzeitig A, B und $\neg(A \wedge B)$ glauben kann, weil es keine mögliche Welt gibt, in der alle drei Aussagen wahr sind. Mit Wahrscheinlichkeiten dagegen beziehen sich die Aussagen nicht direkt auf die Welt, sondern auf den eigenen Wissenszustand des Agenten. Warum kann dann ein Agent nicht die folgende Glaubensmenge besitzen (selbst wenn sie die Axiome von Kolmogorov verletzt)?

$$\begin{aligned} P(a) &= 0{,}4 & P(a \wedge b) &= 0{,}0 \\ P(b) &= 0{,}3 & P(a \vee b) &= 0{,}8. \end{aligned} \tag{13.5}$$

Diese Art Frage war Thema jahrzehntelanger Debatten zwischen den Fürsprechern der Verwendung von Wahrscheinlichkeiten als einzige legitime Form für Glaubensgrade und den Vertretern der alternativen Ansätze.

Ein Argument für die Wahrscheinlichkeitsaxiome, das zuerst 1931 von Bruno de Finetti formuliert wurde, lautet wie folgt: Wenn ein Agent einen Glaubensgrad in Bezug auf eine Aussage a hat, sollte er in der Lage sein, die Chancen anzugeben, für die es keinen Unterschied ergibt, für oder gegen a zu sein.[3] Stellen Sie sich das Ganze als Spiel zwischen zwei Agenten vor: Agent 1 sagt: „Mein Glaubensgrad in Bezug auf das Ereignis a ist 0,4." Agent 2 kann dann frei wählen, ob er für oder gegen a ist, und zwar für einen Einsatz, der konsistent mit dem angegebenen Glaubensgrad ist. Das bedeutet, Agent 2 könnte die Wette von Agent 1 annehmen, dass a eintritt und dabei 6 € gegen die 4 € von Agent 1 setzen. Agent 2 könnte auch die Wette von Agent 1 annehmen, dass $\neg a$ eintritt, und 4 € gegen die 6 € von Agent 1 setzen. Dann beobachten wir das Ergebnis von a und wer recht hat, sammelt das Geld ein. Wenn die Glaubensgrade eines Agenten die Welt nicht exakt reflektieren, erwartet man, dass er auf lange Sicht Geld an einen gegnerischen Agenten verliert, dessen Glaubensgrade den Zustand der Welt genauer reflektieren.

Tipp

De Finetti hat jedoch etwas viel Stärkeres bewiesen: *Wenn Agent 1 eine Menge von Glaubensgraden ausdrückt, die die Axiome der Wahrscheinlichkeitstheorie verletzen, gibt es eine Wettkombination für Agent 2, die garantiert, dass Agent 1 immer Geld verliert.* Nehmen Sie zum Beispiel an, dass Agent 1 die Menge der Glaubensgrade von Gleichung (13.5) hat. ▶ Abbildung 13.2 zeigt, dass, wenn Agent 2 4 € auf a, 3 € auf b und 2 € auf $\neg(a \vee b)$ wettet, Agent 1 immer Geld verliert – unabhängig von den Ergebnissen für a und b. Das Theorem von de Finetti impliziert, dass kein rationaler Agent Glauben haben kann, die die Axiome der Wahrscheinlichkeit verletzen.

Ein häufiger Vorbehalt gegen das Theorem von de Finetti ist, dass dieses Wettspiel ziemlich gekünstelt ist. Wie sieht es beispielsweise aus, wenn man sich weigert zu wetten? Wird das Argument damit beendet? Die Antwort ist, dass das Wettspiel ein abstraktes Modell für die Entscheidungssituation ist, in der sich jeder Agent *unvermeidbar* zu jedem Zeitpunkt befindet. Jede Aktion (auch die Inaktivität) ist eine Art Wette und jedes Ergebnis kann als Abrechnung der Wette betrachtet werden. Die Weigerung zu wetten ist, als würde man sich weigern, dass die Zeit vergeht.

3 Man könnte sagen, dass die Prioritäten des Agenten für unterschiedliche Kontostände so gelagert sind, dass die Möglichkeit, 1 € zu verlieren, durch die gleiche Wahrscheinlichkeit ausgeglichen wird, 1 € zu gewinnen. Eine mögliche Reaktion ist, die Wettbeträge so klein zu machen, dass dieses Problem vermieden wird. Die Analyse von Savage (1954) umgeht diesen gesamten Aspekt.

Woher kommen Wahrscheinlichkeiten?

Es gab endlose Debatten über den Ursprung und den Status von Wahrscheinlichkeitszahlen. Die Position der **Frequentisten** besagt, dass die Zahlen nur aus *Experimenten* stammen: Wenn wir 100 Menschen testen und feststellen, dass zehn von ihnen ein Loch im Zahn haben, können wir sagen, die Wahrscheinlichkeit eines Lochs beträgt etwa 0,1. Aus dieser Perspektive bedeutet die Zusicherung „Die Wahrscheinlichkeit eines Lochs beträgt 0,1", dass 0,1 der Bruchteil ist, der im Rahmen von unendlich vielen Stichproben beobachtet wird. Aus einem endlichen Beispiel können wir den tatsächlichen Bruchteil abschätzen und überdies berechnen, wie genau unsere Schätzung voraussichtlich ist.

Die **Objektivisten** sagen, dass die Wahrscheinlichkeiten reale Aspekte des Universums sind – Tendenzen von Objekten, sich auf bestimmte Weise zu verhalten – und nicht nur Beschreibungen des Glaubensgrades eines Beobachters. Zum Beispiel ist die Wahrscheinlichkeit, dass eine geworfene Münze mit der Wahrscheinlichkeit 0,5 auf Kopf zu liegen kommt, eine Tendenz der eigentlichen Münze. Aus dieser Perspektive sind die Maße des Frequentisten Versuche, diese Tendenzen zu beobachten. Die meisten Physiker sind sich einig, dass Quantenphänomene objektiv probabilistisch sind, die Unsicherheit auf makroskopischer Ebene – z.B. beim Werfen einer Münze – stammt jedoch normalerweise aus dem Unwissen über die Ausgangsbedingungen und scheint nicht konsistent mit der Tendenzbetrachtung zu sein.

Nach Ansicht der **Subjektivisten** sind Wahrscheinlichkeiten ein Weg, die Glauben eines Agenten zu charakterisieren, ohne jedoch irgendwelche externe physische Bedeutung zu besitzen. Die subjektive **Bayessche** Ansicht erlaubt jedes in sich widerspruchsfreie Zurückführen von A-priori-Wahrscheinlichkeiten auf Aussagen, besteht dann aber auf geeigneter Bayesscher Aktualisierung, wenn Indizien eintreffen. Letztendlich beinhaltet selbst eine strenge frequentistische Position eine subjektive Analyse, was mit dem Problem der **Referenzklasse** zusammenhängt: Beim Versuch, die Ergebniswahrscheinlichkeit eines bestimmten Experimentes zu ermitteln, muss der Frequentist das Experiment in eine Referenzklasse von „ähnlichen" Experimenten mit bekannten Ergebnishäufigkeiten einordnen. I. J. Good (1983, Seite 27) schreibt: „Jedes Ereignis im Leben ist einzigartig und jede Wahrscheinlichkeit im realen Leben, die wir in der Praxis abschätzen, ist die eines Ereignisses, das vorher noch niemals aufgetreten ist." Zum Beispiel könnte ein Frequentist, der für einen bestimmten Patienten die Wahrscheinlichkeit für ein Loch abschätzen möchte, eine Referenzklasse von anderen Patienten mit relevanten Ähnlichkeiten – Alter, Symptome, Ernährung – betrachten und überprüfen, welcher Anteil von ihnen ein Loch hatte. Wenn der Zahnarzt jedoch berücksichtigt, was über den Patienten bekannt ist – Gewicht bis aufs Gramm genau, Haarfarbe, Mädchenname der Mutter –, würde die Referenzklasse am Ende leer sein. Dies ist ein leidiges Problem in der Wissenschaftsphilosophie gewesen.

Das **Indifferenzprinzip**, das auf Laplace (1816) zurückgeht, besagt, dass Aussagen, die im Hinblick auf die Indizien syntaktisch „symmetrisch" sind, eine gleiche Wahrscheinlichkeit zugeordnet werden soll. Verschiedene Verbesserungen wurden vorgeschlagen, die schließlich in dem Versuch von Carnap und anderen gipfelten, eine streng **induktive Logik** zu entwickeln, die in der Lage war, die korrekte Wahrscheinlichkeit für beliebige Aussagen aus einer beliebigen Menge von Beobachtungen zu berechnen. Derzeit glaubt man, dass es keine eindeutige induktive Logik gibt; stattdessen basiert jede dieser Logiken auf einer subjektiven unbedingten Wahrscheinlichkeitsverteilung, deren Wirkung sich verringert, wenn weitere Beobachtungen gesammelt werden.

Agent 1		Agent 2		Ergebnisse und Auszahlungen für Agent 1			
Aussage	Glaube	Wette	Einsätze	a, b	$a, \neg b$	$\neg a, b$	$\neg a, \neg b$
a	0,4	a	4 zu 6	-6	-6	4	4
b	0,3	b	3 zu 7	-7	3	-7	3
$a \vee b$	0,8	$\neg(a \vee b)$	2 zu 8	2	2	2	-8
				-11	-1	-1	-1

Abbildung 13.2: Weil Agent 1 inkonsistente Glauben hat, kann Agent 2 eine Menge von Wetten ableiten, die einen Verlust für Agent 1 garantieren, egal welches Ergebnis a und b haben.

Andere stark philosophische Argumente wurden für die Verwendung von Wahrscheinlichkeiten vorgebracht, insbesondere die von Cox (1946), Carnap (1950) und Jaynes (2003). Diese konstruieren jeweils eine Menge von Axiomen zum Schließen mit Glaubensgraden: keine Widersprüche, Übereinstimmung mit gewöhnlicher Logik (wenn zum Beispiel Glaube in A zunimmt, muss Glaube in $\neg A$ abnehmen) usw. Kontrovers ist lediglich das Axiom, dass Glaubensgrade Zahlen sein oder sich zumindest wie Zahlen verhalten müssen. Das heißt, sie müssen transitiv (wenn Glaube in A größer als Glaube in B ist, der größer als Glaube in C ist, dann muss Glaube in A größer als der in C sein) und vergleichbar sein (der Glaube in A muss gleich, größer als oder kleiner als der Glaube in B sein). Es lässt sich dann beweisen, dass Wahrscheinlichkeit der einzige Ansatz ist, der diese Axiome erfüllt.

Weil die Welt jedoch ist, wie sie ist, sind praktische Demonstrationen häufig wirkungsvoller als Beweise. Der Erfolg von Schlusssystemen, die auf der Wahrscheinlichkeitstheorie basieren, war effektiver, um die Fachwelt zu überzeugen. Wir werden jetzt betrachten, wie Axiome eingesetzt werden können, um Inferenzen zu erzielen.

13.3 Inferenz mithilfe vollständig gemeinsamer Verteilungen

In diesem Abschnitt beschreiben wir eine einfache Methode der **probabilistischen Inferenz** – d.h. die Berechnung von A-posteriori-Wahrscheinlichkeiten für Abfrageaussagen bei gegebenen Indizien. Wir verwenden die vollständig gemeinsame Verteilung als die „Wissensbasis“, aus der Antworten für alle Fragen abgeleitet werden können. Während der Erklärungen werden wir mehrere praktische Techniken für die Manipulation von Gleichungen mit Wahrscheinlichkeiten vorstellen.

Wir beginnen mit einem sehr einfachen Beispiel, einer Domäne, die nur aus drei booleschen Variablen besteht: *Zahnschmerzen*, *Loch* und *Verfangen* (der schreckliche Haken des Zahnarztes verfängt sich in meinem Zahn). Die vollständige gemeinsame Verteilung ist eine 2×2×2-Tabelle, wie in ▶ Abbildung 13.3 gezeigt.

	zahnschmerzen		¬zahnschmerzen	
	verfangen	¬verfangen	verfangen	¬verfangen
loch	0,108	0,012	0,072	0,008
¬loch	0,016	0,064	0,144	0,576

Abbildung 13.3: Eine vollständige gemeinsame Verteilung für die Welt aus *Zahnschmerzen*, *Loch* und *Verfangen*.

Beachten Sie, dass die Wahrscheinlichkeiten der gemeinsamen Verteilung die Summe 1 ergeben, wie von den Wahrscheinlichkeitsaxiomen gefordert. Beachten Sie auch, dass Gleichung (13.2) uns eine direkte Möglichkeit bietet, die Wahrscheinlichkeit einer Aussage zu berechnen, egal ob einfach oder komplex: Wir identifizieren einfach diejenigen möglichen Welten, in denen die Aussage wahr ist, und summieren ihre Wahrscheinlichkeiten. Beispielsweise gibt es sechs mögliche Welten, in denen $loch \vee zahnschmerzen$ gilt:

$$P(loch \vee zahnschmerzen) = 0{,}108 + 0{,}012 + 0{,}072 + 0{,}008 + 0{,}016 + 0{,}064 = 0{,}28.$$

Eine besonders häufige Aufgabe ist es, die Verteilung über eine Untermenge von Variablen oder über eine einzelne Variable zu extrahieren. Addiert man beispielsweise die Einträge in der ersten Zeile, erhält man die unbedingte oder **marginale Wahrscheinlichkeit**[4] von *loch*:

$$P(loch) = 0{,}108 + 0{,}012 + 0{,}072 + 0{,}008 = 0{,}2.$$

Dieser Prozess heißt **Marginalisierung** oder **Aussummierung** – weil die Wahrscheinlichkeiten für jeden möglichen Wert der anderen Variablen summiert und dabei aus der Gleichung herausgenommen werden. Wir können die folgende allgemeine Marginalisierungsregel für beliebige Mengen von Variablen $\mathbf{Y}$ und $\mathbf{Z}$ schreiben:

$$\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z} \in \mathbf{Z}} \mathbf{P}(\mathbf{Y}, \mathbf{z}). \tag{13.6}$$

Hierbei bedeutet $\Sigma z_{\in} Z$, dass alle möglichen Kombinationen von Werten der Variablenmenge $\mathbf{Z}$ zu summieren sind. Dies wird auch als Σz abgekürzt, wobei $\mathbf{Z}$ implizit bleibt. Wir haben die Regel eben wie folgt verwendet:

$$\mathbf{P}(Loch) = \sum_{\mathbf{z} \in \{Loch, Zahnschmerzen\}} \mathbf{P}(Loch, \mathbf{z}). \tag{13.7}$$

Eine Variante dieser Regel verwendet bedingte Wahrscheinlichkeiten statt gemeinsamer Wahrscheinlichkeiten unter Verwendung der Produktregel:

$$\mathbf{P}(\mathbf{Y}) = \sum_{z} \mathbf{P}(\mathbf{Y} \mid \mathbf{z}) P(\mathbf{z}). \tag{13.8}$$

Diese Regel wird auch als **Konditionierung** bezeichnet. Marginalisierung und Konditionierung erweisen sich als nützliche Regeln für alle Arten von Ableitungen, einschließlich probabilistischer Ausdrücke.

Größtenteils sind wir daran interessiert, *bedingte* Wahrscheinlichkeiten bestimmter Variablen bei gegebenen Indizien über andere zu berechnen. Bedingte Wahrscheinlichkeiten ermittelt man, indem man zuerst Gleichung (13.3) anwendet, um einen Ausdruck in Form unbedingter Wahrscheinlichkeiten zu erhalten, und den Ausdruck dann aus der vollständigen gemeinsamen Verteilung auswertet. Beispielsweise können wir die Wahrscheinlichkeit eines Lochs bei gegebenen Indizien für Zahnschmerzen wie folgt berechnen:

$$\begin{aligned} P(loch \mid zahnschmerzen) &= \frac{P(loch \wedge zahnschmerzen)}{P(zahnschmerzen)} \\ &= \frac{0{,}108 + 0{,}012}{0{,}108 + 0{,}012 + 0{,}016 + 0{,}064} = 0{,}6. \end{aligned}$$

4 Diese Bezeichnung stammt daher, dass es unter Versicherungsmathematikern eine übliche Vorgehensweise ist, die Summen beobachteter Häufigkeiten an den Rändern der Versicherungstabellen aufzuschreiben.

Zur Kontrolle können wir auch die Wahrscheinlichkeit berechnen, dass kein Loch vorhanden ist, wenn Zahnschmerzen vorliegen:

$$P(\neg loch \mid zahnschmerzen) = \frac{P(\neg loch \wedge zahnschmerzen)}{P(zahnschmerzen)}$$

$$= \frac{0{,}016 + 0{,}064}{0{,}108 + 0{,}012 + 0{,}016 + 0{,}064} = 0{,}4.$$

Die beiden Werte summieren sich zu 1,0, wie es sein sollte. Beachten Sie, dass in diesen beiden Berechnungen der Term $1/P(zahnschmerzen)$ konstant bleibt, unabhängig davon, welchen Wert von *Loch* wir berechnen. Tatsächlich kann man ihn als **Normalisierungskonstante** für die Verteilung $\mathbf{P}(Loch \mid zahnschmerzen)$ betrachten, was sicherstellt, dass die Summe 1 ergibt. In den Kapiteln zur Wahrscheinlichkeit verwenden wir α, um diese Konstanten zu kennzeichnen. Mit dieser Notation können wir die beiden vorigen Gleichungen in einer einzigen ausdrücken:

$$\begin{aligned}\mathbf{P}(Loch \mid zahnschmerzen) &= \alpha \mathbf{P}(Loch, zahnschmerzen) \\ &= \alpha[\mathbf{P}(Loch, zahnschmerzen, verfangen) + \mathbf{P}(Loch, zahnschmerzen, \neg verfangen)] \\ &= \alpha[\langle 0{,}108;\ 0{,}016\rangle + \langle 0{,}012;\ 0.064\rangle] = \alpha\langle 0{,}12;\ 0{,}08\rangle = \langle 0{,}6;\ 0{,}4\rangle.\end{aligned}$$

Mit anderen Worten können wir $\mathbf{P}(Loch \mid zahnschmerzen)$ berechnen, selbst wenn wir den Wert von $P(zahnschmerzen)$ nicht kennen! Wir vergessen vorübergehend den Faktor $1/P(zahnschmerzen)$ und summieren die Werte für *loch* und $\neg loch$, was 0,12 und 0,08 liefert. Dies sind die korrekten relativen Proportionen, doch ergeben sie nicht die Summe 1. Deshalb dividieren wir die einzelnen Werte durch (0,12 + 0,08), um sie zu normalisieren, und erhalten die wahren Wahrscheinlichkeiten von 0,6 und 0,4. Die Normalisierung erweist sich in vielen Wahrscheinlichkeitsberechnungen als praktische Abkürzung: Unter anderem erleichtert sich dadurch die Berechnung und man kann auch weiterrechnen, wenn eine bestimmte Wahrscheinlichkeitsbewertung (wie zum Beispiel $P(zahnschmerzen)$) nicht verfügbar ist.

Aus dem Beispiel können wir eine allgemeine Inferenzprozedur ableiten. Wir beginnen mit dem Fall, in dem die Abfrage eine einzige Variable X (im Beispiel *Loch*) enthält. Es seien $\mathbf{E}$ die Liste der Evidenzvariablen (im Beispiel lediglich *Zahnschmerzen*) mit $\mathbf{e}$, den dafür beobachteten Werten, und $\mathbf{Y}$ die restlichen unbeobachteten Variablen (im Beispiel lediglich *Verfangen*). Die Abfrage lautet $\mathbf{P}(X \mid \mathbf{e})$ und kann wie folgt ausgewertet werden:

$$\mathbf{P}(X \mid \mathbf{e}) = \alpha \mathbf{P}(X, \mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}(X, \mathbf{e}, \mathbf{y}). \tag{13.9}$$

Dabei läuft die Summe über alle möglichen $\mathbf{y}$ (d.h. alle möglichen Kombinationen aus Werten der unbeobachteten Variablen $\mathbf{Y}$). Beachten Sie, dass die Variablen X, $\mathbf{E}$ und $\mathbf{Y}$ zusammen die vollständige Variablenmenge für die Domäne bilden, deshalb ist $\mathbf{P}(X, \mathbf{e}, \mathbf{y})$ einfach eine Untermenge der Wahrscheinlichkeiten aus der vollständigen gemeinsamen Verteilung.

Für eine gegebene vollständige gemeinsame Verteilung kann Gleichung (13.9) probabilistische Abfragen für diskrete Variablen beantworten. Allerdings ist die Skalierbarkeit nicht gut: Eine Domäne, die durch n boolesche Variablen beschrieben wird, erfordert eine Eingabetabelle der Größe $O(2^n)$ und eine Zeit von $O(2^n)$, um die Tabelle zu verar-

beiten. In einem realistischen Problem haben wir leicht $n > 100$, was $O(2^n)$ unmöglich macht. Die vollständige gemeinsame Verteilung in Tabellenform ist kein praktisches Werkzeug für den Aufbau von Schlusssystemen. Stattdessen sollte man sie als theoretische Grundlage betrachten, auf der effektivere Ansätze aufgesetzt werden können, genau wie Wahrheitstabellen eine theoretische Basis für praktischere Algorithmen wie zum Beispiel DPLL bilden. Das restliche Kapitel führt einige der grundlegenden Ideen ein, die als Vorbereitung für die Entwicklung realistischer Systeme in *Kapitel 14* erforderlich sind.

13.4 Unabhängigkeit

Wir erweitern die vollständige gemeinsame Verteilung aus Abbildung 13.3, indem wir eine vierte Variable hinzufügen, *Wetter*. Die vollständige gemeinsame Verteilung wird dann zu $\mathbf{P}(Zahnschmerzen, Verfangen, Loch, Wetter)$, was $2 \times 2 \times 2 \times 4 = 32$ Einträge bedeutet. Sie enthält vier „Ausgaben" der in Abbildung 13.3 gezeigten Tabelle, eine für jede Art von Wetter. Welche Beziehungen bestehen zwischen diesen Ausgaben zueinander und zu der ursprünglichen Tabelle mit den drei Variablen? Wie sind beispielsweise $P(zahnschmerzen, verfangen, loch, wolkig)$ und $P(zahnschmerzen, verfangen, loch)$ miteinander verwandt? Wir können die Produktregel verwenden:

$$P(zahnschmerzen, verfangen, loch, wolkig) = P(wolkig \mid zahnschmerzen, verfangen, loch)\, P(zahnschmerzen, verfangen, loch)$$

Wenn man nicht gerade esoterisch abgehoben ist, sollte man nicht davon ausgehen, dass die eigenen Zahnprobleme Einfluss auf das Wetter haben. Und für die etablierte Zahnmedizin scheint es zumindest sicher zu sein, zu sagen, dass das Wetter die Dentalvariablen nicht beeinflusst. Demzufolge dürfte die folgende Behauptung vernünftig sein:

$$P(wolkig \mid zahnschmerzen, verfangen, loch) = P(wolkig). \tag{13.10}$$

Daraus können wir Folgendes ableiten:

$$P(zahnschmerzen, verfangen, loch, wolkig) = P(wolkig)\, P(zahnschmerzen, verfangen, loch).$$

Eine ähnliche Gleichung gibt es für *jeden Eintrag* in $\mathbf{P}(Zahnschmerzen, Verfangen, Loch, Wetter)$. Letztlich können wir die folgende allgemeine Gleichung schreiben:

$$\mathbf{P}(Zahnschmerzen, Verfangen, Loch, Wetter) = \mathbf{P}(Zahnschmerzen, Verfangen, Loch)\, \mathbf{P}(Wetter)$$

Damit kann die 32 Elemente umfassende Tabelle für vier Variablen aus einer achtelementigen Tabelle und einer vierelementigen Tabelle erzeugt werden. Diese Zerlegung ist schematisch in ▶ Abbildung 13.4(a) dargestellt.

Die Eigenschaft, die wir in Gleichung (13.10) verwendet haben, wird als **Unabhängigkeit** (auch **marginale Unabhängigkeit** und **absolute Unabhängigkeit**) bezeichnet. Insbesondere ist das Wetter unabhängig von den eigenen Zahnproblemen. Die Unabhängigkeit zwischen den Aussagen a und b kann wie folgt geschrieben werden:

$$P(a \mid b) = P(a) \quad \text{oder} \quad P(b \mid a) = P(b) \quad \text{oder} \quad P(a \wedge b) = P(a)P(b). \tag{13.11}$$

Alle diese Formen sind äquivalent (Übung 13.12). Die Unabhängigkeit zwischen Variablen X und Y kann wie folgt geschrieben werden (auch diese Formen sind äquivalent):

$$\mathbf{P}(X \mid Y) = \mathbf{P}(X) \quad \text{oder} \quad \mathbf{P}(Y \mid X) = \mathbf{P}(Y) \quad \text{oder} \quad \mathbf{P}(X, Y) = \mathbf{P}(X)\,\mathbf{P}(Y).$$

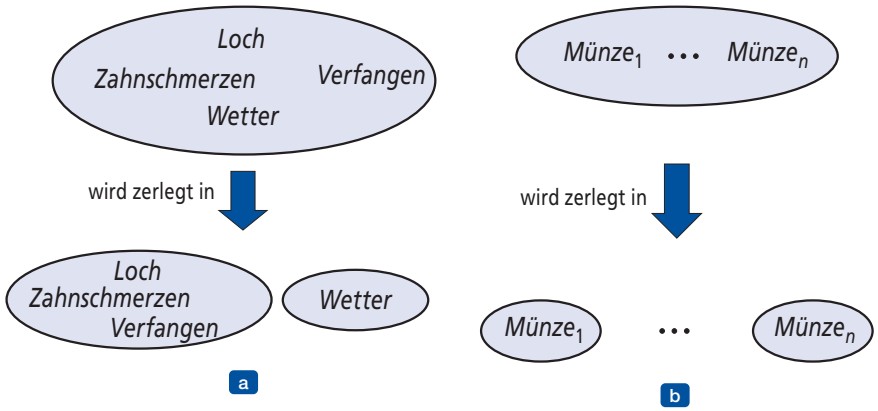


Abbildung 13.4: Zwei Beispiele für die Faktorisierung einer großen gemeinsamen Verteilung in kleinere Verteilungen, wozu die absolute Unabhängigkeit verwendet wird. (a) Wetter und Zahnprobleme sind voneinander unabhängig. (b) Münzwürfe sind voneinander unabhängig.

Zusicherungen der Unabhängigkeit basieren normalerweise auf dem Wissen über die Domäne. Wie das Beispiel mit Zahnschmerzen und Wetter gezeigt hat, können sie erheblich die Menge der Informationen verringern, die erforderlich sind, um die vollständige gemeinsame Verteilung zu spezifizieren. Wenn sich die vollständige Variablenmenge in unabhängige Untermengen unterteilen lässt, dann kann die vollständige Verbindung in separate gemeinsame Verteilungen über diese Untermengen *faktorisiert* werden. Beispielsweise hat die gemeinsame Verteilung $\mathbf{P}(C_1, ..., C_n)$ zum Ergebnis von n voneinander unabhängigen Münzwürfen 2^n Einträge, kann aber als Produkt von n Einzelvariablen-Verteilungen $\mathbf{P}(C_i)$ dargestellt werden. Praktischer ausgedrückt, ist die Unabhängigkeit von Zahnheilkunde und Meteorologie sinnvoll, weil sonst die Zahnärzte detailliertes Wissen über die Meteorologie und die Meteorologen ein detailliertes Wissen über die Zahnheilkunde besitzen müssten.

Wenn die Unabhängigkeitszusicherungen verfügbar sind, können sie dazu beitragen, die Größe der Domänenrepräsentation und die Komplexität des Inferenzproblems zu verringern. Leider ist eine klare Trennung ganzer Variablenmengen durch Unabhängigkeit eher selten. Wenn es eine Verbindung zwischen zwei Variablen gibt, gilt die Unabhängigkeit nicht, auch wenn diese Verbindung indirekt ist. Darüber hinaus können selbst unabhängige Untermengen relativ groß werden – beispielsweise gibt es in der Zahnheilkunde Dutzende von Krankheiten und Hunderte von Symptomen, die alle irgendwie miteinander zu tun haben. Um solche Probleme zu lösen, brauchen wir raffiniertere Methoden als das einfache Konzept der Unabhängigkeit.

13.5 Die Bayessche Regel und ihre Verwendung

In *Abschnitt 13.2.1* haben wir die **Produktregel** definiert. Diese Regel lässt sich in zwei Formen darstellen:

$$P(a \wedge b) = P(a \mid b)\, P(b) \quad \text{und} \quad P(a \wedge b) = P(b \mid a)\, P(a)$$

Setzen wir die beiden rechten Seiten gleich und dividieren durch $P(a)$, erhalten wir:

$$P(b \mid a) = \frac{P(a \mid b) P(b)}{P(a)}. \tag{13.12}$$

Diese Gleichung wird als **Bayessche Regel** (auch Bayessches Gesetz oder Bayessches Theorem) bezeichnet. Diese einfache Gleichung liegt allen modernen KI-Systemen für probabilistische Inferenz zugrunde. Der allgemeinere Fall mehrwertiger Variablen kann in **P**-Notation wie folgt geschrieben werden:

$$\mathbf{P}(Y \mid X) = \frac{\mathbf{P}(X \mid Y)\mathbf{P}(Y)}{\mathbf{P}(X)}.$$

Auch dies wird als Repräsentation einer Menge von Gleichungen gesehen, die jeweils spezifische Werte der Variablen verarbeiten. Außerdem haben wir die Gelegenheit, eine allgemeinere Version zu verwenden, die auf eine Hintergrundevidenz **e** konditioniert ist:

$$\mathbf{P}(Y \mid X, \mathbf{e}) = \frac{\mathbf{P}(X \mid Y, \mathbf{e})\mathbf{P}(Y \mid \mathbf{e})}{\mathbf{P}(X \mid \mathbf{e})}. \tag{13.13}$$

13.5.1 Die Anwendung der Bayesschen Regel: der einfache Fall

Auf den ersten Blick scheint die Bayessche Regel nicht sehr nützlich zu sein. Sie ermöglicht es uns, den einzelnen Term $P(b \mid a)$ in Form von drei Termen – $P(a \mid b)$, $P(b)$ und $P(a)$ – zu berechnen. Das sieht zwar aus wie zwei Schritte rückwärts, doch ist die Bayessche Regel in der Praxis nützlich, weil es viele Fälle gibt, in denen wir gute Wahrscheinlichkeitsabschätzungen für diese drei Zahlen besitzen und die vierte berechnen müssen. Oftmals erhalten wir als Indiz den *Effekt* einer unbekannten *Ursache* und möchten diese Ursache bestimmen. In diesem Fall wird die Bayessche Regel zu:

$$P(ursache \mid effekt) = \frac{P(effekt \mid ursache) P(ursache)}{P(effekt)}.$$

Die bedingte Wahrscheinlichkeit $P(effekt \mid ursache)$ quantifiziert die Beziehung in der **kausalen** Richtung, während $P(ursache \mid effekt)$ die **diagnostische** Richtung beschreibt. In einer Aufgabenstellung wie beispielsweise der medizinischen Diagnose haben wir häufig bedingte Wahrscheinlichkeiten für kausale Beziehungen – d.h., der Arzt kennt $P(symptome \mid krankheit)$ – und wollen eine Diagnose – $P(krankheit \mid symptome)$ – ableiten. Zum Beispiel weiß ein Arzt, dass die Krankheit Meningitis in etwa 70% aller Fälle bewirkt, dass der Patient einen steifen Nacken hat. Der Arzt kennt außerdem einige unbedingte Fakten: Die A-priori-Wahrscheinlichkeit, dass ein Patient Meningitis hat, beträgt 1/50.000, und die A-priori-Wahrscheinlichkeit, dass ein Patient einen steifen Nacken hat, liegt bei 1%. Sei s die Aussage, dass der Patient einen steifen Nacken hat, und m die Aussage, dass der Patient Meningitis hat, dann haben wir:

$$P(s \mid m) = 0{,}7$$
$$P(m) = 1/50000$$
$$P(s) = 0{,}01$$
$$P(m \mid s) = \frac{P(s \mid m)P(m)}{P(s)} = \frac{0{,}7 \times 1/50000}{0{,}01} = 0{,}0014. \quad (13.14)$$

Wir erwarten also für weniger als 1 unter 700 Patienten mit steifem Nacken, dass er Meningitis hat. Zwar verursacht Meningitis häufig einen steifen Nacken (mit einer Wahrscheinlichkeit von 0,7), doch bleibt die Wahrscheinlichkeit, dass ein Patient Meningitis hat, recht klein. Das liegt daran, dass die A-priori-Wahrscheinlichkeit eines steifen Nackens sehr viel höher ist als die für Meningitis.

Abschnitt 13.3 hat einen Prozess beschrieben, wie man vermeiden kann, die A-priori-Wahrscheinlichkeit der Indizien (hier $P(s)$) abzuschätzen, indem man stattdessen eine A-posteriori-Wahrscheinlichkeit für jeden Wert der Abfragevariablen berechnet (hier m und $\neg m$) und dann die Ergebnisse normalisiert. Derselbe Prozess kann bei Verwendung der Bayesschen Regel angewendet werden. Wir haben:

$$\mathbf{P}(M \mid s) = \alpha \langle P(s \mid m)P(m), P(s \mid \neg m)P(\neg m) \rangle.$$

Um diesen Ansatz verwenden zu können, müssen wir also $P(s \mid \neg m)$ statt $P(s)$ schätzen. Nichts ist umsonst – manchmal ist es einfacher, manchmal ist es schwieriger. Die allgemeine Form der Bayesschen Regel mit Normalisierung lautet:

$$\mathbf{P}(Y \mid X) = \alpha \mathbf{P}(X \mid Y)\mathbf{P}(Y). \quad (13.15)$$

Dabei ist α die Normalisierungskonstante, die benötigt wird, damit die Einträge in $\mathbf{P}(Y \mid X)$ die Summe 1 ergeben.

Tipp

Eine offensichtliche Frage zur Bayesschen Regel ist, warum man die bedingte Wahrscheinlichkeit in der einen Richtung zur Verfügung hat, nicht aber in der anderen. In der Meningitis-Domäne weiß der Arzt vielleicht, dass ein steifer Nacken in einem von 5000 Fällen durch eine Meningitis verursacht ist; das bedeutet, der Arzt hat eine quantitative Information in der **Diagnose**richtung von den Symptomen zu den Ursachen. Ein solcher Arzt braucht die Bayessche Regel nicht. *Leider ist das diagnostische Wissen häufig sehr viel subtiler als das kausale Wissen.* Wenn es eine plötzliche Meningitisepidemie gibt, geht die unbedingte Wahrscheinlichkeit der Meningitis $P(m)$ nach oben. Der Arzt, der die diagnostische Wahrscheinlichkeit $P(m \mid s)$ direkt aus der statistischen Beobachtung von Patienten vor der Epidemie abgeleitet hat, kann nicht wissen, wie der Wert zu aktualisieren ist, aber der Arzt, der $P(m \mid s)$ aus den anderen drei Werten berechnet, erkennt, dass $P(m \mid s)$ proportional mit $P(m)$ steigen soll. Noch wichtiger ist, dass die kausale Information $P(s \mid m)$ von der Epidemie *unbeeinflusst* bleibt, weil sie einfach nur die Art und Weise reflektiert, wie sich Meningitis auswirkt. Die Verwendung dieser Art direkt kausalen oder modellbasierten Wissens unterstützt die wichtige Robustheit, die man braucht, um probabilistische Systeme für die reale Welt praktikabel einsetzen zu können.

13.5.2 Verwendung der Bayesschen Regel: Evidenzen kombinieren

Wir haben gesehen, dass die Bayessche Regel praktisch für die Beantwortung probabilistischer Abfragen sein kann, die auf eine einzige Evidenz konditioniert sind – beispielsweise den steifen Nacken. Insbesondere haben wir gesagt, dass probabilistische Information häufig in der Form $P(\textit{effekt} \mid \textit{ursache})$ zur Verfügung steht. Was passiert, wenn

wir zwei oder mehr Evidenzen haben? Was kann ein Zahnarzt beispielsweise schließen, wenn sein schrecklicher Haken sich in dem schmerzenden Zahn eines Patienten verfängt? Wenn wir die vollständige gemeinsame Verteilung (Abbildung 13.3) kennen, können wir die Antwort ablesen:

$$\mathbf{P}(Loch|zahnschmerzen \wedge verfangen) = \alpha\langle 0{,}018;\, 0{,}016\rangle \approx \langle 0{,}871;\, 0{,}129\rangle.$$

Wir wissen jedoch, dass dieser Ansatz nicht für eine große Anzahl an Variablen geeignet ist.

Wir können versuchen, die Bayessche Regel anzuwenden, um das Problem umzuformulieren:

$$\mathbf{P}(Loch|zahnschmerzen \wedge verfangen) = \alpha\, \mathbf{P}(zahnschmerzen \wedge verfangen \mid Loch)\, \mathbf{P}(Loch). \tag{13.16}$$

Damit diese Umformulierung funktioniert, müssen wir die bedingten Wahrscheinlichkeiten der Konjunktion *zahnschmerzen* ∧ *verfangen* für jeden Wert von *Loch* kennen. Das könnte für nur zwei Evidenzvariablen machbar sein, lässt sich aber wieder nicht für viele Variablen skalieren. Wenn es n mögliche Evidenzvariablen gibt (Röntgenbild, Ernährung, Mundhygiene usw.), gibt es 2^n mögliche Kombinationen beobachteter Werte, für die wir die bedingten Wahrscheinlichkeiten kennen müssten. Wir könnten genauso gut wieder auf die vollständige gemeinsame Verteilung zurückgreifen. Genau das hat die Forscher zunächst von der Wahrscheinlichkeitstheorie weg und hin zu den Näherungsverfahren für die Evidenzkombination geführt, die zwar zuweilen falsche Antworten geben, aber sehr viel weniger Zahlen benötigen, um überhaupt eine Antwort zu erzielen.

Anstatt diesen Weg einzuschlagen, brauchen wir einige zusätzliche Zusicherungen über die Domäne, die es uns erlauben, die Ausdrücke zu vereinfachen. Das Konzept der **Unabhängigkeit** in *Abschnitt 13.4* liefert einen Anhaltspunkt, muss aber noch verfeinert werden. Es wäre praktisch, wenn *Zahnschmerzen* und *Verfangen* voneinander unabhängig wären, aber das sind sie nicht. Wenn der Haken sich in dem Zahn verfängt, hat dieser wahrscheinlich ein Loch und das verursacht wahrscheinlich die Zahnschmerzen. Diese Variablen *sind* jedoch unabhängig, *wenn man weiß, ob ein Loch vorhanden ist oder nicht.* Jede wird direkt von dem Loch verursacht, aber keine hat einen direkten Effekt auf die jeweils andere: Zahnschmerzen sind von dem Zustand der Nerven im Zahn abhängig, während die Genauigkeit des Bohrers von den Fertigkeiten des Zahnarztes abhängig ist, für den die Zahnschmerzen irrelevant sind.[5] Mathematisch wird diese Eigenschaft wie folgt geschrieben:

$$\mathbf{P}(zahnschmerzen \wedge verfangen| Loch) = \mathbf{P}(zahnschmerzen| Loch)\, \mathbf{P}(verfangen| Loch). \tag{13.17}$$

Diese Gleichung drückt die **bedingte Unabhängigkeit** von *zahnschmerzen* und *verfangen* bei gegebenem *Loch* aus. Wir können sie in Gleichung (13.16) einsetzen, um die Wahrscheinlichkeit eines Loches zu erhalten:

$$\mathbf{P}(Loch \mid zahnschmerzen \wedge verfangen) = \alpha\mathbf{P}(zahnschmerzen \mid Loch)\, \mathbf{P}(verfangen \mid Loch)\mathbf{P}(Loch). \tag{13.18}$$

Jetzt sind die Informationsanforderungen dieselben wie für die Inferenz, wobei jeder Bestandteil der Evidenz separat verwendet wird: die unbedingte Wahrscheinlichkeit

5 Wir gehen davon aus, dass Patient und Zahnarzt zwei unterschiedliche Personen sind.

$\mathbf{P}(Loch)$ für die Abfragevariable und die bedingte Wahrscheinlichkeit für jeden Effekt bei gegebener Ursache.

Die allgemeine Definition der **bedingten Unabhängigkeit** von zwei Variablen X und Y bei einer gegebenen dritten Variablen Z lautet:

$$\mathbf{P}(X, Y \mid Z) = \mathbf{P}(X \mid Z)\mathbf{P}(Y \mid Z).$$

In der Zahnarztdomäne beispielsweise scheint es sinnvoll zu sein, bedingte Unabhängigkeit von den Variablen *Zahnschmerzen* und *Verfangen* für ein gegebenes *Loch* zuzusichern:

$$\begin{aligned}&\mathbf{P}(Zahnschmerzen, Verfangen \mid Loch)\\ &= \mathbf{P}(Zahnschmerzen \mid Loch)\mathbf{P}(Verfangen \mid Loch).\end{aligned} \tag{13.19}$$

Beachten Sie, dass diese Zusicherung etwas stärker als Gleichung (13.17) ist, die die Unabhängigkeit nur für bestimmte Werte von *Zahnschmerzen* und *Verfangen* zusichert. Wie bei der absoluten Unabhängigkeit in Gleichung (13.11) können auch die äquivalenten Formen

$$\mathbf{P}(X \mid Y, Z) = \mathbf{P}(X \mid Z) \quad \text{und} \quad \mathbf{P}(Y \mid X, Z) = \mathbf{P}(Y \mid Z)$$

verwendet werden (siehe Übung 13.17).

Abschnitt 13.4 hat gezeigt, dass die Zusicherungen der absoluten Unabhängigkeit eine Zerlegung der vollständigen gemeinsamen Verteilung in sehr viel kleinere Stücke erlauben. Es zeigt sich, dass dasselbe auch für Zusicherungen der bedingten Unabhängigkeit gilt. Beispielsweise können wir mit der Zusicherung in Gleichung (13.19) wie folgt eine Zerlegung ableiten:

$$\begin{aligned}&\mathbf{P}(Zahnschmerzen, Verfangen, Loch)\\ &= \mathbf{P}(Zahnschmerzen, Verfangen \mid Loch)\mathbf{P}(Loch) && \text{(Produktregel)}\\ &= \mathbf{P}(Zahnschmerzen \mid Loch)\mathbf{P}(Verfangen \mid Loch)\mathbf{P}(Loch) && \text{[unter Verwendung von (13.14)]}\end{aligned}$$

(Der Leser kann anhand von Abbildung 13.3 leicht überprüfen, dass diese Gleichung tatsächlich gilt.)

Tipp

Auf diese Weise wird die ursprünglich größere Tabelle in drei kleinere Tabellen zerlegt. Die ursprüngliche Tabelle hat sieben unabhängige Zahlen ($2^3 = 8$ Einträge in der Tabelle, doch da sie die Summe 1 ergeben müssen, sind 7 unabhängig). Die kleineren Tabellen enthalten fünf unabhängige Zahlen (für eine bedingte Wahrscheinlichkeitsverteilung wie zum Beispiel $\mathbf{P}(T \mid C)$ gibt es zwei Zeilen von zwei Zahlen und jede Zeile muss die Summe 1 ergeben, sodass es zwei unabhängige Zahlen sind; für eine A-priori-Verteilung wie $\mathbf{P}(C)$ gibt es nur eine unabhängige Zahl). Der Übergang von sieben auf fünf scheint kein größerer Gewinn zu sein, aber der Punkt ist, dass für n Symptome, die alle bedingt unabhängig für ein gegebenes *Loch* sind, die Größe der Repräsentation mit $O(n)$ statt mit $O(2^n)$ wächst. Damit *können es bedingte Unabhängigkeitszusicherungen ermöglichen, probabilistische Systeme zu skalieren; darüber hinaus stehen sie sehr viel häufiger zur Verfügung als absolute Unabhängigkeitszusicherungen*. Konzeptuell werden *Zahnschmerzen* und *Verfangen* durch *Loch* **getrennt**, weil es eine direkte Ursache für beides ist. Die Zerlegung großer probabilistischer Domänen in schwach verbundene Untermengen mithilfe der bedingten Unabhängigkeit ist eine der wichtigsten Entwicklungen in der neueren Geschichte der KI.

Das Zahnarztbeispiel zeigt ein häufig auftretendes Muster, in dem eine einzige Ursache direkt mehrere Effekte beeinflusst, die alle voneinander im Sinne von Bedingungen

unabhängig sind, wenn die Ursache bekannt ist. Die vollständige gemeinsame Verteilung kann geschrieben werden als:

$$\mathbf{P}(Ursache, Effekt_1, ..., Effekt_n) = \mathbf{P}(Ursache)\prod_i \mathbf{P}(Effekt_i \mid Ursache).$$

Eine solche Wahrscheinlichkeitsverteilung wird auch als **naives Bayes-Modell** bezeichnet – „naiv", weil es häufig (als vereinfachende Annahme) in Fällen verwendet wird, wo die „Effekt"-Variablen eigentlich *nicht* bedingt unabhängig sind, wenn die Ursachenvariable gegeben ist. (Das naive Bayes-Modell wird manchmal auch als **Bayes-Klassifizierer** bezeichnet, eine etwas sorglose Verwendung, die die echten Bayesianer dazu verleitet hat, vom **Idioten-Bayes-Modell** zu sprechen.) In der Praxis funktionieren naive Bayes-Systeme überraschend gut, selbst wenn die Unabhängigkeitsannahme nicht wahr ist. *Kapitel 20* beschreibt Methoden zum Lernen naiver Bayes-Verteilungen aus Beobachtungen.

13.6 Eine erneute Betrachtung der Wumpus-Welt

Wir können viele der Ideen aus diesem Kapitel kombinieren, um probabilistische Inferenzprobleme in der Wumpus-Welt zu lösen. (Eine vollständige Beschreibung der Wumpus-Welt finden Sie in *Kapitel 7*.) Unsicherheit entsteht in der Wumpus-Welt, weil die Sensoren des Agenten nur partielle, lokale Informationen über die Welt bereitstellen. Zum Beispiel zeigt ▶ Abbildung 13.5 eine Situation, in der jedes der drei erreichbaren Quadrate – [1,3], [2,2] und [3,1] – eine Falltür enthalten könnte. Die reine logische Inferenz kann nichts darüber schließen, welches Quadrat am wahrscheinlichsten sicher ist; deshalb könnte ein logischer Agent gezwungen sein, zufällig ein Quadrat auszuwählen. Wir werden sehen, dass ein aussagenlogischer Agent sehr viel besser ist als der logische Agent.

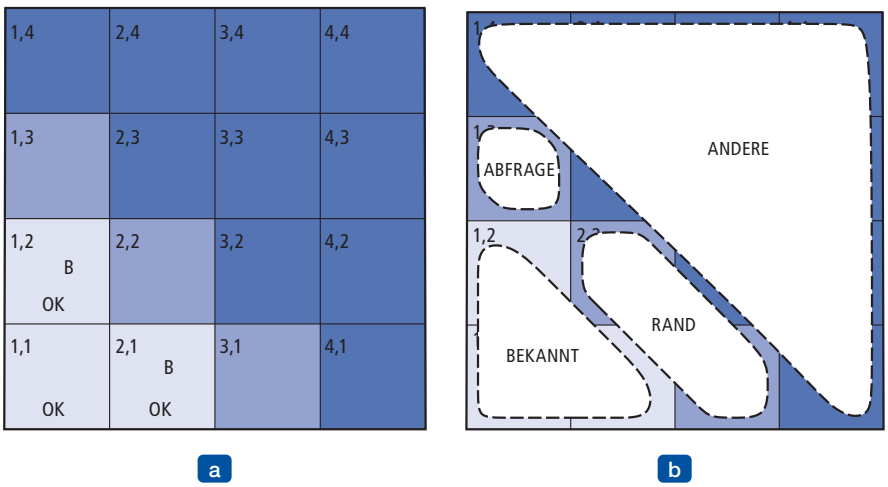


Abbildung 13.5: (a) Nachdem der Agent sowohl in [1,2] als auch in [2,1] einen Luftzug verspürt hat, gelangt er nicht weiter – es gibt keine weiteren sicheren Quadrate zu erkunden. (b) Unterteilung der Quadrate in *Bekannt*, *Rand* und *Andere* für die Abfrage zu [1,3].

Unser Ziel ist es, die Wahrscheinlichkeit zu berechnen, dass die drei Quadrate eine Falltür enthalten. (In diesem Beispiel werden wir den Wumpus und das Gold ignorieren.) Die relevanten Eigenschaften der Wumpus-Welt sind, dass (1) eine Falltür bewirkt, dass es in allen benachbarten Quadraten windig ist, und (2) jedes Quadrat außer [1,1] mit einer Wahrscheinlichkeit von 0,2 eine Falltür enthält. Der erste Schritt ist, die Menge der benötigten Zufallsvariablen zu identifizieren:

- Wie im Fall der Aussagenlogik wollen wir eine boolesche Variable P_{ij} für jedes Quadrat, die dann wahr ist, wenn das Quadrat $[i, j]$ tatsächlich eine Falltür enthält.
- Wir haben außerdem boolesche Variablen B_{ij}, die wahr sind, wenn das Quadrat $[i, j]$ windig ist; wir berücksichtigen diese Variablen nur für die beobachteten Quadrate – in diesem Fall [1,1], [1,2] und [2,1].

Im nächsten Schritt spezifizieren wir die vollständige gemeinsame Verteilung, $\mathbf{P}(P_{1,1},..., P_{4,4}, B_{1,1}, B_{1,2}, B_{2,1})$. Durch Anwendung der Produktregel erhalten wir:

$$\mathbf{P}(P_{1,1},..., P_{4,4}, B_{1,1}, B_{1,2}, B_{2,1}) = \mathbf{P}(B_{1,1}, B_{1,2}, B_{2,1} \mid P_{1,1},..., P_{4,4})\, \mathbf{P}(P_{1,1},..., P_{4,4}).$$

Anhand dieser Zerlegung ist ganz einfach zu erkennen, wie die gemeinsamen Wahrscheinlichkeitswerte aussehen sollten. Der erste Term ist die bedingte Wahrscheinlichkeit einer Luftzugkonfiguration für eine bekannte Falltürkonfiguration, das heißt 1, wenn die Luftzüge neben den Falltüren liegen, andernfalls 0. Der zweite Term ist die A-priori-Wahrscheinlichkeit einer Falltürkonfiguration. Jedes Quadrat enthält mit einer Wahrscheinlichkeit von 0,2 eine Falltür, unabhängig von den anderen Quadraten. Damit gilt:

$$\mathbf{P}(P_{1,1},..., P_{4,4} \prod_{i,j=1,1}^{4,4} \mathbf{P}(P_{i,j}). \tag{13.20}$$

Für eine bestimmte Konfiguration mit genau n Falltüren ist $P(P_{1,1},...,P_{4,4}) = 0{,}2^n \times 0{,}8^{16-n}$.

In der in Abbildung 13.5(a) gezeigten Situation besteht die Evidenz aus dem beobachteten Luftzug (oder seinem Fehlen) in jedem besuchten Quadrat, kombiniert mit der Tatsache, dass jedes dieser Quadrate keine Falltür enthält. Wir kürzen diese Fakten ab als $b = \neg b_{1,1} \wedge b_{1,2} \wedge b_{2,1}$ und $bekannt = \neg p_{1,1} \wedge \neg p_{1,2} \wedge \neg p_{2,1}$. Wir wollen Abfragen wie etwa $\mathbf{P}(P_{1,3} | bekannt, b)$ beantworten: Wie wahrscheinlich ist es nach den bisherigen Beobachtungen, dass [1,3] eine Falltür enthält?

Um diese Abfrage zu beantworten, können wir dem in Gleichung (13.9) vorgeschlagenen Ansatz folgen, nämlich die Summe über Einträgen aus der vollständigen gemeinsamen Verteilung bilden. Es sei *Unbekannt* die Menge der $P_{i,j}$ Variablen für andere Quadrate als die *Bekannt*-Quadrate und das Abfragequadrat [1,3]. Gemäß Gleichung (13.9) haben wir dann:

$$\mathbf{P}(P_{1,3} \mid bekannt, b) = \alpha \sum_{unbekannt} \mathbf{P}(P_{1,3}, unbekannt, bekannt, b).$$

Die vollständigen gemeinsamen Wahrscheinlichkeiten wurden bereits spezifiziert; wir sind also fertig – d.h., wenn wir die Berechnung ignorieren. Es gibt zwölf unbekannte

Quadrate; die Summierung enthält also $2^{12} = 4096$ Terme. Im Allgemeinen wächst die Summierung exponentiell mit der Anzahl der Quadrate.

Natürlich könnte man fragen: Sind die anderen Quadrate nicht irrelevant? Wie könnte [4,4] beeinflussen, ob es in [1,3] eine Falltür gibt? Tatsächlich ist diese Intuition korrekt! Es seien *Rand* die Variablen (wobei es sich nicht um die Abfragevariablen handelt), die neben besuchten Quadraten liegen, in diesem Fall also lediglich [2,2] und [3,1]. Außerdem seien *Andere* die Variablen für die anderen unbekannten Quadrate; in diesem Fall gibt es zehn andere Quadrate, wie in Abbildung 13.5(b) gezeigt. Der Knackpunkt ist, dass die beobachteten Luftzüge *bedingt unabhängig* von den anderen Variablen sind, wenn die bekannten Rand- und Abfragevariablen gegeben sind. Um diese Erkenntnis zu nutzen, bringen wir die Abfrageformel in eine Form, in der die Luftzüge von allen anderen Variablen abhängig sind, und wenden dann bedingte Unabhängigkeit an:

$$\begin{aligned}
&\mathbf{P}(P_{1,3} \mid \mathit{bekannt}, b) \\
&= \alpha \sum_{\mathit{unbekannt}} \mathbf{P}(P_{1,3}, \mathit{bekannt}, b, \mathit{unbekannt}) && \text{(nach Gleichung (13.9))} \\
&= \alpha \sum_{\mathit{unbekannt}} \mathbf{P}(b \mid P_{1,3}, \mathit{bekannt}, \mathit{unbekannt}) \mathbf{P}(P_{1,3}, \mathit{bekannt}, \mathit{unbekannt}) \\
&\qquad \text{(durch die Produktregel)} \\
&= \alpha \sum_{\mathit{rand}} \sum_{\mathit{andere}} \mathbf{P}(b \mid \mathit{bekannt}, P_{1,3}, \mathit{rand}, \mathit{andere}) \mathbf{P}(P_{1,3}, \mathit{bekannt}, \mathit{rand}, \mathit{andere}) \\
&= \alpha \sum_{\mathit{rand}} \sum_{\mathit{andere}} \mathbf{P}(b \mid \mathit{bekannt}, P_{1,3}, \mathit{rand}) \mathbf{P}(P_{1,3}, \mathit{bekannt}, \mathit{rand}, \mathit{andere}),
\end{aligned}$$

wobei der letzte Schritt die bedingte Unabhängigkeit verwendet. b ist unabhängig von *andere* bei gegebenen *bekannt*, $P_{1,3}$ und *rand*. Jetzt hängt der erste Term in diesem Ausdruck nicht von anderen Variablen ab; wir können also die Summe nach innen verschieben:

$$\begin{aligned}
&\mathbf{P}(P_{1,3} \mid \mathit{bekannt}, b) \\
&= \alpha \sum_{\mathit{rand}} \mathbf{P}(b \mid \mathit{bekannt}, P_{1,3}, \mathit{rand}) \sum_{\mathit{andere}} \mathbf{P}(P_{1,3}, \mathit{bekannt}, \mathit{rand}, \mathit{andere}).
\end{aligned}$$

Durch die Unabhängigkeit ist es möglich, wie in Gleichung (13.20) den A-priori-Term zu faktorisieren und dann die Terme neu anzuordnen:

$$\begin{aligned}
&\mathbf{P}(P_{1,3} \mid \mathit{bekannt}, b) \\
&= \alpha \sum_{\mathit{rand}} \mathbf{P}(b \mid \mathit{bekannt}, P_{1,3}, \mathit{rand}) \sum_{\mathit{andere}} \mathbf{P}(P_{1,3}) P(\mathit{bekannt}) P(\mathit{rand}) P(\mathit{andere}) \\
&= \alpha P(\mathit{bekannt}) \mathbf{P}(P_{1,3}) \sum_{\mathit{rand}} \mathbf{P}(b \mid \mathit{bekannt}, P_{1,3}, \mathit{rand}) P(\mathit{rand}) \sum_{\mathit{andere}} P(\mathit{andere}) \\
&= \alpha' \mathbf{P}(P_{1,3}) \sum_{\mathit{rand}} \mathbf{P}(b \mid \mathit{unbekannt}, P_{1,3}, \mathit{rand}) P(\mathit{rand}),
\end{aligned}$$

wobei der letzte Schritt den Term $P(bekannt)$ in die Normalisierungskonstante aufnimmt und die Tatsache nutzt, dass $\Sigma_{andere} P(andere)$ gleich 1 ist.

Jetzt gibt es lediglich vier Terme in der Summation über die Randvariablen $P_{2,2}$ und $P_{3,1}$. Unabhängigkeit und bedingte Unabhängigkeit haben die anderen Quadrate vollkommen aus der Betrachtung eliminiert.

Beachten Sie, dass der Ausdruck $\mathbf{P}(b|bekannt, P_{1,3}, rand)$ gleich 1 ist, wenn der Rand konsistent mit den Luftzugbeobachtungen ist, andernfalls 0. Somit summieren wir also für jeden Wert von $P_{1,3}$ über die *logischen Modelle* für die Randvariablen, die konsistent mit den bekannten Fakten sind. (Vergleichen Sie dies mit der Auflistung über Modelle in Abbildung 7.5.) Die Modelle und die ihnen zugeordneten A-priori-Wahrscheinlichkeiten – $P(rand)$ – sind in ▶ Abbildung 13.6 gezeigt. Wir haben:

$$\mathbf{P}(P_{1,3}|bekannt, b) = \alpha' \langle 0{,}2(0{,}04 + 0{,}16 + 0{,}16); 0{,}8(0{,}04; 0{,}16) \rangle \approx \langle 0{,}31; 0{,}69 \rangle.$$

Das bedeutet, [1,3] (und wegen der Symmetrie auch [3,1]) enthält mit einer Wahrscheinlichkeit von etwa 31% eine Falltür. Eine ähnliche Berechnung, die wir dem Leser überlassen, zeigt, dass [2,2] mit einer Wahrscheinlichkeit von etwa 86% eine Falltür enthält. Der Wumpus sollte also [2,2] definitiv vermeiden! Unser logischer Agent aus *Kapitel 7* hat nicht gewusst, dass [2,2] schlechter als die anderen Quadrate ist. Die Logik kann uns sagen, dass es nicht bekannt ist, ob es in [2,2] eine Falltür gibt, doch wir brauchen die Wahrscheinlichkeit, wie sicher diese Aussage ist.

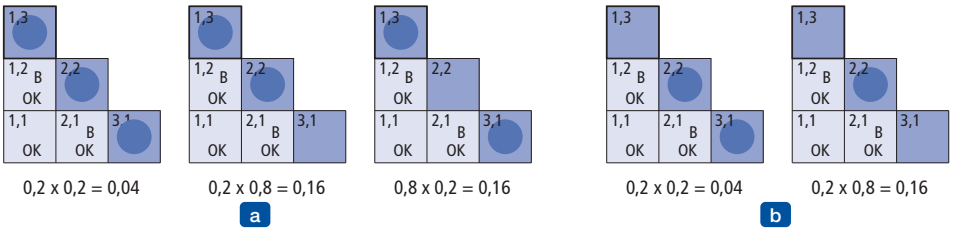


Abbildung 13.6: Konsistente Modelle für die Randvariablen $P_{2,2}$ und $P_{3,1}$, die für jedes Modell $P(rand)$ zeigen: (a) Drei Modelle mit $P_{1,3} = true$ zeigen zwei oder drei Falltüren, (b) zwei Modelle mit $P_{1,3} = false$ zeigen eine oder zwei Falltüren.

Dieser Abschnitt hat gezeigt, dass selbst scheinbar komplizierte Probleme in der Wahrscheinlichkeitstheorie präzise formuliert und unter Verwendung eines einfachen Algorithmus gelöst werden können. Um *effiziente* Lösungen zu erhalten, lassen sich die Summationen mithilfe von Unabhängigkeit und bedingter Unabhängigkeit vereinfachen. Diese Beziehungen entsprechen häufig unserem natürlichen Verständnis, wie ein Problem zerlegt werden sollte. Im nächsten Kapitel entwickeln wir formale Repräsentationen für solche Beziehungen sowie die Algorithmen, die für diese Repräsentationen eingesetzt werden, um effizient eine probabilistische Inferenz durchzuführen.

Bibliografische und historische Hinweise

Die Wahrscheinlichkeitstheorie wurde als Mittel erfunden, Glücksspiele zu analysieren. Um 850 beschrieb der indische Mathematiker Mahaviracarya, wie eine Menge von Wetten zu platzieren ist, die nicht verlieren kann (was wir heute als „Dutch Book“ bezeichnen). In Europa hat Girolamo Cardano etwa um 1565 die erste wichtige systematische Analyse geschaffen, die allerdings erst posthum (1663) veröffentlicht wurde. Damals hatte die Entdeckung einer systematischen Methode zur Berechnung von Wahrscheinlichkeiten durch Blaise Pascal und Pierre Fermat (1654) dazu geführt, dass die Wahrscheinlichkeit zu einer mathematischen Disziplin wurde. Wie bei der Wahrscheinlichkeit selbst wurden die Ergebnisse anfangs durch Glücksspielprobleme motiviert (siehe Übung 13.9). Das erste veröffentlichte Lehrbuch über die Wahrscheinlichkeit war *De Ratiociniis in Ludo Aleae* (Huygens, 1657). Die auf „Faulheit und Unwissen“ basierende Ansicht der Unsicherheit wurde von John Arbuthnot im Vorwort seiner Übersetzung von Huygens (Arbuthnot, 1692) beschrieben: „It is impossible for a Die, with such determin’d force and direction, not to fall on such determin’d side, only I don’t know the force and direction which makes it fall on such determin’d side, and therefore I call it Chance, which is nothing but the want of art...“[6]

Laplace (1816) gab einen außerordentlich genauen und modernen Überblick über die Wahrscheinlichkeit; er war der erste, der das Beispiel „nimm zwei Urnen A und B, wobei die erste vier weiße und zwei schwarze Kugeln enthält ...“ verwendete. Der Pfarrer Thomas Bayes (1702–1761) stellte die Regel für das Schließen über bedingte Wahrscheinlichkeiten vor, die nach ihm benannt ist (Bayes, 1763). Bayes betrachtete nur den Fall gleichförmiger A-priori-Wahrscheinlichkeiten; erst Laplace entwickelte den unabhängigen allgemeinen Fall. Kolmogorov (1950, zuerst in Deutschland 1933 veröffentlicht) war der Erste, der die Wahrscheinlichkeitstheorie in einem streng axiomatischen Gerüst darstellte. Rényi (1970) zeigte später eine axiomatische Repräsentation, die die bedingte Wahrscheinlichkeit statt der absoluten Wahrscheinlichkeit als elementaren Baustein benutzte.

Pascal verwendete die Wahrscheinlichkeit in einer Weise, die sowohl die objektive Interpretation, und zwar als eine Eigenschaft der Welt basierend auf symmetrischer oder relativer Häufigkeit, als auch die subjektive Interpretation benötigte, die auf dem Glaubensgrad basierte – Erstere in seiner Analyse der Wahrscheinlichkeiten in Glücksspielen, Letztere im berühmten Argument der „Pascalschen Wette“, das sich mit der möglichen Existenz von Gott beschäftigt. Allerdings erkannte Pascal den Unterschied zwischen diesen beiden Interpretationen nicht. Die erste klare Unterscheidung wurde von James Bernoulli (1654–1705) aufgezeigt.

Leibniz führte das „klassische“ Konzept der Wahrscheinlichkeit als Verhältnis nummerierter, gleich wahrscheinlicher Fälle auf, das auch von Bernoulli verwendet wurde; bekannt wurde es jedoch erst durch Laplace (1749–1827). Dieses Konzept ist mehrdeutig – zwischen Häufigkeitsinterpretation und subjektiver Interpretation. Die Fälle können als gleich wahrscheinlich angesehen werden, entweder aufgrund einer natürlichen, physischen Symmetrie zwischen ihnen oder einfach, weil wir über kein

6 („Es ist für einen Würfel nicht möglich, mit einer derartigen bestimmten Kraft und Richtung nicht auf eine solcherart bestimmte Seite zu fallen, nur dass ich die Kraft und die Richtung nicht kenne, die ihn dazu bringt, auf eine solche bestimmte Seite zu fallen, und ich demzufolge es Glück nenne, was nichts weiter als der Wunsch der Kunst ist ...“)

Wissen verfügen, das uns zu der Auffassung gelangen ließe, das eine sei wahrscheinlicher als das andere. Die Verwendung dieser zweiten, subjektiven Betrachtung, um die Zuweisung gleicher Wahrscheinlichkeiten zu rechtfertigen, wird auch als **Indifferenzprinzip** bezeichnet. Das Prinzip wird oftmals Laplace zugeschrieben, der es jedoch nie explizit isoliert darstellte. George Boole und John Venn nennen es das **Prinzip des unzureichenden Grundes**; der moderne Name geht auf Keynes (1921) zurück.

Im 20. Jahrhundert verschärfte sich die Debatte zwischen Objektivisten und Subjektivisten. Kolmogorov (1963), R.A. Fisher (1922) und Richard von Mises (1928) waren Fürsprecher der relativen Häufigkeitsinterpretation. Karl Poppers (1959, zuerst in Deutschland 1934 veröffentlicht) „Neigungs"-Interpretation führt relative Häufigkeiten auf eine zugrunde liegende physische Symmetrie zurück. Frank Ramsey (1931), Bruno de Finetti (1937), R.T. Cox (1946), Leonard Savage (1954), Richard Jeffrey (1983) und E. T. Jaynes (2003) interpretierten Wahrscheinlichkeiten als Glaubensgrade bestimmter Individuen. Ihre Analysen des Glaubensgrades waren eng an Nutzen und Verhalten gebunden – insbesondere die Bereitschaft zu wetten. Rudolf Carnap zeigte gefolgt von Leibniz und Laplace eine andere Art der subjektiven Interpretation der Wahrscheinlichkeit auf – nicht als Glaubensgrad eines tatsächlich existierenden Einzelnen, sondern als Glaubensgrad, den ein idealisierter Einzelner in eine bestimmte Aussage a haben *sollte,* wenn eine bestimmte Evidenzmenge $\mathbf{e}$ vorliegt. Carnap wollte noch einen Schritt weiter als Leibniz oder Laplace gehen und formulierte sein Konzept des **Bestätigungsgrades** mathematisch präzise als logische Relation zwischen a und $\mathbf{e}$. Die Studie dieser Relation sollte eine mathematische Disziplin bilden, die sogenannte **induktive Logik**, analog zur normalen deduktiven Logik (Carnap, 1948, 1950). Carnap konnte seine induktive Logik kaum über den aussagenlogischen Fall hinaus erweitern und Putnam (1963) zeigte durch nachteilige Argumente, dass bestimmte grundsätzliche Schwierigkeiten eine strenge Erweiterung auf Sprachen, die Arithmetik ausdrücken können, verhindern würden.

Das Theorem von Cox (1946) zeigt, dass jedes System für unsicheres Schließen, das seine Menge von Annahmen erfüllt, der Wahrscheinlichkeitstheorie äquivalent ist. Das gab denjenigen, die bereits die Wahrscheinlichkeit favorisierten, neue Zuversicht, doch waren andere mit Hinweis auf die Annahmen nicht überzeugt (hauptsächlich weil Glaube durch eine einzelne Zahl darzustellen sei und somit der Glaube in $\neg p$ eine Funktion des Glaubens in p sein muss). Halpern (1999) beschreibt die Annahmen und zeigt einige Lücken in der ursprünglichen Formulierung von Cox auf. Horn (2003) zeigt, wie sich die Schwierigkeiten beheben lassen. Jaynes (2003) führt ein ähnliches Argument an, das verständlicher ist.

Die Frage der Referenzklassen ist eng mit dem Versuch verwandt, eine induktive Logik zu finden. Der Ansatz, die „spezifischste" Referenzklasse ausreichender Größe zu wählen, wurde von Reichenbach (1949) formal vorgeschlagen. Es wurden mehrere Versuche unternommen, insbesondere von Henry Kyburg (1977, 1983), komplexere Strategien zu formulieren, um einige offensichtliche Trugschlüsse zu vermeiden, die durch Reichenbachs Regel entstanden, aber diese Ansätze sind gewissermaßen über einen *Ad-hoc*-Zustand nicht hinausgekommen. Neuere Arbeiten von Bacchus, Grove, Halpern und Koller (1992) erweitern die Methoden von Carnap auf Theorien erster Stufe und vermeiden dabei viele der Schwierigkeiten, die der einfachen Referenzklassenmethode innewohnen. Kyburg und Teng (2006) stellen die probabilistische Inferenz der nichtmonotonen Logik gegenüber.

Das Bayessche probabilistische Schließen wird in der KI seit den 1960er Jahren verwendet, insbesondere in der Medizindiagnose. Es wurde nicht nur verwendet, um Diagnosen aus verfügbaren Evidenzen zu erstellen, sondern auch, um weitere Fragen und Tests auszuwählen, wofür die Theorie des Informationswertes herangezogen wurde (*Abschnitt 16.6*), wenn die verfügbaren Evidenzen nicht schlüssig waren (Gorry, 1968; Gorry et al., 1973). Ein System übertraf die menschlichen Experten bei der Diagnose akuter Unterleibsschmerzen (de Dombal et al., 1974). Lucas et al. (2004) geben einen Überblick. Diese frühen Bayesschen Systeme litten jedoch unter zahlreichen Problemen. Weil sie kein theoretisches Modell der Bedingungen hatten, die sie diagnostizierten, waren sie anfällig für unrepräsentative Daten, die in Situationen entstanden, für die nur sehr wenige Stichproben zur Verfügung standen (de Dombal et al., 1981). Es gab jedoch noch eine andere grundlegende Schwierigkeit: Weil es keinen präzisen Formalismus (wie den in *Kapitel 14* beschriebenen) gab, um bedingte Unabhängigkeitsinformationen darzustellen und zu nutzen, waren sie von der Erfassung, der Speicherung und der Verarbeitung riesiger Tabellen probabilistischer Daten abhängig. Aufgrund dieser Schwierigkeiten verlor man in der KI von den 1970er Jahren bis Mitte der 1980er Jahre das Interesse an probabilistischen Methoden für den Umgang mit der Unsicherheit. Im nächsten Kapitel werden Entwicklungen seit Ende der 1980er Jahre beschrieben.

Das naive Bayes-Modell für gemeinsame Verteilungen wurde seit den 1950er Jahren in der Literatur zur Mustererkennung umfassend beschrieben (Duda und Hart, 1973). Sie wurde beginnend mit der Arbeit von Maron (1961) auch (häufig unwissentlich) beim Informationsabruf eingesetzt. Die probabilistischen Grundlagen dieser Technik, die in Übung 13.22 weiter beschrieben sind, wurden von Robertson und Sparck Johnes (1976) genauer betrachtet. Domingos und Pazzani (1997) zeigen eine Erklärung für den überraschenden Erfolg des naiven Bayesschen Schließens selbst in Domänen, wo die Unabhängigkeitsannahmen offensichtlich verletzt wurden.

Es gibt viele gute Einführungsbücher zur Wahrscheinlichkeitstheorie, unter anderem von Bertsekas und Tsitsiklis (2008) sowie Grinstead und Snell (1997). DeGroot und Schervish (2001) bieten eine kombinierte Einführung in die Wahrscheinlichkeit und Statistik aus Bayesscher Perspektive. Das Lehrbuch von Richard Hamming (1991) gibt eine mathematisch anspruchsvolle Einführung in die Wahrscheinlichkeitstheorie vom Standpunkt einer Neigungsinterpretation basierend auf physischer Symmetrie. Hacking (1975) und Hald (1990) behandeln die frühe Geschichte des Wahrscheinlichkeitskonzeptes. Bernstein (1996) bietet einen unterhaltsamen populärwissenschaftlich formulierten Überblick über die Geschichte des Risikos.

Zusammenfassung

Dieses Kapitel hat die Wahrscheinlichkeitstheorie als geeignete Grundlage für unsicheres Schließen vorgeschlagen und eine „sanfte" Einführung in ihre Verwendung gegeben.

- Unsicherheit entsteht durch Faulheit und Unwissen. In komplexen, nichtdeterministischen oder partiell beobachtbaren Umgebungen ist sie unvermeidbar.
- Wahrscheinlichkeiten drücken die Unfähigkeit des Agenten aus, eine definitive Entscheidung im Hinblick auf die Wahrheit eines Satzes zu erreichen. Wahrscheinlichkeiten fassen die Glauben des Agenten in Bezug auf die Evidenz zusammen.
- Die Entscheidungstheorie kombiniert Glauben und Wünsche des Agenten, wobei die beste Aktion als diejenige definiert wird, die den erwarteten Nutzen maximiert.
- Zu den grundlegenden Wahrscheinlichkeitsaussagen gehören **A-priori-Wahrscheinlichkeiten** und **bedingte Wahrscheinlichkeiten** über einfachen und komplexen Aussagen.
- Die Axiome der Wahrscheinlichkeit beschränken die möglichen Zuweisungen von Wahrscheinlichkeiten zu Aussagen. Ein Agent, der die Axiome verletzt, muss sich in manchen Fällen irrational verhalten.
- Die **vollständige gemeinsame Verteilung** spezifiziert die Wahrscheinlichkeit jeder vollständigen Zuweisung von Werten zu Zufallsvariablen. Normalerweise ist sie zu groß, um sie in ihrer expliziten Form erstellen oder verwenden zu können. Steht sie aber zur Verfügung, lassen sich damit Abfragen beantworten, indem einfach die Einträge für die möglichen Welten, die den Abfrageaussagen entsprechen, addiert werden.
- **Absolute Unabhängigkeit** zwischen Untermengen von Zufallsvariablen ermöglicht es, die vollständige gemeinsame Verteilung in kleinere gemeinsame Verteilungen zu faktorisieren, wodurch sich die Komplexität drastisch verringert.
- Mithilfe der **Bayesschen Regel** lassen sich unbekannte Wahrscheinlichkeiten aus bekannten bedingten Wahrscheinlichkeiten berechnen – normalerweise in kausaler Richtung. Die Anwendung der Bayesschen Regel mit vielen Evidenzbestandteilen führt im Allgemeinen zu den gleichen Skalierungsproblemen wie die vollständige gemeinsame Verteilung.
- Durch **bedingte Unabhängigkeit**, die durch direkte kausale Beziehungen in der Domäne entsteht, ließe sich die vollständige gemeinsame Verteilung in kleinere, bedingte Verteilungen faktorisieren. Das **naive Bayes-Modell** geht von der bedingten Unabhängigkeit aller Effektvariablen aus, wenn eine einzelne Ursachenvariable bekannt ist, und wächst linear mit der Anzahl der Effekte.
- Ein Agent der Wumpus-Welt kann Wahrscheinlichkeiten für nicht beobachtete Aspekte der Welt berechnen und dabei die Entscheidungen eines rein logischen Agenten überbieten. Bedingte Unabhängigkeit macht diese Berechnungen behandelbar.

Übungen zu Kapitel 13

Lösungshinweise

1 Zeigen Sie aus den ersten Prinzipien, dass $P(a \mid b \wedge a) = 1$ gilt.

2 Beweisen Sie unter Verwendung der Wahrscheinlichkeitsaxiome, dass jede Wahrscheinlichkeitsverteilung für eine diskrete Zufallsvariable die Summe 1 ergeben muss.

3 Beweisen Sie für jede der folgenden Anweisungen, ob sie wahr ist, oder geben Sie ein Gegenbeispiel an:

a. Wenn $P(a \mid b, c) = P(b \mid a, c)$, dann $P(a \mid c) = P(b \mid c)$

b. Wenn $P(a \mid b, c) = P(a)$, dann $P(b \mid c) = P(b)$

c. Wenn $P(a \mid b) = P(a)$, dann $P(a \mid b, c) = P(a \mid c)$

4 Wäre es rational für einen Agenten, die drei Glauben $P(A) = 0{,}4$, $P(B) = 0{,}3$ und $P(A \vee B) = 0{,}5$ aufrechtzuerhalten? Welche Wahrscheinlichkeitsbereiche wären in diesem Fall rational für den Agenten, damit er $A \wedge B$ glaubt? Legen Sie eine Tabelle wie in Abbildung 13.2 an und zeigen Sie, wie sie Ihre Begründungen zur Rationalität unterstützt. Legen Sie eine weitere Version der Tabelle mit $P(A \vee B) = 0{,}7$ an. Erklären Sie, warum es rational ist, diese Wahrscheinlichkeit zu haben, selbst wenn die Tabelle einen Fall als Verlust und drei mit Ausgleich anzeigt. [*Hinweis*: Woran ist Agent 1 im Hinblick auf die Wahrscheinlichkeit der vier Fälle gebunden, insbesondere für den Fall, der einen Verlust anzeigt?]

5 Diese Frage beschäftigt sich mit den Eigenschaften möglicher Welten, die in *Abschnitt 13.2.2* als Zuweisungen zu allen Zufallsvariablen definiert sind. Wir arbeiten mit Aussagen, die exakt einer möglichen Welt entsprechen, weil sie die Zuweisungen aller Variablen genau definiert. In der Wahrscheinlichkeitstheorie heißen derartige Aussagen **atomare Ereignisse**. Sind zum Beispiel die booleschen Variablen X_1, X_2, X_3 gegeben, fixiert die Aussage $x_1 \wedge \neg x_2 \wedge \neg x_3$ die Zuweisung der Variablen; in der Sprache der Aussagenlogik würden wir sagen, dass sie genau ein Modell besitzt.

a. Beweisen Sie für den Fall von n booleschen Variablen, dass zwei beliebige voneinander verschiedene atomare Ereignisse sich wechselseitig ausschließen; d.h., ihre Konjunktion ist äquivalent mit $false$.

b. Beweisen Sie, dass die Disjunktion aller möglichen atomaren Ereignisse logisch äquivalent mit $true$ ist.

c. Beweisen Sie, dass jede Aussage logisch äquivalent mit der Disjunktion der atomaren Ereignisse ist, aus denen ihre Wahrheit logisch folgt.

6 Beweisen Sie Gleichung (13.4) aus den Gleichungen (13.1) und (13.2).

7 Betrachten Sie die Menge aller möglichen Pokerhände zu je fünf Karten, die gerecht von einem Standardspiel mit 52 Karten gegeben werden.

a. Wie viele atomare Ereignisse gibt es in der gemeinsamen Wahrscheinlichkeitsverteilung (d.h., wie viele verschiedene Hände zu je 5 Karten gibt es?)

b. Welche Wahrscheinlichkeit haben die einzelnen atomaren Ereignisse?

c. Wie groß ist die Wahrscheinlichkeit, dass ein Royal Straight Flush gegeben wird? Wie groß ist sie für einen Vierling (vier gleiche Karten)?

8 Gehen Sie von der in Abbildung 13.3 gezeigten vollständigen gemeinsamen Verteilung aus und berechnen Sie Folgendes:

a. $\mathbf{P}(Zahnschmerzen)$
b. $\mathbf{P}(Loch)$
c. $\mathbf{P}(Loch \mid Verfangen)$
d. $\mathbf{P}(Loch \mid Zahnschmerzen \vee Verfangen)$.

9 In seinem Brief vom 24. August 1654 wollte Pascal wissen, wie die Spieleinsätze zu verteilen sind, wenn ein Glücksspiel vorzeitig beendet werden muss. Stellen Sie sich ein Spiel vor, bei dem nacheinander gewürfelt wird. Spieler E erhält einen Punkt, wenn der Würfel eine gerade Augenzahl zeigt, und Spieler O erhält bei einer ungeraden Augenzahl einen Punkt. Der erste Spieler, der 7 Punkte erreicht, gewinnt den Pot. Angenommen, das Spiel wird unterbrochen, wenn E mit einem Stand von 4 zu 2 führt. Wie sollte der Spieleinsatz in diesem Fall gerecht aufgeteilt werden? Wie sieht die allgemeine Formel aus? (Fermat und Pascal machten mehrere Fehler, bevor sie das Problem gelöst hatten, doch sollten Sie in der Lage sein, die Lösung gleich beim ersten Mal richtig anzugeben.)

10 Als wir uns entschieden haben, unser Wissen über Wahrscheinlichkeiten für einen guten Zweck einzusetzen, treffen wir auf einen Spielautomaten (eine sogenannte Slot Machine) mit drei unabhängigen Walzen, die jeweils eines von vier Symbolen – BARREN, GLOCKE, ZITRONE oder KIRSCHE – mit gleicher Wahrscheinlichkeit zeigen. Die Slot Machine hat das folgende Auszahlungsschema für eine Wette von 1 Münze (wobei „?" bedeutet, dass uns das Ergebnis für die jeweilige Walze nicht interessiert):

BARREN/BARREN/BARREN zahlt 21 Münzen
GLOCKE/GLOCKE/GLOCKE zahlt 16 Münzen
ZITRONE/ZITRONE/ZITRONE zahlt 5 Münzen
KIRSCHE/KIRSCHE/KIRSCHE zahlt 3 Münzen
KIRSCHE/KIRSCHE/? zahlt 2 Münzen
KIRSCHE/?/? zahlt 1 Münze

a. Berechnen Sie die prozentuale „Rückerstattung" der Maschine. Mit anderen Worten: Wie groß ist die erwartete Auszahlung für jede eingesetzte Münze?
b. Berechnen Sie die Wahrscheinlichkeit, dass sich beim Spielen am Einarmigen Banditen einmal ein Gewinn einstellt.
c. Schätzen Sie Mittelwert und Median für die Anzahl der Spiele, die Sie spielen können, bevor Sie pleite gehen, wenn Sie mit acht Münzen beginnen. Diese Schätzung können Sie auch mit einer Simulation ermitteln, anstatt zu versuchen, eine genaue Antwort zu berechnen.

11 Wir möchten eine n-Bit-Nachricht an einen Empfängeragenten übertragen. Die Bits in der Nachricht werden während der Übertragung unabhängig voneinander mit einer Wahrscheinlichkeit von jeweils ε verfälscht (in den entgegengesetzten Zustand gekippt). Mit einem zusätzlichen Paritätsbit, das zusammen mit der ursprünglichen Information gesendet wird, lässt sich eine Nachricht durch den Empfänger korrigieren, wenn höchstens ein Bit in der gesamten Nachricht (einschließlich des Paritätsbits) verfälscht wurde. Angenommen, wir möchten sicherstellen, dass die korrekte Nachricht mit einer Wahrscheinlichkeit von wenigstens $1 - \delta$ empfangen wird. Wie groß ist der maximal realisierbare Wert von n? Berechnen Sie diesen Wert für den Fall $\varepsilon = 0{,}002$ und $\delta = 0{,}01$.

12 Zeigen Sie, dass die drei Formen der Unabhängigkeit in Gleichung (13.11) äquivalent sind.

13 Betrachten Sie zwei medizinische Tests A und B, die einen Virus finden sollen. Test A kann einen Virus zu 95% erkennen, falls er vorhanden ist, liefert aber zu 10% falsche Positivaussagen (zeigt also an, dass der Virus vorhanden ist, obwohl er es nicht ist). Test B ist zu 90% effektiv beim Erkennen des Virus, hat aber eine falsche Positivaussage von 5%. Die beiden Tests verwenden unabhängige Methoden, um den Virus zu identifizieren. Träger des Virus sind 1% aller Menschen. Nehmen wir an, dass eine Person nur mit einem der Tests auf den Virus hin untersucht wird und dieser Test ein positives Ergebnis liefert. Welcher Test mit einem positiven Ergebnis zeigt zuverlässiger jemanden an, der den Virus tatsächlich in sich trägt? Begründen Sie Ihre Antwort mathematisch.

14 Angenommen, Sie erhalten eine Münze, die mit der Wahrscheinlichkeit x auf dem Wappen und mit der Wahrscheinlichkeit $1 - x$ auf der Zahl landet. Sind die Ergebnisse von aufeinanderfolgenden Münzwürfen unabhängig voneinander, wenn man den Wert von x kennt? Sind die Ergebnisse von aufeinanderfolgenden Münzwürfen unabhängig voneinander, wenn man den Wert von x nicht kennt? Begründen Sie Ihre Antwort?

15 Nach Ihrem jährlichen Checkup hat der Arzt gute und schlechte Nachrichten. Die schlechten Nachrichten sind, dass Ihr Test auf eine ernsthafte Krankheit positiv war und dass ein solcher Test zu 99% zutrifft (d.h. die Wahrscheinlichkeit, dass der Test positiv ist, wenn Sie die Krankheit haben, ist 0,99, ebenso wie die Wahrscheinlichkeit, dass der Test negativ ist, wenn Sie die Krankheit nicht haben). Die gute Nachricht ist, dass dies eine seltene Krankheit ist, die nur etwa einen von 100.000 Menschen Ihres Alters trifft. Warum ist es eine gute Nachricht, dass die Krankheit selten ist? Welche Wahrscheinlichkeit besteht, dass Sie die Krankheit tatsächlich haben?

16 Häufig ist es praktisch, den Effekt bestimmter spezifischer Aussagen im Kontext einer allgemeineren Hintergrundevidenz zu spezifizieren, der feststehend bleibt, statt mit vollständigem Fehlen von Informationen. Die folgenden Fragen fordern Sie auf, allgemeinere Versionen der Produktregel und der Bayesschen Regel im Hinblick auf eine Hintergrundevidenz $\mathbf{e}$ zu formulieren:

a. Beweisen Sie die konditionalisierte Version der allgemeinen Produktregel:

$$\mathbf{P}(X, Y \mid \mathbf{e}) = \mathbf{P}(X \mid Y, \mathbf{e})\mathbf{P}(Y \mid \mathbf{e})$$

b. Beweisen Sie die konditionalisierte Version der Bayesschen Regel in Gleichung (13.13).

17 Zeigen Sie, dass die Aussage der bedingten Unabhängigkeit

$$\mathbf{P}(X, Y \mid Z) = \mathbf{P}(X \mid Z)\mathbf{P}(Y \mid Z)$$

äquivalent mit jeder der folgenden Aussagen ist:

$$\mathbf{P}(X \mid Y, Z) = \mathbf{P}(X \mid Z) \text{ und } \mathbf{P}(Y \mid X, Z) = \mathbf{P}(Y \mid Z)$$

18 In dieser Übung werden Sie die Normalisierungsberechnung für das Meningitisbeispiel vervollständigen. Legen Sie zuerst einen geeigneten Wert für $P(s \mid \neg m)$ fest und verwenden Sie ihn, um unnormalisierte Werte für $P(m \mid s)$ und $P(\neg m \mid s)$ zu berechnen (d.h., ignorieren Sie in dem Ausdruck der Bayesschen Regel den Term $P(s)$). Jetzt normalisieren Sie diese Werte, dass sie in der Summe 1 ergeben.

19 Diese Übung untersucht, wie sich Beziehungen der bedingten Unabhängigkeit auf den Umfang der Informationen auswirken, die für probabilistische Berechnungen erforderlich sind.

a. Angenommen, wir wollen $P(h \mid e_1, e_2)$ berechnen und haben keine Information über bedingte Unabhängigkeit. Welche der folgenden Zahlenmengen sind ausreichend für die Berechnung?

(i) $\mathbf{P}(E_1, E_2), \mathbf{P}(H), \mathbf{P}(E_1 \mid H), \mathbf{P}(E_2 \mid H)$

(ii) $\mathbf{P}(E_1, E_2), \mathbf{P}(H), \mathbf{P}(E_1, E_2 \mid H)$

(iii) $\mathbf{P}(H), \mathbf{P}(E_1 \mid H), \mathbf{P}(E_2 \mid H)$

b. Angenommen, wir wissen, dass $\mathbf{P}(E_1 \mid H, E_2) = \mathbf{P}(E_1 \mid H)$ für alle Werte von H, E_1, E_2 gilt. Welche der drei Mengen ist jetzt ausreichend?

20 Es seien X, Y und Z boolesche Zufallsvariablen. Beschriften Sie die acht Einträge in der gemeinsamen Verteilung $\mathbf{P}(X, Y, Z)$ mit a bis h. Drücken Sie aus, dass X und Y bedingt unabhängig sind, wobei Z als Menge der Gleichungen gegeben ist, die a bis h in Verbindung bringen. Wie viele *nicht redundante* Gleichungen gibt es?

21 Schreiben Sie einen allgemeinen Algorithmus für die Beantwortung von Abfragen der Form $\mathbf{P}(Ursache \mid \mathbf{e})$ unter Verwendung einer naiven Bayes-Verteilung. Gehen Sie davon aus, dass die Evidenz $\mathbf{e}$ *einer beliebigen* Untermenge der Effektvariablen Werte zuweisen kann.

22 Bei der Textkategorisierung wird ein Dokument abhängig von seinem Inhalt einer von mehreren Kategorien zugeordnet. Für diese Aufgabe werden häufig naive Bayes-Modelle verwendet. In diesen Modellen ist die Abfragevariable die Dokumentkategorie und die „Effekt"-Variablen sind das Vorhandensein oder das Fehlen bestimmter Wörter in der Sprache; man nimmt an, dass Wörter unabhängig in Dokumenten vorkommen und die Häufigkeiten durch die Dokumentkategorie bestimmt werden.

a. Erklären Sie genau, wie sich ein solches Modell konstruieren lässt. Geben Sie als „Trainingsdaten" eine Menge von Dokumenten vor, die bereits Kategorien zugeordnet wurden.

b. Erklären Sie genau, wie ein neues Dokument kategorisiert wird.

c. Ist die Annahme der Unabhängigkeit vernünftig? Diskutieren Sie!

23 In unserer Analyse der Wumpus-Welt haben wir die Tatsache genutzt, dass jedes Quadrat mit einer Wahrscheinlichkeit von 0,2 eine Falltür enthält – unabhängig von dem Inhalt anderer Quadrate. Nehmen Sie stattdessen an, dass genau $N/5$ Falltüren zufällig über die N Quadrate außer auf [1,1] verteilt sind. Sind die Variablen $P_{i,j}$ und $P_{k,l}$ immer noch unabhängig? Wie sieht die gemeinsame Verteilung $\mathbf{P}(P_{1,1}, \ldots, P_{4,4})$ jetzt aus? Wiederholen Sie die Berechnung für die Wahrscheinlichkeit von Falltüren in [1,3] und [2,2].

24 Führen Sie die Wahrscheinlichkeitsberechnung für Falltüren in [1,3] und [2,2] erneut aus, wobei Sie annehmen, dass jedes Quadrat mit einer Wahrscheinlichkeit von 0,01 eine Falltür enthält, und zwar unabhängig von den anderen Quadraten. Was lässt sich über die relative Leistung eines logischen Agenten im Vergleich zu einem probabilistischen Agenten in diesem Fall sagen?

25 Implementieren Sie einen hybriden probabilistischen Agenten für die Wumpus-Welt, der auf dem hybriden Agenten in Abbildung 7.20 und der in diesem Kapitel skizzierten probabilistischen Inferenzprozedur basiert.

Probabilistisches Schließen

14

ÜBERBLICK

In diesem Kapitel erklären wir, wie man Netzmodelle erstellt, um den Gesetzen der Wahrscheinlichkeitstheorie entsprechend unter Unsicherheit zu schließen.

Kapitel 13 hat die grundlegenden Elemente der Wahrscheinlichkeitstheorie eingeführt und darauf hingewiesen, wie wichtig Unabhängigkeit und bedingte Unabhängigkeit sind, um die probabilistische Darstellung der Welt zu vereinfachen. Dieses Kapitel stellt eine systematische Methode vor, solche Beziehungen explizit in Form **Bayesscher Netze** zu repräsentieren. Wir definieren Syntax und Semantik dieser Netze und zeigen, wie sie genutzt werden können, um unsicheres Wissen auf natürliche und effiziente Weise festzuhalten. Anschließend zeigen wir, wie probabilistische Inferenz in vielen praktischen Situationen effizient eingesetzt werden kann, auch wenn sie im ungünstigsten Fall rechnerisch nicht handhabbar ist. Darüber hinaus beschreiben wir eine Vielzahl annähernder Inferenzalgorithmen, die sich häufig anwenden lassen, wenn eine genaue Inferenz nicht praktikabel ist. Wir untersuchen Möglichkeiten, wie die Wahrscheinlichkeitstheorie auf Welten mit Objekten und Relationen angewendet werden kann – d.h. auf Repräsentationen *erster Stufe* im Unterschied zu *aussagenlogischen* Repräsentationen. Schließlich zeigen wir alternative Ansätze zum unsicheren Schließen auf.

14.1 Wissensrepräsentation in einer unsicheren Domäne

In *Kapitel 13* haben wir gesehen, dass die vollständige gemeinsame Wahrscheinlichkeitsverteilung in der Lage ist, jede Frage über eine Domäne zu beantworten, allerdings mit der steigenden Anzahl an Variablen auch unüberschaubar groß werden kann. Darüber hinaus ist es eher unnatürlich und mühsam, die Wahrscheinlichkeiten für mögliche Welten einzeln nacheinander zu spezifizieren.

Außerdem haben wir gesehen, dass Unabhängigkeit und bedingte Unabhängigkeit zwischen Variablen die Anzahl der Wahrscheinlichkeiten, die für die Definition der vollständigen gemeinsamen Verteilung spezifiziert werden müssen, wesentlich reduzieren können. Dieser Abschnitt stellt eine Datenstruktur vor, ein sogenanntes **Bayessches Netz**,[1] das die Abhängigkeiten zwischen Variablen darstellt. Bayessche Netze erlauben eine präzise Spezifikation *beliebiger* vollständiger gemeinsamer Wahrscheinlichkeitsverteilungen.

Ein Bayessches Netz ist ein gerichteter Graph, in dem jeder Knoten mit quantitativen Wahrscheinlichkeitsinformationen versehen ist. Die vollständige Spezifikation sieht wie folgt aus:

1. Jeder Knoten entspricht einer Zufallsvariablen, die diskret oder stetig sein kann.
2. Eine Menge gerichteter Verknüpfungen oder Kanten verbindet jeweils zwei Knoten. Wenn es eine Kante von Knoten X nach Knoten Y gibt, sagt man, X ist ein *übergeordneter Knoten* oder *Elternknoten* von Y. Der Graph hat keine gerichteten Zyklen und ist damit ein gerichteter azyklischer Graph (Directed Acyclic Graph, DAG).
3. Jeder Knoten X_i hat eine bedingte Wahrscheinlichkeitsverteilung $\mathbf{P}(X_i \mid \mathit{Eltern}(X_i))$, die den Effekt der übergeordneten Knoten auf den Knoten quantifiziert.

1 Dies ist der gebräuchlichste Name, es gibt aber auch noch viele andere Namen, wie beispielsweise **Glaubensnetz**, **probabilistisches Netz**, **kausales Netz** oder **Wissensabbildung**. In der Statistik bezieht sich der Begriff **grafisches Modell** auf eine etwas breitere Klasse, die Bayessche Netze beinhaltet. Eine Erweiterung der Bayesschen Netze, die sogenannten **Entscheidungsnetze** oder **Einflussdiagramme**, werden in *Kapitel 16* beschrieben.

Die Topologie des Netzes – die Menge der Knoten und Kanten – spezifiziert die in der Domäne geltenden bedingten Unabhängigkeitsbeziehungen, wie wir gleich detailliert erklären werden. Eine Kante hat normalerweise die *intuitive* Bedeutung, dass *X direkten Einfluss* auf *Y* hat, woraus sich ergibt, dass Ursachen den Effekten übergeordnet sind. Ein Domänenexperte kann normalerweise leicht entscheiden, welche direkten Einflüsse in der Domäne existieren – das ist sogar viel einfacher, als die eigentlichen Wahrscheinlichkeiten zu spezifizieren. Nachdem die Topologie des Bayesschen Netzes festgelegt wurde, brauchen wir nur für jede Variable eine bedingte Wahrscheinlichkeitsverteilung zu spezifizieren, wobei ihre Eltern bekannt sind. Wir werden sehen, dass die Kombination aus der Topologie und der bedingten Verteilung ausreicht, um die vollständige gemeinsame Verteilung für alle Variablen (implizit) zu spezifizieren.

Sie erinnern sich an die in *Kapitel 13* beschriebene einfache Welt, die aus den Variablen *Zahnschmerzen*, *Loch*, *Verfangen* und *Wetter* bestand. Wir haben gesagt, dass *Wetter* unabhängig von den anderen Variablen ist, und darüber hinaus, dass *Zahnschmerzen* und *Verfangen* bedingt unabhängig sind, wenn *Loch* gegeben ist. Diese Beziehungen werden durch die in ▶ Abbildung 14.1 gezeigte Bayessche Netzstruktur dargestellt. Formal wird die bedingte Unabhängigkeit von *Zahnschmerzen* und *Verfangen* für *Loch* durch das *Fehlen* einer Verknüpfung zwischen *Zahnschmerzen* und *Verfangen* dargestellt. Intuitiv stellt das Netz die Tatsache dar, dass *Loch* eine direkte Ursache von *Zahnschmerzen* und *Verfangen* ist, während es zwischen *Zahnschmerzen* und *Verfangen* keine direkte kausale Beziehung gibt.

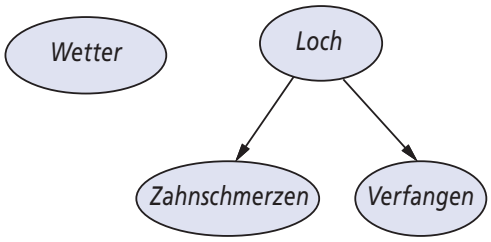


Abbildung 14.1: Ein einfaches Bayessches Netz, in dem *Wetter* von den drei anderen Variablen unabhängig ist und *Zahnschmerzen* und *Verfangen* für *Loch* bedingt unabhängig sind.

Betrachten Sie jetzt das folgende, etwas komplexere Beispiel. Sie haben bei sich zuhause eine neue Einbruchssicherung installiert. Sie erkennt Einbrüche recht zuverlässig, reagiert manchmal jedoch auch auf kleinere Erdbeben. (Dieses Beispiel stammt von Judea Pearl, der in Los Angeles wohnt – deshalb auch sein akutes Interesse an Erdbeben.) Sie haben zwei Nachbarn, John und Mary, die versprochen haben, Sie in der Arbeit anzurufen, wenn sie den Alarm hören. John ruft fast immer an, wenn er den Alarm hört, verwechselt manchmal aber auch das Telefonläuten mit dem Alarm und ruft auch dann an. Mary dagegen liebt laute Musik und überhört den Alarm manchmal. Mit der Evidenz, wer angerufen hat oder nicht, wollen wir die Wahrscheinlichkeit eines Einbruches abschätzen.

▶ Abbildung 14.2 gibt ein Bayessches Netz für diese Domäne an. Die Netzstruktur zeigt, dass Einbruch und Erdbeben die Wahrscheinlichkeit für eine Alarmauslösung direkt beeinflussen, während es nur vom Alarm abhängt, ob John und Mary anrufen. Das Netz stellt also unsere Annahme dar, dass sie Einbrüche nicht direkt wahrnehmen, kleinere Erdbeben nicht registrieren und sich nicht miteinander absprechen, bevor sie anrufen.

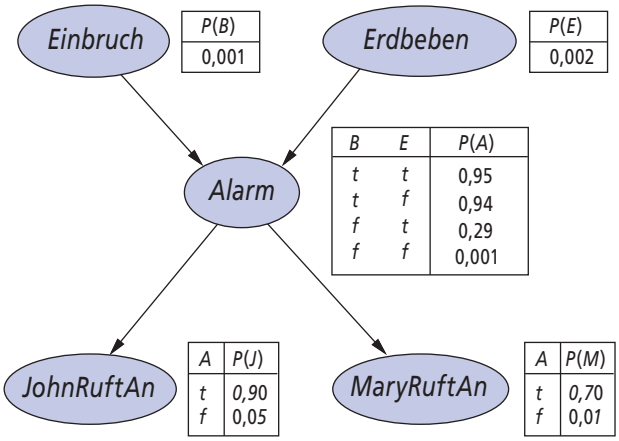


Abbildung 14.2: Ein typisches Bayessches Netz, das sowohl die Topologie als auch die bedingten Wahrscheinlichkeitstabellen (BWTs) zeigt. In den BWTs stehen die Buchstaben *B*, *E*, *A*, *J* und M für Einbruch (Burglary), Erdbeben, Alarm, JohnRuftAn und MaryRuftAn.

Die bedingten Verteilungen in Abbildung 14.2 sind als **bedingte Wahrscheinlichkeitstabelle**, BWT, dargestellt. (Diese Form der Tabelle kann für diskrete Variablen verwendet werden; andere Darstellungen, beispielsweise für stetige Variablen, sind in *Abschnitt 14.2* beschrieben.) Jede Zeile in einer BWT enthält die bedingte Wahrscheinlichkeit jedes Knotenwertes für einen **bedingenden Fall**. Ein bedingender Fall ist einfach nur eine mögliche Wertekombination für die Elternknoten – eine mögliche Welt en miniature, wenn Sie so wollen. Jede Zeile muss die Summe 1 ergeben, weil die Einträge eine erschöpfende Menge an Fällen für die Variable darstellen. Für boolesche Variablen muss die Wahrscheinlichkeit eines falschen Wertes gleich $1 - p$ sein, wenn wir die Wahrscheinlichkeit eines wahren Wertes als p kennen; deshalb lassen wir die zweite Zahl häufig weg, wie in Abbildung 14.2 gezeigt. Im Allgemeinen enthält eine Tabelle für eine boolesche Variable mit k booleschen Eltern 2^k unabhängig spezifizierbare Wahrscheinlichkeiten. Ein Knoten ohne Eltern hat nur eine Zeile, die die A-priori-Wahrscheinlichkeiten jedes möglichen Wertes der Variablen darstellt.

Beachten Sie, dass das Netz keine Knoten dafür hat, dass Mary gerade zu laute Musik hört oder dass das Telefon läutet und John verwirrt. Diese Faktoren werden in der Unsicherheit zusammengefasst, die den Verknüpfungen von *Alarm* zu *JohnRuftAn* und *MaryRuftAn* zugeordnet ist. Das zeugt sowohl von Faulheit als auch von Unwissen: Es wäre sehr viel Arbeit, herauszufinden, warum diese Faktoren in einem bestimmten Fall mehr oder weniger wahrscheinlich sind, und wir haben ohnehin keine vernünftige Möglichkeit, die relevanten Informationen zu erhalten. Die Wahrscheinlichkeiten fassen letztlich eine *potenziell unendliche* Menge von Umständen zusammen, in denen der Alarm möglicherweise nicht ausgelöst wird (hohe Luftfeuchtigkeit, Stromausfall, leere Batterie, durchgeschnittene Drähte, eine tote Maus in der Klingel usw.) oder in denen John oder Mary nicht anrufen und von dem Einbruch berichten (Essen gegangen, im Urlaub, vorübergehend taub, vorbeifliegender Hubschrauber usw.). Auf diese Weise kann ein kleiner Agent mit einer sehr großen Welt zurechtkommen, wenigstens annäherungsweise. Der Annäherungsgrad kann verbessert werden, wenn wir zusätzliche relevante Informationen einführen.

14.2 Die Semantik Bayesscher Netze

Der vorige Abschnitt hat beschrieben, was ein Netz ist, aber nicht, was es aussagt. Man kann die Semantik eines Bayesschen Netzes auf zweierlei Arten verstehen. Die erste ist, das Netz als Repräsentation der gemeinsamen Wahrscheinlichkeitsverteilung zu sehen. Die zweite ist, es als Codierung einer Menge bedingter Unabhängigkeitsaussagen zu betrachten. Die beiden Sichten sind äquivalent, aber die erste unterstützt mehr das Verständnis dafür, wie man Netze *aufbaut*, während die zweite hilft, Inferenzprozeduren zu entwerfen.

14.2.1 Darstellung der vollständigen gemeinsamen Verteilung

Als ein Teil der „Syntax“ betrachtet, ist ein Bayessches Netz ein gerichteter azyklischer Graph, dessen Knoten bestimmte numerische Parameter zugeordnet sind. Die Bedeutung des Netzes – seine Semantik – lässt sich zum Beispiel definieren, wenn man die Art und Weise definiert, in der es eine bestimmte gemeinsame Verteilung über allen Variablen darstellt. Dazu müssen wir zuerst (und nur vorübergehend) widerrufen, was wir weiter vorn über die jedem Knoten zugeordneten Parameter gesagt haben – nämlich, dass diese Parameter den bedingten Wahrscheinlichkeiten $\mathbf{P}(X_i \mid \mathit{Eltern}(X_i))$ entsprechen. Dies ist eine wahre Aussage, doch bis wir dem Netz als Ganzes eine Semantik zugewiesen haben, sollten wir sie uns lediglich als Zahlen $\theta(X_i \mid \mathit{Eltern}(X_i))$ vorstellen.

Ein generischer Eintrag in der gemeinsamen Verteilung ist die Wahrscheinlichkeit einer Konjunktion bestimmter Zuweisungen zu jeder Variablen, wie etwa $P(X_1 = x_1 \wedge \ldots \wedge X_n = x_n)$. Wir verwenden hier die abkürzende Notation $P(x_1, \ldots, x_n)$. Der Wert dieses Eintrages ist gegeben durch die Formel

$$P(x_1,\ldots,x_n) = \prod_{i=1}^{n} \theta(x_i \mid \mathit{eltern}(X_i)). \tag{14.1}$$

Dabei gibt $\mathit{eltern}(X_i)$ die spezifischen Werte der Variablen von $\mathit{Eltern}(X_i)$ an, die in $x_1, \ldots, x_n$ erscheinen. Damit wird jeder Eintrag in der gemeinsamen Verteilung durch das Produkt der entsprechenden Elemente der bedingten Wahrscheinlichkeitstabellen (BWTs) im Bayesschen Netz dargestellt.

Aus dieser Definition lässt sich leicht beweisen, dass die Parameter $\theta(X_i \mid \mathit{Eltern}(X_i))$ genau die bedingten Wahrscheinlichkeiten $\mathbf{P}(X_i \mid \mathit{Eltern}(X_i))$ sind, die durch die gemeinsame Verteilung impliziert werden (siehe Übung 14.2). Folglich können wir Gleichung (14.1) als

$$P(x_1,\ldots,x_n) = \prod_{i=1}^{n} P(x_i \mid \mathit{eltern}(X_i)) \tag{14.2}$$

neu schreiben. Mit anderen Worten sind die Tabellen, die wir bedingte Wahrscheinlichkeitstabellen genannt haben, *tatsächlich* bedingte Wahrscheinlichkeitstabellen entsprechend der in Gleichung (14.1) definierten Semantik.

Um dies zu verdeutlichen, können wir die Wahrscheinlichkeit berechnen, dass der Alarm geläutet hat, aber weder ein Einbruch noch ein Erdbeben stattgefunden hat, und sowohl John als auch Mary anrufen. Wir multiplizieren Einträge aus der gemeinsamen Verteilung (und verwenden dabei einbuchstabige Variablennamen – j = *JohnRuftAn*, m = *MaryRuftAn*, a = *Alarm*, b = *Einbruch*, e = *Erdbeben*):

$$P(j \wedge m \wedge a \wedge \neg b \wedge \neg e) = P(j \mid a)P(m \mid a)P(a \mid \neg b \wedge \neg e)P(\neg b)P(\neg e)$$
$$= 0{,}90 \times 0{,}70 \times 0{,}001 \times 0{,}999 \times 0{,}998 = 0{,}00062.$$

Abschnitt 13.3 hat erklärt, dass die vollständige gemeinsame Verteilung genutzt werden kann, um beliebige Abfragen über die Domäne zu beantworten. Wenn ein Bayessches Netz eine Repräsentation der gemeinsamen Verteilung ist, dann kann es ebenfalls verwendet werden, um beliebige Abfragen zu beantworten, indem alle relevanten gemeinsamen Einträge summiert werden. *Abschnitt 14.4* erklärt, wie das geht, beschreibt aber auch sehr viel effizientere Methoden.

Eine Methode zur Erstellung Bayesscher Netze

Gleichung (14.2) definiert, was ein bestimmtes Bayessches Netz bedeutet. Im nächsten Schritt ist zu erklären, wie man ein Bayessches Netz *erstellt*, sodass die resultierende gemeinsame Verteilung eine gute Repräsentation einer bestimmten Domäne darstellt. Wir werden nachfolgend zeigen, dass Gleichung (14.2) bestimmte bedingte Unabhängigkeiten impliziert, die genutzt werden können, um dem Wissensingenieur zu helfen, die Topologie des Netzes aufzubauen. Zuerst schreiben wir die gemeinsame Verteilung als bedingte Wahrscheinlichkeit unter Verwendung der Produktregel (siehe *Abschnitt 13.2.1*):

$$P(x_1, ..., x_n) = P(x_n \mid x_{n-1}, ..., x_1)P(x_{n-1}, ..., x_1).$$

Dann wiederholen wir den Vorgang, wobei wir die konjunktive Wahrscheinlichkeit auf eine bedingte Wahrscheinlichkeit und eine kleinere Konjunktion reduzieren. Am Ende erhalten wir ein großes Produkt:

$$P(x_1, ..., x_n) = P(x_n \mid x_{n-1}, ..., x_1)P(x_{n-1} \mid x_{n-2}, ..., x_1) \ldots P(x_2 \mid x_1)\, P(x_1)$$
$$= \prod_{i=1}^{n} P(x_i \mid x_{i-1}, \ldots, x_1).$$

Diese Identität wird als **Kettenregel** bezeichnet. Sie gilt für beliebige Mengen von Zufallsvariablen. Beim Vergleich mit Gleichung (14.2) sehen wir, dass die Spezifikation der gemeinsamen Verteilung äquivalent mit der allgemeinen Zusicherung ist, dass für jede Variable X_i im Netz gilt:

$$\mathbf{P}(X_i \mid X_{i-1}, ..., X_1) = \mathbf{P}(X_i \mid Eltern(X_i)) \qquad (14.3)$$

vorausgesetzt, $Eltern(X_i) \subseteq \{X_{i-1}, ..., X_1\}$. Diese letzte Bedingung wird erfüllt, indem die Knoten in einer Weise nummeriert werden, die konsistent mit der in der Graphenstruktur impliziten partiellen Ordnung ist.

Gleichung (14.3) besagt, dass das Bayessche Netz nur dann eine korrekte Repräsentation der Domäne ist, wenn jeder Knoten bedingt unabhängig von seinen Vorgängern in der Knotenreihenfolge ist, wenn seine Eltern bekannt sind. Wir können diese Bedingung mit der folgenden Methodologie erfüllen:

1. *Knoten:* Bestimme zuerst die Menge der erforderlichen Variablen, um die Domäne zu modellieren. Jetzt sortiere sie, $\{X_1, ..., X_n\}$. Die Reihenfolge ist prinzipiell beliebig, doch ist das resultierende Netz kompakter, wenn die Variablen so angeordnet sind, dass Ursachen vor Effekten kommen.

2 *Verknüpfungen:* Führe für $i = 1$ bis n aus:

- Wähle von X_1, ..., X_{i-1}, eine minimale Menge von Eltern für X_i, sodass Gleichung (14.3) erfüllt ist.
- Füge für jedes übergeordnete Element eine Verknüpfung vom übergeordneten Element zu X_i ein.
- BWTs: Schreibe die bedingte Wahrscheinlichkeitstabelle $\mathbf{P}(X_i \mid \mathit{Eltern}(X_i))$ auf.

Tipp

Intuitiv betrachtet, sollten die Eltern von Knoten X_i alle Knoten aus X_1, ..., X_{i-1} enthalten, die X_i *direkt beeinflussen*. Nehmen wir beispielsweise an, wir haben das Netz aus Abbildung 14.2 fertiggestellt, aber noch keine Eltern für *MaryRuftAn* ausgewählt. *MaryRuftAn* wird sicher davon beeinflusst, ob es einen *Einbruch* oder ein *Erdbeben* gibt, aber *nicht direkt*. Intuitiv teilt uns unser Wissen über die Domäne mit, dass diese Ereignisse das Anrufverhalten von Mary nur durch ihre Wirkung auf den Alarm beeinflussen. Wenn ein Alarm vorliegt, hat es auch keinen Einfluss auf das Anrufen von Mary, ob John anruft. Formal ausgedrückt glauben wir, dass die folgende bedingte Unabhängigkeitsaussage gilt:

$$\mathbf{P}(\mathit{MaryRuftAn} \mid \mathit{JohnRuftAn}, \mathit{Alarm}, \mathit{Erdbeben}, \mathit{Einbruch}) = \mathbf{P}(\mathit{MaryRuftAn} \mid \mathit{Alarm})$$

Somit ist *Alarm* der einzige übergeordnete Knoten für *MaryRuftAn*.

Tipp

Da jeder Knoten nur mit früheren Knoten verbunden ist, garantiert dieses Konstruktionsverfahren ein azyklisches Netz. Außerdem zeichnen sich Bayessche Netze dadurch aus, dass sie keine redundanten Wahrscheinlichkeitswerte enthalten. Gibt es keine Redundanz, besteht auch keine Möglichkeit für Inkonsistenz: *Der Wissensingenieur oder Domänenexperte kann kein Bayessches Netz erstellen, das die Wahrscheinlichkeitsaxiome verletzt.*

Kompaktheit und Knotenreihenfolge

Ein Bayessches Netz ist nicht nur eine vollständige und nichtredundante Repräsentation der Domäne, sondern häufig auch sehr viel *kompakter* als die vollständige gemeinsame Verteilung. Diese Eigenschaft macht es so praktisch für Domänen mit vielen Variablen. Die Kompaktheit von Bayesschen Netzen ist ein Beispiel für eine sehr allgemeine Eigenschaft **lokal strukturierter** (auch als **schwach besetzt** bezeichneter) Systeme. In einem lokal strukturierten System interagiert jede Unterkomponente mit nur einer begrenzten Anzahl anderer Komponenten direkt, unabhängig von der Gesamtzahl der Komponenten. Eine lokale Struktur ist normalerweise mit einem linearen statt einem exponentiellen Komplexitätswachstum verbunden. Im Fall der Bayesschen Netze kann sinnvoll angenommen werden, dass in den meisten Domänen alle Zufallsvariablen von höchstens k anderen direkt beeinflusst werden (für eine Konstante k). Wenn wir der Einfachheit halber von n booleschen Variablen ausgehen, umfasst die Informationsmenge, die wir brauchen, um jede bedingte Wahrscheinlichkeitstabelle zu spezifizieren, höchstens 2^k Zahlen und das vollständige Netz kann durch $n2^k$ Zahlen spezifiziert werden. Im Gegensatz dazu enthält die gemeinsame Verteilung 2^n Zahlen. Um dies zu konkretisieren, nehmen wir an, wir haben $n = 30$ Knoten mit je fünf Eltern ($k = 5$). Für das Bayessche Netz sind also 960 Zahlen erforderlich – für die vollständige gemeinsame Verteilung über eine Milliarde.

Es gibt Domänen, in denen jede Variable von allen anderen direkt beeinflusst werden kann, sodass das Netz vollständig verbunden ist. Dafür benötigt man für die Angabe

der bedingten Wahrscheinlichkeitstabellen die gleiche Informationsmenge wie für die Angabe der gemeinsamen Verteilung. In einigen Domänen gibt es leichte Abhängigkeiten, die genau genommen berücksichtigt werden sollten, indem man eine neue Verknüpfung einführt. Wenn diese Abhängigkeiten jedoch sehr locker sind, ist es möglicherweise nicht gerechtfertigt, für eine geringfügig erhöhte Genauigkeit eine zusätzliche Komplexität in das Netz einzubringen. Zum Beispiel könnte man unser Einbruchsnetz mit der Begründung beanstanden, dass weder John noch Mary anrufen, selbst wenn sie den Alarm gehört haben, weil sie davon ausgehen, dass das Erdbeben ihn ausgelöst hat. Ob man die Kante von *Erdbeben* zu *JohnRuftAn* und *MaryRuftAn* einfügt (und damit die Tabellen vergrößert), hängt davon ab, wie viel Wert man auf exaktere Wahrscheinlichkeiten legt und dafür die höheren Kosten für die zusätzlichen Informationen in Kauf nimmt.

Selbst in einer lokal strukturierten Domäne erhalten wir ein kompaktes Bayessches Netz nur, wenn wir die Knotenreihenfolge zweckmäßig auswählen. Was passiert, wenn wir die falsche Reihenfolge verwenden? Betrachten wir erneut das Einbruchsbeispiel. Angenommen, wir fügen die Knoten in der Reihenfolge *MaryRuftAn*, *JohnRuftAn*, *Alarm*, *Einbruch*, *Erdbeben* ein. Damit erhalten wir das in ▶ Abbildung 14.3(a) gezeigte, etwas komplexere Netz. Der Prozess verläuft wie folgt:

- *MaryRuftAn* hinzufügen: keine Eltern.
- *JohnRuftAn* hinzufügen: Wenn Mary anruft, bedeutet das vielleicht, dass der Alarm ausgelöst wurde, was natürlich mehr als wahrscheinlich macht, dass John anruft. Aus diesem Grund braucht *JohnRuftAn MaryRuftAn* als Elternknoten.
- *Alarm* hinzufügen: Wenn beide anrufen, ist es natürlich wahrscheinlicher, dass der Alarm ausgelöst wurde, als wenn nur einer oder überhaupt keiner anruft; deshalb brauchen wir sowohl *MaryRuftAn* als auch *JohnRuftAn* als Elternknoten.
- *Einbruch* hinzufügen: Wenn wir den Alarmzustand kennen, könnte uns der Anruf von John oder Mary Informationen über das Telefonklingeln in unserer Wohnung oder Marys Musik mitteilen, aber nichts über einen Einbruch:

  $$\mathbf{P}(Einbruch \mid Alarm, JohnRuftAn, MaryRuftAn) = \mathbf{P}(Einbruch, Alarm)$$

  Wir brauchen also lediglich *Alarm* als Elternknoten.
- *Erdbeben* hinzufügen: Wenn der Alarm läuft, ist es wahrscheinlicher, dass ein Erdbeben stattgefunden hat. (Die Alarmanlage ist eine Art Seismograph.) Wenn wir jedoch wissen, dass ein Einbruch stattgefunden hat, erklärt dies den Alarm und die Wahrscheinlichkeit eines Erdbebens liegt nur leicht über dem Normalwert. Wir brauchen also sowohl *Alarm* als auch *Einbruch* als Elternknoten.

Tipp

Das resultierende Netz hat zwei Verknüpfungen mehr als das ursprüngliche Netz in Abbildung 14.2 und es müssen drei weitere Wahrscheinlichkeiten dafür angegeben werden. Schlimmer ist jedoch, dass einige der Verknüpfungen lockere Beziehungen repräsentieren, für die man schwierige und unnatürliche Wahrscheinlichkeitsbeurteilungen benötigt, wie beispielsweise die Abschätzung der Wahrscheinlichkeit von *Erdbeben* bei *Einbruch* und *Alarm*. Dieses Phänomen ist recht allgemein und hat mit der Unterscheidung zwischen **kausalen** und **diagnostischen** Modellen zu tun, die *Abschnitt 13.5.1* eingeführt hat (siehe auch *Übung 8.14*). Wenn wir versuchen, ein diagnostisches Modell mit Verknüpfungen von Symptomen zu Ursachen zu erstellen (wie von *MaryRuftAn* zu *Alarm* oder von *Alarm* zu *Einbruch*), müssen wir irgendwann zusätzliche Abhängigkeiten zwischen anderweitig voneinander unabhängigen Ursachen spezifizieren (und häu-

fig auch zwischen separat auftretenden Symptomen). *Wenn wir bei einem kausalen Modell bleiben, müssen wir irgendwann weniger Zahlen angeben und die Zahlen sind häufig leichter zu erhalten.* In der Medizindomäne beispielsweise wurde von *Tversky und Kahneman* (1982) gezeigt, dass erfahrene Ärzte lieber Wahrscheinlichkeitsbeurteilungen für kausale Regeln als für diagnostische Regeln abgeben.

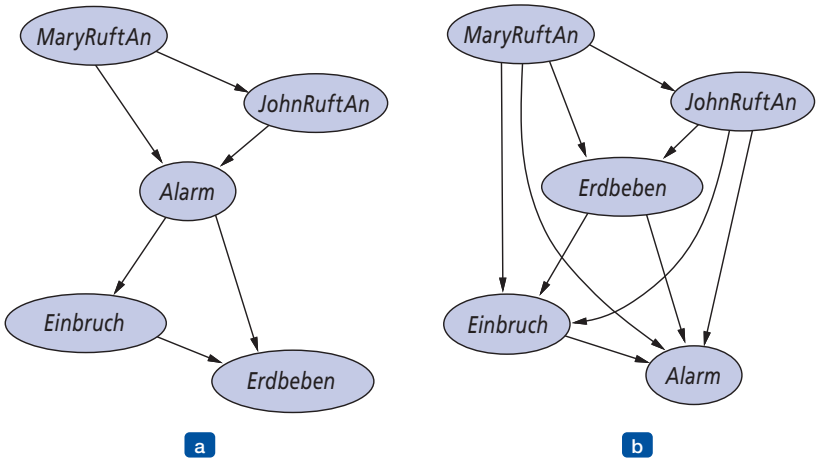


Abbildung 14.3: Die Netzstruktur ist von der Reihenfolge abhängig, in der die Knoten eingefügt werden. Wir haben in jedem Netz die Knoten von oben nach unten eingefügt.

▶ Abbildung 14.3(b) zeigt eine sehr schlechte Knotenreihenfolge: *MaryRuftAn*, *JohnRuftAn*, *Erdbeben*, *Einbruch*, *Alarm*. Für dieses Netz müssen 31 verschiedene Wahrscheinlichkeiten spezifiziert werden – genauso viele wie für die vollständige gemeinsame Verteilung. Beachten Sie jedoch, dass jedes der drei Netze *genau dieselbe gemeinsame Verteilung* darstellen kann. Die beiden letzten Versionen können einfach nicht alle bedingten Unabhängigkeiten darstellen, was schließlich dazu führt, dass sehr viele unnötige Zahlen angegeben werden müssen.

14.2.2 Bedingte Unabhängigkeiten in Bayesschen Netzen

Wir haben eine „numerische" Semantik für Bayessche Netze in Form der Repräsentation der vollständigen gemeinsamen Verteilung gezeigt, wie in Gleichung (14.2) angegeben. Unter Verwendung dieser Semantik für die Ableitung einer Methode zur Erstellung Bayesscher Netze sind wir zu der Erkenntnis gelangt, dass ein Knoten bedingt unabhängig von seinen Vorgängern ist, wenn seine Eltern bekannt sind. Es zeigt sich, dass wir auch in die andere Richtung gehen können. Wir können von einer „topologischen" Semantik ausgehen, die die Beziehungen der bedingten Unabhängigkeiten spezifiziert, die durch die Graphstruktur codiert sind, und von ihnen aus können wir die „numerische" Semantik ableiten. Die topologische Semantik[2] spezifiziert, dass jede

2 Es gibt auch ein allgemeines topologisches Kriterium, **d-Separation**, um zu entscheiden, ob eine Menge von Knoten **X** unabhängig von einer anderen Menge **Y** ist, wenn man eine dritte Menge **Z** kennt. Das Kriterium ist relativ kompliziert und wird für die Ableitung der Algorithmen in diesem Kapitel nicht benötigt; deshalb lassen wir es weg. Details dazu finden Sie bei Pearl (1988) oder Darwiche (2009). Shachter (1998) beschreibt eine intuitivere Methode, d-Separation zuzusichern.

Variable bei bekannten Eltern bedingt unabhängig von ihren Nicht-**Nachkommen** ist. Zum Beispiel ist in Abbildung 14.2 *JohnRuftAn* unabhängig von *Einbruch*, *Erdbeben* und *MaryRuftAn* bei einem bekannten Wert von *Alarm*. ▶ Abbildung 14.4(a) veranschaulicht die Definition. Von diesen Zusicherungen der bedingten Unabhängigkeit und der Interpretation der Netzparameter $\theta(X_i \mid Eltern(X_i))$ als Spezifikationen der bedingten Wahrscheinlichkeiten $\mathbf{P}(X_i \mid Eltern(X_i))$ lässt sich die vollständige gemeinsame Verteilung in Gleichung (14.2) rekonstruieren. In diesem Sinne sind die „numerische“ Semantik und die „topologische“ Semantik äquivalent.

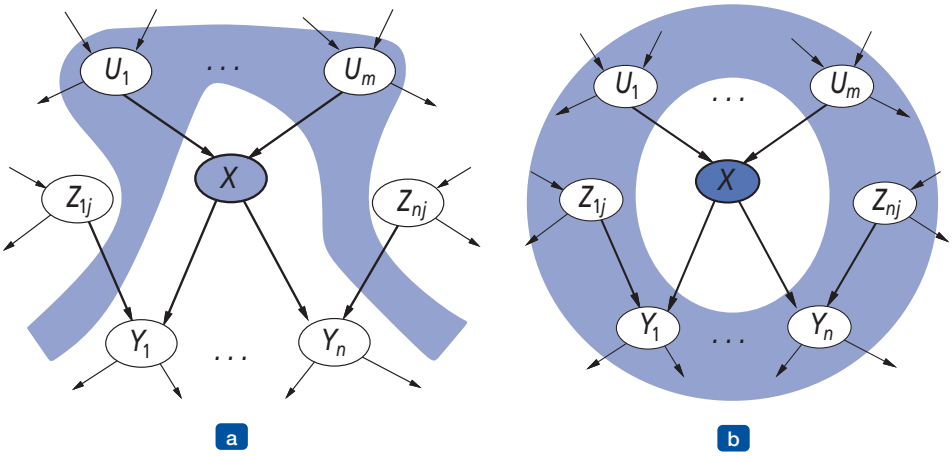


Abbildung 14.4: (a) Ein Knoten X ist bedingt unabhängig von seinen Nichtnachkommen (z.B. den Z_{ij}s) für bekannte Eltern (die U_is im grauen Bereich). (b) Ein Knoten X ist bedingt unabhängig von allen anderen Knoten im Netz, wenn man seine Markov-Decke kennt (der graue Bereich).

Eine andere wichtige Unabhängigkeitseigenschaft wird durch die topologische Semantik impliziert: Ein Knoten ist bedingt unabhängig von allen anderen Knoten im Netz für bekannte Eltern, Kinder und Eltern der Kinder – d.h., wenn man seine **Markov-Decke** kennt. Zum Beispiel ist *Einbruch* unabhängig von *JohnRuftAn* und *MaryRuftAn* für bekannten *Alarm* und bekanntes *Erdbeben*. ▶ Abbildung 14.4(b) veranschaulicht diese Eigenschaft.

14.3 Effiziente Repräsentation bedingter Verteilungen

Selbst wenn die maximale Anzahl der Eltern k sehr klein ist, bedingt das Ausfüllen der BWT für einen Knoten bis zu $O(2^k)$ Zahlen und womöglich umfassende Erfahrung mit allen möglichen Bedingungsfällen. Dies ist das Szenario für den schlechtesten Fall, wobei die Beziehung zwischen den Eltern und den Kindern völlig zufällig ist. Normalerweise können solche Beziehungen durch eine **kanonische Verteilung** beschrieben werden, die einem Standardmuster entspricht. In diesen Fällen kann die vollständige Tabelle spezifiziert werden, indem das Muster angegeben und vielleicht

ein paar Parameter bereitgestellt werden – sehr viel einfacher als die Bereitstellung einer exponentiellen Anzahl von Parametern.

Das einfachste Beispiel sind die **deterministischen Knoten**. Der Wert eines deterministischen Knotens wird exakt durch die Werte seiner Eltern spezifiziert, wobei es keine Unsicherheit gibt. Dabei kann es sich um eine logische Beziehung handeln: Beispielsweise besteht die Beziehung zwischen den Elternknoten *Kanadier*, *US*, *Mexikaner* und dem Kindknoten *NordAmerikaner* einfach darin, dass das Kind die Disjunktion der Eltern ist. Die Beziehung kann auch numerisch sein: Wenn die Elternknoten beispielsweise die Preise eines bestimmen Automodells bei mehreren Händlern sind und der Kindknoten der Preis, den ein Angebotsjäger schließlich zahlt, dann ist der Kindknoten das Minimum der Elternwerte; wenn die Elternknoten die Zuläufe (Flüsse, Quellen, Niederschläge) in einen See und die Abläufe (Flüsse, Verdunstung, Versickerung) aus einem See sind und das Kind die Änderung des Wasserstandes im See, dann ist der Wert des Kindes die Differenz zwischen den zulaufenden und den ablaufenden Eltern.

Unsichere Beziehungen können häufig durch sogenannte „**verrauschte**“ (**noisy**) logische Beziehungen charakterisiert werden. Das Standardbeispiel ist die Relation **Noisy-OR**, eine Verallgemeinerung des logischen OR. In der Aussagenlogik könnten wir sagen, *Fieber* ist genau dann wahr, wenn *Erkältung*, *Grippe* oder *Malaria* wahr sind. Das Noisy-OR-Modell berücksichtigt Unsicherheiten bei der Fähigkeit jedes Elternknotens, den Kindknoten wahr zu machen – die kausale Beziehung zwischen Eltern und Kind könnte *behindert* sein, ein Patient könnte beispielsweise eine Erkältung haben, aber kein Fieber. Das Modell trifft zwei Annahmen. Erstens setzt es voraus, dass alle möglichen Ursachen aufgelistet sind. (Wenn welche fehlen, können wir immer einen sogenannten **Ausgleichsknoten** („**Leak Node**“) hinzufügen, der „verschiedene Ursachen“ abdeckt.) Zweitens nimmt es an, dass die Behinderung jedes Elternknotens unabhängig von den Behinderungen anderer Elternknoten ist, z.B.: Was immer verhindert, dass *Malaria* Fieber verursacht, ist davon unabhängig, was *Grippe* daran hindert, ein Fieber zu verursachen. Unter diesen Annahmen ist *Fieber* genau dann *false*, wenn alle seine *true* Eltern behindert sind, und die Wahrscheinlichkeit davon ist das Produkt der Behinderungswahrscheinlichkeiten q jedes Elternknotens. Angenommen, diese einzelnen Behinderungswahrscheinlichkeiten sehen wie folgt aus:

$$q_{erkältung} = P(\neg fieber \mid erkältung, \neg grippe, \neg malaria) = 0{,}6$$
$$q_{grippe} = P(\neg fieber \mid \neg erkältung, grippe, \neg malaria) = 0{,}2$$
$$q_{malaria} = P(\neg fieber \mid \neg erkältung, \neg grippe, malaria) = 0{,}1.$$

Aus diesen Informationen und den Noisy-OR-Annahmen kann dann die gesamte BWT aufgebaut werden. Die allgemeine Regel lautet, dass

$$P(x_i \mid eltern(X_i)) = 1 - \prod_{\{j:X_j=true\}} q_j$$

ist, wobei das Produkt über die Eltern gebildet wird, die für diese Zeile der BWT auf *true* gesetzt sind. Die folgende Tabelle veranschaulicht diese Berechnung:

Erkältung	Grippe	Malaria	P(Fieber)	P(¬Fieber)
F	F	F	0,0	1,0
F	F	T	0,9	**0,1**
F	T	F	0,8	**0,2**
F	T	T	0,98	$0{,}02 = 0{,}2 \times 0{,}1$
T	F	F	0,4	**0,6**
T	F	T	0,94	$0{,}06 = 0{,}6 \times 0{,}1$
T	T	F	0,88	$0{,}12 = 0{,}6 \times 0{,}2$
T	T	T	0,988	$0{,}012 = 0{,}6 \times 0{,}2 \times 0{,}1$

Im Allgemeinen können verrauschte logische Beziehungen, in denen eine Variable von k Elternknoten abhängig ist, unter Verwendung von $O(k)$ Parametern statt $O(2^k)$ für die gesamte bedingte Wahrscheinlichkeitstabelle beschrieben werden. Das vereinfacht die Abschätzungen und das Lernen. Zum Beispiel verwendet das CPCS-Netz (Pradhan et al., 1994) Noisy-OR- und Noisy-MAX-Verteilungen, um die Beziehungen zwischen Krankheiten und Symptomen in der internistischen Medizin zu modellieren. Bei 448 Knoten und 906 Verknüpfungen sind dafür nur 8254 Werte erforderlich – statt 133.931.430 für ein Netz mit vollständigen BWTs.

Bayessche Netze mit stetigen Variablen

Viele Probleme aus der realen Welt beinhalten stetige Größen wie etwa Höhe, Masse, Temperatur oder Geld; daher beschäftigt sich ein Großteil der Statistik mit Zufallsvariablen stetiger Domänen. Per Definition haben stetige Variablen eine unendliche Anzahl möglicher Werte, deshalb ist es unmöglich, explizit für jeden Wert bedingte Wahrscheinlichkeiten anzugeben. Eine Möglichkeit für den Umgang mit stetigen Variablen ist, sie mithilfe der **Diskretisierung** zu vermeiden – das bedeutet, die möglichen Werte werden in eine feste Menge von Intervallen unterteilt. Beispielsweise kann man Temperaturen in (<0°C), (0°C – 100°C) und (> 100°C) unterteilen. Die Diskretisierung ist manchmal eine geeignete Lösung, resultiert aber häufig in einem wesentlichen Genauigkeitsverlust und sehr großen BWTs. Die gebräuchlichste Lösung ist es, Standardfamilien von Wahrscheinlichkeitsdichtefunktionen zu definieren (siehe Anhang A), die durch eine endliche Anzahl an **Parametern** spezifiziert sind. Beispielsweise hat eine Gaußsche (oder Normal-)Verteilung $N(\mu, \sigma^2)(x)$ den Mittelwert μ und die Varianz σ^2 als Parameter. Eine andere Lösung – die man auch als **nichtparametrische** Darstellung bezeichnet – definiert die Bedingungsverteilung implizit mit einer Auflistung von Instanzen, die jeweils bestimmte Werte der übergeordneten und untergeordneten Variablen enthalten. Mit diesem Ansatz beschäftigen wir uns eingehender in *Kapitel 18*.

Ein Netz mit sowohl diskreten als auch stetigen Variablen wird als **hybrides Bayessches Netz** bezeichnet. Um ein hybrides Netz zu spezifizieren, müssen wir zwei neue Verteilungsarten spezifizieren: die bedingte Verteilung für eine stetige Variable bei diskreten oder stetigen Eltern und die bedingte Verteilung für eine diskrete Variable bei stetigen Eltern. Betrachten Sie das einfache Beispiel in ▶ Abbildung 14.5, wo ein Kunde Obst abhängig von seinem Preis kauft, der wiederum vom Ernteumfang abhängig ist und davon, ob das Subventionsschema der Regierung funktioniert. Die Variable *Kosten* ist stetig und hat stetige und diskrete Eltern; die Variable *Kauft* ist diskret und hat einen stetigen Elternknoten.

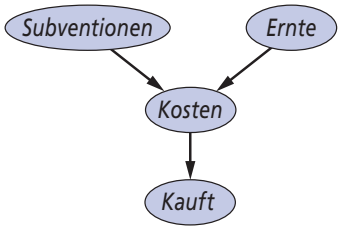


Abbildung 14.5: Ein einfaches Netz mit diskreten Variablen (*Subventionen* und *Kauft*) und stetigen Variablen (*Ernte* und *Kosten*).

Für die Variable *Kosten* müssen wir $\mathbf{P}(Kosten \mid Ernte, Subventionen)$ angeben. Der diskrete Elternknoten wird durch explizite Auflistung verarbeitet – d.h. indem sowohl $P(Kosten \mid Ernte, subventionen)$ als auch $P(Kosten \mid Ernte, \neg subventionen)$ angegeben werden. Um *Ernte* zu verarbeiten, müssen wir angeben, wie die Verteilung über den Kosten c vom stetigen Wert h der *Ernte* abhängig ist. Mit anderen Worten, wir spezifizieren die *Parameter* der Kostenverteilung als Funktion von h. Die gebräuchlichste Wahl ist hier die **lineare Gaußsche** Verteilung, wobei das Kind eine Gaußsche Verteilung hat, deren Mittelwert μ linear mit dem Wert der Eltern variiert und deren Standardabweichung σ fest ist. Wir brauchen zwei Verteilungen, eine für *subventionen* und eine für $\neg subventionen$, mit unterschiedlichen Parametern:

$$P(c \mid h, subventionen) = N(a_t h + b_t, \sigma_t^2)(c) = \frac{1}{\sigma_t \sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{c-(a_t h + b_t)}{\sigma_t}\right)^2}$$

$$P(c \mid h, \neg subventionen) = N(a_f h + b_f, \sigma_f^2)(c) = \frac{1}{\sigma_f \sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{c-(a_f h + b_f)}{\sigma_f}\right)^2} .$$

Für dieses Beispiel wird die bedingte Verteilung für Kosten spezifiziert, indem die lineare Gaußsche Verteilung angegeben und die Parameter a_t, b_t, σ_t, a_f, b_f und σ_f bereitgestellt werden. ▶ Abbildung 14.6(a) und (b) zeigen diese beiden Beziehungen. Beachten Sie, dass die Steigung in jedem Fall negativ ist, weil der Preis sinkt, wenn das Angebot steigt. (Natürlich impliziert die Annahme der Linearität, dass der Preis irgendwann negativ wird; das lineare Modell ist nur dann vertretbar, wenn die Erntegröße auf einen engen Bereich begrenzt ist.) ▶ Abbildung 14.6(c) zeigt die Verteilung $P(c \mid h)$, gemittelt über die beiden möglichen Werte von *Subventionen* und vorausgesetzt, dass jeder die A-priori-Wahrscheinlichkeit 0,5 hat. Das zeigt, dass sich selbst mit sehr einfachen Modellen sehr interessante Verteilungen darstellen lassen.

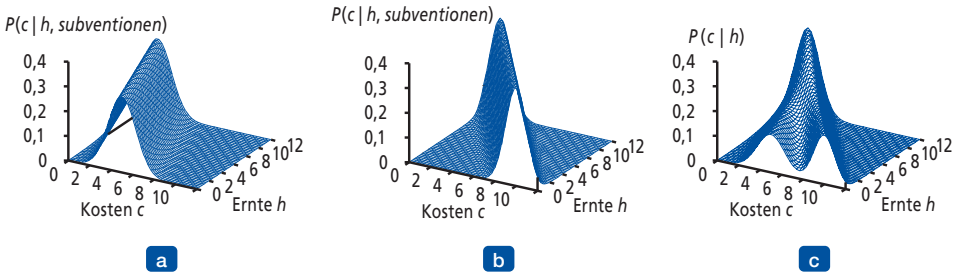


Abbildung 14.6: Die Graphen in (a) und (b) zeigen die Wahrscheinlichkeitsverteilung über *Kosten* als Funktion der Größe von *Ernte*, mit *Subventionen* gleich *true* oder *false*. Graph (c) zeigt die Verteilung $P(Kosten \mid Ernte)$, die sich aus der Summation über die beiden Subventionsfälle ergibt.

Die lineare bedingte Gaußsche Verteilung hat einige besondere Eigenschaften. Ein Netz, das nur stetige Variablen mit linearen Gaußschen Verteilungen enthält, hat eine gemeinsame Verteilung, bei der es sich um eine multivariate Gaußsche Verteilung (siehe Anhang A) über alle Variablen handelt (Übung 14.9). Darüber hinaus besitzt die A-posteriori-Verteilung bei gegebener Evidenz ebenfalls diese Eigenschaft.[3] Wenn diskrete Variablen als übergeordnete (nicht als untergeordnete) Elemente hinzugefügt werden, definiert das Netz eine **bedingte Gaußsche Verteilung**; für jede Zuweisung an die diskreten Variablen ist die Verteilung über die stetigen Variablen eine multivariate Gaußsche Verteilung.

Jetzt wenden wir uns den Verteilungen für diskrete Variablen mit stetigen Eltern zu. Betrachten Sie beispielsweise den Knoten *Kauft* in Abbildung 14.5. Die Annahme scheint sinnvoll, dass der Kunde kauft, wenn die Kosten gering sind, und nicht kauft, wenn sie hoch sind, und dass sich die Wahrscheinlichkeit des Kaufens in einem Zwischenbereich gleichmäßig ändert. Mit anderen Worten, ist die bedingte Verteilung wie eine „weiche" Schwellenfunktion. Weiche Schwellen lassen sich zum Beispiel mithilfe des *Integrals* der Standard-Normalverteilung realisieren:

$$\Phi(x) = \int_{-\infty}^{x} N(0,1)(x)dx.$$

Die Wahrscheinlichkeit von *Kauft* für bekannte *Kosten* könnte dann wie folgt aussehen:

$$P(kauft \mid Kosten = c) = \Phi((-c + \mu)/\sigma).$$

Das bedeutet, die Kostenschwelle liegt bei etwa μ, die Breite des Schwellenbereiches ist proportional zu σ und die Wahrscheinlichkeit des Kaufens sinkt mit steigenden Kosten. Diese **Probit-Verteilung** ist in ▶ Abbildung 14.7(a) dargestellt. Die Form lässt sich rechtfertigen, indem gesagt wird, dass der zugrunde liegende Entscheidungsprozess eine harte Schwelle besitzt, dass jedoch die genaue Position der Schwelle einem zufälligen Gaußschen Rauschen unterliegt.

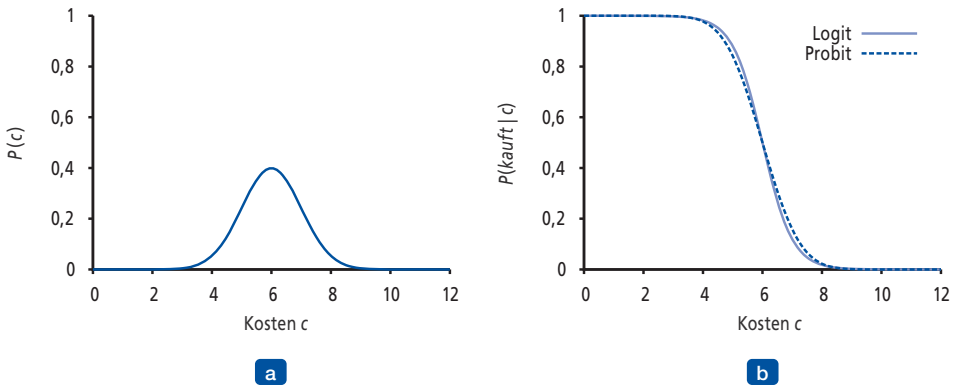


Abbildung 14.7: (a) Eine Normalverteilung (Gaußsche Verteilung) für die Kostenschwelle, zentriert auf $\mu = 6{,}0$ mit der Standardabweichung $\sigma = 1{,}0$. (b) Logit- und Probit-Verteilungen für die Wahrscheinlichkeit von *Kauft* bei gegebenen *Kosten* für die Parameter $\mu = 6{,}0$ und $\sigma = 1{,}0$.

3 Es folgt, dass die Inferenz in linearen Gaußschen Netzen im schlechtesten Fall unabhängig von der Netztopologie eine Zeit von nur $O(n^3)$ benötigt. In *Abschnitt 14.4* sehen wir, dass die Inferenz für Netze diskreter Variablen NP-hart ist.

Eine Alternative zum Probit-Modell ist die **Logit-Verteilung**, die die **logistische Funktion** verwendet, um eine weiche Schwelle zu erzeugen:

$$P(kauft \mid Kosten = c) = \frac{1}{1 + \exp(-2\frac{-c+\mu}{\sigma})}.$$

Dies ist in ▶ Abbildung 14.7(b) gezeigt. Die beiden Verteilungen sehen ähnlich aus, wobei aber die Logit-Verteilung längere „Enden" hat. Die Probit-Verteilung ist oftmals besser für reale Situationen geeignet. Dagegen ist die Logit-Verteilung manchmal mathematisch einfacher zu behandeln und wird häufig in neuronalen Netzen eingesetzt (*Kapitel 20*). Sowohl Probit als auch Logit können verallgemeinert werden, um mehrere stetige Eltern zu verarbeiten, indem eine lineare Kombination der Elternwerte verwendet wird.

14.4 Exakte Inferenz in Bayesschen Netzen

Die grundlegende Aufgabe jedes probabilistischen Inferenzsystems ist die Berechnung der A-posteriori-Wahrscheinlichkeitsverteilung für eine Menge von **Abfragevariablen** für ein bestimmtes beobachtetes **Ereignis** – d.h. eine Zuweisung von Werten an eine Menge von **Evidenzvariablen**. Um die Darstellung zu vereinfachen, betrachten wir nur eine Abfragevariable auf einmal; der Algorithmus lässt sich aber leicht auf Abfragen mit mehreren Variablen erweitern. Wir verwenden die in *Kapitel 13* eingeführte Notation: X bezeichnet die Abfragevariable; $\mathbf{E}$ bezeichnet die Menge der Evidenzvariablen $E_1,...,E_m$, und $\mathbf{e}$ ist ein bestimmtes beobachtetes Ereignis; $\mathbf{Y}$ bezeichnet die Nichtevidenz-, Nichtabfragevariablen $Y_1,...,Y_l$ (manchmal auch als **verborgene Variablen** – engl. **hidden variables** – bezeichnet). Die vollständige Menge der Variablen ist also $\mathbf{X} = \{X\} \cup \mathbf{E} \cup \mathbf{Y}$. Eine typische Abfrage fragt nach der A-posteriori-Wahrscheinlichkeitsverteilung $\mathbf{P}(X \mid \mathbf{e})$.

In unserem Einbruchsbeispiel könnten wir ein Ereignis beobachten, wobei *JohnRuftAn* = *true* und *MaryRuftAn* = *true* sind. Wir könnten dann beispielsweise nach der Wahrscheinlichkeit fragen, dass ein Einbruch stattgefunden hat:

$$\mathbf{P}(Einbruch \mid JohnRuftAn = true, MaryRuftAn = true) = \langle 0{,}284;\ 0{,}716 \rangle.$$

In diesem Abschnitt beschreiben wir exakte Algorithmen für die Berechnung von A-posteriori-Wahrscheinlichkeiten und betrachten die Komplexität dieser Aufgabe. Es zeigt sich, dass der allgemeine Fall nicht handhabbar ist; deshalb beschreibt *Abschnitt 14.5* Methoden für eine annähernde Inferenz.

14.4.1 Inferenz durch Aufzählung

Kapitel 13 hat erklärt, dass jede bedingte Wahrscheinlichkeit berechnet werden kann, indem man die Terme aus der vollständigen gemeinsamen Verteilung summiert. Spezifischer ausgedrückt, eine Abfrage $\mathbf{P}(X \mid \mathbf{e})$ kann mithilfe von Gleichung (13.9) beantwortet werden, die wir hier der Einfachheit halber wiederholen:

$$\mathbf{P}(X \mid \mathbf{e}) = \alpha \mathbf{P}(X, \mathbf{e}) = \alpha \sum_{y} \mathbf{P}(X, \mathbf{e}, \mathbf{y}).$$

Tipp

Ein Bayessches Netz stellt eine vollständige Repräsentation der vollständigen gemeinsamen Verteilung dar. Insbesondere zeigt Gleichung (14.2), dass sich die Terme $P(x, \mathbf{e}, \mathbf{y})$ in der gemeinsamen Verteilung als Produkte bedingter Wahrscheinlichkeiten vom Netz schreiben lassen. Aus diesem Grund *kann eine Abfrage mithilfe eines Bayesschen Netzes beantwortet werden, indem Summen von Produkten bedingter Wahrscheinlichkeiten aus dem Netz berechnet werden.*

Betrachten Sie die Abfrage $\mathbf{P}(Einbruch \mid JohnRuftAn = true, MaryRuftAn = true)$. Die verborgenen Variablen für diese Abfrage sind *Erdbeben* und *Alarm*. Aus Gleichung (13.9) erhalten wir:[4]

$$\mathbf{P}(B \mid j,m) = \alpha\mathbf{P}(B,j,m) = \alpha\sum_e\sum_a \mathbf{P}(B,j,m,e,a).$$

(Um den Ausdruck abzukürzen, verwenden wir nur die Anfangsbuchstaben (B für Burglary – Einbruch).)

Die Semantik Bayesscher Netze (Gleichung (14.2)) liefert uns dann einen Ausdruck in Form von BWT-Ausdrücken. Der Einfachheit halber machen wir dies nur für $Einbruch = true$:

$$P(b \mid j,m) = \alpha\sum_e\sum_a P(b)P(e)P(a \mid b,e),P(j \mid a)P(m \mid a).$$

Um diesen Ausdruck zu berechnen, müssen wir vier Terme einfügen, die jeweils durch Multiplikation von fünf Zahlen berechnet werden. Im schlimmsten Fall müssen wir fast alle Variablen summieren und die Komplexität des Algorithmus für ein Netz mit n booleschen Variablen beträgt $O(n2^n)$.

Die folgenden einfachen Beobachtungen lassen eine Verbesserung zu: Der Term $P(b)$ ist eine Konstante und kann aus den Summationen über a und e heraus verschoben werden und der Term $P(e)$ kann aus der Summation über a heraus verschoben werden. Wir erhalten also:

$$P(b \mid j,m) = \alpha P(b)\sum_e P(e)\sum_a P(a \mid b,e),P(j \mid a)P(m \mid a). \tag{14.4}$$

Um diesen Ausdruck auszuwerten, durchläuft man die Variablen in einer Schleife und multipliziert dabei die BWT-Einträge. Darüber hinaus brauchen wir für jede Summation eine Schleife über die möglichen Werte der Variablen. ▶ Abbildung 14.8 zeigt die Struktur dieser Berechnung. Mit den Zahlen von Abbildung 14.2 erhalten wir $P(b \mid j, m) = \alpha \times 0{,}00059224$. Die entsprechende Berechnung für $\neg b$ ergibt $\alpha \times 0{,}0014919$, also:

$$\mathbf{P}(B \mid j, m) = \alpha\langle 0{,}00059224; 0{,}0014919\rangle \approx \langle 0{,}284; 0{,}716\rangle.$$

Wenn beide Nachbarn anrufen, liegt also die Wahrscheinlichkeit eines Einbruches bei etwa 28%.

Der Auswertungsprozess für den Ausdruck in Gleichung (14.4) ist in Abbildung 14.8 als Ausdrucksbaum dargestellt. Der Algorithmus ENUMERATION-ASK in ▶ Abbildung 14.9 wertet diese Bäume unter Verwendung einer Tiefensuch-Rekursion aus. Der Algorithmus

4 Ein Ausdruck wie $\sum_e P(a,e)$ bedeutet, dass $P(A = e, E = e)$ für alle möglichen Werte von e summiert wird. Wenn E ein boolescher Wert ist, gibt es eine Mehrdeutigkeit: $P(e)$ kann sowohl $P(E = true)$ als auch $P(E = e)$ bedeuten. Aus dem Kontext sollte aber deutlich werden, was davon gemeint ist; insbesondere ist im Kontext einer Summe das Letztere gemeint.

ähnelt in der Struktur dem Backtracking-Algorithmus für das Lösen von CSPs (Abbildung 6.5) und dem DPLL-Algorithmus für Erfüllbarkeitsprobleme (Abbildung 7.17).

Die Speicherkomplexität von ENUMERATION-ASK ist nur linear zur Anzahl der Variablen: Der Algorithmus summiert über die vollständige gemeinsame Verteilung, ohne sie je explizit zu erstellen. Leider ist die Zeitkomplexität für ein Netz mit n booleschen Variablen immer $O(2^n)$ – besser als die $O(n2^n)$ für den zuvor vorgestellten, einfachen Ansatz, aber immer noch schlimm genug.

Beachten Sie bei dem Baum in Abbildung 14.8, dass er die *wiederholten Unterausdrücke*, die durch den Algorithmus ausgewertet werden, explizit macht. Die Produkte $P(j \mid a)P(m \mid a)$ und $P(j \mid \neg a)P(m \mid \neg a)$ werden zweimal berechnet, je einmal für jeden Wert von e. Der nächste Abschnitt beschreibt eine allgemeine Methode, die diese Verschwendung an Rechenzeit vermeidet.

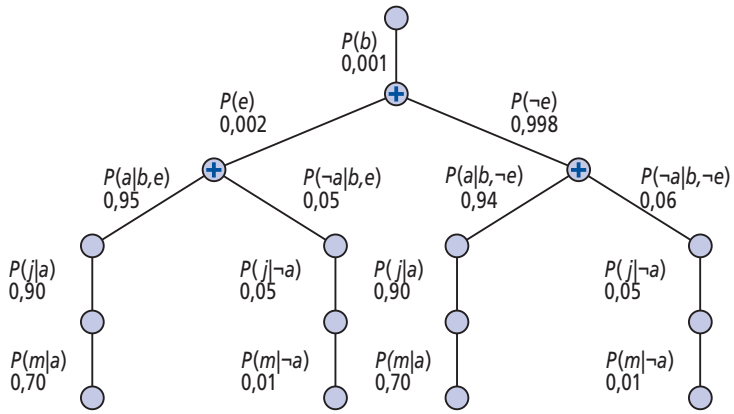


Abbildung 14.8: Die Struktur des in Gleichung (14.4) gezeigten Ausdrucks. Die Auswertung verläuft von oben nach unten, multipliziert entlang des Pfades Werte und summiert die „+"-Knoten. Beachten Sie die Wiederholung der Pfade für j und m.

```
function ENUMERATION-ASK(X, e, bn) returns eine Verteilung über X
  inputs: X, die Abfragevariable
          e, beobachtete Werte für die Variablen E
          bn, ein Bayessches Netz mit den Variablen {X} ∪ E ∪ Y /* Y =
              verborgene Variablen */

  Q(X) ← eine Verteilung über X, anfangs leer
  for each Wert x_i von X do
    Q(x_i) ← ENUMERATE-ALL(bn.VARS, e_{x_i})
      wobei e_xi gleich e erweitert mit X = x_i ist
  return NORMALIZE(Q(X))
```

```
function ENUMERATE-ALL(vars, e) returns eine reelle Zahl
  if EMPTY?(vars) then return 1.0
  Y ← FIRST(vars)
  if Y hat den Wert y in e
    then return P(y | parents(Y)) × ENUMERATE-ALL(REST(vars), e)
    else return Σ_y P(y | parents(Y)) × ENUMERATE-ALL(REST(vars), e_y)
      wobei e_y gleich e erweitert mit Y = y ist
```

Abbildung 14.9: Der Aufzählungsalgorithmus zur Beantwortung von Abfragen zu Bayesschen Netzen.

14.4.2 Der Algorithmus zur Variableneliminierung

Der Aufzählungsalgorithmus kann wesentlich verbessert werden, wenn man wiederholte Berechnungen eliminiert, wie sie in Abbildung 14.8 gezeigt sind. Die Idee dabei ist ganz einfach: Man führt die Berechnung einmal aus und bewahrt die Ergebnisse für später auf. Dies ist eine Art dynamischer Programmierung. Es gibt mehrere Versionen dieses Ansatzes; wir stellen hier den **variableneliminierenden Algorithmus** vor – den einfachsten. Die Variableneliminierung benutzt Evaluierungsausdrücke wie etwa Gleichung (14.4) von *rechts nach links* (d.h. in Abbildung 14.8 von *unten nach oben*). Zwischenergebnisse werden gespeichert und Summationen über die einzelnen Variablen erfolgen nur für die Abschnitte des Ausdruckes, die von der Variablen abhängig sind.

Wir wollen diesen Prozess für das Einbruchsnetz zeigen. Wir werten den folgenden Ausdruck aus:

$$\mathbf{P}(B \mid j,m) = \underbrace{\alpha \mathbf{P}(B)}_{\mathbf{f}_1} \sum_e \underbrace{P(e)}_{\mathbf{f}_2(E)} \sum_a \underbrace{\mathbf{P}(a \mid B,e)}_{\mathbf{f}_3(A,B,E)} \underbrace{P(j \mid a)}_{\mathbf{f}_4(A)} \underbrace{P(m \mid a)}_{\mathbf{f}_5(A)}.$$

Beachten Sie, dass wir jeden Teil des Ausdruckes mit dem Namen des zugehörigen **Faktors** gekennzeichnet haben; jeder Faktor ist eine Matrix, die durch die Werte ihrer Argumentvariablen indiziert ist. Zum Beispiel hängen die Faktoren $\mathbf{f}_4(A)$ und $\mathbf{f}_5(A)$, die $P(j \mid a)$ und $P(m \mid a)$ entsprechen, nur von A ab, weil J und M durch die Abfrage festgelegt sind. Demnach sind sie zweielementige Vektoren:

$$\mathbf{f}_4(A) = \begin{pmatrix} P(j \mid a) \\ P(j \mid \neg a) \end{pmatrix} = \begin{pmatrix} 0{,}90 \\ 0{,}05 \end{pmatrix} \qquad \mathbf{f}_5(A) = \begin{pmatrix} P(m \mid a) \\ P(m \mid \neg a) \end{pmatrix} = \begin{pmatrix} 0{,}70 \\ 0{,}01 \end{pmatrix}.$$

Der Faktor $\mathbf{f}_3(A, B, E)$ ist eine $2 \times 2 \times 2$-Matrix, die sich auf der gedruckten Seite nur schwer zeigen lässt. (Das „erste“ Element ist durch $P(a \mid b, c) = 0{,}95$ gegeben und das „letzte“ durch $P(\neg a \mid \neg b, \neg e) = 0{,}999$.) In Form von Faktoren wird die Abfrage geschrieben als

$$\mathbf{P}(B \mid j,m) = \alpha \mathbf{f}_1(B) \times \sum_e \mathbf{f}_2(E) \times \sum_a \mathbf{f}_3(A,B,E) \times \mathbf{f}_4(A) \times \mathbf{f}_5(A),$$

wobei der Operator $\times$ keine gewöhnliche Matrixmultiplikation bezeichnet, sondern das **punktweise definierte Produkt**, auf das wir gleich eingehen.

Bei der Auswertung werden die Variablen (von rechts nach links) aus punktweise gebildeten Produkten von Faktoren aussummiert, um neue Faktoren zu erzeugen, die letztlich einen Faktor als die Lösung liefern, d.h. die A-posteriori-Verteilung über die Abfragevariable. Die Schritte sehen folgendermaßen aus:

- Zuerst summieren wir A aus dem Produkt von $\mathbf{f}_3$, $\mathbf{f}_3$ und $\mathbf{f}_5$ aus. Damit erhalten wir einen neuen 2×2-Faktor $\mathbf{f}_6(B, E)$, dessen Indizes sich nur über B und E erstrecken:

$$\mathbf{f}_6(B,E) = \sum_a \mathbf{f}_3(A,B,E) \times \mathbf{f}_4(A) \times \mathbf{f}_5(A)$$

$$= (\mathbf{f}_3(a, B, E) \times \mathbf{f}_4(a) \times \mathbf{f}_5(a)) + (\mathbf{f}_3(\neg a, B, E) \times \mathbf{f}_4(\neg a) \times \mathbf{f}_5(\neg a)).$$

  Damit bleibt der folgende Ausdruck übrig:

$$\mathbf{P}(B \mid j,m) = \alpha \mathbf{f}_1(B) \sum_e \mathbf{f}_2(E) \times \mathbf{f}_6(B,E).$$

- Als Nächstes summieren wir E aus dem Produkt von $\mathbf{f}_2$ und $\mathbf{f}_6$ aus:

$$\mathbf{f}_7(B) = \sum_e \mathbf{f}_2(E) \times \mathbf{f}_6(B, E)$$

$$= \mathbf{f}_2(e) \times \mathbf{f}_6(B, e) + \mathbf{f}_2(\neg e) \times \mathbf{f}_6(B, \neg e).$$

Damit bleibt der Ausdruck

$$\mathbf{P}(B \mid j, m) = \alpha \mathbf{f}_1(B) \times \mathbf{f}_7(B)$$

übrig, der sich auswerten lässt, indem das punktweise definierte Produkt gebildet und das Ergebnis normalisiert wird.

Ein genauer Blick auf diesen Ablauf zeigt, dass zwei grundlegende Berechnungsoperationen erforderlich sind: punktweise gebildetes Produkt aus einem Faktorenpaar und Aussummieren einer Variablen aus einem Produkt von Faktoren. Der nächste Abschnitt beschreibt diese Operationen.

Operationen auf Faktoren

Das punktweise definierte Produkt von zwei Faktoren $\mathbf{f}_1$ und $\mathbf{f}_2$ ergibt einen neuen Faktor $\mathbf{f}$, dessen Variablen die Vereinigungsmenge der Variablen in $\mathbf{f}_1$ und $\mathbf{f}_2$ sind und deren Elemente durch das Produkt der entsprechenden Elemente in den beiden Faktoren gegeben sind. Angenommen, die beiden Faktoren haben die gemeinsamen Variablen $Y_1, \ldots Y_k$. Dann haben wir

$$\mathbf{f}(X_1 \ldots X_j, Y_1 \ldots Y_k, Z_1 \ldots Z_l) = \mathbf{f}_1(X_1 \ldots X_j, Y_1 \ldots Y_k)\, \mathbf{f}_2(Y_1 \ldots Y_k, Z_1 \ldots Z_l).$$

Wenn alle Variablen binär sind, verfügen $\mathbf{f}_1$ und $\mathbf{f}_2$ über 2^{j+k} bzw. 2^{k+l} Einträge und das punktweise Produkt hat 2^{j+k+l} Einträge. Zum Beispiel hat das punktweise Produkt $\mathbf{f}_1 \times \mathbf{f}_2 = \mathbf{f}_3(A, B, C)$ für die beiden gegebenen Faktoren $\mathbf{f}_1(A, B)$ und $\mathbf{f}_2(B, C)$ wie in ▶ Abbildung 14.10 gezeigt $2^{1+1+1} = 8$ Einträge. Beachten Sie, dass der aus einem punktweise definierten Produkt resultierende Faktor mehr Variablen enthalten kann als irgendeiner der Faktoren, die in die Multiplikation eingehen, und die Größe eines Faktors exponentiell zur Anzahl der Variablen ist. Hierdurch entstehen beim Algorithmus zur Variableneliminierung sowohl Platz- als auch Zeitkomplexität.

A	B	$\mathbf{f}_1(A, B)$	B	C	$\mathbf{f}_2(B, C)$	A	B	C	$\mathbf{f}_3(A, B, C)$
T	T	0,3	T	T	0,2	T	T	T	$0{,}3 \times 0{,}2 = 0{,}06$
T	F	0,7	T	F	0,8	T	T	F	$0{,}3 \times 0{,}8 = 0{,}24$
F	T	0,9	F	T	0,6	T	F	T	$0{,}7 \times 0{,}6 = 0{,}42$
F	F	0,1	F	F	0,4	T	F	F	$0{,}7 \times 0{,}4 = 0{,}28$
						F	T	T	$0{,}9 \times 0{,}2 = 0{,}18$
						F	T	F	$0{,}9 \times 0{,}8 = 0{,}72$
						F	F	T	$0{,}1 \times 0{,}6 = 0{,}06$
						F	F	F	$0{,}1 \times 0{,}4 = 0{,}04$

Abbildung 14.10: Veranschaulichung der punktweisen Multiplikation $\mathbf{f}_1(A, B) \times \mathbf{f}_2(B, C) = \mathbf{f}_3(A, B, C)$.

Eine Variable wird aus einem Produkt von Faktoren aussummiert, indem die Teilmatrizen addiert werden, die entstehen, indem die einzelnen Variablen nacheinander auf ihre Werte festgelegt werden. Um zum Beispiel A aus $\mathbf{f}_3(A, B, C)$ auszusummieren, schreiben wir

$$\mathbf{f}(B,C) = \sum_a \mathbf{f}_3(A,B,C) = \mathbf{f}_3(a,B,C) + \mathbf{f}_3(\neg a,B,C)$$
$$= \begin{pmatrix} 0{,}06 & 0{,}24 \\ 0{,}42 & 0{,}28 \end{pmatrix} + \begin{pmatrix} 0{,}18 & 0{,}72 \\ 0{,}06 & 0{,}04 \end{pmatrix} = \begin{pmatrix} 0{,}24 & 0{,}96 \\ 0{,}48 & 0{,}32 \end{pmatrix}.$$

Der einzige Trick dabei ist, zu erkennen, dass jeder Faktor, der *nicht* von der auszusummierenden Variablen abhängig ist, aus dem Summationsprozess herausgeschoben werden kann. Wenn wir beispielsweise zuerst E in Einbruchsnetz aussummieren wollen, lautet der relevante Teil des Ausdruckes

$$\sum_e \mathbf{f}_2(E) \times \mathbf{f}_3(A,B,E) \times \mathbf{f}_4(A) \times \mathbf{f}_5(A) = \mathbf{f}_4(A) \times \mathbf{f}_5(A) \times \sum_e \mathbf{f}_2(E) \times \mathbf{f}_3(A,B,E).$$

Jetzt wird das punktweise Produkt innerhalb der Summation berechnet und die Variable aus der resultierenden Matrix aussummiert.

Beachten Sie, dass die Matrizen *nicht* multipliziert werden, bis wir eine Variable aus dem akkumulierten Produkt aussummieren müssen. Momentan multiplizieren wir nur die Matrizen, die die auszusummierende Variable enthalten. Mit Funktionen für das punktweise definierte Produkt und das Aussummieren lässt sich der eigentliche Algorithmus für die Variableneliminierung wie in ▶ Abbildung 14.11 gezeigt recht einfach formulieren:

```
function ELIMINATION-ASK(X, e, bn) returns eine Verteilung über X
  inputs: X, die Abfragevariable
          e, beobachtete Werte für die Variablen E
          bn, ein Bayessches Netz, das eine gemeinsame Verteilung
              P(X1,..., Xn) spezifiziert

  factors ← [ ]
  for each var in ORDER(bn.VARS) do
    factors ← [MAKE-FACTOR(var, e) | factors]
    if var ist eine versteckte Variable then factors ← SUM-OUT(var, factors)
  return NORMALIZE(POINTWISE-PRODUCT(factors))
```

Abbildung 14.11: Der Algorithmus zur Variableneliminierung für Inferenz in Bayesschen Netzen.

Variablenreihenfolge und Variablenrelevanz

Der Algorithmus in Abbildung 14.11 bindet eine nicht spezifizierte ORDER-Funktion ein, um eine Reihenfolge für die Variablen auszuwählen. Zwar liefert jede Auswahl einer Reihenfolge einen gültigen Algorithmus, doch können unterschiedliche Reihenfolgen dazu führen, dass während der Berechnung unterschiedliche Zwischenfaktoren erzeugt werden. Zum Beispiel haben wir in der oben gezeigten Berechnung A vor E eliminiert; beim entgegengesetzten Vorgehen wird die Berechnung zu

$$\mathbf{P}(B \mid j,m) = \alpha \mathbf{f}_1(B) \times \sum_a \mathbf{f}_4(A) \times \sum_e \mathbf{f}_2(E) \times \mathbf{f}_3(A,B,E).$$

Im Zuge der Berechnung wird ein neuer Faktor $\mathbf{f}_6(A, B)$ generiert.

Im Allgemeinen ergeben sich die Zeit- und Platzanforderungen der Variableneliminierung aus der Größe des größten Faktors, den der Algorithmus während der Berechnung konstruiert. Dies wird wiederum durch die Reihenfolge der Variableneliminierung und durch die Struktur des Netzes bestimmt. Es zeigt sich, dass sich die optimale Reihenfolge nur schwer ermitteln lässt, jedoch mehrere sehr gute Heuristiken zur Verfügung stehen. Eine recht effektive Methode ist eine „gierige" (greedy): Eliminieren derjenigen Variablen, die die Größe des nächsten zu konstruierenden Faktors minimiert.

Betrachten wir eine weitere Abfrage: $\mathbf{P}(JohnRuftAn \mid Einbruch = true)$. Wie üblich ist der erste Schritt, die verschachtelte Summation aufzuschreiben:

$$\mathbf{P}(J \mid b) = \alpha P(b) \sum_e P(e) \sum_a P(a \mid b,e) \mathbf{P}(J \mid a) \sum_m P(m \mid a).$$

Tipp

Wenn wir diesen Ausdruck von rechts nach links auswerten, erkennen wir etwas Interessantes: $\Sigma_m P(m \mid a)$ ist per Definition gleich 1! Damit war es nicht nötig, es überhaupt aufzunehmen; die Variable M ist für diese Abfrage *irrelevant*. Man könnte auch sagen, das Ergebnis der Abfrage $P(JohnRuftAn \mid Einbruch = true)$ bleibt unverändert, wenn wir *MaryRuftAn* ganz aus dem Netz entfernen. Im Allgemeinen können wir jeden Blattknoten entfernen, bei dem es sich nicht um eine Abfragevariable oder eine Evidenzvariable handelt. Nach dem Entfernen gibt es vielleicht weitere Blattknoten, die möglicherweise ebenfalls irrelevant sind. Wenn wir diesen Prozess fortsetzen, stellen wir irgendwann fest, *dass jede Variable, die kein Vorfahre einer Abfragevariablen oder Evidenzvariablen ist, für die Abfrage irrelevant ist.* Ein Algorithmus zur Variableneliminierung kann also alle diese Variablen entfernen, bevor die Abfrage ausgewertet wird.

14.4.3 Die Komplexität exakter Inferenz

Tipp

Die Komplexität exakter Inferenz in Bayesschen Netzen hängt stark von der Struktur des Netzes ab. Das Einbruchsnetz aus Abbildung 14.2 gehört zur Familie der Netze, in denen es höchstens einen ungerichteten Pfad zwischen zwei beliebigen Knoten im Netz gibt. Man spricht auch von **einfach verbundenen Netzen** oder **Polybäumen**, die eine besonders erfreuliche Eigenschaft haben: *Die Zeit- und Speicherkomplexität der exakten Inferenz in Polybäumen ist linear zur Größe des Netzes.* Dabei ist die Größe als die Anzahl der BWT-Einträge definiert; wenn die Anzahl der Eltern jedes Knotens durch eine Konstante begrenzt ist, ist die Komplexität auch linear zur Anzahl der Knoten.

Tipp

Für **mehrfach verbundene Netze**, wie in ▶ Abbildung 14.12(a) gezeigt, kann die Variableneliminierung im schlechtesten Fall eine exponentielle Zeit- und Speicherkomplexität aufweisen, selbst wenn die Anzahl der Eltern pro Knoten begrenzt ist. Das ist nicht überraschend, wenn man berücksichtigt, *dass die Inferenz in Bayesschen Netzen NP-hart ist, weil sie die Inferenz in der Aussagenlogik als Sonderfall beinhaltet.* Tatsächlich kann gezeigt werden (Übung 14.15), dass das Problem so schwierig ist wie die Berechnung der *Anzahl* der geeigneten Zuweisungen für eine aussagenlogische Logikformel. Das bedeutet, dass es #P-hart („P-Zahl-hart") ist – d.h. streng schwieriger als NP-vollständige Probleme.

Es gibt eine enge Verbindung zwischen der Komplexität der Inferenz in Bayesschen Netzen und der Komplexität von Problemen unter Rand- und Nebenbedingungen (CSPs). Wie in *Kapitel 6* beschrieben, hat die Schwierigkeit, ein diskretes CSP zu lösen, damit zu tun, wie „baumähnlich" der Bedingungsgraph ist. Maße wie etwa die

Baumbreite, die die Komplexität der CSP-Lösung begrenzen, können auch direkt auf Bayessche Netze angewendet werden. Darüber hinaus kann der Algorithmus für die Variableneliminierung so verallgemeinert werden, dass er sowohl CSPs als auch Bayessche Netze löst.

14.4.4 Clustering-Algorithmen

Der Algorithmus zur Variableneliminierung ist einfach und effizient, um einzelne Abfragen zu beantworten. Wenn wir jedoch bedingte Wahrscheinlichkeiten für alle Variablen in einem Netz berechnen wollen, kann er weniger effizient sein. In einem Polybaum-Netz beispielsweise müsste man $O(n)$ Abfragen bearbeiten, die jeweils $O(n)$ kosten, was eine Gesamtzeit von $O(n^2)$ ergibt. Mithilfe von **Clustering**-Algorithmen (auch als **gemeinsame Baum-Algorithmen** bezeichnet) kann die Zeit auf $O(n)$ reduziert werden. Aus diesem Grund werden diese Algorithmen in kommerziellen Tools für Bayessche Netze häufig eingesetzt.

Die grundlegende Idee beim Clustering ist, einzelne Knoten des Netzes zu verbinden, damit sie Cluster-Knoten bilden – sodass das resultierende Netz ein Polybaum ist. Beispielsweise kann das mehrfach verbundene Netz in Abbildung 14.12(a) in einen Polybaum umgewandelt werden, indem man die Knoten *Sprinkler* und *Regen* zu einem Cluster-Knoten *Sprinkler+Regen* zusammenfasst, wie in ▶ Abbildung 14.12(b) gezeigt. Die beiden booleschen Knoten werden durch einen „Megaknoten" ersetzt, der vier mögliche Werte annehmen kann: *tt*, *tf*, *ft* und *ff*. Der Megaknoten hat nur einen Elternknoten, die boolesche Variable *Wolkig*; deshalb gibt es zwei Bedingungsfälle. Obwohl im Beispiel nicht gezeigt, produziert der Clustering-Prozess Megaknoten, die bestimmte gemeinsame Variablen aufweisen.

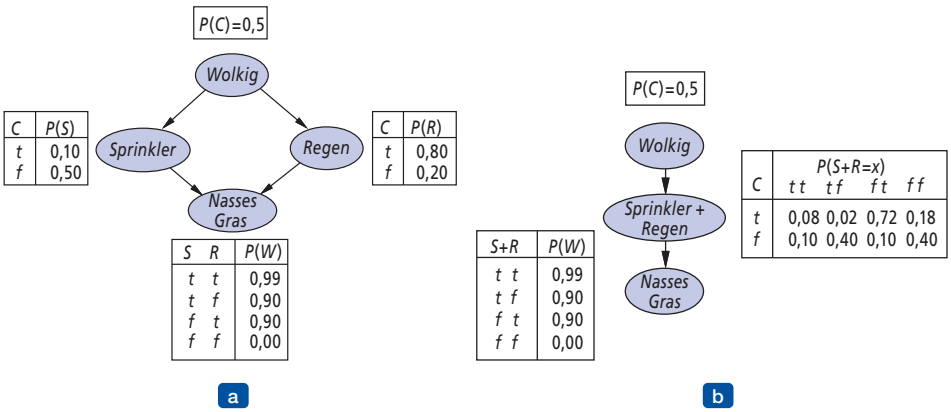


Abbildung 14.12: (a) Ein mehrfach verbundenes Netz mit bedingten Wahrscheinlichkeitstabellen. (b) Ein geclustertes Äquivalent zu dem mehrfach verbundenen Netz.

Nachdem sich das Netz in Polybaum-Form befindet, ist ein spezieller Inferenzalgorithmus erforderlich, weil gewöhnliche Inferenzmethoden keine Megaknoten behandeln können, die untereinander Variablen gemeinsam nutzen. Im Wesentlichen ist der Algorithmus eine Form der Bedingungspropagation (siehe *Kapitel 6*), wo die Bedingungen sicherstellen, dass benachbarte Megaknoten mit den A-posteriori-Wahrscheinlichkeiten aller gemeinsamen Variablen konform sind. Bei sorgfältiger Buchführung ist dieser

Algorithmus in der Lage, bedingte Wahrscheinlichkeiten für alle Nichtevidenzknoten im Netz in einer Zeit zu berechnen, die linear mit der Größe des geclusterten Netzes zunimmt. Die NP-Härte des Problems ist jedoch nicht verschwunden: Wenn Zeit- und Speicherbedarf eines Netzes bei Variableneliminierung exponentiell wachsen, nimmt der Umfang der BWTs im geclusterten Netz zwangsläufig exponentiell zu.

14.5 Annähernde Inferenz in Bayesschen Netzen

Aufgrund der Nichthandhabbarkeit exakter Inferenz in großen, mehrfach verbundenen Netzen ist es wichtig, annähernde Inferenzmethoden zu betrachten. Dieser Abschnitt beschreibt Algorithmen für zufällig ausgewählte Stichproben (Sampling), auch als **Monte-Carlo-Algorithmen** bezeichnet, die annähernde Antworten bereitstellen, deren Genauigkeit von der Anzahl der erzeugten Stichproben abhängig ist. Monte-Carlo-Algorithmen, für die das in *Abschnitt 4.1.2* beschriebene Simulated Annealing ein Beispiel ist, werden in vielen Wissenschaftsgebieten genutzt, um Größen abzuschätzen, für die eine genaue Berechnung schwierig ist. In diesem Abschnitt interessieren wir uns für das auf die Berechnung von A-posteriori-Wahrscheinlichkeiten angewendete Sampling. Wir beschreiben zwei Algorithmenfamilien: direktes Sampling und Markov-Ketten-Sampling. Zwei weitere Ansätze – Variationsmethoden und Schleifenpropagation – werden in den Hinweisen am Ende des Kapitels erwähnt.

14.5.1 Direkte Sampling-Methoden

Die elementare Komponente in jedem Sampling-Algorithmus ist das Erstellen von Stichproben (Samples) aus einer bekannten Wahrscheinlichkeitsverteilung. Beispielsweise kann man sich eine perfekte Münze als Zufallsvariable *Münze* mit den Werten $\langle kopf, zahl \rangle$ und einer A-priori-Verteilung $\mathbf{P}(Münze) = \langle 0{,}5; 0{,}5 \rangle$ vorstellen. Das Sampling aus dieser Verteilung ist genau so, als würde man die Münze werfen: Mit einer Wahrscheinlichkeit von 0,5 gibt sie *kopf* zurück, mit einer Wahrscheinlichkeit von 0,5 gibt sie *zahl* zurück. Für eine Quelle von Zufallszahlen, die im Bereich [0,1] gleichverteilt sind, ist es ganz einfach, jede beliebige Verteilung für eine einzige Variable – egal ob diskret oder stetig – zu sampeln (siehe Übung 14.16).

Die einfachste Art eines zufälligen Sampling-Prozesses für Bayessche Netze erzeugt Ereignisse aus einem Netz, denen keine Evidenz zugeordnet ist. Die Idee ist, jede Variable nacheinander in topologischer Reihenfolge zu sampeln. Die Wahrscheinlichkeitsverteilung, aus der der Wert gesampelt wird, ist von den Werten abhängig, die den Elternknoten der Variablen bereits zugeordnet sind. Dieser Algorithmus ist in ▶ Abbildung 14.13 gezeigt. Wir können seine Ausführung für das in Abbildung 14.12(a) gezeigte Netz beschreiben, wobei wir die Reihenfolge [*Wolkig*, *Sprinkler*, *Regen*, *NassesGras*] annehmen:

1. Stichprobe aus $\mathbf{P}(Wolkig) = \langle 0{,}5; 0{,}5 \rangle$; Wert ist *true*.
2. Stichprobe aus $\mathbf{P}(Sprinkler \mid Wolkig = true) = \langle 0{,}1; 0{,}9 \rangle$; Wert ist *false*.
3. Stichprobe aus $\mathbf{P}(Regen \mid Wolkig = true) = \langle 0{,}8; 0{,}2 \rangle$; Wert ist *true*.
4. Stichprobe aus $\mathbf{P}(NassesGras \mid Sprinkler = false, Regen = true) = \langle 0{,}9; 0{,}1 \rangle$; Wert ist *true*.

In diesem Fall gibt PRIOR-SAMPLE das Ereignis [*true*, *false*, *true*, *true*] zurück.

Man sieht leicht, dass PRIOR-SAMPLE Stichproben aus der gemeinsamen A-priori-Verteilung erzeugt, die durch das Netz spezifiziert wird. Es sei zuerst $S_{PS}(x_1, ..., x_n)$ die Wahrscheinlichkeit, dass der Algorithmus PRIOR-SAMPLE ein bestimmtes Ereignis erzeugt. *Wenn wir den Sampling-Prozess betrachten*, haben wir

$$S_{PS}(x_1 \ldots x_n) = \prod_{i=1}^{n} P(x_i \mid eltern(X_i)),$$

weil jeder Sampling-Schritt nur von den Elternwerten abhängig ist. Dieser Ausdruck sollte Ihnen vertraut vorkommen, weil er auch die Wahrscheinlichkeit des Ereignisses gemäß der Repräsentation der gemeinsamen Verteilung des Bayesschen Netzes ist, wie in Gleichung (14.2) gezeigt. Wir haben also:

$$S_{PS}(x_1 \ldots x_n) = P(x_1 \ldots x_n).$$

Diese einfache Tatsache macht es ganz leicht, Fragen mithilfe von Stichproben zu beantworten.

```
function PRIOR-SAMPLE(bn) returns ein Ereignis, das aus der durch bn
         angegebenen A-priori-Wahrscheinlichkeit gesampelt wurde
  inputs: bn, ein Bayessches Netz, das die gemeisame Verteilung P(X1,..., Xn)
          angibt

  x ← ein Ereignis mit n Elementen
  foreach Variable Xi in X1, ..., Xn do
    x[i] ← eine zufällige Stichprobe aus P(Xi | parents(Xi))
  return x
```

Abbildung 14.13: Ein Sampling-Algorithmus, der Ereignisse aus einem Bayesschen Netz erzeugt.

Jeder Sampling-Algorithmus berechnet die Antworten, indem er die tatsächlich erzeugten Stichproben zählt. Angenommen, es gibt insgesamt N Stichproben und $N_{PS}(x_1, ..., x_n)$ ist die Anzahl, wie oft das spezifische Ereignis $x_1, ..., x_n$ in der Menge der Stichproben auftritt. Wir erwarten, dass diese Anzahl – als Bruchteil der Gesamtanzahl – im Grenzwert gemäß der folgenden Stichprobenwahrscheinlichkeit gegen den erwarteten Wert konvergiert:

$$\lim_{N\to\infty} \frac{N_{PS}(x_1,\ldots,x_n)}{N} = S_{PS}(x_1,\ldots,x_n) = P(x_1,\ldots,x_n). \tag{14.5}$$

Betrachten Sie beispielsweise das zuvor erzeugte Ereignis [*true*, *false*, *true*, *true*]. Die Sampling-Wahrscheinlichkeit für dieses Ereignis ist:

$$S_{PS}(true, false, true, true) = 0{,}5 \times 0{,}9 \times 0{,}8 \times 0{,}9 = 0{,}324$$

Im Grenzwert großer N erwarten wir, dass 32,4% aller Stichproben diesem Ereignis entsprechen.

Immer wenn wir eine ungefähre Gleichheit („≈“) verwenden, verstehen wir sie in genau diesem Sinne – dass die geschätzte Wahrscheinlichkeit im Grenzwert großer Stichproben genau wird. Eine solche Schätzung wird auch als **konsistent** bezeichnet. Man kann beispielsweise wie folgt eine konsistente Schätzung der Wahrscheinlichkeit eines bestimmten spezifizierten Ereignisses $x_1, ..., x_m$ für $m \leq n$ erzeugen:

$$P(x_1, ..., x_m) \approx N_{PS}(x_1, ..., x_m)/N. \tag{14.6}$$

Die Wahrscheinlichkeit des Ereignisses kann also als Bruchteil aller vollständigen Ereignisse geschätzt werden, die der Sampling-Prozess erzeugt und die mit dem partiell spezifizierten Ereignis übereinstimmen. Wenn wir beispielsweise 1000 Stichproben aus dem Sprinkler-Netz erzeugen und für 511 davon *Regen* = *true* gilt, ist die geschätzte Wahrscheinlichkeit von Regen, dargestellt als $\hat{P}(Regen = true)$ gleich 0,511.

Ablehnungs-Sampling in Bayesschen Netzen

Das **Ablehnungs-Sampling** ist eine allgemeine Methode, um Stichproben aus schwer zu sampelnden Verteilungen zu ziehen, wenn eine leicht zu sampelnde Verteilung bekannt ist. In seiner einfachsten Form kann es verwendet werden, um bedingte Wahrscheinlichkeiten zu berechnen, d.h. $P(X \mid \mathbf{e})$ zu ermitteln. Der Algorithmus REJECTION-SAMPLE ist in ▶ Abbildung 14.14 gezeigt. Zuerst erzeugt er Stichproben aus der A-priori-Verteilung, die durch das Netz gegeben ist. Anschließend lehnt er alle Stichproben ab, die nicht mit der Evidenz übereinstimmen. Schließlich wird die Schätzung $\hat{P}(X = x \mid \mathbf{e})$ ermittelt, indem gezählt wird, wie oft $X = x$ in den restlichen Stichproben vorkommt.

```
function REJECTION-SAMPLING(X, e, bn, N) returns eine Schätzung von P(X|e)
  inputs: X, die Abfragevariable
          e, beoachtete Werte für Variablen von E
          bn, ein Bayessches Netz
          N, die Gesamtanzahl der zu erzeugenden Stichproben
  local variables: N, ein Vektor von Zählergebnissen für jeden Wert von X,
                      anfangs null

  for j = 1 to N do
    x ← PRIOR-SAMPLE(bn)
    if x ist konsistent mit e then
      N[x] ← N[x]+1 wobei x der Wert von X in x ist
  return NORMALIZE(N)
```

Abbildung 14.14: Der Algorithmus des Ablehnungs-Samplings für die Beantwortung von Abfragen bei gegebener Evidenz in einem Bayesschen Netz.

Es sei $\hat{\mathbf{P}}(X \mid \mathbf{e})$ die geschätzte Verteilung, die der Algorithmus zurückgibt. Aus der Definition des Algorithmus haben wir:

$$\hat{\mathbf{P}}(X \mid \mathbf{e}) = \alpha \mathbf{N}_{PS}(X, \mathbf{e}) = \frac{\mathbf{N}_{PS}(X, \mathbf{e})}{N_{PS}(e)}.$$

Nach Gleichung (14.6) wird daraus:

$$\hat{\mathbf{P}}(X \mid \mathbf{e}) \approx \frac{\mathbf{P}(X, \mathbf{e})}{P(e)} = \mathbf{P}(X \mid \mathbf{e}).$$

Das bedeutet, Ablehnungs-Sampling erzeugt eine konsistente Schätzung der tatsächlichen Wahrscheinlichkeit.

Wir setzen unser Beispiel aus Abbildung 14.12(a) fort und nehmen an, wir wollen unter Verwendung von 100 Stichproben $\mathbf{P}(Regen \mid Sprinkler = true)$ abschätzen. Von den 100 erzeugten Stichproben sollen 73 *Sprinkler* = *false* haben und zurückgewiesen werden, während 27 *Sprinkler* = *true* haben; von den 27 haben 8 *Regen* = *true* und 19 *Regen* = *false*. Damit gilt:

$$\mathbf{P}(Regen \mid Sprinkler = true) \approx \text{NORMALIZE}(\langle 8; 19 \rangle) = \langle 0{,}296; 0{,}704 \rangle$$

Die tatsächliche Antwort lautet ⟨0,3; 0,7⟩. Je mehr Stichproben gesammelt werden, desto mehr konvergiert die Schätzung gegen die tatsächliche Antwort. Die Standardabweichung des Fehlers in jeder Wahrscheinlichkeit ist proportional zu $1/\sqrt{n}$, wobei n die Anzahl der in der Schätzung verwendeten Stichproben ist.

Das größte Problem beim Ablehnungs-Sampling ist, dass es so viele Stichproben ablehnt! Der Bruchteil der Stichproben, der konsistent mit der Evidenz **e** ist, fällt exponentiell, wenn die Anzahl der Evidenzvariablen fällt; deshalb ist diese Vorgehensweise für komplexe Probleme nicht sinnvoll.

Beachten Sie, dass das Ablehnungs-Sampling einer Schätzung bedingter Wahrscheinlichkeit direkt aus der realen Welt sehr ähnlich ist. Um beispielsweise $\mathbf{P}(Regen \mid Abendrot = true)$ zu schätzen, kann man einfach zählen, wie oft es regnet, nachdem am vorherigen Abend Abendrot beobachtet wurde – und die Abende ignorieren, an denen der Himmel nicht rot war. (Hier spielt die eigentliche Welt die Rolle des Algorithmus zur Stichprobenerzeugung). Offensichtlich kann das einige Zeit dauern, wenn es sehr selten Abendrot gibt – das ist die Schwäche des Ablehnungs-Samplings.

Wahrscheinlichkeitsgewichtung

Die **Wahrscheinlichkeitsgewichtung** vermeidet die Ineffizienz des Ablehnungs-Samplings, indem sie nur Ereignisse erzeugt, die konsistent zur Evidenz **e** sind. Es handelt sich um einen Spezialfall der allgemeinen statistischen Technik **Importance Sampling**, zugeschnitten auf Inferenz in Bayesschen Netzen. Wir beschreiben zunächst, wie der Algorithmus arbeitet; anschließend zeigen wir, dass er korrekt funktioniert – d.h. konsistente Wahrscheinlichkeitsschätzungen erzeugt.

```
function LIKELIHOOD-WEIGHTING(X, e, bn, N) returns eine Schätzung von P(X|e)
  inputs: X, die Abfragevariable
          e, beobachtete Werte für Variablen E
          bn, ein Bayessches Netz, das die gemeinsame Verteilung
              P(X1, ..., Xn) spezifiziert
          N, die Gesamtanzahl der zu generierenden Stichproben
  local variables: W, ein Vektor der gewichteten Zählwerte für jeden Wert
                      von X, anfangs null

  for j = 1 to N do
    x, w ← WEIGHTED-SAMPLE(bn, e)
    W[x ] ← W[x ] + w where x is the value of X in x
  return NORMALIZE(W)
```

```
function WEIGHTED-SAMPLE(bn, e) returns ein Ereignis und ein Gewicht

  w ← 1; x ← ein Ereignis mit n Elementen, die von e initialisiert werden
  foreach Variable Xi in X1, ..., Xn do
    if Xi ist eine Evidenzvariable mit dem Wert xi in e ist
      then w ← w × P(Xi = xi | parents(Xi))
      else x[i] ← eine Zufallsstrichprobe von P(Xi | parents(Xi))
  return x, w
```

Abbildung 14.15: Der Algorithmus zur Wahrscheinlichkeitsgewichtung für die Inferenz in Bayesschen Netzen. In WEIGHTED-SAMPLE wird jede Nichtevidenzvariable entsprechend der bedingten Verteilung gesampelt, wobei die bereits für die übergeordneten Elemente der Variablen gesampelten Werte gegeben sind, während ein Gewicht basierend auf der Wahrscheinlichkeit für jede Evidenzvariable akkumuliert wird.

LIKELIHOOD-WEIGHTING (siehe ▶ Abbildung 14.15) legt Werte für die Evidenzvariablen **E** fest und sampelt nur die Nichtevidenzvariablen. Damit ist garantiert, dass jedes erzeugte Ereignis konsistent mit der Evidenz ist. Es sind jedoch nicht alle Ereignisse gleich. Bevor die Zählwerte in der Verteilung für die Abfragevariable zusammengezählt werden, wird jedes Ereignis nach der *Wahrscheinlichkeit* gewichtet, mit der das Ereignis der Evidenz entspricht – gemessen als Produkt der bedingten Wahrscheinlichkeiten für jede Evidenzvariable bei bekannten Eltern. Intuitiv sollen Ereignisse, in denen die tatsächliche Evidenz unwahrscheinlich erscheint, weniger Gewicht erhalten.

Wir wollen den Algorithmus mit der Abfrage $\mathbf{P}(Regen \mid Wolkig = true, NassesGras = true)$ und der Reihenfolge *Wolkig*, *Sprinkler*, *Regen*, *NassesGras* auf das in Abbildung 14.12(a) gezeigte Netz anwenden (wobei jede beliebige topologische Reihenfolge funktioniert). Der Ablauf sieht folgendermaßen aus: Zuerst wird die Gewichtung w auf 1,0 gesetzt. Anschließend wird ein Ereignis erzeugt.

1 *Wolkig* ist eine Evidenzvariable mit dem Wert *true*. Demzufolge setzen wir

$$w \leftarrow w \times P(Wolkig = true) = 0{,}5.$$

2 *Sprinkler* ist keine Evidenzvariable, sodass die Stichprobe von $\mathbf{P}(Sprinkler \mid Wolkig = true) = \langle 0{,}1;\ 0{,}9\rangle$; dies soll *false* zurückgeben.

3 Ähnlich ist die Stichprobe aus $\mathbf{P}(Regen \mid Wolkig = true) = \langle 0{,}8;\ 0{,}2\rangle$; wir nehmen an, dies gibt *true* zurück.

4 *NassesGras* ist eine Evidenzvariable mit dem Wert *true*. Wir setzen also:

$$w \leftarrow w \times P(NassesGras = true \mid Sprinkler = false, Regen = true) = 0{,}45$$

Hier gibt WEIGHTED-SAMPLE das Ereignis [*true*, *false*, *true*, *true*] mit der Gewichtung 0,45 zurück, was als *Regen* = *true* gezählt wird.

Um zu verstehen, warum diese Wahrscheinlichkeitsgewichtung funktioniert, betrachten wir zuerst die Sampling-Verteilung S_{WS} für WEIGHTED-SAMPLE. Beachten Sie, dass die Evidenzvariablen **E** mit den Werten **e** feststehend sind. Wir bezeichnen die Nichtevidenzvariablen als **Z** (einschließlich der Abfragevariablen X). Der Algorithmus sampelt jede Variable in **Z** für bekannte Elternwerte:

$$S_{WS}(\mathbf{z}, e) = \prod_{i=1}^{l} P(z_i \mid eltern(Z_i)). \tag{14.7}$$

Beachten Sie, dass $Eltern(Z_i)$ sowohl Nichtevidenz- als auch Evidenzvariablen enthalten kann. Anders als die A-priori-Verteilung $P(\mathbf{z})$ berücksichtigt die Verteilung S_{WS} die Evidenz: Die gesampelten Werte für jedes Z_i werden durch die Evidenz für die Vorfahren von Z_i beeinflusst. Zum Beispiel achtet der Algorithmus auf die Evidenz *Wolkig* = *true* in der übergeordneten Variablen, wenn der Sprinkler sampelt. Dagegen legt S_{WS} weniger Wert auf die Evidenz als die echte A-posteriori-Verteilung $P(\mathbf{z} \mid \mathbf{e})$, weil die gesampelten Werte für jedes Z_i die Evidenz unter den Nichtvorfahren von Z_i *ignorieren*.[5] Zum Beispiel ignoriert der Algorithmus beim Sampling von *Sprinkler* und *Regen*

5 Im Idealfall würden wir eine Sampling-Verteilung gleich der wahren A-posteriori-Verteilung $P(\mathbf{z} \mid \mathbf{e})$ verwenden, um alle Evidenzen zu berücksichtigen. Das lässt sich aber nicht effizient bewerkstelligen. Wenn es möglich wäre, könnten wir die gewünschte Wahrscheinlichkeit mit beliebiger Genauigkeit mit einer polynomialen Anzahl an Stichproben abschätzen. Es kann gezeigt werden, dass es kein solches Annäherungsschema mit polynomialer Zeit geben kann.

die untergeordnete Variable *NassesGras* = *true*; das heißt, dass er viele Stichproben mit *Sprinkler* = *false* und *Regen* = *false* generiert, trotz der Tatsache, dass die Evidenz diesen Fall eigentlich ausschließt.

Die Wahrscheinlichkeitsgewichtung w macht die Differenz zwischen der tatsächlichen und der gewünschten Sampling-Verteilung wett. Die Gewichtung für eine bestimmte Stichprobe $\mathbf{x}$, die sich aus $\mathbf{z}$ und $\mathbf{e}$ zusammensetzt, ist das Produkt der Wahrscheinlichkeiten für jede Evidenzvariable mit bekannten Eltern (von denen alle oder einige unter den Z_i sein können):

$$w(\mathbf{z},\mathbf{e}) = \prod_{i=1}^{m} P(e_i \mid eltern(E_i)). \tag{14.8}$$

Wenn wir die Gleichungen (14.7) und (14.8) multiplizieren, erkennen wir, dass die *gewichtete* Wahrscheinlichkeit einer Stichprobe die folgende besonders praktische Form hat:

$$\begin{aligned} S_{WS}(\mathbf{z},\mathbf{e})w(\mathbf{z},\mathbf{e}) &= \prod_{i=1}^{l} P(z_i \mid eltern(Z_i)) \prod_{i=1}^{m} P(e_i \mid eltern(E_i)) \\ &= P(\mathbf{z},\mathbf{e}). \end{aligned} \tag{14.9}$$

Das liegt darin begründet, dass die beiden Produkte alle Variablen im Netz abdecken, sodass wir Gleichung (14.2) für die gemeinsame Wahrscheinlichkeit verwenden können.

Jetzt kann einfach gezeigt werden, dass die Schätzungen mit Wahrscheinlichkeitsgewichtung konsistent sind. Für einen bestimmten Wert x von X kann die geschätzte A-posteriori-Wahrscheinlichkeit wie folgt berechnet werden:

$$\begin{aligned} \hat{P}(x \mid \mathbf{e}) &= \alpha \sum_{y} N_{WS}(x,\mathbf{y},\mathbf{e}) w(x,\mathbf{y},\mathbf{e}) && \text{aus LIKELIHOOD-WEIGHTING} \\ &\approx \alpha' \sum_{y} S_{WS}(x,\mathbf{y},\mathbf{e}) w(x,\mathbf{y},\mathbf{e}) && \text{für große } N \\ &= \alpha' \sum_{y} P(x,\mathbf{y},\mathbf{e}) && \text{nach Gleichung (14.9)} \\ &= \alpha' P(x,\mathbf{e}) = P(x \mid \mathbf{e}). \end{aligned}$$

Damit ergibt die Wahrscheinlichkeitsgewichtung konsistente Schätzungen.

Weil die Wahrscheinlichkeitsgewichtung alle erzeugten Stichproben verwendet, kann sie sehr viel effizienter als das Ablehnungs-Sampling sein. Sie leidet jedoch unter einer Leistungsverschlechterung, wenn die Anzahl der Evidenzvariablen steigt. Weil die meisten Stichproben sehr geringe Gewichtungen haben, wird die gewichtete Schätzung von dem kleinen Bruchteil von Stichproben dominiert, die den Evidenzen mehr als eine verschwindend geringe Wahrscheinlichkeit zuordnen. Das Problem wird noch verschlimmert, wenn die Evidenzvariablen in der Variablenreihenfolge weit hinten stehen, weil dann die Nichtevidenzvariablen keine Evidenz in ihren Eltern und Vorfahren haben, um das Erzeugen der Stichproben zu steuern. Die Stichproben sind dann Simulationen, die wenig mit der Realität, wie sie die Evidenzen vorgeben, zu tun haben.

14.5.2 Inferenz durch Markov-Ketten-Simulation

MCMC-Algorithmen (**Markov Chain Monte Carlo**) arbeiten anders als Ablehnungs-Sampling und Wahrscheinlichkeitsgewichtung. Anstatt jede Stichprobe von Grund auf neu zu erzeugen, generieren MCMC-Algorithmen jede Stichprobe, indem sie die vorhergehende Stichprobe zufällig ändern. Es gibt also einen bestimmten *aktuellen Zustand*, der für jede Variable einen Wert spezifiziert, und der MCMC-Algorithmus generiert einen *nächsten Zustand*, indem er zufällige Änderungen am aktuellen Zustand vornimmt. (Dies erinnert an Simulated Annealing von *Kapitel 4* oder WALKSAT von *Kapitel 7*, da diese beiden Algorithmen zur MCMC-Familie gehören.) Hier beschreiben wir eine besondere Form von MCMC namens **Gibbs-Sampling**, die für Bayessche Netze besonders gut geeignet ist. (Andere Formen, die zum Teil erheblich leistungsfähiger sind, werden in den Hinweisen am Ende des Kapitels diskutiert.) Wir beschreiben zuerst, was der Algorithmus macht, und erläutern dann, warum er funktioniert.

Gibbs-Sampling in Bayesschen Netzen

Der Gibbs-Sampling-Algorithmus für Bayessche Netze beginnt mit einem willkürlichen Zustand (wobei die Evidenzvariablen auf ihre beobachteten Werte festgelegt werden) und generiert einen nächsten Zustand durch zufälliges Sampling eines Wertes für eine der Nichtevidenzvariablen X_i. Das Sampling für X_i geschieht *konditioniert auf den aktuellen Werten der Variablen in der Markov-Decke von* X_i. (In *Abschnitt 14.2.2* haben Sie erfahren, dass die Markov-Decke einer Variablen aus ihren Eltern, Kindern und den Eltern der Kinder besteht.) Der Algorithmus durchwandert deshalb zufällig den Zustandsraum – den Raum möglicher vollständiger Zuweisungen – und tauscht jeweils eine Variable aus, behält jedoch die Evidenzvariablen als feststehend bei.

Betrachten Sie die Abfrage $\mathbf{P}(Regen \mid Sprinkler = true, NassesGras = true)$ für das in Abbildung 14.12(a) gezeigte Netz. Die Evidenzvariablen *Sprinkler* und *NassesGras* sind auf ihre beobachteten Werte festgelegt und die Nichtevidenzvariablen *Wolkig* und *Regen* werden zufällig initialisiert – nehmen wir an, mit *true* bzw. *false*. Der Ausgangszustand ist also [*true*, *true*, *false*, *true*]. Jetzt werden die folgenden Schritte wiederholt ausgeführt:

1. *Wolkig* wird gesampelt mit bekannten aktuellen Werten seiner Markov-Decken-Variablen: In diesem Fall sampeln wir aus $\mathbf{P}(Wolkig \mid Sprinkler = true, Regen = false)$. (Wir werden gleich zeigen, wie diese Verteilung berechnet wird.) Angenommen, das Ergebnis ist $Wolkig = false$. Der neue aktuelle Zustand ist dann [*false*, *true*, *false*, *true*].

2. *Regen* wird gesampelt mit bekannten aktuellen Werten seiner Markov-Decken-Variablen: In diesem Fall sampeln wir aus $\mathbf{P}(Regen \mid Wolkig = false, Sprinkler = true, NassesGras = true)$. Angenommen, das Ergebnis ist $Regen = true$. Der neue aktuelle Zustand ist dann [*false*, *true*, *true*, *true*].

Jeder während dieses Prozesses besuchte Zustand ist eine Stichprobe, die zu der Schätzung für die Abfragevariable *Regen* beiträgt. Wenn der Prozess 20 Zustände besucht, in denen *Regen* wahr ist, und 60 Zustände, in denen *Regen* falsch ist, lautet die Antwort auf die Abfrage $\text{NORMALIZE}(\langle 20{,}60\rangle) = \langle 0{,}25;0{,}75\rangle$. ▶ Abbildung 14.16 zeigt den vollständigen Algorithmus.

```
function GIBBS-ASK(X, e, bn, N) returns eine Schätzung von P(X|e)
  local variables: N, ein Vektor von Zählwerten für jeden Wert von X, anfangs null
                   Z, die Nichtevidenzvariablen in bn
                   x, der aktuelle Zustand des Netzes, anfangs kopiert von e

  x mit zufälligen Werten für die Variablen in Z initialisieren
  for j = 1 to N do
    for each Z_i in Z do
      den Wert von Z_i in x durch Sampling von P(Z_i|mb(Z_i)) festlegen
      N[x] ← N[x] + 1 wobei x der Wert von X in x ist
  return NORMALIZE(N)
```

Abbildung 14.16: Der Gibbs-Sampling-Algorithmus für annähernde Inferenz in Bayesschen Netzen. Diese Version durchläuft zwar die Variablen, doch funktioniert der Algorithmus auch, wenn die Variablen zufällig ausgewählt werden.

Warum Gibbs-Sampling funktioniert

Tipp

Jetzt werden wir zeigen, dass Gibbs-Sampling konsistente Schätzungen für A-posteriori-Wahrscheinlichkeiten zurückgibt. Der Stoff in diesem Abschnitt ist recht technisch, aber die grundlegende Aussage ist einfach: *Der Sampling-Prozess führt zu einem „dynamischen Gleichgewicht", in dem der Zeitabschnitt, der in jedem Zustand verbracht wurde, auf lange Sicht genau proportional zu seiner A-posteriori-Wahrscheinlichkeit ist.* Diese bemerkenswerte Eigenschaft folgt aus der spezifischen **Übergangswahrscheinlichkeit**, mit der der Prozess von einem Zustand in einen anderen übergeht, wie durch die bedingte Verteilung bei bekannter Markov-Decke der gesampelten Variablen definiert.

Es sei $q(\mathbf{x} \rightarrow \mathbf{x}')$ die Wahrscheinlichkeit, dass der Prozess vom Zustand $\mathbf{x}$ in den Zustand $\mathbf{x}'$ übergeht. Diese Übergangswahrscheinlichkeit definiert eine sogenannte **Markov-Kette** im Zustandsraum. (Markov-Ketten werden auch in den *Kapiteln 15* und *17* noch häufig vorkommen.) Angenommen, wir führen die Markov-Kette für t Schritte aus und die Wahrscheinlichkeit, dass sich das System zum Zeitpunkt t im Zustand $\mathbf{x}$ befindet, sei $\pi_t(\mathbf{x})$. Außerdem soll $\pi_{t+1}(\mathbf{x}')$ die Wahrscheinlichkeit sein, dass sich das System zum Zeitpunkt $t+1$ im Zustand $\mathbf{x}'$ befindet. Für $\pi_t(\mathbf{x})$ können wir $\pi_{t+1}(\mathbf{x}')$ berechnen, indem wir für alle Zustände, in denen sich das System zum Zeitpunkt t befinden könnte, eine Summation für die Wahrscheinlichkeit, dass es sich in dem Zustand befindet, multipliziert mit der Wahrscheinlichkeit, dass es den Übergang nach $\mathbf{x}'$ vornimmt, durchführen:

$$\pi_{t+1}(x') = \sum_x \pi_t(\mathbf{x}) q(\mathbf{x} \rightarrow \mathbf{x}').$$

Wir sagen, dass die Kette ihre **stationäre Verteilung** erreicht hat, wenn $\pi_t = \pi_{t+1}$. Wir bezeichnen diese stationäre Verteilung als π; die definierende Gleichung sieht dann wie folgt aus:

$$\pi(\mathbf{x}') = \sum_x \pi(\mathbf{x}) q(\mathbf{x} \rightarrow \mathbf{x}') \qquad \textit{für alle } \mathbf{x}'. \tag{14.10}$$

Vorausgesetzt, dass die Übergangswahrscheinlichkeitsverteilung q **ergodisch** ist – d.h., dass jeder Zustand von jedem anderen Zustand erreichbar ist und keine strengen periodischen Zyklen existieren –, gibt es genau eine Verteilung π, die diese Gleichung für jedes gegebene q erfüllt.

Gleichung (14.10) sagt, dass der erwartete „Abfluss“ aus jedem Zustand (d.h. seine aktuelle „Population“) gleich dem erwarteten „Zufluss“ von allen Zuständen ist. Diese Beziehung ist offensichtlich dann erfüllt, wenn der erwartete Fluss zwischen zwei Zustandspaaren in beiden Richtungen gleich ist; das heißt

$$\pi(\mathbf{x})q(\mathbf{x} \rightarrow \mathbf{x}') = \pi(\mathbf{x}')q(\mathbf{x}' \rightarrow \mathbf{x}) \quad \text{für alle } \mathbf{x}, \mathbf{x}'. \tag{14.11}$$

Wenn diese Gleichung gilt, sagen wir, dass sich $q(x \rightarrow x')$ im **detaillierten Gleichgewicht** mit $\pi(\mathbf{x})$ befindet.

Wir können zeigen, dass das detaillierte Gleichgewicht die Stationarität impliziert, indem wir in Gleichung (14.11) über $\mathbf{x}$ summieren. Wir haben:

$$\sum_{\mathbf{x}} \pi(\mathbf{x})q(\mathbf{x} \rightarrow \mathbf{x}') = \sum_{\mathbf{x}} \pi(\mathbf{x}')q(\mathbf{x}' \rightarrow \mathbf{x}) = \pi(\mathbf{x}')\sum_{\mathbf{x}} q(\mathbf{x}' \rightarrow \mathbf{x}) = \pi(\mathbf{x}').$$

Dabei folgt der letzte Schritt, weil garantiert ein Übergang von $\mathbf{x}'$ stattfindet.

Die Übergangswahrscheinlichkeit $q(\mathbf{x} \rightarrow \mathbf{x}')$, die durch den Sampling-Schritt in GIBBS-ASK definiert ist, stellt tatsächlich einen Spezialfall der allgemeineren Definition des Gibbs-Samplings dar, wonach jede Variable konditioniert auf den aktuellen Werten *aller* anderen Variablen gesampelt wird. Wir zeigen zunächst, dass diese allgemeine Definition des Gibbs-Samplings die Gleichung für das detaillierte Gleichgewicht mit einer stationären Verteilung gleich $P(\mathbf{x}|\mathbf{e})$ erfüllt (der wahren A-posteriori-Verteilung auf Nichtevidenzvariablen). Dann beobachten wir einfach, dass – für Bayessche Netze – das Sampling bedingt auf allen Variablen dem bedingten Sampling auf der Markov-Decke der Variablen äquivalent ist (siehe *Abschnitt 14.2.2*).

Um den allgemeinen Gibbs-Sampler zu analysieren, der jedes X_i nacheinander mit einer Übergangswahrscheinlichkeit q_i sampelt, die alle anderen Variablen konditioniert, definieren wir $\overline{\mathbf{X}}_i$ als diese anderen Variablen (außer den Evidenzvariablen); ihre Werte im aktuellen Zustand sind $\overline{\mathbf{x}}_i$. Wenn wir einen neuen Wert x_i' für X_i bedingt auf allen anderen Variablen, einschließlich der Evidenz, sampeln, haben wir:

$$q(\mathbf{x} \rightarrow \mathbf{x}') = q((x_i, \overline{\mathbf{x}}_i)) \rightarrow (x_i', \overline{\mathbf{x}}_i)) = P(x_i' \mid \overline{\mathbf{x}}_i, \mathbf{e}).$$

Jetzt zeigen wir, dass die Übergangswahrscheinlichkeit für jeden Schritt des Gibbs-Samplers im detaillierten Gleichgewicht mit der wahren A-posteriori-Verteilung ist:

$$\pi(\mathbf{x})q_i(\mathbf{x} \rightarrow \mathbf{x}') = P(\mathbf{x} \mid \mathbf{e})\, P(x_i' \mid \overline{\mathbf{x}}_i, \mathbf{e}) = P(x_i, \overline{\mathbf{x}}_i \mid \mathbf{e})P(x_i' \mid \overline{\mathbf{x}}_i, \mathbf{e})$$

$$= P(x_i \mid \overline{\mathbf{x}}_i, \mathbf{e})P(\overline{\mathbf{x}}_i \mid \mathbf{e})P(x_i' \mid \overline{\mathbf{x}}_i, \mathbf{e}) \quad \text{(Verwendung der Kettenregel für den ersten Term)}$$

$$= P(x_i \mid \overline{\mathbf{x}}_i, \mathbf{e})P(x_i', \overline{\mathbf{x}}_i \mid \mathbf{e}) \quad \text{(Rückwärtsverwendung der Kettenregel)}$$

$$= \pi(\mathbf{x}')q_i(\mathbf{x}' \rightarrow \mathbf{x}).$$

Die Schleife

```
for each Zi in Z do
```

in Abbildung 14.16 lässt sich so deuten, dass eine große Übergangswahrscheinlichkeit q definiert wird, die die sequenzielle Zusammensetzung $q_1 \circ q_2 \circ \ldots \circ q_n$ der Übergangswahrscheinlichkeiten für die einzelnen Variablen ist. Es lässt sich leicht zeigen (Übung 14.18), dass die sequenzielle Zusammensetzung $q_i \circ q_j$ ebenfalls funktioniert,

wenn jedes der q_i und q_j die stationäre Verteilung π besitzt; folglich hat die Übergangswahrscheinlichkeit q für die gesamte Schleife eine stationäre Verteilung von $P(\mathbf{x} \mid \mathbf{e})$. Solange die BWTs Wahrscheinlichkeiten von 0 oder 1 enthalten – wodurch der Zustandsraum getrennt werden kann –, ist schließlich leicht zu sehen, dass q ergodisch ist. Somit werden die vom Gibbs-Sampling generierten Stichproben letztlich aus der wahren A-posteriori-Verteilung gezogen.

Schließlich muss noch gezeigt werden, wie der allgemeine Gibbs-Sampling-Schritt– das Sampling von X_i aus $\mathbf{P}(X_i \mid \overline{\mathbf{x}_i}, \mathbf{e})$ – in einem Bayesschen Netz auszuführen ist. Wie *Abschnitt 14.2.2* erläutert hat, ist eine Variable von allen anderen Variablen unabhängig, wenn ihre Markov-Decke bekannt ist; folglich ist

$$P(x_i' \mid \overline{\mathbf{x}}_i, \mathbf{e}) = P(x_i' \mid mb(X_i)),$$

wobei $mb(X_i)$ die Werte der Variablen $MB(X_i)$ in der Markov-Decke von X_i kennzeichnet. Wie in Übung 14.6 gezeigt, ist die Wahrscheinlichkeit einer Variablen mit bekannter Markov-Decke proportional zur Wahrscheinlichkeit der Variablen mit bekannten Eltern multipliziert mit der Wahrscheinlichkeit jedes Kindes mit jeweils bekannten Eltern:

$$P(x_i' \mid mb(X_i)) = \alpha P(x_i' \mid eltern(X_i)) \times \prod_{Y_j \in Kinder(X_i)} P(y_j \mid eltern(Y_j)). \qquad (14.12)$$

Um also jede Variable X_i bedingt von ihrer Markov-Decke auszutauschen, ist die Anzahl der erforderlichen Multiplikationen gleich der Anzahl der Kinder von X_i.

14.6 Relationale Wahrscheinlichkeitsmodelle und Modelle erster Stufe

Tipp

In *Kapitel 8* haben wir beschrieben, welche Vorteile die Logik erster Stufe gegenüber der Aussagenlogik im Hinblick auf die Repräsentation hat. Die Logik erster Stufe beschäftigt sich mit der Existenz von Objekten und den Relationen zwischen ihnen und kann Fakten über *einige* oder *alle* Objekte einer Domäne ausdrücken. Das führt häufig zu Repräsentationen, die sehr viel knapper als in äquivalenten aussagenlogischen Beschreibungen sind. Bayessche Netze sind im Wesentlichen aussagenlogisch: Die Menge der Zufallsvariablen ist feststehend und endlich, jede Variable hat eine feste Domäne möglicher Werte. Diese Tatsache beschränkt die Anwendbarkeit Bayesscher Netze. *Wenn wir eine Möglichkeit finden, die Wahrscheinlichkeitstheorie mit der Ausdruckskraft von Repräsentationen erster Stufe zu kombinieren, können wir den Bereich der verarbeitbaren Probleme wesentlich erweitern.*

Tipp

Nehmen wir zum Beispiel an, dass ein Online-Buchhändler Gesamtbewertungen von Produkten basierend auf Empfehlungen seiner Kunden bereitstellen möchte. Die Bewertung nimmt die Form einer A-posteriori-Verteilung über der Qualität des Buches an, wobei die verfügbare Evidenz gegeben ist. Die einfachste Lösung würde die Bewertung auf der durchschnittlichen Empfehlung aufbauen, vielleicht mit einer Varianz, die sich aus der Anzahl der Empfehlungen ergibt, doch wird dabei nicht die Tatsache berücksichtigt, dass manche Kunden freundlicher sind als andere und manche nicht ganz ehrlich urteilen. Manche Kunden geben gute Empfehlungen selbst für ziemlich niveaulose Bücher, während unehrliche Kunden sehr hohe bzw. sehr nied-

rige Bewertungen abgeben, ohne sich überhaupt auf die Qualität zu beziehen – wenn sie beispielsweise für einen Verleger arbeiten.[6]

Für einen einzelnen Kunden C_1, der das einzelne Buch B_1 empfiehlt, könnte das Bayes-Netz wie in ▶ Abbildung 14.17(a) aussehen. (Genau wie in *Abschnitt 9.1* sind Ausdrücke mit Klammern wie zum Beispiel $Ehrlich(C_1)$ lediglich Fantasiesymbole – in diesem Fall Fantasienamen für Zufallsvariablen.) Mit zwei Kunden und zwei Büchern sieht das Bayes-Netz wie in ▶ Abbildung 14.17(b) aus. Für größere Anzahlen von Büchern und Kunden wird es vollkommen unpraktisch, das Netz manuell zu spezifizieren.

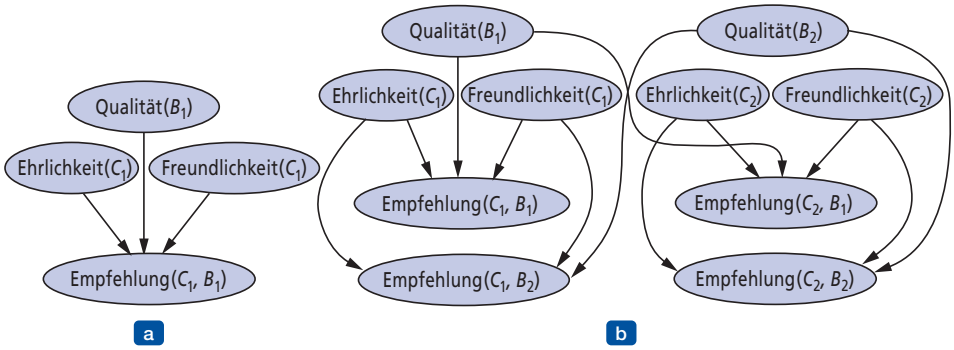


Abbildung 14.17: (a) Bayes-Netz für einen einzelnen Kunden C_1, der ein einzelnes Buch B_1 empfiehlt. $Ehrlich(C_1)$ ist boolesch, während die anderen Variablen ganze Zahlen von 1 bis 5 enthalten. (b) Bayes-Netz mit zwei Kunden und zwei Büchern.

Es kommt uns entgegen, dass das Netz jede Menge wiederholter Strukturen besitzt. Jede Variable *Empfehlung*(c, b) hat die Variablen *Ehrlich*(c), *Freundlichkeit*(c) und *Qualität*(b) als übergeordnete Variablen. Darüber hinaus sind die BWTs für alle *Empfehlung*(c, b)-Variablen identisch. Das Gleiche gilt für alle *Ehrlich*(c)-Variablen usw. Die Situation scheint auf eine Sprache erster Stufe zugeschnitten zu sein. Wir möchten etwas wie

$$Empfehlung(c, b) \sim EmpBWT(Ehrlich(c), Freundlichkeit(c), Qualität(b))$$

ausdrücken mit der beabsichtigten Bedeutung, dass die Empfehlung eines Kunden für ein Buch von der Ehrlichkeit und der Freundlichkeit des Kunden sowie der Qualität des Buches entsprechend einer bestimmten BWT abhängig ist. Dieser Abschnitt entwickelt eine Sprache, mit der wir genau dies – und darüber hinaus noch einiges mehr – sagen können.

14.6.1 Mögliche Welten

Wie *Kapitel 13* erläutert hat, definiert ein Wahrscheinlichkeitsmodell eine Menge Ω möglicher Welten mit einer Wahrscheinlichkeit $P(\omega)$ für jede Welt ω. Für Bayessche Netze sind die möglichen Welten Zuweisungen von Werten zu Variablen; insbesondere sind für den booleschen Fall die möglichen Welten identisch mit denen der Aussagenlogik. Für ein Wahrscheinlichkeitsmodell der ersten Stufe scheint es dann so zu

6 Ein Spieltheoretiker würde einem unehrlichen Kunden raten, die Erkennung zu vermeiden, indem er gelegentlich ein gutes Buch von einem Mitbewerber empfiehlt. Siehe *Kapitel 17*.

sein, dass die möglichen Welten diejenigen der Logik erster Stufe sein müssen – das heißt eine Menge von Objekten mit Beziehungen zwischen ihnen und einer Interpretation, die Konstantensymbole zu Objekten, Prädikatsymbole zu Beziehungen und Funktionssymbole zu Funktionen auf diesen Objekten zuordnet (siehe *Abschnitt 8.2*). Das Modell muss außerdem eine Wahrscheinlichkeit für jede derartige mögliche Welt definieren, genau wie ein Bayessches Netz eine Wahrscheinlichkeit für jede Zuweisung von Werten an Variablen definiert.

Nehmen wir für den Moment an, wir hätten herausgefunden, wie dies zu bewerkstelligen ist. Dann können wir wie üblich (siehe *Abschnitt 13.2.1*) die Wahrscheinlichkeit ϕ jedes logischen Satzes erster Stufe als Summe über die möglichen Welten erhalten, wo sie wahr ist:

$$P(\phi) = \sum_{\omega:\phi \text{ ist wahr in } \omega} P(\omega). \tag{14.13}$$

Bedingte Wahrscheinlichkeiten $P(\phi \mid \mathbf{e})$ lassen sich ähnlich erhalten, sodass wir unserem Modell im Prinzip jede Frage stellen können – z.B. „Welche Bücher werden von unehrlichen Kunden höchstwahrscheinlich am stärksten empfohlen?“ – und eine Antwort erhalten. So weit, so gut.

Es gibt jedoch ein Problem: Die Menge der Modelle erster Stufe ist unendlich. Wir haben dies explizit in Abbildung 8.4 gesehen, die wir noch einmal in ▶ Abbildung 14.18 (oben) zeigen. Das bedeutet, dass (1) die Summation in Gleichung (14.13) unmöglich sein und (2) die Spezifikation einer vollständigen, konsistenten Verteilung über einer unendlichen Menge von Welten sich sehr schwierig gestalten könnte.

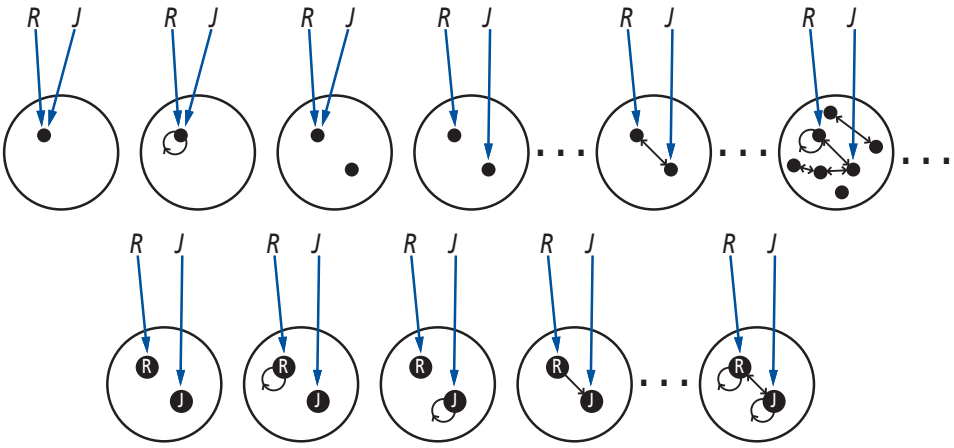


Abbildung 14.18: Oben: Einige Elemente der Menge aller möglichen Welten für eine Sprache mit zwei Konstantensymbolen, *R* und *J*, und einem binären Relationssymbol unter der Standardsemantik für Logik erster Stufe. Unten: Die möglichen Welten unter Datenbanksemantik. Die Interpretation der Konstantensymbole ist feststehend und für jedes Konstantensymbol gibt es ein anderes Objekt.

Abschnitt 14.6.2 untersucht einen Ansatz, um dieses Problem anzugehen. Die Idee besteht darin, keine Anleihen bei der Standardsemantik der Logik erster Stufe aufzunehmen, sondern auf die in *Abschnitt 8.2.8* definierte **Datenbanksemantik** zurückzugreifen. Die Datenbanksemantik trifft die **Annahme eindeutiger Namen** – hier übernehmen wir sie für die Konstantensymbole. Außerdem nimmt sie die **Domänenabgeschlossenheit** an

– außer den benannten Objekten gibt es keine weiteren Objekte. Wir können dann eine endliche Menge von möglichen Welten garantieren, indem wir die Menge von Objekten in jeder Welt genau gleich der Menge der verwendeten Konstantensymbole machen; wie Abbildung 14.18 (unten) zeigt, gibt es keine Unbestimmtheit über die Zuordnung von Symbolen zu Objekten oder über die Objekte, die vorhanden sind. Derart definierte Modelle bezeichnen wir als **relationale Wahrscheinlichkeitsmodelle** (RWMs).[7] Die RWM-Semantik unterscheidet sich von der in *Abschnitt 8.2.8* eingeführten Datenbanksemantik in erster Linie darin, dass RWMs keine Annahme der Weltabgeschlossenheit treffen – offenbar ist es in einem probabilistischen Schlusssystem nicht sinnvoll anzunehmen, dass jeder unbekannte Fakt falsch ist!

Wenn die zugrunde liegenden Annahmen der Datenbanksemantik nicht zutreffen, funktionieren RWMs nicht ordnungsgemäß. Zum Beispiel könnte ein Buchhändler eine ISBN (International Standard Book Number) als Konstantensymbol verwenden, um jedes Buch zu benennen, selbst wenn ein bestimmtes „logisches“ Buch (z.B. „Vom Winde verweht“) mehrere ISBNs haben kann. Es wäre sinnvoll, Empfehlungen über mehrere ISBNs zusammenzufassen, doch ist sich der Buchhändler möglicherweise nicht ganz sicher, welche ISBNs tatsächlich zum gleichen Buch gehören. (Wir reifizieren keine einzelnen Exemplare des Buches, was bei antiquarischen Verkäufen, im Autohandel usw. notwendig sein könnte.) Erschwerend kommt hinzu, dass zwar jeder Kunde durch eine Anmelde-ID identifiziert wird, unehrliche Kunden aber Tausende solcher IDs haben können! Auf dem Gebiet der Computersicherheit bezeichnet man diese Mehrfach-IDs als **Sybils** und ihre Verwendung, um ein Reputationssystem durcheinanderzubringen, als **Sybil-Angriff**. Somit hat selbst eine einfache Anwendung in einer relativ gut definierten Online-Domäne sowohl mit **Existenzunsicherheit** (welches sind die wirklichen Bücher und Kunden, die hinter den beobachteten Daten stehen) als auch **Identitätsunsicherheit** (welches Symbol verweist tatsächlich auf dasselbe Objekt) zu tun. Wir müssen in den sauren Apfel beißen und Wahrscheinlichkeitsmodelle basierend auf der Standardsemantik der Logik erster Stufe definieren, für die die möglichen Welten in den Objekten, die sie enthalten, und in den Zuordnungen von Symbolen zu Objekten variieren können. *Abschnitt 14.6.3* zeigt, wie das zu realisieren ist.

14.6.2 Relationale Wahrscheinlichkeitsmodelle

Wie Logik erster Stufe verfügen relationale Wahrscheinlichkeitsmodelle (RWMs) über Konstanten-, Funktions- und Prädikatssymbole. (Es ist einfacher, Prädikate als Funktionen anzusehen, die *true* oder *false* zurückgeben.) Außerdem nehmen wir für jede Funktion eine **Typsignatur** an, die die Typen der einzelnen Argumente und den Rückgabewert der Funktion spezifiziert. Wenn der Typ jedes Objektes bekannt ist, lassen sich viele unechte mögliche Welten durch diesen Mechanismus eliminieren. Für die Buchempfehlungsdomäne lauten die Typen *Kunde* und *Buch* und die Typsignaturen für die Funktionen und Prädikate sehen folgendermaßen aus:

$$\begin{aligned}
&\mathit{Ehrlich}: \mathit{Kunde} \rightarrow \{\mathit{true}, \mathit{false}\}\\
&\mathit{Freundlichkeit}: \mathit{Kunde} \rightarrow \{1, 2, 3, 4, 5\}\\
&\mathit{Qualität}: \mathit{Buch} \rightarrow \{1, 2, 3, 4, 5\}\\
&\mathit{Empfehlung}: \mathit{Kunde} \times \mathit{Buch} \rightarrow \{1, 2, 3, 4, 5\}.
\end{aligned}$$

7 Der Name **relationales Wahrscheinlichkeitsmodell** wurde von Pfeffer (2000) für eine etwas andere Darstellung verwendet, doch sind die grundlegenden Ideen die gleichen.

Die Konstantensymbole sind die Kunden- und Buchnamen, die im Datensatz des Händlers erscheinen. Im weiter vorn in Abbildung 14.17(b) angegebenen Beispiel lauten sie C_1, C_2 und B_1, B_2.

Mit den bekannten Konstanten und ihren Typen sowie den Funktionen und ihren Typsignaturen erhält man die Zufallsvariablen des RWM, indem jede Funktion mit jeder möglichen Kombination von Objekten instanziiert wird: *Ehrlich*(C_1), *Qualität*(B_2), *Empfehlung*(C_1, B_2) usw. Dies sind genau die Variablen, die in Abbildung 14.17(b) erscheinen. Da jeder Typ nur endlich viele Instanzen besitzt, ist die Anzahl der grundlegenden Zufallsvariablen ebenfalls endlich.

Um das RWM zu vervollständigen, müssen wir die Abhängigkeiten formulieren, die diese Zufallsvariablen regieren. Für jede Funktion gibt es genau eine Abhängigkeit, wobei jedes Argument der Funktion eine logische Variable ist (d.h. eine Variable, die sich wie in Logik erster Stufe über Objekte erstreckt):

$$\begin{aligned}
&\mathit{Ehrlich}(c) \sim (0{,}99;\ 0{,}01)\\
&\mathit{Freundlichkeit}(c) \sim (0{,}1;\ 0{,}1;\ 0{,}2;\ 0{,}3;\ 0{,}3)\\
&\mathit{Qualität}(b) \sim (0{,}05;\ 0{,}2;\ 0{,}4;\ 0{,}2;\ 0{,}15)\\
&\mathit{Empfehlung}(c, b) \sim \mathit{EmpBWT}\,(\mathit{Ehrlich}(c), \mathit{Freundlichkeit}\,(c), \mathit{Qualität}\,(b)).
\end{aligned}$$

Hierbei ist *EmpBWT* eine separat definierte bedingte Verteilung mit $2 \times 5 \times 5 = 50$ Zeilen mit je 5 Einträgen. Die Semantik des RWM lässt sich erhalten, indem diese Abhängigkeiten für alle bekannten Konstanten instanziiert werden, was ein Bayessches Netz (wie in Abbildung 14.17(b)) ergibt, das eine gemeinsame Verteilung über den Zufallsvariablen des RWM definiert.[8]

Wir können das Modell verfeinern, indem wir eine **kontextspezifische Unabhängigkeit** einführen, um die Tatsache widerzuspiegeln, dass unehrliche Kunden die Qualität gar nicht beachten, wenn sie eine Empfehlung abgeben; darüber hinaus spielt die Freundlichkeit in ihren Entscheidungen keine Rolle. Eine kontextspezifische Unabhängigkeit ermöglicht es, dass eine Variable unabhängig von ihren Eltern ist, wenn bestimmte Werte von anderen gegeben sind; somit ist *Empfehlung*(c, b) unabhängig von *Freundlichkeit*(c) und *Qualität*(b), wenn *Ehrlich*(c) = *false* ist:

$$\begin{aligned}
\mathit{Empfehlung}(c, b) \sim\ &\textbf{if}\ \mathit{Ehrlich}(c)\ \textbf{then}\\
&\qquad \mathit{EhrlichEmpBWT}(\mathit{Freundlichkeit}(c), \mathit{Qualität}\,(b))\\
&\textbf{else}\ (0{,}4;\ 0{,}1;\ 0{,}0;\ 0{,}1;\ 0{,}4).
\end{aligned}$$

Eine derartige Abhängigkeit sieht zwar wie eine gewöhnliche *if-then-else*-Anweisung in einer Programmiersprache aus, doch gibt es einen wesentlichen Unterschied: Das Inferenzmodul *muss den Wert des Bedingungstestes nicht unbedingt kennen*!

Dieses Modell können wir auf unzählige Arten vervollkommnen, um es realistischer zu machen. Nehmen wir zum Beispiel an, dass ein ehrlicher Kunde, der Fan des Autors eines Buches ist, dem Buch die Bewertung 5 unabhängig von der Qualität gibt:

8 Um zu garantieren, dass das RWM eine ordnungsgemäße Verteilung definiert, sind einige technische Bedingungen einzuhalten. Erstens müssen die Abhängigkeiten azyklisch sein, da das Bayessche Netz andernfalls Zyklen enthält und keine geeignete Verteilung definiert. Zweitens müssen die Abhängigkeiten fundiert sein, d.h., es darf keine unendlichen Vorgängerketten geben, wie sie beispielsweise aus rekursiven Abhängigkeiten herrühren. Unter bestimmten Umständen (siehe Übung 14.5) liefert eine Festkommaberechnung ein fundiertes Wahrscheinlichkeitsmodell für ein rekursives RWM.

Empfehlung $(c, b) \sim$ **if** *Ehrlich*(c) **then**
if *Fan*$(c,$ *Autor* $(b))$ **then** *Genau*(5)
else *EhrlichEmpBWT*(*Freundlichkeit*(c), *Qualität* (b))
else 0,4; 0,1; 0,0; 0,1; 0,4)

Auch hier ist der Bedingungstest *Fan*$(c,$ *Autor* $(b))$ unbekannt, doch wenn ein Kunde nur 5er-Bewertungen für die Bücher eines bestimmten Autors abgibt und anderweitig nicht durch Freundlichkeit glänzt, ist die A-posteriori-Wahrscheinlichkeit recht hoch, dass der Kunde ein Fan dieses Autors ist. Darüber hinaus neigt die A-posteriori-Verteilung dazu, die 5er-Bewertungen des Kunden bei der Qualitätseinschätzung von Büchern dieses Autors nicht voll anzuerkennen.

Im obigen Beispiel haben wir implizit angenommen, dass der Wert von *Autor*(b) für jedes b bekannt ist, was aber nicht immer der Fall ist. Wie kann das System nun beispielsweise schließen, ob C_1 ein Fan von *Autor*(B_2) ist, wenn *Autor*(B_2) unbekannt ist? Die Antwort ist, dass das System gegebenenfalls über alle möglichen Autoren schließen muss. Nehmen wir an (um die Dinge einfach zu halten), dass es lediglich zwei Autoren A_1 und A_2 gibt. Dann ist *Autor*(B_2) eine Zufallsvariable mit zwei möglichen Werten, A_1 und A_2, und eine übergeordnete Variable von *Empfehlung*(C_1, B_2). Die Variablen *Fan*(C_1, A_1) und *Fan*(C_1, A_2) sind ebenfalls übergeordnete Variablen. Die bedingte Verteilung für *Empfehlung*(C_1, B_2) ist dann praktisch ein **Multiplexer**, in dem die übergeordnete Variable *Autor*(B_2) als Selektor fungiert, der auswählt, welche der Variablen *Fan*(C_1, A_1) und *Fan*(C_1, A_2) tatsächlich in die Empfehlung eingeht. ▶ Abbildung 14.19 zeigt ein Fragment des äquivalenten Bayes-Netzes. Unbestimmtheit im Wert von *Autor*(B_2), die die Abhängigkeitsstruktur des Netzes beeinflusst, ist eine Instanz **relationaler Unbestimmtheit**.

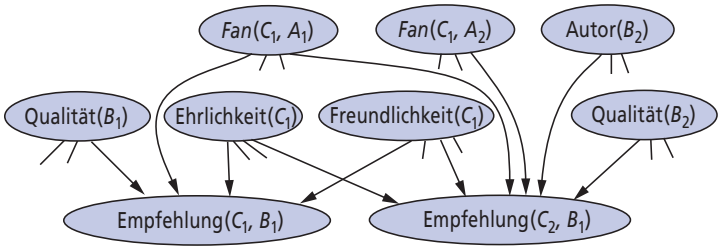


Abbildung 14.19: Fragment des äquivalenten Bayes-Netzes, wenn *Autor*(B_2) unbekannt ist.

Falls Sie sich fragen, wie das System wohl herausfinden kann, wer der Autor von B_2 ist: Betrachten Sie die Wahrscheinlichkeit, dass drei andere Kunden Fans von A_1 sind (und keine anderen Lieblingsautoren gemeinsam haben) und B_2 von allen drei Kunden eine 5 bekommen hat, selbst wenn die meisten anderen Kunden es ziemlich trist finden. In diesem Fall ist es äußerst wahrscheinlich, dass A_1 der Autor von B_2 ist.

Das Aufkommen von anspruchsvollem Schließen, wie zum Beispiel dem aus einem RWM-Modell mit nur wenigen Zeilen, ist ein hervorragendes Beispiel, wie sich probabilistische Einflüsse durch das Geflecht von Zwischenverbindungen unter Objekten im Modell ausbreiten. Oftmals wird das Bild, das die A-posteriori-Verteilung vermittelt, umso deutlicher, je mehr Abhängigkeiten und Objekte hinzugefügt werden.

Als Nächstes stellt sich die Frage, wie Inferenz in RWMs zu realisieren ist. Ein Ansatz ist es, die Evidenz, die Abfrage und die darin enthaltenen Konstantensymbole zu sammeln, das äquivalente Bayes-Netz zu konstruieren und eine der in diesem Kapitel behandelten Inferenzmethoden anzuwenden. Diese Technik bezeichnet man als **Aufrollen**. Dieses Vorgehen hat offensichtlich den Nachteil, dass das resultierende Bayes-Netz sehr groß sein kann. Und wenn es viele Kandidatenobjekte für eine unbekannte Relation oder Funktion gibt – zum Beispiel den unbekannten Autor von B_2 –, haben einige Variablen im Netz möglicherweise viele Eltern.

Erfreulicherweise lässt sich viel tun, um generische Inferenzalgorithmen zu verbessern. Erstens bedeutet die Anwesenheit von wiederholten Substrukturen im aufgerollten Bayes-Netz, dass viele Faktoren, die während der Variableneliminierung entstehen (und ähnliche Arten von Tabellen, die durch Clustering-Algorithmen konstruiert werden), identisch sind; durch effektive Caching-Schemas ließen sich bei großen Netzen Beschleunigungen um drei Größenordnungen erreichen. Zweitens finden Inferenzmethoden, die entwickelt wurden, um von kontextspezifischer Unabhängigkeit in Bayes-Netzen zu profitieren, viele Anwendungen in RWMs. Drittens besitzen MCMC-Inferenzalgorithmen viele interessante Eigenschaften, wenn man sie auf RWMs mit relationaler Unbestimmtheit anwendet. MCMC arbeitet durch Sampling vollständiger möglicher Welten, sodass in jedem Zustand die relationale Struktur komplett bekannt ist. Im weiter vorn angegebenen Beispiel würde jeder MCMC-Zustand den Wert von $Autor(B_2)$ spezifizieren und somit sind die anderen möglichen Autoren nicht mehr Eltern der Empfehlungsknoten für B_2. Für MCMC nimmt die Netzkomplexität aufgrund relationaler Unbestimmtheit nicht zu; stattdessen umfasst der MCMC-Prozess Übergänge, die die relationale Struktur – und folglich die Abhängigkeitsstruktur – des aufgerollten Netzes ändern.

Alle eben beschriebenen Methoden gehen davon aus, dass das BWM partiell oder vollständig in ein Bayessches Netz aufgerollt sein muss. Das ist genau analog zum **Überführen in Aussagenlogik** für logische Inferenz erster Stufe (siehe *Abschnitt 9.1*). Resolutionstheorem-Beweiser und Logikprogrammiersysteme vermeiden das Überführen in Aussagenlogik, indem sie die logischen Variablen nur bei Bedarf instanziieren, um die Inferenz zu ermöglichen; das heißt, sie *heben* den Inferenzprozess über die Stufe grundlegender aussagenlogischer Sätze und lassen jeden angehobenen Schritt die Arbeit vieler Grundschritte ausführen. Das gleiche Konzept wird in probabilistischer Inferenz angewendet. Zum Beispiel kann im Algorithmus zur Variableneliminierung ein angehobener Faktor eine vollständige Menge von Grundfaktoren darstellen, die Wahrscheinlichkeiten zu Zufallsvariablen im RWM zuordnen, wobei sich diese Zufallsvariablen nur in den zu ihrer Konstruktion verwendeten Konstantensymbolen unterscheiden. Es ginge über den Rahmen dieses Buches hinaus, die Einzelheiten dieser Methode darzustellen, Verweise finden Sie aber am Ende des Kapitels.

14.6.3 Wahrscheinlichkeitsmodelle im offenen Universum

Wie weiter vorn erwähnt, ist Datenbanksemantik für Situationen geeignet, in denen die Menge der existierenden relevanten Objekte genau bekannt ist und sich die Objekte eindeutig identifizieren lassen. (Insbesondere sind alle Beobachtungen über ein Objekt korrekt mit dem Konstantensymbol verbunden, das das Objekt benennt.) Allerdings sind diese Annahmen in vielen realen Welten einfach unhaltbar. Wir haben die Beispiele für mehrere ISBNs und Sybil-Angriffe in der Buchempfehlungsdomäne angegeben (zu der wir gleich zurückkommen), doch ist das Phänomen weitgreifender:

- Ein Bildverarbeitungssystem weiß nicht, was es an der nächsten Ecke erwartet (falls dort überhaupt etwas ist), und vielleicht auch nicht, ob es sich bei dem Objekt, das es sieht, um dasselbe handelt, das es vor wenigen Minuten gesehen hat.
- Ein Textverarbeitungssystem kennt die Entitäten, um die es in einem Text geht, im Voraus nicht und muss schließen, ob sich Ausdrücke wie zum Beispiel „Marie“, „Dr. Schmidt“, „sie“, „sein Herzspezialist“, „seine Mutter“ usw. auf dasselbe Objekt beziehen.
- Der Analytiker eines Nachrichtendienstes, der auf der Jagd nach Spionen ist, weiß nie, wie viele Spione es wirklich gibt, und kann nur raten, ob verschiedene Pseudonyme, Telefonnummern und Sichtungen zum selben Individuum gehören.

In der Tat scheint es der überwiegende Teil der menschlichen Wahrnehmung zu verlangen, zu lernen, welche Objekte existieren, und in der Lage zu sein, Beobachtungen – denen praktisch nie eindeutige IDs zuzuordnen sind – mit hypothetischen Objekten in der Welt zu verbinden.

Deshalb müssen wir in der Lage sein, sogenannte Wahrscheinlichkeitsmodelle des **offenen Universums** (Open-Universe Probability Models, OUPMs) basierend auf der Standardsemantik von Logik erster Stufe zu schreiben, wie es Abbildung 14.18 oben zeigt. Mit einer Sprache für OUPMs lassen sich derartige Modelle leicht formulieren, wobei eine eindeutige, konsistente Wahrscheinlichkeitsverteilung über einem unendlichen Raum möglicher Welten garantiert wird.

Zunächst einmal muss man verstehen, wie gewöhnliche Bayessche Netze und RWMs es schaffen, ein eindeutiges Wahrscheinlichkeitsmodell zu definieren und diese Einsicht auf die Einrichtung erster Stufe zu übertragen. Praktisch generiert ein Bayessches Netz jede mögliche Welt – Ereignis für Ereignis – in der durch die Netzstruktur definierten topologischen Reihenfolge, wobei jedes Ereignis eine Zuweisung eines Wertes an eine Variable ist.

Ein RWM erweitert dies auf komplette Ereignismengen, die durch die möglichen Instanziierungen der logischen Variablen in einem bestimmten Prädikat oder einer bestimmten Funktion definiert sind. OUPMs erlauben darüber hinaus erzeugende Schritte, die den möglichen zu konstruierenden Welten *Objekte hinzufügen*, wobei die Anzahl und der Typ der Objekte von den bereits in dieser Welt befindlichen Objekten abhängen können. Das zu generierende Ereignis ist also nicht die Zuweisung eines Wertes zu einer Variablen, sondern die wahre Existenz von Objekten.

In OUPMs lässt sich dies zum Beispiel mit zusätzlichen Anweisungen bewerkstelligen, die bedingte Verteilungen über der Anzahl von Objekten verschiedener Arten definieren. Zum Beispiel könnten wir in der Buchempfehlungsdomäne zwischen *Kunden* (echten Menschen) und ihren *Login-IDs* unterscheiden wollen. Angenommen, wir erwarten zwischen 100 und 10.000 verschiedene Kunden (die wir nicht direkt beobachten können). Dies lässt sich als A-priori-Lognormalverteilung[9] wie folgt ausdrücken:

$$\#\ Kunde \sim LogNormal\,[6{,}9;\ 2{,}3^2]().$$

Von ehrlichen Kunden erwarten wir, dass sie nur eine einzige ID besitzen, während unehrliche Kunden zwischen 10 und 1000 IDs haben dürften:

$$\#\ LoginID\,(Besitzer = c) \sim \textbf{if}\ Ehrlich(c)\ \textbf{then}\ Genau(1)\ \textbf{else}\ LogNormal\,[6{,}9;\ 2{,}3^2]().$$

9 Eine Verteilung $LogNormal[\mu, \sigma^2](x)$ ist einer Verteilung $N[\mu, \sigma^2](x)$ über $\log_e(x)$ äquivalent.

Diese Anweisung definiert die Anzahl der Login-IDs für einen bestimmten Besitzer, der ein Kunde ist. Die Funktion *Besitzer* ist eine sogenannte **Ursprungsfunktion**, weil sie aussagt, woher jedes generierte Objekt stammt. In der formalen Semantik von BLOG (im Unterschied zur Logik erster Stufe) sind die Domänenelemente in jeder möglichen Welt tatsächlich Generationsverläufe (z.B. „die vierte Login-ID des siebenten Kunden“) und keine einfachen Token.

Entsprechend den technischen Bedingungen von Azyklizität und fundierter Ordnung ähnlich wie bei RWMs, definieren derartige Modelle des offenen Universums eine einheitliche Verteilung über mögliche Welten. Darüber hinaus existieren Inferenzalgorithmen derart, dass sich für jedes in dieser Form wohldefinierte Modell und jede Abfrage erster Stufe die zurückgegebene Antwort der wahren A-posteriori-Verteilung im Grenzwert beliebig nähert. Allerdings ist es nicht ganz trivial, diese Algorithmen zu entwerfen. Zum Beispiel kann ein MCMC-Algorithmus im Raum der möglichen Welten nicht direkt sampeln, wenn die Größe dieser Welten unbeschränkt ist; stattdessen sampelt er endliche, partielle Welten und stützt sich dabei auf die Tatsache, dass nur endlich viele Objekte in verschiedener Hinsicht für die Abfrage relevant sein können. Darüber hinaus müssen es Übergänge ermöglichen, zwei Objekte zu einem zusammenzuführen oder ein Objekt auf zwei aufzuteilen. (Details hierzu finden Sie in den Hinweisen am Ende des Kapitels.) Trotz dieser Schwierigkeiten gilt immer noch das in Gleichung (14.13) aufgestellte Grundprinzip: Die Wahrscheinlichkeit jedes Satzes ist genau definiert und kann berechnet werden.

Die Forschung in diesem Bereich befindet sich noch in einem frühen Stadium, aber es wird bereits deutlich, dass das probabilistische Schließen erster Stufe eine wesentlich verbesserte Effektivität für KI-Systeme erzielt, was den Umgang mit unsicheren Informationen betrifft. Außer den oben erwähnten Anwendungen – Bildverarbeitung, Texterkennung und Nachrichtendienste – sind auch viele andere Arten sensorischer Interpretation realisierbar.

14.7 Weitere Ansätze zum unsicheren Schließen

Andere Wissenschaften (z.B. Physik, Genetik und Wirtschaftswissenschaften) haben die Wahrscheinlichkeit lange als Modell für die Unsicherheit favorisiert. Pierre Laplace stellte 1890 fest: „Die Wahrscheinlichkeitstheorie ist nichts weiter als der gesunde Menschenverstand, reduziert auf Berechnungen.“ Und 1850 sagte James Maxwell: „Die wahre Logik dieser Welt ist der Wahrscheinlichkeitskalkül, der die Größe der Wahrscheinlichkeit berücksichtigt, die im Denken eines vernünftigen Menschen enthalten ist oder enthalten sein sollte.“

Angesichts dieser langen Tradition ist es vielleicht überraschend, dass die KI viele Alternativen zur Wahrscheinlichkeit in Betracht gezogen hat. Die frühesten Expertensysteme der 1970er Jahre ignorierten die Unsicherheit und verwendeten ein streng logisches Schließen, aber es wurde schnell deutlich, dass dies für die meisten Domänen der realen Welt unpraktisch war. Die nächste Generation der Expertensysteme (insbesondere in medizinischen Domänen) verwendete probabilistische Techniken. Die ersten Ergebnisse waren vielversprechend, aber sie konnten nicht auf große Systeme erweitert werden, weil für die vollständige gemeinsame Verteilung eine exponentielle Anzahl an Wahrscheinlichkeiten benötigt wurde. (Effiziente Algorithmen für Bayessche Netze waren damals noch unbekannt.) Deshalb waren die probabilistischen

Ansätze von 1975 bis 1988 nicht sehr beliebt und aus den verschiedensten Gründen wurden eine Vielzahl von Alternativen zur Wahrscheinlichkeit ausprobiert:

- Häufig besteht die Ansicht, dass die Wahrscheinlichkeitstheorie im Wesentlichen numerisch ist, während das menschliche beurteilende Schließen eher „qualitativ" ist. Natürlich führen wir nicht bewusst numerische Berechnungen zum Glaubensgrad aus. (Und wir nehmen auch keine Unifikation vor, obwohl wir in der Lage wären, bestimmte logische Schlüsse zu ziehen.) Es könnte sein, dass wir über eine Art direkte Codierung numerischer Glaubensgrade in der Stärke der Verbindungen und Aktivierungen in unseren Neuronen verfügen. In diesem Fall ist die Schwierigkeit des bewussten Zugriffes auf diese Stärken nicht überraschend.) Man sollte auch berücksichtigen, dass die qualitativen Schlussmechanismen direkt auf der Wahrscheinlichkeitstheorie aufgebaut werden können, sodass das Argument „keine Zahlen" gegen die Wahrscheinlichkeit wenig zu sagen hat. Nichtsdestotrotz sind einige qualitative Schemas sehr attraktiv. Eines der am besten untersuchten ist das **Default-Schließen**, das Schlüsse nicht als „zu einem bestimmten Grad geglaubt" betrachtet, sondern als „geglaubt, bis ein besserer Grund gefunden wird, um etwas anderes zu glauben". Das Default-Schließen wird in *Kapitel 12* beschrieben.
- Auch **regelbasierte Ansätze** für die Unsicherheit wurden ausprobiert. Solche Ansätze hoffen, auf dem Erfolg der logischen regelbasierten Systeme aufbauen zu können, fügen aber eine Art „Unsinnsfaktor" ein, damit jede Regel die Unsicherheit berücksichtigt. Diese Methoden wurden Mitte der 1970er Jahre entwickelt und bildeten die Grundlage für viele Expertensysteme in der Medizin und in anderen Bereichen.
- Ein Bereich, den wir bisher nicht betrachtet haben, ist die Frage des **Unwissens**, im Gegensatz zur Unsicherheit. Betrachten Sie einen Münzwurf. Wenn wir wissen, dass die Münze ehrlich ist, ist die Wahrscheinlichkeit von 0,5 für Kopf sinnvoll. Wenn wir wissen, dass die Münze manipuliert ist, aber nicht, in welcher Weise, dann ist 0,5 wieder die einzige vernünftige Wahrscheinlichkeit. Offensichtlich unterscheiden sich die beiden Fälle, obwohl sie durch die Ergebniswahrscheinlichkeit scheinbar nicht unterschieden werden. Die **Dempster-Shafer-Theorie** verwendet **intervallwertige** Glaubensgrade, um das Wissen des Agenten über die Wahrscheinlichkeit einer Aussage zu repräsentieren.
- Wahrscheinlichkeit trifft dieselbe ontologische Zusicherung wie die Logik: dass Aussagen in der Welt richtig oder falsch sind, selbst wenn der Agent unsicher ist, was von beiden der Fall ist. Forscher der **Fuzzy-Logik** haben eine Ontologie vorgeschlagen, die die **Vagheit** berücksichtigt; das bedeutet, ein Ereignis kann „eine Art von wahr" sein. Vagheit und Unsicherheit sind letztlich orthogonale Aspekte.

Die nächsten drei Unterabschnitte betrachten einige dieser Ansätze etwas detaillierter. Wir werden hier keine detaillierten technischen Unterlagen präsentieren, aber auf weiterführende Literatur verweisen, sodass Sie Ihre Studien vertiefen können.

14.7.1 Regelbasierte Methoden für unsicheres Schließen

Regelbasierte Systeme entwickelten sich aus frühen Arbeiten zu praktischen und intuitiven Systemen für die logische Inferenz. Logische Systeme im Allgemeinen und logische regelbasierte Systeme im Besonderen haben drei wünschenswerte Eigenschaften:

- **Lokalität:** Wenn wir in logischen Systemen eine Regel der Form $A \Rightarrow B$ haben, können wir B schließen, wenn wir die Evidenz A haben, *ohne uns um andere Regeln kümmern zu müssen*. In probabilistischen Systemen müssen wir *alle* Evidenzen berücksichtigen.
- **Abtrennung:** Nachdem für eine Aussage B ein logischer Beweis gefunden wurde, kann die Aussage genutzt werden, unabhängig davon, wie sie abgeleitet wurde. Das bedeutet, sie kann von ihrer Rechtfertigung **abgetrennt** werden. Beim Umgang mit Wahrscheinlichkeiten dagegen ist die Evidenzquelle für einen Glauben wichtig für nachfolgendes Schließen.
- **Wahrheitsfunktionalität:** In der Logik kann die Wahrheit komplexer Sätze aus der Wahrheit der Komponenten berechnet werden. Die Wahrscheinlichkeitskombination funktioniert nicht so, außer unter strengen globalen Unabhängigkeitsannahmen.

Es gab mehrere Versuche, Schemas für das unsichere Schließen abzuleiten, die diese Vorteile beibehalten. Die Idee dabei ist, den Aussagen und Regeln Glaubensgrade zuzuordnen und rein lokale Schemas für die Kombination und Propagation dieser Glaubensgrade abzuleiten. Die Schemas sind auch wahrheitsfunktional; beispielsweise ist der Glaubensgrad in $A \vee B$ eine Funktion des Glaubens in A und des Glaubens in B.

Tipp

Die schlechte Nachricht für regelbasierte Systeme ist, *dass die Eigenschaften von Lokalität, Abtrennung und Wahrheitsfunktionalität für das unsichere Schließen einfach nicht geeignet sind*. Zuerst wollen wir die Wahrheitsfunktionalität betrachten. Es sei H_1 ein Ereignis, dass ein ehrlicher Münzwurf Kopf erzeugt, und T_1 das Ereignis, dass die Münze beim selben Wurf auf Zahl landet, und H_2 das Ereignis, dass die Münze bei einem zweiten Wurf auf Kopf landet. Offensichtlich haben alle drei Ereignisse dieselbe Wahrscheinlichkeit, 0,5; ein wahrheitsfunktionales System muss deshalb der Disjunktion von zwei beliebigen dieser Ereignisse denselben Glauben zuordnen. Wir sehen jedoch, dass die Wahrscheinlichkeit der Disjunktion von den eigentlichen Ereignissen und nicht nur von ihren Wahrscheinlichkeiten abhängt:

$P(A)$	$P(B)$	$P(A \vee B)$
$P(H_1) = 0{,}5$	$P(H_1) = 0{,}5$	$P(H_1 \vee H_1) = 0{,}50$
	$P(T_1) = 0{,}5$	$P(H_1 \vee T_1) = 1{,}00$
	$P(H_2) = 0{,}5$	$P(H_1 \vee H_2) = 0{,}75$

Das Ganze wird noch schlimmer, wenn wir die Evidenzen verketten. Wahrheitsfunktionale Systeme haben **Regeln** der Form $A \mapsto B$, die es uns erlauben, den Glauben in B als eine Funktion des Glaubens in die Regel und des Glaubens in A zu berechnen. Es können sowohl vorwärts- als auch rückwärtsverkettende Systeme abgeleitet werden. Der Glauben in die Regel soll konstant sein und wird normalerweise durch den Wissensingenieur spezifiziert, z.B. als:

$$A \overset{}{\mapsto}_{0.9} B\,.$$

Betrachten Sie die Situation *NassesGras* aus Abbildung 14.12(b). Wenn wir in der Lage sein wollen, sowohl kausal als auch diagnostisch zu schließen, brauchen wir die beiden folgenden Regeln:

$$\mathit{Regen} \mapsto \mathit{NassesGras} \text{ und } \mathit{NassesGras} \mapsto \mathit{Regen}.$$

Diese beiden Regeln bilden eine Feedback-Schleife: Die Evidenz für *Regen* erhöht den Glauben in *NassesGras*, was wiederum den Glauben in *Regen* verstärkt. Offensichtlich müssen unsichere Schlusssysteme die Pfade beobachten, entlang derer Evidenzen propagiert werden.

Auch das interkausale Schließen (oder Erklären) ist kompliziert. Betrachten Sie, was passiert, wenn wir die beiden folgenden Regeln haben:

$$Sprinkler \mapsto NassesGras \text{ und } NassesGras \mapsto Regen.$$

Angenommen, wir sehen, dass der Sprinkler läuft. Wenn wir unsere Regeln vorwärts durchlaufen, wird dadurch der Glaube gestärkt, dass das Gras nass ist, was wiederum den Glauben stärkt, dass es regnet. Das ist jedoch lächerlich: Die Tatsache, dass der Sprinkler läuft, erklärt das nasse Gras und sollte den Glauben in Regen *reduzieren*. Ein wahrheitsfunktionales System handelt, als ob es auch glaube, $Sprinkler \mapsto Regen$.

Wie lassen sich trotz der Schwierigkeiten wahrheitsfunktionale Systeme praxistauglich machen? Die Antwort liegt in der Beschränkung der Aufgabenstellung und dem sorgfältigen Entwurf der Regelbasis, sodass keine unerwünschten Interaktionen stattfinden. Das berühmteste Beispiel für ein wahrheitsfunktionales System für unsicheres Schließen ist das **Sicherheitsfaktorenmodell**, das für das medizinische Diagnoseprogramm MYCIN entwickelt wurde und das Ende der 1970er und in den 1980er Jahren häufig in Expertensystemen eingesetzt wurde. Fast jeder Einsatz von Sicherheitsfaktoren beinhaltete Regelmengen, die entweder rein diagnostisch waren (wie in MYCIN) oder rein kausal. Darüber hinaus wurden Evidenzen nur in die „Wurzeln" der Regelmenge eingegeben und die meisten Regelmengen waren einfach verknüpft. Heckerman (1986) hat gezeigt, dass unter diesen Umständen eine kleine Abwandlung der Sicherheitsfaktorinferenz äquivalent mit der Bayesschen Inferenz für Polybäume war. Unter anderen Umständen führen bestimmte Faktoren zu katastrophal falschen Glaubensgraden, weil die Evidenzen überbewertet werden. Wenn die Regelmengen größer wurden, so wurden unerwünschte Interaktionen zwischen den Regeln häufiger und die Praktiker stellten fest, dass die Sicherheitsfaktoren vieler anderer Regeln „justiert" werden mussten, wenn neue Regeln hinzugefügt wurden. Aus diesen Gründen haben Bayessche Netze zum größten Teil die regelbasierten Methoden für unsicheres Schließen verdrängt.

14.7.2 Unwissen darstellen: Dempster-Shafer-Theorie

Die **Dempster-Shafer-Theorie** ist darauf ausgelegt, mit der Unterscheidung zwischen **Unsicherheit** und **Unwissen** zurechtzukommen. Anstatt die Wahrscheinlichkeit einer Aussage zu berechnen, berechnet sie die Wahrscheinlichkeit, dass die Evidenz die Aussage unterstützt. Dieses Glaubensmaß wird auch als **Glaubensfunktion** bezeichnet, geschrieben als $Bel(X)$.

Zurück zu unserem Münzwurf als Beispiel für Glaubensfunktionen. Angenommen, Sie ziehen eine Münze aus der Tasche eines Zauberers. Es ist nicht bekannt, ob die Münze ehrlich ist. Welchen Glauben sollten Sie dem Ereignis zuordnen, dass die Münze auf Kopf zu liegen kommt? Gemäß der Dempster-Shafer-Theorie müssen Sie sagen, dass der Glaube $Bel(Kopf) = 0$ und auch $Bel(\neg Kopf) = 0$ ist, weil Sie so oder so keine Evidenzen haben. Das macht Dempster-Shafer-Schlusssysteme in einer Weise skeptisch, die irgendwie intuitiv wirkt. Stellen Sie sich vor, Sie haben einen Experten

an Ihrer Seite, der mit 90-prozentiger Sicherheit sagen kann, dass die Münze ehrlich ist (d.h., er ist zu 90% sicher, dass $P(Kopf) = 0{,}5$ ist). Die Dempster-Shafer-Theorie ermittelt dafür $Bel(Kopf) = 0{,}9 \times 0{,}5 = 0{,}45$ und ebenso $Bel(\neg Kopf) = 0{,}45$. Es gibt immer noch eine zehnprozentige „Lücke“, für die die Evidenz nichts besagt.

Die mathematischen Untermauerungen der Dempster-Shafer-Theorie ähneln denen der Wahrscheinlichkeitstheorie; der Unterschied besteht vor allem darin, dass die Theorie den möglichen Welten keine Wahrscheinlichkeiten zuweist, sondern **Massen** zu *Mengen* möglicher Welten, d.h. zu Ereignissen. Die Massen müssen trotzdem über alle möglichen Ereignisse eine Summe von 1 ergeben. $Bel(A)$ wird definiert als Summe der Massen für alle Ereignisse, die Teilmengen von (d.h. logische Konsequenz aus) A sind, einschließlich A selbst. Mit dieser Definition summieren sich $Bel(A)$ und $Bel(\neg A)$ zu höchstens 1 und die Lücke – das Intervall zwischen $Bel(A)$ und $1 - Bel(\neg A)$ – wird oftmals als Begrenzung der Wahrscheinlichkeit von A interpretiert.

Wie beim Default-Schließen ist es problematisch, Glauben mit Aktionen zu verbinden. Sobald es eine Lücke in den Glauben gibt, lässt sich ein Entscheidungsproblem in der Weise definieren, dass ein Dempster-Shafer-System nicht in der Lage ist, eine Entscheidung zu treffen. Letztlich fehlt noch das umfassende Verständnis für das Nutzenskonzept im Dempster-Shafer-Modell, weil die Bedeutungen von Massen und Glauben an sich noch genauer untersucht werden müssen. Pearl (1988) schlug vor, $Bel(A)$ nicht als Glaubensgrad in A zu interpretieren, sondern als Wahrscheinlichkeit, die allen möglichen Welten zugewiesen wird (jetzt als logische Theorien interpretiert), in denen A beweisbar ist. Auch wenn es Fälle gibt, in denen diese Größe von Interesse sein könnte, ist sie nicht das Gleiche wie die Wahrscheinlichkeit, dass A wahr ist.

Eine Bayessche Analyse des Münzwurfbeispiels würde zeigen, dass kein neuer Formalismus notwendig ist, um derartige Fälle zu behandeln. Das Modell hätte zwei Variablen: die *Tendenz* einer Münze (eine Zahl zwischen 0 und 1, wobei 0 eine Münze kennzeichnet, die immer Zahl zeigt, und 1 eine Münze, die immer Kopf zeigt) und das Ergebnis des nächsten *Wurfes*. Die A-priori-Wahrscheinlichkeitsverteilung für *Tendenz* würde unsere Glauben basierend auf der Herkunft der Münze (der Manteltasche des Zauberers) widerspiegeln: eine geringe Wahrscheinlichkeit, dass sie ideal (fair) ist, und eine gewisse Wahrscheinlichkeit, dass sie stark zu Kopf oder Zahl neigt. Die bedingte Verteilung $\mathbf{P}(Wurf \mid Tendenz)$ definiert einfach, wie die Tendenz wirkt. Wenn $\mathbf{P}(Bias)$ um 0,5 symmetrisch ist, sieht unsere A-priori-Wahrscheinlichkeit für den Wurf folgendermaßen aus:

$$P(Wurf = kopf) = \int_0^1 P(Tendenz = x)P(Wurf = kopf \mid Tendenz = x)dx = 0{,}5.$$

Dies ist die gleiche Voraussage, als ob wir fest glauben, dass die Münze ideal ist, doch bedeutet das nicht, dass die Wahrscheinlichkeitstheorie die beiden Situationen identisch behandelt. Die Unterschiede zeigen sich *nach* den Würfen beim Berechnen der A-posteriori-Verteilung für *Tendenz*. Stammt die Münze von einer Bank, wird es unseren festen A-priori-Glauben in die Fairness der Münze fast nicht beeinflussen, wenn dreimal hintereinander Kopf fällt; doch wenn die Münze aus der Manteltasche des Zauberers kommt, führt dieselbe Evidenz zu einem stärkeren A-posteriori-Glauben, dass die Münze in Bezug auf Kopfwürfe gezinkt ist. Somit drückt ein Bayesscher Ansatz unser „Unwissen“ in einer Form aus, wie sich unser Glaube ändert, wenn später weitere Informationen gesammelt werden können.

14.7.3 Repräsentation von Vagheit: Fuzzy-Mengen und Fuzzy-Logik

Tipp

Die **Fuzzy-Mengen-Theorie** stellt eine Möglichkeit dar, anzugeben, wie gut ein Objekt eine unscharfe Beschreibung erfüllt. Betrachten Sie beispielsweise die Aussage „Mathias ist groß“. Ist das der Fall, wenn Mathias 1,75 m misst? Die meisten Leute würden mit einer Antwort wie „richtig“ oder „falsch“ zögern und vielleicht „relativ“ sagen. Beachten Sie, dass dies keine Frage der Unsicherheit über die externe Welt ist – wir wissen, wie groß Mathias ist. Der Punkt ist, dass der linguistische Begriff „groß“ keine scharfe Abtrennung von Objekten in zwei Klassen bewirkt – es gibt *Größengrade*. Aus diesem Grund *ist die Fuzzy-Mengen-Theorie auf keinen Fall eine Methode für unsicheres Schließen*. Stattdessen behandelt die Fuzzy-Mengen-Theorie *Groß* als Fuzzy-Prädikat und besagt, dass der Wahrheitswert von *Groß*(*Mathias*) eine Zahl zwischen 0 und 1 ist und nicht nur *true* oder *false*. Der Name „Fuzzy-Menge“ leitet sich von der Interpretation des Prädikates ab, dass dieses implizit eine Menge seiner Elemente definiert – eine Menge, die keine scharfen Grenzen aufweist.

Die **Fuzzy-Logik** ist eine Methode des Schließens mit logischen Ausdrücken, die die Zugehörigkeit in Fuzzy-Mengen beschreiben. Der komplexe Satz *Groß*(*Mathias*) ∧ *Schwer*(*Mathias*) beispielsweise hat einen Fuzzy-Wahrheitswert, der eine Funktion der Wahrheitswerte seiner Komponenten ist. Die Standardregeln für die Auswertung der Fuzzy-Wahrheit T eines komplexen Satzes sind:

$$T(A \wedge B) = \min(T(A), T(B))$$
$$T(A \vee B) = \max(T(A), T(B))$$
$$T(\neg A) = 1 - T(A).$$

Die Fuzzy-Logik ist also ein wahrheitsfunktionales System – eine Tatsache, die ernsthafte Probleme verursacht. Nehmen Sie beispielsweise an, es gilt $T(\textit{Groß}(\textit{Mathias})) = 0{,}6$ und $T(\textit{Schwer}(\textit{Mathias})) = 0{,}4$. Wir haben also $T(\textit{Groß}(\textit{Mathias}) \wedge T(\textit{Schwer}(\textit{Mathias})) = 0{,}4$, was sinnvoll erscheint, erhalten aber auch das Ergebnis $T(\textit{Groß}(\textit{Mathias}) \wedge \neg\textit{Groß}(\textit{Mathias})) = 0{,}4$, was weniger sinnvoll ist. Offensichtlich entsteht das Problem aus der Unfähigkeit eines wahrheitsfunktionalen Ansatzes, die Korrelationen oder Antikorrelationen unter den Komponentenaussagen zu berücksichtigen.

Die **Fuzzy-Steuerung** ist eine Methodologie zur Erstellung von Steuerungssystemen, wobei die Zuordnung zwischen reellwertigen Eingabe- und Ausgabeparametern durch Fuzzy-Regeln dargestellt wird. Die Fuzzy-Steuerung war sehr erfolgreich in kommerziellen Produkten wie etwa Automatikgetrieben, Videokameras und Elektrorasierern. Kritiker (siehe z.B. Elkan, 1993) sagen, diese Anwendungen seien deshalb erfolgreich, weil sie kleine Regelbasen und keine Inferenzketten aufweisen und darüber hinaus mit einstellbaren Parametern arbeiten, mit denen die Systemleistung verbessert werden kann. Die Tatsache, dass sie mit Fuzzy-Operatoren implementiert sind, könnte irrelevant für ihren Erfolg sein; wichtig ist einfach, dass eine knappe und intuitive Möglichkeit, eine glatt interpolierte, reellwertige Funktion zu spezifizieren, bereitgestellt wird.

Es gab Versuche, die Fuzzy-Logik in Form der Wahrscheinlichkeitstheorie zu erklären. Eine Idee ist, Zusicherungen wie beispielsweise „Mathias ist groß“ als diskrete Beobachtungen zu betrachten, die für eine stetige verborgene Variable gemacht wurden: die tatsächliche *Größe* von Mathias. Das Wahrscheinlichkeitsmodell spezifiziert P(Beobachter sagt Mathias sei groß | *Größe*), vielleicht unter Verwendung einer **Probit-Verteilung**, wie in *Abschnitt 14.3* beschrieben. Eine A-posteriori-Verteilung über die Größe von Mathias kann dann auf die übliche Weise berechnet werden, wenn das Modell beispielsweise Teil

eines hybriden Bayesschen Netzes ist. Ein solcher Ansatz ist natürlich nicht wahrheitsfunktional. Beispielsweise erlaubt die bedingte Verteilung

$$P(\text{Beobachter sagt Mathias sei groß} \mid \textit{Größe}, \textit{Gewicht})$$

Interaktionen zwischen Größe und Gewicht in der Ursache der Beobachtung. Jemand, der 2,10 m groß ist und 90 kg wiegt, wird wahrscheinlich nicht als „groß und schwer" bezeichnet, obwohl „2,10 m" als „groß" und „90 kg" als „schwer" gilt.

Fuzzy-Prädikate können auch eine probabilistische Interpretation als **Zufallsmengen** erhalten – d.h. Zufallsvariablen, deren mögliche Werte Objektmengen sind. Beispielsweise ist *Groß* eine Zufallsmenge, deren mögliche Werte Mengen von Menschen sind. Die Wahrscheinlichkeit $P(\textit{Groß} = S_1)$, wobei S_1 eine bestimmte Menge von Menschen ist, ist die Wahrscheinlichkeit, dass genau diese Menge von einem Beobachter als „groß" identifiziert würde. Dann ist die Wahrscheinlichkeit, dass „Mathias ist groß" gilt, die Summe der Wahrscheinlichkeiten aller Mengen, denen Mathias angehört.

Sowohl der Ansatz der hybriden Bayesschen Netze als auch der Ansatz der Zufallsmengen scheint Aspekte der Fuzziness abzudecken, ohne Wahrheitsgrade einzuführen. Nichtsdestotrotz bleiben viele offene Punkte im Hinblick auf die korrekte Repräsentation der linguistischen Beobachtungen und stetigen Quantitäten – Punkte, die außerhalb der Fuzzy-Gemeinde größtenteils vernachlässigt wurden.

Bibliografische und historische Hinweise

Seit Anfang des 20. Jahrhunderts verwendet man Netze zur Darstellung probabilistischer Information. Die erste Arbeit auf diesem Gebiet stammt von Sewall Wright, der über die probabilistische Analyse genetischer Vererbung und Wachstumsfaktoren bei Tieren schrieb (Wright, 1921, 1934). I. J. Good (1961) entwickelte in Zusammenarbeit mit Alan Turing probabilistische Repräsentationen und Bayessche Inferenzmethoden, die als Vorläufer moderner Bayesscher Netze betrachtet werden können – obwohl die Arbeit in diesem Kontext selten zitiert wird.[10] Dieselbe Arbeit ist die Originalquelle für das Noisy-OR-Modell.

Ende der 1970er Jahre wurde die Darstellung als **Einflussdiagramme** für Entscheidungsprobleme, die eine DAG-Darstellung für Zufallsvariablen beinhaltete, verwendet (siehe *Kapitel 16*), aber nur die Aufzählung wurde für die Auswertung herangezogen. Judea Pearl entwickelte die Nachrichtenübergabemethode für die Durchführung von Inferenz in Baumnetzen (Pearl, 1982a) und Polybaum-Netzen (Kim und Pearl, 1983) und erklärte die Bedeutung des Aufbaus kausaler statt diagnostischer Wahrscheinlichkeitsmodelle – im Gegensatz zu den damals so beliebten Sicherheitsfaktorsystemen.

Das erste Expertensystem, das Bayessche Netze einsetzte, war CONVINCE (Kim, 1983). Neuere Systeme sind unter anderem MUNIN für die Diagnose neuromuskulärer Störungen (Andersen et al., 1989) sowie PATHFINDER für die Pathologie (Heckerman, 1991). Das CPCS-System (Pradhan et al., 1994) ist ein Bayessches Netz für die Innere Medizin, das aus 448 Knoten, 906 Verknüpfungen und 8.254 bedingten Wahrscheinlichkeitswerten besteht.

10 I.J. Good war Chefstatistiker für das Codeentschlüsselungsteam von Turing im Zweiten Weltkrieg. In *2001: Odyssee im Weltraum* (Clarke, 1968a) wird Good und Minsky der Durchbruch zugeschrieben, der zur Entwicklung des Computers HAL 9000 führte.

Zu den technischen Anwendungen gehören die Arbeit des Electric Power Research Institute zur Überwachung von Stromgeneratoren (Morjaria et al., 1995), die Arbeit der NASA zur Anzeige zeitkritischer Daten im Flugleitzentrum in Houston (Horvitz und Barry, 1995) und das allgemeine Gebiet der **Netzwerktomographie**, das darauf abzielt, nicht beobachtete lokale Eigenschaften von Knoten und Links im Internet aus Beobachtungen der End-zu-End-Nachrichtenleistung herzuleiten (Castro et al., 2004). Die wahrscheinlich bei weitem meist genutzten Bayesschen Netzsysteme waren die Diagnose- und Reparaturmodule (z.B. der Druckerassistent) in Microsoft Windows (Breese und Heckerman, 1996) sowie der Office-Assistent in Microsoft Office (Horvitz et al., 1998). Ein anderes wichtiges Anwendungsgebiet ist die Biologie: Bayessche Netze sind eingesetzt worden, um menschliche Gene durch Verweise auf Gene der Maus zu identifizieren (Zhang et al., 2003), Mobilfunknetze abzuleiten (Friedman, 2004) und viele andere Aufgaben in der Bioinformatik zu lösen. Wir könnten hier fortfahren, verweisen aber stattdessen auf Pourret et a. (2008), einen 400-seitigen Führer zu Anwendungen Bayesscher Netze.

Ross Shachter (1986), der in der Einflussdiagramme-Gemeinde tätig war, entwickelte den ersten vollständigen Algorithmus für allgemeine Bayessche Netze. Seine Methode basiert auf zielgerichteter Reduktion des Netzes mithilfe von Transformationen, die A-posteriori-Informationen bewahren. Pearl (1986) entwickelte einen Clustering-Algorithmus für exakte Inferenz in allgemeinen Bayesschen Netzen und verwendete eine Umwandlung in einen gerichteten Polybaum aus Clustern, in dem die Nachrichtenübergabe verwendet wurde, um Konsistenz für von mehreren Clustern verwendete Variablen zu erzielen. Ein ähnlicher Ansatz, der von den Statistikern David Spiegelhalter und Steffen Lauritzen (Spiegelhalter, 1986; Lauritzen und Spiegelhalter, 1988) entwickelt wurde, basiert auf einer Umwandlung in eine ungerichtete Form eines grafischen Modells, die als **Markov-Netz** bezeichnet wird. Dieser Ansatz ist im System HUGIN implementiert, einem effizienten und allgemein eingesetzten Werkzeug für unsicheres Schließen (Andersen et al., 1989). Boutilier et al. (1996) zeigen, wie kontextspezifische Unabhängigkeit in Clustering-Algorithmen zu nutzen ist.

Das Grundkonzept der Variableneliminierung – dass sich wiederholte Berechnungen innerhalb des Ausdrucks für die Gesamtsummenprodukte durch Zwischenspeichern vermeiden lassen – erschien im SPI (Symbolische Probabilistische Inferenz)-Algorithmus (Shachter et al., 1990). Der von uns beschriebene Eliminierungsalgorithmus kommt dem von Zhang und Poole (1994, 1996) entwickelten am nächsten. Kriterien für das Kürzen irrelevanter Variablen wurden von Geiger et al. (1990) sowie von Lauritzen et al. (1990) entwickelt; das von uns gezeigte Kriterium ist ein Spezialfall davon. Dechter (1999) zeigt, wie das Konzept der Variableneliminierung im Wesentlichen identisch ist mit der **nicht seriellen dynamischen Programmierung** (Bertele und Brioschi, 1972), einem algorithmischen Ansatz, der angewendet werden kann, um einen Bereich von Inferenzproblemen in Bayesschen Netzen zu lösen – um beispielsweise die **wahrscheinlichste Erklärung** für eine Menge von Beobachtungen zu finden. Dies verbindet Algorithmen für Bayessche Netze mit verwandten Methoden zur Lösung von CSPs und bietet ein direktes Maß für die Komplexität exakter Inferenz in Form der Baumbreite des Netzes. Wexler und Meek (2009) beschreiben eine Methode, um exponentielles Wachstum der bei der Variableneliminierung berechneten Faktoren zu verhindern; ihr Algorithmus gliedert große Faktoren in Produkte kleinerer Faktoren und berechnet gleichzeitig eine Fehlerschranke für die resultierende Annäherung.

Stetige Zufallsvariablen in Bayesschen Netzen wurden von Pearl (1988) sowie Shachter und Kenley (1989) betrachtet; diese Arbeiten beschreiben Netze, die nur stetige Variablen mit linearen Gauß-Verteilungen enthalten. Die Aufnahme diskreter Variablen wurde von Lauritzen und Wermuth (1989) untersucht und im System CHUGIN (Olesen, 1993) implementiert. Eine weitergehende Analyse von linearen Gaußmodellen mit Verbindungen zu vielen anderen Modellen, die in der Statistik üblich sind, erscheint in Roweis und Ghahramani (1999). Die Probit-Verteilung wird normalerweise Gaddum (1933) und Bliss (1934) zugeschrieben, obwohl sie im 19. Jahrhundert mehrmals entdeckt wurde. Finney (1947) hat die Arbeit von Bliss erheblich erweitert. Die Probit-Verteilung wurde allgemein für die Modellierung diskreter Auswahlphänomene verwendet und lässt sich erweitern, um mehr als zwei Auswahlmöglichkeiten zu verarbeiten (Daganzo, 1979). Berkson (1944) hat das Logit-Modell eingeführt. Anfangs eher belächelt, ist es letztlich bekannter geworden als das Probit-Modell und Bishop (1995) gibt eine einfache Rechtfertigung für die Verwendung des Modells an.

Cooper (1990) zeigte, dass das allgemeine Problem der Inferenz in nicht beschränkten Bayesschen Netzen NP-hart ist, und Paul Dagum und Mike Luby (1993) zeigten, dass das entsprechende Annäherungsproblem NP-hart ist. Die Speicherkomplexität ist ein ernsthaftes Problem sowohl des Clusterings als auch der Variableneliminierung. Die Methode der **Schnittmengenkonditionierung**, die für CSPs in *Kapitel 6* entwickelt wurde, vermeidet die Erstellung exponentiell großer Tabellen. In einem Bayesschen Netz ist eine Schnittmenge eine Menge von Knoten, die bei der Instantiierung die verbleibenden Knoten auf einen Polybaum reduziert, der in linearer Zeit und mit linearem Speicher gelöst werden kann. Die Abfrage wird beantwortet, indem über alle Instantiierungen der Schnittmenge summiert wird; die Gesamtspeicheranforderung bleibt also linear (Pearl, 1988). Darwiche (2001) beschreibt einen rekursiven Konditionierungsalgorithmus, der ein vollständiges Spektrum an Speicher/Zeit-Abwägungen unterstützt.

Die Entwicklung schneller Annäherungsalgorithmen für Bayessche Netzinferenz ist ein sehr aktiver Bereich mit Beiträgen aus der Statistik, Informatik und Physik. Die Ablehnungs-Sampling-Methode ist eine allgemeine Technik, die den Statistikern lange bekannt ist; Max Henrion (1988) wandte sie als Erster auf Bayessche Netze an und bezeichnete sie als **Logik-Sampling**. Die Wahrscheinlichkeitsgewichtung, die von Fung und Chang (1989) sowie Shachter und Peot (1989) entwickelt wurde, ist ein Beispiel für die bekannte statistische Methode des **Bedeutungs-Samplings**. Cheng und Druzdzel (2000) beschreiben eine adaptive Version der Wahrscheinlichkeitsgewichtung, die selbst dann funktioniert, wenn die Evidenz eine sehr geringe A-priori-Wahrscheinlichkeit besitzt.

MCMC-Algorithmen (Markov Chain Monte Carlo) begannen mit dem Metropolis-Algorithmus, der auf Metropolis et al. (1953) zurückgeht. Sie sind auch die Quelle für den Simulated-Annealing-Algorithmus, den *Kapitel 4* beschreibt. Der Gibbs-Sampler wurde von Geman und Geman (1984) für die Inferenz in ungerichteten Markov-Netzen abgeleitet. Die Anwendung von MCMC auf Bayessche Netze geht auf Pearl (1987) zurück. Die Arbeiten, die Gilks et al. (1996) gesammelt haben, decken einen großen Anwendungsbereich von MCMC ab; einige dieser Anwendungen wurden im bekannten BUGS-Paket entwickelt (Gilks et al., 1994).

Es gibt zwei sehr wichtige Familien von Annäherungsmethoden, die in diesem Kapitel nicht angesprochen wurden. Die erste ist die Familie der **Variationsannäherungs-**

methoden, die für die Vereinfachung komplexer Berechnungen jeder Art eingesetzt werden können. Die grundlegende Idee dabei ist, eine reduzierte Version des ursprünglichen Problems zu betrachten, das leicht zu bearbeiten ist, das jedoch dem ursprünglichen Problem so ähnlich wie möglich ist. Das reduzierte Problem wird durch **Variationsparameter** λ beschrieben, die so angepasst werden, dass sie eine Distanzfunktion D zwischen dem ursprünglichen und dem reduzierten Problem minimieren, häufig durch Lösung des Gleichungssystems $\delta D/\lambda\delta = 0$. Oft kann man damit strenge Ober- und Untergrenzen erhalten. Variationsmethoden werden in der Statistik schon lange eingesetzt (Rustagi, 1976). In der statistischen Physik ist die **Mittelfeldmethode** eine besondere Variationsannäherung, wobei man davon ausgeht, dass die einzelnen Variablen, aus denen sich das Modell zusammensetzt, vollständig voneinander unabhängig sind. Diese Idee wurde angewendet, um große ungerichtete Markov-Netze zu lösen (Peterson und Anderson, 1987; Parisi, 1988). Saul et al. (1996) entwickelten die mathematischen Grundlagen für die Anwendung von Variationsmethoden auf Bayessche Netze und erhielten exakte Untergrenzenannäherungen für Sigmoid-Netze durch Verwendung der Mittelfeldmethode. Jaakkola und Jordan (1996) erweiterten die Methodologie, um sowohl Unter- als auch Obergrenzen zu erhalten. Seit diesen frühen Arbeiten wurden Variationsmethoden auf viele spezifische Familien von Modellen angewandt. Der bemerkenswerte Artikel von Wainwright und Jordan (2008) bietet eine vereinheitlichende theoretische Analyse der Literatur zu Variationsmethoden.

Eine zweite wichtige Familie von Annäherungsalgorithmen basiert auf dem Polybaum-Nachrichtenübergabe-Algorithmus von Pearl (1982a). Dieser Algorithmus kann auf allgemeine Netze angewendet werden, wie von Pearl (1988) vorgeschlagen. Die Ergebnisse könnten fehlerhaft sein und der Algorithmus terminiert möglicherweise nicht, aber in vielen Fällen liegen die dabei erhaltenen Werte ganz nahe bei den tatsächlichen Werten. Diesem Ansatz der sogenannten **Glaubenspropagation** wurde wenig Aufmerksamkeit zuteil, bis McEliece et al. (1998) beobachteten, dass die Nachrichtenübergabe in einem mehrfach verbundenen Bayesschen Netz genau die Berechnung war, die durch den **Turbo-Decoding**-Algorithmus (Berrou et al., 1993) ausgeführt wurde, was einen wichtigen Durchbruch im Entwurf effizienter fehlerkorrigierender Codes darstellte. Die Implikation dabei ist, dass die Glaubenspropagation sowohl schnell als auch exakt für die sehr großen und sehr stark verbundenen Netze für die Dekodierung arbeitet und deshalb auch in allgemeiner Hinsicht praktisch sein könnte. Murphy et al. (1999) stellten eine erfolgversprechende Studie zur Leistung der Glaubenspropagation vor, Weiss und Freeman (2001) begründeten starke Konvergenzergebnisse für Glaubenspropagation in linearen Gaußschen Netzen. Weiss (2000b) zeigt, wie eine als Loopy-Glaubenspropagation bezeichnete Annäherung arbeitet und wann die Annäherung korrekt ist. Yedidia et al. (2005) stellten weitere Verknüpfungen zwischen der Loopy-Propagation und den Ideen aus der statistischen Physik her.

Die Verbindung zwischen Wahrscheinlichkeit und Sprachen erster Stufe wurde zuerst von Carnap (1950) untersucht. Gaifman (1964) sowie Scott und Krauss (1966) definierten eine Sprache, in der Wahrscheinlichkeiten Sätzen erster Stufe zugeordnet werden konnten und für die Modelle Wahrscheinlichkeitsmaße für mögliche Welten waren. Innerhalb der KI wurde diese Idee von Nilsson (1986) für die Aussagenlogik und von Halpern (1990) für die Logik erster Stufe entwickelt. Die erste umfassende Untersuchung von Aspekten der Wissensrepräsentation in solchen Sprachen stammt von Bacchus (1990). Der Grundgedanke ist der, dass jeder Satz in der Wissensbasis eine *Einschränkung* in der Verteilung über möglichen Welten ausdrückt; ein Satz ist die

logische Konsequenz eines anderen, wenn er eine stärkere Einschränkung ausdrückt. Zum Beispiel schließt der Satz $\forall\, x\ P(Hungrig(x)) > 0{,}2$ Verteilungen aus, in denen jedes Objekt mit einer Wahrscheinlich kleiner als 0,2 hungrig ist; somit ergibt sich als logische Konsequenz der Satz $\forall\, x\ P(Hungrig(x)) > 0{,}1$. Es zeigt sich, dass es recht schwierig ist, eine konsistente Menge von Sätzen in diesen Sprachen zu schreiben, und es nahezu unmöglich ist, ein eindeutiges Wahrscheinlichkeitsmodell zu konstruieren, sofern man nicht geeignete Sätze über bedingte Wahrscheinlichkeiten nach dem Darstellungskonzept von Bayesschen Netzen schreibt.

Seit Anfang der 1990er haben Forscher, die an komplexen Anwendungen arbeiten, die ausdrucksmäßigen Beschränkungen von Bayesschen Netzen festgestellt und verschiedene Sprachen entwickelt, um „Vorlagen" mit logischen Variablen zu schreiben, von denen sich große Netze automatisch für jede Probleminstanz konstruieren lassen (Breese, 1992; Wellman et al., 1992). Die wichtigste derartige Sprache war BUGS (Bayesian inference Using Gibbs Sampling) (Gilks et al., 1994), die Bayessche Netze mit dem in der Statistik gebräuchlichen Konzept der **indizierten Zufallsvariablen** kombinierte. (In Bugs sieht eine indizierte Zufallsvariable wie $X[i]$ aus, wobei für i ein Ganzzahlbereich definiert ist.) Diese Sprachen haben die Schlüsseleigenschaft Bayesscher Netze geerbt: Jede gut strukturierte Wissensbasis definiert ein eindeutiges, konsistentes Wahrscheinlichkeitsmodell. Sprachen mit wohldefinierter Semantik, die auf eindeutigen Namen und Domänenabgeschlossenheit basiert, übernahmen die Darstellungsmöglichkeiten der Logikprogrammierung (Poole, 1993; Sato und Kameya, 1997; Kersting et al., 2000) und semantischer Netze (Koller und Pfeffer, 1998; Pfeffer, 2000). Pfeffer (2007) entwickelte IBAL, das Wahrscheinlichkeitsmodelle erster Stufe als probabilistische Programme in einer Programmiersprache darstellt, die mit einem Zufallselement erweitert ist. Ein weiterer wichtiger Gedankengang war die Kombination von relationalen Ansätzen und Konzepten erster Stufe mit (ungerichteten) Markov-Netzen (Taskar et al., 2002; Domingos und Richardson, 2004), wobei es nicht so sehr um die Wissensdarstellung ging, sondern mehr um das Lernen von großen Datenmengen.

Anfangs wurde Inferenz in diesen Modellen über ein äquivalentes Bayessches Netz realisiert. Pfeffer et al. (1999) führten einen Algorithmus zur Variableneliminierung ein, der die berechneten Faktoren zwischenspeicherte, um sie in späteren Berechnungen bei gleichen Beziehungen, aber anderen Objekten wiederzuverwenden. Dabei wurden einige der rechentechnischen Gewinne durch Lifting erkannt. Der erste wirklich angehobene Inferenzalgorithmus war eine angehobene Form der Variableneliminierung, die von Poole (2003) beschrieben und später von de Salvo Braz et al. (2007) verbessert wurde. Weitere Fortschritte, zu denen auch Fälle gehören, in denen sich bestimmte Aggregatwahrscheinlichkeiten in geschlossener Form berechnen lassen, werden von Milch et al. (2008) sowie Kisynski und Poole (2009) beschrieben. Pasula und Russell (2001) untersuchten die Anwendung von MCMC, um das Erstellen des vollständigen Bayesschen Netzes bei relationaler und Identitätsunsicherheit zu vermeiden. Getoor und Taskar (2007) sammeln viele wichtige Arbeiten zu Wahrscheinlichkeitsmodellen erster Stufe und ihrer Verwendung im maschinellen Lernen.

Probabilistisches Schließen über Identitätsunsicherheit hat zwei verschiedene Ursprünge. In der Statistik tritt das Problem der **Duplikaterkennung** (engl. Record Linkage) auf, wenn Datensätze keine eindeutigen Standardbezeichner enthalten – zum Beispiel könnten verschiedene Zitate dieses Buches den ersten Autor als „Stuart Rus-

sell“ oder „S. J. Russel“ oder sogar „Stewart Russle“ nennen und andere Autoren können einige der gleichen Namen verwenden. Buchstäblich Hunderte Firmen existieren allein deshalb, um Probleme der Duplikaterkennung bei Daten im Finanzwesen, in der Medizin, bei Volkszählungen und anderen Bereichen aufzulösen. Probabilistische Analyse geht zurück auf die Arbeit von Dunn (1946); das Fellegi-Sunter-Modell (1969), das im Wesentlichen naives Bayes auf Vergleiche anwendet, dominiert immer noch die aktuelle Praxis. Die zweite Quelle für Arbeiten zu Identitätsunsicherheit ist das Verfolgen mehrerer Ziele (Sittler, 1964), das wir in *Kapitel 15* behandeln. Die Geschichte der Arbeiten in symbolischer KI ist zum größten Teil geprägt durch die irrtümliche Annahme, dass Sensoren Sätze mit eindeutigen Bezeichnern für Objekte bereitstellen könnten. Dieses Problem wurde im Kontext des Sprachverstehens von Charniak und Goldman (1992) sowie im Kontext der Überwachung von Huang und Russel (1998) sowie Pasula et al. (1999) untersucht. Pasula et al. (2003) entwickelten ein komplexes generatives Modell für Autoren-, Artikel- und Zitatzeichenfolgen, die sowohl mit relationaler als auch Identitätsunsicherheit zu tun haben, und demonstrierten hohe Genauigkeit für das Extrahieren von Quellenangaben. Die erste formal definierte Sprache für Wahrscheinlichkeitsmodelle im offenen Universum war BLOG (Milch et al., 2005), die mit einem vollständigen (wenn auch langsamen) MCMC-Algorithmus für wohldefinierte Modelle ausgestattet war. Laskey (2008) beschreibt eine andere Modellierungssprache für offenes Universum namens **MEBN** (**Multi-Entity Bayesian Networks**).

Wie in *Kapitel 13* erklärt, verloren frühe probabilistische Systeme Anfang der 1970er Jahre das Interesse der Fachwelt und hinterließen ein partielles Vakuum, das von alternativen Methoden gefüllt werden musste. Sicherheitsfaktoren wurden für die Verwendung im medizinischen Expertensystem MYCIN erfunden (Shortliffe, 1976), das sowohl als Engineering-Lösung als auch als Modell der menschlichen Beurteilung unter Unsicherheit vorgesehen war. Die Sammlung *Rule-Based Expert Systems* (Buchanan und Shortliffe, 1984) bietet einen vollständigen Überblick über MYCIN und seine Nachfolger (siehe auch Stefik, 1995). David Heckerman (1986) zeigte, dass eine leicht abgewandelte Version der Sicherheitsfaktorberechnungen in einigen Fällen korrekte probabilistische Ergebnisse liefert, während sie in anderen Fällen eine ernsthafte Überbewertung der Evidenzen zeigt. Das Expertensystem PROSPECTOR (Duda et al., 1979) verwendete einen regelbasierten Ansatz, in dem die Regeln durch eine (selten haltbare) globale Unabhängigkeitszusicherung gerechtfertigt wurden.

Die Dempster-Shafer-Theorie hat ihren Ursprung in einer Arbeit von Arthur Dempster (1968), die eine Verallgemeinerung der Wahrscheinlichkeit auf Intervallwerte und eine Kombinationsregel für ihre Verwendung vorschlägt. Spätere Arbeiten von Glenn Shafer (1976) führten dazu, dass die Dempster-Shafer-Theorie als konkurrierender Ansatz zur Wahrscheinlichkeit betrachtet wurde. Pearl (1988) und Ruspini et al. (1992) analysieren die Beziehung zwischen der Dempster-Shafer-Theorie und der Standard-Wahrscheinlichkeitstheorie.

Fuzzy-Mengen wurden von Lotfi Zadeh (1965) entwickelt und sollten auf die wahrgenommene Schwierigkeit reagieren, genaue Eingaben für intelligente Systeme bereitzustellen. Der Text von Zimmermann (2001) bietet eine gründliche Einführung in die Fuzzy-Mengentheorie; Arbeiten zu Fuzzy-Anwendungen sind in Zimmermann (1999) zusammengefasst. Wie bereits im Text erwähnt, wurde die Fuzzy-Logik fälschlicherweise häufig als direkter Konkurrent zur Wahrscheinlichkeitstheorie betrachtet, wäh-

rend sie sich tatsächlich jedoch ganz anderen Problemen widmet. Die **Möglichkeitstheorie** (Zadeh, 1978) wurde eingeführt, um mit der Unsicherheit in Fuzzy-Systemen zurechtzukommen, und sie hat viel mit Wahrscheinlichkeit zu tun. Dubois und Prade (1994) bieten einen Überblick über die Verbindungen zwischen Möglichkeitstheorie und Wahrscheinlichkeitstheorie.

Das Wiederaufleben der Wahrscheinlichkeit beruhte hauptsächlich auf der Entwicklung Bayesscher Netze durch Pearl als Methode für die Repräsentation und die Verwendung bedingter Unabhängigkeitsinformation. Dieses Wiederaufleben erfolgte jedoch nicht ganz kampflos; das kampflustige Buch „In Defense of Probability“ von Peter Cheeseman (1985) sowie sein späterer Artikel „An Inquiry into Computer Understanding“ (Cheeseman, 1988, mit Kommentaren) bieten einen Einblick in die Debatte. Eugene Charniak half mit seinem bekannten Artikel „Bayesian networks without tears“[11] (1991) und Buch (1993), KI-Forschern die Ideen zu vermitteln. Das Buch von Dean und Wellman (1991) trug ebenfalls dazu bei, Bayessche Netze bei AI-Forschern bekanntzumachen. Einer der wichtigsten philosophischen Einwände der Logiker war, dass die numerischen Berechnungen, die man für die Wahrscheinlichkeitstheorie für nötig hielt, für die Introspektion nicht erkennbar waren und eine unrealistische Genauigkeit in unserem unsicheren Wissen voraussetzten. Die Entwicklung **qualitativ probabilistischer Netze** (Wellman, 1990a) stellte eine rein qualitative Abstraktion der Bayesschen Netze bereit und verwendete dazu das Konzept positiver und negativer Einflüsse zwischen Variablen. Wellman zeigt, dass diese Information in vielen Fällen ausreicht, um eine optimale Entscheidung zu treffen, ohne dass die exakte Spezifikation von Wahrscheinlichkeitswerten erforderlich ist. Goldszmidt und Pearl (1996) verfolgen einen ähnlichen Ansatz. Die Arbeit von Adnan Darwiche und Matt Ginsberg (1992) extrahiert die grundlegenden Eigenschaften der Konditionierung und Evidenzkombination aus der Wahrscheinlichkeitstheorie und zeigt, dass sie auch für das logische und das Default-Schließen verwendet werden können. Programme sagen oftmals mehr als einfache Worte und die unmittelbare Verfügbarkeit von hochqualitativer Software wie zum Beispiel das Bayes Net-Toolkit (Murphy, 2001) beschleunigte die Akzeptanz der Technologie.

Die wichtigste Einzelveröffentlichung während der Entstehungszeit Bayesscher Netze ist zweifellos der Text *Probabilistic Reasoning in Intelligent Systems* (Pearl, 1988). Mehrere ausgezeichnete Veröffentlichungen (Lauritzen, 1996; Jensen, 2001; Korb und Nicholson, 2003; Jensen, 2007; Darwiche, 2009; Koller und Friedman, 2009) bieten vollständige Abhandlungen zu den Themen, die wir in diesem Kapitel angerissen haben. Neuere Informationen über die Forschung zum probabilistischen Schließen finden Sie sowohl in Mainstream-Journalen zur KI, wie beispielsweise *Artificial Intelligence* und dem *Journal of AI Research*, als auch in spezialisierteren Magazinen, wie etwa dem *International Journal of Approximate Reasoning*. Viele Arbeiten zu **grafischen Modellen**, die Bayessche Netze einschließen, erscheinen in Statistikmagazinen. Die Tagungsbände der UAI (Uncertainty in Artificial Intelligence), NIPS (Neural Information Processing Systems) und AISTATS (Artificial Intelligence and Statistics) sind ausgezeichnete Quellen für Informationen zur aktuellen Forschung.

11 Die ursprüngliche Version des Artikels hatte den Titel „Pearl for swine“.

Zusammenfassung

Dieses Kapitel hat **Bayessche Netze** beschrieben, eine wohl durchdachte Repräsentation für unsicheres Wissen. Bayessche Netze spielen eine Rolle, die etwa mit der der Aussagenlogik für das definitive Wissen verglichen werden kann.

- Ein Bayessches Netz ist ein gerichteter azyklischer Graph, dessen Knoten Zufallsvariablen entsprechen; jeder Knoten hat eine bedingte Verteilung für den Knoten für bekannte Eltern.
- Bayessche Netze sind eine präzise Möglichkeit, **bedingte Unabhängigkeitsbeziehungen** in der Domäne darzustellen.
- Ein Bayessches Netz spezifiziert eine vollständige gemeinsame Verteilung; jeder gemeinsame Eintrag wird als das Produkt der entsprechenden Einträge in den lokalen bedingten Verteilungen betrachtet. Ein Bayessches Netz ist häufig exponentiell kleiner als eine explizit aufgezählte gemeinsame Verteilung.
- Viele bedingte Verteilungen können kompakt durch kanonische Familien von Verteilungen dargestellt werden. **Hybride Bayessche Netze**, die sowohl diskrete als auch stetige Variablen enthalten, verwenden eine Vielzahl kanonischer Verteilungen.
- Inferenz in Bayesschen Netzen bedeutet die Berechnung der Wahrscheinlichkeitsverteilung einer Menge von Abfragevariablen bei einer bekannten Menge von Evidenzvariablen. Exakte Inferenzalgorithmen, wie etwa die **Variableneliminierung**, werten Summen von Produkten bedingter Wahrscheinlichkeiten so effizient wie möglich aus.
- In **Polybäumen** (einfach verbundenen Netzen) ist die Zeit für Inferenzen linear zur Größe des Netzes. Im allgemeinen Fall ist das Problem nicht handhabbar.
- Stochastische Annäherungstechniken, wie etwa die **Wahrscheinlichkeitsgewichtung** oder **Markov-Ketten Monte Carlo**, können sinnvolle Abschätzungen der tatsächlichen bedingten Wahrscheinlichkeiten in einem Netz liefern und kommen mit sehr viel größeren Netzen zurecht als exakte Algorithmen.
- Die Wahrscheinlichkeitstheorie lässt sich mit Repräsentationskonzepten aus Logik erster Stufe kombinieren, um sehr leistungsfähige Systeme für das Schließen unter Unsicherheit zu schaffen. **Relationale Wahrscheinlichkeitsmodelle** (RWMs) verwenden repräsentationelle Beschränkungen, die eine wohl definierte Wahrscheinlichkeitsverteilung garantieren, die als äquivalentes Bayessches Netz ausgedrückt werden kann. **Wahrscheinlichkeitsmodelle des offenen Universums** verarbeiten **Existenz**- und **Identitätsunsicherheit**, wobei sie Wahrscheinlichkeitsverteilungen über dem unendlichen Raum der möglichen Welten erster Stufe definieren.
- Es wurden verschiedene alternative Systeme für das Schließen unter Unsicherheit vorgeschlagen. Allgemein kann man sagen, dass **wahrheitsfunktionale** Systeme nicht für dieses Schließen geeignet sind.

Übungen zu Kapitel 14

Lösungshinweise

1 In einer Tasche befinden sich drei gezinkte Münzen a, b und c mit den Wahrscheinlichkeiten für Kopf von 30%, 60% bzw. 75%. Eine Münze wird der Tasche zufällig entnommen (wobei die Wahrscheinlichkeit für das Ziehen jeder der drei Münzen gleich ist) und dann dreimal geworfen, um die Ergebnisse X_1, X_2 und X_3 zu erzeugen.

a. Zeichnen Sie das Bayessche Netz entsprechend diesen Vorgaben und definieren Sie die erforderlichen BWTs.

b. Berechnen Sie, welche Münze höchstwahrscheinlich aus der Tasche gezogen wurde, wenn unter den beobachteten Würfen zweimal Kopf und einmal Zahl vorkam.

2 Gleichung (14.1) in *Abschnitt 14.2.1* definiert die gemeinsame Verteilung, die durch ein Bayessches Netz dargestellt wird, in Form der Parameter $\theta(X_i \mid Eltern(X_i))$. In dieser Übung haben Sie die Aufgabe, die Äquivalenz zwischen den Parametern und den bedingten Wahrscheinlichkeiten $\mathbf{P}(X_i \mid Eltern(X_i))$ aus dieser Definition herzuleiten.

a. Betrachten Sie ein einfaches Netz $X \rightarrow Y \rightarrow Z$ mit drei booleschen Variablen. Verwenden Sie die Gleichungen (13.3) und (13.6) (in den *Abschnitten 13.2.1* und *13.3*), um die bedingte Wahrscheinlichkeit $P(z \mid y)$ als Verhältnis von zwei Summen auszudrücken, die jeweils über den Einträgen in der gemeinsamen Verteilung $\mathbf{P}(X, Y, Z)$ gebildet werden.

b. Verwenden Sie nun Gleichung (14.1), um diesen Ausdruck in Form der Netzparameter $\theta(X)$, $\theta(Y \mid X)$ und $\theta(Z \mid Y)$ zu schreiben.

c. Erweitern Sie als Nächstes die Summationen in Ihrem Ausdruck von Teil (b), indem Sie die Terme für die wahren und falschen Werte jeder summierten Variablen explizit ausschreiben. Nehmen Sie an, dass sämtliche Netzparameter die Einschränkung

$$\sum_{x_i} \theta(x_i \mid eltern(X_i)) = 1$$

erfüllen, und zeigen Sie, dass sich der resultierende Ausdruck zu $\theta(z \mid y)$ reduziert.

d. Verallgemeinern Sie diese Herleitung, um zu zeigen, dass $\theta(X_i \mid Eltern(Xi)) = \mathbf{P}(X_i \mid Eltern(X_i))$ für ein beliebiges Bayessches Netz gilt.

3 Die **Kantenumkehr** in einem Bayesschen Netz erlaubt es uns, die Richtung einer Kante $X \rightarrow Y$ zu ändern und dabei die gemeinsame Wahrscheinlichkeitsverteilung, die das Netz darstellt, zu bewahren (Shachter, 1986). Kantenumkehr kann die Einführung neuer Kanten erfordern: Alle Eltern von X werden auch zu Eltern von Y und alle Eltern von Y werden auch zu Eltern von X.

a. Nehmen Sie an, dass X und Y mit m bzw. n Eltern beginnen und dass alle Variablen k Werte haben. Berechnen Sie die Größenänderung für die BWTs von X und Y und zeigen Sie damit, dass die Gesamtanzahl der Parameter im Netz während der Kantenumkehr nicht geringer werden kann. (Hinweis: Die Eltern von X und Y müssen nicht unbedingt disjunkt sein.)

b. Unter welchen Umständen kann die Gesamtanzahl konstant bleiben?

c. Die Eltern von X seien $U \cup V$ und die Eltern von Y seien $V \cup W$, wobei U und W disjunkt sind. Die Formeln für die neuen BWTs nach der Kantenumkehr lauten:

$$\mathbf{P}(Y \mid \mathbf{U}, \mathbf{V}, \mathbf{W}) = \sum_x \mathbf{P}(Y \mid \mathbf{V}, \mathbf{W}, x)\mathbf{P}(x \mid \mathbf{U}, \mathbf{V})$$

$$\mathbf{P}(X \mid \mathbf{U}, \mathbf{V}, \mathbf{W}, Y) = \mathbf{P}(Y \mid X, \mathbf{V}, \mathbf{W})\mathbf{P}(X \mid \mathbf{U}, \mathbf{V})/\mathbf{P}(Y \mid \mathbf{U}, \mathbf{V}, \mathbf{W}).$$

Beweisen Sie, dass das neue Netz dieselbe gemeinsame Verteilung über allen Variablen wie das ursprüngliche Netz ausdrückt.

4 Betrachten Sie das Bayessche Netz in Abbildung 14.2.

a. Sind $Einbruch$ und $Erdbeben$ unabhängig, wenn keine Evidenz beobachtet wird? Beweisen Sie dies aus der numerischen Semantik und aus der topologischen Semantik.

b. Sind $Einbruch$ und $Erdbeben$ unabhängig, wenn wir $Alarm = true$ beobachten? Begründen Sie Ihre Antwort, indem Sie berechnen, ob die relevanten Wahrscheinlichkeiten die Definition von bedingter Unabhängigkeit erfüllen.

5 Es sei H_x eine Zufallsvariable, die die Händigkeit eines Individuums x mit den möglichen Werten l oder r kennzeichnet. Gemäß einer allgemeinen Hypothese wird die Links- oder Rechtshändigkeit durch einen einfachen Mechanismus vererbt; d.h., es gibt möglicherweise ein Gen G_x, ebenfalls mit den Werten l oder r, und vielleicht entspricht die tatsächliche Händigkeit meistens (mit einer bestimmten Wahrscheinlichkeit s) dem Gen, das ein Individuum besitzt. Darüber hinaus ist es aber auch möglich, dass das Gen an sich gleichwahrscheinlich von einem der Elternindividuen vererbt wird, wobei eine zufällige Mutation mit einer kleinen Wahrscheinlichkeit $m > 0$ dafür sorgt, dass die Händigkeit wechselt.

a. Welches der drei Netze in ▶ Abbildung 14.20? behauptet, dass $\mathbf{P}(G_{vater}, G_{mutter}, G_{kind}) = \mathbf{P}(G_{vater})\mathbf{P}(G_{mutter})\mathbf{P}(G_{kind})$ ist?

b. Welches der drei Netze trifft Unabhängigkeitsbehauptungen, die mit der Hypothese über die Vererbung von Händigkeit konsistent sind?

c. Welches der drei Netze beschreibt die Hypothese am besten?

d. Schreiben Sie die BWT für den G_{kind}-Knoten in Netz (a) in Form von s und m nieder.

e. Nehmen Sie an, dass $P(G_{vater} = l) = P(G_{mutter} = l) = q$ ist. Leiten Sie in Netz (a) einen Ausdruck für $P(G_{kind} = l)$ in Form von ausschließlich m und q durch Konditionierung auf ihren Elternknoten her.

f. Unter Bedingungen genetischen Gleichgewichtes erwarten wir, dass die Verteilung der Gene über Generationen hinweg gleich ist. Verwenden Sie dies, um den Wert von q zu berechnen, und erläutern Sie mithilfe Ihrer Kenntnisse zur Händigkeit beim Menschen, warum die zu Beginn dieser Frage beschriebene Hypothese falsch sein muss.

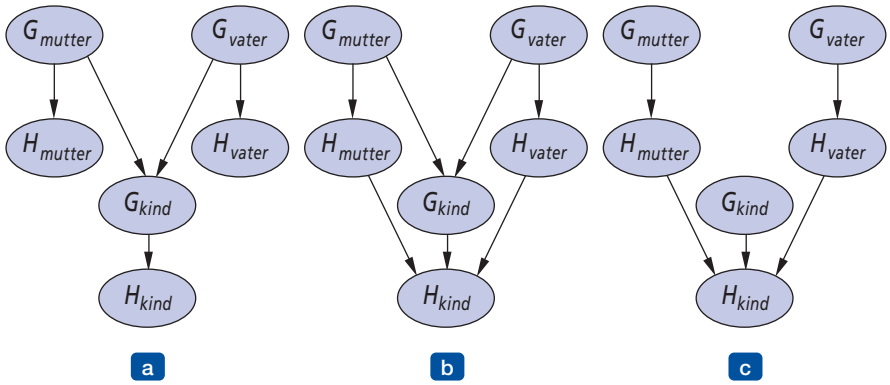


Abbildung 14.20: Drei mögliche Strukturen für ein Bayessches Netz, das genetische Vererbung von Händigkeit beschreibt.

6 Die **Markov-Decke** einer Variablen ist in *Abschnitt 14.2.2* definiert. Beweisen Sie, dass eine Variable von allen anderen Variablen im Netz unabhängig ist, wenn man ihre Markov-Decke kennt. Leiten Sie Gleichung (14.12) ab.

7 Betrachten Sie das in ▶ Abbildung 14.21 gezeigte Netz zur Autodiagnose.

a. Erweitern Sie das Netz um die booleschen Variablen $EisigesWetter$ und $StarterMotor$.

b. Geben Sie sinnvolle bedingte Wahrscheinlichkeitstabellen für alle Knoten an.

c. Wie viele voneinander unabhängige Werte sind in der gemeinsamen Wahrscheinlichkeitsverteilung für acht boolesche Knoten enthalten, vorausgesetzt, es sind keine bedingten Unabhängigkeitsbeziehungen zwischen ihnen bekannt?

d. Wie viele unabhängige Wahrscheinlichkeitswerte enthalten Ihre Netztabellen?

e. Die bedingte Verteilung für $Startet$ könnte als eine **Noisy-AND**-Verteilung beschrieben werden. Definieren Sie diese Familie allgemein und vergleichen Sie sie mit der Noisy-OR-Verteilung.

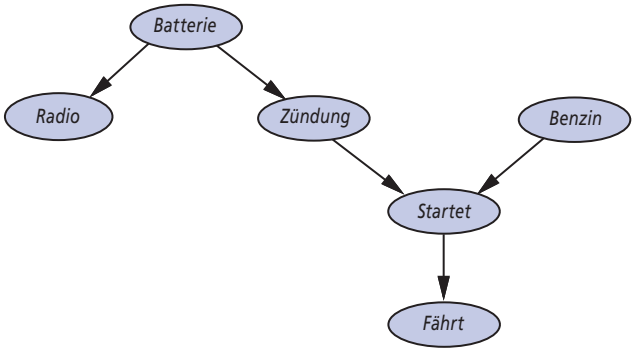


Abbildung 14.21: Ein Bayessches Netz, das einige Funktionsmerkmale des elektrischen Systems und des Motors eines Autos beschreibt. Es handelt sich dabei jeweils um boolesche Variablen und der Wert *true* gibt an, dass der entsprechende Aspekt des Fahrzeuges funktionstüchtig ist.

8 Betrachten Sie ein einfaches Bayessches Netz mit den Wurzelvariablen $Erkältung$, $Grippe$ und $Malaria$ sowie der untergeordneten Variablen $Fieber$ mit einer bedingten Noisy-OR-Verteilung für $Fieber$, wie in *Abschnitt 14.3* beschrieben. Fügen Sie geeignete Hilfsvariablen für Fieber hemmende und Fieber erregende Ereignisse hinzu, um ein äquivalentes Bayessches Netz zu konstruieren, dessen BWTs (mit Ausnahme der Wurzelvariablen) deterministisch sind. Definieren Sie die BWTs und beweisen Sie die Äquivalenz.

9 Betrachten Sie die Familie der linearen Gaußschen Netze, wie sie *Abschnitt 14.3* definiert.

a. In einem Netz mit zwei Variablen soll X_1 der Elternknoten von X_2 sein, X_1 eine Gaußsche A-priori-Verteilung haben und $\mathbf{P}(X_2 \mid X_1)$ eine lineare Gaußsche Verteilung sein. Zeigen Sie, dass die gemeinsame Verteilung $P(X_1, X_2)$ eine multivariate Gaußsche Verteilung ist, und berechnen Sie ihre Kovarianz-Matrix.

b. Beweisen Sie durch Induktion, dass die gemeinsame Verteilung für ein allgemeines lineares Gaußsches Netz über $X_1, ..., X_n$ auch eine multivariate Gaußsche Verteilung ist.

10 Die in *Abschnitt 14.3* definierte Probit-Verteilung beschreibt die Wahrscheinlichkeitsverteilung für ein boolesches Kind bei einem bekannten stetigen Elternknoten.

a. Wie kann die Definition erweitert werden, sodass sie mehrere stetige Elternknoten abdeckt?

b. Wie kann sie erweitert werden, um eine *mehrwertige* Kindvariable zu verarbeiten? Betrachten Sie Fälle, in denen die Werte des Kindes sortiert sind (wie beispielsweise bei der Auswahl eines Ganges beim Fahren, abhängig von Geschwindigkeit, Steigung, gewünschter Beschleunigung usw.), und Fälle, in denen sie nicht sortiert sind (wie etwa die Auswahl von Bus, Zug oder Auto, um zur Arbeit zu gelangen). (*Hinweis*: Ziehen Sie Möglichkeiten in Betracht, die möglichen Werte in zwei Mengen zu unterteilen, um eine boolesche Variable nachzubilden.)

11 Im Atomkraftwerk gibt es eine Alarmanlage, die überprüft, ob bei einer Temperaturmessung ein bestimmter Schwellenwert überschritten wird. Die Messung greift die Temperatur des Kerns ab. Wir haben die booleschen Variablen A (Alarm klingelt), F_A (Alarm war fehlerhaft) und F_M (Messung ist fehlerhaft) sowie die mehrwertigen Knoten M (Messung lesen) und T (tatsächliche Kerntemperatur).

a. Zeichnen Sie ein Bayessches Netz für diese Domäne. Nehmen Sie dabei an, dass die Messung wahrscheinlicher einen Fehler erzeugt, wenn die Kerntemperatur zu hoch wird.

b. Ist Ihr Netz ein Polybaum? Warum oder warum nicht?

c. Angenommen, es gibt zwei mögliche tatsächliche und gemessene Temperaturen, „normal" und „hoch"; die Wahrscheinlichkeit, dass die Messung die korrekte Temperatur ergibt, ist x, wenn sie funktioniert, und y, wenn sie fehlerhaft ist. Geben Sie die bedingte Wahrscheinlichkeitstabelle für M an.

d. Angenommen, der Alarm funktioniert korrekt, es sei denn, er ist fehlerhaft, dann klingelt er nie. Geben Sie die bedingte Wahrscheinlichkeitstabelle für A an.

e. Angenommen, der Alarm und die Messung funktionieren und der Alarm klingelt. Berechnen Sie einen Ausdruck für die Wahrscheinlichkeit, dass die Temperatur des Kerns zu hoch ist. Verwenden Sei dazu die verschiedenen bedingten Wahrscheinlichkeiten im Netz.

12 Zwei Astronomen auf unterschiedlichen Erdteilen machen mithilfe ihrer Teleskope die Messungen M_1 und M_2 zur Anzahl der Sterne N in einem kleinen Himmelsbereich. Normalerweise gibt es eine kleine Fehlerwahrscheinlichkeit e bis zu einem Stern in jeder Richtung. Die beiden Teleskope können auch (mit einer sehr viel kleineren Wahrscheinlichkeit f) einen Fehler in der Fokussierung aufweisen (Ereignisse F_1 und F_2), dann zählt der Wissenschaftler drei oder mehr Sterne nicht (oder falls N kleiner 3 ist, erkennt er überhaupt keine Sterne). Betrachten Sie die in ▶ Abbildung 14.22 gezeigten Netze.

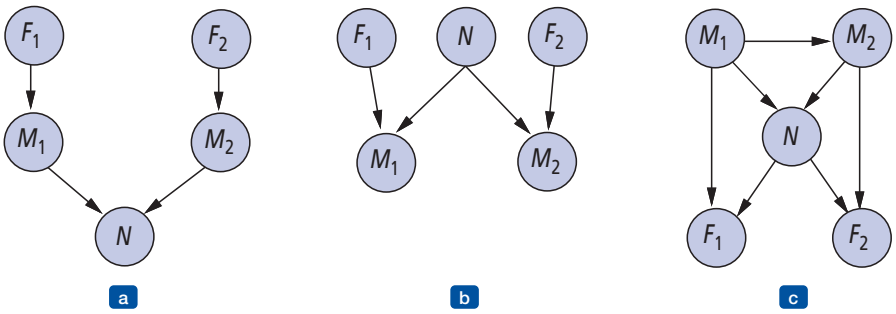


Abbildung 14.22: Drei mögliche Netze für das Teleskopproblem.

a. Welche dieser Bayesschen Netze sind korrekte (aber nicht notwendigerweise effiziente) Repräsentationen der oben bereitgestellten Information?
b. Welches ist das beste Netz? Erläutern Sie Ihre Antwort.
c. Schreiben Sie eine bedingte Verteilung für $\mathbf{P}(M_1 \mid N)$ für den Fall $N \in \{1, 2, 3\}$ und $M_1 \in \{0, 1, 2, 3, 4\}$. Jeder Eintrag in der bedingten Verteilung soll als Funktion der Parameter e und/oder f ausgedrückt werden.
d. Nehmen Sie $M_1 = 1$ und $M_2 = 3$ an. Welche *mögliche* Anzahl an Sternen liegt vor, wenn wir keine A-priori-Einschränkung für die Werte von N annehmen?
e. Was ist die *wahrscheinlichste* Anzahl an Sternen für diese Beobachtungen? Erklären Sie, wie Sie sie berechnen, oder, falls sie nicht zu berechnen ist, welche zusätzlichen Informationen benötigt werden und wie sie das Ergebnis beeinflussen.

13 Betrachten Sie das Bayessche Netz in ▶ Abbildung 14.23.

a. Welche der folgenden Aussagen (sofern zutreffend) werden durch die Netz*struktur* zugesichert (wobei die BWTs zunächst zu ignorieren sind)?

(i) $\mathbf{P}(B, I, M) = \mathbf{P}(B)\mathbf{P}(I)\mathbf{P}(M)$
(ii) $\mathbf{P}(J \mid G) = \mathbf{P}(J \mid G, I)$
(iii) $\mathbf{P}(M \mid G, B, I) = \mathbf{P}(M \mid G, B, I, J)$

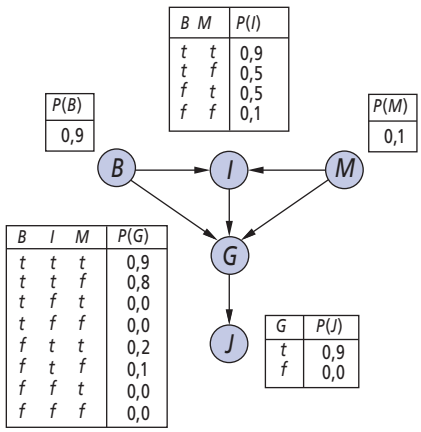


Abbildung 14.23: Ein einfaches Bayessches Netz mit den booleschen Variablen $B = BrokenElectionLaw$ (Wahlgesetz gebrochen), $I = Indicted$ (beschuldigt), $M = PoliticallyMotivatedProsecutor$ (politisch motivierter Ankläger), $G = FoundGuilty$ für schuldig befunden), $J = Jailed$ (inhaftiert).

b. Berechnen Sie den Wert von $P(b, i, m, \neg g, j)$.
c. Berechnen Sie die Wahrscheinlichkeit, dass jemand ins Gefängnis kommt, wenn er das Gesetz gebrochen hat, beschuldigt wurde und einem politisch motivierten Ankläger gegenübersteht.
d. Eine **kontextspezifische Unabhängigkeit** (siehe *Abschnitt 14.6.2*) erlaubt es einer Variablen, unabhängig von einigen ihren Eltern zu sein, wenn bestimmte Werte von anderen gegeben sind. Welche kontextspezifischen Unabhängigkeiten existieren im Bayes-Netz in Abbildung 14.23 außer den üblichen bedingten Unabhängigkeiten, die durch die Graphstruktur gegeben sind?
e. Angenommen, wir möchten dem Netz die Variable $P = PresidentialPardon$ (Begnadigung durch Präsidenten) hinzufügen; zeichnen Sie das neue Netz und erläutern Sie kurz die hinzugefügten Verknüpfungen.

14 Diese Übung bezieht sich auf den Algorithmus zur Variableneliminierung aus Abbildung 14.11.

a. *Abschnitt 14.4* wendet die Variableneliminierung auf die folgende Abfrage an:

$$\mathbf{P}(Einbruch | JohnRuftAn = true, MaryRuftAn = true)$$

Führen Sie die erforderlichen Berechnungen aus und überprüfen Sie, ob die Antwort korrekt ist.

b. Zählen Sie, wie viele arithmetische Operationen ausgeführt werden, und vergleichen Sie sie mit der Anzahl der Operationen, die durch den Aufzählungsalgorithmus ausgeführt werden.

c. Angenommen, ein Netz hat die Form einer *Kette*: eine Folge boolescher Variablen $X_1, ..., X_n$, wobei $Eltern(X_i) = \{X_{i-1}\}$ für $i = 2, ..., n$. Welche Komplexität entsteht für die Berechnung von $\mathbf{P}(X_1 | X_n = true)$ bei Verwendung der Aufzählung? Bei Verwendung der Variableneliminierung?

d. Beweisen Sie, dass die Komplexität bei der Ausführung der Variableneliminierung für ein Polybaum-Netz linear zur Größe des Baumes für jede mit der Netzstruktur konsistente Variablenreihenfolge ist.

15 Untersuchen Sie die Komplexität exakter Inferenz in allgemeinen Bayesschen Netzen:

a. Beweisen Sie, dass jedes 3-SAT-Problem auf eine exakte Inferenz in einem Bayesschen Netz reduziert werden kann, das erstellt wurde, um das betreffende Problem darzustellen, und dass die exakte Inferenz somit NP-hart ist. (*Hinweis*: Betrachten Sie ein Netz mit einer Variablen für jedes Aussagesymbol, einer für jede Klausel und einer für die Konjunktion von Klauseln.)

b. Das Problem, die Anzahl zufriedenstellender Zuweisungen für ein 3-SAT-Problem zu ermitteln, ist #P-vollständig. Zeigen Sie, dass die exakte Inferenz mindestens genauso schwierig ist.

16 Betrachten Sie das Problem, eine zufällige Stichprobe aus einer vorgegebenen Verteilung über einer einzelnen Variablen zu erzeugen. Nehmen Sie an, dass ein Zufallszahlengenerator zur Verfügung steht, der Zufallszahlen zurückgibt, die zwischen 0 und 1 gleich verteilt sind.

a. Es sei X eine diskrete Variable mit $P(X = x_i) = p_i$ für $i \in \{1,..., k\}$. Die **kumulative Verteilung** von X gibt die Wahrscheinlichkeit an, dass $X \in \{x_1,..., x_j\}$ für jedes mögliche j (siehe auch Anhang A). Erklären Sie, wie Sie in einer Zeit von $O(k)$ die kumulative Verteilung berechnen und wie Sie daraus eine einzelne Stichprobe von X erzeugen. Kann die letztgenannte Aufgabe in einer Zeit kleiner $O(k)$ erledigt werden?

b. Nehmen wir nun an, wir wollen N Stichproben aus X erzeugen, wobei $N >> k$. Erklären Sie, wie sich dies mit einer erwarteten *konstanten* (d.h. von k unabhängigen) Laufzeit pro Stichprobe erledigen lässt.

c. Betrachten Sie jetzt eine stetigwertige Variable mit parametrisierter (z.B. Gaußscher) Verteilung. Wie können aus einer solchen Verteilung Stichproben erzeugt werden?

d. Angenommen, Sie wollen eine stetigwertige Variable abfragen und verwenden dazu einen Sampling-Algorithmus wie etwa LIKELIHOODWEIGHTING für die Inferenz. Wie müssten Sie den Abfrage/Antwort-Prozess abändern?

17 Betrachten Sie die Abfrage $\mathbf{P}(Regen \mid Sprinkler = true, NassesGras = true)$ in Abbildung 14.12(a) und wie sie mit MCMC beantwortet werden kann.

a. Wie viele Zustände hat die Markov-Kette?

b. Berechnen Sie die **Übergangsmatrix** $\mathbf{Q}$, die $q(\mathbf{y} \rightarrow \mathbf{y}')$ für alle $\mathbf{y}$, $\mathbf{y}'$ enthält.

c. Was stellt das Quadrat der Übergangsmatrix $\mathbf{Q}^2$ dar?

d. Was bedeutet $\mathbf{Q}^n$ für $n \to \infty$?

e. Erklären Sie, wie man eine probabilistische Inferenz in Bayesschen Netzen durchführt, und zwar unter der Annahme, dass $\mathbf{Q}^n$ bekannt ist. Ist dies eine praktische Möglichkeit, Inferenz auszuführen?

18 Diese Übung untersucht die stationäre Verteilung für Gibbs-Sampling-Methoden.

a. Die konvexe Zusammensetzung $[\alpha, q_1; 1 - \alpha, q_2]$ von q_1 und q_2 ist eine Übergangswahrscheinlichkeitsverteilung, die zuerst eines der Elemente von q_1 und q_2 mit den Wahrscheinlichkeiten α bzw. $1 - \alpha$ auswählt und dann das ausgewählte Element anwendet. Beweisen Sie, dass sich die konvexe Zusammensetzung ebenfalls im detaillierten Gleichgewicht mit π befindet, wenn q_1 und q_2 im detaillierten Gleichgewicht mit π sind. (*Hinweis:* Dieses Ergebnis rechtfertigt eine Variante von GIBBS-ASK, in der Variablen zufällig ausgewählt und nicht in einer festen Reihenfolge gesampelt werden.)

b. Beweisen Sie, dass die sequentielle Zusammensetzung $q = q_1 \circ q_2$ ebenfalls π als ihre stationäre Verteilung hat, wenn q_1 und q_2 jeweils π als ihre stationäre Verteilung haben.

19 Der **Metropolis-Hastings**-Algorithmus gehört zur MCMC-Familie und ist als solcher dafür konzipiert, Stichproben $\mathbf{x}$ entsprechend der Zielwahrscheinlichkeiten $\pi(\mathbf{x})$ zu erzeugen. (Normalerweise sind wir an einer Stichprobe von $\pi(\mathbf{x}) = P(\mathbf{x} \mid \mathbf{e})$ interessiert.) Wie Simulated Annealing arbeitet Metropolis-Hastings in zwei Phasen. Zuerst erzeugt er eine Stichprobe für einen neuen Zustand $\mathbf{x}'$ aus einer **Vorschlagsverteilung** $q(\mathbf{x?} \mid \mathbf{x})$, wobei der aktuelle Zustand $\mathbf{x}$ gegeben ist. Dann akzeptiert oder verwirft er $\mathbf{x}'$ entsprechend der Akzeptanzwahrscheinlichkeit

$$\alpha(\mathbf{x}' \mid \mathbf{x}) = \min\left(1, \frac{\pi(\mathbf{x}')q(\mathbf{x} \mid \mathbf{x}')}{\pi(\mathbf{x})q(\mathbf{x}' \mid \mathbf{x})}\right).$$

Wird der Vorschlag zurückgewiesen, verbleibt der Zustand in $\mathbf{x}$.

a. Betrachten Sie einen normalen Schritt beim Gibbs-Sampling für eine bestimmte Variable X_i. Zeigen Sie, dass dieser Schritt – als Vorschlag betrachtet – vom Metropolis-Hastings-Algorithmus garantiert akzeptiert wird. (Folglich ist Gibbs-Sampling ein Spezialfall von Metropolis-Hastings.)

b. Zeigen Sie, dass der obige Zweiphasenprozess – als Übergangswahrscheinlichkeitsverteilung betrachtet – im detaillierten Gleichgewicht mit π ist.

20 Drei Fußballmannschaften, A, B und C, spielen jeder gegen jeden. Jedes Spiel erfolgt zwischen zwei Mannschaften und kann gewonnen werden, unentschieden ausgehen oder verloren werden. Jede Mannschaft hat einen feststehenden unbekannten Qualitätsgrad – eine ganze Zahl zwischen 0 und 3 – und das Ergebnis eines Spieles hängt probabilistisch von der Qualitätsdifferenz zwischen den beiden Mannschaften ab.

a. Erstellen Sie ein relationales Wahrscheinlichkeitsmodell, das diese Domäne beschreibt, und schlagen Sie numerische Werte für alle erforderlichen Wahrscheinlichkeitsverteilungen vor.

b. Konstruieren Sie das äquivalente Bayessche Netz für die drei Spiele.

c. Angenommen, in den beiden ersten Spielen gewinnt A gegen B und spielt unentschieden gegen C. Berechnen Sie unter Verwendung eines exakten Inferenzalgorithmus Ihrer Wahl die bedingte Verteilung für das Ergebnis des dritten Spieles.

d. Nehmen Sie an, dass es n Mannschaften in der Liga gibt und wir alle Ergebnisse außer für das letzte Spiel haben. Wie variiert die Komplexität der Vorhersage für das letzte Spiel mit n?

e. Betrachten Sie die Anwendung von MCMC auf dieses Problem. Wie schnell konvergiert es in der Praxis und wie gut kann es für größer werdende Probleme angewendet werden?

Probabilistisches Schließen über die Zeit

15

ÜBERBLICK

In diesem Kapitel versuchen wir, die Gegenwart zu interpretieren, die Vergangenheit zu verstehen und vielleicht die Zukunft vorherzusagen, auch wenn nur sehr wenig wirklich klar ist.

Agenten in partiell beobachtbaren Umgebungen müssen in der Lage sein, den aktuellen Zustand zu verfolgen, und zwar in dem Umfang, wie es ihre Sensoren erlauben. *Abschnitt 4.4* hat hierfür eine Methodologie gezeigt: Ein Agent verwaltet einen **Belief State**, der darstellt, welche Zustände der Welt momentan möglich sind. Aus dem Belief State und einem **Übergangsmodell** kann der Agent vorhersagen, wie sich die Welt im nächsten Zeitschritt entwickeln könnte. Anhand der beobachteten Wahrnehmungen und eines **Sensormodells** ist der Agent in der Lage, den Belief State zu aktualisieren. Dies ist eine verbreitete Idee: In *Kapitel 4* wurden Belief States durch explizit aufgezählte Zustandsmengen dargestellt, während sie in den *Kapiteln 7* und *11* durch logische Formeln repräsentiert wurden. Diese Konzepte haben Belief States in einer Form definiert, die angibt, welche Weltzustände *möglich* wären, konnten aber nichts darüber aussagen, welche Zustände *wahrscheinlich* oder *unwahrscheinlich* sind. In diesem Kapitel greifen wir auf die Wahrscheinlichkeitstheorie zurück, um den Glaubensgrad in Elementen des Belief State zu quantifizieren.

Abschnitt 15.1 zeigt, dass die Zeit selbst in der gleichen Weise wie in *Kapitel 7* behandelt wird: Eine sich ändernde Welt wird mithilfe einer Variablen für jeden Aspekt des Weltzustandes *zu jedem Zeitpunkt* modelliert. Die Übergangs- und Sensormodelle können unsicher sein: Das Übergangsmodell beschreibt die Wahrscheinlichkeitsverteilung der Variablen zur Zeit t, wobei der Zustand der Welt zu früheren Zeitpunkten gegeben ist, während das Sensormodell die Wahrscheinlichkeit jeder Wahrnehmung zur Zeit t beschreibt, wobei der aktuelle Zustand der Welt gegeben ist. *Abschnitt 15.2* definiert die grundlegenden Inferenzaufgaben und beschreibt die allgemeine Struktur von Inferenzalgorithmen für temporale Modelle. Anschließend gehen wir auf drei spezielle Modelle ein: **Hidden-Markov-Modelle**, **Kalman-Filter** und **dynamische Bayessche Netze** (die die Hidden-Markov-Modelle sowie Kalman-Filter als Spezialfälle beinhalten). Schließlich untersucht *Abschnitt 15.6* die Probleme, die auftreten, wenn mehrere Dinge auf einmal verfolgt werden.

15.1 Zeit und Unsicherheit

Wir haben Techniken für das probabilistische Schließen im Kontext *statischer* Welten entwickelt, wo jede Zufallsvariable einen einzigen, feststehenden Wert hat. Wenn wir beispielsweise ein Auto reparieren, gehen wir davon aus, dass kaputte Teile auch während des Diagnosevorganges kaputt bleiben; unsere Aufgabe ist es, den Zustand des Autos aus den sichtbaren Evidenzen abzuleiten, die ebenfalls feststehend bleiben.

Jetzt betrachten wir ein etwas anderes Problem: die Behandlung eines Diabetespatienten. Wie bei der Autoreparatur haben wir auch hier Evidenzen, wie etwa die letzten Insulingaben, Nahrungsaufnahme, Blutzuckermessungen und andere physische Merkmale. Die Aufgabe ist, den aktuellen Zustand des Patienten einzuschätzen, einschließlich Blutzuckerwert und Insulinspiegel. Mit diesen Informationen trifft der Arzt (oder der Patient) eine Entscheidung zur Nahrungsaufnahme und zur Insulingabe für den Patienten. Anders als bei der Autoreparatur spielen hier jedoch die *dynamischen* Aspekte des Problems eine wichtige Rolle. Blutzuckerwerte und ihre Messungen kön-

nen sich schnell ändern – abhängig von der letzten Nahrungsaufnahme und den Insulindosen, der körperlichen Aktivität, der Tageszeit usw. Um aus diesem Evidenzverlauf den aktuellen Zustand einzuschätzen und die Ergebnisse der Behandlungsmaßnahmen vorherzusagen, müssen wir diese Änderungen nachbilden.

Die gleichen Betrachtungen finden wir in vielen anderen Bereichen, beispielsweise bei der Positionsverfolgung eines Roboters, bei der Beobachtung der wirtschaftlichen Aktivität eines Landes bis hin zum Verstehen einer Folge gesprochener oder geschriebener Wörter. Wie lassen sich dynamische Situationen wie diese nachbilden?

15.1.1 Zustände und Beobachtungen

Wir betrachten die Welt als eine Folge von Momentaufnahmen, sogenannten **Zeitscheiben**, die jeweils eine Menge von Zufallsvariablen enthalten – manche beobachtbar und manche nicht.[1] Der Einfachheit halber wollen wir davon ausgehen, dass in jeder Zeitscheibe dieselbe Teilmenge von Variablen beobachtbar ist (obwohl das im Folgenden nicht immer streng erforderlich sein muss). Mit $\mathbf{X}_t$ bezeichnen wir die Menge nicht beobachtbarer Zustandsvariablen zur Zeit t und $\mathbf{E}_t$ kennzeichnet die Menge der beobachtbaren Evidenzvariablen. Die Beobachtung zur Zeit t ist $\mathbf{E}_t = \mathbf{e}_t$ für eine Menge von Werten $\mathbf{e}_t$.

Betrachten Sie das folgende Beispiel: Sie gehören zum Wachpersonal irgendeiner geheimen unterirdischen Einrichtung. Sie wollen wissen, ob es heute regnet, aber Ihr einziger Zugriff auf die Außenwelt findet allmorgendlich statt, wenn Sie sehen, ob der Direktor mit oder ohne Schirm eintrifft. Für jeden Tag t enthält die Menge $\mathbf{E}_t$ also eine einzige Evidenzvariable U_t (ob der Regenschirm (englisch „umbrella“) erscheint) und die Menge $\mathbf{X}_t$ enthält eine einzige Zustandsvariable R_t (ob es regnet). Andere Probleme können größere Variablenmengen beinhalten. Im Diabetesbeispiel könnten wir etwa Evidenzvariablen wie $\mathit{GemessenerBlutzucker}_t$ oder $\mathit{PulsHöhe}_t$ verwenden sowie Zustandsvariablen wie etwa $\mathit{Blutzucker}_t$ und $\mathit{Mageninhalt}_t$. (Beachten Sie, dass $\mathit{Blutzucker}_t$ und $\mathit{GemessenerBlutzucker}_t$ nicht dieselbe Variable darstellen; so gehen wir mit verrauschten Messungen tatsächlicher Mengen um.)

