Skriptum zur Vorlesung

Algorithmen und Datenstrukturen 2

Univ.-Prof. Dr. Günther Raidl
Univ.-Ass. Dr. Bin Hu

WINTERSEMESTER 2010/11
VERSION 2.22
INSTITUT FÜR COMPUTERGRAPHIK UND ALGORITHMEN
TECHNISCHE UNIVERSITÄT WIEN

Inhaltsverzeichnis

Vorwort

Die Vorlesung (VO) *Algorithmen und Datenstrukturen 2* ist die Fortsetzung der Lehrveranstaltung *Algorithmen und Datenstrukturen 1* und wendet sich primär an Studierende der Fachrichtungen Informatik, Wirtschaftsinformatik und Mathematik. Während in *Algorithmen und Datenstrukturen 1* die Grundlagen des Designs und der Analyse von Algorithmen sowie grundlegende Datenstrukturen behandelt wurden, untersuchen wir nun speziellere Algorithmen und Datenstrukturen in vielfältigen Anwendungsbereichen.

Natürlich können wir in der zweistündigen VO nicht alle möglichen Anwendungsbereiche für Algorithmen abdecken. Ziel der Lehrveranstaltung ist es, exemplarisch verschiedene Bereiche aufzuzeigen, die häufig in der Praxis auftreten und Ihnen verdeutlichen sollen, dass für die meisten auftretenden Probleme bereits effiziente Algorithmen oder zumindest generelle Methoden in der Literatur bekannt sind. Sie sollten also nicht jedes Mal „das Rad neu erfinden".

Die in der VO behandelten Themen zeigen auch, wie vielseitig die Algorithmentheorie ist. Vielleicht macht Ihnen ja das eine oder andere Thema, wie z.B. die Optimierung oder die geometrischen Algorithmen, besonders viel Spaß, und Sie wollen sich darin vertiefen. Dann melden Sie sich einfach bei uns und/oder besuchen Sie weiterführende Veranstaltungen zu diesen Bereichen. Beispielsweise bieten wir derzeit u.a. die LVAs *Algorithmen auf Graphen*, *Approximationsalgorithmen*, *Effiziente Algorithmen*, *Fortgeschrittene Algorithmen und Datenstrukturen*, *Heuristische Optimierungsverfahren* und *Geometrische Algorithmen* an. Natürlich können Sie auch Seminare, Praktika und Diplomarbeiten bei uns machen. Wir würden uns freuen, Sie auch nach erfolgreichem Abschluss von *Algorithmen und Datenstrukturen 2* wiederzusehen und wünschen Ihnen viel Erfolg für Ihre Zukunft.

Wien im September 2010

Günther Raidl und Bin Hu

Organisatorisches

Die Vorlesung findet im WS 2010/11 jeweils am Donnerstag von 14:15–15:45 Uhr im EI7 statt. Sie wird von Univ.-Prof. Günther Raidl und Univ.-Ass. Bin Hu vom Institut für Computergraphik und Algorithmen, Arbeitsbereich Algorithmen und Datenstrukturen, gehalten.

Das vorliegende Skriptum ist nicht dazu da, um ausschließlich daraus den Stoff zu lernen, sondern dient als begleitende Unterlage zur Vorlesung. Darüber hinausgehend empfehlen wir zur weiteren Vertiefung im nächsten Abschnitt bzw. am Ende der einzelnen Kapitel einige Bücher bzw. ausgewählte Artikel.

Ab dem WS 2006/07 ist *Algorithmen und Datenstrukturen 2* den erneuerten Studienplänen folgend eine reine Vorlesung, d.h. im Gegensatz zu früheren Semestern gibt es keinen verpflichteten Übungsteil mehr. Dennoch raten wir dringend, sich mit den Inhalten der VO tiefergehend an Hand von Übungsbeispielen auseinanderzusetzen. Hierzu finden Sie im Anhang eine Sammlung ausgewählter Aufgabenstellungen. Weiters können Sie auch auf der Homepage zur Lehrveranstaltung frühere Prüfung- bzw. Übungsangaben finden.

Bei Fragen, Problemen, Anregungen und Beschwerden wenden Sie sich bitte an uns. Wir haben für Sie dazu die email-Hotline `algodat2-ws10@ads.tuwien.ac.at` eingerichtet. Natürlich stehen wir Ihnen auch persönlich zu Verfügung. Kommen Sie aber bitte ausschließlich in den Sprechstundenzeiten oder nach vorheriger Vereinbarung. Die aktuellen Sprechstundenzeiten erfahren Sie auf unserer Homepage:

`http://www.ads.tuwien.ac.at`

Aktuelle Informationen zur LVA finden Sie auf unserer Webseite unter:

`http://www.ads.tuwien.ac.at/teaching/LVA/186170.html`

Wir wünschen Ihnen

Viel Erfolg!

Literaturliste

Die Bücher (1)-(4) beinhalten für viele der hier behandelten Themenbereiche ein einführendes Kapitel. Ein interessantes multimediales Buch ist (5), das interaktive Vorlesungen zu weiterführenden Algorithmen enthält. (6) ist ein immer noch aktueller Klassiker im Algorithmen-Bereich. Daneben finden Sie am Ende jedes Kapitels weitere Literaturhinweise, die sich speziell auf die dort behandelten Kapitel beziehen. (7) und (8) bieten eine detailierte Einführung zu lokalen Suchverfahren und Metaheuristiken. Darüber hinaus steht es Ihnen frei, jedes andere Buch, das den Stoff behandelt, auszuwählen.

(1) R. Sedgewick: „Algorithmen in Java, Teil 1–4", 3. Auflage, Pearson Education, 2003

(2) R. Sedgewick: „Algorithms in Java, Part 5", Pearson Education, 2004

(3) T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: „Introduction to Algorithms", 3. Auflage, MIT Press, Cambridge, 2009

(4) T. Ottmann und P. Widmayer: „Algorithmen und Datenstrukturen", 4. Auflage, Spektrum Akademischer Verlag, 2002

(5) T. Ottman (Hrsg.): "Prinzipien des Algorithmenentwurfs", Spektrum Akademischer Verlag, Berlin 1998

(6) A.V. Aho, J.E. Hopcroft und J.D. Ullman: "Data Structures and Algorithms", Addison-Wesley, 1982

(7) F. Glover und G. Kochenberger: „Handbook of Metaheuristics", Springer, 2003

(8) H. Hoos und T. Stützle: „Stochastic Local Search", Morgan Kaufman, 2004

Übungsbeispiele

Im Anhang dieses Skriptums befindet sich eine Sammlung ausgewählter Übungsbeispiele zur Vertiefung des Vorlesungsstoffes. Das Lösen dieser Beispiele steht Ihnen frei. Wir werden für sie im Laufe des Semesters Lösungsansätze auf unserer Webseite veröffentlichen, dennoch raten wir Ihnen dringend, diese Beispiele zunächst selbst und ohne Hilfestellung zu lösen bzw. dies zumindest zu versuchen.

Dankesworte

An dieser Stelle sei allen gedankt, die zur Entstehung dieses Skriptums beigetragen haben: Verantwortlich für den Inhalt sind Günther Raidl und Bin Hu. Besonderer Dank gilt Jakob Puchinger und Martin Schönhacker für die sorgfältige Überarbeitung, sowie Petra Mutzel, Gunnar Klau, Andreas Kerren, Raul Fechete und René Weiskircher, deren Vorlagen uns als wesentliche Basis für diese Ausgabe des Skriptums gedient haben.

Kapitel 1

Suchen in Texten

Dieser Abschnitt beschäftigt sich mit dem Suchen einer gegebenen Zeichenfolge P (Muster bzw. „pattern") in einem Text T. Dieses Problem (Pattern Matching, String Matching) tritt häufig in der Textverarbeitung auf, etwa in Form der „Suchfunktion" in Texteditoren. Auch in der Molekularbiologie wird häufig nach Mustern in einer gegebenen DNS-Sequenz gesucht. Meist ist dabei die Länge M des Musters relativ klein im Vergleich zur Länge N des zu durchsuchenden Textes.

Wir betrachten in diesem Abschnitt das folgende Problem: Gegeben sind eine Text-Zeichenfolge T der Länge N und eine Muster-Zeichenfolge P („Pattern") der Länge M, jeweils aus einem endlichen Alphabet Σ. Gesucht sind alle Vorkommen von P innerhalb von T, d. h. jeweils die Anfangs-Indizes i ($1 \leq i \leq N - M + 1$). Wir nehmen dabei an, dass der Text und das Muster in jeweils einem Feld von $T[1 \ldots N]$ bzw. $P[1 \ldots M]$ gespeichert ist. Formal ist das Problem folgendermaßen definiert.

Pattern Matching Problem
Gegeben: eine Zeichenkette (Text) $T_1 \ldots T_N$ von Zeichen aus einem endlichen Alphabet Σ und eine Zeichenkette (Muster/Pattern) $P_1 \ldots P_M$, ebenfalls mit $P_i \in \Sigma, 1 \leq i \leq M$.
Gesucht: ein oder alle Vorkommen von $P_1 \ldots P_M$ in $T_1 \ldots T_N$, d. h. jene Indizes i mit $1 \leq i \leq N - M + 1$, so dass $\forall j = 1, \ldots, M : T_{i+j-1} = P_j$.

Es gelten die folgenden Voraussetzungen:

- In der Regel gilt: $N \gg M > 1$. Dies wird im allgemeinen gegeben sein. Ein Beispiel ist das *Oxford English Dictionary* mit > 600.000 Definitionen, in dem man normalerweise nach einem Wort der Länge bis maximal 20 sucht.
- Wir nehmen ein *Dynamisches Setting* an: Text und Pattern ändern sich häufig. Denn bei gleich bleibendem Text wäre es sinnvoller, eine Index-Datenstruktur des Textes aufzubauen (z.B. *Suffix Trees*).

1.1 Naives Verfahren

Die offensichtliche Methode besteht darin, das Muster der Reihe nach (von links nach rechts) an jeden Teilstring des Textes mit Länge M anzulegen und dann zu prüfen, ob tatsächlich Übereinstimmung an der momentanen Stelle vorliegt.

Algorithmus 1 Naive-Search (T, P)

Eingabe: Text $T = T[1..N]$ und Muster $P = P[1..M]$
Ausgabe: Ausgabe aller Vorkommen von P in T

```
1:  i = 0; // Position vor der Anlegestelle von P
2:  solange i ≤ N − M {
3:      j = 1;
4:      solange (j ≤ M&&P[j] == T[i + j]) {
5:          j = j + 1;
6:      }
7:      falls j == M + 1 dann {
8:          Ausgabe: „P an Stelle i + 1 im Text gefunden";
9:      }
10:     i = i + 1;
11: }
```

Das Programm verwendet einen Textzeiger i, der jeweils auf den Index vor der Anlegestelle des Musters zeigt. Stimmen die Zeichen an der Stelle $i + j$ überein („match"), wird j hochgesetzt und das nächste Zeichen wird verglichen. Andernfalls („mismatch") wird das Muster um eine Stelle weiter rechts angesetzt. Wurde j M-Mal hintereinander hochgezählt, wurde das Muster gefunden.

Analyse der Laufzeit: Das Muster wird genau $N - M + 1$ Mal an den Text angelegt. Im schlimmsten Fall müssen bei jedem solchen Anlegen $\Theta(M)$ Vergleiche durchgeführt werden (was in der Praxis nur selten vorkommt). Insgesamt ergibt dies eine Worst-Case Laufzeit von $O(NM)$. Das Problem des naiven Algorithmus ist, dass die während der Vergleiche gewonnene Information nicht benutzt, sondern vergessen wird. Man nennt dies auch einen Algorithmus „ohne Gedächtnis".

Die folgenden beiden Abschnitte behandeln Algorithmen mit Gedächtnis.

1.2 Verfahren von Knuth-Morris-Pratt

Die Idee hierbei ist es, die Informationen, die wir jeweils bis zu einem „Mismatch" über den Text an den jeweiligen Stellen erhalten haben, auszunutzen. Dabei wird versucht, das Muster

nach jedem Mismatch um mehr als nur eine Position nach rechts zu rücken. Ziel ist es, zu verhindern, dass die Vergleiche noch einmal über bereits bekannte Zeichen geführt werden müssen.

Beispiel:

Position:	1	2	3	4	5	6	7	8	9	.	.	.	.	.
Text 1:	N	A	N	A	N	A		D	A	S		I	S	T
Muster (erstes Anlegen):	A	N	A	N	A	S								
Muster (zweites Anlegen):		A	N	A	N	A	S							

Beim ersten Anlegen des Musters „ANANAS“ an den Text „NANANA DAS IST“ erfolgt sogleich ein Mismatch. Jedoch beim zweiten Anlegen (an Position 2) erfolgen fünf Übereinstimmungen bevor der erste Mismatch an Position 7 auftaucht. Wegen der Übereinstimmungen kennen wir bereits die fünf Zeichen des Textes vor der aktuellen Position, d. h. diese brauchen wir nicht noch einmal anzuschauen. Doch wie weit müssen wir das Muster nach rechts schieben? Es ist folgendes zu beachten:

(1) Nach der Verschiebung muss garantiert sein, dass links von der aktuellen Position im Muster nur Zeichen stehen, die alle mit dem jeweiligen Zeichen im Text übereinstimmen.

(2) Dabei müssen wir beachten, dass wir das Muster keinesfalls zu weit nach rechts schieben dürfen.

Die erste Eigenschaft (1) ist in unserem Beispiel bei Position 4 im Text sowie bei Position 6 im Text erfüllt. Würden wir das Muster auf Anlegeposition 6 legen, dann könnte uns eventuell ein Vorkommen des Musters im Text (Pattern Matching) verloren gehen. Es wäre also in diesem Fall richtig, das Muster um 2 Positionen nach rechts zu schieben.

Wir nehmen an:

- Die letzten q gelesenen Zeichen im Text stimmen mit den ersten q Zeichen des Musters überein. Es sei P_q das Teilmuster von $P[1]$ bis $P[q]$.
- Das gerade gelesene i-te Zeichen im Text ist verschieden vom $q+1$-ten Zeichen im Muster.

Wir bestimmen vom Anfangsstück des Musters mit Länge q (dem gematchten Teil des Musters) ein Endstück maximaler Länge $l < q$, das ebenfalls Anfangsstück des Musters ist.

Das Muster kann dann um $q-l$ Stellen nach rechts geschoben werden. Position $l+1$ ist dann im Muster die erste Stelle, die man mit dem i-ten Zeichen im Text als nächstes vergleichen muss.

Beispiel:

q	abababababca	l	Shift nach rechts ($= q - l$)
1	a#	0	1
2	ab#	0	2
3	aba#	1	2
4	abab#	2	2
5	ababa#	3	2
6	ababab#	4	2
7	abababa#	5	2
8	abababab#	6	2
9	ababababc#	0	9
10	abababababca	1	9 (kompletter Match)

Algorithmus 2 zeigt eine Realisierung der Knuth-Morris-Pratt Idee. Die Werte von l für $j = 1, \ldots, M$ werden dabei vom Algorithmus InitNext(P) (s. Algorithmus 3) vorausberechnet und in dem Feld $next[1..M]$ abgespeichert. Im Wesentlichen geschieht die Berechnung von InitNext(P) sehr ähnlich wie die eigentliche Idee des Knuth-Morris-Pratt Algorithmus: das Muster wird mit sich selbst verglichen.

1.2.1 Analyse von Knuth-Morris-Pratt

Die Korrektheit des Knuth-Morris-Pratt Algorithmus ergibt sich aus der obigen Argumentation und der korrekten Berechnung des Feldes *next*[] in InitNext(P), die wir näher diskutieren wollen.

Fall 1: $P[l+1] == P[q]$ in Zeile (3). In diesem Fall ist das Endstück des Musters P_q, das Anfangsstück von P_q ist, das bisherige (von P_{q-1}) vereinigt mit $P[q]$.

Fall 2: $P[l+1] \neq P[q]$ in Zeile (3). Dann geht man am besten genauso vor wie im Knuth-Morris-Pratt Algorithmus, nämlich man vergleicht der Reihe nach $P[q]$ mit $next[l]$, $next[next[l]], \ldots$, bis man bei 0 angekommen ist.

Analyse der Laufzeit: Im Algorithmus wird der Index i genau N Mal und der Index j maximal N Mal erhöht. Der Index j kann allerdings höchstens so oft zurückgesetzt werden, wie er erhöht wurde, also insgesamt höchstens N Mal. Das gleiche Argument gilt auch für Algorithmus InitNext(P). Dort kann l höchstens so oft zurückgesetzt werden, wie es

Algorithmus 2 Knuth-Morris-Pratt(T, P)

Eingabe: Text $T = T[1..N]$ und Muster $P = P[1..M]$
Ausgabe: Ausgabe aller Vorkommen von P in T

```
1: InitNext(P); // Initialisiere das Feld next[]
2: j = 0;
3: für i = 1, ..., N {
4:    solange (j > 0&&P[j + 1] ≠ T[i]) {
5:       j = next[j];
6:    }
7:    falls P[j + 1] == T[i] dann {
8:       j = j + 1;
9:    }
10:   falls j == M dann {
11:      Ausgabe: „P an Stelle i − j + 1 im Text gefunden“;
12:      j = next[j];
13:   }
14: }
```

Algorithmus 3 Prozedur InitNext(P)

Eingabe: Muster $P = P[1..M]$
Ausgabe: Initialisiert das Feld *next*[] für Muster P

```
1: next[1] = 0; l = 0;
2: für q = 2, ..., M {
3:    solange (l > 0)&&(P[l + 1] ≠ P[q]) {
4:       l = next[l];
5:    }
6:    falls P[l + 1] == P[q] dann {
7:       l = l + 1;
8:    }
9:    next[q] = l;
10: }
```

insgesamt erhöht wurde, und das ist maximal M. Wir erhalten also als Gesamtlaufzeit des Knuth-Morris-Pratt Algorithmus $\Theta(N + M)$.

Bemerkung: Bei Knuth-Morris-Pratt wird der Text ausschließlich sequentiell durchlaufen, da der Index i niemals zurückgesetzt wird. Dies ist in manchen Anwendungen von Vorteil, z.B. wenn der Text auf externen Medien gespeichert ist, die nur sequentiell in einer Richtung durchlaufen werden können (z.B. Bänder).

1.3 Verfahren von Boyer-Moore

Die Grundidee von Boyer-Moore ist die folgende: Eine Verschiebung des Musters bei Mismatch ist hier davon abhängig, welches Zeichen im Text für den Mismatch verantwortlich ist und wo dieses im Muster auftritt. Dazu wird das Muster von links nach rechts angelegt, aber die Zeichen werden von rechts nach links gelesen. Die Motivation hierfür liegt in der Beobachtung, dass die meisten Textzeichen im Muster nicht auftauchen (da die Musterlänge im Verhältnis zum Alphabet bzw. Text sehr klein ist), und so das Muster sehr oft um die ganze Länge verschoben werden kann.

Wenn man allerdings nur diese Idee (*Last-Verschiebung*) implementiert, dann kann es im schlimmsten Fall zu einer Laufzeit von $O(NM)$ kommen. Deswegen verwendet man noch ein zweites Verschiebungskriterium, das so weit verschiebt, bis die letzten gematchten Zeichen im Text mit den ersten Zeichen in P übereinstimmen (*Suffix-Verschiebung*).

Es gibt also zwei verschiedene Verschiebungskriterien, die jeweils unabhängig voneinander berechnet werden und von denen dann die größere Verschiebung durchgeführt wird. Algorithmus Boyer-Moore(T, P) (s. Algorithmus 4) gibt den Algorithmus an. Dabei wird zunächst die Berechnung der Verschiebungstabellen *last*[] und *suffix*[] aufgerufen.

Die Ersetzung von Zeile (11) und (13) durch den Befehl $i = i + 1$ ergibt einen naiven Algorithmus mit Laufzeit $\Theta(NM)$.

1.3.1 Berechnung von *last*[]

Wir betrachten die Situation, dass die erste Nichtübereinstimmung bei j auftrat, d.h. $P[j] \neq T[i + j]$ für ein $j, 1 \leq j \leq M$. Das Zeichen an dieser Stelle im Text sei c. In diesem Fall verschieben wir P so weit nach rechts, bis c im Text über dem rechtesten Zeichen gleich c in P ist. Falls c nicht in P vorkommt, dann kann P bis hinter die Position $i + j$ verschoben werden.

Lemma 1.1 *Sei k der größte Index aus $1 \leq k \leq M$, für den gilt $T[i + j] = P[k]$, falls*

Algorithmus 4 Boyer-Moore (T, P)

Eingabe: Text $T = T[1..N]$ und Muster $P = P[1..M]$
Ausgabe: Ausgabe aller Vorkommen von P in T

```
1: InitLast(P);
2: InitSuffix(P);
3: i = 0; // Position vor der Anlegestelle von P
4: solange i ≤ N − M {
5:    j = M;
6:    solange (j > 0&&P[j] == T[i + j]) {
7:       j = j − 1;
8:    }
9:    falls j == 0 dann {
10:      Ausgabe: „P an Stelle i + 1 im Text gefunden“;
11:      i = i + suffix[1];
12:   } sonst {
13:      i = i + max(suffix[j], j − last[T[i + j]]);
14:   }
15: }
```

vorhanden, sonst $k = 0$. Dann ist es korrekt, i um $j - k$ zu erhöhen.

Beweis: Fall 1: $k = 0$. Dann sind alle anderen Zeichen links von j ungleich $T[i + j] \longrightarrow i = i + j\surd$
Fall 2: $k < j$. Das rechteste Erscheinen des Mismatch-Zeichens c in T liegt links von j in P. In diesem Fall ist eine Verschiebung um $j - k > 0$ nach rechts korrekt.
Fall 3: $k > j$. Hier wäre $j - k < 0$ und es wird keine *last*-Verschiebung durchgeführt. □

Der Algorithmus InitLast(P) (s. Algorithmus 5) berechnet jeweils die Position des letzten Vorkommens jedes Zeichens aus dem Alphabet Σ im Muster P.

Algorithmus 5 Prozedur InitLast(P)

Eingabe: Muster $P = P[1..M]$

```
1: für alle c aus dem Alphabet Σ {
2:    last[c] = 0;
3: }
4: für j = 1, ..., M {
5:    last[P[j]] = j;
6: }
```

1.3.2 Berechnung von *suffix*[]

Verschiebe P so weit nach rechts, bis die bisher untersuchten gematchten Zeichen im Text mit den ersten Zeichen im Muster übereinstimmen. Falls keinerlei Übereinstimmung herrscht, kann das Muster bis ganz hinter die Position $i + j$ geschoben werden.

Wir betrachten wieder die Situation, dass der erste Mismatch bei j aufgetaucht ist, d.h. $P[j] \neq T[i + j]$.

Wir definieren das *Suffix* als die kleinste Verschiebung s, so dass

$$\begin{aligned} P[j+1-s] &= P[j+1] \\ \vdots &\quad \vdots \\ P[M-s] &= P[M] \end{aligned}$$

Eine alternative Sichtweise ist die folgende: $suffix[j] = M - k$, $0 \leq k < M$, so dass $P[j+1], \ldots, P[M]$ Suffix von P_k ist, wobei $P_k = P[1] \ldots P[k]$.

Zur Berechnung unterscheiden wir zwei Fälle.

1. Fall: Das Suffix befindet sich nur am Anfang von P.
Dann gilt $suffix[j] = M - \pi[M]$ $\forall j$, wobei $\pi[M]$ die Länge des längsten Präfixes von P ist, sodass $P_{\pi[M]}$ das echte Suffix von P darstellt.

2. Fall: Das Suffix befindet sich nicht nur am Anfang von P.
In diesem Fall kann das Suffix im *invertierten* (d.h. von hinten nach vorne gelesenen) Muster P^{-1} mit Hilfe von *next*[] (s. Abschnitt 1.2) ausgerechnet werden. Es gilt $suffix[j] \leq q - next[q]$ für alle $1 \leq q \leq M$ und $j = M - next[q]$. Intuitiv suchen wir alle diejenigen Längen q, deren Endungen mit dem gesuchten Präfix übereinstimmen. Und das sind genau diejenigen, für die gilt $next[q] = M - j$. Das Suffix kann berechnet werden als

$$suffix[j] = \min(\{M - \pi[M]\} \cup \{q - next[q] : 1 \leq q \leq M \text{ und } j = M - next[q]\})$$

Der Algorithmus InitSuffix(P) (s. Algorithmus 6) bestimmt zunächst im gegebenen Muster das Suffix, das Fall 1 entspricht (d. h. am Anfang des Musters). Danach bestimmt er im invertierten Muster P^{-1} das Suffix, das Fall 2 entspricht. Dazu berechnet er im invertierten Muster das Feld $\overline{next}$[].

1.3.3 Analyse von Boyer-Moore

Man kann zeigen, dass die Laufzeit des Boyer-Moore Verfahrens in der Ordnung von $\Theta(M+N)$ liegt. Dies gilt aber nur, wenn beide Verschiebeverfahren verwendet werden.

Algorithmus 6 Prozedur InitSuffix(P)

Eingabe: Muster $P = P[1..M]$

```
1: Berechne InitNext(P) → next[]
2: Berechne InitNext(P^{-1}) → \overline{next}[]
3: für j = 1,...,M {
4:    suffix[j] = M − next[M];
5: }
6: für q = 1,...,M {
7:    j = M − \overline{next}[q];
8:    falls suffix[j] > q − \overline{next}[q] dann {
9:       suffix[j] = q − \overline{next}[q];
10:   }
11: }
```

Die Verschiebung mittels *last* ist sehr einfach, die Suffix-Verschiebung etwas aufwändiger zu programmieren. Aus diesem Grund wird der Algorithmus in der Praxis oft ohne die Suffix-Verschiebung implementiert. Das Verfahren ist in der Praxis in beiden Fällen sehr schnell.

1.4 Signaturverfahren von Karp und Rabin

Eine andere Möglichkeit besteht darin, mit Hilfe einer Hashfunktion jeden Teilstring eines Textes auf einen Zahlenwert abzubilden. Ist diese Funktion h hinreichend „gut", dann werden identische Teilstrings in den meisten Fällen auf identische Funktionswerte abgebildet. Man könnte also h auch auf das gesuchte Muster anwenden und dann überprüfen, ob diese „Signatur" (in der Literatur auch als „Fingerprint" bezeichnet) irgendwo im Text vorkommt.

Man berechnet also einen Hash-Wert $h(P_1, P_2, \ldots, P_M)$ für das gesuchte Muster, der dann der Reihe nach mit den Hash-Werten $h(T_{i+1}, T_{i+2}, \ldots, T_{i+M})$ für die jeweils aktuellen Teilstrings des zu durchsuchenden Textes verglichen wird. Findet man einen identischen Funktionswert, so werden die einzelnen Zeichen an der betreffenden Stelle überprüft, um sicher zu stellen, dass nicht zufällig ein anderer Teilstring den gleichen Hash-Wert erzeugt hat.

Es lässt sich mit Recht einwenden, dass bisher ja nur der direkte Vergleich von Zeichen durch ein Berechnungsverfahren ersetzt wurde, das wieder auf alle M Zeichen ab der jeweiligen Position im Text zugreifen muss. Das ergäbe keinerlei Verbesserung gegenüber dem naiven Verfahren. Der Trick liegt nun aber darin, dass man eine Hashfunktion wählt, die *inkrementell* berechnet werden kann. Wenn man aus einem Wert der Hashfunktion jenen für die nächste Position berechnen kann, ohne die Zeichen im Überlappungsbereich nochmals betrachten zu müssen, sinkt der ursprüngliche Aufwand $O(MN)$ plötzlich auf $O(M + N)$.

Ein Beispiel für eine inkrementell berechenbare Hashfunktion lässt sich über die Interpretation jeder Zeichenkette als d-adische Zahl ableiten, wobei d die Größe des Alphabets ist. Dann kann man für die d-adische Zahl k eine einfache Hashfunktion z.B. als $h(k) = k \mod p$ definieren, wobei p eine geeignete, ausreichend große Primzahl ist. Die inkrementelle Berechnung funktioniert durch „Verschieben" der Stellenwerte in k. Die höchstwertige Stelle (das am weitesten links stehende Zeichen) wird aus der Zahl entfernt, die ganze Zahl mit der Basis d multipliziert (was die Stellen im d-adischen System nach links verschiebt), und an der Einerstelle wird das nächste Zeichen hinzugenommen.

Es lässt sich zeigen, dass das Verfahren von Karp und Rabin mit hoher Wahrscheinlichkeit nur $O(M + N)$ Schritte benötigt. Wenn die arithmetischen Operationen in einem konkreten Anwendungsfall ausreichend schnell durchgeführt werden können, ist es durchaus eine interessante Alternative zu den gängigen Verfahren mit direktem Textvergleich.

1.5 Approximative Zeichenkettensuche

Das Problem der exakten Suche von Zeichenketten in Texten lässt sich verallgemeinern auf die Suche nach *ungefähr* passenden Textstellen. Zum Beispiel könnte man nach einer Stelle suchen, sodass sich der gefundene Textausschnitt nur durch eine bestimmte maximale Anzahl von Zeichen vom Muster unterscheidet.

Für das naive Verfahren ist diese Verallgemeinerung relativ leicht durchzuführen, aber auch mit dem üblichen algorithmischen Aufwand verbunden. Man kann einfach den Teil des Algorithmus, der die Prüfung auf Übereinstimmung durchführt, mit einem Zähler für die Anzahl der Mismatches versehen. Unterschreitet diese Anzahl einen bestimmten Wert, hat man einen approximativen Treffer gefunden.

Es ist deutlich schwieriger, ein effizientes Verfahren zu finden, und die Details überschreiten auch den Rahmen dieses Skriptums. Es sei aber erwähnt, dass man zum Beispiel Editieroperationen definieren kann, durch die bei Ausführung in einer bestimmten Reihenfolge der Textausschnitt an einer Fundstelle in das exakte Suchmuster umgewandelt werden kann. Die Anzahl nötiger Operationen kann in der Folge als Kriterium herangezogen werden. Um die beste Abfolge von Editieroperationen, die sogenannte Editierdistanz, zu bestimmen, kann zum Beispiel lineare Programmierung eingesetzt werden.

1.6 Weiterführende Literatur

Suchen in Texten wird z.B. in den Büchern von Sedgewick und Cormen, Leiserson und Rivest (s. Literaturliste (1) bis (3)) ausführlich beschrieben.

Kapitel 2

Randomisierte Algorithmen

Unter einem **Randomisierten Algorithmus** (bzw. Probabilistischen Algorithmus) verstehen wir einen deterministischen Algorithmus, der als zusätzliche Elementaroperationen Zufallsexperimente durchführen kann.

Man unterscheidet grundsätzlich zwei Klassen von randomisierten Algorithmen, nämlich Las-Vegas-Verfahren, die stets ein korrektes Ergebnis berechnen, und Monte-Carlo-Verfahren, die ein Ergebnis berechnen, das mit einer gewissen Fehlerwahrscheinlichkeit behaftet ist.

Ein wesentlicher Unterschied zwischen randomisierten und deterministischen Algorithmen besteht in der Analyse. Während man bei deterministischen Algorithmen in erster Linie die worst-case Komplexität betrachtet, analysiert man bei randomisierten Algorithmen die sogenannte *expected-case* Komplexität, die das Verhalten des Algorithmus über alle möglichen Ausgänge der durchgeführten Zufallsexperimente mittelt.

Tabelle 2.1 zeigt weitere Unterschiede. Während der Aufbau deterministischer Algorithmen sehr komplex sein kann, ist der Aufbau randomisierter Algorithmen meist sehr einfach. Durch die expected-case Analyse ist jedoch die Analyse relativ komplex im Vergleich zur Analyse von deterministischen Algorithmen. Die Laufzeit randomisierter Algorithmen kann garantiert sein, oft hängt sie jedoch vom Ausgang der Zufallsexperimente ab.

2.1 Randomisiertes Quicksort

Beim klassischen Quicksort-Verfahren besteht eine latente Gefahr der Entartung, wenn nur ein einziges Pivot-Element, z.B. immer vom Ende des zu sortierenden Bereichs, gewählt wird. Verbessert wird das Verfahren üblicherweise durch die Auswahl des mittleren von drei

Algorithmus	Deterministisch	Randomisiert
Zufallsexperimente	nein	ja
Aufbau	komplex	einfach
Analyse	einfach	komplex
Laufzeit	garantiert	variabel/probabilistisch
Korrektheit	garantiert	garantiert/probabilistisch

Tabelle 2.1: Unterschiede zwischen deterministischen und randomisierten Algorithmen

Elementen, die am Anfang, in der Mitte und am Ende der Folge entnommen werden. Aber auch dadurch kann eine Entartung nur etwas gemildert, jedoch nicht verhindert werden.

Dem randomisierten Quicksort, das zur Klasse der Las-Vegas-Verfahren zählt, liegt ein einfacher Gedanke zugrunde: Das Pivot-Element wird nicht mehr deterministisch an einer bestimmten Position entnommen, sondern zufällig aus dem gesamten Bereich der zu sortierenden (Teil-) Folge gewählt. Die Wahrscheinlichkeit, gewählt zu werden, ist für jedes Element gleich groß. Man kann zeigen, dass sich dann für die Laufzeit ein Erwartungswert von $O(n \log n)$ ergibt.

Dieser Wert ist schon vom „klassischen" Algorithmus her bekannt, aber es besteht ein wichtiger Unterschied: Während der Erwartungswert bei der deterministischen Version des Algorithmus aus einer Mittelung über alle möglichen Eingabefolgen entsteht, ergibt er sich beim randomisierten Algorithmus durch die zufällige Auswahl der Pivot-Elemente. Für den deterministischen Algorithmus gibt es aber Eingabefolgen, die *immer* eine Entartung der Laufzeit verursachen, während das beim randomisierten Quicksort höchstens durch Zufall vorkommen kann, sich aber durch die neue Auswahl der Pivot-Elemente bei neuerlicher Sortierung der gleichen Eingabefolge in der Regel dramatisch ändern kann.

Interessant ist eine alternative Betrachtungsweise, durch die der Zufall auf überraschende Art ins Spiel gebracht werden kann. Statt nämlich ein zufälliges Pivot-Element zu wählen, könnte man auch den naiven deterministischen Algorithmus verwenden, wenn zu Beginn die Eingabefolge zufallsgesteuert permutiert wird. Das bedeutet, dass wieder jedes Element die gleiche Chance hat, als Pivot-Element gewählt zu werden. Auch wenn also der Gedanke zunächst absurd erscheinen mag, eine zu sortierende Folge erst einmal gründlich zu mischen, verbessert sich so der Erwartungswert wie beschrieben. Diese Vorgangsweise wird manchmal auch als *Input-Randomisierung* bezeichnet.

2.2 Randomisierter Primzahltest

Im Folgenden wenden wir ein Monte-Carlo Verfahren, nämlich den Algorithmus von Miller-Rabin, auf das Primzahlproblem an.

Primzahltest
Gegeben: Ganze Zahl $n > 2$ (üblicherweise sehr groß)
Gesucht: Antwort auf die Frage: Ist n eine Primzahl?

Dieses Problem ist in der Kryptographie (z.B. für das RSA-Verfahren) eines der zentralen Probleme überhaupt. Die Zahlen n, um die es dort geht, sind Teile von *Schlüsseln*, die bereits jetzt eine Länge von über 100 Stellen besitzen. Mit steigender Rechnerkapazität wird diese Länge weiter ansteigen.

Weil man solche langen Zahlen nicht mehr in konstanter Zeit addieren kann, gehen wir hier zur sogenannten *Bitkomplexität* über. D.h., wir stellen n als Binärzahl mit genau $k = \lceil \log(n+1) \rceil$ Bits dar und zählen die Bitoperationen einzeln.

Das folgende Lemma zeigt, dass etwa jede $(\ln n)$-te Zahl eine Primzahl ist.

Lemma 2.1 *Sei $\pi(n)$ die Anzahl der Primzahlen $\leq n$. Für $n \geq 17$ gilt:*

$$\frac{n}{\ln n} \leq \pi(n) \leq 1,2551 \frac{n}{\ln n}$$

Für den Grenzwert gilt:

$$\lim_{n \to \infty} \frac{\pi(n)}{n/\ln n} = 1$$

Zunächst untersuchen wir die naive *Divisionsmethode*. Diese testet nacheinander, ob n durch 2 oder durch eine ungerade Zahl aus dem Bereich $\{3, \ldots, \lfloor \sqrt{n} \rfloor\}$ teilbar ist. Die Anzahl der Divisionen ist in diesem Fall in $O(\sqrt{n}) = O(2^{\frac{k}{2}})$, wobei $k = \Theta(\log n)$ (Bitkomplexität). Somit ist das naive Verfahren für große Zahlen nicht praktikabel.

