

Entwurfs- und Analysemethoden für Algorithmen

Skript zur Vorlesung SS 2016

Sascha Klüppelholz

Inhaltsverzeichnis

1 Algorithmisches Problemlösen	6
1.1 Zum Begriff „Algorithmus“	6
1.2 Die Güte von Algorithmen	11
1.3 Beispiele	18
1.3.1 Das Maxsummenproblem	19
1.3.2 Suchen in sortierten Arrays	20
1.3.3 Der Euklid-Algorithmus	24
2 Sortieren und Selektieren	28
2.1 Internes Sortieren durch Vergleiche	28
2.1.1 Einfache Sortierverfahren mit quadratischer Laufzeit	28
2.1.2 Mergesort	30
2.1.3 Quicksort	34
2.1.4 Untere Schranke für interne Sortierverfahren mit Vergleichen	41
2.2 Sortieren in linearer Zeit	46
2.2.1 Countingsort	47
2.2.2 Bucketsort	49
2.2.3 Radixsort	52
2.3 Externes Sortieren	55
2.4 Selektieren	57
3 Grundlegende Datenstrukturen	63
3.1 Bitvektoren, Arrays und Listen	63
3.1.1 Dünnbesetzte Matrizen (sparse matrices)	65
3.1.2 Stacks und Queues	68
3.1.3 Skiplisten	71

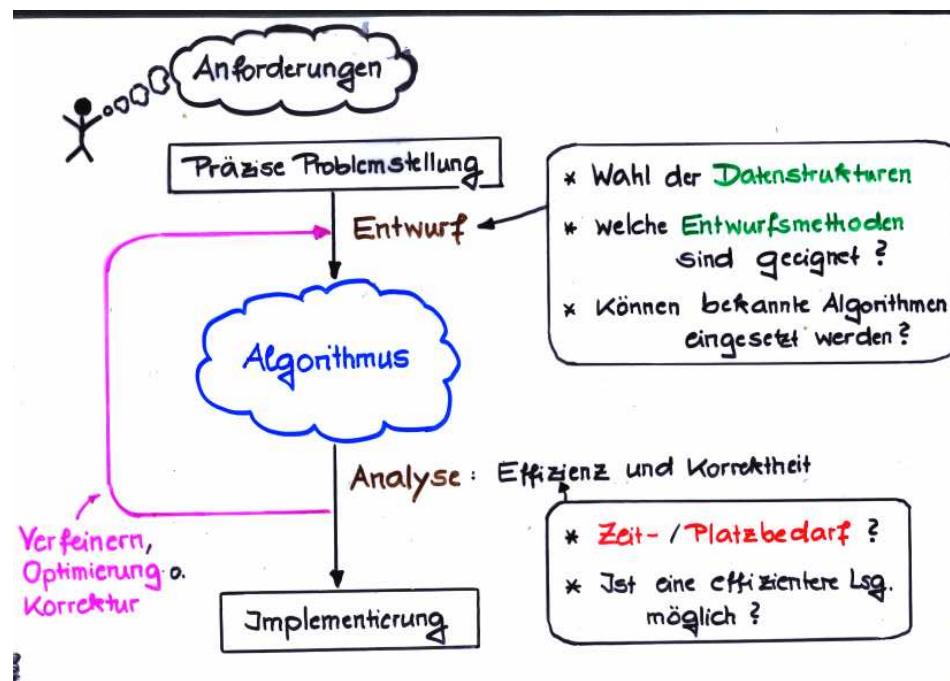
3.2	Graphen und Bäume	78
3.2.1	Grundbegriffe	78
3.2.2	Implementierung von Graphen	83
3.2.3	Traversierungsmethoden: Tiefen- und Breitensuche	84
3.2.4	Kürzeste Wege (single source, Einheitskosten)	91
3.2.5	Gerichtete Bäume	95
3.2.6	Binärbäume	105
3.2.7	Ungerichtete Bäume	111
3.3	Priority Queues (Heaps)	121
3.3.1	Minimumsheaps	121
3.3.2	Heapsort	125
4	Verwaltung dynamischer Mengen	130
4.1	Hashing	131
4.1.1	Hashing mit Kollisionslisten	134
4.1.2	Hashing mit offener Adressierung	136
4.1.3	Universelles Hashing	142
4.1.4	Kollisionslisten beschränkter Länge	146
4.2	Suchbäume	148
4.2.1	Binäre Suchbäume und AVL-Bäume	148
4.2.2	B-Bäume	178
5	Entwurfsmethoden für Algorithmen	191
5.1	Divide & Conquer	191
5.1.1	Das Master-Theorem	192
5.1.2	Matrizenmultiplikation nach Strassen	197
5.2	Backtracking und Branch & Bound	200
5.2.1	Backtracking	202
	Das n -Damen-Problem	202
	Graphfärben	208
	Das Rucksackproblem	215
5.2.2	Branch & Bound	223
	Die LIFO-Methode	224
	Die Least Cost Methode	229
5.3	Spielbäume	236
5.3.1	Das Minimax-Verfahren	241
5.3.2	$\alpha\beta$ -Pruning	243
5.4	Die Greedy-Methode	249
5.4.1	Huffman-Codes	250
5.4.2	Matroide	259
5.4.3	Auftragsplanung mit Schlussterminen	264
5.5	Dynamisches Programmieren	268
5.5.1	Erstellung regulärer Ausdrücke für endliche Automaten	269
5.5.2	Das Wortproblem für kontextfreie Sprachen	276

5.5.3	Optimalitätsprinzip	282
5.5.4	Optimale Suchbäume	283
5.5.5	Das Problem des Handlungsreisenden.	291
6	Algorithmen auf Graphen	310
6.1	Wegeprobleme mit gewichteten Kanten	310
6.1.1	Algorithmus von Dijkstra	315
6.1.2	Algorithmus von Bellman und Ford	324
6.1.3	Algorithmus von Floyd	329
6.1.4	Algorithmus von Warshall	336
6.2	Algorithmen für ungerichtete Graphen	337
6.2.1	Ungerichtete Bäume	337
6.2.2	Zyklentest in ungerichteten Graphen	341
6.2.3	Minimale aufspannende Bäume	344
6.2.4	Der Algorithmus von Prim	345
6.2.5	Der Kruskal-Algorithmus und UNION-FIND Wälder	350
6.3	DFS-Kantenklassifizierung und Zyklentest in Digraphen	360
6.4	Topologisches Sortieren	368
6.5	Starke Zusammenhangskomponenten	376
6.5.1	Algorithmus von Aho, Hopcroft und Ullman	378
6.5.2	Der Digraph der starken Zusammenhangskomponenten	381
6.5.3	Elimination von Kettenregeln in kontextfreien Grammatiken	382
6.6	Das Netzwerkflussproblem	386
6.6.1	Grundbegriffe	386
6.6.2	Das MaxFlow-MinCut-Theorem	388
6.6.3	Der Algorithmus von Ford & Fulkerson	391
6.6.4	Die Edmond-Karp Strategie für die Suche nach zunehmenden Pfaden	403
6.7	Bipartites Matching	408
6.7.1	Reduktion des Matchingproblems auf das Netzwerkflussproblem	409
6.7.2	Perfektes Matching und der Satz von Hall	412
7	Mustererkennung	416
7.1	Das Verfahren von Knuth, Morris und Pratt	417
7.2	Das Verfahren von Boyer-Moore	422
7.3	Der Algorithmus von Karp & Rabin	433
8	Geometrische Algorithmen	441
8.1	Scanline-Algorithmen	441
8.1.1	Das Sichtbarkeitsproblem	442
8.1.2	Das Schnittproblem für achsenparallele Segmente	445
8.1.3	Das allgemeine Liniensegment-Schnittproblem	448
8.2	Geometrisches DIVIDE & CONQUER	450
8.3	Das Voronoi-Diagramm	455

9 Algebraische und arithmetische Probleme	460
9.0.1 Matrizenmultiplikation nach Strassen	460
10 Randomisierung	462
10.1 MONTE CARLO und LAS VEGAS Algorithmen	466
10.2 Mustererkennung: der Algorithmus von Karp & Rabin	467
10.3 Primzahltest: der Algorithmus von Miller & Rabin	472
10.4 Randomisierte Datenstrukturen	479
10.4.1 Skiplisten	479

Ein paar einleitende Worte

Die Vorlesung beschäftigt sich mit dem Entwurf und der Analyse von Algorithmen zum effizienten Problemlösen. Neben der Vorstellung weiterführender Entwurfskonzepte liegt der Schwerpunkt der Veranstaltung vor allem auf der formalen Analyse aus dem Grundstudium bekannter sowie in dieser Vorlesung vorgestellter algorithmischer Verfahren hinsichtlich ihrer Platz- und Zeitkomplexität.



1 Algorithmisches Problemlösen

1.1 Zum Begriff „Algorithmus“

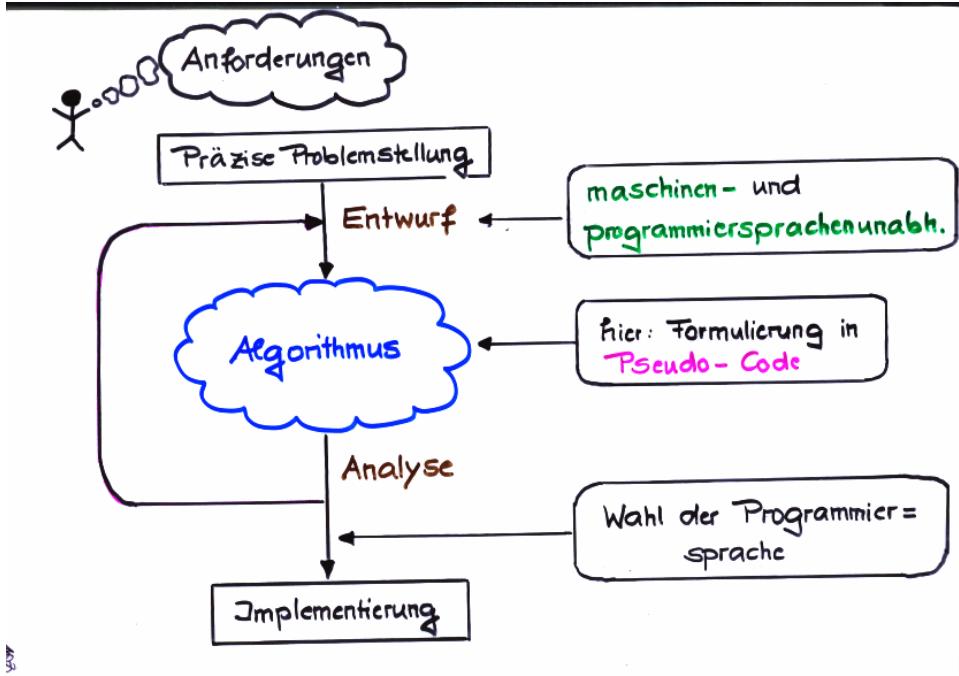
Informell kann ein *Algorithmus* als eine Rechenvorschrift bezeichnet werden, die sich aus elementaren, maschinell nachvollziehbaren Schritten zusammensetzt.¹ Formalisierungen des Begriffs sind u.a. durch abstrakte Rechnermodelle (z.B. Registermaschinen oder Turingmaschinen) möglich. Zwei wesentliche Bedingungen an Algorithmen, die sich aus solchen Formalisierungen ableiten lassen sind:

- *Finitheit der Beschreibung*: der Algorithmus muß sich als endlicher Text niederschreiben lassen, dessen Bestandteile sogenannte Anweisungen (Schritte) sind.
- *Effektivität*: die einzelnen Schritte müssen prinzipiell mit einem realen Rechner ausführbar sein.

Pseudo-Code. Die hier behandelten Algorithmen werden in einer simplen informellen Algorithmensprache angegeben. Diese verwendet:

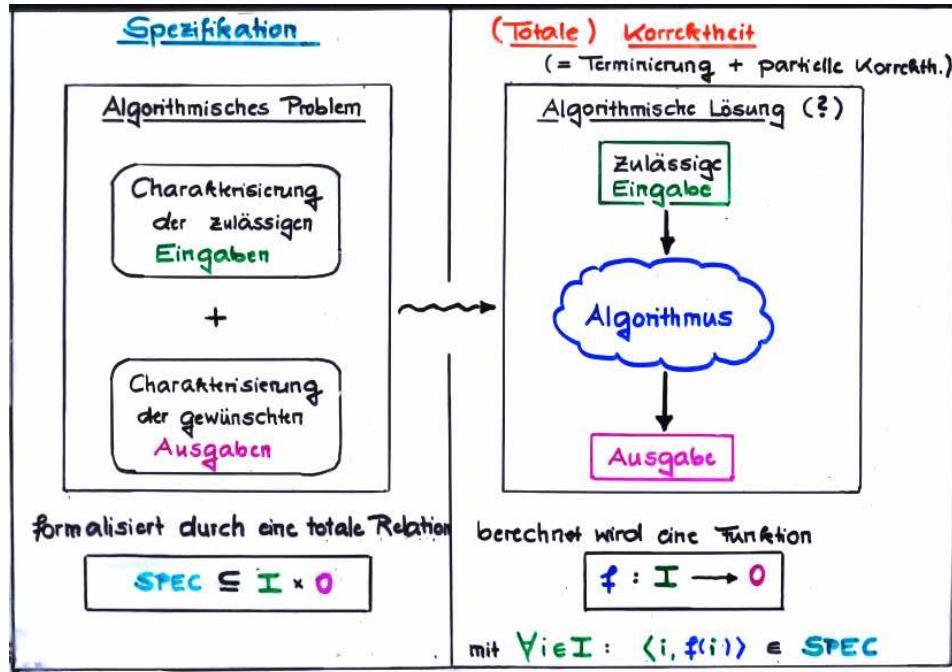
- Zuweisungen $x := \dots$
- Schleifen WHILE ... DO ... OD, FOR ... DO ... OD, etc.
- bedingte Anweisungen IF ... THEN ... FI oder IF ... THEN ... ELSE ... FI,
- die Anweisung RETURN „...“, die einen Wert zurückgibt und den Algorithmus beendet.

¹Das Wort „Algorithmus“ ist aus dem Namen Mohammed al-Khowarizmi eines persischen Mathematikers (ca. 800 n. Chr.) abgeleitet.



Oft benutzen wir umgangssprachliche Formulierungen von Anweisungen oder Anweisungsfolgen. Teilweise stehen diese für „Untermodule“, die separat erläutert werden. Weiter verwenden wir häufig saloppe Schreibweisen, z.B. für die Komponenten eines Arrays (wie x_i statt $x[i]$) oder funktionsähnliche Schreibweisen für die Komponenten eines Records (wie $key(x)$ statt $x.key$ oder $x.key()$). Oft gebrauchen wir auch die mathematischen Bezeichner für Vereinigung (\cup), Mengenschnitt (\cap), Mengendifferenz (\setminus) oder Formulierungen wie „füge x zu Q hinzu“ oder „entferne x aus Q “ anstelle der entsprechenden Operationen auf dem jeweiligen Datentyp. Die Angabe von Ein-/Ausgabeanweisungen (READ/WRITE) wird in vielen Fällen vernachlässigt. An welche Stellen diese einzufügen sind, sollte sich aus dem Kontext ergeben. Weiter verzichten wir auf die Angabe von Variablendeclarationen. Diese ergeben sich aus dem Kontext oder werden verbal angegeben.

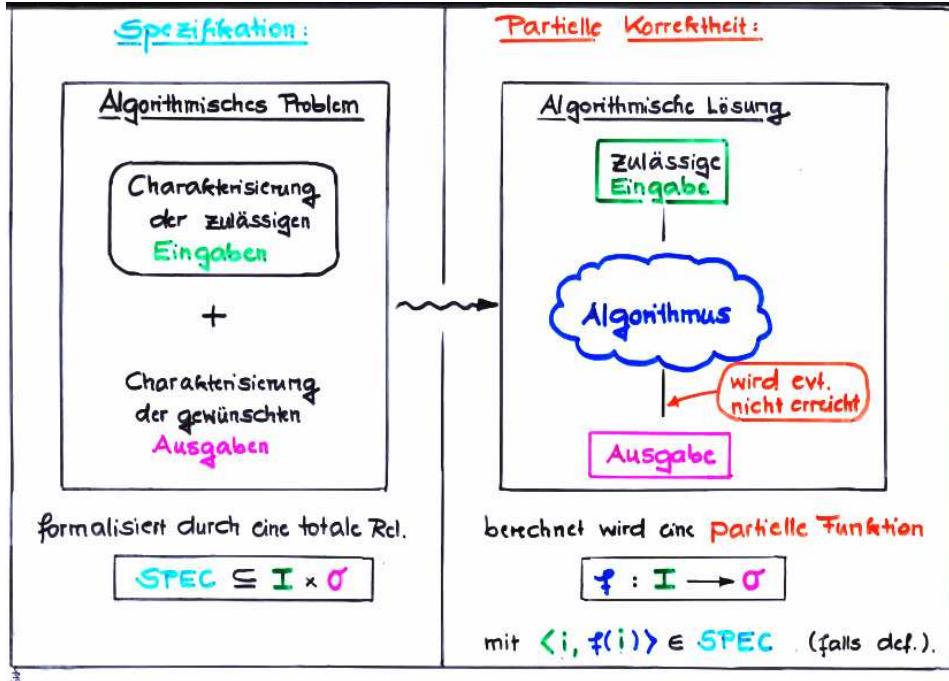
Formalisierung der Begriffe Spezifikation und Korrektheit. Eine mathematisch präzise Formulierung eines algorithmischen Problems ist eine Spezifikation. Unter einer algorithmischen Lösung eines Problems ist ein Algorithmus zu verstehen, der bzgl. der Spezifikation (total) korrekt ist.



Salopp formuliert bezeichnet *totale Korrektheit* (oft auch nur *Korrektheit* genannt) die Eigenschaft, daß der Algorithmus für alle zulässigen Eingaben anhält **und** eine im Sinne der Spezifikation korrekte Ausgabe liefert.

- Die *Spezifikation* eines Problems besteht aus
 - einer Menge I von zulässigen Eingabewerten
 - einer Menge O von Ausgabewerten
 - einer totalen Relation² $SPEC \subseteq I \times O$.
- (*Totale*) *Korrektheit*: Ein Algorithmus löst ein durch ein Tripel $\langle I, O, SPEC \rangle$ spezifiziertes Problem, wenn für alle zulässigen Eingabewerte $i \in I$ gilt: der Algorithmus terminiert bei Eingabe i und berechnet einen Wert $o \in O$, sodaß $(i, o) \in SPEC$.
- *Partielle Korrektheit*: Ein Algorithmus heißt partiell korrekt bzgl. eines durch $\langle I, O, SPEC \rangle$ spezifizierten Problems, wenn für alle Eingabewerte $i \in I$ gilt: falls der Algorithmus bei Eingabe i terminiert, dann wird ein Wert $o \in O$ berechnet mit $(i, o) \in SPEC$.

²Totale Relation $SPEC$ bedeutet hier, daß zu jedem $i \in I$ ein $o \in O$ existiert, sodaß $(i, o) \in SPEC$



Beispiel 1.1.1 (Spezifikationen). Das Primzahl-Problem (die Frage, ob eine gegebene ganze Zahl ≥ 3 eine Primzahl ist) kann durch die drei Komponenten $I = \mathbb{N}_{\geq 3}$, $O = \{\text{JA}, \text{NEIN}\}$ und $SPEC$ formalisiert werden, wobei $SPEC$ die Menge aller Paare (\bar{p}, NEIN) und (p, JA) ist, so daß p, \bar{p} ganze Zahlen ≥ 3 sind, p eine Primzahl sowie \bar{p} zusammengesetzt ist.

Als weiteres Beispiel betrachten wir ein Approximationsverfahren für die Logarithmusfunktion (zur Basis 2). Hier muß zusätzlich die gewünschte Genauigkeit, welche die Näherungslösung haben soll, festgelegt werden. So könnte die Spezifikation bspw. aus den Komponenten $I = \mathbb{R}_{>0}$, $O = \mathbb{R}$ und der Relation $SPEC$ aller Zahlenpaare $(x, y) \in I \times O$ mit $|y - \log x| < 10^{-8}$ bestehen.³ Man beachte, daß es für gegebenes $x \in I$ durchaus mehrere (im konkreten Fall unendlich viele) $y \in O$ geben kann, so daß $(x, y) \in SPEC$. An dieser Stelle entsteht ggf. ein Freiheitsgrad bei der Wahl der durch einen Algorithmus konkret berechneten Funktion $f : I \rightarrow O$. Für die totale Korrektheit ist lediglich sicherzustellen, daß $(x, f(x)) \in SPEC$ für alle $x \in I$.

□

Beispiel 1.1.2 (Ein partiell korrekter Algorithmus). Als Trivial-Beispiel für einen partiell korrekten Algorithmus betrachten wir die verbal formulierte Problemstellung

„berechne $2x$ für ein gegebenes $x \in \mathbb{Z}$ “.

Als Spezifikation können wir $I = O = \mathbb{Z}$ und $SPEC = \{(x, y) \in \mathbb{Z} \times \mathbb{Z} : y = 2 \cdot x\}$ verwenden. Algorithmus 1 ist dann partiell, aber nicht total korrekt.

³Hier sowie im Folgenden steht \log für die Logarithmusfunktion zur Basis 2, also $\log = \log_2$.

Algorithmus 1 Partiell korrekter Algorithmus zur Berechnung von $f(x) = 2x$

lese den Eingabewert x ;

$y := x$;

WHILE $y < 0$ **DO**

$y := y - 1$;

OD

$y := 2 * x$;

Gib y aus.

Der angegebene Algorithmus berechnet die partielle Funktion $f : \mathbb{Z} \rightarrow \mathbb{Z}$ mit $f(x) = 2x$, falls $x \geq 0$ und $f(x) = \perp$ („undefined“) für $x < 0$, da er für negatives x nicht anhält. Zur Erinnerung: Seien A, B beliebige Mengen. Eine *Funktion* (oder *Abbildung*, wobei manchmal auch der Zusatz „total“ gemacht wird) mit dem Definitionsbereich A und Wertebereich B ist eine Teilmenge f von $A \times B$, so daß es zu jedem $a \in A$ genau ein $b \in B$ mit $(a, b) \in f$ gibt. Schreibweise: $f(a) = b$ statt $(a, b) \in f$. Eine *partielle Funktion* $f : A \rightarrow B$ ist eine Funktion mit einem Definitionsbereich $A' \subseteq A$ und dem Wertebereich B . Dies ist gleichbedeutend damit, daß f eine Teilmenge von (A, B) ist, so daß es zu jedem $a \in A$ höchstens ein $b \in B$ gibt mit $(a, b) \in f$. Anstelle von $(a, b) \in f$ schreibt man $f(a) = b$. Ist f an einer Stelle $a \in A$ undefined (d.h. es gibt kein $b \in B$ mit $(a, b) \in f$), so schreibt man $f(a) = \perp$, wobei \perp ein neues Element ist, welches nicht in B vorkommt, und als Symbol für „undefined“ zu lesen ist. Partielle Funktionen können somit als (totale) Funktionen $A \rightarrow B \cup \{\perp\}$ aufgefaßt werden. \square

Abstraktion. Ein wichtiges Konzept für den Entwurf von Algorithmen sind Abstraktionstechniken, bei denen irrelevante Details der (in Umgangssprache formulierten) Aufgabenstellung verworfen werden. Derartige Abstraktionen können den Schritt von der informellen Problemstellung zur Spezifikation erleichtern und dienen u.a. dazu, sich über das eigentliche Kernproblem der Aufgabenstellung klar zu werden.

