Vorbesprechung

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S

Wozu studieren wir Algorithmen und Datenstrukturen?

Motivation:

- Wichtig für fast alle Zweige der Informatik
- Schlüsselrolle in moderner technologischer Innovation
- Sicht auf Prozesse außerhalb von Informatik und Technik (Quantenmechanik, Wirtschaftsmodelle, Evolution, ...)
- Herausfordernd und macht Spaß

Agenda

Das AlgoDat-Team

Organisation

Beurteilung

Allgemeines

Das AlgoDat-Team

Stefan Szeider

Günther Raidl

Martin Nöllenburg

Martin Kronegger

Michael Bernreiter

Cornelius Brand

Alexander Dobler

Jan Dreier

Wolfgang Dvořák

Robert Ganian

Enrico Iurlano

Friedrich Slivovsky

P. R. Vaidyanathan

Manuel Sorge

Markus Wallinger

Asinger, Bankosegger, Bauer, Capova, Depian, Fenz, Guttmann, Haumer, Lindner, Mayerl, Schober, Seeliger, Uhlig, Zwick

TISS und TUWEL

- **TISS:** Campusinformationssystem der TU Wien (`https://tiss.tuwien.ac.at`)
  - Anmeldung zur LVA
  - Anmeldung zur Übungsgruppe
  - Anmeldung zu Tests

- **TUWEL:** E-Learning Plattform der TU Wien (`https://tuwel.tuwien.ac.at`)
  - Vorlesungsfolien und Aufzeichnungen der Vorlesung
  - Eingangstest
  - Aussendungen (Nachrichtenforum) $\rightarrow$ in TUWEL hinterlegte E-Mail Adresse regelmäßig abfragen!
  - Diskussionsforum
  - Übungsblätter und Programmieraufgaben: Angaben, Hochladen der Lösungen, Ankreuzen der gelösten Aufgaben, Tafelleistung
  - Anmeldung zu Abgabegespräche für Programmieraufgaben
  - Links zu relevanten Zoom Meetings
  - Punkte und Ergebnisse
  - Weiterführende Literatur / Materialien
  - etc.

Überblick

- Vorlesung mit Übung
  - LVA-Anmeldung in TISS (bis So. 05.03.2023 um 23:59 Uhr)
  - Nach Anmeldung: Zugang zum TUWEL-Kurs und weitere Teilnahme
  - Arbeitsaufwand: 8 ECTS entsprechen ca. 200h (ECTS-breakdown in TISS)
- Vorlesungen (keine Anwesenheitspflicht)
  - dienstags 12-14 Uhr (c.t.)
  - donnerstags 11-13 Uhr (c.t.)
- Leistungsüberprüfung
  - Eingangstest
  - 2+1 schriftliche Teilprüfungen („Tests“)
  - 7 Übungsblätter
  - 7 Programmieraufgaben

Modus: Präsenz vs. Distance Learning

	Präsenz	Distance Learning
Vorlesung	Hörsaal	Zoom
Übung	Hörsaal Beginn: c.t.	Zoom Beginn: s.t.
Abgabegespräche Programmieraufgaben	Zoom	Zoom
Tests	Hörsaal	Zoom +2h Zeit Puffer

- Derzeitiger Stand: alles im Präsenzmodus
  - Modusänderung für Teile möglich, z.B.: alles bis auf Tests in Distance Learning
- Zoom:
  - Hinweis: Aufzeichnungen sind nicht erlaubt (generell, trifft nicht nur bei AlgoDat zu)
  - Ausnahme: wir zeichnen die Vorlesungen und Repetitorien auf ($\rightarrow$ TUWEL)
  - Unbedingt vollständigen Namen anzeigen, sonst kein Einlass (Ausnahme: Vorlesung)
    $\rightarrow$ Zoom-Pro Account der TU Wien (`http://www.sss.tuwien.ac.at/sss/zoo/`)
    $\rightarrow$ per Single-Sign-On (SSO) einloggen (nicht: Account mit TU-Adresse)
- Kopie von Studierendenausweis in TUWEL hochladen (Online-Identitätskontrolle)

Eingangstest

- In TUWEL zu absolvieren
  - Ab heute 20 Uhr **bis Fr. 17.03.2023, 20 Uhr**
  - Bis zur Deadline beliebig viele Versuche, bestes Ergebnis zählt
- **Voraussetzung für die weitere Teilnahme an der LVA**
  - Mindestanforderung: 80%
- **Es wird ein Zeugnis ausgestellt, falls Ergebnis $\geq$80%**
- Themen:
  - Mathematische Grundlagen (Wissen setzen wir voraus)
    - Rechenregeln für Exponenten
    - Rechenregeln für Logarithmen
    - Reihen
    - Rechenregeln für Modulorechnung
  - Laufzeitfunktionen (wird rechtzeitig in der Vorlesung behandelt)
    - Laufzeitfunktionsgraphen
    - Laufzeitabschätzung
    - Laufzeiten von Algorithmen
    - Notationen

Schriftliche Teilprüfungen („Tests“) (1/3)

Termine:

- Test 1: Fr. 12.05.2023, 17 Uhr (c.t.)
- Test 2: Fr. 30.06.2023, 17 Uhr (c.t.)
- Nachtragstest: wahrscheinlich Fr. 29.9.2023, 17 Uhr (c.t.) ($\rightarrow$ TUWEL)
  - Ein Test über gesamten Inhalt der LVA

Ablauf:

- Voraussetzung für alle Tests: Eingangstest $\geq$80% und positive Note möglich
- Schriftlich (auch online auf Papier), Dauer jeweils ca. 90 Minuten
- Jeweils maximal 100 Punkte
- Keine Unterlagen, keine Hilfsmittel erlaubt
- Infos zur Anmeldung und weitere Details kommen in TUWEL-Aussendung

Schriftliche Teilprüufungen („Tests“) (2/3)

Inhalt:

- Inhalte der Vorlesung und der Übung
- Schwerpunkt liegt auf Verständnis, nicht auf Reproduzieren alter Aufgaben
  $\rightarrow$ bei jedem Test werden neue Aufgaben erstellt

Repetitorien:

- Wiederholung vor den Tests (keine Anwesenheitspflicht)
- Termine: siehe TUWEL

Schriftliche Teilprüfungen („Tests“) (3/3)

Voraussetzungen / notwendiges Equipment bei Distance Learning:

- Stabile Internetverbindung und Kamera für Zoom
- Heller und ruhiger Arbeitsplatz (während der Prüfung alleine und ungestört)
- Bildschirm oder Drucker für Angaben (im Kamerabild)
- Kamera:
  - weit genug weg, seitlich/schräg vorne, im Bild: Sie inkl. Hände, Bildschirm, Arbeitsbereich
  - schwenkbar, sodass ganzer Raum abgefilmt werden kann (nur nach Aufforderung)
- Scanner (empfohlen: Smartphone mit Scanner-App; scannen im Kamerabild)
- Falls nicht möglich: mind. drei Wochen vorher E-Mail an algodat@ac.tuwien.ac.at

Übungen in Kleingruppen (1/3)

- Kleingruppen mit je ca. 25 TeilnehmerInnen (**Anwesenheitspflicht**)
  - Anwesenheitspflicht bedeutet: nur bei Anwesenheit können Sie Punkte bekommen; $\rightarrow$ bei Fehlen **keine** Entschuldigung / Krankmeldung / etc. notwendig
- Voraussetzung: Anmeldung zu einer Übungsgruppe ($\rightarrow$ positiver Eingangstest)
  - **Anmeldezeitraum:** Mo. 20.03.2023, 20 Uhr bis Mo. 27.03.2023, 20 Uhr
  - Platzvergabe nach „first come, first served“
  - Eigenverantwortlichkeit bei Terminüberschneidungen mit anderen LVAs
- Bei Verhinderung: **Kolloquium am Semesterende** für max. eine Übung
  - Voraussetzungen: PDF mit Lösungen zur ursprünglichen Deadline in TUWEL hochgeladen und Aufgaben in TUWEL gekreuzt, positive Gesamtnote noch möglich

Ablauf der Abgabe:

- **Abgabe und Ankreuzen** ausgearbeiteter Aufgaben **in TUWEL**
  - Hochladen einer einzigen PDF-Datei pro Übung (≤ 10 MB)
  - Eingescannte bzw. abfotografierte Lösungen sind zulässig, sofern diese gut lesbar sind
- **Deadline: Montag 20 Uhr in der Übungswoche**
  - **Deadlines sind strikt!** Es werden keine Nachabgaben akzeptiert

Übungen in Kleingruppen (2/3)

Ankreuzen von Aufgaben, wenn:

- alle Teilaufgaben gelöst wurden,
- die Lösung im hochgeladenen PDF enthalten ist,
- der Lösungsweg ($\neq$ Endergebnis) erkennbar ist und
- die Lösung eine Eigenleistung ist (kein Plagiat, nicht mit AI-Tool erstellt, etc.)

In der Übung:

- Präsentation der Aufgaben durch Studierende
- Fragen zur Aufgabe und der jeweiligen Stoffgebiete
- Bewertung der Tafelleistung (0 bis 100%)
- eine korrekte Lösung ist nicht zwingend erforderlich für positive Tafelleistung
- arbeiten Sie mit (bemerkbar machen, ÜbungsgruppenleiterIn nimmt Sie dran)

Übungen in Kleingruppen (3/3)

„Spielregeln“ in der Übung:

- gekreuzte Aufgaben zählen nur bei Anwesenheit während gesamter Präsentation
- Distance Learning:
  - Kamera an (sonst keine Anwesenheit)
  - Mikrofon aus (Ausnahme: wenn Sie präsentieren oder etwas beitragen)
- Wenn eine angekreuzte Aufgabe nicht präsentiert werden will:
  - Verlust aller angekreuzten Aufgaben des jeweiligen Übungsblattes
  - 0% Tafelleistung
- Wenn ein angekreuztes nicht präsentiert werden kann, da (fast) kein Wissen zu der Aufgabe bzw. zu den jeweiligen Themengebieten vorhanden ist
  - Verlust der angekreuzten Aufgabe (in Ausnahmefällen können ÜbungsgruppenleiterInnen auch alle Aufgaben des aktuellen Übungsblattes aberkennen)
  - 0% Tafelleistung

Programmieraufgaben

- 7 Programmieraufgaben, dazu 2 Abgabegespräche
  - Programmieraufgaben 1 bis 3
  - Programmieraufgaben 4 bis 7
- Fragestunden vor der Deadline
- Um Punkte zu erhalten: erfolgreich am Abgabegespräch teilnehmen
- Hochgeladener Source-Code muss mit Framework aus TUWEL kompilieren und lauffähig sein

Inhalt:

- Algorithmen implementieren
- Java-Framework wird zur Verfügung gestellt
- Fragen in den Angaben sind in einem (kurzen) Bericht zu beantworten und als PDF hochzuladen

Eigenständige Leistung

Plagiate, Lösungen von AI-Tools, etc. werden ausnahmslos nicht toleriert!

- Gemeinsames Überlegen und Diskutieren ist in Ordnung
- **Wir erwarten eigenständiges Lösen der Aufgaben**
  $\rightarrow$ Ausarbeitungen und Programme müssen selbstständig erstellt werden
- Auch Kopieren einzelner (Teil-)Aufgaben ist ein Plagiat
- Minimale Änderungen am Source-Code (Variablenumbenennungen, Umformatierungen, etc.) sind keine eigenständige Leistung

Konsequenzen: Plagiat / Lösung von AI-Tool führt zum Aberkennen aller Aufgaben des Übungsblattes bzw. aller Punkte des Abgabegesprächs für alle beteiligten Personen!

Bonuspunkte

- (Freiwillige) TUWEL Online-Tests
- Zählen nur bei bereits positiver Gesamtnote
- Je Test gilt:
  - Maximal 10 Versuche
  - Bester Versuch zählt
  - Deadline: 14.07.2023 um 20 Uhr
  - x Prozent ergeben folgende Punkte:

    $x \in [80, 90)$ 1 Punkt
    $x \in [90, 100)$ 2 Punkte
    $x = 100$ 3 Punkte

Beurteilung

Beurteilungskriterium	Maximum	Mindestanforderung
Bester schriftlicher Test	100	50
Zweitbester schriftlicher Test	100	50
Anzahl korrekt gekreuzter Aufgaben	42	25
Punkte durchschnittliche Tafelleistung	25	15
Programmieraufgaben	33	0
Summe	300	

Berechnung der Punkte für die durchschnittliche Tafelleistung d: $\left\lfloor \frac{d}{4} \right\rfloor$

Zumindest eine Mindestanforderung nicht erfüllt: N5 Nicht genügend

Wenn alle Mindestanforderungen erfüllt:

Summe der Punkte aller Beurteilungskriterien und Bonuspunkte

$[260, 300]$ S1 Sehr gut
$[220, 259]$ U2 Gut
$[180, 219]$ B3 Befriedigend
$[140, 179]$ G4 Genügend

Bei Fragen und Problemen...

1. Fragen Sie eine Kollegin / einen Kollegen
2. Fragen Sie uns vor, während oder nach der Vorlesung / Übung
3. Bei allgemeinen Fragen: nutzen Sie das TUWEL-Diskussionsforum (helfen Sie sich auch gegenseitig!)
4. Bei individuellen Anfragen: E-Mail von Ihrer Studierenden E-Mail Adresse an algodat@ac.tuwien.ac.at
5. Kommen Sie in die Online-Sprechstunde (siehe TUWEL)

Inhalte (1/2)

- Einführung: Stable Matching
- Analyse von Algorithmen
  - Asymptotische Komplexität
  - Ω, O, Θ-Notationen
- Graphen und Graphenalgorithmen
  - Tiefen- und Breitensuche
- Algorithmen-Paradigmen
  - Greedy
  - Divide-and-Conquer
- Suche
  - Binäre Suchbäume
  - Balancierte Suchbäume (AVL, B, B*)
  - Hashverfahren
- Datenstrukturen in der praktischen Anwendung

Inhalte (2/2)

- Polynomialzeitreduktion
- Effizient lösbare Spezialfälle schwerer Probleme
- Effiziente Lösung von Optimierungsproblemen
  - Branch-and-Bound
  - Dynamische Programmierung
  - Approximation
  - Heuristische Verfahren

Wir wünschen Ihnen viel Erfolg und
ein spannendes Semester!

Das Stable-Matching-Problem

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 1. März 2023
Vorlesungsfolien

Stable-Matching-Problem

Gegeben seien n Kinder und n Gastfamilien, die an einem Austauschprogramm für Schülerinnen und Schülern teilnehmen.

Ziel: Finde eine passende Zuordnung von Kindern zu Gastfamilien.

- Jedes Kind hat eine Präferenzliste von Gastfamilien.
- Jede Gastfamilie hat eine Präferenzliste von Kindern.

Kinder: Xaver, Yvonne, Zola.

Gastfamilien: Abel, Boole, Church.

höchste Präferenz ↓ niedrigste Präferenz ↓

	1.	2.	3.
Xaver	Abel	Boole	Church
Yvonne	Boole	Abel	Church
Zola	Abel	Boole	Church

Präferenzlisten der Kinder

höchste Präferenz ↓ niedrigste Präferenz ↓

	1.	2.	3.
Abel	Yvonne	Xaver	Zola
Boole	Xaver	Yvonne	Zola
Church	Xaver	Yvonne	Zola

Präferenzlisten der Familien

Stable-Matching-Problem

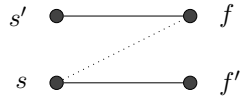
Perfektes Matching: Jedem Kind wird genau eine Familie zugewiesen.

- Jedes Kind bekommt genau eine Gastfamilie.
- Jede Gastfamilie bekommt genau ein Kind.

Instabiles Paar:

- In einem Matching M ist ein nicht zugewiesenes Paar s-f instabil, wenn ein Kind s und eine Familie f sich gegenseitig gegenüber ihren aktuellen Partnern bevorzugen.
- Das instabile Paar s-f könnte seine Situation durch Verlassen der aktuellen Partner verbessern.

s für *student*, f für *family*



Stable-Matching-Problem

Stable Matching: Perfektes Matching ohne instabile Paare. Es besteht daher für kein Paar der Anreiz, durch gemeinsames Handeln die Zuteilung zu unterlaufen.

Stable-Matching-Problem: Ausgehend von den Präferenzlisten von n Kindern und n Familien, finde ein Stable Matching, wenn eines existiert.

höchste Präferenz ↓ niedrigste Präferenz ↓

	1.	2.	3.
Xaver	Abel	Boole	Church
Yvonne	Boole	Abel	Church
Zola	Abel	Boole	Church

Präferenzlisten der Kinder

höchste Präferenz ↓ niedrigste Präferenz ↓

	1.	2.	3.
Abel	Yvonne	Xaver	Zola
Boole	Xaver	Yvonne	Zola
Church	Xaver	Yvonne	Zola

Präferenzlisten der Familien

Stable-Matching-Problem

Frage: Ist die Zuordnung X-C, Y-B, Z-A stabil?

höchste Präferenz ↓ niedrigste Präferenz ↓

	1.	2.	3.
Xaver	Abel	Boole	Church
Yvonne	Boole	Abel	Church
Zola	Abel	Boole	Church

Präferenzlisten der Kinder

höchste Präferenz ↓ niedrigste Präferenz ↓

	1.	2.	3.
Abel	Yvonne	Xaver	Zola
Boole	Xaver	Yvonne	Zola
Church	Xaver	Yvonne	Zola

Präferenzlisten der Familien

Stable-Matching-Problem

Frage: Ist die Zuordnung X-C, Y-B, Z-A stabil?
Antwort: Nein. Boole und Xaver können ihre Situation verbessern (Xaver-Boole ist ein instabiles Paar).

höchste Präferenz ↓ niedrigste Präferenz ↓

	1.	2.	3.
Xaver	Abel	Boole	Church
Yvonne	Boole	Abel	Church
Zola	Abel	Boole	Church

Präferenzlisten der Kinder

höchste Präferenz ↓ niedrigste Präferenz ↓

	1.	2.	3.
Abel	Yvonne	Xaver	Zola
Boole	Xaver	Yvonne	Zola
Church	Xaver	Yvonne	Zola

Präferenzlisten der Familien

Stable-Matching-Problem

Frage: Ist die Zuordnung X-A, Y-B, Z-C stabil?
Antwort: Ja. Es gibt kein instabiles Paar.

höchste Präferenz ↓ niedrigste Präferenz ↓

	1.	2.	3.
Xaver	Abel	Boole	Church
Yvonne	Boole	Abel	Church
Zola	Abel	Boole	Church

Präferenzlisten der Kinder

höchste Präferenz ↓ niedrigste Präferenz ↓

	1.	2.	3.
Abel	Yvonne	Xaver	Zola
Boole	Xaver	Yvonne	Zola
Church	Xaver	Yvonne	Zola

Präferenzlisten der Familien

Stable-Matching-Problem: Fragen

Frage: Gibt es immer ein Stable Matching?
Hinweis: Das ist nicht von vornherein klar!

Frage: Kann ein Stable Matching effizient gefunden werden?
Hinweis: Brute-Force-Ansatz (alle möglichen Zuordnungen ausprobieren) betrachtet $n!$ viele mögliche Lösungen, was extrem ineffizient ist.

Gale-Shapley-Algorithmus: Wir stellen einen Algorithmus vor, mit dem wir beide Fragen mit „Ja“ beantworten können.

Gale–Shapley-Algorithmus (GS-Algorithmus)

Gale-Shapley-Algorithmus:

- 1962 gaben David Gale und Lloyd Shapley einen Algorithmus zum Auffinden von Stable Matchings an.
- Shapley bekam für seine Arbeiten (einschließlich Stable Matching) den Wirtschaftsnobelpreis 2012.

  D. Gale and L. S. Shapley: College Admissions and the Stability of Marriage, American Mathematical Monthly, Vol. 69, 1962, Seite 9–15

Anwendung: Das Stable-Matching-Problem hat viele Anwendungen, z.B. bei der Zuteilung von Medizinstudenten an das erste Krankenhaus, in dem sie ihren Turnus ableisten.

Gale and Shapley 1962

COLLEGE ADMISSIONS AND THE STABILITY OF MARRIAGE

D. GALE* AND L. S. SHAPLEY, Brown University and the RAND Corporation

1. Introduction. The problem with which we shall be concerned relates to the following typical situation: A college is considering a set of n applicants of which it can admit a quota of only q. Having evaluated their qualifications, the admissions office must decide which ones to admit. The procedure of offering admission only to the q best-qualified applicants will not generally be satisfactory, for it cannot be assumed that all who are offered admission will accept. Accordingly, in order for a college to receive q acceptances, it will generally have to offer to admit more than q applicants. The problem of determining how many and which ones to admit requires some rather involved guesswork. It may not be known (a) whether a given applicant has also applied elsewhere; if this is known it may not be known (b) how he ranks the colleges to which he has applied; even if this is known it will not be known (c) which of the other colleges will offer to admit him. A result of all this uncertainty is that colleges can expect only that the entering class will come reasonably close in numbers to the desired quota, and be reasonably close to the attainable optimum in quality.

Gale–Shapley-Algorithmus (GS-Algorithmus)

Gale-Shapley-Algorithmus:

```
Kennzeichne jede Familie/jedes Kind als frei
while ein Kind ist frei und kann noch eine Familie wählen
    Wähle solch ein Kind s aus
    f ist erste Familie in der Präferenzliste von s,
    die von s noch nicht gewählt wurde
    if f ist frei
        Kennzeichne s und f als einander zugeordnet
    elseif f bevorzugt s gegenüber ihrem aktuellen Partner s'
        Kennzeichne s und f als einander zugeordnet und s' als frei
    else
        f weist s zurück
```

Beispiel für Ablauf

Ausgangssituation:

- 4 Kinder (W-Z) mit Präferenzlisten (links).
- 4 Familien (A-D) mit Präferenzlisten (rechts).

	1.	2.	3.	4.
W	B	A	C	D
X	C	B	A	D
Y	B	C	A	D
Z	B	A	D	C

	1.	2.	3.	4.
A	X	W	Y	Z
B	W	Y	X	Z
C	Z	Y	W	X
D	X	W	Y	Z

Beispiel für Ablauf

Erste Zuordnung:

- Wähle erstes freies Kind aus (z.B. W).
- Dieses wählt die erste Familie in seiner Präferenzliste aus (in diesem Fall B).
- Da B frei ist, werden die beiden einstweilig einander zugeordnet.
- Aktuelle Zuordnungen: W-B.

	1.	2.	3.	4.
W	B	A	C	D
X	C	B	A	D
Y	B	C	A	D
Z	B	A	D	C

	1.	2.	3.	4.
A	X	W	Y	Z
B	W	Y	X	Z
C	Z	Y	W	X
D	X	W	Y	Z

Beispiel für Ablauf

Zweite Zuordnung:

- Wähle nächstes freies Kind aus (z.B. X).
- Dieses wählt die erste Familie in seiner Präferenzliste aus (in diesem Fall C).
- Da C frei ist, werden die beiden einstweilig einander zugeordnet.
- Aktuelle Zuordnungen: W-B, X-C.

	1.	2.	3.	4.
W	B	A	C	D
X	C	B	A	D
Y	B	C	A	D
Z	B	A	D	C

	1.	2.	3.	4.
A	X	W	Y	Z
B	W	Y	X	Z
C	Z	Y	W	X
D	X	W	Y	Z

Beispiel für Ablauf

Dritte Zuordnung:

- Wähle nächstes freies Kind aus (z.B. Y).
- Dieses wählt die erste Familie in seiner Präferenzliste aus (in diesem Fall B).
- B ist aber W zugeordnet. In der Präferenzliste von B steht W vor Y, daher lässt B das Y abblitzen.
- Y wählt die nächste Familie in seiner Präferenzliste aus (in diesem Fall C).
- C bevorzugt Y vor ihrem Partner X, daher wird ihre Zuordnung zu X gelöst und statt dessen werden C und Y einander zugeordnet.
- Aktuelle Zuordnungen: W-B, Y-C.

	1.	2.	3.	4.
W	B	A	C	D
X	C	B	A	D
Y	B	C	A	D
Z	B	A	D	C

	1.	2.	3.	4.
A	X	W	Y	Z
B	W	Y	X	Z
C	Z	Y	W	X
D	X	W	Y	Z

Beispiel für Ablauf

Vierte Zuordnung:

- Wähle nächstes freies Kind aus, es ist X, das wieder frei geworden ist.
- X wählt die zweite Familie (B) auf seiner Präferenzliste aus.
- B bevorzugt W vor X.
- X wählt die dritte Familie (A) auf seiner Präferenzliste aus.
- A ist frei und die beiden werden einander zugeordnet.
- Aktuelle Zuordnungen: W-B, X-A, Y-C.

	1.	2.	3.	4.
W	B	A	C	D
X	C	B	A	D
Y	B	C	A	D
Z	B	A	D	C

	1.	2.	3.	4.
A	X	W	Y	Z
B	W	Y	X	Z
C	Z	Y	W	X
D	X	W	Y	Z

Beispiel für Ablauf

Fünfte Zuordnung:

- Wähle nächstes freies Kind aus (nur mehr Z übrig).
- Z wählt die erste Familie (B) auf seiner Präferenzliste aus. B bevorzugt aber W vor Z.
- Z wählt die zweite Familie (A) auf seiner Präferenzliste aus. A bevorzugt aber X vor Z.
- Z wählt die dritte Familie (D) auf seiner Präferenzliste aus.
- D ist frei, also werden Z und D einander zugeordnet.
- Aktuelle Zuordnungen: W-B, X-A, Y-C, Z-D.
- Es ist nun kein Kind mehr frei, und der Algorithmus terminiert. Wir haben ein Stable Matching gefunden.

	1.	2.	3.	4.
W	B	A	C	D
X	C	B	A	D
Y	B	C	A	D
Z	B	A	D	C

	1.	2.	3.	4.
A	X	W	Y	Z
B	W	Y	X	Z
C	Z	Y	W	X
D	X	W	Y	Z

Korrektheitsbeweis: Terminierung

Operation 1: Kinder wählen Familien in absteigender Reihenfolge aus.

Operation 2: Sobald eine Familie zugewiesen wurde, bleibt sie zugewiesen, die Zuteilung kann sich aber ändern.

Behauptung: Algorithmus terminiert nach höchstens n^2 Iterationen der while-Schleife.

Beweis: In jeder Iteration der while-Schleife wählt ein Kind eine Familie aus. Es gibt nur n^2 Möglichkeiten dafür. □

	1.	2.	3.	4.	5.
Valentin	A	B	C	D	E
Werner	B	C	D	A	E
Xaver	C	D	A	B	E
Yvonne	D	A	B	C	E
Zola	A	B	C	D	E

	1.	2.	3.	4.	5.
Abel	W	X	Y	Z	V
Boole	X	Y	Z	V	W
Church	Y	Z	V	W	X
Dijkstra	Z	V	W	X	Y
Euler	V	W	X	Y	Z

$n(n-1)+1$ vorläufige Zuordnungen erfoderlich

Korrektheitsbeweis: Abschluss

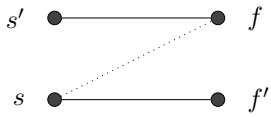
Behauptung: Alle Kinder und Familien werden zugewiesen.
Beweis: (durch Widerspruch)

- Angenommen, Kind s wurde nach Terminierung des Algorithmus nicht zugewiesen.
- Dann wurde auch eine Familie (z.B. f) nach Terminierung des Algorithmus nicht zugewiesen.
- Damit wurde f nie ausgewählt.
- Aber s hat jede Familie in der Liste ausgewählt, da es ja am Ende nicht zugewiesen wurde. $\square$

Korrektheitsbeweis: Stabilität

Behauptung: Nachdem der Algorithmus terminiert, existieren keine instabilen Paare.
Beweis: (durch Widerspruch)

- Angenommen, s-f ist ein instabiles Paar: s bevorzugt f gegenüber seinem aktuellen Partner f' und f bevorzugt s gegenüber seinem aktuellen Partner s' in einem Gale-Shapley-Matching.



Korrektheitsbeweis: Stabilität

Behauptung: Nachdem der Algorithmus terminiert, existieren keine instabilen Paare.
Beweis: (durch Widerspruch)

- Angenommen, s-f ist ein instabiles Paar: s bevorzugt f gegenüber seinem aktuellen Partner f' und f bevorzugt s gegenüber seinem aktuellen Partner s' in einem Gale-Shapley-Matching.
- Fall 1: s hat f nie ausgewählt.
  - $\Rightarrow$ s bevorzugt seinen GS-Partner f' gegenüber f.
  - $\Rightarrow$ s-f ist nicht instabil.
- Fall 2: s hat f ausgewählt.
  - $\Rightarrow$ f hat s zurückgewiesen (gleich oder später)
  - $\Rightarrow$ f bevorzugt s' gegenüber s.
  - $\Rightarrow$ s-f ist nicht instabil.
- In jedem Fall ist s-f nicht instabil, was ein Widerspruch ist. □

Kinder wählen Familien in absteigender Reihenfolge der Präferenzen aus.
Bei Familien kann sich die Situation nur verbessern

Zusammenfassung

Stable-Matching-Problem: Gegeben seien n Kinder und n Familien und ihre Präferenzen. Finde ein Stable Matching, wenn eines existiert.

Frage: Gibt es immer ein Stable Matching?

Frage: Kann ein Stable Matching effizient gefunden werden?

Gale-Shapley-Algorithmus: Findet garantiert ein Stable Matching für jede Problemeingabe (in Form von Präferenzlisten). Da der Algorithmus höchstens n^2 Iterationen benötigt, findet er ein Stable Matching auf effiziente Weise.

Effiziente Implementierung

Zeitaufwand:

- Stable Matching benötigt höchstens n^2 Iterationen.
- Einzelne Schritte in einer Iteration können naiv (z.B. lineare Suche in Listen) implementiert werden, das benötigt eine "Größenordnung" von n Schritten.
- Dann benötigt man insgesamt höchstens eine "Größenordnung" von n^3 Schritten.
- Das ist immer noch besser, als ein Brute-Force-Ansatz der alle möglichen Zuordnungen durchprobiert (es gibt $n!$ mögliche Zuordnungen).

Nächste Vorlesung:

- Mit asymptotischer Analyse können wir das exakt ausdrücken.
- Mit besseren Datenstrukturen können wir das auch schneller lösen.

Graphen

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 9. März 2023
Vorlesungsfolien

Grundlegende Definitionen und Anwendungen

Graphen

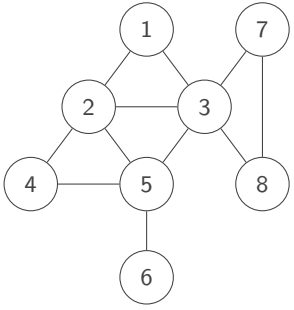
Graphen: Graphen sind ein wichtiges Werkzeug um Netzwerke, Zusammenhänge und Strukturen zu modellieren.

Beispiel: Wiener U-Bahn Linien

Ungerichtete Graphen

Ungerichteter Graph: $G = (V, E)$

- V = Menge der Knoten (*vertices, nodes*).
- E = Menge der Kanten zwischen Paaren von Knoten (*edges*).
- Notation für Kante zwischen Knoten a und b: (a, b) bzw. (b, a).
- Alternativ wird auch $a - b$ bzw. $b - a$ verwendet.
- Parameter für Größen: $n = |V|$, $m = |E|$



$V = \{1,2,3,4,5,6,7,8\}$

$E = \{1\text{-}2,1\text{-}3,2\text{-}3,2\text{-}4,2\text{-}5,3\text{-}5,3\text{-}7,3\text{-}8,4\text{-}5,5\text{-}6,7\text{-}8\}$

$n = 8$

$m = 11$

Ungerichtete Graphen: Weitere Definitionen

Adjazent, inzident, Nachbarschaft: Sei $e = (u, v)$ eine Kante in E.

- u und v sind adjazent, d.h. u ist Nachbar von v und v ist Nachbar von u.
- v (bzw. u) und e sind inzident.
- $(u, v) = (v, u)$.

Knotengrad (*degree*): $deg(v)$ bezeichnet den Knotengrad des Knotens v.

- $deg(v)$ entspricht der Anzahl der zu v inzidenten Kanten.
- Es gilt: $\sum_{v \in V} deg(v) = 2 \cdot \mid E \mid$ (Handshaking-Lemma).

Ungerichtete Graphen: Weitere Definitionen

Grundlegende Definitionen:

- Mehrfachkante: Mehrere Kanten zwischen zwei Knoten.
- Schleife: Eine Kante, die einen Knoten mit sich selbst verbindet.

Schlichter Graph: Ein ungerichteter Graph ohne Mehrfachkanten und ohne Schleifen.

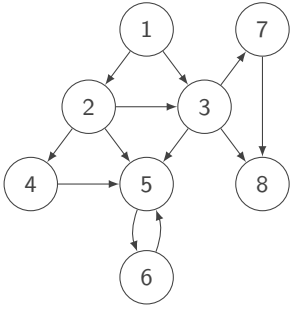
Hinweise:

- In dieser Vorlesung werden, wenn nicht anders verlautbart, schlichte Graphen betrachtet.
- Bei bestimmten Problemstellungen werden gewichtete Graphen verwendet, bei denen Knoten und/oder Kanten eine reelle Zahl zugeordnet bekommen.

Gerichtete Graphen

Gerichteter Graph (Digraph): $G = (V, E)$

- V = Menge der Knoten (*vertices, nodes*).
- E = Menge der gerichtete Kanten (*arcs*) zwischen Paaren von Knoten.
- Notation für Kante von a zu b: (a, b) bzw. $a \to b$
- $(a, b) \neq (b, a)$



$V = \{1,2,3,4,5,6,7,8\}$

$E = \{1 \to 2, 1 \to 3, 2 \to 3, 2 \to 4, 2 \to 5, 3 \to 5,$
$3 \to 7, 3 \to 8, 4 \to 5, 5 \to 6, 6 \to 5, 7 \to 8\}$

$n = 8$

$m = 12$

Hinweis: Kanten in entgegengesetzter Richtung sind auch in schlichten Digraphen erlaubt.

Gerichtete Graphen: Weitere Definitionen

Eingangsknotengrad: $deg^-(v)$ ist die Anzahl der eingehenden inzidenten Kanten.

Ausgangsknotengrad: $deg^+(v)$ ist die Anzahl der ausgehenden inzidenten Kanten.

Es gilt: $deg(v) = deg^+(v) + deg^-(v)$.

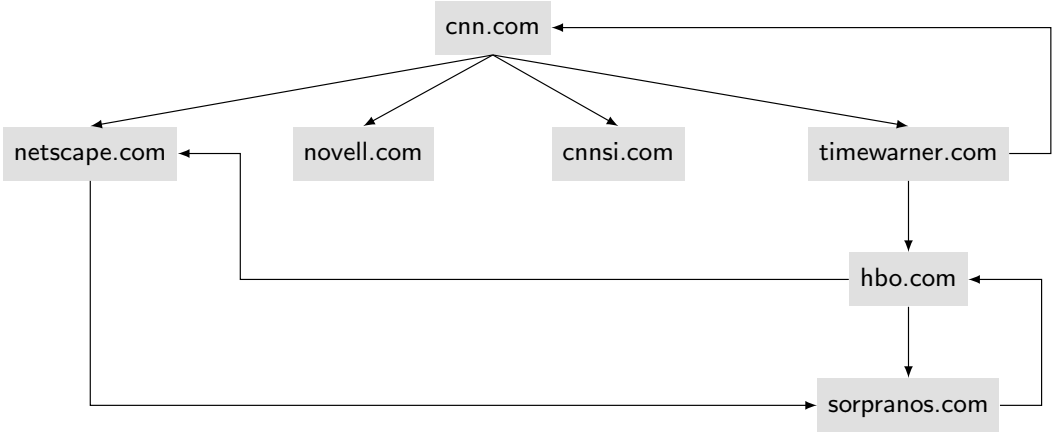
Einige Anwendungen von Graphen

Graph	*Knoten*	*Kanten*
Verkehr	Kreuzungen	Straßen
Netzwerke	Computer	Glasfaserkabel
World Wide Web	Webseiten	Hyperlinks
Sozialer Bereich	Personen	Beziehungen
Nahrungsnetz	Spezies	Räuber-Beute-Beziehung
Software	Funktionen	Funktionsaufrufe
Scheduling	Aufgaben	Ablaufeinschränkungen
elektronische Schaltungen	Gatter	Leitungen

World Wide Web

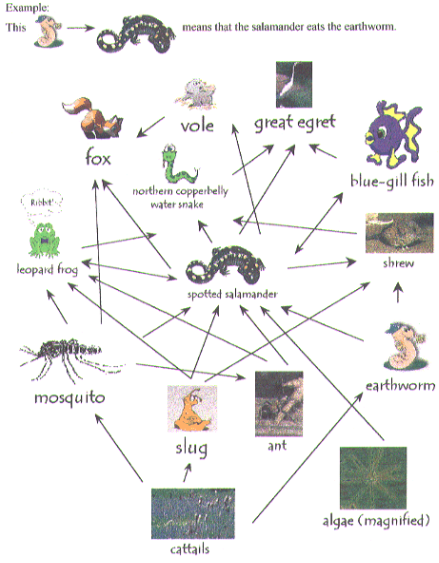
Web Graph:

- Knoten: Webseiten.
- Kante: Hyperlink von einer Seite zur anderen.



Ökologisches Nahrungsnetz

Nahrungsnetz als Graph: Knoten = Spezies, Kante = von der Beute zum Raubtier.

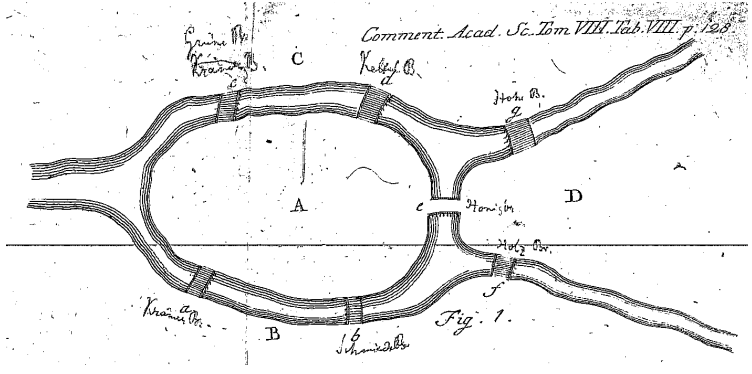


Königsberger Brückenproblem [Euler 1736]

SOLVTIO PROBLEMATIS AD GEOMETRIAM SITVS PERTINENTIS.

AVCTORE

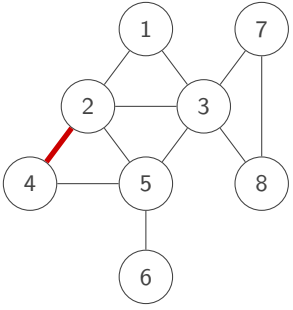
Leonh. Eulero.



Repräsentation von Graphen: Adjazenzmatrix

Adjazenzmatrix: n-mal-n Matrix mit $A_{uv} = 1$ wenn (u, v) eine Kante ist.

- Knoten: 1,2,. . . ,n.
- Zwei Einträge für jede ungerichtete Kante.
- Für gewichtete Graphen: Reelle Matrix statt Boolesche Matrix.
- Platzbedarf in $\Theta(n^2)$.
- Überprüfen, ob (u, v) eine Kante ist, hat Laufzeit $\Theta(1)$.
- Aufzählen aller Kanten hat eine Laufzeit von $\Theta(n^2)$.

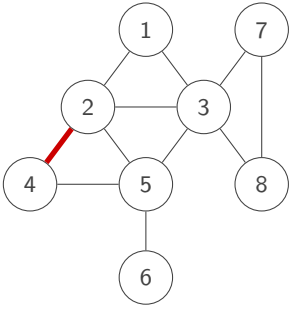


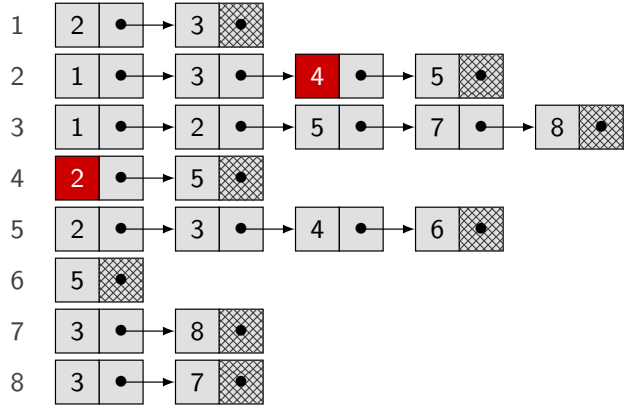
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	**1**	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	**1**	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Repräsentation von Graphen: Adjazenzlisten

Adjazenzlisten: Array von Listen. Index ist die Knotennummer.

- Knoten: 1,2,. . . ,n.
- Zwei Einträge für jede Kante.
- Für gewichtete Graphen: Speichere Gewicht in Liste.
- Platzbedarf in $\Theta(m+n)$.
- Überprüfen, ob (u,v) eine Kante ist, hat eine Laufzeit von $O(\deg(u))$.
- Aufzählen aller Kanten hat eine Laufzeit von $\Theta(m+n)$.





Adjazenzmatrix oder Adjazenzlisten

Kantenanzahl:

- Ein Graph kann bis zu $m = \frac{n(n-1)}{2} = \binom{n}{2} = \Theta(n^2)$ viele Kanten enthalten.
- Graphen sind dicht (dense) falls $m = \Theta(n^2)$.
- Graphen sind licht (sparse) falls $m = O(n)$.
- Für dichte Graphen sind beide Darstellungsformen (Adjazenzmatrix oder Adjazenzlisten) vergleichbar.

Praxis:

- Graphen, die sich aus Anwendungen ergeben, enthalten aber oft erheblich weniger Kanten.
- Typischerweise gilt dann $m = O(n)$.
- In diesem Fall ist die Darstellung mittels Adjazenzlisten günstiger.

Hinweis: Wenn wir sagen, dass ein Algorithmus auf Graphen in Linearzeit läuft, gehen wir von einer Darstellung mit Adjazenzlisten aus und betrachten eine Laufzeit von $O(n + m)$.

Ungerichtete Graphen: Kanten ausgeben

Adjazenzmatrix: Adjazenzmatrix M gegeben, n Knoten nummeriert von 0 bis $n-1$

```
for u ← 0 bis n − 2
    for v ← u + 1 bis n − 1
        if M[u, v] = 1
            Gib Kante (u, v) aus
```

Adjazenzliste: n Knoten nummeriert von 0 bis $n-1$, jeder Knoten besitzt Liste der adjazenten Knoten

```
for u ← 0 bis n − 1
    foreach Kante (u, v) inzident zu u
        if u < v
            Gib Kante (u, v) aus
```

Gerichtete Graphen: Kanten ausgeben

Adjazenzmatrix: Adjazenzmatrix M gegeben, n Knoten nummeriert von 0 bis $n-1$

```
for u ← 0 bis n − 1
    for v ← 0 bis n − 1
        if M[u, v] = 1
            Gib Kante (u, v) aus
```

Adjazenzliste: n Knoten nummeriert von 0 bis $n-1$, jeder Knoten besitzt Liste der adjazenten Knoten

```
for u ← 0 bis n − 1
    foreach Kante (u, v) inzident zu u
        Gib Kante (u, v) aus
```

Kantenzüge und Pfade

Definition: Ein Kantenzug (eng: non-simple path) in einem ungerichteten Graphen $G = (V, E)$ ist eine Folge von Knoten $v_1, v_2, \ldots, v_{k-1}, v_k$, $k \geq 1$, mit der Eigenschaft, dass jedes aufeinanderfolgende Paar v_i, v_{i+1} durch eine Kante in E verbunden ist. Die Länge des Kantenzugs ist $k - 1$.

Definition: Ein Pfad oder Weg (eng: simple path) in einem ungerichteten Graphen $G = (V, E)$ ist ein Kantenzug $v_1, v_2, \ldots, v_k$ bei dem sich kein Knoten wiederholt, also bei dem $v_i \neq v_j$ für alle $1 \leq i, j \leq k$ gilt.

Hinweis: Wir sagen auch: Der Pfad geht von v_1 nach v_k und wir bezeichnen den Pfad als v_1-v_k-Pfad.

Achtung: Die Begriffe Pfad, Weg und Kantenzug werden in der Literatur nicht einheitlich verwendet.

Zusammenhang und Distanz

Definition: Knoten u ist von Knoten v in einem Graph G erreichbar, falls G einen u-v-Pfad enthält.

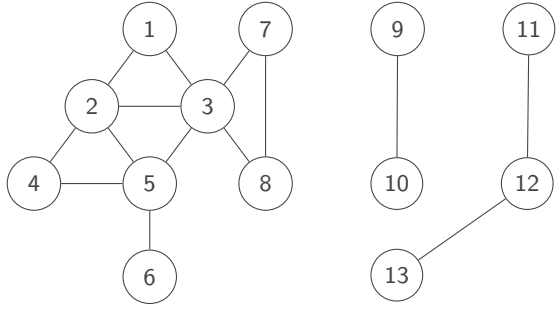
Definition: Ein ungerichteter Graph ist zusammenhängend, wenn jedes Paar von Knoten u und v von einander erreichbar ist.

Definition: Die Distanz zwischen Knoten u und v in einem ungerichteten Graphen ist die Länge eines kürzesten u-v-Pfades.

Hinweis: Falls u von v nicht erreichbar ist, nehmen wir die Distanz als ∞ an.

Zusammenhang: Beispiel

Nicht zusammenhängender Graph:

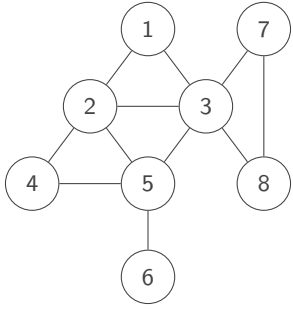


Nicht zusammenhängend: Es gibt zum Beispiel keinen Pfad vom Knoten 1 zu Knoten 10.

Beispiel für Zusammenhang: Die Knoten 1 bis 8 und ihre inzidenten Kanten bilden einen zusammenhängenden Graphen.

Kreis

Definition: Ein Kreis (eng: simple cycle) ist ein Kantenzug $v_1, v_2, \ldots, v_k$ in dem $v_1 = v_k$, $k \geq 4$, und die ersten $k-1$ Knoten alle unterschiedlich sind. Die Länge des Kreises ist $k-1$.



Beispiel für Kreis: $C =$ 1,2,4,5,3,1

Pfade und Kreise in gerichteten Graphen

Pfad: Ein Kantenzug in einem gerichteten Graphen $G = (V, E)$ ist eine Folge von Knoten $v_1, v_2, \ldots, v_{k-1}, v_k$, $k \geq 1$, mit der Eigenschaft, dass jedes aufeinanderfolgende Paar v_i, v_{i+1} durch eine gerichtete Kante (v_i, v_{i+1}) in E verbunden ist. Ein Pfad ist ein Kantenzug bei dem alle Knoten unterschiedlich sind.

Hierbei gilt:

- Der Pfad geht von einem Startknoten u zu einen Endknoten v (u-v-Pfad). Die Umkehrung muss aber nicht gelten.
- v kann von u aus erreicht werden, falls ein u-v-Pfad existiert.
- Kürzeste u-v-Kantenzüge sind Pfade.

Kreis: Ein gerichteter Kreis ist ein Kantenzug $v_1, v_2, \ldots, v_{k-1}, v_k$ in dem $v_1 = v_k$, $k \geq 3$, und die ersten $k - 1$ Knoten alle unterschiedlich sind.

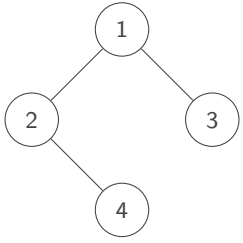
Pfade und Kreise als Graphen

Falls ein Graph G aus nur einem Pfad oder nur einem Kreis besteht, so nennen wir den ganzen Graphen einen Pfad/Kreis. Formal sagen wir:

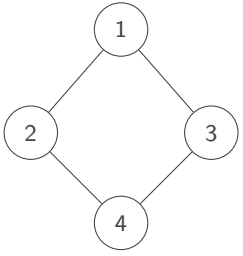
Pfad: Ein Graph G ist ein Pfad, falls es eine Aufzählung $v_1, v_2, \ldots, v_k$ der Knoten von G gibt, so dass es in G genau dann eine Kante zwischen zwei Knoten v_i und v_j gibt, falls $j = i + 1$.

Kreis: Ein Graph G ist ein Kreis, falls es eine Aufzählung $v_1, v_2, \ldots, v_k$ der Knoten von G gibt, so dass es in G genau dann eine Kante zwischen zwei Knoten v_i und v_j gibt, falls entweder $j = i + 1$ oder $i = 1$ und $j = k$ gilt.

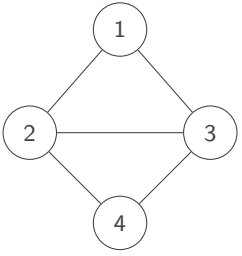
Pfade und Kreise als Graphen: Beispiele



Pfad



Kreis



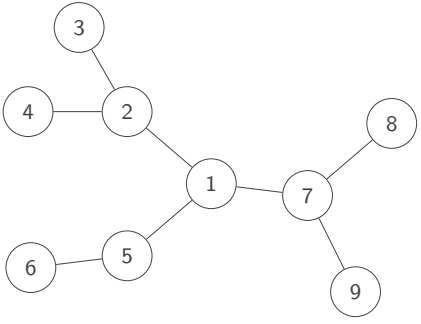
Weder Kreis noch Pfad

Bäume

Definition: Ein ungerichteter Graph ist ein Baum, wenn er zusammenhängend ist und keinen Kreis enthält.

Theorem: Sei G ein ungerichteter Graph mit n Knoten. Jeweils zwei der nachfolgenden Aussagen implizieren die dritte Aussage:

- G ist zusammenhängend.
- G enthält keinen Kreis.
- G hat $n - 1$ Kanten.



Bäume

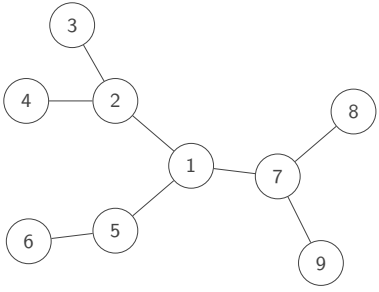
Theorem: Sei G ein ungerichteter Graph. G ist ein Baum genau dann wenn es für jedes Paar von Knoten u und v genau eine Pfad von u nach v gibt.

Beweis: G ist zusammenhängenden genau dann wenn es für jedes Paar von Knoten u und v mindestens einen Pfad von u nach v gibt. G enthält keinen Kreis, genau dann wenn es für jedes Paar von Knoten u und v maximal einen Pfad von u nach v gibt.

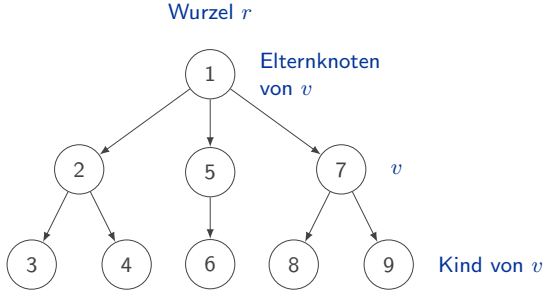
Wurzelbaum (*rooted tree, arborescence*)

Wurzelbaum: Gegeben sei ein Baum T. Wähle einen Wurzelknoten r und gib jeder Kante eine Richtung von r weg.

Bedeutung: Modelliert hierarchische Strukturen.



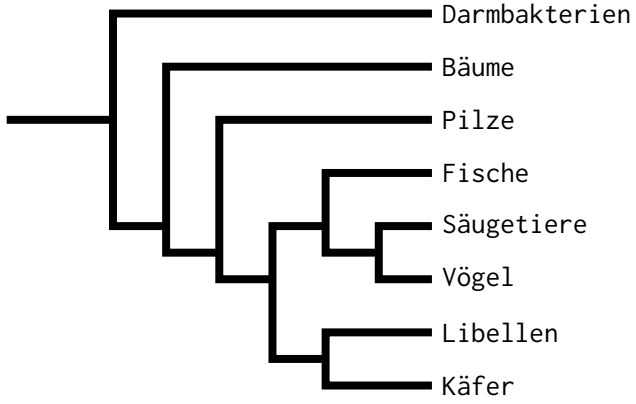
ein Baum



Ein entsprechender Wurzelbaum mit Wurzelknoten 1

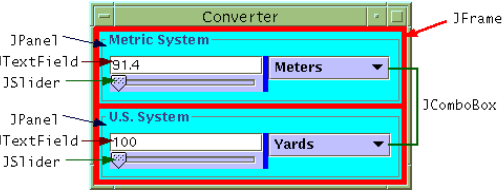
Phylogenetischer Baum

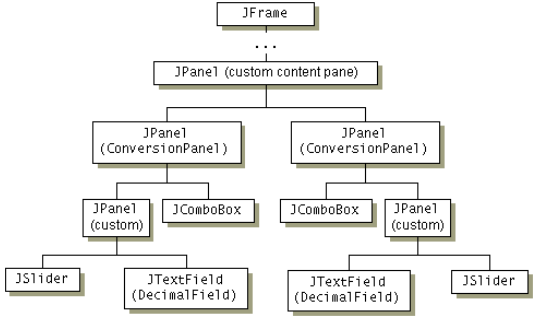
Phylogenetischer Baum: Beschreibt die evolutionären Beziehungen zwischen verschiedenen Arten.



GUI-Hierarchien

GUI-Hierarchien: Beschreiben die Organisation von GUI-Komponenten.

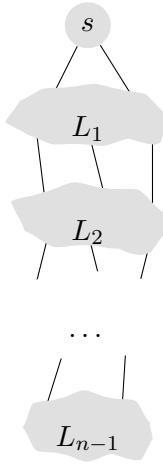




Durchmusterung von Graphen (*Graph Traversal*)

Breitensuche (*Breadth First Search, BFS*)

BFS Ansatz: Untersuche alle Knoten von einem Startknoten s ausgehend in alle möglichen Richtungen, wobei die Knoten Ebene für Ebene abgearbeitet werden.



BFS Algorithmus:

- $L_0 = \{s\}$.
- $L_1 =$ alle Nachbarn von L_0.
- $L_2 =$ alle Knoten, die nicht zu L_0 oder zu L_1 gehören und die über eine Kante mit einem Knoten in L_1 verbunden sind.
- $L_{i+1} =$ alle Knoten, die nicht zu einer vorherigen Ebene gehören und die über eine Kante mit einem Knoten in L_i verbunden sind.

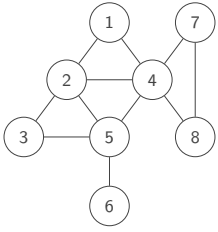
Anwendung der Breitensuche

s-t Zusammenhangsproblem: Existiert zwischen zwei gegebenen Knoten s und t ein Pfad?

s-t kürzester Pfad: Wie viele Kanten hat ein kürzester Pfad zwischen s und t ($=$ Distanz zwischen s und t)?

Anwendungen:

- Facebook.
- Labyrinth durchschreiten.
- Kevin-Bacon-Zahl.
- Die kleinste Anzahl an Hops (kürzester Pfad) zwischen zwei Knoten in einem Kommunikationsnetzwerk.



Breitensuche: Theorem

Theorem: Für jede Ebene $i = 0, 1, \ldots$ gilt, dass L_i alle Knoten mit Distanz i von s beinhaltet.

Beweis: Angenommen, sei $v_0, v_1, v_2, \ldots, v_n$ ein kürzester Pfad zwischen v_0 und v_n.

- v_0 liegt in L_0.
- v_1 liegt in L_1, da v_1 ein Nachbar von v_0 ist.
- v_2 liegt in L_2, da v_2 ein Nachbar von v_1 ist und kein Nachbar von v_0 sein kann, da es ansonsten einen kürzeren Pfad zwischen v_0 und v_n geben würde.
- Für alle weiteren Knoten gilt die gleiche Argumentation, d.h. v_n liegt schließlich in L_n. $\square$

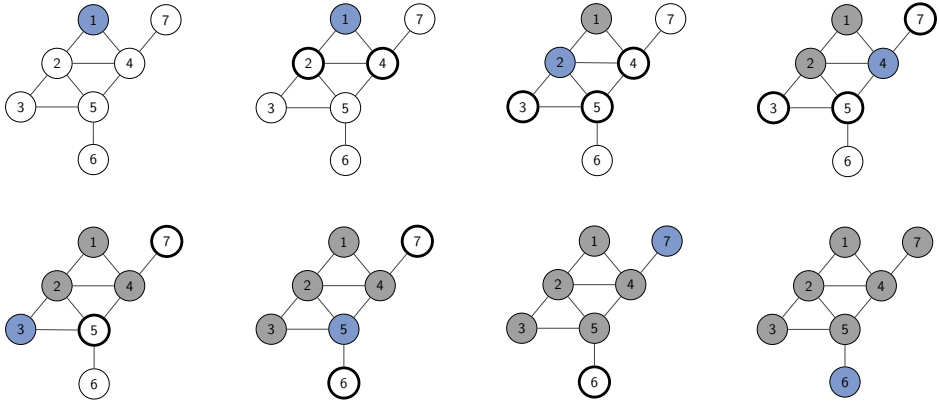
Breitensuche: Implementierung mit einer Queue

Implementierung: Array Discovered, Queue Q, Graph $G = (V, E)$, Startknoten s.

```
BFS(G,s):
Discovered[s] ← true
Discovered[v] ← false für alle anderen Knoten v ∈ V
Q ← {s}
while Q ist nicht leer
    Entferne ersten Knoten u aus Q
    Führe Operation auf u aus (z.B. Ausgabe)
    foreach Kante (u,v) inzident zu u
        if !Discovered[v]
            Discovered[v] ← true
            Füge v zu Q hinzu
```

Breitensuche: Beispiel

Möglicher Ablauf: Startknoten = 1, bearbeitete Knoten sind grau, aktiver Knoten ist blau, alle anderen Knoten sind weiß, Knoten in Queue sind mit dicken Rahmen gekennzeichnet.



Breitensuche: Analyse

Theorem: BFS hat eine Laufzeit von $O(m + n)$.

Laufzeit: Für die Laufzeitabschätzung müssen wir drei Teile betrachten:

- Initialisierung vor der while-Schleife
- while-Schleife
- foreach-Schleife

Breitensuche: Analyse

Initialisierung vor der while-Schleife:

- Jeder Knoten wird genau einmal betrachtet
- Pro Knoten können die Anweisungen in konstanter Zeit ausgeführt werden.
- Daher benötigt die Initialisierung $O(n)$ Zeit.

while-Schleife:

- Jeder Knoten u wird höchstens einmal in Q gegeben, denn nachdem er das erste mal in Q gegeben wird, wird ja Discovered[u]=true gesetzt.
- Daher wird die while-Schleife für jeden Knoten höchstens einmal durchlaufen.

Breitensuche: Analyse

foreach-Schleife:

- Sei u der gerade aktuelle Knoten bevor die foreach-Schleife ausgeführt wird.
- Dann werden in der foreach-Schleife alle Knoten v in der Adjazenzliste von u betrachtet.
- Das sind genau $deg(u)$ viele. Daher wird die Schleife $deg(u)$ mal durchlaufen. Die einzelnen Anweisungen in der Schleife benötigen konstante Zeit.

Gesamt:

- Insgesamt beträgt die Laufzeit also $O(n + \sum_{u \in V} deg(u))$.
- Da $\sum_{u \in V} deg(u) = 2m$, liegt die Laufzeit in $O(n + m)$.

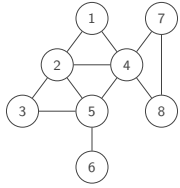
BFS-Baum

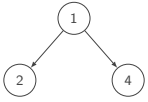
BFS-Baum: Breitensuche erzeugt einen Baum (BFS-Baum), dessen Wurzel ein Startknoten s ist und der alle von s erreichbaren Knoten beinhaltet.

Aufbau: Man startet bei s. Wird nun ein Knoten v in der Ebene L_j gefunden, ist er zu mindestens einem Knoten u der Ebene L_{j-1} benachbart. Der Knoten u von dem aus v gefunden wurde wird ausgewählt und zum Elternknoten von v im BFS-Baum gemacht.

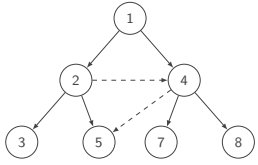
BFS-Baum: Eigenschaft

Eigenschaft: Sei T ein BFS-Baum von $G = (V, E)$ und sei (x, y) eine Kante von G. Dann können sich die Ebenen von x und y höchstens um 1 unterscheiden.

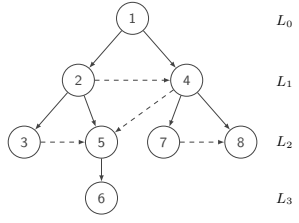




(a)



(b)



(c)

Breitensuche: Ermitteln der Ebenen

Anwendung von BFS: Ermitteln der Ebene jedes einzelnen Knotens.

Implementierung: Array Level, Queue Q, Graph $G = (V, E)$, Startknoten s.

```
BFS(G,s):
Level[s] ← 0
Level[v] ← -1 für alle anderen Knoten v ∈ V
Q ← s
while Q ist nicht leer
    Entferne ersten Knoten u aus Q
    foreach Kante (u,v) inzident zu u
        if  Level[v] == -1
            Level[v] ← Level[u] + 1
            Füge v zu Q hinzu
```

Tiefensuche (*Depth First Search, DFS*)

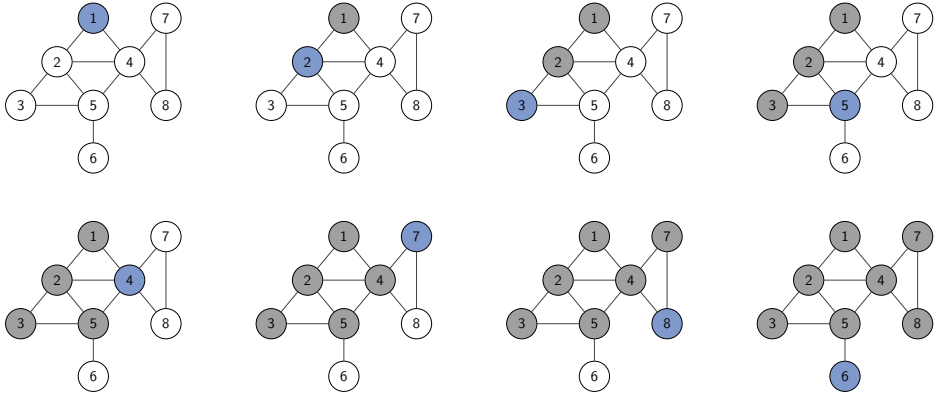
DFS Ansatz: Von einem besuchten Knoten u wird zuerst immer zu einem weiteren noch nicht besuchten Nachbarknoten gegangen (DFS-Aufruf), bevor die weiteren Nachbarknoten von u besucht werden.
DFS Algorithmus: Startknoten s, globales Array Discovered, Graph $G = (V, E)$.

```
DFS(G,s):
Discovered[v] ← false für alle Knoten v ∈ V
DFS1(G,s)

DFS1(G,u):
Discovered[u] ← true
Führe Operation auf u aus (z.B. Ausgabe)
foreach Kante (u,v) inzident zu u
    if  !Discovered[v]
        DFS1(G,v)
```

Tiefensuche: Beispiel

Möglicher Ablauf: Startknoten = 1, bearbeitete Knoten sind grau unterlegt, aktiver Knoten ist blau, alle anderen Knoten sind weiß.



Tiefensuche: Analyse

Theorem: DFS hat eine Laufzeit von $O(m+n)$.

Laufzeit: Für Laufzeitabschätzung betrachten wir:

- Initialisierung
- foreach-Schleife

Initialisierung:

- Initialisierung vor dem Aufruf von DFS1 in $O(n)$ Zeit.
- DFS1(G,u) wird für jeden Knoten u höchstens einmal aufgerufen.

Tiefensuche: Analyse

foreach-Schleife in DFS1(G,u):

- Es werden alle Knoten v in der Adjazenzliste von u betrachtet. Das sind genau $deg(u)$ viele.
- Daher wird die Schleife $deg(u)$ mal durchlaufen.
- Die einzelnen Anweisungen in der Schleife benötigen konstante Zeit (außer dem rekursiven Aufruf DFS1(G,v), aber dessen Laufzeit wird ja in der Analyse für den Knoten v berücksichtigt).

Gesamt:

- Insgesamt beträgt die Laufzeit also $O(n + \sum_{u \in V} deg(u))$.
- Da $\sum_{u \in V} deg(u) = 2m$, erhalten wir eine Laufzeit von $O(n + m)$.

Tiefensuche: Durchmusterung

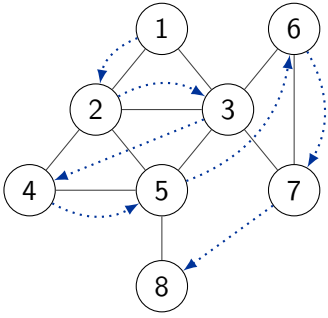
Durchmusterung: Durchmusterung bei DFS unterscheidet sich von der bei BFS.

- Es wird zunächst versucht, möglichst weit vom Startknoten weg zu kommen.
- Gibt es in der Nachbarschaft keine möglichen Knoten, dann wird durch den rekursiven Aufstieg bis zu einer möglichen Verzweigung zurückgegangen (Backtracking).

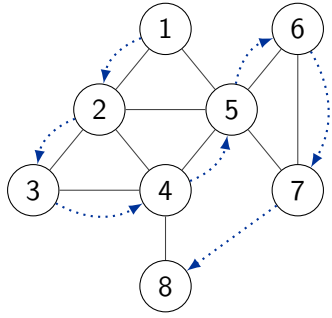
Beispiel

Vergleich: Tiefensuche und Breitensuche im Vergleich.

Breitensuche:



Tiefensuche:



Zusammenhangskomponente

Zusammenhang (Wiederholung): Ein ungerichteter Graph ist zusammenhängend, wenn für jedes Paar von Knoten u und v ein Pfad zwischen u und v existiert.

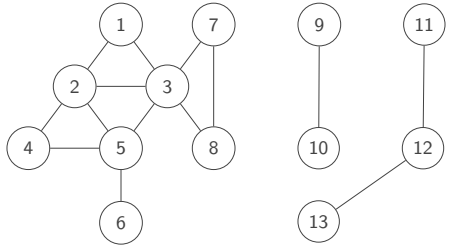
Nicht zusammenhängend: Gibt es zwischen einem Paar von Knoten keinen Pfad, dann ist der Graph nicht zusammenhängend.

Teilgraph: Ein Graph $G_1 = (V_1, E_1)$ heißt Teilgraph von $G_2 = (V_2, E_2)$, wenn seine Knotenmenge V_1 Teilmenge von V_2 und seine Kantenmenge E_1 Teilmenge von E_2 ist, also $V_1 \subseteq V_2$ und $E_1 \subseteq E_2$ gilt.

Zusammenhangskomponente: Einen maximalen zusammenhängenden Teilgraphen eines beliebigen Graphen nennt man Zusammenhangskomponente. Ein nicht zusammenhängender Graph zerfällt in seine Zusammenhangskomponenten.

Zusammenhangskomponente

Beispiel: Ein nicht zusammenhängender Graph mit 3 Zusammenhangskomponenten.



Zusammenhangskomponente

Zusammenhangskomponente: Finde alle Knoten, die von s aus erreicht werden können.

Lösung:

- Rufe DFS(G,s) oder BFS(G,s) auf.
- Ein Knoten u ist von s genau dann erreichbar, wenn Discovered[u]=true ist.

Zusammenhangskomponenten zählen

DFSNUM Algorithmus: Startknoten s, globales Array Discovered, Graph $G = (V, E)$.

```
DFSNUM(G):
Discovered[v] ← false für alle Knoten v ∈ V
i ← 0
foreach Knoten v ∈ V
    if Discovered[v] = false
        i ← i + 1
        DFS1(G,v)
return i
```

Zusammenhangskomponenten zählen

Laufzeit: Die Laufzeit liegt in $O(n+m)$.

Analyse:

- Sei $G=(V,E)$ der gegebene Graph und $G_1=(V_1,E_1),\ldots,G_r=(V_r,E_r)$ seine Zusammenhangskomponenten. Sei $|V|=n$ und $|E|=m$, sowie $|V_i|=n_i$ und $|E_i|=m_i$, für $1\leq i\leq r$.
- Klarerweise gilt $n=n_1+\ldots+n_r$ und $m=m_1+\ldots+m_r$.
- Für jede einzelne Zusammenhangskomponente G_i $(1\leq i\leq r)$ führt der Algorithmus eine Tiefensuche aus. Dies hat eine Laufzeit von $O(n_i+m_i)$.
- Die Initialisierung benötigt $O(n)$ Zeit.
- Insgesamt erhalten wir eine Laufzeit von $O(n+\sum_{i=1}^{r}(n_i+m_i))=O(2n+m)=O(n+m)$.

Zusammenhang in gerichteten Graphen

Suche in gerichteten Graphen

Gerichtete Erreichbarkeit: Gegeben sei ein Knoten s, finde alle Knoten, die von s aus erreicht werden können.

Gerichteter kürzester s-t Pfad: Gegeben seien zwei Knoten s und t, ermittle einen kürzesten Pfad von s nach t.

Suche in gerichteten Graphen: BFS und DFS können auch auf gerichtete Graphen angewendet werden.

Beispiel Webcrawler: Starte von einer Webseite s. Finde alle Webseiten, die von s aus direkt oder indirekt verlinkt sind.

Starker Zusammenhang

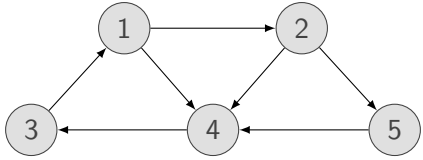
Definition: Knoten u und v in einem gerichteten Graphen sind gegenseitig erreichbar, wenn es einen Pfad von u zu v und einen Pfad von v zu u gibt.

Definition: Ein gerichteter Graph ist stark zusammenhängend, wenn jedes Paar von Knoten gegenseitig erreichbar ist.

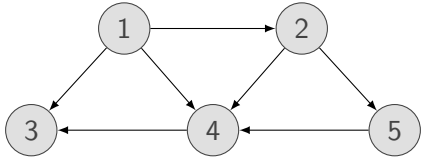
Hinweis: Ein gerichteter Graph heißt schwach zusammenhängend, falls der zugehörige ungerichtete Graph (also der Graph, der entsteht, wenn man jede gerichtete Kante durch eine ungerichtete Kante ersetzt) zusammenhängend ist.

Starker Zusammenhang: Beispiel

Stark zusammenhängend:



Nicht stark zusammenhängend (aber schwach zusammenhängend): Knoten 1 kann von keinem anderen Knoten erreicht werden, vom Knoten 3 führt kein Pfad weg.

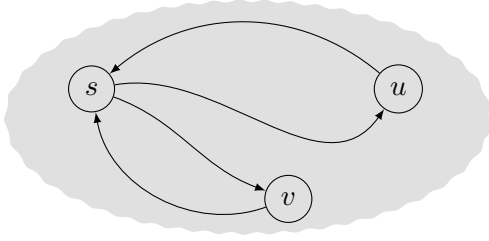


Starker Zusammenhang

Lemma: Sei s ein beliebiger Knoten in einem gerichteten Graphen G. G ist stark zusammenhängend dann und nur dann, wenn jeder Knoten von s aus und s von jedem Knoten aus erreicht werden kann.

Beweis: $\Rightarrow$ Folgt aus der Definition.
Beweis: $\Leftarrow$ Pfad von u zu v: verbinde u-s Pfad mit s-v Pfad. Pfad von v zu u: verbinde v-s Pfad mit s-u Pfad. □

- *auch ok, wenn Pfade überlappen*



Starker Zusammenhang: Algorithmus

Theorem: Laufzeit für die Überprüfung, ob G stark zusammenhängend ist, liegt in $O(m + n)$.

Beweis:

- Wähle einen beliebigen Knoten s.
- Führe BFS mit Startknoten s in G aus.
- Führe BFS mit Startknoten s in G^{rev} aus.
- Gib true zurück dann und nur dann, wenn alle Knoten in beiden BFS-Ausführungen erreicht werden können.
- Korrektheit folgt unmittelbar aus dem vorherigen Lemma. □

□ *umgekehrte Orientierung von jeder Kante in* G

DAGs und Topologische Sortierung

Gerichteter azyklischer Graph (*Directed Acyclic Graph, DAG*)

Definition: Ein DAG ist ein gerichteter Graph, der keine gerichteten Kreise enthält.

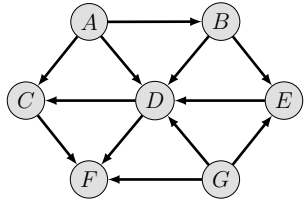
Beispiel: Knoten: Aufgaben, Kanten: Reihenfolgebeschränkungen
Kante (u, v) bedeutet, Aufgabe u muss vor Aufgabe v erledigt werden.

Definition: Wir nennen eine Knoten v ohne eingehende Kanten in einem gerichteten Graphen (i.e., $deg^-(v) = 0$) Quelle.

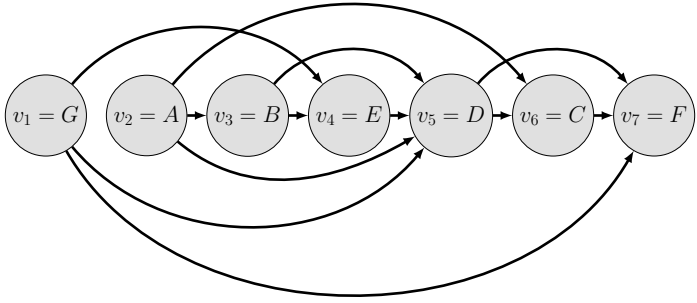
Definition: Eine topologische Sortierung eines gerichteten Graphen $G = (V, E)$ ist eine lineare Ordnung seiner Knoten, bezeichnet mit $v_1, v_2, \ldots, v_n$, sodass für jede Kante (v_i, v_j) gilt, dass $i < j$.

Topologische Sortierung: Beispiel

Ein DAG:



Eine topologische Sortierung:



Reihenfolgebeschränkung

Reihenfolgebeschränkung: Kante (u, v) bedeutet, dass Aufgabe u vor v bearbeitet werden muss.

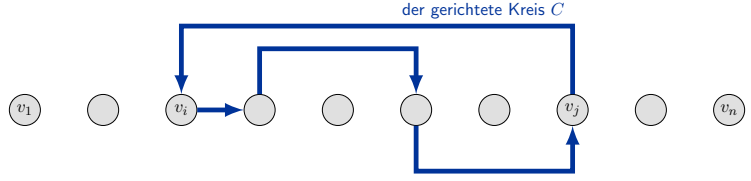
Anwendungen:

- Voraussetzungen bei Kursen: Kurs u muss vor Kurs v absolviert werden.
- Übersetzung: Modul u muss vor Modul v übersetzt werden.
- Pipeline von Prozessen: Ausgabe von Prozess u wird benötigt, um die Eingabe von v zu bestimmen.

Gerichteter azyklischer Graph

Lemma: Wenn G eine topologische Sortierung hat, dann ist G ein DAG.
Beweis: (durch Widerspruch)

- Wir nehmen an, dass G eine topologische Sortierung $v_1, \ldots, v_n$ und auch einen gerichteten Kreis C besitzt.
- Sei v_i der Knoten mit dem kleinsten Index in C und sei v_j der Knoten direkt vor v_i in C; daher gibt es die Kante (v_j, v_i).
- Durch die Wahl von i gilt, dass $i < j$.
- Andererseits, da (v_j, v_i) eine Kante ist und $v_1, \ldots, v_n$ eine topologische Sortierung ist, müsste eigentlich $j < i$ sein. Widerspruch. □



die angenommene topologische Sortierung: $v_1, \ldots, v_n$

Gerichteter azyklischer Graph

Lemma: Wenn G eine topologische Sortierung hat, dann ist G ein DAG.

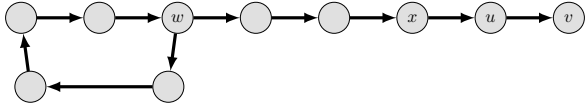
Frage: Hat jeder DAG eine topologische Sortierung?

Frage: Wenn ja, wie berechnen wir diese?

Gerichteter azyklischer Graph

Lemma: Wenn G ein DAG ist, dann hat G eine Quelle.
Beweis: (durch Widerspruch)

- Wir nehmen an, G ist ein DAG ohne Quelle.
- Wähle einen beliebigen Knoten v und folge den Kanten von v aus rückwärts. Da v zumindest eine eingehende Kante (u, v) besitzt, können wir rückwärts zu u gelangen.
- Da u zumindest eine eingehende Kante (x, u) hat, können wir rückwärts zu x gelangen.
- Das wird so oft wiederholt, bis man einen Knoten w zweimal besucht.
- Sei C die Sequenz von Knoten die zwischen zwei Besuchen von w durchlaufen wurde. C ist ein Kreis. □



Gerichteter azyklischer Graph

Lemma: G ist ein DAG genau dann wenn jeder Teilgraph von G eine Quelle hat.
Beweis:

- Angenommen G ist ein DAG, dann ist offensichtlich auch jeder Teilgraph von G ein DAG (Das Entfernen von Knoten kann keine Kreise produzieren). Deswegen hat jeder Teilgraph von G eine Quelle.
- Angenommen G ist kein DAG. Dann enthält G einen Kreis als Teilgraph. Ein Kreis hat keine Quelle.

Gerichteter azyklischer Graph - Erkennen eines DAG mittels wiederholtem Löschen von Kanten

```
while G hat mindestens einen Knoten
    if G hat eine Quelle
        Wähle eine Quelle v aus
        Gib v aus
        Lösche v und alle inzidenten Kanten aus G
    else return G ist kein DAG
return G ist ein DAG
```

Hinweis:

- Ein Knoten kann im Lauf des Algorithmus zur Quelle werden.
- Falls G ein DAG ist, gibt dieser Algorithmus eine topologische Sortierung aus.

Gerichteter azyklischer Graph

Lemma: Wenn G ein DAG ist, dann hat G eine topologische Sortierung.

Beweis:

- Falls G ein DAG ist, können wir eine topologische Sortierung berechnen.
- Falls G kein DAG ist, enthält G eine Kreis $v_1, \ldots v_n$. In der Ordnung einer topologischen Sortierung müsste dann $v_1 < \ldots v_n < v_1$ gelten. Dann ist die Ordnung allerdings keine lineare Ordnung.

Topologische Sortierung

Algorithmus: Effiziente Implementierung des Löschalgorithmus: Löschen von Knoten wird mittels Hilfsarray count simuliert. Es wird zusätzlich eine anfangs leere Liste L verwendet.

```
foreach v ∈ V
    count[v] ← 0
foreach v ∈ V
    foreach Kante (v, w) ∈ E
        count[w] ← count[w]+1
foreach v ∈ V
    if count[v] = 0
        Gib v zur Liste L am Anfang hinzu
while L ist nicht leer
    Sei v erstes Element in L, lösche v aus L
    Gib v aus
    foreach Kante (v, w) ∈ E
        count[w] ← count[w]−1
        if count[w] = 0
            Gib w zur Liste L am Anfang hinzu
```

Topologische Sortierung: Laufzeit

Theorem: Algorithmus findet eine topologische Sortierung in $O(n+m)$ Zeit.

Laufzeit: Dazu betrachten wir die folgenden Teile:

- Initialisierung
  - Erste foreach-Schleife für count.
  - Zwei verschachtelte foreach-Schleifen.
  - Dritte foreach-Schleife für Generierung der Liste.
- while-Schleife (mit foreach-Schleife).

Initialisierung:

- Die erste foreach-Schleife für die Initialisierung von count benötigt $O(n)$ Zeit.
- Bei den verschachtelten foreach-Schleifen wird die innere foreach-Schleife für jeden Knoten v genau $deg^+(v)$ mal ausgeführt. Daher benötigt man dafür $O(n+m)$ Zeit.
- Die Generierung der Liste L durch die dritte foreach-Schleife benötigt $O(n)$ Zeit.
- Daher benötigt die Initialisierung $O(n+m)$ Zeit.