2.2.1 Algorithmus von Miller-Rabin

Der Algorithmus von Miller-Rabin ist ein randomisiertes Verfahren zum Primzahltest und gehört der Klasse der Monte-Carlo Verfahren an. D.h., das Ergebnis ist mit einer gewissen Fehlerwahrscheinlichkeit behaftet. Die Idee beruht auf dem folgenden wichtigen Satz von Fermat.

Satz 2.1 (Satz von Fermat)
Ist n eine Primzahl, so gilt für alle $a \in \{1, \ldots, n-1\}$

$$a^{n-1} \equiv 1 \mod n. \tag{2.1}$$

Der Satz von Fermat liefert ein nützliches Kriterium zum Test der Primalität von n. Fixieren wir z.B. eine Basis a, und liefert unser Test

$$a^{n-1} \not\equiv 1 \mod n,$$

so ist n sicherlich keine Primzahl, und a heißt *Zeuge* für die Zusammengesetztheit von n. Tatsächlich gibt es sogenannte *Pseudoprimzahlen*, die die Gleichung für ein a zwar erfüllen, aber keine Primzahlen sind. Jedoch liefert bereits dieser einfache Test für 100-stellige Zahlen n und $a = 2$ eine sehr kleine Fehlerrate von etwa 10^{-13}. Weiterhin gibt es die sogenannten *Carmichael-Zahlen*, die für alle $a \in \mathbb{Z}_n*$ die Gleichung 2.1 erfüllen, aber keine Primzahlen sind. Dabei ist $\mathbb{Z}_n*$ der Restklassenring bezüglich n definiert als

$$\mathbb{Z}_n* = \{a \in \{1, \ldots, n-1\} \mid \text{ggt}(a, n) = 1\}.$$

Carmichael-Zahlen sind allerdings sehr selten: es gibt nur 255 von ihnen, die kleiner als 10^8 sind.

Diese Überlegungen führen uns direkt zur Idee des Algorithmus von Miller-Rabin (siehe Algorithmus 7). Wir wählen eine Basis a zufällig und gleich verteilt aus der Menge $\{1, \ldots, n-1\}$ aus und machen den Test

$$a^{n-1} \equiv 1 \mod n.$$

Wenn die Gleichung nicht erfüllt ist, dann haben wir einen Zeugen für die Zusammengesetztheit von n, andernfalls wählen wir eine weitere Basis. Dieser Test wird insgesamt s Mal wiederholt. Das zugrunde liegende Prinzip nennt man *Random Search*, da ein Raum mit unbekannter Struktur zufällig auf Elemente mit einer gewünschten Eigenschaft durchsucht wird.

Algorithmus 7 Primzahltest von Miller-Rabin

```
1: für i = 1, ..., s {
2:   falls Zeuge (Random (1, n − 1), n) dann {
3:     Retourniere „nicht prim“
4:   }
5: }
6: Retourniere „prim“
```

Es fehlt nun noch ein effizienter Algorithmus für die Berechnung von $a^{n-1} \mod n$. Algorithmus 8 (*Zeuge*) löst das Problem durch *Repeated Squaring*. Sei $(b_{k-1}, \ldots, b_0)$ die Binärdarstellung von $n-1$. Die Funktion *Zeuge()* vermeidet große Zahlen, indem sie die Vertauschbarkeit von Multiplikation und Modulo-Berechnung ausnutzt:

$$a^{2c} \mod n = (a^c \mod n)^2 \mod n \tag{2.2}$$

$$a^{c+1} \mod n = (a^c \mod n) \cdot a \mod n \tag{2.3}$$

Algorithmus 8 Zeuge (a, n)

```
1: b = (b_{k-1}, ..., b_0) = Binärdarstellung von n - 1;
2: d = 1; // c = 0
3: für i = k - 1, ..., 0 {
4:    d = (d * d) mod n; // c = 2c
5:    falls b_i == 1 dann {
6:       d = (d * a) mod n; // c = c + 1
7:    }
8: }
9: falls (d ≠ 1) dann {
10:   Retourniere true; // a ist Zeuge für „n ist nicht prim“
11: } sonst {
12:   Retourniere false; // a ist kein Zeuge
13: }
```

Dabei gilt in jedem Schleifendurchlauf die Invariante

$$d = a^c \mod n,$$

wobei c durch Verdoppeln und Inkrementieren von 0 auf $n-1$ erhöht wird. Dabei gilt in jedem Schleifendurchlauf i, dass $c = (b_{k-1}, \ldots, b_i)$. Somit gilt nach dem letzten Schleifendurchlauf $i = 0$, dass $c = (b_{k-1}, \ldots, b_0) = n$.

Analyse der Laufzeit: Die Multiplikation zweier k-Bit Zahlen (Zeile 3) hat Bitkomplexität $O(k^2)$. Dies sind $O(k^3)$ für jeden der s Aufrufe von *Zeuge()*.

Satz 2.2 *Der Monte-Carlo Algorithmus von Miller-Rabin zum Test der Primalität von n mit Binärdarstellung $(b_{k-1}, \ldots, b_0)$ hat eine polynomielle Laufzeit von $O(s \cdot k^3)$.*

Analyse der Fehlerrate: Man kann das folgende Theorem mit Hilfe von gruppentheoretischen Überlegungen beweisen. Unter anderem gehören alle Nicht-Zeugen zur Menge $\mathbb{Z}_N*$.

Satz 2.3 *Für $n > 2$, ungerade und nicht prim, gibt es mindestens $\frac{n-1}{2}$ Zeugen für die Zusammengesetztheit.*

Daraus folgt, dass die zufällige Auswahl von a in jeder Iteration mit einer Wahrscheinlichkeit von mindestens $1/2$ einen Zeugen für die Zusammengesetztheit einer nicht-primen Zahl n liefert. Da der Algorithmus *Miller-Rabin* nur dann ein falsches Testergebnis ausgibt, wenn nach s Iterationen immer noch kein Zeuge gefunden worden ist und die Zahl n tatsächlich nicht prim ist, erhalten wir insgesamt eine Fehlerwahrscheinlichkeit von höchstens $\frac{1}{2^s}$. Tatsächlich ist die Fehlerrate in der Praxis um einige Größenordnungen kleiner.

2.3 Randomisierte Datenstrukturen: Skiplisten

Datenstrukturen heißen *randomisiert*, wenn einige oder alle der Methoden zum Aktualisieren der Struktur eine Zufallskomponente enthalten. Oft sind solche Strukturen sehr effizient in der Praxis und einfach zu implementieren. In der Laufzeitanalyse wird mit Erwartungswerten gearbeitet. Die dabei nötige Wahrscheinlichkeitsrechnung ist nicht immer trivial. Meist sind die worst-case Schranken deutlich schlechter als die Erwartungswerte, weshalb es für eine einzelne Operation keine guten Laufzeitgarantien gibt.

In diesem Abschnitt behandeln wir die randomisierte Datenstruktur der **Skiplisten**, die das Wörterbuchproblem lösen. Sie stellen Methoden zum Einfügen, Löschen und Suchen von Datensätzen bereit und jede dieser Operationen hat eine erwartete Laufzeit von $O(\log n)$ wenn n die Anzahl der gespeicherten Elemente ist.

Skiplisten sind eine Verallgemeinerung von verketteten Listen. In beiden Fällen sind die Daten in Containern abgelegt, die einen Schlüssel (zur Vereinfachung gehen wir von positiven ganzen Zahlen aus) und einen Zeiger auf den nachfolgenden Container enthalten. Im Gegensatz zu einfach verketteten Listen können Container in Skiplisten auch Zeiger auf Container enthalten, die weiter hinten in der Liste stehen als der direkte Nachfolger. Deshalb kann man bei der Suche nach einem Schlüssel Container überspringen (überspringen = engl. „to skip").

In einer Skipliste hat jeder Container c eine *Höhe* $h(c)$, die eine nicht-negative ganze Zahl ist. Diese ist um 1 kleiner als die Anzahl der Zeiger des Containers auf nachfolgende Container. Die Zeiger sind nummeriert von 0 bis $h(c)$. Der Zeiger i für $0 \leq i \leq h(c)$ zeigt auf den Container, der 2^i Positionen hinter Container c steht, oder auf das ausgezeichnete Endelement der Skipliste, falls die Liste nicht lang genug ist. Wir bezeichnen i auch als das *Niveau* des Zeigers.

Hat ein Container c die Höhe 3, so enthält er 4 Zeiger. Der Zeiger mit dem höchsten Niveau (Niveau 3) zeigt dabei auf den Container, der $2^3 = 8$ Positionen hinter c in der Liste steht, während der Zeiger mit Niveau 2 auf den Container 4 Positionen hinter c zeigt. Je größer die Höhe eines Containers, desto mehr der folgenden Container können wir bei einer Suche überspringen (falls das gesuchte Element nicht in dem übersprungenen Bereich liegt).

2.3.1 Perfekte Skiplisten

Abbildung 2.1 zeigt eine sogenannte *perfekte Skipliste*. Es gibt zwei Pseudo-Container, die keine Daten enthalten. Einer dieser Behälter markiert den Anfang der Liste (*Kopf*) und einer das Ende. Das Ende hat einen Schlüssel, der größer als alle Schlüssel in der Skipliste ist. Kopf und Ende haben die gleiche Höhe wie der höchste echte Container. Für alle echten

Container gilt: Jeder 2^i-te Container hat einen Zeiger auf den 2^i Positionen weiter hinten stehenden Container oder auf das Ende, falls kein solcher Container existiert. Dadurch ergibt sich die Höhe für jeden Container in der perfekten Skipliste.

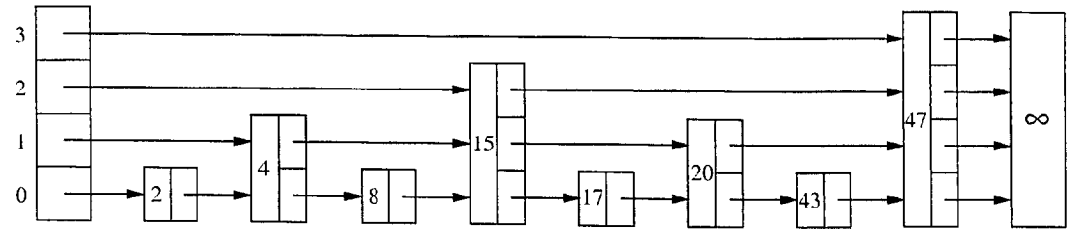


Abbildung 2.1: Eine perfekte Skipliste mit 8 Containern

Sei nun n die Anzahl der echten Container in der Skipliste. Wir nehmen zur Vereinfachung an, dass $n = 2^l$ für eine natürliche Zahl l gilt. Wir wollen nun alle Zeiger in unserer perfekten Skipliste zählen. Ignorieren wir die Zeiger, die vom Kopf ausgehen, so ist die Anzahl aller Zeiger:

$$n + \frac{n}{2} + \frac{n}{4} + \cdots + 1 = \sum_{i=0}^{\log n} \frac{n}{2^i} = 2n - 1 \quad .$$

Dies liegt daran, dass es n Container mit Zeigern auf Niveau 1 gibt, $\frac{n}{2}$ Container mit Zeigern auf Niveau 2, und so weiter. Es ist leicht zu sehen, dass der Kopf $\log n + 1$ Zeiger hat. Also ist die Gesamtzahl der Zeiger in einer Skipliste $O(n)$. Es folgt, dass die gesamte Datenstruktur $O(n)$ Platz benötigt.

Wie sucht man nun in einer perfekten Skipliste nach einem Element? Wir gehen im folgenden Pseudocode davon aus, dass die Höhe von Container c als $c.höhe$ bezeichnet wird und $c.next[i]$ für $i \leq c.höhe$ der Container ist, auf den der Zeiger von c auf Niveau i zeigt (also der Container, der 2^i Positionen hinter c kommt, bzw. das Ende). Für die Skipliste L bezeichnet $L.kopf$ den Kopf der Liste und $L.höhe$ die Höhe des Kopfes.

Führen wir Algorithmus 9 auf der perfekten Skipliste von Abbildung 2.1 aus, um den Container mit Schlüssel 17 zu finden, so vergleichen wir der Reihe nach mit den Schlüsseln 47, 15, 47, 20 und 17. Der Suchalgorithmus hat die Eigenschaft, dass man bei einer perfekten Skipliste für jedes Niveau i höchstens einem Zeiger folgen muss d.h. die Anweisung $p = p.next[i]$ wird für jedes i höchstens ein Mal durchgeführt. Die Bedingung in der **solange**-Anweisung kann für ein i zwei Mal geprüft werden, ist beim zweiten Mal aber mit Sicherheit nicht erfüllt.

Aus diesen Beobachtungen und aus der Tatsache, dass eine perfekte Skipliste mit n Elementen eine Höhe hat, die logarithmisch mit der Anzahl der enthaltenen Elemente wächst, folgt eine Laufzeit von $O(\log n)$ für jeden Suchvorgang. Problematisch ist aber das Einfügen und Löschen in einer perfekten Skipliste, weil dabei eventuell eine komplette Reorganisation der Liste nötig wäre, um weiterhin logarithmische Laufzeit für das Suchen

Algorithmus 9 Suchalgorithmus $Suche(L, x)$ für Skiplisten

Eingabe: Skipliste L, ganze Zahl x
Ausgabe: Zeiger auf Container mit Schlüssel x falls existent und NULLsonst

```
p = L.kopf
für i = L.höhe bis 0 {
  // Folge Zeigern auf Niveau i
  solange p.next[i].key < x {
    p = p.next[i]
  }
}
p = p.next[0]
falls p.key == x dann {
  // x kommt an Position p in L vor
  return p
} sonst {
  // x kommt nicht in L vor
  return NULL
}
```

zu gewährleisten. Die Reorganisation hätte dann Laufzeit $O(n)$.

2.3.2 Randomisierte Skiplisten

Die Idee ist nun, nicht mehr zu fordern, dass der Container auf Position i eine bestimmte Höhe hat, sondern dass die Verteilung der Höhen in der Skipliste in etwa der Verteilung in einer perfekten Skipliste entspricht. Es soll von jeder Höhe also ungefähr gleich viele Container wie in einer perfekten Skipliste geben, und diese sollen auch gleichmäßig in der Liste verteilt sein.

Die Methode, um dies zu erreichen, ist verblüffend einfach. Will man in die Skipliste ein neues Element einfügen, so sucht man zuerst die Position, an der es eingefügt werden soll. Dazu verwendet man den schon vorgestellten Suchalgorithmus. Dann fügt man dort einen neuen Container ein, dessen Höhe man durch folgendes Zufallsexperiment bestimmt: Erst setzt man die Höhe auf 0. Dann wirft man eine Münze und vergrößert die Höhe um 1, falls „Zahl" geworfen wird. Dies tut man solange, bis zum ersten Mal „Kopf" erscheint. Daraus ergibt sich folgende Wahrscheinlichkeit, dass die Höhe des neuen Containers c gleich i ist:

$$\text{prob}(c.höhe = i) = \frac{1}{2^{i+1}} = 2^{-i-1}$$

Verwendet man zum Einfügen in eine Skipliste dieses Verfahren, so bezeichnet man die entstandene Datenstruktur als *randomisierte Skipliste*. Abbildung 2.2 zeigt ein Beispiel für

eine randomisierte Skipliste.

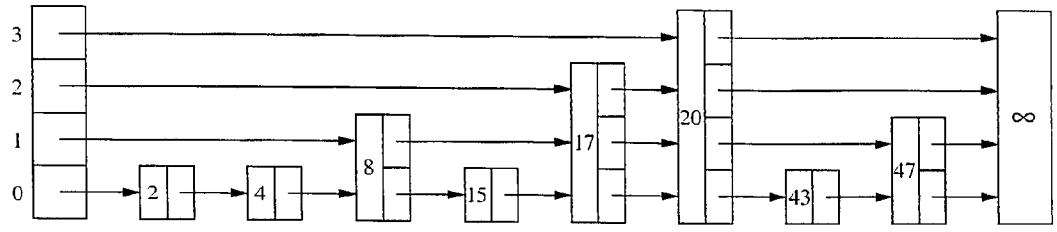


Abbildung 2.2: Eine randomisierte Skipliste

In einer konkreten Implementierung der Skipliste wird man natürlich die maximale Höhe eines Containers beschränken. In der folgenden Analyse werden wir aber davon ausgehen, dass die Höhe unbeschränkt ist. Das ist keine große Einschränkung, denn die Wahrscheinlichkeit, dass z.B. ein bestimmter Container eine Höhe ≥ 15 hat, ist

$$\sum_{i=15}^{\infty} 2^{-i-1} \leq 2^{-15} \quad ,$$

also extrem klein.

Nehmen wir nun also an, wir wollen einen neuen Container c mit Schlüssel x in die Skipliste einfügen und benutzen eine Funktion $randomhöhe()$, welche das oben beschriebene Münzwurfexperiment durchführt, um die Höhe von c zu bestimmen.

Unsere Suche nach x hat beim Container c_l geendet. Dieser Container hat den größten Schlüssel der Liste, der nicht größer als x ist. Wir wollen c also direkt hinter c_l in die Liste einfügen. Dazu müssen wir auch einige Zeiger der Skipliste umhängen. Sei p ein Zeiger auf Niveau i mit $i \leq c.höhe$, der in einem Container c_1 links von c anfängt und auf einen Container c_2 rechts von c zeigt. Wir müssen p in zwei Zeiger aufspalten: der erste beginnt in c_1 und zeigt auf c, während der zweite in c beginnt und auf c_2 zeigt.

Um dies zu ermöglichen, sammelt man bei der Suche nach dem Schlüssel x die Quellen all dieser Zeiger in einem Zeiger-Array $update$: Für jedes i enthält $update[i]$ einen Zeiger auf den am weitesten rechts liegenden Container mit Höhe i links von c. Dadurch kann das Aufspalten der Zeiger effizient durchgeführt werden. Ist die Höhe von c größer als die bisherige Listenhöhe, muss auch die Höhe des Kopfes und des Endes der Liste aktualisiert werden. Algorithmus 10 zeigt im Detail, wie ein neuer Container in die Skipliste eingefügt wird.

Das Entfernen eines Schlüssels x aus der Skipliste erfolgt völlig analog zum Einfügen. Man sucht wieder nach x und merkt sich in $update[i]$ für jedes i einen Zeiger auf das jeweils am weitesten rechts gelegene Element in L mit Höhe i, das gerade noch links von x liegt. Dies ermöglicht wieder das effiziente Umhängen der Zeiger. Ist die maximale Listenhöhe

Algorithmus 10 Einfügen in eine randomisierte Skipliste

Eingabe: Skipliste L und einzufügender Schlüssel x
Ausgabe: Ein neuer Container mit Schlüssel x wird in L eingefügt.

```
p = L.kopf
für i = L.höhe bis 0 {
  solange p.next[i].key < x {
    p = p.next[i]
  }
  update[i] = p
}
p = p.next[0]
falls p.key == x dann {
  // Schlüssel x kommt schon vor
} sonst {
  // einfügen
  neuehöhe = randomhöhe()
  falls neuehöhe > L.höhe dann {
    // neues Element direkt mit Kopfelement verknüpfen und Listenhöhe aktualisieren
    für i = L.höhe + 1 bis neuehöhe {
      update[i] = L.kopf
    }
    L.höhe = neuehöhe
  }
  // erzeuge neues Element mit Höhe neuehöhe und Schlüssel x
  p = newContainer(x, neuehöhe)
  für i = 0 bis neuehöhe {
    // füge p in die Niveau-i-Listen jeweils unmittelbar nach dem Element update[i] ein
    p.next[i] = update[i].next[i]
    update[i].next[i] = p
  }
}
```

der Skipliste durch das Entfernen gesunken, muss man den Kopf und das Ende entsprechend aktualisieren. Die Details zeigt Algorithmus 11.

Algorithmus 11 Entfernen eines Elements aus einer Skipliste

Eingabe: Skipliste L und Schlüssel x
Ausgabe: Der Container mit Schlüssel x wird aus L entfernt.

```
p = L.kopf
für i = L.höhe bis 0 {
  solange p.next[i].key < x {
    p = p.next[i]
  }
  update[i] = p
}
p = p.next[0]
falls p.key == x dann {
  // Container p entfernen und gegebenenfalls Listenhöhe aktualisieren
  für i = 0 bis p.höhe {
    // entferne p aus Niveau-i-Liste
    update[i].next[i] = p.next[i]
  }
  solange L.höhe ≥ 1 und L.kopf.next[L.höhe].key = ∞ {
    L.höhe = L.höhe − 1
  }
}
```

Eine besondere Eigenschaft, die Skiplisten von anderen Datenstrukturen unterscheidet, welche das Wörterbuchproblem lösen (z.B. Suchbäume), ist der Umstand, dass die Liste nach Entfernen von Elementen die selbe Struktur hat, als wäre das Element nie in der Liste gewesen. Also kann eine Skipliste nie durch Entfernen eines Elements entarten, und die „Zufälligkeit" der Struktur bleibt erhalten. Das Aussehen einer Skipliste hängt nicht davon ab, in welcher Reihenfolge die Elemente eingefügt oder entfernt wurden. Deshalb kann sie nie degenerieren.

2.3.3 Analyse Randomisierter Skiplisten

Zuerst wollen wir die Laufzeit für die Suche nach einem Element abschätzen. Abbildung 2.3 zeigt die Suche nach Schlüssel 16 in der randomisierten Skipliste von Abbildung 2.2. Die gestrichelten Pfeile zeigen an, welche Zeiger die Suche durchläuft. Die Suche verläuft auch bei der randomisierten Skipliste gleich wie bei der perfekten Skipliste (siehe Algorithmus 9).

Wir wollen den Erwartungswert der Länge des Suchpfades für einen Schlüssel x berechnen. Zuerst berechnen wir die erwartete Höhe einer Skipliste. Die Wahrscheinlichkeit, dass ein

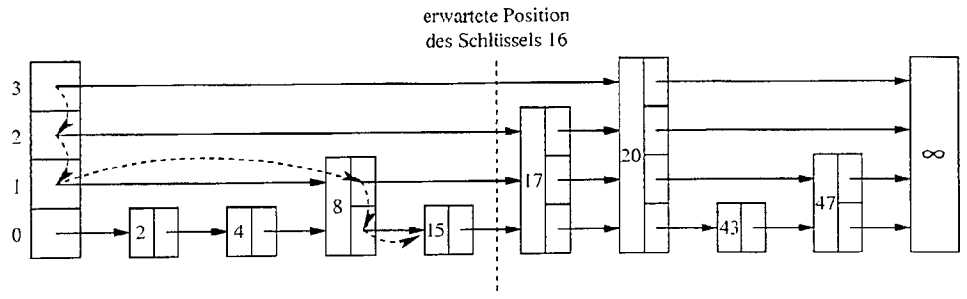


Abbildung 2.3: Verlauf der Suche nach Schlüssel 16 in der randomisierten Skipliste von Abbildung 2.2

beliebiger Container mindestens die Höhe i hat, ist gleich der Wahrscheinlichkeit, i Mal hintereinander mit einer fairen Münze „Kopf" zu werfen (wir beginnen bei Höhe 0). Dies ergibt eine Wahrscheinlichkeit von $\frac{1}{2^i}$. Also ist die Wahrscheinlichkeit, dass es mindestens einen Container mit Höhe i in der Skipliste gibt, höchstens

$$P_i \leq \frac{n}{2^i} .$$

Das liegt daran, dass die Wahrscheinlichkeit für das Auftreten mindestens eines Ereignisses aus einer Menge höchstens so groß ist wie die Summe der Wahrscheinlichkeiten aller einzelnen Ereignisse.

Nun ist die Höhe h einer Skipliste aber immer gleich der Höhe ihres höchsten Containers. Also ist die Wahrscheinlichkeit, dass eine Skipliste höher als i ist, gleich der Wahrscheinlichkeit, dass es mindestens einen Container mit Höhe $i+1$ gibt, also höchstens P_{i+1}. Zum Beispiel ist die Wahrscheinlichkeit für $h > 3\log n - 1$ höchstens

$$P(h > 3\log n - 1) = P_{3\log n} = \frac{n}{2^{3\log n}} = \frac{n}{n^3} = \frac{1}{n^2} .$$

Allgemeiner gesagt ist die Wahrscheinlichkeit, dass für eine gegebene Konstante $c > 1$ die Höhe einer Skipliste größer als $c\log n - 1$ ist, höchstens $1/n^{c-1}$. Deshalb ist die Höhe einer Skipliste mit großer Wahrscheinlichkeit $O(\log n)$.

Bei einer Suche in einer Skipliste führen wir zwei verschiedene Operationen durch:

1. Wir gehen von einem Container zum folgenden auf dem gleichen Niveau.
2. Wir bleiben im selben Container und gehen ein Niveau hinunter.

Da die Höhe der Skipliste mit großer Wahrscheinlichkeit $O(\log n)$ ist, führen wir mit großer Wahrscheinlichkeit $O(\log n)$ Schritte vom zweiten Typ durch.

Nun müssen wir noch die Anzahl der Schritte vom ersten Typ abschätzen. Sei n_i die Anzahl der Container, die wir auf Niveau i durchlaufen. Keiner dieser Container (außer dem ersten) kann eine Höhe größer als i haben. Andernfalls hätten wir ihn schon auf einem höheren Niveau erreicht. Also ist die Wahrscheinlichkeit, dass wir einen Container auf Niveau i durchlaufen, gerade $\frac{1}{2}$, denn mit Wahrscheinlichkeit $\frac{1}{2}$ wäre der Container noch mindestens ein Niveau höher. Daher ist der Erwartungswert von n_i gleich der erwarteten Anzahl von Münzwürfen, bevor wir zum ersten Mal „Kopf" werfen. „Kopf" entspricht dabei dem Ereignis, dass wir einen Container erreichen, der höher ist als i. Dieser muss deshalb einen zu großen Schlüssel haben, weshalb wir einen Level absteigen.

Der Erwartungswert der Anzahl der Würfe, bis zum ersten Mal „Kopf" fällt, ist 2. Also ist der Erwartungswert für die Anzahl der Container, die wir auf einem Niveau durchlaufen $O(1)$. Da die Höhe der Skipliste mit großer Wahrscheinlichkeit $O(\log n)$ ist, folgt für die Suche ebenfalls eine erwartete Laufzeit von $O(\log n)$.

Auf ähnliche Weise kann man zeigen, dass auch Einfügen und Löschen eine erwartete Laufzeit von $O(\log n)$ haben. Somit haben also alle Operationen eine erwartete logarithmische Laufzeit.

Weil die Implementierung eine Skipliste einfacher ist als jene der meisten anderen Datenstrukturen zur Lösung des Wörterbuchproblems mit garantiert logarithmischer worst-case Laufzeit, wird diese Datenstruktur auch in der Praxis gerne verwendet.

2.4 Weiterführende Literatur

Eine kurze Einführung in das Themengebiet der randomisierten Algorithmen finden Sie in dem Artikel „Randomisierte Algorithmen" von T. Roos im Buch „Prinzipien des Algorithmenentwurfs" von T. Ottmann, Spektrum Akademischer Verlag, 1998, siehe Literaturliste (4), das auch Vorlage für diesen Teil des Skriptums war.

Als Basis zu unserer Beschreibung der Skiplisten wurde die vierten Auflage von „Algorithmen und Datenstrukturen" von Ottmann und Widmayer, Spektrum Akademischer Verlag, herangezogen.

Für weitergehende Literatur empfehlen wir das Buch von R. Motwani und P. Raghavan, „Randomized Algorithms", Cambridge University Press, 1995.

Kapitel 3

Geometrische Algorithmen

In der Praxis kommt es immer wieder vor, dass geometrische Daten gegeben sind, d.h. Daten, die gewisse Koordinaten im d-dimensionalen Raum besitzen. Denken Sie z.B. an Städte auf einer Landkarte oder auch an Komponenten im Chip-Design. Geometrische Algorithmen, auch „Algorithmische Geometrie" oder „Computational Geometry" genannt, nutzen die geometrischen Eigenschaften der Daten aus, um effiziente Algorithmen zu erhalten. Anwendungsgebiete liegen in der Bildverarbeitung, Computergraphik, Geographie, im CAD, oder auch im Chip-Layout.

Ein grundlegendes Prinzip im Bereich der geometrischen Algorithmen ist das *Scan-Line Prinzip*, das wir im folgenden Abschnitt betrachten. Ein weiteres wichtiges Problemfeld ist die Bereichssuche, die im darauffolgenden Abschnitt behandelt wird.

3.1 Scan-Line Prinzip

Das Scan-Line Prinzip ist ein wichtiges Paradigma im Bereich der geometrischen Algorithmen. Es ist immer dann anwendbar, wenn eine Menge von Objekten auf einer 2-dimensionalen Fläche gegeben ist.

Die grundlegende Idee ist, eine vertikale Linie von links nach rechts über die Objektmenge zu führen, um dadurch das zweidimensionale (statische) Problem in eine dynamische Folge von eindimensionalen Problemen umzuwandeln.

Die *Scan-Line* (oder auch *Sweep-Line*) L teilt zu jeder Zeit die Objekte ein in

- tote Objekte, die vollständig links von L liegen;
- aktive Objekte, die gegenwärtig von L geschnitten werden;

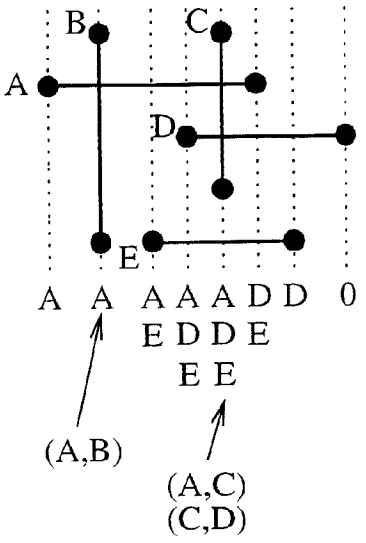


Abbildung 3.1: Eine Probleminstanz für den Schnitt von iso-orientierten Liniensegmenten

- inaktive Objekte, die künftig von L geschnitten werden.

Wir betrachten im Folgenden das Schnittproblem für Liniensegmente, in dem es darum geht, einen oder alle Schnittpunkte von Liniensegmenten zu bestimmen. Zunächst diskutieren wir den einfacheren Fall der iso-orientierten Liniensegmente.

3.1.1 Das Schnittproblem für iso-orientierte Liniensegmente

Bei diesem Problem geht es darum, alle Schnittpunkte einer Menge von senkrechten und horizontalen Geradensegmenten zu bestimmen.

Gegeben: Menge $S = \{s_1, \ldots, s_n\}$ von insgesamt n vertikalen und horizontalen Liniensegmenten in der Ebene
Gesucht: alle Paare sich schneidender Segmente in S

Ein Beispiel für eine solche Problemstellung finden Sie in Abbildung 3.1 (die Buchstaben unter der Zeichnung werden später erklärt). Eine naive Methode zur Lösung des Problems wäre es, alle Paare von Segmenten auf Schnitt zu testen. Dies ergäbe eine Gesamtlaufzeit $\Theta(n^2)$. Tatsächlich gibt es Probleminstanzen, für die keine schnellere Lösung existiert, weil es $\Theta(n^2)$ Schnittpunkte gibt. Wenn allerdings die Anzahl der Schnittpunkte klein ist und nicht quadratisch in n wächst, dann liefert der Scan-Line-Algorithmus eine bessere Laufzeit.

Wir machen eine vereinfachende Annahme: Alle Anfangs- und Endpunkte horizontaler

Segmente und alle vertikalen Segmente haben paarweise verschiedene x-Koordinaten und alle horizontale Segmente haben paarweise verschiedene y-Koordinaten .

Algorithmus 12 verwendet das Scan-Line Prinzip für die Lösung unseres Problems. Er nutzt dabei folgende Beobachtungen aus:

- Trifft man mit der Scan-Line L auf ein vertikales Segment s, so kann s nur Schnittpunkte mit den gerade aktiven horizontalen Segmenten haben.
- Man muss die Scan-Line nicht kontinuierlich über die Fläche führen. Es genügt, sie jeweils an Haltepunkten zu beobachten. Haltepunkte sind die x-Koordinaten der Anfangs- und Endpunkte horizontaler Segmente und die x-Koordinaten vertikaler Segmente.

Algorithmus 12 Scan-Line für Schnitt iso-orientierter Liniensegmente

```
1:  Q: Menge der Haltepunkte in aufsteigender x-Reihenfolge;
2:  L = ∅ ; // Menge der aktiven Segmente, aufsteigend nach y sortiert
3:  solange Q ≠ ∅ {
4:     p: nächster Haltepunkt von Q;
5:     Entferne p aus Q;
6:     falls p ist linker Endpunkt eines horizontalen Segments s dann {
7:        Füge s in L ein;
8:     } sonst {
9:        falls p ist rechter Endpunkt eines horizontalen Segments s dann {
10:          Entferne s aus L;
11:       } sonst {
12:          // p ist x-Wert eines vertikalen Segments s mit unterem Endpunkt (p, y_u) und
             oberem Endpunkt (p, y_o)
13:          Bestimme alle horizontalen Segmente t aus L, deren y-Koordinaten y(t) im
             Bereich [y_u, y_o] liegen, und gib (s, t) als Paar sich schneidender Segmente aus;
14:       }
15:    }
16: }
```

Die Spalten von Buchstaben unter den Haltepunkten in Abbildung 3.1 geben jeweils den Inhalt der Menge L an. Darunter finden Sie die von dem Algorithmus berechneten Paare von sich kreuzenden Segmenten mit Pfeilen zu den Haltepunkten, bei denen sie ausgegeben werden.

Allerdings lässt der Algorithmus noch die Realisierung der geordneten Menge L offen. Hierfür wird eine Datenstruktur benötigt, die die folgenden Operationen effizient ausführen kann. Dies sind:

- Einfügen eines neuen Elements
- Entfernen eines Elements
- Bestimmen aller Elemente, die in einen gegebenen Bereich $[y_u, y_o]$ fallen *(Bereichsabfrage)*

Eine mögliche Lösung bieten balancierte Binärbäume (z.B. AVL-Bäume). Damit können die Einfüge- und Entfernungs-Operationen in einer Zeit $O(\log|L|)$ durchgeführt werden. Die Bereichsabfrage in Zeile 12 von Algorithmus 12 könnte dann folgendermaßen realisiert werden:

1. Suche in T den Knoten p mit kleinstem Schlüssel $\geq y_u$ und gib p als Segment aus, falls sein Schlüssel $\leq y_o$.
2. Laufe T ab Knoten p in Inorder-Reihenfolge durch, bis Knoten q mit Schlüssel $> y_o$ erreicht wird, gib alle dabei durchlaufenen Segmente außer q aus.

Algorithmus 13 zeigt die Realisierung des ersten Schritts. Die Realisierung des zweiten Schritts wird hier nicht ausgeführt, weil es sich um eine schlichte Inorder-Durchmusterung handelt.

Algorithmus 13 Suche Knoten p mit kleinstem Schlüssel $\geq x$

```
1: p = root; q = NULL;
2: solange (p ≠ NULL&& p.key ≠ x) {
3:    q = p;
4:    falls x < p.key dann {
5:       p = p.leftchild;
6:    } sonst {
7:       p = p.rightchild;
8:    }
9: }
10: // Schlüssel == x gefunden
11: falls p ≠ NULLdann Retourniere p;
12: // leerer Baum
13: falls root == NULLdann Retourniere NULL;
14: // Baum besteht aus einem Knoten
15: falls q == root dann Retourniere NULL bzw. root abhängig von root.key;
16: falls x < q.key dann Retourniere q;    sonst Retourniere Successor(q);
```

Analyse der Laufzeit

Wir betrachten zunächst die Laufzeit der Bereichssuche: Ist r die Anzahl der Elemente im Bereich $[y_u, y_o]$, so ist die Laufzeit für eine Bereichsabfrage $O(\log n + r)$, bestehend aus $O(\log n)$ für den ersten Schritt der Bereichssuche und $O(r)$ für den zweiten. Somit ergibt sich für den Scan-Line Algorithmus eine Gesamtlaufzeit von $O(n \log n + R)$, wobei R die Anzahl der sich schneidenden Segmente ist. Der Platzverbrauch ist hingegen nur linear: $O(n)$.

Bemerkungen

1. Das Scan-Line Verfahren ist dem naiven Verfahren überlegen, wenn R schwächer als quadratisch mit der Anzahl der Segmente wächst.

2. Man kann zeigen, dass im schlechtesten Fall $\Omega(n \log n + R)$ Schritte erforderlich sind, um das Schnittproblem zu lösen. Somit ist das hier beschriebene Verfahren *worst case optimal*.