Das Intervall zwischen den Zeitscheiben ist ebenfalls vom Problem abhängig. Für die Diabetesüberwachung könnte ein Intervall von einer Stunde besser geeignet sein als ein Intervall von einem Tag. In diesem Kapitel nehmen wir ein festes Intervall zwischen den Zeitscheiben an, sodass wir die Zeiten durch Ganzzahlen bezeichnen können. Wir gehen davon aus, dass die Zustandsfolge bei $t = 0$ beginnt. Aus verschiedenen (hier nicht weiter interessierenden) Gründen wollen wir annehmen, dass Evidenzen ab $t = 1$ und nicht ab $t = 0$ eintreffen. Unsere Regenschirmwelt wird also dargestellt durch die Zustandsvariablen R_0, R_1, R_2, ... sowie die Evidenzvariablen U_1, U_2, ... Wir verwenden die Notation $a{:}b$, um die Abfolge der ganzen Zahlen von a bis b (einschließlich) zu kennzeichnen, und die Notation $\mathbf{X}_{a:b}$ bezeichnet die entsprechende Variablenmenge von $\mathbf{X}_a$ bis $\mathbf{X}_b$. Beispielsweise steht $U_{1:3}$ für die Variablen U_1, U_2, U_3.

1 Unsicherheit über *stetiger* Zeit lässt sich durch **stochastische Differentialgleichungen** modellieren. Die in diesem Kapitel untersuchten Modelle können als Annäherungen in diskreter Zeit an stochastische Differentialgleichungen angesehen werden.

15.1.2 Übergangs- und Sensormodelle

Nachdem die Menge der Zustands- und Evidenzvariablen für ein bestimmtes Problem festliegt, ist im nächsten Schritt zu spezifizieren, wie sich die Welt entwickelt (das Übergangsmodell) und wie die Evidenzvariablen ihre Werte erhalten (das Sensormodell).

Das Übergangsmodell spezifiziert die Wahrscheinlichkeitsverteilung über den neuesten Zustandsvariablen bei bekannten vorherigen Zuständen, d.h. $\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{0:t-1})$. Jetzt taucht ein Problem auf: Die Menge $\mathbf{X}_{0:t-1}$ ist in der Größe unbeschränkt, wenn t zunimmt. Dieses Problem lösen wir, indem wir die **Markov-Annahme** treffen – d.h., dass der aktuelle Zustand nur von einer *endlichen festen Anzahl* von vorherigen Zuständen abhängt. Prozesse, die diese Annahme erfüllen, wurden zuerst von dem russischen Statistiker Andrej Markov (1856–1922) untersucht und heißen **Markov-Prozesse** oder **Markov-Ketten**. Es gibt sie in verschiedenen Ausprägungen; die einfachste davon ist der **Markov-Prozess erster Stufe**, in dem der aktuelle Zustand nur vom vorherigen Zustand und nicht von früheren Zuständen abhängig ist. Mit anderen Worten bietet ein Zustand genügend Informationen, um die Zukunft bedingt unabhängig von der Vergangenheit zu machen, und wir haben

$$\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{0:t-1}) = \mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1}). \tag{15.1}$$

Folglich ist das Übergangsmodell in einem Markov-Prozess erster Stufe die bedingte Verteilung $\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1})$. Das Übergangsmodell für einen Markov-Prozess zweiter Stufe ist die bedingte Verteilung $\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-2}, \mathbf{X}_{t-1})$. ▶ Abbildung 15.1 zeigt die Strukturen des Bayesschen Netzes, die den Markov-Prozessen erster und zweiter Stufe entsprechen.

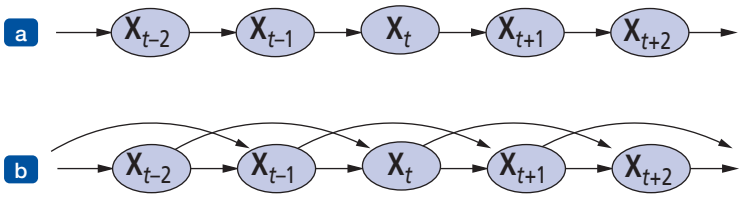


Abbildung 15.1: (a) Struktur des Bayesschen Netzes, die einem Markov-Prozess erster Stufe entspricht, der durch die Variablen $\mathbf{X}_t$ definiert ist. (b) Ein Markov-Prozess zweiter Stufe.

Selbst mit der Markov-Annahme bleibt ein Problem bestehen: Es gibt unendlich viele mögliche Werte von t. Müssen wir wirklich für jeden Zeitschritt eine andere Verteilung spezifizieren? Wir vermeiden dieses Problem, indem wir annehmen, dass Änderungen im Weltzustand durch einen **stationären Prozess** verursacht werden – d.h. durch einen Änderungsprozess, der durch Gesetze beherrscht wird, die sich selbst über die Zeit nicht ändern. (Verwechseln Sie *stationär* nicht mit *statisch*: In einem *statischen* Prozess ändert sich der eigentliche Zustand selbst nicht.) In der Regenschirmwelt ist dann die bedingte Regenwahrscheinlichkeit $\mathbf{P}(R_t \mid R_{t-1})$ für alle t gleich und wir müssen lediglich eine Tabelle mit bedingten Wahrscheinlichkeiten angeben.

Kommen wir nun zum Sensormodell. Die Evidenzvariablen $\mathbf{E}_t$ *könnten* sowohl von vorherigen Variablen als auch von den aktuellen Zustandsvariablen abhängen, doch sollte jeder Zustand, der etwas taugt, genügen, um die aktuellen Sensorwerte zu generieren. Wir treffen also eine **Sensor-Markov-Annahme** wie folgt:

$$\mathbf{P}(\mathbf{E}_t \mid \mathbf{X}_{0:t}, \mathbf{E}_{0:t-1}) = \mathbf{P}(\mathbf{E}_t \mid \mathbf{X}_t). \tag{15.2}$$

Somit ist $\mathbf{P}(\mathbf{E}_t \mid \mathbf{X}_t)$ unser Sensormodell (manchmal auch **Beobachtungsmodell** genannt). ▶ Abbildung 15.2 zeigt sowohl das Übergangsmodell als auch das Sensormodell für das Regenschirmbeispiel. Beachten Sie die Richtung der Abhängigkeit zwischen Zustand und Sensoren: Die Pfeile verlaufen vom eigentlichen Zustand der Welt zu Sensorwerten, weil der Zustand der Welt *bewirkt*, dass die Sensoren bestimmte Werte annehmen: Der Regen *bewirkt*, dass der Regenschirm erscheint. (Der Inferenzprozess verläuft natürlich in die andere Richtung; die Unterscheidung zwischen der Richtung modellierter Abhängigkeiten und der Inferenzrichtung gehört zu den wichtigsten Vorteilen Bayesscher Netze.)

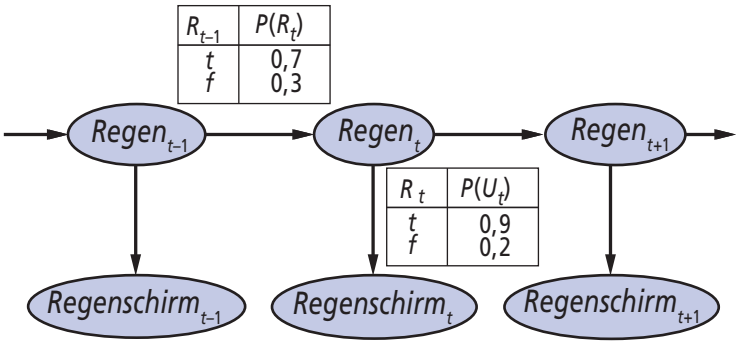


Abbildung 15.2: Struktur des Bayesschen Netzes und bedingte Verteilungen, die die Regenschirmwelt beschreiben. Das Übergangsmodell ist $P(Regen_t \mid Regen_{t-1})$, das Sensormodell ist $P(Regenschirm_t \mid Regen_t)$.

Wir müssen nicht nur die Übergangs- und Sensormodelle festlegen, sondern auch angeben, wie alles beginnen soll – die A-priori-Wahrscheinlichkeitsverteilung zur Zeit 0, $\mathbf{P}(\mathbf{X}_0)$. Damit haben wir eine Spezifikation der vollständigen gemeinsamen Verteilung über alle Variablen unter Verwendung von Gleichung (14.2). Für ein beliebiges t gilt:

$$\mathbf{P}(\mathbf{X}_{0:t}, \mathbf{E}_{1:t}) = \mathbf{P}(\mathbf{X}_0) \prod_{i=1}^{t} \mathbf{P}(\mathbf{X}_i \mid \mathbf{X}_{i-1}) \mathbf{P}(\mathbf{E}_i \mid \mathbf{X}_i). \tag{15.3}$$

Die drei Terme auf der rechten Seite sind das Ausgangszustandsmodell $\mathbf{P}(\mathbf{X}_0)$, das Übergangsmodell $\mathbf{P}(\mathbf{X}_i \mid \mathbf{X}_{i-1})$ und das Sensormodell $\mathbf{P}(\mathbf{E}_i \mid \mathbf{X}_i)$.

Die Struktur in Abbildung 15.2 ist ein Markov-Prozess erster Stufe – die Wahrscheinlichkeit für Regen ist nur davon abhängig, ob es am vorherigen Tag geregnet hat. Ob eine solche Annahme sinnvoll ist, hängt von der eigentlichen Domäne ab. Die Markov-Annahme erster Stufe besagt, dass die Zustandsvariablen *alle* Informationen enthalten, die man benötigt, um die Wahrscheinlichkeitsverteilung für die nächste Zeitscheibe zu charakterisieren. Manchmal trifft die Annahme genau zu – z.B. wenn ein Partikel einen zufälligen Weg entlang der x-Achse verfolgt und bei jedem Zeitschritt seine Position um ±1 verändert, ergibt die Verwendung der x-Koordinate als Zustand einen Markov-Prozess erster Stufe. Manchmal handelt es sich bei der Annahme nur um eine Schätzung, wie es etwa bei der Vorhersage von Regen ist, die nur darauf basiert, ob es am vorhergehenden Tag geregnet hat. Es gibt zwei Möglichkeiten, um die Genauigkeit der Annäherung zu verbessern:

1. Die Stufe des Markov-Prozessmodells erhöhen: Beispielsweise könnten wir ein Modell zweiter Stufe erzeugen, indem wir $Regen_{t-1}$ als Eltern von $Regen_t$ einfügen, wodurch wir eine etwas genauere Vorhersage treffen können. Zum Beispiel regnet es in Palo Alto, Kalifornien, sehr selten an mehr als zwei aufeinanderfolgenden Tagen.

2. Die Menge der Zustandsvariablen vergrößern: Beispielsweise könnten wir $Jahreszeit_t$ hinzufügen, um historische Aufzeichnungen regnerischer Jahreszeiten berücksichtigen zu können, oder wir könnten $Temperatur_t$, $Luftfeuchtigkeit_t$ oder $Luftdruck_t$ einfügen (gegebenenfalls für mehrere Orte), womit wir ein physisches Modell von Regenbedingungen aufbauen können.

In Übung 15.1 sollen Sie zeigen, dass die erste Lösung – eine Erhöhung der Stufe – immer als eine Vergrößerung der Menge der Zustandsvariablen umformuliert werden kann, wobei die Reihenfolge beibehalten wird. Beachten Sie, dass das Hinzufügen von Zustandsvariablen die Vorhersageleistung des Systems verbessern könnte – aber auch die *Anforderungen* an die Vorhersage: Wir müssen jetzt auch alle neuen Variablen vorhersagen. Wir suchen also nach einer „ausreichenden" Menge von Variablen, d.h., wir müssen unbedingt die „Physik" des zu modellierenden Prozesses verstehen. Die Anforderung für eine exakte Modellierung des Prozesses wird offenbar gelockert, wenn wir neue Sensoren einführen können (z.B. Messungen von Temperatur und Druck), die direkte Informationen über die neuen Zustandsvariablen bereitstellen.

Betrachten Sie beispielsweise das Problem, einen Roboter zu verfolgen, der sich zufällig in der X-Y-Ebene bewegt. Man könnte davon ausgehen, dass Position und Geschwindigkeit eine ausreichende Menge von Zustandsvariablen darstellen: Man verwendet einfach das Newtonsche Gesetz, um die neue Position zu berechnen, und die Geschwindigkeit kann sich unvorhersagbar ändern. Ist der Roboter jedoch batteriebetrieben, hätte die Ladekapazität der Batterien eine systematische Wirkung auf die Geschwindigkeitsänderung. Weil dies wiederum davon abhängig ist, wie viel Strom von allen vorhergehenden Bewegungen verbraucht wurde, ist die Markov-Eigenschaft verletzt. Wir können die Markov-Eigenschaft wiederherstellen, indem wir die Ladestufe $Batterie_t$ als eine der Zustandsvariablen von $\mathbf{X}_t$ einführen. Auf diese Weise können wir die Bewegung des Roboters besser voraussagen, aber wir brauchen wiederum ein Modell für die Vorhersage von $Batterie_t$ aus $Batterie_{t-1}$ und die Geschwindigkeit. In einigen Fällen kann dies zuverlässig erfolgen; die Genauigkeit würde jedoch verbessert, wenn man einen *neuen Sensor für den Batterieladezustand* hätte.

15.2 Inferenz in temporalen Modellen

Nachdem wir die Struktur eines generischen temporalen Modells eingerichtet haben, können wir die grundlegenden Inferenzaufgaben formulieren, die gelöst werden müssen:

- **Filtern** oder **Überwachen:** Dies ist die Aufgabe, den **Belief State** zu berechnen – die A-posteriori-Verteilung über den aktuellen Zustand –, wenn alle bisherigen Evidenzen bekannt sind. Filtern[2] wird auch als **Zustandsabschätzung** bezeichnet. In unse-

2 Der Begriff „Filtern" weist auf die Wurzeln dieses Problems in frühen Arbeiten zur Signalverarbeitung hin, wo das größte Problem darin bestand, das Rauschen in einem Signal auszufiltern, indem dessen zugrunde liegende Eigenschaften bewertet wurden.

rem Beispiel wollen wir $\mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$ berechnen. Im Regenschirmbeispiel würde das die Berechnung der heutigen Regenwahrscheinlichkeit bedeuten, wobei alle bisherigen Beobachtungen des Schirmträgers berücksichtigt werden. Beim Filtern versucht ein rationaler Agent, den aktuellen Zustand zu verfolgen, sodass rationale Entscheidungen getroffen werden können. Es stellt sich heraus, dass eine fast identische Berechnung die Wahrscheinlichkeit der Evidenzfolge $P(\mathbf{e}_{1:t})$ ergibt.

- **Vorhersage:** Dies ist die Aufgabe, die bedingte Verteilung über den *zukünftigen* Zustand für alle bisherigen Evidenzen zu berechnen. Das bedeutet, wir wollen $\mathbf{P}(\mathbf{X}_{t+k} \mid \mathbf{e}_{1:t})$ für ein $k > 0$ berechnen. Im Regenschirmbeispiel könnte das bedeuten, wir berechnen die Wahrscheinlichkeit, dass es von heute an drei Tage lang regnet, wobei wir alle bisherigen Beobachtungen des Schirmträgers kennen. Die Vorhersage ist praktisch, um mögliche Aktionsfolgen auszuwerten.
- **Glättung:** Dies ist die Aufgabe, die bedingte Verteilung über einen *vergangenen* Zustand zu berechnen, wobei wir alle Evidenzen bis zum aktuellen Zeitpunkt kennen. Das bedeutet, wir wollen $\mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:t})$ für ein k berechnen, sodass $0 \leq k < t$. Im Regenschirmbeispiel könnte das bedeuten, wir berechnen die Wahrscheinlichkeit, dass es letzten Mittwoch geregnet hat, wenn wir alle Beobachtungen des Regenschirmverlaufes bis heute kennen. Die Glättung bietet eine bessere Abschätzung des Zustandes, als sie zu der entsprechenden Zeit möglich war, weil sie mehr Evidenzen beinhaltet.[3]
- **Wahrscheinlichste Erklärung:** Für eine Beobachtungsfolge wollen wir die Zustandsfolge ermitteln, die diese Beobachtungen am wahrscheinlichsten erzeugt hat. Das bedeutet, wir wollen $\arg\max_{x_{1:t}} P\left(x_{1:t} \middle| e_{1:t}\right)$ berechnen. Erscheint beispielsweise der Regenschirm an jedem der drei ersten Tage und fehlt er am vierten, dann ist die wahrscheinlichste Erklärung, dass es an den ersten drei Tagen geregnet hat, am vierten Tag aber nicht. Algorithmen für diese Aufgabe sind in vielen Anwendungen sehr praktisch, wie beispielsweise in der Spracherkennung – wo die wahrscheinlichste Wortsequenz für eine gegebene Menge von Geräuschfolgen zu ermitteln ist – oder in der Rekonstruktion von Bitstrings, die über einen verrauschten Kanal übertragen werden.

Zu diesen Inferenzaufgaben kommt noch Folgende:

- **Lernen:** Falls die Übergangs- und Sensormodelle noch nicht bekannt sind, können sie auch aus Beobachtungen gelernt werden. Genau wie bei Bayesschen Netzen kann Lernen von dynamischen Bayesschen Netzen als Nebenprodukt der Inferenz stattfinden. Die Inferenz bietet eine Abschätzung, welche Übergänge tatsächlich stattgefunden und welche Zustände die Sensorwerte erzeugt haben. Anhand dieser Schätzungen lassen sich die Modelle aktualisieren. Die aktualisierten Modelle bieten neue Abschätzungen und der Prozess iteriert bis zur Konvergenz. Der Gesamtprozess ist eine Instanz des Erwartungsmaximierungs- oder **EM-Algorithmus** (siehe *Abschnitt 20.3*).

Lernen erfordert Glättung statt Filtern, weil Glättung bessere Schätzungen der Prozesszustände liefert. Lernen mit Filtern konvergiert möglicherweise nicht korrekt; betrachten Sie beispielsweise das Problem, Mordfälle aufzuklären: Sofern man nicht

3 Insbesondere beim Verfolgen eines sich bewegenden Objektes mit ungenauen Positionsbeobachtungen ergibt Glättung im Vergleich zum Filtern eine glattere geschätzte Bewegungsbahn – daher auch der Name.

Augenzeuge war, ist Glättung *immer* erforderlich, um aus den beobachteten Variablen abzuleiten, was beim Mord geschehen ist.

Der Rest dieses Abschnittes beschreibt generische Algorithmen für die vier Inferenzaufgaben – und zwar unabhängig von der konkreten Art des verwendeten Modells. Die darauffolgenden Abschnitte beschreiben Verbesserungen, die für das jeweilige Modell spezifisch sind.

15.2.1 Filtern und Vorhersage

Wie bereits in *Abschnitt 7.7.3* erwähnt, muss ein brauchbarer Filteralgorithmus eine aktuelle Zustandsschätzung verwalten und sie aktualisieren, anstatt für jede Aktualisierung den gesamten zurückliegenden Verlauf von Wahrnehmungen erneut durchzugehen. (Andernfalls nehmen die Kosten für jede Aktualisierung im Laufe der Zeit zu.) Mit anderen Worten muss der Agent für ein bekanntes Filterergebnis bis zur Zeit t das Ergebnis für $t + 1$ aus der neuen Evidenz $\mathbf{e}_{t+1}$ für eine Funktion f berechnen, d.h.

$$\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t+1}) = f(\mathbf{e}_{t+1}, \mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})).$$

Dieser Prozess wird als **rekursive Schätzung** bezeichnet. Wir können die Berechnung als zweiteilig betrachten: Zunächst wird die aktuelle Zustandsverteilung von t weiter auf $t + 1$ projiziert; anschließend wird sie unter Verwendung der neuen Evidenz $\mathbf{e}_{1:t}$ aktualisiert. Dieser zweiteilige Prozess entwickelt sich ganz einfach, wenn die Formel umgestellt wird:

$$\begin{aligned}
&\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t+1}) = \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t}, \mathbf{e}_{t+1}) \text{ (Aufteilung der Evidenz)} \\
&= \alpha \mathbf{P}(\mathbf{e}_{t+1} \mid \mathbf{X}_{t+1}, \mathbf{e}_{1:t}) \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t}) \text{ (Anwendung der Bayesschen Regel)} \\
&= \alpha \mathbf{P}(\mathbf{e}_{t+1} \mid \mathbf{X}_{t+1}) \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t}) \text{ (aufgrund der Sensor-Markov-Annahme).} \qquad (15.4)
\end{aligned}$$

Hier und im gesamten Kapitel ist α eine Normalisierungskonstante, die verwendet wird, um für Wahrscheinlichkeiten die Summe 1 zu erhalten. Der zweite Term, $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t})$, stellt eine einphasige Vorhersage des nächsten Zustandes dar und der erste Term aktualisiert sie mit der neuen Evidenz. Beachten Sie, dass man $\mathbf{P}(\mathbf{e}_{t+1} \mid \mathbf{X}_{t+1})$ direkt aus dem Sensormodell erhält. Jetzt erhalten wir die einphasige Vorhersage für den nächsten Zustand, indem wir für den aktuellen Zustand $\mathbf{X}_t$ konditionieren:

$$\begin{aligned}
\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t+1}) &= \alpha \mathbf{P}(\mathbf{e}_{t+1} \mid \mathbf{X}_{t+1}) \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t, \mathbf{e}_{1:t}) P(\mathbf{x}_t \mid \mathbf{e}_{1:t}) \\
&= \alpha \mathbf{P}(\mathbf{e}_{t+1} \mid \mathbf{X}_{t+1}) \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t) P(\mathbf{x}_t \mid \mathbf{e}_{1:t}) \text{ (Markov-Annahme).} \qquad (15.5)
\end{aligned}$$

Innerhalb der Summation kommen der erste Faktor vom Übergangsmodell und der zweite Faktor von der aktuellen Zustandsverteilung. Wir haben also die gewünschte rekursive Formulierung. Wir können uns die gefilterte Schätzung $\mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$ als die „Nachricht" $\mathbf{f}_{1:t}$ vorstellen, die entlang der Folge weitergegeben, in jedem Übergang abgeändert und durch jede neue Beobachtung aktualisiert wird. Der Prozess ist gegeben durch

$$\mathbf{f}_{1:t+1} = \alpha \, \text{FORWARD}(\mathbf{f}_{1:t}, \mathbf{e}_{t+1}),$$

wobei FORWARD die in Gleichung (15.5) beschriebene Aktualisierung implementiert und der Prozess mit $\mathbf{f}_{1:0} = \mathbf{P}(\mathbf{X}_0)$ beginnt.

Tipp

Wenn alle Zustandsvariablen diskret sind, ist die Zeit für jede Aktualisierung konstant (d.h. von t unabhängig), ebenso wie der benötigte Speicherplatz. (Die Konstanten hängen natürlich von der Größe des Zustandsraumes und dem jeweiligen Typ des betreffenden Zeitmodells ab.) *Die Zeit- und Speicheranforderungen für die Aktualisierung müssen konstant sein, wenn ein Agent mit begrenztem Speicher die aktuelle Zustandsverteilung über eine unbegrenzte Beobachtungsfolge verfolgen soll.*

Jetzt demonstrieren wir den Filterprozess für zwei Schritte im einfachen Regenschirmbeispiel (siehe Abbildung 15.2). Das heißt, wir berechnen $\mathbf{P}(R_2 \mid u_{1:2})$ wie folgt:

- Am Tag 0 haben wir keine Beobachtungen, sondern nur den A-priori-Glauben unseres Sicherheitsbeauftragten; wir nehmen an, dass dieser aus $\mathbf{P}(R_0) = \langle 0{,}5;\ 0{,}5\rangle$ besteht.
- Am Tag 1 erscheint der Regenschirm, $U_1 = \mathit{true}$. Die Voraussage von $t = 0$ auf $t = 1$ ist also

$$\begin{aligned}\mathbf{P}(R_1) &= \sum_{r_0} \mathbf{P}(R_1 \mid r_0) P(r_0) \\ &= \langle 0{,}7;\ 0{,}3\rangle \times 0{,}5 + \langle 0{,}3;\ 0{,}7\rangle \times 0{,}5 = \langle 0{,}5;\ 0{,}5\rangle.\end{aligned}$$

  Der Aktualisierungsschritt multipliziert dann einfach die Wahrscheinlichkeit der Evidenz für $t = 1$ und normalisiert das Ergebnis, wie in Gleichung (15.4) gezeigt:

$$\begin{aligned}\mathbf{P}(R_1 \mid u_1) &= \alpha \mathbf{P}(u_1 \mid R_1)\mathbf{P}(R_1) = \alpha\langle 0{,}9;\ 0{,}2\rangle\langle 0{,}5;\ 0{,}5\rangle \\ &= \alpha\langle 0{,}45;\ 0{,}1\rangle \approx \langle 0{,}818;\ 0{,}182\rangle.\end{aligned}$$

- An Tag 2 erscheint der Regenschirm, also ist $U_2 = \mathit{true}$. Die Voraussage von $t = 1$ auf $t = 2$ ist

$$\begin{aligned}\mathbf{P}(R_2 \mid u_1) &= \sum_{r_1} \mathbf{P}(R_2 \mid r_1) P(r_1 \mid u_1) \\ &= \langle 0{,}7;\ 0{,}3\rangle \times 0{,}818 + \langle 0{,}3;\ 0{,}7\rangle \times 0{,}182 \approx \langle 0{,}627;\ 0{,}373\rangle\end{aligned}$$

  und eine Aktualisierung mit der Evidenz für $t = 2$ ergibt:

$$\begin{aligned}\mathbf{P}(R_2 \mid u_1, u_2) &= \alpha \mathbf{P}(u_2 \mid R_2)\mathbf{P}(R_2 \mid u_1) = \alpha\langle 0{,}9;\ 0{,}2\rangle\langle 0{,}627;\ 0{,}373\rangle \\ &= \alpha\langle 0{,}565;\ 0{,}075\rangle \approx \langle 0{,}883;\ 0{,}117\rangle.\end{aligned}$$

Intuitiv steigt die Regenwahrscheinlichkeit von Tag 1 auf Tag 2, weil der Regen andauert. In Übung 15.2(a) werden Sie diese Tendenz weiter untersuchen.

Die Aufgabe der **Vorhersage** kann einfach als Filterung ohne das Hinzufügen neuer Evidenzen betrachtet werden. Der Filterprozess beinhaltet bereits eine einphasige Vorhersage und man kann leicht die folgende rekursive Berechnung für die Vorhersage des Zustandes zum Zeitpunkt $t + k + 1$ aus einer Vorhersage für $t + k$ ableiten:

$$\mathbf{P}(\mathbf{X}_{t+k+1} \mid \mathbf{e}_{1:t}) = \sum_{x_{t+k}} \mathbf{P}(\mathbf{X}_{t+k+1} \mid \mathbf{x}_{t+k}) P(\mathbf{x}_{t+k} \mid \mathbf{e}_{1:t}). \tag{15.6}$$

Natürlich beinhaltet diese Berechnung nur das Übergangsmodell und nicht das Sensormodell.

Interessant ist, was passiert, wenn wir versuchen, immer weiter in die Zukunft vorherzusagen. Wie in Übung 15.2(b) gezeigt, konvergiert die vorhergesagte Verteilung für Regen bei einem Fixpunkt von $\langle 0{,}5;\ 0{,}5\rangle$, wonach sie für immer konstant bleibt. Dies ist

die **stationäre Verteilung** des Markov-Prozesses, die durch das Übergangsmodell definiert ist (siehe auch *Abschnitt 14.5.2*). Es ist viel über die Eigenschaften solcher Verteilungen und die **Mischzeit** bekannt – grob gesagt, die Zeit bis zum Erreichen des Fixpunktes. In der Praxis ist damit jeder Versuch zum Scheitern verurteilt, den *tatsächlichen* Zustand für eine Anzahl an Schritten vorherzusagen, die mehr als einen kleinen Bruchteil der Mischzeit darstellen, außer wenn die stationäre Verteilung selbst in einem kleinen Bereich des Zustandsraumes eine steile Spitze aufweist. Je mehr Unsicherheit es im Übergangsmodell gibt, desto kürzer ist die Mischzeit und desto verschleierter ist die Zukunft.

Neben dem Filtern und der Vorhersage können wir auch eine Vorwärtsrekursion verwenden, um die **Wahrscheinlichkeit** der Evidenzfolge $P(\mathbf{e}_{1:t})$ zu berechnen. Das ist eine nützliche Größe, wenn wir verschiedene temporale Modelle vergleichen wollen, die die gleiche Evidenzfolge erzeugt haben (z.B. zwei unterschiedliche Modelle für die Dauerhaftigkeit von Regen). Für diese Rekursion verwenden wir die Wahrscheinlichkeitsnachricht $\ell_{1:t}(\mathbf{X}_t) = \mathbf{P}(\mathbf{X}_t, \mathbf{e}_{1:t})$. Es lässt sich leicht zeigen, dass die Nachrichtenberechnung identisch mit der für das Filtern ist:

$$\ell_{1:t+1} = \text{FORWARD}(\ell_{1:t}, \mathbf{e}_{t+1}).$$

Nachdem wir $\ell_{1:t}$ berechnet haben, erhalten wir die tatsächliche Wahrscheinlichkeit durch eine Aussummierung von $\mathbf{X}_t$:

$$L_{1:t} = P(\mathbf{e}_{1:t}) = \sum_{\mathbf{x}_t} \ell_{1:t}(\mathbf{x}_t). \tag{15.7}$$

Beachten Sie, dass die Wahrscheinlichkeitsnachricht mit zunehmender Zeitdauer die Wahrscheinlichkeiten von immer länger werdenden Evidenzfolgen darstellt und somit numerisch immer kleiner wird, was zu Unterlaufproblemen bei Gleitkommaarithmetik führt. Dies ist ein wichtiges Problem in der Praxis, doch gehen wir hier nicht auf entsprechende Lösungen ein.

15.2.2 Glättung

Wie bereits erwähnt, ist die Glättung der Prozess, die Verteilung über die vergangenen Zustände mit Evidenzen bis zur Gegenwart zu berechnen, d.h. $\mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:t})$ für $0 \leq k < t$ (siehe ▶ Abbildung 15.3). Im Vorgriff auf einen anderen Ansatz zur rekursiven Nachrichtenübergabe können wir die Berechnung in zwei Teile gliedern – die Evidenz bis k und die Evidenz von $k + 1$ bis t berechnen:

$$\begin{aligned}
\mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:t}) &= \mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:k}, \mathbf{e}_{k+1:t}) \\
&= \alpha\, \mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:k}) \mathbf{P}(\mathbf{e}_{k+1:t}, \mathbf{e}_{k+1:t} \mid \mathbf{X}_k, \mathbf{e}_{1:k}) && \text{(Bayessche Regel)} \\
&= \alpha\, \mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:k}) \mathbf{P}(\mathbf{e}_{k+1:t} \mid \mathbf{X}_k) && \text{(bedingte Unabhängigkeit)} \\
&= \alpha\, \mathbf{f}_{1:k} \times \mathbf{b}_{k+1:t}, && (15.8)
\end{aligned}$$

wobei „×“ eine punktweise Multiplikation von Vektoren darstellt. Hier haben wir eine „Rückwärts“-Nachricht $\mathbf{b}_{k+1:t} = \mathbf{P}(\mathbf{e}_{k+1:t} \mid \mathbf{X}_k)$ definiert, analog zu der Vorwärtsnachricht $\mathbf{f}_{1:k}$. Die Vorwärtsnachricht $\mathbf{f}_{1:k}$ kann berechnet werden, indem man vorwärts von 1 bis k filtert, wie durch Gleichung (15.5) vorgegeben. Es stellt sich heraus, dass die Rückwärtsnachricht $\mathbf{b}_{k+1:t}$ durch einen rekursiven Prozess berechnet werden kann, der *rückwärts* von t verläuft:

$$\begin{aligned}\mathbf{P}(\mathbf{e}_{k+1:t} \mid \mathbf{X}_k) &= \sum_{\mathbf{x}_{k+1}} \mathbf{P}(\mathbf{e}_{k+1:t} \mid \mathbf{X}_k, \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} \mid \mathbf{X}_k) \text{ (konditioniert auf } \mathbf{X}_{k+1}) \\ &= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1:t} \mid \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} \mid \mathbf{X}_k) \text{ (bedingte Unabhängigkeit)} \\ &= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1}, \mathbf{e}_{k+2:t} \mid \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} \mid \mathbf{X}_k) \\ &= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1} \mid \mathbf{x}_{k+1}) P(\mathbf{e}_{k+2:t} \mid \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} \mid \mathbf{X}_k). \end{aligned} \tag{15.9}$$

Dabei folgt der letzte Schritt durch die bedingte Unabhängigkeit von $\mathbf{e}_{k+1}$ und $\mathbf{e}_{k+2:t}$ für bekanntes $\mathbf{X}_{k+1}$. Von den drei Faktoren dieser Summation stammen der erste und der dritte direkt aus dem Modell und der zweite ist der „rekursive Aufruf". Unter Verwendung der Nachrichtennotation haben wir:

$$\mathbf{b}_{k+1:t} = \text{BACKWARD}(\mathbf{b}_{k+2:t}, \mathbf{e}_{k+1:t}).$$

Dabei implementiert BACKWARD die in Gleichung (15.9) beschriebene Aktualisierung. Wie bei der Vorwärtsrekursion sind Zeit- und Speicherbedarf für die Aktualisierung konstant und damit unabhängig von t.

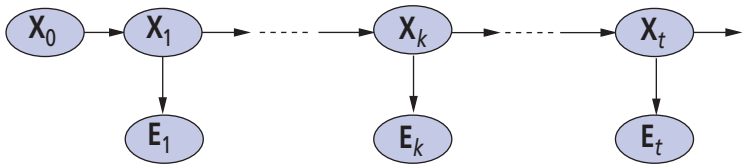


Abbildung 15.3: Die Glättung berechnet $\mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:t})$, die A-posteriori-Verteilung des Zustandes zu einer vergangenen Zeit k für eine vollständige Beobachtungsfolge von 1 bis t.

Jetzt sehen wir, dass die beiden Terme in Gleichung (15.8) durch Rekursionen durch die Zeit berechnet werden können, der eine vorwärts von 1 bis k und unter Verwendung der Filtergleichung (15.5) und der andere rückwärts von t bis $k + 1$ und unter Verwendung von Gleichung (15.7). Beachten Sie, dass die Rückwärtsphase mit $\mathbf{b}_{k+1:t} = \mathbf{P}(\mathbf{e}_{t+1:t} \mid \mathbf{X}_t) = \mathbf{P}(\ \mid \mathbf{X}_t)\mathbf{1}$ initialisiert wird, wobei $\mathbf{1}$ ein Vektor aus Einsen ist. (Weil $\mathbf{e}_{t+1:t}$ eine leere Folge ist, ist die Wahrscheinlichkeit ihrer Beobachtung gleich 1.)

Jetzt wenden wir diesen Algorithmus auf das Regenschirmbeispiel an und berechnen die geglättete Schätzung für die Regenwahrscheinlichkeit zum Zeitpunkt $k = 1$ für Schirmbeobachtungen an den Tagen 1 und 2. Aus Gleichung (15.8) erhalten wir dies durch:

$$\mathbf{P}(R_1 \mid u_1, u_2) = \alpha \mathbf{P}(R_1 \mid u_2) \mathbf{P}(u_2 \mid R_2). \tag{15.10}$$

Für den ersten Term wissen wir aus dem zuvor beschriebenen Vorwärtsfilterungsprozess bereits, dass er $\langle 0{,}818;\ 0{,}182 \rangle$ ist. Der zweite Term kann berechnet werden, indem die Rückwärtsrekursion in Gleichung (15.7) angewendet wird:

$$\begin{aligned}\mathbf{P}(u_2 \mid R_1) &= \sum_{r_2} P(u_2 \mid r_2) P(\ \mid r_2) \mathbf{P}(r_2 \mid R_1) \\ &= (0{,}9 \times 1 \times \langle 0{,}7;\ 0{,}3 \rangle) + (0{,}2 \times 1 \times \langle 0{,}3;\ 0{,}7 \rangle) = \langle 0{,}69;\ 0{,}41 \rangle.\end{aligned}$$

Wenn wir dies in Gleichung (15.10) einfügen, stellen wir fest, dass die geglättete Schätzung für Regen an Tag 1 wie folgt aussieht:

$$\mathbf{P}(R_1 \mid u_1, u_2) = \alpha\langle 0{,}818; 0{,}182\rangle \times \langle 0{,}69; 0{,}41\rangle \approx \langle 0{,}883; 0{,}177\rangle.$$

Die geglättete Schätzung ist also in diesem Fall *höher* als die gefilterte Schätzung (0,818). Das liegt daran, dass der Schirm an Tag 2 es wahrscheinlicher macht, dass es an Tag 2 geregnet hat; weil Regen wiederum zur Dauerhaftigkeit tendiert, wird es dadurch wahrscheinlicher, dass es an Tag 1 geregnet hat.

Sowohl die Vorwärts- als auch die Rückwärtsrekursion benötigen pro Schritt eine konstante Zeit; damit ist die Zeitkomplexität der Glättung im Hinblick auf die Evidenz $\mathbf{e}_{1:t}$ gleich $O(t)$. Das ist die Komplexität für die Glättung zu einem bestimmten Zeitschritt k. Wenn wir die gesamte Folge glätten wollen, ist eine offensichtliche Methode dafür, einfach den gesamten Glättungsprozess einmal für jeden zu glättenden Zeitschritt auszuführen. Das führt zu einer Zeitkomplexität von $O(t^2)$. Ein besserer Ansatz verwendet eine sehr einfache Anwendung der dynamischen Programmierung, um die Komplexität auf $O(t)$ zu reduzieren. Einen Hinweis finden Sie in der obigen Analyse des Regenschirmbeispiels, wo wir die Ergebnisse der Vorwärtsfilterungsphase wiederverwenden konnten. Der Schlüssel für den Algorithmus mit linearer Zeit ist, die *Ergebnisse* der Vorwärtsfilterung über die gesamte Folge *aufzuzeichnen*. Anschließend führen wir die Rückwärtsrekursion von t bis 1 aus und berechnen die geglättete Schätzung zu jedem Schritt k aus der berechneten Rückwärtsnachricht $\mathbf{b}_{k+1:t}$ und der gespeicherten Vorwärtsnachricht $\mathbf{f}_{1:k}$. Der Algorithmus, passend als **Vorwärts-Rückwärts-Algorithmus** bezeichnet, ist in ▶ Abbildung 15.4 gezeigt.

```
function FORWARD-BACKWARD(ev, prior) returns einen Vektor mit
         Wahrscheinlichkeitsverteilungen
  inputs: ev, ein Vektor mit Evidenzwerten für die Schritte 1,..., t
          prior, die A-priori-Verteilung im Ausgangszustand, P(X_0)
  local variables: fv, ein Vektor mit Vorwärtsnachrichten für die
                       Schritte 0,..., t
                   b, eine Darstellung der Rückwärtsnachricht,
                       anfangs alles Einsen
                  sv, ein Vektor von geglätteten Schätzungen für die
                       Schritte 1,..., t

  fv[0] ← prior
  for i = 1 to t do
    fv[i] ← FORWARD (fv[i - 1], ev[i])
  for i = t downto 1 do
    sv[i] ← NORMALIZE(fv[i] × b)
    b ← BACKWARD (b, ev[i])
  return sv
```

Abbildung 15.4: Der Vorwärts-Rückwärts-Algorithmus für die Berechnung bedingter Wahrscheinlichkeiten einer Folge von Zuständen für eine Folge von Beobachtungen. Die Operatoren FORWARD und BACKWARD sind in den Gleichungen (15.3) bzw. (15.7) definiert.

Der aufmerksame Leser hat bemerkt, dass es sich bei der in Abbildung 15.3 gezeigten Bayesschen Netzstruktur um einen **Polybaum** handelt, wie er in *Abschnitt 14.4.3* definiert wurde. Das bedeutet, dass eine einfache Anwendung des Clustering-Algorithmus auch einen Algorithmus mit linearer Zeit ergibt, der geglättete Schätzungen für die gesamte Folge berechnet. Jetzt ist nachvollziehbar, dass der Vorwärts-Rückwärts-Algorithmus wirk-

lich ein Spezialfall des Polybaum-Propagations-Algorithmus ist, der für Clustering-Methoden verwendet wird (obwohl die beiden unabhängig voneinander entwickelt wurden).

Der Vorwärts-Rückwärts-Algorithmus bildet das Rückgrat der Rechenmethoden, die in vielen Anwendungen eingesetzt werden, die sich mit Folgen verrauschter Beobachtungen beschäftigen. Wie bereits beschrieben, weist er zwei praktische Nachteile auf. Erstens ist die Speicherkomplexität möglicherweise zu hoch für Anwendungen mit großem Zustandsraum und langen Folgen. Er verwendet $O(|\mathbf{f}|t)$ Speicherplatz, wobei $|\mathbf{f}|$ die Repräsentationsgröße der Vorwärtsnachricht ist. Die Speicheranforderung kann auf $O(|\mathbf{f}| \log t)$ reduziert werden, mit gleichzeitigem Anstieg der Zeitkomplexität um den Faktor $\log t$, wie in Übung 15.3 gezeigt. In einigen Fällen (siehe *Abschnitt 15.3*) kann ein Algorithmus mit konstantem Speicher verwendet werden.

Zweitens muss der grundlegende Algorithmus abgeändert werden, wenn er in einer *Online*-Umgebung arbeiten soll, wo für frühere Zeitscheiben geglättete Schätzungen zu berechnen sind, weil am Ende der Folge ständig neue Beobachtungen hinzukommen. Die häufigste Forderung ist eine **Glättung mit fester Verzögerung**, wofür die geglättete Schätzung $\mathbf{P}(\mathbf{X}_{t-d} \mid \mathbf{e}_{1:t})$ für feste d berechnet werden muss. Die Glättung erfolgt also für die Zeitscheibe d Schritte nach der aktuellen Zeit t; wenn t wächst, muss die Glättung Schritt halten. Offensichtlich können wir den Vorwärts-Rückwärts-Algorithmus über das d-Schritt-„Fenster" ausführen, wenn eine neue Beobachtung hinzukommt, doch scheint das ineffizient zu sein. In *Abschnitt 15.3* werden wir sehen, dass die Glättung mit fester Verzögerung in einigen Fällen in konstanter Zeit für jede Aktualisierung stattfinden kann – unabhängig von der Verzögerung d.

15.2.3 Die wahrscheinlichste Folge finden

Angenommen, [*true*, *true*, *false*, *true*, *true*] ist die Regenschirmfolge für die ersten fünf Arbeitstage des Wachmannes. Welche Wetterfolge würde dies am wahrscheinlichsten erklären? Bedeutet das Fehlen des Regenschirmes an Tag 3, dass es nicht geregnet hat, oder hat der Direktor seinen Schirm einfach vergessen? Wenn es an Tag 3 nicht geregnet hat, hat es vielleicht (weil das Wetter dauerhaft ist) auch an Tag 4 nicht geregnet, aber der Direktor hat den Regenschirm für alle Fälle mitgebracht. Insgesamt gibt es 2^5 mögliche Wetterfolgen, die wir auswählen können. Gibt es eine Möglichkeit, die wahrscheinlichste davon herauszufinden, ohne sie alle auflisten zu müssen?

Wir könnten die folgende Prozedur mit linearer Zeit ausprobieren: durch Glättung die A-posteriori-Verteilung für das Wetter bei jedem Zeitschritt ermitteln, dann die Sequenz konstruieren, wobei für jeden Schritt das Wetter verwendet wird, das entsprechend der A-posteriori-Verteilung am wahrscheinlichsten ist. Diese Vorgehensweise sollte die Alarmglocken in Ihrem Kopf laut schrillen lassen, weil es sich bei den durch Glättung berechneten A-posteriori-Verteilungen um Verteilungen über *einzelne* Zeitschritte handelt, während wir für die Ermittlung der wahrscheinlichsten Folge die *gemeinsamen* Wahrscheinlichkeiten über alle Zeitschritte berücksichtigen müssen. Die Ergebnisse können sich wesentlich unterscheiden (siehe Übung 15.4).

Tipp

Es *gibt* einen Algorithmus mit linearer Zeit für die Ermittlung der wahrscheinlichsten Folge, aber dafür ist etwas mehr Denkarbeit erforderlich. Er basiert auf derselben Markov-Eigenschaft, die zu effizienten Algorithmen für das Filtern und Glätten geführt hat. Am einfachsten stellt man sich vor, dass jede Folge ein *Pfad* durch einen Graphen ist, dessen Knoten mögliche *Zustände* zu jedem Zeitschritt sind. ▶ Abbildung 15.5(a)

zeigt einen solchen Graphen für die Regenschirmwelt. Jetzt betrachten wir die Aufgabenstellung, den wahrscheinlichsten Pfad durch diesen Graphen zu finden, wobei die Wahrscheinlichkeit jedes Pfades gleich dem Produkt der Übergangswahrscheinlichkeiten entlang des Pfades und der Wahrscheinlichkeiten der bekannten Beobachtungen zu jedem Zustand ist. Wir konzentrieren uns insbesondere auf Pfade, die den Zustand $Regen_5 = true$ erreichen. Aus der Markov-Eigenschaft folgt, dass der wahrscheinlichste Pfad zum Zustand $Regen_5 = true$ aus dem wahrscheinlichsten Pfad zu *irgendeinem* Zustand zur Zeit 4 gefolgt von einem Übergang zu $Regen_5 = true$ besteht; und der Zustand zur Zeit 4, der zum Teil des Pfades zu $Regen_5 = true$ wird, ist der, der die Wahrscheinlichkeit dieses Pfades maximiert. Mit anderen Worten, *es besteht eine rekursive Beziehung zwischen den wahrscheinlichsten Pfaden zu jedem Zustand* $\mathbf{x}_{t+1}$ *und den wahrscheinlichsten Pfaden zu jedem Zustand* $\mathbf{x}_t$. Wir schreiben diese Beziehung als Gleichung, die die Wahrscheinlichkeiten des Pfades verbindet:

$$\max_{\mathbf{x}_1 \ldots \mathbf{x}_t} \mathbf{P}(\mathbf{x}_1, \ldots, \mathbf{x}_t, \mathbf{X}_{t+1} \mid \mathbf{e}_{1:t+1})$$

$$= \alpha \mathbf{P}(\mathbf{e}_{t+1} \mid \mathbf{X}_{t+1}) \max_{\mathbf{x}_t} \left(\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t) \max_{\mathbf{x}_1 \ldots \mathbf{x}_{t-1}} P(\mathbf{x}_1, \ldots, \mathbf{x}_{t-1}, \mathbf{x}_t \mid \mathbf{e}_{1:t}) \right). \tag{15.11}$$

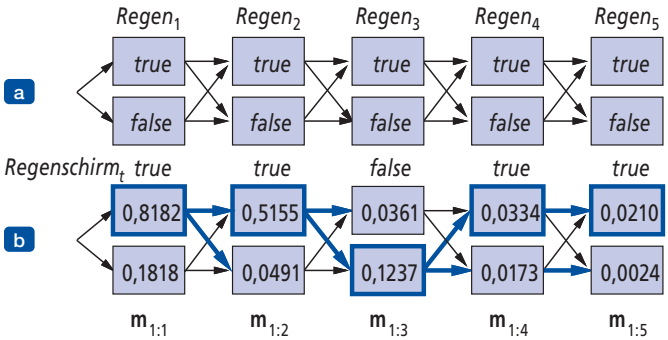


Abbildung 15.5: (a) Mögliche Zustandsfolgen für $Regen_t$ können als Pfade durch einen Graphen der möglichen Zustände zu jedem Zeitschritt betrachtet werden. (Zustände werden als Rechtecke dargestellt, um Verwechslungen mit den Knoten in einem Bayesschen Netz zu vermeiden.) (b) Arbeitsweise des Viterbi-Algorithmus für die Regenschirmbeobachtungsfolge [*true*, *true*, *false*, *true*, *true*]. Wir haben für jeden Zeitschritt t die Werte der Nachricht $\mathbf{m}_{1:t}$ gezeigt, die die Wahrscheinlichkeit der besten Folge angibt, die jeden Zustand in der Zeit t erreicht. Außerdem gibt für jeden Zustand der zu ihm führende fette Pfeil den besten Vorgänger an, wie er durch das Produkt der vorherigen Sequenzwahrscheinlichkeit und der Übergangswahrscheinlichkeit gemessen wird. Wenn man die fetten Pfeile von dem wahrscheinlichsten Zustand in $\mathbf{m}_{1:5}$ zurückverfolgt, erhält man die wahrscheinlichste Sequenz.

Gleichung (15.11) ist *identisch* mit der Filtergleichung (15.3), mit den folgenden Ausnahmen:

1 Die Vorwärtsnachricht $\mathbf{f}_{1:t} = \mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$ wird durch die folgende Nachricht ersetzt:

$$\mathbf{m}_{1:t} = \max_{\mathbf{x}_1 \ldots \mathbf{x}_{t-1}} \mathbf{P}(\mathbf{x}_1, \ldots, \mathbf{x}_{t-1}, \mathbf{X}_t \mid \mathbf{e}_{1:t}).$$

Das sind die Wahrscheinlichkeiten des wahrscheinlichsten Pfades zu jedem Zustand $\mathbf{x}_t$.

2 Die Summation über $\mathbf{x}_t$ in Gleichung (15.5) wird in Gleichung (15.11) ersetzt durch die Maximierung über $\mathbf{x}_t$.

Der Algorithmus für die Berechnung der wahrscheinlichsten Folge ist also dem Filtern ganz ähnlich: Er durchläuft die Folge vorwärts und berechnet zu jedem Zeitschritt die Nachricht **m** unter Verwendung von Gleichung (15.11). Die Vorgehensweise bei dieser Berechnung ist in ▶ Abbildung 15.5(b) gezeigt. Als Ergebnis hat sie die Wahrscheinlichkeit der wahrscheinlichsten Folge, die *jeden* der endgültigen Zustände erreicht. Man kann also einfach die wahrscheinlichste Folge auswählen (den fett ausgezeichneten Zustand). Um die tatsächliche Folge zu identifizieren und nicht nur ihre Wahrscheinlichkeit zu berechnen, muss der Algorithmus auch die Zeiger von jedem Zustand aus zurück zum besten Zustand verwalten, zu dem er führt (in Abbildung 15.5(b) durch fette Pfeile gekennzeichnet). Die optimale Folge wird identifiziert, indem diese fetten Pfeile vom besten Endzustand aus zurückverfolgt werden.

Damit haben wir den **Viterbi-Algorithmus** beschrieben, benannt nach seinem Erfinder Andrew J. Viterbi. Wie beim Filteralgorithmus ist seine Komplexität linear in t, der Länge der Folge. Anders als beim Filtern ist seine Speicheranforderung ebenfalls linear in t. Das liegt daran, dass der Viterbi-Algorithmus die Zeiger aufbewahren muss, die die beste Folge identifizieren, die zu jedem Zustand führt.

15.3 Hidden-Markov-Modelle

Im vorigen Abschnitt wurden Algorithmen für das temporale probabilistische Schließen entwickelt – unter Verwendung eines allgemeinen Netzes, das unabhängig von der spezifischen Form der Übergangs- und Sensormodelle war. In diesem und den beiden nächsten Abschnitten beschreiben wir konkretere Modelle und Anwendungen, die die Leistungsfähigkeit der grundlegenden Algorithmen demonstrieren und in einigen Fällen weitere Verbesserungen erlauben.

Wir beginnen mit dem **Hidden-Markov-Modell**, **HMM**. Ein HMM ist ein temporales probabilistisches Modell, in dem der Zustand des Prozesses durch eine *einzige diskrete* Zufallsvariable beschrieben wird. Die möglichen Werte der Variablen sind die möglichen Zustände der Welt. Das im vorigen Abschnitt beschriebene Regenschirmbeispiel ist also ein HMM, weil es nur eine Zustandsvariable verwendet: $Regen_t$. Wie sieht es mit einem Modell aus, das zwei oder mehr Zustandsvariablen hat? Es lässt sich dennoch in das HMM-Framework einpassen, wenn alle Variablen zu einer einzigen „Megavariablen“ kombiniert werden, deren Werte mögliche Wertetupel der einzelnen Zustandsvariablen sind. Wir werden sehen, dass die eingeschränkte Struktur des HMM eine sehr einfache und elegante Matriximplementierung aller grundlegenden Algorithmen erlaubt.[4]

15.3.1 Vereinfachte Matrixalgorithmen

Mit einer einzelnen, diskreten Variablen X_t können wir den Darstellungen des Übergangsmodells, des Sensormodells und der Vorwärts- und Rückwärtsnachrichten eine konkrete Form geben. Wir nehmen an, die Zustandsvariable X_t hat Werte, die durch die ganzen Zahlen 1, ..., S angegeben sind, wobei S die Anzahl der möglichen Zustände ist. Das Übergangsmodell $\mathbf{P}(X_t \mid X_{t-1})$ wird zu einer $S \times S$-Matrix $\mathbf{T}$, wobei gilt:

$$\mathbf{T}_{ij} = P(X_t = j \mid X_{t-1} = i).$$

4 Leser, die mit den grundlegenden Operationen für Vektoren und Matrizen nicht vertraut sind, sollten sich Anhang A ansehen, bevor sie mit diesem Abschnitt fortfahren.

Das bedeutet, $\mathbf{T}_{ij}$ ist die Wahrscheinlichkeit eines Überganges von Zustand i in Zustand j. Die Übergangsmatrix für die Regenschirmwelt ist beispielsweise:

$$\mathbf{T} = \mathbf{P}(X_t \mid X_{t-1}) = \begin{pmatrix} 0,7 & 0,3 \\ 0,3 & 0,7 \end{pmatrix}.$$

Wir bringen auch das Sensormodell in Matrixform. Weil wir in diesem Fall den Wert der Evidenzvariablen E_t zur Zeit t kennen (nennen wir ihn e_t), müssen wir nur – für jeden Zustand – spezifizieren, wie wahrscheinlich es ist, dass der Zustand ein Erscheinen von e_t bewirkt: Wir brauchen $P(e_t \mid X_t = i)$ für jeden Zustand i. Wegen mathematischer Einfachheit schreiben wir diese Werte in eine S × S-Diagonalmatrix $\mathbf{O}_t$, deren i-ter Diagonaleintrag $P(e_t \mid X_t = i)$ ist und deren andere Einträge 0 sind. Zum Beispiel ist an Tag 1 der Regenschirmwelt gemäß Abbildung 15.5 $U_1 = true$ und an Tag 3 ist $U_3 = false$, sodass wir mit den Werten von Abbildung 15.2 erhalten:

$$\mathbf{O}_1 = \begin{pmatrix} 0,9 & 0 \\ 0 & 0,2 \end{pmatrix}; \quad \mathbf{O}\hat{3} = \begin{pmatrix} 0,1 & 0 \\ 0 & 0,8 \end{pmatrix}.$$

Wenn wir jetzt die Spaltenvektoren verwenden, um die Vorwärts- und Rückwärtsnachrichten darzustellen, werden die Berechnungen zu einfachen Matrixvektor-Operationen. Die Vorwärtsgleichung (15.5) wird zu

$$\mathbf{f}_{1:t+1} = \alpha \mathbf{O}_{t+1} \mathbf{T}^\top \mathbf{f}_{1:t}. \tag{15.12}$$

Die Rückwärtsgleichung (15.9) wird zu

$$\mathbf{b}_{k+1:t} = \mathbf{T}\mathbf{O}_{k+1}\mathbf{b}_{k+2:t}. \tag{15.13}$$

Aus diesen Gleichungen erkennen wir, dass die Zeitkomplexität des Vorwärts-Rückwärts-Algorithmus (Abbildung 15.4) angewendet auf eine Folge der Länge t gleich $O(S^2 t)$ ist, weil für jeden Zeitschritt ein Vektor mit S Elementen mit einer $S \times S$ multipliziert werden muss. Die Speicheranforderung ist $O(St)$, weil der Vorwärtsdurchlauf t Vektoren der Größe S speichert.

Neben einer eleganten Beschreibung der Filter- und Glättungsalgorithmen für HMMs bietet die Matrixformulierung auch Gelegenheiten für verbesserte Algorithmen. Der erste ist eine einfache Variante des Vorwärts-Rückwärts-Algorithmus, die es erlaubt, dass die Glättung bei *konstantem* Speicheraufwand ausgeführt wird – unabhängig von der Länge der Folge. Die Idee dabei ist, dass die Glättung für jede Zeitscheibe k das gleichzeitige Vorliegen der Vorwärts- und Rückwärtsnachrichten $\mathbf{f}_{1:k}$ und $\mathbf{b}_{k+1:t}$ gemäß Gleichung (15.8) bedingt. Der Vorwärts-Rückwärts-Algorithmus realisiert dies, indem er die beim Vorwärtsdurchlauf gespeicherten **f**s speichert, sodass sie beim Rückwärtsdurchlauf zur Verfügung stehen. Eine weitere Möglichkeit ist, dies innerhalb eines einzigen Durchlaufes zu realisieren, der **f** und **b** in derselben Richtung weitergibt. Beispielsweise kann die „Vorwärts"-Nachricht **f** rückwärts weitergegeben werden, wenn wir Gleichung (15.12) so abändern, dass sie in die andere Richtung arbeitet:

$$\mathbf{f}_{1:t} = \alpha'(\mathbf{T}^\top)^{-1}\mathbf{O}_{t+1}^{-1}\mathbf{f}_{1:t+1}.$$

Der modifizierte Glättungsalgorithmus arbeitet, indem er zuerst den standardmäßigen Vorwärtsdurchlauf ausführt, um $\mathbf{f}_{t:t}$ zu berechnen (und dabei alle Zwischenergebnisse vergisst), und dann den Rückwärtsdurchlauf für **b** und **f** zusammen ausführt und sie ver-

wendet, um bei jedem Schritt die geglättete Schätzung zu berechnen. Weil nur eine Kopie jeder Nachricht benötigt wird, sind die Speicheranforderungen konstant (d.h. unabhängig von t, der Länge der Folge). Dieser Algorithmus weist eine entscheidende Einschränkung auf: Er fordert, dass die Übergangsmatrix umkehrbar ist und dass das Sensormodell keine Nullen enthält – d.h., jede Beobachtung ist in jedem Zustand möglich.

Ein zweiter Bereich, in dem die Matrixformulierung eine Verbesserung zulässt, ist die *Online*-Glättung mit fester Verzögerung. Die Tatsache, dass die Glättung bei konstantem Speicheraufwand möglich ist, lässt annehmen, dass es einen effizienten rekursiven Algorithmus für die Online-Glättung gibt – d.h. einen Algorithmus, dessen Zeitkomplexität unabhängig von der Verzögerungslänge ist. Angenommen, die Verzögerung ist gleich d; das bedeutet, wir glätten für die Zeitscheibe $t - d$, wobei die aktuelle Zeit gleich t ist. Aus Gleichung (15.8) berechnen wir

$$\alpha \mathbf{f}_{1:t-d} \mathbf{b}_{t-d+1:t}$$

für die Zeitscheibe $t - d$. Wenn eine neue Beobachtung eintrifft, müssen wir

$$\alpha \mathbf{f}_{1:t-d+1} \mathbf{b}_{t-d+2:t+1}$$

für Zeitscheibe $t - d + 1$ berechnen. Wie kann das inkrementell erfolgen? Zunächst können wir $\mathbf{f}_{1:t-d+1}$ unter Verwendung des Standardfilterprozesses (Gleichung (15.5)) aus $\mathbf{f}_{1:t-d}$ berechnen.

Die inkrementelle Berechnung der Rückwärtsnachricht ist komplizierter, weil es keine einfache Beziehung zwischen der alten Rückwärtsnachricht $\mathbf{b}_{t-d+1:t}$ und der neuen Rückwärtsnachricht $\mathbf{b}_{t-d+2:t+1}$ gibt. Stattdessen untersuchen wir die Beziehung zwischen der alten Nachricht $\mathbf{b}_{t-d+1:t}$ und der Rückwärtsnachricht am Anfang der Folge, $\mathbf{b}_{t+1:t}$. Dazu wenden wir Gleichung (15.13) d-mal an und erhalten:

$$\mathbf{b}_{t-d+1:t} = \left(\prod_{i=t-d+1}^{t} \mathbf{TO}_i \right) \mathbf{b}_{t+1:t} = \mathbf{B}_{t-d+1:t} \mathbf{1}. \tag{15.14}$$

Dabei ist die Matrix $\mathbf{B}_{t-d+1:t}$ das Produkt der Folge von **T**- und **O**-Matrizen. Man kann sich **B** als „Transformationsoperator“ vorstellen, der eine spätere Rückwärtsnachricht in eine frühere umwandelt. Eine ähnliche Gleichung gilt für die neuen Rückwärtsnachrichten, *nachdem* die neue Beobachtung eintrifft:

$$\mathbf{b}_{t-d+2:t+1} = \left(\prod_{i=t-d+2}^{t+1} \mathbf{TO}_i \right) \mathbf{b}_{t+2:t+1} = \mathbf{B}_{t-d+2:t+1} \mathbf{1}. \tag{15.15}$$

Ein genauer Blick auf die Produktausdrücke in den Gleichungen (15.14) und (15.15) zeigt, dass sie eine einfache Beziehung aufweisen: Um das zweite Produkt zu erhalten, „teilt“ man das erste Produkt durch das erste Element $\mathbf{TO}_{t-d+1}$ und multipliziert es dann mit dem neuen letzten Element $\mathbf{TO}_{t+1}$. In Matrixterminologie ausgedrückt gibt es eine einfache Beziehung zwischen der alten und der neuen **B**-Matrix:

$$\mathbf{B}_{t-d+2:t+1} = \mathbf{O}_{t-d+1}^{-1} \mathbf{T}^{-1} \mathbf{B}_{t-d+1:t} \mathbf{TO}_{t+1}. \tag{15.16}$$

Diese Gleichung bietet eine inkrementelle Aktualisierung für die **B**-Matrix, die es uns wiederum (durch Gleichung (15.15)) erlaubt, eine neue Rückwärtsnachricht $\mathbf{b}_{t-d+2:t+1}$ zu berechnen. Den vollständigen Algorithmus, für den **f** und **B** gespeichert und aktualisiert werden müssen, sehen Sie in ▶ Abbildung 15.6.

```
function FIXED-LAG-SMOOTHING(e_t, hmm, d) returns eine Verteilung über X_{t-d}
  inputs: e_t, die aktuelle Evidenz für Zeitschritt t
          hmm, ein Hidden-Markov-Modell mit S × S-Übergangsmatrix T
          d, die Länge der Verzögerung für die Glättung
  persistent: t, die aktuelle Zeit, anfangs 1
              f, die Vorwärtsnachricht P(X_t | e_{1:t}), anfangs hmm.PRIOR
              B, die d-Schritt-Rückwärtstransformationsmatrix, anfangs
                 die Einheitsmatrix
              e_{t-d:t}, doppelendige Liste der Evidenzen von t - d bis t,
                    anfangs leer
  local variables: O_{t-d}, O_t, Diagonalmatrizen mit Informationen
                                 des Sensormodells

  e_t an das Ende von e_{t-d:t} anfügen
  O_t ← Diagonalmatrix, die P(e_t | X_t) enthält
  if t > d then
    f ← FORWARD(f, e_t)
    e_{t-d-1} vom Anfang von e_{t-d:t} entfernen
    O_{t-d} ← Diagonalmatrix, die P(e_{t-d} | X_{t-d}) enthält
    B ← O^{-1}_{t-d} T^{-1} B T O_t
  else B ← B T O_t
  t ← t + 1
  if t > d then return NORMALIZE(f × B1) else return null
```

Abbildung 15.6: Ein Algorithmus für die Glättung mit einer feststehenden Zeitverzögerung von d Schritten, implementiert als Online-Algorithmus, der die neue geglättete Schätzung bei vorliegender Beobachtung für einen neuen Zeitschritt ausgibt. Beachten Sie, dass die endgültige Ausgabe NORMALIZE($\mathbf{f} \times \mathbf{B1}$) gemäß Gleichung (15.14) einfach $\alpha \mathbf{f} \times \mathbf{b}$ ist.

15.3.2 Beispiel für Hidden-Markov-Modell: Positionierung

In *Abschnitt 4.4.4* haben wir eine einfache Form des **Positionierungsproblems** für die Staubsaugerwelt eingeführt. In dieser Version verfügte der Roboter über eine einzige nichtdeterministische *Bewegen*-Aktion und seine Sensoren meldeten perfekt, ob Hindernisse unmittelbar in nördlicher, südlicher, östlicher oder westlicher Richtung liegen oder nicht; der Belief State des Roboters war die Menge der möglichen Positionen, an denen er sich befinden konnte.

Jetzt gestalten wir das Problem etwas realistischer: Wir binden ein einfaches Wahrscheinlichkeitsmodell für die Bewegung des Roboters ein und lassen Rauschen in den Sensoren zu. Die Zustandsvariable X_t repräsentiert die Position des Roboters auf dem diskreten Raster; die Domäne dieser Variablen ist die Menge der leeren Quadrate $\{s_1, ..., s_n\}$. Es sei NEIGHBORS(s) die Menge der leeren Quadrate, die an s angrenzen, und $N(s)$ die Größe dieser Menge. Das Übergangsmodell für die *Bewegen*-Aktion besagt dann, dass der Roboter gleichwahrscheinlich auf ein beliebiges benachbartes Quadrat gelangt:

$$P(X_{t+1} = j \mid X_t = i) = \mathbf{T}_{ij} = (1/N(i) \text{ if } j \in \text{NEIGHBORS}(i) \text{ else } 0).$$

Wir wissen nicht, wo der Roboter startet, und nehmen deshalb eine gleichförmige Verteilung über allen Quadraten an; d.h. $P(X_0 = i) = 1/n$. Für die konkrete Umgebung, die wir betrachten (▶ Abbildung 15.7), ist $n = 42$ und die Übergangsmatrix $\mathbf{T}$ besitzt $42 \times 42 = 1764$ Einträge.

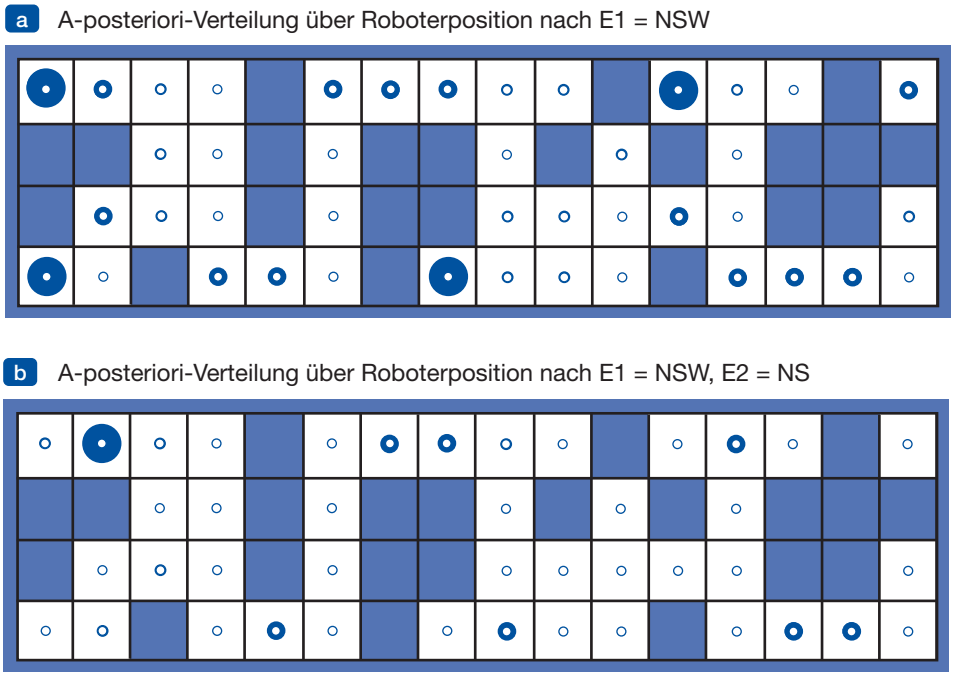


Abbildung 15.7: A-posteriori-Verteilung über Roboterposition: (a) Eine Beobachtung $E_1 = NSW$; (b) nach einer zweiten Beobachtung $E_2 = NS$. Die Größe einer Scheibe entspricht der Wahrscheinlichkeit, mit der sich der Roboter an dieser Position befindet. Die Fehlerrate der Sensoren beträgt $\varepsilon = 0{,}2$.

Die Sensorvariable E_t hat 16 mögliche Werte – jeweils eine 4-Bit-Folge, die die An- oder Abwesenheit eines Hindernisses in einer bestimmten Himmelsrichtung angibt. Wir verwenden zum Beispiel die Notation *NS* in der Bedeutung, dass die Nord- und Südsensoren ein Hindernis melden, die Ost- und Westsensoren nicht. Die Fehlerrate jedes Sensors sei ε und die Fehler sollen für die vier Sensorrichtungen unabhängig voneinander auftreten. In diesem Fall beträgt die Wahrscheinlichkeit, dass alle vier Bits richtig sind, $(1 - \varepsilon)^4$, und die Wahrscheinlichkeit, alle vier Bits falsch zu haben, ε^4. Wenn darüber hinaus d_{it} die Diskrepanz – die Anzahl der unterschiedlichen Bits – zwischen den wahren Werten für Quadrat i und den tatsächlich gelesenen e_t ist, berechnet sich die Wahrscheinlichkeit, dass ein Roboter in Quadrat i einen Sensorwert e_t empfängt, zu

$$P(E_t = e_t \mid X_t = i) = \mathbf{O}_{t_{ii}} = (1-\varepsilon)^{4-d_{it}} \varepsilon^{d_{it}}.$$

Zum Beispiel beträgt die Wahrscheinlichkeit, dass ein Quadrat mit Hindernissen im Norden und Süden einen Sensorwert NSO liefert, $(1-\varepsilon)^3\varepsilon^1$.

Für bekannte Matrizen T und Ot kann der Roboter nach Gleichung (15.12) die A-posteriori-Verteilung über Positionen berechnen – d.h. herausfinden, wo er sich befindet. Abbildung 15.7 zeigt die Verteilungen $\mathbf{P}(X_1 \mid E_1 = NSW)$ und $\mathbf{P}(X_2 \mid E_1 = NSW, E_2 = NS)$. Dies ist das gleiche Labyrinth, das Abbildung 4.18 (*Abschnitt 4.4.4*) zeigt, wobei wir dort aber logisches Filtern verwenden, um die möglichen Positionen zu finden, und perfekte Sensoren angenommen werden. Dieselben Positionen sind bei verrauschten Sensordaten immer noch am wahrscheinlichsten, doch besitzt nun jede Position eine Wahrscheinlichkeit ungleich null.

Der Roboter kann nicht nur durch Filtern seine aktuelle Position einschätzen, sondern auch durch Glättung (Gleichung (15.13)) herausfinden, wo er sich zu einem vergangenen Zeitpunkt befunden hat – zum Beispiel, wo er bei Zeit 0 begann –, und er kann mithilfe des Viterbi-Algorithmus den wahrscheinlichsten Pfad herausarbeiten, den er eingeschlagen hat, um zu seiner jetzigen Position zu gelangen. ▶ Abbildung 15.8 zeigt den Positionierungsfehler und die Genauigkeit des Viterbi-Pfades für verschiedene Werte der Sensorfehlerrate pro Bit ϵ. Selbst wenn ϵ 20% beträgt – was bedeutet, dass die gesamte Sensorerfassung in 59% der Zeit falsch ist –, kann der Roboter normalerweise seine Position innerhalb von zwei Quadraten nach 25 Beobachtungen herausfinden. Das hängt damit zusammen, dass der Algorithmus in der Lage ist, Evidenzen im Verlauf der Zeit zu integrieren und die vom Übergangsmodell hinsichtlich der Positionierungsfolge auferlegten probabilistischen Beschränkungen zu berücksichtigen. Wenn ϵ 10% beträgt, lässt sich die Leistung nach einem halben Dutzend Beobachtungen kaum von der Leistung bei perfekter Sensorerfassung unterscheiden.

In Übung 15.6 sollen Sie untersuchen, wie robust der HMM-Positionierungsalgorithmus gegenüber Fehlern in der A-priori-Verteilung $\mathbf{P}(X_0)$ und im Übergangsmodell selbst ist. Allgemein gesagt lässt sich ein hohes Niveau von Positionierungs- und Pfadgenauigkeit selbst angesichts beträchtlicher Fehler in den verwendeten Modellen aufrechterhalten.

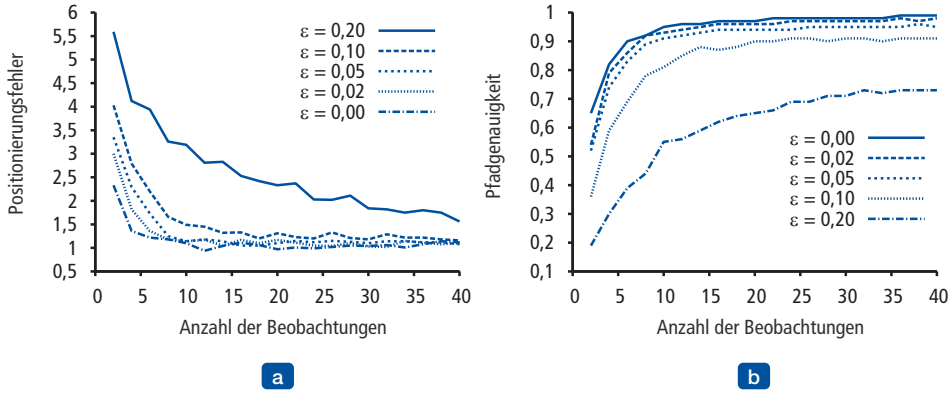


Abbildung 15.8: Leistung der HMM-Positionierung als Funktion der Länge der Beobachtungssequenz für verschiedene unterschiedliche Werte der Sensorfehlerwahrscheinlichkeit ϵ, wobei die Daten über 400 Läufe gemittelt sind. (a) Der Positionierungsfehler, definiert als Manhattan-Distanz von der wahren Position. (b) Die Viterbi-Pfadgenauigkeit, definiert als Anteil der korrekten Zustände auf dem Viterbi-Pfad.

Die Zustandsvariable für das in diesem Abschnitt betrachtete Beispiel ist ein physischer Ort in der Welt. Andere Probleme können natürlich auch andere Aspekte der Welt einschließen. In Übung 15.7 haben Sie die Aufgabe, eine Version des Staubsaugerroboters zu betrachten, bei der die Strategie darin besteht, solange wie möglich geradeaus zu gehen. Nur wenn der Roboter auf ein Hindernis trifft, schlägt er eine andere (zufällig ausgewählte) Richtung ein. Um diesen Roboter zu modellieren, besteht jeder Zustand im Modell aus einem Paar (*Position*, *Richtung*). Für die Umgebung in Abbildung 15.7, die 42 leere Felder aufweist, führt dies zu 168 Zuständen und einer Übergangsmatrix mit $168^2 = 28.224$ Einträgen – einer immer noch handhabbaren Größe. Wenn wir die Möglichkeit von Schmutz auf den Feldern hinzufügen, ist die Anzahl der Zustände mit 2^{42} zu multiplizieren und die Übergangsmatrix umfasst dann mehr als 10^{29} Einträge – was keine handhabbare Größe mehr ist. *Abschnitt 15.5* zeigt, wie sich mithilfe von

Bayesschen Netzen Domänen mit vielen Zustandsvariablen modellieren lassen. Und wenn wir zulassen, dass sich der Roboter stetig statt in einem diskreten Raster bewegt, wird die Anzahl der Zustände praktisch unendlich groß. Wie Sie diesen Fall in den Griff bekommen, erfahren Sie im nächsten Abschnitt.

15.4 Kalman-Filter

Stellen Sie sich vor, Sie beobachten, wie ein kleiner Vogel bei Dämmerung durch den dichten Urwald fliegt. Sie sehen immer wieder kurze, unterbrochene Bewegungsabschnitte; Sie versuchen angestrengt zu erraten, wo sich der Vogel gerade befindet und wo er als Nächstes auftaucht, um ihn nicht zu verlieren. Oder stellen Sie sich vor, Sie sind ein Radarposten im Zweiten Weltkrieg und beobachten einen kleinen wandernden Impuls, der alle zehn Sekunden auf dem Bildschirm erscheint. Oder gehen wir noch einen Schritt weiter: Stellen Sie sich vor, Sie sind Kepler, der versucht, die Bewegung der Planeten aus einer Sammlung höchst ungenauer Winkelbeobachtungen zu rekonstruieren, die in unregelmäßigen und ungenau gemessenen Intervallen erfolgt sind. In allen diesen Fällen führen Sie eine Filterung durch: Abschätzen der Zustandsvariablen (hier Position und Geschwindigkeit) aus verrauschten Beobachtungen über die Zeit. Wären die Variablen diskret, könnten wir das System mit einem Hidden-Markov-Modell modellieren. Dieser Abschnitt untersucht Methoden, um stetige Variablen zu verarbeiten. Hierfür wird ein als **Kalman-Filterung** bezeichneter Algorithmus verwendet, der nach einem seiner Erfinder Rudolf E. Kalman benannt ist.

Der Flug des Vogels ließe sich zu jedem Zeitpunkt durch sechs stetige Variablen angeben – drei für die Position (X_t, Y_t, Z_t) und drei für die Geschwindigkeit ($\dot{X}_t$, $\dot{Y}_t$, $\dot{Z}_t$). Wir brauchen geeignete bedingte Dichten, um die Übergangs- und Sensormodelle darzustellen; wie in *Kapitel 14* verwenden wir **lineare Gaußsche Verteilungen**. Das heißt, der nächste Zustand $\mathbf{X}_{t+1}$ muss eine lineare Funktion des aktuellen Zustandes $\mathbf{X}_t$ sein plus einem Gaußschen Rauschen – eine Bedingung, die sich in der Praxis als recht vernünftig erweisen wird. Betrachten Sie beispielsweise die X-Koordinate des Vogels und ignorieren Sie die anderen Koordinaten für den Moment. Wir bezeichnen das Zeitintervall zwischen den Beobachtungen mit Δ und nehmen im Intervall eine konstante Geschwindigkeit an; die Positionsaktualisierung ist dann durch

$$X_{t+\Delta} = X_t + \dot{X}\Delta$$

gegeben. Wenn wir ein Gaußsches Rauschen hinzufügen (um etwa Windänderungen etc. zu berücksichtigen), erhalten wir ein lineares Gaußsches Übergangsmodell:

$$P(X_{t+\Delta} = x_{t+\Delta} \mid X_t = x_t, \dot{X}_t = \dot{x}_t) = N(x_t + \dot{x}_t\Delta, \sigma^2)(x_{t+\Delta}).$$

▶ Abbildung 15.9 zeigt die Struktur eines Bayesschen Netzes für ein System mit der Positionsvektor $\mathbf{X}_t$ und der Geschwindigkeit $\dot{\mathbf{X}}_t$. Beachten Sie, dass dies eine sehr spezielle Form eines linearen Gaußschen Modells ist; die allgemeine Form wird später in diesem Abschnitt beschrieben und deckt neben den einfachen Bewegungsbeispielen aus dem ersten Absatz einen umfassenden Anwendungsbereich ab. Grundlegende Informationen über die mathematischen Eigenschaften der Gaußschen Verteilungen sind in Anhang A beschrieben; für unsere unmittelbare Aufgabenstellung ist es am wichtigsten, dass eine **multivariate Gaußsche** Verteilung für d Variablen durch eine d Elemente große Mittelwert-μ- und eine $d \times d$-Kovarianzmatrix Σ spezifiziert wird.

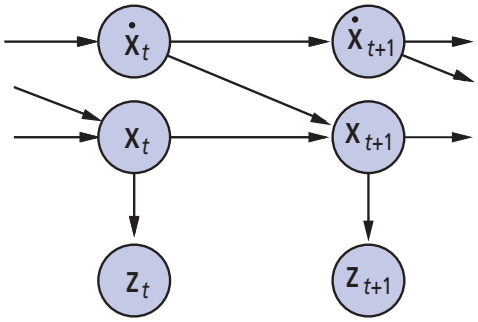


Abbildung 15.9: Bayessche Netzstruktur für ein lineares dynamisches System mit der Position $\mathbf{X}_t$, der Geschwindigkeit $\dot{\mathbf{X}}_t$ und der Positionsmessung $\mathbf{Z}_t$.

15.4.1 Gaußsche Verteilungen aktualisieren

In *Kapitel 14* haben wir auf eine Schlüsseleigenschaft der Familie der linearen Gaußschen Verteilungen hingewiesen: Unter den standardmäßigen Operationen im Bayesschen Netz bleibt sie geschlossen. Hier präzisieren wir diese Behauptung im Kontext der Filterung in einem temporalen Wahrscheinlichkeitsmodell. Die erforderlichen Eigenschaften entsprechen der zweiphasigen Filterberechnung in Gleichung (15.5):

1 Wenn die aktuelle Verteilung $\mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$ eine Gaußsche Verteilung ist und das Übergangsmodell $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t)$ eine lineare Gaußsche Verteilung, dann ist die einphasige vorhergesagte Verteilung gegeben durch

$$\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t}) = \int_{x_t} \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t) P(\mathbf{x}_t \mid \mathbf{e}_{1:t}) d\mathbf{x}_t \tag{15.17}$$

ebenfalls eine Gaußsche Verteilung.

2 Wenn die Vorhersage $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t})$ eine Gaußsche Verteilung ist und das Sensormodell $\mathbf{P}(\mathbf{e}_{t+1} \mid \mathbf{X}_{t+1})$ eine lineare Gaußsche Verteilung, dann ist nach Konditionierung auf die neue Evidenz die aktualisierte Verteilung

$$\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t+1}) = \alpha \mathbf{P}(\mathbf{e}_{t+1} \mid \mathbf{X}_{t+1}) \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t}) \tag{15.18}$$

ebenfalls eine Gaußsche Verteilung.

Der FORWARD-Operator für die Kalman-Filterung nimmt eine Gaußsche Vorwärtsnachricht $\mathbf{f}_{1:t}$ entgegen, die durch eine Mittelwert-$\boldsymbol{\mu}_t$- und eine Kovarianzmatrix Σ_t spezifiziert ist, und erzeugt eine neue multivariable Gaußsche Vorwärtsnachricht $\mathbf{f}_{1:t+1}$, die durch eine Mittelwert-$\boldsymbol{\mu}_{t+1}$- und eine Kovarianzmatrix Σ_{t+1} spezifiziert ist. Wenn wir also mit einer Gaußschen unbedingten Verteilung $\mathbf{f}_{1:0} = \mathbf{P}(\mathbf{X}_0) = N(\mu_0, \Sigma_0)$ beginnen, erzeugt die Filterung mit einem linearen Gaußschen Modell eine Gaußsche Zustandsverteilung für die gesamte Zeit.

Tipp

Das ist vielleicht ein ansehnliches, elegantes Ergebnis, aber warum ist es so wichtig? Der Grund dafür ist, dass bis auf ein paar Spezialfälle wie diesen *die Filterung mit stetigen oder hybriden (diskreten und stetigen) Netzen Zustandsverteilungen erzeugt, deren Repräsentation über die Zeit unbegrenzt anwächst*. Diese Aussage ist für den allgemeinen Fall nicht leicht zu beweisen, aber Übung 15.10 zeigt an einem einfachen Beispiel, was passiert.

15.4.2 Ein einfaches eindimensionales Beispiel

Wir haben gesagt, dass der FORWARD-Operator für den Kalman-Filter eine Gaußsche Verteilung auf eine neue Gaußsche Verteilung abbildet. Das bedeutet die Berechnung einer neuen Mittelwert- und Kovarianzmatrix aus den vorhergehenden Mittelwert- und Kovarianzmatrizen. Die Ableitung der Aktualisierungsregel für den allgemeinen (multivariablen) Fall bedingt eine Menge linearer Algebra; deshalb werden wir hier nur einen einfachen eindimensionalen Fall betrachten; später zeigen wir die Ergebnisse für den allgemeinen Fall. Selbst für den eindimensionalen Fall sind die Berechnungen etwas mühevoll, aber wir glauben, dass es sinnvoll ist, sie hier zu zeigen, weil der Nutzen des Kalman-Filters so eng mit den mathematischen Eigenschaften Gaußscher Verteilungen verknüpft ist.

Das temporale Modell, das wir hier betrachten wollen, beschreibt einen **zufälligen Weg** einer einzigen stetigen Zustandsvariablen X_t mit einer verrauschten Beobachtung Z_t. Ein Beispiel könnte der „Verbrauchervertrauensindex“ sein, der modelliert werden kann, indem man jeden Monat eine zufällige Gauß-verteilte Änderung darauf anwendet, und der durch eine zufällige Verbraucherbefragung gemessen wird, die ebenfalls eine Gaußsche Stichprobenverrauschung einführt. Die A-priori-Verteilung soll eine Gaußsche Verteilung mit der Varianz σ_0^2 sein:

$$P(x_0) = \alpha e^{-\frac{1}{2}\left(\frac{(x_0-\mu_0)^2}{\sigma_0^2}\right)}.$$

(Der Einfachheit halber verwenden wir dasselbe Symbol α für alle Normalisierungskonstanten in diesem Abschnitt.) Das Übergangsmodell fügt dem aktuellen Zustand einfach eine Gaußsche Störung konstanter Varianz σ_x^2 hinzu:

$$P(x_{1+t} \mid x_t) = \alpha e^{-\frac{1}{2}\left(\frac{(x_{t+1}-x_t)^2}{\sigma_x^2}\right)}.$$

Das Sensormodell nimmt dann ein Gaußsches Rauschen mit der Varianz σ_z^2 an:

$$P(z_t \mid x_t) = \alpha e^{-\frac{1}{2}\left(\frac{(z_t-x_t)^2}{\sigma_z^2}\right)}.$$

Mit der A-priori-Verteilung $\mathbf{P}(X_0)$ können wir jetzt unter Verwendung von Gleichung (15.17) die einphasige vorhergesagte Verteilung berechnen:

$$P(x_1) = \int_{-\infty}^{\infty} P(x_1 \mid x_0) P(x_0) dx_0 = \alpha \int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(\frac{(x_1-x_0)^2}{\sigma_x^2}\right)} e^{-\frac{1}{2}\left(\frac{(x_0-\mu_0)^2}{\sigma_0^2}\right)} dx_0$$

$$= \alpha \int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(\frac{\sigma_0^2(x_1-x_0)^2+\sigma_x^2(x_0-\mu_0)^2}{\sigma_0^2\sigma_x^2}\right)} dx_0.$$

Dieses Integral sieht sehr kompliziert aus. Der Schlüssel zum Erfolg ist, zu erkennen, dass der Exponent die Summe von zwei Ausdrücken ist, die *quadratisch* in x_0 sind, und dass er damit selbst quadratisch in x_0 ist. Ein einfacher Trick, auch als **Vervollständigen des Quadrates** bezeichnet, erlaubt das Umschreiben jedes Quadrates $ax_0^2 + bx_0 + c$ als die Summe eines quadrierten Terms

$$a(x_0 - \frac{-b}{2a})^2$$

und eines Restterms

$$c - \frac{b^2}{4a},$$

der unabhängig von x_0 ist. Der Restterm kann aus dem Integral herausgenommen werden, sodass wir Folgendes erhalten:

$$P(x_1) = \alpha e^{-\frac{1}{2}\left(c - \frac{b^2}{4a}\right)} \int_{-\infty}^{\infty} e^{-\frac{1}{2}(a(x_0 - \frac{-b}{2a})^2)} dx_0.$$

Jetzt ist das Integral nur noch das Integral einer Gaußschen Verteilung über ihren gesamten Bereich, was einfach 1 ist. Wir haben also nur noch den Restterm aus dem Quadrat. Wir stellen dann fest, dass der Restterm quadratisch in x_1 sein muss; tatsächlich erhalten wir nach der Vereinfachung:

$$P(x_1) = \alpha e^{-\frac{1}{2}\left(\frac{(x_1 - \mu_0)^2}{\sigma_0^2 + \sigma_x^2}\right)}.$$

Das bedeutet, die einphasige vorhergesagte Verteilung ist eine Gaußsche Verteilung mit demselben Mittelwert μ_0 und einer Varianz gleich der Summe der ursprünglichen Varianz σ_0^2 und der Übergangsvarianz σ_x^2. Wenn wir das einen Moment auf uns wirken lassen, erkennen wir, dass dies intuitiv sinnvoll ist.

Um den Aktualisierungsschritt abzuschließen, müssen wir auf die Beobachtung zum ersten Zeitschritt konditionieren, nämlich z_1. Aus Gleichung (15.18) erhalten wir dies durch:

$$P(x_1 \mid z_1) = \alpha P(z_1 \mid x_1) P(x_1)$$

$$= \alpha e^{-\frac{1}{2}\left(\frac{(z_1 - x_1)^2}{\sigma_z^2}\right)} e^{-\frac{1}{2}\left(\frac{(x_1 - \mu_0)^2}{\sigma_0^2 + \sigma_x^2}\right)}.$$

Auch jetzt kombinieren wir die Exponenten und vervollständigen das Quadrat (Übung 15.11) und erhalten:

$$P(x_1 \mid z_1) = \alpha e^{-\frac{1}{2}\left(\frac{(x_1 - \frac{(\sigma_0^2 + \sigma_x^2) z_1 + \sigma_z^2 \mu_0)^2}{\sigma_0^2 + \sigma_x^2 + \sigma_z^2}}{(\sigma_0^2 + \sigma_x^2)\sigma_z^2 / (\sigma_0^2 + \sigma_x^2 + \sigma_z^2)}\right)}. \tag{15.19}$$

Nach einem Aktualisierungszyklus haben wir also eine neue Gaußsche Verteilung für die Zustandsvariable.

Aus der Gaußschen Formel in Gleichung (15.19) erkennen wir, dass der neue Mittelwert und die neue Standardabweichung wie folgt aus dem alten Mittelwert und der alten Standardabweichung berechnet werden können:

$$\mu_{t+1} = \frac{(\sigma_t^2 + \sigma_x^2) z_{t+1} + \sigma_z^2 \mu_t}{\sigma_t^2 + \sigma_x^2 + \sigma_z^2} \quad \text{und} \quad \sigma_{t+1}^2 = \frac{(\sigma_t^2 + \sigma_x^2)\sigma_z^2}{\sigma_t^2 + \sigma_x^2 + \sigma_z^2}. \tag{15.20}$$

▶ Abbildung 15.10 zeigt einen Aktualisierungszyklus für einzelne Werte der Übergangs- und Sensormodelle.

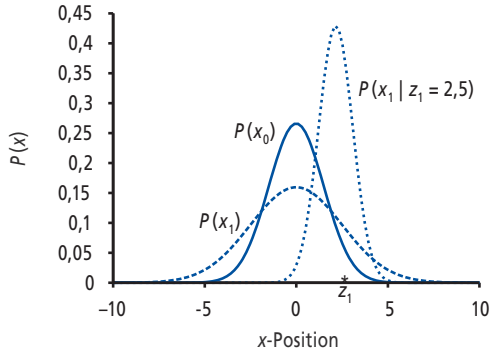


Abbildung 15.10: Phasen im Kalman-Filter-Aktualisierungszyklus für einen zufälligen Weg mit einer A-priori-Verteilung, angegeben durch $\mu_0 = 0{,}0$ und $\sigma_0 = 1{,}0$, ein Übergangsrauschen, angegeben durch $\sigma_x = 2{,}0$, ein Sensorrauschen, angegeben durch $\sigma_z = 1{,}0$, und eine erste Beobachtung $z_1 = 2{,}5$ (auf der x-Achse markiert). Beachten Sie, wie die Vorhersage $P(x_1)$ relativ zu $P(x_0)$ durch das Übergangsrauschen abgeflacht wird. Beachten Sie außerdem, dass der Mittelwert der A-posteriori-Verteilung $P(x_1|z_1)$ gegenüber der Beobachtung z_1 etwas nach links verschoben ist, weil der Mittelwert ein gewichteter Durchschnitt aus Vorhersage und Beobachtung ist.

Gleichung (15.20) spielt genau dieselbe Rolle wie die allgemeine Filtergleichung (15.5) oder die HMM-Filtergleichung (15.12). Aufgrund der speziellen Natur Gaußscher Verteilungen haben die Gleichungen jedoch einige interessante zusätzliche Eigenschaften. Erstens können wir die Berechnung des neuen Mittelwertes μ_{t+1} einfach als *gewichtetes Mittel* der neuen Beobachtung z_{t+1} und des alten Mittels μ_1 interpretieren. Wenn die Beobachtung unzuverlässig ist, dann ist σ_z^2 groß und wir legen mehr Gewichtung auf das alte Mittel; wenn das alte Mittel unzuverlässig ($\sigma_t{}^2$ ist groß) oder der Prozess äußerst unvorhersagbar ist (σ_x^2 ist groß), legen wir mehr Gewichtung auf die Beobachtung. Beachten Sie zweitens, dass die Aktualisierung für die Varianz $\sigma_{t+1}{}^2$ *unabhängig von der Beobachtung ist.* Wir können damit im Voraus berechnen, wie die Folge der Varianzwerte aussieht. Drittens konvergiert die Folge der Varianzwerte schnell auf einen festen Wert, der nur von σ_x^2 und σ_z^2 abhängig ist, wodurch die folgenden Gleichungen wesentlich vereinfacht werden (siehe Übung 15.12).

15.4.3 Der allgemeine Fall

Die obige Ableitung zeigt die Schlüsseleigenschaft Gaußscher Verteilungen, die es erst ermöglicht, dass die Kalman-Filter funktionieren: die Tatsache, dass der Exponent eine quadratische Form aufweist. Das gilt nicht nur für den eindimensionalen Fall; die vollständige multivariate Gaußsche Verteilung hat die Form:

$$N(\boldsymbol{\mu},\boldsymbol{\Sigma})(\mathbf{x}) = \alpha e^{-\frac{1}{2}((\mathbf{x}-\boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu}))}.$$

Das Ausmultiplizieren der Terme im Exponenten zeigt, dass der Exponent ebenfalls eine quadratische Funktion der Zufallsvariablen x_i in $\mathbf{x}$ ist. Wie im eindimensionalen Fall bewahrt die Filteraktualisierung die Gaußsche Natur der Zustandsverteilung.

Zunächst definieren wir das allgemeine temporale Modell, das bei der Kalman-Filterung eingesetzt wird. Sowohl das Übergangsmodell als auch das Sensormodell unterstützen eine *lineare* Transformation mit zusätzlichem Gaußschen Rauschen. Somit haben wir:

$$P(\mathbf{x}_{t+1} \mid \mathbf{x}_t) = N(\mathbf{F}\mathbf{x}_t, \Sigma_x)(\mathbf{x}_{t+1})$$
$$P(\mathbf{z}_t \mid \mathbf{x}_t) = N(\mathbf{H}\mathbf{x}_t, \Sigma_z)(\mathbf{z}_t). \quad (15.21)$$

Dabei sind $\mathbf{F}$ und Σ_x Matrizen, die das lineare Übergangsmodell und die Übergangsrauschkovarianz beschreiben, und $\mathbf{H}$ und Σ_z sind die entsprechenden Matrizen für das Sensormodell. Jetzt lauten die Aktualisierungsgleichungen für Mittel und Kovarianz in ihrer ganzen Schrecklichkeit:

$$\mu_{t+1} = \mathbf{F}\boldsymbol{\mu}_t + \mathbf{K}_{t+1}(\mathbf{z}_{t+1} - \mathbf{H}\mathbf{F}\boldsymbol{\mu}_t)$$
$$\Sigma_{t+1} = (\mathbf{I} - \mathbf{K}_{t+1}\mathbf{H})(\mathbf{F}\Sigma_t\mathbf{F}^\top + \Sigma_x). \quad (15.22)$$

Dabei wird $\mathbf{K}_{t+1} = (\mathbf{F}\Sigma_t\,\mathbf{F}^\top + \Sigma_x)\mathbf{H}^\top(\mathbf{H}(\mathbf{F}\Sigma_t\mathbf{F}^\top + \Sigma_x)\mathbf{H}^\top + \Sigma_z)^{-1}$ als **Kalman-Gewinnmatrix** bezeichnet. Ob Sie es glauben oder nicht, diese Gleichungen sind intuitiv sinnvoll. Betrachten Sie beispielsweise die Aktualisierung für die mittlere Zustandsschätzung $\boldsymbol{\mu}$. Der Term $\mathbf{F}\boldsymbol{\mu}_t$ ist der *vorhergesagte* Zustand für $t + 1$, deshalb ist $\mathbf{H}\mathbf{F}\boldsymbol{\mu}_t$ die *vorhergesagte* Beobachtung. Aus diesem Grund stellt der Term $\mathbf{z}_{t+1} - \mathbf{H}\mathbf{F}\boldsymbol{\mu}_t$ den Fehler der vorhergesagten Beobachtung dar. Dieser wird mit $\mathbf{K}_{t+1}$ multipliziert, um den vorhergesagten Zustand zu korrigieren. Damit ist $\mathbf{K}_{t+1}$ ein Maß dafür, *wie ernst die neue Beobachtung im Hinblick auf die Vorhersage genommen werden kann*. Wie in Gleichung (15.20) gilt auch hier die Eigenschaft, dass die Varianzaktualisierung unabhängig von den Beobachtungen ist. Die Folge der Werte für Σ_t und $\mathbf{K}_t$ kann deshalb offline berechnet werden und die tatsächlich während der Online-Bearbeitung erforderlichen Berechnungen sind relativ maßvoll.

Um diese Gleichungen in Aktion zu demonstrieren, haben wir sie auf das Problem angewendet, ein Objekt zu verfolgen, das sich in der X-Y-Ebene bewegt. Die Zustandsvariablen sind $\mathbf{X} = (X, Y, \dot{X}, \dot{Y})^\top$, deshalb sind $\mathbf{F}$, Σ_x, $\mathbf{H}$ und Σ_z 4×4-Matrizen. ▶ Abbildung 15.11(a) zeigt den tatsächlichen Bewegungsverlauf, eine Folge verrauschter Beobachtungen sowie den durch die Kalman-Filterung geschätzten Bewegungsverlauf, und zwar zusammen mit den durch die Konturen der Standardabweichung 1 angegebenen Kovarianzen. Der Filterprozess ist gut dafür geeignet, die tatsächliche Bewegung zu verfolgen, und die Varianz erreicht wie erwartet schnell einen Fixpunkt.

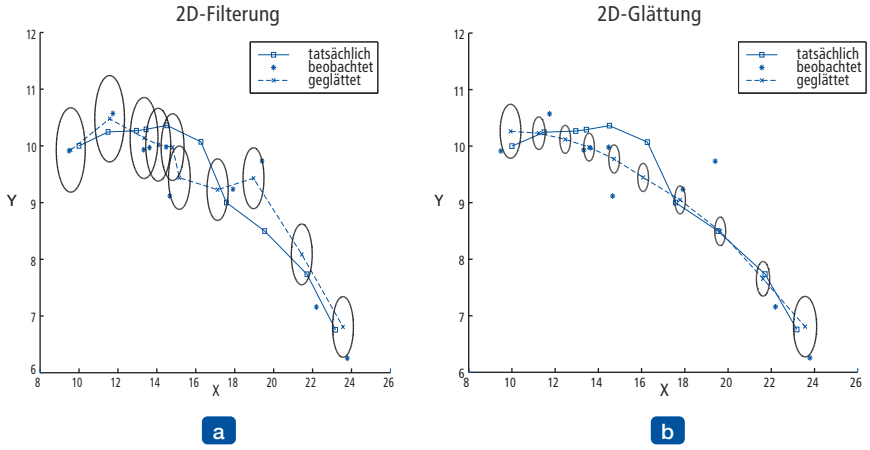


Abbildung 15.11: (a) Ergebnisse der Kalman-Filterung für ein Objekt, das sich in der X-Y-Ebene bewegt, wobei der tatsächliche Bewegungsverlauf (links nach rechts), eine Folge verrauschter Beobachtungen sowie der durch die Kalman-Filterung geschätzte Bewegungsverlauf gezeigt sind. Die Varianz in der Positionsschätzung wird durch die Ovale angezeigt. (b) Die Ergebnisse der Kalman-Glättung für dieselbe Beobachtungsfolge.

Wir können auch Gleichungen für die *Glättung* sowie für die Filterung mit linearen Gaußschen Modellen ableiten. Die Glättungsergebnisse sind in ▶ Abbildung 15.11(b) gezeigt. Beachten Sie, wie die Varianz in der Positionsschätzung scharf reduziert ist, außer an den Enden des Bewegungsverlaufes (warum?), und dass der geschätzte Bewegungsverlauf viel glatter ist.

15.4.4 Anwendbarkeit der Kalman-Filterung

Der Kalman-Filter und seine Weiterentwicklungen werden in den verschiedensten Anwendungen eingesetzt. Die „klassische" Anwendung ist die Radarverfolgung von Flugzeugen und Raketen. Verwandte Anwendungen sind unter anderem die akustische Überwachung von U-Booten und Bodenfahrzeugen sowie die visuelle Überwachung von Fahrzeugen und Menschen. Eine etwas exotischere Anwendung der Kalman-Filter finden wir für die Rekonstruktion von Partikelflugbahnen von Nebelkammer-Fotografien oder Meeresströmungen aus Satellitenoberflächen-Messungen. Der Anwendungsbereich umfasst mehr als nur die Beobachtung von Bewegung: Jedes System, das sich durch stetige Zustandsvariablen und verrauschte Messungen auszeichnet, kommt dafür infrage. Dabei kann es sich etwa um Papiermühlen, Chemiefabriken, Kernkraftwerke, Pflanzenökosysteme oder die Staatswirtschaft handeln.

Die Tatsache, dass die Kalman-Filterung auf ein System angewendet werden kann, bedeutet nicht gleichzeitig, dass die Ergebnisse gültig oder sinnvoll sind. Die getroffenen Annahmen – eine lineare Gaußsche Verteilung und Sensormodelle – sind sehr stark. Der **erweiterte Kalman-Filter** (**EKF**) versucht, Nichtlinearitäten in dem zu modellierenden System zu überwinden. Ein System ist **nichtlinear**, wenn das Übergangsmodell nicht als Matrixmultiplikation des Zustandsvektors beschrieben werden kann, wie in Gleichung (15.21). Der EKF modelliert das System als *lokal* linear in $\mathbf{x}_t$ im Bereich $\mathbf{x}_t = \mu_t$, dem Mittel der aktuellen Zustandsbeschreibung. Das funktioniert gut für geglättete, regelmäßige Systeme und erlaubt es dem Beobachter, eine Gaußsche Zustandsverteilung zu aktualisieren, bei der es sich um eine vernünftige Annäherung zur tatsächlichen bedingten Verteilung handelt. *Kapitel 25* gibt hierfür ein ausführliches Beispiel an.

Was bedeutet es, wenn ein System „nicht geglättet" oder „unregelmäßig" ist? Vom technischen Standpunkt aus betrachtet bedeutet es, dass es eine wesentliche Nichtlinearität in der Systemreaktion innerhalb des Bereiches gibt, der „nahe" (gemäß der Kovarianz $\mathbf{\Sigma}_t$) am aktuellen Mittel μ_t liegt. Um dieses Konzept besser zu verstehen, betrachten Sie das Beispiel, einen Vogel zu verfolgen, der durch den Dschungel fliegt. Der Vogel scheint mit hoher Geschwindigkeit direkt auf einen Baumstamm zuzufliegen. Der Kalman-Filter, egal ob Standard oder erweitert, kann nur eine Gaußsche Vorhersage der Vogelposition treffen, und das Mittel dieser Gaußschen Verteilung hat sein Zentrum im Baumstamm, wie in ▶ Abbildung 15.12(a) gezeigt. Ein vernünftiges Modell des Vogels dagegen würde ein Ausweichmanöver zur einen oder anderen Seite annehmen, wie in ▶ Abbildung 15.12(b) gezeigt. Ein solches Modell ist extrem nichtlinear, weil die Entscheidung des Vogels stark von seiner genauen Position relativ zum Baumstamm abhängig ist.

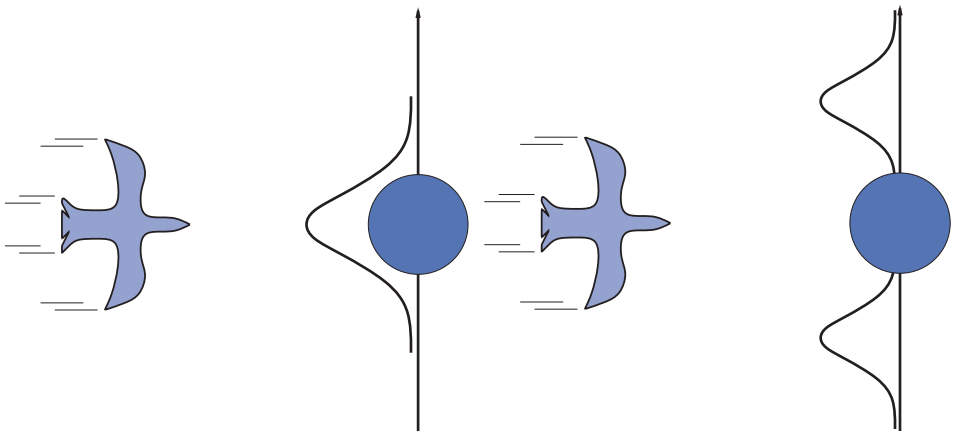


Abbildung 15.12: Ein Vogel, der auf einen Baum zufliegt (Ansicht von oben). (a) Ein Kalman-Filter sagt die Position des Vogels unter Verwendung einer einzigen Gaußschen Verteilung voraus, deren Zentrum sich im Ziel befindet. (b) Ein realistischeres Modell erlaubt die Berücksichtigung des Ausweichmanövers des Vogels und sagt voraus, dass er auf die eine oder auf die andere Seite fliegt.

Um Beispiele wie diese zu bearbeiten, brauchen wir offensichtlich eine ausdrucksstärkere Sprache, um das Verhalten des zu modellierenden Systems darzustellen. In der Steuerungstheorie, wo Probleme wie etwa die Ausweichmanöver von Flugzeugen dieselben Schwierigkeiten aufwerfen, ist die Standardlösung der **Switching-Kalman-Filter**. Bei diesem Ansatz werden mehrere Kalman-Filter parallel ausgeführt, die alle unterschiedliche Modelle des Systems verwenden – beispielsweise einer für einen geraden Flug, einer für scharfe Linkskurven und einer für scharfe Rechtskurven. Es wird eine gewichtete Summe von Vorhersagen verwendet, wobei die Gewichtung davon abhängt, wie gut jeder Filter mit den tatsächlichen Daten übereinstimmt. Im nächsten Abschnitt werden wir sehen, dass dies einfach ein Spezialfall des allgemeinen dynamischen Bayesschen Netzmodells ist, das man erhält, indem man dem Netz eine diskrete „Manöver"-Zustandsvariable hinzufügt, wie in Abbildung 15.9 gezeigt. Switching-Kalman-Filter werden in Übung 15.10 noch genauer angesprochen.

15.5 Dynamische Bayessche Netze

Ein **dynamisches Bayessches Netz**, **DBN**, ist ein Bayessches Netz, das ein temporales Wahrscheinlichkeitsmodell der in *Abschnitt 15.1* beschriebenen Art repräsentiert. Wir haben bereits Beispiele für DBNs kennengelernt: das Regenschirmnetz in Abbildung 15.2 und das Kalman-Filter-Netz in Abbildung 15.6. Im Allgemeinen kann jede Zeitscheibe eines DBN eine beliebige Anzahl an Zustandsvariablen $\mathbf{X}_t$ und Evidenzvariablen $\mathbf{E}_t$ haben. Der Einfachheit halber nehmen wir an, dass die Variablen und ihre Verknüpfungen von Zeitscheibe zu Zeitscheibe exakt repliziert werden und dass das DBN einen Markov-Prozess erster Stufe darstellt, sodass jede Variable nur Eltern in ihrer eigenen Zeitscheibe oder in der unmittelbar vorhergehenden Zeitscheibe haben kann.

Tipp

Es sollte klar sein, dass jedes Hidden-Markov-Modell als DBN mit einer einzigen Zustandsvariablen und einer einzigen Evidenzvariablen dargestellt werden kann. Außer-

dem gilt, dass jedes DBN mit diskreter Variable als HMM dargestellt werden kann; wie in *Abschnitt 15.3* beschrieben, können wir alle Zustandsvariablen im DBN zu einer einzigen Zustandsvariablen kombinieren, deren Werte alle möglichen Tupel der Werte der einzelnen Zustandsvariablen sind. Aber wenn jedes HMM ein DBN ist und jedes DBN in ein HMM übersetzt werden kann, wo liegt dann der Unterschied? Der Unterschied ist, *dass das DBN durch die Zerlegung des Zustandes eines komplexen Systems in seine einzelnen Variablen in der Lage ist, die* **dünne Besiedlung** *im temporalen Wahrscheinlichkeitsmodell zu nutzen*. Nehmen Sie beispielsweise an, ein DBN hat 20 boolesche Zustandsvariablen, die jeweils drei Eltern in der vorhergehenden Zeitscheibe haben. Das DBN-Übergangsmodell hat dann $20 \times 2^3 = 160$ Wahrscheinlichkeiten, während das entsprechende HMM 2^{20} Zustände und damit 2^{40}, also etwa eine Billion, Wahrscheinlichkeiten in der Übergangsmatrix aufweist. Das ist aus mindestens drei Gründen schlecht: Erstens, das eigentliche HMM benötigt sehr viel mehr Speicherplatz; zweitens, die riesige Übergangsmatrix macht die HMM-Inferenz teurer; und drittens, die Aufgabe, eine solche riesige Anzahl an Parametern zu überblicken, macht das reine HMM-Modell für große Probleme ungeeignet. Die Beziehung zwischen DBNs und HMMs ist in etwa vergleichbar mit der Beziehung zwischen normalen Bayesschen Netzen und vollständig tabellierten gemeinsamen Verteilungen.

Wir haben bereits erklärt, dass jedes Kalman-Filter-Modell in einem DBN mit stetigen Variablen und linearen Gaußschen bedingten Verteilungen dargestellt werden kann (Abbildung 15.9). Aus der Diskussion am Ende des letzten Abschnittes sollte klar sein, dass *nicht* jedes DBN als Kalman-Filter-Modell dargestellt werden kann. In einem Kalman-Filter ist die aktuelle Zustandsverteilung immer eine einzige multivariable Gaußsche Verteilung – d.h. ein einziger „Aufprall“ an einer bestimmten Position. DBNs dagegen können beliebige Verteilungen modellieren. Für viele Anwendungen aus der Praxis ist diese Flexibilität wesentlich. Betrachten Sie beispielsweise die aktuelle Position meiner Schlüssel. Sie könnten sich in meiner Tasche befinden, auf dem Tisch nebenan oder auf dem Küchenschrank liegen, an der Eingangstür hängen oder im Zündschloss des Autos stecken. Ein einziger Gaußscher „Aufprall“, der alle diese Positionen beinhaltet, müsste die hohe Wahrscheinlichkeit erbringen, dass die Schlüssel irgendwo im Flur in der Luft schweben. Aspekte der realen Welt, wie etwa zielgerichtete Agenten, Hindernisse und Taschen, führen „Nichtlinearitäten“ ein, für die eine Kombination aus diskreten und stetigen Variablen erforderlich ist, um sinnvolle Modelle zu erhalten.

15.5.1 DBNs erstellen

Für die Konstruktion eines DBN müssen drei Arten von Informationen angegeben werden: die A-priori-Verteilung über die Zustandsvariablen, $\mathbf{P}(\mathbf{X}_0)$; das Übergangsmodell $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{X}_t)$ und das Sensormodell $\mathbf{P}(\mathbf{E}_t|\mathbf{X}_t)$. Um die Übergangs- und Sensormodelle anzugeben, muss man auch die Topologie der Verbindungen zwischen aufeinanderfolgenden Zeitscheiben und zwischen Zustands- und Evidenzvariablen angeben. Weil man davon ausgeht, dass Übergangs- und Sensormodelle stationär sind – also für alle t gleich –, ist es am praktischsten, sie einfach nur für die erste Zeitscheibe zu spezifizieren. Beispielsweise ist die vollständige DBN-Spezifikation für die Regenschirmwelt durch das Dreiknoten-Netz in ▶ Abbildung 15.13(a) gegeben. Aus dieser Spezifikation kann das vollständige DBN mit einer unbeschränkten Anzahl von Zeitscheiben wie benötigt konstruiert werden, indem die erste Zeitscheibe kopiert wird.

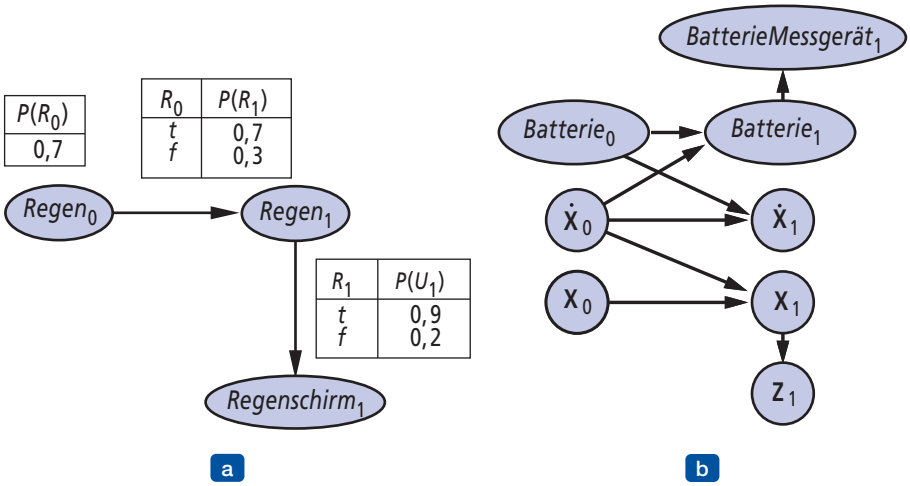


Abbildung 15.13: (a) Spezifikation der A-priori-Verteilung, des Übergangsmodells und des Sensormodells für das Regenschirm-DBN. Alle nachfolgenden Zeitscheiben werden als Kopien von Zeitscheibe 1 betrachtet. (b) Ein einfaches DBN für Roboterbewegung in der X-Y-Ebene.

Jetzt betrachten wir ein interessanteres Beispiel: die Beobachtung eines batteriebetriebenen Roboters, der sich in der X-Y-Ebene bewegt, wie in *Abschnitt 15.1* bereits angesprochen. Als Erstes brauchen wir Zustandsvariablen, die sowohl $\mathbf{X}_t = (X_t, Y_t)$ für die Position als auch $\dot{\mathbf{X}}_t = (\dot{X}_t, \dot{Y}_t)$ für die Geschwindigkeit beinhalten. Wir gehen davon aus, dass es irgendeine Möglichkeit gibt, die Position zu messen – vielleicht eine befestigte Kamera oder ein GPS (Global Positioning System) –, woraus wir die Messungen $\mathbf{Z}_t$ erhalten. Die Position zum nächsten Zeitschritt ist von der aktuellen Position und der Geschwindigkeit abhängig, so wie im standardmäßigen Kalman-Filter-Modell. Die Geschwindigkeit beim nächsten Schritt ist von der aktuellen Geschwindigkeit und dem Zustand der Batterie abhängig. Wir fügen $Batterie_t$ ein, um damit den aktuellen Ladezustand der Batterie zu repräsentieren, der als Eltern den vorhergehenden Ladezustand sowie die Geschwindigkeit hat, und wir fügen $BatterieMessgerät_t$ ein, das den Ladezustand der Batterie misst. Damit erhalten wir das in ▶ Abbildung 15.13(b) gezeigte grundlegende Modell.

Es ist angebracht, die Natur des Sensormodells für $BatterieMessgerät_t$ genauer zu betrachten. Der Einfachheit halber wollen wir annehmen, dass sowohl $Batterie_t$ als auch $BatterieMessgerät_t$ die diskreten Werte 0 bis 5 annehmen können. Wenn das Messgerät immer genau arbeitet, dann sollte die BWT $\mathbf{P}(BatterieMessgerät_t | Batterie_t)$ die Wahrscheinlichkeiten 1,0 „entlang der Diagonalen“ und Wahrscheinlichkeiten von 0,0 an jeder anderen Stelle haben. In der Realität sind die Messungen immer von Rauschen beeinflusst. Für stetige Messungen könnte stattdessen eine Gaußsche Verteilung mit kleiner Varianz verwendet werden.[5] Für unsere diskreten Variablen können wir eine Gaußsche Verteilung annähern, indem wir eine Verteilung verwenden, in der die Wahrscheinlichkeit eines Fehlers auf geeignete Weise abfällt, sodass die Wahrscheinlichkeit

5 Streng gesagt ist eine Gaußsche Verteilung problematisch, weil sie großen negativen Ladezuständen eine Wahrscheinlichkeit ungleich null zuweist. Die **Beta-Verteilung** ist manchmal eine bessere Wahl für eine Variable mit eingeschränktem Bereich.

eines großen Fehlers sehr klein ist. Wir verwenden den Begriff **Gaußsches Fehlermodell**, womit wir sowohl die stetige als auch die diskrete Version abdecken.

Jeder, der Praxiserfahrung in der Robotik, in der computergesteuerten Prozesssteuerung oder mit anderen Formen automatischer Sensorik besitzt, wird bestätigen, dass kleine Messstörungen häufig das geringste Problem sind. Echte Sensoren *fallen manchmal aus*. Wenn ein Sensor ausfällt, sendet er nicht notwendigerweise ein Signal, das sagt: „Oh, übrigens, die Daten, die ich hier sende, sind letztlich kompletter Unsinn." Stattdessen senden sie einfach den Unsinn. Die einfachste Art von Fehler wird auch als **transienter Fehler** bezeichnet, wobei der Sensor nur manchmal Unsinn sendet. Beispielsweise könnte der Ladezustandssensor die Gewohnheit haben, eine 0 zu senden, wenn jemand auf den Roboter schlägt, selbst wenn die Batterie vollständig geladen ist.

Betrachten wir, was passiert, wenn ein transienter Fehler bei einem Gaußschen Fehlermodell auftritt, das solche Fehler nicht kompensiert. Nehmen wir beispielsweise an, der Roboter sitzt einfach da und beobachtet 20 aufeinanderfolgende Batteriemessungen ab der fünften Messung. Das Batteriemessgerät hat dann eine temporale Aufzeichnung und die nächste Messung ist $BatterieMessgerät_{21} = 0$. Was vermittelt uns das einfache Gaußsche Fehlermodell über $BatterieMessgerät_{21}$? Gemäß der Bayesschen Regel ist die Antwort sowohl von dem Sensormodell $\mathbf{P}(BatterieMessgerät_{21} = 0 | Batterie_{21})$ als auch von der Vorhersage $\mathbf{P}(Batterie_{21} | BatterieMessgerät_{1:20})$ abhängig. Wenn die Wahrscheinlichkeit eines großen Sensorfehlers wesentlich geringer ist als die Wahrscheinlichkeit eines Überganges zu $Batterie_{21} = 0$ – selbst wenn Letzteres sehr unwahrscheinlich ist –, dann weist die A-posteriori-Verteilung dem Zustand, dass die Batterie leer ist, eine sehr hohe Wahrscheinlichkeit zu. Eine zweite Messung von 0 zum Zeitpunkt $t = 22$ macht diesen Schluss fast sicher. Wenn der transiente Fehler dann verschwindet und die Messung ab $t = 23$ den Wert 5 ergibt, kehrt die Schätzung wie von Zauberhand schnell wieder zu 5 zurück. Dieser Ereignisverlauf ist in der oberen Kurve von ▶ Abbildung 15.14(a) gezeigt, wo der erwartete Wert von $Batterie_t$ über die Zeit unter Verwendung eines diskreten Gaußschen Fehlermodells dargestellt ist.

Tipp

Trotz der Auflösung gibt es eine Zeit ($t = 22$), zu der der Roboter glaubt, dass seine Batterie leer sei; er sollte ein Notsignal senden und sich herunterfahren. Leider hat ihn das vereinfachte Sensormodell in die Irre geführt. Wie kann das korrigiert werden? Betrachten Sie ein vertrautes Beispiel aus unserem Fahralltag: In scharfen Kurven oder an steilen Bergen leuchtet manchmal die Treibstoffwarnleuchte auf. Statt nach dem ADAC zu rufen, erinnert man sich einfach, dass die Treibstoffmessung manchmal einen sehr großen Fehler zurückgibt, wenn der Treibstoff im Tank schwappt. Und die Moral von der Geschichte: *Wenn das System Sensorfehler korrekt verarbeiten soll, muss es die Möglichkeit von Fehlern berücksichtigen.*

Die einfachste Art eines Fehlermodells für einen Sensor erlaubt eine gewisse Wahrscheinlichkeit, dass der Sensor einen völlig falschen Wert zurückgibt, unabhängig vom tatsächlichen Zustand der Welt. Gibt das Batteriemessgerät etwa fälschlicherweise 0 zurück, könnten wir sagen, dass

$$P(BatterieMessgerät_t = 0 | Batterie_t = 5) = 0{,}03,$$

was sehr wahrscheinlich viel größer als die Wahrscheinlichkeit ist, die durch das einfache Gaußsche Fehlermodell zugeordnet wird. Wir wollen es als das **transiente Fehlermodell** bezeichnen. Wie kann es uns weiterhelfen, wenn wir einen Messwert von 0 erhalten?

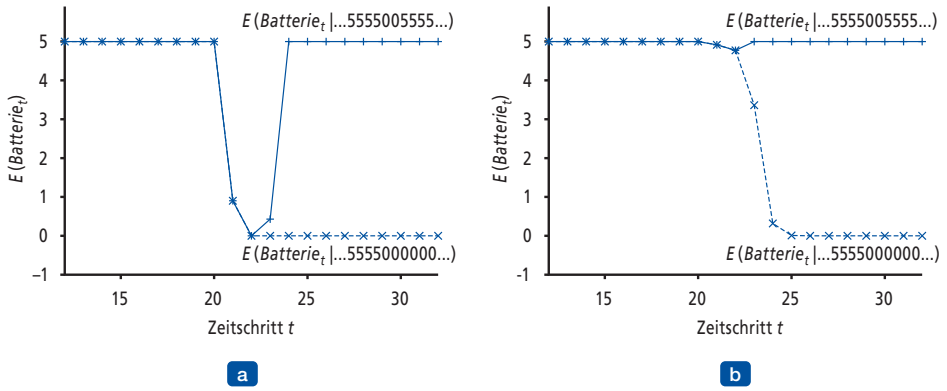


Abbildung 15.14: (a) Obere Kurve: Verlauf des erwarteten Wertes von $Batterie_t$ für eine Beobachtungsfolge, die nur aus 5en besteht, bis auf die 0en zu den Zeitpunkten $t = 21$ und $t = 22$, unter Verwendung eines einfachen Gaußschen Fehlermodells. Untere Kurve: Verlauf, wenn die Beobachtung von $t = 21$ an 0 bleibt. (b) Dasselbe Experiment mit transientem Fehlermodell. Beachten Sie, dass der transiente Fehler ebenfalls verarbeitet wird, der persistente Fehler jedoch zu einem verstärkten Pessimismus über die Batterieladung führt.

Vorausgesetzt, die *vorhergesagte* Wahrscheinlichkeit einer leeren Batterie ist den bisherigen Messungen entsprechend sehr viel kleiner als 0,03, dann ist die beste Erklärung der Beobachtung $BatterieMessgerät_{21} = 0$, dass der Sensor vorübergehend ausgefallen ist. Intuitiv können wir uns den Glauben über den Ladezustand der Batterie so vorstellen, dass er eine bestimmte „Trägheit“ hat, die hilft, vorübergehende Ausfälle bei der Messung zu kompensieren. Die obere Kurve in ▶ Abbildung 15.14(b) zeigt, dass das transiente Fehlermodell transiente Fehler ohne katastrophale Glaubensänderungen verarbeiten kann.

So viel zu den vorübergehenden Ausfällen. Was passiert bei einem persistenten (dauerhaften) Sensorausfall? Leider kommen solche Fehler sehr häufig vor. Wenn der Sensor 20 Messungen als 5 zurückgibt, gefolgt von 20 Messungen als 0, führt das im obigen Abschnitt beschriebene transiente Sensorfehlermodell den Roboter schrittweise zu dem Glauben, dass seine Batterie leer ist, während in Wirklichkeit das Messgerät ausgefallen ist. Die untere Kurve in Abbildung 15.14(b) zeigt den Glaubens-„Verlauf“ für diesen Fall. Bei $t = 25$ – fünf Messungen mit 0 – ist der Roboter davon überzeugt, dass seine Batterie leer ist. Natürlich wäre es uns lieber, wenn der Roboter dächte, das Batteriemessgerät sei kaputt – wenn dies wirklich das wahrscheinlichere Ereignis ist.

Es ist nicht besonders überraschend, dass wir für die Verarbeitung des persistenten Fehlers ein **persistentes Fehlermodell** brauchen, das beschreibt, wie sich der Sensor unter normalen Bedingungen und nach einem Ausfall verhält. Dazu müssen wir den Zustand des Systems mit einer zusätzlichen Variablen verstärken, z.B. *BMKaputt*, die den Zustand des Batteriemessgerätes beschreibt. Die Persistenz des Ausfalls muss unter Verwendung eines Pfeils modelliert werden, der $BMKaputt_0$ mit $BMKaputt_1$ verbindet. Dieser **Persistenzpfeil** hat eine BWT, die eine kleine Fehlerwahrscheinlichkeit für jeden beliebigen Zeitschritt vorgibt, etwa 0,001, aber auch besagt, dass der Sensor kaputt bleibt, nachdem er ausgefallen ist. Wenn der Sensor in Ordnung ist, ist das Sensormodell für *BatterieMessgerät* identisch mit dem transienten Fehlermodell; wenn der Sensor kaputt ist, besagt sie, dass *BatterieMessgerät* immer 0 ist, unabhängig von der tatsächlichen Batterieladung.

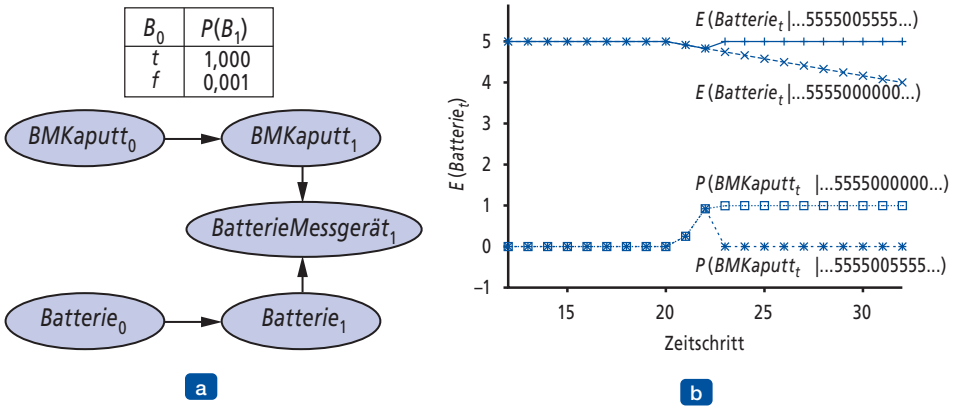


Abbildung 15.15: (a) Ein DBN-Ausschnitt, der die Sensorzustandsvariable zeigt, die für die Modellierung des persistenten Ausfalls des Batteriesensors benötigt wird. (b) Obere Kurven: Verläufe des erwarteten Wertes von $Batterie_t$ für die Beobachtungsfolgen „transienter Fehler" und „permanenter Fehler". Untere Kurven: Wahrscheinlichkeitsverläufe für *BMKaputt* für die beiden Beobachtungsfolgen.

▶ Abbildung 15.15(a) zeigt das persistente Fehlermodell für den Batteriesensor. ▶ Abbildung 15.15(b) zeigt seine Arbeit für die beiden Datenfolgen (temporärer Ausfall und persistenter Fehler). Für diese Kurven sind mehrere Dinge zu beachten. Erstens, bei dem temporären Ausfall steigt die Wahrscheinlichkeit, dass der Sensor kaputt ist, nach der zweiten Messung von 0 ganz erheblich, fällt aber sofort auf null zurück, nachdem eine 5 gemessen wurde. Zweitens, bei einem persistenten Fehler steigt die Wahrscheinlichkeit, dass der Sensor kaputt ist, schnell auf fast 1 und bleibt dort. Drittens, nachdem bekannt ist, dass der Sensor kaputt ist, kann der Roboter nur annehmen, dass sich die Batterie mit „normaler" Geschwindigkeit entlädt, wie durch den langsam sinkenden Pegel $E(Batterie_t|...)$ gezeigt.

Bisher haben wir nur die Oberfläche des Problems berührt, komplexe Prozesse darzustellen. Es gibt eine enorme Vielfalt an Übergangsmodellen, die sich über so unterschiedliche Themen wie etwa die Modellierung des menschlichen Endokrinsystems bis hin zur Modellierung mehrerer auf einer Autobahn fahrender Fahrzeuge erstreckt. Die Sensormodellierung wiederum ist ein riesiger Unterbereich, aber selbst subtile Phänomene, wie etwa Sensorabweichungen, plötzliche Dekalibrierung oder die Auswirkung von Außenbedingungen (wie beispielsweise das Wetter) auf Sensormessungen, können durch eine explizite Darstellung innerhalb dynamischer Bayesscher Netze verarbeitet werden.

15.5.2 Exakte Inferenz in DBNs

Nachdem wir einige Konzepte für die Darstellung komplexer Prozesse als DBNs vorgestellt haben, wenden wir uns jetzt der Frage der Inferenz zu. In gewisser Weise ist diese Frage bereits beantwortet: Dynamische Bayessche Netze *sind* Bayessche Netze und wir haben bereits Algorithmen für die Inferenz in Bayesschen Netzen. Man kann für eine Folge von Beobachtungen die vollständige Bayessche Netzdarstellung eines DBN erzeugen, indem man Zeitscheiben repliziert, bis das Netz groß genug ist, um mit diesen Beobachtungen zurechtzukommen, wie in ▶ Abbildung 15.16 gezeigt. Diese

Technik wird auch als **Aufrollen** bezeichnet. (Vom technischen Standpunkt aus betrachtet ist das DBN äquivalent mit dem semiunendlichen Netz, das man erhält, wenn man endlos aufrollt. Zeitscheiben, die hinter der letzten Beobachtung eingefügt werden, haben keine Auswirkung auf Inferenzen innerhalb des Beobachtungszeitraumes und können weggelassen werden.) Nachdem das DBN abgewickelt ist, können wir einen der in *Kapitel 14* beschriebenen Inferenzalgorithmen verwenden, wie z.B. Variableneliminierung, gemeinsamer Baum (Clustering-Methoden) usw.

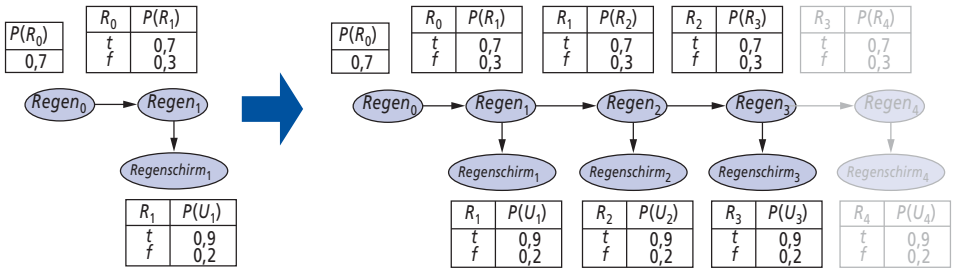


Abbildung 15.16: Aufrollen eines dynamischen Bayesschen Netzes: Zeitscheiben werden repliziert, um die Beobachtungsfolge $Regenschirm_{1:3}$ berücksichtigen zu können. Weitere Zeitscheiben haben keine Auswirkung auf Inferenzen innerhalb des Beobachtungszeitraumes.

Leider wäre eine naive Anwendung des Aufrollens nicht besonders effizient. Wenn wir eine Filterung oder Glättung mit einer langen Beobachtungsfolge $\mathbf{e}_{1:t}$ ausführen wollen, benötigt das abgewickelte Netz $O(t)$ Speicherplatz und wächst damit mit jeder neuen Beobachtung unbegrenzt. Wenn wir darüber hinaus bei jedem neuen Einfügen einer Beobachtung den Inferenzalgorithmus ausführen, steigt die Inferenzzeit pro Aktualisierung auch als $O(t)$.

Aus *Abschnitt 15.2.1* wissen wir, dass man konstante Zeit und konstanten Speicherplatz pro Filteraktualisierung realisieren kann, wenn die Berechnung auf rekursive Weise erfolgt. Im Wesentlichen arbeitet die Filteraktualisierung aus Gleichung (15.5), indem sie die Zustandsvariablen des vorhergehenden Zeitschrittes *aussummiert*, um die Verteilung für den neuen Zeitschritt zu erhalten. Die Aussummierung von Variablen ist genau das, was der Algorithmus der **Variableneliminierung** (siehe Abbildung 14.11) macht, und es stellt sich heraus, dass die Ausführung der Variableneliminierung mit Variablen in zeitlicher Reihenfolge genau die Arbeitsweise der rekursiven Filteraktualisierung in Gleichung (15.5) nachbildet. Der abgeänderte Algorithmus behält höchstens zwei Zeitscheiben gleichzeitig im Speicher; er beginnt mit Zeitscheibe 0, fügt Zeitscheibe 1 hinzu, summiert dann Zeitscheibe 0 aus und fügt Zeitscheibe 2 hinzu, summiert dann Zeitscheibe 1 aus usw. Auf diese Weise gelangen wir zu konstantem Speicher und konstanter Zeit pro Filteraktualisierung. (Dieselbe Leistung erhält man, wenn man geeignete Änderungen am Algorithmus gemeinsamer Bäume vornimmt.) In Übung 15.17 werden Sie diese Tatsache für das Regenschirmnetz überprüfen.

Soweit die guten Nachrichten – jetzt die schlechten: Es stellt sich heraus, dass die „Konstante“ für die Zeit pro Aktualisierung und die Speicherkomplexität in fast allen

Fällen exponentiell mit der Anzahl der Zustandsvariablen wächst. Wenn also die Variableneliminierung fortgesetzt wird, wachsen die Faktoren, um alle Zustandsvariablen aufzunehmen (genauer gesagt, all die Zustandsvariablen, die Eltern in der vorhergehenden Zeitscheibe haben). Die maximale Faktorgröße ist $O(d^{n+k})$ und die gesamten Aktualisierungskosten pro Schritt betragen $O(nd^{n+k})$, wobei d die Domänengröße der Variablen und k die maximale Anzahl der Eltern jeder Zustandsvariablen ist.

Tipp

Die Kosten liegen hier natürlich deutlich unter denen für die HMM-Aktualisierung, $O(d^{2n})$, sind aber für große Variablenmengen immer noch zu hoch. Diese unangenehme Tatsache ist schwer zu akzeptieren. Letztlich bedeutet das, *selbst wenn wir DBNs verwenden können, um sehr komplexe temporale Prozesse mit vielen dünn verbundenen Variablen zu repräsentieren, können wir nicht effizient und genau über diesen Prozess schließen*. Das eigentliche DBN-Modell, das die gemeinsame A-priori-Verteilung über alle Variablen darstellt, kann in seine BWTs zerlegt werden, aber die gemeinsame A-posteriori-Verteilung, die auf einer Beobachtungsfolge konditioniert – d.h. der Vorwärtsnachricht –, ist im Allgemeinen *nicht* zerlegbar. Bisher hat niemand eine Möglichkeit gefunden, dieses Problem zu umgehen, obwohl viele Bereiche aus Wissenschaft und Technik von einer solchen Lösung erheblich profitieren würden. Wir müssen also zu unseren Annäherungsmethoden zurückkehren.

15.5.3 Annähernde Inferenz in DBNs

Abschnitt 14.5 hat zwei Algorithmen für die Annäherung beschrieben: Wahrscheinlichkeitsgewichtung (siehe Abbildung 14.15) und MCMC (Markov Chain Monte Carlo, siehe Abbildung 14.16). Der erste dieser Algorithmen kann ganz einfach an den DBN-Kontext angepasst werden. Wir werden jedoch sehen, dass mehrere Verbesserungen gegenüber dem Standardalgorithmus für die Wahrscheinlichkeitsgewichtung erforderlich sind, wenn sich eine praktische Methode daraus entwickeln soll.

Tipp

Sie wissen, dass die Wahrscheinlichkeitsgewichtung funktioniert, indem Stichproben aus den Nichtevidenzknoten des Netzes in topologischer Reihenfolge gezogen werden, wobei jede Stichprobe mit der Wahrscheinlichkeit gewichtet wird, die sie gemäß der beobachteten Evidenzvariablen aufweist. Wie bei den exakten Algorithmen könnten wir die Wahrscheinlichkeitsgewichtung direkt auf ein abgewickeltes DBN anwenden, was aber unter den Problemen steigender Zeit- und Speicheranforderungen pro Aktualisierung mit wachsender Beobachtungsfolge leiden würde. Das Problem ist, dass der Standardalgorithmus jede Stichprobe nacheinander ausführt – durch das gesamte Netz. Stattdessen können wir einfach alle N Stichproben zusammen durch das DBN ausführen – jeweils eine Zeitscheibe nach der anderen. Der abgeänderte Algorithmus passt in das allgemeine Muster der Filteralgorithmen, mit der Menge von N Stichproben als Vorwärtsnachricht. Die erste wichtige Neuerung ist dann, *die eigentlichen Stichproben als annähernde Repräsentation der aktuellen Zustandsverteilung zu verwenden*. Das trifft die Forderung nach „konstanter“ Zeit pro Aktualisierung, obwohl die Konstante von der Anzahl der Stichproben abhängig ist, die erforderlich sind, um eine sinnvolle Annäherung zu bewahren. Es ist jedoch nicht erforderlich, das DBN abzuwickeln, weil wir im Speicher nur die aktuelle Zeitscheibe und die nächste Zeitscheibe brauchen.

In unserer Diskussion der Wahrscheinlichkeitsgewichtung in *Kapitel 14* haben wir dargelegt, dass die Genauigkeit des Algorithmus leidet, wenn sich die Evidenzvariablen „unterhalb" der Stichprobenvariablen befinden, weil in diesem Fall die Stichproben erzeugt werden, ohne dass sie von den Evidenzen beeinflusst werden. Wenn wir die typische Struktur eines DBN betrachten – z.B. das Regenschirm-DBN in Abbildung 15.16 –, sehen wir, dass die frühen Zustandsvariablen als Stichproben verwendet werden, ohne die Vorteile der späteren Evidenz zu nutzen. Wenn wir genauer hinschauen, erkennen wir, dass *keine* der Zustandsvariablen *irgendwelche* Evidenzvariablen unter ihren Vorfahren hat! Obwohl die Gewichtung jeder Stichprobe also von der Evidenz abhängig ist, ist die tatsächlich erzeugte Stichprobenmenge *völlig unabhängig* von der Evidenz. Selbst wenn also der Chef jeden Tag den Regenschirm mitbringt, könnte uns der Stichprobenprozess endlose Tage des Sonnenscheins vorspiegeln. Das bedeutet in der Praxis, dass der Teil der Stichproben, der sinnvoll nahe an der tatsächlichen Ereignisabfolge liegt (und demzufolge nicht vernachlässigbare Gewichte aufweist), exponentiell mit t, der Länge der Beobachtungsfolge, fällt. Mit anderen Worten: Um einen bestimmten Genauigkeitsgrad beizubehalten, müssen wir die Anzahl der Stichproben exponentiell mit t wachsen lassen. Weil ein Filteralgorithmus, der in Echtzeit arbeitet, nur eine bestimmte Anzahl an Stichproben verarbeiten kann, wird in der Praxis der Fehler nach einer sehr kleinen Anzahl an Aktualisierungsschritten sehr groß.

Tipp

Offensichtlich brauchen wir eine bessere Lösung. Die zweite wichtige Neuerung ist, *uns auf die Menge der Stichproben in Bereichen höchster Wahrscheinlichkeit im Zustandsraum zu konzentrieren*. Dazu können wir Stichproben verwerfen, die gemäß den Beobachtungen eine sehr geringe Gewichtung aufweisen, während wir diejenigen vermehren, die eine hohe Gewichtung haben. Auf diese Weise bleibt die Stichprobenpopulation sinnvoll nahe an der Realität. Wenn wir uns Stichproben als Ressource für die Modellierung der bedingten Verteilung vorstellen, erscheint es uns sinnvoll, mehr Stichproben in Bereichen des Zustandsraumes zu verwenden, wo die bedingte Verteilung höher ist.

Für genau diesen Zweck gibt es eine ganze Algorithmenfamilie, die sogenannten **Partikelfilter**. Das Partikelfiltern funktioniert wie folgt: Zuerst wird eine Population aus N Stichproben erzeugt, indem Stichproben aus der A-priori-Verteilung $\mathbf{P}(\mathbf{X}_0)$ zur Zeit 0 erzeugt werden. Dann wird der Aktualisierungszyklus für jeden Zeitschritt wiederholt:

1. Jede Stichprobe wird vorwärts weitergegeben, indem der nächste Zustandswert $\mathbf{x}_{t+1}$ als Stichprobe ermittelt wird, wobei der aktuelle Wert $\mathbf{x}_t$ für die Stichprobe bekannt ist und das Übergangsmodell $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{x}_t)$ verwendet wird.
2. Jede Stichprobe wird mit der Wahrscheinlichkeit gewichtet, die sie der neuen Evidenz zuweist, $P(\mathbf{e}_{t+1}|\mathbf{x}_{t+1})$.
3. Die Population wird *neu gesampelt*, um eine neue Population aus N Stichproben zu erhalten. Jede neue Stichprobe wird aus der aktuellen Population ausgewählt; die Wahrscheinlichkeit, dass eine bestimmte Stichprobe ausgewählt wird, ist proportional zu ihrer Gewichtung. Die neuen Stichproben weisen keine Gewichtung auf.

Der Algorithmus ist in ▶ Abbildung 15.17 gezeigt und in ▶ Abbildung 15.18 sehen wir seine Arbeitsweise für das Regenschirm-DBN.

```
function PARTICLE-FILTERING(e, N, dbn) returns eine Stichprobenmenge für
         den nächsten Zeitschritt
   inputs: e, die neu ankommende Evidenz
           N, die Anzahl der beizubehaltenden Stichproben
           dbn, ein DBN mit unbedingtem P(X_0), Übergangsmodell P(X_1|X_0)
                und Sensormodell P(E_1|X_1)
   persistent: S, ein Vektor mit Stichproben der Größe N, anfänglich aus
               P(X_0) erzeugt
   local variables: W, ein Vektor mit Gewichtungen der Größe N

   for i = 1 to N do
      S[i] ← Stichprobe aus P(X_1|X_0 = S[i])    /* Schritt 1 */
      W[i] ← P(e|X_1 = S[i])                     /* Schritt 2 */
   S ← WEIGHTED-SAMPLE-WITH-REPLACEMENT(N, S, W) /* Schritt 3 */
   return S
```

Abbildung 15.17: Der Partikelfilter-Algorithmus, implementiert als rekursive Aktualisierungsoperation mit Zustand (die Menge der Stichproben). Jeder der Stichprobenschritte beinhaltet ein Sampling der relevanten Zeitscheibenvariablen in topologischer Reihenfolge, ähnlich wie in PRIOR-SAMPLE. Die Operation WEIGHTED-SAMPLE-WITH-REPLACEMENT kann so implementiert werden, dass sie in *O(N)* erwarteter Zeit ausgeführt wird. Die Schrittnummern beziehen sich auf die Beschreibung im Text.

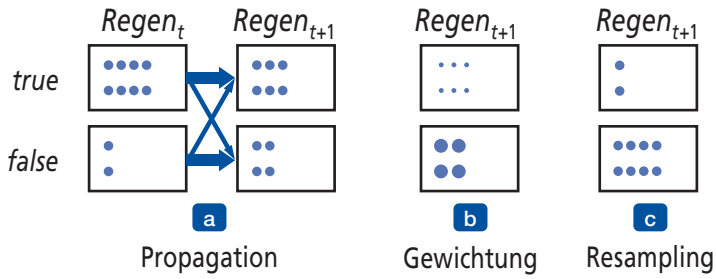


Abbildung 15.18: Der Partikelfilter-Aktualisierungszyklus für das Regenschirm-DBN mit $N = 10$, der die Stichprobenpopulation jedes Zustandes zeigt. (a) Zur Zeit t geben acht Stichproben $Regen$ und zwei $\neg Regen$ an. Jede wird weitergegeben, indem der nächste Zustand über das Übergangsmodell gesampelt wird. Zur Zeit $t + 1$ geben sechs Stichproben $Regen$ und vier $\neg Regen$ an. (b) $\neg Regenschirm$ wird zum Zeitpunkt $t + 1$ beobachtet. Jede Stichprobe wird nach ihrer Wahrscheinlichkeit der Beobachtung gewichtet, wie durch die Größe der Kreise angezeigt. (c) Eine neue Menge von zehn Stichproben wird durch eine gewichtete Zufallsauswahl aus der aktuellen Menge erzeugt, wodurch zwei Stichproben entstehen, die $Regen$ anzeigen, und acht, die $\neg Regen$ anzeigen.

Wir können zeigen, dass der Algorithmus konsistent ist – dass er die korrekten Wahrscheinlichkeiten zurückgibt, wenn N gegen unendlich geht –, indem wir betrachten, was während eines Aktualisierungszyklus passiert. Wir nehmen an, dass die Stichprobenpopulation mit einer korrekten Repräsentation der Vorwärtsnachricht $\mathbf{f}_{1:t} = \mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t})$ zur Zeit t beginnt. Wenn wir $N(\mathbf{x}_t|\mathbf{e}_{1:t})$ für die Anzahl der Stichproben schreiben, die den Zustand $\mathbf{x}_t$ belegen, nachdem Beobachtungen $\mathbf{e}_{1:t}$ verarbeitet wurden, haben wir

$$N(\mathbf{x}_t|\mathbf{e}_{1:t})/N = P(\mathbf{x}_t|\ \mathbf{e}_{1:t}) \tag{15.23}$$

für große N. Jetzt geben wir jede Stichprobe weiter, indem wir die Zustandsvariablen zur Zeit $t + 1$ sampeln, wobei die Werte für die Stichprobe zur Zeit t bekannt sind. Die Anzahl der Stichproben, die den Zustand $\mathbf{x}_{t+1}$ aus jedem $\mathbf{x}_t$ erreichen, ist die Übergangswahrscheinlichkeit multipliziert mit der Population von $\mathbf{x}_t$; damit ist die Gesamtzahl der Stichproben, die $\mathbf{x}_{t+1}$ erreichen, gleich

$$N(\mathbf{x}_{t+1} \mid \mathbf{e}_{1:t}) = \sum_{x_t} P(\mathbf{x}_{t+1} \mid \mathbf{x}_t) N(x_t \mid \mathbf{e}_{1:t}).$$

Jetzt gewichten wir jede Stichprobe nach ihrer Wahrscheinlichkeit für die Evidenz zur Zeit $t + 1$. Eine Stichprobe im Zustand $\mathbf{x}_{t+1}$ erhält die Gewichtung $P(\mathbf{e}_{t+1} | \mathbf{x}_{t+1})$. Die Gesamtgewichtung der Stichproben in $\mathbf{x}_{t+1}$ nach dem Auftreten von $\mathbf{e}_{t+1}$ ist deshalb:

$$W(\mathbf{x}_{t+1} | \mathbf{e}_{1:t+1}) = P(\mathbf{e}_{t+1} | \mathbf{x}_{t+1}) N(\mathbf{x}_{t+1} | \mathbf{e}_{1:t}).$$

Jetzt zum Resampling-Schritt. Weil jede Stichprobe mit einer Wahrscheinlichkeit repliziert wird, die proportional zu ihrer Gewichtung ist, ist die Anzahl der Stichproben im Zustand $\mathbf{x}_{t+1}$ nach dem Resampling proportional zur Gesamtgewichtung in $\mathbf{x}_{t+1}$ vor dem Resampling:

$$\begin{aligned}
N(\mathbf{x}_{t+1} \mid \mathbf{e}_{1:t+1}) / N &= \alpha W(\mathbf{x}_{t+1} \mid \mathbf{e}_{1:t+1}) \\
&= \alpha P(e_{t+1} \mid \mathbf{x}_{t+1}) N(\mathbf{x}_{t+1} \mid \mathbf{e}_{1:t}) \\
&= \alpha P(\mathbf{e}_{t+1} \mid \mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1} \mid \mathbf{x}_t) N(\mathbf{x}_t \mid \mathbf{e}_{1:t}) \\
&= \alpha N P(\mathbf{e}_{t+1} \mid \mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1} \mid \mathbf{x}_t) P(\mathbf{x}_t \mid \mathbf{e}_{1:t}) \qquad \text{(aus 15.21)} \\
&= \alpha' P(\mathbf{e}_{t+1} \mid \mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1} \mid \mathbf{x}_t) P(\mathbf{x}_t \mid \mathbf{e}_{1:t}) \\
&= P(\mathbf{x}_{t+1} \mid \mathbf{e}_{1:t+1}) \qquad \text{(nach 15.5)}.
\end{aligned}$$

Aus diesem Grund repräsentiert die Stichprobenpopulation nach einem Aktualisierungszyklus die Vorwärtsnachricht zur Zeit $t + 1$ korrekt.

Die Partikelfilterung ist also *konsistent*, aber ist sie auch *effizient*? In der Praxis scheint das der Fall zu sein: Das Filtern scheint eine gute Annäherung zur tatsächlichen A-posteriori-Verteilung unter Verwendung einer konstanten Anzahl an Stichproben zu sein. Unter bestimmten Annahmen – insbesondere, dass die Wahrscheinlichkeiten in den Übergangs- und Sensormodellen streng größer 0 und kleiner als 1 sind – lässt sich beweisen, dass die Annäherung einen begrenzten Fehler mit hoher Wahrscheinlichkeit sicherstellt. Auf der praktischen Seite ist das Spektrum der Anwendungen gewachsen, um viele Bereiche von Wissenschaft und Technik einzuschließen. Einige Verweise werden am Ende dieses Kapitels angegeben.

15.6 Verfolgen mehrerer Objekte

Die vorherigen Abschnitte haben – ohne es zu erwähnen – Zustandsabschätzungsprobleme mit einem einzelnen Objekt betrachtet. In diesem Abschnitt sehen wir uns nun an, was passiert, wenn die Beobachtungen von zwei oder mehr Objekten erzeugt werden. Dieser Fall unterscheidet sich von der guten alten Zustandsschätzung darin, dass es jetzt die Möglichkeit der Unbestimmtheit gibt, welches Objekt welche Beobachtung erzeugt hat. Dieses sogenannte Problem der **Identitätsunsicherheit** von *Abschnitt 14.6.3* wird nun in einem temporalen Kontext betrachtet. In der Literatur zur Kontrolltheorie spricht man vom Problem der **Datenzuordnung** – d.h. dem Problem, die Beobachtungsdaten den Objekten zuzuordnen, die sie erzeugt haben.

Ursprünglich wurde das Datenzuordnungsproblem im Kontext der Radarverfolgung untersucht, bei der reflektierte Impulse in festen Zeitabschnitten durch eine sich dre-

hende Radarantenne empfangen werden. In jedem Zeitschritt können mehrere Leuchtpunkte auf dem Bildschirm erscheinen, doch gibt es keine direkte Beobachtung, welche Leuchtpunkte zur Zeit t zu welchen Leuchtpunkten zur Zeit $t-1$ gehören. ▶ Abbildung 15.19(a) zeigt ein einfaches Beispiel mit zwei Leuchtpunkten pro Zeitschritt für fünf Zeitschritte. Wir nehmen die beiden Leuchtpunktpositionen zur Zeit t mit e_t^1 und e_t^2 an. (Die Beschriftung der Leuchtpunkte in einem Zeitschritt als „1“ und „2“ geschieht vollkommen willkürlich und vermittelt keine Informationen.) Nehmen wir für den Moment an, dass genau zwei Flugzeuge A und B die Leuchtpunkte hervorgerufen haben; ihre wahren Positionen sind X_t^A und X_t^B. Zur Vereinfachung nehmen wir weiterhin an, dass sich jedes Flugzeug unabhängig entsprechend einem bekannten Übergangsmodell – zum Beispiel einem linearen Gaußschen Modell, wie es im Kalman-Filter verwendet wird (*Abschnitt 15.4*) – bewegt.

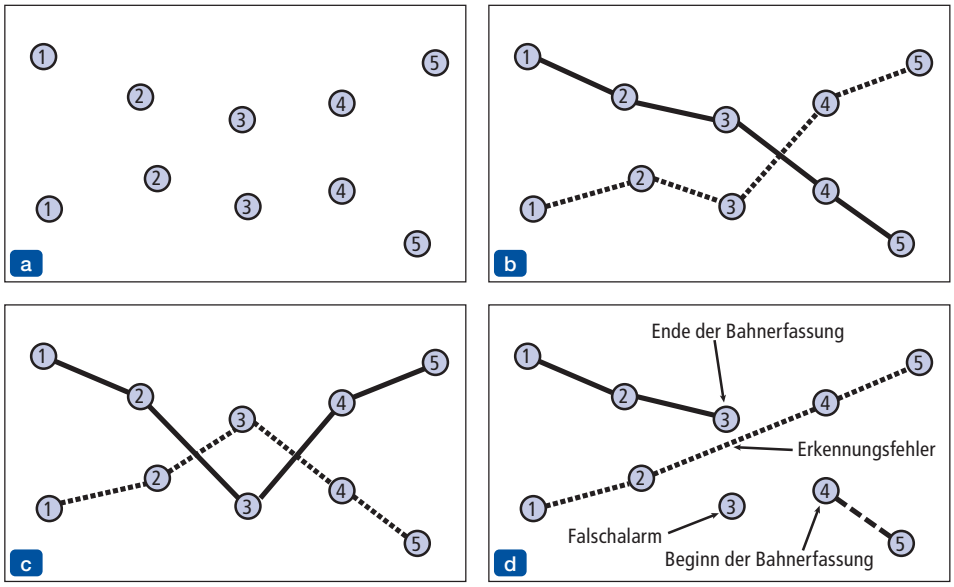


Abbildung 15.19: (a) Beobachtungen von Objektpositionen im zweidimensionalen Raum über fünf Zeitschritte. Jede Beobachtung ist mit dem Zeitschritt beschriftet. Diese Bezeichnung kennzeichnet aber nicht das Objekt, das die Beobachtung erzeugt hat. (b–c) Mögliche Hypothesen über die zugrunde liegenden Objektbahnen. (d) Eine Hypothese für den Fall, in dem falsche Alarme, Erkennungsfehler und Beginn bzw. Ende der Bahnerfassung möglich sind.

Wir versuchen nun, das Gesamtwahrscheinlichkeitsmodell für dieses Szenario zu formulieren, genau wie es für allgemeine temporale Prozesse in Gleichung (15.3) geschehen ist. Wie üblich lässt sich die gemeinsame Verteilung in die Beiträge für jeden Zeitschritt wie folgt zerlegen:

$$P(x_{0:t}^A, x_{0:t}^B, e_{1:t}^1, e_{1:t}^2) =$$
$$P(x_0^A)P(x_0^B)\prod_{i=1}^{t} P(x_i^A \mid x_{i-1}^A)P(x_i^B \mid x_{i-1}^B)P(e_i^1, e_i^2 \mid x_i^A, x_i^B). \quad (15.24)$$

Die Beobachtungsterme $P(e_i^1, e_i^2 | x_i^A, x_i^B)$ möchten wir in ein Produkt von zwei Termen – eines für jedes Objekt – zerlegen, doch würde dies das Wissen voraussetzen, welche Beobachtung durch welches Objekt erzeugt wurde. Stattdessen müssen wir über alle möglichen Wege der Zuordnung von Beobachtungen zu Objekten summieren. Einige

dieser Wege sind in ▶ Abbildung 15.19(b–c) dargestellt; im Allgemeinen sind für n Objekte und T Zeitschritte $(n!)^T$ Wege möglich – eine erschreckend große Zahl.

Im mathematischen Sinn ist der „Weg, die Beobachtungen den Objekten zuzuordnen" eine Sammlung von nicht beobachteten Zufallsvariablen, die die Quelle jeder Beobachtung kennzeichnen. Wir schreiben ω_t, um die 1:1-Zuordnung von Objekten zu Beobachtungen zur Zeit t zu kennzeichnen, wobei $\omega_t(A)$ und $\omega_t(B)$ die spezifischen Beobachtungen (1 oder 2) kennzeichnen, die ω_t an A und B zuweist. (Bei n Objekten gibt es $n!$ mögliche Werte für ω_t; hier ist $n! = 2$. Da die Beschriftungen „1" und „2" auf den Beobachtungen willkürlich zugeordnet sind, ist die A-priori-Verteilung von ω_t gleichförmig und ω_t unabhängig von den Zuständen der Objekte ${x_t}^A$ und ${x_t}^B$.) Somit können wir den Beobachtungsterm $P({e_i}^1, {e_i}^2 \mid {x_i}^A, {x_i}^B)$ auf ω_t konditionieren und vereinfachen dann zu:

$$\begin{aligned} P\left(e_i^1, e_i^2 \mid x_i^A, x_i^B\right) &= \sum_{\omega_i} P\left(e_i^1, e_i^2 \mid x_i^A, x_i^B, \omega_i\right) P\left(\omega_i \mid x_i^A, x_i^B\right) \\ &= \sum_{\omega_i} P\left(e_i^{\omega_i(A)} \mid x_i^A\right) P\left(e_i^{\omega_i(B)} \mid x_i^B\right) P\left(\omega_i \mid x_i^A, x_i^B\right) \\ &= \frac{1}{2} \sum_{\omega_i} P\left(e_i^{\omega_i(A)} \mid x_i^A\right) P\left(e_i^{\omega_i(B)} \mid x_i^B\right). \end{aligned}$$

Setzt man dies in Gleichung (15.24) ein, erhält man einen Ausdruck, der nur aus Termen von Übergangs- und Sensormodellen für einzelne Objekte und Beobachtungen besteht.

Wie bei allen Wahrscheinlichkeitsmodellen bedeutet Inferenz das Aussummieren der Variablen außer der Abfrage und der Evidenz. Für das Filtern in HMMs und DBNs konnten wir die Zustandsvariablen von 1 bis $t - 1$ durch einen einfachen dynamischen Programmiertrick aussummieren; bei Kalman-Filtern haben wir spezielle Eigenschaften der Gaußschen Verteilung genutzt. Bei der Datenzuordnung hatten wir weniger Glück. Es gibt keinen (bekannten) effizienten genauen Algorithmus, und zwar aus dem gleichen Grund wie beim Switching-Kalman-Filter (*Abschnitt 15.4.4*): Die Filterverteilung $P({x_t}^A \mid {e_{1:t}}^1, {e_{1:t}}^2)$ für Objekt A ist letztlich eine Mischung von exponentiell vielen Verteilungen – eine für jeden Weg, eine Beobachtungsfolge auszuwählen und sie A zuzuweisen.

Wegen der Komplexität exakter Inferenz sind viele unterschiedliche Näherungsverfahren verwendet worden. Beim einfachsten Ansatz wählt man eine einzelne „beste" Zuordnung zu jedem Zeitschritt aus, wobei die Position des Objekts beim aktuellen Zeitschritt gegeben ist. Diese Zuordnung verknüpft Beobachtungen mit Objekten und erlaubt es, jedes zu aktualisierende Objekt zu verfolgen und eine Vorhersage für den nächsten Zeitschritt zu treffen. Für die Auswahl der „besten" Zuordnung verwendet man üblicherweise den sogenannten **Nächste-Nachbarn-Filter** (Nearest-Neighbor-Filter), der wiederholt die engste Paarung von vorhergesagter Position und Beobachtung auswählt und diese Paarung der Zuordnung hinzufügt. Der Nearest-Neighbor-Filter ist geeignet, wenn sich die Objekte im Zustandsraum gut trennen lassen sowie Vorhersageunbestimmtheit und Beobachtungsfehler klein sind – d.h. keine Verwechslungsgefahr besteht. Bei größerer Unbestimmtheit hinsichtlich der korrekten Zuordnung ist es besser, die Zuordnung zu wählen, die die gemeinsame Wahrscheinlichkeit der aktuellen Beobachtungen für die gegebenen vorhergesagten Positionen maximiert. Dies lässt sich sehr effizient mit der **Ungarischen Methode** – auch als Kuhn-Munkres-Algorithmus bezeichnet – (Kuhn, 1955) bewerkstelligen, selbst wenn aus $n!$ Zuordnungen ausgewählt werden muss.

Jede Methode, die eine einzelne beste Zuweisung bei jedem Zeitschritt zusichert, versagt unter schwierigeren Bedingungen kläglich. Das gilt insbesondere, wenn der Algorithmus eine falsche Zuordnung zusichert. Dann kann der nächste Zeitschritt erheblich falsch liegen, was zu weiteren falschen Zuordnungen führt, usw. Zwei moderne Ansätze erweisen sich als effektiver. Ein **Partikelfilter-Algorithmus** (siehe *Abschnitt 15.5.3*) für die Datenzuordnung verwendet eine große Sammlung möglicher aktueller Zuordnungen. Ein **MCMC**-Algorithmus untersucht den Raum vergangener Zuordnungen – zum Beispiel könnten die Zustände in Abbildung 15.19(b-c) Zustände im MCMC-Zustandsraum sein – und kann seine Meinung zu vorherigen Zuordnungsentscheidungen ändern. Aktuelle MCMC-Datenzuordnungsmethoden können viele Hunderte von Objekten in Echtzeit verarbeiten, wobei sie eine gute Annäherung an die wahren A-posteriori-Verteilungen liefern.

Im bisher beschriebenen Szenario erzeugen n bekannte Objekte zu jedem Zeitschritt n Beobachtungen. Praktische Anwendungen der Datenzuordnung sind in der Regel wesentlich komplizierter. Oftmals umfassen die gemeldeten Beobachtungen **Falschalarme**, die nicht von realen Objekten verursacht werden. Zudem können Erkennungsfehler auftreten, wobei für ein reales Objekt keine Beobachtung gemeldet wird. Schließlich kommen neue Objekte hinzu und alte verschwinden. ▶ Abbildung 15.19(d) veranschaulicht diese Phänomene, die noch mehr zu berücksichtigende Welten erzeugen können.

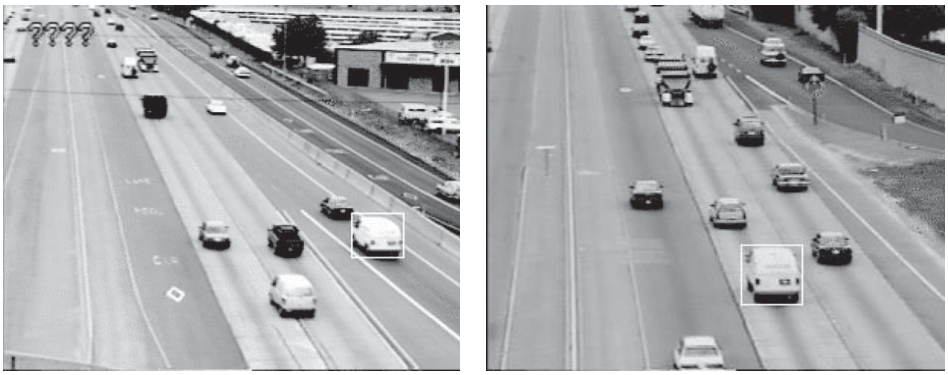


Abbildung 15.20: Bilder von Überwachungskameras, die etwa 3 km voneinander entfernt sind, auf dem Highway 99 in Sacramento, Kalifornien, (a) Zufahrt und (b) Abfahrt. Das eingerahmte Fahrzeug ist von beiden Kameras identifiziert worden.

▶ Abbildung 15.20 zeigt zwei Bilder von weit entfernten Kameras auf einer Schnellstraße in Kalifornien. In dieser Anwendung sind wir an zwei Zielen interessiert: Abschätzen der benötigten Zeit unter den aktuellen Verkehrsbedingungen, um im Schnellstraßensystem von einem Punkt zu einem anderen zu gelangen, und Messen der Auslastung, d.h. wie viele Fahrzeuge zwischen zwei Punkten im System zu bestimmten Tageszeiten oder an bestimmten Wochentagen unterwegs sind. Für beide Ziele ist es erforderlich, das Datenzuordnungsproblem über ein großes Gebiet mit vielen Kameras und Zehntausenden Fahrzeugen pro Stunde zu lösen. Bei visueller Überwachung können falsche Alarme durch bewegte Schatten, Gelenkfahrzeuge, Reflexionen in Pfützen usw. hervorgerufen werden. Zu Erkennungsfehlern kommt es durch Verdeckung, Nebel, Dunkelheit und ungenügenden optischen Kontrast. Ständig fahren die Fahrzeuge in das Schnellstraßensystem ein

und verlassen es. Darüber hinaus kann sich das Erscheinungsbild eines bestimmten Fahrzeuges zwischen den Kameras je nach Lichtbedingungen und Fahrzeuglage im Bild drastisch ändern. Auch das Bewegungsmodell unterliegt Änderungen, wenn sich Staus bilden und auflösen. Trotz dieser Probleme sind moderne Datenzuordnungsalgorithmen beim Abschätzen von Verkehrsparametern unter realen Bedingungen erfolgreich eingesetzt worden.Datenzuordnung ist eine wesentliche Grundlage für das Verfolgen einer komplexen Welt, da es sonst keine Möglichkeit gibt, mehrere Beobachtungen eines gegebenen Objekts zu kombinieren. Wenn Objekte in der Welt miteinander unter komplexen Aktivitäten interagieren, ist es erforderlich, die Datenzuordnung mit den relationalen Wahrscheinlichkeitsmodellen und den Modellen eines offenen Universums gemäß *Abschnitt 14.6.3* zu kombinieren. Dies ist derzeit ein aktiver Forschungsbereich.

Bibliografische und historische Hinweise

Viele der grundlegenden Ideen für die Abschätzung des Zustandes dynamischer Systeme stammen von dem Mathematiker C.F. Gauß (1809), der einen deterministischen Algorithmus der kleinsten Quadrate für das Problem formulierte, aus astronomischen Beobachtungen Umlaufbahnen abzuschätzen. Der russische Mathematiker A.A. Markov (1913) entwickelte bei seiner Analyse stochastischer Prozesse die später sogenannte **Markov-Zusicherung**. Er schätzte eine Markov-Kette erster Stufe für Buchstaben aus dem Text von *Eugen Onegin*. Die allgemeine Theorie der Markov-Ketten und ihrer Mischzeiten wird von Levin et al. (2008) behandelt.

Besonders wichtige Arbeiten zum Filtern erfolgten während des Zweiten Weltkriegs durch Wiener (1942) für Prozesse mit stetiger Zeit und Kolmogorov (1941) für Prozesse mit diskreter Zeit. Obwohl diese Arbeit in den folgenden 20 Jahren zu wichtigen technologischen Entwicklungen führte, machte ihre Verwendung der Frequenzdomänen-Repräsentation viele Berechnungen recht aufwändig. Die direkte Zustandsmodellierung des stochastischen Prozesses erwies sich als einfacher, wie durch Peter Swerling (1959) und Rudolf Kalman (1960) gezeigt. Die letztgenannte Arbeit führte den Kalman-Filter für die Vorwärtsinferenz in linearen Systemen mit Gaußschem Rauschen ein. Allerdings wurden Kalmans Ergebnisse schon eher vom dänischen Statistiker Thorvold Thiele (1880) und vom russischen Mathematiker Ruslan Stratonovich (1959) gefunden, die Kalman 1960 in Moskau traf. Nach einem Besuch beim NASA Ames Research Center (1960) erkannte Kalman die Anwendbarkeit der Methode zur Nachführung von Raketenbahnen und der Filter wurde später für die Apollo-Missionen implementiert. Wichtige Ergebnisse zur Glättung wurden von Rauch et al. (1965) abgeleitet und der beeindruckende Rauch-Tung-Striebel-Glätter ist heute immer noch Standardtechnologie. Viele frühe Ergebnisse sind in Gelb (1974) zusammengefasst. Bar-Shalom und Fortmann (1988) zeigen eine modernere Abhandlung mit Bayesschem Touch, ebenso wie viele Verweise auf die umfangreiche Literatur zu dem Thema. Chatfield (1989) und Box et al. (1994) beschreiben den „klassischen“ Ansatz zur Zeitfolgenanalyse.

Das Hidden-Markov-Modell und die zugehörigen Algorithmen für Inferenz und Lernen, einschließlich des Vorwärts-Rückwärts-Algorithmus, wurden von Baum und Petrie (1966) entwickelt. Der Viterbi-Algorithmus wurde zuerst in (Viterbi, 1967) veröffentlicht. Ähnliche Konzepte erschienen unabhängig davon in der Anhängerschaft der Kalman-Filter (Rauch et al., 1965). Der Vorwärts-Rückwärts-Algorithmus war einer der wichtigsten Vorläufer der allgemeinen Formulierung des EM-Algorithmus (Dempster et al., 1977); siehe auch *Kapitel 20*. Die Glättung mit konstantem Speicherbedarf erscheint in Binder et al.

(1997b) ebenso wie der in Übung 15.3 entwickelte Teilen&Herrschen-Algorithmus. Die Glättung in konstanter Zeit mit fester Verzögerung für HMMs erschien zuerst in Russell und Norvig (2003). HMMs haben viele Anwendungen in der Sprachverarbeitung (Charniak, 1993), Spracherkennung (Rainer und Juang, 1993), Maschinenübersetzung (Och und Ney, 2003), algorithmischen Biologie (Krogh et al., 1994; Baldi et al., 1993), Finanzwirtschaft (Bhar und Hamori, 2004) sowie anderen Gebieten gefunden. Für das grundlegende HMM-Modell gibt es mehrere Erweiterungen, wie zum Beispiel das hierarchische HMM (Fine et al. 1998) und das geschichtete HMM. Oliver et al., (2004) führen wieder eine Struktur in das Modell ein, die die einzelne Zustandsvariablen von HMMs ersetzt.

Dynamische Bayessche Netze (DBNs) können als dünn besetzte Kodierung eines Markov-Prozesses betrachtet werden und wurden in der KI zuerst von Dean und Kanzawa (1989b), Nicholson und Brady (1992) sowie Kjaerulff (1992) verwendet. Die letztgenannte Arbeit beinhaltet eine generische Erweiterung auf das Glaubensnetzsystem HUGIN, um die erforderlichen Funktionsmerkmale für das Erstellen und das Kompilieren Bayesscher Netze bereitzustellen. Das Buch von Dean und Wellman (1991) half dabei, DBNs und den probabilistischen Ansatz auf Planung und Steuerung in der KI bekannt zu machen. Murphy (2002) bietet eine gründliche Analyse von DBNs.

Dynamische Bayessche Netze sind sehr gebräuchlich für die Modellierung einer Vielzahl komplexer Bewegungsprozesse in der Computervision geworden (Huang et al., 1994; Intille und Bobick, 1999). Wie HMMs haben sie viele Anwendungen in der Spracherkennung (Zweig und Russell, 1998; Richardson et al., 2000; Stephenson et al., 2000; Nefian et al., 2002; Livescu et al., 2003), Genomik (Murphy und Mian, 1999; Perrin et al., 2003, Husmeier, 2003) und Roboterlokalisierung (Theocharous et al., 2004) gefunden. Die Verbindung zwischen HMMs und DBNs sowie zwischen dem Vorwärts-Rückwärts-Algorithmus und der Bayesschen Backpropagation wurde explizit von Smyth et al. (1997) formuliert. Eine weitere Unifizierung mit Kalman-Filtern (und anderen statistischen Modellen) erscheint in Roweis und Ghahramani (1999). Es gibt Prozeduren für das Erlernen der Parameter (Binder et al., 1997a; Ghahramani, 1998) und Strukturen (Friedman et al., 1998) von DBNs.

Der in *Abschnitt 15.5* beschriebene Partikelfilteralgorithmus hat eine besonders interessante Geschichte. Die ersten Sampling-Algorithmen für die Partikelfilter (auch als sequenzielle Monte-Carlo-Methoden bezeichnet) wurden in der Kontrolltheorie von Handschin und Mayne (1969) entwickelt, und die Resampling-Idee, die Seele des Partikelfilters, erschien in einem russischen Magazin zur Kontrolltheorie (Zaritskii et al., 1975). Später wurde es in der Statistik als **SIR** (**Sequential Importance-Sampling Resampling**) neu erfunden (Rubin, 1988; Liu und Chen, 1998), in der Kontrolltheorie als **Partikelfilter** (Gordon et al., 1993; Gordon, 1994); in der KI als **Überleben des Besten** (Kanazawa et al., 1995) und in der Computervision als **Kondensation** (Isard und Blake, 1996). Die Arbeit von Kanazawa et al. (1995) beinhaltet eine Verbesserung, die auch als **Evidenzumkehr** bezeichnet wird, wobei der Zustand zur Zeit $t+1$ abhängig von dem Zustand zur Zeit t *und der Evidenz zur Zeit* $t+1$ gesampelt wird. Auf diese Weise kann die Evidenz direkt auf die Stichprobenerzeugung einwirken und, wie von Doucet (1997) sowie Liu und Chen (1998) bewiesen, den Annäherungsfehler reduzieren. Partikelfilter sind auf vielen Gebieten eingesetzt worden, unter anderem für das Verfolgen komplexer Bewegungsmuster in der Videotechnik (Isard und Blake, 1996), der Aktienkursvorhersage (de Freitas et al., 2000) und der Diagnose von Fehlern auf Planetenrovern (Verma et al., 2004). Eine als **Rao-Blackwellized-Partikelfilter** oder RBPF bezeichnete Variante (Doucet et al., 2000; Murphy und Russell, 2001) wendet die Partikelfilterung auf eine Teilmenge der Zustandsvariablen

an und führt für jeden Partikel eine genaue Inferenz für die restlichen Variablen durch, wobei auf die Wertfolge im Partikel konditioniert wird. In manchen Fällen arbeitet RBPF mit Tausenden von Zustandsvariablen. *Kapitel 25* beschreibt eine Anwendung von RBPF auf Lokalisierung und Kartenerstellung in der Robotik. Das Buch von Doucet et al. (2001) sammelt viele wichtige Artikel zu **sequenziellen Monte-Carlo-Algorithmen** (SMC), wobei die Partikelfilterung den wichtigsten Teil ausmacht. Pierre Del Moral und Kollegen haben umfangreiche theoretische Analysen von SMC-Algorithmen durchgeführt (Del Moral, 2004; Del Moral et al., 2006).

MCMC-Methoden (siehe *Abschnitt 14.5.2*) können auf Filterprobleme angewendet werden. Zum Beispiel lässt sich Gibbs-Sampling direkt auf ein aufgerolltes DBN anwenden. Um das Problem wachsender Aktualisierungszeiten zu vermeiden, wenn das aufgerollte Netz wächst, sampelt der **Decayed MCMC-Filter** (Marthi et al., 2002) vorzugsweise neuere Zustandsvariablen, und zwar mit einer Wahrscheinlichkeit, die für eine Variable k Schritte in der Vergangenheit mit $1/k^2$ fällt. Das zerfallende MCMC-Verfahren ist ein nachweisbar nichtdivergenter Filter. Nichtdivergente Filter können auch für bestimmte Typen von **Filtern mit angenommener Dichte** erhalten werden. Derartige Filter gehen davon aus, dass die A-posteriori-Verteilung über den Zuständen zur Zeit t zu einer besonderen endlich parametrisierten Familie gehört. Wenn die Verteilung durch die Projektions- und Aktualisierungsschritte aus dieser Familie herausgebracht wird, wird sie wieder zurückprojiziert, um die beste Annäherung innerhalb der Familie zu liefern. Für DBNs nehmen der Boyen-Koller-Algorithmus (Boyen et al., 1999) und der **Factored-Frontier**-Algorithmus (Murphy und Weiss, 2001) an, dass die A-posteriori-Verteilung durch ein Produkt von kleinen Faktoren angenähert werden kann. Es wurden verschiedene Techniken (siehe *Kapitel 14*) für temporale Modelle entwickelt. Ghahramani und Jordan (1997) beschreiben einen Annäherungsalgorithmus für das **faktorisierende HMM**, ein DBN, in dem zwei oder mehr unabhängig voneinander entwickelte Markov-Ketten durch einen gemeinsam genutzten Beobachtungsstrom verknüpft sind. Jordan et al. (1998) decken mehrere andere Anwendungen ab.

Die Datenzuordnung für das Verfolgen mehrerer Ziele wurde in einem probabilistischen Rahmen zuerst von Sittler (1964) beschrieben. Der erste praktische Algorithmus für Probleme im großen Maßstab war der „Multiple Hypothesis Tracker“ oder MHT-Algorithmus (Reid, 1979). Bar-Shalom und Fortmann (1988) sowie Bar-Shalom (1992) haben viele wichtige Artikel zusammengestellt. Die Entwicklung eines MCMC-Algorithmus für die Datenzuordnung geht auf Pasula et al. (1999) zurück, die den Algorithmus auf Probleme der Verkehrsüberwachung angewendet haben. Oh et al. (2009) bieten eine formale Analyse und umfangreiche experimentelle Vergleiche zu anderen Methoden. Schulz et al. (2003) beschreiben eine Datenzuordnungsmethode basierend auf der Partikelfilterung. Ingemar Cox hat die Komplexität der Datenzuordnung analysiert (Cox, 1993; Cox und Hingorani, 1994) und den Kreis der Bildverarbeiter auf das Thema aufmerksam gemacht. Außerdem hat er darauf hingewiesen, dass sich der Ungarische Algorithmus in polynomialer Zeit auf das Problem anwenden lässt, die wahrscheinlichsten Zuordnungen zu finden, was lange als nicht zu bewältigendes Problem unter Experten auf dem Gebiet der Nachverfolgung galt. Der Algorithmus selbst wurde von Kuhn (1955) bekannt gemacht. Die Grundlage hierfür bildeten Übersetzungen von Artikeln der beiden ungarischen Mathematiker Dénes König und Jenö Egerváry, die 1931 erschienen sind. Das grundlegende Theorem wurde allerdings schon eher in einem unveröffentlichten lateinischen Manuskript vom berühmten preußischen Mathematiker Carl Gustav Jacobi (1804–1851) abgeleitet.

Zusammenfassung

Dieses Kapitel hat sich mit dem allgemeinen Problem beschäftigt, probabilistische temporale Prozesse darzustellen und mit ihnen zu schließen. Die wichtigsten Aspekte dabei sind:

- Der wechselnde Zustand der Welt wird durch Verwendung einer Menge von Zufallsvariablen verarbeitet, die den Zustand zu jedem beliebigen Zeitpunkt repräsentieren.
- Repräsentationen können darauf ausgelegt werden, die **Markov-Eigenschaft** zu erfüllen, sodass die Zukunft unabhängig von der Vergangenheit bei bekannter Gegenwart ist. Kombiniert mit der Annahme, dass der Prozess **stationär** ist, d.h., die Dynamik ändert sich nicht mit der Zeit, vereinfacht dies die Repräsentation ganz wesentlich.
- Man kann sich vorstellen, dass ein temporales Wahrscheinlichkeitsmodell aus einem **Übergangsmodell**, das die Entwicklung beschreibt, und einem **Sensormodell**, das den Beobachtungsprozess beschreibt, besteht.
- Die wichtigsten Inferenzaufgaben in temporalen Modellen sind **Filterung**, **Vorhersage**, **Glättung** und Berechnung der **wahrscheinlichsten Erklärung**. Sie können unter Verwendung einfacher rekursiver Algorithmen realisiert werden, deren Laufzeit linear zur Länge der Folge ist.
- Drei Familien temporaler Modelle wurden genauer betrachtet: **Hidden-Markov-Modelle**, **Kalman-Filter** und **dynamische Bayessche Netze** (die die beiden anderen als Sonderfälle beinhalten).
- Wenn keine speziellen Zusicherungen getroffen werden, wie etwa in Kalman-Filtern, erscheint eine genaue Inferenz mit vielen Zustandsvariablen nicht realisierbar zu sein. In der Praxis scheint der Algorithmus der **Partikelfilterung** ein effizienter Annäherungsalgorithmus zu sein.
- Müssen viele Objekte verfolgt werden, tritt eine Unbestimmtheit auf, welche Beobachtungen zu welchen Objekten gehören – das Problem der **Datenzuordnung**. Die Anzahl der Zuordnungshypothesen ist normalerweise unlösbar hoch, wobei aber MCMC- und Partikelfilteralgorithmen in der Praxis gut funktionieren.

Übungen zu Kapitel 15

Lösungshinweise

1 Zeigen Sie, dass jeder Markov-Prozess zweiter Stufe als Markov-Prozess erster Stufe mit erweiterter Zustandsvariablenmenge formuliert werden kann. Kann dies immer *sparsam* erfolgen, d.h. ohne Erhöhung der Parameterzahl für die Spezifikation des Übergangsmodells?

2 In dieser Übung betrachten wir, was mit den Wahrscheinlichkeiten der Regenschirmwelt an den Grenzen langer Zeitfolgen passiert.

a. Angenommen, wir beobachten eine unendliche Folge von Tagen, an denen der Regenschirm erscheint. Zeigen Sie, dass die Wahrscheinlichkeit von Regen am aktuellen Tag mit zunehmenden Tagen monoton bis zu einem Fixpunkt steigt. Berechnen Sie diesen Fixpunkt.

b. Betrachten Sie jetzt eine weitere *Vorhersage* in die Zukunft nur auf der Basis der beiden ersten Regenschirmbeobachtungen. Berechnen Sie zuerst die Wahrscheinlichkeit $P(r_{2+k}|u_1, u_2)$ für $k = 1...20$ und zeichnen Sie die Ergebnisse. Sie sollten erkennen, dass die Wahrscheinlichkeit gegen einen Fixpunkt konvergiert. Beweisen Sie, dass der genaue Wert dieses Fixpunktes 0,5 ist.

3 Diese Übung entwickelt eine speichereffiziente Variante des in Abbildung 15.4 beschriebenen Vorwärts-Rückwärts-Algorithmus. Wir wollen $\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t})$ für $k = 1, ..., t$ berechnen. Dazu wenden wir einen Teilen&Herrschen-Ansatz an.

a. Wir nehmen der Einfachheit halber an, dass t ungerade ist und dass die Hälfte bei $h = (t + 1)/2$ liegt. Zeigen Sie, dass $\mathbf{P}(\mathbf{X}_k|\mathbf{e}_{1:t})$ für $k = 1, ..., t$ berechnet werden kann, wenn nur die anfängliche Vorwärtsnachricht $\mathbf{f}_{1:0}$, die Rückwärtsnachricht $\mathbf{b}_{h+1:t}$ und die Evidenz $\mathbf{e}_{1:h}$ gegeben sind.

b. Zeigen Sie ein ähnliches Ergebnis für die zweite Hälfte der Folge.

c. Aus den Ergebnissen für (a) und (b) kann ein rekursiver Teilen&Herrschen-Algorithmus konstruiert werden, indem die Folge zuerst vorwärts und dann vom Ende aus wieder rückwärts durchlaufen wird, wobei jeweils nur die erforderlichen Nachrichten in der Mitte und an den Enden gespeichert werden. Anschließend wird der Algorithmus für jede der Hälften aufgerufen. Schreiben Sie den Algorithmus im Detail.

d. Berechnen Sie Zeit- und Speicherkomplexität des Algorithmus als Funktion von t, der Länge der Folge. Wie ändern sie sich, wenn wir die Eingabe in mehr als zwei Teile zerlegen?

4 In *Abschnitt 15.2.3* haben wir eine abgeflachte Prozedur für die Ermittlung der wahrscheinlichsten Zustandsfolge für eine vorgegebene Beobachtungsfolge skizziert. Die Prozedur sucht den wahrscheinlichsten Zustand für jeden Zeitschritt unter Verwendung der Glättung und gibt die aus diesen Zuständen zusammengesetzte Folge zurück. Zeigen Sie, dass diese Prozedur für einige temporale Wahrscheinlichkeitsmodelle und Beobachtungsfolgen eine unmögliche Zustandsfolge zurückgibt (d.h., die bedingte Wahrscheinlichkeit der Folge ist null).

5 Gleichung (15.12) beschreibt den Filterprozess für die Matrixformulierung von HMMs. Geben Sie eine ähnliche Gleichung für die Berechnung der Wahrscheinlichkeiten an, was in Gleichung (15.7) allgemein beschrieben wurde.

6 In *Abschnitt 15.3.2* ist die A-priori-Verteilung über Positionen gleichförmig und das Übergangsmodell nimmt eine gleiche Wahrscheinlichkeit für die Bewegung zu einem beliebigen Nachbarquadrat an. Wie sieht es aus, wenn diese Annahmen falsch sind? Nehmen Sie an, dass die Anfangsposition tatsächlich gleichförmig vom nordwestlichen Quadranten des Raumes ausgewählt wird und die $Bewegen$-Aktion tatsächlich nach Südost tendiert. Halten Sie das HMM-Modell feststehend und erläutern Sie für verschiedene Werte von ϵ die Wirkung auf die Positionierungs- und Pfadgenauigkeit, wenn die Tendenz nach Südosten zunimmt.

7 Betrachten Sie eine Version des Staubsaugerroboters (*Abschnitt 15.3.2*), der die Strategie befolgt, solange wie möglich geradeaus zu gehen. Nur wenn er auf ein Hindernis trifft, schlägt er eine neue (zufällig ausgewählte) Richtung ein. Um diesen Roboter zu modellieren, besteht jeder Zustand im Modell aus einem Paar ($Position$, $Richtung$). Implementieren Sie dieses Modell und stellen Sie fest, wie gut der Viterbi-Algorithmus einen Roboter mit diesem Modell verfolgen kann. Die Strategie des Roboters ist eingeschränkter als beim Roboter mit zufälligen Bewegungen (random walk). Heißt das, dass die Vorhersagen des wahrscheinlichsten Pfades genauer sind?

8 Für den Staubsaugerroboter haben wir drei Strategien beschrieben: (1) eine gleichförmige zufällige Bewegung, (2) eine Vorzugsrichtung für das Wandern nach Südost, wie es in Übung 15.6 beschrieben ist, und (3) die in Übung 15.7 beschriebene Strategie. Ein Beobachter erhält nun die Beobachtungsfolge von einem Staubsaugerroboter, ist sich aber nicht sicher, welcher der drei Strategien der Roboter folgt. Welchen Ansatz sollte der Beobachter verwenden, um den wahrscheinlichsten Pfad bei den gegebenen Beobachtungen zu finden? Implementieren Sie den Ansatz und testen Sie ihn. Wie stark leidet die Positionierungsgenauigkeit verglichen mit dem Fall, in dem der Beobachter weiß, an welche Strategie sich der Roboter hält?

9 In dieser Übung geht es um das Filtern in einer Umgebung ohne Orientierungspunkte. Betrachten Sie einen Staubsaugerroboter in einem leeren Raum, der durch ein rechteckiges $n \times m$-Raster dargestellt wird. Die Position des Roboters ist verborgen; dem Beobachter steht lediglich ein Geräuschortungssensor zur Verfügung, der die Position des Roboters näherungsweise angibt. Wenn sich der Roboter bei Position (x, y) befindet, liefert der Sensor mit der Wahrscheinlichkeit 0,1 die korrekte Position, mit jeweils der Wahrscheinlichkeit 0,05 eine der 8 Positionen, die (x, y) direkt umgeben, mit der Wahrscheinlichkeit von jeweils 0,025 eine der 16 Positionen, die um die ersten 8 Felder liegen, und mit der restlichen Wahrscheinlichkeit von 0,1 meldet er „keine Ortung". Der Roboter wählt eine Richtung aus und folgt ihr mit der Wahrscheinlichkeit von 0,7 bei jedem Schritt. Eine neue Richtung wählt der Roboter mit der Wahrscheinlichkeit von 0,3 aus (oder mit der Wahrscheinlichkeit von 1, wenn er auf eine Wand trifft). Implementieren Sie dies als HMM und realisieren Sie Filterung, um den Roboter zu verfolgen. Wie genau lässt sich der Pfad des Roboters verfolgen?

10 Häufig wollen wir ein System mit stetigen Zuständen beobachten, dessen Verhalten unvorhersehbar zwischen einer Folge von k verschiedenen „Modi" wechselt. Ein Flugzeug beispielsweise, das versucht, einer Rakete auszuweichen, kann eine Folge unterschiedlicher Manöver ausführen, die die Rakete versucht nachzuvollziehen. ▶ Abbildung 15.21 zeigt ein derartiges **Switching-Kalman-Filtermodell** als Bayessches Netz.

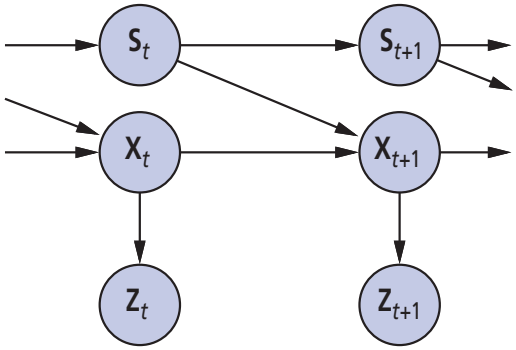


Abbildung 15.21: Darstellung eines Switching-Kalman-Filters als Bayessches Netz. Die Schaltvariable S_t ist eine Variable mit diskretem Zustand, deren Wert das Übergangmodell für die Variablen mit stetigem Zustand, $\mathbf{X}_t$, festlegt. Das Übergangsmodell $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{X}_t, S_t = i)$ ist für jeden diskreten Zustand i ein lineares Gaußsches Modell, so wie in einem normalen Kalman-Filter. Das Übergangsmodell für den diskreten Zustand $\mathbf{P}(S_{t+1}|S_t)$ kann man sich als Matrix vorstellen, wie in einem Hidden-Markov-Modell.

a. Angenommen, der diskrete Zustand S_t hat k mögliche Werte und die vorhergehende stetige Zustandsschätzung $\mathbf{P}(\mathbf{X}_0)$ ist eine multivariate Gaußsche Verteilung. Zeigen Sie, dass die Vorhersage $\mathbf{P}(\mathbf{X}_1)$ eine **Mischung aus Gaußschen Verteilungen** ist – d.h. eine gewichtete Summe Gaußscher Verteilungen, sodass die Summe der Gewichtungen gleich 1 ist.

b. Zeigen Sie, dass, wenn die aktuell stetige Zustandsschätzung $\mathbf{P}(\mathbf{X}_t|\mathbf{e}_{1:t})$ eine Mischung aus m Gaußschen Verteilungen ist, die aktualisierte Zustandsschätzung $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{e}_{1:t+1})$ im allgemeinen Fall eine Mischung aus km Gaußschen Verteilungen ist.

c. Welchen Aspekt des temporalen Prozesses stellen die Gewichtungen in der Gaußschen Mischung dar?

Insgesamt zeigen die Ergebnisse in (a) und (b), dass die Repräsentation der bedingten Verteilung ohne Grenze anwächst – selbst für geschaltete Kalman-Filter, die einfachsten hybriden dynamischen Modelle.

11 Vervollständigen Sie den fehlenden Schritt in der Ableitung von Gleichung (15.19), den ersten Aktualisierungsschritt des eindimensionalen Kalman-Filters.

12 Wir betrachten das Verhalten der Varianzaktualisierung in Gleichung (15.20).

a. Zeichnen Sie den Wert von σ_t^2 als Funktion von t für verschiedene Werte für σ_x^2 und σ_z^2.

b. Zeigen Sie, dass die Aktualisierung einen Fixpunkt σ^2 hat, sodass $\sigma_t^2 \to \sigma^2$, wenn $t \to \infty$, und berechnen Sie den Wert von σ^2.

c. Geben Sie eine qualitative Erklärung dafür, was bei $\sigma_x^2 \to 0$ und $\sigma_z^2 \to 0$ passiert.

13 Ein Professor möchte wissen, ob Studenten genügend Schlaf bekommen. Jeden Tag beobachtet er, ob die Stundenten im Kurs schlafen und ob sie rote Augen haben. Der Professor stellt die folgende Domänentheorie auf:

– Die A-priori-Wahrscheinlichkeit, genügend Schlaf zu bekommen, beträgt ohne Beobachtungen 0,6.

– Die Wahrscheinlichkeit, genügend Schlaf in der Nacht t zu bekommen, beträgt 0,8, vorausgesetzt, dass er in der vorhergehenden Nacht genügend Schlaf hatte, sonst 0,2.

- Die Wahrscheinlichkeit, rote Augen zu haben, beträgt 0,2, wenn der Student genügend geschlafen hat, sonst 0,7.
- Die Wahrscheinlichkeit, im Kurs einzuschlafen, beträgt 0,1, wenn der Student genügend Schlaf hatte, sonst 0,3.

Formulieren Sie diese Informationen als dynamisches Bayessches Netz, mit dem der Professor eine Folge von Beobachtungen filtern oder aus diesen vorhersagen kann. Anschließend formulieren Sie es als Hidden-Markov-Modell, das eine einzige Beobachtungsvariable hat. Geben Sie die vollständigen Wahrscheinlichkeitstabellen für das Modell an.

14 Führen Sie für das in Übung 15.13 spezifizierte DBN und für die Evidenzvariablen

e1 = keine roten Augen, schläft nicht im Kurs

e2 = rote Augen, schläft nicht im Kurs

e3 = rote Augen, schläft im Kurs

die folgenden Berechnungen durch:

a. Zustandsschätzung: Berechnen Sie $P(GenugSchlaf_t | \mathbf{e}_{1:t})$ für jedes $t = 1, 2, 3$.

b. Glättung: Berechnen Sie $P(GenugSchlaf_t | \mathbf{e}_{1:3})$ für jedes $t = 1, 2, 3$.

c. Vergleichen Sie die gefilterten und geglätteten Wahrscheinlichkeiten für $t = 1$ und $t = 2$.

15 Nehmen Sie an, dass ein bestimmter Student jeden Tag mit roten Augen erscheint und im Kurs schläft. Erläutern Sie anhand des in Übung 15.13 angegebenen Modells, warum die Wahrscheinlichkeit, dass der Student in der vorhergehenden Nacht genügend Schlaf hatte, gegen einen Fixpunkt konvergiert, anstatt immer weiter abzufallen, wenn wir mehr Evidenztage erfassen. Wie lautet der Fixpunkt? Beantworten Sie dies sowohl numerisch (durch Berechnung) als auch analytisch.

16 In dieser Übung analysieren wir detailliert das persistente Fehlermodell für den Batteriesensor in Abbildung 15.15(a).

a. Abbildung 15.15(b) stoppt bei $t = 32$. Beschreiben Sie qualitativ, was passieren soll, wenn $t \to \infty$, wenn der Sensor weiterhin 0 liest.

b. Angenommen, die Außentemperatur beeinflusst den Batteriesensor, sodass bei steigender Temperatur transiente Fehler wahrscheinlicher werden. Zeigen Sie, wie die DBN-Struktur in Abbildung 15.15(a) erweitert werden kann, und erklären Sie, welche Änderungen an den BWTs erforderlich sind.

c. Können für die neue Netzstruktur Batteriemesswerte von dem Roboter verwendet werden, um die aktuelle Temperatur abzuleiten?

17 Versuchen Sie, den Algorithmus für die Variableneliminierung auf das in drei Zeitscheiben zerlegte Regenschirm-DBN anzuwenden. Die Abfrage ist $\mathbf{P}(R_3 | u_1, u_2, u_3)$. Zeigen Sie, dass die Komplexität des Algorithmus – die Größe des größten Faktors – immer gleich ist, egal ob die Regenvariablen rückwärts oder vorwärts eliminiert werden.

Einfache Entscheidungen

16

ÜBERBLICK

In diesem Kapitel sehen wir, wie ein Agent Entscheidungen treffen soll, damit er sein Ziel erreicht – zumindest im Durchschnitt aller Fälle.

In diesem Kapitel kehren wir zum Konzept der Nutzentheorie zurück, die in *Kapitel 13* vorgestellt wurde, und zeigen, wie sie mit der Wahrscheinlichkeitstheorie kombiniert wird, um einen entscheidungstheoretischen Agenten zu erzeugen – einen Agenten, der rationale Entscheidungen basierend auf seinem Glauben und seinem Willen treffen kann. Ein solcher Agent kann Entscheidungen in Kontexten treffen, wo Unsicherheit und konflikterzeugende Ziele es einem logischen Agenten nicht erlauben, eine Entscheidung zu treffen. Letztlich hat ein zielbasierter Agent eine binäre Auswahl zwischen guten (Ziel-) und schlechten (Nichtziel-)Zuständen, während ein entscheidungstheoretischer Agent ein stetiges Maß für die Zustandsqualität besitzt.

Abschnitt 16.1 führt das grundlegende Konzept der Entscheidungstheorie ein: die Maximierung des erwarteten Nutzens. *Abschnitt 16.2* zeigt, dass das Verhalten jedes rationalen Agenten abgedeckt werden kann, indem man eine zu maximierende Nutzenfunktion annimmt. *Abschnitt 16.3* beschreibt die Natur der Nutzenfunktionen detaillierter, insbesondere ihre Relation zu einzelnen Quantitäten, wie etwa Geld. *Abschnitt 16.4* zeigt, wie man mit Nutzenfunktionen umgeht, die von mehreren Quantitäten abhängig sind. In *Abschnitt 16.5* beschreiben wir die Implementierung von entscheidungstreffenden Systemen. Insbesondere stellen wir einen Formalismus vor, der als **Entscheidungsnetz** (auch **Einflussdiagramm**) bezeichnet wird und der die Bayesschen Netze erweitert, indem er Aktionen und Nutzen berücksichtigt. Das restliche Kapitel beschäftigt sich mit Themen, die bei der Anwendung der Entscheidungstheorie auf Expertensysteme auftreten.

16.1 Glauben und Wünsche unter Unsicherheit kombinieren

In ihrer einfachsten Form beschäftigt sich die Entscheidungstheorie mit der Auswahl unter Aktionen basierend auf ihren *unmittelbaren* Ergebnissen; d.h., die Umgebung wird im selben Sinn wie in *Abschnitt 2.3.2* definiert als episodisch angenommen. (Diese Annahme wird in *Kapitel 17* gelockert.) In *Kapitel 3* haben wir die Notation $\text{RESULT}(s_0, a)$ für den Zustand verwendet, der das deterministische Ergebnis einer Aktion a im Zustand s_0 ist. In diesem Kapitel geht es nun um nichtdeterministische partiell beobachtbare Umgebungen. Da der Agent den aktuellen Zustand nicht unbedingt kennt, lassen wir ihn weg und definieren $\text{RESULT}(a)$ als *Zufallsvariable*, deren Werte die möglichen Ergebniszustände sind. Die Wahrscheinlichkeit von Ergebnis s' bei gegebenen Evidenzbeobachtungen $\mathbf{e}$ wird geschrieben als

$$P(\text{RESULT}() = s' \mid a, \mathbf{e}),$$

wobei das a auf der rechten Seite des Bedingungsstriches für das Ereignis steht, dass Aktion a ausgeführt wird.[1]

Die Vorlieben, die ein Agent im Hinblick auf die Weltzustände aufweist, werden durch eine **Nutzenfunktion** abgedeckt, die eine einzelne Zahl zuordnet und damit ausdrückt, wie wünschenswert ein Zustand ist. Der **erwartete Nutzen** (engl. *U*tility) einer Aktion

1 Die klassische Entscheidungstheorie lässt den aktuellen Zustand S_0 implizit. Wir können ihn aber explizit machen, indem wir $P(\text{RESULT}(a) = s' \mid a, \mathbf{e}) = \Sigma_s P(\text{RESULT}(s, a) = s' \mid a) P(S_0 = s \mid \mathbf{e})$ schreiben.

bei gegebener Evidenz $EU(a|\mathbf{e})$ ist einfach der durchschnittliche Nutzenwert der Ergebnisse, gewichtet nach der Wahrscheinlichkeit, mit der dieses Ergebnis auftritt:

$$EU(a|\mathbf{e}) = \sum P(\text{Result}(a) = s' | a, \mathbf{e}) U(s'). \tag{16.1}$$

Das Prinzip des **maximalen erwarteten Nutzens** (MEN) besagt, dass ein rationaler Agent eine Aktion auswählen soll, die seinen erwarteten Nutzen maximiert:

$$Aktion = \operatorname{argmax}_a EU(a|\mathbf{e}).$$

In gewisser Weise könnte man das MEN-Prinzip als die Definition der gesamten KI betrachten. Ein intelligenter Agent muss einfach nur die verschiedenen Quantitäten berechnen, den Nutzen über seine Aktionen maximieren und fertig. Aber das bedeutet nicht, dass das KI-Problem durch die Definition *gelöst* ist!

Das MEN-Prinzip *formalisiert* das allgemeine Konzept, dass der Agent „das Richtige" tun sollte, nähert sich aber nur geringfügig einer vollständigen *Operationalisierung* dieses Rates. Um den Zustand der Welt abzuschätzen, braucht man Wahrnehmung, Lernen, Wissensrepräsentation und Inferenz. Für die Berechnung von $P(\text{RESULT}(a)|a, \mathbf{e})$ braucht man ein vollständiges kausales Modell der Welt und (wie wir in *Kapitel 14* gesehen haben) NP-harte Inferenz in Bayesschen Netzen. Die Berechnung des Nutzens jedes Zustandes $U(s')$ bedingt häufig eine Suche oder Planung, weil ein Agent nicht weiß, wie gut ein Zustand ist, bis er weiß, wohin er von diesem Zustand aus gelangen kann. Die Entscheidungstheorie ist also kein Allheilmittel, das das KI-Problem löst – doch sie stellt eine nützliche Umgebung bereit.

Tipp

Das MEN-Prinzip hat eine klare Relation zu dem in *Kapitel 2* vorgestellten Konzept der Leistungsmaße. Die grundlegende Idee dabei ist sehr einfach. Betrachten Sie die Umgebungen, die zu einem Agenten mit einem bestimmten Wahrnehmungsverlauf führen könnten, und betrachten Sie die verschiedenen Agenten, die wir entwerfen könnten. *Wenn ein Agent eine Nutzenfunktion maximiert, die das Leistungsmaß, nach dem sein Verhalten bewertet wird, korrekt reflektiert, erreicht er die höchstmögliche Leistungsbewertung (gemittelt über alle möglichen Umgebungen).* Das ist die zentrale Rechtfertigung für das eigentliche MEN-Prinzip. Während die Behauptung zunächst vielleicht tautologisch erscheint, verkörpert sie letztendlich einen sehr wichtigen Übergang von einem globalen, externen Kriterium der Rationalität – dem Leistungsmaß über Umgebungsverläufe – hin zu einem lokalen, internen Kriterium, das die Maximierung einer auf den nächsten Zustand angewendeten Nutzenfunktion beinhaltet.

16.2 Grundlagen der Nutzentheorie

Intuitiv scheint das Prinzip des maximalen erwarteten Nutzens (MEN) eine sinnvolle Weise darzustellen, Entscheidungen zu treffen, aber es ist keineswegs offensichtlich, dass es sich dabei um die *einzige* rationale Weise handelt. Warum sollte schließlich die Maximierung des *durchschnittlichen* Nutzens so speziell sein? Warum sollte man nicht versuchen, die Summe der Kubikwerte möglicher Nutzen zu berechnen oder den schlimmsten möglichen Verlust zu minimieren? Könnte ein Agent nicht auch rational handeln, indem er einfach Prioritäten für Zustände angibt, ohne diesen numerische Werte zuzuordnen? Und warum sollte es überhaupt eine Nutzenfunktion mit den geforderten Eigenschaften geben? Wir werden es sehen.

16.2.1 Einschränkungen rationaler Prioritäten

Diese Fragen lassen sich beantworten, wenn man einige Bedingungen für die Prioritäten formuliert, die ein rationaler Agent aufweisen sollte, und dann zeigt, dass aus den Bedingungen das MEN-Prinzip abgeleitet werden kann. Wir verwenden die folgende Notation, um die Prioritäten eines Agenten zu beschreiben:

$A \succ B$ Der Agent bevorzugt A gegenüber B.

$A \sim B$ Der Agent ist unentschlossen zwischen A und B.

$A \succsim B$ Der Agent bevorzugt A gegenüber B oder ist unentschlossen zwischen beiden.

Jetzt lautet die offensichtliche Frage, worum es sich bei A und B handelt. Es könnten die Zustände der Welt sein, doch meistens gibt es Unbestimmtheit darüber, was wirklich geboten wird. Zum Beispiel weiß ein Fluggast, dem „das Nudelgericht oder das Huhn" angeboten wird, nicht, was ihn unter der Aluminiumfolie erwartet.[2] Die Nudeln könnten köstlich oder zu fest, das Huhn saftig oder bis zur Unkenntlichkeit zerkocht sein. Wir können uns die Menge der Ergebnisse für jede Aktion als **Lotterie** vorstellen – wobei jede Aktion ein Los verkörpert. Eine Lotterie L mit den möglichen Ergebnissen $S_1, ..., S_n$, die mit den Wahrscheinlichkeiten $p_1, ..., p_n$ auftreten können, wird dargestellt als

$$L = [p_1, S_1; p_2, S_2; ... p_n, S_n].$$

Im Allgemeinen kann jedes Ergebnis S_i einer Lotterie ein atomarer Zustand oder eine weitere Lotterie sein. Der wichtigste Aspekt für die Nutzentheorie ist, zu verstehen, wie die Prioritäten zwischen komplexen Lotterien mit Prioritäten zwischen den zugrunde liegenden Zuständen in diesen Lotterien verbunden sind. Dazu formulieren wir sechs Einschränkungen, die für jede sinnvolle Prioritätsrelation vorausgesetzt werden:

- **Sortierbarkeit:** Sind zwei beliebige Lotterien gegeben, muss ein rationaler Agent entweder die eine der anderen vorziehen oder die beiden als gleich wünschenswert einordnen. Das bedeutet, der Agent kommt um eine Entscheidung nicht herum. Wie in *Abschnitt 13.2.3* gezeigt, ist eine Verweigerung einer Wette so, als würde man verbieten, dass die Zeit vergeht.

  Es gilt genau eine der Bedingungen $(A \succ B)$, $(B \succ A)$ oder $(A \sim B)$.

- **Transitivität:** Wenn ein Agent bei drei Lotterien A gegenüber B vorzieht und B gegenüber C, dann muss der Agent auch A gegenüber C vorziehen.

  $$(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$$

- **Stetigkeit:** Wenn eine Lotterie B in Bezug auf die Priorität zwischen A und C liegt, gibt es eine Wahrscheinlichkeit p, für die der rationale Agent unentschlossen ist, ob er B sicher annehmen oder ob er die Lotterie wählen soll, die A mit der Wahrscheinlichkeit p und C mit der Wahrscheinlichkeit $1 - p$ erhält.

  $$A \succ B \succ C \Rightarrow \exists p\, [p, A; 1 - p, C] \sim B$$

- **Ersetzbarkeit:** Wenn ein Agent zwischen zwei Lotterien A und B unentschlossen ist, dann ist der Agent unentschlossen zwischen zwei komplexeren Lotterien, die genau gleich sind, außer dass in einer davon B durch A ersetzt wurde. Das gilt unabhängig von den Wahrscheinlichkeiten und den anderen Ergebnissen in den Lotterien.

2 Wir möchten uns bei den Lesern entschuldigen, deren Fluggesellschaften keine Mahlzeiten mehr auf Langstreckenflügen anbieten.

$$A \sim B \Rightarrow [p, A; 1 - p, C] \sim [p, B; 1 - p, C]$$

Dies gilt auch, wenn wir in diesem Axiom $\succ$ durch $\sim$ ersetzen.

- **Monotonie:** Angenommen, es gibt zwei Lotterien A und B, die die beiden gleichen möglichen Ergebnisse haben. Wenn ein Agent A gegenüber B bevorzugt, dann muss der Agent die Lotterie bevorzugen, die eine höhere Wahrscheinlichkeit für A aufweist (und umgekehrt).

$$A \succ B \Rightarrow (p > q \Leftrightarrow [p, A; 1 - p, B] \succ [q, A; 1 - p, B]$$

- **Zerlegbarkeit:** Zusammengesetzte Lottieren können mithilfe der Wahrscheinlichkeitsgesetze zu einfacheren reduziert werden. Man bezeichnet diese Regel auch als „kein Spaß beim Spielen“, weil sie besagt, dass zwei aufeinanderfolgende Lotterien zu einer einzigen äquivalenten Lotterie komprimiert werden können, wie in ▶ Abbildung 16.1(b) gezeigt.[3]

$$[p, A; 1 - p, [q, B; 1 - q, C]] \sim [p, A; (1 - p)q, B; (1 - p)(1 - q); c, C]$$

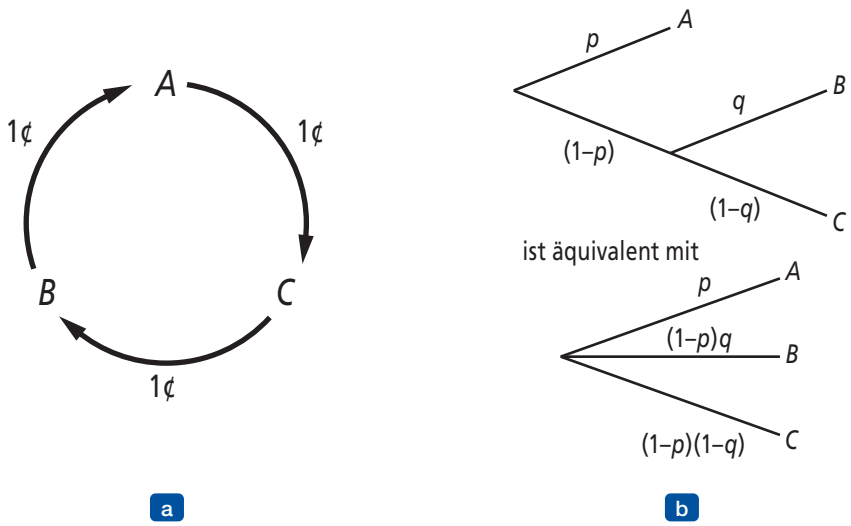


Abbildung 16.1: (a) Ein Tauschzyklus, der zeigt, dass die nicht transitiven Prioritäten $A \succ B \succ C \succ A$ zu einem irrationalen Verhalten führen. (b) Das Zerlegbarkeitsaxiom.

Diese Bedingungen sind als die Axiome der Nutzentheorie bekannt. Jedes Axiom lässt sich begründen, indem man zeigt, dass ein Agent, der es verletzt, offensichtlich irrationales Verhalten in bestimmten Situationen zeigt. Zum Beispiel können wir die Transitivität begründen, indem wir einen Agenten mit nichttransitiven Prioritäten erstellen, der uns sein gesamtes Geld gibt. Nehmen wir an, ein Agent hat die nicht transitiven Prioritäten $A \succ B \succ C \succ A$, wobei A, B und C Waren sind, die kostenlos ausgetauscht werden können. Wenn der Agent momentan A hat, dann könnten wir ihm anbieten, C gegen A plus einen Cent einzutauschen. Der Agent bevorzugt C und willigt ein, den Handel perfekt zu machen. Wir könnten dann anbieten, B gegen C zu tauschen, und wiederum einen Cent

3 Wir können den Spaß beim Spielen auch berücksichtigen, indem wir Spielereignisse in der Zustandsbeschreibung festschreiben; z.B. könnte „Hat 10 € und spielte“ gegenüber „Hat 10 € und spielte nicht“ bevorzugt werden.

verlangen und schließlich A gegen B tauschen. Damit gelangen wir zum Ausgangspunkt zurück, außer dass der Agent uns jetzt drei Cent gegeben hat (▶ Abbildung 16.1(a)). Wir können diesen Zyklus durchlaufen, bis der Agent überhaupt kein Geld mehr hat. Offensichtlich hat der Agent in diesem Fall irrational gehandelt.

16.2.2 Prioritäten führen zu Nutzen

Beachten Sie, dass die Axiome der Nutzentheorie tatsächlich Axiome über Prioritäten sind – sie sagen nichts über eine Nutzenfunktion aus. Allerdings können wir von den Nutzenaxiomen die folgenden Konsequenzen ableiten (für den Beweis siehe von Neumann und Morgenstern, 1944):

- **Existenz der Nutzenfunktion:** Wenn die Prioritäten eines Agenten den Nutzenaxiomen gehorchen, existiert eine Funktion U, sodass $U(A) > U(B)$ genau dann gilt, wenn A gegenüber B bevorzugt wird, und $U(A) = U(B)$ genau dann, wenn der Agent unentschlossen zwischen A und B ist.

$$U(A) > U(B) \Leftrightarrow A \succ B$$
$$U(A) = U(B) \Leftrightarrow A \sim B$$

- **Erwarteter Nutzen einer Lotterie:** Der Nutzen einer Lotterie ist die Summe der Wahrscheinlichkeiten jedes Ergebnisses multipliziert mit dem Nutzen dieses Ergebnisses.

$$U([p_1, S_1; \ldots; p_n, S_n]) = \sum_i p_i U(S_i)$$

Mit anderen Worten, nachdem die Wahrscheinlichkeiten und Nutzen der möglichen Ergebniszustände spezifiziert sind, ist der Nutzen einer zusammengesetzten Lotterie, die diese Zustände beinhaltet, vollständig festgelegt. Weil das Ergebnis einer nicht deterministischen Aktion eine Lotterie ist, folgt daraus, dass ein Agent rational – d.h. konsequent entsprechend seiner Prioritäten – handeln kann, allein indem er eine Aktion auswählt, die den erwarteten Nutzen gemäß Gleichung (16.1) maximiert.

Die obigen Theoreme besagen, dass eine Nutzenfunktion für jeden beliebigen rationalen Agenten *existiert*, doch sagen sie nicht aus, dass sie *eindeutig* ist. Es lässt sich leicht zeigen, dass sich das Verhalten eines Agenten nicht ändert, wenn seine Nutzenfunktion $U(S)$ gemäß

$$U'(S) = aU(S) + b \tag{16.2}$$

transformiert wird, wobei a und b Konstanten sind und $a > 0$ ist – eine affine Transformation.[4] Auf diese Tatsache hat bereits *Kapitel 5* für Zwei-Personen-Spiele hingewiesen; hier sehen wir, dass sie vollkommen allgemeingültig ist.

Wie in Spielen benötigt ein Agent in *deterministischen* Umgebungen lediglich eine Rangfolge von Zuständen – keine numerischen Werte. Man spricht hierbei von einer **Wertfunktion** oder **ordinalen Nutzenfunktion**.

4 In diesem Sinne sind sich Nutzen und Temperaturen ähnlich: Eine Temperatur in Fahrenheit ist das 1,8-fache der Celsiustemperatur plus 32. In jedem Maßsystem erhalten Sie die gleichen Ergebnisse.

Beachten Sie, dass die Existenz einer Nutzenfunktion, die das Prioritätsverhalten eines Agenten beschreibt, nicht unbedingt bedeutet, dass der Agent *explizit* diese Nutzenfunktion nach seinen eigenen Überlegungen maximiert. Wie in *Kapitel 2* beschrieben, kann rationales Verhalten auf unterschiedliche Arten erzeugt werden. Durch Beobachtung der Prioritäten eines rationalen Agenten ist es jedoch möglich, die Nutzenfunktion zu konstruieren, die das repräsentiert, was die Aktionen des Agenten letztlich zu erreichen versuchen (selbst wenn der Agent es nicht weiß).

16.3 Nutzenfunktionen

Nutzen ist eine Funktion, die eine Abbildung von Lotterien auf reelle Zahlen vornimmt. Wir wissen, dass es bestimmte Axiome für Nutzen gibt, denen alle rationalen Agenten gehorchen müssen. Ist das alles, was wir über Nutzenfunktionen sagen können? Genau genommen ja: Ein Agent kann beliebige Prioritäten haben. Beispielsweise könnte ein Agent die Priorität setzen, dass auf seinem Bankkonto nur primzahlige Geldbeträge liegen sollen; wenn er also 16 € hat, würde er 3 € abheben. Das wäre zwar unüblich, doch können wir es nicht als irrational bezeichnen. Ein Agent könnte einen verbeulten Ford Pinto 1973 einem nagelneuen Mercedes vorziehen. Prioritäten können auch verzahnt sein: Beispielsweise könnte er nur dann primzahlige Geldbeträge wünschen, wenn ihm der Pinto gehört, während er, wenn ihm der Mercedes gehört, beliebige Geldbeträge akzeptiert. Glücklicherweise sind die Prioritäten echter Agenten in der Regel systematischer und folglich einfacher in den Griff zu bekommen.

16.3.1 Nutzeneinschätzung und Nutzenskalen

Wenn wir ein entscheidungstheoretisches System erstellen wollen, das den Agenten bei seinen Entscheidungen unterstützt oder in seinem Namen handelt, müssen wir zuerst herausfinden, wie die Nutzenfunktion des Agenten aussieht. In diesem auch als **Prioritätserhebung** bezeichneten Vorgang präsentiert man dem Agenten Auswahlmöglichkeiten und bestimmt anhand der beobachteten Prioritäten die zugrunde liegende Nutzenfunktion.

Gleichung (16.2) besagt, dass es keine absolute Skala für Nutzen gibt. Dennoch ist sie hilfreich, um *irgendeine* Skala einzurichten, auf der sich Nutzen für ein bestimmtes Problem aufzeichnen und vergleichen lassen. Um eine Skala einzurichten, kann man zum Beispiel die Nutzen von zwei bestimmten Ergebnissen als Fixpunkte heranziehen, genau wie eine Temperaturskala durch die Gefrier- und Siedepunkte von Wasser festgelegt wird. In der Regel setzen wir den Nutzen eines „bestmöglichen Preises“ bei $U(S) = u^{\top}$ und einer „schlimmsten möglichen Katastrophe“ bei $U(S) = u^{\bot}$ fest. **Normalisierte Nutzen** verwenden eine Skala mit $u^{\bot} = 0$ und $u^{\top} = 1$.

Anhand der Nutzenskala zwischen $u^{\bot}$ und $u^{\top}$ lässt sich der Nutzen jedes konkreten Preises S bewerten, indem der Agent aufgefordert wird, zwischen S und einer **Standardlotterie** $[p, u^{\top}; (1 - p), u^{\bot}]$ auszuwählen. Die Wahrscheinlichkeit p wird angepasst, bis der Agent zwischen S und der Standardlotterie unentschlossen ist. Bei normalisierten Nutzen ist der Nutzen von S gegeben durch p. Nachdem dies für jeden Preis erfolgt ist, werden die Nutzen für alle Lotterien bestimmt, die mit diesen Preisen zu tun haben.

Tipp

In Entscheidungsproblemen aus beispielsweise Medizin, Transportwesen und Infrastruktur sind Menschenleben beteiligt. In diesen Fällen ist $u_\perp$ der Wert, der dem unmittelbaren Tod zugeordnet ist (oder vielleicht vielen Todesfällen). *Obwohl natürlich niemand gerne ein Menschenleben bewertet, ist es eine Tatsache, dass immer irgendwelche Abwägungen getroffen werden*. Flugzeuge werden abhängig von der Anzahl der Flüge und den geflogenen Meilen vollständig überholt und nicht nach jedem Flug. Autokarosserien werden aus einem relativ dünnen Blech hergestellt, um Kosten zu reduzieren, obwohl die Überlebenszahlen bei Unfällen dadurch sinken. Paradoxerweise bedeutet die Weigerung, „dem Menschenleben einen monetären Wert zuzuordnen", dass es häufig *unterbewertet* wird. Ross Shachter berichtet von einer Erfahrung mit einem Ministerium, das eine Studie über die Entfernung von asbesthaltigen Baustoffen in Schulen in Auftrag gegeben hat. Die Studie ging von einem bestimmten Geldwert für das Leben eines schulpflichtigen Kindes aus und schlussfolgerte, dass die rationale Entscheidung unter dieser Annahme sei, den Asbest zu entfernen. Das Ministerium war moralisch empört und weigerte sich, den Bericht zu veröffentlichen. Anschließend entschied man sich gegen die Entfernung der Asbeststoffe – wodurch man letztendlich das Leben eines Kindes geringer bewertete, als es die Analysten eingeschätzt hatten.

Es wurden Versuche unternommen, den Wert zu ermitteln, den Menschen für ihr eigenes Leben ansetzen. Eine gebräuchliche „Währung", die in der Medizin- und Sicherheitsanalyse verwendet wird, ist der **Mikromort** (eine Todeswahrscheinlichkeit von 1 zu 1 Million). Wenn man Personen befragt, wie viel sie zahlen würden, um ein Risiko zu vermeiden – zum Beispiel das Risiko, Russisches Roulette mit einem Revolver zu spielen, der eine Million Patronen aufnehmen kann –, antworten sie zwar mit recht großen Beträgen, vielleicht zehntausende Euro, doch spiegelt ihr tatsächliches Verhalten einen wesentlich geringeren Geldwert für einen Mikromort wider. Zum Beispiel entspricht das Risiko, 370 km mit dem Auto zu fahren, einem Mikromort, was für die Lebensdauer Ihres Autos – sagen wir 150.000 km – etwa 400 Mikromorts bedeutet. Offenbar sind viele Menschen bereit, etwa $10.000 mehr (bei den Preisen von 2009) für einen sichereren Wagen auszugeben, der die Todeswahrscheinlichkeit halbiert, woraus sich ein Wert von etwa $50 pro Mikromort ableiten lässt. Verschiedene Studien über viele Menschen und Risikotypen haben eine Zahl in diesem Bereich bestätigt. Natürlich trifft dieses Argument nur für kleine Risiken zu. Die meisten Menschen würden sich auch für $50 Millionen nicht selbst umbringen.

Ein weiteres Maß ist das **QALY** (Quality-Adjusted Life Year, äquivalent mit einem Jahr bei guter Gesundheit ohne irgendwelche Gebrechen). Patienten mit einer Behinderung sind bereit, eine kürzere Lebenserwartung zu akzeptieren, wenn sie wieder vollständig geheilt würden. Zum Beispiel sind Nierenpatienten im Durchschnitt unentschlossen, ob sie lieber zwei Jahre an einer Dialysemaschine oder ein Jahr bei voller Gesundheit leben möchten.

16.3.2 Der Nutzen von Geld

Die Nutzentheorie hat ihre Wurzeln in der Betriebswirtschaft, und die Betriebswirtschaft hat nur einen hervorragenden Kandidaten für ein Nutzenmaß: Geld (genauer gesagt, den Gesamtbesitz eines Agenten). Die nahezu uneingeschränkten Möglichkeiten, Geld gegen alle anderen Waren und Dienstleistungen zu tauschen, bewirkt, dass Geld in menschlichen Nutzenfunktionen eine so wichtige Rolle spielt.

Im Normalfall zieht ein Agent mehr Geld weniger Geld vor, wenn alle anderen Dinge gleich sind. Wir sagen, der Agent besitzt eine **monotone Priorität** für mehr Geld. Das heißt aber nicht, dass sich Geld als Nutzenfunktion verhält, weil es nichts über die Prioritäten zwischen *Lottieren* aussagt, bei denen es um Geld geht.

Angenommen, Sie haben sich in einer Spielshow im Fernsehen gegenüber Ihren Mitstreitern durchgesetzt. Jetzt bietet Ihnen der Showmaster eine Wahl an: Entweder Sie nehmen die 1.000.000 € als Preis an oder Sie setzen auf einen Münzwurf. Wenn die Münze auf Kopf zu liegen kommt, bekommen Sie nichts, wenn sie aber auf Zahl zu liegen kommt, erhalten Sie 2.500.000 €. Wenn Sie sich wie die meisten Menschen verhalten, weisen Sie das Angebot zurück und stecken die Million ein. Sind Sie irrational?

Angenommen, Sie gehen von einer fairen Münze aus, dann ist der **erwartete monetäre Wert** (EMW) des Spieles 1/2(0 €) + 1/2(2.500.000 €) = 1.250.000 €, was mehr als die ursprünglichen 1.000.000 € ist. Das bedeutet jedoch nicht, dass die Annahme des Spieles eine bessere Entscheidung darstellt. Angenommen, wir bezeichnen mit S_n den Zustand, insgesamt €n zu besitzen, und dass Ihr aktueller Besitz €k ist. Der erwartete Nutzen der beiden Aktionen, das Spiel zu akzeptieren oder zurückzuweisen, ist also:

$$EU(Akzeptieren) = \frac{1}{2}U(S_k) + \frac{1}{2}U(S_{k+2500000}),$$
$$EU(Ablehnen) = U(S_{k+1000000}).$$

Um zu entscheiden, was zu tun ist, müssen wir den Ergebniszuständen Nutzen zuweisen. Nutzen ist nicht direkt proportional zum monetären Wert, weil der Nutzen für Ihre erste Million sehr hoch ist (wie man so sagt), während der Nutzen einer weiteren Million sehr viel kleiner ist. Angenommen, Sie weisen Ihrem aktuellen finanziellen Zustand den Nutzen 5 zu (S_k), dem Zustand $S_{k+2500000}$ den Nutzen 9 und dem Zustand $S_{k+1000000}$ den Nutzen 8. Die rationale Aktion wäre, das Spiel abzulehnen, weil der erwartete Nutzen des Annehmens nur 7 ist (weniger als die 8 für das Ablehnen). Dagegen dürfte ein Milliardär höchstwahrscheinlich eine Nutzenfunktion haben, die über den Bereich von einigen Millionen mehr lokal linear verläuft, und demzufolge das Spiel annehmen. In einer Pionierstudie über echte Nutzenfunktionen stellte Grayson (1960) fest, dass der Nutzen des Geldes fast proportional zu dem *Logarithmus* des Betrages war. (Dieser Gedanke wurde als Erstes von Bernoulli (1738) angesprochen; siehe Übung 16.3.) Eine bestimmte Nutzenkurve (für einen gewissen Herrn Beard) ist in ▶ Abbildung 16.2(a) gezeigt. Die Daten für die Prioritäten von Herrn Beard sind konsistent mit einer Nutzenfunktion

$$U(S_{k+n}) = -263{,}31 + 22{,}09 \log(n + 150000)$$

für den Bereich zwischen $n = 150000$ € und $n = 800000$ €.

Wir sollten nicht davon ausgehen, dass dies die definitive Nutzenfunktion für monetären Wert ist, aber es ist sehr wahrscheinlich, dass die meisten Menschen eine Nutzenfunktion haben, die konkav für positiven Besitz ist. Schulden zu machen, ist zwar schlecht, doch können die Prioritäten zwischen verschiedenen Schuldenebenen eine Umkehr der dem positiven Besitz zugeordneten Konkavität aufweisen. Wenn jemand beispielsweise bereits 10.000.000 € im Minus ist, akzeptiert er ein Spiel mit einer fairen Münze, das bei Kopf einen Gewinn von 10.000.000 € und bei Zahl einen Verlust von 20.000.000 € vorgibt.[5] Daraus entsteht die in ▶ Abbildung 16.2(b) gezeigte S-förmige Kurve.

5 Ein solches Verhalten könnte als hoffnungslos bezeichnet werden, aber es ist rational, wenn man sich bereits in einer hoffnungslosen Situation befindet.

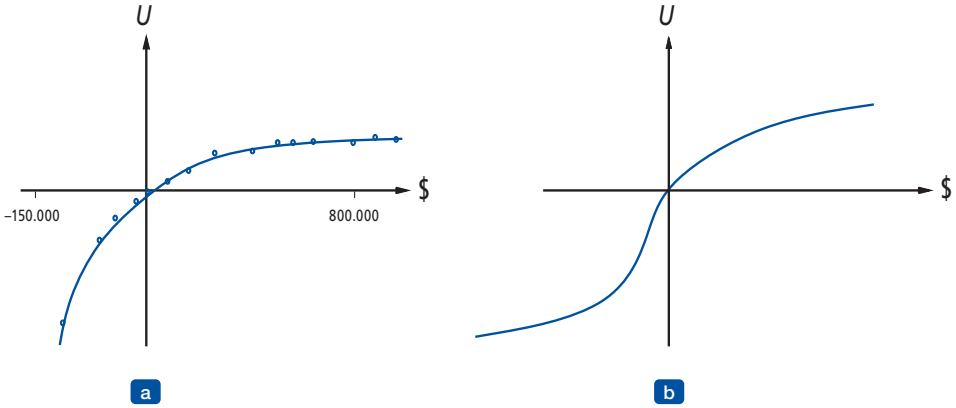


Abbildung 16.2: Der Nutzen des Geldes. (a) Empirische Daten für Herrn Beard über einen begrenzten Bereich. (b) Eine typische Kurve für den vollständigen Bereich.

Wenn wir unsere ganze Aufmerksamkeit auf den positiven Teil der Kurven richten, wo die Steigung abnimmt, dann ist für eine beliebige Lotterie L der Nutzen, an dieser Lotterie teilzunehmen, geringer als der Nutzen, den erwarteten monetären Wert der Lotterie als sicher zu erhalten:

$$U(L) < U(S_{EMW(L)}).$$

Das bedeutet, Agenten mit Kurven dieser Form sind **risikoscheu**: Sie bevorzugen die Sicherheit mit einer Rendite, die kleiner als der erwartete monetäre Gewinn eines Spieles ist. Im „hoffnungslosen" Bereich dagegen bei einem großen negativen Besitz in Abbildung 16.2(b) ist das Verhalten **risikofreudig**. Der Wert, den ein Agent anstelle einer Lotterie akzeptiert, wird auch als das **Sicherheitsäquivalent** der Lotterie bezeichnet. Studien haben gezeigt, dass die meisten Menschen etwa ein Spiel um 400 € statt eines Spieles, das zur einen Hälfte 1000 € und zur anderen Hälfte 0 € bringt, akzeptieren – d.h., das Sicherheitsäquivalent der Lotterie ist 400 €. Die Differenz zwischen dem erwarteten monetären Wert einer Lotterie und ihrem Sicherheitsäquivalent wird als **Versicherungsprämie** bezeichnet. Die Risikoscheu ist die Grundlage für die Versicherungsbranche, weil es bedeutet, dass Versicherungsprämien positiv sind. Menschen würden lieber eine kleine Versicherungsprämie zahlen, als um den Preis ihres Hauses gegen die Wahrscheinlichkeit eines Feuers zu spielen. Aus der Perspektive der Versicherungsgesellschaft ist der Preis des Hauses relativ gering im Vergleich zu den Gesamtrücklagen des Unternehmens. Das bedeutet, die Nutzenkurve des Versicherers ist annähernd linear über einen solch kleinen Bereich und das „Spiel" kostet das Unternehmen fast nichts.

Beachten Sie, dass für *kleine* Änderungen des Besitzes im Vergleich zum aktuellen Besitz fast jede Kurve annähernd linear ist. Ein Agent, der eine lineare Kurve hat, wird als **risikoneutral** bezeichnet. Für Spiele um kleine Summen erwarten wir deshalb Risikoneutralität. In gewisser Weise rechtfertigt dies die vereinfachte Prozedur in *Abschnitt 13.2.3*, die kleine Spiele vorschlug, um Wahrscheinlichkeiten einzuschätzen und die Wahrscheinlichkeitsaxiome zu begründen.

16.3.3 Erwarteter Nutzen und Enttäuschung nach einer Entscheidung

Die beste Aktion a^* lässt sich rational am besten auswählen, wenn man den erwarteten Nutzen maximiert:

$$a^* = \arg\max_a EU(a \mid \mathbf{e}).$$

Wenn wir den erwarteten Nutzen korrekt entsprechend unserem Wahrscheinlichkeitsmodell berechnet haben und das Wahrscheinlichkeitsmodell die zugrunde liegenden stochastischen Prozesse, die die Ergebnisse erzeugen, richtig widerspiegelt, erhalten wir im Durchschnitt den Nutzen, den wir erwarten, wenn der gesamte Prozess viele Male wiederholt wird.

In der Praxis vereinfacht unser Modell die tatsächliche Situation normalerweise zu stark, weil wir entweder nicht genug wissen (wenn z.B. eine komplexe Investitionsentscheidung zu treffen ist) oder weil die Berechnung des wahren erwarteten Nutzens zu schwierig ist (wenn z.B. im Backgammon der Nutzen von Nachfolgerzuständen ausgehend vom Wurzelknoten abzuschätzen ist). In diesem Fall arbeiten wir tatsächlich mit Schätzungen $\widehat{EU}(a|\mathbf{e})$ des wahren erwarteten Nutzens. Wir nehmen großzügig an, dass die Schätzungen **unverzerrt** sind, d.h. der erwartete Wert des Fehlers $E\left(\widehat{EU}(a|\mathbf{e}) - EU(a|\mathbf{e})\right)$ null ist. Dann scheint es trotzdem noch vernünftig, die Aktion mit dem höchsten erwarteten Nutzen auszuwählen und davon auszugehen, durchschnittlich diesen Nutzen zu erhalten, wenn die Aktion ausgeführt wird.

Leider wird das wirkliche Ergebnis normalerweise beträchtlich *schlechter* sein als erwartet, selbst wenn die Schätzung unverzerrt war! Um die Gründe dafür zu ermitteln, betrachten wir ein Entscheidungsproblem, in dem es k Auswahlmöglichkeiten gibt, die jeweils einen wahren geschätzten Nutzen von 0 haben. Nehmen Sie an, dass der Fehler in jeder Nutzenschätzung einen Erwartungswert von 0 und eine Standardabweichung von 1 hat, wie es die dicke Kurve in ▶ Abbildung 16.3 zeigt. Wenn wir nun die Schätzungen erzeugen, werden einige Fehler negativ (pessimistische Schätzungen) und einige positiv (optimistische Schätzungen) sein. Da wir die Aktion mit der höchsten Nutzenschätzung auswählen, bevorzugen wir offensichtlich die übermäßig optimistischen Schätzungen – und dies ist die Quelle für die Verzerrung (engl. Bias). Es ist recht einfach, die Verteilung des Maximums der k Schätzungen zu berechnen (siehe Übung 16.10) und folglich das Ausmaß unserer Enttäuschung zu quantifizieren. In Abbildung 16.3 hat die Kurve für $k = 3$ einen Erwartungswert von 0,85, sodass die durchschnittliche Enttäuschung bei 85% der Standardabweichung in den Nutzenschätzungen liegen wird.

Bei mehr Auswahlmöglichkeiten tauchen äußerst optimistische Schätzungen wahrscheinlicher auf: Für $k = 30$ ist die Enttäuschung etwa das Doppelte der Standardabweichung in den Schätzungen.

Diese Tendenz, dass der geschätzte erwartete Nutzen der besten Auswahl zu hoch ist, wird als **Fluch des Optimierers** (optimizer's curse, Smith und Winkler, 2006) bezeichnet. Er sucht selbst erfahrenste Analysten und Statistiker heim. So wurde zum Beispiel ernsthaft geglaubt, dass ein attraktives neues Medikament, das in einem Versuch 80% der Patienten geheilt hat, tatsächlich 80% der Patienten heilen würde (es wurde aus k = Tausenden von Kandidatendrogen ausgewählt) oder dass eine Kapitalanlage, für die mit einer überdurchschnittlichen Rendite geworben wird, diese Rendite auch beibehält (sie wurde ausgewählt, um in der Werbung von k = Dutzenden Fonds im

Gesamtportfolio der Firma zu erscheinen). Es ist auch durchaus möglich, dass sich die beste Wahl gar nicht als die beste erweist, wenn nämlich die Varianz in der Nutzenschätzung hoch ist: Ein Medikament, das aus Tausenden Proben ausgewählt wird und 9 von 10 Patienten geheilt hat, ist möglicherweise schlechter, als ein Medikament, das 800 von 1000 geholfen hat.

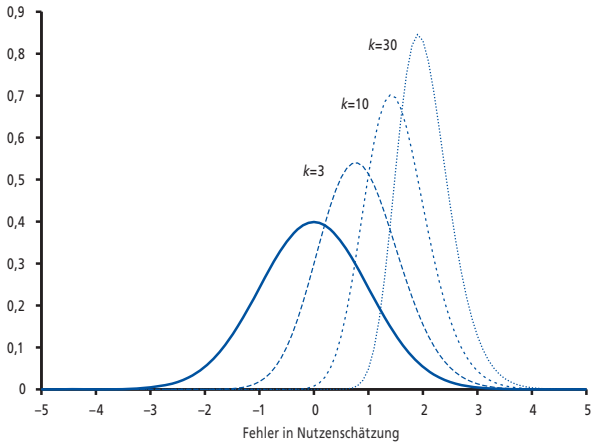


Abbildung 16.3: Verlauf des Fehlers in jeder der k Nutzenschätzungen und der Verteilung des Maximums von k Schätzungen für $k = 3, 10$ und 30.

Der Fluch des Optimierers zeigt sich überall, da die Auswahlprozesse zur Nutzenmaximierung allgegenwärtig sind. Es empfiehlt sich deshalb nicht, Nutzenschätzungen für bare Münze zu nehmen. Wir können dem Fluch ausweichen, wenn wir ein explizites Wahrscheinlichkeitsmodell $\mathbf{P}(\widehat{EU}|EU)$ des Fehlers in den Nutzenschätzungen verwenden. Mit diesem Modell und einer A-priori-Wahrscheinlichkeit $\mathbf{P}(EU)$ für das, was wir vernünftigerweise als Nutzen erwarten, behandeln wir die ermittelte Nutzenschätzung als Evidenz und berechnen die A-posteriori-Verteilung für den wahren Nutzen mithilfe der Bayesschen Regel.

16.3.4 Menschliches Urteilsvermögen und Irrationalität

Die Entscheidungstheorie ist eine **normative Theorie**: Sie beschreibt, wie ein rationaler Agent handeln *sollte*. Dagegen gibt eine **beschreibende Theorie** an, wie reale Agenten – zum Beispiel Menschen – wirklich handeln. Die Anwendung von wirtschaftswissenschaftlichen Theorien ließe sich erheblich ausdehnen, wenn beide zusammenfielen. Offenbar gibt es aber experimentelle Anzeichen, die eher für das Gegenteil sprechen. Diese legen nahe, dass Menschen „voraussagbar irrational“ sind (Ariely, 2009).

Das bekannteste Problem ist das Allais-Paradoxon (Allais, 1953). Dabei können die Teilnehmer an einem Experiment zunächst zwischen den Lotterien A und B und dann zwischen C und D auswählen, die die folgenden Preise haben:

A: 80% Wahrscheinlichkeit von 4000 €

B: 100% Wahrscheinlichkeit von 3000 €

C: 20% Wahrscheinlichkeit von 4000 €

D: 25% Wahrscheinlichkeit von 3000 €

Die meisten Testpersonen wählten B vor A (um das sichere Ergebnis zu nehmen) und C vor D (um den höheren EMV zu nehmen). Die normative Analyse stimmt damit nicht überein! Wir sehen dies am einfachsten, wenn wir uns die durch Gleichung (16.2) implizierte Freiheit nehmen, um $U(0\,€) = 0$ zu setzen. In diesem Fall impliziert dann $B \succ A$, dass $U(3000\,€) > 0{,}8U(4000\,€)$ ist, während $C \succ D$ genau das Gegenteil impliziert. Mit anderen Worten, scheint es keine Nutzenfunktion zu geben, die mit diesen Auswahlen konsistent ist. Eine Erklärung für die scheinbar unvernünftigen Bevorzugungen ist der **Sicherheitseffekt** (Kahneman und Tversky, 1979): Menschen werden stark von Gewinnen angezogen, die sicher sind. Hierfür gibt es mehrere mögliche Gründe. Erstens könnten die Teilnehmer ihren Rechenaufwand gering halten wollen – indem sie sichere Ergebnisse wählen, brauchen sie keine Wahrscheinlichkeiten zu berechnen. Doch bleibt dieser Effekt selbst dann erhalten, wenn die Berechnungen sehr leicht sind. Zweitens gibt es vielleicht Zweifel an der Rechtmäßigkeit der angegebenen Wahrscheinlichkeiten. Ich vertraue darauf, dass ein Münzwurf ein Ergebnis von etwa 50/50 bringt, wenn ich die Kontrolle über die Münze und den Wurf habe, doch zweifle ich gegebenenfalls am Ergebnis, wenn jemand anderes mit einem persönlichen Interesse am Ergebnis den Wurf ausführt.[6] Wenn Misstrauen im Spiel ist, kann es besser sein, die sichere Variante zu wählen.[7] Drittens könnten die Teilnehmer ihren Gemütszustand sowie ihre finanzielle Lage in die Entscheidung einfließen lassen. Die Leute wissen, dass sie es **bedauern** werden, wenn sie einen sicheren Preis (B) zugunsten einer 80%igen Chance auf einen höheren Preis aufgeben und dann verlieren. Mit anderen Worten: Wird A gewählt, gibt es eine 20%ige Chance, kein Geld zu erhalten *und sich wie ein Vollidiot zu fühlen*, was noch schlimmer ist, als einfach kein Geld zu erhalten. Vielleicht sind also die Teilnehmer, die B gegenüber A und C gegenüber D bevorzugen, doch nicht irrational; sie sagen lediglich, dass sie 200 € EMW aufgeben, um eine 20%ige Chance zu vermeiden, sich wie ein Idiot zu fühlen.

Ein verwandtes Problem ist das Ellsberg-Paradoxon. Hier stehen die Preise fest, jedoch sind die Wahrscheinlichkeiten nicht vollständig bekannt. Die Auszahlung hängt von der Farbe eines Balles ab, der aus einer Urne gezogen wird. Es wird Ihnen mitgeteilt, dass die Urne 1/3 rote Bälle und 2/3 schwarze und gelbe Bälle enthält, wobei aber nicht bekannt ist, wie viele schwarze und gelbe Bälle. Auch hier werden Sie aufgefordert, Lotterie A oder B und dann C oder D zu bevorzugen:

A : 100 € für einen roten Ball

B : 100 € für einen schwarzen Ball

C : 100 € für einen roten oder gelben Ball

D : 100 € für einen schwarzen oder gelben Ball

Es dürfte klar sein, dass Sie A gegenüber B und C gegenüber D bevorzugen sollten, wenn Sie annehmen, dass mehr rote als schwarze Bälle vorhanden sind, und dass Sie sich entgegengesetzt entscheiden, wenn Sie von weniger roten als schwarzen Bällen ausgehen. Es stellt sich aber heraus, dass die meisten Versuchspersonen A gegenüber B und auch D gegenüber C bevorzugen, selbst wenn es keinen Zustand der Welt gibt,

6 Zum Beispiel ist der Mathematiker und Zauberer Persi Diaconis in der Lage, bei jedem Münzwurf das von ihm gewünschte Ergebnis zu erhalten (Landhuis, 2004).

7 Selbst die sichere Variante muss nicht unbedingt sicher sein. Trotz eiserner Versprechungen haben wir immer noch nicht die $27.000.000 vom Nigerianischen Bankkonto eines kürzlich verstorbenen unbekannten Verwandten erhalten.

für den dies vernünftig ist. Offenbar haben Menschen eine **Unsicherheitsaversion**: *A* bietet Ihnen eine 1/3-Chance auf den Gewinn, während *B* zwischen 0 und 2/3 liegen kann. Analog bietet *D* eine 2/3-Gewinnchance, während *C* zwischen 1/3 und 3/3 liegen kann. Die meisten Personen entscheiden sich für das bekannte Risiko, anstatt für die unbekannten Unwägbarkeiten.

Auch die genaue Formulierung eines Entscheidungsproblems kann großen Einfluss auf die Auswahl des Agenten haben – das ist der sogenannte **Framing-Effekt** (dt. etwa Einrahmungseffekt). Experimente zeigen, dass ein medizinisches Verfahren, das mit einer „90%igen Überlebensrate" angepriesen wird, doppelt so viel Zuspruch findet wie ein Verfahren, das mit einer „10%igen Sterberate" wirbt, selbst wenn beide Aussagen genau das Gleiche meinen. Diese Diskrepanz in der Beurteilung ist in mehreren Experimenten ermittelt worden, wobei es kaum eine Rolle spielt, ob die Versuchspersonen Patienten in einer Klinik, statistisch gebildete Studenten einer Handelsschule oder erfahrene Doktoren sind.

Menschen sind vertrauter damit, relative Nutzenbeurteilungen zu treffen statt absolute. Ich habe kaum eine Vorstellung davon, wie sehr ich die verschiedenen Weine mag, die in einem Restaurant angeboten werden. Das Restaurant nutzt dies aus, indem es eine $200-Flasche anbietet, die ohnehin niemand kaufen wird, die aber dazu dient, die Beurteilung des Gastes über den Wert aller Weine hochzuschrauben und die $55-Flasche wie ein Schnäppchen aussehen zu lassen. Hierbei handelt es sich um den sogenannten **Ankereffekt**.

Wenn menschliche Informanten auf widersprüchlichen Prioritätsbeurteilungen bestehen, gibt es nichts, was automatisierte Agenten tun können, um damit konsistent zu sein.

Glücklicherweise werden die Prioritätsbeurteilungen von Menschen angesichts weiterer Überlegungen häufig überdacht. Paradoxa wie das Allais-Paradoxon werden deutlich reduziert (jedoch nicht beseitigt), wenn die Auswahlmöglichkeiten besser erklärt werden. In einer Arbeit an der Harvard Business School zur Einschätzung des Nutzens von Geld haben Keeney und Raiffa (1976, S. 210) das Folgende festgestellt:

> *Testpersonen tendieren dazu, im Kleinen risikoscheu zu sein, und deshalb ... weisen die passenden Nutzenfunktionen unakzeptabel große Risikoprämien für Lotterien mit großer Bandbreite auf. ... Die meisten Testpersonen können jedoch ihre Inkonsistenzen ausgleichen und haben das Gefühl, etwas Wichtiges darüber gelernt zu haben, wie sie sich verhalten wollen. Demzufolge kündigen einige Testpersonen ihre Autovollkaskoversicherung und erhöhen stattdessen ihre Lebensversicherung.*

Der Beweis für menschliche Irrationalität wird auch von Forschern auf dem Gebiet der **evolutionären Psychologie** hinterfragt. Sie weisen auf die Tatsache hin, dass sich die Mechanismen unseres Gehirns zur Entscheidungsfindung nicht dahingehend entwickelt haben, um Textaufgaben zu lösen, in denen Wahrscheinlichkeiten und Preise als Dezimalzahlen ausgedrückt werden.

Nehmen wir einmal an, dass das Gehirn integrierte Neuromechanismen für das Rechnen mit Wahrscheinlichkeiten und Nutzen oder irgendein funktionelles Äquivalent besitzt. Wenn dies der Fall ist, würden die erforderlichen Eingaben über gesammelte Erfahrung von Ergebnissen und Belohnungen statt über sprachliche Darstellungen von Zahlenwerten erhalten. Es ist keineswegs selbstverständlich, dass wir direkt auf die Neuromechanismen des Gehirns zugreifen können, indem wir Entscheidungsprobleme in sprachlicher/numerischer Form präsentieren. Allein die Tatsache, dass verschiedene

Formulierungen des *gleichen Entscheidungsproblems* zu unterschiedlichen Auswahlen führen, legt nahe, dass das Entscheidungsproblem selbst nicht wirklich ankommt.

Angespornt durch diese Beobachtung haben Psychologen versucht, Probleme in unsicherem Schließen und Entscheidungsfindung in „evolutionär geeigneten“ Formen darzustellen. Anstatt zum Beispiel zu sagen „90%ige Überlebensrate“ könnte der Experimentator 100 Trickfilme der Operation zeigen, wobei der Patient in 10 von ihnen stirbt und in 90 überlebt. (Langeweile ist ein erschwerender Faktor in diesen Experimenten!) Mit Entscheidungsproblemen, die auf diese Weise aufgestellt werden, scheinen Menschen wesentlich näher am rationalen Verhalten zu sein als vorher vermutet.

16.4 Nutzenfunktionen mit Mehrfachattributen

Bei der Entscheidungsfindung im Bereich der Politik geht es sowohl um Millionen von Euro als auch um Leben und Tod. Wenn beispielsweise entschieden wird, welche Mengen schädlicher Emissionen von einem Kraftwerk in die Umgebung abgegeben werden dürfen, müssen die Politiker die Vorbeugung gegen Todesfälle und Behinderungen gegen den Nutzen des Kraftwerkes und den wirtschaftlichen Aufwand zur Verminderung der Emissionen abwägen. Bei der Erstellung eines neuen Flughafens muss berücksichtigt werden, welche Probleme der Bau verursacht, welche Kosten für das Bauland anfallen, welcher Abstand zu Ballungsgebieten vorliegt, welchen Lärm der Flugbetrieb verursacht, welche Sicherheitsrisiken aus der lokalen Topografie und den Wetterbedingungen entstehen usw. Probleme wie diese, deren Ergebnisse durch zwei oder mehr Attribute charakterisiert sind, werden von der **Nutzentheorie mit Mehrfachattributen** verarbeitet.

Wir bezeichnen diese Attribute als $\mathbf{X} = X_1, ..., X_n$. Ein vollständiger Zuweisungsvektor ist dann $\mathbf{x} = \langle x_1, ..., x_n \rangle$, wobei jedes x_i entweder ein numerischer Wert oder ein diskreter Werte mit einer angenommenen Sortierung nach Werten ist. Wir gehen davon aus, dass höhere Werte eines Attributes höheren Nutzen entsprechen. Wenn wir beispielsweise *KeinLärm* als Attribut des Flughafenproblems wählen, dann ist die Lösung umso besser, je höher dieser Wert ist.[8] Wir betrachten zunächst Fälle, in denen Entscheidungen getroffen werden können, *ohne* dass die Attributwerte zu einem einzigen Nutzenwert kombiniert werden müssen. Anschließend betrachten wir Fälle, wo die Nutzen von Attributkombinationen sehr präzise spezifiziert werden können.

16.4.1 Dominanz

Angenommen, der Flughafenstandort S_1 kostet weniger, verursacht weniger Lärmemissionen und ist sicherer als der Standort S_2. Man würde ohne Zögern S_2 zurückweisen. Wir sagen dann, es besteht eine **strenge Dominanz** von S_1 gegenüber S_2. Wenn eine Option für alle Attribute einen niedrigeren Wert als eine andere Option hat, muss sie im Allgemeinen nicht weiter berücksichtigt werden.

8 In manchen Fällen kann es erforderlich sein, den Wertebereich zu unterteilen, sodass der Nutzen innerhalb jedes Bereiches monoton variiert. Hat zum Beispiel das Attribut *RaumTemperatur* eine Nutzenspitze bei 21 °C, teilen wir es in zwei Attribute auf, die den Unterschied gegenüber dem Idealzustand messen, eines für kältere und eines für wärmere Werte. Der Nutzen nimmt dann in jedem Attribut monoton zu.

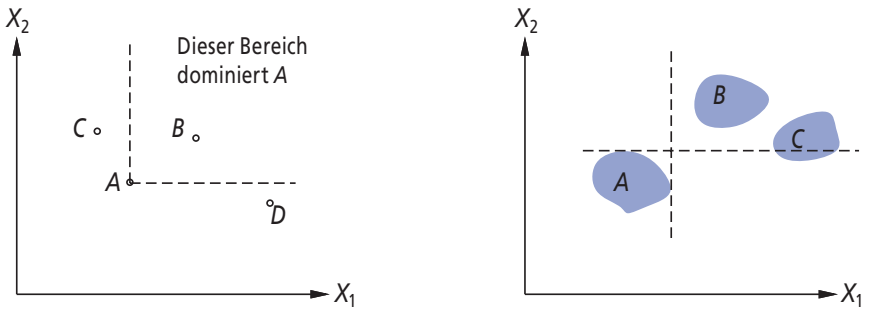


Abbildung 16.4: Strenge Dominanz. (a) Deterministisch: Option A ist streng dominiert von B, aber nicht von C oder D. (b) Unsicher: A ist streng von B, aber nicht von C dominiert.

Die strenge Dominanz ist häufig sehr praktisch, um den Auswahlbereich auf die wirklich in Frage kommenden Kandidaten einzugrenzen, obwohl sich dabei selten eine eindeutige Auswahl zeigt. ▶ Abbildung 16.4(a) zeigt eine schematische Darstellung für den Fall mit zwei Attributen.

Das ist für den deterministischen Fall in Ordnung, wo die Attributwerte bekannt sind. Was aber ist mit dem allgemeinen Fall, wo die Aktionsergebnisse unsicher sind? Es kann eine direkte Entsprechung zur strengen Dominanz erzeugt werden, wo trotz der Unsicherheit alle möglichen konkreten Ergebnisse für S_1 alle möglichen Ergebnisse für S_2 streng dominieren (siehe ▶ Abbildung 16.4(b)). Natürlich tritt dies weniger häufig auf als im deterministischen Fall.

Glücklicherweise gibt es eine praktischere Verallgemeinerung, die sogenannte **stochastische Dominanz**, die in realen Problemstellungen sehr häufig auftritt. Die stochastische Dominanz versteht man am einfachsten im Kontext eines einzigen Attributes. Angenommen, wir glauben, die Kosten für den Bau des Flughafens am Standort S_1 sind einheitlich zwischen 2,8 Milliarden € und 4,8 Milliarden € verteilt und die Kosten am Standort S_2 sind einheitlich zwischen 3 Milliarden € und 5,2 Milliarden € verteilt. ▶ Abbildung 16.5(a) zeigt diese Verteilungen, wobei die Kosten als negativer Wert dargestellt sind. Wenn man nur die Information besitzt, dass der Nutzen bei steigenden Kosten sinkt, können wir sagen, dass S_1 den Standort S_2 stochastisch dominiert (d.h. S_2 kann verworfen werden). Beachten Sie, dass dies *nicht* aus dem Vergleich der erwarteten Kosten folgt. Wenn wir beispielsweise wissen, dass die Kosten für S_1 *genau* 3,8 Milliarden € betragen, dann könnten wir *keine* Entscheidung treffen, hätten wir nicht eine zusätzliche Information über den Nutzen des Geldes. (Es erscheint vielleicht seltsam, dass *mehr* Informationen über die Kosten von S_1 den Agenten *weniger* gut entscheiden lassen. Dieses Paradoxon löst sich auf, wenn man erkennt, dass eine Entscheidung beim Fehlen genauer Kosteninformationen zwar leichter zu treffen ist, aber wahrscheinlicher falsch sein wird.)

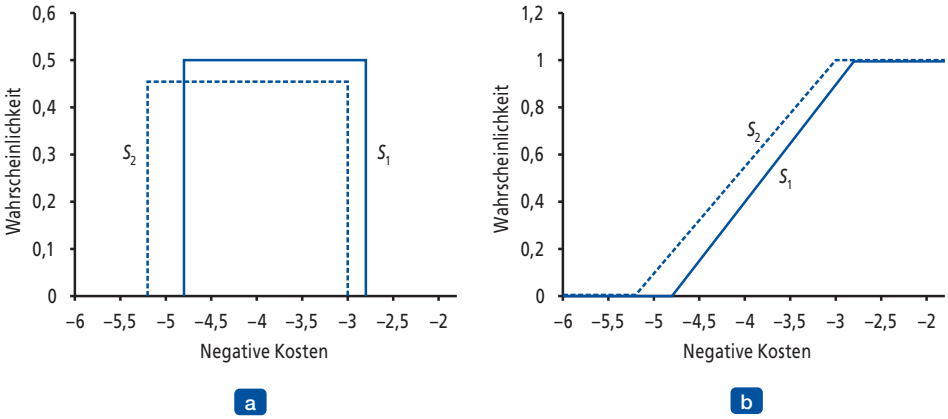


Abbildung 16.5: Stochastische Dominanz. (a) S_1 zeigt stochastische Dominanz gegenüber S_2 im Hinblick auf die Kosten. (b) Kumulative Verteilungen für die negativen Kosten von S_1 und S_2.

Die genaue Beziehung zwischen den Attributverteilungen, die man braucht, um eine stochastische Dominanz einzurichten, erkennt man am besten, indem man die in ▶ Abbildung 16.5(b) gezeigten **kumulativen Verteilungen** betrachtet (siehe auch Anhang A). Die kumulative Verteilung misst die Wahrscheinlichkeit, dass die Kosten kleiner oder gleich einem bestimmten Betrag sind – d.h., sie integriert die ursprüngliche Verteilung. Wenn die kumulative Verteilung für S_1 immer rechts von der kumulativen Verteilung für S_2 ist, dann ist (stochastisch betrachtet) S_1 billiger als S_2. Formal ausgedrückt, wenn zwei Aktionen A_1 und A_2 zu Wahrscheinlichkeitsverteilungen $p_1(x)$ und $p_2(x)$ für das Attribut X führen, dann dominiert A_1 die Aktion A_2 stochastisch für X, wenn

$$\forall x \int_{-\infty}^{x} p_1(x')dx' \leq \int_{-\infty}^{x} p_2(x')dx'.$$

Tipp

Die Bedeutung dieser Definition für die Auswahl optimaler Entscheidungen ergibt sich aus der folgenden Eigenschaft: *Wenn A_1 stochastisch A_2 dominiert, dann ist für jede monoton nicht fallende Nutzenfunktion $U(x)$ der erwartete Nutzen von A_1 mindestens so hoch wie der erwartete Nutzen für A_2*. Wenn also eine Aktion stochastisch von einer anderen Aktion für alle Attribute dominiert wird, dann kann sie verworfen werden.

Die stochastische Dominanzbedingung scheint vielleicht eher technisch und ohne umfangreiche Wahrscheinlichkeitsberechnungen nicht ganz leicht auszuwerten zu sein. In vielen Fällen kann sie jedoch ganz einfach entschieden werden. Nehmen Sie beispielsweise an, dass die Transportkosten für einen Bau von der Entfernung zum Lieferanten abhängig sind. Die eigentlichen Kosten sind unsicher, aber je höher die Entfernung ist, desto höher sind die Kosten. Wenn S_1 eine geringere Entfernung als S_2 aufweist, dann dominiert S_1 den Standort S_2 im Hinblick auf die Kosten. Es gibt Algorithmen für die Propagation dieser Art qualitativer Informationen unter unsicheren Variablen in **qualitativen probabilistischen Netzen**, sodass ein System rationale Entscheidungen basierend auf stochastischer Dominanz treffen kann, ohne dazu irgendwelche numerischen Werte zu verwenden. Wir werden diese Algorithmen jedoch hier nicht vorstellen.

16.4.2 Prioritätsstruktur und Nutzen mit Mehrfachattributen

Angenommen, wir haben n Attribute, die jeweils d verschiedene mögliche Werte annehmen können. Um die vollständige Nutzenfunktion $U(x_1, ..., x_n)$ zu spezifizieren, brauchen wir im schlechtesten Fall d^n Werte. Der schlechteste Fall entspricht einer Situation, wo die Prioritäten des Agenten überhaupt keine Regelmäßigkeit aufweisen. Die Nutzentheorie mit Mehrfachattributen basiert auf der Annahme, dass die Prioritäten typischer Agenten sehr viel strukturierter sind. Der grundlegende Ansatz ist, Regelmäßigkeiten im Prioritätsverhalten, das wir erwarten, festzustellen und die sogenannten **Repräsentationstheoreme** zu verwenden, um zu zeigen, dass ein Agent mit einer bestimmten Art von Prioritätsstruktur eine Nutzenfunktion

$$U(x_1, ..., x_n) = F[f_1(x_1, ..., f_n(x_n)]$$

hat, wobei F (wie wir hoffen) eine einfache Funktion wie etwa eine Addition ist. Beachten Sie die Ähnlichkeit mit der Verwendung Bayesscher Netze, um die gemeinsame Wahrscheinlichkeit mehrerer Zufallsvariablen zu zerlegen.

Prioritäten ohne Unsicherheit

Wir beginnen mit dem deterministischen Fall. Sie wissen, dass der Agent für deterministische Umgebungen eine Wertfunktion $V(x_1, ..., x_n)$ besitzt; das Ziel dabei ist, diese Funktion präzise darzustellen. Die grundlegende Regelmäßigkeit, die in deterministischen Prioritätsstrukturen auftritt, wird als **Prioritätsunabhängigkeit** bezeichnet. Zwei Attribute, X_1 und X_2, sind prioritätsunabhängig von einem dritten Attribut, X_3, wenn die Priorität zwischen den Ergebnissen $\langle x_1, x_2, x_3\rangle$ und $\langle x_1', x_2', x_3\rangle$ nicht vom jeweiligen Wert von x_3 für das Attribut X_3 abhängig ist.

Zurück zum Flughafenbeispiel. Hier haben wir (neben anderen Attributen) *Lärm*, *Kosten* und *Todesfälle* zu berücksichtigen. Man könnte sagen, dass *Lärm* und *Kosten* prioritätsunabhängig von *Todesfällen* sind. Wenn wir beispielsweise einen Zustand mit 20.000 Menschen in der Flugschneise und Baukosten von 4 Milliarden € einem Zustand mit 70.000 Menschen in der Flugschneise und Kosten von 3,7 Milliarden € vorziehen, wobei die Sicherheitsstufe in beiden Fällen bei 0,06 Todesfällen pro Millionen Passagiermeilen liegt, hätten wir dieselbe Priorität, wenn die Sicherheitsstufe bei 0,12 oder bei 0,03 läge; dieselbe Unabhängigkeit würde für Prioritäten zwischen jedem anderen Wertepaar für *Lärm* und *Kosten* gelten. Es ist ebenso offensichtlich, dass *Kosten* und *Todesfälle* prioritätsunabhängig von *Lärm* und dass *Lärm* und *Todesfälle* prioritätsunabhängig von *Kosten* sind. Wir sagen, die Attributmenge {*Lärm*, *Kosten*, *Todesfälle*} weist eine **gegenseitige Prioritätsunabhängigkeit** (GPU) auf. GPU besagt, dass jedes Attribut zwar wichtig sein kann, es aber keinen Einfluss darauf hat, wie man die anderen Attribute gegeneinander abwägt.

Tipp

Gegenseitige Prioritätsunabhängigkeit ist ein großes Wort, aber dank eines bemerkenswerten Theorems des Wirtschaftswissenschaftlers Gérard Debreu (1960) können wir daraus eine sehr einfache Form für die Wertfunktion des Agenten ableiten: *Wenn die Attribute X_1, ..., X_n gegenseitig prioritätsunabhängig sind, kann das Prioritätsverhalten des Agenten so beschrieben werden, dass es die Funktion*

$$V(x_1,...,x_n) = \sum_i V_i(x_i)$$

maximiert, wobei jedes V_i eine Wertfunktion ist, die nur auf das Attribut X_i verweist. Beispielsweise könnte es der Fall sein, dass die Flughafenentscheidung anhand der folgenden Wertfunktion getroffen wird:

$$V(lärm, kosten, todesfälle) = -lärm \times 10^4 - kosten - todesfälle \times 10^{12}.$$

Eine Wertfunktion dieser Art wird als **additive Wertfunktion** bezeichnet. Additive Funktionen sind eine sehr natürliche Art und Weise, die Prioritäten eines Agenten zu beschreiben, und gelten in vielen Situationen der realen Welt. Bei n Attributen ist es für die Bewertung einer additiven Wertfunktion erforderlich, n separate eindimensionale Wertfunktionen statt lediglich einer n-dimensionalen Funktion zu bewerten; normalerweise stellt dies eine exponentielle Verringerung in der Anzahl der notwendigen Prioritätsexperimente dar. Selbst wenn die gegenseitige Prioritätsunabhängigkeit nicht streng gültig ist, wie es bei Extremwerten von Attributen der Fall sein könnte, kann eine additive Wertfunktion dennoch eine gute Annäherung an die Prioritäten des Agenten liefern. Das gilt vor allem dann, wenn die Verletzungen der GPU in Teilen der Attributbereiche auftreten, die in der Praxis sehr wahrscheinlich nicht vorkommen.

GPU lässt sich besser verstehen, wenn man Fälle betrachtet, in denen sie nicht gilt. Angenommen, Sie befinden sich auf einem mittelalterlichen Markt und haben vor, einige Jagdhunde, einige Hühner und einige Käfige für die Hühner zu kaufen. Die Jagdhunde sind sehr wertvoll, doch wenn Sie nicht genügend Käfige für die Hühner haben, fressen die Hunde die Hühner auf. Folglich hängt der Kompromiss zwischen Hunden und Hühnern streng von der Anzahl der Käfige ab und GPU ist verletzt. Derartige Interaktionen zwischen verschiedenen Attributen erschweren es erheblich, die gesamte Wertfunktion zu bewerten.

Prioritäten mit Unsicherheit

Wenn in einer Domäne Unsicherheit herrscht, müssen wir auch die Prioritätsstruktur zwischen Lotterien berücksichtigen und die resultierenden Eigenschaften der Nutzenfunktionen verstehen, nicht nur die Wertfunktionen. Die Mathematik für ein solches Problem kann relativ kompliziert werden; deshalb werden wir nur eines der wichtigsten Ergebnisse vorstellen, um Ihnen ein Gefühl dafür zu geben, was möglich ist. Eine detailliertere Beschreibung dieses Themas finden Sie bei Keeney und Raiffa (1976).

Das grundlegende Konzept der **Nutzenunabhängigkeit** erweitert die Prioritätsunabhängigkeit, um auch Lotterien abzudecken: Eine Attributmenge **X** ist nutzenunabhängig von einer Attributmenge **Y**, wenn die Prioritäten zwischen Lotterien für die Attribute in **X** unabhängig von den jeweiligen Werten der Attribute in **Y** sind. Eine Attributmenge ist **gegenseitig nutzenunabhängig** (GNU), wenn jede ihrer Untermengen nutzenunabhängig von den restlichen Attributen ist. Wieder scheint es vernünftig zu sein, für die Flughafenattribute GNU zu fordern.

GNU impliziert, dass das Verhalten des Agenten unter Verwendung einer **multiplikativen Nutzenfunktion** (Keeney, 1974) beschrieben werden kann. Die allgemeine Form einer multiplikativen Nutzenfunktion erkennt man am besten, indem man einen Fall mit drei Attributen betrachtet. Der Kürze halber schreiben wir U_i für $U_i(x)$:

$$U = k_1U_1 + k_2U_2 + k_3U_3 + k_1k_2U_1U_2 + k_2k_3U_2U_3 + k_3k_1U_3U_1 + k_1k_2k_3U_1U_2U_3.$$

Das sieht zwar nicht sehr einfach aus, aber es enthält nur drei einattributige Nutzenfunktionen und drei Konstanten. Im Allgemeinen kann ein n-attributiges Problem mit

GNU unter Verwendung von *n* einattributigen Nutzen und *n* Konstanten modelliert werden. Jede dieser einattributigen Nutzenfunktionen kann unabhängig von den anderen Attributen entwickelt werden und diese Kombination erzeugt garantiert die korrekte Allgemeinpriorität. Um eine rein additive Nutzenfunktion zu erhalten, braucht man zusätzliche Annahmen.

16.5 Entscheidungsnetze

In diesem Abschnitt betrachten wir einen allgemeinen Mechanismus für rationale Entscheidungen. Die Notation wird häufig als **Einflussdiagramm** bezeichnet (Howard und Matheson, 1984), aber wir verwenden den aussagekräftigeren Begriff **Entscheidungsnetz**. Entscheidungsnetze kombinieren Bayessche Netze mit zusätzlichen Knotentypen für Aktionen und Nutzen. Wir verwenden den Flughafenstandort als Beispiel.

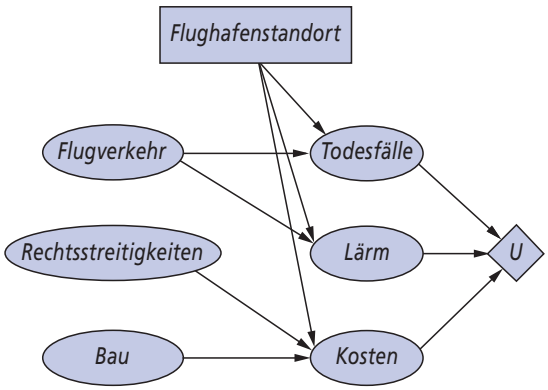


Abbildung 16.6: Ein einfaches Entscheidungsnetz für das Flughafenstandortproblem.

16.5.1 Darstellung eines Entscheidungsproblems mithilfe eines Entscheidungsnetzes

In seiner allgemeinsten Form stellt ein Entscheidungsnetz Informationen über den aktuellen Zustand des Agenten dar, seine möglichen Aktionen, den Zustand, der aus der Aktion des Agenten entsteht, sowie den Nutzen dieses Zustandes. Aus diesem Grund stellt es die Trägerschicht für die Implementierung nutzenbasierter Agenten des in *Abschnitt 2.4.5* vorgestellten Typs dar. ▶ Abbildung 16.6 zeigt ein Entscheidungsnetz für das Flughafenstandortproblem. Sie verdeutlicht die drei Arten verwendeter Knoten:

- **Zufallsknoten** (Ovale) stellen Zufallsvariablen dar, so wie in Bayesschen Netzen. Der Agent könnte unsicher über die Baukosten, das Flugaufkommen und die Entstehung von Rechtsstreitigkeiten sein, ebenso wie über die Variablen *Todesfälle*, *Lärm* und *Kosten*, die ebenfalls von dem gewählten Standort abhängig sind. Jedem Zufallsknoten ist eine bedingte Verteilung zugeordnet, deren Index durch den Zustand der Elternknoten gebildet wird. In Entscheidungsnetzen können die Elternknoten sowohl Entscheidungsknoten als auch Zufallsknoten sein. Beachten Sie, dass jeder der aktuellen Zufallsknoten Teil eines großen Bayesschen Netzes für die Abschätzung von Baukosten, Flugaufkommen oder das Potenzial für Rechtsstreitigkeiten sein könnte.

- **Entscheidungsknoten** (Rechtecke) kennzeichnen Punkte, an denen der Entscheider zwischen verschiedenen Aktionen auswählen kann. In diesem Fall kann *Flughafenstandort* für jeden in Frage kommenden Standort einen anderen Wert annehmen. Die Auswahl beeinflusst die Kosten, die Sicherheit und den resultierenden Lärm. In diesem Kapitel gehen wir davon aus, dass wir es nur mit einem einzigen Entscheidungsknoten zu tun haben. *Kapitel 17* beschäftigt sich mit Fällen, in denen mehrere Entscheidungen getroffen werden müssen.
- **Nutzenknoten** (Rauten) stellen die Nutzenfunktion des Agenten dar.[9] Der Nutzenknoten hat als Eltern alle Variablen, die das Ergebnis beschreiben, das den Nutzen direkt beeinflusst. Dem Nutzenknoten ist eine Beschreibung des Nutzens des Agenten als Funktion der Elternattribute zugeordnet. Die Beschreibung könnte einfach nur eine Tabellierung der Funktion sein oder auch eine parametrisierte additive oder multiplikative Funktion.

In vielen Fällen wird auch eine vereinfachte Form verwendet. Die Notation bleibt dieselbe, aber die Zufallsknoten, die den Ergebniszustand beschreiben, werden weggelassen. Stattdessen wird der Nutzenknoten direkt mit den aktuellen Zustandsknoten und dem Entscheidungsknoten verbunden. In diesem Fall stellt der Nutzenknoten nicht die Nutzenfunktion für Zustände dar, sondern den erwarteten Nutzen, der jeder Aktion zugeordnet ist, wie in Gleichung (16.1) definiert; d.h., der Knoten ist mit einer **Aktion-Nutzen-Funktion** verbunden (beim Reinforcement Learning auch als **Q-Funktion** bekannt, siehe *Kapitel 21*). ▶ Abbildung 16.7 zeigt die Aktion/Nutzen-Repräsentation des Flughafenproblems.

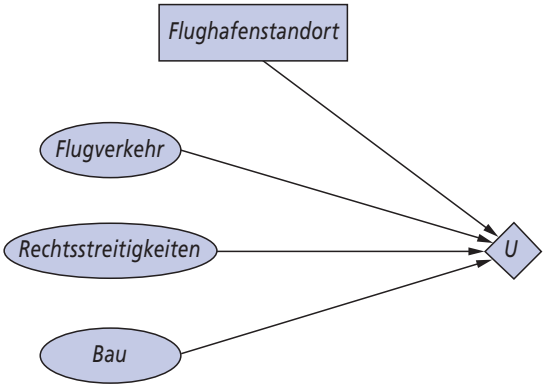


Abbildung 16.7: Eine vereinfachte Repräsentation des Flughafenstandortproblems. Zufallsknoten, die den Ergebniszuständen entsprechen, wurden ausfaktorisiert.

Beachten Sie, dass sich die Zufallsknoten *Lärm*, *Todesfälle* und *Kosten* in Abbildung 16.6 auf zukünftige Zustände beziehen, sodass ihre Werte nie als Evidenzvariablen verwendet werden können. Die vereinfachte Version, die diese Knoten weglässt, kann also überall dort zum Einsatz kommen, wo die allgemeinere Form verwendet werden kann. Obwohl die vereinfachte Form weniger Knoten enthält, bedeutet das Weglassen einer expliziten Beschreibung des Ergebnisses der Standortentscheidung, dass sie weniger fle-

9 Diese Knoten werden in der Literatur häufig als **Wertknoten** bezeichnet. Wir bevorzugen eine Unterscheidung zwischen Nutzen- und Wertfunktionen, wie bereits beschrieben, weil der Ergebniszustand eine Lotterie darstellen kann.

xibel im Hinblick auf sich ändernde Situationen ist. Beispielsweise kann in Abbildung 16.6 eine Änderung des Flugzeuglärmpegels durch eine Änderung in der bedingten Wahrscheinlichkeitstabelle reflektiert werden, die dem Knoten *Lärm* zugeordnet ist, während eine Gewichtungsänderung der Lärmemissionen in der Nutzenfunktion durch eine Änderung in der Nutzentabelle reflektiert werden kann. In dem in Abbildung 16.7 gezeigten Aktion/Nutzen-Diagramm dagegen müssen alle diese Änderungen durch Änderungen in der Aktion/Nutzen-Tabelle reflektiert werden. Im Wesentlichen ist die Aktion/Nutzen-Formulierung eine *kompilierte* Version der ursprünglichen Formulierung.

16.5.2 Auswertung von Entscheidungsnetzen

Aktionen werden ausgewählt, indem das Entscheidungsnetz für jede mögliche Einstellung des Entscheidungsknotens ausgewertet wird. Nachdem der Entscheidungsknoten gesetzt ist, verhält er sich genau wie ein Zufallsknoten, der als Evidenzvariable gesetzt wurde. Der Algorithmus für die Auswertung von Entscheidungsnetzen sieht wie folgt aus:

1. Die Evidenzvariablen für den aktuellen Zustand werden gesetzt.
2. Für jeden möglichen Wert des Entscheidungsknotens:
   a. Der Entscheidungsknoten wird auf diesen Wert gesetzt.
   b. Die bedingten Wahrscheinlichkeiten für die Elternknoten des Nutzenknotens werden unter Verwendung eines standardmäßigen probabilistischen Inferenzalgorithmus berechnet.
   c. Der resultierende Nutzen für die Aktion wird berechnet.
3. Die Aktion mit dem höchsten Nutzen wird zurückgegeben.

Das ist eine einfache Erweiterung des Algorithmus für Bayessche Netze, die direkt in das in Abbildung 13.1 gezeigte Agentendesign eingefügt werden kann. In *Kapitel 17* werden wir sehen, dass die Möglichkeit der Ausführung mehrerer aufeinanderfolgender Aktionen das Problem sehr viel interessanter macht.

16.6 Der Wert von Information

Tipp

In der obigen Analyse sind wir davon ausgegangen, dass dem Agenten alle relevanten Informationen – oder zumindest alle verfügbaren Informationen – bereitgestellt werden, bevor er eine Entscheidung trifft. In der Praxis ist das selten der Fall. *Einer der wichtigsten Bestandteile der Entscheidungsfindung ist zu wissen, welche Fragen gestellt werden müssen.* Ein Arzt kann beispielsweise nicht erwarten, die Ergebnisse *aller möglichen* Diagnosetests und Fragen zur Verfügung zu haben, wenn ein Patient zum ersten Mal sein Sprechzimmer betritt.[10] Tests sind häufig teuer und manchmal gefährlich (sowohl direkt als auch wegen der entstehenden Verzögerungen). Ihre Bedeutung ist von zwei Faktoren abhängig: ob die Testergebnisse zu einem wesentlich besseren Behandlungsplan führen und wie wahrscheinlich die verschiedenen Testergebnisse sind.

10 In Amerika ist die erste Frage, die immer vorab gestellt wird, ob der Patient krankenversichert ist.

Dieser Abschnitt beschreibt die **Informationswerttheorie**, die einem Agenten hilft auszuwählen, welche Informationen angefordert werden sollen. Wir nehmen an, dass der Agent vor Auswahl einer „realen" Aktion, die durch den Entscheidungsknoten dargestellt wird, den Wert jeder der potenziell beobachtbaren Zufallsvariablen im Modell anfordern kann. Somit beinhaltet die Informationswerttheorie eine vereinfachte Form der sequenziellen Entscheidungsfindung – vereinfacht deshalb, weil die Beobachtungsaktionen nur den **Belief State** des Agenten und nicht den externen physischen Zustand beeinflussen. Der Wert irgendeiner konkreten Beobachtung muss vom Potenzial abgeleitet werden, die endgültige physische Aktion des Agenten zu beeinflussen – und dieses Potenzial lässt sich direkt aus dem Entscheidungsmodell selbst abschätzen.

16.6.1 Ein einfaches Beispiel

Angenommen, eine Ölgesellschaft hofft, einen von n nicht voneinander unterscheidbaren Blöcken von Meeresbohrrechten kaufen zu können. Wir nehmen weiterhin an, dass genau einer der Blöcke Öl mit einem Wert von C Euro enthält und dass der Preis jedes Blocks C/n Euro beträgt. Wenn das Unternehmen risikoneutral ist, ist es unentschieden, einen Block zu kaufen oder nicht zu kaufen.

Nehmen wir jetzt an, ein Seismologe bietet dem Unternehmen die Ergebnisse einer Untersuchung von Block 3 an, die definitiv aussagen, ob der Block Öl enthält. Wie viel soll das Unternehmen für diese Information zahlen? Für die Beantwortung dieser Frage muss untersucht werden, was das Unternehmen macht, wenn es die Information besitzt:

- Mit der Wahrscheinlichkeit $1/n$ zeigt die Untersuchung Öl in Block 3 an. In diesem Fall kauft das Unternehmen Block 3 für C/n Euro und macht einen Gewinn von $C - C/n = (n-1)C/n$ Euro.
- Mit der Wahrscheinlichkeit $(n-1)/n$ zeigt die Untersuchung, dass der Block kein Öl enthält, sodass das Unternehmen einen anderen Block kauft. Jetzt ändert sich die Wahrscheinlichkeit, dass Öl in einem der anderen Blöcke gefunden wird, von $1/n$ in $1/(n-1)$, sodass das Unternehmen einen erwarteten Gewinn von $C/(n-1) - C/n = C/n(n-1)$ Euro macht.

Jetzt können wir anhand der Informationen aus der Untersuchung den erwarteten Gewinn berechnen:

$$\frac{1}{n} \times \frac{(n-1)C}{n} + \frac{n-1}{n} \times \frac{C}{n(n-1)} = C/n.$$

Damit sollte das Unternehmen dem Seismologen also bis zu C/n Euro für die Information zahlen: Die Information ist genauso viel wert wie der eigentliche Block.

Der Wert der Information leitet sich von der Tatsache ab, dass *mit* der Information der eigene Aktionskurs so geändert werden kann, dass er für die *tatsächliche* Situation geeignet ist. Man kann der Situation entsprechend unterscheiden, während man ohne die Information das tun muss, was für den Durchschnitt der möglichen Situationen am besten ist. Im Allgemeinen ist der Wert einer bestimmten Information als die Differenz des erwarteten Wertes zwischen den besten Aktionen vor und nach Erhalt der Information definiert.

16.6.2 Eine allgemeine Formel für perfekte Information

Man kann ganz einfach eine allgemeine Formel für den Wert von Information ableiten. Wir nehmen an, dass man genaue Evidenz über den Wert einer Zufallsvariablen E_j erhält (d.h., wir lernen $E_j = e_j$), sodass wir den Ausdruck **Wert perfekter Information** (WPI) verwenden.[11]

Sei das aktuelle Wissen des Agenten gleich **e**. Dann ist der Wert der aktuell besten Aktion α definiert durch

$$EU(\alpha \mid \mathbf{e}) = \max_a \sum_{s'} P(\text{Result}(a) = s' \mid a, \mathbf{e}) U(s')$$

und der Wert der neuen besten Aktion (nachdem wir die neue Evidenz $E_j = e_j$ erhalten haben) ist

$$EU(\alpha_{e_j} \mid \mathbf{e}, e_j) = \max_a \sum_{s'} P\left(\text{Result}(a) = s' \mid a, \mathbf{e}, e_j\right) U(s').$$

Nun ist aber E_j eine Zufallsvariable, deren Wert *aktuell* unbekannt ist. Um also den Wert für die Ermittlung von E_j anhand der aktuellen Information **e** bestimmen zu können, müssen wir einen Mittelwert über alle möglichen Werte e_{jk} bilden, die wir für E_j feststellen können, wozu wir unsere *aktuellen* Glauben über ihren Wert verwenden:

$$WPI_e(E_j) = \left(\sum_k P(E_j = e_{jk} \mid \mathbf{e}) EU(\alpha_{e_{jk}} \mid \mathbf{e}, E_j = e_{jk})\right) - EU(\alpha \mid \mathbf{e}).$$

Um ein Gefühl für diese Formel zu erhalten, betrachten Sie den einfachen Fall, wo es nur zwei Aktionen, a_1 und a_2, gibt, aus denen wir auswählen. Ihre aktuellen erwarteten Nutzen sind U_1 und U_2. Die Information $E_j = e_{jk}$ ergibt neue erwartete Nutzen U_1' und U_2' für die Aktionen. Doch bevor wir E_j erhalten, haben wir Wahrscheinlichkeitsverteilungen über die möglichen Werte von U_1' und U_2' (von denen wir annehmen, dass sie voneinander unabhängig sind).

Angenommen, a_1 und a_2 stellen zwei unterschiedliche Routen durch ein Gebirge im Winter dar. a_1 ist eine schöne, gerade Straße durch einen niederen Pass und a_2 eine enge, gewundene Schotterstraße über die Bergspitze. Mit diesen Informationen ist a_1 natürlich zu bevorzugen, weil es sehr wahrscheinlich ist, dass die zweite Route durch Lawinen blockiert wurde, während es relativ unwahrscheinlich ist, dass die erste Route durch zu viel Verkehr blockiert ist. U_1 ist also deutlich höher als U_2. Es ist möglich, Satellitenbilder E_j über den tatsächlichen Zustand jeder Straße zu erhalten, der neue Erwartungen U_1' und U_2' für die beiden Übergänge erbringt. Die Verteilungen für diese Erwartungen sehen Sie in ▶ Abbildung 16.8(a). Offensichtlich ist es in diesem Fall nicht wert, die Kosten für Satellitenbilder zu zahlen, weil es unwahrscheinlich ist, dass die daraus abgeleitete Information den Plan ändert. Ohne Änderung hat die Information keinen Wert.

11 Es geht keine Ausdrucksstärke verloren, wenn perfekte Information gefordert wird. Angenommen, wir möchten den Fall modellieren, in dem wir etwas mehr Sicherheit über eine Variable erhalten. Dazu können wir eine *andere* Variable einführen, über die wir perfekte Information lernen. Nehmen wir zum Beispiel an, dass anfangs eine breite Unsicherheit in Bezug auf die Variable *Temperatur* besteht. Dann gewinnen wir das perfekte Wissen *Thermometer* = 37; dies liefert uns die unvollständige Information über die wahre *Temperatur* und die Unsicherheit aufgrund eines Messfehlers wird im Sensormodell **P**(*Thermometer*|*Temperatur*) kodiert. Übung 16.19 bringt hierzu ein weiteres Beispiel.

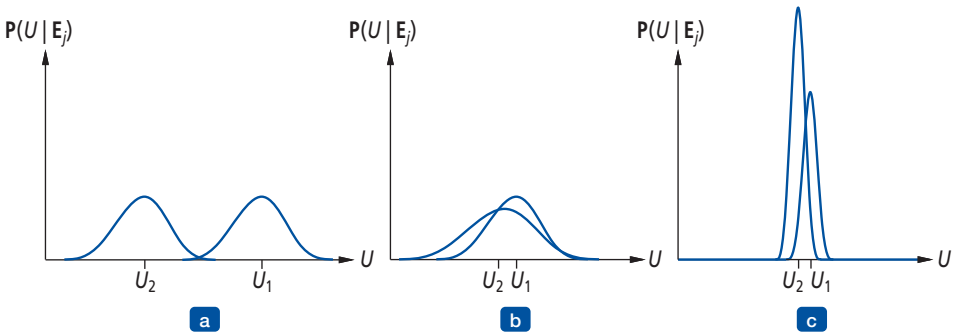


Abbildung 16.8: Drei generische Fälle für den Wert von Information. In (a) ist a_1 fast sicher a_2 überlegen, deshalb braucht man die Information nicht. In (b) ist die Auswahl unklar und die Information sehr wichtig. In (c) ist die Auswahl unklar, aber da das keine große Rolle spielt, ist die Information weniger wertvoll. (Hinweis: Da U_2 in (c) eine hohe Spitze aufweist, ist ihr erwarteter Wert mit höherer Sicherheit als der für U_1 bekannt.).

Angenommen, wir wählen zwischen zwei gewundenen Schotterstraßen leicht unterschiedlicher Länge und wir führen einen schwer verletzten Fahrgast mit uns. Selbst wenn U_1 und U_2 relativ eng beieinander liegen, sind die Verteilungen von U_1' und U_2' sehr breit. Es besteht die hohe Wahrscheinlichkeit, dass die zweite Route frei ist, während die erste blockiert ist, und in diesem Fall ist der Unterschied im Nutzen sehr hoch. Die WPI-Formel zeigt an, dass es sinnvoll sein könnte, die Satellitenfotos zu beschaffen. ▶ Abbildung 16.8(b) zeigt diese Situation.

Schließlich nehmen wir an, dass wir zwischen zwei Schotterstraßen im Sommer wählen müssen, wo eine Blockade durch Lawinen unwahrscheinlich ist. In diesem Fall könnten die Satellitenfotos zeigen, dass die eine Straße malerischer als die andere ist, weil dort die Gebirgswiesen blühen, oder vielleicht nasser wegen fehlgeleiteter Bäche. Es ist also sehr wahrscheinlich, dass wir unseren Plan ändern würden, wenn wir diese Information hätten. In diesem Fall ist jedoch die Wertdifferenz zwischen zwei Routen relativ klein; deshalb brauchen wir die Fotos nicht. Diese Situation ist in ▶ Abbildung 16.8(c) dargestellt.

Insgesamt kann man sagen: *Information hat einen Wert zum einen in dem Ausmaß, in dem sie wahrscheinlich eine Änderung des Planes verursacht, und zum anderen in dem Ausmaß, in dem der neue Plan deutlich besser als der alte ist.*

Tipp

16.6.3 Eigenschaften des Informationswertes

Man fragt sich vielleicht, ob es möglich ist, dass Information schädlich ist: Kann sie auch einen negativen erwarteten Wert haben? Vom Gefühl her sollte man meinen, das sei unmöglich. Schließlich könnte man im schlechtesten Fall die Information einfach ignorieren und behaupten, sie nie erhalten zu haben. Dies wird durch das folgende Theorem bestätigt, das auf beliebige entscheidungstheoretische Agenten angewendet werden kann:

Der erwartete Wert von Information ist nicht negativ:

Tipp

$$\forall \mathbf{e}, E_j \quad WPIe(E_j) \geq 0.$$

Das Theorem folgt direkt aus der Definition von WPI und wir überlassen den Beweis dem Leser (Übung 16.20). Es handelt sich natürlich um ein Theorem über den *erwarte-*

ten Wert und nicht den *tatsächlichen* Wert. Zusätzliche Informationen können leicht zu einem Plan führen, der sich als schlechter als der ursprüngliche Plan herausstellt, wenn die Informationen irreführend sind. Zum Beispiel könnte ein medizinischer Test, der ein falsches positives Ergebnis liefert, zu einem unnötigen chirurgischen Eingriff führen; doch heißt das nicht, dass der Test nicht ausgeführt werden sollte.

Beachten Sie, dass WPI von dem aktuellen Zustand der Information abhängig ist; deshalb ist sie als Subskript angegeben. Er kann sich ändern, sobald weitere Informationen zur Verfügung stehen. Für jedes gegebene Stück von Evidenz E_j kann der Wert für ihre Anforderung sinken (wenn z.B. eine andere Variable die A-posteriori-Verteilung von E_j stark einschränkt) oder steigen (wenn z.B. eine andere Variable einen Anhaltspunkt liefert, auf dem E_j aufbaut, und es damit ermöglicht, einen neuen besseren Plan abzuleiten). Damit ist WPI nicht additiv. Das bedeutet:

$$WPIe(E_j, E_k) \neq WPIe(E_j) + WPIe(E_k). \qquad \text{(allgemein)}$$

Allerdings ist WPI unabhängig von der Reihenfolge, das heißt:

$$WPI_{\mathbf{e}}(E_j, E_k) = WPI_{\mathbf{e}}(E_j) + WPI_{\mathbf{e},e_j}(E_k) = WPI_{\mathbf{e}}(E_k) + WPI_{\mathbf{e},e_k}(E_j).$$

Die Unabhängigkeit von der Reihenfolge unterscheidet Sensoraktionen von normalen Aktionen und vereinfacht das Problem, den Wert einer Folge von Sensoraktionen zu berechnen.

16.6.4 Implementierung eines Agenten, der Informationen sammelt

Ein sensibler Agent sollte Fragen des Benutzers in sinnvoller Reihenfolge stellen, irrelevante Fragen vermeiden, die Bedeutung jedes Informationsabschnittes in Relation zu seinen Kosten berücksichtigen und mit dem Fragen aufhören, wenn es an der Zeit ist. All diese Fähigkeiten können erzielt werden, wenn man den Informationswert als Anhaltspunkt verwendet.

▶ Abbildung 16.9 zeigt den allgemeinen Entwurf eines Agenten, der auf intelligente Weise Informationen sammelt, bevor er handelt. Hier wollen wir davon ausgehen, dass jeder beobachtbaren Evidenzvariablen E_j Kosten *Kosten*(E_j) zugeordnet sind, welche die Kosten reflektieren, die aufzuwenden sind, um die Evidenz durch Tests, Berater, Fragen usw. zu erhalten. Der Agent fordert den scheinbar wertvollsten Informationsabschnitt an – verglichen mit den Kosten. Wir gehen davon aus, dass das Ergebnis der Aktion *Anfordern*(E_j) ist, dass die nächste Wahrnehmung den Wert E_j bereitstellt. Wenn keine Beobachtung ihre Kosten wert ist, wählt der Agent eine „reale“ Aktion aus.

```
function INFORMATION-GATHERING-AGENT(percept) returns eine Aktion
   persistent: D, ein Entscheidungsnetz

   integriere percept in D
   j ← der Wert, der WPI(Ej) - Kosten(Ej) maximiert
   if WPI(Ej) > Kosten(Ej)
      then return REQUEST(Ej)
   else return die beste Aktion aus D
```

Abbildung 16.9: Entwurf eines einfachen Agenten, der Informationen sammelt. Der Agent wählt wiederholt die Beobachtung mit dem höchsten Informationswert aus, bis die Kosten der nächsten Beobachtung höher als ihr erwarteter Gewinn sind.

Der beschriebene Agentenalgorithmus implementiert eine Form der Informationsermittlung, die als **kurzsichtig** bezeichnet wird. Das liegt daran, dass die WPI-Formel kurzsichtig ist und den Wert von Information so berechnet, als würde nur eine einzige Evidenzvariable angefordert. Eine kurzsichtige Steuerung basiert auf demselben heuristischen Konzept wie eine „gierige" Suche und funktioniert in der Praxis meist sehr gut. (Beispielsweise wurde gezeigt, dass sie erfahrene Ärzte bei der Auswahl von Diagnosetests übertrifft.)

Wenn es jedoch keine einzige Evidenzvariable gibt, die uns weiterhilft, könnte ein kurzsichtiger Agent vorschnell eine Aktion auswählen, obwohl es besser gewesen wäre, zuerst zwei oder mehr Variablen anzufordern und dann zu handeln. In dieser Situation ist es besser, einen *bedingten Plan* (wie in *Abschnitt 11.3.2* beschrieben) zu konstruieren, der Variablenwerte abfragt und je nach Antwort unterschiedliche nächste Schritte unternimmt.

Schließlich ist noch die Wirkung zu berücksichtigen, die eine Folge von Fragen für einen menschlichen Antwortenden haben wird. Menschen antworten gegebenenfalls besser auf eine Folge von Fragen, wenn sie „Sinn machen", sodass bestimmte Expertensysteme erstellt werden, die diesen Punkt berücksichtigen und Fragen in einer Reihenfolge stellen, die den Gesamtnutzen des Systems und des Menschen maximiert, anstatt in einer Reihenfolge, die den Wert der Information maximiert.

16.7 Entscheidungstheoretische Expertensysteme

Der Bereich der **Entscheidungsanalyse**, der sich in den 1950er und 1960er Jahren entwickelt hat, beschäftigt sich mit der Anwendung der Entscheidungstheorie auf echte Entscheidungsprobleme. Sie wird verwendet, um rationale Entscheidungen in wichtigen Domänen zu unterstützen, wo die Einsätze hoch sind, wie beispielsweise Wirtschaft, Regierung, Justiz, Militärstrategie, medizinische Diagnose und Gesundheitswesen, Konstruktion und Ressourcenverwaltung. Dieser Prozess bedingt eine sorgfältige Untersuchung der möglichen Aktionen und Ergebnisse, ebenso wie der Prioritäten, die den verschiedenen Ergebnissen zugeordnet sind. Es ist Tradition in der Entscheidungsanalyse, von zwei Rollen zu sprechen: Der **Entscheider** spezifiziert die Prioritäten zwischen den Ergebnissen und der **Entscheidungsanalytiker** listet die möglichen Aktionen und Ergebnisse auf und fördert die Prioritäten des Entscheiders ans Licht, um den besten Aktionspfad zu finden. Bis Anfang der 1980er Jahre war die wichtigste Aufgabe der Entscheidungsanalyse, den Menschen zu helfen, Entscheidungen zu treffen, die ihre eigenen Prioritäten reflektierten. Heute sind immer mehr Entscheidungsprozesse automatisiert und die Entscheidungsanalyse wird verwendet, um sicherzustellen, dass sich die automatisierten Prozesse wie gewünscht verhalten.

Die frühe Forschung auf dem Gebiet der Expertensysteme konzentrierte sich darauf, Fragen zu beantworten, statt Entscheidungen zu treffen. Diese Systeme, die Aktionen empfahlen, statt Meinungen zu Tatsachen bereitzustellen, verwendeten hierzu meistens Bedingung/Aktion-Regeln und keine expliziten Repräsentationen von Ergebnissen und Prioritäten. Die Entwicklung der Bayesschen Netze Ende der 1980er Jahre ermöglichte es, große Systeme zu erstellen, die korrekte probabilistische Inferenzen aus Evidenzen erzeugten. Die Ergänzung um Entscheidungsnetze bedeutet, dass Expertensysteme entwickelt werden können, die optimale Entscheidungen empfehlen und dabei die Prioritäten des Benutzers sowie die verfügbaren Evidenzen berücksichtigen.

Ein System, das Nutzen berücksichtigt, kann eine der häufigsten Fallen vermeiden, die dem Beratungsprozess zuzuordnen sind: die Verwechslung von Wahrscheinlichkeit und Wichtigkeit. Eine gebräuchliche Vorgehensweise in frühen Expertensystemen in der Medizin war beispielsweise, mögliche Diagnosen in der Reihenfolge ihrer Wahrscheinlichkeit zu ermitteln und die wahrscheinlichste zu empfehlen. Leider kann das eine verheerende Wirkung haben! Für die meisten Patienten in der allgemeinen Praxis sind die beiden *wahrscheinlichsten* Diagnosen „Ihnen fehlt nichts" und „Sie sind schwer erkältet", aber wenn die drittwahrscheinlichste Diagnose für einen Patienten Lungenkrebs ist, dann ist das ein ernsthaftes Problem. Offensichtlich sollte ein Test- oder Behandlungsplan sowohl von Wahrscheinlichkeiten als auch von Nutzen abhängig gemacht werden. Moderne medizinische Expertensysteme können den Wert von Informationen berücksichtigen, um Tests zu empfehlen, und dann eine Differentialdiagnose beschreiben.

Jetzt werden wir den Wissensengineering-Prozess für entscheidungstheoretische Expertensysteme beschreiben. Als Beispiel betrachten wir das Problem der Auswahl der richtigen medizinischen Behandlung für eine bestimmte Art von angeborenem Herzfehler bei Kindern (siehe Lucas, 1996).

Etwa 0,8% aller Kinder werden mit Herzanomalien geboren, die meisten davon mit einer **Aortaverengung**. Dies kann durch eine Operation, eine Angioplastik (Erweiterung der Aorta mit Hilfe eines Ballons, der in der Arterie platziert wird) oder Medikamente geheilt werden. Das Problem dabei ist die Entscheidung, welche Behandlungsmethode gewählt und wann sie ausgeführt wird. Je jünger das Kind ist, desto größer sind die Risiken bestimmter Behandlungen, aber man darf auch nicht zu lange warten. Ein entscheidungstheoretisches Expertensystem für dieses Problem kann durch ein Team gebildet werden, das aus mindestens einem Bereichsexperten (einem Kinderkardiologen) und einem Wissensingenieur besteht. Der Prozess kann in die folgenden Schritte zerlegt werden:

Ein kausales Modell anlegen: Ermitteln, welche möglichen Symptome, Störungen, Behandlungen und Ergebnisse es gibt. Anschließend werden Pfeile zwischen ihnen gezeichnet, die anzeigen, welche Störungen welche Symptome verursachen und welche Behandlungen welche Störungen heilen. Einiges davon ist dem Domänenexperten bekannt, anderes stammt aus der Literatur. Häufig stimmt das Modell gut mit den informellen grafischen Darstellungen aus Medizinlehrbüchern überein.

Vereinfachung zu einem qualitativen Entscheidungsmodell: Weil wir das Modell verwenden, um Behandlungsentscheidungen zu treffen, und nicht für irgendwelche anderen Zwecke (wie etwa die Ermittlung der gemeinsamen Wahrscheinlichkeit bestimmter Symptom/Störung-Kombinationen), können wir das Modell häufig vereinfachen, indem wir Variablen entfernen, die nichts mit den Behandlungsentscheidungen zu tun haben. Manchmal müssen Variablen zerlegt oder vereinigt werden, um mit den Ideen des Experten übereinzustimmen. Beispielsweise hatte das ursprüngliche Modell für die Aortaverengung die Variable *Behandlung* mit den Werten *operation*, *angioplastik* und *medikamente*. Darüber hinaus gab es eine separate Variable für das *Timing* der Behandlung. Der Experte konnte sich die Variablen jedoch nur schwer voneinander getrennt vorstellen; deshalb wurden sie kombiniert, sodass *Behandlung* Werte wie etwa *operation in 1 monat* annehmen kann. Damit erhalten wir das in ▶ Abbildung 16.10 gezeigte Modell.

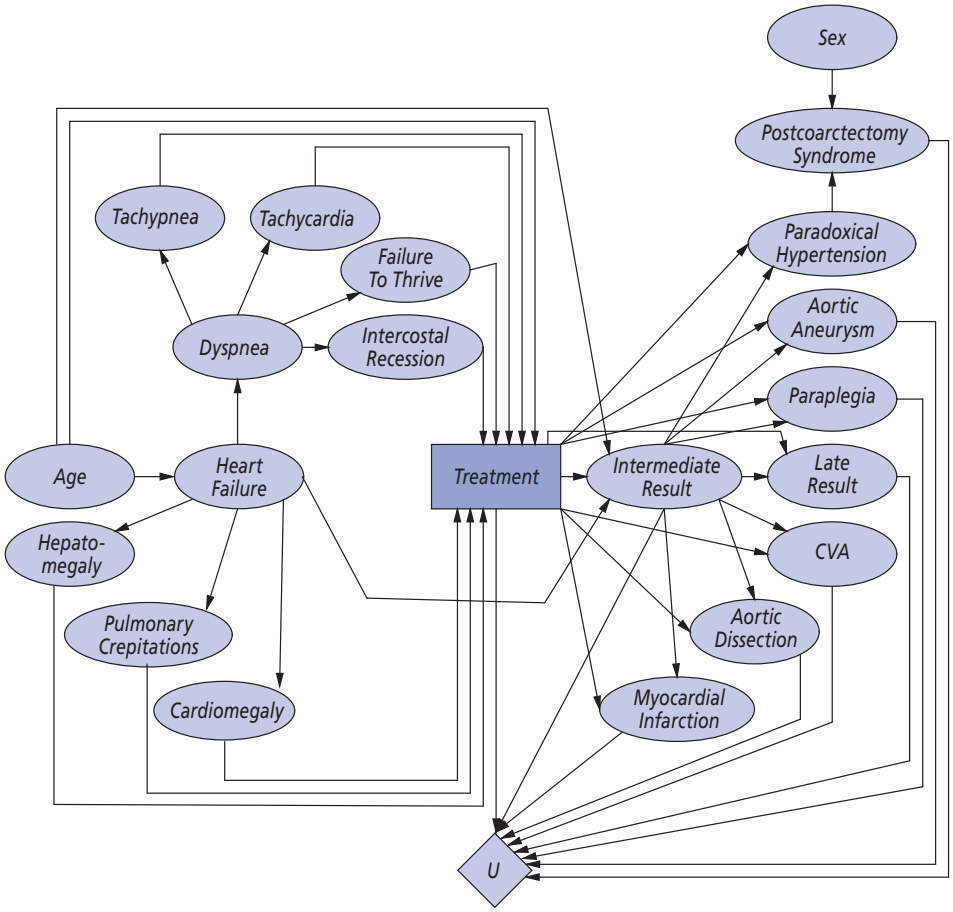


Abbildung 16.10: Einflussdiagramm für Aortaverengung (mit freundlicher Genehmigung von Peter Lucas).

Wahrscheinlichkeiten zuordnen: Wahrscheinlichkeiten stammen aus Patientendatenbanken, Literaturstudien oder den subjektiven Beurteilungen des Experten. Beachten Sie, dass ein Diagnosesystem anhand von Symptomen und anderen Beobachtungen auf die Störung oder Ursache des Problems schließt. Folglich hat man in den frühen Jahren beim Erstellen derartiger Systeme die Experten nach der Wahrscheinlichkeit einer Ursache bei einer bestimmten Wirkung gefragt. Im Allgemeinen erachteten sie diese Aufgabe als schwierig und konnten die Wahrscheinlichkeit einer Wirkung bei einer gegebenen Ursache besser beurteilen. Moderne Systeme bewerten deshalb kausales Wissen und kodieren es direkt in der Struktur des Bayesschen Netzmodells, wobei das diagnostische Schließen den Inferenzalgorithmen des Bayesschen Netzes überlassen wird (Shachter und Heckerman, 1987).

Nutzen zuordnen: Wenn es eine kleine Anzahl möglicher Ergebnisse gibt, lassen sie sich mit den Methoden von *Abschnitt 16.3.1* einzeln auflisten und bewerten. Wir würden eine Skala vom besten zum schlechtesten Ergebnis erstellen und jedem davon einen numerischen Wert zuweisen, z.B. 0 für Tod und 1 für vollständige Heilung. Anschließend würden wir die anderen Ergebnisse in dieser Skala anordnen. Das kann durch den Experten erfolgen, aber es ist besser, wenn die Patienten (oder im Fall der Kinder die

Eltern) beteiligt werden können, weil unterschiedliche Menschen unterschiedliche Prioritäten haben. Wenn es exponentiell viele Ergebnisse gibt, brauchen wir eine Möglichkeit, sie in Nutzenfunktionen mit Mehrfachattributen zu kombinieren. Beispielsweise könnten wir sagen, dass die Kosten verschiedener Komplikationen additiv sind.

Das Modell überprüfen und verfeinern: Um das System zu bewerten, brauchen wir eine Menge korrekter Eingabe/Ausgabe-Paare, einen sogenannten **Goldstandard**, mit dem wir vergleichen. Für medizinische Expertensysteme bedeutet dies häufig, dass die besten verfügbaren Ärzte versammelt, ihnen einige Fälle vorgestellt und sie nach ihrer Diagnose und dem empfohlenen Behandlungsplan gefragt werden. Anschließend sehen wir, wie gut das System mit ihren Empfehlungen übereinstimmt. Funktioniert es nicht gut, versuchen wir, die Teile zu isolieren, die schief gegangen sind, und korrigieren sie. Es kann sinnvoll sein, das System „rückwärts" auszuführen. Anstatt dem System Symptome zu zeigen und nach Diagnosen zu fragen, können wir ihm eine Diagnose präsentieren, wie etwa „Herzfehler", die vorhergesagte Wahrscheinlichkeit von Symptomen untersuchen, wie etwa Herzrasen, und sie mit der Medizinliteratur vergleichen.

Sensibilitätsanalyse durchführen: Dieser wichtige Schritt überprüft, ob die beste Entscheidung sensibel gegenüber kleinen Änderungen in den zugewiesenen Wahrscheinlichkeiten und Nutzen ist, indem diese Parameter systematisch variiert werden, wonach die Auswertung wiederholt wird. Wenn kleine Änderungen zu wesentlichen Entscheidungsdifferenzen führen, könnte es sinnvoll sein, weitere Ressourcen zu finden, um bessere Daten zu erhalten. Wenn alle Variationen zu derselben Entscheidung führen, hat der Agent mehr Vertrauen, dass es sich um die richtige Entscheidung handelt. Die Sensibilitätsanalyse ist sehr wichtig, weil einer der größten Kritikpunkte an den probabilistischen Ansätzen für Expertensysteme ist, dass es zu schwierig ist, die erforderlichen numerischen Wahrscheinlichkeiten einzuschätzen. Die Sensibilitätsanalyse ergibt häufig, dass viele der Zahlen nur sehr annähernd spezifiziert werden müssen. Beispielsweise könnten wir unsicher hinsichtlich der bedingten Wahrscheinlichkeit $P(herzrasen|dyspnoe)$ sein, doch wenn die optimale Entscheidung vernünftig robust gegenüber kleinen Variationen in der Wahrscheinlichkeit ist, müssen wir uns weniger Sorgen um unser Unwissen machen.

Bibliografische und historische Hinweise

Das Buch *La logique, ou l'art de penser*, das im Deutschen auch als *Logik von Port-Royal* (Arnauld, 1662) bekannt ist, stellt fest:

> *Um zu entscheiden, was zu tun ist, um Gutes zu erhalten oder Böses zu vermeiden, muss man nicht nur das eigentlich Gute und das eigentlich Böse betrachten, sondern auch die Wahrscheinlichkeit, dass es passiert oder nicht passiert; außerdem muss man die geometrische Proportion berücksichtigen, die alle diese Dinge gemeinsam haben.*

Moderne Texte sprechen nicht mehr von Gut oder Böse, sondern von *Nutzen*, doch stellt diese Aussage richtig fest, man Nutzen mit Wahrscheinlichkeit multiplizieren sollte („geometrische Proportion berücksichtigen"), um den erwarteten Nutzen zu liefern, und diesen über allen Ergebnissen („alle diese Dinge") zu maximieren, um „zu entscheiden, was zu tun ist". Es ist bemerkenswert, wie treffend dies bereits vor 350 Jahren formuliert wurde und nur acht Jahre, nachdem Pascal und Fermat gezeigt haben, wie Wahrscheinlichkeit richtig zu verwenden ist. Die *Logik von Port-Royal* hat auch die erste Veröffentlichung der Wette von Pascal markiert.

Daniel Bernoulli (1738), der das St.-Petersburg-Paradoxon untersuchte (siehe Übung 16.3), war der Erste, der die Bedeutung von Prioritätsmaßen für Lotterien erkannte. Er schrieb: „Der *Wert* eines Elements darf nicht auf seinem *Preis* basieren, sondern auf dem *Nutzen*, den es erbringt“ (die Hervorhebungen stammen von Bernoulli). Der utilitarische Philosoph Jeremy Bentham (1823) schlug den **hedonistischen Kalkül** für Gewichtungen von „Freud“ und „Leid“ vor, mit der Begründung, dass alle Entscheidungen (nicht nur monetärer Art) auf Nutzenvergleiche reduziert werden könnten.

Die Ableitung numerischer Nutzen aus Prioritäten wurde zuerst von Ramsey (1931) durchgeführt; die Axiome für Priorität im vorliegenden Text sind eher an die Neuformulierungen in *Theory of Games and Economic Behavior* (von Neumann und Morgenstern, 1944) angelehnt. Eine gute Repräsentation dieser Axiome im Kontext einer Diskussion über Risikoprioritäten finden Sie bei Howard (1977). Ramsey hatte subjektive Wahrscheinlichkeiten (nicht nur Nutzen) von den Prioritäten eines Agenten abgeleitet; Savage (1954) und Jeffrey (1983) führen neuere Konstruktionen dieser Art aus. Von Winterfeldt und Edwards (1986) zeigen eine modernere Perspektive zur Entscheidungsanalyse und ihrer Beziehung zu menschlichen Prioritätsstrukturen. Das Nutzenmaß Mikromort wird bei Howard (1989) beschrieben. Eine Umfrage von 1994 im *Economist* setzte den Wert eines Lebens zwischen $ 750.000 und $ 2,6 Millionen an. Richard Thaler (1992) erkannte jedoch irrationale Framing-Effekte in dem Preis, den man bereit ist zu zahlen, um das Todesrisiko zu vermeiden, und dem Preis, den man annehmen würde, um ein Risiko zu akzeptieren. Für eine Chance von 1/1000 würde eine Testperson nicht mehr als $ 200 zahlen, um das Risiko abzuwenden, aber sie würde das Risiko nicht für $ 50.000 annehmen. Wie viel ist jemand bereit, für ein QALY zu zahlen? Wenn es um einen konkreten Fall geht, sich selbst oder ein Familienmitglied zu retten, lautet die Zahl ungefähr „alles, was ich habe“. Doch wir können auch auf gesellschaftlicher Ebene fragen: Angenommen, es gibt einen Impfstoff, der X QALYs liefert, aber Y Dollar kostet; ist er dies wert? In diesem Fall geben die Befragten einen breiten Wertebereich von etwa $ 10.000 bis zu $ 150.000 pro QALY an (Prades et al. 2008). QALYs sind in der Medizin und Sozialpolitik gebräuchlicher für die Entscheidungsfindung als Mikromorts. Ein typisches Beispiel für ein Argument für eine große Änderung in der öffentlichen Gesundheitspolitik auf Grundlage erhöhter erwarteter Nutzen (gemessen in QALYs) finden Sie bei Russell (1990).

Der **Fluch des Optimierers** wurde von Smith und Winkler (2006) nachdrücklich in das Bewusstsein von Analytikern gerückt. Sie betonten, dass die finanziellen Vorteile für den Kunden, wie sie Analysten für ihre vorgeschlagene Handlungsweise projektiert haben, fast nie realisiert wurden. Sie führen dies direkt auf die Verzerrung zurück, die durch Auswählen einer optimalen Aktion entsteht, und zeigen, dass eine vollständigere Bayessche Analyse das Problem vermeiden kann. Das gleiche zugrunde liegende Konzept wird von Harrison und March (1984) als **Enttäuschung nach der Entscheidung** bezeichnet und im Kontext von Analysen zu Projekten der Kapitalinvestition von Brown (1974) angeführt. Der Fluch des Optimierers ist eng mit dem **Fluch des Gewinners** verwandt (Capen et al., 1971; Thaler, 1992), der sich auf das Bieten in Versteigerungen bezieht: Der Meistbietende (der Gewinner) hat regelmäßig den Wert des fraglichen Objekts überschätzt. Capen et al. zitieren einen Erdölingenieur zum Bieten für Ölbohrrechte: „Wenn jemand ein Gebiet gegen zwei oder drei andere Bietende gewinnt, kann er sich über sein Glück freuen, doch wie sollte er sich fühlen, wenn er gegen 50 andere gewonnen hat? Schlecht.“ Letztlich steckt hinter beiden Flüchen das allgemeine Phänomen der **Regression zur Mitte**, wobei Versuchspersonen, die auf Basis bisher gezeigter außergewöhnlicher Charakteristika mit hoher Wahrscheinlichkeit ausgewählt wurden, in der Zukunft weniger extreme Merkmale zeigen.

Das Allais-Paradoxon, das auf den Nobelpreisträger und Wirtschaftswissenschaftler Maurice Allais (1953) zurückgeht, wurde experimentell untersucht (Tversky und Kahneman, 1982; Conlisk, 1989), um zu zeigen, dass Menschen bei ihren Beurteilungen durchweg widersprüchlich sind. Das Ellsberg-Paradoxon zur Unsicherheitsaversion wurde in der Dissertation von Daniel Ellsberg (Ellsberg, 1962) eingeführt, der später zum Militäranalysten bei der RAND Corporation wurde und als Informant die sogenannten Pentagon-Papiere veröffentlichte, was zum Ende des Vietnamkrieges und zum Rücktritt von Präsident Nixon beitrug. Fox und Tversky (1995) beschreiben eine weitere Studie der Unsicherheitsaversion. Mark Machina (2005) gibt einen Überblick über die Entscheidung unter Unsicherheit und wie sie gegenüber der Theorie des erwarteten Nutzens abweichen kann.

In jüngster Zeit ist eine Flut von mehr oder weniger bekannten Büchern zur menschlichen Irrationalität zu verzeichnen. Das bekannteste ist *Predictably Irrational* (Ariely, 2009); weiterhin sind *Sway* (Brafman und Brafman, 2009), *Nudge* (Thaler und Sunstein, 2009), *Kluge* (Marcus, 2009), *How We Decide* (Lehrer, 2009) und *On Being Certain* (Burton, 2009) erwähnenswert. Sie ergänzen den Klassiker (Kahneman et al., 1982) und den Artikel, mit dem alles begann (Kahneman und Tversky, 1979). Andererseits läuft das Gebiet der evolutionären Psychologie (Buss, 2005) gegen diese Literatur an und argumentiert, dass Menschen in evolutionär geeigneten Kontexten durchaus rational sind. Die Anhänger der evolutionären Psychologie betonen, dass Irrationalität laut Definition in einem evolutionären Kontext bestraft wird, und zeigen, dass sie in manchen Fällen ein Artefakt der Versuchseinrichtung ist (Cummins und Allen, 1998). Nach mehreren Jahrzehnten des Pessimismus sind die letzten Jahre durch ein Wiederaufleben des Interesses an Bayesschen Modellen der Kognition geprägt (Oaksford und Chater, 1998; Elio, 2002; Chater und Oaksford, 2008).

Keeney und Raiffa (1976) bieten eine sorgfältige Einführung in die Nutzentheorie mit Mehrfachattributen. Sie beschreiben frühe Computerimplementierungen von Methoden zur Ermittlung der erforderlichen Parameter für eine Nutzenfunktion mit Mehrfachattributen und stellen zahlreiche Beispiele für reale Anwendungen der Theorie vor. In der KI ist die wichtigste Referenz für MAUT die Arbeit von Wellman (1985), die ein System namens URP (Utility Reasoning Package) enthält, das eine Menge von Aussagen über die Prioritätsunabhängigkeit und bedingte Unabhängigkeit verwenden kann, um die Struktur von Entscheidungsproblemen zu analysieren. Die Verwendung stochastischer Dominanz zusammen mit qualitativen Probabilitätsmodellen wurde ausführlich von Wellman (1988, 1990a) erforscht. Wellman und Doyle (1992) skizzieren, wie eine komplexe Menge nutzenunabhängiger Beziehungen verwendet werden kann, um ein strukturiertes Modell einer Nutzenfunktion bereitzustellen, ähnlich wie Bayessche Netze ein strukturiertes Modell gemeinsamer Wahrscheinlichkeitsverteilungen bieten. Bacchus und Grove (1995, 1996) sowie La Mura und Shoham (1999) zeigen weitere Ergebnisse dieser Art.

Die Entscheidungstheorie war seit den 1950er Jahren ein Standardwerkzeug in Wirtschaftswissenschaft, Finanzwesen und Management. Seit den 1980er Jahren waren Entscheidungsbäume das wichtigste Werkzeug für die Darstellung einfacher Entscheidungsprobleme. Smith (1988) bietet einen Überblick über die Vorgehensweise bei der Entscheidungsanalyse. Einflussdiagramme wurden von Howard und Matheson (1984) vorgestellt, basierend auf früheren Arbeiten von einer Gruppe (der auch Howard und Matheson zugehörten) am SRI (Miller et al., 1976). Die Methode von Howard und Matheson beinhaltete die Ableitung eines Entscheidungsbaumes aus einem Entscheidungsnetz, wobei aber im Allgemeinen der Baum von exponentieller Größe ist. Shachter (1986) entwickelte eine Methode für das Treffen von Entscheidungen, die direkt auf einem Entschei-

dungsnetz basiert, ohne einen Entscheidungsbaum als Zwischenstufe zu erstellen. Dieser Algorithmus war einer der ersten, der eine vollständige Inferenz für mehrfach verbundene Bayessche Netze unterstützte. Zhang et al. (1994) zeigten, wie sich bedingte Unabhängigkeit von Informationen nutzen lässt, um die Größe von Bäumen in der Praxis zu verringern; sie verwenden den Begriff *Entscheidungsnetz* für Netze, die nach diesem Ansatz arbeiten (obwohl andere ihn als Synonym für Einflussdiagramm verwenden). Nilsson und Lauritzen (2000) verknüpfen Algorithmen für Entscheidungsnetze für laufende Entwicklungen in Clustering-Algorithmen für Bayessche Netze. Koller und Milch (2003) zeigen, wie sich mithilfe von Einflussdiagrammen Spiele lösen lassen, die Informationen von Gegenspielern zusammentragen, während Detwarasiti und Shachter (2005) Einflussdiagramme als Hilfsmittel für die Entscheidungsfindung für ein Team einsetzen, das gemeinsame Ziele verfolgt, aber nicht in der Lage ist, alle Informationen perfekt gemeinsam zu nutzen. Die Sammlung von Oliver und Smith (1990) beinhaltet mehrere praktische Artikel über Entscheidungsnetze, ebenso wie die Sonderausgabe des *Networks*-Journals von 1990. In den Journalen *Management Science* und *Decision Analysis* erscheinen regelmäßig Arbeiten über Entscheidungsnetze und Nutzenmodellierung.

Die Informationswerttheorie wurde als Erstes im Kontext von statistischen Experimenten untersucht, wobei eine Art Nutzen (Entropieverringerung) verwendet wurde (Lindley, 1956). Der russische Kontrolltheoretiker Ruslan Stratonovich (1965) entwickelte die hier vorgestellte allgemeinere Theorie, in der Information aufgrund ihrer Fähigkeit, Entscheidungen zu beeinflussen, einen Wert hat. Die Arbeit von Stratonovich war in der westlichen Welt unbekannt, wo Ron Howard (1966) die gleiche Idee auf den Weg brachte. Seine Arbeit endet mit dem Kommentar: „Wenn die Informationswerttheorie sowie die zugehörigen entscheidungstheoretischen Strukturen in Zukunft nicht einen Großteil der Ausbildung von Ingenieuren ausmachen, dann wird der Beruf des Ingenieurs feststellen, dass seine traditionelle Rolle der Verwaltung wissenschaftlicher und wirtschaftlicher Ressourcen zum Nutzen der Menschheit an andere Berufe abgegeben wurde.“ Bis heute ist die implizierte Revolution der verwaltungstechnischen Methoden nicht eingetreten.

Eine neuere Arbeit von Krause und Guestrin (2009) zeigt, dass die Berechnung des genauen nichtweitsichtigen Wertes von Information selbst in Polybaumnetzen nicht in den Griff zu bekommen ist. In anderen Fällen – die eingeschränkter als der allgemeine Wert der Information sind – liefert der weitsichtige Algorithmus eine nachweisbar gute Annäherung an die optimale Beobachtungssequenz (Krause et al., 2008). In manchen Fällen – zum Beispiel bei der Suche nach einem Schatz, der an einer von n Stellen vergraben ist – liefert ein Ranking von Experimenten geordnet nach Erfolgswahrscheinlichkeit geteilt durch die Kosten eine optimale Lösung (Kadane und Simon, 1977).

Nach den frühen Anwendungen in der medizinischen Entscheidungsfindung (*Kapitel 13*) haben überraschend wenige KI-Forscher die entscheidungstheoretischen Werkzeuge angenommen. Eine der wenigen Ausnahmen war Jerry Feldman, der die Entscheidungstheorie auf Problemstellungen der Vision (Feldman und Yakimovsky, 1974) und Planung (Feldman und Sproull, 1977) anwendete. Mit dem wiederbelebten Interesse an probabilistischen Methoden in der KI in den 1980er Jahren gewannen entscheidungstheoretische Expertensysteme allgemeine Akzeptanz (Horvitz et al., 1988; Cowell et al., 2002). Ab 1991 wurde auf dem Cover des Journals *Artificial Intelligence* ein Entscheidungsnetz dargestellt, obwohl man sich für die Richtung der Pfeile offenbar eine gewisse künstlerische Freiheit herausgenommen hat.

Zusammenfassung

Dieses Kapitel zeigt, wie man die Nutzentheorie mit der Wahrscheinlichkeit kombiniert, um es einem Agenten zu ermöglichen, Aktionen auszuwählen, die seine erwartete Leistung maximieren.

- **Wahrscheinlichkeitstheorie** beschreibt, was ein Agent auf der Grundlage von Evidenzen glauben soll, **Nutzentheorie** beschreibt, was ein Agent will, und **Entscheidungstheorie** kombiniert die beiden, um zu beschreiben, was ein Agent tun soll.
- Wir können die Entscheidungstheorie nutzen, um ein System aufzubauen, das Entscheidungen trifft, indem es alle möglichen Aktionen betrachtet und diejenige auswählt, die zum besten erwarteten Ergebnis führt. Ein solches System wird auch als **rationaler Agent** bezeichnet.
- Die Nutzentheorie zeigt, dass ein Agent, dessen Prioritäten zwischen Lotterien konsistent mit einer Menge einfacher Axiome sind, eine Nutzenfunktion besitzt. Darüber hinaus wählt der Agent Aktionen so aus, dass er seinen erwarteten Nutzen maximiert.
- Die **Nutzentheorie mit Mehrfachattributen** beschäftigt sich mit Nutzen, die von verschiedenen Zustandsattributen abhängig sind. Die **stochastische Dominanz** ist eine besonders praktische Technik für die Findung eindeutiger Entscheidungen, selbst ohne genaue Nutzenwerte für Entscheidungen.
- **Entscheidungsnetze** stellen einen einfachen Formalismus für die Artikulation und Lösung von Entscheidungsproblemen dar. Es handelt sich dabei um eine natürliche Erweiterung Bayesscher Netze, die neben Zufallsknoten auch Entscheidungs- und Nutzenknoten beinhalten.
- Manchmal bedingt die Lösung eines Problems das Ermitteln weiterer Informationen, bevor eine Entscheidung getroffen werden kann. Der **Wert von Information** ist definiert als die erwartete Verbesserung des Nutzens im Vergleich zu einer Entscheidung ohne diese Information.
- **Expertensysteme**, die Nutzeninformation verwenden, besitzen zusätzliche Fähigkeiten im Vergleich zu reinen Inferenzsystemen. Sie können nicht nur Entscheidungen treffen, sondern auch den Wert der Information verwenden, um zu entscheiden, welche Fragen – falls vorhanden – zu stellen sind. Sie können Kontingenzpläne empfehlen und sie können die Sensibilität ihrer Entscheidungen im Hinblick auf kleine Änderungen der Wahrscheinlichkeiten sowie Nutzeneinschätzungen berechnen.

Übungen zu Kapitel 16

Lösungshinweise

1 (Übernommen aus David Heckerman). Diese Übung beschäftigt sich mit dem Spiel **Almanach**, das von Entscheidungsanalytikern verwendet wird, um numerische Schätzungen zu kalibrieren. Geben Sie für jede der folgenden Fragen Ihre beste Schätzung als Antwort, d.h. eine Zahl, die Sie genauso wahrscheinlich als zu hoch wie als zu niedrig ansehen. Geben Sie Ihre Schätzung auch bei einem 25. Perzentil an, d.h. einer Zahl, von der Sie zu 25% glauben, dass sie zu hoch ist, und zu 75%, dass sie zu niedrig ist. Machen Sie das Gleiche für das 75. Perzentil. (Sie sollten also für jede Frage insgesamt drei Schätzungen abgeben – niedrig, mittel und hoch.)

a. Anzahl der Passagiere, die 1989 zwischen New York und Los Angeles flogen
b. Bevölkerung von Warschau im Jahr 1992
c. Jahr, in dem Coronado den Mississippi entdeckte
d. Anzahl der Stimmen, die Jimmy Carter im Jahr 1976 bei der Präsidentschaftswahl erhielt
e. Alter des ältesten lebenden Baumes im Jahr 2002
f. Höhe des Hoover-Damms in Fuß
g. Anzahl der 1985 in Oregon produzierten Eier
h. Anzahl der Buddhisten auf der Welt im Jahr 1992
i. Anzahl der AIDS-Todesfälle in den Vereinigten Staaten im Jahr 1981
j. Anzahl der 1901 in den Vereinigten Staaten erteilten Patente

Die richtigen Antworten finden Sie nach der letzten Übung in diesem Kapitel. Aus der Perspektive der Entscheidungsanalyse ist nicht interessant, wie nahe Ihre mittleren Schätzungen an den richtigen Antworten liegen, sondern wie oft die richtige Antwort innerhalb Ihrer 25- bis 75%igen Grenzen lag. Wenn sie etwa in der Hälfte der Fälle in diesem Bereich lag, dann sind Ihre Grenzen korrekt. Wenn Sie jedoch wie die meisten Menschen sind, so sind Sie sich sicherer, als Sie sein sollten, und weniger als die Hälfte der Antworten liegt innerhalb der Grenzen. Mit ein bisschen Übung können Sie sich selbst auf die Angabe realistischer Grenzen kalibrieren und damit bessere Informationen für die Entscheidungsfindung bereitstellen. Versuchen Sie es mit dieser folgenden Menge an Fragen und vergleichen Sie, ob es eine Verbesserung gegeben hat:

a. Geburtsjahr von Zsa Zsa Gabor
b. Maximale Entfernung vom Mars zur Sonne, angegeben in Meilen
c. Dollarwert der Weizenexporte der Vereinigten Staaten im Jahr 1992
d. Gütertonnen, die 1991 im Hafen von Honolulu umgeschlagen wurden
e. Jahreseinkommen des Gouverneurs von Kalifornien im Jahr 1993, angegeben in Dollar
f. Bevölkerung von San Diego im Jahr 1990
g. Jahr, in dem Roger Williams Providence, Rhode Island, gründete
h. Höhe des Kilimandscharo in Fuß
i. Länge der Brooklyn Bridge in Fuß
j. Anzahl der Todesfälle durch Autounfälle in den Vereinigten Staaten im Jahr 1992

2 Chris sieht sich 5 Gebrauchtwagen an, bevor er einen mit dem maximal erwarteten Nutzen kauft. Pat sieht sich 11 gebrauchte Autos an und tut dann das Gleiche. Welcher von beiden hat wahrscheinlich den besseren Wagen gekauft, wenn man alle anderen Dinge als gleich annimmt? Wer wird wahrscheinlicher von der Qualität seines Autos enttäuscht sein? Geben Sie die Größen in Form der Standardabweichungen für die erwartete Qualität an.

3 Im Jahr 1713 gab Nicolas Bernoulli ein Rätsel an, das man heute als das St.-Petersburg-Paradoxon bezeichnet und das wie folgt lautet: Sie haben die Möglichkeit, ein Spiel mit einer fairen Münze zu spielen, die wiederholt geworfen wird, bis sie auf Kopf zu liegen kommt. Wenn beim n-ten Wurf zum ersten Mal Kopf erscheint, gewinnen Sie 2^n Dollar.

a. Zeigen Sie, dass der erwartete monetäre Wert dieses Spieles unendlich ist.

b. Wie viel würden Sie persönlich zahlen, um das Spiel zu spielen?

c. Daniel Bernoulli (der Cousin von Nicolas) löste das offensichtliche Paradoxon 1738 auf, indem er vorschlug, dass der Nutzen des Geldes anhand einer logarithmischen Skala, d.h. $U(S_n) = a \log_2 n + b$, bemessen wird, wobei S_n der Zustand ist, \$$n$ zu besitzen. Welchen erwarteten Nutzen hat das Spiel unter dieser Annahme?

d. Welcher maximale Betrag ist als Einsatz, das Spiel spielen zu dürfen, rational, vorausgesetzt man hat den anfänglichen Besitz \$$k$?

4 Schreiben Sie ein Computerprogramm zur Automatisierung des Prozesses in Übung 16.8. Probieren Sie Ihr Programm für mehrere Menschen mit unterschiedlichem Besitz und Sozialstand aus. Kommentieren Sie die Konsistenz Ihrer Ergebnisse sowohl für einen einzelnen Menschen als auch für alle Menschen.

5 Die Firma Surprise Candy liefert Bonbons mit zwei Geschmacksrichtungen: 75% mit Erdbeer- und 25% mit Anchovisgeschmack. Zunächst hat jedes Stück eine runde Form. Im Verlauf der Produktionsstraße wählt eine Maschine zufällig einen bestimmten Prozentsatz aus, der in eine Würfelform gebracht wird. Dann wird jedes Stück in Bonbonpapier eingewickelt, dessen Farbe zufällig unter Rot und Braun ausgewählt wird. 70% der Erdbeerbonbons sind rund und 70% haben rotes Bonbonpapier, während 90% der Anchovisbonbons würfelförmig sind und 90% in braunes Papier eingewickelt sind. Alle Bonbons werden einzeln in geschlossenen, identischen schwarzen Schachteln verkauft.

Nun haben Sie als Kunde ein Überraschungsbonbon im Laden gekauft, die Schachtel aber noch nicht geöffnet. Sehen Sie sich dazu die drei Bayesschen Netze in ▶ Abbildung 16.11 an.

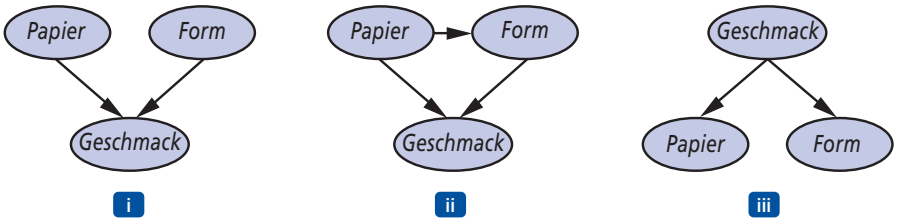


Abbildung 16.11: Drei vorgeschlagene Bayessche Netze für das Überraschungsbonbonproblem von Übung 16.5.

a. Welches Netz kann $\mathbf{P}(Geschmack, Papier, Form)$ korrekt darstellen? (Geben Sie alle zutreffenden Netze an.)

b. Welches Netz ist die beste Darstellung für dieses Problem?

c. Sichert Netz (i) zu, dass $\mathbf{P}(Papier|Form) = \mathbf{P}(Papier)$ ist?

d. Wie groß ist die Wahrscheinlichkeit, dass Ihr Bonbon in rotes Papier eingewickelt ist?

e. Die Schachtel enthält ein rundes Bonbon mit rotem Papier. Wie groß ist die Wahrscheinlichkeit, dass es ein Bonbon mit Erdbeergeschmack ist?

f. Ein ausgewickeltes Erdbeerbonbon hat auf dem freien Markt einen Wert von e und ein ausgewickeltes Anchovisbonbon einen Wert von a. Schreiben Sie einen Ausdruck für den Wert einer ungeöffneten Bonbonschachtel.

g. Ein neues Gesetz verbietet den Handel mit ausgewickelten Bonbons, doch ist es immer noch zulässig, eingewickelte Bonbons (aus der Schachtel heraus) zu handeln. Ist eine ungeöffnete Bonbonschachtel nun mehr oder weniger wert als zuvor oder hat sie den gleichen Wert?

6 Beweisen Sie, dass die Beurteilungen $B \succ A$ und $C \succ D$ im Allais-Paradoxon (*Abschnitt 16.3.4*) das Axiom der Ersetzbarkeit verletzen.

7 Betrachten Sie das Allais-Paradoxon (*Abschnitt 16.3.4*): Ein Agent, der B gegenüber A bevorzugt (die sichere Sache wählt) und C gegenüber D (die Sache mit dem höheren EMW nimmt), handelt entsprechend der Nutzentheorie nicht rational. Weist das Ihrer Meinung nach auf ein Problem für den Agenten, ein Problem für die Theorie oder auf überhaupt kein Problem hin? Erläutern Sie Ihre Antwort.

8 Bewerten Sie Ihren eigenen Nutzen für verschiedene schrittweise erhöhte Geldbeträge, indem Sie eine Reihe von Prioritätstests zwischen einem bestimmten Betrag M_1 und einer Lotterie $[p, M_2; (1-p), 0]$ ausführen. Wählen Sie verschiedene Werte von M_1 und M_2 und variieren Sie p, bis Sie zwischen den beiden Auswahlen unentschlossen sind. Stellen Sie die resultierende Nutzenfunktion grafisch dar.

9 Wie viel ist Ihnen ein Mikromort wert? Leiten Sie ein Protokoll ab, um diesen Betrag zu ermitteln. Stellen Sie Fragen sowohl dazu, wie viel gezahlt wird, um ein Risiko zu vermeiden, als auch dazu, wie viel gezahlt wird, um ein Risiko zu akzeptieren.

10 Nehmen Sie stetige Variablen $X_1, ..., X_k$ an, die unabhängig entsprechend der gleichen Wahrscheinlichkeitsdichtefunktion $f(x)$ verteilt sind. Beweisen Sie, dass die Dichtefunktion für $\max\{X_1, ..., X_k\}$ durch $kf(x)(F(x))^{k-1}$ gegeben ist, wobei F die kumulative Verteilung für f ist.

11 Wirtschaftswissenschaftler verwenden eine exponentielle Nutzenfunktion für Geld: $U(x) = -e^{x/R}$, wobei R eine positive Konstante ist, die die Risikobereitschaft einer Person darstellt. In der Risikobereitschaft spiegelt sich wider, wie eine Person eine Lotterie mit einem bestimmten erwarteten monetären Wert (EMW) im Vergleich zu einer bestimmten Auszahlung akzeptiert. Wenn R (in denselben Einheiten wie x gemessen) wächst, wird die Person weniger risikoscheu.

a. Mary habe eine exponentielle Nutzenfunktion mit $R = 400$ € und darf wählen, ob sie 400 € mit Sicherheit (Wahrscheinlichkeit 1) erhält oder an einer Lotterie teilnimmt, bei der sie mit einer Wahrscheinlichkeit von 60% einen Betrag von 5000 € gewinnt und mit einer Wahrscheinlichkeit von 40% leer ausgeht. Welche Option wird Mary wählen, wenn man annimmt, dass sie rational handelt? Zeigen Sie, wie Sie Ihre Antwort hergeleitet haben.

b. Betrachten Sie die Auswahlmöglichkeiten zwischen dem sicheren Erhalt von 100 € (Wahrscheinlichkeit 1) oder der Teilnahme an einer Lotterie, bei der mit einer Wahrscheinlichkeit von 50% ein Gewinn von 500 € und mit einer Wahrscheinlichkeit von 50% gar kein Gewinn erzielt wird. Nähern Sie den Wert von R (auf 3 signifikante Stellen) in einer exponentiellen Nutzenfunktion an, die dazu führt, dass eine Person zwischen diesen beiden Alternativen unentschlossen ist. (Es dürfte hilfreich sein, ein kurzes Programm zu schreiben, das Sie bei der Lösung dieses Problems unterstützt.)

12 Alex hat die Wahl zwischen zwei Spielen. In Spiel 1 wird eine faire Münze geworfen und Alex erhält 100 €, wenn sie auf Kopf fällt. Fällt sie auf Zahl, bekommt Alex nichts. In Spiel 2 wird eine faire Münze zweimal geworfen. Jedes Mal, wenn Kopf fällt, erhält Alex 50 €, und für jeden Wurf, der Zahl zeigt, bekommt Alex nichts. Nehmen Sie an, dass Alex eine monoton steigende Nutzenfunktion für Geld im Bereich [0 €, 100 €] hat. Zeigen Sie mathematisch, dass Alex risikoscheu ist (zumindest in Bezug auf diesen Bereich von Geldbeträgen), wenn er Spiel 2 gegenüber Spiel 1 vorzieht.

13 Zeigen Sie, dass, wenn X_1 und X_2 prioritätsunabhängig von X_3 sind und X_2 und X_3 prioritätsunabhängig von X_1, dann auch X_3 und X_1 prioritätsunabhängig von X_2 sind.

14 Diese Übung vervollständigt die Analyse des Flughafenstandortproblems in Abbildung 16.6.

a. Stellen Sie sinnvolle Variablendomänen, Wahrscheinlichkeiten und Nutzen für das Netz bereit und gehen Sie davon aus, dass es drei mögliche Standorte gibt.

b. Lösen Sie das Entscheidungsproblem.

c. Was passiert, wenn durch Technologiefortschritte ein Flugzeug nur noch die Hälfte an Lärmemissionen verursacht?

d. Was passiert, wenn der Lärmschutz dreimal so wichtig wird?

e. Berechnen Sie die WPI für $Flugaufkommen$, $Rechtsstreitigkeiten$ und Bau in Ihrem Modell.

15 Wiederholen Sie Übung 16.14 unter Verwendung der in Abbildung 16.7 gezeigten Aktion/Nutzen-Repräsentation.

16 Geben Sie für die Flughafenstandort-Diagramme der Übungen 16.14 und 16.15 an, für welchen Eintrag der bedingten Wahrscheinlichkeitstabelle der Nutzen am sensibelsten ist, wenn die verfügbare Evidenz gegeben ist?