Topologische Sortierung: Analyse

while-Schleife:

- Jeder Knoten v wird höchstens einmal aus L entnommen.
- Daher wird die while-Schleife für jeden Knoten höchstens einmal durchlaufen.

foreach-Schleife:

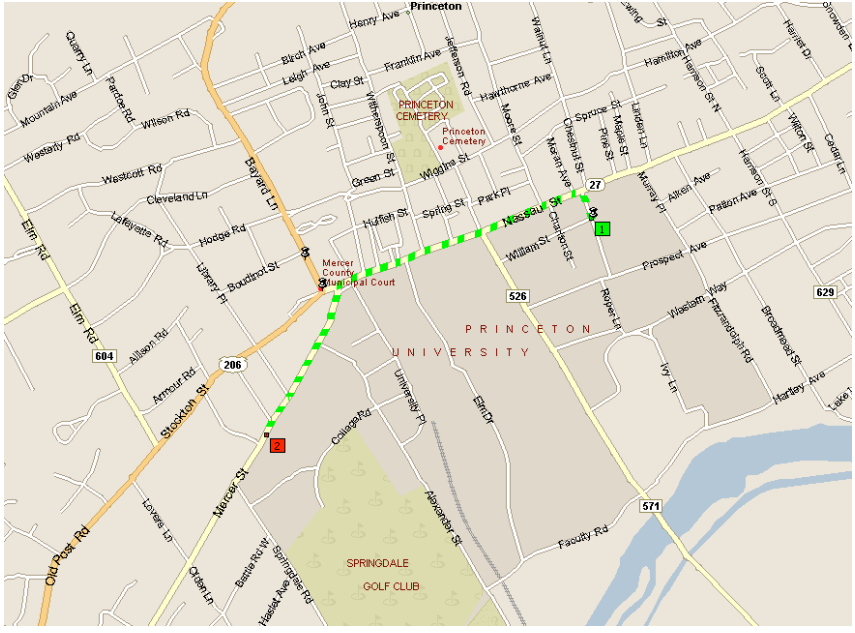
- Sei v der gerade aktuelle Knoten bevor die foreach-Schleife ausgeführt wird.
- Dann werden in der foreach-Schleife alle Knoten w in der Adjazenzliste von v betrachtet.
- Das sind genau $deg^+(v)$ viele. Daher wird die Schleife $deg^+(v)$ mal durchlaufen. Die einzelnen Anweisungen in der Schleife benötigen konstante Zeit.
- Jeder Knoten w wird höchstens einmal in L eingefügt.

Topologische Sortierung: Analyse

Gesamt:

- Initialisierung liegt in $O(n + m)$
- while-Schleife liegt in $O(n + m)$
- Daher liegt auch die gesamte Laufzeit in $O(n + m)$

Kürzeste Pfade in einem gewichteten Graphen



Kürzester Pfad vom Informatikinstitut in Princeton zu Einsteins Haus.

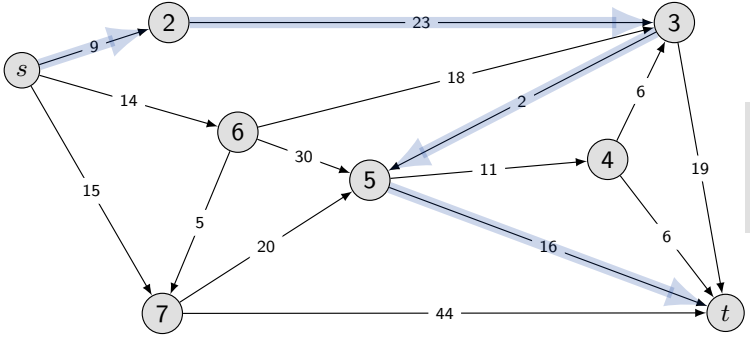
Kürzester Pfad (*Shortest Path Problem*)

Netzwerk für kürzesten Pfad:

- Gerichteter Graph $G = (V, E)$.
- Start s, Ziel t.
- Länge $\ell_e \geq 0$ ist die Länge der Kante e (Gewicht).

Kürzester Pfad: Finde kürzesten gerichteten Pfad von s nach t.

Kürzester Pfad = Pfad mit den geringsten Kosten, wobei die Kosten eines Pfades die Summe der Gewichte seiner Kanten sind.



Kosten des Pfades
s-2-3-5-t
$= 9 + 23 + 2 + 16$
$= 50.$

Dijkstra 1959

Numerische Mathematik 1, 269—271 (1959)

A Note on Two Problems in Connexion with Graphs

By

E. W. DIJKSTRA

We consider n points (nodes), some or all pairs of which are connected by a branch; the length of each branch is given. We restrict ourselves to the case where at least one path exists between any two nodes. We now consider two problems.

Algorithmus von Dijkstra

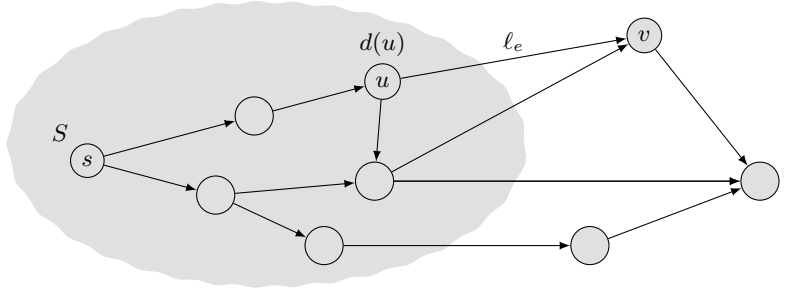
Algorithmus von Dijkstra:

- Verwalte eine Menge S von untersuchten Knoten, für die wir die Kosten $d(u)$ eines kürzeste s-u-Pfades ermittelt haben.
- Initialisiere $S = \{s\}$, $d(s) = 0$.
- Wähle wiederholt einen nicht untersuchten Knoten v, für den der folgende Wert am kleinsten ist:

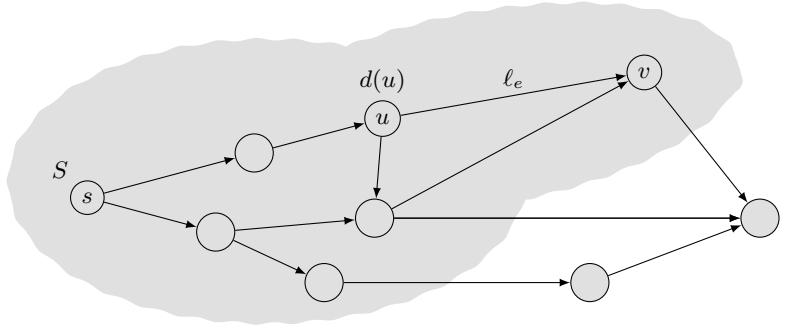
$$\min_{e=(u,v):u\in S} d(u) + \ell_e,$$

  d.h. die Länge eines kürzesten Pfades zu einem u im untersuchten Teil des Graphen, gefolgt von einer einzigen Kante (u, v).
- Füge v zu S hinzu und setze $d(v) = \min_{e=(u,v):u\in S} d(u) + \ell_e$.
- Extrahieren des Pfades entweder durch Merken des Vorgängerknotens oder mittels eines eigenen Algorithmus, der nach Dijkstra ausgeführt wird.

Algorithmus von Dijkstra: Menge S



Algorithmus von Dijkstra: Menge S



Dijkstra-Algorithmus: Implementierung

Implementierung: Wir werden zwei Implementierungen für S betrachten:

- Eine einfach verkettete Liste.
- Eine Vorrangwarteschlange (*priority queue*) von nicht untersuchten Knoten, geordnet nach den Kosten d.
  Ein Eintrag in der Queue besteht aus dem Knotenindex und den dazugehörigen Kosten.

Dijkstra-Algorithmus

Algorithmus: Arrays `Discovered` und d, Graph $G = (V, E)$, Liste L, Startknoten s.

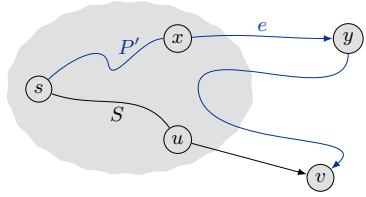
```
Dijkstra(G,s):
Discovered[v] ← false für alle Knoten v ∈ V
d[s] ← 0
d[v] ← ∞ für alle anderen Knoten v ∈ V \ {s}
L ← V
while L ist nicht leer
    wähle u ∈ L mit kleinstem Wert d[u]
    lösche u aus L
    Discovered[u] ← true
    foreach Kante e = (u,v) ∈ E
        if !Discovered[v]
            d[v] ← min(d[v], d[u]+ℓ_e)
```

Algorithmus von Dijkstra: Korrektheitsbeweis

Invariante: Für jeden Knoten $u \in S$, ist $d(u)$ die Länge eines kürzesten s-u Pfades.
Beweis: (durch Induktion nach $|S|$)
Induktionsanfang: $|S| = 1$ ist trivial.
Induktionsbehauptung: Angenommen, wahr für $|S| = k \geq 1$.

- Sei v der nächste zu S hinzugefügte Knoten und sei (u, v) die gewählte Kante.
- Ein kürzester s-u Pfad plus (u, v) ist ein s-v Pfad der Länge $d(v)$.

- Wir betrachten einen beliebigen s-v Pfad P. Wir werden zeigen, dass er nicht kürzer als $d(v)$ ist.
- Sei $e = (x, y)$ die erste Kante in P die S verlässt und sei P' der Teilpfad zu x.
- P ist schon zu lange, wenn er S verlässt.



$$\text{Länge}(P) \geq \text{Länge}(P') + \ell_e \geq d(x) + \ell_e \geq d(y) \geq d(v)$$

Nicht-negative Gewichte *Induktionsbehauptung* *Definition von* $d(y)$
Dijkstra-Algorithmus wählt v *anstatt* y

Analyse: Dijkstra-Algorithmus mit Liste

Theorem: Der Dijkstra-Algorithmus, implementiert mit einer Liste, hat eine Worst-Case-Laufzeit von $O(n^2)$.

Laufzeiten:

- Initialisierung der Arrays benötigt $O(n)$ Zeit.
- Die while-Schleife wird n-mal ausgeführt und darin muss in jeder Iteration der Knoten u mit dem kleinsten Wert für $d[u]$ gefunden werden. Das liegt in $O(n^2)$ Zeit.
- Die foreach-Schleife wird insgesamt (über alle Iterationen der while-Schleife) höchstens m-mal ausgeführt. Für jeden Knoten werden seine ausgehenden Kanten nur einmal betrachtet und insgesamt gibt es nur m Kanten.
- Daher beträgt die Laufzeit $O(n + n^2 + m)$ und somit $O(n^2)$. $\square$

Wir werden sehen, dass der Dijkstra-Algorithmus mit einer Worst-Case-Laufzeit von $O((n + m) \log n)$ implementiert werden kann. Für lichte Graphen ist das effizienter als $O(n^2)$.

Priority Queue (Vorrangwarteschlange)

Priority Queue:

- Eine Priority Queue ist eine Datenstruktur, die eine Menge S von Elementen verwaltet.
- Jedes Element $v \in S$ hat einen dazugehörigen Wert i, der die Priorität von v beschreibt.
- Kleinere Werte repräsentieren höhere Prioritäten.

Operationen: Alle mit Laufzeit in $O(\log n)$.

- Einfügen eines Elements in die Menge S.
- Löschen eines Elements aus der Menge S.
- Finden eines Elements mit dem kleinsten Wert (höchster Priorität).

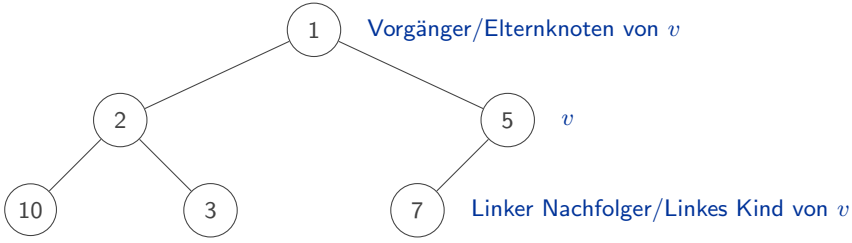
Frage: Wie erreicht man eine Laufzeit in $O(\log n)$?
Antwort: Mit einer bestimmten Datenstruktur, dem Heap.

Heap

Heap: Ein Heap (Min-Heap) ist ein binärer Wurzelbaum, dessen Knoten mit $\leq$ total geordnet sind, sodass gilt:

- Ist u ein linkes oder rechtes Kind von v, dann gilt $v \leq u$ (Heap-Eigenschaft für Min-Heap).
- Alle Ebenen von Knoten bis auf die letzte sind vollständig aufgefüllt.
- Die letzte Ebene des Baumes muss linksbündig aufgefüllt werden.

Beispiel:



Repräsentation eines Heaps

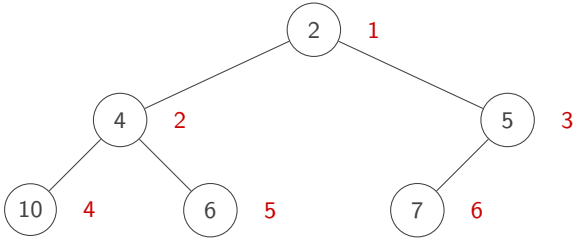
Effiziente Repräsentation: Knoten des Baums ebenenweise in einem Array speichern.

Effiziente Berechnung:

- Die beiden Nachfolgerknoten eines Knotens an der Position k befinden sich an den Positionen $2k$ und $2k+1$. Sein Elternknoten befindet sich an der Position $\lfloor \frac{k}{2} \rfloor$.
- Damit obige Rechnung immer funktioniert, wird das Array ab Index 1 belegt.
- Würde man bei Index 0 anfangen, dann würden sich die Berechnungen folgendermaßen ändern: Nachfolger links auf $2k+1$, Nachfolger rechts auf $2k+2$, Elternknoten auf $\lfloor \frac{k-1}{2} \rfloor$.

Beispiel für Heap-Repräsentation

Heap:



Array: 6 Einträge, erster Platz unbelegt (mit 0 initialisiert).

Index	0	1	2	3	4	5	6
Wert	0	2	4	5	10	6	7

Heapify-up

Einfügen eines neuen Elements: Bei einem Heap mit n Elementen wird das neue Element an Position $n+1$ eingefügt. Wir gehen dabei davon aus, dass noch genügend Plätze im Array frei sind.

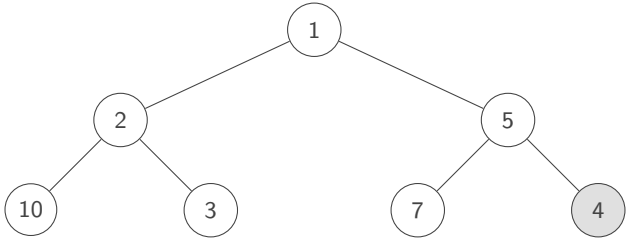
Heap-Bedingung: Die Heap-Bedingung kann durch das neue Element verletzt werden.

Reparieren: Durch Operation Heapify-up (für Heap-Array H an Position i) in $O(\log n)$ Zeit. Aufruf nach dem Einfügen des neuen Elements: Heapify-up(H,$n+1$).

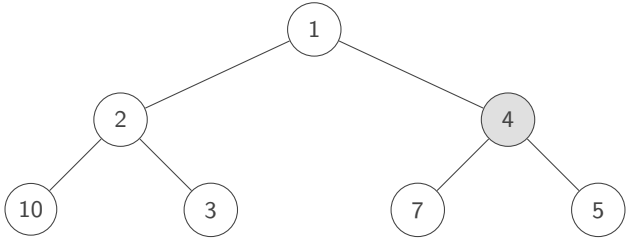
```
Heapify-up(H,i):
if i > 1
    j ← ⌊i/2⌋
    if H[i] < H[j]
        Vertausche die Array-Einträge H[i] und H[j]
        Heapify-up(H,j)
```

Beispiel für Heapify-up

Einfügen von 4:



Verschieben von 4:



Heapify-down

Löschen eines Elements: Element wird an Stelle i gelöscht. Das Element an Stelle n (bei n Elementen) wird an die freie Stelle verschoben.

Heap-Bedingung: Die Heap-Bedingung kann durch das neue Element an der Stelle i verletzt werden.

Reparieren:

- Eingefügtes Element ist zu groß: Benutze Heapify-down, um das Element auf eine untere Ebene zu bringen.
- Eingefügtes Element ist zu klein: Benutze Heapify-up (wie beim Einfügen) von der Stelle i aus.

Hinweis: Beim Heap wird typischerweise die Wurzel entfernt und daher wird dann nur Heapify-down benutzt.

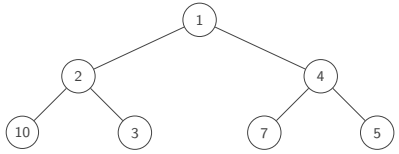
Laufzeit für Löschen: Für Heap-Array H an Position i in $O(\log n)$ Zeit.

Heapify-down

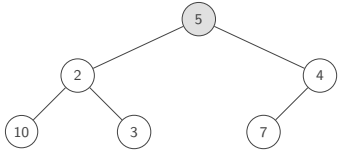
```
Heapify-down(H,i):
n ← length(H)-1
if 2 · i > n
    return
elseif 2 · i < n
    left ← 2 · i, right ← 2 · i + 1
    j ← Index des kleineren Wertes von H[left] und H[right]
else
    j ← 2 · i
if H[j] < H[i]
    Vertausche die Arrayeinträge H[i] und H[j]
    Heapify-down(H,j)
```

Beispiel für Heapify-down

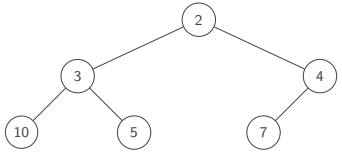
Ursprünglicher Heap:



Löschen von 1, verschieben von 5:



Heapify-down (zwei Mal)



Operationen auf Heap

Operationen auf Heap:

- Insert(H,v): Element v in den Heap H einfügen. Hat der Heap n Elemente, dann liegt die Laufzeit in $O(\log n)$.
- FindMin(H): Findet das Minimum im Heap H. Laufzeit ist konstant (da Wurzel).
- Delete(H,i): Löscht das Element im Heap H an der Stelle i. Für einen Heap mit n Elementen liegt die Laufzeit in $O(\log n)$.
- ExtractMin(H): Kombination von FindMin und Delete und daher in $O(\log n)$.

Erstellen eines Heaps

Erstellen: Das Erstellen eines Heaps aus einem Array A mit Größe n, das noch nicht die Heapeigenschaft erfüllt:

```
Init(A,n):
for i = ⌊n/2⌋ bis 1
    Heapify-down(A,i)
```

Erstellen eines Heaps: Analyse

Laufzeit: $O(n)$ ergibt sich aus folgender Berechnung:

- Einfachheitshalber nehmen wir an, der Binärbaum ist vollständig und hat n Knoten.
- Es folgt, dass $n = 2^{h+1} - 1$ wobei h die Höhe des Baumes ergibt.
- Wir lassen den Index j über die Ebenen E_j des Baumes laufen, wobei mit E_0 die Ebene mit den Blättern des Baumes bezeichnet und E_h die Ebene mit der Wurzel.
- Es folgt, dass Ebene E_j genau 2^{h-j} Knoten enthält und der Aufwand zum Einfügen eines Elements auf Ebene E_j proportional zu j ist.
- Insgesamt ergibt sich also ein Aufwand von $\sum_{j=0}^{h} j2^{h-j}$, den wir folgendermaßen abschätzen:

$$\sum_{j=0}^{h} j2^{h-j} \;=\; \sum_{j=0}^{h} j\frac{2^h}{2^j} \;=\; 2^h \sum_{j=0}^{h} \frac{j}{2^j} \;\leq\; 2^h 2 \;=\; 2^{h+1} \;=\; n+1 \;=\; O(n)$$

$\leq$ folgt aus $\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$. $=$ da $n = 2^{h+1} - 1$.

Dijkstra-Algorithmus: Effizientere Variante

Algorithmus:

- Arrays `Discovered` und d, Graph $G = (V, E)$, Startknoten s.
- Verwende Vorrangwarteschlange Q, in der die Knoten v nach dem Wert $d[v]$ geordnet sind.

```
Dijkstra(G,s):
Discovered[v] ← false für alle Knoten v ∈ V
d[s] ← 0
d[v] ← ∞ für alle anderen Knoten v ∈ V \ {s}
Q ← V
while Q ist nicht leer
    wähle u ∈ Q mit kleinstem Wert d[u]
    lösche u aus Q
    Discovered[u] ← true
    foreach Kante e = (u,v) ∈ E
        if !Discovered[v]
            if d[v] > d[u]+ℓ_e
                lösche v aus Q
                d[v] ← d[u]+ℓ_e
                füge v zu Q hinzu
```

Analyse: Dijkstra-Algorithmus mit Vorrangwarteschlange

Theorem: Der Dijkstra-Algorithmus, implementiert mit einer Vorrangwarteschlange, hat eine Worst-Case-Laufzeit von $O((n+m)\log n)$.

Laufzeiten:

- Initialisierung der Arrays benötigt $O(n)$ Zeit.
- Die while-Schleife wird n-mal ausgeführt und darin muss in jeder Iteration der Knoten u mit dem kleinsten Wert für $d[u]$ aus der Queue gelöscht werden ($O(\log n)$).
- Die foreach-Schleife liegt in $O(m \log n)$ Zeit. Für jeden Knoten werden seine ausgehenden Kanten nur einmal betrachtet und insgesamt gibt es nur m Kanten. Bei einer Neuberechnung muss aber die Queue reorganisiert werden (diese Operation liegt in $O(\log n)$).
- Daher beträgt die Laufzeit $O(n + n\log n + m\log n)$ und somit $O((n+m)\log n)$. □

Dijkstra-Algorithmus: Abschließender Vergleich

Wir vergleichen die Laufzeit des Dijkstra-Algorithmus bei Verwendung von Listen, Vorrangwarteschlange als Heap und Vorrangwarteschlange als Fibonacci-Heap (diese verbesserte Datenstruktur haben wir nicht besprochen).

Tabelle Vergleich verschiedener Datenstrukturen für Dijkstra-Algorithmus.

Liste	**Heap**	**FibHeap**
$O(n^2)$	$O((n+m)\log n)$	$O(m+n\log n)$

Graphen

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 9. März 2023
Vorlesungsfolien

Grundlegende Definitionen und Anwendungen

Graphen

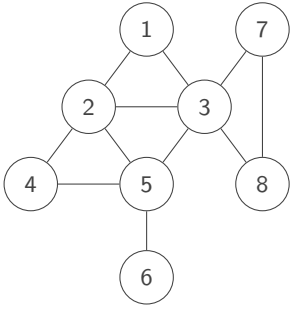
Graphen: Graphen sind ein wichtiges Werkzeug um Netzwerke, Zusammenhänge und Strukturen zu modellieren.

Beispiel: Wiener U-Bahn Linien

Ungerichtete Graphen

Ungerichteter Graph: $G = (V, E)$

- V = Menge der Knoten (*vertices, nodes*).
- E = Menge der Kanten zwischen Paaren von Knoten (*edges*).
- Notation für Kante zwischen Knoten a und b: (a, b) bzw. (b, a).
- Alternativ wird auch $a - b$ bzw. $b - a$ verwendet.
- Parameter für Größen: $n = |V|$, $m = |E|$



$V = \{1,2,3,4,5,6,7,8\}$

$E = \{1\text{-}2,1\text{-}3,2\text{-}3,2\text{-}4,2\text{-}5,3\text{-}5,3\text{-}7,3\text{-}8,4\text{-}5,5\text{-}6,7\text{-}8\}$

$n = 8$

$m = 11$

Ungerichtete Graphen: Weitere Definitionen

Adjazent, inzident, Nachbarschaft: Sei $e = (u, v)$ eine Kante in E.

- u und v sind adjazent, d.h. u ist Nachbar von v und v ist Nachbar von u.
- v (bzw. u) und e sind inzident.
- $(u, v) = (v, u)$.

Knotengrad (*degree*): $deg(v)$ bezeichnet den Knotengrad des Knotens v.

- $deg(v)$ entspricht der Anzahl der zu v inzidenten Kanten.
- Es gilt: $\sum_{v \in V} deg(v) = 2 \cdot \mid E \mid$ (Handshaking-Lemma).

Ungerichtete Graphen: Weitere Definitionen

Grundlegende Definitionen:

- Mehrfachkante: Mehrere Kanten zwischen zwei Knoten.
- Schleife: Eine Kante, die einen Knoten mit sich selbst verbindet.

Schlichter Graph: Ein ungerichteter Graph ohne Mehrfachkanten und ohne Schleifen.

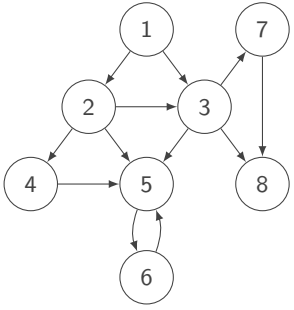
Hinweise:

- In dieser Vorlesung werden, wenn nicht anders verlautbart, schlichte Graphen betrachtet.
- Bei bestimmten Problemstellungen werden gewichtete Graphen verwendet, bei denen Knoten und/oder Kanten eine reelle Zahl zugeordnet bekommen.

Gerichtete Graphen

Gerichteter Graph (Digraph): $G = (V, E)$

- V = Menge der Knoten (*vertices, nodes*).
- E = Menge der gerichtete Kanten (*arcs*) zwischen Paaren von Knoten.
- Notation für Kante von a zu b: (a, b) bzw. $a \to b$
- $(a, b) \neq (b, a)$



$V = \{1,2,3,4,5,6,7,8\}$

$E = \{1 \to 2, 1 \to 3, 2 \to 3, 2 \to 4, 2 \to 5, 3 \to 5,$
$3 \to 7, 3 \to 8, 4 \to 5, 5 \to 6, 6 \to 5, 7 \to 8\}$

$n = 8$

$m = 12$

Hinweis: Kanten in entgegengesetzter Richtung sind auch in schlichten Digraphen erlaubt.

Gerichtete Graphen: Weitere Definitionen

Eingangsknotengrad: $deg^-(v)$ ist die Anzahl der eingehenden inzidenten Kanten.

Ausgangsknotengrad: $deg^+(v)$ ist die Anzahl der ausgehenden inzidenten Kanten.

Es gilt: $deg(v) = deg^+(v) + deg^-(v)$.

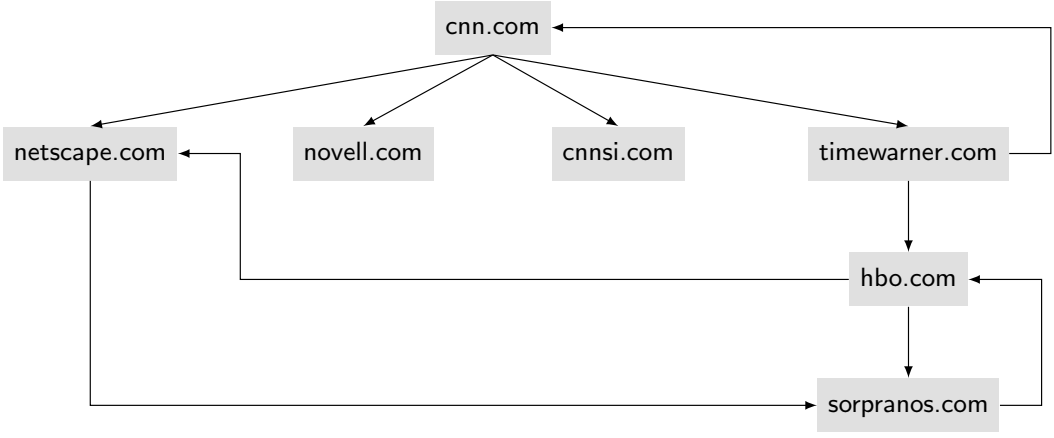
Einige Anwendungen von Graphen

Graph	*Knoten*	*Kanten*
Verkehr	Kreuzungen	Straßen
Netzwerke	Computer	Glasfaserkabel
World Wide Web	Webseiten	Hyperlinks
Sozialer Bereich	Personen	Beziehungen
Nahrungsnetz	Spezies	Räuber-Beute-Beziehung
Software	Funktionen	Funktionsaufrufe
Scheduling	Aufgaben	Ablaufeinschränkungen
elektronische Schaltungen	Gatter	Leitungen

World Wide Web

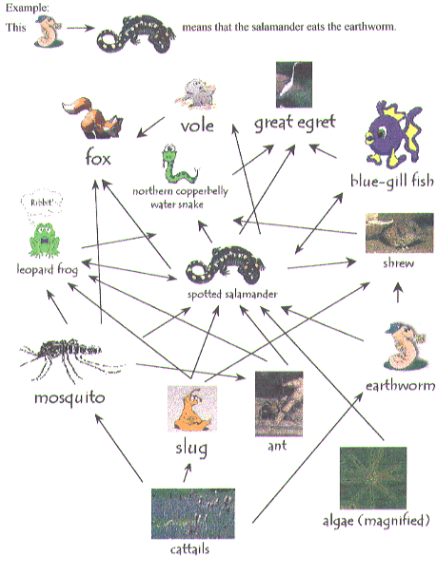
Web Graph:

- Knoten: Webseiten.
- Kante: Hyperlink von einer Seite zur anderen.



Ökologisches Nahrungsnetz

Nahrungsnetz als Graph: Knoten = Spezies, Kante = von der Beute zum Raubtier.

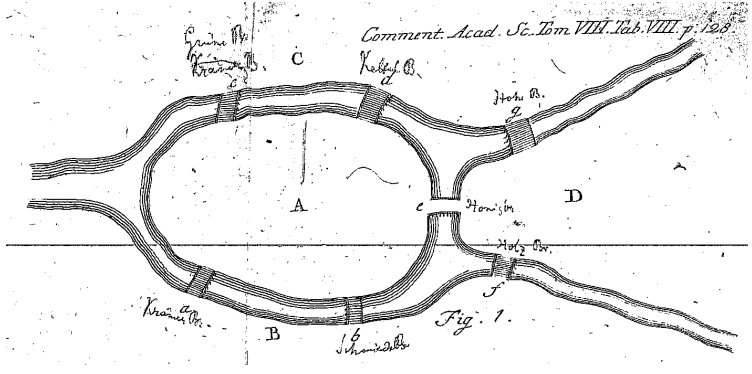


Königsberger Brückenproblem [Euler 1736]

SOLVTIO PROBLEMATIS AD GEOMETRIAM SITVS PERTINENTIS.

AVCTORE

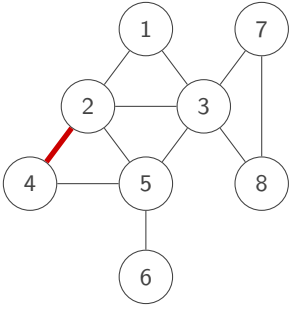
Leonh. Eulero.



Repräsentation von Graphen: Adjazenzmatrix

Adjazenzmatrix: n-mal-n Matrix mit $A_{uv} = 1$ wenn (u, v) eine Kante ist.

- Knoten: 1,2,. . . ,n.
- Zwei Einträge für jede ungerichtete Kante.
- Für gewichtete Graphen: Reelle Matrix statt Boolesche Matrix.
- Platzbedarf in $\Theta(n^2)$.
- Überprüfen, ob (u, v) eine Kante ist, hat Laufzeit $\Theta(1)$.
- Aufzählen aller Kanten hat eine Laufzeit von $\Theta(n^2)$.

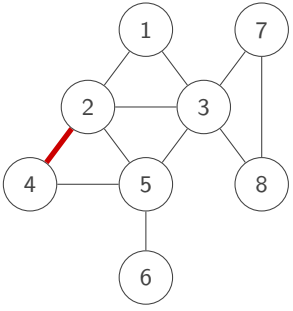


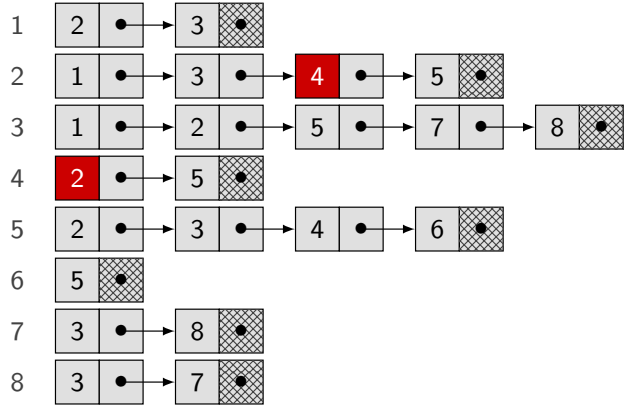
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	**1**	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	**1**	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Repräsentation von Graphen: Adjazenzlisten

Adjazenzlisten: Array von Listen. Index ist die Knotennummer.

- Knoten: 1,2,. . . ,n.
- Zwei Einträge für jede Kante.
- Für gewichtete Graphen: Speichere Gewicht in Liste.
- Platzbedarf in $\Theta(m+n)$.
- Überprüfen, ob (u,v) eine Kante ist, hat eine Laufzeit von $O(\deg(u))$.
- Aufzählen aller Kanten hat eine Laufzeit von $\Theta(m+n)$.





Adjazenzmatrix oder Adjazenzlisten

Kantenanzahl:

- Ein Graph kann bis zu $m = \frac{n(n-1)}{2} = \binom{n}{2} = \Theta(n^2)$ viele Kanten enthalten.
- Graphen sind dicht (dense) falls $m = \Theta(n^2)$.
- Graphen sind licht (sparse) falls $m = O(n)$.
- Für dichte Graphen sind beide Darstellungsformen (Adjazenzmatrix oder Adjazenzlisten) vergleichbar.

Praxis:

- Graphen, die sich aus Anwendungen ergeben, enthalten aber oft erheblich weniger Kanten.
- Typischerweise gilt dann $m = O(n)$.
- In diesem Fall ist die Darstellung mittels Adjazenzlisten günstiger.

Hinweis: Wenn wir sagen, dass ein Algorithmus auf Graphen in Linearzeit läuft, gehen wir von einer Darstellung mit Adjazenzlisten aus und betrachten eine Laufzeit von $O(n + m)$.

Ungerichtete Graphen: Kanten ausgeben

Adjazenzmatrix: Adjazenzmatrix M gegeben, n Knoten nummeriert von 0 bis $n-1$

```
for u ← 0 bis n − 2
    for v ← u + 1 bis n − 1
        if M[u, v] = 1
            Gib Kante (u, v) aus
```

Adjazenzliste: n Knoten nummeriert von 0 bis $n-1$, jeder Knoten besitzt Liste der adjazenten Knoten

```
for u ← 0 bis n − 1
    foreach Kante (u, v) inzident zu u
        if u < v
            Gib Kante (u, v) aus
```

Gerichtete Graphen: Kanten ausgeben

Adjazenzmatrix: Adjazenzmatrix M gegeben, n Knoten nummeriert von 0 bis $n-1$

```
for u ← 0 bis n − 1
    for v ← 0 bis n − 1
        if M[u, v] = 1
            Gib Kante (u, v) aus
```

Adjazenzliste: n Knoten nummeriert von 0 bis $n-1$, jeder Knoten besitzt Liste der adjazenten Knoten

```
for u ← 0 bis n − 1
    foreach Kante (u, v) inzident zu u
        Gib Kante (u, v) aus
```

Kantenzüge und Pfade

Definition: Ein Kantenzug (eng: non-simple path) in einem ungerichteten Graphen $G = (V, E)$ ist eine Folge von Knoten $v_1, v_2, \ldots, v_{k-1}, v_k$, $k \geq 1$, mit der Eigenschaft, dass jedes aufeinanderfolgende Paar v_i, v_{i+1} durch eine Kante in E verbunden ist. Die Länge des Kantenzugs ist $k - 1$.

Definition: Ein Pfad oder Weg (eng: simple path) in einem ungerichteten Graphen $G = (V, E)$ ist ein Kantenzug $v_1, v_2, \ldots, v_k$ bei dem sich kein Knoten wiederholt, also bei dem $v_i \neq v_j$ für alle $1 \leq i, j \leq k$ gilt.

Hinweis: Wir sagen auch: Der Pfad geht von v_1 nach v_k und wir bezeichnen den Pfad als v_1-v_k-Pfad.

Achtung: Die Begriffe Pfad, Weg und Kantenzug werden in der Literatur nicht einheitlich verwendet.

Zusammenhang und Distanz

Definition: Knoten u ist von Knoten v in einem Graph G erreichbar, falls G einen u-v-Pfad enthält.

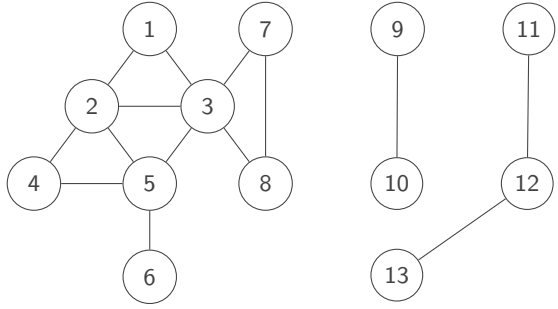
Definition: Ein ungerichteter Graph ist zusammenhängend, wenn jedes Paar von Knoten u und v von einander erreichbar ist.

Definition: Die Distanz zwischen Knoten u und v in einem ungerichteten Graphen ist die Länge eines kürzesten u-v-Pfades.

Hinweis: Falls u von v nicht erreichbar ist, nehmen wir die Distanz als ∞ an.

Zusammenhang: Beispiel

Nicht zusammenhängender Graph:

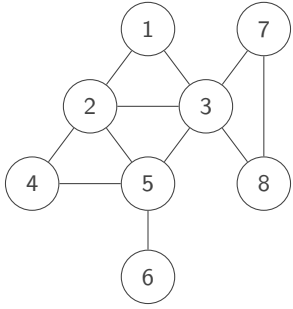


Nicht zusammenhängend: Es gibt zum Beispiel keinen Pfad vom Knoten 1 zu Knoten 10.

Beispiel für Zusammenhang: Die Knoten 1 bis 8 und ihre inzidenten Kanten bilden einen zusammenhängenden Graphen.

Kreis

Definition: Ein Kreis (eng: simple cycle) ist ein Kantenzug $v_1, v_2, \ldots, v_k$ in dem $v_1 = v_k$, $k \geq 4$, und die ersten $k-1$ Knoten alle unterschiedlich sind. Die Länge des Kreises ist $k-1$.



Beispiel für Kreis: $C =$ 1,2,4,5,3,1

Pfade und Kreise in gerichteten Graphen

Pfad: Ein Kantenzug in einem gerichteten Graphen $G = (V, E)$ ist eine Folge von Knoten $v_1, v_2, \ldots, v_{k-1}, v_k$, $k \geq 1$, mit der Eigenschaft, dass jedes aufeinanderfolgende Paar v_i, v_{i+1} durch eine gerichtete Kante (v_i, v_{i+1}) in E verbunden ist. Ein Pfad ist ein Kantenzug bei dem alle Knoten unterschiedlich sind.

Hierbei gilt:

- Der Pfad geht von einem Startknoten u zu einen Endknoten v (u-v-Pfad). Die Umkehrung muss aber nicht gelten.
- v kann von u aus erreicht werden, falls ein u-v-Pfad existiert.
- Kürzeste u-v-Kantenzüge sind Pfade.

Kreis: Ein gerichteter Kreis ist ein Kantenzug $v_1, v_2, \ldots, v_{k-1}, v_k$ in dem $v_1 = v_k$, $k \geq 3$, und die ersten $k - 1$ Knoten alle unterschiedlich sind.

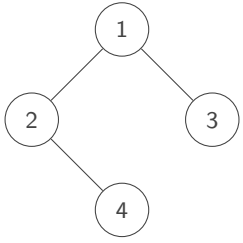
Pfade und Kreise als Graphen

Falls ein Graph G aus nur einem Pfad oder nur einem Kreis besteht, so nennen wir den ganzen Graphen einen Pfad/Kreis. Formal sagen wir:

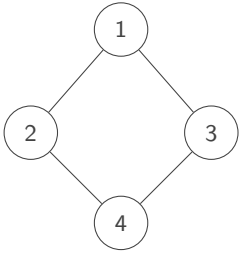
Pfad: Ein Graph G ist ein Pfad, falls es eine Aufzählung $v_1, v_2, \ldots, v_k$ der Knoten von G gibt, so dass es in G genau dann eine Kante zwischen zwei Knoten v_i und v_j gibt, falls $j = i + 1$.

Kreis: Ein Graph G ist ein Kreis, falls es eine Aufzählung $v_1, v_2, \ldots, v_k$ der Knoten von G gibt, so dass es in G genau dann eine Kante zwischen zwei Knoten v_i und v_j gibt, falls entweder $j = i + 1$ oder $i = 1$ und $j = k$ gilt.

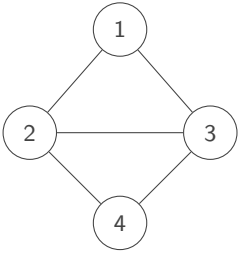
Pfade und Kreise als Graphen: Beispiele



Pfad



Kreis



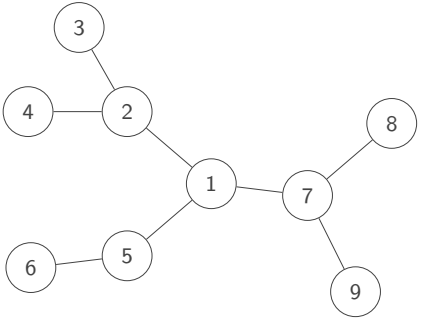
Weder Kreis noch Pfad

Bäume

Definition: Ein ungerichteter Graph ist ein Baum, wenn er zusammenhängend ist und keinen Kreis enthält.

Theorem: Sei G ein ungerichteter Graph mit n Knoten. Jeweils zwei der nachfolgenden Aussagen implizieren die dritte Aussage:

- G ist zusammenhängend.
- G enthält keinen Kreis.
- G hat $n - 1$ Kanten.



Bäume

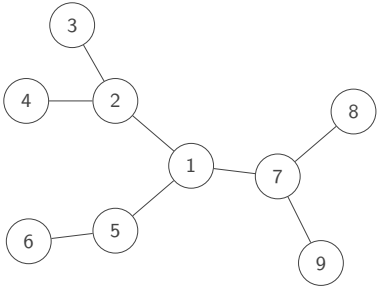
Theorem: Sei G ein ungerichteter Graph. G ist ein Baum genau dann wenn es für jedes Paar von Knoten u und v genau eine Pfad von u nach v gibt.

Beweis: G ist zusammenhängenden genau dann wenn es für jedes Paar von Knoten u und v mindestens einen Pfad von u nach v gibt. G enthält keinen Kreis, genau dann wenn es für jedes Paar von Knoten u und v maximal einen Pfad von u nach v gibt.

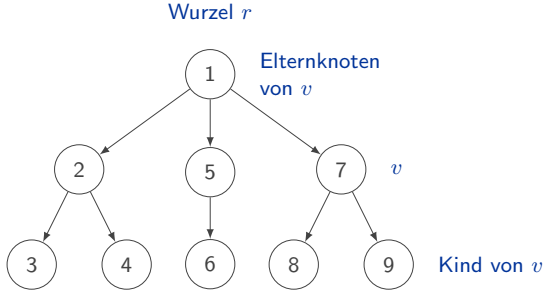
Wurzelbaum (*rooted tree, arborescence*)

Wurzelbaum: Gegeben sei ein Baum T. Wähle einen Wurzelknoten r und gib jeder Kante eine Richtung von r weg.

Bedeutung: Modelliert hierarchische Strukturen.



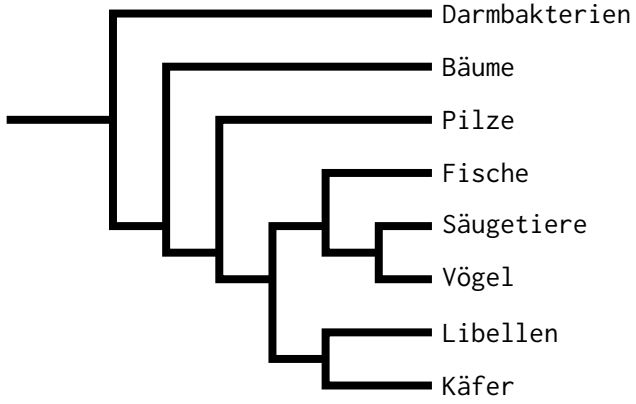
ein Baum



Ein entsprechender Wurzelbaum mit Wurzelknoten 1

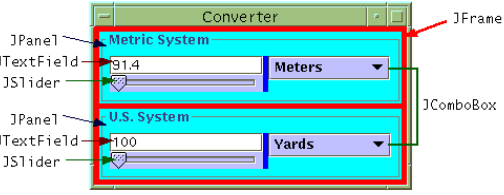
Phylogenetischer Baum

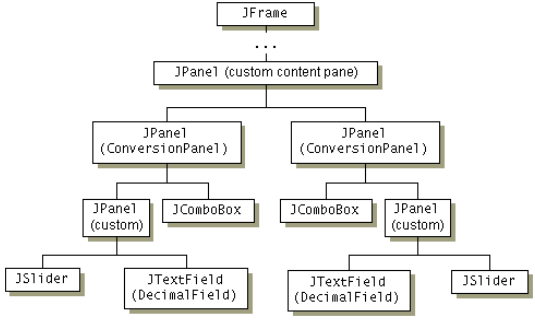
Phylogenetischer Baum: Beschreibt die evolutionären Beziehungen zwischen verschiedenen Arten.



GUI-Hierarchien

GUI-Hierarchien: Beschreiben die Organisation von GUI-Komponenten.

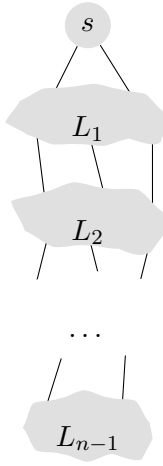




Durchmusterung von Graphen (*Graph Traversal*)

Breitensuche (*Breadth First Search, BFS*)

BFS Ansatz: Untersuche alle Knoten von einem Startknoten s ausgehend in alle möglichen Richtungen, wobei die Knoten Ebene für Ebene abgearbeitet werden.



BFS Algorithmus:

- $L_0 = \{s\}$.
- $L_1 =$ alle Nachbarn von L_0.
- $L_2 =$ alle Knoten, die nicht zu L_0 oder zu L_1 gehören und die über eine Kante mit einem Knoten in L_1 verbunden sind.
- $L_{i+1} =$ alle Knoten, die nicht zu einer vorherigen Ebene gehören und die über eine Kante mit einem Knoten in L_i verbunden sind.

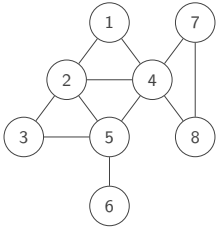
Anwendung der Breitensuche

s-t Zusammenhangsproblem: Existiert zwischen zwei gegebenen Knoten s und t ein Pfad?

s-t kürzester Pfad: Wie viele Kanten hat ein kürzester Pfad zwischen s und t ($=$ Distanz zwischen s und t)?

Anwendungen:

- Facebook.
- Labyrinth durchschreiten.
- Kevin-Bacon-Zahl.
- Die kleinste Anzahl an Hops (kürzester Pfad) zwischen zwei Knoten in einem Kommunikationsnetzwerk.



Breitensuche: Theorem

Theorem: Für jede Ebene $i = 0, 1, \ldots$ gilt, dass L_i alle Knoten mit Distanz i von s beinhaltet.

Beweis: Angenommen, sei $v_0, v_1, v_2, \ldots, v_n$ ein kürzester Pfad zwischen v_0 und v_n.

- v_0 liegt in L_0.
- v_1 liegt in L_1, da v_1 ein Nachbar von v_0 ist.
- v_2 liegt in L_2, da v_2 ein Nachbar von v_1 ist und kein Nachbar von v_0 sein kann, da es ansonsten einen kürzeren Pfad zwischen v_0 und v_n geben würde.
- Für alle weiteren Knoten gilt die gleiche Argumentation, d.h. v_n liegt schließlich in L_n. $\square$

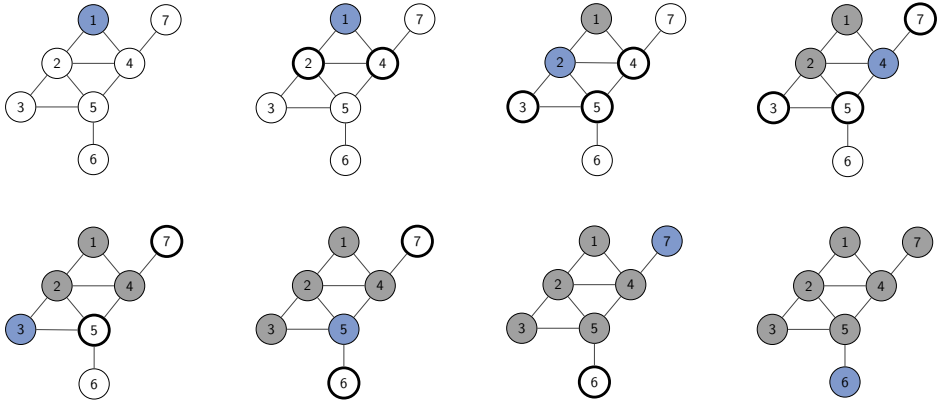
Breitensuche: Implementierung mit einer Queue

Implementierung: Array Discovered, Queue Q, Graph $G = (V, E)$, Startknoten s.

```
BFS(G,s):
Discovered[s] ← true
Discovered[v] ← false für alle anderen Knoten v ∈ V
Q ← {s}
while Q ist nicht leer
    Entferne ersten Knoten u aus Q
    Führe Operation auf u aus (z.B. Ausgabe)
    foreach Kante (u,v) inzident zu u
        if !Discovered[v]
            Discovered[v] ← true
            Füge v zu Q hinzu
```

Breitensuche: Beispiel

Möglicher Ablauf: Startknoten = 1, bearbeitete Knoten sind grau, aktiver Knoten ist blau, alle anderen Knoten sind weiß, Knoten in Queue sind mit dicken Rahmen gekennzeichnet.



Breitensuche: Analyse

Theorem: BFS hat eine Laufzeit von $O(m + n)$.

Laufzeit: Für die Laufzeitabschätzung müssen wir drei Teile betrachten:

- Initialisierung vor der while-Schleife
- while-Schleife
- foreach-Schleife

Breitensuche: Analyse

Initialisierung vor der while-Schleife:

- Jeder Knoten wird genau einmal betrachtet
- Pro Knoten können die Anweisungen in konstanter Zeit ausgeführt werden.
- Daher benötigt die Initialisierung $O(n)$ Zeit.

while-Schleife:

- Jeder Knoten u wird höchstens einmal in Q gegeben, denn nachdem er das erste mal in Q gegeben wird, wird ja Discovered[u]=true gesetzt.
- Daher wird die while-Schleife für jeden Knoten höchstens einmal durchlaufen.

Breitensuche: Analyse

foreach-Schleife:

- Sei u der gerade aktuelle Knoten bevor die foreach-Schleife ausgeführt wird.
- Dann werden in der foreach-Schleife alle Knoten v in der Adjazenzliste von u betrachtet.
- Das sind genau $deg(u)$ viele. Daher wird die Schleife $deg(u)$ mal durchlaufen. Die einzelnen Anweisungen in der Schleife benötigen konstante Zeit.

Gesamt:

- Insgesamt beträgt die Laufzeit also $O(n + \sum_{u \in V} deg(u))$.
- Da $\sum_{u \in V} deg(u) = 2m$, liegt die Laufzeit in $O(n + m)$.

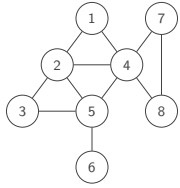
BFS-Baum

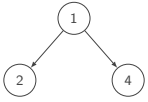
BFS-Baum: Breitensuche erzeugt einen Baum (BFS-Baum), dessen Wurzel ein Startknoten s ist und der alle von s erreichbaren Knoten beinhaltet.

Aufbau: Man startet bei s. Wird nun ein Knoten v in der Ebene L_j gefunden, ist er zu mindestens einem Knoten u der Ebene L_{j-1} benachbart. Der Knoten u von dem aus v gefunden wurde wird ausgewählt und zum Elternknoten von v im BFS-Baum gemacht.

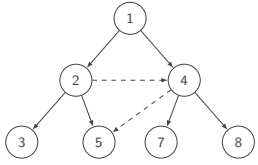
BFS-Baum: Eigenschaft

Eigenschaft: Sei T ein BFS-Baum von $G = (V, E)$ und sei (x, y) eine Kante von G. Dann können sich die Ebenen von x und y höchstens um 1 unterscheiden.

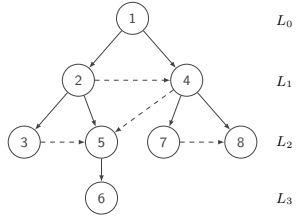




(a)



(b)



(c)

Breitensuche: Ermitteln der Ebenen

Anwendung von BFS: Ermitteln der Ebene jedes einzelnen Knotens.

Implementierung: Array Level, Queue Q, Graph $G = (V, E)$, Startknoten s.

```
BFS(G,s):
Level[s] ← 0
Level[v] ← -1 für alle anderen Knoten v ∈ V
Q ← s
while Q ist nicht leer
    Entferne ersten Knoten u aus Q
    foreach Kante (u,v) inzident zu u
        if  Level[v] == -1
            Level[v] ← Level[u] + 1
            Füge v zu Q hinzu
```

Tiefensuche (*Depth First Search, DFS*)

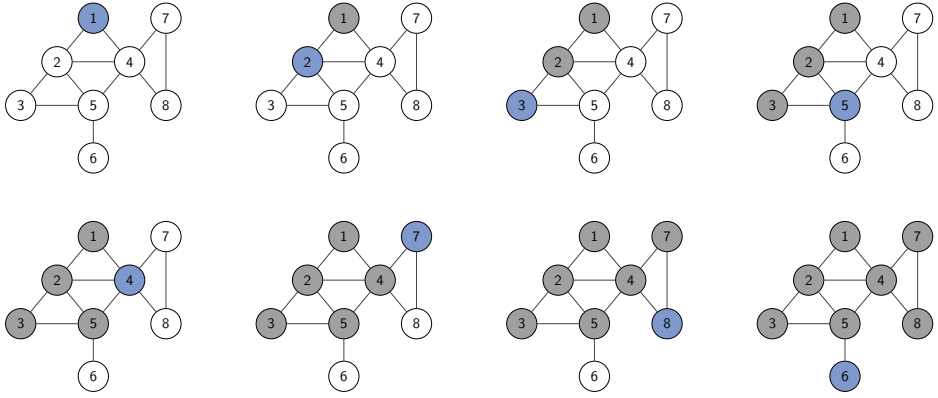
DFS Ansatz: Von einem besuchten Knoten u wird zuerst immer zu einem weiteren noch nicht besuchten Nachbarknoten gegangen (DFS-Aufruf), bevor die weiteren Nachbarknoten von u besucht werden.
DFS Algorithmus: Startknoten s, globales Array Discovered, Graph $G = (V, E)$.

```
DFS(G,s):
Discovered[v] ← false für alle Knoten v ∈ V
DFS1(G,s)

DFS1(G,u):
Discovered[u] ← true
Führe Operation auf u aus (z.B. Ausgabe)
foreach Kante (u,v) inzident zu u
    if  !Discovered[v]
        DFS1(G,v)
```

Tiefensuche: Beispiel

Möglicher Ablauf: Startknoten = 1, bearbeitete Knoten sind grau unterlegt, aktiver Knoten ist blau, alle anderen Knoten sind weiß.



Tiefensuche: Analyse

Theorem: DFS hat eine Laufzeit von $O(m+n)$.

Laufzeit: Für Laufzeitabschätzung betrachten wir:

- Initialisierung
- foreach-Schleife

Initialisierung:

- Initialisierung vor dem Aufruf von DFS1 in $O(n)$ Zeit.
- DFS1(G,u) wird für jeden Knoten u höchstens einmal aufgerufen.

Tiefensuche: Analyse

foreach-Schleife in DFS1(G,u):

- Es werden alle Knoten v in der Adjazenzliste von u betrachtet. Das sind genau $deg(u)$ viele.
- Daher wird die Schleife $deg(u)$ mal durchlaufen.
- Die einzelnen Anweisungen in der Schleife benötigen konstante Zeit (außer dem rekursiven Aufruf DFS1(G,v), aber dessen Laufzeit wird ja in der Analyse für den Knoten v berücksichtigt).

Gesamt:

- Insgesamt beträgt die Laufzeit also $O(n + \sum_{u \in V} deg(u))$.
- Da $\sum_{u \in V} deg(u) = 2m$, erhalten wir eine Laufzeit von $O(n + m)$.

Tiefensuche: Durchmusterung

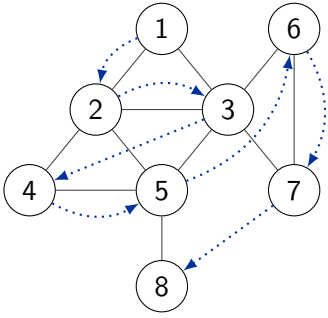
Durchmusterung: Durchmusterung bei DFS unterscheidet sich von der bei BFS.

- Es wird zunächst versucht, möglichst weit vom Startknoten weg zu kommen.
- Gibt es in der Nachbarschaft keine möglichen Knoten, dann wird durch den rekursiven Aufstieg bis zu einer möglichen Verzweigung zurückgegangen (Backtracking).

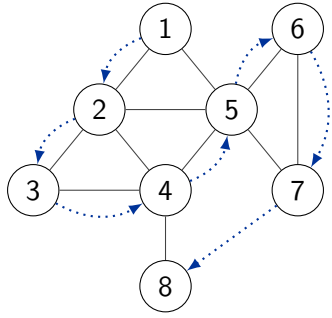
Beispiel

Vergleich: Tiefensuche und Breitensuche im Vergleich.

Breitensuche:



Tiefensuche:



Zusammenhangskomponente

Zusammenhang (Wiederholung): Ein ungerichteter Graph ist zusammenhängend, wenn für jedes Paar von Knoten u und v ein Pfad zwischen u und v existiert.

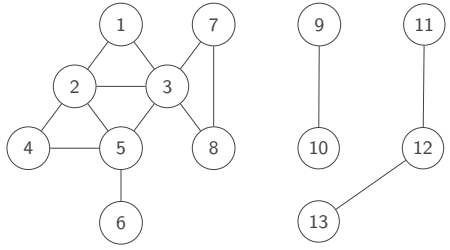
Nicht zusammenhängend: Gibt es zwischen einem Paar von Knoten keinen Pfad, dann ist der Graph nicht zusammenhängend.

Teilgraph: Ein Graph $G_1 = (V_1, E_1)$ heißt Teilgraph von $G_2 = (V_2, E_2)$, wenn seine Knotenmenge V_1 Teilmenge von V_2 und seine Kantenmenge E_1 Teilmenge von E_2 ist, also $V_1 \subseteq V_2$ und $E_1 \subseteq E_2$ gilt.

Zusammenhangskomponente: Einen maximalen zusammenhängenden Teilgraphen eines beliebigen Graphen nennt man Zusammenhangskomponente. Ein nicht zusammenhängender Graph zerfällt in seine Zusammenhangskomponenten.

Zusammenhangskomponente

Beispiel: Ein nicht zusammenhängender Graph mit 3 Zusammenhangskomponenten.



Zusammenhangskomponente

Zusammenhangskomponente: Finde alle Knoten, die von s aus erreicht werden können.

Lösung:

- Rufe DFS(G,s) oder BFS(G,s) auf.
- Ein Knoten u ist von s genau dann erreichbar, wenn Discovered[u]=true ist.

Zusammenhangskomponenten zählen

DFSNUM Algorithmus: Startknoten s, globales Array Discovered, Graph $G = (V, E)$.

```
DFSNUM(G):
Discovered[v] ← false für alle Knoten v ∈ V
i ← 0
foreach Knoten v ∈ V
    if Discovered[v] = false
        i ← i + 1
        DFS1(G,v)
return i
```

Zusammenhangskomponenten zählen

Laufzeit: Die Laufzeit liegt in $O(n+m)$.

Analyse:

- Sei $G=(V,E)$ der gegebene Graph und $G_1=(V_1,E_1),\ldots,G_r=(V_r,E_r)$ seine Zusammenhangskomponenten. Sei $|V|=n$ und $|E|=m$, sowie $|V_i|=n_i$ und $|E_i|=m_i$, für $1\leq i\leq r$.
- Klarerweise gilt $n=n_1+\ldots+n_r$ und $m=m_1+\ldots+m_r$.
- Für jede einzelne Zusammenhangskomponente G_i $(1\leq i\leq r)$ führt der Algorithmus eine Tiefensuche aus. Dies hat eine Laufzeit von $O(n_i+m_i)$.
- Die Initialisierung benötigt $O(n)$ Zeit.
- Insgesamt erhalten wir eine Laufzeit von $O(n+\sum_{i=1}^{r}(n_i+m_i))=O(2n+m)=O(n+m)$.

Zusammenhang in gerichteten Graphen

Suche in gerichteten Graphen

Gerichtete Erreichbarkeit: Gegeben sei ein Knoten s, finde alle Knoten, die von s aus erreicht werden können.

Gerichteter kürzester s-t Pfad: Gegeben seien zwei Knoten s und t, ermittle einen kürzesten Pfad von s nach t.

Suche in gerichteten Graphen: BFS und DFS können auch auf gerichtete Graphen angewendet werden.

Beispiel Webcrawler: Starte von einer Webseite s. Finde alle Webseiten, die von s aus direkt oder indirekt verlinkt sind.

Starker Zusammenhang

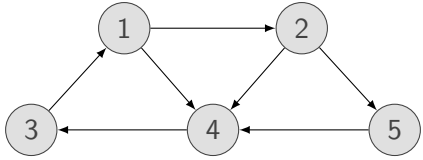
Definition: Knoten u und v in einem gerichteten Graphen sind gegenseitig erreichbar, wenn es einen Pfad von u zu v und einen Pfad von v zu u gibt.

Definition: Ein gerichteter Graph ist stark zusammenhängend, wenn jedes Paar von Knoten gegenseitig erreichbar ist.

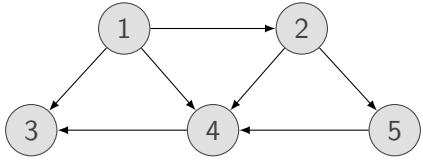
Hinweis: Ein gerichteter Graph heißt schwach zusammenhängend, falls der zugehörige ungerichtete Graph (also der Graph, der entsteht, wenn man jede gerichtete Kante durch eine ungerichtete Kante ersetzt) zusammenhängend ist.

Starker Zusammenhang: Beispiel

Stark zusammenhängend:



Nicht stark zusammenhängend (aber schwach zusammenhängend): Knoten 1 kann von keinem anderen Knoten erreicht werden, vom Knoten 3 führt kein Pfad weg.

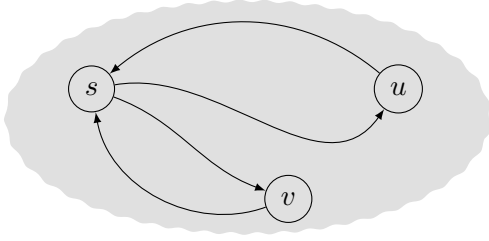


Starker Zusammenhang

Lemma: Sei s ein beliebiger Knoten in einem gerichteten Graphen G. G ist stark zusammenhängend dann und nur dann, wenn jeder Knoten von s aus und s von jedem Knoten aus erreicht werden kann.

Beweis: $\Rightarrow$ Folgt aus der Definition.
Beweis: $\Leftarrow$ Pfad von u zu v: verbinde u-s Pfad mit s-v Pfad. Pfad von v zu u: verbinde v-s Pfad mit s-u Pfad. □

- *auch ok, wenn Pfade überlappen*



Starker Zusammenhang: Algorithmus

Theorem: Laufzeit für die Überprüfung, ob G stark zusammenhängend ist, liegt in $O(m + n)$.

Beweis:

- Wähle einen beliebigen Knoten s.
- Führe BFS mit Startknoten s in G aus.
- Führe BFS mit Startknoten s in G^{rev} aus.
- Gib true zurück dann und nur dann, wenn alle Knoten in beiden BFS-Ausführungen erreicht werden können.
- Korrektheit folgt unmittelbar aus dem vorherigen Lemma. □

□ *umgekehrte Orientierung von jeder Kante in* G

DAGs und Topologische Sortierung

Gerichteter azyklischer Graph (*Directed Acyclic Graph, DAG*)

Definition: Ein DAG ist ein gerichteter Graph, der keine gerichteten Kreise enthält.

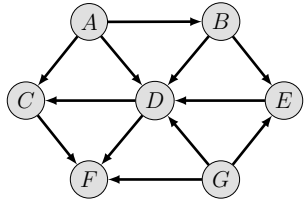
Beispiel: Knoten: Aufgaben, Kanten: Reihenfolgebeschränkungen
Kante (u, v) bedeutet, Aufgabe u muss vor Aufgabe v erledigt werden.

Definition: Wir nennen eine Knoten v ohne eingehende Kanten in einem gerichteten Graphen (i.e., $deg^-(v) = 0$) Quelle.

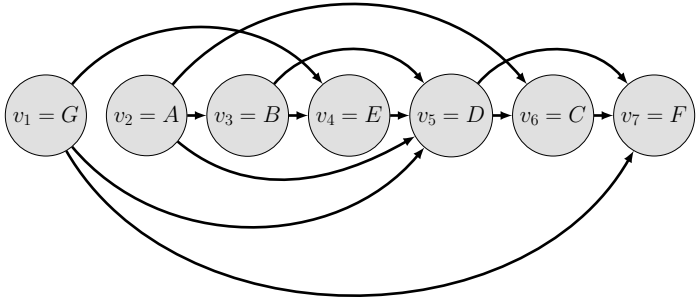
Definition: Eine topologische Sortierung eines gerichteten Graphen $G = (V, E)$ ist eine lineare Ordnung seiner Knoten, bezeichnet mit $v_1, v_2, \ldots, v_n$, sodass für jede Kante (v_i, v_j) gilt, dass $i < j$.

Topologische Sortierung: Beispiel

Ein DAG:



Eine topologische Sortierung:



Reihenfolgebeschränkung

Reihenfolgebeschränkung: Kante (u, v) bedeutet, dass Aufgabe u vor v bearbeitet werden muss.

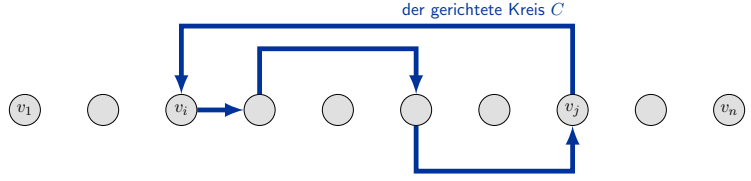
Anwendungen:

- Voraussetzungen bei Kursen: Kurs u muss vor Kurs v absolviert werden.
- Übersetzung: Modul u muss vor Modul v übersetzt werden.
- Pipeline von Prozessen: Ausgabe von Prozess u wird benötigt, um die Eingabe von v zu bestimmen.

Gerichteter azyklischer Graph

Lemma: Wenn G eine topologische Sortierung hat, dann ist G ein DAG.
Beweis: (durch Widerspruch)

- Wir nehmen an, dass G eine topologische Sortierung $v_1, \ldots, v_n$ und auch einen gerichteten Kreis C besitzt.
- Sei v_i der Knoten mit dem kleinsten Index in C und sei v_j der Knoten direkt vor v_i in C; daher gibt es die Kante (v_j, v_i).
- Durch die Wahl von i gilt, dass $i < j$.
- Andererseits, da (v_j, v_i) eine Kante ist und $v_1, \ldots, v_n$ eine topologische Sortierung ist, müsste eigentlich $j < i$ sein. Widerspruch. □



die angenommene topologische Sortierung: $v_1, \ldots, v_n$

Gerichteter azyklischer Graph

Lemma: Wenn G eine topologische Sortierung hat, dann ist G ein DAG.

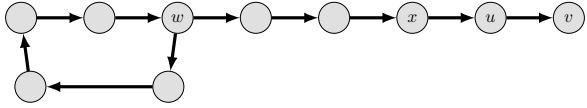
Frage: Hat jeder DAG eine topologische Sortierung?

Frage: Wenn ja, wie berechnen wir diese?

Gerichteter azyklischer Graph

Lemma: Wenn G ein DAG ist, dann hat G eine Quelle.
Beweis: (durch Widerspruch)

- Wir nehmen an, G ist ein DAG ohne Quelle.
- Wähle einen beliebigen Knoten v und folge den Kanten von v aus rückwärts. Da v zumindest eine eingehende Kante (u, v) besitzt, können wir rückwärts zu u gelangen.
- Da u zumindest eine eingehende Kante (x, u) hat, können wir rückwärts zu x gelangen.
- Das wird so oft wiederholt, bis man einen Knoten w zweimal besucht.
- Sei C die Sequenz von Knoten die zwischen zwei Besuchen von w durchlaufen wurde. C ist ein Kreis. □



Gerichteter azyklischer Graph

Lemma: G ist ein DAG genau dann wenn jeder Teilgraph von G eine Quelle hat.
Beweis:

- Angenommen G ist ein DAG, dann ist offensichtlich auch jeder Teilgraph von G ein DAG (Das Entfernen von Knoten kann keine Kreise produzieren). Deswegen hat jeder Teilgraph von G eine Quelle.
- Angenommen G ist kein DAG. Dann enthält G einen Kreis als Teilgraph. Ein Kreis hat keine Quelle.

Gerichteter azyklischer Graph - Erkennen eines DAG mittels wiederholtem Löschen von Kanten

```
while G hat mindestens einen Knoten
    if G hat eine Quelle
        Wähle eine Quelle v aus
        Gib v aus
        Lösche v und alle inzidenten Kanten aus G
    else return G ist kein DAG
return G ist ein DAG
```

Hinweis:

- Ein Knoten kann im Lauf des Algorithmus zur Quelle werden.
- Falls G ein DAG ist, gibt dieser Algorithmus eine topologische Sortierung aus.

Gerichteter azyklischer Graph

Lemma: Wenn G ein DAG ist, dann hat G eine topologische Sortierung.

Beweis:

- Falls G ein DAG ist, können wir eine topologische Sortierung berechnen.
- Falls G kein DAG ist, enthält G eine Kreis $v_1, \ldots v_n$. In der Ordnung einer topologischen Sortierung müsste dann $v_1 < \ldots v_n < v_1$ gelten. Dann ist die Ordnung allerdings keine lineare Ordnung.

Topologische Sortierung

Algorithmus: Effiziente Implementierung des Löschalgorithmus: Löschen von Knoten wird mittels Hilfsarray count simuliert. Es wird zusätzlich eine anfangs leere Liste L verwendet.

```
foreach v ∈ V
    count[v] ← 0
foreach v ∈ V
    foreach Kante (v, w) ∈ E
        count[w] ← count[w]+1
foreach v ∈ V
    if count[v] = 0
        Gib v zur Liste L am Anfang hinzu
while L ist nicht leer
    Sei v erstes Element in L, lösche v aus L
    Gib v aus
    foreach Kante (v, w) ∈ E
        count[w] ← count[w]−1
        if count[w] = 0
            Gib w zur Liste L am Anfang hinzu
```

Topologische Sortierung: Laufzeit

Theorem: Algorithmus findet eine topologische Sortierung in $O(n+m)$ Zeit.

Laufzeit: Dazu betrachten wir die folgenden Teile:

- Initialisierung
  - Erste foreach-Schleife für count.
  - Zwei verschachtelte foreach-Schleifen.
  - Dritte foreach-Schleife für Generierung der Liste.
- while-Schleife (mit foreach-Schleife).

Initialisierung:

- Die erste foreach-Schleife für die Initialisierung von count benötigt $O(n)$ Zeit.
- Bei den verschachtelten foreach-Schleifen wird die innere foreach-Schleife für jeden Knoten v genau $deg^+(v)$ mal ausgeführt. Daher benötigt man dafür $O(n+m)$ Zeit.
- Die Generierung der Liste L durch die dritte foreach-Schleife benötigt $O(n)$ Zeit.
- Daher benötigt die Initialisierung $O(n+m)$ Zeit.

Topologische Sortierung: Analyse

while-Schleife:

- Jeder Knoten v wird höchstens einmal aus L entnommen.
- Daher wird die while-Schleife für jeden Knoten höchstens einmal durchlaufen.

foreach-Schleife:

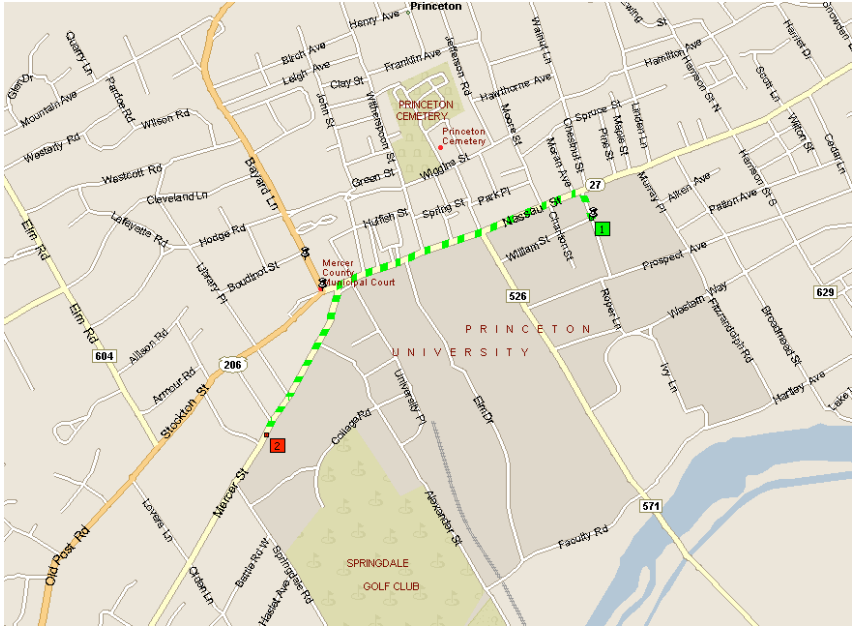
- Sei v der gerade aktuelle Knoten bevor die foreach-Schleife ausgeführt wird.
- Dann werden in der foreach-Schleife alle Knoten w in der Adjazenzliste von v betrachtet.
- Das sind genau $deg^+(v)$ viele. Daher wird die Schleife $deg^+(v)$ mal durchlaufen. Die einzelnen Anweisungen in der Schleife benötigen konstante Zeit.
- Jeder Knoten w wird höchstens einmal in L eingefügt.

Topologische Sortierung: Analyse

Gesamt:

- Initialisierung liegt in $O(n + m)$
- while-Schleife liegt in $O(n + m)$
- Daher liegt auch die gesamte Laufzeit in $O(n + m)$

Kürzeste Pfade in einem gewichteten Graphen



Kürzester Pfad vom Informatikinstitut in Princeton zu Einsteins Haus.

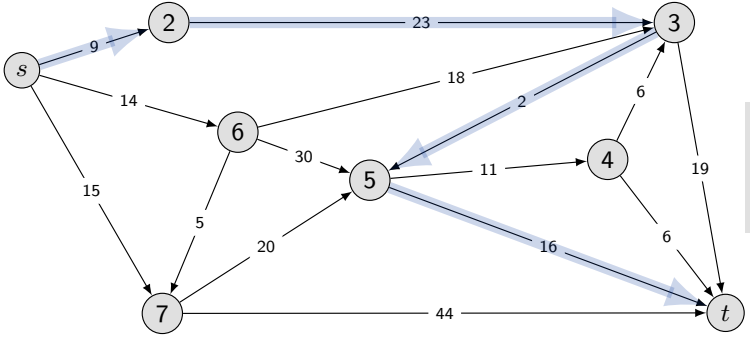
Kürzester Pfad (*Shortest Path Problem*)

Netzwerk für kürzesten Pfad:

- Gerichteter Graph $G = (V, E)$.
- Start s, Ziel t.
- Länge $\ell_e \geq 0$ ist die Länge der Kante e (Gewicht).

Kürzester Pfad: Finde kürzesten gerichteten Pfad von s nach t.

Kürzester Pfad = Pfad mit den geringsten Kosten, wobei die Kosten eines Pfades die Summe der Gewichte seiner Kanten sind.



Kosten des Pfades
s-2-3-5-t
$= 9 + 23 + 2 + 16$
$= 50.$

Dijkstra 1959

Numerische Mathematik 1, 269—271 (1959)

A Note on Two Problems in Connexion with Graphs

By

E. W. DIJKSTRA

We consider n points (nodes), some or all pairs of which are connected by a branch; the length of each branch is given. We restrict ourselves to the case where at least one path exists between any two nodes. We now consider two problems.

Algorithmus von Dijkstra

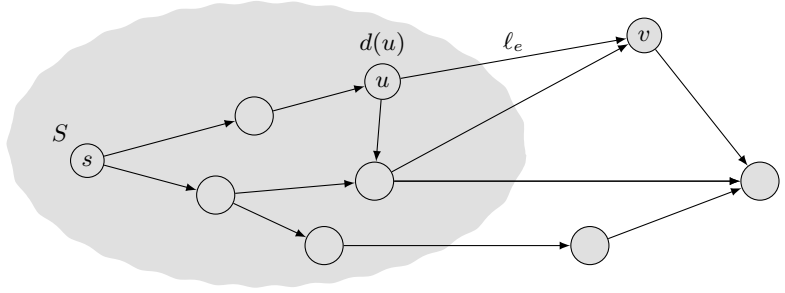
Algorithmus von Dijkstra:

- Verwalte eine Menge S von untersuchten Knoten, für die wir die Kosten $d(u)$ eines kürzeste s-u-Pfades ermittelt haben.
- Initialisiere $S = \{s\}$, $d(s) = 0$.
- Wähle wiederholt einen nicht untersuchten Knoten v, für den der folgende Wert am kleinsten ist:

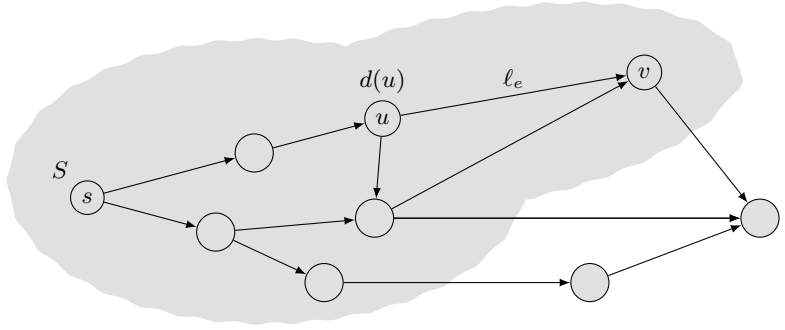
$$\min_{e=(u,v):u\in S} d(u) + \ell_e,$$

  d.h. die Länge eines kürzesten Pfades zu einem u im untersuchten Teil des Graphen, gefolgt von einer einzigen Kante (u, v).
- Füge v zu S hinzu und setze $d(v) = \min_{e=(u,v):u\in S} d(u) + \ell_e$.
- Extrahieren des Pfades entweder durch Merken des Vorgängerknotens oder mittels eines eigenen Algorithmus, der nach Dijkstra ausgeführt wird.

Algorithmus von Dijkstra: Menge S



Algorithmus von Dijkstra: Menge S



Dijkstra-Algorithmus: Implementierung

Implementierung: Wir werden zwei Implementierungen für S betrachten:

- Eine einfach verkettete Liste.
- Eine Vorrangwarteschlange (*priority queue*) von nicht untersuchten Knoten, geordnet nach den Kosten d.
  Ein Eintrag in der Queue besteht aus dem Knotenindex und den dazugehörigen Kosten.

Dijkstra-Algorithmus

Algorithmus: Arrays `Discovered` und d, Graph $G = (V, E)$, Liste L, Startknoten s.

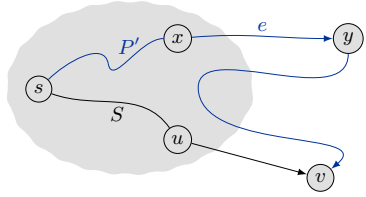
```
Dijkstra(G,s):
Discovered[v] ← false für alle Knoten v ∈ V
d[s] ← 0
d[v] ← ∞ für alle anderen Knoten v ∈ V \ {s}
L ← V
while L ist nicht leer
    wähle u ∈ L mit kleinstem Wert d[u]
    lösche u aus L
    Discovered[u] ← true
    foreach Kante e = (u,v) ∈ E
        if !Discovered[v]
            d[v] ← min(d[v], d[u]+ℓ_e)
```

Algorithmus von Dijkstra: Korrektheitsbeweis

Invariante: Für jeden Knoten $u \in S$, ist $d(u)$ die Länge eines kürzesten s-u Pfades.
Beweis: (durch Induktion nach $|S|$)
Induktionsanfang: $|S| = 1$ ist trivial.
Induktionsbehauptung: Angenommen, wahr für $|S| = k \geq 1$.