3.1.2 Schnitt von allgemeinen Liniensegmenten

Bei diesem Problem ist eine Menge von Liniensegmenten in allgemeiner Lage gegeben und wieder sollen die Paare sich schneidender Segmente berechnet werden. Im Gegensatz zum vorigen Problem sind nun nicht nur waagerechte und senkrechte Segmente erlaubt.

Gegeben: Menge von n Liniensegmenten $s_1, \ldots, s_n$ in der Ebene
Gesucht: Menge echter Schnittpunkte (echte Schnittpunkte sind Kreuzungen von Segmenten, die keine Endpunkte sind)

Um den Algorithmus nicht durch viele Fallunterscheidungen zu komplizieren, machen wir auch hier wieder vereinfachende Annahmen, die man auch zusammengefasst *allgemeine Lage* der Segmente nennt:

1. Der Schnitt zweier Segmente ist stets leer oder genau ein Punkt.

2. Es schneiden sich nie mehr als zwei Segmente in einem Punkt.

3. Alle Endpunkte und Schnittpunkte haben paarweise verschiedene x-Koordinaten.

Abbildung 3.2 zeigt eine Menge von Geradensegmenten in allgemeiner Lage. Wir benutzen auch hier wieder einen Scan-Line-Algorithmus mit einer senkrechten Scan-Line, die sich von links nach rechts über die Fläche bewegt.

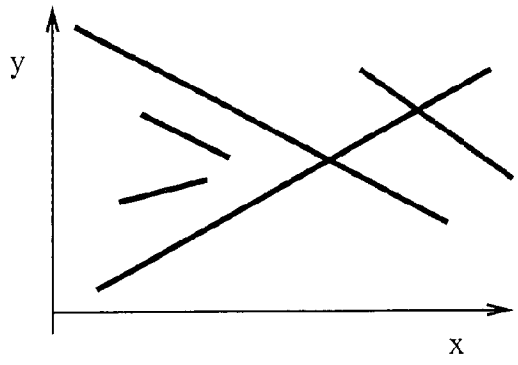


Abbildung 3.2: Eine Menge von Geradensegmenten in allgemeiner Lage

Beobachtung 1: Wenn sich das Segment s_i mit dem Segment s_j bei x-Koordinate x_s schneidet, dann sind s_i und s_j bei x-Koordinate $x_s - \varepsilon$ Nachbarn auf der Scan-Line für ein hinreichend kleines aber positives ε. Daraus ergibt sich, dass es genügt, auf der Scan-Line benachbarte Segmente auf Schnitt zu testen.

Beobachtung 2: Am Schnittpunkt zweier Segmente vertauscht sich die Reihenfolge der betroffenen Segmente auf der Scan-Line. Daraus ergeben sich folgende Konsequenzen:

1. Auch Schnittpunkte müssen Ereignisse sein.
2. Schnittpunkte müssen während des Scans in die Ereignisdatenstruktur eingefügt werden.

Wir definieren ein Ereignis als einen Haltepunkt der Scan-Line. Es gibt folgende drei Typen von Ereignissen:

1. Anfang eines Segments;
2. Ende eines Segments;
3. Schnitt zweier Segmente.

Zusammen mit den jeweiligen x-Koordinaten merken wir uns auch die Indizes der betroffenen Segmente. Wir führen eine *Ereignisdatenstruktur* (*ES*) mit den folgenden Operationen ein:

1. *NächstesEreignis()*: liefert das Ereignis mit der kleinsten x-Koordinate aus *ES* und löscht es.

2. *FügeEin(Ereignis)*: fügt ein neues Ereignis in *ES* ein.

Diese Datenstruktur kann z.B. durch einen balancierten Baum oder durch einen Heap implementiert werden. Bei beiden Implementierungen ist die Laufzeit der Operationen logarithmisch in der Größe von *ES*. Die Initialisierung erfolgt mit den Anfangs- und Endpunkten der Liniensegmente, sortiert nach ihren x-Koordinaten.

Weiterhin benötigen wir eine Scan-Line-Status-Struktur (*SSS*). Diese speichert die Menge der aktiven Segmente, geordnet nach der y-Koordinate ihres momentanen Schnittpunkts mit der Scan-Line. Die Datenstruktur stellt folgende Operationen zur Verfügung:

1. *FügeEin(Segment)*: Fügt ein Segment in *SSS* ein
2. *Entferne(Segment)*: Entfernt ein Segment aus *SSS*
3. *Vorg(Segment)*: Liefert den Vorgänger (oberhalb) eines Segments auf der Scan-Line
4. *Nachf(Segment)*: Liefert den Nachfolger (unterhalb) eines Segments auf der Scan-Line
5. *Vertausche(Segment1, Segment2)*: Vertauscht die Reihenfolge zweier Segmente auf der Scan-Line

Diese Struktur kann z.B. durch einen blattorientierten balancierten binären Suchbaum mit Verkettung der Blätter (z.B. B-Baum) realisiert werden und wird mit der leeren Menge initialisiert.

Algorithmus 14 zeigt unseren Scan-Line Algorithmus zur Lösung des Problems.

Die Funktion *TesteSchnittErzeugeEreignis(Segment1,Segment2)* berechnet die x-Koordinate des Schnittpunkts zweier Segmente und erzeugt ein entsprechendes Ereignis, falls sich die Segmente schneiden und der Schnittpunkt sich rechts von der aktuellen Position des Scanlines befindet. Ansonsten wird ein leeres Ereignis erzeugt. Ein Schnitt-Ereignis speichert die Indizes der beiden betroffenen Segmente als *SegmentU* (jenes Segment, das vor dem Schnitt unterhalb des zweiten war) und *SegmentO* (jenes Segment, das vor dem Schnitt oben war).

Analyse der Laufzeit

Der Schnitt von Liniensegmenten kann in Zeit $O((n+k)\log n)$ implementiert werden, wobei n die Anzahl der Segmente und k die Anzahl der Schnittpunkte bezeichnet. Der benötigte Speicherplatz ist $O(n^2)$.

Algorithmus 14 Scan-Line für das Segment-Schnitt Problem in allgemeiner Lage

```
Initialisiere ES und SSS;
Sortiere die 2n Endpunkte aufsteigend nach x-Koordinate;
Speichere resultierende Ereignisse in ES;
solange ES nicht leer {
  E = ES.NächstesEreignis();
  falls E ist Segment Anfang dann {
    SSS.FügeEin(E.Segment);
    VS = SSS.Vorg(E.Segment);
    E' = TesteSchnittErzeugeEreignis(E.Segment,VS);
    falls E' ≠ ∅ und E' ∉ ES dann {
      ES.FügeEin(E');
    }
    NS = SSS.Nachf(E.Segment);
    E' = TesteSchnittErzeugeEreignis(E.Segment,NS);
    falls E' ≠ ∅ und E' ∉ ES dann {
      ES.FügeEin(E');
    }
  }
  falls E ist Segment Ende dann {
    VS = SSS.Vorg(E.Segment);
    NS = SSS.Nachf(E.Segment);
    SSS.Entferne(E.Segment);
    E' = TesteSchnittErzeugeEreignis(VS,NS)
    falls E' ≠ ∅ und E' ∉ ES dann {
      ES.FügeEin(E');
    }
  }
  falls E ist Schnittpunkt dann {
    Gib E.SegmentO und E.SegmentU aus;
    SSS.Vertausche(E.SegmentO,E.SegmentU);
    VS= SSS.Vorg(E.SegmentU);
    E' =TesteSchnittErzeugeEreignis(E.SegmentU,VS);
    falls E' ≠ ∅ und E' ∉ ES dann {
      ES.FügeEin(E');
    }
    NS = SSS.Nachf(E.SegmentO);
    E' = TesteSchnittErzeugeEreignis(E.SegmentO,NS);
    falls E' ≠ ∅ und E' ∉ ES dann {
      ES.FügeEin(E');
    }
  }
}
```

Das Sortieren der $2n$ Endpunkte nach ihrer x-Koordinate erfolgt in Zeit $O(n \log n)$. Insgesamt gibt es $2n + k$ verschiedene Ereignisse, von denen sich nie mehr als $O(n^2)$ Ereignisse in *ES* befinden (weil es maximal $O(n^2)$ Schnittpunkte gibt). Der Zugriff auf *ES* ist also in Zeit $O(\log n^2) = O(\log n)$ möglich. In *SSS* befinden sich nie mehr als n Elemente, weshalb ein Zugriff auf diese Datenstruktur auch nur Zeit $O(\log n)$ benötigt.

Die Schleife wird maximal $2n + k$ mal durchlaufen, und bei jedem Durchlauf werden höchstens fünf Operationen auf *ES* und *SSS* durchgeführt. Der Speicherplatzbedarf ist im schlimmsten Fall quadratisch, weil die Kreuzungen in der Ereignisdatenstruktur abgelegt werden und es quadratisch viele Kreuzungen geben kann.

3.2 Mehrdimensionale Bereichssuche

Bei der mehrdimensionalen Bereichssuche geht es darum, effizient die Menge all der Punkte in einem mehrdimensionalen Raum zu finden, die innerhalb eines Suchbereichs liegen. Die Menge aller Punkte ist dabei schon im Vorhinein bekannt, und es geht darum, mehrere Bereichsanfragen möglichst schnell zu beantworten.

Die zentrale Idee ist dabei, die Menge aller Punkte in einer Datenstruktur abzulegen, welche das Ausführen einer Bereichsabfrage beschleunigt. Dies zahlt sich aus, wenn auf der gleichen Punktmenge mehrere Anfragen mit unterschiedlichen Suchbereichen gestellt werden. Die Problemstellung lautet wie folgt:

Gegeben: k-dimensionaler rechteckiger Bereich D und n Punkte im k-dimensionalen Raum.
Gesucht: Finde alle Punkte, die in D liegen.

Beispiele:

- $k = 1$: Aufzählen aller Elemente in einer Folge mit Schlüssel zwischen a und b
- $k = 2$: Aufzählen aller Städte in einem Quadrat mit 100 km Seitenlänge und dem Mittelpunkt Wien
- $k > 2$: Datenbankabfragen, z.B. Aufzählen aller Personen, die
  - Informatik studiert haben,
  - zwischen 25 und 39 Jahre alt sind,
  - ein jährliches Einkommen zwischen 30.000 EUR und 50.000 EUR haben, und
  - kein Handy besitzen.

Satz 3.1 *Eindimensionale Bereichssuche kann mit $O(n \log n)$ Schritten für die Vorverarbeitung und $O(R + \log n)$ Schritten für die Bereichssuche ausgeführt werde, wobei R die Anzahl der Punkte ist, die tatsächlich im Bereich liegen.*

Beweis: Man legt die Folge in einem balancierten Suchbaum ab, z.B. in einem AVL-Baum. □

Wir wollen für höhere Dimensionen nun ein ähnlich gutes Verfahren entwickeln. Eine naive Lösung des Problems für höhere Dimensionen ist das sequentielle Durchlaufen aller Punkte, wobei man jeweils testet, ob der gerade betrachtete Punkt im Bereich D liegt. Dies bedeutet einen Aufwand von $O(n)$ Schritten für jede Bereichsabfrage.

Liegt bei jeder Anfrage mindestens ein konstanter Anteil aller Punkte im Suchbereich D, so ist dieses naive Verfahren asymptotisch optimal. Liefert aber jede Suchanfrage nur eine kleine konstante Anzahl von Punkten als Ergebnis, so ist das Verfahren sehr ineffizient. Wir wollen einen *outputsensitiven* Algorithmus, bei dem die Laufzeit von der Anzahl der im Suchbereich liegenden Punkte und somit von der Größe der Ausgabe abhängig ist.

Man könnte nun ein regelmäßiges Gitter über die Menge aller Punkte legen, um die Anfragen schneller zu bearbeiten. Dies liefert gute Ergebnisse, wenn die Punkte gleichmäßig verteilt sind. Sind die Punkte aber in bestimmten Bereichen des Raumes konzentriert, so gibt es viele leere Gitterzellen, während andere Gitterzellen sehr viele Punkte enthalten. Das folgende Verfahren umgeht dieses Problem.

3.2.1 Zweidimensionale Bäume

Die Idee ist, den zweidimensionalen Raum aufzuteilen, ähnlich wie wir den eindimensionalen Raum mittels binärer Suchbäume aufgeteilt haben. Anders als bei den binären Suchbäumen verwenden wir hier alternierend die x- und y-Koordinaten als Schlüssel.

Wir konstruieren also einen binären Baum, der eine Aufteilung der Ebene repräsentiert. Dabei enthält jeder Knoten des Baumes einen der n Punkte. Verwenden wir an einem Knoten v, der den Punkt p repräsentiert, die y-Koordinate zur Bestimmung des linken und rechten Teilbaums, so enthält der linke Teilbaum von v jene Punkte, die unterhalb von p liegen, und der rechte Teilbaum jene Punkte, die oberhalb von p liegen. Verwenden wir aber die x-Koordinate, so enthält der linke Teilbaum von v die Punkte links von p und der rechte Teilbaum die Punkte rechts von p.

Sind die Teilbäume von v durch die x-Koordinate bestimmt worden, so werden die Teilbäume der Kinder von v durch die y-Koordinate bestimmt. Sind aber die Teilbäume von v durch die y-Koordinate bestimmt worden, so werden die Teilbäume der Kinder von

v durch die x-Koordinate bestimmt. Es liegt also jeder Punkt der gegebenen Punktmenge auf einem vertikalen oder horizontalen Segment, welches die Zerlegung definiert, die an dem entsprechenden Knoten im binären Baum vorgenommen wurde. Abbildung 3.3 zeigt ein Beispiel für diese Aufteilung der Ebene.

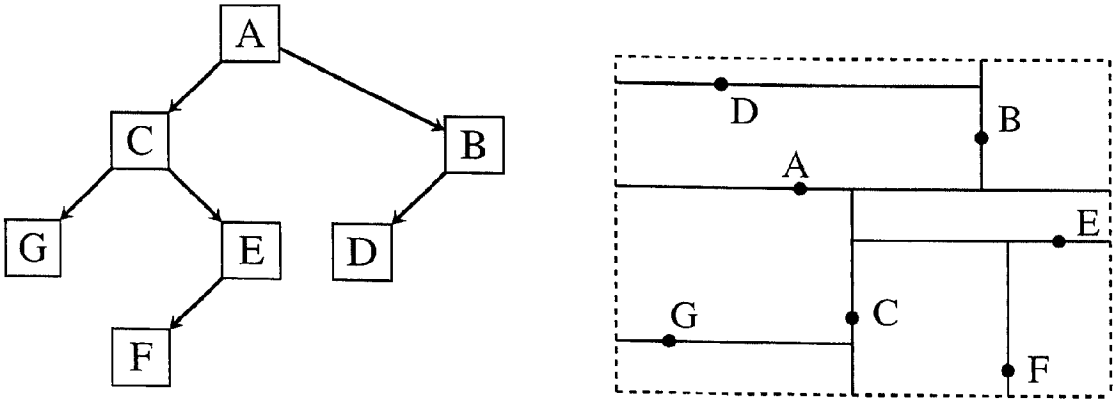


Abbildung 3.3: Ein 2-D-Suchbaum und die dadurch definierte Zerlegung der Ebene

Die Suche in einem 2-D-Baum funktioniert ganz ähnlich wie die Suche in einem binären Suchbaum. Allerdings muss man bei jedem durchlaufenen Knoten darauf achten, ob die Teilbäume des Knotens durch die x- oder die y-Koordinate bestimmt sind. Außerdem kann es sein, dass die Suche nach den in einem Bereich enthaltenen Punkten in beiden Teilbäumen eines Knotens durchgeführt werden muss.

Algorithmus 15 stellt die Bereichssuche in einem 2-D-Baum als Funktion dar. Der Aufruf erfolgt in der Form $Bereichssuche(r, vert, D)$, wobei r die Wurzel des Baums ist und wir davon ausgehen, dass die Kinder der Wurzel durch die y-Koordinate bestimmt sind.

Es stellt sich nun noch die Frage, wie man einen solchen Baum effizient aufbauen kann, damit eine schnelle Abarbeitung der Suchanfragen möglich ist. Es muss sich um einen balancierten Baum handeln, damit für kleine Ergebnismengen logarithmische Laufzeit in n erreicht werden kann. Wir erreichen dies, indem wir stets am Median–Punkt der aktuellen Folge aufteilen. Dadurch wird garantiert, dass in den neu entstehenden Teilen jeweils gleich viele Punkte enthalten sind, wobei es zu einem maximalen Unterschied von einem Punkt kommen kann. Wir gehen dazu wie folgt vor:

1. Wir sortieren alle Punkte sowohl nach der x- als auch nach der y-Koordinate. Dadurch erhalten wir zwei sortierte Folgen X und Y.
2. Wir teilen die Folge Y am Median–Element und machen dieses zur Wurzel des Baums. Es entstehen zwei neue Folgen Y_1 und Y_2. Nun teilen wir auch die Folge X in zwei

Algorithmus 15 *Bereichssuche(Knoten p, Richtung d,Bereich D)*

```
falls p ≠ NULL dann {
  falls d == vertikal dann {
    min = D.y1;
    max = D.y2;
    coord = p.y;
    rNeu = horizontal;
  } sonst {
    min = D.x1;
    max = D.x2;
    coord = p.x;
    rNeu = vertikal;
  }
  falls p ∈ D dann Ausgabe von p;
  falls min ≤ coord dann Bereichssuche(p.left, rNeu, D);
  falls coord ≤ max dann Bereichssuche(p.right, rNeu, D);
}
```

Teile, sodass Folge X_1 die selben Punkte wie Y_1 enthält und X_2 die selben Punkte wie Y_2 (natürlich jeweils nach x-Koordinate sortiert).

3. Wir führen nun dasselbe rekursiv mit den Folgen X_i und Y_i für $i \in \{1, 2\}$ durch, wobei wir nun aber die X-Folgen am Median–Element aufteilen. Wir teilen immer weiter abwechselnd am Median der X- und Y-Folgen auf, bis die Folgen jeweils nur noch aus einem Punkt bestehen. Diese Punkte sind dann die Blätter des Baumes.

Am Beispiel der Punkte in der Punktmenge von Abbildung 3.3 geht der Aufbau des Baumes wie folgt vonstatten. Sortiert man erst alle Punkte in den Folgen X und Y, so erhält man:

X: G D A C B F **E**
Y: F G C **E** A B D

Das Median–Element der Y-Folge ist **fett** gedruckt. Nach der ersten Unterteilung sehen die Folgen so aus:

X_1: G **C** F
Y_1: F G **C**

X_2: D **A** B
Y_2: **A** B D

Im nächsten Schritt wird dann an den **fett** gedruckten Median–Elementen der X-Folgen aufgeteilt. Es ist klar, dass dieser Aufteilungsschritt in Zeit $O(l)$ durchgeführt werden kann, wenn l die Länge der Folge vor der Teilung ist.

Algorithmus 16 zeigt eine Funktion, die einen balancierten 2-D-Baum in Zeit $O(n \log n)$ aufbaut. Es wird davon ausgegangen, dass das Feld X die Menge aller Punkte sortiert nach x-Koordinate enthält und Feld Y die Menge aller Punkte sortiert nach y-Koordinate.

Algorithmus 16 *2D-Aufbau(l,r,knoten,Xdir)*

Ausgabe: Konstruktion eines balancierten 2-D-Baums

```
1: falls l ≤ r dann {
2:   m = ⌈(l+r)/2⌉;
3:   falls Xdir == true dann {
4:     knoten.eintrag = X[m];
5:     Partitioniere Feld(Y, l, r, m);
6:   } sonst {
7:     knoten.eintrag = Y[m];
8:     Partitioniere Feld(X, l, r, m);
9:   }
10:  links = neuer linker Nachfolgeknoten;
11:  rechts = neuer rechter Nachfolgeknoten;
12:  Aufbau(l, m − 1, links, !Xdir);
13:  Aufbau(m + 1, r, rechts, !Xdir);
14: }
```

Da das Aufteilen einer Folge in linearer Zeit möglich ist, ergibt sich für die Laufzeit des Algorithmus die folgende Rekursionsformel:

$$T(n) = 2T(\frac{n}{2}) + cn$$

Diese Formel haben wir schon bei der Funktion Mergesort gesehen, und wie schon dort erläutert ist die Lösung der Gleichung $\Theta(n \log n)$. Der Aufwand für eine Bereichsabfrage ist $O(\sqrt{n} + R)$, wobei R die Anzahl der Punkte im Bereich D ist. Der Beweis wird hier wegen seiner Komplexität nicht ausgeführt. Es ist aber deutlich zu sehen, dass der Aufwand der Bereichsabfrage für kleine R deutlich geringer ist als bei der naiven Methode, bei der einfach alle Punkte geprüft werden.

3.2.2 Höhere Dimensionen

Man kann die obigen Algorithmen recht einfach an höhere Dimensionen anpassen, indem man reihum nach den Dimensionen aufteilt (beim Baumaufbau) bzw. entscheidet, welchen Teilbaum man betrachtet (bei der Bereichssuche). Man kann dann zeigen, dass der Aufbau des entsprechenden Baumes Zeit $\Theta(kn \log n)$ und die Bereichssuche Zeit $O(kn^{1-\frac{1}{k}} + R)$ benötigt, wobei k die Dimension ist.

3.3 Weiterführende Literatur

Die beiden hier aufgeführten Problemfelder (Scan-Line-Algorithmen und mehrdimensionale Bereichssuche) sind besonders gut beschrieben in dem Buch „Algorithmen in Java“ von Robert Sedgewick, siehe Literaturliste (1).

Kapitel 4

Algorithmen für große Datenmengen

In den letzten Jahren werden immer größere Datenmengen gesammelt und verarbeitet. Denken wir an die riesigen Mengen von DNA-Kodierungen, die jedes Jahr in der Molekularbiologie gesammelt werden, oder die Daten, die jeden Tag von Satelliten zur Erde gesendet werden. Andere Schlagwörter sind: Wissensdatenbanken, Internet-Daten oder Verlagsdatenbanken. Die Datenmengen sind hierbei so riesengroß, dass sie nicht mehr im Arbeitsspeicher Platz haben. Deswegen liegen diese auf Externspeichern, wie z.B. auf magnetischen Festplatten. Die Algorithmen, die wir bisher kennengelernt haben, wurden für das RAM-Modell entwickelt. Dabei ist die Annahme, dass es unbegrenzten Speicher gibt und jeder Speicherzugriff gleich teuer ist.

Abbildung 4.1 zeigt das hierarchische Speichermodell moderner Computer und typische Kapazitäten der einzelnen Speichereinheiten. Hauptspeicherzugriffe sind ungefähr 100 Mal langsamer als Cache-Zugriffe, und Externspeicherzugriffe (im Folgenden I/O genannt) sind ungefähr 1000 mal langsamer als Hauptspeicherzugriffe und somit bis zu 100.000 mal langsamer als Cache-Zugriffe.

Das Problem ist aktueller denn je, denn:

(1) Die Geschwindigkeit der Prozessoren verbessert sich um ca. 30%-50% im Jahr; die Geschwindigkeit des Speichers hingegen verbessert sich nur um 7%-10% pro Jahr.

(2) Es gibt vermehrt wichtige large-scale Anwendungen, wie z.B. Geographische Informationssysteme, Bioinformatik (DNA, Datenbank), Suchen im Web.

Hierzu ein aktuelles Zitat aus der Informatik-Community: "One of the few resources increasing faster than the speed of computer hardware is the amount of data to be processed."[1]

[1] Aus dem Call-for-Papers für die IEEE InfoVis 2003

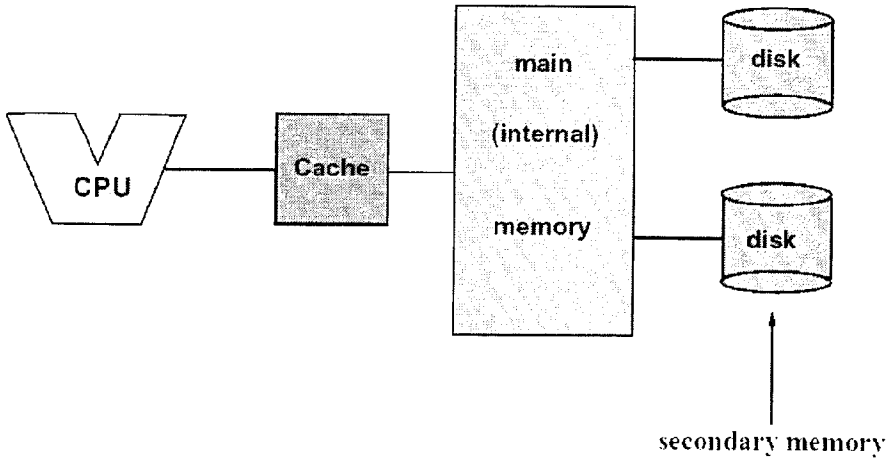


Abbildung 4.1: Hierarchisches Speichermodell moderner Computer

Ein Speicherzugriff vom Arbeitsspeicher in den Cache bzw. vom Externspeicher (Sekundärspeicher) in den Arbeitsspeicher liefert jeweils einen ganzen **Block** von Daten zurück. Dies wird jedoch von den klassischen Algorithmen nicht beachtet. Wendet man die klassischen Algorithmen auf große Datenmengen an, dann erhält man meist keine Lokalität bei Speicherzugriffen, was zu sehr vielen unnötigen Speicherzugriffen führt. Das folgende Beispiel soll die Problematik verdeutlichen.

Beispiel: Wir betrachten die folgenden Programmstücke. Dabei seien B und C Felder der Größe N (N sehr groß), die als Indexarrays benutzt werden, um das Feld A durchzulaufen. `RandomPermute(`B`)` nimmt das Feld B und permutiert die Inhalte zufällig. D.h. nach dem Aufruf enthält das Feld C die gleichen Inhalte wie das Feld B, nur in einer zufälligen Reihenfolge.

(1) Für $i = 0$ bis $N - 1$ tue: $B[i] = i$

(2) $C =$ RandomPermute(B);

(3) Für $i = 0$ bis $N - 1$ tue: $A[B[i]] = A[B[i]] + 1$;

(4) Für $i = 0$ bis $N - 1$ tue: $A[C[i]] = A[C[i]] + 1$;

Die beiden Programmstücke (3) und (4) machen genau das gleiche: sie laufen jeweils alle Elemente des Feldes A ab und zählen 1 dazu. Allerdings läuft (3) das Feld A linear durch, während (4) das Feld A zufällig durchläuft. Abbildung 4.2 zeigt die Laufzeitunterschiede für die beiden Programmstücke. Die Laufzeittests wurden auf einem Rechner mit CPU 2.40GHz

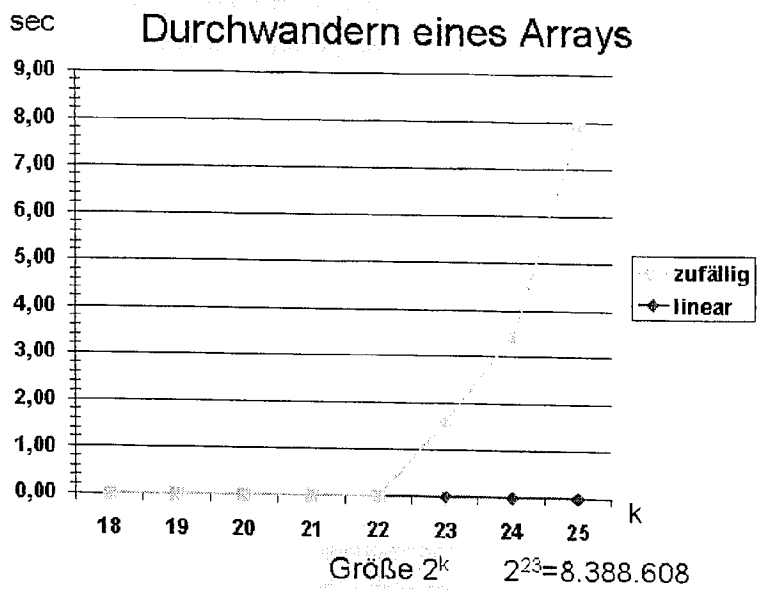


Abbildung 4.2: Laufzeitunterschiede zwischen linearem und zufälligem Durchwandern eines Feldes der Größe 2^k

und Cache-Größe 512 KB ausgeführt. Obwohl die Komplexität beider Programmstücke äquivalent ist, so ist doch die zweite Version deutlich langsamer als die erste. Zum Beispiel ist die Laufzeit für $N = 2^{25} = 33.554.432$ für das lineare Durchwandern 0,39 Sekunden, während sie für das zufällige Durchwandern 7,89 beträgt. Dies ist ein Faktor von 20 (!).

Während man im ersten Programmstück für $N = 34$ Mio. ca. 66 Festplattenzugriffe hat, weil ein Zugriff nicht nur den Inhalt von $A[1]$, sondern auch die Inhalte von $A[2], A[3], \ldots$, in den Cache bringt, sodass die folgenden Additionsaufrufe keine eigenen Zugriffe benötigen, hat man im zweiten Fall tatsächlich ungefähr $N = 34$ Mio. Zugriffe auf den Externspeicher.

Bereits dieses einfache Programmstück zeigt, dass es auch für einfache Schleifen wichtig ist, auf Lokalität der Speicherzugriffe zu achten, wenn man es mit großen Datenmengen zu tun hat.

In diesem Kapitel geht es also um die Entwicklung von Algorithmen und Datenstrukturen, die die Lokalität der Datenzugriffe beachten. In Kapitel 3.1 werden externe Speichermodelle und ein Beispiel eines Externspeicheralgorithmus diskutiert und analysiert. Kapitel 3.2 beschäftigt sich mit sogenannten *Cache-optimalen* Algorithmen.

4.1 Externspeicher-Algorithmen

4.1.1 Virtuelles Speichermanagement des Betriebssystems

Der virtuelle Speicher ist in *Seiten* (*pages*) fester Größe eingeteilt. Diese befinden sich teilweise im Hauptspeicher und teilweise im Sekundärspeicher. Sobald Seiten im Sekundärspeicher angesprochen werden, werden diese in den Hauptspeicher transportiert (*page in*). Im Gegenzug muss dafür ein nicht mehr benötigtes Stück vom Hauptspeicher in den Sekundärspeicher kopiert werden (*page out*). Der Arbeitsspeicher wird als Pool von virtuellen Seiten gesehen, die entweder frei oder belegt sind.

Das Betriebssystem versucht nun durch intelligente Strategien (*caching and prefetching*), den I/O-Engpass zu minimieren. Jedoch sind dies sehr allgemeine Algorithmen, die naturgemäß nicht für das jeweilige Problem zugeschnitten sein können. Wenn nun klassische Algorithmen auf den Sekundärspeicher zugreifen, so geschieht dies meist unstrukturiert, d. h. auf Daten wird "durcheinander" zugegriffen, ohne eine mögliche Lokalität der Zugriffe auszunützen. Das Problem dabei ist, dass die meiste Zeit damit verbracht wird, die Daten zwischen externem und internem Speicher hin- und herzutransportieren.

Die sogenannten *Externspeicheralgorithmen* (*External Memory Algorithms*) sollen dieses Problem lösen. Sie gehen davon aus, dass der Speicher in einen begrenzten Hauptspeicher und in eine gewisse Anzahl von Sekundärspeicherplatten geteilt ist, die jeweils unterschiedliche Zugriffszeiten sowie Zugriffseigenschaften besitzen.

Während ein Zugriff im Hauptspeicher immer jeweils eine Zeiteinheit benötigt und eine Speicherzelle anspricht, geht man davon aus, dass ein Zugriff im Sekundärspeicher deutlich länger benötigt, und jeweils einen zusammenhängenden Block mit den Daten mehrerer Speicherzellen zurückliefert.

4.1.2 Das theoretische Sekundärspeichermodell (EM-Modell)

Das EM-Modell benutzt die folgenden Parameter:

- N= Anzahl der Elemente in der Input-Instanz;
- M= Anzahl der Elemente, die in den Hauptspeicher passen;
- B= Anzahl der Elemente, die in einen Block passen,

wobei $M < N$ und $1 \leq B \leq M/2$.

Das älteste theoretische Sekundärspeichermodell wurde 1988 von Aggerwal und Vitter vorgeschlagen. Es sieht vor, dass ein Rechner aus einer CPU mit einem schnellen Hauptspeicher der Größe M besteht. Der Sekundärspeicher wird als eine Platte (*disk*) modelliert, die durch $P \geq 1$ voneinander unabhängige Zugriffsköpfe (Lese- und Schreibköpfe) angesprochen werden kann. Ein externer Zugriff (I/O) bewegt jeweils B Elemente von der Platte zum Hauptspeicher oder umgekehrt. Falls $P > 1$, können also $P * B$ Elemente in einer I/O-Operation bewegt werden. Dieses Modell ist jedoch nicht praxisnah, weil in der Regel bei Systemen mit mehreren Zugriffsköpfen diese nicht unabhängig voneinander steuerbar sind.

Daher wurde dieses Modell 1994 von Vitter und Shriver so verändert, dass D verschiedene, voneinander unabhängige Festplatten vorhanden sind. Nun ist es kein Problem, auch in der Praxis in einer I/O-Operation $D * B$ Elemente zu bewegen. Das derart modifizierte Modell ist momentan das Standardmodell in diesem Bereich und wird auch *Parallel Disk Modell* genannt. Abb. 4.3 zeigt das *Parallel Disk Modell*.

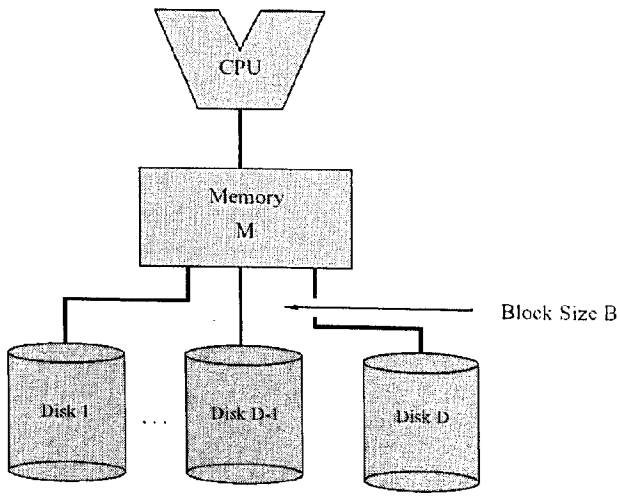


Abbildung 4.3: Das *Parallel Disk* Speichermodell von Vitter und Shriver

Eine weitere Annahme beider EM-Modelle ist, dass Blöcke nicht geteilt werden können und dass Verarbeitungsoperationen nur mit Daten im Hauptspeicher ausführbar sind.

In diesem Modell basiert die Analyse von Algorithmen auf folgenden Parametern:

(i) Anzahl der ausgeführten I/O-Operationen;

(ii) Anzahl der ausgeführten CPU-Operationen (RAM-Modell);

(iii) Anzahl der belegten Blöcke auf dem Sekundärspeicher.

In der Folge gehen wir immer von dem *Parallel Disk Modell* als unserem EM-Modell aus und setzen der Einfachheit halber $D = 1$.

4.1.3 Untere Schranken im EM-Modell

Wir betrachten im Folgenden die Mindestanzahl von notwendigen I/O-Operationen für simple Standardaufgaben (d.h. untere Schranken, $D = 1$).

- Das *Einlesen einer Menge* von N Elementen benötigt mindestens $\Theta(N/B)$ I/O-Operationen.
- Die *Suche in dynamischen Daten* von N Elementen benötigt mindestens $\Theta(logN/logB) = \Theta(log_B N)$ I/O-Operationen.
- Das *Sortieren einer Menge* von N Elementen benötigt mindestens $\Theta(\frac{N}{B} \log_{1+M/B}(1 + N/B))$ I/O-Operationen (ohne Beweis; Literaturtipp für Interessierte: A. Aggarwal und J.S. Vitter, The input/output complexity of sorting and related problems, *Communications of the ACM*, 1116–1127, 1988).

Bemerkung: Das EM-Standardmodell unterscheidet nicht zwischen zufälligen I/O-Plattenzugriffen und sequentiellen I/O-Zugriffen, obwohl letztere in der Praxis deutlich schneller sind. Es gibt EM-Modelle (z.B. von M. Farach, P. Ferragina und S. Muthakristnan, Overcoming the memory bottleneck in suffix tree construction, *Proc. of the 39th Annual Symposium on Foundations of Computer Science*, 174–185, IEEE Computer Society 1998), welche dies mit in Betracht ziehen.