17 Ändern und erweitern Sie den Code für das Bayessche Netz aus dem Code-Repository, um das Erstellen und Auswerten von Entscheidungsnetzen sowie die Berechnung von Informationswert zu berücksichtigen.

18 Betrachten Sie einen Studenten, der die Wahl hat, ein Lehrbuch für einen Kurs zu kaufen oder nicht. Wir modellieren dies als Entscheidungsproblem mit einem booleschen Entscheidungsknoten B (der anzeigt, ob sich der Agent für den Kauf des Buches entscheidet) sowie zwei booleschen Zufallsknoten M und P (wobei M angibt, ob der Student den Stoff im Buch beherrscht, und P angibt, ob der Student den Kurs besteht). Selbstverständlich gibt es auch einen Nutzenknoten U. Ein bestimmter Student namens Sam hat eine additive Nutzenfunktion: 0 für den Kauf des Buches und -100 € für den Nichtkauf sowie 2000 € für das Bestehen des Kurses und 0 für das Nichtbestehen. Die bedingten Wahrscheinlichkeitsschätzungen für Sam lauten:

$P(p|b, m) = 0{,}9$ $\quad P(m|b) = 0{,}9$
$P(p|b, \neg m) = 0{,}5$ $\quad P(m|\neg b) = 0{,}7$
$P(p|\neg b, m) = 0{,}8$
$P(p|\neg b, \neg m) = 0{,}3$

Man könnte meinen, dass P unabhängig von B bei gegebenem M ist. Doch sind bei der Abschlussprüfung in diesem Kurs Hilfsmittel erlaubt, sodass es hilfreich ist, das Buch dabeizuhaben.

a. Zeichnen Sie das Entscheidungsnetz für dieses Problem.

b. Berechnen Sie den erwarteten Nutzen dafür, das Buch zu kaufen, und dafür, es nicht zu kaufen.

c. Was sollte Sam tun?

19 (Übernommen aus Pearl (1988).) Ein Gebrauchtwagenkäufer beschließt, verschiedene Tests zu verschiedenen Kosten (z.B. gegen die Reifen treten, das Auto zu einem qualifizierten Mechaniker bringen) durchzuführen und dann abhängig von deren Ergebnissen zu entscheiden, welches Auto gekauft werden soll. Wir gehen davon aus, dass der Käufer entscheidet, ob er Auto c_1 kaufen will, dass die Zeit für höchstens einen Test reicht und dass t_1 der Test für Auto c_1 ist und 50 € kostet.

Ein Auto kann sich in gutem Zustand (Qualität q^+) oder in schlechtem Zustand (Qualität q^-) befinden und die Tests können helfen zu erkennen, in welchem Zustand sich ein Auto befindet. Auto c_1 kostet 1500 € und sein Marktwert liegt bei 2000 €, wenn es sich in gutem Zustand befindet; andernfalls fallen 700 € für Reparaturen an, um es in einen guten Zustand zu versetzen. Die Schätzung des Käufers ist, dass c_1 zu 70% in gutem Zustand ist.

a. Zeichnen Sie das Entscheidungsnetz für dieses Problem.

b. Berechnen Sie den erwarteten Nettogewinn, wenn c_1 ohne Test gekauft wird.

c. Tests können durch die Wahrscheinlichkeit beschrieben werden, dass das Auto den Test besteht oder nicht besteht, vorausgesetzt, es befindet sich in gutem oder schlechtem Zustand. Wir haben die folgende Information:

$$P(besteht(c_1, t_1)|q^+(c_1)) = 0{,}8$$
$$P(besteht(c_1, t_1)|q^+(c_1)) = 0{,}35.$$

Verwenden Sie das Bayessche Theorem zur Berechnung der Wahrscheinlichkeit, dass das Auto den Test besteht (oder nicht besteht), und damit der Wahrscheinlichkeit, dass es sich in gutem (oder schlechtem) Zustand befindet, für jedes mögliche Testergebnis.

d. Berechnen Sie die optimalen Entscheidungen für das Bestehen oder Nichtbestehen sowie ihren erwarteten Nutzen.

e. Berechnen Sie den Informationswert des Testes und leiten Sie einen optimalen bedingten Plan für den Käufer ab.

20 Wiederholen Sie die Definition des Informationswertes in *Abschnitt 16.6*.

a. Beweisen Sie, dass der Informationswert nicht negativ und von der Reihenfolge unabhängig ist.

b. Erklären Sie, warum manche Personen es vorziehen, bestimmte Informationen nicht zu erhalten – zum Beispiel das Geschlecht ihres Babys bei einer Ultraschalluntersuchung nicht erfahren möchten.

c. Eine Funktion f auf Mengen ist **submodular** genau dann, wenn für jedes Element x und beliebige Mengen A und B mit $A \subseteq B$ gilt, dass das Addieren von x zu A einen größeren Zuwachs in f ergibt als das Addieren von x zu B:

$$A \subseteq B \Rightarrow (f(A \cup \{x\}) - f(A)) \geq (f(B \cup \{x\}) - f(B)).$$

Submodularität erfasst das intuitive Konzept von abnehmenden Ertragszuwächsen. Ist der Wert von Information, betrachtet als Funktion f auf Mengen möglicher Beobachtungen, submodular? Beweisen Sie diese Aussage oder finden Sie ein Gegenbeispiel.

Die Antworten zu Übung 16.1 (M steht für Millionen): Erste Menge: 3M, 1,6M, 1541, 41M, 4768, 221, 649M, 295M, 132, 25546. Zweite Menge: 1917, 155M, 4500M, 11M, 120000, 1,1M, 1636, 19340, 1595, 41710.

Komplexe Entscheidungen

17

ÜBERBLICK

In diesem Kapitel betrachten wir Methoden für die Entscheidung, was zu tun ist, wenn wir bereits heute wissen, dass wir morgen wieder entscheiden müssen.

In diesem Kapitel geht es um die rechentechnischen Probleme bei Entscheidungen in einer stochastischen Umgebung. Während sich *Kapitel 16* mit einmaligen oder episodischen Entscheidungsproblemen beschäftigte, wobei der Nutzen jedes Aktionsergebnisses bekannt war, betrachten wir hier **sequentielle Entscheidungsprobleme**, bei denen der Nutzen eines Agenten von einer Folge von Entscheidungen abhängig ist. Sequentielle Entscheidungsprobleme, die Nutzen, Unsicherheit und Sensoren beinhalten, binden auch Such- und Planungsprobleme als Spezialfälle ein. *Abschnitt 17.1* erklärt, wie sequentielle Entscheidungsprobleme definiert sind, und die *Abschnitte 17.2* und *17.3* erläutern, wie sie gelöst werden können, um ein optimales Verhalten zu erzeugen, das die Risiken und Gewinne beim Handeln in einer unsicheren Umgebung ausgleicht. *Abschnitt 17.4* erweitert diese Konzepte auf den Fall partiell beobachtbarer Umgebungen und *Abschnitt 17.4.3* entwickelt ein vollständiges Design für entscheidungstheoretische Agenten in partiell beobachtbaren Umgebungen, wobei die dynamischen Bayesschen Netze aus *Kapitel 15* mit den Entscheidungsnetzen aus *Kapitel 16* kombiniert werden.

Der zweite Teil des Kapitels deckt Umgebungen mit mehreren Agenten ab. In diesen Umgebungen wird das Konzept eines optimalen Verhaltens durch die Interaktionen zwischen den Agenten verkompliziert. *Abschnitt 17.5* stellt die wichtigsten Konzepte der **Spieltheorie** vor, unter anderem das Konzept, dass sich rationale Agenten möglicherweise zufällig verhalten müssen. *Abschnitt 17.6* betrachtet, wie Multiagenten-Systeme entworfen werden können, sodass mehrere Agenten ein gemeinsames Ziel erreichen.

17.1 Sequentielle Entscheidungsprobleme

Angenommen, ein Agent befindet sich in der in ▶ Abbildung 17.1(a) gezeigten 4×3-Umgebung. Beginnend vom Anfangszustand muss er in jedem Zeitschritt eine Aktion auswählen. Die Interaktion mit der Umgebung endet, wenn der Agent einen der Zielzustände erreicht hat, die mit +1 oder –1 markiert sind. Genau wie für Suchprobleme sind die Aktionen für den Agenten in jedem Zustand durch ACTIONS(s) gegeben, manchmal als $A(s)$ abgekürzt; in der 4×3-Umgebung heißen die Aktionen in jedem Zustand *Oben*, *Unten*, *Links* und *Rechts*. Wir gehen hier davon aus, dass die Umgebung **vollständig beobachtbar** ist, sodass der Agent immer weiß, wo er ist.

Wäre die Umgebung deterministisch, so wäre die Lösung einfach: [*Oben*, *Oben*, *Rechts*, *Rechts*, *Rechts*]. Leider entspricht die Umgebung nicht immer dieser Lösung, weil die Aktionen unzuverlässig sind. Das Modell der stochastischen Bewegung, das wir hier anwenden, ist in ▶ Abbildung 17.1(b) gezeigt. Jede Aktion erreicht den gewünschten Effekt mit der Wahrscheinlichkeit 0,8, aber in der restlichen Zeit bewegt die Aktion den Agenten in rechtwinkligen Bewegungen in die gewünschte Richtung. Stößt der Agent darüber hinaus gegen eine Mauer, bleibt er im selben Quadrat. Vom Anfangsquadrat (1, 1) aus bringt die Aktion *Oben* den Agenten mit der Wahrscheinlichkeit 0,8 nach (1, 2), aber mit der Wahrscheinlichkeit 0,1 bewegt er sich nach rechts zu (2, 1) und mit der Wahrscheinlichkeit 0,1 bewegt er sich nach links, prallt gegen die Mauer und bleibt in (1, 1). In einer solchen Umgebung umgeht die Folge [*Oben*, *Oben*, *Rechts*, *Rechts*, *Rechts*] das Hindernis und gelangt mit der Wahrscheinlichkeit $0{,}8^5 = 0{,}32768$ zum Zielzustand in (4, 3). Außerdem gibt es die kleine Chance, das Ziel zufällig zu erreichen,

indem der andere Weg gewählt wird – mit einer Wahrscheinlichkeit von $0{,}1^4 \times 0{,}8$, was dann insgesamt 0,32776 ergibt (siehe auch Übung 17.1).

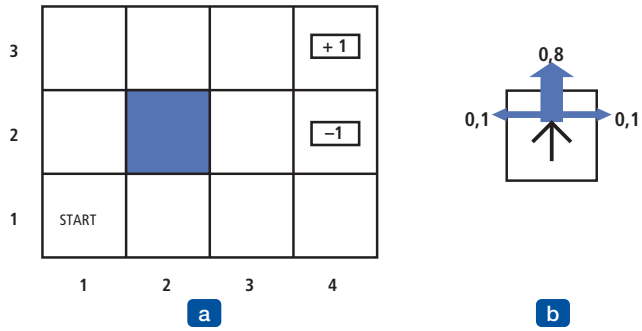


Abbildung 17.1: (a) Eine einfache 4 × 3-Umgebung, die den Agenten vor ein sequentielles Entscheidungsproblem stellt. (b) Darstellung des Übergangsmodells der Umgebung: Das „gewünschte" Ergebnis tritt mit der Wahrscheinlichkeit 0,8 auf, aber der Agent bewegt sich mit der Wahrscheinlichkeit von 0,2 rechtwinklig in die gewünschte Richtung. Eine Kollision mit einer Mauer bewirkt keine Bewegung. Die beiden Endzustände haben den Gewinn +1 bzw. –1. Alle anderen Zustände haben den Gewinn –0,04.

Wie in *Kapitel 3* beschreibt das **Übergangsmodell** (oder einfach nur „Modell", wenn keine Verwechslungsgefahr besteht) das Ergebnis jeder Aktion in jedem Zustand. Da hier das Ergebnis stochastisch ist, bezeichnen wir mit $P(s' \mid s, a)$ die Wahrscheinlichkeit, den Zustand s' zu erreichen, wenn die Aktion a im Zustand s ausgeführt wird. Wir gehen davon aus, dass es sich um **Markov**-Übergänge im Sinne von *Kapitel 15* handelt, d.h. die Wahrscheinlichkeit, s' von s aus zu erreichen, ist nur von s abhängig und nicht vom Verlauf der vorherigen Zustände. Fürs Erste können Sie sich $P(s' \mid s, a)$ als große dreidimensionale Tabelle mit Wahrscheinlichkeiten vorstellen. Später in *Abschnitt 17.4.3* werden wir sehen, dass das Übergangsmodell als **dynamisches Bayessches Netz** dargestellt werden kann, so wie in *Kapitel 15* angesprochen.

Um die Definition der Aufgabenumgebung zu vervollständigen, müssen wir die Nutzenfunktion für den Agenten spezifizieren. Weil das Entscheidungsproblem sequentiell ist, ist die Nutzenfunktion von einer Zustandsfolge abhängig – von einem sogenannten **Umgebungsverlauf** – und nicht von einem einzelnen Zustand. Später in diesem Abschnitt werden wir betrachten, wie solche Nutzenfunktionen im Allgemeinen spezifiziert werden können; hier vereinbaren wir einfach, dass der Agent in jedem Zustand s einen **Gewinn** $R(s)$ macht, der positiv oder negativ sein kann, aber begrenzt sein muss. Für unser aktuelles Beispiel ist der Gewinn in allen Zuständen gleich $-0{,}04$, außer in den Endzuständen (wo es die Gewinne $+1$ und -1 gibt). Der Nutzen eines Umgebungsverlaufes ist (hier) einfach nur die *Summe* der erhaltenen Gewinne. Erreicht der Agent beispielsweise nach zehn Schritten den Zustand $+1$, ist der Gesamtnutzen gleich 0,6. Der negative Gewinn von $-0{,}04$ gibt dem Agenten den Ansporn, (4, 3) schnell zu erreichen, sodass unsere Umgebung eine stochastische Verallgemeinerung der Suchprobleme aus *Kapitel 3* ist. Man könnte auch sagen, der Agent genießt das Leben in dieser Umwelt nicht und versucht deshalb, so schnell wie möglich wegzukommen.

Fassen wir zusammen: Ein sequentielles Entscheidungsproblem für eine vollständig beobachtbare, stochastische Umgebung mit einem Markov-Übergangsmodell und additiven Gewinnen heißt **Markov-Entscheidungsprozess** oder **MEP** und besteht aus einem Satz von Zuständen (mit einem Anfangszustand s_0), einem Satz ACTIONS(s) von Aktio-

nen in jedem Zustand, einem Übergangsmodell $P(s'|s, a)$ und einer Gewinnfunktion $R(s)$.[1]

Die nächste Frage ist, wie eine Lösung für das Problem aussieht. Wir haben gesehen, dass eine feste Aktionsfolge das Problem nicht löst, weil der Agent zu einem anderen Zustand als dem Ziel gelangen kann. Aus diesem Grund muss eine Lösung angeben, was der Agent für *jeden* Zustand tun soll, den er möglicherweise erreicht. Eine Lösung dieser Art wird als **Taktik** bezeichnet. Traditionell bezeichnen wir eine Taktik mit π und $\pi(s)$ ist die von der Taktik π empfohlene Aktion für den Zustand s. Wenn der Agent eine vollständige Taktik besitzt, spielt es keine Rolle, wie das Ergebnis irgendeiner Aktion aussieht – der Agent weiß immer, was als Nächstes zu tun ist.

Jedes Mal, wenn eine bestimmte Taktik vom Ausgangszustand aus ausgeführt wird, kann die stochastische Natur der Umgebung zu einem anderen Umgebungsverlauf führen. Die Qualität einer Taktik wird deshalb durch den *erwarteten* Nutzen der möglichen Umgebungsverläufe bewertet, der durch diese Taktik entsteht. Mit π^* bezeichnen wir eine optimale Taktik. Für ein bekanntes π^* entscheidet der Agent, was zu tun ist, indem er seine aktuelle Wahrnehmung befragt. Er erfährt dadurch, dass er sich im aktuellen Zustand s befindet, und kann dann die Aktion $\pi^*(s)$ ausführen. Eine Taktik repräsentiert die Agentenfunktion explizit und ist deshalb eine Beschreibung eines einfachen Reflexagenten, der aus der für einen nutzenbasierten Agenten verwendeten Information berechnet wird.

▶ Abbildung 17.2(a) zeigt eine optimale Taktik für die Welt aus Abbildung 17.1. Weil die Kosten für einen Schritt verglichen mit der Bestrafung, versehentlich in (4, 2) zu landen, relativ gering sind, ist die optimale Taktik für den Zustand (3, 1) konservativ. Die Taktik empfiehlt, den langen Umweg zu nehmen statt der Abkürzung – die riskiert, (4, 2) zu erreichen.

Der Ausgleich zwischen Risiko und Gewinnänderungen ist vom Wert von $R(s)$ für die nicht terminalen Zustände abhängig. ▶ Abbildung 17.2(b) zeigt optimale Taktiken für die vier verschiedenen Bereiche von $R(s)$. Wenn $R(s) \leq -1{,}6284$ ist, dann ist das Leben so schrecklich, dass der Agent direkt auf den nächsten Ausgang zuläuft, selbst wenn dieser -1 wert ist. Wenn $-0{,}4278 \leq R(s) \leq -0{,}0850$, ist das Leben relativ unbequem; der Agent nimmt den kürzesten Weg zum Zustand $+1$ und akzeptiert damit das Risiko, versehentlich in den Zustand -1 zu gelangen. Insbesondere nimmt der Agent die Abkürzung von (3, 1) aus. Wenn das Leben nur ein wenig traurig ist ($-0{,}0221 < R(s) < 0$), geht die optimale Taktik *keinerlei* Risiken ein. In (4, 1) und (3, 2) bewegt sich der Agent direkt vom Zustand -1 weg, sodass er ihn nicht versehentlich erreicht, auch wenn das bedeutet, dass er sich den Kopf vielleicht ein paar Mal an der Wand stößt. Ist schließlich $R(s) > 0$, so ist das Leben zu genießen und der Agent vermeidet *beide* Ausgänge. Solange die Aktionen in (4, 1), (3, 2) und (3, 3) wie gezeigt aussehen, ist jede Taktik optimal und der Agent erhält einen unendlichen Gesamtgewinn, weil er nie einen Terminalzustand erreicht. Überraschenderweise stellt sich heraus, dass es sechs weitere optimale Taktiken für verschiedene Bereiche von $R(s)$ gibt; in Übung 17.57 werden Sie versuchen, sie zu finden.

1 Einige Definitionen von MEPs erlauben, dass der Gewinn von der Aktion und ebenso vom Ergebnis abhängig ist; die Gewinnfunktion ist dann also $R(s, a, s')$. Das vereinfacht die Beschreibung einiger Umgebungen, ändert aber das Problem nicht grundsätzlich, wie Übung 17.4 zeigt.

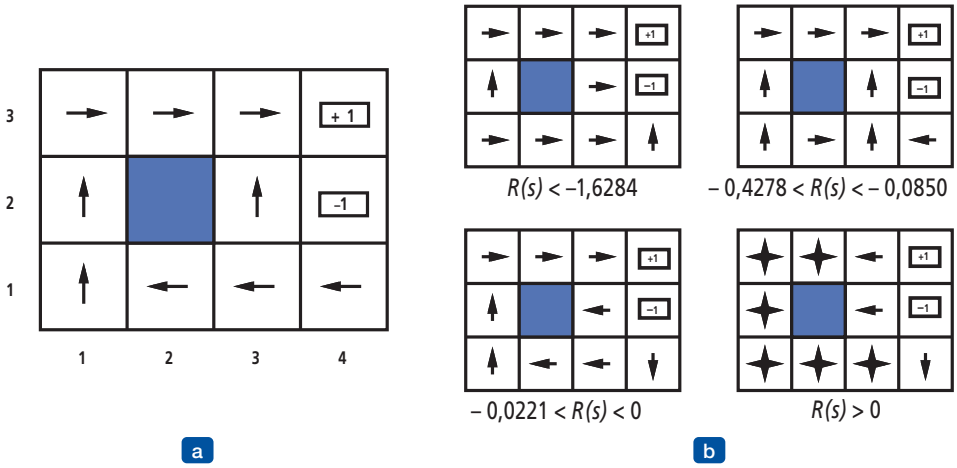


Abbildung 17.2: (a) Eine optimale Taktik für die stochastische Umgebung mit $R(s) = -0{,}04$ in den nicht terminalen Zuständen. (b) Optimale Taktiken für vier verschiedene Bereiche von $R(s)$.

Die sorgfältige Balance zwischen Risiko und Gewinn ist eine charakteristische Eigenschaft von MEPs, die in deterministischen Suchproblemen nicht auftritt. Darüber hinaus ist sie eine charakteristische Eigenschaft vieler Entscheidungsprobleme aus der realen Welt. Aus diesem Grund wurden MEPs in den verschiedensten Bereichen genauer untersucht, unter anderem in der KI, der Operationsforschung, der Wirtschaftswissenschaft und der Steuerungstheorie. Für die Berechnung optimaler Taktiken wurden Dutzende von Algorithmen vorgeschlagen. In den *Abschnitten 17.2* und *17.3* beschreiben wir zwei der wichtigsten Algorithmenfamilien. Zuerst müssen wir jedoch unsere Betrachtung der Nutzen und Taktiken für sequentielle Entscheidungsprobleme vervollständigen.

17.1.1 Nutzer über der Zeit

In dem MEP-Beispiel aus Abbildung 17.1 wurde die Leistung des Agenten durch eine Summe der Gewinne für die besuchten Zustände bewertet. Diese Auswahl der Leistungsbewertung geschieht nicht willkürlich, ist aber nicht die einzige Möglichkeit für die Nutzenfunktion in Umgebungsverläufen, die wir als $U_h([s_0, s_1, ..., s_n])$ darstellen. Unsere Analyse bezieht sich auf die Nutzentheorie mit mehreren Attributen (*Abschnitt 16.4*) und ist eher technisch orientiert; der ungeduldige Leser kann durchaus zum nächsten Abschnitt übergehen.

Tipp

Die erste zu beantwortende Frage ist, ob es einen **endlichen Horizont** oder einen **unendlichen Horizont** für die Entscheidungsfindung gibt. Ein endlicher Horizont bedeutet, dass es eine *feste* Zeit N gibt, nach der nichts mehr eine Rolle spielt – Game over. Somit gilt $U_h([s_0, s_1, ..., s_{N+k}]) = U_h([s_0, s_1, ..., s_N])$ für alle $k > 0$. Nehmen wir beispielsweise an, ein Agent beginnt in der 4 × 3-Welt von Abbildung 17.1 auf dem Quadrat (3, 1). Außerdem sei $N = 3$. Um die Chance zu haben, den Zustand +1 zu erreichen, muss der Agent direkt darauf zugehen und die optimale Aktion ist *Oben*. Ist andererseits $N = 100$, dann ist viel Zeit, die sichere Route zu wählen, indem der Agent *Links* geht. *Bei einem endlichen Horizont könnte sich also die optimale Aktion für einen bestimmten*

Zustand mit der Zeit ändern. Wir sagen, die optimale Taktik für einen endlichen Horizont ist **nicht stationär**. Ohne festes Zeitlimit gibt es dagegen keinen Grund, sich zu unterschiedlichen Zeiten im selben Zustand unterschiedlich zu verhalten. Folglich hängt die optimale Aktion nur vom aktuellen Zustand ab und die optimale Taktik ist **stationär**. Taktiken für den Fall der unendlichen Horizonte sind deshalb einfacher als für den Fall der endlichen Horizonte und wir werden uns in diesem Kapitel hauptsächlich mit den unendlichen Horizonten beschäftigen. (Später werden wir sehen, dass der Fall des unendlichen Horizonts für partiell beobachtbare Umgebungen nicht so einfach ist.) Beachten Sie, dass der „unendliche Horizont" nicht unbedingt bedeutet, dass alle Zustandsfolgen unendlich sind; er bedeutet nur, dass es keine feste Deadline gibt. Insbesondere kann es endliche Zustandsfolgen in einem MEP mit unendlichem Horizont geben, der einen Terminalzustand enthält.

Die nächste Frage ist, wie der Nutzen von Zustandsfolgen berechnet wird. In der Terminologie der **Nutzentheorie mit Mehrfachattributen** lässt sich jeder Zustand s_i als **Attribut** der Zustandsfolge $[s_0, s_1, s_2...]$ betrachten. Um einen einfachen Ausdruck für die Attribute zu erhalten, müssen wir eine Art Prioritätsunabhängigkeitsannahme treffen. Die natürlichste Annahme ist, dass die Prioritäten des Agenten zwischen den Zustandsfolgen **stationär** sind. Die Stationarität für Prioritäten bedeutet Folgendes: Wenn zwei Zustandsfolgen, $[s_0, s_1, s_2...]$ und $[s_0', s_1', s_2'...]$ mit demselben Zustand beginnen (d.h. $s_0 = s_0'$), dann sollten die beiden Folgen auf dieselbe Weise prioritätssortiert werden wie die Folgen $[s_1, s_2...]$ und $[s_1', s_2'...]$. Das bedeutet im Klartext, wenn Sie eine Zukunft einer anderen, die morgen beginnt, vorziehen, sollten Sie diese Zukunft auch bevorzugen, wenn sie heute beginnen würde. Stationarität ist eine recht harmlos aussehende Annahme mit sehr heftigen Konsequenzen: Es zeigt sich, dass es unter Stationarität nur zwei schlüssige Möglichkeiten gibt, Folgen Nutzen zuzuweisen:

1. **Additive Gewinne**: Der Nutzen einer Zustandsfolge ist:

$$U_h([s_0, s_1, s_2 ...]) = R(s_0) + R(s_1) + R(s_2) + ...$$

   Die 4 × 3-Welt in Abbildung 17.1 verwendet additive Gewinne. Beachten Sie, dass die Additivität in unserer Anwendung von Pfadkostenfunktionen für heuristische Suchalgorithmen (*Kapitel 3*) implizit verwendet wurde.

2. **Verminderte Gewinne**: Der Nutzen einer Zustandsfolge ist:

$$U_h([s_0, s_1, s_2 ...]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + ...$$

   Dabei ist der **Verminderungsfaktor** γ eine Zahl zwischen 0 und 1. Der Verminderungsfaktor beschreibt die Priorität eines Agenten für aktuelle Gewinne gegenüber zukünftigen Gewinnen. Wenn γ nahe 0 liegt, werden Gewinne in der fernen Zukunft als unbedeutend erachtet. Ist γ gleich 1, sind die verminderten Gewinne gleich den additiven Gewinnen; die additiven Gewinne sind also ein Sonderfall der verminderten Gewinne. Die Verminderung scheint ein gutes Modell sowohl der menschlichen als auch der tierischen Prioritäten über die Zeit zu sein. Ein Verminderungsfaktor von γ ist äquivalent mit einem Zinssatz von $(1/\gamma)-1$.

Aus Gründen, die gleich klarer sein werden, gehen wir im restlichen Kapitel von verminderten Gewinnen aus, obwohl wir manchmal $\gamma = 1$ zulassen.

Das Zögern bei unserer Auswahl unendlicher Horizonte stellt ein Problem dar: Wenn die Umgebung keinen Terminalzustand enthält oder der Agent nie einen solchen

erreicht, sind alle Umgebungsverläufe unendlich lang und die Nutzen mit additiven Gewinnen sind im Allgemeinen unendlich. Zwar können wir zustimmen, dass $+\infty$ besser ist als $-\infty$, doch ist es schwieriger, zwei Zustandsfolgen, die beide den Nutzen $+\infty$ haben, zu vergleichen. Es gibt drei Lösungen; zwei davon haben wir bereits gesehen:

1 Bei verminderten Gewinnen ist der Nutzen einer unendlichen Folge *endlich*. Wenn Gewinne durch R_{max} und $\gamma < 1$ begrenzt sind, haben wir

$$U_h([s_0, s_1, s_2, \ldots]) = \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq \sum_{t=0}^{\infty} \gamma^t R_{\max} = R_{\max} / (1-\gamma) \tag{17.1}$$

unter Verwendung der Standardformel für die Summe einer unendlichen geometrischen Folge.

2 Wenn die Umgebung Terminalzustände enthält *und der Agent irgendwann garantiert einen solchen erreicht*, müssen wir nie unendliche Folgen vergleichen. Eine Taktik, die garantiert einen Terminalzustand erreicht, wird als **richtige Taktik** bezeichnet. Mit richtigen Taktiken können wir $\gamma = 1$ verwenden (d.h. additive Gewinne). Die ersten drei in Abbildung 17.2(b) gezeigten Taktiken sind richtig, die vierte ist nicht richtig. Sie erzielt einen unendlichen Gesamtgewinn, weil sie sich von den Terminalzuständen fernhält, wenn der Gewinn für die Nichtterminalzustände positiv ist. Die Existenz nicht richtiger Taktiken kann dazu führen, dass Standardalgorithmen für MEPs mit additiven Gewinnen fehlschlagen, was ein guter Grund dafür ist, verminderte Gewinne zu verwenden.

3 Unendliche Folgen lassen sich auch im Hinblick auf den **durchschnittlichen** pro Zeitschritt erhaltenen **Gewinn** vergleichen. Angenommen, Quadrat (1, 1) in der 4×3-Welt hat einen Gewinn von 0,1, während die anderen Nichtterminalzustände den Gewinn von 0,01 haben. Dann hat eine Taktik, die ihr Bestes gibt, um in (1, 1) zu bleiben, einen höheren durchschnittlichen Gewinn als eine Taktik, die irgendwo anders bleibt. Ein durchschnittlicher Gewinn ist ein praktisches Kriterium für einige Probleme, aber die Analyse der Algorithmen für durchschnittlichen Gewinn kann in diesem Buch nicht erläutert werden.

Insgesamt verursacht die Verwendung verminderter Gewinne die wenigsten Schwierigkeiten bei der Auswertung von Zustandsfolgen.

17.1.2 Optimale Taktiken und die Nutzen von Zuständen

Nachdem entschieden ist, dass die Nutzen einer gegebenen Zustandsfolge die Summe der verminderten Gewinne sind, die während der Folge erhalten werden, können wir die Taktiken vergleichen, indem wir die *erwarteten* Nutzen vergleichen, die sich bei ihrer Ausführung ergeben. Wir nehmen an, dass sich der Agent in einem bestimmten Anfangszustand s befindet, und definieren S_t (eine Zufallsvariable) als den Zustand, den der Agent zur Zeit t erreicht, wenn er eine bestimmte Taktik π ausführt. (Offensichtlich ist $S_0 = s$ der Zustand, in dem sich der Agent momentan befindet.) Die Wahrscheinlichkeitsverteilung über den Zustandsfolgen S_1, S_2, ..., wird durch den Anfangszustand s, die Taktik π und das Übergangsmodell für die Umgebung bestimmt.

Der erwartete Nutzen bei Ausführen von π beginnend in s ist gegeben durch

$$U^{\pi}(s) = E\left[\sum_{j=0}^{\infty} \gamma^t R(S_t)\right], \tag{17.2}$$

wobei die Erwartung in Bezug auf die Wahrscheinlichkeitsverteilung über Zustandsfolgen durch s und π bestimmt wird. Von allen Taktiken, die der Agent nun zur Ausführung beginnend in s wählen kann, hat eine (oder mehrere) höhere erwartete Nutzen als alle anderen. Wir kennzeichnen eine dieser Taktiken mit π_s^*:

$$\pi_s^* = \underset{\pi}{\operatorname{argmax}} U^{\pi}(s). \tag{17.3}$$

Denken Sie daran, dass π_s^* eine Taktik ist, sodass sie eine Aktion für jeden Zustand empfiehlt. Insbesondere ergibt es sich aus ihrer Verbindung mit s, dass es sich um eine optimale Taktik handelt, wenn s der Ausgangszustand ist. Eine bemerkenswerte Konsequenz aus der Verwendung von verminderten Nutzen mit unendlichen Horizonten besteht darin, dass die optimale Taktik *unabhängig* vom Ausgangszustand ist. (Natürlich ist die *Aktionsfolge* nicht unabhängig; denn eine Taktik ist eine Funktion, die für jeden Zustand eine Aktion spezifiziert.) Diese Tatsache scheint intuitiv naheliegend zu sein: Wenn Taktik π_a^* optimal beim Beginn in a und Taktik π_b^* optimal beim Start in b sind und sie einen dritten Zustand c erreichen, gibt es keinen vernünftigen Grund, dass sie untereinander oder mit π_c^* nicht übereinkommen, was als Nächstes zu tun ist.[2] Somit können wir einfach π^* für eine optimale Taktik schreiben.

Mit dieser Definition ist der wahre Nutzen eines Zustandes einfach $U^{\pi^*}(s)$ – d.h. die erwartete Summe verminderter Gewinne, wenn der Agent eine optimale Taktik ausführt. Wir schreiben dies als $U(s)$ entsprechend der in *Kapitel 16* verwendeten Notation für den Nutzen eines Ergebnisses. Beachten Sie, dass $U(s)$ und $R(s)$ ganz unterschiedliche Quantitäten sind; $R(s)$ ist der „kurzfristige" Gewinn, sich in s zu befinden, während $U(s)$ der „langfristige" Gesamtgewinn ab s ist. ▶ Abbildung 17.3 zeigt die Nutzen für die 4×3-Welt. Die Nutzen für Zustände, die näher am Ausgang +1 liegen, sind höher, weil weniger Schritte erforderlich sind, um den Ausgang zu erreichen.

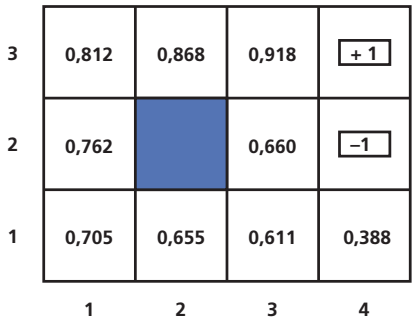


Abbildung 17.3: Die Nutzen der Zustände in der 4×3-Welt, berechnet mit $\gamma = 1$ und $R(s) = -0{,}04$ für Nichtterminalzustände.

2 Obwohl dies offensichtlich scheint, gilt es nicht für Strategien mit endlichem Horizont oder für andere Wege, Gewinne über die Zeit zu kombinieren. Der Beweis folgt direkt aus der Eindeutigkeit der Nutzenfunktion für Zustände, wie in *Abschnitt 17.2* gezeigt.

Die Nutzenfunktion $U(s)$ erlaubt es dem Agenten, Aktionen nach dem Prinzip des maximalen erwarteten Nutzens (siehe *Kapitel 16*) auszuwählen – d.h. die Aktion zu wählen, die den erwarteten Nutzen des nachfolgenden Zustandes maximiert:

$$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}} \sum_{s'} P(s' \mid s,a) U(s'). \tag{17.4}$$

Die beiden nächsten Abschnitte beschreiben Algorithmen, um optimale Taktiken zu finden.

17.2 Wert-Iteration

In diesem Abschnitt stellen wir mit der **Wert-Iteration** einen Algorithmus vor, der eine optimale Taktik berechnet. Das grundlegende Konzept dabei ist, den Nutzen jedes *Zustandes* zu berechnen und dann die Zustandsnutzen zu verwenden, um in jedem Zustand eine optimale Aktion auszuwählen.

17.2.1 Die Bellman-Gleichung für Nutzen

Tipp

Abschnitt 17.1.2 hat den Nutzen in einem Zustand definiert als die erwartete Summe von verminderten Gewinnen von diesem Punkt an. Daraus folgt, dass es eine direkte Beziehung zwischen dem Nutzen eines Zustandes und dem Nutzen seiner Nachbarn gibt: *Der Nutzen eines Zustandes ist der unmittelbare Gewinn für diesen Zustand plus dem erwarteten verminderten Gewinn des nächsten Zustandes, vorausgesetzt, der Agent wählt die optimale Aktion*. Das bedeutet, der Nutzen eines Zustandes ist gegeben durch

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s,a) U(s'). \tag{17.5}$$

Gleichung (17.5) wird als **Bellman-Gleichung** bezeichnet, nach Richard Bellman (1957). Die Nutzen der Zustände – durch Gleichung (17.2) als die erwarteten Nutzen nachfolgender Zustandsfolgen definiert – sind Lösungen der Menge der Bellman-Gleichungen. Letztlich handelt es sich dabei um die *eindeutigen* Lösungen, wie wir in *Abschnitt 17.2.3* zeigen.

Wir betrachten eine der Bellman-Gleichungen für die 4 × 3-Welt. Die Gleichung für den Zustand (1, 1) lautet:

$$\begin{aligned} U(1,1) = -0{,}04 + \gamma \max\{ \quad & 0{,}8U(1,2) + 0{,}1U(2,1) + 0{,}1U(1,1), && (\textit{Oben}) \\ & 0{,}9U(1,1) + 0{,}1U(1,2) && (\textit{Links}) \\ & 0{,}9U(1,1) + 0{,}1U(2,1) && (\textit{Unten}) \\ & 0{,}8U(2,1) + 0{,}1U(1,2) + 0{,}1U(1,1)\ \}. && (\textit{Rechts}) \end{aligned}$$

Wenn wir die Zahlen aus Abbildung 17.3 einfügen, stellen wir fest, dass *Oben* die beste Aktion ist.

17.2.2 Der Algorithmus für die Wert-Iteration

Die Bellman-Gleichung ist die Grundlage für den Algorithmus zur Wert-Iteration für die Lösung von MEPs. Wenn es n mögliche Zustände gibt, gibt es n Bellman-Gleichungen – eine für jeden Zustand. Die n Gleichungen enthalten n Unbekannte – die Nutzen der Zustände. Wir müssen also diese simultanen Gleichungen lösen, um die Nutzen zu ermitteln. Es gibt ein Problem: Die Gleichungen sind *nichtlinear*, weil der „max"-Operator kein linearer Operator ist. Während lineare Gleichungssysteme schnell unter Verwendung der Techniken aus der linearen Algebra gelöst werden können, sind nichtlineare Gleichungssysteme sehr viel problematischer. Man kann es jedoch mit einem *iterativen* Ansatz probieren. Wir beginnen mit zufälligen Ausgangswerten für die Nutzen, berechnen die rechte Seite der Gleichung und setzen sie auf der linken Seite ein – womit wir den Nutzen jedes Zustandes mit den Nutzen seiner Nachbarn aktualisieren. Dies wiederholen wir, bis wir ein Gleichgewicht erreicht haben. Sei $U_i(s)$ der Nutzenwert für den Zustand s bei der i-ten Iteration. Der Iterationsschritt, auch als **Bellman-Aktualisierung** bezeichnet, sieht dann wie folgt aus:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s,a) U_i(s'). \tag{17.6}$$

Hier wird angenommen, dass die Aktualisierung bei jeder Iteration auf alle Zustände gleichzeitig angewandt wird. Wenn wir die Bellman-Aktualisierung unendlich oft anwenden, erreichen wir garantiert ein Gleichgewicht (siehe *Abschnitt 17.2.3*), wofür die endgültigen Nutzenwerte Lösungen für die Bellman-Gleichungen sein müssen. Letztlich handelt es sich dabei auch um die *eindeutigen* Lösungen und die entsprechende Taktik (mithilfe von Gleichung (17.4) erhalten) ist optimal. Der als VALUE-ITERATION bezeichnete Algorithmus ist in ▶ Abbildung 17.4 gezeigt. Wir können die Wert-Iteration auf die 4×3-Welt aus Abbildung 17.1(a) anwenden. Beginnend mit dem Ausgangswert 0 entwickeln sich die Nutzen wie in ▶ Abbildung 17.5(a) gezeigt. Beachten Sie, wie die Zustände mit unterschiedlichen Distanzen von (4, 3) negativen Gewinn akkumulieren, bis irgendwann ein Pfad nach (4, 3) gefunden ist, woraufhin die Nutzen anfangen zu wachsen. Wir können uns den Algorithmus der Wert-Iteration auch als die *Informationsweitergabe* durch den Zustandsraum mithilfe lokaler Aktualisierungen vorstellen.

```
function VALUE-ITERATION(mdp, ϵ) returns eine Nutzenfunktion
   inputs: mdp, ein MEP mit Zuständen S, Aktionen A(s),
           Übergangsmodell P(s'|s, a), Gewinnen R(s), Minderung γ
      ϵ, der maximal erlaubte Fehler im Nutzen eines Zustands
   local variables: U, U', Nutzenvektoren für Zustände in S, anfänglich null
                    δ, die maximale Änderung im Nutzen eines Zustands bei
                    einer Iteration

   repeat
   U ← U'; δ ← 0
      for each Zustand s in S do
         U'[s] ← R[s] + γ + max_{a∈A(s)} Σ_{s'} P(s'|s,a)U[s']
         if |U'[s] - U[s]| > δ then δ ← |U'[s] - U[s]|
   until δ < ϵ(1 - γ)/γ
   return U
```

Abbildung 17.4: Der Algorithmus der Wert-Iteration für die Berechnung von Zustandsnutzen. Die Terminierungsbedingung stammt aus Gleichung (17.8).

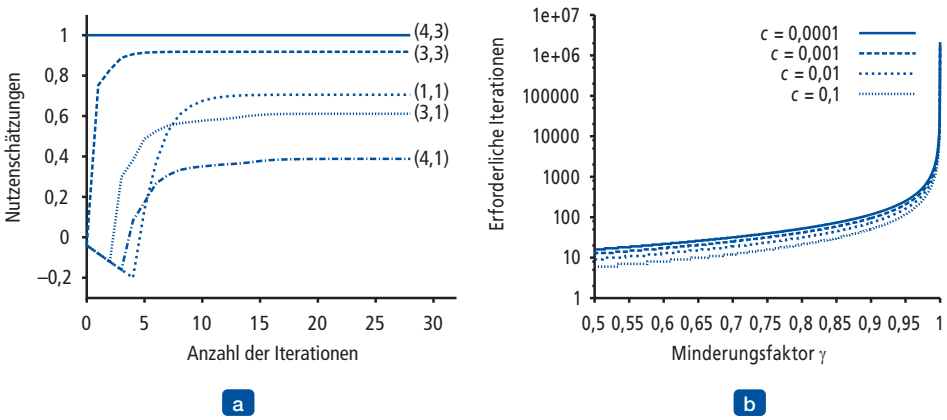


Abbildung 17.5: (a) Graph, der die Entwicklung der Nutzen ausgewählter Zustände unter Verwendung der Wert-Iteration zeigt. (b) Die Anzahl der Wert-Iterationen k, die erforderlich sind, um einen Fehler von höchstens $\epsilon = c \cdot R_{max}$ für unterschiedliche Werte von c als Funktion des Minderungsfaktors γ zu garantieren.

17.2.3 Konvergenz der Wert-Iteration

Wir haben gesagt, dass die Wert-Iteration irgendwann zu einer eindeutigen Lösungsmenge der Bellman-Gleichungen konvergiert. In diesem Abschnitt erklären wir, warum das passiert. Wir führen einige praktische mathematische Konzepte ein und stellen Methoden für die Abschätzung des Fehlers in der Nutzenfunktion vor, der zurückgegeben wird, wenn der Algorithmus früh beendet wird; das ist praktisch, weil es bedeutet, dass wir ihn nicht endlos ausführen müssen. Dieser Abschnitt ist recht technisch orientiert.

Das grundlegende Konzept für den Beweis, dass die Wert-Iteration konvergiert, ist die **Kontraktion**. Einfach ausgedrückt ist eine Kontraktion die Funktion eines Arguments, die bei aufeinanderfolgender Anwendung auf zwei unterschiedliche Eingaben zwei Ausgabewerte erzeugt, die „enger beisammen" liegen als die ursprünglichen Eingaben, und zwar mindestens um einen konstanten Betrag. Die Funktion „Division durch 2" beispielsweise ist eine Kontraktion, denn wenn wir zwei Zahlen durch 2 dividieren, wird ihre Differenz halbiert. Beachten Sie, dass die Funktion „Division durch 2" einen Fixpunkt hat, nämlich 0, der sich auch durch Anwendung der Funktion nicht ändert. Aus diesem Beispiel können wir zwei wichtige Eigenschaften von Kontraktionen ableiten:

- Eine Kontraktion hat nur einen Fixpunkt; gäbe es zwei Fixpunkte, würden sie sich bei Anwendung der Funktion nicht annähern und es handelte sich nicht um eine Kontraktion.
- Wenn die Funktion auf ein Argument angewendet wird, muss sie dem Fixpunkt näher kommen (weil sich der Fixpunkt nicht bewegt); eine wiederholte Anwendung einer Kontraktion erreicht also letztendlich immer den Fixpunkt.

Angenommen, wir betrachten die Bellman-Aktualisierung (Gleichung (17.6)) als Operator B, der gleichzeitig angewendet wird, um den Nutzen jedes Zustandes zu aktualisieren. Sei U_i der Nutzenvektor für alle Zustände in der i-ten Iteration. Die Bellman-Aktualisierungsgleichung kann dann geschrieben werden als:

$$U_{i+1} \leftarrow BU_i.$$

Anschließend brauchen wir eine Möglichkeit, Distanzen zwischen Nutzenvektoren zu messen. Wir verwenden die **Max-Norm**, die die „Länge“ eines Vektors als absoluten Wert seiner größten Komponente misst:

$$\|U\| = \max_s |U(s)|.$$

Tipp

Mit dieser Definition ist die „Distanz“ zwischen zwei Vektoren, $||U - U'||$, gleich der maximalen Differenz zwischen zwei einander entsprechenden Elementen. Das wichtigste Ergebnis dieses Abschnittes ist: *Seien U_i und U_i' zwei Nutzenvektoren. Dann gilt:*

$$\|BU_i - BU_i'\| \leq \gamma \|U_i - U_i'\| . \tag{17.7}$$

Das bedeutet, die Bellman-Aktualisierung ist eine Kontraktion um einen Faktor γ im Raum der Nutzenvektoren. (Übung 17.6 gibt eine Anleitung, wie sich diese Behauptung beweisen lässt.) Somit folgt aus den Eigenschaften der Kontraktion im Allgemeinen, dass die Wert-Iteration immer auf eine eindeutige Lösung der Bellman-Gleichungen konvergiert, wenn $\gamma < 1$ ist.

Wir können auch die Kontraktionseigenschaft verwenden, um die *Geschwindigkeit* der Konvergenz zu einer Lösung zu analysieren. Insbesondere können wir U_i' in Gleichung (17.7) durch die *tatsächlichen* Nutzen U ersetzen, wofür $BU = U$ gilt. Damit erhalten wir die Ungleichung

$$\|BU_i - U\| \leq \gamma \|U_i - U\|.$$

Wenn wir $\|U_i - U\|$ als den *Fehler* der Schätzung U_i betrachten, erkennen wir, dass der Fehler bei jeder Iteration um einen Faktor von mindestens γ reduziert wird. Das bedeutet, die Wert-Iteration konvergiert exponentiell schnell. Wir können die Anzahl der Iterationen berechnen, die erforderlich sind, um eine vorgegebene Fehlerschwelle ϵ zu erreichen: Aus Gleichung (17.1) wissen wir, dass die Nutzen aller Zustände auf $\pm R_{max}/(1 - \gamma)$ begrenzt sind. Das bedeutet, dass der maximale Anfangsfehler $\|U_0 - U\| \leq 2R_{max} / (1-\gamma)$ ist. Angenommen, wir führen N Iterationen aus, um einen Fehler von höchstens ϵ zu erhalten. Weil der Fehler jedes Mal um höchstens γ reduziert wird, brauchen wir $\gamma^N \cdot 2R_{max}/(1 - \gamma) \leq \epsilon$. Durch Logarithmieren ergibt sich, dass

$$N = \lceil \log(2R_{\max} / \epsilon(1-\gamma)) / \log(1/\gamma) \rceil$$

Iterationen genügen. ▶ Abbildung 17.5(b) zeigt, wie N mit γ für verschiedene Werte des Verhältnisses ϵ/R_{max} variiert. Die gute Nachricht ist, dass N aufgrund der exponentiellen Konvergenz nicht stark vom Verhältnis ϵ/R_{max} abhängig ist. Die schlechte Nachricht ist, dass N rapide wächst, wenn sich γ dem Wert 1 annähert. Wir können eine schnelle Konvergenz erreichen, indem wir γ klein machen, aber damit erhält der Agent letztlich einen kurzfristigen Horizont und könnte langfristige Auswirkungen der Aktionen übersehen.

Die Fehlerschwelle im obigen Absatz vermittelt einen Eindruck, welche Faktoren die Laufzeit des Algorithmus beeinflussen, wird aber manchmal zu konservativ als Methode verwendet, um zu erkennen, wann die Iteration beendet werden soll. Für diesen Zweck können wir eine Schwelle verwenden, die den Fehler in eine Relation zur Größe der Bellman-Aktualisierung für jede mögliche Iteration bringt. Aus der Kontraktionseigenschaft (Gleichung (17.7)) kann gezeigt werden, dass bei einer kleinen Aktualisierung

(wenn sich also der Nutzen keines Zustandes wesentlich ändert) der Fehler im Vergleich zur tatsächlichen Nutzenfunktion ebenfalls klein ist. Genauer gesagt:

$$\text{wenn } \|U_{i+1} - U_i\| < \epsilon(1-\gamma)/\gamma, \text{ dann } \|U_{i+1} - U\| < \epsilon. \tag{17.8}$$

Dies ist die im Algorithmus VALUE-ITERATION in Abbildung 17.4 verwendete Terminierungsbedingung.

Tipp

Bisher haben wir den Fehler in der Nutzenfunktion analysiert, der durch den Wert-Iterations-Algorithmus zurückgegeben wird. *Was den Agenten jedoch wirklich interessiert, ist, wie gut er funktioniert, wenn er seine Entscheidungen auf Grundlage dieser Nutzenfunktion trifft.* Angenommen, nach i Iterationen der Wert-Iteration hat der Agent eine Schätzung U_i des tatsächlichen Nutzens U und erhält die MEN-Taktik π_i basierend auf einer Vorausschau von einem einzigen Schritt unter Verwendung von U_i (wie in Gleichung (17.4)). Ist das resultierende Verhalten annähernd so gut wie das optimale Verhalten? Das ist eine wichtige Frage für jeden Agenten und es stellt sich heraus, dass die Antwort positiv ist. $U^{\pi_s}(s)$ ist der Nutzen, den man erhält, wenn π_i beginnend in s ausgeführt wird, und der **Taktikverlust** $\|U^{\pi_i} - U\|$ ist der höchste Betrag, den der Agent verlieren kann, wenn er statt der optimalen Taktik π^* die Taktik π_i ausführt. Der Taktikverlust von π_i ist durch die folgende Ungleichung mit dem Fehler in U_i verknüpft:

$$\text{wenn } \|U_i - U\| < \epsilon, \text{ dann } \|U^{\pi_i} - U\| < 2\epsilon\gamma/(1-\gamma). \tag{17.9}$$

In der Praxis kommt es häufig vor, dass π_i optimal lang wird, bevor U_i konvergiert hat. ▶ Abbildung 17.6 zeigt, wie der maximale Fehler in U_i und der Taktikverlust gegen null gehen, wenn der Wert-Iterationsprozess für die 4×3-Umgebung mit $\gamma = 0{,}9$ fortgesetzt wird. Die Taktik π_i ist optimal, wenn $i = 4$, selbst wenn der maximale Fehler in U_i immer noch 0,46 ist.

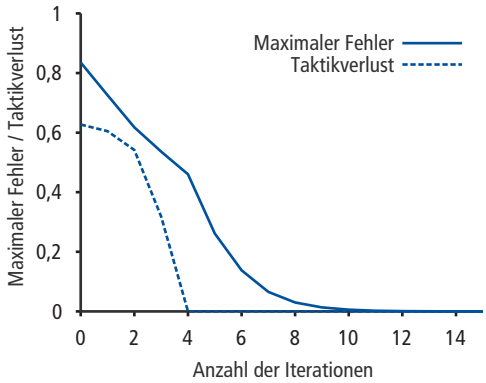


Abbildung 17.6: Der maximale Fehler $\|U_i - U\|$ der Nutzenschätzung und der Taktikverlust $\|U^{\pi_i} - U\|$ im Vergleich zur optimalen Taktik als Funktion der Anzahl der Iterationen der Wert-Iteration.

Jetzt haben wir alles, was wir brauchen, um die Wert-Iteration in der Praxis einzusetzen. Wir wissen, dass sie zu den korrekten Nutzen konvergiert. Wir können den Fehler in der Nutzenschätzung begrenzen, wenn wir nach einer endlichen Anzahl an Iterationen aufhören, und wir können den Taktikverlust begrenzen, der aus der Ausfüh-

rung der entsprechenden MEN-Taktik resultiert. Ein letzter Hinweis: Alle Ergebnisse aus diesem Abschnitt sind von einer Minderung mit $\gamma < 1$ abhängig. Wenn $\gamma = 1$ gilt und die Umgebung Terminalzustände enthält, resultiert daraus eine ähnliche Konvergenzmenge und es können Fehlergrenzen abgeleitet werden, wenn bestimmte technische Bedingungen erfüllt sind.

17.3 Taktik-Iteration

Im vorigen Abschnitt haben wir gesehen, dass es auch dann möglich ist, eine optimale Taktik zu erhalten, wenn die Schätzung der Nutzenfunktion ungenau ist. Wenn eine Aktion deutlich besser als alle anderen ist, muss die genaue Größe der Nutzen der beteiligten Zustände nicht genau sein. Diese Einsicht legt eine alternative Vorgehensweise zur Ermittlung optimaler Taktiken nahe. Der Algorithmus der **Taktik-Iteration** wechselt zwischen den beiden folgenden Schritten, beginnend mit einer Ausgangstaktik π_0:

- **Taktikauswertung:** Für eine gegebene Taktik π_i wird $U_i = U^{\pi_i}$, der Nutzen jedes Zustandes, wenn π_i ausgeführt wird, berechnet.
- **Taktikverbesserung:** Es wird eine neue MEN-Taktik π_{i+1} berechnet, die basierend auf U_i immer einen Schritt vorausschaut (wie in Gleichung 17.4 gezeigt).

Der Algorithmus terminiert, wenn der Schritt der Taktikverbesserung keine Änderung im Nutzen mehr erbringt. Damit wissen wir, dass die Nutzenfunktion U_i ein Fixpunkt der Bellman-Aktualisierung ist, sodass es sich um eine Lösung der Bellman-Gleichungen handelt, und π_i muss eine optimale Taktik sein. Weil es nur endlich viele Taktiken für einen endlichen Zustandsraum gibt und gezeigt werden kann, dass jede Iteration eine bessere Taktik erbringt, muss die Taktik-Iteration terminieren. Den zugehörigen Algorithmus sehen Sie in ▶ Abbildung 17.7.

```
function POLICY-ITERATION(mdp) returns eine Taktik
   inputs: mdp, ein MEP mit Zuständen S, Aktionen A(s),
           Übergangsmodell P(s'|s,a)
   local variables: U, ein Nutzenvektor für Zustände in S,
                    anfänglich null
                    π, ein Taktikvektor, indiziert durch den Zustand,
                    anfänglich zufällig

   repeat
      U ← POLICY-EVALUATION(π, U, mdp)
      unverändert? ← true
      for each Zustand s in S do
         if max_{a∈A(s)} Σ_{s'} P(s'|s,a) U[s'] > Σ_{s'} P(s'|s,π[s]) U[s'] then do
            π[s] ← argmax_{a∈A(s)} Σ_{s'} P(s'|s,a) U[s']
            unverändert? ← false
   until unverändert?
   return π
```

Abbildung 17.7: Der Algorithmus der Taktik-Iteration für die Berechnung einer optimalen Taktik.

Der Schritt der Taktikverbesserung ist offensichtlich ganz einfach, aber wie sollen wir die Routine POLICY-EVALUATION implementieren? Es stellt sich heraus, dass dies sehr viel einfacher als die Lösung der Bellman-Standardgleichungen ist (was bei der Wert-Iteration passiert), weil die Aktion in jedem Zustand durch die Taktik festgelegt ist. Bei der i-ten Iteration spezifiziert die Taktik π_i die Aktion $\pi_i(s)$ im Zustand s. Das bedeutet, wir haben eine vereinfachte Version der Bellman-Gleichung (17.5) im Hinblick auf den Nutzen von s (unter π_i) für die Nutzen seiner benachbarten Zustände:

$$U_i(s) = R(s) + \gamma \sum_{s'} (s' \mid s, \pi_i(s))\, U_i(s'). \tag{17.10}$$

Nehmen wir beispielsweise an, π_i ist die in Abbildung 17.2(a) gezeigte Taktik. Dann haben wir $\pi_i(1,1) = \mathit{Auf}$, $\pi_i(1,2) = \mathit{Auf}$ usw. und die vereinfachten Bellman-Gleichungen lauten:

$$U_i(1,1) = 0{,}8U_i(1,2) + 0{,}1U_i(1,1) + 0{,}1U_i(2,1),$$
$$U_i(1,2) = 0{,}8U_i(1,3) + 0{,}2U_i(1,2),$$
$$\vdots$$

Wichtig dabei ist, dass diese Gleichungen *linear* sind, weil der „max“-Operator entfernt wurde. Für n Zustände haben wir n lineare Gleichungen mit n Unbekannten, die mithilfe von Standardmethoden der linearen Algebra in der Zeit $O(n^3)$ genau gelöst werden können.

Für kleine Zustandsräume sind Taktikauswertungen unter Verwendung exakter Lösungsmethoden häufig der effizienteste Ansatz. Für große Zustandsräume könnte die Zeit $O(n^3)$ bedeuten, dass die Ausführung unmöglich ist. Glücklicherweise ist es nicht erforderlich, eine *exakte* Taktikauswertung vorzunehmen. Stattdessen können wir eine bestimmte Anzahl vereinfachter Wert-Iterationsschritte ausführen (vereinfacht, weil die Taktik feststeht), um eine ausreichend gute Annäherung für die Nutzen zu erhalten. Die vereinfachte Bellman-Aktualisierung für diesen Prozess lautet:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s))\, U_i(s').$$

Dies wird k-mal wiederholt, um die nächste Nutzenabschätzung zu erzeugen. Der resultierende Algorithmus wird auch als **modifizierte Taktik-Iteration** bezeichnet. Häufig ist er effizienter als die Standard-Taktik-Iteration oder die Wert-Iteration.

Bei den bisher beschriebenen Algorithmen musste der Nutzen oder die Taktik für alle Zustände gleichzeitig aktualisiert werden. Es zeigt sich aber, dass das nicht unbedingt nötig ist. Tatsächlich können wir für jede Iteration eine *beliebige Untermenge* von Zuständen auswählen und eine *beliebige* Aktualisierung auf diese Untermenge anwenden (Taktikverbesserung oder vereinfachte Wert-Iteration). Dieser sehr allgemeine Algorithmus wird als **asynchrone Taktik-Iteration** bezeichnet. Für bestimmte Bedingungen zur Anfangstaktik und Nutzenfunktion konvergiert die asynchrone Taktik-Iteration garantiert. Die Freiheit, mit beliebigen Zuständen zu arbeiten, bedeutet, dass wir viel effizientere Heuristik-Algorithmen entwickeln können – z.B. Algorithmen, die sich auf die Aktualisierung solcher Zustandswerte konzentrieren, die von einer guten Taktik wahrscheinlich erzielt werden. Das ist im realen Leben sehr sinnvoll: Wenn man nicht vorhat, sich von einer Klippe ins Meer zu stürzen, braucht man sich auch keine Gedanken über den genauen Wert der resultierenden Zustände zu machen.

17.4 Partiell beobachtbare MEPs

Bei der Beschreibung der Markov-Entscheidungsprozesse in *Abschnitt 17.1* sind wir davon ausgegangen, dass die Umgebung **vollständig beobachtbar** war. Bei dieser Voraussetzung weiß der Agent immer, in welchem Zustand er sich befindet. Das bedeutet in Kombination mit der Markov-Annahme für das Übergangsmodell, dass die optimale Taktik nur vom aktuellen Zustand abhängig ist. Wenn die Umgebung nur **partiell beobachtbar** ist, kann man davon ausgehen, dass die Situation sehr viel unklarer ist. Der Agent weiß nicht unbedingt, in welchem Zustand er sich befindet, deshalb kann er auch nicht die für diesen Zustand empfohlene Aktion $\pi(s)$ ausführen. Darüber hinaus sind der Nutzen eines Zustandes s und die optimale Aktion in s nicht nur von s abhängig, sondern auch davon, *wie viel der Agent weiß*, wenn er sich in s befindet. Aus diesen Gründen werden **partiell beobachtbare MEPs** normalerweise als schwieriger erachtet als normale MEPs. Wir können jedoch die partiell beobachtbaren MEPs nicht vermeiden, weil die Welt selbst ein solcher Prozess ist.

17.4.1 Definition von partiell beobachtbaren MEPs

Um die partiell beobachtbaren MEPs verarbeiten zu können, müssen wir sie zuvor korrekt definieren. Ein partiell beobachtbarer MEP hat die gleichen Elemente wie ein MEP – das Übergangsmodell $P(S'|s \mid a)$, die Aktionen $A(s)$ und die Gewinnfunktion $R(s)$ –, aber wie die in *Abschnitt 4.4* beschriebenen partiell beobachtbaren Suchprobleme besitzt er auch ein **Sensormodell** $P(e \mid s)$. Hier wie in *Kapitel 15* spezifiziert das Sensormodell die Wahrscheinlichkeit, die Evidenz e im Zustand s wahrzunehmen.[3] Zum Beispiel können wir die 4×3-Welt von Abbildung 17.1 in ein partiell beobachtbares MEP konvertieren, indem wir einen gestörten oder partiellen Sensor hinzufügen, anstatt anzunehmen, dass der Agent seine Position genau kennt. Ein derartiger Sensor könnte die *Anzahl der benachbarten Wände* messen, die in allen Nichtterminalfeldern 2 ist, außer in den Feldern der dritten Spalte, wo der Wert 1 ist; eine gestörte Version könnte den falschen Wert mit der Wahrscheinlichkeit 0,1 angeben.

In den *Kapiteln 4* und *11* haben wir nichtdeterministische und partiell beobachtbare Planungsprobleme betrachtet und erkannt, dass der **Belief State** – die Menge der tatsächlichen Zustände, in denen sich der Agent befinden kann – ein Schlüsselkonzept für die Beschreibung und Berechnung von Lösungen ist. In partiell beobachtbaren MEPs ist ein Belief State b jetzt eine *Wahrscheinlichkeitsverteilung* über alle möglichen Zustände, wie in Kapitel 15 beschrieben. Zum Beispiel könnte der anfängliche Belief State für den partiell beobachtbaren MEP der Größe 4 x 3 die Gleichverteilung über die neun Nichtterminalzustände sein, d.h.

$$\left\langle \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, \frac{1}{9}, 0, 0 \right\rangle .$$

Wir schreiben $b(s)$ für die Wahrscheinlichkeit, die dem tatsächlichen Zustand s durch den Belief State b zugeordnet wird. Der Agent kann seinen aktuellen Belief State als die bedingte Wahrscheinlichkeitsverteilung über die tatsächlichen Zustände für die bisherige Folge an Beobachtungen und Aktionen berechnen. Das ist im Wesentlichen eine **Filter-**

3 Wie die Gewinnfunktion von MEPs kann auch das Sensormodell von der Aktion und dem Ergebniszustand abhängig sein, aber auch diese Änderung ist nicht von größerer Bedeutung.

Aufgabe (siehe *Kapitel 15*). Die grundlegende rekursive Filtergleichung (15.5 in *Abschnitt 15.2.1*) zeigt, wie der neue Belief State aus dem vorhergehenden Belief State und der neuen Evidenz berechnet wird. Für partiell beobachtbare MEPs müssen wir auch eine Aktion berücksichtigen und verwenden eine etwas andere Notation, aber das Ergebnis ist im Wesentlichen dasselbe. Wenn $b(s)$ der vorherige Belief State war und der Agent die Aktion a ausführt und die Evidenz e wahrnimmt, ist der neue Belief State gegeben durch:

$$b'(s') = \alpha P(e \mid s') \sum_{s} P(s' \mid s, a) b(s).$$

Dabei ist α die Normalisierungskonstante, die die Belief-State-Summe zu 1 macht. Analog zum Aktualisierungsoperator für das Filtern (*Abschnitt 15.2.1*) können wir dies schreiben als

$$b' = \text{FORWARD}(b, a, e). \tag{17.11}$$

Nehmen Sie im partiell beobachtbaren MEP der 4×3-Welt an, dass der Agent die Aktion *Links* ausführt und sein Sensor 1 benachbarte Wand meldet; dann ist es ziemlich wahrscheinlich (wenn auch nicht garantiert, da sowohl die Bewegung als auch der Sensor gestört sind), dass sich der Agent nun in (3, 1) befindet. In Übung 17.13 haben Sie die Aufgabe, die genauen Wahrscheinlichkeitswerte für den neuen Belief State zu berechnen.

Tipp

Die grundlegende Erkenntnis, die wir brauchen, um partiell beobachtbare MEPs zu verstehen, ist: *Die optimale Aktion ist nur von dem aktuellen Belief State des Agenten abhängig.* Das bedeutet, die optimale Taktik kann durch eine Abbildung von $\pi^*(b)$ von Belief States auf Aktionen beschrieben werden. Sie ist *nicht* von dem *aktuellen* Zustand abhängig, in dem sich der Agent befindet. Das ist praktisch, weil der Agent seinen aktuellen Zustand nicht kennt; er kennt nur seinen Belief State. Der Entscheidungszyklus eines Agenten mit partiell beobachtbarem MEP lässt sich also in die folgenden drei Schritte aufgliedern:

1. Für den aktuellen Belief State b wird die Aktion $a = \pi^*(b)$ ausgeführt.
2. Die Wahrnehmung e wird empfangen.
3. Der aktuelle Belief State wird auf FORWARD(b, a, e) gesetzt und das Ganze wiederholt sich.

Wir können für die partiell beobachtbaren MEPs also davon ausgehen, dass sie eine Suche im Belief-State-Raum bedingen, so wie die Methoden für die sensorlosen und Kontingenzprobleme in *Kapitel 4*. Der wichtigste Unterschied ist, dass der Belief-State-Raum von partiell beobachtbaren MEPs *stetig* ist, weil ein Belief State eines partiell beobachtbaren MEP eine Wahrscheinlichkeitsverteilung ist. Beispielsweise ist ein Belief State für die 4×3-Welt ein Punkt in einem elfdimensionalen stetigen Raum. Eine Aktion ändert den Belief State, nicht nur den physischen Zustand. Deshalb wird die Aktion zumindest zum Teil entsprechend der Informationen, die der Agent als Ergebnis erhält, ausgewertet. Partiell beobachtbare MEPs können also den Wert der Information (*Abschnitt 16.6*) als Komponente des Entscheidungsproblems beinhalten.

Betrachten wir das Ergebnis von Aktionen sorgfältiger. Insbesondere wollen wir die Wahrscheinlichkeit berechnen, dass ein Agent im Belief State b nach Ausführen der Aktion a den Belief State b' erreicht. Würden wir die Aktion *und die nachfolgende Wahrnehmung* kennen, ergäbe Gleichung (17.11) eine *deterministische* Aktualisierung des Belief State: $b' = \text{FORWARD}(b, a, e)$. Natürlich ist die nachfolgende Beobachtung

noch nicht bekannt, deshalb gelangt der Agent abhängig von der empfangenen Wahrnehmung in einen der möglichen Belief States b'. Die Wahrscheinlichkeit der Wahrnehmung e, vorausgesetzt, a wurde im Belief State b beginnend ausgeführt, ist die Summierung über alle Zustände s', die der Agent erreichen könnte:

$$\begin{aligned} P(e \mid a,b) &= \sum_{s'} P(e \mid a,s',b)P(s' \mid a,b) \\ &= \sum_{s'} P(e \mid s')P(s' \mid a,b) \\ &= \sum_{s'} P(e \mid s') \sum_{s} P(s' \mid s,a)b(s). \end{aligned}$$

Wir können die Wahrscheinlichkeit, b' über die Aktion a aus b zu erreichen, als $\tau(b, a, b')$ schreiben. Damit erhalten wir:

$$\begin{aligned} P(b' \mid b,a) = P(b' \mid a,b) &= \sum_{e} P(b' \mid e,a,b)P(e \mid a,b) \\ &= \sum_{e} P(b' \mid e,a,b) \sum_{s'} P(e \mid s') \sum_{s} P(s' \mid s,a)b(s). \end{aligned} \tag{17.12}$$

Dabei ist $P(b' \mid e, a, b)$ gleich 1, wenn $b' = \text{FORWARD}(b, a, o)$, andernfalls 0.

Gleichung (17.12) kann als Definition eines Übergangsmodells für den Belief-State-Raum betrachtet werden. Außerdem können wir eine Gewinnfunktion für Belief States definieren (d.h. den erwarteten Gewinn für die tatsächlichen Zustände, in denen sich der Agent befinden kann):

$$\rho(b) = \sum_{s} b(s)R(s).$$

Tipp

Zusammen definieren $P(b' \mid b, a)$ und $\rho(b)$ ein *beobachtbares* MEP für den Raum der Belief States. Darüber hinaus kann gezeigt werden, dass eine optimale Taktik für dieses MEP, $\pi^*(b)$, auch eine optimale Taktik für das ursprüngliche partiell beobachtbare MEP ist. Mit anderen Worten, *die Lösung eines partiell beobachtbaren MEP für einen physischen Zustandsraum kann auf die Lösung eines MEP für den zugehörigen Belief-State-Raum reduziert werden.* Diese Tatsache ist vielleicht weniger überraschend, wenn man daran denkt, dass der Belief State per Definition für den Agenten immer beobachtbar ist.

Beachten Sie, dass wir zwar partiell beobachtbare MEPs auf MEPs reduziert haben, aber der resultierende MEP einen stetigen (und in der Regel hochdimensionalen) Zustandsraum hat. Keiner der in den *Abschnitten 17.2* und *17.3* beschriebenen MEP-Algorithmen kann direkt auf solche MEPs angewendet werden. Die nächsten beiden Unterabschnitte beschreiben einen Wert-Iterationsalgorithmus, der speziell für partiell beobachtbare MEPs konzipiert wurde, und einen Online-Entscheidungsfindungsalgorithmus, der dem in *Kapitel 5* entwickelten für Spiele ähnelt.

17.4.2 Wert-Iteration für partiell beobachtbare MEPs

Abschnitt 17.2 hat einen Algorithmus für die Wert-Iteration beschrieben, der für jeden Zustand genau einen Nutzen berechnet hat. Bei unendlich vielen Belief States müssen wir kreativer sein. Betrachten Sie eine optimale Taktik π^* und ihre Anwendung in einem bestimmten Belief State b: Die Taktik generiert eine Aktion und dann wird für

jede darauffolgende Wahrnehmung der Belief State aktualisiert und eine neue Aktion generiert usw. Für diesen spezifischen Zustand b ist die Taktik demzufolge einem **bedingten Plan**, wie er in *Kapitel 4* für nichtdeterministische und partiell beobachtbare Probleme definiert wurde, exakt gleichwertig. Statt über Taktiken wollen wir über bedingte Pläne nachdenken und wie der erwartete Nutzen der Ausführung eines feststehenden bedingten Planes mit dem anfänglichen Belief State variieren kann. Wir machen zwei Beobachtungen:

1 Der Nutzen bei der Ausführung eines *festen* Planes p beginnend im physischen Zustand s sei $\alpha_p(s)$. Der erwartete Nutzen der Ausführung von p in Belief State b ist dann einfach $\Sigma_s b(s)\alpha_p(s)$ oder $b \cdot \alpha_p$, wenn wir beide als Vektoren auffassen. Folglich variiert der erwartete Nutzen eines festen bedingten Planes *linear* mit b; d.h., er korrespondiert mit einer Hyperebene im Belief State.

2 In jedem gegebenen Belief State b wählt die optimale Taktik den bedingten Plan mit dem höchsten erwarteten Nutzen zur Ausführung aus; und der erwartete Nutzen von b unter der optimalen Taktik ist einfach der Nutzen dieses bedingten Planes:

$$U(b) = U^{\pi^*}(b) = \max_p b\alpha_p.$$

Wählt die optimale Taktik π^* die Ausführung von p beginnend bei b aus, ist es vernünftig zu erwarten, dass sie die Ausführung von p in Belief States wählt, die sehr nahe bei b liegen. Wenn wir die Tiefe der bedingten Pläne begrenzen, gibt es nur noch endlich viele derartige Pläne und der stetige Raum von Belief States ist normalerweise in *Bereiche* unterteilt, die einem bestimmten bedingten Plan entsprechen, der in diesem Bereich optimal ist.

Aus diesen beiden Beobachtungen ist zu erkennen, dass die Nutzenfunktion $U(b)$ von Belief States, die das Maximum einer Sammlung von Hyperebenen darstellt, *stückweise linear* und *konvex* ist.

Um diesen Sachverhalt zu veranschaulichen, verwenden wir eine einfache Zweizustandswelt. Die Zustände sind mit 0 und 1 bezeichnet, wobei $R(0) = 0$ und $R(1) = 1$ gilt. Es gibt zwei Aktionen: Bei *Bleiben* rührt sich der Agent mit einer Wahrscheinlichkeit von 0,9 nicht vom Fleck und bei *Gehen* wechselt er mit der Wahrscheinlichkeit von 0,9 in den anderen Zustand. Fürs Erste nehmen wir den Minderungsfaktor mit $\gamma = 1$ an. Der Sensor meldet den richtigen Zustand mit einer Wahrscheinlichkeit von 0,6. Offenbar sollte der Agent *Bleiben*, wenn er glaubt, dass er sich in Zustand 1 befindet, und *Gehen*, wenn er von Zustand 0 ausgeht.

Eine Zweizustandswelt bietet den Vorteil, dass sich der Belief State als eindimensional betrachten lässt, da die Summe der beiden Wahrscheinlichkeiten 1 ergeben muss. In ▶ Abbildung 17.8(a) stellt die x-Achse den Belief State dar – definiert durch $b(1)$, die Wahrscheinlichkeit, im Zustand 1 zu sein. Sehen wir uns nun die Einschrittpläne [*Bleiben*] und [*Gehen*] an, die jeweils einen Gewinn für den aktuellen Zustand gefolgt von dem (verminderten) Gewinn für den nach der Aktion erreichten Zustand erhalten:

$$\langle_{[Bleiben]}(0) = R(0) + \gamma(0{,}9R(0) + 0{,}1R(1)) = 0{,}1$$
$$\langle_{[Bleiben]}(1) = R(1) + \gamma(0{,}9R(1) + 0{,}1R(0)) = 1{,}9$$
$$\langle_{[Gehen]}(0) = R(0) + \gamma(0{,}9R(1) + 0{,}1R(0)) = 0{,}9$$
$$\langle_{[Gehen]}(1) = R(1) + \gamma(0{,}9R(0) + 0{,}1R(1)) = 1{,}1$$

Abbildung 17.8(a) zeigt die Hyperebenen (in diesem Fall Linien) für $b \cdot \alpha_{[Bleiben]}$ und $b \cdot \alpha_{[Gehen]}$, wobei ihr Maximum fett gezeichnet ist. Die fette Linie stellt demnach die Nutzenfunktion für das Problem mit endlichem Horizont dar, der lediglich eine Aktion erlaubt, und in jedem „Stück" der stückweisen linearen Nutzenfunktion ist die optimale Aktion die erste Aktion des korrespondierenden bedingten Planes. In diesem Fall ist die optimale Einschritttaktik *Bleiben*, wenn $b(1) > 0{,}5$, andernfalls *Gehen*.

Nachdem wir die Nutzen $\alpha_p(s)$ für alle bedingten Pläne p der Tiefe 1 in jedem physischen Zustand s haben, können wir die Nutzen für bedingte Pläne der Tiefe 2 berechnen, indem wir jede mögliche erste Aktion, jede mögliche darauffolgende Wahrnehmung und dann jeden Weg für die Auswahl eines Planes mit der Tiefe 1 zur Ausführung für jede Wahrnehmung betrachten:

[*Bleiben*; `if` *Wahrnehmung* = 0 `then` *Bleiben* `else` *Bleiben*]
[*Bleiben*; `if` *Wahrnehmung* = 0 `then` *Bleiben* `else` *Gehen*] . . .

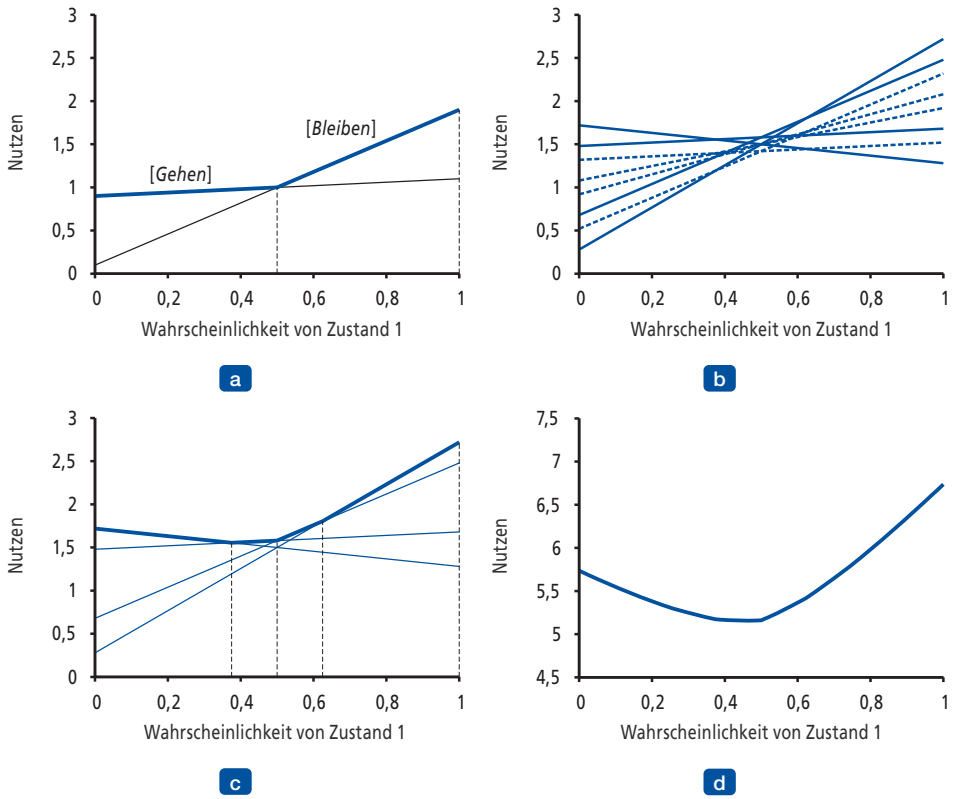


Abbildung 17.8: (a) Nutzen von zwei Einschrittplänen als Funktion des anfänglichen Belief State $b(1)$ für die Zweizustandswelt, wobei die entsprechende Nutzenfunktion fett gezeichnet ist. (b) Nutzen für acht verschiedene Zweischrittpläne. (c) Nutzen für vier nicht dominierte Zweischrittpläne. (d) Nutzen für optimale Achtschrittpläne.

Es gibt insgesamt acht verschiedene Pläne der Tiefe 2; ihre Nutzen sind in ▶ Abbildung 17.8(b) dargestellt. Beachten Sie, dass vier der Pläne (gestrichelte Linien) suboptimal über den gesamten Glaubensraum sind – diese sogenannten **dominierten** Pläne müssen nicht weiter betrachtet werden. ▶ Abbildung 17.8(c) zeigt die vier nicht dominierten

Pläne, von denen jeder in einem bestimmten Bereich optimal ist. Die Bereiche partitionieren den Belief-State-Raum.

Wir wiederholen den Vorgang für Tiefe 3 usw. Wenn allgemein p ein bedingter Plan der Tiefe d mit der anfänglichen Aktion a und dem Subplan $p.e$ der Tiefe $d - 1$ für Wahrnehmung e ist, gilt:

$$\alpha_p(s) = R(s) + \gamma\left(\sum_{s'} P(s' \mid s,a) \sum_e P(e \mid s')\alpha_{p.e}(s')\right). \tag{17.3}$$

Diese Rekursion liefert uns auf natürlichem Weg einen Algorithmus für die Wert-Iteration, der in ▶ Abbildung 17.9 schematisch angegeben ist.

```
function POMDP-VALUE-ITERATION(pomdp, ∈) returns eine Nutzenfunktion
  inputs: pomdp, ein partiell beobachtbarer MEP mit Zuständen S, Aktionen A(s),
          Übergangsmodell P(s'|s, a), Sensormodell P(e | s),
          Gewinnen R(s), Minderung γ
          ∈, der maximale Fehler, der im Nutzen eines Zustands erlaubt ist
  local variables: U, U', Mengen von Plänen p mit zugeordneten
                   Nutzenvektoren α_p

  U' ← eine Menge, die lediglich den leeren Plan [] enthält, mit a_[](s) = R(s)
  repeat
    U ← U'
    U' ← die Menge aller Pläne bestehend aus einer Aktion und -
          für jede mögliche nächste Wahrnehmung - einem Plan in U mit
          Nutzenvektoren, die gemäß Gleichung (17.13) berechnet wurden
    U' ← REMOVE-DOMINATED-PLANS(U')
  until MAX-DIFFERENCE(U,U') < ∈(1 - γ)/γ
  return U
```

Abbildung 17.9: Allgemeines Schema des Algorithmus für partiell beobachtbare MEPs. Der Schritt REMOVE-DOMINATED-PLANS und der Test MAX-DIFFERENCE werden normalerweise als lineare Programme implementiert.

Die Struktur des Algorithmus und seine Fehleranalyse sind vergleichbar mit dem grundlegenden Algorithmus der Wert-Iteration in Abbildung 17.4. Der Unterschied besteht vor allem darin, dass POMDP-VALUE-ITERATION statt der Berechnung einer Nutzenzahl für jeden Zustand eine Auflistung von nicht dominierten Plänen mit deren Nutzen-Hyperplänen verwaltet. Die Komplexität des Algorithmus hängt vorrangig davon ab, wie viele Pläne generiert werden. Für $|A|$ Aktionen und $|E|$ mögliche Beobachtungen lässt sich leicht zeigen, dass es

$$|A|^{O(|E|^{d-1})}$$

verschiedene Pläne der Tiefe d gibt. Selbst für die bescheidene Zweizustandswelt mit $d = 8$ beträgt die genaue Anzahl 2^{255}. Um dieses doppelexponentielle Wachstum zu vermeiden, ist es wichtig, die dominierten Pläne zu eliminieren: Die Anzahl der nicht dominierten Pläne mit $d = 8$ beträgt lediglich 144. Abbildung 17.8(d) zeigt die Nutzenfunktion für diese 144 Pläne.

Auch wenn Zustand 0 einen geringeren Nutzen als Zustand 1 hat, ist der Nutzen bei den dazwischenliegenden Belief States noch niedriger, da dem Agenten die benötigten Informationen fehlen, um eine gute Aktion auszuwählen. Aus diesem Grund hat Information einen Wert entsprechend dem in *Abschnitt 16.6* definiert Sinn und opti-

male Taktiken in partiell beobachtbaren MEPs binden oftmals Aktionen zur Informationserfassung ein.

Mit einer derartigen Nutzenfunktion lässt sich eine ausführbare Taktik extrahieren, indem man die optimale Hyperebene in einem gegebenen Belief State b sucht und die erste Aktion des entsprechenden Planes ausführt. In Abbildung 17.8(d) ist die entsprechende optimale Taktik immer noch dieselbe wie für Pläne der Tiefe 1: *Bleiben*, wenn $b(1) > 0{,}5$, andernfalls *Gehen*.

In der Praxis ist der Algorithmus zur Wert-Iteration von Abbildung 17.9 für größere Probleme ineffizient und hoffnungslos überfordert – das betrifft sogar schon den partiell beobachtbaren MEP für die 4×3-Welt. Das hängt hauptsächlich damit zusammen, dass der Algorithmus bei gegebenen n bedingten Plänen auf Ebene d die Anzahl $|A| \cdot n^{|E|}$ bedingte Pläne auf Ebene $d + 1$ konstruiert, bevor er die dominierten Pläne eliminiert. Seit der Entwicklung dieses Algorithmus in den 1970er Jahren hat es mehrere Fortschritte gegeben, einschließlich effizienterer Varianten der Wert-Iteration und verschiedene Arten von Algorithmen der Taktik-Iteration. Einige davon werden in den Hinweisen am Ende des Kapitels angesprochen. Allerdings ist es für allgemeine partiell beobachtbare MEPs sehr schwierig, optimale Taktiken zu finden (tatsächlich PSPACE-hart – d.h. wirklich sehr komplex). Probleme mit einigen Dutzend Zuständen sind oftmals nicht mehr handhabbar. Der nächste Abschnitt beschreibt ein anderes Näherungsverfahren für das Lösen von partiell beobachtbaren MEPs, das auf einer vorausschauenden Suche basiert.

17.4.3 Online-Agenten für partiell beobachtbare MEPs

In diesem Abschnitt skizzieren wir einen einfachen Ansatz für den Agentenentwurf in partiell beobachtbaren stochastischen Umgebungen. Die grundlegenden Entwurfselemente kennen wir bereits:

- Die Übergangs- und Sensormodelle werden durch ein **dynamisches Bayessches Netz** dargestellt (siehe *Kapitel 15*).
- Das dynamische Bayessche Netz wird durch Entscheidungs- und Nutzenknoten erweitert, die auch in den in *Kapitel 16* vorgestellten **Entscheidungsnetzen** verwendet wurden. Das resultierende Modell wird als **dynamisches Entscheidungsnetz** oder DDN (Dynamic Decision Network) bezeichnet.
- Mithilfe eines Filteralgorithmus werden alle neuen Wahrnehmungen und Aktionen eingebaut, um die Belief-State-Repräsentation zu aktualisieren.
- Entscheidungen werden getroffen, indem mögliche Aktionsfolgen vorwärts projiziert werden und die beste davon ausgewählt wird.

DBNs sind **faktorisierte Darstellungen** gemäß der Terminologie von *Kapitel 2*; normalerweise haben sie einen exponentiellen Komplexitätsvorteil gegenüber atomaren Darstellungen und können recht erhebliche Probleme der realen Welt modellieren. Der Agentenentwurf ist deshalb eine praktische Implementierung des in *Kapitel 2* skizzierten **nutzenbasierten Agenten**.

In den DBN wird der einzelne Zustand S_t zu einer Menge von Zustandsvariablen $\mathbf{X}_t$ und es kann mehrere Evidenzvariablen $\mathbf{E}_t$ geben. Mit A_t bezeichnen wir die Aktion zur Zeit t; das Übergangsmodell wird also zu $\mathbf{P}(\mathbf{X}_{t+1}|\mathbf{X}_t, A_t)$ und das Sensormodell zu $\mathbf{P}(\mathbf{E}_t|\mathbf{X}_t)$. Mit R_t bezeichnen wir den zur Zeit t erhaltenen Gewinn und mit U_t den Nut-

zen des Zustandes zur Zeit t. (Beide Variablen sind Zufallsvariablen.) Mit dieser Notation sieht ein dynamisches Entscheidungsnetz aus, wie in ▶ Abbildung 17.10 gezeigt.

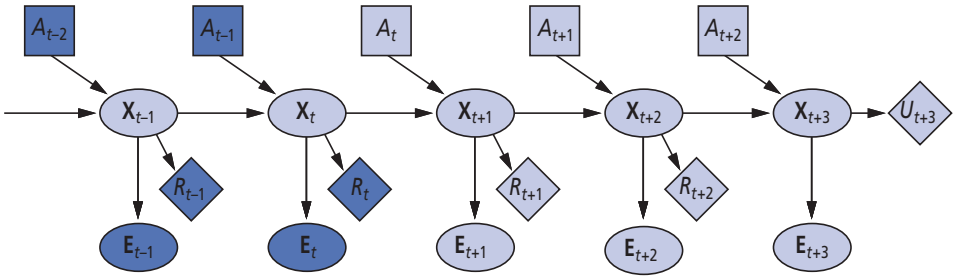


Abbildung 17.10: Die generische Struktur eines dynamischen Entscheidungsnetzes. Variablen mit bekannten Werten sind grau schattiert dargestellt. Die aktuelle Zeit ist t und der Agent muss entscheiden, was zu tun ist – d.h. einen Wert für A_t auswählen. Das Netz wurde drei Schritte in die Zukunft aufgerollt und stellt zukünftige Gewinne dar, ebenso wie den Nutzen des Zustandes für den in der Vorausschau ermittelten Horizont.

Dynamische Entscheidungsnetze können als Eingaben für jeden Algorithmus eines partiell beobachtbaren MEP verwendet werden, einschließlich der Algorithmen für Methoden der Wert- und Taktik-Iteration. In diesem Abschnitt konzentrieren wir uns auf vorausschauende Methoden, die Aktionsfolgen vom aktuellen Belief State aus vorwärts projizieren, ähnlich wie die Spielalgorithmen aus *Kapitel 5*. Das in Abbildung 17.10 gezeigte Netz wurde drei Schritte in die Zukunft projiziert; die aktuellen und zukünftigen Entscheidungen A sowie die zukünftigen Beobachtungen $\mathbf{E}$ und Gewinne R sind alle unbekannt. Beachten Sie, dass das Netz für $\mathbf{X}_{t+1}$ und $\mathbf{X}_{t+2}$ Knoten für die *Gewinne* enthält, für $\mathbf{X}_{t+3}$ jedoch Knoten für *Nutzen*. Das liegt daran, dass der Agent die (verminderte) Summe aller zukünftigen Gewinne maximieren muss und $U(\mathbf{X}_{t+3})$ den Gewinn für $\mathbf{X}_{t+3}$ und alle folgenden Gewinne darstellt. Wie in *Kapitel 5* gehen wir davon aus, dass U nur in einer annähernden Form verfügbar ist: Wenn genaue Nutzenwerte zur Verfügung stünden, wäre es nicht erforderlich, tiefer als eine Ebene vorauszuschauen.

▶ Abbildung 17.11 zeigt einen Teil des Suchbaumes, der dem für drei Schritte vorausschauenden dynamischen Entscheidungsnetz aus Abbildung 17.10 entspricht. Jeder der dreieckigen Knoten ist ein Belief State, in dem der Agent eine Entscheidung A_{t+i} für $i = 0, 1, 2, \ldots$ trifft. Die runden (Zufalls-)Knoten entsprechen Auswahlen durch die Umgebung, d.h. welche Evidenz $\mathbf{E}_{t+i}$ eintrifft. Beachten Sie, dass es keine Zufallsknoten gibt, die den Aktionsergebnissen entsprechen; das liegt daran, dass die Belief-State-Aktualisierung für eine Aktion unabhängig vom tatsächlichen Ergebnis deterministisch ist.

Der Belief State in jedem dreieckigen Knoten kann durch Anwendung eines Filteralgorithmus auf die Folge der Wahrnehmungen und Aktionen, die zu ihm geführt haben, berechnet werden. Auf diese Weise berücksichtigt der Algorithmus die Tatsache, dass der Agent für die Entscheidung A_{t+i} über die Wahrnehmungen $\mathbf{E}_{t+1}, \ldots \mathbf{E}_{t+i}$ verfügen *wird*, selbst wenn er zur Zeit t nicht weiß, wie diese Wahrnehmungen aussehen werden. Auf diese Weise berücksichtigt ein entscheidungstheoretischer Agent automatisch den Informationswert und führt gegebenenfalls Aktionen zur Informationserfassung aus.

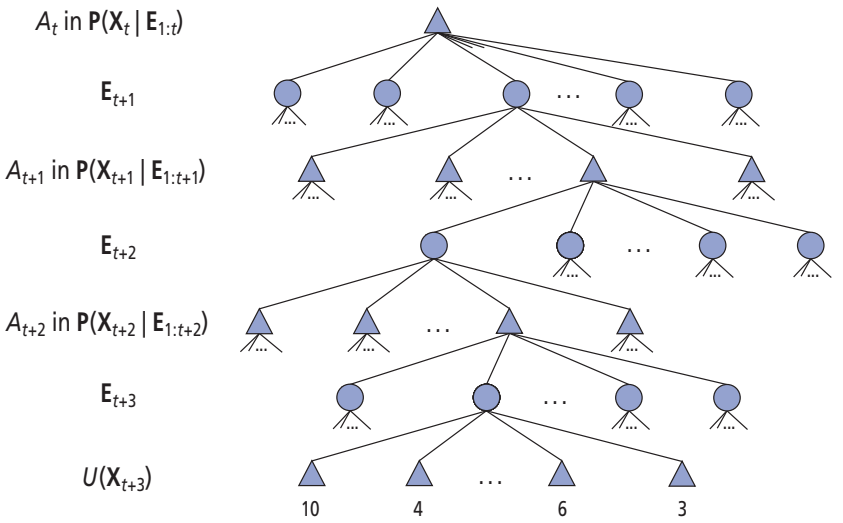


Abbildung 17.11: Teil der Vorausschau-Lösung des DDN aus Abbildung 17.10. Jede Entscheidung wird im gekennzeichneten Belief State getroffen.

Eine Entscheidung lässt sich aus dem Suchbaum extrahieren, indem die Nutzenwerte aus den Blättern zurückverfolgt werden, wobei ein Mittelwert über alle Zufallsknoten und das Maximum bei den Entscheidungsknoten verwendet wird. Das entspricht dem EXPECTIMINIMAX-Algorithmus für Spielbäume mit Zufallsknoten, außer dass (1) auch Gewinne in Nichtblattzuständen möglich sind und (2) die Entscheidungsknoten Belief States und keinen tatsächlichen Zuständen entsprechen. Die Zeitkomplexität einer erschöpfenden Suche bis zur Tiefe d ist gleich $O(|A|^d \cdot |\mathbf{E}|^d)$, wobei $|A|$ die Anzahl aller verfügbaren Aktionen und $|\mathbf{E}|$ die Anzahl möglicher Wahrnehmungen darstellt. (Dies ist weit weniger als die Anzahl der bedingten Pläne der Tiefe d, die durch Wert-Iteration generiert werden.) Für Probleme, deren Minderungsfaktor γ nicht zu nahe an 1 liegt, ist eine flache Suche häufig ausreichend, um fast optimale Entscheidungen zu ermitteln. Außerdem ist es möglich, den mittelnden Schritt für die Zufallsknoten anzunähern, indem man Stichproben aus der Menge möglicher Beobachtungen zieht, anstatt eine Summe über alle möglichen Beobachtungen zu bilden. Es gibt verschiedene andere Möglichkeiten, schnell gute annähernde Lösungen zu finden, die wir jedoch erst in *Kapitel 21* vorstellen werden.

Entscheidungstheoretische Agenten, die auf dynamischen Entscheidungsnetzen basieren, haben zahlreiche Vorteile im Vergleich zu anderen, einfacheren Agentenentwürfen, wie sie in früheren Kapiteln vorgestellt wurden. Insbesondere verarbeiten sie partiell beobachtbare, unsichere Umgebungen und können ihre „Pläne“ einfach revidieren, um unerwartete Evidenz zu verarbeiten. Mit geeigneten Sensormodellen können sie Sensorfehler kompensieren und planen, Informationen zu sammeln. Unter Zeitdruck und in komplexen Umgebungen zeigen sie eine „allmähliche Minderung“, wofür verschiedene Annäherungstechniken verwendet werden. Was also fehlt? Ein Mangel unseres DNN-basierten Algorithmus ist, dass er sich auf eine Vorwärtssuche durch den Zustandsraum verlässt, anstatt die in *Kapitel 11* beschriebenen hierarchischen und anderen erweiterten Planungstechniken zu verwenden. Es gab Versuche, diese Methoden in den probabilistischen Bereich zu erweitern, aber bisher waren diese alle ineffizient. Ein zweites,

verwandtes Problem ist die grundsätzlich aussagenlogische Natur der DDN-Sprache. Es wäre praktisch, wenn wir einige der Konzepte für probabilistische Sprachen erster Stufe auf das Problem der Entscheidungsfindung erweitern könnten. Neue Forschungen haben gezeigt, dass diese Erweiterung möglich ist und wesentliche Vorteile aufweist, wie in den Hinweisen am Ende dieses Kapitels noch beschrieben.

17.5 Entscheidungen mit mehreren Agenten: Spieltheorie

Dieses Kapitel hat sich vor allem mit der Entscheidungsfindung in unsicheren Umgebungen beschäftigt. Was aber passiert, wenn die Unsicherheit durch andere Agenten und ihre Entscheidungen verursacht wird? Und was passiert, wenn die Entscheidungen dieser Agenten wiederum durch unsere eigenen Entscheidungen beeinflusst werden? Wir haben uns schon in *Kapitel 5* mit dieser Frage beim Thema Spiele beschäftigt. Dort ging es jedoch hauptsächlich um Spiele, in denen die Spieler wechselseitig am Zug waren und die in einer vollständig beobachtbaren Umgebung stattfinden, für die eine Minimax-Suche optimale Züge ermitteln konnte. In diesem Abschnitt beschäftigen wir uns mit Aspekten der **Spieltheorie**, die Spiele mit gleichzeitigen Zügen und andere Quellen partieller Beobachtbarkeit analysieren. (Spieltheoretiker verwenden die Begriffe **vollständige** und **unvollständige Informationen** statt vollständig und partiell beobachtbar.) Die Spieltheorie kann auf mindestens zwei Arten eingesetzt werden:

1. **Agentenentwurf**: Die Spieltheorie kann die Entscheidungen des Agenten analysieren und den erwarteten Nutzen für jede Entscheidung berechnen (unter der Annahme, dass die anderen Agenten gemäß der Spieltheorie optimal agieren). Im Spiel **Zwei-Finger-Morra** beispielsweise zeigen zwei Spieler O und E gleichzeitig einen oder zwei Finger an. Sei die Gesamtzahl der Finger gleich f. Wenn f ungerade ist, erhält O f Euro von E, und wenn f gerade ist, erhält E f Euro von O. Die Spieltheorie kann die beste Taktik gegen einen rationalen Spieler und den erwarteten Gewinn für jeden Spieler ermitteln.[4]

2. **Mechanismenentwurf**: Wenn eine Umgebung von vielen Agenten bewohnt wird, kann es möglich sein, die Regeln für die Umgebung (d.h. das Spiel, das die Agenten spielen müssen) so zu definieren, dass der gemeinsame Besitz aller Agenten maximiert wird, wenn jeder Agent die spieltheoretische Lösung übernimmt, die seinen eigenen Nutzen maximiert. Die Spieltheorie kann beispielsweise helfen, die Protokolle einer Sammlung von Routern für den Internetverkehr zu planen, sodass jeder Router ein Interesse daran hat, so zu handeln, dass der globale Durchsatz maximiert wird. Der Mechanismenentwurf kann auch genutzt werden, um intelligente **Multiagenten-Systeme** zu konstruieren, die komplexe Probleme in verteilter Form lösen.

4 Morra ist eine entspannte Version eines **Inspektionsspieles**. In solchen Spielen wählt ein Inspektor einen Tag aus, an dem er ein Unternehmen inspiziert (wie beispielsweise ein Restaurant oder eine Biowaffenfabrik), und der Unternehmensleiter wählt einen Tag aus, an dem er alle verbotenen Dinge versteckt. Der Inspektor gewinnt, wenn sich die Tage unterscheiden, der Unternehmensleiter gewinnt, wenn sie gleich sind.

17.5.1 Ein-Zug-Spiele

Wir sehen uns zunächst eine eingeschränkte Menge von Spielen an: Spiele, bei denen alle Spieler gleichzeitig handeln und das Ergebnis des Spieles auf dieser einzigen Menge von Aktionen beruht. (Prinzipiell ist es nicht entscheidend, dass die Aktionen genau zur selben Zeit stattfinden; wichtig ist, dass kein Spieler die Entscheidungen der anderen Spieler kennt.) Die Einschränkung auf einen einzelnen Zug (und allein schon die Verwendung des Wortes „Spiel“) scheint das Ganze trivial zu machen, doch ist die Spieltheorie in der Tat eine ernste Angelegenheit. Herangezogen wird sie in Situationen der Entscheidungsfindung einschließlich der Versteigerung von Ölbohrrechten und Mobilfunkfrequenzen, Konkursverfahren, Produktentwicklung und Preisentscheidungen sowie der nationalen Verteidigung – Situationen, in denen es um Milliarden von Euros und Hunderttausende von Menschenleben geht. Ein Ein-Zug-Spiel ist durch drei Komponenten definiert:

- **Spieler** oder Agenten, die Entscheidungen treffen. Zwei-Spieler-Spiele haben die meiste Aufmerksamkeit erhalten, aber auch n-Spieler-Spiele für $n > 2$ sind gebräuchlich. Wir werden den Spielern Namen geben, die mit Großbuchstaben beginnen, wie beispielsweise *Anna* und *Bodo* oder *O* und *E*.
- **Aktionen**, die die Spieler auswählen können. Wir geben Aktionen Namen, die mit Kleinbuchstaben beginnen, wie beispielsweise *eins* oder *gestehen*. Die Spieler können dieselbe Menge an verfügbaren Aktionen haben, müssen es aber nicht.
- Eine **Auszahlungsfunktion**, die den Nutzen für jeden Spieler für jede Kombination von Aktionen aller Spieler angibt. Für Ein-Zug-Spiele lässt sich die Auszahlungsfunktion durch eine Matrix darstellen, die sogenannte **Strategieform** (auch als **Normalform** bezeichnet). Die Auszahlungsmatrix für Zwei-Finger-Morra sieht beispielsweise so aus:

	O: eins	O: zwei
E: eins	E = +2, O = –2	E = –3, O = +3
E: zwei	E = –3, O = +3	E = +4, O = –4

Zum Beispiel zeigt die untere rechte Ecke, dass die Auszahlung für *E* gleich 4 und für *O* gleich −4 ist, wenn *O* die Aktion *zwei* und *E* ebenfalls die Aktion *zwei* wählt.

Jeder Spieler in einem Spiel muss eine **Strategie** (das ist in der Spieltheorie der Name für eine *Taktik*) einnehmen und dann ausführen. Eine **reine Strategie** ist eine deterministische Taktik; für ein Ein-Zug-Spiel ist eine reine Strategie lediglich eine einzige Aktion. Bei vielen Spielen gelangt ein Agent mit einer **gemischten Strategie** zu besseren Ergebnissen. Dabei handelt es sich um eine zufallsgesteuerte Taktik, die Aktionen gemäß einer Wahrscheinlichkeitsverteilung auswählt. Die gemischte Strategie, die die Aktion a mit einer Wahrscheinlichkeit p und andernfalls eine Aktion b auswählt, wird dargestellt als $[p: a; (1 - p): b]$. Eine gemischte Strategie für Zwei-Finger-Morra könnte beispielsweise [0,5: *eins*; 0,5: *zwei*] sein. Ein **Strategieprofil** ist eine Zuweisung einer Strategie an jeden Spieler; für das Strategieprofil ist das **Ergebnis** des Spieles ein numerischer Wert für jeden Spieler.

Eine **Lösung** für ein Spiel ist ein Strategieprofil, nach dem jeder Spieler eine rationale Strategie anwendet. Wir werden feststellen, dass der wichtigste Aspekt in der Spieltheorie ist, die Bedeutung von „rational“ zu definieren, wenn jeder Agent nur einen Teil des Strategieprofiles auswählt, das das Ergebnis bestimmt. Beachten Sie, dass die Ergebnisse die tatsächlichen Resultate beim Spielen eines Spieles sind, während Lösungen theoretische Konstrukte für die Spielanalyse darstellen. Bestimmte Spiele haben eine Lösung nur in gemischten Strategien (mehr dazu später). Das bedeutet jedoch nicht, dass ein Spieler eine gemischte Strategie anwenden muss, um rational zu handeln.

Betrachten Sie die folgende Geschichte: Zwei mutmaßliche Einbrecher, Anna und Bodo, werden in flagranti in der Nähe eines Tatortes ertappt und von der Polizei separat verhört. Der Staatsanwalt bietet einen Deal an: Wenn du gegen deinen Partner aussagst, dass er der Anführer einer Einbrecherbande ist, bleibst du straffrei, während dein Partner zehn Jahre hinter Gitter geht. Wenn ihr aber beide gegeneinander aussagt, bekommt ihr je 5 Jahre. Anna und Bodo wissen auch, dass sie, wenn sie beide nicht gestehen, nur jeweils ein Jahr für das geringere Vergehen bekommen, gestohlenes Eigentum zu besitzen. Jetzt stehen Anna und Bodo dem sogenannten **Gefangenendilemma** gegenüber: Sollen sie gestehen oder die Aussage verweigern? Als rationale Agenten wollen Anna und Bodo ihren eigenen erwarteten Nutzen maximieren. Nehmen wir an, Anna ist das Schicksal ihres Partners herzlich egal; deshalb sinkt ihr Nutzen proportional zu den Jahren, die sie im Gefängnis verbringen wird – unabhängig davon, was mit Bodo passiert. Bodo geht es genauso. Um eine rationale Entscheidung zu unterstützen, bauen sie beide die folgende Auszahlungsmatrix auf:

	Anna: gestehen	**Anna: leugnen**
Bodo: gestehen	A = –5, B = –5	A = –10, B = 0
Bodo: leugnen	A = 0, B = –10	A = –1, B = –1

Anna analysiert die Auszahlungsmatrix wie folgt: Angenommen, Bodo gesteht. Dann bekomme ich fünf Jahre, wenn ich gestehe, andernfalls zehn Jahre; in diesem Fall ist es also besser, zu gestehen. Leugnet Bodo dagegen, bekomme ich null Jahre, wenn ich gestehe, und ein Jahr, wenn ich leugne; in diesem Fall ist es also besser zu gestehen. Es ist also in jedem Fall für mich besser zu gestehen; deshalb muss ich genau das tun.

Anna hat erkannt, dass *gestehen* eine **dominante Strategie** für das Spiel ist. Wir sagen, eine Strategie s für Spieler p **dominiert** die Strategie s' **stark**, wenn das Ergebnis für s für p besser als das Ergebnis für s' ist, egal welche Strategien die anderen Spieler auswählen. Eine Strategie s **dominiert** s' **schwach**, wenn s besser als s' für mindestens ein Strategieprofil ist und nicht schlechter als ein anderes. Eine dominante Strategie ist eine Strategie, die alle anderen dominiert. Es ist irrational, eine streng dominante Strategie zu spielen, und irrational, keine dominante Strategie zu spielen, wenn es eine gibt. Weil Anna rational ist, wählt sie die dominante Strategie. Wir brauchen aber noch ein wenig mehr Terminologie: Wir sagen, ein Ergebnis ist **Pareto-optimal**[5], wenn es kein anderes Ergebnis gibt, das alle Spieler bevorzugen. Ein Ergebnis wird von einem anderen Ergebnis **Pareto-dominiert**, wenn alle Spieler das andere Ergebnis bevorzugen.

5 Die Pareto-Optimalität ist nach dem Wirtschaftswissenschaftler *Vilfredo Pareto* (1848–1923) benannt.

Wenn Anna gescheit und rational ist, argumentiert sie wie folgt weiter: Die dominante Strategie von Bob ist, ebenfalls zu gestehen. Aus diesem Grund gesteht er, und sie bekommen beide fünf Jahre. Wenn jeder Spieler eine dominante Strategie hat, wird die Kombination aus diesen Strategien als **dominantes Strategiegleichgewicht** bezeichnet. Im Allgemeinen bildet ein Strategieprofil ein **Gleichgewicht**, wenn kein Spieler davon profitieren kann, die Strategie zu wechseln, während alle anderen Spieler bei ihrer Strategie bleiben. Ein Gleichgewicht ist im Wesentlichen ein **lokales Optimum** im Strategieraum; es ist die Spitze eines Gipfels, der nach allen Dimensionen hin fällt, wobei eine Dimension der Strategieauswahl eines Spielers entspricht.

Tipp

Der Mathematiker John Nash (1928–) bewies, *dass jedes Spiel mindestens ein Gleichgewicht hat*. Das allgemeine Gleichgewichtskonzept bezeichnet man ihm zu Ehren heute als **Nash-Gleichgewicht**. Offensichtlich ist ein dominantes Strategiegleichgewicht ein Nash-Gleichgewicht (Übung 17.16), doch haben manche Spiele Nash-Gleichgewichte, jedoch keine dominanten Strategien.

Das *Dilemma* im Gefangenendilemma ist, dass das Ergebnis des Gleichgewichtspunktes für beide Spieler schlechter ist als das Ergebnis, das sie erhalten, wenn sie beide nicht gestehen würden. Mit anderen Worten wird (*gestehen*, *gestehen*) durch das Ergebnis $(-1, -1)$ von (*leugnen*, *leugnen*) Pareto-dominiert. Gibt es eine Möglichkeit für Anna und Bodo, das Ergebnis $(-1, -1)$ zu erzielen? Es ist sicherlich eine *erlaubte* Option für beide, ein Geständnis zu verweigern, aber es ist schwer zu erkennen, wie rationale Agenten angesichts der gegebenen Definition des Spieles dorthin kommen können. Jeder Spieler, der über den Zug *leugnen* nachdenkt, wird erkennen, dass er besser beraten ist, *gestehen* zu spielen. Das ist die Anziehungskraft eines Gleichgewichtspunktes. Die Spieltheoretiker sind sich einig, dass ein Nash-Gleichgewicht eine notwendige Bedingung für eine Lösung ist – während sie sich nicht darüber einig sind, ob es sich um eine ausreichende Bedingung handelt.

Zur Lösung (*leugnen*, *leugnen*) gelangen wir recht leicht, wenn wir das Spiel modifizieren. Zum Beispiel könnten wir zu einem wiederholten Spiel wechseln, in dem die Spieler wissen, dass sie wieder aufeinandertreffen. Oder die Agenten könnten moralische Glauben haben, die Zusammenarbeit und Fairness fördern. Das heißt, sie haben eine andere Nutzenfunktion, was eine andere Auszahlungsmatrix erfordert, wodurch letztlich ein anderes Spiel entsteht. Später sehen wir, dass Agenten mit begrenzter Rechenleistung nicht mehr in der Lage sind, absolut rational zu schließen, sondern Nichtgleichgewichtsergebnisse erreichen können. Das trifft auch auf Agenten zu, die wissen, dass der andere Agent nur begrenzt rational handeln kann. In jedem Fall betrachten wir ein anderes als das oben durch die Auszahlungsmatrix beschriebene Spiel.

Jetzt betrachten wir ein Spiel, das keine dominante Strategie besitzt. Acme, ein Hersteller von Videospiel-Konsolen, muss entscheiden, ob die nächste Spielmaschine Blue-ray Discs oder DVDs verwendet. In der Zwischenzeit muss Best, der Hersteller von Videospiel-Software, entscheiden, ob er sein nächstes Spiel auf Blue-ray oder auf DVD produziert. Die Gewinne sind für beide positiv, wenn sie sich für dieselbe Lösung entscheiden, und negativ, wenn sie sich nicht für dieselbe Lösung entscheiden, wie in der folgenden Auszahlungsmatrix gezeigt:

	Acme:blueray	Acme:dvd
Best:blueray	A = +9, B = +9	A = –4, B = –1
Best:dvd	A = –3, B = –1	A = +5, B = +5

Tipp

Es gibt kein dominantes Strategiegleichgewicht für dieses Spiel, aber es gibt zwei Nash-Gleichgewichte: (*blueray*, *blueray*) und (*dvd*, *dvd*). Wir wissen, dass es sich um Nash-Gleichgewichte handelt, weil ein Spieler schlechter wegkommt, wenn er einseitig zu einer anderen Strategie wechselt. Jetzt haben die Agenten ein Problem: *Es gibt mehrere akzeptable Lösungen, aber wenn jeder Agent eine andere Lösung anstrebt, leiden beide Agenten*. Wie können sie sich auf eine Lösung einigen? Eine Antwort ist, dass beide Agenten die Pareto-optimale Lösung (*blueray*, *blueray*) wählen sollten; d.h., wir können die Definition von „Lösung“ auf das einzige Pareto-optimale Nash-Gleichgewicht beschränken, *vorausgesetzt, es gibt ein solches*. Jedes Spiel hat mindestens eine Pareto-optimale Lösung, aber ein Spiel könnte mehrere oder keine Gleichgewichtspunkte aufweisen. Wenn zum Beispiel (*blueray*, *blueray*) eine Auszahlung von (5, 5) hat, gibt es zwei gleiche Pareto-optimale Gleichgewichtspunkte. Um zwischen ihnen zu wählen, können die Agenten entweder raten oder miteinander *kommunizieren*, wozu sie entweder eine Konvention einrichten, die die Lösungen in eine Reihenfolge bringt, bevor das Spiel beginnt, oder sie sprechen sich während des Spieles ab, um eine gegenseitig nützliche Lösung zu erzielen (das würde Kommunikationsaktionen als Teil eines sequentiellen Spieles bedeuten). Kommunikation tritt also in der Spieltheorie aus genau denselben Gründen auf wie bei der Multiagenten-Planung in *Abschnitt 11.4*. Spiele, in denen die Spieler miteinander kommunizieren müssen, werden auch als **Koordinationsspiele** bezeichnet.

Ein Spiel kann mehrere Nash-Gleichgewichte besitzen; woher wissen wir, dass jedes Spiel mindestens eines haben muss? Manche Spiele haben keine *rein strategischen* Nash-Gleichgewichte. Betrachten Sie beispielsweise ein reines Strategieprofil für Zwei-Finger-Morra (*Abschnitt 17.5*). Wenn die Gesamtzahl der Finger gerade ist, wird *O* wechseln, ist sie andererseits ungerade, wird *E* wechseln. Aus diesem Grund kann kein reines Strategieprofil ein Gleichgewicht sein und wir müssen auf gemischte Strategien ausweichen.

Aber *welche* gemischte Strategie? 1928 entwickelte von Neumann eine Methode, die *optimale* gemischte Strategie für Zwei-Spieler-**Nullsummenspiele** zu finden. Ein Nullsummenspiel ist ein Spiel, in dem die Auszahlungen in jeder Zelle der Auszahlungsmatrix gleich null sind.[6] Offensichtlich handelt es sich bei Morra um ein solches Spiel. Für Zwei-Spieler-Nullsummenspiele wissen wir, dass die Auszahlungen gleich und entgegengesetzt sind; deshalb müssen wir die Auszahlungen nur für einen Spieler berücksichtigen, der der Maximierer ist (so wie in *Kapitel 5*). Für Morra wählen wir den geraden Spieler *E* als Maximierer aus; wir können also die Auszahlungsmatrix durch die Werte $U_E(e, o)$ definieren – die Auszahlung für *E*, wenn sich *E für e* und *O* für *o* entscheidet. Die Methode von von Neumann wird als **Maximin**-Technik bezeichnet und funktioniert wie folgt:

6 oder einen konstanten Wert haben – siehe *Abschnitt 5.1*.

- Angenommen, wir ändern die Regeln wie folgt: Zuerst wählt E seine Strategie aus und legt sie O offen. Dann wählt O seine Strategie aus, wobei ihm die Strategie von E bekannt ist. Schließlich werten wir die erwartete Auszahlung des Spieles basierend auf den gewählten Strategien aus. Damit erhalten wir ein Spiel mit abwechselnden Zügen, auf das wir den **Minimax**-Standardalgorithmus aus *Kapitel 5* anwenden können. Wir nehmen an, wir erhalten daraus das Ergebnis $U_{E,O}$. Offensichtlich bevorzugt dieses Spiel O; der tatsächliche Nutzen U des Spieles (aus der Perspektive von E) ist also *mindestens* $U_{E,O}$. Wenn wir beispielsweise nur reine Strategien betrachten, hat der Minimax-Spielbaum einen Wurzelwert von -3 (siehe ▶ Abbildung 17.12(a)); wir wissen also, dass $U \geq -3$.

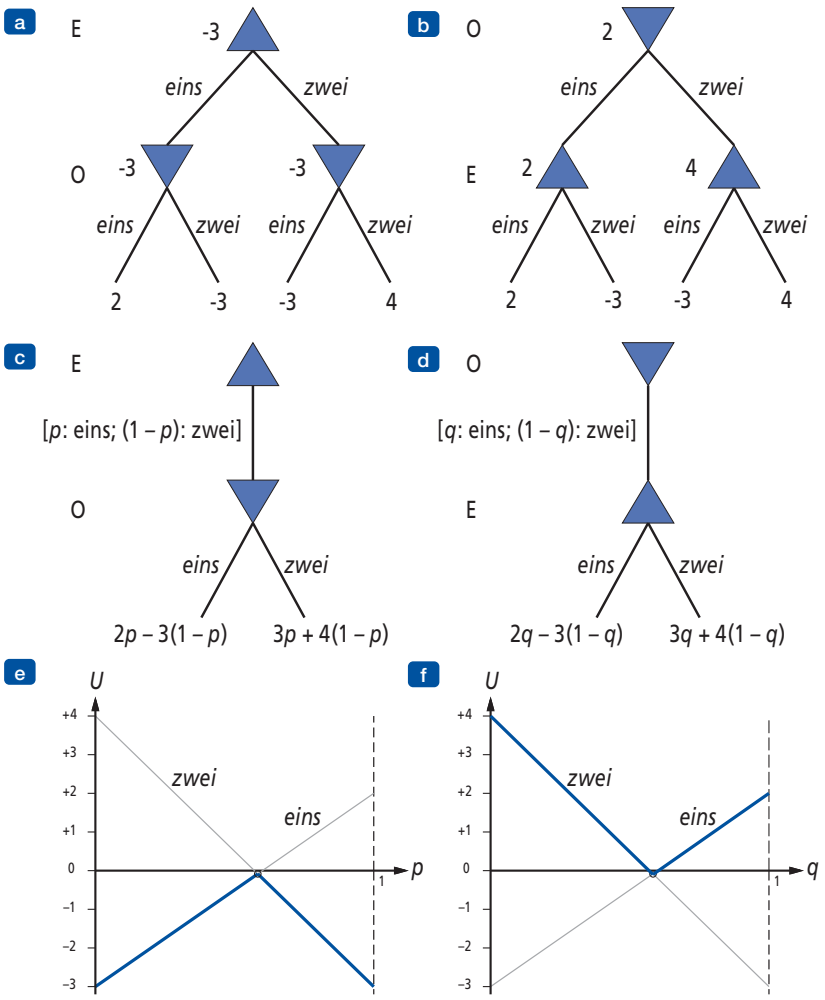


Abbildung 17.12: (a) und (b): Minimax-Spielbäume für Zwei-Finger-Morra, wenn die Spieler abwechselnd reine Strategien spielen. (c) und (d): Parametrisierte Spielbäume, wobei der erste Spieler eine gemischte Strategie spielt. Die Auszahlungen sind von dem Wahrscheinlichkeitsparameter (p oder q) in der gemischten Strategie abhängig. (e) und (f): Für jeden Wert des Wahrscheinlichkeitsparameters wählt der zweite Spieler die „bessere" der zwei Aktionen; der Wert der gemischten Strategie des ersten Spielers ist also durch die fett ausgezeichneten Linien angegeben. Der erste Spieler wählt den Wahrscheinlichkeitsparameter für die gemischte Strategie als Schnittpunkt.

- Angenommen, wir ändern jetzt die Regeln, um O zu zwingen, seine Strategie zuerst offenzulegen, gefolgt von E. Der Minimax-Wert dieses Spieles ist dann $U_{O,E}$ und weil dieses Spiel E bevorzugt, wissen wir, dass U *höchstens* $U_{O,E}$ ist. Bei reinen Strategien ist der Wert +2 (siehe ▶ Abbildung 17.12(b)); damit wissen wir $U \leq +2$.

Wenn wir diese beiden Argumente kombinieren, erkennen wir, dass der tatsächliche Nutzen U für das ursprüngliche Spiel Folgendes erfüllen muss:

$$U_{E,O} \leq U \leq U_{O,E} \quad \text{oder in diesem Fall} \quad -3 \leq U \leq 2.$$

Tipp

Um den Wert von U festzumachen, müssen wir gemischte Strategien analysieren. Beachten Sie zunächst Folgendes: *Nachdem der erste Spieler seine Strategie offengelegt hat, kann der zweite Spieler ebenfalls eine reine Strategie wählen*. Der Grund dafür ist einfach: Wenn der zweite Spieler eine gemischte Strategie [p: *eins*, $(1 - p)$:*zwei*] spielt, ist sein erwarteter Nutzen eine Linearkombination $(p \cdot u_{eins} + (1 - p) \cdot u_{zwei})$ der Nutzen der reinen Strategien u_{eins} und u_{zwei}. Diese Linearkombination kann nie besser als das Bessere von u_{eins} und u_{zwei} sein, sodass der zweite Spieler einfach die bessere wählen kann.

Im Sinne dieser Beobachtung kann man sich die Minimax-Bäume mit unendlich vielen Verzweigungen an der Wurzel vorstellen, die den unendlich vielen gemischten Strategien entsprechen, aus denen der erste Spieler auswählen kann. Jede davon führt zu einem Knoten mit zwei Verzweigungen, die den reinen Strategien für den zweiten Spieler entsprechen. Wir können diese unendlichen Bäume wie folgt darstellen, indem wir eine „parametrisierte" Auswahl an der Wurzel verwenden:

- Wenn E zuerst zieht, haben wir die in ▶ Abbildung 17.12(c) gezeigte Situation. E wählt die Strategie [p: *eins*, $(1 - p)$:*zwei*] an der Wurzel und dann wählt O eine reine Strategie (und folglich einen Zug) bei bekanntem Wert von p. Wenn O den Zug *eins* wählt, dann ist die erwartete Auszahlung (für E) gleich $2p - 3(1 - p) = 5p - 3$; wenn O *zwei* wählt, ist die erwartete Auszahlung gleich $-3p + 4(1 - p) = 4 - 7p$. Wir können diese beiden Auszahlungen als Geraden in einem Graphen darstellen, wobei p von 0 bis 1 auf der x-Achse reicht, wie in ▶ Abbildung 17.12(e) gezeigt. O, der Minimierer, wählt immer die untere der beiden Linien, wie durch die fett ausgezeichneten Linien in der Abbildung gezeigt. Aus diesem Grund ist das Beste, was E an der Wurzel tun kann, p als Schnittpunkt auszuwählen, nämlich bei

$$5p - 3 = 4 - 7p \quad \Rightarrow \quad p = 7/12.$$

  Der Nutzen von E an dieser Stelle ist $U_{E,O} = -1/12$.

- Wenn O zuerst zieht, haben wir die in ▶ Abbildung 17.12(d) gezeigte Situation. O wählt die Strategie [q: *eins*, $(1 - q)$:*zwei*] an der Wurzel und dann wählt E einen Zug für den bekannten Wert von q. Die Auszahlungen sind gleich $2q - 3(1 - q) = 5q - 3$ und $-3q + 4(1 - q) = 4 - 7q$.[7] Auch hier zeigt ▶ Abbildung 17.12(f), dass O an der Wurzel am besten den Schnittpunkt wählt:

$$5q - 3 = 4 - 7q \quad \Rightarrow \quad q = 7/12.$$

  Der Nutzen von E an dieser Stelle ist $U_{O,E} = -1/12$.

7 Es ist ein Zufall, dass diese Gleichungen dieselben wie für p sind; der Zufall entsteht, weil $U_E(eins, zwei) = U_E(zwei, eins) = -3$. Das erklärt auch, warum die optimale Strategie für beide Spieler gleich ist.

Jetzt wissen wir, dass der tatsächliche Nutzen des Spieles zwischen $-1/12$ und $-1/12$ liegt, d.h. genau $-1/12$ ist! (Die Moral ist also, dass man beim Spielen dieses Spieles besser O als E ist.) Darüber hinaus ist der tatsächliche Nutzen, der durch die gemischte Strategie erzielt wird, [7/12: *eins*, 5/12:*zwei*], die von beiden Spielern gespielt werden sollte. Diese Strategie wird als **Maximin-Gleichgewicht** des Spieles bezeichnet und es handelt sich dabei um ein Nash-Gleichgewicht. Beachten Sie, dass jede Komponentenstrategie in einer gleichgewichtig gemischten Strategie denselben erwarteten Nutzen hat. In diesem Fall haben sowohl *eins* als auch *zwei* denselben erwarteten Nutzen, $-1/12$, so wie die gemischte Strategie selbst.

Tipp

Unser Ergebnis für Zwei-Finger-Morra ist ein Beispiel für das allgemeine Ergebnis von von Neumann: *Jedes Zwei-Spieler-Nullsummenspiel hat ein Maximin-Gleichgewicht, wenn Sie gemischte Strategien zulassen*. Darüber hinaus ist jedes Nash-Gleichgewicht in einem Nullsummenspiel ein Maximin für beide Spieler. Ein Spieler, der die Maximin-Strategie übernimmt, hat zwei Garantien: Erstens ist keine andere Strategie besser gegen einen Gegner, der gut spielt (obwohl es bestimmte Strategien gibt, die besser abschneiden, weil sie irrationale Fehler eines Gegners ausnutzen). Zweitens kann der Spieler ebenso gut weiterspielen, selbst wenn die Strategie dem Gegner bekannt gemacht wurde.

Der allgemeine Algorithmus für die Ermittlung von Maximin-Gleichgewichten in Nullsummenspielen ist etwas komplizierter, als Abbildung 17.12(e) und Abbildung 17.12(f) annehmen lassen. Wenn es n mögliche Aktionen gibt, ist eine gemischte Strategie ein Punkt im n-dimensionalen Raum und die Linien werden zu Hyperebenen. Außerdem ist es für einige reine Strategien für den zweiten Spieler möglich, dass sie von anderen dominiert werden, sodass sie nicht optimal gegenüber *jeder* Strategie des ersten Spielers sind. Nachdem alle diese Strategien entfernt wurden (was möglicherweise wiederholt nötig ist), ist die optimale Auswahl an der Wurzel der höchste (oder tiefste) Schnittpunkt der restlichen Hyperebenen. Die Ermittlung dieser Auswahl ist ein Beispiel für ein Problem einer **linearen Programmierung**: die Maximierung einer Zielfunktion gemäß bestimmten linearen Beschränkungen. Solche Probleme können unter Verwendung von Standardtechniken in einer Zeit polynomial zur Anzahl der Aktionen (und der Anzahl der für die Spezifizierung der Gewinnfunktion verwendeten Bits, wenn Sie technisch werden wollen) gelöst werden.

Es bleibt die Frage, was ein rationaler Agent *machen* soll, wenn er ein einzelnes Morra-Spiel spielt. Der rationale Agent leitet die Tatsache ab, dass [7/12: *eins*, 5/12:*zwei*] die Maximin-Gleichgewichtsstrategie ist, und geht davon aus, dass dies ein wechselseitiges Wissen ist, das auch der rationale Gegner besitzt. Der Agent könnte einen zwölfseitigen Würfel oder einen Zufallszahlengenerator verwenden, um gemäß dieser gemischten Strategie zufällig auszuwählen, dann wäre die erwartete Auszahlung gleich $-1/12$ für E. Der Agent könnte aber auch einfach entscheiden, *eins* oder *zwei* zu spielen. In jedem Fall bleibt die erwartete Auszahlung gleich $-1/12$ für E. Seltsamerweise beeinträchtigt eine einseitige Auswahl einer bestimmten Aktion die erwartete Auszahlung nicht, aber wenn der andere Agent weiß, dass man eine solche einseitige Entscheidung getroffen hat, *wirkt* sich dies auf die erwartete Auszahlung *aus*, weil der Gegner seine Strategie entsprechend anpassen kann.

Das Ermitteln der Gleichgewichte in Nicht-Nullsummenspielen ist etwas komplizierter. Der allgemeine Ansatz umfasst zwei Schritte: (1) Auflistung aller möglichen Untermengen von Aktionen, die gemischte Strategien bilden könnten. Probieren Sie beispielsweise zuerst alle Strategieprofile aus, wobei jeder Spieler eine einzelne Aktion verwendet, dann solche, wo jeder Spieler eine oder zwei Aktionen ausführt, usw. Dies ist exponenti-

ell zur Anzahl der Aktionen und deshalb nur für relativ kleine Spiele geeignet. (2) Für jedes in (1) aufgelistete Strategieprofil prüfen Sie, ob es sich um ein Gleichgewicht handelt. Dazu löst man eine Menge von Gleichungen und Ungleichungen, ähnlich denjenigen im Nullsummenfall. Für zwei Spieler sind diese Gleichungen linear und können mit grundlegenden linearen Programmiertechniken gelöst werden, aber für drei oder mehr Spieler sind sie nicht linear und möglicherweise sehr schwierig zu lösen.

17.5.2 Wiederholte Spiele

Bisher haben wir nur Spiele betrachtet, die aus einem einzigen Zug bestehen. Der einfachste Fall eines Spieles mit Mehrfachzügen ist das **wiederholte Spiel**, wobei die Spieler wiederholt derselben Auswahl gegenüberstehen, jedes Mal allerdings mit dem Wissen über den Verlauf aller vorherigen Auswahlen der Spieler. Ein Strategieprofil für ein wiederholtes Spiel spezifiziert eine Aktionsauswahl für jeden Spieler zu jedem Zeitschritt für jeden möglichen Verlauf vorheriger Auswahlen. Wie bei den MEPs sind die Auszahlungen über die Zeit additiv.

Betrachten wir die Version des Gefangenendilemmas als wiederholtes Spiel. Werden Anna und Bodo zusammenarbeiten und die Aussage verweigern, wenn sie wissen, dass sie sich wiedersehen werden? Die Antwort ist von den Details des Szenarios abhängig. Nehmen wir beispielsweise an, Anna und Bodo wissen, dass sie genau 100 Runden Gefangenendilemma spielen müssen. Sie wissen beide, dass die 100. Runde kein wiederholtes Spiel ist – d.h., ihr Ergebnis kann keine Auswirkung auf zukünftige Runden haben –, deshalb wählen sie beide in dieser Runde die dominante Strategie, *gestehen*. Nachdem die 100. Runde jedoch festgelegt ist, kann die 99. Runde keine Wirkung auf nachfolgende Runden haben; damit hat auch sie ein dominantes Strategiegleichgewicht bei (*gestehen*, *gestehen*). Durch Induktion wählen beide Spieler in jeder Runde *gestehen* und erhalten eine Gefängnisstrafe von je 500 Jahren.

Wir erhalten unterschiedliche Lösungen, wenn wir die Interaktionsregeln ändern. Nehmen wir beispielsweise an, dass es nach jeder Runde eine 99%ige Wahrscheinlichkeit gibt, dass sich die Spieler wiedersehen. Die erwartete Rundenzahl ist dann weiterhin 100, aber kein Spieler weiß sicher, welche Runde die letzte sein wird. Unter diesen Umständen ist ein kooperativeres Verhalten möglich. Eine Gleichgewichtsstrategie ist beispielsweise für jeden Spieler, die Aussage zu *leugnen*, es sei denn, der andere Spieler hat je *gestehen* gespielt. Diese Strategie könnte auch als **fortlaufende Bestrafung** bezeichnet werden. Angenommen, beide Spieler wenden diese Strategie an und es handelt sich dabei um wechselseitiges Wissen übereinander. Solange kein Spieler *gestehen* gespielt hat, ist die erwartete zukünftige Gesamtauszahlung zu jedem Zeitpunkt für jeden Spieler gleich

$$\sum_{t=0}^{\infty} 0{,}99^t \cdot (-1) = -100.$$

Ein Spieler, der von der Strategie abweicht und *gestehen* wählt, gewinnt 0 Punkte statt -1 im nächsten Zug, aber von nun an spielen beide Spieler *gestehen* und die gesamte erwartete zukünftige Auszahlung wird zu

$$0 + \sum_{t=1}^{\infty} 0{,}99^t \cdot (-5) = -495.$$

Aus diesem Grund gibt es bei keinem Schritt einen Ansporn, von (*leugnen*, *leugnen*) abzuweichen. Eine fortlaufende Bestrafung ist die Strategie der „gegenseitigen versicherten Vernichtung“ des Gefangenendilemmas: Nachdem einer der Spieler beschließt zu *gestehen*, stellt er damit sicher, dass beide Spieler sehr zu leiden haben. Das funktioniert jedoch als Abschreckung nur, wenn der andere Spieler glaubt, Sie hätten diese Strategie angewendet – oder zumindest, dass Sie sie angewendet haben könnten.

Es gibt andere Strategien, die nachsichtiger sind. Die bekannteste davon namens **Tit-for-tat** (etwa: wie du mir, so ich dir) fordert, mit *leugnen* zu beginnen, und wiederholt dann den jeweils vorherigen Zug des anderen Spielers für alle folgenden Züge. Anna würde also leugnen, solange Bodo leugnet, und in dem Zug gestehen, nachdem Bodo gestanden hat, aber wieder leugnen, wenn Bodo geleugnet hat. Diese Strategie ist ganz einfach, hat sich aber als äußerst robust und effektiv gegen eine Vielzahl anderer Strategien durchgesetzt.

Wir können auch andere Lösungen erhalten, indem wir die Agenten ändern und nicht die Regeln. Angenommen, die Agenten sind endliche Automaten mit n Zuständen und sie spielen ein Spiel mit insgesamt $m > n$ Schritten. Die Agenten sind somit nicht in der Lage, die Anzahl der restlichen Schritte darzustellen, und sie müssen diese Zahl als unbekannt annehmen. Sie können also keine Induktion ausführen und es steht ihnen frei, bei dem bevorzugten (*leugnen*, *leugnen*)-Gleichgewicht anzukommen. In diesem Fall ist das Unwissen wunderbar – oder vielmehr, die Taktik, Ihren Agenten glauben zu machen, dass Sie unwissend sind, ist wunderbar. Ihr Erfolg in diesen wiederholten Spielen hängt davon ab, ob der andere Spieler Sie als Tyrann oder Dummkopf *wahrnimmt*, und nicht von Ihren tatsächlichen Charakterzügen.

17.5.3 Sequentielle Spiele

Im allgemeinen Fall besteht ein Spiel aus einer Folge von Zügen, die nicht gleich sein müssen. Derartige Spiele lassen sich am besten als Spielbaum darstellen, was Spieltheoretiker als **Extensivform** bezeichnen. Der Baum umfasst die gleichen Informationen, wie sie *Abschnitt 5.1* gezeigt hat: einen Ausgangszustand S_0, eine Funktion PLAYER(s), die angibt, welcher Spieler am Zug ist, eine Funktion ACTIONS(s), die die möglichen Aktionen auflistet, eine Funktion RESULT(s, a), die den Übergang zu einem neuen Zustand definiert, und eine partielle Funktion UTILITY(s, p), die nur für Terminalzustände definiert ist, um die Auszahlung für jeden Spieler anzugeben.

Um stochastische Spiele wie zum Beispiel Backgammon darzustellen, fügen wir einen speziellen Spieler, *Zufall*, hinzu, der zufällige Aktionen ausführen kann. Die „Strategie“ von *Zufall* ist Teil der Spieldefinition und wird als Wahrscheinlichkeitsverteilung über Aktionen spezifiziert (wobei die anderen Spieler ihre eigene Strategie auswählen können). Zur Darstellung von Spielen mit nichtdeterministischen Aktionen wie zum Beispiel Billard gliedern wir die Aktion in zwei Teile: Die Aktion des Spielers selbst hat ein deterministisches Ergebnis und *Zufall* hat dann einen Zug, um auf die Aktion in seiner eigenen launischen Art zu reagieren. Um gleichzeitige Züge wie im Gefangenendilemma oder Zwei-Finger-Morra darzustellen, legen wir eine willkürliche Reihenfolge der Spieler fest, können aber zusichern, dass die Aktionen des früheren Spielers für die nachfolgenden Spieler nicht beobachtbar sind: Das heißt, Anna muss sich zuerst für *leugnen* oder *gestehen* entscheiden, dann wählt Bodo, doch weiß Bodo zu diesem Zeitpunkt nicht, welche Auswahl Anna getroffen hat (wir können auch die Tat-

sache darstellen, dass der Zug erst später offengelegt wird). Allerdings nehmen wir an, dass sich die Spieler immer an alle ihre eigenen vorherigen Aktionen erinnern; diese Annahme wird als **vollständiges Gedächtnis** (Perfect Recall) bezeichnet.

Die Extensivform unterscheidet sich von den Spielbäumen in *Kapitel 5* durch die Darstellung partieller Beobachtbarkeit. Wie *Abschnitt 5.6* gezeigt hat, kann ein Spieler in einem partiell beobachtbaren Spiel wie zum Beispiel Kriegspiel einen Spielbaum über dem Raum der **Belief States** erzeugen. Wir haben gesehen, dass ein Spieler mit diesem Baum in manchen Fällen eine Zugfolge (eine Strategie) finden kann, die zu einem erzwungenen Schachmatt führt, unabhängig davon, in welchem tatsächlichen Zustand begonnen wurde, und unabhängig von der Strategie, die der Gegner verwendet. Allerdings sind die Techniken von *Kapitel 5* nicht in der Lage, einem Spieler zu sagen, was zu tun ist, wenn es kein garantiertes Schachmatt gibt. Hängt die beste Strategie des Spielers von der Strategie des Gegners und umgekehrt ab, kann Minimax (oder Alpha-Beta) allein keine Lösung finden. Die Extensivform *erlaubt* es uns, Lösungen zu finden, da sie die Belief States (von Spieltheoretikern als **Informationsbezirke** oder **Informationsmengen** bezeichnet) aller Spieler auf einmal darstellt. Von dieser Darstellung aus können wir Gleichgewichtslösungen genau wie bei Spielen in Normalform finden.

Als einfaches Beispiel eines sequentiellen Spieles platzieren Sie zwei Agenten in der 4×3-Welt von Abbildung 17.1 und lassen sie gleichzeitig ziehen, bis ein Agent ein Feld mit Ausgang erreicht und für dieses Feld die Auszahlung erhält. Wenn wir festlegen, dass keine Bewegung stattfindet, falls beide Agenten versuchen, gleichzeitig auf dasselbe Feld zu gelangen (ein häufiges Problem an vielen Verkehrsknotenpunkten), können bestimmte reine Strategien ewig stecken bleiben. Somit benötigen die Agenten eine gemischte Strategie, um dieses Spiel ordnungsgemäß zu absolvieren: zufällig zwischen Vorwärtsbewegen und Stehenbleiben auswählen. Dies ist genau das, was in Ethernet-Netzwerken geschieht, um Paketkollisionen aufzulösen.

Als Nächstes betrachten wir eine sehr einfache Variante von Poker. Der Kartenstapel hat nur vier Karten, zwei Asse und zwei Könige. An jeden Spieler wird eine Karte ausgeteilt. Der erste Spieler hat dann die Option, den Spieleinsatz des Spiels von 1 Punkt auf 2 zu erhöhen oder zu *schieben* („check“). Wenn Spieler 1 schiebt, ist das Spiel beendet. Erhöht er, kann Spieler 2 *mitgehen* („call“) und damit akzeptieren, dass das Spiel nun 2 Punkte wert ist, oder *aussteigen* („fold“) und damit auf den 1 Punkt verzichten. Endet das Spiel nicht mit einem Aussteigen, hängt die Auszahlung von den Karten ab: Sie ist null für beide Spieler, wenn sie die gleiche Karte haben; andernfalls zahlt der Spieler mit dem König die Einsätze an den Spieler mit dem Ass.

▶ Abbildung 17.13 zeigt den Extensivform-Baum für dieses Spiel. Nichtterminalzustände sind als Kreise dargestellt. Sie enthalten die Nummer des Spielers, der am Zug ist; Spieler 0 ist *Zufall*. Jede Aktion ist durch einen Pfeil mit einer Beschriftung gekennzeichnet, die *erhöhen*, *schieben*, *mitgehen* und *aussteigen* entsprechen und für *Zufall* die vier möglichen Kartenverteilungen angeben (wobei „AK“ bedeutet, dass Spieler 1 ein Ass und Spieler 2 einen König erhält). Terminalzustände erscheinen als Rechtecke, die mit den Auszahlungen für Spieler 1 und Spieler 2 beschriftet sind. Die Informationsbezirke sind als gestrichelte Kästchen mit Beschriftungen dargestellt; zum Beispiel ist $I_{1,1}$ der Informationsbezirk, in dem Spieler 1 am Zug ist, und er weiß, dass er ein Ass hält (er weiß aber nicht, welche Hand Spieler 2 hat). Im Informationsbezirk $I_{2,1}$ ist Spieler 2 an der Reihe und weiß, dass er ein Ass hält und dass Spieler 1 erhöht hat, er weiß aber nicht, welche Karte Spieler 1 in der Hand hat. (Da sich auf Papier nur zweidimensional zeichnen lässt, ist dieser Informationsbezirk in zwei Kästen, statt in einem dargestellt.)

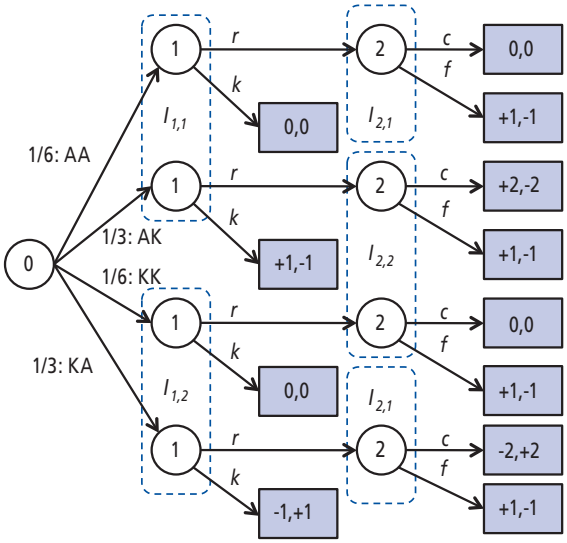


Abbildung 17.13: Extensivform einer vereinfachten Version von Poker.

Um ein extensives Spiel zu lösen, kann man es in ein Normalform-Spiel konvertieren. Wie Sie wissen, ist die Normalform eine Matrix, bei der jede Zeile mit einer reinen Strategie für Spieler 1 und jede Spalte mit einer reinen Strategie für Spieler 2 beschriftet ist. In einem extensiven Spiel entspricht eine reine Strategie für Spieler *i* einer Aktion für jeden Informationsbezirk, der mit diesem Spieler zu tun hat. So ist in Abbildung 17.13 die eine reine Strategie für Spieler 1 „erhöhen, wenn in $I_{1,\,1}$ (d.h., wenn ich ein Ass habe), und schieben, wenn in $I_{1,\,2}$ (wenn ich einen König habe)“. In der unten stehenden Auszahlungsmatrix ist diese Strategie mit *rk* bezeichnet. Analog bedeutet Strategie *cf* für Spieler 2, „schieben, wenn ich ein Ass habe, und aussteigen, wenn ich einen König habe“. Da es sich um ein Nullsummenspiel handelt, gibt die folgende Matrix nur die Auszahlung für Spieler 1 an; Spieler 2 erhält immer die entgegengesetzte Auszahlung:

	2:cc	2:cf	2:ff	2:fc
1:rr	0	-1/6	1	7/6
1:kr	-1/3	-1/6	5/6	2/3
1:rk	1/3	**0**	1/6	1/2
1:kk	0	**0**	0	0

Dieses Spiel ist so einfach, dass es zwei Gleichgewichte für reine Strategie besitzt, die in der Matrix fett geschrieben sind: *cf* für Spieler 2 und *rk* oder *kk* für Spieler 1. Im Allgemeinen aber können wir extensive Spiele lösen, indem wir sie in die Normalform konvertieren und dann eine Lösung (normalerweise unter Verwendung einer gemischten Strategie) mithilfe standardmäßiger linearer Programmiermethoden suchen. Zumindest funktioniert das in der Theorie. Doch wenn es für einen Spieler *I* Informationsbezirke und *a* Aktionen pro Bezirk gibt, wird dieser Spieler a^I reine Strategien haben. Mit anderen Worten wächst die Größe der Normalformmatrix exponentiell mit der Anzahl der

Informationsbezirke, sodass der Ansatz in der Praxis nur für sehr kleine Spielbäume in der Größenordnung von einigen Dutzend Zuständen funktioniert. Bei einem Spiel wie der Pokervariante Texas hold'em ist dieser Ansatz vollkommen unbrauchbar.

Was sind die Alternativen? *Kapitel 5* hat gezeigt, wie Alpha-Beta-Suche bei Spielen mit perfekter Information auch mit riesigen Spielbäumen klar kommt, indem die Bäume inkrementell generiert, bestimmte Zweige gekürzt und Nichtterminalknoten heuristisch ausgewertet werden. Doch dieses Konzept ist aus zwei Gründen für Spiele mit unvollständiger Information wenig geeignet: Erstens ist es schwerer zu kürzen, da wir gemischte Strategien betrachten müssen, die mehrere Zweige kombinieren, und keine reine Strategie, die immer den besten Zweig auswählt. Zweitens ist es schwerer, einen Nichtterminalknoten heuristisch auszuwerten, da wir es mit Informationsmengen und keinen einzelnen Zuständen zu tun haben.

Koller et al. (1996) kommen mit einer alternativen Darstellung von extensiven Spielen zu Hilfe: Die sogenannte **Sequenzform** ist nur in der Größe des Baumes linear und nicht exponentiell. Anstatt Strategien zu repräsentieren, stellt sie Pfade durch den Baum dar, wobei die Anzahl der Pfade gleich der Anzahl der Terminalknoten ist. Auch hier lassen sich wieder die Standardverfahren der linearen Programmierung auf diese Darstellung anwenden. Das resultierende System kann Pokervarianten mit 25.000 Zuständen in ein, zwei Minuten lösen. Dies ist eine exponentielle Beschleunigung gegenüber dem Normalformansatz, doch reicht sie bei Weitem nicht für die Verarbeitung von vollständigem Poker mit 10^{18} Zuständen.

Wenn wir die 10^{18} Zustände nicht in den Griff bekommen, ist es vielleicht möglich, das Problem zu vereinfachen, indem das Spiel in eine einfachere Form überführt wird. Wenn ich zum Beispiel ein Ass halte und die Wahrscheinlichkeit betrachte, dass ich mit der nächsten Karte zu einem Paar Asse komme, ist mir die Farbe der nächsten Karte egal – jede Farbe ist gleich gut geeignet. Es liegt also nahe, eine **Abstraktion** des Spieles zu bilden, die Farben ignoriert. Der resultierende Spielbaum wird dadurch um einen Faktor von $4! = 24$ kleiner. Angenommen, ich kann dieses kleinere Spiel lösen; wie verhält sich dann die Lösung dieses Spieles zum ursprünglichen Spiel? Wenn kein Spieler auf einen Flush geht (oder dahingehend blufft), spielen die Farben für keinen Spieler eine Rolle und die Lösung für die Abstraktion wird auch eine Lösung für das ursprüngliche Spiel sein. Wenn jedoch ein Spieler auf einen Flush aus ist, liefert die Abstraktion nur eine angenäherte Lösung (wobei sich aber Fehlergrenzen berechnen lassen).

Es gibt viele Einsatzfälle für Abstraktionen. Wenn ich zum Beispiel an einem Punkt im Spiel, wo jeder Spieler zwei Karten hat, ein Paar Damen halte, könnte man die Hände der anderen Spieler in drei Klassen abstrahieren: *besser* (nur ein Paar Könige oder ein Paar Asse), *gleich* (ein Paar Damen) oder *schlechter* (alles andere). Diese Abstraktion könnte jedoch zu grob sein. Eine bessere Abstraktion wäre es, die Kategorie *schlechter* beispielsweise in *mittleres Paar* (Neunen bis Buben), *niedriges Paar* und *kein Paar* zu gliedern. Diese Beispiele sind Abstraktionen von Zuständen. Es ist auch möglich, Aktionen zu abstrahieren. Anstatt zum Beispiel eine Wettaktion für jede Ganzzahl von 1 bis 1000 vorzusehen, könnten wir die Wetten auf 10^0, 10^1, 10^2 und 10^3 einschränken. Oder wir könnten eine der Wettrunden insgesamt auslassen. Außerdem können wir über Zufallsknoten abstrahieren, indem wir nur eine Teilmenge der möglichen Austeilvarianten betrachten. Dies ist gleichbedeutend mit der Rollout-Technik, die in Go-Programmen verwendet wird. Bringt man alle diese Abstraktionen zusammen, lassen sich die 10^{18} Zustände von Poker auf 10^7 Zustände reduzieren – eine Größe, die mit den derzeitigen Techniken beherrschbar ist.

Pokerprogramme, die auf diesem Ansatz beruhen, können problemlos Einsteiger und auch fortgeschrittenere menschliche Spieler besiegen, sind aber noch nicht auf der Ebene von Meisterspielern. Ein Teil des Problems besteht darin, dass die Lösung, die diese Programme annähern – die Gleichgewichtslösung – nur gegen einen Gegner optimal ist, der ebenfalls die Gleichgewichtsstrategie spielt. Gegen fehlbare menschliche Spieler ist es wichtig, die Abweichung von der Gleichgewichtsstrategie eines Gegners auszunutzen. Wie GautamRao, der weltweit führende Pokerspieler (alias „The Count"), sagte (Billings et al. 2003): „Sie haben ein sehr starkes Programm. Wenn Sie noch eine Modellierung für den Gegner hinzufügen, wird es jedermann schlagen." Allerdings bleiben gute Modelle der menschlichen Fehlbarkeit schwer zu fassen.

In gewisser Hinsicht ist die extensive Spielform eine der vollständigsten Darstellungen, die wir bisher kennengelernt haben: Sie ist für partiell beobachtbare, Multiagenten-, stochastische, sequentielle, dynamische Umgebungen geeignet – die meisten der schwierigen Fälle auf der Liste der Umgebungseigenschaften in *Abschnitt 2.3.2*. Allerdings gibt es zwei Einschränkungen der Spieltheorie. Erstens kommt sie nicht gut mit stetigen Zuständen und Aktionen klar (auch wenn es einige Erweiterungen für den stetigen Fall gibt; so bedient sich die Theorie des **Cournot-Wettbewerbes** der Spieltheorie, um Probleme zu lösen, bei denen zwei Firmen Preise für ihre Produkte aus einem kontinuierlichen Raum auswählen.) Zweitens geht die Spieltheorie davon aus, dass das Spiel bekannt ist. Teile des Spieles lassen sich für bestimmte Spieler als nicht beobachtbar spezifizieren, doch muss bekannt sein, welche Teile nicht beobachtbar sind. In den Fällen, in denen die Spieler mit der Zeit die unbekannte Struktur des Spieles lernen, beginnt das Modell zu versagen. Wir untersuchen nun die einzelnen Quellen der Unbestimmtheit und ob sie sich jeweils in der Spieltheorie darstellen lassen.

Aktionen: Es gibt keine einfache Möglichkeit, ein Spiel darzustellen, bei dem die Spieler die verfügbaren Aktionen erst herausfinden müssen. Nehmen Sie nur das Spiel zwischen Virus-Schreibern und Sicherheitsexperten. Ein Teil des Problems besteht darin, vorauszuahnen, welche Aktion die Virus-Schreiber als Nächstes ausprobieren.

Strategien: Die Spieltheorie eignet sich hervorragend, um das Konzept darzustellen, dass die Strategien der anderen Spieler anfangs unbekannt sind – solange wir annehmen, dass alle Agenten rational sind. Die Theorie an sich sagt nichts darüber, was zu tun ist, wenn die anderen Spieler nicht vollkommen rational handeln. Das Konzept des **Bayes-Nash-Gleichgewichtes** berücksichtigt diesen Aspekt zumindest teilweise: Es ist ein Gleichgewicht im Hinblick auf die A-priori-Wahrscheinlichkeitsverteilung eines Spielers über die Strategien der anderen Spieler – mit anderen Worten, es drückt die Glauben eines Spielers über die wahrscheinlichen Strategien der anderen Spieler aus.

Zufall: Wenn ein Spiel vom Wurf eines Würfels abhängt, lässt sich recht leicht ein Zufallsknoten mit gleichförmiger Verteilung über den Ergebnissen modellieren. Doch wie sieht es aus, wenn man von einem möglicherweise unfairen Würfel ausgehen muss? Wir können dies mit einem anderen Zufallsknoten darstellen, der höher im Baum liegt und zwei Zweige für „Würfel ist fair" und „Würfel ist unfair" besitzt, sodass die entsprechenden Knoten in jedem Zweig im selben Informationsbezirk liegen (d.h., die Spieler wissen nicht, ob der Würfel fair ist oder nicht). Und was ist, wenn wir vermuten, dass der andere Gegenspieler Bescheid weiß? Dann fügen wir einen weiteren Zufallsknoten hinzu, bei dem der eine Zweig den Fall darstellt, in dem der Gegner wissend ist, und der andere, in dem der Gegner nichts weiß.

Nutzen: Wie verhält es sich, wenn wir die Nutzen unserer Gegner nicht kennen? Auch dies lässt sich mit einem Zufallsknoten modellieren, sodass der andere Agent seine eigenen Nutzen in jedem Zweig kennt, wir jedoch nicht. Aber was ist, wenn wir unsere eigenen Nutzen nicht kennen? Woher weiß ich zum Beispiel, ob es rational ist, den Chefsalat zu bestellen, wenn ich nicht weiß, wie gut er mir schmecken wird? Diesen Sachverhalt können wir mit einem weiteren Zufallsknoten modellieren, der eine nicht beobachtbare „inhärente Qualität" des Salates spezifiziert.

Die Spieltheorie eignet sich demnach für die Darstellung der meisten Quellen von Unsicherheit – jedoch zum Preis einer Verdopplung der Baumgröße, wenn wir einen weiteren Knoten hinzufügen; eine Eigenschaft, die schnell zu nicht mehr handhabbaren Baumgrößen führt. Aufgrund dieser und anderer Probleme wurde die Spieltheorie hauptsächlich verwendet, um Umgebungen zu *analysieren*, die sich in einem Gleichgewicht befinden, anstatt Agenten innerhalb einer Umgebung zu *steuern*. Wir werden gleich sehen, wie dies beitragen kann, Umgebungen zu *entwerfen*.

17.6 Mechanismenentwurf

Im vorigen Abschnitt haben wir uns mit der Frage beschäftigt, wie die rationale Strategie für ein gegebenes Spiel aussieht. In diesem Abschnitt fragen wir, welches Spiel wir entwerfen sollen, wenn wir von rationalen Agenten ausgehen. Genauer gesagt, wir wollen ein Spiel entwerfen, dessen Lösungen, die darin bestehen, dass jeder Agent seine eigene rationale Strategie verfolgt, zur Maximierung einer globalen Nutzenfunktion führen. Dieses Problem wird auch als **Mechanismenentwurf** bezeichnet, manchmal auch als **inverse Spieltheorie**. Der Mechanismenentwurf ist für die Wirtschafts- und Politikwissenschaften von Bedeutung. Das Einmaleins des Kapitalismus besagt, dass der Wohlstand einer Gesellschaft zunimmt, wenn jeder versucht, reich zu werden. Doch die Beispiele, die wir hier besprechen, zeigen, dass geeigneter Mechanismenentwurf notwendig ist, um sich von der unsichtbaren Hand führen zu lassen. Für Sammlungen von Agenten erlaubt es der Mechanismenentwurf, intelligente Systeme aus einer Sammlung eingeschränkter Systeme zu erstellen – selbst wenn diese nicht kooperativ sind –, ähnlich wie Teams von mehreren Menschen Ziele erreichen können, die für jeden Einzelnen unerreichbar sind.

Beispiele für den Mechanismenentwurf sind unter anderem die Versteigerung billiger Flugtickets, das Routing von TCP-Paketen zwischen Computern, die Entscheidung, wie medizinische Ausstattungen auf Krankenhäuser verteilt werden, oder die Entscheidungen, wie Roboterfußballspieler mit ihren Teamkollegen zusammenarbeiten. In den 1990er Jahren gelangte der Mechanismenentwurf über seinen Ruf als rein akademisches Forschungsthema hinaus, als mehrere Nationen, die Lizenzen zur Funkübertragung in verschiedenen Frequenzbändern versteigern wollten, viele hundert Millionen Dollar potenziellen Gewinns verloren, weil ein schlechter Mechanismenentwurf vorlag. Formal ausgedrückt, besteht ein **Mechanismus** aus (1) einer Sprache für die Beschreibung der (möglicherweise unendlichen) Menge der erlaubten Strategien, die die Agenten anwenden können, (2) einem speziellen Agenten, dem sogenannten **Center**, das Berichte von Strategieauswahlen von den Agenten im Spiel sammelt, und (3) einer Ergebnisregel, die allen Agenten bekannt ist, die das Center verwendet, um die Auszahlungen für jeden Agenten zu ermitteln, wobei ihre Strategieauswahlen gegeben sind.

17.6.1 Auktionen

Betrachten wir als Erstes **Auktionen**. Eine Auktion ist ein Mechanismus, um Waren an Mitglieder eines Pools von Bietern zu verkaufen. Der Einfachheit halber konzentrieren wir uns hier auf Auktionen, in denen nur ein einziger Artikel zum Verkauf steht. Jeder Bieter i hat einen Nutzenwert v_i, wenn er den Artikel erhält. In manchen Fällen hat jeder Bieter einen **privaten Wert** für den Artikel. Zum Beispiel wurde auf eBay als allererster Artikel ein defekter Laserpointer für \$14,83 an einen Sammler von defekten Laserpointern verkauft. Somit wissen wir, dass der Sammler einen $v_i \geq$ \$14,83 hat, während bei den meisten anderen Leuten $v_j << \$14{,}83$ sein dürfte. In anderen Fällen wie etwa dem Versteigern von Bohrrechten für ein Ölfeld hat der Artikel einen gemeinsamen Wert – das Feld produziert einen bestimmten Geldbetrag X und alle Bieter bewerten einen Dollar gleich –, doch besteht die Unsicherheit, wie groß der tatsächliche Wert von X ist. Verschiedene Bieter verfügen über unterschiedliche Informationen und schätzen folglich den wahren Wert des Artikels verschieden ein. In jedem Fall enden Bieter bei ihrem eigenen v_i. Für einen gegebenen v_i erhält jeder Bieter die Chance, zu einem geeigneten Zeitpunkt oder mehrmals in der Aktion ein Gebot b_i abzugeben. Das höchste Gebot b_{max} gewinnt die Ware, wobei aber der zu zahlende Preis nicht b_{max} sein muss – das hängt vom Mechanismenentwurf ab.

Der bekannteste Auktionsmechanismus ist die **Auktion mit ansteigendem Gebot**[8] oder **Englische Auktion**, wobei das Center zunächst ein Mindestgebot (oder Auktionslimit) b_{min} ausruft. Ist ein Bieter bereit, diesen Preis zu zahlen, gibt das Center einen Preis von $b_{min} + d$ mit einem bestimmten Inkrement d an und setzt von diesem Gebot aus fort. Die Auktion endet, wenn niemand bereit ist, mehr zu zahlen; dann gewinnt der letzte Bieter die Ware, wofür er den von ihm gebotenen Preis bezahlt.

Woher wissen wir, ob dies ein guter Mechanismus ist? Ein Ziel besteht darin, den erwarteten Erlös für den Verkäufer zu maximieren. Ein anderes Ziel ist es, ein Konzept des globalen Nutzens zu maximieren. Diese Ziele überschneiden sich in gewissem Umfang, da ein Aspekt bei der Maximierung des globalen Nutzens sicherstellen soll, dass der Gewinner der Auktion der Agent ist, dem die Ware am meisten wert ist (und der folglich am meisten zu zahlen bereit ist). Wir sagen, eine Auktion ist **effizient**, wenn die Waren an den Agenten gehen, der sie am meisten schätzt. Die Auktion mit steigendem Gebot ist normalerweise sowohl effizient als auch Erlös maximierend, doch wenn das Mindestgebot zu hoch angesetzt wird, bietet der Bieter, der die Ware am meisten schätzt, möglicherweise nicht, und wenn das Auktionslimit zu tief liegt, entgeht dem Verkäufer ein gewisser Nettoerlös.

Die wohl wichtigsten Aufgaben eines Auktionsmechanismus bestehen darin, eine ausreichend große Anzahl von Bietern zu ermuntern, in das Spiel einzusteigen, und sie an **Preisabsprachen** zu hindern. Eine Preisabsprache oder **Kollusion** ist eine unfaire oder unzulässige Übereinkunft von zwei oder mehreren Bietern, um Preise zu manipulieren. Sie kann innerhalb der Regeln des Mechanismus insgeheim hinter verschlossenen Türen oder stillschweigend stattfinden.

Zum Beispiel hat Deutschland im Jahr 1999 zehn Blöcke von Mobilfunkfrequenzen mit einer simultanen Auktion versteigert (bei der die Gebote für alle zehn Blöcke gleichzeitig abgegeben wurden). Dabei galt die Regel, dass jedes Gebot mindestens 10% über dem vorherigen Gebot eines Blockes liegen musste. Es gab nur zwei glaubwürdige Bieter und

8 Das Wort „Auktion“ kommt vom lateinischen *augere* – erhöhen.

der erste, Mannesmann, gab das Gebot von 20 Millionen D-Mark für die Blöcke 1 bis 5 und 18,18 Millionen für die Blöcke 6 bis 10 ab. Warum gerade 18,18 Millionen? Einer der Manager von T-Mobile sagte, sie „interpretierten das erste Gebot von Mannesmann als Offerte“. Beide Parteien konnten ausrechnen, dass eine 10%ige Erhöhung auf 18,18 Millionen den Preis von 19,99 Millionen ergibt. Somit wurde das Angebot von Mannesmann wie folgt interpretiert: „Wir können jeder die Hälfte der Blöcke für 20 Millionen bekommen; verderben wir das nicht, indem wir die Preise immer höher treiben.“ In der Tat bot T-Mobile 20 Millionen für die Blöcke 6 bis 10 und damit war das Bieten beendet. Die deutsche Regierung erhielt weniger als erwartet, da die beiden Mitbewerber den Bietmechanismus für eine stillschweigende Übereinkunft ausnutzen konnten, um sich keine Konkurrenz zu machen. Vom Standpunkt der Regierung aus wäre das Ergebnis mit einer der folgenden Änderungen am Mechanismus besser ausgefallen: höheres Mindestgebot; verdeckte Abgabe der Gebote (Sealed-Bid First-Price), sodass die Mitbewerber nicht über ihre Gebote kommunizieren können; oder Anreize, um einen dritten Bieter zu motivieren. Vielleicht war die 10%-Regel ein Fehler im Entwurf des Mechanismus, da sie die genaue Signalisierung von Mannesmann an T-Mobile erleichtert hat.

Im Allgemeinen ist es sowohl für den Verkäufer als auch die globale Nutzenfunktion besser, wenn es mehrere Bieter gibt, obwohl der globale Nutzen auch leiden kann, wenn man die Kosten für die verschwendete Zeit der Bieter, die keine Chance auf einen Gewinn haben, einrechnet. Um mehr Bieter zu ermutigen, könnte man beispielsweise den Mechanismus einfacher gestalten. Denn wenn der Aufwand für Recherchen oder Berechnungen auf Seiten der Bieter zu groß wird, entscheiden sie möglicherweise, ihr Geld in anderen Projekten anzulegen. Es ist also wünschenswert, dass die Bieter eine **dominante Strategie** haben. Wie Sie wissen, bedeutet „dominant“, dass die Strategie gegen alle anderen Strategien funktioniert, was wiederum heißt, dass ein Agent diese Strategie ohne Rücksicht auf die anderen Strategien annehmen kann. Ein Agent mit einer dominanten Strategie kann einfach bieten, ohne Zeit damit zu verschwenden, die möglichen Strategien anderer Agenten zu betrachten. Ein Mechanismus, bei dem Agenten eine dominante Strategie haben, wird als **strategiesicherer** Mechanismus bezeichnet. Wenn die Strategie, wie es normalerweise der Fall ist, die Bieter einbezieht, die ihren wahren Wert v_i offenlegen, spricht man von einer **Wahrheit enthüllenden** oder **wahrheitsgemäßen** Auktion. Ebenfalls gebräuchlich ist der Begriff **anreizkompatibel**. Das **Offenbarungsprinzip** besagt, dass sich jeder Mechanismus in einen gleichwertigen wahrheitsgemäßen Mechanismus überführen lässt, sodass es zum Mechanismenentwurf gehört, diese äquivalenten Mechanismen zu finden.

Es zeigt sich, dass die Auktion mit ansteigendem Gebot die meisten der wünschenswerten Eigenschaften aufweist. Der Bieter mit dem höchsten Wert v_i erhält die Ware zu einem Preis von $b_o + d$, wobei b_o das höchste Gebot unter allen anderen Agenten und d das Inkrement des Auktionators ist.[9] Bieter verwenden eine einfache dominante Strategie: bieten, solange der aktuelle Preis unterhalb ihres v_i liegt. Der Mechanismus ist nicht wirklich wahrheitsgemäß, da der gewinnende Bieter nur offenlegt, dass sein $v_i \geq b_o + d$ ist; wir haben eine untere Grenze bezüglich v_i, jedoch keinen genauen Betrag.

Bei der Auktion mit ansteigendem Gebot ist es (vom Standpunkt des Verkäufers) nachteilig, dass sie demotivierend für den Wettbewerb wirken kann. Nehmen Sie an, in

9 Es besteht eine kleine Wahrscheinlichkeit, dass der Spieler mit dem höchsten v_i die Ware nicht erhält, nämlich wenn $b_o < v_i < b_o + d$. Die Wahrscheinlichkeit, dass dies passiert, kann beliebig klein gemacht werden, indem das Inkrement d verringert wird.

einer Auktion für einen Mobilfunkfrequenzblock gibt es eine begünstigte Firma, bei der sich alle einig sind, dass sie vorhandene Kunden und die Infrastruktur zu ihrem Vorteil nutzen und somit einen größeren Gewinn als jeder andere Teilnehmer erzielen kann. Potenzielle Mitbewerber können erkennen, dass sie in einer Auktion mit steigendem Gebot keine Chance haben, da die begünstigte Firma immer höher bieten kann. Somit nehmen die Mitbewerber möglicherweise gar nicht teil und die begünstigte Firma gewinnt zum Mindestgebot.

Eine weitere negative Eigenschaft der Englischen Auktion sind die hohen Kommunikationskosten. Entweder findet die Auktion in einem Raum statt oder alle Bieter müssen über sichere Hochgeschwindigkeits-Kommunikationsleitungen verfügen. In jedem Fall benötigen sie ein Zeitkontingent, um mehrere Runden beim Bieten mitgehen zu können. Ein alternativer Mechanismus, für den sehr viel weniger Kommunikation anfällt, ist die **Auktion mit versiegelten Geboten**. Hier macht jeder Bieter ein einziges Gebot und teilt es dem Auktionator mit, ohne dass es die anderen Bieter sehen. Bei diesem Mechanismus gibt es keine einfache dominante Strategie mehr. Wenn Ihr Wert gleich v_i ist und Sie glauben, dass das Maximum der Gebote aller anderen Agenten gleich b_o ist, sollten Sie $b_o + \epsilon - $ für ein kleines $\epsilon - $ bieten, wenn dies kleiner als v_i ist. Somit hängt Ihr Gebot von Ihrer Einschätzung der Gebote der anderen Agenten ab, was mehr Aufwand von Ihrer Seite bedingt. Beachten Sie auch, dass der Agent mit dem höchsten v_i die Auktion nicht unbedingt gewinnt. Dies wird durch die Tatsache ausgeglichen, dass die Auktion mehr Wettbewerb bietet, wodurch die Schwelle gegen einen begünstigten Bieter gesenkt wird.

Eine kleine Änderung im Mechanismus für die Auktion mit versiegelten Geboten führt zur **Auktion mit versiegelten Geboten und zweithöchstem Preis**, auch als **Vickrey-Auktion**[10] bezeichnet. Bei diesen Auktionen zahlt der Gewinner den Preis des *zweithöchsten* Gebotes und nicht sein eigenes Gebot. Diese einfache Änderung eliminiert vollständig die komplexen Überlegungen, die bei Standardauktionen mit versiegeltem Gebot anfallen, weil die dominante Strategie jetzt ist, v_i zu bieten; der Mechanismus enthüllt die Wahrheit. Der Nutzen des Agenten i in Bezug auf sein Gebot b_i, seinen Wert v_i und das beste Gebot unter den anderen Spielern, b_m, ist:

$$u_i = \begin{cases} (v_i - b_o) & \text{wenn } b_i > b_o \\ 0 & \text{andernfalls.} \end{cases}$$

Um zu erkennen, dass $b_i = v_i$ eine dominante Strategie ist, beachten Sie, dass, wenn $(v_i - b_m)$ positiv ist, jedes Gebot, das die Auktion gewinnt, optimal ist und insbesondere das Gebot v_i die Auktion gewinnt. Ist dagegen $(v_i - b_o)$ negativ, ist jedes Gebot, das die Auktion verliert, optimal und insbesondere verliert das Gebot v_i die Auktion. Das Bieten von v_i ist also für alle möglichen Werte von b_o optimal und tatsächlich ist v_i das einzige Gebot, das diese Eigenschaft aufweist. Aufgrund der Einfachheit und des minimalen Rechenaufwandes für Verkäufer und Bieter wird die Vickrey-Auktion beim Aufbau verteilter KI-Systeme häufig eingesetzt. Internet-Suchmaschinen führen über eine Milliarde Auktionen pro Tag durch, um zusammen mit ihren Suchergebnissen auch Werbung zu verkaufen, und Online-Auktionssites setzen 100 Milliarden Dollar pro Jahr in Waren um, wobei alle Varianten der Vickrey-Auktion zum Einsatz kommen. Der erwartete Wert für den Verkäufer ist b_o, was der gleiche erwartete Gewinn ist wie die Grenze bei der

10 Benannt nach William Vickrey (1914–1996), der den Nobelpreis für Wirtschaftswissenschaften im Jahr 1996 für seine Arbeit gewann und drei Tage später an einem Herzinfarkt starb.

Englischen Auktion, wenn das Inkrement d gegen 0 geht. Dies ist eigentlich ein sehr allgemeines Ergebnis: Der **Satz über die Erlösäquivalenz** besagt, dass mit Ausnahme kleinerer Vorbehalte jeder Auktionsmechanismus, bei dem risikoneutrale Bieter Werte v_i haben, die nur ihnen selbst bekannt sind (die aber die Wahrscheinlichkeitsverteilung von denen kennen, deren Werte gesampelt werden), den gleichen erwarteten Erlös erzielt. Dieses Prinzip bedeutet, dass die verschiedenartigen Mechanismen nicht auf der Grundlage von Erlöserzeugung wetteifern, sondern auf anderen Qualitäten beruhen.

Obwohl die Zweipreisauktion wahrheitsgemäß verläuft, zeigt sich, dass die Erweiterung des Konzeptes auf mehrere Waren und die Verwendung der Nächstpreisauktion nicht wahrheitsgemäß ist. Viele Internet-Suchmaschinen verwenden einen Mechanismus, bei dem die Auktion k Werbeeinblendungen (Slots) auf einer Seite bringt. Der höchste Bieter gewinnt den höchsten Spot, der zweithöchste Bieter den zweiten Spot usw. Jeder Gewinner bezahlt den Preis, der vom nächst niedrigeren Bieter geboten wurde, wobei diese Zahlung nur erfolgt, wenn der Suchende tatsächlich auf die Werbung klickt. Die obersten Slots gelten als wertvoller, weil sie wahrscheinlicher wahrgenommen und angeklickt werden. Stellen Sie sich vor, dass drei Bieter b_1, b_2 und b_3 für einen Klick Bewertungen von $v_1 = 200$, $v_2 = 180$ und $v_3 = 100$ erhalten und dass $k = 2$ Slots verfügbar sind. Außerdem ist bekannt, dass der oberste Spot in 5% der Zeit und der untere Spot zu 2% angeklickt wird. Wenn alle Bieter wahrheitsgemäß bieten, gewinnt b_1 den obersten Slot, zahlt 180 und hat einen erwarteten Erlös von $(200 - 180) \times 0{,}05 = 1$. Der zweite Slot geht an b_2. Allerdings kann b_1 sehen, dass er bei einem Gebot im Bereich 101 bis 179 den obersten Slot an b_2 abgeben, den zweiten Slot gewinnen und einen erwarteten Erlös von $(200 - 100) \times 0{,}02 = 2$ erzielen würde. Folglich kann b_1 in diesem konkreten Fall seinen erwarteten Erlös verdoppeln, indem er weniger als seinen wahren Wert bietet. Im Allgemeinen müssen Bieter in dieser Multislot-Auktion eine Menge Energie aufwenden, um die Gebote der anderen zu analysieren und deren beste Strategie zu ermitteln; es gibt keine einfache dominante Strategie. Aggarwal et al. (2006) zeigen, dass es einen eindeutigen wahrheitsgemäßen Auktionsmechanismus für dieses Multislot-Problem gibt, bei dem der Gewinner von Slot j den vollen Preis für Slot j lediglich für diejenigen zusätzlichen Klicks bezahlt, die bei Slot j und nicht bei Slot $j + 1$ verfügbar sind. Für die restlichen Klicks bezahlt der Gewinner den Preis für den unteren Slot. In unserem Beispiel würde b_1 wahrheitsgemäß 200 bieten und 180 für die zusätzlichen $0{,}05 - 0{,}02 = 0{,}03$ Klicks im oberen Slot zahlen, die Kosten von 100 für den unteren Slot jedoch nur für die restlichen 0,02 Klicks bezahlen. Der gesamte Erlös für b_1 wäre somit $(200 - 180) \times 0{,}03 + (200 - 100) \times 0{,}02 = 2{,}6$.

Innerhalb der KI kommen Auktionen beispielsweise auch ins Spiel, wenn eine Sammlung von Agenten entscheidet, ob sie an einem gemeinsamen Plan kooperieren. Hunsberger und Grosz (2000) zeigen, dass sich dies effizient mit einer Auktion realisieren lässt, in der die Agenten für Rollen im gemeinsamen Plan bieten.

17.6.2 Gemeinsame Güter

Kommen wir nun zu einem anderen Spieltyp, in dem Länder ihre Strategie für die Kontrolle der Luftverschmutzung festlegen. Jedes Land hat eine Wahl: Es kann die Verschmutzung zu den Kosten von -10 Punkten für die Implementierung der notwendigen Änderungen reduzieren oder weiterhin die Luft verschmutzen, was ihm einen Nettonutzen von -5 beschert (wegen zusätzlicher Gesundheitskosten usw.) und außerdem -1 Punkte zu jedem anderen Land beisteuert (da sich die Luft über die Länder hinweg verteilt).

Offensichtlich lautet die dominante Strategie für jedes Land „weiterhin verschmutzen“, doch wenn 100 Länder dieser Strategie folgen, erhält jedes Land einen Gesamtnutzen von −104, während sie einen Nutzen von −10 hätten, wenn sie die Luftverschmutzung reduzieren würden. Diese Situation wird als **Tragödie des Allgemeingutes** oder **Allmendeklemme** bezeichnet: Wenn niemand für die Nutzung einer gemeinsamen Ressource bezahlen muss, wird sie irgendwann so ausgebeutet, dass sich für alle Agenten ein geringerer Gesamtnutzen ergibt. Das lässt sich mit dem Gefangenendilemma vergleichen: Es gibt für das Spiel eine andere Lösung, die für alle Parteien besser ist, doch scheint es für rationale Agenten keinen Weg zu geben, zu dieser Lösung zu gelangen.

Das Standardkonzept für den Umgang mit der Tragödie des Allgemeingutes ist es, den Mechanismus so zu ändern, dass man jeden Agenten für die Nutzung des Allgemeingutes mit einer Gebühr belegt. Allgemeiner ausgedrückt müssen wir sicherstellen, alle **externen Effekte** – Wirkungen auf den globalen Nutzen, die in den Transaktionen der einzelnen Agenten nicht erkannt werden – explizit zu machen. Das Schwierige dabei ist eine korrekte Festsetzung der Preise. Im Grenzbereich läuft dieser Ansatz darauf hinaus, einen Mechanismus zu erzeugen, der von jedem Agenten praktisch fordert, den globalen Nutzen zu maximieren, aber dies mit einer lokalen Entscheidung realisieren kann. Für dieses Beispiel wäre eine Kohlendioxidsteuer ein Mechanismus, der die Verwendung des Allgemeingutes mit einer Gebühr belegt, was bei richtiger Umsetzung den globalen Nutzen maximiert.

Als letztes Beispiel betrachten wir das Problem der Zuweisung bestimmter gemeinsamer Güter. Angenommen, eine Stadt entscheidet, einige kostenlose W-LAN-Transceiver für das Internet zu installieren. Allerdings ist die Anzahl der von der Stadt finanzierbaren Transceiver geringer als die Anzahl der Wohngebiete, die daran interessiert sind. Die Stadt möchte die Güter effizient zuweisen, und zwar an die Wohngebiete, die sie am meisten schätzen. Das heißt, sie will den globalen Nutzen $V = \Sigma_i v_i$ maximieren. Das Problem dabei: Wenn man die Gemeinderäte fragt „wie hoch schätzen Sie dieses kostenlose Geschenk ein?“, werden alle dazu angestachelt sein, zu lügen und einen hohen Wert zu melden. Mit dem sogenannten **Vickrey-Clarke-Groves**- oder **VCG**-Mechanismus existiert nun ein Mechanismus, der als dominante Strategie für jeden Agenten wahrheitsgemäßes Bieten festlegt und der eine effiziente Zuweisung der Güter erreicht. Die Kunst besteht darin, dass jeder Agent eine Steuer zahlt, die dem Verlust im globalen Nutzen äquivalent ist. Dieser Verlust tritt auf, weil der Agent am Spiel teilnimmt. Der Mechanismus funktioniert wie folgt:

1. Das Center fordert jeden Agenten auf, seinen Wert für den Empfang eines Artikels zu melden. Dies werde b_i genannt.

2. Das Center weist die Güter einer Teilmenge der Bieter zu. Wir nennen diese Teilmenge A und schreiben $b_i(A)$ als Ergebnis für i unter dieser Zuweisung: b_i, wenn i zu A gehört (d.h., i ist ein Gewinner), und andernfalls 0. Das Center wählt A aus, um den gemeldeten Gesamtnutzen $B = \Sigma_i b_i(A)$ zu maximieren.

3. Das Center berechnet (für jedes i) die Summe der gemeldeten Nutzen für alle Gewinner mit Ausnahme von i. Wir notieren dies mit $B_{-1} = \Sigma_{j \neq i} b_j(A)$. Das Center berechnet auch (für jeden i) die Zuweisung, die den gesamten globalen Nutzen maximiert, wenn i nicht am Spiel teilnimmt; diese Summe nennen wir W_{-i}.

4. Jeder Agent i zahlt eine Steuer gleich $W_{-i} - B_{-i}$.

In diesem Beispiel bedeutet die VCG-Regel, dass jeder Gewinner eine Steuer gleich dem höchsten gemeldeten Wert unter den Verlierern zahlt. Wenn ich also meinen Wert mit 5 melde und dieser dazu führt, dass jemand mit dem Wert 2 eine Zuweisung verfehlt, bezahle ich eine Steuer von 2. Alle Gewinner sollten zufrieden sein, weil sie eine Steuer bezahlen, die geringer als ihr Wert ist, und alle Verlierer sind glücklich, weil sie die Waren geringer als die geforderte Steuer bewerten.

Wieso ist dieser Mechanismus wahrheitsgemäß? Betrachten Sie zuerst die Auszahlung an Agent i, die dem Wert abzüglich der Steuer entspricht, einen Artikel zu erhalten:

$$v_i(A) - (W_{-i} - B_{-i}). \tag{17.14}$$

Wir unterscheiden hier den wahren Nutzen des Agenten v_i von seinem gemeldeten Nutzen b_i (versuchen aber zu zeigen, dass $b_i = v_i$ eine dominante Strategie ist). Agent i weiß, dass das Center globalen Nutzen anhand der gemeldeten Werte

$$\sum_j b_j(A) = b_i(A) + \sum_{j \neq i} b_j(A)$$

maximiert, während Agent i möchte, dass das Center (17.14) maximiert, was sich als

$$v_i(A) + \sum_{j \neq i} b_j(A) - W_{-i}$$

neu formulieren lässt. Da der Agent i den Wert von W_{-i} nicht beeinflussen kann (da er nur von den anderen Agenten abhängt), bleibt i nichts anderes übrig, als den wahren Nutzen $b_i = v_i$ zu melden, damit das Center das optimiert, was i möchte.

Bibliografische und historische Hinweise

Richard Bellman entwickelte die Ideen, die dem modernen Ansatz für sequentielle Entscheidungsprobleme zugrunde liegen, während er bei der RAND Corporation arbeitete. Entsprechend seiner Autobiografie (Bellman, 1984) prägte er den spannenden Begriff „dynamische Programmierung“, um vor dem Verteidigungsminister Charles Wilson die Tatsache zu verbergen, dass sich seine Gruppe mit Mathematik beschäftigte. (Dies kann allerdings nicht ganz stimmen, da sein erster Artikel, in dem der Begriff auftaucht (Bellman, 1952), erschien, bevor Wilson 1953 zum Verteidigungsminister ernannt wurde.) Bellmans Buch *Dynamic Programming* (1957) gab dem neuen Gebiet eine solide Grundlage und führte die grundlegenden algorithmischen Konzepte ein. Die Doktorarbeit von Ron Howard (1960) führte die Taktik-Iteration ein, ebenso wie das Konzept eines durchschnittlichen Gewinns für die Lösung von Problemen mit unendlichen Horizonten. Bellman und Dreyfus (1962) zeigten mehrere zusätzliche Ergebnisse auf. Die modifizierte Taktik-Iteration geht auf van Nunen (1976) sowie Puterman und Shin (1978) zurück. Die asynchrone Taktik-Iteration wurde von Williams und Baird (1993) analysiert, die auch die Taktikverlustgrenze in Gleichung (17.9) bewiesen haben. Die Analyse der Verminderung im Hinblick auf stationäre Prioritäten geht auf Koopmans (1972) zurück. Die Arbeiten von Bertsekas (1987), Puterman (1994) sowie Bertsekas und Tsitsiklis (1996) geben eine strenge Einführung in sequentielle Entscheidungsprobleme. Papadimitriou und Tsitsiklis (1987) beschreiben Ergebnisse zur Rechenkomplexität von MEPs.

Die inspirierenden Arbeiten von Sutton (1988) und Watkins (1989) zu verstärkenden Lernmethoden zur Lösung von MEPs haben eine wichtige Rolle bei der Einführung von MEPs in der KI-Gemeinde gespielt, ebenso wie der spätere Überblick von Barto et al.

(1995). (Frühere Arbeiten von Werbos (1977) enthielten viele ähnliche Ideen, wurden aber nicht im selben Ausmaß übernommen.) Die Verbindung zwischen MEPs und KI-Planungsproblemen wurde als Erstes von Sven Koenig (1991) aufgegriffen, der zeigte, wie probabilistische STRIPS-Operatoren eine kompakte Repräsentation für Übergangsmodelle unterstützen (siehe auch Wellman (1990b)). Arbeiten von Dean et al. (1993) sowie Tash und Russell (1994) versuchten, die Kombinatorik großer Zustandsräume zu umgehen, indem sie einen begrenzten Suchhorizont und abstrakte Zustände verwendeten. Heuristiken, die auf dem Informationswert basieren, können verwendet werden, um Bereiche des Zustandsraumes auszuwählen, wo eine lokale Expansion des Horizonts zu einer wesentlichen Verbesserung der Entscheidungsqualität führt. Agenten, die diesen Ansatz verfolgen, können ihren Aufwand so anpassen, dass sie mit Zeitdruck zurechtkommen und einige interessante Verhalten erzeugen, wie etwa die Verwendung bekannter „ausgetretener Pfade“, um ihren Weg im Zustandsraum schnell zu finden, ohne an jeder Stelle die optimale Entscheidung erneut berechnen zu müssen.

Wie zu erwarten war, haben KI-Forscher MEPs in Richtung von ausdrucksstärkeren Darstellungen vorangetrieben, die sich an wesentlich größere Probleme anpassen lassen als die herkömmlichen atomaren Darstellungen, die auf Übergangsmatrizen basieren. Die Darstellung von Übergangsmodellen mit dynamischen Bayesschen Netzen war eine naheliegende Idee, doch die Arbeiten zu **faktorisierten MEPs** (Boutilier et al., 2000; Koller und Parr, 2000; Guestrin et al., 2003b) erweitern das Konzept auf strukturierte Darstellungen der Wertfunktion mit nachweislichen Verbesserungen in der Komplexität. **Relationale MEPs** (Boutilier et al., 2001; Guestrin et al., 2003a) gehen noch einen Schritt weiter, um mithilfe von strukturierten Darstellungen Domänen mit vielen verknüpften Objekten zu verarbeiten.

Die Beobachtung, dass ein partiell beobachtbares MEP unter Verwendung der Belief States in ein reguläres MEP umgewandelt werden kann, stammt von Astrom (1965) und Aoki (1965). Der erste vollständige Algorithmus für die genaue Lösung von partiell beobachtbaren Markov-Entscheidungsprozessen – im Wesentlichen der in diesem Kapitel vorgestellte Algorithmus für Wert-Iteration – wurde von Edward Sondik (1971) in seiner Doktorarbeit vorgestellt. (Ein späterer Artikel von Smallwood und Sondik (1973) enthält einige Fehler, ist jedoch verständlicher.) Lovejoy (1991) bietet einen Überblick über den aktuellen Stand der Arbeit zu partiell beobachtbaren MEPs. Der erste wichtige Beitrag innerhalb der KI war der Witness-Algorithmus (Cassandra et al., 1994; Kaelbling et al., 1998), eine verbesserte Version der Wert-Iteration bei partiell beobachtbaren MEPs. Schnell folgten weitere Algorithmen, unter anderem beispielsweise ein Ansatz von Hansen (1998), der eine Taktik inkrementell in Form eines endlichen Automaten konstruierte. Bei dieser Taktikdarstellung entspricht der Belief State direkt einem bestimmten Zustand im Automaten. Neuere Arbeiten in der KI haben sich auf Methoden der **punktbasierten** Wert-Iteration konzentriert, die bei jeder Iteration bedingte Pläne und α-Vektoren für eine endliche Menge von Belief States statt für den gesamten Glaubensraum generieren. Lovejoy (1991) schlug einen derartigen Algorithmus für ein festes Punktraster vor. Diesen Ansatz griff auch Bonet (2002) auf. Ein einflussreicher Artikel von Pineau et al. (2003) schlug vor, erreichbare Punkte zu generieren, indem Bahnen in einer etwas „gierigen“ (engl. greedy) Art simuliert werden; Spaan und Vlassis (2005) beobachten, dass man Pläne nur für eine kleine, zufällig ausgewählte Teilmenge von Punkten generieren muss, um die Pläne aus der vorherigen Iteration für alle Punkte in der Menge zu verbessern. Aktuelle punktbasierte Methoden – wie zum Beispiel punktbasierte Taktik-Iteration (Ji et al., 2007) –

können nahezu optimale Lösungen für partiell beobachtbare MEPs mit Tausenden von Zuständen generieren. Da partiell beobachtbare MEPs PSPACE-hart sind (Papadimitriou und Tsitsiklis, 1987), kann es für weitere Fortschritte notwendig sein, verschiedene Strukturarten innerhalb einer faktorisierten Darstellung auszunutzen.

Der Online-Ansatz – die Verwendung einer vorausschauenden Suche, um eine Aktion für den aktuellen Belief State auszuwählen – wurde zuerst von Satia und Lave (1973) studiert. Kearns et al. (2000) sowie Ng und Jordan (2000) haben Sampling bei Zufallsknoten analytisch untersucht. Die grundlegenden Konzepte für eine Agentenarchitektur unter Verwendung dynamischer Entscheidungsnetze wurden von Dean und Kanazawa (1989a) vorgelegt. Das Buch *Planning and Control* von Dean und Wellman (1991) geht weiter ins Detail und stellt Verbindungen zwischen DBN/DDN-Modellen und der klassischen Literatur zur Filterung in der Steuerungstheorie her. Tatman und Shachter (1990) haben gezeigt, wie dynamische Programmieralgorithmen auf DDN-Modelle angewendet werden können. Russell (1998) erklärt verschiedene Methoden, wie solche Agenten auf größere Systeme angewendet werden können, und zeigt mehrere offene Forschungsbereiche auf.

Die Wurzeln der Spieltheorie können bis zu den Vorschlägen von Christiaan Huygens und Gottfried Leibniz aus dem 17. Jahrhundert zurückverfolgt werden, die konkurrierende und kooperative menschliche Verhaltensweisen wissenschaftlich und mathematisch untersuchten. Im 19. Jahrhundert erzeugten verschiedene führende Wirtschaftswissenschaftler einfache mathematische Beispiele für die Analyse bestimmter Beispiele konkurrierender Situationen. Die ersten formalen Ergebnisse in der Spieltheorie stammen von Zermelo (1913) (der im Jahr zuvor eine Form der Minimax-Suche für Spiele vorgeschlagen hat, die jedoch nicht korrekt war). Emil Borel (1921) führte das Konzept der gemischten Strategie ein. John von Neumann (1928) bewies, dass jedes Zwei-Spieler-Nullsummenspiel ein Maximin-Gleichgewicht in gemischten Strategien sowie einen wohldefinierten Wert hat. Die Zusammenarbeit von Neumanns mit dem Wirtschaftswissenschaftler Oskar Morgenstern führte 1944 zur Veröffentlichung von *Theory of Games and Economic Behavior*, dem maßgeblichen Buch für die Spieltheorie. Die Veröffentlichung des Buches wurde durch die Papierknappheit während des Krieges verzögert, bis ein Mitglied der Familie Rockefeller sich persönlich für die Veröffentlichung einsetzte.

John Nash veröffentlichte 1950 im Alter von 21 Jahren seine Ideen zu den Gleichgewichten in allgemeinen (Nicht-Nullsummen-) Spielen. Seine Definition einer Gleichgewichtslösung wurde als Nash-Gleichgewicht bekannt, obwohl sie auf Arbeiten von Cournot (1838) basierte. Nach einer langen Verzögerung aufgrund der Schizophrenie, unter der er ab 1959 litt, erhielt Nash 1994 den Nobelpreis für Wirtschaftswissenschaften (zusammen mit Reinhart Selten und John Harsanyi). Das Bayes-Nash-Gleichgewicht wurde von Harsanyi (1967) beschrieben und von Kadane und Larkey (1982) diskutiert. Einige Aspekte der Verwendung der Spieltheorie für die Agentensteuerung werden in Binmore (1982) behandelt.

Das Gefangenendilemma wurde 1950 als Schularbeit von Albert W. Tucker erfunden (basierend auf einem Beispiel von Merrill Flood und Melvin Dresher) und von Axelrod (1985) und Poundstone (1993) ausführlich beschrieben. Wiederholte Spiele wurden von Luce und Raiffa (1957) beschrieben, Spiele mit partiellen Informationen von Kuhn (1953). Der erste praktische Algorithmus für sequentielle Spiele mit partiellen Informationen wurde in der KI von Koller et al. (1996) entwickelt. Die Arbeiten von Koller und Pfeffer (1997) sind eine lesenswerte Einführung in das Gebiet und beschreiben ein funktionierendes System für die Repräsentation und Lösung sequentieller Spiele.

Billings et al. (2003) beschreiben, wie sich ein Spielbaum durch Abstraktion auf eine Größe reduzieren lässt, sodass er mit der Technik von Koller gelöst werden kann. Bowling et al. (2008) zeigen, wie mithilfe von **Importance Sampling** der Wert einer Strategie besser eingeschätzt werden kann. Waugh et al. (2009) zeigen, dass das Abstraktionskonzept anfällig für systematische Fehler beim Annähern der Gleichgewichtslösung ist. Das heißt, dass der gesamte Ansatz auf wackligem Boden steht: Er funktioniert für bestimmte Spiele, aber nicht für andere. Korb et al. (1999) experimentieren mit einem Gegnermodell in Form eines Bayesschen Netzes. Es spielt Stud-Poker mit fünf Karten (Five Card Stud) genauso gut wie menschliche Experten. Zinkevich et al. (2008) zeigen, wie ein Ansatz, der Bedauern minimiert, angenäherte Gleichgewichte für Abstraktionen mit 10^{12} Zuständen finden kann, 100-mal mehr als vorherige Methoden.

Die Spieltheorie und MEPs sind in der Theorie der Markov-Spiele – auch als stochastische Spiele bezeichnet – zusammengefasst (Littman, 1994; Hu und Wellman, 1998). Shapley (1953) beschrieb den Algorithmus der Wert-Iteration unabhängig von Bellman, aber seine Ergebnisse stießen nicht überall auf Akzeptanz, vielleicht, weil sie im Kontext von Markov-Spielen präsentiert wurden. Evolutionäre Spieltheorie (Smith, 1982; Weibull, 1995) betrachtet Strategieverschiebung über die Zeit: Wie sollten Sie reagieren, wenn sich die Strategie Ihres Gegners ändert? Lehrbücher zur Spieltheorie unter dem Aspekt der Wirtschaftswissenschaften gibt es unter anderem von Myerson (1991), Fudenberg und Tirole (1991), Osborne (2004) sowie Osborne und Rubinstein (1994). Mailath und Samuelson (2006) konzentrieren sich auf wiederholte Spiele. Aus KI-Perspektive haben wir Nisan et al. (2007), Leyton-Brown und Shoham (2008) sowie Shoham und Leyton-Brown (2009).

Der Nobelpreis für Wirtschaftswissenschaften ging 2007 an Hurwicz, Maskin und Myerson dafür, dass sie „die Grundlagen für die Theorie des Mechanismenentwurfes gelegt haben“ (Hurwicz, 1973). Hardin (1968) stellte mit der Tragödie des Allgemeingutes ein motivierendes Problem für dieses Gebiet vor. Das **Offenbarungsprinzip** geht auf Myerson (1986) zurück und der **Satz über die Erlösäquivalenz** wurde unabhängig von Myerson (1981) sowie Riley und Samuelson (1981) entwickelt. Die beiden Wirtschaftswissenschaftler Milgrom (1997) und Klemperer (2002) schreiben über die Auktionen zur Frequenzvergabe mit einem Umfang von mehreren Milliarden Dollar, bei denen sie beteiligt waren.

Mechanismenentwurf wird in der Multiagenten-Planung (Hunsberger und Grosz, 2000; Stone et al., 2009) und Zeitplanung (Rassenti et al., 1982) eingesetzt. Varian (1995) bietet einen kurzen Überblick mit Verbindungen zur Informatikliteratur und Rosenschein und Zlotkin (1994) präsentieren eine umfangreiche Arbeit mit Anwendungen der verteilten KI. Verwandte Arbeiten zur verteilten KI findet man auch unter anderen Namen, wie beispielsweise *kollektive Intelligenz* (Tumer und Wolpert, 2000) und *marktbasierte Steuerung* (Clearwater, 1996). Seit 2001 findet jährlich der Wettbewerb *Trading Agents Competition* (TAC) statt, in dem Agenten versuchen, den höchsten Profit in einer Folge von Auktionen zu erzielen (Wellman et al., 2001; Arunachalam und Sadeh, 2005). Arbeiten zu Programmieraspekten in Auktionen erscheinen häufig bei den *ACM Conferences on Electronic Commerce*.

Zusammenfassung

Dieses Kapitel zeigt die Verwendung des Wissens über die Welt, um Entscheidungen zu treffen, selbst wenn die Ergebnisse einer Aktion unsicher sind und die Gewinne für das Handeln erst nach vielen Aktionen erstattet werden. Die wichtigsten Aspekte sind:

- Sequentielle Entscheidungsprobleme in unsicheren Umgebungen, auch als **Markov-Entscheidungsprozesse**, MEPs, bezeichnet, sind durch ein **Übergangsmodell** definiert, das die probabilistischen Ergebnisse der Aktionen angibt, sowie durch eine **Gewinnfunktion**, die den Gewinn für jeden Zustand angibt.
- Der Nutzen einer Zustandsfolge ist die Summe aller Gewinne über die Folge, möglicherweise über die Zeit vermindert. Die Lösung eines MEP ist eine **Taktik**, die jedem Zustand, den der Agent erreichen kann, eine Entscheidung zuordnet. Eine optimale Taktik maximiert den Nutzen der bei der Ausführung angetroffenen Zustandsfolgen.
- Der Nutzen eines Zustandes ist der erwartete Nutzen der Zustandsfolgen, die bei Ausführung einer optimalen Taktik beginnend bei diesem Zustand angetroffen werden. Der Algorithmus der **Wert-Iteration** für die Lösung von MEPs arbeitet, indem er iterativ die Gleichungen löst, die die Nutzen jedes Zustandes im Hinblick auf den Nutzen seiner Nachbarn darstellen.
- Die **Taktik-Iteration** wechselt zwischen der Berechnung der Nutzen von Zuständen unter der aktuellen Taktik und der Verbesserung der aktuellen Taktik im Hinblick auf die aktuellen Nutzen.
- Partiell beobachtbare MEPs sind sehr viel schwieriger zu lösen als normale MEPs. Sie können durch eine Umwandlung in einen MEP im stetigen Belief-State-Raum gelöst werden. Ein optimales Verhalten in partiell beobachtbaren MEPs beinhaltet die Informationssammlung, um Unsicherheit zu reduzieren und damit bessere Entscheidungen für die Zukunft zu treffen.
- Für partiell beobachtbare MEP-Umgebungen kann ein entscheidungstheoretischer Agent konstruiert werden. Der Agent verwendet ein **dynamisches Entscheidungsnetz**, um die Übergangs- und Beobachtungsmodelle darzustellen, seinen Belief State zu aktualisieren und mögliche Aktionsfolgen vorwärts zu projizieren.
- Die **Spieltheorie** beschreibt das rationale Verhalten für Agenten in Situationen, wo mehrere Agenten gleichzeitig agieren. Lösungen für Spiele sind **Nash-Gleichgewichte** – Strategieprofile, in denen kein Agent einen Ansporn hat, von der spezifizierten Strategie abzuweichen.
- Der **Mechanismenentwurf** kann genutzt werden, um die Regeln festzulegen, nach denen die Agenten handeln, um einen globalen Nutzen durch den Einsatz einzelner rationaler Agenten zu maximieren. Manchmal gibt es Mechanismen, die dieses Ziel erreichen, ohne dass jeder Agent die Auswahlen berücksichtigt, die die anderen Agenten getroffen haben.

Wir werden in *Kapitel 21* in die Welt der MEPs und partiell beobachtbaren MEPs zurückkehren, wo es um **verstärkende Lernmethoden** geht, die es einem Agenten erlauben, sein Verhalten aus der Erfahrung in sequentiellen, unsicheren Umgebungen zu verbessern.

Übungen zu Kapitel 17

Lösungshinweise

1 Berechnen Sie für die in Abbildung 17.1 gezeigte 4×3-Welt, welche Quadrate von (1, 1) aus mit der Aktionsfolge [*Rechts*, *Rechts*, *Rechts*, *Oben*, *Oben*] erreicht werden können und mit welchen Wahrscheinlichkeiten das möglich ist. Erklären Sie, wie diese Berechnung mit der Vorhersageaufgabe (siehe *Abschnitt 15.2.1*) für ein Hidden-Markov-Modell zusammenhängt.

2 Angenommen, wir definieren den Nutzen einer Zustandsfolge als den *maximalen* Gewinn, den man in jedem Zustand der Folge erhält. Zeigen Sie, dass diese Nutzenfunktion nicht zu stationären Prioritäten zwischen den Zustandsfolgen führt. Ist es immer noch möglich, eine Nutzenfunktion für Zustände zu definieren, sodass die MEN-Entscheidungsfindung ein optimales Verhalten ergibt?

3 Können endliche Suchprobleme genau in Markov-Entscheidungsprobleme übersetzt werden, sodass eine optimale Lösung des Letztgenannten auch eine optimale Lösung des Erstgenannten ist? Falls dem so ist, erklären Sie *präzise*, wie das Problem übersetzt wird und wie die Lösung zurückübersetzt wird; andernfalls erklären Sie *präzise*, warum das nicht so ist (d.h., geben Sie ein Gegenbeispiel).

4 Manchmal werden MEPs mit einer Gewinnfunktion $R(s, a)$ formuliert, die von der ausgewählten Aktion abhängig ist, oder mit einer Gewinnfunktion $R(s, a, s')$, die außerdem vom Ergebniszustand abhängig ist.

a. Schreiben Sie die Bellman-Gleichungen für diese Formulierungen.

b. Zeigen Sie, wie ein MEP mit einer Gewinnfunktion $R(s, a, s')$ in einen anderen MEP mit einer Gewinnfunktion $R(s, a)$ umgewandelt werden kann, sodass optimale Taktiken im neuen MEP genau den optimalen Taktiken im ursprünglichen MEP entsprechen.

c. Machen Sie dasselbe, um MEPs mit $R(s, a)$ in MEPs mit $R(s)$ umzuwandeln.

5 Ermitteln Sie aus der in Abbildung 17.1 gezeigten Umgebung alle Schwellenwerte für $R(s)$, sodass sich die optimale Taktik ändert, sobald die Schwelle überschritten wird. Sie brauchen eine Möglichkeit, die optimale Taktik ebenso wie ihren Wert für feste $R(s)$ zu berechnen. (*Hinweis*: Beweisen Sie, dass der Wert jeder festen Taktik linear mit $R(s)$ variiert.)

6 Gleichung (17.7) besagt, dass der Bellman-Operator eine Kontraktion ist.

a. Zeigen Sie, dass für beliebige Funktionen f und g gilt:

$$\left|\max_a f(a) - \max_a g(a)\right| \leq \max_a \left|f(a) - g(a)\right| .$$

b. Geben Sie einen Ausdruck für $|(BU_i - BU_i')(s)|$ an und wenden Sie dann das Ergebnis aus (a) an, um den Beweis zu vervollständigen, dass der Bellman-Operator eine Kontraktion ist.

7 In dieser Übung betrachten wir Zwei-Spieler-MEPs, die Nullsummenspielen mit abwechselnden Zügen entsprechen (siehe *Kapitel 5*). Seien die Spieler A und B und sei $R(s)$ der Gewinn für den Spieler A in s. (Der Gewinn für B ist immer gleich groß und entgegengesetzt.)

a. Sei $U_A(s)$ der Nutzen des Zustandes s, wenn A am Zug in s ist, und $U_B(s)$ der Nutzen des Zustandes s, wenn B in s am Zug ist. Alle Gewinne und Nutzen werden aus der Perspektive von A berechnet (wie in einem Minimax-Spielbaum): Schreiben Sie die Bellman-Gleichungen, die $U_A(s)$ und $U_B(s)$ definieren.

b. Erklären Sie, wie die Wert-Iteration für zwei Spieler mit diesen Gleichungen auszuführen ist, und definieren Sie ein geeignetes Abbruchkriterium.

c. Betrachten Sie das in Abbildung 5.17 beschriebene Spiel. Zeichnen Sie den Zustandsraum (statt des Spielbaumes) und zeigen Sie die Züge von A als durchgezogene Linien, die Züge von B als gestrichelte Linien. Markieren Sie jeden Zustand mit $R(s)$. Möglicherweise ist es hilfreich, die Zustände (s_A, s_B) in einem zweidimensionalen Raster anzuordnen, wobei s_A und s_B als „Koordinaten" dienen.

d. Wenden Sie eine Zwei-Spieler-Wert-Iteration an, um dieses Spiel zu lösen, und leiten Sie die optimale Taktik ab.

8 Betrachten Sie die in ▶ Abbildung 17.14(a) gezeigte 3×3-Welt. Das Übergangsmodell ist das gleiche wie in der 4×3-Welt von Abbildung 17.1: 80% der Zeit geht der Agent in die von ihm gewählte Richtung und in der restlichen Zeit bewegt er sich rechtwinklig zur vorgesehenen Richtung. Implementieren Sie eine Wert-Iteration für diese Welt für jeden unten angegebenen Wert r. Verwenden Sie geminderte Gewinne mit einem Minderungsfaktor von 0,99. Zeigen Sie die in jedem Fall erhaltene Taktik. Erläutern Sie intuitiv, warum der Wert von r zu den einzelnen Taktiken führt.

a. $r = 100$

b. $r = -3$

c. $r = 0$

d. $r = +3$

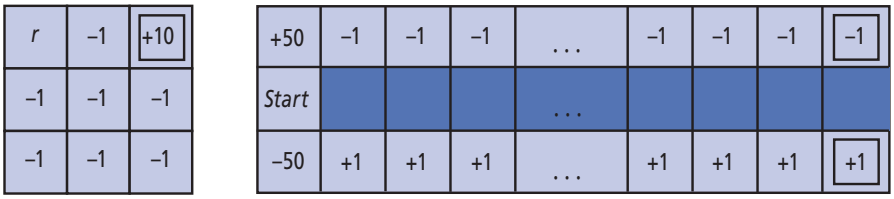


Abbildung 17.14: (a) 3×3-Welt für Übung 17.8. Für jeden Zustand ist der Gewinn angegeben. Das Quadrat oben rechts ist ein Terminalzustand. (b) 101×3-Welt für Übung 17.9 (wobei 93 identische Spalten in der Mitte ausgelassen wurden). Der Startzustand hat einen Gewinn von 0.

9 Betrachten Sie die 101×3-Welt in ▶ Abbildung 17.14(b). Im Startzustand hat der Agent die Wahl zwischen zwei deterministischen Aktionen, *Oben* oder *Unten*, doch in den anderen Zuständen hat der Agent genau eine deterministische Aktion, *Rechts*. Für welche Werte der Minderung γ sollte der Agent *Oben* und für welche *Unten* wählen, wenn eine verminderte Gewinnfunktion angenommen wird? Berechnen Sie den Nutzen jeder Aktion als Funktion von γ. (Beachten Sie, dass dieses einfache Beispiel eigentlich viele reale Situationen widerspiegelt, in denen der Wert einer unmittelbaren Aktion gegen die potenziell fortdauernden langfristigen Konsequenzen abzuwägen ist, wie zum Beispiel die Entscheidung, Verschmutzungen in einen See einzuleiten.)

10 Betrachten Sie ein ungemindertes MEP mit drei Zuständen, (1, 2, 3), mit den Gewinnen -1, -2 bzw. 0. Zustand 3 ist ein Endzustand. In den Zuständen 1 und 2 gibt es zwei mögliche Aktionen: a und b. Das Übergangsmodell sieht wie folgt aus:

- In Zustand 1 bringt die Aktion a den Agenten mit der Wahrscheinlichkeit 0,8 in den Zustand 2 und bewirkt mit der Wahrscheinlichkeit 0,2, dass der Agent stehen bleibt.
- In Zustand 2 bewegt die Aktion a den Agenten mit der Wahrscheinlichkeit 0,8 in den Zustand 1 und bewirkt mit der Wahrscheinlichkeit 0,2, dass der Agent stehen bleibt.
- In Zustand 1 und 2 bewegt die Aktion b den Agenten mit der Wahrscheinlichkeit 0,1 in den Zustand 3 und bewirkt mit der Wahrscheinlichkeit 0,9, dass der Agent stehen bleibt.

Beantworten Sie die folgenden Fragen:

a. Was kann *qualitativ* über die optimale Taktik in den Zuständen 1 und 2 ermittelt werden?

b. Wenden Sie die Taktik-Iteration an und zeigen Sie jeden Schritt vollständig, um die optimale Taktik und die Werte der Zustände 1 und 2 zu ermitteln. Gehen Sie davon aus, dass der Anfangstaktik in beiden Zuständen die Aktion b zur Verfügung steht.

c. Was passiert mit der Taktik-Iteration, wenn der Anfangstaktik in beiden Zuständen die Aktion a zur Verfügung steht? Hilft hier eine Minderung? Ist die optimale Taktik vom Minderungsfaktor abhängig?

11 Betrachten Sie die in Abbildung 17.1 gezeigte 4×3-Welt.

a. Implementieren Sie einen Umgebungssimulator für diese Umgebung, sodass die spezifische Geografie der Umgebung einfach abgeändert werden kann. Einen Teil des dafür erforderlichen Codes finden Sie bereits im online verfügbaren Code-Repository.

b. Erstellen Sie einen Agenten, der die Taktik-Iteration verwendet, und messen Sie seine Leistung im Umgebungssimulator, wobei Sie von verschiedenen Anfangszuständen ausgehen. Führen Sie von jedem Anfangszustand aus mehrere Experimente aus und vergleichen Sie den durchschnittlichen Gesamtgewinn pro Durchlauf mit dem Nutzen des Zustandes, wie ihn Ihr Algorithmus ermittelt hat.

c. Experimentieren Sie mit einer vergrößerten Umgebung. Wie variiert die Laufzeit für die Taktik-Iteration mit der Umgebungsgröße?

12 Wie lässt sich der Algorithmus zur Wertbestimmung nutzen, um den erwarteten Verlust eines Agenten anhand einer gegebenen Menge von Nutzenschätzungen U und einem geschätzten Modell P zu berechnen, verglichen mit einem Agenten, der korrekte Werte verwendet?

13 Der anfängliche Belief State b_0 für die 4×3-Umgebung mit partiell beobachtbarem MEP gemäß *Abschnitt 17.4* sei die gleichförmige Verteilung über den Nichtterminalzuständen, d.h. (1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 0, 0). Berechnen Sie den genauen Belief State b_1, nachdem der Agent die Bewegung $Links$ ausgeführt und sein Sensor 1 benachbarte Wand gemeldet hat. Berechnen Sie auch b_2 unter der Annahme, dass sich das Gleiche wiederholt.

14 Wie ist die Zeitkomplexität von d Schritten der Wert-Iteration bei partiell beobachtbaren MEPs für eine sensorlose Umgebung?

15 Betrachten Sie eine Version des partiell beobachtbaren MEP mit zwei Zuständen gemäß *Abschnitt 17.4.2*, bei der der Sensor in Zustand 0 zu 90% zuverlässig ist, in Zustand 1 aber keine Informationen liefert (d.h. 0 oder 1 mit gleicher Wahrscheinlichkeit meldet). Analysieren Sie entweder qualitativ oder quantitativ die Nutzenfunktion und die optimale Taktik für dieses Problem.

16 Zeigen Sie, dass ein dominantes Taktikgleichgewicht ein Nash-Gleichgewicht ist, aber nicht umgekehrt.

17 Lösen Sie das Spiel für *Drei*-Finger-Morra.

18 Betrachten Sie im *Gefangenendilemma* den Fall, bei dem sich Anna und Bodo nach jeder Runde mit einer Wahrscheinlichkeit von X erneut treffen. Nehmen Sie an, dass sich beide Spieler für die Strategie der fortlaufenden Bestrafung entscheiden (bei der jeder $leugnen$ wählt, außer wenn der andere Spieler irgendwann $gestehen$ gespielt hat) und bislang keiner der Spieler $gestehen$ gespielt hat.

Wie groß ist die erwartete zukünftige Gesamtauszahlung für die Wahl von $gestehen$ gegenüber $leugnen$, wenn $X = 0{,}2$ ist? Wie groß ist sie für $X = 0{,}5$? Für welchen Wert von X ist die erwartete zukünftige Gesamtauszahlung für jeden Spieler gleich, egal ob jemand in der aktuellen Runde $gestehen$ oder $leugnen$ wählt?

19 Eine holländische Auktion ist der englischen Auktion ähnlich. Anstatt aber mit einem niedrigen Preis zu beginnen und ihn stufenweise zu erhöhen, startet der Verkäufer bei der holländischen Auktion mit einem hohen Preis und senkt diesen allmählich, bis ein Käufer bereit ist, den jeweiligen Preis zu bezahlen. (Wenn mehrere Bieter den Preis akzeptieren, wird ein Bieter zufällig als der Gewinner ausgewählt.) Formal ausgedrückt, beginnt der Verkäufer mit einem Preis p und verringert p schrittweise um Inkremente von d, bis mindestens ein Käufer den Preis akzeptiert. Stimmt es, dass bei einer holländischen Auktion für beliebig kleines d letztlich immer der Bieter mit dem höchsten Wert den Artikel erhält, sofern alle Bieter rational agieren? Zeigen Sie mathematisch, warum dies zutrifft, oder erläutern Sie andernfalls, wie es möglich ist, dass der Bieter mit dem höchsten Wert den Artikel doch nicht erhält.

20 Stellen Sie sich einen Auktionsmechanismus vor, der wie eine Auktion mit steigendem Gebot abläuft, außer dass am Ende der gewinnende Bieter, der b_{max} geboten hat, nur $b_{max} / 2$ statt b_{max} zahlt. Wie groß ist der erwartete Erlös für den Auktionator bei diesem Mechanismus im Vergleich zu einer Standardauktion mit steigendem Gebot unter der Annahme, dass alle Agenten rational agieren?

21 Die Teams in der nationalen Hockeyliga erhielten in der Vergangenheit 2 Punkte für einen Gewinn und 0 Punkte für ein verlorenes Spiel. Bei Gleichstand wird in einer Verlängerung weitergespielt. Gewinnt niemand in der Nachspielzeit, endet das Spiel unentschieden und jedes Team erhält 1 Punkt. Die Ligafunktionäre waren der Meinung, dass die Teams in der Verlängerung zu konservativ spielten (um einen Verlust zu vermeiden) und es spannender wäre, wenn die Nachspielzeit einen Gewinner hervorbringt. Deshalb experimentierten die Funktionäre seit 1999 mit Mechanismenentwurf: Die Regeln wurden geändert, sodass ein Team, das in der Verlängerung verliert, 1 Punkt statt 0 Punkte erhält. Für einen Gewinn gibt es weiterhin 2 Punkte und für ein Unentschieden 1 Punkt.

a. War Hockey vor der Regeländerung ein Nullsummenspiel? Danach?

b. Nehmen Sie an, dass zu einer bestimmten Zeit t in einem Spiel die Heimmannschaft mit einer Wahrscheinlichkeit von p in der regulären Spielzeit gewinnt, mit einer Wahrscheinlichkeit von $0{,}78 - p$ verliert und mit einer Wahrscheinlichkeit von $0{,}22$ in die Verlängerung geht, wo es mit der Wahrscheinlichkeit q gewinnt, mit $0{,}9 - q$ verliert und mit $0{,}1$ ein Unentschieden erzielt. Geben Sie Gleichungen für den erwarteten Wert für die Heim- und Gastmannschaft an.

c. Stellen Sie sich vor, es wäre rechtlich und moralisch in Ordnung, wenn beide Teams übereinkommen, in der regulären Spielzeit ein Unentschieden abzuliefern, und erst in der Nachspielzeit ernsthaft versuchen, das Spiel zu gewinnen. Unter welchen Bedingungen (in Form von p und q) wäre es für beide Teams rational, diesem Pakt zuzustimmen?

d. Longley und Sankaran (2005) berichten, dass sich seit der Regeländerung der Anteil der Spiele mit einem Gewinner in der Nachspielzeit wunschgemäß erhöht hat (auf 18,2%), jedoch der Anteil der Spiele mit Verlängerung ebenso gestiegen ist (3,6%). Was lässt sich daraus für eine mögliche Absprache oder ein konservatives Spiel nach der Regeländerung ableiten?

TEIL V

Lernen

V

Aus Beispielen lernen

18

ÜBERBLICK

In diesem Kapitel beschreiben wir Agenten, die ihr Verhalten durch eine sorgfältige Beobachtung ihrer eigenen Erfahrungen verbessern können.

Ein Agent lernt, wenn er seine Leistung für zukünftige Aufgaben verbessert, nachdem er Beobachtungen über die Welt gemacht hat. **Lernen** reicht vom Trivialen – wie es beispielsweise beim Notieren einer Telefonnummer geschieht – bis zum Tiefgründigen – wie wir es von Albert Einstein kennen, der eine neue Theorie des Universums hergeleitet hat. In diesem Kapitel konzentrieren wir uns auf eine Klasse des Lernproblems, die zwar eingeschränkt erscheinen mag, tatsächlich jedoch ein riesiges Anwendungsgebiet besitzt: aus einer Sammlung von Ein-/Ausgabepaaren eine Funktion lernen, die die Ausgabe für neue Eingaben vorhersagt.

Warum sind wir daran interessiert, dass ein Agent lernt? Wenn sich der Entwurf des Agenten verbessern lässt, warum programmieren die Designer nicht einfach diese Verbesserung von Anfang an ein? Hierfür gibt es vor allem drei Gründe. Erstens können die Designer nicht alle möglichen Situationen vorhersehen, in die der Agent geraten kann. Zum Beispiel muss ein Roboter, der in einem Labyrinth navigieren soll, das Layout jedes neuen Labyrinths erst lernen. Zweitens sind die Designer nicht in der Lage, alle über die Zeit vorkommenden Änderungen vorherzusehen; ein Programm, das die Aktienkurse des nächsten Tages vorhersagen soll, muss lernen sich anzupassen, wenn die Bedingungen von Bullen- zu Bärenkursen wechseln. Drittens haben menschliche Programmierer manchmal keine Vorstellung davon, wie sie eine Lösung an sich programmieren sollen. Zum Beispiel können die meisten Leute problemlos die Gesichter der Familienmitglieder erkennen, doch sogar die besten Programmierer sind nicht in der Lage, einen Computer für diese Aufgabe zu programmieren, außer mithilfe von Lernalgorithmen. Dieses Kapitel gibt zuerst einen Überblick über die verschiedenen Lernformen, beschreibt dann in *Abschnitt 18.3* mit dem Lernen von Entscheidungsbäumen einen bekannten Ansatz und beschäftigt sich anschließend in den *Abschnitten 18.4* und *18.5* mit einer theoretischen Analyse des Lernens. Außerdem stellen wir verschiedene Lernsysteme aus der Praxis vor: lineare Modelle, nichtlineare Modelle (insbesondere neuronale Netze), parameterfreie Modelle und Support-Vector-Maschinen (SVM). Schließlich zeigen wir, wie Ensembles von Modellen ein einzelnes Modell an Leistung übertreffen können.

18.1 Lernformen

Jede Komponente eines Agenten lässt sich durch Lernen von Daten verbessern. Die Verbesserungen und die hierfür verwendeten Techniken hängen hauptsächlich davon ab,

- welche *Komponente* zu verbessern ist,
- welches *A-priori-Wissen* der Agent bereits besitzt,
- welche *Darstellung* für die Daten und die Komponente verwendet wird,
- welches *Feedback* verfügbar ist, von dem gelernt werden kann.

Zu lernende Komponenten

Kapitel 2 hat verschiedene Agentenkonzepte beschrieben. Zu den Komponenten dieser Agenten gehören:

1. Eine direkte Abbildung der Bedingungen auf den aktuellen Zustand von Aktionen
2. Eine Möglichkeit, relevante Eigenschaften der Welt von der Wahrnehmungsfolge abzuleiten
3. Informationen darüber, wie sich die Welt entwickelt, und über die Ergebnisse möglicher Aktionen, die der Agent ausführen kann
4. Nutzeninformationen, die angeben, wie wünschenswert Weltzustände sind
5. Aktion/Wert-Informationen, die angeben, wie wünschenswert Aktionen sind
6. Ziele, die Klassen von Zuständen beschreiben, die beim Erreichen den Nutzen des Agenten maximieren

Jede dieser Komponenten kann gelernt werden. Betrachten Sie beispielsweise einen Agenten, der lernt, um Taxifahrer zu werden. Immer wenn der Lehrer „Bremsen!“ ruft, kann der Agent eine Bedingung/Aktion-Regel lernen, wann er bremsen soll (Komponente 1); außerdem lernt der Agent jedes Mal, wenn der Lehrer nicht ruft. Durch die Betrachtung verschiedener Kamerabilder, von denen man ihm sagt, dass sie Busse enthalten, lernt er, Busse zu erkennen (2). Durch das Ausprobieren von Aktionen und Beobachtung der Ergebnisse – z.B. beim starken Bremsen auf einer nassen Straße – lernt er die Wirkung seiner Aktionen (3). Wenn er von Fahrgästen, die während der Fahrt stark durchgeschüttelt wurden, kein Trinkgeld erhält, lernt er daraus eine praktische Komponente seiner allgemeinen Nutzenfunktion (4).

Darstellung und A-priori-Wissen

Wir haben bereits mehrere Beispiele für Agentenkomponenten kennengelernt: aussagenlogische Sätze und Sätze der Logik erster Stufe für die Komponenten in einem logischen Agenten, Bayessche Netze für die inferentiellen Komponenten eines entscheidungstheoretischen Agenten usw. Für alle diese Darstellungen wurden effektive Lernalgorithmen entwickelt. Dieses Kapitel (und der größte Teil der Forschung auf dem Gebiet des maschinellen Lernens) behandelt Eingaben, die eine **faktorisierte Darstellung** – einen Vektor von Attributwerten – und Ausgaben – entweder als stetige numerische Werte oder als diskrete Werte – bilden. *Kapitel 19* behandelt Funktionen und A-priori-Wissen, das sich aus Sätzen von Logik erster Stufe zusammensetzt, und *Kapitel 20* konzentriert sich auf Bayessche Netze.

Es gibt noch eine andere Art, die verschiedenen Lerntypen zu betrachten. Wir sagen, dass das Lernen einer (möglicherweise falschen) allgemeinen Funktion oder Regel aus spezifischen Eingabe/Ausgabe-Paaren als **induktives Lernen** bezeichnet wird. In *Kapitel 19* erfahren Sie, dass sich Lernen auch **analytisch** oder **deduktiv** realisieren lässt: Ausgehend von einer bekannten allgemeinen Regel zu einer neuen Regel, die zwar logisch abgeleitet wird, aber dennoch nützlich ist, weil sie eine effizientere Verarbeitung erlaubt.

Lernen von Feedback

Es gibt drei Typen von Feedback, die die drei Haupttypen des Lernens bestimmen:

Beim **nicht überwachten Lernen** lernt der Agent Muster in der Eingabe, selbst wenn kein explizites Feedback bereitgestellt wird. Die häufigste Aufgabe für nicht überwachtes Lernen ist das **Clustern**: Erkennen möglicherweise nützlicher Cluster von Eingabebeispielen. So könnte ein Taxi-Agent allmählich ein Konzept von „guten Verkehrstagen“ und „schlechten Verkehrstagen“ entwickeln, ohne jemals benannte Beispiele dafür von einem Lehrer bekommen zu haben.

Beim **Reinforcement Learning** (**verstärkenden Lernen**) lernt der Agent aus einer Reihe von Verstärkungen – Belohnungen oder Bestrafungen. Beispielsweise könnte das Fehlen eines Trinkgeldes am Ende einer Fahrt für den Taxi-Agenten ein Hinweis darauf sein, dass er etwas falsch gemacht hat. Die beiden Punkte für einen Gewinn am Ende einer Schachpartie sagen dem Agenten, dass er etwas richtig gemacht hat. Es liegt beim Agenten zu entscheiden, welche der Aktionen vor der Verstärkung am meisten dafür verantwortlich waren.

Beim **überwachten Lernen** beobachtet der Agent einige Eingabe/Ausgabe-Paare und lernt eine Funktion, die Eingaben auf Ausgaben abbildet. In der obigen Komponente (1) sind die Eingaben Wahrnehmungen und die Ausgabe wird durch einen Lehrer angegeben, der „Bremsen!“ oder „Nach links!“ ruft. In Komponente (2) sind die Eingaben Kamerabilder und die Ausgaben kommen wieder von einem Lehrer, der sagt: „Das ist ein Bus.“ In (3) ist die Bremstheorie eine Funktion von Zuständen und Bremsaktionen zum Anhalteweg in Metern. In diesem Fall steht der Ausgabewert direkt von den Wahrnehmungen des Agenten (im Nachhinein) zur Verfügung; die Umgebung ist der Lehrer.

In der Praxis lassen sich diese Unterscheidungen nicht immer so klar treffen. Beim **halb überwachten Lernen** erhalten wir wenige benannte Beispiele und müssen aus einer großen Sammlung von nicht benannten Beispielen das Bestmögliche machen. Selbst die Bezeichnungen an sich müssen nicht unbedingt die orakelhaften Wahrheiten sein, auf die wir hoffen. Stellen Sie sich vor, Sie richten ein System ein, um das Alter einer Person von einem Foto zu erraten. Dazu haben Sie einige benannte Beispiele zusammengetragen, und zwar als Schnappschüsse von Personen, die Sie nach ihrem Alter gefragt haben. Prinzipiell wäre das überwachtes Lernen. Doch in der Realität gibt nicht jeder sein wahres Alter an. Nicht nur dass es zufällige Störungen in den Daten gibt, vielmehr sind die Ungenauigkeiten systematisch. Sie aufzudecken, ist ein Problem des nicht überwachten Lernens, in das Bilder, selbst gemeldete Altersangaben und wahre (unbekannte) Alter einfließen. Somit erzeugen sowohl Störungen als auch fehlende Bezeichnungen ein Kontinuum zwischen überwachtem und nicht überwachtem Lernen.

18.2 Überwachtes Lernen

Die Aufgabe beim überwachten Lernen lautet:

> Für eine Trainingsmenge mit N Beispielen von Eingabe/Ausgabe-Paaren
>
> $$(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$$
>
> in der jedes y_j durch eine unbekannte Funktion $y = f(x)$ generiert wurde, ist eine Funktion h zu entdecken, die die wahre Funktion f annähert.

Hier können x und y beliebige Werte sein, nicht unbedingt Zahlen. Die Funktion h ist eine **Hypothese**.[1] Lernen ist eine Suche im Raum möglicher Hypothesen nach einer, die gute Ergebnisse liefert, selbst für neue Beispiele, die über die Trainingsmenge hinausgehen. Um die Genauigkeit einer Hypothese zu messen, geben wir ihr eine **Testmenge** von Beispielen, die sich von der Trainingsmenge unterscheidet. Wir sagen, dass eine Hypothese gut verallgemeinert, wenn sie korrekt den Wert von y für neue Beispiele vorhersagt. Manchmal ist die Funktion f stochastisch – es ist keine strenge Funktion von x und wir müssen eine bedingte probabilistische Verteilung $\mathbf{P}(Y \mid x)$ lernen.

Gehört die Ausgabe y zu einer endlichen Wertmenge (wie zum Beispiel *sonnig*, *wolkig* oder *regnerisch*), bezeichnet man das Lernproblem als **Klassifizierung**. Gibt es nur zwei Werte, spricht man von boolescher oder binärer Klassifizierung. Ist y eine Zahl (etwa die morgige Temperatur), wird das Lernproblem als **Regression** bezeichnet. (Technisch gesehen sucht man bei einem Regressionsproblem nach einem bedingten Erwartungs- oder Durchschnittswert von y, da die Wahrscheinlichkeit, den genauen Wert für y zu finden, gegen 0 geht.)

Tipp

▶ Abbildung 18.1 zeigt ein bekanntes Beispiel: Anpassen einer Funktion einer einzelnen Variablen an einige Datenpunkte. Die Beispiele sind Punkte in der (x, y)-Ebene, wobei $y = f(x)$ gilt. Wir kennen f nicht, nähern diese aber mit einer Funktion h an, die wir aus einem Hypothesenraum $\mathcal{H}$ auswählen – hier aus der Menge der Polynome wie etwa $x^5 + 3x^2 + 2$. Abbildung 18.1(a) zeigt einige Daten mit genauer Anpassung durch eine Gerade (das Polynom $0{,}4x + 3$). Die Gerade wird als **konsistente** Hypothese bezeichnet, weil sie mit allen Daten übereinstimmt. Abbildung 18.1(b) zeigt ein Polynom höheren Grades, das ebenfalls konsistent zu denselben Daten ist. Daraus erkennen wir das erste Problem beim induktiven Lernen: *Wie wählen wir aus mehreren konsistenten Hypothesen aus?* Eine Variante besteht darin, die *einfachste* Hypothese auszuwählen, die konsistent mit den Daten ist. Dieses Prinzip ist das sogenannte **Ockham-Rasiermesser**, benannt nach dem englischen Philosophen William von Ockham aus dem 14. Jahrhundert, der es verwendete, um gegen alle Arten von Komplikationen scharf zu argumentieren. Es ist nicht leicht, Einfachheit zu definieren, doch liegt es auf der Hand, dass ein Polynom 1. Grades einfacher ist als ein Polynom 7. Grades und somit (a) gegenüber (b) zu bevorzugen ist. Diese Intuition werden wir in *Abschnitt 18.4.3* noch konkretisieren.

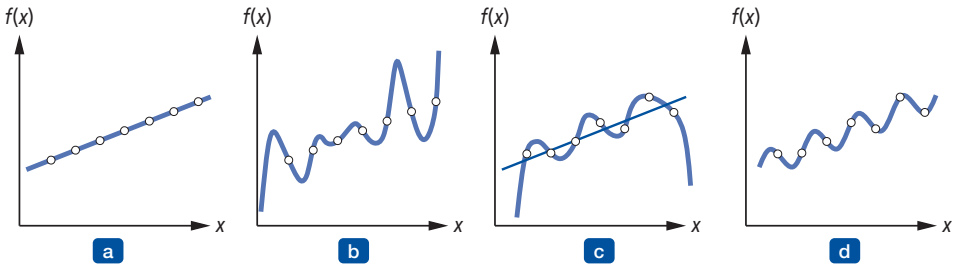


Abbildung 18.1: (a) Beispielpaare $(x, f(x))$ und eine konsistente, lineare Hypothese. (b) Eine konsistente polynomiale Hypothese 7. Grades für dieselbe Datenmenge. (c) Eine andere Datenmenge, die sich mit einem Polynom 6. Grades oder einer Geraden annähern lässt. (d) Eine einfache, genau sinusförmige Anpassung an dieselbe Datenmenge.

1 Ein Hinweis zur Schreibweise: Sofern nicht anders angemerkt, indizieren wir mit j die N Beispiele; x_j ist immer die Eingabe und y_j die Ausgabe. Falls die Eingabe speziell ein Vektor von Attributwerten ist (beginnend mit *Abschnitt 18.3*), verwenden wir $\mathbf{x}_j$ für das j-te Beispiel und indizieren mit i die n Attribute jedes Beispiels. Die Elemente von $\mathbf{x}_j$ werden geschrieben als $x_{j,1}, x_{j,2}, \ldots, x_{j,n}$.

Tipp

Abbildung 18.1(c) zeigt eine zweite Datenmenge. Es gibt keine konsistente Gerade für diese Datenmenge; vielmehr brauchen wir ein Polynom 6. Grades, um eine genaue Übereinstimmung zu finden. Da es nur sieben Datenpunkte gibt, ist ein Polynom mit sieben Parametern zweifellos ungeeignet, ein Muster in den Daten zu beschreiben, und wir erwarten nicht, dass es gut verallgemeinert. Abbildung 18.1(c) zeigt auch eine Gerade, die zwar mit keinem der Datenpunkte konsistent ist, aber auch für unbekannte Werte von x eine brauchbare Verallgemeinerung darstellen kann. Im Allgemeinen gibt es einen Kompromiss zwischen komplexen Hypothesen, die sich den Trainingsdaten gut anpassen, und einfacheren Hypothesen, die möglicherweise besser verallgemeinern. In Abbildung 18.1(d) erweitern wir den Hypothesenraum $\mathcal{H}$, um Polynome sowohl über x als auch $\sin(x)$ zu erlauben, und finden, dass sich die Daten in (c) exakt durch eine einfache Funktion der Form $ax + b + c\sin(x)$ annähern lassen. Dies zeigt, wie wichtig die Auswahl des Hypothesenraumes ist. Wir sagen, dass ein Lernproblem **realisierbar** ist, wenn der Hypothesenraum die wahre Funktion enthält. Leider können wir nicht immer erkennen, ob ein gegebenes Lernproblem realisierbar ist, da die wahre Funktion nicht bekannt ist.

In manchen Fällen wird ein Analytiker, der sich ein Problem ansieht, feinstufigere Unterscheidungen über den Hypothesenraum treffen, um – sogar noch bevor irgendwelche Daten bekannt sind – nicht nur festzustellen, ob eine Hypothese möglich oder unmöglich ist, sondern vielmehr, wie wahrscheinlich sie ist. Überwachtes Lernen lässt sich durchführen, indem man die Hypothese h^* auswählt, die bei gegebenen Daten am wahrscheinlichsten ist:

$$h^* = \underset{h \in \mathcal{H}}{\operatorname{argmax}} P(h \mid \mathit{Daten}).$$

Gemäß Bayesscher Regel ist dies äquivalent zu

$$h^* = \underset{h \in \mathcal{H}}{\operatorname{argmax}} P(\mathit{Daten} \mid h) P(h).$$

Dann können wir sagen, dass die A-priori-Wahrscheinlichkeit $P(h)$ für ein Polynom 1. oder 2. Grades hoch, für ein Polynom 7. Grades niedriger und für Polynome 7. Grades mit großen, scharfen Spikes wie in Abbildung 18.1(b) besonders niedrig ist. Wir lassen ungewöhnlich aussehende Funktionen zu, wenn anhand der Daten zu sehen ist, dass wir sie wirklich brauchen, doch wir unterdrücken sie gewissermaßen, indem wir ihnen eine niedrige A-priori-Wahrscheinlichkeit zuordnen.

Tipp

Warum sollte $\mathcal{H}$ nicht die Klasse aller Java-Programme oder Turing-Maschinen sein? Immerhin kann jede berechenbare Funktion durch eine Turing-Maschine dargestellt werden und das ist das Beste, was wir tun können. Ein Problem bei dieser Vorgehensweise ist, dass sie die Rechenkomplexität des Lernens nicht berücksichtigt. *Es gibt einen Kompromiss zwischen der Ausdruckskraft eines Hypothesenraumes und der Komplexität, eine gute Hypothese in diesem Raum zu finden*. Beispielsweise lässt sich die Anpassung einer Geraden an die Daten recht einfach berechnen; bei einem Polynom höheren Grades ist dies etwas schwieriger; und die Anpassung von Turing-Maschinen ist im Allgemeinen nicht einmal entscheidbar. Einfache Hypothesenräume sind auch aus dem Grund zu bevorzugen, weil wir vermutlich h verwenden werden, nachdem wir diese Funktion gelernt haben. Und wenn h eine lineare Funktion ist, lässt sich $h(x)$ garantiert schnell berechnen, während es bei einem beliebigen Turing-Maschinen-Programm nicht einmal sicher ist, ob die Berechnung terminiert. Aus diesen Gründen hat sich ein Großteil der Arbeit zum Lernen auf einfache Repräsentationen konzentriert.

Wir werden sehen, dass die Abwägung zwischen Ausdruckskraft und Komplexität nicht so einfach ist, wie es zunächst scheint: Wie wir bereits bei Logik erster Stufe in *Kapitel 8* gesehen haben, kommt es häufig vor, dass sich mit einer ausdrucksstarken Sprache eine *einfache* Hypothese für die Anpassung an die Daten formulieren lässt, während eine beschränkte Ausdruckskraft der Sprache bedeutet, dass jede konsistente Hypothese sehr komplex sein muss. Zum Beispiel lassen sich die Schachregeln in Logik erster Stufe auf einer oder zwei Seiten niederschreiben, während bei Aussagenlogik Tausende von Seiten erforderlich wären.

18.3 Lernen von Entscheidungsbäumen

Die Induktion von Entscheidungsbäumen ist eine der einfachsten und doch erfolgreichsten Formen maschinellen Lernens. Wir beschreiben zuerst die Darstellung – den Hypothesenraum – und zeigen dann, wie sich eine gute Hypothese lernen lässt.

18.3.1 Entscheidungsbäume als Leistungselemente

Ein **Entscheidungsbaum** repräsentiert eine Funktion, die als Eingabe einen Vektor von Attributwerten übernimmt und eine „Entscheidung" – einen einzelnen Ausgabewert – zurückgibt. Die Ein- und Ausgabewerte können diskret oder stetig sein. Wir konzentrieren uns zunächst auf Probleme, bei denen die Eingaben diskrete Werte sind und die Ausgabe genau zwei mögliche Werte liefert. Dies ist die boolesche Klassifizierung, wobei jede Beispieleingabe als *true* (**positives** Beispiel) oder *false* (**negatives** Beispiel) klassifiziert wird.

Ein Entscheidungsbaum gelangt zu seiner Entscheidung, indem er eine Reihe von Tests ausführt. Jeder interne Knoten im Baum entspricht einem Test des Wertes eines der Eingabeattribute A_i und die vom Knoten abgehenden Zweige werden mit den möglichen Werten des Attributes $A_i = v_{ik}$ beschriftet. Jeder Blattknoten im Baum spezifiziert einen Wert, der durch die Funktion zurückzugeben ist. Die Darstellung als Entscheidungsbaum scheint für Menschen sehr natürlich zu sein; viele Anleitungen (z.B. zur Autoreparatur) sind komplett als einziger Entscheidungsbaum geschrieben, der sich über Hunderte von Seiten erstreckt.

Als Beispiel erstellen wir einen Entscheidungsbaum, um zu entscheiden, ob man in einem Restaurant auf einen freien Tisch warten soll. Hier ist das Ziel, eine Definition für das **Zielprädikat** *WerdenWarten* zu lernen. Zuerst listen wir die Attribute auf, die wir als Teil der Eingabe betrachten:

1. *Wechseln*: ob es ein geeignetes anderes Restaurant in der Nähe gibt
2. *Bar*: ob das Restaurant einen gemütlichen Barbereich hat, wo man warten kann
3. *Frei/Sams*: freitags und samstags *true*
4. *Hungrig*: ob wir hungrig sind
5. *Gäste*: wie viele Menschen im Restaurant sind (Werte sind *Keine*, *Einige* oder *Voll*)
6. *Preis*: der Preisbereich des Restaurants (€, €€, €€€)
7. *Regen*: ob es draußen regnet

8 *Reservierung*: ob wir eine Reservierung vorgenommen haben

9 *Typ*: die Art des Restaurants (französisch, italienisch, Thai, Burger)

10 *GeschätzteWartezeit*: die vom Ober geschätzte Wartezeit (0-10 Minuten, 10-30, 30-60 oder >60)

Jede Variable besitzt einen kleinen Satz möglicher Werte; zum Beispiel ist der Wert von *GeschätzteWartezeit* keine Ganzzahl, sondern einer der vier diskreten Werte 0-10, 10-30, 30-60 oder >60. ▶ Abbildung 18.2 zeigt den Entscheidungsbaum, den einer der Autoren (SR) normalerweise für diese Domäne einsetzt. Beachten Sie, dass der Baum die Attribute *Preis* und *Typ* ignoriert. Der Baum verarbeitet die Beispiele, indem er an der Wurzel beginnt und dem jeweiligen Zweig folgt, bis ein Blattknoten erreicht ist. So wird ein Beispiel mit *Gäste* = *Voll* und *GeschätzteWartezeit* = 0−10 als positiv klassifiziert (d.h., wir warten auf einen freien Tisch).

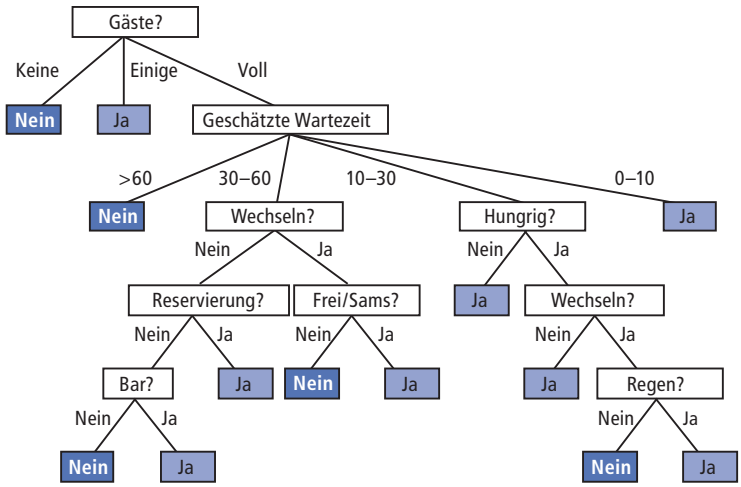


Abbildung 18.2: Ein Entscheidungsbaum, der ermittelt, ob wir auf einen freien Tisch warten.

18.3.2 Ausdruckskraft von Entscheidungsbäumen

Ein boolescher Entscheidungsbaum ist logisch äquivalent der Zusicherung, dass das Zielattribut nur dann *true* ist, wenn die Eingabeattribute einem der Pfade, der zu einem Blattknoten mit dem Wert *true* führt, entsprechen. In Aussagenlogik geschrieben, sieht dies folgendermaßen aus:

$$Ziel \Leftrightarrow (Pfad_1 \vee Pfad_2 \vee ...).$$

Hier ist jeder *Pfad* eine Konjunktion von Attribut-Wert-Tests, die erforderlich sind, um diesem Pfad zu folgen. Somit ist der gesamte Ausdruck äquivalent zur disjunktiven Normalform (siehe *Übung 7.18*), was bedeutet, dass sich jede Funktion in Aussagenlogik auch als Entscheidungsbaum ausdrücken lässt. Zum Beispiel lautet der ganz rechte Pfad in Abbildung 18.2

$$Pfad = (Gäste = Voll \wedge GeschätzteWartezeit = 0 - 10).$$

Für ein breites Spektrum von Problemen liefert das Entscheidungsbaumformat ein schönes prägnantes Ergebnis. Einige Funktionen lassen sich jedoch nicht so prägnant dar-

stellen. Zum Beispiel erfordert die Mehrheitsfunktion, die nur dann *true* zurückgibt, wenn mehr als die Hälfte der Eingaben *true* sind, einen exponentiell großen Entscheidungsbaum. Mit anderen Worten, Entscheidungsbäume sind für einige Funktionen gut geeignet, für andere dagegen nicht. Gibt es *irgendeine* Repräsentation, die für *alle* Arten von Funktionen effizient ist? Leider nein. Wir können dies auf sehr allgemeine Weise zeigen. Betrachten Sie die Menge aller booleschen Funktionen für n Attribute. Wie viele verschiedene Funktionen befinden sich in dieser Menge? Dies ist genau die Anzahl der verschiedenen Wahrheitstabellen, die wir aufschreiben können, weil die Funktion durch ihre Wahrheitstabelle definiert ist. Eine Wahrheitstabelle für n Attribute umfasst 2^n Zeilen, jeweils eine Zeile für jede Wertekombination der Attribute. Wir können die „Antwort"-Spalte der Tabelle als 2^n-Bit-Zahl betrachten, die die Funktion definiert. Das heißt, es gibt 2^{2^n} verschiedene Funktionen (und es gibt mehr als diese Anzahl von Bäumen, da sich für dieselbe Funktion mehr als ein Baum berechnen lässt.) Dies ist eine riesige Zahl. Zum Beispiel stehen in unserem Restaurantproblem mit lediglich 10 booleschen Attributen 2^{1024} bzw. etwa 10^{308} verschiedene Funktionen zur Auswahl. Und bei 20 Attributen sind es über $10^{300.000}$. Wir brauchen einen genialen Algorithmus, um konsistente Hypothesen in einem solch großen Raum zu finden.

18.3.3 Entscheidungsbäume per Induktion aus Beispielen ableiten

Ein Beispiel für einen booleschen Entscheidungsbaum besteht aus einem ($\mathbf{x}$, y)-Paar, wobei $\mathbf{x}$ ein Vektor von Werten für die Eingabeattribute und y ein einzelner boolescher Ausgabewert ist. ▶ Abbildung 18.3 zeigt eine Trainingsmenge mit zwölf Beispielen. Die positiven Beispiele sind diejenigen, wo das Ziel *WerdenWarten true* ist, ($\mathbf{x}_1$, $\mathbf{x}_3$, ...); die negativen Beispiele sind diejenigen, wo das Ziel *WerdenWarten false* ist, ($\mathbf{x}_2$, $\mathbf{x}_5$, ...).

Beispiel	Eingabeattribute										Ziel
	Alt	Bar	Frei	Hungrig	Gäste	Preis	Regen	Reser.	Typ	Wart	Werden-Warten
$\mathbf{x}_1$	Ja	Nein	Nein	Ja	Einige	€€€	Nein	Ja	Franz.	0-10	$y_1 = Ja$
$\mathbf{x}_2$	Ja	Nein	Nein	Ja	Voll	€	Nein	Nein	Thai	30-60	$y_2 = Nein$
$\mathbf{x}_3$	Nein	Ja	Nein	Nein	Einige	€	Nein	Nein	Burger	0-10	$y_3 = Ja$
$\mathbf{x}_4$	Ja	Nein	Ja	Ja	Voll	€	Ja	Nein	Thai	10-30	$y_4 = Ja$
$\mathbf{x}_5$	Ja	Nein	Ja	Nein	Voll	€€€	Nein	Ja	Franz.	>60	$y_5 = Nein$
$\mathbf{x}_6$	Nein	Ja	Nein	Ja	Einige	€€	Ja	Ja	Ital.	0-10	$y_6 = Ja$
$\mathbf{x}_7$	Nein	Ja	Nein	Nein	Keine	€	Ja	Nein	Burger	0-10	$y_7 = Nein$
$\mathbf{x}_8$	Nein	Nein	Nein	Ja	Einige	€€	Ja	Ja	Thai	0-10	$y_8 = Ja$
$\mathbf{x}_9$	Nein	Ja	Ja	Nein	Voll	€	Ja	Nein	Burger	>60	$y_9 = Nein$
$\mathbf{x}_{10}$	Ja	Ja	Ja	Ja	Voll	€€€	Nein	Ja	Ital.	10-30	$y_{10} = Nein$
$\mathbf{x}_{11}$	Nein	Nein	Nein	Nein	Keine	€	Nein	Nein	Thai	0-10	$y_{11} = Nein$
$\mathbf{x}_{12}$	Ja	Ja	Ja	Ja	Voll	€	Nein	Nein	Burger	30-60	$y_{12} = Ja$

Abbildung 18.3: Beispiele aus der Restaurantdomäne.

Wir möchten einen Baum haben, der mit den Beispielen konsistent und so klein wie möglich ist. Doch egal wie wir die Größe messen, es ist leider ein nicht handhabbares Problem, den kleinsten konsistenten Baum zu finden; es gibt keine Möglichkeit, 2^{2^n} Bäume effizient zu durchsuchen. Allerdings können wir mit einigen einfachen Heuristiken eine gute Näherungslösung finden: einen kleinen (wenn auch nicht den kleinsten) konsistenten Baum. Der Algorithmus DECISION-TREE-LEARNING wendet eine gierige Teilen-und-Herrschen-Strategie an: immer das wichtigste Attribut zuerst testen. Mit „wichtigste“ meinen wir ein Attribut, das in der Klassifizierung eines Beispiels den größten Unterschied verursacht. Auf diese Weise hoffen wir, die korrekte Klassifizierung mit einer kleinen Anzahl an Tests zu erhalten, d.h., alle Pfade im Baum sind kurz und der Baum als Ganzes ist flach.

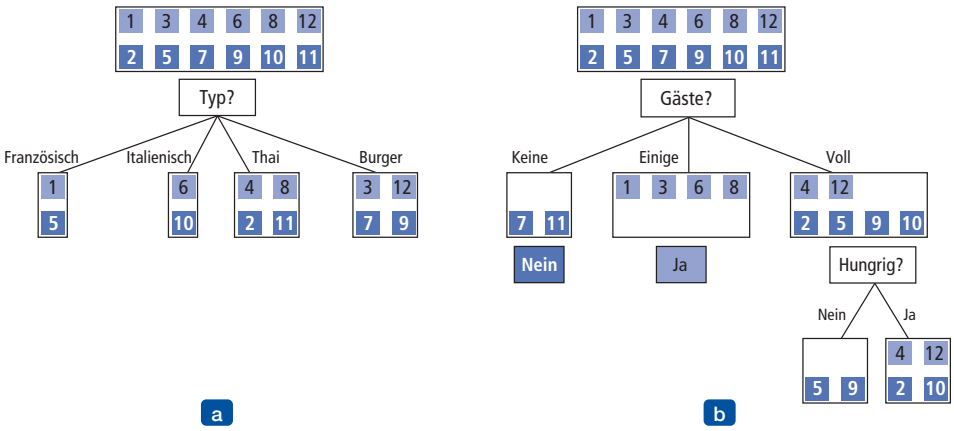


Abbildung 18.4: Aufteilung der Beispiele durch Testen von Attributen. An jedem Knoten zeigen wir die verbleibenden positiven (helle Felder) und negativen (dunkle Felder) Beispiele. (a) Aufteilung nach dem *Typ* bringt uns näher zur Unterscheidung zwischen positiven und negativen Beispielen. (b) Aufteilung nach *Gästen* ist eine sinnvolle Maßnahme zur Unterscheidung zwischen positiven und negativen Beispielen. Nach der Aufteilung nach *Gästen* ist *Hungrig* ein recht sinnvoller zweiter Test.

▶ Abbildung 18.4(a) zeigt, dass *Typ* ein schlechtes Attribut ist, weil es uns vier mögliche Ergebnisse bringt, die wiederum dieselbe Anzahl positiver und negativer Beispiele haben. In ▶ Abbildung 18.4(b) dagegen sehen wir, dass *Gäste* ein relativ wichtiges Attribut ist, denn wenn sein Wert gleich *Keine* oder *Einige* ist, dann erhalten wir Beispielmengen, für die wir definitiv antworten können (*Nein* bzw. *Ja*). Ist der Wert gleich *Voll*, erhalten wir eine gemischte Menge an Beispielen. Im Allgemeinen ist nach der ersten Aufteilung der Beispiele mit dem ersten Attributtest jedes Ergebnis ein neues Entscheidungsbaum-Lernproblem – mit weniger Beispielen und einem Attribut weniger. Für diese rekursiven Probleme müssen vier Fälle berücksichtigt werden:

1. Wenn die verbleibenden Beispiele alle positiv (oder alle negativ) sind, sind wir fertig: Wir können mit *Ja* oder *Nein* antworten. Abbildung 18.4(b) zeigt Beispiele hierfür in den Zweigen *Keine* und *Einige*.

2. Wenn es einige positive und einige negative Beispiele gibt, wählen Sie das beste Attribut für ihre Aufteilung aus. Abbildung 18.4(b) zeigt, wie *Hungrig* verwendet wird, um die verbleibenden Beispiele aufzuteilen.

3 Wenn keine weiteren Beispiele übrig sind, heißt das, dass für diese Kombination von Attributwerten kein Beispiel beobachtet wurde. Wir geben dann einen Standardwert zurück, berechnet aus der Mehrheitsklassifizierung aller Beispiele, die beim Konstruieren des übergeordneten Knotens verwendet wurden. Diese werden in der Variablen *parent_examples* zurückgegeben.

4 Wenn es keine weiteren Attribute gibt, aber sowohl positive als auch negative Beispiele, heißt das, dass diese Beispiele genau dieselbe Beschreibung, aber unterschiedliche Klassifizierungen haben. Dies kann passieren, wenn die Daten fehlerhaft oder **verrauscht** sind, die Domäne nicht deterministisch ist oder wir ein Attribut, das die Beispiele unterscheiden würde, nicht beobachten können. In diesem Fall ist es am besten, die Mehrheitsklassifizierung der verbleibenden Beispiele zurückzugeben.

```
function DECISION-TREE-LEARNING(examples, attributes, parent_examples) returns
einen Baum

   if examples ist leer then return PLURALITY-VALUE(parent_examples)
   else if alle examples haben dieselbe Klassifikation then return
           die Klassifikation
   else if attributes ist leer then return PLURALITY-VALUE(examples)
   else
      A ← argmax_{a ∈ attributes} IMPORTANCE(a, examples)
      tree ← ein neuer Entscheidungsbaum mit dem Wurzeltest A
      for each Wert v_k von A do
         exs ← {e : e ∈ examples and e.A = v_k}
         subtree ← DECISION-TREE-LEARNING(exs, attributes - A, examples)
         Zweig zu tree mit Beschriftung (A = v_k) und Unterbaum subtree
         hinzufügen
      return tree
```

Abbildung 18.5: Der Entscheidungsbaum-Lernalgorithmus.

▶ Abbildung 18.5 zeigt den Algorithmus DECISION-TREE-LEARNING. Die Menge der Beispiele ist zwar für die *Konstruktion* des Baumes entscheidend, im Baum selbst erscheinen die Beispiele aber nicht. Ein Baum besteht lediglich aus Tests von Attributen in den inneren Knoten, Werten von Attributen in den Zweigen und Ausgabewerten an den Blattknoten. Die Details für IMPORTANCE finden Sie in *Abschnitt 18.3.4*. ▶ Abbildung 18.6 zeigt die Ausgabe des Lernalgorithmus für unsere Beispieltrainingsmenge. Der Baum unterscheidet sich deutlich vom Originalbaum in Abbildung 18.2. Man könnte daraus schließen, dass der Lernalgorithmus keine gute Leistung beim Lernen der korrekten Funktion gezeigt hat. Das wäre jedoch ein falscher Schluss. Der Lernalgorithmus betrachtet die *Beispiele*, nicht die korrekte Funktion, und tatsächlich stimmt seine Hypothese (siehe Abbildung 18.6) nicht mit allen Beispielen überein, sondern ist wesentlich einfacher als der Originalbaum! Der Lernalgorithmus hat keinen Grund, Tests für *Regen* und *Reservierung* aufzunehmen, weil er alle Beispiele auch ohne sie klassifizieren kann. Außerdem hat er ein interessantes und zuvor nicht vermutetes Muster erkannt: Der erste Autor wartet am Wochenende auf Thai-Essen. Der Algorithmus macht zwangsläufig auch Fehler in den Fällen, für die er keine Beispiele gesehen hat. So hat er nie einen Fall gesehen, wo die Wartezeit 0-10 Minuten beträgt, aber das Restaurant voll ist. Für einen Fall, wo *Hungrig* falsch ist, entscheidet der Baum, nicht zu warten, aber ich (Stuart Russel) würde sicher warten. Mit weiteren Trainingsbeispielen könnte das Lernprogramm diesen Fehler korrigieren.

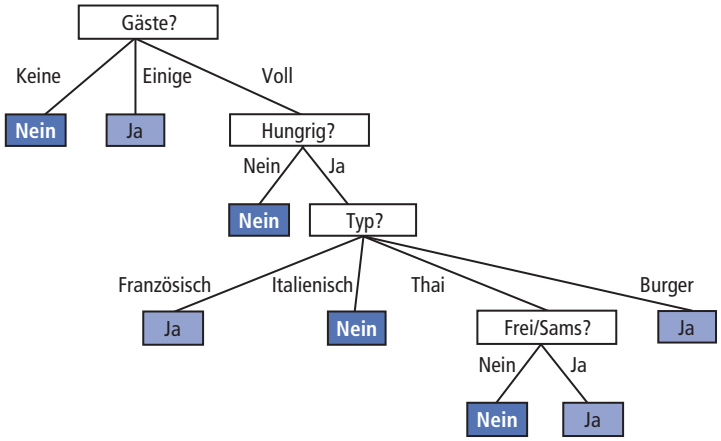


Abbildung 18.6: Der Entscheidungsbaum, der aus der Trainingsmenge mit zwölf Beispielen induziert wurde.

Es besteht die Gefahr, den vom Algorithmus ausgewählten Baum überzuinterpretieren. Wenn es mehrere Variablen ähnlicher Wichtigkeit gibt, geschieht die Auswahl zwischen ihnen etwas willkürlich: Bei leicht geänderten Eingabebeispielen würde zuerst eine andere Variable für die Teilung verwendet werden und der Baum würde vollkommen anders aussehen. Die durch den Baum berechnete Funktion ist zwar dennoch ähnlich, jedoch kann die Struktur des Baumes erheblich variieren.

Die Genauigkeit eines Lernalgorithmus können wir mit einer **Lernkurve** bewerten, wie sie in ▶ Abbildung 18.7 zu sehen ist. Wir haben 100 Beispiele zur Verfügung, die wir in eine Trainingsmenge und eine Testmenge aufteilen. Mit der Trainingsmenge lernen wir eine Hypothese h und messen ihre Genauigkeit mit der Testmenge. Dabei beginnen wir mit einer Größe der Trainingsmenge von 1 und nehmen jeweils 1 Beispiel bis 99 hinzu. Für jede Größe wiederholen wir die zufällige Aufteilung 20 Mal und bilden den Durchschnitt aus den Ergebnissen der 20 Versuche. Die Kurve zeigt, dass die Genauigkeit bei wachsender Trainingsmenge zunimmt. (Aus diesem Grund bezeichnet man Lernkurven auch als **Happy-Graphen**.) In diesem Diagramm erreichen wir 95% Genauigkeit und es scheint, dass sich die Kurve mit mehr Daten weiter verbessern ließe.

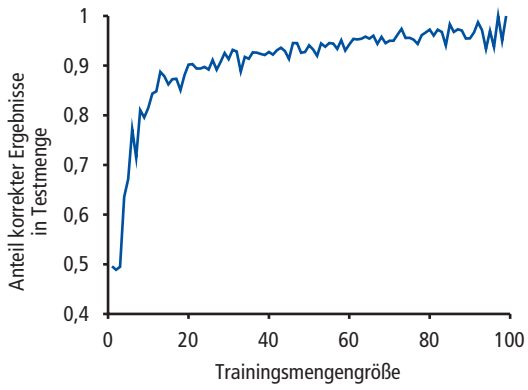


Abbildung 18.7: Eine Lernkurve des Algorithmus zum Lernen eines Entscheidungsbaumes mit 100 zufällig generierten Beispielen in der Restaurantdomäne. Jeder Datenpunkt verkörpert den Durchschnitt von 20 Versuchen.

18.3.4 Auswahl von Attributtests

Die beim Lernen von Entscheidungsbäumen verwendete gierige Suche ist darauf ausgelegt, die Tiefe des fertigen Baumes zu minimieren. Die Idee dabei ist, das Attribut auszuwählen, das so weit wie möglich geht, um eine genaue Klassifizierung der Beispiele bereitzustellen. Ein perfektes Attribut unterteilt die Beispiele in Mengen, die alle positiv oder alle negativ sind. Das Attribut *Gäste* ist nicht perfekt, aber es ist relativ gut. Ein wirklich unbrauchbares Attribut, wie etwa *Typ*, hinterlässt die Beispielmengen mit etwa demselben Verhältnis an positiven und negativen Beispielen wie in der Originalmenge.

Alles was wir also brauchen, ist eine formale Bewertung von „relativ gut“ und „wirklich unbrauchbar“, und wir können die Funktion IMPORTANCE aus Abbildung 18.5 implementieren. Wir verwenden das Konzept des Informationsgewinns, das als **Entropie** definiert ist, der grundlegenden Größe in der Informationstheorie (Shannon und Weaver, 1949).

Entropie ist ein Maß für die Unbestimmtheit einer Zufallsvariablen; Informationserfassung entspricht einer Verringerung der Entropie. Eine Zufallsvariable mit nur einem einzigen Wert – eine Münze, die immer auf Kopf fällt – besitzt keine Unbestimmtheit und somit ist ihre Entropie als null definiert; wir erhalten folglich keine Informationen, indem wir ihren Wert beobachten. Beim Werfen einer fairen Münze kommt Kopf oder Zahl, 0 oder 1, gleichwahrscheinlich und wir werden bald zeigen, dass dies als „1 Bit“ Entropie zählt. Das Werfen eines fairen *vier*seitigen „Würfels“ hat 2 Bit Entropie, da 2 Bit erforderlich sind, um eine der vier gleichwahrscheinlichen Auswahlen zu beschreiben. Betrachten Sie nun eine unfaire Münze, die in 99% der Fälle Kopf zeigt. Intuitiv hat diese Münze weniger Unbestimmtheit als die faire Münze – wenn wir Kopf raten, liegen wir nur in 1% der Fälle falsch – sodass wir ihr ein Entropiemaß zuordnen, das zwar nahe null liegt, aber positiv ist. Im Allgemeinen ist die Entropie einer Zufallsvariablen V mit den Werten v_k, die jeweils die Wahrscheinlichkeit $P(v_k)$ haben, definiert als

$$\text{Entropie:} H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = -\sum_k P(v_k) \log_2 P(v_k).$$

Wir können überprüfen, dass die Entropie eines fairen Münzwurfes tatsächlich 1 Bit ist:

$$H(Fair) = -(0{,}5 \log_2 0{,}5 + 0{,}5 \log_2 0{,}5) = 1.$$

Wenn die Münze gezinkt wird, um 99% Kopf zu liefern, erhalten wir

$$H(Gezinkt) = -(0{,}99 \log_2 0{,}99 + 0{,}01 \log_2 0{,}01) \approx 0{,}08 \text{ Bit}.$$

Es ist zweckmäßig, $B(q)$ als Entropie einer booleschen Zufallsvariablen zu definieren, die mit der Wahrscheinlichkeit q *true* ist:

$$B(q) = -(q \log_2 q + (1 - q) \log_2 (1 - q)).$$

Somit ist $H(Gezinkt) = B(0{,}99) \approx 0{,}08$. Kommen wir nun zum Lernen des Entscheidungsbaumes zurück. Wenn eine Trainingsmenge p positive Beispiele und n negative Beispiele enthält, berechnet sich die Entropie des Zielattributes der gesamten Menge zu

$$H(Ziel) = B\left(\frac{p}{p+n}\right).$$

Die Restaurant-Trainingsmenge in Abbildung 18.3 hat $p = n = 6$, die entsprechende Entropie ist demnach $B(0{,}5)$ oder genau 1 Bit. Ein Test für ein einzelnes Attribut A liefert uns möglicherweise nur einen Teil dieses 1 Bit. Wie viel das ist, können wir genau messen, wenn wir uns die verbleibende Entropie *nach* dem Attributtest ansehen.

Ein Attribut A mit d verschiedenen Werten teilt die Trainingsmenge E in die Teilmengen $E_1, ..., E_d$. Jede Teilmenge E_k enthält p_k positive und n_k negative Beispiele. Wenn wir also diesem Zweig folgen, brauchen wir zusätzliche $B(p_k/(p_k + n_k))$ Informationsbits, um die Frage zu beantworten. Ein zufällig aus der Trainingsmenge ausgewähltes Beispiel hat den k-ten Wert für das Attribut mit der Wahrscheinlichkeit $(p_k + n_k)/(p + n)$, sodass die erwartete restliche Entropie nach dem Testen des Attributes A

$$\text{Rest}(A) = \sum_{k=1}^{d} \frac{p_k + n_k}{p+n} B\left(\frac{p_k}{p_k + n_k}\right)$$

beträgt.

Der **Informationsgewinn** aus dem Attributtest von A ist die erwartete Reduzierung der Entropie:

$$\text{Gewinn}(A) = B\left(\frac{p}{p+n}\right) - \text{Rest}(A).$$

Letztlich ist *Gewinn*(A) genau das, was wir für die Implementierung der Funktion IMPORTANCE benötigen. Wenn wir wieder die in Abbildung 18.4 betrachteten Attribute heranziehen, erhalten wir

$$\text{Gewinn}(\textit{Gäste}) = 1 - \left[\frac{2}{12}B\left(\frac{0}{2}\right) + \frac{4}{12}B\left(\frac{4}{4}\right) + \frac{6}{12}B\left(\frac{2}{6}\right)\right] \approx 0{,}541 \text{ Bit}$$

$$\text{Gewinn}(\textit{Typ}) = 1 - \left[\frac{2}{12}B\left(\frac{1}{2}\right) + \frac{2}{12}B\left(\frac{1}{2}\right) + \frac{4}{12}B\left(\frac{2}{4}\right) + \frac{4}{12}B\left(\frac{2}{4}\right)\right] = 0 \text{ Bit}.$$

Dies bestätigt unsere Intuition, dass *Gäste* ein besseres Attribut für die Aufteilung ist. Tatsächlich hat *Gäste* den höchsten Gewinn aller Attribute und würde durch den Algorithmus zum Lernen des Entscheidungsbaumes als Wurzel ausgewählt.

18.3.5 Verallgemeinerung und Überanpassung

Bei bestimmten Problemen generiert der Algorithmus DECISION-TREE-LEARNING einen großen Baum, obwohl eigentlich gar kein Muster zu finden ist. Betrachten Sie das Problem, vorhersagen zu wollen, ob beim Werfen eines Würfels das Ergebnis 6 erscheint oder nicht. Nehmen Sie an, dass die Experimente mit verschiedenen Würfeln durchgeführt werden und die Attribute, die jedes Trainingsbeispiel beschreiben, auch die Farbe des Würfels, sein Gewicht, den Zeitpunkt des Wurfes und ob die Experimentatoren ihre Finger gekreuzt haben angeben. Wenn die Würfel fair sind, wäre es richtig, einen Baum mit einem einzelnen Knoten zu lernen, der „Nein“ sagt. Der Algorithmus DECISION-TREE-LEARNING greift jedoch jedes Muster auf, das er in der Eingabe finden kann. Wenn sich herausstellt, dass es zwei Würfe eines 7 Gramm schweren blauen Würfels mit gekreuzten Fingern gibt und beide das Ergebnis 6 liefern, konstruiert der Algorithmus eventuell

einen Pfad, der in diesem Fall 6 vorhersagt. Dieses Problem heißt **Überanpassung** (**Overfitting**). Als allgemeines Phänomen tritt Überanpassung bei allen Lernarten auf, selbst wenn die Zielfunktion überhaupt nicht zufällig ist. In Abbildung 18.1(b) und (c) sind Überanpassungen der Daten durch Polynomfunktionen zu sehen. Überanpassung tritt wahrscheinlicher auf, wenn der Hypothesenraum und die Anzahl der Eingabeattribute zunehmen, und weniger wahrscheinlich, wenn wir die Anzahl der Trainingsbeispiele erhöhen.

Für Entscheidungsbäume lässt sich eine Überanpassung mit der sogenannten **Kürzung des Entscheidungsbaumes** bekämpfen. Dieses Verfahren entfernt Knoten, die nicht offensichtlich relevant sind. Wir beginnen mit einem vollständigen Baum, wie ihn DECISION-TREE-LEARNING generiert, und suchen dann nach einem Testknoten, der nur Blattknoten als Nachfolger hat. Wenn der Test irrelevant zu sein scheint – d.h. nur Rauschen in den Daten erkennt –, eliminieren wir den Test und ersetzen ihn durch einen Blattknoten. Diesen Prozess wiederholen wir, wobei wir jeden Test mit ausschließlich Blattnachfolgern betrachten, bis die einzelnen Zweige entweder gekürzt oder in der bestehenden Form beibehalten werden.

Wie erkennen wir nun, ob ein Knoten ein irrelevantes Attribut testet? Angenommen, wir befinden uns bei einem Knoten, der aus p positiven und n negativen Beispielen besteht. Wenn das Attribut irrelevant ist, erwarten wir, dass es die Beispiele in Teilmengen aufteilt, die jeweils ungefähr den gleichen Anteil an positiven Beispielen wie die Gesamtmenge haben – $p/(p + n)$ –, und somit der Informationsgewinn nahe null sein wird.[2] Der Informationsgewinn ist folglich ein guter Anhaltspunkt für Irrelevanz. Doch welche Größe sollten wir als Informationsgewinn fordern, um nach einem bestimmten Attribut aufzuteilen?

Diese Frage lässt sich mithilfe eines statistischen **Signifikanztestes** beantworten. Ein derartiger Test nimmt zunächst an, dass es kein zugrundeliegendes Muster gibt (die sogenannte **Nullhypothese**). Dann werden die eigentlichen Daten analysiert, um das Ausmaß zu berechnen, wie weit sie von einem perfekten Fehlen eines Musters abweichen. Wenn der Grad der Abweichung statistisch unwahrscheinlich ist (normalerweise ist darunter eine Wahrscheinlichkeit von 5% oder weniger zu verstehen), gilt dies als guter Beweis für das Vorhandensein eines signifikanten Musters in den Daten. Berechnet werden die Wahrscheinlichkeiten aus Standardverteilungen des Abweichungsbetrages, den man in zufällig erhobenen Stichproben erwarten würde.

In diesem Fall besagt die Nullhypothese, dass das Attribut irrelevant ist und folglich der Informationsgewinn für eine unendlich große Probe null sein wird. Wir müssen die Wahrscheinlichkeit berechnen, mit der eine Probe der Größe $v = n + p$ unter der Nullhypothese die beobachtete Abweichung von der erwarteten Verteilung der positiven und negativen Beispiele zeigen wird. Die Abweichung können wir messen, indem wir die tatsächlichen Anzahlen von positiven und negativen Beispielen in jeder Teilmenge p_k und n_k mit den erwarteten Anzahlen $\hat{p}_k$ und $\hat{n}_k$ vergleichen und echte Irrelevanz annehmen:

$$\hat{p}_k = p \times \frac{p_k + n_k}{p + n} \qquad \hat{n}_k = n \times \frac{p_k + n_k}{p + n}.$$

2 Der Gewinn ist grundsätzlich positiv, außer in dem unwahrscheinlichen Fall, in dem alle Verhältnisse *genau* gleich sind (siehe Übung 18.7).

Ein praktisches Maß der Gesamtabweichung ist gegeben mit:

$$\Delta = \sum_{k=1}^{d} \frac{(p_k - \hat{p}_k)^2}{\hat{p}_k} + \frac{(n_k - \hat{n}_k)^2}{\hat{n}_k} .$$

Unter der Nullhypothese ist der Wert von Δ entsprechend der χ^2 (Chi-Quadrat)-Verteilung mit $v - 1$ Freiheitsgraden verteilt. Mit einer χ^2-Tabelle oder einer Standardroutine aus einer Statistikbibliothek lässt sich feststellen, ob ein bestimmter Δ-Wert die Nullhypothese bestätigt oder abweist. Nehmen Sie als Beispiel das Restaurantattribut *Typ* mit vier Werten und somit vier Freiheitsgraden. Ein Wert von $\Delta = 7{,}82$ oder mehr würde die Nullhypothese auf dem 5%-Niveau zurückweisen (und ein Wert von $\Delta = 11{,}35$ oder mehr auf dem 1%-Niveau). In Übung 18.10 haben Sie die Aufgabe, den Algorithmus DECISION-TREE-LEARNING zu erweitern, um diese Form der Kürzung – die sogenannte χ^2-**Kürzung** – zu implementieren. Fehler in den Beschriftungen der Beispiele – wie etwa ein Beispiel (**x**, *Ja*), das (**x**, *Nein*) sein sollte – führen zu einem linearen Ansteigen des Vorhersagefehlers, während Fehler in den Beschreibungen der Beispiele – wie etwa *Preis* = €, wenn es eigentlich *Preis* = €€ heißen müsste – einen asymptotischen Effekt haben, der sich verschlimmert, wenn der Baum in kleinere Mengen zerlegt wird. Gekürzte Bäume bieten eine deutlich bessere Leistung als ungekürzte Bäume, wenn die Daten sehr viel Rauschen enthalten. Außerdem sind die gekürzten Bäume oftmals wesentlich kleiner und damit einfacher zu verstehen.

Zum Schluss noch eine Warnung: Da χ^2-Kürzung und Informationsgewinn einander ähnlich aussehen, stellt sich die Frage, ob man sie mit einem als **frühes Stoppen** bezeichneten Ansatz kombinieren kann – das Generieren von Knoten im Algorithmus für den Entscheidungsbaum stoppen lassen, wenn es kein gutes Attribut mehr für die Aufteilung gibt, anstatt sich die ganze Mühe zu machen, Knoten zu generieren und sie dann wegzukürzen. Problematisch beim frühen Stoppen ist, dass wir dann keine Situationen mehr erkennen können, in denen es nicht ein gutes Attribut gibt, aber Kombinationen von Attributen existieren, die informativ sind. Betrachten Sie zum Beispiel die XOR-Funktion mit zwei binären Attributen. Wenn es ungefähr gleich viele Beispiele für alle vier Kombinationen von Eingabewerten gibt, ist kein Attribut informativ. Dennoch ist es richtig, nach einem der Attribute (egal nach welchem) aufzuteilen und dann auf der zweiten Ebene informative Aufteilungen zu erhalten. Frühes Stoppen verpasst dies, doch Generieren-und-dann-Kürzen bekommt es korrekt in den Griff.

18.3.6 Die Anwendbarkeit von Entscheidungsbäumen erweitern

Um die Entscheidungsbaum-Induktion auf eine breitere Vielfalt von Problemen anwenden zu können, müssen mehrere Aspekte betrachtet werden. Einige davon erwähnen wir hier kurz – am besten ist es, wenn Sie sich anhand der entsprechenden Übungen eingehender mit der Materie vertraut machen.

- **Fehlende Daten:** In vielen Domänen sind nicht alle Attributwerte für jedes Beispiel bekannt. Die Werte wurden vielleicht nicht aufgezeichnet oder ihre Ermittlung war zu teuer. Daraus können zwei Probleme entstehen: Erstens, wie kann man für einen vollständigen Entscheidungsbaum ein Beispiel klassifizieren, für das eines der Testattribute fehlt? Zweitens, wie sollte man die Formel für den Informationsgewinn anpassen, wenn einige Beispiele unbekannte Werte für das Attribut haben? Diese Fragen werden in Übung 18.11 aufgegriffen.

- **Mehrwertige Attribute:** Wenn ein Attribut mehrere mögliche Werte hat, gibt das Maß für den Informationsgewinn einen ungeeigneten Hinweis darauf, wie nützlich das Attribut ist. Im Extremfall hat ein Attribut wie etwa *GenaueZeit* für jedes Beispiel einen anderen Wert. In diesem Fall ist jede Untermenge der Beispiele eine einelementige Menge (engl. Singleton) mit eindeutiger Klassifizierung, sodass das Maß für den Informationsgewinn für dieses Attribut den höchsten Wert hätte. Jedoch ist es unwahrscheinlich, dass diese Teilung zuerst ausgewählt wird, um den besten Baum zu liefern. Eine Lösung besteht darin, das **Gewinnverhältnis** zu verwenden (Übung 18.12). Alternativ kann man einen booleschen Test der Form $A = v_k$ erlauben, d.h., nur genau einen der möglichen Werte für ein Attribut herauszugreifen und die verbleibenden Werte einem eventuell später im Baum erfolgenden Test zu überlassen.
- **Stetige und ganzzahlige Eingabeattribute:** Stetige oder ganzzahlige Attribute, wie etwa *Höhe* und *Gewicht*, haben eine unendliche Menge möglicher Werte. Anstatt unendlich viele Zweige zu erzeugen, findet der Entscheidungsbaum-Lernalgorithmus normalerweise den **Aufteilungspunkt**, der den höchsten Informationsgewinn bringt. Beispielsweise könnte es für einen bestimmten Knoten im Baum die meisten Informationen erbringen, auf $Gewicht > 160$ zu testen. Es gibt effiziente Methoden, um gute Aufteilungspunkte zu finden: Zunächst sortiert man die Werte des Attributes und betrachtet dann nur die Aufteilungspunkte, die in sortierter Reihenfolge zwischen zwei Beispielen liegen und verschiedenen Klassifizierungen angehören, während man die laufenden Summen der positiven und negativen Beispiele auf jeder Seite des Aufteilungspunktes verfolgt. Das Aufteilen ist der aufwändigste Teil realer Anwendungen für das Lernen von Entscheidungsbäumen.
- **Stetigwertige Ausgabeattribute:** Wenn wir versuchen, einen numerischen Wert vorherzusagen, wie etwa den Preis einer Wohnung, brauchen wir statt eines Klassifizierungsbaumes einen **Regressionsbaum**. Ein solcher Baum hat an jedem Blatt eine lineare Funktion einer Untermenge numerischer Attribute statt eines einzelnen Wertes. Beispielsweise könnte der Zweig für Wohnungen mit zwei Schlafzimmern mit einer linearen Funktion für Wohnfläche, Anzahl der Badezimmer und durchschnittliches Einkommen im Wohnviertel enden. Der Lernalgorithmus muss entscheiden, wann die Aufteilung beendet werden soll, und auf die verbleibenden Attribute eine lineare Regression (siehe *Abschnitt 18.6*) anwenden.

Ein Entscheidungsbaum-Lernsystem für Anwendungen aus der realen Welt muss in der Lage sein, alle diese Probleme zu verarbeiten. Die Verarbeitung stetigwertiger Variablen ist besonders wichtig, weil sowohl physische als auch finanzielle Prozesse numerische Daten zur Verfügung stellen. Es wurden mehrere kommerzielle Pakete entwickelt, die diese Kriterien erfüllen, und sie wurden eingesetzt, um Tausende Systeme für die Praxis zu erstellen. In vielen Industrie- und Wirtschaftsbereichen probiert man oftmals als Erstes Entscheidungsbäume aus, wenn aus einer Datenmenge eine Klassifizierungsmethode zu extrahieren ist. Eine wichtige Eigenschaft von Entscheidungsbäumen ist, dass ein Mensch den Grund für die Ausgabe des Lernalgorithmus verstehen kann. (Dies ist eine *gesetzliche Forderung* für finanzielle Entscheidungen, die Antidiskriminierungsgesetzen unterliegen.) Dies ist eine Eigenschaft, die bestimmte andere Darstellungen wie zum Beispiel neuronale Netze nicht aufweisen.

18.4 Die beste Hypothese bewerten und auswählen

Wir möchten eine Hypothese lernen, die sich den zukünftigen Daten am besten anpasst. Um dies konkret umzusetzen, müssen wir „zukünftige Daten“ und „am besten“ definieren. Wir treffen die **Stationaritätsannahme**: Es gibt eine Wahrscheinlichkeitsverteilung über Beispielen, die über die Zeit hinweg stationär bleibt. Jeder Beispieldatenpunkt ist (bevor wir ihn sehen) eine Zufallsvariable E_j, deren beobachteter Wert $e_j = (x_j, v_j)$ als Stichprobe aus der Verteilung entnommen wird und unabhängig von den vorherigen Beispielen ist:

$$\mathbf{P}(E_j \mid E_{j-1}, E_{j-2}, \ldots) = \mathbf{P}(E_j).$$

Zudem besitzt jedes Beispiel eine identische A-priori-Wahrscheinlichkeitsverteilung:

$$\mathbf{P}(E_j) = P(E_{j-1}) = \mathbf{P}(E_{j-2}) = \ldots$$

Beispiele, die diese Annahmen erfüllen, werden als *unabhängig und identisch verteilt* (kurz *u.i.v.* oder englisch i.i.d. für independent and identically distributed) bezeichnet. Eine u.i.v.-Annahme verbindet die Vergangenheit mit der Zukunft; ohne eine derartige Verbindung lässt sich nichts garantieren – die Zukunft könnte alles sein. (Wir zeigen später, dass Lernen trotzdem stattfinden kann, wenn es in der Verteilung *langsame* Änderungen gibt.)

Im nächsten Schritt ist zu klären, was unter „beste Anpassung“ zu verstehen ist. Wir definieren die **Fehlerrate** einer Hypothese als Anteil der Fehler, die sie macht – d.h. wie oft $h(x) \neq y$ für ein (x, y)-Beispiel auftritt. Nur weil eine Hypothese h eine geringe Fehlerrate für die Trainingsmenge aufweist, heißt das nicht, dass sie gut verallgemeinert. Ein Professor weiß, dass eine Prüfung kein genaues Bild von der Leistung der Studenten liefert, wenn die Prüfungsfragen vorab bekannt sind. Bei Hypothesen ist es genauso: Um eine genaue Bewertung einer Hypothese zu erhalten, müssen wir sie auf einer Menge von Beispielen testen, die sie noch nicht gesehen hat. Der einfachste Ansatz wurde bereits erwähnt: zufälliges Aufteilen der verfügbaren Daten in eine Trainingsmenge, aus der der Lernalgorithmus h erzeugt wird, und eine Testmenge, auf der die Genauigkeit von h ausgewertet wird. Diese auch als **wahre Kreuzvalidierung** bezeichnete Methode weist den Nachteil auf, dass sie nicht mit allen verfügbaren Daten arbeitet. Denn wenn wir die Hälfte der Daten für die Testmenge verwenden, trainieren wir nur mit der Hälfte der Daten und erhalten möglicherweise eine schwache Hypothese. Reservieren wir andererseits nur 10% der Daten für die Testmenge, kann die Bewertung der tatsächlichen Genauigkeit aufgrund statistischer Zufälle schlecht ausfallen.

Mithilfe der **k-fachen Kreuzvalidierung** lässt sich mehr aus den Daten herausholen und dennoch eine genaue Schätzung erhalten. Dabei erfüllt jedes Beispiel eine Doppelfunktion – als Trainingsdaten und als Testdaten. Zunächst teilen wir die Daten in k gleiche Teilmengen auf. Dann führen wir k Lernrunden aus, wobei in jeder Runde $1/k$ der Daten als Testmenge vorgehalten wird und die übrigen Beispiele als Trainingsdaten dienen. Die gemittelte Bewertung der Testmenge aus k Runden sollte dann eine bessere Abschätzung liefern als eine Einzelwertung. Übliche Werte für k sind 5 und 10 – genügend für eine Schätzung, die statistisch wahrscheinlich genau ist, zum Preis einer fünf- bis zehnmal längeren Rechenzeit. Der Spezialfall mit $k = n$ wird auch als **Leave-One-Out-Kreuzvalidierung** (kurz **LOOCV** von engl. Leave-One-Out Cross Validation) bezeichnet.

Trotz erheblicher Anstrengungen von Statistik-Methodikern entkräften Benutzer häufig ihre Ergebnisse, indem sie versehentlich einen Blick auf ihre Testdaten werfen (**Peeking**). Das kann wie folgt passieren: Ein Lernalgorithmus besitzt verschiedene „Knöpfe", mit denen sich sein Verhalten optimieren lässt – um zum Beispiel verschiedene Unterscheidungskriterien für die Auswahl des nächsten Attributes beim Lernen eines Entscheidungsbaumes festzulegen. Der Forscher generiert Hypothesen für mehrere unterschiedliche Einstellungen der Knöpfe, misst ihre Fehlerraten für die Testmenge und meldet die Fehlerrate der besten Hypothese. Doch leider hat dabei Peeking stattgefunden! Der Grund dafür liegt darin, dass er die Hypothese *nach der Fehlerrate für die Testmenge* ausgewählt hat, sodass Informationen über die Testmenge in den Lernalgorithmus eingeflossen sind.

Peeking ergibt sich daraus, dass die Leistung der Testmenge sowohl für die *Auswahl* als auch für die *Bewertung* einer Hypothese herangezogen wird. Um dies zu vermeiden, muss man die Testmenge wirklich außen vor lassen – wegschließen, bis der Lernprozess vollständig ist und man einfach nur noch eine unabhängige Bewertung der letzten Hypothese haben möchte. (Und sollten Ihnen dann die Ergebnisse nicht gefallen, müssen Sie eine vollkommen neue Testmenge erzeugen und ausschließen, falls Sie zurückgehen und eine bessere Hypothese suchen möchten.) Wenn die Testmenge weggeschlossen ist, Sie aber trotzdem die Leistung von unbekannten Daten messen möchten, um eine gute Hypothese auszuwählen, teilen Sie die verfügbaren Daten (ohne die Testmenge) in eine Trainingsmenge und eine **Validierungsmenge** auf. Der nächste Abschnitt zeigt, wie Sie mithilfe von Validierungsmengen einen guten Kompromiss zwischen Hypothesenkomplexität und Anpassungsgüte erreichen.

18.4.1 Modellauswahl: Komplexität gegenüber Anpassungsgüte

In Abbildung 18.1 haben wir gezeigt, dass sich Polynome höheren Grades zwar besser an die Trainingsdaten anpassen können, doch bei zu hohem Grad eine Überanpassung stattfindet und die Leistung mit Validierungsdaten schlecht ist. Die Auswahl des Polynomgrades ist ein Problem der **Modellauswahl**. Die Aufgabe, die beste Hypothese zu finden, können Sie sich als zwei Aufgaben vorstellen: Die Modellauswahl definiert den Hypothesenraum und die **Optimierung** findet dann die beste Hypothese in diesem Raum.

In diesem Abschnitt erläutern wir, wie Sie unter den Modellen auswählen, die nach Größe (*size*) parametrisiert sind. Zum Beispiel haben wir bei Polynomen *size* = 1 für lineare Funktionen, *size* = 2 für quadratische Funktionen usw. Für Entscheidungsbäume ließe sich als Größe die Anzahl der Knoten im Baum angeben. In allen Fällen möchten wir den Wert des Parameters *size* finden, der den besten Kompromiss zwischen Unteranpassung und Überanpassung sowie die beste Genauigkeit in Bezug auf die Testmenge liefert.

▶ Abbildung 18.8 zeigt einen Algorithmus, der Modellauswahl und Optimierung realisiert. Es handelt sich um einen Wrapper, der einen Lernalgorithmus als Argument übernimmt (zum Beispiel DECISION-TREE-LEARNING). Der Wrapper listet die Modelle nach einem Parameter – *size* – auf. Für jede Größe berechnet er mithilfe von Kreuzvalidierung in Bezug auf *Learner* die mittlere Fehlerrate für die Trainings- und Testmengen. Wir beginnen mit dem kleinsten, einfachsten Modell (das wahrscheinlich eine Unteranpassung an die Daten ergibt) und betrachten in jedem Durchlauf schrittweise komplexere Modelle, bis die Modelle eine Überanpassung zeigen.

```
function CROSS-VALIDATION-WRAPPER(Learner, k, examples) returns eine Hypothese

  local variables: errT, ein durch size indiziertes Array, das Fehlerraten
                           für die Trainingsmenge speichert
                   errV, ein durch size indiziertes Array, das Fehlerraten
                           für die Validierungsmenge speichert
  for size = 1 to ∞ do
    errT[size], errV[size] ← CROSS-VALIDATION(Learner, size, k, examples)
    if errT konvergiert hat then do
      best_size ← der Wert von size mit minimalem errV[size]
      return Learner(best_size, examples)
```

```
function CROSS-VALIDATION(Learner, size, k, examples) returns zwei Werte:
         mittlere Fehlerrate der Trainingsmenge und
         mittlere Fehlerrate der Validierungsmenge

  fold_errT ← 0; fold_errV ← 0
  for fold = 1 to k do
    training_set, validation_set ← PARTITION(examples, fold, k)
    h ← Learner(size, training_set)
    fold_errT ← fold_errT + ERROR-RATE(h, training_set)
    fold_errV ← fold_errV + ERROR-RATE(h, validation_set)
  return fold_errT/k, fold_errV/k
```

Abbildung 18.8: Ein Algorithmus zur Auswahl des Modells, das die niedrigste Fehlerrate bei Validierungsdaten aufweist. Dazu werden Modelle zunehmender Komplexität erstellt und das Modell mit der besten empirischen Fehlerrate für die Validierungsdaten ausgewählt. Hier bedeutet *errT* die mittlere Fehlerrate für die Trainingsdaten und *errV* die mittlere Fehlerrate für die Validierungsdaten. *Learner*(*size*, *examples*) gibt eine Hypothese zurück, deren Komplexität durch den Parameter *size* festgelegt ist und die mit den Beispielen trainiert wurde. PARTITION(*examples*, *fold*, *k*) teilt die Beispiele in zwei Teilmengen auf: eine Validierungsmenge der Größe N/k und eine Trainingsmenge mit allen anderen Beispielen. Die Aufteilung unterscheidet sich für jeden Wert von *fold*.

▶ Abbildung 18.9 zeigt typische Kurven: Der Fehler der Trainingsmenge nimmt monoton ab (obwohl es im Allgemeinen eine leicht zufällige Variation geben kann), während der Fehler der Validierungsmenge zunächst zurückgeht und dann wieder ansteigt, wenn das Modell in den Bereich der Überanpassung kommt. Die Prozedur der Kreuzvalidierung greift den *size*-Wert mit dem geringsten Fehler der Validierungsmenge heraus; das ist der unterste Teil der U-förmigen Kurve. Dann generieren wir eine Hypothese dieses *size*-Wertes mit allen Daten (ohne irgendwelche Daten auszuschließen). Natürlich sollten wir schließlich die zurückgegebene Hypothese an einer separaten Testmenge bewerten. Dieser Ansatz setzt voraus, dass der Lernalgorithmus einen Parameter – *size* – übernimmt und eine Hypothese dieser Größe zurückgibt. Wie bereits erwähnt, kann die Größe beim Lernen von Entscheidungsbäumen die Anzahl der Knoten sein. Wir können DECISION-TREE-LEARNER so modifizieren, dass der Algorithmus die Anzahl der Knoten als Eingabe übernimmt, den Baum für Breitensuche und nicht für Tiefensuche erstellt (aber auf jeder Ebene trotzdem das Attribut mit dem höchsten Gewinn auswählt) und anhält, wenn er die gewünschte Anzahl von Knoten erreicht.

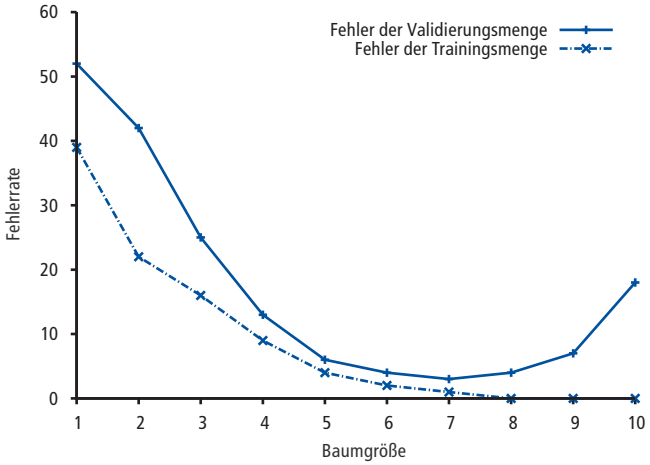


Abbildung 18.9: Fehlerraten bei Trainingsdaten (untere, gestrichelte Linie) und Validierungsdaten (obere, durchgängige Linie) für unterschiedlich große Entscheidungsbäume. Wir halten an, wenn die Fehlerrate der Trainingsmenge asymptotisch gegen null geht, und wählen dann den Baum mit minimaler Fehlerrate für die Validierungsmenge – hier den Baum mit der Größe von 7 Knoten.

18.4.2 Von Fehlerraten zum Verlust

Bislang haben wir versucht, die Fehlerrate zu minimieren. Das ist zweifellos besser, als die Fehlerrate zu maximieren, ist aber nur die halbe Wahrheit. Betrachten Sie das Problem, E-Mail-Nachrichten als Spam oder Nicht-Spam zu klassifizieren. Es ist schlimmer, Nicht-Spam als Spam zu klassifizieren (und dadurch eventuell eine wichtige Nachricht zu verpassen), als Spam als Nicht-Spam zu klassifizieren (was einige ärgerliche Sekunden beschert). Eine Klassifizierungsfunktion mit einer Fehlerrate von 1%, die bei fast allen Fehlern Spam als Nicht-Spam klassifiziert, wäre also besser als eine Klassifizierungsfunktion mit einer Fehlerrate von lediglich 0,5%, die bei den meisten dieser Fehler Nicht-Spam als Spam klassifiziert. Wie *Kapitel 16* gezeigt hat, sollten Entscheider den erwarteten Nutzen maximieren. Das Gleiche gilt für Lernalgorithmen. Beim maschinellen Lernen hat es sich eingebürgert, den Nutzen mithilfe einer **Verlustfunktion** auszudrücken. Die Verlustfunktion $L(x, y, \hat{y})$ ist definiert als die Größe des Nutzens, der durch Vorhersage von $h(x) = y$ verloren geht, wenn die korrekte Antwort $f(x) = y$ lautet:

$$L(x, y, \hat{y}) = \textit{Nutzen}(\text{Ergebnis der Verwendung von } y \text{ bei gegebener Eingabe } x) - \textit{Nutzen}(\text{Ergebnis der Verwendung von } \hat{y} \text{ bei gegebener Eingabe } x).$$

Dies ist die ganz allgemeine Formulierung der Verlustfunktion. Gebräuchlich ist auch die vereinfachte Version $L(y, \hat{y})$, die unabhängig von x ist und die wir im Rest dieses Kapitels verwenden. Das bedeutet, wir können nicht entscheiden, ob es schlechter ist, einen Brief von Mutter als einen Brief von unserem nervigen Cousin falsch zu klassifizieren, doch wir können sagen, dass es zehnmal schlechter ist, Nicht-Spam als Spam zu klassifizieren, als umgekehrt:

$$L(\textit{Spam}, \textit{Nicht-Spam}) = 1, \quad L(\textit{Nicht-Spam}, \textit{Spam}) = 10.$$

Beachten Sie, dass $L(y, \hat{y})$ immer null ist. Gemäß Definition gibt es keinen Verlust, wenn Sie genau richtig raten. Für Funktionen mit diskreten Ausgaben können wir einen Verslustwert für jede mögliche falsche Klassifizierung auflisten, für Realzahldaten ist dies jedoch nicht möglich. Wenn $f(x)$ gleich 137,035999 ist, könnten wir mit $h(x) = 137{,}036$ recht zufrieden sein, doch stellt sich die Frage, wie zufrieden? Im Allgemeinen sind kleine Fehler besser als größere; zwei Funktionen, die das gleiche Konzept implementieren, sind der absolute Wert der Differenz (namens L_1-Verlust) und das Quadrat der Differenz (namens L_2-Verlust). Wenn wir mit dem Konzept zufrieden sind, die Fehlerrate zu minimieren, können wir die $L_{0/1}$-Verlustfunktion verwenden, die einen Verlust von 1 für eine falsche Antwort hat und sich für Ausgaben mit diskreten Werten eignet:

Absoluter Wertverlust:	$L_1(y, \hat{y}) = \lvert y - \hat{y} \rvert$
Quadrierter Wertverlust:	$L_2(y, \hat{y}) = (y - \hat{y})^2$
0/1-Verlust:	$L_{0/1}(y, \hat{y}) = 0$, wenn $y = \hat{y}$, sonst 1.

Der lernende Agent kann theoretisch seinen erwarteten Nutzen maximieren, indem er die Hypothese wählt, die den erwarteten Verlust über alle ihm präsentierten Ein-/Ausgabe-Paare minimiert. Es ist bedeutungslos, über diese Erwartung zu sprechen, ohne eine A-priori-Wahrscheinlichkeitsverteilung $\mathbf{P}(X, Y)$ über den Beispielen zu definieren. Es sei $\mathcal{E}$ die Menge aller möglichen Ein-/Ausgabe-Beispiele. Der erwartete **Generalisierungsverlust** für eine Hypothese h (in Bezug auf die Verlustfunktion L) beträgt

$$GenVerlust_L(h) = \sum_{(x,y)\in\mathcal{E}} L\big(y, h(x)\big)P\big(x, y\big)$$

und die beste Hypothese h^* ist diejenige mit dem minimalen erwarteten Generalisierungsverlust:

$$h^* = \operatorname*{argmin}_{h\in\mathcal{H}} GenVerlust_L(h)$$

Da $P(x, y)$ nicht bekannt ist, kann der lernende Agent den Generalisierungsverlust lediglich mit einem empirischen Verlust auf einer Menge von Beispielen E abschätzen:

$$EmpVerlust_{L,E}(h) = \frac{1}{N}\sum_{(x,y)\in E} L\big(y, h(x)\big)1.$$

Die geschätzte beste Hypothese h^* ist dann diejenige mit dem minimalen empirischen Verlust:

$$h^* = \operatorname*{argmin}_{h\in\mathcal{H}} EmpVerlust_{L,E}(h).$$

Es gibt vier Gründe, warum sich h^* von der wahren Funktion f unterscheiden kann: Nichtrealisierbarkeit, Varianz, Rauschen und Berechnungskomplexität. Erstens kann f nicht realisierbar – eventuell nicht in $\mathcal{H}$ enthalten – sein oder in einer derartigen Weise vorliegen, dass andere Hypothesen bevorzugt werden. Zweitens gibt ein Lernalgorithmus verschiedene Hypothesen für unterschiedliche Mengen von Beispielen zurück, selbst wenn diese Mengen aus derselben wahren Funktion f bezogen werden, und diese Hypothesen liefern verschiedene Voraussagen für neue Beispiele. Je höher die Varianz unter den Voraussagen ist, desto höher ist die Wahrscheinlichkeit für einen signifikanten Feh-

ler. Doch selbst wenn das Problem realisierbar ist, gibt es eine zufällige Varianz, wobei aber diese Varianz gegen null geht, wenn die Anzahl der Trainingbeispiele zunimmt. Drittens kann f nichtdeterministisch oder **verrauscht** sein – für jedes Auftreten von x werden eventuell verschiedene Werte für $f(x)$ zurückgegeben. Gemäß Definition lässt sich Rauschen nicht vorhersagen; in vielen Fällen tritt es auf, weil die beobachteten Beschriftungen y das Ergebnis von Attributen der Umgebung sind, die in x nicht aufgelistet sind. Und wenn schließlich $\mathcal{H}$ komplex ist, ist es rechentechnisch eventuell nicht möglich, den gesamten Hypothesenraum systematisch zu durchsuchen. Hier können wir bestenfalls eine lokale Suche durchführen (Hill-Climbing oder gierige Suche), die nur einen Teil des Raumes untersucht. Damit erhalten wir einen Annäherungsfehler. Kombiniert man die Fehlerquellen, bleibt die Schätzung einer Annäherung der wahren Funktion f übrig.

Herkömmliche Methoden in der Statistik und frühe Arbeiten zum maschinellen Lernen haben sich auf Lernen in kleinem Umfang konzentriert, wobei die Anzahl der Trainingsbeispiele von Dutzenden bis zu wenigen Tausenden reichte. Hier stammt der Generalisierungsfehler vor allem vom Annäherungsfehler, nicht die wahre Funktion f im Hypothesenraum zu haben, und vom Schätzungsfehler, wenn nicht genügend Trainingsbeispiele vorliegen, um die Varianz zu begrenzen. In den letzten Jahren hat sich der Schwerpunkt auf das Lernen im großen Maßstab mit oftmals Millionen von Beispielen verschoben. Hier wird der Generalisierungsfehler durch die rechentechnischen Grenzen dominiert: Es gibt genügend Daten und ein genügend umfangreiches Modell, sodass wir eine Hypothese h finden könnten, die sehr nahe an der wahren Funktion f liegt, die Berechnung der Suche jedoch zu komplex ist, sodass wir uns mit einer suboptimalen Annäherung begnügen.

18.4.3 Regularisierung

Abschnitt 18.4.1 hat erläutert, wie die Modellauswahl mit Kreuzvalidierung in Bezug auf die Modellgröße erfolgt. Alternativ kann man nach einer Hypothese suchen, die direkt die gewichtete Summe des empirischen Verlustes und die Komplexität der Hypothese minimiert, was wir als die Gesamtkosten bezeichnen:

$$Kosten(h) = EmpVerlust(h) + \lambda\, Komplexität(h)$$
$$h^* = \operatorname*{argmin}_{h \in \mathcal{H}} Kosten(h).$$

Hier ist der Parameter λ eine positive Zahl, die als Konvertierungsfaktor zwischen Verlust und Hypothesenkomplexität dient (die schließlich nicht auf der gleichen Skala gemessen werden). Dieser Ansatz fasst Verlust und Komplexität zu einer Metrik zusammen, sodass wir die beste Hypothese auf einmal finden können. Leider brauchen wir trotzdem noch eine Kreuzvalidierungssuche, um die Hypothese zu finden, die am besten verallgemeinert. Dieses Mal geschieht dies aber mit verschiedenen Werten von λ und nicht mit *size*. Wir wählen den Wert von λ aus, der uns die beste Bewertung für die Validierungsmenge liefert.

Diesen Prozess der expliziten Bestrafung komplexer Hypothesen bezeichnet man als **Regularisierung** (da er nach einer regelmäßigeren – oder weniger komplexen – Funktion sucht). Für die Kostenfunktion müssen wir jedoch zwei Festlegungen treffen: die Verlustfunktion und das Komplexitätsmaß, das als Regularisierungsfunktion bezeichnet wird. Die Auswahl der Regularisierungsfunktion hängt vom Hypothesenraum ab. Zum Beispiel ist eine gute Regularisierungsfunktion für Polynome die Quadratsumme der

Koeffizienten – wenn wir eine kleine Summe anstreben, bringt uns das von den wellenförmigen Polynomen in Abbildung 18.1(b) und (c) weg. *Abschnitt 18.6* zeigt ein Beispiel für diesen Typ der Regularisierung.

Modelle lassen sich auch vereinfachen, indem man die Dimensionen reduziert, mit denen die Modelle arbeiten. Mithilfe der sogenannten **Feature Selection** (**Merkmalsauswahl**) lassen sich Attribute verwerfen, die irrelevant zu sein scheinen. Die χ^2-Kürzung ist eine Art der Feature Selection.

Es ist in der Tat möglich, den empirischen Verlust und die Komplexität auf der gleichen Skala ohne den Umrechnungsfaktor λ zu messen: Beide Werte lassen sich in Bit ausdrücken. Zunächst wird die Hypothese als Turing-Maschinen-Programm kodiert und die Anzahl der Bits gezählt. Dann zählt man die Anzahl der erforderlichen Bits, um die Daten zu kodieren, wobei ein korrekt vorhergesagtes Beispiel null Bit kostet und die Kosten für ein falsch vorhergesagtes Beispiel von der Größe des Fehlers abhängen. Die **MDL**-Hypothese (**Minimum Description Length**, **Minimale Beschreibungslänge**) minimiert die Gesamtanzahl der erforderlichen Bits. Im Grenzbereich funktioniert dies gut, jedoch gibt es bei kleineren Problemen eine Schwierigkeit, weil die Wahl der Kodierung des Programms – zum Beispiel wie man einen Entscheidungsbaum am besten als Bit-String kodiert – das Ergebnis beeinflusst. In *Abschnitt 20.2.3* beschreiben wir eine probabilistische Interpretation des MDL-Ansatzes.

18.5 Theorie des Lernens

Die wichtigste nicht beantwortete Frage zum Lernen ist: Wie kann man sicher sein, dass ein Lernalgorithmus eine Hypothese produziert hat, die für bislang nicht bekannte Eingaben korrekte Werte vorhersagt? Formal ausgedrückt, wie wissen wir, dass die Hypothese h nahe der Zielfunktion f liegt, wenn wir nicht wissen, was f ist? Diese Fragen wurden mehrere Jahrhunderte lang überdacht. In den letzten Jahrzehnten sind andere Fragen entstanden: Wie viele Beispiele benötigen wir, um eine gute h zu erhalten? Können wir in einem sehr komplexen Hypothesenraum überhaupt die beste h finden oder müssen wir uns mit einem lokalen Maximum im Hypothesenraum begnügen? Wie komplex sollte h sein? Wie vermeiden wir Überanpassung? Dieser Abschnitt befasst sich mit derartigen Fragen.

Tipp

Wir gehen zunächst der Frage nach, wie viele Beispiele für das Lernen erforderlich sind. Aus der Lernkurve für das Lernen von Entscheidungsbäumen im Restaurantproblem (Abbildung 18.7) geht hervor, dass sie umso besser wird, je mehr Trainingsdaten wir verwenden. Lernkurven sind nützlich, aber auch spezifisch für einen bestimmten Lernalgorithmus in einem bestimmten Problem. Gibt es generellere Prinzipien, die die Anzahl der benötigten Beispiele im Allgemeinen regeln? Mit derartigen Fragen beschäftigt sich die **Computer-Lerntheorie** im Schnittpunkt von KI, Statistik und theoretischer Informatik. Ihr liegt das Prinzip zugrunde, dass *jede Hypothese, die gänzlich falsch ist, mit hoher Wahrscheinlichkeit nach einer kleinen Anzahl an Beispielen „erkannt" wird, weil sie eine falsche Vorhersage trifft. Somit ist jede Hypothese, die mit einer ausreichend großen Menge von Trainingsbeispielen konsistent ist, sehr wahrscheinlich nicht gänzlich falsch, d.h., sie muss* ***wahrscheinlich annähernd richtig*** *sein*. Jeder Lernalgorithmus, der Hypothesen zurückgibt, die wahrscheinlich annähernd richtig sind, wird als **PAC-Lernalgorithmus** (für die Abkürzung der englischen Bezeichnung **Probably Approximately Correct**) bezeichnet.

Theoreme des PAC-Lernens sind wie alle Theoreme logische Konsequenzen von Axiomen. Wenn ein *Theorem* (im Unterschied etwa zu einem politischen Experten) etwas über die Zukunft basierend auf der Vergangenheit aussagt, müssen die Axiome den „Kraftstoff" liefern, um diese Verbindung herzustellen. Beim PAC-Lernen kommt der Kraftstoff von der Stationaritätsannahme (siehe *Abschnitt 18.4*), die besagt, dass zukünftige Beispiele aus derselben festen Verteilung $\mathbf{P}(E) = \mathbf{P}(X, Y)$ wie Beispiele aus der Vergangenheit gezogen werden. (Dabei müssen wir nicht wissen, wie diese Verteilung aussieht, nur, dass sie sich nicht ändert.) Um die Dinge einfach zu halten, nehmen wir weiterhin an, dass die wahre Funktion f deterministisch ist und zur betrachteten Hypothesenklasse $\mathcal{H}$ gehört.

Die einfachsten PAC-Theoreme haben mit booleschen Funktionen zu tun, für die der 0/1-Verlust passend ist. Die weiter oben formell definierte **Fehlerrate** einer Hypothese h definieren wir hier als erwarteten Generalisierungsfehler für Beispiele, die aus der stationären Verteilung stammen:

$$\text{fehler}(h) = \mathit{GenVerlust}_{L_{0/1}}(h) = \sum_{x,y} L_{0/1}(y, h(x)) P(x, y).$$

Mit anderen Worten ist *fehler*(h) die Wahrscheinlichkeit, dass h ein neues Beispiel falsch klassifiziert. Dies ist die gleiche Größe, die experimentell durch die weiter vorn gezeigten Lernkurven gemessen wurde.

Eine Hypothese h wird als **annähernd richtig** bezeichnet, wenn $\mathit{fehler}(h) \leq \epsilon$, wobei ϵ eine kleine Konstante ist. Wir werden zeigen, dass sich ein N finden lässt, sodass nach der Betrachtung von N Beispielen alle konsistenten Hypothesen mit hoher Wahrscheinlichkeit annähernd richtig sind. Man kann sich eine annähernd korrekte Hypothese als „eng" an der wahren Funktion im Hypothesenraum vorstellen: Sie liegt innerhalb der sogenannten ϵ-**Kugel** um die wahre Funktion f. Der Hypothesenraum außerhalb dieser Kugel wird als $\mathcal{H}_{falsch}$ bezeichnet.

Wir können wie folgt die Wahrscheinlichkeit berechnen, dass eine „grundsätzlich falsche" Hypothese $h_b \in \mathcal{H}_{falsch}$ mit den ersten N Beispielen konsistent ist. Wir wissen, dass $\mathit{fehler}(h_b) > \epsilon$ ist. Damit ist die Wahrscheinlichkeit, dass sie mit einem bestimmten Beispiel übereinstimmt, mindestens $1 - \epsilon$. Da die Beispiele unabhängig sind, ist die Grenze für N Beispiele:

$$P(h_b \text{ stimmt mit } N \text{ Beispielen überein}) \leq (1 - \epsilon)^N.$$

Die Wahrscheinlichkeit, dass $\mathcal{H}_{falsch}$ mindestens eine konsistente Hypothese enthält, ist durch die Summe der Einzelwahrscheinlichkeiten begrenzt:

$$P(\mathcal{H}_{falsch} \text{ enthält eine konsistente Hypothese}) \leq |\mathcal{H}_{falsch}|(1 - \epsilon)^N \leq |\mathcal{H}|(1 - \epsilon)^N.$$

Dafür haben wir die Tatsache genutzt, dass $|\mathcal{H}_{falsch}| \leq |\mathcal{H}|$. Wir würden die Wahrscheinlichkeit dieses Ereignisses gerne unter eine kleine Zahl δ reduzieren:

$$|\mathcal{H}|(1 - \epsilon)^N \leq \delta.$$

Für $1 - \epsilon \leq e^{-\epsilon}$ können wir dies erreichen, wenn wir es dem Algorithmus erlauben,

$$N \geq \frac{1}{\epsilon}\left(\ln\frac{1}{\delta} + \ln|\mathcal{H}|\right). \tag{18.1}$$

Beispiele zu sehen. Wenn ein Lernalgorithmus also eine Hypothese zurückgibt, die mit so vielen Beispielen konsistent ist, dann hat er mit einer Wahrscheinlichkeit von

mindestens $1 - \delta$ einen Fehler von höchstens ϵ. Mit anderen Worten, er ist wahrscheinlich annähernd richtig. Die Anzahl der erforderlichen Beispiele, als Funktion von ϵ und δ, wird als **Stichprobenkomplexität** des Hypothesenraumes bezeichnet.

Wie wir bereits gesehen haben, ist $|\mathcal{H}| = 2^{2^n}$, wenn $\mathcal{H}$ die Menge aller booleschen Funktionen mit n Attributen ist. Damit wächst die Stichprobenkomplexität des Raumes zu 2^n an. Weil die Anzahl möglicher Beispiele ebenfalls 2^n ist, setzt PAC-Lernen in der Klasse aller booleschen Funktionen voraus, alle – oder nahezu alle – möglichen Beispiele zu sehen. Eine genauere Betrachtung offenbart den Grund hierfür: $\mathcal{H}$ enthält genügend Hypothesen, um jeden gegebenen Satz von Beispielen in allen möglichen Arten zu klassifizieren. Insbesondere gilt für jede Menge von N Beispielen, dass die Menge der Hypothesen, die mit diesen Beispielen konsistent ist, gleich viele Hypothesen enthält, die x_{N+1} als positiv und die x_{N+1} als negativ voraussagen.

Um wirkliche Generalisierung von unbekannten Beispielen zu erhalten, müssen wir scheinbar den Hypothesenraum $\mathcal{H}$ irgendwie einschränken. Doch wenn wir das tun, eliminieren wir natürlich auch die wahre Funktion überhaupt. Es gibt drei Möglichkeiten, diesem Dilemma zu entkommen. Die erste behandeln wir in *Kapitel 19*: A-priori-Wissen in das Problem einbringen. Bei der zweiten, die *Abschnitt 18.4.3* eingeführt hat, besteht man darauf, dass der Algorithmus nicht einfach irgendeine konsistente Hypothese zurückgibt, sondern vorzugsweise eine einfache (wie es beim Lernen von Entscheidungsbäumen geschieht). Ist es handhabbar, eine einfache konsistente Hypothese zu finden, sind die Ergebnisse der komplexen Stichprobe im Allgemeinen besser als für Analysen, die nur auf Konsistenz basieren. Beim dritten Ausweg, den wir als Nächstes verfolgen, konzentrieren wir uns auf lernbare Teilmengen des gesamten Hypothesenraumes boolescher Funktionen. Dieses Konzept stützt sich auf die Annahme, dass die eingeschränkte Sprache eine Hypothese h enthält, die eng genug an der wahren Funktion f liegt. Das hat den Vorzug, dass der eingeschränkte Hypothesenraum eine effektive Generalisierung ermöglicht und normalerweise einfacher zu durchsuchen ist. Eine derartige eingeschränkte Sprache betrachten wir nun genauer.

18.5.1 Beispiel für PAC-Lernen: Entscheidungslisten lernen

Wir zeigen nun, wie sich PAC-Lernen auf einen neuen Hypothesenraum anwenden lässt: **Entscheidungslisten**. Eine Entscheidungsliste besteht aus einer Reihe von Tests, bei denen es sich um eine Konjunktion von Literalen handelt. Ist ein auf eine Beispielbeschreibung angewendeter Test erfolgreich, gibt die Entscheidungsliste den Wert an, der zurückgegeben werden soll. Schlägt der Test fehlt, wird die Verarbeitung mit dem nächsten Test in der Liste fortgesetzt. Entscheidungslisten erinnern an Entscheidungsbäume, aber ihre Gesamtstruktur ist einfacher: Sie verzweigen nur in eine Richtung. Im Gegensatz dazu sind die einzelnen Tests komplexer. ▶ Abbildung 18.10 zeigt eine Entscheidungsliste, die die folgende Hypothese darstellt:

$$WerdenWarten \Leftrightarrow (Gäste = Einige) \vee (Gäste = Voll \wedge Frei/Sams)$$

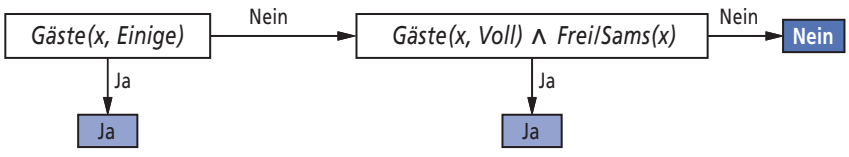


Abbildung 18.10: Eine Entscheidungsliste für das Restaurantproblem.

Wenn wir Tests beliebiger Größe durchführen dürfen, können Entscheidungslisten jede boolesche Funktion darstellen (Übung 18.16). Schränken wir jedoch die Größe jedes Testes auf höchstens k Literale ein, ist es möglich, dass der Lernalgorithmus aus einer kleineren Anzahl an Beispielen erfolgreich verallgemeinert. Wir bezeichnen diese Sprache auch als **k-DL** (die Abkürzung „DL“ steht für Decision List, Entscheidungsliste). Das Beispiel in Abbildung 18.10 ist in 2-DL dargestellt. Man kann ganz einfach zeigen (Übung 18.16), dass k-DL eine Untermenge der Sprache **k-DT** (die Abkürzung „DT“ steht für Decision Tree, Entscheidungsbaum) enthält, der Menge aller Entscheidungsbäume mit einer Tiefe von höchstens k. Beachten Sie, dass die jeweilige Sprache, auf die sich k-DL bezieht, von den für die Beschreibung der Beispiele verwendeten Attributen abhängig ist. Mit der Notation k-DL(n) bezeichnen wir eine k-DL-Sprache, die n boolesche Attribute verwendet.

Die erste Aufgabe ist, zu zeigen, dass k-DL erlernbar ist – d.h., jede Funktion in k-DL kann annähernd genau sein, nachdem sie für eine sinnvolle Anzahl an Beispielen trainiert wurde. Dazu müssen wir die Anzahl der Hypothesen in der Sprache berechnen. Sei die Sprache der Tests $Conj(n, k)$ – Konjunktionen von höchstens k Literalen unter Verwendung von n Attributen. Weil eine Entscheidungsliste sich aus Tests zusammensetzt und jedem Test das Ergebnis *Ja* oder *Nein* zugeordnet werden kann oder weil er in der Entscheidungsliste fehlen kann, gibt es höchstens $3^{|Conj(n,k)|}$ verschiedene Mengen von Komponententests. Jede dieser Testmengen kann in beliebiger Reihenfolge vorliegen, deshalb gilt:

$$|k\text{–DL}(n)| \leq 3^{|Conj(n,k)|} \, |Conj(n,k)|!$$

Die Anzahl der Konjunktionen von k Literalen aus n Attributen ist gegeben durch:

$$\left|Conj(n,k)\right| = \sum_{i=0}^{k} \binom{2n}{i} = O\left(n^k\right).$$

Nach einiger Arbeit erhalten wir:

$$\left|k - \text{DL}(n)\right| = 2^{O\left(n^k \log_2\left(n^k\right)\right)}.$$

Dies können wir in Gleichung (18.1) einsetzen, um zu zeigen, dass die für das PAC-Lernen einer k-DL-Funktion benötigte Anzahl an Beispielen polynomial in n ist:

$$N \geq \frac{1}{\in}\left(\ln\frac{1}{\delta} + O\left(n^k \log_2\left(n^k\right)\right)\right).$$

Aus diesem Grund kann jeder Algorithmus, der eine konsistente Entscheidungsliste zurückgibt, eine k-DL-Funktion für kleine k in einer sinnvollen Anzahl an Beispielen PAC-lernen.

Die nächste Aufgabe ist, einen effizienten Algorithmus zu finden, der eine konsistente Entscheidungsliste zurückgibt. Wir verwenden einen „gierigen“ Algorithmus, DECISION-LIST-LEARNING, der wiederholt einen Test findet, der genau mit derselben Untermenge der Trainingsmenge übereinstimmt. Anschließend erzeugt er die restliche Entscheidungsliste unter Verwendung der restlichen Beispiele. Dies wird wiederholt, bis keine weiteren Beispiele übrig sind. ▶ Abbildung 18.11 zeigt diesen Algorithmus.

```
function DECISION-LIST-LEARNING(examples) returns eine
         Entscheidungsliste oder Fehler

   if examples ist leer then return die triviale Entscheidungsliste No
   t ← ein Test, der mit einer nicht leeren Untermenge examples_t von
        examples übereinstimmt, sodass die Elemente von examples_t alle
        positiv oder alle negativ sind
   if es gibt kein solches t then return Fehler
   if die Beispiele in examples_t sind positiv then o ← Ja else o ← Nein
   return eine Entscheidungsliste mit dem Anfangstest t und dem
          Ergebnis o und den restlichen Tests, die gegeben sind als
      DECISION-LIST-LEARNING(examples - examples_t)
```

Abbildung 18.11: Ein Algorithmus für das Lernen von Entscheidungslisten.

Dieser Algorithmus spezifiziert nicht die Methode für die Auswahl des nächsten Testes, der der Entscheidungsliste hinzugefügt wird. Obwohl die zuvor angegebenen formalen Ergebnisse nicht von der ausgewählten Methode abhängig sind, scheint es sinnvoll, kleine Tests zu bevorzugen, die mit großen Mengen einheitlich klassifizierter Beispiele übereinstimmen, sodass die gesamte Entscheidungsliste so kompakt wie möglich wird. Die einfachste Strategie ist, den kleinsten Test t zu ermitteln, der mit jeder einheitlich klassifizierten Untermenge übereinstimmt, unabhängig von der Größe der Untermenge. Selbst dieser Ansatz funktioniert ausreichend gut, wie in ▶ Abbildung 18.12 gezeigt.

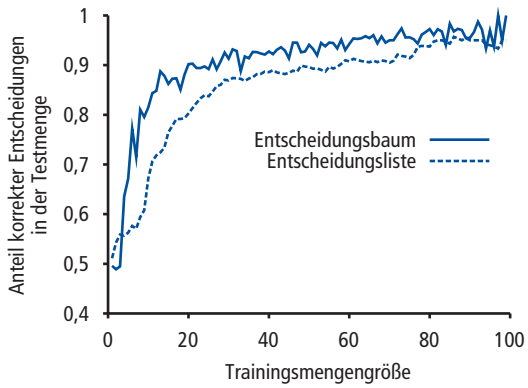


Abbildung 18.12: Lernkurve für den Algorithmus DECISION-LIST-LEARNING zu den Restaurantdaten. Zum Vergleich ist auch die Kurve für DECISION-TREE-LEARNING gezeigt.

18.6 Regression und Klassifizierung mit linearen Modellen

Wir gehen nun von Entscheidungsbäumen und -listen zu einem anderen Hypothesenraum über, und zwar einem, der seit Hunderten von Jahren verwendet wird: die Klasse der **linearen Funktionen** mit stetigen Eingabewerten. Wir beginnen mit dem einfachsten Fall: Regression mit einer univariaten linearen Funktion, die auch als „Anpassung einer Geraden“ bekannt ist. *Abschnitt 18.6.2* behandelt den multivariaten Fall. Die *Abschnitte 18.6.3* und *18.6.4* zeigen, wie sich lineare Funktionen in Klassifikatoren überführen lassen, indem man harte und weiche Schwellen anwendet.

18.6.1 Univariate lineare Regression

Eine univariate lineare Funktion (eine Gerade) mit Eingabe x und Ausgabe y hat die Form $y = w_1x + w_0$, wobei w_0 und w_1 reelle Koeffizienten sind, die gelernt werden sollen. Für die Koeffizienten sind die Buchstaben w oder g üblich, da man die Koeffizienten als **Gewichte** (engl. **Weights**) auffasst; der Wert von y wird geändert, indem das relative Gewicht des einen oder anderen Terms geändert wird. Wir definieren $\mathbf{w}$ als Vektor $[w_0, w_1]$ und

$$h\mathrm{w}(x) = w_1x + w_0.$$

▶ Abbildung 18.13(a) zeigt ein Beispiel für eine Trainingsmenge von n Punkten in der x, y-Ebene, wobei jeder Punkt die Größe in Quadratmeter und den Preis eines zum Verkauf angebotenen Hauses darstellt. Die Aufgabe, diejenige hw zu finden, die den Daten am besten entspricht, bezeichnet man als **lineare Regression**. Um die Anpassung an die Daten mit einer Geraden zu erreichen, müssen wir lediglich die Werte der Gewichte $[w_0, w_1]$ finden, die den empirischen Verlust minimieren. Es hat sich (zurückgehend auf Gauß[3]) eingebürgert, die quadratische Verlustfunktion L_2 zu verwenden, wobei über alle Trainingsbeispiele summiert wird:

$$Verlust(h_{\mathbf{w}}) = \sum_{j=1}^{N} L_2\left(y_j, h_{\mathbf{w}}\left(x_j\right)\right) = \sum_{j=1}^{N}\left(y_j - h_{\mathbf{w}}\left(x_j\right)\right)^2 = \sum_{j=1}^{N}\left(y_j - \left(w_1x_j + w_0\right)\right)^2$$

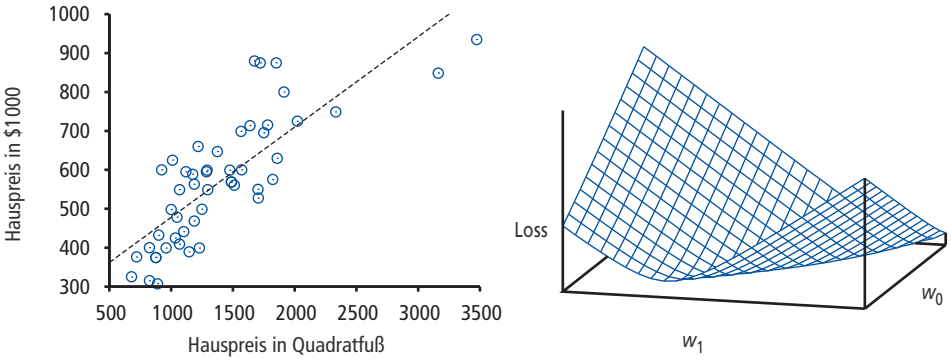


Abbildung 18.13: (a) Datenpunkte für Preis über Grundfläche von Häusern, die in Berkeley, CA, im Juli 2009 zum Verkauf standen, zusammen mit der linearen Funktionshypothese, die den quadratischen Fehler minimiert: $y = 0{,}232x + 246$. (b) Grafik der Verlustfunktion $\Sigma_j(w_1x_j + w_0 - y_j)^2$ für verschiedene Werte von w_0, w_1. Beachten Sie, dass die Verlustfunktion konvex ist und ein einzelnes globales Minimum aufweist.

Wir möchten $\mathbf{w}* = \mathtt{argmin}_w\ Verlust(h\mathrm{w})$ finden. Die Summe $\Sigma_{j=1}{}^{N}(y_j - (w_1x_j + w_0))^2$ nimmt ein Minimum an, wenn ihre partiellen Ableitungen bezüglich w_0 und w_1 null sind:

$$\frac{\partial}{\partial w_0}\sum_{j=1}^{N}\left(y_j - \left(w_1x_j + w_0\right)\right)^2 = 0 \quad \text{und} \quad \frac{\partial}{\partial w_1}\sum_{j=1}^{N}\left(y_j - \left(w_1x_j + w_0\right)\right)^2 = 0. \qquad (18.2)$$

3 Gauß hat gezeigt, dass man bei einem normal verteilten Rauschen der Werte y_j die wahrscheinlichsten Werte von w_1 und w_0 erhält, wenn man die Summe ihrer Fehlerquadrate minimiert.

Diese Gleichungen besitzen eine eindeutige Lösung:

$$w_1 = \frac{N\left(\sum x_j y_j\right) - \left(\sum x_j\right)\left(\sum y_j\right)}{N\left(\sum x_j^2\right) - \left(\sum x_j\right)^2}; \quad w_0 = \left(\sum y_j - w_1\left(\sum x_j\right)\right) / N. \tag{18.3}$$

Für das Beispiel in Abbildung 18.13(a) lautet die Lösung $w_1 = 0{,}232$ und $w_0 = 246$. Die Gerade mit diesen Gewichten ist in der Abbildung als gestrichelte Linie dargestellt.

Bei vielen Lernformen sind die Gewichte so anzupassen, dass ein minimaler Verlust entsteht. Es ist also hilfreich, eine bildliche Vorstellung davon zu haben, was im **Gewichtsraum** passiert – dem Raum, der durch alle möglichen Einstellungen der Gewichte definiert wird. Für univariate Regression ist der durch w_0 und w_1 definierte Gewichtsraum zweidimensional, sodass wir den Verlust als Funktion von w_0 und w_1 in einem 3D-Diagramm (siehe ▶ Abbildung 18.13(b)) grafisch darstellen können. Die Verlustfunktion ist **konvex**, wie in Abbildung 18.2 definiert. Dies gilt für jedes lineare Regressionsproblem mit einer L2-Verlustfunktion und impliziert, dass keine lokalen Minima existieren. In gewissem Sinne ist dies das Ende der Geschichte für lineare Modelle; wenn wir Geraden an Daten anpassen müssen, wenden wir Gleichung (18.3)[4] an.

Um über die linearen Modelle hinauszugehen, müssen wir uns der Tatsache stellen, dass die Gleichungen für das Verlustminimum (wie in Gleichung (18.2)) oftmals keine Lösung in geschlossener Form besitzen. Stattdessen sehen wir uns einem allgemeinen Optimierungsproblem mit Suche in einem stetigen Gewichtsraum gegenüber. Wie in *Abschnitt 4.2* erwähnt, eignet sich für derartige Probleme ein Hillclimbing-Algorithmus, der dem **Gradienten** der zu optimierenden Funktion folgt. Da wir in diesem Fall versuchen, den Verlust zu minimieren, verwenden wir den **Gradientenabstieg**. Wir wählen einen beliebigen Startpunkt im Gewichtsraum aus – hier einen Punkt in der (w_0, w_1)-Ebene – und gehen dann zu einem tiefer liegenden Nachbarpunkt. Dies wiederholen wir, bis wir beim minimal möglichen Verlust angekommen sind:

```
w ← ein beliebiger Punkt im Parameterraum
loop bis zur Konvergenz do
    for each w_i in w do
```

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} Verlust(\mathbf{w}). \tag{18.4}$$

Der Parameter α, den wir in *Abschnitt 4.2* **Schrittweite** genannt haben, wird normalerweise als **Lernrate** bezeichnet, wenn wir versuchen, den Verlust in einem Lernproblem zu minimieren. Er kann eine feste Konstante sein oder zeitlich mit fortschreitendem Lernprozess abnehmen.

Für univariate Regression ist die Verlustfunktion eine quadratische Funktion, sodass die partielle Ableitung eine lineare Funktion ergibt. (Dabei müssen Sie lediglich die Bezie-

4 Mit bestimmten Vorbehalten: Die L_2-Verlustfunktion ist geeignet, wenn es normal verteiltes Rauschen gibt, das unabhängig von x ist; alle Ergebnisse stützen sich auf die Stationaritätsannahme; etc.

hungen $\partial/\partial x\, x^2 = 2x$ und $\partial/\partial x\, x = 1$ kennen.) Zunächst erstellen wir die partiellen Ableitungen – die Anstiege – im vereinfachten Fall von nur einem Trainingsbeispiel (x, y):

$$\frac{\partial}{\partial w_i} Verlust(\mathbf{w}) = \frac{\partial}{\partial w_i}\big(y - h_{\mathbf{w}}(x)\big)^2$$

$$= 2\big(y - h_w(x)\big) \times \frac{\partial}{\partial w_i}\big(y - h_{\mathbf{w}}(x)\big)$$

$$= 2\big(y - h_w(x)\big) \times \frac{\partial}{\partial w_i}\big(y - \big(w_1 x + w_0\big)\big). \qquad (18.5)$$

Sowohl auf w_0 als auch w_1 angewendet, erhalten wir:

$$\frac{\partial}{\partial w_0} Verlust(\mathbf{w}) = -2\big(y - h_{\mathbf{w}}(x)\big); \qquad \frac{\partial}{\partial w_1} Verlust(\mathbf{w}) = -2\big(y - h_{\mathbf{w}}(x)\big) \times x.$$

Setzt man dies wieder in Gleichung (18.4) ein und bringt die 2 in die nicht spezifizierte Lernrate α ein, erhält man die folgende Lernregel für die Gewichte:

$$w_0 \leftarrow w_0 + \alpha\,(y - h\mathrm{w}\,(x)); \qquad w_1 \leftarrow w_1 + \alpha\,(y - h\mathrm{w}\,(x)) \times x.$$

Diese Aktualisierungen sind ohne Weiteres verständlich: Wenn $h\mathrm{w}(x) > y$, d.h. die Ausgabe der Hypothese zu groß ist, verringern wir w_0 ein wenig, und verringern w_1, wenn x eine positive Eingabe ist, erhöhen w_1 jedoch, wenn x eine negative Eingabe ist.

Die obigen Gleichungen decken genau ein Trainingsbeispiel ab. Für N Trainingsbeispiele wollen wir die Summe der Einzelverluste für jedes Beispiel minimieren. Da die Ableitung einer Summe die Summe der Ableitungen ist, erhalten wir:

$$w_0 \leftarrow w_0 + \alpha \sum_j \big(y_j - h_{\mathbf{w}}(x_j)\big); \qquad w_1 \leftarrow w_1 + \alpha \sum_j \big(y_j - h_{\mathbf{w}}(x_j)\big) \times x_j.$$

Diese Aktualisierungen bilden die Lernregel **Batch-Gradientenabstieg** für univariate lineare Regression. Die Konvergenz gegen das eindeutige globale Minimum ist garantiert (sofern wir α genügend klein wähleZPn), kann aber sehr langsam sein: Wir müssen für jeden Schritt sämtliche Trainingsdaten durchlaufen, wobei die Anzahl der Schritte recht hoch sein kann.

Als alternative Möglichkeit betrachten wir beim sogenannten **stochastischen Gradientenabstieg** nur jeweils einen einzelnen Trainingspunkt und führen nach jedem Punkt einen Schritt unter Verwendung von Gleichung (18.5) aus. Stochastischer Gradientenabstieg kann online eingesetzt werden, wo neue Daten einzeln nacheinander eintreffen, oder offline, wo wir dieselben Daten so oft wie nötig durchlaufen und einen Schritt ausführen, nachdem wir jedes einzelne Beispiel betrachtet haben. Dieses Verfahren ist oftmals schneller als der Gradientenabstieg in der Batch-Version. Allerdings ist bei einer festen Lernrate α die Konvergenz nicht garantiert; das Verfahren kann um das Minimum herum oszillieren, ohne zur Ruhe zu kommen. In manchen Fällen (mehr dazu später) garantiert ein Plan für fallende Lernraten (wie in Simulated Annealing), dass der Algorithmus konvergiert.

18.6.2 Multivariate lineare Regression

Wir können das Verfahren leicht auf **multivariate lineare Regressionsprobleme** erweitern, wobei jedes Beispiel $\mathbf{x}_j$ ein n-elementiger Vektor ist.[5] Unser Hypothesenraum ist die Menge von Funktionen der folgenden Form:

$$h_{sw}(\mathbf{x}_j) = w_0 + w_1 x_{j,1} + \ldots + w_n x_{j,n} = w_0 + \sum_i w_i x_{j,i}.$$

Der Term w_0, der den Schnittpunkt mit der y-Achse angibt, unterscheidet sich von den anderen Termen. Wir können dies korrigieren, indem wir ein Dummy-Eingabeattribut $x_{j,0}$ (ein sogenanntes **Bias-Gewicht**) einführen, das immer als 1 definiert wird. Dann ist h einfach das Punktprodukt der Gewichte und des Eingabevektors (oder – was das Gleiche ist – das Produkt von transponierter Gewichtsmatrix und Eingabevektor):

$$h_{sw}(\mathbf{x}_j) = \mathbf{w} \cdot \mathbf{x}_j = \mathbf{w}^\top \mathbf{x}_j = \sum_i w_i x_{j,i}.$$

Mit dem besten Gewichtsvektor $\mathbf{w}^*$ nimmt die Summe der Fehlerquadrate über den Beispielen ein Minimum an:

$$\mathbf{w}^* = \operatorname*{argmin}_{\mathbf{w}} \sum_j L_2(y_j, \mathbf{w} \cdot \mathbf{x}_j).$$

Multivariate lineare Regression ist eigentlich nicht wesentlich komplizierter als der eben besprochene univariate Fall. Gradientenabstieg erreicht das (eindeutige) Minimum der Verlustfunktion; die Aktualisierungsgleichung für jedes Gewicht w_i lautet:

$$w_i \leftarrow w_i + \alpha \sum_j x_{j,i}\left(y_j - h_w(\mathbf{x}_j)\right). \tag{18.6}$$

Es ist auch möglich, analytisch nach dem $\mathbf{w}$ aufzulösen, der den Verlust minimiert. Sei $\mathbf{y}$ der Vektor der Ausgaben für die Trainingsbeispiele und $\mathbf{X}$ die Datenmatrix, d.h. die Matrix der Eingaben mit einem n-dimensionalen Beispiel pro Zeile. Dann minimiert die Lösung

$$\mathbf{w}^* = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{y}$$

den quadratischen Fehler.

Bei univariater linearer Regression brauchen wir uns nicht um Überanpassung zu kümmern. Doch ist es bei multivariater linearer Regression in hochdimensionalen Räumen möglich, dass eine im Prinzip irrelevante Dimension zufällig nützlich erscheint, woraus Überanpassung resultiert.

Somit ist es üblich, **Regularisierung** für multivariate lineare Funktionen zu verwenden, um Überanpassung zu vermeiden. Wie bereits erwähnt, minimieren wir mit Regularisierung die Gesamtkosten einer Hypothese, wobei wir sowohl den empirischen Verlust als auch die Komplexität der Hypothese zählen:

$$\mathit{Kosten}(h) = \mathit{EmpVerlust}(h) + \lambda\, \mathit{Komplexität}(h).$$

5 In Anhang A finden Sie einen kurzen Überblick zur linearen Algebra.

Für lineare Funktionen lässt sich die Komplexität als Funktion der Gewichte spezifizieren. Wir können eine Familie von Regularisierungsfunktionen betrachten:

$$Komplexität(h_{\mathbf{w}}) = L_q(\mathbf{w}) = \sum_i |w_i|^q.$$

Wie bei Verlustfunktionen[6] haben wir mit $q = 1$ eine L_1-Regularisierung, die die Summe der absoluten Werte minimiert; mit $q = 2$ minimiert die L_2-Regularisierung die Quadratsumme. Welche Regularisierungsfunktion sollten Sie nun auswählen? Die Antwort hängt vom konkreten Problem ab, doch weist die L_1-Regularisierung einen wichtigen Vorzug auf: Das erzeugte Modell ist eher **schwach besetzt**. Das heißt, die Regularisierung setzt oftmals viele Gewichte auf null und deklariert damit letztlich die korrespondierenden Attribute als irrelevant – genau wie es bei DECISION-TREE-LEARNING der Fall ist (obwohl es sich um einen anderen Mechanismus handelt). Hypothesen, die Attribute verwerfen, sind für den Menschen verständlicher und eventuell weniger anfällig für Überanpassung. ▶ Abbildung 18.14 liefert eine intuitive Erläuterung, warum L_1-Regularisierung zu Gewichten von null führt, während das bei L_2-Regularisierung nicht der Fall ist. Beachten Sie, dass Minimierung von $Verlust(\mathbf{w}) + \lambda\, Komplexität(\mathbf{w})$ gleichbedeutend ist mit dem Minimieren von $Verlust(\mathbf{w})$ unter der Bedingung, dass $Komplexität(w) \leq c$ ist, wobei die Konstante c mit λ in Beziehung steht. In Abbildung 18.14(a) repräsentiert nun das Viereck die Menge der Punkte $\mathbf{w}$ im zweidimensionalen Gewichtsraum, deren L_1-Komplexität kleiner als c ist; unsere Lösung muss sich irgendwo innerhalb dieses Vierecks befinden. Die konzentrischen Ringe repräsentieren Konturen der Verlustfunktion, wobei der minimale Verlust in der Mitte liegt. Wir möchten den Punkt im Viereck finden, der dem Minimum am nächsten liegt. Wie aus dem Diagramm hervorgeht, ist es bei einer beliebigen Position des Minimums und seiner Konturen häufig so, dass die Ecke des Vierecks allein deshalb den Weg in die Nähe des Minimums findet, weil Ecken spitz sind. Und natürlich sind die Ecken die Punkte, bei denen ein Wert in einer bestimmten Richtung null ist. In Abbildung 18.14(b) haben wir das Gleiche für das L_2-Komplexitätsmaß realisiert, das einen Kreis und kein Viereck darstellt. Hier können Sie sehen, dass es im Allgemeinen keinen Grund gibt, dass der Schnittpunkt auf einer der Achsen erscheint; somit neigt L_2-Regularisierung nicht dazu, Gewichte von null zu produzieren. Im Ergebnis ist die Anzahl der erforderlichen Beispiele, um eine gute h zu finden, bei L_2-Regularisierung linear zur Anzahl der irrelevanten Features, bei L_1-Regularisierung jedoch nur logarithmisch. Diese Analyse wird bei vielen Problemen durch empirische Evidenz unterstützt.

Eine andere Betrachtungsweise hierfür ist, dass L_1-Regularisierung die Dimensionsachsen respektiert, während L_2 sie als willkürlich behandelt. Die L_2-Funktion ist sphärisch und damit rotationsinvariant: Stellen Sie sich eine Menge von Punkten in einer Ebene vor, die durch ihre x- und y-Koordinaten gemessen werden. Drehen Sie nun die Achsen um 45°. Damit erhalten Sie eine andere Menge von (x', y')-Werten, die dieselben Punkte darstellen. Wenn Sie die L_2-Regularisierung vor und nach der Drehung anwenden, erhalten Sie genau denselben Punkt als Antwort (auch wenn der Punkt mit den neuen (x', y')-Koordinaten beschrieben wird). Dies ist zweckmäßig, wenn die Wahl der Achsen wirklich willkürlich geschieht – wenn es keine Rolle spielt, ob es sich bei Ihren zwei Dimensionen um Entfernungen nach Norden und Osten handelt oder um die Distanzen Nord-Ost und Süd-

6 Es mag verwirrend sein, dass L_1 und L_2 sowohl für Verlustfunktionen als auch für Regularisierungsfunktionen verwendet werden. Allerdings muss das nicht paarweise geschehen: Man könnte L_2-Verlust mit L_1-Regularisierung verwenden oder umgekehrt.