- Sei v der nächste zu S hinzugefügte Knoten und sei (u, v) die gewählte Kante.
- Ein kürzester s-u Pfad plus (u, v) ist ein s-v Pfad der Länge $d(v)$.

- Wir betrachten einen beliebigen s-v Pfad P. Wir werden zeigen, dass er nicht kürzer als $d(v)$ ist.
- Sei $e = (x, y)$ die erste Kante in P die S verlässt und sei P' der Teilpfad zu x.
- P ist schon zu lange, wenn er S verlässt.



$$\text{Länge}(P) \geq \text{Länge}(P') + \ell_e \geq d(x) + \ell_e \geq d(y) \geq d(v)$$

Nicht-negative Gewichte *Induktionsbehauptung* *Definition von* $d(y)$
Dijkstra-Algorithmus wählt v *anstatt* y

Analyse: Dijkstra-Algorithmus mit Liste

Theorem: Der Dijkstra-Algorithmus, implementiert mit einer Liste, hat eine Worst-Case-Laufzeit von $O(n^2)$.

Laufzeiten:

- Initialisierung der Arrays benötigt $O(n)$ Zeit.
- Die while-Schleife wird n-mal ausgeführt und darin muss in jeder Iteration der Knoten u mit dem kleinsten Wert für $d[u]$ gefunden werden. Das liegt in $O(n^2)$ Zeit.
- Die foreach-Schleife wird insgesamt (über alle Iterationen der while-Schleife) höchstens m-mal ausgeführt. Für jeden Knoten werden seine ausgehenden Kanten nur einmal betrachtet und insgesamt gibt es nur m Kanten.
- Daher beträgt die Laufzeit $O(n + n^2 + m)$ und somit $O(n^2)$. $\square$

Wir werden sehen, dass der Dijkstra-Algorithmus mit einer Worst-Case-Laufzeit von $O((n + m) \log n)$ implementiert werden kann. Für lichte Graphen ist das effizienter als $O(n^2)$.

Priority Queue (Vorrangwarteschlange)

Priority Queue:

- Eine Priority Queue ist eine Datenstruktur, die eine Menge S von Elementen verwaltet.
- Jedes Element $v \in S$ hat einen dazugehörigen Wert i, der die Priorität von v beschreibt.
- Kleinere Werte repräsentieren höhere Prioritäten.

Operationen: Alle mit Laufzeit in $O(\log n)$.

- Einfügen eines Elements in die Menge S.
- Löschen eines Elements aus der Menge S.
- Finden eines Elements mit dem kleinsten Wert (höchster Priorität).

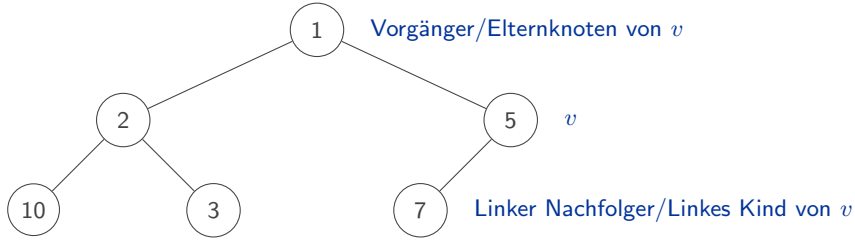
Frage: Wie erreicht man eine Laufzeit in $O(\log n)$?
Antwort: Mit einer bestimmten Datenstruktur, dem Heap.

Heap

Heap: Ein Heap (Min-Heap) ist ein binärer Wurzelbaum, dessen Knoten mit $\leq$ total geordnet sind, sodass gilt:

- Ist u ein linkes oder rechtes Kind von v, dann gilt $v \leq u$ (Heap-Eigenschaft für Min-Heap).
- Alle Ebenen von Knoten bis auf die letzte sind vollständig aufgefüllt.
- Die letzte Ebene des Baumes muss linksbündig aufgefüllt werden.

Beispiel:



Repräsentation eines Heaps

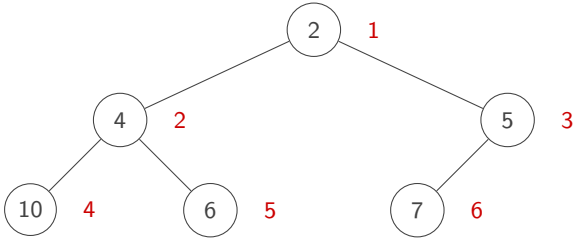
Effiziente Repräsentation: Knoten des Baums ebenenweise in einem Array speichern.

Effiziente Berechnung:

- Die beiden Nachfolgerknoten eines Knotens an der Position k befinden sich an den Positionen $2k$ und $2k+1$. Sein Elternknoten befindet sich an der Position $\lfloor \frac{k}{2} \rfloor$.
- Damit obige Rechnung immer funktioniert, wird das Array ab Index 1 belegt.
- Würde man bei Index 0 anfangen, dann würden sich die Berechnungen folgendermaßen ändern: Nachfolger links auf $2k+1$, Nachfolger rechts auf $2k+2$, Elternknoten auf $\lfloor \frac{k-1}{2} \rfloor$.

Beispiel für Heap-Repräsentation

Heap:



Array: 6 Einträge, erster Platz unbelegt (mit 0 initialisiert).

Index	0	1	2	3	4	5	6
Wert	0	2	4	5	10	6	7

Heapify-up

Einfügen eines neuen Elements: Bei einem Heap mit n Elementen wird das neue Element an Position $n+1$ eingefügt. Wir gehen dabei davon aus, dass noch genügend Plätze im Array frei sind.

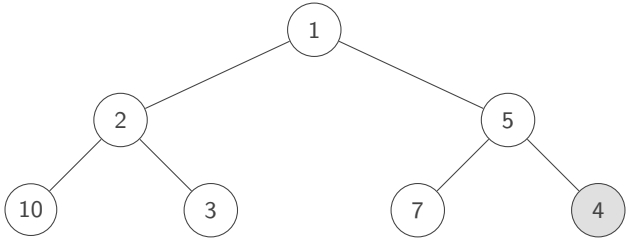
Heap-Bedingung: Die Heap-Bedingung kann durch das neue Element verletzt werden.

Reparieren: Durch Operation Heapify-up (für Heap-Array H an Position i) in $O(\log n)$ Zeit. Aufruf nach dem Einfügen des neuen Elements: Heapify-up(H,$n+1$).

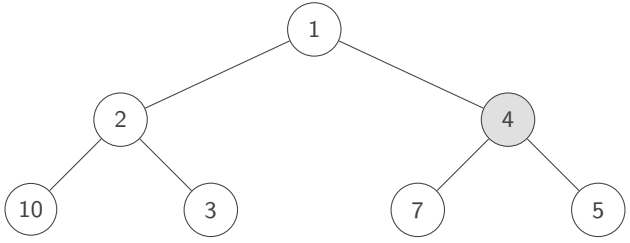
```
Heapify-up(H,i):
if i > 1
    j ← ⌊i/2⌋
    if H[i] < H[j]
        Vertausche die Array-Einträge H[i] und H[j]
        Heapify-up(H,j)
```

Beispiel für Heapify-up

Einfügen von 4:



Verschieben von 4:



Heapify-down

Löschen eines Elements: Element wird an Stelle i gelöscht. Das Element an Stelle n (bei n Elementen) wird an die freie Stelle verschoben.

Heap-Bedingung: Die Heap-Bedingung kann durch das neue Element an der Stelle i verletzt werden.

Reparieren:

- Eingefügtes Element ist zu groß: Benutze Heapify-down, um das Element auf eine untere Ebene zu bringen.
- Eingefügtes Element ist zu klein: Benutze Heapify-up (wie beim Einfügen) von der Stelle i aus.

Hinweis: Beim Heap wird typischerweise die Wurzel entfernt und daher wird dann nur Heapify-down benutzt.

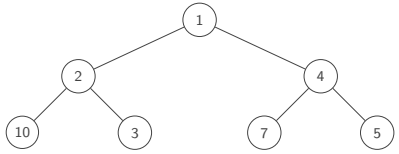
Laufzeit für Löschen: Für Heap-Array H an Position i in $O(\log n)$ Zeit.

Heapify-down

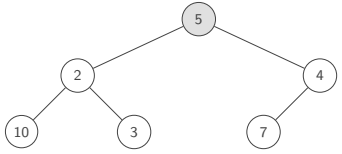
```
Heapify-down(H,i):
n ← length(H)-1
if 2 · i > n
    return
elseif 2 · i < n
    left ← 2 · i, right ← 2 · i + 1
    j ← Index des kleineren Wertes von H[left] und H[right]
else
    j ← 2 · i
if H[j] < H[i]
    Vertausche die Arrayeinträge H[i] und H[j]
    Heapify-down(H,j)
```

Beispiel für Heapify-down

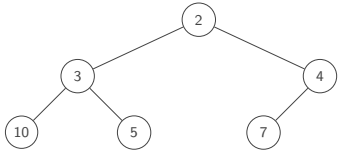
Ursprünglicher Heap:



Löschen von 1, verschieben von 5:



Heapify-down (zwei Mal)



Operationen auf Heap

Operationen auf Heap:

- Insert(H,v): Element v in den Heap H einfügen. Hat der Heap n Elemente, dann liegt die Laufzeit in $O(\log n)$.
- FindMin(H): Findet das Minimum im Heap H. Laufzeit ist konstant (da Wurzel).
- Delete(H,i): Löscht das Element im Heap H an der Stelle i. Für einen Heap mit n Elementen liegt die Laufzeit in $O(\log n)$.
- ExtractMin(H): Kombination von FindMin und Delete und daher in $O(\log n)$.

Erstellen eines Heaps

Erstellen: Das Erstellen eines Heaps aus einem Array A mit Größe n, das noch nicht die Heapeigenschaft erfüllt:

```
Init(A,n):
for i = ⌊n/2⌋ bis 1
    Heapify-down(A,i)
```

Erstellen eines Heaps: Analyse

Laufzeit: $O(n)$ ergibt sich aus folgender Berechnung:

- Einfachheitshalber nehmen wir an, der Binärbaum ist vollständig und hat n Knoten.
- Es folgt, dass $n = 2^{h+1} - 1$ wobei h die Höhe des Baumes ergibt.
- Wir lassen den Index j über die Ebenen E_j des Baumes laufen, wobei mit E_0 die Ebene mit den Blättern des Baumes bezeichnet und E_h die Ebene mit der Wurzel.
- Es folgt, dass Ebene E_j genau 2^{h-j} Knoten enthält und der Aufwand zum Einfügen eines Elements auf Ebene E_j proportional zu j ist.
- Insgesamt ergibt sich also ein Aufwand von $\sum_{j=0}^{h} j2^{h-j}$, den wir folgendermaßen abschätzen:

$$\sum_{j=0}^{h} j2^{h-j} \;=\; \sum_{j=0}^{h} j\frac{2^h}{2^j} \;=\; 2^h \sum_{j=0}^{h} \frac{j}{2^j} \;\leq\; 2^h 2 \;=\; 2^{h+1} \;=\; n+1 \;=\; O(n)$$

$\leq$ folgt aus $\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$. $=$ da $n = 2^{h+1} - 1$.

Dijkstra-Algorithmus: Effizientere Variante

Algorithmus:

- Arrays `Discovered` und d, Graph $G = (V, E)$, Startknoten s.
- Verwende Vorrangwarteschlange Q, in der die Knoten v nach dem Wert $d[v]$ geordnet sind.

```
Dijkstra(G,s):
Discovered[v] ← false für alle Knoten v ∈ V
d[s] ← 0
d[v] ← ∞ für alle anderen Knoten v ∈ V \ {s}
Q ← V
while Q ist nicht leer
    wähle u ∈ Q mit kleinstem Wert d[u]
    lösche u aus Q
    Discovered[u] ← true
    foreach Kante e = (u,v) ∈ E
        if !Discovered[v]
            if d[v] > d[u]+ℓ_e
                lösche v aus Q
                d[v] ← d[u]+ℓ_e
                füge v zu Q hinzu
```

Analyse: Dijkstra-Algorithmus mit Vorrangwarteschlange

Theorem: Der Dijkstra-Algorithmus, implementiert mit einer Vorrangwarteschlange, hat eine Worst-Case-Laufzeit von $O((n+m)\log n)$.

Laufzeiten:

- Initialisierung der Arrays benötigt $O(n)$ Zeit.
- Die while-Schleife wird n-mal ausgeführt und darin muss in jeder Iteration der Knoten u mit dem kleinsten Wert für $d[u]$ aus der Queue gelöscht werden ($O(\log n)$).
- Die foreach-Schleife liegt in $O(m \log n)$ Zeit. Für jeden Knoten werden seine ausgehenden Kanten nur einmal betrachtet und insgesamt gibt es nur m Kanten. Bei einer Neuberechnung muss aber die Queue reorganisiert werden (diese Operation liegt in $O(\log n)$).
- Daher beträgt die Laufzeit $O(n + n\log n + m\log n)$ und somit $O((n+m)\log n)$. □

Dijkstra-Algorithmus: Abschließender Vergleich

Wir vergleichen die Laufzeit des Dijkstra-Algorithmus bei Verwendung von Listen, Vorrangwarteschlange als Heap und Vorrangwarteschlange als Fibonacci-Heap (diese verbesserte Datenstruktur haben wir nicht besprochen).

Tabelle Vergleich verschiedener Datenstrukturen für Dijkstra-Algorithmus.

Liste	**Heap**	**FibHeap**
$O(n^2)$	$O((n+m)\log n)$	$O(m+n\log n)$

Greedy-Algorithmen

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 23. März 2023
Vorlesungsfolien

Einleitung

Algorithmen: Paradigmen

Greedy: Erstelle inkrementell eine Lösung, bei der nicht vorausschauend ein lokales Kriterium zur Wahl der jeweils nächsten hinzuzufügenden Lösungskomponente verwendet wird.

Divide-and-Conquer: Teile ein Problem in Teilprobleme auf. Löse jedes Teilproblem unabhängig und kombiniere die Lösung für die Teilprobleme zu einer Lösung für das ursprüngliche Problem.

Greedy: Einführendes Beispiel

Geld wechseln: Gegeben sei eine Stückelung von Münzen (z.B. Euromünzen in Cent): 1, 2, 5, 10, 20, 50, 100, 200.

Gesucht: Methode, um einen Betrag mit der kleinstmöglichen Anzahl an Münzen herauszugeben.

Beispiel:

- 37 Cent
- Optimale Lösung: 1×20, 1×10, 1×5, 1×2

Hinweis: Es kann auch mehr als eine Lösung geben.

- Stückelung von Münzen: 1, 5, 10, 20, 25, 50
- Betrag: 30
- 1×20 und 1×10 sowie 1×25 und 1×5 sind optimale Lösungen.

Geld wechseln: Greedy-Algorithmus

Greedy-Ansatz: Für Betrag S.

```
while S ≠ 0
    Finde die Münze mit größtem Wert x, sodass x ≤ S
    Benutze ⌊S/x⌋ Münzen von Wert x
    S ← S mod x
```

Geld wechseln: Greedy-Algorithmus

Greedy-Ansatz konkreter:

- Werte von m Münzen in einem Array w.
- Es gilt w[0] $>$ w[1] $> \ldots >$ w[$m-1$] $= 1$.
- Betrag S gegeben.
- Anzahl jeder einzelnen Münze, um S zu wechseln, wird in einem Array num gespeichert.
- num[i] enthält Anzahl der Münzen von Wert w[i].

for $i \leftarrow 0$ bis $m-1$
    num[i] $\leftarrow \left\lfloor \frac{S}{\texttt{w[}i\texttt{]}} \right\rfloor$
    $S \leftarrow S \mod \texttt{w[}i\texttt{]}$

Greedy-Algorithmus: Allgemeines

Greedy-Algorithmus:

- Eine Lösung wird schrittweise aufgebaut, in jedem Schritt wird das Problem auf ein kleineres Problem reduziert.
- Greedy-Prinzip: Füge jeweils eine lokal am attraktivsten erscheinende Lösungskomponente hinzu.
- Einmal getätigte Entscheidungen werden nicht mehr zurückgenommen.
- Meist einfach zu konstruieren und zu implementieren.
- Kann eine optimale Lösung liefern, muss es i.A. aber nicht.

Greedy-Algorithmus: Optimalität

Optimale Lösung: Für eine Stückelung von 1, 5 und 10 kann gezeigt werden, dass der Greedy-Algorithmus eine optimale Lösung liefert.

Beweis:

- Wir gehen von irgendeiner optimalen Lösung aus.
- Die Lösung kann nicht mehr als vier 1er haben, da fünf davon durch einen 5er ersetzt werden können.
- Die Lösung kann auch nicht mehr als einen 5er haben, da zwei davon durch einen 10er ersetzt werden können.
- Daher muss die Anzahl der 10er im Greedy-Algorithmus und in einem optimalen Algorithmus gleich sein.
- Die Anzahl der restlichen Münzen kann dann maximal 9 ergeben.
- Daher muss man nur den Fall ≤ 9 betrachten.

Greedy-Algorithmus: Optimalität

Beweis (Fortsetzung):

- Jeder Betrag < 5 kann nur durch 1er abgedeckt werden und der optimale Algorithmus und der Greedy-Algorithmus benutzen die gleiche Anzahl von 1er.
- Wenn der Betrag zwischen 5 und 9 (beide inklusive) ist, dann haben der optimale Algorithmus und der Greedy-Algorithmus genau einen 5er und der Rest wird mit 1ern aufgefüllt.
- Der Greedy-Algorithmus liefert daher die gleiche Anzahl an Münzen wie der optimale Algorithmus.

Hinweis: Für Euromünzen kann ähnlich gezeigt werden, dass der Greedy-Algorithmus optimal ist.

Greedy-Algorithmus: Optimalität

Nicht optimal:

- Gegeben sei eine Stückelung von 1, 5, 10, 20, 25.
- Bei dieser Stückelung liefert der Greedy-Algorithmus nicht immer eine optimale Lösung.

Beispiel: Mit $S = 40$.

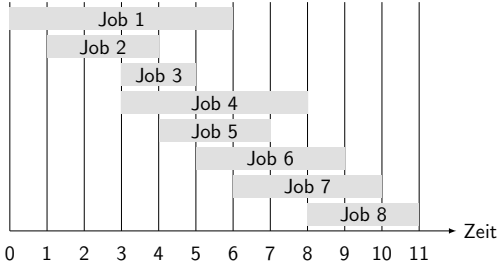
- Greedy-Algorithmus liefert 1×25, 1×10, 1×5.
- Optimale Lösung ist 2×20.

Zeitplanung von Jobs (*Interval Scheduling*)

Interval Scheduling

Interval Scheduling:

- Gegeben: Jobs $j = 1, \ldots, n$.
- Job j startet zum Zeitpunkt s_j und endet zum Zeitpunkt f_j.
- Zwei Jobs sind kompatibel, wenn sie sich nicht überlappen.
- Ziel: Finde größte Teilmenge von paarweise kompatiblen Jobs.



Beispiele: Job 2 und 5 sind kompatibel, Job 2 und 3 sind nicht kompatibel.

Interval Scheduling: Greedy-Algorithmus

Greedy-Ansatz: Betrachte die Jobs in einer natürlichen Ordnung. Wähle einen Job wenn er kompatibel (nicht überlappend) mit den bisher gewählten Jobs ist.

Mögliche Greedy-Strategien:

- [Früheste Startzeit] Berücksichtige Jobs in aufsteigender Reihenfolge von s_j.
- [Früheste Beendigungszeit] Berücksichtige Jobs in aufsteigender Reihenfolge von f_j.
- [Kleinstes Intervall] Berücksichtige Jobs in aufsteigender Reihenfolge von $f_j - s_j$.
- [Wenigste Konflikte] Zähle für jeden Job j die Anzahl c_j der nicht kompatiblen Jobs. Berücksichtige Jobs in aufsteigender Reihenfolge von c_j.

Interval Scheduling: Greedy-Algorithmus

Greedy-Ansatz: Betrachte die Jobs in einer natürlichen Ordnung. Wähle einen Job wenn er kompatibel (nicht überlappend) mit den bisher gewählten Jobs ist.

Gegenbeispiel für früheste Startzeit

Gegenbeispiel für kleinstes Intervall

Gegenbeispiel für wenigste Konflikte

Früheste Beendigungszeit: Gegenbeispiel? Nein!

Interval Scheduling: Greedy-Algorithmus

Greedy-Algorithmus: Berücksichtige Jobs in aufsteigender Reihenfolge der Beendigungszeit.
Wähle einen Job, wenn er kompatibel mit den bisher gewählten Jobs ist.

```
Sortiere Jobs nach Beendigungszeit, sodass f_1 ≤ f_2 ≤ ··· ≤ f_n
A ← ∅
for j ← 1 bis n
    if Job j ist kompatibel zu A
        A ← A ∪ {j}
return A
```

- *Menge der ausgewählten Jobs*

Interval Scheduling: Greedy-Algorithmus

Greedy-Algorithmus: Pseudocode mit angepassten Indexwerten und Array.

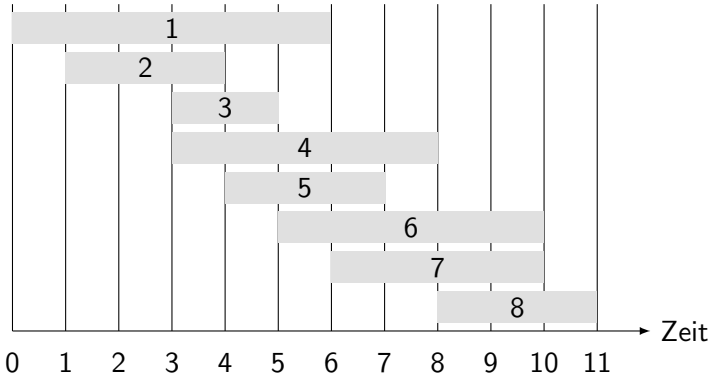
```
Sortiere Jobs nach Beendigungszeit, sodass f_1 ≤ f_2 ≤ ··· ≤ f_n
A ← ∅
t ← 0
for j ← 1 bis n
    if t ≤ s_j
        A ← A ∪ {j}
        t ← f_j
return A
```

Interval Scheduling: Greedy-Algorithmus

Implementierung: Laufzeit in $O(n \log n)$.

- Jobs werden nach Beendigungszeit sortiert und nummeriert. Wenn $f_i \leq f_j$, dann $i < j$. Die Sortierung läuft in $O(n \log n)$.
- Jobs werden vom ersten Job beginnend in der Reihenfolge ansteigender Werte für f_i ausgewählt.
- Sei die Beendigungszeit des aktuellen Jobs t:
  - Dann wird in den nachfolgenden Jobs der erste Job j gesucht, für den gilt: $s_j \geq t$.
  - Dieser Job wird der neue aktuelle Job und die Suche wird von diesem Job aus fortgesetzt.
- Der Greedy-Algorithmus kann in einem Durchlauf realisiert werden, d.h. die Laufzeit ohne Sortieren liegt in $O(n)$.
- Somit liegt die Gesamtlaufzeit in $O(n \log n)$.

Interval Scheduling: Beispiel



Jobs: Nach Beendigungszeit sortiert

Job i	2	3	1	5	4	6	7	8
s_i	1	3	0	4	3	5	6	8
f_i	4	5	6	7	8	10	10	11

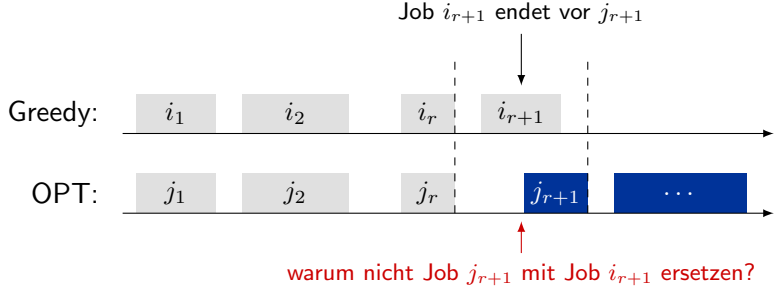
Lösung: Jobs 2, 5 und 8

Interval Scheduling: Analyse

Theorem: Der Greedy-Algorithmus liefert immer eine optimale Lösung.

Beweis: (durch Widerspruch)

- Angenommen, der Algorithmus liefert keine optimale Lösung.
- Sei $i_1, i_2, \ldots, i_k$ die Menge von Jobs, die vom Algorithmus ausgewählt wird.
- Sei $j_1, j_2, \ldots, j_m$ die Menge von Jobs in einer optimalen Lösung mit $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ für größtmögliches r.

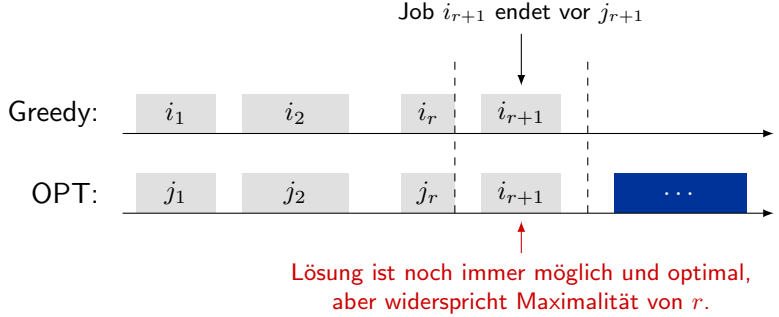


Interval Scheduling: Analyse

Theorem: Der Greedy-Algorithmus liefert immer eine optimale Lösung.

Beweis: (durch Widerspruch)

- Angenommen, der Algorithmus liefert keine optimale Lösung.
- Sei $i_1, i_2, \ldots, i_k$ die Menge von Jobs, die vom Algorithmus ausgewählt wird.
- Sei $j_1, j_2, \ldots, j_m$ die Menge von Jobs in einer optimalen Lösung mit $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$ für größtmögliches r.

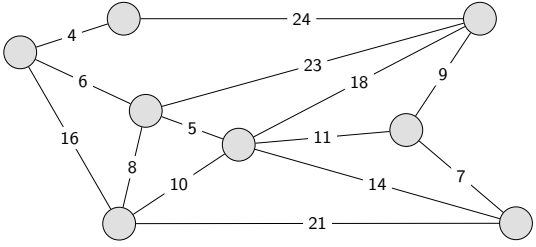


Minimaler Spannbaum

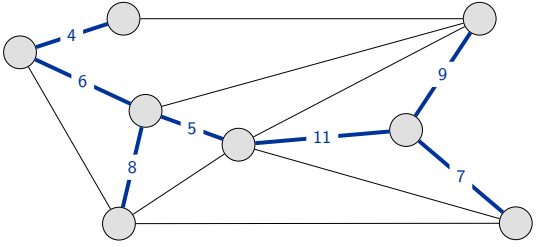
Minimaler Spannbaum

Gegeben: Ein zusammenhängender schlichter Graph $G = (V, E)$ mit reellwertigen Kantengewichten $c_e = c_{uv} = c_{vu}$ für $e = (u, v) \in E$.

Minimaler Spannbaum: Ein minimaler Spannbaum (*Minimum Spanning Tree, MST*) ist ein Teilgraph $G_T = (V, T)$ von G mit gleicher Knotenmenge und einer Teilmenge der Kanten $T \subseteq E$, sodass er ein aufspannender Baum mit minimaler Summe der Kantengewichte ist.



$G = (V, E)$



$T, \sum_{e \in E} c_e = 50$

MST-Problem

MST-Problem: Finde in einem zusammenhängenden schlichten Graph $G = (V, E)$ mit reellwertigen Kantengewichten c_e einen minimalen Spannbaum, d.h. einen zusammenhängenden, zyklenfreien Untergraphen $G_T = (V, T)$ mit $T \subseteq E$, dessen Kanten alle Knoten aufspannen und für den $cost(T) = \sum_{e \in T} c_e$ so klein wie möglich ist.

Aufwand: Es gibt exponentiell viele Spannbäume und daher wäre ein Brute-Force-Durchprobieren aller Spannbäume nicht effizient.

Lösung: Algorithmen, die in diesem Abschnitt vorgestellt werden.

Anwendungen

Das MST-Problem ist ein fundamentales Problem mit vielen unterschiedlichen Anwendungen:

- Basis für den Entwurf von Netzwerken.
  - Telefonie, Elektrizität, Kabelfernsehen, Computernetze, Straßenverkehrsnetze
- Approximationsalgorithmen für schwere Probleme.
  - Problem des Handlungsreisenden (*Travelling Salesman Problem*), Steinerbaum Problem

Greedy-Algorithmen

Algorithmen:

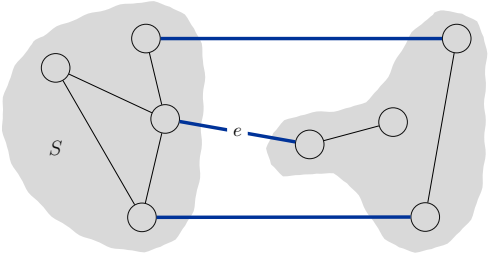
- Algorithmus von Prim: Starte mit einem beliebigen Startknoten s. Füge in jedem Schritt eine billigste Kante e zu T hinzu, die genau einen noch nicht angebundenen Knoten mit dem bisherigen Baum verbindet.
- Algorithmus von Kruskal: Starte mit $T = \emptyset$. Betrachte die Kanten in aufsteigender Reihenfolge ihrer Kosten. Füge Kante e nur dann zu T hinzu, wenn dadurch kein Kreis erzeugt wird.

Beide Algorithmen erzeugen immer einen MST.

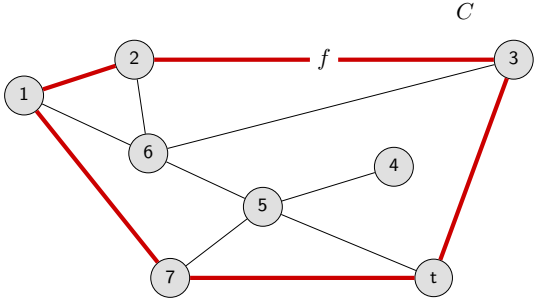
Greedy-Algorithmen: Lemmata

Kantenschnittlemma: Sei S eine beliebige Teilmenge von Knoten und sei e die minimal gewichtete Kante mit genau einem Endknoten in S. Dann enthält der MST die Kante e.

Kreislemma: Sei C ein beliebiger Kreis und sei f die maximal gewichtete Kante in C. Dann enthält der MST f nicht.



e ist im MST

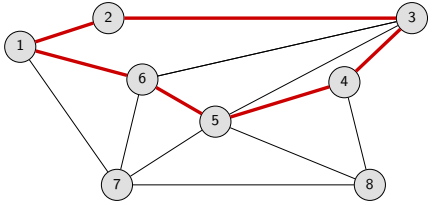


f ist nicht im MST

Vereinfachende Annahme: Alle Kantengewichte sind unterschiedlich, dadurch ist der MST eindeutig.

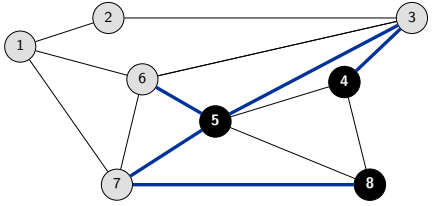
Kreise und Schnitte

Kreis: Ein Kreis ist ein Kantenzug $v_1, v_2, \ldots, v_{k\text{-}1}, v_k$ in dem $v_1 = v_k$, $k \geq 4$, und die ersten $k-1$ Knoten alle unterschiedlich sind. Alternativ kann ein Kreis als Menge $E(C)$ von Kanten der Form a-b, b-c, c-d, $\ldots$, y-z, z-a gesehen werden.



Kreis $E(C) = \{1\text{-}2, 2\text{-}3, 3\text{-}4, 4\text{-}5, 5\text{-}6, 6\text{-}1\}$

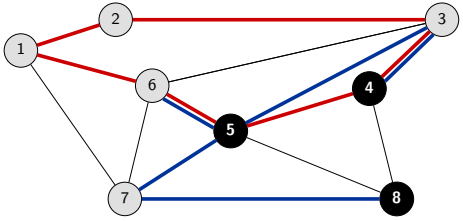
Kantenschnittmenge: Sei S eine Teilmenge der Knoten. Die dazugehörige Kantenschnittmenge D ist die Menge jener Kanten, die genau einen Endpunkt in S haben.



$S = \{4,5,8\}$
Schnittmenge $D = \{5\text{-}6, 5\text{-}7, 3\text{-}4, 3\text{-}5, 7\text{-}8\}$

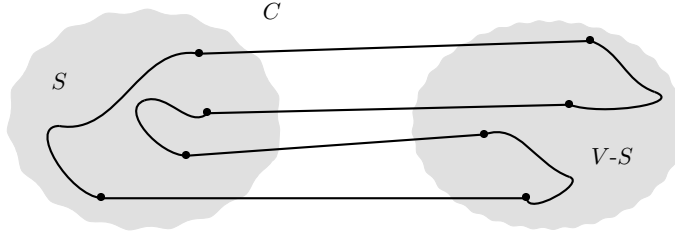
Kreise und Schnitte: Paritätslemma

Behauptung: Ein beliebiger Kreis und eine beliebige Kantenschnittmenge haben eine gerade Anzahl von Kanten gemeinsam.



Kreis $E(C) = \{1\text{-}2, 2\text{-}3, 3\text{-}4, 4\text{-}5, 5\text{-}6, 6\text{-}1\}$
Schnittmenge $D = \{3\text{-}4, 3\text{-}5, 5\text{-}6, 5\text{-}7, 7\text{-}8\}$
Durchschnitt $= \{3\text{-}4, 5\text{-}6\}$

Beweis: (durch Bild)



Beweis des Kantenschnittlemmas

Kantenschnittlemma: Sei S eine beliebige Teilmenge von Knoten und sei e die minimal gewichtete Kante mit genau einem Endknoten in S. Dann enthält der MST T^* die Kante e.

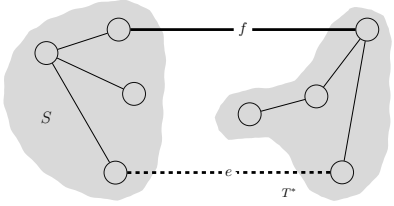
Annahme für Beweis: Alle Kantengewichte c_e sind unterschiedlich, vereinfacht Beweis.

Hinweis: Man kann zu allen Kosten kleine Störwerte hinzufügen, um die Annahme, dass alle Kanten unterschiedliches Gewicht haben müssen, zu vermeiden.

Beweis des Kantenschnittlemmas

Beweis: (Austauschargument)

- Angenommen e gehört nicht zu T^*.
- Das Hinzufügen von e zu T^* erzeugt einen Kreis $E(C)$ in T^*.
- Kante e ist sowohl im Kreis $E(C)$ als auch in der Schnittmenge D von S.
- Paritätslemma $\Rightarrow$ es existiert eine andere Kante, sagen wir f, die sich sowohl in $E(C)$ als auch in D befindet.
- $T' = T^* \cup \{e\} - \{f\}$ ist auch ein aufspannender Baum.
- Da $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$.
- Das ist ein Widerspruch zur Annahme, dass T^* minimal ist. $\square$



Beweis des Kreislemmas

Kreislemma: Sei $E(C)$ ein beliebiger Kreis in G und sei f die maximal gewichtete Kante in $E(C)$. Dann enthält kein MST die Kante f.

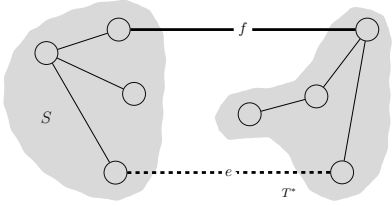
Annahme für Beweis: Alle Kantengewichte c_e sind unterschiedlich, vereinfacht Beweis.

Hinweis: Man kann zu allen Kosten kleine Störwerte hinzufügen, um die Annahme, dass alle Kanten unterschiedliches Gewicht haben müssen, zu vermeiden.

Beweis des Kreislemmas

Beweis: (Austauschargument)

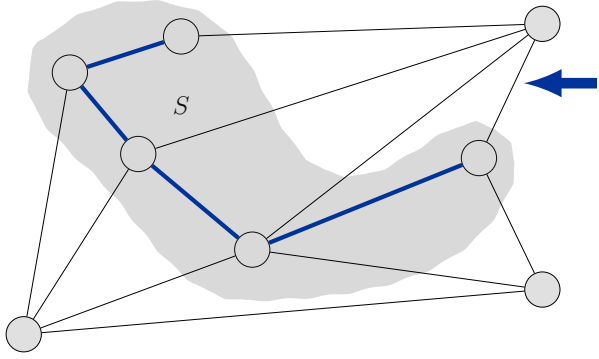
- Angenommen f gehört zu T^*
- Löschen von f aus T^* erzeugt eine Teilmenge S von Knoten in T^*.
- Kante f ist sowohl im Kreis $E(C)$ als auch in der Schnittmenge D von S.
- Paritätslemma $\Rightarrow$ es existiert eine andere Kante, sagen wir e, die sich sowohl in $E(C)$ als auch in D befindet.
- $T' = T^* \cup \{e\} - \{f\}$ ist auch ein aufspannender Baum.
- Da $c_e < c_f$, cost$(T') <$ cost(T^*).
- Das ist ein Widerspruch zur Annahme, dass T^* minimal ist. $\square$



Algorithmus von Prim

Algorithmus von Prim: [Jarnìk 1930, Dijkstra 1957, Prim 1959]

- Initialisiere S mit einem beliebigen Knoten.
- Wende das Kantenschnittlemma auf S an.
- Füge die minimal gewichtete Kante e in der Schnittmenge von S zu T hinzu und füge den Knoten u (Endknoten von e der sich noch nicht in S befindet) zu S hinzu.

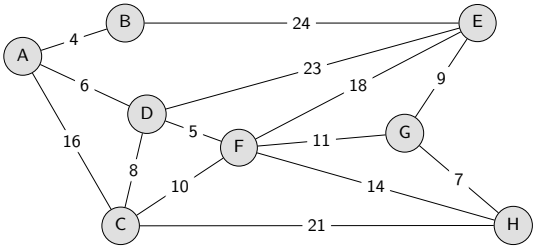


Algorithmus von Prim: Implementierung

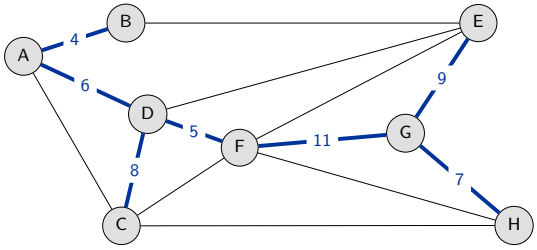
Annahme: Alle Kantengewichte sind unterschiedlich.

```
Prim(G, c):
foreach (v ∈ V)
    A[v] ← ∞
Initialisiere eine leere Priority Queue Q
foreach (v ∈ V)
    Füge v in Q ein
S ← ∅
while Q ist nicht leer
    u ← entnehme minimales Element aus Q
    S ← S ∪ {u}
    foreach Kante e = (u, v) inzident zu u
        if v ∉ S und c_e < A[v]
            Verringere die Priorität A[v] auf c_e
```

Algorithmus von Prim: Beispiel



$G = (V, E)$



$T, \sum_{e \in E} c_e = 50$

Start:

- Start bei A (willkürlich gewählt, alle Knoten gleiche Priorität)
- Priority Queue zu Beginn: A, B, C, D, E, F, G, H

Ausgewählt	Resultierende Priority Queue	Knotenmenge S	Gewicht
A	B, D, C, E, F, G, H	A	0
B	D, C, E, F, G, H	A, B	4
D	F, C, E, G, H	A, B, D	10
F	C, G, H, E	A, B, D, F	15
C	G, H, E	A, B, D, F, C	23
G	H, E	A, B, D, F, C, G	34
H	E	A, B, D, F, C, G, H	41
E		A, B, D, F, C, G, H, E	50

Algorithmus von Prim: Analyse

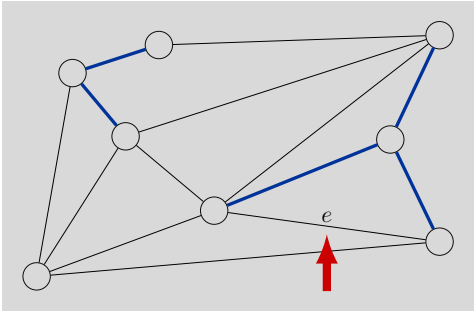
Implementierung: Benutze eine Priority Queue wie bei Dijkstra.

- Verwalte eine Menge von bearbeiteten Knoten S.
- Verwalte jeden unbearbeiteten Knoten v mit Kosten A[v] in der Priority Queue.
- A[v] sind die Kosten der billigsten Kante von v zu einem Knoten in S.
- Laufzeit in $O(n^2)$, wenn die Priority Queue mit einem Array implementiert ist.
- Laufzeit in $O(m \log n)$ mit einem binären Heap (Min-Heap).

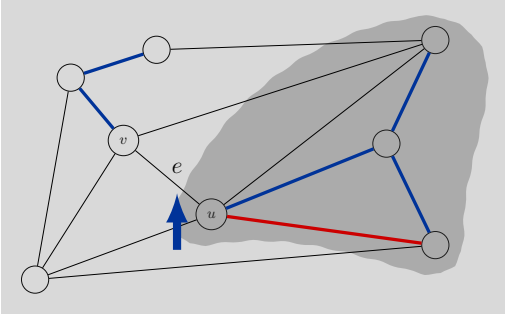
Algorithmus von Kruskal

Algorithmus von Kruskal: [Kruskal, 1956]

- Bearbeite Kanten in aufsteigender Reihenfolge der Kantengewichte.
- Fall 1: Wenn das Hinzufügen von e zu T einen Kreis erzeugt, verwerfe e gemäß des Kreislemmas.
- Fall 2: Sonst füge $e = (u, v)$ in T gemäß des Kantenschnittlemmas ein.



Fall 1



Fall 2

Algorithmus von Kruskal: Implementierung

Implementierung:

```
Kruskal(G, c):
Sortiere Kantengewichte so, dass c_1 ≤ c_2 ≤ ··· ≤ c_m
T ← ∅
foreach (u ∈ V) erzeuge eine einelementige Menge mit u
for i ← 1 bis m
    (u, v) = e_i
    if  u und v sind in verschiedenen Mengen
        T ← T ∪ {e_i}
        Vereinige die Mengen mit u und v
return T
```

- *sind u und v in unterschiedlichen Zusammenhangskomponenten?*
- *Vereinige zwei Komponenten*

Algorithmus von Kruskal: Implementierung

Sind u und v in verschiedenen Zusammenhangskomponenten?

Einfache Möglichkeit: Verwende Tiefen- oder Breitensuche

Effizienter: Benutze die sogenannte Union-Find-Datenstruktur.

- Verwalte die Teilmenge aller Knoten für jede Zusammenhangskomponente.
- $O(m \log n)$ für die Sortierung ($m \leq n^2 \Rightarrow \log m$ ist $O(\log n)$).

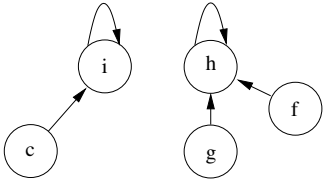
Union-Find-Datenstruktur: Abstrakter Datentyp

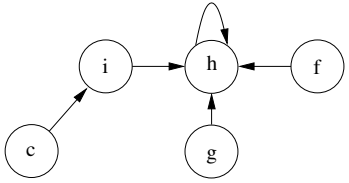
Abstrakter Datentyp: Dynamische Disjunkte Mengen (DDM)

Familie $S = \{S_1, S_2, \ldots, S_k\}$ disjunkter Teilmengen einer Menge M.
Jedes S_i hat einen Repräsentanten.

- makeset(v): Erzeugt eine Menge $\{v\} = S_v$; v ist Repräsentant von S_v
- union(v, w): Vereinigt Mengen S_v und S_w deren Repräsentanten v und w sind; neuer Repräsentant ist ein beliebiges $u \in S_v \cup S_w$
- findset(v): Liefert Repräsentanten der Menge S mit $v \in S$

Die Union Find Datenstruktur





v	a	b	c	d	e	f	g	h	i
parent$[v]$	a	b	i	d	e	h	h	h	h

Die Union Find Datenstruktur: Implementierung

Einfache Implementierung:

- makeset (v):

$$\texttt{parent}[v] = v$$

- union (v, w):

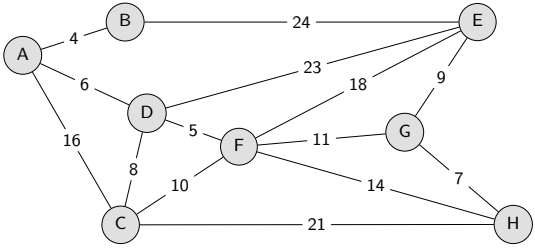
$$\texttt{parent}[v] = w$$

- findset (v):

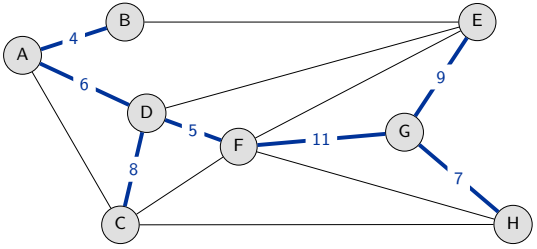
```
h = v
while parent[h] ≠ h
    h = parent[h]
return h;
```

Laufzeit: Mit einer verbesserten Implementierung kann eine in der Praxis nahezu konstante Laufzeit für jede der drei Operationen erreicht werden.

Algorithmus von Kruskal: Beispiel



$G = (V, E)$



$T, \sum_{e \in E} c_e = 50$

Start: Kanten sortiert nach Gewicht (kleinstes zuerst): (A,B), (D,F), (A,D), (G,H), (C,D), (E,G), (C,F), (F,G), (F,H), (A,C) ...

Mengen	Kante	Hinzu?	T
{**A**},{**B**},{C},{D},{E},{F},{G},{H}	(A,B)	Ja	{(A,B)}
{A,B},{C},{**D**},{E},{**F**},{G},{H}	(D,F)	Ja	{(A,B), (D,F)}
{**A**,B},{C},{**D**,F},{E},{G},{H}	(A,D)	Ja	{(A,B), (D,F), (A,D)}
{A,B,D,F},{C},{E},{**G**},{**H**}	(G,H)	Ja	{(A,B), (D,F), (A,D), (G,H)}
{A,B,**D**,F},{**C**},{E},{G,H}	(C,D)	Ja	{(A,B), (D,F), (A,D), (G,H), (C,D)}
{A,B,C,D,F},{**E**},{**G**,H}	(E,G)	Ja	{(A,B), (D,F), (A,D), (G,H), (C,D), (E,G)}
{A,B,**C**,D,**F**},{E,G,H}	(C,F)	Nein	{(A,B), (D,F), (A,D), (G,H), (C,D), (E,G)}
{A,B,C,D,**F**},{E,**G**,H}	(F,G)	Ja	{(A,B), (D,F), (A,D), (G,H), (C,D), (E,G), (F,G)}
{A,B,C,D,E,F,G,H}	...	...	...

- *Ab jetzt werden keine weiteren Kanten mehr aufgenommen!*

Kruskal und Prim im Vergleich

Laufzeit von Kruskal:

- Union-Find-Operation ist praktisch in konstanter Zeit möglich, d.h. der zweite Teil des Kruskal-Algorithmus hat nahezu lineare Laufzeit.
- Der Gesamtaufwand wird nun durch das Kantensortieren bestimmt und ist somit $O(m \log n)$.

Laufzeit von Prim:

- Wird als Priority Queue ein klassischer Heap verwendet, dann ist der Gesamtaufwand $O(m \log n)$.
- Wird ein sogenannter Fibonacci-Heap verwendet, so reduziert sich die Laufzeit auf $O(m + n \log n)$.

Anwendung in der Praxis:

- Für dichte Graphen ($m = \Theta(n^2)$) ist Prims Algorithmus besser geeignet.
- Für dünne Graphen ($m = \Theta(n)$) ist Kruskals Algorithmus besser geeignet.

Divide-and-Conquer

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 28. März 2023
Vorlesungsfolien

Algorithmen: Paradigmen

Greedy: Erstelle inkrementell eine Lösung, bei der nicht vorausschauend ein lokales Kriterium zur Wahl der jeweils nächsten hinzuzufügenden Lösungskomponente verwendet wird.

Divide-and-Conquer: Teile ein Problem in Teilprobleme auf. Löse jedes Teilproblem unabhängig und kombiniere die Lösung für die Teilprobleme zu einer Lösung für das ursprüngliche Problem.

Teile und Herrsche (*Divide-and-Conquer*)

Divide-and-Conquer: Allgemeines Prinzip, das häufig zu effizienten Problemlösungsstrategien führt.

Vorgehensweise:

- Teile Problem in mehrere Teile auf (meistens zwei).
- Löse jeden Teil rekursiv.
- Fasse Lösungen der Subprobleme zu einer Gesamtlösung zusammen.

Divide et impera.
Veni, vidi, vici.
Julius Caesar

Mergesort

Sortieren: Wiederholung

Primitive Sortierverfahren:

- Bubblesort
- Insertionsort
- Selectionsort

Laufzeit: Die Laufzeit dieser Verfahren liegt im Worst- und Average-Case immer in $\Theta(n^2)$.

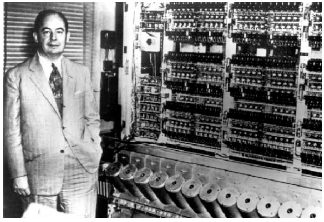
Frage: Kann man im Worst- und Average-Case schneller sortieren?

Antwort: Ja. Mergesort ist ein Beispiel dafür.

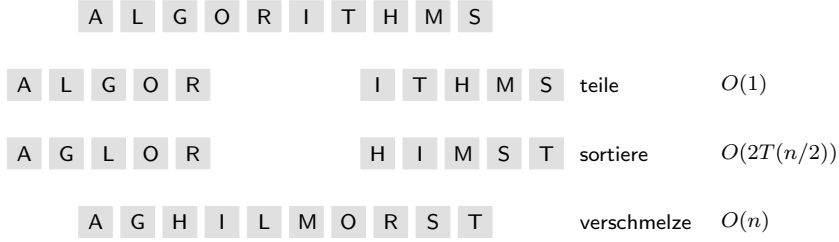
Mergesort (Sortieren durch Mischen)

Mergesort:

- Teile Array in zwei Hälften.
- Sortiere jede Hälfte rekursiv.
- Verschmelze zwei Hälften zu einem sortierten Ganzen.



John von Neumann (1945)



Mergesort

Pseudocode:

- Mergesort für ein Array A.
- Sortiert den Bereich A[l] bis A[r].

```
Mergesort(A,l,r):
if l < r
    m ← ⌊(l + r)/2⌋
    Mergesort(A,l,m)
    Mergesort(A,m + 1,r)
    Merge(A,l,m,r)
```

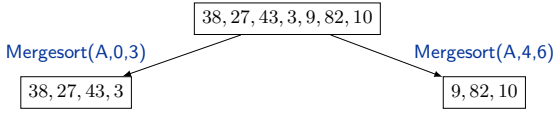
Aufruf: Mergesort(A, 0, $n-1$) für ein Array A mit n Elementen.

Mergesort: Beispiel

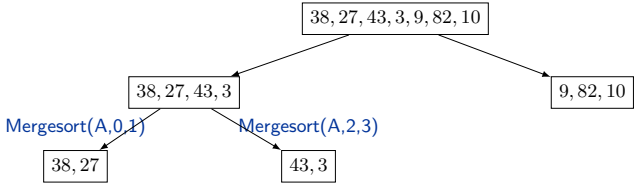
Mergesort(A,0,6)

$$\boxed{38, 27, 43, 3, 9, 82, 10}$$

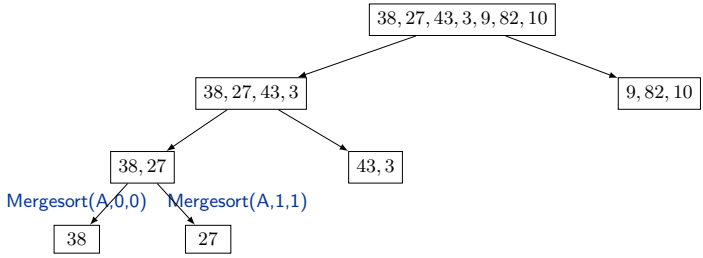
Mergesort: Beispiel



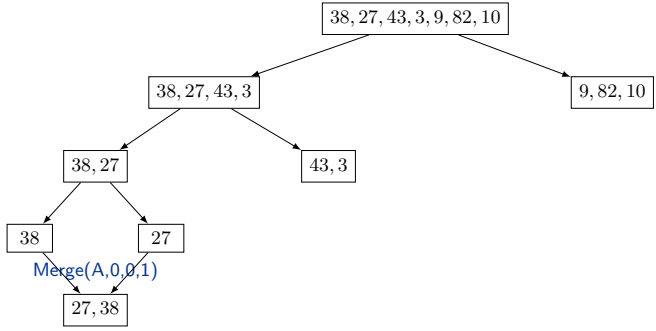
Mergesort: Beispiel



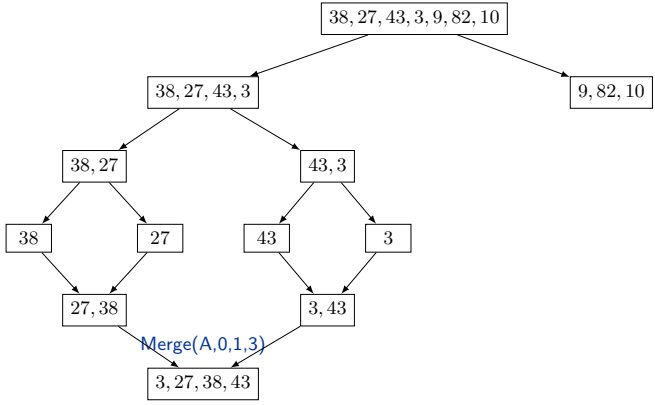
Mergesort: Beispiel



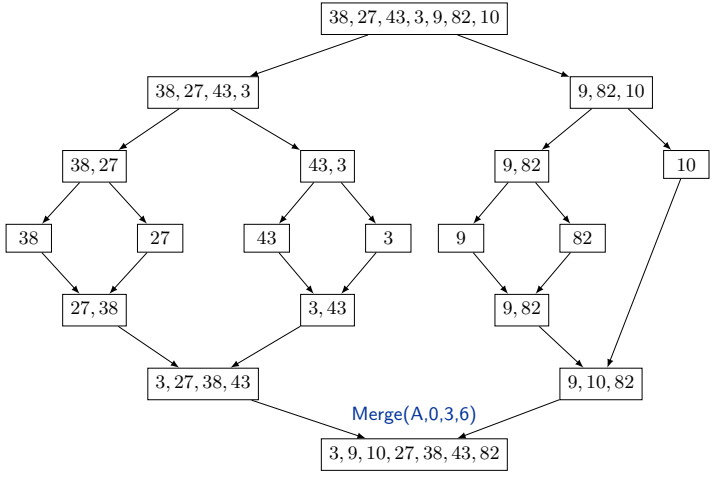
Mergesort: Beispiel



Mergesort: Beispiel



Mergesort: Beispiel



Merging (Verschmelzen)

Vorgehen: Verschmelze zwei sortierte Listen zu einer sortierten Gesamtliste.

Wie kann man effizient verschmelzen?

- Benutze temporäres Array.
- Durchlaufe beide Listen vom Anfang an.
- Führe die Elemente beider Listen im Reißverschlussverfahren zusammen, übernehme dabei jeweils das kleinste Element der beiden Listen.
- Hat lineare Laufzeit.

Verschmelzen

Pseudocode: Merge auf ein Array A. Verwendet Hilfsarray B.

```
Merge(A,l,m,r):
i ← l, j ← m + 1, k ← l
while i ≤ m und j ≤ r
    if A[i] ≤ A[j]
        B[k] ← A[i], i ← i + 1
    else
        B[k] ← A[j], j ← j + 1
    k ← k + 1
if i > m
    for h ← j bis r
        B[k] ← A[h], k ← k + 1
else
    for h ← i bis m
        B[k] ← A[h], k ← k + 1
for h ← l bis r
    A[h] ← B[h]
```

Eine nützliche Rekursionsgleichung

Definition: $C(n) =$ Anzahl der Schlüsselvergleiche (*comparisons*) in Mergesort bei einer Eingabegröße n.

Mergesort Rekursion:

$$C(n) \leq \begin{cases} 0 & \text{wenn } n = 1 \\ \underbrace{C\left(\lceil n/2 \rceil\right)}_{\text{linke Hälfte}} + \underbrace{C\left(\lfloor n/2 \rfloor\right)}_{\text{rechte Hälfte}} + \underbrace{n}_{\text{Verschmelzen}} & \text{sonst} \end{cases}$$

Lösung: $C(n) = O(n \log_2 n)$.

Eine nützliche Rekursionsgleichung

Beweis: Wir beschreiben mehrere Wege das $O(n \log_2 n)$ zu beweisen.

Annahmen:

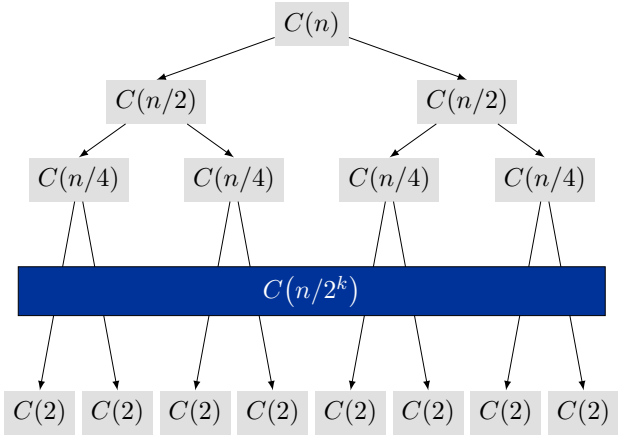
- Wir nehmen anfänglich an, dass n eine Zweierpotenz ist.
- Für ein allgemeines n' mit $\frac{n}{2} < n' < n$ (wobei n eine Zweierpotenz ist) gilt dann:

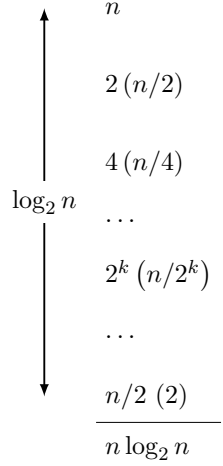
$$C(n') = O\left(n \log n\right)$$

da $O(\frac{n}{2} \log \frac{n}{2}) = O(n \log n)$ gilt.

Beweis durch Rekursionsbaum

$$C(n) \leq \begin{cases} 0 & \text{wenn } n = 1 \\ \underbrace{2C\left(n/2\right)}_{\text{Sortieren beider Hälften}} + \underbrace{n}_{\text{Verschmelzen}} & \text{sonst} \end{cases}$$





Beweis durch Auflösen der Rekursion

Behauptung: Wenn $C(n)$ die Rekursion erfüllt, dann $C(n) \leq n \log_2 n$.

$$C(n) \leq \begin{cases} 0 & \text{wenn } n = 1 \\ \underbrace{2C\,(n/2)}_{\text{Sortieren beider Hälften}} + \underbrace{n}_{\text{Verschmelzen}} & \text{sonst} \end{cases}$$

Beweis: Für $n > 1$:

$$\begin{aligned} \frac{C(n)}{n} &\leq \frac{2C(n/2)}{n} + 1 = \frac{C(n/2)}{n/2} + 1 \\ &\leq \frac{2C(n/4)}{n/2} + \frac{n/2}{n/2} + 1 = \frac{C(n/4)}{n/4} + 1 + 1 \\ &\leq \dots \\ &\leq \frac{C(n/n)}{n/n} + \underbrace{1 + \dots + 1}_{\log_2 n} \\ &= \log_2 n \end{aligned}$$

Beweis durch Induktion

Behauptung: Wenn $C(n)$ die Rekursion erfüllt, dann $C(n) \leq n \log_2 n$.

$$C(n) \leq \begin{cases} 0 & \text{wenn } n = 1 \\ \underbrace{2C\,(n/2)}_{\text{Sortieren beider Hälften}} + \underbrace{n}_{\text{Verschmelzen}} & \text{sonst} \end{cases}$$

Beweis: (durch Induktion auf n)

- Induktionsanfang: $n = 1$.
- Induktionsbehauptung: $C(n) \leq n \log_2 n$.
- Ziel: Zeige, dass $C(2n) \leq 2n \log_2 2n$.

$$\begin{aligned} C(2n) &\leq 2C(n) + 2n \\ &\leq 2n \log_2 n + 2n \\ &= 2n(\log_2 n + 1) \\ &= 2n(\log_2(2n/2) + 1) \\ &= 2n(\log_2 2n - \log_2 2 + 1) \\ &= 2n(\log_2 2n - 1 + 1) \\ &= 2n \log_2 2n \end{aligned}$$

Analyse der Mergesort-Rekursion

Behauptung: Wenn $C(n)$ die folgende Rekursion erfüllt, dann $C(n) \leq n\lceil \log_2 n \rceil$.

$$C(n) \leq \begin{cases} 0 & \text{wenn } n = 1 \\ \underbrace{C\left(\lceil n/2 \rceil\right)}_{\text{linke Hälfte}} + \underbrace{C\left(\lfloor n/2 \rfloor\right)}_{\text{rechte Hälfte}} + \underbrace{n}_{\text{Verschmelzen}} & \text{sonst} \end{cases}$$

Beweis: (durch Induktion auf n)

- Induktionsanfang: $n = 1$.
- Definiere $n_1 = \lceil n/2 \rceil$, $n_2 = \lfloor n/2 \rfloor$.
- Induktionsschritt: Ist wahr für $1, 2, \ldots, n-1$.

$$\begin{aligned} C(n) &\leq C(n_1) + C(n_2) + n \\ &\leq n_1 \lceil \log_2 n_1 \rceil + n_2 \lceil \log_2 n_2 \rceil + n \\ &\leq n_1 \lceil \log_2 n_1 \rceil + n_2 \lceil \log_2 n_1 \rceil + n \\ &= n \lceil \log_2 n_1 \rceil + n \\ &\leq n \left(\lceil \log_2 n \rceil - 1 \right) + n \\ &= n \lceil \log_2 n \rceil \end{aligned}$$

$$\begin{aligned} n_1 &= \lceil n/2 \rceil \\ &\leq \lceil 2^{\lceil \log_2 n \rceil}/2 \rceil \\ &= 2^{\lceil \log_2 n \rceil}/2 \\ &\Rightarrow \log_2 n_1 \leq \lceil \log_2 n \rceil - 1 \end{aligned}$$

Analyse der Mergesort-Rekursion

Asymptotische untere Schranke für $C(n)$: Das Verschmelzen für n Elemente benötigt zumindest $C_{\text{best}} = \lfloor \frac{n}{2} \rfloor$ Schlüsselvergleiche.

Daher gilt:

$$C_{\text{best}}(n) = C_{\text{worst}}(n) = C_{\text{avg}}(n) = \Theta(n \log n)$$

Allgemein: Die Laufzeit von Mergesort wird durch die Anzahl der notwendigen Vergleiche dominiert.

Daher gilt:

$$T_{\text{best}}(n) = T_{\text{worst}}(n) = T_{\text{avg}}(n) = \Theta(C(n)) = \Theta(n \log n)$$

Quicksort

Quicksort

Quicksort: Benutzt auch das Divide-and-Conquer-Prinzip, aber auf eine andere Art und Weise.

Teile: Wähle „Pivotelement“ x aus Folge A,
z.B. das an der letzten Stelle stehende Element.
Teile A ohne x so in zwei Teilfolgen A_1 und A_2, dass gilt:

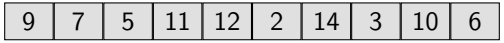
- A_1 enthält nur Elemente $\leq x$.
- A_2 enthält nur Elemente $\geq x$.

Herrsche:

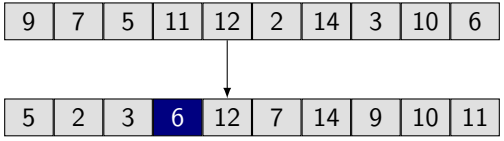
- Rekursiver Aufruf für A_1.
- Rekursiver Aufruf für A_2.

Kombiniere: Bilde A durch Hintereinanderfügen von A_1, x, A_2

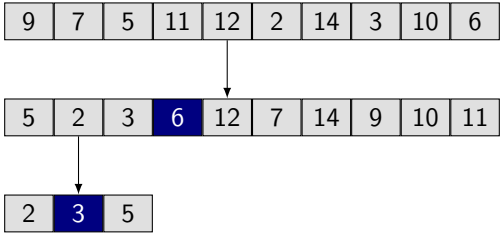
Quicksort: Beispiel



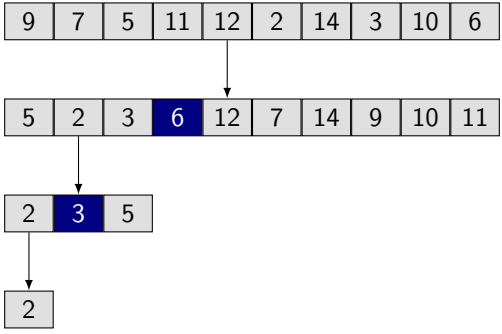
Quicksort: Beispiel



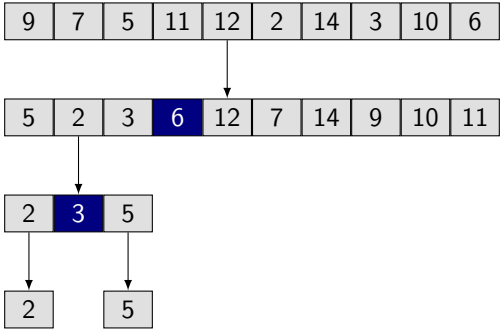
Quicksort: Beispiel



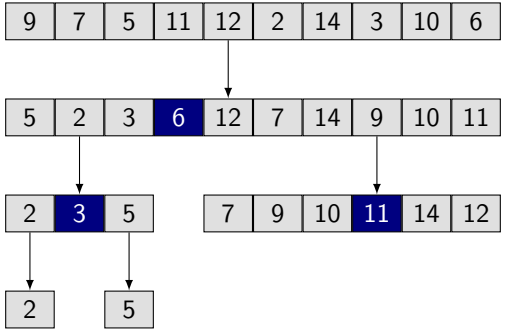
Quicksort: Beispiel



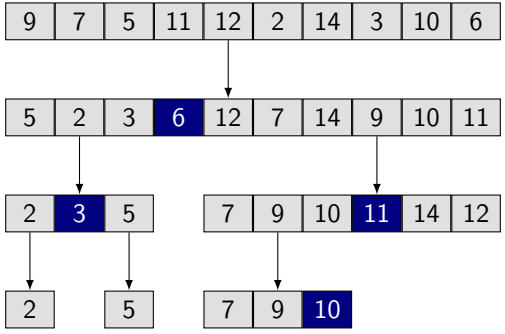
Quicksort: Beispiel



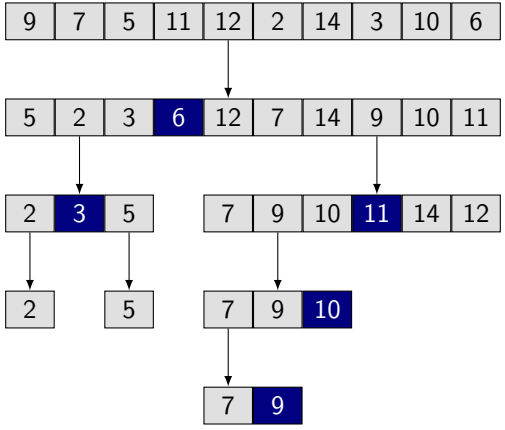
Quicksort: Beispiel



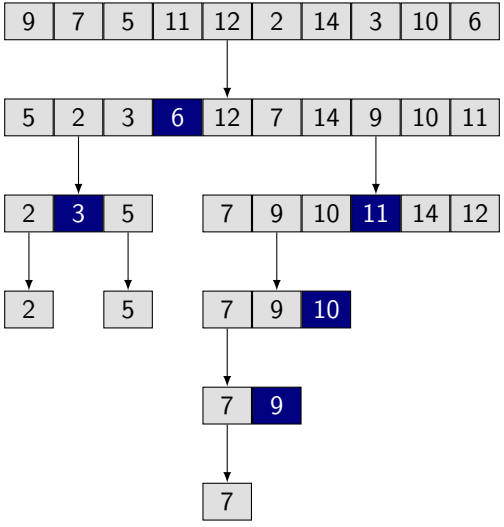
Quicksort: Beispiel



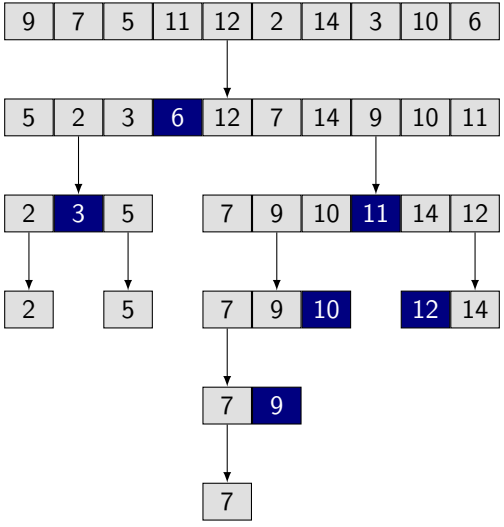
Quicksort: Beispiel



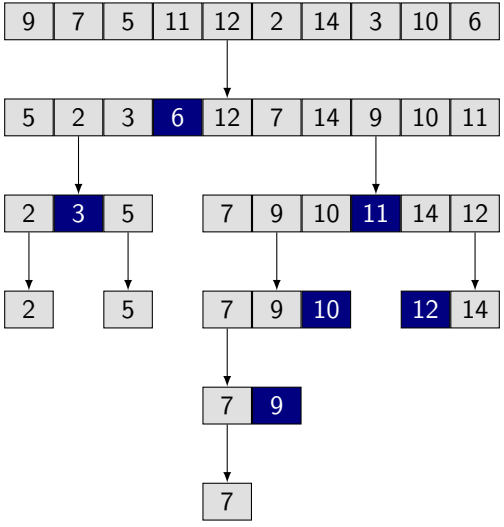
Quicksort: Beispiel



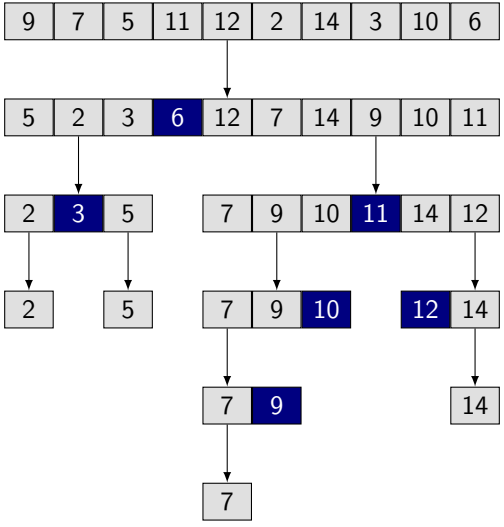
Quicksort: Beispiel



Quicksort: Beispiel



Quicksort: Beispiel



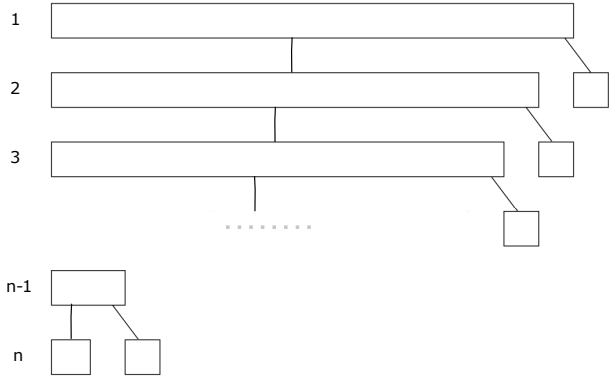
Quicksort: Analyse

Best-Case:

- Die beiden Teilfolgen haben immer (ungefähr) die gleiche Länge.
- Die Höhe des Aufrufbaumes ist $\Theta(\log n)$.
- Auf jeder Aufrufebene werden $\Theta(n)$ Vergleiche durchgeführt.
- Die Anzahl der Vergleiche und die Laufzeit liegen in $\Theta(n \log n)$.

Quicksort: Analyse

Worst-Case: Jede (Teil-)Folge wird immer beim letzten (oder immer beim ersten) Element geteilt.



Die Anzahl der Vergleiche ist $\Theta(n^2)$.

Quicksort: Analyse

Worst-Case: Mögliches Szenario:

- Aufsteigend sortierte Folge und es wird immer das hinterste Element als Pivotelement gewählt.
- Alle restlichen Elemente sind kleiner als das Pivotelement und daher wird die Rekursion nur für die linke Seite (alle restlichen Elemente außer dem Pivotelement) weitergeführt. Die rechte Seite ist leer (Terminierung der Rekursion).
- Die Größe der Teilfolge in der nächsten Rekursionsstufe verringert sich immer nur um 1!
- $\Theta(n)$ rekursive Aufrufe.
- Die Laufzeit liegt in $\Theta(n^2)$.
- Der zum Array zusätzlich benötigte Speicherplatz ist durch die $\Theta(n)$ rekursiven Aufrufe ebenfalls $\Theta(n)$.

Quicksort: Analyse

„Vermeiden“ des Worst-Case in der Praxis:

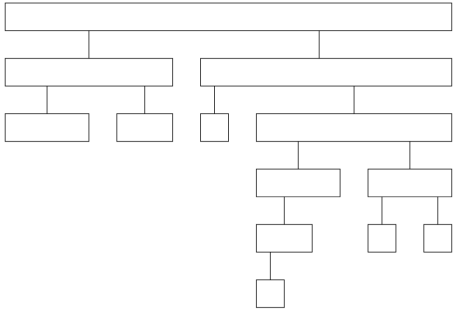
- Pivotelement immer zufällig wählen.
  - „Randomisierter Quicksort“.
  - Es ist nicht mehr die sortierte Folge das Worst-Case-Szenario.
- Betrachte jeweils das erste, letzte und mittlere Element (an der Postion $\lfloor \frac{n}{2} \rfloor$) und nimm den Median als Pivotelement.

Quicksort: Analyse

Average-Case:

- Komplizierter Beweis der zeigt, dass die Anzahl der Vergleiche und die Laufzeit dafür auch in $\Theta(n \log n)$ liegen.

Beispiel für Average-Case: Jede (Teil-)Liste wird nahe der Mitte geteilt.



Die Anzahl der Vergleiche ist $\Theta(n \log n)$.

Speicherplatzkomplexität

Speicherplatzkomplexität: Ist ein Maß für das Anwachsen des Speicherbedarfs eines Algorithmus in Abhängigkeit von der Eingabegröße.

Beispiele für Speicherplatzkomplexität:

- Quicksort: Worst-Case liegt in $\Theta(n)$, Best/Average-Case liegt in $\Theta(\log n)$
- Mergesort: Best/Average/Worst-Case liegt in $\Theta(n)$

Praxis: In der Praxis wird daher Quicksort sehr oft Mergesort vorgezogen.

Vergleich von Sortierverfahren

Tabelle: Laufzeit und Vergleiche.

Sortierverfahren	**Best-Case**	**Average-Case**	**Worst-Case**
Insertionsort	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Selectionsort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Mergesort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Quicksort	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n^2)$

Tabelle: Zusätzlicher Speicher.

Sortierverfahren	**Best-Case**	**Average-Case**	**Worst-Case**
Insertionsort	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Selectionsort	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Mergesort	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Quicksort	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$

Einsatz von Sortierverfahren

Quicksort:

- Wird sehr oft in allgemeinen Sortiersituationen bevorzugt.

Mergesort:

- Mergesort wird hauptsächlich für das Sortieren von Listen verwendet.
- Wird auch für das Sortieren von Dateien auf externen Speichermedien eingesetzt.
  - Dabei wird aber eine iterative Version von Mergesort (Bottom-up-Mergesort) verwendet, bei der nur $\log n$-mal eine Datei sequentiell durchgegangen wird.

Eine untere Schranke

Ausgangslage: Wir haben veschiedene Sortieralgorithmen kennengelernt. Die Worst-Case Laufzeit liegt im Bereich $O(n \log n)$ bis $O(n^2)$.

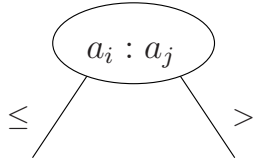
Frage: Geht es besser als in $O(n \log n)$ Zeit, z.B. $O(n)$?

Antwort: Wir zeigen für das allgemeine Sortierproblem unter der Annahme, dass n verschiedene Schlüssel sortiert werden müssen, dass $O(n \log n)$ optimal ist.

Entscheidungsbaum

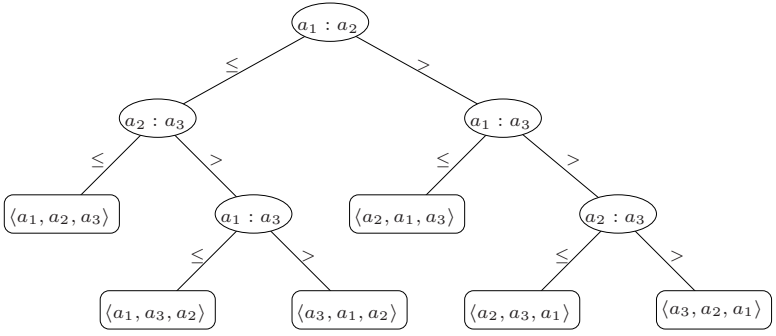
Entscheidungsbaum:

- Die Knoten entsprechen einem Vergleich von Elementen a_i und a_j.
- Die Blätter entsprechen den Permutationen.



Beispiel Entscheidungsbaum

Beispiel: Entscheidungsbaum des Insertionsort Algorithmus für $\langle a_1, a_2, a_3 \rangle$.



Beispiel Entscheidungsbaum

Anzahl der Schlüsselvergleiche: Die Anzahl der Schlüsselvergleiche im Worst-Case C_{worst} entspricht genau der Anzahl der Knoten auf dem längsten Pfad von der Wurzel bis zu einem Blatt minus 1.

Frage: Wie lautet die untere Schranke für die Höhe eines Entscheidungsbaums?

Satz: Jeder Entscheidungsbaum für die Sortierung von n paarweise verschiedenen Schlüsseln hat die Höhe $\Omega(n \log_2 n)$.

Entscheidungsbaum: Beweis

Beweis:

- Betrachte einen Entscheidungsbaum der Höhe h, der n unterschiedliche Schlüssel sortiert.
- Der Baum hat mindestens $n!$ Blätter.
- $n! \leq 2^h$, das impliziert $h \geq \log_2(n!)$.
- Hilfsrechnung:
  $n! \geq n \cdot (n-1) \cdot (n-2) \cdots \lceil \frac{n}{2} \rceil \geq (\lceil \frac{n}{2} \rceil)^{\lceil \frac{n}{2} \rceil} \geq \frac{n}{2}^{\frac{n}{2}}$
- $h \geq \log_2(n!) \geq \log_2 \left(\frac{n}{2}^{\frac{n}{2}} \right) \geq \frac{n}{2} \log_2 \frac{n}{2} = \frac{n}{2}(\log_2 n - 1) = \Omega(n \log_2 n)$

Untere Schranke für allgemeines Sortierproblem

Satz: Jedes allgemeine Sortierverfahren benötigt zum Sortieren von n paarweise verschiedenen Schlüsseln mindestens $\Omega(n \log n)$ Laufzeit im Worst-Case.

Folgerung: Mergesort ist ein asymptotisch zeitoptimaler Sortieralgorithmus. Es geht (asymptotisch) nicht besser!

Lineare Sortierverfahren

Beobachtung: Das Ergebnis für die untere Schranke basiert auf der Annahme, dass man keine Eigenschaften der zu sortierenden Elemente ausnutzt, sondern lediglich den Vergleichsoperator.

Praxis: Tatsächlich sind aber meist Wörter über ein bestimmtes Alphabet (d.h. einer bestimmten Definitionsmenge) gegeben, z.B.:

- Wörter über $\{a, b, \ldots, z, A, B, \ldots, Z\}$.
- Dezimalzahlen.
- Ganzzahlige Werte aus einem kleinen Bereich.