Reduktion. Zusätzlich lassen sich oftmals Reduktionsverfahren anwenden, die es erlauben, das gegebene (formal spezifizierte) Problem mit Hilfe eines Algorithmus' für ein bekanntes Problem zu lösen. Während der Abstraktionsschritt die Kreativität erfordert und nicht formal beschrieben werden kann, läßt sich das Konzept der Reduktion präzisieren. Zur Erinnerung: seien $\langle I, O, SPEC \rangle$, $\langle I', O', SPEC' \rangle$ die Spezifikationen zweier algorithmischer Probleme P und P' . P heißt auf P' *reduzierbar*, falls es einen Algorithmus A gibt, der zu jedem Paar $(i, o') \in I \times O'$ ein Paar $(i', o) \in I' \times O$ berechnet, so daß gilt:

$$(i', o') \in SPEC' \implies (i, o) \in SPEC.$$

Mit anderen Worten: aus einer algorithmischen Lösung für P' ergibt sich mit Hilfe von A eine algorithmische Lösung für P .

Beispiel 1.1.3 (Abstraktion und Reduktion). Das Turnierplanungsproblem fragt nach einem Spielplan, bei dem n Tennisspielerinnen im Gruppensystem („jede gegen jede“)

gegeneinander antreten sollen, mit der Nebenbedingung, daß keine Spielerin an einem Tag zwei oder mehr Spiele zu absolvieren hat. Dieses läßt sich auf das *Graph-Färbe-Problem* reduzieren und daher mit den dafür bekannten Algorithmen lösen.

Tennis-Turnier-Planungs-Problem:

Gesucht: Spielplan für n Tennisspieler (Spielmodus: Gruppensystem),

so daß

- * jeder Spieler höchstens einmal pro Tag spielt
- * die Anzahl der Spieldate **minimal** ist.

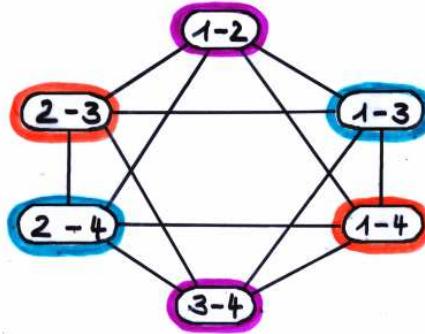
(Vorauss: $\lfloor \frac{n}{2} \rfloor$ Tennisplätze stehen zur Verfügung)

Meldeliste:

- 1 **Pete Sampras**
- 2 **Wayne Ferreira**
- 3 **Marat Safin**
- 4 **Jewgeni Kafelnikow**

Spielplan:

1. Tag: 1-2 , 3-4
2. Tag: 2-3 , 1-4
3. Tag: 2-4 , 1-3



Zunächst verwerfen wir in einem ersten Abstraktionsschritt alle irrelevanten Details (etwa daß es sich um ein Tennisturnier handelt) und organisieren die Spiele in einem ungerichteten Graphen. Wir errichten eine Kante zwischen je zwei Spielen, die nicht am selben Tag statt finden dürfen. Die Tage des Turnierplans stellen wir durch Farben dar und fragen nun nach einer Färbung der Knoten, so daß je zwei Knoten, die durch eine Kante miteinander verbunden sind, nicht dieselbe Farbe haben.

□

1.2 Die Güte von Algorithmen

Totale Korrektheit ist eine funktionale Eigenschaft, die völlig von quantitativen Aspekten (Laufzeit, Platzbedarf) abstrahiert.⁴ Zwei total korrekte Algorithmen für dasselbe Problem können von grundsätzlich verschiedener Qualität sein. Aus praktischer Sicht kann eine der Lösungen als „hoffnungslos schlecht“ und damit als unbrauchbar verworfen werden, während die andere für praktische Zwecke einsetzbar ist.

Die Laufzeit eines Algorithmus ist nicht nur ein Maß für die „Wartezeit“ bis ein Ergebnis vorliegt, sondern hat auch Einfluß auf die maximale Größe der Eingabemenge, die in

⁴Tatsächlich spielen neben Zeit und Platz in der Praxis noch andere Kriterien eine Rolle, die situationsbedingt zu gewichten sind. Dazu zählen u.a. die Einfachheit und Verständlichkeit, die Entwicklungskosten oder die Rechengenauigkeit bei numerischen Verfahren.

einem vorgegebenen Zeitintervall verarbeitet werden kann. (Vgl. Tabelle 1). Wir gehen hier davon aus, daß ein Rechner vorliegt, welcher 1.000 „elementare“ Rechenschritte pro Sekunde durchführt. Umgekehrt benötigt ein Algorithmus mit linearer Kostenfunktion

Kostenfunktion $T(n)$ (für die Laufzeit)	Maximale Eingabegröße bei Rechenzeit		
	1 Sekunde	1 Minute	1 Stunde
n	1 000	60 000	3 600 000
$n \cdot \log n$	140	4 895	204 094
n^2	31	244	1 897
n^3	10	39	153
2^n	9	15	21

Tabelle 1: Maximale Eingabegröße für vorgegebene Laufzeit

$T(n) = n$ für Eingaben der Größe $n = 80$ lediglich 0.08 Sekunden, während man für exponentielle Laufzeit $T(n) = 2^n$ nicht ernsthaft mit einem Ergebnis rechnen sollte, selbst wenn man 1 Mio. Rechenschritte pro Sekunde annimmt, da die benötigte Rechenzeit dann $2^{80}/1.000.000$ Sekunden $> 2^{34}$ Jahre beträgt. Das ist in etwa das geschätzte Alter des Universums. Die Tatsache, daß die Geschwindigkeit moderner Rechner kontinuierlich zunimmt, hat gerade auf Algorithmen mit „schlechter“ Zeitkomplexität kaum Einfluß. Tabelle 2 dient der Illustration dieser Aussage.

Kostenfunktion $T(n)$ (für die Laufzeit)	Maximale Eingabelänge	
	heute	bald (10-mal schneller)
n	x	$10x$
n^2	x	$3.16 x$
n^3	x	$2.15 x$
2^n	x	$x + 3.3$

Tabelle 2: Maximale Eingabegröße in Abhängigkeit der Technologie

Kostenmaße für Algorithmen lassen sich mit abstrakten Rechnermodellen formalisieren, etwa Turing- oder Registermaschinen. Die exakte Angabe der Kostenfunktion, die man durch Übersetzen des Pseudo-Codes in Programme für derartige „Low-level-Maschinenmodelle“ erhalten würde, ist meist extrem aufwendig. Stattdessen wird die Laufzeit und der Platzbedarf an den *dominannten* Operationen und Daten gemessen. Üblich ist die Angabe der *Größenordnungen* anstelle der präzisen Kostenfunktionen. Die Angabe der Größenordnungen ist oft sehr viel einfacher und sinnvoll, da die Größenordnungen weitgehend unabhängig von der konkreten Implementierung sind.

Zur Erinnerung: Asymptotische Notationen. Die (exakte) Größenordnung wird mit dem Operator Θ angegeben. Häufig ist es jedoch schwierig, die exakte Größenordnung anzugeben. Man weicht deshalb auf die Angabe einer *oberen Schranke* für T aus, die mit dem Operator \mathcal{O} angegeben wird. D.h. die Kostenfunktion T wird durch eine Funktion

f mit $T \in \mathcal{O}(f)$ abgeschätzt. Der Operator Ω wird zur Angabe von unteren Schranken verwendet. Üblich ist die Angabe einer „einfach verständlichen“ Funktion f als Schranke, z.B. $f(n) = n^k$ (Potenzen der Eingabegröße n) oder $f(n) = a^n$ (Exponentialfunktionen).

Definition 1.2.1. [O, Ω und Θ] Sei $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ eine Funktion.

- $\mathcal{O}(f) =$ Menge aller Funktionen $g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, für die es eine Konstante $C > 0$ und ein $n_0 \in \mathbb{N}$ gibt, so daß

$$g(n) \leq C \cdot f(n) \quad \text{für alle } n \geq n_0.$$

- $\Omega(f) =$ Menge aller Funktionen $g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, für die es eine Konstante $C > 0$ und ein $n_0 \in \mathbb{N}$ gibt, so daß

$$f(n) \leq C \cdot g(n) \quad \text{für alle } n \geq n_0.$$

- $\Theta(f) =$ Menge aller Funktionen $g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, für die es Konstanten $C_1, C_2 > 0$ und ein $n_0 \in \mathbb{N}$ gibt, so daß

$$C_1 \cdot f(n) \leq g(n) \leq C_2 \cdot f(n) \quad \text{für alle } n \geq n_0.$$

Üblich sind saloppe Schreibweisen: z.B. $g(n) = \mathcal{O}(f(n))$ statt $g \in \mathcal{O}(f)$. Gebräuchlich ist die Verwendung des Gleichheitszeichens anstelle der Inklusion, wobei „Gleichungen“ mit den asymptotischen Symbolen \mathcal{O} , Ω oder Θ von links nach rechts zu lesen sind. So steht $\mathcal{O}(f_1(n)) = \mathcal{O}(f_2(n))$ bspw. für $\mathcal{O}(f_1) \subseteq \mathcal{O}(f_2)$. \square

Der Zusammenhang zwischen \mathcal{O} , Ω und Θ ist wie folgt:

$$\begin{aligned} g(n) = \mathcal{O}(f(n)) \text{ gdw. } f(n) = \Omega(g(n)) \\ g(n) = \Theta(f(n)) \text{ gdw. } g(n) = \Omega(f(n)) \text{ und } g(n) = \mathcal{O}(f(n)) \end{aligned}$$

Definition 1.2.1 lässt sich in der offensichtlichen Weise für *partielle* Funktionen $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ mit unendlichem gemeinsamen Wertebereich erweitern. Ebenso verwenden wir häufig die offensichtliche Erweiterung von Definition 1.2.1 für mehrstellige (partielle) Funktionen. Etwa für $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ ist $\mathcal{O}(f(n, m))$ die Menge aller Funktionen $g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$, für die es eine Konstante $C > 0$ und natürliche Zahlen n_0, m_0 gibt, so daß $g(n, m) \leq C \cdot f(n, m)$ für alle $n \geq n_0$ und $m \geq m_0$. Z.B.

$$3n^2 \cdot \log m + 7n^3m^4 = \mathcal{O}(n^3m^4) \text{ und}$$

$$3n^2 \cdot \log m + 7n^3m^4 = \mathcal{O}(\text{poly}(n, m))$$

Hierbei schreiben wir $\mathcal{O}(\text{poly}(n))$ für die Menge aller polynomiell beschränkten Funktionen $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$. Also,

$$\mathcal{O}(\text{poly}(n)) = \bigcup_{k \geq 0} \mathcal{O}(n^k).$$

$\mathcal{O}(\text{poly}(n, m))$ bezeichnet die natürliche Erweiterung auf zweistellige Funktionen.

Bemerkung 1.2.2. Da bei der Angabe der Größenordnung der Kostenfunktion eines Algorithmus konstante Faktoren unterdrückt werden, kann es für relativ „kleine“ Eingaben dazukommen, daß ein Algorithmus mit einer schlechteren asymptotischen Laufzeit dennoch schneller als ein Algorithmus mit einer besseren asymptotischen Laufzeit ist. Beispielsweise trifft dies auf Algorithmen zu, deren Rechenzeiten durch die Kostenfunktionen $T_1(n) = 1000 \cdot n$ und $T_2(n) = 2 \cdot n^2$ gegeben sind. Der vermeintlich bessere Algorithmus mit der linearen Laufzeit ist dem Algorithmus mit quadratischem Aufwand jedoch erst für Eingaben der Größe > 500 überlegen, da $T_1(n) > T_2(n)$ für $n \leq 500$.

□

Uniformes Kostenmaß für Algorithmen in Pseudo-Code. Die Begriffe uniformes und logarithmisches Kostenmaß sollten aus den Grundstudiumsvorlesung bekannt sein. Das logarithmische Kostenmaß ist zwar präziser und liegt den Definitionen der Komplexitätsklassen (etwa P, NP, PSPACE, etc.) zugrunde, der Umgang mit dem uniformen Kostenmaß ist jedoch sehr viel einfacher. Tatsächlich ist das uniforme Kostenmaß oft ausreichend und wir werden nur in Sonderfällen auf das logarithmische Kostenmaß ausweichen.

Salopp formuliert werden unter dem uniformen Kostenmaß die Kosten wie folgt errechnet:

- Jede Anweisung (elementarer Befehl bzw. Schritt) benötigt eine Zeiteinheit, z.B. Vergleich von Zahlen, Addition, Multiplikation, etc.
- Der Speicherplatzbedarf wird an der Anzahl der benutzten elementaren Speicherreinheiten gemessen.

Dabei wird von Zahlen oder anderen Dateneinheiten (z.B. Strings) ausgegangen, die in einigen Speicherwörtern realer Rechner Platz haben und deren Darstellungsgröße während der Ausführung des Algorithmus nicht gravierend anwächst. Zudem wird angenommen, daß alle im Algorithmus verwendeten Daten im Hauptspeicher abgelegt sind.

Abweichungen vom uniformen Kostenmaß. Für Algorithmen, die mit Dateneinheiten arbeiten, deren Darstellungsgröße stark variieren kann oder die „gängigen Maße“ überschreitet, ist das uniforme Kostenmaß unrealistisch. So ist etwa für den nachfolgenden Algorithmus 2, der als Eingabe eine natürliche Zahl n hat und der den Wert 2^{2^n} berechnet, das *logarithmische Kostenmaß* vorzuziehen.

Algorithmus 2 Algorithmus zur Berechnung von $f(x) = 2^{2^x}$

```

 $x := 2;$ 
FOR  $i = 1, \dots, n$  DO
     $x := x * x;$ 
OD
Return  $x$ 

```

Dem logarithmischen Kostenmaß liegt die Sichtweise zugrunde, daß der betreffende Algorithmus mit einer deterministischen Turingmaschine realisiert wird. Es berücksichtigt daher die Darstellungsgröße der Operanden (z.B. als Zahlen im Binärsystem) für die Kosten der elementaren Anweisungen. Unter dem logarithmischen Kostenmaß kann die Zeit- und Platzkomplexität des obigen Algorithmus zur Berechnung von 2^{2^n} durch $\Omega(2^n)$ angegeben werden, während das uniforme Kostenmaß konstanten Platzbedarf und die Zeitkomplexität $\mathcal{O}(n)$ induziert. Man beachte, daß die lineare Laufzeit, die sich für die Berechnung von 2^{2^n} unter dem uniformem Kostenmaß ergibt, als unrealistisch zu verwerfen ist, da bereits in der Ausgabe exponentiell viele Ziffern geschrieben werden müssen. Eine weitere Situation, in der Abweichungen vom uniformen Kostenmaß sinnvoll sind, betrifft Verfahren, die auf einem Hintergrundspeicher (z.B. Cloudspeicher, Magnetbänder, etc.) abgelegte Daten verwenden. Für solche Verfahren sind meist die Anzahl (bzw. Häufigkeit) der Datentransporte sowie die Größe der zwischen dem Haupt- und Hintergrundspeicher zu transportierenden Daten relevante Faktoren für die Laufzeit.

Analyse von Algorithmen. Die Kostenfunktionen für die Laufzeit oder den Platzbedarf werden üblicherweise entweder für den schlimmsten Fall oder im Mittel bestimmt. Die Kostenfunktionen werden stets in Abhängigkeit der relevanten *Eingabegröße* (n) erstellt. Typischerweise hat eine Kostenfunktion die Gestalt $T : \mathbb{N} \rightarrow \mathbb{N}$ (oder $T : \mathbb{N}^k \rightarrow \mathbb{N}$, falls die Gesamtgröße der Eingabe an k Werten gemessen wird).⁵

- *Worst-Case-Analyse:* Die Kostenfunktion gibt die Kosten im schlimmsten Fall an. Liegt z.B. eine einstellige Kostenfunktion $T : \mathbb{N} \rightarrow \mathbb{N}$ vor, dann steht $T(n)$ für die maximalen Kosten, die für eine Eingabe der Größe n entstehen können.
- *Average-Case-Analyse:* Die erwarteten Kosten werden durch eine asymptotische Schranke abgeschätzt, wobei stochastische Annahmen über die Verteilung der Eingabewerte gemacht werden. Für eine einstellige Kostenfunktion $T : \mathbb{N} \rightarrow \mathbb{N}$ steht $T(n)$ für den Erwartungswert der Kosten, die durch Eingaben der Größe n entstehen können.

Was unter der Größe der Eingabedaten eines Algorithmus zu verstehen ist, läßt sich für den informellen Algorithmusbegriff, den wir hier verwenden, nicht allgemein festlegen. Beispielsweise kann für (auf Vergleiche beruhende) Sortierverfahren die Anzahl der zu sortierenden Datensätze eine sinnvolle Einheit sein (vgl. Abschnitt 2), während für viele Graphalgorithmen die Kostenfunktion bzgl. zwei Parametern n und m erstellt wird, welche für die Anzahl an Knoten bzw. Kanten stehen. In diesem Fall liegt also eine Kostenfunktion des Typs $T : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ vor.

Für Algorithmen, deren Eingabe natürliche Zahlen sind, wird die Kostenfunktion manchmal in Abhängigkeit der Zahlenwerte anstelle deren Darstellungsgröße angegeben, etwa i

⁵Streng genommen sollten wir den Wertebereich \mathbb{N} durch $\mathbb{N} \cup \{\infty\}$ ersetzen, um den Fall abzufangen, daß der vorliegende Algorithmus nicht für alle Eingaben terminiert. Wir setzen jedoch voraus, daß vor der Kostenanalyse die Terminierung nachgewiesen wurde, und können uns daher auf Kostenfunktionen des angegebenen Typs beschränken.

als Eingabewert anstelle der Eingabegröße $n = \lceil \log(i + 1) \rceil$. Dies ist häufig sinnvoll, da es das Rechnen und Abschätzen der Kostenfunktion erleichtert, jedoch sollte man im Auge behalten, daß man damit mit den Kosten $t(i)$ für das konkrete Eingabedatum i anstelle der Kostenfunktion $T(n)$ für Eingaben der Größe n argumentiert. Man beachte, daß sich die Komplexitätsklassen und Begriffe wie polynomielle Zeitbeschränkung (s.u.) stets auf die Eingabegröße (also die Kostenfunktion $T(n)$) beziehen.

Für eine Analyse der Rechenzeit zählt man meist nicht die exakte Anzahl an Rechenschritten, sondern weicht auf gewisse *dominante Rechenschritte* aus. Beispielsweise kann die Rechenzeit eines Sortieralgorithmus durch eine Kostenfunktion des Typs $T : \mathbb{N} \rightarrow \mathbb{N}$ dargestellt werden, wobei $T(n)$ für die maximale oder mittlere Anzahl von Vergleichen steht. Man unterdrückt damit das eigentliche Operieren mit den Datensätzen (nämlich das Vergleichen von Datensätzen) und konzentriert sich stattdessen auf die für das Sortierproblem typische Operationen.

Der benötigte Speicherplatz wird oftmals nur an den „zusätzlich“ gespeicherten Daten gemessen, die nicht zur Eingabe gehören und nicht in den betreffenden Variablen gespeichert werden können, ignoriert jedoch den Platzbedarf für die Eingabe selbst. Man spricht auch von *zusätzlichem Platzbedarf*. Man beachte, daß dies konsistent mit der Definition von Platzkomplexitätsklassen ist. Andererseits kann jedoch das Eingabeformat entscheidenden Einfluss auf den tatsächlich benötigten Speicherplatz (inklusive Eingabe) haben.

Polynomiell-zeitbeschränkte Algorithmen. Unter einem *polynomiell-zeitbeschränkten* Algorithmus versteht man einen Algorithmus, der sich mit Hilfe einer polynomiell-zeitbeschränkten deterministischen Turingmaschine realisieren läßt. In der hier verwendeten Terminologie bedeutet dies, daß der betreffende Algorithmus stets terminiert und unter dem logarithmischen Kostenmaß eine polynomiell beschränkte Kostenfunktion für die worst-case Rechenzeit hat. Dabei ist die worst-case Rechenzeit durch eine Kostenfunktion $T : \mathbb{N} \rightarrow \mathbb{N}$ zu bemessen, wobei $T(n)$ für die maximale Anzahl an Rechenschritten bei einer Eingaben der Größe n steht. So ist z.B. der folgende Algorithmus 3, der als Eingabe eine natürliche Zahl $n \geq 2$ hat, *nicht* polynomiell zeitbeschränkt.

Algorithmus 3 (hinsichtlich der Eingabegröße nicht-polynomiell-zeitbeschränkt)

```

sum := 0;
i := 1;
WHILE  $i \leq n$  DO
    sum := sum + i;
    i := i + 1;
OD
return sum

```

Dies mag zunächst verwundern, da der Algorithmus unter dem uniformen Kostenmaß die Laufzeit $T(n) = \Theta(n)$ und unter dem logarithmischen Kostenmaß die Laufzeit

$T_{\log}(n) = \Theta(n \cdot \log n)$ hat. Beide Kostenfunktionen sind zwar polynomiell beschränkt, allerdings im Eingabewert n , und nicht in der Eingabegröße. Diese ist nämlich $\log n$ (und es gilt z.B. $T_{unif}(n) = \Theta(2^{\log n})$). Mit dem selben Argument ist auch der naive Zusammengesetztheitstest, welcher als Eingabe eine natürliche Zahl $n \geq 3$ hat und durch sukzessives „Ausprobieren“ aller ganzer Zahlen $x \in \{2, 3, \dots, n - 1\}$ alle potentiellen positiven Teiler betrachtet, *nicht* polynomiell zeitbeschränkt.