4.2 Prioritätswarteschlangen am Externspeicher

4.2.1 Prioritätswarteschlangen

Eine *Prioritätswarteschlange* (*Priority Queue*) ist eine Datenstruktur, die eine Menge von Elementen speichert, welche jeweils aus einem Tupel *Information* und *Prioritätswert* (Schlüssel, *Key*) bestehen. Diese Datenstruktur unterstützt die folgenden Operationen:

- `Get_Min`: Ausgabe des Elements mit kleinstem Schlüssel
- `Del_Min`: Ausgabe und Entfernung des Elements mit kleinstem Schlüssel aus der Liste

- `Insert`: Einfügen eines neuen Elements in die Warteschlange

Prioritätswarteschlangen tauchen in einer Vielzahl von Anwendungen auf, etwa bei der kombinatorischen Optimierung (z.B. bei Dijkstras Algorithmus zur Bestimmung kürzester Wege), beim Sortieren (z.B. Heapsort), oder bei Scheduling bzw. Simulationen.

Es gibt eine Vielzahl von Realisierungen für Prioritätswarteschlangen. Wir haben z.B. in *Algorithmen und Datenstrukturen 1* einen binären Heap oder auch binäre Suchbäume wie die AVL-Bäume kennengelernt.

Aufgabe: Überlegen Sie sich, wieviel Aufwand die Operationen `Get_Min`, `Del_Min`, und `Insert` jeweils in $theta$-Notation verursachen, wenn Sie für die Realisierung binäre Heaps bzw. binäre Suchbäume verwenden.

Als geeignete Datenstruktur für riesige Datenmengen haben wir bereits die B-Bäume kennengelernt. Im folgenden diskutieren wir die spezielle Externspeicherdatenstruktur *Externe Array-Heaps*. Wie wir sehen werden, sind Externe Array-Heaps deutlich besser als die B-Baum-Datenstruktur zur Realisierung einer Prioritätswarteschlange geeignet.

4.2.2 Externe Array-Heaps

Der externe Array-Heap besteht aus zwei Teilen: einer internen Datenstruktur (im folgenden Heap H genannt) im Arbeitsspeicher und einer externen Datenstruktur, die aus einer Menge von sortierten Feldern (*Arrays*) unterschiedlicher Länge besteht. Diese Felder sind in L Schichten $L_i, 1 \leq i \leq L$ eingeteilt; jede Schicht besteht aus $\mu = cM/B - 1$ Feldern (die im folgenden *Slots* genannt werden) der Länge $l_i = (cM)^i/B^{i-1}$ für $c < 1$ (z.B. $c = 1/7$). Jeder dieser Slots ist entweder leer oder er enthält eine sortierte Folge von höchstens l_i Elementen. Abbildung 4.4 zeigt die Darstellung eines Externen Array-Heaps.

Die folgenden Berechnungen dienen dazu, Ihnen eine Vorstellung der Größenordnungen zu geben. Zum Beispiel gilt für $c = 1/7$:

$$\mu = \frac{M}{7B} - 1,\ l_1 = \frac{M}{7},\ l_2 = \frac{M^2}{49B} = l_1\frac{M}{7B} = l_1[\mu + 1]$$

Die letzte Beziehung kann verallgemeinert werden. Sie gilt für alle Schichten L_i.

Lemma 4.1 *Es gilt:* $l_{i+1} = l_i(\mu + 1)$

Beweis:

$$l_{i+1} = (cM)^{i+1}/B^i = l_i(cM)/B = l_i(\mu + 1)$$

□

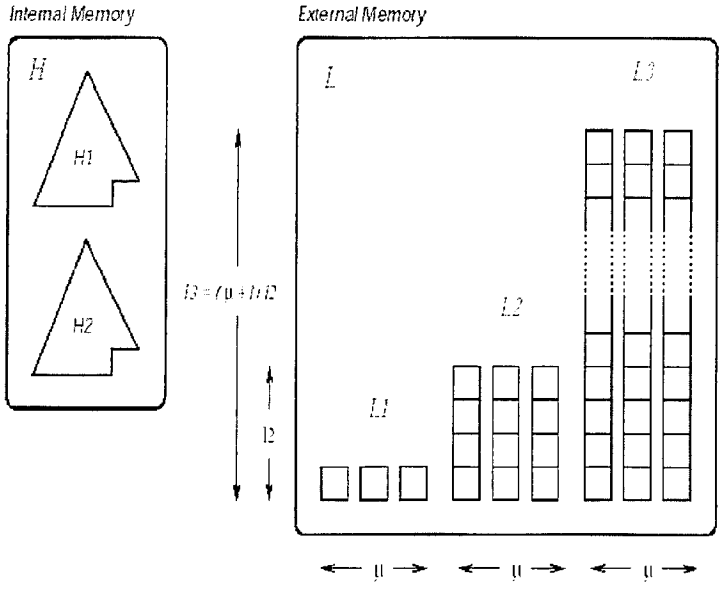


Abbildung 4.4: Externe Array-Heaps

Daraus folgt, dass die Anzahl der Plätze für Elemente in L_{i+1} so gewählt wurde, dass sie genau der Anzahl aller möglichen Plätze von Elementen in L_i entspricht (das sind also $l_i * \mu$), plus zusätzlich l_i Elementen.

Die Operation `Insert` fügt die Elemente immer in den Heap H ein. Ist dafür kein Platz mehr in H vorhanden (d.h. H läuft über), dann werden zunächst $l_1 = cM$ dieser Elemente in den Sekundärspeicher bewegt. Zunächst wird versucht, sie in die erste Schicht L_1 einzutragen. Dies geschieht in sortierter Folge, und zwar in einen noch freien Slot von L_1. Falls es in L_1 keinen freien Slot gibt, dann werden **alle** Elemente in L_1 mit den $l_1 = cM$ Elementen aus H zu einer sortierten Liste gemischt, die dann in einen freien Slot von L_2 geschrieben wird. Falls auch dort kein freier Slot existiert, wird dieser Prozess solange wiederholt, bis ein freier Slot gefunden wurde.

Die Operation `Del_Min` erhält die Invariante, dass sich das kleinste Element immer in H befindet. Um dies effizient zu ermöglichen, wird der Heap H in zwei Heaps H_1 und H_2 aufgeteilt. H_1 enthält maximal $2cM$ Elemente und speichert jeweils die neu eingefügten Elemente ab. H_2 speichert maximal die kleinsten B Elemente aus jedem belegten Slot j in L_i für alle $i = 1, \ldots, L$.

Es befinden sich also maximal

$$2cM + B\mu L = 2cM + B(\frac{cM}{B})L = cM(2 + L)$$

Elemente im Hauptspeicher. Weiterhin wird $(\mu + 1)B = cM$ zusätzlicher interner Speicher benötigt, um die μ Slots plus die *Overflow*-Folge zu mischen. Da M die Gesamtanzahl der Elemente im Hauptspeicher bezeichnet, muss also gelten

$$M \geq cM(3 + L), \text{ also } L \leq \frac{1 - 3c}{c}.$$

Das heißt, z.B. bei einem Wert von $c = 1/7$ muss $L \leq 4$ sein. L kann also im Folgenden als Konstante betrachtet werden.

Die folgenden Operationen sind für Externe Array-Heaps nützlich:

- **Merge-Level** (i, S, S'): produziert eine sortierte Folge S' durch das Mischen der sortierten Folge der μ Slots in L_i (inklusive der jeweils kleinsten Elemente, die sich zu diesem Zeitpunkt in H_2 befinden) und der sortierten Sequenz S. Diese Operation benötigt $O((|S| + l_{i+1})/B + 1) = O(l_{i+1}/B)$ I/O-Operationen.
- **Store** (i, S): nimmt an, dass L_i einen leeren Slot enthält und die Folge S eine Länge im Bereich $[l_i/2, l_i]$ besitzt. S wird in einen leeren Slot von L_i geschrieben, und seine B kleinsten Elemente werden nach H_2 bewegt. Diese Operation benötigt $O(|S|/B+1) = O(l_i/B)$ I/O-Operationen.
- **Load** (i, j): holt die nächsten B kleinsten Elemente vom j-ten Slot aus L_i nach H_2. Dafür wird eine I/O-Operation benötigt.
- **Compact** (i): nimmt an, dass mindestens zwei Slots in Level L_i existieren, deren Gesamtanzahl an Elementen, eingeschlossen diejenigen in H_2, höchstens l_i ist. Diese beiden Slots (plus ihre kleinsten Elemente, die sich zum aktuellen Zeitpunkt in H_2 befinden) werden gemischt und in einen freien Slot j' von L_i eingetragen. Damit ist nach dem Aufruf ein zusätzlicher Slot in L_i frei geworden. Die kleinsten B Elemente von j' werden nach H_2 übertragen. Dies benötigt insgesamt $O(l_i/B)$ I/O-Operationen.

Die Operationen `Insert` und `Del_Min` benutzen diese Operationen folgendermaßen:

`Insert`: Ein neues Element wird zunächst in H_1 eingefügt. Wenn H_1 voll wird, dann werden die größten $l_1 = cM$ Elemente von H_1 (Folge S) nach L_1 bewegt (die kleinsten B Elemente unter diesen bleiben im Hauptspeicher, sie werden nach H_2 kopiert). Sei $i = 1$. Falls L_i einen leeren Slot enthält, dann wird `Store`(i, S) aufgerufen. Sonst enthalten alle Slots von L_i mindestens $l_i/2$ Elemente, sie werden mit S gemischt zur Sequenz S' (`Merge-Level`(i, S, S')). S' wird nun zur nächsten Schicht L_{i+1} transferiert, usw. Diese Schleife wird höchstens L Mal durchgeführt (bis wir den höchsten Level erreichen).

`Del_Min`: Das kleinste Element befindet sich entweder im internen Heap H_1 oder im internen Heap H_2. In beiden Fällen entfernt man das Element. Im ersten Fall ist nichts weiter

zu tun. Im zweiten Fall korrespondiert das entfernte Element mit einem Slot j einer Schicht L_i. Falls dieses Element das letzte in H_2 ist, das zu dem assoziierten Slot j der Schicht L_i gehört, dann werden die nächsten B Elemente von Slot j nach H_2 mittels der Operation `Load`(i, j) bewegt. Nach jeder `Load`(i, j)-Operation werden die Größen der Slots getestet, und bei Bedarf wird `Compact`(i) aufgerufen. Diese Operation mischt bei Bedarf zwei kleine Sequenzen und bewegt die B kleinsten Elemente der neuen Folge in den internen Heap H_2.

Bemerkung: Es gilt zu jedem Zeitpunkt, dass in jeder Schicht L_i höchstens ein Slot existiert, der weniger als $l_i/2$ Elemente besitzt. Das werden wir später beweisen.

4.2.3 Korrektheit und Analyse

Zunächst analysieren wir die Korrektheit der beschriebenen Externen Array-Heap Datenstruktur.

Lemma 4.2 *Das kleinste Element ist immer im internen Speicher H_1 oder H_2.*

Beweis: Das kleinste Element wurde entweder neu eingefügt und befindet sich deswegen in H_1, oder es gehört zu einem bestimmten Slot in einer Schicht der externen Struktur. Da die Folgen innerhalb der Slots sortiert sind, befindet sich das kleinste Element im ersten Block dieses Slots und wurde deswegen nach H_2 bewegt. Es ist leicht zu sehen, dass diese Invariante unter den verwendeten Operationen bestehen bleibt. □

Lemma 4.3 *Es ist immer ein freier Platz für neue Elemente vorhanden.*

Beweis: Neue Elemente kommen entweder von einer unteren Schicht L_{i-1} durch die Operation `Merge-Level`$(i-1, S, S')$ verbunden mit der Operation `Store`(i, S') oder durch die reine Operation `Store`(i, S'). Lemma 4.1 besagt jedoch, dass für die Operation `Merge-Level`() genügend Platz vorhanden ist. □

Für die Analysen des folgenden Abschnitts nehmen wir an, dass $cM > 3B$ ist.

Lemma 4.4 *Nach N Operationen existieren höchstens $L \leq log_{cM/B}(N/B)$ Schichten.*

Beweis: Im schlimmsten Fall gibt es keine Element-Entfernungen. Deswegen bewegt jeder Überlauf die maximal mögliche Elementanzahl von einer Schicht in die nächste. Die Anzahl der Elemente in k Schichten ist somit

$$\sum_{i=1}^{k} \frac{(cM)^i}{B^{i-1}}\mu = \sum_{i=1}^{k} \left(\frac{(cM)^i}{B^{i-1}} \left(\frac{cM}{B} - 1 \right) \right) < \frac{(cM)^{k+2}}{B^{k+1}} .$$

Wir müssen das kleinste k wählen, sodass

$$cM(\frac{cM}{B})^{k+1} \geq N .$$

Wegen $cM > 3B$ gilt auch

$$cM(cM)^{k+1} \geq B(cM)^{k+1} \geq N .$$

Also folgt

$$k \geq log_{cM/B}(N/B) - 1 .$$

□

Im Folgenden analysieren wir die I/O-Operationen für Externe Array-Heaps.

Lemma 4.5 `Store`(i, S) *benötigt höchstens* $3l_i/B$ *I/O-Operationen.* `Compact`$(i+1)$ *und* `Merge-Level`(i, S, S') *benötigen höchstens* $3l_{i+1}/B$ *I/O-Operationen.*

Beweis: Wenn `Store`(i, S) ausgeführt wird, wird S zunächst gelesen, und dann wird es in L_i gespeichert. Wegen $l_i/B > 3$ (da $l_i \geq l_1 = cM > 3B$) benötigt `Store`(i, S) höchstens $2\lceil l_i/B \rceil \leq 3l_i/B$ I/O-Operationen. `Merge-Level`(i, S, S') liest und schreibt jedes Element der Folge S und alle Elemente aller Slots in L_i höchstens ein Mal. Dies ergibt

$$2\lceil (|S| + \mu l_i)/B \rceil \leq 2\lceil l_{i+1}/B \rceil \leq 3l_{i+1}/B$$

I/O-Operationen. Seien j und k die beiden Slots (plus deren Blocks in H_2), die durch die Operation `Compact`(i) gemischt werden, und seien e_j bzw. e_k die Anzahl der Elemente in j und k. `Compact`(i) benötigt höchstens

$$\lceil e_j/B \rceil + \lceil e_k/B \rceil + \lceil l_i/B \rceil \leq (e_j + e_k)/B + l_i/B + 3 \leq 2l_i/B + 3 \leq 3l_i/B$$

I/O-Operationen, wegen $l_i/B > 3$. □

4.2.4 Amortisierte Analyse zur Abschätzung der I/O-Operationen

Für die Analyse der I/O-Operationen verwenden wir die sogenannte *amortisierte Analyse*. Amortisierte Analyse ist ein allgemeines Verfahren, das dazu dient, die durchschnittlichen Kosten pro Operation für eine beliebige Folge von Operationen nach oben hin abzuschätzen. Das Verfahren der amortisierten Analyse wird besonders gern im Zusammenhang mit dynamischen Datenstrukturen verwendet. Eine klassische Worst-Case-Analyse würde für jede Einzeloperation die schlechtestmögliche Schrittzahl abschätzen, welche jedoch in einer Folge von Operationen eventuell eher selten auftaucht. Diese Abschätzung würde also

für eine Folge von Operationen im Allgemeinen eine unrealistisch schlechte Abschätzung ergeben, während die amortisierende Analyse die verschiedenen Fälle gegeneinander aufrechnet.

Die amortisierte Analyse verwendet eine Technik, die als Bankkonto-Paradigma bekannt geworden ist. Dabei wird jedem bei der Bearbeitung der Folge auftretenden Zustand ein Kontostand zugeordnet. Eine Einheit auf dem Konto repräsentiert eine Kosteneinheit bei der Abschätzung der Gesamtkosten. Ein Beispiel der Anwendung der Technik der amortisierten Analyse sehen wir im Beweis zum folgenden Satz.

Satz 4.1 *Sei* $N \leq B(\frac{cM}{b})^{1/c-3}, 0 < c < 1/3$ *und* $cM > 3B$. *In einer Folge von* N *Operationen der Art* `Insert` *und* `Del_Min` *benötigt* `Insert` $\frac{18}{B}(log_{cM/B}(N/B))$ *amortisierte I/O-Operationen und* `Del_Min` *7/B amortisierte I/O-Operationen.*

Beweis: Mit jedem nicht-leeren Slot j der Schicht L_i assoziieren wir einen Deposit D_{ij} (eine "Einzahlung"), der für $6x/B$ Punkte zählt, wobei x die Anzahl der leeren Einträge ist. Anfangs sind alle Slots leer, d. h. die Konstostände sind alle leer. Mit dem internen Heap H assoziieren wir einen Kontostand, der immer dafür sorgen wird, dass mindestens eine Einheit Guthaben vorhanden ist, sobald die `Load()`-Operation aufgerufen wird. Denn dieses Mindestguthaben wird benötigt, um die `Load()`-Operation zu bezahlen.

Jedes Element erhält beim Einfügen ein Guthaben von $18L/B$ Kosteneinheiten. Wir zeigen, dass davon je $18/B$ Einheiten genügen, um von einer Schicht zur nächsten Schicht zu wandern (Operation `Merge-Level()` inklusive der `Store()`-Operation). Wir ordnen also jedem Element in jeder Schicht i, $i = 1, \ldots, L$, $18/B$ Einheiten zur eventuellen Benutzung zu. Bei Entfernung eines Elements werden $7/B$ Einheiten Guthaben im internen Heap belassen.

Im Fall eines Überlaufs benutzen wir die Operationen `Merge-Level`(i, S, S') und danach `Store`$(i+1, S')$. Nach Lemma 4.5 benötigt jede Operation `Merge-Level`(i, S, S') und jede `Store`$(i+1, S')$-Operation höchstens je $3l_{i+1}/B$ I/O-Operationen. Dies sind zusammen $6l_{i+1}/B$ I/O-Operationen.

Weil die Größe von S' zwischen $l_{i+1}/2$ und l_{i+1} ist, können diese Kosten durch je $12/B$ Einheiten aus den Guthaben jener Elemente genommen werden, die von dieser Operation betroffen sind. Denn: Wir bewegen mindestens $l_{i+1}/2$ Elemente von einer Schicht L_i zur nächsten Schicht L_{i+1}. (Dabei nehmen wir an, dass L_0 dem internen Heap H_1 entspricht.) Dabei benutzen wir deren Guthaben aus Schicht L_{i+1}. Dies ergibt also zusammen $(12/B)(l_{i+1}/2) = 6l_{i+1}/B$. Nach der `Store()`-Operation ist nun ein Slot j in Schicht L_{i+1} teilweise gefüllt, der vorher leer war. D. h. wir müssen dafür sorgen, dass das Guthaben $D_{i+1,j}$ auf den erforderlichen Wert $6x/B$ aufgefüllt wird, wobei x die Anzahl der leeren Einträge in j bezeichnet.

Wir wissen jedoch, dass x höchstens $l_{i+1}/2$ ist, d.h. wir müssen $D_{i+1,j}$ auf höchstens

$6l_{i+1}/(2B) = 3l_{i+1}/B$ Einheiten auffüllen. Diese erhalten wir, indem wir von jedem der $|S'|$ Elemente, die gerade durch `Store`$(i+1, S')$ in den Slot j gewandert sind, $6/B$ Einheiten abziehen. Insgesamt haben wir also $18/B$ Einheiten Guthaben benötigt, um ein Element von L_i nach L_{i+1} zu bewegen; dies sind die amortisierten Kosten einer Einfüge-Operation für eine Schicht. Insgesamt über alle Schichten ergeben sich $18L/B$ Einheiten für eine `Insert`-Operation.

Im Falle einer `Del_Min`-Operation werden $7/B$ Einheiten Guthaben im internen Heap belassen. Die Kosten für jede benötigte `Load()`-Operation betragen eine Einheit, die wir jeweils aus dem Guthaben von H bezahlen. Das geht sich aus, denn: Die `Load`(i,j)-Operation wird jeweils nach B erfolgreichen Entfernungen aus H_2, die zu dem Slot j assoziiert waren, ausgeführt; von jedem der entfernten Elemente werden jeweils $1/B$ Einheiten für diese `Load()`-Operation zur Verfügung gestellt, das ergibt zusammen $B(1/B) = 1$.

Die anderen $6/B$ Einheiten Guthaben je entferntem Element werden auf das Konto des Slots j, assoziiert mit der Load (i,j)-Operation, d.h. $D_{i,j}$, gutgeschrieben, um die Invariante zu erhalten. Denn in diesem Slot sind nach der `Load()`-Operation B nicht-gefüllte Plätze hinzugekommen, was zur Folge hat, dass das Guthaben $D_{i,j}$ um $6B/B = 6$ Einheiten erhöht werden muss.

Die `Compact`(i)-Operation benötigt $3l_i/B$ Einheiten. Dies kann allein durch die Guthaben der betroffenen Slots $D_{i,j}$ und $D_{i,k}$ bezahlt werden (es seien j und k die beiden betroffenen Slots). Denn die Gesamtanzahl der nicht-belegten Plätze vor der Ausführung der `Compact`(i)-Operation in den Slots j und k ist mindestens l_i. Also beträgt das Gesamtguthaben von $D_{i,j}$ und $D_{i,k}$ mindestens $6l_i/B$ Einheiten. Nach dem Mischen der beiden Slots wird ein Slot der beiden, z.B. k, leer, und Slot j erhält höchstens $l_i/2$ nicht-belegte Plätze. D.h. wir können $3l_i/B$ des Gesamtguthabens als Bezahlung für die `Compact`(i)-Operation benutzen, die anderen $3l_i/B$ Einheiten bleiben als Guthaben bei $D_{i,j}$. Damit ist der Satz bewiesen. □

Die obere Schranke für N resultiert aus den Platzbeschränkungen: Der Heap benötigt $cM(3 + log_{cM/B}(N/B))$ internen Speicherplatz (siehe nächstes Theorem). Die Schranke folgt aus der Lösung der Ungleichung $cM(3 + log_{cM/B}(N/B)) \leq M$ für N.

Als Nächstes analysieren wir den Speicherplatzbedarf.

Lemma 4.6 *Jede Schicht L_i enthält höchstens einen Slot, der nicht-leer ist und aus weniger als $L_i/2$ Elementen besteht.*

Beweis: Ein Slot j kann durch die Operation `Load`(i,j) Elemente verlieren. Wir nehmen als Induktionsannahme an, dass Slot j in Schicht L_i als einziger dieser Schicht nicht-leer ist und weniger als $l_i/2$ Elemente besitzt. Nach der Load(i,j)-Operation wird von der `Del_Min`-

Operation getestet, ob zwei Slots existieren, deren Gesamtelementanzahl kleiner oder gleich l_i ist. Falls dies der Fall ist, wird die Operation `Compact` aufgerufen. Wir nehmen an, dass dies für Slot j und Slot k der Fall ist. Dann jedoch wird ein Slot, sagen wir Slot k, geleert, und der andere, Slot j, wird aufgefüllt. Andernfalls ist Slot j immer noch der einzige nicht-leere Slot in Schicht L_i mit weniger als $L_i/2$ Elementen. □

Satz 4.2 *Die Gesamtanzahl der benützten Blöcke ist höchstens* $2(X/B) + L$, *wobei* X *die Anzahl der Elemente im Heap* H *ist. Der Gesamtspeicherplatz im Arbeitsspeicher beträgt* $cM(3 + L)$.

Beweis: In jedem Slot jeder Schicht existiert höchstens ein nur teilweise gefüllter Speicherblock, nämlich der oberste. Weiterhin wissen wir, dass maximal ein nicht-leerer Slot pro Schicht L_i existiert, der weniger als $l_i/2$ Elemente besitzt; dieser kann im schlechtesten Fall aus nur einem einzigen Block bestehen, der noch dazu nur teilweise gefüllt ist. Von diesen Fällen existieren zusammen höchstens L. Für die anderen Slots jedoch, die mindestens halb voll sind, gibt es mindestens auch einen voll beschriebenen Block (aus $L_1 = cM$, $l_i > cM$ für $i \geq 2$, und $cM > 3B$ folgt, dass $l_i/2 > 3/2B$ für $i \geq 1$).

Das heißt, dass die Anzahl dieser teilweise beschriebenen Blöcke nicht größer als die Anzahl der ganz belegten Blöcke ist. Diese ist X/B. Daraus folgt die Beschränkung von $2(X/B)$. Zusammen ergibt dies $2(X/B) + L$. H_1 speichert $2cM$ Elemente und H_2 speichert einen Block pro Slot, das sind $\mu BL = (cM/B-1)BL \leq cML$. Zusätzlich wird Platz für $(\mu+1)B$ Elemente zum Mischen der Slots benötigt. Zusammen sind dies $2cM + cML + cM = cM(L + 3)$. □

Was bedeutet nun die Schranke für N aus Satz 4.2 in der Praxis?

Wir betrachten den Fall $M = 10^9$ und $B = 10^6$ genauer. In diesem Fall wäre

$$N \leq 10^6(c10^3)^{1/c-3}.$$

Wenn wir $c = 1/7$ annehmen, dann folgt daraus

$$N \leq 10^6(\frac{10^{12}}{7^4}) = 0,416 * 10^{15}$$

und $L \leq 4$ (siehe oben).

Selbsttest-Aufgabe: Überlegen Sie sich eine genaue Realisierung für die beschriebene Datenstruktur Array-Heap. Welche Datenstruktur verwenden Sie für den Heap H_1 (H_1 und H_2 müssen keine echten Heaps sein)? Beschreiben Sie die Operation des Überlaufs im Heap beim Einfügen eines Elements, wenn die größten Elemente zu L_1 bewegt werden. Welche Datenstrukturen verwenden Sie für den Heap H_2? Welche Datenstruktur verwenden Sie für die Slots und Schichten? Analysieren Sie die Laufzeiten für die Operationen

Insert/Delete_min time performance of the external queues (in secs)

N [$*10^6$]	radix heap	array heap	buffer tree	buffer tree (orig.)	B-tree
1	6/24	18/11	56/34	62/43	11287/259
5	17/97	74/63	148/309	235/345	66210/1389
10	35/178	353/89	201/882	415/741	-
25	85/372	724/295	311/2833	1096/2302	-
50	164/853	1437/645	445/6085	3462/7592	-
75	246/1416	2157/1005	569/9880	7042/11907	-
100	325/1957	2888/1408	734/19666	12508/16909	-
150	478/3084	4277/2297	*	25051/27181	-
200	628/4036	5653/3234	*	*	-

Random/Total I/Os for external queues

N [$*10^6$]	radix heap	array heap	buffer tree	buffer tree (orig.)
1	44/420	24/720	228/668	228/668
5	422/3550	120/4560	16722/21970	13381/15501
10	1124/8620	168/9440	35993/47297	30950/35374
25	2780/21820	570/29520	93789/123285	80495/97879
50	7798/56830	1288/66160	190147/249955	179809/201921
75	12466/89370	2016/102480	286513/376625	275713/310977
100	17736/124740	2776/139760	383518/504134	381023/426615
150	27604/192500	4216/210080	*	595157/659933
200	38284/211570	5712/284320	*	

Insert/Delete_min time performance of the internal queues (in secs)

N [$*10^6$]	Fibonacci heap	k-ary heap	pairing heap	radix heap
1	3/32	4/33	3/19	3/11
2	6/73	8/75	6/45	5/27
5	17/208	21/210	14/126	11/71
7.5	172800*/-	32/344	22/207	18/124
10	-/-	43/482	30/291	23/162
20	-/-	172800*/-	172800*/-	172800*/-

Abbildung 4.5: Experimentelle Laufzeit für insert-all-delete-all

`Merge-Level`(i, S, S'), `Store`(i, S), `Load`(i, j) und `Compact`(i) im RAM-Modell. Beschreiben Sie eine Realisierung der Operationen `Insert` und `Del_Min` und analysieren Sie ihre Laufzeit.

4.2.5 Experimenteller Laufzeitvergleich

Andreas Crauser hat in seiner Dissertation acht verschiedene Implementierungen für Prioritätswarteschlangen experimentell getestet. Darunter waren vier Realisierungen, die speziell für Externspeicheranwendungen entwickelt wurden, wie Array-Heaps, Externe Radix-Heaps, Buffer Bäume, und B-Bäume. Die vier restlichen Realisierungen waren klassische Algorithmen im RAM-Modell, wie Fibonacci-Heaps, K-ary-Heaps, Pairing-Heaps und interne Radix-Heaps.

Time performance on mixed operations			
N[$*10^6$]	radix heap	array heap	buffer-tree
50	544	770	4996
75	609	945	5862
100	619	1027	6029
Random/Total I/Os on mixed operations			
50	2935/19615	22325/26997	153321/177201
75	5128/26752	24256/28384	171615/196647
100	5782/30094	24220/28380	171658/196578

Abbildung 4.6: Experimentelle Laufzeit und I/O-Operationen für die gemischten Operationen

Die erste Reihe von Experimenten fügt zunächst N Elemente in die Prioritätswarteschlange ein und entfernt diese am Ende wieder. Eine zweite Reihe von Experimenten führt die Insert- und Del-Min-Operationen in gemischter Reihenfolge durch, nachdem zunächst 20 Millionen Elemente eingefügt worden sind. Eine Insert-Operation erfolgt dabei mit Wahrscheinlichkeit $1/3$, und eine Del-Min-Operation mit Wahrscheinlichkeit $2/3$.

Abbildung 4.5 zeigt die experimentellen Resultate. Beim Vergleich zwischen den Externspeicheralgorithmen schnitten B-Bäume, die wir bereits aus der Vorlesung *Algorithmen und Datenstrukturen 1* kennen, am schlechtesten ab. Es handelte sich dabei um eine Implementierung als B*-Baum, so dass ein Knoten mit allen notwendigen Informationen und Links genau auf eine Seite passt. Sie benötigen für `Insert` im schlechtesten Fall $O(\log_B N)$ I/O-Operationen und besitzen experimentell die längste Rechenzeit. Dies heißt nicht, dass B-Bäume schlechte Externspeicheralgorithmen sind; sie sind einfach nicht so gut als Realisierung der Datenstruktur Prioritätswarteschlange geeignet wie auf Heaps basierende Datenstrukturen.

Externe Radix Heaps führen zu den schnellsten Verfahren. Jedoch ist diese Datenstruktur nur beschränkt anwendbar, nämlich nur dann, wenn die Schlüssel ganzzahlig sind und die `Del_Min` Operationen aufsteigende Schlüssel beinhalten (wie z.B. bei Dijkstras kürzestem Wege Algorithmus).

Crausers Implementierung der Array-Heaps ist bei `Insert` bis zum Faktor 10 langsamer als Radix Heaps und Buffer Bäume, allerdings um einen Faktor ≤ 9 schneller bei der `Del_Min` Operation. Insgesamt sind sie ca. 3 Mal so schnell wie Buffer Bäume und ca. um einen Faktor $2,5$ langsamer als Radix Bäume. Array-Heaps benötigen nur wenig mehr I/O-Operationen als Radix Heaps.

Die vier internen Realisierungen brechen bereits bei $N = 20$ Mio. Operationen ein. Die

Ergebnisse für die zweite Testserie sind ähnlich (s. Tabelle 4.6).

4.3 Cache-optimale Algorithmen

Der Cache befindet sich zwischen dem (schnellen) Prozessor und dem (langsamen) Arbeitsspeicher, und enthält jeweils jene Regionen des Arbeitsspeichers, die zuletzt referenziert wurden. Zugriffe auf Speicherbereiche, die sich im Cache befinden (sogenannte *Hits*), können mit Prozessorgeschwindigkeit bearbeitet werden. Zugriffe auf Speicherbereiche, die sich nicht in Cache befinden (sogenannte *Misses*), ziehen, bevor sie ausgeführt werden können, zunächst eine Operation des Caches nach sich, um diese Daten in den Cache zu holen.

Caches funktionieren, weil die meisten Programme eine hohe Lokalität besitzen:

Zeitliche Lokalität existiert, wenn der gleiche Speicherbereich öfters innerhalb einer kurzen Zeitspanne angesprochen wird. Caches nutzen zeitliche Lokalität aus, indem sie versuchen, angesprochene Speicherbereiche möglichst lange im Cache zu lassen.

Örtliche Lokalität existiert, wenn öfter Speicherbereiche angesprochen werden, die nahe beieinander liegen. Caches nutzen örtliche Lokalität aus, indem sie bei jedem *Miss* jeweils ganze Speicherblöcke in den Cache holen.

Caches können durch zwei Parameter charakterisiert werden:

(i) Seitengröße L (in Byte)

(ii) Kapazität Z (in Byte)

Ein Cache kann maximal Z Bytes aufnehmen. Diese sind unterteilt in Seiten der Größe L. Es gibt also auch hier — wie bereits im Externspeichermodell — eine Speicherhierarchie auf 2 Ebenen, bei dem die untere Ebene Kapazität Z besitzt und die Daten zwischen den beiden Ebenen in Blöcken der Größe L hin- und herverschoben werden.

Als Folgerung daraus, kann man auch hier das EM-Modell von Aggarwal und Vitter (1988) zugrunde legen. Das heißt, dass auch alle Externspeicheralgorithmen, die für das EM-Modell entwickelt wurden, grundsätzlich auch zur Cache-Optimierung verwendet werden können.

Allerdings enthalten moderne Computer tatsächlich nicht nur zwei Speicherschichten, sondern mehrere. Typische Komponenten sind: Register, Level 1 Cache, Level 2 Cache, Arbeitsspeicher und Externspeicher. Diesem Aspekt tragen Frigo, Leiserson, Prokop

und Ramachandran (1995) Rechnung, indem sie das EM-Modell zum *Cache-Oblivious* Speichermodell (CO-Modell) erweitert haben.

Das *Cache-Oblivious* Speichermodell sieht vor, dass die Seitengröße L und die Kapazität Z unbekannt sind. Eine Algorithmen-Analyse im CO-Modell muss also für alle Blockgrößen und Kapazitäten gelten, folglich daher auch für alle Ebenen der Speicherhierarchie.

Indem man einen Algorithmus für unbekanntes C und Z optimiert, optimiert man automatisch die Speicherzugriffe auf jeder Ebene. Dies erhöht außerdem die Portabilität von Programmen, die im EM-Modell speziell für ein festes L und Z bzw. B und M entwickelt wurden.

4.3.1 Das ideale Cache-Modell von Frigo et al.

Das *ideale Cache-Modell* von Frigo et al. ist in Abbildung 4.7 dargestellt. Das Modell sieht eine 2-Ebenen Speicherhierarchie vor, die aus einem idealen Cache mit Z Bytes und einem unendlich großen Arbeitsspeicher besteht. Der Cache ist in Cache-Zeilen aufgeteilt, die jeweils aus L Bytes bestehen und jeweils in einem Schritt zwischen Cache und Arbeitsspeicher bewegt werden können. Dabei wird angenommen, dass der Cache hoch ist, d. h. es gilt $Z = \Omega(L^2)$. Der Prozessor kann jeweils nur Speicherplätze ansprechen, die in einer Zeile des Cache gespeichert werden. Der ideale Cache benützt eine optimale off-line Strategie, indem er jeweils diejenigen Speicherzeilen im Cache ersetzt, deren Zugriff am weitesten in der Zukunft liegt.

Die Analyse erfolgt durch die beiden Parameter

(i) Anzahl der ausgeführten CPU-Operationen im RAM-Modell;

(ii) Cache-Komplexität $Q(n, Z, L)$: Anzahl der Cache-Misses, abhängig von Z und L.

Ein Algorithmus heißt *„Cache-Aware" (Cache-bewusst)* wenn er Parameter enthält, mit denen die Cache-Komplexität für die gegebene Cache-Größe Z und Zeilenlänge L optimiert werden kann. Andernfalls heißt der Algorithmus *„Cache-Oblivious" (Cache-ignorierend).*

Man kann zeigen, dass ein Cache-Oblivious Algorithmus, der für das 2-Ebenen Hierarchiespeichermodell optimiert wurde, auch für mehrere Ebenen optimal ist.

Im Folgenden werden wir einen Cache-Aware Algorithmus sowie einen Cache-Oblivious Algorithmus für das Problem der Matrizen-Multiplikation kennenlernen, die beide bezüglich der Anzahl der I/O-Operationen optimal sind.

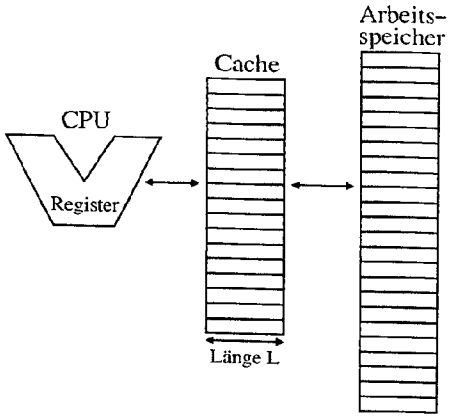


Abbildung 4.7: Das ideale Cache-Modell von Frigo et al.