Lineare Sortierverfahren: Diese Information über die Art der Werte und ihren Wertebereich kann man zusätzlich ausnützen und dadurch im Idealfall Verfahren erhalten, die auch im Worst-Case in linearer Zeit sortieren.

Lineare Sortierverfahren: Beispiel Countsort

Eingabe: n Zahlen im Bereich 0 bis z im Array A, Hilfsarray Counts, $z < n$

```
for j ← 0 bis z
    Counts[j] ← 0
for i ← 0 bis n − 1
    Counts[A[i]] ← Counts[A[i]] + 1
i ← 0
for j ← 0 bis z
    for k ← 0 bis Counts[j] − 1
        A[i] ← j
        i ← i + 1
```

Komplexität: Erste Schleife in $\Theta(z)$, zweite Schleife in $\Theta(n)$, dritte (mit vierter) Schleife kann nur maximal n Zahlen bearbeiten und ist daher auch in $\Theta(n)$. Damit läuft der Algorithmus in $\Theta(z + n + n) = \Theta(n)$ (da $z = O(n)$).

Inversionen zählen

Inversionen zählen

Eine Musikwebsite versucht Übereinstimmungen im Musikgeschmack verschiedener Benutzer zu finden.

- Ein Benutzer bewertet n Songs indem er diese reiht.
- Die Musikseite überprüft in einer Datenbank, ob es weitere Benutzer mit einem ähnlichen Musikgeschmack gibt.

Ähnlichkeitsmaß: Anzahl der Inversionen zwischen zwei Rangfolgen.

- Rangfolge von Benutzer 1: $1, 2, \ldots, n$.
- Rangfolge von Benutzer 2: $a_1, a_2, \ldots, a_n$.
- Songs i und j sind invertiert, wenn $i < j$, aber $a_i > a_j$.

Inversionen zählen

Beispiel:

Songs

	A	B	C	D	E
Benutzer 1	1	2	3	4	5
Benutzer 2	1	3	4	2	5

Inversionen
3-2, 4-2

Brute-Force-Ansatz: Überprüfe alle $\Theta(n^2)$ Paare i und j.

Anwendungen

Anwendungen:

- Filtern von ähnlichen Präferenzen, wie z.B. ähnlichem Musikgeschmack (s.o.).
- Aggregieren von Rängen (z.B. für Metasuche im Web) durch finden einer Rangfolge, die eine minimale Anzahl an Inversionen im Vergleich zu den gegeben Rangfolgen hat.
- In der Statistik wird die Anzahl der Inversionen zwischen zwei Rangfolgen für den Tau-Test, einen nichtparametrischen Hypothesentest, benutzt.

Inversionen zählen: Divide-and-Conquer

Divide-and-Conquer:

1	5	4	8	10	2	6	9	12	11	3	7

Inversionen zählen: Divide-and-Conquer

Divide-and-Conquer:

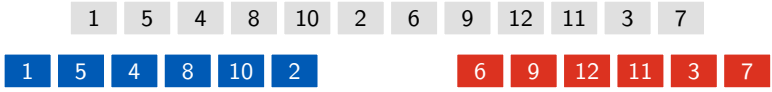
- Divide: Teile Liste in zwei Teile.

Divide: $O(1)$

Inversionen zählen: Divide-and-Conquer

Divide-and-Conquer:

- Divide: Teile Liste in zwei Teile.
- Conquer: Zähle rekursiv die Anzahl der Inversionen in jeder Hälfte.



Divide: $O(1)$

Conquer: $2T(n/2)$

5 blau-blau Inversionen

5-4, 5-2, 4-2, 8-2, 10-2

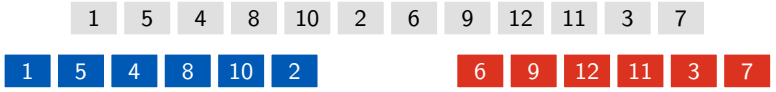
8 rot-rot Inversionen

6-3, 9-3, 9-7, 12-3,
12-7, 12-11, 11-3, 11-7

Inversionen zählen: Divide-and-Conquer

Divide-and-Conquer:

- Divide: Teile Liste in zwei Teile.
- Conquer: Zähle rekursiv die Anzahl der Inversionen in jeder Hälfte.
- Combine: Zähle Inversionen, wobei a_i und a_j sich in unterschiedlichen Hälften befinden, und retourniere die Summe von drei Größen.



Divide: $O(1)$

Conquer: $2T(n/2)$

5 blau-blau Inversionen

8 rot-rot Inversionen

9 blau-rot Inversionen

Combine: ???

5-3, 4-3, 8-6, 8-3, 8-7, 10-6, 10-9, 10-3, 10-7

Gesamt $= 5 + 8 + 9 = 22$.

Inversionen zählen: Kombinieren

Kombinieren: Zähle blau-rot Inversionen.

- Es wird angenommen, dass jede Hälfte sortiert ist.
- Zähle Inversionen, wobei a_i und a_j sich in unterschiedlichen Hälften befinden.
- Verschmelze zwei sortierte Hälften in eine sortierte Folge.
  - *Um die Invariante der Sortierung aufrechtzuerhalten*

9 blau-rot Inversionen: 4+2+2+1+0+0

Zählen: $O(n)$

Verschmelzen: $O(n)$

Inversionen zählen: Implementierung

Precondition: [Merge-and-Count] A und B sind sortiert. Postcondition: [Sort-and-Count] L ist sortiert.

```
Sort-and-Count(L):
if Liste L hat genau ein Element
    return 0 und die Liste L

Unterteile die Liste in zwei Hälften A und B
(r_A, A) ← Sort-and-Count(A)
(r_B, B) ← Sort-and-Count(B)
(r, L)   ← Merge-and-Count(A, B)

return r_A + r_B + r und die sortierte Liste L
```

Inversionen zählen: Implementierung

```
Merge-and-Count(A,B):
i ← 1, j ← 1
count ← 0
while beide Listen sind nicht leer
    if a_i ≤ b_j
        Füge a_i zur Ergebnisliste hinzu und erhöhe i
    else
        Füge b_j zur Ergebnisliste hinzu und erhöhe j
        count ← count + die Anzahl der restlichen Elemente in A
Füge den Rest der nicht leeren Liste zur Ergebnisliste hinzu
return count und sortierte Folge (Verschmelzung beider Hälften)
```

Inversionen zählen: Analyse

Verschmelzen: Merge-And-Count läuft in $O(n)$
(wie Merge-Schritt bei Mergesort).

Laufzeit: Die Laufzeit von Sort-and-Count lässt sich dann beschreiben mit:

$$T(n) \leq T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) \quad \Rightarrow \quad T(n) = O(n \log n)$$

Hinweis: Sort-and-Count ist nichts anderes als ein Mergesort, der noch zusätzlich Inversionen zählt.

Dichtestes Punktpaar (*Closest Pair of Points*)

Dichtestes Punktpaar

Dichtestes Paar: Gegeben seien n Punkte in der Ebene. Finde ein Paar mit der kleinsten euklidischen Distanz zwischen den beiden Punkten.

Fundamentale geometrische Formen:

- Grafik, maschinelles Sehen, geographische Informationssysteme, molekulare Modellierung, Flugsicherung.
- Spezialfall von nächster Nachbar, Euklidischer MST, Voronoi.

  Schnelle Algorithmen für das dichteste Punktpaar waren die Inspiration für schnelle Algorithmen für dieses Problem.

Brute-Force-Ansatz: Überprüfe alle Paare von Punkten p und q mit $\Theta(n^2)$ Vergleichen.

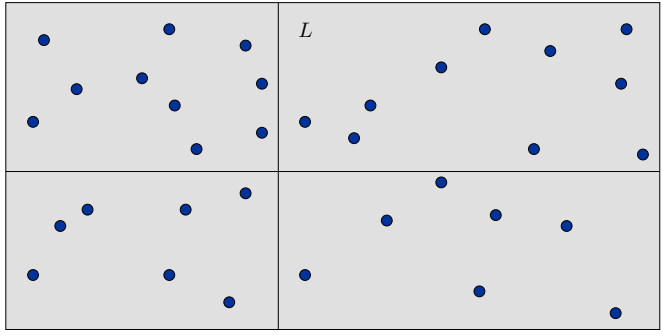
1-D-Version: $O(n \log n)$, wenn sich alle Punkte auf einer Linie befinden.

Annahme: Zwei Punkte haben nicht dieselbe x-Koordinate.

um Darstellung sauberer zu machen.

Dichtestes Punktpaar: Ein erster Versuch

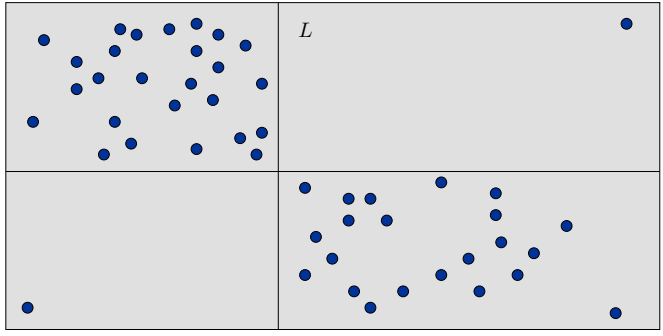
Teile: Teile die Region in 4 Quadranten.



Dichtestes Punktpaar: Ein erster Versuch

Teile: Teile die Region in 4 Quadranten.

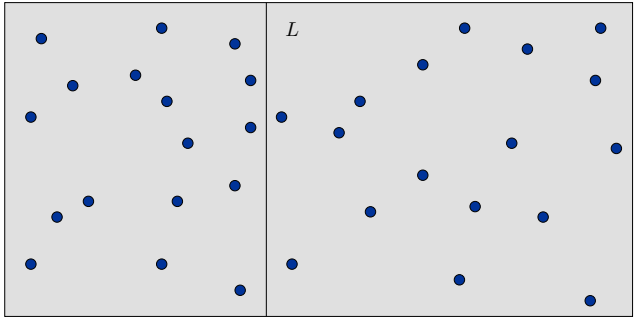
Problem: Es ist i.A. nicht möglich $n/4$ Punkte in jedem Quadranten sicherzustellen.



Dichtestes Punktpaar

Algorithmus:

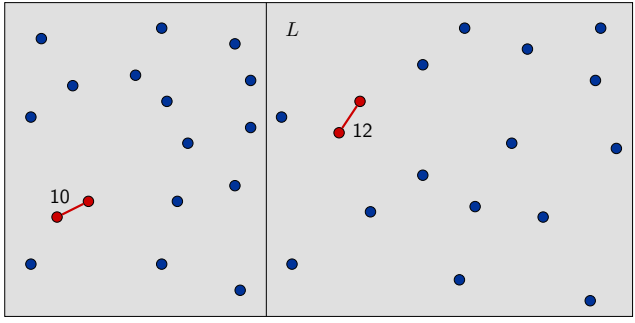
- Teile: Zeichne eine vertikale Linie L, sodass sich ungefähr $\frac{1}{2}n$ Punkte auf jeder Seite befinden.



Dichtestes Punktpaar

Algorithmus:

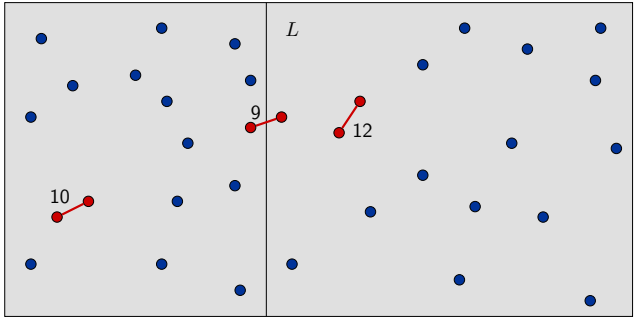
- Teile: Zeichne eine vertikale Linie L, sodass sich ungefähr $\frac{1}{2}n$ Punkte auf jeder Seite befinden.
- Herrsche: Finde ein dichtestes Punktpaar auf jeder Seite rekursiv.



Dichtestes Punktpaar

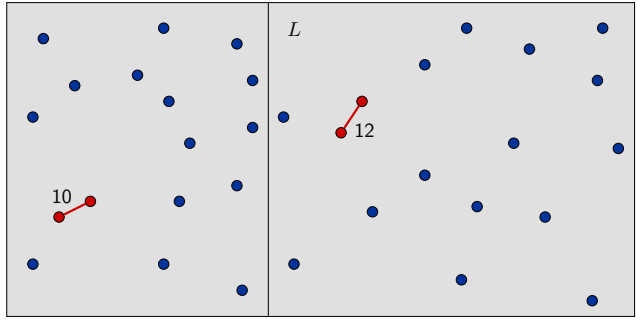
Algorithmus:

- Teile: Zeichne eine vertikale Linie L, sodass sich ungefähr $\frac{1}{2}n$ Punkte auf jeder Seite befinden.
- Herrsche: Finde ein dichtestes Punktpaar auf jeder Seite rekursiv.
- Kombiniere: Finde ein dichtestes Punktpaar mit einem Punkt auf jeder Seite (scheint in $\Theta(n^2)$ zu liegen).
- Retourniere die Beste von den drei Lösungen.



Dichtestes Punktpaar

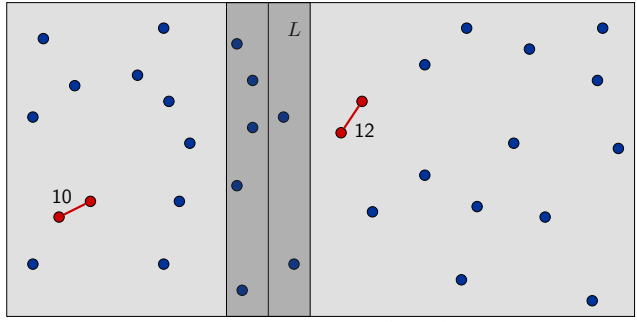
Aufgabe: Finde das dichteste Punktpaar mit einem Punkt auf jeder Seite
unter der Annahme, dass die Distanz $< \delta$ (= Minimum der minimalen Punktabstände in der linken und rechten Hälfte) ist.



Dichtestes Punktpaar

Aufgabe: Finde das dichteste Punktpaar mit einem Punkt auf jeder Seite
unter der Annahme, dass die Distanz $< \delta$ (= Minimum der minimalen Punktabstände in der linken und rechten Hälfte) ist.

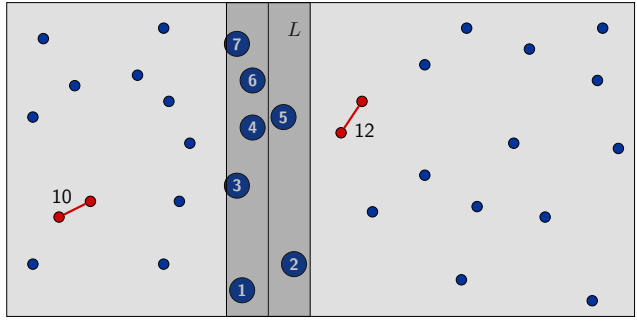
- Beobachtung: Man muss nur die Punkte mit einem Abstand kleiner δ von L berücksichtigen.



Dichtestes Punktpaar

Aufgabe: Finde das dichteste Punktpaar mit einem Punkt auf jeder Seite
unter der Annahme, dass die Distanz $< \delta$ (= Minimum der minimalen Punktabstände in der linken und rechten Hälfte) ist.

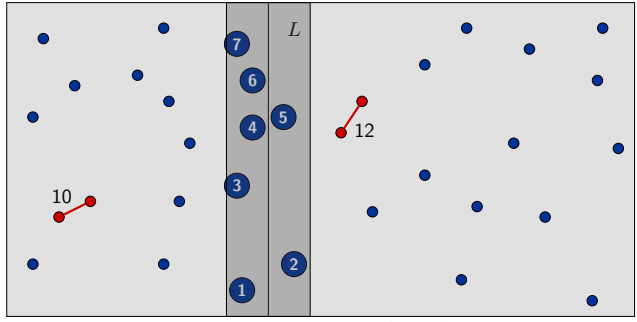
- Beobachtung: Man muss nur die Punkte mit einem Abstand kleiner δ von L berücksichtigen.
- Sortiere Punkte im 2δ-Streifen nach ihren y-Koordinaten.



Dichtestes Punktpaar

Aufgabe: Finde das dichteste Punktpaar mit einem Punkt auf jeder Seite
unter der Annahme, dass die Distanz $< \delta$ (= Minimum der minimalen Punktabstände in der linken und rechten Hälfte) ist.

- Beobachtung: Man muss nur die Punkte mit einem Abstand kleiner δ von L berücksichtigen.
- Sortiere Punkte im 2δ-Streifen nach ihren y-Koordinaten.
- Man muss nur die Distanzen von jedem dieser Punkte zu max. 11 in der sortierten Liste nachfolgenden Punkten überprüfen!



Dichtestes Punktpaar

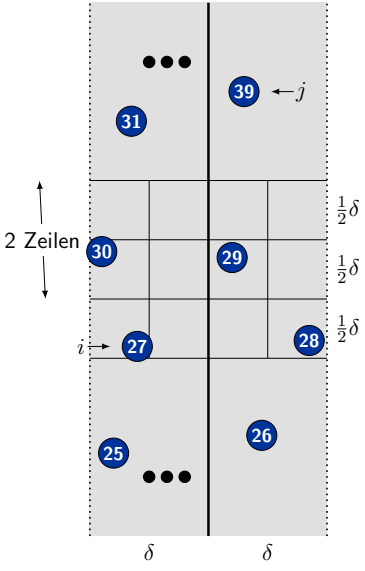
Definition: Sei s_i der Punkt im 2δ-Streifen mit der i-ten kleinsten y-Koordinate.

Behauptung: Wenn $|i - j| \geq 12$, dann ist die Distanz zwischen s_i und s_j zumindest δ.

Beweis:

- Keine zwei Punkte liegen im gleichen $\frac{1}{2}\delta$-mal-$\frac{1}{2}\delta$ Bereich.
- Zwei Punkte, die zumindest 2 ganze Zeilen von einander entfernt sind, haben eine Distanz von $\geq 2\left(\frac{1}{2}\delta\right)$. $\square$

Fakt (ohne Beweis): Noch immer wahr wenn wir 12 durch 7 ersetzen.



Dichtestes Punktpaar

Dichtestes Paar($p_1, \ldots, p_n$):

if $n = 1$ **return** ∞

Berechne Trennlinie L, sodass Punkte auf zwei Hälften aufgeteilt werden. $O(n \log n)$

$\delta_1 = $ Dichtestes Paar(linke Hälfte)
$\delta_2 = $ Dichtestes Paar(rechte Hälfte)
$\delta = \min(\delta_1, \delta_2)$ $2T(n/2)$

Lösche alle Punkte die weiter als δ von der Trennlinie L entfernt sind $O(n)$

Sortiere die restlichen Punkte nach y-Koordinate. $O(n \log n)$

Scanne die Punkte in y-Reihenfolge und vergleiche die Distanz zwischen jedem Punkt und den nächsten 11 Nachbarn. Wenn eine dieser Distanzen kleiner als δ ist, ändere δ $O(n)$

return δ.

Dichtestes Punktpaar: Analyse

Laufzeit:

$$T(n) \leq 2T(n/2) + O(n \log n) \quad \Rightarrow \quad T(n) = O(n \log^2 n)$$

Frage: Können wir $O(n \log n)$ erreichen?

Antwort: Ja. Die Punkte im Streifen müssen nicht immer sortiert werden.

- Jede Rekursion liefert zwei Listen zurück: Alle Punkte sortiert nach y-Koordinate, und alle Punkte sortiert nach x-Koordinate.
- Sortiere durch Verschmelzen zweier vorsortierter Listen.

$$T(n) \leq 2T(n/2) + O(n) \quad \Rightarrow \quad T(n) = O(n \log n)$$

Suchbäume

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 18. April 2023

Vorlesungsfolien

Wörterbuchproblem

Wörterbuch:

- Als Wörterbuch wird eine Menge von Elementen eines gegebenen Grundtyps bezeichnet, auf der man die Operationen Einfügen, Suchen und Entfernen ausführen kann.
- Für jedes Element wird ein Schlüssel k (key) und seine Nutzdaten gespeichert.
- Alle Elemente sind über den Schlüssel identifizierbar.
- Wir nehmen hier beispielhaft $k \in \mathbb{N}$ an.
- Die Operationen sind nur vom Schlüssel abhängig, sodass wir zur weiteren Vereinfachung annehmen, dass ein Wörterbuch aus einer Menge ganzzahliger Schlüssel besteht.

Wörterbuchproblem: Finde eine geeignete Datenstruktur zusammen mit möglichst effizienten Algorithmen zum Einfügen, Suchen und Entfernen von Schlüsseln.

Suchverfahren

Suchen:

- Ein wichtiges Thema der Algorithmentheorie.
- Das Suchen ist eine der grundlegendsten Operationen, die man immer wieder mit Computern ausführt:
  - Suchen von Datensätzen in Datenbanken
  - Suchen nach Wörtern in Wörterbüchern
  - Suchen von Stichwörtern im WWW

Array mit n Schlüsseln: In VU Programmkonstruktion besprochen.

- Lineare Suche in $O(n)$ Zeit.
- Binäre Suche in $O(\log n)$ Zeit.
- Einfügen in ein sortiertes Array in $O(n)$ Zeit.
- Entfernen aus einem sortierten Array in $O(n)$ Zeit.

Frage: Geht es effizienter?
Antwort: Ja, z.B. mit Suchbäumen.

Binäre Suchbäume

Suchbäume

Allgemein: Suchbäume existieren in unterschiedlichen Ausprägungen und bieten i.A. die Möglichkeit folgende Operationen effizient zu implementieren:

- Einfügen eines neuen Elements.
- Suche eines Elements.
- Entfernen eines Elements.
- Durchlaufen aller gespeicherten Elemente in geordneter Reihenfolge.
- Finden des kleinsten/größten Elements.
- Finden eines nächstkleineren/nächstgrößeren Elements.

Wurzelbäume

Baum:

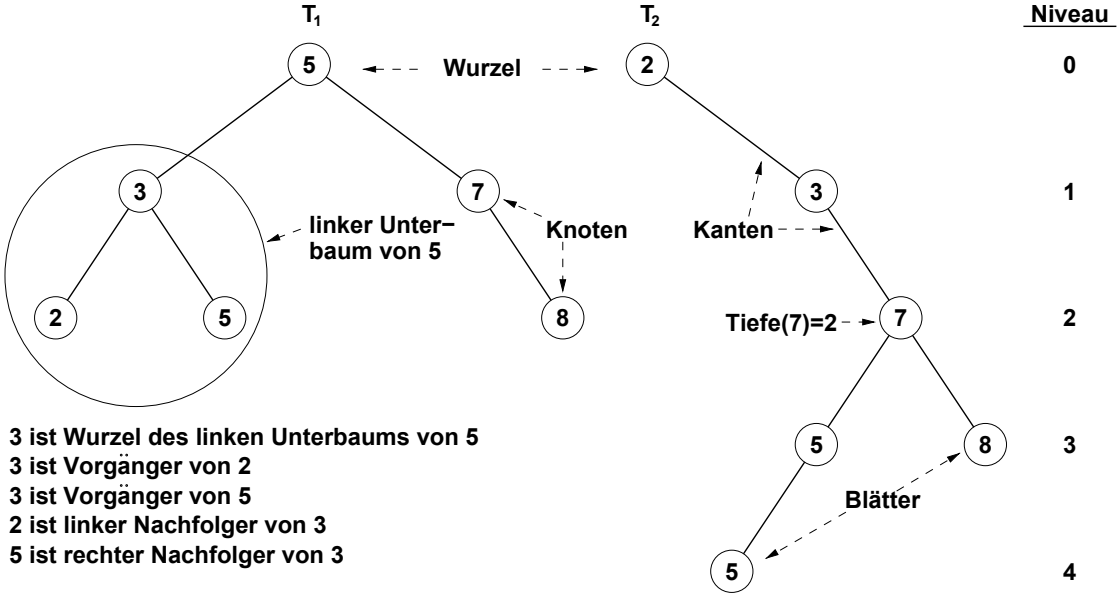
- Wir betrachten hier im Speziellen Wurzelbäume
  (siehe Kapitel über Graphen).
- Die Kinder (Nachfolger) eines jeden Knotens sind in einer bestimmten Reihenfolge gegeben.

Wurzel:

- Ist der einzige Knoten ohne Vorgänger.
- Alle anderen Knoten können genau über einen Pfad von der Wurzel erreicht werden.
- Jeder Knoten ist gleichzeitig die Wurzel eines Unterbaumes, der aus ihm selbst und seinen direkten und indirekten Nachfolgern besteht.

Binärer Baum: Jeder Knoten besitzt höchstens zwei Kinder,
ein „linkes“ und ein „rechtes“ Kind.

Binäre Bäume



3 ist Wurzel des linken Unterbaums von 5
3 ist Vorgänger von 2
3 ist Vorgänger von 5
2 ist linker Nachfolger von 3
5 ist rechter Nachfolger von 3

Wurzelbäume

Tiefe (Niveau) eines Knotens: Tiefe

- Die Tiefe eines Knotens ist die Anzahl der Kanten auf dem Pfad von der Wurzel zu diesem Knoten.
- Die Tiefe ist eindeutig, da nur ein solcher Pfad existiert.
- Die Tiefe der Wurzel ist 0.
- Die Menge aller Knoten mit gleicher Tiefe im Baum wird auch als Ebene oder Niveau bezeichnet.

Wurzelbäume: Höhe

Höhe eines (Teil-)baumes:

- Die Höhe $h(T_r)$ eines (Teil-)Baumes T_r mit dem Wurzelknoten r ist die Länge eines längsten Pfades in dem Teilbaum von r zu einem beliebigen Knoten v in T_r.
- Ein Baum mit nur einem (Wurzel-)Knoten hat die Höhe 0.
- Die Höhe des leeren Baumes definieren wir mit -1.

Blatt:

- Ein Knoten heißt Blatt, wenn er keine Kinder besitzt.
- Alle anderen Knoten nennt man innere Knoten.

Implementierung von binären Bäumen

Knoten x:

$x.key$	Schlüssel von x
$x.left$	Verweis auf linkes Kind ($null$ wenn nicht vorhanden)
$x.right$	Verweis auf rechtes Kind ($null$ wenn nicht vorhanden)
$x.info$	Nutzdaten im Knoten (hier als optional betrachtet)
$x.parent$	Verweis auf Vorgänger von x (optional; $null$ wenn Wurzel)

- Der Zugriff auf den Baum erfolgt über einen Verweis auf den Wurzelknoten, z.B. $root$.
- Ist der Baum leer, so gilt $root = null$.

Durchmusterungen (Traversals): Inorder

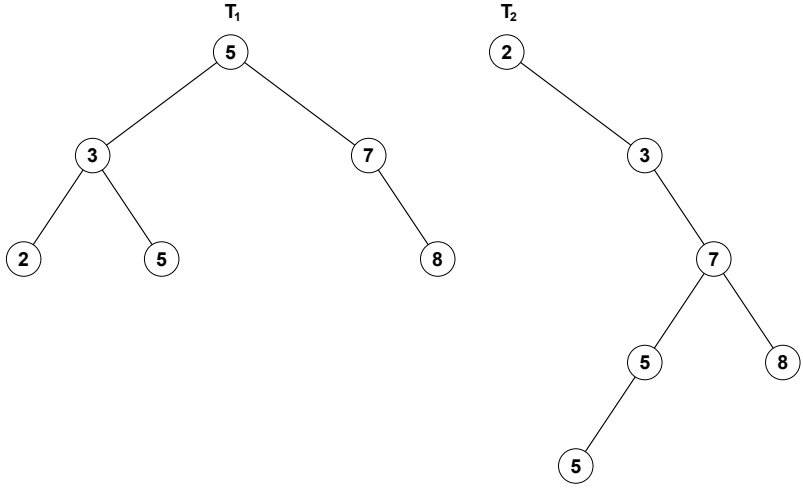
Inorder-Durchmusterung: Behandle rekursiv zunächst den linken Unterbaum, dann die Wurzel, dann den rechten Unterbaum.

Aufruf: Inorder($root$).

```
Inorder(p)
if p ≠ null
    Inorder(p.left)
    Bearbeite p (z.B. Ausgabe)
    Inorder(p.right)
```

Laufzeit: Die Laufzeit liegt für $n \geq 1$ Knoten in $\Theta(n)$, da für jeden Knoten genau ein rekursiver Aufruf erfolgt, maximal $2n$ zusätzliche Aufrufe für leere Unterbäume hinzukommen, und jeder Aufruf ohne Folgeaufrufe eine Laufzeit von $\Theta(1)$ hat.

Inorder: Beispiel

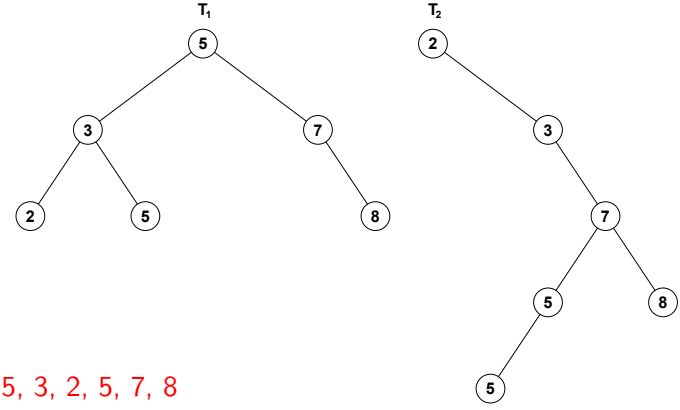


Für beide gezeigten Bäume ist die Inorder-Durchmusterungsreihenfolge 2, 3, 5, 5, 7, 8.

Durchmusterungen: Preorder

Preorder-Durchmusterung: Behandle rekursiv zunächst die Wurzel, dann den linken Unterbaum, danach den rechten Unterbaum.

Beispiel:

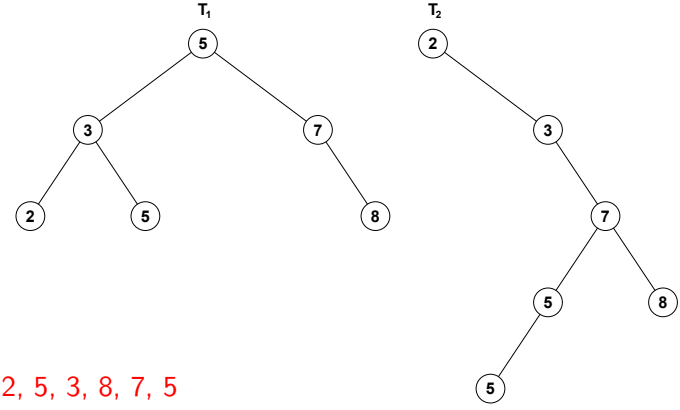


- Für T_1: 5, 3, 2, 5, 7, 8
- Für T_2: 2, 3, 7, 5, 5, 8

Durchmusterungen: Postorder

Postorder-Durchmusterung: Behandle rekursiv zunächst den linken Unterbaum, dann den rechten Unterbaum, danach die Wurzel.

Beispiel:



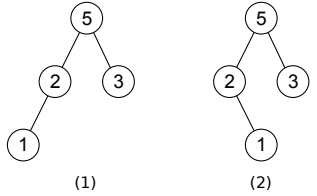
- Für T_1: 2, 5, 3, 8, 7, 5
- Für T_2: 5, 5, 8, 7, 3, 2

Durchmusterungen: Theorem

Theorem (ohne Beweis): Ein binärer Baum kann immer eindeutig aus der Inorder- und Preorder-Durchmusterungsfolge der Elemente rekonstruiert werden, wenn die Elemente paarweise unterschiedlich sind. Gleiches gilt für Inorder zusammen mit Postorder, nicht jedoch für Preorder und Postorder.

Beispiel:

- Inorder für (1): 1, 2, 5, 3
- Inorder für (2): 2, 1, 5, 3
- Preorder für beide: 5, 2, 1, 3
- Postorder für beide : 1, 2, 3, 5



Binäre Suchbäume

Ein binärer Suchbaum ist ein binärer Baum der in jedem Knoten x folgende binäre Suchbaumeigenschaft erfüllt:

- Ist y ein Knoten des linken Unterbaumes von x so gilt:

$$y.key \leq x.key.$$

- Ist z ein Knoten des rechten Unterbaumes von x so gilt:

$$z.key \geq x.key.$$

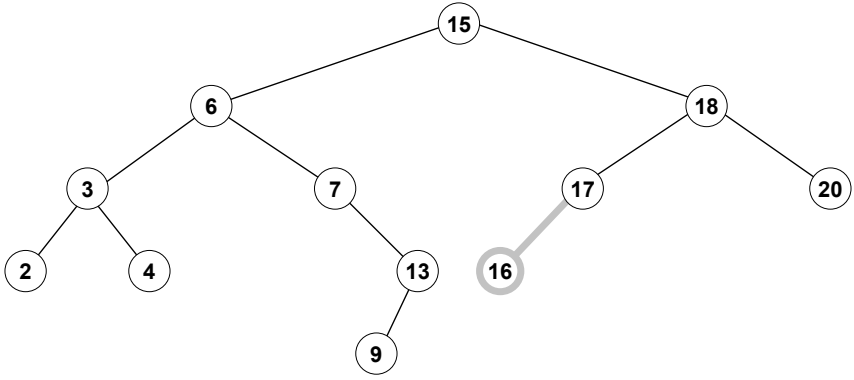
Hinweis: In einer konkreten Implementierung muss man sich entscheiden, ob man gleiche Schlüssel immer links oder immer rechts speichert.

Operationen auf binären Suchbäumen

Typische Operationen:

- Einfügen / Entfernen eines Elements
- Suchen nach einem Element
- Minimum / Maximum: kleinstes / größtes Element finden
- Predecessor / Successor:
  voriges / nächstes Element entsprechend der Sortierreihenfolge finden
- Durchmusterung

Operationen auf binären Suchbäumen



Minimum(7) = 7
Minimum(15) = 2

Maximum(15) = 20
Maximum(6) = 13

Predecessor(15) = 13
Predecessor(9) = 7

Successor(15) = 17
Successor(13) = 15

Insert(16)

Suchen

Eingabe: Baum mit Wurzel p und gesuchter Schlüssel s

Rückgabewert: Knoten mit Schlüssel s oder $null$, falls s nicht vorhanden ist.

```
Search(p,s):
while p ≠ null und p.key ≠ s
    if s < p.key
        p ← p.left
    else
        p ← p.right
return p
```

Minimum

Eingabe: Baum mit Wurzel p.

Rückgabewert: Knoten mit dem kleinsten Schlüssel.

```
Minimum(p):
if p = null
    return null
while p.left ≠ null
    p ← p.left
return p
```

Vorgehen: Solange beim linken Kind weitergehen, bis es keinen linken Nachfolger mehr gibt.

Maximum

Eingabe: Baum mit Wurzel p.

Rückgabewert: Knoten mit dem größten Schlüssel.

```
Maximum(p):
if p = null
    return null
while p.right ≠ null
    p ← p.right
return p
```

Vorgehen: Solange beim rechten Kind weitergehen, bis es keinen rechten Nachfolger mehr gibt.

Successor

Eingabe: Knoten p in Baum.

Rückgabewert: Nächster Knoten entsprechend der Inorder-Durchmusterungsreihenfolge oder $null$.

```
Successor(p):
if p.right ≠ null
    return Minimum(p.right)
else
    q ← p.parent
    while q ≠ null und p = q.right
        p ← q
        q ← q.parent
    return q
```

Predecessor

Eingabe: Knoten p in Baum.

Rückgabewert: Vorhergehender Knoten entsprechend der Inorder-Durchmusterungsreihenfolge oder $null$.

```
Predecessor(p):
if p.left ≠ null
    return Maximum(p.left)
else
    q ← p.parent
    while q ≠ null und p = q.left
        p ← q
        q ← q.parent
    return q
```

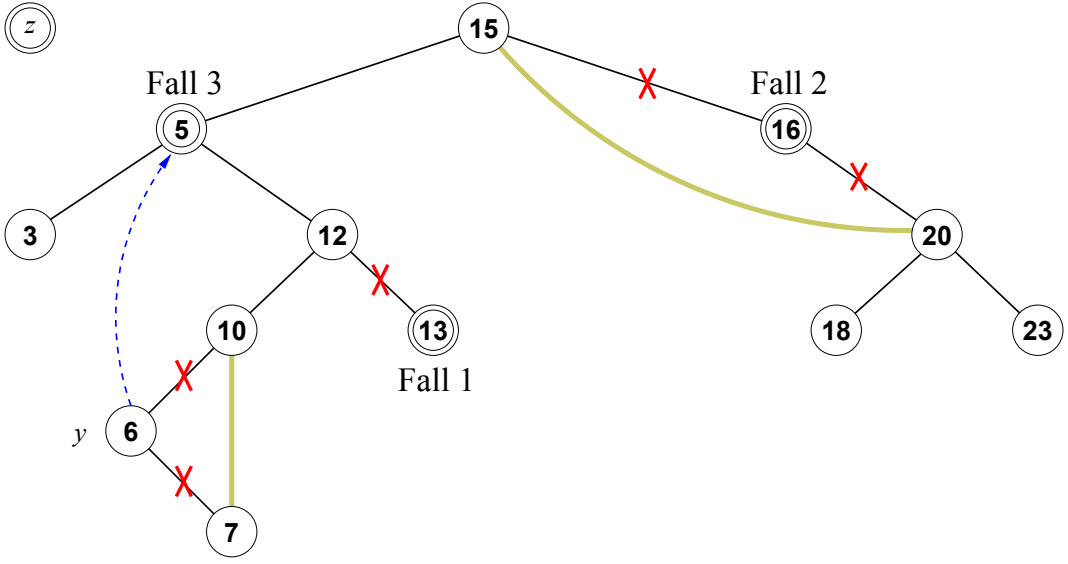
Einfügen

Eingabe: Baum mit Wurzel $root$ und ein neuer Knoten q.
Hinweis: $root$ wird bei leerem Baum verändert (Referenzparameter)

```
Insert(root, q):
r ← null, p ← root
while p ≠ null
    r ← p
    if q.key < p.key
        p ← p.left
    else
        p ← p.right
q.parent ← r, q.left ← null, q.right ← null
if r = null
    root ← q
else
    if q.key < r.key
        r.left ← q
    else
        r.right ← q
```

Entfernen eines Knotens

Entfernen: Knoten z soll entfernt werden. Dabei müssen drei Fälle unterschieden werden:



Entfernen eines Knotens

Fall 1: z hat keine Kinder (z.B. Knoten 13). In diesem Fall kann z einfach entfernt werden.

Fall 2: z hat ein Kind (z.B. Knoten 16). Dieser Fall entspricht dem Entfernen aus einer verketteten linearen Liste.

Fall 3: z hat zwei Kinder (z.B. Knoten 5). Wir bringen an die Stelle des zu löschenden Knotens einen Ersatzknoten und löschen den Ersatzknoten an seiner ursprünglichen Position.
Geeigneter Ersatzknoten für z:

- Der Knoten mit dem größten Schlüssel des linken Unterbaumes (Predecessor) oder
- der Knoten mit dem kleinsten Schlüssel des rechten Unterbaumes (Successor)

Hinweis: Der Ersatzknoten hat in seiner ursprünglichen Position immer maximal einen Nachfolger und wird schließlich gemäß Fall 1 oder 2 entfernt.

Entfernen eines Knotens

Eingabe: Baum mit Wurzel $root$ und ein Knoten q.
Hinweis: $root$ kann verändert werden (Referenzparameter)

```
Remove(root, q):
if q.left = null oder q.right = null
    r ← q
else
    r ← Successor(q), q.key ← r.key, q.info ← r.info
if r.left ≠ null
    p ← r.left
else
    p ← r.right
if p ≠ null
    p.parent ← r.parent
if r.parent = null
    root ← p
else
    if r = r.parent.left
        r.parent.left ← p
    else
        r.parent.right ← p
```

Laufzeiten

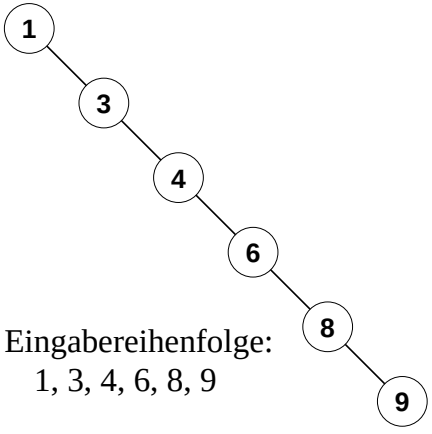
Worst-Case:

- Der Zeitaufwand für Suchen, Einfügen, Entfernen, Minimum, Maximum, Predecessor und Successor in einem binären Suchbaum mit der Höhe h liegt in $O(h)$.
- In jeder Operation muss man ungünstigstenfalls einem Pfad von der Wurzel zu einem tiefsten Blatt folgen.

Strukturen von Suchbäumen

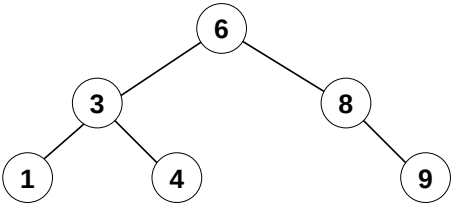
Die Struktur eines binären Suchbaums hängt von der Eingabereihenfolge ab.

T_1 (Höhe=5)



Eingabereihenfolge:
1, 3, 4, 6, 8, 9

T_2 (Höhe=2)



Eingabereihenfolge, z.B.:
6, 8, 3, 4, 1, 9

Einen binären Suchbaum aus einer sortierten Eingabereihenfolge aufzubauen führt zur Entartung in eine lineare Liste!

Laufzeiten

Vollständig balancierter Baum:

- Alle inneren Knoten bis auf jene in der vorletzten Ebene haben 2 Nachfolger.
- Hat die Höhe $h(T) = \lfloor \log_2 n \rfloor$ und daher benötigen alle Operationen bis auf das Durchmustern $O(\log n)$ Zeit.

Zu einer Liste entarteter Baum:

- Ist der Baum jedoch entartet und hat eine Höhe $h(T) = O(n)$, dann benötigen alle Operationen $O(n)$ Zeit!

Laufzeiten

Average-Case:

- Die erwartete durchschnittliche Suchpfadlänge (über alle mögliche binären Suchbäume für n unterschiedliche Elemente) ist für große n nur ca. 40% länger als im Idealfall, d.h in $O(\log n)$.

  Für einen ausführlichen Beweis siehe Seite 277ff in:
  T. Ottmann und P. Widmayer: Algorithmen und Datenstrukturen, 5. Auflage, Spektrum Akademischer Verlag, 2012

Hinreichend balancierter Baum: Für ein garantiertes logarithmisches Zeitverhalten genügen auch hinreichend balancierte Bäume, die wir im nächsten Abschnitt besprechen werden.

Die bisher betrachteten Suchbäume werden im Gegensatz zu den folgenden Bäumen, bei denen wir die Struktur speziell beeinflussen, auch konkreter natürliche Suchbäume genannt.

Balancierte Bäume

Idee: Suchbaum wird durch geeignete Randbedingungen und entsprechende Umordnungsoperationen effizient ausbalanciert, um zu garantieren, dass seine Höhe logarithmisch bleibt.

Möglichkeiten:

- Höhenbalancierte Bäume: Die Höhe der Unterbäume eines jeden Knotens unterscheidet sich jeweils um höchstens eine Konstante voneinander.
- Gewichtsbalancierte Bäume: Die Anzahl der Knoten in den Unterbäumen jedes Knotens unterscheidet sich höchstens um einen konstanten Faktor.
- (a,b)-Bäume ($2 \leq a \leq b$): Jeder Knoten (außer der Wurzel) hat zwischen a und b Kinder und alle Blätter haben den gleichen Abstand zur Wurzel; z.B. $a = 2$, $b = 4$.

Wir betrachten in dieser Vorlesung ein Beispiel für höhenbalancierte Bäume (AVL-Bäume) und eines für (a,b)-Bäume (B-Bäume).

AVL-Bäume (Adelson-Velski/Landis-Bäume)

AVL-Bäume

Geschichte: Der historisch erste Vorschlag aus 1962 sind AVL-Bäume, die auf Adelson-Velski und Landis zurückgehen.

Generelle Idee: Durch eine Forderung an die Höhendifferenz der beiden Teilbäume eines jeden Knotens wird ein Degenerieren von Suchbäumen verhindert.

Kritische Operationen: Nur Einfügen und Löschen sind kritische Operationen und erfordern eine speziellere Behandlung. Alle anderen Operationen, die den Baum nicht verändern (Suchen, ...), funktionieren wie bisher.

AVL-Bäume

Balance: Balance eines Knotens v: $bal(v) = h_2 - h_1$

- h_1 ... Höhe des linken Unterbaumes von v
- h_2 ... Höhe des rechten Unterbaumes von v

Balancierter Knoten: Ein Knoten v heißt balanciert, wenn $bal(v) \in \{-1, 0, 1\}$.

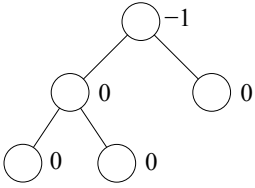
AVL-Bedingung:

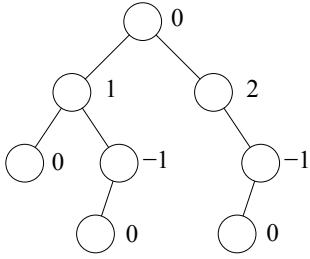
- Ein AVL-Baum ist ein binärer Suchbaum, in dem alle Knoten balanciert sind.
- Für jeden Knoten v gilt: Die Höhe des linken Teilbaums unterscheidet sich von der Höhe des rechten Teilbaums um höchstens 1.

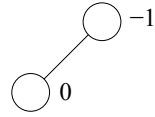
Hinweis: Die Höhe eines leeren Baumes haben wir mit -1 definiert.

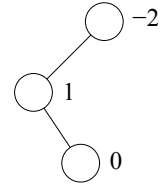
Balance von Knoten

Balance: Beispiele für die Balance von Knoten in binären Bäumen.









AVL-Bäume: Grundlegende Idee

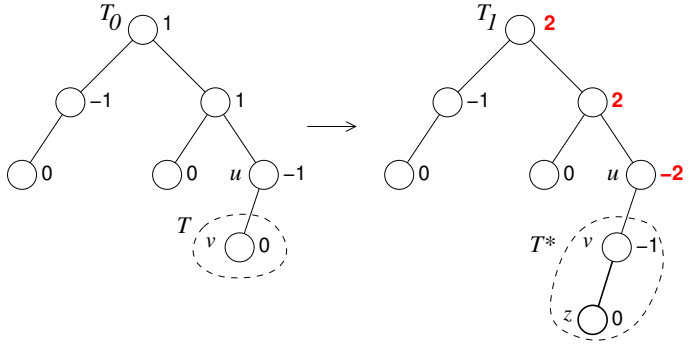
Grundlegende Idee zum Einfügen und Entfernen:

- Führe das Einfügen und Entfernen wie bisher aus.
- Überprüfe danach die Balance in möglicherweise betroffenen Knoten und führe gegebenenfalls eine Rebalancierung
  (lokale Umordnung) durch, um die Balance in allen Knoten wieder herzustellen.

AVL-Ersetzung

Definition: Eine AVL-Ersetzung

- ist eine Operation (z.B. Einfügen, Löschen),
- die einen Unterbaum T eines Knotens
- durch einen modifizierten (gültigen) AVL-Baum T^* ersetzt,
- dessen Höhe um höchstens 1 von der Höhe von T abweicht.

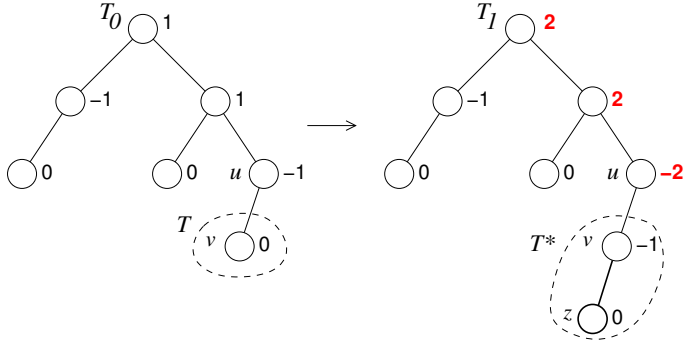


Das Beispiel zeigt, dass es in darüberliegenden Knoten zur Verletzung der Balance kommen kann.

Rebalancierung: Grundlagen

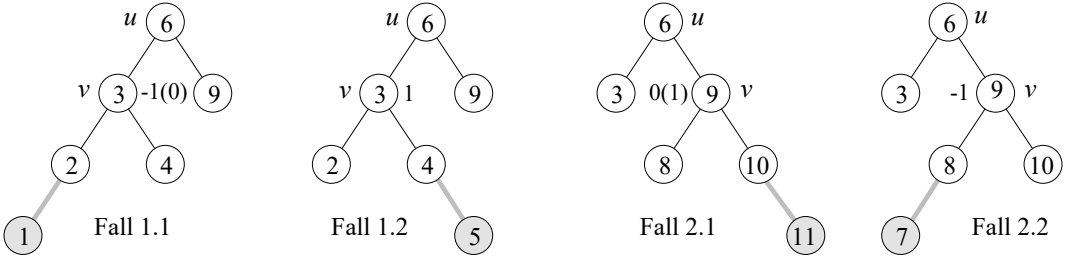
Definitionen:

- Sei T_0 ein gültiger AVL-Baum vor der AVL-Ersetzung und T_1 der unbalancierte Baum hinterher.
- Sei u der unbalancierte Knoten ($bal(u) \in \{-2, +2\}$) maximaler Tiefe. An diesem wird mit der Rebalancierung gestartet.



Rebalancierung: Überblick

Balancierung: Vier Beispiele für AVL-Bäume, die durch das Einfügen eines Knotens aus der Balance geraten sind.



Wenn $bal(u) = -2$ und v ist das linke Kind von u und

- $bal(v) \in \{-1, 0\} \rightarrow$ Fall 1.1
- $bal(v) = 1 \rightarrow$ Fall 1.2

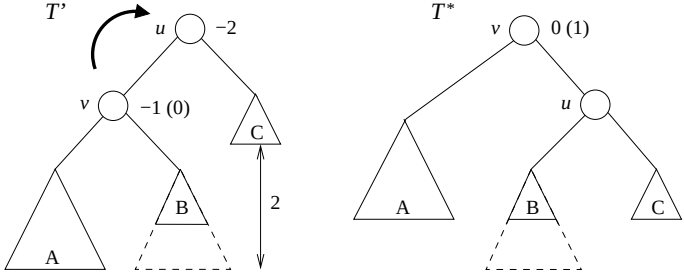
Wenn $bal(u) = 2$ und v ist das rechte Kind von u und

- $bal(v) \in \{0, 1\} \rightarrow$ Fall 2.1
- $bal(v) = -1 \rightarrow$ Fall 2.2

Rebalancierung: Fall 1.1

Fall 1.1: Sei $bal(u) = -2$. Sei v das linke Kind von u und $bal(v) \in \{-1, 0\}$.

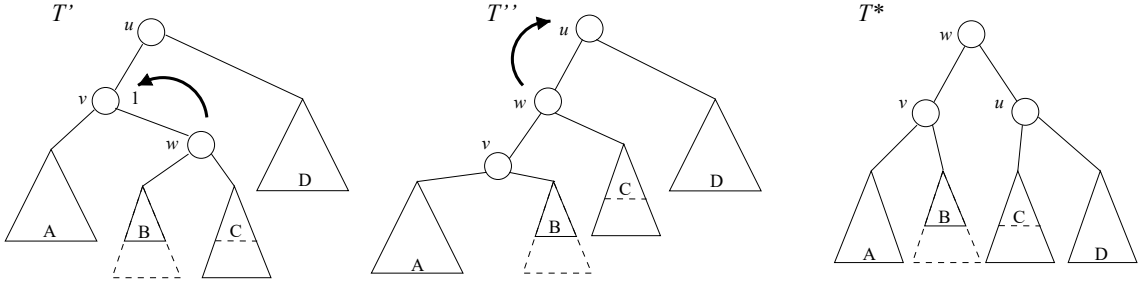
- Der linke Unterbaum des linken Kindes von u ist höher als oder gleich hoch wie der rechte.
- Rebalancierung von u durch einfache Rotation nach rechts an u, d.h. u wird rechtes Kind von v.
- Das rechte Kind von v wird als linkes Kind an u abgegeben.



Rebalancierung: Fall 1.2

Fall 1.2: Sei $bal(u) = -2$. Sei v das linke Kind von u und $bal(v) = 1$.

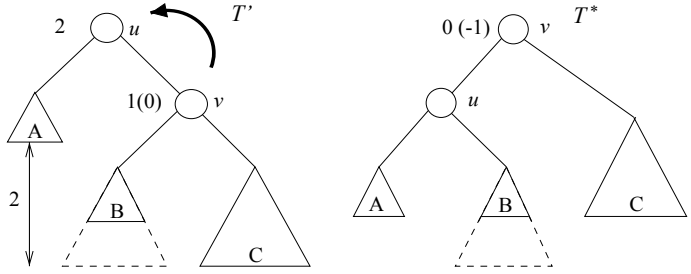
- Der rechte Unterbaum des linken Kindes von u ist höher als der linke.
- Dann existiert das rechte Kind w von v.
- Rebalancierung durch eine Rotation nach links an v und eine anschließende Rotation nach rechts an u
  (Doppelrotation links-rechts).



Rebalancierung: Fall 2.1

Fall 2.1: Sei $bal(u) = 2$. Sei v das rechte Kind von u und $bal(v) \in \{0, 1\}$.

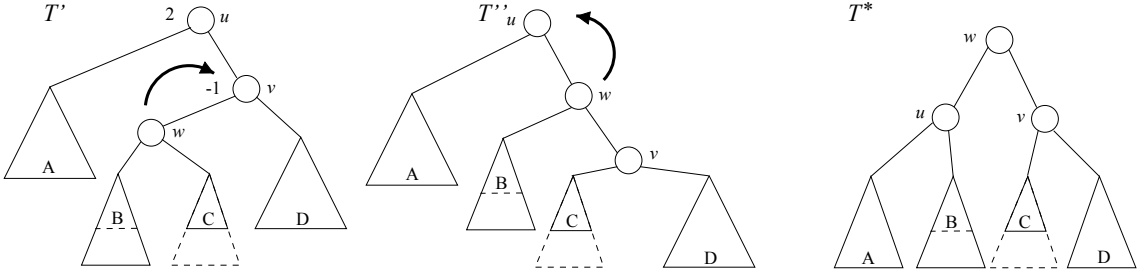
- Der rechte Unterbaum des rechten Kindes von u ist höher als oder gleich hoch wie der linke.
- Rebalancierung von u durch einfache Rotation nach links an u, d.h. u wird linkes Kind von v.
- Das linke Kind von v wird als rechtes Kind an u abgegeben.



Rebalancierung: Fall 2.2

Fall 2.2: Sei $bal(u) = 2$. Sei v das rechte Kind von u und $bal(v) = -1$.

- Der linke Unterbaum des rechten Kindes von u ist höher als der rechte.
- Dann existiert das linke Kind w von v.
- Rebalancierung durch eine Rotation nach rechts an v und eine anschließende Rotation nach links an u
  (Doppelrotation rechts-links).



Höhe

Implementierung: Wir ergänzen alle Knoten u um ein Attribut $u.height$, das jeweils die Höhe des (Unter-)baumes mit der Wurzel u angibt.

Hilfsfunktion: Wir definieren folgende Hilfsfunktion zur Ermittlung der Höhe eines Baumes:

- Eingabe: Teilbaum mit Wurzel u.
- Rückgabewert: Höhe des Teilbaums.

```
Height(u):
if u = null
    return −1
else
    return u.height
```

Einfache Rotation nach rechts

Eingabe: Teilbaum mit Wurzel u.

Rückgabewert: Neue Wurzel v des rebalancierten Teilbaums.

```
RotateToRight(u):
v ← u.left
u.left ← v.right
v.right ← u
u.height ← max(Height(u.left), Height(u.right)) + 1
v.height ← max(Height(v.left), Height(u)) + 1;
return v
```

Doppelrotation links-rechts

Eingabe: Teilbaum mit Wurzel u.

Rückgabewert: Neue Wurzel des rebalancierten Teilbaums.

```
DoubleRotateLeftRight(u):
u.left ← RotateToLeft(u.left)
return RotateToRight(u)
```

Einfügen in einen AVL-Baum

Eingabe: Wurzel p (wird mögl. geändert), neuer Knoten q.
Rückgabewert: Höhe des Teilbaums.

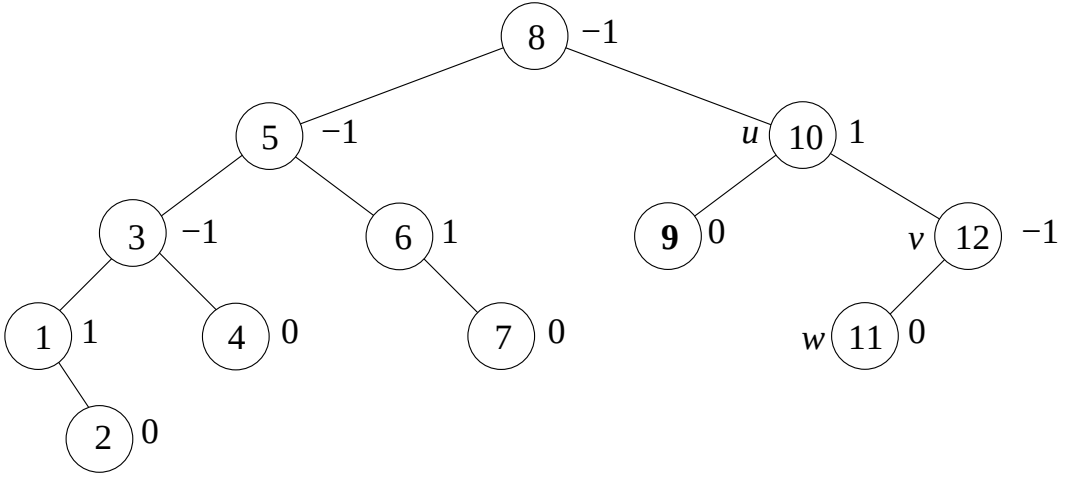
```
Insert(p, q):
if p = null
    p ← q, q.left ← q.right ← null, q.height ← 0
else
    if q.key < p.key
        Insert(p.left, q)
        if Height(p.right) − Height(p.left) = −2
            if Height(p.left.left) ≥ Height(p.left.right)
                p ← RotateToRight(p)
            else
                p ← DoubleRotateLeftRight(p)
    elseif q.key > p.key
        ....
    else
        Knoten q schon vorhanden
p.height ← max(Height(p.left),Height(p.right)) + 1
```

Einfügen in einen AVL-Baum

```
Insert(p, q):
if p = null
    p ← q, q.left ← q.right ← null, q.height ← 0
else
    if q.key < p.key
        ...
    elseif q.key > p.key
        Insert(p.right, q)
        if Height(p.right) − Height(p.left) = 2
            if Height(p.right.right) ≥ Height(p.right.left)
                p ← RotateToLeft(p)
            else
                p ← DoubleRotateRightLeft(p)
    else
        Knoten q schon vorhanden
p.height ← max(Height(p.left),Height(p.right)) + 1
```

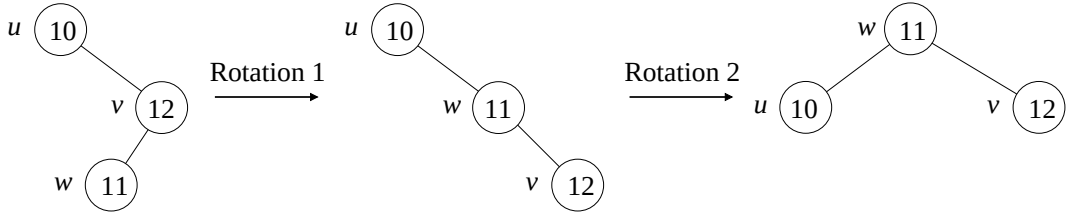
Beispiel: Entfernen eines Elementes aus AVL-Baum

Ausgangssituation: 9 soll entfernt werden.



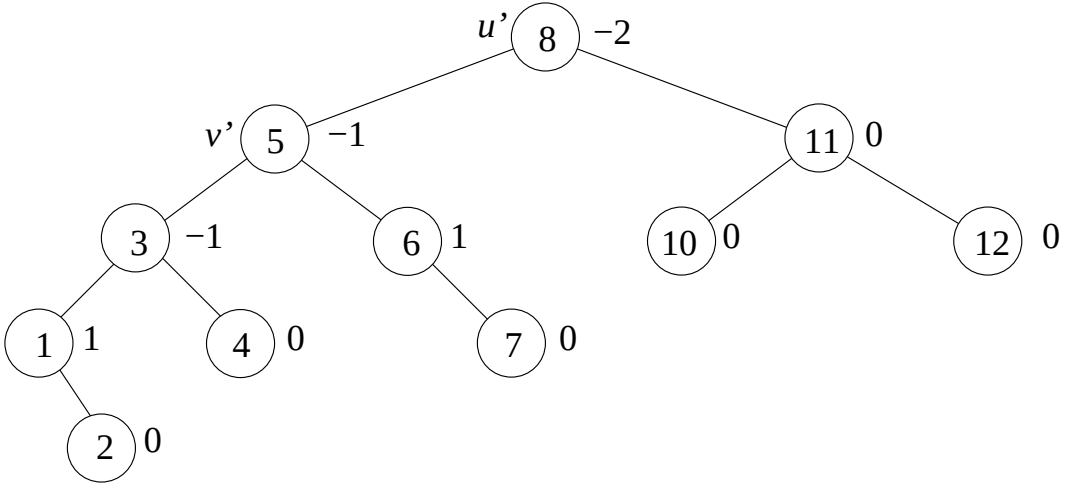
Beispiel: Entfernen eines Elementes aus AVL-Baum

Entfernen: Erfordert Doppelrotation rechts-links.



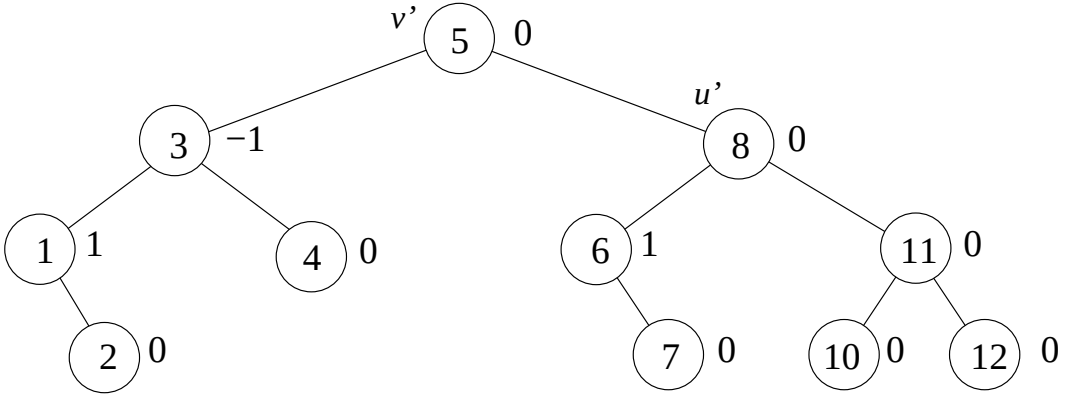
Beispiel: Entfernen eines Elementes aus AVL-Baum

Ergebnis: 9 wurde entfernt, Unterbaum rotiert, Höhe der Wurzel hat sich verändert.



Entfernen eines Elementes aus einem AVL-Baum

Abschluss: Einfache Rotation nach rechts über die Wurzel.



Analyse von AVL-Bäumen

Theorem: Die Höhenbedingung eines AVL-Baumes stellt sicher, dass die Höhe eines AVL-Baums mit $n \geq 1$ Knoten durch $O(\log n)$ beschränkt ist.

Beweis: Einen ausführlichen Beweis findet man ab Seite 284ff in:
T. Ottmann und P. Widmayer: Algorithmen und Datenstrukturen, 5. Auflage, Spektrum Akademischer Verlag, 2012

Analyse von AVL-Bäumen

Rotieren: Der Zeitaufwand zum Ausführen einer einzelnen Rotation oder Doppelrotation ist konstant.

Worst-Case:

- Beim Einfügen gibt es maximal eine (Doppel-)Rotation.
- Beim Entfernen werden im Worst-Case auf dem Suchpfad von der betroffenen Stelle weg bis zur Wurzel Rotationen bzw. Doppelrotationen durchgeführt.

Aufwand: Die Laufzeit der Operationen Einfügen und Entfernen liegt daher immer in $O(\log n)$.

B-Bäume und B*-Bäume

B-Baum: Motivation

Bisherige Suchbäume: Gut geeignet, wenn sich die Daten im Hauptspeicher befinden (internes Suchen).

Große Datenmengen:

- Bei sehr großen Datenmengen (z.B. Datenbanken) werden die Daten auf Festplatten oder anderen externen Speichern abgelegt.
- Bei Bedarf werden Teile davon in den Hauptspeicher geladen.
- Ein Zugriff auf den externen Speicher benötigt deutlich mehr Zeit als ein Zugriff auf den Hauptspeicher.

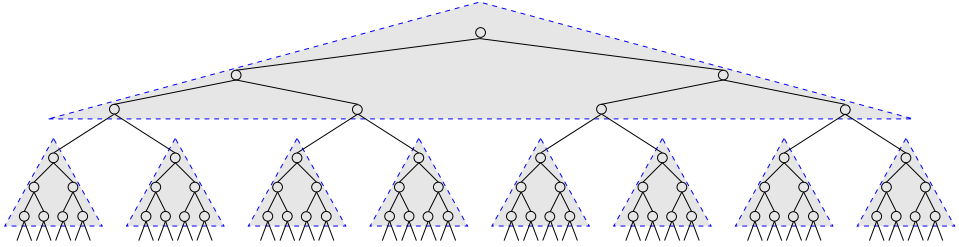
Beispiel: Binärer Baum mit N Datensätzen, extern gespeichert.

- Verweise auf die linken und rechten Unterbäume sind jeweils Adressen auf einem externen Speichermedium.
- Suche nach einem Schlüssel benötigt daher ungefähr $\log_2 n$ externe Zugriffe.
- Bei $N = 10^6$ sind das ca. 20 externe Speicherzugriffe.

B-Baum: Motivation

Annahme: Externe Speichermedien weisen i.A. eine blockorientierte Struktur auf. Bei einem externen Speicherzugriff wird immer eine gesamte Speicherseite (*page, block*) gelesen oder geschrieben.

Grundsätzliche Idee: „Zusammenfassen mehrerer Knoten“, sodass jeweils eine Speicherseite bestmöglich genutzt wird.



Vorteil: Weniger Zugriffe auf Speicherseiten notwendig.

Beispiel:

- Baum mit $N = 10^6$ Schlüssel, 128 Schlüssel pro Speicherseite.
- Man benötigt nur 3 externe Speicherzugriffe.

B-Baum: Definition

B-Bäume: Sie sind eine Verallgemeinerung von binären Suchbäumen und setzen die Idee der Gruppierung in Speicherseiten in die Praxis um.

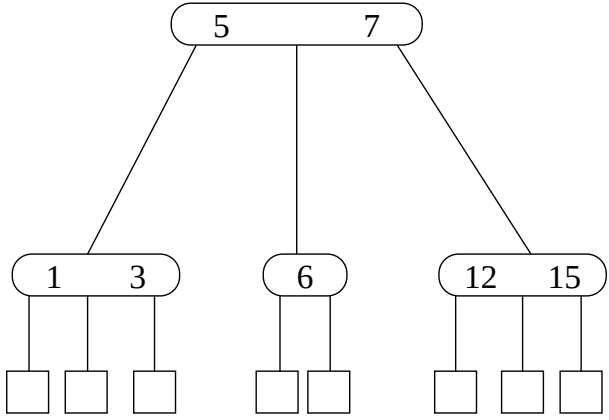
Definition: B-Baum der Ordnung m (Bayer und McCreight, 1972)

1. Alle Blätter haben gleiche Tiefe und sind leere Knoten.
2. Jeder Knoten hat bis zu m Kinder.
3. Jeder innere Knoten außer der Wurzel hat mindestens $\lceil \frac{m}{2} \rceil$ Kinder. Die Wurzel hat mindestens 2 Kinder.
4. Jeder Knoten mit l Schlüssel hat $l + 1$ Kinder.
5. Für jeden Knoten mit Schlüsseln $s_1, \ldots, s_l$ und Kindern $v_0, \ldots, v_l$ gilt: $\forall i = 1, \ldots, l$:
   - Alle Schlüssel in $T_{v_{i-1}}$ sind kleiner gleich s_i, und
   - s_i ist kleiner gleich allen Schlüsseln in T_{v_i}.

   (T_{v_i} bezeichnet den Teilbaum mit Wurzel v_i.)

B-Baum: Beispiel

Beispiel: B-Baum der Ordnung $m = 3$ (2-3 Baum) mit 7 Schlüsseln und 8 Blättern.



Implementierung eines B-Baum-Knotens

Implementierung:

- $p.l$ – Anzahl der Schlüssel
- $p.key[1], \ldots, p.key[l]$ – Schlüssel $s_1, \ldots s_l$
- $p.info[1], \ldots, p.info[l]$ – Datenfelder zu Schlüsseln $s_1, \ldots, s_l$
- $p.child[0], \ldots, p.child[l]$ – Verweis auf Kinderknoten $v_0, \ldots, v_l$

Blätter: Diese markieren wir hier durch $p.l = 0$.

Anmerkung: In einer realen Implementierung brauchen die leeren Blätter nicht explizit gespeichert zu werden. Uns erleichtern sie hier aber die Definitionen und Überlegungen.

Suche

Eingabe: B-Baum mit Wurzel p und ein Schlüssel x.

Rückgabewert: Knoten mit Schlüssel x oder $null$, falls x nicht vorhanden ist.

```
Search(p,x):
i ← 1
while i ≤ p.l und x > p.key[i]
    i ← i + 1
if i ≤ p.l und x = p.key[i]
    return (p, i)
if p.l = 0
    return null
else
    return Search(p.child[i − 1], x)
```

Einfügen

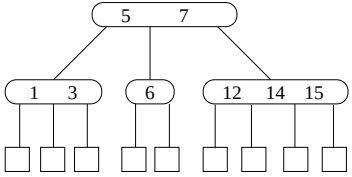
Einfügen im B-Baum:

- Schlüssel suchen $\rightarrow$ endet in Blatt p_0
- Sei p Vorgänger von p_0 und $p.child[i]$ zeigt auf p_0
- Schlüssel zwischen s_i und s_{i+1} in p einfügen, neues Blatt erzeugen
- Wenn $p.l < m$: fertig
  sonst: p splitten
  - p' : $s_1, \ldots, s_{\lceil m/2 \rceil - 1}$ $\quad$ p'' : $s_{\lceil m/2 \rceil + 1}, \ldots, s_m$
  - Schlüssel $s_{\lceil m/2 \rceil}$ in Vorgänger von p einfügen

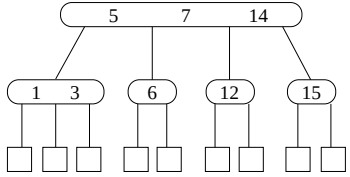
Anmerkung: B-Bäume wachsen in Ihrer Höhe immer nur durch Splitten der Wurzel!

Einfügen: Beispiel

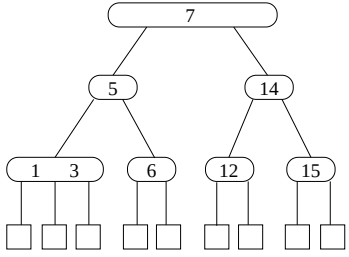
Beispiel: Einfügen von Schlüssel 14.



(a)



(b)



(c)

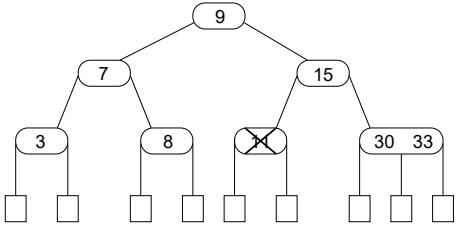
Entfernen

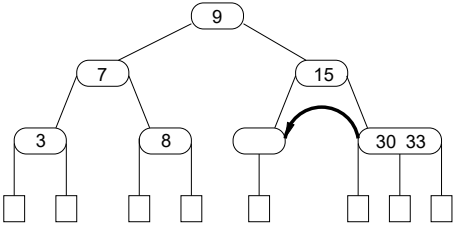
Entfernen aus einem B-Baum:

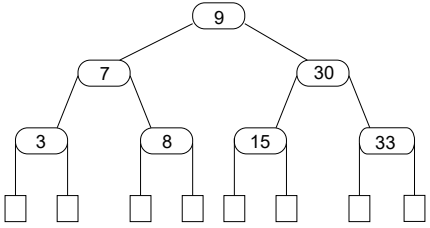
- Schlüssel suchen
- Falls Schlüssel nicht in unterster Ebene:
  mit Ersatzschlüssel aus unterster Ebene tauschen
  - Ersatzschlüssel:
    kleinster im rechten Unterbaum oder größter im linken,
    vgl. Entfernen im binären Suchbaum mit zwei Nachfolgern
- Datensatz mit Blattknoten entfernen
- Falls nun der Knoten zu wenige Schlüssel hat:
  - Übernehme Schlüssel vom linken oder rechten Geschwisterknoten wenn dieser $> \lceil m/2 \rceil$ Nachfolger hat
  - oder verschmelze mit linkem oder rechten Geschwisterknoten mit $\lceil m/2 \rceil$ Nachfolgern
  - Dabei Trennelement im Elternknoten mitberücksichtigen!

Entfernen: Beispiel

Beispiel: 11 aus B-Baum (Ordnung 3) entfernen.

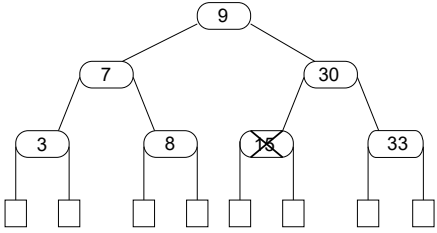


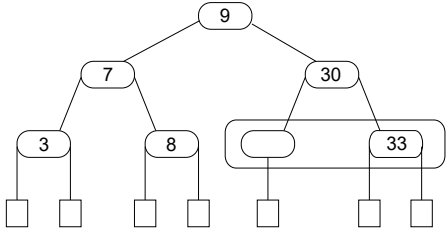


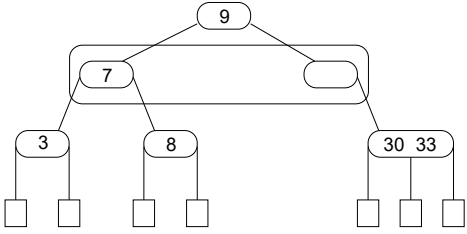


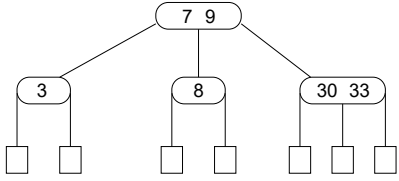
Entfernen: Beispiel

Beispiel: 15 aus B-Baum (Ordnung 3) entfernen.









Eigenschaften von B-Bäumen

Lemma: Die Anzahl der Blätter in einem B-Baum T ist immer um 1 größer als die Anzahl der Schlüssel.

Beweis: mit Induktion über die Höhe h

- Gilt für $h = 1$: Wurzel mit k Blättern, $k - 1$ Schlüssel, $2 \leq k \leq m$
- Annahme: Lemma gilt für Höhe h.
- Wir zeigen, dass es auch für Höhe $h + 1$ gilt:
  In einem B-Baum mit Höhe $h + 1$ gibt es
  k Unterbäume $T_1, \ldots, T_k$ mit Höhe h mit $n_1, \ldots, n_k$ Blättern. Diese haben
  $(n_1 - 1), \ldots, (n_k - 1)$ Schlüssel.
  Wurzel: $k - 1$ Schlüssel
  $\rightarrow$ Insgesamt gibt es $\sum_{i=1}^{k} n_i$ Blätter und
  $\sum_{i=1}^{k}(n_i - 1) + (k - 1) = \sum_{i=1}^{k} n_i - 1$ Schlüssel.

Eigenschaften von B-Bäumen

Wieviele Blätter kann es in einem B-Baum der Höhe h geben?

- Die minimale Anzahl an Blättern $N_{\min}$ wird erreicht, wenn die Wurzel nur 2 und jeder andere innere Knoten nur $\lceil m/2 \rceil$ Nachfolger hat, d.h. $N_{\min} = 2\lceil m/2 \rceil^{h-1}$.
- Die maximale Anzahl an Blättern $N_{\max}$ wird erreicht, wenn jeder innere Knoten die maximal mögliche Anzahl m von Nachfolger hat, d.h. $N_{\max} = m^h$.

Eigenschaften von B-Bäumen

Korollar: Für die Höhe h eines B-Baumes mit N Schlüsseln und $(N+1)$ Blättern muss gelten:

$$N_{\min} = 2\lceil m/2 \rceil^{h-1} \quad \leq \quad (N+1) \quad \leq \quad m^h = N_{\max}$$

und somit auch

$$\log_m(N+1) \quad \leq \quad h \quad \leq \quad 1 + \log_{\lceil m/2 \rceil}\left(\frac{N+1}{2}\right)$$

D.h., wir haben gezeigt, dass B-Bäume immer eine Höhe $h = \Theta(\log N)$ haben, und somit sind die Operationen Suchen, Einfügen und Entfernen auch wieder effizient in $\Theta(\log N)$ Zeit durchführbar.

B*-Bäume

Variante von B-Bäumen

Definition:

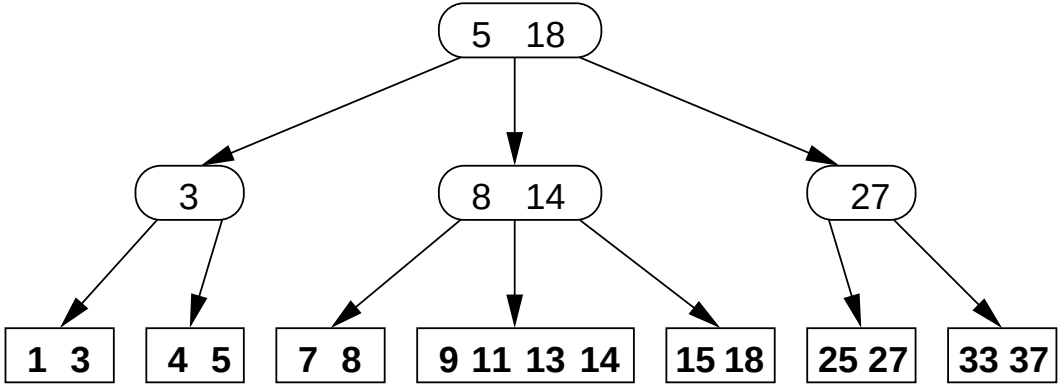
- Komplette Datensätze nur in Blättern:
  k^* bis $2k^*$ Datensätze, keine Verweise
  (k^* ist ein von m unabhängiger Parameter)

- In Zwischenknoten nur Schlüssel und Verweise
  Schlüssel: immer der des größten Elements im vorangehenden Unterbaum

Motivation und Konsequenz: m kann größer gewählt werden,
T hat geringere Höhe!

Beispiel zu B*-Bäumen

Beispiel: Ein B*-Baum mit $m = 3$ und $k^* = 2$



B*-Bäume

- Suchen: Man muss im Unterschied zum B-Baum in jedem Fall bis zu einem Blatt gehen. Das erhöht die mittlere Anzahl von Schritten jedoch kaum, zumal der B*-Baum i.A. ja auch geringere Höhe hat.
- Einfügen: Das Prinzip ist das gleiche wie beim B-Baum. Kleine Unterschiede ergeben sich dadurch, dass es nur einen trennenden Schlüssel (keine trennenden Datensätze) gibt.
- Entfernen: Der zu entfernende Datensatz liegt immer in einem Blatt. Verweise in darüberliegenden Zwischenknoten müssen ggfs. aktualisiert werden.