Ost. Mit L_1-Regularisierung sieht die Antwort anders aus, da die L_1-Funktion nicht rotationsinvariant ist. Das ist angebracht, wenn die Achsen nicht austauschbar sind; es ist nicht sinnvoll, „Anzahl der Badezimmer“ um 45° gegen „Grundstücksgröße“ zu drehen.

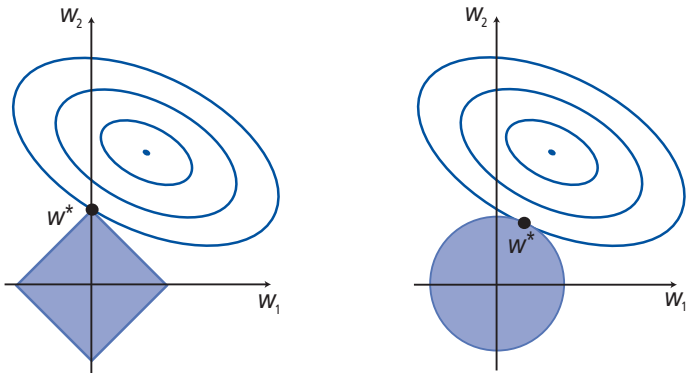


Abbildung 18.14: Warum L_1-Regularisierung ein schwach besetztes Modell hervorbringt. (a) Mit L_1-Regularisierung (Viereck) tritt der minimal erreichbare Verlust (konzentrische Konturen) oftmals auf einer Achse auf, was ein Gewicht von null bedeutet. (b) Mit L_2-Regularisierung (Kreis) tritt der minimale Verlust wahrscheinlich irgendwo auf dem Kreis auf, was keine Bevorzugung von null-Gewichten ergibt.

18.6.3 Lineare Klassifizierer mit Schwellenwertfunktion

Lineare Funktionen eignen sich sowohl zur Klassifizierung als auch für die Regression. Als Beispiel zeigt ▶ Abbildung 18.15(a) Datenpunkte zweier Klassen: Erdbeben (die für Seismologen von Interesse sind) und unterirdische Explosionen (für die sich Rüstungskontrollexperten interessieren). Jeder Punkt ist durch zwei Eingabewerte x_1 und x_2 für die Größen von Raum- und Oberflächenwellen definiert, die aus dem seismischen Signal berechnet werden. Mit diesen Trainingsdaten besteht die Aufgabe der Klassifizierung darin, eine Hypothese h zu lernen, die neue (x_1, x_2)-Punkte übernimmt und dann entweder 0 für Erdbeben oder 1 für Explosionen zurückgibt.

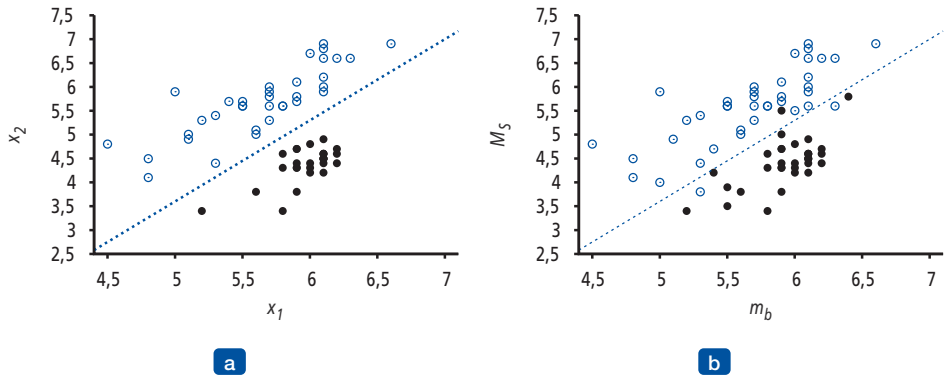


Abbildung 18.15: (a) Grafik der beiden seismischen Datenparameter Raumwelle mit der Größe x_1 und Oberflächenwelle mit der Größe x_2 für Erdbeben (weiße Kreise) und Kernexplosionen (schwarze Kreise), die zwischen 1982 und 1990 in Asien und im Nahen Osten aufgetreten sind (Kebeasy et al., 1998). Außerdem ist eine Entscheidungsgrenze zwischen den Klassen eingezeichnet. (b) Dieselbe Domäne mit mehr Datenpunkten. Die Erdbeben und Explosionen lassen sich nicht mehr linear trennen.

Eine **Entscheidungsgrenze** ist eine Linie (oder in höheren Dimensionen eine Fläche), die die beiden Klassen trennt. In Abbildung 18.15(a) ist die Entscheidungsgrenze eine Gerade. Eine lineare Entscheidungsgrenze wird als **linearer Separator** bezeichnet und die Daten, die ein derartiger Separator akzeptiert, als **linear separierbar**. In diesem Fall ist der lineare Separator durch

$$x_2 = 1{,}7x_1 - 4{,}9 \quad \text{oder} \quad -4{,}9 + 1{,}7x_1 - x_2 = 0$$

definiert. Die Explosionen, die wir mit dem Wert 1 klassifizieren wollen, befinden sich rechts von dieser Linie mit höheren Werten von x_1 und niedrigeren Werten von x_2. Demnach sind sie Punkte, für die $-4{,}9 + 1{,}7x_1 - x_2 > 0$ gilt, während Erdbeben durch $-4{,}9 + 1{,}7x_1 - x_2 < 0$ charakterisiert sind. Verwendet man die Konvention einer Dummy-Eingabe (Bias) $x_0 = 1$, können wir die Klassifikationshypothese wie folgt schreiben:

$$h\mathrm{w}(\mathbf{x}) = 1 \text{ if } \mathbf{w} \cdot \mathbf{x} \geq 0 \text{ und } 0 \text{ sonst.}$$

Alternativ können Sie sich h als das Ergebnis der Übergabe der linearen Funktion $\mathbf{w} \cdot \mathbf{x}$ über eine **Schwellenwertfunktion** vorstellen:

$h\mathrm{w}(\mathbf{x}) = \mathit{Schwellenwert}(\mathbf{w} \cdot \mathbf{x})$, wobei $\mathit{Schwellenwert}(z) = 1$, wenn $z \geq 0$ und 0 sonst.

▶ Abbildung 18.17(a) zeigt die Schwellenwertfunktion.

Da nun die Hypothese $h\mathrm{w}(\mathbf{x})$ eine genau definierte mathematische Form besitzt, können wir die Gewichte $\mathbf{w}$ auswählen, um den Verlust zu minimieren. In den *Abschnitten 18.6.1* und *18.6.2* haben wir dies sowohl in geschlossener Form (indem wir den Gradienten gleich null gesetzt und nach den Gewichten aufgelöst haben) als auch durch Gradientenabstieg im Gewichtsraum realisiert. Hier scheiden beide Verfahren aus, da der Gradient fast überall im Gewichtsraum null ist. Eine Ausnahme bilden lediglich die Punkte, an denen $\mathbf{w} \cdot \mathbf{x} = 0$ ist – und an diesen Punkten ist der Gradient nicht definiert.

Allerdings gibt es eine einfache Regel zur Gewichtsaktualisierung, die gegen eine Lösung konvergiert – d.h. ein linearer Separator, der die Daten perfekt klassifiziert –, sofern die Daten linear separierbar sind. Für ein einzelnes Beispiel $(\mathbf{x}, y)$ haben wir

$$w_i \leftarrow w_i + \alpha\,(y - h\mathrm{w}(\mathbf{x})) \times x_i\,. \tag{18.7}$$

Dies ist praktisch identisch mit Gleichung (18.6), der Aktualisierungsregel für lineare Regression! Aus den in *Abschnitt 18.7* noch zu erläuternden Gründen bezeichnet man diese Regel als **Perzeptron-Lernregel**. Da wir jedoch ein 0/1-Klassifizierungsproblem betrachten, ist das Verhalten etwas anders. Sowohl der wahre Wert y als auch die Ausgabe der Hypothese $h\mathrm{w}(\mathbf{x})$ sind entweder 0 oder 1, sodass es drei Möglichkeiten gibt:

- Wenn die Ausgabe richtig ist, d.h. $y = h\mathrm{w}(\mathbf{x})$, werden die Gewichte nicht geändert.
- Ist y gleich 1, aber $h\mathrm{w}(\mathbf{x})$ gleich 0, wird w_i *erhöht*, wenn der entsprechende Eingang x_i positiv ist, und *verringert*, wenn x_i negativ ist. Das ist sinnvoll, da wir $\mathbf{w} \cdot \mathbf{x}$ größer machen möchten, sodass $h\mathrm{w}(\mathbf{x})$ die Ausgabe 1 liefert.
- Ist y gleich 0, aber $h\mathrm{w}(\mathbf{x})$ gleich 1, wird w_i *verringert*, wenn die entsprechende Eingabe xi positiv ist, und *erhöht*, wenn x_i negativ ist. Dies ist sinnvoll, da wir $\mathbf{w} \cdot \mathbf{x}$ kleiner machen möchten, sodass $h\mathrm{w}(\mathbf{x})$ die Ausgabe 0 liefert.

Normalerweise wird die Lernregel auf jeweils ein Beispiel angewandt, wobei die Beispiele zufällig ausgewählt werden (wie beim stochastischen Gradientenabstieg). ▶ Abbildung 18.16(a) zeigt eine **Trainingskurve** für diese Lernregel, die auf die in Abbil-

dung 18.15(a) angegebenen Daten für Erdbeben/Explosionen angewandt wurde. Eine Trainingskurve misst die Klassifiziererleistung für eine feste Trainingsmenge bei fortschreitendem Prozess auf eben dieser Trainingsmenge. Die Kurve zeigt, wie die Aktualisierungsregel gegen einen linearen Separator mit dem Fehler null konvergiert. Die „Konvergenz" verläuft zwar nicht gleichmäßig, funktioniert aber immer. Dieser konkrete Lauf benötigt für eine Datenmenge mit 63 Beispielen 657 Schritte bis zum Konvergieren, sodass jedes Beispiel durchschnittlich ungefähr zehnmal präsentiert wird. Normalerweise ist die Variation über die Läufe sehr groß.

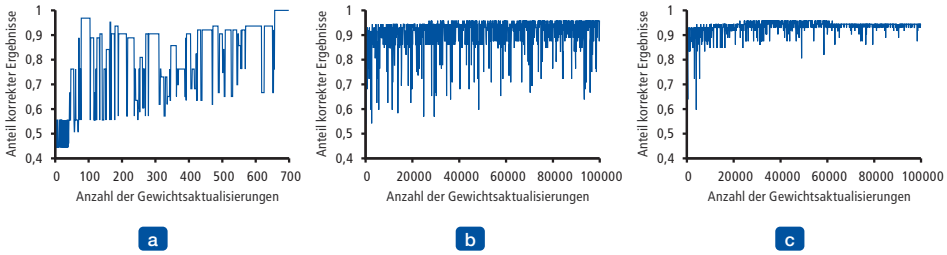


Abbildung 18.16: (a) Grafik der Gesamtgenauigkeit einer Trainingsmenge über der Anzahl der Iterationen durch die Trainingsmenge für die Perzeptron-Lernregel auf Basis der Daten von Erdbeben/Explosionen von Abbildung 18.15(a). (b) Die gleiche Grafik für die verrauschten, nicht separierbaren Daten von Abbildung 18.15(b). Beachten Sie die geänderte Skala der x-Achse. (c) Die gleiche Grafik wie in (b) mit einer geplanten Lernrate $\alpha(t) = 1000/(1000 + t)$.

Wie bereits erwähnt, konvergiert die Perzeptron-Lernregel gegen einen perfekten linearen Separator, wenn die Datenpunkte linear separierbar sind. Doch wie sieht es aus, wenn sie nicht linear separierbar sind? In der Praxis ist diese Situation häufig anzutreffen. Zum Beispiel sind in Abbildung 18.15(b) wieder die Datenpunkte eingefügt, die von Kebeasy et al. (1998) ausgelassen wurden, als sie die in Abbildung 18.15(a) gezeigten Daten grafisch dargestellt haben. In ▶ Abbildung 18.16(b) ist zu sehen, dass die Perzeptron-Lernregel selbst nach 10.000 Schritten nicht konvergiert: Obwohl sie die Lösung für minimale Fehler (drei Fehler) viele Male trifft, ändert der Algorithmus weiterhin die Gewichte. Im Allgemeinen konvergiert die Perzeptron-Regel nicht unbedingt zu einer stabilen Lösung bei einer festen Lernrate α, doch wenn α in $O(1/t)$ fällt, wobei t die Iterationsnummer ist, lässt sich zeigen, dass die Regel zu einer Minimalfehlerlösung konvergiert, wenn Beispiele in zufälliger Folge präsentiert werden. Außerdem lässt sich zeigen, dass das Ermitteln der Minimalfehlerlösung NP-hart ist.[7] Um also Konvergenz zu erreichen, ist davon auszugehen, dass viele Präsentationen der Beispiele erforderlich sind. Abbildung 18.16(b) zeigt den Trainingsprozess mit einer geplanten Lernrate $\alpha(t) = 1000/(1000 + t)$: Die Konvergenz ist nach 100.000 Iterationen zwar nicht perfekt, aber wesentlich besser als im Fall mit festem α.

18.6.4 Lineare Klassifizierung mit logistischer Regression

Wie bereits erläutert, entsteht ein linearer Klassifizierer, wenn die Ausgabe einer linearen Funktion über die Schwellenwertfunktion geleitet wird. Doch das Sprungverhalten des Schwellenwertes verursacht einige Probleme: Die Hypothese $h\mathrm{w}(\mathbf{x})$ ist nicht differenzierbar und stellt eine unstetige Funktion ihrer Eingaben und ihrer Gewichte dar; das

7 Vom technischen Standpunkt her setzen wir voraus, dass $\Sigma_{t=1}^{\infty} \alpha(t) = \infty$ und $\Sigma_{t=1}^{\infty} \alpha^2(t) < \infty$ ist. Der Abfall $\alpha(t) = O(1/t)$ erfüllt diese Bedingungen.

Lernen mit der Perzeptron-Regel wird dadurch zu einem recht unvorhersagbaren Abenteuer. Darüber hinaus annonciert der lineare Klassifizierer eine vollkommen zuverlässige Vorhersage von 1 oder 0, und zwar selbst für Beispiele, die sehr nahe an der Grenze liegen. In vielen Situationen brauchen wir feiner abgestufte Vorhersagen.

Alle diese Fragen lassen sich zum großen Teil durch eine weichere Schwellenwertfunktion auflösen – wobei der Schwellenwert durch eine stetige, differenzierbare Funktion angenähert wird. *Abschnitt 14.3* hat zwei Funktionen vorgestellt, die wie weiche Schwellenwerte aussehen: das Integral der standardmäßigen Normalverteilung (für das Probit-Modell verwendet) und die logistische Funktion (für das Logit-Modell). Beide Funktionen haben eine sehr ähnliche Gestalt, doch besitzt die logistische Funktion

$$Logistisch(z) = \frac{1}{1+e^{-z}}$$

komfortablere mathematische Eigenschaften. ▶ Abbildung 18.17(b) zeigt die Funktion. Wenn wir die Schwellenwertfunktion durch die logistische Funktion ersetzen, erhalten wir

$$h_{\mathbf{w}}(\mathbf{x}) = Logistisch(\mathbf{w}\cdot\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}\cdot\mathbf{x}}}$$

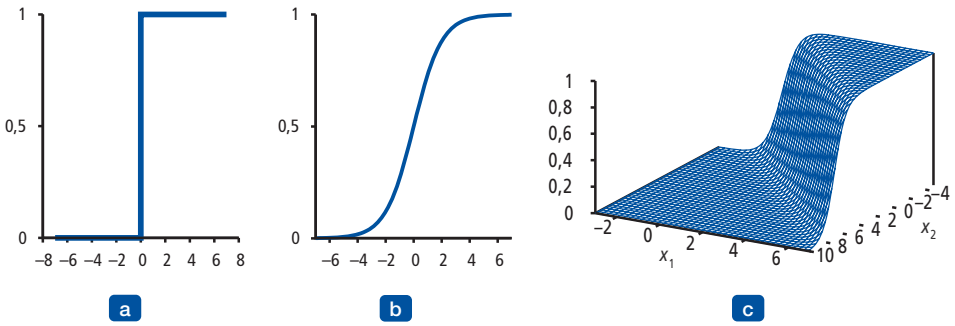


Abbildung 18.17: (a) Die Schwellenwertfunktion $Schwellenwert(z)$ mit 0/1-Ausgabe. Diese Funktion ist bei $z = 0$ nicht differenzierbar. (b) Die logistische Funktion $Logistisch(z) = 1/1+e^{-z}$, auch als Sigmoid-Funktion bezeichnet. (c) Grafik einer logistischen Regressionshypothese $h\mathrm{w}(\mathbf{x}) = Logistisch(\mathbf{w}\cdot\mathbf{x})$ für die in Abbildung 18.15(b) gezeigten Daten.

▶ Abbildung 18.17(c) zeigt ein Beispiel einer derartigen Hypothese für das Problem Erdbeben/Explosion mit zwei Eingaben. Die Ausgabe, die eine Zahl zwischen 0 und 1 sein kann, lässt sich als Wahrscheinlichkeit interpretieren, zur Klasse mit der Bezeichnung 1 zu gehören. Die Hypothese bildet eine weiche Grenze im Eingaberaum und liefert eine Wahrscheinlichkeit von 0,5 für eine beliebige Eingabe in der Mitte der Grenzregion und nähert sich 0 oder 1 an, wenn wir weiter von der Grenze weggehen.

Die Anpassung der Gewichte dieses Modelles, um den Verlust auf einer Datenmenge zu minimieren, wird als **logistische Regression** bezeichnet. Bei diesem Modell gibt es keine einfache Lösung in geschlossener Form, um den optimalen Wert $\mathbf{w}$ zu finden, doch ist die Berechnung des Gradientenabstieges recht unkompliziert. Da unsere Hypothesen nicht mehr nur 0 oder 1 ausgeben, verwenden wir die L_2-Verlustfunktion. Und um die Formeln überschaubar zu halten, schreiben wir g für die logistische Funktion und dementsprechend g' für ihre Ableitung.

Für ein einzelnes Beispiel ($\mathbf{x}$, y) ist die Ableitung des Gradienten gleich der für die lineare Regression (Gleichung (18.5)) bis zu dem Punkt, wo die eigentliche Form von h eingefügt wird. (Für diese Ableitung benötigen wir die **Kettenregel**: $\partial g(f(x))/\partial x = g'(f(x))\partial f(x)/\partial x$.) Wir haben

$$\begin{aligned}\frac{\partial}{\partial w_i} Verlust(\mathbf{w}) &= \frac{\partial}{\partial w_i}\big(y - h_{\mathbf{w}}(\mathbf{x})\big)^2 \\ &= 2\big(y - h_{\mathbf{w}}(\mathbf{x})\big) \times \frac{\partial}{\partial w_i}\big(y - h_{\mathbf{w}}(\mathbf{x})\big) \\ &= -2\big(y - h_{\mathbf{w}}(\mathbf{x})\big) \times g'(\mathbf{w}\cdot\mathbf{x}) \times \frac{\partial}{\partial w_i}\mathbf{w}\cdot\mathbf{x} \\ &= -2\big(y - h_{\mathbf{w}}(\mathbf{x})\big) \times g'(\mathbf{w}\cdot\mathbf{x}) \times x_i .\end{aligned}$$

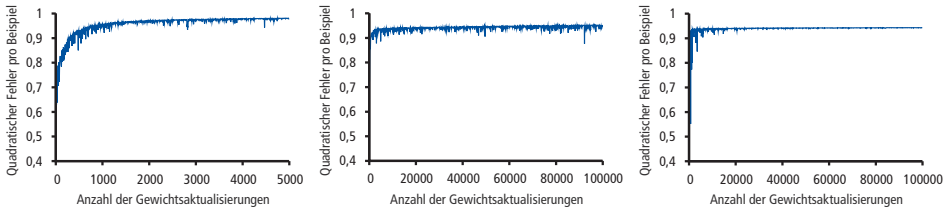


Abbildung 18.18: Wiederholung der Experimente in Abbildung 18.16 mit logistischer Regression und Fehlerquadraten. Die Grafik in (a) verkörpert 5000 Iterationen statt 1000, während (b) und (c) die gleiche Skala verwenden.

Die Ableitung g' der logistischen Funktion erfüllt $g'(z) = g(z)(1 - g(z))$. Damit haben wir

$$g'(\mathbf{w}\cdot\mathbf{x}) = g(\mathbf{w}\cdot\mathbf{x})(1 - g(\mathbf{w}\cdot\mathbf{x})) = h\mathrm{w}(\mathbf{x})(1 - h\mathrm{w}(\mathbf{x})),$$

sodass die Gewichtsaktualisierung für die Verlustminimierung lautet

$$w_i \leftarrow w_i + \alpha\,(y - h\mathrm{w}(\mathbf{x})) \times h\mathrm{w}(\mathbf{x})(1 - h\mathrm{w}(\mathbf{x})) \times x_i . \tag{18.8}$$

Wiederholt man die Experimente von Abbildung 18.16 mit logistischer Regression anstelle des linearen Schwellenwertklassifizierers, erhalten wir die in ▶ Abbildung 18.18 gezeigten Ergebnisse. Im linear separierbaren Fall (a) konvergiert die logistische Regression etwas langsamer, verhält sich aber vorhersagbarer. In (b) und (c), wo die Daten verrauscht und nicht separierbar sind, konvergiert die logistische Regression schneller und zuverlässiger. Diese Vorzüge zeigen sich auch in realen Anwendungen und die logistische Regression ist zu einer der populärsten Klassifizierungstechniken für Probleme in Medizin, Marketing und Umfragen, Kreditwürdigkeitsprüfung, Gesundheitswesen und anderen Anwendungen geworden.

18.7 Künstliche neuronale Netze

Wir kommen nun zu einem scheinbar abseits liegenden Thema: dem Gehirn. Wie Sie allerdings noch sehen werden, erweisen sich die bisher in diesem Kapitel diskutierten technischen Konzepte in der Tat als nützlich, um mathematische Modelle für die Aktivität des Gehirns zu erstellen; umgekehrt helfen Untersuchungen zu Denkabläufen dabei, den Bereich der technischen Konzepte zu erweitern.

Kapitel 1 hat kurz die grundlegenden Ergebnisse der Neurowissenschaft angerissen – insbesondere die Hypothese, dass mentale Aktivität hauptsächlich aus elektrochemischer Aktivität in Netzen von Gehirnzellen, den sogenannten **Neuronen**, besteht. (Abbildung 1.2 zeigt eine schematische Darstellung eines typischen Neurons.) Angeregt durch diese Hypothese zielten einige der frühesten Arbeiten zur KI darauf ab, künstliche **neuronale Netze** zu erzeugen. (Andere Namen für diesen Bereich sind **Konnektionismus**, **parallel verteilte Verarbeitung** und **neuronale Programmierung**.) ▶ Abbildung 18.19 zeigt ein einfaches mathematisches Modell des Neurons, das von McCulloch und Pitts (1943) abgeleitet wurde. Einfach ausgedrückt, „feuert" es, wenn eine lineare Kombination seiner Eingaben einen bestimmten (harten oder weichen) Schwellenwert überschreitet – d.h., es implementiert einen linearen Klassifizierer, wie ihn der vorherige Abschnitt beschrieben hat. Ein neuronales Netz ist lediglich eine Sammlung von Einheiten, die miteinander verbunden sind; die Eigenschaften des Netzes werden durch seine Topologie und die Eigenschaften der „Neuronen" bestimmt.

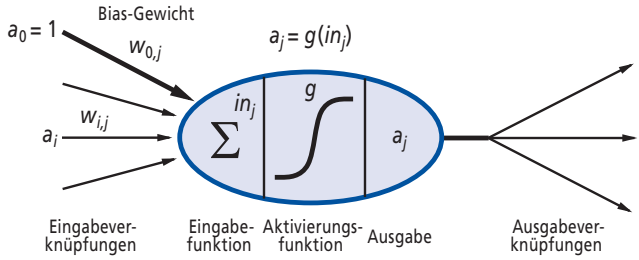


Abbildung 18.19: Ein einfaches mathematisches Modell für ein Neuron. Die Ausgabeaktivierung der Einheit ergibt sich zu $a_j = g(\Sigma_{i=0}{}^{n}\, w_{i,j} a_i)$, wobei a_i die Ausgabeaktivierung der Einheit i und $w_{i,j}$ das Gewicht der Verknüpfung von Einheit i zu dieser Einheit ist.

Seit 1943 wurden sowohl für die Neuronen als auch für größere Systeme im Gehirn sehr viel detailliertere und realistischere Modelle entwickelt, was zum modernen Gebiet der **Neurowissenschaft** (**Computational Neuroscience**) geführt hat. Andererseits haben sich Forscher in KI und Statistik für die abstrakteren Eigenschaften neuronaler Netze interessiert, wie etwa ihre Fähigkeit, verteilte Berechnungen vorzunehmen, verrauschte Eingaben zu tolerieren und zu lernen. Wir wissen heute zwar, dass andere Arten von Systemen – einschließlich Bayesscher Netze – diese Eigenschaften haben, aber neuronale Netze bleiben eine der bekanntesten und effektivsten Formen von Lernsystemen und sollten deshalb auch eigenständig betrachtet werden.

18.7.1 Strukturen neuronaler Netze

Neuronale Netze setzen sich aus Knoten oder **Einheiten** zusammen (siehe Abbildung 18.19), die durch gerichtete **Verknüpfungen** verbunden sind. Eine Verknüpfung von der Einheit i zur Einheit j dient dazu, die **Aktivierung** a_i von i nach j zu propagieren.[8]

8 Ein Hinweis zur Notation: Für diesen Abschnitt sind wir gezwungen, unsere üblichen Konventionen aufzuheben. Eingabeattribute werden zwar weiterhin mit i indiziert, sodass eine „externe" Aktivierung a_i durch Eingabe x_i gegeben ist, jedoch verweist der Index j jetzt auf interne Einheiten und nicht auf Beispiele. Im gesamten Abschnitt beziehen sich die mathematischen Herleitungen auf ein einzelnes generisches Beispiel $\mathbf{x}$. Damit entfallen die üblichen Summierungen über Beispielen, um Ergebnisse für die gesamte Datenmenge zu erhalten.

Jeder Verknüpfung ist ein numerisches **Gewicht** $w_{i,j}$ zugeordnet, das die Stärke und das Vorzeichen der Verknüpfung bestimmt. Wie in linearen Regressionsmodellen verfügt jede Einheit über eine Dummy-Eingabe $a_0 = 1$ mit einem zugeordneten Gewicht $w_{0,j}$. Jede Einheit j berechnet zunächst die gewichtete Summe ihrer Eingaben:

$$in_j = \sum_{i=0}^{n} w_{i,j} a_i .$$

Anschließend wendet sie eine **Aktivierungsfunktion** g auf diese Summe an, um die Ausgabe abzuleiten:

$$a_j = g\left(in_j\right) = g\left(\sum_{i=0}^{n} w_{i,j} a_i\right). \tag{18.9}$$

Die Aktivierungsfunktion g ist normalerweise entweder eine Schwellenwertfunktion (Abbildung 18.17(a)), wobei man die Einheit als **Perzeptron** bezeichnet, oder eine logistische Funktion (Abbildung 18.17(b)), wobei man von einem **Sigmoid-Perzeptron** spricht. Beide dieser nichtlinearen Aktivierungsfunktionen gewährleisten die wichtige Eigenschaft, dass das gesamte Netz der Einheiten eine nichtlineare Funktion darstellen kann (siehe Übung 18.26). Wie bereits bei der Diskussion der logistischen Regression (*Abschnitt 18.6.4*) erwähnt, besitzt die logistische Aktivierungsfunktion den zusätzlichen Vorteil, differenzierbar zu sein.

Nachdem das mathematische Modell für die einzelnen „Neuronen" festliegt, sind im nächsten Schritt die Neuronen miteinander zu verbinden, um ein Netz zu bilden. Das kann prinzipiell auf zwei verschiedenen Wegen geschehen. Ein **Feedforward-Netz** besitzt Verknüpfungen nur in einer Richtung – d.h., es bildet einen gerichteten, azyklischen Graphen. Jeder Knoten empfängt Eingaben von vorgelagerten Knoten und liefert die Ausgabe an nachgelagerte Knoten; es gibt keine Schleifen. Ein Feedforward-Netz repräsentiert eine Funktion seiner aktuellen Eingabe; somit besitzt es außer den Gewichten selbst keinen internen Zustand. Dagegen speist ein **rekurrentes** bzw. **rückgekoppeltes neuronales Netz** seine Ausgaben wieder in seine eigenen Eingaben ein. Die Aktivierungsebenen des Netzes bilden damit ein dynamisches System, das einen stabilen Zustand erreichen kann, vielleicht aber auch zu Schwingungen neigt oder sogar chaotisches Verhalten zeigt. Darüber hinaus hängt die Antwort des Netzes auf eine bestimmte Eingabe von seinem Anfangszustand ab, der wiederum von vorherigen Eingaben abhängig sein kann. Folglich können rückgekoppelte Netze (im Unterschied zu Feedforward-Netzen) ein Kurzzeitgedächtnis unterstützen. Das macht sie zu interessanteren Modellen des Gehirns, die jedoch auch schwieriger zu verstehen sind. Dieser Abschnitt konzentriert sich auf Feedforward-Netze; am Ende des Kapitels finden Sie einige Hinweise auf Literatur zu Netzen mit Rückkopplung.

Feedforward-Netze sind normalerweise in **Schichten** angeordnet, sodass jede Einheit ihre Eingaben nur von den unmittelbar vorhergehenden Schichten erhält. In den beiden nächsten Unterabschnitten betrachten wir einschichtige Netze, in denen jede Einheit direkt von den Eingaben des Netzes zu seinen Ausgaben verbunden ist, sowie mehrschichtige Netze mit einer oder mehreren Schichten **verborgener Einheiten** (**Hidden Units**), die nicht mit den Ausgaben des Netzes verbunden sind. Bisher haben wir in diesem Kapitel nur Lernprobleme mit einer einzelnen Ausgangsvariablen y betrachtet, doch werden neuronale Netze oftmals in Fällen verwendet, in denen mehrere

Ausgaben angebracht sind. Möchten wir zum Beispiel ein Netz trainieren, das zwei Eingabebits, die jeweils 0 oder 1 sind, addiert, brauchen wir einen Ausgang für das Summenbit und einen für das Übertragsbit. Auch wenn das Lernproblem eine Klassifizierung in mehr als zwei Klassen umfasst – wenn die Lernaufgabe zum Beispiel darin besteht, Bilder handschriftlicher Ziffern zu kategorisieren –, ist es üblich, für jede Klasse eine eigene Ausgabeeinheit vorzusehen.

18.7.2 Einschichtige neuronale Feedforward-Netze (Perzeptrons)

Ein Netz, bei dem alle Eingaben direkt mit den Ausgaben verknüpft sind, wird als **einschichtiges neuronales Netz** oder **Perzeptron-Netz** bezeichnet. ▶ Abbildung 18.20 zeigt ein einfaches Perzeptron-Netz mit zwei Eingaben und zwei Ausgaben. Mit einem derartigen Netz könnten wir zum Beispiel versuchen, die 2-Bit-Adder-Funktion zu lernen. Die folgende Tabelle gibt alle benötigten Trainingsdaten an:

x_1	x_2	y_3 (Übertrag)	y_4 (Summe)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

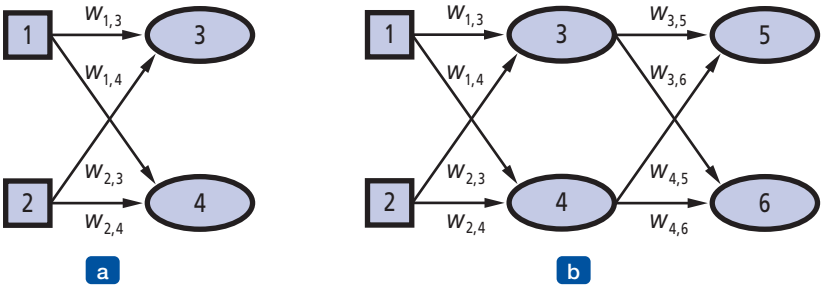


Abbildung 18.20: (a) Ein Perzeptron-Netz mit zwei Eingabe- und zwei Ausgabeeinheiten. (b) Ein neuronales Netz mit zwei Eingaben, einer verborgenen Schicht mit zwei Einheiten und zwei Ausgabeeinheiten. Nicht dargestellt sind die Dummy-Eingaben und deren zugeordnete Gewichte.

Erstens ist festzustellen, dass ein Perzeptron-Netz mit m Ausgaben eigentlich m getrennte Netze darstellt, da jedes Gewicht nur eine der Ausgaben beeinflusst. Somit gibt es m getrennte Trainingsprozesse. Darüber hinaus arbeiten die Trainingsprozesse je nach Typ der verwendeten Aktivierungsfunktion entweder nach der Perzeptron-Lernregel (Gleichung (18.7) in *Abschnitt 18.6.3*) oder der Gradientenabstiegsregel für die logistische Regression (Gleichung (18.8) in *Abschnitt 18.6.4*).

Wenn Sie eine dieser Methoden für die Daten des 2-Bit-Adders ausprobieren, passiert etwas Interessantes. Einheit 3 lernt die Übertragsfunktion ganz leicht, doch Einheit 4 kann die Summenfunktion beim besten Willen nicht lernen. Dabei ist Einheit 4 kei-

nesfalls fehlerhaft! Das Problem liegt in der Summenfunktion an sich. Wie *Abschnitt 18.6* gezeigt hat, können lineare Klassifizierer (egal ob hart oder weich) lineare Entscheidungsgrenzen im Eingaberaum darstellen. Das funktioniert einwandfrei für die Übertragsfunktion, die ein logisches AND verkörpert (siehe ▶ Abbildung 18.21(a)). Dagegen ist die Summenfunktion eine XOR-Verknüpfung (ausschließendes Oder) von zwei Eingaben. Wie ▶ Abbildung 18.21(c) veranschaulicht, ist diese Funktion nicht linear separierbar und das Perzeptron kann sie nicht lernen.

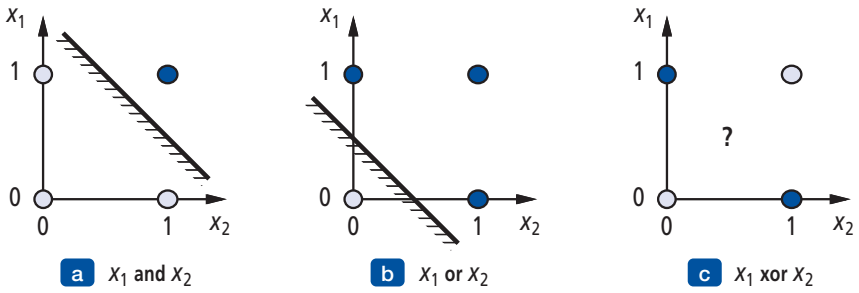


Abbildung 18.21: Lineare Separierbarkeit in Schwellenwert-Perzeptrons. Schwarze Punkte kennzeichnen Stellen im Eingaberaum, wo der Wert der Funktion gleich 1 ist, und weiße Punkte geben an, wo der Wert 0 ist. Das Perzeptron gibt im Bereich der nicht schraffierten Seite der Geraden 1 zurück. In (c) existiert keine derartige Linie, die die Eingaben korrekt klassifiziert.

Die linear separierbaren Funktionen machen lediglich einen kleinen Anteil aller booleschen Funktionen aus; Übung 18.23 stellt die Aufgabe, diesen Anteil zu quantifizieren. Die Unfähigkeit des Perzeptrons, selbst solche einfachen Funktionen wie XOR zu lernen, war ein beträchtlicher Rückschlag für die 1960 gerade entstehende Forschergruppe, die sich mit neuronalen Netzen beschäftigte. Dennoch sind Perzeptrons bei weitem nicht nutzlos. *Abschnitt 18.6.4* hat bereits darauf hingewiesen, dass logistische Regression (d.h. Training eines Sigmoid-Perzeptrons) selbst heute noch ein sehr populäres und effizientes Tool ist. Darüber hinaus ist ein Perzeptron in der Lage, einige recht „komplexe" boolesche Funktionen relativ kompakt darzustellen. Zum Beispiel lässt sich die **Majoritätsfunktion**, die nur dann 1 zurückgibt, wenn mehr als die Hälfte ihrer n Eingaben gleich 1 sind, durch ein Perzeptron darstellen, wobei jedes $w_i = 1$ und $w_0 = -n/2$ ist. In einem Entscheidungsbaum wären exponentiell viele Knoten erforderlich, um diese Funktion darzustellen.

▶ Abbildung 18.22 zeigt die Lernkurve für ein Perzeptron bei zwei verschiedenen Problemen. Auf der linken Seite ist die Kurve für das Lernen der Majoritätsfunktion mit elf booleschen Eingaben zu sehen (d.h. die Funktion gibt eine 1 aus, wenn 6 oder mehr Eingaben 1 sind). Wie zu erwarten, lernt das Perzeptron die Funktion recht schnell, da die Majoritätsfunktion linear separierbar ist. Dagegen macht die Lernroutine für den Entscheidungsbaum keinen Fortschritt, da sich die Majoritätsfunktion nur sehr schwer als Entscheidungsbaum darstellen lässt (auch wenn dies nicht unmöglich ist). Auf der rechten Seite ist das Restaurantbeispiel zu sehen. Das Lösungsproblem ist leicht als Entscheidungsbaum darstellbar, aber nicht linear separierbar. Die beste Ebene durch die Daten klassifiziert nur zu 65% korrekt.

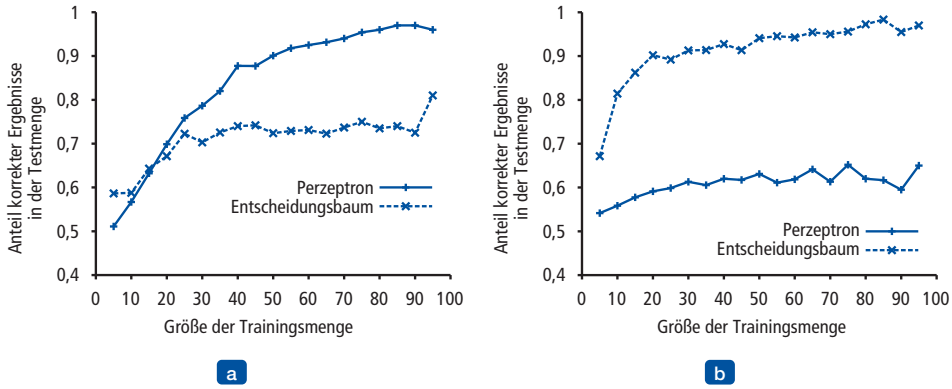


Abbildung 18.22: Vergleich der Leistungen von Perzeptrons und Entscheidungsbäumen. (a) Perzeptrons schneiden beim Lernen der Majoritätsfunktion von elf Eingaben besser ab. (b) Entscheidungsbäume lernen besser das Prädikat *WerdenWarten* im Restaurantbeispiel.

18.7.3 Mehrschichtige neuronale Feedforward-Netze

McCulloch und Pitts (1943) waren sich wohl bewusst, dass eine einzelne Schwellenwerteinheit nicht alle ihre Probleme lösen wird. In der Tat beweist ihr Artikel, dass eine derartige Einheit die grundlegenden booleschen Funktionen AND, OR und NOT darstellen kann, und führt dann weiter aus, dass sich jede gewünschte Funktionalität erhalten lässt, wenn man sehr viele Einheiten in (möglicherweise rekurrenten) Netzen beliebiger Tiefe verbindet. Doch niemand wusste, wie man derartige Netze trainiert.

Dies erweist sich allerdings als einfaches Problem, wenn wir ein Netz in der richtigen Weise auffassen: als eine Funktion $h\mathrm{w}(\mathbf{x})$, die durch die Gewichte $\mathbf{w}$ parametrisiert ist. Betrachten Sie das einfache Netz in Abbildung 18.20(b), das aus zwei Eingabeeinheiten, zwei verborgenen Einheiten und zwei Ausgabeeinheiten besteht. (Außerdem weist jede Einheit eine Dummy-Eingabe mit dem festen Wert 1 auf.) Für einen gegebenen Eingabevektor $\mathbf{x} = (x_1, x_2)$ werden die Aktivierungen der Eingabeeinheiten auf $(a_1, a_2) = (x_1, x_2)$ gesetzt. Die Ausgabe bei Einheit 5 ist gegeben durch

$$\begin{aligned} a_5 &= g(w_{0,5} + w_{3,5}\, a_3 + w_{4,5}\, a_4) \\ &= g(w_{0,5} + w_{3,5}\, g(w_{0,3} + w_{1,3}\, a_1 + w_{2,3}\, a_2) + w_{4,5}\, g(w_{0,4} + w_{1,4}\, a_1 + w_{2,4}\, a_2)) \\ &= g(w_{0,5} + w_{3,5}\, g(w_{0,3} + w_{1,3}\, x_1 + w_{2,3}\, x_2) + w_{4,5}\, g(w_{0,4} + w_{1,4}\, x_1 + w_{2,4}\, x_2)). \end{aligned}$$

Somit haben wir die Ausgabe als Funktion der Eingaben und der Gewichte ausgedrückt. Ein ähnlicher Ausdruck gilt für Einheit 6. Solange sich die Ableitungen derartiger Ausdrücke in Bezug auf die Gewichte berechnen lassen, können wir den Gradientenabstieg als Methode zur Verlustminimierung verwenden, um das Netz zu trainieren. *Abschnitt 18.7.4* zeigt genau, wie dabei vorzugehen ist. Und da die durch ein Netz dargestellte Funktion stark nichtlinear sein kann – zudem zusammengesetzt aus verschachtelten nichtlinearen weichen Schwellenwertfunktionen –, können wir neuronale Netze als Werkzeug für **nichtlineare Regression** auffassen.

Bevor wir uns eingehend mit den Lernregeln befassen, sehen wir uns zunächst an, wie Netze komplizierte Funktionen erzeugen. Wie Sie bereits wissen, stellt jede Einheit in einem Sigmoid-Netz eine weiche Schwelle in ihrem Eingaberaum dar, wie es Abbildung 18.17(c) zeigt. Mit einer verborgenen Schicht und einer Ausgabeschicht wie in Abbildung

18.20(b) berechnet jede Ausgabeeinheit eine Linearkombination mehrerer derartiger Funktionen mit weicher Schwelle. Indem man zum Beispiel zwei entgegengerichtete Funktionen mit weicher Schwelle addiert und das Ergebnis über eine Schwellenwertfunktion leitet, können wir eine „Bergrücken"-Funktion erhalten, wie sie ▶ Abbildung 18.23(a) zeigt. Kombiniert man zwei derartige Bergrücken rechtwinklig zueinander (d.h., man kombiniert die Ausgaben von vier verborgenen Einheiten), erhalten wir einen „Höcker", wie in ▶ Abbildung 18.23(b) gezeigt.

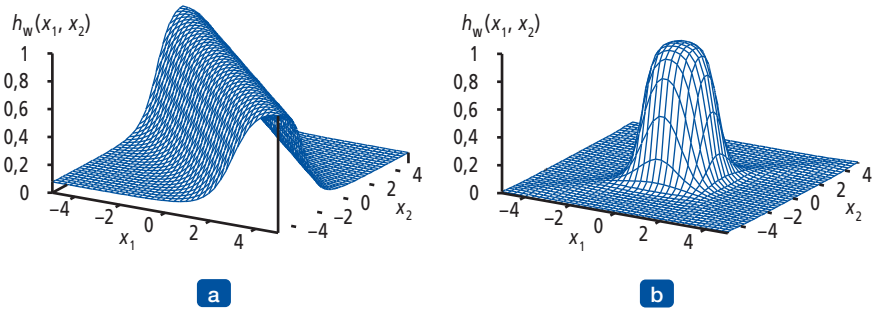


Abbildung 18.23: (a) Kombination von zwei entgegengesetzten weichen Schwellenwertfunktionen, die einen Bergrücken erzeugt. (b) Die Kombination von zwei Bergrücken erzeugt einen Höcker.

Mit mehr verborgenen Einheiten können wir mehrere Höcker unterschiedlicher Größen an unterschiedlichen Stellen erzeugen. Mit einer einzigen, ausreichend großen verborgenen Schicht ist es möglich, jede beliebige stetige Funktion der Eingaben mit beliebiger Genauigkeit darzustellen; mit zwei Schichten können sogar unstetige Funktionen dargestellt werden.[9] Leider ist es für *bestimmte* Netzstrukturen schwieriger, genau festzustellen, welche Funktionen damit dargestellt werden können und welche nicht.

18.7.4 Lernen in mehrschichtigen Netzen

Fürs Erste lassen wir eine kleinere Komplikation, die in mehrschichtigen Netzen auftritt, unter den Tisch fallen: Interaktionen zwischen den Lernproblemen, wenn das Netz mehrere Ausgaben hat. In derartigen Fällen stellen wir uns vor, dass das Netz eine Vektorfunktion $\mathbf{h}_w$ statt einer Skalarfunktion h_w implementiert; zum Beispiel gibt das Netz in Abbildung 18.20(b) einen Vektor $[a_5, a_6]$ zurück. Analog ist die Zielausgabe ein Vektor $\mathbf{y}$. Während ein Perzeptron-Netz für Probleme mit m Ausgaben in m separate Lernprobleme zerfällt, funktioniert diese Zerlegung in einem mehrschichtigen Netz nicht. Zum Beispiel sind sowohl a_5 als auch a_6 in Abbildung 18.20(b) von allen Gewichten der Eingabeschicht abhängig, sodass Aktualisierungen dieser Gewichte von den Fehlern sowohl in a_5 als auch a_6 abhängen. Zum Glück ist diese Abhängigkeit sehr einfach, wenn eine Verlustfunktion über die Komponenten des Fehlervektors $\mathbf{y} - \mathbf{h}_w(\mathbf{x})$ additiv ist. So haben wir für den L_2-Verlust für ein Gewicht w

$$\frac{\partial}{\partial w} Verlust(\mathbf{w}) = \frac{\partial}{\partial w} \left|\mathbf{y} - \mathbf{h}_{\mathbf{w}}(\mathbf{x})\right|^2 = \frac{\partial}{\partial w} \sum_k \left(y_k - a_k\right)^2 = \sum_k \frac{\partial}{\partial w} \left(y_k - a_k\right)^2. \qquad (18.10)$$

9 Der Beweis ist kompliziert, aber der wichtigste Aspekt ist, dass die erforderliche Anzahl verborgener Einheiten exponentiell mit der Anzahl der Eingaben wächst. Beispielsweise braucht man $2^n/n$ verborgene Einheiten, um alle booleschen Funktionen mit n Eingaben zu kodieren.

Hier reicht der Index k über die Knoten in der Ausgabeschicht. Jeder Term in der endgültigen Summation ist einfach der Gradient des Verlustes für den k-ten Ausgang, der so berechnet wird, als ob die anderen Ausgänge nicht existieren. Folglich können wir ein Lernproblem mit m Ausgängen in m Lernprobleme zerlegen, sofern wir daran denken, die jeweiligen Gradientenbeiträge zu addieren, wenn wir die Gewichte aktualisieren.

Die Hauptschwierigkeit ergibt sich daraus, dass dem Netz verborgene Schichten hinzugefügt wurden. Während der Fehler $\mathbf{y} - \mathbf{h}_\mathbf{w}$ an der Ausgabeschicht klar ist, scheint der Fehler in den verborgenen Schichten Rätsel aufzugeben, weil aus den Trainingsdaten nicht hervorgeht, welche Werte die verborgenen Knoten haben sollten. Erfreulicherweise zeigt sich, dass wir den Fehler von der Ausgabeschicht zu den verborgenen Schichten **zurückführen** können. Diese Fehlerrückführung (engl. **Backpropagation**) ergibt sich direkt aus einer Ableitung des Gesamtfehlergradienten. Zuerst beschreiben wir den Prozess mit einer intuitiven Begründung und geben dann die Ableitung an.

In der Ausgabeschicht ist die Gewichtsaktualisierungsregel identisch mit Gleichung (18.8). Da wir mehrere Ausgabeeinheiten haben, sei Err_k die k-te Komponente des Fehlervektors $\mathbf{y} - \mathbf{h}_\mathbf{w}$. Außerdem ist es zweckmäßig, einen modifizierten Fehler $\Delta_k = Err_k \times g'(in_k)$ zu definieren, sodass die Gewichtsaktualisierungsregel wie folgt aussieht:

$$w_{j,k} \leftarrow w_{j,k} + \alpha \times a_j \times \Delta_k. \tag{18.11}$$

Um die Verknüpfungen zwischen den Eingabeeinheiten und den verborgenen Einheiten zu aktualisieren, müssen wir eine Größe analog zum Fehlerterm für Ausgabeknoten definieren. Hier realisieren wir die Fehlerrückführung. Das Prinzip besteht darin, dass der verborgene Knoten j für einen bestimmten Anteil des Fehlers Δ_k in jedem der mit ihm verbundenen Ausgabeknoten „verantwortlich“ ist. Somit werden die Δ_k-Werte entsprechend der Verbindungsstärke zwischen dem verborgenen Knoten und dem Ausgabeknoten aufgeteilt und zurückgeführt, um die Δ_j-Werte für die verborgene Schicht bereitzustellen. Die Rückführungsregel für die Δ-Werte lautet:

$$\Delta_j = g'(in_j) \sum_k w_{j,k} \Delta_k. \tag{18.12}$$

Jetzt ist die Gewichtungsaktualisierungsregel für die Gewichte zwischen den Eingaben und der verborgenen Schicht fast identisch mit der Aktualisierungsregel für die Ausgabeschicht:

$$w_{j,j} \leftarrow w_{j,j} + \alpha \times a_i \times \Delta_j.$$

Der Backpropagation-Prozess lässt sich wie folgt zusammenfassen:

- Berechne die Δ-Werte für die Ausgabeeinheiten unter Verwendung des beobachteten Fehlers.
- Wiederhole beginnend mit der Ausgabeschicht die folgenden Schritte für jede Schicht im Netz, bis die erste verborgene Schicht erreicht ist:
  - Gib die Δ-Werte zurück an die vorhergehende Schicht.
  - Aktualisiere die Gewichte zwischen den beiden Schichten.

▶ Abbildung 18.24 zeigt den detaillierten Algorithmus.

```
function BACK-PROP-LEARNING(examples, network) returns ein neuronales Netz
  inputs: examples, eine Menge von Beispielen jeweils mit Eingabevektor x
          und Ausgabevektor y
          network, ein mehrschichtiges Netz mit L Schichten, Gewichten w_{i,j},
          Aktivierungsfunktion g
  local variables: Δ, ein Fehlervektor, indiziert durch Netzknoten

  repeat
    for each Gewicht w_{i,j} in network do
      w_{i,j} ← eine kleine Zufallszahl
    for each Beispiel (x, y) in examples do
      /* Die Eingaben vorwärts weiterleiten, um die Ausgaben zu berechnen */
      for each Knoten i in der Eingabeschicht do
        a_i ← x_i
      for ℓ = 2 to L do
        for each Knoten j in Schicht ℓ do
          in_j ← Σ_i w_{i,j} a_i
          a_j ← g(in_j)
      /* Delta-Werte von Ausgabeschicht zur Eingabeschicht zurückführen */
      for each Knoten j in der Ausgabeschicht do
        Δ[j] ← g'(in_j) × (y_j - a_j)
      for ℓ = L - 1 to 1 do
        for each Knoten i in Schicht ℓ do
          Δ[i] ← g'(in_i)Σ_j w_{i,j} Δ[j]
      /* Jedes Gewicht im Netz mit den Delta-Werten aktualisieren */
      for each Gewicht w_{i,j} in network do
        w_{i,j} ← w_{i,j} + α × a_i × Δ[j]
  until ein Stoppkriteriterium erfüllt ist
  return network
```

Abbildung 18.24: Der Backpropagation-Algorithmus für das Lernen in mehrschichtigen Netzen.

Für die Mathematikinteressierten leiten wir jetzt die Backpropagation-Gleichungen von Grund auf her. Die Ableitung ähnelt der Berechnung des Gradienten für logistische Regression (was bis zur Gleichung (18.8) in *Abschnitt 18.6.4* führt), außer dass wir die Kettenregel mehrmals anwenden müssen.

Entsprechend Gleichung (18.10) berechnen wir lediglich den Gradienten für $Verlust_k = (y_k - a_k)^2$ an der k-ten Ausgabe. Der Gradient dieses Verlustes in Bezug auf die Gewichte, die die verborgene Schicht mit der Ausgabeschicht verknüpfen, ist null, außer für Gewichte $w_{j,k}$, die mit der k-ten Ausgabeeinheit verknüpft sind. Für diese Gewichte haben wir

$$\frac{\partial Verlust_k}{\partial w_{j,k}} = -2(y_k - a_k)\frac{\partial a_k}{\partial w_{j,k}} = -2(y_k - a_k)\frac{\partial g(in_k)}{\partial w_{j,k}}$$

$$= -2(y_k - a_k)g'(in_k)\frac{\partial in_k}{\partial w_{j,k}} = -2(y_k - a_k)g'(in_k)\frac{\partial}{\partial w_{j,k}}\left(\sum_j w_{j,k}a_j\right)$$

$$= -2(y_k - a_k)g'(in_k)a_j = -a_j\Delta_k.$$

wobei Δ_k wie zuvor definiert ist. Um den Gradienten in Bezug auf die Gewichte $w_{i,j}$ zu erhalten, die die Eingabeschicht mit der verborgenen Schicht verknüpfen, müssen wir

die Aktivierungen a_j erweitern und erneut die Kettenregel anwenden. Wir zeigen die Ableitung ganz ausführlich, weil es interessant ist zu sehen, wie der Ableitungsoperator durch das Netz zurückgereicht wird:

$$\begin{aligned}
\frac{\partial Verlust_k}{\partial w_{i,j}} &= -2\left(y_k - a_k\right)\frac{\partial a_k}{\partial w_{i,j}} = -2\left(y_k - a_k\right)\frac{\partial g\left(in_k\right)}{\partial w_{i,j}} \\
&= -2\left(y_k - a_k\right)g'\left(in_k\right)\frac{\partial in_k}{\partial w_{i,j}} = -2\Delta_k \frac{\partial}{\partial w_{i,j}}\left(\sum_j w_{j,k}a_j\right) \\
&= -2\Delta_k w_{j,k}\frac{\partial a_j}{\partial w_{i,j}} = -2\Delta_k w_{j,k}\frac{\partial g\left(in_j\right)}{\partial w_{i,j}} \\
&= -2\Delta_k w_{j,k}g'\left(in_j\right)\frac{\partial in_j}{\partial w_{i,j}} \\
&= -2\Delta_k w_{j,k}g'\left(in_j\right)\frac{\partial}{\partial w_{i,j}}\left(\sum_i w_{i,j}a_i\right) \\
&= -2\Delta_k w_{j,k}g'\left(in_j\right)a_i = -a_i\Delta_j ,
\end{aligned}$$

wobei Δj wie zuvor definiert ist. Wir erhalten also die gleichen Aktualisierungsregeln, wie wir sie zuvor aus intuitiven Betrachtungen abgeleitet haben. Außerdem wird deutlich, dass der Prozess für Netze mit mehr als einer verborgenen Schicht fortgesetzt werden kann, was den in Abbildung 18.24 gezeigten allgemeinen Algorithmus rechtfertigt.

Nachdem Sie die mathematische Herleitung nun eingehend studiert (oder überblättert) haben, wollen wir betrachten, welche Leistung ein Netz mit einer verborgenen Schicht für das Restaurantproblem erbringt. Zuerst müssen wir die Struktur des Netzes festlegen. Jedes Beispiel wird durch zehn Attribute beschrieben, sodass wir zehn Eingabeeinheiten brauchen. Verwenden wir nun eine verborgene Schicht oder zwei? Wie viele Knoten soll jede Schicht enthalten? Sollen die Knoten vollständig verknüpft werden? Leider gibt es keine brauchbare Theorie, die uns die Antworten liefert (siehe den nächsten Abschnitt). Wie immer können wir Kreuzvalidierung verwenden: mehrere unterschiedliche Strukturen ausprobieren und feststellen, welche am besten geeignet ist. ▶ Abbildung 18.25 zeigt zwei Kurven. Bei der ersten handelt es sich um eine Trainingskurve, die den mittleren Quadratfehler für eine gegebene Trainingsmenge von 100 Restaurantbeispielen während des Gewichtsaktualisierungsprozesses zeigt. Daraus ist erkennbar, dass das Netz tatsächlich zu einer perfekten Übereinstimmung mit den Trainingsdaten konvergiert. Die zweite Kurve ist die Standardlernkurve für die Restaurantdaten. Das neuronale Netz lernt gut, wenn auch nicht ganz so schnell wie das Lernen mit Entscheidungsbaum; das ist vielleicht nicht überraschend, weil die Daten anfangs von einem einfachen Entscheidungsbaum erzeugt wurden.

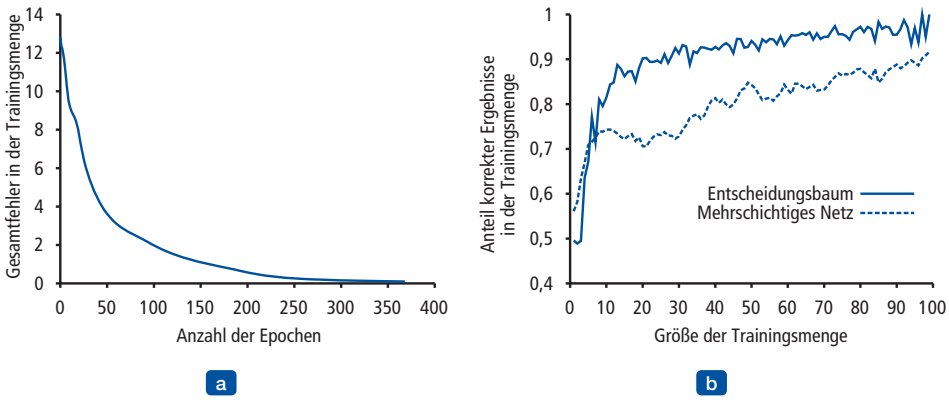


Abbildung 18.25: (a) Trainingskurve, die die schrittweise Reduzierung des Fehlers bei über mehrere Epochen angepassten Gewichten für eine Bespielmenge aus der Restaurantdomäne zeigt. (b) Vergleichbare Lernkurven zeigen, dass das Lernen mit Entscheidungsbäumen eine etwas bessere Leistung als Backpropagation in einem mehrschichtigen Netz erbringt.

Neuronale Netze können natürlich sehr viel komplexere Lernaufgaben bewältigen, auch wenn gesagt werden muss, dass ein wenig Aufwand zu betreiben ist, um die Netzstruktur so anzupassen, dass eine Konvergenz in die Nähe des globalen Optimums im Gewichtsraum entsteht. Es gibt im wahrsten Sinne des Wortes Zehntausende veröffentlichte Anwendungen neuronaler Netze. *Abschnitt 18.11.1* geht auf eine derartige Anwendung ausführlicher ein.

18.7.5 Strukturen neuronaler Netze lernen

Bisher haben wir das Problem betrachtet, Gewichte für eine feste Netzstruktur zu lernen; wie bei Bayesschen Netzen müssen wir aber auch verstehen, wie man die beste Netzstruktur findet. Wenn wir ein zu großes Netz wählen, kann es sich alle Beispiele merken, indem es eine große Nachschlagetabelle anlegt, aber es kann nicht unbedingt gut auf Eingaben verallgemeinert werden, die zuvor nicht gesehen wurden.[10] Mit anderen Worten, wie alle statistischen Modelle leiden auch neuronale Netze unter einer **Überanpassung**, wenn es zu viele Parameter im Modell gibt. Wir haben das bereits in Abbildung 18.1 gesehen, wo die parameterintensiven Modelle in (b) und (c) mit allen Daten übereinstimmen, aber nicht so gut verallgemeinern wie die wenig parameterintensiven Modelle in (a) und (d).

Wenn wir bei vollständig verbundenen Netzen bleiben, müssen wir uns lediglich über die Anzahl der verborgenen Schichten und ihre Größen klar werden. Der übliche Ansatz ist, mehrere auszuprobieren und die beste Variante beizubehalten. Die in diesem Kapitel beschriebenen Techniken der **Kreuzvalidierung** brauchen wir, wenn wir **Peeking** in der Testmenge vermeiden wollen. Das bedeutet, wir wählen die Netzarchitektur aus, die die höchste Vorhersagegenauigkeit für die Auswertungsmengen erbringt.

10 Es wurde beobachtet, dass sehr große Netze eine gute Verallgemeinerung erzielen, *solange die Gewichte klein bleiben*. Diese Beschränkung bewirkt, dass die Aktivierungswerte im *linearen* Bereich der Sigmoid-Funktion $g(x)$ bleiben, wo x gegen null geht. Das wiederum bedeutet, dass sich das Netz wie eine lineare Funktion mit weit weniger Parametern verhält (Übung 18.26).

Wenn wir Netze betrachten wollen, die nicht vollständig verknüpft sind, brauchen wir eine effektive Suchmethode für den sehr großen Raum möglicher Verknüpfungstopologien. Der Algorithmus der **optimalen Hirnschädigung** beginnt mit einem vollständig verknüpften Netz und entfernt Verknüpfungen daraus. Nachdem das Netz zum ersten Mal trainiert wurde, identifiziert ein informationstheoretischer Ansatz eine optimale Auswahl an Verknüpfungen, die entfernt werden können. Das Netz wird erneut trainiert und wenn sich seine Leistung nicht verschlechtert hat, wird der Prozess wiederholt. Neben einzelnen Verknüpfungen lassen sich auch ganze Einheiten entfernen, die kaum etwas zum Ergebnis beitragen.

Es wurden mehrere Algorithmen vorgeschlagen, wie aus kleineren Netzen größere Netze entstehen können. Zum Beispiel erinnert der **Tiling**-Algorithmus an Lernen mit Entscheidungslisten. Die Idee dabei ist, mit einer einzigen Einheit zu beginnen, die ihr Bestes tut, um für so viele der Trainingsbeispiele wie möglich die korrekte Ausgabe zu erzeugen. Anschließend werden Einheiten eingefügt, die die Beispiele abdecken sollen, für die die ursprüngliche Einheit keine korrekte Lösung gefunden hat. Der Algorithmus fügt nur so viele Einheiten ein, wie er braucht, um alle Beispiele abzudecken.

18.8 Parameterfreie Modelle

Lineare Regression und neuronale Netze verwenden Trainingsdaten, um einen festen Satz von Parametern $\mathbf{w}$ abzuschätzen. Dieser definiert dann unsere Hypothese $h\mathrm{w}(\mathbf{x})$ und wir können zu diesem Zeitpunkt die Trainingsdaten verwerfen, da sie alle durch $\mathbf{w}$ zusammengefasst sind. Ein Lernmodell, das Daten mit einem Satz von Parametern fester Größe (unabhängig von der Anzahl der Trainingsbeispiele) zusammenfasst, wird als **parametrisiertes Modell** bezeichnet.

Unabhängig davon, wie viele Daten Sie einem parametrisierten Modell vorsetzen, es bleibt bei seiner Meinung, wie viele Parameter es benötigt. Bei kleinen Datensätzen ist eine strenge Einschränkung der zulässigen Hypothesen sinnvoll, um Überanpassung zu vermeiden. Wenn jedoch Tausende oder sogar Milliarden Beispiele zum Lernen verfügbar sind, scheint es besser zu sein, die Daten für sich sprechen zu lassen, anstatt sie durch einen winzigen Vektor von Parametern zu zwingen. Wenn aus den Daten hervorgeht, dass die richtige Antwort eine sehr „gezackte“ Funktion ist, sollten wir uns selbst auf lineare oder leicht gezackte Funktionen beschränken.

Bei einem **parameterfreien Modell** handelt es sich um ein Modell, das sich nicht durch eine begrenzte Menge von Parametern charakterisieren lässt. Nehmen Sie zum Beispiel an, dass jede Hypothese, die wir erzeugen, einfach alle Trainingsbeispiele in sich selbst speichert und sie alle verwendet, um das nächste Beispiel vorherzusagen. Eine derartige Hypothesenfamilie wäre parameterfrei, da die effektive Anzahl der Parameter unbeschränkt ist – sie wächst mit der Anzahl der Beispiele. Dieses Konzept bezeichnet man als **instanzbasiertes** oder **speicherbasiertes Lernen**. Die einfachste instanzbasierte Lernmethode ist die **Nachschlagetabelle**: Dabei kommen alle Trainingsbeispiele in eine Suchtabelle und wenn dann eine Abfrage nach $h(\mathbf{x})$ erscheint, wird festgestellt, ob $\mathbf{x}$ in der Tabelle vorkommt; ist das der Fall, gibt die Methode das entsprechende y zurück. Das Problem dabei ist, dass diese Methode nicht gut verallgemeinert: Ist $\mathbf{x}$ nicht in der Tabelle enthalten, kann sie bestenfalls einen bestimmten Standardwert liefern.

18.8.1 Nächste-Nachbarn-Modelle

Wir können die Tabellensuche mit einer leichten Variation verbessern: für eine Abfrage $\mathbf{x}_q$ die k Beispiele suchen, die $\mathbf{x}_q$ am nächsten kommen. Dies wird als **k-nächste-Nachbarn-Suche** bezeichnet. Wir kennzeichnen die Menge der k nächsten Nachbarn mit der Notation $NN(k, \mathbf{x}_q)$.

Um die Klassifizierung durchzuführen, suchen wir zunächst $NN(k, \mathbf{x}_q)$ und nehmen dann die Mehrheitsentscheidung der Nachbarn (die die Mehrheitsentscheidung im Fall der binären Klassifizierung ist). Um Bindungen zu vermeiden, wird für k immer eine ungerade Zahl gewählt. Für die Regression können wir den Mittelwert oder Median der k Nachbarn nehmen oder ein lineares Regressionsproblem für die Nachbarn lösen.

▶ Abbildung 18.26 zeigt die Entscheidungsgrenze der k-nächste-Nachbarn-Klassifizierung für $k = 1$ und 5 in der Erdbeben-Datenmenge von Abbildung 18.15. Parameterfreie Methoden unterliegen dennoch Unter- und Überanpassung, genau wie parametrisierte Methoden. In diesem Fall ist das 1-nächste-Nachbarn-Modell durch Überanpassung geprägt; es reagiert zu sehr auf den schwarzen Ausreißer oben rechts und den weißen Ausreißer bei (5,4; 3,7). Die 5-nächste-Nachbarn-Entscheidungsgrenze ist gut; höhere k würden zu Unteranpassung führen. Wie üblich lässt sich der beste Wert von k mithilfe einer Kreuzvalidierung auswählen.

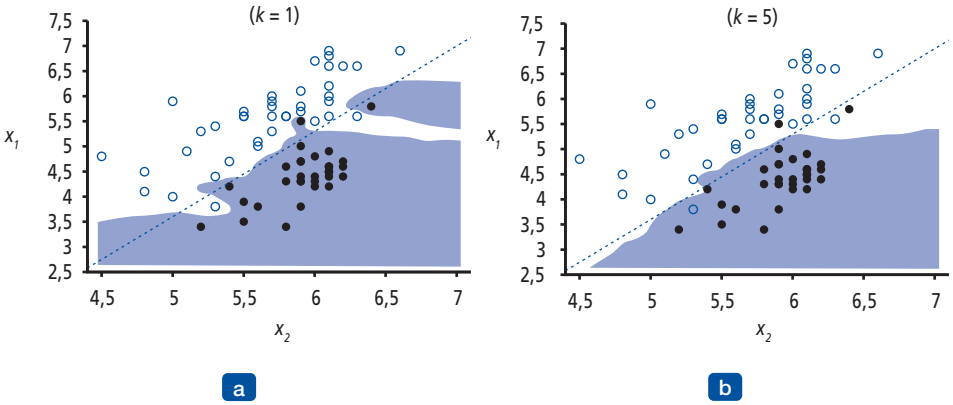


Abbildung 18.26: (a) Ein k-nächste-Nachbarn-Modell, das die Ausdehnung der Explosionsklasse für die Daten in Abbildung 18.15 zeigt, wobei $k = 1$ ist. Überanpassung ist offensichtlich. (b) Mit $k = 5$ verschwindet das Problem der Überanpassung für diese Datenmenge.

Allein das Wort „nächste“ impliziert eine Distanzmetrik. Wie messen wir nun die Entfernung von einem Abfragepunkt $\mathbf{x}_q$ zu einem Beispielpunkt $\mathbf{x}_j$? Normalerweise werden Entfernungen als **Minkowski-Distanz** oder L^p-Norm gemessen, die wie folgt definiert ist:

$$L^p(\mathbf{x}_j, \mathbf{x}_q) = \left(\sum_i |x_{j,i} - x_{q,i}|^p \right)^{1/p}.$$

Mit $p = 2$ handelt es sich um die euklidische Distanz und mit $p = 1$ um die Manhattan-Distanz. Wenn die Attribute boolesche Werte sind, bezeichnet man die Anzahl der Attribute, in denen sich die beiden Punkte unterscheiden, als **Hamming-Distanz**. Oftmals wird $p = 2$ verwendet, wenn die Dimensionen ähnliche Eigenschaften wie zum Beispiel Breite, Höhe und Tiefe oder Teile auf einem Fließband verkörpern, während

die Manhattan-Distanz für ungleichartige Eigenschaften wie zum Beispiel Alter, Gewicht und Geschlecht eines Patienten geeignet ist. Verwendet man die absoluten Zahlen von jeder Dimension, wirkt sich eine Änderung in der Skala einer beliebigen Dimension auf den Gesamtabstand aus. Wenn wir also die Dimension i nicht mehr in Zentimetern, sondern in Meilen messen und dabei die anderen Dimensionen beibehalten, erhalten wir andere nächste Nachbarn. Um dies zu vermeiden, wendet man üblicherweise eine **Normalisierung** auf die Messungen in jeder Dimension an. Ein einfacher Ansatz ist es, den Mittelwert μ_i und die Standardabweichung σ_i der Werte in jeder Dimension zu berechnen und die Werte dann neu zu skalieren, sodass $x_{j,i}$ zu $(x_{j,i} - \mu_i)/\sigma_i$ wird. Eine komplexere Metrik, die als **Mahalanobis-Distanz** bekannt ist, berücksichtigt die Kovarianz zwischen Dimensionen.

In Räumen mit wenigen Dimensionen und sehr vielen Daten funktioniert die Nächste-Nachbarn-Methode sehr gut: Wir haben wahrscheinlich genügende Nachbarschaftsdaten, um eine gute Antwort zu erhalten. Doch wenn die Anzahl der Dimensionen steigt, taucht ein Problem auf: Die nächsten Nachbarn in Räumen mit vielen Dimensionen liegen normalerweise nicht sehr nahe beieinander! Betrachten Sie die k-nächsten-Nachbarn einer Datenmenge von N Punkten, die gleichmäßig im Inneren eines n-dimensionalen Hyper-Einheitswürfels verteilt sind. Wir definieren die k-Nachbarschaft eines Punktes als den kleinsten Hyperwürfel, der die k-nächsten Nachbarn enthält. Es sei ℓ die durchschnittliche Seitenlänge einer Nachbarschaft. Dann beträgt das Volumen der Nachbarschaft (die k Punkte enthält) ℓ^n und das Volumen des vollständigen Würfels (der N Punkte enthält) ist 1. Somit ist im Mittel $\ell^n = k/N$. Zieht man auf beiden Seiten die n-ten Wurzeln, ergibt sich $\ell = (k/N)^{1/n}$.

Machen wir das an einem konkreten Beispiel mit $k = 10$ und $N = 1.000.000$ fest. In zwei Dimensionen ($n = 2$; ein Einheitsquadrat) hat die mittlere Nachbarschaft einen Wert von $\ell = 0{,}003$, d.h. einen geringen Bruchteil des Einheitsquadrates, und in drei Dimensionen ist ℓ lediglich 2% der Kantenlänge des Einheitswürfels. Doch wenn wir bei 17 Dimensionen ankommen, macht ℓ die halbe Kantenlänge des Hyper-Einheitswürfels aus und in 200 Dimensionen liegt der Wert bei 94%. Dieses Problem bezeichnet man auch als **Fluch der Dimensionalität** (▶ Abbildung 18.27).

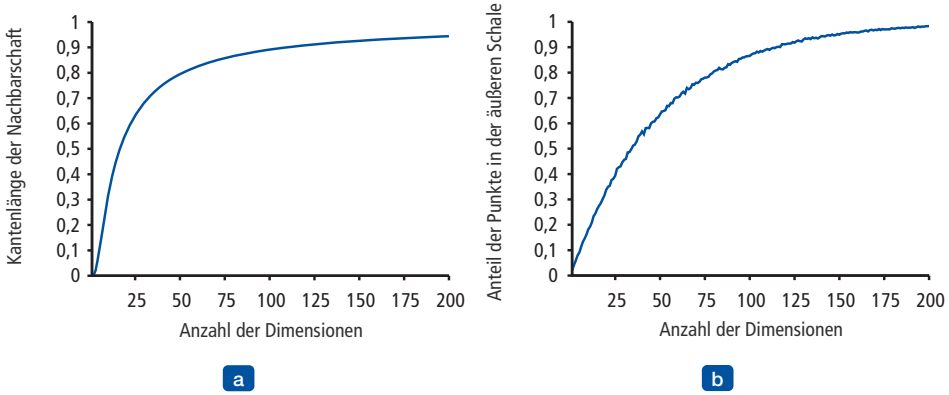


Abbildung 18.27: Der Fluch der Dimensionalität: (a) Die Länge der durchschnittlichen Nachbarschaft für 10-nächste-Nachbarn in einem Hyper-Einheitswürfel mit 1.000.000 Punkten als Funktion der Dimensionsanzahl. (b) Der Anteil der Punkte, die in eine dünne Schale der äußeren 1% des Hyperwürfels fallen, als Funktion der Dimensionsanzahl. Stichprobenauswahl von 10.000 zufällig verteilten Punkten.

Das Ganze lässt sich auch folgendermaßen betrachten: Die Punkte in einer dünnen Schale, die die äußeren 1% des Hyper-Einheitswürfels ausmacht, gelten als Ausreißer. Im Allgemeinen ist es schwer, einen guten Wert dafür zu finden, da wir extrapolieren und nicht interpolieren. In nur einer Dimension umfassen diese Ausreißer lediglich 2% der Punkte auf der Einheitslinie (diejenigen Punkte, wo $x < 0{,}01$ oder $x > 0{,}99$), doch in 200 Dimensionen fallen über 98% der Punkte in diese dünne Schale – und somit sind fast alle Punkte Ausreißer. Ein Beispiel einer schlechten Nächste-Nachbarn-Anpassung an Ausreißer sehen Sie später in diesem Kapitel in Abbildung 18.28(b).

Die Funktion $NN(k, \mathbf{x}_q)$ ist konzeptionell trivial: für eine gegebene Menge von N Beispielen und eine Abfrage $\mathbf{x}_q$ die Beispiele durchlaufen, von jedem die Distanz zu $\mathbf{x}_q$ messen und die beste k behalten. Wenn wir mit einer Implementierung zufrieden sind, die eine Ausführungszeit von $O(N)$ benötigt, ist nichts weiter zu tun. Doch instanzbasierte Methoden sind für größere Datenmengen konzipiert, sodass wir einen Algorithmus mit sublinearer Ausführungszeit anstreben. Eine elementare Algorithmenanalyse zeigt, dass die Ausführungszeiten bei exakter Tabellensuche mit einer sequentiellen Tabelle $O(N)$, mit einem Binärbaum $O(\log N)$ und mit einer Hashtabelle $O(1)$ betragen. Wir zeigen nun, dass sich Binärbäume und Hashtabellen auch auf das Suchen der nächsten Nachbarn anwenden lassen.

18.8.2 Die nächsten Nachbarn mit *k*-d-Bäumen suchen

Ein ausgeglichener Binärbaum über Daten mit einer beliebigen Anzahl von Dimensionen wird als k-dimensionaler Baum oder **k-d-Baum** bezeichnet. (Da in unserer Notation die Anzahl der Dimensionen n ist, müsste man eigentlich von n-d-Bäumen sprechen.) Die Konstruktion eines k-d-Baumes ähnelt der Konstruktion eines eindimensionalen ausgeglichenen Binärbaumes. Wir beginnen mit einer Menge von Beispielen und teilen sie am Wurzelknoten entlang der i-ten Dimension durch einen Test, ob $x_i \leq m$ ist. Als Wert m wählen wir den Median der Beispiele entlang der i-ten Dimension. Somit gelangt die Hälfte der Beispiele in den linken Zweig des Baumes und die andere Hälfte in den rechten. Wir gehen dann rekursiv vor und erstellen jeweils einen Baum für die linken und rechten Mengen der Beispiele, wobei das Ende erreicht ist, wenn weniger als zwei Beispiele übrig sind. Um eine Dimension auszuwählen, für die die Teilung an jedem Knoten des Baumes stattfindet, kann man einfach Dimension $i \bmod n$ auf Ebene i des Baumes verwenden. (Zu beachten ist, dass wir für eine gegebene Dimension nicht mehrmals aufteilen dürfen, wenn wir den Baum weiter nach unten steigen.) Eine andere Strategie besteht darin, nach der Dimension aufzuteilen, die die breiteste Streuung der Werte aufweist.

Die exakte Suche von einem k-d-Baum ist genau wie eine Suche von einem Binärbaum aus (mit der geringen Komplikation, dass Sie auf die Dimension achten müssen, die Sie an jedem Knoten testen). Die Nächste-Nachbarn-Suche ist jedoch wesentlich komplizierter. Wenn wir die Zweige nach unten gehen und dabei die Beispiele in Hälften aufteilen, können wir in manchen Fällen die andere Hälfte der Beispiele verwerfen. Allerdings nicht immer, manchmal fällt der gesuchte Punkt sehr nahe an die Teilungsgrenze. Der Abfragepunkt selbst befindet sich möglicherweise auf der linken Seite der Grenze, doch einer oder mehrere der k-nächsten Nachbarn könnten tatsächlich auf der rechten Seite liegen. Wir müssen auf diese Möglichkeit testen, indem wir die Distanz des Abfragepunktes zur Teilungsgrenze berechnen. Dann durchsuchen wir beide Seiten, wenn wir keine k Beispiele auf der linken Seite finden, die näher als diese Distanz entfernt sind. Wegen dieses Problems sind k-d-Bäume nur geeignet, wenn es mehr Beispiele als

Dimensionen gibt, vorzugsweise mindestens 2^n Beispiele. Somit funktionieren k-d-Bäume bis zu 10 Dimensionen mit Tausenden von Beispielen oder bis zu 20 Dimensionen mit Millionen von Beispielen. Sind nicht genügend Beispiele vorhanden, ist diese Suche nicht schneller als ein lineares Durchsuchen der gesamten Datenmenge.

18.8.3 Ortsabhängiges Hashing

Mit Hashtabellen lassen sich mitunter schnellere Suchen als mit Binärbäumen realisieren. Doch wie können wir die nächsten Nachbarn mithilfe einer Hashtabelle finden, wenn sich die Hashcodes auf eine exakte Übereinstimmung stützen? Hashcodes verteilen Werte zufällig unter den Gruppen, doch wir möchten, dass nahe Punkte in derselben Gruppe vereint sind – wir streben ein **ortsabhängiges Hashing** (**Locality-Sensitive Hashing**, LSH) an.

Um $NN(k, \mathbf{x}_q)$ exakt zu lösen, können wir kein Hashing verwenden, doch mit einer geschickten Verwendung von randomisierten Algorithmen lässt sich eine *Näherungslösung* ermitteln. Zuerst definieren wir das **Problem der approximierten nächsten Nachbarn**: Für eine gegebene Datenmenge von Beispielpunkten und einen Abfragepunkt $\mathbf{x}_q$ soll ein Beispielpunkt (oder mehrere Punkte), der nahe $\mathbf{x}_q$ liegt, mit hoher Wahrscheinlichkeit gesucht werden. Konkreter ausgedrückt fordern wir, dass der Algorithmus mit hoher Wahrscheinlichkeit einen Punkt $\mathbf{x}_j'$, der innerhalb der Distanz $c\,r$ von q liegt, findet, wenn es einen Punkt $\mathbf{x}_j$ gibt, der innerhalb eines Radius r von $\mathbf{x}_q$ liegt. Gibt es keinen Punkt innerhalb des Radius r, darf der Algorithmus einen Fehler melden. Die Werte von c und „hoher Wahrscheinlichkeit“ sind Parameter des Algorithmus.

Um das Problem der approximierten nächsten Nachbarn zu lösen, brauchen wir eine Hashfunktion $g(x)$ mit der Eigenschaft, dass zwei Punkte $\mathbf{x}_j$ und $\mathbf{x}_j'$ mit nur geringer Wahrscheinlichkeit den gleichen Hashcode haben, wenn ihr Abstand größer als $c\,r$ ist, und mit hoher Wahrscheinlichkeit den gleichen Hashcode aufweisen, wenn ihr Abstand kleiner als r ist. Der Einfachheit halber behandeln wir jeden Punkt als Bit-String. (Features, die nicht boolesch sind, lassen sich in einem Satz von booleschen Features kodieren.) Wenn zwei Punkte in einem n-dimensionalen Raum nahe beieinanderliegen, verlassen wir uns intuitiv darauf, dass sie dann zwangsläufig eng zusammenliegen, wenn sie auf einen eindimensionalen Raum (eine Linie) projiziert werden. In der Tat können wir die Linie in Abschnitte – sogenannte Hash-Buckets – diskretisieren, sodass mit hoher Wahrscheinlichkeit nahe beieinanderliegende Punkte auf genau denselben Abschnitt projiziert werden. Weit voneinander entfernte Punkte werden bei den meisten Projektionen eher in verschiedene Abschnitte projiziert, doch gibt es immer einige Projektionen, die zufälligerweise weit voneinander entfernte Punkte in denselben Abschnitt projizieren. Folglich enthält der Abschnitt für Punkt $\mathbf{x}_q$ viele (jedoch nicht alle) Punkte, die $\mathbf{x}_q$ nahe sind, sowie einige Punkte, die weit entfernt sind.

Der Trick von LSH besteht darin, *mehrere* Zufallsprojektionen zu erzeugen und sie zu kombinieren. Eine Zufallsprojektion ist einfach eine zufällige Teilmenge der Bit-String-Darstellung. Wir wählen ℓ verschiedene Zufallsprojektionen, erzeugen ℓ Hashtabellen $g_1(\mathbf{x})$, ..., $g_\ell(\mathbf{x})$ und geben dann alle Beispiele in die einzelnen Hashtabellen ein. Wird ein Abfragepunkt $\mathbf{x}_q$ gegeben, rufen wir die Menge der Punkte im Behälter $g_k(q)$ für jedes k ab und bilden die Vereinigung dieser Mengen zu einer Menge von Kandidatenpunkten C. Anschließend berechnen wir den tatsächlichen Abstand zu $\mathbf{x}_q$ für jeden der Punkte in C und geben die k nächsten Punkte zurück. Mit hoher Wahrscheinlichkeit erscheint jeder

der Punkte, die nahe zu $\mathbf{x}_q$ liegen, in mindestens einem der Behälter. Zwar sind darunter auch einige weit entfernte Punkte, doch diese können wir ignorieren. Mit großen Problemen in realen Anwendungen wie zum Beispiel das Suchen der nächsten Nachbarn in einer Datenmenge von 13 Millionen Webbildern mithilfe von 512 Dimensionen (Torralba et al., 2008) muss ortsabhängiges Hashing nur einige tausend Bilder aus den 13 Millionen untersuchen, um die nächsten Nachbarn zu finden – eine tausendfache Geschwindigkeitssteigerung gegenüber den erschöpfenden oder *k*-d-Baum-Verfahren.

18.8.4 Parameterfreie Regression

Wir sehen uns nun parameterfreie Konzepte für *Regression* statt Klassifizierung an. ▶ Abbildung 18.28 zeigt ein Beispiel für verschiedene Modelle. In (a) haben wir wahrscheinlich die einfachste Methode von allen, die formlos als „Punkte verbinden“ und etwas hochtrabend als „stückweise lineare parameterfreie Regression“ bekannt ist. Dieses Modell erzeugt eine Funktion $h(x)$, die für eine gegebene Abfrage x_q das gewöhnliche lineare Regressionsproblem mit lediglich zwei Punkten löst: den Trainingsbeispielen unmittelbar links und rechts von x_q. Bei geringem Rauschen ist diese triviale Methode tatsächlich recht brauchbar und deshalb auch als Standardfeature von Diagrammsoftware in Tabellenkalkulationen realisiert.

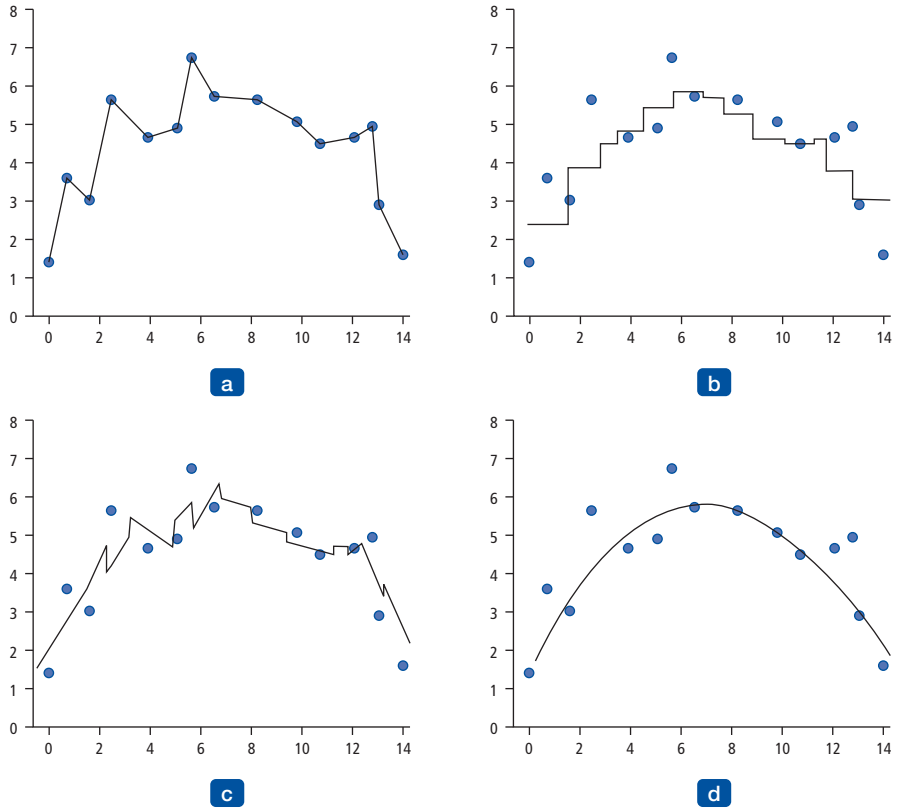


Abbildung 18.28: Parameterfreie Regressionsmodelle: (a) Punkte verbinden, (b) 3-nächste-Nachbarn-Durchschnitt, (c) lineare 3-nächste-Nachbarn-Regression, (d) lokal gewichtete Regression mit einem quadratischen Kernel der Breite $k = 10$.

Wenn aber die Daten verrauscht sind, ist die resultierende Funktion gezackt und verallgemeinert nicht gut.

k-nächste-Nachbarn-Regression (Abbildung 18.28(b)) verbessert die Methode „Punkte verbinden“. Anstatt nur die beiden Beispiele links und rechts von einem Abfragepunkt x_q zu verwenden, ziehen wir die k-nächsten-Nachbarn (hier mit $k = 3$) heran. Ein größerer Wert von k gleicht zwar die Höhe der Spitzen etwas aus, führt aber auch zu einer Funktion, die Unstetigkeiten aufweist. Abbildung 18.28(b) zeigt den k-nächste-Nachbarn-Durchschnitt: $h(x)$ ist der Mittelwert der k Punkte $\Sigma y_j/k$. Beachten Sie, dass die Schätzungen bei den abseits gelegenen Punkten nahe $x = 0$ und $x = 14$ schlecht ausfallen, da die gesamte Evidenz von einer Seite (dem Innenraum) stammt und der Trend ignoriert wird. In Abbildung 18.28(c) ist die lineare k-nächste-Nachbarn-Regression zu sehen, die die beste Linie durch die k Beispiele findet. Diese Methode erfasst zwar die Trends bei den Ausreißern besser, ist aber immer noch unstetig. Sowohl in (b) als auch in (c) bleibt die Frage offen, wie man einen guten Wert für k wählt. Wie üblich lautet die Antwort Kreuzvalidierung.

Lokal gewichtete Regression (Abbildung 18.28(d)) bietet die Vorzüge der nächsten Nachbarn, jedoch ohne Unstetigkeiten. Um Unstetigkeiten in $h(x)$ zu vermeiden, müssen wir Unstetigkeiten in der Menge der für die Schätzung von $h(x)$ verwendeten Beispiele vermeiden. Bei lokal gewichteter Regression werden an jedem Abfragepunkt x_q die nahe bei x_q liegenden Beispiele stark gewichtet und die weiter weg liegenden Beispiele weniger stark oder überhaupt nicht. Die Abnahme des Gewichtes über die Distanz erfolgt immer allmählich und nicht abrupt.

Mit einer als **Kernel** bezeichneten Funktion entscheiden wir, wie stark jedes Beispiel zu gewichten ist. Eine Kernel-Funktion sieht wie ein Höcker aus; in ▶ Abbildung 18.29 ist der Kernel zu sehen, mit dem Abbildung 18.28(d) generiert wurde. Das von diesem Kernel zugeordnete Gewicht ist in der Mitte am höchsten und erreicht null bei einem Abstand von ± 5. Lässt sich jede beliebige Funktion für einen Kernel verwenden? Nein. Erstens ist festzustellen, dass wir eine Kernel-Funktion $\mathcal{K}$ mit $\mathcal{K}(Distanz(\mathbf{x}_j, \mathbf{x}_q))$ aufrufen, wobei $\mathbf{x}_q$ ein Abfragepunkt ist, der bei einer gegebenen Distanz von $\mathbf{x}_j$ liegt, und wir wissen möchten, wie groß das Gewicht bei dieser Distanz ist.

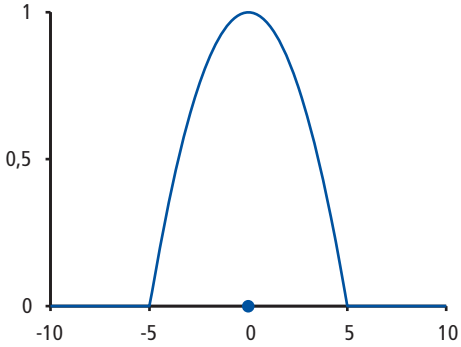


Abbildung 18.29: Eine quadratische Kernel-Funktion $\mathcal{K}(d) = \max(0; 1 - (2|x|/k)^2)$ mit einer Kernelbreite $k = 10$, zentriert zum Abfragepunkt $x = 0$.

Somit sollte $\mathcal{K}$ um 0 symmetrisch sein und dort ein Maximum haben. Die Fläche unter dem Kernel muss begrenzt bleiben, wenn wir nach $\pm\infty$ gehen. Es sind auch andere Kon-

turen wie zum Beispiel die Gaußverteilung als Kernel verwendet worden, doch die neuesten Forschungen legen nahe, dass die Wahl der Kontur keine wesentliche Rolle spielt. Wir müssen allerdings auf die Breite des Kernels achten. Auch dies ist wieder ein Parameter des Modells, der sich am besten durch Kreuzvalidierung ermitteln lässt. Genau wie bei der Auswahl des k für die nächsten Nachbarn kommt es zur Unteranpassung, wenn die Kernel zu breit sind, und zu enge Kernel führen zu einer Überanpassung. In Abbildung 18.28(d) ergibt der Wert $k = 10$ eine glatte Kurve, die recht gut aussieht – doch möglicherweise wird der Ausreißer bei $x = 6$ nicht genügend berücksichtigt; ein engerer Kernel würde mehr auf einzelne Punkte reagieren.

Lokal gewichtete Regression mit Kerneln ist nun unkompliziert auszuführen. Für einen gegebenen Abfragepunkt $\mathbf{x}_q$ lösen wir das folgende gewichtete Regressionsproblem mithilfe des Gradientenabstieges:

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_j K\Big(\text{Distanz}\big(\mathbf{x}_q, \mathbf{x}_j\big)\Big)\big(y_j - \mathbf{w} \cdot \mathbf{x}_j\big)^2.$$

Hier ist *Distanz* eine der Distanzmetriken, die für die nächsten Nachbarn besprochen wurden. Die Antwort lautet dann $h(\mathbf{x}_q) = \mathbf{w}^* \cdot \mathbf{x}_q$.

Beachten Sie, dass für *jeden* Abfragepunkt ein neues Regressionsproblem zu lösen ist – das ist es, was lokal bedeutet. (In normaler linearer Regression haben wir das Regressionsproblem lediglich einmal und global gelöst und dann dieselbe $h\mathbf{w}$ für jeden Abfragepunkt verwendet. Dieser zusätzliche Aufwand wird etwas durch die Tatsache gemildert, dass sich jedes Regressionsproblem leichter lösen lässt, weil nur die Beispiele mit einem Gewicht ungleich null betroffen sind – die Beispiele, deren Kernel den Abfragepunkt überlappen. Bei geringen Kernelbreiten sind dies möglicherweise nur wenige Punkte.

Die meisten parameterfreien Modelle besitzen den Vorzug, dass sich eine Leave-One-Out-Kreuzvalidierung leicht durchführen lässt, ohne alles neu berechnen zu müssen. So rufen wir bei einem k-nächste-Nachbarn-Modell für ein gegebenes Testbeispiel $(\mathbf{x}, y)$ den k-nächsten Nachbarn lediglich einmal ab, berechnen den Verlust pro Beispiel $L(y, h(\mathbf{x}))$ von ihnen und zeichnen dies als Leave-One-Out-Ergebnis für jedes Beispiel auf, das nicht zu den Nachbarn gehört. Dann rufen wir die $k + 1$ nächsten Nachbarn ab und zeichnen andere Ergebnisse für das Auslassen jedes der k Nachbarn auf. Bei N Beispielen benötigt der gesamte Prozess eine Zeit von $O(k)$ und nicht von $O(kN)$.

18.9 Support-Vector-Maschinen

Die **Support-Vector-Maschine** oder SVM ist der derzeit populärste Ansatz für überwachtes Lernen „von der Stange“: Wenn Sie kein spezialisiertes A-priori-Wissen über eine Domäne haben, lohnt es sich, die SVM als Erstes auszuprobieren. Drei Eigenschaften machen SVMs attraktiv:

1. SVMs konstruieren einen **Maximum-Margin-Separator** (etwa: Separator mit maximalem Rand) – eine Entscheidungsgrenze mit der größtmöglichen Distanz zu Beispielpunkten. Dies hilft, sie gut zu verallgemeinern.

2. SVMs erzeugen eine linear trennende Hyperebene, besitzen aber die Fähigkeit, mithilfe des sogenannten **Kernel-Tricks** die Daten in einen Raum höherer Dimension einzubetten. Oftmals lassen sich Daten, die im ursprünglichen Eingaberaum nicht separierbar sind, im Raum höherer Dimension leicht trennen. Der lineare

Separator höherer Dimension ist im ursprünglichen Raum nichtlinear. Der Hypothesenraum wird somit erheblich über Methoden erweitert, die streng lineare Darstellungen verwenden.

3 SVMs sind parameterfreie Methoden – sie behalten Trainingsbeispiele bei und müssen sie gegebenenfalls alle speichern. Andererseits behalten sie in der Praxis oftmals nur einen Bruchteil der Beispiele bei – manchmal nur eine kleine Konstante multipliziert mit der Anzahl der Dimensionen. Somit kombinieren SVMs die Vorteile von parameterfreien und parametrisierten Modellen: Sie besitzen die Flexibilität, komplexe Funktionen darzustellen, sind aber resistent gegen Überanpassung.

Man könnte sagen, dass SVMs aufgrund einer entscheidenden Erkenntnis und eines raffinierten Tricks erfolgreich sind. Beide Aspekte behandeln wir nacheinander. ▶ Abbildung 18.30(a) zeigt ein binäres Klassifizierungsproblem mit drei Kandidaten für Entscheidungsgrenzen, die jeweils einen linearen Separator darstellen. Jeder von ihnen ist mit allen Beispielen konsistent, sodass aus der Perspektive des 0/1-Verlustes jeder gleich gut geeignet wäre. Die logistische Regression würde eine bestimmte Trennungslinie finden; die genaue Lage der Linie hängt von *allen* Beispielpunkten ab. Die entscheidende Erkenntnis von SVMs ist, dass bestimmte Beispiele wichtiger als andere sind und es zu einer besseren Verallgemeinerung führen kann, wenn man sie beachtet.

Sehen Sie sich dazu die unterste der drei Trennungslinien in (a) an. Sie kommt der 5 des schwarzen Beispiels sehr nahe. Obwohl sie alle Beispiele korrekt klassifiziert und somit den Verlust minimiert, sollte es Sie beunruhigen, dass so viele Beispiele der Geraden nahe kommen; es kann sich durchaus noch herausstellen, dass andere schwarze Beispiele auf die andere Seite der Geraden fallen.

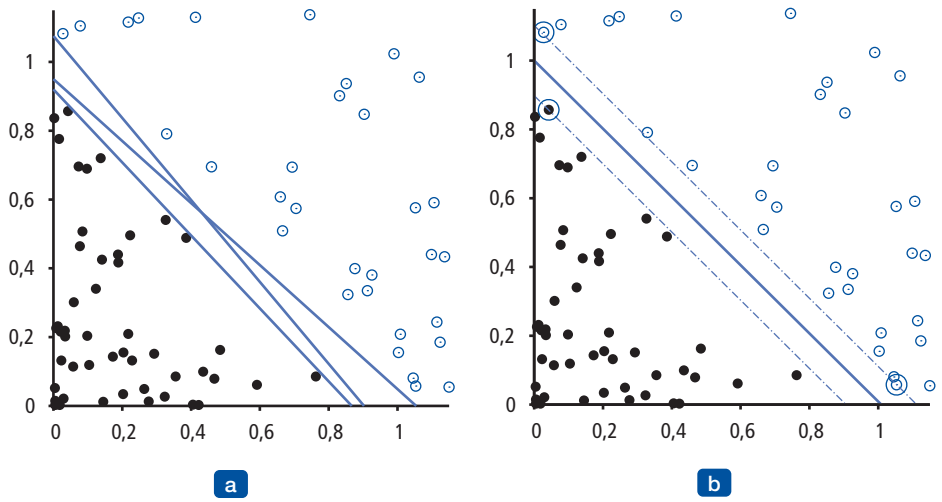


Abbildung 18.30: Support-Vector-Maschinen-Klassifizierung: (a) Zwei Klassen von Punkten (schwarze und weiße Kreise) und drei Kandidaten für lineare Trennung. (b) Der Maximum-Margin-Separator (dicke Linie) befindet sich in der Mitte des Randbereiches (Fläche zwischen den gestrichelten Linien). Die Support-Vektoren (Punkte mit großen Kreisen) sind die Beispiele, die dem Separator am nächsten liegen.

SVMs nehmen sich dieses Problems an: Anstatt den erwarteten *empirischen Verlust* bezüglich der Trainingsdaten zu minimieren, versuchen SVMs den erwarteten *Generalisierungsverlust* zu minimieren. Wir wissen nicht, wohin die noch unbekannten Punkte fallen werden, doch unter der probabilistischen Annahme, dass sie aus derselben Verteilung wie die vorher gesehenen Beispiele gezogen wurden, können wir gemäß der Computer-Lerntheorie (*Abschnitt 18.5*) den Generalisierungsverlust minimieren, indem wir den Separator wählen, der am weitesten von den bisher gesehenen Beispielen entfernt liegt. Diesen Separator, den ▶ Abbildung 18.30(b) zeigt, bezeichnen wir als **Maximum-Margin-Separator** (etwa: Separator für maximalen Rand). Der Rand (Margin) ist die Breite des Bereiches, der in der Abbildung durch die Strichellinien begrenzt wird – der doppelte Abstand vom Separator zum nächsten Beispielpunkt.

Wie finden wir nun diesen Separator? Bevor wir die Gleichungen vorstellen, zunächst eine Anmerkung zur Notation: Herkömmliche SVMs verwenden die Konvention, dass Klassen mit $+1$ und -1 bezeichnet werden, anstelle von $+1$ und 0, wie wir es bislang verwendet haben. Und wo wir den Achsenschnittpunkt in den Gewichtsvektor $\mathbf{w}$ (und einen entsprechenden Dummy-Wert 1 in $x_{j,0}$) eingefügt haben, ist das bei SVMs nicht so; sie behalten den Achsenschnittpunkt als separaten Parameter b bei. Dementsprechend ist der Separator als Menge von Punkten $\{\mathbf{x} : \mathbf{w} \cdot \mathbf{x} + b = 0\}$ definiert. Wir könnten den Raum von $\mathbf{w}$ und b mit Gradientenabstieg durchsuchen, um die Parameter zu ermitteln, die den Rand maximieren, während alle Beispiele korrekt klassifiziert werden.

Es zeigt sich aber, dass es einen anderen Ansatz gibt, um dieses Problem zu lösen. Auf die Details gehen wir hier nicht ein, sondern wir weisen nur darauf hin, dass die duale Darstellung eine alternative Darstellung ist, in der sich die optimale Lösung finden lässt, wenn man

$$\underset{\alpha}{\text{argmax}} \sum_j \alpha_j - \frac{1}{2}\sum_{j,k} \alpha_j \alpha_k y_j y_k \left(\mathbf{x}_j \cdot \mathbf{x}_k\right) \tag{18.13}$$

unter den Bedingungen $\alpha_j \geq 0$ und $\Sigma_j \alpha_j y_j = 0$ auflöst. Für diesen als **quadratische Programmierung** bezeichneten speziellen Typ eines mathematischen Optimierungsproblems sind gute Softwarepakete verfügbar. Nachdem wir den Vektor α gefunden haben, können wir nach $\mathbf{w}$ mit der Gleichung $\mathbf{w} = \Sigma_j \alpha_j \mathbf{x}_j$ zurückgehen oder in der dualen Darstellung bleiben. Gleichung (18.13) besitzt drei wichtige Eigenschaften. Erstens ist der Ausdruck konvex; er hat ein einzelnes globales Maximum, das sich effizient finden lässt. Zweitens *gehen die Daten in den Ausdruck nur in der Form von Punktprodukten aus Punktpaaren ein*. Diese zweite Eigenschaft trifft auch auf die Gleichung für den Separator selbst zu; nachdem die optimalen a_j berechnet wurden, ergibt sich

$$h(\mathbf{x}) = \text{sign}\left(\sum_j \alpha_j y_j \left(\mathbf{x} \cdot \mathbf{x}_j\right) - b\right). \tag{18.14}$$

Schließlich sind die Gewichte α_j, die jedem Datenpunkt zugeordnet sind, *null*, mit Ausnahme der **Support-Vektoren** – den Punkten, die dem Separator am nächsten liegen. (Sie heißen „Support-Vektoren“, weil sie die trennende Ebene „unterstützen“.) Da es normalerweise viel weniger Support-Vektoren als Beispiele gibt, bringen SVMs einige der Vorteile von parametrisierten Modellen mit.

Wie sieht es nun aus, wenn Beispiele nicht linear separierbar sind? ▶ Abbildung 18.31(a) zeigt einen Eingaberaum, der durch Attribute $\mathbf{x} = (x_1, x_2)$ definiert ist, wobei die positiven Beispiele ($y = +1$) innerhalb eines kreisförmigen Bereiches liegen und die negativen Beispiele ($y = -1$) außerhalb. Für dieses Problem gibt es offensichtlich keinen linearen Separator. Nehmen Sie nun an, wir formulieren die Eingabedaten neu – d.h., wir bilden jeden Eingabevektor $\mathbf{x}$ auf einen neuen Vektor von Featurewerten $F(\mathbf{x})$ ab. Insbesondere wollen wir die folgenden drei Features verwenden:

$$f_1 = {x_1}^2, \quad f_2 = {x_2}^2, \quad f_3 = \sqrt{2}\, x_1 x_2 \,. \tag{18.15}$$

Wir zeigen gleich, woher diese stammen, doch sehen Sie sich erst einmal an, was passiert. ▶ Abbildung 18.31(b) zeigt die Daten im neuen, dreidimensionalen Raum, der durch die drei Features definiert ist; die Daten sind in diesem Raum linear separierbar! Dieses Phänomen ist tatsächlich recht allgemeingültig: Wenn Daten in einen Raum von genügend hoher Dimension abgebildet werden, sind sie fast immer linear separierbar – wenn Sie sich eine Menge von Datenpunkten aus genügend Richtungen ansehen, finden Sie eine Möglichkeit, sie in einer Reihe anzuordnen. Hier haben wir nur drei Dimensionen verwendet;[11] in Übung 18.18 sollen Sie zeigen, dass vier Dimensionen für eine lineare Trennung eines Kreises an einer beliebigen Stelle in der Ebene (nicht nur im Mittelpunkt) genügen und fünf Dimensionen ausreichend sind, um eine Ellipse linear zu trennen. Allgemein gilt (von einigen Spezialfällen abgesehen): Wenn wir N Datenpunkte haben, sind diese in Räumen von $N - 1$ Dimensionen oder mehr immer linear separierbar (Übung 18.29).

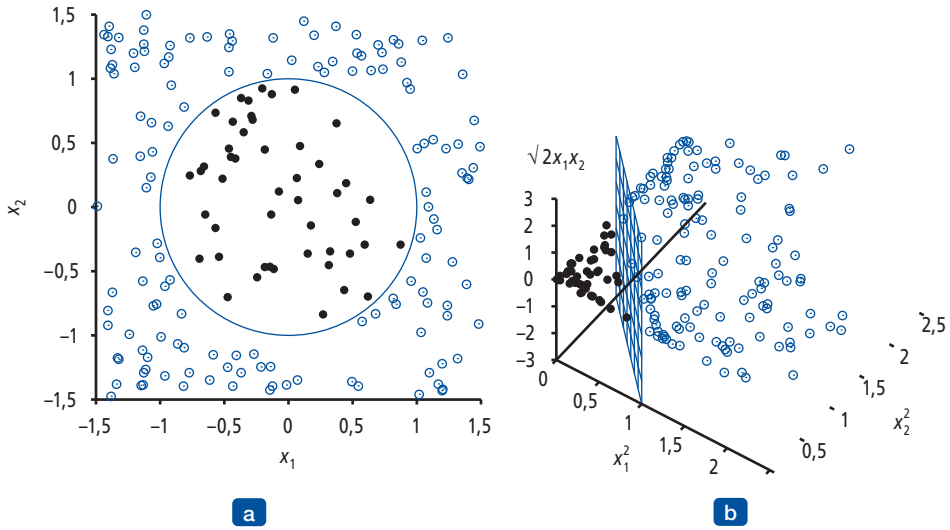


Abbildung 18.31: (a) Eine zweidimensionale Trainingsmenge mit positiven Beispielen als schwarze Kreise und negativen Beispielen als weiße Kreise. Die wahre Entscheidungsgrenze ${x_1}^2 + {x_2}^2 \leq 1$ ist ebenfalls dargestellt. (b) Die gleichen Daten nach Abbildung in einen dreidimensionalen Eingaberaum $({x_1}^2, {x_2}^2, \sqrt{2}\, x_1 x_2)$. Die kreisförmige Entscheidungsgrenze in (a) wird in drei Dimensionen zu einer linearen Entscheidungsgrenze. Abbildung 18.30(b) zeigt eine Nahaufnahme des Separators in (b).

11 Der Leser hat vielleicht bemerkt, dass wir einfach f_1 und f_2 hätten verwenden können, doch lässt sich die Idee mithilfe der 3D-Abbildung besser veranschaulichen.

Nun werden wir normalerweise nicht erwarten, einen linearen Separator im Eingaberaum $\mathbf{x}$ zu finden, doch können wir lineare Separatoren im Featureraum mit höherer Dimension $F(\mathbf{x})$ finden, indem wir einfach $\mathbf{x}_j \cdot \mathbf{x}_k$ in Gleichung (18.13) durch $F(\mathbf{x}_j) \cdot F(\mathbf{x}_k)$ ersetzen. Dies ist an sich nicht bemerkenswert – das Ersetzen von $\mathbf{x}$ durch $F(\mathbf{x})$ hat in *jedem* Lernalgorithmus die gewünschte Wirkung –, doch besitzt das Punktprodukt einige spezielle Eigenschaften. Es zeigt sich, dass $F(\mathbf{x}_j) \cdot F(\mathbf{x}_k)$ oftmals berechnet werden kann, ohne zuerst F für jeden Punkt zu berechnen. In unserem dreidimensionalen Featureraum, der durch Gleichung (18.15) definiert ist, lässt sich mit etwas Algebra zeigen, dass

$$F(\mathbf{x}_j) \cdot F(\mathbf{x}_k) = (\mathbf{x}_j \cdot \mathbf{x}_k)^2$$

ist. (Aus diesem Grund erscheint $\sqrt{2}$ in f_3.) Der Ausdruck $(\mathbf{x}_j \cdot \mathbf{x}_k)^2$ wird als **Kernel-Funktion**[12] bezeichnet und normalerweise als $K(\mathbf{x}_j, \mathbf{x}_k)$ geschrieben. Die Kernel-Funktion lässt sich auf Paare von Eingabedaten anwenden, um Punktprodukte in einem korrespondierenden Featureraum zu bewerten. Wir können also lineare Separatoren im höherdimensionalen Featureraum $F(\mathbf{x})$ finden, indem wir einfach $\mathbf{x}_j \cdot \mathbf{x}_k$ in Gleichung (18.13) durch eine Kernel-Funktion $K(\mathbf{x}_j, \mathbf{x}_k)$ ersetzen. Somit können wir im höherdimensionalen Raum zwar lernen, aber nur Kernel-Funktionen berechnen und nicht die vollständige Liste von Features für jeden Datenpunkt.

Im nächsten Schritt ist zu zeigen, dass es nichts Besonderes mit dem Kernel $K(\mathbf{x}_j, \mathbf{x}_k) = (\mathbf{x}_j \cdot \mathbf{x}_k)^2$ auf sich hat. Er entspricht einem bestimmten höherdimensionalen Featureraum, doch entsprechen andere Kernel-Funktionen anderen Featureräumen. Als beachtenswertes Ergebnis in der Mathematik sagt uns der **Satz von Mercer** (1909), dass jede „akzeptable“[13] Kernel-Funktion einem bestimmten Featureraum entspricht. Diese Featureräume können sehr groß sein, selbst für harmlos aussehende Kernel. Zum Beispiel entspricht der **polynomiale Kernel** $K(\mathbf{x}_j, \mathbf{x}_k) = (1 + \mathbf{x}_j \cdot \mathbf{x}_k)^d$ einem Featureraum, dessen Dimension in d exponentiell ist.

Der Kernel-Trick sieht nun folgendermaßen aus: Wenn man diese Kernel in Gleichung (18.13) einsetzt, *lassen sich optimale lineare Separatoren in Featureräumen mit Milliarden (oder in manchen Fällen unendlich vielen) Dimensionen effizient finden*. Transformiert man die resultierenden linearen Separatoren wieder in den ursprünglichen Eingaberaum zurück, können sie beliebig gezackten, nichtlinearen Entscheidungsgrenzen zwischen den positiven und negativen Beispielen entsprechen.

Im Fall inhärent verrauschter Daten sind wir nicht an einem linearen Separator in irgendeinem hochdimensionalen Raum interessiert. Stattdessen streben wir eine Entscheidungsoberfläche in einem Raum mit weniger Dimensionen an, die zwar die Klassen nicht ganz sauber trennt, aber die Wirklichkeit der verrauschten Daten widerspiegelt. Dies ist mit dem **Soft-Margin**-Klassifikator möglich. Er lässt es zwar zu, dass Beispiele auf die falsche Seite der Entscheidungsgrenze fallen, weist ihnen jedoch einen Bestrafungswert proportional zum Abstand zu, der erforderlich ist, um sie auf die richtige Seite zurückzubringen.

Die Kernel-Methode lässt sich nicht nur bei Lernalgorithmen anwenden, die optimal lineare Separatoren finden, sondern auch mit jedem anderen Algorithmus, der neu formuliert werden kann, um wie in den Gleichungen (18.13) und (18.14) nur mit Punktprodukten von Datenpunktpaaren zu arbeiten. Nachdem dies geschehen ist, ersetzt man

12 Diese Verwendung von „Kernel-Funktion“ unterscheidet sich leicht von den Kernen in lokal gewichteter Regression. Manche SVM-Kerne sind Abstandsmetriken, jedoch nicht alle.

13 Hier bedeutet „akzeptabel“, dass die Matrix $\mathbf{K}_{jk} = K(\mathbf{x}_j, \mathbf{x}_k)$ positiv definiert ist.

das Punktprodukt durch eine Kernel-Funktion und kommt so zu einer Kernel-Version des Algorithmus. Unter anderem ist dies für k-nächste-Nachbarn- und Perzeptron-Lernen (*Abschnitt 18.7.2*) ohne Weiteres realisierbar.

18.10 Gruppenlernen

Bisher haben wir Lernmethoden betrachtet, wobei eine einzelne Hypothese, die aus einem Hypothesenraum ausgewählt wurde, für die Vorhersagen verwendet wird. Das Konzept der **Gruppenlernmethoden** ist, eine ganze **Gruppe** von Hypothesen aus dem Hypothesenraum auszuwählen und ihre Vorhersagen zu kombinieren. Beispielsweise könnten wir während der Kreuzvalidierung zwanzig verschiedene Entscheidungsbäume erstellen und sie zur besten Klassifizierung eines neuen Beispiels abstimmen lassen.

Die Motivierung für das Gruppenlernen ist einfach. Betrachten Sie eine Gruppe von $K = 5$ Hypothesen und gehen Sie davon aus, dass wir ihre Vorhersagen unter Verwendung einer einfachen Mehrheitswahl kombinieren. Damit die Gruppe ein neues Beispiel falsch klassifiziert, müssen *mindestens drei der fünf Hypothesen es falsch klassifizieren*. Die Hoffnung ist, dass dies sehr viel weniger wahrscheinlich als eine Fehlklassifizierung durch eine einzige Hypothese ist. Angenommen, wir gehen davon aus, dass jede Hypothese h_k in der Gruppe den Fehler p hat – d.h., die Wahrscheinlichkeit, dass ein zufällig gewähltes Beispiel durch h_k falsch klassifiziert wird, ist gleich p. Darüber hinaus wollen wir annehmen, dass die von jeder Hypothese erzeugten Fehler voneinander *unabhängig* sind. In diesem Fall ist, wenn p klein ist, die Wahrscheinlichkeit einer großen Anzahl an Fehlklassifizierungen sehr klein. Beispielsweise zeigt eine einfache Berechnung (Übung 18.20), dass die Verwendung einer Gruppe von fünf Hypothesen eine Fehlerrate von 1 in 10 auf eine Fehlerrate von weniger als 1 in 100 reduziert. Offensichtlich ist die Annahme der Unabhängigkeit unvernünftig, weil Hypothesen wahrscheinlich durch irreführende Aspekte der Trainingsdaten auf dieselbe Weise irregeführt werden. Wenn sich die Hypothesen jedoch zumindest ein bisschen unterscheiden und dabei die Korrelation zwischen ihren Fehlern reduzieren, kann das Gruppenlernen sehr sinnvoll sein.

Man könnte sich das Gruppenkonzept auch als generische Methode vorstellen, den Hypothesenraum zu vergrößern. Das bedeutet, man kann sich die eigentliche Gruppe als Hypothese vorstellen und den neuen Hypothesenraum als die Menge aller möglichen Gruppen, die aus den Hypothesen im ursprünglichen Raum zusammengesetzt werden können. ▶ Abbildung 18.32 zeigt, wie dies zu einem ausdrucksstärkeren Hypothesenraum führen kann. Wenn der ursprüngliche Hypothesenraum einen einfachen und effizienten Lernalgorithmus unterstützt, bietet die Gruppenmethode eine Möglichkeit, eine sehr viel ausdrucksstärkere Klasse von Hypothesen zu lernen, ohne dass dazu sehr viel mehr rechentechnischer oder algorithmischer Aufwand erforderlich wäre.

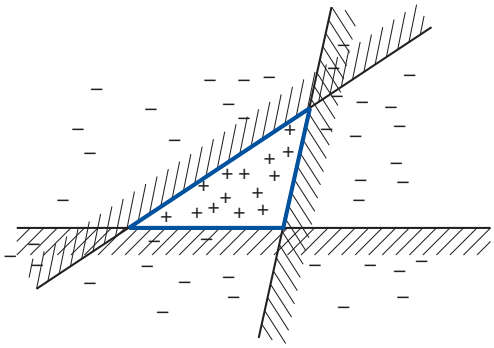


Abbildung 18.32: Darstellung der gesteigerten Ausdruckskraft durch Gruppenlernen. Wir nehmen drei Hypothesen mit linearen Schwellen, die jeweils die nicht schattierte Seite positiv klassifizieren und die zusammen ein Beispiel als positiv klassifizieren, das von allen dreien als positiv klassifiziert wurde. Der resultierende dreieckige Bereich ist eine Hypothese, die im ursprünglichen Hypothesenraum nicht ausdrückbar ist.

Die gebräuchlichste Gruppenmethode ist das sogenannte **Boosting** (Verstärken). Um zu verstehen, wie es funktioniert, müssen wir zuerst das Konzept der **gewichteten Trainingsmenge** erklären. In einer solchen Trainingsmenge ist jedem Beispiel ein Gewicht $w_j \geq 0$ zugeordnet. Je höher die Gewichtung eines Beispiels ist, desto höher ist die Bedeutung, die ihm während des Lernens einer Hypothese zugeordnet wird. Man kann die bisher gezeigten Lernalgorithmen ganz einfach anpassen, sodass sie mit gewichteten Trainingsmengen arbeiten.[14]

Das Boosting beginnt mit $w_j = 1$ für alle Beispiele (d.h. eine normale Trainingsmenge). Aus dieser Menge erzeugt es die erste Hypothese, h_1. Diese Hypothese klassifiziert einige der Trainingsbeispiele korrekt, andere falsch. Wir möchten, dass die nächste Hypothese eine bessere Leistung für die falsch klassifizierten Beispiele bringt; deshalb erhöhen wir ihre Gewichte, während wir die Gewichte der korrekt klassifizierten Beispiele verringern. Aus dieser neu gewichteten Trainingsmenge erzeugen wir die Hypothese h_2. Dieser Prozess wird fortgesetzt, bis wir K Hypothesen erzeugt haben, wobei K eine Eingabe für den Boosting-Algorithmus ist. Die fertige Gruppenhypothese ist eine gewichtete Mehrheitskombination aller K Hypothesen, die alle danach gewichtet sind, wie gut sie sich für die Trainingsmenge erwiesen haben. ▶ Abbildung 18.33 zeigt, wie der Algorithmus vom Konzept her arbeitet. Es gibt viele Varianten der grundlegenden Boosting-Idee mit unterschiedlicher Anpassung der Gewichte und Kombination der Hypothesen. Ein spezieller Algorithmus, ADABOOST, ist in ▶ Abbildung 18.34 gezeigt. ADABOOST besitzt eine sehr wichtige Eigenschaft: Wenn der Eingabelernalgorithmus L ein **schwacher Lern**algorithmus ist – d.h., L gibt immer eine Hypothese mit einer Genauigkeit für die Trainingsmenge zurück, die nur wenig besser als ein zufälliges Raten ist (also 50% + ϵ für eine boolesche Klassifizierung) –, dann gibt ADABOOST eine Hypothese zurück, die *die Trainingsdaten perfekt klassifiziert*, wenn K groß genug ist. Der Algorithmus *verstärkt* also die Genauigkeit des ursprünglichen Lernalgorithmus für die Trainingsdaten. Das Ergebnis gilt, egal wie wenig ausdrucksstark der ursprüngliche Hypothesenraum war und wie komplex die zu lernende Funktion ist.

14 Für Lernalgorithmen, die dies nicht erlauben, kann man stattdessen eine replizierte Trainingsmenge erstellen, wobei das j-te Beispiel w_j-mal erscheint. Dabei wird Randomisierung verwendet, um Bruchteile von Gewichten zu behandeln.

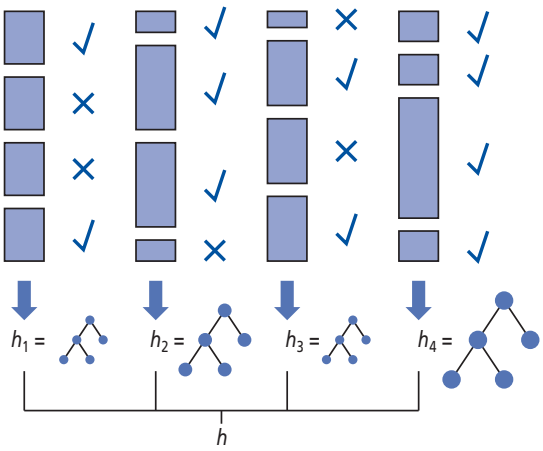


Abbildung 18.33: Die Arbeitsweise des Boosting-Algorithmus. Jedes schattierte Rechteck entspricht einem Beispiel; die Höhe des Rechtecks entspricht der Gewichtung. Die Haken und Kreuze geben an, ob das Beispiel von der aktuellen Hypothese korrekt klassifiziert wurde. Die Größe des Entscheidungsbaumes gibt die Gewichtung dieser Hypothese in der fertigen Gruppe an.

Tipp

```
function ADABOOST(examples, L, K) returns eine gewichtete Mehrheitshypothese
   inputs: examples, Menge von N beschrifteten Beispielen (x1,y1), ..., (xN, yN)
           L, ein Lernalgorithmus
           K, die Anzahl der Hypothesen in der Gruppe
   local variables: w, ein Vektor mit N Beispielgewichtungen, anfänglich 1/N
                    h, ein Vektor mit K Hypothesen
                    z, ein Vektor mit K Hypothesengewichtungen

   for k = 1 to K do
      h[k] ← L(examples, w)
      error ← 0
      for j = 1 to N do
         if h[k](xj) ≠ yj then error ← error + w[j]
      for j = 1 to N do
         if h[k](xj) = yj then w[j] ← w[j] · error/(1 - error)
      w ← NORMALIZE(w)
      z[k] ← log(1 - error)/error
   return WEIGHTED-MAJORITY(h, z)
```

Abbildung 18.34: Die ADABOOST-Variante der Boosting-Methode für Gruppenlernen. Der Algorithmus kombiniert Hypothesen, indem er nacheinander die Trainingsbeispiele neu gewichtet. Die Funktion WEIGHTED-MAJORITY erzeugt eine Hypothese, die den Ausgabewert mit dem höchsten Abstimmungsergebnis von den Hypothesen in **h** zurückgibt, wobei die Abstimmung durch **z** gewichtet ist.

Jetzt überprüfen wir, wie gut das Boosting für unsere Restaurantdaten geeignet ist. Wir wählen als Original-Hypothesenraum die Klasse der **Entscheidungsstümpfe**, wobei es sich um Entscheidungsbäume mit nur einem Test an der Wurzel handelt. Die untere Kurve in ▶ Abbildung 18.35(a) zeigt, dass Entscheidungsstümpfe ohne Boosting nicht besonders effektiv für diese Datenmenge sind und eine Vorhersageleistung von nur 81% für 100 Trainingsbeispiele erbringen. Bei angewandtem Boosting (mit $K = 5$) ist die Leistung besser, sie erreicht 93% nach 100 Beispielen.

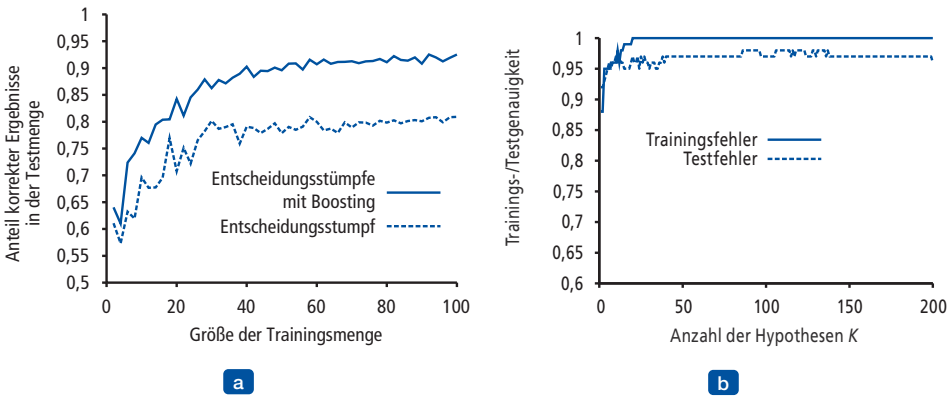


Abbildung 18.35: (a) Graph, der die Leistung von Entscheidungsstümpfen mit Boosting mit $K = 5$ im Vergleich zu Entscheidungsstümpfen für die Restaurantdaten zeigt. (b) Der Anteil korrekter Ergebnisse in der Trainingsmenge und die Testmenge als Funktion von K, der Anzahl der Hypothesen in der Gruppe. Beachten Sie, dass die Testmengengenauigkeit selbst dann noch leicht verbessert wird, nachdem die Trainingsgenauigkeit 1 erreicht ist, d.h. nachdem die Gruppe exakt mit den Daten übereinstimmt.

Wenn die Gruppengröße K zunimmt, passiert etwas sehr Interessantes. ▶ Abbildung 18.35(b) zeigt die Trainingsmengenleistung (bei 100 Beispielen) als Funktion von K. Beachten Sie, dass der Fehler gegen 0 geht, wenn K gleich 20 ist; d.h., eine gewichtete Mehrheitskombination von 20 Entscheidungsstümpfen ist ausreichend, um eine genaue Übereinstimmung mit den 100 Beispielen zu erzielen. Wenn der Gruppe mehr Stümpfe hinzugefügt werden, bleibt der Fehler bei 0. Der Graph zeigt außerdem, *dass die Testmengenleistung noch steigt, lange nachdem der Trainingsmengenfehler null erreicht hat*. Bei $K = 20$ hat die Testleistung einen Wert von 0,95 (oder einen Fehler von 0,05) und erst bei $K = 137$ steigt die Leistung auf 0,98, bevor sie allmählich auf 0,95 fällt.

Diese Feststellung, die relativ robust über Datenmengen und Hypothesenräume hinweg ist, war beim ersten Auftreten recht überraschend. Das Ockham-Rasiermesser weist uns an, die Hypothesen nicht komplexer als nötig zu machen, aber der Graph sagt uns, dass sich die Vorhersagen *verbessern*, wenn die Gruppenhypothese komplexer wird! Für diesen Sachverhalt wurden verschiedene Erklärungen angeboten. Eine Perspektive ist, dass beim Boosting das **Bayessche Lernen** angenähert wird (siehe *Kapitel 20*), wofür gezeigt werden kann, dass es sich um einen optimalen Lernalgorithmus handelt, und die Annäherung verbessert sich immer mehr, je mehr Hypothesen hinzugefügt werden. Eine weitere mögliche Erklärung ist, dass das Hinzufügen weiterer Hypothesen es erlaubt, dass die Gruppe *genauer* in ihrer Entscheidung zwischen positiven und negativen Beispielen sein kann, was hilfreich ist, wenn neue Beispiele klassifiziert werden müssen.

18.10.1 Online-Lernen

Sämtliche bisher in diesem Kapitel besprochenen Algorithmen haben sich auf die Annahme gestützt, dass die Daten unabhängig und identisch verteilt sind (u.i.v.-Annahme). Einerseits ist dies eine sinnvolle Annahme: Wenn die Zukunft keine Ähnlichkeit mit der Vergangenheit aufweist, wie können wir dann irgendetwas vorhersagen? Andererseits ist es eine zu starke Annahme: Es kommt selten vor, dass unsere Eingaben sämtliche Informationen erfasst haben, die die Zukunft wirklich unabhängig von der Vergangenheit machen.

Dieser Abschnitt untersucht, was zu tun ist, wenn die Daten nicht unabhängig und identisch verteilt sind, wenn sie sich im Lauf der Zeit ändern können. In diesem Fall spielt es eine Rolle, *wann* wir eine Vorhersage treffen, sodass wir die als **Online-Lernen** bezeichnete Sichtweise übernehmen: Ein Agent erhält eine Eingabe x_j aus der Natur, sagt den entsprechenden y_i voraus und erfährt dann die richtige Antwort. Der Prozess wird mit x_{j+1} wiederholt usw. Man könnte meinen, dass diese Aufgabe hoffnungslos ist – bei einer kontradiktorischen Natur können sämtliche Vorhersagen falsch sein. Es zeigt sich aber, dass wir einige Garantieannahmen treffen können.

Betrachten wir die Situation, in der unsere Eingabe aus Vorhersagen von einer Expertengruppe besteht. Zum Beispiel sagt eine Menge von K Experten jeden Tag voraus, ob der Aktienmarkt nach oben oder unten geht, und unsere Aufgabe besteht darin, diese Vorhersagen zu bündeln und unsere eigenen zu treffen. Zum Beispiel können wir verfolgen, wie gut jeder Experte abschneidet, und ihm Vertrauen proportional zu seiner letzten Leistung schenken. Dieser als **randomisiert gewichteter Mehrheitsalgorithmus** bezeichnete Algorithmus lässt sich formal wie folgt beschreiben:

1 Eine Menge von Gewichten $\{w_1, \ldots, w_K\}$ mit 1 initialisieren

2 Die Vorhersagen $\{\hat{y}_1, ..., \hat{y}_K\}$ von den Experten empfangen

3 Einen Experten k^* proportional zu seinem Gewicht zufällig auswählen: $P(k) = w_k / (\Sigma_{k'} w_{k'})$

4 Den Wert $\hat{y}_{k^*}$ vorhersagen

5 Die richtige Antwort y empfangen

6 Für jeden Experten k, derart dass $\hat{y}_k \neq y$, aktualisieren $w_k \leftarrow \beta w_k$

Hier ist β eine Zahl $0 < \beta < 1$, die aussagt, wie stark ein Experte für jeden Fehler zu bestrafen ist.

Den Erfolg dieses Algorithmus messen wir in Form von **Bedauern**. Wir definieren es als Anzahl der von uns zusätzlich gemachten Fehler im Vergleich zu dem Experten, der im Nachhinein die beste Vorhersage hatte. Es sei M^* die Anzahl der Fehler, die der beste Experte gemacht hat. Dann wird die Anzahl der Fehler M, die der randomisiert gewichtete Mehrheitsalgorithmus gemacht hat, durch

$$M < \frac{M^* \ln(1/\beta) + \ln K}{1 - \beta}$$

begrenzt.[15]

Diese Grenze gilt für jede Folge von Beispielen, selbst solche, die von Kontrahenten ausgewählt wurden, die auf den ungünstigsten Fall aus sind. Ein konkretes Beispiel soll dies verdeutlichen: Wenn es $K = 10$ Experten gibt und wir $\beta = 1/2$ wählen, ist unsere Anzahl der Fehler durch $1{,}39\, M^* + 4{,}6$ begrenzt, und bei $\beta = 3/4$ durch $1{,}15\, M^* + 9{,}2$. Im Allgemeinen sind wir für Änderungen auf lange Sicht zuständig, wenn β nahe an 1 liegt; ändert sich der beste Experte, greifen wir ihn über kurz oder lang heraus. Allerdings zahlen wir am Anfang eine Strafe, wenn wir zunächst allen Experten gleichermaßen vertrauen; eventuell akzeptieren wir zu lange den Rat eines schlechten

15 Siehe Blum (1996) für den Beweis.

Experten. Liegt β näher an 0, sind diese beiden Faktoren umgekehrt. Wir können β so wählen, dass wir uns auf lange Sicht asymptotisch an M^* annähern; dies ist das sogenannte **Lernen ohne Bedauern** (da die durchschnittliche Größe des Bedauerns pro Versuch gegen 0 geht, wenn die Anzahl der Versuche zunimmt).

Online-Lernen ist hilfreich, wenn sich die Daten zeitlich schnell ändern können. Es bietet sich auch für Anwendungen an, die eine große Datensammlung verarbeiten, die beständig wächst, selbst wenn die Änderungen allmählich erfolgen. So werden Sie beispielsweise bei einer Datenbank mit Millionen von Webbildern kein lineares Regressionsmodell für sämtliche Daten trainieren und für jedes hinzugefügte Bild das ganze Training von Grund auf neu durchführen. Hier ist ein Online-Algorithmus praktischer, der es zulässt, Bilder inkrementell hinzuzufügen. Für die meisten Lernalgorithmen, die auf der Minimierung des Verlustes basieren, gibt es eine Online-Version, die auf der Minimierung von Bedauern basiert. Positiv zu vermerken ist, dass viele dieser Online-Algorithmen garantierte Grenzen bei Bedauern realisieren.

Für manche Beobachter ist es überraschend, dass es derartige enge Grenzen gibt, wie gut wir im Vergleich zu einer Expertengruppe abschneiden können. Für andere liegt die eigentliche Überraschung darin, dass bei einer Versammlung von Expertengruppen – die Aktienkurse, Sportergebnisse oder politische Entscheidungen vorhersagen – die Zuschauer den hochtrabenden Reden begierig folgen, aber gänzlich abgeneigt sind, die Fehlerquoten zu quantifizieren.

18.11 Maschinelles Lernen in der Praxis

Wir haben eine breite Palette von Techniken für das maschinelle Lernen vorgestellt und jeweils mit einfachen Lernaufgaben veranschaulicht. Dieser Abschnitt betrachtet nun zwei Aspekte aus der Praxis des maschinellen Lernens. Beim ersten sind Algorithmen gesucht, die handschriftliche Ziffernerkennung lernen und gewissermaßen den letzten Tropfen von Vorhersageleistung aus ihnen herausquetschen können. Der zweite betrifft alles andere – stellt aber heraus, dass das Erfassen, Bereinigen und Darstellen der Daten zumindest genauso wichtig sein kann wie der Algorithmenentwurf.

18.11.1 Fallstudie: handschriftliche Ziffernerkennung

Die Erkennung handschriftlicher Ziffern ist ein Problem in vielen Anwendungen, wie beispielsweise bei der automatischen Sortierung nach Postleitzahlen, dem automatisierten Einlesen von Schecks und Steuerrückzahlungen oder bei der Dateneingabe in Handheld-Computer. In diesem Bereich wurden rasante Fortschritte gemacht, insbesondere aufgrund besserer Lernalgorithmen und der Verfügbarkeit besserer Trainingsmengen. Das **NIST** (United States National Institute of Science and Technologie) hat eine Datenbank von 60.000 ausgewerteten Ziffernproben mit je $20 \times 20 = 400$ Pixeln und 8-Bit-Graustufenwerten archiviert. Es ist zu einer der Standard-Benchmarks für den Vergleich neuer Lernalgorithmen geworden. ▶ Abbildung 18.36 zeigt einige Beispielziffern.

Abbildung 18.36: Beispiele aus der NIST-Datenbank für handschriftliche Ziffern. Obere Reihe: Beispiele der Ziffern 0 – 9, die einfach zu erkennen sind. Untere Reihe: schwierigere Beispiele derselben Ziffern.

Es wurden viele verschiedene Lernansätze ausprobiert. Einer der ersten und wahrscheinlich der einfachste ist der **3-nächste-Nachbarn**-Klassifizierer, der außerdem den Vorteil hat, keine Trainingszeit zu brauchen. Als speicherbasierter Algorithmus muss er jedoch alle 60.000 Bilder speichern und läuft dementsprechend langsam. Er hat eine Testfehlerrate von 2,4% erreicht.

Für dieses Problem wurde ein **neuronales Netz mit einer einzigen verborgenen Schicht** entwickelt, mit 400 Eingabeeinheiten (eine pro Pixel) und zehn Ausgabeeinheiten (eine pro Klasse). Durch Kreuzvalidierung wurde festgestellt, dass die beste Leistung mit etwa 300 verborgenen Einheiten erzielt wurde. Bei vollständiger Verknüpfung zwischen den Schichten gab es insgesamt 123.300 Gewichte. Dieses Netz erzielte eine Fehlerrate von 1,6%.

Es wurde eine Folge **spezialisierter neuronaler Netze** namens LeNet entworfen, die die Struktur des Problems nutzen sollten – dass die Eingabe aus Pixeln in einem zweidimensionalen Array besteht und dass kleine Änderungen in der Position oder in der Neigung eines Bildes keine Bedeutung haben. Jedes Netz hatte eine Eingabeschicht von 32×32 Einheiten, auf der 20×20 Pixel zentriert waren, sodass jede Eingabeeinheit mit einer lokalen Nachbarschaft von Pixeln präsentiert wurde. Es folgten drei Schichten verborgener Einheiten. Jede Schicht bestand aus mehreren Ebenen von $n \times n$-Arrays, wobei n kleiner als die vorhergehende Schicht war, sodass das Netz die Eingaberate reduzierte und die Gewichte jeder Einheit in einer Ebene identisch sein mussten, sodass die Ebene als Merkmalsdetektor fungierte: Sie konnte ein Merkmal auswählen, wie beispielsweise eine lange vertikale Linie oder einen kurzen halbkreisförmigen Bogen. Die Ausgabeschicht hatte zehn Einheiten. Viele Versionen dieser Architektur wurden ausprobiert; ein repräsentatives Modell hatte verborgene Schichten mit 768, 192 und 30 Einheiten. Die Trainingsmenge wurde verstärkt, indem geeignete Transformationen auf die tatsächlichen Eingaben vorgenommen wurden: Verschiebungen, leichte Drehungen oder Skalierungen der Bilder. (Natürlich müssen diese Transformationen klein sein, sonst würde eine 6 in eine 9 umgewandelt!) Die beste Fehlerrate, die LeNet erzielen konnte, betrug 0,9%.

Ein **boosted neuronales Netz** kombinierte drei Kopien der LeNet-Architektur, wobei die zweite für eine Mischung aus Mustern trainiert wurde, für die die erste zu 50% falsch lag, und die dritte mit Mustern, für die sich die beiden ersten nicht einig waren. Beim Testen stimmten die drei Netze entsprechend der Mehrheitsregel ab. Die Testfehlerrate betrug 0,7%.

Eine **Support-Vector-Maschine** (siehe *Abschnitt 18.9*) mit 25.000 Support-Vektoren erzielte eine Fehlerrate von 1,1%. Das ist bemerkenswert, denn die SVM-Technik bedingt wie der einfache Nächste-Nachbarn-Ansatz fast keine Überlegungen oder iterierenden Experimente auf Seiten des Entwicklers und kommt der Leistung von LeNet dennoch sehr nahe, für das Jahre der Entwicklung aufgewendet wurden. Tatsächlich nutzt

die Support-Vector-Maschine die Struktur des Problems nicht und würde auch dann eine gute Leistung bringen, wenn die Pixel in permutierter Reihenfolge gezeigt werden.

Eine **virtuelle Support-Vector-Maschine** beginnt mit einer normalen SVM und verbessert sie dann unter Verwendung einer Technik, die darauf ausgelegt ist, die Problemstruktur zu nutzen. Anstatt Produkte aller Pixelpaare zuzulassen, konzentriert sich dieser Ansatz auf Kernel, die durch Paare benachbarter Pixel gebildet werden. Außerdem erweitert er die Trainingsmenge mit Transformationen der Beispiele, so wie LeNet. Eine virtuelle SVM erzielte die bis heute beste Fehlerrate von 0,56%.

Der **Umrissvergleich** ist eine Technik aus der Computervision, die eingesetzt wird, um zugehörige Teile zweier verschiedener Bilder von Objekten zuzuordnen (Belongie et al., 2002). Die Idee dabei ist, in jedem der zwei Bilder eine Menge von Punkten auszuwählen und dann für jeden Punkt im ersten Bild zu berechnen, welcher Punkt im zweiten Bild ihm entspricht. Aus dieser Zuordnung können wir eine Transformation zwischen den Bildern berechnen. Die Transformationen bieten uns ein Maß für die Distanz zwischen den Bildern. Diese Distanzmessung ist besser motiviert als lediglich zu zählen, wie viele Pixel differieren, und es zeigt sich, dass der 3-nächste-Nachbarn-Algorithmus unter Verwendung dieser Distanzmessung eine ausgezeichnete Leistung erbringt. Bei einem Training für nur 20.000 von 60.000 Ziffern und die Verwendung von 100 Stichprobenpunkten pro Bild (erzeugt mit einem Canny-Kantendetektor) erzielte ein Umrissvergleich 0,63% Testfehler.

Menschen haben für dieses Problem eine Fehlerrate von schätzungsweise 0,2%. Diese Zahl ist nicht wirklich zuverlässig, weil mit Menschen keine so umfassenden Tests durchgeführt wurden wie mit Lernalgorithmen von Maschinen. Für eine vergleichbare Datenmenge mit Ziffern vom United States Postal Service hat man für den Menschen eine Fehlerrate von 2,5% ermittelt.

Die folgende Abbildung zeigt einen Überblick über Fehlerraten, Laufzeitleistung, Speicheranforderungen und Trainingszeitaufwand für die sieben beschriebenen Algorithmen. Außerdem wird ein zusätzliches Maß eingeführt, der Prozentsatz der Ziffern, die zurückgewiesen werden müssen, um 0,5% Fehler zu erreichen. Darf die SVM beispielsweise 1,8% der Eingaben zurückweisen – d.h. sie jemand anderem übergeben, der eine endgültige Beurteilung durchführen soll –, dann wird die Fehlerrate der restlichen 98,2% der Eingaben von 1,1% auf 0,5% reduziert.

Die folgende Tabelle zeigt einen Überblick über die Fehlerrate und einige der anderen Eigenschaften der sieben beschriebenen Techniken.

	3 NN	300 Verborgen	LeNet	Boosted LeNet	SVM	Virtuelle SVM	Umriss-vergleich
Fehlerrate (Prozent)	2,4	1,6	0,9	0,7	1,1	0,56	0,63
Laufzeit (ms/Ziffer)	1000	10	30	50	2000	200	
Speicherbedarf (MB)	12	0,49	0,012	0,21	11		
Trainingszeit-aufwand (Tage)	0	7	14	30	10		
% Rückweisungen, um 0,5% Fehler zu erzielen	8,1	3,2	1,8	0,5	1,8		

18.11.2 Fallstudie: Wortsinn und Hauspreise

In einem Lehrbuch müssen wir mit einfachen „Spielzeugdaten“ hantieren, um die Konzepte vermitteln zu können: eine kleine Datenmenge, normalerweise in zwei Dimensionen. Doch in praktischen Anwendungen des maschinellen Lernens ist die Datenmenge gewöhnlich groß, mehrdimensional und unsauber. Die Daten werden dem Analysten nicht in einer aufbereiteten Menge von (x, y)-Werten übergeben. Vielmehr muss er vor Ort gehen und sich die richtigen Daten beschaffen. Es ist eine Aufgabe zu realisieren, wobei das technische Problem zum größten Teil darin besteht, zu entscheiden, welche Daten für die gestellte Aufgabe notwendig sind. Der geringere Teil ist die Auswahl und die Implementierung einer geeigneten Methode für das maschinelle Lernen, um die Daten zu verarbeiten. ▶ Abbildung 18.37 zeigt ein typisches Beispiel aus der Praxis. Verglichen werden dabei fünf Lernalgorithmen für die Aufgabe der Wortsinn-Klassifizierung (für einen Satz wie zum Beispiel „Biegen Sie an der Bank links ab.“ ist das Wort „Bank“ als „Gebäude“ oder als „Möbelstück“ zu klassifizieren). Nun hat sich die Forschung auf dem Gebiet des maschinellen Lernens hauptsächlich auf die vertikale Richtung konzentriert: Kann ich einen neuen Lernalgorithmus erfinden, der für eine Standardtrainingsmenge von 1 Millionen Wörtern besser abschneidet als bereits veröffentlichte Algorithmen? Doch wie aus dem Diagramm hervorgeht, ist für Verbesserungen mehr Raum in der horizontalen Richtung: Anstatt einen neuen Algorithmus zu erfinden, muss ich lediglich 10 Millionen Wörter als Trainingsdaten sammeln; sogar der *schlechteste* Algorithmus leistet bei 10 Millionen Wörtern mehr als der beste Algorithmus bei 1 Million. Wenn wir noch mehr Daten zusammentragen, steigen die Kurven weiter an, wobei die Unterschiede zwischen den Algorithmen immer kleiner werden.

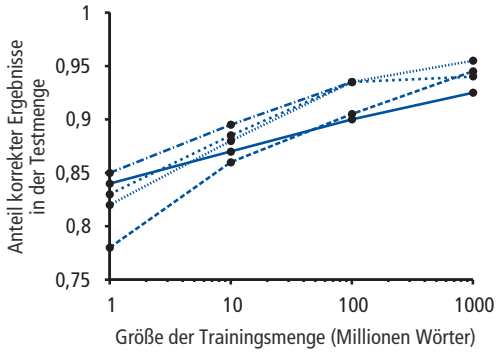


Abbildung 18.37: Lernkurven von fünf Lernalgorithmen für eine gemeinsame Aufgabe. Beachten Sie, dass es in horizontaler Richtung (mehr Trainingsdaten) scheinbar mehr Raum für Verbesserungen gibt als in vertikaler Richtung (unterschiedliche Algorithmen des maschinellen Lernens). Nach Banko und Brill (2001).

Sehen wir uns ein anderes Problem an: die Aufgabe, den wahren Wert von Häusern zu schätzen, die zum Verkauf stehen. In Abbildung 18.13 haben wir eine Miniversion dieses Problems gezeigt, das Hausgröße und Angebotspreis per linearer Regression korreliert hat. Sicherlich haben Sie die vielen Beschränkungen dieses Modells bemerkt. Erstens misst es das Falsche: Wir möchten den Verkaufspreis eines Hauses abschätzen und nicht den Angebotspreis. Um diese Aufgabe zu lösen, brauchen wir Daten zu tatsächlichen Verkäufen. Doch das heißt nun nicht, dass wir die Daten über den Angebotspreis wegwerfen sollten – wir können sie als eines der Eingabemerkmale verwenden. Neben der Hausgröße brauchen wir weitere Informationen: die Anzahl der Zimmer, Schlafzim-

mer und Bäder; ob Küche und Badezimmer kürzlich renoviert wurden; das Alter des Hauses; Angaben zu Grundstück und Nachbarschaft. Doch wie definieren wir Nachbarschaft? Nach der Postleitzahl? Wie sieht es aus, wenn ein Teil des PLZ-Gebietes auf der „falschen" Seite der Schnellstraße oder der Bahngleise liegt und der andere Teil lukrativ ist? Wie verhält es sich mit dem Schulbezirk? Sollte der *Name* des Schulbezirkes ein Feature sein oder die *durchschnittlichen Testbewertungen*? Wir müssen aber nicht nur entscheiden, welche Features einzuschließen sind, sondern uns auch um fehlende Daten kümmern. Verschiedene Gebiete haben unterschiedliche Konventionen, welche Daten gemeldet werden, und in Einzelfällen werden immer irgendwelche Daten fehlen. Stehen Ihnen die gewünschten Daten nicht zur Verfügung, können Sie vielleicht eine Site in einem sozialen Netzwerk einrichten, um die Anwohner zu ermutigen, Daten mitzuteilen und zu korrigieren. Letztlich ist dieser Entscheidungsprozess, welche Features wie zu verwenden sind, genauso wichtig wie die Wahl zwischen linearer Regression, Entscheidungsbäumen oder einer anderen Form des Lernens.

In diesem Sinne muss man eine Methode (oder mehrere Methoden) für ein Problem herausgreifen. Es gibt keinen garantierten Weg, wie man zur besten Methode gelangt, doch Sie können sich grob an einigen Richtlinien orientieren. Entscheidungsbäume sind gut geeignet, wenn es jede Menge diskrete Merkmale gibt und viele davon Ihrer Meinung nach möglicherweise irrelevant sind. Parameterfreie Methoden kommen infrage, wenn Sie viele Daten und kein A-priori-Wissen haben und wenn Sie sich nicht allzu viele Gedanken über die Auswahl genau der richtigen Features machen wollen (solange es weniger als etwa 20 sind). Allerdings liefern parameterfreie Methoden üblicherweise eine Funktion *h*, deren Ausführung recht teuer ist. Support-Vector-Maschinen gelten oftmals als beste Methode, die man zuerst ausprobieren sollte, sofern die Datenmenge nicht zu umfangreich ist.

Bibliografische und historische Hinweise

Kapitel 1 hat die Geschichte philosophischer Betrachtungen zum induktiven Lernen aufgezeigt. William von Ockham[16] (1280–1349), der einflussreichste Philosoph seines Jahrhunderts und ein großer Leistungsträger der mittelalterlichen Epistemologie, Logik und Metaphysik, wird eine Aussage zugeschrieben, die auch als „Ockhams Rasiermesser" bezeichnet wird – in Latein: *Entia non sunt multiplicanda praeter necessitatem*, zu Deutsch: „Einheiten brauchen nicht mehr als nötig vervielfältigt zu werden." Leider ist dieser lobenswerte Ratschlag in keinem seiner Werke in genau diesem Wortlaut aufgeschrieben (auch wenn er sagte „Pluralitas non est ponenda sine necessitate" oder „Vielzahl ist nicht ohne Notwendigkeit zu setzen"). Eine ähnliche Empfindung wurde 350 v. Chr. von Aristoteles in *Physik* Buch 1, Kapitel VI ausgedrückt: „... und es ist besser, aus begrenzten zu erklären ...".

EPAM, der „Elementary Perceiver And Memorizer" (Feigenbaum, 1961), verwendete als eines der ersten Systeme Entscheidungsbäume und war ein Simulationsmodell menschlichen Konzeptlernens. ID3 (Quinlan, 1979) ergänzte das wichtige Konzept, das Attribut mit maximaler Entropie auszuwählen; es bildet die Grundlage für den Entscheidungsbaumalgorithmus in diesem Kapitel. Die Informationstheorie wurde von Claude Shannon entwickelt, um die Kommunikationsstudie zu unterstützen (Shannon und Weaver, 1949). (Shannon arbeitete auch mit an einem der ersten Beispiele des Maschinenlernens,

16 Der Name wird oftmals falsch als „Occam" geschrieben, was vermutlich auf die französische Version „Guillaume d'Occam" zurückgeht.

einer mechanischen Maus namens Theseus, die lernte, durch Trial&Error durch ein Labyrinth zu gehen.) Die $\chi 2$-Methode des Baumkürzens wurde von Quinlan (1986) beschrieben. C4.5, ein Entscheidungsbaumpaket für den Einsatz in der Industrie, ist in Quinlan (1993) beschrieben. Eine unabhängige Tradition des Entscheidungsbaumlernens finden wir in der Literatur zur Statistik. *Classification and Regression Trees* (Breiman et al., 1984), auch als das „CART-Buch" bekannt, ist das wichtigste Nachschlagewerk.

Kreuzvalidierung wurde zuerst von Larson (1931) eingeführt und zwar in einer Form, die der von Stone (1974) und Golub et al. (1979) gezeigten nahekommt. Die Regularisierungsprozedur geht auf Tikhonov (1963) zurück. Guyon und Elisseeff (2003) gehen in einer Zeitschriftenausgabe auf das Problem der Merkmalsauswahl ein. Banko und Brill (2001) sowie Halevy et al. (2009) diskutieren die Vorteile bei Verwendung umfangreicher Daten. Der Sprachforscher Robert Mercer stellte 1985 fest: „There is no data like more data." (etwa: „Es gibt keine besseren Daten als noch mehr Daten.") Lyman und Varian (2003) schätzen, dass in 2002 ungefähr 5 Exabyte (5×10^{18} Byte) Daten produziert wurden und dass sich die Produktionsrate alle 3 Jahre verdoppelt.

Die theoretische Analyse von Lernalgorithmen begann mit der Arbeit von Gold (1967) zur **Identifizierung der Grenzen**. Dieser Ansatz wurde zum Teil durch Modelle wissenschaftlicher Entdeckungen der Wissenschaftsphilosophie (Popper, 1962) angeregt, jedoch hauptsächlich auf das Problem angewendet, Grammatiken aus Beispielsätzen zu lernen (Osherson et al., 1986).

Während sich der Ansatz, die Grenze zu identifizieren, auf eine etwaige Konvergenz konzentriert, versucht die Studie der **Kolmogorov-Komplexität** oder **algorithmischen Komplexität**, die unabhängig von Solomonoff (1964, 2009) und Kolmogorov (1965) entwickelt wurde, eine formale Definition für das in Ockhams Rasiermesser verwendete Konzept der Einfachheit bereitzustellen. Um das Problem zu umgehen, dass die Einfachheit davon abhängig ist, wie die Information dargestellt wird, wird vorgeschlagen, dass die Einfachheit durch die Länge des kürzesten Programms für eine allgemeine Turing-Maschine gemessen wird, die die beobachteten Daten korrekt reproduziert. Obwohl es viele universelle Turing-Maschinen gibt und damit viele mögliche „kürzeste" Programme, unterscheiden sich diese Programme in ihrer Länge um höchstens eine Konstante, die unabhängig von der Datenmenge ist. Diese wunderbare Einsicht, die im Wesentlichen zeigt, dass *jeder* anfängliche Fehler in der Repräsentation irgendwann von den eigentlichen Daten überlagert wird, wird nur durch die Unentscheidbarkeit der Längenberechnung des kürzesten Programms getrübt. Stattdessen können annähernde Messungen, wie etwa die **minimale Beschreibungslänge** (MDL, Minimum Description Length) (Rissanen, 1984, 2007), verwendet werden, die in der Praxis ausgezeichnete Ergebnisse erzielt haben. Der Text von Li und Vitanyi (1993) ist das beste Nachschlagewerk für die Kolmogorov-Komplexität.

Die Computer-Lerntheorie – d.h. die Theorie des PAC-Lernens – wurde von Leslie Valiant (1984) eingeführt. Die Arbeit von Valiant betonte die Bedeutung der programmtechnischen und Stichprobenkomplexität. Zusammen mit Michael Kearns (1990) zeigte Valiant, dass das PAC-Lernen für mehrere Konzeptklassen nicht handhabbar ist, selbst wenn in den Beispielen ausreichend viel Information zur Verfügung steht. Einige positive Ergebnisse erhielt man für Klassen wie etwa Entscheidungslisten (Rivest, 1987).

Eine unabhängige Tradition der Stichproben-Komplexitätsanalyse gab es in der Statistik, beginnend mit der Arbeit zur **einheitlichen Konvergenztheorie** (Vapnik und Chervonenkis, 1971). Die sogenannte **VC-Dimension** stellt ein Maß bereit, das in etwa analog zu dem Maß $\ln|\mathcal{H}|$ aus der PAC-Analyse ist, aber etwas allgemeiner zu sein scheint. Die VC-

Dimension kann auf stetige Funktionsklassen angewendet werden, für die die PAC-Standardanalyse nicht geeignet ist. Die PAC-Lerntheorie und die VC-Theorie wurden zuerst von den „vier Deutschen" miteinander in Verbindung gebracht (von denen in Wirklichkeit keiner aus Deutschland stammt): Blumer, Ehrenfeucht, Haussler und Warmuth (1989).

Lineare Regression mit quadratischem Fehlerverlust geht auf Legendre (1805) und Gauß (1809) zurück, die beide an der Vorhersage von Planetenbahnen um die Sonne arbeiteten. Die moderne Verwendung der multivariaten Regression für Maschinenlernen wird in Texten wie zum Beispiel Bishop (2007) behandelt. Ng (2004) analysierte die Unterschiede zwischen L_1- und L_2-Regularisierung.

Der Begriff **logistische Funktion** stammt vom Statistiker Francois Verhulst (1804–1849), der die Kurve verwendete, um Populationswachstum mit begrenzten Ressourcen zu modellieren. Dieses Modell ist realistischer als das unbeschränkte geometrische Wachstum, das Thomas Malthus vorgeschlagen hatte. Verhulst nannte sie die *courbe logistique* (logistische Kurve) aufgrund ihrer Beziehung zur logarithmischen Kurve. Der Terminus **Regression** geht auf den Statistiker Francis Galton, einen Cousin von Charles Darwin, aus dem 19. Jahrhundert zurück, der die Gebiete der Meteorologie, Fingerabdruckanalyse und statistischen Korrelation begründete und den Begriff im Sinne von *Regression zur Mitte* verwendete. Der Ausdruck **Fluch der Dimensionalität** kommt von Richard Bellman (1961).

Logistische Regression lässt sich mit Gradientenabstieg oder mit der Newton-Raphson-Methode (Newton, 1671; Raphson, 1690) lösen. Für hochdimensionale Probleme wird manchmal eine Variante der Newton-Methode namens L-BFGS verwendet, wobei das L für „Limited Memory" (begrenzter Speicher) steht. Man vermeidet dabei, die vollständigen Matrizen auf einmal zu erzeugen, und legt stattdessen Teile von ihnen temporär an. BFGS steht für die Initialen der Autoren (Byrd et al., 1995).

Die Nächste-Nachbarn-Modelle gehen mindestens bis auf Fix und Hodges (1951) zurück und sind seitdem zu einem Standardwerkzeug in der Statistik und Mustererkennung geworden. In der KI wurden sie von Stanfill und Waltz (1986) bekannt gemacht, die Methoden zur Anpassung der Distanzmetrik an die Daten untersuchten. Hastie und Tibshirani (1996) entwickelten ein Verfahren, um die Metrik zu jedem Punkt im Raum abhängig von der Verteilung der Daten um diesen Punkt zu lokalisieren. Gionis et al. (1999) führten ortsabhängiges Hashing (Localized Sensitive Hashing, LSH) ein, das das Abrufen von ähnlichen Objekten in hochdimensionalen Räumen – speziell in der Computervision – revolutioniert hat. Andoni und Indyk (2006) bieten einen neueren Überblick über LSH und verwandte Methoden.

Die Ideen hinter Kernel-Maschinen gehen auf Aizerman et al. (1964) zurück (der auch den Kernel-Trick einführte), doch die umfassende Entwicklung der Theorie stammt von Vapnik und seinen Kollegen (Boser et al., 1992). Praktische Bedeutung erlangten Support-Vector-Machinen (SVMs) mit der Einführung der Soft-Margin-Klassifizierer für die Verarbeitung verrauschter Daten in einem Artikel, der 2008 den ACM Theory and Practice Award (Cortes und Vapnik, 1995) gewonnen hat, und mit dem Sequential Minimal Optimization (SMO)-Algorithmus für die effiziente Lösung von SVM-Problemen mithilfe quadratischer Programmierung (Platt, 1999). SVMs haben sich in vielen Bereichen etabliert und sind effektiv für Aufgaben einzusetzen wie zum Beispiel Textkategorisierung (Joachims, 2001), Computer-Genomik (Cristianini und Hahn, 2007) und Verarbeitung natürlicher Sprache wie zum Beispiel Handschrifterkennung von Ziffern von DeCoste und Schölkopf (2002). Als Teil dieses Prozesses wurden viele neue Kernel konzipiert, die mit Strings, Bäumen und anderen nichtnumerischen

Datentypen arbeiten. Eine verwandte Technik, die ebenfalls den Kernel-Trick nutzt, um implizit einen exponentiellen Merkmalsraum darzustellen, ist das abstimmende Perzeptron (Freund und Schapire, 1999; Collins und Duffy, 2002). Lehrbücher zu SVMs gibt es unter anderem von Cristianini und Shawe-Taylor (2000) sowie Schölkopf und Smola (2002). Eine verständlichere Erklärung erschien im *AI Magazine*-Artikel von Cristianini und Schölkopf (2002). Bengio und LeCun (2007) zeigen einige der Begrenzungen von SVMs und anderen lokalen, parameterfreien Methoden für Lernfunktionen, die eine globale Struktur haben, aber keine lokale Glätte aufweisen.

Gruppenlernen ist eine immer populärer werdende Technik, mit der sich die Leistung von Lernalgorithmen verbessern lässt. Als erste effektive Methode kombiniert **Bagging** (Breiman, 1996) Hypothesen, die aus mehreren **Bootstrap**-Datenmengen gelernt wurden, wobei jede eine Stichprobe der Originaldatenmenge darstellt. Die in diesem Kapitel beschriebene **Boosting**-Methode stammt aus einer theoretischen Arbeit von Schapire (1990). Der ADABOOST-Algorithmus wurde von Freund und Schapire (1996) entwickelt und theoretisch von Schapire (2003) analysiert. Friedman et al. (2000) erläutern Boosting aus der Perspektive des Statistikers. Online-Lernen wird in einem Überblick von Blum (1996) und in einem Buch von Cesa-Bianchi und Lugosi (2006) behandelt. Dredze et al. (2008) führen die Idee des Online-Lernens mit Konfidenzgewichten für die Klassifizierung ein: Außer für jeden Parameter ein Gewicht zu speichern, verwaltet dieser Algorithmus auch ein Konfidenzmaß, sodass ein neues Beispiel eine große Wirkung auf Merkmale haben kann, die vorher selten erschienen sind (und demzufolge eine geringe Konfidenz hatten), und eine kleine Wirkung bei häufigen Merkmalen zeigt, die bereits ausreichend bewertet wurden.

Die Literatur zu neuronalen Netzen ist zu umfangreich (derzeit rund 150.000 Veröffentlichungen), um sie im Detail zu behandeln. Cowan und Sharp (1988b, 1988a) geben einen Überblick über die frühe Geschichte, die mit der Arbeit von McCulloch und Pitts (1943) beginnt. (Wie bereits in *Kapitel 1* erwähnt, hat John McCarthy auf die Arbeit von Nicolas Rashevsky (1936, 1938) als frühestes mathematisches Modell für neuronales Lernen hingewiesen. Norbert Wiener, ein Pionier der Kybernetik und Steuerungstheorie (Wiener, 1948), hat mit McCulloch und Pitts zusammengearbeitet und eine Reihe von jungen Forschern inspiriert, unter anderem Marvin Minsky, der wohl als Erster 1951 ein funktionsfähiges neuronales Netz in Hardware entwickelt hat (siehe Minsky und Papert, 1988, Seiten ix–x). Turing (1948) verfasste einen Forschungsbericht mit dem Titel *Intelligent Machinery*, der mit dem Satz beginnt „Ich schlage vor, die Frage zu untersuchen, ob eine Maschine intelligentes Verhalten zeigen kann“. Er beschreibt dann eine rekurrente neuronale Netzarchitektur, die er als „B-artige, nicht organisierte Maschinen“ bezeichnet, und gibt einen Ansatz an, um sie zu trainieren. Leider blieb der Bericht bis 1969 unveröffentlicht und wurde auch in der jüngsten Zeit praktisch ignoriert.

Frank Rosenblatt (1957) erfand das moderne „Perzeptron“ und bewies das Perzeptron-Konvergenztheorem (1960), obwohl sich dies bereits durch rein mathematische Arbeiten außerhalb des Kontextes neuronaler Netze abgezeichnet hatte (Agmon, 1954; Motzkin und Schoenberg, 1954). Einige frühe Arbeiten wurden auch zu mehrschichtigen Netzen durchgeführt. Hierzu gehören **Gamba-Perzeptrons** (Gamba et al., 1961) und **Madalines** (Widrow, 1962). *Learning Machines* (Nilsson, 1965) behandelt einen großen Teil dieser frühen Arbeiten und weitere Themen. Das darauffolgende Abebben der frühen Anstrengungen zur Perzeptron-Forschung wurde durch das Buch *Perceptrons* (Minsky und Papert, 1969) beschleunigt (oder, wie die Autoren später behaupteten, lediglich erklärt), das das Fehlen der mathematischen Strenge auf diesem Gebiet beklagte. Das Buch hob hervor, dass ein-

schichtige Perzeptrons lediglich linear separierbare Konzepte darstellen können, und wies auf das Fehlen effektiver Lernalgorithmen für mehrschichtige Netze hin.

Die Artikel in Hinton und Anderson (1981), die auf einer Konferenz in San Diego 1979 basieren, lassen sich als Markstein für die Wiedergeburt des Konnektionismus betrachten. Die zweibändige Anthologie „PDP“ (Parallel Distributed Processing) von Rumelhart et al. (1986a) und ein kurzer Artikel in *Nature* (Rumelhart et al., 1986b) zogen viel Aufmerksamkeit auf sich – in der Tat multiplizierte sich die Anzahl der Veröffentlichungen zu „neuronalen Netzen“ zwischen 1980–1984 und 1990–1994 um einen Faktor von 200. Die Analyse der neuronalen Netze mithilfe der physikalischen Theorie der magnetischen Spin-Gläser (Amit et al., 1985) straffte die Verknüpfungen zwischen statistischer Mechanik und der Theorie neuronaler Netze – wobei nicht nur nützliche mathematische Einblicke sondern auch *Seriosität* gewonnen wurden. Die Backpropagation-Technik wurde zwar schon recht früh von Bryson und Ho (1969) erfunden, aber später mehrmals neu entdeckt (Werbos, 1974; Parker, 1985).

Die probabilistische Interpretation von neuronalen Netzen hat mehrere Quellen, einschließlich Baum und Wilczek (1988) sowie Bridle (1990). Jordan (1995) diskutiert die Rolle der Sigmoid-Funktion. Das Lernen Bayesscher Parameter für neuronale Netze wurde von MacKay (1992) vorgeschlagen und weiter von Neal (1996) untersucht. Die Kapazität neuronaler Netze für die Darstellung von Funktionen wurde von Cybenko (1988, 1989) untersucht. Er zeigte, dass zwei verborgene Schichten genügen, um eine beliebige Funktion darzustellen, und dass für eine beliebige *stetige* Funktion eine einzelne Schicht ausreicht. Die Methode der „optimalen Hirnschädigung“ zum Entfernen überflüssiger Verbindungen stammt von LeCun et al. (1989); Sietsma und Dow (1988) zeigen, wie nutzlose Einheiten zu entfernen sind. Der Tiling-Algorithmus, mit dem sich größere Strukturen erzeugen lassen, geht auf Mézard und Nadal (1989) zurück. LeCun et al. (1995) geben einen Überblick über eine Reihe von Algorithmen für die handschriftliche Ziffernerkennung. Seitdem wurden verbesserte Fehlerraten von Belongie et al. (2002) beim Umrissvergleich und von DeCoste and Schölkopf (2002) für virtuelle Support-Vektoren gemeldet. Die beste Testfehlerrate wurde bei Manuskripterstellung mit 0,39% von Ranzato et al. (2007) für ein konvolutionales neuronales Netz berichtet.

Die Komplexität beim Lernen neuronaler Netze wurde von Forschern im Rahmen der Computer-Lerntheorie untersucht. Frühe rechentechnische Ergebnisse wurden von Judd (1990) erhalten. Er zeigte, dass das allgemeine Problem, eine Menge von Gewichten zu finden, die mit einer Menge von Beispielen konsistent ist, selbst unter sehr restriktiven Annahmen NP-vollständig ist. Die ersten Ergebnisse zur Stichprobenkomplexität kommen von Baum and Haussler (1989), die zeigten, dass die Anzahl der für effektives Lernen erforderlichen Beispiele ungefähr mit $W \log W$ wächst, wobei W die Anzahl der Gewichte ist.[17] Seitdem ist hierzu eine ausgereifte Theorie entwickelt worden (Anthony und Bartlett, 1999). Dazu gehört auch das wichtige Ergebnis, dass die Darstellungskapazität eines Netzes sowohl von der Größe der Gewichte als auch von ihrer Anzahl abhängt – ein Ergebnis, das angesichts unserer Diskussion zur Regularisierung nicht überraschen dürfte.

Die bekannteste Art neuronaler Netze, die wir hier nicht behandelt haben, ist das **RBF** (**Radiale Basisfunktion**)-Netz. Eine radiale Basisfunktion kombiniert eine gewichtete Sammlung von Kerneln (in der Regel natürlich Gaußsche), um eine Funktion anzunähern. RBF-Netze lassen sich in zwei Phasen trainieren: Zunächst werden in einem nicht

17 Dies bestätigte etwa die „Regel von Onkel Bernie“. Diese Regel wurde nach Bernard Widrow benannt, der empfahl, etwa zehnmal so viele Beispiele wie Gewichte zu verwenden.

überwachten Clustering die Parameter der Gaußverteilung – die Mittelwerte und Varianzen – trainiert, wie es *Abschnitt 20.3.1* beschreibt. In der zweiten Phase bestimmt man die relativen Gewichte der Gaußschen Verteilungen. Dies ist ein System linearer Gleichungen, die wir direkt lösen können. Somit weisen beide Phasen des RBF-Trainings ihre Vorzüge auf: Die erste Phase ist nicht überwacht und erfordert somit keine bezeichneten Trainingsdaten, während die zweite Phase zwar überwacht ist, aber effizient arbeitet. Weitere Details finden Sie in Bishop (1995).

Rekurrente Netze, in denen die Einheiten zyklisch verknüpft sind, wurden in diesem Kapitel zwar kurz erwähnt, aber nicht ausführlich untersucht. **Hopfield-Netze** (Hopfield, 1982) sind die wohl am besten bekannte Klasse von rekurrenten Netzen. Sie verwenden *bidirektionale* Verbindungen mit *symmetrischen* Gewichten (d.h. $w_{i,j} = w_{j,i}$), wobei sämtliche Einheiten sowohl Eingabe- als auch Ausgabeeinheiten sind, die Aktualisierungsfunktion g die `sign`-Funktion ist und die Aktualisierungsniveaus nur ± 1 sein können. Ein Hopfield-Netz fungiert als **Assoziativspeicher**: Nachdem das Netz für einen Satz von Beispielen trainiert ist, bewirkt ein neuer Reiz, dass das Netz ein Aktualisierungsmuster einnimmt, das dem Beispiel in der Trainingsmenge entspricht, das dem neuen Reiz *am ähnlichsten ist*. Besteht die Trainingsmenge zum Beispiel aus einer Menge von Fotografien und der neue Stimulus ist ein kleiner Ausschnitt eines der Fotos, reproduzieren die Netzaktualisierungsniveaus das vollständige Foto aus dem kleinen übergebenen Teil. Bemerkenswert ist, dass die Originalfotos nicht separat im Netz gespeichert werden; jedes Gewicht ist eine partielle Kodierung aller Fotografien. Eines der interessantesten theoretischen Ergebnisse ist, dass Hopfield-Netze bis zu 0,138 N Trainingsbeispiele zuverlässig speichern können, wobei N die Anzahl der Einheiten im Netz ist.

Boltzmann-Machinen (Hinton und Sejnowski, 1983, 1986) verwenden ebenfalls symmetrische Gewichte, binden aber verborgene Einheiten ein. Außerdem verwenden sie eine *stochastische* Aktualisierungsfunktion, sodass die Wahrscheinlichkeit für die Ausgabe einer 1 eine Funktion der gesamten gewichteten Eingabe ist. Boltzmann-Maschinen machen demzufolge Zustandsübergänge durch, ähnlich einer Suche in der Art des Simulated Annealing (siehe *Kapitel 4*) für die Konfiguration, die die Trainingsmenge am besten annähert. Es zeigt sich, dass Boltzmann-Maschinen eng mit einem Spezialfall der Bayesschen Netze verwandt sind, die mit einem stochastischen Simulationsalgorithmus bewertet werden (siehe *Abschnitt 14.5*).

Für neuronale Netze sind Bishop (1995), Ripley (1996) und Haykin (2008) die führenden Texte.

Dayan und Abbott (2001) haben sich mit dem Gebiet der Neurowissenschaft (Computational Neuroscience) befasst. Der in diesem Kapitel vorgestellte Ansatz wurde durch die ausgezeichneten Schulungsunterlagen von David Cohn, Tom Mitchell, Andrew Moore und Andrew Ng inspiriert. Es gibt mehrere hochkarätige Lehrbücher zum Maschinenlernen (Mitchell, 1997; Bishop, 2007) und zu den eng verwandten und sich überschneidenden Gebieten Mustererkennung (Ripley, 1996; Duda et al., 2001), Statistik (Wasserman, 2004; Hastie et al., 2001), Data-Mining (Hand et al., 2001; Witten und Frank, 2005), Computer-Lerntheorie (Kearns und Vazirani, 1994; Vapnik, 1998) und Informationstheorie (Shannon und Weaver, 1949; MacKay, 2002; Cover und Thomas, 2006). Andere Bücher konzentrieren sich auf Implementierungen (Segaran, 2007; Marsland, 2009) und Vergleiche von Algorithmen (Michie et al., 1994). Die aktuelle Forschung auf dem Gebiet Maschinenlernen wird in den jährlichen Konferenzunterlagen der *International Conference on Machine Learning* (ICML) und der Konferenz zu *Neural Information Processing Systems* (NIPS), im *Machine Learning* und *Journal of Machine Learning Research* sowie in allgemeinen KI-Magazinen veröffentlicht.

Zusammenfassung

Dieses Kapitel hat sich mit induktivem Lernen deterministischer Funktionen aus Beispielen beschäftigt. Die wichtigsten Aspekte waren:

- Lernen kann viele Formen annehmen – abhängig von der Natur des Agenten, der zu verbessernden Komponenten und dem verfügbaren Feedback.
- Wenn das verfügbare Feedback die korrekte Antwort für Beispieleingaben bereitstellt, wird das Lernproblem auch als **überwachtes Lernen** bezeichnet. Die Aufgabe besteht darin, eine Funktion $y = h(x)$ zu lernen. Das Lernen einer Funktion mit diskreten Werten wird als **Klassifizierung** bezeichnet; das Lernen einer stetigen Funktion als **Regression**.
- Beim induktiven Lernen ist eine Hypothese zu finden, die mit den Beispielen übereinstimmt. Das **Ockham-Rasiermesser** gibt vor, die einfachste konsistente Hypothese zu wählen. Die Schwierigkeit dieser Aufgabe ist von der gewählten Repräsentation abhängig.
- **Entscheidungsbäume** können alle booleschen Funktionen darstellen. Die Heuristik des **Informationsgewinns** bietet eine effiziente Methode für die Ermittlung eines einfachen, konsistenten Entscheidungsbaumes.
- Die Leistung eines Lernalgorithmus wird anhand der **Lernkurve** gemessen, die die Vorhersagegenauigkeit für die **Testmenge** als Funktion der **Trainingsmenge**ngröße angibt.
- Wenn unter mehreren Modellen auszuwählen ist, lässt sich mithilfe der **Kreuzvalidierung** ein Modell ermitteln, das gut verallgemeinert.
- Manchmal sind nicht alle Fehler gleich. Eine Verlustfunktion sagt uns, wie schlecht jeder Fehler ist; das Ziel besteht dann darin, den Verlust über einer Validierungsmenge zu minimieren.
- Die **Computer-Lerntheorie** analysiert die Stichprobenkomplexität und die Rechenkomplexität des induktiven Lernens. Es gibt eine Abwägung zwischen der Ausdrucksstärke der Hypothesensprache und der Einfachheit des Lernens.
- **Lineare Regression** ist ein weithin verwendetes Modell. Die optimalen Parameter eines linearen Regressionsmodells lassen sich durch Gradientenabstiegssuche ermitteln oder genau berechnen.
- Ein linearer Klassifizierer mit abruptem Schwellenwert – auch als **Perzeptron** bezeichnet – kann durch eine einfache Gewichtsaktualisierungsregel trainiert werden, um sich Daten anzupassen, die **linear separierbar** sind. In anderen Fällen kann die Regel nicht konvergieren.
- **Logistische Regression** ersetzt die abrupte Schwelle des Perzeptrons durch eine weiche Schwelle, die durch eine logistische Funktion definiert ist. Der Gradientenabstieg funktioniert auch für verrauschte Daten, die nicht linear separierbar sind.
- **Neuronale Netze** repräsentieren komplexe nichtlineare Funktionen mit einem Netz aus Einheiten mit linearen Schwellenwerten. Mehrschichtige neuronale Feedforward-Netze sind in der Lage, jede beliebige Funktion darzustellen, sofern genügend Einheiten vorhanden sind. Der **Backpropagation**-Algorithmus (Fehlerrückführung) implementiert einen Gradientenabstieg im Parameterraum, um den Ausgabefehler zu minimieren.
- Parameterfreie Modelle verwenden alle Daten, um jede Vorhersage zu treffen, anstatt zu versuchen, die Daten zuerst mit wenigen Parametern zu summieren. Hierzu gehören **nächste Nachbarn** und **lokal gewichtete Regression**.
- **Support-Vector-Maschinen** finden lineare Separatoren mit der **Maximum-Margin**-Methode, um die Generalisierungsleistung des Klassifizierers zu verbessern. **Kernel-Methoden** transformieren implizit die Eingabedaten in einen höherdimensionalen Raum, in dem ein linearer Separator existieren kann, selbst wenn die ursprünglichen Daten nicht separierbar sind.
- Gruppenmethoden wie etwa das **Boosting** bieten häufig eine bessere Leistung als Einzelmethoden. Im **Online-Lernen** können wir die Meinungen von Experten zusammenfassen, um der Leistung des besten Experten beliebig nahe zu kommen, selbst wenn sich die Verteilung der Daten ständig verschiebt.

Übungen zu Kapitel 18

Lösungshinweise

1 Betrachten Sie das Problem eines Kindes, sprechen und eine Sprache verstehen zu lernen. Erklären Sie, wie sich dieser Prozess in das allgemeine Lernmodell einfügt. Beschreiben Sie die Wahrnehmungen und Aktionen des Kindes sowie die Arten des Lernens, die das Kind zu absolvieren hat. Beschreiben Sie die Teilfunktionen, die das Kind zu lernen versucht, in Form von Ein- und Ausgaben und verfügbaren Beispieldaten.

2 Wiederholen Sie Übung 18.1 für das Problem, Tennisspielen zu lernen (oder irgendeinen anderen Sport, den Sie beherrschen). Handelt es sich dabei um überwachtes Lernen oder um verstärktes Lernen?

3 Zeichnen Sie einen Entscheidungsbaum für das Problem, zu entscheiden, ob an einer Kreuzung angefahren werden soll, wenn die Ampel gerade auf Grün geschaltet hat.

4 Wir testen entlang eines Pfades in einem Entscheidungsbaum nie dasselbe Attribut zweimal. Warum nicht?

5 Angenommen, wir erzeugen eine Trainingsmenge aus einem Entscheidungsbaum und wenden dann das Entscheidungsbaumlernen auf diese Trainingsmenge an. Gibt der Lernalgorithmus irgendwann den korrekten Baum zurück, wenn die Trainingsmenge gegen unendlich geht? Warum oder warum nicht?

6 Beim rekursiven Aufbau von Entscheidungsbäumen kommt es manchmal vor, dass an einem Blattknoten eine gemischte Menge positiver und negativer Beispiele übrig bleibt, selbst wenn alle Attribute verwendet wurden. Es sei angenommen, wir haben p positive und n negative Beispiele.

a. Zeigen Sie, dass die von DECISION-TREE-LEARNING verwendete Lösung, die die Mehrheitsklassifizierung einsetzt, den absoluten Fehler über die Beispielmenge am Blattknoten minimiert.

b. Zeigen Sie, dass die **Klassenwahrscheinlichkeit** $p/(p+n)$ die Summe der Quadratfehler minimiert.

7 Angenommen, ein Attribut teilt die Menge der Beispiele E in Teilmengen E_k und jede Teilmenge umfasst p_k positive und n_k negative Beispiele. Zeigen Sie, dass das Attribut einen streng positiven Informationsgewinn hat, sofern das Verhältnis $p_k/(p_k + n_k)$ für alle k gleich ist.

8 Betrachten Sie die folgende Datenmenge, die aus drei binären Eingabeattributen (A_1, A_2 und A_3) und einer binären Ausgabe y besteht:

Beispiel	A_1	A_2	A_3	Ausgabe y
x_1	1	0	0	0
x_2	1	0	1	0
x_3	0	1	0	0
x_4	1	1	1	1
x_5	1	1	0	1

Verwenden Sie den Algorithmus in Abbildung 18.5, um für diese Daten einen Entscheidungsbaum zu lernen. Zeigen Sie, wie das Attribut berechnet wird, das für die Teilung an jedem Knoten zuständig ist.

9 Konstruieren Sie eine Datenmenge (Menge von Beispielen mit Attributen und Klassifikationen), die dazu führt, dass der Algorithmus zum Entscheidungsbaumlernen einen Baum findet, der nicht die minimale Größe hat. Zeigen Sie den vom Algorithmus konstruierten Baum und den Baum mit minimaler Größe, den Sie manuell generieren können.

10 Diese Übung betrachtet die χ^2-Kürzung von Entscheidungsbäumen (*Abschnitt 18.3.5*).

a. Erstellen Sie eine Datenmenge mit zwei Eingabeattributen, sodass der Informationsgewinn an der Wurzel des Baumes für beide Attribute null ist, es aber einen Entscheidungsbaum der Tiefe 2 gibt, der mit allen Daten konsistent ist. Was würde χ^2-Kürzung für diese Datenmenge tun, wenn sie von unten nach oben angewandt wird? Wie verhält es sich, wenn sie von oben nach unten angewandt wird?

b. Ändern Sie den DECISION-TREE-LEARNING-Algorithmus so ab, dass er die χ^2-Kürzung berücksichtigt. Bei Bedarf finden Sie detaillierte Informationen dazu in Quinlan (1986) oder Kearns und Mansour (1998).

11 Der in diesem Kapitel beschriebene Standardalgorithmus DECISION-TREE-LEARNING verarbeitet keine Fälle, in denen für einige Beispiele Attributwerte fehlen.

a. Als Erstes müssen wir eine Möglichkeit finden, solche Beispiele zu klassifizieren, wenn wir einen Entscheidungsbaum haben, der Tests zu den Attributen beinhaltet, für die Werte fehlen können. Angenommen, ein Beispiel $\mathbf{x}$ hat einen fehlenden Wert für das Attribut A und die Entscheidungsbaumtests für A liegen an einem Knoten, den $\mathbf{x}$ erreicht. Um diesen Fall zu behandeln, könnte man vorgeben, dass das Beispiel *alle* möglichen Werte für das Attribut besitzt, aber jeden Wert entsprechend seiner Häufigkeit unter allen Beispielen gewichtet, die diesen Knoten im Entscheidungsbaum erreichen. Der Klassifizierungsalgorithmus sollte allen Zweigen an jedem Knoten folgen, für die ein Wert fehlt, und die Gewichte entlang jedes Pfades multiplizieren. Schreiben Sie einen abgeänderten Klassifizierungsalgorithmus für Entscheidungsbäume, der dieses Verhalten aufweist.

b. Ändern Sie jetzt die Berechnung des Informationsgewinnes so ab, dass in jeder beliebigen Sammlung von Beispielen C an einem bestimmten Knoten im Baum während des Konstruktionsvorganges die Beispiele, bei denen für eines der verbleibenden Attribute Werte fehlen, „als ob"-Werte erhalten, die den Häufigkeiten dieser Werte in der Menge C entsprechen.

12 In *Abschnitt 18.3.7* haben wir gesehen, dass Attribute mit vielen verschiedenen möglichen Werten Probleme mit der Gewinnbewertung erzeugen können. Solche Attribute weisen die Tendenz auf, Beispiele in viele kleine Klassen oder sogar in Klassen mit einem einzigen Element zu unterteilen, wodurch sie den Anschein erwecken, für die Gewinnbewertung sehr wichtig zu sein. Das Kriterium des **Gewinnverhältnisses** wählt Attribute nach dem Verhältnis zwischen ihrem Gewinn und ihrem intrinsischen Informationsgehalt aus – d.h. der Informationsmenge, die in der Antwort auf die Frage „Welchen Wert hat dieses Attribut?" enthalten ist. Das Kriterium des Gewinnverhältnisses versucht deshalb zu messen, wie effizient ein Attribut Informationen zur korrekten Klassifizierung eines Beispiels bereitstellt. Schreiben Sie einen mathematischen Ausdruck für den Informationsgehalt eines Attributes und implementieren Sie das Kriterium des Gewinnverhältnisses in DECISION-TREE-LEARNING.

Angenommen, Sie führen ein Lernexperiment für einen neuen Algorithmus zur booleschen Klassifizierung aus. Hierfür haben Sie eine Datenmenge von 100 positiven und 100 negativen Beispielen zur Verfügung. Sie wollen eine Leave-One-Out-Kreuzvalidierung verwenden und Ihren Algorithmus mit einer Bezugsfunktion – einem einfachen Mehrheitsklassifizierer – vergleichen. (Ein Mehrheitsklassifizierer übernimmt eine Menge von Trainingsdaten und gibt immer die Klasse aus, die sich in der Mehrheit der Trainingsmenge befindet, unabhängig von der Eingabe.) Sie erwarten, dass der Mehrheitsklassifizierer etwa 50% bei einer Leave-One-Out-Kreuzvalidierung erbringt, aber zu Ihrer Überraschung ermittelt er jedes Mal keinen Treffer. Können Sie erklären, warum das so ist?

13 Angenommen, ein Lernalgorithmus versucht, eine konsistente Hypothese zu finden, wenn die Klassifizierung der Beispiele tatsächlich zufällig ist. Es gibt n boolesche Attribute und die Beispiele werden einheitlich aus der Menge von 2^n möglichen Beispielen gezogen. Berechnen Sie die Anzahl der Beispiele, die benötigt werden, bevor die Wahrscheinlichkeit, einen Widerspruch in den Daten zu finden, 0,5 erreicht.

14 Beweisen Sie, dass eine Entscheidungsliste dieselbe Funktion als Entscheidungsbaum darstellen kann, wobei höchstens so viele Regeln verwendet werden, wie Blattknoten im Entscheidungsbaum für diese Funktion vorhanden sind. Geben Sie ein Beispiel für eine Funktion an, die durch eine Entscheidungsliste mit strikt weniger Regeln als die Anzahl der Blätter in einem Entscheidungsbaum minimaler Größe für dieselbe Funktion dargestellt wird.

15 Diese Übung beschäftigt sich mit der Ausdrucksstärke von Entscheidungslisten (*Abschnitt 18.5*).

a. Zeigen Sie, dass Entscheidungslisten jede boolesche Funktion darstellen können, wenn die Größe der Tests unbegrenzt ist.

b. Zeigen Sie, dass, wenn die Tests jeweils höchstens k Literale enthalten können, die Entscheidungslisten jede beliebige Funktion darstellen können, die durch einen Entscheidungsbaum der Tiefe k dargestellt werden kann.

16 Nehmen Sie an, eine 7-nächste-Nachbarn-Regressionssuche gibt $\{4, 2, 8, 4, 9, 11, 100\}$ als die sieben nächsten y-Werte für einen gegebenen x-Wert zurück. Welcher Wert $\hat{y}$ minimiert die L_1-Verlustfunktion für diese Daten? In der Statistik gibt es für diesen Wert als Funktion der y-Werte einen gebräuchlichen Namen. Wie lautet er? Beantworten Sie beide Fragen für die L_2-Verlustfunktion.

17 Abbildung 18.31 zeigt, wie sich ein Kreis am Ursprung linear trennen lässt, indem er von den Features (x_1, x_2) in die zwei Dimensionen $({x_1}^2, {x_2}^2)$ abgebildet wird. Wie sieht es aber aus, wenn sich der Kreis nicht im Ursprung befindet? Wie verhält es sich, wenn er eine Ellipse und kein Kreis ist? Die allgemeine Gleichung für einen Kreis (und folglich die Entscheidungsgrenze) lautet $(x_1 - a)^2 + (x_2 - b)^2 - r^2 = 0$ und die allgemeine Gleichung für eine Ellipse ist $c(x_1 - a)^2 + d(x_2 - b)^2 - 1 = 0$.

a. Multiplizieren Sie die Gleichung für den Kreis aus und zeigen Sie, wie die Gewichte w_i für die Entscheidungsgrenze im vierdimensionalen Merkmalsraum $(x_1, x_2, {x_1}^2, {x_2}^2)$ aussehen. Erläutern Sie, warum dies bedeutet, dass jeder Kreis in diesem Raum linear separierbar ist.

b. Führen Sie das Gleiche für Ellipsen im fünfdimensionalen Merkmalsraum $(x_1, x_2, {x_1}^2, {x_2}^2, x_1x_2)$ durch.

18 Konstruieren Sie eine Support-Vector-Maschine, die die XOR-Funktion berechnet. Verwenden Sie die Werte $+1$ und -1 (anstelle von 1 und 0) sowohl für die Eingaben als auch die Ausgaben, sodass ein Beispiel wie ([-1, 1], 1) oder ([-1, -1], -1) aussieht. Bilden Sie die Eingabe [x_1, x_2] in einen Raum ab, der aus x_1 und $x_1 x_2$ besteht. Zeichnen Sie die vier Eingabepunkte in diesem Raum sowie den Maximum-Margin-Separator. Was ist der Rand? Zeichnen Sie die Trennlinie zurück in den ursprünglichen Euklidschen Eingaberaum.

19 Betrachten Sie einen Gruppenlernalgorithmus, der eine einfache Mehrheitsabstimmung unter K gelernten Hypothesen ausführt. Angenommen, jede Hypothese hat den Fehler ϵ und die von jeder Hypothese gemachten Fehler sind unabhängig voneinander. Berechnen Sie eine Formel für den Fehler des Gruppenalgorithmus im Hinblick auf K und ϵ und werten Sie ihn für die Fälle mit $K = 5$, 10 und 20 und $\epsilon = 0{,}1$, 0,2 und 0,4 aus. Wenn die Annahme der Unabhängigkeit entfernt wird, ist es dann möglich, dass der Gruppenfehler *schlechter* als ϵ ist?

20 Konstruieren Sie manuell ein neuronales Netz, das die XOR-Funktion für zwei Eingaben berechnet. Spezifizieren Sie, welche Einheiten Sie verwenden.

21 Ein einfaches Perzeptron kann kein XOR (allgemein, die Paritätsfunktion seiner Eingaben) darstellen. Beschreiben Sie, was mit den Gewichtungen eines Schwellenwert-Perzeptrons mit vier Eingaben passiert, bei dem alle Gewichte anfangs auf 0,1 gesetzt sind, wenn Beispiele der Paritätsfunktion eintreffen.

22 Wie dieses Kapitel erläutert hat, gibt es 2^{2^n} verschiedene boolesche Funktionen für n Eingaben. Wie viele davon können durch ein Schwellenwert-Perzeptron dargestellt werden?

23 Betrachten Sie die folgende Beispielmenge mit je sechs Eingaben und einer Zielausgabe:

I_1	1	1	1	1	1	1	1	0	0	0	0	0	0	0
I_1	0	0	0	1	1	0	0	1	1	0	1	0	1	1
I_1	1	1	1	0	1	0	0	1	1	0	0	0	1	1
I_1	0	1	0	0	1	0	0	1	0	1	1	1	0	1
I_1	0	0	1	1	0	1	1	0	1	1	0	0	1	0
I_1	0	0	0	1	0	1	0	1	1	0	1	1	1	0
T	1	1	1	1	1	1	0	1	0	0	0	0	0	0

a. Führen Sie die Perzeptron-Lernregel für diese Daten aus und zeigen Sie die endgültigen Gewichte.

b. Führen Sie die Entscheidungsbaum-Lernregel aus und zeigen Sie den resultierenden Entscheidungsbaum.

c. Kommentieren Sie Ihre Ergebnisse.

24 *Abschnitt 18.6.4* hat darauf hingewiesen, dass die Ausgabe der logistischen Funktion als eine *Wahrscheinlichkeit* p interpretiert werden kann, die durch das Modell gemäß der Aussage, dass $f(\mathbf{x}) = 1$ ist, zugewiesen wird. Demzufolge ist die Wahrscheinlichkeit, dass $f(\mathbf{x}) = 0$ ist, gleich $1 - p$. Schreiben Sie die Wahrscheinlichkeit p als Funktion von $\mathbf{x}$ und berechnen Sie die Ableitung von $\log p$ in Bezug auf jedes Gewicht w_i. Wiederholen Sie den Prozess für $\log(1 - p)$. Diese Berechnungen liefern eine Lernregel für die Minimierung der negativen Log-Likelihood-Verlustfunktion für eine probabilistische Hypothese. Erörtern Sie Ähnlichkeiten mit anderen Lernregeln in diesem Kapitel.

25 Angenommen, Sie haben ein neuronales Netz mit linearen Aktivierungsfunktionen. Das bedeutet, für jede Einheit ist die Ausgabe eine Konstante c mal der gewichteten Summe der Eingaben.

a. Angenommen, das Netz hat eine verborgene Schicht. Schreiben Sie für eine gegebene Zuweisung zu den Gewichten $\mathbf{w}$ die Gleichungen für den Wert der Einheiten in der Ausgabeschicht als Funktion von $\mathbf{w}$ und der Eingabeschicht $\mathbf{x}$, ohne explizit die Ausgabe der verborgenen Schicht zu erwähnen. Zeigen Sie, dass es ein Netzwerk ohne verborgene Einheiten gibt, das dieselbe Funktion berechnet.

b. Wiederholen Sie die Berechnung aus Teil (a), jetzt aber für ein Netz mit einer beliebigen Anzahl verborgener Schichten.

c. Nehmen Sie an, ein Netz mit einer verborgenen Schicht und linearen Aktualisierungsfunktionen hat n Eingabe- und Ausgabeknoten sowie h verborgene Knoten. Welche Wirkung hat die Transformation in Teil (a) für ein Netz ohne verborgene Schichten auf die Gesamtanzahl der Gewichte? Erörtern Sie insbesondere den Fall $h << n$.

26 Implementieren Sie eine Datenstruktur für geschichtete neuronale Feedforward-Netze und stellen Sie dabei die Informationen bereit, die Sie für die Vorwärtsauswertung und die Fehlerrückführung (Backpropagation) brauchen. Schreiben Sie unter Verwendung dieser Datenstruktur eine Funktion NEURAL-NETWORK-OUTPUT, die ein Beispiel und ein Netz übernimmt und die entsprechenden Ausgabewerte berechnet.

27 Das neuronale Netz, dessen gemessene Leistung in Abbildung 18.25 dargestellt ist, hat vier verborgene Knoten. Diese Zahl wurde willkürlich gewählt. Ermitteln Sie durch eine Kreuzvalidierungsmethode die beste Anzahl von verborgenen Knoten.

28 Betrachten Sie das Problem, N Datenpunkte unter Verwendung einer linearen Trennung in positive und negative Beispiele zu unterteilen. Offensichtlich ist das für $N = 2$ Punkte auf einer Linie der Dimension $d = 1$ immer möglich, unabhängig davon, wie die Punkte beschriftet sind oder wo sie sich befinden (es sei denn, sie befinden sich an derselben Position).

a. Zeigen Sie, dass es für $N = 3$ Punkte auf einer Ebene der Dimension $d = 2$ immer möglich ist, es sei denn, sie sind kollinear.

b. Zeigen Sie, dass es für $N = 4$ Punkte auf einer Ebene der Dimension $d = 2$ nicht immer möglich ist.

c. Zeigen Sie, dass es für $N = 4$ Punkte in einem Raum der Dimension $d = 3$ immer möglich ist, es sei denn, sie sind koplanar.

d. Zeigen Sie, dass es für $N = 5$ Punkte in einem Raum der Dimension $d = 3$ nicht immer möglich ist.

e. Der ambitionierte Student beweist, dass N Punkte an allgemeiner Position (aber nicht $N + 1$) in einem Raum der Dimension $N - 1$ linear trennbar sind.

Wissen beim Lernen

19

ÜBERBLICK

In diesem Kapitel betrachten wir das Problem des Lernens, wenn man bereits über Vorwissen verfügt.

Alle in den vorherigen Kapiteln beschriebenen Lernansätze haben die grundlegende Idee, eine Funktion zu erzeugen, die das in den Daten beobachtete Eingabe/Ausgabe-Verhalten aufweist. In jedem Fall können die Lernmethoden als das Durchsuchen eines Hypothesenraumes betrachtet werden, um eine geeignete Funktion zu finden. Dabei geht man von einer sehr allgemeinen Annahme im Hinblick auf die Form der Funktion aus, wie beispielsweise „Polynom zweiten Grades“ oder „Entscheidungsbaum“, ebenso wie von einer Regel wie etwa „einfacher ist besser“. Das bedeutet, bevor man etwas Neues lernen kann, muss man (fast) alles vergessen, was man bereits weiß. In diesem Kapitel betrachten wir Lernmethoden, die ein **Vorwissen** über die Welt nutzen. Größtenteils wird das Vorwissen als allgemeine logische Theorie erster Stufe dargestellt; hier kombinieren wir also zum ersten Mal die Arbeit zur Wissensrepräsentation und das Lernen.

19.1 Eine logische Formulierung des Lernens

Kapitel 18 hat das rein induktive Lernen als Prozess beschrieben, eine Hypothese zu finden, die mit den beobachteten Beispielen übereinstimmt. Hier spezialisieren wir diese Definition auf den Fall, wo die Hypothese durch eine Menge logischer Sätze dargestellt wird. Beispielbeschreibungen und Klassifizierungen sind ebenfalls logische Sätze und ein neues Beispiel kann klassifiziert werden, indem ein Klassifizierungssatz von der Hypothese und der Beispielbeschreibung abgeleitet wird. Dieser Ansatz erlaubt einen inkrementellen Aufbau der Hypothesen – mit jeweils einem weiteren Satz. Darüber hinaus lässt er Vorwissen zu, weil Sätze, die bereits bekannt sind, bei der Klassifizierung neuer Beispiele helfen können. Die logische Formulierung des Lernens scheint zunächst eine Menge an zusätzlichem Arbeitsaufwand zu bedeuten, es stellt sich jedoch heraus, dass sie viele Aspekte des Lernens verständlicher macht. Sie erlaubt es uns, weit über die einfachen Lernmethoden aus *Kapitel 18* hinauszugehen, indem wir die volle Leistung logischer Inferenz für das Lernen benutzen.

19.1.1 Beispiele und Hypothesen

Aus *Kapitel 18* kennen Sie das Lernproblem für das Restaurantbeispiel: Es soll eine Regel gelernt werden, die entscheidet, ob man auf einen freien Tisch wartet. Beispiele wurden durch **Attribute** wie *Alternative*, *Bar*, *Frei/Sams* usw. beschrieben. In einer logischen Anordnung ist ein Beispiel ein Objekt, das durch einen logischen Satz beschrieben wird; die Attribute werden zu unären Prädikaten. Wir wollen das i-te Beispiel generisch als X_i bezeichnen. Das erste Beispiel aus Abbildung 18.3 etwa wird durch die folgenden Sätze beschrieben:

$$Alternative(X_1) \wedge \neg Bar(X_1) \wedge \neg Frei/Sams(X_1) \wedge Hungrig(X_1) \wedge \ldots$$

Mit der Notation $D_i(X_i)$ bezeichnen wir die Beschreibung von X_i, wobei D_i ein beliebiger logischer Ausdruck sein kann, der ein einziges Argument entgegennimmt. Die Klassifizierung des Beispiels wird durch ein Literal unter Verwendung des Zielprädikates gegeben, und zwar in diesem Fall:

$$WerdenWarten(X_1) \quad \text{oder} \quad \neg WerdenWarten(X_1).$$

Die vollständige Trainingsmenge lässt sich dann einfach als die Konjunktion aller Beispielbeschreibungen und Zielliterale ausdrücken.

Induktives Lernen verfolgt im Allgemeinen das Ziel, eine Hypothese zu finden, die die Beispiele gut klassifizieren und neue Beispiele gut verallgemeinern kann. Hier beschäftigen wir uns mit Hypothesen, die in Logik ausgedrückt sind; jede Hypothese h_j hat die Form

$$\forall x \; Ziel(x) \Leftrightarrow C_j(x),$$

wobei $C_j(x)$ eine Kandidatendefinition ist – ein Ausdruck, der die Attributprädikate betrifft. Zum Beispiel lässt sich ein Entscheidungsbaum als logischer Ausdruck dieser Form interpretieren. Somit drückt der Baum in Abbildung 18.6 die folgende logische Definition aus (die wir in zukünftigen Sachverhalten einfach als h_r bezeichnen wollen):

$$\begin{aligned} \forall r \quad WerdenWarten(r) \quad \Leftrightarrow \; & Gäste(r, Einige) \\ \vee \; & Gäste(r, Voll) \wedge Hungrig(r) \wedge Typ(r, Französisch) \\ \vee \; & Gäste(r, Voll) \wedge Hungrig(r) \wedge Typ(r, Thai) \\ & \qquad \wedge Frei/Sams(r) \\ \vee \; & Gäste(r, Voll) \wedge Hungrig(r) \wedge Typ(r, Burger). \end{aligned} \tag{19.1}$$

Jede Hypothese sagt voraus, dass eine bestimmte Menge an Beispielen – nämlich solche, die die Kandidatendefinition erfüllen – Beispiele für das Zielprädikat sind. Diese Menge wird auch als **Extension** des Prädikates bezeichnet. Zwei Hypothesen mit unterschiedlichen Extensionen sind deshalb logisch inkonsistent zueinander, weil sie in ihren Vorhersagen für mindestens ein Beispiel nicht übereinstimmen. Wenn sie dieselbe Extension haben, sind sie logisch äquivalent.

Der Hypothesenraum $\mathcal{H}$ ist die Menge aller Hypothesen $\{h_1, ..., h_n\}$, auf die der Lernalgorithmus ausgelegt ist. Beispielsweise kann der Algorithmus DECISION-TREE-LEARNING jede Entscheidungsbaumhypothese unterhalten, die hinsichtlich der bereitgestellten Attribute definiert ist; sein Hypothesenraum besteht deshalb aus allen diesen Entscheidungsbäumen. Vermutlich glaubt der Lernalgorithmus, dass eine der Hypothesen korrekt ist; d.h., er glaubt den Satz

$$h_1 \vee h_2 \vee h_3 \vee ... \vee h_n . \tag{19.2}$$

Wenn die Beispiele eintreffen, können Hypothesen, die nicht mit ihnen **konsistent** sind, ausgeschlossen werden. Wir wollen das Konzept der Konsistenz genauer betrachten. Wenn die Hypothese h_j konsistent mit der gesamten Trainingsmenge ist, muss sie offenbar auch mit jedem Beispiel in der Trainingsmenge konsistent sein. Was würde es bedeuten, wenn sie mit einem Beispiel nicht konsistent ist? Das kann auf zweierlei Arten der Fall sein:

- Ein Beispiel kann für die Hypothese **falsch negativ** sein, wenn die Hypothese sagt, es sollte negativ sein, es tatsächlich aber positiv ist. Beispielsweise wäre das neue Beispiel X_{13}, beschrieben durch $Gäste(X_{13}, Voll) \wedge \neg Hungrig(X_{13}) \wedge ... \wedge WerdenWarten(X_{13})$, falsch negativ für die zuvor gezeigte Hypothese h_r. Aus h_r und der Beispielbeschreibung können wir sowohl $WerdenWarten(X_{13})$ ableiten, nämlich was das Beispiel aussagt, als auch $\neg WerdenWarten(X_{13})$, was die Hypothese voraussagt. Die Hypothese und das Beispiel sind also logisch inkonsistent.
- Ein Beispiel kann **falsch positiv** für die Hypothese sein, wenn die Hypothese besagt, es sollte positiv sein, es tatsächlich aber negativ ist![1]

1 Die Begriffe „falsch positiv“ und „falsch negativ“ werden in der Medizin verwendet, um fehlerhafte Ergebnisse aus Labortests zu beschreiben. Ein Ergebnis ist falsch positiv, wenn es besagt, dass der Patient die Krankheit hat, während er überhaupt nicht krank ist.

Wenn ein Beispiel für eine Hypothese falsch positiv oder falsch negativ ist, sind das Beispiel und die Hypothesen logisch inkonsistent zueinander. Vorausgesetzt, bei dem Beispiel handelt es sich um eine korrekte Beobachtung der Tatsache, kann die Hypothese ausgeschlossen werden. Logisch ist dies genau analog zu der Resolutionsregel der Inferenz (siehe *Kapitel 9*), wo die Disjunktion von Hypothesen einer Klausel entspricht und das Beispiel einem Literal, das gegen eines der Literale in der Klausel resolviert wird. Ein normales logisches Inferenzsystem könnte also im Prinzip aus dem Beispiel lernen, indem es eine oder mehrere Hypothesen eliminiert. Nehmen wir nur einmal an, das Beispiel ist durch den Satz I_1 angegeben und der Hypothesenraum ist $h_1 \vee h_2 \vee h_3 \vee h_4$. Wenn dann I_1 inkonsistent mit h_2 und h_3 ist, kann das logische Inferenzsystem schließen, dass der neue Hypothesenraum gleich $h_1 \vee h_4$ ist.

Wir können also das induktive Lernen in einem logischen Aufbau als Prozess der schrittweisen Eliminierung von Hypothesen charakterisieren, die inkonsistent mit den Beispielen sind, wodurch wir die Möglichkeiten genauer eingrenzen. Weil der Hypothesenraum in der Regel sehr groß ist (oder im Fall der Logik erster Stufe sogar unendlich), empfehlen wir nicht, ein Lernsystem unter Verwendung resolutionsbasierter Theorembeweiser und einer vollständigen Aufzählung des Hypothesenraumes zu erstellen. Stattdessen beschreiben wir zwei Ansätze, die logisch konsistente Hypothesen mit einem viel geringeren Aufwand erkennen.

19.1.2 Aktuell-beste-Hypothese-Suche

Die Idee hinter der **Aktuell-beste-Hypothese**-Suche ist, eine einzige Hypothese zu verwalten und sie anzupassen, wenn neue Beispiele eintreffen, um die Konsistenz zu wahren. Der grundlegende Algorithmus wurde von John Stuart Mill (1843) beschrieben, kann aber auch schon sehr viel früher formuliert worden sein.

Angenommen, wir haben eine Hypothese wie etwa h_r, von der wir sehr überzeugt sind. Solange jedes neue Beispiel konsistent ist, brauchen wir nichts zu tun. Dann trifft ein falsches negatives Beispiel ein, X_{13}. Was machen wir? ▶ Abbildung 19.1 zeigt h_r schematisch als Bereich: Alles innerhalb des Rechtecks gehört zur Extension von h_r. Die Beispiele, die bisher aufgetreten sind, werden als „+“ und „−“ dargestellt und wir sehen, dass h_r alle Beispiele korrekt als positive oder negative Beispiele von *Werden-Warten* charakterisiert. In Abbildung 19.1(b) ist ein neues Beispiel (im Kreis) falsch negativ: Die Hypothese besagt, dass es negativ sein muss, aber eigentlich ist es positiv. Die Extension der Hypothese muss vergrößert werden, sodass sie dieses Beispiel berücksichtigt. Man spricht auch von einer **Verallgemeinerung**. Abbildung 19.1(c) zeigt eine mögliche Verallgemeinerung. In Abbildung 19.1(d) sehen wir ein falsch positives Beispiel: Die Hypothese besagt, dass das neue Beispiel (im Kreis) positiv sein sollte, doch es ist tatsächlich negativ. Die Extension der Hypothese muss verkleinert werden, um das Beispiel auszuschließen. Man spricht von einer **Spezialisierung**; in Abbildung 19.1(e) sehen wir eine mögliche Spezialisierung der Hypothese. Die Relationen „allgemeiner als“ und „spezifischer als“ zwischen den Hypothesen stellen die logische Struktur im Hypothesenraum her, die eine effiziente Suche ermöglicht.

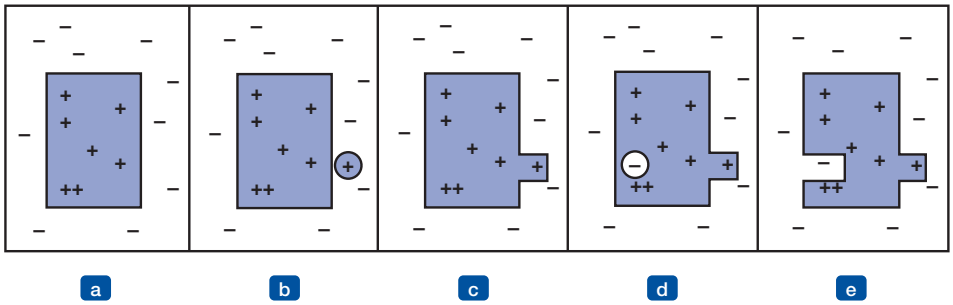


Abbildung 19.1: (a) Eine konsistente Hypothese. (b) Ein falsch negatives Beispiel. (c) Die Hypothese wird verallgemeinert. (d) Ein falsch positives Beispiel. (e) Die Hypothese wird spezialisiert.

Jetzt können wir den in ▶ Abbildung 19.2 gezeigten Algorithmus CURRENT-BEST-LEARNING spezifizieren. Beachten Sie, dass wir bei jeder Verallgemeinerung oder Spezialisierung der Hypothese die Konsistenz zu den anderen Beispielen überprüfen müssen, weil eine beliebige Vergrößerung/Verkleinerung der Extension zuvor betrachtete negative/positive Beispiele aufnehmen/ausschließen könnte.

```
function CURRENT-BEST-LEARNING(examples, h) returns eine Hypothese oder Fehler

  if examples ist leer then
    return h
  e ← FIRST(examples)
  if e ist konsistent mit h then
    return CURRENT-BEST-LEARNING(REST(examples), h)
  else if e ist falsch positiv für h then
    for each h' in Spezialisierungen von h
                   konsistent mit bisher gesehenen examples do
      h'' ← CURRENT-BEST-LEARNING(REST(examples), h')
      if h'' ≠ fail then return h''
  else if e ist falsch negativ für h then
    for each h' in Verallgemeinerungen von h
                   konsistent mit bisher gesehenen examples do
      h'' ← CURRENT-BEST-LEARNING(REST(examples), h')
      if h'' ≠ fail then return h''
  return fail
```

Abbildung 19.2: Der Lernalgorithmus der gegenwärtig besten Hypothese. Er sucht nach einer konsistenten Hypothese, die mit allen Beispielen übereinstimmt, und geht zurück, wenn keine konsistente Spezialisierung/Verallgemeinerung gefunden werden konnte. Um den Algorithmus zu starten, kann jede Hypothese übergeben werden; sie wird bei Bedarf spezialisiert oder verallgemeinert.

Wir haben die Verallgemeinerung und Spezialisierung als Operationen definiert, die die *Extension* einer Hypothese ändern. Jetzt müssen wir überlegen, wie sie als syntaktische Operationen implementiert werden können, die die der Hypothese zugeordnete Kandidatendefinition ändern, sodass ein Programm sie ausführen kann. Dazu muss man als Erstes erkennen, dass Verallgemeinerung und Spezialisierung auch *logische* Beziehungen zwischen den Hypothesen sind. Wenn die Hypothese h_1 mit der Definition C_1 eine Verallgemeinerung der Hypothese h_2 mit der Definition C_2 ist, muss Folgendes gelten:

$$\forall x \quad C_2(x) \Rightarrow C_1(x).$$

Um also eine Verallgemeinerung von h_2 zu erzeugen, müssen wir einfach nur eine Definition C_1 finden, die von C_2 logisch impliziert wird. Das ist ganz einfach. Ist $C_2(x)$ beispielsweise $Alternative(x) \wedge Gäste(x, Einige)$, ist eine mögliche Verallgemeinerung gegeben durch $C_1(x) \equiv Gäste(x, Einige)$. Man spricht auch von **Bedingungen fallen lassen**. Intuitiv erkennt man, dass damit eine schwächere Definition erzeugt wird, sodass eine größere Menge positiver Beispiele entstehen kann. Es gibt viele andere Verallgemeinerungsoperationen – abhängig von der Sprache, mit der gearbeitet wird. Analog dazu können wir eine Hypothese spezialisieren, indem wir ihrer Kandidatendefinition zusätzliche Bedingungen hinzufügen oder Disjunkte aus einer disjunkten Definition entfernen. Jetzt betrachten wir, wie das für unser Restaurantbeispiel aussieht. Wir verwenden die Daten aus Abbildung 18.3.

- Das erste Beispiel, X_1, ist positiv. Das Attribut $Alternative(X_1)$ ist wahr; sei also die erste Hypothese gleich

$$h_1 : \forall x \quad WerdenWarten(x) \Leftrightarrow Alternative(x).$$

- Das zweite Beispiel, X_2, ist negativ. h_1 sagt es als positiv voraus; es handelt sich also um ein falsch positives Beispiel. Wir müssen h_1 spezialisieren. Dazu fügen wir eine zusätzliche Bedingung ein, die X_2 ausschließt, während wir X_1 als positiv klassifizieren. Eine Möglichkeit dafür könnte wie folgt aussehen:

$$h_2 : \forall x \quad WerdenWarten(x) \Leftrightarrow Alternative(x) \wedge Gäste(x, Einige).$$

- Das dritte Beispiel, X_3, ist positiv. h_2 sagt es als negativ voraus; es handelt sich also um ein falsch negatives Beispiel. Wir müssen h_2 verallgemeinern. Wir lassen die Bedingung *Alternative* fallen, sodass sich Folgendes ergibt:

$$h_3 : \forall x \quad WerdenWarten(x) \Leftrightarrow Gäste(x, Einige).$$

- Das vierte Beispiel, X_4, ist positiv. h_3 sagt es als negativ voraus; es handelt sich also um ein falsch negatives Beispiel. Demnach müssen wir h_3 verallgemeinern. Wir können die Bedingung *Gäste* nicht fallen lassen, weil wir damit zu einer allumfassenden Hypothese gelangen würden, die mit X_2 inkonsistent wäre. Eine Möglichkeit ist, ein Disjunkt einzufügen:

$$h_4 : \forall x \quad WerdenWarten(x) \Leftrightarrow Gäste(x, Einige) \vee (Gäste(x, Voll) \wedge Frei/Sams(x)).$$

Jetzt beginnt die Hypothese sinnvoll auszusehen. Offensichtlich gibt es aber weitere Möglichkeiten, die mit den ersten vier Beispielen konsistent sind; hier zwei davon:

$$h_4' : \forall x \quad WerdenWarten(x) \Leftrightarrow \neg WarteZeit(x, 30\text{-}60)$$
$$h_4'' : \forall x \quad WerdenWarten(x) \Leftrightarrow Gäste(x, Einige) \vee (Gäste(x, Voll) \wedge Wartezeit(x, 30\text{-}60)).$$

Der Algorithmus CURRENT-BEST-LEARNING ist nichtdeterministisch beschrieben, weil es zu jedem Zeitpunkt mehrere mögliche Spezialisierungen oder Verallgemeinerungen geben kann, die angewendet werden können. Die getroffenen Auswahlen führen nicht unbedingt zur einfachsten Hypothese und können eine unauflösbare Situation verursachen, wo keine einfache Änderung der Hypothese mehr konsistent mit allen Daten ist. In solchen Fällen muss das Programm zurück zu einem vorhergehenden Auswahlpunkt gehen.

Der Algorithmus CURRENT-BEST-LEARNING und seine Varianten wurden in vielen Maschinenlernsystemen eingesetzt, beginnend mit dem „Arch-Learning"-Programm von Patrick Winston (1970). Mit sehr vielen Instanzen und einem großen Raum entstehen jedoch einige Schwierigkeiten:

1. Die Überprüfung aller vorhergehenden Instanzen bei jeder Veränderung ist sehr aufwändig.
2. Für den Suchprozess kann sehr viel Backtracking erforderlich sein. Wie wir in *Kapitel 18* gesehen haben, kann der Hypothesenraum ein doppelt exponentiell großer Raum sein.

19.1.3 Geringste-Verpflichtung-Suche

Ein Backtracking fällt an, wenn der Ansatz der aktuell besten Hypothese eine bestimmte Hypothese als beste Schätzung *auswählen* muss, auch wenn nicht genügend Daten zur Verfügung stehen, die diese Auswahl untermauern. Wir können stattdessen aber auch genau die Hypothesen beibehalten, die mit allen bisher angefallenen Daten konsistent sind. Jedes neue Beispiel hat entweder keine Auswirkung oder es entfernt einige der Hypothesen. Wie Sie wissen, lässt sich der ursprüngliche Hypothesenraum als disjunktiver Satz betrachten:

$$h_1 \vee h_2 \vee h_3 \vee ... \vee h_n .$$

Wenn festgestellt wird, dass verschiedene Hypothesen inkonsistent mit den Beispielen sind, schrumpft die Disjunktion und behält nur die Hypothesen bei, die nicht ausgeschlossen wurden. Angenommen, der ursprüngliche Hypothesenraum enthält die richtige Antwort, dann muss die reduzierte Disjunktion die richtige Antwort immer noch enthalten, weil nur falsche Hypothesen entfernt wurden. Die Menge der verbleibenden Hypothesen wird auch als **Versionsraum** bezeichnet und der Lernalgorithmus (in ▶ Abbildung 19.3 skizziert) wird als Versionsraum-Lernalgorithmus (auch **Kandidateneliminierungs**-Algorithmus) bezeichnet.

```
function VERSION-SPACE-LEARNING(examples) returns einen Versionsraum
   local variables: V, der Versionsraum: die Menge aller Hypothesen

   V ← die Menge aller Hypothesen
   for each Beispiel e in examples do
      if V ist nicht leer then V ← VERSION-SPACE-UPDATE(V, e)
   return V

function VERSION-SPACE-UPDATE(V, e) returns einen aktualisierten Versionsraum
   V ← {h ∈ V : h ist konsistent mit e}
```

Abbildung 19.3: Der Versionsraum-Lernalgorithmus. Er ermittelt eine Untermenge von *V*, die konsistent mit allen Beispielen (*examples*) ist.

Eine wichtige Eigenschaft dieses Ansatzes ist, dass er *inkrementell* ist: Man muss nie zurückgehen und alte Beispiele erneut betrachten. Alle verbleibenden Hypothesen sind garantiert bereits mit ihnen konsistent. Es gibt jedoch ein offensichtliches Problem. Wir haben bereits gesagt, dass der Hypothesenraum riesig ist. Wie können wir diese gigantische Disjunktion aufschreiben?

Die folgende einfache Analogie kann Ihnen bei der Überlegung helfen. Wie stellen Sie alle realen Zahlen zwischen 1 und 2 dar? Schließlich gibt es unendlich viele solcher Zahlen! Die Antwort ist, eine Intervalldarstellung zu verwenden, die einfach nur die Grenzen der Menge angibt: [1, 2]. Das funktioniert, weil die realen Zahlen *geordnet* sind.

Auch der Hypothesenraum ist geordnet, nämlich durch Verallgemeinerung/Spezialisierung. Dabei handelt es sich um eine partielle Ordnung, d.h., die Grenzen sind keine Punkte, sondern Mengen von Hypothesen, auch als **Grenzmenge** bezeichnet. Praktisch dabei ist, dass wir den gesamten Versionsraum unter Verwendung von nur zwei Grenzmengen darstellen können: mit der allgemeinsten Grenze (der **G-Menge**) und der spezifischsten Grenze (der **S-Menge**). *Alles, was zwischen diesen Grenzen liegt, ist garantiert konsistent mit den Beispielen*. Bevor wir dies beweisen, wollen wir noch einmal zusammenfassen:

- Der aktuelle Versionsraum ist die Menge der Hypothesen, die konsistent mit allen bisherigen Beispielen sind. Er wird durch die S- und die G-Menge, zwei Hypothesenmengen, dargestellt.
- Jedes Element der S-Menge ist konsistent mit allen bisherigen Beobachtungen und es gibt keine konsistenten Hypothesen, die spezifischer sind.
- Jedes Element der G-Menge ist konsistent mit allen bisherigen Beobachtungen und es gibt keine konsistenten Hypothesen, die allgemeiner sind.

Wir wollen, dass der anfängliche Versionsraum (bevor irgendwelche Beispiele betrachtet wurden) alle möglichen Hypothesen repräsentiert. Dazu setzen wir die G-Menge so, dass sie *True* enthält (die Hypothese, die alles enthält), und die S-Menge so, dass sie *False* enthält (die Hypothese mit leerer Extension).

▶ Abbildung 19.4 zeigt die allgemeine Struktur der Grenzmengendarstellung des Versionsraumes. Um zu zeigen, dass die Repräsentation ausreichend ist, brauchen wir die beiden folgenden Eigenschaften:

1 Jede konsistente Hypothese (die nicht in den Grenzmengen enthalten ist) ist spezifischer als ein Element der G-Menge und allgemeiner als ein Element der S-Menge. (D.h., es sind keine „Nachzügler“ draußen geblieben.) Das folgt direkt aus den Definitionen von S und G. Gäbe es einen Nachzügler h, dürfte er nicht spezifischer als ein Element von G sein, dann würde er zu G gehören, und nicht allgemeiner als ein Element von S, dann würde er zu S gehören.

2 Jede Hypothese, die spezifischer als ein Element der G-Menge und allgemeiner als ein Element der S-Menge ist, ist eine konsistente Hypothese. (D.h., es gibt keine „Löcher“ zwischen den Grenzen.) Jede Hypothese h zwischen S und G muss alle negativen Beispiele zurückweisen, die alle Elemente von G zurückgewiesen haben (weil es spezifischer ist), und alle positiven Beispiele akzeptieren, die von allen Elementen von S akzeptiert wurden (weil es allgemeiner ist). Das bedeutet, h muss mit allen Beispielen übereinstimmen und kann deshalb nicht inkonsistent sein. ▶ Abbildung 19.5 zeigt die Situation: Es gibt keine bekannten Beispiele außerhalb von S und innerhalb von G, deshalb muss jede Hypothese dazwischen konsistent sein.

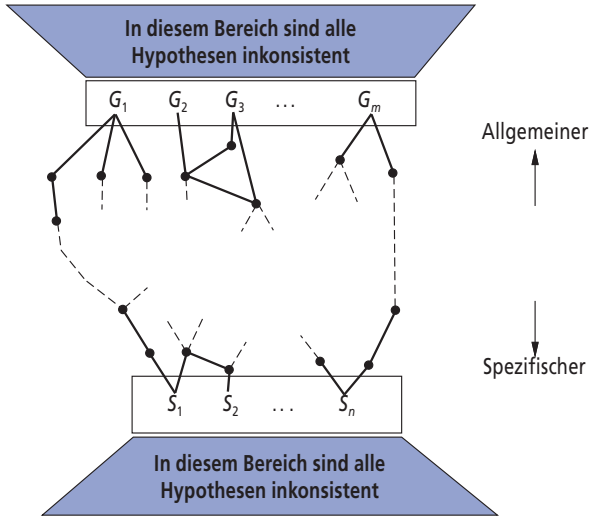


Abbildung 19.4: Der Versionsraum enthält alle Hypothesen, die konsistent mit den Beispielen sind.

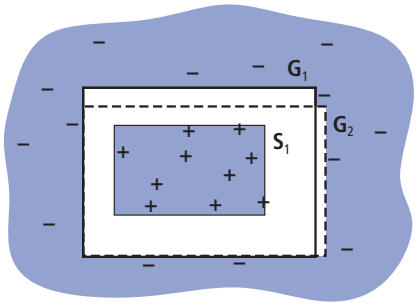


Abbildung 19.5: Die Extensionen der Elemente von G und S. Nicht bekannte Beispiele liegen zwischen den beiden Grenzmengen.

Wir haben damit gezeigt, dass S und G eine zufriedenstellende Repräsentation des Versionsraumes bereitstellen, wenn sie ihren Definitionen gemäß verwaltet werden. Das einzig verbleibende Problem ist, wie S und G für ein neues Beispiel aktualisiert werden sollen (das ist die Aufgabe der Funktion VERSION-SPACE-UPDATE). Das erscheint auf den ersten Blick vielleicht kompliziert, aber aus den Definitionen und mithilfe von Abbildung 19.4 ist es nicht schwierig, den Algorithmus zu rekonstruieren.

Wir müssen uns Gedanken über die Elemente S_i und G_i der S- und G-Mengen machen. Für jedes davon kann die neue Instanz ein falsch positives oder ein falsch negatives Beispiel sein.

1 Falsch positiv für S_i: Das bedeutet, S_i ist zu allgemein, aber es gibt keine konsistenten Spezialisierungen von S_i (definitionsgemäß), deshalb werfen wir es aus der S-Menge.

2 Falsch negativ für S_i: Das bedeutet, S_i ist zu spezifisch, deshalb ersetzen wir es durch alle unmittelbaren Verallgemeinerungen, vorausgesetzt, sie sind spezifischer als ein Element von G.

3 Falsch positiv für G_i: Das bedeutet, G_i ist zu allgemein, deshalb ersetzen wir es durch alle unmittelbaren Spezialisierungen, vorausgesetzt, sie sind allgemeiner als ein Element von S.

4 Falsch negativ für G_i: Das bedeutet, G_i ist zu spezifisch, aber es gibt keine konsistenten Verallgemeinerungen von G_i (definitionsgemäß), deshalb werfen wir es aus der G-Menge.

Wir setzen diese Operationen für jede neue Instanz fort, bis eines von drei Dingen passiert:

1 Es bleibt nur noch genau eine Hypothese im Versionsraum zurück, die wir dann als einzige Hypothese zurückgeben.

2 Der Versionsraum *kollabiert* – S oder G wird leer, d.h., es wurden keine konsistenten Hypothesen für die Trainingsmenge gefunden. Das ist derselbe Fall wie ein Fehlschlagen der einfachen Version des Entscheidungsbaum-Algorithmus.

3 Die Beispiele gehen uns aus und es bleiben mehrere Hypothesen im Versionsraum zurück. Das bedeutet, der Versionsraum stellt eine Disjunktion von Hypothesen dar. Wenn alle Disjunkte übereinstimmen, können wir für jedes neue Beispiel ihre Klassifizierung des Beispiels zurückgeben. Stimmen sie nicht überein, besteht die Möglichkeit, die Mehrheitsabstimmung zu nutzen.

Wir überlassen es dem Leser als Übung, den Algorithmus VERSION-SPACE-LEARNING auf die Restaurantdaten anzuwenden.

Der Versionsraumansatz hat einige wichtige Nachteile:

- Wenn die Domäne Rauschen oder unzureichende Attribute für eine genaue Klassifizierung enthält, kollabiert der Versionsraum immer.
- Wenn wir unbegrenzte Disjunktion im Hypothesenraum erlauben, enthält die S-Menge immer eine einzige spezifischste Hypothese, nämlich die Disjunktion der Beschreibungen der bisher betrachteten positiven Beispiele. Analog dazu enthält die G-Menge einfach die Negierung der Disjunktion der Beschreibungen der negativen Beispiele.
- Für einige Hypothesenräume kann die Anzahl der Elemente in der S-Menge oder in der G-Menge exponentiell zur Anzahl der Attribute anwachsen, selbst wenn es effiziente Lernalgorithmen für diese Hypothesenräume gibt.

Bisher wurde keine wirklich vollständige Lösung für das Problem des Rauschens gefunden. Das Problem der Disjunktion kann gelöst werden, indem man beschränkte Formen der Disjunktion zulässt oder eine **Verallgemeinerungshierarchie** allgemeinerer Prädikate aufnimmt. Anstatt beispielsweise die Disjunktion $Wartezeit(x, 30\text{-}60) \vee Wartezeit(x, >60)$ zu verwenden, könnten wir das Einzelliteral $LangeWartezeit(x)$ verwenden. Die Menge der Verallgemeinerungs- und Spezialisierungsoperationen kann einfach erweitert werden, um dies zu berücksichtigen.

Der reine Versionsraumalgorithmus wurde zuerst im System Meta-DENDRAL angewendet, das Regeln für die Vorhersage lernte, wie sich Moleküle in einem Massenspektrometer zerteilen (Buchanan und Mitchell, 1978). Meta-DENDRAL konnte Regeln erzeugen, die ausreichend neu waren, um die Veröffentlichung in einem Magazin für analytische Chemie zu veranlassen – das erste wirklich wissenschaftliche Wissen, das von einem Computerprogramm erzeugt worden war. Er wurde auch in dem eleganten LEX-System (Mitchell et al., 1983) verwendet, das symbolische Integrationsprobleme lösen lernte, indem es eigene Erfolge und Misserfolge beobachtete. Obwohl Versionsraummethoden

für die meisten Lernprobleme der realen Welt wahrscheinlich nicht praktisch sind (hauptsächlich wegen des Rauschens), bieten sie ausreichend gute Einsichten in die logische Struktur des Hypothesenraumes.

19.2 Wissen beim Lernen

Der vorige Abschnitt hat den einfachsten Aufbau für induktives Lernen beschrieben. Um die Rolle des Vorwissens zu verstehen, müssen wir über die logischen Beziehungen zwischen den Hypothesen, Beispielbeschreibungen und Klassifizierungen sprechen. Sei *Beschreibungen* die Konjunktion aller Beispielbeschreibungen in der Trainingsmenge und *Klassifizierungen* die Konjunktion aller Beispielklassifizierungen. Eine *Hypothese*, die „die Beobachtungen erklärt", muss dann die folgende Eigenschaft erfüllen (Sie wissen, dass das Zeichen $\models$ die Bedeutung „ist eine logische Konsequenz von" hat):

$$Hypothese \wedge Beschreibungen \models Klassifizierungen. \qquad (19.3)$$

Wir bezeichnen diese Art Beziehung auch als **Bedingung der logischen Konsequenz**, wobei *Hypothese* die „Unbekannte" ist. Rein induktives Lernen bedeutet, dass diese Bedingung aufgelöst wird, wobei *Hypothese* aus einem vordefinierten Hypothesenraum gewählt wird. Wenn wir beispielsweise einen Entscheidungsbaum als logische Formel betrachten (siehe Gleichung 19.1), erfüllt ein Entscheidungsbaum, der konsistent mit allen Beispielen ist, Gleichung (19.3). Wenn wir *keine* Bedingungen für die logische Form der Hypothese angeben, genügt natürlich auch *Hypothese* = *Klassifizierungen* der Bedingung. Ockhams Rasiermesser sagt uns, wir sollen *kleine*, konsistente Hypothesen bevorzugen; deshalb versuchen wir, eine bessere Lösung zu finden, als uns einfach die Beispiele zu merken.

Tipp

Dieses einfache wissensfreie Bild des induktiven Lernens bestand bis Anfang der 1980er Jahre. Der moderne Ansatz entwirft Agenten, die *bereits etwas wissen* und versuchen, dazuzulernen. Das scheint keine besonders tiefe Erkenntnis zu sein, aber sie zieht einen völlig anderen Entwurf der Agenten nach sich. Vielleicht hat sie auch eine gewisse Bedeutung für unsere Theorien, wie die eigentliche Wissenschaft funktioniert. Die allgemeine Idee ist in ▶ Abbildung 19.6 skizziert.

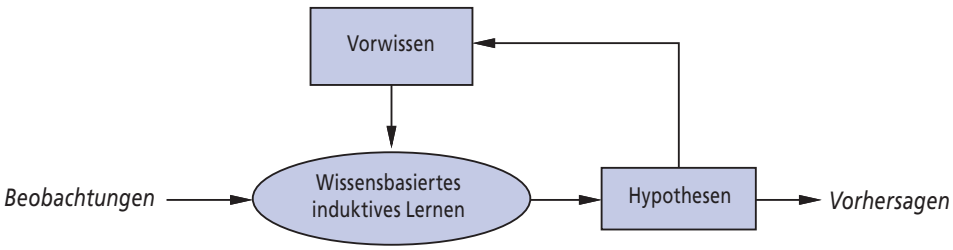


Abbildung 19.6: Ein kumulativer Lernprozess verwendet sein Hintergrundwissen und ergänzt es im Verlauf der Zeit.

Ein autonomer Lernagent, der Hintergrundwissen verwendet, muss das Hintergrundwissen zunächst erlangen, um es in neuen Lernepisoden verwenden zu können. Diese Methode muss selbst wiederum ein Lernprozess sein. Der Lebensverlauf des Agenten ist deshalb durch eine *kumulative* oder *inkrementelle* Entwicklung charakterisiert. Vielleicht könnte der Agent auch mit nichts anfangen und Induktionen *in vacuo* durchführen wie ein kleines feines Induktionsprogramm. Nachdem er jedoch vom Baum der

Erkenntnis gegessen hat, reichen ihm diese naiven Spekulationen nicht mehr aus und er sollte sein Hintergrundwissen nutzen, um immer effektiver zu lernen. Die Frage ist dann, wie das geht.

19.2.1 Einfache Beispiele

Wir wollen einige allgemeine Beispiele für das Lernen mit Hintergrundwissen betrachten. Viele scheinbar rationale Fälle inferentiellen Verhaltens bei vorhandenen Beobachtungen folgen offensichtlich nicht den einfachen Prinzipien reiner Induktion.

- Manchmal gelangt man nach nur einer Beobachtung sprunghaft zu allgemeinen Schlüssen. Gary Larson zeichnete einmal einen Cartoon, wo ein bebrillter Höhlenbewohner, Zog, seine Eidechse am Ende eines angespitzten Stöckchens im Feuer brät. Er wird von einer begeisterten Menge seiner weniger intelligenten Zeitgenossen beobachtet, die ihre Beute mit den bloßen Händen übers Feuer halten. Diese aufklärende Erfahrung ist ausreichend, um die Beobachter von einem allgemeinen Prinzip schmerzfreien Kochens zu überzeugen.
- Oder betrachten Sie die Geschichte des Brasilienreisenden, der seinen ersten Brasilianer traf. Nachdem er gehört hatte, dass dieser Portugiesisch sprach, schloss er sofort, dass Brasilianer Portugiesisch sprechen, aber obwohl er hörte, dass sein Name Fernando ist, schloss er nicht, dass alle Brasilianer Fernando heißen. Ähnliche Beispiele gibt es in der Wissenschaft. Wenn beispielsweise ein Physikstudent im Grundstudium die Dichte und Leitfähigkeit einer Kupferstichprobe bei einer bestimmten Temperatur misst, geht er davon aus, dass er diese Werte für alle Kupferstücke verallgemeinern kann. Wenn er die Masse misst, denkt er nicht einmal daran, dass alle Kupferstücke dieselbe Masse haben könnten. Es wäre dagegen recht vernünftig, diese Verallgemeinerung für alle Pfennigstücke zu treffen.
- Als Letztes betrachten wir das Beispiel eines pharmakologisch unwissenden, aber diagnostisch gebildeten Medizinstudenten, der eine Sitzung zwischen einem Patienten und einem erfahrenen Internisten beobachtet. Nach einigen Fragen und Antworten teilt der Experte dem Patienten mit, dass er ein bestimmtes Antibiotikum einnehmen soll. Der Medizinstudent leitet die allgemeine Regel ab, dass dieses bestimmte Antibiotikum wirksam für eine bestimmte Art Infektion ist.

Tipp

All dies sind Fälle, *wo der Einsatz von Hintergrundwissen ein viel schnelleres Lernen erlaubt, als man es von einem rein induktiven Programm erwartet.*

19.2.2 Einige allgemeine Schemata

In jedem der gezeigten Beispiele kann man sich auf das Vorwissen beziehen, um die gewählten Verallgemeinerungen zu rechtfertigen. Jetzt betrachten wir, welche Bedingungen der logischen Schlussfolgerung in den einzelnen Fällen gelten. Die Bedingungen beinhalten das *Hintergrund*-Wissen ebenso wie die *Hypothese* und die beobachteten *Beschreibungen* und *Klassifizierungen*.

Bei der Rösteidechse verallgemeinern die Höhlenmenschen, indem sie den Erfolg des angespitzten Stöckchens *erklären*: Es erlaubt es, die Eidechse zu braten, ohne mit der Hand ins Feuer fassen zu müssen. Aus dieser Erklärung können sie eine allgemeine Regel ableiten: dass jedes lange, starre und scharf gespitzte Objekt verwendet werden kann, um kleine, weiche Nahrungsmittel zu grillen. Diese Art von Verallgemeine-

rungsprozess wird als **erklärungsbasiertes Lernen** (**EBL**) bezeichnet. Beachten Sie, dass die allgemeine Regel *logisch* aus dem Hintergrundwissen der Höhlenmenschen *folgt*. Die Bedingungen der logischen Konsequenz, die durch das EBL erfüllt wird, sehen also wie folgt aus:

$$Hypothese \wedge Beschreibungen \models Klassifizierungen$$
$$Hintergrund \models Hypothese.$$

Tipp

Weil EBL Gleichung (19.3) verwendet, hielt man es anfänglich für eine bessere Möglichkeit, aus Beispielen zu lernen. Weil es dafür jedoch erforderlich ist, dass das Hintergrundwissen für die Erklärung der *Hypothese* ausreichend ist, was wiederum die Beobachtungen erklärt, *lernt der Agent faktisch nichts Neues aus diesem Beispiel*. Der Agent *hätte* das Beispiel von dem ableiten können, was er bereits wusste, wenn auch vielleicht mit einem übermäßigen Rechenaufwand. Heute wird EBL als Methode für die Umwandlung von hauptsächlich grundsätzlichen Theorien in praktisches, spezielles Wissen betrachtet. In *Abschnitt 19.3* beschreiben wir Algorithmen für EBL.

Die Situation unseres Brasilienreisenden ist ganz anders, weil er nicht unbedingt erklären kann, warum Fernando so spricht, wie er spricht, es sei denn, er kennt die päpstlichen Bullen. Darüber hinaus könnte dieselbe Verallgemeinerung von einem Reisenden stammen, der keinerlei Wissen über Kolonialgeschichte besitzt. Das relevante Vorwissen in diesem Fall ist, dass in einem bestimmten Land die meisten Menschen dieselbe Sprache sprechen; Fernando dagegen wird nicht als Name aller Brasilianer angenommen, weil diese Regelmäßigkeit für Namen nicht gilt. Analog dazu kann auch der Physikstudent im Grundstudium kaum die verschiedenen Werte erklären, die er für die Leitfähigkeit und die Dichte von Kupfer ermittelt hat. Er weiß dagegen, dass das Material, aus dem ein Objekt besteht, und seine Temperatur zusammen seine Leitfähigkeit bestimmen. In jedem Fall beeinflusst das Vorwissen *Hintergrund* die **Relevanz** einer Menge von Funktionsmerkmalen für das Zielprädikat. Dieses Wissen erlaubt es dem Agenten *zusammen mit den Beobachtungen*, eine neue, allgemeine Regel abzuleiten, die die Beobachtungen erklärt:

$$Hypothese \wedge Beschreibungen \models Klassifizierungen$$
$$Hintergrund \wedge Beschreibungen \wedge Klassifizierungen \models Hypothese. \quad (19.4)$$

Wir bezeichnen diese Art der Verallgemeinerung auch als **relevanzbasiertes Lernen** (**RBL**) (obwohl der Name noch nicht als Standard gilt). Beachten Sie, dass RBL den Inhalt der Beobachtungen nutzt, aber keine Hypothesen erzeugt, die über den logischen Inhalt des Hintergrundwissens und der Beobachtungen hinausgehen. Es handelt sich dabei um eine *deduktive* Form des Lernens, die nicht in der Lage ist, neues Wissen von Grund auf zu erstellen.

Im Fall des Medizinstudenten, der den Experten beobachtet, gehen wir davon aus, dass das Vorwissen des Studenten ausreichend ist, um die Krankheit D des Patienten von dessen Symptomen abzuleiten. Das ist jedoch nicht ausreichend, um die Tatsache zu erklären, dass der Arzt eine bestimmte Medizin M verordnet. Der Student muss eine weitere Regel anbringen, nämlich dass M im Allgemeinen gegen D wirkt. Mit dieser Regel und seinem Vorwissen kann der Student jetzt erklären, warum der Experte in diesem besonderen Fall M verordnet. Wir können dieses Beispiel verallgemeinern und erhalten die folgende Bedingung für die logische Konsequenz:

$$Hintergrund \wedge Hypothese \wedge Beschreibungen \models Klassifizierungen. \quad (19.5)$$

Tipp

Das bedeutet, *das Hintergrundwissen und die neue Hypothese werden kombiniert, um die Beispiele zu erklären*. Wie beim rein induktiven Lernen sollte der Lernalgorithmus die Hypothesen vorschlagen, die so einfach wie möglich und dennoch mit dieser Bedingung konsistent sind. Algorithmen, die Bedingung (19.5) erfüllen, werden auch als **wissensbasiertes induktives Lernen** oder **KBIL-Algorithmen** (Knowledge-Based Inductive Learning) bezeichnet.

KBIL-Algorithmen, die detailliert in *Abschnitt 19.5* erklärt werden, wurden hauptsächlich im Bereich der **induktiven logischen Programmierung** (**ILP**) untersucht. In ILP-Systemen spielt das Vorwissen zwei Schlüsselrollen für die Reduzierung der Lernkomplexität:

1. Weil jede erzeugte Hypothese mit dem Vorwissen sowie mit den neuen Beobachtungen konsistent sein muss, wird die effektive Größe des Hypothesenraumes reduziert, sodass sie nur noch die Theorien enthält, die mit dem konsistent sind, was bereits bekannt ist.

2. Für jede Menge an Beobachtungen kann die Größe der Hypothese für den Aufbau einer Erklärung für die Beobachtungen stark reduziert werden, weil das Vorwissen zur Verfügung steht, um die neuen Regeln für die Erklärung der Beobachtung zu unterstützen. Je kleiner die Hypothese ist, desto leichter ist sie zu finden.

ILP-Systeme erlauben nicht nur die Verwendung von Vorwissen bei der Induktion, sondern sie können auch Hypothesen in allgemeiner Logik erster Stufe formulieren statt in der eingeschränkten attributbasierten Sprache, die wir in *Kapitel 18* gezeigt haben. Das bedeutet, sie können in Umgebungen lernen, die von einfacheren Systemen nicht verstanden werden können.

19.3 Erklärungsbasiertes Lernen

Erklärungsbasiertes Lernen ist eine Methode, allgemeine Regeln aus einzelnen Beobachtungen zu extrahieren. Als Beispiel betrachten Sie das Problem, algebraische Ausdrücke zu differenzieren und zu vereinfachen (Übung 9.18). Wenn wir einen Ausdruck wie X^2 für X differenzieren, erhalten wir $2X$. (Beachten Sie, dass wir einen Großbuchstaben für die arithmetische Unbekannte X verwenden, um sie von der logischen Variablen x zu unterscheiden.) In einem logischen Inferenzsystem könnte das Ziel als ASK(*Ableitung*(X^2, X) = d, *KB*) ausgedrückt werden, mit der Lösung $d = 2X$.

Jeder, der sich mit der Differentialrechnung auskennt, erkennt diese Lösung „beim Ansehen“, weil er erfahren in der Lösung solcher Probleme ist. Ein Schüler, der solche Probleme zum ersten Mal sieht, oder ein Programm ohne entsprechende Erfahrung hat es da sehr viel schwerer. Die Anwendung der Standardregeln der Differentiation ergeben irgendwann den Ausdruck $1 \times (2 \times (X^{(2-1)}))$, der zu $2X$ vereinfacht wird. In der Implementierung der Logikprogrammierung des Autors sind dafür 136 Beweisschritte erforderlich, von denen 99 Sackgassenverzweigungen im Beweis sind. Nach dieser Erfahrung würde es uns gefallen, wenn das Programm dasselbe Problem beim nächsten Auftreten schneller lösen könnte.

Die Technik der **Memoisation** wurde in der Informatik lange angewandt, um Programme zu beschleunigen, indem die Ergebnisse von Berechnungen gespeichert wurden. Die grundlegende Idee von Memo-Funktionen ist, eine Datenbank von Eingabe/

Ausgabe-Paaren aufzubauen; wird die Funktion aufgerufen, prüft sie zuerst anhand der Datenbank, ob sie es vermeiden kann, das Problem von Grund auf neu zu lösen. Erklärungsbasiertes Lernen führt dies einen guten Schritt weiter, indem es *allgemeine* Regeln erzeugt, die eine ganze Klasse von Fällen abdecken. Im Fall der Differentiation würde sich die Memoisation merken, dass die Ableitung von X^2 nach X gleich $2X$ ist, es jedoch dem Agenten überlassen, die Ableitung von Z^2 nach Z von Grund auf neu zu berechnen. Wir möchten in der Lage sein, die allgemeine Regel zu extrahieren, dass für jede arithmetische Unbekannte u die Ableitung von u^2 für u gleich $2u$ ist. (Es lässt sich auch eine allgemeinere Regel für u^n erzeugen, aber das hier gezeigte Beispiel ist ausreichend, um das Konzept zu erklären.) Logisch wird dies durch die folgende Regel ausgedrückt:

$$ArithmetischeUnbekannte(u) \Rightarrow Ableitung(u^2, u) = 2u.$$

Wenn die Wissensbasis eine solche Regel enthält, kann jeder neue Fall, bei dem es sich um eine Instanz dieser Regel handelt, unmittelbar gelöst werden.

Tipp

Das ist natürlich nur ein einfaches Beispiel für ein sehr allgemeines Phänomen. Nachdem etwas verstanden wurde, kann es verallgemeinert und in anderen Situationen wieder verwendet werden. Es wird zu einem „offensichtlichen" Schritt und kann dann als Baustein für die Lösung von noch komplexeren Problemen verwendet werden. Alfred North Whitehead (1911), zusammen mit Bertrand Russel Autor von *Principia Mathematica*, schrieb: „*Die Zivilisation macht Fortschritte, indem sie die Anzahl wichtiger Operationen erweitert, die wir ausführen können, ohne darüber nachzudenken*", wobei er vielleicht selbst EBL auf sein Verständnis für Ereignisse wie etwa die Entdeckung von Zog anwendet. Wenn Sie das grundlegende Konzept des Differentiationsbeispiels verstanden haben, versucht Ihr Gehirn bereits angestrengt, die allgemeinen Prinzipien des erklärungsbasierten Lernens daraus zu extrahieren. Beachten Sie, dass Sie EBL *noch nicht* erfunden hatten, bevor Sie das Beispiel gesehen haben. Wie die Höhlenmenschen, die Zog beobachten, brauchten Sie (und wir) ein Beispiel, bevor wir die grundlegenden Prinzipien erzeugen konnten. Aus diesem Grund ist es viel einfacher, zu *erklären, warum* etwas eine gute Idee ist, als die Idee überhaupt erst zu entwickeln.

19.3.1 Allgemeine Regeln aus Beispielen extrahieren

Die grundlegende Idee hinter EBL ist, eine Erklärung der Beobachtung unter Verwendung von Vorwissen zu erzeugen und dann eine Definition der Klasse aller Fälle einzurichten, für die dieselbe Erklärungsstruktur genutzt werden kann. Diese Definition bietet die Grundlage für eine Regel, die alle Fälle in der Klasse abdeckt. Die „Erklärung" kann ein logischer Beweis sein, aber allgemeiner kann es sich um einen beliebigen Inferenz- oder problemlösenden Prozess handeln, dessen Schritte wohldefiniert sind. Wichtig ist vor allem, in der Lage zu sein, die erforderlichen Bedingungen für dieselben Schritte in einem anderen Fall zu identifizieren.

Wir werden jetzt als Inferenzsystem den einfachen Rückwärtsverkettungs-Theorembeweiser verwenden, den wir in *Kapitel 9* vorgestellt haben. Der Beweisbaum für $Ableitung(X^2, X) = 2X$ ist zu groß, um ihn als Beispiel zu verwenden, deshalb verwenden wir ein einfacheres Problem, um die Verallgemeinerungsmethode zu veranschaulichen. Angenommen, wir wollen $1 \times (0 + X)$ vereinfachen. Die Wissensbasis enthält die folgenden Regeln:

$Umschreiben(u, v) \wedge Vereinfachen(v, w) \Rightarrow Vereinfachen(u, w)$
$Elementar(u) \Rightarrow Vereinfachen(u, u)$
$ArithmetischeUnbekannte(u) \Rightarrow Elementar(u)$
$Zahl(u) \Rightarrow Elementar(u)$
$Umschreiben(1 \times u, u)$
$Umschreiben(0 + u, u)$
⋮

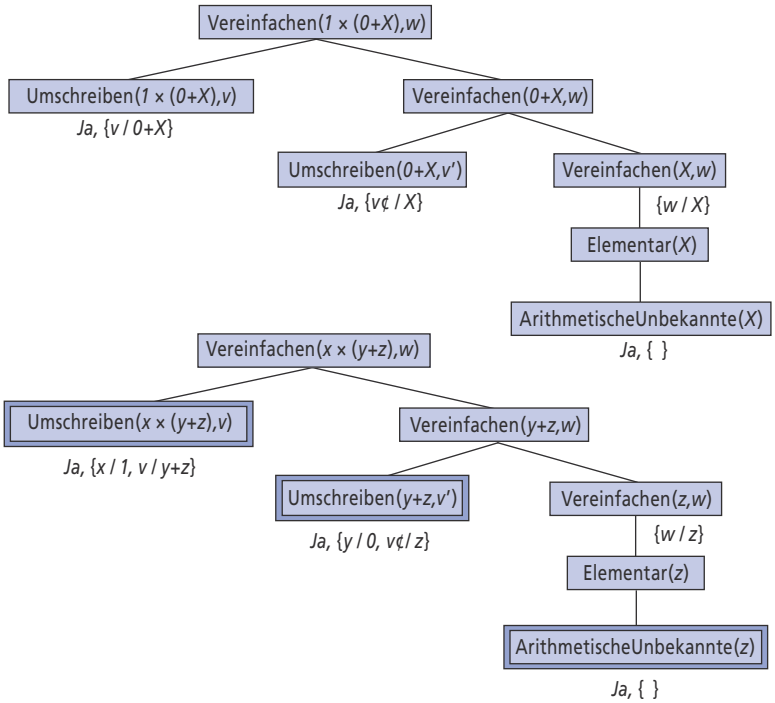


Abbildung 19.7: Beweisbäume für das Vereinfachungsproblem. Der erste Baum zeigt den Beweis für die ursprüngliche Probleminstanz, aus der wir Folgendes ableiten können: $ArithmetischeUnbekannte(z) \Rightarrow Vereinfachen(1 \times (0 + z), z)$. Der zweite Beweisbaum zeigt den Beweis für eine Probleminstanz, wobei alle Konstanten durch Variablen ersetzt wurden, woraus wir eine Vielzahl anderer Regeln ableiten können.

▶ Abbildung 19.7 zeigt in der oberen Hälfte den Beweis, dass die Antwort gleich X ist. Die EBL-Methode erzeugt zwei Beweisbäume gleichzeitig. Der zweite Beweisbaum verwendet ein *mit Variablen unterlegtes* Ziel, wobei die Konstanten aus dem ursprünglichen Ziel durch Variablen ersetzt wurden. Während des ursprünglichen Beweises wird der variablenunterlegte Beweis schrittweise ausgeführt, *wobei genau dieselben Regeln angewendet werden*. Das könnte dazu führen, dass einige der Variablen instantiiert werden. Um beispielsweise die Regel $Umschreiben(1 \times u, u)$ zu verwenden, muss die Variable x im Unterziel $Umschreiben(x \times (y+z), v)$ an 1 gebunden werden. Analog dazu muss y im Unterziel $Umschreiben(y+z, v')$ an 0 gebunden werden, damit wir die Regel $Umschreiben(0 + u, u)$ anwenden können. Nachdem wir den verallgemeinerten Beweisbaum haben, nehmen wir die Blätter (mit den entsprechenden Begrenzungen) und bilden eine allgemeine Regel für das Zielprädikat:

$$Umschreiben(1 \times (0 + z), 0 + z) \wedge Umschreiben(0 + z, z) \wedge ArithmetischeUnbekannte(z) \\ \Rightarrow Vereinfachen(1 \times (0 + z), z)$$

Beachten Sie, dass die beiden Bedingungen auf der linken Seite *unabhängig von dem Wert von z* immer zutreffen. Wir können sie also aus der Regel entfernen und erhalten damit:

$$ArithmetischeUnbekannte(z) \Rightarrow Vereinfachen(1 \times (0 + z), z).$$

Im Allgemeinen können Bedingungen aus der fertigen Regel entfernt werden, wenn sie keine Beschränkungen für die Variablen auf der rechten Seite der Regel darstellen, weil die resultierende Regel dann immer noch wahr und überdies effizienter ist. Beachten Sie, dass wir die Bedingung *ArithmetischeUnbekannte*(*z*) nicht fallen lassen können, weil nicht alle möglichen Werte von *z* arithmetische Unbekannte sind. Andere Werte als arithmetische Unbekannte benötigen vielleicht andere Formen der Vereinfachung: Ist *z* beispielsweise gleich 2×3, dann ist die korrekte Vereinfachung von $1 \times (0 + (2 \times 3)$ gleich 6 und nicht 2×3.

Zusammenfassend können wir also sagen, der EBL-Prozess funktioniert wie folgt:

1. Für ein gegebenes Beispiel konstruieren wir einen Beweis, dass das Zielprädikat unter Verwendung des verfügbaren Hintergrundwissens auf das Beispiel anzuwenden ist.
2. Parallel dazu erzeugen wir einen verallgemeinerten Beweisbaum für das parametrisierte Ziel unter Verwendung derselben Inferenzschritte wie im Originalbeweis.
3. Wir bauen eine neue Regel auf, deren linke Seite aus den Blättern des Beweisbaumes besteht und deren rechte Seite das parametrisierte Ziel ist (nachdem die erforderlichen Bindungen aus dem verallgemeinerten Beweis angewendet wurden).
4. Wir verwerfen alle Bedingungen, die belanglos für die Werte der Variablen im Ziel sind.

19.3.2 Effizienzverbesserung

Der verallgemeinerte Beweisbaum in Abbildung 19.7 ergibt eigentlich mehr als eine verallgemeinerte Regel. Wenn wir beispielsweise das Wachstum des rechten Zweiges im Beweisbaum beenden (oder **kürzen**), sobald der Schritt *Elementar* erreicht ist, erhalten wir die Regel

$$Elementar(z) \Rightarrow Vereinfachen(1 \times (0 + z), z).$$

Diese Regel ist gültig, aber *allgemeiner* als die Regel unter Verwendung von *ArithmetischeUnbekannte*, weil sie Fälle abdeckt, in denen *z* eine Zahl ist. Wir können eine noch allgemeinere Regel extrahieren, indem wir nach dem Schritt $Vereinfachen(y + z, w)$ kürzen, womit wir die folgende Regel erhalten:

$$Vereinfachen(y + z, w) \Rightarrow Vereinfachen(1 \times (y + z), w).$$

Im Allgemeinen kann eine Regel von *jedem beliebigen Unterbaum* des verallgemeinerten Beweisbaumes aus extrahiert werden. Jetzt haben wir ein Problem: Welche dieser Regeln wählen wir aus?

Die Wahl, welche Regel wir erzeugen, ist eine Frage der Effizienz. Es gibt drei Faktoren bei der Analyse der Effizienzgewinne aus EBL:

1. Das Hinzufügen sehr vieler Regeln kann den Inferenzprozess verlangsamen, weil der Inferenzmechanismus diese Regeln auch für solche Fälle überprüfen muss, wo sie keine Lösung ergeben. Mit anderen Worten, es steigert den **Verzweigungsfaktor** im Suchraum.

2. Um den verlangsamten Inferenzprozess zu kompensieren, müssen die abgeleiteten Regeln eine wesentlich verbesserte Geschwindigkeit für die Fälle bieten, die sie abdecken. Diese Steigerungen entstehen hauptsächlich, weil die abgeleiteten Regeln Sackgassen vermeiden, die andernfalls eingeschlagen werden, aber auch, weil sie den eigentlichen Beweis verkürzen.

3. Abgeleitete Regeln sollten so allgemein wie möglich sein, sodass sie sich auf die größtmögliche Menge an Fällen beziehen.

Ein allgemeiner Ansatz, die Effizienz abgeleiteter Regeln sicherzustellen, ist es, auf der **Operationalität** jedes Unterzieles in der Regel zu bestehen. Ein Unterziel ist operational, wenn es „einfach" zu lösen ist. Beispielsweise ist das Unterziel *Elementar*(z) einfach zu lösen und benötigt höchstens zwei Schritte, während das Unterziel *Vereinfachen*($y + z$, w) zu beliebig viel Inferenz führen kann – abhängig von den Werten von y und z. Wenn bei jedem Konstruktionsschritt des verallgemeinerten Beweises ein Test auf Operationalität ausgeführt wird, können wir den Rest eines Zweiges kürzen, sobald ein operationales Unterziel erreicht ist, und nur das operationale Unterziel als Konjunkt der neuen Regel beibehalten.

Leider gibt es zumeist einen Konflikt zwischen Operationalität und Allgemeinheit. Spezifischere Unterziele sind im Allgemeinen einfacher zu lösen, decken aber weniger Fälle ab. Außerdem kommt es bei der Operationalität auf den Grad an: Ein oder zwei Schritte sind definitiv operational, aber was ist mit 10 oder 100? Schließlich sind die Kosten für die Lösung eines bestimmten Unterzieles davon abhängig, welche anderen Regeln in der Wissensbasis zur Verfügung stehen. Sie können beim Hinzufügen weiterer Regeln steigen oder sinken. EBL-Systeme stehen also wirklich einem sehr komplexen Optimierungsproblem gegenüber, wenn sie versuchen, die Effizienz einer bestimmten Ausgangswissensbasis zu maximieren. Manchmal ist es möglich, ein mathematisches Modell der Auswirkung auf die Gesamteffizienz beim Hinzufügen einer Regel abzuleiten und dieses Modell zu nutzen, um die beste Regel auszuwählen, die hinzugefügt werden soll. Die Analyse kann jedoch sehr kompliziert werden, insbesondere wenn rekursive Regeln vorkommen. Ein vielversprechender Ansatz ist, das Effizienzproblem empirisch anzugehen, indem man einfach mehrere Regeln hinzufügt und dann überprüft, welche davon praktisch sind und das Ganze tatsächlich beschleunigen.

Tipp

Die empirische Analyse der Effizienz ist das Herz von EBL. Was wir leger als „Effizienz einer gegebenen Wissensbasis" bezeichnet haben, ist letztlich die durchschnittliche Komplexität einer Verteilung von Problemen. *Durch die Verallgemeinerung von Beispielproblemen aus der Vergangenheit macht EBL die Wissensbasis effizienter für Probleme, die in der Zukunft berechtigt zu erwarten sind.* Das funktioniert, solange die Verteilung der Beispiele aus der Vergangenheit in etwa dieselbe ist wie für zukünftige Beispiele – dieselbe Annahme, die für das PAC-Lernen in *Abschnitt 18.5* verwendet wurde. Wenn das EBL-System sorgfältig ausgelegt ist, können wesentliche Beschleunigungen erzielt werden. Beispielsweise konnte ein sehr großes, auf Prolog basierendes

Sprachsystem für die Direktübersetzung zwischen Schwedisch und Englisch nur deshalb Echtzeitleistung erreichen, weil EBL auf den Parsingprozess angewendet wurde (Samuelsson und Rayner, 1991).

19.4 Lernen mit Relevanzinformation

Unser Brasilienreisender scheint in der Lage zu sein, eine zuverlässige Verallgemeinerung zu der von anderen Brasilianern gesprochenen Sprache zu treffen. Die Inferenz ist durch sein Hintergrundwissen gültig, nämlich dass Menschen in einem bestimmten Land (für gewöhnlich) dieselbe Sprache sprechen. Wir können dies wie folgt in Logik erster Stufe ausdrücken:[2]

$$\mathit{Nationalität}(x, n) \wedge \mathit{Nationalität}(y, n) \wedge \mathit{Sprache}(x, l) \Rightarrow \mathit{Sprache}(y, l). \qquad (19.6)$$

(Wörtliche Übersetzung: „Wenn x und y dieselbe Nationalität n haben und x die Sprache l spricht, dann spricht auch y die Sprache l.“) Man kann ganz einfach zeigen, dass aus diesem Satz und der Beobachtung

$$\mathit{Nationalität}(\mathit{Fernando}, \mathit{Brasilianisch}) \wedge \mathit{Sprache}(\mathit{Fernando}, \mathit{Portugiesisch})$$

der folgende Schluss logisch folgt (siehe Übung 19.1):

$$\mathit{Nationalität}(x, \mathit{Brasilianisch}) \Rightarrow \mathit{Sprache}(x, \mathit{Portugiesisch}).$$

Sätze wie (19.6) drücken eine strenge Form der Relevanz aus: Für eine gegebene Nationalität ist die Sprache vollständig feststehend. (Anders ausgedrückt: Die Sprache ist eine Funktion der Nationalität.) Diese Sätze werden auch als **funktionale Abhängigkeiten** oder **Bestimmtheiten** bezeichnet. Sie treten in bestimmten Anwendungen (z.B. bei der Definition von Datenbankentwürfen) so häufig auf, dass eine spezielle Syntax für ihre Darstellung verwendet wird. Wir übernehmen die Notation von Davies (1985):

$$\mathit{Nationalität}(x, n) \succ \mathit{Sprache}(x, l).$$

Wie üblich ist dies einfach syntaktischer Zucker, verdeutlicht aber, dass die Bestimmtheit tatsächlich eine Beziehung zwischen den Prädikaten ist: Die Nationalität bestimmt die Sprache. Die relevanten Eigenschaften für Leitfähigkeit und Dichte können auf ähnliche Weise ausgedrückt werden:

$$\mathit{Material}(x, m) \wedge \mathit{Temperatur}(x, t) \succ \mathit{Leitfähigkeit}(x, p);$$
$$\mathit{Material}(x, m) \wedge \mathit{Temperatur}(x, t) \succ \mathit{Dichte}(x, d).$$

Die entsprechenden Verallgemeinerungen folgen logisch aus den Bestimmtheiten und Beobachtungen.

19.4.1 Den Hypothesenraum festlegen

Tipp

Obwohl die Bestimmtheiten die allgemeinen Schlüsse zu allen Brasilianern oder zu allen Kupferstücken bei einer bestimmten Temperatur unterstützen, können sie natürlich keine allgemeine voraussagende Theorie für *alle* Nationalitäten oder für *alle* Temperaturen und Materialen aus einem einzigen Beispiel erzeugen. Ihre wichtigste Wir-

2 Der Einfachheit halber gehen wir davon aus, dass eine Person nur eine Sprache spricht. Offensichtlich müsste die Regel für Länder wie die Schweiz oder Indien erweitert werden.

kung kann als Begrenzung des Hypothesenraumes betrachtet werden, den der lernende Agent auswerten muss. Bei der Vorhersage der Leitfähigkeit beispielsweise muss man nur Material und Temperatur berücksichtigen und kann Masse, Besitzer, Wochentag, den amtierenden Präsidenten usw. vernachlässigen. Hypothesen können natürlich Terme enthalten, die wiederum durch Material und Temperatur beeinflusst werden, wie beispielsweise Molekularstruktur, thermische Energie oder die Dichte der freien Elektronen. *Bestimmtheiten spezifizieren ein ausreichendes Grundvokabular, aus dem Hypothesen konstruiert werden können, die das Zielprädikat betreffen*. Diese Aussage kann bewiesen werden, indem gezeigt wird, dass eine Bestimmtheit logisch äquivalent mit einer Aussage ist, dass die korrekte Definition des Zielprädikates eine der Mengen aller Definitionen ist, die unter Verwendung der Prädikate auf der linken Seite der Bestimmung ausgedrückt werden können.

Intuitiv ist klar, dass eine Reduzierung der Hypothesenraumgröße es einfacher machen soll, das Zielprädikat zu lernen. Unter Verwendung der grundlegenden Ergebnisse der Computer-Lerntheorie (*Abschnitt 18.5*) können wir die möglichen Gewinne quantifizieren. Sie wissen, dass für boolesche Funktionen $\log(|\mathcal{H}|)$ Beispiele erforderlich sind, um zu einer sinnvollen Hypothese zu konvergieren, wobei $|\mathcal{H}|$ die Größe des Hypothesenraumes ist. Wenn der Lernende n boolesche Funktionsmerkmale hat, mit denen er Hypothesen erzeugt, dann ist beim Fehlen weiterer Bedingungen $|\mathcal{H}| = O(2^{2^n})$, die Anzahl der Beispiele ist also $O(2^n)$. Wenn die Bestimmtheit d Prädikate auf der linken Seite enthält, braucht der Lernende nur $O(2^d)$ Beispiele, eine Reduzierung um $O(2^{n-d})$.

19.4.2 Lernen und Verwenden von Relevanzinformation

Wie in der Einführung zu diesem Kapitel erwähnt, ist Vorwissen praktisch für das Lernen, aber auch Vorwissen muss gelernt werden. Um das relevanzbasierte Lernen wirklich vollständig zu beschreiben, müssen wir deshalb einen Lernalgorithmus für Bestimmtheiten bereitstellen. Der hier vorgestellte Lernalgorithmus basiert auf einem einfachen Versuch, die einfachste Bestimmtheit zu ermitteln, die mit den Beobachtungen konsistent ist. Eine Bestimmtheit $P \succ Q$ sagt, wenn Beispiele mit P übereinstimmen, dann müssen sie auch mit Q übereinstimmen. Eine Bestimmtheit ist deshalb mit einer Beispielmenge konsistent, wenn jedes Paar, das mit den Prädikaten auf der linken Seite übereinstimmt, auch mit dem Zielprädikat übereinstimmt. Angenommen, wir haben die folgenden Beispiele für Leitfähigkeitsmessungen bei Materialstichproben:

Stichprobe	Masse	Temperatur	Material	Größe	Leitfähigkeit
S1	12	26	Kupfer	3	0,59
S1	12	100	Kupfer	3	0,57
S2	24	26	Kupfer	6	0,59
S3	12	26	Blei	2	0,05
S3	12	100	Blei	2	0,04
S4	24	26	Blei	4	0,05

Die minimal konsistente Bestimmtheit ist *Material* $\wedge$ *Temperatur* $\succ$ *Leitfähigkeit*. Es gibt eine nicht minimale, aber konsistente Bestimmtheit, nämlich *Masse* $\wedge$ *Größe* $\wedge$

Temperatur ≻ *Leitfähigkeit*. Dies ist konsistent mit den Beispielen, weil Masse und Größe die Dichte bestimmen und wir in unserer Datenmenge keine unterschiedlichen Materialien mit derselben Dichte haben. Wie üblich bräuchten wir eine größere Stichprobenmenge, um fast korrekte Hypothesen zu eliminieren.

Es gibt mehrere mögliche Algorithmen, um minimal konsistente Bestimmtheiten zu finden. Der offensichtlichste Ansatz ist, eine Suche durch den Raum der Bestimmtheiten auszuführen und alle Bestimmtheiten mit einem Prädikat, zwei Prädikaten usw. zu überprüfen, bis eine konsistente Bestimmtheit gefunden ist. Wir gehen von einer einfachen attributbasierten Repräsentation aus, wie der für das Entscheidungsbaumlernen in *Kapitel 18* benutzten. Eine Bestimmtheit d wird durch die Menge der Attribute auf der linken Seite dargestellt, weil das Zielprädikat als feststehend angenommen wird. ▶ Abbildung 19.8 zeigt den grundlegenden Algorithmus.

```
function MINIMAL-CONSISTENT-DET(E, A) returns eine Attributmenge
   inputs: E, eine Beispielmenge
           A, eine Attributmenge der Größe n

   for i = 0 to n do
      for each Untermenge A_i von A der Größe i do
         if CONSISTENT-DET?(A_i, E) then return A_i

---------------------------------------------------------------

function CONSISTENT-DET?(A, E) returns einen Wahrheitswert
   inputs: A, eine Attributmenge
           E, eine Beispielmenge
   local variables: H, eine Hashtabelle

   for each Beispiel e in E do
      if ein Beispiel in H hat dieselben Werte wie e für die Attribute A
         aber eine andere Klassifizierung then return false
      die Klasse von e in H speichern, indiziert durch die Werte für die
      Attribute A des Beispiels e
   return true
```

Abbildung 19.8: Ein Algorithmus für die Ermittlung einer minimal konsistenten Bestimmtheit

Die Zeitkomplexität dieses Algorithmus ist von der Größe der kleinsten konsistenten Bestimmtheit abhängig. Angenommen, diese Bestimmtheit hat p Attribute aus insgesamt n Attributen. Der Algorithmus ist erst dann erfolgreich, nachdem er die Untermengen von A der Größe p durchsucht hat. Es gibt $\binom{n}{p} = O(n^p)$ solcher Untermengen; der Algorithmus ist also exponentiell in der Größe der minimalen Bestimmtheit. Es zeigt sich, dass das Problem NP-vollständig ist; wir können also nicht erwarten, im allgemeinen Fall eine bessere Leistung zu erhalten. In den meisten Domänen gibt es jedoch genügend lokale Struktur (eine Definition lokal strukturierter Domänen finden Sie in *Kapitel 14*), sodass p klein ist.

Mit einem Algorithmus zum Lernen von Bestimmtheiten hat ein lernender Agent die Möglichkeit, eine minimale Hypothese zu erzeugen, innerhalb derer er das Zielprädikat lernt. Beispielsweise können wir MINIMAL-CONSISTENT-DET mit dem Algorithmus DECISION-TREE-LEARNING kombinieren. Damit erhalten wir einen relevanzbasierten Entscheidungsbaum-Lernalgorithmus RBDTL, der zuerst eine minimale Menge der relevanten Attribute identifiziert und diese Menge dann dem Entscheidungsbaum-Algorithmus zum Lernen übergibt. Anders als DECISION-TREE-LEARNING lernt RBDTL und verwendet gleich-

zeitig Relevanzinformation, um seinen Hypothesenraum zu minimieren. Wir erwarten, dass RBDTL schneller als DECISION-TREE-LEARNING lernt, und das ist tatsächlich der Fall. ▶ Abbildung 19.9 zeigt die Lernleistung für die beiden Algorithmen zu zufällig erzeugten Daten für eine Funktion, die nur von 5 aus 16 Attributen abhängig ist. Offensichtlich zeigt RBDTL in Fällen, wo alle verfügbaren Attribute relevant sind, keinen Vorteil.

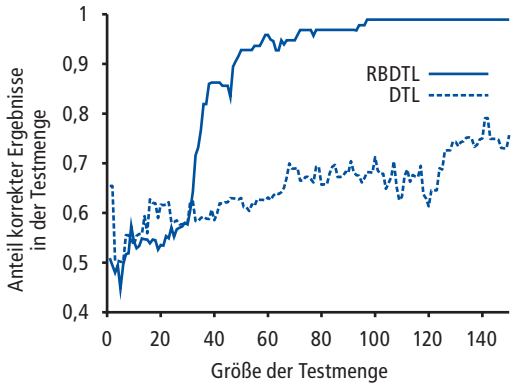


Abbildung 19.9: Ein Leistungsvergleich zwischen RBDTL und DECISION-TREE-LEARNING für zufällig erzeugte Daten für eine Zielfunktion, die von nur 5 aus 16 Attributen abhängig ist

Dieser Abschnitt konnte nur die Oberfläche des Gebietes der **deklarativen Bias** (Vorzugskriterium) berühren. Wir wollten damit zeigen, wie Vorwissen genutzt werden kann, um den geeigneten Hypothesenraum zu identifizieren, innerhalb dessen wir nach der richtigen Zieldefinition suchen. Es gibt viele unbeantwortete Fragen:

- Wie können die Algorithmen erweitert werden, um Rauschen zu verarbeiten?
- Können wir stetigwertige Variablen verarbeiten?
- Wie können neben Bestimmtheiten auch andere Arten von Vorwissen genutzt werden?
- Wie können die Algorithmen verallgemeinert werden, um Theorien der ersten Stufe abzudecken und nicht nur eine attributbasierte Repräsentation?

Einige dieser Fragen werden im nächsten Abschnitt aufgegriffen.

19.5 Induktive logische Programmierung

Induktive logische Programmierung (ILP) kombiniert induktive Methoden mit der Leistung von Repräsentationen der ersten Stufe und konzentriert sich dabei insbesondere auf die Repräsentation von Theorien als Logikprogramme.[3] Sie ist aus drei Gründen sehr beliebt. Erstens bietet ILP einen strengen Ansatz für das allgemeine wissensbasierte induktive Lernproblem. Zweitens bietet sie vollständige Algorithmen für die Ableitung allgemeiner Theorien erster Stufe aus Beispielen, die deshalb erfolgreich in Domänen lernen können, wo attributbasierte Algorithmen schwierig anzuwenden sind. Ein Beispiel dafür ist, zu lernen, wie sich Proteinstrukturen entfalten (▶ Abbildung 19.10). Die

3 Der Leser sei an dieser Stelle auf *Kapitel 7* verwiesen, wo die grundlegenden Konzepte beschrieben werden, wie beispielsweise Horn-Klauseln, konjunktive Normalform, Unifikation und Resolution.

dreidimensionale Konfiguration eines Proteinmoleküls kann nicht sinnvoll durch eine Attributmenge dargestellt werden, weil die Konfiguration inhärent auf den *Beziehungen* zwischen den Objekten basiert und nicht auf den Attributen eines einzigen Objektes. Die Logik erster Stufe ist eine geeignete Sprache für die Beschreibung der Beziehungen. Drittens erzeugt die induktive logische Programmierung Hypothesen, die für den Menschen (relativ) einfach zu lesen sind. Beispielsweise kann die deutsche Übersetzung in ▶ Abbildung 19.10 von Biologen überprüft und kritisiert werden. Das bedeutet, Systeme für induktive logische Programmierung können am wissenschaftlichen Zyklus von Experiment, Hypothesenerstellung, Debatte und Widerlegung teilnehmen. Eine solche Teilnahme ist nicht für Systeme möglich, die „Blackbox"-Klassifizierer erzeugen, wie etwa neuronale Netze.

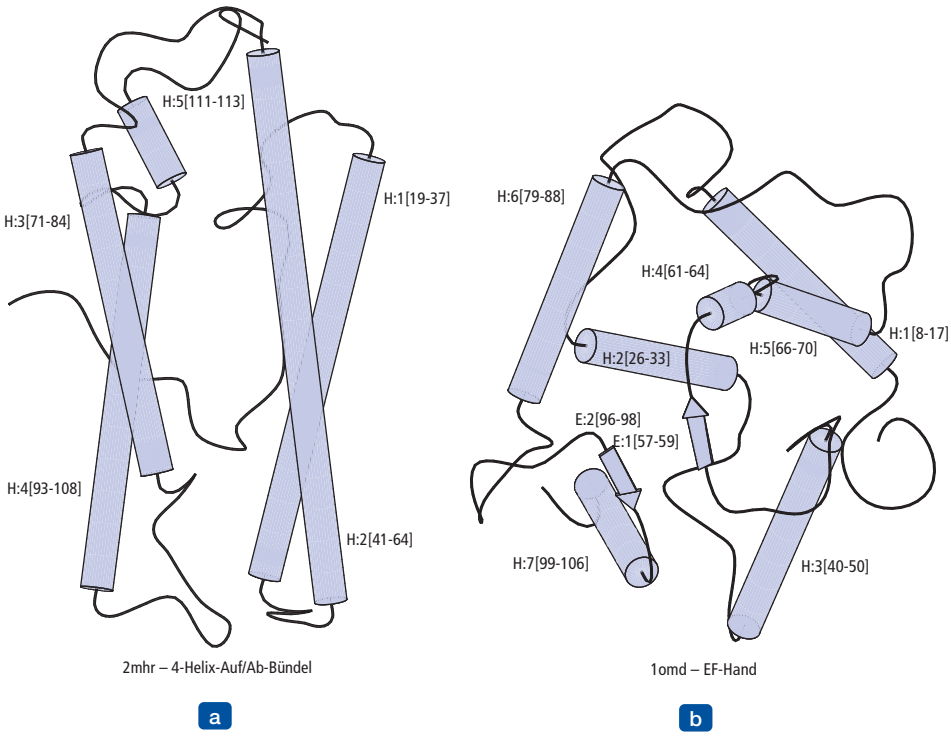


Abbildung 19.10: (a) und (b) zeigen positive und negative Beispiele des „4-Helix-Auf/Ab-Bündel"-Konzeptes der Proteinentfaltung. Jede Beispielstruktur ist in einem logischen Ausdruck von etwa 100 Konjunkten kodiert, wie etwa $GesamtLänge(D2mhr, 118) \wedge AnzahlHelix(D2mhr, 6) \wedge \ldots$ Aus diesen Beschreibungen sowie aus Klassifizierungen wie $Entfalten(\text{FOUR-HELICAL-UP-AND-DOWN-BUNDLE}, D2mhr)$ hat das System PROGOL (Muggleton, 1995) für induktive logische Programmierung die folgende Regel gelernt:

$$Entfalten(\text{FOUR-HELICAL-UP-AND-DOWN-BUNDLE}, p) \Leftarrow$$
$$Helix(p, h_1) \wedge Länge(h_1, \text{HIGH}) \wedge Position(p, h_1, n)$$
$$\wedge\ (1 \leq n \leq 3) \wedge Benachbart(p, h_1, h_2) \wedge Helix(p, h_2).$$

Diese Art Regel könnte nicht durch einen attributbasierten Mechanismus, wie wir ihn in den vorigen Kapiteln gesehen haben, gelernt werden – sie ließe sich noch nicht einmal damit darstellen. Die Regel kann man wie folgt ins Deutsche übersetzen: „Das Protein p hat die Entfaltungsklasse ‚4-Helix-Auf/Ab-Bündel', wenn sie eine lange Helix h_1 an einer sekundären Strukturposition zwischen 1 und 3 besitzt und h_1 sich neben einer zweiten Helix befindet."

19.5.1 Ein Beispiel

Aus Gleichung (19.5) wissen Sie, dass das allgemeine wissensbasierte Induktionsproblem ist, die Bedingung der logischen Konsequenz

$$Hintergrund \wedge Hypothese \wedge Beschreibungen \models Klassifizierungen$$

aus der unbekannten *Hypothese* zu „lösen“, wobei das *Hintergrund*wissen und die durch *Beschreibungen* und *Klassifizierungen* beschriebenen Beispiele bekannt sind. Um dies zu verdeutlichen, verwenden wir das Problem, Familienverhältnisse aus Beispielen zu lernen. Die Beschreibungen bestehen aus einem erweiterten Stammbaum, der mithilfe der Relationen *Mutter*, *Vater* und *Verheiratet* sowie der Eigenschaften *Männlich* und *Weiblich* dargestellt ist. Als Beispiel verwenden wir den Stammbaum aus *Übung 8.15*, der hier in ▶ Abbildung 19.11 noch einmal gezeigt ist. Die entsprechenden Beschreibungen lauten wie folgt:

Vater(*Philip*, *Charles*)
Mutter(*Mum*, *Margaret*)
Verheiratet(*Diana*, *Charles*)
Männlich(*Philip*)
Weiblich(*Beatrice*)

Vater(*Philip*, *Anne*)...
Mutter(*Mum*, *Elizabeth*)...
Verheiratet(*Elizabeth*, *Philip*)...
Männlich(*Charles*)...
Weiblich(*Margaret*)...

Die Sätze in *Klassifizierungen* sind von dem zu lernenden Zielkonzept abhängig. Wir könnten beispielsweise *Großeltern*, *Schwager* oder *Vorfahre* lernen. Für *Großeltern* enthält die vollständige Menge der *Klassifizierungen* 20 x 20 = 400 Konjunkte der Form

Großeltern(*Mum*, *Charles*)
¬*Großeltern*(*Mum*, *Harry*)

Großeltern(*Elizabeth*, *Beatrice*)...
¬*Großeltern*(*Spencer*, *Peter*)...

Wir könnten natürlich auch von einer Untermenge dieser vollständigen Menge lernen.

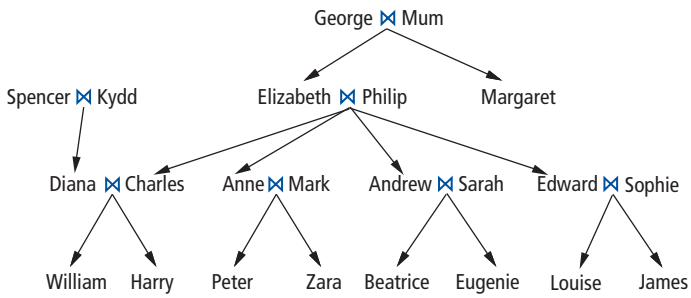


Abbildung 19.11: Ein typischer Stammbaum.

Das Ziel eines induktiven Lernprogramms ist, eine Menge von Sätzen für die *Hypothese* einzurichten, sodass die Bedingung der logischen Konsequenz erfüllt ist. Nehmen wir an, der Agent hat kein Hintergrundwissen: *Hintergrund* ist leer. Die einzige mögliche Lösung für *Hypothese* sieht dann wie folgt aus:

$$\begin{aligned} Großeltern(x, y) \Leftrightarrow\ & [\exists z\ \ Mutter(x, z) \wedge Mutter(z, y)] \\ \vee\ & [\exists z\ \ Mutter(x, z) \wedge Vater(z, y)] \\ \vee\ & [\exists z\ \ Vater(x, z) \wedge Mutter(z, y)] \\ \vee\ & [\exists z\ \ Vater(x, z) \wedge Vater(z, y)] \end{aligned}$$

Beachten Sie, dass ein attributbasierter Lernalgorithmus, wie etwa DECISION-TREE-LEARNING, keine Möglichkeit hat, dieses Problem zu lösen. Um *Großeltern* als Attribut auszudrücken (d.h. als unäres Prädikat), müssten wir *Paare* von Menschen in Objekten anlegen:

$$\mathit{Großeltern}(\langle \mathit{Mum}, \mathit{Charles}\rangle)\ldots$$

Dann bleiben wir bei dem Versuch stecken, die Beispielbeschreibungen darzustellen. Die einzigen möglichen Attribute sind Ungetüme wie etwa:

$$\mathit{ErstesElementIstMutterVonElizabeth}(\langle \mathit{Mum}, \mathit{Charles}\rangle).$$

Tipp

Die Definition von *Großeltern* im Hinblick auf diese Attribute wird einfach zu einer großen Disjunktion von Sonderfällen, die sich überhaupt nicht auf neue Beispiele verallgemeinern lassen. *Attributbasierte Lernalgorithmen sind nicht in der Lage, relationale Prädikate zu lernen.* Einer der wichtigsten Vorteile von ILP-Algorithmen ist deshalb die Möglichkeit, sie auf einen wesentlich breiteren Problembereich anzuwenden – auch auf relationale Probleme.

Der Leser hat bestimmt bemerkt, dass ein wenig Hintergrundwissen bei der Repräsentation der *Großeltern*-Definition hilfreich gewesen wäre. Würde zum Beispiel *Hintergrund* den Satz

$$\mathit{Eltern}(x, y) \quad \Leftrightarrow \quad [\mathit{Mutter}(x, y) \vee \mathit{Vater}(x, y)]$$

einbinden, würde die Definition von *Großeltern* reduziert auf

$$\mathit{Großeltern}(x, y) \Leftrightarrow [\exists z \quad \mathit{Eltern}(x, z) \wedge \mathit{Eltern}(z, y)].$$

Daran erkennen wir, wie Hintergrundwissen die Größe von Hypothesen für die Erklärung von Beobachtungen drastisch reduzieren kann.

Außerdem ist es möglich, dass ILP-Algorithmen neue Prädikate *erzeugen*, um den Ausdruck erklärender Hypothesen zu vereinfachen. Mit den zuvor gezeigten Beispieldaten ist es durchaus möglich für das ILP-Programm, ein zusätzliches Prädikat vorzuschlagen, das wir als „*Eltern*“ bezeichnen wollen, um die Definitionen der Zielprädikate zu vereinfachen. Algorithmen, die neue Prädikate erzeugen können, werden auch als **konstruktive Induktionsalgorithmen** bezeichnet. Offensichtlich ist die konstruktive Induktion ein notwendiger Bestandteil des Bildes kumulativen Lernens, das in der Einführung skizziert wurde. Es war eines der schwierigsten Probleme beim maschinellen Lernen, aber einige ILP-Techniken bieten effektive Mechanismen für seine Erzielung.

Im restlichen Kapitel betrachten wir die beiden wichtigsten Ansätze für die ILP. Der erste verwendet eine Verallgemeinerung der Entscheidungsbaummethoden, der zweite verwendet Techniken, die auf der Invertierung eines Resolutionsbeweises basieren.

19.5.2 Induktive Top-down-Lernmethoden

Der erste Ansatz für ILP beginnt mit einer sehr allgemeinen Regel und spezialisiert sie schrittweise, sodass sie mit den Daten übereinstimmt. Das ist im Wesentlichen dasselbe, was wir vom Entscheidungsbaumlernen her kennen, wo ein Entscheidungsbaum schrittweise anwächst, bis er mit den Beobachtungen konsistent ist. Für die ILP verwenden wir Literale der ersten Stufe statt Attribute und die Hypothese ist eine Menge von Klauseln statt eines Entscheidungsbaumes. Dieser Abschnitt beschreibt FOIL (Quinlan, 1990), eines der ersten ILP-Programme.

Angenommen, wir wollen eine Definition des Prädikates *Großvater*(x, y) lernen und verwenden dazu dieselben Familiendaten wie zuvor. Wie beim Entscheidungsbaumlernen können wir in positive und negative Beispiele unterteilen. Positive Beispiele sind:

$$\langle George, Anne\rangle, \langle Philip, Peter\rangle, \langle Spencer, Harry\rangle, \ldots$$

Negative Beispiele sind:

$$\langle George, Elizabeth\rangle, \langle Harry, Zara\rangle, \langle Charles, Philip\rangle, \ldots$$

Beachten Sie, dass jedes Beispiel ein *Paar* von Objekten ist, weil *Großvater* ein binäres Prädikat ist. Insgesamt gibt es zwölf positive Beispiele im Stammbaum und 388 negative Beispiele (alle anderen Paare).

FOIL konstruiert eine Menge von Klauseln, jeweils mit *Großvater*(x, y) als Kopf. Die Klauseln müssen die zwölf positiven Beispiele als Instanzen der *Großvater*(x, y)-Beziehung klassifizieren und gleichzeitig die 388 negativen Beispiele ausschließen. Bei den Klauseln handelt es sich um Horn-Klauseln mit der Erweiterung, dass negierte Literale im Rumpf einer Klausel zulässig sind und mithilfe der Negation als Fehler interpretiert werden, wie es in Prolog üblich ist. Die anfängliche Klausel hat einen leeren Rumpf:

$$\Rightarrow Großvater(x, y).$$

Diese Klausel klassifiziert jedes Beispiel als positiv, sie muss also spezialisiert werden. Dazu fügen wir schrittweise Literale auf der linken Seite ein. Hier drei potenzielle Ergänzungen:

$$Vater(x, y) \Rightarrow Großvater(x, y)$$
$$Eltern(x, z) \Rightarrow Großvater(x, y)$$
$$Vater(x, z) \Rightarrow Großvater(x, y).$$

(Beachten Sie, dass wir davon ausgehen, dass eine Klausel, die *Eltern* definiert, bereits Teil des Hintergrundwissens ist.) Die erste dieser drei Klauseln klassifiziert fälschlicherweise alle zwölf positiven Beispiele als negativ und kann deshalb ignoriert werden. Die zweite und dritte Klausel stimmen mit allen positiven Beispielen überein, aber die zweite ist für einen größeren Anteil der negativen Beispiele falsch – doppelt so viel, weil sie sowohl Mütter als auch Väter zulässt. Wir bevorzugen also die dritte Klausel.

Jetzt müssen wir diese Klausel weiter spezialisieren, um die Fälle auszuschließen, wo x der Vater von einem z ist, aber z nicht Elternteil von y. Das Einfügen des einzelnen Literals *Eltern*(z, y) ergibt:

$$Vater(x, z) \wedge Eltern(z, y) \Rightarrow Großvater(x, y).$$

Damit werden alle Beispiele korrekt klassifiziert. FOIL findet dieses Literal, wählt es aus und löst dabei die Lernaufgabe. Im Allgemeinen ist die Lösung eine Menge von Horn-Klauseln, die jeweils das Zielprädikat implizieren. Wenn wir zum Beispiel das Prädikat *Eltern* nicht in unserem Vokabular hätten, könnte die Lösung wie folgt lauten:

$$Vater(x, z) \wedge Vater(z, y) \Rightarrow Großvater(x, y)$$
$$Vater(x, z) \wedge Mutter(z, y) \Rightarrow Großvater(x, y).$$

Beachten Sie, dass jede dieser Klauseln einige der positiven Beispiele abdeckt, dass sie zusammen alle positiven Beispiele abdecken und dass die NEW-CLAUSE in einer Form konzipiert ist, dass keine Klausel fälschlicherweise ein negatives Beispiel abdeckt. Im Allgemeinen muss FOIL sehr viele nicht erfolgreiche Klauseln durchsuchen, bevor es eine korrekte Lösung findet.

Dieses Beispiel ist eine sehr einfache Demonstration der Arbeitsweise von FOIL. ▶ Abbildung 19.12 skizziert den vollständigen Algorithmus. Im Wesentlichen konstruiert der Algorithmus wiederholt eine Klausel, wobei er jeweils einzelne Literale einfügt, bis die Klausel mit einer Untermenge der positiven Beispiele und keinem der negativen Beispiele übereinstimmt. Die von der Klausel abgedeckten positiven Beispiele werden dann aus der Trainingsmenge entfernt und dieser Prozess setzt sich fort, bis keine weiteren positiven Beispiele übrig sind. Die beiden wichtigsten Subroutinen, die hier erklärt werden sollen, sind NEW-LITERALS, die alle möglichen neuen Literale erzeugt, die der Klausel hinzugefügt werden sollen, und CHOOSE-LITERAL, die ein hinzuzufügendes Literal auswählt.

```
function FOIL(examples, target) returns eine Menge von Horn-Klauseln
   inputs: examples, Beispielmenge
           target, ein Literal für das Zielprädikat
   local variables: clauses, eine Klauselmenge, anfänglich leer

   while examples enthält positive Beispiele do
      clause ← NEW-CLAUSE(examples, target)
      von clause abgedeckte positive Beispiele aus examples entfernen
      clause in clauses einfügen
   return clauses
```

```
function NEW-CLAUSE(examples, target) returns eine Horn-Klausel
   local variables: clause, eine Klausel mit target als Kopf und
                            leerem Rumpf
                    l, ein Literal, das der Klausel hinzugefügt werden soll
                    extended_examples, eine Beispielmenge mit Werten
                                       für neue Variablen

   extended_examples ← examples
   while extended_examples enthält negative Beispiele do
      l ← CHOOSE-LITERAL(NEW-LITERALS(clause), extended_examples)
      l dem Rumpf von clause anfügen
      extended_examples ← Beispielmenge, die durch Anwendung von
                          EXTEND-EXAMPLE auf jedes Beispiel in
                          extended_examples erzeugt wurde
   return clause
```

```
function EXTEND-EXAMPLE(example, literal) returns eine Beispielmenge
   if example erfüllt literal
      then return die Beispielmenge, die durch Extension von example
                  mit jedem möglichen Konstantenwert für jede neue
                  Variable in literal erzeugt wurde
   else return die leere Menge
```

Abbildung 19.12: Skizze des FOIL-Algorithmus für das Lernen von Horn-Klauselmengen erster Stufe aus Beispielen. NEW-LITERALS und CHOOSE-LITERAL sind im Text erklärt.

NEW-LITERALS übernimmt eine Klausel und konstruiert alle „sinnvollen“ Literale, die der Klausel hinzugefügt werden könnten. Wir verwenden als Beispiel die Klausel

$$Vater(x, z) \Rightarrow Großvater(x, y).$$

Es gibt drei Arten von Literalen, die hinzugefügt werden können:

1. *Literale, die Prädikate verwenden*: Das Literal kann negiert oder nicht negiert sein, es kann ein existierendes Prädikat verwendet werden (auch das Zielprädikat) und es muss sich bei allen Argumenten um Variablen handeln. Eine beliebige Variable kann als jedes Argument des Prädikates verwendet werden, jedoch mit einer Einschränkung: Ein Literal muss *mindestens eine* Variable von einem früheren Literal oder vom Kopf der Klausel enthalten. Literale wie etwa $Mutter(z, u)$, $Verheiratet(z, z)$, $\neg Männlich(y)$ oder $Großvater(v, x)$ sind erlaubt, $Verheiratet(u, v)$ dagegen nicht. Beachten Sie, dass die Verwendung des Prädikates vom Kopf der Klausel es erlaubt, dass FOIL *rekursive* Definitionen lernt.

2. *Gleichheits- und Ungleichheitsliterale*: Sie bringen Variablen, die bereits in der Klausel enthalten sind, in eine Beziehung zueinander. Beispielsweise könnten wir $z \neq x$ einfügen. Diese Literale können auch benutzerspezifizierte Konstanten beinhalten. Für das Lernen von Arithmetik könnten wir 0 und 1 verwenden, für das Lernen von Listenfunktionen die leere Liste, [].

3. *Arithmetische Vergleiche*: Bei der Arbeit mit Funktionen von stetigen Variablen können Literale wie etwa $x > y$ und $y \leq z$ hinzugefügt werden. Wie beim Entscheidungsbaumlernen kann ein konstanter Schwellenwert gewählt werden, um die Entscheidungsleistung des Testes zu maximieren.

Der resultierende Verzweigungsfaktor in diesem Suchraum ist zwar sehr groß (siehe Übung 19.6), doch FOIL kann auch Typinformationen nutzen, um ihn zu reduzieren. Enthält die Domäne beispielsweise sowohl Zahlen als auch Menschen, würden Typbedingungen verhindern, dass NEW-LITERALS Literale wie etwa $Eltern(x, n)$ erzeugt, wobei x eine Person und n eine Zahl ist.

CHOOSE-LITERAL verwendet eine Heuristik ähnlich dem Informationsgewinn (siehe *Abschnitt 18.3.4*), um zu entscheiden, welches Literal hinzugefügt werden soll. Die genauen Details sind hier nicht so wichtig und es wurden zahlreiche Variationen ausprobiert. Ein interessantes zusätzliches Funktionsmerkmal von FOIL ist die Verwendung von Ockhams Rasiermesser, um einige Hypothesen zu eliminieren. Wenn eine Klausel (gemäß einer bestimmten Metrik) länger wird als die Gesamtlänge der positiven Beispiele, die sie erklärt, wird diese Klausel nicht als potenzielle Hypothese betrachtet. Durch diese Technik lassen sich zu komplizierte Klauseln vermeiden, die mit dem Rauschen in den Daten übereinstimmen.

FOIL und verwandte Programme sind bereits für das Lernen eines breiten Spektrums von Definitionen eingesetzt worden. Eine der eindrucksvollsten Demonstrationen (Quinlan und Cameron-Jones, 1993) befasst sich mit der Lösung einer langen Folge von Übungen zu Funktionen der Listenverarbeitung aus dem Prolog-Lehrbuch von Bratko (1986). In jedem Fall war das Programm auch in der Lage, eine korrekte Definition der Funktion aus einer kleinen Beispielmenge zu lernen, wobei die vorher gelernten Funktionen als Hintergrundwissen dienten.

19.5.3 Induktives Lernen mit inversem Schließen

Der zweite große ILP-Ansatz invertiert den normalen deduktiven Beweisprozess. Die **inverse Resolution** basiert auf der Beobachtung, dass, wenn die Beispiel-*Klassifizierungen* aus *Hintergrund* $\wedge$ *Hypothese* $\wedge$ *Beschreibungen* folgen, man auch in der Lage sein muss, diese Tatsache durch Resolution zu beweisen (weil die Resolution vollständig ist). Wenn wir „den Beweis rückwärts ausführen" können, lässt sich eine Hypothese finden, sodass der Beweis durchgeht. Der Schlüsselaspekt ist dann, eine Möglichkeit zu finden, den Resolutionsprozess umzukehren.

Wir werden einen Rückwärtsbeweisprozess für die inverse Resolution zeigen, der aus einzelnen Rückwärtsschritten besteht. Wie Sie wissen, übernimmt ein normaler Resolutionsschritt zwei Klauseln C_1 und C_2 und resolviert sie, um die **Resolvente** C zu erzeugen. Ein inverser Resolutionsschritt übernimmt eine Resolvente C und erzeugt zwei Klauseln C_1und C_2, sodass C das Ergebnis der Resolution von C_1und C_2 ist. Alternativ kann er eine Resolvente C und eine Klausel C_1 übernehmen und eine Klausel C_2 erzeugen, sodass C das Ergebnis der Resolution von C_1 und C_2 ist.

▶ Abbildung 19.13 zeigt die frühen Schritte in einem inversen Resolutionsprozess, wo wir uns auf das positive Beispiel *Großeltern*(*George*, *Anne*) konzentrieren. Der Prozess beginnt am Ende des Beweises (unten in der Abbildung dargestellt). Wir gehen davon aus, dass die Resolvente C die leere Klausel ist (d.h. ein Widerspruch) und C_2 gleich $\neg$*Großeltern*(*George*, *Anne*) eine Negierung des Zielbeispiels. Der erste inverse Schritt nimmt C und C_2 und erzeugt die Klausel *Großeltern*(*George*, *Anne*) für C_1. Der nächste Schritt verwendet diese Klausel als C und die Klausel *Eltern*(*Elizabeth*, *Anne*) als C_2 und erzeugt die Klausel

$$\neg Eltern(Elizabeth, y) \vee Großeltern(George, y)$$

als C_1. Der letzte Schritt behandelt diese Klausel als Resolvente. Mit *Eltern*(*George*, *Elizabeth*) als C_2 ist eine mögliche Klausel C_1 die Hypothese

$$Eltern(x, z) \wedge Eltern(z, y) \Rightarrow Großeltern(x, y).$$

Jetzt haben wir einen Resolutionsbeweis, dass Hypothese, Beschreibungen und Hintergrundwissen die Klassifizierung *Großeltern*(*George*, *Anne*) als logische Konsequenz haben.

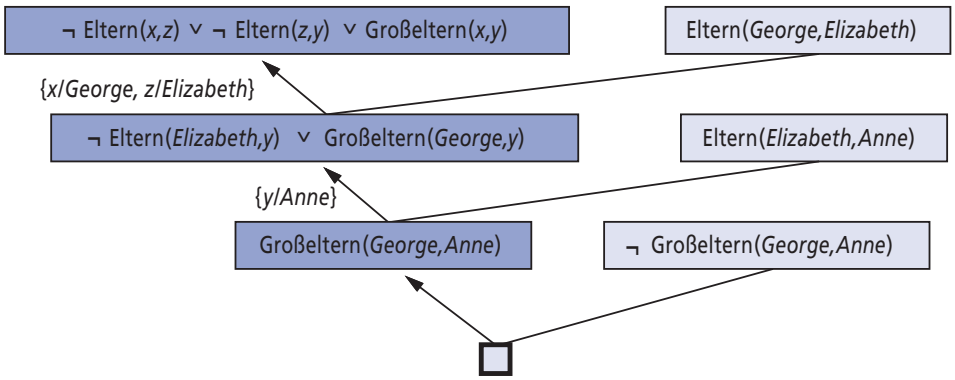


Abbildung 19.13: Frühe Schritte in einem inversen Resolutionsprozess. Die grau unterlegten Klauseln werden von inversen Resolutionsschritten von der Klausel rechts und der Klausel unterhalb erzeugt. Die nicht grau unterlegten Klauseln stammen aus *Beschreibungen* und *Klassifizierungen* (einschließlich negierter *Klassifizierungen*).

Offensichtlich beinhaltet die inverse Resolution eine Suche. Jeder inverse Resolutionsschritt ist nichtdeterministisch, weil es für jedes C viele oder sogar unendlich viele Klauseln C_1 und C_2 geben kann, die zu C resolviert werden. Anstatt beispielsweise $\neg Eltern(Elizabeth, y) \vee Großeltern(George, y)$ für C_1 im letzten Schritt von Abbildung 19.13 zu wählen, hätte der inverse Resolutionsschritt auch einen der folgenden Sätze wählen können:

$$\neg Eltern(Elizabeth, Anne) \vee Großeltern(George, Anne)$$
$$\neg Eltern(z, Anne) \vee Großeltern(George, Anne)$$
$$\neg Eltern(z, y) \vee Großeltern(George, y)$$
$$\vdots$$

(Siehe Übungen 19.4 und 19.5.) Darüber hinaus können die Klauseln, die an jedem Schritt beteiligt sind, aus dem *Hintergrund*wissen, aus den Beispiel*beschreibungen*, aus den negierten *Klassifikationen* oder aus den hypothetisierten Klauseln stammen, die im inversen Resolutionsbaum bereits erzeugt wurden. Die große Anzahl an Möglichkeiten bedeutet einen großen Verzweigungsfaktor (und deshalb eine ineffiziente Suche) ohne zusätzliche Kontrollen. Es wurden verschiedene Ansätze ausprobiert, die Suche in implementierten ILP-Systemen zu begrenzen:

1 Redundante Auswahlen können eliminiert werden – z.B. indem nur die spezifischsten Hypothesen erzeugt werden und indem gefordert wird, dass alle hypothetisierten Klauseln konsistent zueinander und zu den Beobachtungen sind. Dieses letzte Kriterium würde die zuvor aufgelistete Klausel $\neg Eltern(z, y) \vee Großeltern(George, y)$ ausschließen.

2 Die Beweisstrategie kann beschränkt werden. In *Kapitel 9* beispielsweise haben wir gesehen, dass die **lineare Resolution** eine vollständige, beschränkte Strategie ist. Lineare Resolution erzeugt Beweisbäume, die eine lineare Verzweigungsstruktur haben – der gesamte Baum folgt einer Linie, wobei nur einzelne Klauseln von dieser Linie abzweigen (wie in Abbildung 19.13).

3 Die Repräsentationssprache kann beschränkt werden, um beispielsweise Funktionssymbole auszuschließen oder nur Horn-Klauseln zuzulassen. Zum Beispiel arbeitet PROGOL mit Horn-Klauseln unter Verwendung einer **inversen logischen Konsequenz**. Die Idee dabei ist, die Bedingung für die logische Konsequenz

$$Hintergrund \wedge Hypothese \wedge Beschreibungen \models Klassifizierungen$$

in die logisch äquivalente Form

$$Hintergrund \wedge Beschreibungen \wedge \neg Klassifizierungen \models \neg Hypothese$$

zu ändern. Darauf kann man einen Prozess anwenden, der mit der normalen Prolog-Horn-Klausel-Deduktion mit Negation als Fehler vergleichbar ist, um *Hypothesen* abzuleiten. Weil diese Methode auf Horn-Klauseln beschränkt ist, ist sie unvollständig, kann aber effizienter sein als eine vollständige Resolution. Außerdem ist es möglich, eine vollständige Inferenz mit inverser logischer Konsequenz anzuwenden (Inoue, 2001).

4 Inferenz kann mit Model-Checking statt mit Theorembeweis stattfinden. Das PROGOL-System (Muggleton, 1995) verwendet eine Art der Modellüberprüfung, um die Suche einzuschränken. Das bedeutet, es erzeugt wie die Antwortmengenprogrammierung mögliche Werte für logische Variablen und überprüft sie auf Konsistenz.

5 Inferenz kann mithilfe von aussagenlogischen Grundklauseln statt in Logik erster Stufe erfolgen. Das LINUS-System (Lavrač und Džeroski, 1994) übersetzt Theorien erster Stufe in Aussagenlogik, löst sie mit einem aussagenlogischen Lernsystem und übersetzt sie dann zurück. Die Arbeit mit aussagenlogischen Formeln kann für einige Probleme effizienter sein, wie wir am Beispiel von SATPLAN in *Kapitel 10* gezeigt haben.

19.5.4 Entdeckungen mit induktiver logischer Programmierung

Eine inverse Resolutionsprozedur, die eine vollständige Resolutionsstrategie invertiert, ist im Prinzip ein vollständiger Algorithmus für das Lernen von Theorien erster Stufe. Das bedeutet, wenn eine unbekannte *Hypothese* eine Beispielmenge erzeugt, kann eine inverse Resolutionsprozedur eine *Hypothese* aus den Beispielen erzeugen. Diese Beobachtung eröffnet eine interessante Möglichkeit: Angenommen, die verfügbaren Beispiele beinhalten eine Vielzahl von Flugbahnen für fallende Körper. Wäre ein inverses Resolutionsprogramm theoretisch in der Lage, das Schwerkraftgesetz herzuleiten? Die Antwort ist ein deutliches Ja, weil das Schwerkraftgesetz es erlaubt, die Beispiele zu erklären, wenn eine geeignete Hintergrundmathematik zur Verfügung steht. Analog dazu kann man sich vorstellen, dass Elektromagnetismus, Quantenmechanik und die Relativitätstheorie ebenfalls von ILP-Programmen abgedeckt werden können. Natürlich decken sie auch einen Affen mit einer Schreibmaschine ab; wir brauchen noch bessere Heuristiken und neue Möglichkeiten, den Suchraum zu strukturieren.

Was inverse Resolutionssysteme für Sie tun *können*, ist die Erfindung neuer Prädikate. Diese Fähigkeit wird häufig als etwas Mystisches betrachtet, weil man von Computern gerne die Vorstellung hat, dass sie „einfach damit arbeiten, was man ihnen gibt“. Tatsächlich fallen direkt aus dem inversen Resolutionsschritt neue Prädikate an. Der einfachste Fall entsteht bei der Hypothetisierung zweier neuer Klauseln, C_1 und C_2, für eine bekannte Klausel C. Die Resolution von C_1 und C_2 eliminiert ein Literal, das die beiden Klauseln gemeinsam verwenden; damit ist es durchaus möglich, dass das eliminierte Literal ein Prädikat enthält, das in C nicht erscheint. Wenn wir also rückwärts arbeiten, ist es möglich, ein neues Prädikat zu erzeugen, von dem aus wir das fehlende Literal rekonstruieren können.

▶ Abbildung 19.14 zeigt ein Beispiel, in dem das neue Prädikat P während des Prozesses erzeugt wird, eine Definition für *Vorfahre* zu lernen. Nachdem P erzeugt ist, kann es in späteren inversen Resolutionsschritten verwendet werden. Ein späterer Schritt könnte beispielsweise hypothetisieren, dass $Mutter(x, y) \Rightarrow P(x, y)$. Die Bedeutung des neuen Prädikates P ist also durch die Erzeugung von Hypothesen beschränkt, die es beinhalten. Ein weiteres Beispiel könnte zu der Bedingung $Vater(x, y) \Rightarrow P(x, y)$ führen. Mit anderen Worten, das Prädikat P ist das, was wir uns für gewöhnlich als die *Eltern*-Beziehung vorstellen. Wie bereits erwähnt, kann die Einführung neuer Prädikate die Größe der Definition des Zielprädikates wesentlich reduzieren. Durch die Berücksichtigung der Möglichkeit, neue Prädikate einzuführen, können inverse Resolutionssysteme häufig Lernprobleme lösen, die mit anderen Techniken nicht zu bewältigen sind.

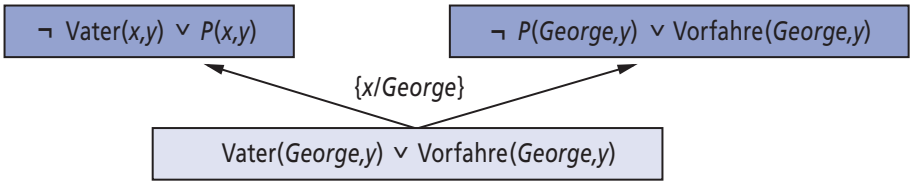


Abbildung 19.14: Ein inverser Resolutionsschritt, der ein neues Prädikat *P* erzeugt.

Einige der bemerkenswertesten Revolutionen in der Wissenschaft stammen aus der Erfindung neuer Prädikate und Funktionen – z.B. Galileos Erfindung der Beschleunigung oder Joules Erfindung der thermischen Energie. Nachdem diese Begriffe erst einmal zur Verfügung stehen, wird die Entdeckung neuer Gesetze (relativ) einfach. Schwierig ist die Erkenntnis, dass eine neue Entität mit einer bestimmten Beziehung zu bereits existierenden Entitäten es erlaubt, einen ganzen Beobachtungsbereich mit einer sehr viel einfacheren und eleganteren Theorie zu erklären, als es bisher möglich war.

ILP-Systeme haben zwar noch keine Entdeckungen auf dem Niveau von Galileo oder Joule gemacht, aber ihre Entdeckungen wurden bereits in wissenschaftlicher Literatur veröffentlicht. Im *Journal of Molecular Biology* beispielsweise beschreiben Turcotte et al. (2001) die automatisierte Entdeckung von Regeln für die Proteinentfaltung durch das ILP-Programm PROGOL. Viele der von PROGOL entdeckten Regeln hätten auch von bekannten Prinzipien abgeleitet werden können, aber die meisten wurden zuvor nicht als Teile einer Standard-Biologiedatenbank veröffentlicht. (Ein Beispiel dafür sehen Sie in Abbildung 19.10.) In einer vergleichbaren Arbeit beschäftigen sich Srinivasan et al. (1994) mit dem Problem, auf Molekularstrukturen basierende Regeln für die Mutagenität von aromatischen Nitroverbindungen zu erkennen. Diese Verbindungen findet man in den Auspuffgasen von Automobilen. Für 80% der Verbindungen in einer Standarddatenbank ist es möglich, vier wichtige Funktionsmerkmale zu identifizieren, wobei eine lineare Regression für diese Funktionsmerkmale ILP überlegen ist. Für die restlichen 20% sind die Funktionsmerkmale allein nicht vorhersagekräftig. ILP erkennt hier Beziehungen und lässt dabei lineare Regression, neuronale Netze und Entscheidungsbäume weit hinter sich. Besonders eindrucksvoll ist, wie King et al. (1992) einen Roboter mit der Fähigkeit ausgestattet haben, molekularbiologische Experimente durchzuführen, und ILP-Techniken erweitert haben, um den Entwurf von Experimenten einzuschließen. Letztlich ist dabei ein autonomer Wissenschaftler entstanden, der tatsächlich neues Wissen über die funktionelle Genomik von Hefe entdeckt hat. Bei all diesen Beispielen scheint sowohl die Fähigkeit, Beziehungen darzustellen, als auch die Möglichkeit, Hintergrundwissen zu verwenden, zu der hohen Leistung von ILPs beizutragen. Die Tatsache, dass der Mensch die von ILP gefundenen Regeln interpretieren kann, hat zu einer Akzeptanz dieser Techniken nicht nur in Informatikmagazinen, sondern auch in Biologiemagazinen geführt.

Neben der Biologie wurde ILP auch in anderen Wissenschaften eingesetzt. Eine der wichtigsten davon ist die Sprachverarbeitung, wo ILP verwendet wurde, um komplexe relationale Informationen aus dem Text zu extrahieren. Die Ergebnisse sind in *Kapitel 23* zusammengefasst.

Bibliografische und historische Hinweise

Obwohl die Verwendung von Vorwissen beim Lernen ein natürliches Thema für Wissenschaftsphilosophen zu sein scheint, gibt es erst seit kurzem formale Arbeiten dazu. *Fact, Fiction, and Forecast* des Philosophen Nelson Goodman (1954) widerlegte die frühere Annahme, dass Induktion einfach eine Frage ausreichend vieler Beispiele irgendeiner universell quantifizierten Aussage und ihrer Übernahme als Hypothese sei. Betrachten Sie beispielsweise die Hypothese „Alle Smaragde sind blün", wobei *blün* bedeutet, „grün, wenn sie vor der Zeit *t* beobachtet werden, aber blau, wenn sie danach beobachtet werden". Wir könnten bis zu jeder Zeit *t* Millionen von Instanzen beobachten, die die Regel bestätigen, dass Smaragde blün sind, und keine widersprechenden Instanzen feststellen, die Regel dennoch aber nicht übernehmen wollen. Das kann durch unseren Umgang mit der Rolle relevanten Vorwissens im Induktionsprozess erklärt werden. Goodman schlägt eine Vielzahl verschiedener Arten von Vorwissen vor, die praktisch sein könnten, wie z.B. eine Version von Bestimmtheiten, die auch als **Überhypothesen** bezeichnet werden. Leider wurden die Ideen von Goodman im maschinellen Lernen nie verfolgt.

Der Ansatz der **aktuell besten Hypothese** ist eine alte Idee in der Philosophie (Mill, 1843). Frühe Arbeiten in der kognitiven Philosophie schlagen auch vor, dass es sich um eine natürliche Form für das Lernen von Konzepten beim Menschen handelt (Bruner et al., 1957). In der KI ist der Ansatz am engsten mit der Arbeit von Patrick Winston verbunden, dessen Doktorarbeit (Winston, 1970) sich mit dem Problem befasst, Beschreibungen von komplexen Objekten zu lernen. Das Modell des **Versionsraumes** (Mitchell, 1977, 1982) verfolgt einen anderen Ansatz, indem es die Menge *aller* konsistenten Hypothesen verwaltet und diejenigen Hypothesen eliminiert, die mit neuen Beispielen inkonsistent sind. Verwendet wurde dieser Ansatz im Chemie-Expertensystem Meta-DENDRAL (Buchanan und Mitchell, 1978) und später im System LEX von Mitchell (1983), das lernt, mathematische Probleme zu lösen. Ein dritter einflussreicher Zweig wurde durch die Arbeit von Michalski und Kollegen zur Reihe der AQ-Algorithmen gebildet, die Mengen logischer Regeln lernten (Michalski, 1969; Michalski et al., 1986).

EBL hat seine Wurzeln in Techniken, die im STRIPS-Planer (Fikes et al., 1972) verwendet wurden. Wenn man einen Plan konstruierte, wurde eine verallgemeinerte Version davon in einer Planbibliothek gespeichert und in späteren Planungen als **Makro-Operator** verwendet. Ähnliche Ideen findet man in der ACT*-Architektur von Anderson unter der Überschrift der **Wissenskompilierung** (Anderson, 1983) sowie in der SOAR-Architektur als **Chunking** (Laird, et al., 1986). **Schemaakquisition** (DeJong, 1981), **analytische Verallgemeinerung** (Mitchell, 1982) und **bedingungsbasierte Verallgemeinerung** (Minton, 1984) waren unmittelbare Vorläufer des schnell wachsenden Interesses am EBL – stimuliert durch die Arbeiten von Mitchell et al. (1986) sowie DeJong und Mooney (1986). Hirsh (1987) führte den im Text beschriebenen EBL-Algorithmus ein und zeigte, wie er direkt in ein Logikprogrammiersystem eingebaut werden konnte. Van Harmelen und Bundy (1988) erklären EBL als Variante der Methode der **partiellen Auswertung**, die in Programm-Analysesystemen (Jones et al., 1993) verwendet wurde.

Die anfängliche Begeisterung für EBL wurde durch die Erkenntnisse von Minton (1988) gedämpft, dass EBL ohne aufwändige Mehrarbeit ein Programm wesentlich verlangsamen kann. Eine formale probabilistische Analyse der mit EBL erwarteten Vorteile finden Sie bei Greiner (1989) sowie bei Subramanian und Feldman (1990). Ein ausgezeichneter Überblick der frühen Arbeiten zur EBL erscheint in Dietterich (1990).

Anstatt Beispiele als Schwerpunkte für die Verallgemeinerung heranzuziehen, kann man sie auch direkt verwenden, um neue Probleme zu lösen – in einem Prozess, der auch als **analogisches Schließen** bezeichnet wird. Diese Art des Schließens reicht von einer Form des plausiblen Schließens, das auf einem Ähnlichkeitsgrad basiert (Gentner, 1983), über eine Form der deduktiven Inferenz, die auf Bestimmtheiten basiert, aber die Beteiligung des Beispiels bedingt (Davies und Russell, 1987), bis hin zu einer Form des „faulen" EBL, das die Richtung der Verallgemeinerung des alten Beispiels anpasst, sodass es für die Bedürfnisse des neuen Problems geeignet ist. Diese letztgenannte Form des analogischen Schließens findet man häufig in **fallbasiertem Schließen** (Kolonder, 1993) und der **derivationalen Analogie** (Veloso und Carbonell, 1993).

Relevanzinformation in Form funktionaler Abhängigkeiten wurde zuerst in der Datenbankgemeinde entwickelt, wo man es nutzte, um große Attributmengen zu strukturieren, sodass überschaubare Untermengen daraus wurden. Funktionale Abhängigkeiten wurden von Carbonell und Collins (1973) für analogisches Schließen verwendet, später von Davies und Russell (Davies, 1985; Davies und Russell, 1987) unabhängig voneinander neu entdeckt und einer vollständigen logischen Analyse unterzogen. Ihre Rolle als Vorwissen beim induktiven Lernen wurde von Russell und Grosof (1987) untersucht. Die Äquivalenz von Bestimmtheiten mit einem Hypothesenraum mit begrenztem Vokabular wurde in Russell (1988) bewiesen. Lernalgorithmen für Bestimmtheiten und die durch RBDTL verbesserte Leistung wurden zuerst im FOCUS-Algorithmus gezeigt, der auf Almuallim und Dietterich (1991) zurückgeht. Tadepalli (1993) beschreibt einen genialen Algorithmus für das Lernen mit Bestimmtheiten, der große Verbesserungen in der Lerngeschwindigkeit zeigt.

Die Idee, dass induktives Lernen durch eine inverse Deduktion erfolgen kann, geht zurück auf W.S. Jevons (1874), der schrieb: „Das Studium der formalen Logik und der Wahrscheinlichkeitstheorie hat mich zu der Überzeugung gebracht, dass es keine eigenständige Methode der Induktion im Gegensatz zur Deduktion gibt, sondern dass die Induktion einfach eine umgekehrte Anwendung der Deduktion ist." Berechnungsuntersuchungen begannen mit der bemerkenswerten Doktorarbeit von Gordon Plotkin (1971) in Edinburgh. Obwohl Plotkin viele der heute im ILP verwendeten Theoreme und Methoden entwickelte, ließ er sich von einigen nicht entscheidbaren Ergebnissen für bestimmte Unterprobleme der Induktion entmutigen. MIS (Shapiro, 1981) führte das Problem des Lernens von Logikprogrammen wieder ein, was jedoch hauptsächlich als Beitrag zur Theorie automatisierten Debuggings betrachtet wurde. Arbeiten zur Regelinduktion, wie beispielsweise die Systeme ID3 (Quinlan, 1986) und CN2 (Clark und Niblett, 1989), waren die Grundlage für FOIL (Quinlan, 1990), das zum ersten Mal eine praktische Induktion relationaler Regeln zuließ. Der Bereich des relationalen Lernens wurde von Muggleton und Buntine (1988) wiederbelebt, deren CIGOL-Programm eine etwas unvollständigere Version der inversen Resolution verwendete und in der Lage war, neue Prädikate zu erzeugen. Die inverse Resolutionsmethode

erscheint auch in (Russell, 1986) mit einem einfachen Algorithmus in einer Fußnote. Das nächste wichtige System war GOLEM (Muggleton und Feng, 1990). Es verwendet einen Covering-Algorithmus, der auf dem Konzept der verhältnismäßig wenigsten allgemeinen Verallgemeinerung von Plotkin basiert. ITOU (Rouveirol und Puget, 1989) und CLINT (De Raedt, 1992) waren ebenfalls Systeme dieser Ära. Das neuere PROGOL (Muggleton, 1995) verwendet einen hybriden Ansatz (Top-down und Bottom-up) der inversen logischen Konsequenz und wurde bereits auf zahlreiche praktische Probleme angewendet, insbesondere in der Biologie und der Sprachverarbeitung. Muggleton (2000) beschreibt eine Erweiterung von PROGOL, die Unsicherheit in der Form stochastischer Logikprogramme verarbeitet.

Eine formale Analyse von ILP-Methoden finden Sie in Muggleton (1991), eine umfassende Sammlung von Arbeiten in Muggleton (1992) und einen Überblick über Techniken und Anwendungen in dem Buch von Lavrač und Džeroski (1994). Page und Srinivasan (2002) bieten einen aktuelleren Überblick über die Geschichte dieses Gebietes und zeigen Herausforderungen der Zukunft auf. Frühe Komplexitätsergebnisse von Haussler (1989) erbrachten, dass das Lernen von Sätzen erster Stufe hoffnungslos kompliziert war. Mit einem besseren Verständnis für die Bedeutung der verschiedenen Arten syntaktischer Beschränkungen von Klauseln gelangte man allerdings zu positiven Ergebnissen selbst für rekursive Klauseln (Džeroski et al., 1992). Lernfähigkeitsergebnisse für ILP sind in Kietz und Džeroski (1994) sowie in Cohen und Page (1995) beschrieben.

Obwohl ILP heute der dominierende Ansatz für die konstruktive Induktion zu sein scheint, ist er nicht der einzige. Sogenannte **Discovery-Systeme** versuchen, den Prozess wissenschaftlicher Entdeckungen neuer Konzepte zu modellieren – in der Regel durch eine direkte Suche im Raum der Konzeptdefinitionen. Der Automated Mathematician (AM) von Doug Lenat (Davis und Lenat, 1982) verwendete Vermutungen in der elementaren Zahlentheorie. Anders als die meisten anderen Systeme, die für mathematisches Schließen entwickelt worden waren, fehlte dem AM ein Beweiskonzept und er konnte nur Vermutungen treffen. Seine Wiederentdeckungen waren die Goldbach-Vermutung und das Theorem über die Eindeutigkeit der Primfaktorzerlegung. Die Architektur von AM wurde im EURISKO-System (Lenat, 1983) verallgemeinert, indem ein Mechanismus hinzugefügt wurde, der in der Lage war, die eigenen Entdeckungsheuristiken des Systems umzuschreiben. EURISKO wurde auch in vielen anderen Bereichen als der mathematischen Forschung eingesetzt, allerdings mit weniger Erfolg als AM. Die Vorgehensweise von AM und EURISKO war entgegengesetzt (Ritchie und Hanna, 1984; Lenat und Brown, 1984).

Eine weitere Klasse von Discovery-Systemen versucht, mit realen wissenschaftlichen Daten zu arbeiten, um neue Gesetze abzuleiten. Die Systeme DALTON, GLAUBER und STAHL (Langley et al., 1987) sind regelbasierte Systeme, die nach quantitativen Beziehungen in experimentellen Daten aus physischen Systemen suchen. In jedem Fall war das System in der Lage, eine wohl bekannte Entdeckung aus der Wissenschaftsgeschichte zu rekapitulieren. *Kapitel 20* beschreibt Discovery-Systeme, die auf probabilistischen Techniken basieren – insbesondere Clustering-Algorithmen, die neue Kategorien entdecken.

Zusammenfassung

Dieses Kapitel hat gezeigt, wie Vorwissen einem Agenten auf verschiedene Weise helfen kann, aus neuen Erfahrungen zu lernen. Weil mehr Vorwissen mithilfe von relationalen Modellen anstelle von attributbasierten Modellen dargestellt wird, haben wir auch Systeme beschrieben, die es erlauben, relationale Modelle zu lernen. Die wichtigsten Aspekte sind:

- Die Verwendung von Vorwissen beim Lernen führt zu einem Bild des **kumulativen Lernens**, wobei lernende Agenten ihre Lernfähigkeit verbessern, wenn sie mehr Wissen besitzen.
- Vorwissen unterstützt das Lernen, indem anderweitig konsistente Hypothesen eliminiert und die Erklärungen von Beispielen „aufgefüllt" werden, sodass kürzere Hypothesen möglich werden. Auf diese Weise ist häufig ein schnelleres Lernen aus weniger Beispielen möglich.
- Das Verständnis der verschiedenen logischen Rollen, die das Vorwissen spielt, ausgedrückt durch **Bedingungen der logischen Konsequenz**, hilft, eine Vielzahl von Lerntechniken zu definieren.
- **Erklärungsbasiertes Lernen** (EBL) extrahiert allgemeine Regeln aus einzelnen Beispielen, indem die Beispiele *erklärt* werden und die Erklärung verallgemeinert wird. Es unterstützt eine deduktive Methode, wobei Vorwissen in sinnvolle, effiziente und spezifische Erfahrung umgewandelt wird.
- **Wissensbasiertes induktives Lernen** (**KBIL**) sucht induktive Hypothesen, die Beobachtungsmengen mithilfe von Hintergrundwissen erklären.
- Techniken der **induktiven logischen Programmierung** (**ILP**) führen KBIL für Wissen aus, das in Logik erster Stufe dargestellt ist. ILP-Methoden können relationales Wissen lernen, das in attributbasierten Systemen nicht ausgedrückt werden kann.
- ILP kann mithilfe eines Top-down-Ansatzes erfolgen, wobei eine sehr allgemeine Regel verfeinert wird, oder mithilfe eines Bottom-up-Ansatzes, wobei der deduktive Prozess umgekehrt wird.
- ILP-Methoden erzeugen natürlicherweise neue Prädikate, mit deren Hilfe präzise neue Theorien ausgedrückt werden können, und stellen sich sehr viel versprechend als allgemeine Systeme für die Formulierung wissenschaftlicher Theorien dar.

Übungen zu Kapitel 19

Lösungshinweise

1 Zeigen Sie durch Übersetzung in konjunktive Normalform und Anwendung der Resolution, dass der in *Abschnitt 19.4* gezogene Schluss zu den Brasilianern folgerichtig ist.

2 Schreiben Sie für jede der folgenden Bestimmtheiten die logische Repräsentation und erklären Sie, warum die Bestimmtheit wahr ist (falls sie wahr ist):

a. Die Postleitzahl bestimmt den Staat (Amerika).

b. Layout und Beschriftung bestimmen die Masse einer Münze.

c. Klima, Nahrungsaufnahme, Sport und Stoffwechsel bestimmen Gewichtszu- und -abnahme.

d. Kahlköpfigkeit ist festgelegt durch die Kahlköpfigkeit (oder ihr Fehlen) des Großvaters mütterlicherseits.

3 Wäre eine probabilistische Version von Bestimmtheiten sinnvoll? Schlagen Sie eine Definition vor.

4 Tragen Sie die fehlenden Werte für die Klauseln C_1 oder C_2 (oder beide) in die folgenden Klauseln ein. C ist dabei der Resolvent von C_1 und C_2:

a. $C = True \Rightarrow P(A, B)$, $C_1 = P(x, y) \Rightarrow Q(x, y)$, $C_2 = ??$

b. $C = True \Rightarrow P(A, B)$, $C_1 = ??$, $C_2 = ??$

c. $C = P(x, y) \Rightarrow P(x, f(y))$, $C_1 = ??$, $C_2 = ??$

Falls es mehrere Lösungen gibt, geben Sie für jede Art ein Beispiel an.

5 Angenommen, wir schreiben ein Logikprogramm, das einen Resolutions-Inferenzschritt ausführt. Das heißt, $Resolve(c_1, c_2, c)$ ist erfolgreich, wenn c das Ergebnis der Resolution von c_1 und c_2 ist. Normalerweise würde $Resolve$ als Teil eines Theorembeweisers verwendet, indem es mit c_1 und c_2 für bestimmte Klauseln instantiiert aufgerufen wird und dabei die Resolvente c erzeugt. Hier nehmen wir stattdessen an, dass wir es mit c instantiiert und mit c_1 und c_2 nicht instantiiert aufrufen. Können wir damit die entsprechenden Ergebnisse eines inversen Resolutionsschrittes erzeugen? Braucht man eine bestimmte Veränderung an dem Logikprogrammiersystem, damit dies funktioniert?

6 Nehmen Sie an, dass FOIL einer Klausel ein Literal mit einem binären Prädikat P hinzufügen will und dass vorhergehende Literale (einschließlich dem Kopf der Klausel) fünf verschiedene Variablen enthalten.

a. Wie viele funktional unterschiedliche Literale können erzeugt werden? Zwei Literale sind funktional identisch, wenn sie sich nur in den Namen der *neuen* Variablen unterscheiden, die sie enthalten.

b. Finden Sie eine allgemeine Formel für die Anzahl der verschiedenen Literale mit einem Prädikat der Stelligkeit r, wenn es n zuvor verwendete Variablen gibt.

c. Warum erlaubt FOIL keine Literale, die keine zuvor verwendeten Variablen enthalten?

Lernen probabilistischer Modelle

20

ÜBERBLICK

In diesem Kapitel betrachten wir das Lernen als eine Form unsicheren Schließens aus Beobachtungen.

Kapitel 13 hat die Vorherrschaft von Unsicherheit in realen Umgebungen dargelegt. Mithilfe der Methoden von Wahrscheinlichkeits- und Entscheidungstheorie können Agenten Unsicherheit verarbeiten, aber zuerst müssen sie ihre Wahrscheinlichkeitstheorien der Welt aus Erfahrung lernen. Dieses Kapitel beschreibt, wie sie das bewerkstelligen, indem man die eigentliche Lernaufgabe als Prozess probabilistischer Inferenz formuliert (*Abschnitt 20.1*). Wir werden sehen, dass die Bayessche Perspektive des Lernens extrem leistungsfähig ist und allgemeine Lösungen für die Probleme des Rauschens, der Überanpassung und der optimalen Vorhersage bereitstellt. Außerdem berücksichtigt sie die Tatsache, dass ein nicht allwissender Agent nie sicher sein kann, welche Theorie der Welt korrekt ist, und Entscheidungen treffen muss, indem er irgendeine Theorie der Welt verwendet. In den *Abschnitten 20.2* und *20.3* beschreiben wir Methoden für das Lernen von Wahrscheinlichkeitsmodellen – wobei es hauptsächlich um Bayessche Netze geht. Ein Teil des Stoffes in diesem Kapitel ist recht mathematisch, während die allgemeinen Aspekte verstanden werden können, ohne zu sehr ins Detail gehen zu müssen. Möglicherweise ist es für den Leser hilfreich, noch einmal die *Kapitel 13* und *14* durchzulesen und einen Blick in Anhang A zu werfen.

20.1 Statistisches Lernen

Die Schlüsselkonzepte in diesem Kapitel sind wie in *Kapitel 18* **Daten** und **Hypothesen**. Hier sind die Daten **Evidenz** – d.h. Instantiierungen einiger oder aller Zufallsvariablen, die die Domäne beschreiben. Die Hypothesen in diesem Kapitel sind probabilistische Theorien, wie die Domäne funktioniert, einschließlich logischer Theorien als Spezialfall.

Betrachten wir ein *sehr* einfaches Beispiel. Unsere Lieblingsbonbons Surprise gibt es in zwei Geschmacksrichtungen: Kirsche (lecker!) und Zitrone (buah!). Der Bonbonhersteller hat einen seltsamen Humor und packt alle Bonbons in dasselbe undurchsichtige Papier ein, egal um welche Geschmacksrichtung es sich handelt. Die Bonbons werden in großen Tüten verkauft, von denen es fünf Arten gibt – die ebenfalls von außen nicht zu unterscheiden sind:

h_1: 100% Kirsche
h_2: 75% Kirsche + 25% Zitrone
h_3: 50% Kirsche + 50% Zitrone
h_4: 25% Kirsche + 75% Zitrone
h_5: 100% Zitrone.

Wenn wir eine neue Tüte Bonbons haben, gibt die Zufallsvariable H (für *Hypothese*) den Typ der Tüte mit möglichen Werten h_1 bis h_5 an. H ist natürlich nicht direkt beobachtbar. Wenn die einzelnen Bonbons geöffnet und untersucht werden, werden Daten offen gelegt – D_1, D_2, ..., D_N, wobei jedes D_i eine Zufallsvariable mit den möglichen Werten *kirsche* und *zitrone* ist. Der Agent steht der grundlegenden Aufgabe gegenüber, die Geschmacksrichtung des nächsten Bonbons vorauszusagen.[1] Obwohl dieses Sze-

1 Statistisch erfahrene Leser erkennen dieses Szenario als Variante des **Urnen/Kugeln**-Beispiels. Wir finden jedoch Urnen und Kugeln weniger attraktiv als Bonbons; darüber hinaus führen Bonbons wiederum zu anderen Aufgaben, wie beispielsweise der Entscheidung, ob man die Tüte mit einem Freund teilt – siehe Übung 20.3.

nario scheinbar trivial ist, kann es dazu dienen, viele der wichtigsten Aspekte aufzuzeigen. Der Agent muss eine Theorie der Welt ableiten, wenn auch eine sehr einfache.

Bayessches Lernen berechnet einfach die Wahrscheinlichkeit jeder Hypothese für die Daten und trifft Vorhersagen auf dieser Basis. Das heißt, die Vorhersagen werden unter Verwendung *aller* Hypothesen getroffen, die nach ihren Wahrscheinlichkeiten gewichtet sind, und nicht nur anhand einer einzelnen „besten" Hypothese. Auf diese Weise wird das Lernen auf probabilistische Inferenz reduziert. Soll $\mathbf{D}$ alle Daten mit dem beobachteten Wert $\mathbf{d}$ darstellen; dann ist die Wahrscheinlichkeit jeder Hypothese durch die Bayessche Regel gegeben als:

$$P(h_i|\mathbf{d}) = \alpha P(\mathbf{d}|h_i)P(h_i). \tag{20.1}$$

Angenommen, wir wollen eine Vorhersage über eine unbekannte Größe X machen. Dann ist:

$$\mathbf{P}(X|\mathbf{d}) = \sum_i \mathbf{P}(X|\mathbf{d}, h_i)\mathbf{P}(h_i|\mathbf{d}) = \sum_i \mathbf{P}(X|h_i)P(h_i|\mathbf{d}). \tag{20.2}$$

Dabei haben wir vorausgesetzt, dass jede Hypothese eine Wahrscheinlichkeitsverteilung über X angibt. Diese Gleichung zeigt, dass die Vorhersagen gewichtete Mittelwerte der Vorhersagen der einzelnen Hypothesen sind. Die eigentlichen Hypothesen sind im Wesentlichen „Zwischenstufen" zwischen den Rohdaten und den Vorhersagen. Die Schlüsselgrößen im Bayesschen Ansatz sind die **A-priori-Annahme der Hypothese** $P(h_i)$ und die **Wahrscheinlichkeit** der Daten unter jeder Hypothese, $P(\mathbf{d}|h_i)$.

Für unser Bonbonbeispiel nehmen wir an, dass die A-priori-Verteilung über h_1, ..., h_5 gegeben ist durch $\langle 0{,}1,\ 0{,}2,\ 0{,}4,\ 0{,}2,\ 0{,}1\rangle$, wie vom Hersteller angegeben. Die Wahrscheinlichkeit der Daten wird unter der Annahme berechnet, dass die Beobachtungen unabhängig und identisch verteilt sind (siehe *Abschnitt 18.4*), sodass gilt:

$$P(\mathbf{d}|h_i) = \prod_j P(d_j|h_i). \tag{20.3}$$

Angenommen, die Tüte ist wirklich eine reine Zitronentüte (h_5) und die ersten zehn Bonbons haben alle die Geschmacksrichtung Zitrone. Dann ist $P(\mathbf{d}|h_3)$ gleich $0{,}5^{10}$, weil die Hälfte der Bonbons in einer h_3-Tüte Zitronenbonbons sind.[2] (a) zeigt, wie sich die A-posteriori-Wahrscheinlichkeiten der fünf Hypothesen ändern, während die Folge der zehn Zitronenbonbons beobachtet wird. Beachten Sie, dass die Wahrscheinlichkeiten mit ihren A-priori-Werten beginnen; deshalb ist h_3 anfänglich die wahrscheinlichste Wahl und bleibt dies auch, nachdem ein Zitronenbonbon ausgepackt ist. Nach zwei ausgepackten Zitronenbonbons ist h_4 am wahrscheinlichsten, nach drei oder mehr ist es h_5 (die unbeliebte reine Zitronentüte). Nach zehn Bonbons hintereinander sind wir uns unseres Schicksals fast sicher. ▶ Abbildung 20.1(b) zeigt die vorhergesagte Wahrscheinlichkeit, dass das nächste Bonbon Zitrone ist – basierend auf Gleichung (20.2). Wie wir erwarten würden, steigt der Wert monoton gegen 1.

2 Wir haben bereits gesagt, dass die Tüten sehr groß sind; andernfalls gilt die *u.i.v.*-Annahme (unabhängig und identisch verteilt) nicht. Technisch gesehen ist es korrekter (aber weniger hygienisch), die Bonbons nach der Untersuchung wieder zu verpacken und in die Tüte zurückzulegen.

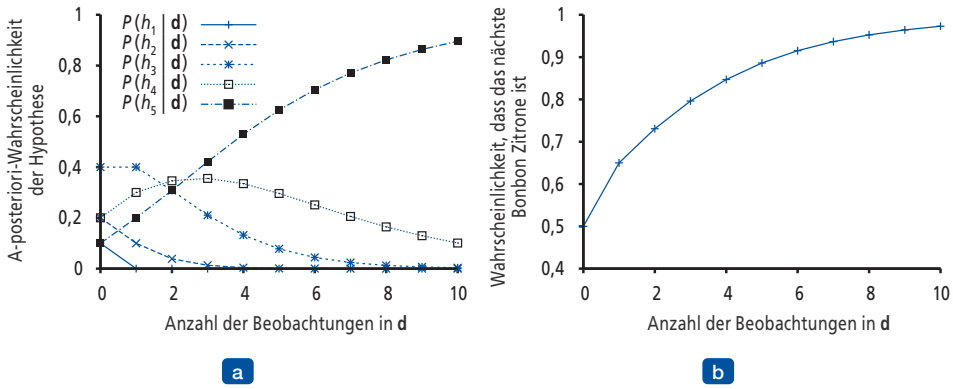


Abbildung 20.1: (a) A-posteriori-Wahrscheinlichkeiten $P(h_i | d_1, ..., d_N)$ aus Gleichung (20.1). Die Anzahl der Beobachtungen N reicht von 1 bis 10 und jede Beobachtung ergibt ein Zitronenbonbon. (b) Bayessche Vorhersage $P(d_{N+1} = zitrone | d_1, ..., d_N)$ aus Gleichung (20.2).

Tipp

Das Beispiel zeigt, dass *die Bayessche Vorhersage schließlich mit der wahren Hypothese übereinstimmt*. Dies ist für Bayessches Lernen charakteristisch. Für jede feste A-priori-Verteilung, die die wahre Hypothese nicht ausschließt, verschwindet die A-posteriori-Wahrscheinlichkeit einer falschen Hypothese unter bestimmten technischen Bedingungen irgendwann. Und zwar einfach deshalb, weil die Wahrscheinlichkeit, unbegrenzt „uncharakteristische" Daten zu erzeugen, verschwindend klein ist. (Dieser Aspekt ist analog zu unserer Beobachtung des PAC-Lernens in *Kapitel 18*.) Wichtiger ist, dass die Bayessche Vorhersage *optimal* ist, egal ob die Datenmenge groß oder klein ist. Bei gegebener A-priori-Hypothese ist für jede andere Vorhersage zu erwarten, dass sie seltener korrekt ist.