4.3.2 Ein Cache-Aware Algorithmus zur Matrix-Multiplikation

Wir betrachten die Multiplikation zweier $n \times n$ Matrizen A und B. Wir nehmen an, dass die Matrizen jeweils zeilenweise gespeichert sind, und dass n sehr groß ist, d.h. $n > L$. Der folgende Algorithmus ist *Cache-Aware*:

Algorithmus 17 BLOCK-MULT (A, B, C, n)

```
1: für i = 1 bis n/s {
2:   für j = 1 bis n/s {
3:     für k = 1 bis n/s {
4:       MULT(A_ik, B_kj, C_ij, s)
5:     }
6:   }
7: }
```

Dabei ist $MULT(A, B, C, s)$ die Standard-Prozedur, die $C = C + AB$ auf $s \times s$ Untermatrizen in Zeit $O(s^3)$ berechnet. Dabei wird angenommen, dass s Teiler von n ist, ansonsten wäre der Code etwas aufwändiger. Außerdem wurde die Initialisierung von C weggelassen.

Abhängig von der Cachegröße des Rechners kann der Parameter s so bestimmt werden, dass die Cache-Komplexität optimiert werden kann. Es handelt sich also dabei um einen Algorithmus, der "Cache-Aware" ist.

Um die Cache-Komplexität zu minimieren, wählen wir s als den größten Wert, für den die drei $s \times s$ Untermatrizen zusammen in den Cache passen. Eine $s \times s$ Untermatrix benötigt $\Theta(s + s^2/L)$ Cache-Zeilen. Das liegt daran, dass die Matrix nicht unbedingt mit den Grenzen

der Blöcke oder Zeilen abschließt, die gleichzeitig aus dem Hauptspeicher in den Cache geladen werden. Im schlimmsten Fall ist der Platzbedarf der Matrix im Cache deshalb

$$\Theta(s(\lceil s/L \rceil)) = \Theta(s(1 + s/L)) = \Theta(s + s^2/L)$$

Aus unserer Annahme, dass $Z = \Omega(L^2)$ ist, folgt $s = \Theta(\sqrt{Z})$.

Folglich benötigt jeder Aufruf von $MULT()$ höchstens $Z/L = \Theta(s^2/L)$ Cache-Misses, um die drei Untermatrizen von A, B und C in den Cache zu bringen. Daraus folgt insgesamt eine Cache-Komplexität von

$$\Theta(1 + n^2/L + (n/\sqrt{Z})^3(Z/L)) = \Theta(1 + n^2/L + n^3\sqrt{Z}/L),$$

denn der Algorithmus muss n^2 Elemente einlesen, die auf $\lceil n^2/L \rceil$ Cache-Zeilen verteilt sind.

4.3.3 Ein Cache-Oblivious Algorithmus zu Matrixmultiplikationen

Wir wollen eine $m \times n$ Matrix A mit einer $n \times p$ Matrix B *cache-oblivious* multiplizieren. Der Algorithmus REC-MULT arbeitet rekursiv und folgt dem Divide&Conquer-Prinzip:

Fall 1: Falls $m \geq max\{n, p\}$, wird die Matrix A horizontal in Matrizen A_1 und A_2 mit jeweils $\lceil m/2 \rceil$ und $\lfloor m/2 \rfloor$ Zeilen gesplittet. Es folgen zwei Aufrufe der Form A_1B und A_2B. Hierbei wird das Gesetz

$$\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1B \\ A_2B \end{pmatrix}$$

ausgenutzt.

Fall 2: Andernfalls, falls $n \geq max\{m, p\}$ gilt, werden beide Matrizen aufgeteilt, nämlich A vertikal in A_1 und A_2 mit jeweils $\lceil n/2 \rceil$ und $\lfloor n/2 \rfloor$ Spalten und B horizontal in B_1 und B_2 mit jeweils $\lceil n/2 \rceil$ und $\lfloor n/2 \rfloor$ Zeilen. Hierbei wird das Gesetz

$$\begin{pmatrix} A_1, A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1B_1 + A_2B_2$$

genutzt.

Fall 3: Andernfalls, falls gilt $p \geq max\{m, n\}$. In diesem Fall wird B vertikal aufgesplittet in B_1 und B_2 mit jeweils $\lceil p/2 \rceil$ und $\lfloor p/2 \rfloor$ Spalten. Es gilt

$$A(B_1, B_2) = (AB_1, AB_2).$$

Fall 4: Falls $m = n = p = 1$ gilt, dann werden die beiden Elemente multipliziert und zur resultierenden Matrix C hinzuaddiert.

Analyse des Algorithmus REC-MULT

Satz 4.3 *Der Algorithmus REC-MULT zur Multiplikation zweier Matrizen der Größe $m \times n$ und $n \times p$ benötigt $\Theta(mnp)$ Zeit im RAM-Modell. Die Cache-Komplexität beträgt $\Theta(m + n + p + (mn + np + mp)/L + mnp\sqrt{Z}/L)$ Cache-Misses.*

Beweis: [ohne Beweis]; für interessierte Studierende: siehe Frigo, Leiserson, Prokop und Ramachandran. □

Korollar 4.1 *Zur Multiplikation zweier $n \times n$ Matrizen benötigt der Algorithmus REC-MULT $\Theta(n^3)$ Zeit im RAM-Modell und $\Theta(n + n^3\sqrt{Z}/L)$ Cache-Komplexität.*

Intuitiv benutzt der Algorithmus den Cache effektiv, denn sobald ein Unterproblem ganz in den Cache passt, können dessen Unterprobleme ohne einen einzigen Cache-Miss gelöst werden. Deswegen sind Divide&Conquer-Algorithmen grundsätzlich sehr gut für große Datenmengen geeignet.

Ein Verfahren mit besserer Laufzeit ist das Verfahren von Strassen (1968) zur Matrixmultiplikation. Auch dieses Verfahren beruht auf dem Divide&Conquer-Prinzip. Jedoch wird hierbei jede Untermatrix in der Höhe geviertelt und in der Breite in Stücke von jeweils ungefähr $n/2$ Spalten zerlegt.

Die Komplexität von Strassens Algorithmus zur Matrixmultiplikation zweier $n \times n$ Matrizen ist

$$\Theta(N^{log7}) = \Theta(N^{2,81}) \text{ mit } \Theta(n + n^2/L + n^{log7}\sqrt{Z}/L).$$

Auch dieser Algorithmus ist Cache-Oblivious und kommt theoretisch mit weniger Cache-Misses aus. Allerdings ist der Algorithmus von Strassen sehr aufwändig zu implementieren. In der Praxis ist der Algorithmus von Strassen für $n \leq 1.000.000$ noch ungefähr genauso langsam wie die besprochenen Algorithmen zu Matrixmultiplikation. Generell wird der Algorithmus von Strassen als ein eher theoretischer Beitrag gesehen, der damals allerdings sehr überraschend war.

Inzwischen wurden viele Varianten des Algorithmus von Strassen gefunden. Allerdings gilt das Matrixmultiplikationsproblem als eines der interessantesten offenen Probleme, denn man geht davon aus, dass "der beste" Algorithmus für dieses Problem noch nicht gefunden wurde.

4.4 Weiterführende Literatur

Der Abschnitt über Priority Queues hält sich eng an *Kapitel 4: Priority Queues* aus der Dissertation von Andreas Crauser, *LEDA-SM: External Memory Algorithms and Data Structures in Theory and Practice*, Universität des Saarlandes, 2001
`http://www.mpi-sb.mpg.de/~crauser/diss.pdf`.

Eine Kurzfassung gibt es auch in: Klaus Brengel, Andreas Crauser, Paolo Ferragina, Ulrich Meyer: *An Experimental Study of Priority Queues in External Memory*, Proc. of the Workshop on Algorithmic Engineering (WAE '99), Lecture Notes in Computer Science 1668, 345-359, Springer-Verlag, 1999.

Der Abschnitt über Cache-optimale Algorithmen hält sich an: Frigo, Leiserson, Prokop, Ramachandran, *Cache-Oblivious Algorithms*, MIT, Cambridge, IEEE Transactions, 1999.

Originalliteratur zum Thema finden Sie unter: A. Aggarwal und J.S. Vitter: *The input/output complexity of sorting and related problems*, Communications of the ACM 31 (9), 1988, 1116-1127

oder auch: J.S. Vitter und E.A.M. Shriver: *Optimal algorithms for parallel memory I: two-level memories*, Algorithmica, 12(2-3), 1994

Kapitel 5

Tries

Für verschiedene Aufgaben in der Textverarbeitung, wie beispielsweise der automatisierten Überprüfung der Rechtschreibung (Wörterbücher), ist der *Trie* eine wichtige Datenstruktur, die wir deshalb genauer behandeln wollen. Der Name „Trie" kommt vom englischen „re*trie*ve" (wiederauffinden). Er bietet die Möglichkeit, eine große Anzahl an Daten (vor allem Strings) kompakt zu speichern und effizient einzelne Einträge zu suchen. Der Aufwand einer Suche ist dabei praktisch unabhängig von der Anzahl gespeicherter Einträge (vgl. Hashtabellen). Den Trie gibt es in unterschiedlichen Ausprägungen, die wir in weiterer Folge betrachten wollen.

5.1 Radix Trie

Grundsätzlich ist ein *Radix Trie* ein binärer Baum, bei dem jedoch nur die Blätter Datensätze beinhalten, die nach einem Schlüssel geordnet sind.

Jeder innere Knoten wird auch *Routerknoten* genannt und besitzt zwei Verweise $next[0]$ und $next[1]$, die entweder auf weitere Routerknoten – den linken bzw. rechten Teilbaum – verweisen oder den Wert NULL besitzen. Ein *Datenknoten* ist immer ein Blatt und beinhaltet den Schlüssel und eventuelle Anwenderdaten.

Der Schlüssel ist hier immer ein binärer Vektor $(S_1, S_2, \ldots, S_m) \in \{0,1\}^m$ einer vorgegebenen Länge m. Andere Datentypen müssen gegebenenfalls in eine solche Binärrepräsentation fixer Länge umgewandelt werden.

Für den Radix Trie gilt Folgendes:

- In einem Routerknoten sind nie beide Verweise gleich NULL.

E	00101
I	01001
N	01110
U	10101
V	10110

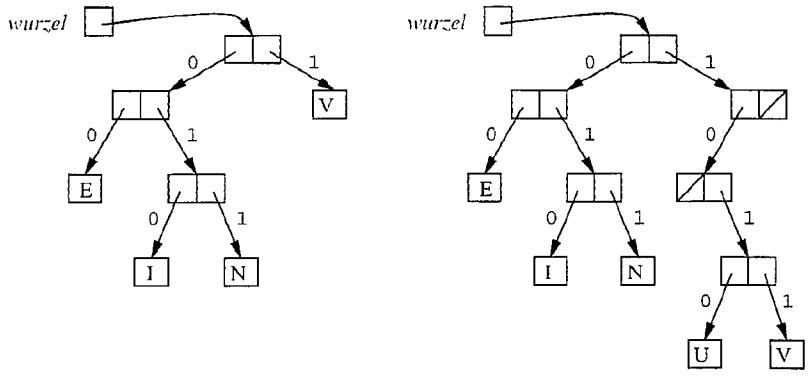


Abbildung 5.1: Ein Radix Trie vor und nach dem Einfügen von 'U'.

- Hat ein Routerknoten nur einen Nachfolger, so ist dies immer ein weiterer Routerknoten, keinesfalls ein Datenknoten. Diese und die vorige Bedingung stellen sicher, dass der Radix Trie keine „unnötigen" Routerknoten besitzt.

- Jeder Radix Trie hat maximal Höhe m und kann 2^m Datensätze aufnehmen. Er kann somit nicht entarten, wie dies etwa beim einfachen binären Suchbaum möglich ist.

- Für jeden Routerknoten der Ebene $i \in \{1, \ldots, m\}$ des Trie gilt, dass in seinem linken Teilbaum nur Datensätze gespeichert sind, deren S_i (das i-te Bit des Schlüssels) gleich 0 ist. Umgekehrt beinhaltet der rechte Teilbaum nur jene Datensätze mit $S_i = 1$.

- Der leere Baum wird lediglich durch eine Verweis-Variable *wurzel* = NULL repräsentiert.

Abbildung 5.1 zeigt einen Radix Trie vor und nach dem Einfügen von 'U' (= `10101`). Im Folgenden betrachten wir die Such-, Einfüge- und Löschoperationen genauer.

5.1.1 Einfügen

Randbedingung: Wir gehen davon aus, dass zwei Datensätze niemals gleiche Schlüssel besitzen. Treten in einer Anwendung dennoch Datensätze mit gleichen Schlüsseln auf, so kann dies beispielsweise durch eine zusätzliche äußere Verkettung gelöst werden, d.h. die Datenknoten sind jeweils lineare Listen, die alle Datensätze mit identen Schlüsseln beinhalten.

Die in Algorithmus 18 gezeigte rekursive Prozedur wird für einen Trie *wurzel* und einen noch nicht darin enthaltenen Schlüssel S wie folgt aufgerufen:

Algorithmus 18 Einfügen (**var** p, S, i)

Eingabe: Radix Trie mit Wurzel p; Schlüssel S; Ebene i
Variable(n): Knoten q

```
1: // Fügt Schlüssel S in den Radix Trie mit Wurzel p ein
2: falls p == NULL dann {
3:    p = Erzeuge Datenknoten für Datensatz mit Schlüssel S;
4: } sonst falls p ist Routerknoten dann {
5:    Einfügen (p.next[S_i], S, i + 1);
6: } sonst {
7:    // p ist ein Datenknoten; ein neuer Routerknoten wird eingefügt:
8:    q = p;
9:    p = neuer Routerknoten;
10:   p.next[q.S_i] = q;
11:   p.next[1 - q.S_i] = NULL;
12:   Einfügen (p.next[S_i], S, i + 1);
13: }
```

Einfügen (*wurzel*, S, 1);

Stößt die Prozedur auf einen Datenknoten, so werden in den weiteren Schritten so lange Routerknoten erzeugt, bis sich dieser Datensatz und der neu einzufügende in einem Bit unterscheiden. Im Worst-Case wird in einem anfangs aus einem Datensatz bestehenden Trie ein zweiter Datensatz eingefügt, der sich vom ersten nur im letzten Bit unterscheidet. Vergleiche auch Abbildung 5.1.

Anmerkung: Der Fall, dass S bereits im Trie enthalten ist, wurde aus Übersichtlichkeitsgründen nicht berücksichtigt.

Aufwand: Da der Baum maximal Höhe m hat und der Schlüssel zumindest einmal vollständig kopiert werden muss, ist der Aufwand des Einfügens $\Theta(m)$.

5.1.2 Suchen

Siehe Algorithmus 19.

Aufwand: Wiederum ist durch die beschränkte Höhe des Baumes der Aufwand grundsätzlich in $O(m)$. Im erfolgreichen Fall müssen alle Stellen des Schlüssels überprüft werden, daher ist der Aufwand immer $\Theta(m)$. Bei einer erfolglosen Suche findet im besten Fall bereits beim ersten Bit ein Abbruch statt, d.h. eine untere Schranke ist dann $\Omega(1)$.

Algorithmus 19 Suchen (p, S, i)

Eingabe: Radix Trie mit Wurzel p; Schlüssel S; Ebene i

Ausgabe: Aussage, ob Datensatz mit Schlüssel S im Radix Trie mit Wurzel p enthalten ist oder nicht

```
1: falls p == NULL dann {
2:    Datensatz nicht enthalten;
3: } sonst falls p ist Routerknoten dann {
4:    Suchen (p.next[S_i], S, i + 1);
5: } sonst {
6:    // p ist ein Datenknoten:
7:    falls (p.S_i, ..., p.S_m) == (S_i, ..., S_m) dann {
8:       Datensatz gefunden;
9:    } sonst {
10:      Datensatz nicht enthalten;
11:   }
12: }
```

5.1.3 Entfernen

Algorithmus 20 zeigt das Entfernen eines Datensatzes. Dabei werden auch alle Routerknoten, die durch das Entfernen eines Unterknotens nur mehr ein einzelnes Blatt haben würden, entfernt. So ist auch sichergestellt, dass es keine Routerknoten ohne Nachfolger geben kann.

Der Aufwand des Entfernens eines Datensatzes ist ebenfalls $O(m)$.

Eine Eigenschaft aller Tries, die sie von den meisten anderen Baumstrukturen unterscheidet ist, dass es für eine gegebene Menge von Datensätzen immer nur genau *einen bestimmten* Trie gibt, unabhängig davon, in welcher Reihenfolge die Datensätze eingefügt wurden.

In vielen Anwendungen ist ein Nachteil des Radix Trie jedoch, dass die maximale Höhe durch eine große Schlüssellänge m sehr hoch werden kann. Beispiel: Strings, die aus maximal 200 Zeichen kodiert durch je 8 Bit bestehen, haben eine Schlüssellänge $m = 8 \cdot 200 = 1\,600$.

In diesem Fall ist es dann sinnvoll, bei jedem Routerknoten mehr als nur zwei mögliche Nachfolger zu verwalten. Beispielsweise kann ein Routerknoten so viele Nachfolger besitzen, wie es unterschiedliche Zeichen für jede Position eines String-Schlüssels gibt. Dieses Vorgehen führt uns weiter zum Indexed Trie.

Algorithmus 20 Entfernen (**var** p, S, i)

Eingabe: Radix Trie mit Wurzel p; Schlüssel S; Ebene i

```
1: // Entfernt Datensatz mit Schlüssel S aus dem Radix Trie mit Wurzel p
2: falls p == NULL dann {
3:    Datensatz nicht in Trie;
4: } sonst falls p ist Routerknoten dann {
5:    Entfernen (p.next[S_i], S, i + 1);
6:    falls p.next[0] == NULL ∧ p.next[1] ist Datenknoten dann {
7:       p = p.next[1];
8:       // Knoten löschen
9:    } sonst falls p.next[1] == NULL ∧ p.next[0] ist Datenknoten dann {
10:      p = p.next[0];
11:      // Knoten löschen
12:   }
13: } sonst {
14:   // p ist Datenknoten:
15:   falls (p.S_i, ..., p.S_m) == (S_i, ..., S_m) dann {
16:      // Zu löschenden Datensatz gefunden
17:      p = NULL; // Datensatz löschen
18:   } sonst {
19:      Datensatz nicht enthalten;
20:   }
21: }
```

5.2 Indexed Trie

Der *Indexed Trie* dient primär zur Speicherung von Worten, zum Beispiel für eine Rechtschreibprüfung. Die Aufgabe besteht beim Suchen lediglich darin, festzustellen, ob ein Wort „gültig", d.h. in der Datenstruktur enthalten ist oder nicht.

Wir gehen hier als Beispiel davon aus, dass jedes Wort $W = (W_1, W_2, \ldots, W_m)$ aus beliebig vielen Zeichen $m > 0$ des Alphabets $A = \{$'a', 'b',...,'z'$\}$ besteht.

Im Unterschied zum Radix Trie ist nun jeder Knoten ein Feld der Größe $|A|$, das durch die Elemente des Alphabets A indiziert ist und folgende beiden Variablen für jedes $c \in A$ beinhaltet:

- einen Verweis *next* auf einen Nachfolgeknoten und
- eine Boolsche Variable *end*, die angibt, ob ein gültiges Wort mit dem entsprechenden Buchstaben (= Index) an dem Knoten endet.

Es gibt somit keine Blattknoten mehr, in denen die vollständigen Schlüssel gespeichert werden.

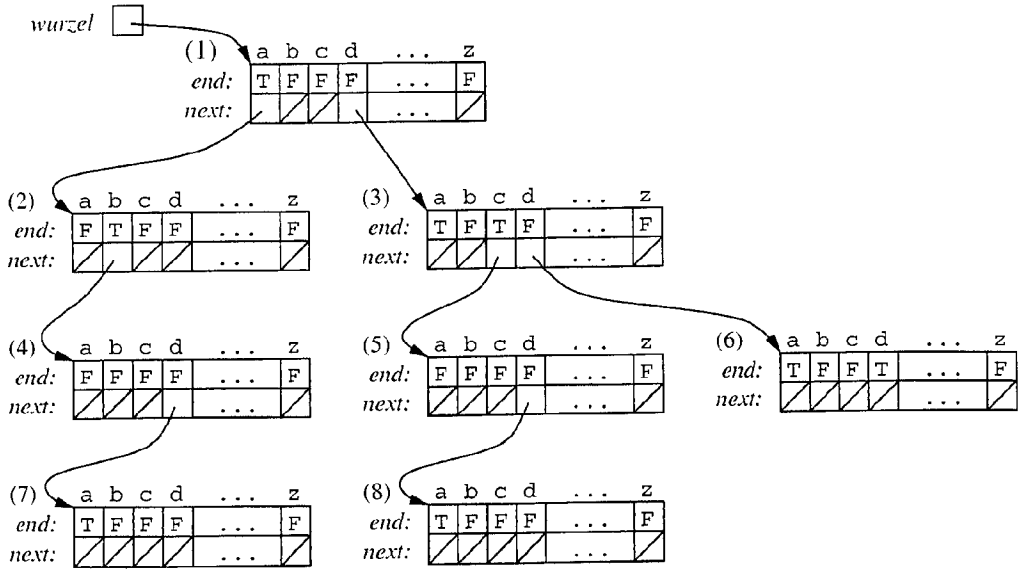


Abbildung 5.2: Ein Indexed Trie, der die Worte {'a', 'ab', 'abda', 'da', 'dc', 'dcda', 'dda', 'ddd'} beinhaltet.

5.2.1 Einfügen

Da nun keine speziellen Blattknoten mehr verwendet werden, muss beim Einfügen eines Wortes $W = (W_1, W_2, \ldots, W_m)$ für jedes Zeichen ein Knoten erzeugt werden, falls dieser nicht bereits existiert. Abbildung 5.2 zeigt ein Beispiel eines Indexed Trie, Algorithmus 21 den Pseudocode für das Einfügen.

Die maximale Höhe des Trie ist immer gleich der Länge des längsten darin enthaltenen Wortes minus eins.
Aufwand: $\Omega(m)$, $O(m \cdot |A|)$, m...Länge des einzufügenden Wortes

5.2.2 Suchen

Algorithmus 22 zeigt den Pseudocode für das Suchen eines Wortes.
Aufwand: $O(m)$

Algorithmus 21 Einfügen (**var** p, W, i)

Eingabe: Indexed Trie mit Wurzel p; Wort W; Buchstabenindex i

```
1: // Fügt Wort W in den Indexed Trie mit Wurzel p ein
2: falls p == NULL dann {
3:    p = Erzeuge Knoten;
4:    für alle c ∈ A {
5:       p[c].next = NULL;
6:       p[c].end = False;
7:    }
8: }
9: falls i == length(W) dann {
10:   p[W_i].end = True;
11: } sonst {
12:   Einfügen (p[W_i].next, W, i + 1);
13: }
```

Algorithmus 22 Suchen (p, W, i)

Eingabe: Indexed Trie mit Wurzel p; Wort W; Buchstabenindex i

Ausgabe: Aussage, ob Wort W im Indexed Trie mit Wurzel p enthalten ist oder nicht

```
1: falls p == NULL dann {
2:    Wort nicht enthalten;
3: }
4: falls i < length(W) dann {
5:    Suchen (p[W_i].next, W, i + 1);
6: } sonst falls p[W_i].end == True dann {
7:    Wort gefunden;
8: } sonst {
9:    Wort nicht enthalten;
10: }
```

5.2.3 Entfernen

Das Entfernen eines Wortes ist in Algorithmus 23 dargestellt.
Aufwand: $O(m \cdot |A|)$

Algorithmus 23 Entfernen (**var** p, W, i)

Eingabe: Indexed Trie mit Wurzel p; Wort W; Buchstabenindex i

```
1: // Entfernt Wort W aus dem Indexed Trie mit Wurzel p, falls es enthalten ist
2: falls p == NULL dann {
3:    Wort nicht enthalten; Abbruch
4: }
5: falls i < length(W) dann {
6:    Entfernen (p[W_i].next, W, i + 1);
7: } sonst falls p[W_i].end == True dann {
8:    // Wort gefunden
9:    p[W_i].end = False;
10: } sonst {
11:    Wort nicht enthalten; Abbruch
12: }
13: falls ∀c ∈ A : p[c].next == NULL ∧ p[c].end == False dann {
14:    p = NULL; // Knoten entfernen
15: }
```

5.2.4 Weitere Anwendungen des Indexed Trie

- Suche in einem (dynamischen) Text gleichzeitig nach mehreren Worten bzw. unterschiedlichen Wortformen. Fortgeschrittenere Varianten des Trie können auch Wortfamilien, wie sie durch reguläre Ausdrücke gegeben sind, repräsentieren. Hier besteht auch ein gewisser Zusammenhang zu endlichen Automaten.

- Ist ein *fixer* Text $T = T_1, \ldots, T_n$ gegeben, für den sehr oft überprüft werden soll, ob ein Wort darin vorkommt, kann ein Trie aus allen möglichen Suffixen von T, d.h. aus $\{(T_1, \ldots, T_n), (T_2, \ldots, T_n), \ldots, (T_n)\}$ aufgebaut werden. Das Überprüfen, ob ein Wort der Länge m Substring von T ist, kann dann in $O(m)$ durchgeführt werden. Die Variablen *end* sind in diesem Fall nicht notwendig. Ein derartiger Trie wird auch als *Suffix-Tree* bezeichnet.

  Beispiel: $T =$ 'ICH_BIN_HIER'.
  Der Suffix-Tree ist der Trie mit folgenden Einträgen:
  'ICH_BIN_HIER', 'CH_BIN_HIER', 'H_BIN_HIER', '_BIN_HIER',
  'BIN_HIER', 'IN_HIER', 'N_HIER', '_HIER', 'HIER', 'IER', 'ER' und 'R'.

5.3 Linked Trie

In den Knoten eines Indexed Trie gibt es oft sehr viele Einträge mit $next == \text{NULL} \wedge end == \text{False}$. Wir nennen solche Einträge *„leere" Einträge*. Dadurch bedingt ist die Datenstruktur nicht sehr kompakt.

Beim Linked Trie wird das Feld für jeden Knoten durch eine lineare Liste ersetzt, die nur *next*- und *end*-Variablen für jene $c \in A$ beinhaltet, für die $next \neq \text{NULL} \vee end \neq \text{False}$. D.h. alle leeren Einträge werden nicht gespeichert. Dafür muss nun zusätzlich immer das jeweilige c vermerkt werden.

Aufwand: Bei größerem $|A|$ und dichten Tries kann durch das notwendige Iterieren durch die linearen Listen der Aufwand für das Suchen deutlich höher als beim Indexed Trie sein: $O(m \cdot |A|)$.

5.4 Suffix Compression

Haben mehrere Worte genau die gleiche Endung (Suffix), so kann man einen Indexed oder Linked Trie durch eine *Suffix Compression* kompaktieren. Dabei speichert man den Subtrie für die gemeinsame Endung nur mehr einmal und verweist darauf mehrfach. Abbildung 5.3 zeigt den Trie aus Abbildung 5.2 nach einer erfolgten Suffix Compression. Die identische Endung `da` der Worte `abda` und `dcda` wird nunmehr durch einen gemeinsamen Subtrie repräsentiert. Der Trie hat damit keine Baumstruktur mehr.

Achtung: Führt man Suffix Compression durch, können die betroffenen Worte nicht mehr individuell verändert werden. Würde man im gezeigten Beispiel das Wort `abdd` einfügen, so würde automatisch auch das Wort `dcdd` im Trie enthalten sein. Suffix Compression ist daher ein Schritt, der erst durchgeführt werden sollte, wenn im Trie keine Änderungen mehr gemacht werden.

Algorithmus 24 zeigt eine abstrakte Form der Suffix Compression auf einem Indexed oder Linked Trie.

Algorithmus 24 Suffix Compression ()

Variable(n): Knoten p und q

1: **solange** zwei Knoten p und q existieren, die gleich sind
(sowohl alle *end*-Flags als auch alle *next*-Verweise!) {
2: Lösche Knoten q;
3: Ersetze im gesamten Trie alle Verweise auf q durch Verweise auf p;
4: }

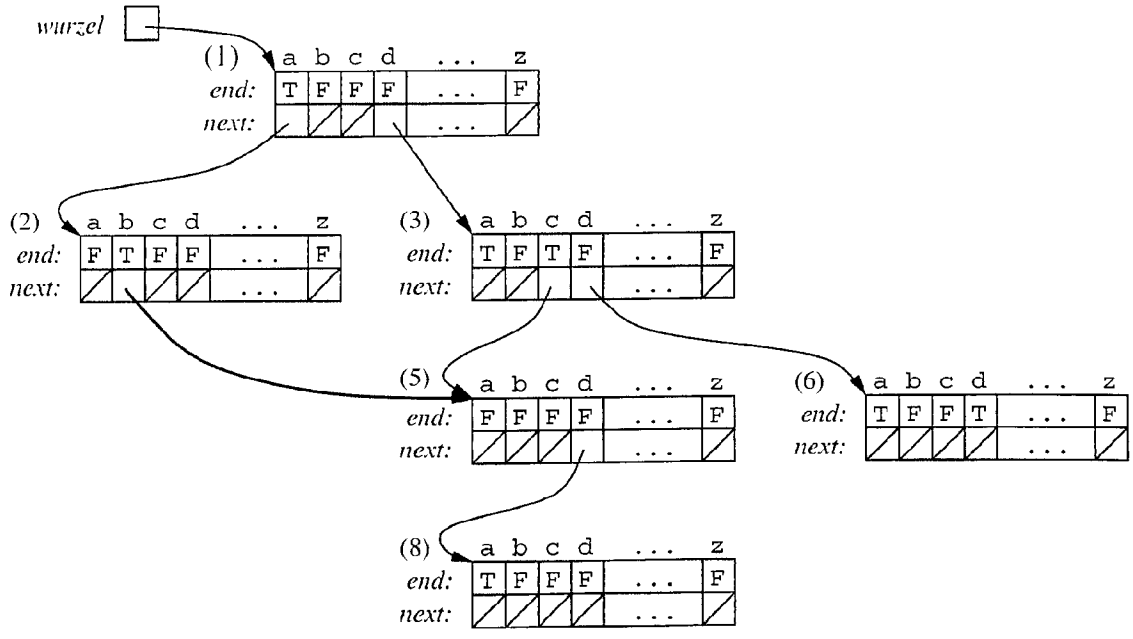


Abbildung 5.3: Der Indexed Trie für die Worte {a, ab, abda, da, dc, dcda, dda, ddd} nach einer Suffix Compression.

5.5 Packed Trie

Für den Indexed Trie gibt es noch eine weitere Möglichkeit der Komprimierung. Die Umwandlung eines Indexed Trie in einen *Packed Trie* ist ein Schritt, der im Allgemeinen eine sehr große Menge an Speicher befreit. Fast alle Felder, die im Indexed Trie leer sind ($end == \text{False} \wedge next == \text{NULL}$), werden eliminiert und es muss dennoch nicht auf die Geschwindigkeitsvorteile des Index-basierten Zugriffs verzichtet werden.

Der Packed Trie wird nicht mehr durch eine Menge von Feldern mit Verweisen dargestellt, sondern durch ein einziges, größeres Feld T, in welchem alle Knoten teilweise ineinander verzahnt gespeichert sind. T ist mit ganzen Zahlen indiziert, und jedes Feldelement besteht aus folgenden drei Komponenten:

- c : Zeichen des Alphabets (= Index eines Knotenelements im Indexed Trie).
- end : Wortende-Markierung.
- $next$: Verweis auf den Beginn des Nachfolgeknotens im Feld (= der Index des ersten Buchstaben des Nachfolgeknotens) oder NULL wenn es keinen Nachfolger gibt.

Beim Aufbau des Packed Trie ist wichtig, dass niemals Elemente kollidieren dürfen, für

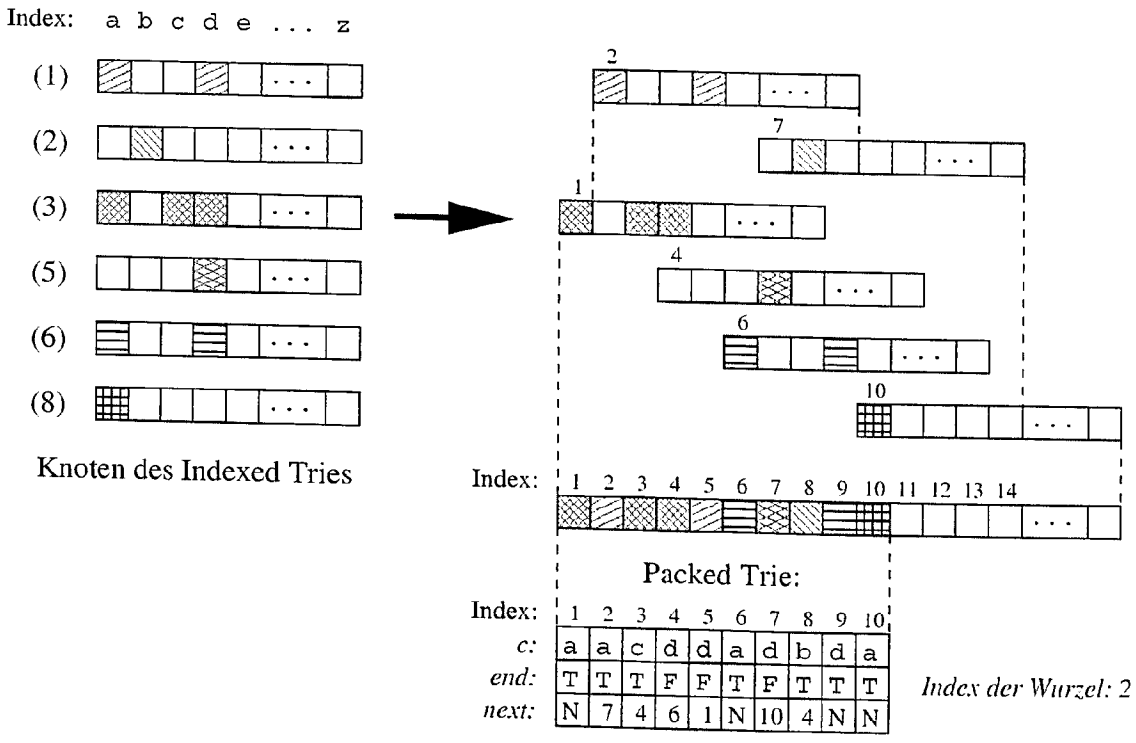


Abbildung 5.4: Der Indexed Trie nach der Suffix Compression wird weiter in einen Packed Trie umgewandelt.

welche im Indexed Trie $end \neq False$ oder $next \neq$ NULL gilt, da sonst Information verloren gehen würde. Weiters dürfen niemals zwei Knoten den gleichen Anfangsindex haben, da sonst nicht zwischen ihren Elementen unterschieden werden kann. Negative Anfangsindizes sind grundsätzlich erlaubt und möglich, wenn ein Knoten mit leeren Elementen beginnt, die nicht gespeichert werden müssen.

Zum Packed Trie gehört auch noch die (globale) Variable *wurzel*, die den Anfangsindex des Wurzelknotens angibt. Der Wurzelknoten muss nämlich nicht unbedingt bei Index 1 beginnen.

Abbildung 5.4 zeigt als Beispiel, wie der Indexed Trie aus Abbildung 5.3 weiter in einen Packed Trie kodiert werden kann. Die leeren Felder am Ende der zusammengefügten Knoten brauchen dabei – wie aus der Abbildung ersichtlich – nicht gespeichert zu werden, sofern bei der Suche eine entsprechende Bereichsüberprüfung der Indizes durchgeführt wird. In diesem Beispiel weist der Packed Trie keine einzige Lücke auf. Das ist aber nicht immer so.

Eine große Menge von Knoten eines Indexed Trie tatsächlich optimal zu komprimieren, d.h. in ein Feld minimaler Länge mit möglichst wenigen Lücken, ist im Allgemeinen ein schwieriges Problem, das praktisch nicht exakt gelöst werden kann. Man erreicht jedoch

mit relativ einfachen greedy-Heuristiken zumeist sehr gute Resultate, da in der Praxis viele Knoten sehr dünn besetzt sind. Oft besitzen sie sogar nur ein einziges belegtes Element und können so als „Lückenfüller" verwendet werden.

In der First-Fit Heuristik, beispielsweise, werden die Knoten nach ihrer Anzahl belegter Elemente absteigend sortiert und dann in dieser Reihenfolge an der ersten geeigneten Stelle im Packed Trie eingeordnet. Das effiziente Auffinden dieser Stelle ist eine Pattern-Matching-Aufgabe, die mit Varianten der Algorithmen zur Textsuche (z.B. Boyer-Moore) effizient gelöst werden kann.