Hashing

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 27. April 2023

Vorlesungsfolien

Hashing

Hashing:

- Alternative Lösungsmöglichkeit des Wörterbuchproblems (siehe Kapitel über Suchbäume).
- Beobachtung: Im Allgemeinen ist lediglich eine kleine Teilmenge K aller möglichen Schlüssel $\mathcal{K}$ gespeichert.
- Idee: Statt in einer Menge von Datensätzen durch Schlüsselvergleiche zu suchen, ermittle die Position eines Elements im Speicher (bzw. einem Array) durch eine arithmetische Berechnung.

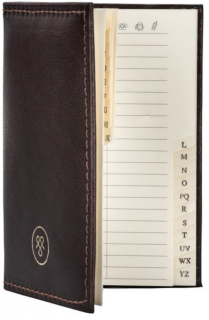
Hashing: Beispiel

Beispiel:

- Vergleich mit einem „Telefonregister“:
  Eine Seite für jeden Anfangsbuchstaben.
- Einfache (schlechte) Berechnung der Position in der Hashtabelle mit der Ordinalzahl (ord) des ersten Buchstabens im Namen s (z.B. $\mathrm{ord}('A') = 0$, $\mathrm{ord}('B') = 1$, $\ldots$).
- Hashfunktion h: hier $h(s) = \mathrm{ord}(s[0])$.

0 ("A")	Anna 123
1 ("B")	Barbara 222
2 ("C")	
3 ("D")	Doris 404, Daniel 343
4 ("E")	
5 ("F")	
6 ("G")	Günther 777
7 ("H")	
...	...
25 ("Z")	

Hashtabelle T



Hashing: Grundlagen

Hashtabelle:

- Wir gehen davon aus, dass die Hashtabelle T mit einer vorgegebenen Tabellengröße m als Array mit den Indizes $0, \ldots, m-1$ realisiert wird.
- Für eine Hashtabelle der Größe m, die aktuell n Schlüssel speichert, ist $\alpha = \frac{n}{m}$ der Belegungsfaktor der Tabelle.

Hashfunktion:

- Wir wählen eine Hashfunktion $h : \mathcal{K} \to \{0, \ldots, m-1\}$, die jedem Schlüssel $k \in \mathcal{K}$ einen eindeutigen – aber i.A. nicht umgekehrt eindeutigen – Hashwert zuordnet.

Hashing

Vorteil: Laufzeit für die Operationen Suchen, Einfügen und Entfernen liegt im Idealfall in $\Theta(1)$, wenn wir annehmen, dass $h(s)$ in konstanter Zeit berechnet werden kann.

Einschränkung:

- Gilt $h(k) = h(k')$ für $k \neq k'$, d.h., zwei verschiedene Schlüssel haben den gleichen Hashwert, so wird dies Kollision genannt.
- $\Theta(1)$ gilt nur unter der Annahme, dass die Anzahl der Kollisionen vernachlässigbar klein ist.
  - Im Erwartungsfall (bei entsprechender Konfiguration) erreichbar.
- Im Worst-Case liegt der Aufwand in $O(n)$.

Hashing

Zu klärende Punkte:

- Wie erfolgt die Kollisionsbehandlung?
  - Verkettung der Überläufer
  - Offene Hashverfahren
- Was ist eine gute Hashfunktion?
  - Divisions-Rest-Methode
  - Multiplikationsmethode
- Wie soll die Tabellengröße m gewählt werden?

Alle diese Aspekte beeinflussen die Güte/Effizienz der Hashtabelle.

Hashfunktionen

Was charakterisiert eine gute Hashfunktion?

Vor allem:

- Verwendete Schlüssel sollen möglichst gleichmäßig auf alle Plätze $0, \ldots, m-1$ der Hashtabelle aufgeteilt werden.
- Auch kleinste Änderungen im Schlüssel sollen zu einem anderen, möglichst unabhängigen Hashwert führen.

Divisions-Rest-Methode

Annahme: $k \in \mathbb{N}$

Berechnung:

$$h(k) = k \bmod m$$

Eigenschaften:

- Die Hashfunktion kann sehr schnell berechnet werden.
- Die richtige Wahl von m ist hier sehr wichtig.
  Eine gute Wahl für m ist eine Primzahl.

Divisions-Rest-Methode

Berechnung:

$$h(k) = k \bmod m$$

Schlechte Wahl für m:

- $m = 2^i$: Nur die letzten i Binärziffern spielen eine Rolle!
- $m = 10^i$: Analog bei Dezimalzahlen.
- $m = r^i$: Analog bei r-adischen Zahlen.
- aber auch $m = r^i \pm j$ für kleines j kann problematisch sein:
  z.B.: $m = 2^7 - 1 = 127$: Buchstaben als Zahlen interpretieren
  ('p' = 112 in ASCII)
  $h(\text{“pt”}) = (112 \cdot 128 + 116) \bmod 127 = 14452 \bmod 127 = 101$
  $h(\text{“tp”}) = (116 \cdot 128 + 112) \bmod 127 = 14960 \bmod 127 = 101$ (Schlüssel in denen zwei Buchstaben vertauscht sind haben hier häufig den gleichen Hashwert)

Divisions-Rest-Methode für Strings

Schlüssel: String $s = (s_1, \ldots, s_l) \in \{0, \ldots, 127\}^l$

Berechnung: $k = 128^{l-1}s_1 + 128^{l-2}s_2 + \ldots + s_l$
Die sehr großen ganzzahligen Werte sind problematisch!

Berechnung mit Horner-Schema:

$$k = (\ldots(s_1 \cdot 128 + s_2) \cdot 128 + s_3) \cdot 128 + \ldots + s_{l-1}) \cdot 128 + s_l$$

Es gilt: $k \bmod m = (\ldots(s_1 \cdot 128 + s_2) \bmod m) \cdot 128 + s_3) \bmod m) \cdot 128 + \ldots$
$+ s_{l-1}) \bmod m) \cdot 128 + s_l) \bmod m$

Konsequenz: Keine Zwischenresultate $> (m-1) \cdot 128 + 127$
Berechnung mit üblichen Integer-Typen so gut möglich.

Multiplikationsmethode

Grundlegende Idee:

- Wir nehmen wieder an: Schlüssel $k \in \mathbb{N}$
- Gegeben: irrationale Zahl A
- Berechnung:

$$h(k) = \lfloor m \underbrace{(k \cdot A - \lfloor k \cdot A \rfloor)}_{\in [0,1)} \rfloor$$

- Der Schlüssel wird mit A multipliziert,
  der ganzzahlige Anteil des Resultats wird abgeschnitten.
- Man erhält einen Wert in $[0, 1)$, dieser wird mit der Tabellengröße m multipliziert, das Ergebnis gerundet.
- Eigenschaft, die gleichmäßige Streuung bestätigt:
  Für eine Schlüsselfolge 1,2,3,...,i liegt $k \cdot A - \lfloor k \cdot A \rfloor$ des nächsten Schlüssels $k = i + 1$ immer in einem größten Intervall zwischen allen zuvor ermittelten Werten, 0 und 1.

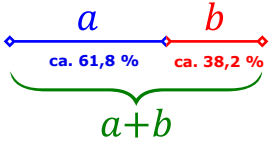
Multiplikationsmethode: Wahl für A

Allgemein:

- Die Wahl von m ist hierbei unkritisch, sofern A eine irrationale Zahl ist.

Beste Wahl für A: Der goldene Schnitt

$$A = \Phi^{-1} = \frac{\sqrt{5}-1}{2} = 0.6180339887...$$



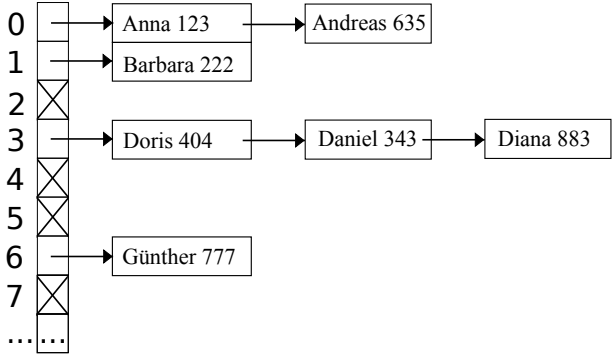
$$\frac{a}{a+b} = \frac{b}{a}$$

Eine Begründung warum Φ^{-1} die beste Wahl ist findet sich in: D.E. Knuth: The Art of Computer Programming, Vol.3: Sorting and Searching, Addison-Wesley, 1973.

Kollisionsbehandlung
Verkettung der Überläufer

Verkettung der Überläufer

Idee: Jedes Element der Hashtabelle ist eine verkettete Liste.



Initialisierung

Eingabe: Hashtabelle T = Array von m Verweisen auf die jeweils ersten Elemente.

Ergebnis: Initialisierte Hashtabelle.

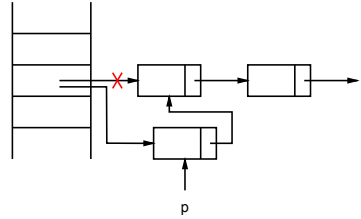
```
Initialize(T, m):
for i ← 0 bis m − 1
    T[i] = null
```

Einfügen

Eingabe: Hashtabelle T und einzufügendes Element p.

Ergebnis: Hashtabelle T mit neu eingetragenem Element p.

```
Insert(T, p):
pos ← h(p.key)
p.next ← T[pos]
T[pos] ← p
```



Hinweis: Der Hashwert (und damit die Position in der Hashtabelle) für $p.key$ wird mit der Funktion h berechnet.

Suchen

Eingabe: Hashtabelle T und gesuchter Schlüssel k.

Rückgabewert: Gesuchtes Element p.

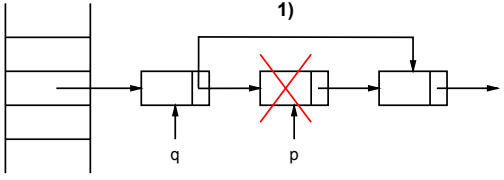
```
Search(T, k):
p =T[h(k)]
while p ≠ null und p.key ≠ k
    p ← p.next
return p
```

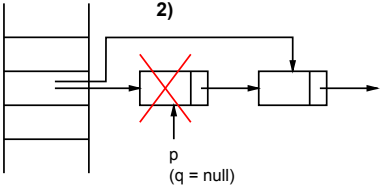
Entfernen eines Elements

Eingabe: Hashtabelle T und Schlüssel k des zu entfernenden Elements (wir gehen davon aus, dass ein Element mit dem gesuchten Schlüssel in T enthalten ist).

Ergebnis: Element mit dem Schlüssel k wurde aus T entfernt.

```
Remove(T, k):
pos ← h(k)
q ← null
p ← T[pos]
while p.key ≠ k
    q ← p
    p ← p.next
if q = null
    T[pos] ← T[pos].next
else
    q.next ← p.next;
```





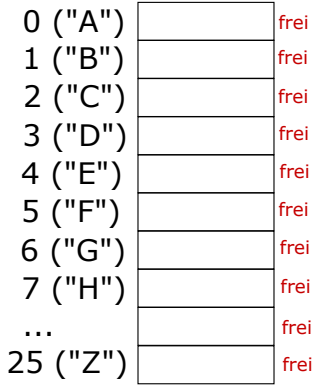
Kollisionsbehandlung
Offene Hashverfahren

Offene Hashverfahren: Grundlegende Idee

Alle Datensätze werden direkt in einem einfachen Array gespeichert,
pro Platz ein Flag $f_i \in \{\textit{frei}, \textit{besetzt}, \textit{wieder frei}\}$.

Kollisionsbehandlung: Wenn ein Platz belegt ist werden in einer bestimmten Reihenfolge weitere Plätze betrachtet (sondiert).

Beispiel: Alle Plätze sind anfangs frei.



Offene Hashverfahren: Grundlegende Idee

Beispiel:

- Hashfunktion: Ordinalzahl des ersten Buchstabens im Namen
- Sondierreihenfolge: einfach die nächste Position

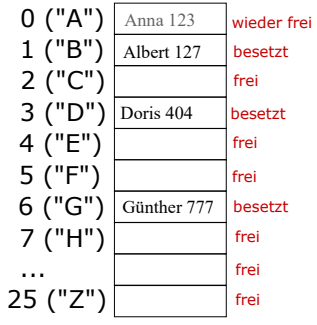
0 ("A")	Anna 123	besetzt
1 ("B")	Albert 127	besetzt
2 ("C")		frei
3 ("D")	Doris 404	besetzt
4 ("E")		frei
5 ("F")		frei
6 ("G")	Günther 777	besetzt
7 ("H")		frei
...		frei
25 ("Z")		frei

- Anna wurde vor Albert eingetragen, Albert kommt daher auf die nächste freie Position 1.
- Ein weiterer Eintrag für Andreas würde an Position 2 gespeichert werden, Armin danach an Position 4, usw.

Offene Hashverfahren: Grundlegende Idee

Beispiel:

- **Entfernen:** Flag wird auf *wieder frei* gesetzt,
  im Beispiel wird Anna entfernt.
- Würde $f_0 = \textit{frei}$ gesetzt werden, so würde Albert nicht mehr gefunden werden da die Sondierung bei Position 0 abbricht!
- Ein neuer Eintrag für Andreas würde wieder an Position 0 gespeichert werden.



Offene Hashverfahren: Im Detail

- Alle Elemente werden – im Gegensatz zur Verkettung der Überläufer – direkt im Array gespeichert.
- Zu jedem Platz $i = 0, \ldots, m-1$ wird ein Flag $f_i \in \{\textit{frei}, \textit{besetzt}, \textit{wieder frei}\}$ gespeichert.
- In der anfangs leeren Tabelle gilt für alle i $f_i = \textit{frei}$.
- Beim Einfügen wird das neue Element am ersten mit *frei* oder *wieder frei* markierten Platz eingefügt, das Flag wird auf *besetzt* gesetzt.
- Die Suche durchmustert alle Plätze bis der gesuchte Schlüssel entweder gefunden wird oder ein Platz als *frei* markiert ist (Schlüssel nicht enthalten).
- Das Entfernen setzt das Flag für das zu entfernende Element auf *wieder frei*.

Offene Hashverfahren: Sondierung

Kollisionsbehandlung: Wenn ein Platz belegt ist, so werden in einer bestimmten Reihenfolge weitere Plätze in Betracht gezogen.

Sondierungsreihenfolge (*Probing*):
Reihenfolge der auszuprobierenden Plätze.

$\rightarrow$ Die Hashfunktion wird zu einer Sondierungsfunktion $h(k, i)$ für Schlüssel k und Positionen $i = 0, 1, \ldots, m-1$ erweitert, die Sonderiungsreihenfolge ist dann $h(k, 0),\ h(k, 1), \ldots, h(k, m-1)$.

Lineares Sondieren

Gegeben: Eine normale Hashfunktion:

$$h' : \mathcal{K} \to \{0, 1, \ldots, m-1\}.$$

Lineares Sondieren: Wir definieren für $i = 0, 1, \ldots, m-1$:

$$h(k, i) = (h'(k) + i) \bmod m.$$

Lineares Sondieren: Beispiel

Beispiel: $m = 8$, $h'(k) = k \bmod m$

Schlüssel und Wert der Hashfunktion:

k	10	19	31	22	14	16
$h'(k)$	2	3	7	6	6	0

Belegung der Hashtabelle:

0	1	2	3	4	5	6	7
14	16	10	19			22	31

Durchschnittliche Zeit: Für eine erfolgreiche Suche $\frac{9}{6} = 1,5$ (siehe Tabelle).

k	10		19		31		22		14		16		
	1	+	1	+	1	+	1	+	3	+	2	=	9

Lineares Sondieren: Probleme

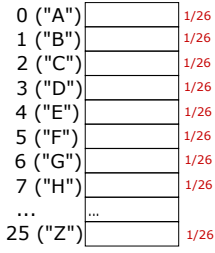
Problem:

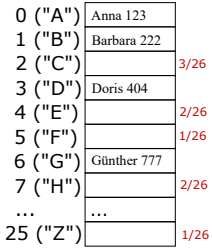
- Nach dem Einfügen ist die Wahrscheinlichkeit für einen neu einzufügenden Schlüssel in der Hashtabelle an einer gewissen Position gespeichert zu werden für die verschiedenen Positionen unterschiedlich.
- Wird ein Platz belegt, dann verändert sich die Wahrscheinlichkeit für das Einfügen an seinem nachfolgenden Platz.

Lineares Sondieren: Probleme

Beispiel:

- In der leeren Tabelle haben alle Plätze die gleiche Wahrscheinlichkeit.
- Nach dem Einfügen von verschiedenen Schlüsseln verändern sich die Wahrscheinlichkeiten. Z.B. werden auf der rechten Seite im Eintrag T[2] nach dem Einfügen von Anna und Barbara alle Schlüssel k mit $h(k) = 0$ oder $h(k) = 1$ oder $h(k) = 2$ gespeichert, im Eintrag T[5] dagegen nur alle Schlüssel k mit $h(k) = 5$.





Lineares Sondieren: Probleme

Probleme:

- Lange belegte Teilstücke der Hashtabelle haben eine stärkere Tendenz zu wachsen als kurze.
- Dieser Effekt wird noch verstärkt, weil lange belegte Teilstücke zu größeren zusammenwachsen, wenn die Lücken zwischen ihnen geschlossen werden.
- Als Folge dieses Phänomens der primären Häufung (*primary clustering*) verschlechtert sich die Effizienz des linearen Sondierens drastisch, sobald sich der Belegungsfaktor α dem Wert 1 nähert.

Uniform Hashing

Uniform Hashing:

- Idealform des Sondierens.
- Jeder Schlüssel erhält mit gleicher Wahrscheinlichkeit eine bestimmte der $m!$ Permutationen von $0, 1, \ldots, m-1$ als Sondierungsreihenfolge zugeordnet.
- Ist in der Praxis schwierig zu implementieren und wird daher mit den nachfolgenden Verfahren approximiert.

Quadratisches Sondieren

Idee: Um die primäre Häufung des linearen Sondierens zu vermeiden, wird beim quadratischen Sondieren für Schlüssel k von $h(k)$ aus mit quadratisch wachsendem Abstand nach einem freien Platz gesucht.

Gegeben: Eine normale Hashfunktion:

$$h' : \mathcal{K} \to \{0, 1, \ldots, m-1\}.$$

Quadratisches Sondieren: Sondierungsfunktion lautet nun:

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

Dabei sind c_1 und c_2 geeignet gewählte Konstanten.

Quadratisches Sondieren: Beispiel

Beispiel: $m = 8$, $h'(k) = k \bmod m$, $c_1 = c_2 = \frac{1}{2}$, gleiche Schlüssel wie vorhin.

Schlüssel und Wert der Hashfunktion:

k	10	19	31	22	14	16
$h'(k)$	2	3	7	6	6	0

0	1	2	3	4	5	6	7
		10	19			22	31

$$14 \to 6 \quad \to \quad 6 + \frac{1}{2}1 + \frac{1}{2}1^2 \bmod 8 = 7$$
$$\to \quad 6 + \frac{1}{2}2 + \frac{1}{2}2^2 \bmod 8 = 1$$

0	1	2	3	4	5	6	7
16	14	10	19			22	31

Quadratisches Sondieren: Beispiel und Analyse

0	1	2	3	4	5	6	7
16	14	10	19			22	31

Durchschnittliche Zeit: Für eine erfolgreiche Suche $\frac{8}{6} \approx 1.33$.

k	10		19		31		22		14		16		
	1	+	1	+	1	+	1	+	3	+	1	=	8

Probleme: Primäre Häufungen werden vermieden, aber ein anderes Phänomen, die sekundären Häufungen (*secondary clustering*) können auftreten.

Güte von Kollisionsbehandlungen

Theoretische Analyseergebnisse:

Durchschnittliche Anzahl der Sondierungen für große m, n:

α	Verkettung		offene Hashverfahren					
			lineares S.		quadr. S.		unif. hashing	
	erfolgreich	erfolglos	er	el	er	el	er	el
0.5	1.250	0.50	1.5	2.5	1.44	2.19	1.39	2
0.9	1.450	0.90	5.5	50.5	2.85	11.40	2.56	10
0.95	1.475	0.95	10.5	200.5	3.52	22.05	3.15	20
1.0	1.500	1.00	—	—	—	—	—	—

er: erfolgreiche Suche, el: erfolglose Suche

Ergebnisse von D.E. Knuth: The Art of Computer Programming, Vol.3: Sorting and Searching, Addison-Wesley, 1973.

Double Hashing

Idee: Die Effizienz des uniformen Sondierens wird bereits annähernd erreicht, wenn man statt einer zufälligen Permutation für die Sondierungsfolge eine zweite Hashfunktion verwendet.

Gegeben: Zwei Hashfunktionen:

$$h_1, h_2 : \mathcal{K} \to \{0, 1, \ldots, m-1\}.$$

Double Hashing: Wir definieren für $i = 0, 1, \ldots, m-1$:

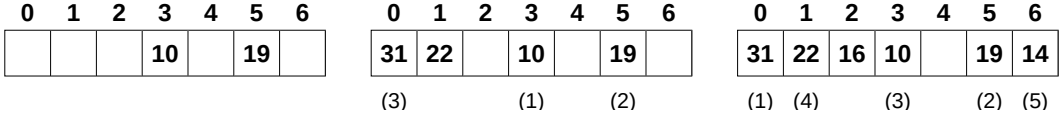
$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m.$$

Wahl von $h_2(k)$: Für alle Schlüssel k muss die Sondierungsfolge alle Plätze $0, \ldots, m$ erreichen. Das bedeutet, dass $h_2(k) \neq 0$ sein muss und m nicht teilen darf.
m sollte eine Primzahl sein, h_2 sollte unabhängig von h_1 gewählt werden.

Double Hashing

Beispiel: $m = 7$, $h_1(k) = k \bmod 7$, $h_2(k) = 1 + (k \bmod 5)$

k	10	19	31	22	14	16
$h_1(k)$	3	5	3	1	0	2
$h_2(k)$	1	5	2	3	5	2



Durchschnittliche Zeit: Für eine erfolgreiche Suche ist $\frac{12}{6} = 2$. Dies ist jedoch ein untypisch schlechtes Beispiel für Double Hashing.

k	10		19		31		22		14		16		
	1	+	1	+	3	+	1	+	5	+	1	=	12

Double Hashing

- Im Allgemeinen ist Double Hashing effizienter als quadratisches Hashing.
- In der Praxis entsprechen die Ergebnisse von Double Hashing nahezu denen von uniformen Hashing.

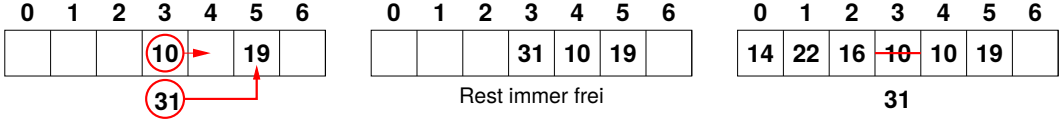
Verbesserung nach Brent [1973]

Idee: Wenn beim Einfügen eines Schlüssels ein sondierter Platz j mit $k' = T[j].key$ belegt ist, setze

$$j_1 = (j + h_2(k)) \bmod m$$
$$j_2 = (j + h_2(k')) \bmod m.$$

Ist nun j_1 besetzt aber j_2 frei, verschiebe k' auf j_2 um für k auf j Platz zu machen.

Angewendet auf unser Beispiel:



Durchschnittliche Zeit: Für eine erfolgreiche Suche ist $\frac{7}{6} \approx 1.17$.

k	10		19		31		22		14		16		
	2	+	1	+	1	+	1	+	1	+	1	=	7

Einfügen nach Brent

Eingabe: Hashtabelle T und neuer Schlüssel k.

```
Insert-Brent(T, k):
j ← h_1(k)
while T[j].status = used
    k' ←T[j].key
    j_1 ←(j + h_2(k)) mod m
    j_2 ←(j + h_2(k')) mod m
    if T[j_1].status ≠ used oder T[j_2].status = used
        j ← j_1
    else
        T[j] ← k
        k ← k'
        j ← j_2
T[j] ← k
T[j].status ← used
```

Analyseergebnis zur Verbesserung nach Brent

Anzahl der Sondierungen: Im Durchschnitt für große m und n:

- Erfolglose Suche $\approx \frac{1}{1-\alpha}$
- Erfolgreiche Suche < 2.5 (unabhängig von α für $\alpha \leq 1$).

Vorteil: Durchschnittlicher Aufwand einer erfolgreichen Suche liegt selbst im Extremfall $\alpha = 1$ in $\Theta(1)$.

Offene Hashverfahren: Eignung und Reorganisation

- Bei offenen Hashverfahren ist der Belegungsfaktor $\alpha = \frac{n}{m}$ immer kleiner (max. gleich) 1, offensichtlich können nicht mehr als m Elemente gespeichert werden.
- Belegungsfaktoren sehr nahe 1 sind i.A. ungünstig, da die Anzahl der zu sondierenden Positionen sehr groß werden kann!
- Gegebenenfalls ist eine Reorganisation notwendig, d.h., dass eine gänzlich neue Hashtabelle z.B. mit doppelter Größe aufgebaut wird, wenn mehr als die ursprünglich erwartete Anzahl an Elementen zu speichern ist (Aufwand i.A. $\Theta(n)$).
- Generell sind offene Hashverfahren besser geeignet, wenn die Anzahl der zu speichernden Elemente vorab bekannt ist und selten oder gar nicht Elemente entfernt werden. Häufiges Entfernen bewirkt, dass sich die Sondierungsketten für die Suche verlängern; eine regelmäßige Reorganisation kann dann ebenfalls sinnvoll bzw. notwendig werden.

Praktische Datenstrukturen in Java
Ein Überblick

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 2. Mai 2023

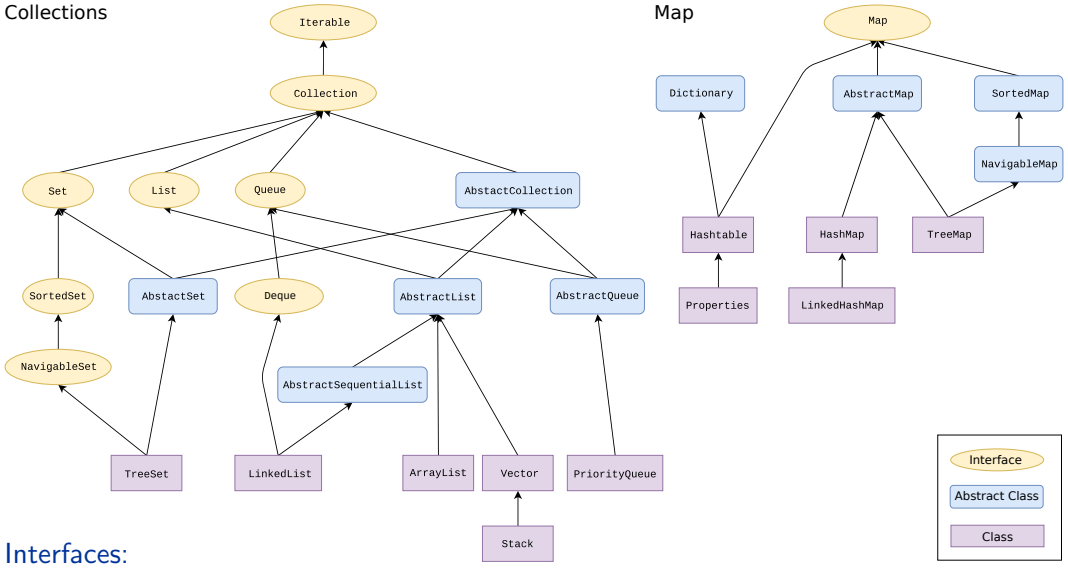
Vorlesungsfolien

Themen

- Java Collections-Framework
- Ausgewählte Klassen
- Algorithmen im Collections-Framework
- Weitere Beispiele für Bibliotheken

Java Collections-Framework

Grundlagen



Interfaces:

- Für unterschiedliche Datenstrukturen.
- Schnittstellen für den Zugriff auf die Datenstrukturen.
- Konkrete Implementierungen realisieren diese Interfaces.

Interfaces

Collection:

- Eine Collection verwaltet Objekte (Elemente).
- Interface enthält generelle Methoden (für alle Collection-Typen).
- Es gibt keine direkte Implementierung dieses Interfaces.

Set:

- Eine Collection, die keine Duplikate enthält.

List:

- Eine geordnete Collection (die Duplikate enthalten kann).
- Elemente können über einen Index angesprochen werden (nicht immer effizient).

Queue/Deque:

- Verwalten von Warteschlangen.
- FIFO, Prioritätswarteschlangen.
- Einfügen und Löschen an beiden Enden bei Deque („double ended queue“).

Interfaces

Map:

- Maps verwalten Schlüssel mit dazugehörigen Werten.

SortedSet und SortedMap:

- Sind spezielle Versionen von Set und Map, bei denen die Elemente (Schlüssel) in aufsteigender Reihenfolge verwaltet werden.

Generelle Implementierungen (Beispiele)

	Interface				
Konzept	Set	List	Queue	Deque	Map
Arrays	–	ArrayList	–	ArrayDeque	–
Bäume	TreeSet	–	–	–	TreeMap
Hashtabellen	HashSet	–	–	–	HashMap
Heap	–	–	PriorityQueue	–	–
Verkettete Listen	–	LinkedList	LinkedList	LinkedList	–

Anmerkung: Interfaces SortedSet und SortedMap werden von TreeSet bzw. TreeMap zusätzlich implementiert.

Ausgewählte Klassen

Listenimplementierungen

ArrayList:

- Indexzugriff auf Elemente ist überall gleich schnell ($O(1)$).
- Einfügen und Löschen ist am Listenende schnell und wird mit wachsender Entfernung vom Listenende langsamer ($O(n)$).

LinkedList:

- Indexzugriff auf Elemente ist an den Enden schnell und wird mit der Entfernung von den Enden langsamer ($O(n)$).
- Einfügen und Löschen ohne Indexzugriff ist überall gleich schnell ($O(1)$).

Queue-Implementierungen

LinkedList ist auch eine Implementierungen von Queue.

PriorityQueue

- Ist als Min-Heap implementiert.
- Einfügen eines Elements und Löschen des ersten Elements in einer Queue der Größe n sind in $O(\log n)$.
- Löschen eines beliebigen Elements aus einer Queue der Größe n ist in $O(n)$.
- Lesen des ersten Elements in einer Queue ist in konstanter Zeit möglich ($O(1)$).

Beispiele zu ArrayList, LinkedList und PriorityQueue

```
import java.util.*;

public class Example {
    public static void main(String[] args) {
        List<Integer> myAL = new ArrayList<>();
        myAL.add(42); myAL.add(17);
        System.out.println(myAL); // [42, 17]

        List<Integer> myLL = new LinkedList<>();
        myLL.add(42); myLL.add(17);
        System.out.println(myLL); // [42, 17]


        PriorityQueue<Integer> myPQ = new PriorityQueue<>();
        myPQ.add(750);
        myPQ.add(500);
        myPQ.add(900);
        myPQ.add(100);

        while (!myPQ.isEmpty()) {
            System.out.print(myPQ.remove() + " ");
        } // 100 500 750 900
    }
}
```

Set-Implementierungen: TreeSet

TreeSet:

- Implementiert als Rot-Schwarz-Baum.
- null-Elemente sind nicht erlaubt.
- Die Laufzeit von Einfügen, Suchen und Löschen eines Elements liegt bei einem Baum mit n Elementen in $O(\log n)$.
- Auf die Elemente eines TreeSets muss eine Ordnung definiert sein (müssen vergleichbar sein, d.h. das Interface Comparable implementieren).

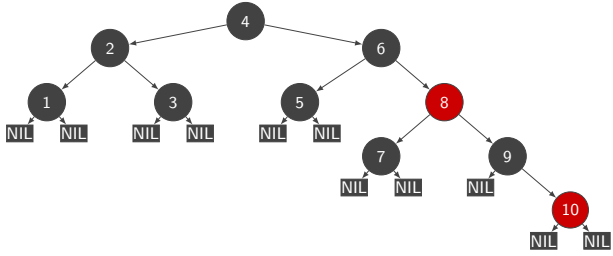
Rot-Schwarz-Baum:

- Variante eines balancierten Suchbaums.
- Kann als Spezialfall eines B-Baums der Ordnung 4 betrachtet werden.
- Einfügen und Löschen ist effizienter als in B-Bäumen solange die Datenmenge nicht zu groß ist.

Rot-Schwarz-Baum

- Selbstbalanzierender binärer Suchbaum: jeder Knoten entweder rot oder schwarz.
- Die Wurzel ist schwarz. Jeder Blattknoten (NIL) ist schwarz.
- Kindknoten von einem roten Knoten sind schwarz.
- Für jeden Knoten gilt, dass alle Pfade von diesem Knoten zu nachfolgenden Blattknoten die gleiche Anzahl an schwarzen Knoten beinhalten.
- Einfügen und Löschen können eine Rebalancierung erfordern.

Beispiel:



Set-Implementierungen: HashSet

HashSet:

- null-Elemente sind zulässig.
- Einfügen, Suchen und Löschen sind in konstanter Zeit möglich (abhängig von der Verteilung der Einträge in der Hashtabelle).
- Aber: Rehashing kann zu Performance-Problemen führen (siehe auch API-Beschreibung).

Implementierung:

- Hashing mit Verkettung der Überläufer. Falls die Liste zu lange wird, wird sie in eine TreeMap umgewandelt.
- Maximaler Belegungsfaktor $= 0.75$.
- Bei Überschreitung des Belegungsfaktors verdoppelt sich die Größe. (Größe der Hashtabelle ist immer eine Zweierpotenz).

Beispiele zu TreeSet und HashSet

```
import java.util.*;

public class Example {
    public static void main(String[] args) {
        Set<String> myTS = new TreeSet<>();
        myTS.add("apples"); myTS.add("bananas"); myTS.add("strawberries"); myTS.add("grapes");
        myTS.add("strawberries"); // adding duplcate elements will be ignored
        System.out.println(myTS); // [apples, bananas, grapes, strawberries]


        Set<String> myHS = new HashSet<>();
        myHS.add("apples"); myHS.add("bananas"); myHS.add("strawberries"); myHS.add("grapes");
        myHS.add("strawberries"); // adding duplcate elements will be ignored
        System.out.println(myHS); // [strawberries, bananas, apples, grapes]
    }
}
```

Verschiedene Implementierungen von Hashing

Beispiele für Verkettung der Überläufer:

	Belegungsfaktor	Wachstumsfaktor	Tabellengröße
Java	0.75	2	Zweierpotenzen
C++	1	>2	Primzahlen
Go	6.5	2	Zweierpotenzen

Beispiele für offenes Hashing:

	Belegungsfaktor	Wachstumsfaktor	Tabellengröße	Sondierung
Python	0.66	2	Zweierpotenzen	Quadratisch
Ruby	0.5	2	Zweierpotenzen	Quadratisch

Anmerkung: Die Sondierungsfunktion verwendet in beiden Fällen zusätzlich eine sogenannte Perturbation.

Maps

- Maps sind eine Verallgemeinerung von Arrays mit einem beliebigen Indextyp (nicht nur int).
- Eine Map ist eine Menge von Schlüssel-Werte Paaren.
  - Schlüssel müssen innerhalb einer Map eindeutig sein.
  - Werte müssen nicht eindeutig sein.

Arten von Maps: Wie bei Sets gibt es grundsätzlich zwei Versionen.

- HashMap: Implementierung wie HashSet.
- TreeMap: Implementierung wie TreeSet.

Anmerkung: Genau genommen sind Tree- und HashMap die primitiven Implementierungen die von Tree- und HashSet verwendet werden.

Operationen auf Maps: Bestimmte Methoden wie z.B. put, get, containsKey, containsValue, remove etc. werden angeboten.

Beispiele zu HashMap und TreeMap

```
import java.util.*;
public class Example {  public static void main(String[] args) {
      Map<String, Integer> myHM = new HashMap<>();
      myHM.put("one", 1); myHM.put("two", 2); myHM.put("three", 3); myHM.put("four", 4);
      myHM.putIfAbsent("five", 5); // only add if key does not exist or is mapped to `null`

      System.out.println(myHM); // {one=1, two=2, three=3, four=4, five=5}

      String id = "three";
      if(myHM.containsKey(id)) {
         System.out.println("Found key "+id+". Value: "+myHM.get(id)); // Found key three. Value: 3
      }


      Map<String, Integer> myTM = new TreeMap<>();
      myTM.put("one", 1); myTM.put("two", 2); myTM.put("three", 3); myTM.put("four", 4);
      myTM.putIfAbsent("four", 444); // only add if key does not exist or is mapped to `null`

      System.out.println(myTM); // {four=4, one=1, three=3, two=2}

      id = "four";
      if(myTM.containsKey(id)) {
         System.out.println("Found key " + id + ". Value: " + myTM.get(id)); // Found key four. Value: 4
      }
}}
```

Abstrakte Klassen

- Unterstützen die Entwicklung neuer Klassen im Framework.
- Implementieren einen Teil eines Interfaces und lassen bestimmte Teile noch offen.

Neue Klasse implementieren:

- Auswählen einer geeigneten abstrakten Klasse von der geerbt wird.
- Implementierung aller abstrakten Methoden.
- Sollte die Collection modifizierbar sein, dann müssen auch einige konkrete Methoden überschrieben werden (siehe API).
- Testen der neuen Klasse (inklusive Performance).

Abstrakte Klassen

Beispiele:

- AbstractCollection
- AbstractSet
- AbstractList (basierend auf Array)
- AbstractSequentialList (basierend auf verketteter Liste)
- AbstractQueue
- AbstractMap

API: API-Dokumentation jeder abstrakten Klasse beschreibt genau, wie man eine Klasse ableiten muss:

- Grundlegende Implementierung.
- Welche Methoden müssen implementiert werden.
- Welche Methoden müssen überschrieben werden, wenn man Modifikationen zulassen möchte.

Algorithmen im Collections-Framework

Algorithmen im Collections-Framework

Algorithmen:

- Das Collections-Framework bietet auch Algorithmen für die Verarbeitung von Container-Klassen an.
- Diese Algorithmen werden als statische Methoden (polymorphe Methoden) in der Hilfsklasse Collections gesammelt.

Beispiele:

- sort sortiert die Elemente einer generischen Liste nach aufsteigender Größe.
- binarySearch sucht ein Element in der sortierten Liste (Voraussetzung) und liefert einen Index zurück, wenn das Element gefunden wurde (ansonsten eine negative Zahl).
- max liefert das größte Element einer Collection.
- shuffle mischt die Elemente einer generischen Liste zufällig.

Beispiele zu Algorithmen im Collections-Framework

```
import java.util.*;
public class Example {
    public static void main(String[] args) {
        List<Integer> myAL = new ArrayList<>(Arrays.asList(5, 3, 1, 4));
        System.out.println(myAL); // [5, 3, 1, 4]
        Collections.sort(myAL);
        System.out.println(myAL); // [1, 3, 4, 5]

        System.out.println(Collections.max(myAL)); // 5

        System.out.println(Collections.binarySearch(myAL, 5)); // 3
        System.out.println(Collections.binarySearch(myAL, 42)); // -5

        Collections.shuffle(myAL);
        System.out.println(myAL); // z.B.: [1, 4, 3, 5]
    }
}
```

Algorithmen im Collections-Framework: Beispiel sort

Sortieren von Collections:

- Zum Sortieren von Collections wird TimSort verwendet.
- TimSort ist eine Kombination von Mergesort und Insertionsort.
- TimSort wurde 2002 von Tim Peters für die Verwendung in Python entwickelt.
- Heute wird TimSort unter anderem in Python, Java, Android und Google Chrome verwendet.

TimSort

Grundlegende Idee: Finde schon sortierte Stücke (Runs) des Inputs S, die dann paarweise verschmelzt werden (Merging).

```
TimSort(S)
runs ← partitioniere S in Runs
R ← leerer Stack
while runs ≠ ∅
    entferne Run r von runs and pushe r auf den Stack R
    merge_collapse(R) // ausbalanciertes Verschmelzen
while height(R) ≠ 1
    verschmelze die beiden oberen Runs am Stack
```

TimSort: Berechnen von Runs

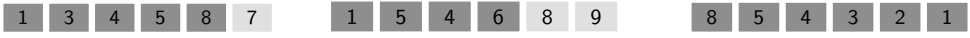
Min_Run:

- Runs sind mindestens Min_Run lang.
- Min_Run wird so gewählt, dass $\frac{\text{Größe des Arrays}}{\text{Min\_Run}}$ eine Zweierpotenz oder etwas kleiner als eine Zweierpotenz ist.
- Experimente zeigen, dass ein Wert zwischen 32 und 64 für Min_Run optimal ist.

Finden von Runs:

- Startend bei dem ersten Element, das noch nicht in einem Run ist, füge so lange Elemente zum Run hinzu, wie die Sequenz entweder aufsteigend oder *strikt* absteigend ist.
- Falls die Sequenz strikt absteigend war, invertiere die Reihenfolge der Elemente.
- Falls die Länge der Sequenz größer Min_Run ist, ist damit ein Run gefunden.
- Andernfalls füge solange nachfolgende Elemente hinzu bis Min_Run erreicht ist. Sortiere den entstandenen Run mittels (binärem) Insertionsort.

Runs mit Min_Run = 4:



TimSort: Verschmelzen der Runs (Merging)

- Nachdem das Array in Runs aufgeteilt ist und diese sortiert sind, werden diese wie bei Mergesort verschmolzen.
- Merging ist am effizientesten, wenn die Anzahl der Runs eine Zweierpotenz ist und die Runs möglichst gleich groß sind.
- Da Runs im Allgemeinen sehr unterschiedliche Größen haben können, wird versucht mithilfe von Invarianten eine sinnvolle Merge-Reihenfolge zu finden (Beachte: Nur „benachbarte“ Runs werden verschmolzen).
- Zusätzlich wird beim Merging eine Strategie namens Galloping verwendet, um möglicherweise vorhandene Strukturen in den Runs zu nutzen.

TimSort: Invarianten und merge_collapse

Stack $\mathcal{R}$ von Runs mit Aufbau $\mathcal{R}[1], \ldots, \mathcal{R}[\text{height}(\mathcal{R})]$, wobei $\mathcal{R}[1]$ top-of-stack ist. Die Länge von Run $\mathcal{R}[i]$ wird mit r_i bezeichnet.

```
merge_collapse(R)
while height(R) > 1
    if height(R) > 2 und r_3 ≤ r_2 + r_1
        if  r_3 < r_1
            verschmelze R[2] und R[3] am Stack
        else
            verschmelze R[1] und R[2] am Stack
    elseif height(R) > 1 und r_2 ≤ r_1
        verschmelze R[1] und R[2] am Stack
    else break
```

Nach Ausführung von merge_collapse($\mathcal{R}$) gelten folgende Invarianten:

(1) $r_3 > r_2 + r_1$

(2) $r_2 > r_1$

TimSort: Beispiel zu Invarianten

Wiederholung Invarianten: (1) $r_3 > r_2 + r_1$, (2) $r_2 > r_1$

Länge der Runs am Stack werden in der Reihenfolge $r_1, r_2, r_3, \ldots$ angegeben.

- 15, 67, 138. Invarianten erfüllt.
- Nächster Run hat Größe 16. Invariante (2) nicht erfüllt! (16, 15, 67, 138)
- Mergen ergibt: 31, 67, 138. Invarianten erfüllt.
- Nächster Run hat Größe 17. Invarianten erfüllt! (17, 31, 67, 138)
- Nächster Run hat Größe 15. Invariante (1) nicht erfüllt! (15, 17, 31, 67, 138)
- Mergen ergibt: 32, 31, 67, 138. Invariante (2) nicht erfüllt.
- Mergen ergibt: 63, 67, 138. Invarianten erfüllt.

TimSort: Galloping

Lineares Merging:

- Arrays A und B sollen verschmelzt (gemerget) werden.
- Vergleiche $A[0]$ und $B[0]$ und verschiebe den kleineren Wert in das Merge-Array.
- Ineffizient falls immer das selbe Array gewinnt.
- TimSort speichert, wie oft dasselbe Array gewinnt und wechselt in den Galloping-Mode falls dasselbe Array mehr als Min_Galloping mal gewinnt.

$A =$ | 1 | 3 | 4 | 5 | 7 | 8 | $B =$ | 11 | 15 | 16 | 18 | 21 | 23 |

Galloping:

- Falls A Min_Galloping mal gewonnen hat, suche das erste Element $A[i]$ in A das größer als $B[0]$ ist.
- Kopiere $A[0]$ bis $A[i-1]$ in das Merge-Array.
- Suche nach $A[i]$ mittels galoppierender binärer Suche, d.h. vergleiche $B[0]$ mit $A[1], A[3], A[7], \ldots, A[2^k - 1]$ bis $B[0]$ kleiner ist, dann binäre Suche.

Vorteile und Nachteile von TimSort

Vorteile:

- Extrem schnell auf Arrays, die bereits viel Struktur haben (Best-Case: $O(n)$).
- Average- und Worst-Case $O(n \log n)$ wie Mergesort.
- Im Durchschnitt weniger Objekt-Vergleiche als Quicksort.
- Stabil.

Nachteile:

- Zusätzlicher Speicher: Average- und Worst-Case: $O(n)$. (Optimierter Quicksort: $O(\log n)$)
- Im Durchschnitt langsamer als ein optimierter Quicksort, wenn das Vergleichen von Elementen sehr effizient möglich ist

Kompromisslösung: TimSort wird nur auf Objekt-Arrays verwendet. Für das Sortieren von Arrays mit primitiven Datentypen (z.B. int) ist eine spezielle Quicksort Variante als sort in der Hilfsklasse Arrays vorgesehen.

Sortieren: Stabilität (Wiederholung)

Stabiles Sortierverfahren: Ein stabiles Sortierverfahren ist ein Sortieralgorithmus, der die Reihenfolge der Datensätze, deren Sortierschlüssel gleich sind, bewahrt.

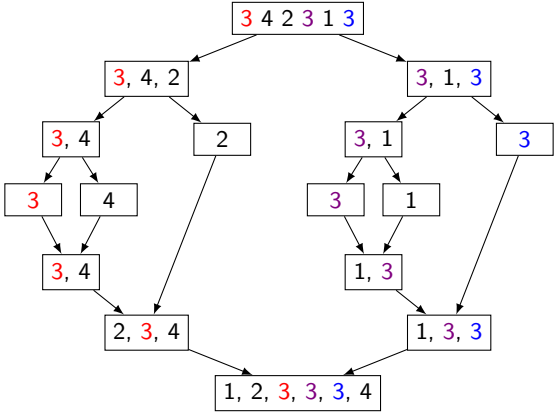
Beispiel:

- Liste von Personendaten mit Abteilungsnummer (innerhalb der Abteilung sortiert):
  3 - Daniel, 4 - Maria, 2 - Paul, 3 - Erika, 1 - Anton, 3 - Sarah
- Sortierung nach Abteilungsnummer mit stabilem Sortierverfahren:
  1 - Anton, 2 - Paul, 3 - Daniel, 3 - Erika, 3 - Sarah, 4 - Maria

Stabilität: Mergesort

Mergesort ist stabil.

Beispiel zur Illustration der Idee mit vereinfachter Ausgangssequenz 3 4 2 3 1 3



Stabilität: Selectionsort

Selectionsort ist nicht stabil.

Beispiel mit vereinfachter Ausgangssequenz 3 4 2 3 1 3

3 4 2 3 1 3

1 4 2 3 3 3

1 2 4 3 3 3

1 2 3 4 3 3

1 2 3 3 4 3

1 2 3 3 3 4

Stabilität: Algorithmen in dieser Vorlesung

Beispiele für stabile Sortierverfahren:

- Insertionsort
- Mergesort
- TimSort

Hinweis: Korrekte Implementierung erforderlich!

Beispiele für instabile Sortierverfahren:

- Selectionsort
- Quicksort

Weitere Bibliotheken

Weitere Beispiele für Bibliotheken

Eclipse Collections

- https://www.eclipse.org/collections/
- Optimierte Sets und Maps, Immutable Collections, Collections für primitive Datentypen, Multimaps, Bimaps, Verschiedene Iterationsstile

Google Guava

- https://github.com/google/guava
- Unterstützt z.B. Multisets, Multimaps, Bimaps

Apache Commons Collections

- https://commons.apache.org/proper/commons-collections/

Brownies Collections

- http://www.magicwerk.org/page-collections-overview.html
- z.B. High Performance Listen (GapList, BigList)

JGraphT

- https://github.com/jgrapht/jgrapht
- Algorithmen und Datenstrukturen für Graphen

Polynomialzeitreduktion

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 3. Mai 2023
Vorlesungsfolien

Effizient lösbare Probleme

Wiederholung aus dem Kapitel „Analyse von Algorithmen“:
Wir bezeichnen ein Problem als effizient lösbar, wenn es in Polynomialzeit gelöst werden kann. Für ein solches Problem existiert ein Algorithmus mit einer Laufzeit $O(n^c)$

- $n =$ Eingabegröße (z.B. in Bits)
- $c =$ konstanter Exponent

Effizient lösbare Probleme werden auch als handhabbar (*tractable*) bezeichnet.

Cobham–Edmonds Annahme:

- Die Annahme, Handhabbarkeit mit Lösbarkeit in Polynomialzeit gleichzusetzten, geht auf Alan Cobham and Jack Edmonds zurück, die das in den 1960er-Jahren vorgeschlagen haben.
- Diese Annahme hat sich weitgehend durchgesetzt und die Informatikforschung der letzten 50 Jahre geprägt.

Edmonds 1963, Cobham 1965

PATHS, TREES, AND FLOWERS

JACK EDMONDS

1. Introduction. A *graph* G for purposes here is a finite set of elements called *vertices* and a finite set of elements called *edges* such that each edge *meets* exactly two vertices, called the *end-points* of the edge. An edge is said to *join* its end-points.

A *matching* in G is a subset of its edges such that no two meet the same vertex. We describe an efficient algorithm for finding in a given graph a matching of maximum cardinality. This problem was posed and partly solved by C. Berge; see Sections 3.7 and 3.8.

THE INTRINSIC COMPUTATIONAL DIFFICULTY OF FUNCTIONS

ALAN COBHAM
I.B.M. Research Center, Yorktown Heights, N. Y., U.S.A.

The subject of my talk is perhaps most directly indicated by simply asking two questions: first, is it harder to multiply than to add? and second, why? I grant I have put the first of these questions rather loosely; nevertheless, I think the answer, ought to be: *yes*. It is the second, which asks for a justification of this answer which provides the challenge.

Diskussion

Rechtfertigung der Annahme:

- In der Praxis haben polynomielle Algorithmen meist kleine Konstanten und kleine Exponenten.
- Das Überwinden der exponentiellen Schranke von Brute-Force-Algorithmen legt meist eine wichtige Struktur des Problems offen.

Ausnahmen/Kritik:

- Einige polynomielle Algorithmen haben große Konstanten und/oder große Exponenten und sind praktisch unbrauchbar.
- Einige Algorithmen mit exponentieller (oder schlechterer) Laufzeit werden oft benutzt, da Worst-Case-Instanzen sehr selten auftreten oder die Instanzen klein genug sind.

Probleme klassifizieren: P or not P?

Ziel: Klassifiziere Probleme in solche, die in Polynomialzeit gelöst werden können und in solche, die nicht in Polynomialzeit gelöst werden können.

Erfordert nachweislich mehr als polynomielle Zeit:

- Hält eine gegebene Turingmaschine nach höchstens k Schritten?
- Gegeben sei eine Brettbelegung für eine n-mal-n Generalisierung von Schach. Kann Schwarz garantiert gewinnen?

Probleme klassifizieren: P or not P?

Kein polynomieller Algorithmus bekannt, aber auch kein Nachweis, dass mehr als polynomielle Zeit erforderlich:

- Gegeben ein Graph G und eine Zahl k, enthält G mindestens k Knoten, die paarweise nicht adjazent sind?
- Lassen sich die Knoten eines gegebenen Graphen mit 3 Farben färben, sodass Paare adjazenter Knoten unterschiedliche Farben haben?
- Ist eine gegebene aussagenlogische Formel erfüllbar?

Für viele fundamentale Problem wurde noch keine Klassifikation (polynomiell/exponnentiell) gefunden.

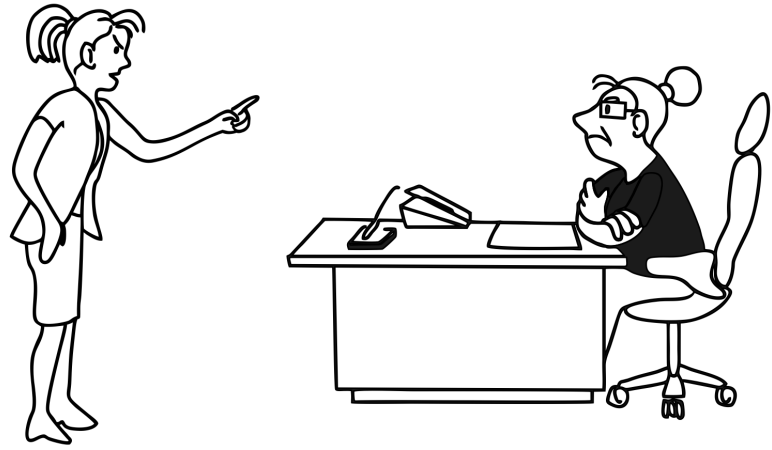
Das ist sehr unzufriedenstellend!

Wie sollen wir damit umgehen, wenn wir ein Problem nicht in Polynomialzeit lösen können?

Keine gute Lösung

“I can’t find an efficient algorithm, I guess I’m just too dumb.”

Wäre die beste Lösung, aber oft nicht möglich



“I can’t find an efficient algorithm, because no such algorithm is possible!”

Gute Kompromisslösung

“I can’t find an efficient algorithm, but neither can all these famous people.”

Äquivalenz bezüglich Lösbarkeit in Polynomialzeit

Dieser Foliensatz: Wir zeigen, dass viele dieser fundamentalen Probleme vom Berechnungsaufwand her im gewissen Sinne äquivalent und eigentlich nur unterschiedliche Erscheinungsformen eines einzigen wirklich schweren Problems sind.

Ein wichtiges Konzept dabei sind so genannte Polynomialzeitreduktionen, mit denen ein Problem in ein anderes „übersetzt“ werden kann.

Ja/Nein-Problem

Einschränkung: Einfachheitshalber beschränken wir uns auf Ja/Nein-Probleme.

Ja/Nein-Problem (*decision problem*):

- Ein Ja/Nein-Problem ist ein Problem, für das die Lösung eine Ja- oder Nein-Antwort ist.
- Im Gegensatz dazu wird bei funktionalen Problemen oder Optimierungsproblemen eine Lösung (Lösungsmenge) oder mehr als ein Bit retourniert.

Beispiel und Unterscheidung zu anderen Problemen:

- Ja/Nein-Problem: Gibt es für einen gewichteten Graphen einen Spannbaum mit Kosten $\leq k$? Das kann mit Ja oder Nein beantwortet werden.
- Funktionales Problem: Finde in einem gewichteten Graphen einen Spannbaum mit Kosten $\leq k$.
- Optimierungsproblem: Finde in einem gewichteten Graphen einen Spannbaum mit minimalen Kosten (MST).

Polynomialzeitreduktionen

Frage: Angenommen, wir können ein Problem in Polynomialzeit lösen. Welche anderen Probleme können wir in Polynomialzeit lösen?

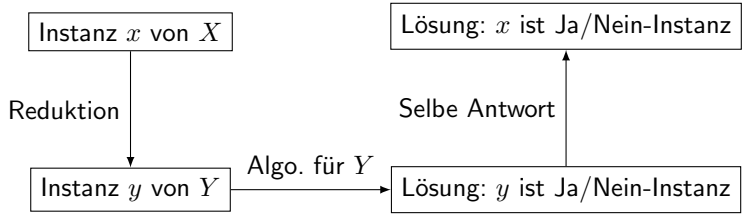
Frage: Wie stellen wir eine Verbindung zu anderen Problemen her?

Reduktion: Problem X kann polynomiell auf ein Problem Y reduziert werden.

Polynomialzeitreduktionen

Informell: Kann X polynomiell auf Y reduziert werden, und wir können Y effizient lösen, dann können wir auch X effizient lösen.

Wir reduzieren dazu eine Instanz x von X auf eine Instanz y vom Y, dann lösen wir y. Ist y eine Ja-Instanz von Y, dann ist x auch eine Ja-Instanz von X.



Polynomialzeitreduktionen

Definition: Eine Polynomialzeitreduktion von Problem X auf Problem Y ist ein Algorithmus R, der für jede Instanz x von X eine Instanz y von Y berechnet, sodass folgendes gilt:

1. x ist eine Ja-Instanz von X genau dann, wenn y eine Ja-Instanz von Y ist.
2. Der Algorithmus R läuft in Polynomialzeit (d.h. es gibt eine Konstante c, sodass der Algorithmus R die Instanz y in Zeit $O(n^c)$ berechnet, wobei n die Eingabegröße für x ist).

Notation: Wir schreiben $X \leq_P Y$, falls es eine Polynomialzeitreduktion von X auf Y gibt. In dem Fall sagen wir auch, dass „X auf Y polynomiell reduzierbar ist“.

Hinweis: Falls $X \leq_P Y$, dann können wir auch sagen, dass „Y mindestens so schwer ist wie X“.

Lösung durch Reduktion

Idee: Wir lösen ein Problem durch Polynomialzeitreduktion auf ein anderes Problem, das handhabbar ist.

Es gilt: Wenn $X \leq_P Y$ und Y kann in polynomieller Zeit gelöst werden, dann kann auch X in polynomieller Zeit gelöst werden.

Beweis:

- Sei R ein Algorithmus der X auf Y reduziert, der in Zeit $O(n^a)$ läuft (für Instanzen von X der Größe n, $a \geq 1$).
- Sei A ein Algorithmus der Y in Zeit $O(n^b)$ löst (für Instanzen von Y der Größe n, $b \geq 1$).
- Sei nun x eine Instanz von X der Größe n.
- Wir wenden R an und erhalten in $O(n^a)$ Zeit eine Instanz y von Y.
- die Größe von y ist höchstens $O(n^a)$, da y in Zeit $O(n^a)$ erzeugt wurde.
- Wir lösen y mit A in Zeit $O((n^a)^b) = O(n^{ab})$.
- Insgesamt haben wir x in Zeit $O(n^a) + O(n^{ab}) = O(n^{ab})$ gelöst, das ist polynomiell in n. $\square$

Nicht-Handhabbarkeit zeigen

Idee: Wir zeigen, dass ein Problem nicht handhabbar ist, in dem wir ein anderes nicht handhabbares Problem darauf reduzieren.

Es gilt: Wenn $X \leq_P Y$ und X kann nicht in polynomieller Zeit gelöst werden, dann kann Y nicht in polynomieller Zeit gelöst werden.

Beweis:

- Angenommen Y wäre in polynomieller Zeit lösbar.
- $X \leq_P Y$ bedeutet, dass wir X in polynomieller Zeit auf Y reduzieren können.
- Da die Reduktion $X \leq_P Y$ und das Lösen von Y in polynomieller Zeit ausführbar sind, ist X in polynomieller Zeit lösbar. Das ist ein Widerspruch zur Annahme. $\square$

Äquivalenz: Wenn $X \leq_P Y$ und $Y \leq_P X$, dann schreiben wir $X \equiv_P Y$.

Transitivität

Es gilt: Ist $X \leq_P Y$ und $Y \leq_P Z$, dann folgt daraus $X \leq_P Z$.

Beweis:

- Sei R_1 der Algorithmus für $X \leq_P Y$, der in $O(n^a)$ läuft, $a \geq 1$.
- Sei R_2 der Algorithmus für $Y \leq_P Z$, der in $O(n^b)$ läuft $b \geq 1$.
- Sei n die Größe von x
- Dann benötigt der Aufruf von R_1 auf x höchstens $O(n^a)$ Zeit.
- Weiters erzeugt R_1 eine Ausgabe x' mit einer Größe von $O(n^a)$.
- Daher benötigt der darauffolgende Aufruf von R_2 auf x' eine Laufzeit von $O(n^{ab})$.
- Damit ist aber die Reduktion von X auf Z in polynomieller Zeit möglich. $\square$

Polynomialzeitreduktionen angeben

- Im Folgenden werden wir einige Beispiele von Polynomialzeitreduktionen betrachten.
- Wenn wir eine solche Reduktion angeben, müssen wir zwei Eigenschaften sicherstellen (siehe die Definition einer Polynomialzeitreduktion)
  1. die Reduktion ist korrekt, d.h., Ja-Instanzen werden auf Ja-Instanzen abgebildet, und Nein-Instanzen auf Nein-Instanzen
  2. die Reduktion ist polynomiell, d.h., die Reduktion kann in Polynomialzeit ausgeführt werden.
- In manchen Fällen ist Korrektheit einfach zu zeigen, in anderen Fällen die Polynomialzeit.

Reduktion durch einfache Äquivalenz

Grundlegende Reduktionsstrategien:

- Reduktion durch einfache Äquivalenz.
- Reduktion eines Spezialfalls auf den allgemeinen Fall.
- Reduktion durch Kodierung mit Gadgets.

Independent Set

Independent Set: Ein Independent Set eines Graphen $G = (V, E)$ ist eine Teilmenge $S \subseteq V$, in der es keine zwei adjazenten Knoten gibt.

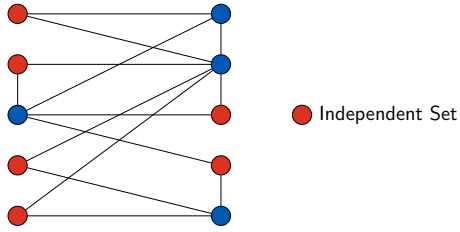
INDEPENDENT SET: Gegeben sei ein Graph $G = (V, E)$ und eine ganze Zahl k. Gibt es ein Independent Set S, sodass $|S| \geq k$ gilt?

Wichtig: Die Zahl k ist Teil der Eingabe und keine Konstante.

Hinweis: Ja/Nein-Probleme werden ab jetzt mit dieser Schreibweise (INDEPENDENT SET) gekennzeichnet.

Independent Set

Independent Set: Gegeben sei ein Graph $G = (V, E)$ und eine ganze Zahl k. Gibt es ein Independent Set S, sodass $|S| \geq k$ gilt?

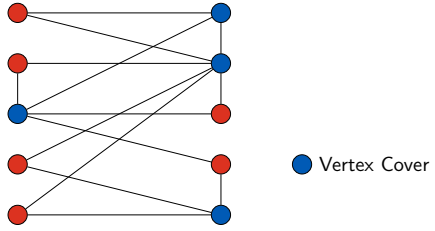


Beispiel: Existiert ein Independent Set der Größe ≥ 6? Ja.
Beispiel: Existiert ein Independent Set der Größe ≥ 7? Nein.

Vertex Cover

Vertex Cover: Ein Vertex Cover eines Graphen $G = (V, E)$ ist eine Menge $S \subseteq V$, sodass jede Kante des Graphen zu mindestens einem Knoten aus S inzident ist.

VERTEX COVER: Gegeben sei ein Graph $G = (V, E)$ und eine ganze Zahl k. Gibt es ein Vertex Cover S von G, sodass $|S| \leq k$?

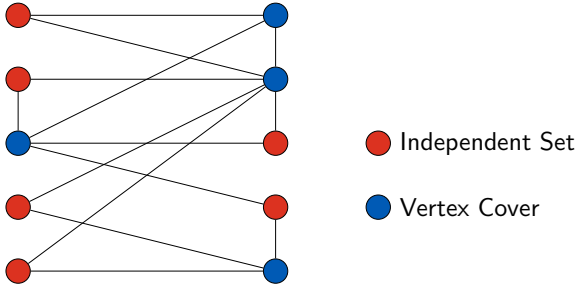


Beispiel: Existiert ein Vertex Cover der Größe ≤ 4? Ja.
Beispiel: Existiert ein Vertex Cover der Größe ≤ 3? Nein.

Vertex Cover und Independent Set

Wir wollen zeigen, dass VERTEX COVER $\equiv_P$ INDEPENDENT SET. Dazu zeigen wir zuerst:

Konversionslemma: Sei $G = (V, E)$ ein Graph und $S \subseteq V$ und $C = V - S$. S ist ein Independent Set von G genau dann wenn C ein Vertex Cover von G ist.



Vertex Cover und Independent Set

Wir wollen zeigen, dass VERTEX COVER $\equiv_P$ INDEPENDENT SET. Dazu zeigen wir zuerst:

Konversionslemma: Sei $G = (V, E)$ ein Graph und $S \subseteq V$ und $C = V - S$. S ist ein Independent Set von G genau dann wenn C ein Vertex Cover von G ist.

Beweis: $\Rightarrow$

- Betrachte eine beliebige Kante $(u, v) \in E$. Wir wollen zeigen, dass mindestens einer der beiden Knoten u, v in C liegt.
- Weil S ein Independent Set ist, muss mindestens einer der beiden Knoten u, v nicht in S liegen. Also liegt mindestens einer der beiden Knoten in C.
- Daher ist C ein Vertex Cover. $\square$

Vertex Cover und Independent Set

Wir wollen zeigen, dass Vertex Cover $\equiv_P$ Independent Set. Dazu zeigen wir zuerst:

Konversionslemma: Sei $G = (V, E)$ ein Graph und $S \subseteq V$ und $C = V - S$. S ist ein Independent Set von G genau dann wenn C ein Vertex Cover von G ist.

Beweis: $\Leftarrow$

- Betrachte zwei Knoten $u \in S$ und $v \in S$. Wir wollen zeigen, dass u und v nicht adjazent sind.
- Aus $u \in S$ und $v \in S$ folgt $u \notin C$ und $v \notin C$.
- u und v können nicht adjazent sein, ansonsten wäre C kein Vertex Cover (weil es die Kante (u, v) nicht überdeckt).
- Also ist S ein Independent Set. $\square$

Also gilt das Lemma.

Vertex Cover und Independent Set

Es gilt: Vertex Cover $\equiv_P$ Independent Set.

Beweis:
Zuerst zeigen wir Vertex Cover $\leq_P$ Independent Set.

- Sei (G, k) eine Instanz von Vertex Cover. Sei n die Anzahl der Knoten von G.
- In Polynomialzeit generieren wir $(G, n - k)$, eine Instanz von Independent Set.
- Die Reduktion ist *korrekt*, da G ein Vertex Cover $\leq k$ hat genau dann wenn G ein Independent Set der Größe $\geq n - k$ hat (folgt aus dem Konversionslemma).
- Die Reduktion ist klarerweise *polynomiell*, da sie ja nur n durch $n - k$ ersetzten braucht. $\square$

Vertex Cover und Independent Set

Es gilt: VERTEX COVER $\equiv_P$ INDEPENDENT SET.

Beweis:

Weiters zeigen wir INDEPENDENT SET $\leq_P$ VERTEX COVER

- Sei (G, k) eine Instanz von INDEPENDENT SET. Sei n die Anzahl der Knoten von G.
- In Polynomialzeit generieren wir $(G, n-k)$ eine Instanz von VERTEX COVER.
- Die Reduktion ist korrekt, da G ein Independent Set $\geq k$ hat genau dann wenn G ein Vertex Cover Set der Größe $\leq n-k$ hat (folgt aus dem Konversionslemma).
- Die Reduktion ist klarerweise *polynomiell*, da sie ja nur n durch $n-k$ ersetzten braucht. $\square$

Wir haben also beide Richtungen gezeigt, und es folgt die Äquivalenz.

Beispiel: Spannbäume und Nicht-Blockierer

NICHT-BLOCKIERER: Gegeben ist ein Graph $G = (V, E)$ mit reellwertigen Kantengewichten $c_e = c_{uv} = c_{vu}$ für $e = (u, v) \in E$. Ein *Nicht-Blockierer* ist eine Teilmenge der Kanten $N \subseteq E$, sodass es für alle Knotenpaare $u, v \in V$ einen u-v-Pfad in G gibt, der keine Kante aus N enthält.
Die *Kosten* des Nicht-Blockierers ist gegeben durch $\sum_{e \in N} c_e$.
Ein *maximaler Nicht-Blockierer* ist ein Nicht-Blockierer mit größten Kosten.

MNB: Gegeben ist ein gewichteter Graph G und eine Zahl k. Besitzt G einen Nicht-Blockierer mit Kosten $\geq k$?

Ist das Problem MNB in Polynomialzeit lösbar?

Beispiel: Spannbäume und Nicht-Blockierer

Wir zeigen, dass MNB in Polynomialzeit lösbar ist, indem wir es auf ein bekanntes Problem reduzieren.

SPANNBAUM: Gegeben ist ein Graph $G = (V, E)$ mit reellwertigen Kantengewichten $c_e = c_{uv} = c_{vu}$ für $e = (u, v) \in E$. Ein *Spannbaum* ist eine Teilmenge der Kanten $T \subseteq E$, sodass $G_T = (V, T)$ ein Baum ist.
Die *Kosten* des Baumes ist gegeben durch $\sum_{e \in T} c_e$.
Ein *minimaler Spannbaum* ist ein Spannbaum mit kleinsten Kosten.

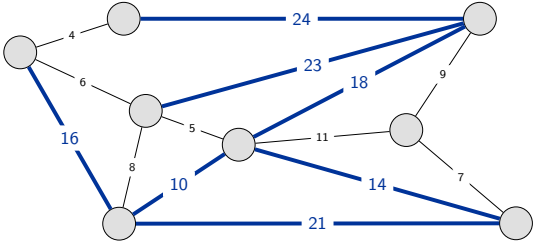
MST^*: Gegeben ist ein gewichteter Graph G und eine Zahl k. Besitzt G einen Spannbaum mit Kosten $\leq k$?

Wir wissen aus dem Kapitel „Greedy-Algorithmen“, dass MST^* in Polynomialzeit gelöst werden kann.

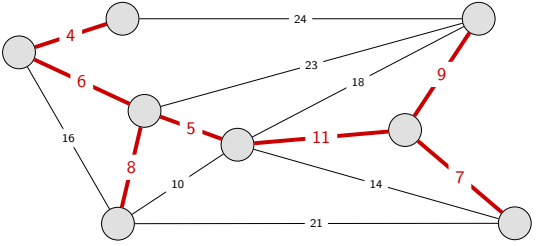
Beispiel: Spannbäume und Nicht-Blockierer

Konversionslemma: Sei $G = (V, E)$ ein gewichteter Graph, $N \subseteq E$ und $T = E - N$. Dann ist N ein maximaler Nicht-Blockierer genau dann wenn T ein minimaler Spannbaum ist. Die Kosten von N sind genau $K := \sum_{e \in E} c_e$ minus der Kosten von T.

Wir lassen den sehr einfachen Beweis als Übung.



N, Kosten $= 126$



T, Kosten $= 50$

Beispiel: Spannbäume und Nicht-Blockierer

Konversionslemma: Sei $G = (V, E)$ ein gewichteter Graph, $N \subseteq E$ und $T = E - N$. Dann ist N ein maximaler Nicht-Blockierer genau dann wenn T ein minimaler Spannbaum ist. Die Kosten von N sind genau $K := \sum_{e \in E} c_e$ minus der Kosten von T.

Wir lassen den sehr einfachen Beweis als Übung.

Es gilt: $\mathrm{MNB} \equiv_P \mathrm{MST}^*$

Beweis:

- $\mathrm{MNB} \leq_P \mathrm{MST}^*$: Wir reduzieren eine Instanz (G, k) von MNB auf die Instanz $(G, K - k)$ von MST^*.
- $\mathrm{MST}^* \leq_P \mathrm{MNB}$: Wir reduzieren eine Instanz (G, k) von MST^* auf die Instanz $(G, K - k)$ von MNB. $\square$

Es folgt daher: MNB ist in Polynomialzeit lösbar.

Reduktion eines Spezialfalls auf den allgemeinen Fall.

Grundlegende Reduktionsstrategien:

- Reduktion durch einfache Äquivalenz.
- Reduktion eines Spezialfalls auf den allgemeinen Fall.
- Reduktion durch Kodierung mit Gadgets.

Set Cover (Mengenüberdeckungsproblem)

Set Cover: Gegeben sei eine Menge U von Elementen und eine Menge $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ von Teilmengen von U. Ein Set Cover ist eine Teilmenge $\mathcal{C} \subseteq \mathcal{S}$, also eine Menge von Mengen, deren Vereinigung U entspricht. $\mathcal{C}$ ist ein Set Cover von $\mathcal{S}$.

SET COVER: Gegeben sei eine Menge U von Elementen, eine Menge $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$ von Teilmengen von U und eine ganze Zahl k. Existiert eine Teilmenge $\mathcal{C} \subseteq \mathcal{S}$ mit $|\mathcal{C}| \leq k$, sodass die Vereinigung von $\mathcal{C}$ gleich U ist?

Beispielhafte Anwendung:

- m verfügbare Softwarekomponenten.
- Menge U von n Eigenschaften, die unser Softwaresystem haben sollte.
- Die i-te Softwarekomponente bietet eine Menge $S_i \subseteq U$ von Eigenschaften an.
- Ziel: Erreiche alle n Eigenschaften mit maximal k Komponenten.

Set Cover: Beispiel

Beispiel:

$U = \{1, 2, 3, 4, 5, 6, 7\}$
$k = 2$

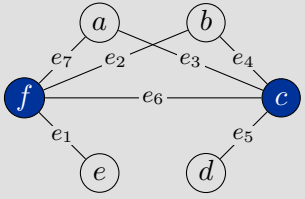
$S_1 = \{3, 7\}$	$S_4 = \{2, 4\}$
$S_2 = \{3, 4, 5, 6\}$	$S_5 = \{5\}$
$S_3 = \{1\}$	$S_6 = \{1, 2, 6, 7\}$

Vertex Cover auf Set Cover reduzieren

Behauptung: Vertex Cover $\leq_P$ Set Cover.
Beweis: Gegeben sei eine Vertex Cover Instanz (G, k) mit $G = (V, E)$.
Wir konstruieren eine Instanz von Set Cover.

- $k = k$,
- $U = E$,
- Für jedes $v \in V$ erzeugen wir eine Menge $S_v = \{e \in E : \ e \text{ inzident zu } v\}$
- $\mathcal{S}$ ist die Menge aller S_v für $v \in V$.
- Die Reduktion ist klarerweise *polynomiell*.

Vertex Cover



$k = 2$

Set Cover
$U = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$ $k = 2$
$S_a = \{e_3, e_7\}$, $S_b = \{e_2, e_4\}$
$S_c = \{e_3, e_4, e_5, e_6\}$, $S_d = \{e_5\}$
$S_e = \{e_1\}$, $S_f = \{e_1, e_2, e_6, e_7\}$

Vertex Cover auf Set Cover reduzieren

Behauptung: Vertex Cover $\leq_P$ Set Cover.
Beweis: Gegeben sei eine Vertex Cover Instanz (G, k) mit $G = (V, E)$.
Wir konstruieren eine Instanz von Set Cover.

- $k = k$,
- $U = E$,
- Für jedes $v \in V$ erzeugen wir eine Menge $S_v = \{e \in E : \ e \text{ inzident zu } v\}$
- $\mathcal{S}$ ist die Menge aller S_v für $v \in V$.
- Die Reduktion ist klarerweise *polynomiell*.
- *Korrektheit:* G hat ein Vertex Cover der Größe $\leq k$ genau dann wenn $\mathcal{S}$ ein Set Cover der Größe $\leq k$ hat.
- $\Rightarrow$: Sei $C = \{v_1, \ldots, v_k\} \subseteq V$ ein Vertex Cover von G. Dann ist $\mathcal{C} = \{S_{v_1}, \ldots, S_{v_k}\}$ ein Set Cover von $\mathcal{S}$.
- $\Leftarrow$: Sei $\mathcal{C} = \{S_{v_1}, \ldots, S_{v_k}\}$ ein Set Cover von $\mathcal{S}$. Dann ist $C = \{v_1, \ldots, v_k\} \subseteq V$ ein Vertex Cover von G. $\square$

Bemerkung

Nicht jede Set Cover Instanz kann durch die Reduktion von Vertex Cover entstehen.

Daher sprechen wir hier von einer „Reduktion eines Spezialfalls auf den allgemeinen Fall“.

Reduktion mit „Gadgets“

Grundlegende Reduktionsstrategien:

- Reduktion durch einfache Äquivalenz.
- Reduktion eines Spezialfalls auf den allgemeinen Fall.
- Reduktion mit „Gadgets“.

Erfüllbarkeit (satisfiability)

Literal: Eine boolesche Variable oder ihre Negation: x_i oder $\overline{x_i}$

Klausel: Eine Disjunktion (logisches Oder) von Literalen:
Beispiel: $C_j = x_1 \vee \overline{x_2} \vee x_3$

Konjunktive Normalform (KNF): Eine aussagenlogische Formel Φ, bei der Klauseln konjunktiv (logisches Und) verknüpft werden.
Beispiel: $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$

Wahrheitsbelegung (*truth assignment*): Eine Wahrheitsbelegung ist eine Funktion f, die jeder Variable einen Wahrheitswert *true* oder *false* zuordnet.

Erfüllen einer Formel: Eine Wahrheitsbelegung f erfüllt eine KNF-Formel Φ, falls jede Klausel von Φ mindestens eine Variable x mit $f(x) =$ *true* oder eine negierte Variable $\overline{x}$ mit $f(x) =$ *false* enthält.

Das Erfüllbarkeitsproblem (SAT)

SAT (*satisfiability*): Gegeben ist eine KNF-Formel Φ. Gibt es eine Wahrheitsbelegung, die Φ erfüllt?

3-SAT: SAT, bei dem jede Klausel genau 3 Literale enthält.

- *Jedes Literal muss sich auf eine unterschiedliche Variable beziehen.*

Beispiel: $(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$
Erfüllende Wahrheitsbelegung: $f(x_1) = \textit{true}$, $f(x_2) = \textit{true}$, $f(x_3) = \textit{false}$.

Bedeutung von SAT

Das Erfüllbarkeitsproblem (SAT) ist in zweierlei Hinsicht von großer Bedeutung:

1. Für SAT gibt es mächtige heuristische Algorithmen (SAT-solver), die große „strukturierte“ Instanzen lösen können.
   → Daher ist SAT ein beliebtes Problem um andere Probleme darauf zu reduzieren (*Lösung durch Reduktion*).
2. Andererseits wird SAT für allgemeine Instanzen als nicht-handhabbar angesehen (SAT ist „NP-vollständig“, was wir auf den nächsten Folien erläutern werden).
   → Daher ist SAT ein beliebtes Problem, das auf andere Probleme reduziert wird, um deren *Nicht-Handhabbarkeit zu zeigen*.

3-SAT auf INDEPENDENT SET reduzieren

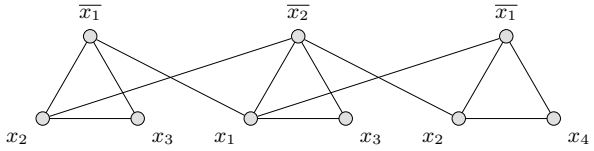
Behauptung: 3-SAT $\leq_P$ INDEPENDENT SET.

Beweis: Gegeben sei eine Instanz Φ von 3-SAT mit k Klauseln. Wir konstruieren eine Instanz (G, k) von INDEPENDENT SET.

- G enthält 3 Knoten für jede Klausel (einen für jedes Literal).
- Verbinde 3 Literale in einer Klausel zu einem Dreieck.
- Verbinde ein Literal mit jeder seiner Negationen.

Die Reduktion ist klarerweise *polynomiell*.

G



$k = 3$

$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

3-SAT auf INDEPENDENT SET reduzieren

Korrektheit: G enthält ein Independent Set der Größe k genau dann wenn Φ erfüllbar ist.

Beweis: ($\Rightarrow$) Sei S ein Independent Set der Größe k.

- S enthält genau einen Knoten pro Dreieck (S kann höchstens einen Knoten pro Dreieck enthalten ansonsten ist S nicht unabhängig; S muss mindestens einen Knoten pro Dreieck enthalten, da $|S| = k$ und es k Dreiecke gibt)
- Wir konstruieren eine Wahrheitsbelegung der Variablen indem wir $x = \textit{true}$ setzen, falls $x \in S$ und $x = \textit{false}$ setzen, falls $\overline{x} \in S$.
- Wegen der Kanten zwischen den Dreiecken kann es zu keinen widersprüchlichen Belegungen ein und der selben Variable kommen.
- Wir setzen die übrigen Variablen beliebig auf *true* oder *false*.
- Diese Wahrheitsbelegung erfüllt alle Klauseln. $\square$

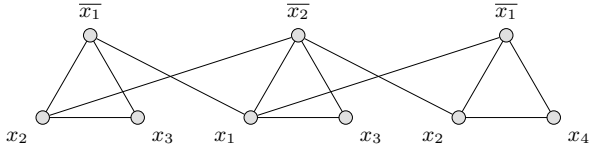
3-Sat auf Independent Set reduzieren

Korrektheit: G enthält ein Independent Set der Größe k genau dann wenn Φ erfüllbar ist.