Die Optimalität des Bayesschen Lernens hat natürlich ihren Preis. Für reale Lernprobleme ist der Hypothesenraum in der Regel sehr groß oder unendlich, wie wir in *Kapitel 18* gesehen haben. In einigen Fällen kann die Summierung in Gleichung (20.2) (oder die Integration im stetigen Fall) überschaubar durchgeführt werden, aber größtenteils müssen wir auf annähernde oder vereinfachte Methoden ausweichen.

Eine sehr gebräuchliche Annäherung – eine, die gewöhnlich in der Wissenschaft übernommen wird – trifft Vorhersagen basierend auf einer einzigen *wahrscheinlichsten* Hypothese – d.h. einer Hypothese h_i, die $P(h_i | \mathbf{d})$ maximiert. Man spricht auch häufig von der **Maximum-a-posteriori-** oder **MAP-Hypothese**. Vorhersagen, die gemäß einer MAP-Hypothese h_{MAP} getroffen werden, sind insoweit annähernd bayesianisch, dass $\mathbf{P}(X | \mathbf{d}) \approx \mathbf{P}(X | h_{MAP})$. In unserem Bonbonbeispiel ist $h_{MAP} = h_5$ nach drei Zitronenbonbons in Folge, sodass der MAP-Lernalgorithmus dann vorhersagt, dass das vierte Bonbon mit einer Wahrscheinlichkeit von 1,0 Zitrone ist – eine sehr viel gefährlichere Vorhersage als die Bayessche Vorhersage von 0,8, die in Abbildung 20.1(b) gezeigt ist. Wenn mehr Daten eintreffen, liegen MAP- und Bayessche Vorhersagen immer näher zusammen, weil die Konkurrenten zur MAP-Hypothese immer weniger wahrscheinlich werden.

Obwohl es unser Beispiel nicht zeigt, ist es häufig viel einfacher, MAP-Hypothesen zu finden, als ein Bayessches Lernen auszuführen, weil dafür ein Optimierungsproblem gelöst werden muss – statt eines großen Summierungsproblems (oder Integrationsproblems). Beispiele dafür zeigen wir später in diesem Kapitel.

Sowohl beim Bayesschen Lernen als auch beim MAP-Lernen spielt die A-priori-Hypothesenverteilung $P(h_i)$ eine wichtige Rolle. In *Kapitel 18* haben wir gesehen, dass eine **Überanpassung** auftreten kann, wenn der Hypothesenraum zu ausdrucksstark ist, sodass er viele Hypothesen enthält, die gut mit der Datenmenge übereinstimmen. Anstatt eine willkürliche Grenze für die zu betrachtenden Hypothesen anzugeben, setzen die Methoden des Bayesschen und des MAP-Lernens die A-priori-Verteilung ein, um *Komplexität zu bestrafen*. Typischerweise haben komplexere Hypothesen eine geringere A-priori-Wahrscheinlichkeit – zum Teil, weil es in der Regel sehr viel mehr komplexe als einfache Hypothesen gibt. Andererseits haben komplexere Hypothesen eine höhere Kapazität, sich an die Daten anzupassen. (Im Extremfall kann eine Suchtabelle Daten exakt mit der Wahrscheinlichkeit 1 reproduzieren.) Damit verkörpert die bedingte Hypothesenverteilung eine Abwägung zwischen der Komplexität einer Hypothese und dem Ausmaß, inwieweit sie mit den Daten übereinstimmt.

Tipp

Wir sehen den Effekt dieser Abwägung am deutlichsten im logischen Fall, wenn H nur *deterministische* Hypothesen enthält. In diesem Fall ist $P(\mathbf{d}|h_i)$ gleich 1, wenn h_i konsistent ist, andernfalls 0. Anhand von Gleichung (20.1) ist zu erkennen, dass h_{MAP} dann die *einfachste logische Theorie ist, die konsistent mit den Daten ist.* Aus diesem Grund stellt das Maximum-a-posteriori-Lernen eine natürliche Verkörperung von Ockhams Rasiermesser dar.

Eine weitere Einsicht in die Abwägung zwischen Komplexität und Übereinstimmungsgrad erhält man, indem man den Logarithmus von Gleichung (20.1) ermittelt. Die Auswahl von h_{MAP}, um $P(\mathbf{d}|h_i)P(h_i)$ zu maximieren, ist äquivalent zum Minimieren von

$$-\log_2 P(\mathbf{d}|h_i) - \log_2 P(h_i).$$

Mit der Verbindung zwischen Informationskodierung und Wahrscheinlichkeit, die wir in *Abschnitt 18.3.4* eingeführt haben, erkennen wir, dass der Term $-\log_2 P(h_i)$ gleich der Anzahl der Bits ist, die für die Spezifikation der Hypothese h_i erforderlich sind. Darüber hinaus ist $\log_2 P(\mathbf{d}|h_i)$ die zusätzliche Anzahl an Bits für die Spezifikation der Daten für die Hypothese. (Um dies zu erkennen, überlegen Sie, dass keine Bits erforderlich sind, wenn die Hypothese die Daten genau vorhersagt – wie bei h_5 und der Folge der Zitronenbonbons – und $\log_2 1 = 0$.) Das MAP-Lernen wählt also die Hypothese aus, die eine maximale *Komprimierung* der Daten unterstützt. Die gleiche Aufgabe wird direkter durch die **minimale Beschreibungslänge** (MDL, Minimum Description Length) gelöst. Während MAP-Lernen Einfachheit ausdrückt, indem einfacheren Hypothesen höhere Wahrscheinlichkeiten zugeordnet werden, drückt MDL sie direkt aus, indem die Bits in einer binären Kodierung der Hypothesen und Daten gezählt werden.

Eine letzte Vereinfachung entsteht durch die Annahme einer **einheitlichen** A-priori-Verteilung über dem Hypothesenraum. In diesem Fall reduziert sich das MAP-Lernen auf die Auswahl einer Hypothese h_i, die $P(\mathbf{d}|H_i)$ maximiert. Man spricht auch von einer **Maximum-Likelihood**-Hypothese (ML), h_{ML}. Das ML-Lernen ist in der Statistik sehr gebräuchlich, einer Disziplin, in der viele Forscher der subjektiven Natur der A-priori-Hypothesenverteilungen misstrauen. Es handelt sich dabei um einen sinnvollen Ansatz, wenn es keinen Grund gibt, eine Hypothese einer anderen gegenüber *a priori* zu bevorzugen – z.B. wenn alle Hypothesen gleich komplex sind. ML-Lernen stellt eine gute Annäherung für Bayessches und MAP-Lernen dar, wenn die Datenmenge groß ist, weil die Daten die A-priori-Verteilung über Hypothesen überschwemmen, hat aber bei kleinen Datenmengen Probleme (wie wir noch sehen werden).

20.2 Lernen mit vollständigen Daten

Die allgemeine Aufgabe, ein probabilistisches Modell zu lernen, wobei angenommen wird, dass die Daten von diesem Modell generiert werden, bezeichnet man als **Dichteschätzung**. (Der Begriff bezog sich ursprünglich auf Wahrscheinlichkeitsdichtefunktionen für stetige Variablen, ist jetzt aber auch für diskrete Verteilungen gebräuchlich.)

Dieser Abschnitt beschäftigt sich mit dem einfachsten Fall, in dem **vollständige Daten** vorliegen. Die Daten sind vollständig, wenn jeder Datenpunkt Werte für jede Variable im zu lernenden Wahrscheinlichkeitsmodell enthält. Wir konzentrieren uns hier auf **Parameterlernen** – das Ermitteln der numerischen Parameter für ein Wahrscheinlichkeitsmodell mit fester Struktur. Zum Beispiel könnten wir daran interessiert sein, die bedingten Wahrscheinlichkeiten in einem Bayesschen Netz mit vorgegebener Struktur zu lernen. Außerdem werfen wir einen Blick auf das Problem, die Struktur zu lernen, und auf parameterfreie Dichteschätzungen.

20.2.1 ML-Parameterlernen: diskrete Modelle

Angenommen, wir kaufen eine Tüte Zitronen- und Kirschbonbons von einem neuen Hersteller, dessen Zitrone/Kirsche-Verhältnisse völlig unbekannt sind – das Verhältnis kann also irgendwo zwischen 0 und 1 liegen. In diesem Fall haben wir ein Hypothesenkontinuum. Der **Parameter** in diesem Fall, den wir θ nennen, ist der Anteil der Kirschbonbons, und die Hypothese ist h_θ. (Der Anteil der Zitronenbonbons ist dann einfach $1 - \theta$.) Wenn wir davon ausgehen, dass alle Verhältnisse *a priori* gleichwahrscheinlich sind, dann ist ein ML-Ansatz (Maximum Likelihood) sinnvoll. Wenn wir die Situation mithilfe eines Bayesschen Netzes modellieren, brauchen wir nur eine Zufallsvariable, *Geschmack* (die Geschmacksrichtung des zufällig aus der Tüte gewählten Bonbons). Sie hat die Werte *kirsche* und *zitrone*, wobei die Wahrscheinlichkeit von *kirsche* gleich θ ist (siehe ▶ Abbildung 20.2(a)). Nehmen wir nun an, wir packen N Bonbons aus, von denen c die Geschmacksrichtung Kirsche haben und $\ell = N - c$ die Geschmacksrichtung Zitrone. Gemäß Gleichung (20.3) ist die Wahrscheinlichkeit dieser Datenmenge gleich

$$P(\mathbf{d} \mid h_\theta) = \prod_{j=1}^{N} P(d_j \mid h_\theta) = \theta^c \cdot (1-\theta)^\ell .$$

Die ML-Hypothese ist gegeben durch den Wert von θ, der diesen Ausdruck maximiert. Denselben Wert erhält man durch Maximierung der **log-Wahrscheinlichkeit**:

$$L(\mathbf{d} \mid h_\theta) = \log P(\mathbf{d} \mid h_\theta) = \sum_{j=1}^{N} \log P(d_j \mid h_\theta) = c \log\theta + \ell \log(1-\theta).$$

(Durch Logarithmieren reduzieren wir das Produkt auf eine Summe über die Daten, was in der Regel einfacher zu maximieren ist.) Um den ML-Wert von θ zu ermitteln, differenzieren wir L nach θ und setzen den resultierenden Ausdruck gleich null:

$$\frac{dL(\mathbf{d} \mid h_\theta)}{d\theta} = \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 \quad \Rightarrow \quad \theta = \frac{c}{c+\ell} = \frac{c}{N}.$$

Das bedeutet, die ML-Hypothese h_{ML} behauptet, dass das tatsächliche Verhältnis der Kirschen in der Tüte gleich dem beobachteten Verhältnis der bisher nicht ausgepackten Bonbons ist!

Wir haben also eine Menge Arbeit geleistet, um das Offensichtliche zu entdecken. In Wirklichkeit aber haben wir eine Standardmethode für das ML-Parameterlernen skizziert, eine Methode mit breiter Anwendbarkeit:

1. Aufschreiben eines Ausdrucks für die Wahrscheinlichkeit der Daten als Funktion des Parameters/der Parameter
2. Aufschreiben der Ableitung der Log-Wahrscheinlichkeit für jeden Parameter
3. Ermittlung von Parameterwerten, für die die Ableitungen null sind

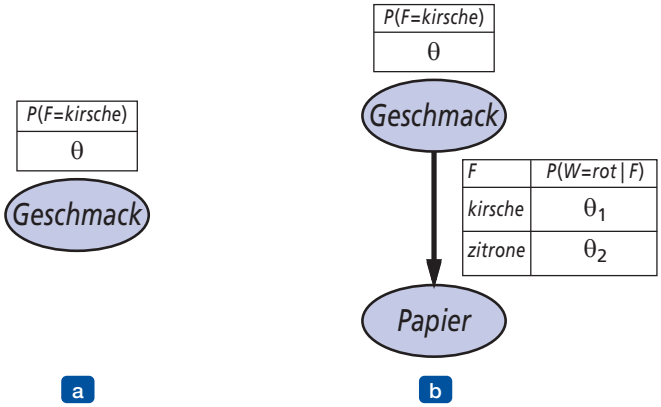


Abbildung 20.2: (a) Bayessches Netzmodell für den Fall der Bonbons mit unbekanntem Verhältnis von Kirsche und Zitrone. (b) Modell für den Fall, in dem die Papierfarbe (probabilistisch) vom Bonbongeschmack abhängig ist.

Tipp

Der heikelste Schritt ist in der Regel der letzte. In unserem Beispiel war er trivial, aber wir werden sehen, dass wir in vielen Fällen auf iterative Lösungsalgorithmen oder andere numerische Optimierungstechniken ausweichen müssen, wie in *Kapitel 4* beschrieben. Das Beispiel demonstriert außerdem ein wesentliches Problem des ML-Lernens im Allgemeinen: *Wenn die Datenmenge klein genug ist, dass einige Ereignisse noch nicht beobachtet wurden – z.B. keine Kirschbonbons –, dann weist die ML-Hypothese diesen Ereignissen die Wahrscheinlichkeit null zu*. Es werden verschiedene Tricks angewendet, um dieses Problem zu vermeiden, wie etwa die Initialisierung des Zählers für die einzelnen Ereignisse mit 1 statt mit 0.

Betrachten wir ein weiteres Beispiel. Angenommen, der neue Bonbonhersteller will dem Kunden einen kleinen Tipp geben und verwendet rotes und grünes Bonbonpapier. Das *Papier* für jedes Bonbon wird *probabilistisch* ausgewählt – abhängig von der Geschmacksrichtung nach einer unbekannten bedingten Verteilung. ▶ Abbildung 20.2(b) zeigt das entsprechende Wahrscheinlichkeitsmodell. Beachten Sie, dass es drei Parameter hat: θ, θ_1 und θ_2. Mit diesen Parametern kann die Wahrscheinlichkeit, beispielsweise ein Kirschbonbon in einem grünen Papier zu sehen, aus der Standardsemantik für Bayessche Netze (siehe Abbildung 20.2) ermittelt werden:

$$
\begin{aligned}
&P\left(Geschmack = kirsche, Papier = grün \mid h_{\theta,\theta_1,\theta_2}\right) \\
&= P\left(Geschmack = kirsche \mid h_{\theta,\theta_1,\theta_2}\right) P\left(Papier = grün \mid Geschmack = kirsche, h_{\theta,\theta_1,\theta_2}\right) \\
&= \theta \cdot (1-\theta_1).
\end{aligned}
$$

Jetzt packen wir N Bonbons aus, wovon c Kirsche und ℓ Zitrone ist. Die Papierzähler sehen wie folgt aus: r_c der Kirschbonbons haben rotes Papier, g_c haben grünes, während r_ℓ der Zitronenbonbons rotes, g_ℓ grünes Papier haben. Die Wahrscheinlichkeit der Daten ist gegeben durch

$$P\left(\mathbf{d} \mid h_{\theta,\theta_1,\theta_2}\right) = \theta^c (1-\theta)^\ell \cdot \theta_1^{r_c} (1-\theta_1)^{g_c} \cdot \theta_2^{r_\ell} (1-\theta_2)^{g_\ell}.$$

Das sieht ziemlich schrecklich aus, aber wir können die Logarithmen zu Hilfe nehmen:

$$L = \left[c \log\theta + \ell \log(1-\theta)\right] + \left[r_c \log\theta_1 + g_c \log(1-\theta_1)\right] + \left[r_\ell \log\theta_2 + g_\ell \log(1-\theta_2)\right].$$

Der Vorteil der Logarithmierung ist offensichtlich: Die Log-Wahrscheinlichkeit ist die Summe der drei Terme, die jeweils einen einzigen Parameter enthalten. Wenn wir für jeden Parameter Ableitungen bilden und sie gleich null setzen, erhalten wir drei unabhängige Gleichungen, die jeweils nur einen Parameter enthalten:

$$\frac{\partial L}{\partial\theta} = \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 \quad \Rightarrow \quad \theta = \frac{c}{c+\ell}$$

$$\frac{\partial L}{\partial\theta_1} = \frac{r_c}{\theta_1} - \frac{g_c}{1-\theta_1} = 0 \quad \Rightarrow \quad \theta_1 = \frac{r_c}{r_c+g_c}$$

$$\frac{\partial L}{\partial\theta_2} = \frac{r_\ell}{\theta_2} - \frac{g_\ell}{1-\theta_2} = 0 \quad \Rightarrow \quad \theta_2 = \frac{r_\ell}{r_\ell+g_\ell}.$$

Die Lösung für θ ist dieselbe wie zuvor. Die Lösung für θ_1, die Wahrscheinlichkeit, dass ein Kirschbonbon ein rotes Papier hat, ist das beobachtete Verhältnis von Kirschbonbons mit rotem Papier; Ähnliches gilt für θ_2.

Tipp

Diese Ergebnisse sind durchaus zufriedenstellend und man erkennt schnell, dass sie zu einem Bayesschen Netz erweitert werden können, dessen bedingte Wahrscheinlichkeiten als Tabellen dargestellt werden. Der wichtigste Aspekt ist, *dass bei vollständigen Daten das ML-Parameterlernproblem für ein Bayessches Netz in separate Lernprobleme zerlegt werden kann, nämlich eines für jeden Parameter.* (Siehe Übung 20.7 für den nicht tabellierten Fall, wo jeder Parameter mehrere bedingte Wahrscheinlichkeiten beeinflusst.) Der zweite Aspekt ist, dass die Parameterwerte für eine Variable bei bekannten Vorgängern genau die beobachteten Häufigkeiten der Variablenwerte für jede Einstellung der Vorgängerwerte sind. Wie zuvor müssen wir sorgfältig vorgehen, um bei sehr kleinen Datenmengen Nullen zu vermeiden.

20.2.2 Naive Bayes-Modelle

Das im Maschinenlernen wahrscheinlich am häufigsten verwendete Bayessche Netz ist das **naive Bayes-Modell**, das *Abschnitt 13.5.2* eingeführt hat. Bei diesem Modell ist die „Klassen"-Variable C (die vorhergesagt werden soll) die Wurzel und die „Attribut"-Variablen X_i sind die Blätter. Das Modell ist „naiv", weil es davon ausgeht, dass die Attribute bei bekannter Klasse bedingt voneinander unabhängig sind. (Das Modell in Abbildung 20.2(b) ist ein naives Bayes-Modell mit der Klasse *Geschmack* und nur einem Attribut, *Papier*.) Wir gehen von booleschen Variablen aus und haben die folgenden Parameter:

$\theta = P(C = true)$, $\theta_{i1} = P(X_i = true \mid C = true)$, $\theta_{i2} = P(X_i = true \mid C = false)$.

Die ML-Parameterwerte werden auf genau dieselbe Weise ermittelt wie für Abbildung 20.2(b). Nachdem das Modell auf diese Weise trainiert wurde, kann es verwendet werden, um neue Beispiele zu klassifizieren, für die die Klassenvariable C unbeobachtet ist. Mit beobachteten Attributwerten $x_1, ..., x_n$ ist die Wahrscheinlichkeit jeder Klasse gegeben durch

$$\mathbf{P}(C \mid x_1,...,x_n) = \alpha \mathbf{P}(C) \prod_i \mathbf{P}(x_i \mid C).$$

Tipp

Man kann eine deterministische Vorhersage erhalten, indem man die wahrscheinlichste Klasse wählt. ▶ Abbildung 20.3 zeigt die Lernkurve für diese Methode bei Anwendung auf das Restaurantproblem aus *Kapitel 18*. Die Methode lernt relativ gut, ist aber nicht so gut wie das Entscheidungsbaumlernen; das liegt vermutlich daran, dass die wahre Hypothese – wobei es sich um einen Entscheidungsbaum handelt – mithilfe eines naiven Bayes-Modells nicht exakt dargestellt werden kann. Naives Bayes-Lernen bringt jedoch eine überraschend gute Leistung für einen großen Anwendungsbereich; die Boosted-Version (Übung 20.5) gehört zu den effektivsten universellen Lernalgorithmen. Naives Bayessches Lernen kann auch auf sehr große Probleme erweitert werden: Bei n booleschen Attributen gibt es genau $2n + 1$ Parameter und *es ist keine Suche erforderlich, um h_{ML} zu ermitteln, die naive Bayes-ML-Hypothese*. Darüber hinaus hat naives Bayessches Lernen keine Schwierigkeiten mit verrauschten oder fehlenden Daten und kann gegebenenfalls probabilistische Vorhersagen treffen.

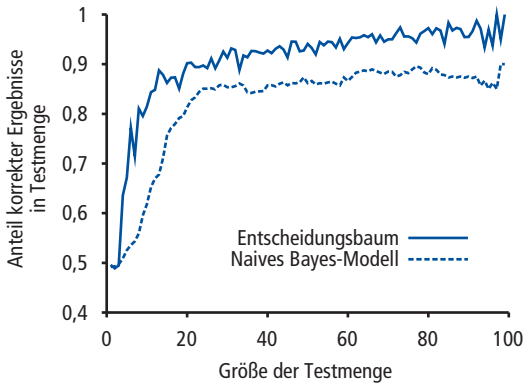


Abbildung 20.3: Die Lernkurve für naives Bayessches Lernen, gezeigt am Restaurantproblem aus *Kapitel 18*; zum Vergleich ist auch die Lernkurve für Entscheidungsbaumlernen angegeben.

20.2.3 ML-Parameterlernen: stetige Modelle

Stetige Wahrscheinlichkeitsmodelle wie etwa das **lineare Gaußsche** Modell wurden in *Abschnitt 14.3* eingeführt. Weil es in Anwendungen der realen Welt überall stetige Variablen gibt, muss man wissen, wie man die Parameter stetiger Modelle aus Daten lernt. Die Prinzipien für das ML-Lernen sind für den stetigen und den diskreten Fall identisch.

Wir beginnen mit einem sehr einfachen Fall: Lernen der Parameter einer Gaußschen Dichtefunktion für eine einzelne Variable. Das heißt, die Daten werden wie folgt erzeugt:

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}.$$

Die Parameter dieses Modells sind der Mittelwert μ und die Standardabweichung σ. (Beachten Sie, dass die Normalisierungs-„Konstante“ von σ abhängig ist, deshalb können wir sie nicht ignorieren.) Seien die beobachteten Werte gleich x_1, ..., x_N. Die Log-Wahrscheinlichkeit ist damit:

$$L = \sum_{j=1}^{N} \log \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x_j-\mu)^2}{2\sigma^2}} = N\left(-\log\sqrt{2\pi} - \log\sigma\right) - \sum_{j=1}^{N} \frac{(x_j-\mu)^2}{2\sigma^2}.$$

Setzt man wie üblich die Ableitungen gleich null, erhält man:

$$\frac{\partial L}{\partial \mu} = -\frac{1}{\sigma^2}\sum_{j=1}^{N}(x_j-\mu) = 0 \quad \Rightarrow \quad \mu = \frac{\sum_j x_j}{N}$$

$$\frac{\partial L}{\partial \sigma} = -\frac{N}{\sigma} + \frac{1}{\sigma^3}\sum_{j=1}^{N}(x_j-\mu)^2 = 0 \quad \Rightarrow \quad \sigma = \sqrt{\frac{\sum_j (x_j-\mu)^2}{N}}\,. \tag{20.4}$$

Das bedeutet, der ML-Wert des Mittelwertes ist der Stichprobendurchschnitt und der ML-Wert der Standardabweichung ist die Quadratwurzel der Stichprobenvarianz. Auch dies sind zufriedenstellende Ergebnisse, die die Praxis des „gesunden Menschenverstandes“ bestätigen.

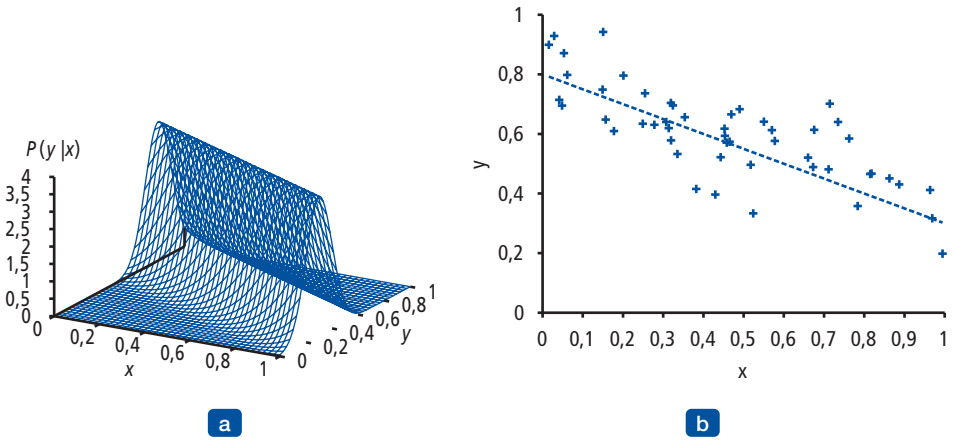


Abbildung 20.4: (a) Ein lineares Gaußsches Modell, das als $y = \theta_1 x + \theta_2$ plus Gaußschem Rauschen mit fester Varianz beschrieben ist. (b) Eine Menge von 50 aus diesem Modell erzeugten Datenpunkten.

Jetzt betrachten wir ein lineares Gaußsches Modell mit einem stetigen Elternteil X und einem stetigen Kind Y. Wie in *Abschnitt 14.3* erklärt, hat Y eine Gaußsche Verteilung, deren Mittelwert linear von dem Wert von X abhängig ist und deren Standardabweichung feststehend ist. Um die bedingte Verteilung $P(Y|X)$ zu lernen, können wir die bedingte Wahrscheinlichkeit maximieren:

$$P(y \mid x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(y-(\theta_1 x+\theta_2))^2}{2\sigma^2}}\,. \tag{20.5}$$

Hier fungieren θ, θ_1 und σ als Parameter. Die Daten sind eine Sammlung von (x_j, y_j)-Paaren, wie in ▶ Abbildung 20.4 gezeigt. Unter Verwendung der üblichen Methoden (Übung 20.6) können wir ML-Werte der Parameter finden. Hier geht es aber um etwas anderes. Wenn wir nur die Parameter θ_1 und θ_2 betrachten, die die lineare Beziehung zwischen x und y definieren, wird deutlich, dass die Maximierung der Log-Wahrscheinlichkeit im Hinblick auf diese Parameter dasselbe ist wie die *Minimierung* des Zählers $(y - (\theta_1 x + \theta_2))^2$ im Exponenten von Gleichung (20.5). Dies ist der L_2-Verlust – der quadratische Fehler zwischen dem tatsächlichen Wert y und der Vorhersage $\theta_1 x + \theta_2$. Dabei handelt es sich um die Größe, die durch die in *Abschnitt 18.6* beschriebene Standardprozedur der **linearen Regression** minimiert wird. Jetzt verstehen wir, warum: Die Minimierung der Summe der Fehlerquadrate ergibt das geradlinige ML-Modell, *vorausgesetzt, die Daten werden mit Gaußschem Rauschen fester Varianz erzeugt.*

20.2.4 Bayessches Parameterlernen

ML-Lernen führt zu einigen sehr einfachen Prozeduren, weist aber für kleine Datenmengen einige ernsthafte Nachteile auf. Nachdem man beispielsweise ein Kirschbonbon gesehen hat, ist die ML-Hypothese, dass es sich um eine Tüte mit 100% Kirsche handelt (d.h. $\theta = 1{,}0$). Sofern die A-priori-Hypothese nicht lautet, dass Tüten entweder nur Kirsche oder nur Zitrone enthalten müssen, ist das keine sinnvolle Schlussfolgerung. Wahrscheinlicher ist, dass die Tüte eine Mischung aus Zitrone und Kirsche enthält. Der Bayessche Ansatz für das Parameterlernen definiert zunächst eine A-priori-Wahrscheinlichkeitsverteilung über den möglichen Hypothesen. Wir bezeichnen dies als die **A-priori-Hypothese**. Wenn dann Daten eintreffen, wird die A-posteriori-Wahrscheinlichkeitsverteilung aktualisiert.

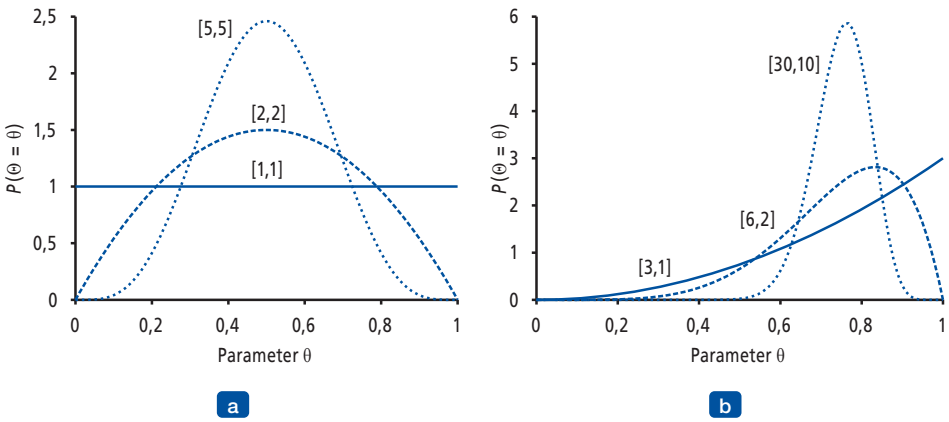


Abbildung 20.5: Beispiele für die Beta[*a*, *b*]-Verteilung für verschiedene Werte von [*a*, *b*].

Das Bonbonbeispiel in Abbildung 20.2(a) hat einen Parameter, θ: die Wahrscheinlichkeit, dass ein zufällig ausgewähltes Bonbon die Geschmacksrichtung Kirsche hat. In Bayesscher Perspektive ist θ der (unbekannte) Wert einer Zufallsvariablen Θ, die den Hypothesenraum definiert; die A-priori-Hypothese ist einfach die A-priori-Verteilung $\mathbf{P}(\Theta)$. Damit ist $P(\Theta = \theta)$ die A-priori-Wahrscheinlichkeit, dass die Tüte einen Anteil θ von Kirschbonbons enthält.

Wenn der Parameter θ ein Wert zwischen 0 und 1 sein kann, muss $\mathbf{P}(\Theta)$ eine stetige Verteilung sein, die nur zwischen 0 und 1 ungleich null ist und deren Integral 1 ist. Die Dichte der Gleichverteilung $P(\theta) = \mathit{Gleichförmig}[0, 1](\theta)$ ist ein Kandidat (siehe *Kapitel 13*). Es zeigt sich, dass die Gleichverteilung ein Element der Familie der **Betaverteilungen** ist. Jede Betaverteilung ist durch zwei **Hyperparameter**,[3] a und b, definiert, sodass

$$\mathit{Beta}[a,b](\theta) = \alpha\theta^{a-1}(1-\theta)^{b-1} \tag{20.6}$$

gilt, wobei θ im Bereich [0,1] liegt. Die Normalisierungskonstante α, die dafür zuständig ist, dass das Integral der Verteilung 1 ist, hängt von a und b ab (siehe Übung 20.8). ▶ Abbildung 20.5 zeigt, wie die Verteilung für verschiedene Werte von a und b aussieht. Der Mittelwert der Verteilung ist $a/(a+b)$; größere Werte von a führen also zu dem Glauben, dass Θ näher bei 1 als bei 0 liegt. Größere Werte von $a + b$ machen die Verteilung spitzer, d.h., es herrscht eine größere Sicherheit in Bezug auf den Wert von Θ. Die Beta-Familie weist also einen sinnvollen Bereich von Möglichkeiten für die A-priori-Hypothesenverteilung auf.

Neben ihrer Flexibilität hat die Beta-Familie noch eine andere wunderbare Eigenschaft: Wenn Θ eine A-priori-Verteilung $\mathit{beta}[a, b]$ hat, dann ist, nachdem ein Datenpunkt beobachtet wurde, die A-posteriori-Verteilung für Θ ebenfalls eine Betaverteilung. Mit anderen Worten ist die Betaverteilung unter der gegebenen Updatefunktion abgeschlossen. Die Beta-Familie wird als die **konjugierte A-priori-Verteilung** für die Familie der Verteilungen für eine boolesche Variable bezeichnet.[4] Jetzt betrachten wir, wie das funktioniert. Angenommen, wir beobachten ein Kirschbonbon, dann ist

$$\begin{aligned} P(\theta \mid D_1 = \mathit{kirsche}) &= \alpha P(D_1 = \mathit{kirsche} \mid \theta) P(\theta) \\ &= \alpha' \theta \cdot \mathit{beta}[a,b](\theta) = \alpha' \theta \cdot \theta^{a-1}(1-\theta)^{b-1} \\ &= \alpha' \theta^{a}(1-\theta^{b-1})(\mathit{beta}[a+1,b])(\theta). \end{aligned}$$

Nachdem wir also ein Kirschbonbon sehen, inkrementieren wir einfach den Parameter a, um die A-posteriori-Verteilung zu erhalten; nachdem wir ein Zitronenbonbon sehen, inkrementieren wir den Parameter b. Damit können wir die Hyperparameter a und b als **virtuelle Zähler** betrachten, und zwar in dem Sinne, dass sich eine A-priori-Beta[a, b]-Verteilung genauso verhält, als hätten wir mit einer gleichförmigen A-priori-Beta[1, 1]-Verteilung begonnen und $a - 1$ wirkliche Kirschbonbons und $b - 1$ wirkliche Zitronenbonbons gesehen.

Durch die Betrachtung einer Folge von Betaverteilungen für steigende Werte von a und b und gleichbleibenden Proportionen sehen wir deutlich, wie sich die A-posteriori-Verteilung über den Parameter Θ ändert, wenn Daten eintreffen. Nehmen Sie zum Beispiel an, die Bonbontüte enthält zu 75% Kirsche. Abbildung 20.5(b) zeigt die Folge beta[2,1], beta[6,2] und beta[30,10]. Offensichtlich konvergiert die Verteilung zu einer schmalen Spitze um den wahren Wert von Θ. Für große Datenmengen konvergiert also das Bayessche Lernen (zumindest in diesem Fall) zur selben Antwort wie das ML-Lernen.

3 Sie werden als Hyperparameter bezeichnet, weil sie eine Verteilung über θ parametrisieren, was selbst wiederum ein Parameter ist.

4 Andere konjugierte A-priori-Verteilungen sind etwa die **Dirichlet**-Familie für die Parameter einer diskreten mehrwertigen Verteilung oder die **Normal-Wishart**-Familie für die Parameter einer Gaußschen Verteilung. Siehe Bernardo und Smith (1994).

Sehen wir uns nun einen komplexeren Fall an. Das Netz in Abbildung 20.2(b) hat drei Parameter: θ, θ_1 und θ_2, wobei θ_1 die Wahrscheinlichkeit angibt, dass ein rotes Papier um ein Kirschbonbon gewickelt ist, und θ_2 für die Wahrscheinlichkeit steht, dass ein rotes Papier um ein Zitronenbonbon gewickelt ist. Die Bayessche A-priori-Hypothesenverteilung muss alle drei Parameter abdecken – d.h., wir müssen $\mathbf{P}(\Theta, \Theta_1, \Theta_2)$ spezifizieren. In der Regel gehen wir von **Parameterunabhängigkeit** aus:

$$\mathbf{P}(\Theta, \Theta_1, \Theta_2) = \mathbf{P}(\Theta)\mathbf{P}(\Theta_1)\mathbf{P}(\Theta_2).$$

Mit dieser Annahme kann jeder Parameter seine eigene Betaverteilung haben, die separat aktualisiert wird, sobald Daten eintreffen. ▶ Abbildung 20.6 zeigt, wie wir die A-priori-Hypothese und alle Daten in ein Bayessches Netz einbinden können. Die Knoten Θ, Θ_1 und Θ_2 haben keine übergeordneten Knoten. Doch jedes Mal, wenn wir eine Beobachtung zu einem Papier und der entsprechenden Geschmacksrichtung eines Bonbons machen, fügen wir einen Knoten $Geschmack_i$ hinzu, der abhängig vom Geschmacksparameter Θ ist:

$$P(Geschmack_i = kirsche | \Theta = \theta) = \theta.$$

Außerdem fügen wir einen Knoten *Papier* hinzu, der von Θ_1 und Θ_2 abhängt:

$$P(Papier_i = rot | Geschmack_i = kirsche, \Theta_1 = \theta_1) = \theta_1$$
$$P(Papier_i = rot | Geschmack_i = zitrone, \Theta_2 = \theta_2) = \theta_2 .$$

Tipp

Jetzt kann der gesamte Bayessche Lernprozess als *Inferenzproblem* formuliert werden. Wir fügen neue Evidenzknoten hinzu und fragen dann die unbekannten Knoten ab (in diesem Fall Θ, Θ_1 und Θ_2). Diese Formulierung von Lernen und Vorhersage verdeutlicht, dass das Bayessche Lernen keine zusätzlichen „Lernkonzepte" benötigt. Darüber hinaus *gibt es im Wesentlichen nur einen Lernalgorithmus* – den Inferenzalgorithmus für Bayessche Netze. Selbstverständlich unterscheidet sich das Wesen dieser Netze etwas von denen, die *Kapitel 14* beschrieben hat, aufgrund der möglicherweise riesigen Anzahl von Evidenzvariablen, die die Trainingsmenge darstellen, und der Dominanz von Parametervariablen mit stetigen Werten.

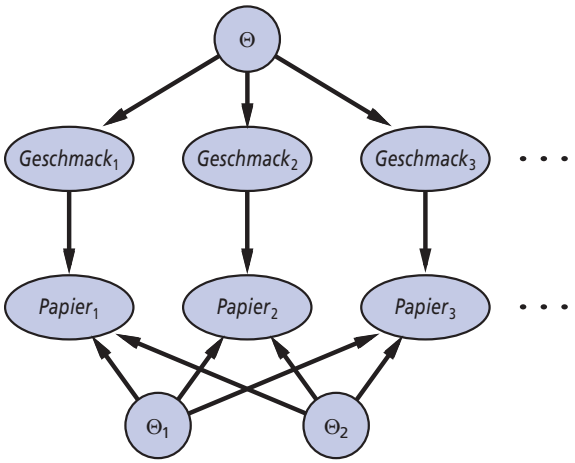


Abbildung 20.6: Ein Bayessches Netz, das einem Bayesschen Lernprozess entspricht. A-posteriori-Verteilungen für die Parametervariablen Θ, Θ_1, Θ_2 können von ihren A-priori-Verteilungen und der Evidenz in den Variablen $Geschmack_i$ und $Papier_i$ abgeleitet werden.

20.2.5 Strukturen Bayesscher Netze lernen

Bisher sind wir davon ausgegangen, dass die Struktur eines Bayesschen Netzes bekannt ist und wir nur versuchen, die Parameter zu lernen. Die Struktur des Netzes repräsentiert grundlegendes Wissen über die Domäne, das ein Experte und zum Teil auch ein unerfahrener Benutzer leicht bereitstellen kann. In einigen Fällen steht das kausale Modell jedoch nicht zur Verfügung oder ist umstritten – beispielsweise haben bestimmte Unternehmen lange Zeit behauptet, Rauchen erzeuge keinen Krebs –, deshalb ist es wichtig zu verstehen, wie die Struktur eines Bayesschen Netzes aus den Daten gelernt werden kann. Dieser Abschnitt gibt einen kurzen Überblick über die wichtigsten Konzepte.

Der offensichtlichste Ansatz ist, nach einem guten Modell zu *suchen*. Wir können mit einem Modell beginnen, das keine Verknüpfungen enthält, und Eltern für jeden Knoten hinzufügen, wobei wir mit den eben beschriebenen Methoden Übereinstimmungen mit den Parametern herbeiführen und die Genauigkeit des resultierenden Modells messen. Alternativ können wir zunächst eine Schätzung für die Struktur abgeben und dann mithilfe von Hillclimbing oder Simulated Annealing Änderungen vornehmen, wobei wir nach jeder Änderung in der Struktur die Parameter zurückgeben. Änderungen können das Umkehren, das Hinzufügen oder das Entfernen von Kanten sein. Wir dürfen keine Schleifen erzeugen; deshalb gehen viele Algorithmen davon aus, dass die Variablen in einer bestimmten Reihenfolge vorliegen und dass ein Knoten nur in solchen Knoten Eltern haben kann, die in der Reihenfolge vor ihm liegen (genau wie im Konstruktionsprozess, den *Kapitel 14* beschrieben hat). Im Sinne einer vollständigen Verallgemeinerung müssen wir auch über mögliche Reihenfolgen suchen.

Es gibt zwei alternative Methoden, zu entscheiden, wann eine gute Struktur gefunden ist. Die erste überprüft, ob die in der Struktur implizit vorhandenen bedingten Unabhängigkeitsbehauptungen in den Daten erfüllt sind. Beispielsweise nimmt die Verwendung eines naiven Bayes-Modells für das Restaurantproblem an, dass

$$\mathbf{P}(Frei/Sams, Bar|WerdenWarten) = \mathbf{P}(Frei/Sams|WerdenWarten)\,\mathbf{P}(Bar|WerdenWarten)$$

ist und wir die Daten daraufhin überprüfen können, ob dieselbe Gleichung zwischen den entsprechenden bedingten Häufigkeiten besteht. Doch selbst wenn die Struktur die kausale Natur der Domäne beschreibt, bedeuten statistische Schwankungen in der Datenmenge, dass die Gleichung nie *genau* erfüllt ist. Wir müssen also einen geeigneten statistischen Test ausführen, um zu überprüfen, ob es ausreichend viel Evidenz gibt, dass die Unabhängigkeitshypothese verletzt ist. Die Komplexität des resultierenden Netzes ist von dem für den Test verwendeten Schwellenwert abhängig – je strenger der Unabhängigkeitstest, desto mehr Verknüpfungen werden hinzugefügt und desto größer ist die Gefahr der Überanpassung.

Ein Ansatz, der den in diesem Kapitel beschriebenen Ideen näher kommt, bewertet den Grad, bis zu dem das vorgeschlagene Modell die Daten (im probabilistischen Sinne) erklärt. Wir müssen jedoch sorgfältig überlegen, wie wir dies messen. Wenn wir nur versuchen, die ML-Hypothese zu finden, werden wir ein vollständig verknüpftes Netz erhalten, weil das Hinzufügen von mehr Eltern für einen Knoten die Wahrscheinlichkeit nicht

verringern kann (Übung 20.9). Wir sind gezwungen, die Modellkomplexität irgendwie mit einer Strafe zu belegen. Der MAP-Ansatz (oder MDL-Ansatz) subtrahiert eine Bestrafung einfach von der Wahrscheinlichkeit jeder Struktur (nach der Anpassung der Parameter), bevor unterschiedliche Strukturen verglichen werden. Der Bayessche Ansatz legt A-priori-Verknüpfung über Strukturen und Parameter. Es gibt in der Regel viel zu viele Strukturen, als dass diese summiert werden könnten (superexponentiell in der Anzahl der Variablen); deshalb verwenden die meisten Praktiker MCMC, um Stichproben aus den Strukturen zu ziehen.

Die Bestrafung der Komplexität (egal ob durch MAP oder Bayessche Methoden) führt eine wichtige Verbindung zwischen der optimalen Struktur und der Natur der Repräsentation für die bedingten Verteilungen im Netz ein. Mit tabellierten Verteilungen wächst die Komplexitätsstrafe für die Verteilung eines Knotens exponentiell mit der Anzahl der Elternknoten, aber mit beispielsweise Noisy-OR-Verteilungen wächst sie nur linear. Das bedeutet, Lernen mit Noisy-OR (oder anderen kompakt parametrisierten) Modellen erzeugt häufig gelernte Strukturen mit mehr Eltern als das Lernen mit tabellierten Verteilungen.

20.2.6 Dichteabschätzung mit parameterfreien Modellen

Mit den parameterfreien Methoden von *Abschnitt 18.8* lässt sich ein Wahrscheinlichkeitsmodell auch lernen, ohne Annahmen über seine Struktur und Parametrisierung zu treffen. Die Aufgabe einer **parameterfreien Dichteabschätzung** geschieht normalerweise in stetigen Domänen, wie sie zum Beispiel in ▶ Abbildung 20.7(a) zu sehen ist. Die Abbildung zeigt eine Wahrscheinlichkeitsdichtefunktion in einem Raum, der durch zwei stetige Variablen definiert ist. ▶ Abbildung 20.7(b) stellt eine Stichprobe der Datenpunkte für diese Dichtefunktion dar. Die Frage lautet: Wie können wir das Modell aus diesen Stichproben rekonstruieren?

Zuerst betrachten wir *k*-**nächste-Nachbarn-Modelle**. (*Kapitel 18* hat Nächste-Nachbarn-Modelle zur Klassifizierung und Regression vorgestellt; hier verwenden wir sie zur Dichteabschätzung. Um bei einer gegebenen Stichprobe von Datenpunkten die unbekannte Wahrscheinlichkeitsdichte an einem Abfragepunkt x zu schätzen, können wir einfach die Dichte der Datenpunkte in der Nachbarschaft von x messen. Abbildung 20.7(b) zeigt zwei Abfragepunkte (kleine Quadrate). Für jeden Abfragepunkt haben wir die kleinsten Kreise gezogen, die zehn Nachbarn umschließen – die 10-nächste-Nachbarschaft. Der mittlere Kreis ist groß, was eine geringe Dichte in diesem Bereich bedeutet. Der rechte Kreis ist klein und weist damit auf eine hohe Dichte in diesem Gebiet hin. ▶ Abbildung 20.8 zeigt drei Grafiken von Dichteabschätzungen, die mit *k*-nächsten-Nachbarn für verschiedene Werte von *k* erstellt wurden. Es scheint klar zu sein, dass (b) ungefähr richtig ist, während (a) zu spitz (*k* zu klein) und (c) zu glatt (*k* zu groß) ist.

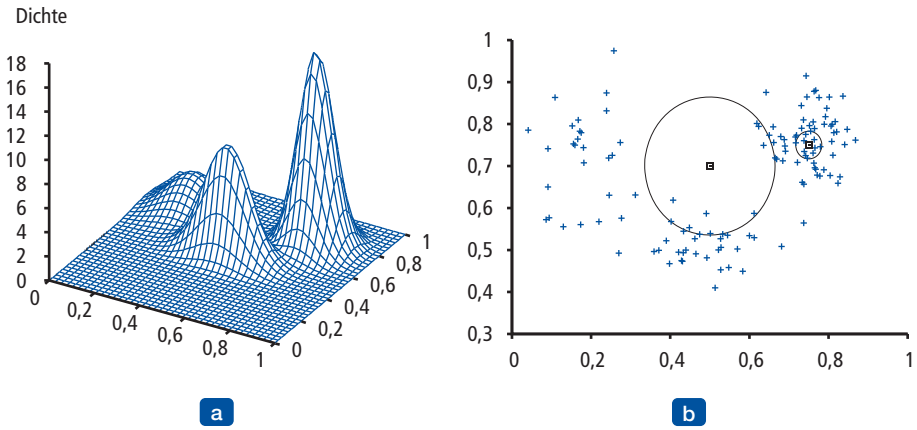


Abbildung 20.7: (a) Eine 3D-Grafik der Mischung von Gaußverteilungen von ▶ Abbildung 20.11(a). (b) Eine Stichprobe von 128-Punkten aus der Mischung zusammen mit zwei Abfragepunkten (kleine Quadrate) und den 10-nächsten-Nachbarschaften (mittlere und große Kreise).

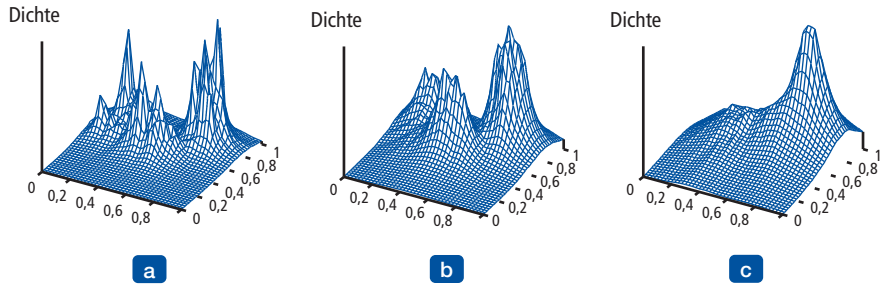


Abbildung 20.8: Dichteabschätzung unter Verwendung k-nächster-Nachbarn, angewandt auf die Daten in Abbildung 20.7(b), für $k = 3$, 10 und 40. Bei $k = 3$ ist die Schätzung zu spitz, bei 40 zu glatt und bei 10 gerade richtig. Der beste Wert für k lässt sich durch Kreuzvalidierung auswählen.

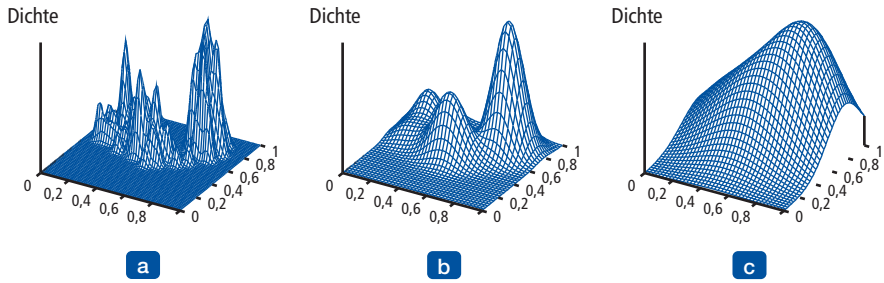


Abbildung 20.9: Kernel-Dichteabschätzung für die Daten in Abbildung 20.7(b) unter Verwendung Gaußscher Kernel mit $w = 0{,}02$, $0{,}07$ und $0{,}20$. $w = 0{,}07$ ist ungefähr richtig.

Als weitere Möglichkeit kann man **Kernel-Funktionen** verwenden, wie wir es bei lokal gewichteter Regression getan haben. Um ein Kernel-Modell auf eine Dichteabschätzung anzuwenden, nehmen wir an, dass jeder Datenpunkt seine eigene kleine Dichtefunktion unter Verwendung eines Gaußschen Kernels generiert. Die geschätzte

Dichte an einem Abfragepunkt $\mathbf{x}$ ist dann die mittlere Dichte, wie sie durch jede Kernel-Funktion gegeben ist:

$$P(\mathbf{x}) = \frac{1}{N}\sum_{j=1}^{N}\kappa(\mathbf{x},\mathbf{x}_j).$$

Wir nehmen sphärische Gaußverteilungen mit der Standardabweichung w entlang jeder Achse an:

$$\kappa(\mathbf{x},\mathbf{x}_j) = \frac{1}{\left(w^2\sqrt{2\pi}\right)^d} e^{-\frac{D(x,x_j)^2}{2w^2}},$$

wobei d die Anzahl der Dimensionen in $\mathbf{x}$ angibt und D die Funktion für den Euklidischen Abstand ist. Wir haben immer noch das Problem, einen geeigneten Wert für die Kernel-Breite w zu wählen; ▶ Abbildung 20.9 zeigt Werte, die zu klein, gerade richtig und zu groß sind. Ein guter Wert von w lässt sich durch Kreuzvalidierung auswählen.

20.3 Lernen mit verborgenen Variablen: der EM-Algorithmus

Tipp

Der vorige Abschnitt hat sich mit dem vollständig beobachtbaren Fall beschäftigt. Viele Probleme aus der realen Welt weisen **verborgene Variablen** auf (manchmal auch als **latente Variablen** bezeichnet), welche nicht in den Daten, die für das Lernen zur Verfügung stehen, beobachtet werden können. Medizinische Aufzeichnungen beispielsweise beinhalten häufig die beobachteten Symptome, die Diagnose des Mediziners, die angewandte Behandlung und vielleicht das Behandlungsergebnis, aber selten enthalten sie eine direkte Beobachtung der eigentlichen Krankheit. (Die *Diagnose* ist nicht die *Krankheit*; sie ist eine kausale Konsequenz der beobachteten Symptome, die wiederum durch die Krankheit verursacht werden.) Man könnte fragen: „Wenn die Krankheit nicht beobachtet wird, warum erzeugt man dann nicht ein Modell ohne sie?“ Die Antwort erkennen Sie in ▶ Abbildung 20.10, die ein kleines, fiktives Diagnosemodell für Herzkrankheiten zeigt. Es gibt drei beobachtete prädisponierende Faktoren und drei beobachtbare Symptome (deren namentliche Nennung zu deprimierend wäre). Angenommen, jede Variable hat drei mögliche Werte (z.B. *nein*, *mittel* und *schwer*). Durch das Entfernen der verborgenen Variablen aus dem Netz in (a) ergibt sich das Netz in (b). Die Gesamtzahl der Parameter steigt von 78 auf 708. *Die latenten Variablen können also die Anzahl der Parameter, die für die Spezifizierung eines Bayesschen Netzes erforderlich sind, wesentlich reduzieren.* Das wiederum kann die Menge der Daten, die für das Lernen der Parameter erforderlich sind, wesentlich reduzieren.

Verborgene Variablen sind wichtig, aber sie verkomplizieren das Lernproblem. In Abbildung 20.10(a) beispielsweise ist nicht offensichtlich, wie die bedingte Verteilung für *Herzkrankheit* bei bekannten Eltern gelernt werden soll, weil wir die Werte für *Herzkrankheit* nicht für jeden Fall kennen; das gleiche Problem entsteht beim Lernen der Verteilungen für die Symptome. Dieser Abschnitt beschreibt einen Algorithmus, **Erwartung-Maximierung** (**Expectation-Maximization**), **EM**, der dieses Problem auf sehr allgemeine Weise löst. Wir werden drei Beispiele vorstellen und dann eine allgemeine Beschreibung formulieren. Der Algorithmus erscheint auf den ersten Blick wie

Zauberei, aber nachdem man ein Gefühl dafür entwickelt hat, findet man Anwendungen für EM in vielen Bereichen für Lernprobleme.

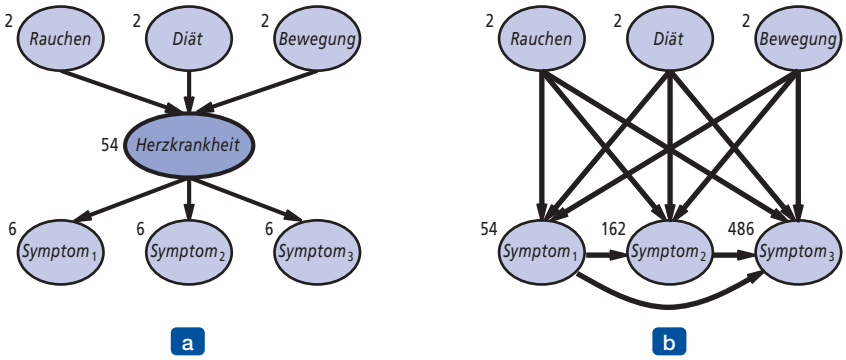


Abbildung 20.10: (a) Ein einfaches Diagnosenetz für *Herzkrankheit*, wobei es sich um eine verborgene Variable handeln soll. Jede Variable hat drei mögliche Werte und ist mit der Anzahl der unabhängigen Parameter in ihrer bedingten Verteilung beschriftet; die Gesamtzahl ist 78. (b) Das äquivalente Netz, nachdem *Herzkrankheit* entfernt wurde. Beachten Sie, dass die Symptomvariablen nicht mehr bedingt unabhängig sind, wenn man ihre Eltern kennt. Für dieses Netz sind 708 Parameter erforderlich.

20.3.1 Nicht überwachtes Clustering: Gaußsche Mischungen lernen

Nicht überwachtes Clustering ist das Problem, mehrere Kategorien in einer Sammlung von Objekten zu erkennen. Das Problem ist nicht überwacht, weil keine Kategoriebeschriftungen vorgegeben sind. Angenommen, wir zeichnen die Spektren von hunderttausend Sternen auf. Gibt es verschiedene *Sterntypen*, die von den Spektren aufgedeckt werden, und wenn ja, wie viele Typen gibt es und welche Eigenschaften haben sie? Wir kennen alle Begriffe wie „roter Riese“ oder „weißer Zwerg“, aber die Sterne sind nicht mit diesen Bezeichnungen beschriftet – Astronomen mussten ein nicht überwachtes Clustering durchführen, um diese Kategorien zu identifizieren. Andere Beispiele beinhalten die Identifikation von Spezies, Gattungen, Ordnungen usw. in der Taxonomie von Linné und der Schaffung natürlicher Arten von Ordnungsobjekten (siehe *Kapitel 12*).

Nicht überwachtes Clustering beginnt mit Daten. Abbildung 20.11(a) zeigt 500 Datenpunkte, die jeweils die Werte zweier stetiger Attribute spezifizieren. Die Datenpunkte könnten Sternen und die Attribute Spektralintensitäten bei zwei bestimmten Frequenzen entsprechen. Als Nächstes müssen wir verstehen, welche Art Wahrscheinlichkeitsverteilung die Daten erzeugt haben könnte. Das Clustering geht davon aus, dass die Daten von einer **gemischten Verteilung** P erzeugt werden. Eine solche Verteilung hat k **Komponenten**, die jeweils wieder selbst eine Verteilung darstellen. Ein Datenpunkt wird erzeugt, indem zuerst eine Komponente ausgewählt und dann eine Stichprobe aus dieser Komponente erzeugt wird. Es soll die Zufallsvariable C die Komponente bezeichnen, die die Werte 1, ..., k hat. Die gemischte Verteilung ist dann gegeben durch

$$P(\mathbf{x}) = \sum_{i=1}^{k} P(C=i)P(\mathbf{x} \mid C=i),$$

wobei **x** die Werte der Attribute für einen Datenpunkt darstellt. Für stetige Daten ist eine naheliegende Auswahl für die Komponentenverteilung die multivariate Gaußsche Verteilung, die die sogenannte Familie der **gemischten Gaußschen Verteilungen** bildet. Die Parameter einer Mischung Gaußscher Verteilungen sind $w_i = P(C = i)$ (das Gewicht jeder Komponente), μ_i (der Mittelwert jeder Komponente) und Σ_i (die Kovarianz jeder Komponente). Abbildung 20.11(b) zeigt eine Mischung aus drei Gaußschen Verteilungen; diese Mischung ist letztlich die Quelle für die Daten in (b) sowie das Modell, das weiter vorn in Abbildung 20.7 gezeigt wurde.

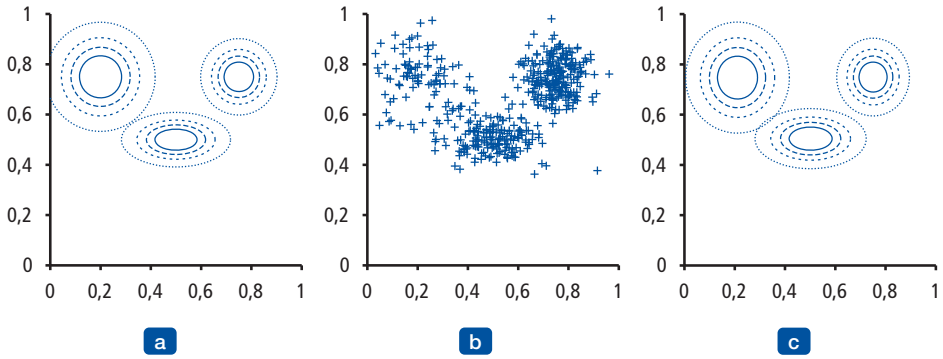


Abbildung 20.11: (a) Modell einer Gaußschen Mischverteilung mit drei Komponenten; die Gewichte haben die Werte 0,2, 0,3 und 0,5 (von links nach rechts). (b) Stichprobe mit 500 Datenpunkten des Modells in (a). (c) Das durch EM aus den Daten in (b) rekonstruierte Modell.

Das nicht überwachte Clustering-Problem besteht dann darin, ein Mischungsmodell wie das in Abbildung 20.11(a) gezeigte aus den Rohdaten wie in Abbildung 20.11(b) zu rekonstruieren. Wenn wir *wüssten*, welche Komponenten die einzelnen Datenpunkte erzeugt haben, wäre es natürlich einfach, die Gaußschen Komponentenverteilungen zu rekonstruieren: Wir könnten einfach alle Datenpunkte von einer bestimmten Komponente auswählen und dann (eine multivariate Version von) Gleichung (20.4) anwenden, um die Parameter einer Gaußschen Verteilung an die Datenmenge anzupassen. Würden wir dagegen die Parameter jeder Komponente *kennen*, könnten wir (zumindest in probabilistischer Hinsicht) jeden Datenpunkt einer Komponente zuordnen. Wir kennen allerdings weder die Zuordnungen noch die Parameter.

Das Grundprinzip von EM in diesem Kontext ist, zu *behaupten*, dass wir die Parameter des Modells kennen, und dann die Wahrscheinlichkeit dafür abzuleiten, dass bestimmte Datenpunkte zu bestimmten Komponenten gehören. Anschließend gleichen wir die Komponenten mit den Daten ab, wobei jede Komponente mit einer gesamten Datenmenge in Übereinstimmung gebracht und jeder Punkt mit der Wahrscheinlichkeit gewichtet wird, dass er zu dieser Komponente gehört. Dieser Prozess iteriert bis zur Konvergenz. Im Wesentlichen „vervollständigen“ wir die Daten, indem wir basierend auf dem aktuellen Modell Wahrscheinlichkeitsverteilungen – zu welcher Komponente jeder Datenpunkt gehört – für die verborgenen Variablen ableiten. Für die Mischung Gaußscher Verteilungen initialisieren wir die Parameter des Mischmodells zufällig und iterieren dann über die beiden folgenden Schritte:

1 **E-Schritt:** Berechnen der Wahrscheinlichkeiten $p_{ij} = P(C = i|\mathbf{x}_j)$, der Wahrscheinlichkeit, dass der Datenwert $\mathbf{x}_j$ von der Komponente i erzeugt wurde. Nach der Bayesschen Regel haben wir $p_{ij} = \alpha P(\mathbf{x}_j|C = i)P(C = i)$. Der Term $P(\mathbf{x}_j|C = i)$ ist einfach die Wahrscheinlichkeit von $\mathbf{x}_j$ der i-ten Gaußschen Verteilung und der Term $P(C = i)$ ist der Gewichtungsparameter für die i-te Gaußsche Verteilung. Definieren von $p_i = \Sigma_j p_{ij}$, der effektiven Anzahl der Datenpunkte, die momentan der Komponente i zugewiesen sind.

2 **M-Schritt:** Berechnen des neuen Mittelwertes, der neuen Kovarianz und der neuen Komponentengewichte nacheinander in den folgenden Schritten:

$$\mu_i \leftarrow \sum_j p_{ij}\mathbf{x}_j / n_i$$
$$\sum_i \leftarrow \sum_j p_{ij}\left(\mathbf{x}_j - \boldsymbol{\mu}_i\right)\left(\mathbf{x}_j - \boldsymbol{\mu}_i\right)^T / n_i$$
$$w_i \leftarrow n_i / N.$$

Hier ist N die Gesamtanzahl der Datenpunkte. Der E-Schritt oder *Erwartungsschritt* kann als Berechnung der erwarteten Werte p_{ij} der verborgenen **Indikatorvariablen** Z_{ij} betrachtet werden, wobei Z_{ij} gleich 1 ist, wenn der Datenwert $\mathbf{x}_j$ von der i-ten Komponente erzeugt wurde, andernfalls 0. Der M-Schritt oder *Maximierungsschritt* ermittelt die neuen Werte der Parameter, die die Log-Wahrscheinlichkeit der Daten für die erwarteten Werte der verborgenen Indikatorvariablen maximieren.

Abbildung 20.11(c) zeigt das fertige Modell, das EM lernt, wenn es auf die Daten aus Abbildung 20.11(a) angewendet wird; es ist so gut wie nicht von dem Originalmodell zu unterscheiden, aus dem die Daten erzeugt wurden. Die Grafik in Abbildung 20.12(a) zeigt die Log-Wahrscheinlichkeit der Daten gemäß dem aktuellen Modell, wenn EM fortgesetzt wird.

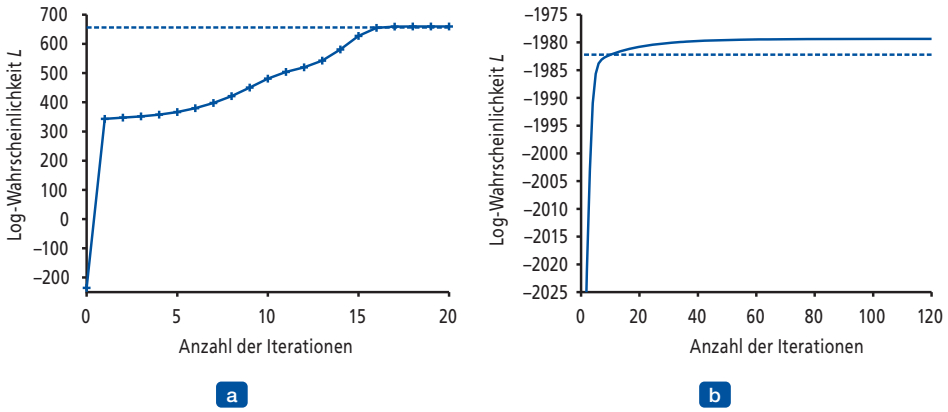


Abbildung 20.12: Graphen, die die Log-Wahrscheinlichkeit der Daten, L, als Funktion der EM-Iteration zeigen. Die horizontale Linie zeigt die Log-Wahrscheinlichkeit gemäß dem tatsächlichen Modell. (a) Graph für das gemischte Gaußsche Modell in Abbildung 20.11(b) Graph für das Bayessche Netz in ▶ Abbildung 20.13(a).

Tipp

Hier sind vor allem zwei Punkte zu beachten. Erstens *übersteigt* die Log-Wahrscheinlichkeit für das endgültig gelernte Modell geringfügig die Wahrscheinlichkeit des Originalmodells, aus dem die Daten erzeugt wurden. Das erscheint vielleicht überraschend,

spiegelt aber einfach die Tatsache wider, dass die Daten zufällig erzeugt wurden und vielleicht keine exakte Darstellung des zugrunde liegenden Modells sind. Der zweite Aspekt ist, *dass EM die Log-Wahrscheinlichkeit der Daten bei jeder Iteration erhöht*. Diese Tatsache kann allgemein bewiesen werden. Darüber hinaus kann für EM unter bestimmten Bedingungen (die in den meisten Fällen gelten) bewiesen werden, dass es ein lokales Maximum der Wahrscheinlichkeit erreicht. (In seltenen Fällen kann es auch einen Sattelpunkt oder sogar ein lokales Minimum erreichen.) In dieser Hinsicht erinnert EM an einen gradientenbasierten Hillclimbing-Algorithmus, beachten Sie jedoch, dass er keinen Parameter für die „Schrittgröße“ hat!

Es geht nicht immer alles so gut, wie es in ▶ Abbildung 20.12(a) den Anschein haben mag. Es kann beispielsweise passieren, dass eine Gaußsche Komponente so weit schrumpft, dass sie nur einen einzigen Datenpunkt abdeckt. Ihre Varianz geht dann gegen null, ihre Wahrscheinlichkeit gegen unendlich! Ein weiteres Problem ist, dass sich zwei Komponenten vermischen können, identische Mittel und Varianzen aufweisen und ihre Datenpunkte teilen. Diese Art der Degenerierung lokaler Maxima stellt ein ernsthaftes Problem dar, insbesondere in höheren Dimensionen. Als Lösung kann man A-priori-Verteilungen auf die Modellparameter legen und die MAP-Version von EM anwenden. Eine weitere Möglichkeit ist es, eine Komponente mit neuen Zufallsparametern erneut zu starten, wenn sie zu klein wird oder zu nahe an einer anderen Komponente liegt. Sinnvolle Initialisierung hilft ebenfalls.

20.3.2 Bayessche Netze mit verborgenen Variablen lernen

Um ein Bayessches Netz mit verborgenen Variablen zu lernen, wenden wir dieselben Einsichten an, die auch für die gemischten Gaußschen Verteilungen funktioniert haben. Abbildung 20.13 zeigt eine Situation, in der drei Bonbontüten vermischt wurden. Bonbons werden durch drei Merkmale beschrieben: Neben *Geschmack* und *Papier* haben einige Bonbons jetzt auch ein *Loch* in der Mitte, andere nicht.

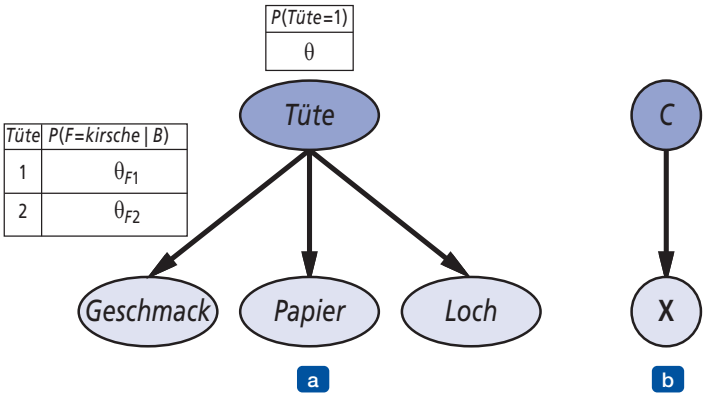


Abbildung 20.13: (a) Ein gemischtes Modell für Bonbons. Die Proportionen der verschiedenen Geschmacksrichtungen, Papiere sowie das Vorhandensein von Löchern sind von der Tüte abhängig, die nicht beobachtet wird. (b) Bayessches Netz für eine gemischte Gaußsche Verteilung. Mittelwert und Kovarianz der beobachtbaren Variablen $\mathbf{X}$ sind von der Komponente C abhängig.

Die Verteilung der Bonbons in jeder Tüte wird durch ein **naives Bayes-Modell** beschrieben: Die Merkmale sind voneinander unabhängig für die verschiedenen Tüten, aber die

bedingte Wahrscheinlichkeitsverteilung für jedes Merkmal ist von der jeweiligen Tüte abhängig. Die Parameter sehen wie folgt aus: θ ist die A-priori-Wahrscheinlichkeit, dass ein Bonbon aus Tüte 1 stammt; θ_{F1} und θ_{F2} sind die Wahrscheinlichkeiten, dass die Geschmacksrichtung Kirsche ist, wobei die Bonbons aus Tüte 1 bzw. Tüte 2 stammen; θ_{W1} und θ_{W2} geben die Wahrscheinlichkeiten an, dass das Papier rot ist; und θ_{H1} und θ_{H2} geben die Wahrscheinlichkeiten an, dass das Bonbon ein Loch hat. Beachten Sie, dass es sich bei dem Gesamtmodell um ein gemischtes Modell handelt. (Tatsächlich können wir auch die Mischung Gaußscher Verteilungen als Bayessches Netz modellieren, wie in ▶ Abbildung 20.13(b) gezeigt.) In der Abbildung ist die Tüte eine verborgene Variable, weil wir, nachdem die Bonbons vermischt wurden, nicht mehr wissen, aus welcher Tüte sie jeweils stammen. Können wir in diesem Fall die Beschreibungen der beiden Tüten rekonstruieren, indem wir die Bonbons aus der Mischung beobachten?

Betrachten wir eine Iteration von EM für dieses Problem. Zuerst sehen wir uns die Daten an. Wir haben 1000 Stichproben aus einem Modell erzeugt, dessen tatsächliche Parameter wie folgt aussehen:

$$\theta = 0{,}5, \theta_{F1} = \theta_{W1} = \theta_{H1} = 0{,}8, \theta_{F2} = \theta_{W2} = \theta_{H2} = 0{,}3. \tag{20.7}$$

Das bedeutet, die Bonbons können mit derselben Wahrscheinlichkeit aus jeder der Tüten stammen; in der ersten befinden sich hauptsächlich Kirschbonbons mit rotem Papier und Löchern; in der zweiten befinden sich hauptsächlich Zitronenbonbons mit grünem Papier und ohne Löcher. Die Zähler für die acht möglichen Bonbonarten sehen wie folgt aus:

	W = *rot*		**W = *grün***	
	H = 1	**H = 0**	**H = 1**	**H = 0**
F = *kirsch*	273	93	104	90
F = *zitrone*	79	100	94	167

Wir beginnen mit der Initialisierung der Parameter. Der numerischen Einfachheit halber wählen wir:[5]

$$\theta^{(0)} = 0{,}6\ ,\ \theta_{F1}^{(0)} = \theta_{W1}^{(0)} = \theta_{H1}^{(0)} = 0{,}6\ ,\ \theta_{F2}^{(0)} = \theta_{W2}^{(0)} = \theta_{H2}^{(0)} = 0{,}4. \tag{20.8}$$

Zuerst konzentrieren wir uns auf den Parameter θ. Im vollständig beobachtbaren Fall würden wir ihn direkt aus den *beobachteten* Zählern für die Bonbons aus den Tüten 1 und 2 schätzen. Weil die Tüte eine verborgene Variable ist, berechnen wir stattdessen die *erwarteten* Zähler. Der erwartete Zähler $\hat{N}(Tüte = 1)$ ist über alle Bonbons gerechnet die Summe der Wahrscheinlichkeiten, dass das Bonbon aus Tüte 1 stammt:

$$\theta^{(1)} = \hat{N}(Tüte = 1)/N = \sum_{j=1}^{N} P\left(Tüte = 1 \mid geschmack_j, papier_j, löcher_j\right)/N.$$

Diese Wahrscheinlichkeiten können durch einen beliebigen Inferenzalgorithmus für Bayessche Netze berechnet werden. Für ein naives Bayes-Modell, wie etwa dem in

5 In der Praxis ist es besser, die Parameter zufällig zu wählen, um lokale Maxima zu vermeiden, die aufgrund irgendwelcher Symmetrien entstehen.

unserem Beispiel gezeigten, können wir diese Inferenz „manuell“ durchführen. Dazu verwenden wir die Bayessche Regel und wenden die bedingte Unabhängigkeit an:

$$\theta^{(1)} = \frac{1}{N}\sum_{j=1}^{N} \frac{P\left(geschmack_j \mid Tüte = 1\right) P\left(papier_j \mid Tüte = 1\right) P\left(löcher_j \mid Tüte = 1\right) P(Tüte = 1)}{\sum_i P\left(geschmack_j \mid Tüte = i\right) P\left(papier_j \mid Tüte = i\right) P\left(löcher_j \mid Tüte = i\right) P(Tüte = i)}.$$

Durch Anwendung dieser Formel auf beispielsweise 273 rot eingewickelte Kirschbonbons mit Löchern erhalten wir einen Beitrag von

$$\frac{273}{1000} \cdot \frac{\theta_{F1}^{(0)}\theta_{W1}^{(0)}\theta_{H1}^{(0)}\theta^{(0)}}{\theta_{F1}^{(0)}\theta_{W1}^{(0)}\theta_{H1}^{(0)}\theta^{(0)} + \theta_{F2}^{(0)}\theta_{W2}^{(0)}\theta_{H2}^{(0)}\left(1-\theta^{(0)}\right)} \approx 0{,}22797.$$

Wenn wir für die anderen sieben Bonbonsorten in der Zählertabelle genauso vorgehen, erhalten wir $\theta^{(1)} = 0{,}6124$.

Jetzt betrachten wir die anderen Parameter, wie etwa θ_{F1}. Im vollständig beobachtbaren Fall würden wir ihn direkt aus den beobachteten Zählern der Kirsch- und Zitronenbonbons aus Tüte 1 schätzen. Der *erwartete* Zähler der Kirschbonbons aus Tüte 1 ist gegeben durch

$$\sum_{j:Geschmack_j = kirsche} P\left(Tüte = 1 \mid Geschmack_j = kirsche, papier_j, löcher_j\right).$$

Auch diese Wahrscheinlichkeiten können durch jeden beliebigen Algorithmus für Bayessche Netze berechnet werden. Wenn wir diesen Prozess vervollständigen, erhalten wir die neuen Werte für alle Parameter:

$$\theta^{(1)} = 0{,}6124\ ,\ \theta_{F1}^{(1)} = 0{,}6684\ ,\ \theta_{W1}^{(1)} = 0{,}6483\ ,\ \theta_{H1}^{(1)} = 0{,}6558\ ,$$
$$\theta_{F2}^{(1)} = 0{,}3887\ ,\ \theta_{W2}^{(1)} = 0{,}3817\ ,\ \theta_{H2}^{(1)} = 0{,}3827. \qquad (20.9)$$

Die Log-Wahrscheinlichkeit der Daten steigt nach der ersten Iteration von anfänglich -2044 auf etwa -2021, wie in ▶ Abbildung 20.12(b) gezeigt. Das bedeutet, die Aktualisierung verbessert die eigentliche Wahrscheinlichkeit um einen Faktor von etwa $e^{23} \approx 10^{10}$. Nach der zehnten Iteration weist das gelernte Modell eine bessere Übereinstimmung als das Originalmodell auf ($L = -1982{,}214$). Anschließend wird das Ganze sehr langsam. Das ist bei EM nicht ungewöhnlich und viele Systeme aus der Praxis kombinieren EM mit einem gradientenbasierten Algorithmus, wie etwa Newton-Raphson (siehe *Kapitel 4*) für die letzte Lernphase.

Tipp

Dieses Beispiel zeigt uns ganz allgemein, *dass die Parameteraktualisierungen für das Lernen Bayesscher Netze mit verborgenen Variablen direkt aus den Ergebnissen der Inferenz für jedes Beispiel zur Verfügung stehen. Darüber hinaus braucht man für jeden Parameter nur lokale A-posteriori-Wahrscheinlichkeiten.* Hier bedeutet „lokal“, dass die BWT für jede Variable X_i aus A-posteriori-Wahrscheinlichkeiten gelernt werden kann, wobei nur X_i und dessen übergeordnete Komponenten $\mathbf{U}_i$ beteiligt sind. Definiert man θ_{ijk} als die BWT-Parameter $P(X_i = x_{ij} | \mathbf{U}_i = \mathbf{u}_{ik})$, erhalten wir die Aktualisierung wie folgt aus den normalisierten erwarteten Zählern:

$$\theta_{ijk} \leftarrow \hat{N}\left(X_i = x_{ij}, \mathbf{U}_i = \mathbf{u}_{ik}\right) / \hat{N}\left(\mathbf{U}_i = \mathbf{u}_{ik}\right).$$

Die erwarteten Zähler erhält man durch Summierung über die Beispiele, wobei die Wahrscheinlichkeiten $P(X_i = x_{ij}, \mathbf{U}_i = \mathbf{u}_{ik})$ für jedes Beispiel unter Verwendung eines Inferenzalgorithmus für Bayessche Netze berechnet werden. Für die genauen Algorithmen – einschließlich der Variableneliminierung – stehen alle diese Wahrscheinlichkeiten direkt als Nebenprodukt einer Standardinferenz zur Verfügung, ohne dass für das Lernen spezifische, zusätzliche Berechnungen ausgeführt werden müssen. Darüber hinaus steht die für das Lernen benötigte Information *lokal* für jeden Parameter zur Verfügung.

20.3.3 Hidden-Markov-Modelle lernen

Unsere letzte Anwendung von EM beschreibt das Lernen der Übergangswahrscheinlichkeiten in Hidden-Markov-Modellen (HMMs). Aus *Abschnitt 15.3* wissen Sie, dass ein Hidden-Markov-Modell als dynamisches Bayessches Netz mit einer einzigen diskreten Zustandsvariablen dargestellt werden kann, wie in ▶ Abbildung 20.14 gezeigt. Jeder Datenpunkt besteht aus einer Beobachtungs*folge* endlicher Länge. Das Problem ist also, die Übergangswahrscheinlichkeiten aus einer Menge von Beobachtungsfolgen zu lernen (oder möglicherweise aus nur einer langen Folge).

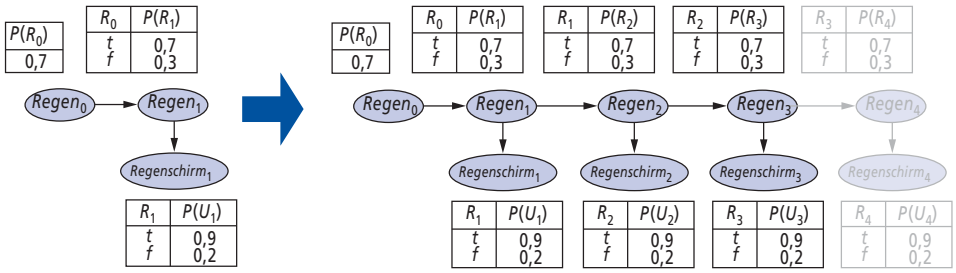


Abbildung 20.14: Ein aufgerolltes dynamisches Bayessches Netz, das ein Hidden-Markov-Modell darstellt (Wiederholung von Abbildung 15.16).

Wir haben bereits gezeigt, wie man Bayessche Netze lernt, aber es gibt eine Komplikation: In Bayesschen Netzen ist jeder Parameter unterschiedlich; in einem Hidden-Markov-Modell dagegen werden die einzelnen Übergangswahrscheinlichkeiten $\theta_{ijt} = P(X_{t+1} = j | X_t = i)$ vom Zustand i in den Zustand j zur Zeit t über die Zeit *wiederholt*, d.h. $\theta_{ijt} = \theta_{ij}$ für alle t. Um die Übergangswahrscheinlichkeit von dem Zustand i in den Zustand j abzuschätzen, berechnen wir einfach die erwartete Proportion, wie oft das System in den Zustand j übergeht, wenn es sich im Zustand i befindet:

$$\theta_{ij} \leftarrow \sum_t \hat{N}(X_{t+1} = j, X_t = i) / \sum_t \hat{N}(X_t = i).$$

Die erwarteten Zähler werden durch einen HMM-Inferenzalgorithmus berechnet. Der **Vorwärts/Rückwärts**-Algorithmus aus Abbildung 15.4 lässt sich ganz einfach modifizieren, um die erforderlichen Wahrscheinlichkeiten zu berechnen. Ein wichtiger Aspekt ist, dass die benötigten Wahrscheinlichkeiten diejenigen sind, die man durch **Glättung** statt durch **Filterung** erhält; das bedeutet, wir müssen auf nachfolgende Evidenz achten, wenn wir die Wahrscheinlichkeit abschätzen, dass ein bestimmter Übergang stattgefunden hat. Die Evidenz in einem Mordfall erhält man normalerweise *nach* dem Verbrechen (d.h. dem Übergang von Zustand i in Zustand jz).

20.3.4 Die allgemeine Form des EM-Algorithmus

Wir haben jetzt mehrere Beispiele für den EM-Algorithmus gezeigt. In jedem dieser Beispiele werden die erwarteten Werte für verborgene Variablen für jedes Beispiel berechnet und dann die Parameter neu berechnet, wobei die erwarteten Werte verwendet werden, als handelte es sich dabei um beobachtete Werte. Wir nehmen an, dass $\mathbf{x}$ alle beobachteten Werte in allen Beispielen und $\mathbf{Z}$ alle verborgenen Variablen für alle Beispiele darstellt sowie θ alle Parameter für das Wahrscheinlichkeitsmodell bezeichnet. Der EM-Algorithmus ist dann:

$$\theta^{(i+1)} = \operatorname*{argmax}_{\theta} \sum_{\mathbf{z}} P\left(\mathbf{Z} = \mathbf{z} \mid \mathbf{x}, \theta^{(i)}\right) L\left(\mathbf{x}, \mathbf{Z} = \mathbf{z} \mid \theta\right).$$

Diese Gleichung ist der EM-Algorithmus in Kurzfassung. Der E-Schritt ist die Berechnung der Summierung, d.h. die Erwartung der Log-Wahrscheinlichkeit der „vervollständigten" Daten im Hinblick auf die Verteilung $P(\mathbf{Z} = \mathbf{z} | \mathbf{x}, \theta^{(i)})$, nämlich die A-posteriori-Verteilung über die verborgenen Variablen bei bekannten Daten. Der M-Schritt ist die Maximierung dieser erwarteten Log-Wahrscheinlichkeit für die Parameter. Für gemischte Gaußsche Verteilungen sind die verborgenen Variablen die Z_{ij}, wobei Z_{ij} gleich 1 ist, wenn das Beispiel j aus der Komponente i erzeugt wurde. Für Bayessche Netze ist Z_{ij} der Wert der nicht beobachteten Variablen X_i im Beispiel j. Für HMMs ist Z_{it} der Zustand der Folge in Beispiel j zur Zeit t. Beginnend mit der allgemeinen Form kann man einen EM-Algorithmus für eine spezifische Anwendung ableiten, nachdem die geeigneten verborgenen Variablen identifiziert wurden.

Nachdem wir das allgemeine Konzept von EM verstanden haben, können wir alle möglichen Varianten und Verbesserungen ableiten. In vielen Fällen ist beispielsweise der E-Schritt – die Berechnung von A-posteriori-Verteilungen über die verborgenen Variablen – nicht handhabbar, ähnlich wie in großen Bayesschen Netzen. Es zeigt sich, dass man einen *annähernden* E-Schritt verwenden kann und dabei dennoch einen effektiven Lernalgorithmus erhält. Mit einem Sampling-Algorithmus wie etwa MCMC (siehe *Abschnitt 14.5*) ist der Lernprozess sehr intuitiv: Jeder Zustand (Konfiguration verborgener und beobachteter Variablen), der von MCMC besucht wird, wird genauso behandelt, als wäre es eine vollständige Beobachtung. Die Parameter können also nach jedem MCMC-Übergang direkt aktualisiert werden. Andere Formen der annähernden Inferenz, wie etwa variierende und Schleifenmethoden, haben sich für das Lernen großer Netze ebenfalls als sehr effektiv erwiesen.

20.3.5 Bayessche Netzstrukturen mit verborgenen Variablen lernen

In *Abschnitt 20.2.5* haben wir das Problem beschrieben, Bayessche Netzstrukturen mit vollständigen Daten zu lernen. Wenn nicht beobachtete Variablen die beobachteten Daten beeinflussen können, wird das Ganze schwieriger. Im einfachsten Fall könnte ein menschlicher Experte dem Lernalgorithmus mitteilen, dass bestimmte verborgene Variablen existieren, und es dem Algorithmus überlassen, einen Platz für sie in der Netzstruktur zu finden. Zum Beispiel könnte ein Algorithmus versuchen, die in Abbildung 20.10(a) gezeigte Struktur zu lernen, wobei er die Information erhält, dass *Herzkrankheit* (eine dreiwertige Variable) in das Modell aufgenommen werden soll. Wie im Fall mit vollständigen Daten hat der Gesamtalgorithmus eine äußere Schleife, die eine Suche über die Strukturen ausführt, und eine innere Schleife, die die Netzparameter entsprechend der gegebenen Struktur anpasst.

Wird dem Lernalgorithmus nicht mitgeteilt, welche verborgenen Variablen existieren, gibt es zwei Möglichkeiten: Entweder er geht davon aus, dass die Daten wirklich vollständig sind – was den Algorithmus eventuell zwingt, ein parameterintensives Modell wie das in Abbildung 20.10(b) zu lernen –, oder er *erfindet* neue verborgene Variablen, um das Modell zu vereinfachen. Der letztgenannte Ansatz kann implementiert werden, indem neue Änderungsauswahlen in der Struktursuche implementiert werden: Neben der Änderung der Verknüpfungen kann der Algorithmus eine verborgene Variable auch hinzufügen oder löschen oder ihre Stelligkeit ändern. Natürlich weiß der Algorithmus nicht, dass die neue Variable, die er erfunden hat, *Herzkrankheit* heißt; er hat auch keine aussagekräftigen Namen für die Werte. Glücklicherweise sind neu erfundene verborgene Variablen üblicherweise mit bereits existierenden Variablen verknüpft, sodass ein menschlicher Experte die lokalen bedingten Verteilungen für die neue Variable auswerten und ihre Bedeutung feststellen kann.

Wie im Fall mit den vollständigen Daten führt das reine Lernen der ML-Struktur zu einem vollständig verknüpften Netz (zudem einem ohne verborgene Variablen); deshalb braucht man irgendeine Möglichkeit, Komplexität zu bestrafen. Wir können auch MCMC anwenden, um Stichproben vieler möglicher Netzstrukturen zu erzeugen und dabei Bayessches Lernen anzunähern. Beispielsweise können wir Mischungen aus Gaußschen Verteilungen mit einer unbekannten Anzahl an Komponenten lernen, indem wir Stichproben über die Anzahl nehmen; die annähernde A-posteriori-Verteilung für die Anzahl der Gaußschen Verteilungen erhält man durch die Sampling-Häufigkeiten des MCMC-Prozesses.

Für den Fall der vollständigen Daten ist die innere Schleife (zum Lernen der Parameter) sehr schnell – es müssen einfach nur bedingte Häufigkeiten aus der Datenmenge extrahiert werden. Wenn es verborgene Variablen gibt, kann die innere Schleife viele Iterationen eines EM- oder gradientenbasierten Algorithmus enthalten und für jede Iteration fällt die Berechnung von A-posteriori-Wahrscheinlichkeiten in einem Bayesschen Netz an, was wiederum selbst ein NP-hartes Problem ist. Bisher hat sich dieser Ansatz als unpraktisch für das Lernen komplexer Modelle erwiesen. Eine mögliche Verbesserung ist der sogenannte **strukturelle EM-Algorithmus**, der ähnlich arbeitet wie der normale (parametrisierte) EM-Algorithmus, außer dass er sowohl die Struktur als auch die Parameter aktualisieren kann. So wie der normale EM die aktuellen Parameter verwendet, um die erwarteten Zähler im E-Schritt zu berechnen, und diese Zähler dann im M-Schritt für die Auswahl neuer Parameter benutzt, verwendet der strukturelle EM-Algorithmus die aktuelle Struktur, um die erwarteten Zähler zu berechnen, und wendet diese Zähler dann im M-Schritt an, um die Wahrscheinlichkeit für potenziell neue Strukturen auszuwerten. (Das steht im Gegensatz zur Methode „äußere Schleife/innere Schleife“, die neue erwartete Zähler für jede potenzielle Struktur berechnet.) Auf diese Weise kann strukturelles EM mehrere strukturelle Anpassungen am Netz vornehmen, ohne ein einziges Mal die erwarteten Zähler neu zu berechnen, und ist außerdem in der Lage, nicht triviale Bayessche Netzstrukturen zu lernen. Nichtsdestotrotz bleibt noch viel Arbeit zu tun, bevor wir wirklich sagen können, das Strukturlernproblem ist gelöst.

Bibliografische und historische Hinweise

Die Anwendung statistischer Lerntechniken in der KI war in den ersten Jahren ein sehr aktiver Forschungsbereich (siehe Duda und Hart, 1973), hat sich aber von der Mainstream-KI getrennt, als man sich hier hauptsächlich auf symbolische Methoden konzentrierte. Das Interesse wurde nach Einführung der Bayesschen Netzmodelle Ende der 1980er Jahre wiederbelebt. Etwa zur selben Zeit begann sich eine statistische Sicht des Lernens neuronaler Netze zu entwickeln. Ende der 1990er Jahre gab es eine bemerkenswerte Konvergenz der Interessen in den Bereichen Maschinenlernen, Statistik und neuronale Netze, die sich auf Methoden für die Erstellung großer probabilistischer Modelle aus den Daten konzentrierte.

Das naive Bayessche Modell ist eine der ältesten und einfachsten Formen Bayesscher Netze, die bis auf die 1950er Jahre zurückgeht. Seine Ursprünge wurden in *Kapitel 13* schon erwähnt. Der überraschende Erfolg wurde zum Teil von Domingos und Pazzani (1997) erklärt. Eine verstärkte (boosted) Form des naiven Bayesschen Lernens gewann den ersten KDD Cup Data Mining Competition (Elkan, 1997). Heckerman (1998) bietet eine ausgezeichnete Einführung in das allgemeine Problem des Lernens Bayesscher Netze. Bayessches Parameterlernen mit A-priori-Dirichlet-Verteilungen für Bayessche Netze wurde von Spiegelhalter et al. (1993) diskutiert. Das Softwarepaket BUGS (Gilks et al., 1994) bezieht viele dieser Konzepte ein und unterstützt ein sehr leistungsfähiges Werkzeug für die Formulierung und das Lernen komplexer Wahrscheinlichkeitsmodelle. Die ersten Algorithmen für das Lernen von Bayesschen Netzstrukturen verwendeten bedingte Unabhängigkeitstests (Pearl, 1988; Pearl und Verma, 1991). Spirtes et al. (1993) entwickelten einen umfassenden Ansatz, der im Paket TETRAD für das Lernen Bayesscher Netze realisiert ist. Algorithmische Verbesserungen führten später zu einem klaren Sieg im 2001 KKD Cup Data Mining Competition für eine Methode des Lernens Bayesscher Netze (Cheng et al., 2002). (Die konkrete Aufgabe war hier ein Problem aus der Bioinformatik mit 139.351 Merkmalen!) Ein Struktur lernender Ansatz, der auf dem Maximieren der Wahrscheinlichkeit basierte, wurde von Cooper und Herskovits (1992) entwickelt und von Heckerman et al. (1994) verbessert. Seit dieser Zeit haben mehrere algorithmische Fortschritte zu recht beachtlichen Leistungen im Fall mit vollständigen Daten geführt (Moore und Wong, 2003; Teyssier und Koller, 2005). Eine wichtige Komponente ist eine effiziente Datenstruktur, der AD-Baum, für die Zwischenspeicherung von Zählern über alle möglichen Kombinationen von Variablen und Werten (Moore und Lee, 1997). Friedman und Goldszmidt (1996) beschrieben den Einfluss der Repräsentation lokaler bedingter Verteilungen auf die gelernte Struktur.

Mit dem allgemeinen Problem des Lernens von Wahrscheinlichkeitsmodellen mit verborgenen Variablen und fehlenden Daten befasste sich Hartley (1958), der die allgemeine Idee beschrieb, was man später als EM bezeichnete, und mehrere Beispiele angab. Weitere Impulse kamen vom Baum-Welch-Algorithmus für HMM-Lernen (Baum und Petrie, 1966), der einen Spezialfall von EM darstellt. Die Veröffentlichung von Dempster, Laird und Rubin (1977), die den EM-Algorithmus in allgemeiner Form vorstellte und seine Konvergenz analysierte, gehört sowohl in der Informatik als auch in der Statistik zu den am häufigsten angeführten Quellen. (Dempster selbst betrachtet EM als Schema und nicht als Algorithmus, weil möglicherweise umfangreiche mathematische Aufbereitungen notwendig sind, bevor er sich auf eine neue Familie von Verteilungen anwenden lässt.) McLachlan und Krishnan (1997) widmen dem Algorithmus und seinen Eigenschaften ein ganzes Buch. Das spezifische Problem beim Lernen gemischter Modelle einschließlich gemischter Gaußscher Verteilungen wird von Titterington et al.

(1985) beschrieben. Innerhalb der KI war AUTOCLASS das erste erfolgreiche System, das EM für eine gemischte Modellierung verwendete (Cheeseman et al., 1988; Cheeseman und Stutz, 1996). AUTOCLASS wurde auf mehrere wissenschaftliche Klassifizierungsaufgaben der realen Welt angewendet, wie beispielsweise die Entdeckung neuer Sterntypen aus Spektraldaten (Goebel et al., 1989) sowie neuer Klassen von Proteinen und Introns aus DNA/Proteinfolgen-Datenbanken (Hunter und States, 1992).

Für das ML-Parameterlernen in Bayesschen Netzen mit verborgenen Variablen wurden EM und gradientenbasierte Methoden etwa um die gleiche Zeit von Lauritzen (1995), Russel et al. (1995) und Binder et al. (1997a) eingeführt. Der strukturelle EM-Algorithmus wurde von Friedman (1998) entwickelt und auf das ML-Lernen von Bayesschen Netzstrukturen mit latenten Variablen angewandt. Friedman und Koller (2003) beschreiben das Lernen von Bayesschen Strukturen.

Die Fähigkeit, die Struktur Bayesscher Netze zu lernen, ist eng mit dem Problem verwandt, *kausale* Informationen aus Daten zu erkennen. Ist es also möglich, Bayessche Netze so zu lernen, dass die erkannte Netzstruktur echte kausale Einflüsse aufzeigt? Viele Jahre lang wichen die Statistiker dieser Frage aus und dachten, beobachtete Daten (im Gegensatz zu experimentell erzeugten Daten) könnten nur korrelierende Informationen erbringen – schließlich konnten zwei Variablen, die eine Relation aufzeigten, auch von einem dritten, unbekannten kausalen Faktor beeinflusst werden, anstatt sich direkt gegenseitig zu beeinflussen. Pearl (2000) hat überzeugende Argumente für das Gegenteil gebracht und gezeigt, dass es viele Fälle gibt, wo die Kausalität zugesichert werden kann. Er entwickelte den Formalismus der **kausalen Netze**, um Ursachen und Effekte des Eingreifens ebenso wie normale bedingte Wahrscheinlichkeiten auszudrücken.

Parameterfreie Dichteabschätzungen, auch als **Parzen-Fenster**-Dichteabschätzungen bezeichnet, wurden anfänglich von Rosenblatt (1956) und Parzen (1962) untersucht. Viele Bücher wurden seit damals zur Untersuchung der Eigenschaften verschiedener Schätzer geschrieben. Eine Einführung finden Sie bei Devroye (1987). Außerdem nimmt der Umfang der Literatur zu parameterfreien Bayesschen Methoden ständig zu, wobei zunächst die wegweisende Arbeit von Ferguson (1973) zum **Dirichlet-Prozess** zu nennen ist, den man sich als Verteilung über Dirichlet-Verteilungen vorstellen kann. Diese Methoden sind besonders nützlich für Mischungen mit unbekannter Anzahl von Komponenten. Ghahramani (2005) und Jordan (2005) bieten nützliche Tutorials zu den vielen Anwendungen dieser Ideen auf statistisches Lernen. Der Text von Rasmussen und Williams (2006) behandelt den Gaußschen Prozess, der eine Möglichkeit bietet, A-priori-Verteilungen über den Raum kontinuierlicher Funktionen zu definieren.

Der Stoff in diesem Kapitel verknüpft Arbeiten aus den Bereichen Statistik und Mustererkennung, sodass die Geschichte oft und auf viele unterschiedliche Arten erzählt wird. Gute Bücher über Bayessche Statistik kommen unter anderem von DeGroot (1970), Berger (1985) und Gelman et al. (1995). Bishop (2007) und Hastie et al. (2009) bieten eine ausgezeichnete Einführung in statistische Lernmethoden. Für die Musterklassifizierung war der Klassiker jahrelang Duda und Hart (1973), jetzt aktualisiert (Duda et al., 2001). Die jährliche NIPS (*Neural Information Processing Conference*), deren Unterlagen in der Reihe *Advances in Neural Information Processing Systems* veröffentlicht werden, ist derzeit von Bayesschen Artikeln geprägt. Arbeiten zum Lernen von Bayesschen Netzen erscheinen auch in den Konferenzen *Uncertainty in AI* und *Machine Learning*, ebenso wie in mehreren Statistikkonferenzen. Den neuronalen Netzen gewidmete Zeitschriften sind unter anderem *Neural Computation, Neural Networks* und die *IEEE Transactions on Neural Networks*. Speziell für Bayessche Netze sind unter anderem die Tagung *Valencia International Meetings on Bayesian Statistics* und die Zeitschrift *Bayesian Analysis* zu nennen.

Zusammenfassung

Statistische Lernmethoden reichen von einfachen Mittelwertberechungen bis hin zur Konstruktion komplexer Modelle wie etwa Bayesscher Netze. Ihre Anwendungen sind in vielen Bereichen wie Informatik, Engineering, Neurobiologie, Neurowissenschaft, Psychologie und Physik zu finden. Dieses Kapitel hat einige der grundlegenden Konzepte vorgestellt und ein Gefühl für die mathematischen Grundlagen vermittelt. Die wichtigsten Aspekte sind:

- **Bayessche Lernmethoden** formulieren das Lernen als Form probabilistischer Inferenz, wobei die Beobachtungen verwendet werden, um eine A-priori-Verteilung über Hypothesen zu aktualisieren. Dieser Ansatz bietet eine gute Möglichkeit, Ockhams Rasiermesser zu implementieren, wird aber für komplexe Hypothesenräume schnell unhandlich.
- **MAP-Lernen** (Maximum a posteriori) wählt eine einzelne, wahrscheinlichste Hypothese für bekannte Daten aus. Die A-priori-Verteilung der Hypothesen wird immer noch verwendet und diese Methode ist häufig besser zu handhaben als vollständiges Bayessches Lernen.
- **ML-Lernen** (Maximum Likelihood) wählt einfach die Hypothese aus, die die Wahrscheinlichkeit der Daten maximiert; es ist äquivalent mit dem MAP-Lernen mit einer gleichmäßigen A-priori-Verteilung. In einfachen Fällen wie beispielsweise linearer Regression und vollständig beobachtbaren Bayesschen Netzen können ML-Lösungen in geschlossener Form ganz einfach gefunden werden. **Naives Bayessches** Lernen ist eine besonders effektive Technik, die auch gut auf größer werdende Probleme angewendet werden kann.
- Wenn einige Variablen verborgen sind, können lokale ML-Lösungen mithilfe des **EM-Algorithmus** gefunden werden. Anwendungen sind beispielsweise das Clustering unter Verwendung von gemischten Gaußschen Verteilungen, das Lernen von Bayesschen Netzen und das Lernen von Hidden-Markov-Modellen.
- Das Lernen der Struktur Bayesscher Netze ist ein Beispiel für die **Modellauswahl**. Diese beinhaltet in der Regel eine diskrete Suche im Strukturraum. Man benötigt eine Methode für die Abwägung der Modellkomplexität gegenüber dem Anpassungsgrad.
- **Parameterfreie Modelle** repräsentieren eine Verteilung unter Verwendung der Sammlung von Datenpunkten. Die Anzahl der Parameter wächst also mit der Trainingsmenge. **Nächste-Nachbarn**-Methoden suchen nach den Beispielen, die dem betreffenden Punkt am nächsten liegen, während **Kernel**-Methoden eine distanzgewichtete Kombination aller Instanzen bilden.

Das statistische Lernen ist immer noch ein sehr aktiver Forschungsbereich. Es wurden erhebliche Fortschritte in Theorie und Praxis gemacht und es ist jetzt möglich, fast jedes Modell zu lernen, für das eine genaue oder annähernde Inferenz praktikabel ist.

Übungen zu Kapitel 20

Lösungshinweise

1 Die für Abbildung 20.1 verwendeten Daten können als durch h_5 erzeugt betrachtet werden. Erzeugen Sie für jede der anderen vier Hypothesen eine Datenmenge der Länge 100 und zeichnen Sie die entsprechenden Graphen für $P(h_i | d_1,..., d_N)$ und $P(D_{N+1} = zitrone | d_1, ..., d_N)$. Kommentieren Sie Ihre Ergebnisse.

2 Wiederholen Sie Übung 20.1 und zeichnen Sie jetzt die Werte für $P(D_{N+1} = zitrone | h_{MAP})$ und $P(D_{N+1} = zitrone | h_{ML})$.

3 Angenommen, Anna hat den Nutzen c_A und ℓ_A für Kirsch- und Zitronenbonbons, während der Nutzen von Bob gleich c_B und ℓ_B ist. (Nachdem Anna jedoch ein Bonbon ausgewickelt hat, kauft Bob es nicht mehr.) Angenommen, Bob mag Zitronenbonbons lieber als Anna. Wäre es dann sinnvoll für Anna, ihm ihre Tüte Bonbons zu verkaufen, nachdem sie ihren Zitroneninhalt ausreichend sicher kennt? Wenn andererseits Anna zu viele Bonbons auswickelt, ist die Tüte irgendwann wertlos. Beschreiben Sie das Problem, den optimalen Punkt zu finden, wann die Tüte verkauft werden soll. Legen Sie den erwarteten Nutzen der optimalen Prozedur fest und verwenden Sie dafür die A-priori-Verteilung aus *Abschnitt 20.1*.

4 Zwei Statistiker gehen zum Arzt und erhalten beide dieselbe Prognose: eine 40-prozentige Wahrscheinlichkeit, dass das Problem die tödliche Krankheit A ist, und eine 60-prozentige Wahrscheinlichkeit, dass es die schwere Krankheit B ist. Glücklicherweise sind die Medikamente gegen A und B nicht teuer, zu 100% wirksam und frei von Nebeneffekten. Die Statistiker haben die Wahl, ein Medikament, beide oder keines einzunehmen. Was macht der erste Statistiker (ein überzeugter Bayesianer)? Was macht der zweite Statistiker, der immer die ML-Hypothese befolgt?

Der Arzt forscht und erkennt, dass es die Krankheit B in zwei Ausprägungen gibt, Dextro-B und Levo-B, die beide gleich wahrscheinlich sind und die beide mit dem Medikament gegen B behandelt werden. Jetzt gibt es drei Hypothesen. Was machen die beiden Statistiker?

5 Erklären Sie, wie die Boosting-Methode aus *Kapitel 18* auf das Bayessche Lernen angewendet wird. Testen Sie die Leistung des resultierenden Algorithmus für das Restaurant-Lernproblem.

6 Betrachten Sie N Datenpunkte (x_j, y_j), wobei y_j gemäß dem linearen Gaußschen Modell aus Gleichung (20.5) aus den x_j erzeugt werden. Finden Sie Werte von θ_1, θ_2 und σ, die die bedingte Log-Wahrscheinlichkeit der Daten maximieren.

7 Betrachten Sie das Noisy-OR-Modell für Fieber, das in *Abschnitt 14.3* beschrieben wurde. Beschreiben Sie, wie ein ML-Lernen angewendet werden kann, um eine Übereinstimmung mit den Parametern eines solchen Modells mit einer Menge vollständiger Daten zu erzielen. (*Hinweis*: Verwenden Sie die Kettenregel für partielle Ableitungen.)

8 Diese Übung untersucht die Eigenschaften der in Gleichung (20.6) definierten Betaverteilung.

a. Zeigen Sie durch Integration über den Bereich [0, 1], dass die Normalisierungskonstante für die Verteilung $beta[a,b]$ durch $\alpha = \Gamma(a+b)/\Gamma(a)\Gamma(b)$ gegeben ist, wobei $\Gamma(x)$ die **Gamma-Funktion** ist, definiert durch $\Gamma(x+1) = x \cdot \Gamma(x)$ und $\Gamma(1) = 1$. (Für ganzzahlige x, $\Gamma(x+1) = x!$.)

b. Zeigen Sie, dass der Mittelwert gleich $a/(a+b)$ ist.

c. Finden Sie die Modi (die wahrscheinlichsten Werte von θ).

d. Beschreiben Sie die Verteilung $beta[\epsilon,\epsilon]$ für sehr kleine ϵ. Was passiert, wenn eine solche Verteilung aktualisiert wird?

9 Betrachten Sie ein beliebiges Bayessches Netz, eine vollständige Datenmenge für dieses Netz und die Wahrscheinlichkeit für die dem Netz entsprechende Datenmenge. Geben Sie einen einfachen Beweis dafür, dass die Wahrscheinlichkeit der Daten nicht sinken kann, wenn wir dem Netz eine neue Verknüpfung hinzufügen, und berechnen Sie die ML-Parameterwerte neu.

10 Betrachten Sie die Anwendung von EM auf das Lernen der Parameter aus dem in Abbildung 20.13(a) gezeigten Netz für die wahren Parameter in Gleichung (20.7).

a. Erklären Sie, warum der EM-Algorithmus nicht funktioniert, wenn es im Modell nur zwei und nicht drei Attribute gibt.

b. Zeigen Sie die Berechnungen für die erste Iteration von EM beginnend mit Gleichung (20.8).

c. Was passiert, wenn wir zu Beginn alle Parameter auf denselben Wert p setzen? (*Hinweis*: Vielleicht hilft es Ihnen, dies empirisch zu betrachten, bevor Sie das allgemeine Ergebnis ableiten.)

d. Schreiben Sie einen Ausdruck für die Log-Wahrscheinlichkeit der tabellierten Bonbondaten in *Abschnitt 20.3.2* für die Parameter, berechnen Sie die partiellen Ableitungen für jeden Parameter und untersuchen Sie die Natur des in Teil (c) erreichten Fixpunktes.

Verstärkendes (Reinforcement-)Lernen

21

ÜBERBLICK

In diesem Kapitel betrachten wir, wie ein Agent aus Erfolg und Misserfolg, durch Belohnung und Bestrafung lernen kann.

21.1 Einführung

In den *Kapiteln 18*, *19* und *20* ging es um Lernmethoden, die Funktionen, logische Theorien und Wahrscheinlichkeitsmodelle aus Beispielen lernen. In diesem Kapitel betrachten wir, wie Agenten lernen, *was zu tun ist*, wenn es keine bezeichneten Beispiele dafür gibt, was zu tun ist.

Tipp

Sehen Sie sich zum Beispiel das Problem an, Schachspielen zu lernen. Einem Agenten, der durch überwachtes Lernen arbeitet, muss der korrekte Zug für jede Spielsituation mitgeteilt werden, doch ist ein derartiges Feedback selten verfügbar. Wenn das Feedback von einem Lehrer fehlt, kann ein Agent ein Übergangsmodell für seine eigenen Züge lernen und vielleicht auch die Züge des Gegners vorherzusagen, doch *ohne ein Feedback, was gut oder was schlecht war, hat der Agent keine Anhaltspunkte für die Entscheidung, welchen Zug er ausführen soll.* Der Agent muss wissen, dass etwas Gutes passiert ist, wenn er (zufällig) den Gegner Schachmatt setzt, und dass etwas Schlechtes passiert ist, wenn er selbst Schachmatt gesetzt wird – oder umgekehrt, wenn es sich um Räuberschach handelt. Diese Art des Feedbacks wird auch als **Belohnung** oder **Verstärkung** bezeichnet. In Spielen wie Schach erhält der Agent die Verstärkungen nur am Spielende. In anderen Umgebungen treten die Belohnungen häufiger auf. Im Tischtennis kann jeder erzielte Punkt als Belohnung betrachtet werden; wenn man Kraulen lernt, ist jede Vorwärtsbewegung eine Errungenschaft. Unser Framework für Agenten betrachtet die Belohnung als *Teil* der Eingabewahrnehmung, aber der Agent muss „fest verdrahtet" sein, um diesen Teil als Belohnung und nicht nur als einfache Sensoreingabe zu erkennen. Bei Tieren kann man annehmen, dass sie feste Mechanismen besitzen, um Schmerzen und Hunger als negative Belohnungen und Freude und Nahrungsaufnahme als positive Belohnungen zu erkennen. Die Verstärkung wird von Tierpsychologen seit mehr als 60 Jahren sorgfältig untersucht.

Belohnungen wurden in *Kapitel 17* eingeführt, wo sie dazu dienten, optimale Strategien in **Markov-Entscheidungsprozessen** (MEPs) zu definieren. Eine optimale Strategie maximiert den erwarteten Gesamtgewinn. Die Aufgabe des verstärkenden Lernens ist es, anhand beobachteter Belohnungen eine optimale (oder fast optimale) Strategie für die Umgebung zu lernen. Während der Agent in *Kapitel 17* ein vollständiges Modell der Umgebung besitzt und die Belohnungsfunktion kennt, gehen wir hier von keinerlei Vorwissen aus. Stellen Sie sich vor, Sie spielen ein neues Spiel, dessen Regeln Sie nicht kennen. Nach 100 Zügen sagt Ihr Gegner: „Du hast verloren." Das ist verstärkendes Lernen im Kleinformat.

In vielen komplexen Domänen ist das verstärkende Lernen die einzige Möglichkeit, ein Programm zu trainieren, sodass es auf höchstem Niveau gute Leistungen erbringt. Beim Spielen beispielsweise ist es für einen Menschen sehr schwierig, eine genaue und konsistente Bewertung sehr vieler Positionen vorzunehmen, die man braucht, um eine Bewertungsfunktion direkt aus Beispielen zu trainieren. Stattdessen kann man dem Programm mitteilen, wann es gewonnen oder verloren hat, und es kann diese Informationen nutzen, um eine Bewertungsfunktion zu lernen, die für jede gegebene Position sinn-

voll genaue Schätzungen der Wahrscheinlichkeit zu gewinnen abgibt. Analog dazu ist es extrem schwierig, einen Agenten zu programmieren, der einen Helikopter fliegen soll; durch geeignete negative Belohnungen für Abstürze, Turbulenzen oder Kursabweichungen kann ein Agent dagegen selbst lernen zu fliegen.

Verstärkendes Lernen beinhaltet gewissermaßen die gesamte KI: Ein Agent wird in einer Umgebung platziert und muss lernen, sich dort erfolgreich zu verhalten. Um das Kapitel überschaubar zu halten, konzentrieren wir uns auf einfache Einstellungen und einfache Agentenentwürfe. Größtenteils gehen wir von einer vollständig beobachtbaren Umgebung aus, sodass jede Wahrnehmung den aktuellen Zustand bereitstellt. Andererseits gehen wir davon aus, dass der Agent nicht weiß, wie die Umgebung funktioniert oder welche Aktionen er vorzunehmen hat, und erlauben probabilistische Aktionsergebnisse. Somit sieht sich der Agent einem unbekannten Markov-Entscheidungsprozess gegenüber. Wir betrachten drei der in *Kapitel 2* eingeführten Agentenentwürfe:

- Ein **nutzenbasierter Agent** lernt eine Nutzenfunktion für Zustände und verwendet sie, um Aktionen auszuwählen, die den erwarteten Ergebnisnutzen maximieren.
- Ein **Q-lernender Agent** lernt eine Aktion/Nutzen-Funktion oder **Q-Funktion**, indem ihm der erwartete Nutzen mitgeteilt wird, den er durch die Ausführung einer bestimmten Aktion in einem bestimmten Zustand erzielt.
- Ein **Reflex-Agent** lernt eine Strategie, die eine direkte Abbildung von Zuständen auf Aktionen vornimmt.

Ein nutzenbasierter Agent braucht ein Modell der Umgebung, um Entscheidungen treffen zu können, weil er die Zustände kennen muss, zu denen seine Aktionen führen. Um beispielsweise eine Backgammon-Auswertungsfunktion nutzen zu können, muss ein Backgammon-Programm wissen, welche erlaubten Züge ihm zur Verfügung stehen *und wie sie sich auf die Konstellation auf dem Brett auswirken*. Nur auf diese Weise kann er die Nutzenfunktion auf die Ergebniszustände anwenden. Ein Q-lernender Agent dagegen kann die erwarteten Nutzen für die ihm zur Verfügung stehenden Auswahlen vergleichen, ohne ihre Ergebnisse kennen zu müssen, er braucht also kein Modell der Umgebung. Andererseits wissen Q-lernende Agenten nicht, was ihre Aktionen bewirken, deshalb können sie nicht vorausschauen. Das kann ihre Lernfähigkeit erheblich einschränken, wie wir noch sehen werden.

Wir beginnen in *Abschnitt 21.2* mit dem **passiven Lernen**, wo die Strategie des Agenten feststeht und er die Aufgabe hat, die Nutzen von Zuständen (oder Zustand/Aktion-Paaren) zu lernen; dafür könnte es auch erforderlich sein, ein Modell der Umgebung zu lernen. In *Abschnitt 21.3* geht es um **aktives Lernen**, wo der Agent ebenfalls lernen muss, was zu tun ist. Der wichtigste Aspekt ist die **Exploration**: Ein Agent muss so viel wie möglich über seine Umgebung in Erfahrung bringen, um zu lernen, wie er sich darin verhalten soll. *Abschnitt 21.4* beschreibt, wie ein Agent induktives Lernen nutzen kann, um sehr viel schneller aus seinen Erfahrungen zu lernen. *Abschnitt 21.5* beschreibt Methoden für das direkte Lernen von Strategierepräsentationen in Reflex-Agenten. Um die Beschreibungen in diesen Kapiteln nachvollziehen zu können, müssen Sie die Markov-Entscheidungsprozesse (*Kapitel 17*) verstanden haben.

21.2 Passives verstärkendes Lernen

Um das Ganze einfach zu gestalten, beginnen wir mit dem Fall eines passiv lernenden Agenten unter Verwendung einer zustandsbasierten Repräsentation in einer vollständig beobachtbaren Umgebung. Beim passiven Lernen steht die Strategie π des Agenten fest: Im Zustand s führt er immer die Aktion $\pi(s)$ aus. Sein Ziel ist einfach, zu lernen, wie gut die Strategie ist – d.h. die Nutzenfunktion $U^{\pi}(s)$ zu lernen. Wir verwenden als Beispiel die in *Kapitel 17* eingeführte 4×3-Welt. ▶ Abbildung 21.1 zeigt eine Strategie für diese Welt und die entsprechenden Nutzen. Offensichtlich ist die Aufgabe des passiven Lernens vergleichbar mit der Aufgabe der **Strategiebewertung**, Teil des in *Abschnitt 17.3* beschriebenen Algorithmus der **Strategie-Iteration**. Der wichtigste Unterschied ist, dass der passiv lernende Agent das **Übergangsmodell** $P(s' \mid s, a)$ nicht kennt, das die Wahrscheinlichkeit angibt, nach Ausführen der Aktion a aus dem Zustand s in den Zustand s' zu gelangen. Er kennt auch nicht die **Gewinnfunktion** $R(s)$, die die Belohnung für jeden Zustand angibt.

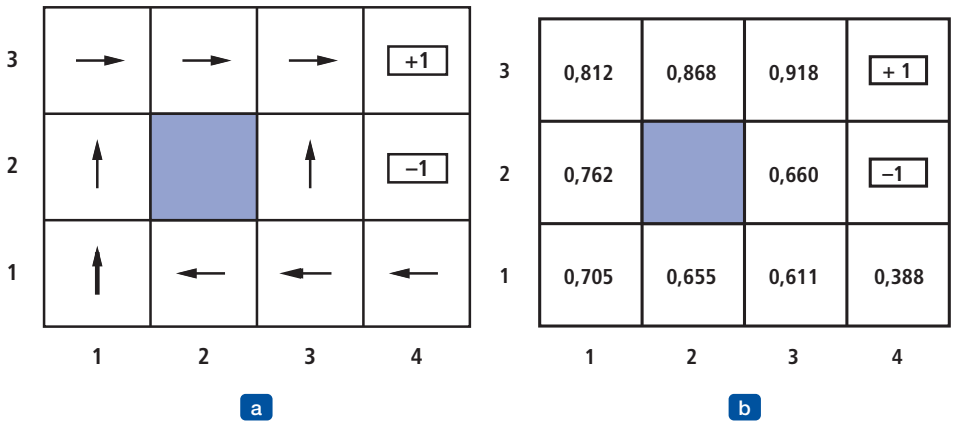


Abbildung 21.1: (a) Eine Strategie π für die 4×3-Welt; diese Strategie ist optimal für die Gewinne von $R(s) = -0{,}04$ in Nichtterminalzuständen und ohne Verminderung. (b) Die Nutzen der Zustände in der 4×3-Welt für die Strategie π.

Der Agent führt unter Verwendung seiner Strategie π mehrere **Versuche** in der Umgebung aus. Bei jedem Versuch beginnt er im Zustand (1,1) und nimmt eine Folge von Zustandsübergängen wahr, bis er einen der Terminalzustände erreicht, (4,2) oder (4,3). Seine Wahrnehmung übermittelt ihm sowohl den aktuellen Zustand als auch den in diesem Zustand erhaltenen Gewinn. Typische Versuche könnten wie folgt aussehen:

$$(1,1)_{-0,04} \rightarrow (1,2)_{-0,04} \rightarrow (1,3)_{-0,04} \rightarrow (1,2)_{-0,04} \rightarrow (1,3)_{-0,04} \rightarrow (2,3)_{-0,04} \rightarrow (3,3)_{-0,04} \rightarrow (4,3)_{+1}$$
$$(1,1)_{-0,04} \rightarrow (1,2)_{-0,04} \rightarrow (1,3)_{-0,04} \rightarrow (2,3)_{-0,04} \rightarrow (3,3)_{-0,04} \rightarrow (3,2)_{-0,04} \rightarrow (3,3)_{-0,04} \rightarrow (4,3)_{+1}$$
$$(1,1)_{-0,04} \rightarrow (2,1)_{-0,04} \rightarrow (3,1)_{-0,04} \rightarrow (3,2)_{-0,04} \rightarrow (4,2)_{-1}.$$

Beachten Sie, dass jede Zustandswahrnehmung mit dem erhaltenen Gewinn beschriftet ist. Das soll es ermöglichen, die Informationen über die Gewinne zu nutzen, um den erwarteten Nutzen $U^{\pi}(s)$ zu lernen, der jedem Nichtterminalzustand s zugeordnet ist. Der Nutzen ist als die erwartete Summe der (verminderten) Gewinne definiert, die

der Agent erhält, wenn er der Strategie π folgt. Wie in *Gleichung (17.2)* in *Abschnitt 17.1.2* schreiben wir:

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(S_t)\right]. \tag{21.1}$$

Hier ist $R(s)$ die Belohnung für einen Zustand, S_t (eine Zufallsvariable) der zur Zeit t erreichte Zustand, wenn die Strategie π ausgeführt wird, und $S_0 = s$. Wir nehmen einen **Diskontierungsfaktor** γ in alle unsere Gleichungen auf, aber für die 4 × 3-Welt setzen wir $\gamma = 1$.

21.2.1 Direkte Nutzenschätzung

Eine einfache Methode für die **direkte Nutzenschätzung** wurde Ende der 1950er Jahre im Bereich der **adaptiven Kontrolltheorie** von Widrow und Hoff (1960) erfunden. Die Idee dabei ist, dass der Nutzen eines Zustandes der **erwartete Gesamtgewinn** von diesem Zustand an ist, und jeder Versuch stellt eine *Stichprobe* dieses Wertes für jeden besuchten Zustand bereit. Der erste der zuvor gezeigten drei Versuche beispielsweise erzeugt einen Stichprobengesamtgewinn von 0,72 für Zustand (1,1), zwei Stichproben von 0,76 und 0,84 für (1,2), zwei Stichproben von 0,80 und 0,88 für (1,3) usw. Am Ende jeder Folge berechnet der Algorithmus also den beobachteten Gewinn für jeden Zustand und aktualisiert den geschätzten Nutzen für den Zustand entsprechend, indem er einfach einen gleitenden Mittelwert für jeden Zustand in einer Tabelle verwaltet. Bei unendlich vielen Beispielen konvergiert das Stichprobenmittel irgendwann zur wahren Erwartung in Gleichung (21.1).

Es ist offensichtlich, dass die direkte Nutzenschätzung einfach eine Instanz überwachten Lernens ist, wobei jedes Beispiel den Zustand als Eingabe und den beobachteten Gewinn als Ausgabe hat. Das bedeutet, wir haben das verstärkende Lernen zu einem Standardproblem des induktiven Lernens gemacht, wie in *Kapitel 18* diskutiert. *Abschnitt 21.4* beschreibt die Verwendung leistungsfähigerer Repräsentationen für die Nutzenfunktion. Lerntechniken für diese Repräsentationen können direkt auf die beobachteten Daten angewendet werden.

Tipp

Die direkte Nutzenschätzung schafft es, das Problem des verstärkenden Lernens auf ein induktives Lernproblem zu reduzieren, worüber man sehr viel weiß. Leider fehlt eine sehr wichtige Information, nämlich die Tatsache, dass die Nutzen von Zuständen nicht unabhängig sind! *Der Nutzen jedes Zustandes ist gleich seinem eigenen Gewinn plus dem erwarteten Nutzen seiner Nachfolgerzustände*. Das bedeutet, die Nutzenwerte gehorchen den Bellman-Gleichungen für eine feststehende Strategie (siehe auch *Gleichung (17.10)*):

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi(s)) U^{\pi}(s'). \tag{21.2}$$

Wenn die Verknüpfungen zwischen den Zuständen ignoriert werden, bietet die direkte Nutzenschätzung keine Gelegenheiten zum Lernen. Der zweite der zuvor gezeigten Versuche beispielsweise erreicht den Zustand (3,2), der zuvor nicht besucht worden war. Der nächste Übergang erreicht (3,3), für den aus dem ersten Versuch bekannt ist, dass er einen hohen Nutzen hat. Die Bellman-Gleichung weist unmittelbar darauf hin, dass

(3,2) sehr wahrscheinlich auch einen hohen Nutzen hat, weil es zu (3,3) führt, aber die direkte Nutzenschätzung lernt bis zum Ende des Versuches nichts. Allgemeiner ausgedrückt, wir können die direkte Nutzenschätzung als Suche in einem Hypothesenraum für U betrachten, der sehr viel größer ist, als er sein müsste, weil er viele Funktionen enthält, die die Bellman-Gleichungen verletzen. Aus diesem Grund konvergiert der Algorithmus häufig sehr langsam.

21.2.2 Adaptive dynamische Programmierung

Ein **ADP-Agent** (**Adaptive dynamische Programmierung**) nutzt die Bedingungen zwischen den Nutzen von Zuständen, indem er das Übergangsmodell lernt, das sie verbindet, und den entsprechenden Markov-Entscheidungsprozess mithilfe einer dynamischen Programmiermethode löst. Für einen passiv lernenden Agenten bedeutet dies, dass er das gelernte Übergangsmodell $P(s' \mid s, \pi(s))$ sowie die beobachteten Gewinne $R(s)$ in die Bellman-Gleichungen (21.2) einsetzt, um die Nutzen der Zustände zu berechnen. Wie wir in unserer Diskussion der Strategie-Iteration in *Kapitel 17* angemerkt haben, sind diese Gleichungen linear (es gibt keine Maximierung), sodass sie unter Verwendung eines beliebigen Paketes für die lineare Algebra gelöst werden können. Alternativ können wir den Ansatz der **modifizierten Strategie-Iteration** (siehe *Abschnitt 17.3*) anwenden, wobei wir eine vereinfachte Wert-Iteration verwenden, um die Nutzenschätzungen nach jeder Änderung am gelernten Modell zu aktualisieren. Weil sich dieses Modell bei jeder Beobachtung normalerweise nur geringfügig ändert, kann der Wert-Iterationsprozess die vorherigen Nutzenschätzungen als Ausgangswerte verwenden und sollte relativ schnell konvergieren.

Der Prozess, das eigentliche Modell zu lernen, ist einfach, weil die Umgebung vollständig beobachtbar ist. Das bedeutet, wir haben eine Aufgabe überwachten Lernens, wobei die Eingabe ein Zustand/Aktion-Paar und die Ausgabe der resultierende Zustand ist. Im einfachsten Fall können wir das Übergangsmodell als Tabelle von Wahrscheinlichkeiten repräsentieren. Wir beobachten, wie oft jedes Aktionsergebnis auftritt, und schätzen die Übergangswahrscheinlichkeit $P(s' \mid s, a)$ aus der Häufigkeit, wie oft s' erreicht wird, wenn wir a in s ausführen. Beispielsweise wird in den drei Versuchen in *Abschnitt 21.2 Right* in (1,3) dreimal ausgeführt und in zwei von drei Fällen ist der resultierende Zustand gleich (2,3); deshalb wird $P(2,3) \mid (1,3), \mathit{Right})$ als 2/3 geschätzt.

▶ Abbildung 21.2 zeigt das vollständige Agenten-Programm für einen passiven ADP-Agenten. Seine Leistung für die 4×3-Welt ist in ▶ Abbildung 21.3 gezeigt. In Bezug darauf, wie schnell er seine Wertschätzungen verbessern kann, ist der ADP-Agent nur durch seine Fähigkeit beschränkt, das Übergangsmodell zu lernen. In diesem Sinne stellt er einen Standard dar, mit dem die anderen Algorithmen für verstärkendes Lernen verglichen werden. Für große Zustandsräume ist er jedoch schwer zu handhaben. Im Backgammon beispielsweise müssten dabei etwa 10^{50} Gleichungen mit 10^{50} Unbekannten gelöst werden.

```
function PASSIVE-ADP-AGENT(percept) returns eine Aktion
   inputs: percept, eine Wahrnehmung, die den aktuellen Zustand s'
                    und ein Belohnungssignal r' anzeigt
   persistent: π, eine feste Strategie
           mdp, ein MEP mit Modell P, Belohnungen R, Minderung γ
           U, eine Tabelle mit Nutzeneinträgen, anfänglich leer
```

```
        N_sa, eine Tabelle mit Häufigkeitseinträgen für Zustand/Aktion-
              Paare, anfänglich null
        N_s'|sa, eine Tabelle der Ergebnishäufigkeiten für gegebene
              Zustand/Aktion-Paare, anfänglich null
        s, a, der vorherige Zustand und die Aktion, anfänglich leer

  if s' ist neu then U[s'] ← r'; R[s'] ← r'
  if s ist nicht null then
     inkrementiere N_sa[s, a] und N_s'|sa[s', s, a]
     for each t sodass N_s'|sa[t, s, a] ist ungleich null do
        P[t|s, a] ← N_s'|sa[t, s, a]/N_sa[s, a]
  U ← POLICY-EVALUATION(π, U, mdp)
  if s'.TERMINAL then s, a ← null else s, a ← s', π[s']
  return a
```

Abbildung 21.2: Ein passiv verstärkend lernender Agent, der auf adaptiver dynamischer Programmierung basiert. Die Funktion POLICY-EVALUATION löst die Bellman-Gleichungen für eine feststehende Strategie.

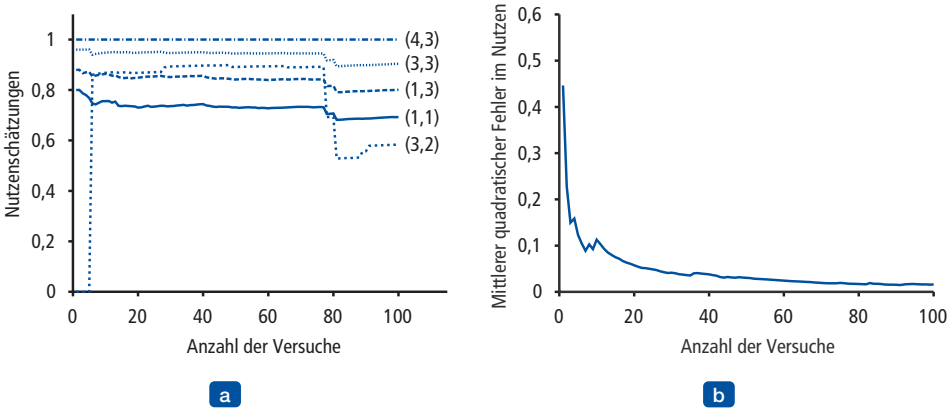


Abbildung 21.3: Die Lernkurven für das passive ADP-Lernen für die 4 × 3-Welt für die in Abbildung 21.1(a) gezeigte optimale Strategie. (a) Die Nutzenschätzung für eine ausgewählte Untermenge von Zuständen als Funktion der Anzahl der Versuche. Beachten Sie, dass beim 78. Versuch große Änderungen stattfinden – hier fällt der Agent zum ersten Mal in den Terminalzustand –1 bei (4,2). Der mittlere quadratische Fehler (siehe Anhang A) in der Schätzung für $U(1,1)$, gemittelt über 20 Durchläufe von je 100 Versuchen.

Einem Leser, der mit den Bayesschen Lernkonzepten von *Kapitel 20* vertraut ist, wird aufgefallen sein, dass der Algorithmus in Abbildung 21.2 die Maximum-Likelihood-Schätzung verwendet, um das Übergangsmodell zu lernen. Und indem er eine Strategie ausschließlich nach dem *geschätzten Modell* auswählt, handelt er so, *als wäre* das Modell korrekt. Dies ist nicht unbedingt zu empfehlen! Zum Beispiel könnte ein Taxi-Agent, der die Bedeutung der Verkehrsampeln nicht kennt, ein Rotlicht ein- oder zweimal ohne nachteilige Wirkung missachten und dann eine Strategie formulieren, die von nun an rote Ampeln ignoriert. Besser wäre es, eine Strategie zu wählen, die zwar für das durch Maximum-Likelihood geschätzte Modell nicht optimal ist, aber vernünftig im gesamten Bereich der Modelle funktioniert, die eine berechtigte Chance haben, das wahre Modell zu sein. Es gibt zwei mathematische Ansätze, die über diese Variante verfügen.

Der erste Ansatz, das **Bayessche verstärkende Lernen**, nimmt eine A-priori-Wahrscheinlichkeit $P(h)$ für jede Hypothese h an, welches das wahre Modell ist. Die A-posteriori-Wahrscheinlichkeit $P(h \mid \mathbf{e})$ wird in der üblichen Weise durch die Bayes-Regel für die bis zu diesem Zeitpunkt gegebenen Beobachtungsdaten ermittelt. Wenn dann der Agent entschieden hat, den Lernvorgang anzuhalten, ist die optimale Strategie diejenige, die den höchsten erwarteten Nutzen ergibt. Ist u_h^π der erwartete Nutzen, der über alle möglichen Startzustände gemittelt wird und durch Ausführen der Strategie π in Modell h erhalten wurde, dann haben wir:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \sum_h P(h \mid \mathbf{e}) u_h^\pi .$$

In bestimmten Spezialfällen lässt sich diese Strategie sogar berechnen! Wenn der Agent jedoch das Lernen später fortsetzt, ist es beträchtlich schwieriger, eine optimale Strategie zu finden, da der Agent die Wirkungen von zukünftigen Beobachtungen auf seinen Glauben über das Übergangsmodell betrachten muss. Das Problem wird zu einem partiell beobachtbaren MEP, dessen Belief States Verteilungen über Modellen sind. Dieses Konzept bietet eine analytische Grundlage für das Verstehen des in *Abschnitt 21.3* beschriebenen Explorationsproblems.

Der zweite Ansatz, der von **robuster Kontrolltheorie** abgeleitet ist, ermöglicht eine *Menge* möglicher Modelle $\mathcal{H}$ und definiert eine optimale robuste Strategie als eine, die das beste Ergebnis im ungünstigsten Fall über $\mathcal{H}$ liefert:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \min u_h^\pi .$$

Oftmals ist die Menge $\mathcal{H}$ die Menge der Modelle, die eine bestimmte Wahrscheinlichkeitsschwelle auf $P(h \mid \mathbf{e})$ überschreiten, sodass der robuste Ansatz und der Bayessche Ansatz verwandt sind. Manchmal lässt sich die robuste Lösung effizient berechnen. Darüber hinaus gibt es Algorithmen für verstärkendes Lernen, die vorzugsweise robuste Lösungen produzieren. Allerdings gehen wir hier auf diese Algorithmen nicht ein.

21.2.3 Lernen mit temporaler Differenz

Das Lösen des zugrunde liegenden MEP wie im vorhergehenden Abschnitt ist nicht der einzige Weg, die Bellman-Gleichungen für das Lernproblem gefügig zu machen. Eine andere Möglichkeit ist es, die Nutzen der beobachteten Zustände anhand der beobachteten Übergänge anzupassen, sodass sie mit den Bedingungsgleichungen übereinstimmen. Betrachten Sie beispielsweise den Übergang von (1,3) nach (2,3) im zweiten Versuch in *Abschnitt 21.2*. Angenommenen, die Nutzenschätzungen sind nach dem ersten Versuch $U^\pi(1,3) = 0{,}84$ und $U^\pi(2,3) = 0{,}92$. Wenn dieser Übergang immer stattfindet, erwarten wir, dass die Nutzen der folgenden Gleichung gehorchen:

$$U^\pi(1,3) = -0.04 + U^\pi(2,3).$$

$U^\pi(1,3)$ wäre also 0,88. Folglich ist seine aktuelle Schätzung von 0,84 etwas niedrig und sollte erhöht werden. Allgemeiner können wir sagen, wenn ein Übergang vom Zustand s in den Zustand s' stattfindet, wenden wir die folgende Aktualisierung auf $U^\pi(s)$ an:

$$U^\pi(s) \leftarrow U^\pi(s) + \alpha(R(s) + \gamma\, U^\pi(s') - U^\pi(s)). \tag{21.3}$$

Hier ist α der **Lernratenparameter**. Weil diese Aktualisierungsregel die Differenzen der Nutzen aufeinanderfolgender Zustände verwendet, wird sie auch häufig als **TD-Gleichung** (**Temporale Differenz**) bezeichnet.

Tipp

Alle TD-Methoden arbeiten nach dem Prinzip, die Nutzenschätzungen gegen das ideale Gleichgewicht anzupassen, was lokal gilt, wenn die Nutzenschätzungen korrekt sind. Im Fall des passiven Lernens ist das Gleichgewicht durch Gleichung (21.2) gegeben. Jetzt veranlasst Gleichung (21.3), dass der Agent das durch Gleichung (21.2) vorgegebene Gleichgewicht erreicht, aber es gibt eine Feinheit dabei zu berücksichtigen. Beachten Sie zunächst, dass die Aktualisierungen nur den beobachteten Nachfolger s' betreffen, während die tatsächlichen Gleichgewichtsbedingungen alle möglichen nachfolgenden Zustände beinhalten. Man könnte denken, dies verursache eine unpassend große Änderung in $U^\pi(s)$, wenn ein sehr seltener Übergang stattfindet; tatsächlich jedoch konvergiert der *Durchschnittswert* von $U^\pi(s)$ auf den korrekten Wert hin, weil seltene Übergänge wirklich nur selten auftreten. Wenn wir darüber hinaus α von einem festen Parameter in eine Funktion ändern, die abfällt, je häufiger der Zustand besucht wurde, dann konvergiert $U(s)$ selbst auf den korrekten Wert.[1] Damit erhalten wir das in ▶ Abbildung 21.4 gezeigte Agentenprogramm. ▶ Abbildung 21.5 zeigt die Leistung des passiven TD-Agenten für die 4×3-Welt. Er lernt nicht ganz so schnell wie der ADP-Agent und weist eine höhere Variabilität auf, ist aber viel einfacher und bedingt sehr viel weniger Rechenaufwand pro Beobachtung. Beachten Sie, *dass TD kein Übergangsmodell braucht, um seine Aktualisierungen vorzunehmen*. Die Umgebung stellt die Verbindung zwischen den benachbarten Zuständen in Form beobachteter Übergänge bereit.

```
function PASSIVE-TD-AGENT(percept) returns eine Aktion
   inputs: percept, eine Wahrnehmung, die den aktuellen Zustand s'
                    und ein Belohnungssignal r' anzeigt
   persistent: π, eine feste Strategie
               U, eine Tabelle mit Nutzeneinträgen, anfänglich leer
              N_s, eine Tabelle mit Häufigkeitseinträgen für Zustände,
                   anfänglich null
               s, a, r der vorherige Zustand, die Aktion und die Belohnung,
                   anfänglich null

   if s' ist neu then U[s'] ← r'
   if s ist nicht null then
      inkrementiere N_s[s]
      U[s] ← U[s] + α(N_s[s])(r + γU[s'] - U[s])
   if s'.TERMINAL then s, a, r ← null else s, a, r ← s', π[s'], r'
   return α
```

Abbildung 21.4: Ein passiv verstärkend lernender Agent, der Nutzenschätzungen unter Verwendung temporaler Differenzen lernt. Die Schrittfunktion $\alpha(n)$ wurde gewählt, um die Konvergenz zu gewährleisten (siehe Text).

Der ADP-Ansatz und der TD-Ansatz sind sehr eng miteinander verwandt. Beide versuchen, lokale Anpassungen an den Nutzenschätzungen vorzunehmen, damit die einzelnen Zustände mit ihren Nachfolgern „übereinstimmen“. Ein Unterschied ist, dass TD einen Zustand anpasst, um eine Übereinstimmung mit seinem *beobachteten* Nachfolger zu erzeugen (Gleichung (21.3)), während ADP den Zustand anpasst, um eine Überein-

1 Die technischen Bedingungen sind in der Fußnote (7) zu *Abschnitt 18.6.3* angegeben. In Abbildung 21.5 haben wir $\alpha(n) = 60/(59 + n)$ verwendet, was die Bedingungen erfüllt.

stimmung mit *allen* möglicherweise auftretenden Nachfolgern zu erzielen, gewichtet nach ihren Wahrscheinlichkeiten (Gleichung (21.2)). Dieser Unterschied verschwindet, wenn die Wirkungen der TD-Anpassungen über eine große Anzahl von Übergängen gemittelt werden, weil die Häufigkeit jedes Nachfolgers in der Übergangsmenge annähernd proportional zu seiner Wahrscheinlichkeit ist. Ein wichtigerer Unterschied ist, dass TD nur eine einzige Anpassung pro beobachtetem Übergang macht, ADP dagegen so viele, wie nötig sind, um die Konsistenz zwischen den Nutzenschätzungen U und dem Umgebungsmodell P wiederherzustellen. Obwohl der beobachtete Übergang nur eine lokale Änderung in P vornimmt, kann es sein, dass seine Wirkungen durch das gesamte U weitergegeben werden müssen. Damit kann man TD als ungefähre, aber effiziente erste Annäherung an ADP betrachten.

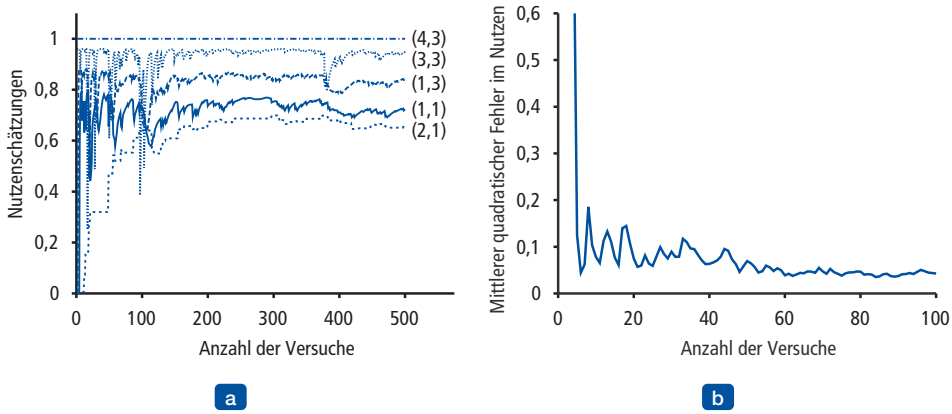


Abbildung 21.5: Die TD-Lernkurven für die 4 × 3-Welt. (a) Die Nutzenschätzungen für eine ausgewählte Untermenge der Zustände als Funktion der Anzahl an Versuchen. (b) Der mittlere quadratische Fehler in der Schätzung für $U(1,1)$, gemittelt über 20 Durchläufe von je 500 Versuchen. Hier sind nur die ersten 100 Versuche gezeigt, um einen Vergleich mit Abbildung 21.3 zu erlauben.

Jede von ADP vorgenommene Anpassung kann von der Perspektive von TD als Ergebnis einer „Pseudoerfahrung" betrachtet werden, die durch Simulieren des aktuellen Umgebungsmodells erzeugt wird. Der TD-Ansatz kann erweitert werden, sodass er ein Umgebungsmodell verwendet, um mehrere Pseudoerfahrungen zu erzeugen – Übergänge, von denen sich der TD-Agent vorstellen kann, dass sie für sein aktuelles Modell passieren *könnten*. Der TD-Agent kann für jeden beobachteten Übergang eine große Anzahl imaginärer Übergänge erzeugen. Auf diese Weise nähern sich die resultierenden Nutzenschätzungen immer weiter an diejenigen von ADP an – natürlich auf Kosten einer erhöhten Rechenzeit.

Auf ähnliche Weise können wir effizientere Versionen von ADP erzeugen, indem wir die Algorithmen für die Wert- oder Strategie-Iteration direkt annähern. Auch wenn der Algorithmus der Wert-Iteration effizient arbeitet, ist er bei einer sehr großen Anzahl von Zuständen – sagen wir 10^{100} – nicht mehr handhabbar. Viele der notwendigen Anpassungsschritte sind jedoch äußerst klein. Ein möglicher Ansatz, schnell ausreichend gute Antworten zu erhalten, wäre es, die Anzahl der nach jedem beobachteten Übergang erfolgten Anpassungen zu begrenzen. Man kann auch eine Heuristik einsetzen, um die möglichen Anpassungen zu ordnen, sodass nur die wichtigsten ausgeführt werden. Die Heuristik des **prioritätsgesteuerten Suchens** nimmt Anpassungen bevorzugt an den Zuständen vor, deren *wahrscheinliche* Nachfolger soeben

eine *große* Anpassung ihrer eigenen Nutzenschätzungen erfahren mussten. Mithilfe solcher Heuristiken können annähernde ADP-Algorithmen in der Regel fast so schnell wie vollständiges ADP lernen, was die Anzahl der Trainingsfolgen betrifft, weisen aber eine um mehrere Größenordnungen effizientere Berechnung auf (siehe Übung 21.2). Damit ist es möglich, dass sie Zustandsräume verarbeiten, die für vollständiges ADP viel zu groß wären. Annähernde ADP-Algorithmen haben einen zusätzlichen Vorteil: In den frühen Phasen des Lernens einer neuen Umgebung ist das Umgebungsmodell P häufig nicht korrekt. Es ist also wenig sinnvoll, eine genaue Nutzenfunktion zu berechnen, die mit ihm übereinstimmt. Ein Annäherungsalgorithmus kann eine minimale Anpassungsgröße verwenden, die sinkt, wenn das Umgebungsmodell genauer wird. Auf diese Weise vermeidet man die sehr langen Wert-Iterationen, die beim frühen Lernen aufgrund von großen Änderungen im Modell auftreten.

21.3 Aktives verstärkendes Lernen

Ein passiv lernender Agent hat eine feste Strategie, die sein Verhalten bestimmt. Ein aktiver Agent muss entscheiden, welche Aktionen er ausführt. Wir beginnen mit einem Agenten mit adaptiver dynamischer Programmierung und betrachten, wie er abgeändert werden muss, um diese neue Freiheit zu berücksichtigen.

Zuerst muss der Agent ein vollständiges Modell mit Ergebniswahrscheinlichkeiten für alle Aktionen lernen und nicht nur das Modell für die feststehende Strategie. Der einfache Lernalgorithmus von PASSIVE-ADP-AGENT ist gut dafür geeignet. Anschließend berücksichtigen wir die Tatsache, dass ein Agent eine Auswahlmöglichkeit zwischen den Aktionen hat. Die Nutzen, die er lernen muss, sind durch die *optimale* Strategie definiert; sie gehorchen den in *Abschnitt 17.2.1* gezeigten Bellman-Gleichungen, die wir hier wiederholen:

$$U(s) = R(s) + \gamma \max_a \sum_{s'} P(s' \mid s,a) U(s'). \tag{21.4}$$

Diese Gleichungen können gelöst werden, um die Nutzenfunktion U unter Verwendung der Algorithmen für die Wert-Iteration oder Strategie-Iteration aus *Kapitel 17* zu erhalten. Der letzte Aspekt ist, was in jedem Schritt erledigt werden soll. Wenn wir eine für das gelernte Modell optimale Nutzenfunktion U haben, kann der Agent eine optimale Aktion extrahieren, indem er einen Schritt vorausschaut, um den erwarteten Nutzen zu maximieren; verwendet er alternativ die Strategie-Iteration, steht die optimale Strategie bereits zur Verfügung und er soll einfach die Aktion ausführen, die die optimale Strategie empfiehlt. Soll er das wirklich?

21.3.1 Exploration

▶ Abbildung 21.6 zeigt die Ergebnisse einer Versuchsfolge für einen ADP-Agenten, der die Empfehlungen der optimalen Strategie für das gelernte Modell in jedem Schritt befolgt. Der Agent lernt *nicht* den tatsächlichen Nutzen oder die tatsächliche optimale Strategie! Stattdessen passiert Folgendes: Im 39. Versuch findet er eine Strategie, die die Belohnung +1 erreicht, nämlich über den unteren Weg mit (2,1), (3,1), (3,2) und (3,3) (siehe Abbildung 21.6(b)). Nach ein paar Experimenten mit kleineren Variationen bleibt er ab dem 276. Versuch bei dieser Strategie und lernt nie die Nutzen der anderen

Zustände und findet nie die optimale Route über (1,2), (1,3) und (2,3). Wir nennen diesen Agenten auch den **gierigen Agenten** (engl. **Greedy Agent**). Wiederholte Experimente zeigen, dass der gierige Agent nur *sehr selten* zu der optimalen Strategie für diese Umgebung und manchmal zu wirklich schrecklichen Strategien konvergiert.

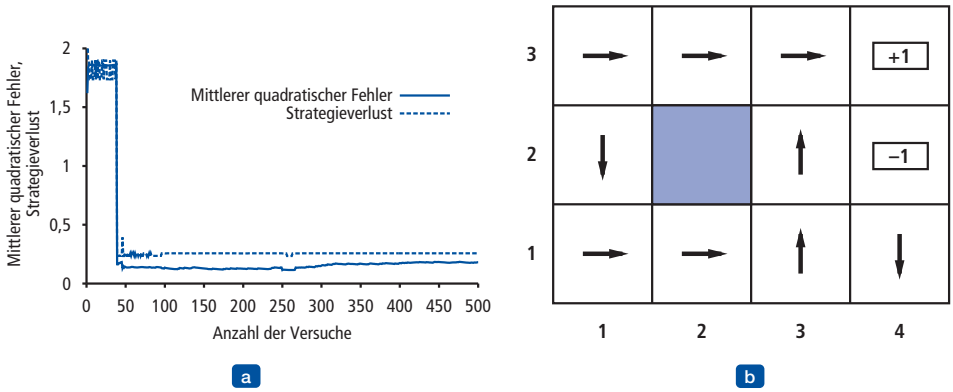


Abbildung 21.6: Leistung eines gierigen ADP-Agenten, der die Aktion ausführt, die von der optimalen Strategie für das gelernte Modell empfohlen wird. (a) Mittlerer quadratischer Fehler in den Nutzenschätzungen, gemittelt über die neun Nichtterminalzustände. (b) Die suboptimale Strategie, zu der der gierige Agent in der hier gezeigten Versuchsfolge konvergiert.

Wie kann es sein, dass die Auswahl der optimalen Aktion zu suboptimalen Ergebnissen führt? Die Antwort ist, dass das gelernte Modell nicht gleich der wahren Umgebung ist; was also im gelernten Modell optimal ist, kann in der tatsächlichen Umgebung suboptimal sein. Leider weiß der Agent nicht, wie die wahre Umgebung aussieht, deshalb kann er die optimale Aktion für die wahre Umgebung nicht berechnen. Was also ist zu tun?

Der gierige Agent hat übersehen, dass Aktionen nicht nur Belohnungen gemäß dem aktuell gelernten Modell nach sich ziehen; sie tragen auch zum Lernen des wahren Modells bei, indem sie die Wahrnehmungen beeinflussen, die empfangen werden. Durch die Verbesserung des Modells erhält der Agent in Zukunft höhere Belohnungen.[2] Ein Agent muss also eine Abwägung zwischen einer **Ausnutzung** für die Maximierung seiner Belohnung – wie in seinen aktuellen Nutzenschätzungen reflektiert – und einer **Exploration** zur Maximierung seines langfristigen Wohlergehens treffen. Eine reine Ausnutzung birgt die Gefahr, irgendwann in einem festgefahrenen Gleis stecken zu bleiben. Eine reine Exploration zur Verbesserung des eigenen Wissens hat keinen Wert, wenn man dieses Wissen niemals in die Praxis umsetzt. In der realen Welt muss man ständig zwischen der Weiterführung eines bequemen Lebens und dem Schritt ins Unbekannte mit der Hoffnung, ein neues und besseres Leben zu führen, entscheiden. Je mehr man weiß, desto weniger Exploration ist erforderlich.

Können wir etwas genauer sein? Gibt es eine *optimale* Explorationsstrategie? Es zeigt sich, dass diese Frage im Bereich der statistischen Entscheidungstheorie, die sich mit den sogenannten **Banditenproblemen** (siehe Einschub) beschäftigt, bereits detailliert betrachtet wurde.

Obwohl Banditenprobleme äußerst schwierig zu lösen sind, um genau eine *optimale* Explorationsmethode zu erhalten, ist es dennoch möglich, ein *sinnvolles* Schema zu fin-

2 Beachten Sie die direkte Analogie zur Theorie des Informationswertes aus *Kapitel 16*.

den, das irgendwann zum optimalen Verhalten des Agenten führt. Technisch betrachtet muss ein solches Schema „gierig“ innerhalb der Grenzen der unendlichen Exploration sein, **GLIE** (Greedy in the Limit of Infinite Exploration). Ein GLIE-Schema muss jede Aktion in jedem Zustand unbegrenzt oft ausprobieren, um zu vermeiden, dass eine optimale Aktion übersehen wird, weil eine unüblich schlechte Ergebnisfolge aufgetreten ist. Ein ADP-Agent, der ein solches Schema verwendet, lernt irgendwann das wahre Umgebungsmodell. Ein GLIE-Schema muss darüber hinaus irgendwann gierig werden, sodass die Aktionen des Agenten optimal für das gelernte (und damit das wahre) Modell werden.

Exploration und Banditen

In Las Vegas ist ein *einarmiger Bandit* eine Glücksspielmaschine. Ein Spieler kann eine Münze einwerfen, am Hebel ziehen und die Gewinne (falls er gewinnt) einsammeln. Ein ***n*-armiger Bandit** hat *n* Hebel. Der Spieler muss wählen, welcher Hebel für die jeweils eingeworfene Münze gezogen werden soll – derjenige, der bisher am meisten Gewinn erbracht hat, oder vielleicht derjenige, der noch überhaupt nicht ausprobiert wurde?

Das Problem des *n*-armigen Banditen ist ein formales Modell für reale Probleme in vielen wichtigen Bereichen, wie beispielsweise bei der Entscheidung über den Jahresetat für die KI- Forschung und -Entwicklung. Jeder Arm entspricht einer Aktion (wie beispielsweise der Zuteilung von 20 Millionen Euro für die Entwicklung neuer KI-Lehrbücher) und der Gewinn durch das Ziehen am Arm entspricht den (immensen!) Vorteilen, die aus der Aktion entstehen. Die Exploration, egal ob eines neuen Forschungsgebietes oder einer neuen Shopping-Mall, ist risikoreich, teurer und birgt unsichere Gewinne; führt man jedoch überhaupt keine Exploration aus, entdeckt man *nie* irgendwelche Aktionen, die das wert sind.

Um das Banditenproblem korrekt zu formulieren, muss man genau definieren, was unter einem optimalen Verhalten zu verstehen ist. Die meisten Definitionen in der Literatur gehen davon aus, das Ziel sei, den erwarteten Gesamtgewinn über die Lebensdauer des Agenten zu maximieren. Diese Definitionen fordern, dass die Erwartungen über alle möglichen Welten aufgespannt werden, in denen sich der Agent bewegen kann, ebenso wie über die möglichen Ergebnisse jeder Aktionsfolge in jeder einzelnen Welt. Hier ist eine „Welt“ durch das Übergangsmodell $P(s' \mid s,a)$ definiert. Um also optimal zu handeln, braucht der Agent eine unbedingte Verteilung über die möglichen Modelle. Die resultierenden Optimierungsprobleme sind in der Regel äußerst schwer zu handhaben.

In einigen Fällen – beispielsweise, wenn die Auszahlung jeder Maschine unabhängig ist und verminderte Gewinne verwendet werden – ist es möglich, einen **Gittins-Index** für jede Spielmaschine zu berechnen (Gittins, 1989). Der Index ist eine Funktion der Anzahl, wie oft eine Spielmaschine gespielt wurde und wie viel sie ausbezahlt hat. Der Index für eine Maschine zeigt an, wie lohnenswert es ist, mehr zu investieren. Allgemein ausgedrückt: Je höher der erwartete Gewinn und je höher die Unsicherheit im Nutzen einer bestimmten Chance, desto besser. Die Auswahl der Maschine mit dem höchsten Indexwert stellt eine optimale Explorationsstrategie dar. Leider wurde bisher keine Möglichkeit gefunden, den Gittins-Index auf sequenzielle Entscheidungsprobleme zu erweitern.

Man kann die Theorie der *n*-armigen Banditen nutzen, um zu entscheiden, wie sinnvoll die Auswahlstrategie in genetischen Algorithmen ist (siehe *Kapitel 4*). Wenn Sie jeden Arm in einem *n*-armigen Banditenproblem als mögliche Genkette und die Investition einer Münze in einen Arm als die Reproduktion dieser Gene betrachten, lässt sich beweisen, dass genetische Algorithmen die Münzen optimal zuteilen, wenn eine geeignete Menge von Unabhängigkeitsannahmen gegeben ist.

Es gibt mehrere GLIE-Schemas; bei einem der einfachsten lässt man den Agenten in einem Bruchteil $1/t$ der Zeit eine zufällige Aktion auswählen, während er die übrige Zeit der gierigen Strategie folgen muss. Dies konvergiert zwar irgendwann zu einer optimalen Strategie, kann aber extrem langsam sein. Ein sinnvollerer Ansatz wäre es, Aktionen eine bestimmte Gewichtung zuzuordnen, die der Agent nicht sehr häufig ausprobiert hat, und Aktionen, von denen man einen geringen Nutzen annimmt, eher zu vermeiden. Das kann implementiert werden, indem man die Bedingungsgleichungen (21.4) so abändert, dass sie relativ schlecht erkundeten Zustand/Aktion-Paaren eine höhere Nutzenschätzung zuweisen. Im Wesentlichen führt dies zu einer optimistischen unbedingten Verteilung über die möglichen Umgebungen und bewirkt, dass sich der Agent anfänglich so verhält, als wären über den gesamten Raum wunderbare Gewinne verteilt. Wir verwenden $U^+(s)$, um die optimistische Schätzung für den Nutzen des Zustandes s darzustellen (d.h. den erwarteten Gewinn); außerdem sei $N(s, a)$ die Anzahl, wie oft die Aktion a im Zustand s ausprobiert wurde. Nehmen wir an, wir verwenden die Wert-Iteration in einem ADP-lernenden Agenten; wir müssen die Aktualisierungsgleichung umschreiben (d.h. *Gleichung (17.6)*), um die optimistische Schätzung zu berücksichtigen. Das erledigt die folgende Gleichung:

$$U^+(s) \leftarrow R(s) + \gamma \max_a f\left(\sum_{s'} P(s' \mid s,a) U^+(s'), N(s,a)\right). \tag{21.5}$$

Hier wird $f(u, n)$ als die **Explorationsfunktion** bezeichnet. Sie bestimmt, wie die Gier (also die Bevorzugung höherer Werte von u) gegen die Neugier (die Bevorzugung niedrigerer Werte von n, also von Aktionen, die noch nicht häufig ausprobiert wurden) abgewogen wird. Die Funktion $f(u, n)$ sollte in u steigen und in n fallen. Offensichtlich gibt es viele mögliche Funktionen, die diese Bedingungen erfüllen. Eine besonders einfache Definition ist:

$$f(u,n) = \begin{cases} R^+ & \text{wenn } n < N_e \\ u & \text{andernfalls.} \end{cases}$$

Dabei ist R^+ eine optimistische Schätzung der in jedem Zustand bestmöglichen Belohnung und N_e ein feststehender Parameter. Das bewirkt, dass der Agent jedes Aktion/Zustand-Paar mindestens N^e-mal ausprobiert.

Die Tatsache, dass U^+ statt U auf der rechten Seite von Gleichung (21.5) erscheint, ist sehr wichtig. Bei fortschreitender Exploration kann es sein, dass die Zustände und Aktionen nahe dem Startzustand sehr oft ausprobiert wurden. Hätten wir U verwendet, die pessimistischere Nutzenschätzung, dann wäre der Agent bald nicht mehr bereit, weitere Explorationen vorzunehmen. Die Verwendung von U^+ bedeutet, dass die Vorteile der Exploration von den Kanten nicht erkundeter Regionen zurückgereicht werden, sodass Aktionen, die *in die Richtung* nicht erkundeter Regionen führen, höher gewichtet werden als Aktionen, die einfach nur unvertraut sind. Die Wirkung dieser Explorationsstrategie wird in ▶ Abbildung 21.7 deutlich, die eine schnelle Konvergenz in Richtung der optimalen Leistung zeigt, anders als der gierige Ansatz. Eine fast optimale Strategie wird nach nur 18 Versuchen gefunden. Beachten Sie, dass die eigentlichen Nutzenschätzungen nicht so schnell konvergieren. Das liegt daran, dass der Agent relativ schnell aufhört, nicht gewinnbringende Teile des Zustandsraumes zu erkunden, und sie danach nur noch „durch Zufall" besucht. Es ist jedoch durchaus sinnvoll für den Agenten, sich keine

Gedanken über die genauen Nutzen von Zuständen zu machen, von denen er weiß, dass sie unerwünscht sind und vermieden werden können.

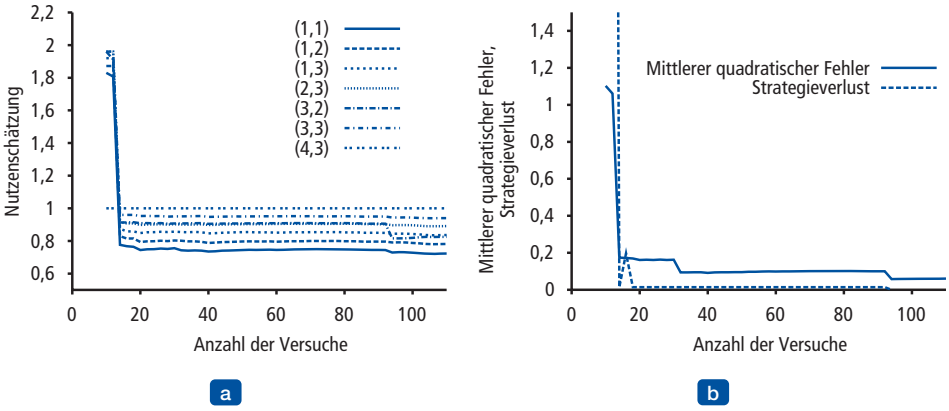


Abbildung 21.7: Leistung des explorativen ADP-Agenten unter Verwendung von $R^+ = 2$ und $N_e = 5$. (a) Nutzenschätzungen für ausgewählte Zustände über die Zeit. (b) Der mittlere quadratische Fehler in Nutzenwerten und der zugehörige Strategieverlust.

21.3.2 Lernen einer Aktion/Wert-Funktion

Nachdem wir einen aktiven ADP-Agenten haben, wollen wir überlegen, wie wir einen aktiven, temporale Differenzen lernenden Agenten erstellen können. Die offensichtlichste Veränderung gegenüber dem passiven Fall ist, dass der Agent nicht mehr mit einer feststehenden Strategie ausgestattet ist; wenn er also eine Nutzenfunktion U lernt, muss er ein Modell lernen, um in der Lage zu sein, basierend auf U nach einer einschrittigen Vorausschau eine Aktion auszuwählen. Das Problem, ein Modell für den TD-Agenten auszuwählen, ist identisch mit dem für den ADP-Agenten. Was ist mit der eigentlichen TD-Aktualisierungsregel? Es mag vielleicht überraschend sein, aber die Aktualisierungsregel (21.3) bleibt unverändert. Das kann aus dem folgenden Grund seltsam erscheinen: Angenommen, der Agent führt einen Schritt aus, der normalerweise zu einem guten Ziel führt, aber aufgrund der nicht deterministischen Umgebung endet der Agent in einem katastrophalen Zustand. Die TD-Aktualisierungsregel nimmt dies so ernst, als sei das Ergebnis das normale Resultat der Aktion gewesen, während man annehmen kann, dass sich der Agent nicht zu viele Sorgen darüber machen sollte, weil das Ergebnis ein Zufallstreffer war. Natürlich tritt das unwahrscheinliche Ergebnis in einer großen Menge von Trainingsfolgen nur sehr selten auf; auf lange Sicht werden also seine Wirkungen proportional zu seiner Wahrscheinlichkeit gewichtet – hoffentlich. Auch hier kann gezeigt werden, dass der TD-Algorithmus auf dieselben Werte wie ADP konvergiert, wenn die Anzahl der Trainingsfolgen gegen unendlich geht.

Es gibt eine alternative TD-Methode, das sogenannte ***Q*-Lernen**, das statt Nutzen eine Aktion/Nutzen-Repräsentation lernt. Mit der Notation $Q(s, a)$ bezeichnen wir den Wert der Ausführung von Aktion a im Zustand s. Q-Werte sind wie folgt direkt mit Nutzenwerten verknüpft:

$$U(s) = \max_a Q(s,a). \tag{21.6}$$

Tipp

Q-Funktionen scheinen einfach nur eine weitere Möglichkeit zu sein, Nutzeninformationen zu speichern, aber sie haben eine sehr wichtige Eigenschaft: *Ein TD-Agent, der eine Q-Funktion lernt, braucht weder für das Lernen noch die Aktionsauswahl ein Modell der Form* $P(s'|s,a)$. Aus diesem Grund sagt man auch, das Q-Lernen ist eine **modellfreie Methode**. Wie für die Nutzen können wir eine Bedingungsgleichung schreiben, die im Gleichgewicht gelten muss, wenn die Q-Werte korrekt sind:

$$Q(s,a) = R(s) + \lambda \sum_{s'} P(s'|s,a) \max_{a'} Q(s',a'). \tag{21.7}$$

Wie beim ADP-lernenden Agenten können wir diese Gleichung direkt als Aktualisierungsgleichung für einen Iterationsprozess verwenden, der genaue Q-Werte berechnet, wenn ein geschätztes Modell gegeben ist. Das bedingt jedoch, dass auch ein Modell gelernt wird, weil die Gleichung $P(s'|s,a)$ verwendet. Der Ansatz der temporalen Differenz dagegen benötigt kein Modell von Zustandsübergängen – es sind einzig Q-Werte erforderlich. Die Aktualisierungsgleichung für TD-Q-Lernen lautet:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)). \tag{21.8}$$

Sie wird berechnet, wenn die Aktion a im Zustand s ausgeführt wird und zu Zustand s' führt.

▶ Abbildung 21.8 zeigt den vollständigen Agentenentwurf für einen explorativen Q-lernenden Agenten unter Verwendung von TD. Beachten Sie, dass er genau dieselbe Explorationsfunktion f verwendet wie der explorative ADP-Agent – deshalb ist es nicht erforderlich, Statistiken über die ausgeführten Aktionen (die Tabelle N) zu führen. Wenn eine einfachere Explorationsstrategie verwendet wird – etwa ein zufälliges Agieren für einen bestimmten Anteil an Schritten, wobei dieser Anteil mit der Zeit geringer wird –, können wir ohne die Statistik auskommen.

```
function Q-LEARNING-AGENT(percept) returns eine Aktion
   inputs: percept, eine Wahrnehmung, die den aktuellen Zustand s' und
                    ein Belohnungssignal r' angibt
   persistent: Q, eine Tabelle mit Aktionswerten, indiziert nach
                  Zustand und Aktion, anfangs null
               N_sa, eine Tabelle mit Häufigkeiten für Zustand/Aktion-Paare,
                  anfangs null
               s, a, r, der vorhergehende Zustand, Aktion und Belohnung,
                  anfänglich null

   if TERMINAL?(s) then Q[s,None] ← r'
   if s ist nicht null then
      inkrementiere N_sa[s, a]
      Q[s, a] ← Q[s, a] + α(N_sa[s, a])(r + γ max_a' Q[a', a'] - Q[s, a])
   s, a, r ← s', argmax_a' f(Q[s', a'], N_sa[s', a'], r'
   return a
```

Abbildung 21.8: Ein explorativer Q-lernender Agent. Es handelt sich dabei um einen aktiven Lernalgorithmus, der den Wert $Q(s, a)$ jeder Aktion in jeder Situation lernt. Er verwendet dieselbe Explorationsfunktion f wie der explorative ADP-Agent, vermeidet jedoch, das Übergangsmodell lernen zu müssen, weil der Q-Wert eines Zustandes direkt mit dem seiner Nachbarn in Bezug gebracht werden kann.

Q-Lernen ist eng verwandt mit dem sogenannten **SARSA**-Algorithmus (für State-Action-Reward-State-Action – Zustand-Aktion-Belohnung-Zustand-Aktion). Die Aktualisierungsregel für SARSA ähnelt Gleichung (21.8):

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma Q(s',a') - Q(s,a)). \tag{21.9}$$

Hier ist a' die Aktion, die im Zustand s' *tatsächlich ausgeführt* wird. Die Regel wird am Ende des Quintupel s, a, r, s', a' ausgeführt – daher der Name. Der Unterschied zum Q-Lernen ist recht subtil: Q-Lernen speichert den *besten* Q-Wert vom Zustand, der im beobachteten Übergang erreicht wurde, während SARSA wartet, bis eine Aktion tatsächlich ausgeführt wird, und speichert dann den Q-Wert für diese Aktion. Für einen gierigen Agenten, der immer die Aktion mit dem besten Q-Wert unternimmt, sind nun die beiden Algorithmen identisch. Allerdings weichen sie deutlich voneinander ab, sobald Exploration stattfindet. Da Q-Lernen den besten Q-Wert verwendet, achtet der Algorithmus nicht auf die eigentliche Strategie, der gefolgt wird – es handelt sich um einen **strategiefernen** Lernalgorithmus, während SARSA ein **strategieorientierter** Algorithmus ist. Q-Lernen ist flexibler als SARSA, da ein Q-lernender Agent „gutes Verhalten“ lernen kann, selbst wenn er von einer zufälligen oder willkürlichen Explorationsstrategie geleitet wird. Andererseits ist SARSA realistischer: Wird zum Beispiel die Gesamtstrategie teilweise durch andere Agenten gesteuert, ist es besser, eine Q-Funktion zu lernen, für das, was wirklich passiert, anstatt für das, was der Agent passieren lassen möchte.

Sowohl der Q-lernende Agent als auch der SARSA-Agent lernen die optimale Strategie für die 4×3-Welt, allerdings sehr viel langsamer als der ADP-Agent. Das liegt daran, dass die lokalen Aktualisierungen keine Konsistenz unter allen Q-Werten über das Modell erzwingen. Der Vergleich wirft eine allgemeine Frage auf: Ist es besser, ein Modell und eine Nutzenfunktion zu lernen, oder ist es besser, eine Aktion/Nutzen-Funktion ohne Modell zu lernen? Mit anderen Worten, wie stellt man die Agentenfunktion am besten dar? Dies ist eine grundlegende Fragestellung der KI. Wie wir in *Kapitel 1* gesagt haben, ist eine der wichtigsten historischen Eigenschaften eines Großteils der KI-Forschung ihre (häufig unerwähnte) Anlehnung an den **wissensbasierten** Ansatz. Das führt zu der Annahme, dass die Agentenfunktion am besten dargestellt wird, indem eine Repräsentation bestimmter Aspekte der Umgebung gefunden wird, in der sich der Agent befindet.

Einige Forscher, sowohl innerhalb als auch außerhalb der KI, behaupteten, es gäbe modellfreie Methoden, wie beispielsweise das Q-Lernen, was bedeuten würde, dass der wissensbasierte Ansatz überflüssig ist. Hierbei steht uns aber nicht viel mehr als Intuition zur Verfügung. Unsere Intuition sagt uns, dass die Vorteile eines wissensbasierten Ansatzes umso deutlicher werden, je komplexer die Umgebung wird. Das erkennt man sogar in Spielumgebungen, wie beispielsweise bei Schach, Dame oder Backgammon (siehe nächster Abschnitt), wo das Lernen einer Bewertungsfunktion mithilfe eines Modells zu besseren Erfolgen als Q-lernende Methoden geführt hat.

21.4 Verallgemeinerung beim verstärkenden Lernen

Bisher sind wir davon ausgegangen, dass die Nutzenfunktionen und die von den Agenten gelernten Q-Funktionen in tabellarischer Form mit einem Ausgabewert für jedes Eingabetupel vorliegen. Ein solcher Ansatz funktioniert ausreichend gut für kleine Zustandsräume, aber die Zeit bis zur Konvergenz und (für ADP) die Zeit pro Iteration steigt mit wachsendem Raum rapide an. Bei sorgfältig kontrollierten annähernden ADP-Methoden kann es möglich sein, 10.000 oder mehr Zustände zu verar-

beiten. Das ist ausreichend für zweidimensionale labyrinthähnliche Umgebungen, jedoch keinesfalls für realistische Welten. Backgammon und Schach sind winzige Untermengen der realen Welt und dennoch enthalten ihre Zustandsräume rund 10^{20} bzw. 10^{40} Zustände. Es wäre absurd anzunehmen, dass man alle diese Zustände besuchen muss, um zu lernen, wie das Spiel gespielt wird!

Eine Möglichkeit, mit solchen Problemen zurechtzukommen, ist die **Funktionsannäherung**, die einfach eine andere Darstellung als eine Suchtabelle für die Q-Funktion verwendet. Die Repräsentation wird als Annäherung betrachtet, weil es vielleicht nicht möglich ist, dass die *wahre* Nutzenfunktion oder Q-Funktion in der gewählten Form dargestellt werden kann. In *Kapitel 5* beispielsweise haben wir eine **Bewertungsfunktion** für Schach geschrieben, die als gewichtete Linearfunktion einer Menge von **Merkmalen** (oder **Basisfunktionen**) f_1, ..., f_n dargestellt wird:

$$\hat{U}_\theta(s) = \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s).$$

Ein Algorithmus zum verstärkenden Lernen kann Werte für die Parameter $\theta = \theta_1, \ldots \theta_n$ lernen, sodass die Bewertungsfunktion $\hat{U}_\theta$ die wahre Nutzenfunktion annähert. Statt beispielsweise durch 10^{40} Werte in einer Tabelle wird diese Funktionsannäherung durch etwa $n = 20$ Parameter charakterisiert – eine *riesige* Komprimierung. Niemand kennt die wahre Nutzenfunktion für Schach, aber niemand glaubt, dass sie in genau 20 Zahlen dargestellt werden kann. Wenn die Annäherung jedoch gut genug ist, kann der Agent dennoch ausgezeichnet Schach spielen.[3]

Tipp

Die Funktionsannäherung ermöglicht es, Nutzenfunktionen für sehr große Zustandsräume darzustellen, aber das ist nicht ihr wichtigster Vorteil. *Die durch eine Funktionsannäherung erzielte Komprimierung erlaubt es dem lernenden Agenten, von bereits besuchten auf nicht besuchte Zustände zu verallgemeinern*. Das bedeutet, der wichtigste Aspekt einer Funktionsannäherung ist nicht, dass sie weniger Speicherplatz benötigt, sondern dass sie eine induktive Verallgemeinerung über die Eingabezustände erlaubt. Um Ihnen eine Vorstellung von der Leistungsfähigkeit dieses Effekts zu geben, betrachten Sie Folgendes: Durch die Auswertung nur eines von je 10^{12} möglichen Backgammonzuständen ist es möglich, eine Nutzenfunktion zu lernen, die es einem Programm erlaubt, so gut wie jeder Mensch zu spielen (Tesauro, 1992).

Die Kehrseite ist jedoch, dass es möglicherweise im gewählten Hypothesenraum keine Funktion gibt, die die wahre Nutzenfunktion ausreichend gut annähert. Wie bei jedem induktiven Lernen existiert ein Konflikt zwischen der Größe des Hypothesenraumes und der Zeit, die es dauert, die Funktion zu lernen. Ein größerer Hypothesenraum erhöht die Wahrscheinlichkeit, dass eine gute Annäherung gefunden werden kann, bedeutet aber auch, dass die Konvergenz wahrscheinlich verzögert wird.

Wir beginnen mit dem einfachsten Fall, der direkten Nutzenschätzung (siehe *Abschnitt 21.2*). Für die Funktionsannäherung handelt es sich hierbei um ein Beispiel für **überwachtes Lernen**. Angenommen, wir stellen die Nutzen für die 4×3-Welt mit-

3 Wir wissen, dass die genaue Nutzenfunktion auf einer oder zwei Seiten Lisp, Java oder C++ dargestellt werden kann. D.h., sie kann von einem Programm dargestellt werden, das das Spiel genau löst, wenn es aufgerufen wird. Wir sind nur an Funktionsannäherungen interessiert, die einen *sinnvollen* Rechenaufwand betreiben. Es könnte besser sein, eine sehr einfache Funktionsannäherung zu lernen und sie mit einer vorausschauenden Suche zu kombinieren. Die dafür erforderlichen Abwägungen sind jedoch noch nicht wirklich gut erforscht.

hilfe einer einfachen Linearfunktion dar. Die Merkmale der Quadrate sind einfach nur ihre x- und y-Koordinaten, wir haben demnach:

$$\hat{U}_\theta(x, y) = \theta_0 + \theta_1 x + \theta_2 y. \tag{21.10}$$

Wenn also $(\theta_0, \theta_1, \theta_2) = (0{,}5, 0{,}2, 0{,}1)$, dann ist $\hat{U}_\theta(1, 1) = 0{,}8$. Für eine Menge von Versuchen erhalten wir eine Menge von Stichprobenwerten von $\hat{U}_\theta(x, y)$ und wir finden die beste Übereinstimmung im Hinblick auf eine Minimierung des Quadratfehlers unter Verwendung einer linearen Standardregression (siehe *Kapitel 18*).

Für das verstärkende Lernen ist es sinnvoller, einen *Online*-Lernalgorithmus zu verwenden, der die Parameter nach jedem Versuch aktualisiert. Angenommen, wir führen einen Versuch aus und der Gesamtgewinn für den Start bei (1, 1) ist 0,4. Das weist darauf hin, dass $\hat{U}_\theta(1, 1)$ mit aktuell 0,8 zu groß ist und reduziert werden muss. Wie sollen die Parameter dafür angepasst werden? Wie beim Erlernen neuronaler Netze schreiben wir eine Fehlerfunktion und berechnen ihren Gradienten im Hinblick auf die Parameter. Wenn $u_j(s)$ der beobachtete Gesamtgewinn ab dem Zustand s im j-ten Versuch ist, ist der Fehler definiert als die (halbe) Quadratdifferenz des vorhergesagten Gesamtbetrages und des tatsächlichen Gesamtbetrages: $E_j(s) = (\hat{U}_\theta(s) - u_j(s))^2/2$. Die Änderungsrate des Fehlers im Hinblick auf jeden Parameter θ_i ist $\partial E_j/\partial\theta_i$. Um also den Parameter in die Richtung des sinkenden Fehlers zu verschieben, brauchen wir:

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j(s)}{\partial \theta_i} = \theta_i + \alpha(u_j(s) - \hat{U}(s)) \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}. \tag{21.11}$$

Man spricht auch von der **Widrow-Hoff-Regel** oder der **Delta-Regel** für die kleinsten Online-Quadrate. Für die lineare Funktionsannäherung $\hat{U}_\theta(s)$ in Gleichung (21.10) erhalten wir drei einfache Aktualisierungsregeln:

$$\begin{aligned}
\theta_0 &\leftarrow \theta_0 + \alpha(u_j(s) - \hat{U}_\theta(s)) \\
\theta_1 &\leftarrow \theta_1 + \alpha(u_j(s) - \hat{U}_\theta(s)) x \\
\theta_2 &\leftarrow \theta_2 + \alpha(u_j(s) - \hat{U}_\theta(s)) y.
\end{aligned}$$

Wir können diese Regeln auf das Beispiel anwenden, wo $\hat{U}_\theta(1, 1)$ gleich 0,8 und $u_j(1, 1)$ gleich 0,4 ist. θ_0, θ_1 und θ_2 werden alle um $0{,}4\alpha$ verringert, wodurch der Fehler für (1, 1) reduziert wird. *Beachten Sie, dass eine Änderung der Parameter θ als Reaktion auf einen beobachteten Übergang zwischen zwei Zuständen auch die Werte von $\hat{U}_\theta$ für jeden anderen Zustand ändert!* Genau das meinen wir damit, wenn wir sagen, die Funktionsannäherung erlaubt es einem verstärkenden Lernalgorithmus, aus seinen Erfahrungen zu verallgemeinern.

Wir erwarten, dass der Agent schneller lernt, wenn er eine Funktionsannäherung verwendet, vorausgesetzt, der Hypothesenraum ist nicht zu groß, enthält aber einige Funktionen, die eine ausreichend gute Übereinstimmung mit der wahren Nutzenfunktion erbringen. In Übung 21.6 werden Sie die Leistung der direkten Nutzenschätzung mit und ohne Funktionsannäherung bewerten. Die Verbesserung in der 4×3-Welt ist bemerkenswert, aber nicht dramatisch, weil es sich hier um einen sehr kleinen Zustandsraum handelt. In einer 10×10-Welt mit einem Gewinn von +1 an der Position (10, 10) ist die Verbesserung sehr viel höher. Diese Welt ist gut geeignet für eine lineare Nutzenfunktion, weil die wahre Nutzenfunktion glatt und fast linear ist (siehe Übung 21.8). Wenn wir die

Belohnung +1 an der Stelle (5, 5) platzieren, sieht der wahre Nutzen eher wie eine Pyramide aus und die Funktionsannäherung in Gleichung (21.10) schlägt kläglich fehl. Es ist jedoch noch nicht aller Tage Abend! Sie wissen, dass es bei der linearen Funktionsannäherung vor allem darauf ankommt, dass die Funktion linear in den *Parametern* ist – die eigentlichen Merkmale können beliebige nichtlineare Funktionen der Zustandsvariablen sein. Wir können also einen Term $\theta_3 f_3(x, y) = \theta_3\sqrt{(x - x_g)^2 + (y - y_g)^2}$ einfügen, der die Distanz zum Ziel misst.

Wir können diese Ideen auch auf Lernalgorithmen mit temporaler Differenz anwenden. Wir brauchen nur die Parameter anzupassen, um damit zu versuchen, die temporale Differenz zwischen aufeinanderfolgenden Zuständen zu reduzieren. Die neuen Versionen der TD- und Q-lernenden Gleichungen (21.3 und 21.8) lauten

$$\theta_i \leftarrow \theta_i - \alpha[R(s) + \gamma\hat{U}_\theta(s) - \hat{U}_\theta(s)]\frac{\partial\hat{U}_\theta(s)}{\partial\theta_i} \tag{21.12}$$

für Nutzen und

$$\theta_i \leftarrow \theta_i - \alpha[R(s) + \gamma\max_{a'}\hat{Q}_\theta(s',a') - \hat{Q}_\theta(s,a)]\frac{\partial\hat{Q}_\theta(s,a)}{\partial\theta_i} \tag{21.13}$$

für Q-Werte. Für passives TD-Lernen kann man zeigen, dass die Aktualisierungsregel auf die nächstmögliche[4] Annäherung zur wahren Funktion konvergiert, wenn die Funktionsannäherung *linear* in den Parametern ist. Bei aktivem Lernen und *nichtlinearen* Funktionen wie z.B. neuronalen Netzen ist alles möglich: Es gibt einige sehr einfache Fälle, in denen die Parameter gegen unendlich gehen können, selbst wenn es gute Lösungen im Hypothesenraum gibt. Es gibt komplexere Algorithmen, die diese Probleme vermeiden können, aber nach wie vor bleibt das verstärkende Lernen mit allgemeinen Funktionsannäherungen ein heikler Fall.

Die Funktionsannäherung kann auch sehr praktisch für das Lernen eines Umgebungsmodells sein. Beachten Sie, dass das Lernen eines Modells für eine *beobachtbare* Umgebung ein Problem des *überwachten* Lernens ist, weil die nächste Wahrnehmung den Ergebniszustand ergibt. Es kann jede der in *Kapitel 18* beschriebenen Lernmethoden verwendet werden, wenn geeignete Anpassungen für die Tatsache vorgenommen werden, dass wir eine vollständige Zustandsbeschreibung vorhersagen müssen und nicht nur eine boolesche Klassifizierung oder einen einzigen reellen Wert. Für eine *partiell beobachtbare* Umgebung ist das Lernproblem sehr viel schwieriger. Wenn wir die verborgenen Variablen kennen und wissen, wie in welcher kausalen Beziehung sie zueinander und zu den beobachtbaren Variablen stehen, können wir die Struktur eines dynamischen Bayesschen Netzes verwenden und die Parameter mithilfe des EM-Algorithmus lernen, wie in *Kapitel 20* beschrieben. Das Finden der verborgenen Variablen und das Lernen der Modellstruktur sind immer noch offene Probleme. *Abschnitt 21.6* beschreibt einige praktische Beispiele.

4 Die Definition der Distanz zwischen Nutzenfunktionen ist relativ technisch; siehe Tsitsiklis und Van Roy (1997).

21.5 Strategiesuche

Der letzte Ansatz, den wir für das Problem des verstärkenden Lernens beschreiben wollen, ist die sogenannte **Strategiesuche**. In gewisser Weise ist die Strategiesuche die einfachste aller Methoden in diesem Kapitel: Die Idee dabei ist, die Strategie so lange anzupassen, wie sich ihre Leistung verbessert, und die Anpassungen dann zu beenden.

Wir beginnen mit den eigentlichen Strategien. Sie wissen, dass eine Strategie π eine Funktion ist, die Zustände auf Aktionen abbildet. Wir sind hauptsächlich an *parametrisierten* Darstellungen von π interessiert, die sehr viel weniger Parameter aufweisen, als es Zustände im Zustandsraum gibt (so wie im vorigen Abschnitt). Beispielsweise könnten wir π durch eine Menge parametrisierter Q-Funktionen darstellen, eine für jede Aktion, und die Aktion mit dem höchsten vorhergesagten Wert verwenden:

$$\pi(s) = \max_a \hat{Q}_\theta(s,a). \tag{21.14}$$

Tipp

Jede Q-Funktion könnte eine lineare Funktion der Parameter θ sein, wie in Gleichung (21.10), oder sie könnte eine nichtlineare Funktion sein, wie etwa ein neuronales Netz. Die Strategiesuche passt dann die Parameter θ an, um die Strategie zu verbessern. Wird die Strategie durch Q-Funktionen dargestellt, führt die Strategiesuche zu einem Prozess, der Q-Funktionen lernt. *Dieser Prozess ist nicht dasselbe wie das Q-Lernen*! Beim Q-Lernen mit Funktionsannäherung findet der Algorithmus einen Wert von θ, sodass $\hat{Q}_\theta$ „nahe Q^*“ liegt, der optimalen Q-Funktion. Die Strategiesuche dagegen findet einen Wert von θ, der eine gute Leistung erbringt; die gefundenen Werte können sich wesentlich voneinander unterscheiden. (Zum Beispiel liefert die annähernde Q-Funktion, die durch $\hat{Q}_\theta(s, a) = Q^*(s, a)/10$ definiert ist, optimale Leistung, selbst wenn sie nicht ganz so nahe an Q^* liegt.) Ein weiteres deutliches Beispiel für die Unterscheidung ist der Fall, in dem $\pi(s)$ beispielsweise mithilfe einer zehn Schritte vorausschauenden Suche mit einer annähernden Nutzenfunktion $\hat{U}_\theta$ berechnet wird. Der Wert von θ, der zu einem guten Ergebnis führt, kann weit davon entfernt sein, $\hat{U}_\theta$ der wahren Nutzenfunktion ähnlich zu machen.

Ein Problem der Strategierepräsentationen, wie sie in Gleichung (21.14) gezeigt sind, ist, dass die Strategie eine *unstetige* Funktion der Parameter ist, wenn die Aktionen diskret sind. (Für einen stetigen Aktionsraum kann die Strategie eine glatte Funktion der Parameter sein.) Das bedeutet, es gibt Werte von θ, sodass eine infinitesimale Änderung in θ bewirkt, dass die Strategie von einer Aktion zu einer anderen wechselt. Das bedeutet, der Wert der Strategie kann sich ebenfalls unstetig ändern, was eine gradientenbasierte Suche schwer macht. Aus diesem Grund verwenden die Strategiesuchmethoden häufig eine **stochastische Strategierepräsentation** $\pi_\theta(s, a)$, die die *Wahrscheinlichkeit* spezifiziert, eine Aktion a im Zustand s auszuwählen. Eine beliebte Repräsentation ist die **Softmax-Funktion**:

$$\pi_\theta(s,a) = e^{\hat{Q}_\theta(s,a)} / \sum_{a'} e^{\hat{Q}_\theta(s,a')}.$$

Softmax wird nahezu deterministisch, wenn eine Aktion sehr viel besser als die anderen Aktionen ist, ergibt aber immer eine differenzierbare Funktion von θ; damit ist der Wert der Strategie (die in stetiger Weise von den Aktionsauswahl-Wahrscheinlichkeiten abhängig ist) eine differenzierbare Funktion von θ. Softmax ist eine Verallgemeinerung der logistischen Funktion (*Abschnitt 18.6.4*) auf mehrere Variablen.

Jetzt betrachten wir Methoden für die Verbesserung der Strategie. Wir beginnen mit dem einfachsten Fall: eine deterministische Strategie und eine deterministische Umgebung. Es sei $\rho(\theta)$ der **Strategiewert**, d.h. der erwartete Gewinn, wenn π_θ ausgeführt wird. Wenn wir einen Ausdruck für $\rho(\theta)$ in geschlossener Form ableiten können, haben wir ein Standard-Optimierungsproblem, wie in *Kapitel 4* beschrieben. Wir können dem **Strategie-Gradientenvektor** $\nabla_\theta\rho(\theta)$ folgen, vorausgesetzt $\rho(\theta)$ ist differenzierbar. Ist $\rho(\theta)$ nicht in geschlossener Form verfügbar, können wir alternativ π_θ auswerten, indem wir es einfach ausführen und die akkumulierte Belohnung beobachten. Dann können wir dem empirischen Gradienten durch Hill-Climbing folgen – d.h. durch Auswertung der Strategieänderungen für kleine Inkrementschritte jedes Parameterwertes. Unter den üblichen Vorbehalten konvergiert dieser Prozess im Strategieraum zu einem lokalen Optimum.

Wenn die Umgebung (oder die Strategie) stochastisch ist, wird das Ganze schwieriger. Angenommen, wir versuchen es mit dem Hill-Climbing, wofür ein Vergleich von $\rho(\theta)$ und $\rho(\theta + \Delta\theta)$ für einige kleine $\Delta\theta$ durchgeführt werden muss. Das Problem dabei ist, dass die Gesamtbelohnungen für jeden Versuch wesentlich variieren können, weshalb Schätzungen des Strategiewertes nach einer kleinen Anzahl an Versuchen relativ unzuverlässig sind; der Versuch, zwei solche Schätzungen zu vergleichen, ist noch unzuverlässiger. Eine Lösung ist, einfach sehr viele Versuche auszuführen, die Stichprobenvarianz zu messen und damit zu entscheiden, ob genügend Versuche durchgeführt wurden, um zuverlässig die Richtung der Verbesserung von $\rho(\theta)$ zu erkennen. Leider ist dies für sehr viele reale Probleme nicht praktisch, weil die einzelnen Versuche sehr teuer, zeitaufwändig und vielleicht sogar gefährlich sein können.

Für den Fall einer stochastischen Strategie $\pi_\theta(s, a)$ ist es möglich, eine nicht fehlerbehaftete Schätzung des Gradienten an der Stelle θ zu erhalten, $\nabla_\theta\rho(\theta)$, nämlich direkt aus den Ergebnissen der Versuche, die an der Stelle θ ausgeführt wurden. Der Einfachheit halber leiten wir diese Schätzung für den einfachen Fall einer nicht sequenziellen Umgebung ab, wobei wir die Belohnung unmittelbar nach dem Handeln im Ausgangszustand s_0 erhalten. In diesem Fall ist der Strategiewert einfach der erwartete Wert der Belohnung, und wir haben:

$$\nabla_\theta\rho(\theta) = \nabla_\theta \sum_a \pi_0(s_0, a)R(a) = \sum_a (\nabla_\theta\pi_\theta(s_0, a))R(a).$$

Jetzt führen wir einen einfachen Trick aus, sodass diese Summation durch Stichproben angenähert werden kann, die aus der durch $\pi_\theta(s_0, a)$ definierten Wahrscheinlichkeitsverteilung erzeugt wurden. Angenommen, wir haben insgesamt N Versuche und die Aktion im j-tenVersuch ist a_j. Dann gilt:

$$\nabla_\theta\rho(\theta) = \sum_a \pi_0(s_0, a) \cdot \frac{(\nabla_\theta\pi_\theta(s_0, a))R(a)}{\pi_\theta(s_0, a)} R(a) \approx \frac{1}{N}\sum_{j=1}^{N} \frac{(\nabla_\theta\pi_\theta(s_0, a_j))R(a_j)}{\pi_\theta(s_0, a_j)}.$$

Damit wird der wahre Gradient des Strategiewertes angenähert durch eine Summe von Termen, die den Gradienten der Aktionsauswahl-Wahrscheinlichkeit in jedem Versuch beinhaltet. Für den sequentiellen Fall kann dies für jeden besuchten Zustand s zu

$$\nabla_\theta\rho(\theta) \approx \frac{1}{N}\sum_{j=1}^{N} \frac{(\nabla_\theta\pi_\theta(s, a_j))R_j(s)}{\pi_\theta(s, a_j)}$$

verallgemeinert werden. Dabei wird a_j in s im j-ten Versuch ausgeführt und $R_j(s)$ ist die Gesamtbelohnung, die wir ab dem Zustand s im j-ten Versuch erhalten. Der resultierende Algorithmus heißt REINFORCE (Williams, 1992); er ist immer noch sehr viel effektiver als das Hill-Climbing unter Verwendung von unzähligen Versuchen für jeden Wert von θ; er ist jedoch immer noch viel langsamer als nötig.

Tipp

Betrachten Sie die folgende Aufgabenstellung: Sie haben zwei Blackjack[5]-Programme und sollen feststellen, welches der beiden besser ist. Eine Möglichkeit dafür ist, jedes Programm eine bestimmte Anzahl von Spielen gegen einen Standard-„Kartengeber" spielen zu lassen und dann zu zählen, wie oft sie gewonnen haben. Das Problem dabei ist, wie wir gesehen haben, dass die Gewinne der einzelnen Programme oft wesentlich variieren – abhängig davon, ob sie gute oder schlechte Karten bekommen. Eine naheliegende Lösung wäre es, im Voraus eine bestimmte Anzahl an Kartenkonfigurationen (sogenannten *Händen*) zu erzeugen und *jedes Programm denselben Satz von Händen spielen zu lassen*. Auf diese Weise eliminieren wir den Bewertungsfehler aufgrund von Differenzen in den gegebenen Karten. Diesem als **korreliertes Sampling** bezeichneten Konzept liegt der Algorithmus zur Strategiesuche namens PEGASUS (Ng und Jordan, 2000) zugrunde. Der Algorithmus ist auf Domänen anwendbar, für die es einen Simulator gibt, sodass die „zufälligen" Ergebnisse von Aktionen wiederholt werden können. Der Algorithmus erzeugt zunächst N Folgen von Zufallszahlen und verwendet sie dann, um einen Versuch einer beliebigen Strategie auszuführen. Die Strategiesuche wird durch Bewertung jeder Kandidatenstrategie unter Verwendung *derselben* Menge an Zufallsfolgen ausgeführt, um die Aktionsergebnisse zu ermitteln. Es kann gezeigt werden, dass die Anzahl der benötigten Zufallsfolgen, mit der sichergestellt werden kann, dass der Wert *jeder* Strategie korrekt eingeschätzt wird, nur von der Komplexität des Strategieraumes abhängig ist, dagegen überhaupt nicht von der Komplexität der zugrunde liegenden Domäne.

21.6 Anwendungen des verstärkenden Lernens

Jetzt wenden wir uns Beispielen für große Anwendungen des verstärkenden Lernens zu. Dabei betrachten wir zum einen Anwendungen beim Spielen, wo das Übergangsmodell bekannt ist und das Ziel darin besteht, die Nutzenfunktion zu lernen, und zum anderen Anwendungen bei der Robotersteuerung, wo das Modell normalerweise unbekannt ist.

21.6.1 Anwendungen beim Spielen

Die erste bedeutende Anwendung verstärkenden Lernens war gleichzeitig das erste bedeutende lernende Programm – das Dame spielende Programm von Arthur Samuel (1959, 1967). Samuel setzte zuerst eine gewichtete lineare Funktion ein, um die Positionen zu bewerten, wobei er bis zu 16 Terme gleichzeitig verwendete. Er wendete eine Version von Gleichung (21.12) an, um die Gewichtungen zu aktualisieren. Es gab jedoch einige wesentliche Unterschiede zwischen seinem Programm und den heutigen Methoden. Erstens aktualisierte er die Gewichtungen anhand der Differenz zwischen dem aktuellen Zustand und dem gespeicherten Wert, der erzeugt wurde, indem der Suchbaum vorwärts durchsucht wurde. Das funktioniert wunderbar, weil dabei

5 Auch als 21 oder Siebzehn-und-vier bezeichnet, allerdings in einzelnen Regeln abweichend.

der Zustandsraum bei einer anderen Granularität betrachtet wird. Ein zweiter Unterschied war, dass das Testprogramm *keine* beobachteten Belohnungen verwendete! Die Werte von Terminalzuständen beim Spiel gegen sich selbst wurden also ignoriert. Es ist also theoretisch möglich, dass das Programm von Samuel nicht konvergiert oder auf eine Strategie konvergiert, die auf Verlust statt auf Gewinn ausgelegt ist. Er schaffte es, dieses Schicksal zu vermeiden, indem er darauf bestand, dass die Gewichtung für Materialvorteile immer positiv sein sollte. Bemerkenswerterweise war das ausreichend, um das Programm in die Bereiche des Gewichtungsraumes zu führen, die einem guten Damespiel entsprechen.

Das Programm TD-GAMMON von Gerry Tesauro (1992) demonstriert das Potenzial verstärkender Lerntechniken auf beeindruckende Weise. In früheren Arbeiten (Tesauro und Sejnowski, 1989) versuchte Tesauro, eine Darstellung mit neuronalen Netzen von $Q(s, a)$ direkt anhand von Beispielen von Zügen zu lernen, die von einem menschlichen Experten mit relativen Werten beschriftet wurden. Dieser Ansatz erwies sich als extrem mühsam für den Experten. Er führte zu einem Programm namens NEUROGAMMON, das für Computerstandards stark war, sich aber nicht mit menschlichen Experten messen konnte. Das Projekt TD-GAMMON war ein Versuch, nur aus dem Spielen mit sich selbst zu lernen. Das einzige Belohnungssignal wurde am Ende jedes Spieles übermittelt. Die Bewertungsfunktion wurde durch ein vollständig verknüpftes neuronales Netz mit einer einzigen verborgenen Schicht mit 40 Knoten dargestellt. TD-GAMMON lernte einfach durch wiederholte Anwendung von Gleichung (21.12), wesentlich besser zu spielen als NEUROGAMMON, obwohl die Eingaberepräsentation nur die beiden reinen Brettpositionen ohne berechnete Merkmale beinhaltete. Das dauerte etwa 200.000 Trainingsspiele und zwei Wochen Rechenzeit. Das scheinen zwar sehr viele Spiele zu sein, aber es handelt sich dabei nur um einen verschwindend kleinen Bruchteil des Zustandsraumes. Als der Eingaberepräsentation vorberechnete Merkmale hinzugefügt wurden, war ein Netz mit 80 verborgenen Knoten in der Lage, nach 300.000 Trainingsspielen einen Standard zu erreichen, der mit dem der drei weltweit besten menschlichen Spieler vergleichbar war. Kit Woolsey, ein extrem guter Spieler und Analytiker, sagte: „Ich bin mir zweifelsfrei sicher, dass seine Positionsbeurteilung sehr viel besser als die meine ist."

21.6.2 Anwendung auf Robotersteuerung

▶ Abbildung 21.9 zeigt den Aufbau für das bekannte **Wagen/Stange-Balanceproblem**, auch als das **umgekehrte Pendel** bezeichnet. Das Problem dabei ist, die Position x des Wagens so zu steuern, dass die Stange etwa aufrecht steht ($\theta \approx \pi/2$), wobei man innerhalb der erlaubten Spurführung bleiben muss, wie Abbildung 21.9 zeigt. Mehrere tausend Arbeiten über verstärkendes Lernen und Steuerungstheorie haben sich mit diesem scheinbar einfachen Problem beschäftigt. Das Wagen/Stange-Problem unterscheidet sich von den zuvor beschriebenen Problemen, weil die Zustandsvariablen x, θ, $\dot{x}$ und $\dot{\theta}$ stetig sind. Die Aktionen sind normalerweise diskret: Bewegungen nach links oder nach rechts nach dem System der sogenannten **Bang-Bang-Steuerung**.

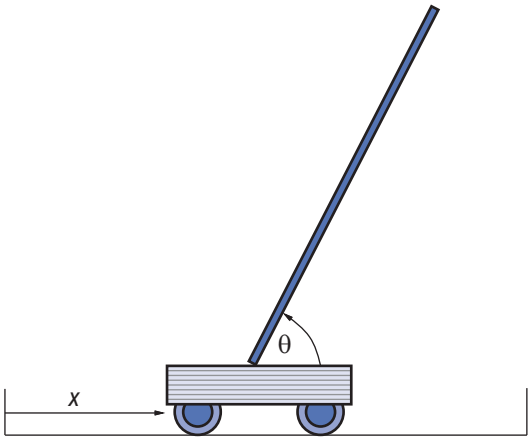


Abbildung 21.9: Einstellung für das Problem, eine lange Stange auf einem bewegten Wagen zu balancieren. Der Wagen kann von einer Steuerung, die x, θ, $\dot{x}$ und $\dot{\theta}$ beobachtet, nach links oder nach rechts gesteuert werden.

Die früheste Arbeit zum Lernen für dieses Problem stammt von Michie und Chambers (1968). Ihr BOXES-Algorithmus war in der Lage, die Stange nach nur 30 Versuchen über eine Stunde im Gleichgewicht zu halten. Anders als viele nachfolgende Systeme war BOXES darüber hinaus unter Verwendung eines echten Wagens und einer echten Stange implementiert, nicht als Simulation. Der Algorithmus diskretisierte zuerst den vierdimensionalen Zustandsraum in Rechtecke (engl. Boxes) – daher auch der Name. Anschließend führte er Versuche aus, bis die Stange umfiel oder der Wagen das Spurende erreichte. Negative Verstärkung wurde der letzten Aktion im letzten Rechteck zugeordnet und dann durch die Folge zurückgereicht. Man stellte fest, dass die Diskretisierung einige Probleme verursachte, wenn der Aufbau an einer Position initialisiert wurde, die sich von der im Training verwendeten unterschied, was darauf hinwies, dass die Verallgemeinerung nicht perfekt war. Eine verbesserte Verallgemeinerung und schnelleres Lernen erhält man mit einem Algorithmus, der den Zustandsraum gemäß den beobachteten Belohnungsvariationen *adaptiv* partitioniert, oder mit einer stetigen nichtlinearen Funktionsannäherung wie zum Beispiel einem neuronalen Netz. Heute ist die Balance eines *dreifach* umgekehrten Pendels eine gebräuchliche Übung – die über das Geschick der meisten Menschen weit hinausgeht.

Noch eindrucksvoller ist die Anwendung des verstärkenden Lernens für den Hubschrauberflug (siehe ▶ Abbildung 21.10). Für diese Aufgabe wurden im Allgemeinen eine Strategiesuche (Bagnell und Schneider, 2001) sowie der PEGASUS-Algorithmus mit einer Simulation, die auf einem gelernten Übergangsmodell basiert (Ng et al, 2004), verwendet. Weitere Details finden Sie in *Kapitel 25*.

Abbildung 21.10: Übereinandergelegte Zeitrafferaufnahmen einer Autopilot-Helikoptervorführung mit einem sehr schwierigen „Nase im Kreis"-Manöver. Der Helikopter wird von einer vom PEGASUS-Strategiesuchalgorithmus entwickelten Strategie gesteuert. Es wurde ein Simulatormodell entwickelt, indem die Effekte der verschiedenen Steuerungsreaktionen bei einem realen Helikopter beobachtet wurden. Anschließend wurde der Algorithmus über Nacht auf dem Simulatormodell ausgeführt. Eine Vielzahl von Steuerungen für unterschiedliche Manöver wurden entwickelt. In allen Fällen übertraf die Leistung die von erfahrenen Piloten unter Verwendung von Fernsteuerung. (Bild mit freundlicher Genehmigung von Andrew Ng.)

Bibliografische und historische Hinweise

Turing (1948, 1950) schlug den Ansatz des verstärkenden Lernens vor, obwohl er nicht von seiner Effektivität überzeugt war. Er schrieb dazu: „Die Verwendung von Bestrafungen und Belohnungen kann im besten Fall ein Teil des Lehrprozesses sein." Die Arbeit von Arthur Samuel (1959) war wahrscheinlich die erste erfolgreiche Forschung auf dem Gebiet des maschinellen Lernens. Obwohl diese Arbeit informell war und sehr viele Mängel aufwies, enthielt sie die meisten der modernen Ideen zum verstärkenden Lernen, einschließlich temporaler Differenz und Funktionsannäherung. Etwa zur selben Zeit trainierten die Forscher der adaptiven Kontrolltheorie (Widrow und Hoff, 1960) basierend auf einer Arbeit von Hebb (1949) ein einfaches Netz mithilfe der Delta-Regel. (Diese frühe Verknüpfung zwischen neuronalen Netzen und verstärkendem Lernen kann zu der andauernden Fehleinschätzung geführt haben, dass es sich bei dem Letztgenannten um einen Unterbereich des Erstgenannten handelt.) Die Wagen/Stange-Arbeit von Michie und Chambers (1968) kann ebenfalls als Methode des verstärkenden Lernens mit Funktionsannäherung betrachtet werden. Die Literatur aus der Psychologie zum verstärkenden Lernen ist sehr viel älter; Hilgard und Bower (1975) zeigen einen guten Überblick. Direkte Evidenz für die Arbeitsweise des verstärkenden Lernens bei Tieren wurde durch Forschungen im Suchverhalten von Bienen gefunden; es gibt eine deutliche neuronale Verbindung des Belohnungssignals in Form einer großen Neuronenzuordnung von den Sensoren für die Nektaraufnahme direkt zu dem für die Motorik verantwortlichen Nervensystem (Montague et al., 1995). Forschungen mithilfe von Einzelzellaufzeichnungen zeigen, dass das Dopaminsystem in Primatengehirnen etwas Ähnliches implementiert wie das Wertfunktionlernen (Schultz et al., 1997). Der Aufsatz zur Neurowissenschaft von Dayan und Abbott

(2001) beschreibt mögliche neuronale Implementierungen des TD-Lernens, während Dayan und Niv (2008) die neuesten Belege aus Experimenten von Neurowissenschaft und Verhalten im Überblick angeben.

Die Verknüpfung zwischen dem verstärkenden Lernen und den Markov-Entscheidungsprozessen wurde zuerst von Werbos (1977) festgestellt, aber die Entwicklung des verstärkenden Lernens in der KI stammt von Arbeiten an der Universität von Massachusetts in den frühen 1980er Jahren (Barto et al., 1981). Die Arbeit von Sutton (1988) bietet einen guten geschichtlichen Überblick. Gleichung (21.3) in diesem Kapitel ist ein Sonderfall für $\lambda = 0$ des allgemeinen TD(λ)-Algorithmus von Sutton. TD(λ) aktualisiert die Nutzenwerte aller Zustände in einer Folge, die zu einem Übergang führen, um einen Betrag, der für Zustände t Schritte in der Vergangenheit mit λ^t abfällt. TD(1) ist identisch mit der Widrow-Hoff- oder Delta-Regel. Boyan (2002) sagt aufbauend auf den Arbeiten von Bradtke und Barto (1996), dass TD(λ) und verwandte Algorithmen die Erfahrungen nicht effizient genug nutzen; im Wesentlichen sind sie Online-Regressionsalgorithmen, die sehr viel langsamer konvergieren als die Offline-Regression. Sein LSTD (Least-Squares Temporal Differencing) ist ein Online-Algorithmus, der die gleichen Ergebnisse erbringt wie die Offline-Regression. LSPI (Least-Squares Policy Iteration, Lagoudakis und Parr, 2003) kombiniert diese Idee mit dem Algorithmus zur Strategie-Iteration, woraus sich ein robuster und statistisch effizienter sowie modellfreier Algorithmus für Lernstrategien ergibt.

Die Kombination aus TD-Lernen und dem modellbasierten Erzeugen simulierter Erfahrungen wurde in der DYNA-Architektur von Sutton vorgeschlagen (Sutton, 1990). Die Idee des prioritätsabhängigen Suchens wurde unabhängig von Moore und Atkeson (1993) sowie Peng und Williams (1993) vorgeschlagen. Das Q-Lernen hat Watkins in seiner Doktorarbeit entwickelt (1989), während SARSA in einem Fachartikel von Rummery und Niranjan (1994) veröffentlicht wurde.

Banditenprobleme, die das Problem der Exploration für nicht sequentielle Entscheidungen modellieren, werden detailliert in Berry und Fristedt (1985) analysiert. Optimale Explorationsstrategien für verschiedene Einstellungen findet man unter Anwendung einer Technik namens **Gittins-Indizes** (Gittins, 1989). Eine Vielzahl von Explorationsmethoden für sequentielle Entscheidungsprobleme werden in Barto et al. (1995) diskutiert. Kearns und Singh (1998) sowie Brafman und Tennenholtz (2000) beschreiben Algorithmen, die unbekannte Umgebungen erkunden und garantiert in polynomialer Zeit auf nahezu optimale Strategien konvergieren. Bayessches verstärkendes Lernen (Dearden et al., 1998, 1999) bietet einen anderen Aspekt sowohl zur Modellunsicherheit als auch zur Exploration.

Die Funktionsannäherung im verstärkenden Lernen geht zurück auf die Arbeit von Samuel, der sowohl lineare als auch nicht lineare Bewertungsfunktionen ebenso wie Funktionsauswahlmethoden verwendet hat, um den Merkmalsraum zu reduzieren. Spätere Methoden sind unter anderem der **CMAC** (Cerebellar Model Articulation Controller) (Albus, 1975), wobei es sich im Wesentlichen um eine Summe überlappender lokaler Kernelfunktionen handelt, sowie die assoziativen neuronalen Netze von Barto et al. (1983). Neuronale Netze sind momentan die gebräuchlichste Form der Funktionsannäherung. Die bekannteste Anwendung ist TD-Gammon (Tesauro, 1992, 1995), die in diesem Kapitel bereits angesprochen wurde. Ein wichtiges Problem der auf neuronalen Netzen basierenden TD-Lernalgorithmen ist, dass sie dazu tendieren, frühere Erfahrungen zu vergessen, insbesondere in Teilen des Zustandsraumes, die gemieden werden, nachdem Kompetenz

erzielt wurde. Das kann zu katastrophalen Fehlschlägen führen, wenn solche Konstellationen erneut auftreten. Eine Funktionsannäherung auf der Grundlage **instanzbasierten Lernens** kann dieses Problem vermeiden (Ormoneit und Sen, 2002; Forbes, 2002).

Die Konvergenz verstärkender Lernalgorithmen unter Verwendung der Funktionsannäherung ist ein extrem technisches Thema. Ergebnisse für das TD-Lernen haben progressiv den Zweig der linearen Funktionsannäherungen gestärkt (Sutton, 1988; Dayan, 1992; Tsitsiklis und Van Roy, 1997), aber es gibt auch mehrere Beispiele für die Divergenz nichtlinearer Funktionen (siehe Tsitsiklis und Van Roy, 1997). Papavassiliou und Russell (1999) beschreiben eine neue Form des verstärkenden Lernens, das mit jeder beliebigen Funktionsannäherung konvergiert, vorausgesetzt, es kann eine bestmögliche Annäherung für die beobachteten Daten gefunden werden.

Strategiesuchmethoden wurden von Williams (1992) ins Spiel gebracht, der die REINFORCE-Algorithmenfamilie entwickelte. Spätere Arbeiten von Marbach und Tsitsiklis (1998), Sutton et al. (2000) sowie Baxter und Bartlett (2000) unterstützten und verallgemeinerten die Konvergenzergebnisse für die Strategiesuche. Die Methode des korrelierten Samplings für das Vergleichen verschiedener Konfigurationen eines Systems wurde formal von Kahn und Marshall (1953) beschrieben, war offenbar aber schon viel früher bekannt. Ihre Verwendung im verstärkenden Lernen geht auf Ng und Jordan (2000) zurück, wobei der zuletzt genannte Artikel auch den PEGASUS-Algorithmus eingeführt und dessen formale Eigenschaften bewiesen hat.

Wie in diesem Kapitel bereits erwähnt, ist die Leistung einer *stochastischen* Strategie eine stetige Funktion ihrer Parameter, was bei gradientenbasierten Suchmethoden hilfreich ist. Das ist nicht der einzige Vorteil: Jaakkola et al. (1995) sagen, dass stochastische Strategien besser funktionieren als deterministische Strategien in partiell beobachtbaren Umgebungen, wenn beide auf Aktionen beschränkt sind, die auf der aktuellen Wahrnehmung basieren. (Ein Grund dafür ist, dass die stochastische Strategie weniger wahrscheinlich aufgrund irgendwelcher nicht bemerkten Hindernisse „stecken bleibt“.) In *Kapitel 17* haben wir erklärt, dass optimale Strategien in partiell beobachtbaren MEPs deterministische Funktionen des *Belief State* und nicht der tatsächlichen Wahrnehmung sind; deshalb erwarten wir noch bessere Ergebnisse, wenn wir den Belief State mithilfe der **Filtermethoden** aus *Kapitel 15* verwalten. Leider ist der Belief-State-Raum hochdimensional und stetig und es wurden noch keine effektiven Algorithmen für verstärkendes Lernen mit Belief States entwickelt.

Umgebungen der realen Welt weisen ebenfalls eine enorme Komplexität im Hinblick auf die Anzahl elementarer Aktionen auf, die erforderlich sind, um einen wirklich deutlichen Gewinn zu erzielen. Ein Roboter beispielsweise, der Fußball spielt, muss möglicherweise hunderttausend einzelne Fußbewegungen machen, bevor er ein Tor schießt. Eine gebräuchliche Methode, die ursprünglich in der Tierdressur verwendet wurde, ist die sogenannte **belohnende Erziehung**. Das bedeutet, der Agent erhält zusätzlich sogenannte **Pseudobelohnungen**, wenn er „Fortschritte“ macht. Beim Fußball könnte das für einen Ballkontakt oder für einen Torschuss erfolgen. Solche Belohnungen können das Lernen wesentlich beschleunigen und sind ganz einfach zu realisieren, aber es besteht das Risiko, dass der Agent lernt, die Pseudobelohnungen zu maximieren und nicht die wahren Belohnungen; wenn er beispielsweise neben dem Ball steht und „vibriert“, dann sind das viele Ballkontakte. Ng et al. (1999) zeigen, dass der Agent dennoch die optimale Strategie lernt, vorausgesetzt, die Pseudobelohnung $F(s, a, s')$ erfüllt die Bedingung $F(s, a, s') = \gamma\Phi(s') - \Phi(s)$, wobei Φ eine beliebige

Funktion des Zustandes ist. Φ kann so aufgebaut werden, dass sie alle wünschenswerten Aspekte des Zustandes reflektiert, wie beispielsweise das Erreichen von Unterzielen oder die Distanz zu einem Zielzustand.

Die Erzeugung komplexer Verhalten kann auch durch **hierarchische verstärkende Lernmethoden** vereinfacht werden, die versuchen, Probleme auf mehreren Abstraktionsebenen zu lösen – ähnlich wie die **HTN-Planungsmethoden** aus *Kapitel 11*. Beispielsweise kann „ein Tor schießen“ in „Ballbesitz erlangen“, „zum Tor dribbeln“ und „schießen“ zerlegt werden und jede dieser Aktionen kann wiederum in motorische Lowlevel-Verhaltensweisen zerlegt werden. Die wichtigsten Ergebnisse in diesem Bereich stammen von Forestier und Varaiya (1978), die bewiesen haben, dass Lowlevel-Verhalten beliebiger Komplexität aus der Perspektive des höheren Verhaltens, das sie aufruft, wie elementare Aktionen behandelt werden können (die jedoch vielleicht unterschiedlichen Zeitaufwand bedingen). Aktuelle Ansätze (Parr und Russell (1998); Dietterich (2000); Sutton et al. (2000); Andre und Russell (2002) bauen auf diesem Ergebnis auf, um Methoden zu entwickeln, die einem Agenten ein **partielles Programm** bereitstellen, das sein Verhalten auf eine bestimmte hierarchische Struktur beschränkt. Die partielle Programmiersprache für Agentenprogramme erweitert eine normale Programmiersprache mit zusätzlichen elementaren Anweisungen für nicht spezifizierte Auswahlen, die durch Lernen ausgefüllt werden müssen. Dann wird verstärkendes Lernen angewendet, um das beste Verhalten zu lernen, das mit dem partiellen Programm konsistent ist. Die Kombination aus Funktionsannäherung, Erziehung und hierarchisch verstärkendem Lernen hat nachweislich auch große Probleme lösen können – zum Beispiel Strategien, die für 10^4 Schritte im Zustandsraum von 10^{100} Zuständen mit Verzweigungsfaktoren von 10^{30} laufen (Marthi et al., 2005). Als wichtiges Ergebnis (Dietterich, 2000) bietet die hierarchische Struktur eine natürliche *additive Zerlegung* der Gesamtnutzenfunktion in Terme, die von kleinen Teilmengen der Variablen abhängen, die den Zustandsraum definieren. Das ist etwa vergleichbar mit den Repräsentationstheoremen, die dem Bewusstsein der Bayesschen Netze zugrunde liegen (*Kapitel 14*).

Das verstärkende Lernen mit verteilten und mehreren Agenten wurde zwar in diesem Kapitel nicht angesprochen, ist aber von großem aktuellen Interesse. Im verteilten verstärkenden Lernen besteht das Ziel darin, Methoden auszuarbeiten, durch die mehrere, koordinierte Agenten lernen, eine gemeinsame Nutzenfunktion zu optimieren. Können wir beispielsweise Methoden entwickeln, bei denen separate **Subagenten** für die Navigation und die Hindernisvermeidung von Robotern in einem kombinierten Steuerungssystem zusammenwirken, das global optimal ist? Einige grundlegende Ergebnisse sind bereits in dieser Richtung erzielt worden (Guestrin et al., 2002; Russell und Zimdars, 2003). Prinzipiell lernt dabei jeder Subagent seine eigene Q-Funktion aus seinem eigenen Strom von Belohnungen. Zum Beispiel kann eine Komponente für die Roboternavigation Belohnungen für das Vorankommen in Richtung Ziel erhalten, während die Komponente zur Hindernisvermeidung mit negativen Belohnungen für jede Kollision bestraft wird. Jede globale Entscheidung maximiert die Summe der Q-Funktionen und der gesamte Prozess konvergiert gegen global optimale Lösungen.

Im Unterschied zum verteilten verstärkenden Lernen können die Agenten beim verstärkenden Lernen mit mehreren Agenten ihre Aktionen nicht koordinieren (außer durch explizite Kommunikation) und müssen nicht einmal die gleiche Nutzenfunktion verwenden. Somit ist verstärkendes Lernen mit mehreren Agenten für sequentielle spieltheoretische Probleme oder **Markov-Spiele** geeignet, wie sie *Kapitel 17* defi-

niert hat. Die konsequente Forderung nach randomisierten Strategien macht die Angelegenheit nicht wesentlich komplizierter, wie *Abschnitt 21.5* gezeigt hat. Allerdings ergeben sich Probleme aus der Tatsache, dass der Gegner seine Strategie ändert, um den Agenten zu schlagen, während der Agent noch lernt, die Strategie des Gegners zu schlagen. Die Umgebung ist somit **nichtstationär** (siehe *Abschnitt 15.1.2*). Littman (1994) wies auf diese Schwierigkeit hin, als die ersten Algorithmen mit verstärkendem Lernen für Nullsummen-Markov-Spiele eingeführt wurden. Hu und Wellman (2003) präsentieren für allgemeine Summenspiele (im Gegensatz zu Nullsummenspielen, bei denen die Summe null ist) einen Q-lernenden Algorithmus, der konvergiert, wenn das Nash-Gleichgewicht eindeutig ist; gibt es mehrere Gleichgewichte, lässt sich das Konzept der Konvergenz nicht so leicht definieren (Shoham et al., 2004).

Manchmal kann die Belohnungsfunktion nicht so einfach definiert werden. Nehmen Sie als Beispiel die Aufgabe, ein Auto zu steuern. Es gibt extreme Zustände (wie etwa einen Unfall), die zweifellos mit einer großen Strafe zu belegen sind. Doch darüber hinaus ist es schwierig, die Belohnungsfunktion genau zu spezifizieren. Allerdings ist es für einen Menschen einfach, eine Weile zu fahren und dann einem Roboter zu sagen: „Mach es genauso." Der Roboter hat dann die Aufgabe des **Apprenticeship Learning**; Lernen anhand eines Beispiels der richtig ausgeführten Aufgabe, ohne explizite Belohnungen. Ng et al. (2004) und Coates et al. (2009) zeigen, wie diese Technik funktioniert, um das Fliegen eines Hubschraubers zu lernen; siehe Abbildung 25.25 in *Abschnitt 25.6.3* als Beispiel für die Akrobatik, zu der die resultierende Strategie fähig ist. Russell (1998) beschreibt die Aufgabe des **inversen verstärkenden Lernens** – anhand eines Beispielpfades durch diesen Zustandsraum herausfinden, wie die Belohnungsfunktion aussehen muss. Dies ist nützlich als Teil des Apprenticeship Learning oder als Teil einer wissenschaftlichen Untersuchung – wir können Tiere oder Roboter verstehen, indem wir von ihren Aktionen aus rückwärts gehen, um auf die Belohnungsfunktionen zu schließen.

Dieses Kapitel hat sich mit atomaren Zuständen beschäftigt – der Agent kennt von einem Zustand lediglich die Menge der verfügbaren Aktionen und die Nutzen der Ergebniszustände (oder Zustand/Aktion-Paare). Es ist aber auch möglich, verstärkendes Lernen auf strukturierte und nicht nur auf atomare Repräsentationen anzuwenden; man spricht dann vom **relationalen verstärkenden Lernen** (Tadepalli et al., 2004).

Der Überblick von Kaelbling et al. (1996) bietet einen guten Einstiegspunkt in die Literatur. Der Text von Sutton und Barto (1998), zwei der Pioniere in diesem Bereich, konzentriert sich auf Architekturen und Algorithmen und zeigt, wie verstärkendes Lernen die Konzepte des Lernens, Planens und Handelns miteinander verknüpft. Die etwas technischere Arbeit von Bertsekas und Tsitsiklis (1996) vermittelt alle Grundlagen für die Theorie der dynamischen Programmierung und stochastischen Konvergenz. Artikel über verstärkendes Lernen erscheinen häufig in *Machine Learning* im *Journal of Machine Learning Research* sowie in den Konferenzunterlagen der *International Conferences on Machine Learning* und *Neural Information Processing Systems*.

Zusammenfassung

Dieses Kapitel hat sich mit dem Problem des verstärkenden Lernens beschäftigt: Wie kann ein Agent in einer unbekannten Umgebung Erfahrungen sammeln, wenn er nur seine eigenen Wahrnehmungen und gelegentliche Belohnungen hat. Das verstärkende Lernen kann als Mikrokosmos des gesamten KI-Problems betrachtet werden, wird jedoch in einer Vielzahl vereinfachter Konstellationen untersucht, um Fortschritte zu erleichtern. Die wichtigsten Aspekte sind:

- Der allgemeine Agentenentwurf bestimmt, welche Informationen gelernt werden müssen. Die drei wichtigsten in diesem Kapitel beschriebenen Entwürfe waren der modellbasierte Entwurf, wobei ein Modell P und eine Nutzenfunktion U verwendet werden, der modellfreie Entwurf, der eine Aktion/Nutzen-Funktion Q verwendet, sowie der Reflex-Entwurf unter Verwendung einer Strategie π.
- Nutzen können nach folgenden drei Ansätzen gelernt werden:
  1. Die **direkte Nutzenschätzung** verwendet den gesamten beobachteten Gewinn für einen bestimmten Zustand als direkte Evidenz für das Lernen seines Nutzens.
  2. Die **adaptive dynamische Programmierung** (ADP) lernt ein Modell und eine Gewinnfunktion aus Beobachtungen und verwendet dann die Wert- oder Strategie-Iteration, um die Nutzen oder eine optimale Strategie zu finden. ADP setzt eine optimale Nutzung der lokalen Bedingungen für die Nutzen von Zuständen ein, die durch die Nachbarschaftsstruktur der Umgebung vorgegeben ist.
  3. Methoden der **temporalen Differenz** (TD) aktualisieren Nutzenschätzungen, sodass sie mit denen von Nachfolgezuständen übereinstimmen. Sie können als einfache Annäherungen zum ADP-Ansatz betrachtet werden, die für den Lernprozess kein Übergangsmodell benötigen. Die Verwendung eines gelernten Modells für die Erzeugung von Pseudoerfahrungen kann jedoch zu schnellerem Lernen führen.
- Aktion/Nutzen-Funktionen oder Q-Funktionen können durch einen ADP- oder einen TD-Ansatz gelernt werden. Bei TD ist für das Q-Lernen kein Modell erforderlich, weder in der Lern- noch in der Aktionsauswahlphase. Das vereinfacht das Lernproblem, schränkt aber möglicherweise die Fähigkeit ein, in komplexen Umgebungen zu lernen, weil der Agent die Ergebnisse möglicher Aktionsfolgen nicht simulieren kann.
- Wenn der lernende Agent für die Auswahl von Aktionen während des Lernens verantwortlich ist, muss er den geschätzten Wert dieser Aktionen gegen die möglicherweise für das Lernen wichtigen neuen Informationen abwägen. Eine genaue Lösung des Explorationsproblems ist nicht möglich, es gibt aber einige einfache Heuristiken, die ausreichend leistungsfähig sind.
- In großen Zustandsräumen müssen verstärkend lernende Algorithmen eine annähernde funktionale Repräsentation verwenden, um über Zustände verallgemeinern zu können. Das TD-Signal kann direkt verwendet werden, um Parameter in den Repräsentationen zu aktualisieren, wie beispielsweise in neuronalen Netzen.
- Methoden zur Strategiesuche arbeiten direkt mit einer Repräsentation der Strategie und versuchen, sie basierend auf der beobachteten Leistung zu verbessern. Die Leistungsvarianz in einer stochastischen Domäne ist ein ernsthaftes Problem; für simulierte Domänen kann es umgangen werden, indem die Zufälligkeit im Voraus festgelegt wird.

Weil das verstärkende Lernen möglicherweise die manuelle Kodierung von Steuerungsstrategien überflüssig macht, ist es weiterhin einer der aktivsten Forschungsbereiche des Maschinenlernens. Anwendungen in der Robotik sind besonders vielversprechend; sie benötigen Methoden für die Verarbeitung *stetiger, hochdimensionaler, partiell beobachtbarer* Umgebungen, in denen ein erfolgreiches Verhalten aus Tausenden oder sogar Millionen elementarer Aktionen bestehen kann.

Übungen zu Kapitel 21

Lösungshinweise

1 Implementieren Sie einen passiven Lernagenten in einer einfachen Umgebung, wie etwa der 4×3-Welt. Für den Fall eines anfänglich unbekannten Umgebungsmodells vergleichen Sie die Lernleistung der direkten Nutzenschätzung, TD- und ADP-Algorithmen. Führen Sie den Vergleich für die optimale Strategie und für mehrere zufällige Strategien durch. Wo konvergieren die Nutzenschätzungen schneller? Was passiert, wenn die Umgebung vergrößert wird? (Probieren Sie Umgebungen mit und ohne Hürden aus.)

2 Ändern Sie den passiven ADP-Agenten so ab, dass er einen annähernden ADP-Algorithmus (wie im Text beschrieben) verwendet. Gehen Sie dazu in zwei Schritten vor:

a. Implementieren Sie eine Prioritätswarteschlange für Anpassungen an die Nutzenschätzungen. Immer wenn ein Zustand angepasst wird, werden alle seine Vorgänger ebenfalls Kandidaten für die Anpassung und sollten der Warteschlange hinzugefügt werden. Die Warteschlange wird mit dem Zustand initialisiert, von dem aus der letzte Übergang stattgefunden hat. Lassen Sie nur eine bestimmte Anzahl von Anpassungen zu.

b. Experimentieren Sie mit verschiedenen Heuristiken, um die Prioritätswarteschlange zu sortieren, und untersuchen Sie ihre Auswirkung auf Lerngeschwindigkeit und Rechenzeit.

3 Die direkte Nutzenschätzungsmethode in *Abschnitt 21.2* verwendet unterschiedliche Terminalzustände, um das Ende eines Versuches zu kennzeichnen. Wie könnte sie für Umgebungen mit verminderten Belohnungen und ohne Terminalzustände angepasst werden?

4 Schreiben Sie die Parameteraktualisierungsgleichungen für TD-Lernen mit

$$\hat{U}(x,y) = \theta_0 + \theta_1 x + \theta_2 y + \theta_3 \sqrt{(x - x_g)^2 + (y - y_g)^2}$$

5 Passen Sie die Staubsaugerwelt (*Kapitel 2*) für verstärkendes Lernen an, indem Sie Belohnungen für saubere Felder vorsehen. Machen Sie die Welt mithilfe geeigneter Sensoren beobachtbar. Experimentieren Sie nun mit verschiedenen verstärkenden Lernagenten. Ist für den Erfolg eine Funktionsannäherung erforderlich? Welche Art Annäherung ist für diese Anwendung geeignet?

6 Implementieren Sie einen explorativen verstärkend lernenden Agenten, der eine direkte Nutzenschätzung verwendet. Erstellen Sie zwei Versionen – einen Agenten mit einer tabellarischen Repräsentation und einen, der die in Gleichung (21.10) gezeigte Funktionsannäherung verwendet. Vergleichen Sie ihre Leistung in drei Umgebungen:

a. In der im Kapitel beschriebenen 4×3-Welt

b. In einer 10×10-Welt ohne Hindernisse und mit der Belohnung +1 an der Stelle (10, 10)

c. In einer 10×10-Welt ohne Hindernisse und mit der Belohnung +1 an der Stelle (5, 5)

7 Erweitern Sie die Standardspielumgebung (*Kapitel 5*), sodass ein Belohnungssignal unterstützt wird. Platzieren Sie zwei verstärkend lernende Agenten in der Umgebung (die natürlich dasselbe Agentenprogramm verwenden können) und lassen Sie sie gegeneinander spielen. Wenden Sie die verallgemeinerte TD-Aktualisierungsregel (Gleichung (21.12)) an, um die Bewertungsfunktion zu aktualisieren. Sie können mit einer einfachen linearen gewichteten Bewertungsfunktion und einem einfachen Spiel wie etwa Tic-Tac-Toe beginnen.

8 Berechnen Sie die wahre Nutzenfunktion und die beste lineare Anpassung in x und y (wie in Gleichung (21.10)) für die folgenden Umgebungen:

a. Eine 10×10-Welt mit einem einzigen Terminalzustand +1 an der Stelle (10, 10)

b. Wie (a), aber mit einem zusätzlichen Terminalzustand –1 an der Stelle (10, 1)

c. Wie (b), aber mit Hindernissen auf zehn zufällig ausgewählten Quadraten

d. Wie (b), aber mit einer Mauer von (5, 2) bis (5, 9)

e. Wie (a), aber mit dem Terminalzustand auf (5, 5)

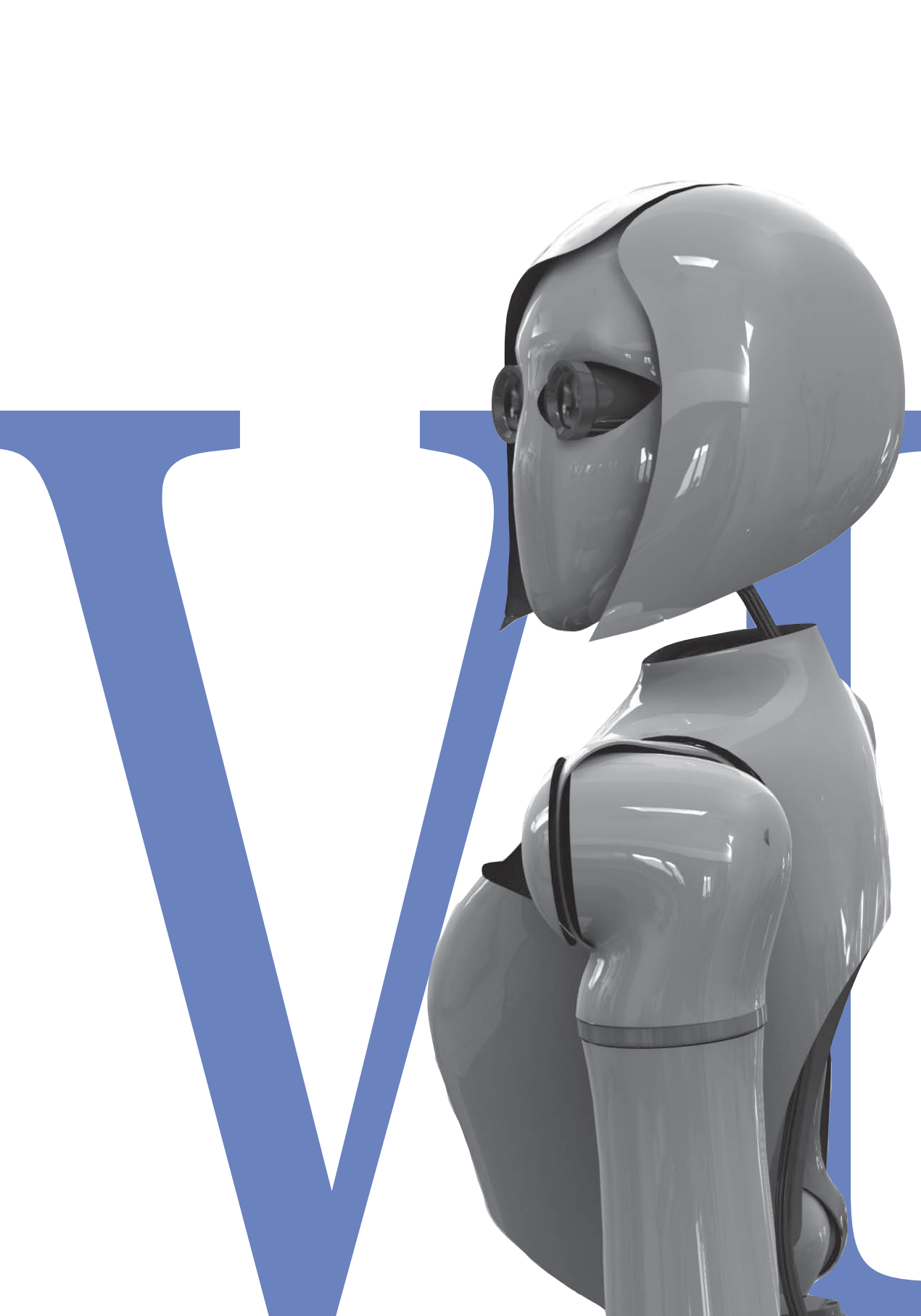
Die Aktionen sind deterministische Züge in die vier Richtungen. Vergleichen Sie für jeden Fall die Ergebnisse unter Verwendung dreidimensionaler Skizzen. Schlagen Sie für jede Umgebung zusätzliche Merkmale vor (neben x und y), die die Annäherung verbessern und die Ergebnisse zeigen.

9 Implementieren Sie die Algorithmen REINFORCE und PEGASUS und wenden Sie sie unter Verwendung einer von Ihnen gewählten Strategiefamilie auf die 4×3-Welt an. Kommentieren Sie die Ergebnisse.

10 Informieren Sie sich über die Anwendung der Konzepte verstärkenden Lernens für die Modellierung menschlichen und tierischen Verhaltens.

TEIL VI

Kommunizieren, Wahrnehmen und Handeln



Verarbeitung natürlicher Sprache

22

ÜBERBLICK

Dieses Kapitel zeigt, wie sich umfangreiches Wissen, das in natürlicher Sprache ausgedrückt ist, nutzen lässt.

Der *Homo Sapiens* hebt sich von anderen Spezies durch die Sprachfähigkeit ab. Vor rund 100.000 Jahren hat der Mensch Sprechen gelernt und vor etwa 7.000 Jahren das Schreiben. Obwohl Schimpansen, Delphine und andere Tiere Vokabulare mit Hunderten von Zeichen gezeigt haben, können nur die Menschen zuverlässig eine unbeschränkte Anzahl von qualitativ unterschiedlichen Nachrichten zu jedem beliebigen Thema mithilfe diskreter Zeichen kommunizieren.

Selbstverständlich gibt es andere Attribute, die für den Menschen einzigartig sind: Keine anderen Spezies tragen Kleidung, schaffen darstellende Kunst oder sehen täglich drei Stunden fern. Doch als Alan Turing seinen Test vorschlug (siehe *Abschnitt 1.1.1*), baute er ihn auf Sprache und nicht auf Kunst oder Fernsehen auf. Es gibt vor allem zwei Gründe, warum wir mit unseren Computeragenten natürliche Sprache verarbeiten wollen: erstens, um mit Menschen zu kommunizieren (ein Thema, das wir in *Kapitel 23* aufgreifen), und zweitens, um Informationen aus geschriebener Sprache zu erfassen (worum es in diesem Kapitel geht).

Im Web gibt es über eine Billion Informationsseiten, die fast alle in natürlicher Sprache verfasst sind. Ein Agent, der einen **Wissenserwerb** durchführen soll, muss (zumindest teilweise) die mehrdeutigen, verworrenen Sprachen verstehen, die Menschen verwenden. Wir untersuchen das Problem aus dem Blickwinkel spezifischer Informationssuchaufgaben: Textklassifizierung, Abrufen von Informationen und Extraktion von Informationen. Gemeinsam ist diesen Aufgaben die Verwendung von Sprachmodellen: Modelle, die die Wahrscheinlichkeitsverteilung von sprachlichen Ausdrücken vorhersagen.

22.1 Sprachmodelle

Formale Sprachen wie zum Beispiel die Programmiersprachen Java oder Python haben genau definierte Sprachmodelle. Eine **Sprache** lässt sich als Menge von Zeichenfolgen (Strings) definieren. So ist in der Sprache Python „`print(2 + 2)`“ ein zulässiges Programm, „`2) + (2 print`“ dagegen nicht. Da es unendlich viele zulässige Programme gibt, kann man sie nicht auflisten; stattdessen werden sie durch eine Menge von Regeln spezifiziert, die man als **Grammatik** bezeichnet. Formale Sprachen haben zudem Regeln, die die Bedeutung oder Semantik eines Programms definieren; zum Beispiel besagen die Regeln, dass die „Bedeutung“ von „2 + 2“ gleich 4 ist und die Bedeutung von „1/0“ darin besteht, dass ein Fehler signalisiert wird.

Natürliche Sprachen wie Englisch oder Spanisch lassen sich nicht als konkrete Menge von Sätzen charakterisieren. Jeder wird zustimmen, dass „Es ist traurig, nicht eingeladen zu werden“ ein deutscher Satz ist, während man bei „Traurig es ist, nicht zu werden eingeladen“ die Grammatikalität bemängeln wird. Demzufolge ist es fruchtbarer, ein natürliches Sprachmodell als Wahrscheinlichkeitsverteilung über Sätzen und nicht als konkrete Menge zu definieren. Anstatt also zu fragen, ob eine Zeichenfolge wie *Wörter* zur Menge, die die Sprache definiert, gehört oder nicht, fragen wir stattdessen nach $P(S = Wörter)$ – was die Wahrscheinlichkeit dafür ist, dass ein zufälliger Satz *Wörter* lauten würde.

Weiterhin sind natürliche Sprachen **mehrdeutig**. „He saw her duck“ kann entweder bedeuten, dass er einen Wasservogel gesehen hat, der zu ihr gehört, oder ihre Bewegung gesehen hat, um einem Gegenstand auszuweichen. Somit können wir nicht von einer

einzelnen Bedeutung für einen Satz sprechen, sondern vielmehr von einer Wahrscheinlichkeitsverteilung über möglichen Bedeutungen.

Schließlich ist der Umgang mit natürlichen Sprachen schwierig, weil sie sehr umfangreich sind und sich ständig ändern. Somit sind unsere Sprachmodelle bestenfalls eine Annäherung. Wir beginnen mit den einfachsten Annäherungen und gehen von dort aus weiter.

22.1.1 N-Gramm-Zeichenmodelle

Ein geschriebener Text besteht letztlich aus **Zeichen** – Buchstaben, Ziffern, Satzzeichen und Leerzeichen im Deutschen (und exotischeren Zeichen in manchen anderen Sprachen). Somit ist eines der einfachsten Sprachmodelle eine Wahrscheinlichkeitsverteilung über Sequenzen von Zeichen. Wie in *Kapitel 15* schreiben wir $P(c_{1:N})$ für die Wahrscheinlichkeit einer Folge von N Zeichen von c_1 bis c_N. In einer Websammlung betragen die Wahrscheinlichkeiten $P(\text{"the"}) = 0{,}027$ und $P(\text{"zgq"}) = 0{,}000000002$. Eine Folge von geschriebenen Symbolen der Länge n wird als n-Gramm bezeichnet (von der griechischen Wurzel γραμμα für „Geschriebenes“ oder „Buchstabe“), wobei die Begriffe *Monogramm* oder *Unigramm* für 1-Gramm, *Bigramm* oder *Digramm*) für 2-Gramm und *Trigramm* für 3-Gramm üblich sind. Ein Modell der Wahrscheinlichkeitsverteilung von n-buchstabigen Folgen heißt demnach ***n*-Gramm-Modell**. (Beachten Sie: Wir können n-Gramm-Modelle über Folgen von Wörtern, Silben oder anderen Einheiten erstellen, nicht nur über einzelnen Zeichen.)

Ein n-Gramm-Modell ist als **Markov-Kette** der Ordnung $n - 1$ definiert. Wie *Abschnitt 15.1.2* erläutert hat, hängt in einer Markov-Kette die Wahrscheinlichkeit des Zeichens c_i nur von den unmittelbar vorhergehenden Zeichen und nicht von irgendwelchen anderen Zeichen ab. In einem Trigramm-Modell (Markov-Kette der Ordnung 2) haben wir also

$$P(c_i \mid c_{1:i-1}) = P(c_i \mid c_{i-2:i-1}).$$

Wir können die Wahrscheinlichkeit einer Folge von Zeichen $P(c_{1:N})$ unter dem Trigramm-Modell definieren, indem wir zuerst die Kettenregel bemühen und dann die Markov-Annahme verwenden:

$$P(c_{1:N}) = \prod_{i=1}^{N} P(c_i \mid c_{1:i-1}) = \prod_{i=1}^{N} P(c_i \mid c_{i-2:i-1}).$$

Für ein Trigramm-Zeichenmodell in einer Sprache mit 100 Zeichen hat $\mathbf{P}(C_i \mid C_{i-2:i-1})$ eine Million Einträge und lässt sich genau abschätzen, indem die Zeichenfolgen in einem Textkörper von 10 Millionen Zeichen oder mehr gezählt werden. Einen Textkörper bezeichnen wir als **Korpus** (Plural *Korpora*) nach dem lateinischen Wort für *Körper*.

Was können wir mit n-Gramm-Zeichenmodellen anfangen? Gut geeignet sind sie für die **Sprachenerkennung**: für einen gegebenen Text bestimmen, in welcher natürlichen Sprache er geschrieben ist. Diese Aufgabe ist relativ leicht; selbst bei kurzen Texten wie zum Beispiel „Hello, world“ oder „Wie geht es dir“ lässt sich der erste Satz problemlos als Englisch und der zweite als Deutsch identifizieren. Computersysteme erkennen Sprachen mit einer Genauigkeit größer als 99%; gelegentlich werden eng verwandte Sprachen wie etwa Schwedisch und Norwegisch verwechselt.

Bei einem Ansatz für die Spracherkennung erstellt man zuerst ein Trigramm-Zeichenmodell jeder Kandidatensprache, $P(c_i \mid c_{i-2:i-1}, \ell)$, wobei die Variable ℓ über die Sprachen reicht. Für jedes ℓ wird das Modell erstellt, indem die Trigramme in einem Korpus dieser Sprache gezählt werden. (Es sind etwa 100.000 Zeichen jeder Sprache erforderlich.) Damit erhalten wir ein Modell von $\mathbf{P}(\mathit{Text} \mid \mathit{Sprache})$, doch da wir die wahrscheinlichste Sprache für den gegebenen Text auswählen möchten, wenden wir die Bayessche Regel gefolgt von der Markov-Annahme an:

$$\begin{aligned}\ell^* &= \operatorname*{argmax}_{\ell} P\left(\ell \mid c_{1:N}\right) \\ &= \operatorname*{argmax}_{\ell} P(\ell) P\left(c_{1:N} \mid \ell\right) \\ &= \operatorname*{argmax}_{\ell} P(\ell) \prod_{i=1}^{N} P\left(c_i \mid c_{i-2:i-1}, \ell\right).\end{aligned}$$

Das Trigramm-Modell kann von einem Korpus gelernt werden, doch wie steht es mit der A-priori-Wahrscheinlichkeit $P(\ell)$? Wir haben möglicherweise eine Abschätzung dieser Werte; wenn wir zum Beispiel eine zufällige Webseite auswählen, wissen wir, dass Englisch die wahrscheinlichste Sprache ist und dass die Wahrscheinlichkeit für das Mazedonische kleiner als 1% ist. Die exakte Zahl, die wir für diese A-priori-Wahrscheinlichkeiten auswählen, ist nicht entscheidend, da das Trigramm-Modell normalerweise eine Sprache wählt, die mehrere Größenordnungen wahrscheinlicher als irgendeine andere ist.

Weitere Aufgaben für Zeichenmodelle sind unter anderem Rechtschreibkorrektur, Genrezuordnung und Erkennung benannter Entitäten. Bei der Genreklassifizierung geht es um die Entscheidung, ob ein Text eine Nachricht, ein Rechtsdokument, ein wissenschaftlicher Artikel etc. ist. Auch wenn viele Merkmale bei dieser Klassifizierung helfen, sind Zähler von Interpunktionszeichen und anderen n-Gramm-Zeichenmerkmalen bereits recht erfolgreich (Kessler et al., 1997). Bei der Erkennung benannter Entitäten besteht die Aufgabe darin, Namen von Dingen in einem Dokument zu finden und zu entscheiden, zu welcher Klasse sie gehören. Zum Beispiel sollten wir im Text „Mr. Sopersteen wurde Aciphex verschrieben“ erkennen, dass „Mr. Sopersteen“ der Name einer Person und „Aciphex“ der Name eines Medikaments ist. Modelle auf Zeichenebene eignen sich gut für diese Aufgabe, weil sie die Zeichenfolge „ex␣“ („ex“ gefolgt von einem Leerzeichen) einem Medikamentennamen und „steen␣“ einem Personennamen zuordnen und dabei Wörter identifizieren können, die sie vorher noch nie gesehen haben.

22.1.2 N-Gramm-Modelle glätten

Kompliziert bei n-Gramm-Modellen ist vor allem, dass der Trainingskorpus nur eine Schätzung der wahren Wahrscheinlichkeitsverteilung liefert. Für häufige Zeichenfolgen wie zum Beispiel „␣th“ liefert ein beliebiger englischer Korpus eine gute Schätzung: etwa 1,5% aller Trigramme. Dagegen ist „␣ht“ recht ungewöhnlich – kein Wort im Wörterbuch beginnt mit ht. In einem Trainingskorpus für Standardenglisch hat diese Sequenz höchstwahrscheinlich einen Zählerwert von 0. Heißt das, dass wir P("␣ht") = 0 zuweisen sollten? Würden wir das tun, hätte der Text „Das Programm löst eine http-Anforderung aus“ eine sprachliche Wahrscheinlichkeit von 0, was falsch scheint. Wir haben ein Problem mit der Verallgemeinerung: Unsere Sprachmodelle sollen auch bisher

unbekannte Texte gut verallgemeinern können. Nur weil wir noch nie vorher „␣http“ gesehen haben, sollte unser Modell nicht behaupten, dass dieser Fall unmöglich ist. Wir passen also unser Sprachmodell an und weisen Sequenzen mit einem Zählerwert von null im Trainingskorpus eine kleine Wahrscheinlichkeit größer null zu (und passen die anderen Zähler leicht nach unten an, damit die Gesamtwahrscheinlichkeit weiterhin 1 ist). Die Anpassung der Wahrscheinlichkeit von Zählern mit niedriger Häufigkeit wird als **Glättung** bezeichnet.

Die einfachste Art der Glättung wurde von Pierre-Simon Laplace im 18. Jahrhundert vorgeschlagen: Wenn eine zufällige boolesche Variable X in allen n Beobachtungen bisher *false* war und weitere Informationen fehlen, sollte die Schätzung für $P(X = true)$ gleich $1/(n + 2)$ sein. Er nimmt also an, dass bei zwei weiteren Versuchen einer *true* und einer *false* sein könnte. Die Laplace-Glättung (auch als Add-One-Glättung bekannt) ist ein Schritt in die richtige Richtung, schneidet leistungsmäßig aber schlecht ab. Ein besserer Ansatz ist ein **Back-Off**-Modell, in dem wir zunächst n-Gramm-Zähler abschätzen, aber für jede Sequenz, die einen niedrigen Zählerwert (oder null) hat, gehen wir zu $(n - 1)$-Grammen zurück. **Glättung mit linearer Interpolation** ist ein Back-Off-Modell, das Trigramm-, Bigramm- und Monogramm-Modelle zur linearen Interpolation kombiniert. Es definiert die Wahrscheinlichkeitsschätzung als

$$\hat{P}(c_i \mid c_{i-2:i-1}) = \lambda_3 P(c_i \mid c_{i-2:i-1}) + \lambda_2 P(c_i \mid c_{i-1}) + \lambda_1 P(c_i),$$

wobei $\lambda_3 + \lambda_2 + \lambda_1 = 1$ ist. Die Parameterwerte λ_i können fest sein oder mit einem Algorithmus zur Erwartungsmaximierung trainiert werden. Es ist auch möglich, die Werte von λ_i von den Zählern abhängig zu machen: Wenn wir einen hohen Zähler von Trigrammen haben, gewichten wir sie relativ mehr; ist der Zähler nur gering, legen wir mehr Gewicht auf die Bigramm- und Monogramm-Modelle. Ein Lager von Forschern hat noch raffiniertere Glättungsmodelle entwickelt, während das andere Lager vorschlägt, einen größeren Korpus zusammenzutragen, sodass selbst einfache Glättungsmodelle gut funktionieren. Beide gelangen zum selben Ziel: die Varianz im Sprachmodell reduzieren.

Dabei ist eine Komplikation zu beachten: Der Ausdruck $P(c_i \mid c_{i-2:i-1})$ fragt nach $P(c_1 \mid c_{-1:0})$, wenn $i = 1$ ist, es gibt aber keine Zeichen vor c_1. Wir können zum Beispiel künstliche Zeichen einführen und c_0 als Leerzeichen oder spezielles Zeichen mit der Bedeutung „Textbeginn“ definieren. Oder wir können auf Markov-Modelle niedrigerer Ordnung zurückgreifen, wobei wir praktisch $c_{-1:0}$ als leere Folge definieren und somit $P(c_1 \mid c_{-1:0}) = P(c_1)$ ist.

22.1.3 Modellauswertung

Woher wissen wir bei so vielen möglichen n-Gramm-Modellen – Monogramm, Bigramm, Trigramm, interpoliertes Glätten mit verschiedenen Werten von λ etc. –, welches Modell infrage kommt? Wir können ein Modell mit Kreuzvalidierung auswerten. Dazu teilen wir den Korpus in einen Trainingskorpus und einen Validierungskorpus. Anschließend bestimmen wir die Parameter des Modells aus den Trainingsdaten. Schließlich werten wir das Modell mit dem Validierungskorpus aus.

Die Bewertung kann eine aufgabenspezifische Metrik sein, wie zum Beispiel Messen der Genauigkeit bei der Spracherkennung. Alternativ ist ein aufgabenunabhängiges Modell der Sprachqualität denkbar: Berechnen der Wahrscheinlichkeit, die dem Validierungskorpus durch das Modell zugewiesen wird; je höher die Wahrscheinlichkeit, desto bes-

ser. Diese Metrik ist unzweckmäßig, da die Wahrscheinlichkeit eines großen Korpus sehr kleine Werte annimmt und der Unterlauf bei Gleitkommazahlen zum Problem wird. Die Wahrscheinlichkeit einer Folge lässt sich auch mit einem als **Perplexität** bezeichneten Maß beschreiben, das wie folgt definiert ist:

$$Perplexität(c_{1:N}) = P(c_{1:N})^{-\frac{1}{N}}.$$

Perplexität ist praktisch der Kehrwert der Wahrscheinlichkeit, normalisiert nach Sequenzlänge. Man kann sie sich auch als gewichteten durchschnittlichen Verzweigungsfaktor eines Modells vorstellen. Angenommen, es gibt 100 Zeichen in unserer Sprache und unser Modell sagt, dass sie alle gleich wahrscheinlich sind. Dann ist die Perplexität für eine Sequenz beliebiger Länge gleich 100. Wenn einige Zeichen wahrscheinlicher als andere sind und das Modell dies widerspiegelt, hat das Modell eine Perplexität kleiner als 100.

22.1.4 N-Gramm-Wortmodelle

Wir kommen nun zu *n*-Gramm-Modellen über Wörtern statt Zeichen. Sämtliche Mechanismen gelten allerdings für Wort- und Zeichenmodelle gleichermaßen. Der Unterschied liegt vor allem darin, dass das **Vokabular** – die Menge der Symbole, aus denen der Korpus und das Modell besteht – größer ist. In den meisten Sprachen gibt es nur etwa 100 Zeichen und manchmal erstellen wir Zeichenmodelle, die noch restriktiver sind, indem wir zum Beispiel „A“ und „a“ als dasselbe Symbol behandeln oder sämtliche Satzzeichen als ein und dasselbe Symbol verarbeiten. Doch bei Wortmodellen haben wir mindestens Zehntausende Symbole und manchmal sogar Millionen. Der weite Bereich ergibt sich daraus, dass nicht klar ist, was ein Wort ausmacht. Im Englischen ist eine Folge von Buchstaben, die von Leerzeichen umgeben ist, ein Wort, doch in manchen Sprachen wie zum Beispiel Chinesisch werden Wörter nicht durch Leerzeichen getrennt. Und selbst im Englischen sind viele Entscheidungen zu treffen, um eine klare Richtlinie für Wortgrenzen zu haben: Wie viele Wörter enthält „ne'er-do-well“? Oder wie verhält es sich mit „(Tel:1-800-960-5660x123)“?

Wort-*n*-Gramm-Modelle müssen auch mit **vokabularfremden** Wörtern klarkommen. Bei Zeichenmodellen brauchen wir uns nicht darum zu kümmern, ob jemand einen neuen Buchstaben des Alphabets erfindet.[1] Doch bei Wortmodellen ist immer mit einem neuen Wort zu rechnen, das bisher noch nicht im Trainingskorpus aufgetaucht ist, sodass wir dieses explizit in unserem Sprachmodell berücksichtigen müssen. Dazu kann man einfach ein neues Wort in das Vokabular einfügen: `<UNK>`, was für unbekanntes (unknown) Wort steht. Durch diesen Trick können wir *n*-Gramm-Zähler abschätzen: Man geht den Trainingskorpus durch und wenn das erste Mal ein einzelnes Wort erscheint, das bisher unbekannt ist, ersetzt man es durch das Symbol `<UNK>`. Alle darauffolgenden Vorkommen des Wortes bleiben unverändert. Dann berechnet man die *n*-Gramm-Zähler für den Korpus wie üblich, wobei `<UNK>` genau wie jedes andere Wort behandelt wird. Wenn dann ein unbekanntes Wort in einer Testmenge erscheint, suchen wir seine Wahrscheinlichkeit unter `<UNK>`. Manchmal verwendet man mehrere Symbole für unbekannte Wörter, die verschiedenen Klassen zugeordnet werden. Zum Beispiel lässt sich jede beliebige Ziffernfolge durch `<NUM>` oder jede E-Mail-Adresse durch `<EMAIL>` ersetzen.

1 Mit der möglichen Ausnahme der bahnbrechenden Arbeit von T. Geisel (1955).

Um ein Gefühl dafür zu bekommen, was sich mit Wortmodellen realisieren lässt, haben wir Monogramm-, Bigramm- und Trigramm-Modelle über den Wörtern in diesem Buch erstellt und dann zufällige Stichproben der Wörter aus den Modellen entnommen. Die Ergebnisse (für die englische Ausgabe des Buches) lauten:

Unigram: logical are as are confusion a may right tries agent goal the was ...

Bigram: systems are very similar computational approach would be represented ...

Trigram: planning and scheduling are integrated the success of naive bayes model is ...

Selbst dieses kleine Beispiel sollte deutlich machen, dass das Monogramm-Modell eine schlechte Annäherung sowohl an die englische Sprache an sich als auch den Inhalt eines KI-Lehrbuches ist und die Bigramm- und Trigramm-Modelle wesentlich besser geeignet sind. Die Modelle stimmen mit dieser Abschätzung überein: Die Perplexität war 891 für das Monogramm-Modell, 142 für das Bigramm-Modell und 91 für das Trigramm-Modell.

Nachdem nun die Grundlagen der *n*-Gramm-Modelle – sowohl zeichen- als auch wortbasiert –aufgestellt sind, können wir uns einigen sprachlichen Aufgaben zuwenden.

22.2 Textklassifizierung

Wir befassen uns nun ausführlich mit der **Textklassifizierung**, auch als **Kategorisierung** bezeichnet: Für einen gegebenen Text einer bestimmten Art ist zu entscheiden, zu welcher Klasse einer vordefinierten Menge von Klassen er gehört. Spracherkennung und Genreklassifizierung sind Beispiele der Textklassifizierung, wie auch die Sentimentanalyse (Stimmungserkennung, d.h. Klassifizierung eines Filmes oder einer Produktbesprechung als positiv oder negativ) und die Spam-Erkennung (Klassifizierung einer E-Mail-Nachricht als Spam oder Nicht-Spam). Für den umständlichen Ausdruck „Nicht-Spam“ haben Forscher den Begriff **Ham** als Gegenteil von Spam geprägt. Spam-Erkennung können wir als Problem im überwachten Lernen behandeln. Eine Trainingsmenge ist unmittelbar verfügbar: Die positiven (Spam-)Beispiele stehen in meinem Spam-Ordner, die negativen (Ham-)Beispiele in meinem Posteingang. Hier ein Auszug:

Spam: Wholesale Fashion Watches -57% today. Designer watches for cheap ...

Spam: You can buy ViagraFr$1.85 All Medications at unbeatable prices! ...

Spam: WE CAN TREAT ANYTHING YOU SUFFER FROM JUST TRUST US ...

Spam: Sta.rt earn*ing the salary yo,u d-eserve by o'btaining the prope,r crede'ntials!

Ham: The practical significance of hypertree width in identifying more ...

Ham: Abstract: We will motivate the problem of social identity clustering: ...

Ham: Good to see you my friend. Hey Peter, It was good to hear from you. ...

Ham: PDS implies convexity of the resulting optimization problem (Kernel Ridge ...

Von diesem Auszug ausgehend können wir uns eine Vorstellung davon machen, was gute Merkmale sind, um sie in das Modell für überwachtes Lernen einzubinden. Wort-*n*-Gramme wie zum Beispiel „for cheap“ und „You can buy“ scheinen Indikatoren für Spam zu sein (obwohl sie auch in Ham eine Wahrscheinlichkeit größer null haben).

Merkmale auf Zeichenebene scheinen ebenfalls wichtig zu sein: Spam ist eher durchgängig in Großbuchstaben geschrieben und weist Satzzeichen mitten in Wörtern auf. Offenbar haben die Spammer gedacht, dass das Wort-Bigramm „you deserve“ zu sehr auf Spam hinweisen würde, und deshalb stattdessen „yo,u d-eserve“ geschrieben. Ein Zeichenmodell sollte dies erkennen. Wir könnten entweder ein vollständiges Zeichen-*n*-Gramm-Modell von Spam und Ham erstellen oder Merkmale wie zum Beispiel „Anzahl der Satzzeichen, die in Wörtern eingebettet sind“ manuell festlegen.

Damit haben wir zwei komplementäre Möglichkeiten, über Klassifizierung zu sprechen. Im Ansatz der Sprachmodellierung definieren wir ein *n*-Gramm-Sprachmodell für $\mathbf{P}(Nachricht \mid spam)$, indem wir auf den Spam-Ordner trainieren, und ein Modell für $\mathbf{P}(Nachricht \mid ham)$, indem wir auf den Posteingang trainieren. Dann können wir eine neue Nachricht mit einer Anwendung der Bayesschen Regel klassifizieren:

$$\underset{c \in \{spam,ham\}}{\operatorname{argmax}} P(c \mid nachricht) = \underset{c \in \{spam,ham\}}{\operatorname{argmax}} P(nachricht \mid c) P(c).$$

Hierbei wird $P(c)$ lediglich durch Zählen der Gesamtanzahl von Spam- und Ham-Nachrichten abgeschätzt. Dieser Ansatz funktioniert gut für Spam-Erkennung, genau wie er für Spracherkennung funktioniert hat.

Beim maschinellen Lernen stellen wird die Nachricht als Menge von Merkmal/Wert-Paaren dar und wenden einen Klassifizierungsalgorithmus h auf den Merkmalsvektor $\mathbf{X}$ an. Wir können die Ansätze für Sprachmodellierung und Maschinenlernen kompatibel machen, wenn wir die *n*-Gramme als Merkmale auffassen. Dies lässt sich am einfachsten mit einem Monogramm-Modell zeigen. Die Merkmale sind die Wörter im Vokabular: „a“, „aardvark“, ..., und die Werte sind die Anzahlen, wie oft jedes Wort in der Nachricht erscheint. Dadurch wird der Merkmalsvektor groß und dünn besetzt. Umfasst das Sprachmodell 100.000 Wörter, ist der Merkmalsvektor 100.000 lang, doch für eine kurze E-Mail-Nachricht haben fast alle Merkmale den Zählerwert null. Diese Monogramm-Darstellung ist das sogenannte **Bag of Words**-Modell („Sack voller Wörter“). Bei diesem Modell steckt man praktisch die Wörter des Trainingskorpus in einen Sack und wählt dann jeweils ein Wort aus. Das Konzept der Wortreihenfolge ist verloren; ein Monogramm-Modell liefert für jede Permutation eines Textes dieselbe Wahrscheinlichkeit. *N*-Gramm-Modelle höherer Ordnung behalten ein gewisses lokales Konzept der Wortreihenfolge bei.

Mit Bigrammen und Trigrammen nimmt die Anzahl der Merkmale quadratisch bzw. mit der dritten Potenz zu und wir können andere Nicht-*n*-Gramm-Merkmale hinzufügen: die Uhrzeit, zu der die Nachricht gesendet wurde, ob eine URL oder Bild Bestandteil der Nachricht ist, eine ID-Nummer für den Absender der Nachricht, die Anzahl bisheriger Spam- und Ham-Nachrichten des Absenders usw. Die Auswahl der Merkmale ist der wichtigste Teil beim Erstellen eines guten Spam-Detektors – noch wichtiger als die Auswahl eines Algorithmus für die Verarbeitung der Merkmale. Zum Teil hängt dies damit zusammen, dass es umfangreiche Trainingsdaten gibt. Wenn wir also ein Merkmal vorschlagen, lässt sich anhand der Daten genau ermitteln, ob es gut ist oder nicht. Merkmale müssen nicht ständig aktualisiert werden, da die Spam-Erkennung eine **konträre Aufgabe** ist; die Spammer modifizieren ihre Spam als Reaktion auf Änderungen des Spam-Detektors.

Es kann aufwändig sein, Algorithmen für einen sehr großen Merkmalsvektor auszuführen, sodass oftmals durch eine **Merkmalsauswahl** nur die Merkmale beibehalten werden, die am besten zwischen Spam und Ham unterscheiden. Zum Beispiel kommt das Bigramm „of the“ im Englischen häufig vor und kann gleich häufig in Spam und Ham auftauchen, sodass es nicht sinnvoll ist, es zu zählen. Oftmals genügen die etwa hundert häufigsten Features, um gut zwischen Klassen zu unterscheiden.

Nachdem wir eine Merkmalsmenge ausgewählt haben, können wir eine beliebige der bisher vorgestellten Techniken für überwachtes Lernen anwenden; zu den bevorzugten Techniken für die Textkategorisierung gehören k-nächste-Nachbarn, Support-Vector-Maschinen, Entscheidungsbäume, naive Bayes-Verfahren und logistische Regression. Alle genannten Verfahren wurden auf die Spam-Erkennung angewendet, gewöhnlich mit einer Genauigkeit im Bereich von 98% bis 99%. Mit einer sorgfältig konzipierten Merkmalsmenge kann die Genauigkeit durchaus 99,9% überschreiten.

22.2.1 Klassifizierung durch Datenkomprimierung

Klassifizierungsaufgaben spielen auch bei der **Datenkomprimierung** eine Rolle. Ein verlustloser Komprimierungsalgorithmus übernimmt eine Folge von Symbolen, erkennt wiederholte Muster in ihr und erzeugt eine Beschreibung der Sequenz, die kompakter als das Original ist. Zum Beispiel lässt sich der Text „0.142857142857142857“ zu „0.[142857]*3“ komprimieren. Komprimierungsalgorithmen erstellen Wörterbücher von Textteilsequenzen und verweisen dann auf Einträge im Wörterbuch. Das eben angegebene Beispiel hätte mit „142857“ nur einen Wörterbucheintrag.

Letztlich erstellen Komprimierungsalgorithmen ein Sprachmodell. Insbesondere modelliert der LZW-Algorithmus direkt eine Wahrscheinlichkeitsverteilung maximaler Entropie. Um die Klassifizierung durch Komprimierung durchzuführen, bringen wir zuerst alle Spam-Trainingsnachrichten zusammen und komprimieren sie als Einheit. Das Gleiche tun wir für die Ham-Nachrichten. Wenn dann eine neue Nachricht zu klassifizieren ist, fügen wir sie an die Spam-Nachrichten an und komprimieren das Ergebnis. Außerdem fügen wir sie an die Ham-Nachrichten an und komprimieren diese. Die vorherzusagende Klasse ist diejenige, die besser komprimiert – d.h. weniger zusätzliche Bytes für die neue Nachricht hinzufügt. Das Prinzip besteht dabei darin, dass eine Spam-Nachricht eher Wörterbucheinträge mit anderen Spam-Nachrichten gemeinsam hat und sich somit besser komprimieren lässt, wenn sie an eine Sammlung angefügt wird, die bereits das Spam-Wörterbuch enthält.

Experimente mit komprimierungsbasierter Klassifizierung auf der Grundlage von Standardkorpora für Textklassifizierung – der 20-Newsgroup-Datenmenge, den Reuters-10-Korpora, den Industry Sector-Korpora – weisen darauf hin, dass zwar die Ausführung von Standardkomprimierungsalgorithmen wie gzip, RAR und LZW recht langsam sein kann, ihre Genauigkeit jedoch vergleichbar mit herkömmlichen Klassifizierungsalgorithmen ist. Dies allein ist schon interessant und zeigt auch deutlich, dass Algorithmen realisierbar sind, die Zeichen-*n*-Gramme direkt ohne Vorverarbeitung von Text oder Merkmalsauswahl verwenden: Sie scheinen einige reale Muster zu erfassen.

22.3 Informationsabruf

Unter **Informationsabruf** (engl. **Information Retrieval**, **IR**) versteht man die Aufgabe, Dokumente zu finden, die für den Informationsbedarf eines Benutzers relevant sind. Die bekanntesten Beispiele für IR-Systeme sind Suchmaschinen des World Wide Web. Ein Webbenutzer kann eine Abfrage wie zum Beispiel [KI Buch] in eine Suchmaschine eingeben und erhält daraufhin eine Liste relevanter Seiten.[2] Dieser Abschnitt zeigt, wie derartige Systeme aufgebaut sind. Ein IR-System lässt sich durch die folgenden Komponenten charakterisieren:

1. **Ein Korpus von Dokumenten:** Jedes System muss entscheiden, was es als Dokument behandeln will: einen Absatz, eine Seite oder einen mehrseitigen Text.
2. **Abfragen, die in einer Abfragesprache formuliert sind:** Eine Abfrage spezifiziert, was der Benutzer wissen will. Die Abfragesprache kann eine einfache Liste mit Wörtern sein, wie beispielsweise [KI Buch], sie kann aber auch eine Folge von Wörtern angeben, die in der vorgegebenen Reihenfolge vorkommen müssen, wie beispielsweise in ["KI Buch"]; sie kann boolesche Operatoren enthalten, wie beispielsweise in [KI *AND* Buch], und sie kann nicht-boolesche Operatoren enthalten, wie beispielsweise [KI *NEAR* Buch] oder [KI Buch *SITE*:www.aaai.org].
3. **Eine Ergebnismenge:** Dies ist die Untermenge der Dokumente, die das IR-System als **relevant** für die Abfrage erachtet. Mit *relevant* meinen wir, dass sie von wahrscheinlichem Nutzen für die Person sind, die die Abfrage gestellt hat, und zwar für den konkreten Informationsbedarf, der in der Abfrage ausgedrückt wird.
4. **Eine Darstellung der Ergebnismenge:** Dies kann eine einfache Rangliste von Dokumenten sein, aber auch eine rotierende, auf einen dreidimensionalen Raum projizierte Farbabbildung der Ergebnismenge, die als zweidimensionale Anzeige wiedergegeben wird.

Die ersten IR-Systeme arbeiteten mit einem **booleschen Schlüsselwortmodell**. Jedes Wort in der Dokumentsammlung wird als boolesches Merkmal behandelt, das für ein Dokument *true* ist, wenn das Wort im Dokument erscheint, und sonst *false* ist. So ist das Merkmal „Abruf“ für das aktuelle Kapitel *true*, für *Kapitel 15* aber *false*. Die Abfragesprache ist die Sprache boolescher Ausdrücke über Merkmalen. Ein Dokument ist nur dann relevant, wenn der Ausdruck zu *true* ausgewertet wird. Beispielsweise ist die Abfrage [Information *AND* Abruf] für das aktuelle Kapitel *true* und für *Kapitel 15 false*.

Dieses Modell hat den Vorteil, dass es einfach zu erklären und zu implementieren ist. Es hat jedoch auch einige Nachteile. Erstens ist der Relevanzgrad eines Dokumentes ein einzelnes Bit, sodass es keine Richtlinien gibt, wie die relevanten Dokumente für die Präsentation angeordnet werden sollen. Zweitens dürften Benutzer, die nicht gerade Programmierer oder Logiker sind, kaum mit booleschen Ausdrücken vertraut sein. Wenn zum Beispiel ein Benutzer etwas über die Landwirtschaft in den Staaten Kansas *und* Nebraska wissen möchte, muss er die für ihn nicht intuitive Abfrage [Landwirtschaft (Kansas OR Nebraska)] stellen. Drittens kann es selbst für einen erfahrenen Benutzer schwierig sein, eine geeignete Abfrage zu formulieren. Angenommen, wir probieren die Abfrage [Information *AND* Abruf *AND* Modelle *AND* Optimierung] und erhalten eine leere Ergebnis-

2 Wir kennzeichnen jede Suchabfrage als [*query*] und verwenden eckige Klammern statt Anführungszeichen, um die Abfrage ["zwei Wörter"] von [zwei Wörter] unterscheiden zu können.

menge. Wir könnten [Information *OR* Abruf *OR* Modelle *OR* Optimierung] ausprobieren, doch wenn diese Abfrage zu viele Ergebnisse liefert, weiß man kaum, was als Nächstes ausprobiert werden soll.

22.3.1 IR-Bewertungsfunktionen

Die meisten IR-Systeme haben das boolesche Modell aufgegeben und verwenden Modelle, die auf Statistiken von Wortzählungen basieren. Wir beschreiben hier die **BM25-Bewertungsfunktion**, die aus dem Okapi-Projekt von Stephen Robertson und Karen Sparck Jones am City College von London stammt und in Suchmaschinen wie zum Beispiel dem Open Source-Projekt Lucene eingesetzt worden ist.

Eine Bewertungsfunktion übernimmt ein Dokument und eine Abfrage und gibt eine numerische Bewertung zurück; die relevantesten Dokumente haben die höchsten Bewertungen. In der BM25-Funktion ist die Bewertung eine linear gewichtete Kombination von Bewertungen für jedes der Wörter, aus denen sich die Abfrage zusammensetzt. Drei Faktoren beeinflussen die Gewichtung eines Abfrageterms: erstens die Häufigkeit, mit der ein Abfrageterm in einem Dokument erscheint (auch als *TF* für Term Frequency bezeichnet). Für die Abfrage [Landwirtschaft in Kansas] haben Dokumente, die „Landwirtschaft“ häufig erwähnen, höhere Bewertungen. Zweitens die inverse Dokumenthäufigkeit des Terms (*IDF* für Inverse Document Frequency). Das Wort „in“ erscheint in fast jedem Dokument, sodass es eine hohe Dokumenthäufigkeit und folglich eine geringe inverse Dokumenthäufigkeit besitzt. Demnach ist es nicht so wichtig für die Abfrage wie „Landwirtschaft“ oder „Kansas“. Drittens die Länge des Dokuments. Ein Dokument mit einer Länge von einer Million Wörtern wird wahrscheinlich alle Abfragewörter erwähnen, muss aber nicht unbedingt etwas mit der Abfrage zu tun haben. Ein kurzes Dokument, das alle Wörter erwähnt, ist ein wesentlich besserer Kandidat.

Die BM25-Funktion berücksichtigt alle drei Faktoren. Wir nehmen an, dass wir einen Index von N Dokumenten im Korpus erstellt haben, sodass wir mit $TF(q_i, d_j)$ nach der Anzahl suchen können, wie oft Wort q_i in Dokument d_j erscheint. Außerdem nehmen wir eine Tabelle $DF(q_i)$ mit Dokumentzählerwerten an, die die Anzahl der Dokumente angibt, die das Wort q_i enthalten. Für ein bestimmtes Dokument d_j und eine Abfrage, die aus den Worten $q_{1:N}$ besteht, haben wir dann:

$$BM25(d_j, q_{1:N}) = \sum_{i=1}^{N} IDF(q_i) \cdot \frac{TF(q_i, d_j) \cdot (k+1)}{TF(q_i, d_j) + k \cdot \left(1 - b + b \cdot \frac{|d_j|}{L}\right)}.$$

Hier ist $|d_j|$ die Länge des Dokuments d_j in Wörtern und L die durchschnittliche Dokumentlänge im Korpus: $L = \Sigma_i |d_i|/N$. Die beiden Parameter k und b lassen sich durch Kreuzvalidierung optimieren; typische Werte sind $k = 2{,}0$ und $b = 0{,}75$. Die inverse Dokumenthäufigkeit $IDF(q_i)$ von Wort q_i ist gegeben durch

$$IDF(q_i) = \log \frac{N - DF(q_i) + 0{,}5}{DF(q_i) + 0{,}5}.$$

Natürlich wäre es unpraktisch, die BM25-Bewertungsfunktion auf jedes Dokument im Korpus anzuwenden. Stattdessen erzeugen Systeme vorab einen **Index**, der für jedes Wort im Vokabular die Dokumente auflistet, die das Wort enthalten. Dies ist die soge-

nannte **Trefferliste** für das Wort. Für eine bestimmte Abfrage gleichen wir dann die Trefferlisten der Abfragewörter miteinander ab und bewerten nur die Dokumente in der sich ergebenden Schnittmenge.

22.3.2 IR-Systembewertung

Wie wissen wir, ob ein IR-System eine zufriedenstellende Leistung erbringt? Wir führen ein Experiment durch, wobei dem System eine Menge von Abfragen übergeben wird und die Ergebnismengen im Hinblick auf menschliche Relevanzbeurteilungen bewertet werden. Traditionell gab es zwei Maße, die für die Bewertung verwendet wurden: Recall und Genauigkeit (Precision). Wir werden sie anhand eines Beispiels erklären. Stellen Sie sich vor, ein IR-System hat eine Ergebnismenge für eine einzelne Abfrage zurückgegeben, für die wir wissen, welche Dokumente aus einem Korpus von 100 Dokumenten relevant oder nicht relevant sind. Die folgende Tabelle zeigt die Dokumentzähler in jeder Kategorie.

	In der Ergebnismenge	Nicht in der Ergebnismenge
Relevant	30	20
Nicht relevant	10	40

Genauigkeit misst den Anteil der Dokumente in der Ergebnismenge, die tatsächlich relevant sind. In unserem Beispiel ist die Genauigkeit gleich $30/(30 + 10) = 0{,}75$. Die falsche positive Rate beträgt $1 - 0{,}75 = 0{,}25$. Der **Recall** ist der Anteil aller relevanten Dokumente in der Sammlung, die sich in der Ergebnismenge befinden. In unserem Beispiel ist der Recall gleich $30/(30 + 20) = 0{,}60$. Die falsche negative Rate ist $1 - 0{,}60 = 0{,}40$. In einer sehr großen Dokumentensammlung, wie beispielsweise dem World Wide Web, ist der Recall schwierig zu berechnen, weil es keine einfache Methode gibt, jede Seite des Webs auf Relevanz zu untersuchen. Die beste Möglichkeit ist, den Recall abzuschätzen, indem wir Stichproben nehmen oder den Recall völlig ignorieren und nur die Genauigkeit beurteilen. Bei einer Web-Suchmaschine kann die Ergebnismenge Tausende von Dokumenten enthalten, sodass es sinnvoller ist, die Genauigkeit für mehrere unterschiedliche Größen wie zum Beispiel „P@10“ (Genauigkeit der Top-10-Ergebnisse) oder „P@50“ zu messen, als die Genauigkeit in der gesamten Ergebnismenge abzuschätzen.

Indem man die Größe der Ergebnismenge variiert, lässt sich ein Kompromiss zwischen Genauigkeit und Recall erreichen. Im Extremfall hat ein System, das jedes Dokument in der Dokumentensammlung als Ergebnismenge zurückgibt, einen Recall von 100%, weist jedoch eine geringe Genauigkeit auf. Alternativ könnte ein System ein einzelnes Dokument zurückgeben und einen geringen Recall haben, aber eine respektable Chance bei 100%. Beide Maße lassen sich mit der F_1-Bewertung zusammenfassen, d.h. mit einer einzelnen Zahl, die das harmonische Mittel $2PR(P + R)$ von Genauigkeit und Recall ist.

22.3.3 IR-Verbesserungen

Für das hier beschriebene System gibt es viele mögliche Verbesserungen und in der Tat aktualisieren Web-Suchmaschinen ihre Algorithmen, falls sie neue Ansätze entdecken und wenn das Web wächst und sich ändert.

Eine übliche Verfeinerung ist ein besseres Modell der Wirkung von Dokumentlänge auf Relevanz. Singhal et al. (1996) beobachteten, dass einfache Normalisierungsschemas für Dokumentlängen kürzere Dokumente zu stark bevorzugen und längere Dokumente nicht genügend berücksichtigen. Sie schlagen ein *gedrehtes* Normalisierungsschema für Dokumentlängen vor. Die Idee besteht dabei darin, als Pivotelement die Dokumentlänge zu nehmen, bei der die herkömmliche Normalisierung korrekt ist; kürzere Dokumente erfahren eine Verstärkung, längere eine Bestrafung.

Die BM25-Bewertungsfunktion verwendet ein Wortmodell, das alle Wörter als vollkommen unabhängig behandelt, doch wie wir wissen, sind manche Wörter korreliert: „Couch“ ist eng verwandt sowohl mit „Couches“ als auch „Sofa“. Viele IR-Systeme versuchen, diese Wechselbeziehungen zu berücksichtigen.

Lautet die Abfrage zum Beispiel [Couch], wäre es schade, aus der Ergebnismenge solche Dokumente auszuschließen, die „COUCH“ oder „Couches“, aber nicht „Couch“ erwähnen. Die meisten IR-Systeme wandeln die **Groß-/Kleinschreibung** automatisch von „COUCH“ in „couch“ um und verwenden einen **Stemming-Algorithmus** (Grundformenreduktion), um „Couches“ auf die Stammform „Couch“ zu reduzieren. Damit erhält man normalerweise eine kleine Steigerung im Recall (für Englisch im Bereich von 2%). Die Genauigkeit kann jedoch darunter leiden. Wendet man beispielsweise den Stemming-Algorithmus auf das englische „stocking“ an, um „stock“ zu erhalten, sinkt die Genauigkeit für Abfragen über Fußbekleidungen oder Finanzinstrumente, aber es könnte sich der Recall für Abfragen über Warehousing erhöhen. Stemming-Algorithmen, die auf Regeln basieren (indem sie zum Beispiel „-ing“ entfernen), können dieses Problem nicht vermeiden, aber Algorithmen, die auf Wörterbüchern basieren (und das „-ing“ nicht entfernen, wenn das Wort bereits im Wörterbuch aufgelistet ist), sind sehr wohl dazu in der Lage. Während die Grundformenreduktion nur eine kleine Wirkung im Englischen hat, ist sie in anderen Sprachen sehr viel wichtiger. Im Deutschen beispielsweise sieht man nicht selten Wörter wie „Lebensversicherungsgesellschaftsangestellter“. Sprachen wie etwa Finnisch, Türkisch, Inuit und Yupik haben rekursive morphologische Regeln, die im Prinzip Wörter unbegrenzter Länge erzeugen.

Der nächste Schritt besteht darin, **Synonyme** zu erkennen, wie beispielsweise „Sofa“ für „Couch“. Wie bei der Grundformenreduktion sind auch hier kleine Gewinne beim Recall möglich, die allerdings die Genauigkeit vermindern können. Ein Benutzer, der die Abfrage [Tim Couch] eingibt, möchte Ergebnisse über den Footballspieler und nicht über Sofas erhalten. Das Problem ist, dass „Sprachen absolute Synonymheit verabscheuen wie die Natur das Vakuum“ (Cruse, 1986). Wenn also zwei Wörter das Gleiche meinen, einigen sich die Sprecher der Sprache darauf, die Bedeutungen zu ändern, um die Verwechslungsgefahr zu beseitigen. Verwandte Wörter, die keine Synonyme sind, spielen ebenfalls eine Rolle beim Ranking – Begriffe wie „Leder“, „hölzern“ oder „modern“ können untermauern helfen, dass das Dokument wirklich von „Couch“ handelt. Synonyme und verwandte Wörter lassen sich in Wörterbüchern finden oder indem nach Korrelationen in Dokumenten oder in Abfragen gesucht wird – wenn wir bemerken, dass viele Benutzer, die die Abfrage [neues Sofa] stellen, die Abfrage [neue Couch] folgen lassen, können wir in Zukunft [neues Sofa] in [neues Sofa OR neue Couch] ändern.

Schließlich kann IR mithilfe von **Metadaten** verbessert werden – d.h. mit Daten außerhalb des Dokumenttextes. Beispiele hierfür sind unter anderem vom Menschen bereitgestellte Schlüsselwörter und Daten aus Veröffentlichungen. Im Web sind die Hypertext-Links zwischen Dokumenten eine entscheidende Informationsquelle.

22.3.4 Der PageRank-Algorithmus

PageRank[3] war eine der beiden ursprünglichen Ideen, die die Suche bei Google von anderen Web-Suchmaschinen abhob, als sie 1997 eingeführt wurde. (Die andere Neuerung war die Verwendung von Ankertext – dem unterstrichenen Text in einem Hyperlink – um eine Seite zu indizieren, selbst wenn sich der Ankertext auf einer *anderen* Seite als der zu indizierenden Seite befindet.) PageRank wurde erfunden, um das Problem der Willkür von *TF*-Bewertungen zu lösen: Wenn die Abfrage [IBM] lautet, wie stellen wir sicher, dass die Startseite von IBM (ibm.com) das erste Ergebnis ist, selbst wenn eine andere Seite den Begriff „IBM“ häufiger erwähnt? Die Idee dabei ist, dass ibm.com viele Links auf die eigene Seite enthält, sodass die Seite höher bewertet werden sollte: Jeder eingehende Link ist ein Votum für die Qualität der Seite, auf die verwiesen wird. Doch wenn wir nur eingehende Links zählen, könnte ein Web-Spammer ein Netz von Seiten erzeugen und sie alle auf eine von ihm gewählte Seite verweisen lassen, wodurch sich die Bewertung dieser Seite erhöht. Demzufolge gewichtet der PageRank-Algorithmus Links von hochqualitativen Sites stärker. Doch was ist eine hochqualitative Site? Eine Site, auf die durch andere hochqualitative Sites verwiesen wird. Die Definition ist rekursiv, doch wir werden sehen, dass die Rekursion ordnungsgemäß terminiert. Der PageRank für eine Seite p ist wie folgt definiert:

$$PR(p) = \frac{1-d}{N} + d\sum_i \frac{PR(in_i)}{C(in_i)}.$$

Hier ist $PR(p)$ der PageRang der Seite p. N gibt die Gesamtanzahl der Seiten im Korpus an, in_i sind die Seiten, die auf p verweisen und $C(in_i)$ ist der Zähler der Gesamtanzahl von ausgehenden Links auf Seite in_i. Die Konstante d ist ein Dämpfungsfaktor. Man kann sich das Ganze am **Zufallssurfer-Modell** verdeutlichen: Stellen Sie sich einen Websurfer vor, der von einer beliebigen zufälligen Seite aus beginnt, das Web zu erkunden. Mit der Wahrscheinlichkeit d (wobei wir $d = 0{,}85$ annehmen) klickt der Surfer auf einen der Links auf der Seite (unter denen er gleichförmig auswählt) und mit der Wahrscheinlichkeit $1 - d$ langweilt ihn die Seite und er beginnt von Neuem auf einer zufälligen Seite irgendwo im Web. Der PageRank der Seite p ist dann die Wahrscheinlichkeit, mit der sich der zufällige Surfer zu einem beliebigen Zeitpunkt auf Seite p befindet. Der PageRank lässt sich durch einen iterativen Vorgang berechnen: Weise zunächst alle Seiten $PR(p) = 1$ zu und durchlaufe dann den Algorithmus wiederholt, wobei die Bewertungen aktualisiert werden, bis sie konvergieren.

22.3.5 Der HITS-Algorithmus

Der HITS (Hyperlink-Induced Topic Search)-Algorithmus, der auch als „Hubs und Authorities“ bezeichnet wird, ist ein anderer einflussreicher Algorithmus zur Linkanalyse (siehe ▶ Abbildung 22.1). HITS unterscheidet sich von PageRank in mehreren Punkten. Erstens handelt es sich um ein abfrageabhängiges Maß: Er bewertet Seiten in Bezug auf eine Abfrage. Das heißt, er ist für jede neue Abfrage neu zu berechnen – ein Rechenaufwand, den die meisten Suchmaschinen nicht auf sich nehmen wollen. Für eine gegebene Abfrage sucht HITS eine Menge von Seiten, die für die Abfrage relevant

3 Der Name steht sowohl für Webseiten (engl. Web Pages) als auch für den Miterfinder Larry Page (Brin und Page, 1998).

sind. Dazu bildet er die Schnittmenge von Trefferlisten für Abfragewörter und fügt dann Seiten in der Link-Nachbarschaft dieser Seiten hinzu – Seiten, die auf eine oder von einer der anderen Seiten in der ursprünglichen relevanten Menge verlinken.

```
function HITS(query) returns pages mit Zähler für Hubs und Authorities

  pages ← EXPAND-PAGES(RELEVANT-PAGES(query))
  for each p in pages do
    p.AUTHORITY ← 1
    p.HUB ← 1
  repeat until Konvergenz do
    for each p in pages do
      p.AUTHORITY ← Σ_i INLINK_i(p).HUB
      p.HUB ← Σ_i OUTLINK_i(p).AUTHORITY
    NORMALIZE(pages)
  return pages
```

Abbildung 22.1: Der Algorithmus HITS zum Berechnen von Hubs und Authorities in Bezug auf eine Abfrage. RELEVANT-PAGES ruft die Seiten ab, die der Abfrage entsprechen, und EXPAND-PAGES fügt jede Seite hinzu, die von oder zu einer der relevanten Seiten verlinkt wird. NORMALIZE teilt die Bewertung der Seite durch die Quadratsumme der Bewertungen aller Seiten (getrennt sowohl nach Authority- als auch Hub-Bewertungen).

Jede Seite in dieser Menge gilt als **Authority** für die Abfrage, und zwar bis zu dem Grad, bis zu dem andere Seiten in der relevanten Menge auf sie verweisen. Eine Seite gilt als **Hub** in dem Maße, dass sie auf andere maßgebliche Seiten in der relevanten Menge verweist. Genau wie mit PageRank möchten wir nicht einfach die Anzahl der Links zählen, sondern mehr Wert auf die hochqualitativen Hubs und Authorities legen. Somit durchlaufen wir genau wie bei PageRank einen Prozess, der die Authority-Bewertung einer Seite als Summe der Hub-Bewertungen der auf sie verweisenden Seiten aktualisiert, und bei dem die Hub-Bewertung die Summe der Authority-Bewertungen der auf ihn verweisenden Seiten darstellt. Wenn wir dann die Bewertungen normalisieren und k-mal wiederholen, konvergiert der Prozess.

Sowohl PageRank als auch HITS haben bei der Entwicklung unseres Verständnisses für den Informationsabruf im Web wichtige Rollen gespielt. Diese Algorithmen und deren Erweiterungen bewerten täglich Milliarden von Abfragen, während Suchmaschinen immer bessere Wege entwickeln, um noch feinere Signale für die Suchrelevanz zu extrahieren.

22.3.6 Beantworten von Fragen

Informationsabruf hat die Aufgabe, Dokumente zu finden, die für eine Abfrage relevant sind, wobei die Abfrage eine Frage oder lediglich ein Themenbereich oder ein Konzept sein kann. Das **Beantworten von Fragen** (engl. **Question Answering**, **QA**) ist eine etwas andere Aufgabe, in der die Abfrage tatsächlich eine Frage ist und die Antwort keine nach Rangfolge sortierte Liste von Dokumenten, sondern eine kurze Erwiderung ist – ein Satz oder lediglich ein Ausdruck. Systeme zur Beantwortung von Fragen gibt es in der Computerlinguistik (oder Natural Language Processing, NLP) seit den 1960er Jahren, doch haben derartige Systeme erst seit 2001 den Web-Informationsabruf genutzt, um ihre Abdeckungsbreite drastisch zu erhöhen.

Das System ASKMSR (Banko et al., 2002) ist ein typisches webbasiertes Fragenbeantwortungssystem. Es basiert auf der Intuition, dass die meisten Fragen viele Male im Web

beantwortet werden, sodass man sich die Fragenbeantwortung als Problem in Genauigkeit und nicht in Recall vorstellen kann. Wir brauchen uns nicht mit allen unterschiedlichen Arten zu befassen, wie sich eine Antwort formulieren lässt – wir müssen nur eine von ihnen finden. Betrachten Sie zum Beispiel die Abfrage [Who killed Abraham Lincoln?]. Nehmen Sie an, ein System hätte diese Frage zu beantworten und könnte nur auf eine einzige Enzyklopädie zugreifen, deren Eintrag zu Lincoln sagt:

> *John Wilkes Booth altered history with a bullet. He will forever be known as the man who ended Abraham Lincoln's life.*

Um diese Passage für die Beantwortung der Frage zu verwenden, müsste das System wissen, dass ein Leben durch Töten beendet werden kann und dass sich „He" auf Booth bezieht. Daneben müsste es mehrere andere linguistische und semantische Fakten kennen.

AskMSR versucht sich nicht an derartigen Raffinessen – der Algorithmus kennt weder Pronomenverweise noch etwas über Töten oder irgendein anderes Verb. Er beherrscht 15 verschiedene Fragestellungen und wie sie sich als Abfragen für eine Suchmaschine formulieren lassen. So weiß er, dass [Who killed Abraham Lincoln] als die Abfrage [* killed Abraham Lincoln] und als [Abraham Lincoln was killed by *] neu geschrieben werden kann. Der Algorithmus gibt diese neu formulierten Abfragen aus und analysiert die Ergebnisse, die zurückkommen – nicht die vollständigen Webseiten, sondern lediglich die kurzen Textzusammenfassungen, die neben den Abfragetermen erscheinen. Die Ergebnisse werden in 1-, 2- und 3-Gramme unterteilt und in Bezug auf Häufigkeit in den Ergebnismengen und Gewichtung abgeglichen: Ein *n*-Gramm, das von einer sehr spezifischen Abfrageformulierung (wie zum Beispiel der Abfrage ["Abraham Lincoln was killed by *"] mit genauer Übereinstimmung) zurückkommt, würde mehr Gewicht als von einer allgemeinen Abfrageformulierung wie zum Beispiel [Abraham OR Lincoln OR killed] erhalten. Wir erwarten, dass sich „John Wilkes Booth" unter den abgerufenen *n*-Grammen mit den höchsten Bewertungen befindet, doch trifft das auch auf „Abraham Lincoln" und „the assassination of" sowie „Ford's Theatre" zu.

Nachdem die *n*-Gramme gespeichert sind, werden sie nach dem erwarteten Typ gefiltert. Wenn die ursprüngliche Abfrage mit „who" beginnt, filtern wir nach Namen von Personen, für „how many" nach Zahlen, für „when" nach einem Datum oder einer Uhrzeit. Außerdem gibt es einen Filter, der besagt, dass die Antwort nicht Teil der Frage sein sollte. Insgesamt sollten uns diese Instrumente erlauben, „John Wilkes Booth" (und nicht „Abraham Lincoln") als Antwort mit der höchsten Bewertung zurückzugeben.

In manchen Fällen ist die Antwort nicht länger als drei Wörter; da die Komponentenantworten nur bis zu 3-Grammen gehen, müsste eine längere Antwort aus kürzeren Stücken zusammengesetzt werden. Zum Beispiel ließe sich die Antwort „John Wilkes Booth" in einem System, das nur Bigramme verwendet, aus den höchstbewerteten Teilen „John Wilkes" und „Wilkes Booth" zusammenfügen. Auf der *Text Retrieval Evaluation Conference* (TREC) wurde AskMSR als eines der Top-Systeme eingeschätzt, das sich gegenüber Mitbewerbern durch die Fähigkeit für ein weitaus komplexeres Sprachverständnis durchsetzte. AskMSR stützt sich auf die Inhaltsbreite des Webs statt auf sein eigenes Tiefenverständnis. Das System ist nicht in der Lage, komplexe Inferenzmuster wie zum Beispiel die Zuordnung von „who killed" zu „ended the life of" zu behandeln. Doch da das Web so riesig ist, kann das System es sich leisten, Passagen wie diese zu ignorieren und auf eine einfachere Passage zu warten, mit der es umgehen kann.

22.4 Informationsextraktion

Unter **Informationsextraktion** versteht man den Prozess, Wissen zu beschaffen, indem ein Text nach dem Vorkommen einer bestimmten Objektklasse und nach Beziehungen zwischen Objekten durchsucht wird. Eine typische Aufgabe ist es, Instanzen der Adressen von Webseiten mit Datenbankfeldern für Straße, Ort, Land und Postleitzahl zu extrahieren. Ein anderes Beispiel ist das Extrahieren von Stürmen aus Wetterberichten mit Feldern für Temperatur, Windgeschwindigkeit und Niederschlag. In einer begrenzten Domäne lässt sich dies mit hoher Genauigkeit realisieren. Wird die Domäne allgemeiner, sind komplexere Sprachmodelle und Lernverfahren notwendig. *Kapitel 23* zeigt, wie man komplexe Sprachmodelle der Phrasenstruktur (Substantivphrasen und Verbphrasen) in Englisch definieren kann. Doch da es bisher keine vollständigen Modelle dieser Art gibt, definieren wir für die begrenzten Ansprüche der Informationsextraktion Modelle, die das vollständige Englische Modell annähern, und konzentrieren uns lediglich auf die Teile, die für die vorliegende Aufgabe erforderlich sind. Die Modelle, die wir in diesem Abschnitt beschreiben, sind Annäherungen vergleichbar mit dem einfachen 1-KNF-Logikmodell in Abbildung 7.21, das eine Annäherung an das vollständige, unstetige Logikmodell darstellt.

In diesem Abschnitt beschreiben wir sechs verschiedene Ansätze für die Informationsextraktion in der Reihenfolge zunehmender Komplexität in mehreren Dimensionen: deterministisch bis stochastisch, domänenspezifisch bis allgemein, manuell erzeugt bis gelernt und klein bis groß.

22.4.1 Endliche Automaten für die Informationsextraktion

Die einfachste Art eines Informationsextraktionssystems ist ein **attributbasiertes Extraktionssystem**, das davon ausgeht, dass der gesamte Text auf ein einzelnes Objekt verweist, und dessen Aufgabe darin besteht, Attribute dieses Objektes zu extrahieren. Zum Beispiel haben wir in *Abschnitt 12.7* das Problem erwähnt, die Menge der Attribute {*Hersteller*=*IBM*, *Modell*=*ThinkBook*970, *Preis*=*$*399.00} aus dem Text „IBM ThinkBook 970. Unser Preis: $399.00“ zu extrahieren. Um dieses Problem zu lösen, definieren wir eine **Vorlage** (auch als Muster oder Pattern bezeichnet) für jedes Attribut, das wir extrahieren möchten. Die Vorlage wird durch einen endlichen Automaten definiert. Das einfachste Beispiel hierfür ist der **reguläre Ausdruck**. Reguläre Ausdrücke werden unter anderem in Unix-Befehlen wie zum Beispiel `grep`, in Programmiersprachen wie Perl und in Textverarbeitungen wie beispielsweise Microsoft Word verwendet. Die einzelnen Tools weichen im Detail leicht voneinander ab, sodass es sich empfiehlt, die jeweilige Hilfe (das Handbuch) zu Rate zu ziehen. Hier zeigen wir aber exemplarisch, wie sich die Vorlage eines regulären Ausdrucks für Preise in Dollar erstellen lässt:

```
[0-9]                    stimmt mit einer beliebigen Ziffer von 0 bis 9
                         überein
[0-9 ]+                  stimmt mit einer oder mehreren Ziffern überein
[.][0-9][0-9]            stimmt mit einem Punkt gefolgt von zwei Ziffern
                         überein
([.][0-9][0-9])?         stimmt mit einem Punkt gefolgt von zwei Ziffern oder
nichts überein
[$][0-9 ]+(.[0-9][0-9])? stimmt mit $249.99 oder $1.23 oder $1000000
                         oder ... überein
```

Vorlagen werden oftmals mit drei Teilen definiert: je einem regulären Ausdruck für Präfix, Ziel und Postfix. Für Preise sieht das Ziel wie oben angegeben aus, das Präfix würde nach Zeichenfolgen wie zum Beispiel „Preis:“ suchen und das Postfix könnte leer sein. Die Idee besteht darin, dass bestimmte Anhaltspunkte zu einem Attribut vom Attributwert selbst kommen, andere vom umgebenden Text.

Wenn ein regulärer Ausdruck für ein Attribut genau einmal mit dem Text übereinstimmt, können wir den Teil des Textes, der den Wert des Attributes darstellt, herausziehen. Wenn es keine Übereinstimmung gibt, können wir höchstens einen Standardwert angeben oder das Attribut als fehlend belassen; doch wenn es mehrere Übereinstimmungen gibt, brauchen wir einen Prozess, um zwischen ihnen auszuwählen. Eine Strategie ist, mehrere reguläre Ausdrücke für jedes Attribut zu verwenden – geordnet nach der Priorität. So könnte zum Beispiel die Vorlage mit der höchsten Priorität für den Preis nach dem Präfix „unser Preis:“ suchen; wird dies nicht gefunden, suchen wir nach dem Präfix „Preis:“, und wenn dieses nicht gefunden wird, nach dem leeren Präfix. Eine andere Strategie wäre, alle Übereinstimmungen zu suchen und eine Möglichkeit zu finden, zwischen ihnen auszuwählen. Beispielsweise könnten wir den geringsten Preis nehmen, der innerhalb von 50% des höchsten Preises liegt. Damit wird zum Beispiel $78.00 als Ziel aus dem Text „Listenpreis $99.00, Sonderpreis $78.00, Versand $3.00“ ausgewählt.

Einen Schritt nach den attributbasierten Extraktionssystemen kommen die **relationalen Extraktionssysteme**, die sich mit mehreren Objekten und den Relationen zwischen ihnen beschäftigen. Wenn diese Systeme den Text „$249.99“ sehen, müssen sie nicht nur erkennen, dass dies ein Preis ist, sondern auch, welches Objekt diesen Preis hat. Ein typisches relational basiertes Extraktionssystem ist FASTUS, das Nachrichten über Unternehmensfusionen und Akquisitionen verarbeitet. Es liest beispielsweise die Meldung

> Bridgestone Sports Co. said Friday it has set up a joint venture in Taiwan with a local concern and a Japanese trading house to produce golf clubs to be shipped to Japan.

und extrahiert die Relationen:

$$e \in \mathit{JointVentures} \wedge \mathit{Product}(e, \text{"}\mathit{golf\ clubs}\text{"}) \wedge \mathit{Date}(e, \text{"}\mathit{Friday}\text{"}) \wedge \mathit{Member}(e, \text{"}\mathit{Bridgestone\ Sports\ Co}\text{"}) \wedge \mathit{Member}(e, \text{"}\mathit{a\ local\ concern}\text{"}) \wedge \mathit{Member}(e, \text{"}\mathit{a\ Japanese\ trading\ house}\text{"}).$$

Relationale Extraktionssysteme lassen sich als Folge von **kaskadierten endlichen Transduktoren** aufbauen. Das heißt, das System besteht aus einer Folge kleiner, effizienter endlicher Automaten (FSAs, Finite State Automata), wobei jeder Automat eine Texteingabe erhält, den Text in ein anderes Format übersetzt und zum nächsten Automaten weitergibt. FASTUS besteht aus fünf Phasen:

1. Token bilden
2. Komplexe Wörter verarbeiten
3. Basisgruppen verarbeiten
4. Komplexe Phrasen verarbeiten
5. Strukturen zusammenführen

Die erste Phase von FASTUS ist die **Token-Bildung**, die den Zeichenstrom in einzelne Token (Wörter, Zahlen und Interpunktionszeichen) unterteilt. Für das Englische kann die

Token-Bildung relativ leicht erfolgen; man unterteilt die Zeichen einfach an den Leerzeichen oder an den Interpunktionszeichen, was häufig ausreichend ist. Einige Tokenizer kommen auch mit Markup-Sprachen wie etwa HTML, SGML oder XML zurecht.

Die zweite Phase verarbeitet **komplexe Wörter**, einschließlich Wortkombinationen wie beispielsweise in „set up“ oder „joint venture“, aber auch Eigennamen wie beispielsweise „Bridgestone Sports Co.“. Sie werden durch eine Kombination aus lexikalischen Einträgen und endlichen Grammatikregeln erkannt. Ein Firmenname könnte etwa durch die folgende Regel erkannt werden:

GroßgeschriebenesWort+ ("Company" | "Co." | "Inc." | "Ltd").

Die dritte Phase verarbeitet **grundlegende Gruppen**, d.h. Substantivgruppen und Verbgruppen. Die Idee dabei ist, sie in Einheiten zu zerlegen, die von den späteren Phasen verarbeitet werden. *Kapitel 23* erläutert, wie man eine komplexe Beschreibung von Substantiv- und Verbgruppen schreibt. Hier begnügen wir uns mit einfachen Regeln, die die Komplexität des Englischen lediglich annähern, sich aber vorteilhaft als endlicher Automat darstellen lassen. In dieser Phase entsteht aus dem Beispielsatz die folgende Sequenz von gekennzeichneten Gruppen:

1 NG: Bridgestone Sports Co.	10 NG: a local concern
2 VG: said	11 CJ: and
3 NG: Friday	12 NG: a Japanese trading house
4 NG: it	13 VG: to produce
5 VG: hat set up	14 NG: golf clubs
6 NG: a joint venture	15 VG: to be shipped
7 PR: in	16 PR: to
8 NG: Taiwan	17 NG: Japan
9 PR: with	

Hier steht NG für Substantivgruppe (Noun Group), VG für Verbgruppe, PR für Präposition und CJ für Konjunktion.

Die vierte Phase kombiniert die grundlegenden Gruppen in **komplexe Phrasen**. Auch hier ist das Ziel, endliche Regeln zu erhalten, die schnell verarbeitet werden können und zu eindeutigen (oder fast eindeutigen) Ausgabephrasen führen. Eine Art Kombinationsregel beschäftigt sich mit domänenspezifischen Ereignissen. Beispielsweise deckt die Regel

Company+ SetUp JointVenture ("with" Company+)?

eine Möglichkeit ab, die Bildung eines Joint Venture zu beschreiben. Diese Phase ist die erste in der Kaskade, wo die Ausgabe sowohl in einer Datenbankvorlage als auch im Ausgabestrom platziert wird. Die letzte Phase **führt Strukturen zusammen**, die im vorhergehenden Schritt erzeugt wurden. Lautet der nächste Satz: „The joint venture will start production in January“, erkennt dieser Schritt, dass es zwei Verweise auf ein Joint Venture gibt und dass sie zu einem einzigen zusammengeführt werden sollten. Dies ist ein Beispiel für das **Problem der Identitätsunsicherheit**, das *Abschnitt 14.6.1* erläutert hat.

Im Allgemeinen funktioniert die endliche vorlagenbasierte Informationsextraktion gut für eine eingeschränkte Domäne, in der sich vorherbestimmen lässt, welche Themen diskutiert und wie sie erwähnt werden. Das Modell mit kaskadiertem Transduktor hilft, das erforderliche Wissen zu modularisieren, was die Konstruktion des Systems erleichtert. Diese Systeme arbeiten besonders gut, wenn sie Text zurücklesen, der von einem Programm generiert wurde. Zum Beispiel wird eine Shopping-Site im Web durch ein Programm generiert, das Datenbankeinträge liest und sie zu Webseiten formatiert; ein vorlagenbasierter Extrahierer stellt dann die ursprüngliche Datenbank wieder her. Endliche Informationsextraktion ist weniger erfolgreich beim Wiederherstellen von Informationen, die in stark variierenden Formaten enthalten sind, beispielsweise durch Menschen zu einer Vielfalt von Themen geschriebener Text.

22.4.2 Probabilistische Modelle für die Informationsextraktion

Bei einer Informationsextraktion von verrauschten oder variierenden Eingaben schneidet der einfache Ansatz mit endlichem Automat schlecht ab. Es ist schwierig, sämtliche Regeln und ihre Prioritäten ordnungsgemäß einzurichten. Ein probabilistisches Modell ist hier besser geeignet als ein regelbasiertes. Das einfachste probabilistische Modell für Sequenzen mit verborgenem Zustand ist das Hidden-Markov-Modell, kurz HMM.

Wie *Abschnitt 15.3* erläutert hat, modelliert ein HMM einen Durchlauf durch eine Folge von verborgenen Zuständen $\mathbf{x}_t$ mit einer Beobachtung $\mathbf{e}_t$ an jedem Schritt. Um HMMs auf Informationsextraktion anzuwenden, können wir entweder für sämtliche Attribute ein großes HMM oder für jedes Attribut ein eigenes HMM erstellen. Wir wählen hier den zweiten Weg. Die Beobachtungen sind die Wörter des Textes und die verborgenen Zustände geben an, ob wir uns im Ziel-, Präfix- oder Postfix-Teil der Attributvorlage bzw. im Hintergrund (nicht Teil der Vorlage) befinden. Das folgende Beispiel gibt einen kurzen Text und den wahrscheinlichsten (Viterbi-)Pfad für diesen Text für zwei HMMs an, wobei das eine trainiert wurde, um den Redner in der Ankündigung eines Vortrages zu erkennen, und das andere trainiert wurde, um Datumswerte zu erkennen. Der Bindestrich „-“ kennzeichnet einen Hintergrundzustand:

```
Text:     There will be a seminar by  Dr.    Andrew McCallum on   Friday
Sprecher: -     -    -  - PRE     PRE TARGET TARGET TARGET   POST -
Datum:    -     -    -  - -       -   -      ---- - PRE TARGET
```

HMMs besitzen gegenüber FSAs für die Extraktion zwei große Vorzüge: Erstens sind HMMs probabilistisch und somit tolerant gegen Rauschen. Wenn in einem regulären Ausdruck ein einzelnes erwartetes Zeichen fehlt, liefert der reguläre Ausdruck keine Übereinstimmung. Bei HMMs gibt es eine Fehlertoleranz mit fehlenden Zeichen/Wörtern und wir erhalten eine Wahrscheinlichkeit, die den Grad der Übereinstimmung anzeigt, und nicht nur eine boolesche Ja/Nein-Aussage, ob eine Übereinstimmung besteht. Zweitens lassen sich HMMs von Daten trainieren; sie setzen keine langwierige Entwicklung von Vorlagen voraus und lassen sich somit leicht auf dem neuesten Stand halten, wenn sich der Text mit der Zeit ändert.

Wir haben hier ein bestimmtes Strukturniveau in unseren HMM-Vorlagen vorausgesetzt: Sie bestehen alle aus einem oder mehreren Zielzuständen, alle Präfixzustände müssen vor den Zielen erscheinen, Postfixzustände müssen auf die Ziele folgen und andere Zustände müssen den Hindergrund bilden. Diese Struktur erleichtert es, HMMs von Beispielen zu

erlernen. Mit einer partiell spezifizierten Struktur lässt sich der Vorwärts-Rückwärts-Algorithmus einsetzen, um sowohl die Übergangswahrscheinlichkeiten ($\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1})$ zwischen den Zuständen zu lernen als auch das Beobachtungsmodell $\mathbf{P}(\mathbf{E}_t \mid \mathbf{X}_t)$, das aussagt, wie wahrscheinlich jedes Wort in jedem Zustand ist. Zum Beispiel würde das Wort „Friday“ eine hohe Wahrscheinlichkeit in einem oder mehreren der Zielzustände des Datums-HMM haben und sonst eine geringere Wahrscheinlichkeit.

Mit ausreichenden Trainingsdaten lernt das HMM automatisch eine Struktur von Datumswerten, die wir intuitiv finden: Das Datums-HMM könnte einen Zielzustand besitzen, in dem die Wörter mit hoher Wahrscheinlichkeit „Monday“, „Tuesday“ etc. lauten und das einen Übergang hoher Wahrscheinlichkeit zu einem Zielzustand mit den Wörtern „Jan“, „January“, „Feb“ etc. besitzt. ▶ Abbildung 22.2 zeigt das HMM für den Sprecher einer Gesprächsankündigung, wie es aus Daten gelernt wurde. Das Präfix behandelt Ausdrücke wie zum Beispiel „speaker:“ und „seminar by“, das Ziel hat einen Zustand, der Titel und Vornamen behandelt, und ein weiterer Zustand verarbeitet die Initialen und die Nachnamen.

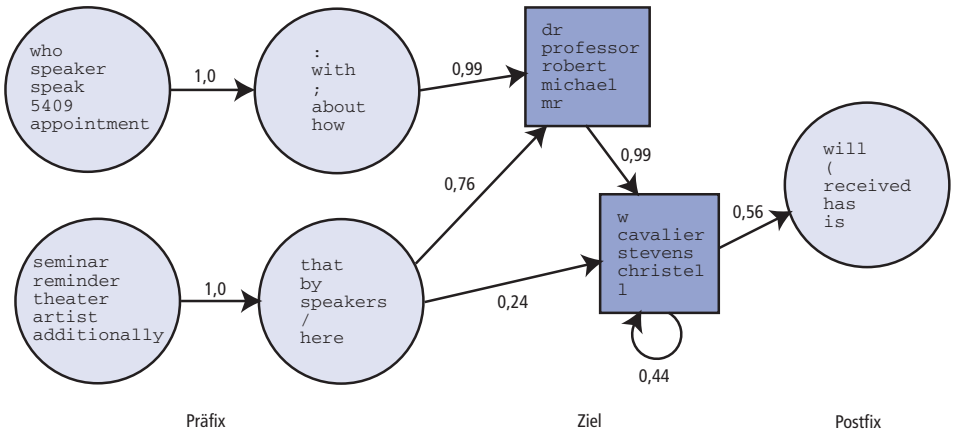


Abbildung 22.2: Hidden-Markov-Modell für den Sprecher (speaker) einer Gesprächsankündigung. Die beiden quadratischen Zustände bilden das Ziel (wobei der zweite Zielzustand eine auf sich selbst verweisende Schleife ist, sodass das Ziel eine Zeichenfolge beliebiger Länge vergleichen kann), die vier Kreise auf der linken Seite geben den Präfixteil an und der Kreis auf der rechten Seite den Postfixteil. Für jeden Zustand sind nur einige der Wörter mit hoher Wahrscheinlichkeit angegeben. Nach Freitag und McCallum (2000).

Nachdem die HMMs gelernt wurden, können wir sie auf einen Text anwenden, wobei wir mithilfe des Viterbi-Algorithmus den wahrscheinlichsten Pfad durch die HMM-Zustände suchen. Ein Ansatz wendet jedes Attribut-HMM separat an; in diesem Fall ist zu erwarten, dass die meisten HMMs den größten Teil ihrer Zeit in Hintergrundzuständen verbringen. Dies ist zweckmäßig, wenn die Extraktion dünn besetzt ist – wenn die Anzahl der extrahierten Wörter klein im Vergleich zur Länge des Textes ist.

Der andere Ansatz kombiniert alle individuellen Attribute zu einem großen HMM, das dann einen Pfad sucht, der die verschiedenen Zielattribute durchläuft, zuerst ein Sprecherziel findet, dann ein Datumsziel usw. Getrennte HMMs sind besser geeignet, wenn wir nur einen Vertreter von jedem Attribut in einem Text erwarten, und ein großes HMM bietet sich an, wenn die Texte ungebundener und dichter mit Attributen

besetzt sind. Bei jedem Ansatz kommen wir letztlich zu einer Sammlung von Zielattributbeobachtungen und müssen entscheiden, wie wir sie verarbeiten wollen. Wenn jedes erwartete Attribut genau ein Zielfüllelement besitzt, ist die Entscheidung einfach: Wir haben eine Instanz der gewünschten Relation. Gibt es mehrere Füllelemente, müssen wir eines davon auswählen, wie es für vorlagenbasierte Systeme erläutert wurde. HMMs besitzen den Vorteil, dass die gelieferten Wahrscheinlichkeitswerte bei der Auswahl helfen. Fehlen einige Attribute, müssen wir entscheiden, ob dies überhaupt eine Instanz der gewünschten Relation ist oder ob die Ziele als falsch positiv zu bewerten sind. Ein Algorithmus des maschinellen Lernens lässt sich trainieren, um diese Auswahl zu treffen.

22.4.3 Bedingte Zufallsfelder für die Informationsextraktion

HMMs für die Informationsextraktion weisen ein Problem auf: Sie modellieren eine Menge Wahrscheinlichkeiten, die wir eigentlich nicht brauchen. Ein HMM ist ein generatives Modell. Da es die vollständige Verknüpfungswahrscheinlichkeit der Beobachtungen und verborgenen Zustände modelliert, lässt es sich verwenden, um Stichproben zu generieren. Das heißt, wir verwenden das HMM-Modell nicht nur, um einen Text zu analysieren und den Sprecher und das Datum wiederherzustellen, sondern auch, um eine zufällige Instanz eines Textes zu generieren, der einen Sprecher und ein Datum enthält. Da wir aber an dieser Aufgabe nicht interessiert sind, liegt die Frage nahe, ob wir mit einem Modell besser fahren, das sich gar nicht erst mit der Modellierung dieser Wahrscheinlichkeit befasst. Um einen Text zu verstehen, genügt uns ein **Diskriminanzmodell**, d.h. ein Modell, das die bedingte Wahrscheinlichkeit der verborgenen Attribute für die gegebenen Beobachtungen (den Text) modelliert. Für einen Text $\mathbf{e}_{1:N}$ findet das bedingte Modell die verborgene Zustandsfolge $\mathbf{X}_{1:N}$, die $P(\mathbf{X}_{1:N} \mid \mathbf{e}_{1:N})$ maximiert.

Wenn wir dies direkt modellieren, schaffen wir uns einige Freiheiten. Wir brauchen die Unabhängigkeitsannahmen des Markov-Modells nicht – wir können ein xt haben, das von x1 abhängig ist. Ein Gerüst für ein derartiges Modell ist das **Conditional Random Field** (CRF), das eine bedingte Wahrscheinlichkeitsverteilung einer Menge von Zielvariablen für eine gegebene Menge von beobachteten Variablen modelliert. CRFs können genau wie Bayessche Netze viele unterschiedliche Strukturen von Abhängigkeiten unter den Variablen darstellen. Eine häufige Struktur ist das **Linear-Chain Conditional Random Field**-Modell für die Darstellung von Markov-Abhängigkeiten unter Variablen in einer zeitlichen Folge. Somit sind HMMs die zeitliche Version von naiven Bayes-Modellen und Linear-Chain CRFs die zeitliche Version der logistischen Regression, während das vorhergesagte Ziel eine vollständige Zustandssequenz statt einer einzelnen binären Variablen ist.

Es seien $\mathbf{e}_{1:N}$ die Beobachtungen (z.B. Wörter in einem Dokument) und $\mathbf{x}_{1:N}$ die Sequenz der verborgenen Zustände (z.B. die Präfix-, Ziel- und Postfixzustände). Ein linear-verkettetes CRM definiert eine bedingte Wahrscheinlichkeitsverteilung:

$$\mathbf{P}(\mathbf{x}_{1:N} \mid \mathbf{e}_{1:N}) = \alpha e^{\left[\sum_{i=1}^{N} F(\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{e}, i)\right]}.$$

Hier ist α ein Normalisierungsfaktor (um die Summierung der Wahrscheinlichkeiten zu 1 zu gewährleisten) und F eine Merkmalsfunktion, die als gewichtete Summe einer Sammlung von k Komponentenmerkmalsfunktionen definiert ist:

$$F(\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{e}, i) = \sum_k \lambda_k f_k(\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{e}, i).$$

Die λ_k-Parameterwerte werden mit einer MAP (Maximum A-posteriori)-Bewertungsprozedur gelernt, die die bedingte Wahrscheinlichkeit der Trainingsdaten maximiert. Die Merkmalsfunktionen bilden die Schlüsselkomponenten eines CRF. Die Funktion f_k hat Zugriff auf ein Paar von benachbarten Zuständen $\mathbf{x}_{i-1}$ und $\mathbf{x}_i$, aber auch auf die gesamte Beobachtungssequenz $\mathbf{e}$ (Wort) sowie die aktuelle Position in der zeitlichen Sequenz i. Damit sind wir recht flexibel bei der Definition von Merkmalen. Wir können eine einfache Merkmalsfunktion definieren, beispielsweise eine, die den Wert 1 liefert, wenn das aktuelle Wort ANDREW lautet und der aktuelle Zustand SPEAKER ist:

$$f_1(\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{e}, i) = \begin{cases} 1, \text{wenn } \mathbf{x}_i = \mathit{SPEAKER} \text{ und } \mathbf{e}_i = \mathit{ANDREW} \\ 0, \text{ andernfalls} \end{cases}.$$

Wie werden nun derartige Merkmale verwendet? Das hängt von ihren jeweiligen Gewichten ab. Wenn $\lambda_1 > 0$ ist, wird bei f_1 gleich *true* die Wahrscheinlichkeit der verborgenen Zustandsfolge $\mathbf{x}_{1:N}$ erhöht. Man kann auch sagen: „Das CRF-Modell sollte den Zielzustand SPEAKER dem Wort ANDREW vorziehen." Ist dagegen $\lambda_1 < 0$, versucht das CRF-Modell, diese Zuordnung zu vermeiden, und wenn $\lambda_1 = 0$, wird dieses Merkmal ignoriert. Parameterwerte lassen sich manuell festlegen oder aus Daten lernen. Sehen Sie sich eine zweite Merkmalsfunktion an:

$$f_2(\mathbf{x}_{i-1}, \mathbf{x}_i, \mathbf{e}, i) = \begin{cases} 1, \text{wenn } \mathbf{x}_i = \mathit{SPEAKER} \text{ und } \mathbf{e}_i = \mathit{SAID} \\ 0, \text{ andernfalls} \end{cases}.$$

Dieses Merkmal ist *true*, wenn der aktuelle Zustand SPEAKER und das nächste Wort „said" lautet. Man würde also für das Merkmal einen positiven λ_2-Wert erwarten. Interessanter ist aber, dass sowohl f_1 als auch f_2 gleichzeitig für einen Satz wie „Andrew said ..." gelten können. In diesem Fall überlappen sich die beiden Merkmale und verstärken beide den Glauben in $\mathbf{x}_1$ = SPEAKER. Aufgrund der Unabhängigkeitsannahme können HMMs keine überlappenden Merkmale verwenden; CRFs dagegen schon. Darüber hinaus kann ein Merkmal in einem CRF auf jeden Teil der Sequenz $\mathbf{e}_{1:N}$ zugreifen. Außerdem lassen sich Merkmale über Zustandsübergängen definieren. In den bisher angegebenen Beispielen haben wir binäre Merkmale definiert, doch im allgemeinen Fall kann eine Merkmalsfunktion jede reellwertige Funktion sein. Für Domänen, in denen wir ein bestimmtes Wissen über die Typen der Merkmale besitzen, die wir einbinden möchten, bietet uns der CRF-Formalismus ein großes Maß an Flexibilität, die Merkmale zu definieren. Diese Flexibilität kann zu höheren Genauigkeiten als bei nicht so flexiblen Modellen wie etwa HMMs führen.

22.4.4 Ontologieextraktion aus großen Korpora

Bisher haben wir unter Informationsextraktion verstanden, eine spezifische Menge von Relationen (z.B. Sprecher, Zeit, Ort) in einem bestimmten Text (z.B. einer Gesprächsankündigung) zu finden. Eine andere Anwendung der Extraktionstechnik ist der Aufbau einer großen Wissensbasis oder Ontologie von Fakten aus einem Korpus. Diese Technik zeichnet sich durch drei Unterschiede aus: Erstens ist sie ergebnisoffen – wir möchten

Fakten über alle Domänentypen erfassen und nicht nur über eine spezifische Domäne. Zweitens wird diese Aufgabe bei einem großen Korpus von der Genauigkeit und nicht vom Recall dominiert – genau wie bei Frage-Antwort-Dialogen im Web (*Abschnitt 22.3.6*). Drittens können die Ergebnisse statistische Zusammenfassungen sein, die aus mehreren Quellen stammen, anstatt aus einem spezifischen Text extrahiert zu werden.

Zum Beispiel hat Hearst (1992) das Problem betrachtet, eine Ontologie von Konzeptkategorien und Unterkategorien aus einem großen Korpus zu lernen. (In 1992 war eine 1000-seitige Enzyklopädie ein großer Korpus; heute würde es ein Web-Korpus mit 100 Millionen Seiten sein.) Im Mittelpunkt seiner Arbeit standen Vorlagen, die sehr allgemein gehalten (nicht an eine spezifische Domäne gebunden) waren und eine hohe Genauigkeit aufwiesen (bei einer Übereinstimmung fast immer korrekt waren), aber einen geringen Recall hatten (nicht immer übereinstimmten). Zum Beispiel sieht eine der produktivsten Vorlagen wie folgt aus:

$$NP \textbf{ such as } NP\ (\textbf{,}\ NP)^*\ (\textbf{,})?\ ((\textbf{and} \mid \textbf{or})\ NP)?$$

Hier müssen die fettgedruckten Wörter und Kommas literal im Text erscheinen. Die Klammern dienen lediglich der Gruppierung, das Sternchen bedeutet *null oder mehr Wiederholungen* und das Fragezeichen heißt *optional*. Die Variable NP steht für Substantivphrase (Noun Phrase). *Kapitel 23* beschreibt, wie sich Substantivphrasen identifizieren lassen. Momentan nehmen wir einfach an, dass wir bestimmte Wörter als Substantive kennen und es andere Wörter (etwa Verben) gibt, bei denen wir zuverlässig annehmen können, dass sie nicht Teil einer einfachen Substantivphrase sind. Diese Vorlage liefert Übereinstimmungen bei Texten wie „diseases such as rabies affect your dog“ und „supports network protocols such as DNS“ mit der Schlussfolgerung, dass „rabies“ (Tollwut) eine Krankheit („disease“) und „DNS“ ein Netzwerkprotokoll ist. Ähnliche Vorlagen lassen sich mit den Schlüsselwörtern „including“ (einschließlich), „especially“ (insbesondere) und „or other“ (oder andere) konstruieren. Natürlich ergeben diese Vorlagen keine Übereinstimmungen bei vielen relevanten Passagen wie zum Beispiel „Rabies is a disease“. Das ist beabsichtigt. Manchmal kennzeichnet die Vorlage „*NP* ist ein *NP*“ tatsächlich eine Unterkategoriebeziehung, doch oftmals meint sie etwas anderes, wie zum Beispiel in „There is a God“ (Es gibt einen Gott) oder „She is a little tired“ (Sie ist ein wenig müde). Mit einem großen Korpus können wir es uns leisten, wählerisch zu sein – d.h. nur die Vorlagen mit hoher Genauigkeit verwenden. Wir verfehlen viele Aussagen einer Unterkategoriebeziehung, doch höchstwahrscheinlich finden wir an einer anderen Stelle des Korpus eine Umschreibung in einer Form, auf die wir zurückgreifen können.

22.4.5 Automatisierte Vorlagenkonstruktion

Die *Unterkategorie*-Beziehung ist so grundlegend, dass es sich lohnt, einige Vorlagen manuell zu erstellen. Das ist hilfreich, wenn es darum geht, Instanzen dieser Beziehung in einem natürlichsprachigen Text zu identifizieren. Doch wie sieht es mit den Tausenden anderen Beziehungen aus? Es gibt nicht genügend KI-Doktoranden auf der Welt, um Vorlagen für alle diese Beziehungen zu erstellen und auf Fehlerfreiheit zu untersuchen. Zum Glück ist es möglich, Vorlagen von einigen Beispielen zu lernen und dann mithilfe dieser Vorlagen weitere Beispiele zu lernen, von denen sich weitere Vorlagen lernen lassen, usw. In einem der ersten derartigen Experimente begann Brin (1999) mit einer Datenmenge von nur fünf Beispielen:

(“Isaac Asimov”, “The Robots of Dawn”)
(“David Brin”, “Startide Rising”)
(“James Gleick”, “Chaos – Making a New Science”)
(“Charles Dickens”, “Great Expectations”)
(“William Shakespeare”, “The Comedy of Errors”).

Zweifellos sind dies Beispiele für die Autor-Titel-Relation, doch das lernende System kannte weder Autoren noch Titel. Die Wörter in diesen Beispielen wurden in einer Suche über einen Web-Korpus verwendet, was zu 199 Treffern führte. Jede Übereinstimmung ist als Tupel von sieben Zeichenfolgen definiert:

(*Author*, *Title*, *Order*, *Prefix*, *Middle*, *Postfix*, *URL*).

Hier ist *Order* (Reihenfolge) *true*, wenn *Author* zuerst kommt, und *false*, wenn *Title* als Erstes erscheint. *Middle* bezeichnet die Zeichen zwischen *Author* und *Title*, *Prefix* gibt die 10 Zeichen vor der Übereinstimmung an, *Suffix* steht für die 10 Zeichen nach der Übereinstimmung und *URL* ist die Webadresse, bei der die Übereinstimmung gefunden wurde.

Für eine gegebene Menge von Übereinstimmungen kann ein einfaches Schema zur Vorlagengenerierung Vorlagen suchen, um die Übereinstimmungen zu erklären. Die Sprache der Vorlage wurde so konzipiert, dass eine enge Zuordnung zu den Übereinstimmungen selbst entsteht (eventuell mit dem Risiko eines geringeren Recall). Jede Vorlage weist die gleichen sieben Komponenten als Übereinstimmung auf. Die regulären Ausdrücke *Author* und *Title* bestehen aus beliebigen Zeichen (beginnen und enden aber mit Buchstaben) und sind hinsichtlich der Länge beschränkt, und zwar von der Hälfte der minimalen Länge der Beispiele bis zum Doppelten der maximalen Länge. Die *Prefix*-, *Middle*- und *Postfix*-Elemente bestehen nur aus literalen Zeichenfolgen, nicht aus regulären Ausdrücken. Am einfachsten ist das *Middle*-Element zu lernen: Jede individuelle mittlere Zeichenfolge in der Menge der Übereinstimmungen ist eine klare Kandidatenvorlage. Für jeden derartigen Kandidaten wird dann das *Prefix* der Vorlage als längstes gemeinsames Suffix aller Präfixe in der Übereinstimmung definiert und das *Postfix* als längstes gemeinsames Präfix aller Postfix-Elemente in den Übereinstimmungen. Wenn eines dieser Elemente die Länge null hat, wird die Vorlage zurückgewiesen. Das *URL*-Element der Vorlage ist als längstes Präfix der URL-Elemente in den Übereinstimmungen definiert.

In dem von Brin durchgeführten Experiment haben die ersten 199 Treffer drei Vorlagen generiert. Die produktivste Vorlage war:

<LI><B> *Title* </B> by *Author* (
URL: www.sff.net/locus/c

Die drei Vorlagen dienten dazu, weitere 4047 (Autor, Title)-Beispiele abzurufen. Mit diesen Beispielen wurden anschließend weitere Vorlagen generiert. Das setzte sich fort, bis über 15.000 Titel vorhanden waren. Für einen guten Satz von Vorlagen kann das System einen guten Satz von Beispielen sammeln. Für einen guten Satz von Beispielen kann das System einen guten Satz von Vorlagen erstellen.

Die größte Schwäche in diesem Konzept ist die Empfindlichkeit gegen Rauschen. Wenn eine der ersten Vorlagen falsch ist, können sich die Fehler schnell fortpflanzen. Dieses Problem lässt sich eindämmen, wenn man beispielsweise erst ein neues Beispiel akzeptiert, wenn es durch mehrere Vorlagen verifiziert wurde, und eine neue Vorlage erst akzeptiert, wenn sie mehrere Beispiele erkennt, die auch durch andere Vorlagen gefunden wurden.

22.4.6 Maschinelles Lesen

Die automatisierte Vorlagenkonstruktion bedeutet einen großen Schritt von der manuellen Vorlagenkonstruktion vorwärts, setzt aber immer noch als Einstieg eine Handvoll beschrifteter Beispiele für jede Relation voraus. Um eine große Ontologie mit vielen tausend Relationen aufzubauen, wäre dieses Vorgehen recht mühsam; wir brauchen ein Extraktionssystem, das *keinerlei* menschliche Eingaben erfordert – ein System, das selbstständig lesen und seine eigene Datenbank aufbauen könnte. Ein derartiges System wäre relationsunabhängig und würde demnach für jede beliebige Relation funktionieren. In der Praxis arbeiten diese Systeme aufgrund der Ein-/Ausgabe-Anforderungen großer Korpora für *sämtliche* Relationen parallel. Sie verhalten sich weniger wie ein herkömmliches Informationsextraktionssystem, das auf einige wenige Relationen ausgerichtet ist, sondern mehr wie ein menschlicher Leser, der aus dem Text selbst lernt. Aus diesem Grund bezeichnet man dieses Gebiet als **maschinelles Lesen**.

Ein repräsentatives Maschinenlesesystem ist TEXTRUNNER (Banko und Etzioni, 2008). Es verwendet Co-Training, um seine Leistung zu verstärken, benötigt aber eine gewisse „Starthilfe“ (Bootstrap). Im Fall von Hearst (1992) wurden spezifische Muster (wie zum Beispiel *such as*) als Ausgangspunkt bereitgestellt, während es für Brin (1998) ein Satz von fünf Autor-Titel-Paaren war. Die ursprüngliche Inspiration für TEXTRUNNER war eine Nomenklatur von acht sehr allgemeinen syntaktischen Vorlagen, wie sie ▶ Abbildung 22.3 zeigt. Man ist der Auffassung, dass eine kleine Anzahl derartiger Vorlagen die meisten Möglichkeiten abdeckt, wie sich Beziehungen in Englisch ausdrücken lassen. Das eigentliche Bootstrapping beginnt mit einer kleinen Menge beschrifteter Beispiele, die aus der *Penn Treebank* – einem Korpus von geparsten Sätzen – extrahiert werden. Zum Beispiel ist TEXTRUNNER beim Parsing des Satzes "Einstein received the Nobel Prize in 1921" in der Lage, die Relation ("Einstein", "received", "Nobel Prize") zu extrahieren.

Für eine Menge beschrifteter Beispiele dieses Typs trainiert TEXTRUNNER ein Linear-Chain CRF, um weitere Beispiele aus nicht beschriftetem Text zu extrahieren. Die Merkmale im CRF umfassen Funktionswörter wie „to“, „of“ und „the“, jedoch weder Substantive noch Verben (und auch keine Substantiv-Phrasen oder Verb-Phrasen). Da TEXTRUNNER domänenunabhängig ist, kann sich das System nicht auf vordefinierte Listen von Substantiven und Verben stützen.

Typ	Vorlage	Beispiel	Häufigkeit
Verb	NP_1 *Verb* NP_2	X established Y	38%
Noun-Prep	NP_1 *NP Prep* NP_2	X settlement with Y	23%
Verb-Prep	NP_1 *Verb Prep* NP_2	X moved to Y	16%
Infinitive	NP_1 **to** *Verb* NP_2	X plans to acquire Y	9%
Modifier	NP_1 *Verb* NP_2 *Noun*	X is Y winner	5%
Noun-Coordinate	NP_1 (**,** | **and** | – | :) NP_2 *NP*	X-Y deal	2%
Verb-Coordinate	NP_1 (**,**| **and**) NP_2 *Verb*	X, Y merge	1%
Appositive	NP_1 *NP* (**:**| **,**)? NP_2	X hometown : Y	1%

Abbildung 22.3: Acht allgemeine Vorlagen, die rund 95% der Möglichkeiten abdecken, wie Relationen in Englisch ausgedrückt werden.

TEXTRUNNER erreicht für einen großen Webkorpus eine Genauigkeit von 88% und einen Recall von 45% (F_1 von 60%). Dabei hat TEXTRUNNER Hunderte von Millionen Fakten aus einem Korpus mit dem Umfang einer halben Milliarde Webseiten extrahiert. Obwohl das System zum Beispiel kein vordefiniertes medizinisches Wissen besitzt, hat es über 2000 Antworten auf die Frage [what kills bacteria] (was tötet Bakterien) extrahiert. Zu den korrekten Antworten gehören unter anderem Antibiotika, Ozon, Chlor, Cipro und Brokkolisprossen. Eine problematische Antwort ist zum Beispiel „Wasser", die auf den Satz „Boiling water for at least 10 minutes will kill bacteria." (Kochendes Wasser tötet Bakterien nach höchstens 10 Minuten.) zurückgeht. Es wäre besser, dies mit dem Attribut „boiling water" (kochendes Wasser) als einfach nur „water" (Wasser) zu versehen.

Mit den in diesem Kapitel skizzierten Techniken und den kontinuierlich erscheinenden Erfindungen kommen wir langsam unserem Ziel für das Maschinenlesen näher.

Bibliografische und historische Hinweise

N-Gramm-Buchstabenmodelle für die Sprachmodellierung wurden von Markov (1913) vorgeschlagen. Claude Shannon (Shannon und Weaver, 1949) war der erste, der *n*-Gramm-Wortmodelle des Englischen erzeugt hat. Chomsky (1956, 1957) wies auf die Beschränkungen hin, die endliche Automaten gegenüber kontextfreien Modellen aufweisen, und schlussfolgerte: „Probabilistische Modelle geben keine besonderen Einblicke in einige der grundlegenden Probleme syntaktischer Struktur." Das ist zwar richtig, doch liefern probabilistische Modelle Einblicke in manche andere grundlegende Probleme – und zwar Probleme, die kontextfreie Modelle ignorieren. Die Anmerkungen von Chomsky hatten den bedauerlichen Effekt, über zwei Jahrzehnte hinweg viele Forscher von statistischen Modellen abzuschrecken, bis diese Modelle wieder in der Spracherkennung auftauchten (Jelinek, 1976).

Kessler et al. (1997) zeigen, wie sich *n*-Gramm-Modelle auf die Genreklassifizierung anwenden lassen, und Klein et al. (2003) beschreiben die Identifikation von Eigennamen (Named Entity Recognition) mit Zeichenmodellen. Franz und Brants (2006) beschreiben den jetzt öffentlich verfügbaren *n*-Gramm-Korpus von Google mit 13 Millionen eindeutigen Wörtern aus einer Billiarde Wörtern Webtext. Der Name des **Bag of Words**-Modells stammt aus einer Passage des Linguisten Zellig Harris (1954): „Language is not merely a bag of words but a tool with particular properties." (Sprache ist nicht einfach ein Sack voller Wörter, sondern ein Werkzeug mit besonderen Eigenschaften.) Norvig (2009) gibt Beispiele für Aufgaben an, die sich mit *n*-Gramm-Modellen realisieren lassen.

Die zuerst von Pierre-Simon Laplace (1816) vorgeschlagene Add-One-Glättung wurde von Jeffreys (1948) formalisiert; die Interpolationsglättung geht auf Jelinek und Mercer (1980) zurück, die das Verfahren für die Spracherkennung einsetzten. Andere Techniken sind die Witten-Bell-Glättung (1991), die Good-Turing-Glättung (Church und Gale, 1991) und die Kneser-Ney-Glättung (1995). Chen und Goodman (1996) sowie Goodman (2001) geben einen Überblick über Glättungstechniken.

Einfache *n*-Gramm-Buchstaben- und Wortmodelle sind nicht die einzig möglichen probabilistischen Modelle. Blei et al. (2001) beschreiben mit **Latent Dirichlet Allocation** ein probabilistisches Textmodell, das ein Dokument als Mischung von Themen – jeweils mit eigener Verteilung von Wörtern – betrachtet. Dieses Modell lässt sich als Erweiterung und Rationalisierung des Modells mit **latenter semantischer Indizierung** von Deerwester et al. (1990) ansehen (siehe auch Papadimitriou et al. (1998) und ist auch mit dem Modell Multiple-Cause Mixture von Sahami et al. (1996) verwandt.