Zur Erinnerung: Wahrscheinlichkeitsverteilungen und Erwartungswerte. Um die *durchschnittliche Laufzeit* oder den *durchschnittlichen Platzbedarf* eines Algorithmus anzugeben, benötigt man eine *Wahrscheinlichkeitsverteilung* für die möglichen Eingaben der Größe n . Wir fassen hier kurz die im Verlauf der Vorlesung benötigten Begriffe zusammen.⁶ Sei S eine endliche nicht-leere Menge von Ereignissen. Eine Wahrscheinlichkeitsverteilung für S ist eine Abbildung $Prob : S \rightarrow [0, 1]$, so daß

$$\sum_{s \in S} Prob(s) = 1.$$

$Prob(s)$ steht für die Wahrscheinlichkeit (engl. probability) des Ereignisses s . In vielen Fällen arbeitet man mit einer Gleichverteilung, d.h. der Wahrscheinlichkeitsverteilung $Prob(s) = \frac{1}{|S|}$ für alle $s \in S$. Dabei bezeichnet $|S|$ die Anzahl der Ereignisse, also die Kardinalität von S . Z.B. $S = \{Kopf, Zahl\}$ beschreibt die möglichen Ereignisse des „Münzwurf-Experiments“. Die Wahrscheinlichkeitsverteilung

$$Prob(Zahl) = Prob(Kopf) = \frac{1}{2}$$

scheint für eine faire Münze geeignet zu sein. Ein weiteres simples Beispiel: $S = \{1, 2, 3, 4, 5, 6\}$ steht für die möglichen Ereignisse eines „Einmal-Würfeln-Experiments“. Die Gleichverteilung $Prob(s) = 1/6$ spiegelt unsere Intuition von der Wahrscheinlichkeit des Würfelergebnisses s wider. Seien S und $Prob$ wie oben und $f : S \rightarrow \mathbb{R}$ eine Funktion. Der *Erwartungswert* von f bzgl. der Wahrscheinlichkeitsverteilung $Prob$ ist durch

$$\mathbb{E}[f] = \sum_{s \in S} Prob(s) \cdot f(s)$$

definiert. Z.B. ist für $S = \{1, 2, \dots, 6\}$ und $f(s) = s$ der Erwartungswert bzgl. der Gleichverteilung

$$\mathbb{E}[f] = \frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \dots + \frac{1}{6} \cdot 6 = \frac{7}{2}.$$

Wir nehmen an, daß ein Algorithmus gegeben ist, dessen Eingaben einer Menge I entstammen. Die Eingabegröße eines Datums $i \in I$ werde an k Parametern gemessen. Die Kostenfunktion für die Laufzeit und den Speicherplatzbedarf sind also Funktionen des

⁶Weitere Details können in jedem Schulbuch zum Thema „Wahrscheinlichkeitsrechnung und Statistik“ nachgelesen werden. Z.B. „Themenhefte Mathematik, Wahrscheinlichkeitsrechnung und Statistik“, Lambacher-Schweizer, Klett Verlag.“

Typs $T : \mathbb{N}^k \rightarrow \mathbb{N}$. Im Folgenden bezeichne $\text{Größe}(i)$ des Datums $i \in I$ und $t(i)$ die Kosten, die für i entstehen. Für die Worst-Case-Analyse arbeitet man mit der Kostenfunktion

$$T(n) = \sup \{ t(i) : i \text{ ist eine Eingabe der Größe } n \}.$$

Für die Average-Case-Analyse steht $T(n)$ Erwartungswert der Kosten $t(i)$ für Eingaben i der Größe n . Ist $\{i \in I : \text{Größe}(i) = n\}$ endlich (oder abzählbar), so sind die erwarteten Kosten durch

$$T_{avg}(n) = \sum_{\substack{i \in I \\ \text{Größe}(i) = n}} \text{Prob}(i) \cdot t(i)$$

gegeben, wobei Prob eine geeignet zu wählende Wahrscheinlichkeitsverteilung für Eingaben der Größe n ist. Die Annahme, daß $\{i \in I : \text{Größe}(i) = n\}$ endlich ist, trifft zwar dann zu, wenn man sich bei dem Begriff der „Eingabegröße“ streng an das Modell von Turingmaschinen hält.⁷ Da wir jedoch häufig davon abweichen und z.B. für Sortieralgorithmen die Anzahl an zu sortierenden Datensätzen als Eingabegröße betrachten, haben wir es oft mit unendlichen vielen Eingaben vorgegebener Größe zu tun. In diesem Fall behilft man sich, indem konkrete Eingaben, für die sich (in etwa) dieselben Kosten ergeben, in disjunkte Gruppen zusammengefasst werden und i in obiger Formel durch die Gruppe von i ersetzt wird. Die Gruppe von i steht dann für ein gewisses Elementarereignis. Wird etwa eine Zahl x in einer sortierten Zahlenfolge x_1, \dots, x_n gesucht, so kann es sinnvoll sein, $x < x_1$, $x > x_n$, $x = x_i$ und $x_{i-1} < x < x_i$ als Elementarereignisse aufzufassen.

Eine typische Vorgehensweise um die durchschnittliche Laufzeit (oder den durchschnittlichen Platzbedarfs) eines Algorithmus zu bestimmen, ist die folgende. Die Eingabemenge I wird in paarweise disjunkte Teilmengen I_1, \dots, I_k unterteilt, so daß für jede der Teilmengen I_l und je zwei Eingabedaten $i_1, i_2 \in I_l$ die Kosten $t(i_1)$ und $t(i_2)$ übereinstimmen. Auf der Menge $S = \{I_1, \dots, I_k\}$ wird eine „möglichst realistische“ Wahrscheinlichkeitsverteilung Prob festgelegt. (Oftmals wird mit der Gleichverteilung $\text{Prob}(I_l) = 1/|S| = 1/k$ gearbeitet. In einigen Anwendungen werden jedoch auch andere Wahrscheinlichkeitsverteilungen benötigt.) Die erwarteten Kosten für die Eingabegröße n sind

$$T(n) = \sum_{l=1}^k \text{Prob}(I_l) \cdot t(I_l),$$

wobei $t(I_l) = t(i)$ für ein (alle) $i \in I_l$.

1.3 Beispiele

Zum „Warmwerden“ betrachten wir drei einfache Beispiele, für die wir die Laufzeit und den Platzbedarf analysieren.

⁷Man beachte, daß es nur endliche viele Wörter der Länge n über dem Eingabealphabet gibt, da das Eingabealphabet von Turingmaschinen als endlich vorausgesetzt wird.

1.3.1 Das Maxsummenproblem

Gegeben ist die Prognose für die Kursschwankungen einer Aktie in den nächsten n Tagen. Für jeden Tag ist eine ganze Zahl gegeben, die für den Punktegewinn bzw. -verlust steht. Insgesamt liegt also eine Zahlenfolge x_1, x_2, \dots, x_n vor, wobei $x_i \in \mathbb{R}$ für die Punkte steht, welche die betreffende Aktie an Tag i voraussichtlich hinzu gewinnt bzw. verliert. Gesucht ist der maximale Gewinn, den man durch einen optimalen Einkaufs- bzw. Verkaufstag erzielen kann.

In einem ersten Abstraktionsschritt stellen wir fest, daß die Interpretation der Zahlen x_i als Gewinn-/Verlustpunkte einer Aktie für die Lösung des Problems völlig unerheblich ist und deshalb in der Spezifikation weggelassen werden kann. Die Fragestellung des Maxsummen-Problems ist wie folgt: Gegeben ist eine Folge x_1, \dots, x_n ganzer Zahlen. Gesucht ist der Wert

$$\max_{1 \leq i \leq j \leq n} \text{sum}(i, j), \quad \text{wobei } \text{sum}(i, j) = \sum_{l=i}^j x_l.$$

Im Folgenden werden wir uns mit Problemstellung in der obigen Form begnügen und verzichten auf die präzise Angabe der Spezifikation als Tripel $\langle I, O, SPEC \rangle$. Wir betrachten drei Lösungen (total korrekte Algorithmen) des Maxsummen-Problems. Unser erster Ansatz hat kubische Laufzeit und konstanten zusätzlichen Speicherbedarf. Diesen verbessern wir schrittweise zu einem Verfahren mit quadratischer und zuletzt linearer Laufzeit und halten den zusätzlichen Speicherbedarf weiterhin konstant.

Naive Lösung mit kubischer Laufzeit. Ein erster Lösungsversuch besteht darin, völlig naiv alle Werte $\text{sum}(i, j)$ zu berechnen (vgl. Algorithmus links in Abbildung 1). Offenbar

$\text{max} := -\infty;$ <u>FOR</u> $i = 1, \dots, n$ <u>DO</u> <u> FOR</u> $j = i, \dots, n$ <u>DO</u> <u> </u> $\text{sum}(i, j) := x_i;$ <u> FOR</u> $\ell = i + 1, \dots, j$ <u>DO</u> <u> </u> $\text{sum}(i, j) := \text{sum}(i, j) + x_\ell$ <u> OD</u> <u> IF</u> $\text{sum}(i, j) > \text{max}$ <u>THEN</u> <u> </u> $\text{max} := \text{sum}(i, j)$ <u> FI</u> <u> OD</u> <u> return</u> $\text{max}.$	$\text{max} := -\infty;$ <u>FOR</u> $i = 1, \dots, n$ <u>DO</u> <u> FOR</u> $j = i, \dots, n$ <u>DO</u> <u> </u> $\text{sum} := x_i;$ <u> FOR</u> $\ell = i + 1, \dots, j$ <u>DO</u> <u> </u> $\text{sum} := \text{sum} + x_\ell$ <u> OD</u> <u> IF</u> $\text{sum} > \text{max}$ <u>THEN</u> <u> </u> $\text{max} := \text{sum}$ <u> FI</u> <u> OD</u> <u> return</u> $\text{max}.$
---	---

Abbildung 1: Naive Lösungen des Maxsummenproblems mit kubischer Laufzeit

erhalten wir die Rechenzeit $\Theta(n^3)$ und quadratischen Platzbedarf für den schlimmsten und mittleren Fall. Zunächst stellt man fest, daß kein Bedarf zur Speicherung der Werte

$sum(i, j)$ besteht und daher eine Variable sum ausreichend ist (statt $\Theta(n^2)$ Variablen $sum(i, j)$). Wir erhalten den Algorithmus rechts in Abbildung 1, der zwar mit konstantem zusätzlichen Platzbedarf auskommt, jedoch ebenfalls kubische Laufzeit hat.

Lösung mit quadratischer Laufzeit. Eine Laufzeitverbesserung erhalten wir durch Ausnutzung der Tatsache, daß

$$sum(i, j + 1) = sum(i, j) + x_{j+1}.$$

Dies führt zu dem in Abbildung 2 angegebenen Algorithmus quadratischer Laufzeit.

```

max := -∞;
FOR i = 1, ..., n DO
    sum := 0;
    FOR j = i, ..., n DO
        sum := sum + xj
        IF sum > max THEN
            max := sum
        FI
    OD
return max.

```

Abbildung 2: Lösung des Maxsummenproblems mit quadratischer Laufzeit

Lösung mit linearer Laufzeit. Lineare Laufzeit erhalten durch die Beobachtung, daß

$$\max_{1 \leq i \leq j \leq n} sum(i, j) = \max_{1 \leq i \leq n} sum(i),$$

wobei

$$sum(i) = \max_{i \leq j \leq n} sum(i, j) = \max\{x_i, x_i + sum(i + 1)\}, \quad i = 1, 2, \dots, n - 1$$

und $sum(n) = x_n$. Siehe Algorithmus im linken Teil von Abbildung 3. Nun haben wir allerdings linearen zusätzlichen Speicherbedarf. Dieser kann aber umgangen werden, da zur Berechnung von $sum(i)$ nur der Wert $sum(i + 1)$ benötigt wird. Wir erhalten den im rechten Teil von Abbildung 3 angegebenen Algorithmus mit linearem Rechenaufwand und konstantem zusätzlichen Platzbedarf.

In diesem Beispiel stimmen die mittleren Kosten mit den Kosten im worst-case überein.

1.3.2 Suchen in sortierten Arrays

Gegeben ist eine aufsteigend sortierte Zahlenfolge x_1, \dots, x_n in Array-Darstellung und eine Zahl x . Gesucht ist ein Index $i \in \{1, \dots, n\}$ mit $x_i = x$, falls $x \in \{x_1, \dots, x_n\}$.

$max := x_n;$ $sum(n) := x_n;$ FOR $i = n - 1, \dots, 1$ DO IF $sum(i + 1) \leq 0$ THEN $sum(i) := x_i$ ELSE $sum(i) := sum(i + 1) + x_i$ FI IF $sum(i) > max$ THEN $max := sum(i)$ FI OD return $max.$	$max := x_n;$ $sum := x_n;$ FOR $i = n - 1, \dots, 1$ DO IF $sum \leq 0$ THEN $sum := x_i$ ELSE $sum := sum + x_i$ FI IF $sum > max$ THEN $max := sum$ FI OD return $max.$
---	--

Abbildung 3: Linearzeit-Lösungen des Maxsummenproblems

Sequentielle Suche. Der naive Ansatz besteht in der Verwendung einer *Sequentiellen Suche*, welche das Array „von links nach rechts“ durchläuft und alle x_i ’s mit dem gesuchten Wert x vergleicht bis x gefunden ist oder das Array-Ende erreicht ist. Siehe Algorithmus 4.

Algorithmus 4 Sequentielle Suche

```

i := 1;  

WHILE  $i \leq n$  und  $x_i \leq x$  DO  

    IF  $x_i = x$  THEN  

        return „ $x$  steht an Position  $i$ “  

    ELSE  

        i := i + 1  

    FI  

OD  

return „ $x$  wurde nicht gefunden“

```

Kostenanalyse der Sequentiellen Suche. Die worst-case Laufzeit ist offenbar $\Theta(n)$, da im schlimmsten Fall das gesamte Array einmal durchlaufen werden muss. Wir erläutern nun die Durchschnittsanalyse. Hierzu greifen wir uns den Vergleich „ $x_i = x$ “ in der IF-Abfrage als dominante Operation heraus und „zählend“ deren mittlere Anzahl an Ausführungen, zunächst im Falle einer erfolglosen Suche. Nimmt man Gleichverteilung für die erfolglose Suche an, dann arbeiten wir mit der Ereignismenge

$$S = \{[x_{j-1} < x < x_j] : j = 1, 2, \dots, n+1\}, \text{ wobei } x_0 = -\infty, x_{n+1} = +\infty$$

und $Prob([x_{j-1} < x < x_j]) = 1/(n+1)$. Die Kosten des Ereignisses $[x_{j-1} < x < x_j]$, gemessen an der Anzahl an Durchführungen des Vergleichs „ $x_i = x$ “, betragen

$$t([x_{j-1} < x < x_j]) = j - 1.$$

Es ergibt sich die erwartete Anzahl an Vergleichen

$$T(n) = \sum_{j=1}^{n+1} \text{Prob}([x_{j-1} < x < x_j]) \cdot t([x_{j-1} < x < x_j]) = \sum_{j=1}^{n+1} \frac{1}{n+1} \cdot (j-1) = \frac{n}{2}.$$

Ebenso kann man die erwartete Anzahl an Vergleichen für die erfolgreiche Suche bestimmen.

Binäre Suche. Effizienter als die Sequentielle Suche ist die *Binäre Suche*, deren Idee darin besteht, zunächst das mittlere Element x_m des sortierten Eingabe-Arrays mit dem gesuchten Wert x zu vergleichen. Im Nicht-Erfolgsfall wird die Suche im linken bzw. rechten Teil-Array fortgesetzt, je nachdem ob x_m kleiner oder größer als x ist.

In Algorithmus 5 ist eine rekursive Formulierung angegeben, die initial mit $\ell = 1$ und $r = n$ aufzurufen ist.

Algorithmus 5 Binäre Suche $\text{BinSearch}(x, \ell, r)$

```

IF  $\ell > r$  THEN
    return „x nicht gefunden“
ELSE
     $m := (\ell + r) \text{ div } 2;$ 
    IF  $x = x_m$  THEN
        return „x steht an Position  $m$ “
    ELSE
        IF  $x < x_m$  THEN
             $\text{BinSearch}(x, \ell, m - 1)$ 
        ELSE
             $\text{BinSearch}(x, m + 1, r)$ 
    FI
FI
FI

```

Kostenanalyse der Binären Suche. Als dominante Operation greifen wir den Vergleich „ $x = x_m$ “ heraus. Im Folgenden bezeichne $T(n)$ die maximale Anzahl an Vergleichen „ $x = x_m$ “, die in $\text{BinSearch}(x, \ell, r)$ bei Eingabelänge $n = r - \ell + 1$ ausgeführt werden. Offenbar ist $T(0) = 0$ (dies entspricht dem Fall $\ell > r$) und $T(1) = 1$ (dies entspricht dem Fall $\ell = r$). Für $n = r - \ell + 1$ erhalten wir: $T(n) \leq 1 + T(\lfloor \frac{n}{2} \rfloor)$. Dabei steht der Summand 1 für den initialen Vergleich „ $x = x_m$ “ in $\text{BinSearch}(x, \ell, r)$ und $T(\lfloor \frac{n}{2} \rfloor)$ für die maximale Anzahl an Vergleichen, die in den Rekursionsaufrufen stattfinden. Man beachte, daß das betrachtete Teil-Array höchstens die Länge $\max\{m - \ell, r - m\} \leq \lfloor \frac{n}{2} \rfloor$ hat. Die worst-case Laufzeit ergibt sich daher durch Lösen der *Rekurrenz*

$$T(n) \leq \begin{cases} 0 & : n < 1 \\ 1 & : n = 1 \\ 1 + T(\lfloor n/2 \rfloor) & : n \geq 2 \end{cases}$$

für die Anzahl $T(n)$ der Vergleiche. Hier sowie im Folgenden bezeichnet $\lfloor z \rfloor$ die größte ganze Zahl, die kleiner oder gleich z ist, und $\lceil z \rceil$ die kleinste ganze Zahl y , so daß $z \leq y$. Bei der Herleitung der Lösung $T(n) = \Theta(\log n)$ stellen zunächst fest, daß T monoton ist. Sei $k = \lceil \log n \rceil$. Dann ist $T(n) \leq T(2^k)$ (und $2^{k-1} < n \leq 2^k$). Für $k > 1$ gilt:

$$\begin{aligned} T(2^k) &\leq 1 + T(2^{k-1}) \\ &\leq 1 + 1 + T(2^{k-2}) = 2 + T(2^{k-2}) \\ &\quad \vdots \\ &\leq r + T(2^{k-r}) \\ &\quad \vdots \\ &\leq k + T(2^{k-k}) = k + T(1) = \lceil \log n \rceil + 1. \end{aligned}$$

Hieraus folgt $T(n) \leq \lceil \log n \rceil + 1 = \mathcal{O}(\log n)$.

Eine asymptotische untere Schranke resultiert aus der Beobachtung, daß wir für $n = 2^k - 1$ in obiger Rechnung Gleichheit erhalten, also

$$T(n) = k + 1 = \lceil \log n \rceil + 1, \text{ falls } n = 2^k - 1.$$

(Dieser schlimmste Fall tritt für eine erfolglose Suche ein.)

Bemerkung 1.3.1. Wir stellen also fest, daß in der erfolglosen Suche im schlimmsten Fall $T(2^k - 1) = k$ und $T(2^k) = k + 1$. D.h. der Sprung in der Anzahl der erforderlichen Rekursionen für den schlimmsten Fall findet genau an den Zweierpotenzen statt.

Aufgrund der Monotonie von T gilt für $2^{k-1} \leq n < 2^k - 1$ und

$$T(n) \geq T(2^{k-1} - 1) = k.$$

Wir erhalten $T(n) = \Omega(\log n)$ und somit $T(n) = \Theta(\log n)$.

Die *mittlere Anzahl* an Vergleichen für eine erfolglose Binäre Suche unter der Annahme der Gleichverteilung, also

$$\text{Prob}([x_{i-1} < x < x_i]) = \frac{1}{n+1}, \quad i = 1, 2, \dots, n+1,$$

wobei $x_0 \stackrel{\text{def}}{=} -\infty$ und $x_{n+1} \stackrel{\text{def}}{=} +\infty$, ergibt sich wie folgt. Für den Fall $n = 2^k - 1$ erhalten wir die Rekurrenz

$$T(n) = 1 + \underbrace{\text{Prob}([x < x_{\frac{n+1}{2}}]) \cdot T\left(\frac{n-1}{2}\right)}_{=\frac{1}{2}} + \underbrace{\text{Prob}([x > x_{\frac{n+1}{2}}]) \cdot T\left(\frac{n-1}{2}\right)}_{=\frac{1}{2}} = 1 + T\left(\frac{n+1}{2}\right)$$

für $n \geq 2$. Wie oben gezeigt folgt hieraus $T(n) = \Theta(\log n)$. Mit denselben Argumenten wie oben genügt die Betrachtung des Falls $n = 2^k - 1$, da man aufgrund der Monotonie

für $2^{k-1} \leq n < 2^k$ eine untere Schranke durch die Betrachtung von $T(2^{k-1} - 1)$ erhält und eine obere Schranke durch die Betrachtung von $T(2^k - 1)$.

Die Binäre Suche in einem sortierten Array der Länge n hat somit die Laufzeit $\mathcal{O}(\log n)$ im schlimmsten Fall. Zur Demonstration, daß die Binäre Suche der Sequentiellen Suche bei Weitem überlegen ist, kann man sich klarmachen, daß für $n = 1.000.000$ die Sequentielle Suche im Mittel $n/2 = 500.000$ Vergleiche benötigt, während die Binäre Suche mit ca. $\log_2(10^6) \approx 20$ Vergleichen selbst im schlimmsten Fall auskommt.

Exponentielle Suche. Für den Fall, daß nach einem Wert x mit $x \leq x_k$ für ein sehr kleines k gesucht wird, ist die Sequentielle Suche der Binären Suche überlegen. Dies fällt zwar nur für sehr große Eingabefolgen ins Gewicht, dennoch wollen wir einen Ausweg mit der sogenannten Exponentiellen Suche skizzieren. Die Idee besteht darin, den Suchvorgang in zwei Phasen zu zerlegen:

Phase 1: Der gesuchte Wert x wird der Reihe nach mit den Werten x_{2^i} verglichen, $i = 0, 1, 2, \dots$, bis entweder das Ende der Eingabefolge erreicht ist oder $x \leq x_{2^i}$.

Phase 2: Ist $x = x_{2^i}$, so ist die Suche erfolgreich beendet. Ebenso kann der Suchprozess eingestellt werden, wenn $x < x_1$ oder $x > x_n$. Andernfalls gilt $x_{2^{i-1}} < x < x_{2^i}$. Wir führen nun eine Binäre Suche im Array-Abschnitt $x_{2^{i-1}+1}, x_{2^{i-1}+2}, \dots, x_{2^i-1}$ durch.

Im schlimmsten Fall benötigt man für Phase 1 und 2 jeweils ca. $\log n$, insgesamt also rund $2 \cdot \log n$, Vergleiche. (Man beachte, daß in Phase 2 das betrachtete Teil-Array stets maximal $n/2$ Elemente hat.) Für den Fall, daß $x_{k-1} < x < x_k$, endet Phase 1 mit dem Wert $i = \lceil \log k \rceil$. Das in Phase 2 analysierte Array besteht aus

$$(2^i - 1) - (2^{i-1} + 1) + 1 = 2^{i-1} - 1 < k$$

Elementen. Die Binäre Suche benötigt daher höchstens $\log k$ Vergleiche. Insgesamt kommt die Exponentielle Suche also mit ca. $2 \cdot \log k$ Vergleichen aus, während mit der Sequentiellen Suche k Vergleiche auszuführen sind.

1.3.3 Der Euklid-Algorithmus

Wir betrachten nun die Fragestellung des *größten gemeinsamen Teilers* zweier ganzer, positiver Zahlen. Gegeben sind zwei natürliche Zahlen x, y mit $x \geq y > 0$. Gesucht ist

$$ggT(x, y) = \max\{z \in \mathbb{N} : z \text{ ist Teiler von } x \text{ und } y\}.$$

Lemma 1.3.2. Seien $x, y \in \mathbb{N}$, $x > y > 0$. Dann gilt:

- $ggT(x, 0) = ggT(x, x) = x$
- $ggT(x, y) = ggT(y, x \bmod y)$.

Zur Erinnerung: Sind x, y ganze Zahlen mit $y \neq 0$, so bezeichnen $q = x \bmod y$ und $r = x \bmod y$ die eindeutig bestimmten ganzen Zahlen, für welche $x = qy + r$ und $0 \leq r < y$ gilt.

Beweis. Die erste Aussage ist klar. Die zweite Aussage ist wie folgt einsichtig. Sei $r = x \bmod y$ und $x = qy + r$.

- Ist d ein Teiler von x und y , dann gibt es ganze Zahlen b und c mit $x = bd$ und $y = cd$. Dann ist

$$bd = x = qy + r = qcd + r,$$

also $r = bd - qcd = (b - qc)d$. Somit ist d ein Teiler von y und r .