Beweis: ($\Leftarrow$) Gegeben sei eine Wahrheitsbelegung f die Φ erfüllt.

- Wir konstruieren ein Independent Set S indem wir von jedem Dreieck einen Knoten ℓ mit $\ell = x$ und $f(x) = \textit{true}$ oder einen Knoten ℓ mit $\ell = \overline{x}$ und $f(x) = \textit{false}$ wählen. (So einen Knoten ℓ gibt es immer, da f erfüllend)
- S ist unabhängig, weil die Kanten zwischen den Dreiecken jeweils zwischen einer Variable und ihrer Negation verlaufen.
- $|S| = k$, weil wir von jedem Dreieck einen Knoten wählen. $\square$

G



$k = 3$

$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

Rückblick

Grundlegende Reduktionsstrategien:

- Einfache Äquivalenz:
  Independent Set $\equiv_P$ Vertex Cover.
- Spezieller Fall auf allgemeinen Fall:
  Vertex Cover $\leq_P$ Set Cover.
- Kodierung mit Gadgets:
  3-Sat $\leq_P$ Independent Set.

Transitivität: Wenn $X \leq_P Y$ und $Y \leq_P Z$, dann $X \leq_P Z$.
Beweisidee: Verbinde zwei Algorithmen.

Beispiel: 3-Sat $\leq_P$ Independent Set $\leq_P$ Vertex Cover $\leq_P$ Set Cover.

Optimierungsprobleme

Ja/Nein-Problem: Existiert ein Vertex Cover der Größe $\leq k$?
Optimierungsproblem: Finde kleinstes Vertex Cover.

Klar:

- Ja/Nein-Problem kann mit dem Optimierungsproblem gelöst werden.
- Berechne kleinstes Vertex Cover C und antworte „Ja“ falls $|C| \leq k$ ist.

Umgekehrte Richtung: Löse Optimierungsproblem mittels (mehrmaligem) Lösen des Ja/Nein-Problems.

Optimierungsproblem für Vertex Cover lösen

Ausgangslage: Es existiert ein Algorithmus VC(G,k), der das Ja/Nein-Problem für ein Vertex Cover der Größe $\leq k$ löst.

Wir möchten mittels VC(G,k) ein kleinstes Vertex Cover finden.

Dazu verwenden wir die folgenden Eigenschaften, die für jeden Graphen $G = (V, E)$ gelten:

1. Sei $v \in V$. Falls C ein Vertex Cover von $G - v$ ist, dann ist $C \cup \{v\}$ ein Vertex Cover von G.
2. Falls G ein Vertex Cover der Grösse $k \geq 1$ besitzt, dann gibt es ein $v \in V$ sodass $G - v$ ein Vertex Cover der Größe $k - 1$ besitzt.

Optimierungsproblem für Vertex Cover lösen

Der Algorithmus OptVC(G) berechnet ein kleinstes Vertex Cover von G mittels mehrmaligen Aufrufs von VC(G,k).

```
OptVC(G):
for k ← 0 bis n − 1
    if VC(G,k)
        return FindVC(G,k)

FindVC(G,k):
if k = 0
    return ∅
else
    foreach v ∈ V(G)
        if VC(G − v,k − 1)
            return {v} ∪ FindVC(G − v,k − 1)
```

Komplexität: Insgesamt wird VC(G,k) höchstens $O(n) + O(n^2) = O(n^2)$ mal aufgerufen.

Optimierungsprobleme

- Analog kann für viele andere Optimierungsprobleme vorgegangen werden.
- Das rechtfertigt unseren Fokus auf Ja/Nein-Probleme.

Definition von NP

NP-Probleme

NP-Probleme: Ein Ja/Nein-Problem ist ein NP-Problem, falls wir Ja-Instanzen mit Hilfe eines Zertifikats effizient überprüfen können.

Zertifikat: Ein Zertifikat t für eine Instanz x ist ein beliebiger Input dessen Größe m polynomiell in der Größe n von x beschränkt ist, das heißt $m \leq p(n)$ gilt für ein Polynom p.

Zertifizierer: Einen Polynomialzeitalgorithmus $C(x, t)$, der Ja-Instanzen x mit Hilfe von Zertifikaten t überprüft, nennt man Zertifizierer.

Anmerkung zur Notation: NP steht für „nicht-deterministisch polynomielle“ Zeit.

Für ein NP-Problem X sagen wir auch „X ist in NP“.

NP-Probleme

Genauer gesagt, der Zertifizierer $C(x, t)$ soll folgende Eigenschaften haben:

- Für jede Ja-Instanz x gibt es ein Zertifikat t (polynomieller Länge), welches den Zertifizierer zum akzeptieren bringt.
  („Zertifizierer kann überzeugt werden“)
- Für keine Nein-Instanz x gibt es ein Zertifikat t, welches den Zertifizierer zum akzeptieren bringt.
  („Zertifizierer kann nicht ausgetrickst werden“)

NP-Probleme

Beispiel VERTEX COVER:

- Für eine Ja-Instanz (G, k) nehmen wir ein Vertex Cover S der Größe $\leq k$ als Zertifikat.
- Wir können in Polynomialzeit überprüfen, ob S tatsächlich ein Vertex Cover der Größe $\leq k$ ist.
  - Die Größe von S lässt sich in Polynomialzeit überprüfen.
  - Danach überprüft man für jede Kante in G, ob diese zumindest einen Endpunkt in S hat. Das lässt sich wiederum in Polynomialzeit überprüfen.

Schlussfolgerung: VERTEX COVER ist ein NP-Problem
(anders gesagt, VERTEX COVER ist in NP).

Beispiel für Zertifizierer und Zertifikate: SAT

SAT: Gegeben sei eine Formel Φ in konjunktiver Normalform. Ist diese Formel erfüllbar?

Zertifikat: Eine Wahrheitsbelegung f für die n booleschen Variablen die Φ erfüllt.

Zertifizierer: Überprüfe ob f die Formel Φ erfüllt.

Beispiel:

$$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)\,;$$

Instanz s

$$f(x_1) = 1,\ f(x_2) = 1,\ f(x_3) = 0,\ f(x_4) = 1$$

Zertifikat t

Schlussfolgerung: SAT ist in NP.

Beispiel für Zertifizierer und Zertifikate: Hamiltonkreisproblem

HAM-CYCLE: Gegeben sei ein ungerichteter Graph G. Existiert ein Kreis C in G der alle Knoten von G genau einmal enthält?

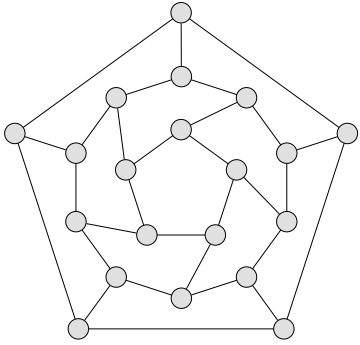
So ein Kreis wird als *Hamiltonkreis* bezeichnet.

Zertifikat: Eine Hamiltonkreis.

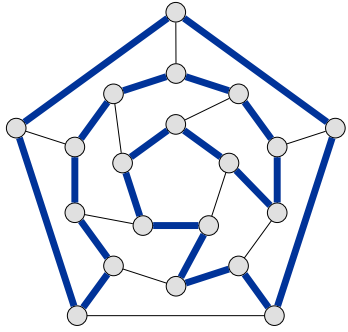
Zertifizierer: Überprüfe, ob der Hamiltonkreis jeden Knoten in V genau einmal enthält und dass es eine Kante zwischen jedem Paar von direkt aufeinander folgenden Knoten in dem Hamiltonkreis und auch vom ersten zum letzten Knoten gibt.

Beispiel für Zertifizierer und Zertifikate: Hamiltonkreisproblem

Beispiel:



Instanz s



Zertifikat t

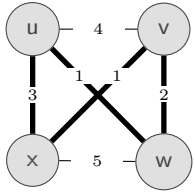
Schlussfolgerung: HAM-CYCLE ist in NP.

Travelling Salesman Problem (TSP)

Travelling Salesman Problem (TSP):

- Man sucht eine Reihenfolge für den Besuch mehrerer Orte (eine „Tour“), sodass die gesamte Reisestrecke eines Handlungsreisenden möglichst kurz ist (Minimierungsproblem).
- Die Strecke zwischen zwei Orten u und v ist als reelle Zahl $d(u, v)$ gegeben.
- Der erste Ort sollte gleich dem letzten Ort sein.

Beispiel: 4 Orte, minimale Tour der Länge 7.



Travelling Salesman Problem (TSP)

Wir bezeichnen mit TSP^* das Ja/Nein-Problem, das nach einer Tour der Länge $\leq k$ fragt.

Reduktion: $\textsc{Ham-Cycle} \leq_P \mathrm{TSP}^*$.

Beweis:

- Gegeben sei eine Instanz $G = (V, E)$ von $\textsc{Ham-Cycle}$.
- Erzeuge n Städte mit einer Distanzfunktion

$$d(u,v) = \begin{cases} 1 & \text{wenn} (u,v) \in E \\ 2 & \text{wenn} (u,v) \notin E \end{cases}$$

- TSP-Instanz besitzt eine Tour der Länge n genau dann, wenn G einen Hamiltonkreis besitzt. $\square$

P, NP

P: Ja/Nein-Probleme, für die polynomielle Algorithmen existieren.

NP: Ja/Nein-Probleme, für die polynomielle Zertifizierer existieren.

Es gilt: P $\subseteq$ NP.

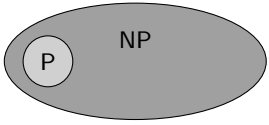
Beweis: Wir betrachten ein beliebiges Problem X in P.

- Nach Definition existiert ein Polynomialzeit-Algorithmus $A(s)$, der X löst.
- Zertifikat: $t = \emptyset$, Zertifizierer $C(s, t) = A(s)$. $\square$

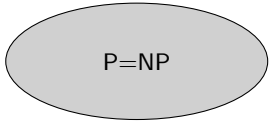
P = NP?

Gilt P = NP? [Cook 1971, Levin 1973]

- Ist das Ja/Nein-Problem so leicht wie das Zertifizierunsgproblem?
- Für die Beantwortung der Frage ist 1 Million US Dollar ausgeschrieben (Clay Mathematics Institute).



Falls P $\neq$ NP



Falls P = NP

P = NP?

Gilt P = NP? [Cook 1971, Levin 1973]

Falls ja: Effiziente Algorithmen für Vertex Cover, Ham-Cycle, TSP*, Sat, ...

Falls nein: Keine effizienten Algorithmen für Vertex Cover, Ham-Cycle, TSP*, Sat, ...

Vorherrschende Meinung zu P = NP: Wahrscheinlich „nein“

NP-Vollständigkeit

NP-Vollständigkeit

Definition: Ein Ja/Nein-Problem Y ist NP-schwer, falls für jedes Problem X in NP gilt, dass $X \leq_p Y$.

Das heißt, jedes NP-Problem X kann in Polynomialzeit auf Y reduziert werden.

NP-schwere Probleme sind also gewissermaßen „mindestens so schwer“ wie alle Probleme in NP.

Definition: Ein Problem Y ist NP-vollständig, falls es sowohl in NP liegt als auch NP-schwer ist.

Die NP-vollständigen Probleme sind also gewissermaßen die „schwersten“ Probleme in NP.

Nach dieser Definition können nur Ja/Nein-Probleme NP-vollständig sein.

NP-Vollständigkeit

Theorem: Sei Y ein NP-vollständiges Problem.
Y ist in polynomieller Zeit lösbar genau dann, wenn P $=$ NP.

Beweis: ($\Leftarrow$) Wenn P $=$ NP, dann kann Y in polynomieller Zeit gelöst werden, da Y sich in NP befindet.
Beweis: ($\Rightarrow$) Angenommen, Y kann in polynomieller Zeit gelöst werden.

- Sei X ein beliebiges Problem in NP. Da $X \leq_p Y$, können wir X in Polynomialzeit lösen. Das impliziert NP $\subseteq$ P.
- Wir wissen bereits, dass P $\subseteq$ NP. Daher P $=$ NP. $\square$

Gibt es ein NP-vollständiges Problem?

Allgemeine Überlegung:

- Das ist nicht von vorne herein klar.
- Es könnte z.B. mehrere schwerste NP-Probleme geben, die nicht jeweils auf einander reduzierbar sind.

Theorem: SAT ist NP-vollständig. [Cook 1971, Levin 1973]

Cook 1971, Levin 1973

The Complexity of Theorem-Proving Procedures

Stephen A. Cook

University of Toronto

Summary

It is shown that any recognition problem solved by a polynomial time-bounded nondeterministic Turing machine can be "reduced" to the problem of finding a certain recursive set of strings on this alphabet, and we are interested in the problem of finding a good lower bound on its possible recognition times. We provide no such lower bound here, but theorem 1 will

ПРОБЛЕМЫ ПЕРЕДАЧИ ИНФОРМАЦИИ

Том IX *1973* *Вып. 3*

КРАТКИЕ СООБЩЕНИЯ

УДК 519.14

УНИВЕРСАЛЬНЫЕ ЗАДАЧИ ПЕРЕБОРА

Л. А. Левин

В статье рассматривается несколько известных массовых задач «переборного типа» и доказывается, что эти задачи можно решать лишь за такое время, за которое можно решать вообще любые задачи указанного типа.

NP-Vollständigkeit nachweisen

Anmerkung: Sobald wir ein erstes Problem als NP-vollständig nachgewiesen haben, fallen die anderen wie Dominosteine.

Rezept um die NP-Vollständigkeit eines Problems Y nachzuweisen:

- Schritt 1. Zeige dass Y in NP ist.
- Schritt 2. Wähle ein NP-vollständiges Problem X.
- Schritt 3. Beweise, dass $X \leq_p Y$.

Rechtfertigung: Wenn X ein NP-vollständiges Problem ist und Y ein Problem in NP mit der Eigenschaft $X \leq_p Y$, dann ist Y NP-vollständig.

NP-Vollständigkeit nachweisen

Rechtfertigung: Wenn X ein NP-vollständiges Problem ist und Y ein Problem in NP mit der Eigenschaft $X \leq_p Y$, dann ist Y NP-vollständig.

Beweis: Sei W ein beliebiges Problem in NP.
Dann gilt $W \leq_p X \leq_p Y$.

□ *durch Definition von NP-vollständig* □ *durch Annahme*

- Durch Transitivität, $W \leq_p Y$.
- Daher ist Y NP-vollständig. □

[Karp 1972] Karp hat mit einem einflussreichen Artikel wesentlich zur Verbreitung der Theorie der NP-Vollständigkeit beigetragen. Er hat 1985 dafür den Turingpreis erhalten.

Karp 1972

REDUCIBILITY AMONG COMBINATORIAL PROBLEMS†

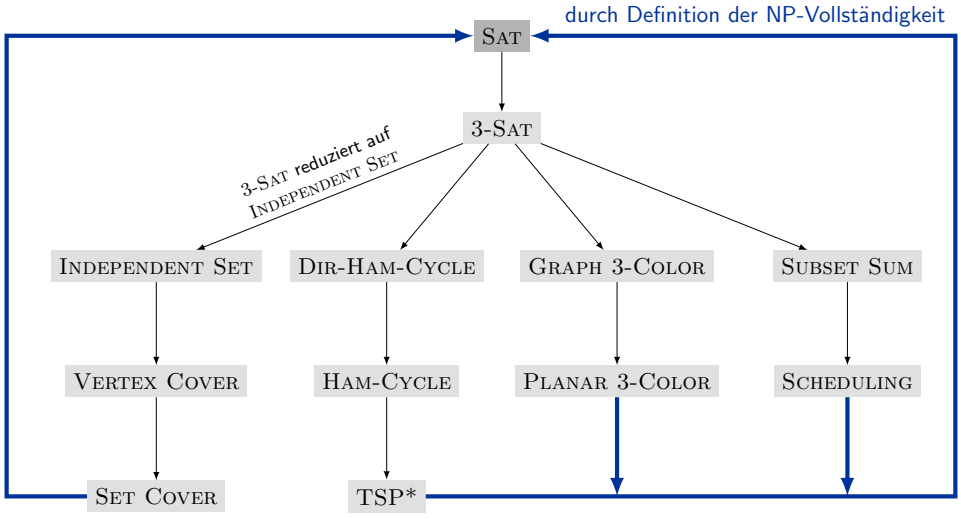
Richard M. Karp

University of California at Berkeley

Abstract: A large class of computational problems involve the determination of properties of graphs, digraphs, integers, arrays of integers, finite families of finite sets, boolean formulas and elements of other countable domains. Through simple encodings from such domains into the set of words over a finite alphabet these problems can be converted into language recognition problems, and we can inquire into their computational complexity. It is reasonable to consider such a problem satisfactorily solved when an algorithm for its solution is found which terminates within a number of steps bounded by a polynomial in the length of the input. We show that a large number of classic unsolved problems of covering, matching, packing, routing, assignment and sequencing are equivalent, in the sense that either each of them possesses a polynomial-bounded algorithm or none of them does.

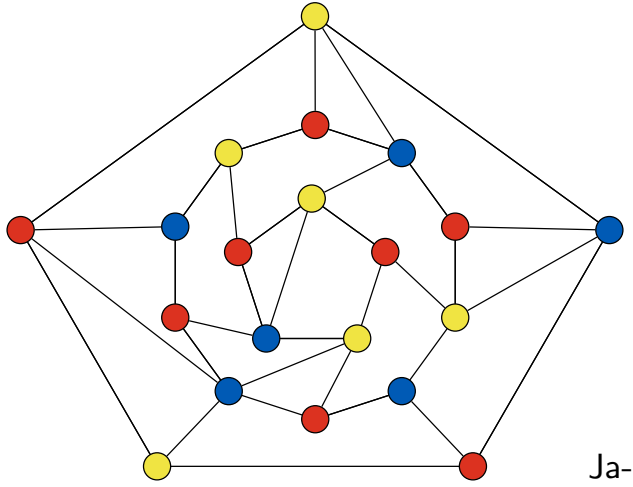
NP-Vollständigkeit

Beobachtung: Alle Probleme in der Abbildung sind NP-vollständig und lassen sich polynomiell auf einander reduzieren.



Beispiel: 3-Color (3-Knotenfärbung)

3-Color: Gegeben sei ein ungerichteter Graph G. Kann man die Knoten des Graphen mit den Farben Rot, Gelb und Blau so einfärben, dass benachbarte Knoten nicht die gleiche Farbe besitzen?



Ja-Instanz

Anwendung: Register-Allocation (Registerzuteilung)

Registerzuteilung: Ist die Zuteilung von Registern zu Programmvariablen. Es dürfen nicht mehr als k Register benutzt werden und zwei Programmvariablen, die zur gleichen Zeit benötigt werden, werden nicht dem gleichen Register zugewiesen.

Register-Interferenz-Graph:

- Knoten sind Variablen in einem Programm.
- Es existiert eine Kante zwischen u und v, wenn es eine Operation gibt, bei der u und v zur gleichen Zeit benötigt werden.

Beobachtung: [Chaitin 1982] Das Problem der Registerzuteilung kann genau dann gelöst werden, wenn der Register-Interferenz-Graph k-färbbar ist.

Fakt: 3-Color $\leq_P$ k-Register-Allocation für eine beliebige Konstante $k \geq 3$.

3-Color

Behauptung: 3-Sat $\leq_P$ 3-Color.

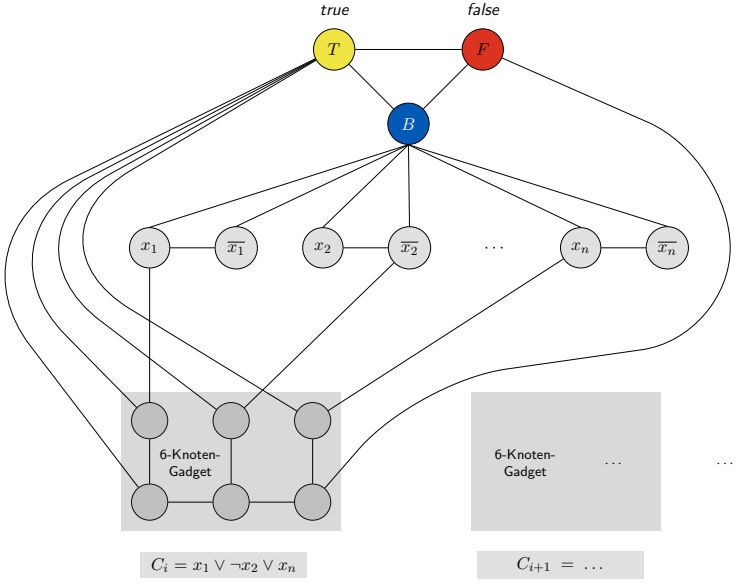
Beweis: Gegeben sei die 3-Sat-Instanz Φ. Wir konstruieren eine Instanz von 3-Color, die genau dann 3-färbbar ist wenn Φ erfüllbar ist.

Konstruktion:

- Für jedes Literal wird ein Knoten erzeugt.
- Erzeuge drei neue Knoten T, F, B. Verbinde diese Knoten zu einem Dreieck und verbinde jedes Literal mit B.
- Verbinde jedes Literal mit seiner Negation.
- Für jede Klausel wird ein Gadget mit 6 Knoten hinzugefügt.
- Jedes Gadget wird mit den Knoten, die den Literalen der entsprechenden Klausel entsprechen, und den Knoten T, F, B verbunden.

  □ *Siehe Abbildungen auf den nächsten Folien.*

3-Color (Visualisierung der Konstruktion)



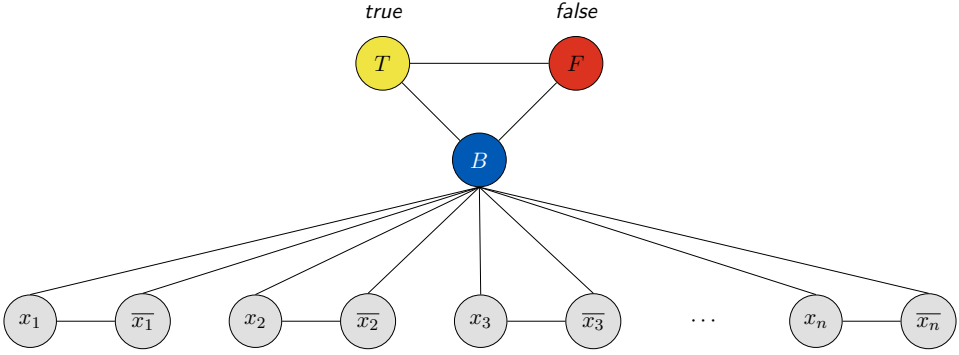
- *Nächste Folien: Beweis fokusiert sich auf Teile der Konstruktion*

3-Color

Behauptung: Graph ist 3-färbbar genau dann, wenn Φ erfüllbar ist.

Beweis: ($\Rightarrow$) Angenommen der Graph ist 3-färbbar.

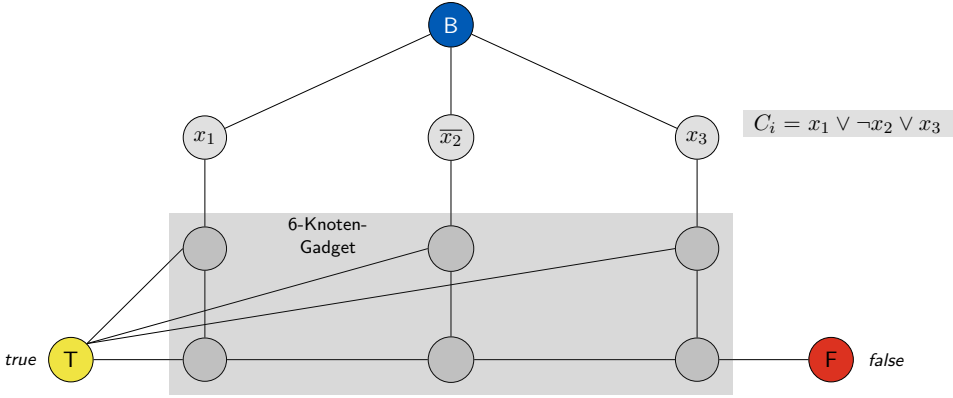
- Betrachte eine 3-Färbung. Wir können ohne Beschränkung der Allgemeinheit annehmen, dass sie den Knoten T gelb, den Knoten F rot, und den Knoten B blau färbt.
- Daher ist jeder Literalknoten rot oder gelb gefärbt.
- Das definiert eine Wahrheitsbelegung f, die die gelben Literale auf *true*, und die roten Literale auf *false* setzt.



3-Color

Behauptung: Graph ist 3-färbbar genau dann, wenn Φ erfüllbar ist.
Beweis: ($\Rightarrow$) Angenommen der Graph ist 3-färbbar.

- Behauptung: Zumindest ein Literal in jeder Klausel wird von f auf *true* gesetzt.

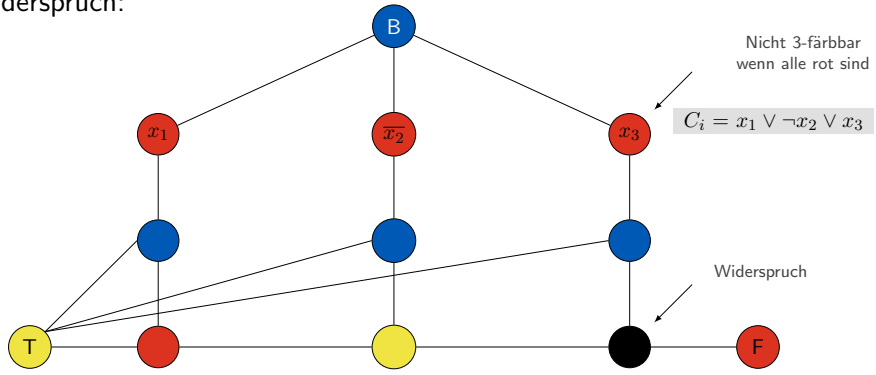


- T, F, B *Knoten verschoben und Kanten nicht eingezeichnet.*

3-Color

Behauptung: Graph ist 3-färbbar genau dann, wenn Φ erfüllbar ist.
Beweis: ($\Rightarrow$) Angenommen der Graph ist 3-färbbar.

- Die Annahme, alle Literale einer Klausel sind auf *false* gesetzt, ergibt einen Widerspruch:



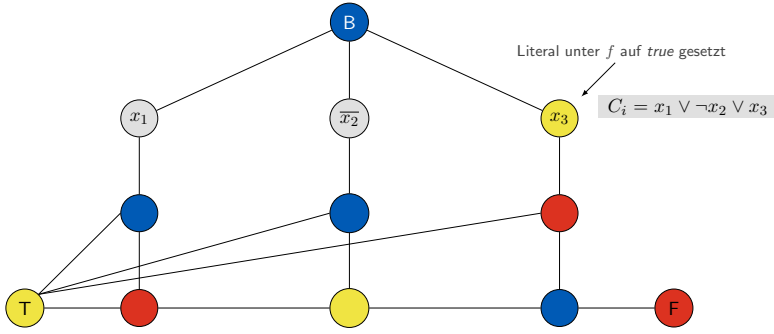
- Daher: Φ ist erfüllbar, und die Richtung $\Rightarrow$ ist gezeigt.

3-Color

Behauptung: Graph ist 3-färbbar genau dann, wenn Φ erfüllbar ist.

Beweis: ($\Leftarrow$) Angenommen die Formel Φ ist erfüllbar.

- Betrachte eine erfüllende Wahrheitsbelegung f.
- Färbe Knoten T gelb, F rot und B blau.
- Färbe Literal gelb, wenn es unter f *true*, ansonsten rot.
- Es ist nicht schwer zu sehen, dass die Färbung auf alle Knoten in den Gadgets erweitert werden kann.



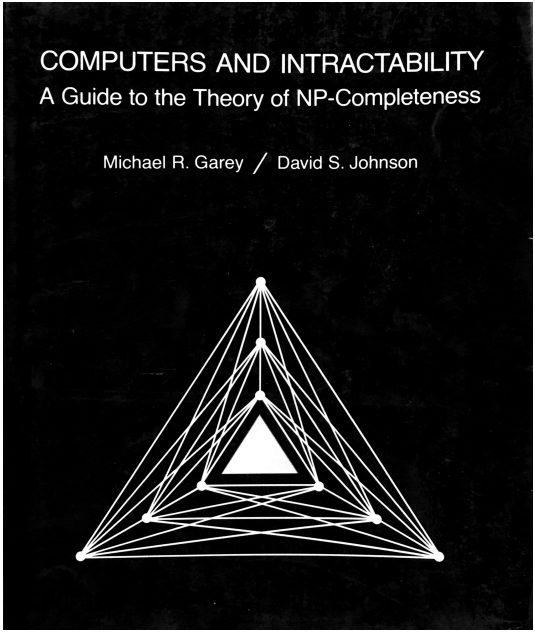
- Daher: der Graph is 3-färbbar, und die Richtung $\Leftarrow$ ist gezeigt.

Auswirkung der NP-Vollständigkeit

Auswirkung der NP-Vollständigkeit:

- Primärer intellektueller Beitrag der Informatik zu anderen Disziplinen.
- 6,000 Zitate pro Jahr (Titel, Abstract, Keywords).
  - Mehr als „Compiler“, „Betriebssysteme“, „Datenbanken“
- Breites Anwendungsspektrum und Klassifizierungsstärke.
- Schon Ende der 1970er Jahre waren hunderte Probleme als NP-vollständig erkannt
- Einflussreiches Buch von Garey und Johnson [1979]

Garey und Johnson 1979



Anwendung auf Optimierungsprobleme

NP-Vollständigkeit ist nur für Ja/Nein-Probleme definiert.

NP-Schwere kann aber in folgender Weise auch auf funktionale Probleme und Optimierungsprobleme angewandt werden.

Beispiel. Betrachte folgendes Problem $\textsc{OptVC}$: Gegeben ist ein Graph G, berechne ein kleinstes Vertex Cover von G.

Theorem: Angenommen P $\neq$ NP. Dann gibt es keinen Polynomialzeitalgorithmus für $\textsc{OptVC}$.

Anwendung auf Optimierungsprobleme

Theorem: Angenommen P $\neq$ NP. Dann gibt es keinen Polynomialzeitalgorithmus für $\textsc{OptVC}$.

Beweis: (durch Widerspruch)

- Angenommen es gäbe einen Polynomialzeitalgorithmus A für $\textsc{OptVC}$.
- Wir verwenden A, um das Ja/Nein-Problem $\textsc{Vertex-Cover}$ in Polynomialzeit zu lösen:
- Auf eine gegebene Instanz (G, k) von $\textsc{Vertex-Cover}$ wenden wir A an, und berechnen in Polynomialzeit ein kleinstes Vertex Cover C von G.
- Falls $|C| \leq k$ antworten wir Ja, ansonsten Nein.
- Da $\textsc{Vertex-Cover}$ NP-vollständig ist, folgt P = NP, ein Widerspruch zur Annahme. $\square$

Terminologie

Es werden oft auch Optmierungsprobleme und funktionale Probleme als „NP-schwer“ bezeichnet, wenn aus ihrere Lösbarkeit in Polynomialzeit $\mathrm{P} = \mathrm{NP}$ folgt.

Wir können also beispielsweise sagen, dass $\textsc{OptVC}$ NP-schwer ist (wir sagen aber nicht, es sei NP-vollständig, da nicht in NP).

NP-Vollständigkeit meistern

Frage: Angenommen, wir müssen ein NP-vollständiges Problem lösen. Wie sollen wir vorgehen?
Antwort: Die Theorie besagt, dass es unwahrscheinlich ist, einen polynomiellen Algorithmus zu finden.

Man muss eine der gewünschten Eigenschaften aufgeben:

- Löse Problem optimal $\rightarrow$ Approximationsalgorithmen, Heuristische Algorithmen.
- Löse Problem in Polynomialzeit $\rightarrow$ Algorithmen mit exponentieller Laufzeit.
- Löse beliebige Instanzen des Problems $\rightarrow$ Identifiziere effizient lösbare Spezialfälle.

NP-Vollständigkeit Spezialfälle

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 15. Mai 2023

Vorlesungsfolien

NP-Vollständigkeit meistern (Wiederholung)

Frage: Angenommen, wir müssen ein NP-vollständiges Problem lösen. Wie sollen wir vorgehen?
Antwort: Die Theorie besagt, dass es unwahrscheinlich ist, einen polynomiellen Algorithmus zu finden.

Man muss eine der gewünschten Eigenschaften opfern:

- Löse Problem optimal
  → Approximationsalgorithmen, Heuristische Algorithmen
- Löse Problem in Polynomialzeit
  → Algorithmen mit exponentieller Laufzeit
- Löse beliebige Instanzen des Problems
  → Identifiziere effizient lösbare Spezialfälle

Überblick

Finden von kleinen Vertex Covers

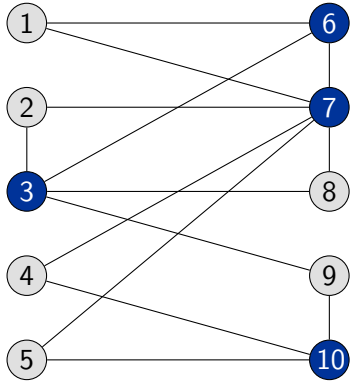
NP-vollständige Probleme auf Bäumen

Knotenfärben in Intervallgraphen

Finden von kleinen Vertex Covers

Vertex Cover (Knotenüberdeckung)

VERTEX COVER: Gegeben sei ein Graph $G = (V, E)$ mit $|V| = n$ Knoten und eine ganze Zahl k. Existiert eine Teilmenge von Knoten $S \subseteq V$, sodass $|S| \leq k$, und für jede Kante $(u, v) \in E$ gilt, dass $u \in S$ oder $v \in S$ (oder beides)?



$k = 4$

$S = \{3, 6, 7, 10\}$

Finden kleiner Vertex Covers

Frage: Was ist möglich, wenn k klein ist?

Brute-Force. $O\left(kn^{k+1}\right)$.

- Probiere alle $\binom{n}{k}$=$O\left(n^k\right)$ Teilmengen der Größe k aus.
- Benötigt $O(kn)$ Zeit um zu überprüfen, ob eine Teilmenge ein Vertex Cover ist.

Ziel: Beschränke die exponentielle Abhängigkeit von k.

Beispiel: $n = 1000$, $k = 10$.
Brute-Force: $kn^{k+1} = 10^{34} \Rightarrow$ undurchführbar.
Besser: $2^k kn \approx 10^7 \Rightarrow$ durchführbar.

Anmerkung: Wenn k eine Konstante ist, dann ist der Algorithmus polynomiell. Wenn k eine kleine Konstante ist, dann ist er auch praktikabel.

Finden kleiner Vertex Covers

Behauptung: Sei (u, v) eine Kante von G. G hat ein Vertex Cover der Größe $\leq k$ genau dann, wenn zumindest einer der Graphen $G - \{u\}$ und $G - \{v\}$ ein Vertex Cover der Größe $\leq k - 1$ hat.

□ *Lösche u und alle inzidenten Kanten.*

Beweis: $\Rightarrow$

- Angenommen G hat ein Vertex Cover S der Größe $\leq k$.
- S enthält entweder u oder v (oder beide). Angenommen, S enthält u.
- Dann ist $S - \{u\}$ ein Vertex Cover von $G - \{u\}$.

Beweis: $\Leftarrow$

- Angenommen S ist ein Vertex Cover von $G - \{u\}$ der Größe $\leq k - 1$.
- Dann ist $S \cup \{u\}$ ein Vertex Cover von G. □

Finden kleiner Vertex Covers

Behauptung: Wenn G ein Vertex Cover der Größe k hat, dann hat $G \leq k(n-1)$ Kanten.

Beweis: Jeder Knoten überdeckt höchstens $n-1$ Kanten. $\square$

Finden kleiner Vertex Covers: Algorithmus

Behauptung: Der folgende Algorithmus bestimmt mit einer Laufzeit in $O\left(2^k kn\right)$, ob G ein Vertex Cover der Größe $\leq k$ hat.

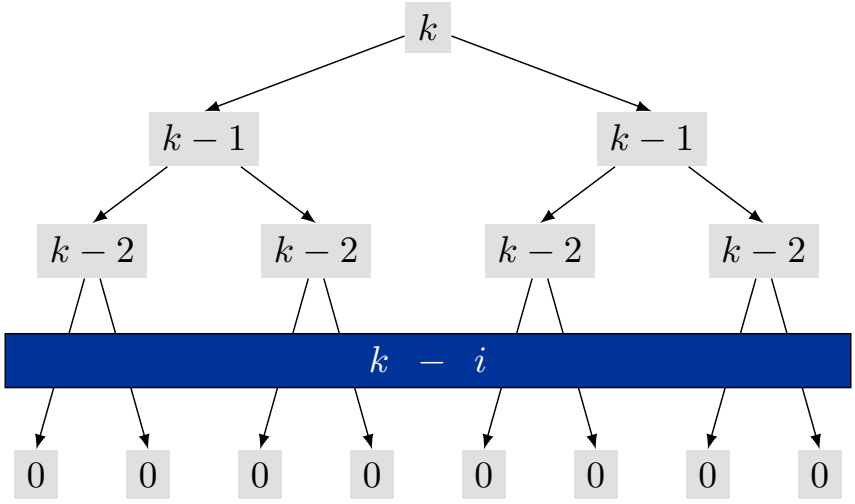
```
Vertex-Cover(G, k):
if G enthält keine Kanten
    return true
if G enthält > k(n − 1) Kanten
    return false
Sei (u, v) eine beliebige Kante von G
a ← Vertex-Cover(G − {u}, k − 1)
b ← Vertex-Cover(G − {v}, k − 1)
return a OR b
```

Beweis:

- Korrektheit folgt aus den zwei vorherigen Behauptungen.
- Es existieren $\leq 2^{k+1}$ Knoten im Rekursionsbaum. Jeder Aufruf benötigt $O(kn)$ Zeit. □

Finden kleiner Vertex Covers: Rekursionsbaum

$$T(n,k) \leq \begin{cases} c & \text{falls } k = 0 \\ cn & \text{falls } k = 1 \\ 2T(n-1,k-1) + ckn & \text{falls } k > 1 \end{cases} \quad \Rightarrow \quad T(n,k) \leq 2^k ckn$$



Lösen NP-vollständiger Probleme auf Bäumen

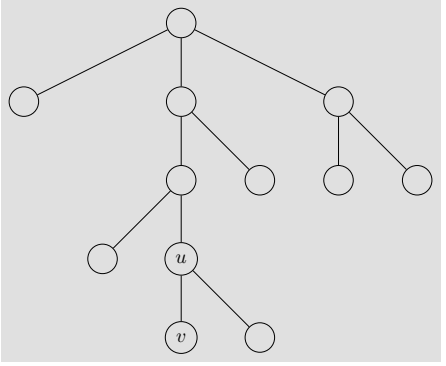
Independent Set auf Bäumen

Independent Set auf Bäumen: Gegeben sei ein Baum. Finde die größte Teilmenge von Knoten, sodass keine zwei Knoten durch eine Kante verbunden werden.

Tatsache: Ein Baum mit zumindest zwei Knoten hat zumindest zwei Blätter.

- *Grad = 1*

Beobachtung: Wenn v ein Blatt ist, dann existiert ein maximales Independent Set, das v enthält.



Beweis: (Austauschargument)

- Wir gehen von einem maximalen Independent Set S aus.
- Wenn $v \in S$, dann ist man fertig.
- Wenn $v \notin S$ und $u \notin S$, dann ist $S \cup \{v\}$ unabhängig $\Rightarrow$ S ist nicht maximal.
- Wenn $v \notin S$ und $u \in S$, dann ist $S \cup \{v\} - \{u\}$ unabhängig.

Independent Set auf Bäumen: Greedy-Algorithmus

Theorem: Der nachfolgende Greedy-Algorithmus findet ein maximales Independent Set in einem Wald $T = (V, E)$
(jede Zusammenhangskomponente des Graphen ist ein Baum).

```
Independent-Set-In-A-Forest(T):
S ← ∅
while T hat zumindest eine Kante
    Sei e = (u, v) eine Kante, sodass v ein Blatt ist
    Füge v zu S hinzu
    Lösche aus V die Knoten u und v (und alle
    zu diesen Knoten inzidenten Kanten)
S ← S ∪ V
return S
```

Beweis: Korrektheit folgt aus der vorherigen Beobachtung. □

Anmerkung: Kann man in $O(n)$ Zeit implementieren, indem man die Knoten in Postorder durchmustert.

Gewichtetes Independent Set auf Bäumen

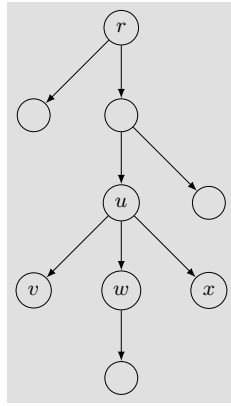
Gewichtetes Independent Set auf Bäumen: Gegeben sei ein Baum und Knotengewichte $w_v > 0$. Finde ein Independent Set S, das $\sum_{v \in S} w_v$ maximiert.

Beobachtung: Wenn (u, v) eine Kante ist, sodass v ein Blatt ist, dann beinhaltet die Lösung entweder u oder sie enthält alle Blattknoten inzident zu u.

Gewichtetes Independent Set auf Bäumen

Lösung mit dynamischer Programmierung: Wähle einen Knoten, z.B. r, als Wurzel aus.

- $OPT_{\text{in}}(u)$ = maximales Gewicht eines Independent Sets des Unterbaums mit Wurzel u, das u enthält.
- $OPT_{\text{out}}(u)$ = maximales Gewicht eines Independent Sets des Unterbaums mit Wurzel u, das u nicht enthält.



Nachfolger$(u) = \{v, w, x\}$

Formulierung:

$$\begin{aligned} OPT_{\text{in}}(u) &= w_u + \textstyle\sum_{v \in \text{Nachfolger}(u)} OPT_{\text{out}}(v) \\ OPT_{\text{out}}(u) &= \textstyle\sum_{v \in \text{Nachfolger}(u)} \max\{OPT_{\text{in}}(v), OPT_{\text{out}}(v)\} \end{aligned}$$

Gewichtetes Independent Set auf Bäumen

Formulierung:

$$
\begin{aligned}
OPT_{\text{in}}(u) &= w_u + \sum_{v \in \text{Nachfolger}(u)} OPT_{\text{out}}(v) \\
OPT_{\text{out}}(u) &= \sum_{v \in \text{Nachfolger}(u)} \max\{OPT_{\text{in}}(v), OPT_{\text{out}}(v)\}
\end{aligned}
$$

Erklärung:

- $OPT_{\text{in}}(u)$ addiert das Gewicht von u (u ist enthalten) und die maximalen Gewichte aller Unterbäume, bei denen die Wurzeln (Nachfolger von u) nicht enthalten sind. Wäre eine Wurzel enthalten, dann wäre es kein Independent Set mehr, da dann diese Wurzel mit u eine Kante gemeinsam hätte.
- $OPT_{\text{out}}(u)$ addiert die maximalen Gewichte aller Unterbäume. Bei einem Unterbaum kann die Wurzel v enthalten sein oder nicht. Zwei Wurzeln zweier Unterbäume von u können niemals miteinander verbunden sein.

Gewichtetes Independent Set auf Bäumen: Implementierung

```
Weighted-Independent-Set-In-A-Tree(T):
Wähle eine Wurzel r aus
foreach Knoten u von T in Postorder
    if u ist ein Blatt
        M_in[u] ← w_u
        M_out[u] ← 0
    else
        M_in[u] ← w_u + Σ_{v∈Nachfolger(u)} M_out[v]
        M_out[u] ← Σ_{v∈Nachfolger(u)} max{M_in[v], M_out[v]}
return max{M_in[r], M_out[r]}
```

- *Stellt sicher, dass ein Knoten nach seinen Nachfolgern besucht wird.*

Gewichtetes Independent Set auf Bäumen: Dynamische Programmierung

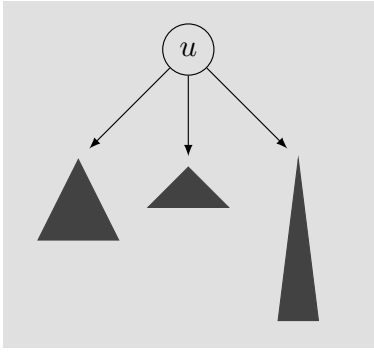
Theorem: Der Algorithmus auf der vorherigen Folie findet ein maximal gewichtetes Independent Set in einem Baum mit einer Laufzeit in $O(n)$.

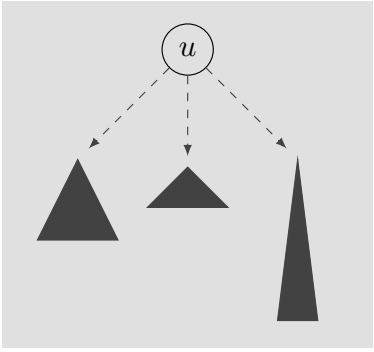
Kann auch das Independent Set (und nicht nur den Wert) finden.

Beweis: Erfordert $O(n)$ Zeit, da wir Knoten in Postorder durchmustern und jede Kante genau einmal überprüfen. □

Kontext

Independent Set auf Bäumen: Dieser strukturierte Spezialfall ist handhabbar, weil wir einen Knoten finden können, der die Verbindung zwischen den Subproblemen in verschiedenen Subbäumen unterbrechen kann.





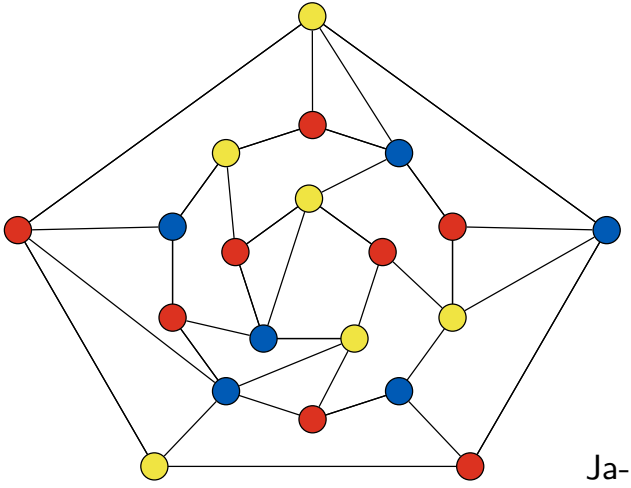
Graphen mit beschränkter Baumweite (elegante Generalisierung von Bäumen):

- Erfassen eine reichhaltige Klasse von Graphen, die in der Praxis auftritt.
- Erlauben die Aufteilung in unabhängige Teile.

Knotenfärben in Intervallgraphen

Knotenfärbeproblem

Erinnerung 3-Color: Gegeben sei ein ungerichteter Graph G. Kann man die Knoten des Graphen mit den Farben Rot, Gelb und Blau so einfärben, dass benachbarte Knoten nicht die gleiche Farbe besitzen?



Ja-Instanz

Knotenfärbeproblem

Erinnerung 3-Color: Gegeben sei ein ungerichteter Graph G. Kann man die Knoten des Graphen mit den Farben Rot, Gelb und Blau so einfärben, dass benachbarte Knoten nicht die gleiche Farbe besitzen?

k-Color: Gegeben sei ein ungerichteter Graph G. Kann man die Knoten des Graphen mit $k \geq 3$ Farben so einfärben, dass benachbarte Knoten nicht die gleiche Farbe besitzen?

k-Color ist NP-vollständig für $k \geq 3$

Als Optimierungsproblem:

Opt-Color: Gegeben sei ein ungerichteter Graph G. Färbe die Knoten des Graphen mit einer minimalen Anzahl Farben so, dass benachbarte Knoten nicht die gleiche Farbe besitzen.

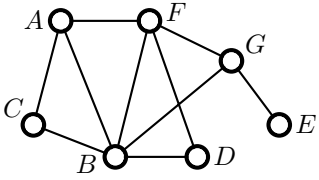
Ziel: Betrachte effizient lösbaren Spezialfall Intervallgraphen

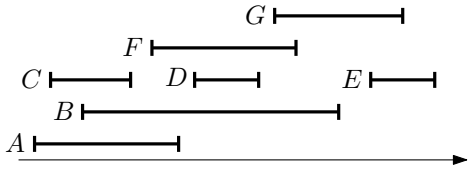
Intervallgraphen

Definition: Intervallgraphen sind Graphen, die sich wie folgt als Schnittgraph von Intervallen in $\mathbb{R}$ repräsentieren lassen:

- ungerichteter Graph $G = (V, E)$
- Intervallmenge $\mathcal{I} = \{I_v \subset \mathbb{R} \mid v \in V\}$
- Kante $(u, v) \in E \Leftrightarrow I_u \cap I_v \neq \emptyset$

Beispiel:



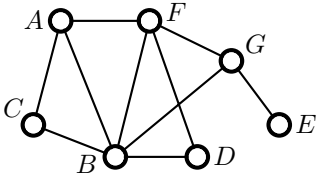


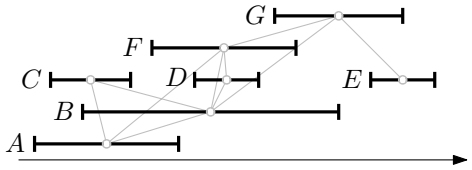
Intervallgraphen

Definition: Intervallgraphen sind Graphen, die sich wie folgt als Schnittgraph von Intervallen in $\mathbb{R}$ repräsentieren lassen:

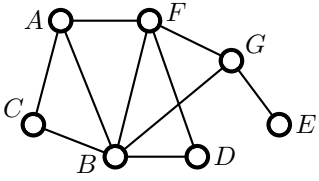
- ungerichteter Graph $G = (V, E)$
- Intervallmenge $\mathcal{I} = \{I_v \subset \mathbb{R} \mid v \in V\}$
- Kante $(u, v) \in E \Leftrightarrow I_u \cap I_v \neq \emptyset$

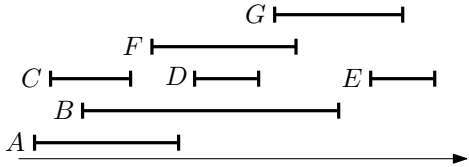
Beispiel:





Färben von Intervallgraphen

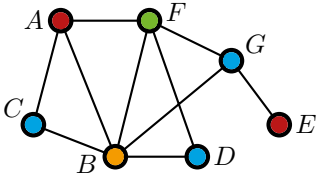


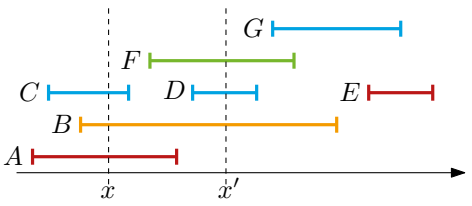


Färben mit einer Farbe:

- äquivalent zu Maximum Independent Set
- äquivalent zu Interval Scheduling (Kap. Greedy-Algorithmen)
- Greedy-Algorithmus EDF (earliest deadline first)?

Färben von Intervallgraphen: Untere Schranke





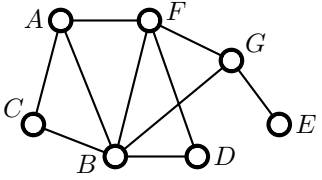
Beobachtungen:

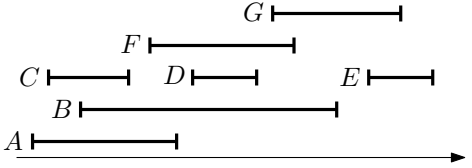
- Liegt ein Punkt $x \in \mathbb{R}$ in k Intervallen, braucht man mindestens k Farben
- Für Tiefe $d := \max_{x \in \mathbb{R}} \{|\{I \in \mathcal{I} \mid x \in I\}|\}$ gilt Färbung von G benötigt $\geq d$ Farben

Frage: Geht es auch immer mit d Farben?

Färben von Intervallgraphen: Algorithmus

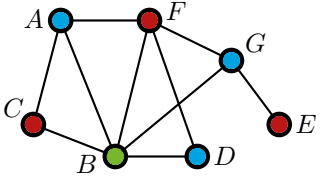
```
Interval-Coloring(I):
Sortiere Intervallgrenzen aufsteigend in Liste L
col_max ← 0
Initialisiere leere Queue Q
foreach Punkt a von L
    if a ist ein Startpunkt
        if Q ist leer
            färbe I = [a, b] mit Farbe col_max
            col_max ← col_max + 1
        else
            entferne c = Q.head und färbe I = [a, b] mit Farbe c
    else //a ist ein Endpunkt
        füge Farbe von I = [b, a] in Q ein
return Färbung von I und Anzahl col_max
```

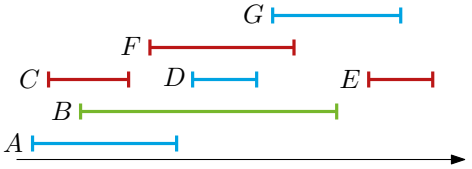




Färben von Intervallgraphen: Algorithmus

```
Interval-Coloring(I):
Sortiere Intervallgrenzen aufsteigend in Liste L
col_max ← 0
Initialisiere leere Queue Q
foreach Punkt a von L
    if a ist ein Startpunkt
        if Q ist leer
            färbe I = [a, b] mit Farbe col_max
            col_max ← col_max + 1
        else
            entferne c = Q.head und färbe I = [a, b] mit Farbe c
    else //a ist ein Endpunkt
        füge Farbe von I = [b, a] in Q ein
return Färbung von I und Anzahl col_max
```





Färben von Intervallgraphen: Analyse

Theorem: Der Algorithmus `Interval-Coloring` berechnet eine minimale Färbung einer Intervallmenge $\mathcal{I}$ und des zugehörigen Intervallgraphen G mit n Knoten in $O(n \log n)$ Zeit.

Beweis:

- Q enthält zu jedem Zeitpunkt nur "freie" Farben
- Wenn keine Farbe frei ist, wird eine neue Farbe gewählt
- Am Ende jedes Intervalls wird dessen Farbe wieder freigegeben
- Färbung gültig, da überlappende Intervalle verschiedene Farben erhalten, also mindestens d
- Es werden genau d Farben $0, \ldots, d-1$ verwendet:
  - sonst wären zum Zeitpunkt der ersten Wahl der Farbe d die Queue Q leer und damit die Tiefe $d+1 \rightarrow$ Widerspruch
- Färbung selbst erfolgt in Zeit $O(n)$, wird allerdings dominiert durch Sortierung der Intervallgrenzen in $O(n \log n)$ Zeit

Ergänzende Literatur

J. Kleinberg and E. Tardos. *Algorithm Design*. Pearson, 2005. Kapitel 10.

Optimierung – Branch-and-Bound

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 22. Mai 2023

Vorlesungsfolien

Kombinatorische Optimierung

Kombinatorische Optimierung: Es geht bei der kombinatorischen Optimierung darum, aus einer (großen) Menge von diskreten Elementen (Gegenstände, Orte usw.) eine Teilmenge zu konstruieren,

- die gewissen Nebenbedingungen entspricht
- und bezüglich einer Kostenfunktion optimal ist (kleinstes Gewicht, kürzeste Strecken, ...).

Kombinatorische Optimierungsaufgaben

Kombinatorische Optimierungsaufgaben: Wir haben bereits mehrere kombinatorische Optimierungsaufgaben in der Vorlesung betrachtet, z.B.

- Minimaler Spannbaum eines Graphen
- Kürzeste Wege in einem Graphen
- Minimales Vertex oder Set Cover
- Maximales Independent Set
- Minimale Knotenfärbung

Algorithmen-Paradigmen: Wir haben Methoden für das Lösen solcher Aufgaben kennengelernt, z.B. Greedy- Konstruktionsverfahren.

Hinweis: Greedy-Konstruktionsverfahren funktionieren aber nicht bei allen Problemen!

Optimierung: Schwere Probleme

Schwere Probleme: In dieser LVA wurden schon einige Probleme besprochen, für die es unwahrscheinlich ist, dass Lösungsverfahren existieren, die alle möglichen Instanzen eines bestimmten Problems in polynomieller Zeit lösen.

Anwendung: In dieser und den nachfolgenden Einheiten werden wir uns mit Verfahren beschäftigen, die bei solchen Problemen grundsätzlich angewendet werden können.

Verfahren für Optimierungsprobleme:

- Branch-and-Bound
- Dynamische Programmierung
- Approximation(salgorithmen)
- Heuristische Verfahren

Optimierung: Roadmap

Branch-and-Bound: Beschränke eine auf Divide-and-Conquer basierende systematische Durchmusterung aller Lösungen mit Hilfe von Methoden, die untere und obere Schranken liefern, und ermittle eine optimale Lösung.

Dynamische Programmierung

Approximation(salgorithmen)

Heuristische Verfahren

Überblick

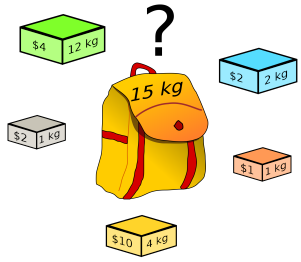
Rucksackproblem

Branch-and-Bound für das Rucksackproblem

Branch-and-Bound für Minimales Vertex Cover

Rucksackproblem

Gegeben: n Gegenstände mit positiven rationalen Gewichten $g_1, \ldots, g_n$ und Werten $w_1, \ldots, w_n$ und eine Kapazität G (auch positiv rational).



Gesucht: Teilmenge S der Gegenstände mit Gesamtgewicht $\leq G$ und maximalem Gesamtwert.

Rucksackproblem: Mathematische Formulierung

Entscheidungsvariablen: Wir führen 0/1-Entscheidungsvariablen für die Wahl der Gegenstände ein:

$$x_1, \ldots, x_n, \text{ wobei } x_i = \begin{cases} 0 & \text{falls Gegenstand } i \text{ nicht gewählt wird} \\ 1 & \text{falls Gegenstand } i \text{ gewählt wird} \end{cases}$$

Mathematische Formulierung: Für n Gegenstände:

Maximiere

$$\sum_{i=1}^{n} w_i x_i,$$

wobei

$$\sum_{i=1}^{n} g_i x_i \leq G, \quad x_i \in \{0, 1\}$$

Rucksackproblem: Enumeration (Backtracking)

Enumeration: Eine Enumeration aller zulässigen Lösungen für das Rucksackproblem entspricht der Aufzählung aller Teilmengen der n-elementigen Menge (bis auf diejenigen Teilmengen, die nicht in den Rucksack passen).

Lösungsvektor und Zielfunktion: Zu jedem aktuellen Lösungsvektor $\vec{x} = (x_1, \ldots, x_n)$ gehört ein Zielfunktionswert (Gesamtwert von $\vec{x}$) w_{curr} und ein Gesamtgewicht g_{curr}. Die bisher beste gefundene Lösung wird in dem globalen Vektor $\vec{x}_{\text{best}}$ und der zugehörige Lösungswert in der globalen Variablen $w_{\max}$ gespeichert.

Prinzip: Wir folgen wiederum dem Prinzip des Divide-and-Conquer.

Rucksackproblem: Enumerationsalgorithmus

Eingabe: Anzahl z der fixierten Variablen in $\vec{x}$; Gesamtwert w_{curr}; Gesamtgewicht g_{curr}; aktueller Lösungsvektor $\vec{x}$.

```
Enum(z, w_curr, g_curr, x⃗):
if g_curr ≤ G
    if w_curr > w_max
        w_max ← w_curr
        x⃗_best ← x⃗
    for i ← z + 1 bis n
        x_i ← 1
        Enum(i, w_curr + w_i, g_curr + g_i, x⃗)
        x_i ← 0
```

Hinweis:

- $w_{\max}$ und $\vec{x}_{\text{best}}$ sind globale Variablen.
- Initialisierung: $w_{\max} = 0$ und $\vec{x}_{\text{best}} = \vec{0}$

Rucksackproblem: Enumerationsalgorithmus

Ablauf: Der Algorithmus wird mit dem Aufruf Enum$(0, 0, 0, \vec{0})$ gestartet.

- In jedem rekursiven Aufruf wird die aktuelle Lösung $\vec{x}$ bewertet.
- Danach werden die Variablen x_1 bis x_z als fixiert betrachtet.
- Der dadurch beschriebene Teil des gesamten Suchraums wird weiter unterteilt: Wir betrachten alle möglichen Fälle, welche Variable x_i (mit $i = z + 1$ bis $i = n$) als nächstes auf 1 gesetzt werden kann.
- Die Variablen x_{z+1} bis x_{i-1} werden gleichzeitig auf 0 fixiert.
- Alle so erzeugten kleineren Unterprobleme werden durch rekursive Aufrufe gelöst.

Komplexität: Es gibt bis zu 2^n rekursive Aufrufe und der Aufwand pro Aufruf (exklusive Rekursion) ist konstant. Daher liegt die Laufzeit in $O(2^n)$.

Branch-and-Bound

Rucksackproblem: Verbesserung der Enumeration

Idee zur Verbesserung: Überprüfen von Zwischenlösungen mit $z < n$ fixierten Variablenwerten.

- Man überprüft, ob es noch möglich sein kann, aus dieser Lösung durch Hinzufügen weiterer Gegenstände eine zu erzeugen, die besser ist als die bisher beste gefundene.
- Wenn es offensichtlich ist, dass keine neue beste Lösung abgeleitet werden kann, dann sind weitere rekursive Aufrufe nicht sinnvoll.
- Das frühzeitige Abbrechen führt zu einer Beschneidung des rekursiven Aufrufbaums.
- Das kann eine erhebliche Beschleunigung bewirken.

Branch-and-Bound: Rucksackproblem

Ansatz:

- Berechne obere Schranke U', und führe den Aufruf nur durch, wenn der Wert $U' > w_{\max}$.
- Sortiere die Gegenstände nach nicht-steigenden Werten $\frac{w_i}{g_i}$.

```
Enum(z, w_curr, g_curr, x⃗):
if g_curr ≤ G
    if w_curr > w_max
        w_max ← w_curr
        x⃗_best ← x⃗
    for i ← z + 1 bis n
        U' ← w_curr + (G − g_curr) · w_i / g_i
        if U' > w_max
            x_i ← 1
            Enum(i, w_curr + w_i, g_curr + g_i, x⃗)
            x_i ← 0
```

Branch-and-Bound: Rucksackproblem

Obere Schranke: $U' \leftarrow w_{\text{curr}} + (G - g_{\text{curr}}) \cdot \frac{w_i}{g_i}$

Erklärung:

- w_{curr} enthält den Wert der bisherigen Zuteilung.
- $G - g_{\text{curr}}$ ist die verbleibende Kapazität im Rucksack.
- Die verbleibende Kapazität wird mit dem aktuell untersuchten Gegenstand i komplett (vielleicht auch mehrmals) aufgefüllt. Hierbei kann es auch zu teilweisen Zuteilungen kommen (z.B. Gegenstand i wird 1,7 mal eingepackt).
- Da die Gegenstände nach nicht-steigenden Werten $\frac{w_i}{g_i}$ sortiert sind, haben alle Gegenstände $i+1, i+2, ..., n$ einen relativen Wert kleiner als der von i.
- Damit wird die obere Schranke U' garantiert größer gleich dem Wert der optimalen Lösung sein (für dieses Teilproblem).

Prinzip von Branch-and-Bound: Maximierungsproblem

Branching: Wie bei der Enumeration üblich wird das Problem rekursiv in kleinere Teilprobleme partitioniert $\rightarrow$ *Divide-and-Conquer*-Prinzip.

Bounding: Für jedes Teilproblem wird

- eine lokale obere Schranke U' (*upper bound*) und
- eine lokale untere Schranke L' (*lower bound*)

berechnet.

Abbruch: Teilprobleme mit $U' \leq L$ (L entspricht einer globalen unteren Schranke) brauchen nicht weiter verfolgt werden!

Schranken:

- Der Wert jeder gültigen Lösung ist eine untere Schranke.
- Obere Schranken werden i. A. separat mit einer sogenannten *Dualheuristik* ermittelt.

Rucksackproblem: Verbesserte untere Schranke

Sortierung: Die Gegenstände werden nicht-steigend nach ihren Werten $\frac{w_i}{g_i}$ sortiert.

Untere Schranke: Man durchläuft alle Gegenstände, deren Variablen noch nicht festgelegt sind, in der sortierten Reihenfolge und packt den jeweils aktuellen Gegenstand ein, falls noch Platz im Rucksack ist. (Greedy-Algorithmus)

Rucksackproblem: Verbesserte obere Schranke

Einfache obere Schranke wurde weiter vorne beschrieben.

Mögliche Verbesserung:

- Alle Gegenstände, deren Variablen noch nicht festgelegt sind, werden in der sortierten Reihenfolge durchlaufen.
- Man packt alle Gegenstände ein, bis man zu dem ersten Gegenstand kommt, der nicht mehr in den Rucksack passt.
- Sei r die noch freie Kapazität des Rucksacks. Dann zählt man $r \cdot w_i/g_i$ noch zu dem Wert der Gegenstände im Rucksack dazu.
- Der letzte Gegenstand wird daher nur teilweise eingepackt.
- Alle verbleibenden Gegenstände werden ignoriert.

Lösung:

- Diese Vorgehensweise liefert in der Regel eine Schranke, der keine gültige Lösung des Rucksackproblems entspricht.
- Falls diese Vorgehensweise zu einer gültigen Lösung führt, dann ist die Lösung (für das betrachtete Teilproblem) optimal.

Maximierungsproblem: Vorgehen

Allgemeines Vorgehen:

- Das Problem wird z.B. durch das Fixieren von Variablen oder Hinzufügen von Randbedingungen in Unterprobleme zerteilt, d.h. der Lösungsraum wird partitioniert.
- Ist für eine (oder mehrere) dieser Teilmengen die für sie berechnete obere Schranke U' nicht größer als die beste überhaupt bisher gefundene untere Schranke L (= Wert der bisher besten Lösung), braucht man die Lösungen in dieser Teilmenge nicht mehr beachten.
- Ist die obere Schranke größer als die beste gegenwärtige untere Schranke, muss man die Teilmengen weiter zerkleinern.
- Man fährt solange mit der Zerteilung fort, bis für alle Lösungsteilmengen die obere Schranke nicht mehr größer ist als die (global) beste untere Schranke.

Maximierungsproblem: Allgemeiner Algorithmus

Eingabe: Instanz I

```
Branch-and-Bound-Max(I):
L ← −∞ oder Wert einer initialen heuristischen Lösung
Π ← {I}
while ∃I′ ∈ Π
    Entferne I′ aus Π
    Berechne für I′ lokale obere Schranke U′ mit Dualheuristik
    if U′ > L
        Berechne für I′ gültige heuristische Lösung → untere Schranke L′
        if L′ > L
            L ← L′
        if U′ > L
            Partitioniere I′ in Teilinstanzen I_1, ..., I_k
            Π = Π ∪ {I_1, ..., I_k}
return beste gefundene Lösung mit Wert L
```

- *Bounding – Fall $U' \leq L$ nicht weiter interessant.*
- *Branching.*

Maximierungsproblem: Allgemeines Verfahren

Allgemeines Verfahren:

- Branch-and-Bound ist ein allgemeines Prinzip (Metaverfahren).
- Es kann auf verschiedenste diskrete Optimierungsprobleme angewendet werden.
- Entscheidend für die Effizienz ist
  - vor allem die Wahl der Heuristiken für U' und L',
  - wie das Branching erfolgt und
  - welche Teilinstanz ausgewählt wird.

Rucksackproblem: Beispiel

Gegeben: 4 Gegenstände, Rucksackkapazität $= 100$

Gegenstand	1	2	3	4
Gewicht g_i	32	16	21	50
Wert w_i	80	20	63	100
Verhältnis w_i/g_i	2.5	1.25	3	2

Sortierung:

- Für jeden Gegenstand i das Verhältnis w_i/g_i berechnen.
- Sortierte Reihenfolge der Gegenstände: 3 (3), 1 (2.5), 4 (2), 2 (1.25)

Rucksackproblem: Beispiel

Branch-and-Bound-Baum:

4 Gegenstände, Rucksackkapazität $= 100$

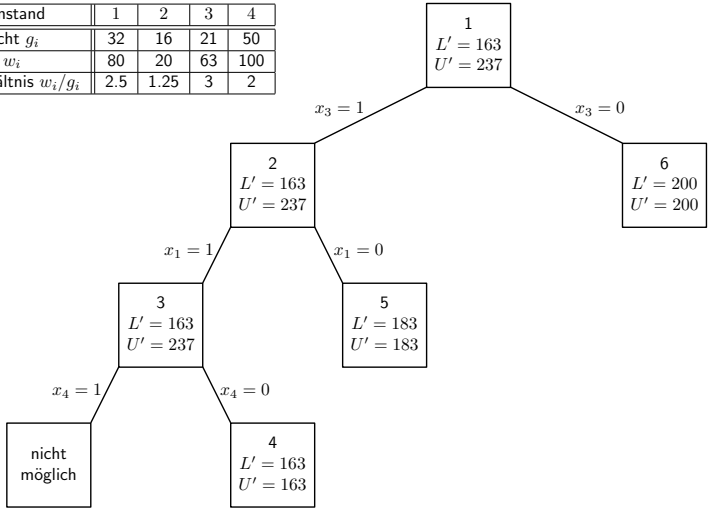
Gegenstand	1	2	3	4
Gewicht g_i	32	16	21	50
Wert w_i	80	20	63	100
Verhältnis w_i/g_i	2.5	1.25	3	2

Rucksackproblem: Beispiel

Branch-and-Bound-Baum:

4 Gegenstände, Rucksackkapazität $= 100$

Gegenstand	1	2	3	4
Gewicht g_i	32	16	21	50
Wert w_i	80	20	63	100
Verhältnis w_i/g_i	2.5	1.25	3	2



Rucksackproblem: Beispiel

Erklärung:

- In 1 (Start) sind noch keine Variablen fixiert.
- In 2 geht man davon aus, dass Gegenstand 3 (x_3) fixiert ist und nur die anderen Gegenstände ausgewählt werden können.
- Eine Fixierung von Gegenstand 3, 1 und 4 führt zu einer unmöglichen Lösung (würde eine Kapazität von 103 benötigen)
- Bei 4 und 5 ist $L = U$ und daher brauchen in diesem Unterbaum keine weiteren Gegenstände hinzugefügt werden (Beschneiden des rekursiven Aufrufbaums).
- Bei 6 ist auch $L = U$ (der Unterbaum braucht nicht mehr untersucht werden) und L ist hier am größten. Daher werden Gegenstand 1, 2 und 4 eingepackt ($x_3 = 0$).

Branch-and-Bound: Auswahl des nächsten Teilproblems

Auswahl des nächsten Teilproblems:

- Welches Teilproblem aus der Liste der offenen Probleme jeweils als nächstes ausgewählt und bearbeitet wird, ist für die grundsätzliche Funktionsweise und Korrektheit von Branch-and-Bound egal.
- Die Auswahl hat jedoch mitunter starke Auswirkungen auf die praktische Laufzeit.

Beispiele für Strategien:

- Best-first
- Depth-first

Branch-and-Bound: Auswahl der Probleme

Best-first:

- Es wird jeweils ein Teilproblem mit der besten dualen Schranke (also der größten oberen Schranke) ausgewählt.
- Dadurch wird immer die kleinstmögliche Anzahl an Teilproblemen abgearbeitet.

Depth-first:

- Es wird jeweils ein zuletzt erzeugtes Teilproblem weiter bearbeitet (vergleiche Tiefensuche bei der Durchmusterung von Graphen).
- Man erhält meist am raschesten eine vollständige und gültige Näherungslösung.
- Häufig wird auch mit einer Depth-first Strategie begonnen und nach Erhalt einer gültigen Lösung mit Best-first fortgesetzt um die Vorteile zu kombinieren.

Branch-and-Bound für Minimales Vertex Cover

Minimierungsproblem: Allgemeiner Algorithmus

Eingabe: Instanz I

```
Branch-and-Bound-Min(I):
U ← ∞ oder Wert einer initialen heuristischen Lösung

Π ← {I}
while ∃I' ∈ Π
    Entferne I' aus Π
    Berechne für I' lokale untere Schranke L' mit Dualheuristik
    if L' < U
        Berechne für I' gültige heuristische Lösung → obere Schranke U'
        if U' entspricht einer gültigen Lösung für I'
            if U' < U
                U ← U'
        if L' < U
            Partitioniere I' in Teilinstanzen I_1, ..., I_k
            Π ← Π ∪ {I_1, ..., I_k}
return beste gefundene Lösung mit Wert U
```

Bounding - Fall $L' \geq U$ nicht weiter interessant. *Branching.*

Branch-and-Bound: Minimales Vertex Cover

Eingabe: Graph $G = (V, E)$ und Knotenmenge $C = \emptyset$

```
MinVertexCover-BranchAndBound(G, C):
U ← |V| − 1
Π ← {(G, C)}
while ∃I′ ∈ Π
    Entferne I′ aus Π
    Berechne für I′ = (G′, C′) lokale untere Schranke L′ mit Matchingheuristik
    if L′ < U
        Berechne für I′ gültige heuristische Lösung mit Greedyheuristik
            → obere Schranke U′
        if U′ < U
            U ← U′
        if L′ < U
            u_max ← Knoten mit maximalem Grad in G′
            Erzeuge Teilinstanzen I_1 = (G′ − {u_max}, C ∪ {u_max}) und
                I_2 = (G′ − {u_max} − N(u_max), C ∪ N(u_max))
            Π ← Π ∪ {I_1, I_2}
return beste gefundene Lösung mit Wert U
```

- *alle Nachbarknoten von* $u_{\max}$

Branch-and-Bound: Minimales Vertex Cover

Untere Schranke: Wird mit Hilfe eines *Matchings* bestimmt.

- Sei ein Graph $G = (V, E)$ gegeben.
- Eine Menge $M \subseteq E$ heißt Matching, wenn keine zwei Kanten aus M einen Knoten gemeinsam haben.

Nicht erweiterbares Matching:

- Nicht erweiterbares Matching (*maximales Matching*) bedeutet, dass es keine Kante $e \in E \setminus M$ gibt, sodass $\{e\} \cup M$ ein gültiges Matching ist.
- Das ist nicht notwendigerweise ein größtes Matching.
- Nicht erweiterbares Matching kann mit einem Greedy-Verfahren gefunden werden.

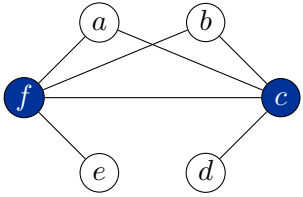
Branch-and-Bound: Minimales Vertex Cover

Berechnung der unteren Schranke L' für die Instanz (G', C'):

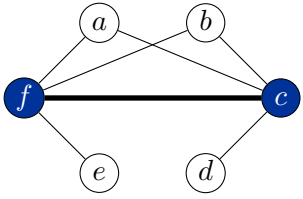
- Man wählt für L' die Größe eines nicht erweiterbaren Matchings.
  - Dabei wird zunächst eine beliebige Kante $e = (u, v)$ gewählt und dann die Knoten u und v und ihre inzidenten Kanten aus G' entfernt.
  - Man fährt mit dieser Prozedur fort, bis keine Kante mehr vorhanden ist.
  - Die Anzahl der gewählten Kanten entspricht der Größe des Matchings.
- Kanten in einem Matching haben keine Knoten gemeinsam.
- Ein Vertex Cover muss zumindest einen Knoten für jede Kante in einem Matching wählen.
- Daher ist die Größe eines Matchings von G' plus die Größe von C' eine untere Schranke für die Größe eines Vertex Covers des Eingabegraphen G.

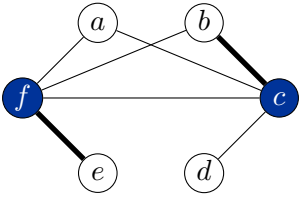
Branch-and-Bound: Beispiel für untere Schranke

Vertex Cover: Vertex Cover mit $k = 2$



Greedy-Matching: 2 Beispiele für Greedy-Matching (fett eingezeichnet Kanten).



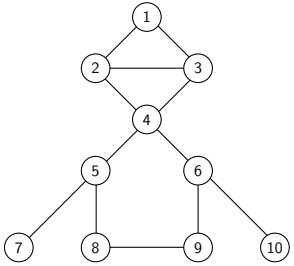


Branch-and-Bound: Minimales Vertex Cover

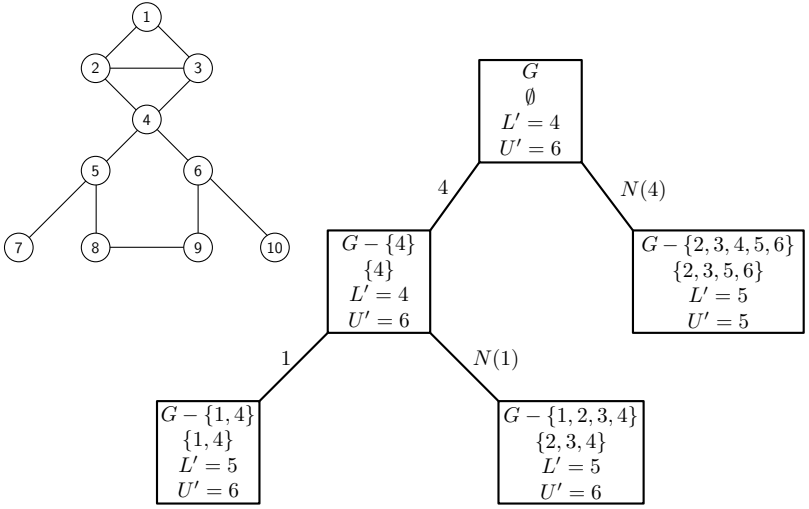
Obere Schranke U': Wird mit Hilfe eines Greedy-Algorithmus bestimmt.

- Sei ein Graph $G' = (V', E')$ und eine Knotenmenge C' gegeben.
- Initialisiere eine Menge $S \leftarrow \emptyset$.
- Sortiere die Knoten nicht-steigend nach dem Knotengrad.
- Durchlaufe V' in dieser Reihenfolge solange der Graph noch Kanten enthält.
  - Füge den Knoten u mit höchstem Knotengrad zu S hinzu.
  - Entferne u und alle seine inzidenten Kanten aus G'.
  - Passe die Reihenfolge der verbleibenden Knoten an.
- Die Menge S ist ein Vertex Cover für G'.
- Daher ist $|C'| + |S|$ eine obere Schranke für die Größe eines minimalen Vertex Covers des Eingabegraphen G.

Minimales Vertex Cover: Beispiel



Minimales Vertex Cover: Beispiel



Branch-and-Bound: Zusammenfassung

- Branch-and-Bound ist eine allgemein für kombinatorische Optimierungsprobleme einsetzbare Technik.
- Sie funktioniert sowohl für Maximierungs- als auch für Minimierungsprobleme.
- Praktisch lassen sich oft hohe Beschleunigungen erreichen, die worst-case Laufzeit bleibt jedoch wie bei der Enumeration.

Vorgehen beim Entwurf von Branch-and-Bound Algorithmen:

- Wie lassen sich (Teil-)Instanzen des Problems ausdrücken?
- Was sind gute Heuristiken für untere und obere Schranken?
- Wie wird eine (Teil-)Instanz in weitere Teilinstanzen partitioniert (Branching)?
- Welche Teilinstanz wird im nächsten Schritt ausgewählt?

Optimierung – Dynamische Programmierung

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 24. Mai 2023

Vorlesungsfolien

Optimierung: Roadmap

Branch-and-Bound

Dynamische Programmierung: Dynamische Programmierung kann dann eingesetzt werden, wenn das Problem aus vielen gleichartigen Teilproblemen besteht und eine optimale Lösung sich aus optimalen Lösungen der Teilprobleme zusammensetzt.

Approximation(salgorithmen)

Heuristische Verfahren

Grundlagen

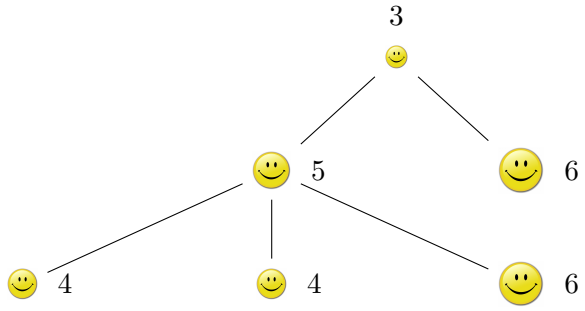
Dynamische Programmierung: Teile das Problem in eine Folge von überlappenden Teilproblemen auf und erstelle und speichere Lösungen für immer größere Teilprobleme unter Verwendung der abgespeicherten Lösungen.