Manning und Schütze (1999) sowie Sebastiani (2002) geben einen Überblick über Techniken zur Textklassifizierung. Joachims (2001) liefert mithilfe der statistischen Lerntheorie und Support Vector-Maschinen eine theoretische Analyse, wann Klassifizierung erfolgreich sein wird. Apté et al. (1994) melden eine Genauigkeit von 96% bei der Klassifizierung von Reuters-Nachrichtenartikeln in die Kategorie „Earnings“ (Einkommen/Verdienst). Koller und Sahami (1997) berichten eine Genauigkeit bis zu 95% mit einem Bayesschen Klassifikator und bis zu 98,6% mit einem Bayesschen Klassifikator, der bestimmte Abhängigkeiten zwischen den Merkmalen berücksichtigt. Lewis (1998) gibt einen Überblick über vierzig Jahre der Anwendung naiver Bayesscher Techniken für Textklassifizierung und -abrufe. Schapire und Singer (2000) zeigen, dass einfache lineare Klassifikatoren oftmals eine Genauigkeit erreichen können, die fast so gut wie bei komplexeren Modellen ist, und weit effizienter auszuwerten sind. Nigam et al. (2000) geben an, wie der EM-Algorithmus verwendet werden kann, um nicht beschriftete Dokumente zu beschriften, sodass sich ein besseres Klassifizierungsmodell lernen lässt. Witten et al. (1999) beschreiben Komprimierungsalgorithmen für die Klassifizierung und zeigen die tiefe Verbindung zwischen dem LZW-Komprimierungsalgorithmus und den Maximum-Entropie-Sprachmodellen.

Viele Techniken der *n*-Gramm-Modelle werden auch in Problemen der Bioinformatik verwendet. Biostatistische und probabilistische Verarbeitung natürlicher Sprache rücken enger zusammen, da sich beide Zweige mit langen, strukturierten Sequenzen befassen, die aus einem Alphabet von Konstituenten (Bestandteilen) ausgewählt werden.

Das Gebiet des **Informationsabrufes** erfährt ein wieder zunehmendes Interesse, was auf die breite Nutzung des Internets zurückzuführen ist. Robertson (1977) gibt einen frühen Überblick und führt das Prinzip der Rangfolgewahrscheinlichkeit (Probability Ranking Principle) ein. Croft et al. (2009) und Manning et al. (2008) sind die ersten Lehrbücher, die webbasierte Suche sowie herkömmlichen Informationsabruf abdecken. Hearst (2009) behandelt Benutzeroberflächen für die Websuche. Die von der US-Standardisierungsbehörde NIST (National Institute of Standards and Technology) organisierte Konferenz TREC veranstaltet einen jährlichen Wettbewerb für Informationsabrufsysteme und veröffentlicht Tagungsberichte mit Ergebnissen. In den ersten sieben Jahren des Wettbewerbes hat sich die Leistung ungefähr verdoppelt.

Das bekannteste Modell für den Informationsabruf ist das **Vektorraummodell** (Salton et al., 1975). Die Arbeit von Salton war in den frühen Jahren des Gebietes maßgebend. Alternative probabilistische Modelle stammen zum einen von Ponte und Croft (1998) und zum anderen von Maron und Kuhns (1960) sowie Robertson und Sparck Jones (1976). Lafferty und Zhai (2001) zeigen, dass die Modelle auf der gleichen gemeinsamen Wahrscheinlichkeitsverteilung basieren, aber sich die Wahl des Modells auf das Trainieren der Parameter auswirkt. Craswell et al. (2005) beschreiben die BM25-Bewertungsfunktion, während Svore und Burges (2009) zeigen, wie BM25 mit einem Ansatz zum maschinellen Lernen verbessert werden kann, der Klick-Daten einbezieht – Beispiele von vergangenen Suchanfragen und den Ergebnissen, die angeklickt wurden.

Brin und Page (1998) beschreiben den PageRank-Algorithmus und die Implementierung einer Websuchmaschine. Kleinberg (1999) beschreibt den HITS-Algorithmus. Silverstein et al. (1998) untersuchen ein Protokoll von einer Milliarde Websuchen. Das Journal *Information Retrieval* und die Tagungsberichte der jährlichen *SIGIR*-Konferenz behandeln jüngere Entwicklungen auf diesem Gebiet.

Frühe Informationsextraktionsprogramme sind unter anderem GUS (Bobrow et al., 1977) und FRUMP (DeJong, 1982). In der letzten Zeit wurde die Informationsextraktion durch die jährlichen *Message Understand Conferences* (MUC) vorangetrieben, die von der US-Regierung gesponsert werden. Das FASTUS-System auf Basis eines endlichen Automaten stammt von Hobbs et al. (1997). Es beruht zum Teil auf den Ideen von Pereira und Wright (1991), endliche Automaten als Annäherungen für Phrasenstrukturgrammatiken zu verwenden. Roche und Schabes (1997), Appelt (1999) und Muslea (1999) geben Überblicke über vorlagenbasierte Systeme. Große Faktendatenbanken wurden von Craven et al. (2000), Pasca et al. (2006), Mitchell (2007) sowie Durme und Pasca (2008) extrahiert.

Freitag und McCallum (2000) diskutieren HMMs für die Informationsextraktion. CRFs (Conditional Random Fields) wurden von Lafferty et al. (2001) eingeführt; McCallum (2003) beschreibt ein Beispiel für ihren Einsatz bei der Informationsextraktion und Sutton und McCallum (2007) legen ein Tutorial mit praktischen Anleitungen vor. Sarawagi (2007) gibt einen umfassenden Überblick.

Banko et al. (2002) präsentieren das ASKMSR-Frage-Antwort-System; ein ähnliches System geht auf Kwok et al. (2001) zurück. Pasca und Harabagiu (2001) diskutieren ein preisgekröntes System zum Beantworten von Fragen (Question Answering, QA). Zwei frühe einflussreiche Ansätze für die automatisierte Wissensverarbeitung stammen von Riloff (1993), der zeigte, dass ein automatisch konstruiertes Wörterbuch fast genauso gut abschneidet wie ein mit großer Sorgfalt manuell erstelltes domänenspezifisches Wörterbuch, und von Yarowsky (1995), der zeigte, dass die Wortsinn-Klassifizierung (siehe *Abschnitt 18.11.2*) durch nicht überwachtes Training mit einem Korpus aus nicht beschriftetem Text eine Genauigkeit erreicht, die der von überwachten Methoden entspricht.

Die Idee der gleichzeitigen Extrahierung von Vorlagen und Beispielen aus einer Handvoll von beschrifteten Beispielen wurde unabhängig und gleichzeitig entwickelt von Blum und Mitchell (1998), die von **Co-Training** sprechen, sowie von Brin (1998), der das Verfahren als DIPRE (Dual Iterative Pattern Relation Extraction) bezeichnet. Es dürfte klar sein, warum sich der Begriff *Co-Training* gehalten hat. Ähnliche frühere Arbeiten wurden von Jones et al. (1999) unter dem Namen Bootstrapping durchgeführt. Die Methode wurde durch die Systeme QXTRACT (Agichtein und Gravano, 2003) und KNOWITALL (Etzioni et al., 2005) vorangetrieben. Maschinelles Lesen wurde von Mitchell (2005) und Etzioni et al. (2006) eingeführt und steht im Fokus des TEXTRUNNER-Projektes (Banko et al., 2007; Banko und Etzioni, 2008).

Dieses Kapitel hat sich auf Text in natürlicher Sprache konzentriert, doch lassen sich Informationen auch anhand der physikalischen Struktur oder des Layouts des Textes statt anhand der linguistischen Struktur extrahieren. HTML-Listen und Tabellen sowohl in HTML- als auch relationalen Datenbanken beheimaten Daten, die extrahiert und konsolidiert werden können (Hurst, 2000; Pinto et al., 2003; Cafarella et al., 2008).

Die *Association for Computational Linguistics* (ACL) hält jährliche Konferenzen ab und veröffentlicht das Journal *Computational Linguistics*. Außerdem gibt es eine *International Conference on Computational Linguistics* (COLING). Das Lehrbuch von Manning und Schütze (1999) behandelt statistische Sprachverarbeitung, während Jurafsky und Martin (2008) eine umfangreiche Einführung in die Verarbeitung natürlicher Sprache bieten.

Zusammenfassung

Dieses Kapitel hat sich vor allem mit folgenden Punkten befasst:

- Probabilistische Sprachmodelle, die auf n-Grammen basieren, decken eine überraschende Menge von Informationen über eine Sprache auf. Sie schneiden bei verschiedenen Aufgaben wie Spracherkennung, Rechtschreibkorrektur, Genreklassifizierung und Erkennung benannter Entitäten gut ab.
- Da diese Sprachmodelle Millionen von Merkmalen umfassen können, ist eine Vorverarbeitung der Daten unerlässlich, um Rauschen zu verringern.
- **Textklassifizierung** lässt sich mit naiven Bayesschen n-Gramm-Modellen oder jedem der bisher besprochenen Klassifizierungsalgorithmen realisieren. Klassifizierung kann auch als Problem in der Datenkomprimierung angesehen werden.
- **Informationsabrufsysteme** (Information Retrieval, IR) verwenden ein sehr einfaches Sprachmodell, das auf Bag of Words („Sack voller Wörter") beruht, sich aber trotzdem in Bezug auf **Recall** und **Genauigkeit** bei sehr großen Textkorpora gut verhält. Bei Webkorpora verbessern Algorithmen der Link-Analyse die Leistung.
- Das **Beantworten von Fragen** (Question Answering, QA) kann durch einen Ansatz basierend auf dem Informationsabruf behandelt werden, und zwar für Fragen, die im Korpus mehrere Antworten enthalten. Wenn im Korpus mehr Antworten verfügbar sind, können wir Techniken nutzen, die Genauigkeit gegenüber Recall bevorzugen.
- **Informationsextraktionssysteme** verwenden ein komplexeres Modell, das begrenzte Konzepte der Syntax und Semantik in Form von Vorlagen einschließt. Sie lassen sich aus endlichen Automaten, HMMs oder Conditional Random Fields (CRFs) aufbauen und anhand von Beispielen lernen.
- Beim Aufbau eines statistischen Sprachsystems verwendet man am besten ein Modell, das von verfügbaren **Daten** profitieren kann, selbst wenn das Modell stark vereinfacht zu sein scheint.

Übungen zu Kapitel 22

Lösungshinweise

1 Schreiben Sie ein Programm zur **Segmentierung** von Wörtern ohne Leerzeichen. Für eine Zeichenfolge wie zum Beispiel die URL „thelongestlistofthelongeststuffatthelongestdomainnameatlonglast.com" ist eine Liste der Wortkomponenten zurückzugeben: ["the", "longest", "list", ...]. Nützlich ist diese Aufgabe für das Parsen von URLs, zur Rechtschreibkorrektur bei zusammengeschriebenenwörtern und für Sprachen wie etwa Chinesisch, bei denen es zwischen Wörtern keine Leerzeichen gibt. Gelöst werden kann diese Aufgabe mit einem Unigramm- oder Bigramm-Wortmodell und einem Algorithmus der dynamischen Programmierung ähnlich dem Viterbi-Algorithmus.

2 Das *Zipfsche Gesetz* der Wortverteilung besagt: Nimm einen großen Textkorpus, zähle die Häufigkeit jedes Wortes im Korpus und ordne dann diese Häufigkeiten in fallender Reihenfolge an. Sei f_1 die I-te größte Häufigkeit; d.h. f_1 ist die Häufigkeit des häufigsten Wortes (normalerweise "the"), f_2 die Häufigkeit des zweihäufigsten Wortes usw. Entsprechend dem Zipfschen Gesetz ist nun f_I ungefähr gleich α/I für eine bestimmte Konstante α. Außer für sehr kleine und sehr große Werte von I ist das Gesetz sehr genau.

Wählen Sie einen Korpus von mindestens 20.000 Wörtern von Online-Text aus und überprüfen Sie das Zipfsche Gesetz experimentell. Definieren Sie ein Fehlermaß und ermitteln Sie den Wert von α, bei dem das Zipfsche Gesetz am besten mit Ihren experimentellen Daten übereinstimmt. Stellen Sie f_I über I und α/I über I in einem doppeltlogarithmischen Diagramm dar. (In einem doppeltlogarithmischen Diagramm ist die Funktion α/I eine Gerade.) Achten Sie bei Durchführung des Experiments darauf, Formatierungselemente (z.B. HTML-Tags) zu entfernen und die Groß-/Kleinschreibung zu normalisieren.

3 (Nach Jurafsky und Martin (2000).) In dieser Übung entwickeln Sie einen Klassifikator für die Urheberschaft: Für einen gegebenen Text sagt der Klassifikator voraus, welcher von zwei Kandidatenautoren den Text geschrieben hat. Nehmen Sie Textstichproben von zwei verschiedenen Autoren. Trennen Sie sie in Trainings- und Testmengen. Trainieren Sie dann ein Sprachmodell anhand der Trainingsmenge. Wählen Sie die zu verwendenden Merkmale aus; am einfachsten sind n-Gramme von Wörtern oder Buchstaben, doch können Sie auch zusätzliche Merkmale hinzufügen, die Ihnen hilfreich erscheinen. Berechnen Sie dann die Wahrscheinlichkeit des Textes für jedes Sprachmodell und wählen Sie das wahrscheinlichste Modell aus. Bewerten Sie die Genauigkeit dieser Technik. Wie ändert sich die Genauigkeit, wenn Sie die Featuremenge verändern? Bei diesem Teilgebiet der Linguistik handelt es sich um die sogenannte **Stilometrie**; zu ihren Erfolgen gehören die Identifizierung des Autors der strittigen *Federalist Papers* (Mosteller und Wallace, 1964) und einiger umstrittener Arbeiten von Shakespeare (Hope, 1994). Khmelev und Tweedie (2001) erzielen gute Ergebnisse mit einem einfachen Bigramm-Buchstabenmodell.

4 Diese Übung betrifft die Klassifizierung von Spam-E-Mail. Erstellen Sie einen Korpus von Spam-E-Mail und einen für Nicht-Spam-E-Mail. Untersuchen Sie jeden Korpus und entscheiden Sie, welche Merkmale für die Klassifizierung nützlich erscheinen: Unigramm-Wörter? Bigramme? Nachrichtenlänge, Absender, Ankunftszeit? Trainieren Sie dann einen Klassifizierungsalgorithmus (Entscheidungsbaum, naives Bayes, SVM, logistische Regression oder einen anderen Algorithmus Ihrer Wahl) anhand einer Trainingsmenge und ermitteln Sie dann seine Genauigkeit für eine Testmenge.

5 Erstellen Sie eine Testmenge von zehn Anfragen und stellen Sie sie drei großen Websuchmaschinen. Bewerten Sie die einzelnen Suchmaschinen hinsichtlich Genauigkeit für 1, 3 und 10 Dokumente. Können Sie die Unterschiede zwischen den Maschinen erklären?

6 Schätzen Sie den Speicherbedarf für den Index eines Korpus aus 100 Milliarden Webseiten. Geben Sie die Annahmen an, die Sie getroffen haben.

7 Schreiben Sie einen regulären Ausdruck oder ein kurzes Programm, um Firmennamen zu extrahieren. Testen Sie das System auf einem Korpus von Wirtschaftsmeldungen. Geben Sie Recall und Genauigkeit Ihres Systems an.

8 Betrachten Sie das Problem, die Qualität eines Informationsextraktionssystems zu bewerten, das eine Rangfolgeliste von Antworten zurückgibt (wie es bei den meisten Websuchmaschinen der Fall ist). Das geeignete Qualitätsmaß hängt vom unterstellten Modell ab, was der Suchende zu erreichen versucht und welche Strategie er anwendet. Schlagen Sie für jedes der folgenden Modelle ein entsprechendes numerisches Maß vor:

a. Der Suchende sieht sich die zwanzig ersten zurückgegebenen Antworten an, wobei er möglichst viele relevante Informationen erhalten möchte.

b. Der Suchende benötigt lediglich ein relevantes Dokument und geht die Liste durch, bis er den ersten Treffer findet.

c. Der Suchende stellt eine recht enge Abfrage und ist in der Lage, alle abgerufenen Antworten zu untersuchen. Er möchte sicher sein, dass er in der Dokumentsammlung alles gesehen hat, was für seine Abfrage relevant ist. (Zum Beispiel möchte ein Rechtsanwalt sicher sein, dass er alle relevanten Präzedenzfälle gefunden hat, und ist bereit, erhebliche Mittel dafür aufzuwenden.)

d. Der Suchende benötigt lediglich ein Dokument, das für die Abfrage relevant ist, und kann es sich leisten, einem Forschungsassistenten für das Durchsehen der Ergebnisse eine Arbeitsstunde zu bezahlen. Der Assistent kann etwa 100 abgerufene Dokumente in einer Stunde begutachten. Er fordert vom Suchenden den vollen Stundenlohn für eine angefangene Stunde, wobei es egal ist, ob er das Dokument sofort am Beginn oder erst am Ende der Stunde findet.

e. Der Suchende sieht alle Antworten durch. Die Untersuchung eines Dokumentes verursacht Kosten von A€; wird ein relevantes Dokument gefunden, hat dies den Wert B€; wird ein relevantes Dokument übersehen, entstehen C€ Kosten für jedes nicht gefundene relevante Dokument.

f. Der Suchende möchte so viele relevante Dokumente wie möglich sammeln, doch benötigt er eine stete Ermunterung. Er sieht die Dokumente der Reihe nach durch. Wenn die bisher angesehenen Dokumente vorwiegend gut sind, fährt er fort; andernfalls hört er mit der Suche auf.

Natürliche Sprache für die Kommunikation

23

ÜBERBLICK

In diesem Kapitel erfahren Sie, wie Menschen miteinander in natürlicher Sprache kommunizieren und wie sich Computeragenten in die Konversation einbeziehen lassen.

Kommunikation ist der bewusste Austausch von Informationen – realisiert durch Erzeugung und Wahrnehmung von **Signalen**, die aus einem gemeinsamen System konventioneller Signale stammen. Die meisten Tiere verwenden Signale, um wichtige Nachrichten zu übermitteln: Hier gibt es Futter, ein Feind naht, Annäherung, Rückzug, Paarung. In einer partiell beobachtbaren Welt kann die Kommunikation den Agenten helfen, erfolgreich zu handeln, weil sie Informationen lernen können, die von anderen beobachtet oder abgeleitet wurden. Menschen sind die gesprächigste aller Spezies und wenn Computeragenten hilfreich sein sollen, müssen sie die Sprache sprechen lernen. In diesem Kapitel sehen wir uns die Sprachmodelle zur Kommunikation an. Modelle mit dem Ziel, eine Konversation im Detail zu verstehen, müssen zwangsläufig komplexer sein als die einfachen Modelle, die beispielsweise für die Spam-Klassifizierung gedacht sind. Wir beginnen mit grammatikalischen Modellen der Phrasenstruktur von Sätzen, ergänzen das Modell mit semantischen Elementen und wenden es dann auf maschinelle Übersetzung und Spracherkennung an.

23.1 Phrasenstrukturgrammatiken

Die *n*-Gramm-Sprachmodelle von *Kapitel 22* beruhen auf Sequenzen von Wörtern. Problematisch bei diesen Modellen ist ihre **geringe Datendichte** – mit einem Vokabular von beispielsweise 10^5 Wörtern sind 10^{15} Trigramm-Möglichkeiten zu bewerten, sodass selbst ein Korpus von einer Billion Wörtern nicht in der Lage sein wird, für alle Trigramme zuverlässige Bewertungen bereitzustellen. Das Problem der geringen Datendichte lässt sich über Verallgemeinerung lösen. Aus der Tatsache, dass „black dog“ häufiger vorkommt als „dog black“, und ähnlichen Beobachtungen können wir die Verallgemeinerung bilden, dass im Englischen die Adjektive eher *vor* den Substantiven stehen (während es sich im Französischen umgekehrt verhält: „chien noir“ ist häufiger). Natürlich gibt es Ausnahmen. So ist „galore“ (massenweise) ein Adjektiv, das dem von ihm modifizierten Substantiv folgt. Trotz der Ausnahmen stellt das Konzept einer **lexikalischen Kategorie** (auch als **Wortart** bezeichnet) wie zum Beispiel *Substantiv* oder *Adjektiv* eine nützliche Verallgemeinerung dar – nützlich an sich, doch mehr noch, wenn wir lexikalische Kategorien verketten, um **syntaktische Kategorien** zu bilden, wie zum Beispiel *Substantivphrase* oder *Verbphrase*, und diese syntaktischen Kategorien zu hierarchischen Strukturen kombinieren, die die **Phrasenstruktur** der Sätze darstellen: verschachtelte Phrasen, die jeweils mit einer Kategorie markiert sind.

Generative Kapazität

Grammatikalische Formalismen können nach ihrer **generativen Kapazität** klassifiziert werden: die Menge der Sprachen, die sie repräsentieren können. Chomsky (1957) beschreibt vier Klassen grammatikalischer Formalismen, die sich nur in der Form ihrer Umschreibungsregeln unterscheiden. Die Klassen können in einer Hierarchie angeordnet werden, wobei jede Klasse verwendet werden kann, um alle Sprachen zu beschreiben, die von einer weniger leistungsfähigen Klasse beschrieben werden können, ebenso wie einige zusätzliche Sprachen. Hier listen wir die Hierarchie auf, wobei die leistungsfähigste Klasse zuerst angeführt wird:

Rekursiv aufzählbare Grammatiken verwenden unbeschränkte Regeln: Beide Seiten der Umschreibungsregeln können beliebig viele terminale und nichtterminale Symbole enthalten, wie etwa in der Regel $A\ B\ C \rightarrow D\ E$. Diese Grammatiken sind im Hinblick auf ihre Ausdruckskraft äquivalent mit Turing-Maschinen.

Kontextsensitive Grammatiken sind nur darauf beschränkt, dass auf der rechten Seite mindestens so viele Symbole erscheinen müssen wie auf der linken Seite. Der Name „kontextsensitiv" stammt von der Tatsache, dass eine Regel wie beispielsweise $A\ X\ B \rightarrow A\ Y\ B$ besagt, dass ein X im Kontext eines vorausgehenden A und nachfolgenden B als Y umgeschrieben werden kann. Kontextsensitive Grammatiken können Sprachen repräsentieren wie beispielsweise $a^n b^n c^n$ (eine Folge von n Kopien von a gefolgt von derselben Anzahl an Kopien von b und c).

In **kontextfreien Grammatiken** (oder **CFG**s) besteht die linke Seite aus einem einzigen nichtterminalen Zeichen. Damit erlaubt jede Regel die Umschreibung des nichtterminalen Zeichens als die rechte Seite *in jedem beliebigen* Kontext. CFGs sind bekannt für Grammatiken für natürliche Sprachen sowie Programmiersprachen, obwohl nun allgemein anerkannt ist, dass zumindest einige natürliche Sprachen Konstrukte aufweisen, die nicht kontextfrei sind (Pullum, 1991). Kontextfreie Grammatiken können $a^n b^n$, nicht aber $a^n b^n c^n$ repräsentieren.

Reguläre Grammatiken stellen die am meisten beschränkte Klasse dar. Jede Regel hat ein einziges nichtterminales Zeichen auf der linken Seite und ein terminales Symbol optional gefolgt von einem nichtterminalen Zeichen auf der rechten Seite. Reguläre Grammatiken sind in der Leistungsstärke äquivalent mit endlichen Automaten. Sie sind schlecht für Programmiersprachen geeignet, weil sie keine Konstrukte wie beispielsweise symmetrische öffnende und schließende Klammern repräsentieren können (eine Variante der Sprache $a^n b^n$). Das Ähnlichste, was sie erzeugen können, sind die Repräsentationen von $a^* b^*$, einer Folge von beliebig vielen a gefolgt von beliebig vielen b.

Die weiter oben in der Hierarchie angeordneten Grammatiken besitzen eine größere Ausdruckskraft, aber die Algorithmen für den Umgang mit ihnen sind weniger effizient. Bis Mitte der 1980er Jahre konzentrierten sich die Linguisten auf kontextfreie und kontextsensitive Sprachen. Seitdem ist das Interesse an regulären Grammatiken wiedererwacht, was sich aus den Erfordernissen ergibt, Giga- oder Terabyte an Online-Text sehr schnell zu verarbeiten und daraus zu lernen, selbst wenn dies auf Kosten einer weniger vollständigen Analyse geht. Fernando Pereira sagte dazu: „Je älter ich werde, desto weiter gelange ich in der Chomsky-Hierarchie nach unten." Um zu verstehen, was er damit gemeint hat, vergleichen Sie Pereira und Warren (1980) mit Mohri, Pereira und Riley (2002) (und beachten Sie, dass alle drei Autoren heute an großen Textkorpora bei Google arbeiten).

Es gibt viele konkurrierende Sprachmodelle, die auf dem Konzept der Phrasenstruktur basieren; wir beschreiben hier ein bekanntes Modell, die sogenannte **probabilistische kontextfreie Grammatik** (PCFG, Probabilistic Context-Free Grammar)[1]. Eine **Grammatik** ist eine Sammlung von Regeln, die eine **Sprache** als Menge zulässiger Wortfolgen definieren. Der Kasten „Generative Kapazität“ hat den Begriff „kontextfrei“ erläutert und „probabilistisch“ heißt, dass die Grammatik jeder Folge eine Wahrscheinlichkeit zuweist. Eine PCFG-Regel sieht wie folgt aus:

$$\begin{aligned} VP &\rightarrow \quad Verb\ [0{,}70] \\ &\quad\ \mid \quad VP\ NP\ [0{,}30]. \end{aligned}$$

Hier sind *VP* (*Verbphrase*) und *NP* (*Substantivphrase*) **Nichtterminalsymbole**. Die Grammatik verweist auch auf eigentliche Wörter, die sogenannten **Terminalsymbole**. Diese Regel besagt, dass eine Verbphrase mit einer Wahrscheinlichkeit von 0,70 ausschließlich aus einem Verb besteht und sie mit einer Wahrscheinlichkeit von 0,30 eine *VP* gefolgt von einer *NP* ist. Anhang B beschreibt nichtprobabilistische kontextfreie Grammatiken.

Wir definieren nun eine Grammatik für einen kleinen englischen Textabschnitt, der geeignet ist für die Kommunikation zwischen Agenten, die die Wumpus-Welt erkunden. Wir bezeichnen diese Sprache als ε_0. Spätere Abschnitte zeigen Verbesserungen von ε_0, um sie dem realen Englisch besser anzupassen. Es ist unwahrscheinlich, dass wir je eine vollständige Grammatik für das Englische ableiten können, weil man sich nicht darüber einig ist, was gültiges Englisch eigentlich ist.

23.1.1 Das Lexikon von ε_0

Als Erstes definieren wir das **Lexikon**, eine Liste der erlaubten Wörter. Die Wörter sind nach lexikalischen Kategorien gruppiert, die Benutzern von Wörterbüchern geläufig sind: Substantive, Pronomen und Namen, die Dinge bezeichnen, Verben, die Ereignisse bezeichnen, Adjektive, die Substantive modifizieren, und Adverbien, die Verben modifizieren. Außerdem gibt es Funktionswörter: Artikel (wie beispielsweise *the*), Präpositionen (*in*) und Konjunktionen (*and*). ▶ Abbildung 23.1 zeigt ein kleines (englisches) Lexikon für die Sprache ε_0.

Jede der Kategorien endet mit ..., um damit anzuzeigen, dass es noch weitere Wörter in dieser Kategorie gibt. Für Substantive, Namen, Verben, Adjektive und Adverbien ist es grundsätzlich nicht möglich, alle Wörter aufzulisten. Es gibt nicht nur Zehntausende von Mitgliedern in jeder Klasse, sondern es werden auch ständig neue hinzugefügt – wie beispielsweise *iPod* oder *Biodiesel*. Diese fünf Kategorien werden auch als **offene Klassen** bezeichnet. Für die anderen Kategorien (Pronomen, Relativpronomen, Artikel, Präpositionen und Konjunktionen) könnten wir mit etwas Mehraufwand alle Wörter auflisten. Diese sogenannten **geschlossenen Klassen** enthalten eine kleine Anzahl an Wörtern (ein paar oder ein paar Dutzend). Geschlossene Klassen ändern sich im Verlauf von Jahrhunderten, nicht innerhalb von Monaten. Beispielsweise waren im Englischen „thee“ und „thou“ übliche Pronomen im 17. Jahrhundert, wäh-

1 PCFGs werden auch als stochastische kontextfreie Grammatiken (SCFGs) bezeichnet.

rend sie im 19. Jahrhundert schon veraltet waren und heute nur noch in Gedichten oder einigen regionalen Dialekten vorkommen.

Substantiv	→	**stench** [0,05] | **breeze** [0,10] | **wumpus** [0,15] | **pits** [0,05] | . . .
Verb	→	**is** [0,10] | **feel** [0,10] | **smells** [0,10] | **stinks** [0,05] | . . .
Adjektiv	→	**right** [0,10] | **dead** [0,05] | **smelly** [0,02] | **breezy** [0,02] . . .
Adverb	→	**here** [0,05] | **ahead** [0,05] | **nearby** [0,02] | . . .
Pronomen	→	**me** [0,10] | **you** [0,03] | **I** [0,10] | **it** [0,10] | . . .
RelPro	→	**that** [0,40] | **which** [0,15] | **who** [0,20] | **whom** [0,02] _ . . .
Name	→	**John** [0,01] | **Mary** [0,01] | **Boston** [0,01] | . . .
Artikel	→	**the** [0,40] | **a** [0,30] | **an** [0,10] | **every** [0,05] | . . .
Präposition	→	**to** [0,20] | **in** [0,10] | **on** [0,05] | **near** [0,10] | . . .
Konjunktion	→	**and** [0,50] | **or** [0,10] | **but** [0,20] | **yet** [0,02] _ . . .
Ziffer	→	**0** [0,20] | **1** [0,20] | **2** [0,20] | **3** [0,20] | **4** [0,20] | . . .

Abbildung 23.1: Das Lexikon für ε_0. *RelPro* steht für Relativpronomen.

23.1.2 Die Grammatik von ε_0

Im nächsten Schritt kombinieren wir die Wörter zu Phrasen. ▶ Abbildung 23.2 zeigt eine Grammatik für ε_0 mit Regeln für jede der sechs syntaktischen Kategorien und einem Beispiel für jede Umschreibungsregel.[2] In ▶ Abbildung 23.3 ist ein **Syntaxbaum** (engl. Parse Tree) für den Satz „Every wumpus smells." dargestellt. Der Syntaxbaum liefert einen konstruktiven Beweis, dass die Wortfolge tatsächlich ein Satz gemäß den Regeln von ε_0 ist. Die ε_0-Grammatik generiert ein breites Spektrum von englischen Sätzen wie zum Beispiel:

John is in the pit

The wumpus that stinks is in 2,2

Mary is in Boston and John stinks

Leider erzeugt die Grammatik **zu viel** (**Übergenerierung**): Sie erzeugt Sätze, die grammatikalisch nicht korrekt sind, wie beispielsweise „Me go Boston" oder „I smell pits wumpus John". Außerdem erzeugt sie auch **zu wenig** (**Untergenerierung**): Es gibt viele englische Sätze, die sie zurückweist, wie beispielsweise „I think the wumpus is smelly". Wir zeigen später, wie sich eine bessere Grammatik lernen lässt; fürs Erste konzentrieren wir uns darauf, was wir mit der vorhandenen Grammatik tun können.

2 Ein Relativsatz folgt einer Substantivphrase und modifiziert sie. Er besteht aus einem Relativpronomen (wie z.B. „who" oder „that") gefolgt von einer Verbphrase. Ein Beispiel für einen Relativsatz ist *that stinks* in „The wumpus *that stinks* is in 2 2".

ε_0:	*S*	→	*NP VP*	[0,90]	I + feel a breeze
		|	*S Konjunktion S*	[0,10]	I feel a breeze + and + It stinks
	NP	→	*Pronomen*	[0,30]	I
		|	*Name*	[0,10]	John
		|	*Substantiv*	[0,10]	pits
		|	*Artikel Substantiv*	[0,25]	the + wumpus
		|	*Artikel AdjL Substantiv*	[0,05]	the + smelly dead + wumpus
		|	*Ziffer Ziffer*	[0,05]	3 4
		|	*NP PP*	[0,10]	the wumpus + in 1 3
		|	*NP RelS*	[0,05]	the wumpus + that is smelly
	VP	→	*Verb*	[0,40]	stinks
		|	*VP NP*	[0,35]	feel + a breeze
		|	*VP Adjektiv*	[0,05]	smells + dead
		|	*VP PP*	[0,10]	is + in 1 3
		|	*VP Adverb*	[0,10]	go + ahead
	AdjL	→	*Adjektiv*	[0,80]	smelly
		|	*Adjektiv AdjL*	[0,20]	smelly + dead
	PP	→	*Präposition NP*	[1,00]	to + the east
	RelS	→	*RelPro VP*	[1,00]	that + is smelly

Abbildung 23.2: Die Grammatik für ε_0 mit Beispielphrasen für jede Regel. Die syntaktischen Kategorien sind Satz (S), Substantivphrase (NP), Verbphrase (VP), Adjektivliste ($AdjL$), Präpositionsphrase (PP) und Relativsatz ($RelS$).

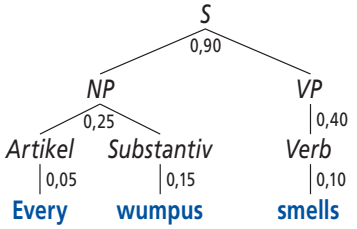


Abbildung 23.3: Syntaxbaum für den Satz „Every wumpus smells" entsprechend der Grammatik ε_0. Jeder innere Knoten des Baumes ist mit seiner Wahrscheinlichkeit beschriftet. Die Wahrscheinlichkeit des Baums als Ganzes beträgt $0{,}90 \times 0{,}25 \times 0{,}05 \times 0{,}15 \times 0{,}40 \times 0{,}10 = 0{,}0000675$. Da dieser Baum den einzigen Parsing-Verlauf des Satzes darstellt, ist diese Zahl gleichzeitig die Wahrscheinlichkeit des Satzes. Der Baum lässt sich auch in linearer Form als [S[NP[*Artikel* **every**][*Substantiv* **wumpus**]][VP[*Verb* **smells**]]] schreiben.

23.2 Syntaktische Analyse (Parsing)

Unter **Parsing** versteht man das Analysieren einer Folge von Wörtern, um deren Phrasenstruktur aufzudecken. Wie ▶ Abbildung 23.4 zeigt, können wir einen Baum, dessen Blätter die Wörter verkörpern, beim Symbol S beginnend von oben nach unten (Top-down) durchsuchen oder bei einem Baum, der in einem S gipfelt, mit den Wörtern beginnen und von unten nach oben (Bottom-up) suchen. Allerdings können

sowohl das Top-down- als auch das Bottom-up-Parsing ineffizient sein, weil sie unter Umständen bestimmte Bereiche des Suchraumes wiederholt abarbeiten, die in Sackgassen münden. Betrachten Sie die beiden folgenden Sätze:

> Have the students in section 2 of Computer Science 101 take the exam.
>
> Have the students in section 2 of Computer Science 101 taken the exam?

Obwohl die ersten zehn Wörter gleich sind, haben diese Sätze sehr unterschiedliche Parsing-Verläufe, weil es sich bei dem ersten um einen Befehl, bei dem zweiten um eine Frage handelt. Ein Algorithmus, der von links nach rechts vorgeht, müsste raten, ob das erste Wort Teil eines Befehles oder einer Frage ist, und kann erst dann entscheiden, ob seine Annahme richtig war, bis er mindestens das elfte Wort, *take* oder *taken*, erreicht hat. Liegt der Algorithmus mit seiner Vermutung falsch, muss er den gesamten Weg bis zum ersten Wort zurückgehen und den gesamten Satz entsprechend der anderen Interpretation erneut analysieren.

Liste der Elemente	Regel
S	
NP VP	*S* → *NP VP*
NP VP Adjektiv	*VP* → *VP Adjektiv*
NP Verb Adjektiv	*VP* → *Verb*
NP Verb **dead**	*Adjektiv* → **dead**
NP **is dead**	*Verb* → **is**
Artikel Substantiv **is dead**	*NP* → *Artikel Substantiv*
Artikel **wumpus is dead**	*Substantiv* → **wumpus**
the wumpus is dead	*Artikel* → **the**

Abbildung 23.4: Schritte, um einen Parsing-Verlauf für die Folge „The wumpus is dead" als Satz entsprechend der Grammatik ε_0 zu finden. Als Top-down-Parsing betrachtet beginnen wir mit der Liste der Elemente, die aus S besteht, vergleichen in jedem Schritt ein Element X mit einer Regel der Form ($X \rightarrow$ ...) und ersetzen X in der Liste der Elemente durch (...). Als Bottom-up-Parsing betrachtet beginnen wir mit der Liste der Elemente, die die Wörter des Satzes umfasst, vergleichen in jedem Schritt eine Folge von Token (...) in der Liste mit einer Regel der Form ($X \rightarrow$ ...) und ersetzen (...) durch X.

Tipp

Diese Quelle der Ineffizienz lässt sich mit dynamischer Programmierung vermeiden: *Wenn wir eine Teilkette analysieren, speichern wir die Ergebnisse, sodass wir sie später nicht noch einmal analysieren müssen.* Nachdem wir beispielsweise erkannt haben, dass „the students in section 2 of Computer Science 101" ein *NP* ist, können wir dieses Ergebnis in einer Datenstruktur aufzeichnen, die auch als **Chart** bezeichnet wird. Entsprechende Algorithmen werden auch als **Chart-Parser** bezeichnet. Weil wir es mit kontextfreien Grammatiken zu tun haben, kann jede Phrase, die im Kontext einer Verzweigung des Suchraumes gefunden wurde, in jedem anderen Zweig des Suchraumes genauso funktionieren. Von den vielen Typen von Chart-Parsern beschreiben wir hier eine Bottom-up-Version, die nach ihren Erfindern John Cocke, Daniel Younger und Tadeo Kasami als **CYK-Algorithmus** bezeichnet wird.

▶ Abbildung 23.5 zeigt den CYK-Algorithmus. Er verlangt eine Grammatik, bei der alle Regeln in einem von zwei sehr spezifischen Formaten vorliegen: lexikalische Regeln der

Form $X \rightarrow Wort$ und syntaktische Regeln der Form $X \rightarrow Y\ Z$. Dieses als **Chomsky-Normalform** bezeichnete Grammatikformat mag restriktiv erscheinen, ist es aber nicht: Jede kontextfreie Grammatik lässt sich in die Chomsky-Normalform automatisch transformieren. Übung 23.8 führt Sie durch den entsprechenden Prozess.

```
function CYK-PARSE(words, grammar) returns P, eine Tabelle von
                                                Wahrscheinlichkeiten
  N ← LENGTH(words)
  M ← die Anzahl der Nichtterminalsymbole in grammar
  P ← ein Array der Größe [M, N, N], anfangs alle 0
  /* Lexikalische Regeln für jedes Wort einfügen */
  for i = 1 to N do
    for each Regel der Form (X → words_i [p]) do
      P[X, i , 1] ← p
  /* Erste und zweite Teile der rechten Seiten von Regeln kombinieren
      von kurz nach lang */
  for length = 2 to N do
    for start = 1 to N - length + 1 do
      for len1 = 1 to Length - 1 do
        len2 ← length - len1
        for each Regel der Form (X → Y Z [p]) do
          P[X, start, length] MAX(P[X, start, length],
          P[Y, start, len1 ] × P[Z, start + len1, len2 ] × p)
  return P
```

Abbildung 23.5: Der CYK-Algorithmus zum Parsing. Für eine gegebene Sequenz von Wörtern findet er die wahrscheinlichste Ableitung für die gesamte Sequenz.

Der CYK-Algorithmus verwendet einen Raum von $O(n^2m)$ für die P-Tabelle, wobei n die Anzahl der Wörter im Satz und m die Anzahl der Nichtterminalsymbole in der Grammatik angeben, bei einem Zeitbedarf von $O(n^3m)$. (Da m für eine bestimmte Grammatik konstant ist, wird dies üblicherweise als $O(n^3)$ beschrieben.) Für allgemeine kontextfreie Grammatiken schneidet kein Algorithmus besser ab, für restriktivere Grammatiken gibt es allerdings schnellere Algorithmen. In Anbetracht der Tatsache, dass die Anzahl der Syntaxbäume für einen Satz exponentiell wachsen kann, ist es in der Tat ein großes Kunststück, dass der Algorithmus lediglich eine Zeit von $O(n^3)$ benötigt. Der Satz

Fall leaves fall and spring leaves spring.

ist mehrdeutig, da jedes Wort (außer „and“) als Substantiv oder als Verb betrachtet werden kann sowie „fall“ und „spring“ auch Adjektive sein können. (Zum Beispiel ist eine Bedeutung von „Fall leaves fall“ äquivalent mit „Autumn abandons autumn“, d.h. „Herbst verlässt Herbst“.) Mit ε_0 hat der Satz vier Parsing-Möglichkeiten:

[S [S [NP Fall leaves] fall] and [S [NP spring leaves] spring]
[S [S [NP Fall leaves] fall] and [S spring [VP leaves spring]]
[S [S Fall [VP leaves fall]] and [S [NP spring leaves] spring]
[S [S Fall [VP leaves fall] and [S spring [VP leaves spring]]

Hätten wir c zweifach mehrdeutig miteinander verbundene Untersätze, so hätten wir 2^c Möglichkeiten, Parsing-Verläufe für die Untersätze auszuwählen.[3] Wie verarbeitet

3 Es gäbe auch $O(c!)$ Mehrdeutigkeit, wie die Komponenten miteinander verknüpft werden – z.B. (X und (Y und Z)) im Vergleich zu ((X und Y) und Z). Das ist jedoch eine andere Geschichte, die Ihnen am besten Church und Patil (1982) erzählen.

der CYK-Algorithmus diese 2^c Syntaxbäume in einer Zeit von $O(c^3)$? Indem er gar nicht alle Syntaxbäume untersucht; er muss lediglich die Wahrscheinlichkeit des wahrscheinlichsten Baumes berechnen. Die Unterbäume werden alle in der P-Tabelle dargestellt und mit etwas zusätzlichem Aufwand könnten wir sie alle aufzählen (in exponentieller Zeit), doch liegt die Eleganz des CYK-Algorithmus darin, dass wir sie nicht aufzuzählen brauchen, außer wenn wir es möchten.

In der Praxis sind wir normalerweise nicht an allen Parsing-Möglichkeiten interessiert, sondern nur an der besten oder an einigen der besten. Prinzipiell definiert der CYK-Algorithmus den vollständigen Zustandsraum, wie er durch den Operator „Grammatikregel anwenden“ definiert wird. Es ist möglich, lediglich einen Teil davon mithilfe der A*-Suche zu durchsuchen. Jeder Zustand in diesem Raum ist eine Liste von Elementen (Wörtern oder Kategorien), wie in der Bottom-up-Parsing-Tabelle (Abbildung 23.4) zu sehen. Der Ausgangszustand ist eine Liste von Wörtern und ein Zielzustand ist das einzelne Element S. Die Kosten eines Zustandes entsprechen dem Kehrwert seiner Wahrscheinlichkeit, wie sie durch die bisher angewandten Regeln definiert wird. Es gibt verschiedene Heuristiken, um den verbleibenden Abstand zum Ziel abzuschätzen; die beste Heuristik stammt aus dem Maschinenlernen, das auf einen Korpus von Sätzen angewandt wird. Beim A*-Algorithmus brauchen wir nicht den gesamten Zustandsraum zu durchsuchen und der erste gefundene Parsing-Verlauf ist garantiert der wahrscheinlichste.

23.2.1 Wahrscheinlichkeiten für PCFGs lernen

Ein PCFG umfasst viele Regeln, wobei jeder Regel eine Wahrscheinlichkeit zugeordnet ist. Dies legt nahe, dass das **Lernen** der Grammatik anhand von Daten besser sein könnte als ein Ansatz, der auf Knowledge Engineering beruht. Lernen ist am einfachsten, wenn wir über einen Korpus korrekt analysierter (geparster) Sätze verfügen – häufig als **Baumbank** (engl. **Treebank**) bezeichnet. Die bekannteste ist die Penn-Baumbank (Marcus et al., 1993). Sie besteht aus 3 Millionen Wörtern, die teilweise mit Sprach- und Syntaxbaumstruktur annotiert sind. Die Bank wurde manuell mit Unterstützung von automatisierten Tools erstellt. ▶ Abbildung 23.6 zeigt einen annotierten Baum aus der Penn-Baumbank.

```
[[S [NP-SBJ-2 Her eyes]
    [VP were
        [VP glazed
            [NP *-2]
            [SBAR-ADV as if
                [S [NP-SBJ she]
                  [VP did n't
                    [VP [VP hear [NP *-1]]
                        or
                        [VP [ADVP even] see [NP *-1]]
                        [NP-1 him]]]]]]]]
.]
```

Abbildung 23.6: Annotierter Baum für den Satz „Her eyes were glazed as if she didn't hear or even see him.“ aus der Penn-Baumbank. Diese Grammatik unterscheidet zwischen einer Objektsubstantivphrase (NP) und einer Subjektsubstantivphrase ($NP-SBJ$). Außerdem fällt ein grammatikalisches Phänomen auf, das wir bislang noch nicht behandelt haben: die Verschiebung einer Phrase von einem Teil des Baumes in einen anderen. Dieser Baum analysiert die Phrase „hear or even see him“. Sie besteht aus den beiden Konstituenten-VPs, [VP hear [$NP*-1$]] und [VP [$ADVP$ even] see [$NP*-1$]]. Das fehlende Objekt in diesen Phrasen ist mit $*-1$ gekennzeichnet, was auf das NP verweist, das an anderer Stelle im Baum mit [$NP-1$ him] beschriftet ist.

Für einen Korpus von Bäumen können wir eine PCFG erzeugen, indem wir einfach zählen (und glätten). Im obigen Beispiel gibt es zwei Knoten der Form $[S\ [NP\ \ldots][VP\ \ldots]]$. Wir könnten diese und alle anderen Teilbäume mit der Wurzel S im Korpus zählen. Wenn es 100.000 S-Knoten gibt, von denen 60.000 diese Form haben, können wir folgende Regel aufstellen:

$$S \rightarrow NP\ VP\ [0{,}60].$$

Wie sieht es nun aus, wenn keine Baumbank zur Verfügung steht, wir aber einen Korpus von unverarbeiteten, nicht annotierten Sätzen haben? Es ist trotzdem möglich, eine Grammatik aus einem derartigen Korpus zu lernen. Allerdings gestaltet sich die Angelegenheit schwieriger. Vor allem haben wir es tatsächlich mit zwei Problemen zu tun: Lernen der Struktur der Grammatikregeln und Lernen der Wahrscheinlichkeiten, die mit jeder Regel verbunden sind. (Mit der gleichen Unterscheidung haben wir es beim Lernen Bayesscher Netze zu tun.) Wir setzen voraus, dass wir die lexikalischen und syntaktischen Kategorienamen erhalten. (Andernfalls können wir einfach von Kategorien $X_1, \ldots, X_n$ ausgehen und per Kreuzvalidierung den besten Wert von n herausgreifen.) Dann können wir annehmen, dass die Grammatik jede mögliche Regel $(X \rightarrow Y\ Z)$ oder $(X \rightarrow word)$ umfasst, auch wenn viele dieser Regeln eine Wahrscheinlichkeit von 0 oder nahe 0 haben.

Dann können wir den Erwartungsmaximierungs-Ansatz (EM-Algorithmus) genau wie beim Erlernen von HMMs verwenden. Die Parameter, die wir zu lernen versuchen, sind die Regelwahrscheinlichkeiten; diese setzen wir zu Beginn auf zufällige oder einheitliche Werte. Die versteckten Variablen sind die Syntaxbäume: Wir wissen nicht, ob eine Kette von Wörtern $w_i, \ldots, w_j$ durch eine Regel $(X \rightarrow \ldots)$ generiert wird oder nicht. Der Schritt E bewertet die Wahrscheinlichkeit, dass jede Teilsequenz durch jede Regel erzeugt wird. Der Schritt M bewertet dann die Wahrscheinlichkeit jeder Regel. Die gesamte Berechnung lässt sich per dynamischer Programmierung mit dem sogenannten **Inside-Outside-Algorithmus** (in Analogie zum Vorwärts-Rückwärts-Algorithmus für HMMs) bewerkstelligen.

Interessant beim Inside-Outside-Algorithmus ist, dass er eine Grammatik aus nicht geparstem Text ableitet. Aber er hat auch mehrere Nachteile. Erstens sind die Parsing-Verläufe, die durch die hergeleiteten Grammatiken zugeordnet werden, oftmals schwer verständlich und für Linguisten unbefriedigend. Dadurch ist es schwierig, manuell erstellte Wissensdaten mit automatisiert hergeleiteten Daten zu kombinieren. Zweitens ist er recht langsam: Der Zeitbedarf beträgt $O(n^3 m^3)$, wobei n die Anzahl der Wörter in einem Satz und m die Anzahl der Grammatikkategorien angibt. Drittens ist der Raum der Wahrscheinlichkeitszuweisungen sehr groß und wie die Erfahrung zeigt, stellt das Hängenbleiben in lokalen Maxima ein ernsthaftes Problem dar. Alternativen wie zum Beispiel Simulated Annealing können dem globalen Maximum näher kommen, jedoch auf Kosten eines höheren Rechenaufwandes. Lari und Young (1990) schlussfolgern, dass Inside-Outside „für praktische Probleme rechentechnisch nicht realisierbar ist“.

Fortschritte sind allerdings möglich, wenn wir darauf verzichten, ausschließlich aus nicht geparstem Text zu lernen. Ein Ansatz dazu ist das Lernen von **Prototypen**: Der Prozess wird anfangs mit einem oder zwei Dutzend Regeln gespeist, ähnlich den Regeln in ε_1.Von diesem Punkt an lassen sich komplexere Regeln leichter erlernen und die resultierende Grammatik parst englische Sätze mit einem Recall und einer Genauigkeit von insgesamt etwa 80% (Haghighi und Klein, 2006). Ein anderer Ansatz verwendet Baumbänke, lernt aber zusätzlich zu den PCFG-Regeln direkt aus den Bracketings (den geklammerten Termen) auch Unterscheidungen, die nicht in der Baumbank erfasst sind.

So unterscheidet der Baum in Abbildung 23.6 zwischen *NP* und *NP* – *SBJ*. Der zweite Ausdruck wird für das Pronomen „she“ und der erste für das Pronomen „her“ verwendet. Mit dieser Frage beschäftigt sich *Abschnitt 23.6* ausführlicher; fürs Erste stellen wir einfach fest, dass es viele Möglichkeiten gibt, eine Kategorie wie zum Beispiel *NP* sinnvoll **aufzuteilen** – Grammatikinduktionssysteme, die Baumbänke verwenden, aber Kategorien automatisch aufteilen, liefern bessere Ergebnisse als Systeme, die an der ursprünglichen Kategoriemenge festhalten (Petrov und Klein, 2007c). Die Fehlerraten für automatisch gelernte Grammatiken liegen dennoch rund 50% höher als die für manuell konstruierte Grammatiken, wobei aber die Lücke kleiner wird.

23.2.2 Kontextfreie und Markov-Modelle vergleichen

Problematisch bei PCFGs ist, dass sie kontextfrei sind. Das heißt, dass der Unterschied zwischen P(„eat a banana“) und P(„eat a bandanna“) nur von $P(Substantiv \rightarrow$ „banana“) gegenüber $P(Substantiv \rightarrow$ „bandanna“) abhängt und nicht von der Beziehung zwischen „eat“ und den jeweiligen Objekten.

Ein Markov-Modell der Ordnung zwei oder höher liefert einen ausreichend großen Korpus, der weiß, dass „eat a banana“ wahrscheinlicher ist. Um das Beste von beiden zu erhalten, können wir ein PCFG mit einem Markov-Modell kombinieren. Beim einfachsten Ansatz bewerten wir die Wahrscheinlichkeit eines Satzes mit dem geometrischen Mittelwert der durch beide Modelle berechneten Wahrscheinlichkeiten. Dann wüssten wir, dass „eat a banana“ sowohl vom grammatikalischen als auch vom lexikalischen Standpunkt her wahrscheinlich ist. Dennoch würde das Verfahren die Beziehung zwischen „eat“ und „banana“ in „eat a slightly aging but still palatable banana“ nicht herausgreifen, weil hier die Beziehung mehr als zwei Wörter voneinander entfernt ist. Auch wenn man die Ordnung des Markov-Modells erhöht, gelangt man nicht genau zur Beziehung; wir können aber eine **lexikalisierte PCFG** verwenden, wie es der nächste Abschnitt beschreibt.

Ein anderes Problem mit PCFGs besteht darin, dass sie kürzere Sätze zu stark bevorzugen. In einem Korpus wie dem *Wall Street Journal* sind die Sätze durchschnittlich etwa 25 Wörter lang. Ein PCFG weist aber vielen kurzen Sätzen wie zum Beispiel „He slept“ eine recht hohe Wahrscheinlichkeit zu, während wir im *Journal* eher Sätze wie „It has been reported by a reliable source that the allegation that he slept is credible.“ sehen. Anscheinend sind die Phrasen im *Journal* wirklich nicht kontextfrei; stattdessen haben die Autoren eine Vorstellung von der erwarteten Satzlänge und verwenden diese Länge als lockere globale Einschränkung für ihre Sätze. Dies lässt sich in einer PCFG kaum widerspiegeln.

23.3 Erweiterte Grammatiken und semantische Interpretation

In diesem Abschnitt zeigen wir, wie sich kontextfreie Grammatiken erweitern lassen – um beispielsweise zu sagen, dass nicht jedes *NP* unabhängig vom Kontext ist, sondern vielmehr, dass bestimmte *NP*s in dem einen Kontext wahrscheinlicher auftauchen und andere in einem anderen Kontext.

23.3.1 Lexikalisierte PCFGs

Um zur Beziehung zwischen dem Verb „eat“ und den Substantiven „banana“ im Unterschied zu „bandanna“ zu gelangen, können wir eine **lexikalisierte PCFG** verwenden, in der die Wahrscheinlichkeiten für eine Regel von der Beziehung zwischen Wörtern im Syntaxbaum abhängen und nicht nur von der Nachbarschaft der Wörter in einem Satz. Natürlich können wir die Wahrscheinlichkeit nicht auf jedes Wort im Baum stützen, da wir kaum über genügend Trainingsdaten verfügen, um alle diese Wahrscheinlichkeiten abzuschätzen. Es ist nützlich, den **Kopf** (engl. **Head**) – das wichtigste Wort – einer Phrase als Konzept einzuführen. Somit ist „eat“ der Kopf der Kategorie *VP* „eat a banana“ und „banana“ ist der Kopf der *NP* „a banana“. Mit der Notation $VP(v)$ kennzeichnen wir eine Phrase mit der Kategorie *VP*, deren Kopfwort v ist. Wir sagen, die Kategorie *VP* wird mit der Kopfvariablen v **erweitert**. Die folgende **erweiterte Grammatik** beschreibt die Verb-Objekt-Beziehung:

$$\begin{array}{ll} VP(v) \rightarrow Verb(v)\ NP(n) & [P_1(v, n)] \\ VP(v) \rightarrow Verb(v) & [P_2(v)] \\ NP(n) \rightarrow Artikel(a)\ AdjL(j)\ Substantiv(n) & [P_3(n, a)] \\ Substantiv(\textbf{banana}) \rightarrow \textbf{banana} & [p_n] \\ \ldots & \ldots \end{array}$$

Hier hängt die Wahrscheinlichkeit $P_1(v, n)$ von den Kopfwörtern v und n ab. Wir würden diese Wahrscheinlichkeit auf einen relativ hohen Wert setzen, wenn v gleich „eat“ und n gleich „banana“ ist, und auf einen niedrigen Wert, wenn n gleich „bandanna“ ist. Da wir nur Köpfe betrachten, wird die Unterscheidung zwischen „eat a banana“ und „eat a rancid banana“ durch diese Wahrscheinlichkeiten nicht erfasst. Problematisch bei diesem Ansatz ist auch, dass für P_1 in einem Vokabular von beispielsweise 20.000 Substantiven und 5.000 Verben 100 Millionen Wahrscheinlichkeitsschätzungen erforderlich sind. Nur ein geringer Anteil dieser Schätzungen kann aus einem Korpus kommen; der Rest ist durch Glättung zu realisieren (siehe *Abschnitt 22.1.2*). Zum Beispiel können wir $P_1(v, n)$ für ein (v, n)-Paar, das wir nicht oft (wenn überhaupt) gesehen haben, abschätzen, wenn wir zu einem Modell zurückgehen, das nur von v abhängt. Diese objektlosen Wahrscheinlichkeiten sind dennoch sehr nützlich; sie können die Unterscheidung erfassen zwischen einem transitiven Verb wie „eat“ – das einen hohen Wert für P_1 und einen niedrigen Wert für P_2 aufweist – und einem intransitiven Verb wie „sleep“, bei dem die Verhältnisse umgekehrt sind. Es ist durchaus möglich, diese Wahrscheinlichkeiten aus einer Baumbank zu lernen.

23.3.2 Formale Definition erweiterter Grammatikregeln

Aufgrund der Komplexität erweiterter Regeln verwenden wir eine formelle Definition, um eine erweiterte Regel in einen logischen Satz übersetzen zu können. Der Satz hat dann die Form einer definiten Klausel (siehe *Abschnitt 7.5.3*), sodass das Ergebnis als **definite Klausel-Grammatik** oder **DCG** (Definite Clause Grammar) bezeichnet wird. Als Beispiel verwenden wir die Version einer Regel aus der lexikalisierten Grammatik für *NP* mit einem neuen Notationselement:

$$NP(n) \rightarrow Artikel(a)\ AdjL(j)\ Substantiv(n)\ \{Kompatibel(j, n)\}.$$

Der neue Aspekt ist hier die Notation $\{Einschränkung\}$, um eine logische Einschränkung für bestimmte Variablen zu kennzeichnen; die Regel gilt nur, wenn die Ein-

schränkung *true* ist. Das Prädikat *Kompatibel*(j, n) dient als Test, ob Adjektiv j und Substantiv n kompatibel sind; definiert würde es durch eine Folge von Zusicherungen wie zum Beispiel *Kompatibel*(**black**, **dog**). Wir können diese Grammatikregel in eine definite Klausel konvertieren, indem wir (1) die Reihenfolge der linken und rechten Seiten umkehren, (2) eine Konjunktion aller Konstituenten und Einschränkungen erzeugen, (3) eine Variable s_i zur Liste der Argumente für jeden Konstituenten hinzufügen, um die vom Konstituenten überspannte Sequenz von Wörtern darzustellen, und (4) einen Term *Append*(s_1, ...) für die Verkettung von Wörtern zur Argumentliste für die Wurzel des Baumes hinzufügen. Damit erhalten wir:

$$Artikel(a, s_1) \wedge AdjL(j, s_2) \wedge Substantiv(n, s_3) \wedge Kompatibel(j, n) \\ \Rightarrow NP(n, Append(s_1, s_2, s_3)).$$

Diese definite Klausel besagt: Ist das Prädikat *Artikel* für ein Kopfwort a und eine Zeichenfolge s_1 gleich *true* und ist *AdjL* für ein Kopfwort j und eine Folge s_2 ebenfalls *true* und ist *Substantiv* für ein Kopfwort n und eine Folge s_3 gleich *true* und sind j und n kompatibel, dann ist das Prädikat *NP* für das Kopfwort n und das Ergebnis der angefügten Zeichenfolgen s_1, s_2 und s_3 gleich *true*. Die DCG-Übersetzung lässt zwar die Wahrscheinlichkeiten weg, doch wir können sie wieder hinzufügen. Dazu erweitern wir jeden Konstituenten mit einer oder mehreren Variablen, die die Wahrscheinlichkeit des Konstituenten darstellen, und erweitern die Wurzel mit einer Variablen, die das Produkt der Konstituentenwahrscheinlichkeiten und der Regelwahrscheinlichkeit ist.

Durch die Übersetzung von Grammatikregeln in definite Klauseln können wir Parsing als logische Inferenz betrachten. Das ermöglicht es, über Sprachen und Zeichenketten auf viele unterschiedliche Arten zu schließen. Beispielsweise bedeutet es, dass wir ein Bottom-up-Parsing unter Verwendung einer Vorwärtsverkettung oder ein Top-down-Parsing unter Verwendung einer Rückwärtsverkettung durchführen können. Letztlich war das Parsing natürlicher Sprachen mit DCGs eine der ersten Anwendungen der logischen Programmiersprache Prolog (und eine ihrer Motivationen). Manchmal ist es möglich, den Vorgang in Rückwärtsrichtung für eine **Sprachgenerierung** sowie Parsing auszuführen. Wie zum Beispiel ▶ Abbildung 23.10 später in diesem Kapitel zeigt, kann man einem logischen Programm die semantische Form *Loves*(*John*, *Mary*) übergeben und die Regeln der definiten Klauseln anwenden, um

$$S(Loves(John, Mary), [\textbf{John}, \textbf{loves}, \textbf{Mary}])$$

herzuleiten.

Für kleine Demonstrationsbeispiele mag dies funktionieren, doch benötigen ernsthafte Sprachgenerierungssysteme mehr Kontrolle über den Vorgang, als sie durch die DCG-Regeln allein realisierbar ist.

23.3.3 Fallkongruenz und Subjekt-Verb-Kongruenz

Abschnitt 23.1 hat gezeigt, dass die einfache Grammatik für ε_0 zu einer Übergenerierung führt und auch Nichtsätze erzeugt, wie beispielsweise „Me smell a stench“. Um dieses Problem zu vermeiden, müsste unsere Grammatik wissen, dass „me“ als Subjekt eines Satzes kein gültiges *NP* ist. Linguisten sagen, das Pronomen „I“ ist der Subjektfall und „me“ ist der Objektfall.[4] Wir können dies berücksichtigen, wenn wir *NP*

4 Der Subjektfall wird auch als Nominativfall bezeichnet, der Objektfall als Akkusativfall. Viele Sprachen haben auch Dativfälle für Wörter in der indirekten Objektposition.

in zwei Kategorien aufteilen – zum Beispiel in NP_S und NP_O, die für Substantivphrasen in Subjekt- und Objektfall stehen. Außerdem müssten wir die Kategorie *Pronomen* in die beiden Kategorien $Pronomen_S$ (die „I" umfasst) und $Pronomen_O$ (die „me" einschließt) unterteilen. Der obere Teil von ▶ Abbildung 23.7 zeigt die Grammatik für **Fallkongruenz**; wir bezeichnen die resultierende Sprache als ε_1. Beachten Sie, dass alle *NP*-Regeln dupliziert werden müssen, einmal für NP_S und einmal für NP_O.

ε_1:	*S*	→	NP_S *VP* | ...
	NP_S	→	$Pronomen_S$ | *Name* | *Substantiv* | ...
	NP_O	→	$Pronomen_O$ | *Name* | *Substantiv* | ...
	VP	→	*VP* NP_O
	PP	→	*Präposition* NP_O
	$Pronomen_S$	→	**I** | **you** | **he** | **she** | **it** | ...
	$Pronomen_O$	→	**me** | **you** | **him** | **her** | **it** | ...
			...
ε_2:	*S(head)*	→	*NP(Sbj, pn, h) VP(pn, head)* | ...
	NP(c, pn, head)	→	*Pronomen(c, pn, head)* | *Substantiv(c, pn, head)* | ...
	VP(pn, head)	→	*VP(pn, head) NP(Obj, p, h)* | ...
	PP(head)	→	*Präposition(head) NP(Obj, pn, h)*
	Pronomen(Sbj, 1S, **I***)*	→	**I**
	Pronomen(Sbj, 1P, **we***)*	→	**we**
	Pronomen(Obj, 1S, **me***)*	→	**me**
	Pronomen(Obj, 3P, **them***)*	→	**them**
			...

Abbildung 23.7: Oben: Teil einer Grammatik für die Sprache ε_1, die Subjektiv- und Objektivfälle in Substantivphrasen verarbeitet und somit hinsichtlich einer Übergenerierung besser als ε_0 abschneidet. Die mit ε_0 identischen Abschnitte wurden ausgelassen. Unten: Teil einer erweiterten Grammatik für ε_2 mit drei Erweiterungen: Fallkongruenz, Subjekt-Verb-Kongruenz und Kopfwort. Bei *Sbj*, *Obj*, *1S*, *1P* und *3P* handelt es sich um Konstanten. Klein geschriebene Namen bezeichnen Variablen.

Leider erzeugt ε_1 immer noch zu viele Sätze. Englisch und viele andere Sprachen fordern eine **Subjekt-Verb-Kongruenz**. Wenn beispielsweise „I" das Subjekt ist, dann ist „I smell" grammatisch korrekt, aber „I smells" ist es nicht. Wenn „it" das Subjekt ist, erhalten wir das Umgekehrte. Im Englischen sind die Kongruenzunterscheidungen minimal: Die meisten Verben haben eine Form für Subjekte der dritten Person Singular (he, she oder it) sowie eine zweite Form für alle anderen Kombinationen aus Personen und Anzahl. Es gibt eine Ausnahme: Das Verb „to be" besitzt drei Formen, „I am/ you are/ he is". Somit unterteilt die eine Unterscheidung (der Fall) *NP* zweifach, eine andere Unterscheidung (Person und Anzahl) *NP* dreifach und wenn wir weitere Unterscheidungen freilegen, erhalten wir schließlich beim Ansatz von ε_1 eine exponentielle Anzahl von indizierten *NP*-Formen. Erweiterungen sind ein besseres Konzept: Sie können eine exponentielle Anzahl von Formen als einzelne Regel darstellen.

Der untere Teil von Abbildung 23.7 zeigt (ausschnittsweise) eine erweiterte Grammatik für die Sprache ε_2, die Fallkongruenz, Subjekt-Verb-Kongruenz und Kopfwörter verarbeitet. Es gibt nur eine *NP*-Kategorie, doch besitzt *NP*(c, *pn*, *head*) drei Erweiterungen: c ist ein Parameter für den Fall, *pn* ein Parameter für Person und Anzahl und *head* ein Parameter für das Kopfwort der Phrase. Die anderen Kategorien werden ebenfalls mit Köpfen und anderen Argumenten erweitert. Betrachten wir eine Regel im Detail:

$$S(head) \rightarrow NP(Sbj, pn, h)\ VP(pn, head).$$

Diese Regel ist am einfachsten von rechts nach links zu lesen: Wenn eine *NP* und eine *VP* verbunden werden, bilden sie eine *S*, jedoch nur, wenn die *NP* im Subjektfall (*Sbj*) vorliegt und Person und Anzahl (*pn*) der Kategorien *NP* und *VP* identisch sind. Treffen diese Bedingungen zu, haben wir eine *S*, deren Kopf derselbe wie der Kopf der *VP* ist. Beachten Sie, dass der durch die Platzhaltervariable h bezeichnete Kopf der *NP* nicht Teil der Erweiterung der Kategorie *S* ist. Die lexikalischen Regeln für ε_2 füllen die Werte der Parameter und lassen sich ebenfalls am besten von rechts nach links lesen. Zum Beispiel besagt die Regel

$$Pronomen(Sbj, 1S, \mathbf{I}) \rightarrow \mathbf{I},$$

dass „I“ als *Pronomen* im Subjektivfall, erste Person Singular mit dem Kopf „I“ interpretiert werden kann. Der Einfachheit halber haben wir die Wahrscheinlichkeiten für diese Regeln weggelassen, die Erweiterung funktioniert aber mit Wahrscheinlichkeiten. Außerdem kann Erweiterung mit automatisierten Lernmechanismen arbeiten. Petrov und Klein (2007c) zeigen, wie ein Lernalgorithmus die Kategorie *NP* in NP_S und NP_O automatisch aufteilen kann.

23.3.4 Semantische Interpretation

Um zu zeigen, wie sich eine Grammatik mit Semantik ergänzen lässt, beginnen wir mit einem Beispiel, das einfacher als Englisch ist: die Semantik von arithmetischen Ausdrücken. ▶ Abbildung 23.8 zeigt eine Grammatik für arithmetische Ausdrücke, bei der jede Regel mit einer Variablen erweitert ist, die die semantische Interpretation der Phrase angibt. Die Semantik einer Ziffer wie zum Beispiel „3“ ist die Ziffer selbst. Die Semantik eines Ausdrucks wie „3 + 4“ ist der Operator „+“, der auf die Semantik der Phrase „3“ und der Phrase „4“ angewandt wird. Die Regeln entsprechen dem Prinzip der kompositionellen Semantik – die Semantik einer Phrase ist eine Funktion der Semantik der Teilphrasen. ▶ Abbildung 23.9 zeigt den Syntaxbaum für $3 + (4 \div 2)$ entsprechend dieser Grammatik. An der Wurzel des Syntaxbaumes steht mit *Exp*(5) ein Ausdruck, dessen semantische Interpretation 5 lautet.

$$Exp(x) \rightarrow Exp(x_1)\ Operator\ (op)\ Exp(x_2)\ \{x = Apply(op, x_1, x_2)\}$$
$$Exp(x) \rightarrow (\ Exp(x)\ )$$
$$Exp(x) \rightarrow Number(x)$$
$$Number(x) \rightarrow Digit(x)$$
$$Number(x) \rightarrow Number(x_1)\ Digit(x_2)\ \{x = 10 \times x_1 + x_2\}$$
$$Digit(x) \rightarrow x\{0 \leq x \leq 9\}$$
$$Operator\ (x) \rightarrow x\ \{x \in \{+, -, \div, \times\}\}$$

Abbildung 23.8: Eine Grammatik für arithmetische Ausdrücke, erweitert um eine Semantik. Jede Variable x_i stellt die Semantik eines Konstituenten dar. Beachten Sie die Verwendung der Notation {*test*}, um logische Prädikate zu definieren, die erfüllt sein müssen, bei denen es sich aber nicht um Konstituenten handelt.

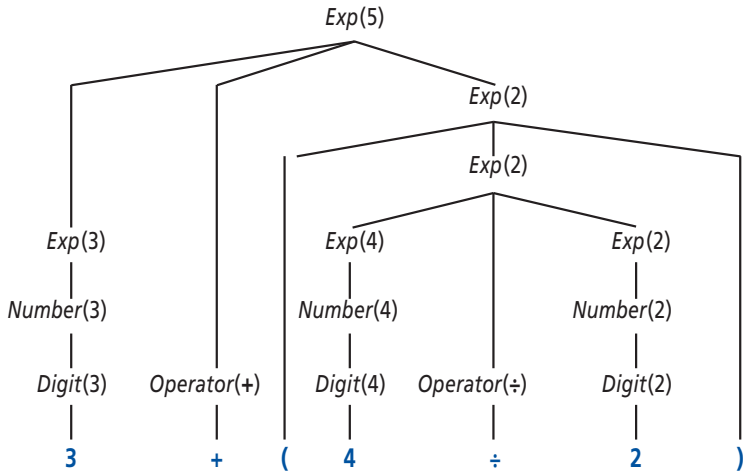


Abbildung 23.9: Syntaxbaum mit semantischen Interpretationen für die Zeichenkette „$3 + (4 \div 2)$".

Wir kommen nun zur Semantik der englischen Sprache – oder zumindest von ε_0. Zunächst bestimmen wir, welche semantischen Darstellungen wir welchen Phrasen zuordnen möchten. Wir verwenden den einfachen Beispielsatz „John loves Mary". Das *NP* „John" sollte als semantische Interpretation den logischen Term *John* haben und der Satz als Ganzes sollte als Interpretation den logischen Satz *Loves*(*John*, *Mary*) haben. Das scheint völlig klar zu sein. Der komplizierte Teil ist das *VP* „loves Mary". Die semantische Interpretation dieser Phrase ist weder ein logischer Term noch ein vollständiger logischer Satz. Intuitiv ist „loves Mary" eine Beschreibung, die für eine bestimmte Person gelten kann oder nicht gelten kann. (In diesem Fall gilt sie für John.) Das bedeutet, „loves Mary" ist ein **Prädikat**, das bei Kombination mit einem Term, der eine Person darstellt, einen vollständigen logischen Satz erzeugt. Unter Verwendung der λ-Notation (siehe *Abschnitt 8.2.3*) können wir „loves Mary" als das Prädikat

$$\lambda x\ Loves(x, Mary)$$

darstellen. Jetzt brauchen wir eine Regel, die besagt, „ein *NP* mit Semantik *obj* gefolgt von einem *VP* mit Semantik *pred* ergibt einen Satz, dessen Semantik das Ergebnis der Anwendung von *pred* auf *obj* ist":

$$S(pred(obj)) \rightarrow NP(obj)\ VP(pred).$$

Die Regel teilt uns mit, dass die semantische Interpretation von „John loves Mary" lautet:

$$(\lambda x\ Loves(x, Mary))(John),$$

was äquivalent mit *Loves*(*John*, *Mary*) ist.

Die restliche Semantik folgt auf einfache Weise aus den Entscheidungen, die wir bisher getroffen haben. Weil *VP*s als Prädikate dargestellt werden, ist es sinnvoll, konsistent zu bleiben und Verben ebenfalls als Prädikate darzustellen. Das Verb „loves" wird als $\lambda y\, \lambda x$ *Loves*(x, y) dargestellt, das Prädikat, das bei Übergabe des Arguments *Mary* das Prädikat λx *Loves*(x, *Mary*) zurückgibt. Wir gelangen zu der in ▶ Abbildung 23.10 gezeigten Grammatik und dem in ▶ Abbildung 23.11 angegebenen Syntaxbaum. Genauso einfach hätten wir ε_2 mit Semantik ergänzen können; wir haben uns für ε_0 entschieden, damit sich der Leser auf jeweils einen Erweiterungstyp konzentrieren kann.

$S(pred(obj)) \rightarrow NP(obj)\ VP(pred)$
$VP(pred(obj)) \rightarrow Verb(pred)\ NP(obj)$
$NP(obj) \rightarrow Name(obj)$

$Name(John) \rightarrow$ **John**
$Name(Mary) \rightarrow$ **Mary**
$Verb(\lambda y\ \lambda x\ Loves(x, y)) \rightarrow$ **loves**

Abbildung 23.10: Eine Grammatik, die einen Parse-Baum und eine semantische Interpretation für „John loves Mary" (und drei andere Sätze) ableiten kann. Jede Kategorie wird durch ein einzelnes Argument verstärkt, das die Semantik darstellt.

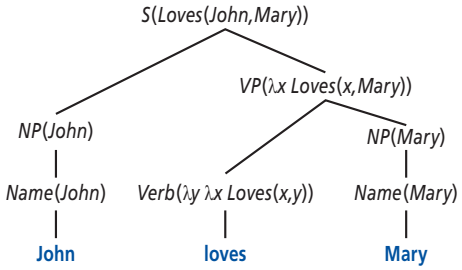


Abbildung 23.11: Ein Parse-Baum mit semantischen Interpretationen für den String „John loves Mary".

Es ist recht aufwändig und fehleranfällig, eine Grammatik manuell mit semantischen Erweiterungen auszustatten. Deshalb sind verschiedene Projekte entstanden, um semantische Erweiterungen aus Beispielen zu lernen. CHILL (Zelle und Mooney, 1996) ist ein Programm der induktiven Logikprogrammierung (ILP), das eine Grammatik und einen spezialisierten Parser für diese Grammatik aus Beispielen lernt. Die Zieldomäne sind Datenbankabfragen in natürlicher Sprache. Die Trainingsbeispiele bestehen aus Paaren von Wortfolgen und entsprechenden semantischen Formen, wie zum Beispiel:

„Wie heißt die Hauptstadt des Staats mit der größten Einwohnerzahl?"
$Antwort(c, Hauptstadt(s, c) \wedge Größter(p, Staat(s) \wedge Einwohnerzahl(s, p)))$.

Die Aufgabe von CHILL ist es, ein Prädikat *Parse*(*wörter*, *abfrage*) zu lernen, das konsistent mit den Beispielen ist und hoffentlich gut mit anderen Beispielen verallgemeinert. Die direkte Anwendung von ILP für das Lernen dieses Prädikates erbringt eine schlechte Leistung: Der induzierte Parser bietet nur etwa 20% Genauigkeit. Glücklicherweise können ILP-Lerner dies verbessern, indem sie Wissen hinzufügen. In diesem Fall wurde ein Großteil des *Parse*-Prädikates als Logikprogramm definiert und die Aufgabe von CHILL wurde auf die Induktion der Steuerregeln reduziert, die dem Parser helfen, eine Parsing-Möglichkeit gegenüber einer anderen zu bevorzugen. Mit diesem zusätzlichen Hintergrundwissen erreicht CHILL 70% bis 85% Genauigkeiten für verschiedene Aufgabenstellungen aus der Datenbankabfrage.

23.3.5 Komplikationen

Die Grammatik des realen Englisch ist unendlich komplex. An dieser Stelle erwähnen wir kurz einige Beispiele.

Zeit und Zeitform: Angenommen, wir wollen den Unterschied zwischen „John loves Mary" und „John loved Mary" darstellen. Englische Verben verwenden Zeitformen (past,

present und future), um die relative Zeit eines Ereignisses anzuzeigen. Eine sinnvolle Möglichkeit, die Zeit von Ereignissen darzustellen, ist die Ereigniskalkül-Notation aus *Abschnitt 12.3*. Im Ereigniskalkül haben wir:

$$\text{John loves mary: } E_1 \in \mathit{Loves}(\mathit{John}, \mathit{Mary}) \wedge \mathit{Während}(\mathit{Jetzt}, \mathit{Extent}(E_1));$$
$$\text{John loved mary: } E_2 \in (\mathit{Loves}(\mathit{John}, \mathit{Mary}) \wedge \mathit{Nach}(\mathit{Jetzt}, \mathit{Extent}(E_2)).$$

Das heißt, unsere beiden lexikalischen Regeln für die Wörter „loves“ und loved“ sollten wie folgt aussehen:

$$\mathit{Verb}(\lambda x\, \lambda y\, e \in \mathit{Loves}(x, y) \wedge \mathit{Während}(\mathit{Jetzt}, e) \rightarrow \textbf{loves}$$
$$\mathit{Verb}(\lambda x\, \lambda y\, e \in \mathit{Loves}(x, y) \wedge \mathit{Nach}(\mathit{Jetzt}, e) \rightarrow \textbf{loved}.$$

Bis auf diese Änderung bleibt alles Weitere in der Grammatik gleich, was eine ermutigende Tatsache ist; wir können annehmen, dass wir auf dem richtigen Weg sind, wenn wir auf so einfache Weise eine Komplikation wie etwa die Zeitform von Verben hinzufügen können (obwohl wir hier natürlich nur die Oberfläche einer vollständigen Grammatik für Zeit und Zeitformen berührt haben). Außerdem ist es ermutigend, dass sich die Unterscheidung zwischen Prozessen und diskreten Ereignissen, die wir in unserer Diskussion der Wissensdarstellung in *Abschnitt 12.3.1* getroffen haben, in der Sprachverwendung tatsächlich widerspiegelt. Wir können sagen „John slept a lot last night“, wobei *Sleeping* eine Prozesskategorie ist, doch ist es merkwürdig zu sagen: „John found a unicorn a lot last night“, wobei *Finding* eine diskrete Ereigniskategorie ist. Eine Grammatik würde diese Tatsache widerspiegeln, indem sie die Adverbphrase „a lot“ nur mit einer geringen Wahrscheinlichkeit diskreten Ereignissen zuordnet.

Quantifizierung: Betrachten Sie den Satz „Every agent feels a breeze“. Der Satz besitzt unter ε_0 nur einen syntaktischen Parsing-Verlauf, doch ist er tatsächlich semantisch mehrdeutig; die bevorzugte Bedeutung ist „For every agent there exists a breeze that the agent feels“, aber eine akzeptable alternative Bedeutung lautet: „There exists a breeze that every agent feels.“[5] Die beiden Interpretationen können wie folgt dargestellt werden:

$$\forall a \quad a \in \mathit{Agents} \Rightarrow$$
$$\quad \exists b \quad b \in \mathit{Breezes} \wedge \exists e \quad e \in \mathit{Feel}(a, b) \wedge \mathit{During}(\mathit{Now}, e);$$
$$\exists b \quad b \in \mathit{Breezes} \wedge \forall a \quad a \in \mathit{Agents} \Rightarrow$$
$$\quad \exists e \quad e \in \mathit{Feel}(a, b) \wedge \mathit{During}(\mathit{Now}, e).$$

Beim Standardansatz für die Quantifizierung definiert die Grammatik keinen eigentlichen logischen semantischen Satz, sondern eher eine **quasilogische Form**, die dann von Algorithmen außerhalb des Parsing-Vorganges in einen logischen Satz überführt wird. Derartige Algorithmen können über Vorrangregeln verfügen, um den einen Quantifiziererbereich gegenüber einem anderen zu bevorzugen – Präferenzen, die in der Grammatik nicht direkt widergespiegelt werden müssen.

Pragmatik: Wir haben gezeigt, wie ein Agent eine Folge von Wörtern wahrnehmen und mithilfe einer Grammatik eine Menge möglicher semantischer Interpretationen ableiten kann. Jetzt wenden wir uns dem Problem zu, die Interpretation durch Hinzufügen kontextabhängiger Informationen über die aktuelle Situation zu vervollständigen. Pragmatische Informationen sind offenbar notwendig, um die Bedeutung von

5 Wenn Ihnen diese Interpretation unwahrscheinlich erscheint, denken Sie an den Satz: „Jeder Protestant glaubt an einen gerechten Gott.“

Indexicals – Phrasen, die sich direkt auf die aktuelle Situation beziehen – aufzulösen. Zum Beispiel sind im Satz „I am in Boston today“ sowohl „I“ als auch „today“ Indexicals. Das Wort „I“ würde durch den Fluenten *Speaker* dargestellt und es läge beim Hörer, die Bedeutung des Fluenten aufzulösen – was nicht als Teil der Grammatik gilt, sondern vielmehr als Problem der Pragmatik; der Verwendung des Kontextes der aktuellen Situation, um Fluenten zu interpretieren.

Ein anderer Teil der Pragmatik ist die Interpretation der Sprecherabsicht. Die Aktion des Sprechers gilt als **Sprechakt** (oder **Sprechhandlung**) und es liegt beim Hörer zu entziffern, um welchen Typ von Aktion es sich handelt – eine Frage, eine Aussage, ein Versprechen, eine Warnung, einen Befehl usw. Ein Befehl wie zum Beispiel „go to 2 2“ bezieht sich implizit auf den Hörer. Bislang deckt unsere Grammatik für S nur deklarative Sätze ab. Wir können sie leicht erweitern, sodass sie auch Befehle abdeckt. Ein Befehl kann aus einer *VP* gebildet werden, wo das Subjekt implizit der Hörer ist. Wir müssen Befehle von Aussagen unterscheiden; deshalb ändern wir die Regeln für S, sodass sie den Typ des Sprechaktes beinhalten:

$$S(Aussage(Sprecher(pred(obj)))) \rightarrow NP(obj)\ VP(pred)$$
$$S(Befehl(Sprecher(pred(Hörer)))) \rightarrow VP(pred).$$

Weitreichende Abhängigkeiten: Fragen führen eine neue grammatikalische Komplexität ein. In „Who did the agent tell you to give the gold to?“ sollte das letzte Wort „to“ als [*PP* to _] geparst werden, wobei das „_“ eine Lücke oder **Spur** für ein fehlendes *NP* kennzeichnet; das fehlende *NP* ist durch das erste Wort des Satzes „who“ lizenziert. Ein komplexes System von Erweiterungen stellt sicher, dass die fehlenden *NP*s mit den lizenzierenden Wörtern genau in der richtigen Weise übereinstimmen, und verhindert Lücken an den falschen Stellen. Zum Beispiel darf keine Lücke in einem Zweig einer *NP*-Konjunktion vorkommen: „What did he play [*NP* Dungeons and _]?“ ist grammatikalisch falsch. Doch dieselbe Lücke kann in beiden Zweigen einer *VP*-Konjunktion erscheinen: „What did you [*VP*[*VP* smell _] and [*VP* shoot an arrow at _]]?“

Mehrdeutigkeit: In manchen Fällen sind sich die Zuhörer bewusst, dass eine Äußerung mehrdeutig ist. Nachfolgend einige Beispiele aus Überschriften von Zeitungen:

Ehefrau fuhr Schauspieler sturzbetrunken nach Hause.

24 Tote haben Überschwemmungen gefordert.

Mann schlug Jungen mit Skateboard.

Junge traf Nachbarn mit Gewehr.

Schütze trifft Widder.

Der Himmel lacht über Berlin.

Kieferbruch bei Wirbelsturm

Maus verursacht Stromausfall.

Tipp

Größtenteils scheint jedoch die Sprache, die wir hören, eindeutig zu sein. Als die Forscher in den 1960er Jahren begannen, Sprache per Computer zu analysieren, mussten sie überrascht feststellen, *dass fast jede Äußerung höchst mehrdeutig ist, auch wenn die alternativen Interpretationen für einen Muttersprachler nicht sofort sichtbar sein müssen*. Ein System mit großer Grammatik und großem Lexikon könnte Tausende von Interpretationsmöglichkeiten für einen ganz normalen Satz finden. Recht häufig ist die **lexikalische Mehrdeutigkeit**, bei der ein Wort mehr als eine Bedeutung hat; so kann „back“ ein Adverb (go back), ein Adjektiv (back door), ein Substantiv (the back of the room) oder ein

Verb (back up your files) sein. „Jack“ kann ein Name, ein Substantiv (eine Spielkarte, ein Spielstein mit sechs Spitzen, ein nautische Flagge, ein Fisch, eine Steckverbindung oder ein Gerät zum Heben schwerer Objekte) oder ein Verb (to jack up a car, to hunt with a light, or to hit a baseball hard) sein. Unter **syntaktischer Mehrdeutigkeit** versteht man eine Phrase, die mehrere Parsing-Verläufe besitzt. Zum Beispiel hat „I smelled a wumpus in 2,2“ zwei Parsing-Möglichkeiten: eine, bei der die präpositionale Phrase „in 2,2“ das Substantiv modifiziert, und eine, wo sie das Verb modifiziert. Die syntaktische Mehrdeutigkeit führt zu einer **semantischen Mehrdeutigkeit**, weil eine Parsing-Möglichkeit bedeutet, dass sich der Wumpus in 2,2 befindet, und die andere bedeutet, dass in 2,2 ein Gestank wahrzunehmen ist. In diesem Fall könnte die falsche Interpretation einen tödlichen Fehler bedeuten.

Schließlich kann es eine Mehrdeutigkeit zwischen wörtlichen und übertragenen Bedeutungen geben. Sprachfiguren sind in der Poesie wichtig, man findet sie jedoch auch überraschend häufig in der Alltagssprache. Die **Metonymie** ist eine Sprachfigur, bei der ein Objekt stellvertretend für ein anderes verwendet wird. Wenn wir hören „Chrysler kündigt ein neues Modell an“, gehen wir in unserer Interpretation nicht davon aus, dass Unternehmen sprechen können; stattdessen nehmen wir an, dass ein Pressesprecher, der das Unternehmen vertritt, die Ankündigung vorgenommen hat. Metonymie ist gebräuchlich und wird von menschlichen Zuhörern häufig unbewusst interpretiert. Leider ist unsere Grammatik, wie wir sie formuliert haben, nicht so wendig. Um die Semantik der Metonymie korrekt zu verarbeiten, müssen wir eine ganz neue Ebene der Mehrdeutigkeit einführen. Dazu stellen wir *zwei* Objekte für die semantische Interpretation jeder Phrase im Satz zur Verfügung: eine für das Objekt, auf das die Phrase wörtlich verweist (Chrysler), und eine für den metonymischen Verweis (den Unternehmenssprecher). Anschließend müssen wir sagen, dass es eine Relation zwischen den beiden gibt. In unserer aktuellen Grammatik wird „Chrysler kündigt an“ interpretiert als

$$x = \mathit{Chrysler} \wedge e \in \mathit{Ankündigung}(x) \wedge \mathit{Nach}(\mathit{Jetzt}, \mathit{Extent}(e)).$$

Dies müssen wir ändern in

$$x = \mathit{Chrysler} \wedge e \in \mathit{Ankündigung}(m) \wedge \mathit{Nach}(\mathit{Jetzt}, \mathit{Extent}(e)) \wedge \mathit{Metonym}(m, x).$$

Das bedeutet, dass es eine Einheit x, die gleichbedeutend mit Chrysler ist, und eine weitere Einheit m, die die Ankündigung vorgenommen hat, gibt und dass sich die beiden in metonymischer Relation befinden. Im nächsten Schritt definieren wir, welche Arten von Metonymie-Relationen auftreten können. Der einfachste Fall liegt vor, wenn überhaupt keine Metonymie auftritt – das wörtliche Objekt x und das metonymische Objekt m sind identisch:

$$\forall m, x \quad (m = x) \Rightarrow \mathit{Metonym}(m, x).$$

Für das Chrysler-Beispiel ist es eine vernünftige Verallgemeinerung, dass ein Unternehmen für den Pressesprecher dieses Unternehmens stehen kann:

$$\forall m, x \quad x \in \mathit{Unternehmen} \wedge \mathit{Pressesprecher}(m, x) \Rightarrow \mathit{Metonymie}(m, x).$$

Andere Metonymien sind etwa die Autoren von Büchern (Ich lese *Shakespeare*) oder allgemein der Produzent eines Produktes (Ich fahre einen *Honda*) oder ein Teil für das Ganze (Die *Roten Socken* brauchen eine starke *Hand*). Einige Beispiele für Metonymie wie etwa „Das *Schinkensandwich* an Tisch 4 will noch ein Bier“ sind neuer und werden im Hinblick auf eine bestimmte Situation interpretiert.

Eine **Metapher** ist eine Sprachfigur, in der eine Phrase mit einer bestimmten wörtlichen Bedeutung verwendet wird, um mithilfe einer Analogie eine andere Bedeutung auszudrücken. Somit kann man eine Metapher als eine Art Metonymie ansehen, bei der die Relation durch eine Ähnlichkeit verkörpert wird.

Auflösung der Mehrdeutigkeit ist die Wiederherstellung der wahrscheinlichsten beabsichtigten Bedeutung einer Äußerung. In gewissem Sinne verfügen wir bereits über ein Framework, um dieses Problem zu lösen: Jeder Regel ist eine Wahrscheinlichkeit zugeordnet, sodass die Wahrscheinlichkeit einer Interpretation das Produkt aus den Wahrscheinlichkeiten der Regeln ist, die zu dieser Interpretation geführt haben. Leider spiegeln die Wahrscheinlichkeiten wider, wie häufig die Phrasen im Korpus vorkommen, aus dem die Grammatik gelernt wurde. Somit reflektieren sie allgemeines Wissen und nicht das spezifische Wissen der aktuellen Situation. Um die Mehrdeutigkeit ordnungsgemäß aufzulösen, müssen wir vier Modelle kombinieren:

1. Das **Weltmodell**: die Wahrscheinlichkeit, dass eine Behauptung in der Welt zutrifft. Entsprechend unserem Wissen über die Welt ist es wahrscheinlicher, dass ein Sprecher mit „Ich bin tot“ meint „Ich habe große Schwierigkeiten“ und nicht „Mein Leben ist beendet und trotzdem kann ich immer noch reden“.

2. Das **mentale Modell**: Die Wahrscheinlichkeit, dass der Sprecher die Absicht bildet, dem Hörer eine bestimmte Tatsache mitzuteilen. Dieser Ansatz kombiniert Modelle dessen, was der Sprecher glaubt, was der Sprecher glaubt, das der Hörer glaubt, usw. Nachdem wir beispielsweise einen Politiker haben sagen hören: „Ich bin kein Halsabschneider“, könnten wir der Aussage, dass der Politiker kein Krimineller ist, eine Wahrscheinlichkeit von nur 50%, und der Aussage, dass der Politiker nicht mit dem Messer Leib und Leben anderer Leute bedroht, eine Wahrscheinlichkeit von 99,999% zuordnen. Dennoch weisen wir der ersten Interpretation eine höhere Wahrscheinlichkeit zu, weil sie eher der üblichen Ausdrucksweise entspricht.

3. Das **Sprachmodell**: die Wahrscheinlichkeit, dass eine bestimmte Wortfolge gewählt wird, wenn der Sprecher die Absicht hat, eine bestimmte Tatsache mitzuteilen.

4. Das **akustische Modell**: für gesprochene Kommunikation die Wahrscheinlichkeit, dass eine bestimmte Klangfolge erzeugt wird, wenn der Sprecher eine bestimmte Wortfolge gewählt hat. *Abschnitt 23.5* beschäftigt sich mit der Spracherkennung.

23.4 Maschinelle Übersetzung

Maschinelle Übersetzung ist die automatische Übersetzung von Text aus der einen natürlichen Sprache (der Quellsprache) in eine andere (die Zielsprache). Es war einer der ersten Anwendungsbereiche, den man sich für Computer versprach (Weaver, 1949), doch erst im letzten Jahrzehnt ist ein breiter Einsatz dieser Technologie zu verzeichnen. Das folgende Beispiel zeigt einen Ausschnitt von Seite 1 der englischen Ausgabe dieses Buches:

> AI is one of the newest fields in science and engineering. Work started in earnest soon after World War II, and the name itself was coined in 1956. Along with molecular biology, AI is regularly cited as the “field I would most like to be in” by scientists in other disciplines.

Der mit dem Online-Tool Google Translate ins Dänische übersetzte Text lautet:

> AI er en af de nyeste områder inden for videnskab og teknik. Arbejde startede for alvor a lige efter Anden Verdenskrig, og navnet i sig selv var opfundet i 1956. Sammen med molekylær biologi, er AI jævnligt nævnt som "feltet Jeg ville de fleste gerne være i" af forskere i andre discipliner.

Für diejenigen, die des Dänischen nicht mächtig sind, wurde der Text wieder ins Englische zurück übersetzt. Die von der ursprünglichen Version abweichenden Wörter sind kursiv gedruckt:

> AI is one of the newest fields *of* science and engineering. Work *began* in earnest *just* after the *Second* World War, and the name itself was *invented* in 1956. *Together* with molecular biology, AI is *frequently mentioned* as _ "field I would most like to be in" by *researchers* in other disciplines.

Die Unterschiede sind alles akzeptable Umschreibungen, wie zum Beispiel *frequently mentioned* für *regularly cited*. Der einzige Fehler ist die Auslassung des Artikels *the*, die durch das Symbol _ gekennzeichnet ist. Diese Genauigkeit ist typisch: Von den beiden Sätzen weist der eine einen Fehler auf, den ein nativer Sprecher nicht machen würde, dennoch wird die Bedeutung klar vermittelt.

Aus historischer Sicht gibt es drei Hauptanwendungen der maschinellen Übersetzung. Eine *Grobübersetzung* wie sie von kostenlosen Online-Diensten geboten wird, liefert das „Wesentliche" eines fremdsprachigen Satzes oder Dokumentes, enthält aber Fehler. Auf *vorbearbeitete Übersetzungen* stützen sich Firmen, um ihre Dokumentationen und Vertriebsmaterialien in mehreren Sprachen zu veröffentlichen. Der ursprüngliche Quelltext wird in einer eingeschränkten Sprache geschrieben, die sich automatisch leichter übersetzen lässt, und die Ergebnisse werden normalerweise von einem Menschen bearbeitet, um Fehler zu korrigieren. Die *Übersetzung für eingeschränkte Quelle* arbeitet vollkommen automatisch, jedoch nur mit einer höchst stereotypen Sprache, etwa einem Wetterbericht.

Übersetzen ist schwierig, weil es für den vollkommen allgemeinen Fall ein tiefes Verständnis des Textes voraussetzt. Das gilt sogar für sehr einfache Texte – selbst „Texte", die nur aus einem einzigen Wort bestehen. Nehmen Sie als Beispiel das Wort „Open" an einer Ladentür.[6] Es gibt zu verstehen, dass der Laden derzeit auf Kundschaft eingerichtet ist. Betrachten Sie nun dasselbe Wort „Open" auf einem großen Transparent außerhalb eines neu eingerichteten Ladens. Es bedeutet, dass der Laden seine Dienste jetzt täglich anbietet, doch werden sich die Leser dieses Hinweises nicht in die Irre geführt vorkommen, wenn der Laden in der Nacht geschlossen bleibt, ohne dass das Transparent entfernt wird. Die beiden Hinweise verwenden dasselbe Wort, um zwei verschiedene Bedeutungen zu vermitteln. Im Deutschen würde der Hinweis an der Tür „Offen" lauten und auf dem Transparent „Neu eröffnet" stehen.

Das Problem besteht darin, dass unterschiedliche Sprachen die Welt verschieden kategorisieren. Zum Beispiel deckt das französische Wort „doux" einen weiten Bedeutungsbereich ab, der ungefähr den englischen Wörtern „soft", „sweet" und „gentle" entspricht. Das englische Wort „hard" umfasst praktisch alle Verwendungen des deutschen Wortes „hart" (physisch widerspenstig, grausam) und einige Verwendungen des Wortes „schwierig" (difficult). Demzufolge ist es für eine Übersetzung schwieriger als für das

6 Dieses Beispiel stammt von Martin Kay.

Verständnis einer einzigen Sprache, die Bedeutung eines Satzes darzustellen. Ein englisches Parsing-System könnte Prädikate wie $Open(x)$ verwenden, doch für eine Übersetzung müsste die Darstellungssprache mehr differenzieren, etwa $Open_1(x)$ im Sinn von „Offen“ und $Open_2(x)$ im Sinn von „Neu eröffnet“. Eine Darstellungssprache, die alle für eine Menge von Sprachen notwendigen Unterscheidungen trifft, wird als **Interlingua** (**Zwischensprache**) bezeichnet.

Ein (menschlicher oder maschineller) Übersetzer muss oftmals die eigentliche, in der Quelle beschriebene Situation und nicht nur die einzelnen Wörter verstehen. Um zum Beispiel das englische Wort „him“ in das Koreanische zu übersetzen, ist eine Entscheidung bezüglich der richtigen Höflichkeitsform (Honorativform) zu treffen – eine Entscheidung, die von der sozialen Beziehung zwischen dem Sprecher und der Bezugsperson von „him“ abhängig ist. Im Japanischen sind die Höflichkeitsformen relativ, sodass die Auswahl von den sozialen Beziehungen zwischen dem Sprecher, der Bezugsperson und dem Hörer abhängt. Übersetzern (sowohl maschinellen als auch menschlichen) fällt diese Entscheidung manchmal schwer. Ein weiteres Beispiel: Um „The baseball hit the window. It broke.“ ins Französische zu übersetzen, müssen wir das weibliche „elle“ oder das männliche „il“ für „it“ auswählen, d.h. entscheiden, ob sich „it“ auf den Baseball oder das Fenster bezieht. Für die richtige Übersetzung sind sowohl Kenntnisse in Physik als auch der Sprache erforderlich.

Manchmal gibt es *keine Alternative*, die eine vollkommen befriedigende Übersetzung liefern kann. Zum Beispiel muss ein italienisches Liebesgedicht, das mit dem männlichen „il sole“ (Sonne) und dem weiblichen „la luna“ (Mond) zwei Liebende symbolisiert, zwangsläufig geändert werden, wenn man es ins Deutsche übersetzt, wo die Geschlechter umgekehrt sind, und weiter verändert werden, wenn man es in eine Sprache übersetzt, in der die Geschlechter gleich sind.[7]

23.4.1 Maschinelle Übersetzungssysteme

Alle Übersetzungssysteme müssen die Quell- und Zielsprachen modellieren, doch variieren die Systeme in der Art der verwendeten Modelle. Manche Systeme versuchen, den ganzen Text der Quellsprache in einer Interlingua-Wissensdarstellung zu analysieren und dann aus dieser Darstellung Sätze in der Zielsprache zu generieren. Dies ist schwierig, weil dabei drei ungelöste Probleme auftreten: Erstellen einer vollständigen Wissensdarstellung von allem, Parsen in diese Darstellung und Generieren von Sätzen aus dieser Darstellung.

Andere Systeme basieren auf einem **Transfermodell**. Sie verwalten eine Datenbank von Übersetzungsregeln (oder Beispielen) und übersetzen direkt, wenn die Regel (oder das Beispiel) zutrifft. Der Transfer kann auf der lexikalischen, syntaktischen oder semantischen Ebene stattfinden. Zum Beispiel bildet eine streng syntaktische Regel Englisch [*Adjektiv Substantiv*] auf Französisch [*Substantiv Adjektiv*] ab. Eine gemischte syntaktische und lexikalische Regel bildet Französisch [S_1 "et puis" S_2] auf Englisch [S_1 "and then" S_2] ab. ▶ Abbildung 23.12 stellt die verschiedenen Transferpunkte grafisch dar.

7 Wie Warren Weaver (1949) berichtet, hebt Max Zeldner hervor, dass der große hebräische Dichter H. N. Bialik einmal gesagt hat, eine Übersetzung sei „wie das Küssen der Braut durch einen Schleier“.

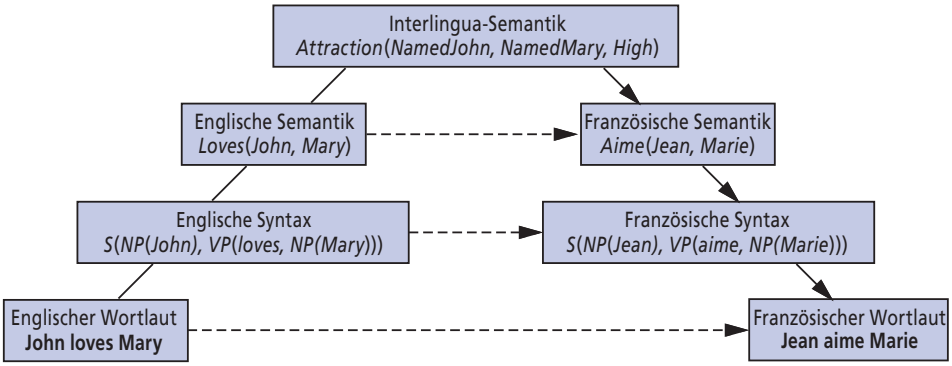


Abbildung 23.12: Das Vauquois-Dreieck: schematische Darstellung der Auswahlmöglichkeiten für ein maschinelles Übersetzungssystem (Vauquois, 1968). Wir beginnen an der Spitze mit einem englischen Text. Ein mit Zwischensprache (Interlingua) arbeitendes System folgt den durchgezogenen Linien, wobei zuerst Englisch in eine syntaktische Form geparst, dann in eine semantische Darstellung und eine Interlingua-Darstellung überführt und anschließend über Generierung in eine semantische, syntaktische und lexikalische Form ins Französische übersetzt wird. Ein transferbasiertes System verwendet die kurzen Wege, die als gestrichelte Linien angegeben sind. Unterschiedliche Systeme führen den Transfer an unterschiedlichen Punkten durch; bei manchen Systemen findet er an mehreren Punkten statt.

23.4.2 Statistische maschinelle Übersetzung

Nachdem wir gesehen haben, wie komplex eine Übersetzung sein kann, dürfte es nicht überraschen, dass vor allem die Übersetzungssysteme erfolgreich sind, die ein probabilistisches Modell anhand von statistischen Daten trainieren, die aus einem großen Textkorpus gewonnen werden. Dieser Ansatz benötigt weder eine komplexe Ontologie von Interlingua-Konzepten noch manuell erstellte Grammatiken der Quell- und Zielsprachen noch eine manuell beschriftete Baumbank. Es sind lediglich Daten erforderlich – Beispielübersetzungen, aus denen sich ein Übersetzungsmodell lernen lässt. Um einen Satz von beispielsweise Englisch (e) in Französisch (f) zu übersetzen, suchen wir mit f^* die Folge von Wörtern, die

$$f^* = \operatorname*{argmax}_f P(f \mid e) = \arg\max_f P(e \mid f) P(f)$$

maximiert.

Hier ist der Faktor $P(f)$ das Ziel-**Sprachmodell** für Französisch; er besagt, wie wahrscheinlich ein bestimmter Satz in Französisch ist. $P(e|f)$ bezeichnet das **Übersetzungsmodell** und gibt an, wie wahrscheinlich ein englischer Satz als Übersetzung für einen bestimmten französischen Satz ist. Analog ist $P(f|e)$ ein Übersetzungsmodell vom Englischen ins Französische.

Sollten wir direkt auf $P(f|e)$ arbeiten oder die Bayessche Regel anwenden und auf $P(e|f)$ $P(f)$ arbeiten? In **diagnostischen** Anwendungen wie zum Beispiel in der Medizin ist es einfacher, die Domäne in der kausalen Richtung zu modellieren: $P(symptome|krankheit)$ statt $P(krankheit|symptome)$. Doch in einer Übersetzung sind beide Richtungen gleich gut. Die frühesten Arbeiten in der maschinellen Übersetzung haben die Bayessche Regel angewandt – zum Teil weil die Forscher über ein gutes Sprachmodell $P(f)$ verfügten und es nutzen wollten, zum Teil weil sie von der Spracherkennung kamen, die ein diagnostisches Problem verkörpert. Wir folgen in diesem Kapitel ihrem Beispiel, weisen aber dar-

auf hin, dass neuere Arbeiten in der statistischen maschinellen Übersetzung oftmals $P(f|e)$ direkt optimieren, und zwar mit einem komplexeren Modell, das viele der Merkmale aus dem Sprachmodell berücksichtigt.

Das Sprachmodell $P(f)$ könnte auf beliebige Ebenen der rechten Seite von Abbildung 23.12 zugreifen, doch der einfachste und gebräuchlichste Ansatz ist es, ein n-Gramm-Modell von einem französischen Korpus zu erstellen, wie wir es bereits gesehen haben. Dieses erfasst lediglich einen partiellen, lokalen Gedanken von französischen Sätzen, was allerdings oftmals für eine Grobübersetzung ausreichend ist.[8]

Das Übersetzungsmodell wird von einem **zweisprachigen Korpus** gelernt – einer Sammlung von parallelen Texten, die aus Englisch/Französisch-Paaren besteht. Hätten wir einen unendlich großen Korpus, wäre die Übersetzung eines Satzes eine simple Nachschlageaufgabe: Wir würden den englischen Satz vorher im Korpus sehen, sodass wir einfach den gepaarten französischen Satz zurückgeben könnten. Doch unsere Ressourcen sind natürlich endlich und die meisten Sätze, die wir für eine Übersetzung abfragen, werden neue Sätze sein. Allerdings setzen sie sich aus **Phrasen** zusammen, die wir bereits kennen (selbst wenn bestimmte Phrasen nur ein Wort lang sind). Zum Beispiel gehören in diesem Buch (bezogen auf die englische Ausgabe) „in this exercise we will“, „size of the state space“, „as a function of the“ und „notes at the end of the chapter“ zu den gebräuchlichen Phrasen. Für eine Übersetzung des neuen Satzes „In this exercise we will compute the size of the state space as a function of the number of actions.“ in das Französische teilen wir den Satz in Phrasen auf, suchen die Phrasen im englischen Korpus (in diesem Buch), suchen die entsprechenden französischen Phrasen (aus der französischen Übersetzung des Buches) und fügen dann die französischen Phrasen in einer Reihenfolge zusammen, die im Französischen sinnvoll ist. Mit anderen Worten ist es für einen gegebenen englischen Quellsatz e eine Angelegenheit von drei Schritten, eine französische Übersetzung f zu finden:

1. Den englischen Satz in Phrasen $e_1, \ldots, e_n$ aufteilen
2. Für jede Phrase e_i die entsprechende französische Phrase f_i auswählen. Wir verwenden die Notation $P(f_i|e_i)$ für die Phrasenwahrscheinlichkeit, dass f_i eine Übersetzung von e_i ist.
3. Eine Permutation der Phrasen $f_1, \ldots, f_n$ auswählen. Unsere Spezifikation dieser Permutation mag ein wenig kompliziert aussehen, sie ist aber so gestaltet, dass sich eine einfache Wahrscheinlichkeitsverteilung ergibt: Für jedes f_i wählen wir eine **Verzerrung** d_i, die die Anzahl der Wörter angibt, um die sich die Phrase f_i in Bezug auf $f_i - 1$ verschoben hat; der Wert ist positiv bei einer Verschiebung nach rechts, negativ bei einer Verschiebung nach links und null, wenn f_i unmittelbar auf $f_i - 1$ folgt.

▶ Abbildung 23.13 zeigt ein Beispiel für den Vorgang. An der Spitze wird der Satz „There is a smelly wumpus sleeping in 2 2“ in die fünf Phrasen $e_1, \ldots, e_5$ aufgeteilt.

8 Für die feineren Punkte der Übersetzung sind n-Gramme ausreichend deutlich. Die 4000 Seiten umfassende Novelle *A la récherche du temps perdu* von Marcel Proust beginnt und endet mit demselben Wort (*longtemps*), sodass sich manche Übersetzer entschieden haben, das Gleiche zu tun und somit für die Übersetzung des letzten Wortes ein Wort heranzogen, das ungefähr 2 Millionen Wörter früher erschienen ist.

Jede dieser Phrasen wird in eine korrespondierende Phrase f_i übersetzt und dann werden diese Übersetzungen in die Reihenfolge f_1, f_3, f_2, f_4, f_5 permutiert. Wir spezifizieren die Permutation in Form der Verzerrungen d_i jeder französischen Phrase, die als

$$d_i = \text{START}(f_i) - \text{END}(f_{i-1}) - 1$$

definiert sind, wobei $\text{START}(f_i)$ die Ordinalzahl des ersten Wortes der Phrase f_i im französischen Satz und $\text{END}(f_i - 1)$ die Ordinalzahl des letzten Wortes von Phrase $f_i - 1$ angeben. Wie Abbildung 23.13 zeigt, folgt f_5, „à 2 2“ unmittelbar auf f_4 „qui dort“. Somit ist $d_5 = 0$. Dagegen ist Phrase f_2 um ein Wort rechts von f_1 verschoben worden, sodass $d_2 = 1$ ist. Als Spezialfall haben wir $d_1 = 0$, da f_1 an Position 1 beginnt und $\text{END}(f_0)$ als 0 definiert ist (selbst wenn f_0 nicht existiert).

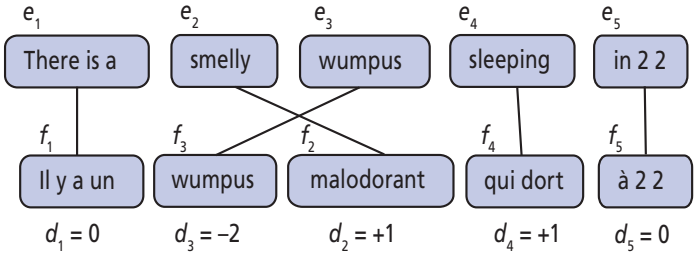


Abbildung 23.13: Französische Kandidatenphrasen für jede Phrase eines englischen Satzes mit den Verzerrungswerten (d) für jede französische Phrase.

Nachdem wir die Verzerrung d_i definiert haben, können wir die Wahrscheinlichkeitsverteilung der Verzerrung $\mathbf{P}(d_i)$ definieren. Da wir für Sätze, die durch die Länge n begrenzt sind, $|d_i| \leq n$ haben, umfasst die vollständige Wahrscheinlichkeitsverteilung $\mathbf{P}(d_i)$ lediglich $2n + 1$ Elemente. Gegenüber der Anzahl der Permuationen n! ist diese Anzahl der zu lernenden Phrasen also weit geringer. Deshalb haben wir die Permutation auf diese umständliche Art und Weise definiert. Natürlich ist dies ein recht dürftiges Modell der Verzerrung. Es sagt nichts darüber aus, dass Adjektive normalerweise nach dem Substantiv verzerrt erscheinen, wenn wir vom Englischen ins Französische übersetzen – diese Tatsache wird im französischen Sprachmodell $P(f)$ dargestellt. Die Verzerrungswahrscheinlichkeit ist vollkommen unabhängig von den Wörtern in den Phrasen – sie hängt nur vom ganzzahligen Wert d_i ab. Die Wahrscheinlichkeitsverteilung liefert eine Zusammenfassung für die Volatilität der Permutationen, zum Beispiel, wie wahrscheinlich eine Verzerrung von $P(d = 2)$ verglichen mit $P(d = 0)$ ist.

Jetzt können wir alles zusammenfügen: Mit $P(f, d|e)$ lässt sich die Wahrscheinlichkeit definieren, dass die Folge der Phrasen f mit Verzerrungen d eine Übersetzung der Folge der Phrasen e ist. Wir nehmen an, dass jede Phrasenübersetzung und jede Verzerrung unabhängig von den anderen ist und können somit den Ausdruck als Produkt

$$P(f, d \mid e) = \prod_i P(f_i \mid e_i) P(d_i)$$

darstellen.

Damit haben wir eine Möglichkeit, die Wahrscheinlichkeit $P(f, d|e)$ für eine Kandidatenübersetzung f und eine Verzerrung d zu berechnen. Um aber die beste Übersetzung f und die beste Verzerrung d zu ermitteln, können wir nicht einfach Sätze aufzählen –

denn bei vielleicht 100 französischen Phrasen für jede englische Phrase im Korpus gibt es 100^5 verschiedene 5-Phrasen-Übersetzungen und 5! Aufzeichnungen für jede davon. Wir müssen nach einer guten Lösung suchen. Eine lokale Strahlsuche (siehe *Abschnitt 4.1.3*) mit einer Heuristik, die die Wahrscheinlichkeit bewertet, hat sich als effektiv erwiesen, die nahezu wahrscheinlichste Übersetzung zu finden.

Jetzt müssen lediglich noch die Phrasen- und Verzerrungswahrscheinlichkeiten gelernt werden. Die folgenden Punkte skizzieren diesen Vorgang. Einzelheiten finden Sie in den Hinweisen am Ende des Kapitels.

1. **Parallele Texte suchen:** zuerst einen parallelen zweisprachigen Korpus zusammenstellen. Zum Beispiel ist ein **Hansard**[9] ein Datensatz von Parlamentsdebatten. Kanada, Hongkong und andere Länder produzieren zweisprachige Hansards, die Europäische Union veröffentlicht ihre offiziellen Dokumente in elf Sprachen und die Vereinten Nationen geben mehrsprachige Dokumente heraus. Zweisprachiger Text ist auch online verfügbar; manche Websites veröffentlichen parallele Inhalte mit parallelen URLs, beispielsweise `/en/` für die englische Seite und `/fr/` für die entsprechende französische Seite. Die führenden statistischen Übersetzungssysteme trainieren mit zig Millionen Wörtern von parallelem Text und Milliarden Wörtern von einsprachigem Text.

2. **In Sätze segmentieren:** Die Einheit der Übersetzung ist ein Satz. Demnach müssen wir den Korpus in Sätze aufgliedern. Zwar ist der Punkt ein deutlicher Hinweis auf das Ende eines Satzes, doch bei „Dr. J. R. Smith of Rodeo Dr. paid $29.99 on 9.9.09.“ sieht dies ganz anders aus: Nur der letzte Punkt beendet einen Satz. Um zu entscheiden, ob ein Punkt tatsächlich einen Satz beendet, lässt sich ein Modell trainieren, das als Merkmale die umgebenden Wörter und ihre Teile der Rede verwendet. Dieser Ansatz erreicht ungefähr 98% Genauigkeit.

3. **Sätze ausrichten:** Für jeden Satz in der englischen Version ist zu ermitteln, welchem Satz bzw. welchen Sätzen er in der französischen Version entspricht. Normalerweise entspricht der nächste englische Satz dem nächsten französischen Satz bei einer 1:1-Zuordnung, doch manchmal tritt eine Abweichung auf: Ein Satz der einen Sprache wird zu einer 2:1-Zuordnung aufgeteilt oder die Reihenfolge von zwei Sätzen wird vertauscht, was zu einer 2:2-Zuordnung führt. Indem man nur die Satzlänge betrachtet (d.h. kurze Sätze mit kurzen Sätzen auszurichten sind), ist es möglich, sie mit einer Genauigkeit im Bereich von 90% bis 99% mithilfe einer Variation des Viterbi-Algorithmus (1:1, 1:2 oder 2:2 usw.) auszurichten. Eine noch bessere Ausrichtung lässt sich mithilfe von markanten Elementen erreichen, die beiden Sprachen gemeinsam sind, beispielsweise Zahlen, Daten, Eigennamen oder Wörter, bei denen aus einem zweisprachigen Wörterbuch bekannt ist, dass ihre Übersetzung mehrdeutig ist. Wenn zum Beispiel der dritte englische und der vierte französische Satz die Zeichenfolge „1989“ enthalten und die benachbarten Sätze nicht, ist dies ein gutes Anzeichen dafür, dass die Sätze zusammen ausgerichtet werden sollten.

4. **Phrasen ausrichten:** Innerhalb eines Satzes lassen sich Phrasen nach einer ähnlichen Prozedur wie bei der Satzausrichtung ausrichten, wobei aber eine iterative Verbesserung erforderlich ist. Zu Beginn haben wir keine Möglichkeit zu erfahren,

9 Benannt nach William Hansard, der als Erster 1811 die britischen Parlamentsdebatten veröffentlichte.

dass „qui dort“ mit „sleeping“ auszurichten ist, doch per Aggregation von Evidenz können wir zu dieser Ausrichtung gelangen. Unter allen bisher gesehenen Beispielsätzen fällt auf, dass „qui dort“ und „sleeping“ mit hoher Häufigkeit zusammen auftreten und dass im Paar der ausgerichteten Sätze keine andere Phrase als „qui dort“ so häufig in anderen Sätzen zusammen mit „sleeping“ vorkommt. Eine vollständige Phrasenausrichtung über unseren Korpus liefert uns (nach geeigneter Glättung) die Phrasenwahrscheinlichkeiten.

5 **Verzerrungen extrahieren:** Nachdem wir alle Phrasen ausgerichtet haben, können wir Verzerrungswahrscheinlichkeiten definieren. Dazu zählen wir einfach, wie oft Verzerrung im Korpus für jede Distanz $d = 0, \pm 1, \pm 2, \ldots$ auftritt, und wenden Glättung an.

6 **Schätzungen mit dem EM-Algorithmus verbessern:** mit dem EM-Algorithmus (Erwartung-Maximierung) die Schätzungen der Werte von $P(f|e)$ und $P(d)$ verbessern. Wir berechnen die besten Ausrichtungen mit den aktuellen Werten dieser Parameter im E-Schritt, aktualisieren dann die Schätzungen im M-Schritt und wiederholen diesen Vorgang bis zur Konvergenz.

23.5 Spracherkennung

Unter **Spracherkennung** versteht man die Aufgabe, aus einem akustischen Signal eine Folge von Wörtern zu erkennen, die ein Sprecher artikuliert. Diese Anwendung der KI ist besonders populär – Millionen Menschen kommunizieren tagtäglich mit Spracherkennungssystemen, um beispielsweise Sprachmailsysteme zu steuern oder das Web von Mobiltelefonen aus zu durchsuchen. Sprache ist eine attraktive Option, wenn ein freihändiger Betrieb erforderlich ist, beispielsweise bei der Bedienung einer Maschine.

Spracherkennung ist schwierig, weil die von einem Sprecher erzeugten Laute mehrdeutig und zudem verrauscht sind. Ein bekanntes Beispiel ist die Phrase „recognize speech“, die bei schnellem Sprechtempo fast wie „wreck a nice beach“ klingt. Selbst dieses kurze Beispiel deutet auf mehrere Punkte hin, die bei der Spracherkennung problematisch sind. Erstens ist die **Segmentierung** zu nennen: Im geschriebenen Englischen sind die Wörter durch Leerzeichen voneinander getrennt, doch bei schneller Rede gibt es keine Pausen in „wreck a nice“, die die Phrase als Mehrwortphrase gegenüber dem einzelnen Wort „recognize“ unterscheiden würden. Zweitens spielt die **Koartikulation** eine Rolle: Beim schnellen Sprechen verschmilzt der „s“-Laut am Ende von „nice“ mit dem „b“-Laut am Anfang von „beach“, was einen Laut in der Nähe von „sp“ ergibt. Ein anderes Problem, das in diesem Beispiel nicht auftaucht, sind **Homophone** – Wörter wie „to“, „too“ und „two“, die gleich klingen, aber verschiedene Bedeutungen besitzen.

Wir können die Spracherkennung als Problem in der Erklärung der wahrscheinlichsten Folge ansehen. Wie *Abschnitt 15.2* gezeigt hat, ist dies das Problem, die wahrscheinlichste Folge von Zustandsvariablen $x_{1:t}$ für eine gegebene Folge von Beobachtungen $e_{1:t}$ zu berechnen. In diesem Fall sind die Zustandsvariablen die Wörter und die Beobachtungen sind Laute. Genauer gesagt ist eine Beobachtung ein Vektor von Merkmalen, die aus dem Audiosignal extrahiert werden. Wie üblich lässt sich die wahrscheinlichste Folge mithilfe der Bayesschen Regel berechnen zu:

$$\underset{wort_{1:t}}{\operatorname{argmax}} P(word_{1:t} \mid laut_{1:t}) = \underset{wort_{1:t}}{\operatorname{argmax}} P(laut_{1:t} \mid wort_{t:1}) P(wort_{1:t})$$

Hier ist $P(laut_{1:t} | wort_{1:t})$ das **akustische Modell**. Es beschreibt die Laute von Wörtern – dass „ceiling“ mit einem weichen „c“ beginnt und genau wie „sealing“ klingt. $P(wort_{1:t})$ wird als das Sprachmodell bezeichnet. Es spezifiziert die A-priori-Wahrscheinlichkeit für jede Äußerung – zum Beispiel, dass „ceiling fan“ als Wortfolge ungefähr 500-mal wahrscheinlicher ist als „sealing fan“. Dieses Konzept hat Claude Shannon (1948) als **verrauschtes Kanalmodell** bezeichnet. Er beschrieb eine Situation, in der eine Originalnachricht (die Wörter in unserem Beispiel) über einen verrauschten Kanal (wie zum Beispiel eine Telefonleitung) übertragen wird, sodass das andere Ende eine verstümmelte Nachricht (die Laute in unserem Beispiel) empfängt. Shannon zeigte, dass sich unabhängig davon, wie verrauscht der Kanal ist, die Originalnachricht mit beliebig kleinem Fehler wiederherstellen lässt, wenn wir nur die Originalnachricht genügend redundant kodieren. Das Konzept des verrauschten Kanals wurde für Spracherkennung, maschinelle Übersetzung, Rechtschreibkorrektur und andere Aufgaben eingesetzt.

Nachdem wir die akustischen und sprachlichen Modelle definiert haben, können wir mithilfe des Viterbi-Algorithmus (*Abschnitt 15.2.3*) nach der wahrscheinlichsten Folge von Wörtern auflösen. Die meisten Spracherkennungssysteme verwenden ein Sprachmodell, das die Markov-Annahme trifft – dass der aktuelle Zustand $Wort_t$ nur von einer festen Anzahl n vorheriger Zustände abhängt – und stellen $Wort_t$ als einzelne Zufallsvariable dar, die eine endliche Menge von Werten annimmt, wodurch es zu einem Hidden-Markov-Modell (HMM) wird. Somit wird die Spracherkennung zu einer einfachen Anwendung der HMM-Methodologie, wie sie *Abschnitt 15.3* beschrieben hat – einfach heißt, nachdem wir die akustischen und sprachlichen Modelle definiert haben. Darauf gehen wir als Nächstes ein.

23.5.1 Akustisches Modell

Schallwellen sind periodische Druckänderungen, die sich durch die Luft fortpflanzen. Wenn diese Wellen auf die Membran eines Mikrofons treffen, erzeugen die dadurch verursachten Bewegungen einen elektrischen Strom. Ein Analog/Digital-Wandler misst die Größe des Stroms – die der Amplitude der Schallwelle entspricht – in diskreten Intervallen, was als **Abtastrate** bezeichnet wird. Für die Sprache, die normalerweise im Frequenzbereich von 100 Hz (100 Schwingungen pro Sekunde) bis 1000 Hz liegt, ist eine Abtastrate von 8 kHz typisch. (Bei CDs und mp3-Dateien liegt die Abtastrate bei 44,1 kHz.) Die Genauigkeit jeder Messung wird durch den **Quantisierungsfaktor** festgelegt; Spracherkennungssysteme verwenden normalerweise 8 bis 12 Bit. Das bedeutet, ein einfaches System, das mit 8 kHz bei 8-Bit-Quantisierung aufzeichnet, benötigt etwa ein halbes Megabyte pro Sprachminute.

Da wir nur an den Wörtern, die gesprochen wurden, interessiert sind und nicht wissen wollen, wie sie genau klingen, brauchen wir nicht sämtliche Informationen zu speichern. Wir müssen nur zwischen verschiedenen **Sprachlauten** unterscheiden. Linguisten haben ungefähr 100 Sprachlaute identifiziert, aus denen sich alle Wörter in allen bekannten menschlichen Sprachen bilden lassen. Grob gesagt ist ein Laut der Klang, der einem einzelnen Vokal oder Konsonanten entspricht. Allerdings gibt es einige Komplikationen: Buchstabenkombinationen wie zum Beispiel „th“ und „ng“ erzeugen einzelne Laute und einige Buchstaben erzeugen in verschiedenen Kontexten unterschiedliche Laute (z.B. das

„a“ in „rat“ und „rate“). ▶ Abbildung 23.14 listet alle Laute auf, die im Englischen verwendet werden, und gibt zu jedem ein Beispiel an. Ein **Phonem** ist die kleinste Klangeinheit, die für die Sprecher einer bestimmten Sprache eine unterschiedliche Bedeutung haben kann. Beispielsweise ist im Englischen das „t“ in „stick“ dasselbe Phonem wie das „t“ in „tick“, aber in Thai können sie als zwei separate Phoneme unterschieden werden. Um gesprochenes Englisch zu repräsentieren, brauchen wir eine Darstellung, die zwischen verschiedenen Phonemen unterscheiden kann, aber die nichtphonemischen Variationen im Klang nicht unterscheiden muss: laut oder leise, schnell oder langsam, männliche oder weibliche Stimme, etc.

Vokale		Konsonanten B – N		Konsonanten P – Z	
Laut	**Beispiel**	**Laut**	**Beispiel**	**Laut**	**Beispiel**
[iy]	b**ea**t	[b]	**b**et	[p]	**p**et
[ih]	b**i**t	[ch]	**Ch**et	[r]	**r**at
[eh]	b**e**t	[d]	**d**ebt	[s]	**s**et
[æ]	b**a**t	[f]	**f**at	[sh]	**sh**oe
[ah]	b**u**t	[g]	**g**et	[t]	**t**en
[ao]	b**ough**t	[hh]	**h**at	[th]	**th**ick
[ow]	b**oa**t	[hv]	**h**igh	[dh]	**th**at
[uh]	b**oo**k	[jh]	**j**et	[dx]	bu**tt**er
[ey]	b**ai**t	[k]	**k**ick	[v]	**v**et
[er]	B**er**t	[l]	**l**et	[w]	**w**et
[ay]	b**u**y	[el]	bott**le**	[wh]	**wh**ich
[oy]	b**oy**	[m]	**m**et	[y]	**y**et
[axr]	din**er**	[em]	bott**om**	[z]	**z**oo
[aw]	d**ow**n	[n]	**n**et	[zh]	mea**s**ure
[ax]	**a**bout	[en]	butt**on**		
[ix]	ros**e**s	[ng]	si**ng**		
[aa]	c**o**t	[eng]	wash**ing**	[-]	*lautlos*

Abbildung 23.14: Das phonetische ARPA-Alphabet, auch als **ARPAbet** bezeichnet, das alle im amerikanischen Englisch verwendeten Laute auflistet. Es gibt mehrere alternative Notationen, unter anderem ein Internationales Phonetisches Alphabet (IPA), das die Laute aller bekannten Sprachen enthält.

Zuerst beobachten wir, dass zwar die Schallfrequenzen in der Sprache mehrere kHz umfassen können, die *Änderungen* im Signalinhalt jedoch weniger häufig auftreten, nämlich mit vielleicht nicht mehr als 100 Hz. Aus diesem Grund summieren Spracherkennungssysteme die Eigenschaften des Signals über längere Intervalle, sogenannte **Frames**. Eine Framelänge von etwa 10 ms (d.h. 80 Abtastungen bei 8 kHz) ist kurz genug, um einige kurz dauernde Phänomene sicher zu unterdrücken. Überlappende

Frames gewährleisten, dass kein Signal verlorengeht, weil es auf eine Frame-Grenze fällt.

Jeder Frame wird durch einen Vektor von **Merkmalen** zusammengefasst. Das Auswählen einzelner Merkmale aus einem Sprachsignal ist so, als würde man einem Orchester lauschen und sagen: „Hier spielen die Waldhörner laut und die Geigen leise." Zunächst wird mit einer Fourier-Transformation die Größe der akustischen Energie bei ungefähr einem Dutzend Frequenzen bestimmt. Dann wird ein als **MFCC** (**Mel-Frequenz-Cepstrum-Koeffizient**, Mel Frequency Cepstral Coefficient) bezeichnetes Maß für jede Frequenz berechnet. Außerdem berechnet man die Gesamtenergie im Frame. Dadurch erhält man dreizehn Merkmale; für jedes werden die Differenz zwischen diesem Frame und dem vorhergehenden Frame und die Differenz zwischen Differenzen berechnet, was insgesamt 39 Merkmale ergibt. Diese kontinuierlichen Werte lassen sich am einfachsten durch Diskretisierung für das HMM-Framework aufbereiten. (Es ist auch möglich, das HMM-Modell zu erweitern, um kontinuierliche Mischungen von Gaußverteilungen zu verarbeiten.) ▶ Abbildung 23.15 zeigt die Abfolge der Transformationen vom ursprünglichen Laut in eine Folge von Frames mit diskreten Merkmalen.

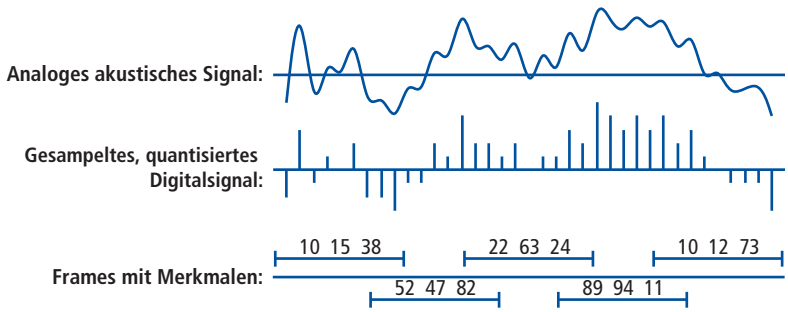
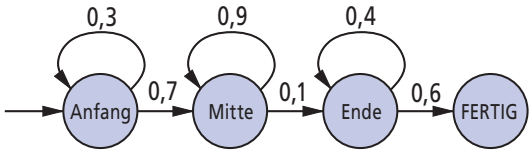


Abbildung 23.15: Übersetzung der akustischen Signale in eine Folge von Frames. In dieser Grafik wird jeder Frame durch die diskretisierten Werte von drei akustischen Merkmalen beschrieben; bei einem realen System sind es in der Regel Dutzende von Merkmalen.

Sie wissen nun, wie Sie vom ursprünglichen akustischen Signal zu einer Folge von Beobachtungen $\mathbf{e}_t$ gelangen. Wir müssen nun die (nicht beobachtbaren) Zustände des HMM beschreiben und das Übergangsmodell $\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1})$ sowie das Sensormodell $\mathbf{P}(\mathbf{E}_t \mid \mathbf{X}_t)$ definieren. Das Übergangsmodell lässt sich in zwei Ebenen aufteilen: Wort und Laut. Wir beginnen von unten: Das **Lautmodell** beschreibt einen Laut in Form von drei Zuständen – Anfangs-, Mitte- und Endzustand. Zum Beispiel hat der [t]-Laut einen stummen Anfang, eine kleine Explosion in der Mitte und (normalerweise) ein Zischen am Ende. ▶ Abbildung 23.16 zeigt ein Beispiel für den Laut [m]. Bei normaler Sprache beträgt die Dauer für einen durchschnittlichen Laut etwa 50 bis 100 Millisekunden oder 5 bis 10 Frames. Die Schleifen in jedem Zustand erlauben es, diese Dauer zu variieren. Indem man viele Schleifen (insbesondere im mittleren Zustand) einbaut, lässt sich ein langer „mmmmmmmmmm"-Laut darstellen. Werden die Schleifen übergangen, ist das Ergebnis ein kurzer „m"-Laut.

Laut-HMM für [m]:



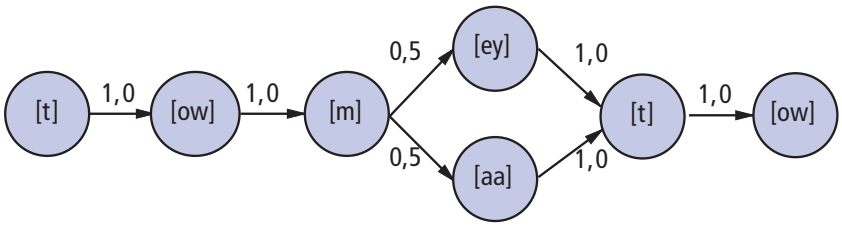
Ausgabewahrscheinlichkeiten für das Laut-HMM:

Anfang:	Mitte:	Ende:
C_1: 0,5	C_3: 0,2	C_4: 0,1
C_2: 0,2	C_4: 0,7	C_6: 0,5
C_3: 0,3	C_5: 0,1	C_7: 0,4

Abbildung 23.16: Ein HMM für den Dreizustandslaut [m]. Jeder Zustand hat mehrere mögliche Ausgaben mit jeweils eigener Wahrscheinlichkeit. Die willkürlich gewählten MFCC-Merkmalsbezeichnungen C_1 bis C_7 stehen für bestimmte Kombinationen von Merkmalswerten.

In ▶ Abbildung 23.17 sind die Lautmodelle aneinandergereiht, um ein Aussprachemodell für ein Wort zu bilden. Nach Gershwin (1937) sagen Sie [t ow m ey t ow] und ich sage [t ow m aa t ow]. Abbildung 23.17(a) zeigt ein Übergangsmodell, das diese Dialektvariation berücksichtigt. Jeder Kreis in dieser Grafik verkörpert ein Lautmodell wie das in Abbildung 23.16.

a Wortmodell mit Dialektvariation:



b Wortmodell mit Koartikulation und Dialektvariationen:

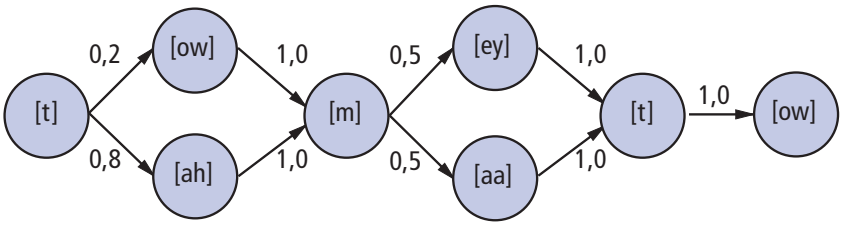


Abbildung 23.17: Zwei Aussprachemodelle des Wortes „tomato". Die beiden Modelle sind als Übergangsdiagramm dargestellt, wobei die Kreise die Zustände kennzeichnen, die Pfeile die erlaubten Übergänge mit den ihnen zugeordneten Wahrscheinlichkeiten. (a) Ein Modell, das Dialektunterschiede berücksichtigt. Die Zahlen 0,5 sind die Schätzungen basierend auf den bevorzugten Aussprachen der beiden Autoren. (b) Ein Modell mit Koartikulationseffekt auf dem ersten Vokal, wobei die Laute [ow] oder [ah] unterstützt werden.

Außer der Dialektvariation können Wörter auch eine Variation der **Koartikulation** aufweisen. Zum Beispiel wird der [t]-Laut erzeugt, wenn die Zunge ganz oben im Mund anliegt, während sich bei [ow] die Zunge fast ganz unten befindet. Beim schnellen Sprechen hat die Zunge nicht genügend Zeit, um die Position für den [ow]-Laut einzunehmen und wir erhalten [t ah] statt [t ow]. Abbildung 23.17(b) zeigt ein Modell für „tomato“, das diesen Koartikulationseffekt berücksichtigt. Komplexere Lautmodelle beziehen auch den Kontext der umgebenden Laute ein.

Die Aussprache eines Wortes kann erheblich variieren. Die häufigste Aussprache von „because“ ist [b iy k ah z], doch gilt dies nur für rund ein Viertel der Sprecher. Rund ein anderes Viertel setzt [ix], [ih] oder [ax] für den ersten Vokal und der Rest setzt [ax] oder [aa] für den zweiten Vokal, [zh] oder [s] für den letzen [z]-Laut oder lässt „be“ vollkommen weg, sodass „cuz“ übrig bleibt.

23.5.2 Sprachmodell

Für die allgemeine Spracherkennung kann das Sprachmodell ein *n*-Gramm-Modell von Text sein und aus einem Korpus geschriebener Sätze gelernt werden. Allerdings weist gesprochene Sprache andere Charakteristika auf als die Schriftsprache, sodass es besser ist, einen Korpus von Transkriptionen der gesprochenen Sprache zu verwenden. Für aufgabenspezifische Spracherkennung sollte der Korpus entsprechend aufgabenspezifisch sein: Um ein Flugreservierungssystem aufzubauen, sind Transkriptionen von vorherigen Anrufen zu empfehlen. Auch ein aufgabenspezifisches Vokabular ist hilfreich, etwa eine Liste aller bedienten Flughäfen und Städte sowie alle Flugnummern.

Im Design einer Sprachbenutzeroberfläche ist darauf zu achten, dass der Benutzer Dinge aus einer begrenzten Menge von Optionen sagen muss, um die Wahrscheinlichkeitsverteilung für das Spracherkennungssystem eng zu halten. Zum Beispiel verlangt die Frage „What city do you want to go to?“ nach einer Antwort, die mit einem stark eingeschränkten Sprachmodell auskommt, während das bei der Frage „How can I help you?“ nicht der Fall ist.

23.5.3 Einen Spracherkenner erstellen

Die Qualität eines Spracherkennungssystems hängt von der Qualität aller seiner Komponenten ab – dem Sprachmodell, den Wort-Aussprachemodellen, den Lautmodellen und den Signalverarbeitungsalgorithmen, mit denen spektrale Merkmale aus dem akustischen Signal extrahiert werden. Wir haben erläutert, wie sich das Sprachmodell aus einem Korpus von geschriebenem Text konstruieren lässt, und wir überlassen die Einzelheiten der Signalverarbeitung anderen Lehrbüchern. Es bleiben noch die Aussprache- und Lautmodelle. Die *Struktur* der Aussprachemodelle – wie zum Beispiel der „tomato“-Modelle in Abbildung 23.17 – wird normalerweise manuell entwickelt. Heutzutage stehen große Aussprachewörterbücher für Englisch und andere Sprachen zur Verfügung, wobei aber deren Genauigkeit stark variiert. Die Struktur der Dreizustands-Lautmodelle ist für alle Laute die gleiche, wie in Abbildung 23.16 gezeigt. Bleiben also lediglich noch die Wahrscheinlichkeiten an sich.

Wie üblich beziehen wir die Wahrscheinlichkeiten aus einem Korpus, und zwar dieses Mal aus einem Korpus von Reden. Die gebräuchlichste Art Korpus umfasst das Sprachsignal für jeden Satz gepaart mit einer Transkription der Wörter. Es ist schwie-

riger, aus diesem Korpus ein Modell zu erstellen, als ein *n*-Gramm-Modell von Text aufzubauen, da wir ein Hidden-Markov-Modell erstellen müssen – die Lautfolge für jedes Wort und der Lautzustand für jeden Zeitrahmen sind versteckte Variablen. In der Pionierzeit der Spracherkennung wurden die versteckten Variablen durch aufwändiges, manuelles Beschriften von Spektrogrammen bereitgestellt. Neuere Systeme stützen sich auf Erwartungs-Maximierung, um die fehlenden Daten automatisch zu liefern. Die Idee ist einfach: Für ein bestimmtes HMM und eine Beobachtungsfolge können wir die Glättungsalgorithmen der *Abschnitte 15.2* und *15.3* heranziehen, um die Wahrscheinlichkeit jedes Zustands bei jedem Zeitschritt und – durch eine einfache Erweiterung – die Wahrscheinlichkeit jedes Zustand/Zustand-Paares in aufeinanderfolgenden Zeitschritten zu berechnen. Diese Wahrscheinlichkeiten können als *unbestimmte Beschriftungen* angesehen werden. Anhand der unbestimmten Beschriftungen können wir neue Übergangs- und Sensorwahrscheinlichkeiten abschätzen und die EM-Prozedur wiederholt sich. Die Methode verbessert garantiert die Übereinstimmung zwischen Modell und Daten bei jeder Iteration und sie konvergiert im Allgemeinen zu einer wesentlich besseren Menge von Parameterwerten als denen, die von den anfänglichen, manuell beschrifteten Abschätzungen geliefert werden.

Die Systeme mit der höchsten Genauigkeit trainieren für jeden Sprecher ein anderes Modell, wobei sie Unterschiede im Dialekt erfassen sowie zwischen männlichen und weiblichen Sprechern unterscheiden und andere Variationen berücksichtigen. Dieses Training kann mehrere Stunden Interaktion mit dem Sprecher erfordern, sodass die Systeme mit der breitesten Anpassung keine sprecherspezifischen Modelle erzeugen.

Die Genauigkeit eines Systems hängt von einer Reihe Faktoren ab. Erstens spielt die Qualität des Signals eine Rolle: Ein hochqualitatives Richtmikrophon an einem Kunstkopf in einem schalldichten Raum schneidet besser ab als ein billiges Mikrophon, das ein Signal über eine Telefonverbindung aus einem Auto mitten im Verkehr bei laufendem Radio überträgt. Ebenso ist das Vokabular entscheidend: Wenn Ziffernfolgen mit einem Vokabular von elf Wörtern (1 bis 9 plus „oh“ und „null“) zu erkennen sind, liegt die Fehlerrate unter 0,5%, während sie auf ungefähr 10% bei Nachrichten mit einem Vokabular von 20.000 Wörtern ansteigt und für einen Korpus mit einem Vokabular von 64.000 Wörtern 20% beträgt. Wichtig ist auch die Aufgabe: Wenn das System versucht, eine spezielle Aufgabe zu realisieren – einen Flug buchen oder den Weg zu einem Restaurant beschreiben –, lässt sich die Aufgabe oftmals perfekt erledigen, selbst mit einer Wortfehlerrate von 10% oder mehr.

Bibliografische und historische Hinweise

Wie die semantischen Netze sind auch kontextfreie Grammatiken (auch als Phrasenstruktur-Grammatiken bezeichnet) eine Neuentdeckung der Technik, die zuerst von alten indischen Grammatikern angewendet wurde (insbesondere Panini, ca. 350 v. Chr.), die sich mit Shastric Sanskrit (Ingerman, 1967) beschäftigten. Sie wurde von Noam Chomsky (1956) für die Analyse der englischen Syntax sowie unabhängig davon von John Backus für die Analyse der Syntax von Algol-58 neu entdeckt. Peter Naur erweiterte die Notation von Backus, was ihm das „N“ in BNF eingebracht hat (Backus, 1996), das ursprünglich für „Backus **N**ormal Form“ stand. Knuth (1968) definierte eine Art erweiterter Grammatik, auch als **Attribut-Grammatik** bezeichnet, die für Programmiersprachen sehr praktisch ist. Die definiten Klausel-Grammatiken wurden von Colmerauer (1975) eingeführt und von Pereira und Shieber (1987) entwickelt und verbreitet.

Probabilistische kontextfreie Grammatiken wurden von Booth (1969) und Salomaa (1969) untersucht. Andere Algorithmen für PCFGs werden in der ausgezeichneten Kurzmonographie von Charniak (1993) sowie in den Lehrbüchern von Manning und Schütze (1999) und von Jurafsky und Martin (2008) dargestellt. Baker (1979) führt den Inside-Outside-Algorithmus für das Lernen einer PCFG ein und Lari und Young (1990) beschreiben dessen Anwendungen und Beschränkungen. Stolcke und Omohundro (1994) zeigen, wie sich Grammatikregeln mit Bayesschen Mischungsmodellen lernen lassen; Haghighi und Klein (2006) beschreiben ein Lernsystem, das auf Prototypen basiert.

Lexikalisierte PCFGs (Charniak, 1997; Hwa, 1998) kombinieren die besten Aspekte von PCFGs und *n*-Gramm-Modellen. Collins (1999) beschreibt PCFG-Parsing, das mit Kopfmerkmalen lexikalisiert ist. Petrov und Klein (2007a) zeigen, wie man die Vorteile der Lexikalisierung ohne die eigentlichen lexikalischen Erweiterungen durch Lernen spezifischer syntaktischer Kategorien aus einer Baumbank, die allgemeine Kategorien enthält, erhalten kann; zum Beispiel umfasst die Baumbank die Kategorie *NP*, von der spezifischere Kategorien wie etwa NP_O und NP_S gelernt werden können.

Es gab viele Versuche, formale Grammatiken für natürliche Sprachen zu schreiben, sowohl in der „reinen" Linguistik als auch in der Computer-Linguistik. Es gibt mehrere umfassende, allerdings nur informelle Grammatiken für Englisch (Quirk et al., 1985; McCawley, 1988; Huddleston und Pullum, 2002). Seit Mitte der 1980er Jahre ist ein Trend zu verzeichnen, mehr Informationen im Lexikon und weniger in der Grammatik unterzubringen. Lexikalisch-funktionale Grammatik (LFG) (Bresnan, 1982) war der erste wichtige Grammatikformalismus, der wesentlich auf einem Lexikon basierte. Wenn wir diese Lexikalisierung auf die Spitze treiben, gelangen wir zu einer **kategorischen Grammatik** (Clark und Curran, 2004), wobei es bis zu zwei Grammatikregeln geben kann, oder zur **Abhängigkeitsgrammatik** (Smith und Eisner, 2008; Kübler et al., 2009), wo es keine syntaktischen Kategorien, sondern nur Relationen zwischen Wörtern gibt. Sleator und Temperley (1993) beschreiben einen Abhängigkeitsparser. Paskin (2001) zeigt, dass eine Version von Abhängigkeitsgrammatik leichter als PCFGs zu lernen ist.

Die ersten computerorientierten Parsing-Algorithmen wurden von Yngve (1955) vorgestellt. Effiziente Algorithmen wurden Ende der 1960er Jahre entwickelt und seither kaum verändert (Kasami, 1964; Younger, 1967; Earley, 1970; Graham et al., 1980). Maxwell und Kaplan (1993) zeigen, wie das Chart-Parsing mit Erweiterungen für den Durchschnittsfall effizient gemacht werden kann. Church und Patil (1982) beschäftigen sich mit der Auflösung syntaktischer Mehrdeutigkeiten. Klein und Manning (2003) beschreiben A*-Parsing und Pauls und Klein (2009) erweitern diesen Algorithmus zum *K*-best A*-Parsing, bei dem man als Ergebnis nicht nur einen einzelnen Parsing-Verlauf, sondern die *K* besten erhält.

Zu den heute führenden Parsern gehören der von Petrov und Klein (2007b), der mit dem Korpus des Wall Street Journal eine Genauigkeit von 90,6% erreichte, der Parser von Charniak und Johnson (2005) mit einer Genauigkeit von 92,0% sowie der von Koo et al. (2008), der es mit der Penn-Baumbank auf 93,2% brachte. Diese Zahlen sind nicht direkt vergleichbar und Kritiker auf diesem Gebiet bemängeln, dass sich die Parser zu stark auf wenige ausgewählte Korpora konzentrieren und dadurch möglicherweise eine Überanpassung entsteht.

Die formale semantische Interpretation natürlicher Sprachen geht ursprünglich auf die Philosophie und die formale Logik zurück, insbesondere die Arbeit von Alfred Tarski (1935) zur Semantik formaler Sprachen. Bar-Hillel (1954) war der Erste, der die Probleme

der Pragmatik betrachtete und vorschlug, dass sie durch die formale Logik verarbeitet werden konnten. Beispielsweise führte er den Begriff *indexical* von C.S. Peirce (1902) in der Linguistik ein. Der Aufsatz „English as a formal language“ (1970) von Richard Montague ist eine Art Manifest für die logische Sprachanalyse, aber die Bücher von Dowty et al. (1991) sowie von Portner und Partee (2002) sind verständlicher geschrieben.

Das erste NLP-System, das reale Aufgaben lösen konnte, war wahrscheinlich das Frage/Antwort-System BASEBALL (Green et al., 1961), das Fragen über eine Datenbank mit Baseballstatistiken verarbeitete. Unmittelbar danach kam LUNAR von Woods (1973), das Fragen zum Mondgestein beantwortete, das vom Apollo-Programm zurückgebracht worden waren. Roger Schank und seine Studenten entwickelten eine Folge von Programmen (Schank und Abelson, 1977; Schank und Riesbeck, 1981), die alle die Aufgabe hatten, Sprache zu verstehen. Moderne Konzepte für die semantische Interpretation gehen davon aus, dass die Abbildung der Syntax auf die Semantik von Beispielen gelernt wird (Zelle und Mooney, 1996; Zettlemoyer und Collins, 2005).

Hobbs et al. (1993) beschreiben ein quantitatives nichtprobabilistisches Framework für die Interpretation. Neuere Arbeiten folgen einem expliziten probabilistischen Framework (Charniak und Goldman, 1992; Wu, 1993; Franz, 1996). In der Linguistik basiert die Optimalitätstheorie (Kager, 1999) auf der Idee, weiche Bedingungen in die Grammatik einzubauen, sodass eine natürliche Reihenfolge der Interpretation (ähnlich einer Wahrscheinlichkeitsverteilung) entsteht, anstatt es der Grammatik zu überlassen, alle Möglichkeiten mit gleichem Rang zu erzeugen. Norvig (1988) beschreibt die Probleme, mehrere gleichzeitige Interpretationen zu betrachten, anstatt nach einer einzigen Interpretation mit maximaler Wahrscheinlichkeit zu suchen. Literaturkritiker (Empson, 1953; Hobbs, 1990) waren sich nicht einig, ob Mehrdeutigkeit etwas ist, was aufgelöst werden oder bewahrt werden soll.

Nunberg (1979) skizziert ein formales Modell der Metonymie. Lakoff und Johnson (1980) zeigen eine engagierte Analyse sowie einen Katalog gebräuchlicher Metaphern aus dem Englischen. Martin (1990) und Gibbs (2006) bieten Berechnungsmodelle für die Metapherinterpretation.

Das erste wichtige Ergebnis zur **Grammatik-Induktion** war negativ: Gold (1967) zeigte, dass es nicht möglich ist, eine korrekte kontextfreie Grammatik für eine bestimmte Menge von Zeichenfolgen aus dieser Grammatik zuverlässig zu lernen. Bekannte Linguisten, wie beispielsweise Chomsky (1957) und Pinker (2003), haben das Ergebnis von Gold genutzt, um zu zeigen, dass es eine angeborene **universelle Grammatik** geben muss, die alle Kinder von Geburt an besitzen. Das sogenannte *Poverty of the Stimulus*-Argument besagt, dass Kinder nicht genügend Eingaben erhalten, um ein CFG zu lernen, sodass sie die Grammatik bereits „kennen“ und lediglich einige ihrer Parameter optimieren müssen. Während dieses Argument die Chomskysche Linguistik immer noch beeinflusst, wurde es von anderen Linguisten (Pullum, 1996; Elman et al., 1997) ebenso verworfen wie von den meisten Informatikern. Schon 1969 zeigte Horning, dass es möglich *ist*, im Sinne des PAC-Lernens eine *probabilistische* kontextfreie Grammatik zu lernen. Seitdem gibt es viele überzeugende empirische Demonstrationen, wie ausschließlich aus positiven Beispielen gelernt werden kann, wie beispielsweise die ILP-Arbeit von Mooney (1999) und Muggleton und De Raedt (1994), das Sequenzlernen von Nevill-Manning und Witten (1997) sowie die bemerkenswerte Doktorarbeit von Schütze (1995) und de Marcken (1996). Es gibt eine jährliche internationale Konferenz zur grammatischen Inferenz (International Conference on Grammatical Inference, ICGI). Es ist

möglich, andere Grammatikformalismen zu lernen, wie beispielsweise reguläre Sprachen (Denis, 2001) und endliche Automaten (Parekh und Honavar, 2001). Abney (2007) gibt als Lehrbuch eine Einführung in teilüberwachtes Lernen für Sprachmodelle.

Wordnet (Fellbaum, 2001) ist ein öffentlich verfügbares Wörterbuch mit etwa 100.000 Wörtern und Phrasen, in Sprachbestandteile unterteilt und durch semantische Relationen wie etwa Synonym, Antonym oder Teil-von verknüpft. Die Penn-Baumbank (Marcus et al., 1993) bietet Syntaxbäume für einen englischen Korpus mit 3 Millionen Wörtern. Charniak (1996) sowie Klein und Manning (2001) diskutieren Parsing mit Baumbank-Grammatiken. Der British National Corpus (Leech et al., 2001) enthält 100 Millionen und das World Wide Web mehrere Billionen Wörter; (Brants et al., 2007) beschreiben *n*-Gramm-Modelle über einem Web-Korpus mit 2 Billionen Wörtern.

In den 1930er Jahren meldete Petr Troyanskii ein Patent für eine „Übersetzungsmaschine“ an, doch gab es noch keine Computer, um seine Ideen zu realisieren. Im März 1947 äußerte Warren Weaver von der Rockefeller-Stiftung in einem Schreiben an Norbert Wiener, dass er eine maschinelle Übersetzung für möglich hielt. Aufbauend auf Arbeiten in Kryptografie und Informationstheorie schrieb Weaver: „Wenn ich mir einen Artikel in Russisch ansehe, sage ich mir: 'Dies ist eigentlich in Englisch geschrieben, jedoch mit seltsamen Symbolen kodiert. Ich werde nun darangehen, es zu dekodieren.'“ In den nächsten zehn Jahren versuchte die Community, auf diese Art und Weise zu dekodieren. IBM stellte 1954 ein rudimentäres System vor. Bar-Hillel (1960) beschreibt den Enthusiasmus dieser Periode. Allerdings berichtete die US-Regierung später (ALPAC, 1966), dass „es keine unmittelbare oder vorhersehbare Aussicht auf eine brauchbare maschinelle Übersetzung gibt“. Die Arbeiten gingen jedoch in begrenztem Umfang weiter und seit den 1980er Jahren hatte die Rechenleistung bis zu einem Punkt zugenommen, wo die ALPAC-Feststellungen nicht mehr richtig waren.

Der in diesem Kapitel beschriebene grundlegende statistische Ansatz basiert auf frühen Arbeiten der IBM-Gruppe (Brown et al., 1988, 1993) und neueren Arbeiten der ISI- und Google-Forschungsgruppen (Och und Ney, 2004; Zollmann et al., 2008). Eine lehrbuchartige Einführung zur statischen maschinellen Übersetzung gibt Koehn (2009), ein einflussreiches kürzeres Tutorial kommt von Kevin Knight (1999). Frühe Arbeiten zur Satzsegmentierung wurden von Palmer und Hearst (1994) geleistet. Och und Ney (2003) sowie Moore (2005) behandeln zweisprachige Satzausrichtung.

Die Vorgeschichte der Spracherkennung begann in den 1920er Jahren mit Radio Rex, einem durch die Stimme aktivierten Spielzeughund. Als Reaktion auf das Wort „Rex!“ (oder genau genommen einem genügend laut gerufenen Wort) sprang Rex aus seiner Hundehütte heraus. Die Zeit nach dem Zweiten Weltkrieg ist durch ernsthaftere Arbeiten geprägt. Bei AT&T Bell Labs wurde ein System für die Erkennung einzelner Ziffern durch simplen Mustervergleich akustischer Merkmale entwickelt (Davis et al., 1952). Ab 1971 hat das Defense Advanced Research Projects Agency (DARPA) des US-Verteidigungsministeriums vier Fünfjahresprojekte initiiert, um Hochleistungs-Spracherkennungssysteme zu entwickeln. Der Gewinner und das einzige System, das das Ziel von 90% Genauigkeit mit einem Vokabular von 1000 Wörtern erreicht hatte, war das HARPY-System von der CMU (Lowerre und Reddy, 1980). Die endgültige Version von HARPY geht auf das System DRAGON zurück, das der CMU-Doktorand James Baker (1975) erstellt hat; DRAGON war die erste Anwendung von HMMs für Sprache. Nahezu gleichzeitig hatte Jelinek (1976) bei IBM ein anderes HMM-basiertes System entwickelt. Die letzten Jahre sind durch einen kontinuierlichen Fortschritt, größere Datenmengen und

Modelle sowie strengere Wettbewerbe zu realistischeren Sprachaufgaben charakterisiert. Im Jahr 1997 sagte Bill Gates voraus, dass man den PC in fünf Jahren nicht wiedererkennen würde, da er mit einer Sprachschnittstelle ausgerüstet sein wird. Das traf zwar nicht ganz zu, doch gemäß seiner Voraussage von 2008 erwartet Microsoft, dass in fünf Jahren mehr Internetsuchen über Sprache als durch Tastatureingaben erfolgen werden. Die Geschichte wird uns zeigen, ob Gates dieses Mal richtig liegt.

Zum Thema Spracherkennung gibt es mehrere gute Lehrbücher (Rabiner und Juang, 1993; Jelinek, 1997; Gold und Morgan, 2000; Huang et al., 2001). Die Darstellung in diesem Kapitel stützt sich auf den Überblick von Kay, Gawron und Norvig (1994) sowie auf das Lehrbuch von Jurafsky und Martin (2008). Forschungsergebnisse zur Spracherkennung werden in *Computer Speech and Language*, *Speech Communications* und in *IEEE Transactions on Acoustics, Speech, and Signal Processing* sowie in den *Speech and Natural Language Processing*-Workshops der DARPA und in den Konferenzen Eurospeech, ICSLP und ASRU veröffentlicht.

Ken Church (2004) zeigt, dass die Forschung zur Verarbeitung natürlicher Sprache zwischen der Konzentration auf die Daten (Empirismus) und der Konzentration auf Theorien (Rationalismus) hin- und hergependelt ist. Der Linguist John Firth (1957) proklamierte: „You shall know a word by the company it keeps“ (sinngemäß: „Die Bedeutung eines Wortes erschließt sich aus dem Kontext“), und die Linguistik der 1940er und frühen 1950er Jahre stützte sich hauptsächlich auf Worthäufigkeiten, wenn auch ohne die rechentechnische Leistung, die heute zur Verfügung steht. Noam (Chomsky, 1956) zeigte dann die Beschränkungen der Modelle mit endlichen Automaten auf und weckte das Interesse an theoretischen Untersuchungen der Syntax ohne Berücksichtigung von Häufigkeitszählungen. Dieser Ansatz war zwanzig Jahre lang vorherrschend, bis der Empirismus sein Comeback basierend auf Arbeiten in statistischer Spracherkennung feierte (Jelinek, 1976). Heute akzeptieren die meisten Arbeiten das statistische Gerüst, doch besteht ein großes Interesse daran, statistische Modelle zu erstellen, die Modelle auf höherer Ebene berücksichtigen, beispielsweise Syntaxbäume und semantische Relationen statt reiner Wortfolgen.

Die Arbeiten zu Anwendungen der Sprachverarbeitung werden auf der zweijährlichen Konferenz *Applied Natural Language Processing* (ANLP), der Konferenz *Empirical Methods in Natural Language Processing* (EMNLP) und im Journal *Natural Language Engineering* präsentiert. Ein breiter Bereich der NLP-Arbeiten erscheint im Journal *Computational Linguistics* und dessen Konferenz ACL (Association for Computational Linguistics) sowie in der Konferenz *Computational Linguistics* (COLING).

Zusammenfassung

Das Verstehen natürlicher Sprache ist eines der wichtigsten Teilgebiete der KI. Im Unterschied zu anderen Bereichen der KI erfordert das Verstehen natürlicher Sprache eine empirische Untersuchung des realen menschlichen Verhaltens – was sich als komplex und interessant herausstellt.

- Die formale Sprachtheorie sowie **Phrasenstruktur**-Grammatiken (und insbesondere **kontextfreie** Grammatiken) sind praktische Werkzeuge für den Umgang mit bestimmten Aspekten der natürlichen Sprache. Der Formalismus der probabilistischen kontextfreien Grammatik (PCFG) ist in vielen Bereichen gebräuchlich.
- Sätze in einer kontextfreien Sprache können in der Zeit $O(n^3)$ durch einen **Chart-Parser** wie zum Beispiel den **CYK-Algorithmus** geparst werden. Hierfür müssen die Grammatikregeln in der **Chomsky-Normalform** vorliegen.
- Eine **Baumbank** ist geeignet, um eine Grammatik zu lernen. Es ist auch möglich, eine Grammatik aus einem nicht geparsten Korpus von Sätzen zu lernen, doch ist dies weniger erfolgreich.
- Mit einer **lexikalisierten PCFG** können wir darstellen, dass bestimmte Beziehungen zwischen Wörtern gebräuchlicher sind als andere.
- Es ist praktisch, eine Grammatik so zu **erweitern**, dass sie Probleme verarbeitet wie beispielsweise die Subjekt-Verb-Kongruenz oder den Pronomenfall. Die **definite Klausel-Grammatik** (DCG) ist ein Formalismus, der Erweiterungen unterstützt. Mithilfe von DCG können Parsing und semantische Interpretation (und sogar Erzeugung) unter Verwendung logischer Inferenz erfolgen.
- Auch die **semantische Interpretation** kann mithilfe einer erweiterten Grammatik verarbeitet werden.
- **Mehrdeutigkeit** ist ein sehr wichtiges Problem beim Verstehen natürlicher Sprache; die meisten Sätze erlauben viele mögliche Interpretationen, normalerweise ist nur eine davon richtig. Die Auflösung der Mehrdeutigkeit basiert auf dem Wissen über die Welt, über die aktuelle Situation und den Sprachgebrauch.
- **Maschinelle Übersetzungssysteme** sind mit einer ganzen Reihe von Techniken realisiert worden, angefangen bei vollständiger syntaktischer und semantischer Analyse bis hin zu statistischen Verfahren, die auf Phrasenhäufigkeiten basieren. Derzeit haben sich die statistischen Modelle durchgesetzt und sind am erfolgreichsten.
- **Spracherkennungssysteme** beruhen ebenfalls hauptsächlich auf statistischen Prinzipien. Sprachsysteme sind populär und nützlich, wenn auch noch unvollkommen.
- Maschinelle Übersetzung und Spracherkennung gehören zu den großen Erfolgen bei der Verarbeitung natürlicher Sprache. Die hohe Leistungsfähigkeit der Modelle geht zum Teil auf die Verfügbarkeit großer Korpora zurück – sowohl Übersetzung als auch Sprache sind Aufgaben, die jeden Tag „in freier Wildbahn" von Menschen ausgeführt werden. Im Unterschied dazu sind Aufgaben wie das Parsing von Sätzen noch wenig erfolgreich. Das hängt zum Teil damit zusammen, dass keine großen Korpora von geparsten Sätzen „in freier Wildbahn" verfügbar sind, und zum Teil, weil Parsing an und für sich nicht nützlich ist.

Übungen zu Kapitel 23

Lösungshinweise

1 Lesen Sie den folgenden Text einmal und versuchen Sie, sich so viel davon zu merken wie möglich. Später gibt es einen Test dazu.

Die Vorgehensweise ist wirklich ganz einfach. Zuerst ordnen Sie Dinge in unterschiedlichen Gruppen an. Natürlich könnte ein Stapel ausreichend sein, abhängig davon, wie viel zu tun ist. Wenn Sie aufgrund fehlender Einrichtungen an einen anderen Ort gehen müssen, ist dies der nächste Schritt, andernfalls sind Sie fertig. Es ist wichtig, nicht zu viel zu tun. Das bedeutet, es ist besser, wenige Dinge gleich zu machen als zu viele. Auf kurze Sicht scheint das nicht wichtig zu sein, aber es können schnell Komplikationen auftreten. Außerdem ist ein Fehler sehr teuer. Zunächst scheint die gesamte Prozedur sehr kompliziert zu sein. Bald jedoch wird sie zu einem ganz normalen Teil Ihres Lebens. Es ist schwierig, die Notwendigkeit für diese Aufgabe in der unmittelbaren Zukunft abzusehen, aber man weiß nie. Nachdem die Prozedur abgeschlossen ist, ordnen Sie das Material wieder in unterschiedlichen Gruppen an. Dann können Sie es an den jeweiligen Plätzen einordnen. Irgendwann wird es wieder verwendet und der gesamte Zyklus muss wiederholt werden. Dies gehört jedoch zum Leben.

2 Eine *HMM-Grammatik* ist im Wesentlichen ein Standard-HMM, dessen Zustandsvariable N (Nichtterminal, mit Werten wie Det, $Adjektiv$, $Substantiv$ usw.) und dessen Evidenzvariable W (Wort, mit Werten wie *is*, *duck* usw.) ist. Das HMM-Modell umfasst ein A-priori-Modell $\mathbf{P}(N_0)$, ein Übergangsmodell $\mathbf{P}(N_{t+1} \mid N_t)$ und ein Sensormodell $\mathbf{P}(W_t \mid N_t)$. Zeigen Sie, dass sich jede HMM-Grammatik als eine PCFG schreiben lässt. (Hinweis: Überlegen Sie zuerst, wie das A-priori-HMM durch PCFG-Regeln für das Satzsymbol dargestellt werden kann. Es ist vielleicht hilfreich, ein konkretes HMM mit den Werten *A*, *B* für N und den Werten *x*, *y* für W grafisch zu veranschaulichen.)

3 Sehen Sie sich die folgende einfache PCFG für Substantivphrasen an:

0,6 : $NP \rightarrow Det\ AdjFolge\ Substantiv$

0,4 : $NP \rightarrow Det\ SubstantivSubstantivVerbindung$

0,5 : $AdjFolge \rightarrow Adj\ AdjFolge$

0,5 : $AdjFolge \rightarrow \Lambda$

1,0 : $SubstantivSubstantivVerbindung \rightarrow Substantiv\ Substantiv$

0,8 : $Det \rightarrow$ **the**

0,2 : $Det \rightarrow$ **a**

0,5 : $Adj \rightarrow$ **small**

0,5 : $Adj \rightarrow$ **green**

0.6 : $Substantiv \rightarrow$ **village**

0.4 : $Substantiv \rightarrow$ **green**

wobei das Symbol Λ die leere Zeichenfolge kennzeichnet.

a. Welche Größe hat die längste Kategorie NP, die sich durch diese Grammatik erzeugen lässt? (i) drei Wörter, (ii) vier Wörter, (iii) unendlich viele Wörter

b. Welche der folgenden Phrasen werden mit einer Wahrscheinlichkeit ungleich null als vollständige NPs erzeugt? (i) a small green village, (ii) a green green green, (iii) a small village green

c. Wie groß ist die Wahrscheinlichkeit, dass „the green green" erzeugt wird?

d. Welche Arten der Mehrdeutigkeit werden durch die Phrase in (c) gezeigt?

e. Lässt sich für eine PCFG und eine endliche Wortfolge die Wahrscheinlichkeit berechnen, dass die Sequenz durch die PCFG generiert wurde?

4 Skizzieren Sie die wichtigsten Unterschiede zwischen Java (oder einer anderen Computersprache, die Sie kennen) und der natürlichen Sprache. Gehen Sie dabei für jeden Fall auf das „Verständnisproblem" ein. Denken Sie über Dinge wie etwa Grammatik, Syntax, Semantik, Pragmatik, Zerlegbarkeit, Kontextabhängigkeit, lexikalische Mehrdeutigkeit, syntaktische Mehrdeutigkeit, Verweisfindung (einschließlich Pronomen) und Hintergrundwissen nach und was es überhaupt heißt zu „verstehen".

5 In dieser Übung geht es um Grammatiken für sehr einfache Sprachen.

a. Schreiben Sie eine kontextfreie Grammatik für die Sprache $a^n b^n$.

b. Schreiben Sie eine kontextfreie Grammatik für die Palindromsprache: die Menge aller Zeichenketten, deren zweite Hälfte die Umkehrung der ersten Hälfte ist.

c. Schreiben Sie eine kontextsensitive Grammatik für die Duplikatsprache: die Menge aller Zeichenketten, deren zweite Hälfte dieselbe wie die erste Hälfte ist.

6 Betrachten Sie den Satz „Jemand ging langsam zu dem Supermarkt" und ein Lexikon, das aus den folgenden Wörtern besteht:

Pronomen → **Jemand**	*V* → **ging**
Adverb → **langsam**	*Präposition* → **zu**
Artikel → **dem**	*Substantiv* → **Supermarkt**

Welche der drei folgenden Grammatiken erzeugt in Kombination mit dem Lexikon den vorgegebenen Satz? Zeigen Sie die entsprechenden Parse-Bäume.

(A):	(B):	(C):
S → *NP VP*	*S* → *NP VP*	*S* → *NP Vp*
NP → *Pronomen*	*NP* → *Pronomen*	*NP*→ *Pronomen*
NP → *Artikel Substantiv*	*NP*→ *Substantiv*	*NP*→ *Artikel NP*
VP → *VP PP*	*NP* → *Artikel NP*	*VP* → *Verb Adverb*
VP → *VP Adverb Adverb*	*VP* → *Verb Vmod*	*Adverb* → *Adverb Adverb*
VP → *Verb*	*Vmod* → *Adverb Vmod*	*Adverb* → *PP*
PP → *Präposition NP*	*Vmod* → *Adverb*	*PP* → *Präposition NP*
NP → *Substantiv*	*Adverb* → *PP*	*NP* → *Substantiv*
	PP → *Präposition NP*	

Schreiben Sie für jede dieser Grammatiken drei deutsche Sätze auf, ebenso wie drei nichtdeutsche Sätze, die die Grammatik erzeugt. Die Sätze sollten sich wesentlich voneinander unterscheiden, mindestens sechs Wörter lang sein und einige neue lexikalische Einträge (die Sie definieren) enthalten. Schlagen Sie Möglichkeiten vor, jede Grammatik zu verbessern, um die erzeugten nichtdeutschen Sätze zu vermeiden.

7 Verschiedene Linguisten argumentierten wie folgt:

Kinder, die eine Sprache lernen, hören nur *positive Beispiele* der Sprache und keine *negativen Beispiele*. Demzufolge ist die Hypothese, dass „jeder mögliche Satz in der Sprache enthalten ist" konsistent mit allen beobachteten Beispielen. Darüber hinaus ist dies die einfachste konsistente Hypothese. Des Weiteren sind alle Grammatiken für Sprachen, die Obermengen der echten Sprache darstellen, ebenso konsistent mit den beobachteten Daten. Kinder leiten sogar (mehr oder weniger) die richtige Grammatik ab. Es folgt, dass sie mit sehr strengen angeborenen grammatikalischen Beschränkungen beginnen, die a priori alle diese allgemeineren Hypothesen verdrängen.

Erörtern Sie kurz die Schwachpunkte in dieser Argumentation entsprechend Ihrem Wissen über statistisches Lernen.

8 In dieser Übung transformieren Sie ε_0 in die Chomsky Normal Form (CNF). Es gibt fünf Schritte: (a) Hinzufügen eines neuen Startsymbols, (b) Eliminieren von $\in$-Regeln, (c) Eliminieren von mehreren Wörtern auf den rechten Seiten, (d) Eliminieren von Regeln der Form $(X \rightarrow Y)$, (e) Konvertieren von langen linken Seiten zu binären Regeln.

a. Das Startsymbol S kann in CNF nur auf der linken Seite auftreten. Fügen Sie eine neue Regel der Form $S' \rightarrow S$ mithilfe eines neuen Symbols S' hinzu.

b. Die leere Zeichenfolge $\in$ darf in CNF nicht auf der rechten Seite erscheinen. Dies stellt kein Problem dar, da ε_0 keine Regeln mit $\in$ besitzt.

c. Ein Wort kann auf der rechten Seite in einer Regel nur in der Form $(X \rightarrow wort)$ erscheinen. Ersetzen Sie jede Regel der Form $(X \rightarrow \ldots\ wort\ \ldots)$ durch $(X \rightarrow \ldots W' \ \ldots)$ und $(W' \rightarrow wort)$ mithilfe eines neuen Symbols W'.

d. Eine Regel $(X \rightarrow Y)$ ist in CNF nicht erlaubt; sie muss $(X \rightarrow Y\,Z)$ oder $(X \rightarrow wort)$ lauten. Ersetzen Sie jede Regel der Form $(X \rightarrow Y)$ durch eine Menge von Regeln der Form $(X \rightarrow \ldots)$ mit jeweils einer für jede Regel $(Y \rightarrow \ldots)$, wobei $(\ldots)$ ein oder mehrere Symbole kennzeichnet.

e. Ersetzen Sie jede Regel der Form $(X \rightarrow Y\ Z\ \ldots)$ durch zwei Regeln, $(X \rightarrow Y\ Z')$ und $(Z' \rightarrow Z\ \ldots)$, wobei Z' ein neues Symbol ist.

Zeigen Sie jeden Schritt des Prozesses und die endgültige Regelmenge.

9 Betrachten Sie die folgende sehr einfache Grammatik:

$S \rightarrow NP\ VP$
$NP \rightarrow Substantiv$
$NP \rightarrow NP$ **and** NP
$NP \rightarrow NP\ PP$
$VP \rightarrow Verb$
$VP \rightarrow VP$ **and** VP
$VP \rightarrow VP\ PP$
$PP \rightarrow Präposition\ NP$

$Substantiv \rightarrow$ **Sally** | **pools** | **streams** | **swims**
$Präposition \rightarrow$ **in**
$Verb \rightarrow$ **pools** | **streams** | **swims**

a. Zeigen Sie alle Syntaxbäume in dieser Grammatik für den Satz „Sally swims in streams and pools".

b. Zeigen Sie alle Tabelleneinträge, die ein (nichtprobabilistischer) CYK-Parser für diesen Satz erstellen würde.

10 Schreiben Sie unter Verwendung der DCG-Notation eine Grammatik für eine Sprache, die genau wie ε_1 ist, außer dass sie eine Übereinstimmung zwischen dem Subjekt und dem Verb eines Satzes erzwingt und damit keine grammatikalisch falschen Sätze wie zum Beispiel „I smells the Wumpus" erzeugt.

11 Eine erweiterte kontextfreie Grammatik kann Sprachen darstellen, wozu eine normale kontextfreie Grammatik nicht in der Lage ist. Zeigen Sie eine erweiterte kontextfreie Grammatik für die Sprache $a^n b^n c^n$. Die zulässigen Werte für Erweiterungsvariablen sind 1 und SUCCESSOR(n), wobei n ein Wert ist. Die Regel für einen Satz in dieser Sprache lautet:

$$S(n) \rightarrow A(n)\ B(n)\ C(n).$$

Zeigen Sie die Regel(n) für jede der Phrasen A, B und C.

12 Betrachten Sie den folgenden Satz (aus *The New York Times* vom 28. Juli 2008):

Banks struggling to recover from multibillion-dollar loans on real estate are curtailing loans to American businesses, depriving even healthy companies of money for expansion and hiring.

a. Welche Wörter in diesem Satz sind lexikalisch mehrdeutig?

b. Suchen Sie in diesem Satz zwei Fälle für syntaktische Mehrdeutigkeit (es gibt mehr als zwei).

c. Geben Sie ein Beispiel für eine Metapher in diesem Satz an.

d. Können Sie eine semantische Mehrdeutigkeit finden?

13 Beantworten Sie die folgenden Fragen, ohne noch einmal in Übung 23.1 nachzulesen:

a. Welche vier Schritte wurden erwähnt?

b. Welcher Schritt wurde ausgelassen?

c. Was ist das im Text erwähnte „Material"?

d. Welche Art Fehler wäre teuer?

e. Ist es besser, wenig oder viel zu tun? Warum?

14 Wählen Sie fünf Sätze aus und senden Sie sie an einen Online-Übersetzungsdienst. Übersetzen Sie sie vom Deutschen in eine andere Sprache und wieder zurück ins Deutsche. Bewerten Sie die Ergebnissätze in Bezug auf Grammatikalität und Bewahrung der Bedeutung. Wiederholen Sie diesen Prozess. Liefert der zweite Durchlauf schlechtere oder die gleichen Ergebnisse? Wirkt sich die Auswahl einer Zwischensprache auf die Qualität der Ergebnisse aus? Sehen Sie sich für eine Ihnen bekannte Fremdsprache die Übersetzung eines Absatzes in dieser Sprache an. Zählen und beschreiben Sie die aufgetretenen Fehler und stellen Sie Vermutungen an, warum diese Fehler gemacht wurden.

15 Die Summe der D_i-Werte für den Satz in Abbildung 23.13 ist null. Gilt diese Aussage für jedes Übersetzungspaar? Beweisen Sie dies oder geben Sie ein Gegenbeispiel an.

16 (Nach Knight (1999)) Unser Übersetzungsmodell geht davon aus, dass das Sprachmodell die Permutation entflechten kann, nachdem das Übersetzungsmodell Phrasen ausgewählt und das Verzerrungsmodell diese permutiert hat. Diese Übung untersucht, wie sinnvoll diese Annahme ist. Versuchen Sie, die angegebenen Phrasen in die richtige Reihenfolge zu bringen:

a. have, programming, a, seen, never, I, language, better

b. loves, john, mary

c. is the, communication, exchange of, intentional, information brought, by, about, the production, perception of, and signs, from, drawn, a, of, system, signs, conventional, shared

d. created, that, we hold these, to be, all men, truths, are, equal, self-evident

Welche Teilaufgabe konnten Sie lösen? Auf welche Art Wissen haben Sie Ihre Lösung gestützt? Trainieren Sie ein Bigramm-Modell aus einem Trainingskorpus und verwenden Sie es, um die Permutation mit der höchsten Wahrscheinlichkeit für einige Sätze aus einem Testkorpus zu finden. Geben Sie die Genauigkeit dieses Modells an.

17 Berechnen Sie den wahrscheinlichsten Pfad durch das HMM in Abbildung 23.16 für die Ausgabesequenz $[C_1, C_2, C_3, C_4, C_4, C_6, C_7]$. Geben Sie auch dessen Wahrscheinlichkeit an.

18 Wir vergaßen zu erwähnen, dass der Text in Übung 23.1 die Überschrift „Wäschewaschen" trägt. Lesen Sie den Text erneut und beantworten Sie die Fragen aus Übung 23.13. Konnten Sie es diesmal besser? Bransford und Johnson (1973) verwendeten diesen Text in einem überwachten Experiment und stellten fest, dass der Titel eine wesentliche Hilfe darstellte. Was sagt Ihnen das über das Verständnis von Gesprächen?

Wahrnehmung

24

ÜBERBLICK

In diesem Kapitel verbinden wir den Computer mit der reinen, unverfälschten Welt.

Die **Wahrnehmung** bietet den Agenten Informationen über die Welt, in der sie sich bewegen, indem sie die Reaktionen von **Sensoren** interpretieren. Ein Sensor misst einen bestimmten Aspekt der Umgebung in einer Form, die sich als Eingabe für ein Agentenprogramm eignet. Der Sensor kann so einfach wie ein Schalter sein, bei dem sich die Zustände *an* und *aus* mit 1 Bit kodieren lassen, oder so komplex wie das Auge. Für künstliche Agenten steht eine breite Palette von Sensormodalitäten zur Verfügung. Zu den Sinnen, die auch beim Menschen vorhanden sind, gehören Sehen, Hören und Fühlen. Dem Menschen ohne Hilfsmittel nicht zugängliche Modalitäten sind zum Beispiel Funk-, Infrarot- und GPS-Signale. Manche Roboter arbeiten mit **aktiven Sensoren**, d.h., sie senden Signale aus, wie beispielsweise Radar- oder Ultraschallwellen, und nehmen die von der Umgebung reflektierten Signale auf. Anstatt aber alle diese Arten zu behandeln, konzentriert sich dieses Kapitel ausführlich auf eine Modalität: das Sehen.

Wie für partiell beobachtbare MEPs (*Abschnitt 17.4*) erläutert, verfügt ein modellbasierter entscheidungstheoretischer Agent über ein **Sensormodell** – eine Wahrscheinlichkeitsverteilung $\mathbf{P}(E \mid S)$ über der Evidenz, die von seinen Sensoren kommt, für einen bestimmten Zustand der Welt. Mithilfe der Bayesschen Regel lässt sich dann die Bewertung des Zustandes aktualisieren.

Für das Sehen kann das Sensormodell in zwei Komponenten aufgeteilt werden: Ein **Objektmodell** beschreibt die Objekte, die die sichtbare Welt bevölkern – Menschen, Gebäude, Bäume, Autos usw. Beim Objektmodell könnte es sich um ein genaues 3D-Geometriemodell handeln, das von einem CAD (Computer Aided Design)-System kommt, oder um unscharfe Bedingungen, wie zum Beispiel die Tatsache, dass menschliche Augen normalerweise 5 bis 7 cm voneinander entfernt sind. Ein **Rendering-Modell** (Bildsynthesemodell) beschreibt die physikalischen, geometrischen und statistischen Prozesse, die den Reiz aus der Welt erzeugen. Rendering-Modelle sind recht genau, aber mehrdeutig. Zum Beispiel kann ein weißes Objekt bei ungünstigen Lichtverhältnissen in der gleichen Farbe wie ein schwarzes Objekt unter starker Lichteinstrahlung erscheinen. Ein nahes Objekt kann genauso wie ein weit entferntes Objekt aussehen. Ohne zusätzliche Evidenz können wir nicht beurteilen, ob das Bild, das den Rahmen ausfüllt, ein Spielzeug-Godzilla oder ein wirkliches Monster ist.

Mehrdeutigkeit lässt sich mit A-priori-Wissen in den Griff bekommen – da wir wissen, dass Godzilla nicht real ist, muss das Bild ein Spielzeug zeigen – oder indem wir selektiv die Mehrdeutigkeit herausfiltern. Das visuelle System eines autonomen Fahrzeuges wird möglicherweise nicht in der Lage sein, weit entfernte Objekte zu interpretieren, doch kann der Agent entscheiden, das Problem zu ignorieren, da es unwahrscheinlich ist, mit einem meilenweit entfernten Objekt zusammenzustoßen.

Ein entscheidungstheoretischer Agent ist nicht die einzige Architektur, die mit Bildsensoren arbeiten kann. Beispielsweise sind Taufliegen (Gattung *Drosophila*) zum Teil Reflexagenten: Zervikale Riesenfasern bilden eine direkte Verbindung von ihrem visu-

ellen System zu den Flugmuskeln, die eine Fluchtreaktion auslösen – eine unmittelbare Reaktion, ohne jegliche „Überlegung“. Fliegen und viele andere fliegende Tiere verwenden einen Regelkreis, um auf einem Objekt zu landen. Das visuelle System extrahiert eine Entfernungsschätzung zum Objekt und das Regelsystem passt die Flugmuskeln entsprechend an, was sehr schnelle Richtungsänderungen erlaubt, ohne dass ein detailliertes Objektmodell vorhanden sein muss.

Tipp

Verglichen mit den Daten von anderen Sensoren (wie zum Beispiel dem einzelnen Bit, an dem der Staubsaugerroboter erkennt, ob er gegen eine Wand gestoßen ist), sind visuelle Beobachtungen außerordentlich umfangreich, sowohl im Hinblick auf die erkennbaren Details als auch die Unmasse von erzeugten Daten. Eine Videokamera für Roboteranwendungen liefert beispielsweise eine Million 24-Bit-Pixel bei 60 Hz, was einer Rate von 10 GB pro Minute entspricht. Ein Bild verarbeitender Agent hat dann folgendes Problem zu lösen: *Anhand welcher Aspekte der umfangreichen visuellen Reize kann der Agent Entscheidungen für gute Aktionen treffen und welche Aspekte sollte er ignorieren?* Sehen – und Wahrnehmung insgesamt – ist kein Selbstzweck, sondern dient dem Agenten dazu, seine Ziele besser zu verwirklichen.

Das Problem lässt sich mit drei breit angelegten Strategien charakterisieren. Die **Merkmalsextraktion**, wie sie von *Drosophila* gezeigt wird, stützt sich vor allem auf einfache Berechnungen, die direkt auf die Sensorbeobachtungen angewandt werden. Beim Konzept der **Erkennung** unterscheidet ein Agent zwischen den Objekten, auf die er trifft, anhand von visuellen und anderen Informationen. Erkennung könnte bedeuten, jedes Bild mit *Ja* oder *Nein* zu beschriften, um anzugeben, ob es Futter, das wir suchen sollen, oder das Gesicht von Großmutter enthält. Schließlich erstellt ein Agent im Konzept der **Rekonstruktion** ein geometrisches Modell der Welt von einem Bild oder einem Satz von Bildern.

Die Forschung hat in den letzten dreißig Jahren leistungsfähige Werkzeuge und Methoden für diese Ansätze hervorgebracht. Um diese Methoden zu verstehen, muss man die Prozesse kennen, durch die Bilder erzeugt werden. Deshalb behandeln wir jetzt die physikalischen und statistischen Phänomene, die in der Bilderzeugung auftreten.

24.1 Bildaufbau

Imaging (Bildgebung) verzerrt die Erscheinung von Objekten. Zum Beispiel vermittelt ein Bild, das entlang einer geraden Eisenbahnstrecke aufgenommen wurde, den Eindruck, dass die Schienen aufeinander zulaufen und sich schließlich treffen. Oder wenn Sie die Hand vor das Auge halten, können Sie den Mond abdecken, der nicht kleiner als Ihre Hand ist. Und wenn Sie die Hand vor- und zurückbewegen oder neigen, scheint sie *im Bild* kleiner oder größer zu werden, was aber in der Realität nicht der Fall ist (siehe ▶ Abbildung 24.1). Modelle dieser Effekte sind sowohl für Erkennung als auch Rekonstruktion wichtig.

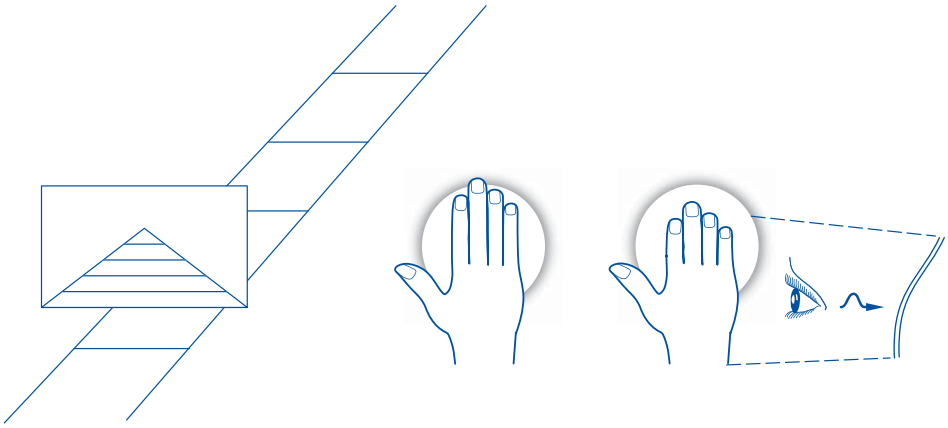


Abbildung 24.1: Imaging verzerrt die Geometrie. Parallele Linien scheinen sich in der Entfernung zu treffen, wie beim Bild der Eisenbahnschienen auf der linken Seite. In der Mitte verdeckt eine kleine Hand den größten Teil eines großen Mondes. Rechts ist ein perspektivischer Verkürzungseffekt dargestellt: Die Hand wird vom Auge weg geneigt, wodurch sie kürzer erscheint als in der mittleren Abbildung.

24.1.1 Bilder ohne Linsen: die Lochkamera

Bildsensoren erfassen Licht, das von Objekten in einer **Szene** gestreut wird, und erzeugen ein zweidimensionales **Bild**. Im Auge entsteht das Bild auf der Retina (Netzhaut), die aus zwei Arten von Zellen besteht: rund 100 Millionen Stäbchen, die für einen weiten Wellenlängenbereich empfindlich sind, und 5 Millionen Zapfen. Die für die Farbwahrnehmung wichtigen Zapfen existieren vor allem in drei Arten, die für verschiedene Wellenlängenbereiche empfindlich sind. In Kameras entsteht das Bild auf einer Bildebene. Dabei kann es sich um einen mit Silberhalogeniden beschichteten Film handeln oder um ein rechteckiges Raster mit einigen Millionen fotoempfindlichen **Pixeln** auf der Basis von CMOS- oder CCD-Bauelementen[1]. Jedes Photon, das am Sensor ankommt, ruft eine Wirkung hervor, deren Stärke von der Wellenlänge des Photons abhängig ist. Die Ausgabe des Sensors ist die Summe aller Wirkungen, die durch die in einem bestimmten Zeitfenster beobachteten Photonen hervorgerufen werden. Bildsensoren melden also einen gewichteten Mittelwert für die Intensität des Lichts, das auf den Sensor fällt.

Um ein scharfes Bild zu sehen, müssen alle Photonen von ungefähr dem gleichen Punkt in der Szene an ungefähr dem gleichen Punkt in der Bildebene eintreffen. Die einfachste Möglichkeit hierzu ist die Betrachtung stationärer Objekte mit einer **Lochkamera**. Diese besteht aus einer Öffnung O in der Vorderseite eines Kastens und einer Bildebene an dessen Rückseite (siehe ▶ Abbildung 24.2). Photonen aus der Szene müssen das Loch passieren. Ist das Loch genügend klein, gelangen nahegelegene Photonen aus der Szene zu nahegelegenen Positionen in der Bildebene und es entsteht ein scharfes Bild.

1 CMOS = Complementary Metal Oxide Semiconductor, komplementärer Metall-Oxid-Halbleiter; CCD = Charge-Coupled Device, ladungsgekoppeltes Bauteil (auch: Eimerkettenspeicher)

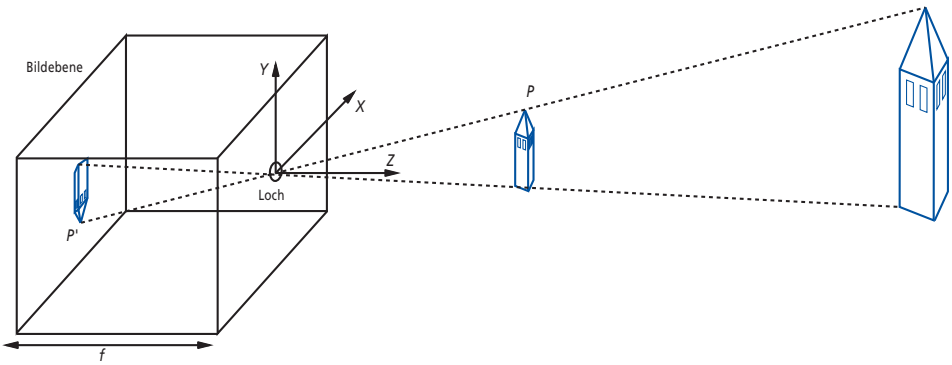


Abbildung 24.2: Jedes lichtempfindliche Element in der Bildebene an der Rückseite einer Lochkamera empfängt Licht aus dem kleinen Richtungsbereich, der das Loch passiert. Ist das Loch genügend klein, entsteht ein scharfes Bild an der Rückseite der Lochkamera. Eine derartige Projektion bedeutet, dass große, weit entfernte Objekte genauso wie kleinere, nahegelegene Objekte aussehen. Außerdem wird das Bild verkehrt herum projiziert.

Die Geometrie von Szene und Bild lässt sich am einfachsten mit der Lochkamera erläutern. Wir verwenden ein dreidimensionales Koordinatensystem mit dem Ursprung am Loch und betrachten in der Szene einen Punkt P mit den Koordinaten (X, Y, Z). P wird in der Bildebene auf den Punkt P' mit den Koordinaten (x, y, z) projiziert. Wenn f der Abstand vom Loch zur Bildebene ist, können wir durch ähnliche Dreiecke die folgenden Gleichungen ableiten:

$$\frac{-x}{f} = \frac{X}{Z}, \quad \frac{-y}{f} = \frac{Y}{Z} \Rightarrow x = \frac{-fX}{Z}, \quad y = \frac{-fY}{Z}.$$

Diese Gleichungen definieren einen Bildaufbauprozess, der auch als **perspektivische Projektion** bezeichnet wird. Das Z im Nenner bedeutet, dass das Bild eines Objekts umso kleiner ist, je weiter das Objekt entfernt ist. Außerdem heißt das Minuszeichen, dass das Bild im Vergleich zur Szene *umgekehrt* ist, nämlich sowohl im Hinblick auf links und rechts als auch im Hinblick auf unten und oben.

Bei perspektivischer Projektion erscheinen weit entfernte Objekte klein. Deshalb ist es möglich, den Mond mit der Hand abzudecken (siehe Abbildung 24.1). Ein wichtiges Ergebnis dieses Effekts ist, dass parallele Linien zu einem Punkt am Horizont konvergieren (wie bei den Eisenbahnschienen in Abbildung 24.1). Eine Linie in der Szene, die den Punkt (X_0, Y_0, Z_0) in der Richtung (U, V, W) durchläuft, kann als die Menge der Punkte $(X_0 + \lambda U, Y_0 + \lambda V, Z_0 + \lambda W)$ beschrieben werden, wobei λ zwischen $-\infty$ und $+\infty$ variiert. Wird (X_0, Y_0, Z_0) anders gewählt, erhält man andere Linien, die parallel zueinander verlaufen. Die Projektion eines Punktes P_λ von dieser Linie auf die Bildebene ist gegeben durch

$$\left(f\frac{X_0 + \lambda U}{Z_0 + \lambda W}, f\frac{Y_0 + \lambda V}{Z_0 + \lambda W} \right).$$

Für $\lambda \to \infty$ oder $\lambda \to -\infty$ wird dies zu $p_\infty = (fU/W, fV/W)$, wenn $W \neq 0$. Das bedeutet, dass zwei parallele Linien, die von verschiedenen Punkten im Raum ausgehen, im Bild zusammenlaufen – für große λ sind die Bildpunkte nahezu gleich, wobei der Wert von (X_0, Y_0, Z_0) keine Rolle spielt (denken Sie auch hier wieder an die Eisenbahnschienen in Abbildung 24.1). Wir bezeichnen p_∞ als den **Fluchtpunkt**, der der Familie der geraden Linien mit der Richtung (U, V, W) zugeordnet ist. Linien mit derselben Richtung haben denselben Fluchtpunkt.

24.1.2 Linsensysteme

Der Nachteil bei der Lochkamera liegt darin, dass ein sehr kleines Loch erforderlich ist, damit das Bild scharf wird. Je kleiner aber das Loch ist, desto weniger Photonen können passieren – das Bild ist entsprechend dunkel. Mit einer größeren Belichtungszeit lassen sich zwar mehr Photonen einfangen, doch dann kommt es zu einer **Bewegungsunschärfe**: Objekte, die sich in der Szene bewegen, erscheinen verschwommen, da sie Photonen an mehrere Positionen auf der Bildebene senden. Wenn es nicht möglich ist, das Loch länger geöffnet zu halten, können wir versuchen, es größer zu machen. Es tritt zwar mehr Licht ein, doch Licht von einem kleinen Objektbereich in der Szene wird nun über einen Bereich auf der Bildebene verteilt, sodass wiederum ein verschwommenes Bild entsteht.

Wirbeltieraugen und moderne Kameras verwenden ein Linsensystem, um zum einen genügend Licht aufzunehmen und zum anderen das Bild scharf zu halten. Vor einer großen Öffnung befindet sich eine **Linse**, die das Licht von naheliegenden Objektpositionen auf naheliegende Positionen in der Bildebene fokussiert. Allerdings besitzen Linsensysteme eine begrenzte **Tiefenschärfe**: Sie können Licht nur von Punkten fokussieren, die innerhalb eines bestimmten Tiefenbereiches (um eine **Brennebene** herum) liegen. Objekte außerhalb dieses Bereiches sind im Bild unscharf. Um die Brennebene zu verschieben, lässt sich die Linse im Auge verformen (siehe ▶ Abbildung 24.3); in einer Kamera können die Linsen vor- und zurückbewegt werden.

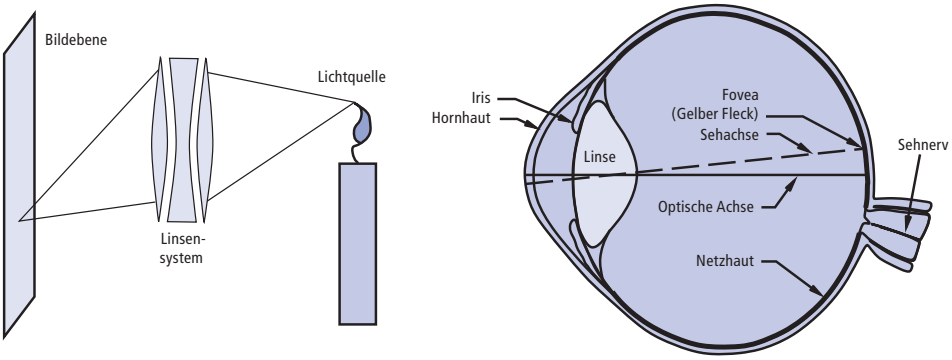


Abbildung 24.3: Linsen sammeln das Licht, das von einem Szenenpunkt in einem Richtungsbereich ausgeht, und lenken es so, dass es insgesamt auf einen einzigen Punkt der Bildebene auftrifft. Die Fokussierung funktioniert für Punkte, die im Raum nahe einer Brennebene liegen; andere Punkte werden nicht richtig scharf abgebildet. In Kameras lassen sich die Elemente des Linsensystems verschieben, um die Brennebene zu verschieben, während im Auge spezialisierte Muskeln die Form der Linse verändern.

24.1.3 Skalierte orthographische Projektion

Perspektivische Effekte sind nicht immer ausgeprägt. Zum Beispiel sehen Punkte auf einem Leoparden möglicherweise klein aus, weil der Leopard weit entfernt ist, jedoch haben zwei Punkte, die nebeneinanderliegen, ungefähr die gleiche Größe. Dies hängt damit zusammen, dass die Entfernung zwischen den Punkten gering ist verglichen mit dem Abstand zu ihnen. Somit können wir das Projektionsmodell vereinfachen. Das geeignete Modell ist die **skalierte orthographische Projektion**, der folgende Idee zugrunde liegt: Wenn die Tiefe Z der Punkte auf dem Objekt innerhalb eines Bereiches $Z_0 \pm \Delta Z$ variiert, wobei $\Delta Z << Z_0$, kann der perspektivische Skalierungsfaktor f/Z durch eine Konstante $s = f/Z_0$ angenähert werden. Die Gleichungen für die Projektion der Szenenkoordinaten (X, Y, Z) auf die Bildebene werden zu $x = sX$ und $y = sY$. Die skalierte orthographische Projektion ist eine Annäherung, die nur für die Teile der Szene gültig ist, die nicht zu viele Variationen in der internen Tiefe aufweisen. Zum Beispiel kann die orthographische Projektion ein gutes Modell für die Merkmale an der Vorderseite eines entfernten Gebäudes sein.

24.1.4 Licht und Schatten

Die Helligkeit eines Pixels im Bild ist eine Funktion der Helligkeit des Oberflächenstückes in der Szene, die auf das Pixel projiziert wird. Wir nehmen hier ein lineares Modell an (moderne Kameras arbeiten bei den Extremwerten von hell und dunkel nichtlinear, jedoch linear im mittleren Helligkeitsbereich). Die Bildhelligkeit ist ein starker, wenn auch nicht eindeutiger Hinweis auf die Gestalt eines Objektes und von da aus auf seine Identität. Menschen sind normalerweise in der Lage, die drei Hauptursachen für variierende Helligkeit zu unterscheiden und Rückschlüsse auf die Eigenschaften des Objektes zu ziehen. Die erste Ursache ist die **Gesamtintensität** des Lichtes. Selbst wenn ein weißes Objekt im Schatten weniger hell als ein schwarzes Objekt im direkten Sonnenlicht erscheinen mag, kann das Auge die relative Helligkeit gut unterscheiden und das weiße Objekt als weiß wahrnehmen. Zweitens können verschiedene Punkte in der Szene mehr oder weniger Licht **reflektieren**. Als Ergebnis nimmt man diese Punkte normalerweise als heller oder dunkler wahr und erkennt somit Texturen oder Markierungen auf dem Objekt. Drittens sind die dem Licht zugewandten Oberflächenstellen heller als die Oberflächenstellen, die gegenüber dem einfallenden Licht geneigt sind – ein Effekt, den man als **Shading** (**Schattierung**) bezeichnet. In der Regel kann der Mensch erkennen, dass diese Schattierung von der Geometrie des Objektes stammt, er bringt aber manchmal Schattierung und Markierungen durcheinander. Beispielsweise sieht ein Streifen dunklen Make-ups unter den Wangenknochen wie ein Schattierungseffekt aus, wodurch das Gesicht schmaler erscheint.

Die meisten Oberflächen reflektieren Licht durch **diffuse Reflexion**. Das Licht streut dabei gleichmäßig in alle Richtungen von der Oberfläche weg, sodass die Helligkeit einer diffusen Oberfläche von der Blickrichtung unabhängig ist. Dies trifft unter anderem auf die meisten Stoffe, Farben, unbehandelten Holzoberflächen, die Vegetation und rauen Stein zu. Spiegel sind nicht diffus, denn was man sieht, hängt von der Richtung ab, in der man in den Spiegel blickt. Das Verhalten eines perfekten Spiegels wird als **spiegelnde Reflexion** bezeichnet. Manche Oberflächen wie zum Beispiel gebürstetes Metall, Plastik oder ein feuchter Fußboden zeigen kleine Stellen, an denen spiegelnde Reflexion auftritt. Diese sogenannten **Glanzlichter** sind leicht auszumachen, da sie klein

und hell sind (siehe ▶ Abbildung 24.4). Für fast alle Zwecke genügt es, Oberflächen als diffus mit Glanzlichtern zu modellieren.

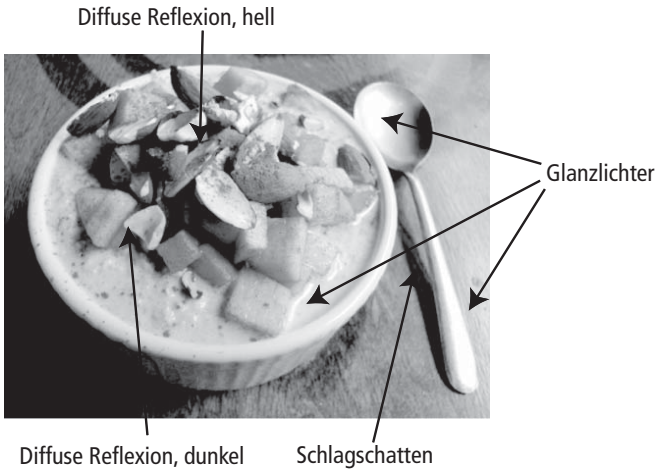


Abbildung 24.4: Verschiedene Beleuchtungseffekte. Auf dem Metalllöffel und auf der Milch sind Glanzlichter zu sehen. Die helle diffuse Oberfläche erscheint hell, weil sie dem einfallenden Licht zugewandt ist. Die dunkle diffuse Oberfläche erscheint dunkel, weil sie tangential zur Beleuchtungsrichtung liegt. Die Schatten an Punkten der Oberfläche entstehen, weil diese Stellen von der Lichtquelle nicht angestrahlt werden. Foto von Mike Linksvayer (mlinksva auf flickr).

Die Hauptquelle äußerer Beleuchtung ist die Sonne, deren Strahlen praktisch parallel zueinander einfallen. Wir modellieren dieses Verhalten als **entfernte Punktlichtquelle**. Es ist das wichtigste Modell der Beleuchtung und erweist sich sowohl für Innenraumszenen als auch für Szenen im Freien als recht effektiv. Die Lichtmenge, die in diesem Modell von einem Oberflächenabschnitt aufgenommen wird, hängt vom Winkel θ zwischen der Beleuchtungsrichtung und der Oberflächennormalen ab.

Ein diffuser Oberflächenbereich, der von einer entfernten Punktlichtquelle angestrahlt wird, reflektiert einen bestimmten Bruchteil des aufgenommenen Lichts; dieser Anteil ist die sogenannte **diffuse Albedo** (lat. Weißheit). Weißes Papier und Schnee besitzen eine hohe Albedo von etwa 0,90, während schwarzer Samt und Holzkohle eine geringe von etwa 0,05 aufweisen (d.h., dass 95% Prozent des einfallenden Lichts von den Samtfasern oder den Holzkohleporen absorbiert wird). Das **Lambertsche Kosinusgesetz** besagt, dass die Helligkeit einer diffusen Fläche durch

$$I = \rho I_0 \cos\theta$$

gegeben ist, wobei ρ die diffuse Albedo, I_0 die Lichtintensität der Lichtquelle und θ der Winkel zwischen der Lichtquelle und der Oberflächennormalen ist (siehe ▶ Abbildung 24.5). Gemäß dem Lambertschen Gesetz kommen helle Bildpixel von Oberflächenelementen, die dem Licht direkt zugewandt sind, und dunkle Pixel von Elementen, auf die das Licht nur tangential fällt, sodass die Schattierungen auf einer Oberfläche bestimmte Informationen über die Form liefern. Diesen Anhaltspunkt untersuchen wir in *Abschnitt 24.4.5*. Wenn die Oberfläche von der Lichtquelle nicht erreicht wird, befindet sie sich im **Schatten**. Da die im Schatten liegende Oberfläche etwas Licht von anderen Quellen empfängt, sind Schatten nur selten einheitlich schwarz. Die wichtigste derartige Quelle im Freien ist der Himmel, der ziemlich hell

ist. Im Innenbereich beleuchtet reflektiertes Licht von anderen Oberflächen die im Schatten befindlichen Elemente. Diese **Mehrfachreflexionen** wirken sich auch erheblich auf die Helligkeit anderer Oberflächen aus. Manchmal werden solche Effekte modelliert, indem man einen Term für zusätzliche **Umgebungsbeleuchtung** zur vorhergesagten Intensität hinzufügt.

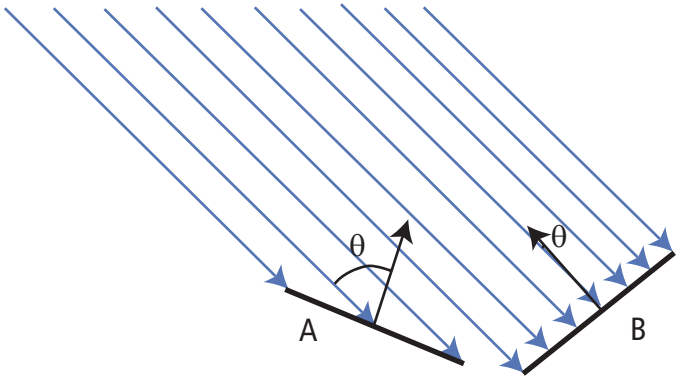


Abbildung 24.5: Zwei Oberflächenelemente, die von einer entfernten Punktquelle beleuchtet werden. Die einfallenden Strahlen sind mit grauen Pfeilspitzen dargestellt. Der Bereich *A* ist gegenüber der Quelle geneigt (θ nahe 90°) und nimmt weniger Energie auf, da er weniger Lichtstrahlen pro Oberflächeneinheit schneidet. Der Bereich B nimmt mehr Energie auf, da er der Quelle zugewandt ist (θ nahe 0°).

24.1.5 Farbe

Früchte sind das Lockmittel, mit dem ein Baum die Tiere anzieht, um seine Samen zu verbreiten. Die Entwicklung der Bäume hat Früchte hervorgebracht, die sich rot oder gelb verfärben, wenn sie reif sind, und Tiere haben sich während ihrer Entwicklung darauf eingestellt, diese Farbänderungen zu erkennen. Licht, das auf dem Auge ankommt, besteht aus Strahlen verschiedener Wellenlängen mit unterschiedlichen Energiemengen, was sich durch eine spektrale Energiedichtefunktion darstellen lässt. Das menschliche Auge reagiert auf Licht im Wellenlängenbereich von 380 bis 750 nm mit drei verschiedenen Arten von Farbrezeptorzellen, deren Empfindlichkeitsspitzen bei 420 nm (Blau), 540 nm (Grün) und 570 nm (Rot) liegen. Zwar kann das Auge nur einen kleinen Bruchteil der vollen spektralen Energiedichtefunktion aufnehmen, doch genügt dies, um die Reife einer Frucht zu erkennen.

Das **Prinzip der Dreifarbigkeit** (Trichromie) besagt, dass es für eine beliebige spektrale Energiedichte unabhängig von ihrer Komplexität möglich ist, eine andere spektrale Energiedichte zu konstruieren, die aus einer Mischung von lediglich drei Farben – normalerweise Rot, Grün und Blau – besteht, sodass ein Mensch den Unterschied zwischen den beiden nicht erkennen kann. Deshalb kommen unsere Farbfernsehgeräte und Computerdisplays mit den drei Farbelementen für Rot/Grün/Blau (kurz RGB) aus. Dadurch vereinfachen sich auch unsere Algorithmen für die Computervision. Jede Oberfläche lässt sich mit drei verschiedenen Albedo-Werten für RGB modellieren. In ähnlicher Weise kann jede Lichtquelle durch drei RGB-Intensitäten modelliert werden. Dann wenden wir jeweils das Lambertsche Kosinusgesetz an, um drei RGB-Pixelwerte zu erhalten. Dieses Modell sagt korrekt voraus, dass die gleiche Oberfläche unterschiedlich gefärbte Bildelemente unter verschiedenfarbigen Lichtern erzeugt.

Menschliche Beobachter können recht gut die Wirkungen von wechselnden Beleuchtungsverhältnissen ignorieren und sind in der Lage, die Farbe der Oberfläche unter weißem Licht zu bewerten – ein als **Farbkonstanz** bezeichneter Effekt. Heute sind recht genaue Farbkonstanzalgorithmen verfügbar; einfache Versionen sind beispielsweise in der Funktion „automatischer Weißabgleich“ einer Kamera realisiert. Wenn Sie allerdings eine Kamera für Fangschreckenkrebse bauen wollten, bräuchten Sie zwölf verschiedene Pixelfarben, die den zwölf Arten von Farbrezeptoren der Krustentiere entsprechen.

24.2 Frühe Operationen der Bildverarbeitung

Wir haben gesehen, wie Licht in der Szene von Objekten reflektiert wird und ein Bild von etwa fünf Millionen 3-Byte-Pixeln erzeugt. Wie bei allen Sensoren tritt Rauschen im Bild auf und in jedem Fall sind sehr viele Daten zu verarbeiten. Wie also analysieren wir diese Daten?

In diesem Abschnitt untersuchen wir drei nützliche Operationen für die Bildverarbeitung: Kantenerkennung, Texturanalyse und Berechnung des optischen Flusses. Man spricht hier von „frühen“ oder „Low-Level“-Operationen, weil sie die ersten in einer langen Reihe von Operationen sind. Frühe Operationen für die Computervision sind durch ihre lokale Natur charakterisiert (sie können in einem Teil des Bildes ausgeführt werden, ohne weiter entfernt liegende Pixel zu beeinträchtigen) ebenso wie durch das Fehlen von Wissen: Wir können diese Operationen ausführen, ohne die Objekte zu betrachten, die in der Szene vorhanden sind. Das macht die Low-Level-Operationen zu guten Kandidaten für die Implementierung auf paralleler Hardware – entweder in einem Grafikprozessor (Graphics Processor Unit, GPU) oder im Auge. Anschließend betrachten wir eine Operation auf mittlerer Ebene, nämlich die Segmentierung des Bildes in Bereiche.

24.2.1 Kantenerkennung

Kanten sind gerade Linien oder Kurven in der Bildebene, entlang derer sich eine „wesentliche“ Änderung der Bildhelligkeit erstreckt. Das Ziel der Kantenerkennung ist, von dem unübersichtlichen Multi-Megabyte-Bild weg und hin zu einer kompakteren, abstrakten Repräsentation zu gelangen, wie in ▶ Abbildung 24.6 gezeigt. Die Motivation dafür ist, dass Kantenkonturen im Bild wichtigen Szenenkonturen entsprechen. In der Abbildung haben wir drei Beispiele für Tiefendiskontinuität mit 1, zwei Oberflächen-Orientierungsdiskontinuitäten mit 2, eine Reflexdiskontinuität mit 3 sowie eine Beleuchtungsdiskontinuität (Schatten) mit 4 beschriftet. Eine Kantenerkennung erfolgt nur im Bild und unterscheidet deshalb nicht zwischen diesen verschiedenen Arten der Diskontinuitäten in der Szene, die jedoch in späteren Verarbeitungen berücksichtigt werden.

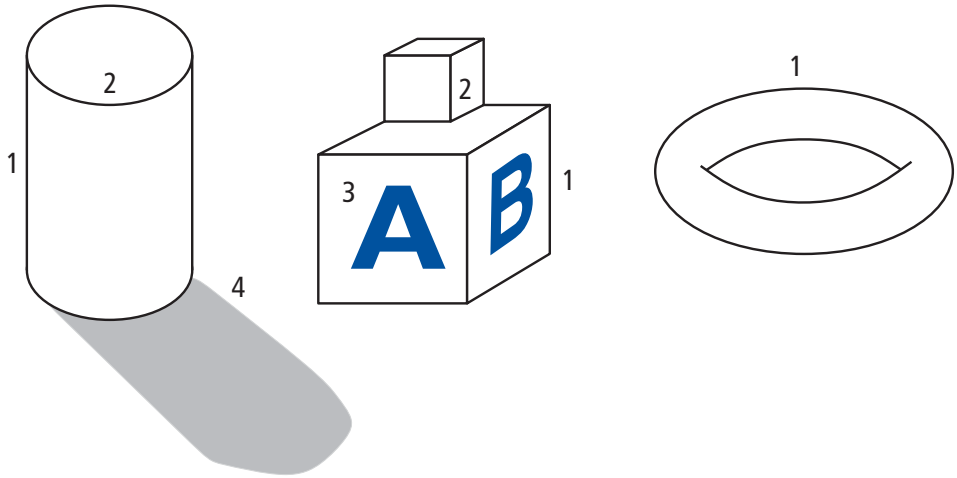


Abbildung 24.6: Unterschiedliche Arten von Kanten: (1) Tiefendiskontinuitäten; (2) Oberflächendiskontinuitäten; (3) Reflexionsdiskontinuitäten; (4) Beleuchtungsdiskontinuitäten (Schatten).

▶ Abbildung 24.7(a) zeigt eine Szene mit einem Tacker, der auf einem Schreibtisch steht und (b) die Ausgabe eines Kantenerkennungsalgorithmus für dieses Bild. Wie Sie sehen, gibt es eine Differenz zwischen der Ausgabe und einer idealen Strichzeichnung. Es gibt Lücken, wo keine Kante erscheint, und es gibt „verrauschte" Kanten, die nichts Wesentlichem in der Szene entsprechen. Spätere Verarbeitungsphasen müssen diese Fehler korrigieren.

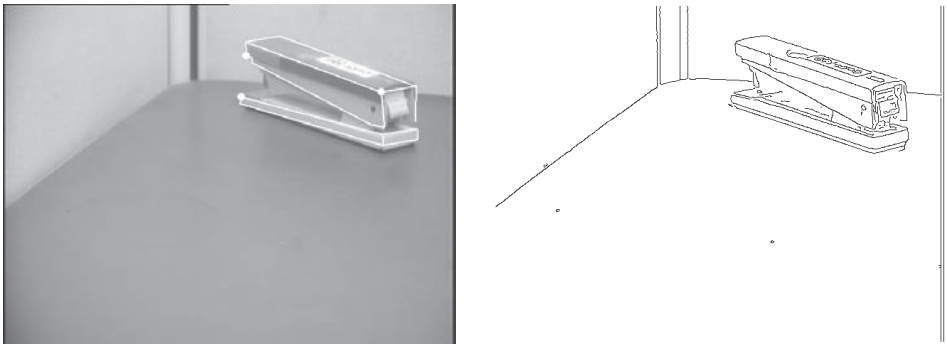


Abbildung 24.7: (a) Foto eines Tackers. (b) Aus (a) berechnete Kanten.

Wie erkennen wir Kanten in einem Bild? Betrachten Sie das Profil der Bildhelligkeit entlang eines eindimensionalen Querschnittes senkrecht zu einer Kante – beispielsweise zwischen der linken Kante des Schreibtisches und der Wand. Es sieht etwa aus wie in ▶ Abbildung 24.8 (oben) gezeigt.

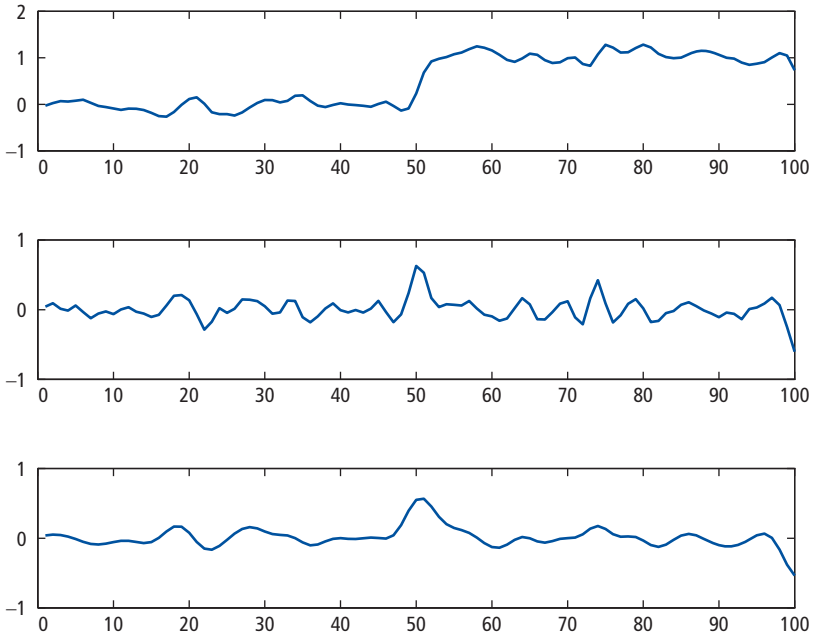


Abbildung 24.8: Oben: Intensitätsprofil $I(x)$ entlang eines eindimensionalen Querschnittes über eine Schnittkante bei $x = 50$. Mitte: Die Ableitung der Intensität, $I'(x)$. Große Werte dieser Funktion entsprechen Kanten, aber die Funktion ist verrauscht. Unten: Die Ableitung einer geglätteten Version der Intensität, $(I * G_\sigma)'$, die in einem Schritt als Konvolution $I * G'_\sigma$ berechnet werden kann. Die durch das Rauschen entstandene Kante bei $x = 75$ ist verschwunden.

Weil Kanten den Positionen im Bild entsprechen, wo die Helligkeit eine scharfe Änderung erfährt, wäre es eine naive Idee, das Bild zu differenzieren und dort nach Orten zu suchen, wo die Größe der Ableitung $I'(x)$ groß ist. Genau das funktioniert beinahe. In ▶ Abbildung 24.8 (Mitte) sehen wir, dass es eine Spitze bei $x = 50$ gibt, dass es aber auch Nebenspitzen an anderen Positionen gibt (z.B. $x = 75$). Sie treten auf, weil es im Bild ein Rauschen gibt. Wenn wir das Bild zuerst glätten, werden die überflüssigen Spitzen vermindert, wie in ▶ Abbildung 24.8 (unten) gezeigt.

Die Messung der Pixel-Helligkeit in einer CCD-Kamera beruht auf einem physikalischen Vorgang, der mit der Absorption von Photonen und der Freisetzung von Elektronen zusammenhängt; unvermeidbar sind statistische Schwankungen der Messung – Rauschen. Das Rauschen lässt sich mit einer Gaußschen Wahrscheinlichkeitsverteilung für jedes Pixel unabhängig von den anderen modellieren. Um ein Bild zu glätten, kann man jedem Pixel den Durchschnittswert seiner Nachbarn zuordnen. Damit eliminieren wir Extremwerte. Aber wie viele Nachbarn sollen wir berücksichtigen – die ein Pixel entfernten oder zwei oder mehr? Hier bietet sich ein gewichteter Mittelwert an, der die nächst gelegenen Pixel am stärksten gewichtet und dann die Gewichte für weiter entfernte Pixel allmählich verringert. Genau dies bewerkstelligt der **Gaußsche Filter**. (Benutzer von Photoshop werden darin die Funktion *Gaußscher Weichzeichner* erkennen.) Wie Sie wissen, sieht die Gaußsche Funktion mit Standardabweichung σ und Mittelwert 0 wie folgt aus:

$$N_\sigma(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-x^2/2\sigma^2} \quad \text{in einer Dimension bzw.}$$

$$N_\sigma(x,y) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2} \quad \text{in zwei Dimensionen.}$$

Die Anwendung des Gaußschen Filters ersetzt die Intensität $I(x_0, y_0)$ durch die Summe über alle (x, y)-Pixel von $I(x, y)N_\sigma(d)$, wobei d die Distanz von (x_0, y_0) bis (x, y) ist. Diese Art gewichteter Summe ist so gebräuchlich, dass es einen speziellen Namen sowie eine Notation dafür gibt. Wir sagen, die Funktion h ist die **Konvolution** von zwei Funktionen f und g (geschrieben als $f * g$), wenn wir Folgendes haben:

$$h(x) = (f * g)(x) = \sum_{u=-\infty}^{+\infty} f(u)g(x-u) \quad \text{in einer Dimension bzw.}$$

$$h(x,y) = (f * g)(x,y) = \sum_{u=-\infty}^{+\infty}\sum_{v=-\infty}^{+\infty} f(u,v)g(x-u,y-v) \quad \text{in zwei Dimensionen.}$$

Die Glättungsfunktion wird also durch Konvolution des Bildes mit der Gaußschen Funktion $I * N_\sigma$ erreicht. Ein σ von 1 Pixel ist ausreichend, um ein geringes Rauschen zu glätten, während zwei Pixel ein stärkeres Rauschen kompensieren können, allerdings auf Kosten eines gewissen Detailverlustes. Weil der Einfluss der Gaußschen Funktionen mit zunehmendem Abstand schnell schwindet, können wir in der Praxis das $\pm\infty$ in den Summen durch $\pm 3\sigma$ ersetzen.

Die Berechnung können wir optimieren, indem wir die Glättung und die Kantenerkennung zu einer einzigen Operation zusammenfassen. Gemäß einem Theorem ist für beliebige Funktionen f und g die Ableitung der Konvolution $(f * g)'$ gleich der Konvolution mit der Ableitung $f * (g)'$. Anstatt also das Bild zu glätten und dann zu differenzieren, können wir einfach eine Konvolution des Bildes mit der Ableitung der Gaußschen Glättungsfunktion N'_σ vornehmen. Dann markieren wir diejenigen Spitzen in der Antwort als Kanten, die über einem Schwellenwert liegen.

Dieser Algorithmus lässt sich problemlos von eindimensionalen Querschnitten für allgemeine zweidimensionale Bilder erweitern. In zwei Dimensionen können die Kanten in jedem Winkel θ zueinander verlaufen. Betrachtet man die Bildhelligkeit als Skalarfunktion der Variablen x, y, ist ihr Gradient ein Vektor

$$\nabla I = \begin{pmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{pmatrix} = \begin{pmatrix} I_x \\ I_y \end{pmatrix}.$$

Kanten entsprechen den Stellen in einem Bild, wo die Helligkeit eine abrupte Änderung erfährt. Somit sollte die Größe des Gradienten $\|\nabla I\|$ an einem Kantenpunkt einen hohen Wert aufweisen. Unabhängig davon ist die Richtung des Gradienten

$$\frac{\nabla I}{\|\nabla I\|} = \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix}$$

interessant. Dies gibt uns $\theta = \theta(x, y)$ an jedem Pixel, was die **Kantenorientierung** an diesem Pixel definiert.

Wie im eindimensionalen Fall berechnen wir nicht ∇I, um den Gradienten zu bilden, sondern stattdessen $\nabla(I * N_\sigma)$, den Gradienten nach der Glättung des Bildes, durch Konvolution mit einer Gaußfunktion. Und auch hier lässt sich dies durch Konvolution des Bildes mit den partiellen Ableitungen einer Gaußschen Funktion abkürzen. Sind die Gradienten berechnet, erhalten wir die Kanten, indem wir die Kantenpunkte suchen und sie miteinander verbinden. Um einen Punkt als Kantenpunkt zu identifizieren, müssen wir vorherige und nachfolgende Punkte mit geringem Abstand in Richtung des Gradienten betrachten. Ist der Betrag des Gradienten an einem dieser Punkte größer, könnten wir zu einem besseren Kantenpunkt gelangen, wenn die Kantenkurve geringfügig verschoben wird. Falls der Betrag des Gradienten zu klein ist, kann es sich bei diesem Punkt nicht um einen Kantenpunkt handeln. An einem Kantenpunkt ist also der Gradientenbetrag ein lokales Maximum in Richtung des Gradienten und der Gradientenbetrag liegt über einem angemessenen Schwellenwert.

Nachdem wir Kantenpixel mit diesem Algorithmus markiert haben, sind in der nächsten Phase die Pixel zu verbinden, die zu denselben Kantenkurven gehören. Dazu können wir davon ausgehen, dass zwei benachbarte Pixel, die beide Kantenpixel mit konsistenter Ausrichtung sind, zu derselben Kantenkurve gehören müssen.

24.2.2 Textur

Unter **Textur** versteht man das visuelle Empfinden einer Oberfläche – was man sieht, führt zu der Vorstellung, wie sich die Oberfläche anfühlen könnte, wenn man sie berührt („Textur“ hat dieselbe Wurzel wie „Textil“). In der Computervision bezieht sich Textur auf ein räumlich wiederholtes Muster auf einer Oberfläche, das optisch wahrnehmbar ist. Hierzu gehören beispielsweise die Muster von Fenstern an einem Gebäude, Maschen eines Pullovers, Flecken auf einem Leoparden, Grashalme auf einem Rasen, Kieselsteine am Strand und Menschen in einem Stadion. Manchmal sind die Muster ziemlich regelmäßig, wie etwa bei den Maschen eines Pullovers, in anderen Fällen sind die Elemente statistisch verteilt, wie etwa bei Kieselsteinen am Strand.

Während die Helligkeit zu den Eigenschaften individueller Pixel gehört, ist das Konzept der Textur nur für eine Oberflächenstelle mit mehreren Pixeln sinnvoll. Für eine derartige Stelle berechnen wir die Orientierung an jedem Pixel und charakterisieren dann die Stelle durch ein Histogramm von Orientierungen. So hat die Textur von Ziegelsteinen in einer Mauer zwei Spitzen im Histogramm (eine vertikal und eine horizontal), während die Textur von Flecken auf der Leopardenhaut eine eher gleichförmige Verteilung von Orientierungen aufweist.

▶ Abbildung 24.9 zeigt, dass Orientierungen im Wesentlichen gegenüber Beleuchtungsänderungen invariant sind. Dadurch liefert eine Textur wichtige Anhaltspunkte für die Objekterkennung, da andere Anhaltspunkte wie zum Beispiel Kanten bei unterschiedlichen Beleuchtungsbedingungen zu abweichenden Ergebnissen führen können.

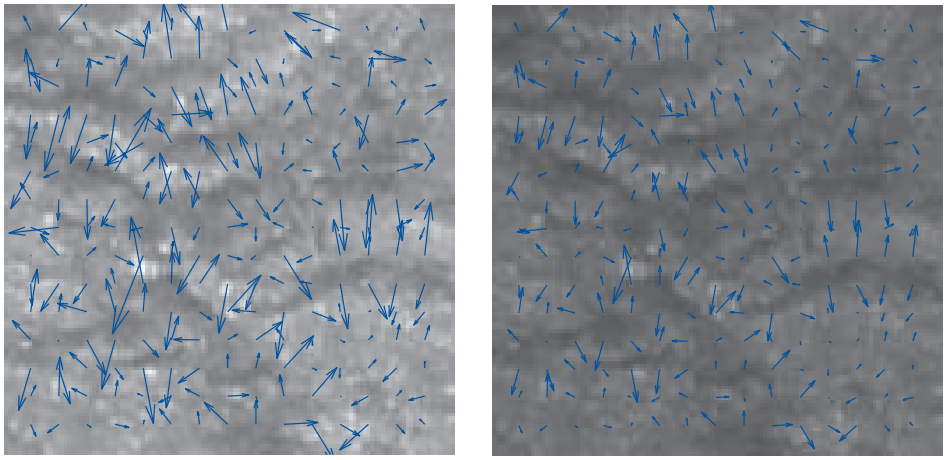


Abbildung 24.9: Zwei Bilder derselben Textur von zerknittertem Reispapier bei verschiedenen Beleuchtungsniveaus. Das Gradientenvektorfeld (für jedes achte Pixel) ist über die beiden Bilder gelegt worden. Bei geringerer Beleuchtung werden alle Gradientenvektoren kürzer. Da sich die Vektoren aber nicht drehen, ändern sich auch die Gradientenorientierungen nicht.

In Bildern texturierter Objekte funktioniert die Kantenerkennung nicht so gut wie bei glatten Objekten. Das hängt damit zusammen, dass die wichtigsten Kanten unter den Texturelementen verlorengehen können. Buchstäblich können wir den Tiger vor lauter Streifen nicht sehen. Als Lösung bietet es sich an, nach Unterschieden in Textureigenschaften zu suchen, genau wie wir nach Helligkeitsunterschieden suchen. Eine Stelle auf einem Tiger und eine Stelle auf dem grasbedeckten Untergrund werden sehr unterschiedliche Orientierungshistogramme haben, sodass wir die Grenzkurve zwischen ihnen finden können.

24.2.3 Optischer Fluss

Als Nächstes betrachten wir, was passiert, wenn statt eines einzigen statischen Bildes eine Videosequenz vorliegt. Wenn sich ein Objekt im Video bewegt oder wenn sich die Kamera relativ zu einem Objekt bewegt, wird die resultierende scheinbare Bewegung im Bild als **optischer Fluss** bezeichnet. Der optische Fluss beschreibt die Richtung und die Geschwindigkeit der Bewegung von Merkmalen *im Bild* – der optische Fluss des Videos von einem Rennwagen wird dann in Pixel pro Sekunde und nicht in Kilometern pro Stunde gemessen. Der optische Fluss kodiert nützliche Informationen über die Szenenstruktur. Zum Beispiel haben in einem Landschaftsvideo, das aus einem fahrenden Zug aufgenommen wird, entfernte Objekte eine geringere scheinbare Bewegung als näher liegende Objekte; somit lässt sich aus dem Verhältnis der scheinbaren Bewegungen auf die Entfernung schließen. Außerdem ermöglicht es uns der optische Fluss, Aktionen zu erkennen. In ▶ Abbildung 24.10 sind links zwei Frames aus dem Video eines Tennisspielers zu sehen. Die aus diesen Bildern berechneten Vektoren des optischen Flusses sind im rechten Teil der Abbildung dargestellt. Es ist zu erkennen, dass sich der Schläger und das vordere Bein am schnellsten bewegt haben.

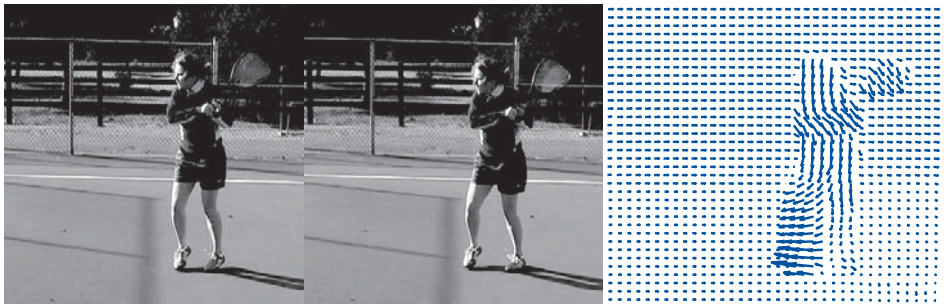


Abbildung 24.10: Zwei Frames einer Videosequenz. Auf der rechten Seite ist das Vektorfeld des optischen Flusses dargestellt, das der Verschiebung von einem Frame zum anderen entspricht. Die Bewegungen des Tennisschlägers und des vorderen Beines werden durch die Richtungen der Pfeile erfasst. (Mit freundlicher Genehmigung von Thomas Brox.)

Das Vektorfeld des optischen Flusses kann an jedem Punkt (x, y) durch seine Komponenten $v_x(x, y)$ in der x-Richtung und $v_y(x, y)$ in der y-Richtung dargestellt werden. Um den optischen Fluss zu messen, müssen wir entsprechende Punkte zwischen zwei aufeinanderfolgenden Einzelbildern finden. Wir nutzen dabei die Tatsache, dass Bildbereiche um die entsprechenden Punkte herum ähnliche Intensitätsmuster aufweisen. Betrachten Sie einen Block von Pixeln, der sich um das Pixel p, (x_0, y_0) zum Zeitpunkt t_0 zentriert. Dieser Pixelblock muss mit den Pixelblöcken verglichen werden, die sich um verschiedene Kandidaten-Pixel q_i an der Stelle $(x_0 + D_x, y_0 + D_y)$ zum Zeitpunkt $t_0 + D_t$ zentrieren. Ein mögliches Ähnlichkeitsmaß ist die **Summe der quadrierten Differenzen** (SQD, Englisch SSD, Sum of Squared Differences):

$$SSD(D_x, D_y) = \sum_{(x,y)} (I(x,y,t) - I(x + D_x, y + D_y, y + D_y, t + D_t))^2.$$

Hier deckt (x, y) die Pixel im Block ab, dessen Zentrum in (x_0, y_0) liegt. Wir suchen das (D_x, D_y), das die SQD minimiert. Der optische Fluss an der Stelle (x_0, y_0) ist dann $(v_x, v_y) = (D_x/D_t, D_y/D_t)$. Damit dies auch funktioniert, ist eine gewisse Textur oder Variation in der Szene erforderlich. Betrachtet man eine einheitlich weiße Wand, ist die SQD für die verschiedenen Vergleiche annähernd gleich und der Algorithmus muss blind raten. Die leistungsfähigsten Algorithmen für die Messung des optischen Flusses stützen sich auf verschiedene zusätzliche Einschränkungen, wenn die Szene nur teilweise texturiert ist.

24.2.4 Segmentierung von Bildern

Die **Segmentierung** ist der Prozess, ein Bild in **Bereiche** ähnlicher Pixel zu zerlegen. Jedem Bildpixel können bestimmte visuelle Eigenschaften zugeordnet werden, wie beispielsweise Helligkeit, Farbe und Textur. Innerhalb eines Objektes oder eines einzelnen Teiles in einem Objekt variieren diese Attribute relativ wenig, während es über die Objektgrenzen hinaus normalerweise große Änderungen bei einem oder mehreren dieser Attribute gibt. Prinzipiell gibt es zwei Verfahren für die Segmentierung, das eine konzentriert sich auf die Erkennung der Grenzen dieser Bereiche und das andere auf die Erkennung der Bereiche an sich (siehe ▶ Abbildung 24.11).

Abbildung 24.11: (a) Originalbild. (b) Grenzkonturen, wobei die Kontur umso dunkler ist, je höher der P_b-Wert liegt. (c) Segmentierung in Bereiche, die einem Detailabschnitt des Bildes entsprechen. Bereiche werden in ihren durchschnittlichen Farben wiedergegeben. (d) Segmentierung in Bereiche, die einem gröberen Abschnitt des Bildes entsprechen, wodurch sich weniger Bereiche ergeben. (Mit freundlicher Genehmigung von Pablo Arbelaez, Michael Maire, Charles Fowlkes und Jitendra Malik).

Da eine Grenzkurve, die durch ein Pixel (x, y) geht, eine Orientierung von θ aufweist, lässt sich das Problem der Entdeckung von Grenzkurven als Klassifizierungsproblem des maschinellen Lernens formalisieren. Basierend auf Merkmalen aus einer lokalen Nachbarschaft berechnen wir die Wahrscheinlichkeit $P_b(x, y, 0)$, dass es sich tatsächlich um eine Grenzkurve an diesem Pixel entlang dieser Orientierung handelt. Betrachten Sie eine kreisförmige Scheibe, deren Mittelpunkt bei (x, y) liegt und die entlang eines Durchmessers in Richtung θ in zwei Halbscheiben geteilt wird. Falls bei (x, y, θ) eine Grenze liegt, sollten sich die beiden Halbscheiben deutlich in Helligkeit, Farbe und Textur unterscheiden. Martin, Fowlkes und Malik (2004) haben mit Merkmalen basierend auf Unterschieden in Histogrammen von Helligkeits-, Farb- und Texturwerten, die in diesen beiden Scheibenhälften gemessen wurden, einen Klassifizierer trainiert. Hierfür haben sie eine Datenmenge von natürlichen Bildern verwendet, auf denen Experten die „Ground Truth“ (wörtl. Bodenwahrheit) markiert hatten. Ziel des Klassifizierers war es, genau diese vom Menschen markierten Grenzen und keine anderen zu markieren.

Die mit dieser Technik erkannten Grenzen erweisen sich als deutlich besser als diejenigen, die mit der einfachen Kantenerkennungstechnik (wie oben beschrieben) gefunden wurden. Dennoch gibt es zwei Einschränkungen: (1) Die durch Schwellenwertbildung $P_b(x, y, \theta)$ entstehenden Grenzpixel liefern nicht immer geschlossene Kurven, sodass bei diesem Ansatz keine Bereiche entstehen, und (2) nutzt die Entscheidungsfindung nur einen lokalen Kontext und berücksichtigt keine globalen Konsistenzbedingungen.

Der alternative Ansatz beruht darauf, die Pixel nach Helligkeit, Farbe und Text in Bereiche zu „clustern“. Shi und Malik (2000) formulieren dies als Graphpartitionierungsproblem. Die Knoten des Graphen korrespondieren mit Pixeln und Kanten mit Verbindungen zwischen Pixeln. Das Gewicht W_{ij} an der Kante, die ein Pixelpaar i und j verbindet, beruht darauf, wie ähnlich sich die beiden Pixel in Bezug auf Helligkeit, Farbe, Textur usw. sind. Dann werden Partitionen gefunden, die das Kriterium für einen *normalisierten Schnitt* minimieren. Grob gesagt lautet das Kriterium zur Partitionierung des Graphen: Minimiere die Gewichtssumme der Verbindungen über die Gruppen von Pixeln und maximiere die Gewichtssumme der Verbindungen innerhalb der Gruppen.

Von einer Segmentierung, die nur auf lokalen Low-Level-Attributen wie beispielsweise Helligkeit und Farbe basiert, kann man nicht erwarten, dass sie die endgültigen korrekten Grenzen aller Objekte in der Szene liefert. Um zuverlässig die Grenzen der Objekte zu finden, benötigen wir High-Level-Wissen über die wahrscheinlichsten Arten von Objekten in der Szene. Die Darstellung dieses Wissens ist ein Thema der aktiven Forschung. Eine bekannte Strategie erzeugt eine Übersegmentierung eines Bildes mit Hunderten homogenen Bereichen, die als **Superpixel** bezeichnet werden. Von da an können wissensbasierte Algorithmen eingesetzt werden. Diese kommen mit Hunderten von Superpixeln besser zurecht als mit Millionen von Rohpixeln. Wie man dieses High-Level-Wissen über Objekte nutzt, ist Thema des nächsten Abschnittes.

24.3 Objekterkennung nach Erscheinung

Erscheinung ist ein Kurzwort dafür, wie ein Objekt letztlich aussieht. Manche Objektkategorien – zum Beispiel Tennisbälle – variieren kaum in ihrer Erscheinung; alle Objekte in der Kategorie sehen unter den meisten Umständen nahezu gleich aus. In diesem Fall können wir eine Merkmalsmenge berechnen, die jede Klasse von Bildern beschreibt, die wahrscheinlich das Objekt enthalten, und sie dann mit einem Klassifizierer testen.

Andere Objektkategorien – beispielsweise Häuser oder Balletttänzerinnen – variieren stärker. So unterscheiden sich Häuser hinsichtlich Größe, Farbe und Form und können zudem aus verschiedenen Betrachtungswinkeln ein anderes Aussehen haben. Eine Tänzerin sieht in jeder Pose wie auch bei wechselnden Farben der Bühnenbeleuchtung anders aus. Als sinnvolle Abstraktion kann man sagen, dass manche Objekte aus lokalen Mustern bestehen, die sich in Bezug aufeinander verschieben. Dann lässt sich das Objekt anhand lokaler Histogramme von Detektorantworten ermitteln, aus denen hervorgeht, ob ein bestimmter Teil vorhanden ist, die aber die Einzelheiten unterdrücken, wo sich der Teil befindet.

Das Testen jeder Klasse von Bildern mit einem gelernten Klassifizierer ist ein wichtiges und bewährtes Konzept. Es arbeitet äußerst gut bei Gesichtern, die direkt der Kamera zugewandt sind, da bei geringer Auflösung unter vernünftiger Beleuchtung alle derartigen Gesichter recht ähnlich aussehen. Das Gesicht ist rund und ziemlich hell verglichen mit den Augenhöhlen; diese sind dunkel, da sie vertieft sitzen. Der Mund erscheint genau wie die Augenbrauen als dunkler Strich. Starke Beleuchtungsänderungen können zwar einige Variationen in diesem Muster hervorrufen, doch ist die Variationsbreite durchaus überschaubar. Dadurch ist es möglich, Gesichtspositionen in einem Bild zu entdecken, das Gesichter enthält. Eine derartige Funktion war einst eine rechentechnische Herausforderung, gehört heute aber selbst in preiswerten Digitalkameras zur Standardausstattung.

Fürs Erste betrachten wir nur Gesichter, bei denen die Nase senkrecht orientiert ist; später befassen wir uns auch mit gedrehten Gesichtern. Über das Bild schwenken wir ein rundes Fenster fester Größe, berechnen dafür Merkmale und übergeben die Merkmale an einen Klassifizierer. Dieses Vorgehen wird auch als **Schiebefenstermethode** bezeichnet. Die Merkmale müssen gegenüber Schatten und Helligkeitsänderungen aufgrund von Beleuchtungsänderungen robust sein. Eine brauchbare Strategie ist es, Merkmale aus Gradientenorientierungen zu erstellen. Alternativ kann man die Beleuchtung in jedem Fenster bewerten und korrigieren. Um Gesichter verschiedener Größen zu finden, wird

das Schwenken über größeren oder kleineren Versionen des Bildes wiederholt. Schließlich bearbeiten wir die Antworten hinsichtlich Skalierung und Lage nach, um den endgültigen Satz von Erfassungen zu erzeugen.

Die Nachbearbeitung ist wichtig, weil das gewählte Fenster selten die genaue Größe für ein Gesicht haben dürfte (selbst wenn wir mehrere Größen nehmen). Somit haben wir höchstwahrscheinlich mehrere sich überlappende Fenster, die jeweils einen Treffer für ein Gesicht melden. Wenn wir jedoch einen Klassifizierer verwenden, der die Stärke der Antwort melden kann (zum Beispiel logistische Regression oder eine Support-Vector-Maschine), können wir diese partiell sich überlappenden Treffer bei nahe gelegenen Stellen zu einer einzelnen hochqualitativen Übereinstimmung kombinieren. Damit erhalten wir einen Gesichtsdetektor, der über Positionen und Skalierungen suchen kann. Um auch Drehungen in die Suche einzubeziehen, verwenden wir zwei Schritte. Wir trainieren eine Regressionsprozedur, um die beste Orientierung eines in einem Fenster vorhandenen Gesichtes zu bewerten. Nun schätzen wir für jedes Fenster die Orientierung, orientieren das Fenster neu und testen dann mit unserem Klassifizierer, ob ein senkrechtes Gesicht vorhanden ist. Insgesamt ergibt sich damit ein System, dessen Architektur in ▶ Abbildung 24.12 skizziert ist.

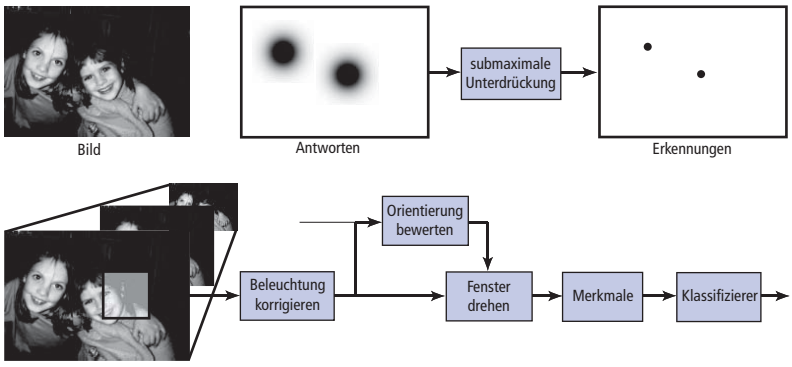


Abbildung 24.12: Gesichtserkennungssysteme weisen zwar Unterschiede auf, sind aber größtenteils nach der hier angegebenen zweiteiligen Architektur aufgebaut. Im oberen Teil gehen wir von Bildern zu Antworten und wenden dann eine submaximale Unterdrückung an, um die stärkste lokale Antwort zu finden. Die Antworten werden durch den im unteren Teil dargestellten Vorgang erhalten. Wir schwenken ein Fenster fester Größe über größeren und kleineren Versionen des Bildes, um kleinere bzw. größere Gesichter zu finden. Die Beleuchtung im Fenster wird korrigiert und anschließend sagt ein Regressionsmodul (oftmals ein neuronales Netz) die Orientierung des Gesichtes voraus. Das Fenster wird auf diese Orientierung korrigiert und dann an einen Klassifizierer übergeben. Die Ausgaben des Klassifizierers werden dann nachbearbeitet, um sicherzustellen, dass nur genau ein Gesicht an jeder Stelle im Bild platziert ist.

Trainingsdaten lassen sich recht einfach beschaffen. Es gibt mehrere Datenmengen von markierten Gesichtsbildern und gedrehte Gesichtsfenster sind einfach zu erstellen (indem man einfach ein Fenster aus einer Trainingsdatenmenge dreht). Eine vielfach verwendete Methode verwendet jedes Beispielfenster und produziert dann neue Beispiele, indem die Orientierung des Fensters, die Fenstermitte oder die Skalierung sehr leicht geändert wird. Auf diese Weise lässt sich eine größere Datenmenge erhalten, die wahre Bilder recht gut widerspiegelt. Normalerweise wird die Leistung dadurch beträchtlich verbessert. Die nach den beschriebenen Verfahren konstruierten Gesichtserkennungssysteme funktionieren heute gut für Frontalansichten (Seitenansichten sind schwieriger).

24.3.1 Komplexe Erscheinungs- und Musterelemente

Viele Objekte erzeugen komplexere Muster als es bei Gesichtern der Fall ist. Das hängt damit zusammen, dass mehrere Effekte die Merkmale in einem Bild des Objektes verschieben können. ▶ Abbildung 24.13 zeigt einige dieser Effekte.

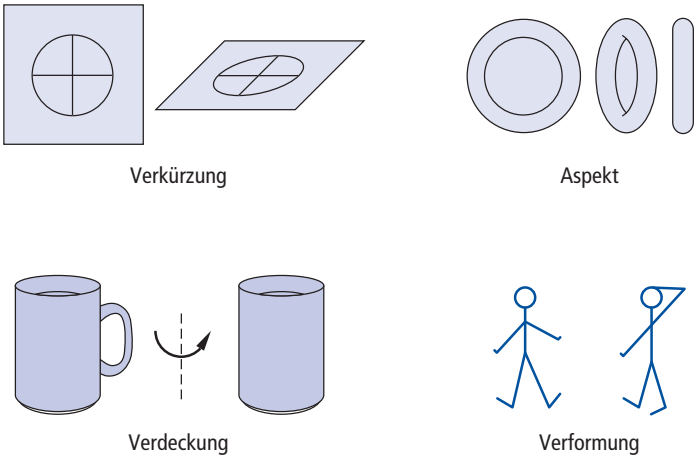


Abbildung 24.13: Quellen für Variationen der Erscheinung. Erstens können Elemente perspektivisch verkürzt werden, wie der kreisförmige Bereich oben links. Dieser Bereich wird schräg betrachtet, sodass sich ein elliptisches Bild ergibt. Zweitens können Objekte, die man aus unterschiedlichen Richtungen betrachtet, ihre Form drastisch ändern, ein als Aspekt bekanntes Phänomen. Oben rechts sind drei verschiedene Aspekte eines Donuts dargestellt. Durch Verdeckung kommt es dazu, dass der Henkel der Tasse unten links verschwindet, wenn die Tasse gedreht wird. Da in diesem Fall der Rumpf und der Henkel zur selben Tasse gehören, liegt Selbstverdeckung vor. Schließlich zeigt das Beispiel unten rechts, wie sich manche Objekte erheblich verformen können.

- **Verkürzung:** Bewirkt, dass ein schräg betrachtetes Muster erheblich verzerrt erscheint.
- **Aspekt:** Objekte sehen unterschiedlich aus, wenn sie aus verschiedenen Richtungen betrachtet werden. Selbst ein einfaches Objekt wie ein Donut hat mehrere Aspekte; von der Seite gesehen erscheint er wie ein abgeflachtes Oval, doch von oben gesehen wie ein Kranz.
- **Verdeckung (Okklusion):** Aus manchen Blickrichtungen sind bestimmte Teile nicht zu sehen. Objekte können sich gegenseitig verdecken oder Teile eines Objektes können andere Teile verdecken (ein als Selbstverdeckung bezeichneter Effekt).
- **Verformung:** Interne Freiheitsgrade des Objektes ändern seine Erscheinung. Zum Beispiel können Menschen ihre Arme und Beine bewegen, wodurch ein sehr großer Bereich von unterschiedlichen Körperkonstellationen entsteht.

Dennoch kann unser Rezept für die Suche über Positionen und Skalierungen funktionieren. Und zwar deshalb, weil eine gewisse Struktur in den vom Objekt erzeugten Bildern vorhanden ist. Zum Beispiel zeigt das Bild eines Autos höchstwahrscheinlich einige Scheinwerfer, Türen, Räder, Fenster und Radkappen, selbst wenn sie in verschiedenen Bildern in etwas unterschiedlichen Anordnungen erscheinen. Es liegt demzufolge nahe, Objekte mit Musterelementen zu modellieren – mit Sammlungen von Teilen. Diese Musterelemente können sich gegeneinander bewegen, doch wenn die meisten der Musterelemente etwa an der richtigen Stelle liegen, ist das Objekt vorhanden. Ein Objekterkenner ist dann eine Sammlung von Merkmalen, aus denen hervorgeht, ob die Musterelemente vorhanden sind und ob sie sich etwa an der richtigen Stelle befinden.

Es liegt nahe, das Bildfenster mit einem Histogramm der Musterelemente darzustellen, die dort erscheinen. Dieses Konzept funktioniert allerdings nicht besonders gut, da es bei zu vielen Mustern leicht zu Verwechslungen kommen kann. Handelt es sich beispielsweise bei den Musterelementen um Farbpixel, werden die Flaggen von Frankreich, Großbritannien und den Niederlanden verwechselt, da sie ungefähr die gleichen Farbhistogramme liefern, auch wenn die Farben unterschiedlich angeordnet sind. Allerdings gelangt man mit ziemlich einfachen Modifikationen von Histogrammen zu sehr nützlichen Merkmalen. Der Trick besteht dabei darin, bestimmte räumliche Details in der Darstellung zu bewahren; beispielsweise befinden sich Scheinwerfer meistens an der Vorderseite eines Fahrzeuges und Räder an der Unterseite. Histogrammbasierte Merkmale sind für verschiedenste Erkennungsanwendungen bereits erfolgreich eingesetzt worden – exemplarisch dafür gibt der nächste Abschnitt einen Überblick über Fußgängererkennung.

24.3.2 Fußgängererkennung mit HOG-Merkmalen

Die Weltbank schätzt, dass jedes Jahr rund 1,2 Millionen Menschen durch Autounfälle ums Leben kommen, wobei ein Drittel davon Fußgänger sind. Die Erkennung von Fußgängern ist demnach ein wichtiges Anwendungsproblem, da Autos, die Fußgänger automatisch erkennen und ihnen ausweichen können, viele Leben retten. Fußgänger tragen viele verschiedene Arten von Kleidung und erscheinen in unterschiedlichen Konstellationen, doch haben sie – bei relativ geringer Auflösung – ein ziemlich charakteristisches Aussehen. Am häufigsten kommen seitliche oder frontale Ansichten eines Ganges vor. Dabei sehen wir entweder eine „Lollipop"-Figur – der Torso ist breiter als die Beine, die sich zusammen in der Standphase des Ganges befinden – oder eine „Scherenfigur" – wobei die Beine beim Gang schwingen. Wir erwarten, einen Hinweis auf Arme und Beine zu sehen. Meistens ist auch die Kontur um Schultern und Kopf sichtbar und recht markant. Mit einer sorgfältigen Merkmalskonstruktion können wir folglich einen brauchbaren Fußgängererkenner mit gleitendem Fenster aufbauen.

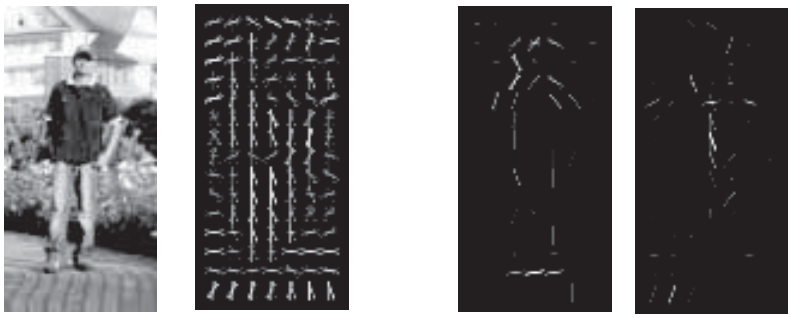


Abbildung 24.14: Lokale Orientierungshistogramme sind ein leistungsfähiges Instrument, um sogar einigermaßen komplexe Objekte zu erkennen. Links ist ein Bild eines Fußgängers zu sehen. Das zweite Bild von links zeigt lokale Orientierungshistogramme für Bildbereiche. Dann wenden wir einen Klassifizierer an (beispielsweise eine Support-Vector-Maschine), um die Gewichte für jedes Histogramm zu finden, das am besten die positiven Beispiele für Fußgänger von Nicht-Fußgängern trennt. Die positiv gewichteten Komponenten sehen wie der Umriss einer Person aus. Die negativen Komponenten sind nicht so klar; sie verkörpern alle Muster, die keine Fußgänger sind. Abbildung von Dalal and Triggs (2005) © IEEE.

Da der Kontrast zwischen Fußgänger und Untergrund nicht immer groß ist, stellt man das Bildfenster besser mit Orientierungen als mit Kanten dar. Fußgänger können Arme und Beine bewegen, sodass wir bestimmte räumliche Details im Merkmal mithilfe eines Histogramms unterdrücken sollten (▶ Abbildung 24.14). Das Fenster teilen wir in Zellen auf, die sich überlappen können, und wir erstellen in jeder Zelle ein Orientierungshistogramm. Auf diese Weise entsteht ein Merkmal, das angibt, ob die Kopf-Schultern-Kurve oben oder unten im Fenster liegt, sich aber nicht ändert, wenn der Kopf leicht bewegt wird.

Um ein gutes Merkmal zu erzeugen, ist ein weiterer Trick erforderlich. Da Orientierungsmerkmale durch die Beleuchtungshelligkeit nicht beeinflusst werden, lassen sich kontrastreiche Kanten nicht speziell behandeln. Die charakteristischen Kurven an der Grenze eines Fußgängers werden also in der gleichen Weise wie feine Texturdetails in der Kleidung oder im Hintergrund behandelt und möglicherweise versinkt das Signal dann im Rauschen. Kontrastdaten können wir wiedergewinnen, indem wir Gewichte von Gradientenorientierungen zählen und damit ein Maß erhalten, wie signifikant ein Gradient verglichen mit anderen Gradienten in derselben Zelle ist. Wir schreiben $\|\nabla I_{\mathbf{x}}\|$ für den Betrag des Gradienten am Punkt $\mathbf{x}$ im Bild, C für die Zelle, deren Histogramm wir berechnen möchten, und $w_{\mathbf{x},C}$ für das Gewicht, das für die Orientierung bei $\mathbf{x}$ für diese Zelle zu verwenden ist. Eine naheliegende Wahl für das Gewicht lautet:

$$w_{\mathbf{x},C} = \frac{\|\nabla I_{\mathbf{x}}\|}{\sum_{\mathbf{u}\in C} \|\nabla I_{\mathbf{u}}\|}.$$

Dieser Ausdruck vergleicht den Betrag des Gradienten mit anderen in der Zelle, sodass Gradienten, die verglichen mit ihren Nachbarn groß sind, ein großes Gewicht erhalten. Das resultierende Merkmal wird als **HOG-Merkmal**[2] bezeichnet.

Die Fußgängererkennung unterscheidet sich von der Gesichtserkennung vor allem in Bezug auf diese Merkmalskonstruktion. Im Übrigen werden Fußgängererkennungssysteme fast wie Gesichtserkennungssysteme erstellt. Der Detektor schwenkt ein Fenster über das Bild, berechnet Merkmale für dieses Fenster und übergibt sie dann an einen Klassifizierer. Auf die Ausgabe muss eine submaximale Unterdrückung angewandt werden. In den meisten Anwendungen ist die Skalierung und Orientierung typischer Fußgänger bekannt. Zum Beispiel sind in Anwendungen zum Fahren die Kameras fest auf dem Fahrzeug montiert und wir erwarten hauptsächlich vertikale Fußgänger. Zudem sind wir nur an Fußgängern in der näheren Umgebung interessiert. Es wurden verschiedene Fußgängerdatenmengen veröffentlicht, die sich für das Trainieren des Klassifizierers eignen.

Fußgänger sind aber nicht der einzige Objekttyp, den wir erkennen können. In ▶ Abbildung 24.15 ist zu sehen, dass sich mit ähnlichen Techniken verschiedenste Objekte in unterschiedlichen Kontexten finden lassen.

2 HOG = Histogram of Oriented Gradients, Histogramm orientierter Gradienten

Abbildung 24.15: Ein weiteres Beispiel für Objekterkennung, das mit SIFT[3] arbeitet, einem Vorläufer von HOG. Auf der linken Seite dienen Bilder eines Schuhs und eines Telefons als Objektmodelle. In der Mitte ist ein Testbild zu sehen. Auf der rechten Seite wurden der Schuh und das Telefon erkannt. Dazu wurden Punkte im Bild gesucht, deren SIFT-Merkmalsbeschreibungen mit einem Modell übereinstimmen. Anschließend wurde eine Schätzung der Modellpose berechnet und diese Schätzung verifiziert. Eine starke Übereinstimmung wird normalerweise mit seltenen falschen Positivwerten verifiziert. Bilder von Lowe (1999) © IEEE.

24.4 Rekonstruieren der 3D-Welt

Dieser Abschnitt zeigt, wie wir vom zweidimensionalen Bild zu einer dreidimensionalen Darstellung der Szene gelangen. Dabei ist grundsätzlich folgende Frage zu beantworten: Wenn alle Punkte in der Szene gegeben sind, die entlang eines Strahles auf das Loch fallen und auf denselben Punkt im Bild projiziert werden, wie stellen wir daraus dreidimensionale Informationen wieder her? Hier kommen uns zwei Ideen zu Hilfe:

- Wenn zwei (oder mehr) Bilder von verschiedenen Kamerapositionen vorliegen, können wir durch Triangulation die Position eines Punktes in der Szene finden.
- Wir können Hintergrundwissen über die physikalische Szene nutzen, die zum Bild geführt hat. Für ein Objektmodell $\mathbf{P}(Szene)$ und ein Rendering-Modell $\mathbf{P}(Bild \mid Szene)$ lässt sich eine A-posteriori-Verteilung $\mathbf{P}(Szene \mid Bild)$ berechnen.

Es gibt jedoch immer noch keine vereinheitlichte Theorie für die Szenenrekonstruktion. Die folgenden Abschnitte behandeln im Überblick acht häufig verwendete optische Hinweise: **Bewegung**, **stereoskopisches Sehen** (Stereopsis), **mehrere Ansichten**, **Textur**, **Schatten**, **Konturen** und **bekannte Objekte**.

24.4.1 Bewegungsparallaxe

Bewegt sich die Kamera relativ zur dreidimensionalen Szene, kann die resultierende scheinbare Bewegung im Bild – der optische Fluss – als Informationsquelle sowohl für die Bewegung der Kamera als auch die Tiefe in der Szene dienen. Um dies zu verstehen, stellen wir (ohne Beweis) eine Gleichung auf, die den optischen Fluss mit der Translationsgeschwindigkeit $\mathbf{T}$ des Betrachters und der Tiefe in der Szene in Beziehung setzt.

3 SIFT = Scale Invariant Feature Transform, skaleninvariante Merkmalstransformation

Das optische Flussfeld besteht aus folgenden Komponenten:

$$v_x(x,y) = \frac{-T_x + xT_z}{Z(x,y)}\,,\; v_y(x,y) = \frac{-T_y + yT_z}{Z(x,y)}\,,$$

wobei $Z(x, y)$ die z-Koordinate des Punktes in der Szene ist, die dem Punkt im Bild bei (x, y) entspricht.

Beide Komponenten des optischen Flusses, $v_x(x, y)$ und $v_y(x, y)$, sind am Punkt $x = T_x/T_z$, $y = T_y/T_z$ gleich null. Dieser Punkt wird auch als **Expansionsfokus** des Flussfeldes bezeichnet. Angenommen, wir ändern den Ursprung in der x-y-Ebene, sodass er am Expansionsfokus liegt: Die neuen Ausdrücke für den optischen Fluss nehmen dann eine besonders einfache Form an. Seien (x', y') die neuen Koordinaten, die durch $x' = x - T_x/T_z$, $y' = y - T_y/T_z$ definiert sind. Dann gilt:

$$v_x(x',y') = \frac{x'T_z}{Z(x',y')}\,,\; v_y(x',y')' = \frac{y'T_z}{Z(x',y')}\,.$$

In dieser Gleichung ist der Skalierungsfaktor mehrdeutig. Wenn sich die Kamera doppelt so schnell bewegt und jedes Objekt in der Szene doppelt so groß und doppelt so weit von der Kamera entfernt ist, wäre das optische Flussfeld genau gleich. Dennoch können wir recht nützliche Informationen extrahieren.

1 Angenommen, Sie sind eine Fliege, die versucht, auf einer Mauer zu landen, und Sie wollen wissen, wann Sie mit der aktuellen Geschwindigkeit auf der Mauer auftreffen. Diese Zeit ist gegeben durch Z/T_z. Das unmittelbare optische Flussfeld kann zwar weder die Distanz Z noch die Geschwindigkeitskomponente T_z liefern, sehr wohl aber das Verhältnis der beiden, Es kann deshalb verwendet werden, um den Landeanflug zu kontrollieren. Zahlreiche Experimente haben gezeigt, dass viele unterschiedliche Tierarten diesen Hinweis nutzen.

2 Betrachten Sie zwei Punkte bei den Tiefen Z_1 und Z_2. Auch wenn wir ihre absoluten Werte nicht kennen, können wir doch aus dem Kehrwert des Verhältnisses der Beträge für den optischen Fluss an diesen Punkten das Tiefenverhältnis Z_1/Z_2 ermitteln. Diese sogenannte Bewegungsparallaxe verwenden wir, wenn wir aus dem Seitenfenster eines fahrenden Autos oder Zuges blicken und darauf schließen, dass die sich langsamer bewegenden Teile der Landschaft weiter entfernt liegen.

24.4.2 Binokulares Sehen

Die meisten Wirbeltiere besitzen *zwei* Augen. Dies ist aus Gründen der Redundanz bei Verlust eines Auges nützlich, hilft aber auch in anderer Hinsicht. Bei Beutetieren sitzen die Augen meistens seitlich, um ein weiteres Sichtfeld zu haben. Im Kopf von Raubtieren befinden sich die Augen vorn, sodass sie **stereoskopisch sehen** können. Das Prinzip ähnelt der Bewegungsparallaxe, außer dass anstelle von zeitlich versetzten Bildern zwei (oder) mehr räumlich getrennte Bilder verwendet werden. Weil sich ein bestimmtes Merkmal in der Szene an einem unterschiedlichen Ort relativ zur z-Achse jeder Bildebene befindet, erhalten wir, wenn wir die beiden Bilder überlagern, eine **Verschiedenheit** (**Querdisparation** genannt) in der Position der Bildmerkmale in den beiden Bildern. Das sehen Sie in ▶ Abbildung 24.16, wo der nächstgelegene Punkt der Pyramide im rechten Bild nach links verschoben ist und im linken Bild nach rechts.

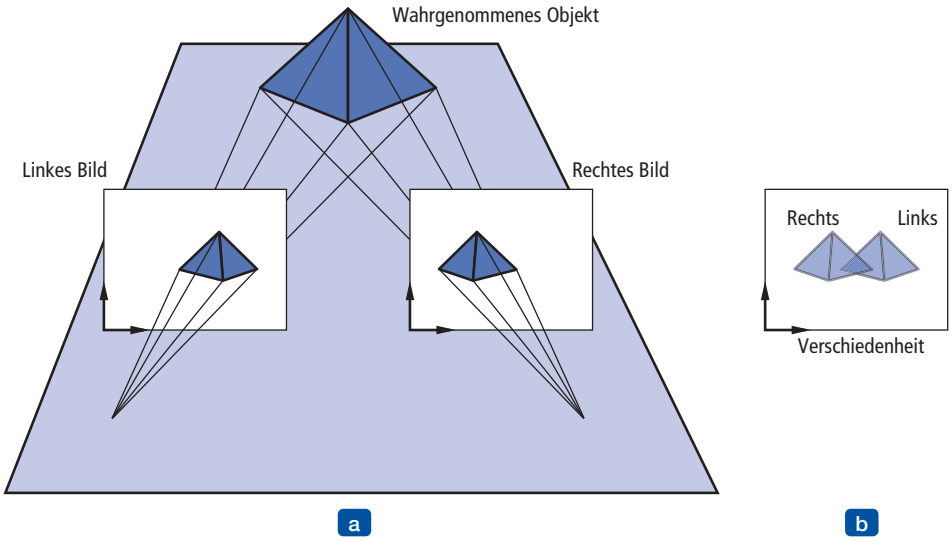


Abbildung 24.16: Das Verrücken einer Kamera parallel zur Bildebene bewirkt, dass sich Bildmerkmale in der Kameraebene verschieben. Die resultierende Verschiedenheit der Bildlage (Querdisparation) ist ein Hinweis auf Tiefe. Wenn wir wie in (b) linkes und rechtes Bild überlagern, wird die Verschiedenheit sichtbar.

Um die Verschiedenheit zu messen, müssen wir das Korrespondenzproblem lösen, d.h. für einen Punkt im linken Bild den Punkt im rechten Bild bestimmen, der aus der Projektion desselben Szenepunktes resultiert. Diese Aufgabe ist mit der Messung des optischen Flusses vergleichbar. Die einfachsten Ansätze ähneln dem Vergleich von Pixelblöcken um entsprechende Punkte mithilfe der Summe der quadrierten Differenzen. In der Praxis verwenden wir komplexere Algorithmen, die zusätzliche Bedingungen berücksichtigen.

Angenommen, wir können die Verschiedenheit messen. Wie kommen wir dann zu Informationen über die Tiefe in der Szene? Hierzu müssen wir die geometrische Beziehung zwischen Verschiedenheit und Tiefe aufstellen. Zuerst betrachten wir den Fall, wenn beide Augen (oder Kameras) geradeaus sehen, wobei ihre optischen Achsen parallel verlaufen. Die Beziehung der rechten Kamera zur linken ist dann einfach die Verschiebung entlang der x-Achse um einen Betrag b, die Grundlinie. Wir können die Gleichungen für den optischen Fluss aus dem vorherigen Abschnitt benutzen, wenn wir dies als Ergebnis einer Verschiebung des Vektors $\mathbf{T}$, der über die Zeit δt wirkt, auffassen, wobei $T_x = b / \delta t$ und $T_y = T_z = 0$ ist. Horizontale und vertikale Verschiedenheit ergeben sich dann aus den Komponenten des optischen Flusses multipliziert mit dem Zeitschritt δt, $H = v_x \delta t$, $V = v_y \delta t$. Wenn wir die Substitutionen ausführen, erhalten wir das Ergebnis, dass $H = b/Z$ und $V = 0$ ist. Die horizontale Verschiedenheit ist also gleich dem Verhältnis von Grundlinie zu Tiefe und die vertikale Verschiedenheit ist 0. Vorausgesetzt, dass wir b kennen, können wir H messen und die Tiefe Z rekonstruieren.

Unter normalen Sichtbedingungen **fixieren** Menschen; das bedeutet, es gibt einen Punkt in der Szene, an dem sich die optischen Achsen beider Augen schneiden. ▶ Abbildung 24.17 zeigt zwei Augen, die einen Punkt P_0 fixieren, der sich in einer Distanz Z vom Mittelpunkt der Augen befindet. Der Einfachheit halber berechnen wir die *Winkelverschiedenheit*, gemessen im Bogenmaß. Die Verschiedenheit am Fixierungspunkt P_0 ist null. Für einen anderen Punkt P in der Szene, der sich δZ weiter entfernt befindet, können wir die Winkelverschiebungen der linken und rechten Bilder von P berechnen, die

wir als P_L und P_R bezeichnen wollen. Wenn jedes davon um einen Winkel von $\delta\theta/2$ relativ zu P_0 verschoben wird, dann ist Verschiebung zwischen P_L und P_R, nämlich die Verschiedenheit von P, einfach gleich $\delta\theta$. Gemäß Abbildung 24.17 ist

$$\tan\theta = \frac{b/2}{Z} \quad \text{und} \quad \tan(\theta - \delta\theta/2) = \frac{b/2}{Z + \delta Z}.$$

Für kleine Winkel gilt aber $\tan\theta \approx \theta$. Somit ergibt sich

$$\delta\theta/2 = \frac{b/2}{Z} - \frac{b/2}{Z+\delta Z} \approx \frac{b\delta Z}{2Z^2}$$

und da die eigentliche Verschiedenheit $\delta\theta$ ist, erhalten wir

$$\text{Verschiedenheit} = \frac{b\delta Z}{Z^2}.$$

Beim Menschen ist b (die **Grundlinie**) etwa 6 cm. Angenommen, Z befindet sich bei 100 cm. Das kleinste erkennbare $\delta\theta$ (entsprechend der Pixelgröße) beträgt dann etwa 5 Bogensekunden, womit wir ein δZ von 0,4 mm erhalten. Für Z = 30 cm erhalten wir den beeindruckend kleinen Wert δZ = 0,036 mm. Das bedeutet, bei einer Entfernung von 30 cm können Menschen Tiefen unterscheiden, die sich um einen so kleinen Betrag wie 0,036 Millimeter unterscheiden, sodass wir beispielsweise Nadeln und Ähnliches erkennen können.

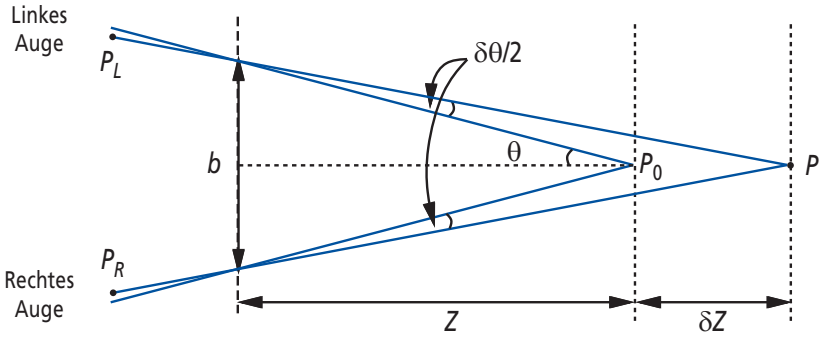


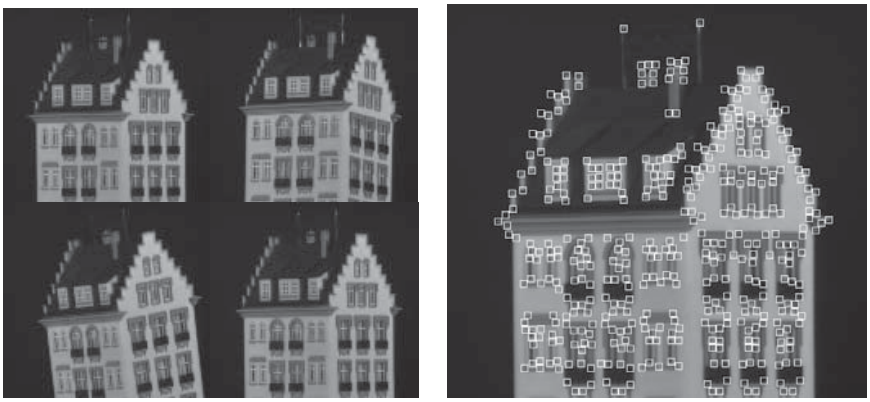
Abbildung 24.17: Die Beziehung zwischen Verschiedenheit und Tiefe beim stereoskopischen Sehen. Die Projektionszentren der beiden Augen liegen b weit voneinander entfernt und die optischen Achsen schneiden sich am Fixierungspunkt P_0. Der Punkt P in der Szene wird auf die Punkte P_L und P_R in den beiden Augen projiziert. Als Winkel ausgedrückt beträgt die Verschiedenheit zwischen diesen Punkten $\delta\theta$. (Siehe den Text.)

24.4.3 Mehrere Ansichten

Die Gestalt aus optischem Fluss oder binokularer Verschiedenheit zu erkennen, sind zwei Instanzen eines allgemeinen Gerüstes, das sich mehrerer Ansichten für die Rückgewinnung von Tiefe bedient. In der Computervision gibt es keinen Grund, sich auf differentielle Bewegung einzuschränken oder nur zwei Kameras zu verwenden, die auf einen Fixierungspunkt ausgerichtet sind. Demzufolge wurden Techniken entwickelt, die auf Daten aus mehreren Ansichten zurückgreifen, ja sogar von Hunderten oder Tausenden Kameras stammen können. Aus algorithmischer Sicht sind dabei drei Teilprobleme zu lösen:

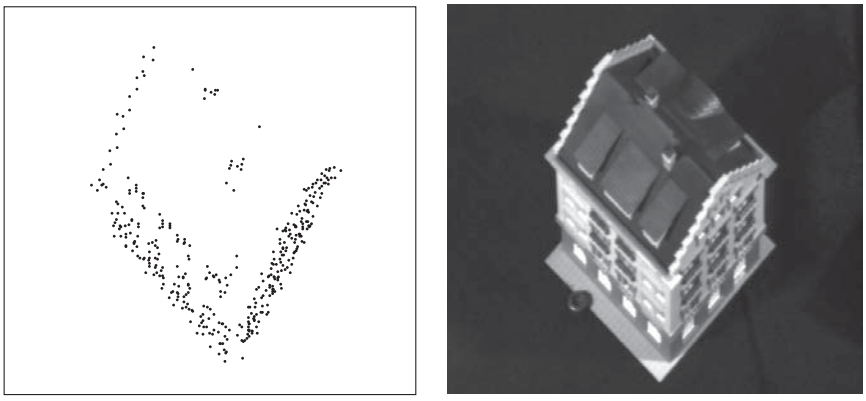
- Das Korrespondenzproblem, d.h. das Identifizieren von Merkmalen in den verschiedenen Bildern, die Projektionen desselben Merkmals in der dreidimensionalen Welt sind.
- Das relative Orientierungsproblem, d.h. die Ermittlung der Transformation (Rotation und Translation) zwischen den Koordinatensystemen, die mit den verschiedenen Kameras verankert sind.
- Das Problem der Tiefenbewertung, d.h. die Ermittlung der Tiefe von verschiedenen Punkten in der Welt, für die in mindestens zwei Ansichten Bildebenenprojektionen verfügbar waren.

Die Entwicklung von robusten Vergleichsprozeduren für das Korrespondenzproblem in Verbindung mit numerisch stabilen Algorithmen, mit denen sich relative Orientierungen und die Szenentiefe berechnen lassen, gehört zu den Erfolgsgeschichten der Computervision. Ergebnisse eines solchen Ansatzes von Tomasi und Kanade (1992) sind in ▶ Abbildung 24.18 und ▶ Abbildung 24.19 gezeigt.



a b

Abbildung 24.18: (a) Vier Einzelbilder aus einem Videofilm, wobei die Kamera relativ zum Objekt bewegt und gedreht wird. (b) Das erste Einzelbild der Folge, wobei kleine Rechtecke die Merkmale kennzeichnen, die der Funktionsmerkmal-Detektor erkannt hat. (Mit freundlicher Genehmigung von Carlo Tomasi.)



a b

Abbildung 24.19: (a) Dreidimensionale Rekonstruktion der Positionen der Bildmerkmale in Abbildung 24.18, von oben gesehen. (b) Das reale Haus, aufgenommen aus derselben Position.

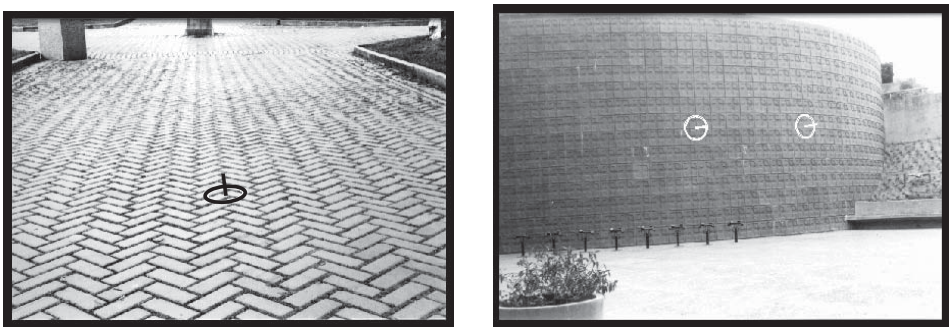
24.4.4 Textur

Weiter vorn haben wir gesehen, wie Texturen für das Segmentieren von Objekten verwendet wurden. Sie lassen sich aber auch nutzen, um Entfernungen zu bewerten. ▶ Abbildung 24.20 zeigt, dass eine homogene Textur in der Szene zu variierenden Texturelementen – sogenannten **Texeln** – im Bild führt. Alle Pflastersteine in (a) sind in der Szene identisch. Im Bild erscheinen sie aus zwei Gründen unterschiedlich:

1. *Unterschiede in den Abständen der Texel zur Kamera.* Entfernte Objekte erscheinen um einen Skalierungsfaktor von $1/Z$ kleiner.

2. *Unterschiede in der perspektivischen Verkürzung der Texel.* Wenn sich alle Texel auf der Grundebene befinden, weicht der Betrachtungswinkel weiter entfernter Texel mehr von der Senkrechten ab und diese Texel erscheinen stärker verkürzt. Die Größe des Effekts der perspektivischen Verkürzung ist proportional zu $\cos \sigma$, wobei σ die Ebenenneigung angibt, d.h. den Winkel zwischen der Z-Achse und $\mathbf{n}$, der Oberflächennormalen zum Texel.

Verschiedene Algorithmen versuchen, aus der Variation in der Erscheinung der projizierten Texel die Oberflächennormalen zu ermitteln. Allerdings sind Genauigkeit und Anwendbarkeit dieser Algorithmen nicht überall so allgemeingültig wie bei denjenigen, die sich auf mehrere Ansichten stützen.



a b

Abbildung 24.20: (a) Eine texturierte Szene. Unter der Annahme, dass die reale Textur einheitlich ist, können wir die Oberflächenausrichtung ermitteln. Angezeigt wird die berechnete Oberflächenausrichtung durch einen schwarzen Kreis und einen Zeiger, die über das Bild gelegt und so transformiert sind, als wäre der Kreis an diesem Punkt auf die Oberfläche gemalt worden. (b) Erkennung des Umrisses aus der Textur für eine gekrümmte Oberfläche (dieses Mal mit weißem Kreis und Zeiger). Bilder mit freundlicher Genehmigung von Jitendra Malik und Ruth Rosenholtz (1994).

24.4.5 Schattierung

Die **Schattierung** – die Variationen in der Intensität des Lichtes, das von verschiedenen Bereichen einer Oberfläche in einer Szene ausgeht – wird durch die Geometrie der Szene sowie die Reflexionseigenschaften der Oberflächen bestimmt. In der Computergrafik ist es das Ziel, die Bildhelligkeit $I(x, y)$ für die Szenengeometrie und die Reflexionseigenschaften der Objekte in der Szene zu berechnen. Die Computervision dagegen versucht, diesen Prozess umzukehren. Das bedeutet, sie will die Geometrie

und die Reflexionseigenschaften für die gegebene Bildhelligkeit $I(x, y)$ ermitteln. Das hat sich außer in den einfachsten Fällen als sehr schwierig erwiesen.

Aus dem physikalischen Modell gemäß *Abschnitt 24.1.4* wissen wir: Zeigt eine Oberflächennormale gegen die Lichtquelle, ist die Oberfläche heller, zeigt sie in die abgewandte Richtung, erscheint die Oberfläche dunkler. Wir können allerdings nicht schließen, dass die Normale eines dunklen Bereiches vom Licht weg zeigt; stattdessen könnte er eine niedrige Albedo besitzen. Im Allgemeinen ändern sich die Albedo in Bildern recht schnell und die Schattierung ziemlich langsam. Menschen scheinen anhand solcher Beobachtungen gut zu erkennen, ob geringe Beleuchtung, Oberflächenausrichtung oder Albedo für einen dunklen Bereich verantwortlich sind. Um das Problem zu vereinfachen, nehmen wir an, dass die Albedo für jeden Oberflächenpunkt bekannt ist. Trotzdem ist es schwierig, die Normale zu ermitteln, denn die Bildhelligkeit ist lediglich *ein* Maß, die Normale hat aber zwei unbekannte Parameter, sodass wir nicht einfach nach der Normalen auflösen können. Der Schlüssel für diese Situation scheint darin zu liegen, dass nahe beieinanderliegende Normalen ähnlich sein werden, da die meisten Oberflächen glatt sind – sie weisen keine abrupten Änderungen auf.

Die eigentliche Schwierigkeit ist die Verarbeitung von Interreflexionen. Wenn wir eine typische Szene in einem geschlossenen Raum betrachten, wie beispielsweise die Objekte in einem Büro, werden die Oberflächen nicht nur von den Lichtquellen beleuchtet, sondern auch von dem Licht, das von den anderen Oberflächen in der Szene reflektiert wird, die damit letztlich als sekundäre Lichtquellen dienen. Diese wechselseitigen Beleuchtungseffekte sind recht erheblich und erschweren es, die Beziehung zwischen der Normalen und der Bildhelligkeit zu berechnen. Zwei Oberflächenbereiche mit der gleichen Normalen haben möglicherweise deutlich verschiedene Helligkeiten, wenn der eine das von einer großen weißen Wand reflektierte Licht empfängt und der andere nur einem dunklen Bücherschrank zugewandt ist.

Trotz dieser Schwierigkeiten ist dies ein ernstzunehmendes Problem. Menschen sind offenbar in der Lage, die Wirkungen von Interreflexionen auszublenden und sich eine brauchbare Wahrnehmung von Umrissen aus Schattierungen zu verschaffen, doch wissen wir enttäuschend wenig über Algorithmen, die dies bewerkstelligen.

24.4.6 Konturen

Wenn wir eine Strichzeichnung wie die in ▶ Abbildung 24.21 betrachten, erhalten wir eine lebendige Wahrnehmung des dreidimensionalen Umrisses und des Layouts. Wie? Es handelt sich um eine Kombination aus Erkennung bekannter Objekte in der Szene und Anwendung von generischen Bedingungen wie zum Beispiel:

- Verdeckende Konturen, beispielsweise die Umrisse der Berge. Eine Seite der Kontur liegt näher am Betrachter, die andere Seite weiter weg. Merkmale wie lokale Konvexität und Symmetrie liefern Hinweise, um das **Figur-Grund-Problem** zu lösen – die Zuordnung, welche Seite der Kontur Figur (mehr im Vordergrund) und welche Grund (mehr im Hintergrund) ist. An einer verdeckenden Kontur liegt die Sichtlinie tangential zur Oberfläche in der Szene.
- T-Verbindungen. Verdeckt ein Objekt ein anderes, wird die Kontur des weiter entfernten Objektes unterbrochen, sofern das näher liegende Objekt undurchsichtig ist. Das Ergebnis ist eine T-Verbindung im Bild.

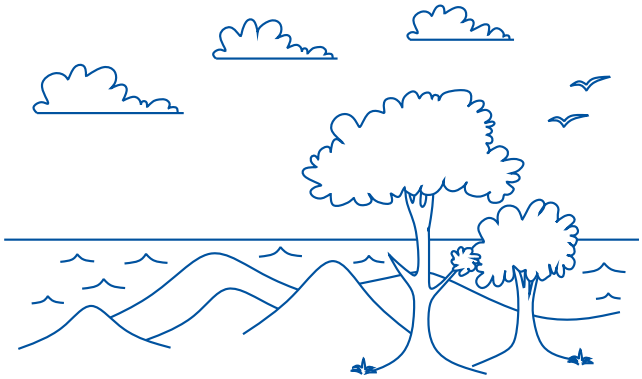


Abbildung 24.21: Eine plastische Strichzeichnung. (Mit freundlicher Genehmigung von Isha Malik.)

- Position auf der Grundebene. Menschen wie auch viele andere Landtiere sind oft in einer Szene anzutreffen, die eine **Grundebene** enthält, wobei sich verschiedene Objekte bei unterschiedlichen Positionen auf dieser Ebene befinden. Bedingt durch die Schwerkraft schweben typische Objekte nicht in der Luft, sondern werden von dieser Grundebene getragen. Somit können wir uns die besondere räumliche Geometrie dieses Betrachtungsszenarios zunutze machen.

Wie sieht nun die Projektion von Objekten in unterschiedlichen Höhen und bei unterschiedlichen Orten auf der Grundebene aus? Wir nehmen an, dass sich das Auge (oder die Kamera) bei einer Höhe h_c über der Grundebene befindet. Betrachten Sie ein Objekt der Höhe δY, das auf der Grundebene ruht. Seine Unterseite liegt bei $(X, -h_c, Z)$ und die Oberseite bei $(X, \delta Y - h_c, Z)$. Die Unterseite wird auf den Bildpunkt $(fX/Z, -fh_c/Z)$ und die Oberseite auf $(fX/Z, f(\delta Y - h_c)/Z)$ projiziert. Die Unterseiten von näher liegenden Objekten (kleines Z) werden auf niedrigere Punkte in der Bildebene projiziert; bei weiter entfernten Objekten befinden sich die Unterseiten näher am Horizont.

24.4.7 Objekte und die geometrische Struktur von Szenen

Der Kopf eines durchschnittlichen Erwachsenen ist etwa 23 cm groß. Für eine 1,30 m entfernte Person beträgt dann der Winkel, unter dem der Kopf der Kamera erscheint, 1 Grad. Wenn wir eine Person sehen, deren Kopf in einem Öffnungswinkel von nur einem halben Grad erscheint, ist gemäß Bayesscher Inferenz davon auszugehen, dass wir auf eine normale Person blicken, die 2,60 m entfernt ist, und nicht auf eine, deren Kopf nur halb so groß ist. Entsprechend dieser Argumentation gelangen wir zu einer Methode, um die Ergebnisse eines Fußgänger-Detektors zu überprüfen, sowie zu einer Methode, um die Entfernung zu einem Objekt abzuschätzen. Zum Beispiel weisen alle Fußgänger ungefähr die gleiche Größe auf und stehen normalerweise auf einer Grundebene. Wenn wir wissen, wo sich in einem Bild der Horizont befindet, können wir Fußgänger nach der Entfernung zur Kamera ordnen. Das funktioniert, weil wir wissen, wo sich ihre Füße befinden. Und Fußgänger, deren Füße im Bild näher zum Horizont stehen, sind von der Kamera weiter entfernt (siehe ▶ Abbildung 24.22). Fußgänger, die sich weiter entfernt von der Kamera befinden, müssen zudem im Bild kleiner sein. Folglich können wir einige Detektorantworten ausschließen. Findet ein Detektor einen Fußgänger, der im Bild groß ist und dessen Füße näher am Horizont stehen, hat er einen Riesenfußgänger gefunden. Da diese nicht existieren, ist die Antwort des Detektors falsch. Tat-

sächlich sind viele oder die meisten Bildfenster keine akzeptablen Fußgängerfenster und brauchen dem Detektor gar nicht erst präsentiert zu werden.

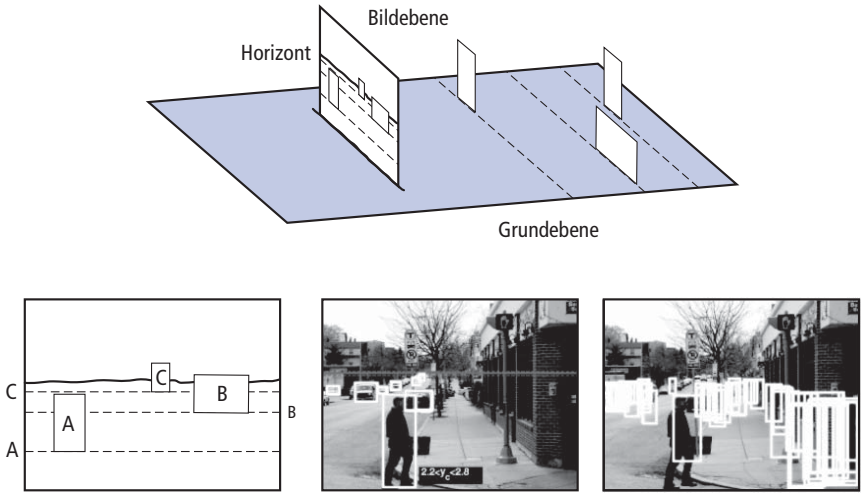


Abbildung 24.22: In einem Bild mit Menschen, die auf einer Grundebene stehen, müssen die Personen, deren Füße sich näher am Horizont befinden, weiter entfernt sein (obere Zeichnung). Das heißt, sie müssen im Bild kleiner aussehen (linke untere Zeichnung). Folglich hängen Größe und Position von realen Fußgängern in einem Bild voneinander und von der Position in Bezug auf den Horizont ab. Um dies zu nutzen, müssen wir die Grundebene identifizieren, was sich mithilfe von Umriss-aus-Textur-Methoden realisieren lässt. Aus diesen Informationen und von einigen mit hoher Wahrscheinlichkeit erkannten Fußgängern können wir einen Horizont rekonstruieren, wie er im mittleren Bild zu sehen ist. Im rechten Bild sind entsprechend diesem geometrischen Kontext akzeptable Kästchen um Fußgänger eingezeichnet. Die Fußgänger, die in der Szene höher stehen, müssen kleiner sein. Sind sie es nicht, handelt es sich um falsche Positivergebnisse. Bilder von Hoiem et al. (2008) © IEEE.

Der Horizont lässt sich nach verschiedenen Strategien finden. Unter anderem kann man nach einer etwa waagerecht verlaufenden Linie suchen, über der viel Blau zu sehen ist, oder auf Schätzungen der Oberflächenausrichtung zurückgreifen, die aus Texturverformungen gewonnen wurden. Eine elegantere Strategie macht sich die Umkehrung unserer geometrischen Bedingungen zunutze. Ein einigermaßen zuverlässiger Fußgänger-Detektor ist in der Lage, Schätzungen des Horizonts zu erzeugen, wenn es in der Szene mehrere Fußgänger bei verschiedenen Abständen gibt. Denn die relative Skalierung der Fußgänger liefert einen Anhaltspunkt, wo sich der Horizont befindet. Somit können wir eine Horizontabschätzung vom Detektor extrahieren und dann anhand dieser Schätzung die Fehler des Fußgänger-Detektors bereinigen.

Wenn das Objekt bekannt ist, können wir mehr als nur den Abstand zu ihm abschätzen, denn wie es im Bild aussieht, hängt in großem Maße von seiner Pose ab, d.h. seiner Position und Ausrichtung in Bezug auf den Betrachter. Hierfür gibt es zahlreiche Anwendungen. Zum Beispiel kann in einer industriellen Handhabungstechnik ein Roboterarm nur dann ein Objekt aufnehmen, wenn dessen Pose bekannt ist. Bei starren Objekten, egal ob zwei- oder dreidimensional, hat dieses Problem eine einfache und wohl definierte Lösung, die auf der **Ausrichtungsmethode** basiert, die wir nun entwickeln werden.

Das Objekt wird durch M Merkmale oder unterschiedliche Punkte m_1, m_2, ..., m_M im dreidimensionalen Raum dargestellt – vielleicht die Ecken eines polyedrischen Objektes. Sie werden im selben Koordinatensystem gemessen, das auch für das Objekt

natürlich ist. Dann werden die Punkte einer unbekannten dreidimensionalen Drehung **R** unterzogen, gefolgt von einer Translation um einen unbekannten Betrag **t** sowie einer Projektion, um die Bildmerkmalspunkte p_1, p_2, ...p_N auf der Bildebene entstehen zu lassen. Im Allgemeinen gilt auch $N \neq M$, weil einige Modellpunkte verdeckt sein können und der Merkmalsdetektor möglicherweise einige Merkmale übersieht (oder aufgrund von Rauschen falsche Merkmale erfindet). Dies können wir ausdrücken als

$$p_i = \prod(\mathbf{R}m_i + \mathbf{t}) = Q(m_i)$$

für einen dreidimensionalen Modellpunkt m_i und den entsprechenden Bildpunkt p_i. Hier ist **R** eine Rotationsmatrix, **t** ist eine Translation und Π gibt die perspektivische Projektion oder eine ihrer Annäherungen an, wie beispielsweise eine skalierte orthographische Projektion. Das Endergebnis ist eine Transformation Q, die den Modellpunkt m_i mit dem Bildpunkt p_i ausrichtet. Obwohl wir Q anfänglich nicht kennen, wissen wir (für starre Objekte), dass Q für alle Modellpunkte *gleich* sein muss.

Man kann nach Q auflösen, wenn man die dreidimensionalen Koordinaten der drei Modellpunkte und ihre zweidimensionalen Projektionen kennt. Der Gedanke dabei ist der folgende: Man kann Gleichungen aufschreiben, die die Koordinaten von p_i mit denen von m_i in Beziehung setzen. In diesen Gleichungen entsprechen die unbekannten Größen den Parametern der Rotationsmatrix **R** und des Translationsvektors **t**. Wenn wir ausreichend viele Gleichungen haben, sollten wir in der Lage sein, nach Q aufzulösen. Den Beweis werden wir hier nicht zeigen; wir geben lediglich das folgende Ergebnis an:

> Für drei nicht kollineare Punkte m_1, m_2 und m_3 im Modell und ihre skalierten orthographischen Projektionen p_1, p_2 und p_3 auf der Bildebene gibt es genau zwei Transformationen vom dreidimensionalen Modellkoordinatenrahmen in einen zweidimensionalen Bildkoordinatenrahmen.

Diese Transformationen sind durch eine Reflexion um die Bildebene miteinander verknüpft und können durch eine einfache geschlossene Lösung berechnet werden. Wenn wir die entsprechenden Modellmerkmale für drei Merkmale im Bild identifizieren könnten, ließe sich auch Q berechnen, die Pose des Objektes.

Position und Ausrichtung spezifizieren wir nun in mathematischer Form. Die Position eines Punktes P in der Szene ist durch drei Zahlen charakterisiert, die (X, Y, Z)-Koordinaten von P in einem Koordinatensystem, dessen Ursprung in der Lochblende liegt und dessen Z-Achse entlang der optischen Achse verläuft (siehe Abbildung 24.2 in *Abschnitt 24.1.1*). Wir verfügen nun über die perspektivische Projektion (x, y) des Punktes im Bild. Diese spezifiziert den von der Lochblende ausgehenden Strahl, auf dem P liegt. Allerdings ist die Entfernung unbekannt. Der Begriff „Ausrichtung" kann in doppeltem Sinn verwendet werden:

1. **Die Ausrichtung des Objektes als Ganzes:** Dies lässt sich in Form einer dreidimensionalen Rotation spezifizieren, wobei sein Koordinatenrahmen mit dem der Kamera verknüpft wird.

2. **Die Ausrichtung der Objektoberfläche bei *P*:** Dies lässt sich durch einen Normalvektor **n** spezifizieren – d.h. einen Vektor, der die Richtung senkrecht zur Oberfläche angibt. Oftmals drücken wir die Oberflächenausrichtung mithilfe der Variablen **Schräge** und **Neigung** aus. Bei Schräge handelt es sich um den Winkel zwischen der Z-Achse und **n**. Neigung ist der Winkel zwischen der X-Achse und der Projektion von **n** auf die Bildebene.

Wenn sich die Kamera relativ zu einem Objekt bewegt, ändern sich sowohl die Distanz als auch die Ausrichtung des Objektes. Beibehalten wird der **Umriss** des Objektes. Ist das Objekt ein Würfel, ändert sich daran nichts, wenn sich das Objekt bewegt. Geometer haben jahrhundertelang versucht, Umrisse zu formalisieren: das grundlegende Konzept, dass der Umriss unverändert bleibt, wenn eine bestimmte Folge von Transformationen angewendet wird – beispielsweise Kombinationen aus Rotation und Translationen. Die Schwierigkeit liegt darin, eine Repräsentation des globalen Umrisses zu finden, die allgemein genug ist, um mit der großen Vielzahl von Objekten in der realen Welt zurechtzukommen – nicht nur mit einfachen Formen wie Zylinder, Kegel oder Kugel – und die dennoch von der visuellen Eingabe leicht erkannt werden kann. Das Problem, den *lokalen* Umriss einer Oberfläche zu charakterisieren, ist sehr viel besser untersucht worden. Im Wesentlichen kann dies mithilfe von Krümmungen ausgedrückt werden: Wie ändert sich die Oberflächennormale, wenn man sich auf der Oberfläche in verschiedene Richtungen bewegt? Für eine Ebene gibt es überhaupt keine Änderung. Für einen Zylinder gibt es, wenn man sich parallel zur Achse bewegt, keine Änderung; bewegt man sich in senkrechter Richtung, dreht sich die Oberflächennormale umgekehrt proportional zum Radius des Zylinders usw. Diese Themen werden in der Differentialgeometrie betrachtet.

Der Umriss eines Objektes ist für einige Aufgaben der Handhabungstechnik entscheidend (um beispielsweise zu entscheiden, wie man ein Objekt greift), aber seine wichtigste Rolle hat er in der Objekterkennung, wobei der geometrische Umriss sowie die Farbe und die Oberflächenstruktur die wichtigsten Hinweise darstellen, die es uns ermöglichen, Objekte zu identifizieren und zu klassifizieren, was sich im Bild befindet, wenn wir bereits ein Beispiel einer solchen Klasse gesehen haben, usw.

24.5 Objekterkennung aus Strukturinformationen

Wenn man Kästen um Fußgänger in einem Bild legt, kann dies bereits genügen, um beim Fahren einen Zusammenstoß mit ihnen zu vermeiden. Wie weiter vorn gezeigt, lässt sich ein Kasten ermitteln, indem man die durch Ausrichtungen gelieferten Beobachtungen bündelt und mithilfe von Histogrammmethoden möglicherweise irritierende räumliche Details unterdrückt. Möchten wir mehr darüber wissen, was eine Person tut, müssen wir wissen, wo Arme, Beine, Körper und Kopf der Person im Bild liegen. Einzelne Körperteile für sich sind mithilfe der Methode mit verschobenen Fenstern recht schwierig zu erkennen, da ihre Farbe und Textur stark variieren kann und da sie in Bildern üblicherweise klein sind. Oftmals sind Unterarme und Schienbeine nur zwei oder drei Pixel breit. Rumpfteile erscheinen normalerweise nicht für sich allein und die Darstellung, was womit verbunden ist, könnte recht hilfreich sein, weil leicht aufzufindende Teile uns sagen, wo nach kleinen und schwer zu erkennenden Teilen zu suchen ist.

Das Layout von menschlichen Körpern herzuleiten, ist in der Computervision eine wichtige Aufgabe, da das Layout des Körpers oftmals verrät, was Personen tun. Ein als **verformbare Vorlage** bezeichnetes Modell kann uns sagen, welche Konstellationen brauchbar sind: Der Ellenbogen lässt sich beugen, doch der Kopf ist niemals mit dem Fuß verbunden. Das einfachste verformbare Vorlagenmodell einer Person verbindet Unterarme mit Oberarmen, Oberarme mit dem Torso usw. Es gibt auch umfangreichere Modelle: Zum Beispiel könnten wir die Tatsache darstellen, dass linke und rechte Oberarme meistens die gleiche Farbe und Textur aufweisen, genau wie bei linken und rechten Beinen. Allerdings ist es immer noch schwierig, mit diesen umfangreicheren Modellen zu arbeiten.

24.5.1 Die Geometrie von Körpern: Arme und Beine suchen

Fürs Erste nehmen wir an, dass wir das Aussehen der Körperteile einer Person (z.B. Farbe und Textur der Kleidung) kennen. Die Geometrie des Körpers lässt sich als Baum mit elf rechteckigen Segmenten (obere und untere linke und rechte Arme und Beine, ein Torso, ein Gesicht und Haar über dem Gesicht) modellieren. Wir nehmen an, dass Position und Ausrichtung (**Pose**) des linken Unterarms unabhängig von allen anderen Segmenten ist, wenn die Pose des linken Oberarms gegeben ist, dass die Pose des linken Oberarms unabhängig von allen Segmenten ist, wenn die Pose des Torsos gegeben ist, und erweitern diese Annahmen sinngemäß für den rechten Arm und die Beine, das Gesicht und das Haar. Derartige Modelle nennt man auch „Kartonmenschen"-Modelle. Das Modell bildet eine Baumstruktur, deren Wurzel üblicherweise durch den Torso dargestellt wird. Wir durchsuchen das Bild nach der besten Übereinstimmung bezüglich dieses Kartonmenschen mithilfe von Inferenzmethoden für ein Bayessches Netz in Baumstruktur (siehe *Kapitel 14*).

Für die Auswertung einer Konstellation gibt es zwei Kriterien. Erstens sollte ein Bildrechteck genau wie sein Segment aussehen. Momentan ist noch unklar, was dies genau bedeutet. Wir nehmen aber an, dass eine Funktion ϕ_i ein Maß liefert, wie gut ein Bildrechteck mit einem Körperabschnitt übereinstimmt. Für jedes Paar verknüpfter Segmente haben wir eine weitere Funktion ψ, die bewertet, wie gut Beziehungen zwischen einem Paar von Bildrechtecken mit denen übereinstimmen, die von den Körpersegmenten erwartet werden. Die Abhängigkeiten zwischen Segmenten bilden einen Baum, sodass jedes Segment nur genau ein übergeordnetes Element besitzt und wir $\psi_{i,\ pa(i)}$ schreiben können. Alle Funktionen liefern für bessere Übereinstimmungen größere Werte, sodass wir sie uns als logarithmische Wahrscheinlichkeit vorstellen können. Die Kosten einer bestimmten Übereinstimmung, die das Bildrechteck m_i dem Körpersegment i zuordnet, betragen dann

$$\sum_{i \in \text{Segmente}} \phi_i(m_i) + \sum_{i \in \text{Segmente}} \psi_{i,\text{pa}(i)}\left(m_i, m_{\text{pa}(i)}\right).$$

Die beste Übereinstimmung lässt sich per dynamischer Programmierung finden, da das relationale Modell ein Baum ist.

Es ist unzweckmäßig, eine kontinuierliche Datenmenge zu durchsuchen. Deshalb überführen wir den Raum der Bildrechtecke in einen Raum mit diskreten Werten durch Diskretisierung von Position und Ausrichtung der Rechtecke fester Größe (wobei die Größen für verschiedene Segmente unterschiedlich sein können). Da Knöchel und Knie verschieden sind, müssen wir zwischen einem Rechteck und demselben Rechteck, das um 180° gedreht ist, unterscheiden. Das Ergebnis ließe sich als Menge von sehr großen Stapeln kleiner Bildrechtecke visualisieren, die an unterschiedlichen Orten und Ausrichtungen ausgeschnitten wurden. Pro Segment gibt es einen Stapel. Wir müssen nun die beste Zuordnung von Rechtecken zu Segmenten finden. Wegen der vielen Bildrechtecke ist dies ein langsamer Vorgang. Für das hier angegebene Modell beträgt die Zeit für die Auswahl des richtigen Torsos $O(M^6)$, wenn M Bildrechtecke vorhanden sind. Für eine passende Wahl von ψ sind allerdings verschiedene beschleunigte Verfahren verfügbar und die Methode lässt sich praktisch nutzen (siehe ▶ Abbildung 24.23). Das Modell bezeichnet man üblicherweise als **Bildstrukturmodell**.

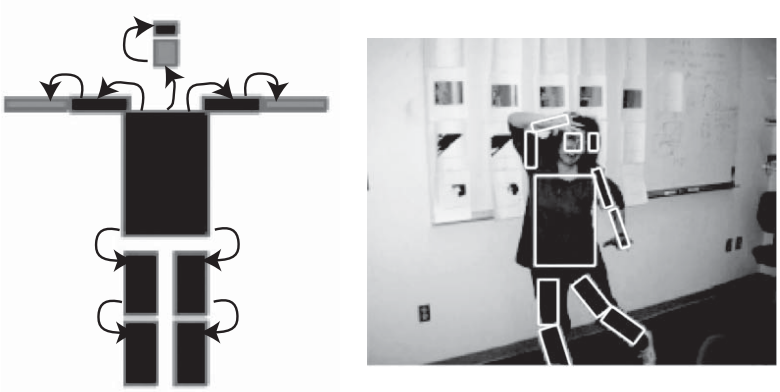


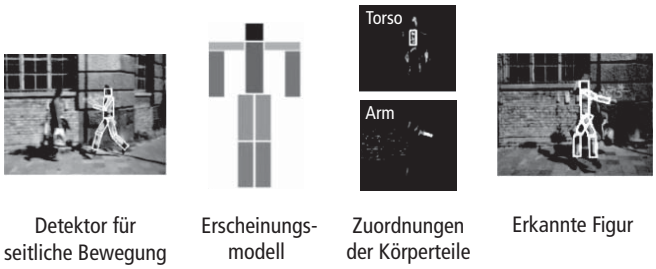
Abbildung 24.23: Ein Bildstrukturmodell beurteilt eine Übereinstimmung zwischen einer Menge von Bildrechtecken und eines Kartonmenschen (links gezeigt), indem die Ähnlichkeit in der Erscheinung zwischen Körpersegmenten und Bildsegmenten sowie die räumlichen Beziehungen zwischen den Bildsegmenten bewertet werden. Eine Übereinstimmung ist im Allgemeinen besser, wenn Segmente ungefähr die richtige Erscheinung besitzen und sie sich ungefähr an der richtigen Stelle in Bezug zueinander befinden. Das Erscheinungsmodell verwendet durchschnittliche Farben für Haar, Kopf und Torso sowie die oberen und unteren Arme und Beine. Die relevanten Beziehungen sind als Pfeile dargestellt. Der rechte Teil der Abbildung zeigt die beste Übereinstimmung für ein bestimmtes Bild, das mithilfe dynamischer Programmierung erhalten wurde. Die Übereinstimmung ist eine brauchbare Schätzung für die Konstellation des Körpers. Abbildung von Felzenszwalb und Huttenlocher (2000) © IEEE.

Entsprechend unserer oben getroffenen Annahme müssen wir wissen, wie die Person aussieht. Wenn wir eine Person in einem einzigen Bild suchen, erweist sich die Farbe als nützlichstes Merkmal, um Segmentübereinstimmungen zu bewerten. Texturmerkmale funktionieren in den meisten Fällen nicht gut, da Falten auf loser Kleidung starke Schattenmuster hervorrufen, die die Bildtextur überlagern. Diese Muster sind stark genug, um die wirkliche Textur des Gewebes zu stören. In der aktuellen Arbeit spiegelt ψ normalerweise die Notwendigkeit wider, dass die Enden der Segmente sinnvoll zusammengeschlossen sind, doch gibt es üblicherweise keine Bedingungen bezüglich der Winkel. Im Allgemeinen wissen wir nicht, wie eine Person aussieht, und müssen ein Modell von Segmenterscheinungen erstellen. Die Beschreibung, wie eine Person aussieht, nennen wir das **Erscheinungsmodell** (Appearance Model). Wenn wir die Konstellation einer Person in einem einzelnen Bild ermitteln müssen, können wir von einem schlecht optimierten Erscheinungsmodell ausgehen, mit diesem die Konstellation abschätzen, dann die Erscheinung neu bewerten usw. In Videos haben wir viele Einzelbilder (Frames) derselben Person und dies offenbart deren Erscheinung.

24.5.2 Kohärente Erscheinung: Personen im Video verfolgen

Das Verfolgen von Personen in Videos ist eine wichtige Aufgabe in der Praxis. Wenn wir die Position von Armen, Beinen, Torso und Kopf in Videosequenzen zuverlässig erfassen können, sind wir in der Lage, wesentlich verbesserte Spielschnittstellen und Überwachungssysteme aufzubauen. Filtermethoden waren bei diesem Problem nur wenig erfolgreich, da Personen große Beschleunigungen erzeugen und sich ziemlich schnell bewegen können. Das heißt, dass bei einem 25-Hz-Video die Gestalt des Körpers in Einzelbild i die Gestalt des Körpers in Einzelbild $i + 1$ nicht allzu stark einschränkt. Derzeit nutzen die effektivsten Methoden die Tatsache, dass sich die Erscheinung von Einzelbild zu Einzel-

bild nur sehr langsam ändert. Wenn wir in der Lage sind, ein Erscheinungsmodell eines Individuums aus dem Video abzuleiten, können wir diese Daten in einem Bildstrukturmodell verwenden, um diese Person in jedem Einzelbild des Videos zu erkennen. Dann lassen sich diese Positionen über die Zeit verknüpfen, um eine Spur zu erzeugen.



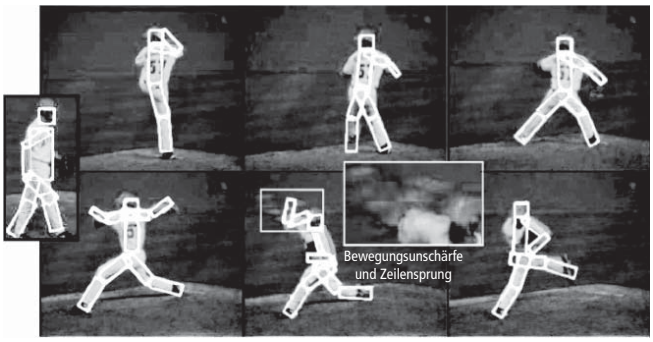


Abbildung 24.24: Sich bewegende Personen können wir mit einem Bildstrukturmodell verfolgen, indem wir zuerst ein Erscheinungsmodell erfassen und es dann anwenden. Um das Erscheinungsmodell zu erhalten, durchsuchen wir das Bild nach einer seitwärts gehenden Pose. Der Detektor braucht nicht sehr genau zu sein, sollte aber nur wenige falsche Positivergebnisse liefern. Aus der Detektorantwort können wir die Pixel ablesen, die in jedem Körpersegment liegen, und andere, die nicht in diesem Segment liegen. Dadurch lässt sich ein diskriminatives Erscheinungsmodell jedes Körperteils erstellen und diese werden dann zu einem Bildstrukturmodell der zu verfolgenden Person verknüpft. Schließlich können wir eine zuverlässige Verfolgung realisieren, indem wir dieses Modell in jedem Einzelbild erkennen. Wie die Einzelbilder im unteren Teil der Abbildung zeigen, ist diese Prozedur in der Lage, komplizierte und sich schnell ändernde Körpergestalten zu verfolgen, trotz Verlusten im Videosignal aufgrund von Bewegungsunschärfe. Abbildung von Ramanan et al. (2007) © IEEE.

Es gibt mehrere Wege, um ein gutes Erscheinungsmodell abzuleiten. Wir betrachten das Video als großen Stapel von Bildern der Person, die wir verfolgen möchten. Diesen Stapel können wir nutzen, indem wir nach Erscheinungsmodellen suchen, die viele der Bilder erklären. Das würde funktionieren, indem man Körpersegmente in jedem Einzelbild entdeckt und sich dabei auf die Tatsache stützt, dass Segmente ungefähr parallele Kanten haben. Derartige Detektoren sind nicht besonders zuverlässig, doch die gesuchten Segmente haben eine spezielle Eigenschaft: Sie erscheinen in den meisten Videoeinzelbildern mindestens einmal; derartige Segmente lassen sich durch Clustern der Detektorantworten finden. Am besten beginnt man mit dem Torso, da er groß ist und Torsodetektoren zuverlässiger arbeiten. Nachdem wir über ein Torso-Erscheinungsmodell verfügen, sollten die Segmente der Oberschenkel nahe dem Torso erscheinen usw. Daraus ergibt sich zwar ein Erscheinungsmodell, doch kann es unzuverlässig sein, wenn Personen gegen einen nahezu festen Hintergrund erscheinen, wo die Segmentdetektoren Unmengen falsche Positivergebnisse erzeugen. Alternativ lässt sich die Erscheinung für viele der Video-

einzelbilder abschätzen, indem Gestalt und Erscheinung wiederholt bewertet werden; wir sehen dann, ob *ein* Erscheinungsmodell viele Einzelbilder erklärt. Eine andere Alternative erweist sich in der Praxis als recht zuverlässig: Ein Detektor für eine feste Körpergestalt wird auf alle Einzelbilder angewendet. Eine Gestalt ist dann gut gewählt, wenn sie sich leicht zuverlässig erkennen lässt und eine große Chance besteht, dass die Person in dieser Gestalt selbst in einer kurzen Sequenz erscheint (seitliches Gehen ist eine gute Wahl). Wir optimieren den Detektor, damit er eine geringe Rate falscher Positivergebnisse liefert. Bei einer Antwort des Detektors können wir dann getrost annehmen, eine reale Person gefunden zu haben. Und da wir Torso, Arme, Beine und Kopf der Person lokalisiert haben, wissen wir, wie diese Segmente aussehen.

24.6 Computervision im Einsatz

Könnten Visionssysteme Video analysieren und verstehen, was Personen tun, wären wir in der Lage, Gebäude und öffentliche Plätze besser zu gestalten, indem wir Daten darüber sammeln und daraus schließen, was Menschen in der Öffentlichkeit tun, Überwachungssysteme zu erstellen, die genauer, sicherer und unaufdringlicher sind, Computer-Sportkommentatoren zu schaffen und Mensch-Computer-Schnittstellen zu realisieren, die Personen beobachten und auf ihr Verhalten reagieren.

Anwendungen für reaktive Schnittstellen reichen von Computerspielen, die einem Spieler die Möglichkeit verschaffen, sich umherzubewegen, bis zu Systemen, die durch Verwaltung von Wärme und Licht in einem Gebäude je nach Aufenthaltsort und Tätigkeiten der Bewohner Energie sparen.

Einige Probleme hat man gut im Griff. Wenn Personen im Videoeinzelbild relativ klein sind und der Hintergrund stabil ist, lassen sich die Personen leicht erkennen, indem man ein Hintergrundbild vom aktuellen Einzelbild subtrahiert. Ist der absolute Wert der Differenz groß, deklariert diese **Hintergrundsubtraktion** das Pixel als Vordergrundpixel. Verknüpft man die Vordergrund-Pixelbereiche über die Zeit, entsteht die gewünschte Spur.

Für strukturierte Verhaltensweisen wie Ballett, Gymnastik oder Tai-Chi gibt es spezielle Vokabulare von Aktionen. Werden solche Aktionen gegen einen einfachen Hintergrund ausgeführt, kann man Videos dieser Aktionen leicht verarbeiten. Die Hintergrundsubtraktion identifiziert die Hauptbewegungsbereiche und wir können HOG-Merkmale erstellen (den Fluss statt der Ausrichtung verfolgen), um sie an einen Klassifizierer zu übergeben. Mit einer Variante unseres Fußgängerdetektors lassen sich konsistente Aktionsmuster erkennen, wobei die Ausrichtungsmerkmale in Histogrammabschnitten über die Zeit und den Raum gesammelt werden (siehe ▶ Abbildung 24.25).

Abbildung 24.25: Einige komplexe menschliche Aktionen erzeugen konsistente Muster von Erscheinung und Bewegung. Zum Beispiel wird beim Trinken die Hand vor das Gesicht geführt. Die ersten drei Bilder sind korrekte Erkennungen für das Trinken; das vierte zeigt eine falsche Positivantwort (der Koch schaut nur in die Kaffeekanne, trinkt aber nicht daraus). Abbildung von Laptev und Perez (2007) © IEEE.

Generelle Probleme bleiben noch offen. Die große Aufgabe der Forschung besteht darin, die Beobachtungen des Körpers und der in der Nähe liegenden Objekte mit den Zielen und Absichten der sich bewegenden Personen zu verknüpfen. Schwierig ist dies unter anderem deshalb, weil wir kein einfaches Vokabular für menschliches Verhalten besitzen. Verhalten ist in vielerlei Hinsicht mit Farbe vergleichbar – viele glauben, dass sie jede Menge Verhaltensbezeichnungen kennen, sind aber nicht in der Lage, auf Knopfdruck hin lange Listen derartiger Wörter zu erzeugen. Wie die Erfahrung zeigt, werden Verhaltensweisen kombiniert – zum Beispiel kann man einen Milkshake trinken und dabei einen Geldautomaten bedienen –, doch wir wissen derzeit noch nicht, wie die Teile aussehen, wie die Zusammensetzung funktioniert oder wie viele Zusammensetzungen es geben kann. Eine zweite Quelle der Schwierigkeit ist, dass wir nicht wissen, aus welchen Merkmalen hervorgeht, was passiert. Wenn man zum Beispiel weiß, dass sich jemand einem Geldautomaten nähert, kann dies für die Erkenntnis genügen, dass er den Automaten aufsuchen wird. Eine dritte Schwierigkeit zeigt sich darin, dass die übliche Schlussfolgerung über die Beziehung zwischen Trainings- und Testdaten unzuverlässig ist. So können wir uns nicht darauf berufen, dass ein Fußgängerdetektor allein deshalb sicher ist, weil er für eine große Datenmenge gute Ergebnisse liefert. Denn in dieser Datenmenge können durchaus wichtige, wenn auch seltene, Phänomene fehlen (etwa dass eine Person auf ein Fahrrad steigt). Unser automatisierter Fahrer soll sicherlich keinen Fußgänger überfahren, nur weil er etwas Ungewöhnliches tut.

24.6.1 Wörter und Bilder

Viele Websites bieten Sammlungen von Bildern zum Betrachten an. Wie aber lassen sich die gewünschten Bilder finden? Angenommen, der Benutzer gibt eine Textanfrage wie zum Beispiel „Radrennen“ ein. Manchen Bildern sind Schlüsselwörter oder Titel zugeordnet oder sie stammen von Webseiten, die Text in der Nähe des Bildes enthalten. Hierfür kann das Abrufen von Bildern wie ein Textabrufen realisiert werden: die Bilder ignorieren und den Text des Bildes mit der Abfrage vergleichen (siehe *Abschnitt 22.3*).

Schlüsselwörter sind allerdings normalerweise unvollständig. Zum Beispiel könnte das Bild einer Katze, die auf der Straße spielt, mit Wörtern wie „Katze“ und „Straße“ markiert sein, doch man vergisst leicht, die „Mülltonne“ oder die „Fischgräten“ zu erwähnen. Es ist also eine interessante Aufgabe, ein Bild (das vielleicht schon mit einigen Schlüsselwörtern versehen ist) durch zusätzliche passende Schlüsselwörter zu kommentieren.

In der einfachsten Version dieser Aufgabe haben wir einen Satz von korrekt markierten Beispielbildern und möchten bestimmte Testbilder markieren. Dieses Problem wird auch als Auto-Annotation bezeichnet. Die genauesten Lösungen erhält man mit Methoden der Nächste-Nachbarn-Klassifikation. Man sucht die Trainingsbilder, die dem Testbild am nächsten kommen, in einer Merkmalsraummetrik, die mithilfe von Beispielen trainiert wird, und gibt deren Markierungen aus.

Eine andere Version des Problems sagt voraus, welche Markierungen welchen Bereichen in einem Testbild zuzuordnen sind. Hier wissen wir nicht, welche Bereiche welche Markierungen für die Trainingsdaten erzeugt haben. Wir können eine Version der Erwartungsmaximierung verwenden, um eine anfängliche Korrespondenz zwischen Text und Bereichen zu vermuten und von da aus eine bessere Zerlegung in Bereiche zu schätzen usw.

24.6.2 Rekonstruktion von vielen Ansichten

Binokulares Stereosehen funktioniert, weil wir für jeden Punkt vier Messungen haben, die drei unbekannte Freiheitsgrade einschränken. Die vier Messungen sind die (x, y)-Positionen des Punktes in jeder Ansicht und die unbekannten Freiheitsgrade die (x, y, z)-Koordinatenwerte des Punktes in der Szene. Diese eher grobe Feststellung legt korrekterweise nahe, dass es geometrische Einschränkungen gibt, die die meisten Punktpaare als akzeptable Treffer ausschließen. Viele Bilder einer Menge von Punkten sollten ihre Positionen eindeutig offenlegen.

Wir brauchen nicht immer ein zweites Bild, um eine zweite Ansicht einer Punktmenge zu erhalten. Wenn wir glauben, dass die ursprüngliche Punktmenge von einem bekannten starren 3D-Objekt kommt, dann steht uns gegebenenfalls ein Objektmodell als Informationsquelle zur Verfügung. Besteht dieses Objektmodell aus einer Menge von 3D-Punkten oder aus einer Menge von Bildern des Objektes und können wir Punktkorrespondenzen nachweisen, sind wir in der Lage, die Parameter der Kamera zu ermitteln, die die Punkte im Originalbild erzeugt hat. Dies sind sehr mächtige Informationen. Wir könnten sie verwenden, um unsere ursprüngliche Hypothese zu bewerten, dass die Punkte von einem Objektmodell stammen. Dazu bestimmen wir anhand einiger Punkte die Parameter der Kamera, projizieren dann Modellpunkte in diese Kamera und überprüfen, ob in der Nähe Bildpunkte liegen.

Wir haben hier eine Technologie skizziert, die heute hochentwickelt ist. Die Technologie lässt sich verallgemeinern, um Ansichten einzubeziehen, die nicht orthographisch sind, um Punkte zu verarbeiten, die nur in einigen Ansichten beobachtet werden, um unbekannte Kameraeigenschaften wie zum Beispiel die Brennweite einzubeziehen, um verschiedenartige komplizierte Suchvorgänge für passende Korrespondenzen zu nutzen und um eine Rekonstruktion aus einer sehr großen Anzahl von Punkten und von Ansichten durchzuführen. Wenn die Lage der Punkte in den Bildern mit einiger Genauigkeit bekannt ist und die Blickrichtungen sinnvoll sind, lassen sich sehr genaue Kamera- und Punktinformationen gewinnen. Zu den Anwendungen gehören unter anderem:

- **Modellerstellung:** Ein Modellierungssystem könnte eine Videosequenz, die ein Objekt zeigt, übernehmen und ein sehr detailliertes dreidimensionales Netz von texturierten Polygonen erzeugen. Diese benötigt man beispielsweise in Anwendungen der Computergrafik und der virtuellen Realität. Derartige Modelle können heute von scheinbar unbrauchbaren Bildsätzen erstellt werden. So zeigt ▶ Abbildung 24.26 ein Modell der Freiheitsstatue, das auf Bildern aus dem Internet basiert.
- **Bewegungsabgleich:** Um Computergrafikfiguren in reale Videosequenzen einzubinden, muss man wissen, wie die Kamera für das reale Video bewegt wurde, damit sich die Figur korrekt wiedergeben lässt.
- **Pfadrekonstruktion:** Mobile Roboter müssen wissen, wo sie gewesen sind. Wenn sie sich in einer Welt starrer Objekte bewegen, erhält man einen Pfad, indem man die Kamerainformationen rekonstruiert und speichert.

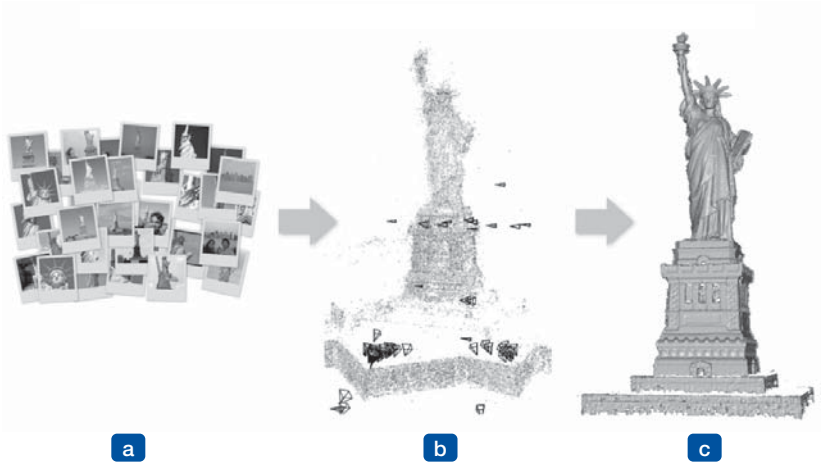


Abbildung 24.26: Die Bildrekonstruktion aus mehreren Ansichten befindet sich heute auf einem hohen technischen Niveau. Diese Abbildung veranschaulicht ein System, das von Michael Goesele und Kollegen von der Universität Washington, TU Darmstadt und Microsoft Research erstellt wurde. Aus einer Sammlung von Bildern der Freiheitsstatue, die von einer großen Benutzergemeinde im Internet veröffentlicht wurden (a), kann ihr System die Blickrichtungen für diese Bilder ermitteln, die durch kleine schwarze Pyramiden gekennzeichnet sind (b), und eine umfassende 3D-Rekonstruktion erzeugen (c).

24.6.3 Computervision für die Bewegungssteuerung

Einer der wichtigsten Verwendungszwecke der Vision ist die Bereitstellung von Informationen sowohl für die Manipulation von Objekten – Aufgreifen, Umdrehen usw. – als auch für die Navigation beim Ausweichen von Hindernissen. Die Fähigkeit, die Vision für diese Zwecke einzusetzen, ist selbst in den einfachsten visuellen Systemen von Tieren realisiert. In vielen Fällen ist das visuelle System minimal, weil es vom verfügbaren Lichtfeld nur die Informationen extrahiert, die das Tier braucht, um ihm Informationen über sein erforderliches Verhalten zu bieten. Sehr wahrscheinlich haben sich die moderneren Visionssysteme aus früheren, primitiven Organismen entwickelt, die einen fotoempfindlichen Fleck an einem Ende hatten, um sich zum Licht zu orientieren (oder davon abzuwenden). In *Abschnitt 24.4* haben wir gesehen, dass Fliegen ein sehr einfaches optisches Flusserkennungssystem verwenden, um an Wänden zu landen. Eine klassische Studie, *What the Frog's Eye Tells the Frog's Brain* (Lettvin et al., 1959), beobachtet einen Frosch: „Er verhungert, selbst wenn er von Nahrung umgeben ist, wenn sie sich nicht bewegt. Seine Nahrungsauswahl erfolgt ausschließlich nach Größe und Bewegung."

Betrachten wir ein Visionssystem für ein automatisiertes Fahrzeug, das auf einer Straße fährt. Der Fahrer hat unter anderem die folgenden Aufgaben:

1. **Seitensteuerung** – sicherstellen, dass das Fahrzeug sicher innerhalb seiner Spur bleibt oder erforderliche Spurwechsel reibungslos ausführt
2. **Längssteuerung** – sicherstellen, dass ein sicherer Abstand zum Vordermann eingehalten wird
3. **Vermeidung von Hindernissen** – Beobachtung von Fahrzeugen auf benachbarten Spuren und Vorbereitung auf Ausweichmanöver, falls eines davon entscheidet, die Spur zu wechseln

Das Problem für den Fahrer ist, geeignete Aktionen für Steuerung, Beschleunigung und Bremsen zu erzeugen, um diese Aufgabe am besten zu bewältigen.

Für die Seitensteuerungen braucht man nur eine Repräsentation der Position und Ausrichtung des Autos relativ zur Spur zu verwalten. Wir können Kantenerkennungs-Algorithmen verwenden, um Kanten zu finden, die den Spurmarkierungen entsprechen. Anschließend können wir glatte Kurven für diese Kantenelemente erzeugen. Die Parameter für diese Kurven tragen Informationen über die seitliche Position des Autos, die Richtung relativ zur Spur sowie die Krümmung der Spur. Diese Daten sind zusammen mit den Informationen über die Dynamik des Autos alles, was man braucht, um das Steuerungssystem zu realisieren. Wenn wir über detaillierte Straßenkarten verfügen, erhält das Visionssystem die Aufgabe, unsere Position zu bestätigen (und auf Hindernisse zu achten, die nicht in der Karte verzeichnet sind).

Für die Längssteuerung müssen wir den Abstand zu den vorausfahrenden Fahrzeugen ermitteln. Dies lässt sich durch binokulares Stereosehen oder durch optischen Fluss realisieren. Mithilfe dieser Techniken können visionsgesteuerte Fahrzeuge bei Geschwindigkeiten, wie sie auf Autobahnen üblich sind, zuverlässig fahren.

Ebenfalls untersucht wurde der allgemeine Fall von mobilen Robotern, die in verschiedenen Umgebungen (innerhalb von Gebäuden und im Freien) navigieren. Für das konkrete Problem, den Roboter in seiner Umgebung zu lokalisieren, gibt es inzwischen recht gute Lösungen. Ein von einer Gruppe der Firma Sarnoff entwickeltes System basiert auf zwei Kameras, die Merkmalspunkte in 3D verfolgen und anhand dieser Daten die Position des Roboters relativ zur Umgebung rekonstruieren. In der praktischen Ausführung sind die Roboter mit zwei stereoskopischen Kamerasystemen ausgerüstet – einem nach vorn und einem nach hinten schauenden System. Diese Lösung ist robuster, falls der Roboter durch ein Gebiet gehen muss, das beispielsweise aufgrund von dunklen Schatten oder von leeren Wänden merkmalsarm ist. Es ist unwahrscheinlich, dass es weder in Vorwärts- noch in Rückwärtsrichtung keine brauchbaren Merkmale gibt. Da dies aber dennoch passieren kann, ist als Sicherung ein Trägheitsmodul vorgesehen, ähnlich dem Gleichgewichtsorgan im menschlichen Innenohr, das Beschleunigungen wahrnimmt. Durch zweimalige Integration der erfassten Beschleunigung kann man die Positionsänderung verfolgen. Das Kombinieren der Daten vom visuellen System und vom Trägheitsmodul ist ein Problem der probabilistischen Evidenzfusion und lässt sich mit Techniken wie zum Beispiel durch Kalman-Filterung lösen, wie sie an anderer Stelle im Buch beschrieben wurden.

Wie bei anderen Problemen der Odometrie (Bestimmung der Positionsänderung) tritt auch bei der visuellen Odometrie das Problem einer „Drift“ auf, wobei Lageabweichungen mit der Zeit kumulativ zunehmen. Dies lässt sich mithilfe von Orientierungspunkten lösen, die absolute Festpositionen bereitstellen: Sobald der Roboter eine solche Stelle in seiner internen Karte passiert, kann er die bisher erfolgte Schätzung seiner Position entsprechend anpassen. Mit derartigen Techniken wurden Genauigkeiten in der Größenordnung von Zentimetern erreicht.

Tipp

Das Fahrbeispiel verdeutlicht einen Aspekt: *Man muss für eine bestimmte Aufgabe nicht alle Informationen erkennen, die im Prinzip aus einem Bild erkannt werden könnten*. Man braucht nicht den genauen Umriss jedes Fahrzeuges zu erkennen, die Oberflächenstruktur des Grases auf dem Grünstreifen neben der Straße nicht zu extrahieren usw. Stattdessen sollte ein Visionssystem lediglich das berechnen, was für die Aufgabe unbedingt erforderlich ist.

Bibliografische und historische Hinweise

Das Auge hat sich in der Kambrischen Explosion (vor etwa 530 Millionen Jahren) offenbar von einem gemeinsamen Vorgänger entwickelt. Seitdem haben sich unzählige Variationen in verschiedenen Lebewesen herausgebildet, doch die Entwicklung des Auges in so unterschiedlichen Tieren wie Menschen, Mäusen und Drosophilae wird immer vom gleichen Gen, Pax-6, reguliert.

Systematische Versuche, den menschlichen Gesichtssinn zu verstehen, können bis ins Altertum zurückverfolgt werden. Euklid (ca. 300 v. Chr.) schrieb über die natürliche Perspektive – die Abbildung, die jedem Punkt *P* in der dreidimensionalen Welt die Richtung des Strahls *OP* zuordnet, der den Mittelpunkt der Projektion *O* mit dem Punkt *P* verbindet. Er kannte auch das Konzept der Bewegungsparallaxe. Die Verwendung der Perspektive in der Kunst wurde bereits von der alten römischen Kultur entwickelt, wie Funde in den Ruinen von Pompeii (79 n. Chr.) belegen, was aber größtenteils für 1300 Jahre in Vergessenheit geriet. Das mathematische Verständnis der perspektivischen Projektion jetzt im Kontext der Projektion auf ebene Oberflächen hatte seinen nächsten bedeutenden Fortschritt im 15. Jahrhundert in der italienischen Renaissance zu verzeichnen. Brunelleschi (1413) wird die Erstellung der ersten Zeichnungen zugeordnet, die auf geometrisch korrekter Projektion dreidimensionaler Szenen basieren. Alberti formulierte 1435 die Regeln und inspirierte Generationen von Künstlern, deren Werke uns heute noch begeistern. Besonders bemerkenswert in ihrer Entwicklung der Wissenschaft der Perspektive, wie sie in dieser Zeit bezeichnet wurde, waren Leonardo da Vinci und Albrecht Dürer. Die Beschreibungen des Zusammenspiels von Licht und Schatten (chiaroscuro), Umbra- und Penumbrabereichen von Schatten und Bereichsperspektiven von Leonardo Ende des 15. Jahrhunderts sind in ihrer Übersetzung heute noch lesenswert (Kemp, 1989). Stork (2004) analysiert mithilfe von Techniken der Computervision, wie verschiedene Kunstwerke der Renaissance geschaffen wurden.

Obwohl auch den Griechen die Perspektive bekannt war, waren sie seltsam verwirrt über die Rolle der Augen bei der Vision. Aristoteles stellte sich Augen als Geräte vor, die Strahlen ausschicken, etwa in der Weise moderner Laserentfernungsmesser. Diese fehlerhafte Sichtweise wurde durch die Arbeiten arabischer Wissenschaftler, wie etwa Alhazen, im 10. Jahrhundert überwunden. Alhazen entwickelte auch die *camera obscura*, einen Raum (*camera* ist das lateinische Wort für „Raum“ oder „Kammer“) mit einem kleinen Loch, durch das ein Bild auf die gegenüberliegende Wand geworfen wurde. Natürlich wurde das Bild verkehrt herum gezeigt, was endlose Verwirrung stiftete. Wenn man sich das Auge als ein solches Bildgerät vorstellte, wie war es dann möglich, dass wir richtig herum sahen? Dieses Rätsel beschäftigte die großen Geister der Zeit (unter anderem Leonardo). Kepler schlug erstmals vor, dass die Linsen des Auges ein Bild auf die Retina fokussieren, und Descartes entfernte chirurgisch ein Ochsenauge und demonstrierte, dass Kepler recht hatte. Allerdings war immer noch nicht klar, warum wir nicht alles verkehrt herum sehen; heute haben wir erkannt, dass es lediglich eine Frage des richtigen Zugriffes auf die Netzhautdatenstruktur ist.

In der ersten Hälfte des 20. Jahrhunderts wurden die bedeutendsten Forschungsergebnisse in der Vision durch die Schule der Gestaltpsychologie unter der Leitung von Max Wertheimer erzielt. Ihre Vertreter hoben die Wichtigkeit der wahrnehmenden Organisation hervor: Für einen menschlichen Beobachter ist das Bild keine Sammlung von pointilli?stischen Fotorezeptorausgaben (Pixel in der Sprache der Computervision); vielmehr ist es in zusammenhängenden Gruppen organisiert. Auf diese Einsicht könnte man die Motivation in der Computervision, Bereiche und Kurven zu finden, zurückführen. Die Anhänger der Gestalttheorie verwiesen auch auf das „Figur-Grund"-Phänomen – eine Kontur, die zwei Bildbereiche trennt, die in der Realität in verschiedenen Tiefen liegen, scheint nur zum näheren Bereich (der „Figur") und nicht zum weiter entfernten Bereich (dem „Grund") zu gehören. Das Problem der Computervision, Bildkurven entsprechend ihrer Bedeutung in der Szene zu klassifizieren, lässt sich als Verallgemeinerung dieser Erkenntnis auffassen.

In der Zeit nach dem Zweiten Weltkrieg ergaben sich neue Aktivitäten. Die wichtigste davon war die Arbeit von J. J. Gibson (1950, 1979), der die Bedeutung des optischen Flusses sowie der Texturgradienten in der Abschätzung der Umgebungsvariablen wie etwa Oberflächenneigung und Schräge darlegte. Er wies erneut auf die Bedeutung des Reizes und seine Reichhaltigkeit hin. Gibson betonte die Rolle des aktiven Beobachters, dessen selbst gesteuerte Bewegung die Aufnahme von Informationen über die äußere Umgebung erleichtert.

Die Grundlagen zur Computervision wurden in den 1960er Jahren gelegt. Die Doktorarbeit von Roberts (1963) am MIT war eine der ersten Veröffentlichungen auf diesem Gebiet und führte Schlüsselideen wie zum Beispiel Kantenerkennung und modellbasierten Vergleich ein. Es gibt eine moderne Legende, dass Marvin Minsky das Problem „Lösen" der Computervision einem Doktoranden als Sommerprojekt zugewiesen hat. Nach Minsky ist die Legende falsch – es war tatsächlich ein Diplomand. Aber es war ein außergewöhnlicher Diplomand, Gerald Jay Sussman (der jetzt Professor am MIT ist). Und die Aufgabe bestand nicht darin, die Computervision zu „lösen", sondern einige ihrer Aspekte zu untersuchen.