5.5.1 Suchen

Abschließend zeigt Algorithmus 25 die Suche nach einem Wort W im Packed Trie T.
Aufwand: $O(m)$

Algorithmus 25 Suchen (T, W, *wurzel*)
Eingabe: Packed Trie T mit Wurzelindex $wurzel$; Wort W
Ausgabe: Aussage, ob Wort W im Packed Trie T enthalten ist oder nicht
Variable(n): Buchstabenindex i; Index k der aktuellen Position im Packed Trie

```
1: i = 1;   k = wurzel;
2: wiederhole
3:    k = k + ord(W_i) − 1;
4:    falls k < 1 ∨ k > length(T) dann {
5:       // Index außerhalb von T:
6:       Wort nicht enthalten;   Abbruch;
7:    }
8:    falls T[k].c ≠ W_i dann {
9:       Wort nicht enthalten;   Abbruch;
10:   }
11:   falls i == length(W) ∧ T[k].end == True dann {
12:      Wort gefunden;   Abbruch;
13:   }
14:   k = T[k].next;
15:   i = i + 1;
16: bis i > length(W) ∨ k == NULL;
17: Wort nicht enthalten;
```

5.6 Weiterführende Literatur

(1) D. Gusfield: „Algorithms on Strings, Trees, and Sequences", Cambridge University Press, 1997

Kapitel 6

Optimierungsalgorithmen

Mit *Optimierungsproblemen* haben wir uns bereits in *Algorithmen und Datenstrukturen 1* beschäftigt. Vereinfacht gesprochen sind das Aufgabenstellungen, zu denen es eine Vielzahl an möglichen Lösungen gibt. Jeder Lösung ist ein bestimmter Wert (Zielfunktionswert, Kosten etc.) zugeordnet, und wir sind an einer „besten" (*optimalen*) Lösung interessiert, d.h. einer mit minimalem oder maximalem Zielfunktionswert.

Wir werden uns hier im Speziellen mit Algorithmen zur Lösung *kombinatorischer Optimierungsprobleme* beschäftigen. Das sind im Wesentlichen Aufgabenstellungen, die möglicherweise äußerst große, aber *abzählbare* (diskrete, endliche) Mengen an Lösungen besitzen. Derartige Probleme treten in der Praxis sehr häufig auf, und einige haben wir bereits in der Vorlesung *Algorithmen und Datenstrukturen 1* kennengelernt, beispielsweise

- das Finden eines minimalen aufspannenden Baumes in einem Graphen
- das Finden eines kürzesten Weges in einem Graphen
- das 0/1-Rucksackproblem

Es sei auch in Erinnerung gerufen, dass kombinatorische Optimierungsprobleme entsprechend ihrer Lösungsschwierigkeit in unterschiedliche Klassen eingeteilt werden. So gibt es die Klasse der Probleme, für die ein Lösungsalgorithmus mit polynomiellem Aufwand bekannt ist („*in P*"). Beispielsweise gehören die beiden ersten der drei oben genannten Aufgaben in diese Kategorie: Minimale Spannbäume können effizient mit den Algorithmen von Kruskal oder Prim gefunden werden, zur Bestimmung von kürzesten Wegen eignen sich auf dynamischer Programmierung basierende Verfahren wie Dijkstras Algorithmus.

Für die Klasse der *NP-schwierigen* Optimierungsprobleme – dazu gehören beispielsweise das 0/1 Rucksackproblem und das Bin Packing Problem – ist es im Gegensatz dazu äußerst

unwahrscheinlich, dass Lösungsverfahren existieren, die alle möglichen Instanzen eines bestimmten Problems in polynomieller Zeit optimal lösen können. Konkreter: Würde ein polynomieller Algorithmus für eines dieser Probleme existieren, so würde das automatisch polynomielle Algorithmen für alle NP-schwierigen Probleme nach sich ziehen. Leider fällt der Großteil der in der Praxis bedeutsamen kombinatorischen Optimierungsaufgaben in die Klasse der NP-schwierigen Probleme.

NP-schwierige Probleme sind aber nicht unlösbar: In *Algorithmen und Datenstrukturen 1* haben Sie *Enumerationsverfahren* kennengelernt, die systematisch alle Lösungen durchmustern und bewerten, um so ein Optimum zu bestimmen. Dieses Vorgehen bedeutet aber für NP-schwierige Aufgaben einen exponentiellen Zeitaufwand und ist im Allgemeinen daher nur für kleine Probleminstanzen praktikabel.

Oft kann eine naive vollständige Enumeration dramatisch verbessert werden, indem man sie zu einer *beschränkten Enumeration* erweitert: Hierbei werden durch geschickt gewählte Ausschlussbedingungen Teilbereiche des Lösungsraumes, in denen eine optimale Lösung nicht liegen kann, frühzeitig erkannt und nicht weiter untersucht. So ein Vorgehen erlaubt häufig das Lösen erheblich größerer Probleminstanzen. Wir werden uns in diesem Kapitel zunächst ausführlicher mit *Branch-and-Bound* Verfahren beschäftigen, die die Idee der beschränkten Enumeration systematisch verfolgen.

Trotz ihrer Leistungsfähigkeit sind aber auch Branch-and-Bound Algorithmen für schwierige Probleme auf Grund ihrer exponentiellen Laufzeit nur eingeschränkt anwendbar. Häufig muss man sich in der Praxis deshalb mit Algorithmen zufrieden geben, die gute aber nicht unbedingt optimale *Näherungslösungen* liefern und dafür in akzeptabler (polynomieller) Zeit laufen.

In diese Richtung gehend werden wir uns zunächst mit *Approximationsalgorithmen* auseinandersetzen, die Garantien zur Qualität gefundener Lösungen geben können. Vor allem für die Praxis von großer Bedeutung sind ferner *Konstruktions-* und *Verbesserungsheuristiken*, denen wir uns in weiterer Folge widmen werden. Im Speziellen werden Sie die *lokale Suche*, *Simulated Annealing*, *Tabu-Suche* und *evolutionäre Algorithmen* kennenlernen. Die drei zuletzt genannten Verfahren fallen in die allgemeinere Klasse der *Metaheuristiken*, die für viele schwierige Probleme in der Praxis die bisher besten bekannten Ergebnisse liefern.

6.1 Branch-and-Bound

Eine spezielle Form der beschränkten Enumeration, die auch auf das Divide-&-Conquer-Prinzip basiert, ist das Branch-and-Bound Verfahren. Dabei werden durch Benutzung von Primal- und Dualheuristiken möglichst große Gruppen von Lösungen als nicht optimal bzw. gültig erkannt und von der vollständigen Enumeration ausgeschlossen.

Die Idee von Branch-and-Bound ist einfach. Wir gehen hier von einem Problem aus, bei dem eine Zielfunktion minimiert werden soll. (Ein Maximierungsproblem kann durch Vorzeichenumkehr in ein Minimierungsproblem überführt werden.)

- Zunächst berechnen wir z.B. mit einer Heuristik eine zulässige (im Allgemeinen nicht optimale) Startlösung mit Wert U und eine untere Schranke L für alle möglichen Lösungswerte (Verfahren hierzu nennt man auch *Dualheuristiken*).

  Falls $U = L$ sein sollte, ist man fertig, denn die gefundene Lösung muss optimal sein.

- Ansonsten wird die Lösungsmenge partitioniert (*Branching*) und die Heuristiken werden auf die Teilmengen angewandt (*Bounding*).

  Ist für eine (oder mehrere) dieser Teilmengen die für sie berechnete untere Schranke L (*lower bound*) nicht kleiner als die beste überhaupt bisher gefundene obere Schranke (*upper bound* = eine Lösung des Problems), braucht man die Lösungen in dieser Teilmenge nicht mehr beachten. Man erspart sich also die weitere Enumeration.

  Ist die untere Schranke kleiner als die beste gegenwärtige obere Schranke, muss man die Teilmengen weiter zerkleinern. Man fährt solange mit der Zerteilung fort, bis für alle Lösungsteilmengen die untere Schranke mindestens so groß ist wie die (global) beste obere Schranke.

Algorithmus 26 beschreibt dieses Grundprinzip im Pseudocode. Die Hauptschwierigkeit besteht darin, „gute" Zerlegungstechniken und einfache Datenstrukturen zu finden, die eine effiziente Abarbeitung und Durchsuchung der einzelnen Teilmengen ermöglichen. Außerdem sollten die Heuristiken möglichst gute Schranken liefern, damit möglichst große Teilprobleme ausgeschlossen werden können.

6.1.1 Branch-and-Bound für das asymmetrische Traveling Salesman Problem

Der „Großvater" aller Branch-and-Bound Algorithmen in der kombinatorischen Optimierung ist das Verfahren von Little, Murty, Sweeny und Karel (1963) zur Lösung des (asymmetrischen) *Traveling Salesman Problems* (Handlungsreisendenproblem, Rundreiseproblem).

Algorithmus 26 Branch-and-Bound für Minimierung (P)

Eingabe: Problem P
Ausgabe: beste gültige Lösung U (und globale obere Schranke)
Variable(n): Liste offener (Sub-)probleme Π; lokale untere Schranke L'; lokale heuristische Lösung U'

```
1: U = ∞;
2: Π = {P};
3: solange ∃P' ∈ Π {
4:    entferne P' von Π;
5:    // Bounding:
6:    berechne für P' lokale untere Schranke L' mit Dualheuristik;
7:    // Fall L' ≥ U braucht nicht weiter verfolgt zu werden!
8:    falls L' < U dann {
9:       berechne für P' gültige heur. Lösung → obere Schranke U';
10:      falls U' < U dann {
11:         U = U'; // neue bisher beste Lösung gefunden
12:         entferne aus Π alle Subprobleme mit lokaler unterer Schranke ≥ U
13:      }
14:      // Fall L' ≥ U braucht nicht weiter verfolgt zu werden!
15:      falls L' < U dann {
16:         // Branching:
17:         partitioniere P' in Teilprobleme P_1, ..., P_k;
18:         Π = Π ∪ {P_1, ..., P_k};
19:      }
20:   }
21: }
```

Asymmetrisches Traveling Salesman Problem (ATSP)

Gegeben: Gerichteter vollständiger Graph $G(V, E)$ mit $N = |V|$ Knoten, die Städten entsprechen, und eine Distanzmatrix C mit $c_{ii} = +\infty$ und $c_{ij} \geq 0$.

Gesucht: Rundtour $T \subset E$ (Hamiltonscher Kreis) durch alle N Städte, die jede Stadt genau einmal besucht und minimalen Distanzwert aufweist:

$$c(T) = \sum_{(i,j) \in T} c_{ij}$$

Die Menge aller zulässigen Lösungen des TSP auf n Städten umfasst $(n-1)!$ Rundreisen. Beispielsweise wäre eine vollständige Enumeration und Evaluation dieses Lösungsraumes auch auf den leistungsfähigsten Rechnern für z.B. $n \geq 40$ praktisch nicht durchführbar.

Es sei hier auch darauf hingewiesen, dass das TSP wohl „*der* Klassiker" unter den NP-schwierigen kombinatorischen Optimierungsproblemen ist. So gibt es tausende wissenschaftliche Veröffentlichungen, die sich mehr oder weniger direkt dem TSP widmen. Diese große Bedeutung genießt das TSP dabei weniger wegen seiner unmittelbaren Relevanz in der Praxis, sondern weil es sich als eines der wichtigsten Benchmark-Probleme durchgesetzt hat, auf denen neue Lösungsverfahren gerne getestet werden, bzw. weil in vielen praktischen Problemen das TSP ein grundlegendes „Kernproblem" darstellt. In der Praxis sehr häufig auftretende Erweiterungen des TSP sind beispielsweise Routenplanungsaufgaben, wie sie Firmen zu lösen haben um ihre Güter Kunden zuzustellen. Hierbei sind häufig verschiedene weitere Randbedingungen oder Entscheidungsvariablen zu berücksichtigen, die die Aufgabenstellung verkomplizieren.

Lösungsidee mit Branch-and-Bound

Zeilen- und Spaltenreduktion der Matrix C und sukzessive Erzeugung von Teilproblemen beschrieben durch $P(EINS, NULL, I, J, C, L, U)$.

Dabei bedeutet:

$EINS$: Menge der bisher auf 1 fixierten Kanten
$NULL$: Menge der bisher auf 0 fixierten Kanten
I: Noch relevante Zeilenindizes von C
J: Noch relevante Spaltenindizes von C
C: Distanzmatrix des Teilproblems
L: Untere Schranke für das Teilproblem
U: Obere Schranke für das globale Problem

Vorgangsweise

1. Das Anfangsproblem ist

   $$P(\emptyset, \emptyset, \{1, \ldots, n\}, \{1, \ldots, n\}, C, 0, \infty),$$

   wobei C die Ausgangsmatrix ist.
   Setze dieses Problem auf eine globale Liste der zu lösenden Probleme.

2. Sind alle Teilprobleme gelöst $\longrightarrow$ STOP.

   Andernfalls wähle ein ungelöstes Teilproblem

   $$P(EINS, NULL, I, J, C, L, U)$$

   aus der Problemliste.

3. **Bounding:**

   a) **Zeilenreduktion:**
   Für alle $i \in I$:

   Berechne das Minimum der i-ten Zeile von C

   $$c_{ij_0} = \min_{j \in J} c_{ij}$$

   Setze $c_{ij} = c_{ij} - c_{ij_0}\ \forall j \in J$ und $L = L + c_{ij_0}$

   b) **Spaltenreduktion:**
   Für alle $j \in J$:

   Berechne das Minimum der j-ten Spalte von C

   $$c_{i_0 j} = \min_{i \in I} c_{ij}$$

   Setze $c_{ij} = c_{ij} - c_{i_0 j}\ \forall i \in I$ und $L = L + c_{i_0 j}$

   c) Ist $L \geq U$, so entferne das gegenwärtige Teilproblem aus der Problemliste und gehe zu 2.

Die Schritte (d) und (e) dienen dazu, eine neue (globale) Lösung des Gesamtproblems zu finden. Dabei verwenden wir die Information der in diesem Teilproblem bereits festgesetzten Entscheidungen.

   d) Definiere den „Nulldigraphen" $G_0 = (V, A)$ mit
   $A = EINS \cup \{(i,j) \in I \times J \mid c_{ij} == 0\}$

   e) Versuche, mittels einer Heuristik, eine Rundtour in G_0 zu finden. Wurde keine Tour gefunden, so gehe zu 4: Branching.
   Sonst hat die gefundene Tour die Länge L. In diesem Fall:

      e_1) Enferne alle Teilprobleme aus der Problemliste, deren lokale, untere Schranke größer gleich L ist.

      e_2) Setze in allen noch nicht gelösten Teilproblemen $U = L$.

      e_3) Gehe zu 2.

4. **Branching:**

   a) Wähle nach einem Plausibilitätskriterium eine Kante $(i,j) \in I \times J$, die 0 bzw. 1 gesetzt wird.

   b) Definiere die neuen Teilprobleme

b_1) $P(EINS \cup \{(i,j)\}, NULL, I\backslash\{i\}, J\backslash\{j\}, C', L + c_{ij}, U)$
wobei C' aus C durch Streichen der Zeile i und der Spalte j entsteht – von i darf nicht noch einmal hinaus- und nach j nicht noch einmal hineingelaufen werden.
Außerdem achten wir darauf, dass es im Unterproblem nicht sofort zu Kurzzyklen kommen kann. Deswegen setzen wir $c'_{ji} = \infty$.

b_2) $P(EINS, NULL \cup \{(i,j)\}, I, J, C'', L, U)$,
wobei C'' aus C entsteht durch $c_{ij} = \infty$.

Überprüfe für beide Teilprobleme, ob Zeilen- und/oder Spaltenreduktionen möglich sind, um die untere Schranke L zu verbessern. Füge diese Teilprobleme zur Problemliste hinzu und gehe zu 2.

Bemerkungen zum Algorithmus

zu 3. Bounding:

- Hinter der Berechnung des Zeilenminimums steckt die folgende Überlegung: Jede Stadt muss in einer Tour genau einmal verlassen werden. Zeile i der Matrix C enthält jeweils die Kosten für das Verlassen von Stadt i. Und das geht auf keinen Fall billiger als durch das Zeilenminimum c_{ij_0}. Das gleiche Argument gilt für das Spaltenminimum, denn jede Stadt muss genau einmal betreten werden, und die Kosten dafür sind in Spalte j gegeben.
- Schritte (a) und (b) dienen nur dazu, eine untere Schranke zu berechnen (Bounding des Teilproblems). Offensichtlich erhält man eine untere Schranke dieses Teilproblems durch das Aufsummieren des Zeilen- und Spaltenminimums.
- Für die Berechnung einer globalen oberen Schranke können Sie statt (d) und (e) auch jede beliebige TSP-Heuristik verwenden.

zu 4. Branching:

- Ein Plausibilitätskriterium ist beispielsweise das folgende:
  Seien
  $$u(i,j) = \min\{c_{ik} \mid k \in J \setminus \{j\}\} + \min\{c_{kj} \mid k \in I \setminus \{i\}\} - c_{ij}$$
  die minimalen Zusatzkosten, die entstehen, wenn wir von i nach j gehen, ohne die Kante (i,j) zu benutzen. Wir wählen eine Kante $(i,j) \in I \times J$, so dass
  $$c_{ij} = 0 \qquad \text{und} \qquad u(i,j) = \max\{u(p,q) \mid c_{pq} = 0\}.$$

- Dahinter steckt die Überlegung: Wenn die Zusatzkosten sehr hoch sind, sollten wir lieber direkt von i nach j gehen. Damit werden „gute" Kanten beim Enumerieren erst einmal bevorzugt. Wichtig dabei ist, dass $c_{ij} = 0$ sein muss, sonst stimmen die Berechnungen von L und U nicht mehr.

- Bei der Definition eines neuen Teilproblems kann es vorkommen, dass die dort (in *EINS* oder *NULL*) fixierten Kanten zu keiner zulässigen Lösung mehr führen können (weil z.B. die zu *EINS* fixierten Kanten bereits einen Kurzzyklus enthalten). Es ist sicher von Vorteil, den Algorithmus so zu erweitern, dass solche Teilprobleme erst gar nicht zu der Problemliste hinzugefügt werden (denn in diesem Zweig des Enumerationsbaumes kann sich keine zulässige Lösung mehr befinden).

Durchführung des Branch-and-Bound Algorithmus an einem ATSP Beispiel

Die Instanz des Problems ist gegeben durch die Distanzmatrix C:

$$C = \begin{pmatrix} \infty & 5 & 1 & 2 & 1 & 6 \\ 6 & \infty & 6 & 3 & 7 & 2 \\ 1 & 4 & \infty & 1 & 2 & 5 \\ 4 & 3 & 3 & \infty & 5 & 4 \\ 1 & 5 & 1 & 2 & \infty & 5 \\ 6 & 2 & 6 & 4 & 5 & \infty \end{pmatrix}$$

Das Ausgangsproblem:

$$P = (\emptyset, \emptyset, \{1, \ldots, 6\}, \{1, \ldots, 6\}, C, 0, +\infty)$$

Bounding:

Die Zeilenreduktion ergibt:

$$C = \begin{pmatrix} \infty & 4 & 0 & 1 & 0 & 5 \\ 4 & \infty & 4 & 1 & 5 & 0 \\ 0 & 3 & \infty & 0 & 1 & 4 \\ 1 & 0 & 0 & \infty & 2 & 1 \\ 0 & 4 & 0 & 1 & \infty & 4 \\ 4 & 0 & 4 & 2 & 3 & \infty \end{pmatrix}$$

$$L = 1 + 2 + 1 + 3 + 1 + 2 = 10$$

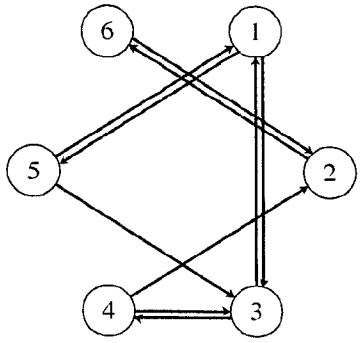


Abbildung 6.1: Der Nulldigraph in Branch-and-Bound

Die Spaltenreduktion ändert hier nichts, da bereits in jeder Spalte das Minimum 0 beträgt.

Nun wird der Nulldigraph aufgestellt (siehe Abbildung 6.1). In diesem Nulldigraphen existiert keine Tour. 6 kann nur über 2 erreicht werden und von 6 müßte wieder auf 2 gegangen werden.

Branching: Wähle Kante (2,6) und setze diese einmal gleich 1 und einmal gleich 0. Dies ergibt zwei neue Teilprobleme.

Das erste neue Teilproblem:

$$P = (\{(2,6)\}, \emptyset, \{1,3,\ldots,6\}, \{1,\ldots,5\}, C', 10, +\infty)$$

$$C' = \begin{pmatrix} \infty & 4 & 0 & 1 & 0 \\ 0 & 3 & \infty & 0 & 1 \\ 1 & 0 & 0 & \infty & 2 \\ 0 & 4 & 0 & 1 & \infty \\ 4 & \infty & 4 & 2 & 3 \end{pmatrix}$$

Die untere Schranke für dieses Teilproblem nach Zeilen- und Spaltenreduktion erhöht sich auf $L = 12$, da die letzte Zeile um 2 reduziert werden kann.

Das zweite neue Teilproblem:

$$P = (\emptyset, \{(2,6)\}, \{1,2,3,4,5,6\}, \{1,2,3,4,5,6\}, C'', 10, +\infty)$$

C'' entsteht aus C nach Setzen von C_{26} auf $+\infty$. Dadurch wird eine Reduktion der 2. Zeile und 6. Spalte um jeweils den Wert 1 möglich, und die untere Schranke ergibt sich zu 12.

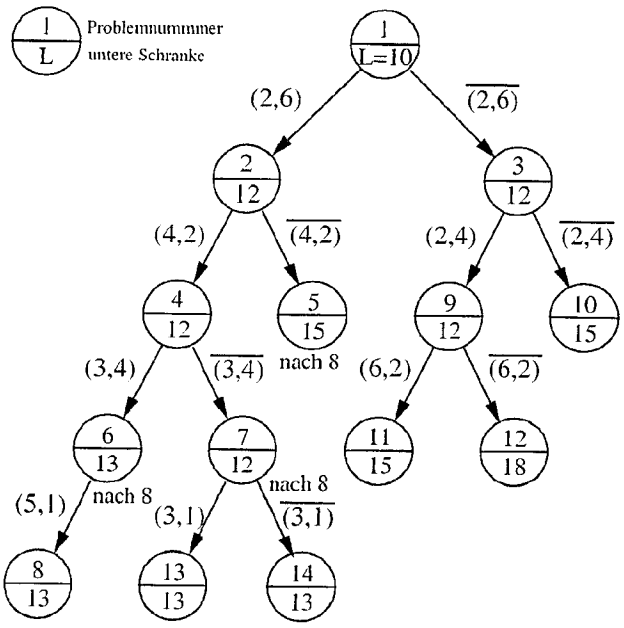


Abbildung 6.2: Möglicher Enumerationsbaum für das Beispiel

Abbildung 6.2 zeigt einen möglichen Enumerationsbaum für dieses Beispiel. (Dabei wird nicht das vorgestellte Plausibilitätskriterium genommen.) Hier wird als nächstes die Kante $(4, 2)$ im Branching betrachtet, was Teilprobleme (4) und (5) erzeugt. Danach wird die Kante $(3, 4)$ fixiert, was die Teilprobleme (6) und (7) begründet. Nach dem Fixieren der Kante $(5, 1)$ wird zum ersten Mal eine globale obere Schranke von 13 gefunden (d.h. eine Tour im Nulldigraphen). Nun können alle Teilprobleme, deren untere Schranke größer gleich 13 ist, aus der Problemliste entfernt werden. In unserem Beispiel wird als nächstes das Teilproblem (3) gewählt, wobei dann über die Kante $(2, 4)$ gebrancht wird, was Teilprobleme (9) und (10) erzeugt. Weil wir mit Teilproblem (10) niemals eine bessere Lösung als 15 erhalten können (untere Schranke), können wir diesen Teilbaum beenden. Wir machen mit Teilproblem (9) weiter und erzeugen die Probleme (11) und (12), die auch sofort beendet werden können. Es bleibt Teilproblem (7), das jedoch auch mit dem Branchen auf der Kante $(3, 1)$ zu „toten" Teilproblemen führt.

Analyse der Laufzeit:

Die Laufzeit ist im Worst-Case exponentiell, denn jeder Branching-Schritt verdoppelt die Anzahl der neuen Teilprobleme. Dies ergibt im schlimmsten Fall bei N Städten eine Laufzeit von 2^N. Allerdings greift das Bounding im Allgemeinen recht gut, so dass man deutlich weniger Teilprobleme erhält. Doch bereits das Berechnen einer Instanz mit $N = 80$ Städten

mit Hilfe dieses Branch-and-Bound Verfahrens dauert schon zu lange. Das vorgestellte Verfahren ist für TSPs mit bis zu 50 Städten anwendbar.

In der Praxis hat sich für die exakte Lösung von TSPs die Kombination von Branch-and-Bound Methoden mit linearer Programmierung bewährt. Solche Verfahren, die als Branch-and-Cut Verfahren bezeichnet werden, sind heute in der Lage in relativ kurzer Zeit TSPs mit ca. 500 Städten exakt zu lösen. Die größten nicht-trivialen bisher exakt gelösten TSPs wurden mit Hilfe von Branch-and-Cut Verfahren gelöst und umfassen ca. 15.000 Städte.

Typische Anwendungen von Branch-and-Bound Verfahren liegen in auch in Brettspielen, wie z.B. Schach oder Springer-Probleme. Die dort erfolgreichsten intelligenten Branch-and-Bound Verfahren funktionieren in Kombination mit parallelen Berechnungen.

6.2 Approximative Algorithmen und Gütegarantien

Ist ein Problem NP-schwierig, so können größere Instanzen im Allgemeinen nicht exakt gelöst werden, d.h. es besteht kaum Aussicht, mit Sicherheit optimale Lösungen in akzeptabler Zeit zu finden. In einer solchen Situation ist man dann auf *Heuristiken* angewiesen, also Verfahren, die zulässige Lösungen für das Optimierungsproblem finden, die aber nicht notwendigerweise optimal sind.

In diesem Abschnitt beschäftigen wir uns mit Heuristiken, die Lösungen ermitteln, über deren Qualität (Güte) man bestimmte Aussagen treffen kann. Man nennt solche Heuristiken mit Gütegarantien *approximative Algorithmen*.

Die „Güte" eines Algorithmus sagt etwas über seine Fähigkeiten aus, optimale Lösungen gut oder schlecht anzunähern. Die formale Definition lautet wie folgt:

Sei A ein Algorithmus, der für jede Probleminstanz P eines Optimierungsproblems Π eine zulässige Lösung mit positivem Wert liefert. Dann definieren wir $c_A(P)$ als den Wert der Lösung des Algorithmus A für Probleminstanz $P \in \Pi$; $c_{\text{opt}}(P)$ sei der optimale Wert für P.

Für Minimierungsprobleme gilt: Falls

$$\frac{c_A(P)}{c_{\text{opt}}(P)} \leq \varepsilon$$

für alle Probleminstanzen P und $\varepsilon > 0$, dann heißt A ein *ε-approximativer* Algorithmus und die Zahl ε heißt *Gütegarantie* von Algorithmus A.

Für Maximierungsprobleme gilt: Falls

$$\frac{c_A(P)}{c_{\text{opt}}(P)} \geq \varepsilon$$

für alle Probleminstanzen P und $\varepsilon > 0$, dann heißt A ein *ε-approximativer* Algorithmus und die Zahl ε heißt *Gütegarantie* von Algorithmus A.

Wir betrachten als Beispiel ein Minimierungsproblem und jeweils die Lösungen einer Heuristik A und die optimale Lösung zu verschiedenen Probleminstanzen P_i:

$$c_A(P_1) = 100 \quad \text{und} \quad c_{\text{opt}}(P_1) = 50$$
$$c_A(P_2) = 70 \quad \text{und} \quad c_{\text{opt}}(P_2) = 40$$

Die Information, die wir daraus erhalten, ist lediglich, dass die Güte der Heuristik A nicht besser als 2 sein kann. Nur, falls

$$\frac{c_A(P_i)}{c_{\text{opt}}(P_i)} \leq 2$$

für alle möglichen Probleminstanzen P_i gilt, ist A ein 2-approximativer Algorithmus.

Bemerkung: Es gilt

- für Minimierungsprobleme: $\varepsilon \geq 1$,
- für Maximierungsprobleme: $\varepsilon \leq 1$,
- $\varepsilon = 1 \Leftrightarrow A$ ist exakter Algorithmus

Im Folgenden betrachten wir verschiedene approximative Algorithmen für das Bin-Packing-Problem und für das Traveling Salesman Problem.

6.2.1 Bin-Packing – Packen von Kisten

Gegeben: Gegenstände $1, \ldots, N$ der Größe w_i; und beliebig viele Kisten der Größe K.
Gesucht: Finde die kleinste Anzahl von Kisten, die alle Gegenstände aufnehmen.

Beispiel: Gegeben sind beliebig viele Kisten der Größe $K = 101$ und 37 Gegenstände der folgenden Größe: 7× Größe 6, 7× Größe 10, 3× Größe 16, 10× Größe 34, und 10× Größe 51.

Die optimale Lösung sieht folgendermaßen aus:

3×	51	34	16		$\sum = 101$
7×	51	34	10	6	$\sum = 101$

Die dargestellte Lösung benötigt nur 10 Kisten. Offensichtlich ist diese auch die optimale Lösung, da jede Kiste auch tatsächlich ganz voll gepackt ist.

First-Fit-Heuristik (FF):
Die Gegenstände werden in beliebiger Reihenfolge behandelt. Jeder Gegenstand wird in die erstmögliche Kiste gelegt, in die er passt.

Unser Beispiel:

1	×	(7×) 6	(5×)10	$\sum = 92$
1	×	(2×)10	(3×)16	$\sum = 68$
5	×	(2×)34		$\sum = 68$
10	×	(1×)51		$\sum = 51$

Das ergibt insgesamt 17 Kisten. Daraus folgt

$$\frac{c_{\mathrm{FF}}(P_1)}{c_{\mathrm{opt}}(P_1)} = \frac{17}{10}$$

für diese Probleminstanz P_1. Dies lässt bisher nur den Rückschluss zu, dass die Gütegarantie der FF-Heuristik auf keinen Fall besser als $\frac{17}{10}$ sein kann.

Analyse der Gütegarantie:
Wir beweisen zunächst eine Güte von asymptotisch 2 für die First-Fit Heuristik.

Theorem: Es gilt: $c_{\mathrm{FF}}(P) \leq 2c_{\mathrm{opt}}(P) + 1$ für alle $P \in \Pi$.
Beweis: Offensichtlich gilt: Jede FF-Lösung füllt alle bis auf eine der belegten Kisten mindestens bis zur Hälfte. Daraus folgt $\forall P \in \Pi$:

$$c_{\mathrm{FF}}(P) \leq \frac{\sum_{j=1}^{N} w_j}{K/2} + 1$$

Da

$$\sum_{j=1}^{N} w_j \leq K c_{\mathrm{opt}}(P)$$

folgt

$$c_{\mathrm{FF}}(P) \leq 2c_{\mathrm{opt}}(P) + 1 \quad \Rightarrow \quad \frac{c_{\mathrm{FF}}(P)}{c_{\mathrm{opt}}(P)} \leq 2 + \frac{1}{c_{\mathrm{opt}}(P)}$$

Man kann sogar noch eine schärfere Güte zeigen. Es gilt (ohne Beweis):

$$\frac{c_{\mathrm{FF}}(P)}{c_{\mathrm{opt}}(P)} < \frac{17}{10} + \frac{2}{c_{\mathrm{opt}}(P)} \qquad \forall P \in \Pi$$

Weiterhin kann man zeigen, dass $\frac{17}{10}$ der asymptotisch beste Approximationsfaktor für die FF-Heuristik ist. Für interessierte Studierende ist der Beweis in Johnson, Demers, Ullman, Garey, Graham in *SIAM Journal on Computing*, vol. 3 no. 4, S. 299-325, 1974, zu finden.

6.2.2 Das symmetrische TSP

Gegeben: ungerichteter vollständiger Graph $K_n = (V, E)$ mit Distanzmatrix $C = (c_{ij})$.
Gesucht: Rundtour kleinsten Gewichts, die alle Städte genau einmal besucht.

Eine beliebte Heuristik für das symmetrische TSP ist die *Spanning Tree Heuristik*. Sie basiert auf der Idee einen minimal aufspannenden Baum zu generieren und daraus eine Tour abzuleiten.

Spanning-Tree-Heuristik (ST)

(1) Bestimme einen minimalen aufspannenden Baum B von K_n.

(2) Verdopple alle Kanten aus $B \rightarrow$ Graph (V, B_2).

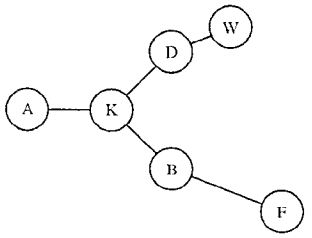
(3) Bestimme eine Eulertour F im Graphen (V, B_2). Eine *Eulertour* ist eine Tour, die jede Kante des Graphen genau einmal enthält. Man kann zeigen, dass ein Graph, in dem jeder Knoten geraden Grad hat, immer eine Eulertour besitzt. Gib dieser Tour eine Orientierung, wähle einen Knoten $i \in V$, markiere i, setze $p = i, T = \emptyset$.

(4) Sind alle Knoten markiert, setze $T = T \cup \{(p, i)\} \rightarrow$ STOP; T ist die Ergebnis-Tour.

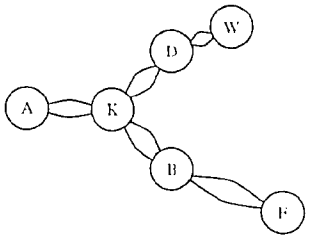
(5) Laufe von p entlang der Orientierung von F bis ein unmarkierter Knoten q erreicht ist. Setze $T = T \cup \{(p, q)\}$, markiere q, setze $p = q$ und gehe zu (4).

Abbildung 6.3 zeigt eine Visualisierung der Spanning-Tree Heuristik anhand eines Beispiels (Städte in Nord-Rhein-Westfalen und Hessen: Aachen, Köln, Bonn, Düsseldorf, Frankfurt, Wuppertal).

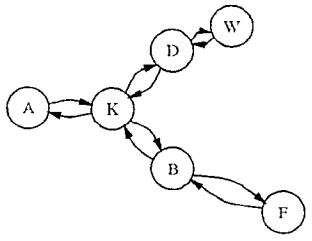
Diskussion der Gütegarantie:
Man kann zeigen: Gibt es ein $\varepsilon > 1$ und einen polynomiellen Algorithmus A, der für jedes symmetrische TSP eine Tour T_A liefert mit $\frac{c_A(P)}{c_{opt}(P)} \leq \varepsilon$, dann ist P=NP.



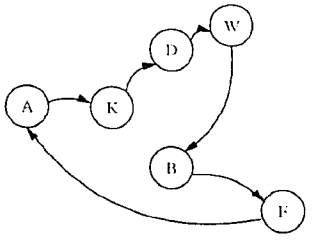
(a) MST



(b) Verdopplung der Kanten



(c) Orientierung



(d) Tour mittels ST-Heuristik

Abbildung 6.3: ST-Heuristik

Theorem: Das Problem, das symmetrische TSP für beliebiges $\varepsilon > 1$ zu approximieren ist NP-schwierig (ohne Beweis).

Doch es gibt auch positive Nachrichten. Wenn die gegebene TSP-Instanz gewisse Eigenschaften aufweist, dann ist es doch approximierbar:

Ein TSP heißt *metrisch* wenn für die Distanzmatrix C die Dreiecks-Ungleichung gilt, d.h. für alle Knoten i, j, k gilt: $c_{ik} \leq c_{ij} + c_{jk}$ und alle $c_{ii} = 0$ sind. Mit anderen Worten kann es nie kürzer sein, von i nach k nicht direkt, sondern über eine andere Stadt j zu laufen. Denkt man beispielsweise an Wegstrecken, so gilt hier oft die Dreiecksungleichung. Gute Nachrichten bringt das folgende Theorem:

Theorem: Für das metrische TSP und die ST-Heuristik gilt:

$$\frac{c_{\mathrm{ST}}(P)}{c_{\mathrm{opt}}(P)} \leq 2 \quad \text{für alle } P \in \Pi.$$

Beweis: Es gilt

$$c_{\mathrm{ST}}(P) \leq c_{B_2}(P) = 2c_B(P) \leq 2c_{\mathrm{opt}}(P).$$

Die erste Abschätzung gilt wegen der Dreiecksungleichung, die letzte, da ein Minimum Spanning Tree die billigste Möglichkeit ist, in einem Graphen alle Knoten zu verbinden.