Optimalitätsprinzip von Bellman: Dynamische Programmierung führt zu einem optimalen Ergebnis genau dann, wenn es sich aus den optimalen Ergebnissen der Subprobleme zusammensetzt.

Effizienz: Hängt von der Vorgehensweise bei der Aufteilung und Ermittlung der Lösungen für die einzelnen Teilprobleme ab.

Wesentlicher Aspekt: Speicherung (*memoization*) von Ergebnissen für Subprobleme zur Wiederverwendung.

Beispiel: Weighted Independent Set auf Bäumen



Geschichte der dynamischen Programmierung

Bellman: [1950er] Leistete Pionierarbeit bei der systematischen Untersuchung der dynamischen Programmierung.

Etymologie:

- Dynamische Programmierung = Zeitablauf planen.
- Verteidigungsminister war ablehnend gegenüber mathematischer Forschung.
- Bellman suchte einen eindrucksvollen Namen, um eine Konfrontation zu vermeiden.

„it's impossible to use dynamic in a pejorative sense“
„something not even a Congressman could object to“

Referenz: Bellman, R. E. Eye of the Hurricane, An Autobiography.

Anwendung der dynamischen Programmierung

Bereiche:

- Bioinformatik
- Informationstheorie
- Operations Research
- Informatik: Theorie, Grafik, Künstliche Intelligenz, Compilerbau ...

Einige bekannte Algorithmen:

- Bellman-Ford-Algorithmus für das Finden kürzester Pfade in Graphen
- Effiziente Methode für das Rucksack-Problem
- Needleman-Wunsch und Smith-Waterman Algorithmen für Genomsequenz-Alignment

Überblick

Einführendes Beispiel: Fibonacci

Gewichtetes Interval Scheduling

Segmented Least Squares

Rucksackproblem

Kürzeste Pfade

Einführendes Beispiel

Fibonacci-Zahlen

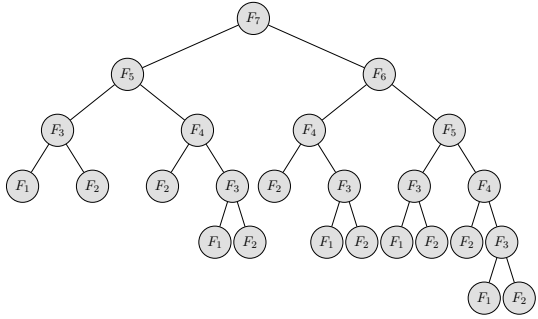
Folge von Fibonacci-Zahlen: $F_1 = F_2 = 1 \quad F_n = F_{n1} + F_{n2} \quad \forall n \geq 3$

Einfacher rekursiver Algorithmus:

```
Fibonacci(n):
if n = 1 oder n = 2
  return 1
else
  return Fibonacci(n − 1) + Fibonacci(n − 2)
```

Fibonacci-Zahlen: Laufzeit

- Gesamtanzahl der Aufrufe für die i-te Fibonacci-Zahl entspricht der i-ten Fibonacci-Zahl (Beispiele: $F_{10} = 55$, $F_{20} = 6765$, $F_{30} = 832040$, $F_{40} = 102334155$).
- Exponentielle Zeitkomplexität, da $F_n = F_{n-1} + F_{n-2} \geq 2F_{n-2}$ und folglich $F_n \geq 2^{\frac{n}{2}-1}$ (mit $F_2 = 2^0$) und daher $F_n \geq (\sqrt{2})^{n-2}$



Dynamische Programmierung (Rekursiv)

Speicherung: Die berechneten Fibonacci-Zahlen zwischenspeichern (z.B. in einem Array F) und in der Berechnung wiederverwenden.

```
for i ← 1 bis n
  F[i] ← leer

Fibonacci(n):
if F[n] ist leer
  if n = 1 oder n = 2
    F[n] ← 1
  else
    F[n] ← Fibonacci(n − 1) + Fibonacci(n − 2)
return F[n]
```

Laufzeit: $O(n)$ (maximal zwei rekursive Aufrufe pro Arrayeintrag)

Dynamische Programmierung (Iterativ)

Speicherung: Die berechneten Fibonacci-Zahlen zwischenspeichern (z.B. in einem Array F) und in der Berechnung wiederverwenden.

```
Linear-Fibonacci(n):
F[1] ← 1
F[2] ← 1
for i ← 3 bis n
  F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

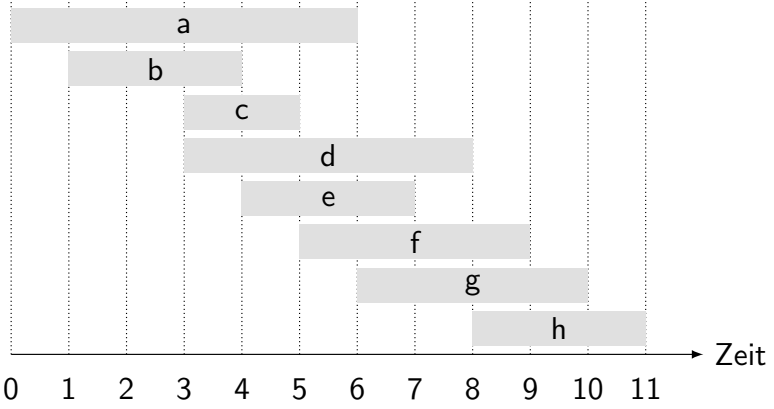
Laufzeit: $O(n)$ (konstanter Aufwand für jeden Schleifendurchlauf)

Gewichtetes Interval Scheduling

Gewichtetes Interval Scheduling

Gewichtetes Interval Scheduling:

- Job j startet zum Zeitpunkt s_j, endet zum Zeitpunkt f_j und hat ein Gewicht $w_j > 0$.
- Zwei Jobs sind kompatibel, wenn sie sich nicht überlappen.
- Ziel: Finde eine Teilmenge maximalen Gewichts von paarweise kompatiblen Jobs.

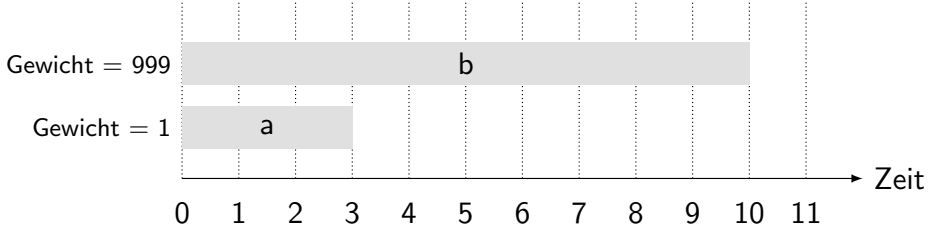


Interval Scheduling: Rückblick

Wiederholung: Greedy-Algorithmus funktioniert, wenn alle Gewichte gleich 1 sind.

- Berücksichtige Jobs in aufsteigender Reihenfolge der Beendigungszeit.
- Füge Job zur Teilmenge hinzu, wenn er kompatibel mit dem zuvor ausgewählten Job ist.

Beobachtung: Greedy-Algorithmus scheitert, wenn beliebige Gewichte erlaubt sind.

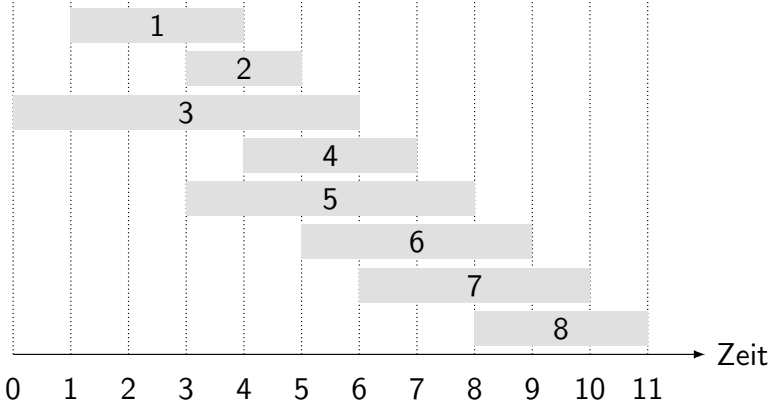


Gewichtetes Interval Scheduling

Notation: Ordne Jobs aufsteigend sortiert nach Beendigungszeit: $f_1 \leq f_2 \leq \cdots \leq f_n$.

Definition: $p(j) =$ größter Index $i < j$, sodass Job i kompatibel zu Job j ist.

Beispiel: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.



Dynamische Programmierung: Binäre Auswahl

Notation: $OPT(j) =$ Wert der optimalen Lösung für das Problem, bestehend aus den Jobs $1, 2, \ldots, j$.

Wir unterscheiden zwei Fälle:

- Fall 1: $OPT(j)$ wird erreicht mit einer Lösung, die den Job j enthält.
- Fall 2: $OPT(j)$ wird erreicht mit einer Lösung, die den Job j nicht enthält.

Konsequenz:

- Fall 1: Die Lösung kann nicht die inkompatiblen Jobs $\{p(j)+1, p(j)+2, \ldots, j-1\}$ enthalten. Daher gilt dann $OPT(j) = w_j + OPT(p(j))$.
- Fall 2: Es gilt $OPT(j) = OPT(j-1)$, da wir wissen, dass die Lösung den Job j nicht enthält. Also gilt:

$$OPT(j) = \begin{cases} 0 & \text{wenn } j = 0 \\ \max\{w_j + OPT(p(j)), OPT(j-1)\} & \text{sonst} \end{cases}$$

Gewichtetes Interval Scheduling: Brute-Force-Ansatz

Brute-Force-Algorithmus:

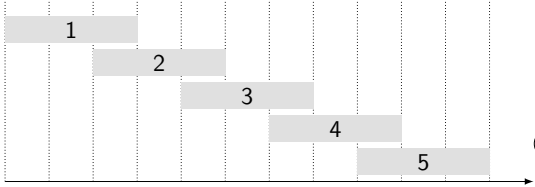
- Eingabe: $n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n$
- Sortiere Jobs nach Beendigungszeit, sodass $f_1 \leq f_2 \leq \cdots \leq f_n$.
- Berechne $p(1), p(2), \ldots, p(n)$

```
Compute-Opt(j):
if j = 0
  return 0
else
  return max(w_j+Compute-Opt(p(j)), Compute-Opt(j − 1))
```

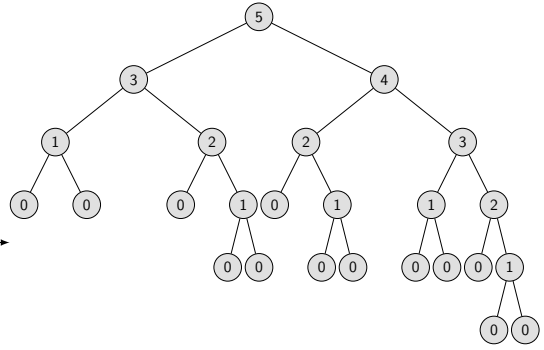
Gewichtetes Interval Scheduling: Brute-Force-Ansatz

Beobachtung: Rekursiver Algorithmus ist ineffizient wegen redundanter Subprobleme
$\Rightarrow$ exponentieller Algorithmus.

Beispiel: Anzahl der rekursiven Aufrufe für eine Gruppe von schichtweise angeordneten Instanzen wächst wie eine Fibonacci-Folge.



$p(1) = 0,\ p(j) = j - 2$



Gewichtetes Interval Scheduling: Speicherung

Speicherung: Speichere Ergebnisse jedes Teilproblems in einem Cache. Berechnung nur, wenn noch nicht gespeichert.

Allgemein:

- Eingabe: n, $s_1, \ldots, s_n$, $f_1, \ldots, f_n$, $w_1, \ldots, w_n$
- Sortiere Jobs nach Beendigungszeit, sodass $f_1 \leq f_2 \leq \cdots \leq f_n$.
- Berechne $p(1), p(2), \ldots, p(n)$

```
for j ← 1 bis n
    M[j] ← leer
M[0] ← 0

M-Compute-Opt(j):
if  M[j] ist leer
    M[j] ← max(w_j + M-Compute-Opt(p(j)), M-Compute-Opt(j − 1))
return M[j]
```

- *globales Array*

Gewichtetes Interval Scheduling: Laufzeit

Behauptung: Version mit Speicherung benötigt $O(n \log n)$ Zeit.

- Sortiere nach Beendigungszeit: $O(n \log n)$.
- Berechne $p(\cdot)$: $O(n \log n)$ mittels binärer Suche (für jedes Interval) auf der nach Beendigungszeit sortierten Folge.
- `M-Compute-Opt(j)`: Jeder Aufruf benötigt $O(1)$ Zeit (ohne die Rekursion) und
  - (i) liefert entweder einen existierenden Wert `M[j]`
  - (ii) oder berechnet einen neuen Eintrag `M[j]` und macht zwei rekursive Aufrufe.
- Maß für den Fortschritt $\varphi =$ die Anzahl der nicht leeren Einträge in `M[]`.
  - Am Anfang gilt $\varphi = 0$, danach $\varphi \leq n$.
  - (ii) Erhöht φ um 1.
- Die gesamte Laufzeit von `M-Compute-Opt(n)` ist $O(n)$.

Gewichtetes Interval Scheduling: Bottom-up

Bottom-up dynamische Programmierung: Iterative Lösung.

Allgemein:

- Eingabe: n, $s_1, \ldots, s_n$, $f_1, \ldots, f_n$, $w_1, \ldots, w_n$.
- Sortiere Jobs nach Beendigungszeit, sodass $f_1 \leq f_2 \leq \cdots \leq f_n$.
- Berechne $p(1), p(2), \ldots, p(n)$

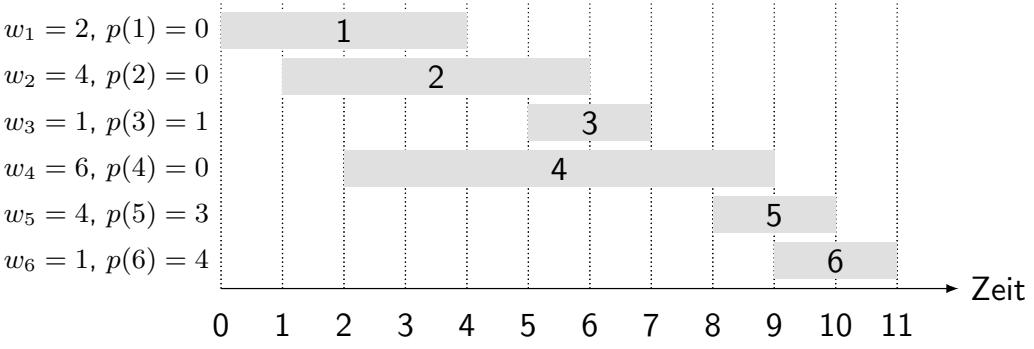
```
Iterative-Compute-Opt():
M[0] ← 0
for j ← 1 bis n
    M[j] ← max(w_j + M[p(j)], M[j − 1])
```

Laufzeit: Die Laufzeit von `Iterative-Compute-Opt` liegt in $O(n)$ (Schleife von 1 bis n).

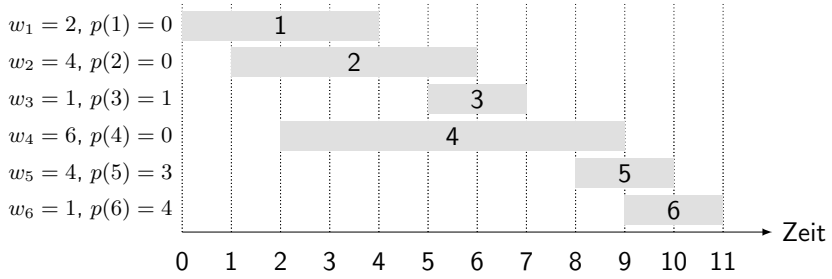
Gewichtetes Interval Scheduling: Beispiel

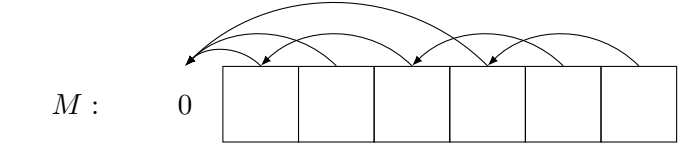
Gegeben:

- $n = 6$ Jobs mit Gewichten $w_i, i = 1 \ldots n$.
- Jobs sind schon sortiert nach Beendigungszeit.



Gewichtetes Interval Scheduling: Beispiel

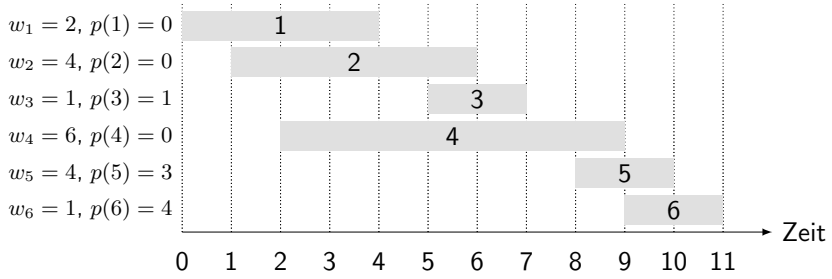


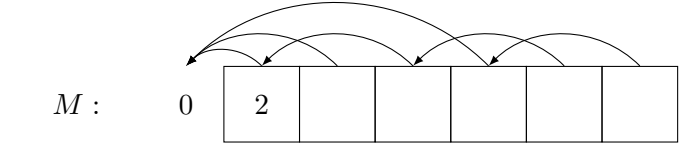


$w_j + M[p(j)]$:

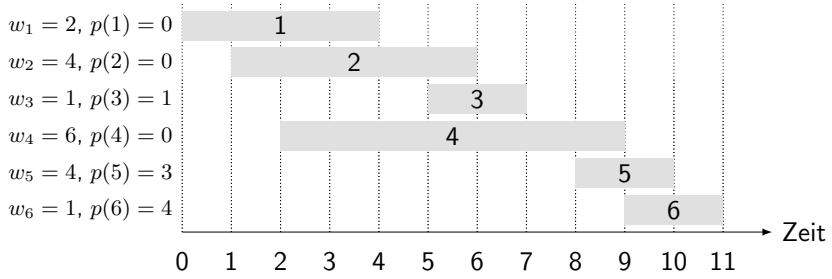
$M[j-1]$:

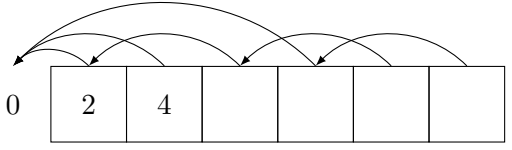
Gewichtetes Interval Scheduling: Beispiel





Gewichtetes Interval Scheduling: Beispiel

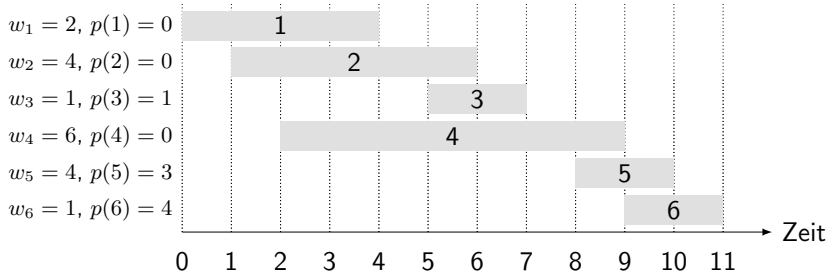


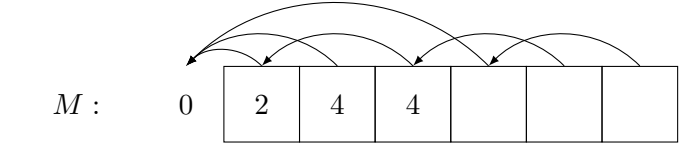


$w_j + M[p(j)]$: 2 4

$M[j-1]$: 0 2

Gewichtetes Interval Scheduling: Beispiel

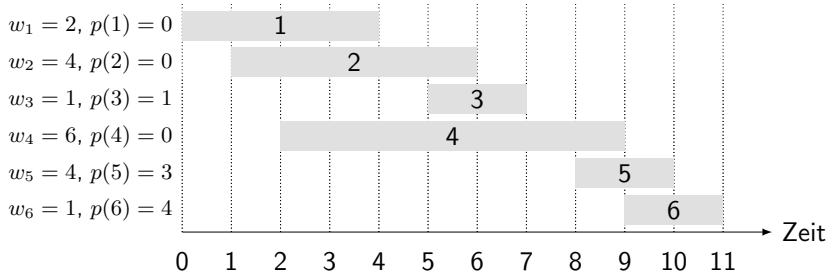


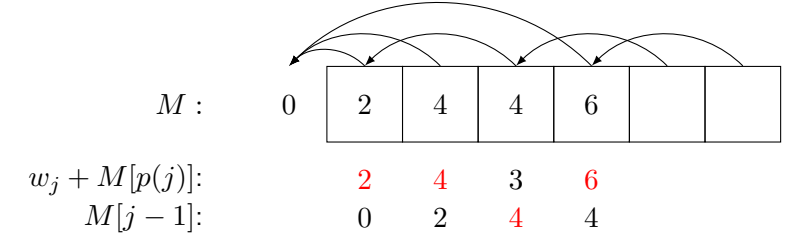


$w_j + M[p(j)]$: 2 4 3

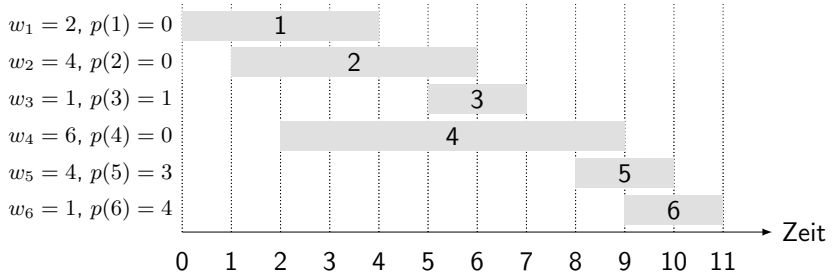
$M[j-1]$: 0 2 4

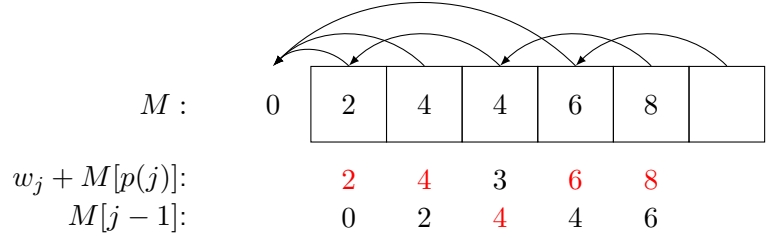
Gewichtetes Interval Scheduling: Beispiel



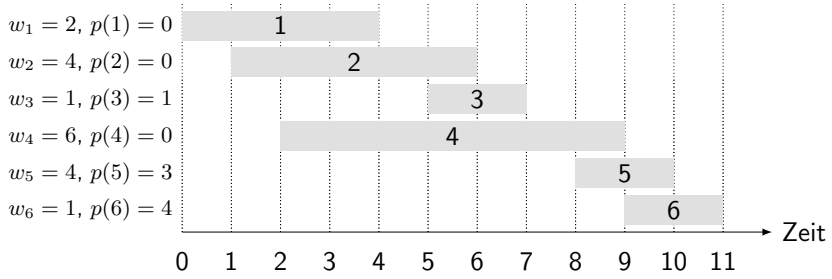


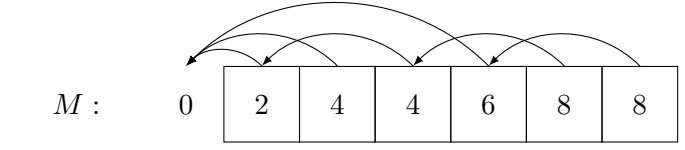
Gewichtetes Interval Scheduling: Beispiel





Gewichtetes Interval Scheduling: Beispiel





Gewichtetes Interval Scheduling: Finden einer Lösung

Frage: Algorithmus berechnet den optimalen Wert. Wie bekommen wir aber die Lösung?
Antwort: Durch eine Nachbearbeitung (Backtracking).

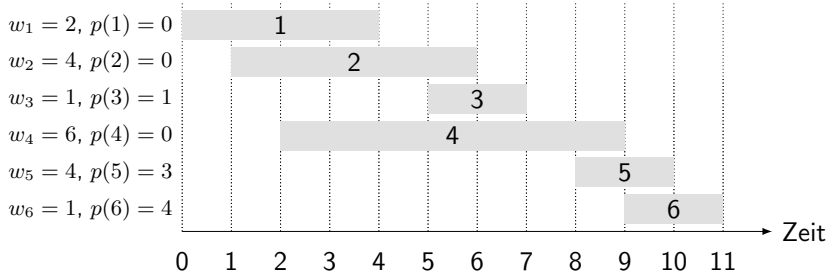
Ablauf:

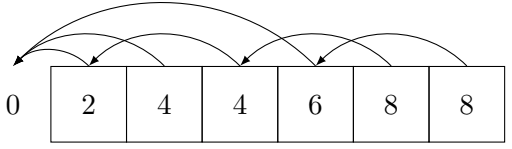
- `M-Compute-Opt(n)` oder `Iterative-Compute-Opt(n)` ausführen
- `Find-Solution(n)` ausführen

```
Find-Solution(j):
if j = 0
    Keine Ausgabe
elseif w_j + M[p(j)] > M[j − 1]
    Gib j aus
    Find-Solution(p(j))
else
    Find-Solution(j − 1)
```

- Anzahl der rekursiven Aufrufe $\leq n \Rightarrow$ Laufzeit $O(n)$.

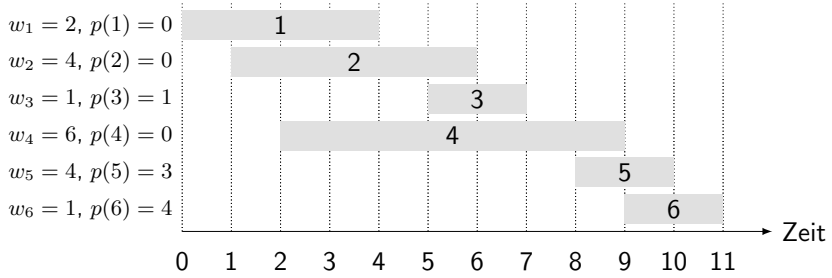
Gewichtetes Interval Scheduling: Beispiel

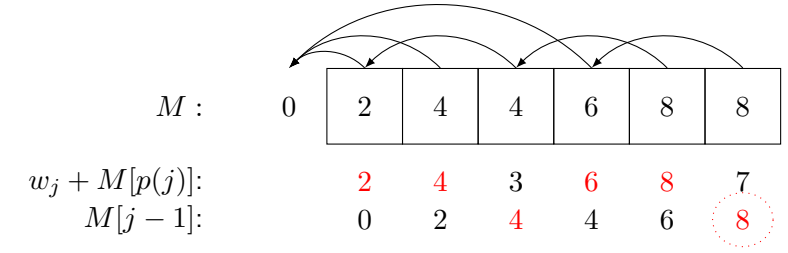




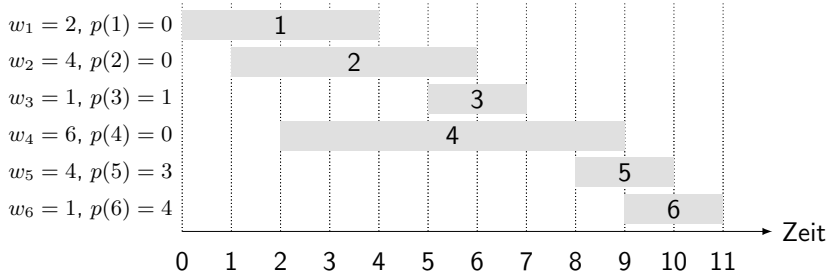
$w_j + M[p(j)]$:	2	4	3	6	8	7
$M[j-1]$:	0	2	4	4	6	8

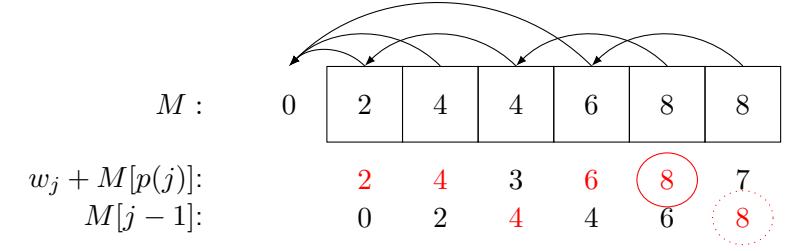
Gewichtetes Interval Scheduling: Beispiel



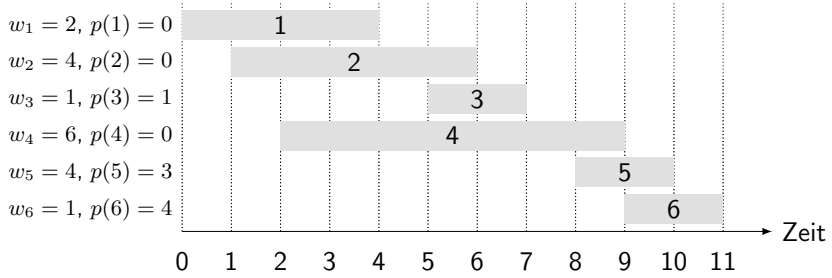


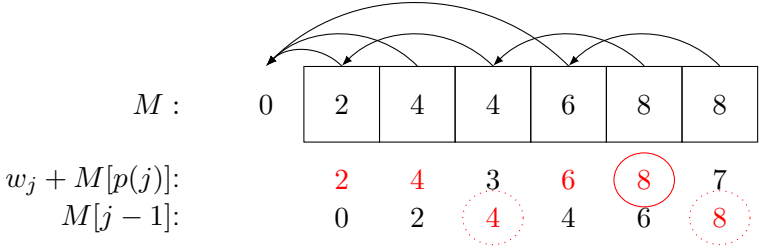
Gewichtetes Interval Scheduling: Beispiel



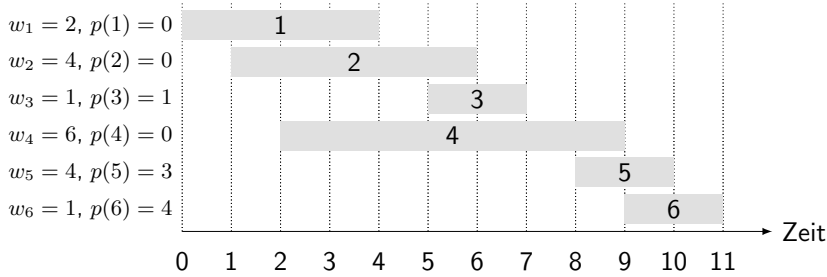


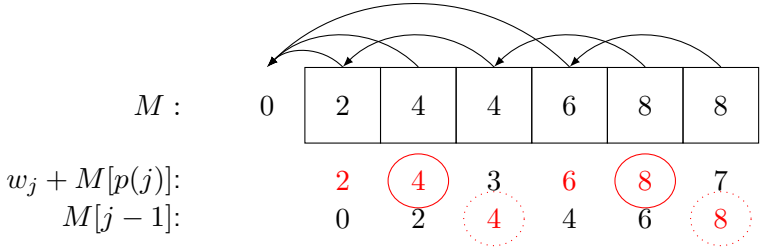
Gewichtetes Interval Scheduling: Beispiel



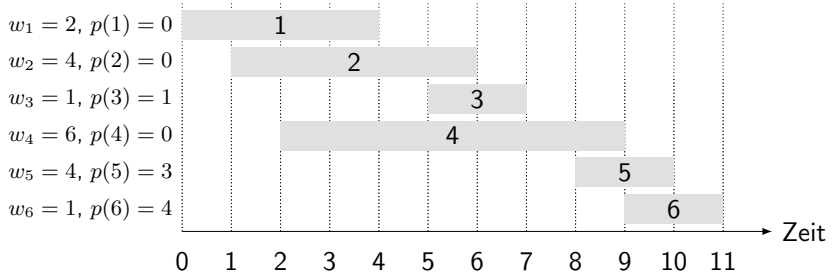


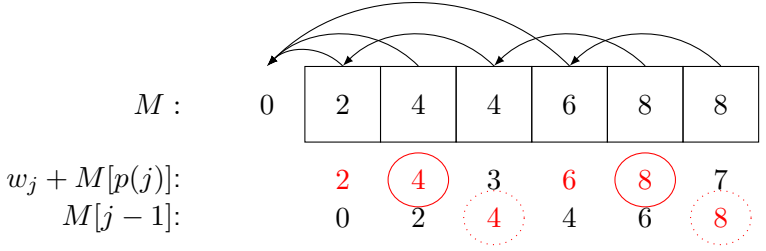
Gewichtetes Interval Scheduling: Beispiel





Gewichtetes Interval Scheduling: Beispiel





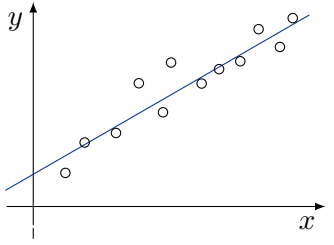
Ergebnis: 2 und 5.

Segmented Least Squares

Least Squares

- Fundamentales Problem in der Statistik und der Numerischen Analyse.
- Gegeben: n Punkte in der Ebene: $(x_1, y_1), \ldots, (x_n, y_n)$.
- Finde eine Gerade $y = ax + b$, welche die Summe des quadrierten Fehlers minimiert:

$$\text{Err} = \sum_{i=1}^{n} (y_i - (ax_i + b))^2$$



Analytische Lösung: der minimale Fehler ist erreicht, wenn

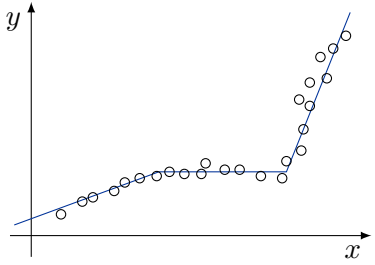
$$a = \frac{n \sum_i x_i y_i - \left(\sum_i x_i\right)\left(\sum_i y_i\right)}{n \sum_i x_i^2 - \left(\sum_i x_i\right)^2}, \; b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented Least Squares

- Punkte durch eine Folge von Geradensegmenten annähern.
- Gegeben: n Punkte in der Ebene $(x_1, y_1), \ldots, (x_n, y_n)$ so dass
- $x_1 < x_2 < \cdots < x_n$, finde eine Folge von Geraden welche eine bestimmte Funktion $f(x)$ minimiert.

Frage: Was ist eine angemessene Wahl für $f(x)$? Die Funktion $f(x)$ sollte sowohl Genauigkeit als auch Sparsamkeit gewährleisten.

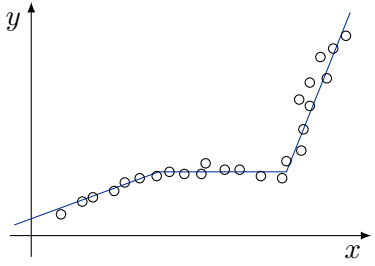
☐ *Höhe des Fehlers* ☐ *Anzahl der Geraden*



Segmented Least Squares

- Punkte durch eine Folge von Geradensegmenten annähern.
- Gegeben: n Punkte in der Ebene $(x_1, y_1), \ldots, (x_n, y_n)$ so dass
- $x_1 < x_2 < \cdots < x_n$, finde eine Folge von Geraden welche:
  - die Summe der quadrierten Fehler E in jedem Segment
  - die Anzahl der Geraden L

  minimiert.
- Tradeoff Funktion: $E + cL$, für eine Konstante $c > 0$.



Dynamischer Ansatz: Segmented Least Squares

Notation

- $OPT(j) =$ minimale Kosten für die Punkte $p_1, p_{i+1}, \ldots, p_j$
- $e(i, j) =$ minimale Summe des quadrierten Fehlers für $p_i, p_{i+1}, \ldots, p_j$

Berechnen von $OPT(j)$:

- letztes Segment nutzt die Punkte $p_i, p_{i+1}, \ldots, p_j$ für ein bestimmtes i
- Kosten $= OPT(i-1) + e(i, j) + c$

$$OPT(j) = \begin{cases} 0 & \text{falls } j = 0 \\ \min\limits_{1 \leq i \leq j} \{OPT(i-1) + e(i, j) + c\} & \text{sonst} \end{cases}$$

Segmented Least Squares: Algorithmus

```
Segmented-Least-Squares( P = {p_1, p_2, ..., p_n} )
    M[0] = 0
    for j ← 1 bis n
        for i ← 1 bis j
            berechne Fehler e(i,j) für Punkte p_i, ..., p_j

    for j ← 1 to n
        M[j] = min_{1≤i≤j} (M[i-1] + e(i,j) + c)

    return M[n]
```

Laufzeit. $O(n^3)$.

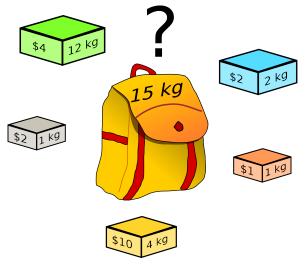
- Flaschenhals ist das Berechnen von $e(i,j)$ für $O(n^2)$ Paare, $O(n)$ pro Paar mit der Formel für Least Squares.
- Finden einer Lösung analog zu Interval Scheduling durch Rückverfolgen der Minimierung

  □ *Kann mithilfe geschickterer Vorberechnung und Wiederverwendung von Zwischenergebnissen zu $O(n^2)$ verbessert werden.*

Rucksackproblem

Rucksackproblem

Gegeben: n Gegenstände mit positiv rationalen Gewichten $g_1, \ldots, g_n$, Werten $w_1, \ldots, w_n$ und einer positiv rationalen Kapazität G.



Gesucht: Teilmenge S der Gegenstände mit Gesamtgewicht $\leq G$ und maximalem Gesamtwert.

Vereinfachung: Der Einfachheit halber nehmen wir im folgenden an, dass sowohl die Gewichte als auch die Kapazität positiv ganzzahlig sind.

Rucksackproblem: Beispiel

Beispiel: $\{3,4\}$ ergibt einen Gesamtwert von 40.

#	Wert	Gewicht	w_i/g_i
1	1	1	1
2	6	2	3
3	18	5	$3\frac{3}{5}$
4	22	6	$3\frac{2}{3}$
5	28	7	4

$G = 11$

Greedy: Füge wiederholt einen Gegenstand mit einem maximalen Verhältnis von w_i/g_i, der in den Rucksack passt, hinzu.
Beispiel: $\{5,2,1\}$ ergibt nur einen Gesamtwert von 35 $\Rightarrow$ Greedy ist nicht optimal.

Idee und Motivation

- Im Kapitel über Branch und Bound haben wir einen Algorithmus mit Laufzeit $O(2^n)$ vorgestellt.
- Hier stellen wir einen Algorithmus mit Laufzeit $O(nG)$ vor, der falls $nG < 2^n$ wesentlich effizienter ist.
- Die Intuition hinter dem Algorithmus ist, dass man nur (Teil-)lösungen mit verschiedenen Gewichten unterscheiden muss.

Dynamische Programmierung: Falscher Ansatz

Definition: $OPT(i) =$ Maximaler Profit für die Teilmenge von den Gegenständen $1, \ldots, i$.

- Fall 1: $OPT(i)$ wird erreicht mit einer Lösung, die den Gegenstand i nicht enthält.
  - Es gilt $OPT(i) = OPT(i-1)$, da wir wissen, dass die Lösung den Gegenstand i nicht enthält.
- Fall 2: $OPT(i)$ wird erreicht mit einer Lösung, die den Gegenstand i enthält.
  - Das Akzeptieren von Gegenstand i impliziert nicht, dass wir andere Gegenstände nicht aufnehmen werden.
  - Ohne zu wissen, welche anderen Gegenstände vor i ausgewählt wurden, können wir nicht sagen, ob wir für i und die nachfolgenden Gegenstände genug Platz haben.

Lösungsansatz: Die Berechnung der Teilprobleme muss die verbleibende Gesamtkapazität berücksichtigen!

Dynamische Programmierung: Gewichtsbeschränkung

Definition: $OPT(i, g)$ = Maximaler Profit für die Teilmenge von den Gegenständen $1, \ldots, i$, mit einer Gewichtsbeschränkung g.

- Fall 1: $OPT(i, g)$ wird erreicht mit einer Lösung, die den Gegenstand i nicht enthält.
  - Es gilt $OPT(i, g) = OPT(i - 1, g)$, da wir wissen, dass die Lösung den Gegenstand i nicht enthält.
- Fall 2: $OPT(i, g)$ wird erreicht mit einer Lösung, die den Gegenstand i enthält.
  - Neue Gewichtsbeschränkung $= g - g_i$.
  - Daher gilt dann $OPT(i, g) = w_i + OPT(i - 1, g - g_i)$.

$$OPT(i, g) = \begin{cases} 0 & \text{wenn } i = 0 \\ OPT(i-1, g) & \text{wenn } g_i > g \\ \max\{OPT(i-1, g), \quad w_i + OPT(i-1, g-g_i)\} & \text{sonst} \end{cases}$$

Rucksackproblem: Bottom-Up

Rucksack: Befülle ein $(n+1) \times (G+1)$ Array.

Eingabe: n, G, $g_1, \ldots, g_n$, $w_1, \ldots, w_n$

```
for g ← 0 bis G
    M[0,g] ← 0

for i ← 1 bis n
    for g ← 0 bis G
        if g_i > g
            M[i,g] ← M[i−1,g]
        else
            M[i,g] ← max{M[i−1,g], w_i + M[i−1, g−g_i]}
return M[n,G]
```

Laufzeit und Platzbedarf: $O(nG)$

Rucksack Algorithmus

$G + 1$ →

$n + 1$ ↓

	0	1	2	3	4	5	6	7	8	9	10	11
$\emptyset$	0	0	0	0	0	0	0	0	0	0	0	0
$\{ 1 \}$	0	1	1	1	1	1	1	1	1	1	1	1
$\{ 1, 2 \}$	0	1	6	7	7	7	7	7	7	7	7	7
$\{ 1, 2, 3 \}$	0	1	6	7	7	18	19	24	25	25	25	25
$\{ 1, 2, 3, 4 \}$	0	1	6	7	7	18	22	24	28	29	29	40
$\{ 1, 2, 3, 4, 5 \}$	0	1	6	7	7	18	22	28	29	34	35	40

OPT: $\{ 4, 3 \}$
Wert $= 22 + 18 = 40$

$G = 11$

Gegenstand	Wert	Gewicht
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Rucksackproblem: Bestimmen der Lösung

Optimale Lösung: Mit Hilfe der Werte im Array M.

```
Find-Solution(M):
i ← n
k ← G
A ← ∅
while i > 0 und k > 0
    if M[i,k] ≠ M[i-1,k]
        A ← A ∪ {i}
        k ← k − g_i
    i ← i − 1
return A
```

Rucksackproblem: Laufzeit

Laufzeit: $O(nG)$.

- Polynomiell in n.
- Aber die Laufzeit hängt auch von der Rucksackkapazität ab.
  - G ist exponentiell in der Eingabelänge, weil Zahlen binär kodiert werden.
  - Die Laufzeit ist somit nicht polynomiell in der Eingabelänge beschränkt.
- „Pseudo-polynomiell.“

Allgemein: Falls $P \neq NP$ kann das Rucksackproblem nicht in Polynomialzeit gelöst werden.

Rucksackproblem: Verbesserung

Speicherung: Man muss eigentlich nicht das gesamte Array speichern.

Verbesserung:

- Man merkt sich nur die letzte Zeile.
- Die Aktualisierung der Einträge erfolgt von rechts nach links.

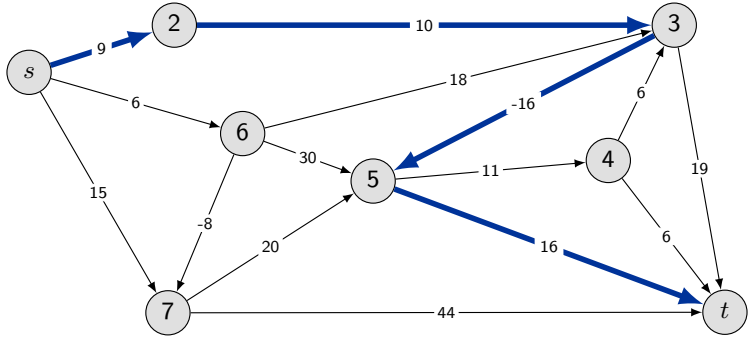
Kürzeste Pfade

Kürzeste Kantenzüge

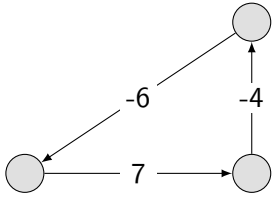
Kürzester Kantenzug: Gegeben sei ein gerichteter Graph $G = (V, E)$, mit Kantengewichten c_{vw}. Finde einen kürzesten Kantenzug zwischen Knoten s und t.

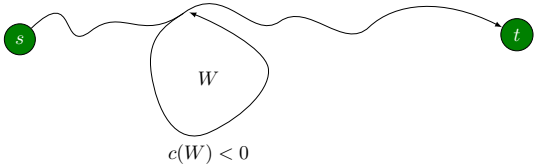
- *erlaube negative Gewichte (bei Dijkstra-Algorithmus nicht erlaubt)*

Beispiel:



Kürzeste Kantenzüge: Kreise mit negativen Kosten (negative Kreise)





Bemerkung: Im Falle von negativen Kreisen:

- keine sinnvolle Definition von kürzesten Kantenzug mehr möglich,
- Definition von kürzesten Pfaden bleibt jedoch sinnvoll.

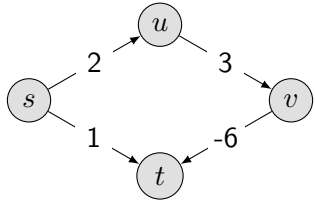
Kürzeste Pfade: Komplexität

Theorem: Das Finden eines kürzesten Pfades in einem gerichteten Graphen mit reellwertigen Kantengewichten ist **NP**-vollständig.

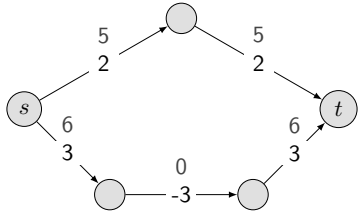
Bemerkungen: Verbietet man negative Kreise wird das Problem jedoch wieder in Polynomialzeit lösbar.

Kürzeste Pfade: Falsche Ansätze

Algorithmus von Dijkstra: Schlägt fehl bei negativen Kantengewichten.



Neugewichtung: Zu jedem Kantengewicht eine Konstante dazugeben schlägt fehl.



Kürzeste Pfade: Bellman's Gleichungen

Sei $G = (V, E)$ ein gerichteter Graph ohne negative Kreise mit Kantengewichten c_{vw} und $t \in V$, dann gilt für die Länge $OPT(v)$ eines kürzesten v-t Pfades P in G:

$$OPT(t) = 0$$

$$OPT(v) = \min_{(v,w)\in E} \{c_{vw} + OPT(w)\}, \forall v \in V, v \neq t.$$

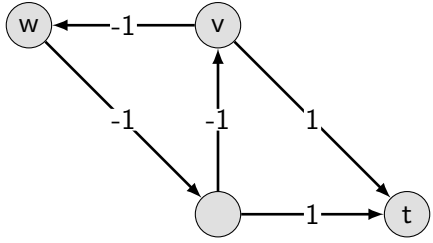
Beweis:

Wir zeigen mit Induktion über die Anzahl Kanten auf dem kürzesten v-t Pfad, dass $OPT(v)$ die Länge dieses kürzesten v-t Pfades ist.

- Aussage gilt für $v = t$ mit 0 Kanten.
- Sei P ein kürzester v-t Pfad mit ℓ Kanten und (v, w) die erste Kante von P.
- Da es keine negativen Kreise gibt, liegt ein Knoten nie zweimal auf einem kürzesten Kantenzug.
- $P - (v, w)$ ist ein kürzester w-t Pfad mit $\ell - 1$ Kanten, der v nicht enthält. Wegen der Induktionsvoraussetzung ist seine Länge $OPT(w)$.
- Damit ist aber $c_{vw} + OPT(w)$ die Länge von P und $OPT(v) \leq c_{vw} + OPT(w)$.
- Wäre $OPT(v) < c_{vw} + OPT(w)$, so gäbe es einen kürzeren v-t Pfad als P über einen anderen Zwischenknoten $w' \rightarrow$ Widerspruch

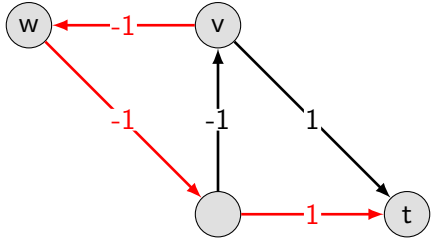
Bellman's Gleichungen: Gegenbeispiel

Falls G negative Kreise enthält, gelten Bellman's Gleichungen nicht mehr:



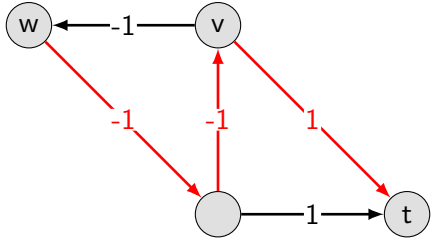
Bellman's Gleichungen: Gegenbeispiel

Falls G negative Kreise enthält, gelten Bellman's Gleichungen nicht mehr:



Bellman's Gleichungen: Gegenbeispiel

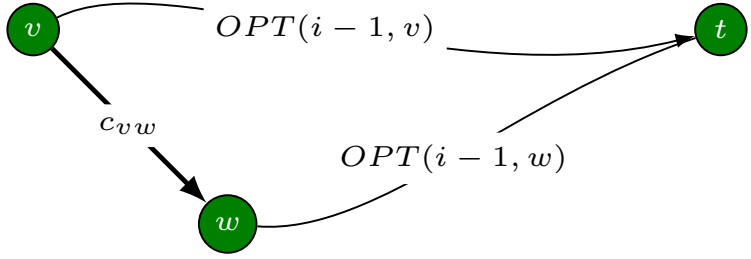
Falls G negative Kreise enthält, gelten Bellman's Gleichungen nicht mehr:



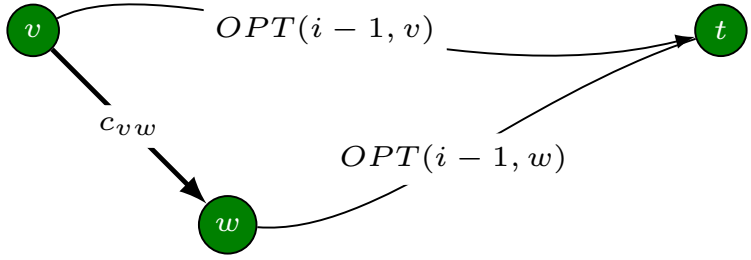
Kürzeste Pfade: Dynamische Programmierung

Definition: $OPT(i, v) =$ Länge eines kürzesten v-t Pfades P, der höchstens i Kanten benutzt.

- Fall 1: P benutzt höchstens $i - 1$ Kanten.
  - $OPT(i, v) = OPT(i - 1, v)$
- Fall 2: P benutzt genau i Kanten.
  - Sei (v, w) die erste Kante auf P. Dann besteht der kürzeste Pfad aus (v, w) und dem besten w-t Pfad, der höchstens $i - 1$ Kanten benutzt.



Kürzeste Pfade: Dynamische Programmierung

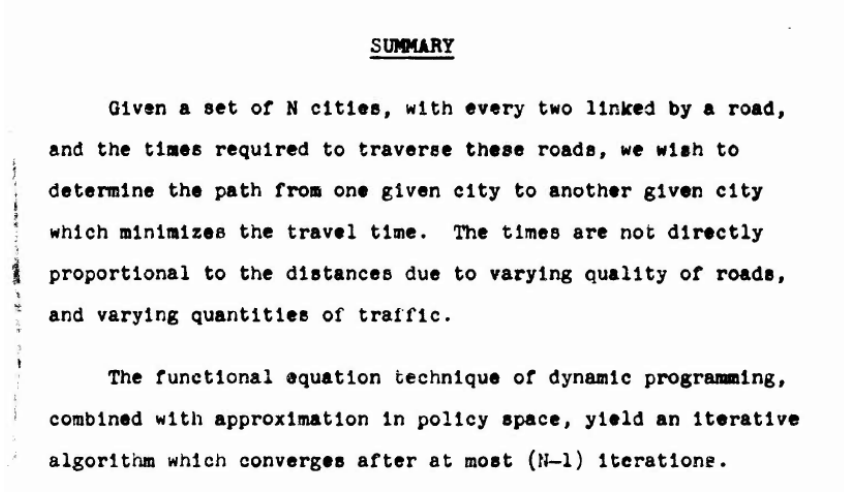


$$OPT(i,v) = \begin{cases} 0 & \text{wenn } i = 0 \text{ und } v = t \\ \infty & \text{wenn } i = 0 \text{ und } v \neq t \\ \min\left\{OPT(i-1,v), \min\limits_{(v,w)\in E}\{c_{vw} + OPT(i-1,w)\}\right\} & \text{ansonsten} \end{cases}$$

Anmerkung: Aufgrund von Bellman's Gleichungen ist $OPT(n-1,v) = OPT(v) =$ Länge eines kürzesten v-t Pfades, wenn es keine negativen Kreise gibt.

Bellman-Ford Algorithmus

Der hier vorgestellte Algorithmus wurde unabhängig von Richard Bellman und Lester Ford im Jahre 1956 entwickelt und wird deshalb auch häufig als Bellman-Ford-Algorithmus bezeichnet.



SUMMARY

Given a set of N cities, with every two linked by a road, and the times required to traverse these roads, we wish to determine the path from one given city to another given city which minimizes the travel time. The times are not directly proportional to the distances due to varying quality of roads, and varying quantities of traffic.

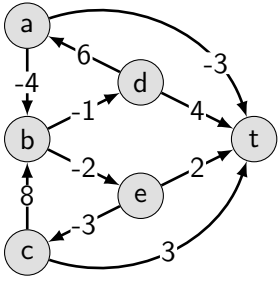
The functional equation technique of dynamic programming, combined with approximation in policy space, yield an iterative algorithm which converges after at most (N-1) iterations.

Kürzeste Pfade: Implementierung

```
Shortest-Path(G,s,t):
foreach node v ∈ V
    M[0,v] ← ∞
M[0,t] ← 0
for i ← 1 bis n − 1
    foreach Knoten v ∈ V
        M[i,v] ← M[i − 1,v]
    foreach Kante (v,w) ∈ E
        M[i,v] ← min (M[i,v],c_vw + M[i − 1,w])
return M[n − 1,s]
```

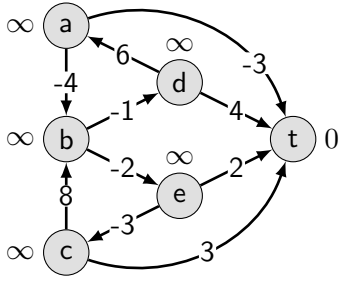
Analyse: $O(mn)$ Zeit, $O(n^2)$ Platz.

Kürzeste Pfade: Beispiel



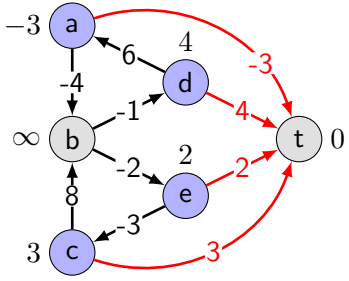
	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Kürzeste Pfade: Beispiel



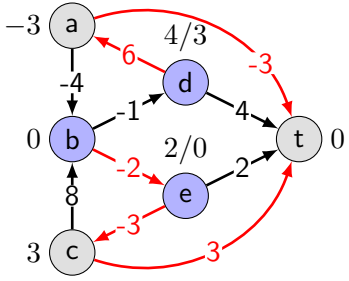
	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Kürzeste Pfade: Beispiel



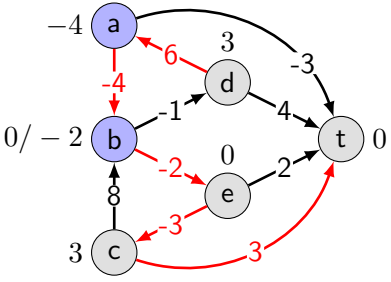
	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Kürzeste Pfade: Beispiel



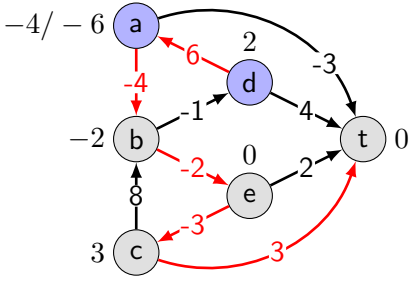
	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Kürzeste Pfade: Beispiel



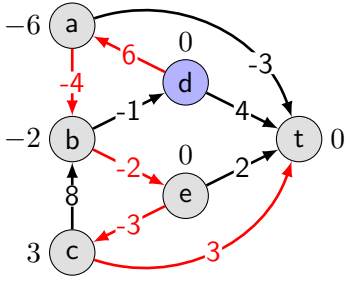
	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Kürzeste Pfade: Beispiel



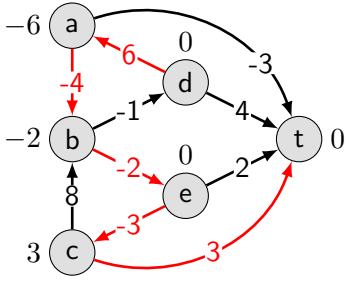
	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Kürzeste Pfade: Beispiel



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Kürzeste Pfade: Beispiel



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Kürzeste Pfade: Implementierung

Einen kürzesten Pfad finden: Verwalte den Nachfolger für jeden Knoten.

Korrektheit der Ergebnisse:

- Der beschriebene Algorithmus liefert immer eine korrekte Lösung, wenn keine negativen Kreise im Graphen vorhanden sind.
- Wenn negative Kreise vorhanden sind, kann der Algorithmus falsche Ergebnisse liefern.

Überprüfung auf negative Kreise:

- Nach dem Terminieren des Algorithmus.
- Für jeden Knoten v wird überprüft: Falls $OPT(n, v) < OPT(n-1, v)$, dann gibt es einen negativen Kreis.

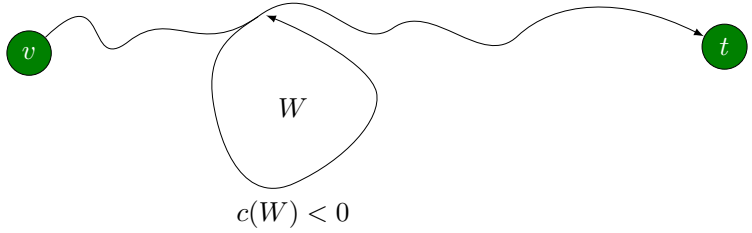
Negative Kreise erkennen

Lemma: Wenn $OPT(n, v) < OPT(n-1, v)$ für einen Knoten v, dann enthält (ein beliebiger) kürzester v-t Kantenzug K mit maximal n Kanten einen Kreis W. Außerdem hat W negative Kosten.

Beweis: durch Widerspruch

- Da $OPT(n, v) < OPT(n-1, v)$, wissen wir, dass K genau n Kanten hat.
- Mit Hilfe des Schubfachprinzips kann man zeigen, dass K einen gerichteten Kreis W enthalten muss.
- Löschen von W ergibt einen v-t Kantenzug mit $< n$ Kanten $\Rightarrow W$ hat negative Kosten.

□



Negative Kreise erkennen

Lemma: Falls G einen negativen Kreis enthält von dem aus t erreicht werden kann, dann gibt es eine Kante (v, u), sodass $OPT(n-1, v) > c_{vu} + OPT(n-1, u)$ (und somit $OPT(n, v) < OPT(n-1, v)$).

Beweis:

- Sei C ein beliebiger Kreis in G auf den Knoten $(v_1, \ldots, v_k)$, dann gilt: $m = \sum_{i=1}^{k} [OPT(n-1, v_i) - OPT(n-1, v_{i+1 \mod k})] = 0$.
- Falls nun C ein negativer Kreis ist, ergibt sich $\sum_{i=1}^{k} c_{v_i v_{i+1 \mod k}} < 0 = m$.
- D.h. es muss mindestens ein i existieren mit $c_{v_i v_{i+1 \mod k}} < OPT(n-1, v_i) - OPT(n-1, v_{i+1 \mod k})$ und somit gilt für die Kante $(v_i, v_{i+1 \mod k})$, dass $c_{v_i v_{i+1 \mod k}} + OPT(n-1, v_{i+1 \mod k}) < OPT(n-1, v_i)$.

□

Negative Kreise erkennen

Aus den beiden vorherigen Lemmas folgt nun:

Theorem: G enthält einen negativen Kreis von dem aus t erreicht werden kann genau dann wenn ein Knoten v mit $OPT(n, v) < OPT(n-1, v)$ existiert.

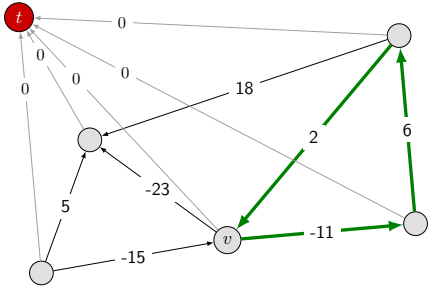
Negative Kreise erkennen

Theorem: Man kann in Zeit $O(nm)$ entscheiden, ob ein Graph einen negativen Kreis hat und diesen im positiven Fall auch ausgeben.

Beweis:

- Gib einen neuen Knoten t hinzu und verbinde alle Knoten mit t mit einer Kante mit den Kosten 0.
- Überprüfe, ob $OPT(n, v) = OPT(n-1, v)$ für alle Knoten v.
  - Wenn ja, dann gibt es keine negativen Kreise
  - Wenn nein, dann extrahiere den Kreis aus dem kürzesten Kantenzug von v zu t mit maximal n Kanten

□



Kürzeste Pfade: Praktische Verbesserungen

Praktische Verbesserungen:

- Verwalte nur ein Array M[v] = kürzester v-t Pfad, den wir bisher gefunden haben.
- Brich den Algorithmus ab, sobald sich nach einer vollen Iteration kein Eintrag in M mehr geändert hat.

Theorem: Beim Ablauf des Algorithmus ist M[v] die Länge eines v-t Pfades und nach i Runden von Updates ist der Wert M[v] nicht größer als die Länge eines kürzesten v-t Pfades, der $\leq i$ Kanten benutzt.

Gesamte Auswirkung:

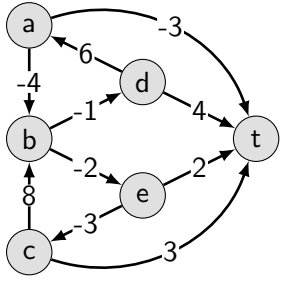
- Speicher: $O(m + n)$.
- Laufzeit: $O(mn)$ Worst-Case, aber wesentlich schneller in der Praxis.

Bellman-Ford: Effiziente Implementierung

```
Push-Based-Shortest-Path(G,s,t):
foreach Knoten v ∈ V
    M[v] ← ∞
    Nachfolger[v] ← ∅

M[t] = 0
for i ← 1 bis n − 1
    foreach Kante (v, w) ∈ E
        if M[v] > M[w] + c_vw
            M[v] ← M[w] + c_vw
            Nachfolger[v] ← w
    if kein M[w] Wert ändert sich in Iteration i, stop.
return M[s]
```

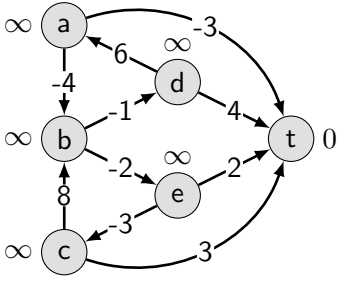
Kürzeste Pfade: Beispiel



	0	1	2
t	0	0	0
a	∞	-6	-6
b	∞	-2	-2
c	∞	3	3
d	∞	0	0
e	∞	0	0

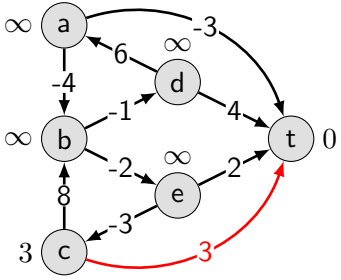
Die Anzahl der Iterationen hängt nun stark von der Reihenfolge ab in der die Kanten in einer Iteration durchlaufen werden.

Kürzeste Pfade: Beispiel



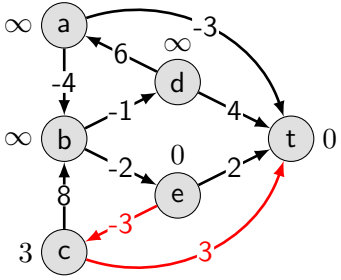
	0	1	2
t	0	0	0
a	∞	-6	-6
b	∞	-2	-2
c	∞	3	3
d	∞	0	0
e	∞	0	0

Kürzeste Pfade: Beispiel



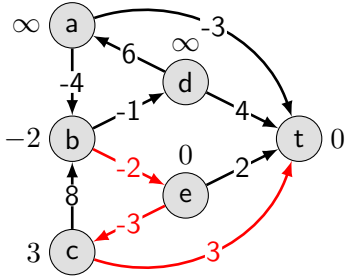
	0	1	2
t	0	0	0
a	∞	-6	-6
b	∞	-2	-2
c	∞	3	3
d	∞	0	0
e	∞	0	0

Kürzeste Pfade: Beispiel



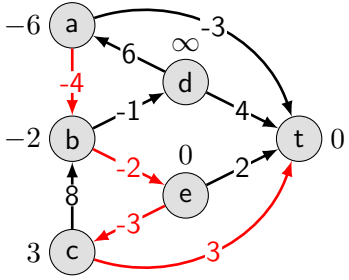
	0	1	2
t	0	0	0
a	∞	-6	-6
b	∞	-2	-2
c	∞	3	3
d	∞	0	0
e	∞	0	0

Kürzeste Pfade: Beispiel



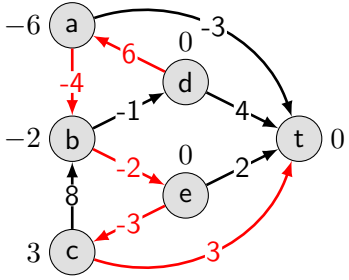
	0	1	2
t	0	0	0
a	∞	−6	−6
b	∞	−2	−2
c	∞	3	3
d	∞	0	0
e	∞	0	0

Kürzeste Pfade: Beispiel



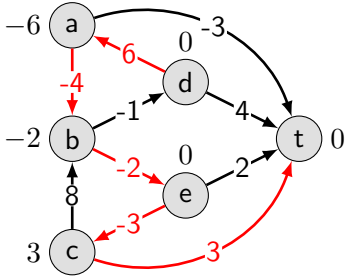
	0	1	2
t	0	0	0
a	∞	−6	−6
b	∞	−2	−2
c	∞	3	3
d	∞	0	0
e	∞	0	0

Kürzeste Pfade: Beispiel



	0	1	2
t	0	0	0
a	∞	-6	-6
b	∞	-2	-2
c	∞	3	3
d	∞	0	0
e	∞	0	0

Kürzeste Pfade: Beispiel



	0	1	2
t	0	0	0
a	∞	-6	-6
b	∞	-2	-2
c	∞	3	3
d	∞	0	0
e	∞	0	0

Finden eines kürzesten Pfades: Korrektheit

Den kürzesten Pfad finden: Verwalte den Nachfolger für jeden Knoten.

Definition: Ein Knoten w ist der Nachfolger eines Knoten v (an einer beliebigen Stelle im Algorithmus), falls $M[v]$ zuletzt auf $c_{vw} + M[w]$ geändert wurde.

Beobachtung: Falls w momentan der Nachfolger von v ist dann gilt:
$M[v] \geq c_{vw} + M[w]$.

Finden eines kürzesten Pfades: Korrektheit

Lemma: Falls der Nachfolgergraph einen Kreis hat (an einer beliebigen Stelle im Algorithmus), dann ist der Kreis negativ.

Beweis:

- Seien $v_1, \ldots, v_k$ die Knoten auf einem Kreis C im Nachfolgergraph und sei (v_k, v_1) die letzte Kante in C, die vom Algorithmus hinzugefügt wurde.
- Dann gilt $M[v_i] \geq c_{v_i v_{i+1}} + M[v_{i+1}]$ für alle i mit $1 \leq i < k$ und (unmittelbar bevor die Kante (v_k, v_1) gesetzt wird) $M[v_k] > c_{v_k v_1} + M[v_1]$.
- Nach Aufsummieren aller Ungleichungen verschwinden die $M[v_i]$-Terme und wir erhalten: $0 > \sum_{i=1}^{k-1} c_{v_i v_{i+1}} + c_{v_k v_1}$, also ist C ein negativer Kreis.

□

Finden eines kürzesten Pfades: Korrektheit

Aus dem vorherigen Lemma folgt, dass falls G keine negativen Kreise enthält, der Nachfolgergraph kreisfrei ist.

Nach Beendigung des Algorithmus auf einem Graphen G ohne negative Kreise, folgt nun:

- jeder Knoten v von dem aus t erreichbar ist hat genau einen Nachfolger w und $OPT(n-1, v) = c_{vw} + OPT(n-1, w)$.
- Also enthält der Nachfolgergraph für jeden solchen Knoten genau einen Pfad nach t und dieser Pfad ist ein kürzester Pfad.

Dynamische Programmierung Zusammenfassung

Vorgehen beim Entwurf von Dynamischen Programmen

- Verstehe und charakterisiere die Struktur des Problems und seiner optimalen Lösungen anhand von überlappenden Teilproblemen und optimalen Teillösungen
- Definiere rekursiv den Wert einer optimalen Lösung
- Berechne und speichere die Werte der optimalen (Teil-)Lösungen in einer Tabelle
- Konstruiere die optimale Lösung durch Rückverfolgung der Optimierungsentscheidungen des Algorithmus

Techniken der Dynamischen Programmierung

- Binäre Auswahl: z.B. gewichtetes Interval Scheduling
- Mehrfachauswahl: z.B. segmented least squares, kürzeste Pfade
- Einführen zusätzlicher Variablen: z.B. Knapsack

Top-down vs. bottom-up: rekursive vs. iterative Berechnung

Ergänzende Literatur

J. Kleinberg and E. Tardos. *Algorithm Design*. Pearson, 2005. Kapitel 6.

Optimierung – Approximation

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 15. Juni 2023

Vorlesungsfolien

Optimierung: Roadmap

Branch-and-Bound

Dynamische Programmierung

Approximation(salgorithmen): Erzeuge in polynomieller Zeit eine Näherungslösung, die eine Gütegarantie besitzt. Die Güte eines Algorithmus sagt etwas über die Fähigkeit aus, optimale Lösungen gut oder schlecht anzunähern.

Heuristische Verfahren

Gütegarantie

Annahmen:

- Sei A ein Algorithmus, der für jede Instanz x eines Problems X eine gültige Lösung mit Lösungswert $c_{\mathrm{A}}(x) > 0$ liefert.
- Sei $c_{\mathrm{opt}}(x) > 0$ der Wert einer optimalen Lösung.

Für Minimierungsprobleme gilt: Falls es ein $\varepsilon > 0$ gibt, sodass für alle Instanzen x von X

$$\frac{c_{\mathrm{A}}(x)}{c_{\mathrm{opt}}(x)} \leq \varepsilon$$

gilt, dann ist A ein ε-Approximationsalgorithmus und der Wert ε heißt Gütegarantie.

Gütegarantie

Für Maximierungsprobleme gilt:

$$\frac{c_{\mathrm{A}}(x)}{c_{\mathrm{opt}}(x)} \geq \varepsilon$$

Es folgt:

- für Minimierungsprobleme: $\varepsilon \geq 1$,
- für Maximierungsprobleme: $0 \leq \varepsilon \leq 1$,
- $\varepsilon = 1 \quad \Leftrightarrow \quad A$ ist ein exakter Algorithmus

Approximationsalgorithmus für Vertex Cover

Vertex Cover

Ein Vertex Cover eines Graphen $G = (V, E)$ ist eine Menge $C \subseteq V$, so dass jede Kante des Graphen zu mindestens einem Knoten aus C inzident ist.

Minimales Vertex Cover: Finde für einen gegebenen Graphen ein Vertex Cover mit kleinster Größe $|C|$.

Aufwand: Ist NP-schwer, d.h. kann i.A. vermutlich nicht in polynomieller Zeit gelöst werden.

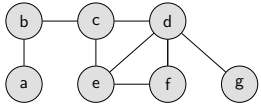
Minimales Vertex Cover: Approximationsalgorithmus

2-Approximationsalgorithmus: Findet ein Vertex Cover in einem Graphen $G = (V, E)$, das höchstens zwei Mal so groß wie ein optimales (minimales) Vertex Cover ist.

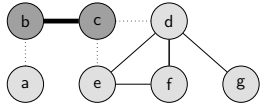
```
Approx-Vertex-Cover(G):
C ← ∅
while E ≠ ∅
    Wähle eine beliebige Kante (u, v) ∈ E
    C ← C ∪ {u, v}
    Entferne aus E alle Kanten, die inzident
          zu u oder v sind
return C
```

Approx-Vertex-Cover: Beispiel

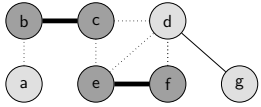
Beispielhafter Ablauf des Algorithmus, Vergleich mit optimaler Lösung.



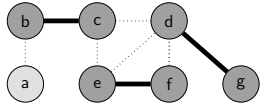
(a) Ausgangssituation



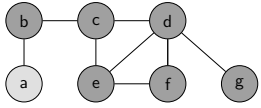
(b) 1. Kante auswählen



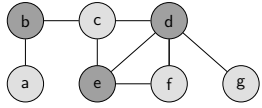
(c) 2. Kante auswählen



(d) 3. Kante auswählen



(e) Ergebnis



(f) Optimale Lösung

Minimales Vertex Cover: Gütegarantie

Theorem: Approx-Vertex-Cover ist ein polynomieller Algorithmus mit einer Gütegarantie von 2.

Laufzeit: Ist polynomiell.

- In der Schleife werden nacheinander Kanten ausgewählt, zwei Knoten zu C hinzugefügt und dann alle inzidenten Kanten gelöscht.
- Mit Adjazenzlisten kann dieser Algorithmus mit Laufzeit $O(n+m)$ implementiert werden.
  - *Wir schreiben $n = |V|$ und $m = |E|$.*

Minimales Vertex Cover: Gütegarantie

Gütegarantie:

- Sei M die Kantenmenge, die vom Algorithmus ausgewählt wird; M ist ein Matching.
- In einem kleinsten Vertex Cover C^* muss gelten: Für jede Kante $e \in M$ existiert ein Knoten $v \in C^*$, der inzident zu e ist.
- Daher muss C^* zumindest einen der Endpunkte jeder Kante $e \in M$ enthalten.
- Es folgt, dass $c_{\text{opt}} = |C^*| \geq |M|$.
- Da Approx-Vertex-Cover (kurz AVC) ein Vertex Cover der Größe $2|M|$ findet, gilt für alle Instanzen x:

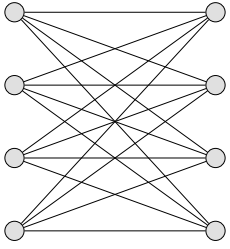
$$c_{\text{AVC}}(x) = 2|M| \leq 2 \cdot c_{\text{opt}}(x)$$

Minimales Vertex Cover: Gütegarantie

Approx-Vertex-Cover: Für einen bipartiten vollständigen Graphen (kurz $K_{n,n}$) ist die Schranke sogar eine scharfe Schranke.

Hinweis: Ein einfacher Graph heißt bipartit oder paar, wenn sich seine Knoten in zwei disjunkte Teilmengen A und B aufteilen lassen, sodass zwischen den Knoten innerhalb einer jeden Teilmenge keine Kanten verlaufen.

Beispiel: Approx-Vertex-Cover würde alle Knoten auswählen. Die optimale Lösung besteht aus den Knoten einer Seite.



Minimales Vertex Cover: Alternativer Algorithmus

Alternativer Algorithmus: Wählt immer einen Knoten mit aktuell maximalem Grad.

```
Approx-Vertex-Cover2(G):
C ← ∅
while E ≠ ∅
    Wähle einen Knoten u mit maximalem Grad im aktuellen Graphen
    C ← C ∪ {u}
    Entferne aus E alle Kanten, die inzident zu u sind
return C
```

Minimales Vertex Cover: Alternativer Algorithmus

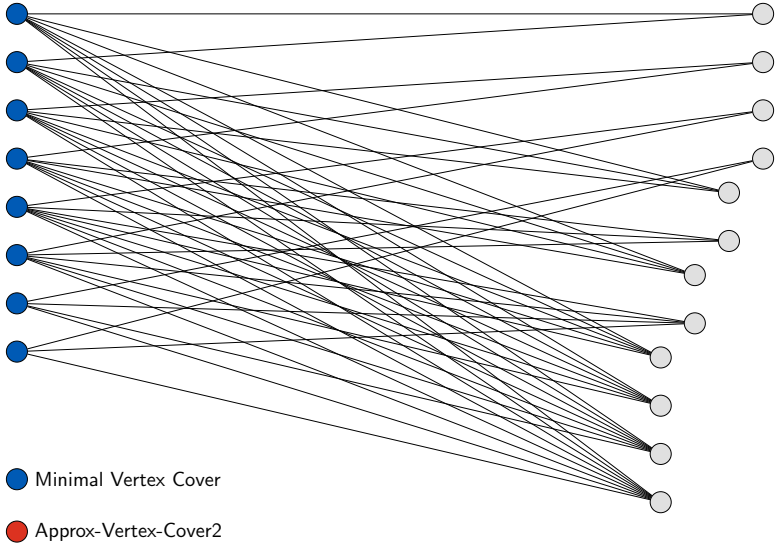
Alternativer Algorithmus: Wählt immer einen Knoten mit aktuell maximalem Grad.