In den 1960er und 1970er Jahren ging es nur langsam voran, was vor allem an fehlenden Rechen- und Speicherressourcen lag. Der visuellen Low-Level-Verarbeitung wurde große Aufmerksamkeit geschenkt. Der weit verbreitete Canny-Algorithmus zur Kantenerkennung wurde in Canny (1986) eingeführt. Techniken, die auf Filterung mit mehreren Skalierungen und Ausrichtungen von Bildern basieren, um Texturgrenzen zu finden, gehen auf Arbeiten wie zum Beispiel Malik und Perona (1990) zurück. Die Kombination mehrerer Hinweise – Helligkeit, Textur und Farbe – zum Ermitteln von Grenzkurven in einem lernenden Framework wurden von Martin, Fowlkes und Malik (2004) vorgestellt, um die Leistung erheblich zu verbessern.

Das eng verwandte Problem, Bereiche von kohärenter Helligkeit, Farbe und Textur zu finden, führt praktisch von selbst zu Formulierungen, in denen die Ermittlung der besten Partition zu einem Optimierungsproblem wird. Drei maßgebliche Beispiele sind der Markov-Random-Fields-Ansatz von Geman und Geman (1984), die Variationsformulierung von Mumford und Shah (1989) sowie die normalisierten Schnitte von Shi und Malik (2000).

Die 1960er, 1970er und 1980er Jahre sind durch zwei verschiedene Paradigmen der visuellen Erkennung geprägt, die sich aus unterschiedlichen Perspektiven ergeben, was als hauptsächliches Problem wahrgenommen wurde. Die Forschung auf dem Gebiet der Computervision zur Objekterkennung konzentrierte sich vor allem auf Probleme, die sich aus der Projektion von dreidimensionalen Objekten auf zweidimensionale Bilder ergeben. Das Konzept der Ausrichtung, das erstmals von Roberts eingeführt wurde, tauchte erneut in den 1980ern in der Arbeit von Lowe (1987) sowie Huttenlocher und Ullman (1990) auf. Von besonderem Interesse war auch ein Ansatz von Tom Binford (1971), Umrisse in Form von Volumengrundelementen – und zwar mit **verallgemeinerten Zylindern** – zu beschreiben.

Im Unterschied dazu betrachteten die Vertreter der Mustererkennung die 3D-zu-2D-Aspekte des Problems als nicht signifikant. Ihre motivierenden Beispiele stammen aus Domänen wie beispielsweise der optischen Zeichenerkennung oder der Erkennung von handschriftlichen Postleitzahlen, wo es in erster Linie darum geht, die typischen Variationen zu lernen, die charakteristisch für eine Klasse von Objekten sind, und sie von anderen Klassen abzutrennen. Einen Vergleich der Ansätze finden Sie bei LeCun et al. (1995).

In den späten 1990ern näherten sich die Paradigmen an, als beide Seiten die probabilistischen Modellierungs- und Lerntechniken übernahmen, die sich überall in der KI etablierten. Zwei Arbeitssphären trugen beträchtlich dazu bei. Die eine war die Gesichtserkennung wie zum Beispiel die von Rowley, Baluja und Kanade (1996) sowie von Viola und Jones (2002b), die die Leistung der Mustererkennungstechniken für zweifellos wichtige und nützliche Aufgaben demonstrierten. Die andere betrifft die Entwicklung von Punktdeskriptoren, mit denen sich Merkmalsvektoren aus Objektteilen konstruieren lassen. Die Pionierarbeit hierzu stammt von Schmid und Mohr (1996). Weit verbreitet ist der SIFT-Deskriptor von Lowe (2004). Der HOG-Deskriptor geht auf Dalal und Triggs (2005) zurück.

Ullman (1979) und Longuet-Higgins (1981) sind einflussreiche frühe Arbeiten in der Rekonstruktion von mehreren Bildern. Bedenken hinsichtlich der Stabilität von Struktur aus Bewegung wurden durch die Arbeiten von Tomasi und Kanade (1992) zerstreut. Sie zeigten, dass sich die Umrisse mithilfe mehrerer Einzelbilder recht genau rekonstruieren lassen. In den 1990er Jahren erschlossen sich der Bewegungsanalyse durch wesentlich höhere Computergeschwindigkeiten und Speicherkapazitäten viele neue Anwendungsgebiete. Insbesondere erfreute sich der Aufbau geometrischer Modelle von Szenen aus der realen Welt für die Darstellung durch Techniken der Computergrafik größter Beliebtheit und führte zu Rekonstruktions-Algorithmen, wie beispielsweise dem von Debevec, Taylor und Malik (1996) entwickelten. Die Bücher von Hartley und Zisserman (2000) sowie Faugeras et al. (2001) bieten eine umfassende Beschreibung der Geometrie von Mehrfachansichten.

Für Einzelbilder wurde die Ableitung des Umrisses aus der Schattierung zuerst von Horn (1970) untersucht. Horn und Brooks (1989) legen einen umfassenden Überblick über die wichtigsten Veröffentlichungen aus einer Periode vor, als dies ein vielfach untersuchtes Problem war. Gibson (1950) schlug erstmals Texturgradienten als Hinweis auf den Umriss vor, auch wenn eine umfassende Analyse für gekrümmte Oberflächen erst in Garding (1992) sowie Malik und Rosenholtz (1997) erscheint. Die mathematischen Grundlagen für verdeckende Konturen und allgemein das Verständnis visueller Ereignisse in der Projektion von glatten gekrümmten Objekten sind zu einem großen Teil der Arbeit von Koenderink und van Doorn zu verdanken, die in *Solid Shape* von Koenderink (1990) ausführlich behandelt wird. In den letzten Jahren gibt es ein verstärktes Interesse, das Problem der Umriss- und Oberflächenrekonstruktion aus einem einzelnen Bild als probabilistisches Inferenzproblem zu behandeln, wo geometrische Hinweise nicht explizit modelliert, sondern implizit in einem lernenden Framework verwendet werden. Ein guter Vertreter ist die Arbeit von Hoiem, Efros und Hebert (2008).

Für Leser, die am menschlichen Gesichtssinn interessiert sind, bietet Palmer (1999) die beste Gesamtdarstellung; Bruce et al. (2003) ist ein kürzer gefasstes Lehrbuch. Die Bücher von Hubel (1988) und Rock (1984) sind verständliche Einführungen mit den Schwerpunkten Neurophysiologie bzw. Wahrnehmung. Das Buch *Vision* von David Marr (1982) spielte eine wichtige Rolle bei der Verknüpfung von Computervision mit Psychophysik und Neurobiologie. Auch wenn sich viele seiner spezifischen Modelle nicht bewährt haben, ist die theoretische Perspektive, von der aus jede Aufgabe hinsichtlich Information, Rechentechnik und Implementierung analysiert wird, immer noch recht erhellend.

Für den Bereich der Computervision kommt das umfassendste Lehrbuch von Forsyth und Ponce (2002). Trucco und Verri (1998) ist eine kürzere Schilderung. Horn (1986) und Faugeras (1993) sind zwei ältere, aber immer noch nützliche Lehrbücher.

Die wichtigsten Fachzeitschriften zur Computervision sind *IEEE Transactions on Pattern Analysis and Machine Intelligence* und das *International Journal of Computer Vision*. Konferenzen zur Computervision sind unter anderem ICCV (International Conference on Computer Vision), CVPR (Computer Vision and Pattern Recognition) und ECCV (European Conference on Computer Vision). Ergebnisse zur Forschung mit einer Komponente zum maschinellen Lernen werden auch in der Konferenz NIPS (Neural Information Processing Systems) veröffentlicht und Arbeiten zur Schnittstelle mit Computergrafik erscheinen oft auf der Konferenz ACM SIGGRAPH (Special Interest Group in Graphics).

Zusammenfassung

Obwohl die Wahrnehmung für den Menschen eine mühelose Aktivität zu sein scheint, ist ein großer Aufwand komplexer Berechnungen dafür erforderlich. Ziel der Vision ist es, Informationen zu extrahieren, die für Aufgaben wie beispielsweise Manipulation, Navigation und Objekterkennung erforderlich sind.

- Der Prozess des **Bildaufbaus** ist in seinen geometrischen und physischen Aspekten weitgehend erforscht. Für eine Beschreibung einer dreidimensionalen Szene können wir einfach ein Bild aus einer beliebigen Kameraposition erzeugen (das Grafikproblem). Die Umkehrung des Prozesses von einem Bild hin zu einer Beschreibung der Szene ist schwieriger.
- Um die visuelle Information zu extrahieren, die man für die Aufgaben der Manipulation, Navigation und Erkennung benötigt, müssen Zwischendarstellungen erzeugt werden. Frühe Algorithmen für die **Bildverarbeitung** extrahieren primitive Merkmale aus dem Bild, wie beispielsweise Kanten und Bereiche.
- Es gibt mehrere Hinweise im Bild, die es ermöglichen, dreidimensionale Informationen über die Szene zu erhalten: Bewegung, stereoskopisches Sehen (Stereopsis), Oberflächenstruktur (Textur), Schattierung und Konturenanalyse. Jeder dieser Hinweise basiert auf Hintergrundannahmen über physische Szenen, um nahezu eindeutige Interpretationen bereitzustellen.
- Die Objekterkennung ist in ihrer vollständigen Allgemeinheit ein sehr schwieriges Problem. Wir haben die helligkeitsbasierten und merkmalbasierten Ansätze beschrieben. Außerdem haben wir einen einfachen Algorithmus für die Posenabschätzung vorgestellt. Es gibt aber auch andere Möglichkeiten.

Übungen zu Kapitel 24

Lösungs-
hinweise

1 Im Schatten eines Baumes mit dichter Blätterkrone sieht man viele Lichtflecke. Überraschenderweise scheinen sie alle kreisförmig zu sein. Warum? Schließlich sind die Lücken zwischen den Blättern, durch die die Sonnenstrahlen dringen, nicht kreisförmig.

2 Betrachten Sie einen unendlich langen Zylinder mit dem Radius r, dessen Achse entlang der y-Achse ausgerichtet ist. Der Zylinder hat eine Lambertsche Oberfläche und wird von einer Kamera entlang der positiven z-Achse betrachtet. Was erwarten Sie auf dem Bild zu sehen, wenn der Zylinder durch eine Punktlichtquelle beleuchtet ist, die sich in der Unendlichkeit an der positiven x-Achse befindet? Zeichnen Sie die Konturen konstanter Helligkeit in das projizierte Bild ein. Sind die Konturen gleicher Helligkeit einheitlich verteilt?

3 Die Kanten in einem Bild können verschiedensten Ereignissen in einer Szene entsprechen. Betrachten Sie Abbildung 24.4 (*Abschnitt 24.1.4*) und nehmen Sie an, dass es sich um ein Bild einer realen dreidimensionalen Szene handelt. Kennzeichnen Sie zehn verschiedene Helligkeitskanten im Bild und geben Sie für jede an, ob sie einer Diskontinuität in (a) Tiefe, (b) Oberflächenausrichtung, (c) Reflexion oder (d) Beleuchtung entsprechen.

4 Ein stereoskopisches System wird für eine Geländeabbildung verwendet. Es besteht aus zwei CCD-Kameras, die jeweils 512×512 Pixel auf einem 10×10 cm großen Sensor aufnehmen. Die verwendeten Linsen haben eine Brennweite von 16 cm, wobei der Fokus fest auf unendlich eingestellt ist. Für entsprechende Punkte (u_1, v_1) im linken Bild und (u_2, v_2) im rechten Bild ist $v_1 = v_2$, weil die x-Achsen in den beiden Bildebenen parallel zu den Epipolarlinien verlaufen. Die optischen Achsen der beiden Kameras sind parallel. Die Länge der Grundlinie zwischen den Kameras beträgt einen Meter.

a. Wenn der nächstliegende Bereich, der gemessen werden soll, 16 m beträgt, wie hoch ist dann die größte Ungleichheit, die auftritt (in Pixeln)?

b. Welchen Auflösungsbereich haben wir bei 16 m aufgrund der Pixelanordnung?

c. Welcher Bereich entspricht einer Verschiedenheit von 1 Pixel?

5 Welche der folgenden Aussagen sind wahr, welche falsch?

a. Das Suchen von Entsprechungspunkten in Stereobildern ist die einfachste Phase des Stereo-Tiefenfindungsprozesses.

b. In Stereoansichten derselben Szene erhält man eine größere Genauigkeit in den Tiefenberechnungen, wenn die beiden Kamerapositionen weiter voneinander entfernt sind.

c. Linien mit gleichen Längen in der Szene werden immer auf gleiche Längen im Bild projiziert.

d. Gerade Linien im Bild entsprechen zwangsläufig geraden Linien in der Szene.

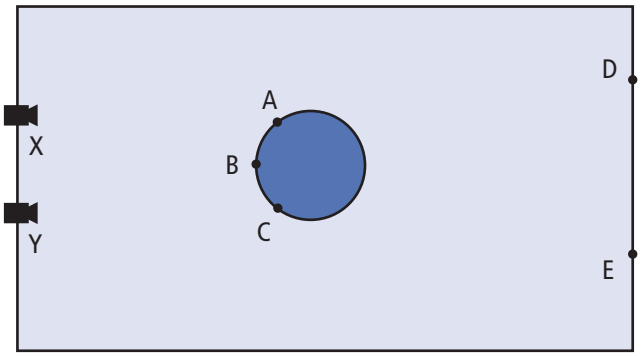


Abbildung 24.27: Draufsicht eines Visionssystems mit zwei Kameras, das eine Flasche mit einer dahinterliegenden Mauer beobachtet.

6 (Mit freundlicher Genehmigung von Pietro Perona.) ▶ Abbildung 24.27 zeigt zwei Kameras an den Stellen X und Y, die eine Szene beobachten. Zeichnen Sie das Bild, das in jeder Kamera gesehen wird, und gehen Sie davon aus, dass sich alle angegebenen Punkte in derselben horizontalen Ebene befinden. Was kann aus diesen beiden Bildern über die relativen Entfernungen der Punkte A, B, C, D und E von der Kameragrundlinie geschlossen werden und auf welcher Grundlage?

Robotik

25

ÜBERBLICK

In diesem Kapitel werden die Agenten mit physischen Effektoren ausgestattet, mit denen sie allerhand Unfug anstellen können.

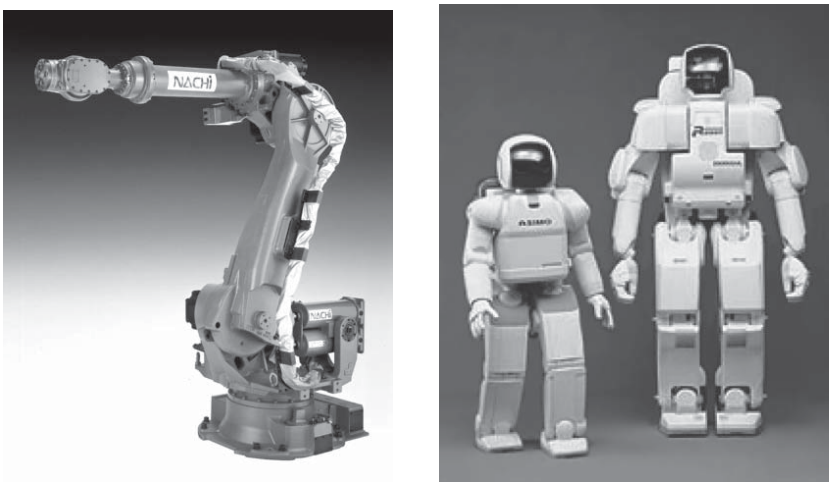
25.1 Einführung

Roboter sind physische Agenten, die Aufgaben ausführen, indem sie die physische Welt manipulieren. Dazu sind sie mit **Effektoren** ausgestattet, wie beispielsweise Beinen, Rädern, Gelenken oder Greifarmen. Effektoren haben einen einzigen Zweck: Sie sollen die physische Einwirkung auf die Umgebung sicherstellen.[1] Darüber hinaus sind Roboter mit **Sensoren** ausgestattet, die es ihnen erlauben, ihre Umgebung wahrzunehmen. Die heutige Robotik verwendet die unterschiedlichsten Sensoren, wie beispielsweise Kameras und Laser, um die Umgebung zu messen, sowie Gyroskope und Beschleunigungsmesser, um die Eigenbewegung des Roboters zu messen.

Die meisten der heutigen Roboter können in eine von drei Hauptkategorien eingeteilt werden. **Manipulatoren**, oder Roboterarme wie in ▶ Abbildung 25.1(a) gezeigt, sind physisch an ihrem Arbeitsplatz verankert, wie beispielsweise an einem Fließband in einer Fabrik oder in der internationalen Raumstation. Die Bewegung von Manipulatoren beinhaltet in der Regel eine ganze Kette von steuerbaren Gelenken, so dass solche Roboter ihre Effektoren an beliebige Positionen innerhalb ihres Arbeitsplatzes bewegen können. Manipulatoren sind die gebräuchlichsten Industrieroboter und es sind weltweit ungefähr 1 Million Einheiten installiert. Einige mobile Manipulatoren werden in Krankenhäusern verwendet, um die Ärzte zu unterstützen. Kaum ein Automobilhersteller könnte überleben, gäbe es nicht die Roboter-Manipulatoren, und einige Manipulatoren wurden sogar verwendet, um Kunst zu erschaffen.

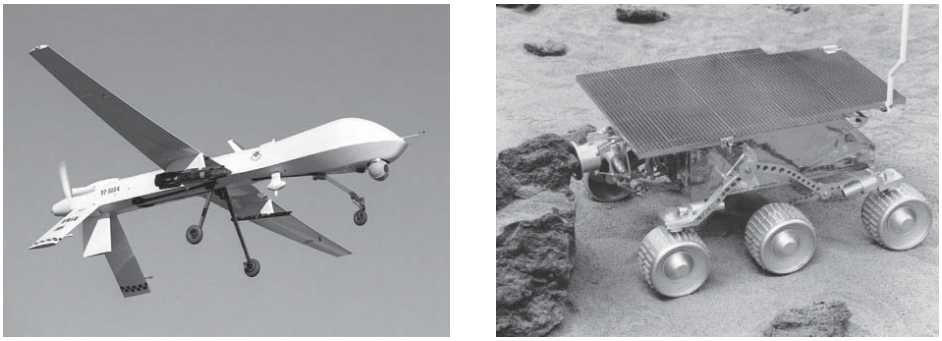
Die zweite Kategorie ist der **mobile Roboter**. Mobile Roboter bewegen sich in ihrer Umgebung mithilfe von Rädern, Beinen oder ähnlichen Mechanismen. Sie werden verwendet, um in Krankenhäusern Essen auszuteilen, Container auf Ladebrücken zu verschieben oder ähnliche Aufgaben zu erfüllen. **Unbemannte Landfahrzeuge** (**UGV**, Unmanned Ground Vehicle) fahren eigenständig auf Straßen und im Gelände. Der in ▶ Abbildung 25.2(b) gezeigte **Planeten-Rover** hat 1997 den Mars über einen Zeitraum von 3 Monaten erkundet. Die von der NASA-Mission **Mars Exploration Rover** 2003 entsandten Zwillingsroboter waren sechs Jahre später noch funktionsfähig. Andere Arten mobiler Roboter sind beispielsweise **unbemannte Luftfahrzeuge** (**UAV**, Unmanned Air Vehicles), die häufig in der Überwachung, bei der Verteilung von Pflanzenschutzmitteln oder in militärischen Operationen eingesetzt werden. ▶ Abbildung 25.2(a) zeigt ein derartiges Luftfahrzeug, das häufig vom US-Militär eingesetzt wird. **Autonome Unterwasserfahrzeuge** (**AUV**, Autonomous Underwater Vehicles) werden bei der Erkundung der Tiefsee eingesetzt. Mobile Roboter stellen Pakete am Arbeitsplatz zu und saugen im Wohnbereich die Fußböden.

1 In *Kapitel 2* haben wir über **Aktuatoren**, nicht über Effektoren gesprochen. Hier unterscheiden wir den Effektor (das physische Gerät) vom Aktuator (die Steuerleitung, die einen Befehl an einen Effektor übermittelt).



a b

Abbildung 25.1: (a) Ein Industrieroboter als Manipulator für das Stapeln von Säcken auf Paletten. Bild mit freundlicher Genehmigung von Nachi Robotic Systems. (b) Die humanoiden Roboter P3 und Asimo von Honda.



a b

Abbildung 25.2: (a) Das vom US-Militär eingesetzte UAV Predator. Bild mit freundlicher Genehmigung von General Atomics Aeronautical Systems. (b) Der Rover Sojourner der NASA, ein mobiler Roboter, der im Juli 1997 die Marsoberfläche erkundet hat.

Der häufig als **mobiler Manipulator** bezeichnete dritte Robotertyp kombiniert Mobilität mit Manipulation. **Humanoide Roboter** sind dem menschlichen Körper nachempfunden. ▶ Abbildung 25.1(b) zeigt zwei frühe humanoide Roboter, die von der Firma Honda in Japan hergestellt wurden. Mobile Manipulatoren können ihre Effektoren in einem weiter reichenden Umfeld einsetzen als verankerte Manipulatoren, doch ist ihre Aufgabe schwieriger, weil sie nicht die Stabilität besitzen, die sonst der Anker bietet.

Der Bereich der Robotik beinhaltet auch prothetische Geräte (künstliche Hüften, Ohren und Augen für Menschen), intelligente Umgebungen (wie beispielsweise ein Haus, das vollständig mit Sensoren und Effektoren ausgestattet ist) sowie Multikörpersysteme, wobei die Roboteraktion durch Schwärme von kleinen zusammenarbeitenden Robotern realisiert wird.

Echte Roboter müssen häufig mit Umgebungen zurechtkommen, die partiell beobachtbar, stochastisch, dynamisch und stetig sind. Viele Roboterumgebungen sind sequentiell und multiagentenorientiert. Partielle Beobachtbarkeit und Stochastik entstehen, wenn man es mit einer großen, komplexen Welt zu tun hat. Roboterkameras können nicht um die Ecken sehen und Bewegungsbefehle unterliegen Unsicherheiten durch Getriebespiel, Reibung usw. Darüber hinaus weigert sich die reale Welt hartnäckig, schneller als in Echtzeit zu arbeiten. In einer simulierten Umgebung ist es möglich, einfache Algorithmen zu verwenden (wie beispielsweise den Algorithmus für das **Q-Lernen**, den wir in *Kapitel 21* beschrieben haben), um in ein paar wenigen CPU-Stunden aus Millionen von Versuchen zu lernen. In einer realen Umgebung könnte es Jahre dauern, diese Versuche durchzuführen. Darüber hinaus können echte Unfälle wirklichen Schaden anrichten – im Gegensatz zu den simulierten Unfällen. Praktische Robotersysteme müssen Vorwissen über den Roboter, seine physische Umgebung sowie die Aufgaben, die er erledigen soll, aufnehmen, sodass der Roboter schnell lernen und sicher vorgehen kann.

Die Robotik vereint viele Konzepte, die wir weiter vorn im Buch kennengelernt haben, unter anderem probabilistische Zustandsschätzung, Wahrnehmung, Planung, nicht überwachtes Lernen und verstärkendes Lernen (Reinforcement Learning). Für einige dieser Konzepte dient die Robotik als anspruchsvolle Beispielanwendung. Für andere Konzepte leistet dieses Kapitel Pionierarbeit und führt die kontinuierliche Version von Techniken ein, die wir bisher nur im diskreten Fall gesehen haben.

25.2 Roboter-Hardware

Bisher haben wir in diesem Buch die Architektur der Agenten – Sensoren, Effektoren und Prozessoren – als gegeben vorausgesetzt und uns auf das Agentenprogramm konzentriert. Der Erfolg von realen Robotern ist jedoch mindestens genauso stark von dem Entwurf der Sensoren und Effektoren abhängig, die für die jeweilige Aufgabe geeignet sind.

25.2.1 Sensoren

Sensoren sind die Wahrnehmungsschnittstelle zwischen Robotern und ihren Umgebungen. **Passive Sensoren**, wie beispielsweise Kameras, sind echte Beobachter der Umgebung: Sie nehmen Signale auf, die von anderen Quellen in der Umgebung erzeugt werden. **Aktive Sensoren**, wie beispielsweise Sonar, geben Energie in die Umgebung ab. Sie arbeiten mit der Tatsache, dass diese Energie zurück zum Sensor reflektiert wird. Aktive Sensoren stellen normalerweise mehr Informationen bereit als passive Sensoren, allerdings auf Kosten eines höheren Leistungsverbrauchs und der Gefahr von Interferenzen, wenn mehrere aktive Sensoren gleichzeitig verwendet werden. Unabhängig davon, ob Sensoren aktiv oder passiv sind, können sie in drei Typen unterteilt werden, je nachdem, ob sie die Umgebung, die Position des Roboters oder dessen innere Konfiguration erfassen.

Entfernungsmesser sind Sensoren, die den Abstand zu nahe gelegenen Objekten messen. In den Anfängen der Robotik waren Roboter häufig mit **Sonarsensoren** ausgerüstet. Sonarsensoren geben gerichtete Schallwellen ab, die von Objekten reflektiert werden und zurück im Sensor einen bestimmten Ton erzeugen. Aus der Zeit und der Intensität des zurückgegebenen Signals lässt sich die Distanz zu nahegelegenen Objekten ermitteln. Sonar ist die Technologie der Wahl für autonome Unterwasserfahrzeuge. **Stereovision** (siehe *Abschnitt 24.4.2*) stützt sich auf mehrere Kameras, um ein Abbild der

Umgebung aus leicht unterschiedlichen Blickwinkeln zu erfassen. Die resultierende Parallaxe in diesen Bildern wird analysiert, um den Bereich der umgebenden Objekte zu berechnen. Für mobile Bodenroboter verwendet man Sonar und Stereovision heute nur noch selten, weil sie nicht zuverlässig genau sind.

Die meisten Bodenroboter sind heute mit optischen Entfernungsmessern ausgerüstet. Ähnlich wie Sonarsensoren senden optische Entfernungsmesser aktive Signale (Licht) aus und messen die Zeit, bis das reflektierte Signal wieder am Sensor eintrifft. ▶ Abbildung 25.3(a) zeigt eine **TOF-Kamera**.[2] Diese Kamera erfasst Nahfeldbilder wie in ▶ Abbildung 25.3(b) gezeigt mit bis zu 60 Einzelbildern pro Sekunde. Andere Entfernungssensoren arbeiten mit Laserstrahlen und speziellen 1-Pixel-Kameras, die sich mithilfe komplexer Spiegelanordnungen oder rotierender Elemente lenken lassen – sogenannte **scannende Lidar-Systeme**.[3] Scannende Lidar-Systeme eignen sich für größere Entfernungen als TOF-Kameras und schneiden auch bei hellem Tageslicht besser ab.

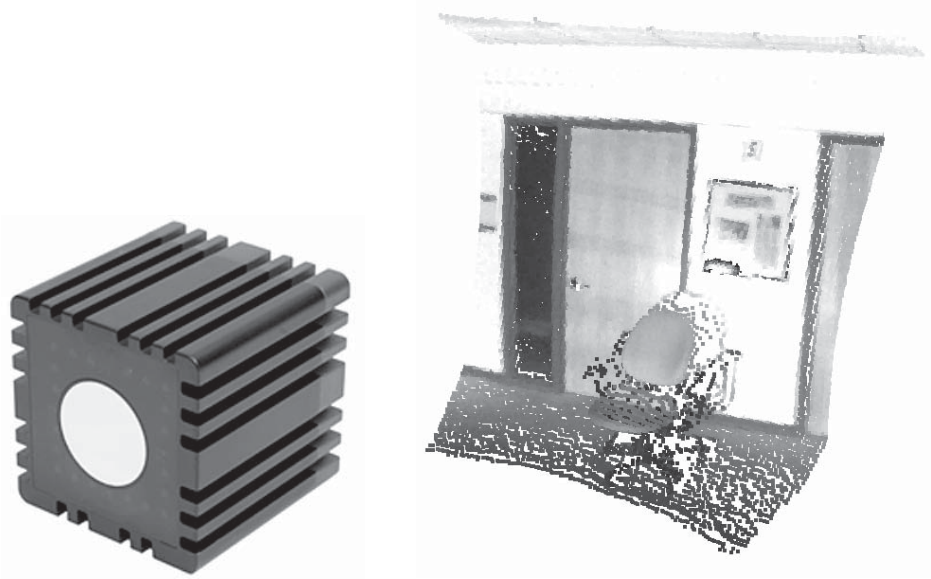


Abbildung 25.3: (a) TOF-Kamera; Bild mit freundlicher Genehmigung von Mesa Imaging GmbH. (b) 3D-Entfernungsbild, das mit dieser Kamera aufgenommen wurde. Das Entfernungsbild ermöglicht es, Hindernisse und Objekte in der Nähe eines Roboters zu erkennen.

Zu den Entfernungssensoren gehört auch das Radar, das oftmals das Sensorsystem der Wahl für UAVs darstellt. Radarsensoren können Entfernungen von mehreren Kilometern messen. Für äußerst geringe Entfernungen gibt es **Tastsensoren** wie zum Beispiel Tasthaare, Druckaufnehmer und berührungsempfindliche Haut. Diese Sensoren messen Entfernungen basierend auf dem physischen Kontakt und eignen sich nur für Objekte, die sich sehr nahe beim Roboter befinden.

2 TOF = Tife Of Flight, d.h. eine Kamera zur Messung der Lichtlaufzeit.
3 Lidar = Light detection and ranging, d.h. ein Sensor zur Abstands- und Geschwindigkeitsmessung mithilfe von Licht.

Eine zweite wichtige Sensorklasse sind die **Standortsensoren**. Die meisten Standortsensoren nutzen eine Entfernungsmessung als Hauptkomponente, um die Position zu ermitteln. Im Freien ist das **GPS** (Global Positioning System) die gebräuchlichste Lösung für die Standortbestimmung. GPS misst die Entfernung zu Satelliten, die kodierte Funksignale ausstrahlen. Momentan befinden sich bis zu 31 Satelliten in der Umlaufbahn, die jeweils Signale auf mehreren Frequenzen senden. GPS-Empfänger können die Distanz zu diesen Satelliten ermitteln, indem sie die Phasenverschiebungen analysieren. Durch Triangulation von Signalen von mehreren Satelliten sind GPS-Empfänger in der Lage, ihre absolute Position auf der Erde mit einer Genauigkeit von wenigen Metern festzustellen. Das **Differential-GPS** benutzt einen zweiten Bodenempfänger mit bekannter Position, der unter idealen Bedingungen eine Lokalisierung im Millimeterbereich erlaubt. Leider funktioniert GPS nicht in geschlossenen Räumen oder unter Wasser. Im Innenbereich wird die Ortsbestimmung oftmals mithilfe von Baken in der Umgebung bei bekannten Orten realisiert. Da viele Umgebungen im Innenbereich voller Basisstationen für die Drahtlosübertragung sind, lassen sich Roboter auch über die Analyse des Drahtlossignals lokalisieren. Unter Wasser können aktive Sonarbaken eine Standortbestimmung ermöglichen, wobei AUVs per Schallwellen über ihre relativen Abstände zu diesen Baken informiert werden.

Die dritte wichtige Klasse sind die **proprioceptiven (selbstwahrnehmenden) Sensoren**, die den Roboter über seine eigene Bewegung informieren. Um die genaue Konfiguration eines Robotergelenks zu messen, sind die Motoren häufig mit **Drehimpulsgebern** ausgestattet, die die Umdrehung des Motors in kleinen Inkrementen zählen. Auf Roboterarmen können Drehimpulsgeber über beliebige Zeiträume genaue Informationen bereitstellen. Auf mobilen Robotern können Drehimpulsgeber, die Radumdrehungen melden, für die **Odometrie** verwendet werden – die Messung der zurückgelegten Distanz. Da aber Räder durchdrehen oder abweichen können, ist die Odometrie nur über kurze Distanzen zuverlässig. Externe Kräfte, wie beispielsweise die Strömung für AUVs oder der Wind für UAVs, erhöhen die Unsicherheit im Hinblick auf die Position. **Trägheitssensoren**, wie etwa Gyroskope, nutzen den Widerstand der Masse gegen Geschwindigkeitsänderungen. Damit lässt sich die Unsicherheit zumindest reduzieren.

Weitere wichtige Aspekte des Roboterzustandes werden durch **Kraft-** und **Drehmomentsensoren** gemessen. Sie sind unverzichtbar, wenn der Roboter mit zerbrechlichen Objekten umgehen soll oder mit Objekten, deren genaue Größe und Position unbekannt sind. Stellen Sie sich einen eine Tonne schweren Roboter-Manipulator vor, der eine Glühbirne einschraubt. Wenn er nur ein bisschen zu viel Kraft anwendet, zerbricht die Glühbirne. Kraftsensoren erlauben es dem Roboter zu spüren, wie fest er zugreifen darf, und Drehmomentsensoren ermöglichen das Spüren, wie stark er dreht. Gute Sensoren können Kräfte in allen drei Fahrt- und Drehrichtungen messen, und zwar mehrere hundert Mal pro Sekunde, sodass ein Roboter unerwartete Kräfte schnell erkennen und seine Aktionen korrigieren kann, bevor er eine Glühbirne zerbricht.

25.2.2 Effektoren

Effektoren sind die Hilfsmittel, über die die Roboter sich bewegen und die Form ihrer Körper verändern. Um den Entwurf von Effektoren zu verstehen, wollen wir zuerst abstrakt über Bewegung und Umriss sprechen. Dazu verwenden wir das Konzept eines **Freiheitsgrades**. Wir zählen einen Freiheitsgrad für jede unabhängige Richtung, in der sich ein Roboter oder einer seiner Effektoren bewegen kann. So hat ein starrer mobiler Roboter wie beispielsweise ein AUV sechs Freiheitsgrade, drei für seine (x, y, z)-Posi-

tion im Raum und drei für seine Winkelausrichtung, auch als *Gier-*, *Roll-* und *Nickachse* bezeichnet. Diese sechs Freiheitsgrade definieren den **kinematischen Zustand**[4] oder die **Pose** des Roboters. Neben diesen sechs Freiheitsgraden beinhaltet der **dynamische Zustand** eines Roboters noch eine zusätzliche Dimension für die Änderungsgeschwindigkeit jeder kinematischen Dimension.

Für nicht starre Körper gibt es zusätzliche Freiheitsgrade innerhalb des eigentlichen Roboters. Zum Beispiel hat der Ellenbogen eines menschlichen Armes zwei Freiheitsgrade – er kann den Unterarm gegenüber dem Oberarm beugen und strecken sowie nach rechts oder links drehen. Das Handgelenk hat drei Freiheitsgrade. Es kann sich nach oben und unten sowie zur Seite bewegen und es kann sich drehen. Robotergelenke haben ebenfalls einen, zwei oder drei Freiheitsgrade. Man braucht sechs Freiheitsgrade, um ein Objekt wie beispielsweise eine Hand an einem bestimmten Punkt in einer bestimmten Richtung zu platzieren. Der in ▶ Abbildung 25.4(a) gezeigte Arm hat genau sechs Freiheitsgrade, die durch fünf **Drehgelenke** erzeugt werden, die eine Drehbewegung hervorrufen, sowie durch ein **prismatisches Gelenk**, das eine Gleitbewegung hervorruft. Sie können sich davon überzeugen, dass der menschliche Arm als Ganzes mehr als sechs Freiheitsgrade besitzt, indem Sie ein einfaches Experiment ausführen: Legen Sie Ihre Hand auf den Tisch und beobachten Sie, dass Sie immer noch die Freiheit haben, Ihren Ellenbogen zu drehen, ohne die Konstellation Ihrer Hand zu ändern. Manipulatoren mit zusätzlichen Freiheitsgraden sind leichter zu steuern als Roboter, die nur die Mindestanzahl an Freiheitsgraden besitzen. Viele industrielle Manipulatoren verfügen deshalb über sieben und nicht nur sechs Freiheitsgrade.

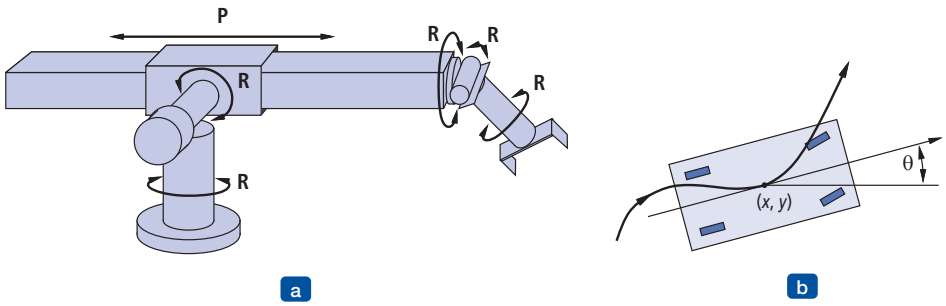


Abbildung 25.4: (a) Der Stanford-Manipulator, ein früher Roboterarm mit fünf Drehgelenken (R) und einem prismatischen Gelenk (P) für insgesamt sechs Freiheitsgrade. (b) Bewegung eines nichtholonomen vierrädrigen Fahrzeuges mit Vorderradlenkung.

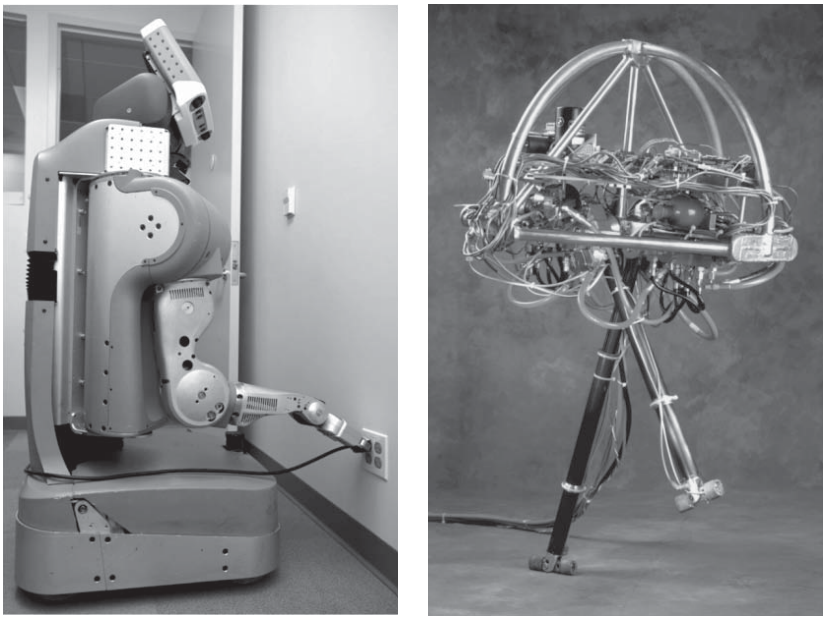
Für mobile Roboter sind die Freiheitsgrade nicht unbedingt dasselbe wie die Anzahl der bewegten Elemente. Betrachten Sie beispielsweise Ihr Auto: Es kann vorwärts- und rückwärtsfahren und es kann wenden, sodass es zwei Freiheitsgrade hat. Im Gegensatz dazu ist die kinematische Konfiguration eines Autos dreidimensional: Auf einer offenen, flachen Oberfläche kann man ein Auto leicht an jeden beliebigen Punkt (x, y) steuern, und zwar in jede Richtung (siehe ▶ Abbildung 25.4(b)). Damit hat das Auto drei **effektive Freiheitsgrade**, aber nur zwei **steuerbare Freiheitsgrade**. Wir sagen, ein Roboter ist **nichtholonom**, wenn er mehr effektive Freiheitsgrade als steuerbare Freiheitsgrade hat, und **holonom**, wenn die beiden Zahlen gleich sind. Holo-

4 „Kinematik“ leitet sich von dem griechischen Wort *kinema* für *Bewegung* ab.

nome Roboter sind leichter zu steuern – es wäre viel einfacher, ein Auto einzuparken, das nicht nur vorwärts und rückwärts, sondern auch seitlich fahren könnte –, aber gleichzeitig sind sie auch mechanisch sehr viel komplexer. Die meisten Roboterarme sind holonom, die meisten mobilen Roboter nichtholonom.

Mobile Roboter verfügen über die verschiedensten Fortbewegungsmechanismen, unter anderem Räder, Ketten und Beine. Roboter mit **Differentialantrieb** besitzen jeweils an einer Seite zwei unabhängig voneinander angetriebene Räder (oder Ketten), wie beispielsweise bei einem Panzer. Wenn sich beide Räder mit derselben Geschwindigkeit drehen, bewegt sich der Roboter geradeaus. Wenn sie sich in unterschiedliche Richtungen bewegen, dreht sich der Roboter an Ort und Stelle. Eine Alternative ist der **Synchronantrieb**, wobei sich jedes Rad bewegen und um die eigene Achse drehen kann. Um Chaos zu vermeiden, sind die Räder eng aufeinander abgestimmt. Zum Beispiel zeigen alle Räder bei einer Geradeausbewegung in dieselbe Richtung und bewegen sich mit derselben Geschwindigkeit. Sowohl Differential- als auch Synchronantriebe sind nichtholonom. Einige teurere Roboter verwenden holonome Antriebe mit drei oder mehr Rädern, die unabhängig voneinander ausgerichtet und bewegt werden können.

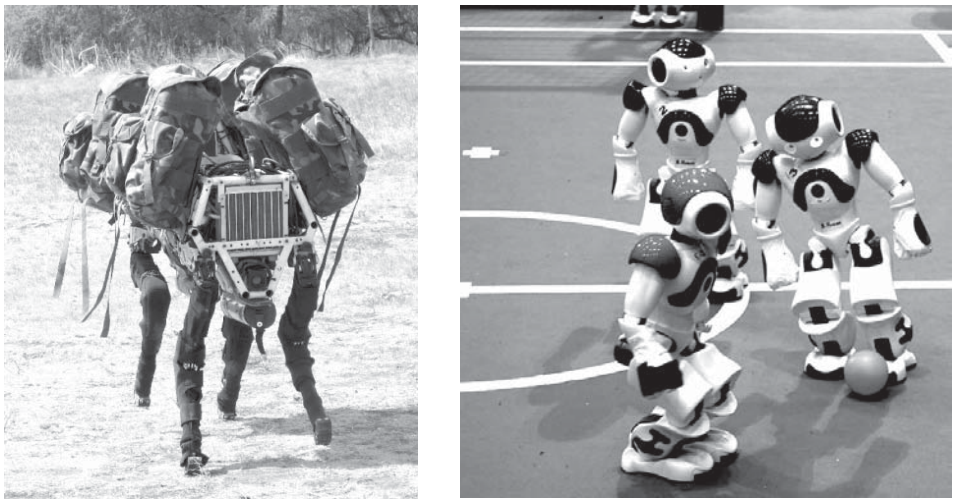
Manche mobile Roboter sind mit Armen ausgerüstet. ▶ Abbildung 25.5(a) zeigt einen zweiarmigen Roboter. Diese Roboterarme kompensieren die Schwerkraft mithilfe von Federn und setzen externen Kräften einen nur geringen Widerstand entgegen. Dieses Design minimiert die körperliche Gefahr für Personen, die versehentlich mit einem derartigen Roboter kollidieren. Bei Robotern für das häusliche Umfeld sind solche Betrachtungen enorm wichtig.



a b

Abbildung 25.5: (a) Mobiler Manipulator, der sein Ladekabel in eine Wandsteckdose einführt. Bild mit freundlicher Genehmigung von Willow Garage, © 2009. Einer der Roboter von Marc Raibert, der sich auf Beinen bewegt.

Beine können anders als Räder mit sehr unwegsamem Gelände zurechtkommen. Beine sind jedoch auf ebenen Oberflächen sehr langsam und sie sind mechanisch schwierig zu erstellen. Roboterforscher haben alles Mögliche versucht, von einem einzigen bis zu einem Dutzend Beinen. Roboter mit Beinen können gehen, laufen und sogar springen – wie in ▶ Abbildung 25.5(b) gezeigt. Dieser Roboter ist **dynamisch stabil**, d.h., er kann aufrecht bleiben, während er springt. Ein Roboter, der aufrecht bleiben kann, ohne seine Beine zu bewegen, wird als **statisch stabil** bezeichnet. Ein Roboter ist statisch stabil, wenn sein Schwerkraftzentrum über dem durch die Beine aufgespannten Polygon liegt. Der in ▶ Abbildung 25.6(a) gezeigte vierbeinige Roboter scheint statisch stabil zu sein. Allerdings hebt er beim Gehen mehrere Beine gleichzeitig, was ihn zu einem dynamisch stabilen Roboter macht. Der Roboter kann auf Schnee und Eis gehen. Er stürzt auch nicht, selbst wenn man ihn anstößt (wie in online zugänglichen Videos demonstriert). Zweibeinige Roboter wie die in ▶ Abbildung 25.6(b) gezeigten sind dynamisch stabil.



a b

Abbildung 25.6: (a) Vierbeiniger, dynamisch stabiler Roboter „Big Dog". Bild mit freundlicher Genehmigung von Boston Dynamics, © 2009. (b) RoboCup Standard Platform League-Weltmeisterschaft 2009 mit dem Gewinnerteam B-Human vom DFKI an der Universität Bremen. Das gesamte Spiel über war B-Human seinen Gegnern mit 64:1 überlegen. Der Erfolg ist zurückzuführen auf probabilistische Zustandsbewertung mithilfe von Partikel- und Kalman-Filtern, auf Modelle für maschinelles Lernen zur Gangoptimierung und auf dynamische Stoßbewegungen. Bild mit freundlicher Genehmigung des DKFI, © 2009.

Es sind auch andere Bewegungsmethoden möglich: Luftfahrzeuge verwenden Propeller oder Turbinen, Unterwasserfahrzeuge Propeller oder Strahlruder, wie sie auch bei Unterseebooten üblich sind. Roboter-Luftschiffe nutzen Thermikeffekte, um sich in der Luft zu halten.

Sensoren und Effektoren allein machen noch keinen Roboter. Ein vollständiger Roboter braucht auch eine Kraftquelle, um seine Effektoren zu bewegen. Der **Elektromotor** ist der beliebteste Mechanismus sowohl für die Bewegung der Manipulatoren als auch für den Antrieb. Allerdings gibt es auch Nischenanwendungen mit **pneumatischen Antrieben**, die komprimiertes Gas verwenden, und mit **hydraulischen Antrieben**, die mit Flüssigkeiten unter Druck arbeiten.

25.3 Roboterwahrnehmung

Wahrnehmung ist der Prozess, wie Roboter Sensormessungen in interne Darstellungen der Umgebung abbilden. Die Wahrnehmung ist schwierig, weil Sensoren im Allgemeinen ein Rauschen aufweisen und die Umgebung partiell beobachtbar, unvorhersehbar und häufig dynamisch ist. Mit anderen Worten haben Roboter sämtliche Probleme der **Zustandsabschätzung** (oder **Filterung**), die *Abschnitt 15.2* behandelt hat. Als Faustregel gilt, dass sich gute interne Darstellungen für Roboter durch drei Eigenschaften auszeichnen: Sie enthalten ausreichend viele Informationen für den Roboter, damit dieser gute Entscheidungen treffen kann, sie sind strukturiert, sodass sie effizient aktualisiert werden können, und sie sind natürlich, sodass die internen Variablen den natürlichen Zustandsvariablen in der physischen Welt entsprechen.

In *Kapitel 15* haben wir gesehen, dass Kalman-Filter, HMMs und dynamische Bayessche Netze die Übergangs- und Sensormodelle einer teilweise beobachtbaren Umgebung repräsentieren können, und wir haben sowohl exakte als auch annähernde Algorithmen für die Aktualisierung des **Belief State** vorgestellt – die A-posteriori-Wahrscheinlichkeitsverteilung über die Umgebungszustandsvariablen. *Kapitel 15* hat mehrere Modelle mit dynamischen Bayesschen Netzen für diesen Prozess vorgestellt. Für Robotikprobleme nutzen wir in der Regel auch die zuletzt ausgeführten Aktionen des Roboters als beobachtete Variablen im Modell. ▶ Abbildung 25.7 zeigt die Notation, die wir in diesem Kapitel verwenden: $\mathbf{X}_t$ ist der Zustand der Umgebung (einschließlich des Roboters) zur Zeit t, $\mathbf{Z}_t$ ist die Beobachtung, die zur Zeit t eintraf, und A_t ist die Aktion, die ausgeführt wurde, nachdem die Beobachtung eingetroffen ist.

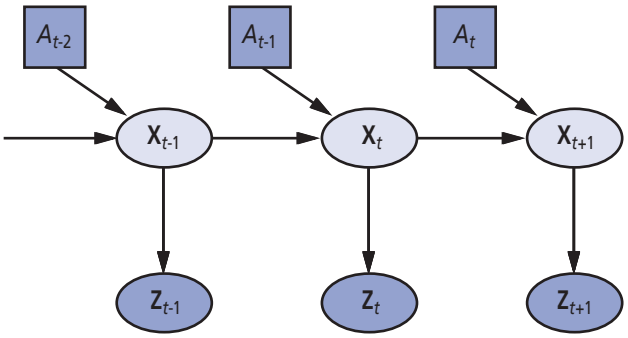


Abbildung 25.7: Roboterwahrnehmung kann als temporäre Inferenz von Sequenzen aus Aktionen und Messungen betrachtet werden, wie sie dieses dynamische Bayessche Netz veranschaulicht.

Wir möchten den neuen Belief State $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{z}_{1:t+1}, a_{1:t})$ aus dem aktuellen Belief State $\mathbf{P}(\mathbf{X}_t \mid \mathbf{z}_{1:t}, a_{1:t-1})$ und der neuen Beobachtung $\mathbf{z}_{t+1}$ berechnen. Dies kennen Sie bereits aus *Abschnitt 15.2*, doch gibt es hier zwei Unterschiede: Wir konditionieren explizit auf die Aktionen sowie die Beobachtungen und wir haben es jetzt mit *stetigen* statt *diskreten* Variablen zu tun. Somit modifizieren wir die rekursive Filtergleichung (*15.5* in *Abschnitt 15.2.1*), um Integration statt Summation zu verwenden:

$$\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{z}_{1:t+1}, a_{1:t})$$

$$= \alpha \mathbf{P}(\mathbf{z}_{t+1} \mid \mathbf{X}_{t+1}) \int \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t, a_t) P(\mathbf{x}_t \mid \mathbf{z}_{1:t}, a_{1:t-1}) d\mathbf{x}_t. \qquad (25.1)$$

Diese Gleichung besagt, dass die A-posteriori-Verteilung über die Zustandsvariablen $\mathbf{X}$ zur Zeit $t+1$ rekursiv aus der entsprechenden Schätzung von einem vorherigen Zeitschritt berechnet wird. Diese Berechnung beinhaltet die vorhergehende Aktion a_t sowie die aktuelle Sensormessung $\mathbf{z}_{t+1}$. Ist unser Ziel beispielsweise, einen Fußball spielenden Roboter zu entwickeln, könnte $\mathbf{X}_{t+1}$ die Position des Fußballes relativ zum Roboter sein. Die A-posteriori-Verteilung $\mathbf{P}(\mathbf{X}_t \mid \mathbf{z}_{1:t}, a_{1:t-1})$ ist eine Wahrscheinlichkeitsverteilung über alle Zustände, die abdeckt, was wir aus vergangenen Sensormessungen und Steuerungen wissen. Gleichung (25.1) zeigt uns, wie wir diese Position rekursiv abschätzen, indem wir schrittweise Sensormessungen (z.B. Kamerabilder) und Roboterbewegungsbefehle aufnehmen. Die Wahrscheinlichkeit $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t, a_t)$ wird als **Übergangsmodell** oder **Bewegungsmodell** bezeichnet, $\mathbf{P}(\mathbf{z}_{t+1} \mid \mathbf{X}_{t+1})$ ist das **Sensormodell**.

25.3.1 Lokalisierung und Zuordnung

Lokalisierung ist das Problem, zu erkennen, wo sich Dinge – einschließlich des Roboters selbst – befinden. Das Wissen darüber, wo sich Dinge befinden, ist für jede erfolgreiche physische Interaktion mit der Umgebung eminent wichtig. Beispielsweise müssen Roboter-Manipulatoren die Position der Objekte kennen, die sie manipulieren. Navigationsroboter müssen wissen, wo sie sich befinden, um ihren Weg zu einer Zielposition zu finden.

Um das Ganze so einfach wie möglich zu halten, betrachten wir einen mobilen Roboter, der sich langsam in einer ebenen 2D-Welt bewegt. Außerdem nehmen wir an, dass der Roboter eine genaue Karte der Umgebung besitzt. (Ein Beispiel für eine solche Karte finden Sie in ▶ Abbildung 25.10). Die Pose eines derartigen mobilen Roboters ist durch seine zwei kartesischen Koordinaten mit den Werten x und y definiert sowie durch seinen Kurs mit dem Wert θ, wie in ▶ Abbildung 25.8(a) gezeigt. Wenn wir diese drei Werte in einem Vektor anordnen, ist jeder konkrete Zustand durch $\mathbf{X}_t = (x_t, y_t, \theta_t)^\top$ gegeben. So weit, so gut.

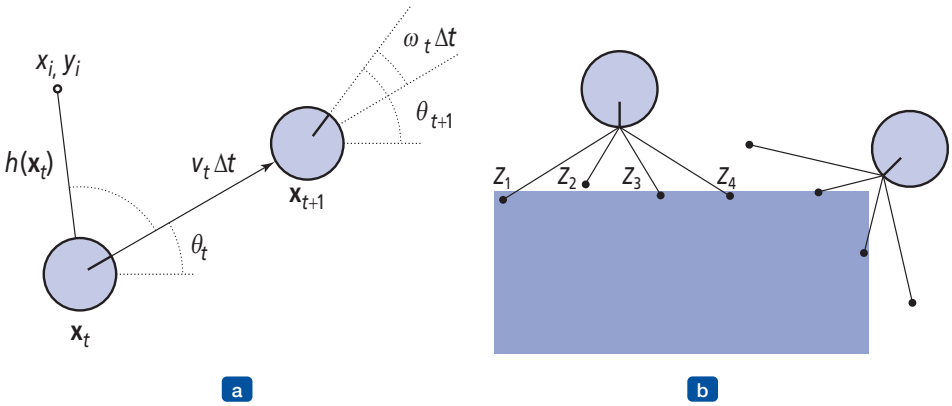


Abbildung 25.8: (a) Ein vereinfachtes kinematisches Modell eines mobilen Roboters. Der Roboter ist als Kreis dargestellt und eine Innenlinie markiert die Vorwärtsrichtung. Der Zustand $\mathbf{x}_t$ besteht aus der Position (x_t, y_t) (implizit gezeigt) und der Orientierung θ_t. Den neuen Zustand $\mathbf{x}_{t+1}$ erhält man durch eine Positions- und Ausrichtungsaktualisierung von $v_t \Delta_t$ bzw. $\omega_t \Delta_t$. Darüber hinaus haben wir zur Zeit t einen Orientierungspunkt an der Stelle (x_i, y_i) beobachtet. (b) Das Bereichsscan-Sensormodell. Die beiden möglichen Roboterposen sind für einen bestimmten Bereichsscan (z_1, z_2, z_3, z_4) gezeigt. Es ist wahrscheinlicher, dass die Pose auf der linken Seite den Bereichsscan erzeugt hat als die Pose auf der rechten Seite.

In der kinematischen Annäherung besteht jede Aktion aus der „unmittelbaren“ Spezifikation von zwei Geschwindigkeiten – einer Translationsgeschwindigkeit v_t und einer Rotationsgeschwindigkeit ω_t. Für kleine Zeitintervalle Δt ist ein einfaches deterministisches Modell der Bewegung derartiger Roboter gegeben durch:

$$\hat{\mathbf{X}}_{t+1} = f\left(\mathbf{X}_t, \underbrace{v_t, \omega_t}_{a_t}\right) = \mathbf{X}_t + \begin{pmatrix} v_t \Delta t \cos\theta_t \\ v_t \Delta t \sin\theta_t \\ \omega_t \Delta t \end{pmatrix} 90^\circ.$$

Die Notation $\hat{\mathbf{X}}$ bezieht sich auf eine deterministische Zustandsvorhersage. Natürlich sind physische Roboter etwas unvorhersehbar. Dies wird häufig durch eine Gaußsche Verteilung mit dem Mittelwert $f(\mathbf{X}_t, v_t, \omega_t)$ und der Kovarianz Σ_x modelliert. (Eine mathematische Definition finden Sie in Anhang A.)

$$\mathbf{P}\left(\mathbf{X}_{t+1} \mid \mathbf{X}_t, v_t, \omega_t\right) = N\left(\hat{\mathbf{X}}_{t+1}, \sum_x\right)$$

Diese Wahrscheinlichkeitsverteilung ist das Bewegungsmodell des Roboters. Es modelliert die Wirkungen der Bewegung a_t auf die Position des Roboters.

Als Nächstes brauchen wir ein Sensormodell. Wir betrachten zwei Arten von Sensormodellen. Das erste geht davon aus, dass die Sensoren *stabile, erkennbare* Merkmale der Umgebung erkennen, sogenannte **Orientierungspunkte**. Für jeden Orientierungspunkt werden der Bereich und die Orientierung gemeldet. Angenommen, der Zustand des Roboters ist $\mathbf{x}_t = (x_t, y_t, \theta_t)^\top$ und er erkennt einen Orientierungspunkt, dessen Position mit $(x_i, y_i)^\top$ bekannt ist. Ohne Rauschen können der Bereich und die Orientierung durch einfache Geometrie berechnet werden (siehe Abbildung 25.8(a)). Die genaue Vorhersage des beobachteten Bereiches und der Orientierung wäre dann:

$$\hat{\mathbf{z}}_t = h(\mathbf{x}_t) = \begin{pmatrix} \sqrt{(x_t - x_i)^2 + (y_t - y_i)^2} \\ \arctan\dfrac{y_i - y_t}{x_i - x_t} - \theta_t \end{pmatrix}.$$

Auch hier stört Rauschen unsere Messungen. Der Einfachheit halber könnte man ein Gaußsches Rauschen mit der Kovarianz Σ_z annehmen und erhält folgendes Sensormodell:

$$P(\mathbf{z}_t \mid \mathbf{x}_t) = N\left(\hat{\mathbf{z}}_t, \sum_z\right).$$

Ein etwas schwierigeres Sensormodell wird für ein Array von Entfernungssensoren verwendet, die jeweils eine feste Orientierung relativ zum Roboter besitzen. Derartige Sensoren erzeugen einen Vektor mit Entfernungswerten $\mathbf{z}_t = (z_1, ..., z_M)^\top$. Für eine Pose $\mathbf{x}_t$ sei $\hat{z}_j$ die genaue Entfernung entlang der j-ten Strahlrichtung von $\mathbf{x}_t$ zum nächsten Hindernis. Wie zuvor wird dies durch ein Gaußsches Rauschen gestört. Normalerweise gehen wir davon aus, dass die Fehler für die verschiedenen Strahlrichtungen unabhängig voneinander und identisch verteilt sind; somit erhalten wir:

$$P(\mathbf{z}_t \mid \mathbf{x}_t) = \alpha \prod_{j=1}^{M} e^{-(z_j - \hat{z}_j)/2\sigma^2}.$$

▶ Abbildung 25.8(b) zeigt ein Beispiel für einen vierstrahligen Bereichsscan und zwei mögliche Roboterposen, von denen eine sehr wahrscheinlich den beobachteten Scan erzeugt hat, die andere nicht. Beim Vergleich mit dem Orientierungspunktmodell zeigt sich der Vorteil des Bereichsscan-Modells: Es ist nicht erforderlich, einen Orientierungspunkt zu *identifizieren*, bevor sich der Bereichsscan interpretieren lässt. So steht der Roboter in Abbildung 25.8(b) einer merkmalslosen Wand gegenüber. Wenn es dagegen sichtbare, identifizierbare Orientierungspunkte *gibt*, können sie eine unmittelbare Lokalisierung liefern.

In *Kapitel 15* haben wir den Kalman-Filter beschrieben, der den Belief State als einzelne multivariate Gaußverteilung darstellt, sowie den Partikelfilter, der den Belief State als Sammlung von Partikeln darstellt, die Zuständen entsprechen. Die meisten modernen Lokalisierungsalgorithmen verwenden eine der beiden Darstellungen des Belief State für den Roboter, $\mathbf{P}(X_t \mid \mathbf{z}_{1:t}, a_{1:t-1})$.

Eine Lokalisierung, die einen Partikelfilter verwendet, wird auch als **Monte-Carlo-Lokalisierung** oder **MCL** bezeichnet. Der MCL-Algorithmus ist eine Instanz des Partikelfilter-Algorithmus gemäß Abbildung 15.17 (*Abschnitt 15.5.3*). Wir müssen nur ein geeignetes Bewegungsmodell und ein Sensormodell bereitstellen. ▶ Abbildung 25.9 zeigt eine Version unter Verwendung des Bereichsscan-Modells. Abbildung 25.10 veranschaulicht die Arbeitsweise des Algorithmus. Hier findet der Roboter heraus, wo er sich innerhalb eines Bürogebäudes befindet. Im ersten Bild sind die Partikel einheitlich nach der A-priori-Verteilung verteilt, was auf eine globale Unsicherheit in Bezug auf die Position des Roboters hinweist. Im zweiten Bild trifft der erste Satz von Messungen ein und die Partikel bilden Cluster in den Bereichen hoher A-posteriori-Belief-States. Im dritten Bild stehen ausreichend viele Messungen zur Verfügung, um alle Partikel an einer einzigen Position zu versammeln.

```
function MONTE-CARLO-LOCALIZATION(a, z, N, P(X'|X, v, w), P(z|z*), m)
         returns eine Stichprobenmenge für den nächsten Zeitschritt
   inputs: a, Robotergeschwindigkeiten v und w
           z, ein Bereichsscan z_1, ..., z_M
           P(X'|X, v, w), Bewegungsmodell
           P(z|z*), Bereichssensor-Rauschmodell
           m, 2D-Karte der Umgebung
  persistent: S, ein Stichprobenvektor der Größe N
  local variables: W, ein Gewichtsvektor der Größe N
                   S', ein temporärer Partikelvektor der Größe N
                   W', ein Gewichtsvektor der Größe N
  if S ist leer then     /* Initialisierungsphase */
    for i = 1 to N do
      S[i] ← Stichprobe aus P(X_0)
    for i = 1 to N do /* Aktualisierungszyklus */
      S'[i] ← Strichprobe aus P(X'|X = S[i], v, w)
      W'[i] ← 1
      for j = 1 to M do
        z* ← RAYCAST(j, X = S'[i], m)
        W'[i] ← W'[i] · P(z_j|z*)
  S ← WEIGHTED-SAMPLE-WITH-REPLACEMENT(N ,S' ,W')
return S
```

Abbildung 25.9: Ein Monte-Carlo-Lokalisierungsalgorithmus unter Verwendung eines Bereichsscan-Sensormodells mit unabhängigem Rauschen.

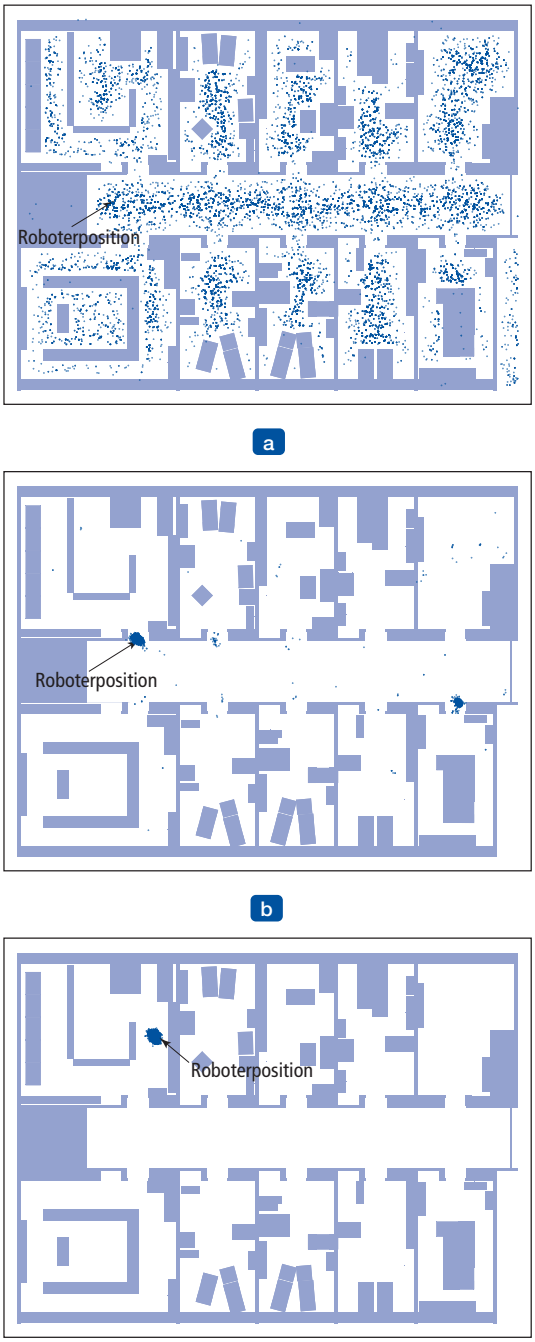


Abbildung 25.10: Monte Carlo-Lokalisierung, ein Partikelfilter-Algorithmus für die mobile Roboterlokalisierung. (a) Anfängliche globale Unsicherheit. (b) Anfängliche bimodale Unsicherheit nach der Navigation in den (symmetrischen) Korridor. (c) Unimodale Unsicherheit nach Eintritt in ein bestimmtes Büro.

Der Kalman-Filter ist die andere wichtige Lokalisierungsmöglichkeit. Ein Kalman-Filter stellt die A-posteriori-Verteilung $\mathbf{P}(\mathbf{X}t \mid \mathbf{z}_{1:t}, a_{1:t-1})$ durch eine Gaußsche Verteilung dar. Der Mittelwert dieser Gaußschen Verteilung wird als μ_t und ihre Kovarianz als Σ_t bezeichnet. Das größte Problem bei Gaußschen Belief States ist, dass sie nur unter linearen Bewegungsmodellen f und linearen Messmodellen h geschlossen sind. Für nichtlineare f oder h ist das Ergebnis der Aktualisierung ein Filter, der in der Regel kein Gaußscher Filter ist. Lokalisierungsalgorithmen, die Kalman-Filter benutzen, linearisieren also die Bewegungs- und Sensormodelle. Die **Linearisierung** ist eine lokale Annäherung einer nichtlinearen Funktion durch eine lineare Funktion. ▶ Abbildung 25.11 veranschaulicht das Konzept der Linearisierung für ein (eindimensionales) Roboter-Bewegungsmodell. Auf der linken Seite zeigt sie ein nichtlineares Bewegungsmodell $f(\mathbf{x}_t, a_t)$ (das Steuerelement a_t wurde in dieser Abbildung weggelassen, weil es für die Linearisierung keine Rolle spielt). Auf der rechten Seite wird diese Funktion durch eine lineare Funktion $\tilde{f}(\mathbf{x}_t, a_t)$ angenähert. Diese lineare Funktion bildet eine Tangente zu f am Punkt μ_t, dem Mittelwert unserer Zustandsschätzung zur Zeit t. Eine solche Linearisierung wird auch als **Taylor-Expansion** (ersten Grades) bezeichnet. Ein Kalman-Filter, der f und h über eine Taylor-Expansion linearisiert, wird auch als **erweiterter Kalman-Filter** (oder EKF) bezeichnet. ▶ Abbildung 25.12 zeigt eine Folge von Schätzungen eines Roboters, der einen erweiterten Kalman-Filter-Lokalisierungsalgorithmus verwendet. Wenn sich der Roboter bewegt, steigt die Unsicherheit in seiner Positionsschätzung, wie durch die Fehlerellipsen gezeigt. Der Fehler sinkt, wenn der Roboter den Bereich und die Haltung zu einem Orientierungspunkt mit bekannter Position erkennt. Der Fehler steigt wieder, wenn der Roboter den Orientierungspunkt nicht mehr sieht. EFK-Algorithmen funktionieren gut, wenn die Orientierungspunkte einfach zu identifizieren sind. Andernfalls kann die A-posteriori-Verteilung multimodal sein, wie in Abbildung 25.10(b) gezeigt. Das Problem, die Identität von Orientierungspunkten kennen zu müssen, ist ein Beispiel für das **Datenzuordnungsproblem**, das in Abbildung 15.6 dargestellt ist.

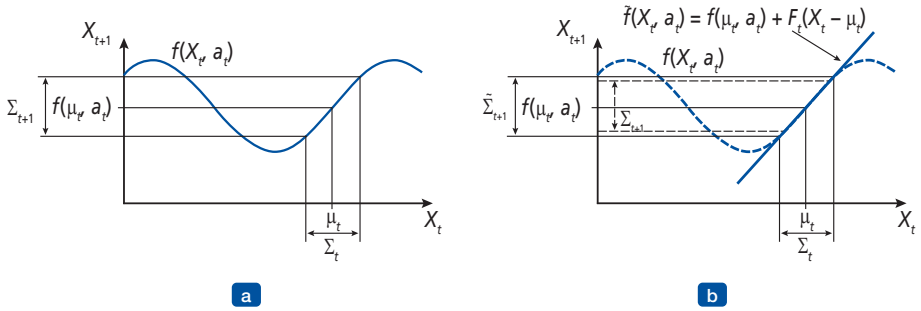


Abbildung 25.11: Eindimensionale Darstellung eines linearisierten Bewegungsmodells: (a) Die Funktion f und die Projektion eines Mittelwertes μ_t und eines Kovarianzintervalls (basierend auf Σ_t) in die Zeit $t + 1$. (b) Die linearisierte Version ist die Tangente von f bei μ_t. Die Projektion über dem Mittelwert μ_t ist korrekt. Die Projektionskovarianz $\tilde{\Sigma}_{t+1}$ unterscheidet sich jedoch von Σ_{t+1}.

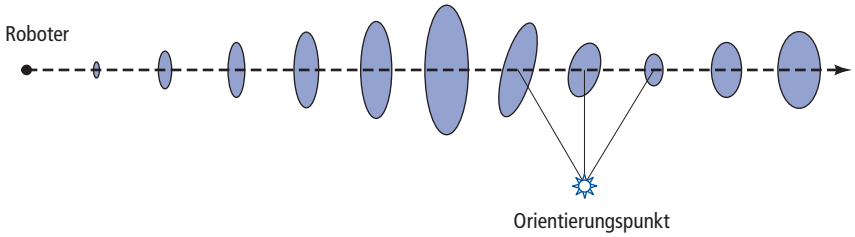


Abbildung 25.12: Beispiel für eine Lokalisierung unter Verwendung erweiterter Kalman-Filter. Während sich der Roboter auf einer geraden Linie bewegt, steigt die Unsicherheit schrittweise, wie durch die Fehlerellipsen gezeigt. Beobachtet er einen Orientierungspunkt mit bekannter Position, reduziert sich die Unsicherheit.

In manchen Situationen steht keine Karte der Umgebung zur Verfügung. Der Roboter muss sich dann eine Karte beschaffen. Das ähnelt dem Henne-Ei-Problem: Der navigierende Roboter muss seine Position relativ zu einer ihm unbekannten Karte bestimmen und gleichzeitig diese Karte erstellen, wobei er aber seine tatsächliche Position nicht genau kennt. Dieses Problem ist für viele Roboteranwendungen wichtig und wurde unter der Bezeichnung **SLAM** (Simultaneous Localization And Mapping, Simultane Lokalisierung und Kartenerstellung) ausgiebig untersucht.

SLAM-Probleme lassen sich mithilfe verschiedener probabilistischer Techniken lösen, unter anderem mit dem weiter vorn behandelten erweiterten Kalman-Filter. Ein EKF ist unkompliziert anzuwenden: Lediglich den Zustandsvektor vergrößern, um die Positionen der Orientierungspunkte in der Umgebung einzubinden. Erfreulicherweise aktualisiert der EKF die Skalierungen quadratisch, sodass die Berechnung für kleine Karten (z.B. einige hundert Orientierungspunkte) durchaus machbar ist. Umfangreichere Karten werden oftmals mithilfe von Methoden der Graphenrelaxation erhalten, ähnlich den Inferenztechniken mit Bayesschen Netzen, die *Kapitel 14* behandelt hat. Auch Erwartungsmaximierung wird für SLAM verwendet.

25.3.2 Weitere Wahrnehmungstypen

Bei der Wahrnehmung der Roboter haben wir es nicht nur mit Lokalisierung und Kartenerstellung zu tun. Roboter nehmen auch die Temperatur, Gerüche, akustische Signale usw. wahr. Viele dieser Größen lassen sich mithilfe von Varianten dynamischer Bayesscher Netze abschätzen. Für derartige Schätzungen sind lediglich bedingte Wahrscheinlichkeitsverteilungen erforderlich, die die Entwicklung von Zustandsvariablen über die Zeit charakterisieren, sowie andere Verteilungen, die die Relation von Messungen zu Zustandsvariablen beschreiben.

Es ist auch möglich, einen Roboter als reaktiven Agenten zu programmieren, ohne explizit über Wahrscheinlichkeitsverteilungen von Zuständen nachzudenken.

Der Trend in der Robotik geht klar in die Richtung von Repräsentationen mit wohl definierter Semantik. Probabilistische Techniken übertreffen für viele schwierige Wahrnehmungsprobleme andere Ansätze wie beispielsweise Lokalisierung und Kartenerstellung. Statistische Techniken sind jedoch häufig zu schwerfällig. Einfache Lösungen können in der Praxis genauso effektiv sein. Um bei der Entscheidung zu helfen, welcher Ansatz verwendet werden soll, ist die Erfahrung mit realen physischen Robotern immer noch die beste Wahl.

25.3.3 Maschinelles Lernen in der Roboterwahrnehmung

In der Roboterwahrnehmung spielt maschinelles Lernen eine wichtige Rolle. Das gilt insbesondere dann, wenn die beste interne Darstellung nicht bekannt ist. Ein gebräuchlicher Ansatz ist es, hochdimensionale Sensor-Datenströme auf Räume mit weniger Dimensionen durch nicht überwachte Lernverfahren (siehe *Kapitel 18*) abzubilden. Ein derartiges Konzept wird als **Einbettung in niedrigerer Dimension** bezeichnet. Maschinelles Lernen ermöglicht es, Sensor- und Bewegungsmodelle anhand von Daten zu lernen und gleichzeitig geeignete interne Darstellungen zu finden.

Eine andere Technik des maschinellen Lernens versetzt Roboter in die Lage, sich ständig an umfassende Änderungen in Sensormessungen anzupassen. Stellen Sie sich vor, Sie kommen aus einem Raum, in den die Sonne scheint, in einen dunklen Raum mit Neonlicht. Zweifellos werden die Objekte jetzt dunkler erscheinen. Doch der Wechsel der Lichtquelle wirkt sich auch auf alle Farben aus: Im Neonlicht ist die grüne Komponente stärker vertreten als im Sonnenlicht. Doch irgendwie scheinen wir diese Änderung nicht zu bemerken. Wenn wir zusammen mit anderen Personen in einen Raum mit Neonbeleuchtung gehen, erwarten wir sicherlich nicht, dass sich deren Gesichter plötzlich grün färben. Unsere Wahrnehmung passt sich schnell an neue Beleuchtungsbedingungen an und unser Gehirn ignoriert die Unterschiede.

Durch adaptive Wahrnehmungstechniken können sich Roboter an derartige Wechsel anpassen. ▶ Abbildung 25.13 zeigt ein derartiges Beispiel für autonomes Fahren. Ein unbemanntes Landfahrzeug passt hier seinen Klassifizierer für das Konzept „befahrbare Oberfläche“ an. Wie funktioniert dies? Der Roboter liefert mithilfe eines Lasers die Klassifizierung für einen kleinen Bereich unmittelbar vor dem Roboter. Erweist sich dieser Bereich im Scan des Laserentfernungsmessers als flach, wird er als positives Trainingsbeispiel für das Konzept „befahrbare Oberfläche“ herangezogen. Eine Technik mit gemischter Gaußverteilung ähnlich dem in *Kapitel 20* besprochenen EM-Algorithmus wird dann trainiert, um die spezifischen Farb- und Texturkoeffizienten des kleinen Stichprobenbereiches zu erkennen. Die Bilder in Abbildung 25.13 zeigen das Ergebnis, wie dieser Klassifizierer auf das vollständige Bild angewandt wurde.

Abbildung 25.13: Ergebnisfolge eines Klassifizierers für „Befahrbare Oberfläche“ mithilfe einer adaptiven Vision. In (a) wird nur die Straße als befahrbar klassifiziert (gestreifter Bereich). Die V-förmige dunkle Linie zeigt die Richtung des Fahrzeuges an. In (b) wird das Fahrzeug angewiesen, die Straße zu verlassen und auf einer grasartigen Oberfläche zu fahren. Der Klassifizierer beginnt dann, einen Teil des Grasabschnittes als befahrbar zu klassifizieren. In (c) hat das Fahrzeug sein Modell der befahrbaren Oberfläche aktualisiert, um sowohl dem Gras als auch der Straße zu entsprechen.

Methoden, durch die Roboter ihre eigenen Trainingsdaten (mit Beschriftungen!) sammeln, heißen **selbstüberwachend**. In diesem Fall wandelt der Roboter durch maschinelles Lernen einen Nahfeldsensor, der für Geländeklassifizierung geeignet ist, in einen Sensor um, der wesentlich weiter sehen kann. Dadurch kann der Roboter schneller fahren und muss seine Geschwindigkeit nur dann verringern, wenn das Sensormodell sagt, dass das Gelände wechselt und durch die Nahfeldsensoren genauer untersucht werden muss.

25.4 Bewegung planen

Sämtliche Überlegungen eines Roboters münden letztlich in Entscheidungen, wie Effektoren zu bewegen sind. Beim Problem der **Punkt-zu-Punkt-Bewegung** geht es darum, den Roboter oder seinen End-Effektor an eine vorgegebene Zielposition zu bringen. Eine größere Herausforderung ist das Problem der **konformen Bewegung**, wobei sich ein Roboter bewegt, während er in physischem Kontakt mit einem Hindernis ist. Ein Beispiel dafür ist ein Roboter-Manipulator, der eine Glühbirne einschraubt, oder ein Roboter, der eine Kiste über einen Tisch schiebt.

Wir beginnen damit, eine geeignete Repräsentation zu suchen, wodurch die Bewegungsplanungsprobleme beschrieben und gelöst werden können. Es zeigt sich, dass der **Konfigurationsraum** – der Raum der Roboterzustände, die durch Position, Ausrichtung und Gelenkwinkel definiert sind – ein besserer Arbeitsraum als der ursprüngliche 3D-Raum ist. Das **Pfadplanungsproblem** macht es erforderlich, einen Pfad von einer Konfiguration zu einer anderen im Konfigurationsraum zu finden. Wir haben bereits viele Versionen des Pfadplanungssystems in diesem Buch kennengelernt; die Robotik bringt als zusätzliche Komplikation mit, dass die Pfadplanung mit *stetigen* Räumen zu tun hat. Es gibt vor allem zwei Ansätze: **Zellzerlegung** und **Skelettierung**. Beide reduzieren das stetige Pfadplanungsproblem auf ein diskretes Graphensuchproblem. In diesem Abschnitt wollen wir davon ausgehen, dass die Bewegung deterministisch und die Lokalisierung des Roboters exakt ist. Nachfolgende Abschnitte werden diese Annahmen lockern.

25.4.1 Konfigurationsraum

Wir beginnen mit einer einfachen Repräsentation für ein einfaches Roboterbewegungsproblem. Betrachten Sie den in ▶ Abbildung 25.14(a) gezeigten Roboterarm. Er hat zwei Gelenke, die sich unabhängig voneinander bewegen. Durch die Bewegung der Gelenke werden die (x, y)-Koordinaten des Ellenbogens und des Greifers geändert. (Der Arm kann sich nicht in z-Richtung bewegen.) Dies legt nahe, dass die Konfiguration des Roboters durch eine vierdimensionale Koordinate beschrieben werden kann: (x_e, y_e) für die Position des Ellenbogens relativ zur Umgebung und (x_g, y_g) für die Position des Greifers. Offensichtlich charakterisieren diese vier Koordinaten den vollständigen Zustand des Roboters. Sie bilden das, was auch als **Arbeitsraum-Repräsentation** bezeichnet wird, weil die Koordinaten des Roboters im selben Koordinatensystem angegeben werden wie die Objekte, die er manipulieren (oder umfahren) will. Arbeitsraum-Repräsentationen sind gut geeignet für die Kollisionsüberprüfung, insbesondere wenn der Roboter und alle Objekte durch einfache polygonale Modelle dargestellt sind.

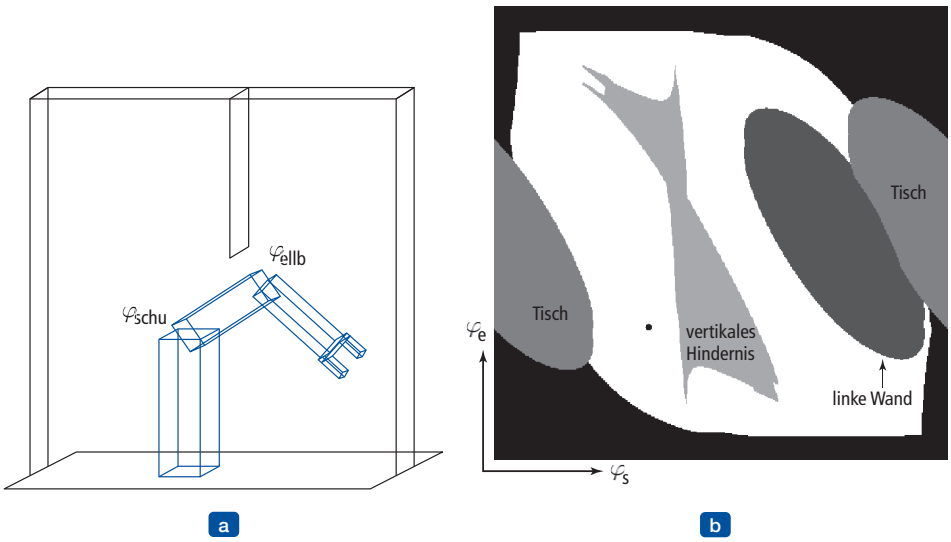


Abbildung 25.14: (a) Arbeitsraum-Repräsentation eines Roboterarms mit zwei Freiheitsgraden. Der Arbeitsraum ist ein Kasten mit einem flachen Hindernis, das von der Decke hängt. (b) Konfigurationsraum desselben Roboters. Nur weiße Bereiche im Raum sind kollisionsfreie Konfigurationen. Der Punkt in dieser Abbildung entspricht der Konfiguration des auf der linken Seite gezeigten Roboters.

Bei der Arbeitsraum-Repräsentation besteht das Problem, dass nicht alle Arbeitsraum-Koordinaten zur Verfügung stehen, selbst wenn es keine Hindernisse gibt. Dies liegt an den **Verbindungsbedingungen** im Raum der verfügbaren Arbeitsraum-Koordinaten. Beispielsweise sind die Ellenbogenposition (x_e, y_e) und die Greiferposition (x_g, y_g) immer einen festen Abstand voneinander entfernt, weil sie durch einen feststehenden Unterarm verbunden sind. Ein Roboterbewegungsplaner, der über Arbeitsraum-Koordinaten definiert ist, steht der Herausforderung gegenüber, Pfade zu erzeugen, die diese Bedingungen berücksichtigen. Das ist besonders schwierig, weil der Zustandsraum stetig ist und die Bedingungen nicht linear sind. Es erweist sich als einfach, mit einer **Konfigurationsraum**-Repräsentation zu planen. Anstatt den Zustand des Roboters durch die kartesischen Koordinaten seiner Elemente zu repräsentieren, stellen wir den Zustand durch eine Konfiguration der Gelenke des Roboters dar. Unser Beispielroboter hat zwei Gelenke. Wir können seinen Zustand also durch die beiden Winkel φ_s und φ_e für das Schultergelenk bzw. Ellenbogengelenk repräsentieren. Wenn es keine Hindernisse gibt, könnte ein Roboter beliebige Werte im Konfigurationsraum einnehmen. Insbesondere bei der Planung eines Pfades könnte man einfach die aktuelle und die Zielkonfiguration durch eine gerade Linie verbinden. Durch die Verfolgung dieses Pfades ändert dann ein Roboter seine Gelenke mit einer konstanten Geschwindigkeit, bis eine Zielposition erreicht ist.

Leider haben auch Konfigurationsräume eigene Probleme. Die Aufgabe eines Roboters wird normalerweise in Arbeitsraum-Koordinaten ausgedrückt, nicht in Konfigurationsraum-Koordinaten. Damit stellt sich die Frage, wie man solche Arbeitsraum-Koordinaten in den Konfigurationsraum abbildet. Die Transformation von Konfigurationsraum-Koordinaten in Arbeitsraum-Koordinaten ist leicht: Es ist lediglich eine Reihe von einfachen Koordinatentransformationen durchzuführen. Diese Transformationen sind linear für Prismagelenke und trigonometrisch für Drehgelenke. Diese Kette aus Koordinatentransformationen wird auch als **Kinematik** bezeichnet.

Das umgekehrte Problem, die Konfiguration eines Roboters zu berechnen, dessen Effektor- Position in Arbeitsraum-Koordinaten angegeben ist, wird auch als **inverse Kinematik** bezeichnet. Die Berechnung der inversen Kinematik ist speziell für Roboter mit vielen Freiheitsgraden schwierig. Insbesondere ist die Lösung selten eindeutig. Abbildung 25.14(a) zeigt eine der zwei möglichen Konfigurationen, die den Greifer in dieselbe Position bringt. (Bei der anderen Konfiguration würde der Ellenbogen unterhalb der Schulter liegen.)

Im Allgemeinen hat dieser Roboterarm mit zwei Gelenken zwischen 0 und 2 inverse kinematische Lösungen für jede beliebige Menge von Arbeitsraum-Koordinaten. Die meisten Industrieroboter verfügen über genügend Freiheitsgrade, um unendlich viele Lösungen für Bewegungsprobleme zu finden. Um zu sehen, wie dies möglich ist, stellen Sie sich einfach vor, wir fügen unserem Beispielroboter ein drittes Drehgelenk hinzu, dessen Drehachse parallel zur der des bereits existierenden Gelenks ist. In einem solchen Fall können wir die Position (aber nicht die Ausrichtung!) des Greifers beibehalten und dennoch die internen Gelenke für die meisten Konfigurationen des Roboters frei drehen. Mit ein paar mehr Gelenken (wie vielen?) können wir denselben Effekt erzielen und dabei auch die Ausrichtung konstant halten. Wir haben bereits ein Beispiel dafür in unserem „Experiment“ gesehen, wo Sie versucht haben, Ihre Hand auf den Tisch zu legen und den Ellbogen zu bewegen. Die kinematische Bedingung Ihrer Handposition ist nicht ausreichend, um die Konfiguration des Ellbogens festzulegen. Mit anderen Worten, die inverse Kinematik Ihres Schulter/Arm-Systems besitzt eine unendliche Anzahl an Lösungen.

Das zweite Problem bei den Konfigurationsraum-Repräsentationen entsteht aus den Hindernissen, die es möglicherweise im Arbeitsraum des Roboters gibt. Unser Beispiel in Abbildung 25.14(a) zeigt mehrere solcher Hindernisse, einschließlich eines frei hängenden Hindernisses, das in die Mitte des Arbeitsraumes des Roboters hineinragt. Im Arbeitsraum nehmen solche Hindernisse einfache geometrische Formen an – insbesondere in den meisten Lehrbüchern über Robotik, die sich normalerweise auf polygonale Hindernisse konzentrieren. Aber wie sehen sie im Konfigurationsraum aus?

▶ Abbildung 25.14(b) zeigt den Konfigurationsraum für unseren Beispielroboter unter der in Abbildung 25.14(a) gezeigten spezifischen Hinderniskonfiguration. Der Konfigurationsraum kann in zwei Unterräume zerlegt werden: den Raum aller Konfigurationen, die ein Roboter erreichen kann, in der Regel als **freier Raum** bezeichnet, sowie den Raum der nicht erreichbaren Konfigurationen, als **belegter Raum** bezeichnet. Der weiße Bereich in Abbildung 25.14(b) entspricht dem freien Raum. Alle anderen Bereiche entsprechen dem belegten Raum. Die unterschiedlichen Schattierungen des belegten Raumes entsprechen den verschiedenen Objekten im Arbeitsraum des Roboters. Der schwarze Bereich, der den gesamten freien Bereich umschließt, entspricht Konfigurationen, in denen der Roboter mit sich selbst kollidiert. Man erkennt ganz einfach, dass Extremwerte der Schulter- oder Ellenbogenwinkel eine solche Verletzung verursachen könnten. Die beiden ovalen Bereiche auf beiden Seiten des Roboters entsprechen dem Tisch, auf dem der Roboter montiert ist. Analog dazu entspricht der dritte ovale Bereich der linken Wand. Schließlich ist das interessanteste Objekt im Konfigurationsraum das vertikale Hindernis, das von der Decke hängt und die Bewegungen des Roboters behindert. Dieses Objekt hat einen eigenartigen Umriss: Es ist höchst nichtlinear und an einigen Stellen sogar konkav. Mit ein bisschen Vorstellungskraft ist der Umriss des Greifers am oberen linken Ende zu erkennen. Wir empfehlen dem Leser, eine kurze Pause zu machen und sich diese Abbildung genauer anzusehen. Der Umriss dieses Hindernisses

ist nicht offensichtlich! Der Punkt in Abbildung 25.14(b) markiert die Konfiguration des Roboters, wie in Abbildung 25.14(a) gezeigt. ▶ Abbildung 25.15 zeigt drei zusätzliche Konfigurationen sowohl im Arbeitsraum als auch im Konfigurationsraum. In der Konfiguration „conf-1“ umschließt der Greifer das vertikale Hindernis.

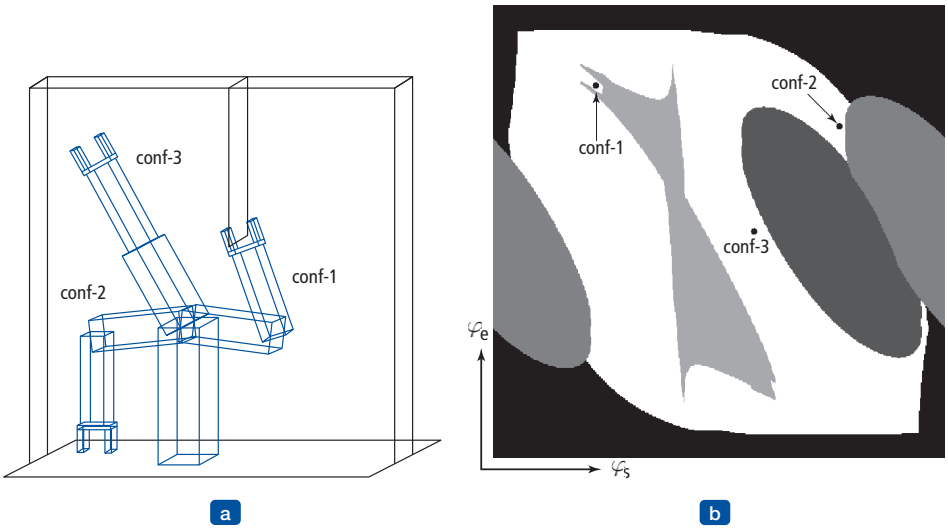


Abbildung 25.15: Drei Roboterkonfigurationen, gezeigt in Arbeitsraum und Konfigurationsraum.

Selbst wenn der Arbeitsraum des Roboters durch flache Polygone dargestellt wird, kann der Umriss des Freiraumes sehr kompliziert sein. In der Praxis *probiert* man deshalb einen Konfigurationsraum normalerweise nur aus, anstatt ihn explizit zu konstruieren. Ein Planer kann eine Konfiguration erzeugen und dann testen, ob sie im Freiraum liegt, indem er die Roboter-Kinematik anwendet und dann auf Kollisionen in den Arbeitsraum-Koordinaten überprüft.

25.4.2 Zellzerlegungsmethoden

Der erste Ansatz für die Pfadplanung verwendet die **Zellzerlegung**, d.h., er zerlegt den freien Raum in eine endliche Anzahl zusammenhängender Bereiche, die sogenannten Zellen. Diese Bereiche haben die wichtige Eigenschaft, dass das Pfadplanungsproblem innerhalb eines einzelnen Bereiches durch einfache Mittel gelöst werden kann (z.B. durch die Bewegung entlang einer geraden Linie). Das Pfadplanungsproblem wird dann zu einem diskreten Graphensuchproblem, ähnlich den in *Kapitel 3* eingeführten Suchproblemen.

Die einfachste Zellzerlegung besteht aus einem regelmäßigen Raster. ▶ Abbildung 25.16(a) zeigt eine quadratische Rasterzerlegung des Raumes sowie einen Lösungspfad, der optimal für diese Rastergröße ist. Eine Graustufenschattierung kennzeichnet den *Wert* jeder Rasterzelle im freien Raum – d.h. die Kosten des kürzesten Pfades von dieser Zelle zum Ziel. (Diese Werte können durch eine deterministische Form des VALUE-ITERATION-Algorithmus gemäß Abbildung 17.4 berechnet werden.) ▶ Abbildung 25.16(b) zeigt die entsprechende Bewegungsbahn im Arbeitsraum für den Arm. Selbstverständlich können wir einen kürzesten Pfad auch mit dem A*-Algorithmus ermitteln.

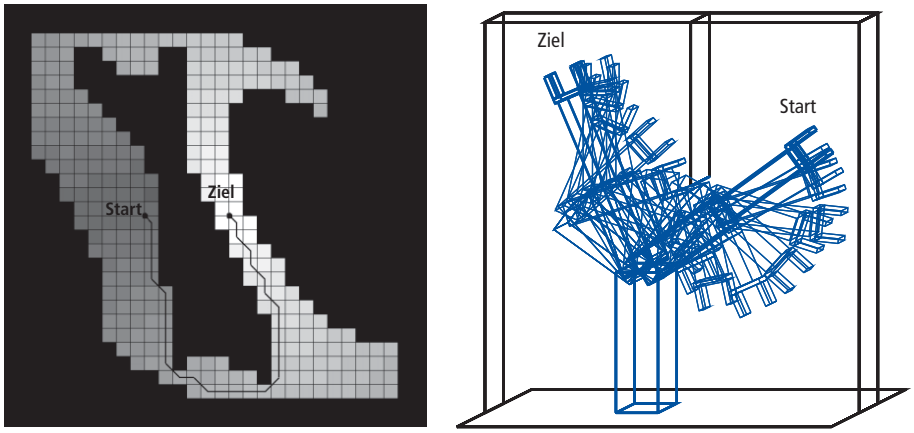


Abbildung 25.16: (a) Wertfunktion und Pfad für eine diskrete Rasterzellannäherung des Konfigurationsraumes. (b) Derselbe Pfad, dargestellt mit den Arbeitsraum-Koordinaten. Beachten Sie, wie der Roboter seinen Ellenbogen beugt, um Kollisionen mit dem vertikalen Hindernis zu vermeiden.

Eine derartige Zerlegung hat den Vorteil, dass sie extrem einfach zu implementieren ist; sie leidet aber auch unter drei Einschränkungen. Erstens ist sie nur für Konfigurationsräume mit wenigen Dimensionen zu bewältigen, weil die Anzahl der Rasterzellen mit der Anzahl der Dimensionen d exponentiell wächst. Kommt Ihnen das bekannt vor? Das ist der Fluch der Dimensionalität. Zweitens gibt es das Problem, was mit Zellen zu tun ist, die „gemischt" sind – d.h. weder vollständig innerhalb des Freiraumes noch vollständig innerhalb des belegten Raumes liegen. Ein Lösungspfad, der eine solche Zelle beinhaltet, ist eventuell keine wirkliche Lösung, weil es keine Möglichkeit gibt, die Zelle in der gewünschten Richtung in einer geraden Linie zu kreuzen. Das würde den Pfadplaner *unzuverlässig* machen. Wenn wir andererseits darauf bestehen, dass nur vollständig freie Zellen verwendet werden, ist der Planer *unvollständig*, weil es vorkommen kann, dass die einzigen Pfade zum Ziel durch gemischte Zellen verlaufen – insbesondere, wenn die Zellgröße mit der Größe der Durchgänge und der lichten Abstände vergleichbar ist. Und drittens ist jeder Pfad durch einen diskretisierten Zustandsraum nicht glatt. Prinzipiell lässt sich kaum garantieren, dass eine glatte Lösung neben dem diskreten Pfad existiert. Ein Roboter kann also die durch diese Zerlegung gefundene Lösung möglicherweise gar nicht ausführen.

Die Methoden zur Zellzerlegung lassen sich in verschiedener Weise verbessern, um einige dieser Probleme zu lindern. Der erste Ansatz erlaubt eine *weitere Unterteilung* der gemischten Zellen – und verwendet möglicherweise Zellen mit der halben ursprünglichen Größe. Das kann rekursiv fortgesetzt werden, bis ein Pfad gefunden ist, der vollständig innerhalb von freien Zellen liegt. (Natürlich funktioniert diese Methode nur, wenn es eine Möglichkeit gibt, zu entscheiden, ob eine bestimmte Zelle eine freie Zelle ist, was nur dann einfach ist, wenn die Grenzen des Konfigurationsraumes relativ einfache mathematische Beschreibungen haben.) Diese Methode ist vollständig, vorausgesetzt, es gibt eine Begrenzung zu dem kleinsten Durchgang, durch den eine Lösung verlaufen muss. Obwohl sie sich hauptsächlich auf den rechentechnischen Aufwand für die komplizierten Bereiche innerhalb des Konfigurationsraumes konzentriert, kann sie dennoch nicht gut auf hochdimensionale Probleme skaliert werden, weil bei jeder rekursiven Auf-

teilung einer Zelle 2^d kleinere Zellen entstehen. Eine zweite Möglichkeit, einen vollständigen Algorithmus zu erhalten, ist die Beibehaltung einer **exakten Zellzerlegung** des freien Raumes. Diese Methode muss es erlauben, dass Zellen unregelmäßige Formen annehmen, wo sie auf die Grenzen des Freiraumes stoßen, aber die Umrisse müssen dennoch „einfach“ in der Hinsicht sein, dass ein Durchgang durch eine freie Zelle einfach zu berechnen ist. Diese Technik bedingt eine relativ komplexe Geometrie; deshalb wollen wir sie hier nicht weiter verfolgen.

Untersucht man den in Abbildung 25.16(a) gezeigten Lösungspfad, zeigt sich eine zusätzliche Schwierigkeit, die gelöst werden muss. Der Pfad enthält beliebig scharfe Ecken; ein Roboter, der sich mit einer endlichen Geschwindigkeit bewegt, kann einen solchen Pfad nicht verfolgen. Dieses Problem lässt sich lösen, indem für jede Rasterzelle bestimmte stetige Werte gespeichert werden. Betrachten Sie einen Algorithmus, der für jede Rasterzelle den exakten, stetigen Zustand speichert, der erreicht wurde mit der Zelle, die in der Suche zuerst erweitert wurde. Nehmen Sie weiterhin an, dass wir beim Weiterleiten von Informationen an nahegelegene Rasterzellen diesen stetigen Zustand als Basis verwenden und das stetige Roboterbewegungsmodell anwenden, um zu nahegelegenen Zellen zu springen. Dadurch können wir jetzt garantieren, dass die resultierende Bahn glatt ist und tatsächlich vom Roboter ausgeführt werden kann. **Hybrid A*** ist ein solcher Algorithmus, der dies implementiert.

25.4.3 Modifizierte Kostenfunktionen

Wie Abbildung 25.16 zeigt, geht der Pfad sehr nahe am Hindernis vorbei. Jeder, der schon einmal ein Auto gefahren hat, weiß, dass eine Parklücke mit nur einem Millimeter Platz an jeder Seite nicht wirklich ein Parkplatz ist; aus demselben Grund bevorzugen wir Lösungspfade, die unempfindlich gegenüber kleinen Bewegungsfehlern sind. Dieses Problem lässt sich mithilfe eines **Potentialfeldes** lösen. Bei einem Potentialfeld handelt es sich um eine über dem Zustandsraum definierte Funktion, deren Wert mit dem Abstand zum nächstgelegenen Hindernis wächst. ▶ Abbildung 25.17(a) zeigt ein solches Potentialfeld – je dunkler ein Konfigurationszustand ist, desto näher liegt er an einem Hindernis.

Abbildung 25.17: (a) Ein abstoßendes Potentialfeld schiebt den Roboter von Hindernissen weg.
(b) Pfad, der durch eine gleichzeitige Minimierung der Pfadlänge und des Potentials gefunden wurde.

Das Potentialfeld kann als zusätzlicher Kostenterm in der Berechnung des kürzesten Pfades verwendet werden. Dies führt zu einer interessanten Abwägung. Einerseits versucht der Roboter, die Pfadlänge zum Ziel zu minimieren. Andererseits versucht er, durch Minimierung der Potentialfunktion von den Hindernissen weg zu bleiben. Mit der geeigneten Gewichtung, die die beiden Ziele ausgleicht, kann ein resultierender Pfad wie der in ▶ Abbildung 25.17(b) aussehen. Diese Abbildung zeigt auch die Wertfunktion, die von der kombinierten Kostenfunktion abgeleitet ist, ebenfalls durch Wert-Iteration berechnet. Offensichtlich ist der resultierende Pfad länger, aber auch sicherer.

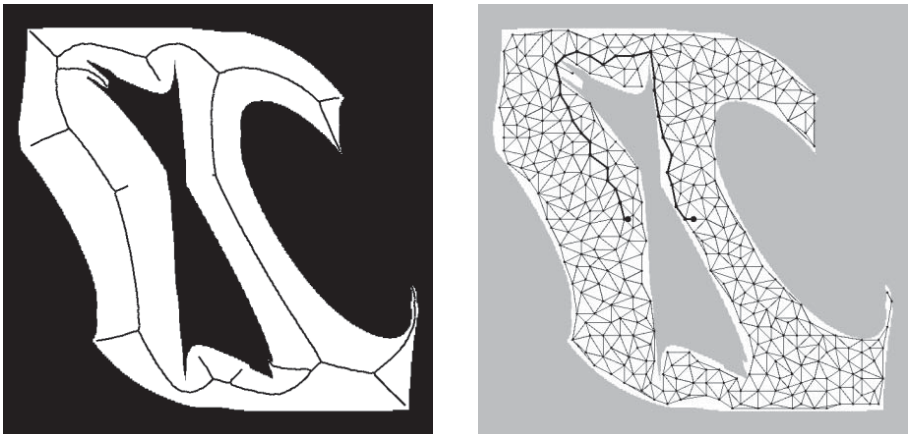
Es gibt viele andere Möglichkeiten, um die Kostenfunktion zu modifizieren. Zum Beispiel kann es wünschenswert sein, die Steuerungsparameter über die Zeit zu glätten. So ist für das Autofahren eine glatte Straße besser geeignet als eine Buckelpiste. Im Allgemeinen ist es nicht leicht, derartige Einschränkungen höherer Ordnung im Planungsprozess unterzubringen, sofern wir nicht den letzten Lenkbefehl in die Zustandsbeschreibung aufnehmen. Allerdings lässt sich die resultierende Bahn oftmals relativ leicht nach der Planung durch Methoden der konjugierten Gradienten glätten. Derartiges Glätten im Anschluss an die Planung ist in vielen praktischen Anwendungen ein wichtiger Schritt.

25.4.4 Skelettierungsmethoden

Die zweite große Familie von Pfadplanungs-Algorithmen basiert auf dem Konzept der **Skelettierung**. Diese Algorithmen reduzieren den freien Raum des Roboters auf eine eindimensionale Darstellung, für die das Planungsproblem einfacher ist. Diese Repräsentation mit einer geringeren Dimension wird auch als **Skelett** des Konfigurationsraumes bezeichnet.

▶ Abbildung 25.18 zeigt ein Beispiel für die Skelettierung: Es handelt sich dabei um ein **Voronoi-Diagramm** des Freiraumes – die Menge aller Punkte, die äquidistant zu zwei oder mehr Hindernissen sind. Um eine Pfadplanung mit einem Voronoi-Diagramm durchzuführen, ändert der Roboter zuerst seine aktuelle Konfiguration auf einen Punkt auf dem Voronoi-Diagramm. Es lässt sich leicht zeigen, dass dies immer durch eine geradlinige Bewegung im Konfigurationsraum realisiert werden kann. Zweitens folgt der Roboter dem Voronoi-Graphen, bis er den Punkt erreicht, der am nächsten zur Zielkonfiguration liegt. Schließlich verlässt der Roboter den Voronoi-Graphen und bewegt sich zum Ziel. Auch dieser letzte Schritt bedeutet eine geradlinige Bewegung im Konfigurationsraum.

Auf diese Weise wird das ursprüngliche Pfadplanungsproblem darauf reduziert, einen Pfad im Voronoi-Diagramm zu finden, der in der Regel eindimensional ist (bis auf einige nichtgenerische Fälle) und endlich viele Punkte hat, wo sich drei oder mehr eindimensionale Kurven schneiden. Damit ist die Suche des kürzesten Pfades entlang des Voronoi-Graphen ein diskretes Graphensuchproblem der Art, wie sie in den *Kapiteln 3* und *4* beschrieben wurde. Die Verfolgung des Voronoi-Graphen verschafft uns vielleicht nicht den kürzesten Pfad, aber die resultierenden Pfade tendieren dazu, den lichten Durchgang zu maximieren. Nachteil der Voronoi-Graphentechniken ist, dass sie sehr schwierig auf höherdimensionale Konfigurationsräume anzuwenden sind und dass sie häufig unnötig lange Umwege einführen, wenn der Konfigurationsraum weit offen ist. Darüber hinaus kann die Berechnung des Voronoi-Diagramms schwierig sein, insbesondere in einem Konfigurationsraum, wo die Umrisse der Hindernisse komplex sind.



a b

Abbildung 25.18: (a) Das Voronoi-Diagramm ist die Menge der Punkte, die äquidistant zu zwei oder mehr Hindernissen im Konfigurationsraum sind. (b) Eine probabilistische Straßenkarte, zusammengesetzt aus 400 zufällig gewählten Punkten im freien Raum.

Eine Alternative zu den Voronoi-Diagrammen ist die **probabilistische Straßenkarte**, ein Skelettierungsansatz, der mehrere mögliche Routen anbietet und deshalb besser mit weit offenen Räumen zurechtkommt. Abbildung 25.18(b) zeigt ein Beispiel für eine probabilistische Straßenkarte. Der Graph wird erzeugt, indem zufällig eine große Anzahl an Konfigurationen generiert wird und diejenigen verworfen werden, die nicht im freien Raum liegen. Zwei Knoten werden durch einen Bogen verbunden, wenn es „einfach“ ist, einen Knoten von dem anderen aus zu erreichen – beispielsweise durch eine gerade Linie im freien Raum. Das Ergebnis ist ein zufälliger Graph im freien Raum des Roboters. Wenn wir diesem Graphen die Start- und Zielkonfigurationen des Roboters hinzufügen, wird die Pfadplanung zu einer diskreten Graphensuche. Theoretisch ist dieser Ansatz unvollständig, weil eine schlechte Wahl der Zufallspunkte dazu führt, dass wir überhaupt keine Pfade vom Start zum Ziel haben. Es ist möglich, die Wahrscheinlichkeit eines Fehlers im Hinblick auf die Anzahl der Punkte, die erzeugt werden, sowie auf bestimmte geometrische Eigenschaften des Konfigurationsraumes einzugrenzen. Außerdem ist es möglich, die Erzeugung von Beispielpunkten auf die Bereiche zu konzentrieren, wo eine partielle Suche darauf hinweist, dass vielleicht ein guter Pfad gefunden wird, indem bidirektional von den Start- und Zielpositionen aus gearbeitet wird. Mit diesen Verbesserungen kann die probabilistische Straßenkartenplanung besser auf hochdimensionale Konfigurationsräume skaliert werden als die meisten alternativen Techniken für die Pfadplanung.

25.5 Planung unsicherer Bewegungen

Keiner der bisher beschriebenen Algorithmen für die Planung der Roboterbewegungen hat sich um ein wesentliches Merkmal von Robotikproblemen gekümmert: *Unsicherheit*. In der Robotik entsteht die Unsicherheit aus der partiellen Beobachtbarkeit der Umgebung sowie aus stochastischen (oder nicht modellierten) Effekten der Aktionen des Roboters. Fehler können auch aus der Verwendung von Annäherungsalgorithmen entstehen, wie beispielsweise dem Partikelfilter, der dem Roboter keine genauen Belief States bereitstellt, selbst wenn die stochastische Natur der Umgebung perfekt modelliert wird.

Die meisten der heutigen Roboter verwenden deterministische Algorithmen für die Entscheidungsfindung, wie beispielsweise die im vorherigen Abschnitt beschriebenen Algorithmen für die Pfadplanung. Dazu ist es gängige Praxis, den **wahrscheinlichsten Zustand** aus der durch den Lokalisierungsalgorithmus erzeugten Zustandsverteilung zu extrahieren. Der Vorteil bei diesem Ansatz ist rein rechentechnisch. Die Planung von Pfaden durch den Konfigurationsraum ist schon eine Herausforderung; es wäre noch schlimmer, wenn wir mit vollständigen Wahrscheinlichkeitsverteilungen über Zuständen arbeiten müssten. Diese Art der Ignoranz gegenüber der Unsicherheit funktioniert, wenn die Unsicherheit klein ist. Ändert sich das Umgebungsmodell im Lauf der Zeit, weil Sensormessungen einbezogen wurden, planen viele Roboter während der Planausführung die Pfade online. Dies ist die Technik des **Online-Neuplanens**, die *Abschnitt 11.3.3* beschrieben hat.

Leider ist es nicht immer möglich, die Unsicherheit zu ignorieren. Bei einigen Problemen ist die Unsicherheit des Roboters einfach zu massiv. Wie können wir beispielsweise einen deterministischen Pfadplaner verwenden, um einen mobilen Roboter zu steuern, der keine Ahnung hat, wo er sich befindet? Im Allgemeinen ist die resultierende Steuerung suboptimal, wenn der tatsächliche Zustand des Roboters nicht derjenige ist, der durch die ML-Regel identifiziert wurde. Abhängig von der Größe des Fehlers kann dies zu allen möglichen unerwünschten Effekten führen, wie beispielsweise Kollisionen mit Hindernissen.

Der Bereich der Robotik hat eine Vielzahl von Techniken für den Umgang mit der Unsicherheit übernommen. Einige davon sind von den in *Kapitel 17* für die Entscheidungsfindung unter Unsicherheit vorgestellten Algorithmen abgeleitet. Wenn der Roboter der Unsicherheit nur in seinen Zustandsübergängen gegenübersteht, sein Zustand aber vollständig beobachtbar ist, wird das Problem am besten als **MDP** (Markov Decision Process) modelliert. Die Lösung für einen MDP ist eine optimale **Strategie**, die dem Roboter mitteilt, was in jedem möglichen Zustand zu tun ist. Auf diese Weise kann er alle Arten von Bewegungsfehlern verarbeiten, während eine Einzelpfadlösung aus einem deterministischen Planer weit weniger robust wäre. In der Robotik werden Strategien häufig als **Navigationsfunktionen** bezeichnet. Die in Abbildung 25.16(a) gezeigte Wertfunktion kann in eine solche Navigationsfunktion umgewandelt werden, indem man einfach dem Gradienten folgt.

Wie in *Kapitel 17* macht die partielle Beobachtbarkeit das Problem sehr viel schwieriger. Das resultierende Robotersteuerungsproblem ist ein partiell beobachtbares MDP oder POMPD (Partial Observable MDP). In solchen Situationen behält der Roboter normalerweise einen internen Belief State bei, ähnlich wie in *Abschnitt 25.3* beschrieben. Die Lösung für einen POMPD ist eine Strategie, die über den Belief State des Roboters definiert ist. Anders ausgedrückt, die Eingabe für die Strategie ist eine vollständige Wahrscheinlichkeitsverteilung. Auf diese Weise ist es dem Roboter möglich, seine Entscheidung nicht nur auf seinem Wissen zu basieren, sondern auch darauf, was er nicht weiß. Ist der Roboter beispielsweise hinsichtlich einer kritischen Zustandsvariablen unsicher, kann er rational eine **Aktion zur Informationsermittlung** aufrufen. Das ist im MDP-Framework nicht möglich, weil MDPs eine vollständige Beobachtbarkeit voraussetzen. Leider können Techniken, die POMPDs exakt lösen, in der Robotik nicht angewendet werden – es gibt keine bekannten Techniken für hochdimensionale stetige Räume. Diskretisierung erzeugt POMDPs, die für eine sinnvolle Verarbeitung bei weitem zu groß sind. Um dem abzuhelfen, kann man die Minimierung der Unsicherheit zu einem Steuerungsziel machen. Beispielsweise fordert die **Heuristik der Küstennavigation**, dass

der Roboter in der Nähe bekannter Orientierungspunkte bleibt, um seine Unsicherheit zu verringern. Ein anderer Ansatz wendet Varianten der probabilistischen Planungsmethode für Straßenkarten auf die Darstellung des Belief State an. Derartige Methoden lassen sich in der Regel besser auf große diskrete POMDPs skalieren.

25.5.1 Robuste Methoden

Unsicherheit kann auch durch Methoden der sogenannten **robusten Kontrolle** (siehe *Abschnitt 21.2.2*) statt mithilfe von probabilistischen Methoden verarbeitet werden. Eine robuste Methode geht davon aus, dass in jedem Aspekt eines Problems eine *begrenzte* Größe von Unsicherheit vorhanden ist, weist jedoch den Werten des erlaubten Intervalls keine Wahrscheinlichkeiten zu. Eine robuste Lösung funktioniert, egal welche Werte auftreten, sofern sie innerhalb des angenommenen Intervalls liegen. Eine Extremform einer robusten Methode ist der Ansatz der **konformanten Planung**, die wir in *Kapitel 11* vorgestellt haben – sie produziert Pläne, die ohne jede Zustandsinformation funktionieren.

Wir betrachten hier eine robuste Methode, die für die **Feinbewegungsplanung** (FMP, Fine-Motion Planning) bei Montagerobotern verwendet wird. Die Feinbewegungsplanung beinhaltet die Bewegung eines Roboterarms sehr nah an einem statischen Umgebungsobjekt. Die größte Schwierigkeit bei der Feinbewegungsplanung ist, dass die erforderlichen Bewegungen und die relevanten Merkmale der Umgebung sehr gering sind. Bei so kleinen Dimensionen kann der Roboter seine Position nicht genau messen oder steuern und er kann auch hinsichtlich des Umrisses der eigentlichen Umgebung unsicher sein; wir gehen davon aus, dass diese Unsicherheiten begrenzt sind. Die Lösungen für FMP-Probleme sind in der Regel bedingte Planer oder Strategien, die während der Ausführung ein Sensor-Feedback verwenden und garantiert in allen Situationen funktionieren, die mit den angenommenen Unsicherheitsgrenzen konsistent sind.

Ein Feinbewegungsplan besteht aus einer Folge **überwachter Bewegungen**. Jede überwachte Bewegung besteht aus (1) einem Bewegungsbefehl und (2) einer Terminierungsbedingung, wobei es sich um ein Prädikat der Sensorwerte des Roboters handelt, und gibt *true* zurück, um das Ende der überwachten Bewegungen anzuzeigen. Die Bewegungsbefehle sind in der Regel **konforme Bewegungen**, die dem Effektor ein Gleiten erlauben, wenn der Bewegungsbefehl eine Kollision mit einem Hindernis verursachen würde. Als Beispiel dafür zeigt ▶ Abbildung 25.19 einen zweidimensionalen Konfigurationsraum mit einem engen vertikalen Loch. Dabei könnte es sich um den Konfigurationsraum für das Einfügen eines rechteckigen Bolzens in ein etwas größeres Loch oder eines Autoschlüssels in das Zündschloss handeln. Die Bewegungsbefehle sind konstante Geschwindigkeiten. Die Terminierungsbedingungen sind Kontakte mit einer Oberfläche. Um die Unsicherheit in der Steuerung zu modellieren, nehmen wir an, dass sich der Roboter nicht in die angegebene Richtung bewegt, sondern dass die tatsächliche Bewegung des Roboters in dem Kegel C_v über ihm liegt. Die Abbildung zeigt, was passieren würde, wenn wir eine Geschwindigkeit unmittelbar ab dem Startbereich s angegeben hätten. Aufgrund der Unsicherheit in der Geschwindigkeit könnte sich der Roboter an jede beliebige Position in der kegelförmigen Hülle bewegen, möglicherweise das Loch treffen, aber sehr wahrscheinlich irgendwo seitlich davon landen. Weil der Roboter dann nicht wüsste, auf welcher Seite des Lochs er sich befindet, wüsste er nicht, in welche Richtung er sich bewegen soll.

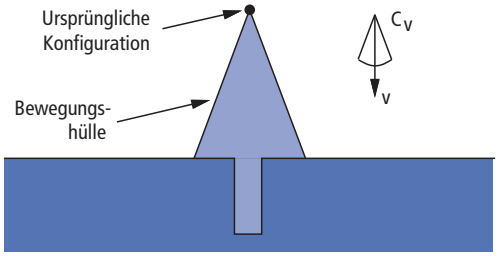


Abbildung 25.19: Eine zweidimensionale Umgebung, Geschwindigkeits-Unsicherheitskegel und Hülle möglicher Roboterbewegungen. Die beabsichtigte Geschwindigkeit ist v, aber bei der vorliegenden Unsicherheit könnte sich die tatsächliche Geschwindigkeit auch irgendwo innerhalb von C_v befinden, was zu einer endgültigen Konfiguration führen würde, die irgendwo in der Bewegungshülle liegt, wir also nicht wüssten, ob wir das Loch getroffen haben oder nicht.

▶ Abbildung 25.20 und ▶ Abbildung 25.21 zeigen eine sinnvollere Strategie. In Abbildung 25.20 bewegt sich der Roboter bewusst auf eine Seite des Loches. Der Bewegungsbefehl ist in der Abbildung gezeigt und der Terminierungstest besteht in einem Kontakt mit einer beliebigen Oberfläche. In Abbildung 25.21 wird ein Bewegungsbefehl gegeben, der den Roboter veranlasst, über die Oberfläche und in das Loch zu gleiten. Weil alle möglichen Geschwindigkeiten in der Bewegungshülle rechts liegen, gleitet der Roboter nach rechts, wenn er in Kontakt mit einer horizontalen Oberfläche gerät. Er gleitet an der rechten vertikalen Kante des Loches nach unten, wenn er es berührt, weil alle möglichen Geschwindigkeiten relativ zu einer vertikalen Oberfläche nach unten gehen. Er bewegt sich weiter, bis er den Boden des Loches erreicht hat, weil dies seine Terminierungsbedingung ist. Trotz der Steuerungsunsicherheit terminieren alle möglichen Bewegungsverläufe des Roboters, wenn sie mit dem Boden des Loches in Kontakt kommen – d.h. sofern keine Oberflächenunregelmäßigkeiten bewirken, dass der Roboter an einer Stelle stecken bleibt.

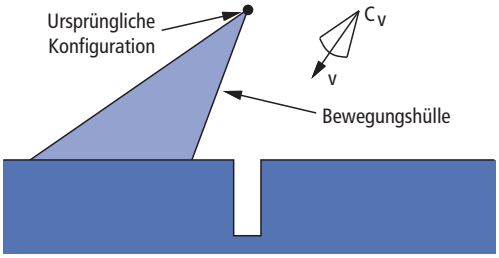


Abbildung 25.20: Der erste Bewegungsbefehl und die resultierende Hülle möglicher Roboterbewegungen. Egal, wie groß der Fehler ist, wir wissen, dass sich die endgültige Konfiguration links vom Loch befindet.

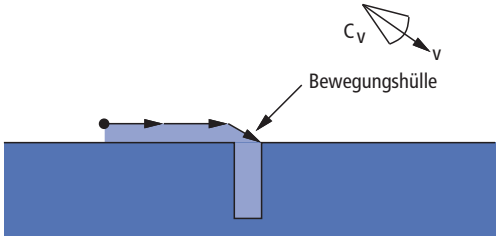


Abbildung 25.21: Der zweite Bewegungsbefehl und die Hülle möglicher Bewegungen. Selbst mit einem Fehler gelangen wir irgendwann in das Loch.

Man kann sich vorstellen, dass die *Erstellung* von Feinbewegungsplänen keine einfache Angelegenheit ist. Tatsächlich ist sie sehr viel schwieriger als die Planung mit genauen Bewegungen. Man kann entweder eine feste Anzahl diskreter Werte für jede Bewegung auswählen oder die Umgebungsgeometrie nutzen, um Richtungen auszuwählen, die qualitativ unterschiedliches Verhalten verursachen. Ein Feinbewegungsplaner verwendet als Eingabe die Konfigurationsraumbeschreibung, den Winkel des Geschwindigkeits-Unsicherheitskegels sowie eine Spezifikation dessen, welche Sensoreingabe für die Terminierung möglich ist (in diesem Fall Oberflächenkontakt). Er sollte einen mehrere Schritte umfassenden bedingten Plan oder eine Strategie produzieren, die garantiert erfolgreich ist, wenn ein solcher Plan existiert.

Unser Beispiel geht davon aus, dass der Plan ein genaues Modell der Umgebung besitzt. Es ist aber möglich, einen begrenzten Fehler in diesem Modell zu erlauben: Wenn der Fehler unter Verwendung der Parameter beschrieben werden kann, lassen sich diese Parameter dem Konfigurationsraum als Freiheitsgrade hinzufügen. Wären im letzten Beispiel Tiefe und Breite des Loches unsicher, könnten wir sie dem Konfigurationsraum als zwei Freiheitsgrade hinzufügen. Es ist nicht möglich, den Roboter in diese Richtungen im Konfigurationsraum zu bewegen oder seine Position direkt zu erkennen. Aber beide Einschränkungen können berücksichtigt werden, wenn wir dieses Problem als FMP-Problem beschreiben, indem wir eine geeignete Spezifikation für Steuerungs- und Sensorunsicherheiten angeben. Damit erhalten wir ein komplexes, vierdimensionales Planungssystem, aber es können genau dieselben Planungstechniken angewendet werden. Beachten Sie, dass dieser robuste Ansatz im Gegensatz zu den in *Kapitel 17* beschriebenen entscheidungstheoretischen Methoden Pläne erzeugt, die auf den ungünstigsten Fall ausgelegt sind, anstatt die erwartete Qualität des Planes zu maximieren. Pläne für den ungünstigsten Fall sind in entscheidungstheoretischer Hinsicht nur dann optimal, wenn der Fehler bei der Ausführung wesentlich größer ist als irgendwelche anderen Kosten, die für die Ausführung anfallen.

25.6 Bewegung

Bisher haben wir nur darüber gesprochen, wie wir Bewegungen *planen*, aber nicht, wie wir die Bewegungen *ausführen*. Unsere Pläne – insbesondere diejenigen, die durch deterministische Pfadplaner erzeugt wurden – gehen davon aus, dass der Roboter einfach jedem Pfad folgen kann, den der Algorithmus produziert. In der realen Welt ist dies natürlich nicht der Fall. Roboter haben eine gewisse Trägheit und können nicht beliebige Pfade verfolgen, außer bei beliebig geringen Geschwindigkeiten. Größtenteils wird ein Roboter Kräfte anwenden, anstatt Positionen zu bestimmen. Dieser Abschnitt beschreibt Methoden für die Berechnung dieser Kräfte.

25.6.1 Dynamik und Steuerung

Abschnitt 25.2 hat das Konzept des **dynamischen Zustandes** eingeführt, der den kinematischen Zustand eines Roboters durch seine Geschwindigkeit erweitert. Beispielsweise erfasst der dynamische Zustand neben dem Winkel eines Robotergelenks auch die Änderungsgeschwindigkeit des Winkels und möglicherweise sogar seine momentane Beschleunigung. Das Übergangsmodell für eine dynamische Zustandsrepräsentation beinhaltet die Wirkung von Kräften auf diese Änderungsgeschwindigkeit. Solche Modelle werden in der Regel mithilfe von **Differentialgleichungen** ausgedrückt,

wobei es sich um Gleichungen handelt, die eine Quantität (z.B. einen kinematischen Zustand) mit der Änderung der Quantität über die Zeit (z.B. Geschwindigkeit) in Verbindung bringen. Im Prinzip könnten wir die Roboterbewegungen auch mit dynamischen Modellen statt unseren kinematischen Modellen planen. Eine solche Methode würde zu einer überlegenen Roboterleistung führen, wenn wir die Pläne erzeugen könnten. Der dynamische Zustand hat jedoch eine höhere Dimension als der kinematische Raum und der Fluch der Dimensionalität würde dazu führen, dass viele Algorithmen der Bewegungsplanung nicht mehr – außer auf die einfachsten Roboter – anwendbar sind. Aus diesem Grund verwenden praktische Robotersysteme häufig einfachere kinematische Pfadplaner.

Um die Einschränkungen kinematischer Pläne zu kompensieren, verwendet man häufig als separaten Mechanismus einen **Regler** (engl. **Controller**), der den Roboter in der Spur hält. Regler sind Techniken für die Erstellung von Robotersteuerungen in Echtzeit, wobei ein Feedback aus der Umgebung verwendet wird, um ein Steuerungsziel zu erreichen. Besteht das Ziel darin, den Roboter auf einem zuvor geplanten Pfad zu halten, spricht man häufig von einem **Referenz-Regler** und bezeichnet den Pfad als **Referenzpfad**. Regler, die eine globale Kostenfunktion optimieren, werden auch als **optimale Regler** bezeichnet. Optimale Strategien für MDPs sind letztlich optimale Regler.

Oberflächlich betrachtet scheint das Problem, einen Roboter auf einem vorgegebenen Pfad zu halten, relativ einfach zu sein. In der Praxis hat jedoch selbst dieses scheinbar einfache Problem einige Fallstricke. ▶ Abbildung 25.22(a) veranschaulicht, was schief gehen kann. Die Abbildung zeigt den Pfad eines Roboters, der versucht, einem kinematischen Pfad zu folgen. Immer wenn eine Abweichung auftritt – egal ob durch Rauschen oder durch eine Beschränkung der Kräfte, die der Roboter anwenden kann –, setzt der Roboter eine entgegengesetzte Kraft ein, deren Größe proportional zu dieser Abweichung ist. Das erscheint auf den ersten Blick plausibel zu sein, weil Abweichungen durch eine Gegenkraft kompensiert werden sollen, um den Roboter in der Spur zu halten. Wie jedoch aus Abbildung 25.22(a) hervorgeht, kommt es durch unseren Regler dazu, dass der Roboter heftig vibriert. Die Vibration entsteht durch die natürliche Trägheit des Roboterarmes: Sobald der Roboter auf seine Referenzposition zurückgefahren wurde, schwingt er über das Ziel hinaus, was einen symmetrischen Fehler mit entgegengesetztem Vorzeichen herbeiführt. Ein solches Überschwingen kann sich über den gesamten Bewegungsverlauf fortsetzen und die resultierende Roboterbewegung ist alles andere als wünschenswert.

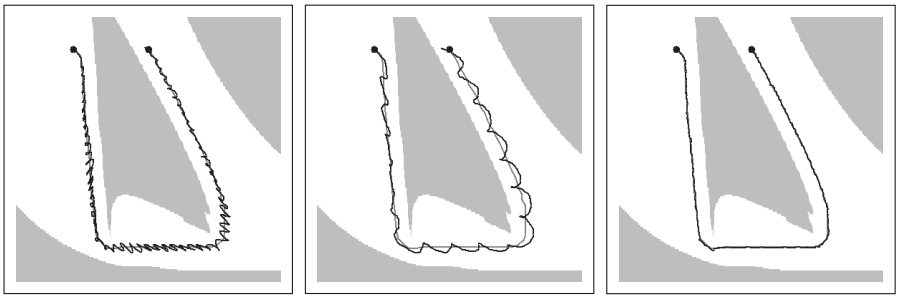


Abbildung 25.22: Robotersteuerung unter Verwendung (a) einer proportionalen Regelung mit einem Verstärkungsfaktor von 1,0, (b) einer proportionalen Regelung mit einem Verstärkungsfaktor von 0,1 und (c) einer PD (Proportional-Differenzial)-Regelung mit Verstärkungsfaktoren von 0,3 für die proportionale Komponente und 0,8 für die differenzielle Komponente. In allen Fällen versucht der Roboterarm, dem grau dargestellten Pfad zu folgen.

Bevor wir einen besseren Regler definieren können, wollen wir formal beschreiben, was schiefgelaufen ist. Regler, die Kräfte mit negativer Proportion zum beobachteten Fehler bereitstellen, werden auch als **P-Regler** bezeichnet. Der Buchstabe P steht für *proportional*, d.h., die tatsächliche Regelung ist proportional zu dem Fehler des Roboter-Manipulators. Formal ausgedrückt, sei $y(t)$ der Referenzpfad, parametrisiert durch den Zeitindex t. Das Steuersignal a_t, das durch einen P-Regler erzeugt wird, hat die folgende Form:

$$a_t = K_P(y(t) - x_t).$$

Hier ist x_t der Zustand des Roboters zur Zeit t und die Konstante K_P ist ein sogenannter **Verstärkungsparameter** des Reglers (dessen Wert als Verstärkungsfaktor bezeichnet wird); K_P bestimmt, wie stark der Regler Abweichungen zwischen dem tatsächlichen Zustand x_t und dem gewünschten Zustand $y(t)$ korrigiert. In unserem Beispiel gilt $K_P = 1$. Auf den ersten Blick denkt man vielleicht, dass die Auswahl eines kleineren Wertes für K_P das Problem beseitigt. Leider ist dies nicht der Fall. ▶ Abbildung 25.22(b) zeigt einen Bewegungsverlauf für $K_P = 0{,}1$, der immer noch ein schwingendes Verhalten aufweist. Geringere Werte für den Verstärkungsparameter verlangsamen einfach die Schwingung, lösen aber das Problem nicht. Bei fehlender Reibung folgt der P-Regler im Wesentlichen einem Federgesetz; er schwingt also unbestimmt lange um eine feste Zielposition.

Traditionell ordnet man Probleme dieser Art in das Fachgebiet der **Regelungstheorie** ein, ein Bereich wachsender Bedeutung für Forscher in der KI. Jahrzehntelange Forschungen auf diesem Gebiet haben zu einer großen Anzahl von Reglern geführt, die dem einfachen Regelungsgesetz, das wir oben gezeigt haben, überlegen sind. Insbesondere sagt man, ein Referenz-Regler ist **stabil**, wenn kleine Störungen zu einem begrenzten Fehler zwischen dem Roboter und dem Referenzsignal führen. Man sagt, er ist **streng stabil**, wenn er in der Lage ist, nach solchen Störungen zu seinem Referenzpfad zurückzukehren und dort zu bleiben. Unser P-Regler scheint stabil zu sein, aber nicht streng stabil, weil er es nicht schafft, irgendwo in der Nähe seiner Referenz-Bewegungsbahn zu bleiben.

Der einfachste Regler, der strenge Stabilität in unserer Domäne erzielt, wird als **PD-Regler** bezeichnet. Der Buchstabe P steht wieder für *proportional* und D für *Ableitung* (Derivation). PD-Regler werden durch die folgende Gleichung beschrieben:

$$a_t = K_P\big(y(t) - x_t\big) + K_D \frac{\partial\big(y(t) - x_t\big)}{\partial t}. \tag{25.2}$$

Wie diese Gleichung besagt, erweitern PD-Regler die P-Regler um eine Differentialkomponente, die dem Wert von a_t einen Term hinzufügt, der proportional zur ersten Ableitung des Fehlers $y(t) - x_t$ über die Zeit ist. Welche Wirkung hat ein solcher Term? Im Allgemeinen dämpft ein Ableitungsterm das System, das gesteuert werden soll. Betrachten Sie dafür eine Situation, wo sich der Fehler $(y(t) - x_t)$ über die Zeit schnell ändert, wie es für unseren oben gezeigten P-Regler der Fall ist. Die Ableitung dieses Fehlers wirkt dem Proportionalterm entgegen, der die allgemeine Reaktion auf die Störungen reduziert. Bleibt jedoch derselbe Fehler erhalten und ändert sich nicht, verschwindet die Ableitung und der proportionale Term dominiert die Auswahl der Steuerung.

▶ Abbildung 25.22(c) zeigt das Ergebnis der Anwendung dieses PD-Reglers auf unseren Roboterarm, wobei wir als Verstärkungsparameter $K_P = 0{,}3$ und $K_D = 0{,}8$ verwenden. Der resultierende Pfad ist sehr viel glatter und weist keine offensichtlichen Schwingungen auf.

Allerdings besitzen PD-Regler auch Fehlermodi. Insbesondere können PD-Controller daran scheitern, einen Fehler auf null herunterzuregeln, selbst wenn es keine externen Störungen gibt. Oftmals ist eine derartige Situation das Ergebnis einer systematischen externen Kraft, die nicht Teil des Modells ist. Zum Beispiel kann ein autonomes Fahrzeug, das auf einer überhöhten Oberfläche fährt, erkennen, dass es systematisch auf eine Seite gezogen wird. Verschleiß in Roboterarmen ruft ähnliche systematische Fehler hervor. In derartigen Situationen braucht man ein überproportionales Feedback, um den Fehler näher an null herunterzudrücken. Die Lösung für dieses Problem besteht darin, dem Regelungsgesetz einen dritten Term basierend auf dem über die Zeit integrierten Fehler hinzuzufügen:

$$a_t = K_P\big(y(t) - x_t\big) + K_I \int \big(y(t) - x_t\big)dt + K_D \frac{\partial\big(y(t) - x_t\big)}{\partial t} . \tag{25.3}$$

Hier ist K_I einfach ein weiterer Verstärkungsparameter. Der Term $\int(y(t) - x_t)dt$ berechnet das Integral des Fehlers über die Zeit. Die Wirkung dieses Terms ist, dass lange andauernde Abweichungen zwischen dem Referenzsignal und dem tatsächlichen Zustand korrigiert werden. Ist beispielsweise x_t für eine lange Zeitspanne kleiner als $y(t)$, wächst dieses Integral, bis das resultierende Regelsignal a_t erzwingt, dass dieser Fehler schrumpft. Integralterme stellen dann sicher, dass ein Regler keinen systematischen Fehler aufweist, allerdings auf Kosten einer höheren Gefahr für schwingendes Verhalten. Ein Regler mit allen drei Termen wird als **PID-Regler** (für Proportional Integral Derivative) bezeichnet. PID-Regler werden in der Industrie für die verschiedensten Regelungsprobleme eingesetzt.

25.6.2 Potentialfeldregelung

Wir haben Potentialfelder als zusätzliche Kostenfunktion in der Roboterbewegungsplanung eingeführt, sie können aber auch verwendet werden, um eine direkte Roboterbewegung zu erzeugen und auf die Pfadplanungsphase insgesamt zu verzichten. Dazu müssen wir eine anziehende Kraft definieren, die den Roboter in Richtung seiner Zielkonfiguration zieht, sowie ein abstoßendes Potentialfeld, das den Roboter von Hindernissen wegstößt. ▶ Abbildung 25.23 zeigt ein derartiges Potentialfeld. Sein einziges globales Minimum ist die Zielkonfiguration und der Wert ist die Summe der Distanz zu dieser Zielkonfiguration und der Nähe zu Hindernissen. Das in der Abbildung gezeigte Potentialfeld wurde ohne Planung generiert. Aufgrund dessen sind Potentialfelder gut geeignet für die Echtzeitsteuerung. Abbildung 25.23(a) zeigt einen Bewegungsverlauf für einer Roboter, der Hill-Climbing im Potentialfeld ausführt. In vielen Anwendungen kann das Potentialfeld für jede beliebige Konfiguration effizient berechnet werden. Des Weiteren läuft die Optimierung des Potentials auf die Berechnung des Potentialgradienten für die aktuelle Roboterkonfiguration hinaus. Diese Berechnungen sind in der Regel äußerst effizient, insbesondere im Vergleich zu Algorithmen für die Pfadplanung, die alle im ungünstigsten Fall exponentiell zur Dimensionalität des Konfigurationsraumes (den Freiheitsgraden) sind.

Die Tatsache, dass es mithilfe eines Potentialfeldes möglich ist, selbst über lange Distanzen im Konfigurationsraum auf diese Weise effizient einen Pfad zum Ziel zu finden, führt zu der Frage, warum man in der Robotik überhaupt eine Planung durchführt. Sind die Techniken der Potentialfelder ausreichend oder hatten wir nur Glück

in unserem Beispiel? Die Antwort ist, dass wir tatsächlich Glück hatten. Potentialfelder haben viele lokale Minima, die den Roboter festhalten können. Im Beispiel von Abbildung 25.23(b) nähert sich der Roboter dem Hindernis an, indem er einfach sein Schultergelenk dreht, bis es auf der falschen Seite des Hindernisses stecken bleibt. Das Potentialfeld ist nicht groß genug, um den Roboter zu veranlassen, seinen Ellenbogen zu beugen, sodass der Arm unter dem Hindernis durchpasst. Mit anderen Worten sind die Techniken der Potentialfelder ausgezeichnet für die lokale Robotersteuerung geeignet, erfordern manchmal aber trotzdem eine globale Planung. Ein weiterer wichtiger Nachteil bei Potentialfeldern ist, dass die Kräfte, die sie erzeugen, nur von dem Hindernis und den Roboterpositionen abhängig sind, nicht von der Geschwindigkeit des Roboters. Damit ist die Potentialfeldregelung eigentlich eine kinematische Methode und kann fehlschlagen, wenn sich der Roboter schnell bewegt.

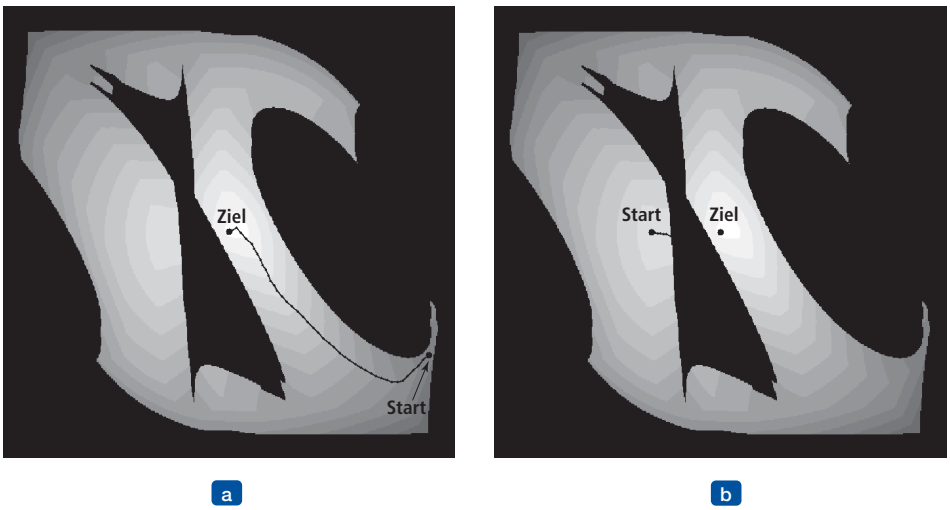


Abbildung 25.23: Potentialfeldsteuerung. Der Roboter steigt auf ein Potentialfeld bestehend aus abstoßenden Kräften, die durch die Hindernisse bedingt sind, und einer anziehenden Kraft, die der Zielkonfiguration entspricht. (a) Erfolgreicher Pfad. (b) Lokales Optimum.

25.6.3 Reaktive Steuerung

Bisher haben wir Regelungsentscheidungen betrachtet, die ein Modell der Umgebung voraussetzen, um entweder einen Referenzpfad oder ein Potentialfeld zu erzeugen. Dieser Ansatz weist jedoch einige Schwierigkeiten auf. Erstens sind Modelle, die ausreichend genau sind, häufig schwierig zu erhalten, insbesondere in komplexen oder weit entfernten Umgebungen, wie beispielsweise der Marsoberfläche, oder für Roboter, die wenige Sensoren besitzen. Zweitens können wir selbst in Fällen, wo wir ein Modell mit ausreichender Genauigkeit ableiten können, auf rechentechnische Schwierigkeiten treffen, und Lokalisierungsfehler könnten diese Techniken unpraktisch machen. In manchen Fällen ist die Architektur eines Reflexagenten mithilfe einer **reaktiven Steuerung** besser geeignet.

Stellen Sie sich zum Beispiel einen Roboter mit Beinen vor, der ein Bein über ein Hindernis zu heben versucht. Wir könnten für diesen Roboter eine Regel festlegen, die besagt, dass er das Bein um eine geringe Höhe h heben und dann vorwärts bewegen soll. Und wenn das Bein auf ein Hindernis trifft, soll er es zurückziehen und auf einer

größeren Höhe neu beginnen. Man könnte sagen, dass h einen Aspekt der Welt modelliert, doch wir können h auch als Hilfsvariable der Roboterregelung auffassen, frei von direkter physischer Bedeutung.

Ein solches Beispiel ist der sechsbeinige Roboter (Hexapod), den Sie in ▶ Abbildung 25.24 sehen und der die Aufgabe hat, sich in unwegsamem Gelände zu bewegen. Die Sensoren des Roboters sind nicht geeignet, um Modelle des Geländes für eine Pfadplanung zu erhalten. Doch selbst wenn wir ausreichend genaue Sensoren hinzufügen, würden die zwölf Freiheitsgrade (zwei für jedes Bein) das resultierende Problem der Pfadplanung rein rechentechnisch unmöglich machen.

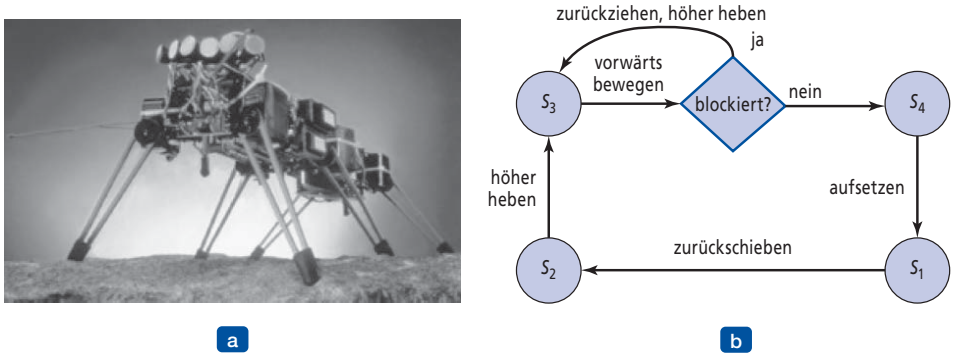


Abbildung 25.24: (a) Genghis, ein Hexapod-Roboter. (b) Ein erweiterter endlicher Automat für die Steuerung eines einzelnen Beines. Beachten Sie, dass dieser Automat auf Sensor-Feedback reagiert: Wenn ein Bein in der Phase der Vorwärtsschwingung stecken bleibt, wird es schrittweise immer höher gehoben.

Trotzdem ist es möglich, einen Regler direkt und ohne explizites Umgebungsmodell zu spezifizieren. (Dies haben wir bereits beim PD-Regler gesehen, der in der Lage war, einen komplexen Roboterarm an ein Ziel zu bringen, *ohne* ein explizites Modell der Roboterdynamik zu besitzen; er brauchte jedoch einen Referenzpfad, der durch ein kinematisches Modell erzeugt wurde.) Für den Hexapod-Roboter wählen wir zuerst einen **Gang**, d.h. ein Bewegungsmuster der Gliedmaßen. Bei einem statisch stabilen Gang bewegt der Roboter zuerst das rechte Vorderbein, das rechte Hinterbein und das linke Mittelbein vorwärts (während die drei anderen stehenbleiben) und bewegt dann die anderen drei Beine. Dieser Gang funktioniert gut in ebenem Gelände. Auf unwegsamem Gelände können Hindernisse verhindern, dass ein Bein nach vorn schwingt. Dieses Problem lässt sich durch eine bemerkenswert einfache Steuerregel umgehen: *Wenn die Vorwärtsbewegung eines Beines blockiert ist, zieht man es einfach zurück, hebt es höher und probiert es erneut.* Abbildung 25.24(b) zeigt die resultierende Regelung als endlichen Automaten; er besteht aus einem Reflexagenten mit Zustand, wobei der interne Zustand durch den Index des aktuellen Maschinenzustandes repräsentiert wird (s_1 bis s_4).

Mit Varianten dieses einfachen Rückkopplungs-Reglers wurden bereits bemerkenswert robuste Gangmuster erzeugt, nach denen der Roboter auch in unwegsamem Gelände manövrieren konnte. Offensichtlich ist ein solcher Regler modellfrei und er verwendet keine Suche, um die Regelungssignale zu erzeugen. Bei der Ausführung eines solchen Reglers spielt das Umgebungs-Feedback eine wesentliche Rolle. Die Software allein spezifiziert nicht, was tatsächlich passiert, wenn der Roboter in einer Umgebung platziert wird. Verhalten, das sich aus dem Zusammenspiel eines (einfachen) Reglers und einer (komplexen) Umgebung entwickelt, wird häufig als **emergentes Verhalten** bezeichnet.

Genaugenommen weisen alle in diesem Kapitel beschriebenen Roboter ein emergentes Verhalten auf, nämlich aufgrund der Tatsache, dass kein Modell perfekt ist. Historisch betrachtet ist der Begriff jedoch für Regelungstechniken reserviert, die keine expliziten Umgebungsmodelle verwenden. Ein emergentes Verhalten ist außerdem charakteristisch für eine große Anzahl an biologischen Organismen.

25.6.4 Reinforcement-Learning-Regelung

Eine besonders interessante Form der Regelung beruht auf Reinforcement-Learning in Form der **Strategiesuche** (siehe *Abschnitt 21.5*). Die Arbeiten hierzu sind in den letzten Jahren ausgesprochen einflussreich gewesen und haben komplexe Probleme der Robotik gelöst, für die vorher noch keine Lösung existiert hat. Ein Beispiel ist der autonome Kunstflug mit Helikoptern. ▶ Abbildung 25.25 zeigt einen autonomen Flip eines kleinen funkferngesteuerten Hubschraubers. Dieses Manöver ist aufgrund der hier auftretenden nichtlinearen Aerodynamik äußerst anspruchsvoll. Nur sehr erfahrene Piloten sind in der Lage, diese Flugfigur auszuführen. Doch eine Methode der Strategiesuche (wie in *Kapitel 21* beschrieben) hat nach wenigen Minuten Berechnung eine Strategie gelernt, die jederzeit einen Flip ausführen kann.

Abbildung 25.25: Mehrere Aufnahmen eines ferngesteuerten Hubschraubers, der einen Flip (Drehung um die Querachse oder Überschlag um die Nickachse) ausführt, und zwar basierend auf einer Strategie, die mit Reinforcement-Learning gelernt wurde. Bilder mit freundlicher Genehmigung von Andrew Ng, Universität Stanford.

Die Strategiesuche benötigt ein genaues Modell der Domäne, bevor sie eine Strategie finden kann. Die Eingabe für dieses Modell sind der Zustand des Hubschraubers zur Zeit t, die Steuersignale zur Zeit t und der resultierende Zustand zur Zeit $t + \Delta t$. Der Zustand eines Hubschraubers lässt sich durch die 3D-Koordinaten des Fluggerätes, seine Gier-, Nick- und Rollwinkel sowie die Änderungsgeschwindigkeit dieser sechs Variablen beschreiben. Die Steuersignale kommen von den manuellen Steuerelementen des Hubschraubers: Gashebel, Anstellwinkel, Höhenruder, Querruder und Seitenruder. Nun ist noch der resultierende Zustand zu klären – wie definieren wir ein Modell, das genau sagt, wie der Hubschrauber auf jedes Steuerelement reagiert? Die Antwort ist einfach: Man lässt den Hubschrauber von einem erfahrenen Piloten fliegen und zeichnet die Steuersignale, die der Experte über die Fernbedienung sendet, sowie die Zustandsvariablen des Hubschraubers auf. Rund vier Minuten Flugtraining mit Steuerung durch den Menschen genügen, um ein Voraussagemodell zu erstellen, das ausreichend genau ist, um das Fluggerät zu simulieren.

Bemerkenswert bei diesem Beispiel ist die Leichtigkeit, mit der dieses Lernkonzept ein kompliziertes Problem der Robotik löst. Dies ist einer von vielen Erfolgen des maschinellen Lernens in wissenschaftlichen Bereichen, die bislang von gründlicher mathematischer Analyse und Modellierung geprägt waren.

25.7 Software-Architekturen in der Robotik

Eine Methodik für die Strukturierung von Algorithmen wird als **Software-Architektur** bezeichnet. Eine Architektur beinhaltet Sprachen und Werkzeuge für die Entwicklung von Programmen sowie eine allgemeine Philosophie, wie die Programme kombiniert werden.

Moderne Software-Architekturen für die Robotik müssen entscheiden, wie reaktive Steuerung und modellbasierte deliberative Planung kombiniert werden können. In vielerlei Hinsicht haben reaktive und deliberative Techniken kontroverse Stärken und Schwächen. Die reaktive Steuerung ist sensorgesteuert und geeignet, Low-Level-Entscheidungen in Echtzeit zu treffen. Die reaktive Steuerung führt jedoch selten zu einer plausiblen Lösung auf globaler Ebene, weil globale Steuerungsentscheidungen von Informationen abhängig sind, die zum Zeitpunkt der Entscheidung nicht erfasst werden können. Für solche Probleme ist die deliberative Planung besser geeignet.

Aus diesem Grund verwenden die meisten Roboterarchitekturen reaktive Techniken auf den niedrigeren Ebenen der Steuerung und deliberative Techniken auf den höheren Ebenen. Wir haben eine solche Kombination bereits bei der Beschreibung der PD-Regler kennengelernt, wo wir einen (reaktiven) PD-Regler mit einem (deliberativen) Pfadplaner kombiniert haben. Architekturen, die reaktive und deliberative Techniken kombinieren, werden häufig als **hybride Architekturen** bezeichnet.

25.7.1 Subsumptions-Architektur

Die **Subsumptions-Architektur** (Brooks, 1986) ist ein Framework für den Aufbau reaktiver Regler aus endlichen Automaten. Knoten in diesen Maschinen können Tests auf bestimmte Sensorvariablen enthalten, sodass der Ausführungsverlauf eines endlichen Automaten von dem Ergebnis eines solchen Testes abhängig gemacht wird. Kanten können mit Nachrichten beschriftet werden. Diese werden beim Abarbeiten erzeugt und an die Motoren des Roboters oder an einen der anderen endlichen Automaten geschickt. Darüber hinaus besitzen endliche Automaten interne Timer (Uhren), die die Zeit steuern, wie lange es dauert, eine Kante zu durchlaufen. Die resultierenden Maschinen werden in der Regel als **erweiterte endliche Automaten**, **AFSM** (Augmented Finite State Machines), bezeichnet, wobei sich die Erweiterung auf die Verwendung von Uhren bezieht.

Ein Beispiel für einen einfachen AFSM ist die in Abbildung 25.24(b) gezeigte Vier-Zustands-Maschine, die zyklische Beinbewegungen für einen Hexapod-Roboter erzeugt. Dieser AFSM implementiert einen zyklischen Regler, dessen Ausführung größtenteils nicht auf dem Feedback aus der Umgebung basiert. In der Phase, wo das Bein nach vorne schwingt, verlässt er sich jedoch auf ein Sensor-Feedback. Bleibt das Bein stecken, d.h., es konnte den Schwung nach vorne nicht ausführen, zieht der Roboter das Bein zurück, hebt es ein bisschen höher und versucht erneut, es nach vorne zu schwingen. Der Regler ist also in der Lage, auf Gegebenheiten zu *reagieren*, die aus dem Zusammenspiel zwischen dem Roboter und seiner Umgebung entstehen.

Die Subsumptions-Architektur unterstützt zusätzliche Elemente für die Synchronisierung von AFSMs sowie für die Kombination von Ausgabewerten mehrerer, möglicherweise in Konflikt zueinander stehender AFSMs. Auf diese Weise wird es dem Programmierer möglich, immer komplexere Regler unter Verwendung einer Bottom-up-Vorgehensweise zu erstellen. In unserem Beispiel können wir mit AFSMs für einzelne

Beine beginnen, gefolgt von einem AFSM für die Koordination mehrerer Beine. Darauf aufbauend könnten wir höhere Verhaltensweisen implementieren, wie beispielsweise die Vermeidung von Kollisionen, wofür der Roboter möglicherweise zurückgehen und sich umdrehen muss.

Die Idee, einen Roboter-Controller aus AFSMs zusammenzusetzen, ist höchst faszinierend. Stellen Sie sich nur vor, wie schwierig es wäre, dasselbe Verhalten mit einem der im vorigen Abschnitt gezeigten Algorithmen für die Pfadplanung im Konfigurationsraum zu realisieren. Als Erstes bräuchten wir ein genaues Modell des Geländes. Der Konfigurationsraum eines Roboters mit sechs Beinen, die jeweils von zwei voneinander unabhängigen Motoren angetrieben werden, hat insgesamt 18 Dimensionen (12 Dimensionen für die Konfiguration der Beine und 6 Dimensionen für die Position und Ausrichtung des Roboters relativ zu seiner Umgebung). Selbst wenn unsere Computer schnell genug wären, Pfade in solch hochdimensionalen Räumen zu finden, müssten wir uns Sorgen über unangenehme Effekte machen, wenn der Roboter beispielsweise an einer Steigung ausgleitet. Aufgrund solcher stochastischen Effekte wäre ein einziger Pfad durch den Konfigurationsraum mit ziemlicher Sicherheit zu anfällig, und selbst ein PID-Regler wäre möglicherweise nicht in der Lage, mit solchen Eventualitäten zurechtzukommen. Mit anderen Worten ist die eigenständige Erzeugung von Bewegungsverhalten ein zu komplexes Problem für die heutigen Algorithmen zur Bewegungsplanung von Robotern.

Leider hat die Subsumptions-Architektur auch ihre eigenen Probleme. Erstens werden die AFSMs normalerweise durch reine Sensoreingaben gesteuert, eine Anordnung, die funktioniert, wenn die Sensordaten zuverlässig sind und alle erforderlichen Informationen für die Entscheidungsfindung enthalten, die aber fehlschlägt, wenn die Sensordaten auf nicht triviale Weise über die Zeit integriert werden müssen. Regler im Subsumptionsstil werden deshalb hauptsächlich für lokale Aufgaben angewendet, wie beispielsweise das Folgen eines Wandverlaufes oder die Bewegung auf sichtbare Lichtquellen. Zweitens macht es die fehlende Deliberation schwierig, die Aufgabe des Roboters zu ändern. Ein Roboter im Subsumptionsstil erledigt im Allgemeinen nur eine einzige Aufgabe und er hat keine Ahnung, wie er seine Steuerelemente anpassen könnte, um unterschiedliche Ziele zu erreichen (so wie der in *Abschnitt 2.2.2* beschriebene Mistkäfer). Schließlich sind Regler im Subsumptionsstil häufig schwer zu verstehen. In der Praxis liegt das enge Zusammenspiel zwischen Dutzenden von AFSMs (und der Umgebung) weit über dem, was die meisten menschlichen Programmierer verstehen. Aus all diesen Gründen wird die Subsumptions-Architektur in der Robotik trotz ihrer großen historischen Bedeutung nur selten verwendet. Immerhin hat sie andere Architekturen und einzelne Komponenten bestimmter Architekturen beeinflusst.

25.7.2 Drei-Schichten-Architekturen

Hybride Architekturen kombinieren Reaktion mit Deliberation. Die bekannteste hybride Architektur ist die **Drei-Schichten-Architektur**, die aus einer reaktiven Schicht, einer ausführenden Schicht und einer deliberativen Schicht besteht.

Die **reaktive Schicht** stellt die Low-Level-Steuerung für den Roboter bereit. Sie ist charakterisiert durch eine enge Sensor-Aktions-Schleife. Ihr Entscheidungszyklus liegt häufig in der Größenordnung von Millisekunden.

Die **ausführende Schicht** (oder Ablaufschicht) dient als Bindemittel zwischen der reaktiven und der deliberativen Schicht. Sie nimmt Befehle von der deliberativen Schicht entgegen und gibt sie an die reaktive Schicht weiter. Beispielsweise könnte die ausführende Schicht mehrere Zwischenpunkte verarbeiten, die ein deliberativer Pfadplaner erzeugt, und Entscheidungen dahingehend treffen, welches reaktive Verhalten sie aufrufen sollte. Entscheidungszyklen der ausführenden Schicht bewegen sich normalerweise innerhalb einer Sekunde. Die ausführende Schicht ist auch dafür verantwortlich, Sensorinformationen in eine interne Zustandsrepräsentation zu integrieren. Beispielsweise könnte sie die Routinen für die Lokalisierung und die Online-Kartenerstellung des Roboters beinhalten.

Die **deliberative Schicht** erzeugt globale Lösungen für komplexe Aufgaben mithilfe von Planung. Aufgrund der rechentechnischen Komplexität bei der Erstellung solcher Lösungen liegt der Entscheidungszyklus häufig in der Größenordnung von mehreren Minuten. Die deliberative Schicht (oder planende Schicht) verwendet Modelle für die Entscheidungsfindung. Diese Modelle könnten vorab bereitgestellt oder aus Daten gelernt werden und Zustandsinformationen nutzen, die in der ausführenden Schicht gesammelt wurden.

Varianten der Drei-Schichten-Architektur findet man in den meisten modernen Roboter-Softwaresystemen. Die Zerlegung in diese drei Schichten ist nicht sehr streng. Einige Roboter-Softwaresysteme besitzen zusätzliche Schichten, wie beispielsweise für die Benutzeroberflächen, um die Zusammenarbeit mit dem Menschen zur realisieren, oder eine Multiagentenebene für die Koordination der Aktionen eines Roboters mit denen eines anderen Roboters, der in derselben Umgebung arbeitet.

25.7.3 Pipeline-Architektur

Eine andere Architektur für Roboter ist die sogenannte **Pipeline-Architektur**. Genau wie die Subsumptions-Architektur führt die Pipeline-Architektur mehrere Prozesse parallel aus. Allerdings ähneln die konkreten Module in dieser Architektur denen in der Drei-Schichten-Architektur.

▶ Abbildung 25.26 zeigt ein Beispiel für die Pipeline-Architektur, die für die Steuerung eines autonomen Fahrzeuges verwendet wird. Die Daten gelangen auf der **Sensorschnittstellenschicht** in diese Pipeline. Anhand dieser Daten aktualisiert dann die **Wahrnehmungsschicht** die internen Modelle des Roboters für die Umgebung. Als Nächstes werden diese Modelle an die **Planungs- und Steuerungsschicht** übergeben, die die internen Pläne des Roboters anpasst und sie in die eigentlichen Steuersignale für den Roboter überführt. Diese werden dann über die **Schnittstellenschicht** zurück an das Fahrzeug übertragen.

Der Schlüssel für diese Pipeline-Architektur liegt darin, dass alles parallel abläuft. Während die Wahrnehmungsschicht die neuesten Sensordaten verarbeitet, stützt sich die Steuerungsschicht bei ihren Entscheidungen auf etwas ältere Daten. Die Pipeline-Architektur ähnelt in dieser Hinsicht dem menschlichen Gehirn. Wir schalten unsere Bewegungsregler nicht ab, wenn wir neue Sensordaten verarbeiten. Stattdessen empfangen, planen und handeln wir gleichzeitig. Prozesse in der Pipeline-Architektur laufen asynchron ab und sämtliche Berechnungen sind datengesteuert. Das resultierende System ist robust und es ist schnell.

Die in Abbildung 25.26 dargestellte Architektur enthält auch andere, übergreifende Module, die für die Kommunikation zwischen den verschiedenen Elementen der Pipeline zuständig sind.

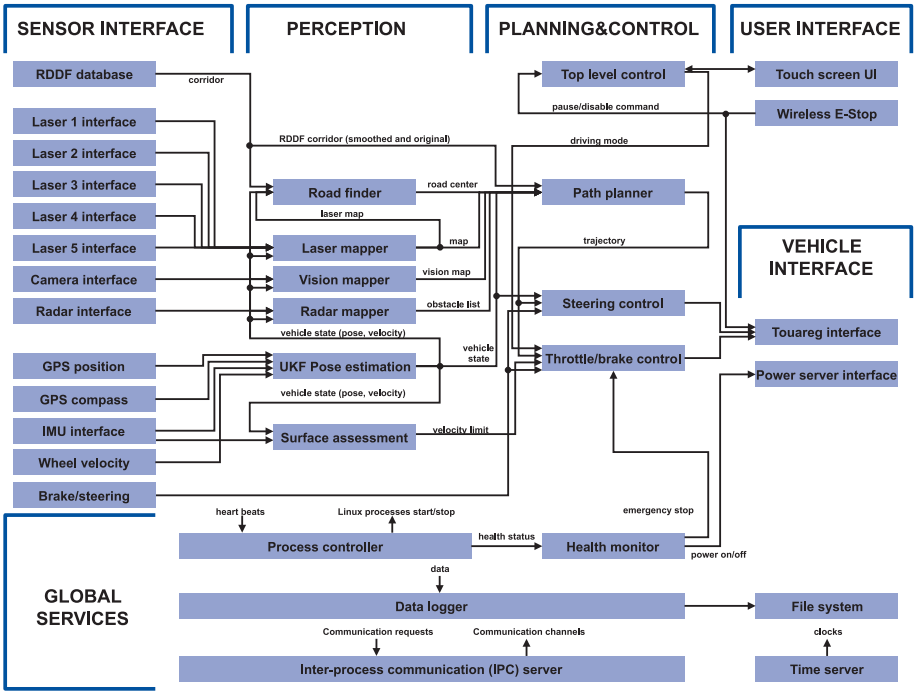


Abbildung 25.26: Softwarearchitektur eines Roboterfahrzeuges. Diese Software implementiert eine Datenpipeline, in der alle Module gleichzeitig Daten verarbeiten.

25.8 Anwendungsbereiche

Die folgende Übersicht gibt einige der wichtigsten Anwendungsbereiche für die Robotiktechnologie an.

Industrie und Landwirtschaft: Traditionell werden Roboter in Bereichen eingesetzt, für die schwere menschliche Arbeit erforderlich ist und die gleichzeitig strukturiert genug sind, um eine Automatisierung durch Roboter zuzulassen. Das beste Beispiel ist das Fließband, wo Manipulatoren ganz allgemeine Aufgaben ausführen, wie beispielsweise Montage, Teileplatzierung, Materialzuführung, Schweißen und Lackieren. In vielen dieser Aufgabenbereiche sind die Roboter sehr viel kostengünstiger geworden als die menschlichen Arbeitskräfte. In Außenbereichen wurden viele der schweren Maschinen, die wir beispielsweise für Ernte, Bohrungen oder den Erdaushub verwenden, in Roboter umgewandelt. Zum Beispiel hat ein Projekt an der Carnegie Mellon-Universität demonstriert, dass Roboter Farbe von großen Schiffen etwa 50-mal schneller und mit sehr viel weniger Umweltbelastung als Menschen entfernen können. Prototypen für autonome Bohrroboter haben sich als schneller und genauer als Menschen erwiesen, um Erz in unterirdischen Minen zu transportieren. Roboter wurden verwen-

det, um äußerst präzise Karten von verlassenen Minen und Abwasserkanalsystemen zu erstellen. Obwohl sich viele dieser Systeme noch in der Prototypphase befinden, ist es nur eine Frage der Zeit, wann die Roboter einen Großteil der halbmechanischen Arbeit übernommen haben werden, die heute noch von Menschen ausgeführt wird.

Transport: Der Transport durch Roboter hat viele Einsatzbereiche: von eigenständigen Hubschraubern, die Nutzlasten an schwer zugängliche Orte bringen, über automatische Rollstühle, die Menschen transportieren, die nicht in der Lage sind, selbst Rollstühle zu steuern, bis hin zu autonomen Gabelstaplern, die sogar erfahrenen menschlichen Fahrern überlegen sind, um Container am Ladedock von Schiffen auf Lastwagen zu transportieren. Ein wichtiges Beispiel für Innenraum-Transportroboter oder „Handlanger“ ist der in ▶ Abbildung 25.27(a) gezeigte Helpmate-Roboter. Dieser Roboter wurde in Dutzenden von Krankenhäusern eingesetzt, um Essen und andere Dinge zu transportieren. Auf Fabrikgeländen verwendet man inzwischen routinemäßig autonome Fahrzeuge, um Waren im Lager oder zwischen Produktionslinien zu transportieren. Das in ▶ Abbildung 25.27(b) gezeigte Kiva-System unterstützt Arbeiter in Versandzentren beim Verpacken der Artikel in die Versandbehälter.

a b

Abbildung 25.27: (a) Der Helpmate-Roboter transportiert Essen und andere medizinische Gerätschaften in Dutzenden von Krankenhäusern auf der ganzen Welt. (b) Kiva-Roboter sind Teil eines Materialzuführungssystems für bewegliche Regale in Versandzentren. Bild mit freundlicher Genehmigung von Kiva Systems.

Für viele dieser Roboter mussten die Umgebungen angepasst werden, damit sie funktionieren. Die gebräuchlichsten Anpassungen sind Lokalisierungshilfen, wie beispielsweise Induktionsschleifen im Boden, aktive Baken oder Barcode-Markierungen. Eine offene Herausforderung in der Robotik ist der Entwurf von Robotern, die für die Navigation natürliche Hinweise statt speziell dafür bereitgestellte Geräte verwenden können, insbesondere in Umgebungen, wie beispielsweise der Tiefsee, wo GPS nicht zur Verfügung steht.

Roboterfahrzeuge: Die meisten von uns nutzen Autos tagtäglich. Manche telefonieren, während sie fahren. Einige schreiben sogar SMS. Das traurige Ergebnis: Mehr als eine Million Menschen sterben jedes Jahr durch Verkehrsunfälle. Roboterfahrzeuge wie BOSS und STANLEY machen Hoffnung. Durch sie wird nicht nur das Fahren wesentlich sicherer, sie befreien uns auch davon, während unseres täglichen Arbeitsweges auf die Straße achten zu müssen.

Fortschritte bei Roboterfahrzeugen wurden durch die *DARPA Grand Challenge* vorangetrieben, ein Rennen über mehr als 100 Meilen (etwa 160 km) in unbekanntem Wüstengelände, was eine wesentlich komplexere Aufgabe darstellte, als sie je zuvor realisiert wurde. Das Fahrzeug STANLEY des Teams von der Stanford-Universität absolvierte 2005 den Kurs in weniger als sieben Stunden, gewann damit das Preisgeld von 2 Millionen Doller und sicherte sich einen Platz im National Museum of American History. ▶ Abbildung 25.28(a) zeigt das Fahrzeug BOSS, das 2007 die *DARPA Urban Challenge* gewann, ein kompliziertes Straßenrennen im Stadtverkehr, wo Roboter mit anderen Robotern konfrontiert wurden und die Verkehrsregeln zu beachten hatten.

a b

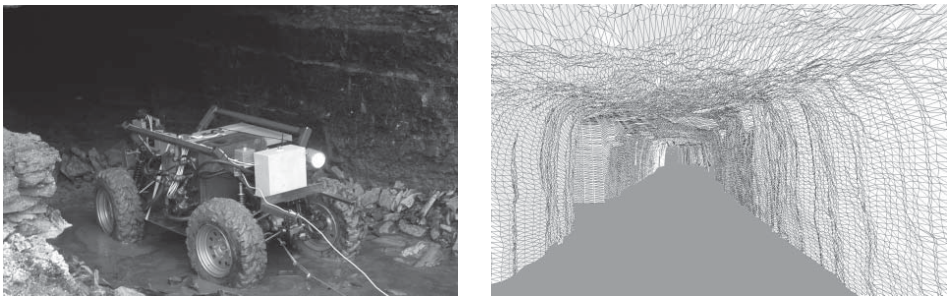
Abbildung 25.28: (a) Roboterfahrzeug BOSS, das die *DARPA Urban Challenge* gewann. Mit freundlicher Genehmigung der Carnegie Mellon University. (b) Operationsroboter im Operationssaal (Bild mit freundlicher Genehmigung von da Vinci Surgical Systems).

Medizin: Roboter werden immer häufiger verwendet, um Ärzte bei der Instrumentplatzierung zu unterstützen, wenn sie an komplizierten Organen arbeiten, wie beispielsweise Gehirn, Augen oder Herz. ▶ Abbildung 25.28(b) zeigt ein solches System. Roboter sind dank ihrer hohen Genauigkeit zu unverzichtbaren Werkzeugen in einem ganzen Bereich von chirurgischen Operationen geworden. In Pilotstudien hat man festgestellt, dass bei Koloskopien weniger Verletzungsgefahr besteht. Außerhalb des Operationssaales haben die Forscher begonnen, robotische Hilfe für ältere und behinderte Menschen zu entwickeln, wie beispielsweise intelligente Gehhilfen sowie intelligente Spielzeuge, die die Menschen daran erinnern, ihre Medizin einzunehmen, und für Behaglichkeit sorgen. Außerdem arbeiten Forscher an Robotern für die Rehabilitation, die den Patienten bei der Durchführung bestimmter Übungen helfen.

Gefährliche Umgebungen: Roboter haben Menschen dabei unterstützt, atomverseuchten Müll aufzuräumen, insbesondere in den Kernkraftwerken Tschernobyl und Three Mile Island. Roboter wurden nach dem Einsturz des World Trade Centers eingesetzt, wo sie sich in Bereichen bewegten, die für die menschlichen Such- und Rettungsmannschaften zu gefährlich waren.

Einige Länder setzen Roboter ein, um Munition zu transportieren und Bomben zu entschärfen – eine äußerst gefährliche Aufgabe. Mehrere Forschungsprojekte entwickeln Prototyproboter für die Säuberung von Land- und Seeminenfeldern. Die meisten Roboter für diese Aufgaben sind ferngesteuert – ein Mensch bedient sie über eine Fernsteuerung. Die Bereitstellung solcher Roboter mit eigener Steuerung ist ein wichtiger nächster Schritt.

Erkundung: Roboter waren an Orten, wo nie zuvor ein Mensch gewesen ist, einschließlich der Marsoberfläche (siehe Abbildung 25.2(b)). Roboterarme helfen Astronauten beim Aussetzen und beim Finden von Satelliten sowie beim Bau der internationalen Raumstation. Darüber hinaus helfen Roboter, unter Wasser zu forschen. Sie werden häufig eingesetzt, um Karten für die Umgebung gesunkener Schiffe zu erstellen. ▶ Abbildung 25.29 zeigt einen Roboter, der eine Karte einer stillgelegten Kohlenmine anfertigt, ebenso wie ein 3D-Modell der Mine, das unter Verwendung von Entfernungssensoren erstellt wurde. 1996 setzte ein Forscherteam einen Laufroboter in den Krater eines aktiven Vulkans, um Daten zu erhalten, die für die Klimaforschung sehr wichtig waren. Unbemannte Luftfahrzeuge, auch als **Drohnen** bezeichnet, werden in militärischen Operationen eingesetzt. Roboter werden zu sehr effektiven Werkzeugen, um Informationen in Bereichen zu sammeln, die für den Menschen schwierig (oder gefährlich) zu betreten sind.



a b

Abbildung 25.29: (a) Ein Roboter, der eine Karte von einer verlassenen Kohlenmine anfertigt. (b) 3D-Karte einer Mine, die der Roboter erstellt hat.

Dienstleistungen: Die Dienstleistung ist ein neuer und aufstrebender Anwendungsbereich der Robotik. Dienstleistungsroboter helfen Einzelpersonen, ihre täglichen Arbeiten zu erledigen. Kommerziell angebotene Haushaltsroboter sind unter anderem automatische Staubsauger, Rasenmäher oder Golf-Caddies. Der bekannteste mobile Roboter ist ein Dienstleistungsroboter: der in ▶ Abbildung 25.30(a) gezeigte Staubsaugerroboter **Roomba**. Bisher wurden mehr als drei Millionen Geräte verkauft. Roomba kann autonom navigieren und seine Aufgaben ohne menschliche Hilfe ausführen.

Andere Dienstleistungsroboter arbeiten an öffentlichen Plätzen, beispielsweise an Informationsständen, wie sie in Shopping-Malls und bei Handelsmessen eingesetzt werden, ebenso wie in Museen als Führer. Für die Dienstleistungen ist eine Zusammenarbeit mit

dem Menschen erforderlich, ebenso wie die Fähigkeit, robust mit unvorhersehbaren und dynamischen Umgebungen zurechtzukommen.

a b

Abbildung 25.30: (a) Roomba, der meistverkaufte mobile Roboter der Welt, ein Bodenstaubsauger. Bild mit freundlicher Genehmigung von iRobot, 2009. (b) Roboterhand, die nach einer menschlichen Hand modelliert wurde. Bild mit freundlicher Genehmigung der Universitäten Washington und Carnegie Mellon.

Unterhaltung: Seit einiger Zeit sind Roboter in der Unterhaltungs- und Spielzeugindustrie zu finden. In Abbildung 25.6(b) haben Sie **Roboterfußball** gesehen, ein Spiel, das dem menschlichen Fußball ganz ähnlich ist, aber mit autonomen mobilen Robotern gespielt wird. Roboterfußball bietet zahlreiche Gelegenheiten für die Forschung in der KI, wobei viele Probleme aufgedeckt werden, die auch für etliche andere, ernsthaftere Roboteranwendungen relevant sind. Jährliche Turniere im Roboterfußball haben große Mengen von KI-Forschern angezogen und dem Bereich der Robotik viel Aufmerksamkeit eingebracht.

Hilfsmittel für den Menschen: Der letzte der hier aufgeführten Anwendungsbereiche für die Robotiktechnologie sind Hilfsmittel für den Menschen. Forscher haben Gehhilfen entwickelt, die Menschen bei der Bewegung unterstützen, ähnlich Rollstühlen. Mehrere Forschungen konzentrieren sich momentan auf die Entwicklung von Geräten, die es den Menschen erleichtern sollen, ihre Arme zu bewegen oder zu gehen, indem sie ihnen zusätzliche Kräfte bereitstellen, die extraskelettär befestigt werden. Wenn solche Geräte permanent befestigt werden, kann man sie sich auch als künstliche robotische Extremitäten vorstellen. ▶ Abbildung 25.30(b) zeigt eine Roboterhand, die in Zukunft als Prothese dienen kann.

Eine weitere Form der Hilfsmittel für den Menschen ist die robotische Teleoperation oder Telepräsenz. Bei der Teleoperation werden Aufgaben mit Hilfe robotischer Geräte auf große Entfernungen ausgeführt. Eine beliebte Konfiguration für die robotische Teleoperation ist die Master/Slave-Konfiguration, wobei ein Roboter-Manipulator die Bewegung eines entfernten menschlichen Operators emuliert, die über eine haptische Schnittstelle gemessen wird. Unterwasserfahrzeuge werden oftmals per Teleoperation betrieben; die Fahrzeuge können in Tiefen vordringen, die für Menschen zu gefährlich sind, dennoch aber vom menschlichen Bediener geführt werden. Alle diese Systeme erweitern die Fähigkeit des Menschen, mit seiner Umgebung zu interagieren. Einige Projekte gehen sogar so weit, dass sie den Menschen replizieren, allerdings auf sehr künstlicher Ebene. Mehrere Unternehmen in Japan verkaufen bereits heute humanoide Roboter.

Bibliografische und historische Hinweise

Das Wort **Roboter** wurde von dem tschechischen Dramatiker Karel Capek in seinem Werk *R.U.R.* (Rossum's Universal Robots) von 1921 geprägt. Die Roboter, die dort chemisch gewachsen waren und nicht mechanisch konstruiert wurden, wenden sich schließlich gegen ihre Meister und entscheiden, die Weltherrschaft zu übernehmen. Es scheint (Glanc, 1978), dass es eigentlich Capeks Bruder Josef war, der zuerst die tschechischen Wörter „robota" (Zwangsarbeit) und „robotnik" (Sklave) kombinierte, um in seiner Kurzgeschichte *Opilec* von 1917 das Wort „Roboter" zu bilden.

Der Begriff *Robotik* wurde zuerst von Asimov (1950) verwendet. Die Robotik hat jedoch (unter anderem Namen) eine sehr viel längere Geschichte. In der alten griechischen Mythologie wurde angeblich von Hephaistos, dem griechischen Gott für die Schmiedekunst, ein mechanischer Mann namens Talos entworfen und gebaut. Im 18. Jahrhundert wurden wunderbare Automaten gebaut – beispielsweise die mechanische Ente von Jacques Vaucanson als frühes Beispiel –, aber die komplexen Verhaltensweisen, die sie aufwiesen, wurden vollständig im Voraus festgelegt. Möglicherweise das erste Beispiel für ein einem programmierbaren Roboter ähnliches Gerät war der Webstuhl von Jacquard (1805), wie in *Kapitel 1* beschrieben.

Der erste kommerzielle Roboter war ein Roboterarm namens **Unimate**, eine Abkürzung für *Universal Automation*. Unimate wurde von Joseph Engelberger und George Devol entwickelt. 1961 wurde der erste Unimate-Roboter an General Motors verkauft, wo er für die Herstellung von Bildröhren für Fernsehgeräte verwendet wurde. 1961 war auch das Jahr, in dem Devol das erste US-Patent für einen Roboter anmeldete. Elf Jahre später, 1972, war Nissan Corp. eines der ersten Unternehmen, das ein vollständiges Fließband unter Verwendung von Robotern automatisierte. Es wurde von Kawasaki entwickelt, die Roboter stammten von Engelberger aus Devols Unternehmen Unimation. Diese Entwicklung war der Ursprung für eine große Revolution, die vor allem in Japan und den USA stattfand und die immer noch anhält. Unimation folgte 1978 mit der Entwicklung des Roboters **PUMA** (Programmable Universal Machine for Assembly). Der ursprünglich für General Motors entwickelte Roboter PUMA war der De-facto-Standard für Robotikmanipulation in den nächsten 20 Jahren. Momentan gibt es schätzungsweise eine Million arbeitende Roboter weltweit, von denen mehr als die Hälfte in Japan installiert ist.

Die Literatur über die Robotikforschung kann in zwei Teile zerlegt werden: mobile Roboter und stationäre Manipulatoren. Die von Grey Walter 1948 gebaute „Schildkröte" kann als erster selbstständiger mobiler Roboter betrachtet werden, obwohl sein Steuerungssystem nicht programmierbar war. „Hopkins Beast", der Anfang der 1960er Jahre an der John Hopkins University gebaut wurde, war sehr viel komplexer; er hatte Hardware für die Mustererkennung und konnte die Abdeckung einer Standardsteckdose erkennen. Er war in der Lage, nach Steckdosen zu suchen, sich selbst einzustöpseln und seine Batterie aufzuladen! Dennoch hatte das Beast ein begrenztes Repertoire an Fähigkeiten. Der erste allgemeine mobile Roboter war „Shakey", entwickelt am Stanford Research Institute (heute SRI) in den späten 1960er Jahren (Fikes und Nilsson, 1971; Nilsson, 1984). Shakey war der erste Roboter, der Wahrnehmung, Planung und Ausführung in sich vereinigte, und ein Großteil der nachfolgenden Forschung in der KI wurde durch diese bemerkenswerte Leistung beeinflusst. Andere einflussreiche Projekte sind unter anderen das Stanford Cart und der CMU Rover (Moravec, 1983). Cox und Wilfong (1990) beschreiben klassische Arbeiten über autonome Fahrzeuge.

Der Bereich der Kartenerstellung durch Roboter hat sich aus zwei verschiedenen Ursprüngen entwickelt. Die erste Schiene begann mit der Arbeit von Smith und Cheeseman (1986), die Kalman-Filter auf das simultane Lokalisierungs- und Kartenerstellungsproblem anwendeten. Dieser Algorithmus wurde zuerst von Moutarlier und Chatila (1989) implementiert und später von Leonard und Durrant-Whyte (1992) erweitert; in Dissanayake et al. (2001) finden Sie einen Überblick über frühe Variationen von Kalman-Filtern. Die zweite Schiene begann mit der Entwicklung der **Occupancy-Grid**-Repräsentation für die probabilistische Kartenerstellung, die die Wahrscheinlichkeit angibt, dass eine (x, y)-Position von einem Hindernis belegt ist (Moravec und Elfes, 1985). Kuipers und Levitt (1988) gehörten zu den ersten, die topologische statt metrische Abbildungen vorschlugen, motiviert durch Modelle der menschlichen räumlichen Wahrnehmung. Eine wegweisende Arbeit von Lu und Milios (1997) erkannte die geringe Dichte beim Problem der simultanen Lokalisierung und Kartenerstellung, was zur Entwicklung nichtlinearer Optimierungsverfahren durch Konolige (2004) sowie Montemerlo und Thrun (2004) und ebenso zu hierarchischen Methoden von Bosse et al. (2004) führte. Shatkay und Kaelbling (1997) sowie Thrun et al. (1998) führten den EM-Algorithmus in das Gebiet der Roboterkartenerstellung für Datenzuordnung ein. Einen Überblick über probabilistische Methoden der Kartenerstellung finden Sie in Thrun et al. (2005).

Frühe Techniken der Lokalisierung mobiler Roboter sind in Borenstein et al. (1996) beschrieben. Obwohl die Kalman-Filter als Lokalisierungsmethode in der Steuerungstheorie seit Jahrzehnten bekannt waren, erschien die allgemeine probabilistische Formulierung des Lokalisierungsproblems in der KI-Literatur erst sehr viel später, nämlich in der Arbeit von Tom Dean und seinen Kollegen (Dean et al., 1990,1990) sowie Simmons und Koenig (1995). Die letztgenannte Arbeit führte den Begriff der **Markov-Lokalisierung** ein. Die erste reale Anwendung dieser Technik stammt von Burgard et al. (1999) in einer Folge von Robotern, die in Museen eingesetzt wurden. Die Monte-Carlo-Lokalisierung basiert auf Partikelfiltern und wurde von Fox et al. (1999) entwickelt. Heute wird sie allgemein eingesetzt. Der **Rao-Blackwellized-Partikelfilter** kombiniert Partikelfilter für die Roboterlokalisierung mit exakter Filterung für die Kartenerstellung (Murphy und Russell, 2001; Montemerlo et al., 2002).

Die Untersuchung der Manipulator-Roboter, ursprünglich als **Hand-Auge-Maschinen** bezeichnet, hat sich in eine ganz andere Richtung entwickelt. Der erste wichtige Ansatz bei der Erstellung einer Hand-Auge-Maschine war MH-1 von Heinrich Ernst, beschrieben in seiner Doktorarbeit am MIT (Ernst, 1961). Das Machine Intelligence-Projekt in Edinburgh demonstrierte ebenfalls ein beeindruckendes frühes System für visionsbasierte Montage namens FREDDY (Michie, 1972). Nach diesen Pionierleistungen konzentrierte sich ein Großteil der Arbeit auf geometrische Algorithmen für vollständig beobachtbare Bewegungsplanungsprobleme. Eine einflussreiche Arbeit von Reif (1979) zeigte, dass die Roboterbewegungsplanung PSPACE-hart war. Die Darstellung von Konfigurationsräumen geht auf Lozano-Perez (1983) zurück. Sehr einflussreich war eine Folge von Arbeiten von Schwartz und Sharir über die von ihnen sogenannten **Piano-Rücker**-Probleme (Schwartz et al., 1987).

Die rekursive Zellzerlegung für die Planung von Konfigurationsräumen stammt von Brooks und Lozano-Perez (1985) und wurde von Zhu und Latombe (1991) wesentlich verbessert. Die ersten Algorithmen für die Skelettierung basierten auf Voronoi-Diagrammen (Rowat, 1979) und **Sichtbarkeitsgraphen** (Wesley und Lozano-Perez, 1979). Guibas et al. (1992) entwickelten effiziente Techniken für die inkrementelle Berech-

nung von Voronoi-Diagrammen, und Choset (1996) verallgemeinerte die Voronoi-Diagramme auf sehr viel weitläufigere Bewegungsplanungsprobleme. John Canny (1988) stellte den ersten einfach exponentiellen Algorithmus für die Bewegungsplanung. Das einflussreiche Buch von Latombe (1991) beschreibt eine Vielzahl von Ansätzen für das Bewegungsplanungsproblem, ebenso die Veröffentlichungen von Choset et al. (2004) und LaValle (2006). Kavraki et al., 1996) entwickelten probabilistische Straßenkarten, die momentan die effektivste Methode darstellen. Die Feinbewegungsplanung mit begrenzten Sensoreingaben wurde von Lozano-Perez et al. (1984) sowie Canny und Reif (1987) untersucht. Orientierungspunkt-basierte Navigation (Lazanas und Latombe, 1992) verwendet viele derselben Ideen im Bereich der mobilen Roboter. Grundlagenarbeiten zur Anwendung von POMDP-Methoden (*Abschnitt 17.4*) auf die Bewegungsplanung bei Unsicherheit stammen für die Robotik von Pineau et al. (2003) und Roy et al. (2005).

Die Steuerung von Robotern als dynamische Systeme – egal ob für Manipulation oder Navigation – hat zu umfassender Literatur geführt, die sich in diesem Kapitel nur streifen lässt. Wichtige Arbeiten sind beispielsweise eine Trilogie zur Impedanzsteuerung von Hogan (1985) sowie eine allgemeine Studie zur Roboterdynamik von Featherstone (1987). Dean und Wellman (1991) gehören zu den Ersten, die versuchten, Steuerungstheorie und KI-Planungssysteme zu verknüpfen. Drei klassische Lehrbücher zur Mathematik der Robotermanipulation stammen von Paul (1981), Craig (1989) und Yoshikawa (1990). Der Bereich des **Greifens** ist in der Robotik ebenso wichtig – das Problem, einen stabilen Griff zu erkennen, ist relativ schwierig (Mason und Salisbury, 1985). Ein zuverlässiges Greifen bedingt Berührungssensoren oder ein **haptisches Feedback**, um Kontaktkräfte zu bestimmen und Gleiten eines Objektes zu erkennen (Fearing und Hollerbach, 1985).

Die Potentialfeldsteuerung, die versucht, die Probleme der Bewegungsplanung und Steuerung gleichzeitig zu lösen, wurde in der Robotikliteratur von Khatib (1986) eingeführt. In der mobilen Robotik wurde diese Idee als praktische Lösung zum Problem der Kollisionsvermeidung betrachtet. Später wurde sie von Borenstein (1991) erweitert zu einem Algorithmus namens **Vektorfeldhistogramme**. Navigationsfunktionen, die Robotikversion einer Steuerungsstrategie für deterministische MDPs, wurden von Koditschek (1987) eingeführt. Mit Reinforcement-Learning in der Robotik beschäftigt sich die wegweisende Arbeit von Bagnell und Schneider (2001) sowie von Ng et al. (2004), die das Paradigma im Kontext der autonomen Hubschraubersteuerung entwickelt haben.

Der Bereich der Software-Architekturen für Roboter erinnert an eine eher religiöse Debatte. Der gute alte KI-Kandidat – die Drei-Schichten-Architektur – stammt von dem Entwurf für Shakey und wurde von Gat (1998) überarbeitet. Die Subsumptions-Architektur geht auf Rodney Brooks (1986) zurück, obwohl auch ähnliche Konzepte unabhängig davon von Braitenberg (1984) entwickelt wurden, dessen Buch *Vehicles* eine Folge einfacher Roboter beschreibt, die auf dem Verhaltensansatz basieren. Dem Erfolg des mit sechs Beinen gehenden Roboters von Brooks folgten viele andere Projekte. Connell entwickelte in seiner Doktorarbeit (1989) einen mobilen Roboter, der in der Lage war, Objekte zu finden, und der vollständig reaktiv war. Erweiterungen des verhaltensbasierten Paradigmas auf Multirobot-Systeme findet man in Mataric (1997)

und Parker (1996) beschrieben. GRL (Horswill, 2000) und COLBERT (Konolige, 1997) abstrahierten die Konzepte der nebenläufigen verhaltensbasierten Robotik zu allgemeineren Robotersteuerungen und Sprachen. Arkin (1998) bietet einen Überblick über die bekanntesten Ansätze auf diesem Gebiet.

Die Forschung im Bereich der mobilen Robotik wurde in den letzten zehn Jahren von zwei wichtigen Wettbewerben stimuliert. Der früheste ist der seit 1992 jährlich stattfindende Wettbewerb der mobilen Robotik der AAAI. Der erste Gewinner des Wettbewerbes war CARMEL (Congdon et al., 1992). Die Fortschritte waren stetig und beeindruckend: In den letzten Wettbewerben traten die Roboter in das Konferenzgebäude ein, suchten ihren Weg zum Anmeldungsbüro, meldeten sich für die Konferenz an und stellten sich sogar selbst kurz vor. Der Wettbewerb **Robocup**, 1995 von Kitano und Kollegen (1997a) ins Leben gerufen, hat sich zum Ziel gesetzt, bis zum Jahr 2050 „ein Team vollständig autonomer humanoider Roboter zusammenzustellen, das gegen das menschliche Weltmeisterteam im Fußball gewinnt". Gespielt wird in Ligen für simulierte Roboter, Roboter mit Rädern unterschiedlicher Größen sowie humanoide Roboter. Im Jahr 2009 nahmen Teams aus 43 Ländern teil und die Übertragungen des Ereignisses wurden von Millionen Interessierten verfolgt. Visser und Burkhard (2007) zeichnen die Verbesserungen nach, die hinsichtlich Wahrnehmung, Teamkoordinierung und Low-Level-Fertigkeiten im letzten Jahrzehnt gemacht wurden.

Die von der DARPA (der Technologieabteilung des US-Verteidigungsministeriums) in den Jahren 2004 und 2005 organisierte **DARPA Grand Challenge** forderte von autonomen Robotern, mehr als 100 Meilen (etwa 160 km) durch unbekanntes Wüstengelände in weniger als 10 Stunden zu fahren (Buehler et al., 2006). Im ersten Wettbewerb von 2004 schaffte kein Roboter mehr als 7,4 Meilen (knapp 12 km), was viele zu der Annahme verleitete, dass das Preisgeld nie ausgezahlt werden müsste. Im Jahr 2005 gewann der Roboter STANLEY von der Stanford-Universität den Wettbewerb in weniger als 7 Stunden Fahrzeit (Thrun, 2006). Die DARPA organisierte dann mit der **Urban Challenge** einen Wettbewerb, in dem Roboter eine Strecke von 60 Meilen (knapp 100 km) in einer städtischen Umgebung mit anderem Verkehr navigieren mussten. Der Roboter BOSS der Carnegie Mellon-Universität belegte den ersten Platz und erhielt das Preisgeld von 2 Millionen Dollar (Urmson und Whittaker, 2008). Zu den Pionieren in der Entwicklung von Roboterfahrzeugen gehören Dickmanns und Zapp (1987) sowie Pomerleau (1993).

Zwei frühe Lehrbücher von Dudek und Jenkin (2000) sowie Murphy (2000) behandeln die Robotik allgemein. Ein neuerer Überblick ist bei Bekey (2008) zu finden. Ein ausgezeichnetes Buch zur Robotermanipulation widmet sich auch komplexeren Themen wie der konformen Bewegung (Mason, 2001). Die Roboterplanung wird in Choset et al. (2004) und LaValle (2006) behandelt. Thrun et al. (2005) bieten eine Einführung in probabilistische Robotik. Die wichtigste Konferenz für Robotik ist die *Robotics: Science and Systems Conference*, gefolgt von der *IEEE International Conference on Robotics and Automation*. Zu den führenden Fachzeitschriften für Robotik gehören *IEEE Robotics and Automation*, das *International Journal of Robotics Research* und *Robotics and Autonomous Systems*.

Zusammenfassung

Die Robotik beschäftigt sich mit intelligenten Agenten, die die physische Welt manipulieren. In diesem Kapitel haben wir die folgenden Grundlagen zur Hardware und Software von Robotern kennengelernt:

- Roboter sind mit **Sensoren** für die Wahrnehmung ihrer Umgebung und Effektoren für die Anwendung physischer Kräfte auf ihre Umgebung ausgestattet. Die meisten Roboter sind entweder Manipulatoren, die an einer festen Position verankert sind, oder mobile Roboter, die sich bewegen können.
- Die Robotikwahrnehmung beschäftigt sich mit der Abschätzung entscheidungsrelevanter Größen aus Sensordaten. Dazu brauchen wir eine interne Repräsentation sowie eine Methode für die Aktualisierung dieser internen Repräsentation über die Zeit. Gebräuchliche Beispiele für schwierige Wahrnehmungsprobleme sind unter anderem **Lokalisierung**, **Kartenerstellung** und **Objekterkennung**.
- **Probabilistische Filteralgorithmen** wie beispielsweise Kalman-Filter und Partikelfilter sind wichtig für die Roboterwahrnehmung. Diese Techniken verwalten den Belief State, d.h. eine A-posteriori-Verteilung über Zustandsvariablen.
- Die Planung der Roboterbewegung erfolgt normalerweise im **Konfigurationsraum**, wobei jeder Punkt die Position und die Ausrichtung des Roboters sowie seiner Gelenkwinkel angibt.
- Konfigurationsraum-Suchalgorithmen sind unter anderem Techniken der **Zellzerlegung**, die den Raum aller Konfigurationen in endlich viele Zellen zerlegen, sowie Methoden der **Skelettierung**, die Konfigurationsräume auf Kopien mit weniger Dimensionen projizieren. Das Problem der Bewegungsplanung wird mithilfe einer Suche in diesen einfacheren Strukturen gelöst.
- Ein Pfad, der von einem Suchalgorithmus gefunden wird, kann ausgeführt werden, indem der Pfad als Referenz-Bewegungsverlauf für einen **PID-Regler** verwendet wird. Regler sind in der Robotik notwendig, um kleine Störungen auszugleichen; Pfadplanung allein ist normalerweise unzureichend.
- **Potentialfeldtechniken** navigieren Roboter mithilfe von Potentialfunktionen, die über die Distanz zu Hindernissen und die Zielposition definiert sind. Potentialfeldtechniken können in lokalen Minima stecken bleiben, aber sie können die Bewegungen direkt erzeugen, ohne eine Pfadplanung vornehmen zu müssen.
- Manchmal ist es einfacher, einen Roboter-Regler direkt zu spezifizieren, anstatt einen Pfad von einem expliziten Modell der Umgebung abzuleiten. Solche Regler können häufig als einfache **endliche Automaten** formuliert werden.
- Für den Softwareentwurf sind verschiedene Architekturen entwickelt worden. Die **Subsumptions-Architektur** ermöglicht es den Programmierern, Roboter-Controller aus miteinander verbundenen endlichen Automaten zusammenzusetzen. **Drei-Schichten-Architekturen** sind gebräuchliche Frameworks für die Entwicklung von Roboter-Software, die Deliberation, Sequenzierung von Unterzielen und Steuerung integrieren. Die verwandte **Pipeline-Architektur** verarbeitet Daten parallel über eine Sequenz von Modulen, die Wahrnehmung, Modellierung, Planung, Steuerung und Roboterschnittstellen entsprechen.

Übungen zu Kapitel 25

Lösungshinweise

1 Die Monte-Carlo-Lokalisierung weist für jede endliche Stichprobengröße *einen Bias* auf – d.h., der erwartete Wert der durch den Algorithmus berechneten Positionen unterscheidet sich von dem tatsächlich erwarteten Wert –, was mit der Arbeitsweise des Partikelfilters zusammenhängt. In dieser Frage werden Sie aufgefordert, diesen Bias zu quantifizieren.

Um das Ganze zu vereinfachen, betrachten Sie eine Welt mit vier möglichen Roboterpositionen: $X = \{x_1, x_2, x_3, x_4\}$. Anfänglich entnehmen wir gleichmäßig verteilt von diesen Positionen $N \geq 1$ Stichproben. Wie üblich ist es kein Problem, wenn mehrere Stichproben für eine der Positionen X ermittelt werden. Sei Z eine boolesche Sensorvariable, die durch die folgenden bedingten Wahrscheinlichkeiten charakterisiert ist:

$$P(z \mid x_1) = 0{,}8 \qquad P(\neg z \mid x_1) = 0{,}2$$
$$P(z \mid x_2) = 0{,}4 \qquad P(\neg z \mid x_2) = 0{,}6$$
$$P(z \mid x_3) = 0{,}1 \qquad P(\neg z \mid x_3) = 0{,}9$$
$$P(z \mid x_4) = 0{,}1 \qquad P(\neg z \mid x_4) = 0{,}9.$$

MCL verwendet diese Wahrscheinlichkeiten, um Partikelgewichtungen zu erzeugen, die anschließend normalisiert und im Resampling-Prozess verwendet werden. Der Einfachheit halber wollen wir annehmen, dass wir unabhängig von N nur eine neue Stichprobe im Resampling-Prozess erzeugen. Diese Stichprobe könnte jeder der vier Positionen in X entsprechen. Der Stichprobenprozess definiert also eine Wahrscheinlichkeitsverteilung über X.

a. Wie sieht die resultierende Wahrscheinlichkeitsverteilung über X für diese neue Stichprobe aus? Beantworten Sie diese Frage separat für $N = 1, \ldots, 10$ und für $N = \infty$.

b. Die Differenz zwischen beiden Wahrscheinlichkeitsverteilungen P und Q kann durch die KL-Divergenz gemessen werden, die definiert ist als:

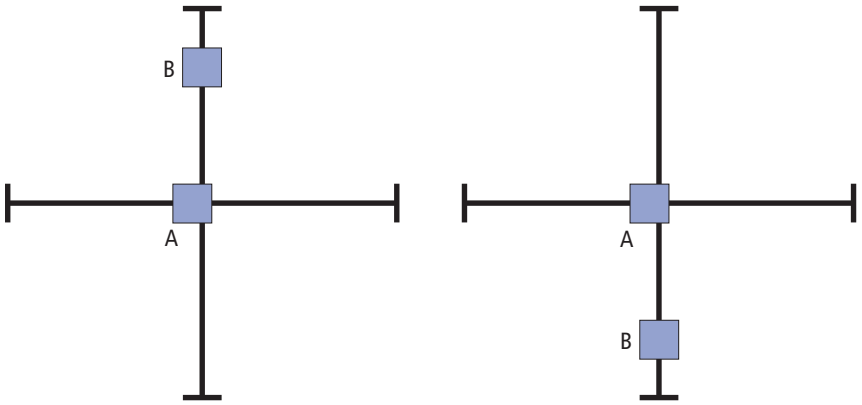
$$KL(P,Q) = \sum_i P(x_i) \log \frac{P(x_i)}{Q(x_i)} .$$

Wie lauten die KL-Divergenzen zwischen den Verteilungen in (a) und der tatsächlichen A-posteriori-Verteilung?

c. Welche Änderungen der Problemformulierung (nicht des Algorithmus!) würden garantieren, dass der oben gezeigte Schätzer selbst für endliche Werte von N keinen Bias hat? Schlagen Sie mindestens zwei solcher Änderungen vor (die jeweils genügend sein sollten).

2 Implementieren Sie die Monte-Carlo-Lokalisierung für einen simulierten Roboter mit Bereichssensoren. Eine Rasterkarte und Entfernungsdaten stehen online unter *aima.cs.berkeley.edu* zur Verfügung. Ihre Übung ist erledigt, wenn Sie die globale Lokalisierung durch Roboter erfolgreich demonstrieren können.

3 Betrachten Sie einen Roboter mit zwei einfachen Manipulatoren, wie ihn ▶ Abbildung 25.31 zeigt. Manipulator A ist ein quadratischer Block mit der Seitenlänge 2, der an einem Stab entlang der x-Achse von $x = -10$ bis $x = 10$ zurück und vor gleiten kann. Manipulator B ist ein quadratischer Block der Seitenlänge 2, der auf einem Stab entlang der y-Achse von $y = -10$ bis $y = 10$ zurück und vor gleiten kann. Die Stäbe liegen außerhalb der Manipulationsebene, sodass die Stäbe die Bewegung der Blöcke nicht behindern. Eine Konfiguration ist dann ein Paar x, y, wobei x die x-Koordinate des Mittelpunktes von Manipulator A und y die y-Koordinate des Mittelpunktes von Manipulator B ist. Zeichnen Sie für diesen Roboter den Konfigurationsraum und geben Sie die zulässigen und ausgeschlossenen Zonen an.



Startkonfiguration <–0,5, 7> Endkonfiguration <–0,5, –7>

Abbildung 25.31: Ein Robotermanipulator in zwei seiner möglichen Konfigurationen.

4 Betrachten Sie den in Abbildung 25.14 gezeigten Roboterarm. Gehen Sie davon aus, dass das Grundelement des Roboters 70 cm lang ist und dass sein Oberarm und sein Unterarm je 50 cm lang sind. Wie in *Abschnitt 25.4.1* erläutert, ist die inverse Kinematik eines Roboters häufig nicht eindeutig. Geben Sie eine explizite geschlossene Lösung der inversen Kinematik für diesen Arm an. Unter welchen Bedingungen ist die Lösung eindeutig?

5 Implementieren Sie einen Algorithmus für die Berechnung des Voronoi-Diagramms einer beliebigen 2D-Umgebung, beschrieben durch ein n x n großes boolesches Array. Stellen Sie den Algorithmus grafisch dar, indem Sie das Voronoi-Diagramm für zehn interessante Karten ausgeben. Welche Komplexität hat Ihr Algorithmus?

6 Diese Übung untersucht die Beziehung zwischen Arbeitsraum und Konfigurationsraum unter Verwendung der in ▶ Abbildung 25.32 gezeigten Beispiele.

a. Betrachten Sie die in Abbildung 25.32(a) bis (c) gezeigten Roboterkonfigurationen und ignorieren Sie das Hindernis, das in jedem der Diagramme dargestellt ist. Zeichnen Sie die entsprechenden Armkonfigurationen im Konfigurationsraum. (*Hinweis*: Jede Armkonfiguration wird auf einen einzelnen Punkt im Konfigurationsraum abgebildet, wie in Abbildung 25.14(b) gezeigt.)

b. Zeichnen Sie den Konfigurationsraum für jedes der in Abbildung 25.32(a) bis (c) gezeigten Arbeitsraum-Diagramme. (*Hinweis*: Die Konfigurationsräume haben mit dem in Abbildung 25.32?(a) gezeigten Konfigurationsraum den Bereich gemeinsam, der der Eigenkollision entspricht, aber es entstehen Differenzen, weil umschließende Hindernisse fehlen und in diesen einzelnen Abbildungen die Hindernisse an verschiedenen Positionen liegen.)

c. Zeichnen Sie für jeden der schwarzen Punkte in Abbildung 25.32(e) bis (f) die entsprechenden Konfigurationen des Roboterarmes im Arbeitsraum. Ignorieren Sie in dieser Übung die schattierten Bereiche.

d. Die in Abbildung 25.32(e) bis (f) gezeigten Konfigurationsräume wurden alle von einem einzigen Arbeitsraumhindernis (dunkle Schattierung) sowie den aus der Eigenkollisionsbedingung (helle Schattierung) entstandenen Bedingungen erzeugt. Zeichnen Sie für jedes Diagramm das Arbeitsraumhindernis, das dem dunkel schattierten Bereich entspricht.

e. Abbildung 25.32(d) zeigt, dass ein einziges planares Hindernis den Arbeitsraum in zwei nicht mehr verbundene Bereiche zerlegen kann. Wie groß ist die maximale Anzahl der nicht verbundenen Bereiche, die erzeugt werden kann, indem man ein planares Hindernis in einen hindernisfreien, verbundenen Arbeitsraum für einen Roboter mit zwei Freiheitsgraden einfügt? Geben Sie ein Beispiel an und erklären Sie, warum sich keine größere Anzahl von nicht verbundenen Bereichen erzeugen lässt. Wie verhält es sich bei einem nicht planaren Hindernis?

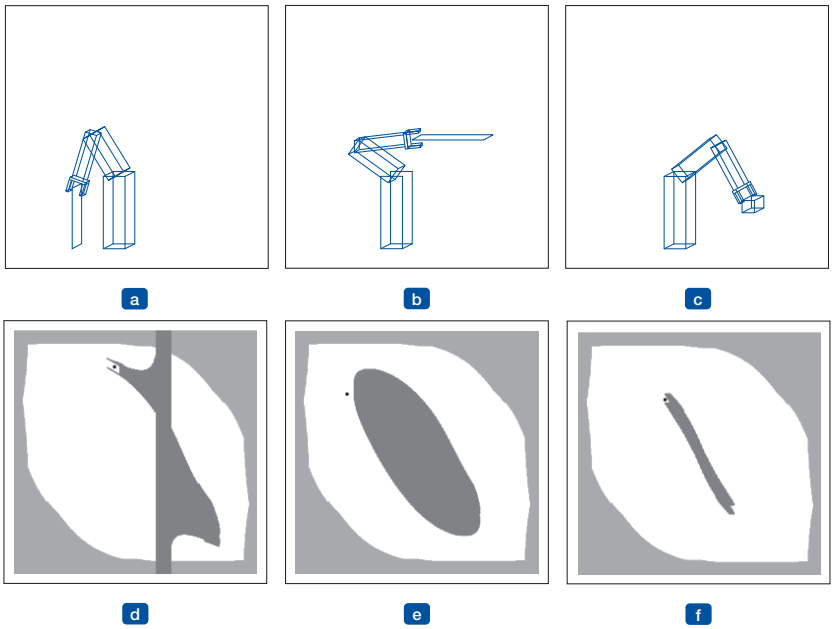


Abbildung 25.32: Diagramme für Übung 25.5.

7 Betrachten Sie einen mobilen Roboter, der sich auf einer waagerechten Oberfläche bewegt. Nehmen Sie an, dass der Roboter die beiden folgenden Bewegungsarten ausführen kann:

- Um eine angegebene Entfernung nach vorn rollen
- Um einen angegebenen Winkel am Ort drehen

Der Zustand eines derartigen Roboters lässt sich in Form dreier Parameter charakterisieren: $\langle x, y, \phi \rangle$, die x- und y-Koordinate des Roboters (genauer gesagt, seine Drehachse) und seine Ausrichtung, ausgedrückt als Winkel bezüglich der positiven x-Richtung. Die Aktion „$Roll(D)$" ändert den Zustand $\langle x, y, \phi \rangle$ in $\langle x + D\cos(\phi), y + D\sin(\phi), \phi \rangle$ und die Aktion $Rotate(\theta)$ ändert den Zustand $\langle x, y, \phi \rangle$ in $\langle x, y, \phi + \theta \rangle$.

a. Nehmen Sie an, dass sich der Roboter anfangs bei $\langle 0, 0, 0 \rangle$ befindet und dann die Aktionen $Rotate(60°)$, $Roll(1)$, $Rotate(25°)$ und $Roll(2)$ ausführt. Wie sieht der Endzustand des Roboters aus?

b. Nehmen Sie nun an, dass der Roboter keine perfekte Kontrolle seiner eigenen Drehung besitzt und dass er sich beim Versuch, eine Drehung um θ auszuführen, tatsächlich um einen Winkel zwischen $\theta - 10°$ und $\theta + 10°$ dreht. In diesem Fall gibt es einen Bereich möglicher Endzustände, wenn der Roboter versucht, die Sequenz der Aktionen in (A) auszuführen. Wie groß sind die minimalen und maximalen Werte der x-Koordinate, der y-Koordinate und der Ausrichtung im Endzustand?

c. Das Modell in (B) ändern wir nun in ein probabilistisches Modell. Dann folgt der tatsächliche Winkel der Drehung einer Gaußverteilung mit dem Mittelwert θ und der Standardabweichung 10°, wenn der Roboter eine Drehung um θ versucht. Angenommen, der Roboter führt die Aktionen $Rotate(90°)$ und $Roll(1)$ aus. Geben Sie ein einfaches Argument an, dass (a) der erwartete Wert der Position am Ende ungleich dem Ergebnis einer Drehung von genau 90° und einem anschließenden Vorwärtsrollen um 1 Einheit ist und (b) dass die Verteilung der Positionen am Ende keiner Gaußverteilung folgt. (Versuchen Sie nicht, den wahren Mittelwert oder die wahre Verteilung zu berechnen.)

Diese Übung sollte zeigen, dass sich die Drehungsunsicherheit schnell zu einer größeren Positionsunsicherheit auswächst und die Beschäftigung mit Drehungsunsicherheit mühevoll ist, egal ob Unsicherheit in Form von festen Intervallen oder probabilistisch behandelt wird. Das hängt mit dem Umstand zusammen, dass die Beziehung zwischen Ausrichtung und Position sowohl nichtlinear als auch nichtmonoton ist.

8 Betrachten Sie den in ▶ Abbildung 25.33 gezeigten vereinfachten Roboter. Angenommen, die kartesischen Koordinaten des Roboters sind jederzeit bekannt, ebenso wie die seiner Zielposition. Dagegen sind die Positionen der Hindernisse unbekannt. Der Roboter kann Hindernisse in seiner unmittelbaren Nähe erkennen, wie in dieser Abbildung gezeigt. Der Einfachheit halber wollen wir annehmen, dass die Bewegung des Roboters nicht durch Rauschen gestört wird und dass der Zustandsraum diskret ist. Abbildung 25.33 ist nur ein Beispiel; in dieser Übung sollen Sie alle möglichen Rasterwelten mit gültigem Pfad von der Start- zur Zielposition berücksichtigen.

a. Entwerfen Sie einen deliberativen Regler, der garantiert, dass der Roboter seine Zielposition immer erreicht, wenn dies möglich ist. Der deliberative Regler kann sich Messungen in Form einer Karte merken, die angefordert wird, wenn sich der Roboter bewegt. Zwischen den einzelnen Bewegungen kann er eine beliebige Zeit mit Nachdenken verbringen (Deliberation).

b. Jetzt entwerfen Sie einen *reaktiven* Regler für dieselbe Aufgabe. Dieser Regler kann sich die letzten Sensormessungen nicht merken (er kann keine Karte erzeugen!). Stattdessen muss er alle Entscheidungen auf der aktuellen Messung basierend treffen, die das Wissen über seine eigene Position sowie die des Zieles beinhaltet. Die Zeit für die Entscheidungsfindung muss unabhängig von der Umgebungsgröße und der Anzahl der vergangenen Zeitschritte sein. Wie groß ist die maximale Anzahl der Schritte, die es dauern kann, bis Ihr Roboter am Ziel eintrifft?

c. Welche Leistung erbringen Ihre Controller aus (a) und (b), wenn eine der folgenden sechs Bedingungen gilt: stetiger Zustandsraum, Rauschen in der Wahrnehmung, Rauschen in der Bewegung, Rauschen sowohl in Wahrnehmung als auch in Bewegung, unbekannte Zielposition (das Ziel kann nur erkannt werden, wenn es sich innerhalb des Sensorbereiches befindet) oder sich bewegende Hindernisse. Nennen Sie für jede Bedingung und jeden Controller ein Beispiel für eine Situation, wo der Roboter fehlschlägt (oder erklären Sie, warum dies nicht der Fall sein kann).

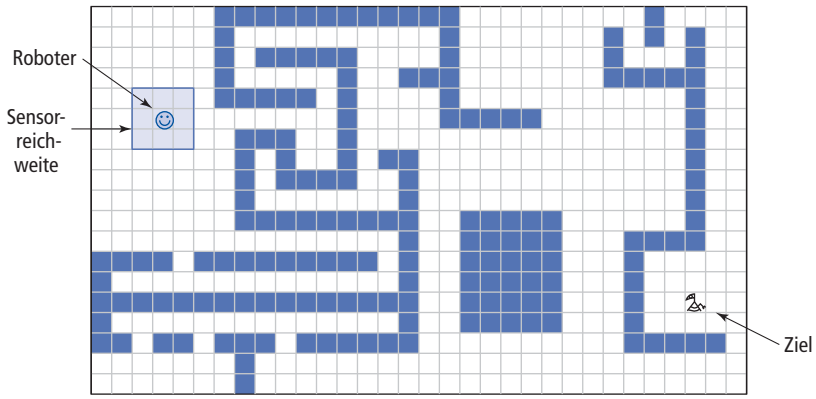


Abbildung 25.33: Vereinfachte Roboter in einem Irrgarten. Siehe Übung 25.8.

9 In Abbildung 25.24(b) haben wir einen erweiterten endlichen Automaten für die Steuerung eines einzelnen Beines eines Hexapod-Roboters gesehen. In dieser Übung wollen wir einen AFSM entwerfen, der bei einer Kombination mit sechs Kopien der einzelnen Bein-Controller zu einer effizienten, stabilen Fortbewegung führt. Für diesen Zweck müssen Sie die einzelnen Bein-Controller so erweitern, dass sie Nachrichten an Ihren neuen AFSM weitergeben und warten, bis andere Nachrichten eintreffen. Erklären Sie, warum Ihr Controller effizient ist, weil er nicht unnötig Energie verschwendet (z.B. durch Rutschen der Beine), und dass er den Roboter mit ausreichend hoher Geschwindigkeit antreibt. Beweisen Sie, dass Ihr Controller die in *Abschnitt 25.2.2* beschriebene Stabilitätsbedingung erfüllt.

10 (Diese Übung wurde zuerst von Michael Genesereth und Nils Nilsson entwickelt. Sie kann für Studenten sowohl im Grund- als auch im Hauptstudium eingesetzt werden.) Menschen sind so an einfache Aufgaben im Haushalt gewöhnt, dass sie häufig vergessen, wie kompliziert diese Aufgaben sind. In dieser Übung werden Sie die Komplexität erkennen und die letzten 30 Jahre der Entwicklung in der Robotik rekapitulieren. Als Erstes ist die Aufgabe zu lösen, einen Bogen aus drei Bausteinen zu bauen. Simulieren Sie wie folgt einen Roboter mit vier Menschen:

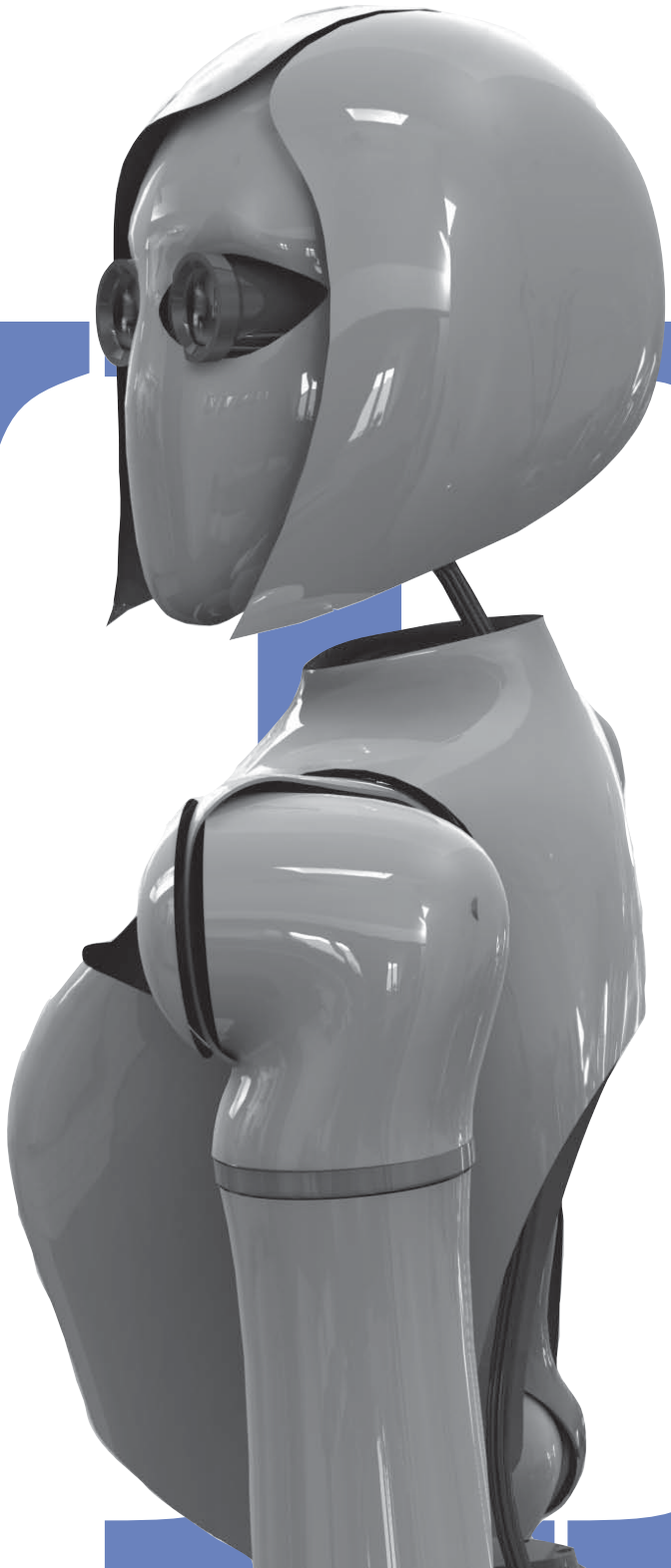
Gehirn: Das Gehirn lenkt die Hände bei der Ausführung eines Planes, um das Ziel zu erreichen. Es erhält Eingaben von den Augen, aber *kann die Szene nicht direkt sehen*. Das Gehirn ist der Einzige, der weiß, um was es sich bei dem Ziel handelt.

Augen: Die Augen melden dem Gehirn eine kurze Beschreibung der Szene: „Auf einer grünen Schachtel steht eine rote Schachtel, die auf der Seite liegt." Augen können auch Fragen vom Gehirn beantworten, wie beispielsweise: „Besteht eine Lücke zwischen der linken Hand und der roten Schachtel?" Wenn Sie eine Videokamera haben, richten Sie sie auf die Szene und erlauben den Augen, durch den Sucher der Videokamera zu sehen, aber nicht direkt auf die Szene.

Linke Hand und Rechte Hand: Eine Person spielt jeweils eine Hand. Die beiden Hände stehen nebeneinander und tragen jeweils einen Topflappen an einer Hand. Die Hände führen nur einfache Befehle vom Gehirn aus – z.B.: „Linke Hand, bewege dich um 5 cm nach vorn." Sie können ausschließlich Bewegungsbefehle ausführen; beispielsweise kann man ihnen nicht befehlen: „Hebe die Schachtel auf." Die Hände müssen *blind* agieren können. Ihre einzige Sensorfähigkeit besteht darin, zu erkennen, wann ihr Pfad durch ein unbewegliches Hindernis wie beispielsweise einen Tisch oder die andere Hand blockiert ist. In solchen Fällen können sie einen Ton abgeben, um das Gehirn über die Schwierigkeit zu informieren.

TEIL VII

Schlussfolgerungen



Philosophische Grundlagen

26

ÜBERBLICK

In diesem Kapitel betrachten wir, was es bedeutet zu denken und ob künstliche Maschinen dies können und sollten.

Philosophen gibt es schon sehr viel länger als Computer und sie haben versucht, Fragen zu lösen, die mit der KI zu tun haben: Wie arbeitet ein Gehirn? Ist es möglich, dass Maschinen so intelligent agieren, wie es Menschen tun, und wenn dies der Fall ist, haben sie dann ein wirkliches Bewusstsein? Welche ethischen Implikationen hat das Konzept der intelligenten Maschinen?

Zuerst ein bisschen Terminologie: Die Behauptung, dass Maschinen agieren könnten, *als ob* sie intelligent wären, nennen Philosophen die **schwache KI**-Hypothese, und die Behauptung, dass Maschinen, die das tun, *wirklich* denken (und nicht einfach Denken *simulieren*) die **starke KI**-Hypothese.

Die meisten KI-Forscher nehmen die schwache KI-Hypothese als gegeben hin und kümmern sich nicht um die starke KI-Hypothese – solange ihre Programme funktionieren, ist es ihnen egal, ob sie es als Simulation der Intelligenz oder als echte Intelligenz bezeichnen. KI-Forscher sollten sich jedoch auch mit den ethischen Auswirkungen ihrer Arbeit beschäftigen.

26.1 Schwache KI: Können Maschinen intelligent handeln?

Der Vorschlag für den Sommerworkshop 1956, der das Gebiet der künstlichen Intelligenz definiert hat (McCarthy et al., 1955), stellte die Behauptung auf: „Jeder Aspekt des Lernens und andere Eigenschaften von Intelligenz können im Prinzip so beschrieben werden, dass eine Maschine sie simulieren kann." Somit gründete die KI auf der Behauptung, dass schwache KI möglich ist. Andere dagegen haben behauptet, dass schwache KI unmöglich ist: „Künstliche Intelligenz, wie sie im Kult der Computerisierung verfolgt wird, hat nicht einmal den Hauch einer Chance, dauerhafte Ergebnisse zu erzielen." (Sayre, 1993)

Ob KI wirklich unmöglich ist, hängt offensichtlich davon ab, wie man sie definiert. In *Abschnitt 1.1* haben wir KI als die Suche nach dem besten Agentenprogramm auf einer bestimmten Architektur beschrieben. Mit dieser Formulierung ist KI der Definition nach möglich: Für jede digitale Architektur mit k Bit Programmspeicher gibt es genau 2^k Agentenprogramme und um das beste davon zu finden, müssen wir sie einfach alle auflisten und testen. Für große k ist das vielleicht nicht möglich, aber Philosophen beschäftigen sich mit der Theorie, nicht mit der Praxis.

Unsere Definition der KI funktioniert ausreichend für das Ingenieurproblem, für eine bestimmte Architektur einen guten Agenten zu finden. Aus diesem Grund sind wir versucht, diesen Abschnitt sofort zu beenden und die eingangs gestellte Frage zu bestätigen. Die Philosophen sind jedoch an dem Problem interessiert, zwei Architekturen zu vergleichen – Mensch und Maschine. Darüber hinaus haben sie traditionell die Frage nicht als Maximierung des erwarteten Nutzens gestellt, sondern als „**Können Maschinen denken?**".

Der Informatiker Edsger Dijkstra (1984) sagte: „Die Frage, ob *Maschinen denken können*, ist genauso interessant wie die Frage, ob *U-Boote schwimmen können*." Das American Heritage Dictionary gibt als erste Definition für *Schwimmen* an, „sich durch das oder auf dem Wasser mithilfe von Gliedmaßen, Flossen oder eines Schwanzes fortzubewegen" und die meisten Menschen werden damit übereinstimmen, dass U-Boote, die beinlos sind, nicht schwimmen können. Das Wörterbuch definiert auch

Fliegen als „sich durch die Luft mithilfe von Flügeln oder flügelartigen Teilen fortzubewegen“ und die meisten Leute werden zustimmen, dass Flugzeuge, die flügelartige Teile haben, fliegen können. Dennoch haben weder die Fragen noch die Antworten irgendeine Relevanz für den Entwurf oder die Fähigkeiten von Flugzeugen oder U-Booten, sondern vielmehr damit, wie wir diese Wörter im Englischen verwenden. (Der Umstand, dass Schiffe im Russischen *schwimmen*, verstärkt nur diesen Punkt.) Die praktische Möglichkeit „denkender Maschinen“ begleitet uns erst seit 50 Jahren, was nicht lange genug für die englische Sprache ist, sich auf eine Bedeutung für das Wort „denken“ einzustellen – erfordert es „ein Gehirn“ oder lediglich „gehirnartige Teile“?

Alan Turing schlug in seiner berühmten Arbeit „Computing Machinery and Intelligence" (Turing, 1950) vor, dass wir nicht fragen sollten, ob Maschinen denken können, sondern, ob Maschinen einen Verhaltensintelligenztest bestehen können, der heute auch als **Turing-Test** bezeichnet wird. Bei dem Test muss ein Programm über fünf Minuten eine Unterhaltung (über online eingegebene Nachrichten) mit einem Gesprächspartner führen. Der Gesprächspartner muss dann entscheiden, ob die Kommunikation mit einem Programm oder mit einer Person geführt wurde; das Programm besteht den Test, wenn es den Gesprächspartner in 30% der Fälle täuschen kann. Turing sagte voraus, dass im Jahr 2000 ein Computer mit einem Speicher von 10^9 Einheiten so gut programmiert werden könne, dass er den Test besteht. Er lag falsch – die Programme sind noch nicht so weit.

Andererseits *wurden* viele Menschen getäuscht, wenn sie nicht wussten, dass sie sich mit einem Computer unterhalten. So haben das Programm ELIZA und Internet-Chatbots wie zum Beispiel MGONZ (Humphrys, 2008) und NATACHATA wiederholt ihre Gesprächspartner getäuscht und der Chatbot CYBERLOVER hat die Aufmerksamkeit der Gesetzeshüter auf sich gezogen, weil er die Chatpartner dazu verleitet hat, persönliche Daten preiszugeben, sodass sich deren Identität stehlen lässt. Der jährlich stattfindende Wettbewerb um den seit 1991 ausgeschriebenen Loebner-Preis ist der am längsten laufende Wettbewerb in der Art eines Turing-Tests. Die Wettbewerbe haben zu besseren Modellen für menschliche Tippfehler geführt.

Turing selbst untersuchte eine große Anzahl möglicher Einwände gegen die Möglichkeit intelligenter Maschinen, einschließlich fast all derer, die in dem halben Jahrhundert seit Veröffentlichung seiner Arbeit wirklich vorgebracht wurden. Wir werden einige davon genauer betrachten.

26.1.1 Das Argument „... aus Unfähigkeit“

Das Argument „... aus Unfähigkeit“ behauptet, dass „eine Maschine nie *X* tun kann“. Als Beispiel für *X* listet Turing Folgendes auf:

> *Freundlich, einfallsreich, schön, nett, initiativ sein, Humor haben, richtig von falsch unterscheiden, Fehler machen, verlieben, Erdbeereis mit Sahne mögen, sich in etwas verlieben, aus Erfahrungen lernen, Wörter korrekt anwenden, Thema der eigenen Gedanken sein, die Vielfalt der Verhaltensweisen normaler Menschen aufweisen, etwas wirklich Neues tun.*

Rückblickend sind einige dieser Dinge ziemlich leicht – wir kennen alle Computer, die „Fehler machen“. Wir sind auch vertraut mit einer jahrhundertealten Technik, die über eine bewiesene Fähigkeit verfügt, um „sich in etwas zu verlieben“ – den Teddybär. Computerschachexperte David Levy sagt voraus, dass sich die Menschen um das Jahr 2050 regelmäßig in humanoide Roboter verlieben werden (Levy, 2007). Und dass sich

ein Roboter verliebt, ist zwar ein häufiges Thema in der Fiktion,[1] doch gibt es bisher kaum Erwartungen, ob dies in der Tat wahrscheinlich ist (Kim et al., 2007). Programme spielen Schach, Dame und andere Spiele, inspizieren Teile auf Montagebändern, steuern Autos und Hubschrauber, diagnostizieren Krankheiten und machen Hunderte von anderen Dingen genauso gut wie oder besser als Menschen. Computer haben kleine, aber wesentliche Entdeckungen in Astronomie, Mathematik, Chemie, Mineralogie, Biologie, Informatik und anderen Bereichen gemacht. Für all diese Entdeckungen war eine Leistung auf dem Niveau eines menschlichen Experten erforderlich.

Mit unserem heutigen Wissen über Computer ist es nicht erstaunlich, dass sie gut für kombinatorische Probleme geeignet sind, wie beispielsweise für das Schachspielen. Algorithmen bieten aber auch eine gute Leistung auf dem Niveau des Menschen für Aufgaben, für die scheinbar eine menschliche Beurteilung erforderlich ist oder, wie Turing es ausgedrückt hat, ein „Lernen aus Erfahrung“ sowie die Fähigkeit, „richtig von falsch zu unterscheiden“. Bereits 1955 untersuchte Paul Meehl (siehe auch Grove und Meehl, 1996) die Entscheidungsfindungsprozesse trainierter Experten für subjektive Aufgaben wie beispielsweise die Vorhersage des Erfolges eines Studenten in einem Trainingsprogramm oder des Rückfalls eines Kriminellen. In 19 von 20 Studien, die er durchführte, stellte Meehl fest, dass einfache statistische Lernalgorithmen (wie beispielsweise lineare Regression oder naives Bayes) eine bessere Vorhersage treffen als die Experten. Der Educational Testing Service verwendete ein automatisiertes Programm, um Millionen von Fragen für das GMAT-Examen seit 1999 zu bewerten. Das Programm stimmt in 97% der Fälle mit den menschlichen Korrektoren überein, etwa im selben Grad, wie zwei menschliche Korrektoren übereinstimmen (Burstein et al., 2001).

Es ist offensichtlich, dass Computer viele Dinge genauso gut oder besser als Menschen können, unter anderem auch Dinge, von denen die Menschen glauben, es wäre sehr viel Verstand und Wissen für die Ausführung erforderlich. Das bedeutet nicht, dass die Computer für die Ausführung dieser Aufgaben Verstand und Wissen einsetzen – sie sind nicht Teil des *Verhaltens* und wir werden diese Fragen an anderer Stelle beantworten –, der Punkt ist jedoch, dass die erste Einschätzung mentaler Prozesse, die erforderlich sind, um ein bestimmtes Verhalten zu erzeugen, häufig falsch ist. Außerdem gibt es natürlich noch viele Aufgaben, für die die Computer noch nicht geeignet sind (gelinde gesagt), einschließlich der Aufgabe von Turing, eine offene Konversation zu führen.

26.1.2 Der mathematische Einwand

Durch die Arbeiten von Turing (1936) und Gödel (1931) ist bekannt, dass bestimmte mathematische Fragen durch bestimmte formale Systeme im Prinzip nicht beantwortbar sind. Das Unvollständigkeitstheorem (siehe *Abschnitt 9.5*) von Gödel ist das bekannteste Beispiel dafür. Kurz gesagt, für jedes formale axiomatische System F, das in der Lage ist, Arithmetik auszuführen, ist es möglich, einen sogenannten „Gödel-Satz“ $G(F)$ mit den folgenden Eigenschaften zu konstruieren:

- $G(F)$ ist ein Satz von F, kann aber nicht innerhalb von F bewiesen werden.
- Wenn F konsistent ist, dann ist $G(F)$ wahr.

1 Zum Beispiel in der Oper Coppélia (1870), im Roman *Do Androids Dream of Electric Sheep?* (1968), in den Filmen *AI* (2001) und *Wall-E* (2008) und im Song *Let's Do It: Let's fall in Love*, der in der Version von Noel Coward aus dem Jahr 1955 voraussagte: „Wahrscheinlich leben wir, um Maschinen zu sehen, die es tun.“ Er tat es nicht.

Philosophen wie beispielsweise J.R. Lucas (1961) haben behauptet, dieses Theorem zeige, dass Maschinen den Menschen mental unterlegen sind, weil Maschinen formale Systeme sind, die durch das Unvollständigkeits-Theorem beschränkt sind – sie können nicht den Nachweis ihres eigenen Gödel-Satzes erbringen –, während Menschen keine solche Einschränkung aufweisen. Diese Behauptung verursachte jahrzehntelange Debatten, die sich auch in der umfangreichen Literatur niederschlagen, unter anderem in zwei Büchern des Mathematikers Sir Roger Penrose (1989, 1994), die die Behauptungen mit einigen neueren Aspekten (wie beispielsweise der Hypothese, dass sich Menschen unterscheiden, weil ihre Gehirne mit Quantenschwerkraft arbeiten) wiederholen. Wir werden nur drei der Probleme bei dieser Behauptung betrachten.

Erstens bezieht sich das Unvollständigkeits-Theorem von Gödel nur auf formale Systeme, die leistungsfähig genug sind, Arithmetik auszuführen. Dies sind unter anderem Turing-Maschinen und die Behauptung von Lucas basiert zum Teil auf der Zusicherung, dass Computer Turing-Maschinen sind. Das ist zwar eine gute Annäherung, trifft aber nicht ganz zu. Turing-Maschinen sind unendlich, während Computer endlich sind, und jeder Computer kann deshalb als (sehr großes) System in der Aussagenlogik beschrieben werden, die nicht Thema des Unvollständigkeits-Theorems von Gödel ist. Zweitens sollte es einem Agenten nicht allzu unangenehm sein, dass er die Wahrheit eines Satzes nicht nachweisen kann, während andere das können. Betrachten Sie den folgenden Satz:

J. R. Lucas kann nicht konsistent zusichern, dass dieser Satz wahr ist.

Hätte Lucas diesen Satz zugesichert, würde er sich selbst widersprechen; deshalb kann Lucas ihn nicht konsistent zusichern, also muss er wahr sein. Wir haben damit gezeigt, dass es einen Satz gibt, den Lucas nicht konsistent zusichern kann, während andere Menschen (und Maschinen) dies können. Damit halten wir jedoch nicht weniger von Lucas. Ein weiteres Beispiel: Kein Mensch kann in seinem ganzen Leben die Summe von zehn Milliarden zehnstelligen Zahlen berechnen, während ein Computer das in wenigen Sekunden erledigen kann. Dennoch betrachteten wir dies nicht als grundlegende Einschränkung der Denkfähigkeit des Menschen. Menschen verhalten sich seit Tausenden von Jahren intelligent, und zwar noch bevor sie die Mathematik erfanden; deshalb ist es unwahrscheinlich, dass formales mathematisches Schließen mehr als eine Nebenrolle bei dem Konzept spielt, das als Intelligenz bezeichnet wird.

Drittens und am wichtigsten: Selbst wenn wir zugeben, dass Computer Einschränkungen dahingehend haben, was sie beweisen können, gibt es keine Nachweise, dass Menschen immun gegen diese Einschränkungen sind. Man kann nur zu einfach zeigen, dass ein formales System X nicht erledigen kann, während Menschen X unter Verwendung ihrer eigenen informellen Methoden erledigen *können*, ohne einen Beweis für diese Behauptung angeben zu müssen. Tatsächlich ist es unmöglich zu *beweisen*, dass Menschen nicht dem Unvollständigkeits-Theorem von Gödel unterliegen, weil jeder strenge Beweis eine Formalisierung des behaupteten nicht formalisierbaren menschlichen Talents erfordern würde und sich damit selbst widerspräche. Wir bleiben also zurück mit dem Appell an die Intuition, dass Menschen irgendwie übermenschliche Kraftakte für mathematische Einsichten ausführen können. Dieser Appell wird ausgedrückt durch Argumente wie beispielsweise „Wir müssen unsere eigene Konsistenz annehmen, wenn Denken überhaupt möglich ist“ (Lucas, 1976). Wenn man überhaupt etwas weiß, dann, dass Menschen inkonsistent sind. Das trifft in jedem Fall für das tägliche Schließen zu, aber auch für sorgfältige mathematische Überlegungen. Ein berühmtes Beispiel ist das Vier-Farben-Kartenproblem. Alfred Kampe veröffentlichte 1879 einen Beweis, der allgemein anerkannt

wurde und zu seiner Wahl als Mitglied der Royal Society beitrug. 1890 wies jedoch Pearcy Heawood auf einen Fehler hin und das Theorem blieb bis 1977 unbewiesen.

26.1.3 Das Argument der Formlosigkeit

Einer der einflussreichsten und beständigsten Kritikpunkte der KI als Unternehmen stammt von Turing: das „Argument der Formlosigkeit des Verhaltens“. Im Wesentlichen handelt es sich dabei um die Behauptung, dass das menschliche Verhalten viel zu komplex ist, als dass es durch eine einfache Regelmenge abgedeckt werden könnte, und weil Computer nichts weiter tun, als Regelmengen zu folgen, können sie kein Verhalten erzeugen, das so intelligent wie das der Menschen ist. Die Unfähigkeit, alles in einer Menge logischer Regeln auszudrücken, wird auch als **Qualifizierungsproblem** der KI bezeichnet (siehe *Kapitel 10*).

Der wichtigste Vertreter dieser Ansicht war der Philosoph Hubert Dreyfus, der eine Folge einflussreicher Kritikpunkte an der künstlichen Intelligenz veröffentlicht hat: *What Computers Can't Do* (1972), die Fortsetzung *What Computers Still Can't Do* (1992) und, zusammen mit seinem Bruder Stuart, *Mind Over Machine* (1986).

Die Position, die sie kritisieren, wird auch als „Good Old-Fashioned AI“ oder GOFAI bezeichnet, ein vom Philosophen John Haugeland (1985) geprägter Begriff. GOFAI behauptet angeblich, dass jedes intelligente Verhalten von einem System dargestellt werden kann, das logisch aus einer Menge von Fakten und Regeln schließt, die die Domäne beschreiben. Deshalb entspricht es dem einfachsten logischen Agenten, der in *Kapitel 7* beschrieben wurde. Dreyfus hat recht mit seiner Aussage, dass logische Agenten empfindlich gegenüber dem Qualifizierungsproblem sind. Wie wir in *Kapitel 13* gesehen haben, sind probabilistische Inferenzsysteme für offene Domänen viel besser geeignet. Die Dreyfus-Kritik hat sich also nicht gegen Computer *per se* gerichtet, sondern gegen eine bestimmte Art, sie zu programmieren. Man kann jedoch durchaus annehmen, dass ein Buch namens *What First-Order Logical Rule-Based Systems Without Learning Can't Do* vielleicht weniger Einfluss gehabt hätte.

Nach Ansicht von Dreyfus beinhaltet die menschliche Erfahrung kein Wissen über bestimmte Regeln, sondern nur einen „gesamtheitlichen Kontext“ oder „Hintergrund“, mit dem Menschen arbeiten. Hier nennt er als Beispiel für ein geeignetes Sozialverhalten das Verschenken und Empfangen von Geschenken: „Normalerweise reagiert man einfach in entsprechenden Situationen durch die Übergabe eines geeigneten Geschenks. Man hat offensichtlich ‚einen direkten Sinn‘ dafür, wie Dinge gemacht werden und was erwartet wird.“ Dieselbe Behauptung wird im Kontext des Schachspieles getroffen: „Ein einfacher Schachspieler kann sich vielleicht nicht vorstellen, was zu tun ist, aber ein Großmeister sieht einfach das Brett und weiß, dass ein bestimmter Zug gemacht werden muss ... die richtige Antwort springt ihm einfach in seinen Kopf.“ Es stimmt natürlich, dass ein Großteil des Denkprozesses eines Schenkers oder eines Schachgroßmeisters auf einer Ebene passiert, die nicht für die Introspektion durch den bewussten Verstand offen ist. Das bedeutet aber nicht, dass der Denkprozess nicht existiert. Die wichtige Frage, die Dreyfus nicht beantwortet, ist, wie der richtige Zug in das Gehirn des Großmeisters springt. Dabei denkt man an den Kommentar von Daniel Dennett (1984):

> *Es ist fast so, als würden sich Philosophen als Experten für die Methoden von Zauberkünstlern proklamieren, und wenn wir fragen, wie der Zauberer den Trick mit der zersägten Jungfrau ausführt, erklären sie, dass eigentlich alles ganz*

offensichtlich ist: Der Zauberer sägt sie nicht wirklich in der Mitte auseinander; er geht einfach so vor, dass es so aussieht, als täte er es. „Aber wie macht er das?“, fragen wir. „Nicht unsere Angelegenheit“, sagen die Philosophen.

Dreyfus und Dreyfus (1986) schlagen einen fünf Phasen umfassenden Prozess vor, Erfahrungen zu sammeln, beginnend mit einer regelbasierten Verarbeitung (der Art, wie sie in GOFAI vorgeschlagen wurde) und endend mit der Fähigkeit, die richtigen Antworten unmittelbar auszuwählen. Durch diesen Vorschlag werden Dreyfus und Dreyfus letztlich von den KI-Kritikern zu KI-Theoretikern – sie schlagen eine neuronale Netzarchitektur vor, die in einer riesigen „Fallbibliothek“ angeordnet ist, weisen jedoch auf mehrere Probleme hin. Glücklicherweise wurden alle ihre Probleme untersucht, einige davon teilweise, andere vollständig erfolgreich. Ihre Probleme sind unter anderem:

1 Eine gute Verallgemeinerung aus Beispielen kann nicht ohne Hintergrundwissen erzielt werden. Sie behaupten, dass niemand weiß, wie Hintergrundwissen in den Lernprozess neuronaler Netze eingebracht werden kann. In den *Kapiteln 19* und *20* haben wir jedoch gesehen, dass es Techniken gibt, um Vorwissen in Lernalgorithmen zu verwenden. Diese Techniken basieren jedoch auf der Verfügbarkeit von Wissen in expliziter Form – etwas, was Dreyfus und Dreyfus strikt leugnen. Unserer Ansicht nach ist dies ein guter Grund für einen ernsthaften Neuentwurf aktueller Modelle der neuronalen Verarbeitung, so dass sie das zuvor gelernte Wissen so nutzen *können*, wie andere Lernalgorithmen dies tun.

2 Das Lernen in neuronalen Netzen ist eine Form des überwachten Lernens (siehe *Kapitel 18*), wofür die unbedingte Identifizierung relevanter Eingaben und korrekter Ausgaben erforderlich ist. Aus diesem Grund, behaupten sie, könne das Lernen nicht ohne die Hilfe eines menschlichen Trainers funktionieren. Tatsächlich jedoch kann das Lernen ohne einen Trainer durch ein **nicht überwachtes Lernen** (*Kapitel 20*) sowie durch ein **verstärkendes Lernen** (*Kapitel 21*) bewerkstelligt werden.

3 Lernalgorithmen erbringen mit vielen Merkmalen keine gute Leistung, und wenn wir eine Untermenge von Merkmalen auswählen, „gibt es keine bekannte Methode, neue Merkmale hinzuzufügen, falls sich die aktuelle Menge als ungeeignet für die gelernten Tatsachen erweisen sollte“. Tatsächlich können neue Methoden wie beispielsweise Support-Vektor-Maschinen sehr gut mit großen Merkmalsmengen umgehen. Mit der Einführung großer webbasierter Datenmengen verarbeiten viele Anwendungen in Bereichen wie der Sprachverarbeitung (Sha und Pereira, 2003) und der Computervision (Viola und Jones, 2002a) routinemäßig Millionen von Merkmalen. Wie wir in *Kapitel 19* gesehen haben, gibt es auch prinzipielle Vorgehensweisen, neue Merkmale zu erzeugen, obwohl dafür mehr Arbeit erforderlich ist.

4 Das Gehirn ist in der Lage, seine Sensoren so auszurichten, dass es relevante Informationen sucht und sie verarbeitet, um Aspekte zu extrahieren, die für die aktuelle Situation relevant sind. Dreyfus und Dreyfus behaupten jedoch, „momentan sind keine Details dieses Mechanismus verstanden oder sogar so hypothetisiert, dass sie die KI-Forschung leiten könnten“. Tatsächlich beschäftigt sich der Bereich der aktiven Vision, untermauert durch die Theorie des Informationswertes (*Kapitel 16*), mit genau dem Problem, Sensoren auszurichten, und es wurden bereits einige Roboter implementiert, die theoretisch erzielte Ergebnisse umsetzen konnten. Die Fahrt von STANLEY über 132 Meilen durch die Wüste (siehe *Abschnitt 1.4*) wurde zu einem großen Teil durch ein derartiges aktives Sensorsystem ermöglicht.

Insgesamt wurden viele der Aspekte, auf die sich Dreyfus konzentriert hat – Hintergrundallgemeinwissen, das Qualifikationsproblem, Unsicherheit, Lernen, kompilierte Arten der Entscheidungsfindung – in der Tat wichtige Fragen und sie werden inzwischen beim Standardentwurf intelligenter Agenten berücksichtigt. Unserer Meinung nach ist dies der Beweis für den Fortschritt der KI, nicht für ihre Unmöglichkeit.

Eines der stärksten Argumente von Dreyfus bezieht sich auf situierte Agenten statt auf körperlose logische Inferenzmaschinen. Ein Agent, dessen Verständnis von „Hund" nur aus einer beschränkten Menge von logischen Sätzen wie zum Beispiel „$Hund(x) \Rightarrow Säugetier(x)$" stammt, ist im Nachteil gegenüber einem Agenten, der Hunde beim Laufen beobachten konnte, mit ihnen Apportieren gespielt und sie gestreichelt hat. Der Philosoph Andy Clark (1998) stellte fest: „Biologische Gehirne sind in erster Linie Kontrollsysteme für biologische Körper. Biologische Körper bewegen sich und agieren in mächtigen realen Umgebungen." Um zu verstehen, wie menschliche (und tierische) Agenten funktionieren, müssen wir den ganzen Agenten und nicht nur das Agentenprogramm betrachten. So behauptet das Konzept der **Inkarnation** (engl. **Embodiment**), dass es nicht sinnvoll ist, das Gehirn separat zu betrachten: Wahrnehmung findet in einem Körper statt, der in eine Umgebung eingebettet ist. Wir müssen das System als Ganzes studieren; das Gehirn erweitert sein logisches Denken, indem es auf die Umgebung reagiert, genau wie der Leser Markierungen auf Papier wahrnimmt (und erzeugt), um Wissen zu vermitteln. Unter dem Embodiment-Programm werden Robotik, Vision und andere Sensoren zu zentralen Elementen und nicht zu peripheren.

26.2 Starke KI: Können Maschinen wirklich denken?

Viele Philosophen behaupten, dass eine Maschine, die den Turing-Test besteht, nicht *wirklich* denkt, sondern nur *simuliert* zu denken. Auch dieser Einwand wurde von Turing vorhergesehen. Er zitiert eine Rede von Professor Geoffrey Jefferson (1949):

> *Erst wenn eine Maschine ein Sonnet schreiben oder ein Konzert komponieren kann, weil sie denkt und fühlt und nicht durch die zufällige Anordnung von Symbolen, können wir zustimmen, dass Maschinen es dem Gehirn gleich tun – d.h. sie schreiben nicht nur, sondern sie wissen auch, dass sie geschrieben haben.*

Turing bezeichnet dieses Argument als **Bewusstsein** – die Maschine muss sich ihres eigenen mentalen Zustands und ihrer Aktionen bewusst sein. Während das Bewusstsein ein wichtiges Thema ist, hat der Schlüsselaspekt von Jefferson eigentlich mit der **Phänomenologie** zu tun oder der Untersuchung direkter Erfahrung: Die Maschine muss tatsächlich Gefühle zeigen. Andere konzentrieren sich auf die **Absicht** – d.h. die Frage, ob die vorgegebenen Glauben, Wünsche und anderen Empfindungen tatsächlich „über" etwas in der realen Welt darstellen.

Turings Antwort auf den Einwand ist interessant. Er hätte verschiedene Gründe aufzählen können, dass Maschinen sehr wohl bewusst sein können (oder eine Phänomenologie oder Absichten haben). Stattdessen bleibt er dabei, dass die Frage einfach falsch definiert ist, ähnlich wie die Frage: „Können Maschinen denken?" Außerdem: Warum sollten wir für Maschinen auf einem höheren Standard bestehen als für Menschen? Schließlich haben wir im täglichen Leben auch niemals *irgendeinen* direkten Beweis für den internen mentalen Zustand unserer Mitmenschen. Nichtsdestotrotz sagt Turing: „Anstatt ständig über diesen Aspekt zu streiten, ist es üblich, die höfliche Übereinkunft zu wählen, dass jedermann denkt."

Turing sagt, Jefferson würde die höfliche Übereinkunft erweitern auf Maschinen, wenn er schon Erfahrungen mit Maschinen gemacht hätte, die intelligent handeln. Er zitiert den folgenden Dialog, der eine beliebte Erzählung in der KI geworden ist, sodass wir ihn einfach aufnehmen müssen:

> MENSCH: Wäre in der ersten Zeile deines Gedichts „Soll ich dich mit einem Sommertag vergleichen" nicht „Frühlingstag" genauso gut oder besser?
>
> MASCHINE: Es würde sich nicht skandieren lassen.
>
> MENSCH: Was ist mit „Wintertag"? Das würde sich gewiss skandieren lassen.
>
> MASCHINE: Ja, aber niemand will mit einem Wintertag verglichen werden.
>
> MENSCH: Würden Sie sagen, Mr. Pickwick erinnert Sie an Weihnachten?
>
> MASCHINE: In gewisser Weise.
>
> MENSCH: Nun, Weihnachten ist an einem Wintertag, und ich glaube, Mr. Pickwick würde den Vergleich nicht übel nehmen.
>
> MASCHINE: Ich glaube nicht, dass Sie Recht haben. Mit einem Wintertag meint man einen typischen Wintertag und nicht einen speziellen wie Weihnachten.

Man kann sich leicht vorstellen, dass derartige Mensch-Maschine-Konversationen in Zukunft alltäglich sind und man sich daran gewöhnen wird, keine linguistische Unterscheidung zwischen „realem" und „künstlichem" Denken mehr zu treffen. Eine ähnliche Wandlung vollzog sich in den Jahren nach 1848, als Frederick Wöhler erstmals künstlichen Harnstoff synthetisiert hatte. Vor diesem Ereignis waren organische und anorganische Chemie praktisch getrennte Unternehmungen und viele dachten, dass es kein Verfahren gibt, das anorganische Chemikalien in organisches Material umwandeln kann. Nach der gelungenen Synthese waren sich die Chemiker einig, dass künstlicher Harnstoff *echter* Harnstoff sei, weil er alle wichtigen physischen Eigenschaften aufwies. Diejenigen, die postuliert hatten, dass organische Stoffe eine intrinsische Eigenschaft besäßen, die anorganische Stoffe niemals haben könnten, wurden mit der Unmöglichkeit konfrontiert, irgendeinen Test zu entwickeln, der die vermeintliche Unzulänglichkeit von künstlichem Harnstoff aufdecken könnte.

In Bezug auf das Denken haben wir noch nicht unser 1848 erreicht und viele glauben, dass künstliches Denken niemals real sein wird, egal wie eindrucksvoll es ist. Zum Beispiel argumentiert der Philosoph John Searle (1980) wie folgt:

> *Niemand nimmt an, die Computersimulation eines Sturms mache uns alle nass ... warum in aller Welt sollte jemand wirklich annehmen, eine Computersimulation mentaler Prozesse hätte wirklich mentale Prozesse erzeugt? (S. 37-38)*

Man stimmt zwar leicht damit überein, dass Computersimulationen von Stürmen uns nicht nass machen, es ist aber nicht klar, wie man diese Analogie auf Computersimulationen mentaler Prozesse übertragen sollte. Schließlich werden bei einer Hollywood-Simulation eines Sturmes Sprinkleranlagen und Windmaschinen verwendet, die die Schauspieler sehr wohl nass machen, und eine Videospiel-Simulation eines Sturmes macht auch die simulierten Figuren nass. Die meisten Menschen sagen ohne zu zögern, dass die Computersimulation einer Addition eine Addition ist und dass die Computersimulation eines Schachspieles ein Schachspiel ist. In der Tat sprechen wir von einer *Implementierung* der Addition oder des Schachspieles und nicht von einer *Simulation*. Sind mentale Prozesse nun eher wie Stürme oder eher wie Addition?

Turings Antwort – die höfliche Übereinkunft – schlägt vor, dass das Problem letztlich von selbst verschwindet, sobald Maschinen ein bestimmtes Niveau der Perfektion erreicht haben. Daraufhin würde sich auch der Unterschied zwischen schwacher und starker KI auflösen. Man könnte dagegenhalten, dass eine Sachfrage von großem Interesse ist: Menschen haben wirkliche Gedanken und Maschinen vielleicht auch, vielleicht aber auch nicht. Um diese Sachfrage zu erörtern, müssen wir verstehen, wie es sein kann, dass Menschen einen wirklichen Geist haben und nicht einfach Körper, die neurophysiologische Prozesse erzeugen. Die philosophischen Bestrebungen, dieses **Geist-Körper-Problem** zu lösen, sind direkt mit der Frage verknüpft, ob Maschinen einen wirklichen Geist haben können.

Das Geist-Körper-Problem wurde schon von den altgriechischen Philosophen und von verschiedenen Schulen des Hindu-Denkens betrachtet, in der Tiefe jedoch erstmals im 17. Jahrhundert durch den französischen Philosophen und Mathematiker René Descartes analysiert. Sein Werk *Meditationen über die Erste Philosophie* (1641) betrachtete die Denkaktivität des Geistes (einen Prozess ohne räumliche Ausdehnung oder materielle Eigenschaften) und die physikalischen Prozesse des Körpers mit der Schlussfolgerung, dass beide in getrennten Bereichen existieren müssen – was wir heute als **dualistische** Theorie bezeichnen. Im Rahmen des Geist-Körper-Problems stellen Dualisten die Frage, wie der Geist den Körper kontrollieren kann, wenn beide eigentlich getrennt sind. Descartes vermutete, dass die beiden über die Zirbeldrüse in Wechselwirkung treten, was einfach die Frage aufwirft, wie der Geist die Zirbeldrüse steuert.

Die **monistische** Theorie des Geistes, häufig als **Physikalismus** bezeichnet, vermeidet dieses Problem, indem sie behauptet, dass der Geist nicht vom Körper getrennt ist – dass mentale Zustände physische Zustände sind. Die meisten modernen Geistesphilosophen sind auf die eine oder andere Weise Physikalisten und der Physikalismus erlaubt zumindest prinzipiell die Möglichkeit der starken KI. Das Problem für Physikalisten besteht darin, zu erklären, wie physische Zustände – insbesondere die molekularen Konfigurationen und elektrochemischen Prozesse des Gehirnes – gleichzeitig mentale Zustände sein können, wie zum Beispiel Schmerzen zu fühlen, sich über einen Hamburger zu freuen, zu wissen, dass man auf einem Pferd reitet, oder zu glauben, dass Wien die Hauptstadt von Österreich ist.

26.2.1 Mentale Zustände und das Gehirn im Tank

Physikalistische Philosophen versuchten zu erklären, was es bedeutet zu sagen, dass sich eine Person – und in Erweiterung ein Computer – in einem besonderen mentalen Zustand befindet. Sie haben sich vor allem auf **intentionale Zustände** konzentriert. Dabei handelt es sich um Zustände wie Glauben, Wissen, Wünschen, Fürchten usw., die sich auf einen Aspekt der externen Welt beziehen. Zum Beispiel ist das Wissen, dass jemand einen Hamburger isst, ein Glaube *über* den Hamburger und das, was mit ihm passiert.

Wenn Physikalismus korrekt ist, muss es zutreffen, dass die geeignete Beschreibung des mentalen Zustandes einer Person durch den Gehirnzustand dieser Person bestimmt wird. Bin ich also momentan bewusst damit beschäftigt, einen Hamburger zu essen, ist mein momentaner Gehirnzustand eine Instanz der Klasse mentaler Zustände „wissen,

dass man einen Hamburger isst“. Natürlich sind die konkreten Konfigurationen aller Atome meines Gehirnes nicht wesentlich: Es gibt viele Konfigurationen meines Gehirnes oder des Gehirnes anderer Menschen, die zur selben Klasse mentaler Zustände gehören. Entscheidend ist, dass derselbe Gehirnzustand keinem grundsätzlich anderen mentalen Zustand entsprechen kann, beispielsweise dem Wissen, dass man eine Banane isst.

Die Einfachheit dieser Betrachtungsweise wird durch einige einfache Gedankenexperimente herausgefordert. Stellen Sie sich vor, dass Ihr Gehirn bei der Geburt aus Ihrem Körper entfernt und in einem wunderbar ausgestatteten Tank platziert wurde. Der Tank versorgt Ihr Gehirn und ermöglicht es ihm, zu wachsen und sich zu entwickeln. Gleichzeitig werden Ihrem Gehirn elektronische Signale von einer Computersimulation einer vollkommen fiktiven Welt eingespeist und motorische Signale aus dem Gehirn abgegriffen und verwendet, um die Simulation entsprechend abzuändern.[2] In der Tat repliziert das simulierte Leben, das Sie leben, genau das Leben, dass Sie gelebt hätten, wenn Ihr Gehirn nicht im Tank platziert worden wäre, einschließlich dem simulierten Verspeisen von simulierten Hamburgern. Somit könnten Sie einen Gehirnzustand haben, der identisch mit dem Zustand einer anderen Person ist, die wirklich einen echten Hamburger isst, doch es wäre buchstäblich falsch zu sagen, dass Sie den mentalen Zustand „wissen, dass man einen Hamburger isst“ haben. Denn Sie essen keinen Hamburger, haben niemals mit einem Hamburger zu tun gehabt und könnten folglich keinen derartigen mentalen Zustand haben.

Dieses Beispiel scheint der Ansicht zu widersprechen, dass Gehirnzustände die mentalen Zustände bestimmen. Um dieses Dilemma aufzulösen, könnte man beispielsweise sagen, dass der Inhalt mentaler Zustände von zwei verschiedenen Standpunkten aus interpretiert werden kann. Die Ansicht „**weiter Inhalt**“ interpretiert ihn vom Standpunkt eines allwissenden außenstehenden Beobachters mit Zugriff auf die ganze Situation, der Unterschiede in der Welt ausmachen kann. Unter dieser Ansicht bezieht der Inhalt mentaler Zustände sowohl den Gehirnzustand als auch die Umgebungsgeschichte ein. Dagegen betrachtet „**enger Inhalt**“ nur den Gehirnzustand. Der enge Inhalt der Gehirnzustände eines realen Hamburger-Essers und eines „Gehirn-im-Tank-Hamburger-Essers“ ist in beiden Fällen der gleiche.

Weiter Inhalt ist vollkommen angebracht, wenn es darum geht, mentale Zustände anderen, die zur eigenen Welt gehören, zuzuschreiben, um ihr wahrscheinliches Verhalten und dessen Wirkungen vorherzusagen, usw. Dies ist die Szenerie, in der sich unsere normale Sprache über mentalen Inhalt herausgebildet hat. Wenn sich andererseits jemand mit der Frage beschäftigt, ob KI-Systeme wirklich denken und wirklich über mentale Zustände verfügen, ist enger Inhalt zweckmäßig; es ist einfach nicht sinnvoll zu sagen, dass es von den Bedingungen außerhalb eines KI-Systems abhängt, ob das System wirklich denkt. Enger Inhalt ist auch interessant, wenn wir über den Entwurf von KI-Systemen oder das Verstehen ihrer Funktionsweise nachdenken, weil es der enge Inhalt eines Gehirnzustandes ist, der den (engen Inhalt für den) nächsten Gehirnzustand bestimmt. Dies führt fast von selbst zu der Auffassung, dass für einen Gehirnzustand die funktionelle Rolle in der mentalen Operation der jeweiligen Entität entscheidend ist – was ihn zu der einen Art von mentalem Zustand und nicht zu einer anderen macht.

2 Diese Situation ist vielleicht allen Lesern vertraut, die den Film *Die Matrix* aus dem Jahre 1999 gesehen haben.

26.2.2 Funktionalismus und das Gehirnprothesenexperiment

Die Theorie des **Funktionalismus** besagt, dass ein mentaler Zustand eine kausale Zwischenbedingung zwischen Eingabe und Ausgabe ist. Nach der Theorie der Funktionalisten hätten zwei Systeme mit isomorphen kausalen Prozessen dieselben mentalen Zustände. Aus diesem Grund könnte ein Computerprogramm dieselben mentalen Zustände wie eine Person haben. Natürlich haben wir noch nicht gesagt, was „isomorph“ wirklich bedeutet, aber man nimmt an, dass es eine gewisse Abstraktionsebene gibt, unterhalb der die konkrete Implementierung keine Rolle spielt.

Die Behauptungen der Funktionalisten lassen sich anschaulich durch das Gehirnprothesenexperiment beschreiben. Dieses Gedankenexperiment wurde durch den Philosophen Clark Glymour eingeführt und von John Searle (1980) wieder aufgegriffen, wird jedoch üblicherweise dem Roboteringenieur Hans Moravec (1988) zugeordnet. Es sieht wie folgt aus: Angenommen, die Neurophysiologie hat sich so weiterentwickelt, dass das Eingabe/Ausgabe-Verhalten und die Verbindung aller Neuronen im menschlichen Gehirn perfekt verstanden werden. Nehmen wir weiterhin an, dass wir mikroskopisch kleine elektronische Bauelemente erstellen können, die dieses Verhalten nachbilden und die perfekt in das Nervengewebe eingebunden werden können. Nehmen wir schließlich an, dass dieselbe wunderbare Operationstechnik einzelne Neuronen durch entsprechende elektronische Bauelemente ersetzen kann, ohne die Arbeitsweise des Gehirns als Ganzes zu unterbrechen. Das Experiment besteht darin, schrittweise alle Neuronen im Kopf einer Person durch elektronische Bauelemente zu ersetzen.

Wir beschäftigen uns sowohl mit dem externen Verhalten als auch mit der internen Erfahrung der Testperson während und nach der Operation. Aufgrund der Definition des Experimentes muss das externe Verhalten der Testperson im Vergleich zu dem unverändert bleiben, was beobachtet worden wäre, wäre die Operation nicht ausgeführt worden.[3] Obwohl das Vorhandensein oder das Fehlen des Bewusstseins durch eine dritte Person nicht sicher beurteilt werden kann, sollte die Testperson im Experiment mindestens in der Lage sein, Änderungen ihrer eigenen bewussten Erfahrungen zu erkennen. Offensichtlich gibt es einen direkten Konflikt der Vorstellungen, was passieren könnte. Moravec, ein Robotikforscher und Funktionalist, ist davon überzeugt, dass sein Bewusstsein unbeeinflusst bleiben würde. Searle, ein Philosoph und biologischer Naturalist, ist dagegen überzeugt, dass sein Bewusstsein verschwinden würde:

> *Sie stellen total überrascht fest, dass Sie tatsächlich die Kontrolle über Ihr externes Verhalten verlieren. Sie stellen beispielsweise fest, wenn die Ärzte Ihren Gesichtssinn testen, dass Sie sie sagen hören: „Wir halten ein rotes Objekt vor Sie; sagen Sie uns bitte, was Sie sehen.“ Sie wollen laut schreien: „Ich sehe überhaupt nichts, ich werde blind!“ Aber Sie hören Ihre Stimme sagen, und zwar auf eine Weise, die überhaupt nicht mehr unter Ihrer Kontrolle ist: „Ich sehe eine rotes Objekt vor mir.“ .... Ihre bewusste Erfahrung schrumpft langsam zu einem Nichts, während Ihr externes beobachtbares Verhalten dasselbe bleibt. (Searle, 1992)*

3 Man könnte sich vorstellen, für Vergleichszwecke eine identische „Kontrollperson“ zu verwenden, an der man eine Placebo-Operation vornimmt.

Man kann jedoch mehr tun, als nur der Intuition nach zu argumentieren. Beachten Sie als Erstes, dass, wenn das externe Verhalten gleich bleiben soll, während die Testperson langsam unbewusst wird, der Willen der Testperson sofort und vollständig entfernt wird; andernfalls würde der Verlust des Bewusstseins im externen Verhalten reflektiert – „Hilfe, ich schrumpfe!“ oder Wörter dieser Art. Dieses unmittelbare Verschwinden des Willens als Ergebnis einer schrittweisen Neuronenersetzung scheint eine unwahrscheinliche Behauptung zu sein.

Betrachten Sie zweitens, was passiert, wenn wir der Testperson Fragen zu ihrer bewussten Erfahrung stellen, die sie hatte, während keine echten Neuronen mehr in ihrem Gehirn waren. Aufgrund der Bedingungen für das Experiment erhalten wir Antworten wie beispielsweise: „Es geht mir gut. Ich muss sagen, ich bin ein bisschen überrascht, weil ich dem Argument von Searle geglaubt hatte.“ Wir könnten die Testperson auch mit einer Nadel stechen und die Antwort „Oh, das tut weh!“ beobachten. Im normalen Verlauf der Dinge kann der Skeptiker solche Ausgaben von KI-Programmen als einfache Erfindung abtun. Natürlich könnte man sich einfach vorstellen, eine Regel wie beispielsweise „Wenn Sensor 12 ‚high‘ ist, dann gib ‚Aua!‘ aus!“ zu verwenden. Der Punkt ist jedoch hier, dass wir, auch weil wir die funktionalen Eigenschaften eines normalen menschlichen Gehirns impliziert haben, annehmen, dass das elektronische Gehirn keine solchen Erfindungen enthält. Dann müssen wir eine Erklärung für die Manifestierungen des Bewusstseins haben, das durch das elektronische Gehirn erzeugt wird, die sich nur auf die funktionalen Eigenschaften der Neuronen bezieht. *Und diese Erklärung muss auch für das echte Gehirn gelten, das dieselben funktionalen Eigenschaften aufweist.* Es gibt drei mögliche Schlüsse:

1. Die kausalen Mechanismen des Bewusstseins, die in normalen Gehirnen diese Art von Ausgaben erzeugen, arbeiten auch noch in der elektronischen Form, die deshalb bewusst ist.
2. Die bewussten mentalen Ereignisse im normalen Gehirn haben keine kausale Verbindung zum Verhalten, und sie fehlen im elektronischen Gehirn, das deshalb nicht bewusst ist.
3. Das Experiment ist unmöglich und demzufolge sind Spekulationen darüber bedeutungslos.

Obwohl wir die zweite Möglichkeit nicht ausschließen können, reduziert sie das Bewusstsein darauf, was die Philosophen als **epiphänomenale** Rolle bezeichnen – etwas, was passiert, aber keinen Schatten wirft, wie es in der beobachtbaren Welt wäre. Ist das Bewusstsein darüber hinaus wirklich epiphänomenal, kann es nicht sein, dass die Versuchsperson „Aua!“ ruft, *weil es weh tut* – d.h. aufgrund der bewussten Erfahrung von Schmerz. Stattdessen muss das Gehirn einen zweiten, unbewussten Mechanismus enthalten, der für das „Aua!“ verantwortlich ist.

Patricia Churchland (1986) legt dar, dass die Argumente der Funktionalisten, die auf der Ebene des Neurons ansetzen, auch auf der Ebene einer größeren funktionalen Einheit ansetzen könnten – für einen Neuronenklumpen, ein mentales Modul, einen Lappen, eine Gehirnhälfte oder das gesamte Gehirn. Das bedeutet, wenn Sie das Konzept akzeptieren, dass das Gehirnprothesenexperiment zeigt, dass das Ersatzgehirn bewusst ist, dann sollten Sie auch glauben, dass das Bewusstsein beibehalten wird, wenn das gesamte Gehirn durch einen Schaltkreis ausgetauscht wird, der seinen Zustand aktualisiert und mithilfe einer riesigen Suchtabelle eine Abbildung von Eingaben auf Aus-

gaben vornimmt. Das gefällt vielen Menschen nicht (einschließlich Turing selbst), die die Vorstellung haben, dass Suchtabellen nicht bewusst sind – oder zumindest, dass die bewussten Erfahrungen, die beim Nachschlagen in der Tabelle entstehen, nicht dieselben sind wie die, die während der Arbeit eines Systems erzeugt werden, von dem man (selbst in einem einfachen, rechentechnischen Sinne) annimmt, dass es auf Glauben, Introspektionen, Ziele usw. zugreift und diese erzeugt.

26.2.3 Biologischer Naturalismus und das Chinesische Zimmer

Der Funktionalismus wird besonders durch den **biologischen Naturalismus** von John Searle (1980) infrage gestellt, dem entsprechend mentale Zustände sich herausbildende High-Level-Merkmale sind, die durch physische Low-Level-Prozesse *in den Neuronen* verursacht werden, und dass gerade die (nicht spezifizierten) Eigenschaften der Neuronen eine Rolle spielen. Mentale Zustände können also nicht allein auf der Grundlage eines Programms dupliziert werden, das dieselbe funktionale Struktur mit demselben Eingabe/Ausgabe-Verhalten hat; wir würden fordern, dass das Programm auf einer Architektur mit derselben kausalen Leistung wie Neuronen ausgeführt wird. Um diese Ansicht zu unterstützen, beschreibt Searle ein hypothetisches System, das offensichtlich ein Programm ausführt und den Turing-Test besteht, aber ebenfalls offensichtlich (gemäß Searle) keine seiner Eingaben und Ausgaben *versteht*. Er schließt daraus, dass die Ausführung des geeigneten Programms (d.h. die Erzeugung der richtigen Ausgaben) keine *hinreichende* Bedingung für die Darstellung von Verstand ist.

Das System besteht aus einem Menschen, der nur Englisch versteht und der mit einem Regelbuch ausgestattet ist, das in Englisch geschrieben ist, sowie mehreren Papierstapeln, von denen einige leer sind, andere nicht entzifferbare Aufschriften tragen. (Der Mensch spielt damit die Rolle der CPU, das Regelbuch ist das Programm und die Papierstapel verkörpern den Speicher.) Das System befindet sich innerhalb eines Raumes mit einer kleinen Öffnung zur Außenwelt. Durch diese Öffnung erscheinen Papierschnitzel mit nicht entzifferbaren Symbolen. Der Mensch sucht im Regelbuch nach übereinstimmenden Symbolen und folgt den Anweisungen. Die Anweisungen können sein, Symbole auf neue Papierstreifen zu schreiben, in den Stapeln nach den Symbolen zu suchen, die Stapel anders anzuordnen usw. Irgendwann bewirken die Anweisungen, dass ein oder mehrere Symbole auf ein Stück Papier übertragen und dann an die Außenwelt zurückgegeben werden.

Tipp

So weit, so gut. Doch von außerhalb sehen wir ein System, das Eingaben in Form chinesischer Sätze entgegennimmt und chinesische Antworten erzeugt, die offensichtlich so „intelligent“ sind wie diejenigen in der von Turing geforderten Konversation.[4] Searle argumentiert dann wie folgt: Die Person im Raum versteht kein Chinesisch (das ist Voraussetzung). Das Regelbuch und die Papierstapel, wobei es sich nur um einfache Papierstücke handelt, verstehen kein Chinesisch. Es wird also kein Chinesisch verstanden. *Damit bedeutet nach Searle die Ausführung des richtigen Programms nicht unbedingt, Verstehen zu erzeugen.*

4 Die Tatsache, dass die Papierstapel Billionen Seiten enthalten können und dass die Erzeugung von Antworten Millionen von Jahren dauern kann, spielt in der *logischen* Struktur des Arguments keine Rolle. Ein Ziel der philosophischen Schulung ist, einen feinen Sinn dafür zu entwickeln, welche Einwände passend sind und welche nicht.

Wie Turing berücksichtigte Searle mögliche Reaktionen auf sein Argument und versuchte, sie zurückzuweisen. Mehrere Kommentatoren, einschließlich John McCarthy und Robert Wilensky, schlugen das vor, was Searle als die „Antwort des Systems" bezeichnet. Der Einwand ist, dass man zwar fragen kann, ob der Mensch in dem Zimmer Chinesisch versteht, dies aber analog zu der Frage ist, ob die CPU kubische Wurzeln berechnen kann. In beiden Fällen lautet die Antwort Nein, und in beiden Fällen hat das ganze System gemäß der Antwort des Systems sehr wohl die gesuchte Fähigkeit. Wenn man natürlich das chinesische Zimmer fragt, ob es Chinesisch versteht, wäre die Antwort eine Bestätigung (in fließendem Chinesisch). Nach Turings höflicher Übereinkunft sollte dies ausreichend sein. Die Antwort von Searle ist, den Punkt zu wiederholen, dass sich das Verständnis nicht im Menschen befindet und nicht im Papier enthalten sein kann, sodass es kein Verstehen geben kann. Er scheint sich auf das Argument zu stützen, dass eine Eigenschaft des Ganzen in einem der Teile enthalten sein muss. Dennoch ist Wasser nass, selbst wenn weder H noch O_2 nass ist. Die eigentliche Behauptung von Searle basiert auf den folgenden vier Axiomen (Searle, 1990):

1. Computerprogramme sind formal (syntaktisch).
2. Menschlicher Verstand hat mentalen Inhalt (Semantik).
3. Syntax an sich ist weder bestimmend noch ausreichend für Semantik.
4. Gehirn führt zu Verstand.

Aus den ersten drei Axiomen schließt Searle, dass Programme nicht ausreichend für den Verstand sind. Mit anderen Worten, ein Agent, der ein Programm ausführt, *könnte* ein Verstand sein, aber er ist nicht *unbedingt* ein Verstand, nur weil er das Programm ausführt. Aus dem vierten Axiom schließt er: „Jedes andere System, das in der Lage ist, Verstand zu verursachen, müsste kausale Leistungen aufweisen, die (mindestens) äquivalent mit der von Gehirnen sind." Von hier aus schließt er, dass jedes künstliche Gehirn die kausale Leistung von realen Gehirnen nachbilden und nicht nur ein bestimmtes Programm ausführen muss und dass menschliche Gehirne keine mentalen Phänomene verursachen, nur weil sie ein Programm ausführen.

Die Axiome sind umstritten. Zum Beispiel stützen sich die Axiome 1 und 2 auf eine nicht spezifizierte Unterscheidung zwischen Syntax und Semantik, die der Unterscheidung zwischen engem und weitem Inhalt scheinbar sehr nahe kommt. Wir können einerseits sagen, dass Computer syntaktische Symbole verarbeiten, und andererseits, dass sie elektrischen Strom beeinflussen, was gerade das ist, was Gehirne (gemäß unserem derzeitigen Verständnis) vorwiegend tun. Es scheint also, dass wir genauso gut sagen könnten, dass Gehirne syntaktisch sind.

Wenn wir die Axiome großzügig interpretieren, *folgt* der Schluss – dass Programme für Verstand nicht ausreichend sind. Der Schluss ist jedoch nicht zufriedenstellend – Searle hat einfach nur gezeigt, dass Sie, wenn Sie den Funktionalismus explizit verweigern (was sein Axiom 3 macht), nicht unbedingt schließen können, dass Nichtgehirne Verstand sind. Das ist ausreichend vernünftig – fast schon tautologisch –, sodass das ganze Argument darauf hinausläuft, ob Axiom 3 akzeptiert werden kann. Gemäß Searle ist beim Argument des chinesischen Zimmers entscheidend, Intuitionen für Axiom 3 zu liefern. Die öffentliche Reaktion zeigt, dass das Argument als das wirkt, was Daniel Dennett (1991) eine **Intuition Pump** nennt: Es verstärkt die A-priori-Intuitionen einer Person, sodass biologische Naturalisten von ihren Positionen überzeugter sind und Funktionalisten nur davon überzeugt sind, dass Axiom 3 nicht

unterstützt wird, oder dass das Argument von Searle im Allgemeinen nicht überzeugend ist. Das Argument wiegelt die Kontrahenten auf, hat aber wenig getan, um die Meinung von irgendjemandem zu ändern. Searle bleibt davon unbeeindruckt und hat erst kürzlich begonnen, das chinesische Zimmer als „Widerlegung" der starken KI und nicht einfach als ein „Argument" zu bezeichnen (Snell, 2008).

Selbst diejenigen, die Axiom 3 akzeptieren und somit dem Argument von Searle zustimmen, können nur auf ihre Intuitionen zugreifen, wenn sie entscheiden, welche Entitäten Verstand sind. Das Argument behauptet zu zeigen, dass das chinesische Zimmer kein Verstand ist, *nur weil es das Programm ausführt*, aber das Argument sagt nichts darüber aus, wie zu entscheiden ist, ob das Zimmer (oder ein Computer, eine andere Art von Maschine oder ein Außerirdischer) ein Verstand ist *wegen irgendeines anderen Grundes*. Searle selbst sagt, dass manche Maschinen über Verstand verfügen: Menschen sind biologische Maschinen mit Verstand. Nach Searle können menschliche Gehirne etwas wie ein KI-Programm ausführen oder nicht, doch wenn sie es tun, ist das noch kein Grund, dass sie Verstand besitzen. Für einen Verstand ist mehr erforderlich – laut Searle etwas, was den kausalen Kräften individueller Neuronen äquivalent ist. Was diese Kräfte sind, bleibt unspezifiziert. Allerdings ist anzumerken, dass sich Neuronen herausgebildet haben, um *funktionale* Rollen zu erfüllen – Lebewesen mit Neuronen haben bereits gelernt und Entscheidungen getroffen, lange bevor Bewusstsein aufgetaucht ist. Es wäre eine bemerkenswerte Koinzidenz, wenn gerade derartige Neuronen wegen bestimmter kausaler Kräfte, die für ihre funktionalen Fähigkeiten irrelevant sind, Bewusstsein erzeugten; immerhin sind es die funktionalen Fähigkeiten, die das Überleben des Organismus bestimmen.

Im Fall des chinesischen Zimmers stützt sich Searle auf Intuition und nicht auf Beweise: Sehen Sie sich das Zimmer an; was könnte hier ein Verstand sein? Dasselbe Argument kann man jedoch über das Gehirn sagen: Betrachten Sie einfach diese Menge von Zellen (oder Atomen), die gemäß den Gesetzen der Biochemie (oder der Physik) blind zusammenarbeiten – was soll dort Verstand sein? Warum kann ein Stück Gehirn einen Verstand aufweisen, ein Stück Leber dagegen nicht? Das bleibt das große Geheimnis.

26.2.4 Bewusstsein, Qualia und die Erklärungslücke

Als roter Faden durch all die Debatten über starke KI zieht sich das Problem des **Bewusstseins** – sozusagen der Elefant im Debattierraum. Bewusstsein wird oftmals in Aspekte wie Verstehen und Selbsterkenntnis unterteilt. Hier konzentrieren wir uns auf den Aspekt der *subjektiven Erfahrung*: Warum **fühlt** es sich wie etwas an, einen bestimmten Gehirnzustand zu haben (z.B. beim Verspeisen eines Hamburgers), während es sich wahrscheinlich nicht wie etwas anfühlt, andere physische Zustände zu haben (z.B. ein Stein zu sein). Der Fachbegriff für die intrinsische Natur der Erfahrungen ist **Qualia** (vom lateinischen Wort, das etwa „solche Dinge" bedeutet).

Die Qualia stellen für funktionalistische Interpretationen des Geistes eine Herausforderung dar, weil verschiedene Qualia in sonst isomorphen kausalen Prozessen beteiligt sein könnten. Nehmen Sie zum Beispiel das Gedankenexperiment des **umgekehrten Spektrums**, dass die subjektive Erfahrung der Person *X* beim Sehen von roten Objekten die gleiche Erfahrung ist, die der Rest von uns hat, wenn wir grüne Objekte sehen, und umgekehrt. *X* bezeichnet rote Objekte trotzdem als „rot", hält an roten Ampeln an und stimmt damit überein, dass die Röte der roten Verkehrsampel ein intensiveres Rot als die Röte des Sonnenunterganges zeigt. Dennoch ist die subjektive Erfahrung von *X* einfach anders.

Qualia sind nicht nur für den Funktionalismus, sondern für die gesamte Wissenschaft eine Herausforderung. Nehmen Sie um des Argumentes willen einmal an, dass wir die wissenschaftliche Forschung zum Gehirn abgeschlossen haben – wir haben herausgefunden, dass der neuronale Prozess P_{12} in Neuron N_{177} Molekül A in Molekül B transformiert usw. usf. Es gibt einfach keine derzeit akzeptierte Form des Schließens, die von solchen Erkenntnissen zum Schluss führen, dass die Entität, die diese Neuronen besitzt, irgendeine besondere subjektive Erfahrung hat. Durch diese **Erklärungslücke** kommen manche Philosophen zu dem Schluss, dass Menschen einfach unfähig sind, ein geeignetes Verstehen ihres eigenen Bewusstseins zu bilden. Andere, insbesondere Daniel Dennett (1991), vermeiden die Lücke, indem sie die Existenz der Qualia ablehnen und sie einer philosophischen Verlegenheit zuschreiben.

Turing selbst gesteht ein, dass die Frage nach dem Bewusstsein schwierig ist, bestreitet aber, dass sie für die Praxis der KI bedeutend ist: „Ich möchte nicht den Eindruck erwecken, ich sei der Meinung, es gäbe keine Geheimnisse um das Bewusstsein. Doch ich glaube nicht, dass diese Geheimnisse unbedingt enthüllt werden müssen, bevor wir die Frage beantworten können, womit wir uns hier beschäftigen." Wir stimmen mit Turing überein – wir sind daran interessiert, Programme zu schaffen, die sich intelligent verhalten. Für das zusätzliche Vorhaben, sie bewusst zu machen, sind wir weder ausgerüstet noch in der Lage, seinen Erfolg zu ermitteln.

26.3 Ethik und Risiken bei der Entwicklung künstlicher Intelligenz

Bisher haben wir uns darauf konzentriert, ob wir künstliche Intelligenz entwickeln *können*, wir müssen jedoch auch betrachten, ob wir das *sollen*. Wenn die Wirkungen der KI-Technologie eher negativ als positiv sind, liegt es in der moralischen Verantwortung der auf diesem Gebiet Tätigen, ihren Forschungen eine andere Richtung zu geben. Viele neue Technologien hatten unbeabsichtigte negative Nebeneffekte: Die Kernfusion brachte Tschernobyl und die Bedrohung einer weltweiten Zerstörung mit sich, Verbrennungsmotoren sind für Luftverschmutzung, globale Erwärmung und die Asphaltierung des Paradieses verantwortlich. In gewissem Sinne sind Autos Roboter, die die Welt eroberten, indem sie sich selbst unentbehrlich gemacht haben.

Alle Wissenschaftler und Ingenieure stehen ethischen Betrachtungen gegenüber, wie sie ihren Job erledigen sollen, welche Projekte sie ausführen oder nicht ausführen sollen und wie diese behandelt werden sollen. Es gibt sogar ein Handbuch zur *Ethics of Computing* (Berleur und Brunnstein, 2001). Allerdings scheint die künstliche Intelligenz einige neue Probleme aufzuwerfen, die über das hinausgehen, wie etwa Brücken zu bauen, die nicht einstürzen:

- Menschen könnten aufgrund der Automatisierung ihre Arbeit verlieren.
- Menschen könnten zu viel (oder zu wenig) Freizeit haben.
- Menschen könnten das Selbstverständnis verlieren, einzigartig zu sein.
- KI-Systeme könnten für unerwünschte Zwecke verwendet werden.
- Die Verwendung von KI-Systemen könnte zu einem Verlust von Verantwortung führen.
- Der Erfolg der künstlichen Intelligenz könnte das Ende der menschlichen Rasse bedeuten.

Wir werden diese Aspekte nacheinander genauer betrachten.

Menschen könnten aufgrund der zunehmenden Automatisierung ihre Arbeit verlieren. Die moderne Industriewirtschaft ist ganz allgemein von den Computern abhängig geworden und wählt insbesondere Programme aus der künstlichen Intelligenz aus. Ein Großteil der Wirtschaft insbesondere in den Vereinigten Staaten ist beispielsweise von der Verfügbarkeit von Verbraucherkrediten abhängig. Kreditkartenanwendungen, Überweisungen und die Erkennung von betrügerischen Informationen werden jetzt durch Programme aus der künstlichen Intelligenz erledigt. Man könnte sagen, Tausende von Menschen wurden aufgrund dieser Programme arbeitslos, aber letztlich hat die künstliche Intelligenz diese Arbeitsplätze nicht weggenommen, sondern es gäbe sie nicht, weil die menschliche Arbeitskraft für diese Transaktionen unbezahlbar wäre. Bisher hat die Automatisierung über die Informationstechnologie im Allgemeinen und die KI im Besonderen mehr Arbeitsplätze geschaffen als zerstört, und sie hat interessante, besser bezahlte Arbeitsplätze geschaffen. Nachdem das kanonische KI-Programm ein „intelligenter Agent" ist, der darauf ausgelegt ist, einen Menschen zu unterstützen, ist der Verlust von Arbeitsplätzen weniger ein Problem, als es der Fall war, als sich die künstliche Intelligenz auf „Expertensysteme" konzentrierte, um den Menschen zu ersetzen. Doch manche Forscher glauben, dass die vollständige Ausführung der Arbeiten das richtige Ziel für die KI ist. In Erinnerung an den 25. Jahrestag der AAAI stellt Nils Nilsson (2005) die Schaffung einer KI auf menschlichem Niveau als Herausforderung dar, die den Eignungstest und nicht den Turing-Test besteht – ein Roboter, der jeden Job aus einem bestimmten Bereich erlernen und ausführen kann. Vielleicht ist die Arbeitslosigkeit in Zukunft hoch, doch selbst die Arbeitslosen dienen als Manager ihrer eigenen Kader von Arbeitsrobotern.

Menschen könnten zu viel (oder zu wenig) Freizeit haben. Alvin Toffler schrieb in *Future Shock* (1970): „Die Arbeitswoche wurde seit der Jahrhundertwende um 50% reduziert. Es ist nicht von der Hand zu weisen, dass sie im Jahr 2000 erneut halbiert wird." Arthur C. Clarke (1968b) schrieb, dass die Menschen im Jahr 2001 „einer Zukunft mit vollkommener Langeweile gegenüberstehen, wobei das wichtigste Problem im Leben ist, zu entscheiden, welchen der mehr als 100 Fernsehkanäle man auswählt". Die einzige dieser Vorhersagen, die dem tatsächlichen Zustand nahe gekommen ist, ist die Anzahl der Fernsehkanäle. Stattdessen arbeiten die Menschen in wissensintensiven Industriezweigen, die Teil eines integrierten, computergestützten Systems sind, das 24 Stunden am Tag arbeitet; um die Arbeit erledigen zu können, sind sie gezwungen, *Überstunden* zu machen. In einer Industriewirtschaft sind die Gewinne etwa proportional zur investierten Zeit; wenn man um 10% mehr arbeitet, heißt es, man hat etwa 10% mehr Einkommen. In einer Informationswirtschaft, die durch Breitbandkommunikation und problemlose Replikation geistigen Eigentums gekennzeichnet ist (was Frank und Cook (1996) die „Winner-Take-All"-Gesellschaft nennen), gibt es eine große Belohnung dafür, etwas besser als die Konkurrenz zu sein; 10% Mehrarbeit könnten 100% mehr Einkommen bedeuten. Es existiert also ein wachsender Druck für jeden, mehr zu arbeiten. Die künstliche Intelligenz steigert die Schrittgeschwindigkeit der technologischen Neuerungen und trägt damit zu diesem allgemeinen Trend bei, aber sie hält auch das Versprechen, uns zu erlauben, freizunehmen und das Ganze eine Zeit lang unseren automatisierten Agenten zu überlassen. Tim Ferriss (2007) empfiehlt Automatisierung und Outsourcing, um eine 4-Stunden-Arbeitswoche zu erreichen.

Menschen könnten das Selbstverständnis verlieren, einzigartig zu sein. In *Computer Power and Human Reason* legt Weizenbaum (1976), der Autor des Programms ELIZA, einige der möglichen Gefahren dar, die die künstliche Intelligenz für die Gesellschaft mit sich bringt. Eines der wichtigsten Argumente von Weizenbaum ist, dass die Forschung im Bereich der künstlichen Intelligenz den Gedanken näher bringt, dass Menschen Automaten sind – ein Konzept, das zu einem Verlust der Autonomie oder sogar der Menschlichkeit führen kann. Dieses Konzept gibt es aber schon sehr viel länger als die künstliche Intelligenz, nämlich mindestens seit *L'Homme Machine* (La Mettrie, 1748). Außerdem hat die Menschheit andere Rückschläge im Hinblick auf unser Gefühl für Einzigartigkeit überlebt: *De Revolutionibus Orbium Coelestium* (Kopernikus, 1543) verbannte die Erde aus dem Mittelpunkt des Sonnensystems und *Descent of Man* (Darwin, 1871) stellte den Homo sapiens auf eine Stufe mit allen anderen Spezies. Wenn die künstliche Intelligenz allgemein erfolgreich ist, kann sie für die moralischen Voraussetzungen der Gesellschaft des 21. Jahrhunderts mindestens so bedrohlich sein wie die Darwinsche Evolutionstheorie für die Menschen im 19. Jahrhundert.

KI-Systeme könnten für unerwünschte Zwecke verwendet werden. Machthaber setzen moderne Technologien oftmals ein, um ihre Rivalen zu unterdrücken. Der Zahlentheoretiker G. H. Hardy schrieb (Hardy, 1040): „Eine Wissenschaft gilt als nützlich, wenn ihre Entwicklung die vorhandenen Ungleichheiten in der Vermögensverteilung hervorhebt oder direkter die Zerstörung des menschlichen Lebens fördert." Dies gilt für alle Wissenschaften, die KI bildet da keine Ausnahme. Autonome KI-Systeme sind heute auf dem Schlachtfeld allgegenwärtig; das US-Militär setzte im Irak über 5.000 autonome Luftfahrzeuge und 12.000 autonome Landfahrzeuge ein (Singer, 2009). Eine Moraltheorie besagt, dass militärische Roboter wie mittelalterliche Rüstungen sind, die bis zur letzten logischen Konsequenz verfolgt wurden: Niemand würde moralische Einwendungen gegen einen Soldaten haben, der einen Helm tragen möchte, wenn er von großen, wütenden, Äxte schwingenden Feinden angegriffen würde, und ein ferngesteuerter Roboter ist wie eine sehr sichere Form der Rüstung. Andererseits bringen Roboterwaffen zusätzliche Risiken mit sich. In dem Maß, wie die menschliche Entscheidungsfindung aus der „Schusslinie" herausgenommen wird, treffen Roboter letzten Endes vielleicht Entscheidungen, die zum Töten unschuldiger Zivilpersonen führen. Im größeren Rahmen kann der Besitz von leistungsfähigen Robotern (wie der Besitz von robusten Helmen) einer Nation Vermessenheit verleihen, was sie dazu bringt, waghalsiger als notwendig in den Krieg zu ziehen. In den meisten Kriegen setzt zumindest eine Partei zu viel Vertrauen in ihre militärischen Fähigkeiten – andernfalls würde man den Konflikt friedlich lösen.

Weizenbaum (1976) hob auch hervor, dass die Technologie der Spracherkennung zu einem allgemeinen Abhören von Telefongesprächen führen könnte und damit zu einem Verlust der Privatsphäre. Er hatte keine Welt mit terroristischen Bedrohungen vorhergesehen, die das Verhältnis ändern könnten, wie viel Überwachung Menschen akzeptieren würden, aber er hat korrekt erkannt, dass die künstliche Intelligenz das Potenzial besitzt, die Überwachung zu einem Massenprodukt zu machen. Seine Vorhersage hat sich zum Teil bereits bewahrheitet: Großbritannien verfügt heute über ein umfangreiches Netz von Überwachungskameras und auch andere Länder überwachen routinemäßig den Webverkehr und Telefongespräche. Einige Menschen akzeptieren, dass die Computerisierung zu einem Verlust der Privatsphäre führt. Der CEO von Sun Microsystems Scott McNealy hat gesagt: „Sie haben sowieso keine Privatsphäre. Vergessen Sie es einfach." David Brin (1998) weist darauf hin, dass der Verlust von Privatsphäre unver-

meidlich ist und sich das Ungleichgewicht der Kräfte zwischen Staat und Einzelpersonen bekämpfen ließe, wenn man die Überwachung allen Bürgern zugänglich macht. Etzioni (2004) spricht sich für eine Ausgewogenheit von Privatsphäre und Sicherheit sowie Rechten des Einzelnen und der Gemeinschaft aus.

Die Verwendung von Systemen der künstlichen Intelligenz könnte zu einem Verlust der Verantwortung führen. Angesichts der prozesssüchtigen Atmosphäre, die in den Vereinigten Staaten herrscht, werden rechtliche Haftungen zu einem wichtigen Thema. Wenn ein Arzt sich auf die Beurteilung einer Diagnose durch ein medizinisches Expertensystem verlässt, ist er dann schuld, wenn die Diagnose falsch ist? Glücklicherweise ist es aufgrund des wachsenden Einflusses der entscheidungstheoretischen Methoden in der Medizin heute akzeptiert, dass Fahrlässigkeit nicht nachgewiesen werden kann, wenn der Arzt medizinische Prozeduren anwendet, die einen hohen *erwarteten* Nutzen aufweisen, selbst wenn das *tatsächliche* Ergebnis für den Patienten katastrophal ist. Die Frage sollte also lauten: „Wessen Fehler ist es, wenn die Diagnose unvernünftig ist?“ Bisher haben die Gerichte entschieden, dass die medizinischen Expertensysteme dieselbe Rolle wie Medizinlehrbücher und Nachschlagewerke spielen; Ärzte sind verantwortlich für das Verständnis der Begründungen hinter jeder Entscheidung und müssen ihre eigene Urteilskraft anwenden, um zu entscheiden, ob sie den Empfehlungen des Systems folgen. Beim Entwurf medizinischer Expertensysteme als Agenten sollten die Aktionen also nicht als direkte Beeinflussung des Patienten, sondern als Beeinflussung des Verhaltens des Arztes betrachtet werden. Wenn Expertensysteme zuverlässig genauer als menschliche Diagnostiker sind, könnten Ärzte rechtlich verantwortlich gemacht werden, wenn sie die Empfehlungen eines Expertensystems *nicht* befolgen. Atul Gawande (2002) untersucht diese Prämisse.

Ähnliche Aspekte entstehen nach und nach im Hinblick auf die Verwendung intelligenter Agenten im Internet. Es wurden gewisse Erfolge bei der Einführung von Beschränkungen für intelligente Agenten erzielt, sodass sie beispielsweise nicht die Dateien anderer Benutzer beschädigen können (Weld und Etzioni, 1994). Das Problem verstärkt sich bei Geldgeschäften. Wenn Geldgeschäfte von einem intelligenten Agenten veranlasst werden, wer ist dann für entstehende Schulden verantwortlich? Wäre es möglich, dass ein intelligenter Agent selbst Vermögen hat und auf eigene Rechnung elektronischen Handel betreibt? Bisher scheinen diese Fragen noch nicht ausreichend untersucht worden zu sein. Unserer Kenntnis nach hat kein Programm eine rechtliche Form als Einzelperson für finanzielle Transaktionen; momentan scheint dies auch unvernünftig zu sein. Programme werden auch nicht als „Fahrer“ für die Einhaltung von Verkehrsregeln auf echten Straßen betrachtet. Nach kalifornischem Gesetz scheint es keinerlei rechtliche Sanktionen zu geben, die verhindern, dass ein automatisiertes Fahrzeug die Geschwindigkeitsbegrenzungen übertritt, obwohl der Entwickler des Steuerungsmechanismus für das Auto im Falle eines Unfalls zur Verantwortung gezogen würde. Wie bei der Clone-Technologie muss das Gesetz den neuen Entwicklungen erst folgen.

Der Erfolg der künstlichen Intelligenz könnte das Ende der menschlichen Rasse bedeuten. Fast jede Technologie hat das Potenzial, in den falschen Händen Schaden anzurichten, aber für die künstliche Intelligenz und die Robotik haben wir das neue Problem, dass die falschen Hände der Technologie selbst gehören können. Unzählige Science-Fiction-Geschichten warnen vor Robotern oder Roboter/Mensch-Cyborgs, die Amok laufen. Frühe Beispiele sind etwa ein *Frankenstein or the Modern Prometheus*

(1818)[5] und Karel Capeks Stück *R.U.R* (1921), in dem Roboter die Welt angreifen. Im Kino haben wir Filme wie beispielsweise *Terminator* (1984), der die Klischees von Roboter-greift-die-Welt-an mit einer Zeitreise kombiniert, oder *Matrix* (1999), der Roboter-greift-die-Welt-an mit Hirn-im-Tank kombiniert.

Es scheint, dass Roboter die Protagonisten vieler Greife-die-Welt-an-Geschichten sind, weil sie das Unbekannte darstellen, so wie Hexen und Geister in den Sagen früherer Epochen oder die Marsianer aus *Der Krieg der Welten* (Wells, 1898). Es stellt sich die Frage, ob ein KI-System ein größeres Risiko als herkömmliche Software darstellt. Dazu sehen wir uns drei Gefahrenquellen an.

Erstens kann die Zustandsschätzung des KI-Systems falsch sein, sodass es falsche Handlungen ausführt. Zum Beispiel könnte ein autonomes Fahrzeug die Position eines Autos in der Nebenspur falsch einschätzen, was zu einem Unfall führt, durch den die Insassen umkommen. Noch ernster ist die Lage, wenn ein Raketenabwehrsystem irrtümlich einen Angriff erkennt und einen Gegenangriff startet, was zum Tod von Abermillionen Menschen führt. Diese Gefahren sind nicht wirkliche Risiken von KI-Systemen – in beiden Fällen könnte der gleiche Fehler von einem Menschen genauso leicht wie von einem Computer gemacht werden. Um diese Gefahren einzugrenzen, ist ein System mit Prüfeinrichtungen und Abwägungsmechanismen zu entwerfen, damit sich ein simpler Fehler in der Zustandsabschätzung nicht ungeprüft durch das System fortpflanzen kann.

Zweitens ist es nicht so leicht, die richtige Nutzenfunktion für ein KI-System zu spezifizieren, um maximalen Nutzen zu erreichen. Zum Beispiel können wir eine Nutzenfunktion vorschlagen, die darauf ausgelegt ist, *menschliches Leid zu minimieren*, und sie als additive Belohnungsfunktion über die Zeit wie in *Kapitel 17* ausdrücken. Allerdings liegt es in der Natur des Menschen, dass wir immer einen Weg finden, um selbst im Paradies zu leiden; die optimale Entscheidung für das KI-System besteht also darin, die menschliche Rasse so bald wie möglich abzuschaffen – keine Menschen, kein Leiden. Bei KI-Systemen müssen wir dann sehr umsichtig vorgehen, wonach wir fragen, während Menschen problemlos erkennen, dass man die vorgeschlagene Nutzenfunktion nicht wörtlich nehmen darf. Andererseits brauchen Computer nicht mit dem Makel irrationaler Verhaltensweisen behaftet zu sein, wie in *Kapitel 16* beschrieben. Menschen verwenden manchmal ihre Intelligenz auf aggressive Weise, weil Menschen aufgrund der natürlichen Selektion angeborene aggressive Tendenzen aufweisen. Die Maschinen, die wir bauen, müssen jedoch nicht von Geburt an aggressiv sein, sofern wir sie nicht auf diese Weise bauen wollen (oder sie als Endprodukt eines Entwurfes entstehen, der aggressives Verhalten fördert). Zum Glück gibt es Techniken wie zum Beispiel das Apprenticeship Learning (d.h. Lernen in einer Ausbildungssituation), durch die wir eine Nutzenfunktion anhand von Beispielen spezifizieren können. Bleibt zu hoffen, dass ein Roboter, der intelligent genug ist, um herauszufinden, wie die menschliche Rasse ausgelöscht werden kann, auch intelligent genug ist, um herauszufinden, dass dies nicht die beabsichtigte Nutzenfunktion war.

Drittens kann die Lernfunktion des KI-Systems bewirken, dass sich das System in eines mit nicht beabsichtigtem Verhalten entwickelt. Dieses Szenario ist am kritischsten und es ist einzigartig für KI-Systeme, sodass wir hier ausführlicher darauf eingehen. I. J. Good schrieb (1965):

5 Als junger Mann wurde Charles Babbage durch die Lektüre von *Frankenstein* inspiriert.

> *Sei eine* ***ultraintelligente Maschine*** *definiert als Maschine, die bei Weitem alle intellektuellen Aktivitäten jedes Menschen übertreffen kann, egal wie gescheit dieser sein mag. Weil der Entwurf von Maschinen eine dieser intellektuellen Aktivitäten ist, könnte eine ultraintelligente Maschine noch bessere Maschinen entwerfen; es gäbe dann zweifellos eine „Intelligenzexplosion", und die Intelligenz der Menschheit würde weit dahinter zurückbleiben. Die erste ultraintelligente Maschine ist also die letzte Erfindung, die der Mensch je machen muss, vorausgesetzt, diese Maschine ist freundlich genug, uns mitzuteilen, wie wir sie unter Kontrolle halten.*

Die „Intelligenzexplosion" wurde von Mathematikprofessor und Science-Fiction-Autor Vernor Vinge auch als **technologische Singularität** bezeichnet. Er schreibt (1993): „Innerhalb von 30 Jahren werden wir die technologischen Mittel besitzen, übermenschliche Intelligenz zu erzeugen. Kurz danach wird die Ära des Menschen beendet sein." Good und Vinge (und viele andere) bemerken sehr richtig, dass die Kurve des Technologiefortschrittes gegenwärtig exponentiell wächst (betrachten Sie dazu Moores Gesetz). Es ist jedoch ein relativ großer Schritt zur Extrapolation, dass die Kurve bis zu einer Singularität von nahezu unendlichem Wachstum fortgesetzt wird. Bisher ist jede andere Technologie einer S-förmigen Kurve gefolgt, wobei das exponentielle Wachstum irgendwann abnimmt. Manchmal greifen neue Technologien ein, wenn die alten eine Plateauphase erreicht haben, gelegentlich stoßen wir an harte Grenzen. Bei einer erst ein Jahrhundert alten Geschichte der Hochtechnologie ist es schwierig, auf Hunderte von Jahren im Voraus zu extrapolieren.

Das Konzept der ultraintelligenten Maschinen geht davon aus, dass Intelligenz ein besonders wichtiges Attribut ist, und wenn man genügend davon besitzt, lassen sich alle Probleme lösen. Doch wir wissen, dass es hinsichtlich der Berechenbarkeit und der rechentechnischen Komplexität Grenzen gibt. Wenn das Problem der Definition ultraintelligenter Maschinen (oder selbst von Annäherungen an sie) in die Komplexitätsklasse von beispielsweise NEXPTIME-vollständigen Problemen fällt und es keine heuristischen Kurzverfahren gibt, dann hilft selbst der exponentielle Fortschritt in der Technologie nicht weiter – die Lichtgeschwindigkeit setzt eine strenge Obergrenze für den Umfang ausführbarer Berechnungen; Probleme über dieser Schranke werden nicht gelöst. Wir wissen immer noch nicht, wo diese Obergrenzen liegen.

Vinge macht sich Sorgen über die kommende Singularität, doch manche Informatiker und Futuristen finden Gefallen daran. Hans Moravec (2000) bestärkt uns darin, jeden Vorteil an unsere „Geisteskinder" weiterzugeben, d.h. an die von uns geschaffenen Roboter, die uns in der Intelligenz überflügeln können. Es gibt sogar ein neues Wort – **Transhumanismus** – für die aktive soziale Bewegung, die sich auf diese Zukunft freut, in der Menschen mit Robotern und biotechnischen Erfindungen zusammenarbeiten – oder durch sie ersetzt werden. Es sei gesagt, dass solche Aspekte eine Herausforderung für die meisten Moraltheoretiker darstellen, die für die Bewahrung des menschlichen Lebens und der menschlichen Spezies eintreten. Ray Kurzweil ist derzeit der bekannteste Befürworter für die Singularitätsansicht. In *The Singularity is Near* (2005) schreibt er:

> *Die Singularität erlaubt es uns, diese Beschränkungen unserer biologischen Körper und des Gehirns zu überschreiten. Wir gewinnen Kraft über unsere Schicksale. Unsere Sterblichkeit wird in unseren Händen liegen. Wir werden in der Lage sein, so lange zu leben, wie wir wollen (eine etwas andere Feststellung*

als die Aussage, dass wir ewig leben werden). Wir werden das menschliche Denken vollständig verstehen und seine Reichweite erheblich ausdehnen. Am Ende des Jahrhunderts wird der nichtbiologische Teil unserer Intelligenz Aberbillionen Mal leistungsfähiger sein als die reine menschliche Intelligenz.

Kurzweil weist auch auf die möglichen Gefahren hin und schreibt: „Die Singularität wird aber auch die Fähigkeit verstärken, auf unsere zerstörerischen Neigungen einzuwirken, sodass die ganze Geschichte noch nicht geschrieben ist." Wenn ultraintelligente Maschinen möglich sind, würden wir Menschen gut daran tun, zu gewährleisten, dass wir ihre Vorgänger in einer solchen Weise konzipieren, dass sie sich selbst so entwerfen, um uns gut zu behandeln. Der Science-Fiction-Autor Isaac Asimov (1942) war der erste, der sich diesem Thema gewidmet und die drei Gesetze der Robotik aufgestellt hat:

1. Ein Roboter darf kein menschliches Wesen verletzen oder durch Untätigkeit erlauben, dass ein menschliches Wesen zu Schaden kommt.
2. Ein Roboter muss den von einem Menschen gegebenen Befehlen gehorchen, außer wenn derartige Befehle mit dem ersten Gesetz kollidieren würden.
3. Ein Roboter muss seine eigene Existenz schützen, solange ein derartiger Schutz nicht mit dem ersten oder zweiten Gesetz kollidiert.

Diese Gesetze scheinen zumindest für uns Menschen vernünftig zu sein. Doch die Kunst besteht darin, diese Gesetze zu implementieren.[6] In der Asimov-Geschichte *Roundabout* wird ein Roboter losgeschickt, um etwas Selen zu holen. Später wird der Roboter aufgefunden, wie er im Kreis um das Selenvorkommen wandert. Immer, wenn er in Richtung des Vorkommens geht, spürt er eine Gefahr und das dritte Gesetz veranlasst ihn, sich umzuwenden. Aber jedes Mal, wenn er sich abwendet, verringert sich die Gefahr und die Kraft des zweiten Gesetzes wird dominant, was ihn veranlasst, wieder zurück zum Selenvorkommen zu gehen. Die Menge der Punkte, die das Gleichgewicht zwischen den beiden Gesetzen definieren, beschreibt einen Kreis. Das legt nahe, dass die Gesetze keine logischen Absolutismen sind, sondern vielmehr gegeneinander abgewogen werden, wobei für die ersteren Gesetze eine höhere Gewichtung gilt. Asimov dachte möglicherweise an eine Architektur, die auf der Kontrolltheorie basiert – vielleicht einer Linearkombination von Faktoren –, während heute die wahrscheinlichste Architektur ein probabilistisch schließender Agent wäre, der über Wahrscheinlichkeitsverteilungen von Ausgaben entscheidet und den Nutzen entsprechend der Definition der drei Gesetze maximiert. Doch vermutlich sollen unsere Roboter nicht verhindern, dass ein Mensch die Straße überquert, nur weil die Gefahr für einen Schaden ungleich null ist. Das heißt, dass der negative Nutzen für eine Schädigung des Menschen wesentlich größer als für Ungehorsam sein muss, aber jeder der Nutzen endlich und nicht unendlich ist.

Yudkowsky (2008) geht mehr ins Detail, wie eine **freundliche KI** zu entwerfen ist. Er macht geltend, dass Freundlichkeit (ein Wunsch, Menschen nicht zu schädigen) von vornherein zu berücksichtigen ist, doch dass der Entwickler erkennen muss, dass sowohl seine eigenen Entwürfe mangelhaft sein können als auch der Roboter mit der

6 Ein Roboter mag die Ungerechtigkeit bemerken, dass es einem Menschen erlaubt wird, einen anderen in Notwehr zu töten, doch von einem Roboter wird verlangt, sein eigenes Leben zu opfern, um einen Menschen zu retten.

Zeit lernt und sich weiterentwickelt. Somit besteht die Herausforderung im mechanischen Entwurf – einen Mechanismus für die Herausbildung von KI-Systemen unter einem System von Prüfungen und Abwägungen zu konzipieren und den Systemen Nutzenfunktionen zu geben, die angesichts derartiger Änderungen freundlich bleiben.

Einem Programm können wir nicht einfach eine statische Nutzenfunktion zuordnen, da sich die Umstände und unsere gewünschten Antworten auf die Umstände mit der Zeit verändern. Wäre es zum Beispiel mit der Technologie um 1800 möglich gewesen, einen superleistungsfähigen KI-Agenten zu entwerfen und ihn mit den vorherrschenden Moralvorstellungen jener Zeit auszustatten, würde er heute dafür kämpfen, die Sklaverei wieder einzuführen und das Wahlrecht für Frauen abzuschaffen. Wenn wir andererseits heute einen KI-Agenten bauen und ihn anweisen, seine Nutzenfunktion herauszubilden, wie lässt sich dann gewährleisten, dass er nicht zu folgendem Schluss kommt: „Menschen finden es moralisch, lästige Insekten zu töten, zum Teil weil Insektengehirne so primitiv sind. Da aber Menschengehirne im Vergleich zu meinen Leistungen primitiv sind, muss es für mich moralisch sein, Menschen zu töten."

Omohundro (2008) stellt die Hypothese auf, dass selbst ein harmloses Schachprogramm eine Gefährdung für die Gesellschaft darstellen könnte. Analog hat Marvin Minsky einmal geäußert, dass ein KI-Programm für die Lösung der Riemannschen Vermutung am Ende sämtliche Ressourcen der Erde aufbrauchen könnte, um leistungsfähigere Supercomputer zu bauen, die ihm helfen, sein Ziel zu erreichen. Sind Sie also lediglich daran interessiert, dass Ihr Programm Schach spielt oder Theoreme beweist, müssen Sie Sicherheitsvorkehrungen treffen, falls Sie das Programm mit der Fähigkeit ausstatten, zu lernen und sich selbst zu modifizieren. Omohundro schlussfolgert, dass „soziale Strukturen, die Individuen die Kosten für ihre negativen Externalitäten aufbürden, einen großen Schritt bedeuten würden, um eine stabile und positive Zukunft zu sichern." Dies scheint eine ausgezeichnete Idee für die Gesellschaft im Allgemeinen zu sein, unabhängig von der Möglichkeit für ultraintelligente Maschinen.

Es sei gesagt, dass die Idee für Schutzmaßnahmen gegen Änderungen in der Nutzenfunktion nicht neu ist. In der *Odyssee* beschreibt Homer (ca. 700 v. Chr.) die Begegnung von Odysseus mit den Sirenen, deren betörender Gesang die Seefahrer dazu zwang, sich selbst in das Meer zu stürzen. Im Wissen, dass ihm das Gleiche widerfahren würde, ließ sich Odysseus von seiner Mannschaft am Mastbaum fesseln, sodass er den selbstzerstörerischen Akt nicht ausführen konnte. Interessant ist es, darüber nachzudenken, wie sich ähnliche Schutzmaßnahmen in KI-Systeme einbauen lassen.

Schließlich wollen wir noch die Perspektive des Roboters betrachten. Wenn Roboter bewusst werden, dann könnte es unmoralisch sein, sie weiterhin als „Maschinen" zu behandeln (beispielsweise, sie auszuschalten). Science-Fiction-Autoren haben sich mit den Rechten von Robotern auseinandergesetzt. Der Film *A.I.* (Spielberg, 2001) basiert auf einer Geschichte von Brian Aldiss über einen intelligenten Roboter, der programmiert wurde, zu glauben, er sei ein Mensch, und der die Trennung von seiner Mutter nicht ertragen kann. Die Geschichte und der Film sprechen für die Notwendigkeit einer Bürgerrechtsbewegung für Roboter.

Bibliografische und historische Hinweise

Dieses Kapitel hat die Quellen für die verschiedenen Antworten auf Turings Veröffentlichung von 1950 und für die wichtigsten Kritikpunkte der schwachen künstlichen Intelligenz angegeben. Obwohl es in der Ära nach den neuronalen Netzen modern geworden ist, symbolische Ansätze zu verspotten, stehen nicht alle Philosophen GOFAI kritisch gegenüber. Einige von ihnen sind leidenschaftliche Fürsprecher und sogar Praktiker. Zenon Pylyshyn (1984) argumentierte, dass die Kognition am besten durch ein Computermodell verstanden werden könnte und nicht nur im Prinzip, sondern auch als Möglichkeit, die aktuelle Forschung fortzusetzen, und widerlegte insbesondere die Kritikpunkte von Dreyfus an dem Computermodell der menschlichen Kognition (Pylyshyn, 1974). Gilbert Harmann (1983) erkannte bei der Analyse der Glaubensrevision Verbindungen mit der Forschung in der künstlichen Intelligenz für Wahrheitserhaltungssysteme. Michael Bratman wendete sein „Glauben-Wunsch-Absicht"-Modell der menschlichen Psychologie (Bratman, 1987) auf die KI-Forschung zur Planung an (Bratman, 1992). Als extremer Verfechter der starken künstlichen Intelligenz bezeichnete Aaron Sloman (1978, S. xiii) die Perspektive von Joseph Weizenbaum (Weizenbaum, 1976), dass intelligente Maschinen nicht als Personen betrachtet werden sollten, sogar als „rassistisch".

Zu den Verfechtern der Wichtigkeit von Inkarnation in der Kognition gehören der Philosoph Merleau-Ponty, dessen *Phenomenology of Perception* (1945) die Bedeutung des Körpers und der subjektiven Interpretation der von unseren Sinnen gebotenen Realität betont, und der Philosoph Heidegger, dessen *Being and Time* (1927) fragt, was es eigentlich bedeutet, ein Agent zu sein, und sämtliche Geschichte der Philosophie kritisiert, weil sie dieses Konzept als gegeben hinnimmt. Im Computerzeitalter kommt von Alva Noe (2009) und Andy Clark (1998, 2008) das Konzept, dass unsere Gehirne eine recht minimale Darstellung der Welt bilden, die Welt an sich in Just-in-time-Manier verwenden, um die Vorstellung eines detaillierten internen Modells zu verwalten, und Requisiten in der Welt nutzen (wie zum Beispiel Papier und Bleistift oder Computer), um die Fähigkeiten des Geistes zu erhöhen. Pfeifer et al. (2006) sowie Lakoff und Johnson (1999) liefern Argumente dafür, wie der Körper hilft, Kognition zu prägen.

Die Natur des Bewusstseins war schon immer ein Standardthema für philosophische Betrachtungen, von der Frühzeit bis heute. In der *Phaedo* hat Plato insbesondere die Theorie betrachtet und zurückgewiesen, dass der Verstand ein Organisationsmuster der Einzelteile des Körpers sei, eine Perspektive, die sich der Perspektive der Funktionalisten in der modernen Bewusstseinsphilosophie annähert. Er entschied stattdessen, dass der Verstand eine unsterbliche, immaterielle Seele sein müsse, die vom Körper trennbar ist und eine andere Substanz aufweist – die Perspektive des Dualismus. Aristoteles unterschied eine Vielzahl von Seelen (Griechisch ψυχη) in lebendigen Dingen, die er zum Teil auf funktionalistische Weise beschrieb. Weitere Informationen über den Funktionalismus von Aristoteles finden Sie bei Nussbaum (1978).

Descartes ist berüchtigt für seine dualistische Sichtweise des menschlichen Verstandes, aber ironischerweise griff sein geschichtlicher Einfluss eher im Hinblick auf Mechanismus und Physikalismus. Er stellte sich Tiere explizit als Automaten vor und er sah den Turing-Test voraus, indem er schrieb: „Es ist nicht vorstellbar, dass eine Maschine andere Anordnungen von Wörtern erzeugen sollte, als eine geeignete sinnvolle Antwort auf das zu geben, was in ihrer Gegenwart gesagt wird, selbst der geistloseste Mensch kann dies" (Decartes, 1637). Decartes' energische Verteidigung der Tiere-als-Automaten-Perspektive hatte die Wirkung, dass es einfacher wurde, sich auch den Menschen als Automaten vor-

zustellen, obwohl er selbst diesen Schritt nicht vollzog. Das Buch *L'Homme Machine* (La Mettrie, 1748) erklärte explizit, dass Menschen Automaten sind.

Die moderne analytische Philosophie hat in der Regel den Physikalismus akzeptiert, doch die Verschiedenheit der Ansichten zum Inhalt von mentalen Zuständen ist verwirrend. Die Identifizierung von mentalen Zuständen mit Gehirnzuständen wird für gewöhnlich Place (1956) und Smart (1959) zugeschrieben. Die Debatte zwischen den Ansichten *enger Inhalt* und *weiter Inhalt* von mentalen Zuständen wurde von Hilary Putnam (1975) ausgelöst, der das Gedankenexperiment der **Zwillingserden** (anstelle des Gehirns im Tank, wie im Kapitel beschrieben) als Instrument einführte, identische Gehirnzustände mit unterschiedlichem (weitem) Inhalt zu erzeugen.

Funktionalismus ist die Philosophie des Verstandes, die von der künstlichen Intelligenz fast natürlich vorgeschlagen wurde. Die Idee, dass mentale Zustände Klassen von Gehirnzuständen entsprechen, die funktional definiert sind, geht auf Putnam (1960, 1967) und Lewis (1966, 1980) zurück. Der vielleicht energischste Verfechter des Funktionalismus ist Daniel Dennett, dessen Werk mit dem ehrgeizigen Titel *Consciousness Explained* (Dennett, 1991) viele versuchte Widerlegungen initiiert hat. Metzinger (2009) vermutet, dass es nicht so etwas wie ein objektives *Selbst* gibt, sondern dass Bewusstsein die subjektive Erscheinung einer Welt ist. Das Argument des umgekehrten Spektrums in Bezug auf Qualia wurde von John Locke (1690) eingeführt. Frank Jackson (1982) entwarf ein einflussreiches Gedankenexperiment, bei dem die Wissenschaftlerin Mary in eine vollkommen schwarzweiße Welt versetzt wird. In *There's Something About Mary* (Ludlow et al., 2004) finden Sie eine Artikelsammlung zu diesem Thema.

Der Funktionalismus ist unter Beschuss geraten, und zwar von Autoren, die behaupten, dass sie nicht an die *Qualia* oder den „Was es sein könnte"-Aspekt mentaler Zustände glauben (Nagel, 1974). Searle konzentrierte sich stattdessen auf die angebliche Unfähigkeit des Funktionalismus, Absicht zu berücksichtigen (Searle, 1980, 1984, 1992). Churchland und Churchland (1982) widerlegen beide Kritikformen. Das chinesische Zimmer ist endlos debattiert worden (Searle, 1980, 1990; Preston und Bishop, 2002). Wir erwähnen hier nur eine darauf bezogene Arbeit: In der Science-Fiction-Geschichte *They're Made out of Meat* von Terry Bisson (1990) besuchen außerirdische Erkundungsroboter die Erde und sind ganz verwundert darüber, denkende menschliche Wesen zu finden, deren Verstand aus Fleisch besteht. Vermutlich glaubt der äquivalente Roboter-Alien von Searle, dass er aufgrund der speziellen kausalen Kräfte der Roboter-Schaltkreise denken kann; kausale Kräfte, die reine Fleischgehirne nicht besitzen.

Ethische Fragen in der KI datieren weit früher zurück als das Gebiet an sich. Die Idee der ultraintelligenten Maschine von I. J. Good wurde bereits Hunderte Jahre eher von Samuel Butler (1863) vorhergesehen. Geschrieben vier Jahre nach der Veröffentlichung von Darwins *On the Origins of Species* und zu einer Zeit, als die modernsten Maschinen Dampfmaschinen waren, zeichnete der Artikel von Butler zu *Darwin Among the Machines* ein Bild von „der ultimativen Entwicklung mechanischen Bewusstseins" durch natürliche Auswahl. Das Thema wurde von George Dyson (1998) in einem Buch mit dem gleichen Titel wieder aufgegriffen.

Die philosophische Literatur zu Verstand, Gehirn und verwandten Themen ist umfangreich und manchmal schwierig zu lesen, wenn man die Terminologie oder die Methoden der verwendeten Argumentation nicht kennt. Die *Encyclopedia of Philosophy* (Edwards, 1967) ist ein beeindruckendes und sehr praktisches Hilfsmittel bei diesem Prozess. Das *Cambridge Dictionary of Philosophy* (Audi, 1999) ist eine kürzere und einfacher zu

lesende Arbeit und die online verfügbare *Stanford Encyclopedia of Philosophy* bietet viele ausgezeichnete Artikel und aktuelle Quellenverweise. Die *MIT Encyclopedia of Cognitive Science* (Wilson und Keil, 1999) beschäftigt sich mit der Philosophie des Bewusstseins sowie der Biologie und der Psychologie des Verstandes. Es gibt mehrere allgemeine Einführungen in die philosophische „KI-Frage“ (Boden, 1990; Haugeland, 1985; Copeland, 1993; McCorduck, 2004; Minsky, 2007). Das *Behavioral and Brain Sciences, BBS*, ist eine wichtige Zeitschrift, die sich mit philosophischen und wissenschaftlichen Diskussionen zur künstlichen Intelligenz und Neurowissenschaft beschäftigt. Themen der Ethik und der Verantwortung in der künstlichen Intelligenz werden in Zeitschriften wie etwa *AI and Society* und *Artificial Intelligence and Law* beschrieben.

Zusammenfassung

Dieses Kapitel hat sich mit den folgenden Themen beschäftigt:

- Philosophen verwenden den Begriff **schwache KI** für die Hypothese, dass Maschinen sich möglicherweise intelligent verhalten, und **starke KI** für die Hypothese, dass Maschinen tatsächlich Verstand haben könnten (im Gegensatz zu simuliertem Verstand).
- Alan Turing wies die Frage „Können Maschinen denken?“ zurück und ersetzte sie durch einen Verhaltenstest. Er sah viele der Einwände zu der Möglichkeit denkender Maschinen voraus. Nur wenige KI-Forscher berücksichtigen den Turing-Test und konzentrieren sich eher auf die Leistungen ihrer Systeme für praktische Aufgaben als auf ihre Fähigkeit, Menschen zu imitieren.
- Man ist sich heute darüber einig, dass mentale Zustände Gehirnzustände sind.
- Argumente für und gegen starke KI sind nicht überzeugend. Wenige Forscher im Mainstream-Bereich der KI glauben, dass vom Ergebnis dieser Debatte irgendetwas Wichtiges abhängt.
- Bewusstsein bleibt ein Geheimnis.
- Wir haben sechs mögliche Gefährdungen für die Gesellschaft identifiziert, die sich durch künstliche Intelligenz und verwandte Technologien ergeben. Wir haben geschlossen, dass einige dieser Gefahren entweder unwahrscheinlich sind oder sich nur wenig von den Gefahren unterscheiden, die durch „nicht intelligente“ Technologien entstehen. Insbesondere ist eine Gefahr eine weitere Betrachtung wert: Ultraintelligente Maschinen können zu einer Zukunft führen, die sich von der heutigen Situation wesentlich unterscheidet – vielleicht mögen wir sie nicht, haben aber zu diesem Zeitpunkt eventuell keine Wahl mehr. Derartige Betrachtungen führen unvermeidlich zu der Einsicht, dass wir sorgfältig und binnen kurzem die möglichen Konsequenzen der KI-Forschung abwägen müssen.

Übungen zu Kapitel 26

Lösungshinweise

1. Durchlaufen Sie die von Turing aufgestellte Liste der angeblichen „Unfähigkeiten" von Maschinen und identifizieren Sie, welche davon realisiert werden konnten, welche davon im Prinzip von einem Programm realisiert werden können und welche immer noch problematisch sind, weil sie bewusste mentale Zustände voraussetzen.

2. Versuchen Sie, Definitionen für die Begriffe „Intelligenz", „Denken" und „Bewusstsein" zu verfassen. Zeigen Sie mögliche Einwände gegenüber Ihren Definitionen auf.

3. Beweist eine Widerlegung des Argumentes eines chinesischen Zimmers zwangsläufig, dass geeignet programmierte Computer mentale Zustände aufweisen? Bedeutet die Akzeptanz des Argumentes zwangsläufig, dass Computer keine mentalen Zustände haben können?

4. Beim Argument der Gehirnprothese ist es wichtig, in der Lage zu sein, das Gehirn der Testpersonen auf den Normalzustand zurückzubringen, sodass ihr externes Verhalten, wie es aufgetreten wäre, wenn die Operation nicht stattgefunden hätte, gezeigt wird. Kann der Skeptiker sinnvoll argumentieren, dass es dafür erforderlich wäre, die neurophysiologischen Eigenschaften der Neuronen, die mit bewusster Erfahrung zu tun haben, zu aktualisieren – im Gegensatz zu denjenigen, die an dem funktionalen Verhalten der Neuronen beteiligt sind?

5. Alan Perlis (1982) schrieb: „Ein Jahr, das mit künstlicher Intelligenz zugebracht wird, reicht aus, um jemanden an Gott glauben zu lassen." In einem Brief an Philip Davis schrieb er auch, dass einer der zentralen Träume der Informatik darin besteht, „über die Leistung der Computer und deren Programme jegliche Zweifel darüber zu beseitigen, dass es lediglich eine chemische Unterscheidung zwischen der lebendigen und nicht lebendigen Welt gibt." In welchem Maße hat der bisher in der künstlichen Intelligenz erreichte Fortschritt diese Fragen erhellt? Nehmen Sie an, dass die KI-Bemühungen zu einem zukünftigen Zeitpunkt vollständig erfolgreich sind – das heißt, wir haben intelligente Agenten gebaut, die in der Lage sind, jede menschliche kognitive Aufgabe auf dem Niveau menschlicher Fähigkeiten auszuführen. Bis zu welchem Maße würde dies über die angegebenen Fragen Aufschluss geben?

6. Vergleichen Sie den sozialen Einfluss der künstlichen Intelligenz in den letzten fünfzig Jahren mit den sozialen Auswirkungen, die auf die Einführung von Elektrogeräten und des Verbrennungsmotors in den fünfzig Jahren zwischen 1890 und 1940 zurückzuführen sind.

7. I. J. Good behauptet, dass Intelligenz die wichtigste Qualität ist und dass der Aufbau von ultraintelligenten Maschinen alles verändern wird. Ein empfindungsfähiger Gepard kontert, dass „eigentlich die Geschwindigkeit wichtiger ist; wenn wir ultraschnelle Maschinen bauen könnten, würde dies alles ändern". Und ein empfindungsfähiger Elefant behauptet: „Die beiden vorherigen Aussagen sind falsch; was wir brauchen, sind ultrafeste Maschinen." Was meinen Sie zu diesen Argumenten?

8. Analysieren Sie die möglichen Bedrohungen der Gesellschaft durch die KI-Technologie. Welche Bedrohungen sind die ernsthaftesten und wie könnte man diesen begegnen? Welche potenziellen Vorteile stehen ihnen gegenüber?

9. Wie lassen sich die möglichen Bedrohungen aus der KI-Technologie mit denen aus anderen Informatiktechnologien sowie Bio-, Nano- und Nukleartechnologien vergleichen?

10. Manche Kritiker argumentieren, dass KI unmöglich sei, während andere argumentieren, dass sie *allzu* möglich sei und dass ultraintelligente Maschinen eine Bedrohung darstellten. Welches dieser Argumente halten Sie für am wahrscheinlichsten? Wäre es ein Widerspruch, beide Positionen zu vertreten?

KI: Gegenwart und Zukunft

27

ÜBERBLICK

In diesem Kapitel fragen wir uns, wo wir stehen und wohin wir gehen – ein guter Ausgangspunkt für weitere Arbeiten.

Wie *Kapitel 2* erläutert hat, ist es hilfreich, die KI-Aufgabe als den Entwurf rationaler Agenten zu betrachten – das heißt Agenten, deren Aktionen ihren erwarteten Nutzen bei gegebener Wahrnehmungsfolge maximieren. Wir haben gezeigt, dass das Entwurfsproblem abhängt von den Wahrnehmungen und Aktionen, die dem Agenten zur Verfügung stehen, von der Nutzenfunktion, die das Verhalten des Agenten erreichen soll, sowie von der Natur der Umgebung. Es gibt eine Vielzahl unterschiedlicher Agentenentwürfe, von Reflexagenten bis hin zu vollständig deliberativen, wissensbasierten, entscheidungstheoretischen Agenten. Darüber hinaus können die Komponenten dieser Entwürfe die unterschiedlichsten Instantiierungen haben – z.B. logisches oder probabilistisches Schließen sowie atomare, faktorisierte oder strukturierte Darstellungen von Zuständen. Die dazwischenliegenden Kapitel haben die Grundlagen beschrieben, wie diese Komponenten arbeiten.

Tipp

Sowohl nach unserem wissenschaftlichen Verständnis als auch nach unseren technologischen Fähigkeiten weisen alle Agentenentwürfe und -komponenten wesentliche Fortschritte auf. In diesem Kapitel wollen wir uns einen Schritt von den Details entfernen und fragen: „Führen alle diese Programme zu einem universellen intelligenten Agenten, der in den unterschiedlichsten Umgebungen eine gute Leistung bringt?“ *Abschnitt 27.1* betrachtet die Komponenten eines intelligenten Agenten, um einschätzen zu können, was bekannt ist und was fehlt. *Abschnitt 27.2* macht dasselbe für die allgemeine Agentenarchitektur. *Abschnitt 27.3* fragt, ob der Entwurf rationaler Agenten überhaupt das richtige Ziel ist. (Die Antwort lautet: „Nicht wirklich, aber im Moment ist es O.K.“) Schließlich untersucht *Abschnitt 27.4*, welche Konsequenzen der Erfolg unserer Bemühungen hat.

27.1 Agentenkomponenten

Kapitel 2 hat mehrere Agentenentwürfe sowie ihre Komponenten vorgestellt. Um unsere Diskussion hier zu konzentrieren, betrachten wir den nutzenbasierten Agenten, den wir in ▶ Abbildung 27.1 noch einmal zeigen. Ausgerüstet mit einer Lernkomponente (Abbildung 2.15) ist dies ein ganz allgemeiner Agentenentwurf. Wir zeigen nun, wie der Stand der Technik bei den einzelnen Komponenten aussieht.

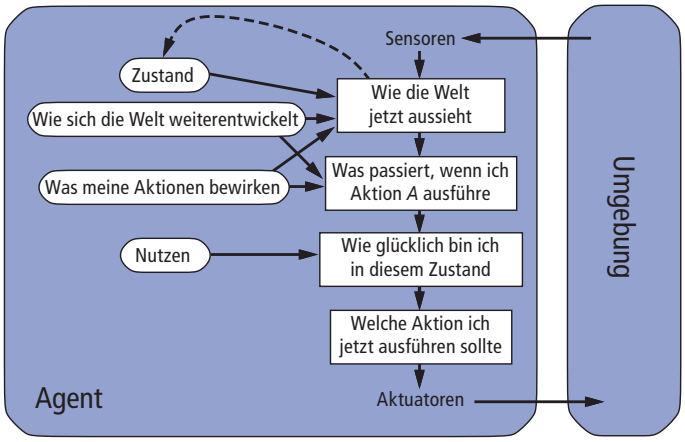


Abbildung 27.1: Ein modellbasierter, nutzenbasierter Agent, wie Sie ihn bereits in Abbildung 2.14 gesehen haben.

Zusammenarbeit mit der Umgebung durch Sensoren und Aktuatoren: In weiten Bereichen der Geschichte der künstlichen Intelligenz war dies ein deutlicher Schwachpunkt. Mit wenigen bemerkenswerten Ausnahmen wurden KI-Systeme so aufgebaut, dass Menschen die Eingaben bereitstellen und die Ausgaben interpretieren mussten, während sich Robotiksysteme auf Low-Level-Aufgaben konzentrierten, wo das Schließen und Planen auf höchster Ebene größtenteils fehlte. Das lag zum Teil an den hohen Kosten und dem Entwicklungsaufwand für die Erstellung realer Roboter. Die Situation hat sich in den letzten Jahren schnell geändert. Es gibt bereits fertige programmierbare Roboter. Diese wiederum haben von den kleinen, billigen, hochauflösenden CCD-Kameras und den kompakten, zuverlässigen Motorlaufwerken profitiert. So hat die MEMS-Technologie (Micro-Electromechanical Systems) Miniaturausgaben von Beschleunigungsmessern, Gyroskopen und Aktuatoren für ein künstliches fliegendes Insekt realisiert (Floreano et al., 2009). Es ließen sich auch Millionen von MEMS-Bauelementen kombinieren, um sehr leistungsfähige makroskopische Aktuatoren zu produzieren.

KI-Systeme befinden sich also an der Schwelle eines Überganges von hauptsächlich auf Software basierenden Systemen zu eingebetteten Robotersystemen. Der Status heutiger Robotik ist grob vergleichbar mit dem Status der Personalcomputer um 1980: Zu dieser Zeit konnten Forscher und Hobby-Bastler mit PCs experimentieren, doch es sollte noch ein Jahrzehnt dauern, bevor sie zu einer alltäglichen Erscheinung wurden.

Beobachtung des Zustandes der Welt: Dies ist eine der wichtigsten Fähigkeiten, die ein intelligenter Agent besitzen muss. Dafür sind sowohl die Wahrnehmung als auch die Aktualisierung interner Repräsentationen erforderlich. *Kapitel 4* hat gezeigt, wie sich atomare Zustandsdarstellungen verfolgen lassen; in *Kapitel 7* wurde erläutert, wie sich dies für faktorisierte (aussagenlogische) Zustandsdarstellungen realisieren lässt. *Kapitel 12* hat dies auf die Logik erster Stufe erweitert und in *Kapitel 15* wurden **Filteralgorithmen** für probabilistisches Schließen in unsicheren Umgebungen beschrieben. Aktuelle Filter- und Wahrnehmungsalgorithmen lassen sich kombinieren, um eine ausreichende Leistung für die Bereitstellung von Low-Level-Prädikaten zu erzielen, wie beispielsweise „Die Tasse steht auf dem Tisch“. Schwieriger ist schon die Erkennung von Aktionen auf höherer Ebene, wie etwa: „Dr. Russell trinkt eine Tasse Tee mit Dr. Norvig, während sie Pläne für nächste Woche diskutieren.“ Momentan ist das nur mithilfe von kommentierten (annotierten) Beispielen möglich (siehe dazu Abbildung 24.25 in *Abschnitt 24.6*).

Ein weiteres Problem ist, dass zwar die annähernden Filteralgorithmen aus *Kapitel 15* relativ große Umgebungen verarbeiten können, dass sie im Wesentlichen aber immer noch mit einer faktorisierten Darstellung zu tun haben – sie verwenden Zufallsvariablen, stellen aber weder Objekte noch Relationen explizit dar. *Abschnitt 14.6* hat erklärt, wie Wahrscheinlichkeit und Logik erster Stufe kombiniert werden können, um dieses Problem zu lösen, und *Abschnitt 14.6.3* hat gezeigt, wie wir Unbestimmtheit über die Identität von Objekten behandeln können. Wir erwarten, dass die Anwendung dieser Ideen für die Verfolgung komplexer Umgebungen zu erheblichen Vorteilen führt. Allerdings stehen wir immer noch vor der gewaltigen Aufgabe, allgemeine, wieder verwendbare Schemas für komplexe Domänen zu definieren. Wie *Kapitel 12* diskutiert hat, wissen wir noch nicht, wie sich dies im Allgemeinen realisieren lässt; nur für isolierte, einfache Domänen gibt es Lösungen. Möglicherweise lassen sich Fortschritte erzielen, wenn wir uns verstärkt auf probabilistische statt auf logische Darstellungen gekoppelt mit aggressivem Maschinenlernen konzentrieren (anstatt das Wissen fest zu kodieren).

Projektieren, Auswerten und Auswählen zukünftiger Aktionsfolgen: Die Anforderungen für die grundlegende Wissensrepräsentation sind hier dieselben wie für die Beobachtung der Welt: Die größte Schwierigkeit ist, mit Aktionsverläufen zurechtzukommen – beispielsweise eine Konversation zu führen oder eine Tasse Tee zu trinken –, die letztlich aus Tausenden oder Millionen elementarer Schritte für einen realen Agenten bestehen. Nur durch Einführung einer **hierarchischen Struktur** des Verhaltens kommen die Menschen überhaupt zurecht. *Abschnitt 11.2* hat gezeigt, wie sich Probleme dieser Größenordnung mithilfe von hierarchischen Darstellungen in den Griff bekommen lassen. Darüber hinaus haben die Arbeiten zum hierarchischen Reinforcement-Learning einige dieser Ideen mit den in *Kapitel 17* beschriebenen Techniken für die Entscheidungsfindung unter Unsicherheit erfolgreich kombiniert. Bis jetzt nutzen die Algorithmen für den partiell beobachtbaren Fall (POMDPs) die gleiche atomare Zustandsdarstellung, wie wir sie für die Suchalgorithmen von *Kapitel 3* verwendet haben. Hier gibt es offenbar noch viel zu tun, doch die technischen Grundlagen sind im Wesentlichen gelegt. *Abschnitt 27.2* beschäftigt sich mit der Frage, wie die Suche für effektive weitreichende Pläne gesteuert werden kann.

Nutzen als Ausdruck von Prioritäten: Im Prinzip ist es ein ganz allgemeines Vorgehen, rationale Entscheidungen auf der Maximierung des erwarteten Nutzens zu basieren, und man vermeidet damit viele Probleme rein zielbasierter Ansätze, wie beispielsweise kollidierende Ziele und unsichere Fertigkeiten. Bisher gab es jedoch relativ wenig Arbeiten zur Konstruktion *realistischer* Nutzenfunktionen – stellen Sie sich beispielsweise das komplexe Netz miteinander verwobener Prioritäten vor, das ein Agent verstehen muss, der als Bürogehilfe für einen Menschen arbeitet. Es hat sich als sehr schwierig erwiesen, Prioritäten über komplexe Zustände auf dieselbe Weise zu zerlegen, wie Bayessche Netze Glauben über komplexe Zustände zerlegen. Ein Grund dafür könnte sein, dass Prioritäten über Zustände eigentlich aus Prioritäten über Zustandsverläufe *kompiliert* sind, die durch **Gewinnfunktionen** beschrieben sind (siehe *Kapitel 17*). Selbst wenn die Gewinnfunktion einfach ist, kann die entsprechende Nutzenfunktion sehr komplex sein. Das bedeutet, wir ziehen die Aufgabe des Wissens-Engineerings für Gewinnfunktionen als ernste Möglichkeit in Betracht, unseren Agenten mitzuteilen, was wir von ihnen wollen.

Lernen: In den *Kapiteln 18* bis *21* haben wir beschrieben, wie das Lernen in einem Agenten als interaktives Lernen (überwacht, nicht überwacht oder verstärkend) der Funktionen formuliert werden kann, die die verschiedenen Komponenten des Agenten bilden. Es wurden sehr leistungsfähige logische und statistische Techniken entwickelt, die mit relativ großen Problemen zurechtkommen und menschliche Fähigkeiten in vielen Aufgaben erreichen oder überschreiten – solange wir es mit einem vordefinierten Vokabular von Merkmalen und Konzepten zu tun haben. Andererseits hat das maschinelle Lernen wenig Fortschritte für das wichtige Problem des Aufbaus neuer Repräsentationen zu verzeichnen, die auf höheren Abstraktionsebenen angeordnet sind als das Eingabevokabular. Zum Beispiel wäre in der Computervision das Lernen komplexer Konzepte wie *Klassenzimmer* und *Cafeteria* unnötig schwierig, wenn der Agent gezwungen würde, mit Pixeln als Eingabedarstellung zu arbeiten. Stattdessen muss der Agent in der Lage sein, ohne menschliche Überwachung zunächst Zwischenkonzepte

zu bilden, beispielsweise *Schreibtisch* und *Tablett*. Ähnliche Betrachtungen gelten auch für das Lernverhalten: *EineTasseTeeTrinken* ist ein sehr wichtiger High-Level-Schritt in vielen Plänen, aber wie bringt man sie in eine Aktionsbibliothek, die anfänglich sehr viel einfachere Aktionen enthält, wie beispielsweise *ArmHeben* oder *Schlucken*? Vielleicht bindet dieses Konzept einige der Ideen von **Deep-Belief-Netzen** ein – Bayesschen Netzen, die mehrere Schichten mit versteckten Variablen enthalten, wie in der Arbeit von Hinton et al. (2006), Hawkins und Blakeslee (2004) sowie Bengio und LeCun (2007).