- Ist umgekehrt d ein Teiler von y und r , dann gibt es ganze Zahlen c und e mit $y = cd$ und $r = ed$. Es folgt

$$x = qy + r = qcd + ed = (qc + e)d.$$

Also ist d ein Teiler von x und y .

Diese Überlegungen zeigen, daß x und y sowie y und $x \bmod y$ dieselben Teiler haben. Insbesondere stimmen die größten gemeinsamen Teiler überein. \square

Der Euklid-Algorithmus (siehe Algorithmus 6) zur Bestimmung des größten gemeinsamen Teilers zweier ganzer Zahlen x, y (von denen wir o.E. $x \geq y$ annehmen können) verwendet eine rekursive Funktion, die $ggT(x, y)$ gemäß Lemma 1.3.2 bestimmt.

Algorithmus 6 Euklid-Algorithmus $ggT(x, y)$ für $x \geq y \geq 1$ oder $x > y = 0$

IF $x = y$ oder $y = 0$ **THEN**

return x

ELSE

return $ggT(y, x \bmod y)$

FI

Z.B. wird der größte gemeinsame Teiler von 42 und 15 durch

$$ggT(42, 15) = ggT(15, 12) = ggT(12, 3) = ggT(3, 0) = 3$$

bestimmt.

Laufzeit des Euklid-Algorithmus. Zunächst stellen wir fest, daß die Rekursionsaufrufe als die dominanten Operationen angesehen werden können. Wir führen nun eine Worst-Case-Analyse durch, indem wir eine Abschätzung für die maximalen Anzahl an Rekursionsaufrufen angeben. Offenbar findet für $x = y \geq 1$ oder $y = 0$ kein Rekursionsaufruf statt. Wir betrachten nun den Fall $x > y \geq 1$ und setzen als Hilfsmittel die *Fibonacci-Zahlen* F_0, F_1, F_2, \dots ein, welche durch folgende Rekursionsformel definiert sind:

$$F_k = \begin{cases} 0 & : \text{falls } k = 0 \\ 1 & : \text{falls } k = 1 \\ F_{k-1} + F_{k-2} & : \text{falls } k \geq 2. \end{cases}$$

Lemma 1.3.3. Sind x, y natürliche Zahlen mit $x > y \geq 1$ und finden k Rekursionsaufrufe bei der Ausführung des Euklid-Algorithmus zur Berechnung von $\text{ggT}(x, y)$ statt, wobei $k \geq 1$, so gilt: $x \geq F_{k+1}$ und $y \geq F_k$.

Beweis. Die Aussage kann durch Induktion nach k bewiesen werden. Für $k = 1$ ist die Aussage klar, da $y \geq F_1 = 1$ und $x > y \geq F_2 = 1$. Für $k \geq 2$ gilt für die Anzahl $t(x, y)$ der Rekursionsaufrufe:

$$k \leq t(x, y) = 1 + t(y, x \bmod y),$$

also $1 \leq k - 1 \leq t(y, x \bmod y)$. Hieraus folgt $y > (x \bmod y) \geq 1$. (Beachte, daß $t(x, 0) = 0$. Daher ist $x \bmod y = 0$ nicht möglich.) Aus der Induktionsvoraussetzung folgt:

$$y \geq F_k, \quad (x \bmod y) \geq F_{k-1}.$$

Mit $q = x \bmod y$ folgt:

$$x = \underbrace{q}_{\geq 1} \cdot \underbrace{y}_{\geq F_k} + \underbrace{x \bmod y}_{\geq F_{k-1}} \geq F_k + F_{k-1} = F_{k+1}.$$

□

Lemma 1.3.4. Für $k \geq 1$ gilt: $F_{k+1} \geq (\frac{3}{2})^{k-1}$

Beweis. Die Aussage kann durch Induktion nach k bewiesen werden. Für $k \in \{1, 2\}$ gilt:

$$F_2 = 1 = (\frac{3}{2})^0 \quad \text{und} \quad F_3 = 2 > 1.5 = (\frac{3}{2})^1$$

Im Induktionsschritt $k - 1, k - 2 \implies k$ ($k \geq 3$) erhalten wir:

$$\begin{aligned} F_{k+1} &= F_k + F_{k-1} \\ &\geq (\frac{3}{2})^{k-2} + (\frac{3}{2})^{k-3} \\ &= (\frac{3}{2})^{k-3} \cdot (\frac{3}{2} + 1) \\ &= (\frac{3}{2})^{k-3} \cdot \frac{5}{2} \\ &> (\frac{3}{2})^{k-3} \cdot \frac{9}{4} \\ &= (\frac{3}{2})^{k-1} \end{aligned}$$

Beachte, daß $\frac{5}{2} = \frac{10}{4} > \frac{9}{4} = (\frac{3}{2})^2$. □

Eine bessere Abschätzung ergibt sich aus der Tatsache, daß

$$F_k = \frac{1}{\sqrt{5}} \cdot \left(\left(\frac{1+\sqrt{5}}{2} \right)^k - \left(\frac{1-\sqrt{5}}{2} \right)^k \right),$$

was ebenfalls durch Induktion nach k bewiesen werden kann. Von dieser Aussage werden wir jedoch keinen Gebrauch machen.

Kombiniert man die Aussagen von Lemma 1.3.3 und Lemma 1.3.4, so erhält man: Ist $x > y \geq 1$ und $t(x, y) = k \geq 2$, so ist $x \geq F_{k+1} \geq (3/2)^{k-1}$. Für die Anzahl an Rekursionsaufrufen ergibt sich:

$$t(x, y) = k \leq 1 + \log_{3/2} x = \mathcal{O}(\log x) = \mathcal{O}(\log x + \underbrace{\log y}_{\leq \log x \text{ da } x > y}).$$

Man beachte, daß sich für alle Zahlen $a, b > 1$ die Logarithmusfunktionen zu den Basen a und b nur um einen konstanten Faktor unterscheiden. Somit ist $\log_a x = \Theta(\log_b x)$, falls $a, b > 1$.

Satz 1.3.5. Die Laufzeit⁸ des Euklid-Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier natürlicher Zahlen $x \geq y \geq 1$ ist $\mathcal{O}(\log x + \log y) = \mathcal{O}(\log x)$.

Die Euklid-Algorithmus ist also polynomiell zeitbeschränkt, da die Eingabegröße in $\Theta(\log x + \log y)$ liegt. Wendet man dagegen die Rekursionsgleichungen $ggT(x, y) = ggT(y, x)$ und

$$ggT(x, y) = ggT(x - y, y) \text{ für } x > y$$

als Grundlage für einen rekursiven Algorithmus an, so kann keine polynomiale Zeitbeschränkung garantiert werden. Betrachtet man nämlich $ggT(x, 1)$, so finden mindestens $x - 1$ Rekursionsaufrufe statt:

$$ggT(x, 1) = ggT(x - 1, 1) = ggT(x - 2, 1) = \dots = ggT(1, 1) = 1.$$

Die genaue Anzahl an Rekursionsaufrufen hängt schlussendlich von der genauen Formulierung des Verfahrens ab.

⁸Hier sowie im Folgenden sprechen wir kurz von Laufzeit und meinen damit die Laufzeit im schlimmsten Fall.

2 Sortieren und Selektieren

Wir diskutieren zunächst die Fragestellung von Sortierproblemen. Gegeben ist eine Zahlenfolge x_1, \dots, x_n . Gesucht ist eine Permutation ℓ_1, \dots, ℓ_n der Zahlen $1, \dots, n$ mit $x_{\ell_1} \leq \dots \leq x_{\ell_n}$. Üblicherweise stehen die Werte x_j für sogenannte *Schlüsselwerte* von Datensätzen d_j mit diversen anderen Attributwerten. Wir schreiben dafür auch $x_j = \text{key}(d_j)$.

2.1 Internes Sortieren durch Vergleiche

Wir setzen voraus, daß die Zahlenfolge x_1, \dots, x_n in einem Array der Länge n abgelegt ist. Die in Abschnitt 2.1 vorgestellten Sortierverfahren werden für den Fall formuliert, daß die Schlüsselwerte x_j (reelle oder ganze) Zahlen sind. Die Verfahren sind jedoch ebenfalls anwendbar für andere „Schlüsseluniversen“, etwa Strings oder Tupel $x_j = \langle x_{j,1}, \dots, x_{j,k} \rangle$ von Werten $x_{j,i}$ gewisser Grundmengen D_i , $i = 1, \dots, k$. Die einzige Voraussetzung die an das Schlüsseluniversum gestellt wird ist, daß eine feste totale Ordnung \leq vorliegt.⁹ Bei den angegebenen Laufzeiten wird jedoch angenommen, daß der Vergleich zweier Schlüsselwerte in konstanter Zeit durchgeführt werden kann. Für Integer-Zahlen fester Länge (etwa dargestellt durch 16 Bits) ist dies unter dem uniformen Kostenmaß zutreffend. Für komplexere Schlüsselwerte, für welche der Vergleich wesentlich aufwendiger ist, ergibt sich die Laufzeit, indem die Anzahl der Vergleiche mit einem entsprechenden Faktor multipliziert wird.

2.1.1 Einfache Sortierverfahren mit quadratischer Laufzeit

Naive Sortierverfahren mit Laufzeit $\Theta(n^2)$ sind z.B. Sortieren durch Minimumssuche (Algorithmus 7), Sortieren durch Einfügen (Algorithmus 8) oder Bubblesort (Algorithmus 9). Die quadratische Laufzeit für die iterierte Minimumssuche im schlimmsten und durchschnittlichen (sogar im besten) Fall ist offensichtlich.

Liegt eine Array-Darstellung für das *Sortieren durch Einfügen* vor, so kann die Position, an welcher x_i einzufügen ist, mit einer Binären Suche erfolgen. Dies ändert allerdings nichts an der quadratischen worst-case Laufzeit, da das „Verschieben der Elemente“ größer als x_i im bereits sortierten Teil-Array linearen Aufwand (pro Schleifendurchlauf) erfordert. Im i -ten Schleifendurchlauf werden $\Theta(i + \log i)$ Schritte benötigt, wobei sich der Summand i auf das Umordnen des Arrays und der Summand $\log i$ auf die Binäre Suche bezieht. Insgesamt ergibt sich damit die Laufzeit $\Theta(n^2)$. Für eine Listenimplementierung entstehen die Kosten $\Theta(i)$ in der i -ten Iteration. Insgesamt also die Kosten $\Theta(n^2)$.

⁹Eine totale Ordnung auf einer Menge \mathcal{U} ist eine binäre, reflexive, transitive, antisymmetrische Relation \leq auf \mathcal{U} , so daß für je zwei Elemente $x_1, x_2 \in \mathcal{U}$ gilt: entweder $x_1 \leq x_2$ oder $x_2 \leq x_1$ (oder beides, also $x_1 = x_2$).

Algorithmus 7 Sortieren durch Minimumssuche

FOR $i = 1, \dots, n - 1$ **DO**
(* bestimme einen Index j mit $x_j = \min\{x_i, x_{i+1}, \dots, x_n\}$ *)
 $j := i$;
FOR $\ell = i + 1, \dots, n$ **DO**
 IF $x_\ell < x_j$ **THEN**
 $j := \ell$
 FI
 OD
 IF $x_i > x_j$ **THEN**
 $x := x_i$; $x_i := x_j$; $x_j := x$;
 (* vertausche x_i und x_j *)
 FI
OD

Algorithmus 8 Sortieren durch Einfügen

FOR $i = 2, \dots, n$ **DO**
 füge x_i in die sortierte Folge x_1, \dots, x_{i-1} ein
OD

Die Idee von *Bubblesort* besteht darin, solange benachbarte Elemente x_i, x_{i+1} zu vertauschen bis die Folge sortiert ist. Liegt eine (aufsteigend) sortierte Eingabefolge vor, so terminiert Bubblesort nach der ersten Iteration mit insgesamt $n - 1$ Vergleichen.

Algorithmus 9 Bubblesort

REPEAT
 FOR $i = 1, \dots, n - 1$ **DO**
 IF $x_i > x_{i+1}$ **THEN**
 vertausche x_i und x_{i+1}
 FI
 OD
UNTIL keine Veränderungen in der letzten Iteration

Dies ist offenbar der beste Fall für Bubblesort. Andernfalls wird in jeder Iteration das k -größte Element an die Position $n - k + 1$ verschoben; die Elemente an den Positionen $n - k + 2, \dots, n$ bleiben unverändert. Daher genügt es, die FOR-Schleife in der k -ten Iteration bis $n - k$ laufen zu lassen. Der schlechteste Fall für Bubblesort tritt daher für eine absteigend sortierte Eingabefolge ein, für welche in der k -ten Iteration das k -größte Element mit $n - k$ Vergleichen an Position $n - k + 1$ geschoben wird. Die worst-case Laufzeit von Bubblesort ist daher

$$\Theta\left(\sum_{k=1}^{n-1}(n-k)\right) = \Theta(n^2).$$

Tatsächlich genügt es sogar, in der FOR-Schleife nur bis zu dem größten Index m zu laufen, so daß in der vorangegangenen Iteration die Elemente x_m und x_{m+1} vertauscht wurden. (Initial ist $m = n - 1$.) Dies ändert jedoch nichts am schlimmsten Fall $m = n - k$ in der k -ten Iteration. Die Beobachtung, daß die Laufzeit von Bubblesort zwischen linear und quadratisch schwankt, hängt damit zusammen, daß in einer Iteration jedes Element höchstens um eine Position nach links verschoben werden kann, aber bis zu $n - 1$ Positionen nach rechts. Ersteres hat zu Folge, daß sobald das kleinste Element in der Eingabefolge an Position i steht, werden mindestens $i - 1$ Iterationen durchgeführt, wobei in der k -ten Iteration mindestens $i - k$ Vergleiche stattfinden. Daher ist

$$\frac{i(i-1)}{2} = \Omega(i^2)$$

eine untere Schranke für die Anzahl an Vergleichen, die Bubblesort durchführt, wenn das Minimum an Position i der Eingabefolge steht. Diese Überlegung kann ausgenutzt werden um nachzuweisen, daß (Gleichverteilung vorausgesetzt) die durchschnittliche Anzahl an Vergleichen $\Theta(n^2)$ ist. Nimmt man nämlich an, daß für jede Position i mit Wahrscheinlichkeit $1/n$ das kleinste Element an Position i steht, so ist die durchschnittliche Anzahl an Vergleichen durch

$$\sum_{i=1}^n \frac{1}{n} \cdot \frac{i(i-1)}{2} = \underbrace{\frac{1}{n} \cdot \sum_{i=1}^n \frac{i(i-1)}{2}}_{=\Omega(n^3)} = \Omega(n^2)$$

nach unten beschränkt. Da auch die worst-case Laufzeit quadratisch ist, sind $\Theta(n^2)$ die asymptotischen mittleren Kosten von Bubblesort.

Für Bubblesort und Sortieren durch Minimumssuche ist unmittelbar klar, daß nur konstanter zusätzlicher Platz benötigt wird. Auch beim Sortieren durch Einfügen kann man mit **einem** Array (ohne nennenswerten zusätzlichen Platzbedarf) auskommen, z.B., indem man im vorderen Array-Teil die bereits sortierte Folge ablegt und im hinteren Array-Teil die noch nicht betrachteten Elemente speichert.

Sortieren durch Einfügen und Bubblesort haben den Vorzug, daß keine Array-Darstellung der Datensätze erforderlich ist, da sie auch leicht mit Listen realisiert werden können. Im Gegensatz zum Sortieren durch Einfügen hat Bubblesort den Vorteil, daß auch für eine Realisierung mit Listen nur konstanter zusätzlicher Speicherbedarf entsteht.

2.1.2 Mergesort

Die Grundidee ist ein rekursiver Ansatz, bei dem für $n \geq 2$ die Zahlenfolge in zwei in etwa gleich große Teilfolgen x_1, \dots, x_m und x_{m+1}, \dots, x_n unterteilt werden und diese (rekursiv) sortiert werden. Die sich ergebenden sortierten Folgen werden dann zu einer sortierten Folge gemischt. Ein Beispiel ist in Abbildung 4 angegeben.

Eine rekursive Formulierung ist in Algorithmus 10 angegeben, wobei der initiale Aufruf mit $\ell = 1$ und $r = n$ statt zu finden hat. Das Mischen erfolgt nach dem Reißverschlußprinzip und ist in Algorithmus 11 angegeben.

Algorithmus 10 *MergeSort(ℓ, r)*

IF $\ell < r$ **THEN**
 $m := (\ell + r) \text{ div } 2;$
 MergeSort(ℓ, m);
 MergeSort($m + 1, r$);
 Mische die sortierten Teilstücke
FI

Algorithmus 11 Mischen der sortierten Zahlenfolgen x_1, \dots, x_m und y_1, \dots, y_k

$i := 1; x := x_1;$
 $j := 1; y := y_1;$
 $\ell := 0;$
WHILE $i \leq m$ oder $j \leq k$ **DO**
 $\ell := \ell + 1;$
 IF $x \leq y$ **THEN**
 $z_\ell := x; i := i + 1;$
 IF $i \leq m$ **THEN**
 $x := x_i$
 ELSE
 (* Ende der x -Folge erreicht *)
 WHILE $j \leq k$ **DO**
 $z_\ell := y_j; j := j + 1, \ell := \ell + 1$
 OD
 FI
 ELSE
 $z_\ell := y; j := j + 1;$
 IF $j \leq k$ **THEN**
 $y := y_j$
 ELSE
 (* Ende der y -Folge erreicht *)
 WHILE $j \leq k$ **DO**
 $z_\ell := x_i; i := i + 1, \ell := \ell + 1$
 OD
 FI
 FI
OD

Beispiel zu Mergesort:

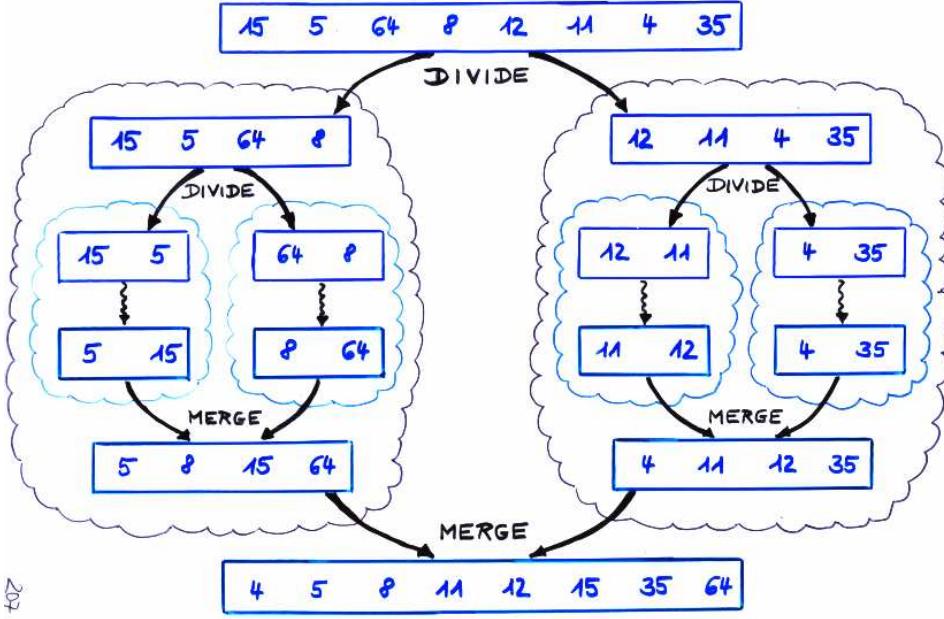


Abbildung 4: Beispiel zu Mergesort

Kostenanalyse für das Mischen. Zunächst stellen wir fest, daß (für beliebiges m und k) das Mischen den Aufwand $\Theta(m + k)$ erfordert, im besten, mittleren und schlimmsten Fall. Dies kann man sich dadurch klarmachen, indem man die Gesamtzahl an Vergleichen „ $x \leq y$ “ zählt. Diese ist durch $m + k - 1$ nach oben beschränkt. Im Kontext von Mergesort ist $m + k = n$. In der Mischphase werden also höchstens $n - 1$ Vergleiche durchgeführt. Tatsächlich ist $2 \cdot \min\{m, k\} - 1$ auch eine untere Schranke für **jedes Mischverfahren**, das als Eingabe zwei sortierte Folgen der Längen m und k hat. Dies kann man sich, etwa für $m \leq k$ anhand der Sortierung

$$x_1 < y_1 < x_2 < y_2 < \dots < x_m < y_m < y_{m+1} < \dots < y_k$$

klarmachen. Für diese sind die Vergleiche zwischen x_i und y_i , $i = 1, \dots, m$, und zwischen y_i und x_{i+1} , $i = 1, \dots, m - 1$, erforderlich, um auf die korrekte Sortierung schließen zu können.

Kostenanalyse von Mergesort. Die Kosten hinsichtlich der Laufzeit im schlechtesten Fall erfolgt durch Lösen der Rekurrenz: $T(1) = 0$ und

$$T(n) = n - 1 + T\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) + T\left(n - \left\lfloor \frac{n+1}{2} \right\rfloor\right)$$

für $n \geq 2$. Bevor wir die angegebene Rekurrenz lösen, machen wir uns deren Bedeutung klar. Für $n = 1$ findet kein Vergleich statt. Dies erklärt den Wert $T(1) = 0$. Der Summand $n - 1$ steht für das Mischen der sortierenden Teilfolgen. $T\left(\lfloor \frac{n+1}{2} \rfloor\right)$ steht für die Kosten,

die durch den Aufruf von $MergeSort(\ell, m)$ entstehen und $T\left(n - \lfloor \frac{n+1}{2} \rfloor\right)$ für die durch $MergeSort(m+1, r)$ verursachten Kosten, wobei $\ell = 1$, $r = n$ und $m = (\ell + r) \text{ div } 2$. Folgender Satz ergibt sich durch Lösen der genannten Rekurrenz:

Satz 2.1.1 (Kosten von Mergesort). Die worst-case Laufzeit von Mergesort ist $\Theta(n \log n)$.