```
Approx-Vertex-Cover2(G):
C ← ∅
while E ≠ ∅
    Wähle einen Knoten u mit maximalem Grad im aktuellen Graphen
    C ← C ∪ {u}
    Entferne aus E alle Kanten, die inzident zu u sind
return C
```

Gütegarantie: Man kann zeigen, dass dieser Algorithmus eine logarithmische Gütegarantie hat. Die scheinbar intelligentere Auswahl führt hier nicht zu einer Verbesserung!

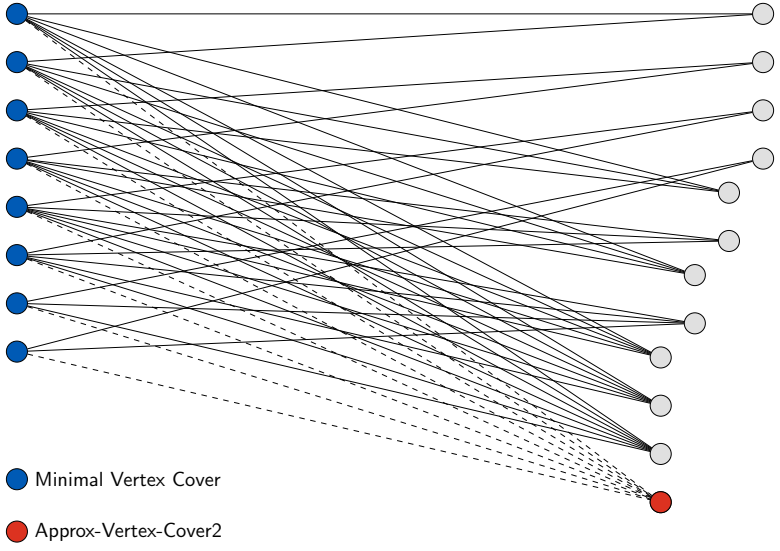
Minimales Vertex Cover: Alternativer Algorithmus

Schlechtes Beispiel:



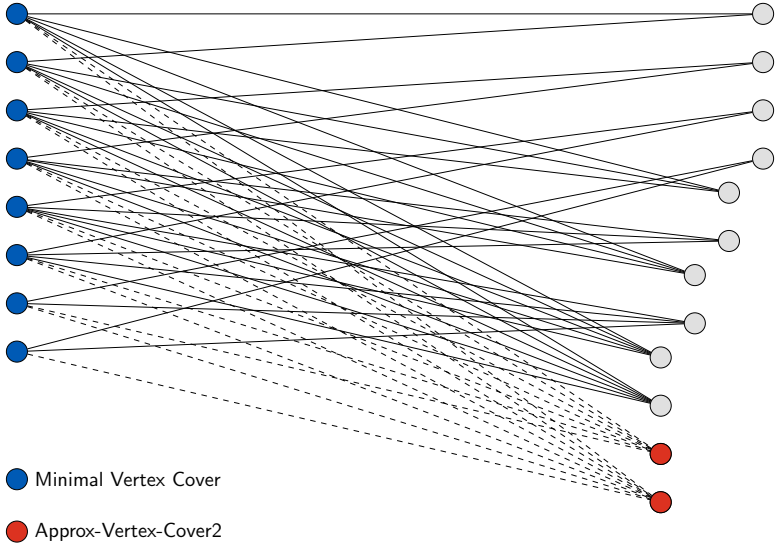
Minimales Vertex Cover: Alternativer Algorithmus

Schlechtes Beispiel:



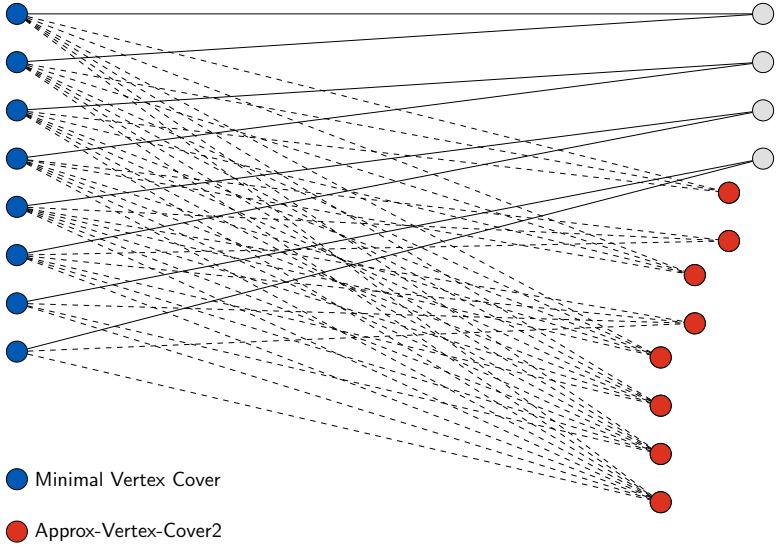
Minimales Vertex Cover: Alternativer Algorithmus

Schlechtes Beispiel:



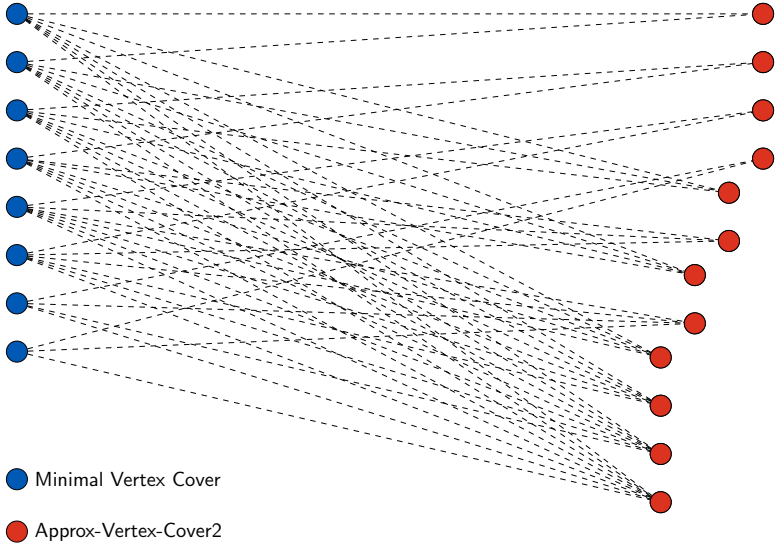
Minimales Vertex Cover: Alternativer Algorithmus

Schlechtes Beispiel:



Minimales Vertex Cover: Alternativer Algorithmus

Schlechtes Beispiel:



Minimales Vertex Cover: Alternativer Algorithmus

Konstruktionsschema für schlechtes Beispiel:

- Wähle n Knoten auf der linken Seite.
- Für $i = 2, \ldots n$ geben wir jeweils eine Menge R_i mit $\lfloor \frac{n}{i} \rfloor$ Knoten auf der rechten Seite hinzu.
- Jeder Knoten aus der Menge R_i hat einen Grad i und ist jeweils mit i Knoten auf der linken Seite verbunden.

Beispiel aus vorheriger Folie:

- Links gibt es 8 Knoten.
- Rechts gibt es 12 Knoten:
  R_2 (4 Knoten mit Grad 2), R_3 (2 Knoten mit Grad 3),
  R_4 (2 Knoten mit Grad 4), R_5 (1 Knoten mit Grad 5),
  R_6 (1 Knoten mit Grad 6), R_7 (1 Knoten mit Grad 7),
  R_8 (1 Knoten mit Grad 8).

Minimales Vertex Cover: Alternativer Algorithmus

Gütegarantie: Für n Knoten auf der linken Seite.

- Approx-Vertex-Cover2 wählt alle Knoten auf der rechten Seite, d.h.

$$\sum_{i=2}^{n} |R_i| = \sum_{i=2}^{n} \left\lfloor \frac{n}{i} \right\rfloor \geq \sum_{i=2}^{n} \left(\frac{n}{i} - 1 \right) \geq n \sum_{i=1}^{n} \frac{1}{i} - 2n = n(H_n - 2)$$

  Knoten. Dabei ist H_n die n-te harmonische Zahl.
- Es gilt $H_n = \ln n + \Theta(1)$.
- Minimales Vertex Cover umfasst n Knoten. Gütegarantie ist daher:

$$\frac{n(H_n - 2)}{n} = \Omega(\log n)$$

Spanning-Tree-Heuristik (ST) für das symmetrische TSP

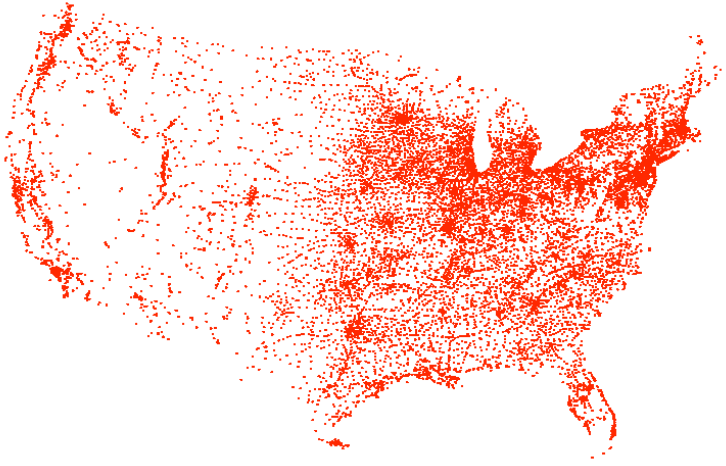
Traveling Salesman Problem (TSP): Wiederholung

Traveling Salesman Problem (TSP):

- Man sucht eine Reihenfolge (Tour) für den Besuch mehrerer Orte, sodass die gesamte Reisestrecke eines Handlungsreisenden möglichst kurz ist (Minimierungsproblem).
- Jeder Ort wird genau einmal besucht und nach dem letzten Ort wird zum ersten zurückgekehrt.

Traveling Salesman Problem (TSP): Beispiel

Beispiel: Die 13509 Städte der USA mit mehr als 500 Einwohner.



Traveling Salesman Problem (TSP): Beispiel

Beispiel: Die 13509 Städte der USA mit mehr als 500 Einwohner: Optimale Tour

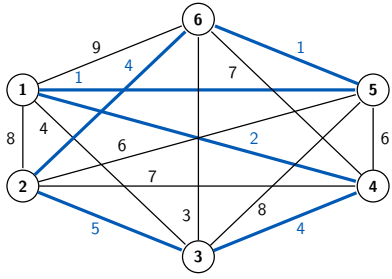
Traveling Salesman Problem (TSP)

Darstellung: Die kürzesten Wege zwischen allen Knotenpaaren können durch einen gewichteten vollständigen Graphen repräsentiert werden.

Vollständiger Graph:

- Ist ein schlichter Graph, in dem jeder Knoten mit jedem anderen Knoten durch eine Kante verbunden ist.
- Ein vollständiger Graph mit n Knoten wird als K_n bezeichnet.

Beispiel: 6 Orte, minimale Tour der Länge 17.



Symmetrisches TSP:

Symmetrisches TSP:

- Gegeben ist ein ungerichteter vollständiger Graph $G = (V, E)$ mit Distanzmatrix c mit $c_{ii} = +\infty$ und $c_{ij} \geq 0$.
- Für alle Knotenpaare (i, j) sind die Distanzen in beide Richtungen identisch, d.h. es gilt $c_{ij} = c_{ji}$.
- Jede Tour hat dieselbe Länge in beide Richtungen.

Hinweis: Das TSP ist NP-schwer
(Beweis durch Reduktion von HAM-CYCLE).

Spanning-Tree-Heuristik (ST) für das sym. TSP

Spanning-Tree-Heuristik: Tour wird aus einem Minimum Spanning Tree (MST) für den gegebenen Graphen G abgeleitet:

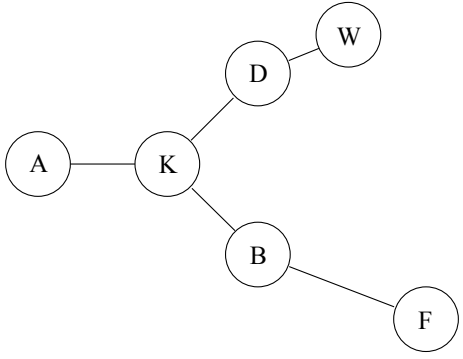
Vorgehen:

1. Bestimme einen MST (V, B_1) von G.
2. Verdopple alle Kanten in B_1, das ergibt Graph (V, B_2).
3. Bestimme eine Eulertour F im Graphen (V, B_2). Gib dieser Tour eine Orientierung, wähle einen Knoten $i \in V$, markiere i, setze $p \leftarrow i, T \leftarrow \emptyset$.
4. Sind alle Knoten markiert, setze $T \leftarrow T \cup \{(p, i)\}$ und retourniere T als Ergebnis-Tour.
5. Laufe von p entlang der Orientierung von F bis ein unmarkierter Knoten q erreicht ist. Setze $T \leftarrow T \cup \{(p, q)\}$, markiere q, setze $p \leftarrow q$ und gehe zu (4).

Spanning-Tree-Heuristik (ST): Beispiel

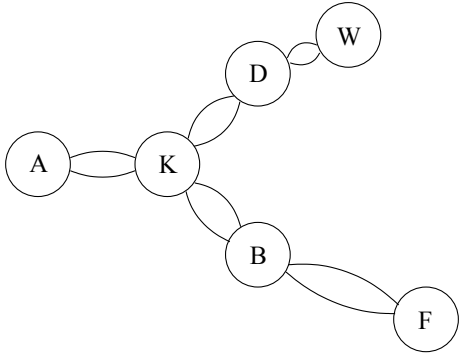
Ausgangspunkt: Vollständiger Graph mit 6 Knoten.

Schritt 1: MST (V, B_1) sieht beispielsweise folgendermaßen aus:



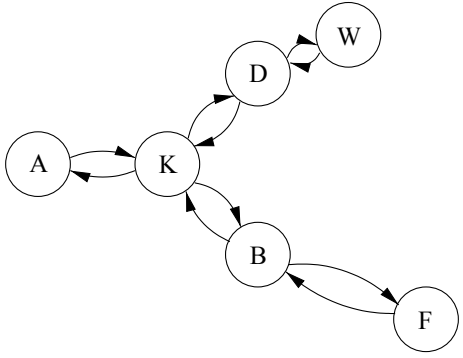
Spanning-Tree-Heuristik (ST): Beispiel

Schritt 2: Alle Kanten im MST werden verdoppelt (V, B_2).



Spanning-Tree-Heuristik (ST): Beispiel

Schritt 3: Bestimme Eulertour F, die jede Kante genau einmal enthält.

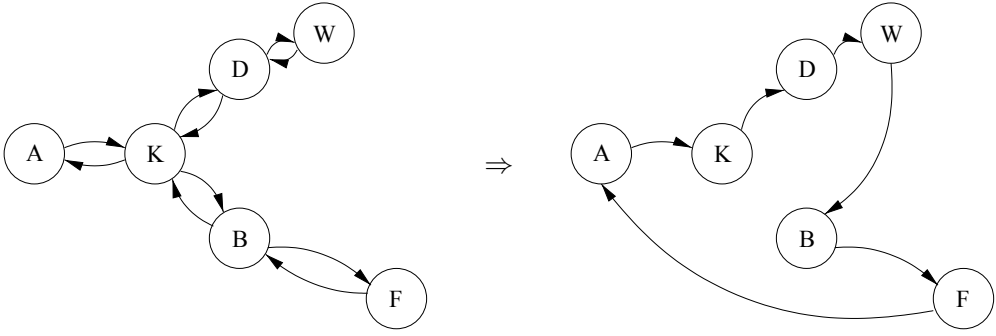


Eulertour: $F =$ (A, K, D, W, D, K, B, F, B, K, A)

Spanning-Tree-Heuristik (ST): Beispiel

Schritt 4 und 5: Eulertour $\Rightarrow$ TSP-Tour

- A ist der Startknoten.
- In der Tour wird jeder Knoten nur einmal besucht.
- Wird ein Knoten besucht, dann wird er markiert.
- Der nächste TSP-Tour-Knoten ist immer der nächste unmarkierte Knoten auf der Eulertour.



Spanning-Tree-Heuristik (ST) für das sym. TSP

Eulertour: Eine Rundtour, die jede Kante des Graphen genau einmal beinhaltet.

Theorem: Eine Eulertour existiert in einem ungerichteten Graphen genau dann wenn er zusammenhängend ist und jeder Knoten einen geraden Grad hat.

Hinweis: Durch die Verdopplung der Kanten in ST hat jeder Knoten geraden Grad.
Eine Eulertour existiert somit immer.

Spanning-Tree-Heuristik (ST): Gütegarantie

Laufzeit: Der erste Schritt (MST finden) kann beispielsweise mit Prims Algorithmus in Zeit $O(n^2)$ gelöst werden. Die restlichen Schritte sind nicht aufwendiger und daher läuft die gesamte Heuristik in Zeit $O(n^2)$.

Algorithmus von Hierholzer (1873)

Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren.

Von Carl Hierholzer.

Mitgetheilt von Chr. Wiener*).

In einem beliebig verschlungenen Linienzuge mögen *Zweige* eines Punktes diejenigen verschiedenen Theile des Zuges heissen, auf welchen man den fraglichen Punkt verlassen kann. Ein Punkt mit mehreren Zweigen heisse ein *Knotenpunkt*, der so vielfach genannt werde, als

Hinweis: Findet eine Eulertour (falls vorhanden) in einem Graphen G in linearer Zeit.

Eulertour: Algorithmus von Hierholzer

Grundlegende Idee:

- Beginnend von einem Startknoten wird ein Pfad (nicht notwendigerweise einfach) mit einer noch nicht benutzten Kante fortgesetzt. Wenn alle Knoten geraden Grad haben, kehrt man wieder zum Ausgangsknoten zurück. Dieser geschlossene Pfad bildet einen Zyklus mit den Knoten $v_1, v_2, ..., v_k, v_1$.

- Die Kanten $E(Z) = \{(v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k), (v_k, v_1)\}$ des gefundenen Zyklus Z werden aus dem Graphen entfernt. Im Restgraphen sind, wenn eine Eulertour vorhanden ist, wiederum alle Knotengrade gerade.

- Gibt es in einem Zyklus Z einen Knoten mit einem Grad größer 0, dann kann man von dort aus wiederum einen Zyklus bilden.

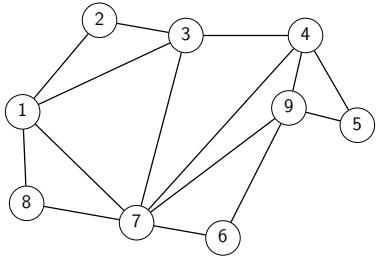
Eulertour: Algorithmus von Hierholzer

Vorgehen:

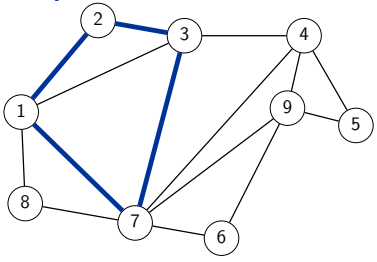
1. Wähle einen beliebigen Knoten v_0 des Graphen und konstruiere von v_0 ausgehend einen geschlossenen Pfad Z in G, der keine Kante in G zweimal durchläuft. Wir nennen Z einen Zyklus.
2. Wenn Z eine Eulertour ist (also alle Knoten beinhaltet), terminiere. Andernfalls:
3. Lösche nun alle Kanten in Z aus G.
4. An einem Knoten von Z, dessen Grad größer 0 ist, lässt man nun einen Zyklus Z' entstehen, der keine Kante in G zweimal enthält.
5. Füge in Z den zweiten Zyklus Z' ein, indem der Startpunkt von Z' beim ersten Auftreten in Z durch alle Knoten von Z' in der durchlaufenen Reihenfolge ersetzt wird.
6. Fahre bei Schritt 2 fort.

Eulertour: Algorithmus von Hierholzer: Beispiel

Ausgangsgraph:



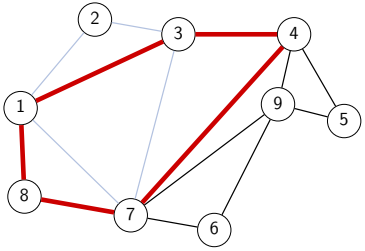
1. Zyklus:



$Z = (1,2,3,7,1)$

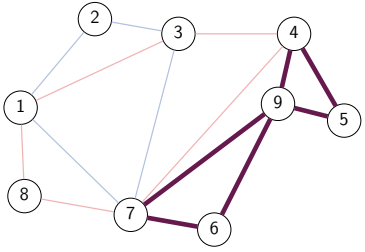
Eulertour: Algorithmus von Hierholzer: Beispiel

2. Zyklus:



$Z' = (1,3,4,7,8,1)$
$Z = (1,3,4,7,8,1,2,3,7,1)$

3. Zyklus:

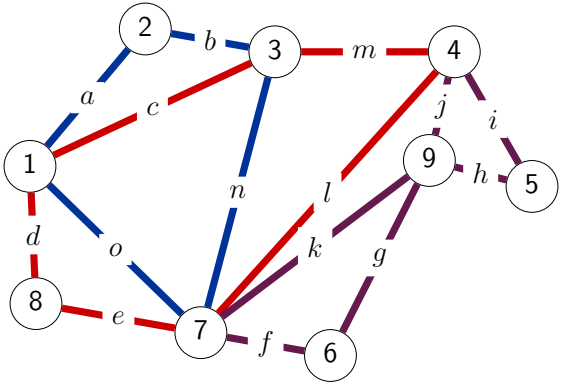


$Z' = (7,6,9,5,4,9,7)$
$Z = (1,3,4,7,6,9,5,4,9,7,8,1,2,3,7,1)$

Eulertour: Algorithmus von Hierholzer: Beispiel

Mögliche Eulertour: Weiteres Beispiel

- Kantenfolge: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o
- Knotenfolge: 1, 2, 3, 1, 8, 7, 6, 9, 5, 4, 9, 7, 4, 3, 7, 1



Laufzeit: Liegt in $O(n+m)$.

Ist das symmetrische TSP 2-approximierbar?

Theorem: Angenommen P$\neq$NP. Dann gibt es keinen polynomiellen 2-Approximationsalgorithmus für das symmetrische TSP.

Beweis: (durch Widerspruch)

- Angenommen, es existiert ein polynomieller Approximationsalgorithmus A mit einer Gütegarantie 2.
- Sei $G = (V, E)$ ein Graph, für den wir bestimmen wollen, ob der Graph einen Hamiltonkreis enthält ($\mathrm{HAM\text{-}CYCLE}$-Problem ist NP-vollständig).
- Wir möchten jetzt A benutzen, um den Hamiltonkreis effizient zu finden.
- Dazu wird G in eine TSP-Instanz $G' = (V, E', c)$ transformiert.

Ist das symmetrische TSP 2-approximierbar?

Beweis (Fortsetzung):

- Transformation von G in eine TSP-Instanz $G' = (V, E', c)$:
  Sei (V, E') der vollständige Graph:
  - $E' = \{(u, v) : u, v \in V, \quad u \neq v\}$.
  - mit Kantenkosten:

$$c_{u,v} = \begin{cases} 1 & \text{wenn} \quad (u, v) \in E \\ 2 \cdot n + 1 & \text{sonst} \end{cases}$$

- G kann in polynomieller Zeit in G' transformiert werden.
- Wenn G einen Hamiltonkreis H enthält, dann werden jeder Kante von H die Kosten 1 zugewiesen und G' enthält eine Tour mit den Kosten n.
- Wenn G keinen Hamiltonkreis enthält, dann benutzt jede Tour in G' mindestens eine Kante, die sich nicht in E befindet. Solch eine Tour hat aber die Kosten von mindestens

$$(2 \cdot n + 1) + (n - 1) = 2 \cdot n + n = 3 \cdot n > 2 \cdot n$$

.

Ist das symmetrische TSP 2-approximierbar?

Beweis (Fortsetzung):

- Daher sind die Kosten einer Tour in G', die kein Hamiltonkreis in G ist, zumindest um den Faktor 3 größer als die einer Tour, die ein Hamiltonkreis in G ist.
- Nun wenden wir Algorithmus A auf die TSP-Instanz G' an.
- Algorithmus A liefert garantiert eine Tour zurück, deren Kosten höchstens um den Faktor 2 über denen einer optimalen Tour liegen.
- Wenn G einen Hamiltonkreis enthält, dann muss A ihn zurückliefern.
- Wenn G keinen Hamiltonkreis enthält, dann liefert A eine Tour mit Kosten größer als $2 \cdot n$.
- Daher kann A benutzt werden, einen Hamiltonkreis in polynomieller Zeit zu finden. Das ist aber ein Widerspruch dazu, dass $\mathrm{HAM\text{-}CYCLE}$ NP-schwer ist. Das könnte nur der Fall sein, wenn P=NP gilt. $\square$

Spanning-Tree-Heuristik (ST): Gütegarantie

Theorem: Ähnlich kann für ein allgemeines $\varepsilon > 1$ gezeigt werden, dass es es keinen polynomiellen ε-Approximationsalgorithmus für das symmetrische TSP gibt.
Das symmetrische TSP ist „nicht approximierbar“.

Beweis: Wie auf den vorherigen Folien, nur gilt jetzt $c(u, v) = \varepsilon \cdot n + 1$ wenn $(u, v) \notin E$. $\square$

Frage: Wozu dann der ganze Aufwand, wenn eine beliebige Approximation nicht effizient möglich ist?

Antwort: Unter einer einfachen Voraussetzung wird das Ergebnis besser.

Spanning-Tree-Heuristik (ST): Metrisches TSP

Metrisches TSP: Ein TSP heißt metrisch, wenn für die Distanzmatrix C die Dreiecksungleichung gilt, d.h. für alle Knoten i, j, k gilt

$$c_{ik} \leq c_{ij} + c_{jk}.$$

Hinweis: Insbesondere ist auch das Euklidische TSP, bei dem den Knoten Punkte in der euklidischen Ebene entsprechen und die Distanzen die euklidischen Distanzen sind, metrisch.

Theorem: Das metrische TSP besitzt einen polynomiellen Approximationsalgorithmus mit einer Gütegarantie von 2, d.h.

$$\frac{c_{\mathrm{ST}}(x)}{c_{\mathrm{opt}}(x)} \leq 2 \quad \text{für alle TSP-Instanzen } x$$

Spanning-Tree-Heuristik (ST): Metrisches TSP

Laufzeit: Die Spanning-Tree-Heuristik läuft in polynomieller Zeit (aufwendigster Schritt ist Ermitteln des MST im ersten Schritt).

Beweis für Gütegarantie: Es gilt

$$c_{\mathrm{ST}}(x) \leq c_{\mathrm{B}_2}(x) = 2c_{\mathrm{B}}(x) \leq 2c_{\mathrm{opt}}(x)$$

- *Gilt wegen der Dreiecksungleichung.*
- *Gilt, da B_2 durch Verdopplung der Kanten aus B entsteht.* *Gilt, da ein MST die kürzeste Möglichkeit ist, in einem Graphen alle Knoten zu verbinden.*

Lastverteilung (Load Balancing)

Lastverteilung

Eingabe: m identische Maschinen; n Jobs (Aufgaben), Job j hat Bearbeitungszeit t_j.

- Jeder Job j muss ununterbrochen auf einer Maschine ausgeführt werden.
- Eine Maschine kann nur einen Job auf einmal ausführen.

Definition: Sei J_i die Teilmenge von Jobs, die Maschine i zugewiesen wurde.
Die Last der Maschine i ist $L_i = \sum_{j \in J_i} t_j$.

Definition: Die Bearbeitungsdauer (*makespan*) ist die maximale Last auf irgendeiner Maschine $L = \max_{i=1,\ldots,n} L_i$.

Lastverteilung: Teile jeden Job einer Maschine so zu, dass die Bearbeitungsdauer minimiert wird.

Lastverteilung: List-Scheduling

List-Scheduling-Algorithmus:

- Berücksichtige n Jobs mit einer fixen Ordnung.
- Greedy-Algorithmus: Teile Job j einer Maschine mit der aktuell kleinsten Last zu.
- $L_i =$ Last auf Maschine i.
- $J_i =$ Jobs, die Maschine i zugewiesen wurden.

```
List-Scheduling(m,n,t_1,t_2,...,t_n):
for i ← 1 bis m
    J_i ← ∅
    L_i ← 0
for j ← 1 bis n
    i = argmin_{k=1,...,m} L_k
    J_i ← J_i ∪ {j}
    L_i ← L_i + t_j
return J_1,...,J_m
```

$\square$ *Maschine i hat geringste Last*

Laufzeit?

Lastverteilung: List-Scheduling

List-Scheduling-Algorithmus:

- Berücksichtige n Jobs mit einer fixen Ordnung.
- Greedy-Algorithmus: Teile Job j einer Maschine mit der aktuell kleinsten Last zu.
- $L_i =$ Last auf Maschine i.
- $J_i =$ Jobs, die Maschine i zugewiesen wurden.

```
List-Scheduling(m,n,t_1,t_2,...,t_n):
for i ← 1 bis m
    J_i ← ∅
    L_i ← 0
for j ← 1 bis n
    i = argmin_{k=1,...,m} L_k
    J_i ← J_i ∪ {j}
    L_i ← L_i + t_j
return J_1, ..., J_m
```

□ *Maschine i hat geringste Last*

Laufzeit: $O(n \log m)$ mit einer Priority-Queue.

Lastverteilung: Analyse von List-Scheduling

Theorem: [Graham, 1966] List-Scheduling ist ein 2-Approximationsalgorithmus.

- Erste Worst-Case-Analyse eines Approximationsalgorithmus.
- Dazu muss man die resultierende Lösung mit der optimalen Bearbeitungsdauer L^* vergleichen.

Lemma 1: Für die optimale Dauer gilt in jedem Fall $L^* \geq \max_j t_j$.

Beweis: Eine Maschine muss den aufwendigsten Job verarbeiten. $\square$

Lemma 2: Für die optimale Dauer gilt in jedem Fall $L^* \geq \frac{1}{m} \sum_j t_j$.

Beweis:

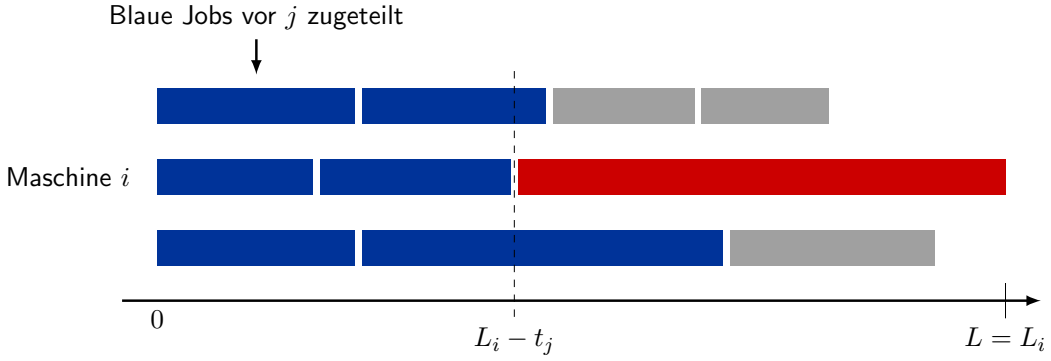
- Die gesamte Verarbeitungszeit ist $\sum_j t_j$.
- Eine der m Maschinen muss zumindest den $1/m$-ten Teil der Arbeit machen. $\square$

Lastverteilung: Analyse von List-Scheduling

Theorem: List-Scheduling ist ein 2-Approximationsalgorithmus.

Beweis: Wir betrachten die Last L_i einer Maschine i, die einen Flaschenhals darstellt.

- Sei j der letzte zugeteilte Job auf Maschine i.
- Wenn Job j Maschine i zugewiesen wird, hat i die geringste Last.
  Die Last vor der Zuteilung ist $L_i - t_j \Rightarrow L_i - t_j \leq L_k$ für alle $1 \leq k \leq m$.



Lastverteilung: Analyse von List-Scheduling

Theorem: List-Scheduling ist ein 2-Approximationsalgorithmus.

Beweis: Wir betrachten die Last L_i einer Maschine i, die einen Flaschenhals darstellt.

- Sei j der letzte zugeteilte Job auf Maschine i.
- Wenn Job j Maschine i zugewiesen wird, hat i die geringste Last.
  Die Last vor der Zuteilung ist $L_i - t_j \Rightarrow L_i - t_j \leq L_k$ für alle $1 \leq k \leq m$.
- Wir summieren alle Ungleichungen über alle k und dividieren durch m:

$$\begin{aligned} L_i - t_j &\leq \frac{1}{m} \sum_k L_k \\ &= \frac{1}{m} \sum_j t_j \\ &\leq L^* \end{aligned}$$

- Nun ist $L_i = \underbrace{(L_i - t_j)}_{\leq L^*} + \underbrace{t_j}_{\leq L^*} \leq 2L^*$. □

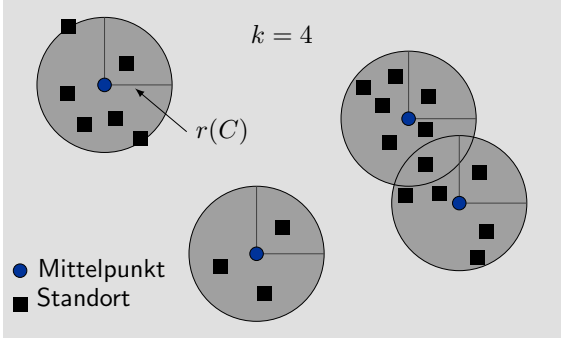
Lemma 2 Lemma 1

Center Selection

Center Selection

Eingabe: Menge von n Standorten $s_1, \ldots, s_n$ und ganze Zahl $k > 0$.

Center-Selection-Problem: Wähle k Mittelpunkte C, so dass die maximale Distanz von einem Standort zu einem nächsten Mittelpunkt minimiert wird.



Center Selection

Eingabe: Menge von n Standorten $s_1, \ldots, s_n$ und ganze Zahl $k > 0$.

Center-Selection-Problem: Wähle k Mittelpunkte C, so dass die maximale Distanz von einem Standort zu einem nächsten Mittelpunkt minimiert wird.

Notation:

- $\text{dist}(x, y) =$ Distanz zwischen x und y.
- $\text{dist}(s_i, C) = \min_{c \in C} \text{dist}(s_i, c) =$ Distanz von s_i zu einem nächsten Mittelpunkt.
- $r(C) = \max_i \text{dist}(s_i, C) =$ kleinster überdeckender Radius.

Ziel: Finde eine Menge von Mittelpunkten C, die $r(C)$ minimiert, unter Berücksichtigung von $|C| = k$.

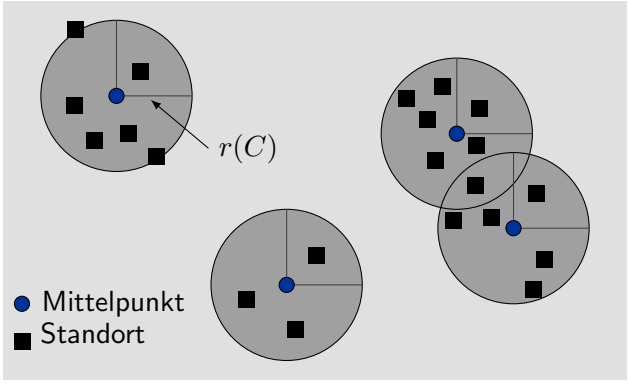
Eigenschaften der Distanzfunktion:

- $\text{dist}(x, x) = 0$ (Identität)
- $\text{dist}(x, y) = \text{dist}(y, x)$ (Symmetrie)
- $\text{dist}(x, y) \leq \text{dist}(x, z) + \text{dist}(z, y)$ (Dreiecksungleichung)

Center Selection: Beispiel

Beispiel: Jeder Standort ist ein Punkt in der Ebene, ein Mittelpunkt kann jeder Punkt in der Ebene sein, dist(x, y) = Euklidische Distanz.

Anmerkung: Es gibt unendlich viele potentielle Lösungen!



Greedy-Algorithmus: Falscher Ansatz

Greedy-Algorithmus: Wähle für den ersten Mittelpunkt einen besten Platz für einen Mittelpunkt. Danach füge iterativ Mittelpunkte so hinzu, dass der Abdeckradius immer am meisten reduziert wird.

Anmerkung: Kann beliebig schlecht sein!

Center Selection: Greedy-Algorithmus

Greedy-Algorithmus: Wähle den ersten Mittelpunkt beliebig aus der Menge aller Standorte. Wähle wiederholt als nächsten Mittelpunkt einen Standort, der am weitesten von jedem existierenden Mittelpunkt entfernt ist.

```
Greedy-Center-Selection(k,n,s_1,s_2,...,s_n)
C = {s_1}
wiederhole k − 1 mal
    Wähle einen Standort s_i mit maximaler dist(s_i,C)
    Füge s_i zu C hinzu
return C
```

- *Standort am weitesten von jedem Mittelpunkt.*

Beobachtung: Nach der Terminierung sind alle Mittelpunkte in C paarweise zumindest $r(C)$ voneinander entfernt.

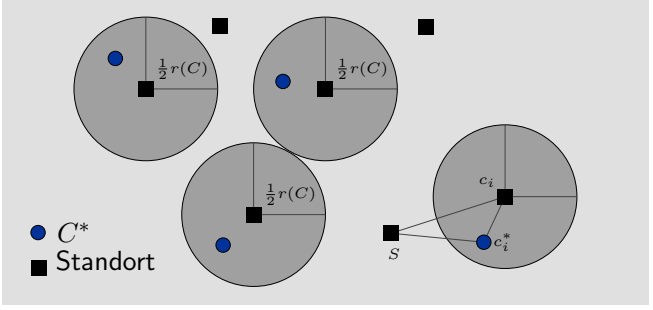
Beweis: Durch Konstruktion des Algorithmus.

Greedy-Algorithmus: Analyse

Theorem: Sei C^* eine optimale Menge von Mittelpunkten. Dann ist $r(C) \leq 2r(C^*)$.
Beweis: (durch Widerspruch) Angenommen $r(C^*) < \frac{1}{2}r(C)$.

- Für jeden Standort c_i in C betrachte den Radius $\frac{1}{2}r(C)$ um ihn herum.
- Nur ein $c_i^* \in C^*$ in jedem Radius; sei c_i der Standort mit c_i^*.
- Betrachte einen beliebigen Standort s und seinen nächsten Mittelpunkt c_i^* in C^*.
- $\text{dist}(s, C) \leq \text{dist}(s, c_i) \leq \text{dist}(s, c_i^*) + \text{dist}(c_i^*, c_i) \leq 2r(C^*)$.
- Daher $r(C) \leq 2r(C^*)$. $\square$

Δ-*Ungleichung* $\leq r(C^*)$ *da* c_i^* *der nächste Mittelpunkt ist.*



Center Selection

Theorem: Sei C^* eine optimale Menge von Mittelpunkten. Dann gilt $r(C) \leq 2r(C^*)$.

Theorem: Greedy-Algorithmus ist ein 2-Approximationsalgorithmus für das Center-Selection-Problem.

Anmerkung: Greedy-Algorithmus platziert Mittelpunkte immer auf Standorten aber erreicht trotzdem eine Gütegarantie von 2 gegenüber einer optimalen Lösung, bei der Mittelpunkte überall platziert werden dürfen.

- *z.B., Punkte in einer Ebene*

Frage: Gibt es Hoffnung auf einen 3/2-Approximationsalgorithmus? 4/3? Eher nein!

Theorem: Es existiert ein ε-Approximationsalgorithmus für das Center-Selection-Problem für ein beliebiges $\varepsilon < 2$ nur dann wenn P = NP gilt.

Optimierung - Heuristische Verfahren

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 22. Juni 2023

Vorlesungsfolien

Optimierung: Roadmap

Branch-and-Bound

Dynamische Programmierung

Approximation(salgorithmen)

Heuristische Verfahren: Erzeuge in praktisch akzeptabler Zeit eine Näherungslösung. In der Praxis kann eine solche Lösung häufig sehr gut oder sogar optimal sein, es gibt aber keine Garantie dafür.

Heuristische Verfahren

Verfahren:

- Konstruktionsverfahren
- Verbesserungsheuristiken – Lokale Suchverfahren
- Metaheuristiken
  - Simulated Annealing
  - Tabu Suche
  - Evolutionäre Algorithmen

Konstruktionsverfahren

Eigenschaften:

- Meist sehr problemspezifisch.
- Häufig intuitiv gestaltet.
- Basiert oft auf dem *Greedy-Prinzip*.

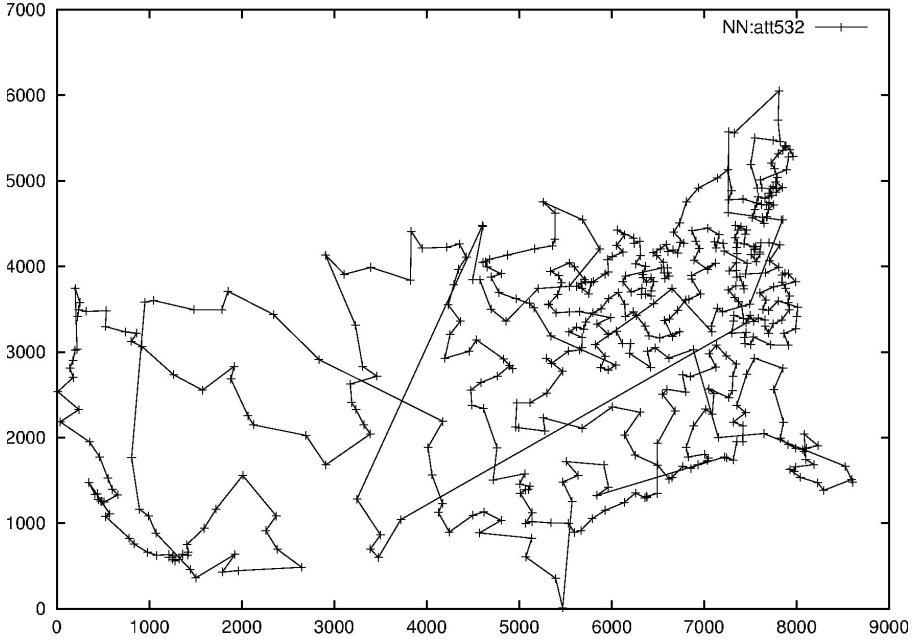
```
Greedy-construction-heuristic:
x ← leere Lösung
while x ist keine vollständige Lösung
    e ← die aktuell nützlichste Erweiterung von x
    x ← x ⊕ e
```

Beispiele für Konstruktionsverfahren

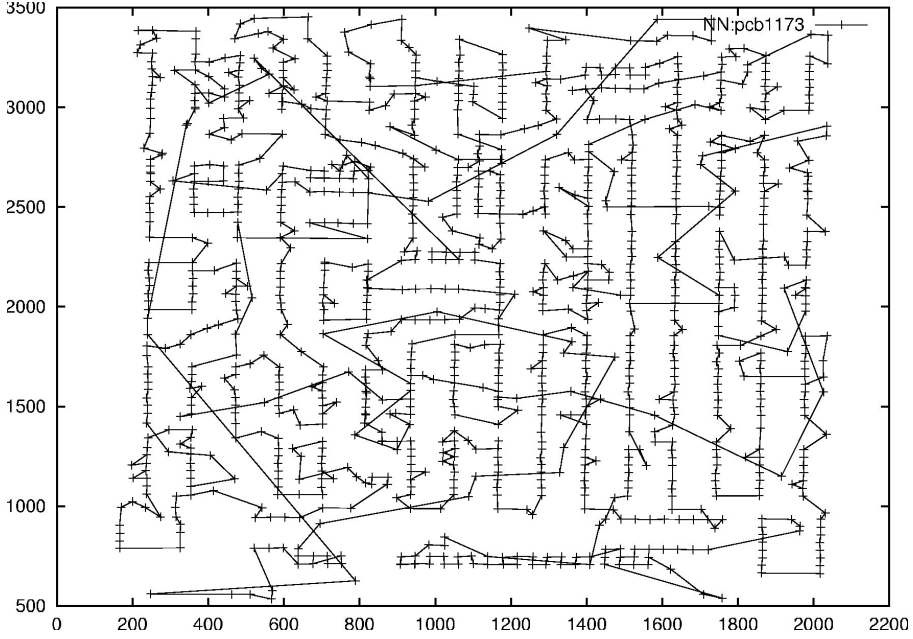
Wir haben in der VO bereits einige Konstruktionsverfahren kennengelernt.

- Minimaler Spannbaum: Greedy Algorithmen von Prim und Kruskal berechnen globales Optimum und sind daher exakte Algorithmen
- 0/1-Rucksackproblem: First-Fit-Heuristik berücksichtigt alle Gegenstände in nicht absteigendem Verhältnis von Wert zu Gewicht und packt jeden passenden Gegenstand ein.
- Vertex Cover Problem
- Lastverteilung: List Scheduling Algorithmus (Gütegarantie 2)
- TSP:
  - Spanning-Tree-Heuristik (Gütegarantie 2 für metrisches TSP)
  - Nearest-Neighbor-Heuristik: Starte bei einem Knoten und gehe immer zum nächsten nicht besuchten Nachbarn.

Beispiel: Nearest-Neighbor-Heuristik für das TSP (1)



Beispiel für Nearest-Neighbor-Heuristik (2)



Weiteres Beispiel: Insertion-Heuristiken für das TSP

Vorgehen:

- Starte mit einer Tour von 3 Knoten oder im Fall von Euklidischen Instanzen mit der konvexen Hülle.
- Füge die restlichen Knoten schrittweise zur Tour hinzu, bis alle Knoten eingebunden sind.

Unterschiedliche Strategien für die Auswahl des nächsten einzufügenden Knoten, keine ist aber immer am besten, z.B.:

- Nearest Insertion: Knoten, der zu einem in der Tour am nächsten
- Cheapest Insertion: Knoten, dessen Einfügen die Tourlänge am wenigsten erhöht
- Farthest Insertion: Knoten, der zu einem bereits in der Tour am weitesten entfernt
- Random Insertion: wähle Knoten zufällig.

Strategien für das Einfügen des neuen Knoten:

- Nach dem nächsten Knoten in der Tour.
- Eine minimale Zunahme der Tourlänge verursachend.

Praktische Ergebnisse oft 10–20% über dem Optimum, aber auch für das metrische TSP keine konstante Gütegarantie!

Lokale Suche

Verbesserungsheuristiken: Lokale Suche

Konstruktionsheuristiken sind oft intuitiv und schnell, liefern aber häufig keine ausreichend guten Lösungen.

Idee: Versuche eine Ausgangslösung durch kleine Änderungen iterativ zu verbessern
$\rightarrow$ Lokale Suche

- i.A. keine Gütegarantien
- in der Praxis aber oft deutliche Verbesserung von Lösungen
- für die meisten Anwendungen akzeptable Laufzeiten

Lokale Suche

Prinzip:

```
x ← Ausgangslösung
while Abbruchkriterium nicht erfüllt
    Wähle x' ∈ N(x)
    if x' besser als x
        x ← x';
```

- $N(x)$: *Nachbarschaft von* x

Design einer Lokalen Suche

Komponenten, die für eine lokale Suchen wichtig sind:

- Lösungsrepräsentation
- Nachbarschaftsstruktur: Welche Lösungen werden von einer aktuellen ausgehend unmittelbar in Erwägung gezogen?
- Schrittfunktion: Wie wird die Nachbarschaft durchsucht?
- Terminierungskriterium

Nachbarschaftsstruktur

Nachbarschaftsstruktur: Eine Nachbarschaftsstruktur ist eine Funktion $N : S \to 2^S$, die jeder gültigen Lösung $x \in S$ eine Menge von Nachbarn $N(x) \subseteq S$ zuweist.

Nachbarschaft: $N(x)$ wird auch Nachbarschaft von x genannt.

Eigenschaften:

- Die Nachbarschaft ist üblicherweise implizit durch mögliche Veränderungen (Züge, Moves) definiert.
- Darstellung als Nachbarschaftsgraph möglich: Knoten entsprechen den Lösungen, eine Kante (u, v) existiert wenn $v \in N(u)$.
- Es gilt Größe der Nachbarschaft vs. Suchaufwand abzuwägen.

Lokale Suche: Vertex Cover

VERTEX COVER: Gegeben sei ein Graph $G = (V, E)$. Finde eine minimale Teilmenge $C \subseteq V$ von Knoten, sodass für jede Kante $(u, v) \in E$ entweder $u \in C$ oder $v \in C$ (oder beide).

Nachbarschaftsstruktur: $C' \in N(C)$ wenn C' aus C durch Löschen eines einzigen Knotens erzeugt werden kann und noch immer ein Vertex Cover ist. $N(C) = O(|V|)$.

Lokale Suche: Starte mit Vertex Cover C, z.B. $C = V$.
Wenn es ein $C' \in N(C)$ gibt, das ein gültiges Vertex Cover ist, so ist dieses immer eine bessere Lösung, da $|C'| = |C| - 1$.
Ersetze C durch ein solches C'.

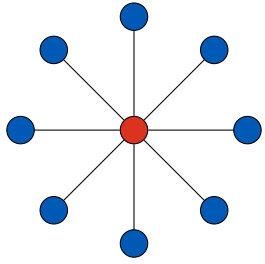
Diese Lokale Suche terminiert nach $O(|V|)$ Schritten.

Lokale Suche: Vertex Cover

Die lokale Suche liefert nicht immer eine optimale Lösung:

Lokales Optimum:
Kein Nachbar ist strikt besser und die lokale Suche kann die aktuelle Lösung daher nicht weiter verbessern.

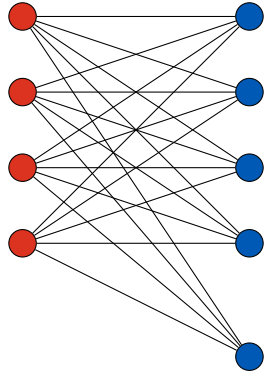
Beispiel 1:



(Globales) Optimum = roter Knoten in der Mitte
Lokales Optimum = alle blauen Knoten

Lokale Suche: Vertex Cover

Beispiel 2:



(Globales) Optimum = alle roten Knoten
Lokales Optimum = alle blauen Knoten

Lokale Suche: Vertex Cover

Beispiel 3:

Optimum: Alle geraden Knoten

Lokales Optimum: Jeder dritte Knoten wird ausgelassen.

Lokale vs. globale Optima

Für die Maximierung einer Zielfunktion $f(x)$ gilt:

- Ein lokales Maximum in Bezug auf eine Nachbarschaftsstruktur N ist eine Lösung x für die gilt: $f(x) \geq f(x')$ für alle $x' \in N(x)$.
- Nachbarschaftsstruktur bestimmt welche Lösungen lokal optimal sind.

Verbesserungsmöglichkeiten:

- Verwendung anderer/größerer Nachbarschaften.
- Iterierte Lokale Suche: Wende Lokale Suche wiederholt auf unterschiedliche Startlösungen an.
- Kombination unterschiedlicher lokaler Suchmethoden.

Lokale Suche: Vertex Cover

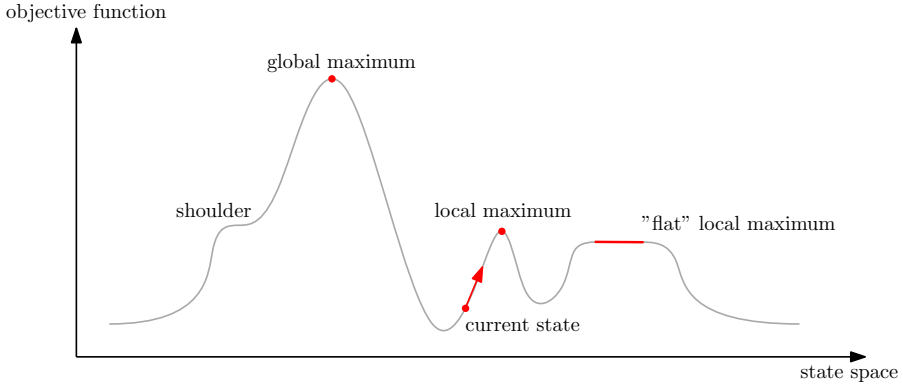
Alternative Nachbarschaft z.B.

- $N'(C)$ beinhaltet auch alle Knotenmengen, die durch Hinzufügen eines Knotens von $V \setminus C$ und Entfernen von zwei Knoten aus C gebildet werden können.
- $O(N'(C)) = O(|V|^3) \rightarrow$ Lokale Suche wird aufwändiger

Lokale vs. globale Optima

Generelles Problem der Lokalen Suche: Es wird nur ein nächstes lokales Optimum gefunden, wir sind aber i.A. an einem globalen Optimum interessiert.

Abstrahierte, eindimensionale Vorstellung des Suchraums:



Lokale Suche: SAT

Gegeben: Eine boolesche Formel in konjunktiver Normalform mit Variablen $x_1, \ldots, x_n$, z.B.

$$(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$$

Gesucht: Variablenzuweisung, sodass alle Klauseln erfüllt werden.

Optimierungsvariante: MAX-SAT - Maximiere die Anzahl der erfüllten Klauseln.

Hinweise:

- Repräsentation von Lösungen mit einem binären Vektor $x = \{0, 1\}^n$.
- NP-vollständig.

k-flip Nachbarschaft für binäre Vektoren

- Nachbarlösungen haben eine Hamming-Distanz $\leq k$, d.h., sie unterscheiden sich in bis zu k Bits.
- Größe der Nachbarschaft: $O(n^k)$

Schrittfunktion – Wahl von $x' \in N(x)$ in der Lokalen Suche

Auswahlmöglichkeiten:

- Best Improvement: Durchsuche $N(x)$ vollständig und nimm eine beste Nachbarlösung.
- First Improvement: Durchsuche $N(x)$ in einer bestimmten Reihenfolge, nimm erste Lösung, die besser als x ist.
- Random Neighbor: Wähle eine zufällige Lösung aus $N(x)$.

Hinweise:

- Wahl kann starken Einfluss auf Performance haben.
- Allgemein ist kein Verfahren immer besser als ein anderes.
- Beispielsweise ist ein Durchlauf von Random Neighbor meist schneller, dafür benötigt Best Improvement oft erheblich weniger Iterationen.

Abbruchkriterium

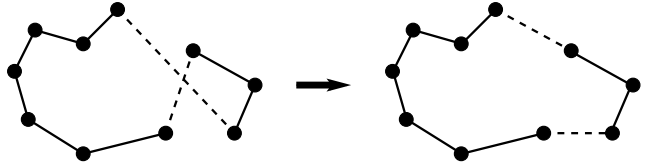
- Meist wird die Lokale Suche beendet, wenn ein lokales Optimum erreicht wurde.
- Bei einer Random Neighbor Schrittfunktion kann ein solches jedoch nicht direkt erkannt werden.
- Manchmal ist eine vollständige lokale Suche auch zu zeitaufwändig.

Alternativen:
Abbruch erfolgt

- nach einer bestimmten Iterationsanzahl oder Zeit.
- wenn eine ausreichend gute Lösung gefunden wurde.
- wenn keine weitere Verbesserung über eine bestimmte Anzahl letzter Iterationen erreicht wurde.

Lokale Suche für das symmetrisches TSP

Nachbarschaftsstruktur: Austausch zweier Kanten
(„2-exchange“ oder „2-opt“)



Größe der Nachbarschaft: $O(n^2)$

Inkrementelle Evaluierung:

- Berechne den Wert einer Nachbarlösung effizient aus dem Wert der aktuellen Lösung unter Berücksichtigung der wegfallenden und hinzukommenden Kanten.
- Benötigt hier nur konstante Zeit im Vergleich zu $O(n)$ Zeit für eine vollständige unabhängige Berechnung des Zielfunktionswerts.

2-opt Lokale Suche für das symmetrische TSP

mit First-Improvement Schrittfunktion

Vorgehen:

(1) Sei $E(T) = \{(i_1, i_2), (i_2, i_3), \ldots, (i_n, i_1)\}$ die Menge der Kanten der aktuellen Tour T und sei $i_{n+1} = i_1$.

(2) Sei $Z = \{\{(i_p, i_{p+1}), (i_q, i_{q+1})\} \subset T \mid 1 \leq p, q \leq n \wedge p + 1 < q\}$
sei die Menge aller Paare nicht nebeneinanderliegender Kanten

(3) Für alle Kantenpaare $\{(i_p, i_{p+1}), (i_q, i_{q+1})\}$ aus Z:
Falls $c_{i_p i_{p+1}} + c_{i_q i_{q+1}} > c_{i_p i_q} + c_{i_{p+1}, i_{q+1}}$:
- $T = T \backslash \{(i_p, i_{p+1}), (i_q, i_{q+1})\} \cup \{(i_p, i_q), (i_{p+1}, i_{q+1})\}$
- gehe zu (2)

(4) retourniere T.

2-opt Lokale Suche für das symmetrische TSP

Laufzeitkomplexität: Da $|N(x)| = O(n^2)$ und jede Nachbarlösung in konstanter Zeit inkrementell evaluiert werden kann, benötigt eine Iteration $O(n^2)$ Zeit.

Frage: Wieviele Iterationen sind notwendig bis ein lokales Optimum erreicht ist?

Worst-Case: Bis zu $O(n!)$ Iterationen (ohne Beweis)!
Die Worst-Case-Laufzeit dieser Lokalen Suche ist daher exponentiell!

Praxis: Dennoch ist das Verfahren auch auf großen Instanzen in meist sehr schnell. Starten wir mit einer sinnvollen Ausgangslösung sind in der Regel nur wenige Iterationen erforderlich, um ein lokales Optimum zu erreichen.

r-opt Nachbarschaft für das Symmetrische TSP

Verallgemeinerung: Die Idee der 2-opt Nachbarschaft kann verallgemeinert werden. Es werden $r \geq 2$ Kanten durch neue ersetzt.

Prinzip der r-opt Lokalen Suche:

(1) Wähle eine beliebige Anfangstour $T = \{(i_1, i_2), (i_2, i_3), \ldots, (i_n, i_1)\}$.

(2) Sei Z die Menge aller r-elementigen Teilmengen von T.

(3) Für alle $R \in Z$: Setze $S = T \setminus R$ und konstruiere alle Touren, die S enthalten. Ist ein S besser als T, setze $T = S$ und gehe zu (2).

(4) T ist das Ergebnis.

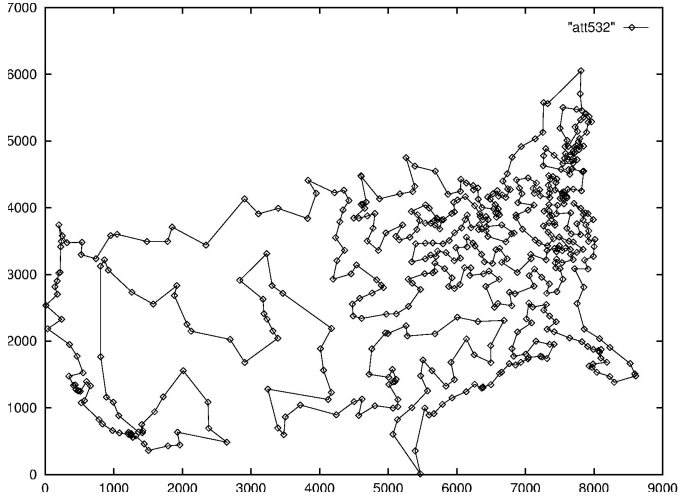
Laufzeitkomplexität der r-opt Lokalen Suche

Größe einer r-opt Nachbarschaft:

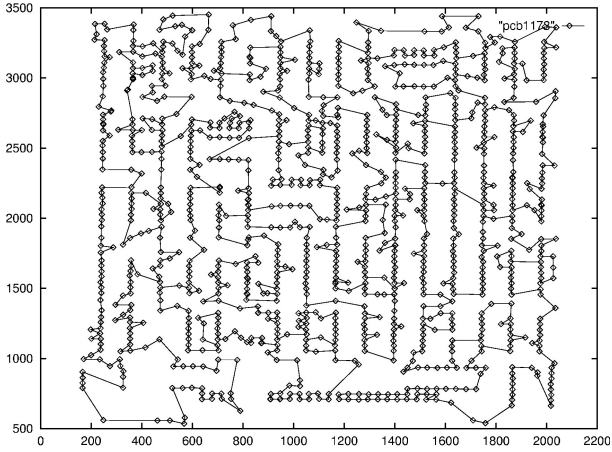
- Es gibt $\binom{n}{r} = O(n^r)$ Möglichkeiten r unterschiedliche Kanten aus einer aktuellen Tour zu entfernen.
- Das Entfernen führt zu r nicht zusammenhängenden Pfaden.
- Diese können auf $O(r!)$ Möglichkeiten zu neuen Touren zusammengefügt werden.
- $|N(x)| = O(n^r \cdot r!) = O(n^r)$

Hinweis: Wie bereits bei 2-opt ist auch hier im allgemeinen Fall die Worst-Case-Anzahl der möglichen Iterationen nicht polynomiell beschränkt.

2-opt Lösung für TSP (1)



2-opt Lösung für TSP (2)



Hinweis: Die Lösung wurde durch eine Lokale Suche mit Random-Neighbor Schrittfunktion gefunden. Es gibt 2 Kanten, die sich kreuzen, daher ist diese Lösung kein lokales Optimum.

Zusammenfassung zur Lokalen Suche für das TSP

Anwendung:

- 2-opt wird sehr häufig eingesetzt, kommt meist auf ca. 6–8% an die optimale Lösung heran.
- 3-opt manchmal verwendet (deutlich zeitaufwändiger), kommt meist auf 3–4% an die optimale Lösung heran.
- 4-opt ist in der Praxis i.A. bereits zu zeitaufwändig.

Hinweis: Die Prozentangaben beziehen sich auf bestimmte Instanzen, die zum Testen der Algorithmen verwendet werden und sollen hier nur einen groben Richtwert vermitteln.

Zusammenfassung zur Lokalen Suche für das TSP

Weitere Nachbarschaftsstrukturen:

- Verschieben eines Knotens an eine andere Position.
  - Für das asymmetrische TSP häufig besser geeignet
- Verschieben einer Teilsequenz an eine andere Position.
- Lin-Kernighan Heuristik (1973):
  - Eine der führenden, schnellen Heuristiken für große TSPs.
  - Kommt meist auf 1–2% an das Optimum heran.
  - Variable Tiefensuche: Anzahl der ausgetauschten Kanten nicht grundsätzlich beschränkt, es werden jedoch nur „vielversprechende“ Kantenaustausche durchprobiert.

Maximaler Schnitt (*Maximal Cut*)

Maximaler Schnitt (MAX-CUT)

MAX-CUT: Gegeben sei ein ungerichteter Graph $G = (V, E)$ mit positiven ganzzahligen Kantengewichten w_{uv} für alle Kanten $(u, v) \in E$. Finde eine Partition der Knoten (A, B), sodass das Gesamtgewicht von Kanten, die Knoten in den unterschiedlichen Partitionen verbinden, maximiert wird.

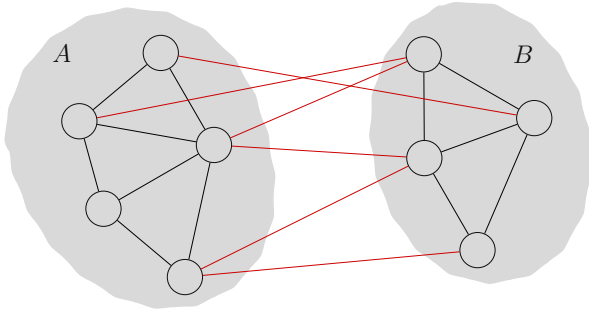
$$w(A, B) := \sum_{u \in A,\ v \in B} w_{uv}$$

Beispielanwendung:

- n Aktivitäten, m Personen.
- Jede Person möchte an zwei Aktivitäten teilnehmen.
- Plane die Aktivitäten am Morgen und am Nachmittag so, dass eine maximale Anzahl an Personen daran teilnehmen kann.

Hinweis: MAX-CUT ist NP-vollständig.

Maximaler Schnitt



1-Flip-Nachbarschaft: Gegeben sei eine Partition (A, B). Verschiebe einen Knoten von A nach B oder von B nach A.

Algorithmus: Ausgehend von einer gültigen Initiallösung ist auf die 1-Flip-Nachbarschaft aufbauend unmittelbar eine einfache Lokale Suche möglich.

Maximaler Schnitt: Analyse der lokalen Suche

Wir können zeigen: Wird die lokale Suche ausgeführt bis eine lokal optimale Lösung erreicht wurde, so gilt eine Approximationsgüte von $1/2$.

Theorem: Sei (A, B) eine lokal optimale Partition und sei (A^*, B^*) eine optimale Partition.
Dann ist $w(A, B) \geq \frac{1}{2} \sum_{e \in E} w_e \geq \frac{1}{2} w(A^*, B^*)$.

- *Gewichte sind nicht negativ*

Maximaler Schnitt: Analyse der lokalen Suche

Beweis:

- Lokale Optimalität bedeutet, dass für alle

$$u \in A : \sum_{v \in A} w_{uv} \leq \sum_{v \in B} w_{uv}$$

  Das Aufsummieren aller Ungleichungen ergibt:

$$2 \sum_{\{u,v\} \subseteq A} w_{uv} \leq \sum_{u \in A, v \in B} w_{uv} = w(A,B)$$

- Ähnlich ist $2 \sum_{\{u,v\} \subseteq B} w_{uv} \leq \sum_{u \in A, v \in B} w_{uv} = w(A,B)$
- Nun gilt,

$$\sum_{e \in E} w_e = \underbrace{\sum_{\{u,v\} \subseteq A} w_{uv}}_{\leq \frac{1}{2} w(A,B)} + \underbrace{\sum_{u \in A, v \in B} w_{uv}}_{w(A,B)} + \underbrace{\sum_{\{u,v\} \subseteq B} w_{uv}}_{\leq \frac{1}{2} w(A,B)} \leq 2w(A,B) \; \square$$

  $\square$ *Jede Kante wird einmal gezählt.*

Metaheuristiken

Metaheuristiken

Metaheuristiken: Sind problemunabhängig formulierte Algorithmen zur heuristischen Lösung schwieriger Optimierungsaufgaben.

Anpassung: Teile dieser Algorithmen müssen an das jeweilige Problem angepasst werden, wie beispielsweise die Nachbarschaftsstruktur auch in der Lokalen Suche.

Wir betrachten hier folgende Metaheuristiken:

- Simulated Annealing
- Tabu-Suche
- Evolutionäre Algorithmen

Simulated Annealing (SA)

S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, *Science* 220(4598), pp. 671–680, 1983.

Inspiriert durch den physikalischen Prozess der langsamen Abkühlung von Materialien zur Erreichung einer stabilen Kristallstruktur, z.B. nach dem Glühen eines Metalls.

Grundlegende Idee: Auch schlechtere Nachbarlösungen werden mit einer bestimmten Wahrscheinlichkeit akzeptiert.

Schrittfunktion: I.A. Random Neighbor

Simulated Annealing

Variablen:

- Z: (Pseudo-)Zufallszahl $\in [0, 1)$
- T: „Temperatur“

```
Simulated-Annealing():
t ← 0
T ← T_init
x ← Ausgangslösung
while Abbruchkriterium nicht erfüllt
    Wähle x' ∈ N(x) zufällig
    if x' besser als x
        x ← x'
    elseif Z < e^{-|f(x')-f(x)|/T}
        x ← x'
    T ← g(T, t)
    t ← t + 1
```

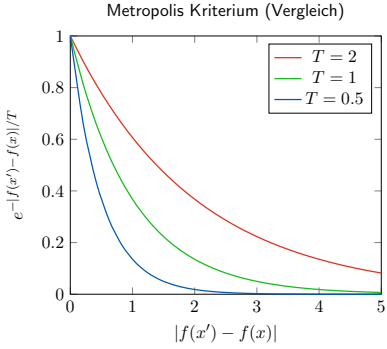
- *Metropolis-Kriterium*

Metropolis Kriterium

Metropolis Kriterium:

$$Z < e^{-|f(x')-f(x)|/T}$$

ist die Akzeptanzbedingung für schlechtere Lösungen.



Eigenschaften:

- Nur geringfügig schlechtere Lösungen werden mit höherer Wahrscheinlichkeit akzeptiert als viel schlechtere.
- Anfangs, bei hoher Temperatur T, werden schlechtere Lösungen mit größerer Wahrscheinlichkeit akzeptiert als im späteren Verlauf bei niedrigerer Temperatur.

Abkühlungsplan

Geometrisches Abkühlen:

- Faustregel für T_{init}: $f_{\max} - f_{\min}$, wobei $f_{\max}$ bzw. $f_{\min}$ eine obere bzw. untere Schranke oder Schätzung für den maximalen/minimalen Zielfunktionswert sind.
- $g(T, t) = T \cdot \alpha, \quad \alpha < 1$ (z.B. 0,999)
- Häufig wird die Temperatur auch über einige (z.B. $|N(x)|$) Iterationen gleich belassen und dann jeweils etwas stärker reduziert.

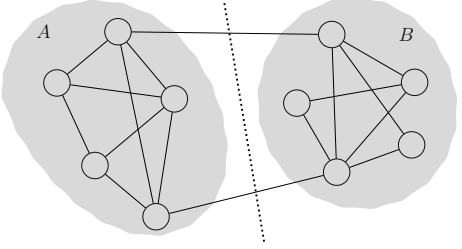
Adaptives Abkühlen: Es wird der Anteil der Verbesserungen an den letzten erzeugten Lösungen gemessen und auf Grund dessen T stärker oder schwächer reduziert.

Beispiel: SA für Graph-Bipartitionierung

Eine der ersten Anwendungen von SA

Definition für Graph-Bipartitionierung:

- Gegeben: Graph $G = (V, E)$
- Gesucht: Partitionierung von G in zwei Knotenmengen A, B, mit $|A| = |B|$, $A \cap B = \emptyset$, und $A \cup B = V$
- Minimiere $|\{(u, v) \in E \mid u \in A, v \in B\}|$



Beispiel: SA für Graph-Bipartitionierung

Repräsentation der Lösung: Charakteristischer Vektor

- $x = (x_1, \ldots, x_n)$, $n = |V|$
- Knoten i wird der Menge A zugewiesen, wenn $x_i = 0$ und B sonst.

Simulated Annealing:

- Nachbarschaft: Tausche jeweils einen Knoten von A und B aus.
- Zufällige Anfangslösung
- Geometrisches Abkühlen, $\alpha = 0.95$
- Iterationen auf jeder Temperaturstufe: $n \cdot (n - 1)$
- Eine der erste Anwendungen von SA

Beispiel: SA für Graph-Bipartitionierung

Verbesserung:

- Einschränkung der Nachbarschaft und Erlauben ungültiger Lösungen.
- Flip-Nachbarschaft (vgl. $\mathrm{MAX\text{-}CUT}$)
  - Verschiebe einen einzelnen Knoten in andere Menge.
  - $|N(x)| = n$ anstatt $n^2/4$
- Modifizierte Zielfunktion:
  $f(A, B) = |\{(u, v) \in E \mid u \in A, v \in B\}| + \gamma(|A| - |B|)^2$
  γ: Faktor für das Ungleichgewicht

Fazit zu Simulated Annealing

- Typischerweise einfach zu implementieren,
  Parametertuning notwendig aber meist nicht so schwer.
- Für viele Probleme gute Resultate, aber häufig können ausgefeiltere Methoden noch bessere Ergebnisse liefern.
- Viele Varianten/Erweiterungen:
  - Dynamische Strategien für das Abkühlen (Wiedererwärmen)
  - Parallelisierung
  - Kombination mit anderen Methoden

Tabu-Suche (TS)

- Basiert auf einem Gedächtnis (History) über den bisherigen Optimierungsverlauf und nutzt dieses um über lokale Optima hinweg zu kommen.
- Vermeidung von Zyklen durch Verbieten des Wiederbesuchens früherer Lösungen.
- I.A. Best Improvement Schrittfunktion: In jedem Schritt wird die beste erlaubte Nachbarlösung angenommen, auch wenn diese schlechter ist als die aktuelle.

Tabu-Suche

Variablen: Bisher beste gefundene Lösung x_{best}, Aktuelle Lösung x, Nachbarlösung x', Tabu-Liste TL, Menge erlaubter Nachbarlösungen X'.

```
Tabu-Suche():
x_best ← x ← Ausgangslösung
TL ← {x}
while Abbruchkriterium nicht erfüllt
    X' ← Teilmenge von N(x) unter Berücksichtigung von TL
    x' ← beste Lösung von X'
    Füge x' zu TL hinzu
    Lösche Elemente aus TL, welche älter als t_L Iterationen sind
    x ← x'
    if x besser als x_best
        x_best ← x
```

Das Gedächtnis: Tabuliste

Eigenschaften:

- Explizites Speichern von vollständigen Lösungen
  Nachteil: speicher- und zeitaufwändig.
- Meist bessere Alternative: Speichern von Tabuattributen, d.h., nur einzelnen Aspekten von besuchten Lösungen.
- Lösungen sind tabu (verboten), falls sie Tabuattribute enthalten.
- Als Tabuattribute werden meist Variablenwerte benutzt, die von durchgeführten Zügen gesetzt wurden. Die Umkehrung der Züge ist dann für t_L Iterationen verboten.
- Wichtiger Parameter: Tabulistenlänge t_L.

Parameter: Tabulistenlänge

Tabulistenlänge:

- Wahl von t_L häufig sehr kritisch!
- Zu kurze Tabulisten können zu Zyklen führen.
- Zu lange Tabulisten verbieten viele mögliche Lösungen und beschränken die Suche stark.
- Geeignete Länge i.A. problemspezifisch.
- Muss experimentell bestimmt werden, oder
  - immer zufällig neu wählen.
  - adaptiv anpassen ($\rightarrow$ *Reactive Tabu Search*).

Aspirationskriterien

Aspirationskriterien:

- Manchmal wird eine Lösung verboten (d.h. ihre Attribute sind in der Tabuliste), obwohl sie sehr gut ist.
- Aspirationskriterium: Überschreibt den Tabu-Status einer „interessanten“ Lösung, d.h. die Lösung darf gewählt werden.
- Beispiel eines oft benutzten Aspirationskriteriums:
  - Eine verbotene Lösung ist besser als die bisher beste.

Beispiel: Tabu-Suche für das Graphenfärbeproblem

Graphenfärbeproblem:

- Gegeben: Graph $G = (V, E)$.
- Gesucht: Weise jedem Knoten $v \in V$ eine Farbe $x_v \in \{1, \ldots, k\}$ zu, sodass für alle Kanten $(u, v) \in E$ gilt $x_u \neq x_v$.

Hinweis: Ist ein NP-vollständiges Problem.

Optimierungsvariante: Minimiere Anzahl der „verletzten“ Kanten.

Beispiel: Tabu-Suche für das Graphenfärbeproblem

Aspekte:

- Evaluierungskriterium: Minimiere Anzahl der „verletzten“ Kanten
- Nachbarschaft: Färbung, die sich genau in der Farbe eines Knoten unterscheidet.
- Tabuattribute: Paare (v, j) mit $v \in V$, $j \in \{1, \ldots, k\}$, d.h. bestimmte Farbzuweisungen (j) zu bestimmten Knoten (v).
- Tabukriterium: Wird Zug $(v, j) \to (v, j')$ durchgeführt, ist Attribut (v, j) für t_L Iterationen verboten.
- Aspirationskriterium: Falls Zug zu besserer Lösung als bisher gefunden führt, ignoriere Tabu-Status und akzeptiere diese Lösung.
- Einschränkung der Nachbarschaft: Betrachte nur Zuweisungen für Knoten, die in eine Kantenverletzung involviert sind.

Fazit zur Tabu-Suche

- Viele weitere unterschiedliche Strategien für
  - Gedächtnis
  - Diversifizierung der Suche
  - Intensivierung in der Nähe gefundener Elitelösungen
- Oft exzellente Ergebnisse und vergleichsweise schnell.
- Meist relativ aufwändiges Fine-Tuning notwendig.

Evolutionäre Algorithmen

Idee: Grundprinzipien der natürlichen Evolution werden auf primitive Weise nachgeahmt, um schwierige Optimierungsaufgaben zu lösen.

- Population: Es wird mit einer Menge von aktuellen Kandidatenlösungen gearbeitet.
- Selektion: Natürliche Auslese („survival of the fittest“); bessere Lösungen bleiben mit größerer Wahrscheinlichkeit erhalten und erzeugen neue Lösungen.
- Rekombination: Neue Lösungen werden durch zufallsgesteuerte Kreuzung bzw. Vererbung von in Eltern vorkommenden Lösungsmerkmalen abgeleitet.
- Mutation: Kleine zufällige Änderung bringt nicht in Eltern vorkommende Lösungsmerkmale ein und dadurch ist eine Variation von Elternlösungen möglich.

Evolutionäre Algorithmen

Prinzip eines evolutionären Algorithmus:

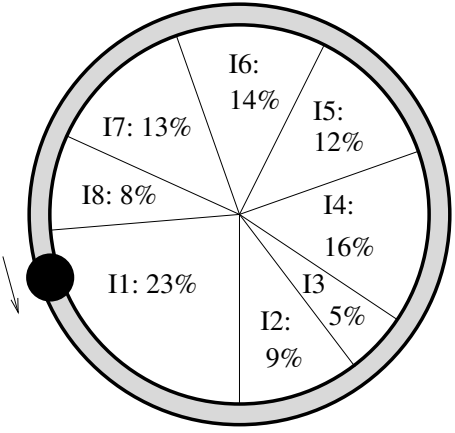
- Selektierte Eltern Q_s
- Zwischenlösungen Q_r

```
Evolutionär():
P ← Menge von Ausgangslösungen
Bewerte(P)
while Abbruchkriterium nicht erfüllt
    Q_s ← Selektion(P)
    Q_r ← Rekombination(Q_s)
    P ← Mutation(Q_r)
    Bewerte(P)
```

Fitness Proportional Selection

Roulette-Wheel Selection:

- Sei $f(x_i) > 0$ der Zielfunktionswert (Fitness) jeder Lösung $x_i \in P$.
- $P = \{x_1, \ldots, x_{|P|}\}$.
- Wir gehen von einem Maximierungsproblem aus.



Selektionswahrscheinlichkeit für Lösung x_i:

$$p_s(x_i) = \frac{f(x_i)}{\sum_{j=1}^{n} f(x_j)}$$

Selektionsdruck

Zu achten ist auf die Verhältnisse zwischen den Selektionswahrscheinlichkeiten der Lösungen in P.

Selektionsdruck: Sei $p_s^{\max} = \max\{p_s(x_1), \ldots, p_s(x_{|P|})\}$ und $\overline{p_s} = 1/|P|$. Dann ist der Selektionsdruck:

$$S = p_s^{\max} / \overline{p_s}$$

- S zu niedrig: Ineffiziente Suche ähnelt einer Zufallssuche.
- S zu hoch: Rascher Verlust der Vielfalt da einzelne Lösungen zu häufig ausgewählt werden, rasche Konvergenz zu lokalem Optimum.

Skalierung der Bewertungsfunktion

Skalierung: Um den Selektionsdruck S zu steuern, wird die Bewertungsfunktion meist skaliert, z.B. über eine lineare Funktion

$$g(x_i) = a \cdot f(x_i) + b$$

mit geeigneten Werten a und b.

Hinweis: Skalierung ist auch notwendig für

- Minimierungsprobleme
- Wenn $f(x_i) < 0$

Alternative: Tournament Selektion

Alternative:

(1) Wähle aus der Population k Lösungen gleichverteilt zufällig.

(2) Die beste der k Lösungen ist die selektierte.

Eigenschaften:

- Relative Unterschiede in der Bewertung spielen keine Rolle.
- Skalierung deshalb nicht erforderlich.
- Selektionsdruck wird über Gruppengröße k gesteuert.

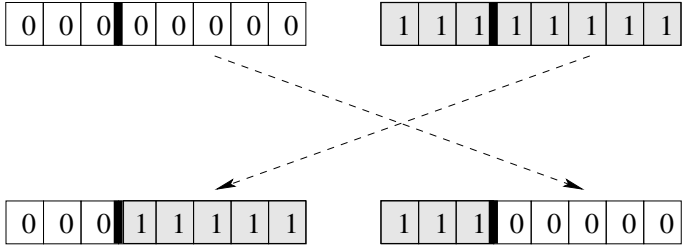
Rekombination

Rekombination: Aus zwei (oder mehreren) Elternlösungen wird eine neue Lösung abgeleitet.

Vererbung: Die neue Lösung sollte möglichst ausschließlich aus den Eigenschaften (Bestandteilen) der Eltern aufgebaut werden.

Meist zufallsbasierte, einfach gehaltene und schnelle Operation.

Beispiel für Bitstrings: 1-point crossover



Mutation

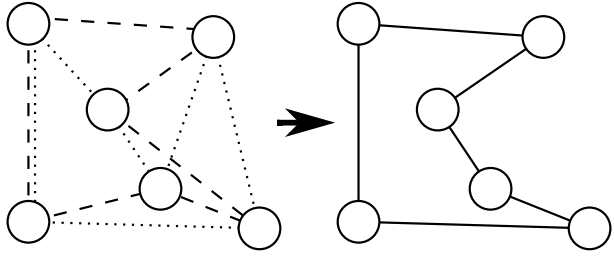
Möglichkeiten:

- Für Bitstrings z.B. Flip eines jeden Bits mit kleiner Wahrscheinlichkeit.
- Allgemein meist ein oder mehrere zufällige Moves in einer sinnvollen Nachbarschaft.

Hinweis: Vergleiche Random-Neighbor-Schrittfunktion der lokalen Suche.

Beispiel: Evolutionärer Algorithmus für TSP

Edge-Recombination: Eine neue TSP-Tour wird zufallsgesteuert möglichst nur aus Kanten aufgebaut, die bereits in zwei Elternlösungen vorkommen.



Mutation: Typischerweise ein zufälliger Move in einer klassischen Nachbarschaft wie z.B. 2-opt oder Verschieben eines Knotens an eine andere Position.

Beispiel: Edge-Recombination für das TSP

Eingabe: Zwei gültige Touren T^1 und T^2.
Ausgabe: Neue abgeleitete Tour T.
Variablen: Aktueller Knoten v, Nachfolgeknoten w, Kandidatenmenge für Nachfolgeknoten W

```
Edge-Recombination(T^1, T^2):
Beginne bei einem beliebigen Startknoten v ← v_0, T ← {}
while es existieren noch unbesuchte Knoten
    Sei W Menge noch unbesuchten Knoten, welche in
    T^1 ∪ T^2 adjazent zu v sind
    if W ≠ {}
        Wähle einen Nachfolgeknoten w ∈ W zufällig aus
    else
        Wähle einen zufälligen noch nicht
        besuchten Nachfolgeknoten w
    T ← T ∪ {(v, w)}, v ← w
Schließe die Tour: T ← T ∪ {(v, v_0)}
```

Fazit zu evolutionären Algorithmen

- Grundprinzip leicht umsetzbar.
- Häufig Kombination mit anderen Methoden sinnvoll:
  - Ausgangslösungen mit Konstruktionsheuristiken erzeugen.
  - Problemspezifisches Wissen in Rekombination und Mutation ausnutzen.
  - Neue Kandidatenlösungen mit lokaler Suche etc. versuchen zu verbessern.
- Lösungsgüte und Laufzeit hängt sehr von den konkreten Operatoren ab.
- Parallelisierung ist gut möglich.