Anmerkung: Ein *euklidisches TSP* ist ein TSP, bei dem den Städten Punkte in der euklidischen Ebene entsprechen und euklidische Distanzen verwendet werden. Da für euklidische Distanzen auch die Dreiecksungleichung gilt, ist jedes euklidisches TSP auch ein metrisches.

Christophides-Heuristik (CH)

Diese 1976 publizierte Heuristik funktioniert ähnlich wie die Spanning-Tree-Heuristik, jedoch wird Schritt (2) durch folgenden Schritt (2') ersetzt. Dadurch erspart man sich die Verdopplung der Kanten.

(2') Sei W die Menge der Knoten in (V, B) mit ungeradem Grad.

a) Bestimme im von W induzierten Untergraphen von K_n ein perfektes Matching M kleinsten Gewichts

b) Setze $B_2 = B \cup M$

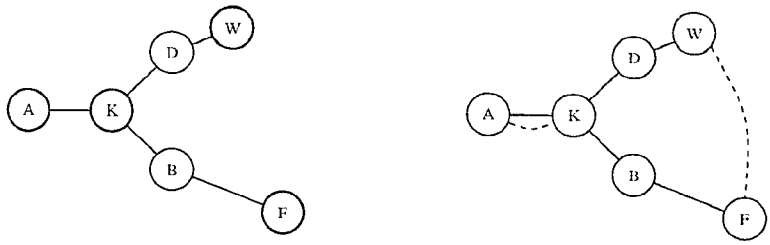
Anmerkungen:

- Das Matching „geht auf", denn $|W|$ ist immer gerade. (Sie haben in *Algorithmen und Datenstrukturen 1* gelernt: In einem ungerichteten Graphen ist die Anzahl der Knoten mit ungeradem Grad immer gerade.)
- Ein *perfektes Matching* in einem Graphen ist eine Kantenmenge, die jeden Knoten genau einmal enthält. Sie ordnet jedem Knoten einen eindeutigen Partnerknoten zu ($\rightarrow$ Alle Knoten werden zu Paaren zusammengefasst).
- Ein solches perfektes Matching mit minimalem Gewicht kann in polynomieller Zeit ($O(n^3)$) gefunden werden.
- Da eine Eulertour existiert, wenn die Knoten eines Graphen alle geraden Grad haben, kann nun in den weiteren Schritten des Algorithmus wiederum sicher eine Rundreise gefunden werden.

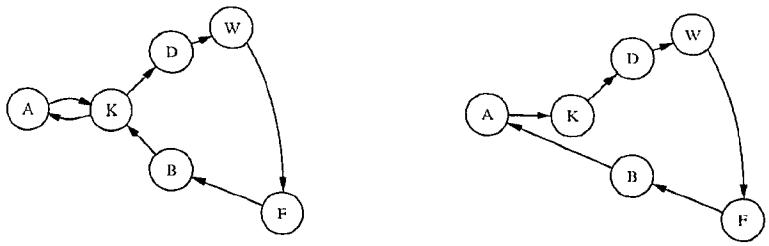
Abbildung 6.4 zeigt die Konstruktion an unserem Beispiel. Dabei betrachten wir in Schritt (2')a) den vollständigen Untergraphen, der von den Knoten A, K, W, F induziert wird.

Analyse der Gütegarantie:
Theorem: Für das metrische TSP und die Christophides-Heuristik gilt:

$$\frac{c_{\mathrm{CH}}(P)}{c_{\mathrm{opt}}(P)} \leq \frac{3}{2} \text{ für alle } P \in \Pi$$



(a) MST (b) B_2



(c) Orientierung (d) Tour

Abbildung 6.4: Die CH-Heuristik an einem Beispiel

Beweis: Seien $i_1, i_2, \ldots, i_{2M}$ die Knoten von B mit ungeradem Grad und zwar so nummeriert, wie sie in einer optimalen Tour T_{opt} vorkommen, d.h. $T_{\text{opt}} = (i_1, \ldots, i_2, \ldots, i_{2M}, \ldots)$. Sei $M_1 = \{(i_1, i_2), (i_3, i_4), \ldots\}$ und $M_2 = \{(i_2, i_3), \ldots, (i_{2M}, i_1)\}$. Es gilt:

$$c_{\text{opt}}(P) \geq c_{M_1}(P) + c_{M_2}(P) \geq c_M(P) + c_M(P)$$

Die erste Abschätzung gilt wegen der Dreiecks-Ungleichung (metrisches TSP). Die zweite Abschätzung gilt, weil M das Matching kleinsten Gewichts ist. Weiters gilt:

$$c_{\text{CH}}(P) \leq c_{B_2}(P) = c_B(P) + c_M(P) \leq c_{\text{opt}}(P) + \frac{1}{2}c_{\text{opt}}(P) = \frac{3}{2}c_{\text{opt}}(P).$$

Bemerkungen:

- Die Christophides-Heuristik (1976) war lange Zeit die Heuristik mit der besten Gütegarantie. 1996 zeigte Arora, dass das Euklidische TSP beliebig nahe approximiert werden kann: die Gütegarantie $\varepsilon > 1$ kann mit Laufzeit $O\left(N^{\frac{1}{\varepsilon-1}}\right)$ approximiert werden. Eine solche Familie von Approximationsalgorithmen wird als *polynomial time approximation scheme* (PTAS) bezeichnet. Für das allgemeine und auch metrische (d.h., die Dreiecksungleichung gilt) TSP kann es jedoch unter der Annahme P≠NP kein PTAS geben.

- Konstruktionsheuristiken für das symmetrische TSP erreichen in der Praxis meistens eine Güte von ca. 10-15% Abweichung von der optimalen Lösung. Die Christophides-Heuristik liegt bei ca. 14%.

6.3 Verbesserungsheuristiken

Die im vorigen Abschnitt gezeigten Heuristiken werden auch als *konstruktive Heuristiken* bezeichnet, da sie neue Lösungen von Grund auf generieren. Im Gegensatz dazu gibt es auch Verfahren, die eine zulässige Lösung als Input verlangen und diese durch lokale Änderungen verbessern. Die Ausgangslösung kann hierbei eine zufällig erzeugte, gültige Lösung sein oder aber durch eine vorangestellte konstruktive Heuristik erzeugt werden.

Wir unterscheiden zwischen (einfacher) lokaler Suche (s. Abschnitt 6.3.1) und sogenannten *Metaheuristiken* wie Simulated Annealing (s. Abschnitt 6.3.2), Tabu-Suche (s. Abschnitt 6.3.3) und evolutionäre Algorithmen (s. Abschnitt 6.3.4).

Derartige Verfahren haben sich für schwierige Optimierungsaufgaben in der Praxis sehr bewährt und liefern häufig ausgezeichnete Ergebnisse, jedoch sei darauf hingewiesen, dass im Gegensatz zu exakten und approximativen Algorithmen im Allgemeinen keinerlei Gütegarantien gegeben werden können.

6.3.1 Lokale Suche

Für die lokale Suche ist die Definition einer *Nachbarschaft* $N(x)$ für jede mögliche Lösung $x \in X$ grundlegend. Eine solche Nachbarschaft wird meist als Menge jener Lösungen definiert, die von einer aktuellen Lösung x aus durch eine bestimmte Art von Änderung – einem sogenannten *Zug* – erreicht werden können. Die gesamte Abbildung aller Lösungen auf die korrespondierenden Mengen von Nachbarlösungen ($X \rightarrow 2^X$) wird auch als *Nachbarschaftsstruktur* bezeichnet und kann als Graph dargestellt werden.

Werden Lösungen beispielsweise durch 0/1-Vektoren repräsentiert, wie das im Rucksackproblem der Fall ist, so könnte $N(x)$ die Menge aller Bitvektoren sein, die sich von x in genau einem Bit unterscheiden. Im Beispiel des Rucksackproblems kann diese Nachbarschaft natürlich auch ungültige Lösungen beinhalten, die dann entsprechend behandelt (z.B. einfach ignoriert) werden müssen. Größere Nachbarschaften können definiert werden, indem man alle Lösungen, die sich in maximal r Bits von x unterscheiden (d.h. deren Hamming-Distanz $\leq r$ ist), als Nachbarlösungen betrachtet. Dabei ist r eine vorgegebene Konstante.

Etwas allgemeiner kann eine sinnvolle Nachbarschaft für ein gegebenes Problem oft

wie folgt durch *Austausch* definiert werden: Entferne aus der aktuellen Lösung x bis zu r Lösungselemente. Sei S die Menge aller verbleibenden Elemente der Lösung. Die Nachbarschaft von x besteht nun aus allen zulässigen Lösungen, die S beinhalten.

Algorithmus 27 Lokale Suche

Eingabe: eine Optimierungsaufgabe
Ausgabe: heuristische Lösung x
Variable(n): Nachbarlösung x' zu aktueller Lösung x

```
1: x = Ausgangslösung;
2: wiederhole
3:    Wähle x' ∈ N(x); // leite eine Nachbarlösung ab
4:    falls x' besser als x dann {
5:       x = x';
6:    }
7: bis Abbruchkriterium erfüllt
8: Ausgabe:
```

Algorithmus 27 zeigt das Prinzip der lokalen Suche. In Zeile 3 wird aus der Nachbarschaft eine neue Lösung x' ausgewählt. Diese Auswahl kann auf folgende Arten erfolgen:

Random neighbor: Es wird eine Nachbarlösung zufällig abgeleitet. Diese Variante ist die schnellste, jedoch ist die neue Lösung x' häufig schlechter als x.

Best improvement: Die Nachbarschaft $N(x)$ wird vollständig durchsucht und die beste Lösung x' wird ausgewählt. Dieser Ansatz ist am zeitaufwändigsten.

First improvement: Die Nachbarschaft $N(x)$ wird systematisch durchsucht, bis die erste Lösung, welche besser als x ist, gefunden wird bzw. alle Nachbarn betrachtet wurden.

Bei lokaler Suche mit der „random neighbor" Auswahlstrategie sind normalerweise wesentlich mehr Iterationen notwendig, um eine gute Endlösung zu finden, als bei den beiden anderen Strategien. Dieser Umstand wird aber oft durch den geringen Aufwand einer einzelnen Iteration ausgeglichen. Welche Strategie in der Praxis am Besten ist, ist vor allem vom konkreten Problem und der eingesetzten Nachbarschaftsstruktur abhängig.

In den Zeilen 4 bis 6 von Algorithmus 27 wird x' als neue aktuelle Lösung akzeptiert, wenn sie besser ist als die bisherige.

Das Abbruchkriterium ist meist, dass in $N(x)$ keine Lösung mehr existiert, die besser als die aktuelle Lösung x ist. Eine solche Endlösung kann durch Fortsetzung der lokalen Suche nicht mehr weiter verbessert werden und wird daher auch als *lokales Optimum in Bezug auf* $N(x)$ bezeichnet. Formal gilt für eine zu minimierende Zielfunktion $f(x)$:

$$x \text{ ist ein lokales Optimum} \quad \leftrightarrow \quad \forall x' \in N(x) \mid f(x') \geq f(x)$$

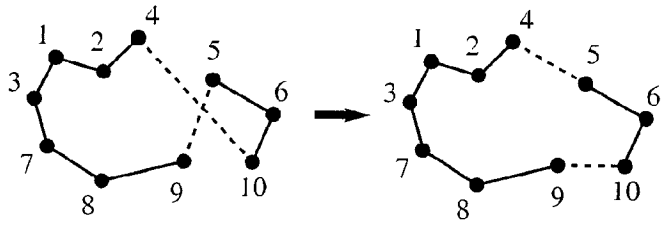


Abbildung 6.5: Prinzip eines Zweieraustausches beim symmetrischen TSP.

Im Allgemeinen ist ein lokales Optimum aber nicht unbedingt ein *globales Optimum*, d.h. die insgesamt beste Lösung, an der wir eigentlich interessiert sind. Jedes globale Optimum ist aber immer auch ein lokales Optimum.

Beispiel: Zweier-Austausch für das Symmetrische TSP (2-Opt)

Wir betrachten das symmetrische Traveling Salesman Problem (TSP), bei dem die Distanzmatrix symmetrisch ist, d.h. $c_{ij} = c_{ji}$. Die sogenannte *2-Opt Nachbarschaft* einer Tour besteht aus allen gültigen Touren, die durch den Austausch von genau zwei Kanten erreicht werden können. D.h., es wird versucht eine Ausgangstour iterativ zu verbessern, indem Paare von Kanten temporär entfernt werden und die so resultierenden zwei Pfade durch neue Kanten verbunden werden. Abbildung 6.5 zeigt ein Beispiel.

Im folgenden Algorithmus wird die Nachbarschaft $N(T)$ einer Tour T systematisch nach einer Verbesserung durchforstet. Die erste Verbesserung wird akzeptiert, was der Auswahlstrategie „first improvement" entspricht. Im Schritt (2) werden speziell nur die Paare von in der Tour *nicht nebeneinander liegenden* Kanten ausgewählt, da das Entfernen von nebeneinander liegenden Kanten nie zu einer neuen Tour führen kann.

(1) Wähle eine beliebige Anfangstour $T = \{(i_1, i_2), (i_2, i_3), \ldots, (i_n, i_1)\}$; sei $i_{n+1} = i_1$

(2) Setze $Z = \{\{(i_p, i_{p+1}), (i_q, i_{q+1})\} \subset T \mid 1 \leq p, q \leq n \wedge p{+}1 < q \wedge (q{+}1) \bmod n \neq p\}$

(3) Für alle Kantenpaare $\{(i_p, i_{p+1}), (i_q, i_{q+1})\}$ aus Z:
Falls $c_{i_p i_{p+1}} + c_{i_q i_{q+1}} > c_{i_p i_q} + c_{i_{p+1} i_{q+1}}$:
setze $T = (T \backslash \{(i_p, i_{p+1}), (i_q, i_{q+1})\}) \cup \{(i_p, i_q), (i_{p+1}, i_{q+1})\}$
gehe zu (2)

(4) T ist das Ergebnis.

Dieser Algorithmus liefert somit eine Tour, die ein lokales Optimum in Bezug auf die 2-Opt Nachbarschaft darstellt. Das muss jedoch nicht die global beste Lösung sein. (Überlegen Sie sich ein Beispiel hierfür als Übungsaufgabe!)

Beispiel: r-Austausch für das Symmetrische TSP (r-Opt)

Bei dieser Verallgemeinerung werden nicht nur Paare von Kanten durch neue Kanten ersetzt, sondern systematisch alle r-Tupel.

(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. Gibt es eine unter diesen, die besser als T ist, wird diese das aktuelle T und gehe zu (2).

(4) T ist das Ergebnis.

Eine Tour, die durch einen r-Austausch nicht mehr verbessert werden kann, heißt in diesem Zusammenhang *r-optimal*.

Bemerkungen:

- Lokale Suche ist eine in der Praxis sehr beliebte Methode, um bereits vorhandene Lösungen oft deutlich und in relativ kurzer Zeit zu verbessern.
- Für das TSP kommt 3-Opt meist sehr nahe an die optimale Lösung (ca. 3-4%). Jedoch erhält man bereits für $N \geq 200$ Städte sehr hohe Rechenzeiten, da der Aufwand pro Iteration bei $O(N^3)$ liegt.
- In der Praxis hat sich für das TSP als beste Heuristik die Heuristik von Lin und Kernighan (1973) herausgestellt, die oft bis zu 1-2% nahe an die Optimallösung herankommt. Bei dieser Heuristik werden generierte Kandidatenlösungen analysiert und r wird aus der aktuellen Situation heraus dynamisch gewählt. Ein solcher Ansatz wird allgemein auch *variable Tiefensuche* genannt.
- Die einfache lokale Suche kann ferner zu einer sog. *variablen Nachbarschaftssuche* erweitert werden, indem strukturiert zwischen unterschiedlichen Nachbarschaftsstrukturen gewechselt wird.

6.3.2 Simulated Annealing

Einfachen lokale Suchverfahren wie 2-Opt ist es oft nicht möglich, mitunter schlechten lokalen Optima zu „entkommen". Eine Vergrößerung der Nachbarschaft kann dieses Problem manchmal lösen, es steigt aber der Aufwand meist allzu stark mit der Problemgröße an. *Simulated Annealing* ist eine allgemeine Erweiterung der lokalen Suche (eine

Algorithmus 28 Simulated Annealing

Eingabe: eine Optimierungsaufgabe
Ausgabe: heuristische Lösung x
Variable(n): Zeit t; aktuelle Temperatur T; Ausgangstemperatur T_{init}; Nachbarlösung x'

```
1:  t = 0;
2:  T = T_init;
3:  x = Ausgangslösung;
4:  wiederhole
5:    Wähle x' ∈ N(x) zufällig; // leite eine Nachbarlösung ab
6:    falls x' besser als x dann {
7:      x = x';
8:    } sonst {
9:      falls Z < e^(-|f(x')-f(x)|/T) dann {
10:       x = x';
11:     }
12:     T = g(T,t);
13:     t = t + 1;
14:   }
15: bis Abbruchkriterium erfüllt
16: Ausgabe: x;
```

Metaheuristik), die es bei skalierbarem Zeitaufwand erlaubt, lokalen Optima grundsätzlich auch zu entkommen, ohne dass die Nachbarschaft vergrößert werden muss.

Abgeleitet wurde dieses Verfahren von dem physikalischen Prozess, ein Metall in einen Zustand möglichst geringer innerer Energie zu bringen, in dem die Teilchen möglichst strukturiert angeordnet sind. Dabei wird das Metall erhitzt und sehr langsam abgekühlt. Die Teilchen verändern anfangs ihre Positionen und damit ihre Energieniveaus sehr stark, mit sinkender Temperatur werden die Bewegungen von lokal niedrigen Energieniveaus weg aber immer geringer.

Algorithmus 28 zeigt das von Kirkpatrick et al. (1983) beschriebene Simulated Annealing. Z ist hierbei eine Zufallszahl $\in [0, 1)$, $f(x) > 0$ die Bewertung der Lösung x.

Die Ausgangslösung kann wiederum entweder zufällig oder mit einer Konstruktionsheuristik generiert werden. In jeder Iteration wird dann ausgehend von der aktuellen Lösung x eine Nachbarlösung x' durch eine kleine zufällige Änderung abgeleitet, was der Auswahlstrategie „random neighbor" entspricht. Ist x' besser als x, so wird x' in jedem Fall als neue aktuelle Lösung akzeptiert. Ansonsten, wenn also x' eine schlechtere Lösung darstellt, wird diese mit einer Wahrscheinlichkeit $p_{accept} = e^{-|f(x')-f(x)|/T}$ akzeptiert. T ist dabei die aktuelle „Temperatur". Dadurch ist die Möglichkeit gegeben, lokalen Optima zu entkommen. Diese Art, Nachbarlösungen zu akzeptieren wird in der Literatur auch *Metropolis-Kriterium* genannt.

Die Wahrscheinlichkeit p_{accept} hängt somit von der Differenz der Bewertungen von x und x' ab – nur geringfügig schlechtere Lösungen werden mit höherer Wahrscheinlichkeit akzeptiert als deutlich schlechtere. Außerdem spielt die Temperatur T eine Rolle, die über die Zeit hinweg kleiner wird: Anfangs werden Verschlechterungen mit größerer Wahrscheinlichkeit erlaubt als später. Wie T initialisiert und in Abhängigkeit der Zeit t (Iteration) vermindert wird, beschreibt das *Cooling-Schema*.

Geometrisches Cooling:

- T_{init}: z.B. $f_{\max} - f_{\min}$
  Sind $f_{\max}$ bzw. $f_{\min}$ nicht bekannt, so werden Schranken bzw. Schätzungen hierfür verwendet.
- $g(T,t) = T \cdot \alpha, \quad \alpha < 1$ (z.B. 0,999)

Adaptives Cooling:

Es wird der Anteil der Verbesserungen an den letzten erzeugten Lösungen gemessen und auf Grund dessen T stärker oder schwächer reduziert.

Als Abbruchkriterium sind unterschiedliche Bedingungen vorstellbar:
Ablauf einer bestimmten Zeit; eine gefundene Lösung ist „gut genug" für die bestimmte Anwendung; oder keine Verbesserung in den letzten k Iterationen.

Damit Simulated Annealing grundsätzlich funktioniert und Chancen bestehen, das globale Optimum zu finden, müssen folgende zwei grundlegende Bedingungen erfüllt sein.

Erreichbarkeit eines Optimums:
Ausgehend von jeder möglichen Lösung muss eine optimale Lösung durch eine Folge von Moves grundsätzlich mit einer Wahrscheinlichkeit größer Null erreichbar sein.

Lokalität der Nachbarschaftsstruktur:
Eine Nachbarlösung wird von ihrer Ausgangslösung durch eine „kleine" Änderung abgeleitet (den Move). Dieser kleine Unterschied muss im Allgemeinen (d.h. mit Ausnahmen) auch eine kleine Änderung der Bewertung bewirken. Sind diese Voraussetzungen erfüllt, spricht man von *hoher Lokalität* – eine effiziente Optimierung ist möglich. Haben eine Ausgangslösung und ihre abgeleitete Nachbarlösung im Allgemeinen große Unterschiede in der Bewertung, ist die Lokalität schwach. Die Optimierung ähnelt dann der Suche „einer Nadel im Heuhaufen" bzw. einer reinen Zufallssuche und ist nicht effizient.

Es folgen ein paar Beispiele für schwierige Probleme und geeignete Moves bzw. Nachbarschaftsdefinitionen.

Beispiel: Traveling Salesman Problem

Für das symmetrische TSP kennen wir bereits die 2-Opt Nachbarschaft, bei der Paare von Kanten ausgetauscht werden und das Resultat wieder eine gültige Tour sein muss. Dieser Move wird häufig auch *Inversion* genannt, da ein Teil der Rundreise ausgewählt und in invertierter Richtung durchlaufen wird.

Es sei an dieser Stelle darauf hingewiesen, dass gültige TSP-Touren auch als Permutationen der Städte dargestellt werden können bzw. jede Permutation der Städte einer gültigen Rundreise entspricht. (Da die Ausgangsstadt nicht fixiert ist, ändert eine Rotation der Permutation nichts an der entsprechenden TSP-Tour.)

Beispielsweise kann die Ausgangs-Tour von Abbildung 6.5 auch durch die Permutation $(1, 2, 4, 10, 6, 5, 9, 8, 7, 3)$ beschrieben werden, und die Tour nach dem Move durch $(1, 2, 4, \mathbf{5}, \mathbf{6}, \mathbf{10}, 9, 8, 7, 3)$; die Teilsequenz $(5, 6, 10)$ wurde somit invertiert.

Für das *asymmetrische* TSP ist dieser Move grundsätzlich auch möglich, jedoch werden im Allgemeinen sehr viele Kanten durch ihre entgegengesetzt gerichteten Gegenstücke ersetzt. Weisen diese stark unterschiedliche Kosten auf, so folgt, dass die Lokalität dieser Nachbarschaftsstruktur relativ gering ist, was wiederum einer effizienten Suche im Wege steht.

Im Falle des asymmetrischen TSPs ist daher der Austausch zweier Städte in der Permutationsdarstellung ein geeigneterer Move. Tauscht man beispielsweise in der Permutation $(1, 2, 4, 10, 6, 5, 9, 8, 7, 3)$ die Städte 2 und 5 so ist das Resultat $(1, \mathbf{5}, 4, 10, 6, \mathbf{2}, 9, 8, 7, 3)$. Hierbei werden immer maximal vier bestehende Kanten durch neue ersetzt.

Zuletzt sei noch angemerkt, dass beide beschriebenen Moves relativ naiv sind, da sie keinerlei Problemwissen wie etwa Kantenkosten ausnutzen. Verbesserungen sind beispielsweise dahingehend möglich, dass versucht wird teurere Kanten mit größerer Wahrscheinlichkeit auszuwählen und diese dann möglichst durch günstigere zu ersetzen. Hierbei muss aber jedenfalls darauf geachtet werden, dass die heuristische Suche nicht allzu leicht in einem schlechten lokalen Optimum hängen bleibt.

Ein Vorteil der beschriebenen einfachen Moves ist, dass sie direkt auch für schwierigere Varianten des TSP, wie dem *blinden TSP*, einsetzbar sind. Bei dieser Variante sind die Kosten einer Tour nicht einfach die Summe fixer Kantenkosten, sondern im Allgemeinen eine nicht näher bekannte oder komplizierte, oft nicht-lineare Funktion. Beispielsweise können die Kosten einer Verbindung davon abhängen, *wann* diese verwendet wird. (Wenn Sie mit einem PKW einerseits in der Stoßzeit, andererseits bei Nacht durch die Stadt fahren, wissen Sie was gemeint ist.)

Beispiel: Mehrdimensionales Rucksack-Problem

Das 0/1-Rucksack-Problem, wie wir es bereits aus *Algorithmen und Datenstrukturen 1* kennen, kann mit exakten Verfahren wie der dynamischen Programmierung auch für sehr große Instanzen im Allgemeinen gut gelöst werden. Zur Erinnerung: Sind die Werte der Gegenstände durch eine Konstante $c_{\max}$ nach oben hin beschränkt (was in der Praxis meist gilt), so ist das 0/1-Rucksack-Problem nicht mehr NP-schwer sondern kann in einer Zeit $O(nc_{\max})$ gelöst werden, wobei n die Anzahl der Gegenstände ist.

In diesem Fall ist der Einsatz von heuristischen Verfahren wie Simulated Annealing daher im Allgemeinen nicht sinnvoll. Es gibt jedoch auch deutlich schwierigere Varianten von Packproblemen, wie das *mehrdimensionale Rucksackproblem*:

Gegeben: n Gegenstände mit Werten $c_i > 0$ und m Ressourcen der Größe $R_1, \ldots, R_m$. Jeder Gegenstand $i = 1, \ldots, n$ verbraucht von jeder Ressource $j = 1, \ldots, m$ eine bestimmte Menge $r_{ij} \geq 0$.

Gesucht: Teilmenge von Gegenständen, beschrieben durch einen Vektor $x \in \{0,1\}^n$, mit maximalem Gesamtwert

$$c(x) = \sum_{i=1}^{n} c_i x_i \,,$$

wobei die vorhandenen Ressourcen nicht überschritten werden dürfen:

$$\sum_{i=1}^{n} r_{ij} x_i \leq R_j \quad \forall j = 1, \ldots, m \,.$$

Natürlich wäre ein sinnvoller Move der bereits bei der lokalen Suche erwähnte Flip eines Bits, um eine Lösungen mit der Hammingdistanz 1 abzuleiten. Allerdings kann es nur allzu leicht sein, dass diese neue Lösung ungültig ist weil sie Resource-Bedingungen verletzt. Derartige ungültige Lösungen zu ignorieren bzw. sehr schlecht zu bewerten ist eine Möglichkeit, aber nicht die effizienteste.

Ein erweiterter Move, der nur zulässige Nachbarlösungen erzeugt, wäre beispielsweise:
Ein zufällig ausgewählter, bisher nicht eingepackter Gegenstand wird zur Lösung hinzugenommen. Überschreitet diese Lösung nun die verfügbare Menge irgendeiner Ressource, wird so lange ein (anderer) eingepackter Gegenstand entfernt, bis die Lösung wieder gültig ist. Die Reihenfolge, in der dabei Gegenstände entfernt werden, wird wiederum zufällig gewählt.

Beispiel: Quadratic Assignment Problem

Gegeben: n Abteilungen und n Standorte eines Betriebs.
$d_{ij} \geq 0, \forall i, j = 1, \ldots, n$, seien die Distanzen zwischen den Standorten.
$f_{ij} \geq 0, \forall i, j = 1, \ldots, n$, sei das Verkehrsaufkommen zwischen den Abteilungen i und j.

Gesucht: Bijektive Zuordnung $p : \{1, \ldots, n\} \rightarrow \{1, \ldots, n\}$ von Abteilungen zu Standorten mit minimalem Gesamtverkehr

$$\sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} d_{p(i),p(j)}$$

Vergleichbar mit dem TSP beschreiben auch hier alle Permutationen von $\{1, \ldots, n\}$ gültige Lösungen. Die 2-Opt Nachbarschaft des symmetrischen TSP ist hier jedoch völlig ungeeignet, da sie hier äußerst geringe Lokalität besitzt: Lösungen sind nun nicht rotationsinvariant; es kommt auf „absolute" Positionen und nicht auf Kanten an.

Ein wesentlich geeigneterer Move ist das Vertauschen zweier zufällig gewählte Abteilungen (vgl. das Vertauschen zweier Städte beim asymmetrischen TSP).

6.3.3 Tabu-Suche

Eine beliebte und in vielen Fällen auch sehr erfolgreiche Alternative zu Simulated Annealing stellt die 1986 von Glover vorgestellte *Tabu-Suche* dar. Auch sie ist eine Erweiterung der einfachen lokalen Suche, die darauf abzielt, lokalen Optima grundsätzlich entkommen zu können, um ein globales Optimum zu finden.

Die Tabu-Suche verwendet ein Gedächtnis über den bisherigen Optimierungsverlauf und nutzt dieses, um bereits erforschte Gebiete des Suchraums (Lösungen) nicht gleich nochmals zu betrachten.

Algorithmus 29 zeigt das Prinzip der Tabu-Suche. Das Gedächtnis ist in diesem Algorithmus in Form der Tabu-Liste TL realisiert. In ihr werden Informationen zu bereits generierten Lösungen gespeichert. Die Nachbarschaft $N(x)$, aus der die Nachfolgelösung bestimmt wird, wird in Zeile 4 durch die Tabuliste so eingeschränkt, dass bereits besuchte Lösungen nicht unmittelbar wieder in Frage kommen. Die Auswahl der konkreten Nachfolgelösung aus der verbleibenden Menge an Nachbarlösungen erfolgt in der Tabu-Suche üblicherweise deterministisch entsprechend der Strategie „best improvement", d.h. die beste Lösung wird ermittelt. Diese wird dann immer als nächste aktuelle Lösung x akzeptiert, auch wenn sie schlechter ist als die Vorgängerlösung. Die insgesamt beste gefundene Lösung wird in x_{best} zurückgeliefert.

Algorithmus 29 Tabu-Suche

Eingabe: eine Optimierungsaufgabe
Ausgabe: beste gefundene Lösung x_{best}
Variable(n): aktuelle Lösung x; Nachbarlösung x'; Tabu-Liste TL; Menge erlaubter Nachbarlösungen X'

1: $x_{\text{best}} = x =$ Ausgangslösung;
2: $TL = \{x\}$;
3: **wiederhole**
4: X' = Teilmenge von $N(x)$ unter Berücksichtigung von TL;
5: x' = beste Lösung von X';
6: füge x' zu TL hinzu;
7: lösche Elemente aus TL, welche älter als t_L Iterationen sind;
8: $x = x'$;
9: **falls** x besser als x_{best} **dann** {
10: $x_{\text{best}} = x$;
11: }
12: **bis** Abbruchkriterium erfüllt
13: Ausgabe: x;

Es gibt zwei Strategien zur Speicherung von Informationen zu generierten Lösungen in der Tabu-Liste.

Explizites Speichern von vollständigen Lösungen

Dieser einfachere Ansatz wurde in Algorithmus 29 angenommen. Zur effizienten Überprüfung, ob eine Lösung aus $N(x)$ in der Tabu-Liste enthalten ist, kann die Tabuliste beispielsweise in Form einer Hashtabelle implementiert werden.

Der Vorteil dieser Methode liegt darin, dass eine generierte Lösung garantiert nicht wieder aufgesucht wird, bis ihr Eintrag aus der Tabu-Liste entfernt wird. Im Algorithmus wäre das über t_L Iterationen. Dem gegenüber stehen zwei entscheidende Nachteile: der hohe Speicherplatzaufwand, der sich aus der Länge t_L der Tabuliste und dem Speicherplatzbedarf einer Lösung ergibt, und der Zeitaufwand zur Überprüfung, ob eine Lösung in der Tabu-Liste enthalten ist bzw. ihrem Eintragen.

Speichern von Lösungsattributen

In diesem Ansatz werden nur einzelne *Attribute* anstatt vollständiger Lösungen in der Tabu-Liste gespeichert. Alle Nachbarlösungen, die ein in der Tabu-Liste gespeichertes Attribut beinhalten, sind damit verboten und dürfen nicht als Nachfolgelösung gewählt werden. Wird ein Zug von einer Lösung x zu einer ausgewählten Nachbarlösung x' vollzogen, so wird als Attribut dasjenige Lösungselement mit seinem Wert in x' gespeichert, welches durch den Zug geändert wurde. Damit wird die Umkehrung des Zuges für t_L Iterationen verhindert.

Der Vorteil dieser Methode ist der wesentlich geringere Speicheraufwand und die raschere Überprüfung von Zügen. Der Nachteil ist, dass mitunter auch Züge zu noch nicht besuchten Lösungen verboten werden, die manchmal entscheidende Verbesserungen bringen könnten. Eine gute Wahl von t_L ist etwas kritischer. Ist t_L zu groß, so kann es sein, dass die Suche zu stark eingeschränkt wird und kaum noch gültige Nachbarlösungen existieren bzw. keine Verbesserungen mehr gefunden werden.

Beispiele

Für das TSP können Kanten als Attribute verwendet werden. Eine durch einen Zug entfernte Kante kommt in die Tabu-Liste und darf erst nach t_L Schritten wieder betrachtet werden. Umgekehrt kann auch eine Tabuliste für die eingefügten Kanten verwaltet werden. Diese dürfen nicht gleich wieder durch andere Kanten ersetzt werden.

Beim mehrdimensionalen Rucksackproblem ist es sinnvoll, den Zustand eines Gegenstandes (eingepackt bzw. nicht eingepackt) als Attribut in der Tabu-Liste zu speichern.

Im Fall des Quadratic Assignment Problems kann als ein Attribut eine Abteilung angesehen werden: Eine gerade zu einem anderen Standort verschobene Abteilung sollte nicht gleich weiter verschoben werden.

6.3.4 Evolutionäre Algorithmen

Unter dem Begriff *evolutionäre Algorithmen* werden eine Reihe von Metaheuristiken (genetische Algorithmen, Evolutionsstrategien, Evolutionary Programming, Genetic Programming, etc.) zusammengefasst, die Grundprinzipien der natürlichen Evolution in einfacher Weise nachahmen. Konkret sind diese Mechanismen vor allem die *Selektion* (natürliche Auslese, „survival of the fittest"), die *Rekombination* (Kreuzung) und die *Mutation* (kleine, zufällige Änderungen).

Ein wesentlicher Unterschied zu Simulated Annealing und der Tabu-Suche ist, dass nun nicht mehr mit nur einer aktuellen Lösung, sondern einer ganze Menge (= *Population*) gearbeitet wird. Durch diese größere *Vielfalt* ist die Suche nach einer optimalen Lösung „breiter" und robuster, d.h. die Chance lokalen Optima zu entkommen ist größer.

Algorithmus 30 zeigt das Schema eines evolutionären Algorithmus.

Ausgangslösungen können wiederum entweder zufällig oder mit schnellen Konstruktionsheuristiken generiert werden. Wichtig ist jedoch, dass sich diese Lösungen unterscheiden und so Vielfalt in der initialien Population gegeben ist.

Algorithmus 30 Grundprinzip eines evolutionären Algorithmus

Eingabe: eine Optimierungsaufgabe
Ausgabe: beste gefundene Lösung
Variable(n): selektierte Eltern Q_s; Zwischenlösungen Q_r

```
1: P = Menge von Ausgangslösungen;
2: bewerte(P);
3: wiederhole
4:    Q_s = Selektion(P);
5:    Q_r = Rekombination(Q_s);
6:    P = Mutation(Q_r);
7:    bewerte(P);
8: bis Abbruchkriterium erfüllt
9: Ausgabe: beste Lösung x ∈ P;
```

Selektion

Die Selektion kopiert aus der aktuellen Population P Lösungen, die in den weiteren Schritten neue Nachkommen produzieren werden. Dabei werden grundsätzlich bessere Lösungen mit höherer Wahrscheinlichkeit gewählt. Schlechtere Lösungen haben aber im Allgemeinen auch Chancen, selektiert zu werden, damit lokalen Optima entkommen werden kann.

Fitness-proportionale Selektion:

Sei $f(x_i) > 0$ die Bewertung (= *Fitness*) jeder Lösung $x_i \in P$, $P = \{x_1, \ldots, x_{|P|}\}$, und gehen wir davon aus, dass diese Funktion maximiert werden soll.

Wir ordnen jeder Lösung x_i eine Selektionswahrscheinlichkeit

$$p_s(x_i) = \frac{f(x_i)}{\sum_{j=1}^{|P|} f(x_j)}$$

zu. Die Auswahl erfolgt nun zufällig entsprechend den Wahrscheinlichkeiten $p_s(x_i)$.