Nun zum Beweis von Satz 2.1.1. Zunächst betrachten wir den Fall, daß $n = 2^k$ eine Zweierpotenz ist. Dies erlaubt es, das Abrunden innerhalb der Rekurrenz zu ignorieren. Wir erhalten für $n \geq 2$:

$$T(n) = n - 1 + 2 \cdot T\left(\frac{n}{2}\right).$$

Durch Induktion nach k läßt sich zeigen, daß $T(n) = n \log n - n + 1$. Wir verfahren wie bei der Analyse der Binären Suche und rechnen „rückwärts“ zur Rekursionsverankerung (Induktionsanfang $n = 1$) zurück:

$$\begin{aligned} T(n) &= n - 1 + 2 \cdot T\left(\frac{n}{2}\right) \\ &= (n - 1) + 2\left(\frac{n}{2} - 1\right) + 4 \cdot T\left(\frac{n}{4}\right) \\ &= (n - 1) + 2\left(\frac{n}{2} - 1\right) + 4\left(\frac{n}{4} - 1\right) + 8 \cdot T\left(\frac{n}{8}\right) \\ &\vdots \\ &= \sum_{j=0}^r 2^j \left(\frac{n}{2^j} - 1\right) + 2^{r+1} \cdot T\left(\frac{n}{2^{r+1}}\right) \end{aligned}$$

Dabei ist $0 \leq r \leq k - 1$, wobei $n = 2^k$. Mit $r = k - 1$ erhalten wir:

$$\begin{aligned} T(n) &= \sum_{j=0}^{k-1} 2^j \cdot \left(\frac{n}{2^j} - 1\right) + 2^k \cdot T\left(\frac{n}{2^k}\right) \\ &= \sum_{j=0}^{k-1} (n - 2^j) + 2^k \cdot T\left(\frac{n}{2^k}\right) \\ &= n \cdot k - \underbrace{\frac{2^k - 1}{2 - 1}}_{=2^k-1=n-1} + n \cdot \underbrace{T(1)}_{=0} \\ &= n \cdot \log n - n - 1 \end{aligned}$$

Der Vollständigkeit wegen erläutern wir noch, wieso die Betrachtung des Sonderfalls von Zweierpotenzen ausreichend ist. Ist n eine beliebige natürliche Zahl, so erhalten wir aufgrund der Monotonie von T :

$$T(2^{\lfloor \log n \rfloor}) \leq T(n) \leq T(2^{\lceil \log n \rceil})$$

Wegen $2^{\lfloor \log n \rfloor} > \frac{n}{2}$ und $2^{\lceil \log n \rceil} < 2n$ ergibt sich durch Einsetzen in obige Formel:

$$\frac{n}{2}(\log n - 1) - \frac{n}{2} + 1 \leq T(n) \leq 2n(\log n + 1) - 2n + 1$$

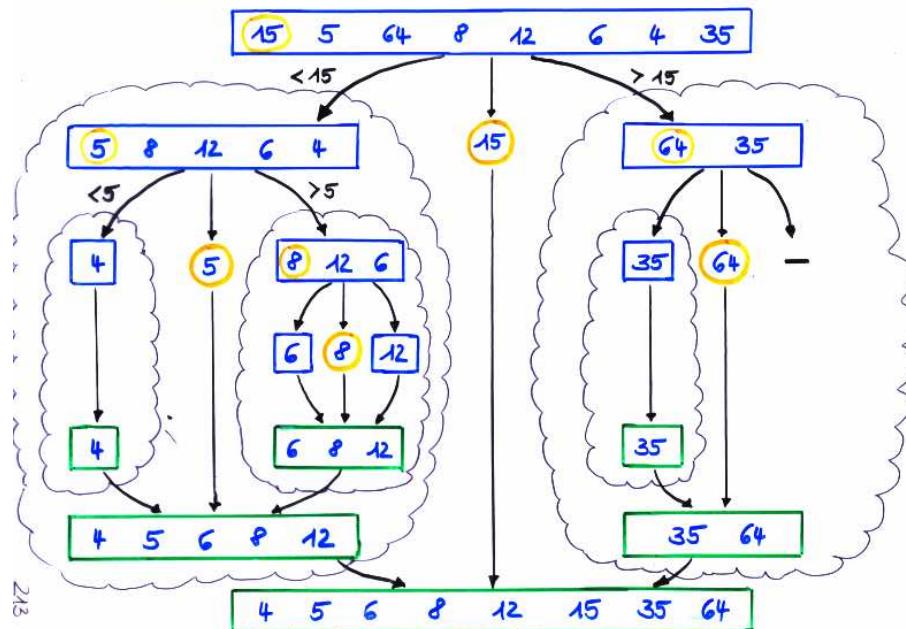
und somit $T(n) = \Theta(n \log n)$.

Die mittlere Laufzeit von Mergesort ist ebenfalls $\Theta(n \log n)$. Dies ergibt sich mit ähnlichen Argumenten und beruht auf der Tatsache, daß auch für den besten Fall die Rekurrenz für Mergesort die Form $T(n) = \Theta(n) + 2T(\frac{n}{2})$ hat, deren Lösung ebenfalls $\Theta(n \log n)$ ist.

2.1.3 Quicksort

Quicksort ist eines der berühmtesten Sortierverfahren, das von Hoare in den 60er Jahren entwickelt wurde und das in der Praxis weit verbreitet ist. Die Grundidee ist die zu sortierende Zahlenfolge x_1, \dots, x_n hinsichtlich eines *Pivotelements* $x \in \{x_1, \dots, x_n\}$ in zwei Folgen zu unterteilen, diese (rekursiv) zu sortieren und dann die sortierten Teilfolgen „zusammenzusetzen“. Die Aufteilung bzgl. des Pivotelements x erfolgt durch die Zerlegung in eine Folge L_1 , die alle Elemente x_i mit $x_i < x$ erfaßt, und eine Folge L_2 , die alle Elemente x_i mit $x_i > x$ enthält. Kommt x mehrfach in x_1, \dots, x_n vor, so können die Duplikate in L_1 oder L_2 aufgenommen werden.

Beispiel zu Quicksort:



Durch geeignetes Vertauschen von Array-Elementen können die Teilfolgen L_1 und L_2 im Array selbst dargestellt werden. Dies wird in dem umgangssprachlich formulierten Algorithmus 12 angedeutet. Dieser ist mit $\ell = 1$ und $r = n$ zu starten. Intuitiv besteht L_1 nach der Umordnung aus den Elementen x_ℓ, \dots, x_{m-1} und L_2 aus den Array-Elementen x_{m+1}, \dots, x_r .

Algorithmus 12 *Quicksort*(ℓ, r)

IF $r - \ell \geq 1$ **THEN**

wähle ein Pivot-Element $x \in \{x_\ell, x_{\ell+1}, \dots, x_{r-1}, x_r\}$.

Ordne den Array-Abschnitt x_ℓ, \dots, x_r so um, daß alle Elemente $< x$ links von x und alle Elemente $> x$ rechts von x stehen.

Sei m die Position von x im umsortierten Array.

Quicksort($\ell, m - 1$);

Quicksort($m + 1, r$);

FI

Z.B. kann das erste Array-Element (also x_1 im ersten Schritt) stets als Pivot-Element gewählt werden. Ebenso kann man natürlich eine andere feste Array-Position wählen (das letzte, das mittlere, etc.) oder auch das Pivot-Element zufällig wählen („randomisiertes Quicksort“).

Eine iterative Formulierung der Umordnungsphase von Quicksort, in welcher das *erste* Array-Element als Pivot-Element ausgewählt wird, ist in Algorithmus 13 angegeben.

Algorithmus 13 die Umordnungsphase von Quicksort

$x := x_1$; (* wähle das erste Element als Pivot-Element *)

$\ell := 2; r := n$;

WHILE $\ell < r$ **DO**

WHILE $x_\ell < x$ und $\ell \leq n$ **DO**

$\ell := \ell + 1$

OD

WHILE $x_r \geq x$ **DO**

$r := r - 1$

OD

IF $\ell < r$ **THEN**

$z := x_\ell; x_\ell := x_r; x_r := z$

FI

OD

(* Setze das Pivot-Element x an die richtige Position *)

IF $r > 1$ **THEN**

$x_1 := x_r; x_r := x$

FI

Für die Kostenanalyse argumentieren wir mit der Anzahl an Vergleichen der Array-Elemente x_i mit dem Pivotelement x , die für das Umordnung (die Zerlegung in Teilfolgen mit Elementen kleiner x bzw. größer x) benötigt werden. In der ersten Iteration sind hierfür $n - 1$ Vergleiche notwendig, da wir jedes Array-Element mit dem Pivot-Element x

vergleichen müssen.¹⁰ Wir setzen im Folgenden eine deterministische (nicht-randomisierte) Version zur Wahl des Pivot-Elements voraus. Auf die randomisierte Variante gehen wir später ein.

Worst-Case Analyse von Quicksort. Für den schlimmsten Fall ergibt sich die Kostenfunktion für die Laufzeit durch Lösen der Rekurrenz $T(0) = T(1) = 0$ und für $n \geq 2$

$$T(n) = (n - 1) + \max_{1 \leq i \leq n} (T(n - i) + T(i - 1)).$$

Diese Rekurrenz sieht zunächst etwas unhandlich aus, jedoch kann man sich leicht überlegen, daß der schlechteste Fall dann eintritt, wenn in jedem Rekursionsschritt eine der beiden Teilfolgen leer ist, also wenn das Pivot-Element das kleinste bzw. größte Element ist. Wählt man z.B. stets das erste Array-Element als Pivot-Element, so liegt dieser Extremfall vor, wenn das ursprüngliche Array bereits sortiert ist. Wir können also obige Rekurrenz durch $T(n) = (n - 1) + T(n - 1)$ ersetzen und erhalten die Lösung

$$T(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

Der Summand i steht für die Anzahl an Vergleichen, welche in der $(n - i + 1)$ -ten Rekursion durchgeführt werden. Im schlimmsten Fall hat Quicksort also (wie auch die naiven Sortierverfahren) die Laufzeit $\Theta(n^2)$.

Bevor wir uns der Analyse im Mittel widmen, betrachten wir kurz den *besten Fall*. Dieser tritt ein, wenn in jedem Rekursionsaufruf die gebildeten Teilfolgen (in etwa) gleich groß sind, also wenn das Pivot-Element das mittlere Element der sortierten Folge ist. Nehmen wir an, daß $n = 2^k$ eine Zweierpotenz ist, dann ist die Anzahl an Vergleichen durch

$$T(n) = n - 1 + 2 \cdot T\left(\frac{n}{2}\right)$$

gegeben, deren Lösung $\Theta(n \log n)$ ist. (Vgl. Analyse von Mergesort.)

Average-Case Analyse von Quicksort. Die erwartete Anzahl an Vergleichen, unter der Annahme einer Gleichverteilung ergibt sich aus der Rekurrenz $T(0) = T(1) = 0$ und für $n \geq 2$

$$T(n) = n - 1 + \frac{1}{n} \cdot \sum_{i=1}^n (T(i - 1) + T(n - i))$$

Diese ergibt sich aus der Annahme, daß das Pivot-Element mit Wahrscheinlichkeit $1/n$ das i -kleinste Element ist (bzw. an i -ter Stelle in der sortierten Folge steht) und daß die erwartete Anzahl an Vergleichen in diesem Fall gleich $(n - 1) + T(i - 1) + T(n - i)$ ist.

¹⁰Für das in Algorithmus 13 angegebene Umordnungsverfahren beträgt die Anzahl an Vergleichen „ $x_\ell < x$ “ und „ $x_r \geq x$ “ höchstens n . Um ebenfalls die obere Schranke $n - 1$ sicherzustellen, kann der letzte Vergleich „ $x_r \geq x$ “ für $r = \ell$ eingespart werden.

Obige Formel ergibt sich durch Aufsummieren dieser Werte, jeweils gewichtet mit der Wahrscheinlichkeit $1/n$.

Zunächst können wir die angegebene Rekurrenz umformulieren zu

$$T(n) = n - 1 + \frac{2}{n} \sum_{i=1}^n T(i-1).$$

Durch Induktion nach n kann man zeigen, daß

$$T(n) = 2(n+1) \cdot H(n) - 4n,$$

wobei

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = \sum_{i=1}^n \frac{1}{i}$$

die n -te Partialsumme der Harmonischen Reihe ist. Der Nachweis dieser Aussage kann durch Induktion nach n erfolgen.

Beweis. Induktionsanfang: $T(0) = T(1) = 0$ und für $n \geq 2$:

$$\begin{aligned}
T(n) &= n - 1 + \frac{2}{n} \cdot \sum_{i=1}^n T(i-1) \\
&= n - 1 + \frac{2}{n} \cdot \sum_{i=1}^{n-1} T(i) \\
&\stackrel{IV}{=} n - 1 + \frac{2}{n} \cdot \sum_{i=1}^{n-1} [2(i+1) \cdot H(i) - 4i] \\
&= n - 1 + \frac{4}{n} \cdot \sum_{i=1}^{n-1} (i+1) \cdot H(i) - \frac{8}{n} \cdot \underbrace{\sum_{i=1}^{n-1} i}_{=\frac{n(n-1)}{2}} \\
&= n - 1 + \frac{4}{n} \cdot \sum_{i=1}^{n-1} (i+1) \cdot H(i) - 4(n-1) \\
&= -3(n-1) + \frac{4}{n} \cdot \sum_{i=1}^{n-1} (i+1) \cdot H(i) \\
&= -3(n-1) + \frac{4}{n} \cdot \sum_{i=1}^{n-1} (i+1) \cdot \sum_{j=1}^i \frac{1}{j} \\
&= -3(n-1) + \frac{4}{n} \cdot \sum_{j=1}^{n-1} \frac{1}{j} \cdot \underbrace{\sum_{i=j}^{n-1} (i+1)}_{=\frac{n(n+1)}{2} - \frac{j(j+1)}{2}} \\
&= -3(n-1) + \frac{4}{n} \cdot \left(\sum_{j=1}^{n-1} \frac{1}{j} \right) \cdot n \cdot \frac{(n+1)}{2} - \frac{2}{n} \cdot \underbrace{\sum_{j=1}^{n-1} (j+1)}_{=\frac{n(n+1)}{2} - 1} \\
&= -3(n-1) + 2(n+1) \cdot H(n-1) - (n+1) + \frac{2}{n} \\
&= -3(n-1) + 2(n+1) \cdot H(n) - \underbrace{2(n+1) \frac{1}{n}}_{=2+\frac{2}{n}} - n - 1 + \frac{2}{n} \\
&= -3(n-1) + 2(n+1) \cdot H(n) - 2 - \frac{2}{n} - n - 1 + \frac{2}{n} \\
&= -3n + 3 + 2(n+1) \cdot H(n) - n - 3 \\
&= 2(n+1) \cdot H(n) - 4n
\end{aligned}$$

□

Eine Integralabschätzung der Partialsummen der Harmonischen Reihe wie in Abbildung 5 angedeutet liefert

$$H(n) = \Theta(\ln n) = \Theta(\log n).$$

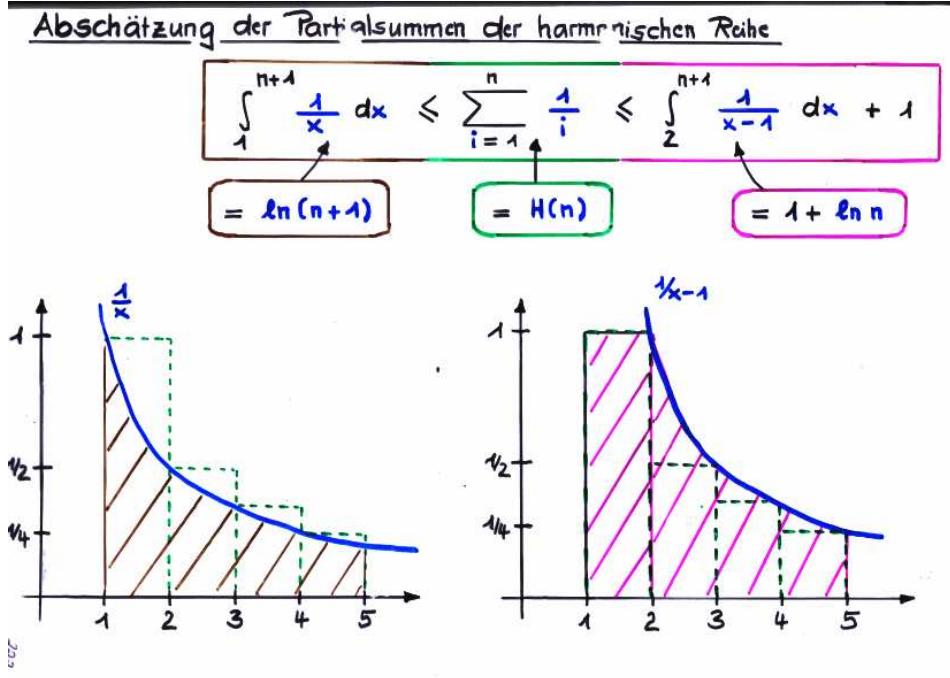


Abbildung 5: Integralabschätzung von $H(n)$

Wir erhalten $T(n) = \Theta(n \log n)$. Wir halten unsere Resultate in folgendem Satz fest:

Satz 2.1.2 (Worst-case Laufzeit von Quicksort). Die worst-case Laufzeit von Quicksort ist $\Theta(n^2)$. Die durchschnittliche Laufzeit von Quicksort ist $\Theta(n \log n)$.

Zwei abschließende Bemerkungen zu Quicksort, die sich auf die Platzkomplexität und die randomisierte Version von Quicksort beziehen.

Bemerkung 2.1.3 (Platzbedarf von Quicksort). Für eine iterative Formulierung kommt man offenbar mit konstantem zusätzlichen Platz aus. Interessanter ist eine rekursive Formulierung wie in Algorithmus 12. Der benötigte zusätzliche Platz wird hier durch die Höhe des Rekursionsstacks dominiert. Da der zweite Rekursionsaufruf ganz am Ende von $Quicksort(\ell, r)$ stattfindet (man spricht auch von einer *tail-recursion*), können unmittelbar vor dem Aufruf der zweiten Rekursion alle Daten, die im Stack für $Quicksort(\ell, r)$ gespeichert sind, gelöscht werden.¹¹ Da pro Rekursionsaufruf nur konstanter Platz im

¹¹Viele Compiler erkennen solche tail-recursions und löschen nicht mehr benötigte Referenzen und Daten „automatisch“.

Rekursionsstack benötigt wird, kann die maximale Höhe des Rekursionsstacks salopp durch

$$S(n) = 1 + S(\underbrace{\text{Länge des linken Teil-Arrays}}_{=m-\ell})$$

angegeben werden. Wir gehen dabei von der Formulierung wie in Algorithmus 12 aus, in der der Rekursionsaufruf für das linke Teil-Array zuerst stattfindet. Der Summand 1 steht in obiger Formel für die (konstanten) Platzeinheiten, die pro Rekursionsaufruf im Stack benötigt werden. Für den schlimmsten Fall haben wir es mit der Rekurrenz

$$S(n) = 1 + S(n - 1)$$

zu tun, da im worst-case das Pivot-Element das größte Element der Zahlenfolge ist. Die Lösung der obigen Gleichung ist $S(n) = \Theta(n)$.

Algorithmus 14 *Quicksort(ℓ, r)*

IF $r - \ell \geq 1$ **THEN**

wähle ein Pivot-Element $x \in \{x_\ell, x_{\ell+1}, \dots, x_{r-1}, x_r\}$.

Ordne den Array-Abschnitt x_ℓ, \dots, x_r so um, daß alle Elemente $< x$ links von x und alle Elemente $> x$ rechts von x stehen.

Sei m die Position von x im umsortierten Array.

IF $m - \ell \leq r - m$ **THEN**

Quicksort($\ell, m - 1$);
Quicksort($m + 1, r$) (* tail-recursion *)

ELSE

Quicksort($m + 1, r$);
Quicksort($\ell, m - 1$) (* tail-recursion *)

FI

FI

Die Reihenfolge, in welcher $Quicksort(\ell, m - 1)$ und $Quicksort(m + 1, r)$ aufgerufen werden, ist jedoch irrelevant. Daher können wir auch stets zuerst die Rekursion für das kürzere Teil-Array aufrufen (siehe Algorithmus 14), um so die Höhe des Rekursionsstacks durch die Rekurrenz

$$S(n) = 1 + S(\underbrace{\text{Länge des kürzere Teil-Arrays}}_{\leq \lfloor \frac{n}{2} \rfloor})$$

zu beschränken. Da das kürzere Teil-Array höchstens $\lfloor \frac{n}{2} \rfloor$ Elemente enthalten kann, erhalten wir die Rekurrenz:

$$S(n) = 1 + S(\lfloor \frac{n}{2} \rfloor)$$

Deren Lösung ist $\Theta(\log n)$. Siehe Analyse der Binären Suche. □

Bemerkung 2.1.4 (Randomisiertes Quicksort). Betrachtet man die *randomisierte Version* von Quicksort, in welcher das Pivot-Element zufällig (mit Wahrscheinlichkeit $1/n$ für jedes der n Array-Elemente) gewählt wird, so erhält man $\Theta(n \log n)$ als erwartete Laufzeit. Dies ergibt sich mit denselben Argumenten, wie sie in der Durchschnittsanalyse eingesetzt wurden. Der Unterschied zur deterministischen Variante und deren mittlere Laufzeit besteht darin, daß die Laufzeit nun eine zufällige Größe ist, die nicht mehr von der konkreten Eingabe abhängt. Für den probabilistischen Fall gibt es also keine „schlechten“ oder „guten“ Eingaben. Ein Vorzug der probabilistischen Variante ist daher, daß Gleichverteilung durch die randomisierte Wahl des Pivot-Elements *garantiert* wird, während für die Durchschnittsanalyse der deterministischen Variante eine derartige Voraussetzung über die Eingaben gemacht wurde. Letzteres kann in konkreten Anwendungen unrealistisch sein. \square

2.1.4 Untere Schranke für interne Sortierverfahren mit Vergleichen

Mergesort ist optimal in dem Sinn, daß sich die asymptotische Laufzeit nicht unterbieten lässt. Diese Beobachtung ist in dem folgenden Satz formuliert.

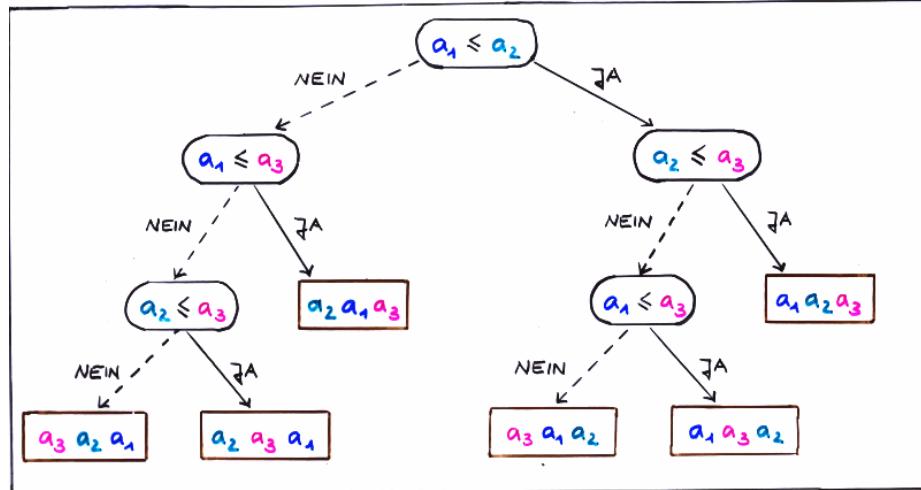
Satz 2.1.5 (Untere Schranke für den schlimmsten Fall). Jedes (allgemeine) interne Sortierverfahren, das auf Vergleichen beruht und keine weiteren Vorkenntnisse über die zu sortierende Zahlenfolge hat, benötigt im schlimmsten Fall $\Omega(n \log n)$ Vergleiche. Dabei ist n die Länge der zu sortierenden Zahlenfolge.

Beweis. Der Beweis von Satz 2.1.5 beruht auf der Inspektion des *Entscheidungsbaums*, der jedem Sortierverfahren mit Vergleichen für festes n zugeordnet werden kann. Dieser zeigt die durchgeföhrten Vergleiche (Knoten) und deren zeitliche Abfolge. Im Folgenden gehen wir davon aus, daß alle redundanten Vergleiche gestrichen wurden und ein Entscheidungsbaum vorliegt, in dem jeder Knoten durch eine geeignet gewählte Eingabefolge erreichbar ist und jeder innere Knoten genau zwei Söhne hat. Die *Höhe* des Entscheidungsbaums ist gleich der Länge eines längsten möglichen Berechnungspfads des betreffenden Sortierverfahrens und gibt damit Aufschluß über die Anzahl an Vergleichen im schlimmsten Fall.¹² Folgende Folie zeigt ein konkretes Beispiel für einen Entscheidungsbaum für $n = 3$ eines möglichen Sortierverfahrens.