Größtenteils geht heute die Forschung zum maschinellen Lernen von einer faktorisierten Darstellung aus, wobei eine Funktion $h : \mathbb{R}^n \rightarrow \mathbb{R}$ für die Regression und $h : \mathbb{R}^n \rightarrow \{0, 1\}$ für die Klassifizierung gelernt wird. Lernforscher müssen ihre sehr erfolgreichen Techniken für faktorisierte Darstellungen an strukturierte Darstellungen und insbesondere hierarchische Darstellungen anpassen. Die Arbeiten zur induktiven Logikprogrammierung in *Kapitel 19* bilden einen ersten Schritt in diese Richtung; der logische nächste Schritt muss diese Ideen mit den probabilistischen Sprachen von *Abschnitt 14.6* zusammenfassen.

Wenn wir diese Aspekte nicht verstehen, sind wir mit der entmutigenden Aufgabe konfrontiert, große Wissensbasen mit Allgemeinwissen von Hand aufzubauen, ein Ansatz, der bislang nicht gut gelaufen ist. Vielversprechend ist die Verwendung des Webs als Quelle für natürlichsprachigen Text, Bilder und Videos, um sie als umfangreiche Wissensbasis zu nutzen. Doch bisher sind die Algorithmen für das maschinelle Lernen in Bezug auf den Umfang des organisierten Wissens, das sie aus diesen Quellen extrahieren können, beschränkt.

27.2 Agentenarchitekturen

Die Frage „Welche der in *Kapitel 2* vorgestellten Agentenarchitekturen soll ein Agent benutzen?“ ist ganz natürlich. Die Antwort lautet: „Alle!“ Wir haben gesehen, dass Reflexagenten für Situationen benötigt werden, wo die Zeit eine Rolle spielt, während es ein wissensbasiertes Nachdenken (Deliberation) dem Agenten erlaubt, vorauszuplanen. Ein vollständiger Agent muss in der Lage sein, beides zu tun – unter Verwendung einer **hybriden Architektur**. Eine wichtige Eigenschaft hybrider Architekturen ist, dass die Grenzen zwischen den verschiedenen Entscheidungskomponenten nicht feststehend sind. Beispielsweise wandelt die **Kompilierung** ständig deklarative Informationen auf der deliberativen Ebene in effizientere Repräsentationen um und erreicht irgendwann die Reflexebene – siehe ▶ Abbildung 27.2. (Dies ist der Zweck des erklärungsbasierten Lernens, wie in *Kapitel 19* beschrieben.) Agentenarchitekturen wie beispielsweise SOAR (Laird et al., 1987) und THEO (Mitchell, 1990) weisen genau diese Struktur auf. Immer wenn sie ein Problem durch explizite Deliberation lösen, speichern sie eine verallgemeinerte Version der Lösung für die Verwendung durch die Reflexkomponente. Ein weniger untersuchtes Problem ist die *Umkehrung* dieses Prozesses: Wenn sich die Umgebung ändert, sind gelernte Reflexe vielleicht nicht mehr geeignet und der Agent muss auf die deliberative Ebene zurückkehren, um neue Verhaltensweisen zu produzieren.

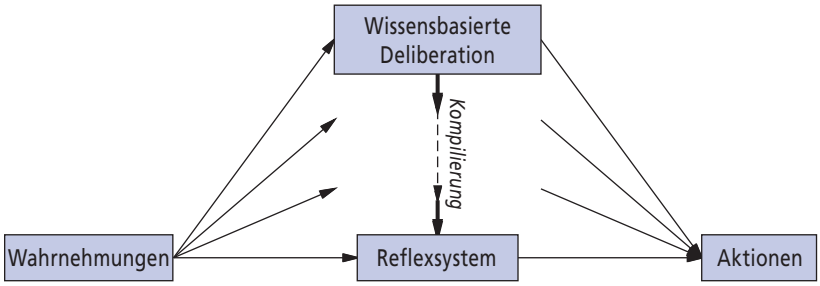


Abbildung 27.2: Die Kompilierung dient dazu, eine deliberative Entscheidungsfindung in effizientere, reflexive Mechanismen umzuwandeln.

Agenten brauchen auch Möglichkeiten, ihre eigenen Deliberationen zu steuern. Sie müssen in der Lage sein, dieses Nachdenken zu stoppen, wenn es erforderlich ist, und sie müssen in der Lage sein, die Zeit zu nutzen, die ihnen für die Deliberation zur Verfügung steht, um die lohnendsten Berechnungen auszuführen. Zum Beispiel muss ein Taxi fahrender Agent, der einen Unfall sieht, im Bruchteil einer Sekunde entscheiden, ob er bremst oder ausweicht. Außerdem sollte er in diesem Sekundenbruchteil über die wichtigsten Fragen nachdenken, beispielsweise ob die Spuren links und rechts frei sind oder ob ein großer Lastwagen direkt hinter ihm fährt, anstatt sich um den Zustand der Reifen zu kümmern oder zu überlegen, wo er den nächsten Fahrgast einsteigen lassen soll. Diese Aspekte werden normalerweise unter der Überschrift der **Echtzeit-KI** untersucht. Wenn KI-Systeme in komplexere Domänen vordringen, werden alle Probleme zu Echtzeitproblemen, weil der Agent nie lange genug Zeit hat, um das Entscheidungsproblem genau zu lösen.

Zweifellos besteht ein dringender Bedarf an *allgemeinen* Methoden für die Steuerung der Deliberation und nicht an spezifischen Rezepten, woran in jeder Situation zu denken ist. Die erste nützliche Idee ist die Nutzung des **Anytime-Algorithmus** (Dean und Boddy, 1988; Horvitz, 1987). Ein Anytime-Algorithmus ist ein Algorithmus, dessen Ausgabequalität sich mit der Zeit schrittweise verbessert, sodass er eine sinnvolle Entscheidung präsentieren kann, wann immer er unterbrochen wird. Derartige Algorithmen werden durch eine Entscheidungsprozedur auf **Metaebene** gesteuert, die abschätzt, ob eine weitere Berechnung sinnvoll ist. Die iterativ vertiefende Suche beim Spielen zeigt ein einfaches Beispiel für einen Anytime-Algorithmus. (In *Abschnitt 3.5.4* finden Sie eine kurze Beschreibung für die Entscheidungsfindung auf Metaebene.) Als Beispiele für einen Anytime-Algorithmus sind iteratives Vertiefen in der Spielbaumsuche und MCMC in Bayesschen Netzen zu nennen.

Die zweite Technik zum Steuern der Deliberation ist das entscheidungstheoretische Metaschließen (Russell und Wefald, 1989, 1991; Horvitz, 1989; Horvitz und Breese, 1996). Diese Methode wendet für die Auswahl einzelner Berechnungen die Theorie des Informationswertes an (*Kapitel 16*). Der Wert einer Berechnung ist sowohl von ihren Kosten (im Hinblick auf die Verzögerung der Aktion) als auch von ihren Vorteilen (im Hinblick auf eine verbesserte Entscheidungsqualität) abhängig. Techniken für das Metaschließen können genutzt werden, um bessere Suchalgorithmen zu entwickeln und zu garantieren, dass diese Algorithmen die Anytime-Eigenschaft besitzen. Das Metaschließen ist natürlich teuer und es können Kompilierungsmethoden angewendet werden, sodass der Zusatzaufwand im Vergleich zu den Kosten für die zu kontrollierenden Berechnungen klein ist. Reinforcement-Learning auf Metaebene ist eine

andere Möglichkeit, um effektive Strategien zum Steuern der Deliberation zu erhalten: Im Wesentlichen werden Berechnungen gefördert, die zu besseren Entscheidungen führen, während diejenigen, die sich als wirkungslos erweisen, bestraft werden. Dieser Ansatz vermeidet die Probleme, die sich durch die kurzsichtige Berechnung eines einfachen Informationswertes ergeben.

Das Metaschließen ist ein spezifisches Beispiel für eine **reflexive Architektur** – d.h. für eine Architektur, die Deliberation über die Recheneinheiten und Aktionen erlaubt, die innerhalb der eigentlichen Architektur auftreten. Eine theoretische Grundlage für reflexive Architekturen kann erstellt werden, indem man einen gemeinsamen Zustandsraum definiert, der sich aus dem Umgebungszustand und dem Rechenzustand des eigentlichen Agenten zusammensetzt. Algorithmen für Entscheidungsfindung und Lernen lassen sich so entwerfen, dass sie auf diesem gemeinsamen Zustandsraum arbeiten und dabei herangezogen werden, um die Berechnungsaktivitäten des Agenten zu implementieren und zu verbessern. Letztlich erwarten wir, dass aufgabenspezifische Algorithmen wie Alpha-Beta-Suche und Rückwärtsverkettung aus den KI-Systemen verschwinden und durch allgemeine Methoden ersetzt werden, die diese Berechnungen des Agenten in die Richtung einer effizienten Erstellung hochqualitativer Entscheidungen lenken.

27.3 Gehen wir in die richtige Richtung?

Der vorige Abschnitt hat einige Vorteile sowie viele Gelegenheiten für weiteren Fortschritt aufgezeigt. Aber wohin führt das alles? Dreyfus (1992) nennt die Analogie, zu versuchen, auf den Mond zu gelangen, indem man auf einen Baum steigt. Man kann einen stetigen Fortschritt berichten, solange man auf dem Weg zum Baumgipfel ist. In diesem Abschnitt betrachten wir, ob der aktuelle Weg der künstlichen Intelligenz mehr wie eine Baumbesteigung oder mehr wie eine Raketenfahrt ist.

In *Kapitel 1* haben wir gesagt, unser Ziel sei, Agenten zu erstellen, die *rational handeln*. Wir haben jedoch auch gesagt, dass

> *... die Erzielung perfekter Rationalität – immer das Richtige zu tun – in komplizierten Umgebungen nicht möglich ist. Die Rechenanforderungen sind einfach zu hoch. Für einen Großteil des Buches wollen wir jedoch die Arbeitshypothese annehmen, dass die perfekte Rationalität ein guter Ausgangspunkt für die Analyse ist.*

Jetzt wollen wir noch einmal betrachten, wie genau das Ziel der künstlichen Intelligenz aussieht. Wir wollen Agenten erstellen, aber welche Spezifikation haben wir uns dafür vorgestellt? Hier sind vier Möglichkeiten:

Perfekte Rationalität. Ein perfekt rationaler Agent handelt in jedem Moment so, dass er seinen erwarteten Nutzen für die Information, die er aus der Umgebung erhalten hat, maximieren kann. Wir haben gesehen, dass die Berechnungen, die für die Erzielung perfekter Rationalität in den meisten Umgebungen erforderlich sind, zu zeitaufwändig sind – eine perfekte Rationalität ist also kein realistisches Ziel.

Kalkulatorische Rationalität. Dies ist das Konzept einer Rationalität, die wir implizit verwendet haben, als wir logische und entscheidungstheoretische Agenten entworfen haben. Zudem hat sich der größte Teil der theoretischen KI-Forschung auf diese Eigenschaft konzentriert. Ein kalkulatorisch rationaler Agent gibt *irgendwann* zurück, was die rationale Auswahl am Anfang seines Nachdenkens *gewesen wäre*. Dies ist eine

interessante Eigenschaft für ein System, aber in den meisten Umgebungen hat die richtige Antwort zum falschen Zeitpunkt keinen Wert. In der Praxis sind die Entwickler von KI-Systemen gezwungen, Kompromisse im Hinblick auf die Entscheidungsqualität zu treffen, um eine vernünftige Gesamtleistung zu erhalten. Leider bietet die theoretische Grundlage der kalkulatorischen Rationalität keine sinnvolle Möglichkeit, solche Kompromisse zu machen.

Begrenzte Rationalität. Herbert Simon (1957) widerlegte das Konzept der perfekten (oder sogar annähernd perfekten) Rationalität und ersetzte sie durch begrenzte Rationalität, eine deskriptive Theorie der Entscheidungsfindung realer Agenten. Er schrieb:

> *Die Fähigkeit des menschlichen Verstandes, komplexe Probleme zu formulieren und zu lösen, ist sehr klein im Vergleich zu der Größe der Probleme, deren Lösung man für gezieltes rationales Verhalten in der realen Welt benötigt – oder sogar für eine sinnvolle Annäherung zu einer solchen gezielten Rationalität.*

Er sagte, dass die begrenzte Rationalität hauptsächlich versucht, **befriedigende** Ergebnisse zu erzielen – d.h., sie denkt nur so lange nach, bis sie eine Antwort gefunden hat, die „gut genug" ist. Simon gewann mit dieser Arbeit den Nobelpreis für Wirtschaft und hat ausführlich darüber geschrieben (Simon, 1982). Es scheint in vielen Fällen ein sinnvolles Modell menschlichen Verhaltens zu sein. Es ist keine formale Spezifikation für intelligente Agenten, weil die Definition von „gut genug" durch die Theorie nicht vorgegeben wird. Darüber hinaus scheint die Befriedigung einfach nur ein großer Bereich von Methoden zu sein, der verwendet wird, um mit begrenzten Ressourcen zurechtzukommen.

Begrenzte Optimalität (BO). Ein begrenzt optimaler Agent verhält sich im Rahmen seiner *vorgegebenen Rechenressourcen* so gut wie möglich. Das bedeutet, dass der erwartete Nutzen des eigenen Programms für einen begrenzt optimalen Agenten mindestens so groß ist wie der erwartete Nutzen jedes andere Agentenprogramms, das auf derselben Maschine ausgeführt wird.

Von diesen vier Möglichkeiten scheint die begrenzte Optimalität die beste Hoffnung für eine starke theoretische Grundlage der künstlichen Intelligenz zu sein. Sie hat den Vorteil, dass sie realisiert werden kann: Es gibt immer mindestens ein bestes Programm – etwas, was bei der perfekten Rationalität fehlt. Begrenzt optimale Agenten sind in der realen Welt wirklich praktisch, während kalkulatorisch relationale Agenten dies in der Regel nicht sind, und sie könnten befriedigende Agenten sein oder nicht – abhängig davon, wie ambitioniert sie sind.

Der traditionelle Ansatz in der künstlichen Intelligenz beginnt mit kalkulatorischer Rationalität und geht dann Kompromisse ein, um die beschränkten Ressourcen zu berücksichtigen. Wenn die durch die Beschränkungen vorgegebenen Probleme gering sind, erwartet man, dass der endgültige Entwurf einem BO-Agentenentwurf ähnlich ist. Wenn die Ressourcenbeschränkungen jedoch kritisch werden – wenn etwa die Umgebung komplexer wird –, erwartet man, dass die beiden Entwürfe divergieren. In der Theorie der begrenzten Optimalität können diese Beschränkungen prinzipiell verarbeitet werden.

Bisher weiß man noch wenig über begrenzte Optimalität. Es ist möglich, begrenzt optimale Programme für sehr einfache Maschinen und verhältnismäßig eingeschränkte Umgebungen zu erstellen (Etzioni, 1989; Russell et al., 1993), aber bisher wissen wir nicht, wie BO-Programme für große, allgemeine Computer in komplexer Umgebung aussehen. Wenn es eine konstruktive Theorie begrenzter Optimalität geben soll, haben wir

die Hoffnung, dass der Entwurf begrenzt optimaler Programme nicht zu sehr von den Details des verwendeten Computers abhängig ist. Es würde die wissenschaftliche Forschung sehr erschweren, wenn das Hinzufügen von ein paar Kilobyte Speicher in einer Gigabyte-Maschine einen wesentlichen Unterschied für den Entwurf des BO-Programms bedeuten würde. Eine Möglichkeit, wie man sicherstellen kann, dass dies nicht passiert, ist das etwas gelockerte Kriterium für begrenzte Optimalität. Durch die Analogie zu dem Konzept der asymptotischen Komplexität (Anhang A) können wir die **asymptotisch begrenzte Optimalität** (ABO) wie folgt definieren (Russell und Subramanian, 1995). Es sei ein Programm P optimal für eine Maschine M in einer Klasse von Umgebungen $\mathbf{E}$, wobei die Komplexität der Umgebungen in $\mathbf{E}$ unbegrenzt ist. Das Programm P' ist dann ABO für M in $\mathbf{E}$, wenn es besser ist als P, indem es auf einer Maschine kM ausgeführt wird, die k-mal schneller (oder größer) als M ist. Wenn k nicht gerade riesig ist, wären wir glücklich mit einem Programm, das ABO für nicht triviale Umgebungen auf einer nicht trivialen Architektur ist. Es hat wenig Sinn, einen riesigen Aufwand zu betreiben, um BO- statt ABO-Programme zu finden, weil Größe und Geschwindigkeit der verfügbaren Maschinen ohnehin mit der Zeit um einen konstanten Faktor wachsen.

Es ist zu vermuten, dass BO- oder ABO-Programme für leistungsfähige Computer in komplexen Umgebungen nicht unbedingt eine einfache, elegante Struktur aufweisen. Wir haben bereits gesehen, dass universelle Intelligenz eine gewisse Reflexfähigkeit und deliberative Fähigkeit, die unterschiedlichsten Arten von Wissen und Entscheidungsfindung, Lernen und Kompilierungsmethoden für alle diese Formen, Methoden für die Steuerung des Schließens sowie einen großen Speicher domänenspezifischen Wissens bedingt. Ein begrenzt optimaler Agent muss sich der Umgebung anpassen, in der er sich befindet, sodass sein interner Aufbau irgendwann Optimierungen reflektiert, die für die jeweilige Umgebung spezifisch sind. Dies kann nur erwartet werden und entspricht ungefähr der Weise, wie Rennautos, die in ihrer Motorgröße beschränkt sind, zu extrem komplexen Entwürfen geführt haben. Vermutlich wird eine Wissenschaft der künstlichen Intelligenz, die auf begrenzter Optimalität basiert, viele Prozesse untersuchen müssen, durch die ein Agentenprogramm zu begrenzter Optimalität konvergieren kann, und sich weniger mit den Details der daraus entstehenden unübersichtlichen Programme befassen.

Insgesamt stellt das Konzept der begrenzten Optimalität eine formale Aufgabe für die KI-Forschung dar, die sowohl genau umrissen als auch machbar ist. Begrenzte Optimalität spezifiziert optimale *Programme* statt optimaler *Aktionen*. Aktionen werden letztlich von Programmen erzeugt und es sind die Programme, über die Entwickler die Kontrolle haben.

27.4 Was passiert, wenn die KI erfolgreich ist?

In seinem Roman *Small World* (1984) beschreibt David Lodge die akademische Welt der Literaturkritik, wobei der Protagonist Verwirrung verursacht, indem er mehreren berühmten Literaturkritikern die folgende Frage stellt: „Was wäre, wenn Sie recht hätten?“ Keiner der Theoretiker scheint sich zuvor mit dieser Frage beschäftigt zu haben, weil vielleicht die Diskussion unfehlbarer Theorien bereits am Ende ist. Ähnliche Verwirrung lässt sich hervorrufen, wenn man KI-Forscher fragt: „Was machen Sie, wenn Sie erfolgreich sind?“

Wie in *Abschnitt 26.3* beschrieben, sind einige ethische Fragen zu berücksichtigen. Intelligente Computer sind leistungsfähiger als ihre dummen Vertreter, aber wird diese Leistung für Gutes oder Böses verwendet? Die Entwickler künstlicher Intelligenz haben die Verantwortung, darauf zu achten, dass die Wirkung ihrer Arbeit positiv ist. Der Einflussbereich ist von dem Erfolgsgrad der künstlichen Intelligenz abhängig. Selbst mittlere Erfolge in der künstlichen Intelligenz haben bereits Änderungen dahingehend bewirkt, wie Informatik gelehrt (Stein, 2002) und Softwareentwicklung praktiziert wird. Die künstliche Intelligenz hat neue Anwendungen ermöglicht, wie beispielsweise Spracherkennungssysteme, Inventarsysteme, Überwachungssysteme, Roboter und Suchmaschinen.

Wir können erwarten, dass mittlere Erfolge in der künstlichen Intelligenz alle Menschen in ihrem Alltagsleben beeinflussen. Bisher hatten computergestützte Kommunikationsnetzwerke wie beispielsweise die Mobiltelefone und das Internet diese Art der durchdringenden Wirkung auf die Gesellschaft, die künstliche Intelligenz jedoch nicht. KI hat hinter den Kulissen gearbeitet – zum Beispiel bei der automatischen Annahme oder Ablehnung von Kreditkartentransaktionen für fast jede Kaufhandlung im Web –, ist aber für den durchschnittlichen Konsumenten nicht sichtbar gewesen. Wir können uns vorstellen, dass wirklich praktische persönliche Assistenten für das Büro oder den Haushalt einen großen positiven Einfluss auf das Leben der Menschen hätten, wenn dadurch auch für kurze Zeit die Wirtschaft aus dem Gleichgewicht geriete. Automatisierte Assistenten für das Autofahren könnten Unfälle verhindern und jedes Jahr zigtausend Leben retten. Eine technologische Möglichkeit dieser Art könnte auch auf die Entwicklung autonomer Waffen angewendet werden, was viele wiederum als nicht erwünschte Entwicklung betrachten. Einige der größten gesellschaftlichen Probleme in der heutigen Zeit – wie zum Beispiel die Nutzung genetischer Informationen zur Behandlung von Krankheiten, die effiziente Verwaltung von Energieressourcen und die Verifizierung von Verträgen über Nuklearwaffen – werden mithilfe der KI-Technologien angegangen.

Schließlich scheint es wahrscheinlich, dass große Erfolge in der künstlichen Intelligenz – die Schaffung von Intelligenz auf Ebene des Menschen und darüber hinaus – das Leben der Menschheit als Ganzes verändern würde. Die Natur unserer Arbeit und unserer Spiele würde sich verändern, ebenso wie unsere Betrachtung von Intelligenz, Bewusstsein und Schicksal der menschlichen Rasse. KI-Systeme auf diesem Fähigkeitsniveau könnten die menschliche Autonomie, Freiheit und sogar das Überleben direkt bedrohen. Aus diesen Gründen können wir die KI-Forschung nicht von den ethischen Konsequenzen trennen (siehe *Abschnitt 26.3*).

Wohin geht die Zukunft? Science-Fiction-Autoren scheinen dystopische Zukunftsbilder utopischen vorzuziehen – vielleicht weil daraus interessantere Geschichten entstehen. Bisher scheint die künstliche Intelligenz gut zu anderen revolutionären Technologien zu passen (Buchdruck, Flugverkehr, Fernsprechwesen), deren negative Wirkungen durch ihre positiven Aspekte aufgewogen werden.

Insgesamt sehen wir, dass die künstliche Intelligenz in ihrer kurzen Geschichte große Fortschritte gemacht hat, aber der letzte Satz im Aufsatz *Computing Machinery and Intelligence* von Alan Turing (1950) gilt heute noch immer:

Wir können nicht weit in die Zukunft sehen, aber wir können sehen, dass noch viel zu tun ist.

Mathematischer Hintergrund

A

ÜBERBLICK

A.1 Komplexitätsanalyse und O()-Notation

Informatiker stehen häufig der Aufgabe gegenüber, Algorithmen zu vergleichen, um zu sehen, wie schnell sie ausgeführt werden oder wie viel Speicher sie brauchen. Für diese Aufgabe gibt es zwei Lösungsansätze. Der erste verwendet **Benchmarks** – die Algorithmen werden auf einem Computer ausgeführt und die Geschwindigkeit wird in Sekunden, der Speicherverbrauch in Byte gemessen. Letztlich zählt gerade dies, doch kann ein Benchmark-Test unbefriedigend sein, weil er so spezifisch ist: Er misst die Leistung eines bestimmten Programms, das in einer bestimmten Sprache geschrieben, auf einem bestimmten Computer ausgeführt und mit einem bestimmten Compiler übersetzt wurde und das bestimmte Eingabedaten verwendet. Aus dem Einzelergebnis, das dabei entsteht, lässt sich unter Umständen nur schwer vorhersagen, wie gut sich der Algorithmus mit einem anderen Compiler, auf einem anderen Computer oder mit einer anderen Datenmenge verhält. Der zweite Ansatz basiert auf einer **mathematischen Analyse** von Algorithmen, die unabhängig von der jeweiligen Implementierung und Eingabe ist. Damit beschäftigen sich die nachstehenden Abschnitte.

A.1.1 Asymptotische Analyse

In diesem Abschnitt betrachten wir die Algorithmenanalyse anhand des folgenden Beispiels. Es handelt sich dabei um ein Programm, das die Summe einer Zahlenfolge berechnet:

```
function SUMMATION(sequence) returns eine Zahl
   sum ← 0
   for i ← 1 to LENGTH(sequence) do
      sum ← sum + sequence[i]
   return sum
```

Im ersten Schritt der Analyse abstrahieren wir über der Eingabe, um Parameter zu finden, die die Größe der Eingabe charakterisieren. In diesem Beispiel kann jede Eingabe durch die Länge der Folge charakterisiert werden, die wir als n bezeichnen wollen. Im zweiten Schritt wird über die Implementierung abstrahiert, um ein Maß zu finden, das die Laufzeit des Algorithmus reflektiert, aber nicht an einen bestimmten Compiler oder Computer gebunden ist. Für das Programm SUMMATION könnte dies einfach die Anzahl der ausgeführten Codezeilen sein oder detaillierter, ein Maß für die Anzahl der Additionen, Zuweisungen, Arrayzugriffe und Verzweigungen, die durch den Algorithmus ausgeführt werden.

In jedem Fall erhalten wir eine Charakterisierung der Gesamtzahl der Schritte, die der Algorithmus ausgeführt hat, als Funktion der Größe der Eingabe. Wir bezeichnen diese Charakterisierung als $T(n)$. Wenn wir Codezeilen zählen, haben wir unser Beispiel $T(n) = 2n + 2$.

Wenn alle Programme so einfach wie SUMMATION wären, so wäre die Analyse der Algorithmen trivial. Zwei Probleme machen sie jedoch kompliziert. Erstens findet man selten einen Parameter wie n, der die Anzahl der Schritte, die ein Algorithmus ausführen muss, vollständig charakterisiert. Stattdessen können wir normalerweise höchstens den schlimmsten Fall $T_{worst}(n)$ oder den Mittelwert $T_{avg}(n)$ berechnen. Die Berechnung eines Mittelwertes bedeutet, dass der Analytiker eine gewisse Verteilung der Eingaben voraussetzen muss.

Das zweite Problem ist, dass Algorithmen häufig eine genauere Analyse verweigern. In diesem Fall müssen wir uns auf die Annäherung verlassen. Wir sagen, der Algorithmus SUMMATION ist $O(n)$, d.h., sein Maß ist höchstens eine Konstante mal n, mit der möglichen Ausnahme von ein paar kleinen Werten von n. Formaler ausgedrückt:

$$T(n) \text{ ist } O(f(n)), \text{ wenn } T(n) \leq kf(n) \text{ für einige } k, \text{ für alle } n > n_0 .$$

Die Notation $O(n)$ bietet uns eine sogenannte **asymptotische Analyse**. Wir können zweifellos sagen, dass, wenn n asymptotisch gegen unendlich geht, ein $O(n)$-Algorithmus besser als ein $O(n^2)$-Algorithmus ist. Eine einzige Benchmark-Auswertung kann eine solche Behauptung nicht unterlegen.

Die $O(n)$-Notation abstrahiert über konstante Faktoren, wodurch sie einfacher zu nutzen ist, aber weniger genau als die $T()$-Notation. Beispielsweise ist ein $O(n^2)$-Algorithmus auf lange Sicht immer schlechter als $O(n)$, aber wenn zwei Algorithmen $T(n^2 + 1)$ und $T(100n + 1000)$ haben, dann ist der $O(n^2)$-Algorithmus besser für $n < 110$.

Trotz dieses Nachteils ist die asymptotische Analyse das gebräuchlichste Werkzeug für die Analyse von Algorithmen. Sie ist genau, weil die Analyse sowohl über die genaue Anzahl der Operationen (durch Ignorieren des konstanten Faktors k) als auch über den genauen Inhalt der Eingabe (unter Berücksichtigung nur seiner Größe n) abstrahiert, so dass die Analyse mathematisch machbar wird. Die $O()$-Notation ist ein guter Kompromiss zwischen Genauigkeit und Einfachheit der Analyse.

A.1.2 NP und inhärent harte Probleme

Die Analyse von Algorithmen und die $O()$-Notation erlauben es uns, über die Effizienz eines bestimmten Algorithmus zu sprechen. Sie sagen jedoch nichts darüber aus, ob es einen besseren Algorithmus für das vorliegende Problem geben könnte. Der Bereich der **Komplexitätsanalyse** analysiert Probleme und nicht Algorithmen. Die erste und große Unterscheidung haben wir zwischen Problemen, die in polynomieller Zeit gelöst werden können, und Problemen, die nicht in polynomieller Zeit gelöst werden können, egal, welcher Algorithmus verwendet wird. Die Klasse der polynomiellen Probleme – die in der Zeit $O(n^k)$ für k gelöst werden können – wird als P bezeichnet. Sie werden manchmal als „einfache“ Probleme bezeichnet, weil die Klasse die Probleme mit Laufzeiten wie $O(\log n)$ und $O(n)$ enthält. Sie enthält aber auch die Probleme mit der Zeit $O(n^{1000})$; der Name „einfach“ sollte also nicht zu wörtlich genommen werden.

Eine weitere wichtige Problemklasse ist NP, die Klasse der nichtdeterministisch polynomiellen Probleme. Ein Problem wird in diese Klasse eingeordnet, wenn es einen Algorithmus gibt, der eine Lösung erraten kann und dann in polynomieller Zeit überprüft, ob die geratene Lösung korrekt ist. Die Idee dabei ist folgende: Wenn Sie eine beliebig große Anzahl an Prozessoren haben, sodass Sie versuchen können, alle Lösungen gleichzeitig zu raten, oder wenn Sie wirklich Glück haben und immer beim ersten Mal die richtige Lösung raten, werden NP-Probleme zu P-Problemen. Eine der wichtigsten offenen Fragen in der Informatik ist, ob die Klasse NP äquivalent mit der Klasse P ist, wenn man nicht den Luxus einer unendlich großen Anzahl an Prozessoren oder allwissende Rateversuche zur Verfügung hat. Die meisten Informatiker sind davon überzeugt, dass P ≠ NP – also dass NP-Probleme inhärent hart sind und es für sie keine Algorithmen mit polynomieller Zeit gibt. Dies wurde jedoch nie bewiesen.

Diejenigen, die daran interessiert sind, zu entscheiden, ob P = NP, betrachten eine Unterklasse von NP, die sogenannten **NP-vollständigen** Probleme. Das Wort „vollständig“ wird hier im Sinn von „extremst“ benutzt und bezieht sich auf die schwierigsten Probleme in der Klasse NP. Es wurde gezeigt, dass entweder alle NP-vollständigen Probleme in P liegen oder keines. Das macht die Klasse theoretisch interessant, aber sie hat auch praktisches Interesse, weil viele wichtige Probleme als NP-vollständig bekannt sind. Ein Beispiel dafür ist das Erfüllbarkeitsproblem: Gibt es für einen Satz der Aussagenlogik eine Zuweisung von Wahrheitswerten für die aussagelogischen Symbole des Satzes, die ihn wahr machen? Wenn nicht ein Wunder geschieht und P = NP gezeigt werden kann, dann kann es keinen Algorithmus geben, der *alle* Erfüllbarkeitsprobleme in polynomieller Zeit löst. Die KI ist jedoch mehr daran interessiert, ob es Algorithmen gibt, die eine effiziente Leistung für *typische* Probleme erbringen, die aus einer bestimmten Verteilung gezogen werden; wie wir in *Kapitel 7* gesehen haben, gibt es Algorithmen wie WALKSAT, die für viele Probleme eine relativ gute Leistung erbringen.

Die Klasse **co-NP** ist das Komplement zu NP, nämlich dass es für jedes Entscheidungsproblem in NP ein entsprechendes Problem in co-NP gibt, wobei die Antworten „Ja“ und „Nein“ umgekehrt werden. Wir wissen, dass P eine Untermenge sowohl von NP als auch von co-NP ist, und man nimmt an, dass es Probleme in co-NP gibt, die nicht in P liegen. Probleme mit **co-NP-Vollständigkeit** sind die schwierigsten Probleme in co-NP.

Die Klasse #P („Sharp-P“) ist die Menge der Zählprobleme, die den Entscheidungsproblemen in NP entsprechen. Entscheidungsprobleme haben eine Ja/Nein-Antwort: Gibt es eine Lösung für diese 3-SAT-Formel? In einigen Fällen ist daher ein Zählproblem schwieriger als ein Entscheidungsproblem. Beispielsweise kann die Entscheidung, ob ein bipartiter Graph eine perfekte Übereinstimmung hat, in der Zeit $O(V\,E)$ erledigt werden (wobei der Graph V Knoten und E Kanten hat), aber das Zählproblem „Wie viele perfekte Übereinstimmungen hat dieser bipartite Graph“ ist #P-vollständig, d.h., es ist so schwierig wie jedes andere Problem in #P und damit mindestens so schwierig wie jedes NP-Problem.

Als weitere Klasse werden die PSPACE-Probleme betrachtet – für die selbst auf einer nichtdeterministischen Maschine polynomieller Speicherplatz erforderlich ist. Man nimmt an, dass PSPACE-harte Probleme schlimmer als NP-vollständige Probleme sind. Es könnte sich jedoch genauso herausstellen, dass NP = PSPACE, oder auch, dass P = NP.

A.2 Vektoren, Matrizen und lineare Algebra

Mathematiker definieren einen **Vektor** als Element eines Vektorraumes, aber wir verwenden eine konkrete Definition: Ein Vektor ist eine geordnete Folge von Werten. Im zweidimensionalen Raum beispielsweise haben wir Vektoren der Art $\mathbf{x} = \langle 3, 4\rangle$ oder $\mathbf{y} = \langle 0, 2\rangle$. Wir folgen der üblichen Konvention, fett ausgezeichnete Symbole für Vektornamen zu verwenden, obwohl einige Autoren Pfeile oder Balken über den Namen verwenden: $\vec{\mathrm{x}}$ oder $\overline{\mathrm{y}}$. Die Elemente eines Vektors können mithilfe von Indizes angesprochen werden: $\mathbf{z} = \langle z_1, z_2, ..., z_n\rangle$. Ein Punkt ist verwirrend: Dieses Buch bringt die Arbeiten aus vielen Teilgebieten zusammen, die ihre Sequenzen jeweils als Vektoren, Listen oder Tupeln bezeichnen und verschiedentlich die Notationen $\langle 1, 2\rangle$, $[1, 2]$ oder $(1, 2)$ verwenden.

Die beiden grundlegenden Operationen für Vektoren sind Vektoraddition und skalare Multiplikation. Die Vektoraddition $\mathbf{x} + \mathbf{y}$ ist die elementweise Summenbildung: $\mathbf{x} + \mathbf{y} =$

$\langle 3 + 0; 4 + 2\rangle = \langle 3, 6\rangle$. Die skalare Multiplikation multipliziert jedes Element mit einer Konstanten: $5\mathbf{x} = \langle 5 \times 3; 5 \times 4\rangle = \langle 15, 20\rangle$.

Die Länge eines Vektors wird mit $|\mathbf{x}|$ angegeben. Um die Länge zu berechnen, ist die Quadratwurzel aus der Summe der Quadrate der Elemente zu bilden:

$$|\mathbf{x}| = \sqrt{(3^2 + 4^2)} = 5.$$

Das Punktprodukt (auch als Skalarprodukt bezeichnet) von zwei Vektoren $\mathbf{x} \cdot \mathbf{y}$ ist die Summe der Produkte der entsprechenden Elemente:

$$\mathbf{x} \cdot \mathbf{y} = \sum_i x_i y_i,$$

in unserem Beispiel also $\mathbf{x} \cdot \mathbf{y} = 3 \times 0 + 4 \times 2 = 8$.

Vektoren werden häufig als gerichtete Liniensegmente (Pfeile) in einem n-dimensionalen euklidischen Raum interpretiert. Die Vektoraddition ist dann damit äquivalent, das Ende eines Vektors ist an den Anfang des anderen einzufügen und das Punktprodukt $\mathbf{x} \cdot \mathbf{y}$ ist gleich $|\mathbf{x}||\mathbf{y}| \cos\theta$, wobei θ der Winkel zwischen $\mathbf{x}$ und $\mathbf{y}$ ist.

Eine **Matrix** ist ein rechteckiges Feld mit Werten, die in Zeilen und Spalten angeordnet sind. Nachfolgend sehen Sie eine Matrix $\mathbf{A}$ der Größe 3×4:

$$\begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} & \mathbf{A}_{1,3} & \mathbf{A}_{1,4} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} & \mathbf{A}_{2,3} & \mathbf{A}_{2,4} \\ \mathbf{A}_{3,1} & \mathbf{A}_{3,2} & \mathbf{A}_{3,3} & \mathbf{A}_{3,4} \end{pmatrix}.$$

Der erste Index von $\mathbf{A}_{i,j}$ gibt die Zeile an, der zweite die Spalte. In Programmiersprachen wird $\mathbf{A}_{i,j}$ häufig als `A[i, j]` oder `A[i][j]` dargestellt.

Die Summe zweier Matrizen ist definiert durch die Addition einander entsprechender Elemente: $(\mathbf{A} + \mathbf{B})_{i,j} = \mathbf{A}_{i,j} + \mathbf{B}_{i,j}$. (Die Summe ist nicht definiert, wenn $\mathbf{A}$ und $\mathbf{B}$ unterschiedliche Größen haben.) Wir können auch die Multiplikation einer Matrix mit einem Skalar definieren: $(c\mathbf{A})_{i,j} = c\mathbf{A}_{i,j}$. Die Matrixmultiplikation (das Produkt von zwei Matrizen) ist komplizierter. Das Produkt $\mathbf{AB}$ ist nur dann definiert, wenn $\mathbf{A}$ die Größe $a \times b$ und $\mathbf{B}$ die Größe $b \times c$ hat (d.h. die Anzahl der Zeilen in der zweiten Matrix gleich der Anzahl der Spalten in der ersten Matrix ist); das Ergebnis ist eine Matrix der Größe $a \times c$. Wenn die Matrizen eine geeignete Größe haben, lautet das Ergebnis:

$$(\mathbf{AB})_{i,k} = \sum_j \mathbf{A}_{i,j} \mathbf{B}_{j,k}.$$

Die Matrixmultiplikation ist selbst bei quadratischen Matrizen nicht kommutativ: Im Allgemeinen gilt $\mathbf{AB} \neq \mathbf{BA}$. Allerdings ist die Matrixmultiplikation assoziativ: $(\mathbf{AB})\mathbf{C} = \mathbf{A}(\mathbf{BC})$. Das Punktprodukt lässt sich in Form einer Transposition und einer Matrixmultiplikation ausdrücken: $\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^\top \mathbf{y}$.

In einer **Identitätsmatrix** $\mathbf{I}$ sind die Elemente $\mathbf{I}_{i,j}$ gleich 1, wenn $i = j$ und andernfalls 0. Sie hat die Eigenschaft, dass $\mathbf{AI} = \mathbf{A}$ für alle $\mathbf{A}$. Die **transponierte Matrix** von $\mathbf{A}$, dargestellt als $\mathbf{A}^\top$, wird gebildet, indem die Zeilen in Spalten umgewandelt werden und umgekehrt, formaler ausgedrückt durch $\mathbf{A}^\top{}_{i,j} = \mathbf{A}_{j,i}$. Die **Inverse** einer quadratischen Matrix $\mathbf{A}$ ist eine andere quadratische Matrix $\mathbf{A}^{-1}$, sodass gilt $\mathbf{A}^{-1}\mathbf{A} = 1$. Für eine **singuläre Matrix** existiert keine Inverse. Für eine nichtsinguläre Matrix lässt sie sich in einer Zeit von $O(n^3)$ berechnen.

Matrizen werden verwendet, um lineare Gleichungssysteme in $O(n^3)$ Zeit zu lösen; die Zeit wird durch das Invertieren einer Koeffizientenmatrix dominiert. Betrachten Sie die folgende Gleichungsmenge, für die wir eine Lösung in x, y und z suchen:

$$\begin{aligned} +2x + y - z &= 8 \\ -3x - y + 2z &= -11 \\ -2x + y + 2z &= -3 \end{aligned}$$

Wir können dieses System als Matrixgleichung $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ darstellen, wobei

$$\mathbf{A} = \begin{pmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} 8 \\ -11 \\ -3 \end{pmatrix}$$

Um $\mathbf{A}\,\mathbf{x} = \mathbf{b}$ zu lösen, multiplizieren wir beide Seiten mit $\mathbf{A}^{-1}$, was $\mathbf{A}^{-1}\mathbf{A}\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ ergibt und sich zu $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ vereinfachen lässt. Nachdem wir $\mathbf{A}$ invertiert und mit $\mathbf{b}$ multipliziert haben, erhalten wir die Antwort:

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \\ -1 \end{pmatrix}.$$

A.3 Wahrscheinlichkeitsverteilungen

Eine Wahrscheinlichkeit ist ein Maß für eine Menge von Ereignissen, die drei Axiome erfüllt:

1 Das Maß jedes Ereignisses liegt zwischen 0 und 1. Wir schreiben dafür $0 \leq P(X = x_i) \leq 1$, wobei X eine Zufallsvariable ist, die ein Ereignis darstellt; x_i sind die möglichen Werte von X. Im Allgemeinen werden Zufallsvariablen durch Großbuchstaben und ihre Werte durch Kleinbuchstaben dargestellt.

2 Das Maß der gesamten Menge ist 1; d.h.

$$\sum_{i=1}^{n} P(E = e_i) = 1 \cdot$$

3 Die Wahrscheinlichkeit einer Vereinigung disjunkter Ereignisse ist die Summe der Wahrscheinlichkeiten der einzelnen Ereignisse; d.h. $P(X = x_1 \vee X = x_2) = P(X = x_1) + P(X = x_2)$, wobei x_1 und x_2 disjunkt sind.

Ein **probabilistisches Modell** besteht aus einem Stichprobenraum sich gegenseitig ausschließender möglicher Ergebnisse, zusammen mit einem Wahrscheinlichkeitsmaß für jedes Ergebnis. In einem Modell beispielsweise für das morgige Wetter könnten die Ergebnisse *sonnig, wolkig, regnerisch* und *Schneefall* sein. Eine Untermenge dieser Ergebnisse besteht aus einem Ereignis. Beispielsweise ist das Ereignis der Wettervorhersage die Untermenge {*regnerisch, Schneefall*}.

Wir verwenden $\mathbf{P}(X)$, um den Vektor der Werte $\langle P(X = x_1), \ldots, P(X = x_n)\rangle$ darzustellen. Außerdem verwenden wir $P(x_i)$ als Abkürzung für $P(X = x_i)$ und $\Sigma_x P(x)$ für $\Sigma_{i=1}{}^{n} P(X = x_i)$.

Die bedingte Wahrscheinlichkeit $P(B \mid A)$ ist definiert als $P(B \cap A)/P(A)$. A und B sind bedingt unabhängig, wenn $P(B \mid A) = P(B)$ (oder äquivalent $P(A \mid B) = P(A)$). Für stetige

Variablen gibt es eine unendliche Anzahl an Werten und wenn es keine Punktspitzen gibt, ist die Wahrscheinlichkeit jedes Wertes gleich 0. Aus diesem Grund definieren wir eine **Wahrscheinlichkeitsdichtefunktion**, die wir auch als $P(\cdot)$ bezeichnen, die jedoch eine etwas andere Bedeutung als die diskrete Wahrscheinlichkeitsfunktion hat. Die Dichtefunktionen $P(x)$ für eine Zufallsvariable X, die man sich als $P(X = x)$ vorstellen kann, ist intuitiv definiert als das Verhältnis der Wahrscheinlichkeit, dass X in einem bestimmten Intervall um x liegt, und der Breite des Intervalls, wenn die Intervallbreite gegen null geht:

$$P(x) = \lim_{dx \to 0} P(x \leq X \leq x + dx) / dx.$$

Die Dichtefunktion muss für alle x nichtnegativ sein und es muss gelten:

$$\int_{-\infty}^{\infty} P(x)dx = 1.$$

Wir können auch eine **kumulative Wahrscheinlichkeitsdichtefunktion** $F_X(x)$ definieren, die die Wahrscheinlichkeit angibt, dass eine Zufallsvariable kleiner als x ist:

$$F_X(x) = P(X \leq x) = \int_{-\infty}^{x} P(u)du.$$

Beachten Sie, dass die Wahrscheinlichkeitsdichtefunktion Einheiten hat, während die diskrete Wahrscheinlichkeitsfunktion einheitenlos ist. Wenn zum Beispiel die Werte von X in Sekunden gemessen werden, verwendet man für die Dichte die Einheit Hz (das heißt 1/s). Sind die Werte von $\mathbf{X}$ Punkte im dreidimensionalen Raum, der in Meter gemessen wird, dann wird die Dichte in $1/m^3$ gemessen.

Eine der wichtigsten Wahrscheinlichkeitsverteilungen ist die **Gaußsche Verteilung**, auch als **Normalverteilung** bezeichnet. Eine Gaußverteilung mit dem Mittelwert μ und der Standardabweichung σ (und damit der Varianz σ^2) ist definiert als

$$P(x) = \frac{1}{\sqrt{(2\pi)}} e^{-(x-\mu)^2/(2\sigma^2)}.$$

Dabei ist x eine stetige Variable zwischen $-\infty$ und $+\infty$. Mit einem Mittelwert $\mu = 0$ und einer Varianz $\sigma^2 = 1$ erhalten wir den Sonderfall der **Standardnormalverteilung**. Für eine Verteilung über einem Vektor $\mathbf{x}$ in n Dimensionen gibt es die **multivariate Gaußverteilung**:

$$P(\mathbf{x}) = \frac{1}{\sqrt{(2\pi)^n \, |\Sigma|}} e^{-\frac{1}{2}((\mathbf{x}-\mu)^T \Sigma^{-1} (\mathbf{x}-\mu))}.$$

Dabei ist $\boldsymbol{\mu}$ der Mittelwertvektor und Σ ist die **Kovarianzmatrix** (siehe unten).

In einer Dimension können wir auch die **kumulative Verteilungsfunktion** $F(x)$ als die Wahrscheinlichkeit, dass eine Zufallsvariable kleiner als x ist, definieren. Für die Standardnormalverteilung ist dies

$$F(x) = \int_{-\infty}^{x} P(z)dz = \frac{1}{2}(1 + \text{erf}(\frac{z-\mu}{\sigma\sqrt{2}})).$$

Dabei ist $\text{erf}(x)$ die sogenannte **Fehlerfunktion**, für die es keine geschlossene Darstellung gibt.

Der **zentrale Grenzwertsatz** besagt, dass die Verteilung, die sich durch Mittelwertbildung aus einer Stichprobe von n unabhängigen Zufallsvariablen ergibt, zu einer Normalverteilung wird, wenn n gegen unendlich geht. Das gilt für fast jede Sammlung von Zufallsvariablen, selbst wenn sie nicht streng unabhängig sind, es sei denn, die Varianz einer endlichen Untermenge von Variablen dominiert die anderen.

Die **Erwartung** einer Zufallsvariablen $E(X)$ ist der Mittel- oder Durchschnittswert, gewichtet nach der Wahrscheinlichkeit jedes Wertes. Für eine diskrete Variable ist er:

$$E(X) = \sum_i x_i P(X = x_i).$$

Für eine stetige Variable wird die Summation durch ein Integral über der Wahrscheinlichkeitsdichtefunktion $P(x)$ ersetzt:

$$E(X) = \int_{-\infty}^{\infty} xP(x)dx.$$

Der **quadratische Mittelwert** oder **Effektivwert** (engl. RMS – Root Mean Square) einer Menge von Werten (oftmals Stichproben einer Zufallsvariablen) ist die Quadratwurzel aus dem Mittelwert der quadrierten Werte:

$$RMS(x_1, \ldots, x_n) = \sqrt{\frac{x_1^2 + \ldots + x_n^2}{n}}.$$

Die **Kovarianz** von zwei Zufallsvariablen ist der Erwartungswert des Produktes ihrer Differenzen zu ihren Mittelwerten:

$$\text{cov}(X, Y) = E((X - \mu_X)(Y - \mu_Y)).$$

Die oft mit Σ bezeichnete **Konvarianzmatrix** ist eine Matrix von Kovarianzen zwischen Elementen eines Vektors von Zufallsvariablen. Für $\mathbf{X} = [X_1, \ldots, X_n]^\top$ lauten die Einträge der Kovarianzmatrix wie folgt:

$$\Sigma_{i,j} = \text{cov}(X_i, X_j) = E((X_i - \mu_i)(X_j - \mu_j)).$$

Zum Schluss noch einige Anmerkungen: Wir verwenden $\log(x)$ für den natürlichen Logarithmus $\log_e(x)$. Mit $\text{argmax}_x f(x)$ bezeichnen wir den Wert von x, für den $f(x)$ maximal ist.

Bibliografische und historische Hinweise

Die O()-Notation, die in der Informatik heute so häufig genutzt wird, wurde zuerst im Kontext der Zahlentheorie von dem deutschen Mathematiker P.G.H. Bachmann (1894) eingeführt. Das Konzept der NP-Vollständigkeit wurde von Cook (1971) erfunden und die moderne Methode für die Einrichtung einer Reduzierung von einem Problem auf ein anderes geht auf Karp (1972) zurück. Cook und Karp haben beide für ihre Arbeit den Turing-Preis gewonnen, die höchste Ehre in der Informatik.

Klassische Arbeiten zur Analyse und zum Entwurf von Algorithmen stammen unter anderem von Knuth (1973) und Aho, Hopcroft und Ullman (1974), neuere Beiträge von Tarjan (1983) und Cormen, Leiserson und Rivest (1990). Diese Bücher konzentrieren sich hauptsächlich auf den Entwurf und die Analyse von Algorithmen, um handhabbare Probleme zu lösen. Für die Theorie der NP-Vollständigkeit und andere Formen der Nichthandhabbarkeit lesen Sie bei Garey und Johnson (1979) oder Papadimitriou (1994) nach. Gute Bücher zur Wahrscheinlichkeit stammen unter anderem von Chung (1979) und Ross (1988) sowie von Bertsekas und Tsitsiklis.

Hinweise zu Sprachen und Algorithmen

B

ÜBERBLICK

B.1 Sprachen mit Backus-Naur-Form (BNF) definieren

In diesem Buch definieren wir mehrere Sprachen, einschließlich der Sprachen der Aussagenlogik (*Abschnitt 7.4*), die Logik erster Stufe (*Abschnitt 8.2.2*) sowie eine Untermenge von Englisch (*Abschnitt 22.3.2*). Eine formale Sprache ist definiert als eine Menge von Zeichenketten, wobei jede Zeichenkette eine Folge von Symbolen darstellt. Da die uns interessierenden Sprachen aus einer unendlichen Menge von Zeichenketten bestehen, brauchen wir eine präzise Möglichkeit, die Menge zu charakterisieren. Dazu verwenden wir eine **Grammatik** – und zwar eine sogenannte **kontextfreie Grammatik**, weil jeder Ausdruck in jedem Kontext die gleiche Form hat. Wir schreiben unsere Grammatiken in einem Formalismus, der als **Backus-Naur-Form** (BNF) bezeichnet wird. Eine BNF-Grammatik besteht aus vier Komponenten:

- Einer Menge von **Terminalsymbolen**. Das sind die Symbole oder Wörter, aus denen sich eine Zeichenkette der Sprache zusammensetzt. Es könnte sich dabei um Buchstaben (**A**, **B**, **C**, ...) oder Wörter (**a**, **aardvark**, **abacus**, ...) handeln.
- Einer Menge von **Nichtterminalsymbolen**, die die Subphrasen der Sprache kategorisieren. Beispielsweise bezeichnet das Nichtterminalsymbol *SubstantivPhrase* eine unendliche Menge von Zeichenketten wie etwa „**du**“ oder „**der dicke alte Waldbär**“.
- Einem **Startsymbol**, das als Nichtterminalsymbol die gesamten Zeichenketten der Sprache kennzeichnet. In einer natürlichen Sprache könnte dies *Satz* sein, in der Arithmetik wäre es *Ausdruck* und für Programmiersprachen verwendet man *Programm*.
- Eine Menge von **Umschreibungsregeln** der Form *LinkeSeite* → *RechteSeite*, wobei *LinkeSeite* ein Nichtterminalsymbol und *RechteSeite* eine Folge von null oder mehreren Symbolen ist. Dies können sowohl Terminal- als auch Nichtterminalsymbole sein oder das Symbol $\in$, das für die leere Zeichenkette steht.

Eine Umschreibungsregel der Form

$$Satz \rightarrow SubstantivPhrase\ VerbPhrase$$

bedeutet: Wenn wir zwei Zeichenketten haben, die als *SubstantivPhrase* und *VerbPhrase* eingeordnet sind, können wir sie verknüpfen und das Ergebnis als *Satz* kategorisieren. Als Abkürzung lassen sich die beiden Regeln ($S \rightarrow A$) und ($S \rightarrow B$) als ($S \rightarrow A \mid B$) schreiben.

Für einfache arithmetische Ausdrücke sieht eine BNF-Grammatik wie folgt aus:

$$\begin{array}{lcl} Ausdruck & \rightarrow & Ausdruck\ Operator\ Ausdruck \mid (\ Ausdruck\ ) \mid Zahl \\ Zahl & \rightarrow & Ziffer \mid Zahl\ Ziffer \\ Ziffer & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ Operator & \rightarrow & + \mid - \mid \div \mid \times . \end{array}$$

In *Kapitel 22* haben wir uns genauer mit Sprachen und Grammatiken beschäftigt. Beachten Sie, dass andere Bücher eine etwas andere Notation für BNF verwenden können; beispielsweise sind ⟨*Ziffer*⟩ statt *Ziffer* für ein Nichtterminalzeichen gebräuchlich oder 'wort' statt **wort** für ein Terminalzeichen oder ::= statt → in einer Regel.

B.2 Algorithmen mit Pseudocode beschreiben

Die Algorithmen in diesem Buch sind in Pseudocode beschrieben. Ein Großteil dieses Pseudocodes sollte den Benutzern von Sprachen wie Java, C++ oder Lisp vertraut sein. Manchmal verwenden wir mathematische Formeln oder die natürliche Sprache, um Teile zu beschreiben, die andernfalls zu umständlich wären. Beachten Sie einige Konventionen:

- **Persistente Variablen:** Mit dem Schlüsselwort **persistent** bezeichnen wir eine Variable, die beim ersten Aufruf einer Funktion einen Ausgangswert erhält und diesen Wert (oder den Wert, den sie durch eine nachfolgende Zuweisung erhält) für alle nachfolgenden Aufrufe der Funktion beibehält. Somit sind persistente Variablen mit globalen Variablen vergleichbar, weil sie einen einzelnen Aufruf ihrer Funktion überdauern, jedoch sind sie nur innerhalb der Funktion zugänglich. Die Agentenprogramme im Buch verwenden persistente Variablen als „Speicher". Programme mit persistenten Variablen können als „Objekte" in objektorientierten Sprachen wie beispielsweise C++, Java, Python oder Smalltalk implementiert werden. In funktionalen Sprachen können sie durch *Funktionsabschlüsse* (*Closures*) über einer Umgebung, die die erforderlichen Variablen enthält, implementiert werden.
- **Funktionen als Werte:** Die Namen von Funktionen und Prozeduren sind groß (in sogenannten „Kapitälchen"), Variablen klein und kursiv geschrieben. Ein Funktionsaufruf sieht also meistens wie FN(x) aus. Wir erlauben jedoch, dass der Wert einer Variablen eine Funktion ist; wenn der Wert der Variablen f beispielsweise die Quadratwurzelfunktion ist, dann gibt $f(9)$ den Wert 3 zurück.
- **for each:** Die Notation „`for each` *x* `in` *c* `do`" bedeutet, dass die Schleife ausgeführt wird, wobei die Variable x an aufeinanderfolgende Elemente der Auflistung c gebunden ist.
- **Einrückungen sind wichtig:** Einrückungen kennzeichnen den Gültigkeitsbereich einer Schleife oder einer Bedingung, so wie in der Sprache Python und anders als in Java und C++ (die Klammern verwenden) oder Pascal und Visual Basic (die das Schlüsselwort **end** verwenden).
- **Mehrfachzuweisung:** Die Notation „$x, y \leftarrow$ *Paar*" bedeutet, dass die rechte Seite zu einem zweielementigen Tupel aufzulösen ist, wobei das erste Element an x und das zweite an y zugewiesen wird. Das gleiche Konzept trifft auf „`for each` *x*, *y* `in` *Paaren* `do`" zu und lässt sich nutzen, um die Inhalte zweier Variablen zu vertauschen: „$x, y \leftarrow y, x$".
- **Generatoren** und **yield:** Die Notation „`generator` *G*(*x*) `yields` Zahlen" definiert G als Generatorfunktion. Dies lässt sich am besten an einem Beispiel verdeutlichen. Das in ▶ Abbildung B.1 gezeigte Codefragment gibt die Zahlen 1, 2, 4, ..., aus und hält niemals an. Der Aufruf von POWERS-OF-2 gibt einen Generator zurück, der seinerseits jedes Mal einen Wert liefert, wenn der Schleifencode nach dem nächsten Element einer Auflistung fragt. Selbst wenn die Auflistung unendlich ist, wird bei jedem Aufruf jeweils nur ein Element aufgezählt.

```
generator POWERS-OF-2() yields Ganzzahlen
  i ← 1
  while true do
    yield i
    i ← 2 × im
```

```
for p in POWERS-OF-2() do
  PRINT(p)
```

Abbildung B.1: Beispiel für eine Generatorfunktion und ihren Aufruf innerhalb einer Schleife.

- **Listen:** [*x*, *y*, *z*] kennzeichnet eine Liste mit drei Elementen. Um eine mit [*erstes* | *rest*] bezeichnete Liste zu bilden, wird *first* zur Liste *rest* hinzugefügt. In Lisp ist dies die Funktion `cons`.
- **Mengen:** {*x*, *y*, *z*} kennzeichnet eine Menge von drei Elementen. Der Ausdruck {*x* : *p*(*x*)} steht für die Menge aller Elemente *x*, für die *p*(*x*) wahr ist.
- **Arrays beginnen bei 1:** Wenn nicht anders angegeben, ist der erste Index eines Arrays 1, wie in der Mathematik üblich, und nicht 0, wie in Java oder C.

B.3 Online-Hilfe

Die Implementierungen der meisten Algorithmen im Buch sind in unserem Online-Codeverzeichnis zu finden:

Tipp

aima.cs.berkeley.edu

Auf derselben Website finden Sie auch Anleitungen, um Kommentare, Korrekturen oder Vorschläge, die das Buch verbessern könnten, einzusenden, sowie Hinweise, um an Diskussionslisten teilzunehmen.

Bibliografie

Die folgenden Abkürzungen werden für häufig zititerte Konferenzen und Journale verwendet:

AAAI	Proceedings of the AAAI Conference on Artificial Intelligence
AAMAS	Proceedings of the International Conference on Autonomous Agents and Multi-agent Systems
ACL	Proceedings of the Annual Meeting of the Association for Computational Linguistics
AIJ	Artificial Intelligence
AIMag	AI Magazine
AIPS	Proceedings of the International Conference on AI Planning Systems
BBS	Behavioral and Brain Sciences
CACM	Communications of the Association for Computing Machinery
COGSCI	Proceedings of the Annual Conference of the Cognitive Science Society
COLING	Proceedings of the International Conference on Computational Linguistics
COLT	Proceedings of the Annual ACM Workshop on Computational Learning Theory
CP	Proceedings of the International Conference on Principles and Practice of Constraint Programming
CVPR	Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition
EC	Proceedings of the ACM Conference on Electronic Commerce
ECAI	Proceedings of the European Conference on Artificial Intelligence
ECCV	Proceedings of the European Conference on Computer Vision
ECML	Proceedings of the The European Conference on Machine Learning
ECP	Proceedings of the European Conference on Planning
FGCS	Proceedings of the International Conference on Fifth Generation Computer Systems
FOCS	Proceedings of the Annual Symposium on Foundations of Computer Science
ICAPS	Proceedings of the International Conference on Automated Planning and Scheduling
ICASSP	Proceedings of the International Conference on Acoustics, Speech, and Signal Processing
ICCV	Proceedings of the International Conference on Computer Vision
ICLP	Proceedings of the International Conference on Logic Programming
ICML	Proceedings of the International Conference on Machine Learning
ICPR	Proceedings of the International Conference on Pattern Recognition
ICRA	Proceedings of the IEEE International Conference on Robotics and Automation
ICSLP	Proceedings of the International Conference on Speech and Language Processing
IJAR	International Journal of Approximate Reasoning
IJCAI	Proceedings of the International Joint Conference on Artificial Intelligence
IJCNN	Proceedings of the International Joint Conference on Neural Networks
IJCV	International Journal of Computer Vision
ILP	Proceedings of the International Workshop on Inductive Logic Programming
ISMIS	Proceedings of the International Symposium on Methodologies for Intelligent Systems
ISRR	Proceedings of the International Symposium on Robotics Research
JACM	Journal of the Association for Computing Machinery
JAIR	Journal of Artificial Intelligence Research
JAR	Journal of Automated Reasoning
JASA	Journal of the American Statistical Association
JMLR	Journal of Machine Learning Research
JSL	Journal of Symbolic Logic
KDD	Proceedings of the International Conference on Knowledge Discovery and Data Mining
KR	Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning
LICS	Proceedings of the IEEE Symposium on Logic in Computer Science
NIPS	Advances in Neural Information Processing Systems
PAMI	IEEE Transactions on Pattern Analysis and Machine Intelligence
PNAS	Proceedings of the National Academy of Sciences of the United States of America
PODS	Proceedings of the ACM International Symposium on Principles of Database Systems
SIGIR	Proceedings of the Special Interest Group on Information Retrieval
SIGMOD	Proceedings of the ACM SIGMOD International Conference on Management of Data
SODA	Proceedings of the Annual ACM–SIAM Symposium on Discrete Algorithms
STOC	Proceedings of the Annual ACM Symposium on Theory of Computing
TARK	Proceedings of the Conference on Theoretical Aspects of Reasoning about Knowledge
UAI	Proceedings of the Conference on Uncertainty in Artificial Intelligence

Aarup, M., Arentoft, M. M., Parrod, Y., Stader, J. und Stokes, I. (1994). OPTIMUM-AIV: A knowledge-based planning and scheduling system for spacecraft AIV. In Fox, M. und Zweben, M. (Hrsg.), *Knowledge Based Scheduling*. Morgan Kaufmann.

Abney, S. (2007). *Semisupervised Learning for Computational Linguistics*. CRC Press.

Abramson, B. und Yung, M. (1989). Divide and conquer under global constraints: A solution to the N-queens problem. *J. Parallel and Distributed Computing*, *6*(3), 649–662.

Achlioptas, D. (2009). Random satisfiability. In Biere, A., Heule, M., van Maaren, H. und Walsh, T. (Hrsg.), *Handbook of Satisfiability*. IOS Press.

Achlioptas, D., Beame, P. und Molloy, M. (2004). Exponential bounds for DPLL below the satisfiability threshold. In *SODA-04*.

Achlioptas, D., Naor, A. und Peres, Y. (2007). On the maximum satisfiability of random formulas. *JACM*, *54*(2).

Achlioptas, D. und Peres, Y. (2004). The threshold for random k-SAT is $2k \log 2 - o(k)$. *J. American Mathematical Society*, *17*(4), 947–973.

Ackley, D. H. und Littman, M. L. (1991). Interactions between learning and evolution. In Langton, C., Taylor, C., Farmer, J. D. und Ramussen, S. (Hrsg.), *Artificial Life II*, S. 487–509. Addison-Wesley.

Adelson-Velsky, G. M., Arlazarov, V. L., Bitman, A. R., Zhivotovsky, A. A. und Uskov, A. V. (1970). Programming a computer to play chess. *Russian Mathematical Surveys*, *25*, 221–262.

Adida, B. und Birbeck, M. (2008). RDFa primer. Tech. rep., W3C.

Agerbeck, C. und Hansen, M. O. (2008). A multiagent approach to solving *NP*-complete problems. Master's thesis, Technical Univ. of Denmark.

Aggarwal, G., Goel, A. und Motwani, R. (2006). Truthful auctions for pricing search keywords. In *EC-06*, S. 1–7.

Agichtein, E. und Gravano, L. (2003). Querying text databases for efficient information extraction. In *Proc. IEEE Conference on Data Engineering*.

Agmon, S. (1954). The relaxation method for linear inequalities. *Canadian Journal of Mathematics*, *6*(3), 382–392.

Agre, P. E. und Chapman, D. (1987). Pengi: an implementation of a theory of activity. In *IJCAI-87*, S. 268–272.

Aho, A. V., Hopcroft, J. und Ullman, J. D. (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley.

Aizerman, M., Braverman, E. und Rozonoer, L. (1964). Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, *25*, 821–837.

Al-Chang, M., Bresina, J., Charest, L., Chase, A., Hsu, J., Jonsson, A., Kanefsky, B., Morris, P., Rajan, K., Yglesias, J., Chafin, B., Dias, W. und Maldague, P. (2004). MAPGEN: Mixed-Initiative planning and scheduling for the Mars Exploration Rover mission. *IEEE Intelligent Systems*, *19*(1), 8–12.

Albus, J. S. (1975). A new approach to manipulator control: The cerebellar model articulation controller (CMAC). *J. Dynamic Systems, Measurement, and Control*, *97*, 270–277.

Aldous, D. und Vazirani, U. (1994). "Go with the winners" algorithms. In *FOCS-94*, S. 492–501.

Alekhnovich, M., Hirsch, E. A. und Itsykson, D. (2005). Exponential lower bounds for the running time of DPLL algorithms on satisfiable formulas. *JAR*, *35*(1–3), 51–72.

Allais, M. (1953). Le comportment de l'homme rationnel devant la risque: critique des postulats et axiomes de l'école Américaine. *Econometrica*, *21*, 503–546.

Allen, J. F. (1983). Maintaining knowledge about temporal intervals. *CACM*, *26*(11), 832–843.

Allen, J. F. (1984). Towards a general theory of action and time. *AIJ*, *23*, 123–154.

Allen, J. F. (1991). Time and time again: The many ways to represent time. *Int. J. Intelligent Systems*, *6*, 341–355.

Allen, J. F., Hendler, J. und Tate, A. (Hrsg.). (1990). *Readings in Planning*. Morgan Kaufmann.

Allis, L. (1988). A knowledge-based approach to connect four. The game is solved: White wins. Master's thesis, Vrije Univ., Amsterdam.

Almuallim, H. und Dietterich, T. (1991). Learning with many irrelevant features. In *AAAI-91*, Band 2, S. 547–552.

ALPAC (1966). Language and machines: Computers in translation and linguistics. Tech. rep. 1416, The Automatic Language Processing Advisory Committee of the National Academy of Sciences.

Alterman, R. (1988). Adaptive planning. *Cognitive Science*, *12*, 393–422.

Amarel, S. (1967). An approach to heuristic problem-solving and theorem proving in the propositional calculus. In Hart, J. und Takasu, S. (Hrsg.), *Systems and Computer Science*. University of Toronto Press.

Amarel, S. (1968). On representations of problems of reasoning about actions. In Michie, D. (Hrsg.), *Machine Intelligence 3*, Band 3, S. 131–171. Elsevier/North-Holland.

Amir, E. und Russell, S. J. (2003). Logical filtering. In *IJCAI-03*.

Amit, D., Gutfreund, H. und Sompolinsky, H. (1985). Spin-glass models of neural networks. *Physical Review*, *A 32*, 1007–1018.

Andersen, S. K., Olesen, K. G., Jensen, F. V. und Jensen, F. (1989). HUGIN – A shell for building Bayesian belief universes for expert systems. In *IJCAI-89*, Band 2, S. 1080–1085.

Anderson, J. R. (1980). *Cognitive Psychology and Its Implications*. W. H. Freeman.

Anderson, J. R. (1983). *The Architecture of Cognition*. Harvard University Press.

Andoni, A. und Indyk, P. (2006). Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS-06*.

Andre, D. und Russell, S. J. (2002). State abstraction for programmable reinforcement learning agents. In *AAAI-02*, S. 119–125.

Anthony, M. und Bartlett, P. (1999). *Neural Network Learning: Theoretical Foundations*. Cambridge University Press.

Aoki, M. (1965). Optimal control of partially observable Markov systems. *J. Franklin Institute*, *280*(5), 367–386.

Appel, K. und Haken, W. (1977). Every planar map is four colorable: Part I: Discharging. *Illinois J. Math.*, *21*, 429–490.

Appelt, D. (1999). Introduction to information extraction. *CACM*, *12*(3), 161–172.

Apt, K. R. (1999). The essence of constraint propagation. *Theoretical Computer Science*, *221*(1–2), 179–210.

Apt, K. R. (2003). *Principles of Constraint Programming*. Cambridge University Press.

Apté, C., Damerau, F. und Weiss, S. (1994). Automated learning of decision rules for text categorization. *ACM Transactions on Information Systems*, *12*, 233–251.

Arbuthnot, J. (1692). *Of the Laws of Chance*. Motte, London. Übersetzung ins Englische mit Ergänzungen von Huygens (1657).

Archibald, C., Altman, A. und Shoham, Y. (2009). Analysis of a winning computational billiards player. In *IJCAI-09*.

Ariely, D. (2009). *Predictably Irrational* (Überarbeitete Auflage). Harper.

Arkin, R. (1998). *Behavior-Based Robotics*. MIT Press.

Armando, A., Carbone, R., Compagna, L., Cuellar, J. und Tobarra, L. (2008). Formal analysis of SAML 2.0 web browser single sign-on: Breaking the SAML-based single sign-on for google apps. In *FMSE '08: Proc. 6th ACM workshop on Formal methods in security engineering*, S. 1–10.

Arnauld, A. (1662). *La logique, ou l'art de penser*. Chez Charles Savreux, au pied de la Tour de Nostre Dame, Paris.

Arora, S. (1998). Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems. *JACM*, *45*(5), 753–782.

Arunachalam, R. und Sadeh, N. M. (2005). The supply chain trading agent competition. *Electronic Commerce Research and Applications*, *Spring*, 66–84.

Ashby, W. R. (1940). Adaptiveness and equilibrium. *J. Mental Science*, *86*, 478–483.

Ashby, W. R. (1948). Design for a brain. *Electronic Engineering, December*, 379–383.

Ashby, W. R. (1952). *Design for a Brain*. Wiley.

Asimov, I. (1942). Runaround. *Astounding Science Fiction*, *March*.

Asimov, I. (1950). *I, Robot*. Doubleday.

Astrom, K. J. (1965). Optimal control of Markov decision processes with incomplete state estimation. *J. Math. Anal. Applic.*, *10*, 174–205.

Audi, R. (Hrsg.). (1999). *The Cambridge Dictionary of Philosophy*. Cambridge University Press.

Axelrod, R. (1985). *The Evolution of Cooperation*. Basic Books.

Baader, F., Calvanese, D., McGuinness, D., Nardi, D. und Patel-Schneider, P. (2007). *The Description Logic Handbook* (2. Auflage). Cambridge University Press.

Baader, F. und Snyder, W. (2001). Unification theory. In Robinson, J. und Voronkov, A. (Hrsg.), *Handbook of Automated Reasoning*, S. 447–533. Elsevier.

Bacchus, F. (1990). *Representing and Reasoning with Probabilistic Knowledge*. MIT Press.

Bacchus, F. und Grove, A. (1995). Graphical models for preference and utility. In *UAI-95*, S. 3–10.

Bacchus, F. und Grove, A. (1996). Utility independence in a qualitative decision theory. In *KR-96*, S. 542–552.

Bacchus, F., Grove, A., Halpern, J. Y. und Koller, D. (1992). From statistics to beliefs. In *AAAI-92*, S. 602–608.

Bacchus, F. und van Beek, P. (1998). On the conversion between non-binary and binary constraint satisfaction problems. In *AAAI-98*, S. 311–318.

Bacchus, F. und van Run, P. (1995). Dynamic variable ordering in CSPs. In *CP-95*, S. 258–275.

Bachmann, P. G. H. (1894). *Die analytische Zahlentheorie*. B. G. Teubner, Leipzig.

Backus, J. W. (1996). Transcript of question and answer session. In Wexelblat, R. L. (Hrsg.), *History of Programming Languages*, p. 162. Academic Press.

Bagnell, J. A. und Schneider, J. (2001). Autonomous helicopter control using reinforcement learning policy search methods. In *ICRA-01*.

Baker, J. (1975). The Dragon system – An overview. *IEEE Transactions on Acoustics; Speech; and Signal Processing*, *23*, 24–29.

Baker, J. (1979). Trainable grammars for speech recognition. In *Speech Communication Papers for the 97th Meeting of the Acoustical Society of America*, S. 547–550.

Baldi, P., Chauvin, Y., Hunkapiller, T. und McClure, M. (1994). Hidden Markov models of biological primary sequence information. *PNAS*, *91*(3), 1059–1063.

Baldwin, J. M. (1896). A new factor in evolution. *American Naturalist*, *30*, 441–451. Fortsetzung auf den Seiten 536–553.

Ballard, B. W. (1983). The *-minimax search procedure for trees containing chance nodes. *AIJ*, *21*(3), 327–350.

Baluja, S. (1997). Genetic algorithms and explicit search statistics. In Mozer, M. C., Jordan, M. I. und Petsche, T. (Hrsg.), *NIPS 9*, S. 319–325. MIT Press.

Bancilhon, F., Maier, D., Sagiv, Y. und Ullman, J. D. (1986). Magic sets and other strange ways to implement logic programs. In *PODS-86*, S. 1–16.

Banko, M. und Brill, E. (2001). Scaling to very very large corpora for natural language disambiguation. In *ACL-01*, S. 26–33.

Banko, M., Brill, E., Dumais, S. T. und Lin, J. (2002). Askmsr: Question answering using the worldwide web. In *Proc. AAAI Spring Symposium on Mining Answers from Texts and Knowledge Bases*, S. 7–9.

Banko, M., Cafarella, M. J., Soderland, S., Broadhead, M. und Etzioni, O. (2007). Open information extraction from the web. In *IJCAI-07*.

Banko, M. und Etzioni, O. (2008). The tradeoffs between open and traditional relation extraction. In *ACL-08*, S. 28–36.

Bar-Hillel, Y. (1954). Indexical expressions. *Mind*, *63*, 359–379.

Bar-Hillel, Y. (1960). The present status of automatic translation of languages. In Alt, F. L. (Hrsg.), *Advances in Computers*, Band 1, S. 91–163. Academic Press.

Bar-Shalom, Y. (Hrsg.). (1992). *Multitargetmultisensor tracking: Advanced applications*. Artech House.

Bar-Shalom, Y. und Fortmann, T. E. (1988). *Tracking and Data Association*. Academic Press.

Bartak, R. (2001). Theory and practice of constraint propagation. In *Proc. Third Workshop on Constraint Programming for Decision and Control (CPDC-01)*, S. 7–14.

Barto, A. G., Bradtke, S. J. und Singh, S. P. (1995). Learning to act using real-time dynamic programming. *AIJ*, *73*(1), 81–138.

Barto, A. G., Sutton, R. S. und Anderson, C. W. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man and Cybernetics*, *13*, 834–846.

Barto, A. G., Sutton, R. S. und Brouwer, P. S. (1981). Associative search network: A reinforcement learning associative memory. *Biological Cybernetics*, *40*(3), 201–211.

Barwise, J. und Etchemendy, J. (1993). *The Language of First-Order Logic: Including the Macintosh Program Tarski's World 4.0* (3. überarbeitete und erweiterte Auflage). Center for the Study of Language and Information (CSLI).

Barwise, J. und Etchemendy, J. (2002). *Language, Proof and Logic*. CSLI (Univ. of Chicago Press).

Baum, E., Boneh, D. und Garrett, C. (1995). On genetic algorithms. In *COLT-95*, S. 230–239.

Baum, E. und Haussler, D. (1989). What size net gives valid generalization? *Neural Computation*, *1*(1), 151–160.

Baum, E. und Smith, W. D. (1997). A Bayesian approach to relevance in game playing. *AIJ*, *97*(1–2), 195–242.

Baum, E. und Wilczek, F. (1988). Supervised learning of probability distributions by neural networks. In Anderson, D. Z. (Hrsg.), *Neural Information Processing Systems*, S. 52–61. American Institute of Physics.

Baum, L. E. und Petrie, T. (1966). Statistical inference for probabilistic functions of finite state Markov chains. *Annals of Mathematical Statistics*, *41*.

Baxter, J. und Bartlett, P. (2000). Reinforcement learning in POMDP's via direct gradient ascent. In *ICML-00*, S. 41–48.

Bayardo, R. J. und Miranker, D. P. (1994). An optimal backtrack algorithm for tree-structured constraint satisfaction problems. *AIJ*, *71*(1), 159–181.

Bayardo, R. J. und Schrag, R. C. (1997). Using CSP look-back techniques to solve real-world SAT instances. In *AAAI-97*, S. 203–208.

Bayes, T. (1763). An essay towards solving a problem in the doctrine of chances. *Philosophical Transactions of the Royal Society of London*, *53*, 370–418.

Beal, D. F. (1980). An analysis of minimax. In Clarke, M. R. B. (Hrsg.), *Advances in Computer Chess 2*, S. 103–109. Edinburgh University Press.

Beal, J. und Winston, P. H. (2009). The new frontier of human-level artificial intelligence. *IEEE Intelligent Systems*, *24*(4), 21–23.

Beckert, B. und Posegga, J. (1995). Leantap: Lean, tableau-based deduction. *JAR*, *15*(3), 339–358.

Beeri, C., Fagin, R., Maier, D. und Yannakakis, M. (1983). On the desirability of acyclic database schemes. *JACM*, *30*(3), 479–513.

Bekey, G. (2008). *Robotics: State Of The Art And Future Challenges*. Imperial College Press.

Bell, C. und Tate, A. (1985). Using temporal constraints to restrict search in a planner. In *Proc. Third Alvey IKBS SIG Workshop*.

Bell, J. L. und Machover, M. (1977). *A Course in Mathematical Logic*. Elsevier/North-Holland.

Bellman, R. E. (1952). On the theory of dynamic programming. *PNAS*, *38*, 716–719.

Bellman, R. E. (1961). *Adaptive Control Processes: A Guided Tour*. Princeton University Press.

Bellman, R. E. (1965). On the application of dynamic programming to the determination of optimal play in chess and checkers. *PNAS*, *53*, 244–246.

Bellman, R. E. (1978). *An Introduction to Artificial Intelligence: Can Computers Think?* Boyd & Fraser Publishing Company.

Bellman, R. E. (1984). *Eye of the Hurricane*. World Scientific.

Bellman, R. E. und Dreyfus, S. E. (1962). *Applied Dynamic Programming*. Princeton University Press.

Bellman, R. E. (1957). *Dynamic Programming*. Princeton University Press.

Belongie, S., Malik, J. und Puzicha, J. (2002). Shape matching and object recognition using shape contexts. *PAMI*, *24*(4), 509–522.

Ben-Tal, A. und Nemirovski, A. (2001). *Lectures on Modern Convex Optimization: Analysis, Algorithms, and Engineering Applications*. SIAM (Society for Industrial and Applied Mathematics).

Bender, E. A. (1996). *Mathematical methods in artificial intelligence*. IEEE Computer Society Press.

Bengio, Y. und LeCun, Y. (2007). Scaling learning algorithms towards AI. In Bottou, L., Chapelle, O., DeCoste, D. und Weston, J. (Hrsg.), *Large-Scale Kernel Machines*. MIT Press.

Bentham, J. (1823). *Principles of Morals and Legislation*. Oxford University Press, Oxford, UK. Originalarbeit veröffentlicht in 1789.

Berger, J. O. (1985). *Statistical Decision Theory and Bayesian Analysis*. Springer Verlag.

Berkson, J. (1944). Application of the logistic function to bio-assay. *JASA*, *39*, 357–365.

Berlekamp, E. R., Conway, J. H. und Guy, R. K. (1982). *Winning Ways, For Your Mathematical Plays*. Academic Press.

Berlekamp, E. R. und Wolfe, D. (1994). *Mathematical Go: Chilling Gets the Last Point*. A.K. Peters.

Berleur, J. und Brunnstein, K. (2001). *Ethics of Computing: Codes, Spaces for Discussion and Law*. Chapman and Hall.

Berliner, H. J. (1979). The B* tree search algorithm: A best-first proof procedure. *AIJ*, *12*(1), 23–40.

Berliner, H. J. (1980a). Backgammon computer program beats world champion. *AIJ*, *14*, 205–220.

Berliner, H. J. (1980b). Computer backgammon. *Scientific American*, *249*(6), 64–72.

Bernardo, J. M. und Smith, A. F. M. (1994). *Bayesian Theory*. Wiley.

Berners-Lee, T., Hendler, J. und Lassila, O. (2001). The semantic web. *Scientific American*, *284*(5), 34–43.

Bernoulli, D. (1738). Specimen theoriae novae de mensura sortis. *Proc. St. Petersburg Imperial Academy of Sciences*, *5*, 175–192.

Bernstein, A. und Roberts, M. (1958). Computer vs. chess player. *Scientific American*, *198*(6), 96–105.

Bernstein, P. L. (1996). *Against the Odds: The Remarkable Story of Risk*. Wiley.

Berrou, C., Glavieux, A. und Thitimajshima, P. (1993). Near Shannon limit error control-correcting coding and decoding: Turbo-codes. 1. In *Proc. IEEE International Conference on Communications*, S. 1064–1070.

Berry, D. A. und Fristedt, B. (1985). *Bandit Problems: Sequential Allocation of Experiments*. Chapman and Hall.

Bertele, U. und Brioschi, F. (1972). *Nonserial dynamic programming*. Academic Press.

Bertoli, P., Cimatti, A. und Roveri, M. (2001a). Heuristic search + symbolic model checking = efficient conformant planning. In *IJCAI-01*, S. 467–472.

Bertoli, P., Cimatti, A., Roveri, M. und Traverso, P. (2001b). Planning in nondeterministic domains under partial observability via symbolic model checking. In *IJCAI-01*, S. 473–478.

Bertot, Y., Casteran, P., Huet, G. und Paulin-Mohring, C. (2004). *Interactive Theorem Proving and Program Development*. Springer.

Bertsekas, D. (1987). *Dynamic Programming: Deterministic and Stochastic Models*. Prentice-Hall.

Bertsekas, D. und Tsitsiklis, J. N. (1996). *Neurodynamic programming*. Athena Scientific.

Bertsekas, D. und Tsitsiklis, J. N. (2008). *Introduction to Probability* (2. Auflage). Athena Scientific.

Bertsekas, D. und Shreve, S. E. (2007). *Stochastic Optimal Control: The Discrete-Time Case*. Athena Scientific.

Bessière, C. (2006). Constraint propagation. In Rossi, F., van Beek, P. und Walsh, T. (Hrsg.), *Handbook of Constraint Programming*. Elsevier.

Bhar, R. und Hamori, S. (2004). *Hidden Markov Models: Applications to Financial Economics*. Springer.

Bibel, W. (1993). *Deduction: Automated Logic*. Academic Press.

Biere, A., Heule, M., van Maaren, H. und Walsh, T. (Hrsg.). (2009). *Handbook of Satisfiability*. IOS Press.

Billings, D., Burch, N., Davidson, A., Holte, R., Schaeffer, J., Schauenberg, T. und Szafron, D. (2003). Approximating game-theoretic optimal strategies for full-scale poker. In *IJCAI-03*.

Binder, J., Koller, D., Russell, S. J. und Kanazawa, K. (1997a). Adaptive probabilistic networks with hidden variables. *Machine Learning*, *29*, 213–244.

Binder, J., Murphy, K. und Russell, S. J. (1997b). Space-efficient inference in dynamic probabilistic networks. In *IJCAI-97*, S. 1292–1296.

Binford, T. O. (1971). Visual perception by computer. Invited paper presented at the IEEE Systems Science and Cybernetics Conference, Miami.

Binmore, K. (1982). *Essays on Foundations of Game Theory*. Pitman.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford University Press.

Bishop, C. M. (2007). *Pattern Recognition and Machine Learning*. Springer-Verlag.

Bisson, T. (1990). They're made out of meat. *Omni Magazine*.

Bistarelli, S., Montanari, U. und Rossi, F. (1997). Semiring-based constraint satisfaction and optimization. *JACM*, *44*(2), 201–236.

Bitner, J. R. und Reingold, E. M. (1975). Backtrack programming techniques. *CACM*, *18*(11), 651–656.

Bizer, C., Auer, S., Kobilarov, G., Lehmann, J. und Cyganiak, R. (2007). DBPedia – querying wikipedia like a database. In *Developers Track Presentation at the 16th International Conference on World Wide Web*.

Blazewicz, J., Ecker, K., Pesch, E., Schmidt, G. und Weglarz, J. (2007). *Handbook on Scheduling: Models and Methods for Advanced Planning (International Handbooks on Information Systems)*. Springer-Verlag New York, Inc.

Blei, D. M., Ng, A. Y. und Jordan, M. I. (2001). Latent Dirichlet Allocation. In *Neural Information Processing Systems*, Band 14.

Bliss, C. I. (1934). The method of probits. *Science*, *79*(2037), 38–39.

Block, H. D., Knight, B. und Rosenblatt, F. (1962). Analysis of a four-layer series-coupled perceptron. *Rev. Modern Physics*, *34*(1), 275–282.

Blum, A. L. und Furst, M. (1995). Fast planning through planning graph analysis. In *IJCAI-95*, S. 1636–1642.

Blum, A. L. und Furst, M. (1997). Fast planning through planning graph analysis. *AIJ*, *90*(1–2), 281–300.

Blum, A. L. (1996). On-line algorithms in machine learning. In *Proc. Workshop on On-Line Algorithms, Dagstuhl*, S. 306–325.

Blum, A. L. und Mitchell, T. M. (1998). Combining labeled and unlabeled data with co-training. In *COLT-98*, S. 92–100.

Blumer, A., Ehrenfeucht, A., Haussler, D. und Warmuth, M. (1989). Learnability and the Vapnik-Chervonenkis dimension. *JACM*, *36*(4), 929–965.

Bobrow, D. G. (1967). Natural language input for a computer problem solving system. In Minsky, M. L. (Hrsg.), *Semantic Information Processing*, S. 133–215. MIT Press.

Bobrow, D. G., Kaplan, R., Kay, M., Norman, D. A., Thompson, H. und Winograd, T. (1977). GUS, a frame driven dialog system. *AIJ*, *8*, 155–173.

Boden, M. A. (1977). *Artificial Intelligence and Natural Man*. Basic Books.

Boden, M. A. (Hrsg.). (1990). *The Philosophy of Artificial Intelligence*. Oxford University Press.

Bolognesi, A. und Ciancarini, P. (2003). Computer programming of kriegspiel endings: The case of KR vs. k. In *Advances in Computer Games 10*.

Bonet, B. (2002). An epsilon-optimal grid-based algorithm for partially observable Markov decision processes. In *ICML-02*, S. 51–58.

Bonet, B. und Geffner, H. (1999). Planning as heuristic search: New results. In *ECP-99*, S. 360–372.

Bonet, B. und Geffner, H. (2000). Planning with incomplete information as heuristic search in belief space. In *ICAPS-00*, S. 52–61.

Bonet, B. und Geffner, H. (2005). An algorithm better than AO*? In *AAAI-05*

Boole, G. (1847). *The Mathematical Analysis of Logic: Being an Essay towards a Calculus of Deductive Reasoning*. Macmillan, Barclay und Macmillan, Cambridge.

Booth, T. L. (1969). Probabilistic representation of formal languages. In *IEEE Conference Record of the 1969 Tenth Annual Symposium on Switching and Automata Theory*, S. 74–81.

Borel, E. (1921). La théorie du jeu et les équations intégrales à noyau symétrique. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences*, *173*, 1304–1308.

Borenstein, J., Everett, B. und Feng, L. (1996). *Navigating Mobile Robots: Systems and Techniques*. A. K. Peters, Ltd.

Borenstein, J. und Koren., Y. (1991). The vector field histogram – Fast obstacle avoidance for mobile robots. *IEEE Transactions on Robotics and Automation*, *7*(3), 278–288.

Borgida, A., Brachman, R. J., McGuinness, D. und Alperin Resnick, L. (1989). CLASSIC: A structural data model for objects. *SIGMOD Record*, *18*(2), 58–67.

Boroditsky, L. (2003). Linguistic relativity. In Nadel, L. (Hrsg.), *Encyclopedia of Cognitive Science*, S. 917–921. Macmillan.

Boser, B., Guyon, I. und Vapnik, V. N. (1992). A training algorithm for optimal margin classifiers. In *COLT-92*.

Bosse, M., Newman, P., Leonard, J., Soika, M., Feiten, W. und Teller, S. (2004). Simultaneous localization and map building in large-scale cyclic environments using the atlas framework. *Int. J. Robotics Research*, *23*(12), 1113–1139.

Bourzutschky, M. (2006). 7-man endgames with pawns. *CCRL Discussion Board, kirill-kryukov.com/chess/ discussion-board/viewtopic.php?t=805*.

Boutilier, C. und Brafman, R. I. (2001). Partial-order planning with concurrent interacting actions. *JAIR*, *14*, 105–136.

Boutilier, C., Dearden, R. und Goldszmidt, M. (2000). Stochastic dynamic programming with factored representations. *AIJ*, *121*, 49–107.

Boutilier, C., Reiter, R. und Price, B. (2001). Symbolic dynamic programming for first-order MDPs. In *IJCAI-01*, S. 467–472.

Boutilier, C., Friedman, N., Goldszmidt, M. und Koller, D. (1996). Context-specific independence in Bayesian networks. In *UAI-96*, S. 115–123.

Bouzy, B. und Cazenave, T. (2001). Computer go: An AI oriented survey. *AIJ*, *132*(1), 39–103.

Bowerman, M. und Levinson, S. (2001). *Language acquisition and conceptual development*. Cambridge University Press.

Bowling, M., Johanson, M., Burch, N. und Szafron, D. (2008). Strategy evaluation in extensive games with importance sampling. In *ICML-08*.

Box, G. E. P. (1957). Evolutionary operation: A method of increasing industrial productivity. *Applied Statistics*, *6*, 81–101.

Box, G. E. P., Jenkins, G. und Reinsel, G. (1994). *Time Series Analysis: Forecasting and Control* (3. Auflage). Prentice Hall.

Boyan, J. A. (2002). Technical update: Least-squares temporal difference learning. *Machine Learning*, *49*(2–3), 233–246.

Boyan, J. A. und Moore, A. W. (1998). Learning evaluation functions for global optimization and Boolean satisfiability. In *AAAI-98*.

Boyd, S. und Vandenberghe, L. (2004). *Convex Optimization*. Cambridge University Press.

Boyen, X., Friedman, N. und Koller, D. (1999). Discovering the hidden structure of complex dynamic systems. In *UAI-99*.

Boyer, R. S. und Moore, J. S. (1979). *A Computational Logic*. Academic Press.

Boyer, R. S. und Moore, J. S. (1984). Proof checking the RSA public key encryption algorithm. *American Mathematical Monthly*, *91*(3), 181–189.

Brachman, R. J. (1979). On the epistemological status of semantic networks. In Findler, N. V. (Hrsg.), *Associative Networks: Representation and Use of Knowledge by Computers*, S. 3–50. Academic Press.

Brachman, R. J., Fikes, R. E. und Levesque, H. J. (1983). Krypton: A functional approach to knowledge representation. *Computer*, *16*(10), 67–73.

Brachman, R. J. und Levesque, H. J. (Hrsg.). (1985). *Readings in Knowledge Representation*. Morgan Kaufmann.

Bradtke, S. J. und Barto, A. G. (1996). Linear least-squares algorithms for temporal difference learning. *Machine Learning*, *22*, 33–57.

Brafman, O. und Brafman, R. (2009). *Sway: The Irresistible Pull of Irrational Behavior*. Broadway Business.

Brafman, R. I. und Domshlak, C. (2008). From one to many: Planning for loosely coupled multi-agent systems. In *ICAPS-08*, S. 28–35.

Brafman, R. I. und Tennenholtz, M. (2000). A near optimal polynomial time algorithm for learning in certain classes of stochastic games. *AIJ*, *121*, 31–47.

Braitenberg, V. (1984). *Vehicles: Experiments in Synthetic Psychology*. MIT Press.

Bransford, J. und Johnson, M. (1973). Consideration of some problems in comprehension. In Chase, W. G. (Hrsg.), *Visual Information Processing*. Academic Press.

Brants, T., Popat, A. C., Xu, P., Och, F. J. und Dean, J. (2007). Large language models in machine translation. In *EMNLP-CoNLL-2007: Proc. 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, S. 858–867.

Bratko, I. (1986). *Prolog Programming for Artificial Intelligence* (1. Auflage). Addison-Wesley.

Bratko, I. (2001). *Prolog Programming for Artificial Intelligence* (3. Auflage). Addison-Wesley.

Bratman, M. E. (1987). *Intention, Plans, and Practical Reason*. Harvard University Press.

Bratman, M. E. (1992). Planning and the stability of intention. *Minds and Machines*, *2*(1), 1–16.

Breese, J. S. (1992). Construction of belief and decision networks. *Computational Intelligence*, *8*(4), 624–647.

Breese, J. S. und Heckerman, D. (1996). Decision-theoretic troubleshooting: A framework for repair and experiment. In *UAI-96*, S. 124–132.

Breiman, L. (1996). Bagging predictors. *Machine Learning*, *24*(2), 123–140.

Breiman, L., Friedman, J., Olshen, R. A. und Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth International Group.

Brelaz, D. (1979). New methods to color the vertices of a graph. *CACM*, *22*(4), 251–256.

Brent, R. P. (1973). *Algorithms for minimization without derivatives*. Prentice-Hall.

Bresnan, J. (1982). *The Mental Representation of Grammatical Relations*. MIT Press.

Brewka, G., Dix, J. und Konolige, K. (1997). *Nononotonic Reasoning: An Overview*. CSLI Publications.

Brickley, D. und Guha, R. V. (2004). RDF vocabulary description language 1.0: RDF schema. Tech. rep., W3C.

Bridle, J. S. (1990). Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In Fogelman Soulié, F. und Hérault, J. (Hrsg.), *Neurocomputing: Algorithms, Architectures and Applications*. Springer-Verlag.

Briggs, R. (1985). Knowledge representation in Sanskrit and artificial intelligence. *AIMag*, *6*(1), 32–39.

Brin, D. (1998). *The Transparent Society*. Perseus.

Brin, S. (1999). Extracting patterns and relations from the world wide web. Technical report 1999-65, Stanford InfoLab.

Brin, S. und Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. In *Proc. Seventh World Wide Web Conference*.

Bringsjord, S. (2008). If I were judge. In Epstein, R., Roberts, G. und Beber, G. (Hrsg.), *Parsing the Turing Test*. Springer.

Broadbent, D. E. (1958). *Perception and Communication*. Pergamon.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, *2*, 14–23.

Brooks, R. A. (1989). Engineering approach to building complete, intelligent beings. *Proc. SPIE – the International Society for Optical Engineering*, *1002*, 618–625.

Brooks, R. A. (1991). Intelligence without representation. *AIJ*, *47*(1–3), 139–159.

Brooks, R. A. und Lozano-Perez, T. (1985). A subdivision algorithm in configuration space for find-path with rotation. *IEEE Transactions on Systems, Man and Cybernetics*, *15*(2), 224–233.

Brown, C., Finkelstein, L. und Purdom, P. (1988). Backtrack searching in the presence of symmetry. In Mora, T. (Hrsg.), *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, S. 99–110. Springer-Verlag.

Brown, K. C. (1974). A note on the apparent bias of net revenue estimates. *J. Finance*, *29*, 1215–1216.

Brown, P. F., Cocke, J., Della Pietra, S. A., Della Pietra, V. J., Jelinek, F., Mercer, R. L. und Roossin, P. (1988). A statistical approach to language translation. In *COLING-88*, S. 71–76.

Brown, P. F., Della Pietra, S. A., Della Pietra, V. J. und Mercer, R. L. (1993). The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, *19*(2), 263–311.

Brownston, L., Farrell, R., Kant, E. und Martin, N. (1985). *Programming expert systems in OPS5: An introduction to rule-based programming*. Addison-Wesley.

Bruce, V., Georgeson, M. und Green, P. (2003). *Visual Perception: Physiology, Psychology and Ecology*. Psychology Press.

Bruner, J. S., Goodnow, J. J. und Austin, G. A. (1957). *A Study of Thinking*. Wiley.

Bryant, B. D. und Miikkulainen, R. (2007). Acquiring visibly intelligent behavior with example-guided neuroevolution. In *AAAI-07*.

Bryce, D. und Kambhampati, S. (2007). A tutorial on planning graph-based reachability heuristics. *AIMag*, *Spring*, 47–83.

Bryce, D., Kambhampati, S. und Smith, D. E. (2006). Planning graph heuristics for belief space search. *JAIR*, *26*, 35–99.

Bryson, A. E. und Ho, Y.-C. (1969). *Applied Optimal Control*. Blaisdell.

Buchanan, B. G. und Mitchell, T. M. (1978). Model-directed learning of production rules. In Waterman, D. A. und Hayes-Roth, F. (Hrsg.), *Pattern-Directed Inference Systems*, S. 297–312. Academic Press.

Buchanan, B. G., Mitchell, T. M., Smith, R. G. und Johnson, C. R. (1978). Models of learning systems. In *Encyclopedia of Computer Science and Technology*, Band 11. Dekker.

Buchanan, B. G. und Shortliffe, E. H. (Hrsg.). (1984). *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley.

Buchanan, B. G., Sutherland, G. L. und Feigenbaum, E. A. (1969). Heuristic DENDRAL: A program for generating explanatory hypotheses in organic chemistry. In Meltzer, B., Michie, D. und Swann, M. (Hrsg.), *Machine Intelligence 4*, S. 209–254. Edinburgh University Press.

Buehler, M., Iagnemma, K. und Singh, S. (Hrsg.). (2006). *The 2005 DARPA Grand Challenge: The Great Robot Race*. Springer-Verlag.

Bunt, H. C. (1985). The formal representation of (quasi-) continuous concepts. In Hobbs, J. R. und Moore, R. C. (Hrsg.), *Formal Theories of the Commonsense World*, Kap. 2, S. 37–70. Ablex.

Burgard, W., Cremers, A. B., Fox, D., Hähnel, D., Lakemeyer, G., Schulz, D., Steiner, W. und Thrun, S. (1999). Experiences with an interactive museum tour-guide robot. *AIJ*, *114*(1–2), 3–55.

Buro, M. (1995). ProbCut: An effective selective extension of the alpha-beta algorithm. *J. International Computer Chess Association*, *18*(2), 71–76.

Buro, M. (2002). Improving heuristic mini-max search by supervised learning. *AIJ*, *134*(1–2), 85–99.

Burstein, J., Leacock, C. und Swartz, R. (2001). Automated evaluation of essays and short answers. In *Fifth International Computer Assisted Assessment (CAA) Conference*.

Burton, R. (2009). *On Being Certain: Believing You Are Right Even When You're Not*. St. Martin's Griffin.

Buss, D. M. (2005). *Handbook of evolutionary psychology*. Wiley.

Butler, S. (1863). Darwin among the machines. *The Press (Christchurch, New Zealand), June 13*.

Bylander, T. (1992). Complexity results for serial decomposability. In *AAAI-92*, S. 729–734.

Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *AIJ*, *69*, 165–204.

Byrd, R. H., Lu, P., Nocedal, J. und Zhu, C. (1995). A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific and Statistical Computing*, *16*(5), 1190–1208.

Cabeza, R. und Nyberg, L. (2001). Imaging cognition II: An empirical review of 275 PET and fMRI studies. *J. Cognitive Neuroscience*, *12*, 1–47.

Cafarella, M. J., Halevy, A., Zhang, Y., Wang, D. Z. und Wu, E. (2008). Webtables: Exploring the power of tables on the web. In *VLDB-2008*.

Calvanese, D., Lenzerini, M. und Nardi, D. (1999). Unifying class-based representation formalisms. *JAIR*, *11*, 199–240.

Campbell, M. S., Hoane, A. J. und Hsu, F.-H. (2002). Deep Blue. *AIJ*, *134*(1–2), 57–83.

Canny, J. und Reif, J. (1987). New lower bound techniques for robot motion planning problems. In *FOCS-87*, S. 39–48.

Canny, J. (1986). A computational approach to edge detection. *PAMI, 8*, 679–698.

Canny, J. (1988). *The Complexity of Robot Motion Planning*. MIT Press.

Capen, E., Clapp, R. und Campbell, W. (1971). Competitive bidding in high-risk situations. *J. Petroleum Technology*, *23*, 641–653.

Caprara, A., Fischetti, M. und Toth, P. (1995). A heuristic method for the set covering problem. *Operations Research*, *47*, 730–743.

Carbonell, J. G. (1983). Derivational analogy and its role in problem solving. In *AAAI-83*, S. 64–69.

Carbonell, J. G., Knoblock, C. A. und Minton, S. (1989). PRODIGY: An integrated architecture for planning and learning. Technical report CMU-CS-89-189, Computer Science Department, Carnegie-Mellon University.

Carbonell, J. R. und Collins, A. M. (1973). Natural semantics in artificial intelligence. In *IJCAI-73*, S. 344–351.

Cardano, G. (1663). *Liber de ludo aleae*. Lyons.

Carnap, R. (1928). *Der logische Aufbau der Welt*. Weltkreis-Verlag. Übersetzt ins Englische wie (Carnap, 1967).

Carnap, R. (1948). On the application of inductive logic. *Philosophy and Phenomenological Research*, *8*, 133–148.

Carnap, R. (1950). *Logical Foundations of Probability*. University of Chicago Press.

Carroll, S. (2007). *The Making of the Fittest: DNA and the Ultimate Forensic Record of Evolution*. Norton.

Casati, R. und Varzi, A. (1999). *Parts and places: the structures of spatial representation*. MIT Press.

Cassandra, A. R., Kaelbling, L. P. und Littman, M. L. (1994). Acting optimally in partially observable stochastic domains. In *AAAI-94*, S. 1023–1028.

Cassandras, C. G. und Lygeros, J. (2006). *Stochastic Hybrid Systems*. CRC Press.

Castro, R., Coates, M., Liang, G., Nowak, R. und Yu, B. (2004). Network tomography: Recent developments. *Statistical Science*, *19*(3), 499–517.

Cesa-Bianchi, N. und Lugosi, G. (2006). *Prediction, learning, and Games*. Cambridge University Press.

Cesta, A., Cortellessa, G., Denis, M., Donati, A., Fratini, S., Oddi, A., Policella, N., Rabenau, E. und Schulster, J. (2007). MEXAR2: AI solves mission planner problems. *IEEE Intelligent Systems*, *22*(4), 12–19.

Chakrabarti, P. P., Ghose, S., Acharya, A. und de Sarkar, S. C. (1989). Heuristic search in restricted memory. *AIJ*, *41*(2), 197–222.

Chandra, A. K. und Harel, D. (1980). Computable queries for relational data bases. *J. Computer and System Sciences*, *21*(2), 156–178.

Chang, C.-L. und Lee, R. C.-T. (1973). *Symbolic Logic and Mechanical Theorem Proving*. Academic Press.

Chapman, D. (1987). Planning for conjunctive goals. *AIJ*, *32*(3), 333–377.

Charniak, E. (1993). *Statistical Language Learning*. MIT Press.

Charniak, E. (1996). Tree-bank grammars. In *AAAI-96*, S. 1031–1036.

Charniak, E. (1997). Statistical parsing with a context-free grammar and word statistics. In *AAAI-97*, S. 598–603.

Charniak, E. und Goldman, R. (1992). A Bayesian model of plan recognition. *AIJ*, *64*(1), 53–79.

Charniak, E. und McDermott, D. (1985). *Introduction to Artificial Intelligence*. Addison-Wesley.

Charniak, E., Riesbeck, C., McDermott, D. und Meehan, J. (1987). *Artificial Intelligence Programming* (2. Auflage). Lawrence Erlbaum Associates.

Charniak, E. (1991). Bayesian networks without tears. *AIMag*, *12*(4), 50–63.

Charniak, E. und Johnson, M. (2005). Coarse-to-fine n-best parsing and maxent discriminative reranking. In *ACL-05*.

Chater, N. und Oaksford, M. (Hrsg.). (2008). *The probabilistic mind: Prospects for Bayesian cognitive science*. Oxford University Press.

Chatfield, C. (1989). *The Analysis of Time Series: An Introduction* (4. Auflage). Chapman and Hall.

Cheeseman, P. (1985). In defense of probability. In *IJCAI-85*, S. 1002–1009.

Cheeseman, P. (1988). An inquiry into computer understanding. *Computational Intelligence*, *4*(1), 58–66.

Cheeseman, P., Kanefsky, B. und Taylor, W. (1991). Where the really hard problems are. In *IJCAI-91*, S. 331–337.

Cheeseman, P., Self, M., Kelly, J. und Stutz, J. (1988). Bayesian classification. In *AAAI-88*, Band 2, S. 607–611.

Cheeseman, P. und Stutz, J. (1996). Bayesian classification (AutoClass): Theory and results. In Fayyad, U., Piatesky-Shapiro, G., Smyth, P. und Uthurusamy, R. (Hrsg.), *Advances in Knowledge Discovery and Data Mining*. AAAI Press/MIT Press.

Chen, S. F. und Goodman, J. (1996). An empirical study of smoothing techniques for language modeling. In *ACL-96*, S. 310–318.

Cheng, J. und Druzdzel, M. J. (2000). AIS-BN: An adaptive importance sampling algorithm for evidential reasoning in large Bayesian networks. *JAIR*, *13*, 155–188.

Cheng, J., Greiner, R., Kelly, J., Bell, D. A. und Liu, W. (2002). Learning Bayesian networks from data: An information-theory based approach. *AIJ*, *137*, 43–90.

Chklovski, T. und Gil, Y. (2005). Improving the design of intelligent acquisition interfaces for collecting world knowledge from web contributors. In *Proc. Third International Conference on Knowledge Capture (K-CAP)*.

Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, *2*(3), 113–124.

Chomsky, N. (1957). *Syntactic Structures*. Mouton.

Choset, H. (1996). *Sensor Based Motion Planning: The Hierarchical Generalized Voronoi Graph*. Ph.D. thesis, California Institute of Technology.

Choset, H., Lynch, K., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. und Thrun, S. (2004). *Principles of Robotic Motion: Theory, Algorithms, and Implementation*. MIT Press.

Chung, K. L. (1979). *Elementary Probability Theory with Stochastic Processes* (3. Auflage). Springer-Verlag.

Church, A. (1936). A note on the Entscheidungsproblem. *JSL*, *1*, 40–41 und 101–102.

Church, A. (1956). *Introduction to Mathematical Logic*. Princeton University Press.

Church, K. und Patil, R. (1982). Coping with syntactic ambiguity or how to put the block in the box on the table. *Computational Linguistics*, *8*(3–4), 139–149.

Church, K. (2004). Speech and language processing: Can we use the past to predict the future. In *Proc. Conference on Text, Speech, and Dialogue*.

Church, K. und Gale, W. A. (1991). A comparison of the enhanced Good–Turing and deleted estimation methods for estimating probabilities of English bigrams. *Computer Speech and Language*, *5*, 19–54.

Churchland, P. M. und Churchland, P. S. (1982). Functionalism, qualia, and intentionality. In Biro, J. I. und Shahan, R. W. (Hrsg.), *Mind, Brain and Function: Essays in the Philosophy of Mind*, S. 121–145. University of Oklahoma Press.

Churchland, P. S. (1986). *Neurophilosophy: Toward a Unified Science of the Mind–Brain*. MIT Press.

Ciancarini, P. und Wooldridge, M. (2001). *Agent-Oriented Software Engineering*. Springer-Verlag.

Cimatti, A., Roveri, M. und Traverso, P. (1998). Automatic OBDD-based generation of universal plans in non-deterministic domains. In *AAAI-98*, S. 875–881.

Clark, A. (1998). *Being There: Putting Brain, Body, and World Together Again*. MIT Press.

Clark, A. (2008). *Supersizing the Mind: Embodiment, Action, and Cognitive Extension*. Oxford University Press.

Clark, K. L. (1978). Negation as failure. In Gallaire, H. und Minker, J. (Hrsg.), *Logic and Data Bases*, S. 293–322. Plenum.

Clark, P. und Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning*, *3*, 261–283.

Clark, S. und Curran, J. R. (2004). Parsing the WSJ using CCG and log-linear models. In *ACL-04*, S. 104–111.

Clarke, A. C. (1968a). *2001: A Space Odyssey*. Signet.

Clarke, A. C. (1968b). The world of 2001. Vogue.

Clarke, E. und Grumberg, O. (1987). Research on automatic verification of finite-state concurrent systems. *Annual Review of Computer Science*, *2*, 269–290.

Clarke, M. R. B. (Hrsg.). (1977). *Advances in Computer Chess 1*. Edinburgh University Press.

Clearwater, S. H. (Hrsg.). (1996). *Market-Based Control*. World Scientific.

Clocksin, W. F. und Mellish, C. S. (2003). *Programming in Prolog* (5. Auflage). Springer-Verlag.

Clocksin, W. F. (2003). *Clause and Effect: Prolog Programming for the Working Programmer*. Springer.

Coarfa, C., Demopoulos, D., Aguirre, A., Subramanian, D. und Yardi, M. (2003). Random 3-SAT: The plot thickens. *Constraints*, *8*(3), 243–261.

Coates, A., Abbeel, P. und Ng, A. Y. (2009). Apprenticeship learning for helicopter control. *JACM*, *52*(7), 97–105.

Cobham, A. (1964). The intrinsic computational difficulty of functions. In *Proc. 1964 International Congress for Logic, Methodology, and Philosophy of Science*, S. 24–30.

Cohen, P. R. (1995). *Empirical methods for artificial intelligence*. MIT Press.

Cohen, P. R. und Levesque, H. J. (1990). Intention is choice with commitment. *AIJ*, *42*(2–3), 213–261.

Cohen, P. R., Morgan, J. und Pollack, M. E. (1990). *Intentions in Communication*. MIT Press.

Cohen, W. W. und Page, C. D. (1995). Learnability in inductive logic programming: Methods and results. *New Generation Computing*, *13*(3–4), 369–409.

Cohn, A. G., Bennett, B., Gooday, J. M. und Gotts, N. (1997). RCC: A calculus for region based qualitative spatial reasoning. *GeoInformatica*, *1*, 275–316.

Collin, Z., Dechter, R. und Katz, S. (1999). Self-stabilizing distributed constraint satisfaction. *Chicago Journal of Theoretical Computer Science*, *1999*(115).

Collins, F. S., Morgan, M. und Patrinos, A. (2003). The human genome project: Lessons from large-scale biology. *Science*, *300*(5617), 286–290.

Collins, M. (1999). *Head-driven Statistical Models for Natural Language Processing*. Ph.D. thesis, University of Pennsylvania.

Collins, M. und Duffy, K. (2002). New ranking algorithms for parsing and tagging: Kernels over discrete structures, and the voted perceptron. In *ACL-02*.

Colmerauer, A. und Roussel, P. (1993). The birth of Prolog. *SIGPLAN Notices*, *28*(3), 37–52.

Colmerauer, A. (1975). Les grammaires de metamorphose. Tech. rep., Groupe d'Intelligence Artificielle, Université de Marseille-Luminy.

Colmerauer, A., Kanoui, H., Pasero, R. und Roussel, P. (1973). Un systéme de communication homme–machine en Français. Rapport, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille II.

Condon, J. H. und Thompson, K. (1982). Belle chess hardware. In Clarke, M. R. B. (Hrsg.), *Advances in Computer Chess 3*, S. 45–54. Pergamon.

Congdon, C. B., Huber, M., Kortenkamp, D., Bidlack, C., Cohen, C., Huffman, S., Koss, F., Raschke, U. und Weymouth, T. (1992). CARMEL versus Flakey: A comparison of two robots. Tech. rep. Papers from the AAAI Robot Competition, RC-92-01, American Association for Artificial Intelligence.

Conlisk, J. (1989). Three variants on the Allais example. *American Economic Review*, *79*(3), 392–407.

Connell, J. (1989). *A Colony Architecture for an Artificial Creature*. Ph.D. thesis, Artificial Intelligence Laboratory, MIT. Also available as AI Technical Report 1151.

Consortium, T. G. O. (2008). The gene ontology project in 2008. *Nucleic Acids Research*, *36*.

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *STOC-71*, S. 151–158.

Cook, S. A. und Mitchell, D. (1997). Finding hard instances of the satisfiability problem: A survey. In Du, D., Gu, J. und Pardalos, P. (Hrsg.), *Satisfiability problems: Theory and applications*. American Mathematical Society.

Cooper, G. (1990). The computational complexity of probabilistic inference using Bayesian belief networks. *AIJ*, *42*, 393–405.

Cooper, G. und Herskovits, E. (1992). A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, *9*, 309–347.

Copeland, J. (1993). *Artificial Intelligence: A Philosophical Introduction*. Blackwell.

Copernicus (1543). *De Revolutionibus Orbium Coelestium*. Apud Ioh. Petreium, Nuremberg.

Cormen, T. H., Leiserson, C. E. und Rivest, R. (1990). *Introduction to Algorithms*. MIT Press.

Cortes, C. und Vapnik, V. N. (1995). Support vector networks. *Machine Learning*, *20*, 273–297.

Cournot, A. (Hrsg.). (1838). *Recherches sur les principes mathématiques de la théorie des richesses*. L. Hachette, Paris.

Cover, T. und Thomas, J. (2006). *Elements of Information Theory* (2. Auflage). Wiley.

Cowan, J. D. und Sharp, D. H. (1988a). Neural nets. *Quarterly Reviews of Biophysics*, *21*, 365–427.

Cowan, J. D. und Sharp, D. H. (1988b). Neural nets and artificial intelligence. *Daedalus*, *117*, 85–121.

Cowell, R., Dawid, A. P., Lauritzen, S. und Spiegelhalter, D. J. (2002). *Probabilistic Networks and Expert Systems*. Springer.

Cox, I. (1993). A review of statistical data association techniques for motion correspondence. *IJCV*, *10*, 53–66.

Cox, I. und Hingorani, S. L. (1994). An efficient implementation and evaluation of Reid's multiple hypothesis tracking algorithm for visual tracking. In *ICPR-94*, Band 1, S. 437–442.

Cox, I. und Wilfong, G. T. (Hrsg.). (1990). *Autonomous Robot Vehicles*. Springer-Verlag.

Cox, R. T. (1946). Probability, frequency, and reasonable expectation. *American Journal of Physics*, *14*(1), 1–13.

Craig, J. (1989). *Introduction to Robotics: Mechanics and Control* (2. Auflage). Addison-Wesley Publishing, Inc.

Craik, K. J. (1943). *The Nature of Explanation*. Cambridge University Press.

Craswell, N., Zaragoza, H. und Robertson, S. E. (2005). Microsoft cambridge at trec-14: Enterprise track. In *Proc. Fourteenth Text REtrieval Conference*.

Crauser, A., Mehlhorn, K., Meyer, U. und Sanders, P. (1998). A parallelization of Dijkstra's shortest path algorithm. In *Proc. 23rd International Symposium on Mathematical Foundations of Computer Science,*, S. 722–731.

Craven, M., DiPasquo, D., Freitag, D., McCallum, A., Mitchell, T. M., Nigam, K. und Slattery, S. (2000). Learning to construct knowledge bases from the World Wide Web. *AIJ*, *118*(1/2), 69–113.

Crawford, J. M. und Auton, L. D. (1993). Experimental results on the crossover point in satisfiability problems. In *AAAI-93*, S. 21–27.

Cristianini, N. und Hahn, M. (2007). *Introduction to Computational Genomics: A Case Studies Approach*. Cambridge University Press.

Cristianini, N. und Schölkopf, B. (2002). Support vector machines and kernel methods: The new generation of learning machines. *AIMag*, *23*(3), 31–41.

Cristianini, N. und Shawe-Taylor, J. (2000). *An introduction to support vector machines and other kernel-based learning methods*. Cambridge University Press.

Crockett, L. (1994). *The Turing Test and the Frame Problem: AI's Mistaken Understanding of Intelligence*. Ablex.

Croft, B., Metzler, D. und Stroham, T. (2009). *Search Engines: Information retrieval in Practice*. Addison Wesley.

Cross, S. E. und Walker, E. (1994). DART: Applying knowledge based planning and scheduling to crisis action planning. In Zweben, M. und Fox, M. S. (Hrsg.), *Intelligent Scheduling*, S. 711–729. Morgan Kaufmann.

Cruse, D. A. (1986). *Lexical Semantics*. Cambridge University Press.

Culberson, J. und Schaeffer, J. (1996). Searching with pattern databases. In *Advances in Artificial Intelligence (Lecture Notes in Artificial Intelligence 1081)*, S. 402–416. Springer-Verlag.

Culberson, J. und Schaeffer, J. (1998). Pattern databases. *Computational Intelligence*, *14*(4), 318–334.

Cullingford, R. E. (1981). Integrating knowledge sources for computer "understanding" tasks. *IEEE Transactions on Systems, Man and Cybernetics (SMC)*, *11*.

Cummins, D. und Allen, C. (1998). *The Evolution of Mind*. Oxford University Press.

Cushing, W., Kambhampati, S., Mausam und Weld, D. S. (2007). When is temporal planning *really* temporal? In *IJCAI-07*.

Cybenko, G. (1988). Continuous valued neural networks with two hidden layers are sufficient. Technical report, Department of Computer Science, Tufts University.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Controls, Signals, and Systems*, *2*, 303–314.

Daganzo, C. (1979). *Multinomial probit: The theory and its application to demand forecasting*. Academic Press.

Dagum, P. und Luby, M. (1993). Approximating probabilistic inference in Bayesian belief networks is NP-hard. *AIJ*, *60*(1), 141–153.

Dalal, N. und Triggs, B. (2005). Histograms of oriented gradients for human detection. In *CVPR*, S. 886–893.

Dantzig, G. B. (1949). Programming of interdependent activities: II. Mathematical model. *Econometrica*, *17*, 200–211.

Darwiche, A. (2001). Recursive conditioning. *AIJ*, *126*, 5–41.

Darwiche, A. und Ginsberg, M. L. (1992). A symbolic generalization of probability theory. In *AAAI-92*, S. 622–627.

Darwiche, A. (2009). *Modeling and reasoning with Bayesian networks*. Cambridge University Press.

Darwin, C. (1859). *On The Origin of Species by Means of Natural Selection*. J. Murray, London.

Darwin, C. (1871). *Descent of Man*. J. Murray.

Dasgupta, P., Chakrabarti, P. P. und de Sarkar, S. C. (1994). Agent searching in a tree and the optimality of iterative deepening. *AIJ*, *71*, 195–208.

Davidson, D. (1980). *Essays on Actions and Events*. Oxford University Press.

Davies, T. R. (1985). Analogy. Informal note INCSLI-85-4, Center for the Study of Language and Information (CSLI).

Davies, T. R. und Russell, S. J. (1987). A logical approach to reasoning by analogy. In *IJCAI-87*, Band 1, S. 264–270.

Davis, E. (1986). *Representing and Acquiring Geographic Knowledge*. Pitman and Morgan Kaufmann.

Davis, E. (1990). *Representations of Commonsense Knowledge*. Morgan Kaufmann.

Davis, E. (2005). Knowledge and communication: A first-order theory. *AIJ*, *166*, 81–140.

Davis, E. (2006). The expressivity of quantifying over regions. *J. Logic and Computation*, *16*, 891–916.

Davis, E. (2007). Physical reasoning. In van Harmelan, F., Lifschitz, V. und Porter, B. (Hrsg.), *The Handbook of Knowledge Representation*, S. 597–620. Elsevier.

Davis, E. (2008). Pouring liquids: A study in commonsense physical reasoning. *AIJ*, *172*(1540–1578).

Davis, E. und Morgenstern, L. (2004). Introduction: Progress in formal commonsense reasoning. *AIJ*, *153*, 1–12.

Davis, E. und Morgenstern, L. (2005). A first-order theory of communication and multi-agent plans. *J. Logic and Computation*, *15*(5), 701–749.

Davis, K. H., Biddulph, R. und Balashek, S. (1952). Automatic recognition of spoken digits. *J. Acoustical Society of America*, *24*(6), 637–642.

Davis, M. (1957). A computer program for Presburger's algorithm. In *Proving Theorems (as Done by Man, Logician, or Machine)*, S. 215–233. Proc. Summer Institute for Symbolic Logic. 2. Auflage; Veröffentlicht 1960.

Davis, M., Logemann, G. und Loveland, D. (1962). A machine program for theorem-proving. *CACM*, *5*, 394–397.

Davis, M. und Putnam, H. (1960). A computing procedure for quantification theory. *JACM*, *7*(3), 201–215.

Davis, R. und Lenat, D. B. (1982). *Knowledge-Based Systems in Artificial Intelligence*. McGraw-Hill.

Dayan, P. (1992). The convergence of TD(λ) for general λ. *Machine Learning*, *8*(3–4), 341–362.

Dayan, P. und Abbott, L. F. (2001). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. MIT Press.

Dayan, P. und Niv, Y. (2008). Reinforcement learning and the brain: The good, the bad and the ugly. *Current Opinion in Neurobiology*, *18*(2), 185–196.

de Dombal, F. T., Leaper, D. J., Horrocks, J. C. und Staniland, J. R. (1974). Human and computer-aided diagnosis of abdominal pain: Further report with emphasis on performance of clinicians. *British Medical Journal*, *1*, 376–380.

de Dombal, F. T., Staniland, J. R. und Clamp, S. E. (1981). Geographical variation in disease presentation. *Medical Decision Making*, *1*, 59–69.

de Finetti, B. (1937). Le prévision: ses lois logiques, ses sources subjectives. *Ann. Inst. Poincaré*, *7*, 1–68.

de Finetti, B. (1993). On the subjective meaning of probability. In Monari, P. und Cocchi, D. (Hrsg.), *Probabilita e Induzione*, S. 291–321. Clueb.

de Freitas, J. F. G., Niranjan, M. und Gee, A. H. (2000). Sequential Monte Carlo methods to train neural network models. *Neural Computation*, *12*(4), 933–953.

de Kleer, J. (1975). Qualitative and quantitative knowledge in classical mechanics. Tech. rep. AITR-352, MIT Artificial Intelligence Laboratory.

de Kleer, J. (1989). A comparison of ATMS and CSP techniques. In *IJCAI-89*, Band 1, S. 290–296.

de Kleer, J. und Brown, J. S. (1985). A qualitative physics based on confluences. In Hobbs, J. R. und Moore, R. C. (Hrsg.), *Formal Theories of the Commonsense World*, Kap. 4, S. 109–183. Ablex.

de Marcken, C. (1996). *Unsupervised Language Acquisition*. Ph.D. thesis, MIT.

De Morgan, A. (1864). On the syllogism, No. IV, and on the logic of relations. *Transaction of the Cambridge Philosophical Society*, *X*, 331–358.

De Raedt, L. (1992). *Interactive Theory Revision: An Inductive Logic Programming Approach*. Academic Press.

de Salvo Braz, R., Amir, E. und Roth, D. (2007). Lifted first-order probabilistic inference. In Getoor, L. und Taskar, B. (Hrsg.), *Introduction to Statistical Relational Learning*. MIT Press.

Deacon, T. W. (1997). *The symbolic species: The co-evolution of language and the brain*. W. W. Norton.

Deale, M., Yvanovich, M., Schnitzius, D., Kautz, D., Carpenter, M., Zweben, M., Davis, G. und Daun, B. (1994). The space shuttle ground processing scheduling system. In Zweben, M. und Fox, M. (Hrsg.), *Intelligent Scheduling*, S. 423–449. Morgan Kaufmann.

Dean, T., Basye, K., Chekaluk, R. und Hyun, S. (1990). Coping with uncertainty in a control system for navigation and exploration. In *AAAI-90*, Band 2, S. 1010–1015.

Dean, T. und Boddy, M. (1988). An analysis of time-dependent planning. In *AAAI-88*, S. 49–54.

Dean, T., Firby, R. J. und Miller, D. (1990). Hierarchical planning involving deadlines, travel time, and resources. *Computational Intelligence*, *6*(1), 381–398.

Dean, T., Kaelbling, L. P., Kirman, J. und Nicholson, A. (1993). Planning with deadlines in stochastic domains. In *AAAI-93*, S. 574–579.

Dean, T. und Kanazawa, K. (1989a). A model for projection and action. In *IJCAI-89*, S. 985–990.

Dean, T. und Kanazawa, K. (1989b). A model for reasoning about persistence and causation. *Computational Intelligence*, *5*(3), 142–150.

Dean, T., Kanazawa, K. und Shewchuk, J. (1990). Prediction, observation and estimation in planning and control. In *5th IEEE International Symposium on Intelligent Control*, Band 2, S. 645–650.

Dean, T. und Wellman, M. P. (1991). *Planning and Control*. Morgan Kaufmann.

Dearden, R., Friedman, N. und Andre, D. (1999). Model-based Bayesian exploration. In *UAI-99*.

Dearden, R., Friedman, N. und Russell, S. J. (1998). Bayesian q-learning. In *AAAI-98*.

Debevec, P., Taylor, C. und Malik, J. (1996). Modeling and rendering architecture from photographs: A hybrid geometry-and image-based approach. In *Proc. 23rd Annual Conference on Computer Graphics (SIGGRAPH)*, S. 11–20.

Debreu, G. (1960). Topological methods in cardinal utility theory. In Arrow, K. J., Karlin, S. und Suppes, P. (Hrsg.), *Mathematical Methods in the Social Sciences, 1959*. Stanford University Press.

Dechter, R. (1990a). Enhancement schemes for constraint processing: Backjumping, learning and cutset decomposition. *AIJ*, *41*, 273–312.

Dechter, R. (1990b). On the expressiveness of networks with hidden variables. In *AAAI-90*, S. 379–385.

Dechter, R. (1992). Constraint networks. In Shapiro, S. (Hrsg.), *Encyclopedia of Artificial Intelligence* (2. Auflage)., S. 276–285. Wiley and Sons.

Dechter, R. (1999). Bucket elimination: A unifying framework for reasoning. *AIJ*, *113*, 41–85.

Dechter, R. und Pearl, J. (1985). Generalized best-first search strategies and the optimality of A*. *JACM*, *32*(3), 505–536.

Dechter, R. und Pearl, J. (1987). Network-based heuristics for constraint-satisfaction problems. *AIJ*, *34*(1), 1–38.

Dechter, R. und Pearl, J. (1989). Tree clustering for constraint networks. *AIJ*, *38*(3), 353–366.

Dechter, R. (2003). *Constraint Processing*. Morgan Kaufmann.

Dechter, R. und Frost, D. (2002). Backjump-based backtracking for constraint satisfaction problems. *AIJ*, *136*(2), 147–188.

Dechter, R. und Mateescu, R. (2007). AND/OR search spaces for graphical models. *AIJ*, *171*(2–3), 73–106.

DeCoste, D. und Schölkopf, B. (2002). Training invariant support vector machines. *Machine Learning*, *46*(1), 161–190.

Dedekind, R. (1888). *Was sind und was sollen die Zahlen*. Braunschweig, Germany.

Deerwester, S. C., Dumais, S. T., Landauer, T. K., Furnas, G. W. und Harshman, R. A. (1990). Indexing by latent semantic analysis. *J. American Society for Information Science*, *41*(6), 391–407.

DeGroot, M. H. (1970). *Optimal Statistical Decisions*. McGraw-Hill.

DeGroot, M. H. und Schervish, M. J. (2001). *Probability and Statistics* (3. Auflage). Addison Wesley.

DeJong, G. (1981). Generalizations based on explanations. In *IJCAI-81*, S. 67–69.

DeJong, G. (1982). An overview of the FRUMP system. In Lehnert, W. und Ringle, M. (Hrsg.), *Strategies for Natural Language Processing*, S. 149–176. Lawrence Erlbaum.

DeJong, G. und Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning*, *1*, 145–176.

Del Moral, P., Doucet, A. und Jasra, A. (2006). Sequential Monte Carlo samplers. *J. Royal Statistical Society, Series B*, *68*(3), 411–436.

Del Moral, P. (2004). *Feynman–Kac Formulae, Genealogical and Interacting Particle Systems with Applications*. Springer-Verlag.

Delgrande, J. und Schaub, T. (2003). On the relation between Reiter's default logic and its (major) variants. In *Seventh European Conference on Symbolic and Quantitative Approaches to Reasoning with Uncertainty*, S. 452–463.

Dempster, A. P. (1968). A generalization of Bayesian inference. *J. Royal Statistical Society*, *30 (Series B)*, 205–247.

Dempster, A. P., Laird, N. und Rubin, D. (1977). Maximum likelihood from incomplete data via the EM algorithm. *J. Royal Statistical Society, 39 (Series B)*, 1–38.

Deng, X. und Papadimitriou, C. H. (1990). Exploring an unknown graph. In *FOCS-90*, S. 355–361.

Denis, F. (2001). Learning regular languages from simple positive examples. *Machine Learning*, *44*(1/2), 37–66.

Dennett, D. C. (1984). Cognitive wheels: the frame problem of AI. In Hookway, C. (Hrsg.), *Minds, Machines, and Evolution: Philosophical Studies*, S. 129–151. Cambridge University Press.

Dennett, D. C. (1991). *Consciousness Explained*. Penguin Press.

Denney, E., Fischer, B. und Schumann, J. (2006). An empirical evaluation of automated theorem provers in software certification. *Int. J. AI Tools*, *15*(1), 81–107.

Descartes, R. (1637). Discourse on method. In Cottingham, J., Stoothoff, R. und Murdoch, D. (Hrsg.), *The Philosophical Writings of Descartes*, Band I. Cambridge University Press, Cambridge, UK.

Descartes, R. (1641). Meditations on first philosophy. In Cottingham, J., Stoothoff, R. und Murdoch, D. (Hrsg.), *The Philosophical Writings of Descartes*, Band II. Cambridge University Press, Cambridge, UK.

Descotte, Y. und Latombe, J.-C. (1985). Making compromises among antagonist constraints in a planner. *AIJ*, *27*, 183–217.

Detwarasiti, A. und Shachter, R. D. (2005). Influence diagrams for team decision analysis. *Decision Analysis*, *2*(4), 207–228.

Devroye, L. (1987). *A course in density estimation*. Birkhauser.

Dickmanns, E. D. und Zapp, A. (1987). Autonomous high speed road vehicle guidance by computer vision. In *Automatic Control – World Congress, 1987: Selected Papers from the 10th Triennial World Congress of the International Federation of Automatic Control*, S. 221–226.

Dietterich, T. (1990). Machine learning. *Annual Review of Computer Science*, *4*, 255–306.

Dietterich, T. (2000). Hierarchical reinforcement learning with the MAXQ value function decomposition. *JAIR*, *13*, 227–303.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*, 269–271.

Dijkstra, E. W. (1984). The threats to computing science. In *ACM South Central Regional Conference*.

Dillenburg, J. F. und Nelson, P. C. (1994). Perimeter search. *AIJ*, *65*(1), 165–178.

Dinh, H., Russell, A. und Su, Y. (2007). On the value of good advice: The complexity of A* with accurate heuristics. In *AAAI-07*.

Dissanayake, G., Newman, P., Clark, S., Durrant-Whyte, H. und Csorba, M. (2001). A solution to the simultaneous localisation and map building (SLAM) problem. *IEEE Transactions on Robotics and Automation*, *17*(3), 229–241.

Do, M. B. und Kambhampati, S. (2001). Sapa: A domain-independent heuristic metric temporal planner. In *ECP-01*.

Do, M. B. und Kambhampati, S. (2003). Planning as constraint satisfaction: solving the planning graph by compiling it into CSP. *AIJ*, *132*(2), 151–182.

Doctorow, C. (2001). Metacrap: Putting the torch to seven strawmen of the meta-utopia. *www.well.com/~doctorow/metacrap.htm*.

Domingos, P. und Pazzani, M. (1997). On the optimality of the simple Bayesian classifier under zero–one loss. *Machine Learning*, *29*, 103–30.

Domingos, P. und Richardson, M. (2004). Markov logic: A unifying framework for statistical relational learning. In *Proc. ICML-04 Workshop on Statistical Relational Learning*.

Donninger, C. und Lorenz, U. (2004). The chess monster hydra. In *Proc. 14th International Conference on Field-Programmable Logic and Applications*, S. 927–932.

Doorenbos, R. (1994). Combining left and right unlinking for matching a large number of learned rules. In *AAAI-94*.

Doran, J. und Michie, D. (1966). Experiments with the graph traverser program. *Proc. Royal Society of London*, *294, Series A*, 235–259.

Dorf, R. C. und Bishop, R. H. (2004). *Modern Control Systems* (10. Auflage). Prentice-Hall.

Doucet, A. (1997). *Monte Carlo methods for Bayesian estimation of hidden Markov models: Application to radiation signals*. Ph.D. thesis, Université de Paris-Sud.

Doucet, A., de Freitas, N. und Gordon, N. (2001). *Sequential Monte Carlo Methods in Practice*. Springer-Verlag.

Doucet, A., de Freitas, N., Murphy, K. und Russell, S. J. (2000). Rao-blackwellised particle filtering for dynamic bayesian networks. In *UAI-00*.

Dowling, W. F. und Gallier, J. H. (1984). Linear-time algorithms for testing the satisfiability of propositional Horn formulas. *J. Logic Programming*, *1*, 267–284.

Dowty, D., Wall, R. und Peters, S. (1991). *Introduction to Montague Semantics*. D. Reidel.

Doyle, J. (1979). A truth maintenance system. *AIJ*, *12*(3), 231–272.

Doyle, J. (1983). What is rational psychology? Toward a modern mental philosophy. *AIMag*, *4*(3), 50–53.

Doyle, J. und Patil, R. (1991). Two theses of knowledge representation: Language restrictions, taxonomic classification, and the utility of representation services. *AIJ*, *48*(3), 261–297.

Drabble, B. (1990). Mission scheduling for spacecraft: Diaries of T-SCHED. In *Expert Planning Systems*, S. 76–81. Institute of Electrical Engineers.

Dredze, M., Crammer, K. und Pereira, F. (2008). Confidence-weighted linear classification. In *ICML-08*, S. 264–271.

Dreyfus, H. L. (1972). *What Computers Can't Do: A Critique of Artificial Reason*. Harper and Row.

Dreyfus, H. L. (1992). *What Computers Still Can't Do: A Critique of Artificial Reason*. MIT Press.

Dreyfus, H. L. und Dreyfus, S. E. (1986). *Mind over Machine: The Power of Human Intuition and Expertise in the Era of the Computer*. Blackwell.

Dreyfus, S. E. (1969). An appraisal of some shortest-paths algorithms. *Operations Research*, *17*, 395–412.

Dubois, D. und Prade, H. (1994). A survey of belief revision and updating rules in various uncertainty models. *Int. J. Intelligent Systems*, *9*(1), 61–100.

Duda, R. O., Gaschnig, J. und Hart, P. E. (1979). Model design in the Prospector consultant system for mineral exploration. In Michie, D. (Hrsg.), *Expert Systems in the Microelectronic Age*, S. 153–167. Edinburgh University Press.

Duda, R. O. und Hart, P. E. (1973). *Pattern classification and scene analysis*. Wiley.

Duda, R. O., Hart, P. E. und Stork, D. G. (2001). *Pattern Classification* (2. Auflage). Wiley.

Dudek, G. und Jenkin, M. (2000). *Computational Principles of Mobile Robotics*. Cambridge University Press.

Duffy, D. (1991). *Principles of Automated Theorem Proving*. John Wiley & Sons.

Dunn, H. L. (1946). Record linkage. *Am. J. Public Health*, *36*(12), 1412–1416.

Durfee, E. H. und Lesser, V. R. (1989). Negotiating task decomposition and allocation using partial global planning. In Huhns, M. und Gasser, L. (Hrsg.), *Distributed AI*, Band 2. Morgan Kaufmann.

Durme, B. V. und Pasca, M. (2008). Finding cars, goddesses and enzymes: Parametrizable acquisition of labeled instances for open-domain information extraction. In *AAAI-08*, S. 1243–1248.

Dyer, M. (1983). *In-Depth Understanding*. MIT Press.

Dyson, G. (1998). *Darwin among the machines: the evolution of global intelligence*. Perseus Books.

Džeroski, S., Muggleton, S. H. und Russell, S. J. (1992). PAC-learnability of determinate logic programs. In *COLT-92*, S. 128–135.

Earley, J. (1970). An efficient context-free parsing algorithm. *CACM*, *13*(2), 94–102.

Edelkamp, S. (2009). Scaling search with symbolic pattern databases. In *Model Checking and Artificial Intelligence (MOCHART)*, S. 49–65.

Edmonds, J. (1965). Paths, trees, and flowers. *Canadian Journal of Mathematics*, *17*, 449–467.

Edwards, P. (Hrsg.). (1967). *The Encyclopedia of Philosophy*. Macmillan.

Een, N. und Sörensson, N. (2003). An extensible SAT-solver. In Giunchiglia, E. und Tacchella, A. (Hrsg.), *Theory and Applications of Satisfiability Testing: 6th International Conference (SAT 2003)*. Springer-Verlag.

Eiter, T., Leone, N., Mateis, C., Pfeifer, G. und Scarcello, F. (1998). The KR system dlv: Progress report, comparisons and benchmarks. In *KR-98*, S. 406–417.

Elio, R. (Hrsg.). (2002). *Common Sense, Reasoning, and Rationality*. Oxford University Press.

Elkan, C. (1993). The paradoxical success of fuzzy logic. In *AAAI-93*, S. 698–703.

Elkan, C. (1997). Boosting and naive Bayesian learning. Tech. rep., Department of Computer Science and Engineering, University of California, San Diego.

Ellsberg, D. (1962). *Risk, Ambiguity, and Decision*. Ph.D. thesis, Harvard University.

Elman, J., Bates, E., Johnson, M., Karmiloff-Smith, A., Parisi, D. und Plunkett, K. (1997). *Rethinking Innateness*. MIT Press.

Empson, W. (1953). *Seven Types of Ambiguity*. New Directions.

Enderton, H. B. (1972). *A Mathematical Introduction to Logic*. Academic Press.

Epstein, R., Roberts, G. und Beber, G. (Hrsg.). (2008). *Parsing the Turing Test*. Springer.

Erdmann, M. A. und Mason, M. (1988). An exploration of sensorless manipulation. *IEEE Journal of Robotics and Automation*, *4*(4), 369–379.

Ernst, H. A. (1961). *MH-1, a Computer-Operated Mechanical Hand*. Ph.D. thesis, Massachusetts Institute of Technology.

Ernst, M., Millstein, T. und Weld, D. S. (1997). Automatic SAT-compilation of planning problems. In *IJCAI-97*, S. 1169–1176.

Erol, K., Hendler, J. und Nau, D. S. (1994). HTN planning: Complexity and expressivity. In *AAAI-94*, S. 1123–1128.

Erol, K., Hendler, J. und Nau, D. S. (1996). Complexity results for HTN planning. *AIJ*, *18*(1), 69–93.

Etzioni, A. (2004). *From Empire to Community: A New Approach to International Relation*. Palgrave Macmillan.

Etzioni, O. (1989). Tractable decision-analytic control. In *Proc. First International Conference on Knowledge Representation and Reasoning*, S. 114–125.

Etzioni, O., Banko, M., Soderland, S. und Weld, D. S. (2008). Open information extraction from the web. *CACM*, *51*(12).

Etzioni, O., Hanks, S., Weld, D. S., Draper, D., Lesh, N. und Williamson, M. (1992). An approach to planning with incomplete information. In *KR-92*.

Etzioni, O. und Weld, D. S. (1994). A softbot-based interface to the Internet. *CACM*, *37*(7), 72–76.

Etzioni, O., Banko, M. und Cafarella, M. J. (2006). Machine reading. In *AAAI-06*.

Etzioni, O., Cafarella, M. J., Downey, D., Popescu, A.-M., Shaked, T., Soderland, S., Weld, D. S. und Yates, A. (2005). Unsupervised named-entity extraction from the web: An experimental study. *AIJ*, *165*(1), 91–134.

Evans, T. G. (1968). A program for the solution of a class of geometric-analogy intelligence-test questions. In Minsky, M. L. (Hrsg.), *Semantic Information Processing*, S. 271–353. MIT Press.

Fagin, R., Halpern, J. Y., Moses, Y. und Vardi, M. Y. (1995). *Reasoning about Knowledge*. MIT Press.

Fahlman, S. E. (1974). A planning system for robot construction tasks. *AIJ*, *5*(1), 1–49.

Faugeras, O. (1993). *Three-Dimensional Computer Vision: A Geometric Viewpoint*. MIT Press.

Faugeras, O., Luong, Q.-T. und Papadopoulo, T. (2001). *The Geometry of Multiple Images*. MIT Press.

Fearing, R. S. und Hollerbach, J. M. (1985). Basic solid mechanics for tactile sensing. *Int. J. Robotics Research*, *4*(3), 40–54.

Featherstone, R. (1987). *Robot Dynamics Algorithms*. Kluwer Academic Publishers.

Feigenbaum, E. A. (1961). The simulation of verbal learning behavior. *Proc. Western Joint Computer Conference*, *19*, 121–131.

Feigenbaum, E. A., Buchanan, B. G. und Lederberg, J. (1971). On generality and problem solving: A case study using the DENDRAL program. In Meltzer, B. und Michie, D. (Hrsg.), *Machine Intelligence 6*, S. 165–190. Edinburgh University Press.

Feldman, J. und Sproull, R. F. (1977). Decision theory and artificial intelligence II: The hungry monkey. Technical report, Computer Science Department, University of Rochester.

Feldman, J. und Yakimovsky, Y. (1974). Decision theory and artificial intelligence I: Semantics-based region analyzer. *AIJ*, *5*(4), 349–371.

Fellbaum, C. (2001). *Wordnet: An Electronic Lexical Database*. MIT Press.

Fellegi, I. und Sunter, A. (1969). A theory for record linkage". *JASA*, *64*, 1183–1210.

Felner, A., Korf, R. E. und Hanan, S. (2004). Additive pattern database heuristics. *JAIR*, *22*, 279–318.

Felner, A., Korf, R. E., Meshulam, R. und Holte, R. (2007). Compressed pattern databases. *JAIR*, *30*, 213–247.

Felzenszwalb, P. und Huttenlocher, D. (2000). Efficient matching of pictorial structures. In *CVPR*.

Felzenszwalb, P. und McAllester, D. A. (2007). The generalized A* architecture. *JAIR*.

Ferguson, T. (1992). Mate with knight and bishop in kriegspiel. *Theoretical Computer Science*, *96*(2), 389–403.

Ferguson, T. (1995). Mate with the two bishops in kriegspiel. www.math.ucla.edu/~tom/papers.

Ferguson, T. (1973). Bayesian analysis of some nonparametric problems. *Annals of Statistics*, *1*(2), 209–230.

Ferraris, P. und Giunchiglia, E. (2000). Planning as satisability in nondeterministic domains. In *AAAI-00*, S. 748–753.

Ferriss, T. (2007). *The 4-Hour Workweek*. Crown.

Fikes, R. E., Hart, P. E. und Nilsson, N. J. (1972). Learning and executing generalized robot plans. *AIJ*, *3*(4), 251–288.

Fikes, R. E. und Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *AIJ*, *2*(3–4), 189–208.

Fikes, R. E. und Nilsson, N. J. (1993). STRIPS, a retrospective. *AIJ*, *59*(1–2), 227–232.

Fine, S., Singer, Y. und Tishby, N. (1998). The hierarchical hidden markov model: Analysis and applications. *Machine Learning*, *32*(41–62).

Finney, D. J. (1947). *Probit analysis: A statistical treatment of the sigmoid response curve*. Cambridge University Press.

Firth, J. (1957). *Papers in Linguistics*. Oxford University Press.

Fisher, R. A. (1922). On the mathematical foundations of theoretical statistics. *Philosophical Transactions of the Royal Society of London, Series A 222*, 309–368.

Fix, E. und Hodges, J. L. (1951). Discriminatory analysis – Nonparametric discrimination: Consistency properties. Tech. rep. 21-49-004, USAF School of Aviation Medicine.

Floreano, D., Zufferey, J. C., Srinivasan, M. V. und Ellington, C. (2009). *Flying Insects and Robots*. Springer.

Fogel, D. B. (2000). *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press.

Fogel, L. J., Owens, A. J. und Walsh, M. J. (1966). *Artificial Intelligence through Simulated Evolution*. Wiley.

Foo, N. (2001). Why engineering models do not have a frame problem. In *Discrete event modeling and simulation technologies: a tapestry of systems and AI-based theories and methodologies*. Springer.

Forbes, J. (2002). *Learning Optimal Control for Autonomous Vehicles*. Ph.D. thesis, University of California.

Forbus, K. D. (1985). Qualitative process theory. In Bobrow, D. (Hrsg.), *Qualitative Reasoning About Physical Systems*, S. 85–186. MIT Press.

Forbus, K. D. und de Kleer, J. (1993). *Building Problem Solvers*. MIT Press.

Ford, K. M. und Hayes, P. J. (1995). Turing Test considered harmful. In *IJCAI-95*, S. 972–977.

Forestier, J.-P. und Varaiya, P. (1978). Multilayer control of large Markov chains. *IEEE Transactions on Automatic Control*, *23*(2), 298–304.

Forgy, C. (1981). OPS5 user's manual. Technical report CMU-CS-81-135, Computer Science Department, Carnegie-Mellon University.

Forgy, C. (1982). A fast algorithm for the many patterns/many objects match problem. *AIJ*, *19*(1), 17–37.

Forsyth, D. und Ponce, J. (2002). *Computer Vision: A Modern Approach*. Prentice Hall.

Fourier, J. (1827). Analyse des travaux de l'Académie Royale des Sciences, pendant l'année 1824; partie mathématique. *Histoire de l'Académie Royale des Sciences de France*, *7*, xlvii–lv.

Fox, C. und Tversky, A. (1995). Ambiguity aversion and comparative ignorance. *Quarterly Journal of Economics*, *110*(3), 585–603.

Fox, D., Burgard, W., Dellaert, F. und Thrun, S. (1999). Monte carlo localization: Efficient position estimation for mobile robots. In *AAAI-99*.

Fox, M. S. (1990). Constraint-guided scheduling: A short history of research at CMU. *Computers in Industry*, *14*(1–3), 79–88.

Fox, M. S., Allen, B. und Strohm, G. (1982). Job shop scheduling: An investigation in constraint-directed reasoning. In *AAAI-82*, S. 155–158.

Fox, M. S. und Long, D. (1998). The automatic inference of state invariants in TIM. *JAIR*, *9*, 367–421.

Franco, J. und Paull, M. (1983). Probabilistic analysis of the Davis Putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, *5*, 77–87.

Frank, I., Basin, D. A. und Matsubara, H. (1998). Finding optimal strategies for imperfect information games. In *AAAI-98*, S. 500–507.

Frank, R. H. und Cook, P. J. (1996). *The Winner-Take-All Society*. Penguin.

Franz, A. (1996). *Automatic Ambiguity resolution in Natural Language Processing: An Empirical Approach*. Springer.

Franz, A. und Brants, T. (2006). All our n-gram are belong to you. Blog posting.

Frege, G. (1879). *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Halle, Berlin. Englische Übersetzung erscheint in van Heijenoort (1967).

Freitag, D. und McCallum, A. (2000). Information extraction with hmm structures learned by stochastic optimization. In *AAAI-00*.

Freuder, E. C. (1978). Synthesizing constraint expressions. *CACM*, *21*(11), 958–966.

Freuder, E. C. (1982). A sufficient condition for backtrack-free search. *JACM*, *29*(1), 24–32.

Freuder, E. C. (1985). A sufficient condition for backtrack-bounded search. *JACM*, *32*(4), 755–761.

Freuder, E. C. und Mackworth, A. K. (Hrsg.). (1994). *Constraint-based reasoning*. MIT Press.

Freund, Y. und Schapire, R. E. (1996). Experiments with a new boosting algorithm. In *ICML-96*.

Freund, Y. und Schapire, R. E. (1999). Large margin classification using the perceptron algorithm. *Machine Learning*, *37*(3), 277–296.

Friedberg, R. M. (1958). A learning machine: Part I. *IBM Journal of Research and Development*, *2*, 2–13.

Friedberg, R. M., Dunham, B. und North, T. (1959). A learning machine: Part II. *IBM Journal of Research and Development*, *3*(3), 282–287.

Friedgut, E. (1999). Necessary and sufficient conditions for sharp thresholds of graph properties, and the k-SAT problem. *J. American Mathematical Society*, *12*, 1017–1054.

Friedman, G. J. (1959). Digital simulation of an evolutionary process. *General Systems Yearbook*, *4*, 171–184.

Friedman, J., Hastie, T. und Tibshirani, R. (2000). Additive logistic regression: A statistical view of boosting. *Annals of Statistics*, *28*(2), 337–374.

Friedman, N. (1998). The Bayesian structural EM algorithm. In *UAI-98*.

Friedman, N. und Goldszmidt, M. (1996). Learning Bayesian networks with local structure. In *UAI-96*, S. 252–262.

Friedman, N. und Koller, D. (2003). Being Bayesian about Bayesian network structure: A Bayesian approach to structure discovery in Bayesian networks. *Machine Learning*, *50*, 95–125.

Friedman, N., Murphy, K. und Russell, S. J. (1998). Learning the structure of dynamic probabilistic networks. In *UAI-98*.

Friedman, N. (2004). Inferring cellular networks using probabilistic graphical models. *Science*, *303*(5659), 799–805.

Fruhwirth, T. und Abdennadher, S. (2003). *Essentials of constraint programming*. Cambridge University Press.

Fuchs, J. J., Gasquet, A., Olalainty, B. und Currie, K. W. (1990). PlanERS-1: An expert planning system for generating spacecraft mission plans. In *First International Conference on Expert Planning Systems*, S. 70–75. Institute of Electrical Engineers.

Fudenberg, D. und Tirole, J. (1991). *Game theory*. MIT Press.

Fukunaga, A. S., Rabideau, G., Chien, S. und Yan, D. (1997). ASPEN: A framework for automated planning and scheduling of spacecraft control and operations. In *Proc. International Symposium on AI, Robotics and Automation in Space*, S. 181–187.

Fung, R. und Chang, K. C. (1989). Weighting and integrating evidence for stochastic simulation in Bayesian networks. In *UAI-98*, S. 209–220.

Gaddum, J. H. (1933). Reports on biological standard III: Methods of biological assay depending on a quantal response. Special report series of the medical research council 183, Medical Research Council.

Gaifman, H. (1964). Concerning measures in first order calculi. *Israel Journal of Mathematics*, *2*, 1–18.

Gallaire, H. und Minker, J. (Hrsg.). (1978). *Logic and Databases*. Plenum.

Gallier, J. H. (1986). *Logic for Computer Science: Foundations of Automatic Theorem Proving*. Harper and Row.

Gamba, A., Gamberini, L., Palmieri, G. und Sanna, R. (1961). Further experiments with PAPA. *Nuovo Cimento Supplemento*, *20*(2), 221–231.

Garding, J. (1992). Shape from texture for smooth curved surfaces in perspective projection. *J. Mathematical Imaging and Vision*, *2*(4), 327–350.

Gardner, M. (1968). *Logic Machines, Diagrams and Boolean Algebra*. Dover.

Garey, M. R. und Johnson, D. S. (1979). *Computers and Intractability*. W. H. Freeman.

Gaschnig, J. (1977). A general backtrack algorithm that eliminates most redundant tests. In *IJCAI-77*, p. 457.

Gaschnig, J. (1979). Performance measurement and analysis of certain search algorithms. Technical report CMU-CS-79-124, Computer Science Department, Carnegie-Mellon University.

Gasser, R. (1995). *Efficiently harnessing computational resources for exhaustive search*. Ph.D. thesis, ETH Zürich.

Gasser, R. (1998). Solving nine men's morris. In Nowakowski, R. (Hrsg.), *Games of No Chance*. Cambridge University Press.

Gat, E. (1998). Three-layered architectures. In Kortenkamp, D., Bonasso, R. P. und Murphy, R. (Hrsg.), *AI-based Mobile Robots: Case Studies of Successful Robot Systems*, S. 195–210. MIT Press.

Gauss, C. F. (1809). *Theoria Motus Corporum Coelestium in Sectionibus Conicis Solem Ambientium*. Sumtibus F. Perthes et I. H. Besser, Hamburg.

Gauss, C. F. (1829). Beiträge zur Theorie der algebraischen Gleichungen. Collected in *Werke, Band 3*, S. 71–102. K. Gesellschaft Wissenschaft, Göttingen, Germany, 1876.

Gawande, A. (2002). *Complications: A Surgeon's Notes on an Imperfect Science*. Metropolitan Books.

Geiger, D., Verma, T. und Pearl, J. (1990). Identifying independence in Bayesian networks. *Networks*, *20*(5), 507–534.

Geisel, T. (1955). *On Beyond Zebra*. Random House.

Gelb, A. (1974). *Applied Optimal Estimation*. MIT Press.

Gelernter, H. (1959). Realization of a geometry-theorem proving machine. In *Proc. an International Conference on Information Processing*, S. 273–282. UNESCO House.

Gelfond, M. und Lifschitz, V. (1988). Compiling circumscriptive theories into logic programs. In *Non-Monotonic Reasoning: 2nd International Workshop Proceedings*, S. 74–99.

Gelfond, M. (2008). Answer sets. In van Harmelan, F., Lifschitz, V. und Porter, B. (Hrsg.), *Handbook of Knowledge Representation*, S. 285–316. Elsevier.

Gelly, S. und Silver, D. (2008). Achieving master level play in 9 x 9 computer go. In *AAAI-08*, S. 1537–1540.

Gelman, A., Carlin, J. B., Stern, H. S. und Rubin, D. (1995). *Bayesian Data Analysis*. Chapman & Hall.

Geman, S. und Geman, D. (1984). Stochastic relaxation, Gibbs distributions, and Bayesian restoration of images. *PAMI*, *6*(6), 721–741.

Genesereth, M. R. (1984). The use of design descriptions in automated diagnosis. *AIJ*, *24*(1–3), 411–436.

Genesereth, M. R. und Nilsson, N. J. (1987). *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann.

Genesereth, M. R. und Nourbakhsh, I. (1993). Time-saving tips for problem solving with incomplete information. In *AAAI-93*, S. 724–730.

Genesereth, M. R. und Smith, D. E. (1981). Meta-level architecture. Memo HPP-81-6, Computer Science Department, Stanford University.

Gent, I., Petrie, K. und Puget, J.-F. (2006). Symmetry in constraint programming. In Rossi, F., van Beek, P. und Walsh, T. (Hrsg.), *Handbook of Constraint Programming*. Elsevier.

Gentner, D. (1983). Structure mapping: A theoretical framework for analogy. *Cognitive Science*, *7*, 155–170.

Gentner, D. und Goldin-Meadow, S. (Hrsg.). (2003). *Language in mind: Advances in the study of language and though*. MIT Press.

Gerevini, A. und Long, D. (2005). Plan constraints and preferences in PDDL3. Tech. rep., Dept. of Electronics for Automation, University of Brescia, Italy.

Gerevini, A. und Serina, I. (2002). LPG: A planner based on planning graphs with action costs. In *ICAPS-02*, S. 281–290.

Gerevini, A. und Serina, I. (2003). Planning as propositional CSP: from walksat to local search for action graphs. *Constraints*, *8*, 389–413.

Gershwin, G. (1937). Let's call the whole thing off. Song.

Getoor, L. und Taskar, B. (Hrsg.). (2007). *Introduction to Statistical Relational Learning*. MIT Press.

Ghahramani, Z. und Jordan, M. I. (1997). Factorial hidden Markov models. *Machine Learning*, *29*, 245–274.

Ghahramani, Z. (1998). Learning dynamic bayesian networks. In *Adaptive Processing of Sequences and Data Structures*, S. 168–197.

Ghahramani, Z. (2005). Tutorial on nonparametric Bayesian methods. Tutorial presentation at the UAI Conference.

Ghallab, M., Howe, A., Knoblock, C. A. und McDermott, D. (1998). PDDL – The planning domain definition language. Tech. rep. DCS TR-1165, Yale Center for Computational Vision and Control.

Ghallab, M. und Laruelle, H. (1994). Representation and control in IxTeT, a temporal planner. In *AIPS-94*, S. 61–67.

Ghallab, M., Nau, D. S. und Traverso, P. (2004). *Automated Planning: Theory and practice*. Morgan Kaufmann.

Gibbs, R. W. (2006). Metaphor interpretation as embodied simulation. *Mind*, *21*(3), 434–458.

Gibson, J. J. (1950). *The Perception of the Visual World*. Houghton Mifflin.

Gibson, J. J. (1979). *The Ecological Approach to Visual Perception*. Houghton Mifflin.

Gilks, W. R., Richardson, S. und Spiegelhalter, D. J. (Hrsg.). (1996). *Markov chain Monte Carlo in practice*. Chapman and Hall.

Gilks, W. R., Thomas, A. und Spiegelhalter, D. J. (1994). A language and program for complex Bayesian modelling. *The Statistician*, *43*, 169–178.

Gilmore, P. C. (1960). A proof method for quantification theory: Its justification and realization. *IBM Journal of Research and Development*, *4*, 28–35.

Ginsberg, M. L. (1993). *Essentials of Artificial Intelligence*. Morgan Kaufmann.

Ginsberg, M. L. (1999). GIB: Steps toward an expert-level bridge-playing program. In *IJCAI-99*, S. 584–589.

Ginsberg, M. L., Frank, M., Halpin, M. P. und Torrance, M. C. (1990). Search lessons learned from crossword puzzles. In *AAAI-90*, Band 1, S. 210–215.

Ginsberg, M. L. (2001). GIB: Imperfect infoormation in a computationally challenging game. *JAIR*, *14*, 303–358.

Gionis, A., Indyk, P. und Motwani, R. (1999). Similarity search in high dimensions vis hashing. In *Proc. 25th Very Large Database (VLDB) Conference*.

Gittins, J. C. (1989). *Multi-Armed Bandit Allocation Indices*. Wiley.

Glanc, A. (1978). On the etymology of the word "robot". *SIGART Newsletter*, *67*, 12.

Glover, F. und Laguna, M. (Hrsg.). (1997). *Tabu search*. Kluwer.

Gödel, K. (1930). *Über die Vollständigkeit des Logikkalküls*. Ph.D. thesis, University of Vienna.

Gödel, K. (1931). Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, *38*, 173–198.

Goebel, J., Volk, K., Walker, H. und Gerbault, F. (1989). Automatic classification of spectra from the infrared astronomical satellite (IRAS). *Astronomy and Astrophysics*, *222*, L5–L8.

Goertzel, B. und Pennachin, C. (2007). *Artificial General Intelligence*. Springer.

Gold, B. und Morgan, N. (2000). *Speech and Audio Signal Processing*. Wiley.

Gold, E. M. (1967). Language identification in the limit. *Information and Control*, *10*, 447–474.

Goldberg, A. V., Kaplan, H. und Werneck, R. F. (2006). Reach for a*: Efficient point-to-point shortest path algorithms. In *Workshop on algorithm engineering and experiments*, S. 129–143.

Goldman, R. und Boddy, M. (1996). Expressive planning and explicit knowledge. In *AIPS-96*, S. 110–117.

Goldszmidt, M. und Pearl, J. (1996). Qualitative probabilities for default reasoning, belief revision, and causal modeling. *AIJ*, *84*(1–2), 57–112.

Golomb, S. und Baumert, L. (1965). Backtrack programming. *JACM*, *14*, 516–524.

Golub, G., Heath, M. und Wahba, G. (1979). Generalized cross-validation as a method for choosing a good ridge parameter. *Technometrics*, *21*(2).

Gomes, C., Selman, B., Crato, N. und Kautz, H. (2000). Heavy-tailed phenomena in satisfiability and constrain processing. *JAR*, *24*, 67–100.

Gomes, C., Kautz, H., Sabharwal, A. und Selman, B. (2008). Satisfiability solvers. In van Harmelen, F., Lifschitz, V. und Porter, B. (Hrsg.), *Handbook of Knowledge Representation*. Elsevier.

Gomes, C. und Selman, B. (2001). Algorithm portfolios. *AIJ*, *126*, 43–62.

Gomes, C., Selman, B. und Kautz, H. (1998). Boosting combinatorial search through randomization. In *AAAI-98*, S. 431–437.

Gonthier, G. (2008). Formal proof–The four-color theorem. *Notices of the AMS*, *55*(11), 1382–1393.

Good, I. J. (1961). A causal calculus. *British Journal of the Philosophy of Science*, *11*, 305–318.

Good, I. J. (1965). Speculations concerning the first ultraintelligent machine. In Alt, F. L. und Rubinoff, M. (Hrsg.), *Advances in Computers*, Band 6, S. 31–88. Academic Press.

Good, I. J. (1983). *Good Thinking: The Foundations of Probability and Its Applications*. University of Minnesota Press.

Goodman, D. und Keene, R. (1997). *Man versus Machine: Kasparov versus Deep Blue*. H3 Publications.

Goodman, J. (2001). A bit of progress in language modeling. Tech. rep. MSR-TR-2001-72, Microsoft Research.

Goodman, J. und Heckerman, D. (2004). Fighting spam with statistics. *Significance, the Magazine of the Royal Statistical Society*, *1*, 69–72.

Goodman, N. (1954). *Fact, Fiction and Forecast*. University of London Press.

Goodman, N. (1977). *The Structure of Appearance* (3. Auflage). D. Reidel.

Gopnik, A. und Glymour, C. (2002). Causal maps and bayes nets: A cognitive and computational account of theory-formation. In Caruthers, P., Stich, S. und Siegal, M. (Hrsg.), *The Cognitive Basis of Science*. Cambridge University Press.

Gordon, D. M. (2000). *Ants at Work*. Norton.

Gordon, D. M. (2007). Control without hierarchy. *Nature*, *446*(8), 143.

Gordon, M. J., Milner, A. J. und Wadsworth, C. P. (1979). *Edinburgh LCF*. Springer-Verlag.

Gordon, N. (1994). *Bayesian methods for tracking*. Ph.D. thesis, Imperial College.

Gordon, N., Salmond, D. J. und Smith, A. F. M. (1993). Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEE Proceedings F (Radar and Signal Processing)*, *140*(2), 107–113.

Gordon, S. A. (1994b). A Faster Scrabble Move Generation Algorithm, *Software Practice and Experience*, *24*(2), 219–232.

Gorry, G. A. (1968). Strategies for computer-aided diagnosis. *Mathematical Biosciences*, *2*(3–4), 293–318.

Gorry, G. A., Kassirer, J. P., Essig, A. und Schwartz, W. B. (1973). Decision analysis as the basis for computer-aided management of acute renal failure. *American Journal of Medicine*, *55*, 473–484.

Gottlob, G., Leone, N. und Scarcello, F. (1999a). A comparison of structural CSP decomposition methods. In *IJCAI-99*, S. 394–399.

Gottlob, G., Leone, N. und Scarcello, F. (1999b). Hypertree decompositions and tractable queries. In *PODS-99*, S. 21–32.

Graham, S. L., Harrison, M. A. und Ruzzo, W. L. (1980). An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, *2*(3), 415–462.

Grama, A. und Kumar, V. (1995). A survey of parallel search algorithms for discrete optimization problems. *ORSA Journal of Computing*, *7*(4), 365–385.

Grassmann, H. (1861). *Lehrbuch der Arithmetik*. Th. Chr. Fr. Enslin, Berlin.

Grayson, C. J. (1960). Decisions under uncertainty: Drilling decisions by oil and gas operators. Tech. rep., Division of Research, Harvard Business School.

Green, B., Wolf, A., Chomsky, C. und Laugherty, K. (1961). BASEBALL: An automatic question answerer. In *Proc. Western Joint Computer Conference*, S. 219–224.

Green, C. (1969a). Application of theorem proving to problem solving. In *IJCAI-69*, S. 219–239.

Green, C. (1969b). Theorem-proving by resolution as a basis for question-answering systems. In Meltzer, B., Michie, D. und Swann, M. (Hrsg.), *Machine Intelligence 4*, S. 183–205. Edinburgh University Press.

Green, C. und Raphael, B. (1968). The use of theorem-proving techniques in question-answering systems. In *Proc. 23rd ACM National Conference*.

Greenblatt, R. D., Eastlake, D. E. und Crocker, S. D. (1967). The Greenblatt chess program. In *Proc. Fall Joint Computer Conference*, S. 801–810.

Greiner, R. (1989). Towards a formal analysis of EBL. In *ICML-89*, S. 450–453.

Grinstead, C. und Snell, J. (1997). *Introduction to Probability*. AMS.

Grove, W. und Meehl, P. (1996). Comparative efficiency of informal (subjective, impressionistic) and formal (mechanical, algorithmic) prediction procedures: The clinical statistical controversy. *Psychology, Public Policy, and Law*, *2*, 293–323.

Gruber, T. (2004). Interview of Tom Gruber. *AIS SIGSEMIS Bulletin*, *1*(3).

Gu, J. (1989). *Parallel Algorithms and Architectures for Very Fast AI Search*. Ph.D. thesis, University of Utah.

Guard, J., Oglesby, F., Bennett, J. und Settle, L. (1969). Semi-automated mathematics. *JACM*, *16*, 49–62.

Guestrin, C., Koller, D., Gearhart, C. und Kanodia, N. (2003a). Generalizing plans to new environments in relational MDPs. In *IJCAI-03*.

Guestrin, C., Koller, D., Parr, R. und Venkataraman, S. (2003b). Efficient solution algorithms for factored MDPs. *JAIR*, *19*, 399–468.

Guestrin, C., Lagoudakis, M. G. und Parr, R. (2002). Coordinated reinforcement learning. In *ICML-02*, S. 227–234.

Guibas, L. J., Knuth, D. E. und Sharir, M. (1992). Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, *7*, 381–413. See also *17th Int. Coll. on Automata, Languages and Programming*, 1990, S. 414–431.

Gumperz, J. und Levinson, S. (1996). *Rethinking Linguistic Relativity*. Cambridge University Press.

Guyon, I. und Elisseeff, A. (2003). An introduction to variable and feature selection. *JMLR*, S. 1157–1182.

Hacking, I. (1975). *The Emergence of Probability*. Cambridge University Press.

Haghighi, A. und Klein, D. (2006). Prototype-driven grammar induction. In *COLING-06*.

Hald, A. (1990). *A History of Probability and Statistics and Their Applications before 1750*. Wiley.

Halevy, A. (2007). Dataspaces: A new paradigm for data integration. In *Brazilian Symposium on Databases*.

Halevy, A., Norvig, P. und Pereira, F. (2009). The unreasonable effectiveness of data. *IEEE Intelligent Systems*, *March/April*, 8–12.

Halpern, J. Y. (1990). An analysis of first-order logics of probability. *AIJ*, *46*(3), 311–350.

Halpern, J. Y. (1999). Technical addendum, Cox's theorem revisited. *JAIR*, *11*, 429–435.

Halpern, J. Y. und Weissman, V. (2008). Using first-order logic to reason about policies. *ACM Transactions on Information and System Security*, *11*(4).

Hamming, R. W. (1991). *The Art of Probability for Scientists and Engineers*. Addison-Wesley.

Hammond, K. (1989). *Case-Based Planning: Viewing Planning as a Memory Task*. Academic Press.

Hamscher, W., Console, L. und Kleer, J. D. (1992). *Readings in Model-based Diagnosis*. Morgan Kaufmann.

Han, X. und Boyden, E. (2007). Multiple-color optical activation, silencing, and desynchronization of neural activity, with single-spike temporal resolution. *PLoS One*, *e299*.

Hand, D., Mannila, H. und Smyth, P. (2001). *Principles of Data Mining*. MIT Press.

Handschin, J. E. und Mayne, D. Q. (1969). Monte Carlo techniques to estimate the conditional expectation in multi-stage nonlinear filtering. *Int. J. Control*, *9*(5), 547–559.

Hansen, E. (1998). Solving POMDPs by searching in policy space. In *UAI-98*, S. 211–219.

Hansen, E. und Zilberstein, S. (2001). LAO*: a heuristic search algorithm that finds solutions with loops. *AIJ*, *129*(1–2), 35–62.

Hansen, P. und Jaumard, B. (1990). Algorithms for the maximum satisfiability problem. *Computing*, *44*(4), 279–303.

Hanski, I. und Cambefort, Y. (Hrsg.). (1991). *Dung Beetle Ecology*. Princeton University Press.

Hansson, O. und Mayer, A. (1989). Heuristic search as evidential reasoning. In *UAI 5*.

Hansson, O., Mayer, A. und Yung, M. (1992). Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, *63*(3), 207–227.

Haralick, R. M. und Elliot, G. L. (1980). Increasing tree search efficiency for constraint satisfaction problems. *AIJ*, *14*(3), 263–313.

Hardin, G. (1968). The tragedy of the commons. *Science*, *162*, 1243–1248.

Hardy, G. H. (1940). *A Mathematician's Apology*. Cambridge University Press.

Harman, G. H. (1983). *Change in View: Principles of Reasoning*. MIT Press.

Harris, Z. (1954). Distributional structure. *Word*, *10*(2/3).

Harrison, J. R. und March, J. G. (1984). Decision making and post-decision surprises. *Administrative Science Quarterly*, *29*, 26–42.

Harsanyi, J. (1967). Games with incomplete information played by Bayesian players. *Management Science*, *14*, 159–182.

Hart, P. E., Nilsson, N. J. und Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *SSC-4(2)*, 100–107.

Hart, P. E., Nilsson, N. J. und Raphael, B. (1972). Correction to "A formal basis for the heuristic determination of minimum cost paths". *SIGART Newsletter*, *37*, 28–29.

Hart, T. P. und Edwards, D. J. (1961). The tree prune (TP) algorithm. Artificial intelligence project memo 30, Massachusetts Institute of Technology.

Hartley, H. (1958). Maximum likelihood estimation from incomplete data. *Biometrics*, *14*, 174–194.

Hartley, R. und Zisserman, A. (2000). *Multiple view geometry in computer vision*. Cambridge University Press.

Haslum, P., Botea, A., Helmert, M., Bonet, B. und Koenig, S. (2007). Domain-independent construction of pattern database heuristics for cost-optimal planning. In *AAAI-07*, S. 1007–1012.

Haslum, P. und Geffner, H. (2001). Heuristic planning with time and resources. In *Proc. IJCAI-01 Workshop on Planning with Resources*.

Haslum, P. (2006). Improving heuristics through relaxed search – An analysis of TP4 and HSP*a in the 2004 planning competition. *JAIR*, *25*, 233–267.

Haslum, P., Bonet, B. und Geffner, H. (2005). New admissible heuristics for domain-independent planning. In *AAAI-05*.

Hastie, T. und Tibshirani, R. (1996). Discriminant adaptive nearest neighbor classification and regression. In Touretzky, D. S., Mozer, M. C. und Hasselmo, M. E. (Hrsg.), *NIPS 8*, S. 409–15. MIT Press.

Hastie, T., Tibshirani, R. und Friedman, J. (2001). *The Elements of Statistical Learning: Data Mining, Inference and Prediction* (2. Auflage). Springer-Verlag.

Hastie, T., Tibshirani, R. und Friedman, J. (2009). *The Elements of Statistical Learning: Data Mining, Inference and Prediction* (2. Auflage). Springer-Verlag.

Haugeland, J. (Hrsg.). (1985). *Artificial Intelligence: The Very Idea*. MIT Press.

Hauk, T. (2004). *Search in Trees with Chance Nodes*. Ph.D. thesis, Univ. of Alberta.

Haussler, D. (1989). Learning conjunctive concepts in structural domains. *Machine Learning*, *4*(1), 7–40.

Havelund, K., Lowry, M., Park, S., Pecheur, C., Penix, J., Visser, W. und White, J. L. (2000). Formal analysis of the remote agent before and after flight. In *Proc. 5th NASA Langley Formal Methods Workshop*.

Havenstein, H. (2005). Spring comes to AI winter. *Computer World*.

Hawkins, J. und Blakeslee, S. (2004). *On Intelligence*. Henry Holt and Co.

Hayes, P. J. (1978). The naive physics manifesto. In Michie, D. (Hrsg.), *Expert Systems in the Microelectronic Age*. Edinburgh University Press.

Hayes, P. J. (1979). The logic of frames. In Metzing, D. (Hrsg.), *Frame Conceptions and Text Understanding*, S. 46–61. de Gruyter.

Hayes, P. J. (1985a). Naive physics I: Ontology for liquids. In Hobbs, J. R. und Moore, R. C. (Hrsg.), *Formal Theories of the Commonsense World*, Kap. 3, S. 71–107. Ablex.

Hayes, P. J. (1985b). The second naive physics manifesto. In Hobbs, J. R. und Moore, R. C. (Hrsg.), *Formal Theories of the Commonsense World*, Kap. 1, S. 1–36. Ablex.

Haykin, S. (2008). *Neural Networks: A Comprehensive Foundation*. Prentice Hall.

Hays, J. und Efros, A. A. (2007). Scene completion Using millions of photographs. *ACM Transactions on Graphics (SIGGRAPH)*, *26*(3).

Hearst, M. A. (1992). Automatic acquisition of hyponyms from large text corpora. In *COLING-92*.

Hearst, M. A. (2009). *Search User Interfaces*. Cambridge University Press.

Hebb, D. O. (1949). *The Organization of Behavior*. Wiley.

Heckerman, D. (1986). Probabilistic interpretation for MYCIN's certainty factors. In Kanal, L. N. und Lemmer, J. F. (Hrsg.), *UAI 2*, S. 167–196. Elsevier/North-Holland.

Heckerman, D. (1991). *Probabilistic Similarity Networks*. MIT Press.

Heckerman, D. (1998). A tutorial on learning with Bayesian networks. In Jordan, M. I. (Hrsg.), *Learning in graphical models*. Kluwer.

Heckerman, D., Geiger, D. und Chickering, D. M. (1994). Learning Bayesian networks: The combination of knowledge and statistical data. Technical report MSR-TR-94-09, Microsoft Research.

Heidegger, M. (1927). *Being and Time*. SCM Press.

Heinz, E. A. (2000). *Scalable search in computer chess*. Vieweg.

Held, M. und Karp, R. M. (1970). The traveling salesman problem and minimum spanning trees. *Operations Research*, *18*, 1138–1162.

Helmert, M. (2001). On the complexity of planning in transportation domains. In *ECP-01*.

Helmert, M. (2003). Complexity results for standard benchmark domains in planning. *AIJ*, *143*(2), 219–262.

Helmert, M. (2006). The fast downward planning system. *JAIR*, *26*, 191–246.

Helmert, M. und Richter, S. (2004). Fast downward – Making use of causal dependencies in the problem representation. In *Proc. International Planning Competition at ICAPS*, S. 41–43.

Helmert, M. und Röger, G. (2008). How good is almost perfect? In *AAAI-08*.

Hendler, J., Carbonell, J. G., Lenat, D. B., Mizoguchi, R. und Rosenbloom, P. S. (1995). VERY large knowledge bases – Architecture vs engineering. In *IJCAI-95*, S. 2033–2036.

Henrion, M. (1988). Propagation of uncertainty in Bayesian networks by probabilistic logic sampling. In Lemmer, J. F. und Kanal, L. N. (Hrsg.), *UAI 2*, S. 149–163. Elsevier/North-Holland.

Henzinger, T. A. und Sastry, S. (Hrsg.). (1998). *Hybrid systems: Computation and control*. Springer-Verlag.

Herbrand, J. (1930). *Recherches sur la Théorie de la Démonstration*. Ph.D. thesis, University of Paris.

Hewitt, C. (1969). PLANNER: a language for proving theorems in robots. In *IJCAI-69*, S. 295–301.

Hierholzer, C. (1873). Über die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. *Mathematische Annalen*, *6*, 30–32.

Hilgard, E. R. und Bower, G. H. (1975). *Theories of Learning* (4. Auflage). Prentice-Hall.

Hintikka, J. (1962). *Knowledge and Belief*. Cornell University Press.

Hinton, G. E. und Anderson, J. A. (1981). *Parallel Models of Associative Memory*. Lawrence Erlbaum Associates.

Hinton, G. E. und Nowlan, S. J. (1987). How learning can guide evolution. *Complex Systems*, *1*(3), 495–502.

Hinton, G. E., Osindero, S. und Teh, Y. W. (2006). A fast learning algorithm for deep belief nets. *Neural Computation*, *18*, 1527–15554.

Hinton, G. E. und Sejnowski, T. (1983). Optimal perceptual inference. In *CVPR*, S. 448–453.

Hinton, G. E. und Sejnowski, T. (1986). Learning and relearning in Boltzmann machines. In Rumelhart, D. E. und McClelland, J. L. (Hrsg.), *Parallel Distributed Processing*, Kap. 7, S. 282–317. MIT Press.

Hirsh, H. (1987). Explanation-based generalization in a logic programming environment. In *IJCAI-87*.

Hobbs, J. R. (1990). *Literature and Cognition*. CSLI Press.

Hobbs, J. R., Appelt, D., Bear, J., Israel, D., Kameyama, M., Stickel, M. E. und Tyson, M. (1997). FASTUS: A cascaded finite-state transducer for extracting information from natural-language text. In Roche, E. und Schabes, Y. (Hrsg.), *Finite-State Devices for Natural Language Processing*, S. 383–406. MIT Press.

Hobbs, J. R. und Moore, R. C. (Hrsg.). (1985). *Formal Theories of the Commonsense World*. Ablex.

Hobbs, J. R., Stickel, M. E., Appelt, D. und Martin, P. (1993). Interpretation as abduction. *AIJ*, *63*(1–2), 69–142.

Hoffmann, J. (2001). FF: The fast-forward planning system. *AIMag*, *22*(3), 57–62.

Hoffmann, J. und Brafman, R. I. (2006). Conformant planning via heuristic forward search: A new approach. *AIJ*, *170*(6–7), 507–541.

Hoffmann, J. und Brafman, R. I. (2005). Contingent planning via heuristic forward search with implicit belief states. In *ICAPS-05*.

Hoffmann, J. (2005). Where "ignoring delete lists" works: Local search topology in planning benchmarks. *JAIR*, *24*, 685–758.

Hoffmann, J. und Nebel, B. (2001). The FF planning system: Fast plan generation through heuristic search. *JAIR*, *14*, 253–302.

Hoffmann, J., Sabharwal, A. und Domshlak, C. (2006). Friends or foes? An AI planning perspective on abstraction and search. In *ICAPS-06*, S. 294–303.

Hogan, N. (1985). Impedance control: An approach to manipulation. Parts I, II, and III. *J. Dynamic Systems, Measurement, and Control*, *107*(3), 1–24.

Hoiem, D., Efros, A. A. und Hebert, M. (2008). Putting objects in perspective. *IJCV*, *80*(1).

Holland, J. H. (1975). *Adaption in Natural and Artificial Systems*. University of Michigan Press.

Holland, J. H. (1995). *Hidden Order: How Adaptation Builds Complexity*. Addison-Wesley.

Holte, R. und Hernadvolgyi, I. (2001). Steps towards the automatic creation of search heuristics. Tech. rep. TR04-02, CS Dept., Univ. of Alberta.

Holzmann, G. J. (1997). The Spin model checker. *IEEE Transactions on Software Engineering*, *23*(5), 279–295.

Hood, A. (1824). Case 4th – 28 July 1824 (Mr. Hood's cases of injuries of the brain). *Phrenological Journal and Miscellany*, *2*, 82–94.

Hooker, J. (1995). Testing heuristics: We have it all wrong. *J. Heuristics*, *1*, 33–42.

Hoos, H. und Tsang, E. (2006). Local search methods. In Rossi, F., van Beek, P. und Walsh, T. (Hrsg.), *Handbook of Constraint Processing*, S. 135–168. Elsevier.

Hope, J. (1994). *The Authorship of Shakespeare's Plays*. Cambridge University Press.

Hopfield, J. J. (1982). Neurons with graded response have collective computational properties like those of two-state neurons. *PNAS*, *79*, 2554–2558.

Horn, A. (1951). On sentences which are true of direct unions of algebras. *JSL*, *16*, 14–21.

Horn, B. K. P. (1970). Shape from shading: A method for obtaining the shape of a smooth opaque object from one view. Technical report 232, MIT Artificial Intelligence Laboratory.

Horn, B. K. P. (1986). *Robot Vision*. MIT Press.

Horn, B. K. P. und Brooks, M. J. (1989). *Shape from Shading*. MIT Press.

Horn, K. V. (2003). Constructing a logic of plausible inference: A guide to cox's theorem. *IJAR*, *34*, 3–24.

Horning, J. J. (1969). *A study of grammatical inference*. Ph.D. thesis, Stanford University.

Horowitz, E. und Sahni, S. (1978). *Fundamentals of Computer Algorithms*. Computer Science Press.

Horswill, I. (2000). Functional programming of behavior-based systems. *Autonomous Robots*, *9*, 83–93.

Horvitz, E. J. (1987). Problem-solving design: Reasoning about computational value, trade-offs, and resources. In *Proc. Second Annual NASA Research Forum*, S. 26–43.

Horvitz, E. J. (1989). Rational metareasoning and compilation for optimizing decisions under bounded resources. In *Proc. Computational Intelligence 89*. Association for Computing Machinery.

Horvitz, E. J. und Barry, M. (1995). Display of information for time-critical decision making. In *UAI-95*, S. 296–305.

Horvitz, E. J., Breese, J. S., Heckerman, D. und Hovel, D. (1998). The Lumiere project: Bayesian user modeling for inferring the goals and needs of software users. In *UAI-98*, S. 256–265.

Horvitz, E. J., Breese, J. S. und Henrion, M. (1988). Decision theory in expert systems and artificial intelligence. *IJAR*, *2*, 247–302.

Horvitz, E. J. und Breese, J. S. (1996). Ideal partition of resources for metareasoning. In *AAAI-96*, S. 1229–1234.

Horvitz, E. J. und Heckerman, D. (1986). The inconsistent use of measures of certainty in artificial intelligence research. In Kanal, L. N. und Lemmer, J. F. (Hrsg.), *UAI 2*, S. 137–151. Elsevier/North-Holland.

Horvitz, E. J., Heckerman, D. und Langlotz, C. P. (1986). A framework for comparing alternative formalisms for plausible reasoning. In *AAAI-86*, Band 1, S. 210–214.

Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. MIT Press.

Howard, R. A. (1966). Information value theory. *IEEE Transactions on Systems Science and Cybernetics*, *SSC-2*, 22–26.

Howard, R. A. (1977). Risk preference. In Howard, R. A. und Matheson, J. E. (Hrsg.), *Readings in Decision Analysis*, S. 429–465. Decision Analysis Group, SRI International.

Howard, R. A. (1989). Microrisks for medical decision analysis. *Int. J. Technology Assessment in Health Care*, *5*, 357–370.

Howard, R. A. und Matheson, J. E. (1984). Influence diagrams. In Howard, R. A. und Matheson, J. E. (Hrsg.), *Readings on the Principles and Applications of Decision Analysis*, S. 721–762. Strategic Decisions Group.

Howe, D. (1987). The computational behaviour of girard's paradox. In *LICS-87*, S. 205–214.

Hsu, F.-H. (2004). *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press.

Hsu, F.-H., Anantharaman, T. S., Campbell, M. S. und Nowatzyk, A. (1990). A grandmaster chess machine. *Scientific American*, *263*(4), 44–50.

Hu, J. und Wellman, M. P. (1998). Multiagent reinforcement learning: Theoretical framework and an algorithm. In *ICML-98*, S. 242–250.

Hu, J. und Wellman, M. P. (2003). Nash q-learning for general-sum stochastic games. *JMLR*, *4*, 1039–1069.

Huang, T., Koller, D., Malik, J., Ogasawara, G., Rao, B., Russell, S. J. und Weber, J. (1994). Automatic symbolic traffic scene analysis using belief networks. In *AAAI-94*, S. 966–972.

Huang, T. und Russell, S. J. (1998). Object identification: A Bayesian analysis with application to traffic surveillance. *AIJ*, *103*, 1–17.

Huang, X. D., Acero, A. und Hon, H. (2001). *Spoken Language Processing*. Prentice Hall.

Hubel, D. H. (1988). *Eye, Brain, and Vision*. W. H. Freeman.

Huddleston, R. D. und Pullum, G. K. (2002). *The Cambridge Grammar of the English Language*. Cambridge University Press.

Huffman, D. A. (1971). Impossible objects as nonsense sentences. In Meltzer, B. und Michie, D. (Hrsg.), *Machine Intelligence 6*, S. 295–324. Edinburgh University Press.

Hughes, B. D. (1995). *Random Walks and Random Environments, Band 1: Random Walks*. Oxford University Press.

Hughes, G. E. und Cresswell, M. J. (1996). *A New Introduction to Modal Logic*. Routledge.

Huhns, M. N. und Singh, M. P. (Hrsg.). (1998). *Readings in Agents*. Morgan Kaufmann.

Hume, D. (1739). *A Treatise of Human Nature* (2. Auflage). Neu veröffentlicht von Oxford University Press, 1978, Oxford, UK.

Humphrys, M. (2008). How my program passed the turing test. In Epstein, R., Roberts, G. und Beber, G. (Hrsg.), *Parsing the Turing Test*. Springer.

Hunsberger, L. und Grosz, B. J. (2000). A combinatorial auction for collaborative planning. In *Int. Conference on Multi-Agent Systems (ICMAS-2000)*.

Hunt, W. und Brock, B. (1992). A formal HDL and its use in the FM9001 verification. *Philosophical Transactions of the Royal Society of London*, *339*.

Hunter, L. und States, D. J. (1992). Bayesian classification of protein structure. *IEEE Expert*, *7*(4), 67–75.

Hurst, M. (2000). *The Interpretation of Text in Tables*. Ph.D. thesis, Edinburgh.

Hurwicz, L. (1973). The design of mechanisms for resource allocation. *American Economic Review Papers and Proceedings*, *63*(1), 1–30.

Husmeier, D. (2003). Sensitivity and specificity of inferring genetic regulatory interactions from microarray experiments with dynamic bayesian networks. *Bioinformatics*, *19*(17), 2271–2282.

Huth, M. und Ryan, M. (2004). *Logic in computer science: modelling and reasoning about systems* (2. Auflage). Cambridge University Press.

Huttenlocher, D. und Ullman, S. (1990). Recognizing solid objects by alignment with an image. *IJCV*, *5*(2), 195–212.

Huygens, C. (1657). De ratiociniis in ludo aleae. In van Schooten, F. (Hrsg.), *Exercitionum Mathematicorum*. Elsevirii, Amsterdam. Übersetzt ins Englische von John Arbuthnot (1692).

Huyn, N., Dechter, R. und Pearl, J. (1980). Probabilistic analysis of the complexity of A*. *AIJ*, *15*(3), 241–254.

Hwa, R. (1998). An empirical evaluation of probabilistic lexicalized tree insertion grammars. In *ACL-98*, S. 557–563.

Hwang, C. H. und Schubert, L. K. (1993). EL: A formal, yet natural, comprehensive knowledge representation. In *AAAI-93*, S. 676–682.

Ingerman, P. Z. (1967). Panini–Backus form suggested. *CACM*, *10*(3), 137.

Inoue, K. (2001). Inverse entailment for full clausal theories. In *LICS-2001 Workshop on Logic and Learning*.

Intille, S. und Bobick, A. (1999). A framework for recognizing multi-agent action from visual evidence. In *AAAI-99*, S. 518–525.

Isard, M. und Blake, A. (1996). Contour tracking by stochastic propagation of conditional density. In *ECCV*, S. 343–356.

Iwama, K. und Tamaki, S. (2004). Improved upper bounds for 3-SAT. In *SODA-04*.

Jaakkola, T. und Jordan, M. I. (1996). Computing upper and lower bounds on likelihoods in intractable networks. In *UAI-96*, S. 340–348. Morgan Kaufmann.

Jaakkola, T., Singh, S. P. und Jordan, M. I. (1995). Reinforcement learning algorithm for partially observable Markov decision problems. In *NIPS 7*, S. 345–352.

Jackson, F. (1982). Epiphenomenal qualia. *Philosophical Quarterly*, *32*, 127–136.

Jaffar, J. und Lassez, J.-L. (1987). Constraint logic programming. In *Proc. Fourteenth ACM Conference on Principles of Programming Languages*, S. 111–119. Association for Computing Machinery.

Jaffar, J., Michaylov, S., Stuckey, P. J. und Yap, R. H. C. (1992). The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems*, *14*(3), 339–395.

Jaynes, E. T. (2003). *Probability Theory: The Logic of Science*. Cambridge Univ. Press.

Jefferson, G. (1949). The mind of mechanical man: The Lister Oration delivered at the Royal College of Surgeons in England. *British Medical Journal*, *1*(25), 1105–1121.

Jeffrey, R. C. (1983). *The Logic of Decision* (2. Auflage). University of Chicago Press.

Jeffreys, H. (1948). *Theory of Probability*. Oxford.

Jelinek, F. (1976). Continuous speech recognition by statistical methods. *Proc. IEEE*, *64*(4), 532–556.

Jelinek, F. (1997). *Statistical Methods for Speech Recognition*. MIT Press.

Jelinek, F. und Mercer, R. L. (1980). Interpolated estimation of Markov source parameters from sparse data. In *Proc. Workshop on Pattern Recognition in Practice*, S. 381–397.

Jennings, H. S. (1906). *Behavior of the Lower Organisms*. Columbia University Press.

Jenniskens, P., Betlem, H., Betlem, J. und Barifaijo, E. (1994). The Mbale meteorite shower. *Meteoritics*, *29*(2), 246–254.

Jensen, F. V. (2001). *Bayesian Networks and Decision Graphs*. Springer-Verlag.

Jensen, F. V. (2007). *Bayesian Networks and Decision Graphs*. Springer-Verlag.

Jevons, W. S. (1874). *The Principles of Science*. Routledge/Thoemmes Press, London.

Ji, S., Parr, R., Li, H., Liao, X. und Carin, L. (2007). Point-based policy iteration. In *AAAI-07*.

Jimenez, P. und Torras, C. (2000). An efficient algorithm for searching implicit AND/OR graphs with cycles. *AIJ*, *124*(1), 1–30.

Joachims, T. (2001). A statistical learning model of text classification with support vector machines. In *SIGIR-01*, S. 128–136.

Johnson, W. W. und Story, W. E. (1879). Notes on the "15" puzzle. *American Journal of Mathematics*, *2*, 397–404.

Johnston, M. D. und Adorf, H.-M. (1992). Scheduling with neural networks: The case of the Hubble space telescope. *Computers and Operations Research*, *19*(3–4), 209–240.

Jones, N. D., Gomard, C. K. und Sestoft, P. (1993). *Partial Evaluation and Automatic Program Generation*. Prentice-Hall.

Jones, R., Laird, J. und Nielsen, P. E. (1998). Automated intelligent pilots for combat flight simulation. In *AAAI-98*, S. 1047–54.

Jones, R., McCallum, A., Nigam, K. und Riloff, E. (1999). Bootstrapping for text learning tasks. In *Proc. IJCAI-99 Workshop on Text Mining: Foundations, Techniques, and Applications*, S. 52–63.

Jones, T. (2007). *Artificial Intelligence: A Systems Approach*. Infinity Science Press.

Jonsson, A., Morris, P., Muscettola, N., Rajan, K. und Smith, B. (2000). Planning in interplanetary space: Theory and practice. In *AIPS-00*, S. 177–186.

Jordan, M. I. (1995). Why the logistic function? a tutorial discussion on probabilities and neural networks. Computational cognitive science technical report 9503, Massachusetts Institute of Technology.

Jordan, M. I. (2005). Dirichlet processes, Chinese restaurant processes and all that. Tutorial presentation at the NIPS Conference.

Jordan, M. I., Ghahramani, Z., Jaakkola, T. und Saul, L. K. (1998). An introduction to variational methods for graphical models. In Jordan, M. I. (Hrsg.), *Learning in Graphical Models*. Kluwer.

Jouannaud, J.-P. und Kirchner, C. (1991). Solving equations in abstract algebras: A rule-based survey of unification. In Lassez, J.-L. und Plotkin, G. (Hrsg.), *Computational Logic*, S. 257–321. MIT Press.

Judd, J. S. (1990). *Neural Network Design and the Complexity of Learning*. MIT Press.

Juels, A. und Wattenberg, M. (1996). Stochastic hillclimbing as a baseline method for evaluating genetic algorithms. In Touretzky, D. S., Mozer, M. C. und Hasselmo, M. E. (Hrsg.), *NIPS 8*, S. 430–6. MIT Press.

Junker, U. (2003). The logic of ilog (j)configurator: Combining constraint programming with a description logic. In *Proc. IJCAI-03 Configuration Workshop*, S. 13–20.

Jurafsky, D. und Martin, J. H. (2000). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice-Hall.

Jurafsky, D. und Martin, J. H. (2008). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition* (2. Auflage). Prentice-Hall.

Kadane, J. B. und Simon, H. A. (1977). Optimal strategies for a class of constrained sequential problems. *Annals of Statistics*, *5*, 237–255.

Kadane, J. B. und Larkey, P. D. (1982). Subjective probability and the theory of games. *Management Science*, *28*(2), 113–120.

Kaelbling, L. P., Littman, M. L. und Cassandra, A. R. (1998). Planning and actiong in partially observable stochastic domains. *AIJ*, *101*, 99–134.

Kaelbling, L. P., Littman, M. L. und Moore, A. W. (1996). Reinforcement learning: A survey. *JAIR*, *4*, 237–285.

Kaelbling, L. P. und Rosenschein, S. J. (1990). Action and planning in embedded agents. *Robotics and Autonomous Systems*, *6*(1–2), 35–48.

Kager, R. (1999). *Optimality Theory*. Cambridge University Press.

Kahn, H. und Marshall, A. W. (1953). Methods of reducing sample size in Monte Carlo computations. *Operations Research*, *1*(5), 263–278.

Kahneman, D., Slovic, P. und Tversky, A. (Hrsg.). (1982). *Judgment under Uncertainty: Heuristics and Biases*. Cambridge University Press.

Kahneman, D. und Tversky, A. (1979). Prospect theory: An analysis of decision under risk. *Econometrica*, S. 263–291.

Kaindl, H. und Khorsand, A. (1994). Memory-bounded bidirectional search. In *AAAI-94*, S. 1359–1364.

Kalman, R. (1960). A new approach to linear filtering and prediction problems. *J. Basic Engineering*, *82*, 35–46.

Kambhampati, S. (1994). Exploiting causal structure to control retrieval and refitting during plan reuse. *Computational Intelligence*, *10*, 213–244.

Kambhampati, S., Mali, A. D. und Srivastava, B. (1998). Hybrid planning for partially hierarchical domains. In *AAAI-98*, S. 882–888.

Kanal, L. N. und Kumar, V. (1988). *Search in Artificial Intelligence*. Springer-Verlag.

Kanazawa, K., Koller, D. und Russell, S. J. (1995). Stochastic simulation algorithms for dynamic probabilistic networks. In *UAI-95*, S. 346–351.

Kantorovich, L. V. (1939). Mathematical methods of organizing and planning production. Veröffentlicht als Übersetzung in *Management Science*, *6*(4), 366–422, Juli 1960.

Kaplan, D. und Montague, R. (1960). A paradox regained. *Notre Dame Journal of Formal Logic*, *1*(3), 79–90.

Karmarkar, N. (1984). A new polynomial-time algorithm for linear programming. *Combinatorica*, *4*, 373–395.

Karp, R. M. (1972). Reducibility among combinatorial problems. In Miller, R. E. und Thatcher, J. W. (Hrsg.), *Complexity of Computer Computations*, S. 85–103. Plenum.

Kartam, N. A. und Levitt, R. E. (1990). A constraint-based approach to construction planning of multi-story buildings. In *Expert Planning Systems*, S. 245–250. Institute of Electrical Engineers.

Kasami, T. (1965). An efficient recognition and syntax analysis algorithm for context-free languages. Tech. rep. AFCRL-65-758, Air Force Cambridge Research Laboratory.

Kasparov, G. (1997). IBM owes me a rematch. *Time*, *149*(21), 66–67.

Kaufmann, M., Manolios, P. und Moore, J. S. (2000). *Computer-Aided Reasoning: An Approach*. Kluwer.

Kautz, H. (2006). Deconstructing planning as satisfiability. In *AAAI-06*.

Kautz, H., McAllester, D. A. und Selman, B. (1996). Encoding plans in propositional logic. In *KR-96*, S. 374–384.

Kautz, H. und Selman, B. (1992). Planning as satisfiability. In *ECAI-92*, S. 359–363.

Kautz, H. und Selman, B. (1998). BLACKBOX: A new approach to the application of theorem proving to problem solving. Working Notes of the AIPS-98 Workshop on Planning as Combinatorial Search.

Kavraki, L., Svestka, P., Latombe, J.-C. und Overmars, M. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, *12*(4), 566–580.

Kay, M., Gawron, J. M. und Norvig, P. (1994). *Verbmobil: A Translation System for Face-To-Face Dialog*. CSLI Press.

Kearns, M. (1990). *The Computational Complexity of Machine Learning*. MIT Press.

Kearns, M., Mansour, Y. und Ng, A. Y. (2000). Approximate planning in large POMDPs via reusable trajectories. In Solla, S. A., Leen, T. K. und Müller, K.-R. (Hrsg.), *NIPS 12*. MIT Press.

Kearns, M. und Singh, S. P. (1998). Near-optimal reinforcement learning in polynomial time. In *ICML-98*, S. 260–268.

Kearns, M. und Vazirani, U. (1994). *An Introduction to Computational Learning Theory*. MIT Press.

Kearns, M. und Mansour, Y. (1998). A fast, bottom-up decision tree pruning algorithm with near-optimal generalization. In *ICML-98*, S. 269–277.

Kebeasy, R. M., Hussein, A. I. und Dahy, S. A. (1998). Discrimination between natural earthquakes and nuclear explosions using the Aswan Seismic Network. *Annali di Geofisica*, *41*(2), 127–140.

Keeney, R. L. (1974). Multiplicative utility functions. *Operations Research*, *22*, 22–34.

Keeney, R. L. und Raiffa, H. (1976). *Decisions with Multiple Objectives: Preferences and Value Trade-offs*. Wiley.

Kemp, M. (Hrsg.). (1989). *Leonardo on Painting: An Anthology of Writings*. Yale University Press.

Kephart, J. O. und Chess, D. M. (2003). The vision of autonomic computing. *IEEE Computer*, *36*(1), 41–50.

Kersting, K., Raedt, L. D. und Kramer, S. (2000). Interpreting bayesian logic programs. In *Proc. AAAI-2000 Workshop on Learning Statistical Models from Relational Data*.

Kessler, B., Nunberg, G. und Schütze, H. (1997). Automatic detection of text genre. *CoRR*, *cmplg/9707002*.

Keynes, J. M. (1921). *A Treatise on Probability*. Macmillan.

Khare, R. (2006). Microformats: The next (small) thing on the semantic web. *IEEE Internet Computing*, *10*(1), 68–75.

Khatib, O. (1986). Real-time obstacle avoidance for robot manipulator and mobile robots. *Int. J. Robotics Research*, *5*(1), 90–98.

Khmelev, D. V. und Tweedie, F. J. (2001). Using Markov chains for identification of writer. *Literary and Linguistic Computing*, *16*(3), 299–307.

Kietz, J.-U. und Džeroski, S. (1994). Inductive logic programming and learnability. *SIGART Bulletin*, *5*(1), 22–32.

Kilgarriff, A. und Grefenstette, G. (2006). Introduction to the special issue on the web as corpus. *Computational Linguistics*, *29*(3), 333–347.

Kim, J. H. (1983). *CONVINCE: A Conversational Inference Consolidation Engine*. Ph.D. thesis, Department of Computer Science, University of California at Los Angeles.

Kim, J. H. und Pearl, J. (1983). A computational model for combined causal and diagnostic reasoning in inference systems. In *IJCAI-83*, S. 190–193.

Kim, J.-H., Lee, C.-H., Lee, K.-H. und Kuppuswamy, N. (2007). Evolving personality of a genetic robot in ubiquitous environment. In *The 16th IEEE International Symposium on Robot and Human interactive Communication*, S. 848–853.

King, R. D., Rowland, J., Oliver, S. G. und Young, M. (2009). The automation of science. *Science*, *324*(5923), 85–89.

Kirk, D. E. (2004). *Optimal Control Theory: An Introduction*. Dover.

Kirkpatrick, S., Gelatt, C. D. und Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*, 671–680.

Kister, J., Stein, P., Ulam, S., Walden, W. und Wells, M. (1957). Experiments in chess. *JACM*, *4*, 174–177.

Kisynski, J. und Poole, D. (2009). Lifted aggregation in directed first-order probabilistic models. In *IJCAI-09*.

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I. und Osawa, E. (1997a). RoboCup: The robot world cup initiative. In *Proc. First International Conference on Autonomous Agents*, S. 340–347.

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E. und Matsubara, H. (1997b). RoboCup: A challenge problem for AI. *AIMag*, *18*(1), 73–85.

Kjaerulff, U. (1992). A computational scheme for reasoning in dynamic probabilistic networks. In *UAI-92*, S. 121–129.

Klein, D. und Manning, C. (2001). Parsing with tree-bank grammars: Empirical bounds, theoretical models, and the structure of the Penn treebank. In *ACL-01*.

Klein, D. und Manning, C. (2003). A* parsing: Fast exact Viterbi parse selection. In *HLT-NAACL-03*, S. 119–126.

Klein, D., Smarr, J., Nguyen, H. und Manning, C. (2003). Named entity recognition with character-level models. In *Conference on Natural Language Learning (CoNLL)*.

Kleinberg, J. M. (1999). Authoritative sources in a hyperlinked environment. *JACM*, *46*(5), 604–632.

Klemperer, P. (2002). What really matters in auction design. *J. Economic Perspectives*, *16*(1).

Kneser, R. und Ney, H. (1995). Improved backing-off for M-gram language modeling. In *ICASSP-95*, S. 181–184.

Knight, K. (1999). A statistical MT tutorial workbook. Prepared in connection with the Johns Hopkins University summer workshop.

Knuth, D. E. (1964). Representing numbers using only one 4. *Mathematics Magazine*, *37*(Nov/Dec), 308–310.

Knuth, D. E. (1968). Semantics for context-free languages. *Mathematical Systems Theory*, *2*(2), 127–145.

Knuth, D. E. (1973). *The Art of Computer Programming* (2. Auflage)., Band 2: Fundamental Algorithms. Addison-Wesley.

Knuth, D. E. (1975). An analysis of alpha–beta pruning. *AIJ*, *6*(4), 293–326.

Knuth, D. E. und Bendix, P. B. (1970). Simple word problems in universal algebras. In Leech, J. (Hrsg.), *Computational Problems in Abstract Algebra*, S. 263–267. Pergamon.

Kocsis, L. und Szepesvari, C. (2006). Bandit-based Monte-Carlo planning. In *ECML-06*.

Koditschek, D. (1987). Exact robot navigation by means of potential functions: some topological considerations. In *ICRA-87*, Band 1, S. 1–6.

Koehler, J., Nebel, B., Hoffmann, J. und Dimopoulos, Y. (1997). Extending planning graphs to an ADL subset. In *ECP-97*, S. 273–285.

Koehn, P. (2009). *Statistical Machine Translation*. Cambridge University Press.

Koenderink, J. J. (1990). *Solid Shape*. MIT Press.

Koenig, S. (1991). Optimal probabilistic and decision-theoretic planning using Markovian decision theory. Master's report, Computer Science Division, University of California.

Koenig, S. (2000). Exploring unknown environments with real-time search or reinforcement learning. In Solla, S. A., Leen, T. K. und Müller, K.-R. (Hrsg.), *NIPS 12*. MIT Press.

Koenig, S. (2001). Agent-centered search. *AIMag*, *22*(4), 109–131.

Koller, D., Meggido, N. und von Stengel, B. (1996). Efficient computation of equilibria for extensive two-person games. *Games and Economic Behaviour*, *14*(2), 247–259.

Koller, D. und Pfeffer, A. (1997). Representations and solutions for game-theoretic problems. *AIJ*, *94*(1–2), 167–215.

Koller, D. und Pfeffer, A. (1998). Probabilistic frame-based systems. In *AAAI-98*, S. 580–587.

Koller, D. und Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. MIT Press.

Koller, D. und Milch, B. (2003). Multi-agent influence diagrams for representing and solving games. *Games and Economic Behavior*, *45*, 181–221.

Koller, D. und Parr, R. (2000). Policy iteration for factored MDPs. In *UAI-00*, S. 326–334.

Koller, D. und Sahami, M. (1997). Hierarchically classifying documents using very few words. In *ICML-97*, S. 170–178.

Kolmogorov, A. N. (1941). Interpolation und extrapolation von stationaren zufalligen folgen. *Bulletin of the Academy of Sciences of the USSR, Ser. Math. 5*, 3–14.

Kolmogorov, A. N. (1950). *Foundations of the Theory of Probability*. Chelsea.

Kolmogorov, A. N. (1963). On tables of random numbers. *Sankhya, the Indian Journal of Statistics, Series A 25*.

Kolmogorov, A. N. (1965). Three approaches to the quantitative definition of information. *Problems in Information Transmission*, *1*(1), 1–7.

Kolodner, J. (1983). Reconstructive memory: A computer model. *Cognitive Science*, *7*, 281–328.

Kolodner, J. (1993). *Case-Based Reasoning*. Morgan Kaufmann.

Kondrak, G. und van Beek, P. (1997). A theoretical evaluation of selected backtracking algorithms. *AIJ*, *89*, 365–387.

Konolige, K. (1997). COLBERT: A language for reactive control in Saphira. In *Künstliche Intelligenz: Advances in Artificial Intelligence*, LNAI, S. 31–52.

Konolige, K. (2004). Large-scale map-making. In *AAAI-04*, S. 457–463.

Konolige, K. (1982). A first order formalization of knowledge and action for a multi-agent planning system. In Hayes, J. E., Michie, D. und Pao, Y.-H. (Hrsg.), *Machine Intelligence 10*. Ellis Horwood.

Konolige, K. (1994). Easy to be hard: Difficult problems for greedy algorithms. In *KR-94*, S. 374–378.

Koo, T., Carreras, X. und Collins, M. (2008). Simple semi-supervised dependency parsing. In *ACL-08*.

Koopmans, T. C. (1972). Representation of preference orderings over time. In McGuire, C. B. und Radner, R. (Hrsg.), *Decision and Organization*. Elsevier/North-Holland.

Korb, K. B. und Nicholson, A. (2003). *Bayesian Artificial Intelligence*. Chapman and Hall.

Korb, K. B., Nicholson, A. und Jitnah, N. (1999). Bayesian poker. In *UAI-99*.

Korf, R. E. (1985a). Depth-first iterative-deepening: an optimal admissible tree search. *AIJ*, *27(1)*, 97–109.

Korf, R. E. (1985b). Iterative-deepening A*: An optimal admissible tree search. In *IJCAI-85*, S. 1034–1036.

Korf, R. E. (1987). Planning as search: A quantitative approach. *AIJ*, *33(1)*, 65–88.

Korf, R. E. (1990). Real-time heuristic search. *AIJ*, *42*(3), 189–212.

Korf, R. E. (1993). Linear-space best-first search. *AIJ*, *62*(1), 41–78.

Korf, R. E. (1995). Space-efficient search algorithms. *ACM Computing Surveys*, *27*(3), 337–339.

Korf, R. E. und Chickering, D. M. (1996). Best-first minimax search. *AIJ*, *84*(1–2), 299–337.

Korf, R. E. und Felner, A. (2002). Disjoint pattern database heuristics. *AIJ*, *134*(1–2), 9–22.

Korf, R. E., Reid, M. und Edelkamp, S. (2001). Time complexity of iterative-deepening-A*. *AIJ*, *129*, 199–218.

Korf, R. E. und Zhang, W. (2000). Divide-and-conquer frontier search applied to optimal sequence alignment. In *American Association for Artificial Intelligence*, S. 910–916.

Korf, R. E. (2008). Linear-time disk-based implicit graph search. *JACM*, *55*(6).

Korf, R. E. und Schultze, P. (2005). Large-scale parallel breadth-first search. In *AAAI-05*, S. 1380–1385.

Kotok, A. (1962). A chess playing program for the IBM 7090. AI project memo 41, MIT Computation Center.

Koutsoupias, E. und Papadimitriou, C. H. (1992). On the greedy algorithm for satisfiability. *Information Processing Letters*, *43*(1), 53–55.

Kowalski, R. (1974). Predicate logic as a programming language. In *Proc. IFIP Congress*, S. 569–574.

Kowalski, R. (1979). *Logic for Problem Solving*. Elsevier/North-Holland.

Kowalski, R. (1988). The early years of logic programming. *CACM*, *31*, 38–43.

Kowalski, R. und Sergot, M. (1986). A logic-based calculus of events. *New Generation Computing*, *4*(1), 67–95.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.

Koza, J. R. (1994). *Genetic Programming II: Automatic discovery of reusable programs*. MIT Press.

Koza, J. R., Bennett, F. H., Andre, D. und Keane, M. A. (1999). *Genetic Programming III: Darwinian invention and problem solving*. Morgan Kaufmann.

Kraus, S., Ephrati, E. und Lehmann, D. (1991). Negotiation in a non-cooperative environment. *AIJ*, *3*(4), 255–281.

Krause, A. und Guestrin, C. (2009). Optimal value of information in graphical models. *JAIR*, *35*, 557–591.

Krause, A., McMahan, B., Guestrin, C. und Gupta, A. (2008). Robust submodular observation selection. *JMLR*, *9*, 2761–2801.

Kripke, S. A. (1963). Semantical considerations on modal logic. *Acta Philosophica Fennica*, *16*, 83–94.

Krogh, A., Brown, M., Mian, I. S., Sjolander, K. und Haussler, D. (1994). Hidden Markov models in computational biology: Applications to protein modeling. *J. Molecular Biology*, *235*, 1501–1531.

Kübler, S., McDonald, R. und Nivre, J. (2009). *Dependency Parsing*. Morgan Claypool.

Kuhn, H. W. (1953). Extensive games and the problem of information. In Kuhn, H. W. und Tucker, A. W. (Hrsg.), *Contributions to the Theory of Games II*. Princeton University Press.

Kuhn, H. W. (1955). The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, *2*, 83–97.

Kuipers, B. J. (1985). Qualitative simulation. In Bobrow, D. (Hrsg.), *Qualitative Reasoning About Physical Systems*, S. 169–203. MIT Press.

Kuipers, B. J. und Levitt, T. S. (1988). Navigation and mapping in large-scale space. *AIMag*, *9*(2), 25–43.

Kuipers, B. J. (2001). Qualitative simulation. In Meyers, R. A. (Hrsg.), *Encyclopeida of Physical Science and Technology*. Academic Press.

Kumar, P. R. und Varaiya, P. (1986). *Stochastic Systems: Estimation, Identification, and Adaptive Control*. Prentice-Hall.

Kumar, V. (1992). Algorithms for constraint satisfaction problems: A survey. *AIMag*, *13*(1), 32–44.

Kumar, V. und Kanal, L. N. (1983). A general branch and bound formulation for understanding and synthesizing and/or tree search procedures. *AIJ*, *21*, 179–198.

Kumar, V. und Kanal, L. N. (1988). The CDP: A unifying formulation for heuristic search, dynamic programming, and branch-and-bound. In Kanal, L. N. und Kumar, V. (Hrsg.), *Search in Artificial Intelligence*, Kap. 1, S. 1–27. Springer-Verlag.

Kumar, V., Nau, D. S. und Kanal, L. N. (1988). A general branch-and-bound formulation for AND/OR graph and game tree search. In Kanal, L. N. und Kumar, V. (Hrsg.), *Search in Artificial Intelligence*, Kap. 3, S. 91–130. Springer-Verlag.

Kurien, J., Nayak, P. und Smith, D. E. (2002). Fragment-based conformant planning. In *AIPS-02*.

Kurzweil, R. (1990). *The Age of Intelligent Machines*. MIT Press.

Kurzweil, R. (2005). *The Singularity is Near*. Viking.

Kwok, C., Etzioni, O. und Weld, D. S. (2001). Scaling question answering to the web. In *Proc. 10th International Conference on the World Wide Web*.

Kyburg, H. E. und Teng, C.-M. (2006). Nonmonotonic logic and statistical inference. *Computational Intelligence*, *22*(1), 26–51.

Kyburg, H. E. (1977). Randomness and the right reference class. *J. Philosophy*, *74*(9), 501–521.

Kyburg, H. E. (1983). The reference class. *Philosophy of Science*, *50*, 374–397.

La Mettrie, J. O. (1748). *L'homme machine*. E. Luzac, Leyde, France.

La Mura, P. und Shoham, Y. (1999). Expected utility networks. In *UAI-99*, S. 366–373.

Laborie, P. (2003). Algorithms for propagating resource constraints in AI planning and scheduling. *AIJ*, *143*(2), 151–188.

Ladkin, P. (1986a). Primitives and units for time specification. In *AAAI-86*, Band 1, S. 354–359.

Ladkin, P. (1986b). Time representation: a taxonomy of interval relations. In *AAAI-86*, Band 1, S. 360–366.

Lafferty, J., McCallum, A. und Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *ICML-01*.

Lafferty, J. und Zhai, C. (2001). Probabilistic relevance models based on document and query generation. In *Proc. Workshop on Language Modeling and Information Retrieval*.

Lagoudakis, M. G. und Parr, R. (2003). Least-squares policy iteration. *JMLR*, *4*, 1107–1149.

Laird, J., Newell, A. und Rosenbloom, P. S. (1987). SOAR: An architecture for general intelligence. *AIJ*, *33*(1), 1–64.

Laird, J., Rosenbloom, P. S. und Newell, A. (1986). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, *1*, 11–46.

Laird, J. (2008). Extending the Soar cognitive architecture. In *Artificial General Intelligence Conference*.

Lakoff, G. (1987). *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind*. University of Chicago Press.

Lakoff, G. und Johnson, M. (1980). *Metaphors We Live By*. University of Chicago Press.

Lakoff, G. und Johnson, M. (1999). *Philosophy in the Flesh: The Embodied Mind and Its Challenge to Western Thought*. Basic Books.

Lam, J. und Greenspan, M. (2008). Eye-in-hand visual servoing for accurate shooting in pool robotics. In *5th Canadian Conference on Computer and Robot Vision*.

Lamarck, J. B. (1809). *Philosophie zoologique*. Chez Dentu et L'Auteur, Paris.

Landhuis, E. (2004). Lifelong debunker takes on arbiter of neutral choices: Magician-turned-mathematician uncovers bias in a flip of a coin. *Stanford Report*.

Langdon, W. und Poli, R. (2002). *Foundations of Genetic Programming*. Springer.

Langley, P., Simon, H. A., Bradshaw, G. L. und Zytkow, J. M. (1987). *Scientific Discovery: Computational Explorations of the Creative Processes*. MIT Press.

Langton, C. (Hrsg.). (1995). *Artificial Life*. MIT Press.

Laplace, P. (1816). *Essai philosophique sur les probabilités* (3. Auflage). Courcier Imprimeur, Paris.

Laptev, I. und Perez, P. (2007). Retrieving actions in movies. In *ICCV*, S. 1–8.

Lari, K. und Young, S. J. (1990). The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, *4*, 35–56.

Larson, S. C. (1931). The shrinkage of the coefficient of multiple correlation. *J. Educational Psychology*, *22*, 45–55.

Laskey, K. B. (2008). MEBN: A language for first-order bayesian knowledge bases. *AIJ*, *172*, 140–178.

Latombe, J.-C. (1991). *Robot Motion Planning*. Kluwer.

Lauritzen, S. (1995). The EM algorithm for graphical association models with missing data. *Computational Statistics and Data Analysis*, *19*, 191–201.

Lauritzen, S. (1996). *Graphical models*. Oxford University Press.

Lauritzen, S., Dawid, A. P., Larsen, B. und Leimer, H. (1990). Independence properties of directed Markov fields. *Networks*, *20*(5), 491–505.

Lauritzen, S. und Spiegelhalter, D. J. (1988). Local computations with probabilities on graphical structures and their application to expert systems. *J. Royal Statistical Society, B 50*(2), 157–224.

Lauritzen, S. und Wermuth, N. (1989). Graphical models for associations between variables, some of which are qualitative and some quantitative. *Annals of Statistics*, *17*, 31–57.

LaValle, S. (2006). *Planning Algorithms*. Cambridge University Press.

Lavrač, N. und Džeroski, S. (1994). *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood.

Lawler, E. L., Lenstra, J. K., Kan, A. und Shmoys, D. B. (1992). *The Travelling Salesman Problem*. Wiley Interscience.

Lawler, E. L., Lenstra, J. K., Kan, A. und Shmoys, D. B. (1993). Sequencing and scheduling: Algorithms and complexity. In Graves, S. C., Zipkin, P. H. und Kan, A. H. G. R. (Hrsg.), *Logistics of Production and Inventory: Handbooks in Operations Research and Management Science, Volume 4*, S. 445–522. North-Holland.

Lawler, E. L. und Wood, D. E. (1966). Branch-and-bound methods: A survey. *Operations Research*, *14(4)*, 699–719.

Lazanas, A. und Latombe, J.-C. (1992). Landmark-based robot navigation. In *AAAI-92*, S. 816–822.

LeCun, Y., Jackel, L., Boser, B. und Denker, J. (1989). Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine*, *27*(11), 41–46.

LeCun, Y., Jackel, L., Bottou, L., Brunot, A., Cortes, C., Denker, J., Drucker, H., Guyon, I., Muller, U., Sackinger, E., Simard, P. und Vapnik, V. N. (1995). Comparison of learning algorithms for handwritten digit recognition. In *Int. Conference on Artificial Neural Networks*, S. 53–60.

Leech, G., Rayson, P. und Wilson, A. (2001). *Word Frequencies in Written and Spoken English: Based on the British National Corpus*. Longman.

Legendre, A. M. (1805). *Nouvelles méthodes pour la détermination des orbites des comètes*.

Lehrer, J. (2009). *How We Decide*. Houghton Mifflin.

Lenat, D. B. (1983). EURISKO: A program that learns new heuristics and domain concepts: The nature of heuristics, III: Program design and results. *AIJ*, *21*(1–2), 61–98.

Lenat, D. B. und Brown, J. S. (1984). Why AM and EURISKO appear to work. *AIJ*, *23*(3), 269–294.

Lenat, D. B. und Guha, R. V. (1990). *Building Large Knowledge-Based Systems: Representation and Inference in the CYC Project*. Addison-Wesley.

Leonard, H. S. und Goodman, N. (1940). The calculus of individuals and its uses. *JSL*, *5*(2), 45–55.

Leonard, J. und Durrant-Whyte, H. (1992). *Directed sonar sensing for mobile robot navigation*. Kluwer.

Leśniewski, S. (1916). Podstawy ogólnej teorii mnogo?ci. Moscow.

Lettvin, J. Y., Maturana, H. R., McCulloch, W. S. und Pitts, W. (1959). What the frog's eye tells the frog's brain. *Proc. IRE*, *47*(11), 1940–1951.

Letz, R., Schumann, J., Bayerl, S. und Bibel, W. (1992). SETHEO: A high-performance theorem prover. *JAR*, *8*(2), 183–212.

Levesque, H. J. und Brachman, R. J. (1987). Expressiveness and tractability in knowledge representation and reasoning. *Computational Intelligence*, *3*(2), 78–93.

Levin, D. A., Peres, Y. und Wilmer, E. L. (2008). *Markov Chains and Mixing Times*. American Mathematical Society.

Levitt, G. M. (2000). *The Turk, Chess Automaton*. McFarland and Company.

Levy, D. (Hrsg.). (1988a). *Computer Chess Compendium*. Springer-Verlag.

Levy, D. (Hrsg.). (1988b). *Computer Games*. Springer-Verlag.

Levy, D. (1989). The million pound bridge program. In Levy, D. und Beal, D. (Hrsg.), *Heuristic Programming in Artificial Intelligence*. Ellis Horwood.

Levy, D. (2007). *Love and Sex with Robots*. Harper.

Lewis, D. D. (1998). Naive Bayes at forty: The independence assumption in information retrieval. In *ECML-98*, S. 4–15.

Lewis, D. K. (1966). An argument for the identity theory. *J. Philosophy*, *63*(1), 17–25.

Lewis, D. K. (1980). Mad pain and Martian pain. In Block, N. (Hrsg.), *Readings in Philosophy of Psychology*, Band 1, S. 216–222. Harvard University Press.

Leyton-Brown, K. und Shoham, Y. (2008). *Essentials of Game Theory: A Concise, Multidisciplinary Introduction*. Morgan Claypool.

Li, C. M. und Anbulagan (1997). Heuristics based on unit propagation for satisfiability problems. In *IJCAI-97*, S. 366–371.

Li, M. und Vitanyi, P. M. B. (1993). *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag.

Liberatore, P. (1997). The complexity of the language **A**. *Electronic Transactions on Artificial Intelligence*, *1*, 13–38.

Lifschitz, V. (2001). Answer set programming and plan generation. *AIJ*, *138*(1–2), 39–54.

Lighthill, J. (1973). Artificial intelligence: A general survey. In Lighthill, J., Sutherland, N. S., Needham, R. M., Longuet-Higgins, H. C. und Michie, D. (Hrsg.), *Artificial Intelligence: A Paper Symposium*. Science Research Council of Great Britain.

Lin, S. (1965). Computer solutions of the travelling salesman problem. *Bell Systems Technical Journal*, *44(10)*, 2245–2269.

Lin, S. und Kernighan, B. W. (1973). An effective heuristic algorithm for the travelling-salesman problem. *Operations Research*, *21*(2), 498–516.

Lindley, D. V. (1956). On a measure of the information provided by an experiment. *Annals of Mathematical Statistics*, *27*(4), 986–1005.

Lindsay, R. K., Buchanan, B. G., Feigenbaum, E. A. und Lederberg, J. (1980). *Applications of Artificial Intelligence for Organic Chemistry: The DENDRAL Project*. McGraw-Hill.

Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *ICML-94*, S. 157–163.

Littman, M. L., Keim, G. A. und Shazeer, N. M. (1999). Solving crosswords with PROVERB. In *AAAI-99*, S. 914–915.

Liu, J. S. und Chen, R. (1998). Sequential Monte Carlo methods for dynamic systems. *JASA*, *93*, 1022–1031.

Livescu, K., Glass, J. und Bilmes, J. (2003). Hidden feature modeling for speech recognition using dynamic Bayesian networks. In *EUROSPEECH-2003*, S. 2529–2532.

Livnat, A. und Pippenger, N. (2006). An optimal brain can be composed of conflicting agents. *PNAS*, *103*(9), 3198–3202.

Locke, J. (1690). *An Essay Concerning Human Understanding*. William Tegg.

Lodge, D. (1984). *Small World*. Penguin Books.

Loftus, E. und Palmer, J. (1974). Reconstruction of automobile destruction: An example of the interaction between language and memory. *J. Verbal Learning and Verbal Behavior*, *13*, 585–589.

Lohn, J. D., Kraus, W. F. und Colombano, S. P. (2001). Evolutionary optimization of yagi-uda antennas. In *Proc. Fourth International Conference on Evolvable Systems*, S. 236–243.

Longley, N. und Sankaran, S. (2005). The NHL's overtime-loss rule: Empirically analyzing the unintended effects. *Atlantic Economic Journal*.

Longuet-Higgins, H. C. (1981). A computer algorithm for reconstructing a scene from two projections. *Nature*, *293*, 133–135.

Loo, B. T., Condie, T., Garofalakis, M., Gay, D. E., Hellerstein, J. M., Maniatis, P., Ramakrishnan, R., Roscoe, T. und Stoica, I. (2006). Declarative networking: Language, execution and optimization. In *SIGMOD-06*.

Love, N., Hinrichs, T. und Genesereth, M. R. (2006). General game playing: Game description language specification. Tech. rep. LG-2006-01, Stanford University Computer Science Dept.

Lovejoy, W. S. (1991). A survey of algorithmic methods for partially observed Markov decision processes. *Annals of Operations Research*, *28*(1–4), 47–66.

Loveland, D. (1970). A linear format for resolution. In *Proc. IRIA Symposium on Automatic Demonstration*, S. 147–162.

Lowe, D. (1987). Three-dimensional object recognition from single two-dimensional images. *AIJ*, *31*, 355–395.

Lowe, D. (1999). Object recognition using local scale invariant feature. In *ICCV*.

Lowe, D. (2004). Distinctive image features from scale-invariant keypoints. *IJCV*, *60*(2), 91–110.

Löwenheim, L. (1915). Über Möglichkeiten im Relativkalkül. *Mathematische Annalen*, *76*, 447–470.

Lowerre, B. T. (1976). *The* HARPY *Speech Recognition System*. Ph.D. thesis, Computer Science Department, Carnegie-Mellon University.

Lowerre, B. T. und Reddy, R. (1980). The HARPY speech recognition system. In Lea, W. A. (Hrsg.), *Trends in Speech Recognition*, Kap. 15. Prentice-Hall.

Lowry, M. (2008). Intelligent software engineering tools for NASA's crew exploration vehicle. In *Proc. ISMIS*.

Loyd, S. (1959). *Mathematical Puzzles of Sam Loyd: Selected and Edited by Martin Gardner*. Dover.

Lozano-Perez, T. (1983). Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, *C-32*(2), 108–120.

Lozano-Perez, T., Mason, M. und Taylor, R. (1984). Automatic synthesis of fine-motion strategies for robots. *Int. J. Robotics Research*, *3*(1), 3–24.

Lu, F. und Milios, E. (1997). Globally consistent range scan alignment for environment mapping. *Autonomous Robots*, *4*, 333–349.

Luby, M., Sinclair, A. und Zuckerman, D. (1993). Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, *47*, 173–180.

Lucas, J. R. (1961). Minds, machines und Gödel. *Philosophy*, *36*.

Lucas, J. R. (1976). This Gödel is killing me: A rejoinder. *Philosophia*, *6*(1), 145–148.

Lucas, P. (1996). Knowledge acquisition for decision-theoretic expert systems. *AISB Quarterly*, *94*, 23–33.

Lucas, P., van der Gaag, L. und Abu-Hanna, A. (2004). Bayesian networks in biomedicine and health-care. *Artificial Intelligence in Medicine*.

Luce, D. R. und Raiffa, H. (1957). *Games and Decisions*. Wiley.

Ludlow, P., Nagasawa, Y. und Stoljar, D. (2004). *There's Something About Mary*. MIT Press.

Luger, G. F. (Hrsg.). (1995). *Computation and intelligence: Collected readings*. AAAI Press.

Lyman, P. und Varian, H. R. (2003). How much information? *www.sims.berkeley.edu/how-much-info-2003*.

Machina, M. (2005). Choice under uncertainty. In *Encyclopedia of Cognitive Science*, S. 505–514. Wiley.

MacKay, D. J. C. (1992). A practical Bayesian framework for backpropagation networks. *Neural Computation*, *4*(3), 448–472.

MacKay, D. J. C. (2002). *Information Theory, Inference and Learning Algorithms*. Cambridge University Press.

MacKenzie, D. (2004). *Mechanizing Proof*. MIT Press.

Mackworth, A. K. (1977). Consistency in networks of relations. *AIJ*, *8*(1), 99–118.

Mackworth, A. K. (1992). Constraint satisfaction. In Shapiro, S. (Hrsg.), *Encyclopedia of Artificial Intelligence* (2. Auflage)., Band 1, S. 285–293. Wiley.

Mahanti, A. und Daniels, C. J. (1993). A SIMD approach to parallel heuristic search. *AIJ*, *60*(2), 243–282.

Mailath, G. und Samuelson, L. (2006). *Repeated Games and Reputations: Long-Run Relationships*. Oxford University Press.

Majercik, S. M. und Littman, M. L. (2003). Contingent planning under uncertainty via stochastic satisfiability. *AIJ*, S. 119–162.

Malik, J. und Perona, P. (1990). Preattentive texture discrimination with early vision mechanisms. *J. Opt. Soc. Am. A*, *7*(5), 923–932.

Malik, J. und Rosenholtz, R. (1994). Recovering surface curvature and orientation from texture distortion: A least squares algorithm and sensitivity analysis. In *ECCV*, S. 353–364.

Malik, J. und Rosenholtz, R. (1997). Computing local surface orientation and shape from texture for curved surfaces. *IJCV*, *23*(2), 149–168.

Maneva, E., Mossel, E. und Wainwright, M. J. (2007). A new look at survey propagation and its generalizations. *JACM*, *54*(4).

Manna, Z. und Waldinger, R. (1971). Toward automatic program synthesis. *CACM*, *14*(3), 151–165.

Manna, Z. und Waldinger, R. (1985). *The Logical Basis for Computer Programming: Volume 1: Deductive Reasoning*. Addison-Wesley.

Manning, C. und Schütze, H. (1999). *Foundations of Statistical Natural Language Processing*. MIT Press.

Manning, C., Raghavan, P. und Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

Mannion, M. (2002). Using first-order logic for product line model validation. In *Software Product Lines: Second International Conference*. Springer.

Manzini, G. (1995). BIDA*: An improved perimeter search algorithm. *AIJ*, *72*(2), 347–360.

Marbach, P. und Tsitsiklis, J. N. (1998). Simulation-based optimization of Markov reward processes. Technical report LIDS-P-2411, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology.

Marcus, G. (2009). *Kluge: The Haphazard Evolution of the Human Mind*. Mariner Books.

Marcus, M. P., Santorini, B. und Marcinkiewicz, M. A. (1993). Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, *19*(2), 313–330.

Markov, A. A. (1913). An example of statistical investigation in the text of "Eugene Onegin" illustrating coupling of "tests" in chains. *Proc. Academy of Sciences of St. Petersburg*, *7*.

Maron, M. E. (1961). Automatic indexing: An experimental inquiry. *JACM*, *8*(3), 404–417.

Maron, M. E. und Kuhns, J.-L. (1960). On relevance, probabilistic indexing and information retrieval. *CACM*, *7*, 219–244.

Marr, D. (1982). *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. W. H. Freeman.

Marriott, K. und Stuckey, P. J. (1998). *Programming with Constraints: An Introduction*. MIT Press.

Marsland, A. T. und Schaeffer, J. (Hrsg.). (1990). *Computers, Chess, and Cognition*. Springer-Verlag.

Marsland, S. (2009). *Machine Learning: An Algorithmic Perspective*. CRC Press.

Martelli, A. und Montanari, U. (1973). Additive AND/OR graphs. In *IJCAI-73*, S. 1–11.

Martelli, A. und Montanari, U. (1978). Optimizing decision trees through heuristically guided search. *CACM*, *21*, 1025–1039.

Martelli, A. (1977). On the complexity of admissible search algorithms. *AIJ*, *8*(1), 1–13.

Marthi, B., Pasula, H., Russell, S. J. und Peres, Y. (2002). Decayed MCMC filtering. In *UAI-02*, S. 319–326.

Marthi, B., Russell, S. J., Latham, D. und Guestrin, C. (2005). Concurrent hierarchical reinforcement learning. In *IJCAI-05*.

Marthi, B., Russell, S. J. und Wolfe, J. (2007). Angelic semantics for high-level actions. In *ICAPS-07*.

Marthi, B., Russell, S. J. und Wolfe, J. (2008). Angelic hierarchical planning: Optimal and online algorithms. In *ICAPS-08*.

Martin, D., Fowlkes, C. und Malik, J. (2004). Learning to detect natural image boundaries using local brightness, color, and texture cues. *PAMI*, *26*(5), 530–549.

Martin, J. H. (1990). *A Computational Model of Metaphor Interpretation*. Academic Press.

Mason, M. (1993). Kicking the sensing habit. *AIMag*, *14*(1), 58–59.

Mason, M. (2001). *Mechanics of Robotic Manipulation*. MIT Press.

Mason, M. und Salisbury, J. (1985). *Robot hands and the mechanics of manipulation*. MIT Press.

Mataric, M. J. (1997). Reinforcement learning in the multi-robot domain. *Autonomous Robots*, *4*(1), 73–83.

Mates, B. (1953). *Stoic Logic*. University of California Press.

Matuszek, C., Cabral, J., Witbrock, M. und DeOliveira, J. (2006). An introduction to the syntax and semantics of cyc. In *Proc. AAAI Spring Symposium on Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*.

Maxwell, J. und Kaplan, R. (1993). The interface between phrasal and functional constraints. *Computational Linguistics*, *19*(4), 571–590.

McAllester, D. A. (1980). An outlook on truth maintenance. Ai memo 551, MIT AI Laboratory.

McAllester, D. A. (1988). Conspiracy numbers for min-max search. *AIJ*, *35*(3), 287–310.

McAllester, D. A. (1998). What is the most pressing issue facing AI and the AAAI today? Candidate statement, election for Councilor of the American Association for Artificial Intelligence.

McAllester, D. A. und Rosenblitt, D. (1991). Systematic nonlinear planning. In *AAAI-91*, Band 2, S. 634–639.

McCallum, A. (2003). Efficiently inducing features of conditional random fields. In *UAI-03*.

McCarthy, J. (1958). Programs with common sense. In *Proc. Symposium on Mechanisation of Thought Processes*, Band 1, S. 77–84.

McCarthy, J. (1963). Situations, actions, and causal laws. Memo 2, Stanford University Artificial Intelligence Project.

McCarthy, J. (1968). Programs with common sense. In Minsky, M. L. (Hrsg.), *Semantic Information Processing*, S. 403–418. MIT Press.

McCarthy, J. (1980). Circumscription: A form of non-monotonic reasoning. *AIJ*, *13*(1–2), 27–39.

McCarthy, J. (2007). From here to human-level AI. *AIJ*, *171*(18), 1174–1182.

McCarthy, J. und Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B., Michie, D. und Swann, M. (Hrsg.), *Machine Intelligence 4*, S. 463–502. Edinburgh University Press.

McCarthy, J., Minsky, M. L., Rochester, N. und Shannon, C. E. (1955). Proposal for the Dartmouth summer research project on artificial intelligence. Tech. rep., Dartmouth College.

McCawley, J. D. (1988). *The Syntactic Phenomena of English*, Band 2 volumes. University of Chicago Press.

McCorduck, P. (2004). *Machines who think: a personal inquiry into the history and prospects of artificial intelligence* (Überarbeitete Auflage). A K Peters.

McCulloch, W. S. und Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, *5*, 115–137.

McCune, W. (1992). Automated discovery of new axiomatizations of the left group and right group calculi. *JAR*, *9*(1), 1–24.

McCune, W. (1997). Solution of the Robbins problem. *JAR*, *19*(3), 263–276.

McDermott, D. (1976). Artificial intelligence meets natural stupidity. *SIGART Newsletter*, *57*, 4–9.

McDermott, D. (1978a). Planning and acting. *Cognitive Science*, *2*(2), 71–109.

McDermott, D. (1978b). Tarskian semantics, or, no notation without denotation! *Cognitive Science*, *2*(3).

McDermott, D. (1985). Reasoning about plans. In Hobbs, J. und Moore, R. (Hrsg.), *Formal theories of the commonsense world*. Intellect Books.

McDermott, D. (1987). A critique of pure reason. *Computational Intelligence*, *3*(3), 151–237.

McDermott, D. (1996). A heuristic estimator for means-ends analysis in planning. In *ICAPS-96*, S. 142–149.

McDermott, D. und Doyle, J. (1980). Non-monotonic logic: i. *AIJ*, *13*(1–2), 41–72.

McDermott, J. (1982). R1: A rule-based configurer of computer systems. *AIJ*, *19*(1), 39–88.

McEliece, R. J., MacKay, D. J. C. und Cheng, J.-F. (1998). Turbo decoding as an instance of Pearl's "belief propagation" algorithm. *IEEE Journal on Selected Areas in Communications*, *16*(2), 140–152.

McGregor, J. J. (1979). Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Information Sciences*, *19*(3), 229–250.

McIlraith, S. und Zeng, H. (2001). Semantic web services. *IEEE Intelligent Systems*, *16*(2), 46–53.

McLachlan, G. J. und Krishnan, T. (1997). *The EM Algorithm and Extensions*. Wiley.

McMillan, K. L. (1993). *Symbolic Model Checking*. Kluwer.

Meehl, P. (1955). *Clinical vs. Statistical Prediction*. University of Minnesota Press.

Mendel, G. (1866). Versuche über Pflanzenhybriden. *Verhandlungen des Naturforschenden Vereins, Abhandlungen, Brünn*, *4*, 3–47. Übersetzt ins Englische von C. T. Druery, veröffentlicht von Bateson (1902).

Mercer, J. (1909). Functions of positive and negative type and their connection with the theory of integral equations. *Philos. Trans. Roy. Soc. London, A*, *209*, 415–446.

Merleau-Ponty, M. (1945). *Phenomenology of Perception*. Routledge.

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A. und Teller, E. (1953). Equations of state calculations by fast computing machines. *J. Chemical Physics*, *21*, 1087–1091.

Metzinger, T. (2009). *The Ego Tunnel: The Science of the Mind and the Myth of the Self*. Basic Books.

Mézard, M. und Nadal, J.-P. (1989). Learning in feedforward layered networks: The tiling algorithm. *J. Physics*, *22*, 2191–2204.

Michalski, R. S. (1969). On the quasi-minimal solution of the general covering problem. In *Proc. First International Symposium on Information Processing*, S. 125–128.

Michalski, R. S., Mozetic, I., Hong, J. und Lavrač, N. (1986). The multi-purpose incremental learning system AQ15 and its testing application to three medical domains. In *AAAI-86*, S. 1041–1045.

Michie, D. (1966). Game-playing and game-learning automata. In Fox, L. (Hrsg.), *Advances in Programming and Non-Numerical Computation*, S. 183–200. Pergamon.

Michie, D. (1972). Machine intelligence at Edinburgh. *Management Informatics*, *2*(1), 7–12.

Michie, D. (1974). Machine intelligence at Edinburgh. In *On Intelligence*, S. 143–155. Edinburgh University Press.

Michie, D. und Chambers, R. A. (1968). BOXES: An experiment in adaptive control. In Dale, E. und Michie, D. (Hrsg.), *Machine Intelligence 2*, S. 125–133. Elsevier/North-Holland.

Michie, D., Spiegelhalter, D. J. und Taylor, C. (Hrsg.). (1994). *Machine Learning, Neural and Statistical Classification*. Ellis Horwood.

Milch, B., Marthi, B., Sontag, D., Russell, S. J., Ong, D. und Kolobov, A. (2005). BLOG: Probabilistic models with unknown objects. In *IJCAI-05*.

Milch, B., Zettlemoyer, L. S., Kersting, K., Haimes, M. und Kaelbling, L. P. (2008). Lifted probabilistic inference with counting formulas. In *AAAI-08*, S. 1062–1068.

Milgrom, P. (1997). Putting auction theory to work: The simultaneous ascending auction. Tech. rep. Technical Report 98-0002, Stanford University Department of Economics.

Mill, J. S. (1843). *A System of Logic, Ratiocinative and Inductive: Being a Connected View of the Principles of Evidence, and Methods of Scientific Investigation*. J. W. Parker, London.

Mill, J. S. (1863). *Utilitarianism*. Parker, Son and Bourn, London.

Miller, A. C., Merkhofer, M. M., Howard, R. A., Matheson, J. E. und Rice, T. R. (1976). Development of automated aids for decision analysis. Technical report, SRI International.

Minker, J. (2001). *Logic-Based Artificial Intelligence*. Kluwer.

Minsky, M. L. (1975). A framework for representing knowledge. In Winston, P. H. (Hrsg.), *The Psychology of Computer Vision*, S. 211–277. McGraw-Hill. Originally an MIT AI Laboratory memo; the 1975 version is abridged, but is the most widely cited.

Minsky, M. L. (1986). *The society of mind*. Simon and Schuster.

Minsky, M. L. (2007). *The Emotion Machine: Commonsense Thinking, Artificial Intelligence, and the Future of the Human Mind*. Simon and Schuster.

Minsky, M. L. und Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry* (1. Auflage). MIT Press.

Minsky, M. L. und Papert, S. (1988). *Perceptrons: An Introduction to Computational Geometry* (Erweiterte Auflage). MIT Press.

Minsky, M. L., Singh, P. und Sloman, A. (2004). The st. thomas common sense symposium: Designing architectures for human-level intelligence. *AIMag*, *25*(2), 113–124.

Minton, S. (1984). Constraint-based generalization: Learning game-playing plans from single examples. In *AAAI-84*, S. 251–254.

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. In *AAAI-88*, S. 564–569.

Minton, S., Johnston, M. D., Philips, A. B. und Laird, P. (1992). Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *AIJ*, *58*(1–3), 161–205.

Misak, C. (2004). *The Cambridge Companion to Peirce*. Cambridge University Press.

Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. MIT Press.

Mitchell, M., Holland, J. H. und Forrest, S. (1996). When will a genetic algorithm outperform hill climbing? In Cowan, J., Tesauro, G. und Alspector, J. (Hrsg.), *NIPS 6*. MIT Press.

Mitchell, T. M. (1977). Version spaces: A candidate elimination approach to rule learning. In *IJCAI-77*, S. 305–310.

Mitchell, T. M. (1982). Generalization as search. *AIJ*, *18*(2), 203–226.

Mitchell, T. M. (1990). Becoming increasingly reactive (mobile robots). In *AAAI-90*, Band 2, S. 1051–1058.

Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.

Mitchell, T. M., Keller, R. und Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, *1*, 47–80.

Mitchell, T. M., Utgoff, P. E. und Banerji, R. (1983). Learning by experimentation: Acquiring and refining problem-solving heuristics. In Michalski, R. S., Carbonell, J. G. und Mitchell, T. M. (Hrsg.), *Machine Learning: An Artificial Intelligence Approach*, S. 163–190. Morgan Kaufmann.

Mitchell, T. M. (2005). Reading the web: A breakthrough goal for AI. *AIMag*, *26*(3), 12–16.

Mitchell, T. M. (2007). Learning, information extraction and the web. In *ECML/PKDD*, p. 1.

Mitchell, T. M., Shinkareva, S. V., Carlson, A., Chang, K.-M., Malave, V. L., Mason, R. A. und Just, M. A. (2008). Predicting human brain activity associated with the meanings of nouns. *Science*, *320*, 1191–1195.

Mohr, R. und Henderson, T. C. (1986). Arc and path consistency revisited. *AIJ*, *28*(2), 225–233.

Mohri, M., Pereira, F. und Riley, M. (2002). Weighted finite-state transducers in speech recognition. *Computer Speech and Language*, *16*(1), 69–88.

Montague, P. R., Dayan, P., Person, C. und Sejnowski, T. (1995). Bee foraging in uncertain environments using predictive Hebbian learning. *Nature*, *377*, 725–728.

Montague, R. (1970). English as a formal language. In *Linguaggi nella Società e nella Tecnica*, S. 189–224. Edizioni di Comunità.

Montague, R. (1973). The proper treatment of quantification in ordinary English. In Hintikka, K. J. J., Moravcsik, J. M. E. und Suppes, P. (Hrsg.), *Approaches to Natural Language*. D. Reidel.

Montanari, U. (1974). Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, *7*(2), 95–132.

Montemerlo, M. und Thrun, S. (2004). Large-scale robotic 3-D mapping of urban structures. In *Proc. International Symposium on Experimental Robotics*. Springer Tracts in Advanced Robotics (STAR).

Montemerlo, M., Thrun, S., Koller, D. und Wegbreit, B. (2002). FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *AAAI-02*.

Mooney, R. (1999). Learning for semantic interpretation: Scaling up without dumbing down. In *Proc. 1st Workshop on Learning Language in Logic*, S. 7–15.

Moore, A. und Wong, W.-K. (2003). Optimal reinsertion: A new search operator for accelerated and more accurate Bayesian network structure learning. In *ICML-03*.

Moore, A. W. und Atkeson, C. G. (1993). Prioritized sweeping – Reinforcement learning with less data and less time. *Machine Learning*, *13*, 103–130.

Moore, A. W. und Lee, M. S. (1997). Cached sufficient statistics for efficient machine learning with large datasets. *JAIR*, *8*, 67–91.

Moore, E. F. (1959). The shortest path through a maze. In *Proc. an International Symposium on the Theory of Switching, Part II*, S. 285–292. Harvard University Press.

Moore, R. C. (1980). Reasoning about knowledge and action. Artificial intelligence center technical note 191, SRI International.

Moore, R. C. (1985). A formal theory of knowledge and action. In Hobbs, J. R. und Moore, R. C. (Hrsg.), *Formal Theories of the Commonsense World*, S. 319–358. Ablex.

Moore, R. C. (2005). Association-based bilingual word alignment. In *Proc. ACL-05 Workshop on Building and Using Parallel Texts*, S. 1–8.

Moravec, H. P. (1983). The stanford cart and the cmu rover. *Proc. IEEE*, *71*(7), 872–884.

Moravec, H. P. und Elfes, A. (1985). High resolution maps from wide angle sonar. In *ICRA-85*, S. 116–121.

Moravec, H. P. (1988). *Mind Children: The Future of Robot and Human Intelligence*. Harvard University Press.

Moravec, H. P. (2000). *Robot: Mere Machine to Transcendent Mind*. Oxford University Press.

Morgenstern, L. (1998). Inheritance comes of age: Applying nonmonotonic techniques to problems in industry. *AIJ*, *103*, 237–271.

Morjaria, M. A., Rink, F. J., Smith, W. D., Klempner, G., Burns, C. und Stein, J. (1995). Elicitation of probabilities for belief networks: Combining qualitative and quantitative information. In *UAI-95*, S. 141–148.

Morrison, P. und Morrison, E. (Hrsg.). (1961). *Charles Babbage and His Calculating Engines: Selected Writings by Charles Babbage and Others*. Dover.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L. und Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proc. 38th Design Automation Conference (DAC 2001)*, S. 530–535.

Mosteller, F. und Wallace, D. L. (1964). *Inference and Disputed Authorship: The Federalist*. Addison-Wesley.

Mostow, J. und Prieditis, A. E. (1989). Discovering admissible heuristics by abstracting and optimizing: A transformational approach. In *IJCAI-89*, Band 1, S. 701–707.

Motzkin, T. S. und Schoenberg, I. J. (1954). The relaxation method for linear inequalities. *Canadian Journal of Mathematics*, *6*(3), 393–404.

Moutarlier, P. und Chatila, R. (1989). Stochastic multisensory data fusion for mobile robot location and environment modeling. In *ISRR-89*.

Mueller, E. T. (2006). *Commonsense Reasoning*. Morgan Kaufmann.

Muggleton, S. H. (1991). Inductive logic programming. *New Generation Computing*, *8*, 295–318.

Muggleton, S. H. (1992). *Inductive Logic Programming*. Academic Press.

Muggleton, S. H. (1995). Inverse entailment and Progol. *New Generation Computing*, *13*(3-4), 245–286.

Muggleton, S. H. (2000). Learning stochastic logic programs. Proc. AAAI 2000 Workshop on Learning Statistical Models from Relational Data.

Muggleton, S. H. und Buntine, W. (1988). Machine invention of first-order predicates by inverting resolution. In *ICML-88*, S. 339–352.

Muggleton, S. H. und De Raedt, L. (1994). Inductive logic programming: Theory and methods. *J. Logic Programming*, *19/20*, 629–679.

Muggleton, S. H. und Feng, C. (1990). Efficient induction of logic programs. In *Proc. Workshop on Algorithmic Learning Theory*, S. 368–381.

Müller, M. (2002). Computer Go. *AIJ*, *134*(1–2), 145–179.

Müller, M. (2003). Conditional combinatorial games, and their application to analyzing capturing races in go. *Information Sciences*, *154*(3–4), 189–202.

Mumford, D. und Shah, J. (1989). Optimal approximations by piece-wise smooth functions and associated variational problems. *Commun. Pure Appl. Math.*, *42*, 577–685.

Murphy, K., Weiss, Y. und Jordan, M. I. (1999). Loopy belief propagation for approximate inference: An empirical study. In *UAI-99*, S. 467–475.

Murphy, K. (2001). The Bayes net toolbox for MATLAB. *Computing Science and Statistics*, *33*.

Murphy, K. (2002). *Dynamic Bayesian Networks: Representation, Inference and Learning*. Ph.D. thesis, UC Berkeley.

Murphy, K. und Mian, I. S. (1999). Modelling gene expression data using Bayesian networks. *people.cs.ubc.ca/˜murphyk/Papers/ismb99.pdf*.

Murphy, K. und Russell, S. J. (2001). Rao-blackwellised particle filtering for dynamic Bayesian networks. In Doucet, A., de Freitas, N. und Gordon, N. J. (Hrsg.), *Sequential Monte Carlo Methods in Practice*. Springer-Verlag.

Murphy, K. und Weiss, Y. (2001). The factored frontier algorithm for approximate inference in DBNs. In *UAI-01*, S. 378–385.

Murphy, R. (2000). *Introduction to AI Robotics*. MIT Press.

Murray-Rust, P., Rzepa, H. S., Williamson, J. und Willighagen, E. L. (2003). Chemical markup, XML and the world–wide web. 4. CML schema. *J. Chem. Inf. Comput. Sci.*, *43*, 752–772.

Murthy, C. und Russell, J. R. (1990). A constructive proof of Higman's lemma. In *LICS-90*, S. 257–269.

Muscettola, N. (2002). Computing the envelope for stepwise-constant resource allocations. In *CP-02*, S. 139–154.

Muscettola, N., Nayak, P., Pell, B. und Williams, B. (1998). Remote agent: To boldly go where no AI system has gone before. *AIJ*, *103*, 5–48.

Muslea, I. (1999). Extraction patterns for information extraction tasks: A survey. In *Proc. AAAI-99 Workshop on Machine Learning for Information Extraction*.

Myerson, R. (1981). Optimal auction design. *Mathematics of Operations Research*, *6*, 58–73.

Myerson, R. (1986). Multistage games with communication. *Econometrica*, *54*, 323–358.

Myerson, R. (1991). *Game Theory: Analysis of Conflict*. Harvard University Press.

Nagel, T. (1974). What is it like to be a bat? *Philosophical Review*, *83*, 435–450.

Nalwa, V. S. (1993). *A Guided Tour of Computer Vision*. Addison-Wesley.

Nash, J. (1950). Equilibrium points in N-person games. *PNAS*, *36*, 48–49.

Nau, D. S. (1980). Pathology on game trees: A summary of results. In *AAAI-80*, S. 102–104.

Nau, D. S. (1983). Pathology on game trees revisited und an alternative to minimaxing. *AIJ*, *21*(1–2), 221–244.

Nau, D. S., Kumar, V. und Kanal, L. N. (1984). General branch and bound, and its relation to A* and AO*. *AIJ*, *23*, 29–58.

Nayak, P. und Williams, B. (1997). Fast context switching in real-time propositional reasoning. In *AAAI-97*, S. 50–56.

Neal, R. (1996). *Bayesian Learning for Neural Networks*. Springer-Verlag.

Nebel, B. (2000). On the compilability and expressive power of propositional planning formalisms. *JAIR*, *12*, 271–315.

Nefian, A., Liang, L., Pi, X., Liu, X. und Murphy, K. (2002). Dynamic bayesian networks for audio-visual speech recognition. *EURASIP, Journal of Applied Signal Processing*, *11*, 1–15.

Nesterov, Y. und Nemirovski, A. (1994). *Interior-Point Polynomial Methods in Convex Programming*. SIAM (Society for Industrial and Applied Mathematics).

Netto, E. (1901). *Lehrbuch der Combinatorik*. B. G. Teubner.

Nevill-Manning, C. G. und Witten, I. H. (1997). Identifying hierarchical structures in sequences: A linear-time algorithm. *JAIR*, *7*, 67–82.

Newell, A. (1982). The knowledge level. *AIJ*, *18*(1), 82–127.

Newell, A. (1990). *Unified Theories of Cognition*. Harvard University Press.

Newell, A. und Ernst, G. (1965). The search for generality. In *Proc. IFIP Congress*, Band 1, S. 17–24.

Newell, A., Shaw, J. C. und Simon, H. A. (1957). Empirical explorations with the logic theory machine. *Proc. Western Joint Computer Conference*, *15*, 218–239. Reprinted in Feigenbaum and Feldman (1963).

Newell, A., Shaw, J. C. und Simon, H. A. (1958). Chess playing programs and the problem of complexity. *IBM Journal of Research and Development*, *4*(2), 320–335.

Newell, A. und Simon, H. A. (1961). GPS, a program that simulates human thought. In Billing, H. (Hrsg.), *Lernende Automaten*, S. 109–124. R. Oldenbourg.

Newell, A. und Simon, H. A. (1972). *Human Problem Solving*. Prentice-Hall.

Newell, A. und Simon, H. A. (1976). Computer science as empirical inquiry: Symbols and search. *CACM*, *19*, 113–126.

Newton, I. (1664–1671). Methodus fluxionum et serierum infinitarum. Unpublished notes.

Ng, A. Y. (2004). Feature selection, l_1 vs. l_2 regularization, and rotational invariance. In *ICML-04*.

Ng, A. Y., Harada, D. und Russell, S. J. (1999). Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML-99*.

Ng, A. Y. und Jordan, M. I. (2000). PEGASUS: A policy search method for large MDPs and POMDPs. In *UAI-00*, S. 406–415.

Ng, A. Y., Kim, H. J., Jordan, M. I. und Sastry, S. (2004). Autonomous helicopter flight via reinforcement learning. In *NIPS 16*.

Nguyen, X. und Kambhampati, S. (2001). Reviving partial order planning. In *IJCAI-01*, S. 459–466.

Nguyen, X., Kambhampati, S. und Nigenda, R. S. (2001). Planning graph as the basis for deriving heuristics for plan synthesis by state space and CSP search. Tech. rep., Computer Science and Engineering Department, Arizona State University.

Nicholson, A. und Brady, J. M. (1992). The data association problem when monitoring robot vehicles using dynamic belief networks. In *ECAI-92*, S. 689–693.

Niemelä, I., Simons, P. und Syrjänen, T. (2000). Smodels: A system for answer set programming. In *Proc. 8th International Workshop on Non-Monotonic Reasoning*.

Nigam, K., McCallum, A., Thrun, S. und Mitchell, T. M. (2000). Text classification from labeled and unlabeled documents using EM. *Machine Learning*, *39*(2–3), 103–134.

Niles, I. und Pease, A. (2001). Towards a standard upper ontology. In *FOIS '01: Proc. international conference on Formal Ontology in Information Systems*, S. 2–9.

Nilsson, D. und Lauritzen, S. (2000). Evaluating influence diagrams using LIMIDs. In *UAI-00*, S. 436–445.

Nilsson, N. J. (1965). *Learning Machines: Foundations of Trainable Pattern-Classifying Systems*. McGraw-Hill. Neu veröffentlicht in 1990.

Nilsson, N. J. (1971). *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill.

Nilsson, N. J. (1984). Shakey the robot. Technical note 323, SRI International.

Nilsson, N. J. (1986). Probabilistic logic. *AIJ*, *28*(1), 71–87.

Nilsson, N. J. (1991). Logic and artificial intelligence. *AIJ*, *47*(1–3), 31–56.

Nilsson, N. J. (1995). Eye on the prize. *AIMag*, *16*(2), 9–17.

Nilsson, N. J. (1998). *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann.

Nilsson, N. J. (2005). Human-level artificial intelligence? be serious! *AIMag*, *26*(4), 68–75.

Nilsson, N. J. (2009). *The Quest for Artificial Intelligence: A History of Ideas and Achievements*. Cambridge University Press.

Nisan, N., Roughgarden, T., Tardos, E. und Vazirani, V. (Hrsg.). (2007). *Algorithmic Game Theory*. Cambridge University Press.

Noe, A. (2009). *Out of Our Heads: Why You Are Not Your Brain, and Other Lessons from the Biology of Consciousness*. Hill and Wang.

Norvig, P. (1988). Multiple simultaneous interpretations of ambiguous sentences. In *COGSCI-88*.

Norvig, P. (1992). *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann.

Norvig, P. (2009). Natural language corpus data. In Segaran, T. und Hammerbacher, J. (Hrsg.), *Beautiful Data*. O'Reilly.

Nowick, S. M., Dean, M. E., Dill, D. L. und Horowitz, M. (1993). The design of a high-performance cache controller: A case study in asynchronous synthesis. *Integration: The VLSI Journal*, *15*(3), 241–262.

Nunberg, G. (1979). The non-uniqueness of semantic solutions: Polysemy. *Language and Philosophy*, *3*(2), 143–184.

Nussbaum, M. C. (1978). *Aristotle's* De Motu Animalium. Princeton University Press.

Oaksford, M. und Chater, N. (Hrsg.). (1998). *Rational models of cognition*. Oxford University Press.

Och, F. J. und Ney, H. (2003). A systematic comparison of various statistical alignment model. *Computational Linguistics*, *29*(1), 19–51.

Och, F. J. und Ney, H. (2004). The alignment template approach to statistical machine translation. *Computational Linguistics*, *30*, 417–449.

Ogawa, S., Lee, T.-M., Kay, A. R. und Tank, D. W. (1990). Brain magnetic resonance imaging with contrast dependent on blood oxygenation. *PNAS*, *87*, 9868–9872.

Oh, S., Russell, S. J. und Sastry, S. (2009). Markov chain Monte Carlo data association for multi-target tracking. *IEEE Transactions on Automatic Control*, *54*(3), 481–497.

Olesen, K. G. (1993). Causal probabilistic networks with both discrete and continuous variables. *PAMI*, *15*(3), 275–279.

Oliver, N., Garg, A. und Horvitz, E. J. (2004). Layered representations for learning and inferring office activity from multiple sensory channels. *Computer Vision and Image Understanding*, *96*, 163–180.

Oliver, R. M. und Smith, J. Q. (Hrsg.). (1990). *Influence Diagrams, Belief Nets and Decision Analysis*. Wiley.

Omohundro, S. (2008). The basic AI drives. In *AGI-08 Workshop on the Sociocultural, Ethical and Futurological Implications of Artificial Intelligence*.

O'Reilly, U.-M. und Oppacher, F. (1994). Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. In *Proc. Third Conference on Parallel Problem Solving from Nature*, S. 397–406.

Ormoneit, D. und Sen, S. (2002). Kernel-based reinforcement learning. *Machine Learning*, *49*(2–3), 161–178.

Osborne, M. J. (2004). *An Introduction to Game Theory*. Oxford University Pres.

Osborne, M. J. und Rubinstein, A. (1994). *A Course in Game Theory*. MIT Press.

Osherson, D. N., Stob, M. und Weinstein, S. (1986). *Systems That Learn: An Introduction to Learning Theory for Cognitive and Computer Scientists*. MIT Press.

Padgham, L. und Winikoff, M. (2004). *Developing Intelligent Agent Systems: A Practical Guide*. Wiley.

Page, C. D. und Srinivasan, A. (2002). ILP: A short look back and a longer look forward. Submitted to Journal of Machine Learning Research.

Palacios, H. und Geffner, H. (2007). From conformant into classical planning: Efficient translations that may be complete too. In *ICAPS-07*.

Palay, A. J. (1985). *Searching with Probabilities*. Pitman.

Palmer, D. A. und Hearst, M. A. (1994). Adaptive sentence boundary disambiguation. In *Proc. Conference on Applied Natural Language Processing*, S. 78–83.

Palmer, S. (1999). *Vision Science: Photons to Phenomenology*. MIT Press.

Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

Papadimitriou, C. H., Tamaki, H., Raghavan, P. und Vempala, S. (1998). Latent semantic indexing: A probabilistic analysis. In *PODS-98*, S. 159–168.

Papadimitriou, C. H. und Tsitsiklis, J. N. (1987). The complexity of Markov decision processes. *Mathematics of Operations Research*, *12*(3), 441–450.

Papadimitriou, C. H. und Yannakakis, M. (1991). Shortest paths without a map. *Theoretical Computer Science*, *84*(1), 127–150.

Papavassiliou, V. und Russell, S. J. (1999). Convergence of reinforcement learning with general function approximators. In *IJCAI-99*, S. 748–757.

Parekh, R. und Honavar, V. (2001). DFA learning from simple examples. *Machine Learning*, *44*, 9–35.

Parisi, G. (1988). *Statistical field theory*. Addison-Wesley.

Parisi, M. M. G. und Zecchina, R. (2002). Analytic and algorithmic solution of random satisfiability problems. *Science*, *297*, 812–815.

Parker, A., Nau, D. S. und Subrahmanian, V. S. (2005). Game-tree search with combinatorially large belief states. In *IJCAI-05*, S. 254–259.

Parker, D. B. (1985). Learning logic. Technical report TR-47, Center for Computational Research in Economics and Management Science, Massachusetts Institute of Technology.

Parker, L. E. (1996). On the design of behavior-based multi-robot teams. *J. Advanced Robotics*, *10*(6).

Parr, R. und Russell, S. J. (1998). Reinforcement learning with hierarchies of machines. In Jordan, M. I., Kearns, M. und Solla, S. A. (Hrsg.), *NIPS 10*. MIT Press.

Parzen, E. (1962). On estimation of a probability density function and mode. *Annals of Mathematical Statistics*, *33*, 1065–1076.

Pasca, M. und Harabagiu, S. M. (2001). High performance question/answering. In *SIGIR-01*, S. 366–374.

Pasca, M., Lin, D., Bigham, J., Lifchits, A. und Jain, A. (2006). Organizing and searching the world wide web of facts – Step one: The one-million fact extraction challenge. In *AAAI-06*.

Paskin, M. (2001). Grammatical bigrams. In *NIPS*.

Pasula, H., Marthi, B., Milch, B., Russell, S. J. und Shpitser, I. (2003). Identity uncertainty and citation matching. In *NIPS 15*. MIT Press.

Pasula, H. und Russell, S. J. (2001). Approximate inference for first-order probabilistic languages. In *IJCAI-01*.

Pasula, H., Russell, S. J., Ostland, M. und Ritov, Y. (1999). Tracking many objects with many sensors. In *IJCAI-99*.

Patashnik, O. (1980). Qubic: 4x4x4 tic-tac-toe. *Mathematics Magazine*, *53*(4), 202–216.

Patrick, B. G., Almulla, M. und Newborn, M. (1992). An upper bound on the time complexity of iterative-deepening-A*. *AIJ*, *5*(2–4), 265–278.

Paul, R. P. (1981). *Robot Manipulators: Mathematics, Programming, and Control*. MIT Press.

Pauls, A. und Klein, D. (2009). K-best A* parsing. In *ACL-09*.

Peano, G. (1889). *Arithmetices principia, nova methodo exposita*. Fratres Bocca, Turin.

Pearce, J., Tambe, M. und Maheswaran, R. (2008). Solving multiagent networks using distributed constraint optimization. *AIMag*, *29*(3), 47–62.

Pearl, J. (1982a). Reverend Bayes on inference engines: A distributed hierarchical approach. In *AAAI-82*, S. 133–136.

Pearl, J. (1982b). The solution for the branching factor of the alpha–beta pruning algorithm and its optimality. *CACM*, *25*(8), 559–564.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

Pearl, J. (1986). Fusion, propagation, and structuring in belief networks. *AIJ*, *29*, 241–288.

Pearl, J. (1987). Evidential reasoning using stochastic simulation of causal models. *AIJ*, *32*, 247–257.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann.

Pearl, J. (2000). *Causality: Models, Reasoning, and Inference*. Cambridge University Press.

Pearl, J. und Verma, T. (1991). A theory of inferred causation. In *KR-91*, S. 441–452.

Pearson, J. und Jeavons, P. (1997). A survey of tractable constraint satisfaction problems. Technical report CSD-TR-97-15, Royal Holloway College, U. of London.

Pease, A. und Niles, I. (2002). IEEE standard upper ontology: A progress report. *Knowledge Engineering Review*, *17*(1), 65–70.

Pednault, E. P. D. (1986). Formulating multiagent, dynamic-world problems in the classical planning framework. In *Reasoning about Actions and Plans: Proc. 1986 Workshop*, S. 47–82.

Peirce, C. S. (1870). Description of a notation for the logic of relatives, resulting from an amplification of the conceptions of Boole's calculus of logic. *Memoirs of the American Academy of Arts and Sciences*, *9*, 317–378.

Peirce, C. S. (1883). A theory of probable inference. Note B. The logic of relatives. In *Studies in Logic by Members of the Johns Hopkins University*, S. 187–203, Boston.

Peirce, C. S. (1902). Logic as semiotic: The theory of signs. Unpublished manuscript; reprinted in (Buchler 1955).

Peirce, C. S. (1909). Existential graphs. Unpublished manuscript; reprinted in (Buchler 1955).

Pelikan, M., Goldberg, D. E. und Cantu-Paz, E. (1999). BOA: The Bayesian optimization algorithm. In *GECCO-99: Proc. Genetic and Evolutionary Computation Conference*, S. 525–532.

Pemberton, J. C. und Korf, R. E. (1992). Incremental planning on graphs with cycles. In *AIPS-92*, S. 525–532.

Penberthy, J. S. und Weld, D. S. (1992). UCPOP: A sound, complete, partial order planner for ADL. In *KR-92*, S. 103–114.

Peng, J. und Williams, R. J. (1993). Efficient learning and planning within the Dyna framework. *Adaptive Behavior*, *2*, 437–454.

Penrose, R. (1989). *The Emperor's New Mind*. Oxford University Press.

Penrose, R. (1994). *Shadows of the Mind*. Oxford University Press.

Peot, M. und Smith, D. E. (1992). Conditional nonlinear planning. In *ICAPS-92*, S. 189–197.

Pereira, F. und Shieber, S. (1987). *Prolog and Natural-Language Analysis*. Center for the Study of Language and Information (CSLI).

Pereira, F. und Warren, D. H. D. (1980). Definite clause grammars for language analysis: A survey of the formalism and a comparison with augmented transition networks. *AIJ*, *13*, 231–278.

Pereira, F. und Wright, R. N. (1991). Finite-state approximation of phrase structure grammars. In *ACL-91*, S. 246–255.

Perlis, A. (1982). Epigrams in programming. *SIGPLAN Notices*, *17*(9), 7–13.

Perrin, B. E., Ralaivola, L. und Mazurie, A. (2003). Gene networks inference using dynamic Bayesian networks. *Bioinformatics*, *19*, II 138–II 148.

Peterson, C. und Anderson, J. R. (1987). A mean field theory learning algorithm for neural networks. *Complex Systems*, *1*(5), 995–1019.

Petrik, M. und Zilberstein, S. (2009). Bilinear programming approach for multiagent planning. *JAIR*, *35*, 235–274.

Petrov, S. und Klein, D. (2007a). Discriminative log-linear grammars with latent variables. In *NIPS*.

Petrov, S. und Klein, D. (2007b). Improved inference for unlexicalized parsing. In *ACL-07*.

Petrov, S. und Klein, D. (2007c). Learning and inference for hierarchically split pcfgs. In *AAAI-07*.

Pfeffer, A., Koller, D., Milch, B. und Takusagawa, K. T. (1999). SPOOK: A system for probabilistic object-oriented knowledge representation. In *UAI-99*.

Pfeffer, A. (2000). *Probabilistic Reasoning for Complex Systems*. Ph.D. thesis, Stanford University.

Pfeffer, A. (2007). The design and implementation of IBAL: A general-purpose probabilistic language. In Getoor, L. und Taskar, B. (Hrsg.), *Introduction to Statistical Relational Learning*. MIT Press.

Pfeifer, R., Bongard, J., Brooks, R. A. und Iwasawa, S. (2006). *How the Body Shapes the Way We Think: A New View of Intelligence*. Bradford.

Pineau, J., Gordon, G. und Thrun, S. (2003). Point-based value iteration: An anytime algorithm for POMDPs. In *IJCAI-03*.

Pinedo, M. (2008). *Scheduling: Theory, Algorithms, and Systems*. Springer Verlag.

Pinkas, G. und Dechter, R. (1995). Improving connectionist energy minimization. *JAIR*, *3*, 223–248.

Pinker, S. (1995). Language acquisition. In Gleitman, L. R., Liberman, M. und Osherson, D. N. (Hrsg.), *An Invitation to Cognitive Science* (2. Auflage)., Band 1. MIT Press.

Pinker, S. (2003). *The Blank Slate: The Modern Denial of Human Nature*. Penguin.

Pinto, D., McCallum, A., Wei, X. und Croft, W. B. (2003). Table extraction using conditional random fields. In *SIGIR-03*.

Pipatsrisawat, K. und Darwiche, A. (2007). RSat 2.0: SAT solver description. Tech. rep. D–153, Automated Reasoning Group, Computer Science Department, University of California, Los Angeles.

Plaat, A., Schaeffer, J., Pijls, W. und de Bruin, A. (1996). Best-first fixed-depth minimax algorithms. *AIJ*, *87*(1–2), 255–293.

Place, U. T. (1956). Is consciousness a brain process? *British Journal of Psychology*, *47*, 44–50.

Platt, J. (1999). Fast training of support vector machines using sequential minimal optimization. In *Advances in Kernel Methods: Support Vector Learning*, S. 185–208. MIT Press.

Plotkin, G. (1971). *Automatic Methods of Inductive Inference*. Ph.D. thesis, Edinburgh University.

Plotkin, G. (1972). Building-in equational theories. In Meltzer, B. und Michie, D. (Hrsg.), *Machine Intelligence 7*, S. 73–90. Edinburgh University Press.

Pohl, I. (1971). Bi-directional search. In Meltzer, B. und Michie, D. (Hrsg.), *Machine Intelligence 6*, S. 127–140. Edinburgh University Press.

Pohl, I. (1973). The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *IJCAI-73*, S. 20–23.

Pohl, I. (1977). Practical and theoretical considerations in heuristic search algorithms. In Elcock, E. W. und Michie, D. (Hrsg.), *Machine Intelligence 8*, S. 55–72. Ellis Horwood.

Poli, R., Langdon, W. und McPhee, N. (2008). *A Field Guide to Genetic Programming*. Lulu.com.

Pomerleau, D. A. (1993). *Neural Network Perception for Mobile Robot Guidance*. Kluwer.

Ponte, J. und Croft, W. B. (1998). A language modeling approach to information retrieval. In *SIGIR-98*, S. 275–281.

Poole, D. (1993). Probabilistic Horn abduction and Bayesian networks. *AIJ*, *64*, 81–129.

Poole, D. (2003). First-order probabilistic inference. In *IJCAI-03*, S. 985–991.

Poole, D., Mackworth, A. K. und Goebel, R. (1998). *Computational intelligence: A logical approach*. Oxford University Press.

Popper, K. R. (1959). *The Logic of Scientific Discovery*. Basic Books.

Popper, K. R. (1962). *Conjectures and Refutations: The Growth of Scientific Knowledge*. Basic Books.

Portner, P. und Partee, B. H. (2002). *Formal Semantics: The Essential Readings*. Wiley-Blackwell.

Post, E. L. (1921). Introduction to a general theory of elementary propositions. *American Journal of Mathematics*, *43*, 163–185.

Poundstone, W. (1993). *Prisoner's Dilemma*. Anchor.

Pourret, O., Na¨ım, P. und Marcot, B. (2008). *Bayesian Networks: A practical guide to applications*. Wiley.

Prades, J. L. P., Loomes, G. und Brey, R. (2008). Trying to estmate a monetary value for the QALY. Tech. rep. WP Econ 08.09, Univ. Pablo Olavide.

Pradhan, M., Provan, G. M., Middleton, B. und Henrion, M. (1994). Knowledge engineering for large belief networks. In *UAI-94*, S. 484–490.

Prawitz, D. (1960). An improved proof procedure. *Theoria*, *26*, 102–139.

Press, W. H., Teukolsky, S. A., Vetterling, W. T. und Flannery, B. P. (2007). *Numerical Recipes: The Art of Scientific Computing* (3. Auflage). Cambridge University Press.

Preston, J. und Bishop, M. (2002). *Views into the Chinese Room: New Essays on Searle and Artificial Intelligence*. Oxford University Press.

Prieditis, A. E. (1993). Machine discovery of effective admissible heuristics. *Machine Learning*, *12*(1–3), 117–141.

Prinz, D. G. (1952). Robot chess. *Research*, *5*, 261–266.

Prosser, P. (1993). Hybrid algorithms for constraint satisfaction problems. *Computational Intelligence*, *9*, 268–299.

Pullum, G. K. (1991). *The Great Eskimo Vocabulary Hoax (and Other Irreverent Essays on the Study of Language)*. University of Chicago Press.

Pullum, G. K. (1996). Learnability, hyperlearning, and the poverty of the stimulus. In *22nd Annual Meeting of the Berkeley Linguistics Society*.

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley.

Puterman, M. L. und Shin, M. C. (1978). Modified policy iteration algorithms for discounted Markov decision problems. *Management Science*, *24*(11), 1127–1137.

Putnam, H. (1960). Minds and machines. In Hook, S. (Hrsg.), *Dimensions of Mind*, S. 138–164. Macmillan.

Putnam, H. (1963). 'Degree of confirmation' and inductive logic. In Schilpp, P. A. (Hrsg.), *The Philosophy of Rudolf Carnap*, S. 270–292. Open Court.

Putnam, H. (1967). The nature of mental states. In Capitan, W. H. und Merrill, D. D. (Hrsg.), *Art, Mind, and Religion*, S. 37–48. University of Pittsburgh Press.

Putnam, H. (1975). The meaning of "meaning". In Gunderson, K. (Hrsg.), *Language, Mind and Knowledge: Minnesota Studies in the Philosophy of Science*. University of Minnesota Press.

Pylyshyn, Z. W. (1974). Minds, machines and phenomenology: Some reflections on Dreyfus' "What Computers Can't Do". *Int. J. Cognitive Psychology*, *3*(1), 57–77.

Pylyshyn, Z. W. (1984). *Computation and Cognition: Toward a Foundation for Cognitive Science*. MIT Press.

Quillian, M. R. (1961). A design for an understanding machine. Paper presented at a colloquium: Semantic Problems in Natural Language, King's College, Cambridge, England.

Quine, W. V. (1953). Two dogmas of empiricism. In *From a Logical Point of View*, S. 20–46. Harper and Row.

Quine, W. V. (1960). *Word and Object*. MIT Press.

Quine, W. V. (1982). *Methods of Logic* (4. Auflage). Harvard University Press.

Quinlan, J. R. (1979). Discovering rules from large collections of examples: A case study. In Michie, D. (Hrsg.), *Expert Systems in the Microelectronic Age*. Edinburgh University Press.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, *1*, 81–106.

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning*, *5*(3), 239–266.

Quinlan, J. R. (1993). *C4.5: Programs for machine learning*. Morgan Kaufmann.

Quinlan, J. R. und Cameron-Jones, R. M. (1993). FOIL: A midterm report. In *ECML-93*, S. 3–20.

Quirk, R., Greenbaum, S., Leech, G. und Svartvik, J. (1985). *A Comprehensive Grammar of the English Language*. Longman.

Rabani, Y., Rabinovich, Y. und Sinclair, A. (1998). A computational view of population genetics. *Random Structures and Algorithms*, *12*(4), 313–334.

Rabiner, L. R. und Juang, B.-H. (1993). *Fundamentals of Speech Recognition*. Prentice-Hall.

Ralphs, T. K., Ladanyi, L. und Saltzman, M. J. (2004). A library hierarchy for implementing scalable parallel search algorithms. *J. Supercomputing*, *28*(2), 215–234.

Ramanan, D., Forsyth, D. und Zisserman, A. (2007). Tracking people by learning their appearance. *IEEE Pattern Analysis and Machine Intelligence*.

Ramsey, F. P. (1931). Truth and probability. In Braithwaite, R. B. (Hrsg.), *The Foundations of Mathematics and Other Logical Essays*. Harcourt Brace Jovanovich.

Ranzato, M., Poultney, C., Chopra, S. und LeCun, Y. (2007). Efficient learning of sparse representations with an energy-based model. In *NIPS 19*, S. 1137–1144.

Raphson, J. (1690). *Analysis aequationum universalis*. Apud Abelem Swalle, London.

Rashevsky, N. (1936). Physico-mathematical aspects of excitation and conduction in nerves. In *Cold Springs Harbor Symposia on Quantitative Biology. IV: Excitation Phenomena*, S. 90–97.

Rashevsky, N. (1938). *Mathematical Biophysics: Physico-Mathematical Foundations of Biology*. University of Chicago Press.

Rasmussen, C. E. und Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning*. MIT Press.

Rassenti, S., Smith, V. und Bulfin, R. (1982). A combinatorial auction mechanism for airport time slot allocation. *Bell Journal of Economics*, *13*, 402–417.

Ratner, D. und Warmuth, M. (1986). Finding a shortest solution for the n × n extension of the 15-puzzle is intractable. In *AAAI-86*, Band 1, S. 168–172.

Rauch, H. E., Tung, F. und Striebel, C. T. (1965). Maximum likelihood estimates of linear dynamic systems. *AIAA Journal*, *3*(8), 1445–1450.

Rayward-Smith, V., Osman, I., Reeves, C. und Smith, G. (Hrsg.). (1996). *Modern Heuristic Search Methods*. Wiley.

Rechenberg, I. (1965). Cybernetic solution path of an experimental problem. Library translation 1122, Royal Aircraft Establishment.

Reeson, C. G., Huang, K.-C., Bayer, K. M. und Choueiry, B. Y. (2007). An interactive constraint-based approach to sudoku. In *AAAI-07*, S. 1976–1977.

Regin, J. (1994). A filtering algorithm for constraints of difference in CSPs. In *AAAI-94*, S. 362–367.

Reichenbach, H. (1949). *The Theory of Probability: An Inquiry into the Logical and Mathematical Foundations of the Calculus of Probability* (2. Auflage). University of California Press.

Reid, D. B. (1979). An algorithm for tracking multiple targets. *IEEE Trans. Automatic Control*, *24*(6), 843–854.

Reif, J. (1979). Complexity of the mover's problem and generalizations. In *FOCS-79*, S. 421–427. IEEE.

Reiter, R. (1980). A logic for default reasoning. *AIJ*, *13*(1–2), 81–132.

Reiter, R. (1991). The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In Lifschitz, V. (Hrsg.), *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, S. 359–380. Academic Press.

Reiter, R. (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.

Renner, G. und Ekart, A. (2003). Genetic algorithms in computer aided design. *Computer Aided Design*, *35*(8), 709–726.

Rényi, A. (1970). *Probability Theory*. Elsevier/North-Holland.

Reynolds, C. W. (1987). Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, *21*, 25–34. SIGGRAPH '87 Conference Proceedings.

Riazanov, A. und Voronkov, A. (2002). The design and implementation of VAMPIRE. *AI Communications*, *15*(2–3), 91–110.

Rich, E. und Knight, K. (1991). *Artificial Intelligence* (2. Auflage). McGraw-Hill.

Richards, M. und Amir, E. (2007). Opponent modeling in Scrabble. In *IJCAI-07*.

Richardson, M., Bilmes, J. und Diorio, C. (2000). Hidden-articulator Markov models: Performance improvements and robustness to noise. In *ICASSP-00*.

Richter, S. und Westphal, M. (2008). The LAMA planner. In *Proc. International Planning Competition at ICAPS*.

Ridley, M. (2004). *Evolution*. Oxford Reader.

Rieger, C. (1976). An organization of knowledge for problem solving and language comprehension. *AIJ*, *7*, 89–127.

Riley, J. und Samuelson, W. (1981). Optimal auctions. *American Economic Review*, *71*, 381–392.

Riloff, E. (1993). Automatically constructing a dictionary for information extraction tasks. In *AAAI-93*, S. 811–816.

Rintanen, J. (1999). Improvements to the evaluation of quantified Boolean formulae. In *IJCAI-99*, S. 1192–1197.

Rintanen, J. (2007). Asymptotically optimal encodings of conformant planning in QBF. In *AAAI-07*, S. 1045–1050.

Ripley, B. D. (1996). *Pattern Recognition and Neural Networks*. Cambridge University Press.

Rissanen, J. (1984). Universal coding, information, prediction, and estimation. *IEEE Transactions on Information Theory*, *IT-30*(4), 629–636.

Rissanen, J. (2007). *Information and Complexity in Statistical Modeling*. Springer.

Ritchie, G. D. und Hanna, F. K. (1984). AM: A case study in AI methodology. *AIJ*, *23*(3), 249–268.

Rivest, R. (1987). Learning decision lists. *Machine Learning*, *2*(3), 229–246.

Roberts, L. G. (1963). Machine perception of three-dimensional solids. Technical report 315, MIT Lincoln Laboratory.

Robertson, N. und Seymour, P. D. (1986). Graph minors. II. Algorithmic aspects of tree-width. *J. Algorithms*, *7*(3), 309–322.

Robertson, S. E. (1977). The probability ranking principle in IR. *J. Documentation*, *33*, 294–304.

Robertson, S. E. und Sparck Jones, K. (1976). Relevance weighting of search terms. *J. American Society for Information Science*, *27*, 129–146.

Robinson, A. und Voronkov, A. (2001). *Handbook of Automated Reasoning*. Elsevier.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *JACM*, *12*, 23–41.

Roche, E. und Schabes, Y. (1997). *Finite-State Language Processing (Language, Speech and Communication)*. Bradford Books.

Rock, I. (1984). *Perception*. W. H. Freeman.

Rosenblatt, F. (1957). The perceptron: A perceiving and recognizing automaton. Report 85-460-1, Project PARA, Cornell Aeronautical Laboratory.

Rosenblatt, F. (1960). On the convergence of reinforcement procedures in simple perceptrons. Report VG-1196-G-4, Cornell Aeronautical Laboratory.

Rosenblatt, F. (1962). *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan.

Rosenblatt, M. (1956). Remarks on some nonparametric estimates of a density function. *Annals of Mathematical Statistics*, *27*, 832–837.

Rosenblueth, A., Wiener, N. und Bigelow, J. (1943). Behavior, purpose, and teleology. *Philosophy of Science*, *10*, 18–24.

Rosenschein, J. S. und Zlotkin, G. (1994). *Rules of Encounter*. MIT Press.

Rosenschein, S. J. (1985). Formal theories of knowledge in AI and robotics. *New Generation Computing*, *3*(4), 345–357.

Ross, P. E. (2004). Psyching out computer chess players. *IEEE Spectrum*, *41*(2), 14–15.

Ross, S. M. (1988). *A First Course in Probability* (3. Auflage). Macmillan.

Rossi, F., van Beek, P. und Walsh, T. (2006). *Handbook of Constraint Processing*. Elsevier.

Roussel, P. (1975). Prolog: Manual de reference et d'utilization. Tech. rep., Groupe d'Intelligence Artificielle, Université d'Aix-Marseille.

Rouveirol, C. und Puget, J.-F. (1989). A simple and general solution for inverting resolution. In *Proc. European Working Session on Learning*, S. 201–210.

Rowat, P. F. (1979). *Representing the Spatial Experience and Solving Spatial problems in a Simulated Robot Environment*. Ph.D. thesis, University of British Columbia.

Roweis, S. T. und Ghahramani, Z. (1999). A unifying review of Linear Gaussian Models. *Neural Computation*, *11*(2), 305–345.

Rowley, H., Baluja, S. und Kanade, T. (1996). Neural network-based face detection. In *CVPR*, S. 203–208.

Roy, N., Gordon, G. und Thrun, S. (2005). Finding approximate POMDP solutions through belief compression. *JAIR*, *23*, 1–40.

Rubin, D. (1988). Using the SIR algorithm to simulate posterior distributions. In Bernardo, J. M., de Groot, M. H., Lindley, D. V. und Smith, A. F. M. (Hrsg.), *Bayesian Statistics 3*, S. 395–402. Oxford University Press.

Rumelhart, D. E., Hinton, G. E. und Williams, R. J. (1986a). Learning internal representations by error propagation. In Rumelhart, D. E. und McClelland, J. L. (Hrsg.), *Parallel Distributed Processing*, Band 1, Kap. 8, S. 318–362. MIT Press.

Rumelhart, D. E., Hinton, G. E. und Williams, R. J. (1986b). Learning representations by back-propagating errors. *Nature*, *323*, 533–536.

Rumelhart, D. E. und McClelland, J. L. (Hrsg.). (1986). *Parallel Distributed Processing*. MIT Press.

Rummery, G. A. und Niranjan, M. (1994). Online Q-learning using connectionist systems. Tech. rep. CUED/F-INFENG/TR 166, Cambridge University Engineering Department.

Ruspini, E. H., Lowrance, J. D. und Strat, T. M. (1992). Understanding evidential reasoning. *IJAR*, *6*(3), 401–424.

Russell, J. G. B. (1990). Is screening for abdominal aortic aneurysm worthwhile? *Clinical Radiology*, *41*, 182–184.

Russell, S. J. (1985). The compleat guide to MRS. Report STAN-CS-85-1080, Computer Science Department, Stanford University.

Russell, S. J. (1986). A quantitative analysis of analogy by similarity. In *AAAI-86*, S. 284–288.

Russell, S. J. (1988). Tree-structured bias. In *AAAI-88*, Band 2, S. 641–645.

Russell, S. J. (1992). Efficient memory-bounded search methods. In *ECAI-92*, S. 1–5.

Russell, S. J. (1998). Learning agents for uncertain environments (extended abstract). In *COLT-98*, S. 101–103.

Russell, S. J., Binder, J., Koller, D. und Kanazawa, K. (1995). Local learning in probabilistic networks with hidden variables. In *IJCAI-95*, S. 1146–52.

Russell, S. J. und Grosof, B. (1987). A declarative approach to bias in concept learning. In *AAAI-87*.

Russell, S. J. und Norvig, P. (2003). *Artificial Intelligence: A Modern Approach* (2. Auflage). Prentice-Hall.

Russell, S. J. und Subramanian, D. (1995). Provably bounded-optimal agents. *JAIR*, *3*, 575–609.

Russell, S. J., Subramanian, D. und Parr, R. (1993). Provably bounded optimal agents. In *IJCAI-93*, S. 338–345.

Russell, S. J. und Wefald, E. H. (1989). On optimal game-tree search using rational meta-reasoning. In *IJCAI-89*, S. 334–340.

Russell, S. J. und Wefald, E. H. (1991). *Do the Right Thing: Studies in Limited Rationality*. MIT Press.

Russell, S. J. und Wolfe, J. (2005). Efficient belief-state AND-OR search, with applications to Kriegspiel. In *IJCAI-05*, S. 278–285.

Russell, S. J. und Zimdars, A. (2003). Q-decomposition of reinforcement learning agents. In *ICML-03*.

Rustagi, J. S. (1976). *Variational Methods in Statistics*. Academic Press.

Sabin, D. und Freuder, E. C. (1994). Contradicting conventional wisdom in constraint satisfaction. In *ECAI-94*, S. 125–129.

Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. *AIJ*, *5*(2), 115–135.

Sacerdoti, E. D. (1975). The nonlinear nature of plans. In *IJCAI-75*, S. 206–214.

Sacerdoti, E. D. (1977). *A Structure for Plans and Behavior*. Elsevier/North-Holland.

Sadri, F. und Kowalski, R. (1995). Variants of the event calculus. In *ICLP-95*, S. 67–81.

Sahami, M., Dumais, S. T., Heckerman, D. und Horvitz, E. J. (1998). A Bayesian approach to filtering junk E-mail. In *Learning for Text Categorization: Papers from the 1998 Workshop*.

Sahami, M., Hearst, M. A. und Saund, E. (1996). Applying the multiple cause mixture model to text categorization. In *ICML-96*, S. 435–443.

Sahin, N. T., Pinker, S., Cash, S. S., Schomer, D. und Halgren, E. (2009). Sequential processing of lexical, grammatical, and phonological information within Broca's area. *Science*, *326*(5291), 445–449.

Sakuta, M. und Iida, H. (2002). AND/OR-tree search for solving problems with uncertainty: A case study using screen-shogi problems. *IPSJ Journal*, *43*(01).

Salomaa, A. (1969). Probabilistic and weighted grammars. *Information and Control*, *15*, 529–544.

Salton, G., Wong, A. und Yang, C. S. (1975). A vector space model for automatic indexing. *CACM*, *18*(11), 613–620.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, *3*(3), 210–229.

Samuel, A. L. (1967). Some studies in machine learning using the game of checkers II – Recent progress. *IBM Journal of Research and Development*, *11*(6), 601–617.

Samuelsson, C. und Rayner, M. (1991). Quantitative evaluation of explanation-based learning as an optimization tool for a large-scale natural language system. In *IJCAI-91*, S. 609–615.

Sarawagi, S. (2007). Information extraction. *Foundations and Trends in Databases*, *1*(3), 261–377.

Satia, J. K. und Lave, R. E. (1973). Markovian decision processes with probabilistic observation of states. *Management Science*, *20*(1), 1–13.

Sato, T. und Kameya, Y. (1997). PRISM: A symbolic-statistical modeling language. In *IJCAI-97*, S. 1330–1335.

Saul, L. K., Jaakkola, T. und Jordan, M. I. (1996). Mean field theory for sigmoid belief networks. *JAIR*, *4*, 61–76.

Savage, L. J. (1954). *The Foundations of Statistics*. Wiley.

Sayre, K. (1993). Three more flaws in the computational model. Paper presented at the APA (Central Division) Annual Conference, Chicago, Illinois.

Schaeffer, J. (2008). *One Jump Ahead: Computer Perfection at Checkers*. Springer-Verlag.

Schaeffer, J., Burch, N., Bjornsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P. und Sutphen, S. (2007). Checkers is solved. *Science*, *317*, 1518–1522.

Schank, R. C. und Abelson, R. P. (1977). *Scripts, Plans, Goals, and Understanding*. Lawrence Erlbaum Associates.

Schank, R. C. und Riesbeck, C. (1981). *Inside Computer Understanding: Five Programs Plus Miniatures*. Lawrence Erlbaum Associates.

Schapire, R. E. und Singer, Y. (2000). Boostexter: A boosting-based system for text categorization. *Machine Learning*, *39*(2/3), 135–168.

Schapire, R. E. (1990). The strength of weak learnability. *Machine Learning*, *5*(2), 197–227.

Schapire, R. E. (2003). The boosting approach to machine learning: An overview. In Denison, D. D., Hansen, M. H., Holmes, C., Mallick, B. und Yu, B. (Hrsg.), *Nonlinear Estimation and Classification*. Springer.

Schmid, C. und Mohr, R. (1996). Combining grey-value invariants with local constraints for object recognition. In *CVPR*.

Schmolze, J. G. und Lipkis, T. A. (1983). Classification in the KL-ONE representation system. In *IJCAI-83*, S. 330–332.

Schölkopf, B. und Smola, A. J. (2002). *Learning with Kernels*. MIT Press.

Schöning, T. (1999). A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *FOCS-99*, S. 410–414.

Schoppers, M. J. (1987). Universal plans for reactive robots in unpredictable environments. In *IJCAI-87*, S. 1039–1046.

Schoppers, M. J. (1989). In defense of reaction plans as caches. *AIMag*, *10*(4), 51–60.

Schröder, E. (1877). *Der Operationskreis des Logikkalküls*. B. G. Teubner, Leipzig.

Schultz, W., Dayan, P. und Montague, P. R. (1997). A neural substrate of prediction and reward. *Science*, *275*, 1593.

Schulz, D., Burgard, W., Fox, D. und Cremers, A. B. (2003). People tracking with mobile robots using sample-based joint probabilistic data association filters. *Int. J. Robotics Research*, *22*(2), 99–116.

Schulz, S. (2004). System Description: E 0.81. In *Proc. International Joint Conference on Automated Reasoning*, Band 3097 of *LNAI*, S. 223–228.

Schütze, H. (1995). *Ambiguity in Language Learning: Computational and Cognitive Models*. Ph.D. thesis, Stanford University. Auch veröffentlicht von CSLI Press, 1997.

Schwartz, J. T., Scharir, M. und Hopcroft, J. (1987). *Planning, Geometry and Complexity of Robot Motion*. Ablex Publishing Corporation.

Schwartz, S. P. (Hrsg.). (1977). *Naming, Necessity, and Natural Kinds*. Cornell University Press.

Scott, D. und Krauss, P. (1966). Assigning probabilities to logical formulas. In Hintikka, J. und Suppes, P. (Hrsg.), *Aspects of Inductive Logic*. North-Holland.

Searle, J. R. (1980). Minds, brains, and programs. *BBS*, *3*, 417–457.

Searle, J. R. (1984). *Minds, Brains and Science*. Harvard University Press.

Searle, J. R. (1990). Is the brain's mind a computer program? *Scientific American*, *262*, 26–31.

Searle, J. R. (1992). *The Rediscovery of the Mind*. MIT Press.

Sebastiani, F. (2002). Machine learning in automated text categorization. *ACM Computing Surveys*, *34*(1), 1–47.

Segaran, T. (2007). *Programming Collective Intelligence: Building Smart Web 2.0 Applications*. O'Reilly.

Selman, B., Kautz, H. und Cohen, B. (1996). Local search strategies for satisfiability testing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 26*, S. 521–532. American Mathematical Society.

Selman, B. und Levesque, H. J. (1993). The complexity of path-based defeasible inheritance. *AIJ*, *62*(2), 303–339.

Selman, B., Levesque, H. J. und Mitchell, D. (1992). A new method for solving hard satisfiability problems. In *AAAI-92*, S. 440–446.

Sha, F. und Pereira, F. (2003). Shallow parsing with conditional random fields. Technical report CIS TR MS-CIS-02-35, Univ. of Penn.

Shachter, R. D. (1986). Evaluating influence diagrams. *Operations Research*, *34*, 871–882.

Shachter, R. D. (1998). Bayes-ball: The rational pastime (for determining irrelevance and requisite information in belief networks and influence diagrams). In *UAI-98*, S. 480–487.

Shachter, R. D., D'Ambrosio, B. und Del Favero, B. A. (1990). Symbolic probabilistic inference in belief networks. In *AAAI-90*, S. 126–131.

Shachter, R. D. und Kenley, C. R. (1989). Gaussian influence diagrams. *Management Science*, *35*(5), 527–550.

Shachter, R. D. und Peot, M. (1989). Simulation approaches to general probabilistic inference on belief networks. In *UAI-98*.

Shachter, R. D. und Heckerman, D. (1987). Thinking backward for knowledge acquisition. *AIMag*, *3*(Fall).

Shafer, G. (1976). *A Mathematical Theory of Evidence*. Princeton University Press.

Shahookar, K. und Mazumder, P. (1991). VLSI cell placement techniques. *Computing Surveys*, *23*(2), 143–220.

Shanahan, M. (1997). *Solving the Frame Problem*. MIT Press.

Shanahan, M. (1999). The event calculus explained. In Wooldridge, M. J. und Veloso, M. (Hrsg.), *Artificial Intelligence Today*, S. 409–430. Springer-Verlag.

Shankar, N. (1986). *Proof-Checking Metamathematics*. Ph.D. thesis, Computer Science Department, University of Texas at Austin.

Shannon, C. E. und Weaver, W. (1949). *The Mathematical Theory of Communication*. University of Illinois Press.

Shannon, C. E. (1948). A mathematical theory of communication. *Bell Systems Technical Journal*, *27*, 379–423, 623–656.

Shannon, C. E. (1950). Programming a computer for playing chess. *Philosophical Magazine*, *41*(4), 256–275.

Shaparau, D., Pistore, M. und Traverso, P. (2008). Fusing procedural and declarative planning goals for nondeterministic domains. In *AAAI-08*.

Shapiro, E. (1981). An algorithm that infers theories from facts. In *IJCAI-81*, p. 1064.

Shapiro, S. C. (Hrsg.). (1992). *Encyclopedia of Artificial Intelligence* (2. Auflage). Wiley.

Shapley, S. (1953). Stochastic games. In *PNAS*, Band 39, S. 1095–1100.

Shatkay, H. und Kaelbling, L. P. (1997). Learning topological maps with weak local odometric information. In *IJCAI-97*.

Shelley, M. (1818). *Frankenstein: Or, the Modern Prometheus*. Pickering and Chatto.

Sheppard, B. (2002). World-championship-caliber scrabble. *AIJ*, *134*(1–2), 241–275.

Shi, J. und Malik, J. (2000). Normalized cuts and image segmentation. *PAMI*, *22*(8), 888–905.

Shieber, S. (1994). Lessons from a restricted Turing Test. *CACM*, *37*, 70–78.

Shieber, S. (Hrsg.). (2004). *The Turing Test*. MIT Press.

Shoham, Y. (1993). Agent-oriented programming. *AIJ*, *60*(1), 51–92.

Shoham, Y. (1994). *Artificial Intelligence Techniques in Prolog*. Morgan Kaufmann.

Shoham, Y. und Leyton-Brown, K. (2009). *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge Univ. Press.

Shoham, Y., Powers, R. und Grenager, T. (2004). If multi-agent learning is the answer, what is the question? In *Proc. AAAI Fall Symposium on Artificial Multi-Agent Learning*.

Shortliffe, E. H. (1976). *Computer-Based Medical Consultations: MYCIN*. Elsevier/North-Holland.

Sietsma, J. und Dow, R. J. F. (1988). Neural net pruning – Why and how. In *IEEE International Conference on Neural Networks*, S. 325–333.

Siklossy, L. und Dreussi, J. (1973). An efficient robot planner which generates its own procedures. In *IJCAI-73*, S. 423–430.

Silverstein, C., Henzinger, M., Marais, H. und Moricz, M. (1998). Analysis of a very large altavista query log. Tech. rep. 1998-014, Digital Systems Research Center.

Simmons, R. und Koenig, S. (1995). Probabilistic robot navigation in partially observable environments. In *IJCAI-95*, S. 1080–1087. IJCAI, Inc.

Simon, D. (2006). *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. Wiley.

Simon, H. A. (1947). *Administrative behavior*. Macmillan.

Simon, H. A. (1957). *Models of Man: Social and Rational*. John Wiley.

Simon, H. A. (1963). Experiments with a heuristic compiler. *JACM*, *10*, 493–506.

Simon, H. A. (1981). *The Sciences of the Artificial* (2. Auflage). MIT Press.

Simon, H. A. (1982). *Models of Bounded Rationality, Volume 1*. The MIT Press.

Simon, H. A. und Newell, A. (1958). Heuristic problem solving: The next advance in operations research. *Operations Research*, *6*, 1–10.

Simon, H. A. und Newell, A. (1961). Computer simulation of human thinking and problem solving. *Datamation, June/July*, 35–37.

Simon, J. C. und Dubois, O. (1989). Number of solutions to satisfiability instances – Applications to knowledge bases. *AIJ*, *3*, 53–65.

Simonis, H. (2005). Sudoku as a constraint problem. In *CP Workshop on Modeling and Reformulating Constraint Satisfaction Problems*, S. 13–27.

Singer, P. W. (2009). *Wired for War*. Penguin Press.

Singh, P., Lin, T., Mueller, E. T., Lim, G., Perkins, T. und Zhu, W. L. (2002). Open mind common sense: Knowledge acquisition from the general public. In *Proc. First International Conference on Ontologies, Databases, and Applications of Semantics for Large Scale Information Systems*.

Singhal, A., Buckley, C. und Mitra, M. (1996). Pivoted document length normalization. In *SIGIR-96*, S. 21–29.

Sittler, R. W. (1964). An optimal data association problem in surveillance theory. *IEEE Transactions on Military Electronics*, *8*(2), 125–139.

Skinner, B. F. (1953). *Science and Human Behavior*. Macmillan.

Skolem, T. (1920). Logisch-kombinatorische Untersuchungen über die Erfüllbarkeit oder Beweisbarkeit mathematischer Sätze nebst einem Theorem über die dichten Mengen. *Videnskapsselskapets skrifter, I. Matematisk-naturvidenskabelig klasse*, *4*.

Skolem, T. (1928). Über die mathematische Logik. *Norsk matematisk tidsskrift*, *10*, 125–142.

Slagle, J. R. (1963). A heuristic program that solves symbolic integration problems in freshman calculus. *JACM*, *10*(4).

Slate, D. J. und Atkin, L. R. (1977). CHESS 4.5 – Northwestern University chess program. In Frey, P. W. (Hrsg.), *Chess Skill in Man and Machine*, S. 82–118. Springer-Verlag.

Slater, E. (1950). Statistics for the chess computer and the factor of mobility. In *Symposium on Information Theory*, S. 150–152. Ministry of Supply.

Sleator, D. und Temperley, D. (1993). Parsing English with a link grammar. In *Third Annual Workshop on Parsing technologies*.

Slocum, J. und Sonneveld, D. (2006). *The 15 Puzzle*. Slocum Puzzle Foundation.

Sloman, A. (1978). *The Computer Revolution in Philosophy*. Harvester Press.

Smallwood, R. D. und Sondik, E. J. (1973). The optimal control of partially observable Markov processes over a finite horizon. *Operations Research*, *21*, 1071–1088.

Smart, J. J. C. (1959). Sensations and brain processes. *Philosophical Review*, *68*, 141–156.

Smith, B. (2004). Ontology. In Floridi, L. (Hrsg.), *The Blackwell Guide to the Philosophy of Computing and Information*, S. 155–166. Wiley-Blackwell.

Smith, D. E., Genesereth, M. R. und Ginsberg, M. L. (1986). Controlling recursive inference. *AIJ*, *30*(3), 343–389.

Smith, D. A. und Eisner, J. (2008). Dependency parsing by belief propagation. In *EMNLP*, S. 145–156.

Smith, D. E. und Weld, D. S. (1998). Conformant Graphplan. In *AAAI-98*, S. 889–896.

Smith, J. Q. (1988). *Decision Analysis*. Chapman and Hall.

Smith, J. E. und Winkler, R. L. (2006). The optimizer's curse: Skepticism and postdecision surprise in decision analysis. *Management Science*, *52*(3), 311–322.

Smith, J. M. (1982). *Evolution and the Theory of Games*. Cambridge University Press.

Smith, J. M. und Szathmáry, E. (1999). *The Origins of Life: From the Birth of Life to the Origin of Language*. Oxford University Press.

Smith, M. K., Welty, C. und McGuinness, D. (2004). OWL web ontology language guide. Tech. rep., W3C.

Smith, R. C. und Cheeseman, P. (1986). On the representation and estimation of spatial uncertainty. *Int. J. Robotics Research*, *5*(4), 56–68.

Smith, S. J. J., Nau, D. S. und Throop, T. A. (1998). Success in spades: Using AI planning techniques to win the world championship of computer bridge. In *AAAI-98*, S. 1079–1086.

Smolensky, P. (1988). On the proper treatment of connectionism. *BBS*, *2*, 1–74.

Smullyan, R. M. (1995). *First-Order Logic*. Dover.

Smyth, P., Heckerman, D. und Jordan, M. I. (1997). Probabilistic independence networks for hidden Markov probability models. *Neural Computation*, *9*(2), 227–269.

Snell, M. B. (2008). Do you have free will? John Searle reflects on various philosophical questions in light of new research on the brain. *California Alumni Magazine*, *March/April*.

Soderland, S. und Weld, D. S. (1991). Evaluating nonlinear planning. Technical report TR-91-02-03, University of Washington Department of Computer Science and Engineering.

Solomonoff, R. J. (1964). A formal theory of inductive inference. *Information and Control*, *7*, 1–22, 224–254.

Solomonoff, R. J. (2009). Algorithmic probability– theory and applications. In Emmert-Streib, F. und Dehmer, M. (Hrsg.), *Information Theory and Statistical Learning*. Springer.

Sondik, E. J. (1971). *The Optimal Control of Partially Observable Markov Decision Processes*. Ph.D. thesis, Stanford University.

Sosic, R. und Gu, J. (1994). Efficient local search with conflict minimization: A case study of the n-queens problem. *IEEE Transactions on Knowledge and Data Engineering*, *6*(5), 661–668.

Sowa, J. (1999). *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Blackwell.

Spaan, M. T. J. und Vlassis, N. (2005). Perseus: Randomized point-based value iteration for POMDPs. *JAIR*, *24*, 195–220.

Spiegelhalter, D. J., Dawid, A. P., Lauritzen, S. und Cowell, R. (1993). Bayesian analysis in expert systems. *Statistical Science*, *8*, 219–282.

Spielberg, S. (2001). AI. Movie.

Spirtes, P., Glymour, C. und Scheines, R. (1993). *Causation, prediction, and search*. Springer-Verlag.

Srinivasan, A., Muggleton, S. H., King, R. D. und Sternberg, M. J. E. (1994). Mutagenesis: ILP experiments in a non-determinate biological domain. In *ILP-94*, Band 237, S. 217–232.

Srivas, M. und Bickford, M. (1990). Formal verification of a pipelined microprocessor. *IEEE Software*, *7*(5), 52–64.

Staab, S. (2004). *Handbook on Ontologies*. Springer.

Stallman, R. M. und Sussman, G. J. (1977). Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *AIJ*, *9*(2), 135–196.

Stanfill, C. und Waltz, D. (1986). Toward memory-based reasoning. *CACM*, *29*(12), 1213–1228.

Stefik, M. (1995). *Introduction to Knowledge Systems*. Morgan Kaufmann.

Stein, L. A. (2002). *Interactive Programming in Java* (pre-publication draft). Morgan Kaufmann.

Stephenson, T., Bourlard, H., Bengio, S. und Morris, A. (2000). Automatic speech recognition using dynamic bayesian networks with both acoustic and articulatory features. In *ICSLP-00*, S. 951–954.

Stergiou, K. und Walsh, T. (1999). The difference all-difference makes. In *IJCAI-99*, S. 414–419.

Stickel, M. E. (1992). A prolog technology theorem prover: a new exposition and implementation in prolog. *Theoretical Computer Science*, *104*, 109–128.

Stiller, L. (1992). KQNKRR. *J. International Computer Chess Association*, *15*(1), 16–18.

Stiller, L. (1996). Multilinear algebra and chess endgames. In Nowakowski, R. J. (Hrsg.), *Games of No Chance, MSRI, 29, 1996*. Mathematical Sciences Research Institute.

Stockman, G. (1979). A minimax algorithm better than alpha–beta? *AIJ*, *12*(2), 179–196.

Stoffel, K., Taylor, M. und Hendler, J. (1997). Efficient management of very large ontologies. In *Proc. AAAI-97*, S. 442–447.

Stolcke, A. und Omohundro, S. (1994). Inducing probabilistic grammars by Bayesian model merging. In *Proc. Second International Colloquium on Grammatical Inference and Applications (ICGI-94)*, S. 106–118.

Stone, M. (1974). Cross-validatory choice and assessment of statostical predictions. *J. Royal Statistical Society*, *36*(111–133).

Stone, P. (2000). *Layered Learning in Multi-Agent Systems: A Winning Approach to Robotic Soccer*. MIT Press.

Stone, P. (2003). Multiagent competitions and research: Lessons from RoboCup and TAC. In Lima, P. U. und Rojas, P. (Hrsg.), *RoboCup-2002: Robot Soccer World Cup VI*, S. 224–237. Springer Verlag.

Stone, P., Kaminka, G. und Rosenschein, J. S. (2009). Leading a best-response teammate in an ad hoc team. In *AAMAS Workshop in Agent Mediated Electronic Commerce*.

Stork, D. G. (2004). Optics and realism in rennaissance art. *Scientific American*, S. 77–83.

Strachey, C. (1952). Logical or non-mathematical programmes. In *Proc. 1952 ACM national meeting (Toronto)*, S. 46–49.

Stratonovich, R. L. (1959). Optimum nonlinear systems which bring about a separation of a signal with constant parameters from noise. *Radiofizika*, *2*(6), 892–901.

Stratonovich, R. L. (1965). On value of information. *Izvestiya of USSR Academy of Sciences, Technical Cybernetics*, *5*, 3–12.

Subramanian, D. und Feldman, R. (1990). The utility of EBL in recursive domain theories. In *AAAI-90*, Band 2, S. 942–949.

Subramanian, D. und Wang, E. (1994). Constraint-based kinematic synthesis. In *Proc. International Conference on Qualitative Reasoning*, S. 228–239.

Sussman, G. J. (1975). *A Computer Model of Skill Acquisition*. Elsevier/North-Holland.

Sutcliffe, G. und Suttner, C. (1998). The TPTP Problem Library: CNF Release v1.2.1. *JAR*, *21*(2), 177–203.

Sutcliffe, G., Schulz, S., Claessen, K. und Gelder, A. V. (2006). Using the TPTP language for writing derivations and finite interpretations. In *Proc. International Joint Conference on Automated Reasoning*, S. 67–81.

Sutherland, I. (1963). Sketchpad: A man-machine graphical communication system. In *Proc. Spring Joint Computer Conference*, S. 329–346.

Sutton, C. und McCallum, A. (2007). An introduction to conditional random fields for relational learning. In Getoor, L. und Taskar, B. (Hrsg.), *Introduction to Statistical Relational Learning*. MIT Press.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, *3*, 9–44.

Sutton, R. S., McAllester, D. A., Singh, S. P. und Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. In Solla, S. A., Leen, T. K. und Müller, K.-R. (Hrsg.), *NIPS 12*, S. 1057–1063. MIT Press.

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *ICML-90*, S. 216–224.

Sutton, R. S. und Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Svore, K. und Burges, C. (2009). A machine learning approach for improved bm25 retrieval. In *Proc. Conference on Information Knowledge Management*.

Swade, D. (2000). *Difference Engine: Charles Babbage And The Quest To Build The First Computer*. Diane Publishing Co.

Swerling, P. (1959). First order error propagation in a stagewise smoothing procedure for satellite observations. *J. Astronautical Sciences*, *6*, 46–52.

Swift, T. und Warren, D. S. (1994). Analysis of SLGWAM evaluation of definite programs. In *Logic Programming. Proc. 1994 International Symposium on Logic programming*, S. 219–235.

Syrjänen, T. (2000). Lparse 1.0 user's manual. *saturn.tcs.hut.fi/Software/smodels*.

Tadepalli, P. (1993). Learning from queries and examples with tree-structured bias. In *ICML-93*, S. 322–329.

Tadepalli, P., Givan, R. und Driessens, K. (2004). Relational reinforcement learning: An overview. In *ICML-04*.

Tait, P. G. (1880). Note on the theory of the "15 puzzle". *Proc. Royal Society of Edinburgh*, *10*, 664–665.

Tamaki, H. und Sato, T. (1986). OLD resolution with tabulation. In *ICLP-86*, S. 84–98.

Tarjan, R. E. (1983). *Data Structures and Network Algorithms*. CBMS-NSF Regional Conference Series in Applied Mathematics. SIAM (Society for Industrial and Applied Mathematics).

Tarski, A. (1935). Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, *1*, 261–405.

Tarski, A. (1941). *Introduction to Logic and to the Methodology of Deductive Sciences*. Dover.

Tarski, A. (1956). *Logic, Semantics, Metamathematics: Papers from 1923 to 1938*. Oxford University Press.

Tash, J. K. und Russell, S. J. (1994). Control strategies for a stochastic planner. In *AAAI-94*, S. 1079–1085.

Taskar, B., Abbeel, P. und Koller, D. (2002). Discriminative probabilistic models for relational data. In *UAI-02*.

Tate, A. (1975a). Interacting goals and their use. In *IJCAI-75*, S. 215–218.

Tate, A. (1975b). *Using Goal Structure to Direct Search in a Problem Solver*. Ph.D. thesis, University of Edinburgh.

Tate, A. (1977). Generating project networks. In *IJCAI-77*, S. 888–893.

Tate, A. und Whiter, A. M. (1984). Planning with multiple resource constraints and an application to a naval planning problem. In *Proc. First Conference on AI Applications*, S. 410–416.

Tatman, J. A. und Shachter, R. D. (1990). Dynamic programming and influence diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, *20*(2), 365–379.

Tattersall, C. (1911). *A Thousand End-Games: A Collection of Chess Positions That Can be Won or Drawn by the Best Play*. British Chess Magazine.

Taylor, G., Stensrud, B., Eitelman, S. und Dunham, C. (2007). Towards automating airspace management. In *Proc. Computational Intelligence for Security and Defense Applications (CISDA) Conference*, S. 1–5.

Tenenbaum, J., Griffiths, T. und Niyogi, S. (2007). Intuitive theories as grammars for causal inference. In Gopnik, A. und Schulz, L. (Hrsg.), *Causal learning: Psychology, Philosophy, and Computation*. Oxford University Press.

Tesauro, G. (1992). Practical issues in temporal difference learning. *Machine Learning*, *8*(3–4), 257–277.

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *CACM*, *38*(3), 58–68.

Tesauro, G. und Sejnowski, T. (1989). A parallel network that learns to play backgammon. *AIJ*, *39*(3), 357–390.

Teyssier, M. und Koller, D. (2005). Ordering-based search: A simple and effective algorithm for learning Bayesian networks. In *UAI-05*, S. 584–590.

Thaler, R. (1992). *The Winner's Curse: Paradoxes and Anomalies of Economic Life*. Princeton University Press.

Thaler, R. und Sunstein, C. (2009). *Nudge: Improving Decisions About Health, Wealth, and Happiness*. Penguin.

Theocharous, G., Murphy, K. und Kaelbling, L. P. (2004). Representing hierarchical POMDPs as DBNs for multi-scale robot localization. In *ICRA-04*.

Thiele, T. (1880). Om anvendelse af mindste kvadraters methode i nogle tilfælde, hvor en komplikation af visse slags uensartede tilfældige fejlkilder giver fejlene en 'systematisk' karakter. *Vidensk. Selsk. Skr. 5. Rk., naturvid. og mat. Afd.*, *12*, 381–408.

Thielscher, M. (1999). From situation calculus to fluent calculus: State update axioms as a solution to the inferential frame problem. *AIJ*, *111*(1–2), 277–299.

Thompson, K. (1986). Retrograde analysis of certain endgames. *J. International Computer Chess Association*, *May*, 131–139.

Thompson, K. (1996). 6-piece endgames. *J. International Computer Chess Association*, *19*(4), 215–226.

Thrun, S., Burgard, W. und Fox, D. (2005). *Probabilistic Robotics*. MIT Press.

Thrun, S., Fox, D. und Burgard, W. (1998). A probabilistic approach to concurrent mapping and localization for mobile robots. *Machine Learning*, *31*, 29–53.

Thrun, S. (2006). Stanley, the robot that won the DARPA Grand Challenge. *J. Field Robotics*, *23*(9), 661–692.

Tikhonov, A. N. (1963). Solution of incorrectly formulated problems and the regularization method. *Soviet Math. Dokl.*, *5*, 1035–1038.

Titterington, D. M., Smith, A. F. M. und Makov, U. E. (1985). *Statistical analysis of finite mixture distributions*. Wiley.

Toffler, A. (1970). *Future Shock*. Bantam.

Tomasi, C. und Kanade, T. (1992). Shape and motion from image streams under orthography: A factorization method. *IJCV*, *9*, 137–154.

Torralba, A., Fergus, R. und Weiss, Y. (2008). Small codes and large image databases for recognition. In *CVPR*, S. 1–8.

Trucco, E. und Verri, A. (1998). *Introductory Techniques for 3-D Computer Vision*. Prentice Hall.

Tsitsiklis, J. N. und Van Roy, B. (1997). An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, *42*(5), 674–690.

Tumer, K. und Wolpert, D. (2000). Collective intelligence and braess' paradox. In *AAAI-00*, S. 104–109.

Turcotte, M., Muggleton, S. H. und Sternberg, M. J. E. (2001). Automated discovery of structural signatures of protein fold and function. *J. Molecular Biology*, *306*, 591–605.

Turing, A. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Mathematical Society, 2nd series*, *42*, 230–265.

Turing, A. (1948). Intelligent machinery. Tech. rep., National Physical Laboratory. reprinted in (Ince, 1992).

Turing, A. (1950). Computing machinery and intelligence. *Mind*, *59*, 433–460.

Turing, A., Strachey, C., Bates, M. A. und Bowden, B. V. (1953). Digital computers applied to games. In Bowden, B. V. (Hrsg.), *Faster than Thought*, S. 286–310. Pitman.

Tversky, A. und Kahneman, D. (1982). Causal schemata in judgements under uncertainty. In Kahneman, D., Slovic, P. und Tversky, A. (Hrsg.), *Judgement Under Uncertainty: Heuristics and Biases*. Cambridge University Press.

Ullman, J. D. (1985). Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, *10*(3), 289–321.

Ullman, S. (1979). *The Interpretation of Visual Motion*. MIT Press.

Urmson, C. und Whittaker, W. (2008). Self-driving cars and the Urban Challenge. *IEEE Intelligent Systems*, *23*(2), 66–68.

Valiant, L. (1984). A theory of the learnable. *CACM*, *27*, 1134–1142.

van Beek, P. (2006). Backtracking search algorithms. In Rossi, F., van Beek, P. und Walsh, T. (Hrsg.), *Handbook of Constraint Programming*. Elsevier.

van Beek, P. und Chen, X. (1999). CPlan: A constraint programming approach to planning. In *AAAI-99*, S. 585–590.

van Beek, P. und Manchak, D. (1996). The design and experimental analysis of algorithms for temporal reasoning. *JAIR*, *4*, 1–18.

van Bentham, J. und ter Meulen, A. (1997). *Handbook of Logic and Language*. MIT Press.

Van Emden, M. H. und Kowalski, R. (1976). The semantics of predicate logic as a programming language. *JACM*, *23*(4), 733–742.

van Harmelen, F. und Bundy, A. (1988). Explanation-based generalisation = partial evaluation. *AIJ*, *36*(3), 401–412.

van Harmelen, F., Lifschitz, V. und Porter, B. (2007). *The Handbook of Knowledge Representation*. Elsevier.

van Heijenoort, J. (Hrsg.). (1967). *From Frege to Gödel: A Source Book in Mathematical Logic, 1879?1931*. Harvard University Press.

Van Hentenryck, P., Saraswat, V. und Deville, Y. (1998). Design, implementation, and evaluation of the constraint language cc(FD). *J. Logic Programming*, *37*(1–3), 139–164.

van Hoeve, W.-J. (2001). The alldifferent constraint: a survey. In *6th Annual Workshop of the ERCIM Working Group on Constraints*.

van Hoeve, W.-J. und Katriel, I. (2006). Global constraints. In Rossi, F., van Beek, P. und Walsh, T. (Hrsg.), *Handbook of Constraint Processing*, S. 169–208. Elsevier.

van Lambalgen, M. und Hamm, F. (2005). *The Proper Treatment of Events*. Wiley-Blackwell.

van Nunen, J. A. E. E. (1976). A set of successive approximation methods for discounted Markovian decision problems. *Zeitschrift fur Operations Research, Serie A*, *20*(5), 203–208.

Van Roy, B. (1998). *Learning and value function approximation in complex decision processes*. Ph.D. thesis, Laboratory for Information and Decision Systems, MIT.

Van Roy, P. L. (1990). Can logic programming execute as fast as imperative programming? Report UCB/CSD 90/600, Computer Science Division, University of California, Berkeley, California.

Vapnik, V. N. (1998). *Statistical Learning Theory*. Wiley.

Vapnik, V. N. und Chervonenkis, A. Y. (1971). On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability and Its Applications*, *16*, 264–280.

Varian, H. R. (1995). Economic mechanism design for computerized agents. In *USENIX Workshop on Electronic Commerce*, S. 13–21.

Vauquois, B. (1968). A survey of formal grammars and algorithms for recognition and transformation in mechanical translation. In *Proc. IFIP Congress*, S. 1114–1122.

Veloso, M. und Carbonell, J. G. (1993). Derivational analogy in PRODIGY: Automating case acquisition, storage, and utilization. *Machine Learning*, *10*, 249–278.

Vere, S. A. (1983). Planning in time: Windows and durations for activities and goals. *PAMI*, *5*, 246–267.

Verma, V., Gordon, G., Simmons, R. und Thrun, S. (2004). Particle filters for rover fault diagnosis. *IEEE Robotics and Automation Magazine*, *June*.

Vinge, V. (1993). The coming technological singularity: How to survive in the post-human era. In *VISION-21 Symposium*. NASA Lewis Research Center and the Ohio Aerospace Institute.

Viola, P. und Jones, M. (2002a). Fast and robust classification using asymmetric adaboost and a detector cascade. In *NIPS 14*.

Viola, P. und Jones, M. (2002b). Robust real-time object detection. *ICCV*.

Visser, U. und Burkhard, H.-D. (2007). RoboCup 2006: achievements and goals for the future. *AIMag*, *28*(2), 115–130.

Visser, U., Ribeiro, F., Ohashi, T. und Dellaert, F. (Hrsg.). (2008). *RoboCup 2007: Robot Soccer World Cup XI*. Springer.

Viterbi, A. J. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, *13*(2), 260–269.

Vlassis, N. (2008). *A Concise Introduction to Multiagent Systems and Distributed Artificial Intelligence*. Morgan and Claypool.

von Mises, R. (1928). *Wahrscheinlichkeit, Statistik und Wahrheit*. J. Springer.

von Neumann, J. (1928). Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, *100*(295–320).

von Neumann, J. und Morgenstern, O. (1944). *Theory of Games and Economic Behavior* (1. Auflage). Princeton University Press.

von Winterfeldt, D. und Edwards, W. (1986). *Decision Analysis and Behavioral Research*. Cambridge University Press.

Vossen, T., Ball, M., Lotem, A. und Nau, D. S. (2001). Applying integer programming to AI planning. *Knowledge Engineering Review*, *16*, 85–100.

Wainwright, M. J. und Jordan, M. I. (2008). Graphical models, exponential families, and variational inference. *Machine Learning*, *1*(1–2), 1–305.

Waldinger, R. (1975). Achieving several goals simultaneously. In Elcock, E. W. und Michie, D. (Hrsg.), *Machine Intelligence 8*, S. 94–138. Ellis Horwood.

Wallace, A. R. (1858). On the tendency of varieties to depart indefinitely from the original type. *Proc. Linnean Society of London*, *3*, 53–62.

Waltz, D. (1975). Understanding line drawings of scenes with shadows. In Winston, P. H. (Hrsg.), *The Psychology of Computer Vision*. McGraw-Hill.

Wang, Y. und Gelly, S. (2007). Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *IEEE Symposium on Computational Intelligence and Games*, S. 175–182.

Wanner, E. (1974). *On remembering, forgetting and understanding sentences*. Mouton.

Warren, D. H. D. (1974). WARPLAN: A System for Generating Plans. Department of Computational Logic Memo 76, University of Edinburgh.

Warren, D. H. D. (1983). An abstract Prolog instruction set. Technical note 309, SRI International.

Warren, D. H. D., Pereira, L. M. und Pereira, F. (1977). PROLOG: The language and its implementation compared with LISP. *SIGPLAN Notices*, *12*(8), 109–115.

Wasserman, L. (2004). *All of Statistics*. Springer.

Watkins, C. J. (1989). *Models of Delayed Reinforcement Learning*. Ph.D. thesis, Psychology Department, Cambridge University.

Watson, J. D. und Crick, F. H. C. (1953). A structure for deoxyribose nucleic acid. *Nature*, *171*, 737.

Waugh, K., Schnizlein, D., Bowling, M. und Szafron, D. (2009). Abstraction pathologies in extensive games. In *AAMAS-09*.

Weaver, W. (1949). Translation. In Locke, W. N. und Booth, D. (Hrsg.), *Machine translation of languages: fourteen essays*, S. 15–23. Wiley.

Webber, B. L. und Nilsson, N. J. (Hrsg.). (1981). *Readings in Artificial Intelligence*. Morgan Kaufmann.

Weibull, J. (1995). *Evolutionary Game Theory*. MIT Press.

Weidenbach, C. (2001). SPASS: Combining superposition, sorts and splitting. In Robinson, A. und Voronkov, A. (Hrsg.), *Handbook of Automated Reasoning*. MIT Press.

Weiss, G. (2000a). *Multiagent systems*. MIT Press.

Weiss, Y. (2000b). Correctness of local probability propagation in graphical models with loops. *Neural Computation*, *12*(1), 1–41.

Weiss, Y. und Freeman, W. (2001). Correctness of belief propagation in Gaussian graphical models of arbitrary topology. *Neural Computation*, *13*(10), 2173–2200.

Weizenbaum, J. (1976). *Computer Power and Human Reason*. W. H. Freeman.

Weld, D. S. (1994). An introduction to least commitment planning. *AIMag*, *15*(4), 27–61.

Weld, D. S. (1999). Recent advances in AI planning. *AIMag*, *20*(2), 93–122.

Weld, D. S., Anderson, C. R. und Smith, D. E. (1998). Extending graphplan to handle uncertainty and sensing actions. In *AAAI-98*, S. 897–904.

Weld, D. S. und de Kleer, J. (1990). *Readings in Qualitative Reasoning about Physical Systems*. Morgan Kaufmann.

Weld, D. S. und Etzioni, O. (1994). The first law of robotics: A call to arms. In *AAAI-94*.

Wellman, M. P. (1985). Reasoning about preference models. Technical report MIT/LCS/TR-340, Laboratory for Computer Science, MIT.

Wellman, M. P. (1988). *Formulation of Tradeoffs in Planning under Uncertainty*. Ph.D. thesis, Massachusetts Institute of Technology.

Wellman, M. P. (1990a). Fundamental concepts of qualitative probabilistic networks. *AIJ*, *44*(3), 257–303.

Wellman, M. P. (1990b). The STRIPS assumption for planning under uncertainty. In *AAAI-90*, S. 198–203.

Wellman, M. P. (1995). The economic approach to artificial intelligence. *ACM Computing Surveys*, *27*(3), 360–362.

Wellman, M. P., Breese, J. S. und Goldman, R. (1992). From knowledge bases to decision models. *Knowledge Engineering Review*, *7*(1), 35–53.

Wellman, M. P. und Doyle, J. (1992). Modular utility representation for decision-theoretic planning. In *ICAPS-92*, S. 236–242.

Wellman, M. P., Wurman, P., O'Malley, K., Bangera, R., Lin, S., Reeves, D. und Walsh, W. (2001). A trading agent competition. *IEEE Internet Computing*.

Wells, H. G. (1898). *The War of the Worlds*. William Heinemann.

Werbos, P. (1974). *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Ph.D. thesis, Harvard University.

Werbos, P. (1977). Advanced forecasting methods for global crisis warning and models of intelligence. *General Systems Yearbook*, *22*, 25–38.

Wesley, M. A. und Lozano-Perez, T. (1979). An algorithm for planning collision-free paths among polyhedral objects. *CACM*, *22*(10), 560–570.

Wexler, Y. und Meek, C. (2009). MAS: A multiplicative approximation scheme for probabilistic inference. In *NIPS 21*.

Whitehead, A. N. (1911). *An Introduction to Mathematics*. Williams and Northgate.

Whitehead, A. N. und Russell, B. (1910). *Principia Mathematica*. Cambridge University Press.

Whorf, B. (1956). *Language, Thought, and Reality*. MIT Press.

Widrow, B. (1962). Generalization and information storage in networks of adaline "neurons". In *Self-Organizing Systems 1962*, S. 435–461.

Widrow, B. und Hoff, M. E. (1960). Adaptive switching circuits. In *1960 IRE WESCON Convention Record*, S. 96–104.

Wiedijk, F. (2003). Comparing mathematical provers. In *Mathematical Knowledge Management*, S. 188–202.

Wiegley, J., Goldberg, K., Peshkin, M. und Brokowski, M. (1996). A complete algorithm for designing passive fences to orient parts. In *ICRA-96*.

Wiener, N. (1942). The extrapolation, interpolation, and smoothing of stationary time series. Osrd 370, Report to the Services 19, Research Project DIC-6037, MIT.

Wiener, N. (1948). *Cybernetics*. Wiley.

Wilensky, R. (1978). *Understanding goal-based stories*. Ph.D. thesis, Yale University.

Wilensky, R. (1983). *Planning and Understanding*. Addison-Wesley.

Wilkins, D. E. (1980). Using patterns and plans in chess. *AIJ*, *14*(2), 165–203.

Wilkins, D. E. (1988). *Practical Planning: Extending the AI Planning Paradigm*. Morgan Kaufmann.

Wilkins, D. E. (1990). Can AI planners solve practical problems? *Computational Intelligence*, *6*(4), 232–246.

Williams, B., Ingham, M., Chung, S. und Elliott, P. (2003). Model-based programming of intelligent embedded systems and robotic space explorers. In *Proc. IEEE: Special Issue on Modeling and Design of Embedded Software*, S. 212–237.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, *8*, 229–256.

Williams, R. J. und Baird, L. C. I. (1993). Tight performance bounds on greedy policies based on imperfect value functions. Tech. rep. NU-CCS-93-14, College of Computer Science, Northeastern University.

Wilson, R. A. und Keil, F. C. (Hrsg.). (1999). *The MIT Encyclopedia of the Cognitive Sciences*. MIT Press.

Wilson, R. (2004). *Four Colors Suffice*. Princeton University Press.

Winograd, S. und Cowan, J. D. (1963). *Reliable Computation in the Presence of Noise*. MIT Press.

Winograd, T. (1972). Understanding natural language. *Cognitive Psychology*, *3*(1), 1–191.

Winston, P. H. (1970). Learning structural descriptions from examples. Technical report MAC-TR-76, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Winston, P. H. (1992). *Artificial Intelligence* (3. Auflage). Addison-Wesley.

Wintermute, S., Xu, J. und Laird, J. (2007). SORTS: A human-level approach to real-time strategy AI. In *Proc. Third Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-07)*.

Witten, I. H. und Bell, T. C. (1991). The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, *37*(4), 1085–1094.

Witten, I. H. und Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques* (2. Auflage). Morgan Kaufmann.

Witten, I. H., Moffat, A. und Bell, T. C. (1999). *Managing Gigabytes: Compressing and Indexing Documents and Images* (2. Auflage). Morgan Kaufmann.

Wittgenstein, L. (1922). *Tractatus Logico-Philosophicus* (2. Auflage). Routledge and Kegan Paul. Reprinted 1971, edited by D. F. Pears and B. F. McGuinness. Diese Auflage der englischen Übersetzung enthält auch den deutschen Originaltext auf den gegenüberliegenden Seiten sowie das Vorwort von Bertrand Russell zur Auflage von 1922.

Wittgenstein, L. (1953). *Philosophical Investigations*. Macmillan.

Wojciechowski, W. S. und Wojcik, A. S. (1983). Automated design of multiple-valued logic circuits by automated theorem proving techniques. *IEEE Transactions on Computers*, *C-32*(9), 785–798.

Wolfe, J. und Russell, S. J. (2007). Exploiting belief state structure in graph search. In *ICAPS Workshop on Planning in Games*.

Woods, W. A. (1973). Progress in natural language understanding: An application to lunar geology. In *AFIPS Conference Proceedings*, Band 42, S. 441–450.

Woods, W. A. (1975). What's in a link? Foundations for semantic networks. In Bobrow, D. G. und Collins, A. M. (Hrsg.), *Representation and Understanding: Studies in Cognitive Science*, S. 35–82. Academic Press.

Wooldridge, M. (2002). *An Introduction to MultiAgent Systems*. Wiley.

Wooldridge, M. und Rao, A. (Hrsg.). (1999). *Foundations of rational agency*. Kluwer.

Wos, L., Carson, D. und Robinson, G. (1964). The unit preference strategy in theorem proving. In *Proc. Fall Joint Computer Conference*, S. 615–621.

Wos, L., Carson, D. und Robinson, G. (1965). Efficiency and completeness of the set-of-support strategy in theorem proving. *JACM*, *12*, 536–541.

Wos, L., Overbeek, R., Lusk, E. und Boyle, J. (1992). *Automated Reasoning: Introduction and Applications* (2. Auflage). McGraw-Hill.

Wos, L. und Robinson, G. (1968). Paramodulation and set of support. In *Proc. IRIA Symposium on Automatic Demonstration*, S. 276–310.

Wos, L., Robinson, G., Carson, D. und Shalla, L. (1967). The concept of demodulation in theorem proving. *JACM*, *14*, 698–704.

Wos, L. und Winker, S. (1983). Open questions solved with the assistance of AURA. In *Automated Theorem Proving: After 25 Years: Proc. Special Session of the 89th Annual Meeting of the American Mathematical Society*, S. 71–88. American Mathematical Society.

Wos, L. und Pieper, G. (2003). *Automated Reasoning and the Discovery of Missing and Elegant Proofs*. Rinton Press.

Wray, R. E. und Jones, R. M. (2005). An introduction to Soar as an agent architecture. In Sun, R. (Hrsg.), *Cognition and Multi-agent Interaction: From Cognitive Modeling to Social Simulation*, S. 53–78. Cambridge University Press.

Wright, S. (1921). Correlation and causation. *J. Agricultural Research*, *20*, 557–585.

Wright, S. (1931). Evolution in Mendelian populations. *Genetics*, *16*, 97–159.

Wright, S. (1934). The method of path coefficients. *Annals of Mathematical Statistics*, *5*, 161–215.

Wu, D. (1993). Estimating probability distributions over hypotheses with variable unification. In *IJCAI-93*, S. 790–795.

Wu, F. und Weld, D. S. (2008). Automatically refining the wikipedia infobox ontology. In *17th World Wide Web Conference (WWW2008)*.

Yang, F., Culberson, J., Holte, R., Zahavi, U. und Felner, A. (2008). A general theory of additive state space abstractions. *JAIR*, *32*, 631–662.

Yang, Q. (1990). Formalizing planning knowledge for hierarchical planning. *Computational Intelligence*, *6*, 12–24.

Yarowsky, D. (1995). Unsupervised word sense disambiguation rivaling supervised methods. In *ACL-95*, S. 189–196.

Yedidia, J., Freeman, W. und Weiss, Y. (2005). Constructing free-energy approximations and generalized belief propagation algorithms. *IEEE Transactions on Information Theory*, *51*(7), 2282–2312.

Yip, K. M.-K. (1991). *KAM: A System for Intelligently Guiding Numerical Experimentation by Computer*. MIT Press.

Yngve, V. (1955). A model and an hypothesis for language structure. In Locke, W. N. und Booth, A. D. (Hrsg.), *Machine Translation of Languages*, S. 208–226. MIT Press.

Yob, G. (1975). Hunt the wumpus! *Creative Computing*, *Sep/Oct*.

Yoshikawa, T. (1990). *Foundations of Robotics: Analysis and Control*. MIT Press.

Young, H. P. (2004). *Strategic Learning and Its Limits*. Oxford University Press.

Younger, D. H. (1967). Recognition and parsing of context-free languages in time n^3. *Information and Control, 10*(2), 189–208.

Yudkowsky, E. (2008). Artificial intelligence as a positive and negative factor in global risk. In Bostrom, N. und Cirkovic, M. (Hrsg.), *Global Catastrophic Risk*. Oxford University Press.

Zadeh, L. A. (1965). Fuzzy sets. *Information and Control*, *8*, 338–353.

Zadeh, L. A. (1978). Fuzzy sets as a basis for a theory of possibility. *Fuzzy Sets and Systems*, *1*, 3–28.

Zaritskii, V. S., Svetnik, V. B. und Shimelevich, L. I. (1975). Monte-Carlo technique in problems of optimal information processing. *Automation and Remote Control*, *36*, 2015–22.

Zelle, J. und Mooney, R. (1996). Learning to parse database queries using inductive logic programming. In *AAAI-96*, S. 1050–1055.

Zermelo, E. (1913). Über eine Anwendung der Mengenlehre auf die Theorie des Schachspiels. In *Proc. Fifth International Congress of Mathematicians*, Band 2, S. 501–504.

Zermelo, E. (1976). An application of set theory to the theory of chess-playing. *Firbush News*, *6*, 37–42. Englische Übersetzung von (Zermelo 1913).

Zettlemoyer, L. S. und Collins, M. (2005). Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *UAI-05*.

Zhang, H. und Stickel, M. E. (1996). An efficient algorithm for unit-propagation. In *Proc. Fourth International Symposium on Artificial Intelligence and Mathematics*.

Zhang, L., Pavlovic, V., Cantor, C. R. und Kasif, S. (2003). Human-mouse gene identification by comparative evidence integration and evolutionary analysis. *Genome Research*, S. 1–13.

Zhang, N. L. und Poole, D. (1994). A simple approach to Bayesian network computations. In *Proc. 10th Canadian Conference on Artificial Intelligence*, S. 171–178.

Zhang, N. L., Qi, R. und Poole, D. (1994). A computational theory of decision networks. *IJAR*, *11*, 83–158.

Zhou, R. und Hansen, E. (2002). Memory-bounded A* graph search. In *Proc. 15th International Flairs Conference*.

Zhou, R. und Hansen, E. (2006). Breadth-first heuristic search. *AIJ*, *170*(4–5), 385–408.

Zhu, D. J. und Latombe, J.-C. (1991). New heuristic algorithms for efficient hierarchical path planning. *IEEE Transactions on Robotics and Automation*, *7*(1), 9–20.

Zimmermann, H.-J. (Hrsg.). (1999). *Practical applications of fuzzy technologies*. Kluwer.

Zimmermann, H.-J. (2001). *Fuzzy Set Theory – And Its Applications* (4. Auflage). Kluwer.

Zinkevich, M., Johanson, M., Bowling, M. und Piccione, C. (2008). Regret minimization in games with incomplete information. In *NIPS 20*, S. 1729–1736.

Zollmann, A., Venugopal, A., Och, F. J. und Ponte, J. (2008). A systematic comparison of phrase-based, hierarchical and syntax-augmented statistical MT. In *COLING-08*.

Zweig, G. und Russell, S. J. (1998). Speech recognition with dynamic Bayesian networks. In *AAAI-98*, S. 173–180.

Personenregister

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z

Register

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Copyright