Zu achten ist auf die Verhältnisse zwischen den Selektionswahrscheinlichkeiten der Lösungen in P. Sei $p_s^{\max} = \max\{p_s(x_1), \ldots, p_s(x_{|P|})\}$ und $\overline{p_s} = 1/|P|$. Als *Selektionsdruck* wird das Verhältnis $S = p_s^{\max}/\overline{p_s}$ bezeichnet, d.h. der Faktor, um den die beste Lösung erwartungsgemäß häufiger selektiert wird als eine durchschnittliche.

Ist der Selektionsdruck zu niedrig (alle Lösungen haben sehr ähnliche Bewertungen), werden gute Lösungen zu wenig bevorzugt und der evolutionäre Algorithmus ähnelt einer Zufallssuche. Ist umgekehrt der Selektionsdruck zu hoch, werden gute Lösungen zu stark bevorzugt, die Vielfalt in der Population verringert sich rasch und der evolutionäre Algorithmus konvergiert zu einem lokalen Optimum.

Um den Selektionsdruck steuern zu können, *skaliert* man die Bewertungsfunktion beispielsweise linear über $g(x_i) = a \cdot f(x_i) + b$ mit geeigneten Werten a und b und verwendet dann diese skalierten Fitnesswerte für die Selektion. Skalierung ist auch notwendig, wenn ein Minimierungsproblem vorliegt (a ist dann negativ) oder $f(x_i) < 0$ sein kann.

Tournament Selektion:

Eine andere häufig eingesetzte Variante zur Selektion einer Lösung funktioniert wie folgt:

(1) Wähle aus der Population k Lösungen gleichverteilt zufällig aus (Mehrfachauswahl ist üblicherweise erlaubt).

(2) Die beste der k Lösungen ist die selektierte.

Für die Selektionswahrscheinlichkeit einer Lösung spielen in der Tournament Selektion die relativen Unterschiede der Fitnesswerte keine Rolle mehr, sondern nur mehr der fitnessbasierte Rang. Eine Skalierung ist deshalb nicht erforderlich. Der Selektionsdruck kann über die Gruppengröße k gesteuert werden.

Rekombination

Die Aufgabe der Rekombination ist es, aus zwei selektierten Elternlösungen eine neue Lösung zu generieren. Vergleichbar mit der bereits beim Simulated Annealing beschriebenen Lokalitätsbedingung ist hier wichtig, dass die neue Lösung möglichst ausschließlich aus Merkmalen der Eltern aufgebaut wird, d.h., dass eine starke *Vererbung* erreicht wird.

Mutation

Die Mutation entspricht weitgehend dem Move zu einer Nachbarlösung bei der lokalen Suche bzw. Simulated Annealing. Sie dient meist dazu, neue oder verlorengegangene Merkmale in die Population einzubringen.

Häufig werden die Rekombination und Mutation nicht immer, sondern nur mit einer bestimmten Wahrscheinlichkeit ausgeführt, sodass vor allem gute Lösungen manchmal auch unverändert in die Nachfolgegeneration übernommen werden.

Wir sehen uns im Folgenden konkrete Beispiele an.

Beispiel: Symmetrisches TSP

Rekombination:

Die wesentlichen Merkmale einer TSP-Lösung sind natürlich die verwendeten Kanten. Deshalb soll möglichst ausschließlich aus den Kanten zweier Elterntouren T^1 und T^2 eine neue Tour T erstellt werden, um eine starke Vererbung zu erreichen. Algorithmus 31 zeigt ein mögliches Vorgehen, die sogenannte *Edge-Recombination* (ERX).

Algorithmus 31 Edge-Recombination(T^1, T^2)

Eingabe: Zwei gültige Touren T^1 und T^2
Ausgabe: Neue abgeleitete Tour T
Variable(n): aktueller Knoten v; Nachfolgeknoten w; Kandidatenmenge für Nachfolgeknoten W

1: beginne bei einem beliebigen Startknoten $v = v_0$; $T = \{\}$;
2: **solange** es noch unbesuchte Knoten gibt {
3: Sei W die Menge der noch unbesuchten Knoten, die in $T^1 \cup T^2$ adjazent zu v sind;
4: **falls** $W \neq \{\}$ **dann** {
5: wähle einen Nachfolgeknoten $w \in W$ zufällig aus;
6: } **sonst** {
7: wähle einen zufälligen noch nicht besuchten Nachfolgeknoten w;
8: }
9: $T = T \cup \{(v, w)\}$; $v = w$;
10: }
11: schließe die Tour: $T = T \cup \{(v, v_0)\}$;

Abb. 6.6 zeigt links zwei übereinandergelegte Elterntouren und rechts eine mögliche, durch Edge-Recombination abgeleitete neue Tour, die nur aus Kanten der Eltern besteht.

Im Algorithmus werden neue Kanten, die nicht von den Eltern stammen, nur dann zu T hinzugefügt, wenn W leer ist („edge-fault"). Um die Wahrscheinlichkeit, dass es zu diesen Situationen kommt, möglichst gering zu halten, kann die Auswahl in Schritt (5) wie folgt verbessert werden.

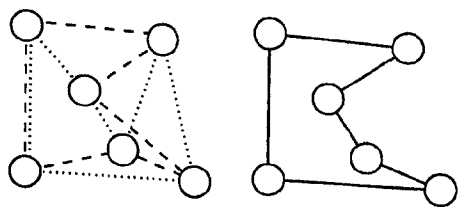


Abbildung 6.6: Beispiel zur Edge-Recombination.

Ermittle für jeden Knoten $w_i \in W$ die Anzahl der gültigen Knoten, die von diesem dann weiter unter Verwendung von Elternkanten angelaufen werden können, d.h. $a_i = |\{u \mid (\{w_i, u\} \in T^1 \cup T^2 \wedge u \text{ noch nicht besucht in } T\}|$.
Wähle ein $w = w_i$ für das a_i minimal ist.

In einer anderen Variante der Edge-Recombination werden die Kantenlängen als lokale, problem-spezifische Heuristik mitberücksichtigt: In Schritt 5 wird entweder immer oder mit höherer Wahrscheinlichkeit die Kante kürzester Länge ausgewählt. Dies führt einerseits zu einer rascheren Konvergenz des evolutionären Algorithmus zu guten Lösungen, kann aber auch ein vorzeitiges „Hängenbleiben" bei weniger guten, lokal-optimalen Touren bewirken.

Mutation: Inversion, d.h. Austausch zweier Kanten

Beispiel: Quadratic Assignment Problem

Rekombination:

Um eine bestmögliche Vererbung zu gewährleisten, wollen wir hier aus zwei Elternlösungen (Abteilungen/Standorte-Zuordnungen) p^1 und p^2 eine neue Lösung p ableiten, in welcher der Standort einer jeden Abteilung immer entweder von p^1 oder von p^2 übernommen wird. Das Ergebnis dabei muss natürlich wiederum eine gültige Permutation sein.

Algorithmus 32 zeigt das sogenannte *Cycle Crossover* (CX). Hierbei wird die erste Abteilung/Standort-Zuordnung vom ersten Elternteil p^1 übernommen. Um das Ziel, dass jede Abteilung entweder von p^1 oder p^2 übernommen wird, zu erreichen, ist man im nächsten Schritt gezwungen, auch die Abteilung a, die sich in p^2 an der ersten Position befindet, vom ersten Elternteil p^1 zu übernehmen. In Schritt 4 wird diese Position ermittelt. Auf diese Weise muß ein ganzer Zyklus von Abteilung/Standort-Zuordnungen übernommen werden, bis man zu einer Abteilung kommt, die bereits übernommen wurde. Alle restlichen Abteilung/Standort-Zuordnungen werden dann ab Zeile 8 von p^2 übernommen.

Ein Beispiel zum Cycle Crossover:

```
p1 =  9 8 2 1 7 4 5 0 6 3
p2 =  1 2 3 4 5 6 7 8 9 0
```

Ein vollständiger Zyklus, beginnend mit '9', wird übernommen:

```
p  =  9 - - 1 - 4 - - 6 -
```

Letztlich werden alle noch leeren Positionen mit den Werten aus p^2 aufgefüllt:

```
p  =  9 2 3 1 5 4 7 8 6 0
```

Mutation: Vertauschen der Standorte zweier Abteilungen.

Algorithmus 32 Cycle Crossover(p^1, p^2)

Eingabe: Zwei Lösungen (Permutationen) p^1 und p^2 der Länge n
Ausgabe: Neue abgeleitete Lösung p

```
1: initialisiere alle p(i) = leer für i = 1,...,n;
2: i = 1;
3: solange p(i) = leer {
4:    p(i) = p^1(i);
5:    a = p^2(i); ermittle j | 1 ≤ j ≤ n ∧ p^1(j) = a;
6:    i = j;
7: }
8: für alle i = 1,...,n {
9:    falls p(i) == leer dann {
10:      p(i) = p^2(i);
11:   }
12: }
```

Beispiel: Mehrdimensionales Rucksackproblem

Initialisierung: $x_i =$ Zufallszahl $\in \{0, 1\} \quad \forall 1 \leq i \leq n$

Rekombination:

Seien x^1 und x^2 die Elternlösungen, x die neue Lösung.

- *1-point crossover:*

  Wähle einen *crossover-point* $c \in \{1, \ldots, n-1\}$ zufällig.

$$\forall i = 1, \ldots, n: \quad x_i = \begin{cases} x_i^1 & \text{für } i \leq c \\ x_i^2 & \text{sonst} \end{cases}$$

- *uniform crossover:*

$$\forall i = 1, \ldots, n: \quad \text{setze per Zufallsentscheidung} \quad x_i = x_i^1 \text{ oder } x_i = x_i^2 .$$

Mutation:

Wähle ein $i \in \{1, \ldots, n\}$ zufällig und setze $x_i = 1 - x_i$.

Problem: Ungültige Lösungen, die Ressourcen überschreiten.

Lösungsmöglichkeiten:

- *Repair-Algorithmus* auf jede generierte (ungültige) Lösung anwenden. Hier z.B. so lange einen zufälligen Gegenstand entfernen, bis die Lösung alle Ressourcenbeschränkungen erfüllt.

- *Bestrafung* zur Bewertung hinzufügen (Lagrange'scher Ansatz)

$$\begin{aligned} \text{maximiere} \quad \sum_{i=1}^{n} x_i \cdot c_i \quad &- \lambda_1 \cdot \max\left(0, \sum_{i=1}^{n} x_i \cdot r_{i1} - R_1\right) - \\ &- \lambda_2 \cdot \max\left(0, \sum_{i=1}^{n} x_i \cdot r_{i2} - R_2\right) - \\ &\ldots \\ &- \lambda_m \cdot \max\left(0, \sum_{i=1}^{n} x_i \cdot r_{im} - R_m\right) \end{aligned}$$

  Adaptive Einstellung der Gewichtungsfaktoren λ_j:

  - Anfang: $\lambda_j = 0 \quad \forall j = 1, \ldots, m$
  - Alle k Generationen:
    Für alle j, für die $\sum_{i=1}^{n} x_i \cdot r_{ij} > R_j$: setze $\lambda_j = \lambda_j + \alpha \quad (\alpha > 0)$.

Abschließend sei zu evolutionären Algorithmen noch angemerkt, dass sie mit anderen Heuristiken und lokalen Verbesserungstechniken sehr effektiv kombiniert werden können. So ist es u.a. möglich

- ...Ausgangslösungen mit anderen Heuristiken zu erzeugen.
- ...in der Rekombination und Mutation Heuristiken einzusetzen (z.B. können beim TSP kostengünstige Kanten mit höherer Wahrscheinlichkeit verwendet werden).
- ...alle (oder einige) erzeugte Lösungen mit einem anderen Verfahren (z.B. 2-Opt) noch lokal weiter zu verbessern.

Auch können die Vorteile paralleler Hardware gut genutzt werden.

Weiterführende Literatur

- F. Glover und G. Kochenberger: „Handbook of Metaheuristics", Springer, 2003
- H. Hoos und T. Stützle: „Stochastic Local Search", Morgan Kaufman, 2004
- E. Aarts und J.K. Lenstra: „Local Search in Combinatorial Optimization", John Wiley & Sons, Chichester, 1997
- Z. Michalewicz: „Genetic Algorithms + Data Structures = Evolution Programs", Springer, 1996

Anhang A

Übungsbeispiele

Aufgabe 1 *Suchen in Texten – Knuth-Morris-Pratt*

Betrachten Sie den Algorithmus von *Knuth-Morris-Pratt* (KMP),
den Text T = BABABBABBABABBABAB und das Muster P = BABBABAB.

1. Geben Sie für das Muster P das $next[]$ Array an.
2. Suchen Sie mittels KMP nach allen Vorkommnissen des Musters P im Text T. Visualisieren Sie jeden Schritt und markieren Sie dabei deutlich die Mismatches.

Aufgabe 2 *Suchen in Texten – Boyer-Moore*

Betrachten Sie den Algorithmus von *Boyer-Moore* (BM),
den Text T = ABABCCAAACBAAACBAA und das Muster P = AAACBAA.

1. Geben Sie das $last[]$ Array an.
2. Geben Sie das $suffix[]$ Array für P an.
3. Suchen Sie mittels BM nach allen Vorkommnissen des Musters P im Text T. Visualisieren Sie in jedem Schritt, wo das Muster angelegt wird. Geben Sie an
   - welche Zeichen gematcht werden,
   - welches Zeichen gegebenenfalls den Missmatch verursacht, und
   - mit welcher Regel ($last$ oder $suffix$) verschoben wird.

Aufgabe 3 *Miller-Rabin-Primzahltest*

Beim Miller-Rabin-Primzahltest wird eine Funktion „Zeuge(a, n)" benutzt, die für zwei positive ganze Zahlen a und n prüft, ob

$$a^{n-1} \equiv 1 \bmod n$$

gilt. Führen Sie den Algorithmus für $a = 11$ und $n = 161$ durch. Geben Sie für jeden Schleifendurchlauf den Wert Ihrer Variablen an. Was sagt das Ergebnis aus?

Aufgabe 4 *Randomisierte Skiplisten*

Fügen Sie in eine anfangs leere Skipliste S_r die folgenden Elemente in der vorgegebenen Reihenfolge ein: $28, 47, 11, 27, 13, 53, 65, 19, 49$. Verwenden Sie hierbei die folgenden zufällig gewählten Höhen: $1, 2, 0, 2, 1, 3, 0, 0, 0$. Zeichnen Sie die resultierende Skipliste.

Suchen Sie nun nach der Zahl 30. Geben Sie die Anzahl der Schlüsselvergleiche (= Länge der Suchpfade, Vergleiche mit ∞ zählen mit!) an, die jeweils nötig sind.

Aufgabe 5 *Modifizierte Perfekte Skiplisten*

Wir betrachten eine Variation der in der Vorlesung betrachteten perfekten Skipliste, bei welcher auf Niveau i anstatt *einem* Element auf Niveau $i-1$ (wie bei der perfekten Skipliste aus der Vorlesung) nun *zwei* Elemente auf Niveau $i-1$ übersprungen werden. Es gilt also für jeden echten Container, dass jeder 3^i-te Container einen Zeiger auf den 3^i Positionen weiter hinten stehenden Container (bzw. auf das Ende) hat.

Zeichnen Sie eine solche modifizierte perfekte Skipliste, die die Elemente 11, 19, 29, 8, 20, 13, 22, 10, 23 und 2 enthält.

Berechnen Sie, wieviele Zeiger diese Skipliste enthält, wenn in ihr 243 Elemente (plus Kopf- und Endelement) existieren. Vergleichen Sie diese Anzahl mit der in der Vorlesung besprochenen perfekten Skipliste.

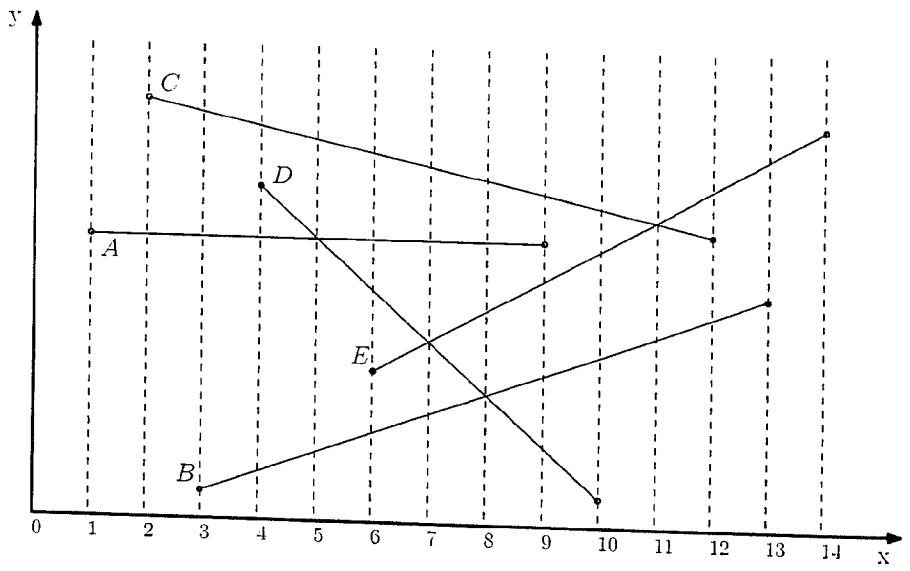
Aufgabe 6 *Modifizierte Perfekte Skiplisten – Pseudocode*

Geben Sie einen Algorithmus in ausführlichem Pseudocode an, der effizient ein Element in die modifizierte perfekte Skipliste aus Aufgabe 5 einfügt. Wie lange dauert das Einfügen in

Ihrem Algorithmus durchschnittlich im Vergleich zur perfekten Skipliste aus der Vorlesung? Begründen Sie Ihre Antwort ausführlich und analysieren Sie die Laufzeit in O-Notation.

Aufgabe 7 *Geometrische Algorithmen – Schnitt von Liniensegmenten*

Der aus der Vorlesung bekannte Algorithmus zum Schnitt von allgemeinen Liniensegmenten wird auf das folgende Beispiel ausgeführt:



Führen Sie den aus der Vorlesung bekannten Algorithmus zum Schnitt von allgemeinen Liniensegmenten für die folgende Menge von Linien aus. Geben Sie dabei für jeden Zeitpunkt (0 ... 14) im Verlauf der Bewegung der Scan-Linie den Zustand der Scan-Line-Status Struktur an.

Wann werden die Schnittpunkte $A \cap D$, $B \cap D$, $C \cap E$ und $D \cap E$ in die Ereignisstruktur ES eingefügt?

Aufgabe 8 *Geometrische Algorithmen – Sichtbarkeitsproblem*

Wir betrachten das folgende Sichtbarkeitsproblem: Zwei *horizontale* Liniensegemente s und s' in einer gegebenen Menge *horizontaler* Liniensegmente S sind gegenseitig sichtbar, wenn es eine *vertikale* Gerade gäbe, die s und s', aber kein weiteres Liniensegment der Menge S zwischen den beiden Liniensegmenten s und s' schneiden würde. Es soll nun nach allen gegenseitig sichtbaren *horizontalen* Liniensegmentpaaren aus der Menge S gesucht werden. Sämtliche x-Werte von Anfangs- und Endpunkten aller Liniensegmente sind paarweise verschieden.

Schreiben Sie unter Zuhilfenahme des Scan-Line Prinzips einen effizienten Algorithmus in Pseudocode, der dieses Problem löst.

Aufgabe 9 *Geometrische Algorithmen – Mehrdimensionale Bereichssuche*

Gegeben ist die folgende Punktmenge in 3D:

$P_0 = (40, 23, 40), P_1 = (32, 41, 41), P_2 = (99, 40, 11), P_3 = (11, 11, 49), P_4 = (27, 76, 23)$
$P_5 = (76, 32, 68), P_6 = (49, 49, 27), P_7 = (23, 68, 32), P_8 = (41, 27, 99), P_9 = (68, 99, 76)$

Zeichnen Sie einen balancierten Baum für die 3-dimensionale Bereichssuche, dessen Knoten den angegebenen Punkten entsprechen. Fangen Sie mit der x-Koordinate an, dann y und schließlich z.

Der Median einer Folge $\langle a_l, \ldots, a_r \rangle$ ist a_m mit $m = \lfloor \frac{l+r}{2} \rfloor$.

Aufgabe 10 *Geometrische Algorithmen – Mehrdimensionale Bereichssuche*

Markieren Sie alle Knoten in Ihrem Baum aus Aufgabe 9, die bei der Bereichssuche nach dem Bereich D vom Bereichssuche-Algorithmus besucht werden. Der Bereich D ist ein 3-dimensionaler Quader gegeben durch die beiden Eckpunkte $d_1 = (30, 20, 25)$ und $d_2 = (50, 39, 55)$.

Aufgabe 11 *Prioritätswarteschlangen*

Welchen Aufwand in Θ-Notation verursachen die Operationen Get_Min, Del_Min und Insert, falls Sie für deren Realisierung binäre Suchbäume verwenden? Begründen Sie Ihre Antwort.

Aufgabe 12 *Externe Array-Heaps*

Gegeben sei ein externer Array-Heap H. Im Skriptum wird beschrieben, wie der Heap H für eine effiziente Implementierung der Operation Del_Min in zwei Heaps H_1 und H_2 aufgeteilt wird.

- Welche Datenstruktur verwenden Sie für den Heap H_1?
- Welche Datenstruktur verwenden Sie für den zweiten Heap H_2?

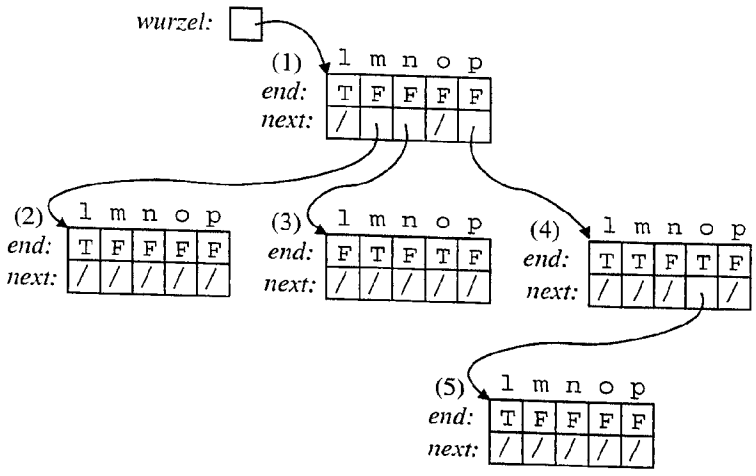
- Welche Datenstruktur verwenden Sie für die Slots und Schichten von externen Array-Heaps? Bedenken Sie, dass von einer geschickten Wahl der Datenstruktur besonders die Effizienz der Delete_Min- und Load-Operationen abhängt.

Aufgabe 13 *Cache-Optimale Algorithmen*

Untersuchen Sie das Cache-Verhalten des klassischen Mergesort-Algorithmus, wie er z.B. im *Algorithmen und Datenstrukturen 1* Skriptum beschrieben wurde. Schlagen Sie eine Modifikation von Mergesort vor, um *Cache-Misses* zu reduzieren (bei vorgegebener Cache-Kapazität C).

Aufgabe 14 *Indexed Tries*

Gegeben seien ein Alphabet Σ={'l', 'm', 'n', 'o', 'p'} und folgender Indexed Trie:



1. Geben Sie alle Wörter an, die der oben dargestellte Indexed Trie enthält.
2. Führen Sie Suffix Compression im oben dargestellten Indexed Trie durch. Kennzeichnen Sie die Änderungen deutlich!
3. Aus dem resultierenden Indexed Trie mit Suffix Compression soll nun ein Packed Trie erstellt werden. Verwenden Sie dazu die Greedy-Heuristik aus der Vorlesung bzw. aus dem Skriptum. Zeigen Sie dabei mit Hilfe einer kleinen Graphik (wie in der Vorlesung bzw. im Skriptum), auf welche Weise die Knoten gepackt werden, und zeichnen Sie den Packed Trie.

Aufgabe 15 *Radix Tries*

Gegeben sei die Menge $A = \{0, 1, \ldots, 9, A, B, C, D, E, F\}$ der Hexadezimalziffern. Um sie in einem Radix Trie speichern zu können, betrachten wir sie jeweils in ihrer üblichen Binärkodierung: $0 = 0000, 1 = 0001, \ldots, F = 1111$.

- Speichern Sie die Hexadezimalziffern $0, 4, 8, 6, 9, A, C, D$ in einem Radix Trie. Fertigen Sie eine Zeichnung des entstehenden Radix Tries an.
- Entfernen Sie die Ziffer A aus dem Radix Trie und fügen Sie stattdessen die Ziffer 1 ein. Fertigen Sie auch eine Zeichnung des dadurch entstehenden modifizierten Radix Tries an.

Aufgabe 16 *Branch-and-Bound*

Wir betrachten das Rucksack-Problem, das sich für den Weihnachtsmann mit vier verschiedenen Geschenken stellt. Die folgende Tabelle gibt jeweils für jedes der Geschenke G_1, G_2, G_3 und G_4 sein Gewicht und seinen Wert an. Der Rucksack des Weihnachtsmanns kann ein Gewicht von 100 Einheiten transportieren.

Geschenk	G_1	G_2	G_3	G_4
Gewicht g_i	50	30	25	15
Wert w_i	10	8	6	1

Sei $F = \langle F_1, \ldots, F_4 \rangle$ die Folge der vier Gegenstände, absteigend sortiert nach dem Wert des Quotienten w_i/g_i.

In einem *Branch-and-Bound* Algorithmus wird das Problem rekursiv in Teilprobleme zerlegt, indem man jeweils in der durch F festgelegten Reihenfolge die Werte der Entscheidungsvariablen x_i der Gegenstände fixiert.

Dadurch ergibt sich ein Branch-and-Bound Baum. Für jeden Knoten v des Baums werden eine obere und eine untere Schranke für die im Teilbaum mit Wurzel v enthaltene beste Lösung bestimmt. Seien nun einige der Variablen x_1 bis x_k für $1 \leq k \leq 4$ fixiert, wobei x_i für $1 \leq i \leq k$ den Wert 1 hat, falls Geschenk F_i eingepackt wird, und 0 sonst.

Die untere Schranke erhält man, indem man alle Gegenstände, deren Variable noch nicht festgelegt ist, in der durch F gegebenen Reihenfolge durchläuft und den aktuellen Gegenstand einpackt, falls noch Platz im Rucksack ist. Man erhält so eine gültige Lösung für das Problem, deren Wert die untere Schranke ist.

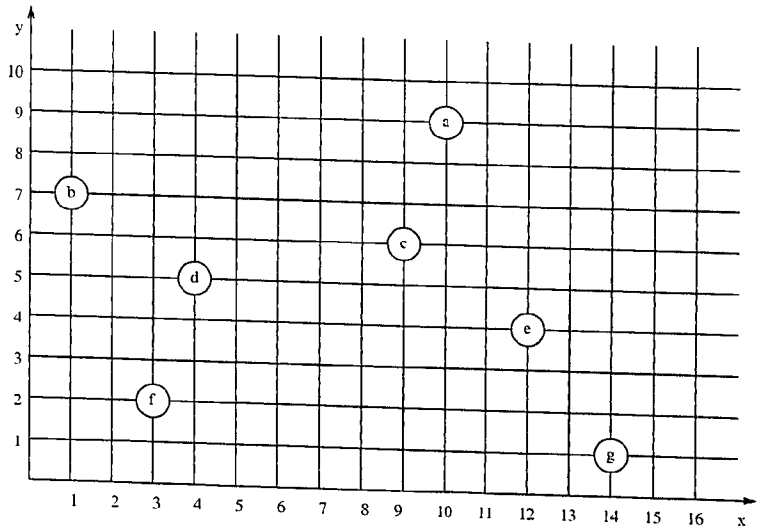
Eine obere Schranke wird berechnet, indem man ebenfalls alle Gegenstände, deren Variable noch nicht festgelegt ist, in der durch F gegebenen Reihenfolge durchläuft. Man packt aber nun alle Gegenstände ein, bis man zu dem ersten Gegenstand F_i kommt, der nicht mehr in den Rucksack passt. Sei r die noch freie Kapazität des Rucksacks. Dann zählt man $r \cdot g_i/w_i$ noch zu dem Wert der Gegenstände im Rucksack dazu, d.h. dieser letzte Gegenstand wird „anteilsmäßig" eingepackt.

Zeichnen Sie den Branch-and-Bound Baum für das Problem und schreiben Sie an jeden Knoten den Wert der unteren und oberen Schranken. Nehmen Sie an, dass beim Fixieren einer neuen Variable immer zuerst das Kind betrachtet wird, bei dem die Variable auf 1 gesetzt wird, und dass Sie den Baum mittels Tiefensuche durchmustern. Achten Sie darauf, dass Sie nur jene Teile des Baums zeichnen, die zur Berechnung der optimalen Lösung nötig sind.

Aufgabe 17 *Spanning Tree Heuristik für das TSP*

Das folgende Diagramm stellt die Knoten eines vollständigen, ungerichteten Graphen dar; die Gewichte der einzelnen Kanten entsprechen jeweils der Manhattan Distanz zwischen den beiden adjazenten Knoten. Wenden Sie die Spanning Tree Heuristik an, um eine möglichst gute Lösung zum Traveling Salesman Problem (TSP) in diesem Graphen zu finden. Beschreiben und veranschaulichen Sie dabei alle Schritte.

Kann die Wahl vom Startknoten das Endergebnis beeinflussen?



Aufgabe 18 *Christophides-Heuristik für das TSP*

Wenden Sie nun die Christophides-Heuristik auf den Graphen in Aufgabe 17 an, um einen möglichst kurzen Rundweg zu finden. Beschreiben und veranschaulichen Sie dabei wieder alle notwendigen Schritte.

Aufgabe 19 *2-OPT*

Überlegen Sie sich zum Euklidischen TSP ein Beispiel, in dem die Verbesserungsheuristik 2-OPT eine Tour nicht weiter verbessern kann, obwohl diese noch nicht global optimal ist.

Aufgabe 20 *r-OPT*

Analysieren Sie die Laufzeit einer Iteration von r-OPT für das Symmetrische TSP auf einem vollständigen Graphen mit N Knoten:

1. Wie groß ist die Menge Z in Abhängigkeit von r?
2. Bis zu wieviele unterschiedliche Möglichkeiten kann es geben, um nach dem Entfernen von r Kanten die entstandenen r Pfade wieder zu einer gültigen Tour zu verbinden?

Geben Sie eine daraus resultierende möglichst scharfe obere Schranke der Laufzeit einer Iteration (eines möglichen Verbesserungsschritts) von r-OPT in O-Notation an.

Aufgabe 21 *Lokale Suche*

Für eine Katastrophenhilfe soll die Lieferung von schwerem Gerät mittels Transporthubschrauber optimiert werden. Konkreter stehen $m = 8$ Hubschrauber für den Transport von n Geräten mit Gewichten $w_1, \ldots, w_n \in \mathbb{R}^+$ zu Verfügung. Für möglichst sichere und gleich schnelle Flüge sollen alle Geräte so auf die Hubschrauber aufgeteilt werden, dass das Ladegewicht der einzelnen Flüge möglichst ausgeglichen ist.

Eine mögliche Lösung dieses Zuteilungsproblems sei durch m Mengen $Z_1, \ldots, Z_m \subset \{1, \ldots, n\}$ repräsentiert, wobei die Menge Z_i, $i = 1, \ldots, m$, alle Gegenstände beinhaltet, die mit Hubschrauber i befördert werden. Es muss jedenfalls gelten:

$$Z_i \cap Z_j = \emptyset \quad \forall i \neq j, \qquad \bigcup_{i=1,\ldots,m} Z_i = V.$$

Das Gesamtgewicht (Ladegewicht) der einem Hubschrauber i zugewiesenen Gegenstände ist

$W(Z_i) = \sum_{j \in Z_i} w_j$. Um ein über alle Hubschrauber möglichst ausgegliches Ladegewicht zu erreichen, minimieren wir das größte auftretende Ladegewicht:

$$\min \quad \max_{i=1,\ldots,m} W(Z_i)$$

1. Definieren Sie eine sinnvolle Nachbarschaftsstruktur für eine lokale Suche durch die genaue Angabe erlaubter Züge. Achten Sie darauf, dass immer nur gültige Lösungen erzeugt werden. (Eine Beschreibung in einfachen Worten reicht.)

2. Wieviele Nachbarlösungen besitzt eine Lösung in der von Ihnen definierten Nachbarschaftsstruktur?

3. Geben Sie einen ausführlichen Pseudocode für eine lokale Suche an, die die zuvor definierte Nachbarschaftsstruktur verwendet und der *best improvement* Strategie folgt.

   Um die Erzeugung einer gültigen Ausgangslösung brauchen Sie sich hierbei nicht zu kümmern.

Aufgabe 22 *Simulated Annealing*

Sie wollen mit Hilfe von Simulated Annealing in einem Graphen $G = (V, E)$, in dem jeder Kante eine Länge zugewiesen ist, einen möglichst *langen* Pfad finden, der jeden Knoten des Graphen höchstens ein Mal besucht. Dazu benötigen Sie eine Nachbarschaftsfunktion, die aus einer gültigen Lösung eine neue gültige Lösung erzeugt. Diese Funktion soll insbesondere die Bedingung *Erreichbarkeit des Optimums* erfüllen.

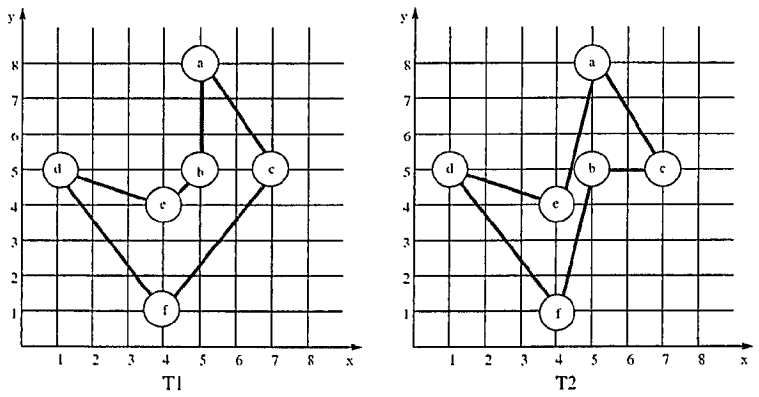
1. Geben Sie eine geeignete Nachbarschaftsfunktion in Pseudocode an. Beachten Sie dabei, dass der Graph G nicht unbedingt zusammenhängend ist.

2. Zeigen Sie, dass Ihre Funktion bei Eingabe eines Pfades stets wieder einen Pfad produziert, der sich vom Eingabe-Pfad unterscheidet und jeden Knoten von G maximal ein Mal besucht.

3. Sei p ein beliebiger Pfad in G. Zeigen Sie, dass man durch mehrmaliges Anwendern Ihrer Nachbarschaftsfunktion aus p einen Pfad maximaler Länge in G konstruieren kann.

Aufgabe 23 *Edge-Recombination Crossover*

Gegeben seien zwei Elterntouren in einem evolutionären Algorithmus für das symmetrische TSP:



Für einen Knoten i bezeichnen $x(i)$ und $y(i)$ dessen Koordinaten auf dem Gitter. Das Gewicht einer Kante $e = (i, j)$ in G sei definiert als die Manhattan-Distanz

$$M(i,j) = |x(i) - x(j)| + |y(i) - y(j)|$$

der Knoten i und j.

Führen Sie Edge-Recombination-Crossover (ERX) an den beiden gegebenen Eltern-TSP-Touren durch. Beginnen Sie bei Knoten d. Schreiben Sie in jeder Iteration auf, welche Möglichkeiten (Elternkanten) zur Wahl stehen und welche gewählt wird. Wenn mehrere Möglichkeiten zu Wahl stehen, soll jeweils die Kante mit dem geringsten Gewicht gewählt werden.

Aufgabe 24 *Cycle Crossover*

Gegeben seien zwei Elternlösungen p^1 und p^2 in einem evolutionären Algorithmus für das Quadratic Assignment Problem (QAP), wobei jede Lösung einer eindeutigen Zuordnung von Abteilungen $(1, \ldots, 9)$ zu Standorten $(1, \ldots, 9)$ entspricht:

	1	2	3	4	5	6	7	8	9
p^1	1	7	3	5	4	6	2	8	9
p^2	3	4	1	2	5	8	9	6	7

Führen Sie Cycle-Crossover (CX) an den beiden gegebenen Elternlösungen für das QAP durch. Erläutern Sie jeden Schritt.

Geben Sie insbesondere an, welcher Standort einer Abteilung zugeordnet wird, von welchem Elternteil die Standortzuordnung übernommen wird und wie der nächste Kandidat für die Zuordnung ermittelt wird.