¹²Zur Erinnerung: Die *Höhe* eines Baums \mathcal{T} ist die Länge eines längsten Wegs in \mathcal{T} , wobei die Weglänge an der Anzahl an durchlaufenen Kanten gemessen wird. Ein längster Pfad in einem Baum führt daher stets von der Wurzel zu einem *Blatt*, womit ein Knoten ohne Nachfolger gemeint ist.

Möglicher Entscheidungsbaum für ein Sortierverfahren ($n = 3$)

zu sortierende Folge: a_1, a_2, a_3



226

Der Entscheidungsbaum jedes auf elementaren Vergleichen basierende Sortierverfahrens hat mindestens $n!$ Blätter (die für die potentiellen Sortierungen der Zahlenfolge stehen) und somit die Mindesthöhe $\log(n!)$. Dies resultiert aus der Tatsache, daß jeder Binärbaum \mathcal{T} der Höhe h höchstens 2^h Blätter hat. Diese Aussage kann leicht durch Induktion nach h bewiesen werden. Im Folgenden sei $\beta(\mathcal{T})$ die Anzahl an Blättern in \mathcal{T} .

- Ein Baum der Höhe $h = 0$ besteht aus der Wurzel. Diese ist ein Blatt. Also gilt $\beta(\mathcal{T}) = 1 = 2^0 = 2^h$.
- Liegt ein Binärbaum der Höhe $h \geq 1$ vor, so haben die beiden Teilbäume \mathcal{T}_L und \mathcal{T}_R höchstens die Höhe $h - 1$.¹³ Nach Induktionsvoraussetzung gilt:

$$\beta(\mathcal{T}_L) \leq 2^{h-1}, \quad \beta(\mathcal{T}_R) \leq 2^{h-1}$$

Wir erhalten $\beta(\mathcal{T}) = \beta(\mathcal{T}_L) + \beta(\mathcal{T}_R) \leq 2 \cdot 2^{h-1} = 2^h$.

Eine Abschätzung von $\log(n!)$ ergibt die untere Schranke $\Omega(n \log n)$. Nimmt man z.B. an, daß n gerade ist, so gilt:

$$n! = \underbrace{n \cdot (n-1) \cdot \dots \cdot (\frac{n}{2}+1)}_{\frac{n}{2} \text{ Faktoren jeweils } \geq \frac{n}{2}} \cdot \underbrace{\frac{n}{2} \cdot \dots \cdot 2 \cdot 1}_{\geq 1} \geq \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

¹³In unserem Fall von Entscheidungsbäumen haben wir es mit Binärbäumen zu tun, in denen jeder innere Knoten zwei (nicht-leere) Teilbäume hat. Die angegebene Höhen-Blatt-Abschätzung gilt jedoch auch für beliebige Binärbäume, in welchen einer der beiden Teilbäume \mathcal{T}_L oder \mathcal{T}_R „leer“ sein kann.

und somit

$$\log(n!) \geq \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2} \cdot \log\left(\frac{n}{2}\right) = \Theta(n \log n).$$

Für ungerades n kann eine analoge Rechnung durchgeführt werden. \square

Bemerkung 2.1.6 (Anzahl an Blättern im Entscheidungsbaum). Im Beweis von Satz 2.1.5 genügte die Aussage, daß $\beta(\mathcal{T}) \geq n!$, wobei \mathcal{T} der Entscheidungsbaum eines Sortierverfahrens mit Vergleichen für Eingabefolgen der Länge n ist. Tatsächlich ist $n!$ sogar die *genaue* Anzahl an Blättern, wenn man voraussetzt, daß keine redundanten Vergleiche vorkommen. Wäre $\beta(\mathcal{T}) > n!$, so gäbe es eine Sortierung x_{k_1}, \dots, x_{k_n} , welche durch mindestens zwei Blätter v und w repräsentiert wird.

Sei u ein Knoten im Entscheidungsbaum, so daß v und w in verschiedenen Teilbäumen von u liegen. Wir nehmen nun an, daß „ $x_i \leq x_j$ “ der durch u dargestellte Vergleich ist. Sei $i = k_\ell$ und $j = k_r$. Ist $\ell < r$ und liegt etwa v in demjenigen Teilbaum, der über den NEIN-Zweig von u erreicht wird, dann wird Knoten v nur für

$$x_{k_\ell} = x_i > x_j = x_{k_r}$$

erreicht. Dies ist nicht möglich, wenn v für die Sortierung $x_{k_1}, \dots, x_{k_\ell}, \dots, x_{k_r}, \dots, x_{k_n}$ steht. Für den Fall $\ell > r$ betrachten wir eine Eingabefolge x_1, \dots, x_n , welche über den JA-Zweig von u zu dem betreffenden Blatt v oder w führt und welche aus paarweise verschiedenen Elementen besteht. Der JA-Zweig von u kann nur durchlaufen werden, wenn

$$x_{k_\ell} = x_i < x_j = x_{k_r}.$$

Dies steht jedoch im Widerspruch zur Annahme, daß $\ell > r$ und daß v und w für die Sortierung $x_{k_1}, \dots, x_{k_r}, \dots, x_{k_\ell}, \dots, x_{k_n}$ stehen. Analoge Argumente schließen andere Vergleiche, etwa „ $x_i < x_j$ “ oder „ $x_i \geq x_j$ “, aus. Diese Überlegungen zeigen, daß $\beta(\mathcal{T}) = n!$. \square

Die untere Schranke $\Omega(n \log n)$ gilt auch für die durchschnittliche Anzahl von Vergleichen für allgemeine Sortierverfahren.

Satz 2.1.7 (Untere Schranke für den durchschnittlichen Fall). Jedes (allgemeine) interne Sortierverfahren, das auf Vergleichen beruht und keine weiteren Vorkenntnisse über die zu sortierende Zahlenfolge hat, benötigt im Mittel $\Omega(n \log n)$ Vergleiche. Dabei ist n die Länge der zu sortierenden Zahlenfolge.

Beweis. Wir argumentieren wie im Beweis von Satz 2.1.5 mit dem Entscheidungsbaum \mathcal{T} , der jedem Sortierverfahren für Eingabefolgen der Länge n zugeordnet werden kann, und zeigen, daß dessen *mittlere Höhe* durch $\Omega(n \log n)$ nach unten beschränkt ist. Die mittlere Höhe $\bar{H}(\mathcal{T})$ ist wie folgt definiert:

$$\bar{H}(\mathcal{T}) = \frac{1}{\beta(\mathcal{T})} \cdot \sum_{v \in B(\mathcal{T})} \text{Tiefe}_{\mathcal{T}}(v)$$

Dabei ist $B(\mathcal{T})$ die Menge aller Blätter in \mathcal{T} . Wie zuvor bezeichnet $\beta(\mathcal{T}) = |B(\mathcal{T})|$ die Anzahl an Blättern in \mathcal{T} . Der Wert $Tiefe_{\mathcal{T}}(v)$ ist der Abstand von v zum Wurzelknoten, d.h., die Anzahl an Kanten, die auf dem Weg von der Wurzel zu v liegen. $\bar{H}(\mathcal{T})$ ist also die erwartete Länge eines Pfads von der Wurzel zu einem Blatt, wobei Gleichverteilung aller möglichen Sortierungen (also Wahrscheinlichkeit $1/\beta(\mathcal{T})$ für jede mögliche Ergebnisfolge x_{k_1}, \dots, x_{k_n}) vorausgesetzt wird. Dies entspricht offenbar der erwarteten Anzahl an Vergleichen des betreffenden Sortierverfahrens. Wir zeigen nun durch Induktion nach der Höhe von \mathcal{T} , daß

$$\bar{H}(\mathcal{T}) \geq \log \beta(\mathcal{T}).$$

Ist die Höhe von \mathcal{T} gleich 0, so besteht \mathcal{T} aus dem Wurzelknoten. Dieser ist ein Blatt der Tiefe 0 und es gilt:

$$\bar{H}(\mathcal{T}) = 0 = \log 1 = \log \beta(\mathcal{T})$$

Wir nehmen nun an, daß \mathcal{T} die Höhe ≥ 1 hat. Weiter seien \mathcal{T}_L und \mathcal{T}_R der linke bzw. rechte Teilbaum (des Wurzelknotens) von \mathcal{T} . Dann ist $B(\mathcal{T})$ die disjunkte Vereinigung von $B(\mathcal{T}_L)$ und $B(\mathcal{T}_R)$ und es gilt $\beta(\mathcal{T}_L), \beta(\mathcal{T}_R) \geq 1$ und $\beta(\mathcal{T}) = \beta(\mathcal{T}_L) + \beta(\mathcal{T}_R)$. Weiter gilt für jeden Knoten v in \mathcal{T}_L :

$$Tiefe_{\mathcal{T}}(v) = Tiefe_{\mathcal{T}_L}(v) + 1$$

und die analoge Formel für alle Knoten in \mathcal{T}_R . Wir erhalten:

$$\begin{aligned} \bar{H}(\mathcal{T}) &= \frac{1}{\beta(\mathcal{T})} \cdot \sum_{v \in B(\mathcal{T})} Tiefe_{\mathcal{T}}(v) \\ &= \frac{1}{\beta(\mathcal{T})} \cdot \left(\sum_{v \in B(\mathcal{T}_L)} Tiefe_{\mathcal{T}}(v) + \sum_{v \in B(\mathcal{T}_R)} Tiefe_{\mathcal{T}}(v) \right) \\ &= \frac{1}{\beta(\mathcal{T})} \cdot \left(\sum_{v \in B(\mathcal{T}_L)} (Tiefe_{\mathcal{T}_L}(v) + 1) + \sum_{v \in B(\mathcal{T}_R)} (Tiefe_{\mathcal{T}_R}(v) + 1) \right) \\ &= \frac{1}{\beta(\mathcal{T})} \cdot \left(\sum_{v \in B(\mathcal{T}_L)} Tiefe_{\mathcal{T}_L}(v) + \sum_{v \in B(\mathcal{T}_R)} Tiefe_{\mathcal{T}_R}(v) \right) + \underbrace{\frac{\beta(\mathcal{T}_L) + \beta(\mathcal{T}_R)}{\beta(\mathcal{T})}}_{=1} \\ &= \frac{\beta(\mathcal{T}_L)}{\beta(\mathcal{T})} \cdot \bar{H}(\mathcal{T}_L) + \frac{\beta(\mathcal{T}_R)}{\beta(\mathcal{T})} \cdot \bar{H}(\mathcal{T}_R) + 1 \end{aligned}$$

Die Induktionsvoraussetzung angewandt auf \mathcal{T}_L und \mathcal{T}_R liefert:

$$\bar{H}(\mathcal{T}_L) \geq \log \beta(\mathcal{T}_L), \quad \bar{H}(\mathcal{T}_R) \geq \log \beta(\mathcal{T}_R).$$

Einsetzen in obige Formel liefert:

$$\begin{aligned}
\bar{H}(\mathcal{T}) &= \frac{\beta(\mathcal{T}_L)}{\beta(\mathcal{T})} \cdot \bar{H}(\mathcal{T}_L) + \frac{\beta(\mathcal{T}_R)}{\beta(\mathcal{T})} \cdot \bar{H}(\mathcal{T}_R) + 1 \\
&\geq \frac{\beta(\mathcal{T}_L)}{\beta(\mathcal{T})} \cdot \log \beta(\mathcal{T}_L) + \frac{\beta(\mathcal{T}_R)}{\beta(\mathcal{T})} \cdot \log \beta(\mathcal{T}_R) + 1 \\
&= \frac{\beta(\mathcal{T}_L) \log \beta(\mathcal{T}_L) + \beta(\mathcal{T}_R) \log \beta(\mathcal{T}_R)}{\beta(\mathcal{T})} + 1
\end{aligned}$$

Da die Funktion $[0, b] \rightarrow \mathbb{R}$, $x \mapsto x \log x + (b - x) \log(b - x)$ ihr Minimum bei $x = \frac{b}{2}$ annimmt¹⁴, erhalten wir mit $b = \beta(\mathcal{T})$:

$$\begin{aligned}
\beta(\mathcal{T}_L) \log \beta(\mathcal{T}_L) + \beta(\mathcal{T}_R) \log \beta(\mathcal{T}_R) &\geq \beta(\mathcal{T}) \cdot \log \frac{\beta(\mathcal{T})}{2} \\
&= \beta(\mathcal{T}) \cdot (\log \beta(\mathcal{T}) - 1)
\end{aligned}$$

Wir setzen dies in obige Formel ein und erhalten:

$$\begin{aligned}
\bar{H}(\mathcal{T}) &\geq \frac{\beta(\mathcal{T}) \cdot (\log \beta(\mathcal{T}) - 1)}{\beta(\mathcal{T})} + 1 \\
&= (\log \beta(\mathcal{T}) - 1) + 1 = \log \beta(\mathcal{T})
\end{aligned}$$

Mit Hilfe von Bemerkung 2.1.6 erhalten wir:

$$\bar{H}(\mathcal{T}) \geq \log \beta(\mathcal{T}) = \log(n!) = \Omega(n \log n)$$

□

Die nachfolgende Folie fasst die Ergebnisse dieses Abschnitts zusammen.

¹⁴Dabei ist $0 \log 0$ als 0 zu lesen.

Laufzeiten von Sortier - Verfahren		[interne Verfahren]
	worst-case	average-case
Sortieren durch Vergleiche Sortiert von nat. Zahlen $\leq N$	"naive" Verfahren	$\Theta(n^2)$
	Mergesort	$\Theta(n \log n)$
	Quicksort	$\Theta(n^2)$
	Heapsort	$\Theta(n \log n)$
	Countingsort	$\Theta(n + N)$
	Bucketsort	$\Theta(n + N)$
Radixsort		$\Theta(k \cdot (n + N))$
lexikographisches Sortieren von k -Tupeln		

2.2 Sortieren in linearer Zeit

In den Sätzen 2.1.5 und 2.1.7 haben wir $\Omega(n \log n)$ als untere Schranke für die Laufzeit von Sortierverfahren mit Vergleichen erhalten. Für Sortierverfahren, die gewisse Vorkenntnisse über die zu sortierende Zahlenfolge verwenden können und nicht auf Vergleiche angewiesen sind, kann diese untere Schranke jedoch unterboten werden.

Gegeben ist eine Folge x_1, \dots, x_n ganzer Zahlen $x_j \in \{0, 1, \dots, N\}$ in Array-Darstellung. Wie zu Beginn von Abschnitt 2 erläutert, stehen die Zahlen x_j für gewisse Attributwerte von (evtl. komplexen) Datensätzen d_j . Die möglichen Attributwerte $x \in \{0, 1, \dots, N\}$ können durchaus mehrfach vorkommen (d.h. $x_i = x_j$ für $i \neq j$ ist zugelassen). Die im Folgenden betrachteten Verfahren benutzen die Kenntnis der Zahl N und beruhen auf dem auf folgender Folie angedeuteten naiven Sortievorgang:

Naives Sortieren:

Sortiere die Zahlenfolge 4 4 1 0 1 4

1. Schritt: Zähle die Anzahl der Vorkommen jeder Zahl $\in \{0, 1, \dots, 4\}$

0	1	2	3	4
I	II			III

2. Schritt: Gib die sortierte Zahlenfolge aus.

$1 \times 0 \rightsquigarrow 0$
$II \times 1 \rightsquigarrow 1 1$
0×2
0×3
$III \times 4 \rightsquigarrow 4 4 4$

Nachteil: Nicht anwendbar für komplexe Datensätze (mit Schlüsselwerten)

2.2.1 Countingsort

Die Grundidee von Countingsort ist die explizite Berechnung der Positionen k_j , an der der j -te Datensatz d_j (mit Attributwert x_j) in einer Sortierung steht. Dazu wird folgende Formel verwendet: k_j ist die Anzahl an Indizes $i \in \{1, \dots, n\}$, so daß entweder $x_i < x_j$ oder $x_i = x_j$ und $i \leq j$. Diese Werte k_j werden berechnet, indem man zunächst die Werte

$$\text{Counter}(\ell) = |\{i \in \{1, \dots, n\} : x_i \leq \ell\}|, \quad \ell = 0, 1, \dots, N$$

in Zeit $\mathcal{O}(n + N)$ bestimmt. Dies ist möglich aufgrund der Beziehung

$$\text{Counter}(\ell) = \text{Counter}(\ell - 1) + |\{i : x_i = \ell\}|, \quad \ell = 1, \dots, N.$$

Liegen die Werte $\text{Counter}(\ell)$ vor, so ergeben sich die Werte k_j durch

$$k_j = \text{Counter}(x_j) - |\{i > j : x_i = x_j\}|.$$

Abbildung 6 auf Seite 48 zeigt ein Beispiel. Eine entsprechende Formulierung dieser Ideen ist in Algorithmus 15 angegeben, welcher die Sortierung der Schlüsselwerte in y_1, \dots, y_n ablegt. Wie zuvor abstrahieren wir hier von den eigentlichen Datensätzen und stellen diese nur durch ihre Schlüsselwerte dar. Die Zuweisung „ $y_k := x_j$ “ in der dritten FOR-Schleife ist so zu lesen, daß der Datensatz d_j mit Schlüsselwert x_j an die k -te Stelle der sortierten Ausgabefolge gesetzt wird. Offenbar hat Countingsort die Laufzeit $\Theta(n + N)$, im schlechtesten und besten Fall. Der zusätzliche Platzbedarf ist $\Theta(N)$ für die Counter und $\Theta(n)$ für die y -Folge, wobei je nach Anwendung auf die explizite Darstellung der y -Folge möglicherweise verzichtet werden kann.

Algorithmus 15 Countingsort für Datensätze mit Schlüsselwerten $x_j \in \{0, 1, \dots, N\}$

FOR $\ell = 0, 1, \dots, N$ DO

$Counter(\ell) := 0$

OD

FOR $i = 1, \dots, n$ DO

$Counter(x_i) := Counter(x_i) + 1$

OD

(* nun ist $Counter(\ell) = |\{i : x_i = \ell\}|$ *)

FOR $\ell = 1, \dots, N$ DO

$Counter(\ell) := Counter(\ell) + Counter(\ell - 1)$

OD

(* nun ist $Counter(\ell) = |\{i : x_i \leq \ell\}|$ *)

FOR $j = n, n-1, \dots, 1$ DO

$k := Counter(x_j);$ (* $k = k_j = |\{i : x_i < \ell\}| + |\{i \leq j : x_i = x_j\}|$ *)

$y_k := x_j;$

$Counter(x_j) := k - 1$

OD

(* die sortierte Zahlenfolge ist $y_1, \dots, y_n.$ *)

Counting-Sort		(stabile Version)														
Sortiere die Zahlenfolge		4 4 1 0 1 4														
<u>1. Schritt:</u> Zähle die Anzahl der Vorkommen jeder Zahl $\in \{0, 1, 2, 3, 4\}$																
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr> <td>I</td><td>II</td><td></td><td></td><td>III</td></tr> </table>			0	1	2	3	4	I	II			III				
0	1	2	3	4												
I	II			III												
<u>2. Schritt:</u> Ermittle die Anzahl an Folgenelementen $\leq \ell$, $\ell \in \{0, 1, \dots, 4\}$																
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr> <td>X</td><td>3</td><td>3</td><td>6</td><td>54</td></tr> </table>			0	1	2	3	4	X	3	3	6	54				
0	1	2	3	4												
X	3	3	6	54												
<u>3. Schritt:</u> Plazieren die Elemente in sortierter Reihenfolge																
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>Nr.</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> <tr> <td></td><td>0</td><td>1</td><td>1</td><td>4</td><td>4</td><td>4</td></tr> </table>			Nr.	1	2	3	4	5	6		0	1	1	4	4	4
Nr.	1	2	3	4	5	6										
	0	1	1	4	4	4										
(von rechts nach links)																

Abbildung 6: Beispiel zu Countingsort

2.2.2 Bucketsort

Die Grundidee von Bucketsort ist die Verwendung von $N + 1$ Buckets B_0, \dots, B_N , in die sukzessive die Datensätze mit den Schlüsselwerten x_j eingefügt werden, wobei der Datensatz mit Schlüsselwert x_j genau dann in Bucket B_k eingefügt wird, wenn $k = x_j$. Anschließend werden die Buckets B_0, \dots, B_N der Reihe nach „entleert“. Dies ist in Abbildung 7 illustriert.

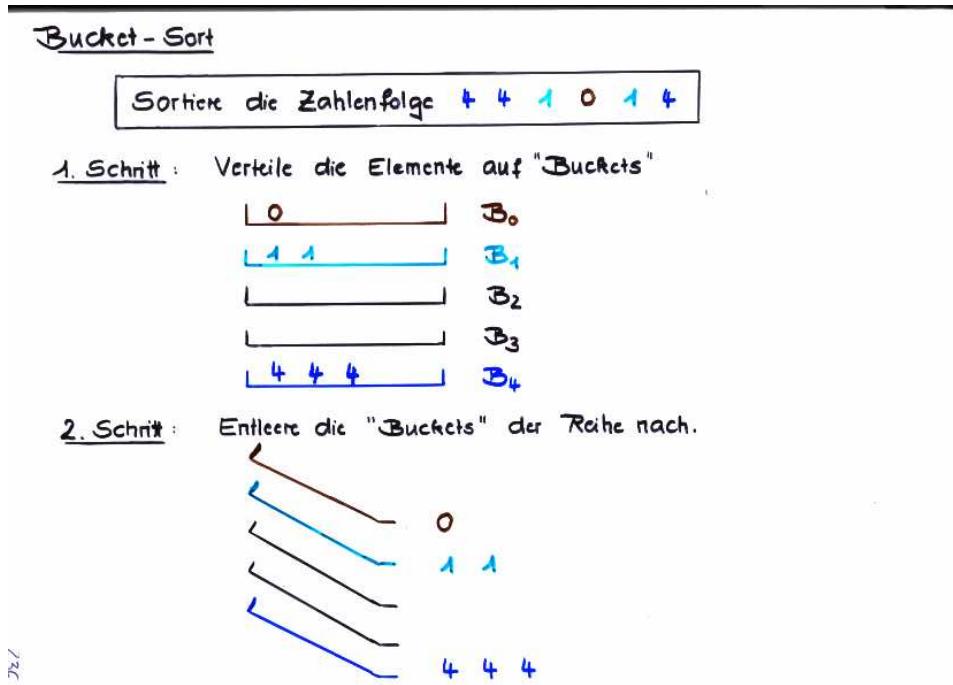


Abbildung 7: Zur Idee von Bucketsort

Algorithmus 16 zeigt eine Formulierung des Verfahrens, wobei wir von den eigentlichen Datensätzen abstrahieren und nur auf die Schlüsselwerte x_j Bezug nehmen.

Die Laufzeit ist offenbar $\Theta(n+N)$. Der zusätzliche Platzbedarf ist $\Theta(N)$ für die Listenköpfe der Buckets und $\Theta(n)$ für die Buckets selbst, insgesamt also $\Theta(n + N)$.

Wir diskutieren nun eine Variante von Bucketsort (die in der Literatur ebenfalls unter dem Stichwort Bucketsort bekannt ist), welche Datensätze mit reellen Schlüsselwerten x_1, \dots, x_n im Intervall $[0, 1]$ sortiert. Hierzu verwenden wir n Buckets B_0, \dots, B_{n-1} , wobei B_k alle Datensätze mit Schlüsselwerten $x_j \in [\frac{k}{n}, \frac{k+1}{n}]$ erfassen soll. Siehe Algorithmus 17. Um das sukzessive Einfügen der x_j 's in die B_k 's zu erleichtern, bietet sich eine Listenstruktur der B_k 's (anstelle einer Array-Darstellung) an. Dies hat jedoch zur Folge, daß im Sortierschritt der einzelnen Listen nur schwerlich auf die typischen Array-Verfahren wie Quicksort zugegriffen werden kann. Stattdessen wird man auf Sortierverfahren ausweichen, welche leicht mit verketteten Listen realisiert werden können (z.B. Sortieren durch Einfügen, Bubblesort oder Mergesort).

Wir analysieren nun die durchschnittlichen Kosten von Bucketsort, unter der Annahme, daß ein Sortierverfahren quadratischer Laufzeit für die einzelnen Buckets eingesetzt wird

Algorithmus 16 Bucket sort (für Schlüsselwerte $x_1, \dots, x_n \in \{0, 1, \dots, N\}$)

(* Verwende $N + 1$ Buckets B_0, \dots, B_N *)

FOR $k = 0, 1, \dots, N$ **DO**

$B_k := \emptyset$ (* initialisiere B_k mit der leeren Liste *)

OD

FOR $j = 1, \dots, n$ **DO**

$k := x_j$;
füge x_j in B_k ein

OD

FOR $k = 0, 1, \dots, N$ **DO**

schreibe die Elemente von B_k in die Ergebnisfolge

OD

Algorithmus 17 Bucket sort (für gleichverteilte reelle Schlüsselwerte $x_1, \dots, x_n \in [0, 1[$)

(* Verwende n Buckets B_0, \dots, B_{n-1} *)

FOR $k = 0, 1, \dots, n - 1$ **DO**

$B_k := \emptyset$ (* initialisiere B_k mit der leeren Liste *)

OD

FOR $j = 1, \dots, n$ **DO**

$k := \lfloor n * x_j \rfloor$; (* $\frac{k}{n} \leq x_j < \frac{k+1}{n}$ *)
füge x_j in die Liste B_k ein

OD

FOR $k = 0, 1, \dots, n - 1$ **DO**

sortiere die Liste B_k mit einem Standardverfahren.

OD

FOR $k = 0, 1, \dots, n - 1$ **DO**

schreibe die Elemente von B_k in die Ergebnisfolge

OD

und daß eine Gleichverteilung der Schlüsselwerte vorliegt, also daß

$$\text{Prob} (x_j \text{ liegt im Intervall } [\frac{k}{n}, \frac{k+1}{n}]) = \frac{1}{n}, \quad k = 0, 1, \dots, n-1.$$

Im Folgenden sei $k \in \{0, 1, \dots, n-1\}$ fest. Zunächst überlegen wir uns, daß die erwartete Länge der Liste B_k gleich 1 ist. Intuitiv sollte das klar sein, da wir n Elemente auf n Listen verteilen und dabei mit jeweils derselben Wahrscheinlichkeit jede der einzelnen Listen auswählen.

Wir wollen dennoch das formale Argument für die erwartete Listenlänge angeben. Hierzu betrachten wir die zufälligen Größen $f_{1,k}, \dots, f_{n,k}$, wobei $f_{j,k}$ den Wert 1 annimmt, falls x_j in B_k eingefügt wird, andernfalls hat $f_{j,k}$ den Wert 0. Offenbar ist die Listenlänge von B_k (aufgefaßt als Zufallsvariable) gleich der Summe der $f_{j,k}$'s:¹⁵

$$\text{Länge von } B_k = f_{1,k} + \dots + f_{n,k}.$$

Da wir Gleichverteilung annehmen, gilt $\text{Prob}(f_{j,k} = 1) = \frac{1}{n}$, und somit

$$\mathbb{E}[f_{j,k}] = \text{Prob}(f_{j,k} = 1) \cdot 1 + \text{Prob}(f_{j,k} = 0) \cdot 0 = \frac{1}{n} + 0 = \frac{1}{n}.$$

Wir erhalten:

$$\mathbb{E}[\text{Länge von } B_k] = \mathbb{E}[f_{1,k}] + \dots + \mathbb{E}[f_{n,k}] = n \cdot \frac{1}{n} = 1.$$

Hier benutzen wir die Linearität des Erwartungswertoperators, d.h., die Tatsache, daß der Erwartungswert der Summe von Zufallsvariablen gleich der Summe der „individuellen“ Erwartungswerte der betreffenden Zufallsvariablen ist.

Erwartete Listenlänge 1 ist ein erstes Indiz für die Effizienz von Algorithmus 17, da wir nun im Sortierschritt mit Listen der Länge 1 rechnen können, so daß der quadratische Sortieraufwand unerheblich ist. Auch hier wollen wir uns die präzise Argumentation überlegen. Der erwartete Gesamtaufwand für den Sortievorgang für Liste B_k errechnet sich wie folgt:

$$\begin{aligned} \mathbb{E}[(\text{Länge von } B_k)^2] &= \mathbb{E}[(f_{1,k} + \dots + f_{n,k})^2] \\ &= \mathbb{E}\left[\sum_{1 \leq i, j \leq n} f_{i,k} \cdot f_{j,k}\right] \\ &= \sum_{1 \leq i, j \leq n} \mathbb{E}[f_{i,k} \cdot f_{j,k}] \\ &= \sum_{\substack{1 \leq i, j \leq n \\ i \neq j}} \mathbb{E}[f_{i,k} \cdot f_{j,k}] + \sum_{1 \leq i \leq n} \mathbb{E}[f_{i,k}^2] \end{aligned}$$

Da $f_{i,k}$ nur Werte in $\{0, 1\}$ annimmt, ist $f_{i,k}^2 = f_{i,k}$ und somit $\mathbb{E}[f_{i,k}^2] = \mathbb{E}[f_{i,k}] = \frac{1}{n}$. Für $i \neq j$ nimmt $f_{i,k} \cdot f_{j,k}$ genau dann den Wert 1 an, wenn $f_{i,k}$ und $f_{j,k}$ den Wert 1 haben,

¹⁵Es handelt sich um ein Bernoulli-Experiment mit den Parametern n und $p = \frac{1}{n}$.

also genau dann, wenn $x_i \in [\frac{i}{n}, \frac{i+1}{n}[$ und $x_j \in [\frac{j}{n}, \frac{j+1}{n}[$. Andernfalls hat $f_{i,k} \cdot f_{j,k}$ den Wert 0. Daher gilt:

$$\mathbb{E}[f_{i,k} \cdot f_{j,k}] = Prob(f_{i,k} = 1) \cdot Prob(f_{j,k} = 1) = \frac{1}{n^2}$$

Wir setzen diese Werte in obige Formel ein und erhalten:

$$\begin{aligned} \mathbb{E}[(\text{Länge von } B_k)^2] &= \sum_{\substack{1 \leq i, j \leq n \\ i \neq j}} \mathbb{E}[f_{i,k} \cdot f_{j,k}] + \sum_{1 \leq i \leq n} \mathbb{E}[f_{i,k}^2] \\ &= \sum_{\substack{1 \leq i, j \leq n \\ i \neq j}} \frac{1}{n^2} + \sum_{1 \leq i \leq n} \frac{1}{n} \\ &= n \cdot (n - 1) \cdot \frac{1}{n^2} + 1 = 2 - \frac{1}{n} \end{aligned}$$

Aufsummieren über alle k liefert den erwarteten Gesamtaufwand von Algorithmus 17:

$$\mathcal{O}\left(\sum_{0 \leq k \leq n-1} \mathbb{E}[(\text{Länge von } B_k)^2]\right) = \mathcal{O}(2n - 1) = \mathcal{O}(n).$$

2.2.3 Radixsort

Der Ausgangspunkt von Radixsort ist eine Folge d_1, \dots, d_n von Datensätzen mit jeweils k Attributwerten

$$key_j(d_i) = x_{i,j} \in \{0, 1, \dots, N\}, \quad i = 1, \dots, n, \quad j = 1, \dots, k.$$

Wir gehen hier zur Vereinfachung davon aus, daß allen Schlüsselattributen derselbe Wertebereich $\{0, 1, \dots, N\}$ zugrunde liegt. Dies ist z.B. für k -ziffrige Dezimalzahlen

$$d_i = \sum_{j=1}^k 10^{k-j} \cdot x_{i,j}, \quad x_{i,j} \in \{0, 1, \dots, 9\}$$

oder auch für Wörter fester Länge k über einem endlichen Alphabet $\Sigma = \{a_0, a_1, \dots, a_N\}$ der Fall. In analoger Weise können Datensätze sortiert werden, deren Schlüsselattribute verschiedene Wertebereiche haben.

Gesucht ist eine Sortierung $d_{\ell_1}, \dots, d_{\ell_n}$ der Datensätze bzgl. *lexikographischer Ordnung*, unter welcher der erste Schlüssel höchste Priorität hat und der k -te die geringste Priorität. D.h., wir fordern $d_{\ell_1} \sqsubseteq d_{\ell_2} \sqsubseteq \dots \sqsubseteq d_{\ell_n}$, wobei

$$\begin{aligned} d_i \sqsubseteq d_j \text{ genau dann, wenn } (x_{i,1}, \dots, x_{i,k}) &= (x_{j,1}, \dots, x_{j,k}) \text{ oder} \\ (x_{i,1}, \dots, x_{i,k}) &\sqsubset (x_{j,1}, \dots, x_{j,k}) \end{aligned}$$

und wobei die Ordnung \sqsubset auf den k -Tupeln von Schlüsselwerten wie folgt definiert ist.

$$(x_{i,1}, \dots, x_{i,k}) \sqsubset (x_{j,1}, \dots, x_{j,k}) \quad \text{gdw. es existiert ein Index } m \in \{1, \dots, k\} \text{ mit} \\ x_{i,1} = x_{j,1}, \dots, x_{i,m-1} = x_{j,m-1} \text{ und } x_{i,m} < x_{j,m}.$$

Sind die d_i 's Dezimalzahlen und Schlüssel $x_{i,k}$ die k -te Ziffern von d_i , so entspricht \sqsubseteq der natürlichen Ordnung \leq .

Zwar ist Counting- oder Bucketsort anwendbar, um die Datensätze d_1, \dots, d_n hinsichtlich \sqsubseteq zu sortieren, jedoch ergibt sich die Laufzeit $\Theta(n + N^k)$. Radixsort wendet stattdessen k -mal ein Sortierverfahren wie Counting- oder Bucketsort an:

- Sortierung hinsichtlich des k -ten Schlüsselwerts
- Sortierung hinsichtlich des $(k - 1)$ -ten Schlüsselwerts
- ...
- Sortierung hinsichtlich des ersten Schlüsselwerts

Um sicherzustellen, daß die Sortierungen der vorangegangenen Iterationen erhalten bleiben, benötigen wir hier den Begriff der Stabilität. Eine *stabile Sortierung* der durch die Datensätze d_1, \dots, d_n gegebenen Folge x_1, \dots, x_n von Schlüsselwerten $x_i = \text{key}(d_i)$ ist eine Permutation $d_{\ell_1}, \dots, d_{\ell_n}$ von d_1, \dots, d_n , so daß¹⁶

1. $x_{\ell_1} \leq \dots \leq x_{\ell_n}$
2. Aus $x_{\ell_i} = x_{\ell_j}$ folgt: $\ell_i < \ell_j$ gdw. $i < j$.

Man spricht von einem stabilen Sortierverfahren, falls die berechnete Sortierung für jede Eingabefolge stabil ist. Beispielsweise sind die angegebenen Formulierungen von Counting- oder Bucketsort stabile Sortierverfahren. Durch den Einsatz von Counting- oder Bucketsort in den einzelnen Sortierphasen wird offenbar die Laufzeit $\Theta(k \cdot (n + N))$ erreicht. Der zusätzliche Platzbedarf ist $\Theta(n + N)$.

Algorithmus 18 Radixsort für n Datensätze mit jeweils k Schlüsselwerten in $\{0, 1, \dots, N\}$

FOR $r = k, k - 1, \dots, 1$ **DO**

sortiere die Datensätze hinsichtlich des r -ten Schlüsselwerts mit einem *stabilen* Sortierverfahren (z.B. Counting- oder Bucketsort)

OD

Bemerkung 2.2.1. Die Stabilität des eingesetzten Sortierverfahrens ist eine wesentliche Voraussetzung in Algorithmus 18. Dies kann man sich am Beispiel der Eingabefolge bestehend aus den drei zweistelligen Dezimalzahlen 22,74,21 (also $k = 2$ und $N = 9$) und klarmachen:

¹⁶Im Kontext von Radixsort ist $x_i = \text{key}_r(d_i)$ für ein $r \in \{1, \dots, k\}$.

$$\begin{array}{ccccc}
 2 & 2 & \text{Sortierung nach} & 2 & 1 \\
 7 & 4 & \xrightarrow{\hspace{2cm}} & 2 & 2 \\
 2 & 1 & \text{zweiter Ziffer} & 7 & 4 & \xrightarrow{\hspace{2cm}} & 2 & 1 \\
 & & & & & & & 7 & 4
 \end{array}$$

Am Beispiel der Eingabefolge 22, 74, 26 wird ersichtlich, weshalb die Behandlung der Schlüsselwerte mit dem k -ten Schlüsselwert beginnend essentiell ist:

$$\begin{array}{ccccc}
 2 & 2 & \text{Sortierung nach} & 2 & 2 \\
 7 & 4 & \xrightarrow{\hspace{2cm}} & 2 & 6 \\
 2 & 6 & \text{erster Ziffer} & 7 & 4 & \xrightarrow{\hspace{2cm}} & 2 & 6
 \end{array}$$

Eine Behandlung der Schlüsselwerte, welche mit dem höchstwertigen (also dem ersten) Schlüsselwert beginnt, ist allerdings möglich, wenn in der i -ten Iteration alle Datensätze, deren $(i-1)$ -te Schlüsselwerte identisch sind, in Gruppen zusammengefasst werden und „gruppenintern“ nach dem i -ten, dem $(i+1)$ -ten, etc. Schlüsselwert sortiert werden. \square

Selbstverständlich spielt Stabilität nicht nur für Sortieren mit Vorkenntnissen eine Rolle. Tatsächlich können einige Array-Verfahren mit Vergleichen (z.B. Quicksort sowie alle anderen in Abschnitt 2.1 behandelten Sortierverfahren) so formuliert werden, daß Stabilität gewährleistet wird. Stabilität ist vor allem dann ein wichtiges Kriterium, wenn die Schlüsselwerte x_i der zu sortierende Zahlenfolge für die Attributwerte einer Folge von Datensätzen stehen, die bezüglich anderen Attributen vorsortiert ist. Liegt beispielsweise eine Personaldatei vor, deren Einträge nach dem Alter der Angestellten sortiert sind, so bleibt die Vorsortierung nach dem Alter durch eine stabile, alphabetische Sortierung erhalten.



2.3 Externes Sortieren

Die bisher betrachteten Sortierverfahren setzen voraus, daß die zu sortierenden Daten im Hauptspeicher untergebracht werden können. Ist die zu sortierende Datenmenge für den Hauptspeicher jedoch zu groß und ist stattdessen auf einem Hintergrundspeicher abgelegt, so ist das relevante Maß für die Laufzeit die *Anzahl der Blocktransporte*, anstelle der Anzahl an Vergleichen, die für die internen Verfahren als dominante Operation angesehen wurden.

Externes Mischen ist eine Variante von Mergesort, mit der sich Daten, die auf einem Hintergrundspeicher abgelegt sind, sortiert werden können. Auf die Daten wird lediglich sequentiell zugegriffen. Die einzigen verwendeten Zugriffsformen sind von der Form „Zurücksetzen und (sequentielles) Lesen“ bzw. „Zurücksetzen und (sequentielles) Schreiben“, die mit jedem gängigen Sekundärspeichermedium realisierbar sind. Die Grundidee ist ein phasenweises Mischen zu immer größer werdenden sortierten Folgen (genannt *Läufe*).

Ausgeglichenes Mischen. Wir erläutern die Vorgehensweise des so genannten ausgeglichenen Mischens am Beispiel der Eingabefolge

$$12, 5, 7, 28, 3, 8, 8, 27$$

und jeweils zwei Ein- und Ausgabebändern. Im ersten Schritt wird die Eingabe abwechselnd auf die beiden Bänder #1 und #2 verteilt, wobei die einzelnen Datensätze als Läufe der Länge 1 angesehen werden.

Band #1	12	7	3	8
Band #2	5	28	8	27
Band #3	—			
Band #4	—			

Im zweiten Schritt werden die Läufe auf Band #1 und #2 zu Läufen der Länge 2 gemischt (wie in der Mischphase von Mergesort) und abwechselnd auf Band #3 und #4 verteilt:

Band #1	—
Band #2	—
Band #3	(5, 12) (3, 8)
Band #4	(7, 28) (8, 27)

Nun werden die Läufe von Band #3 und #4 zu Läufen der Länge 4 gemischt und abwechselnd auf Band #1 und #2 verteilt:

Band #1	(5, 7, 12, 28)
Band #2	(3, 8, 8, 27)
Band #3	—
Band #4	—

Im letzten Schritt erhält man einen sortierten Lauf:

Band #1	—
Band #2	—
Band #3	(3, 5, 7, 8, 8, 12, 27, 28)
Band #4	—

In Algorithmus 19 ist das allgemeine Verfahren skizziert. (Die Kommentare beziehen sich auf den Fall, daß die Anzahl n an zu sortierenden Datensätzen eine Zweierpotenz ist.)

Algorithmus 19 Ausgeglichenes Mischen mit 4 Bändern

(* Initial stehen die Eingabedaten auf Band #3 *)

Verteile die Eingabewerte abwechselnd auf Band #1 und #2.

$(E_1, E_2, A_1, A_2) := (1, 2, 3, 4); \quad (* \text{ in jeder Phase je 2 Ein- und Ausgabebänder *)}$
 $\quad \quad \quad (* \text{ Phasenzähler ist initial } i = 1 *)$

WHILE Band E_2 ist nicht leer **DO**

(* Es stehen je $n/2^i$ Läufe auf Band E_1 und E_2 . Diese haben jeweils die Länge 2^{i-1} . *)

spule die Bänder E_1, E_2 bzw. A_1, A_2 zum sequentiellen Lesen bzw. Schreiben zurück;

mische je zwei Läufe von Band E_1 und Band E_2 und verteile diese abwechselnd auf Band A_1 und A_2 bis das Ende von E_1 und E_2 erreicht ist;

$(E_1, E_2, A_1, A_2) := (A_1, A_2, E_1, E_2) \quad (* \text{ tausche die Rollen der Ein- und Ausgabebänder *)} \quad (* i := i + 1 *)$

OD

(* Sortierte Folge steht auf Band E_1 . *)

Analyse des ausgeglichenen Mischens mit jeweils 2 Ein- und 2 Ausgabebändern.

Sei n die Anzahl der zu sortierenden Datensätze und b die Blockgröße (genauer: die Anzahl an Datensätzen, die in den Hauptspeicher geladen werden).

- Anzahl an Phasen: $\Theta(\log n)$
- Anzahl der Blocktransporte pro Phase: $\Theta\left(\frac{n}{b}\right)$
- Gesamtanzahl an Blocktransporten: $\Theta\left(\frac{n}{b} \cdot \log n\right)$

Der interne Aufwand kann durch $\Theta(n \log n)$ beschränkt werden.

In diesem Stil gibt es eine Reihe weiterer externer Sortierverfahren, welche eine kleinere Gesamtanzahl an Blocktransporten erzielen. Wir verzichten auf weitere Erläuterungen hierzu.

2.4 Selektieren

Maximumssuche. Die einfachste Selektionsaufgabe ist die Bestimmung des Maximums einer Zahlenfolge. Hierzu wird das Array oder die Liste, in dem/der die Eingabefolge x_1, \dots, x_n gespeichert ist, „von links nach rechts“ durchlaufen und das Maximum mit $n - 1$ Vergleichen ermittelt. Offenbar ist $n - 1$ zugleich die beste untere Schranke für eine Maximumssuche (ohne Vorkenntnisse über die Eingabefolge), da pro Vergleich höchstens ein Element als Maximum ausscheiden kann. Selbstverständlich gilt dasselbe für die Frage nach dem Minimum einer Zahlenfolge.

Simultane Maximums- und Minimumssuche. Wir betrachten nun das Problem, das Maximum und das Minimum von einer gegebenen Zahlenfolge x_1, \dots, x_n zu bestimmen. Bestimmt man zuerst das Maximum mit $n - 1$ Vergleichen, dann das Minimum unter den verbliebenen $n - 1$ Elementen mit $n - 2$ Vergleichen, so haben wir insgesamt $2n - 3$ Vergleiche. Es geht jedoch besser, mit insgesamt höchstens $\frac{3}{2}n - 2$ Vergleichen. Hierzu teilen wir die Eingabefolge (gedanklich) in die Folgen x_1, x_3, \dots und x_2, x_4, \dots gebildet aus den ungeraden bzw. ungeraden Indizes auf und führen die Vergleiche $x_{2i-1} < x_{2i}$ durch. Dies sind insgesamt $\frac{n}{2}$ Vergleiche, mit denen wir $\frac{n}{2}$ „Gewinner“ (die jeweils größeren Elemente) und $\frac{n}{2}$ „Verlierer“ (die jeweils kleineren Elemente) bestimmen. Bei Gleichstand entscheidet das Los.¹⁷ Unter den Gewinnern bestimmen wir nun mit $\frac{n}{2} - 1$ Vergleichen das Maximum und unter den Verlierern mit $\frac{n}{2} - 1$ Vergleichen das Minimum. Damit ergeben sich insgesamt $3 \cdot \frac{n}{2} - 2$ Vergleiche.

Mediansuche. Unter dem Median einer Zahlenfolge x_1, \dots, x_n versteht man dasjenige Element x_i , welches in der sortierten Zahlenfolge in der Mitte steht. Für ungerades n ist der Median das Element an Position $\lfloor \frac{n+1}{2} \rfloor$, für gerades n das Element an Position $\frac{n}{2}$. (Für gerades n kann auch das Element an Position $\frac{n}{2} + 1$ als Median definiert werden, das ist Geschmackssache). Unser Ziel ist es nun, einen Algorithmus anzugeben, der als Eingabe eine Zahlenfolge x_1, \dots, x_n in Array-Darstellung hat und den Median ermittelt. Die konzeptionell einfachste Lösung besteht darin, den Umweg über ein Sortierverfahren zu gehen. Jedoch sind damit die Kosten $\Theta(n \log n)$ unvermeidlich, sofern keine Vorkenntnisse über die Werte x_1, \dots, x_n vorliegen. Wir stellen nun ein Linearzeit-Verfahren vor, welches neben der Zahlenfolge x_1, \dots, x_n einen Index $i \in \{1, \dots, n\}$ als Eingabe hat und welches das an i -ter Position der sortierten Folge stehende Element zurückgibt. Den Median erhält man also mit $i = \lfloor \frac{n+1}{2} \rfloor$.

Das Grundschema ist in Algorithmus 20 angegeben. Diesem liegt eine rekursive Formulierung zugrunde, die an die Aufteilung von Quicksort erinnert. Die Aufgabe von

¹⁷Hier sowie im folgenden nehmen wir an, daß n „groß“ ist und schreiben oft Brüche wie $\frac{n}{2}$ statt $\lfloor \frac{n}{2} \rfloor$ oder $\lfloor \frac{n+1}{2} \rfloor$. Selbstverständlich findet in diesem Beispiel für ungerades n ein Element keinen „echten“ Vergleichspartner. Wir vernachlässigen im Folgenden derartige Sonderfälle, die selbstverständlich in einer konkreten Implementierung zu berücksichtigen sind, jedoch keine konzeptionellen Schwierigkeit hervorrufen. Z.B. können wir in diesem Fall das betreffende Element duplizieren und zu den Gewinnern und Verlierern zählen.

$Select(i, \ell, r)$ ist es, in der Zahlenfolge $x_\ell, x_{\ell+1}, \dots, x_{r-1}, x_r$ das i -te Element der sortierten Folge zu ermitteln. Dabei wird $\ell \leq r$ und $1 \leq i \leq r - \ell + 1$ vorausgesetzt.

Algorithmus 20 $Select(i, \ell, r)$

IF $\ell = r$ **THEN**

gib x_ℓ aus

ELSE

bestimme ein Pivot-Element $x \in \{x_\ell, \dots, x_r\}$;

ordne die Folge x_ℓ, \dots, x_r so um, daß alle Elemente $< x$ links von x und alle Elemente $> x$ rechts von x stehen.

sei $k \in \{\ell, \dots, r\}$ die Position von x in der umstrukturierten Folge.

IF $i = k - \ell + 1$ **THEN**

gib x aus

ELSE

IF $i < k - \ell + 1$ **THEN**

$Select(i, \ell, k - 1)$

ELSE

$Select(i - (k - \ell + 1), k + 1, r)$

FI

FI

FI

Randomisierte Version. Wird das Pivot-Element randomisiert gewählt (also $x = x_j$ mit Wahrscheinlichkeit $\frac{1}{r-\ell+1}$ für jedes $j \in \{\ell, \dots, r\}$), so erhalten wir folgende Rekurrenz für die erwartete Laufzeit, gemessen an der Anzahl an benötigten Vergleichen in den Umordnungsphasen:

$$T(n) \leq n - 1 + \frac{1}{n} \cdot \sum_{l=1}^n \max\{T(l-1), T(n-l)\},$$

wobei $n = r - \ell + 1$ die Länge der Eingabefolge ist, von der wir $n \geq 2$ voraussetzen. Weiter ist $T(0) = T(1) = 0$. Die intuitive Erklärung der angegebenen Rekurrenz ist wie für Quicksort. Der Summand $n - 1$ steht für die Anzahl an Vergleichen, die für einen Umordnungsvorgang durchzuführen sind. $\max\{T(l-1), T(n-l)\}$ steht für die Anzahl an Vergleichen, die in den folgenden Rekursionsaufrufen vorgenommen werden.

Zur Vereinfachung setzen wir im Folgenden voraus, daß n gerade ist. Mit einer Indexverschiebung ($j = l - 1$) und Zerlegen der Summe erhalten wir:

$$T(n) \leq n - 1 + \frac{1}{n} \cdot \sum_{j=0}^{\frac{n}{2}-1} \max\{T(j), T(n-j-1)\} + \frac{1}{n} \cdot \sum_{j=\frac{n}{2}}^{n-1} \max\{T(j), T(n-j-1)\}$$

Aufgrund der Monotonie von T ist $\max\{T(j), T(n-j-1)\} = T(J)$, wobei

$$J = \max\{j, n-j-1\} \in \left\{\frac{n}{2}, \dots, n-1\right\}.$$

Dies erlaubt es, die angegebene Rekurrenz (für gerades n) wie folgt umzuformulieren:

$$T(n) \leq n - 1 + \frac{2}{n} \cdot \sum_{J=\frac{n}{2}}^{n-1} T(J)$$

Wir zeigen nun, daß $T(n) = \mathcal{O}(n)$. Hierzu ist eine Konstante $C > 0$ anzugeben, so daß $T(n) \leq Cn$ für hinreichend großes n . Um eine derartige Konstante zu finden (um bzw. deren Existenz nachzuweisen) nehmen wir an, daß $T(J) \leq CJ$ für alle $J < n$ und versuchen hieraus, eine Bedingung für C herzuleiten:

$$\begin{aligned} T(n) &\leq n - 1 + \frac{2}{n} \cdot \sum_{J=\frac{n}{2}}^{n-1} T(J) \\ &\leq n - 1 + \frac{2}{n} \cdot \sum_{J=\frac{n}{2}}^{n-1} CJ \\ &= n - 1 + \frac{2}{n} \cdot C \cdot \left(\frac{n(n-1)}{2} - \frac{\frac{n}{2}(\frac{n}{2}-1)}{2} \right) \\ &= n - 1 + \left(\frac{3}{4}n - \frac{1}{2} \right) \cdot C \end{aligned}$$

Nun versuchen wir C so zu bestimmen, daß

$$n - 1 + \left(\frac{3}{4}n - \frac{1}{2} \right) \cdot C = n \cdot \left(\frac{3}{4}C + 1 \right) - \left(1 + \frac{1}{2}C \right) \leq Cn$$

Tatsächlich stellt man fest, daß dies für $C \geq 4$ gilt, da nämlich:

$$n \cdot \left(\frac{3}{4}C + 1 \right) \leq Cn \quad \text{gdw.} \quad \frac{3}{4}C + 1 \leq C \quad \text{gdw.} \quad 1 \leq \frac{1}{4}C \quad \text{gdw.} \quad 4 \leq C$$

Soweit die Heuristik. Streng genommen ist nun ein formaler Induktionsbeweis zu führen, mit welchem $T(n) \leq 4n$ nachgewiesen wird. (Man beachte, daß hier der Induktionsanfang keine Probleme macht, da $T(0) = T(1) = 0$.)

Die erwarteten Kosten bei einer randomisierten Wahl des Pivotelements stimmen offenbar mit den mittleren Kosten für eine naive deterministische Wahl des Pivot-Elements, etwa stets das erste oder letzte Array-Element, überein, wenn Gleichverteilung vorausgesetzt wird. Man beachte jedoch, daß mit der naiven deterministischen Wahl des Pivot-Elements die Laufzeit im schlimmsten Fall quadratisch ist. Dies resultiert aus der Rekurrenz

$$\begin{aligned} T(n) &= n - 1 + T(n - 1) \\ &= (n - 1) + (n - 2) + T(n - 2) \\ &= (n - 1) + (n - 2) + \dots + 1 = \frac{n \cdot (n - 1)}{2}, \end{aligned}$$

welche man erhält, wenn das Pivot-Element stets das kleinste oder größte Array-Element ist und die Suche in dem jeweils nicht-leeren Array-Abschnitt fortgesetzt wird.

Deterministische Variante. Wir stellen nun eine deterministische Lösung zur Auswahl des Pivot-Elements vor, welche lineare Laufzeit selbst im schlechtesten Fall garantiert. Hierzu verfahren wir wie folgt. Wir beschreiben hier die Wahl des Pivot-Elements für den initialen Aufruf mit $\ell = 1$ und $r = n$. In den folgenden Rekursionsaufrufen sind die Indizes entsprechend zu skalieren.

1. Zunächst bilden wir Fünfergruppen der Eingabefolge und bestimmen deren Mediane.

D.h. wir fassen jeweils fünf aufeinander folgende Array-Elemente zu einer Fünfergruppe zusammen, etwa $(x_1, \dots, x_5), (x_6, \dots, x_{10}), \dots$ für die initiale Eingabefolge. Die Mediane der fünf-elementigen Teilstücke können z.B. durch eine Sortierung mit jeweils zehn Vergleichen bestimmt werden.

2. Sei y_1, \dots, y_m die Folge der im ersten Schritt ermittelten Mediane. Hier ist $m = \frac{n}{5}$ für die initiale Eingabefolge. Wir wenden nun den Selektionsalgorithmus für die y -Folge der Mediane an, um deren Median zu bestimmen. D.h. wir rufen $Select(\frac{n}{10}, 1, \frac{n}{5})$ für die Folge $y_1, \dots, y_{\frac{n}{5}}$ auf.
3. Wir verfahren nun wie in Algorithmus 20. Sei x der in Schritt 2 berechnete Median der y -Folge. Wir sortieren zunächst die Eingabefolge um und „werfen“ alle Elemente $< x$ auf die linke Seite von x , alle Elemente $> x$ auf die rechte Seite von x . Anschließend führen wir einen der beiden Rekursionsaufrufe $Select(i, 1, k - 1)$ oder $Select(i - k, k + 1, n)$ aus, wobei k die Position von x im umsortierten Array ist.

Selbstverständlich ist hier jeweils sicherzustellen, daß die betrachteten Indizes, z.B. $\frac{n}{5}$ oder $\frac{n}{10}$, durch geeignetes Abrunden (etwa $\lceil \frac{n}{5} \rceil$ statt $\frac{n}{5}$) ganzzahlig gemacht werden. Wir vernachlässigen dies auch in der folgenden Kostenanalyse.

Die genannte Vorgehensweise macht selbstverständlich nur für hinreichend großes n (etwa $n \geq n_0 = 20$) Sinn. Für kleines n (also $n < n_0$) wenden wir ein gewöhnliches Sortierverfahren an, um den Median zu bestimmen. Ist eine solche Konstante n_0 festgelegt, so sind die Kosten für $n \leq n_0$ durch eine Konstante C_0 nach oben beschränkt.

Für die nun folgende Kostenanalyse ist der Wert C_0 unerheblich; wir können $T(n) = 1$ für alle $n < n_0$ annehmen. Die worst-case Laufzeit ergibt sich nun aus der Rekurrenz

$$T(n) \leq 10 \cdot \frac{n}{5} + (n - 1) + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10} \cdot n\right)$$

für $n \geq n_0$. Der erste Summand $10 \cdot \frac{n}{5}$ bezeichnet die Kosten, welche sich durch die Bestimmung der Mediane der 5-er Folgen in Schritt 1 mit jeweils zehn Vergleichen ergibt. Der Summand $n - 1$ steht für die in Schritt 3 durchzuführende Umordnung des Arrays. Hierzu sind $n - 1$ Vergleiche nötig. $T\left(\frac{n}{5}\right)$ steht für den Rekursionsaufruf in Schritt 2 zur Bestimmung des Pivot-Elements x (Median der Mediane), $T\left(\frac{7}{10}n\right)$ für den Rekursionsaufruf in Schritt 3. Im schlimmsten Fall hat diese Folge die Länge $\frac{7}{10} \cdot n$.

Dies folgt aus der Beobachtung, daß mindestens $\frac{3}{10}$ aller Elemente der ursprünglichen Eingabefolge x_1, \dots, x_n kleiner oder gleich x und mindestens $\frac{3}{10}$ aller Elemente x_i größer oder gleich x sind.¹⁸ Dies ist in Abbildung 8 angedeutet. Diese ist so zu interpretieren, daß *gedanklich* die Fünfergruppen – jeweils dargestellt durch die Spalten – aufsteigend nach ihren Medianen sortiert sind (in der Skizze von links nach rechts) und die Elemente jeder Fünfergruppe von unten nach oben aufsteigend sortiert sind. Diese Sortierung wird *nicht* im Algorithmus vorgenommen, sie dient lediglich der Anschauung. Die Anzahl an Elementen in den beiden Kästchen beträgt jeweils mindestens

$$\frac{3}{5} \cdot \frac{n-1}{2} + 2 = \frac{3}{10}n - \frac{3}{10} + 2 > \frac{3}{10}n - 1.$$

Zusammen mit dem Median der Mediane (x) liegen also mehr als $\frac{3}{10}n$ Elemente vor, die $\leq x$ bzw. $\geq x$ sind, und die somit nach der Umordnungsphase **nicht** in dem Teil-Array liegen, auf welches die Rekursion angesetzt wird. Unabhängig davon, wie wir verzweigen, liegen also weniger als

$$1 - \frac{3}{10}n = \frac{7}{10}n$$

Elemente in dem relevanten Bereich.

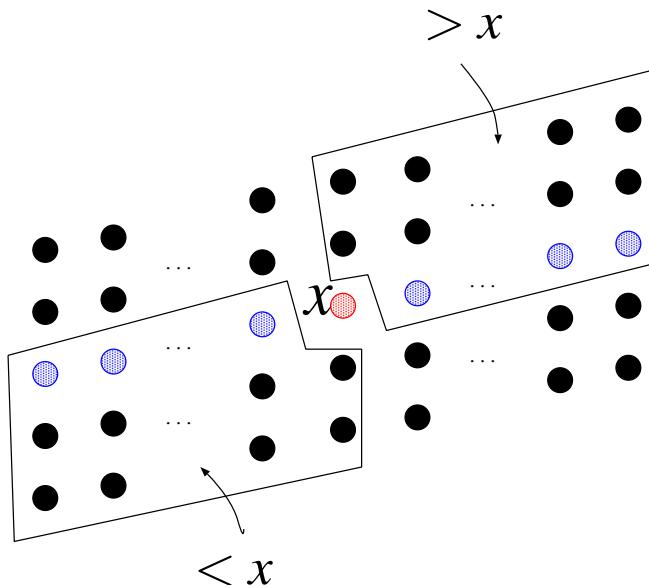


Abbildung 8: mindestens $\frac{3}{10}n$ Elemente sind $\geq x$ bzw. $\leq x$

Wie zuvor kann man nun wieder $T(n) = \mathcal{O}(n)$ gezeigt werden. Hierzu ist eine Konstante C anzugeben, so daß $10 \cdot \frac{n}{5} + (n-1) + T(\frac{n}{5}) + T(\frac{7}{10}n) \leq Cn$ für hinreichend großes n .

¹⁸Die Formulierung „kleiner“ bzw. „größer“ bezieht sich auf den Fall, daß x_1, \dots, x_n paarweise verschieden sind. Im allgemeinen Fall bezieht sich der Begriff „kleiner“ auf eine stabile Sortierung. Wenn also $x_i = x_j$, dann wird x_i als „kleiner“ als x_j angesehen, sofern $i < j$ ist. Dies setzt selbstverständlich eine „stabile“ Umordnung voraus.

Die Konstante C lässt sich heuristisch ermitteln, indem man annimmt, daß $T(\frac{n}{5}) \leq C\frac{n}{5}$ und $T(\frac{7}{10}n) \leq C\frac{7}{10}n$.

Dies liefert:

$$T(n) \leq \underbrace{10 \cdot \frac{n}{5} + (n-1)}_{=3n-1} + \underbrace{T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right)}_{\leq C\frac{9}{10}n} \leq 3n - 1 + C\frac{9}{10}n$$

Wir suchen nun einen Wert für C anhand der Ungleichung

$$3n - 1 + C\frac{9}{10}n \stackrel{!}{\leq} C \cdot n$$

Offenbar muss dann $(3 - \frac{1}{10}C)n \leq 1$ gelten, was z.B. für $C = 30$ erfüllt ist. Durch Induktion nach n kann dann gezeigt werden, daß $T(n) \leq 30n$ für alle n .

Anwendung in Quicksort. Die vorgestellten Linearzeit-Algorithmen zur Mediansuche können in Quicksort zur Bestimmung des Pivot-Elements eingesetzt werden. Es ergibt sich dann die worst-case Zeitkomplexität $\Theta(n \log n)$, da wir es dann mit der Rekurrenz $T(n) = \Theta(n) + 2T(\lfloor \frac{n}{2} \rfloor)$ zu tun haben. Der Summand $\Theta(n)$ steht für die Kosten der Mediansuche und der Umordnungsphase.

3 Grundlegende Datenstrukturen

In diesem Kapitel fassen wir die wesentlichen Merkmale essentieller Datenstrukturen wie Listen, Graphen, Bäume zusammen.

3.1 Bitvektoren, Arrays und Listen

Wir diskutieren kurz die Vor- und Nachteile einfacher Datenstrukturen wie Bitvektoren, Arrays und einigen Listenvarianten.

Bitvektoren. Die Bitvektor-Darstellung einer Teilmenge P einer endlichen Grundmenge V basiert auf der Beschreibung von P durch die charakteristische Funktion

$$\chi_P : V \rightarrow \{0, 1\}, \quad \chi_P(v) = \begin{cases} 1 & : \text{falls } v \in P \\ 0 & : \text{sonst.} \end{cases}$$

Bit-Vektoren lassen sich in der offensichtlichen Weise mit Booleschen Arrays implementieren. Sie haben den Vorteil, daß das Einfügen und Löschen von Elementen sowie der Test, ob ein Element $v \in V$ zu P gehört, in konstanter Zeit durchgeführt werden können. Der Speicherplatzbedarf ist jedoch stets $\Theta(|V|)$, selbst wenn P nur eine sehr „kleine“ Teilmenge von V ist.

(Sortierte) Array-Darstellungen. Die Elemente der darzustellenden Mengen werden bzgl. einer vorgegebenen Indizierung (evtl. Sortierung) in einem Array abgelegt. Der Vorteil ist, daß Zugriffe auf das i -te Element in konstanter Zeit möglich sind. Sortierte Arrays haben den Vorzug, daß die Suche nach einem Schlüsselwert mit der Binärsuche in logarithmischer Zeit vorgenommen werden, jedoch erfordern Einfügen und Löschen eine Umstrukturierung und somit linearen Aufwand. In unsortierten Arrays benötigt man linearen Aufwand für die Suche, jedoch können Einfügen und Löschen in konstanter Zeit realisiert werden, sofern Duplikate zugelassen sind¹⁹ und das zu lösche Element bereits lokalisiert wurde. Allgemein sind Array-Darstellungen wenig geeignet für komplexe Mengenoperationen, wie Mengenvereinigung oder Durchschnitt.

(Lineare) Listen. Die darzustellende Menge wird durch eine Folge x_1, \dots, x_n repräsentiert, in der die Elemente aufgezählt werden.²⁰ Die Folge ist dabei durch das erste Folgenglied und die partielle Nachfolgerfunktion

$$Next(x_i) = \begin{cases} x_{i+1} & \text{if } i = 1, \dots, n-1, \\ \perp & \text{if } i = n \end{cases}$$

¹⁹Werden keine Duplikate zugelassen, so erfordert das Einfügen linearen Aufwand, um zu prüfen, ob das betreffende Element bereits in dem Array vorkommt.

²⁰Für eine Listendarstellung, in der jedes Element der Menge nur einmal aufgeführt wird, kann man annehmen, daß x_1, \dots, x_n die paarweise verschiedenen Elemente der Menge sind. In manchen Fällen kann es sinnvoll sein, auch Duplikate von Elementen der Menge in einer Listendarstellung zuzulassen.

gegeben. Typische Listenoperationen sind Einfügen, Löschen, Suchen eines Schlüsselwerts, Zugriffe auf das erste oder letzte Listenelement, Vereinigung von Listen, das Kopieren von Listen, etc.

Anstelle des ersten Folgenglieds wird häufig ein sogenannter *Listenkopf* verwendet, der als Pseudoelement betrachtet werden kann und etwaige Zusatzinformation über die dargestellte Menge enthalten kann. Häufig wird zusätzlich die partielle Vorgängerfunktion

$$Prev(x_{i+1}) = \begin{cases} x_i & \text{if } i = 1, \dots, n-1 \\ \perp & \text{if } i = 0 \end{cases}$$

dargestellt. Die daraus resultierende doppelte Verkettung vereinfacht einige Operationen. Der Preis hierfür ist ein zusätzlicher Platzbedarf.

Die Verkettung der Listenglieder kann mit Zeigern oder Arrays vorgenommen werden.
Die Implementierung mit Arrays

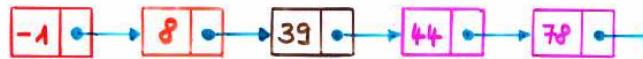
- einem Array $Value[1..N]$, in dem die Elemente x_1, \dots, x_n abgelegt werden,
- einem Array $Next[1..N]$, mit dem die partielle Nachfolgerfunktion dargestellt wird,
- eventuell einem Array $Prev[1..N]$, mit dem die partielle Vorgängerfunktion dargestellt wird,
- einer Variablen $head$, die den Listenkopf repräsentiert und für die Position des ersten Listenglieds steht,²¹

hat den Nachteil, daß eine obere Schranke N für die Listenlänge festgelegt werden muß. Ein wichtiger Vorteil der Array-Implementierung ist, daß die Liste extern gespeichert werden kann.

²¹Für die leere Liste ist ein geeigneter Wert, etwa $head = \perp$, zu verwenden. Soll der Listenkopf mit zusätzlicher Information versehen werden, so kann z.B. ein Record mit diversen Komponenten verwendet werden.

Beispiel für die Listen-Darstellung mit Arrays

Menge $M = \{-1, 8, 39, 44, 78\}$ in Listenform (einfach verkettet)



Mögliche Array-Darstellung							head = 2
Array-Index i	1	2	3	4	5	6	7
Value[i]	39	-1	78	8			44
Next[i]	7	4	1				3

Durchlaufen der Liste (+ Ausgabe der Listenelemente)

```
i := head;
WHILE i ≠ ⊥ DO print(Value[i]); i := Next[i] OD
```

Man beachte, daß bei einer Array-Darstellung der Verkettung auch die freien Listenplätze explizit zu verwalten sind, sofern eine Menge dargestellt wird, die sich durch Einfüge- und Löschoperationen dynamisch verändert. Hierzu können die freien Plätze ebenfalls in einer Liste in Array-Darstellung verwaltet werden, jedoch können die Positionen der Nachfolger in dem bereits vorhandenen Array *Next* abgelegt werden, so daß kein zusätzlicher Platz beansprucht wird (abgesehen von der Variablen *firstfree*, welche die Position des ersten freien Platzes angibt).

3.1.1 Dünnbesetzte Matrizen (sparse matrices)

Die Listendarstellung von $n \times m$ -Matrizen kann durch ein Array $A[1..n]$, dessen i -te Komponente $A[i]$ der Listenkopf einer Liste ist, in welcher die von Null verschiedenen Matrixeinträge der i -ten Zeile verkettet werden. Wir nennen diese Darstellung Variante 1.

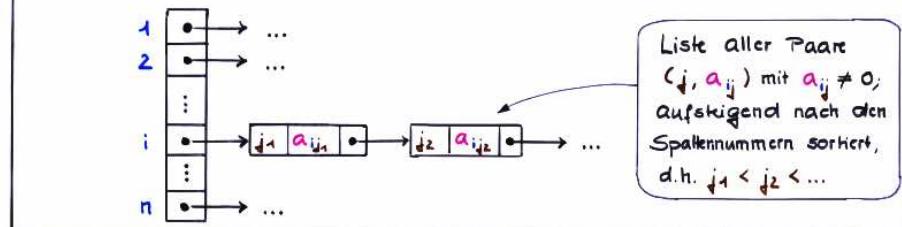
Dünnbesetzte Matrizen:

(Variante 1)

Matrizen mit "vielen" Nullen werden häufig in **Listenform** dargestellt.

Repräsentation einer $(n \times m)$ -Matrix durch ein Feld $A[1..n]$ von Listen

$A[i]$ \triangleq Darstellung der Einträge $\neq 0$ in Zeile i



Beispiel:

$$(3 \times 4)\text{-Matrix } \begin{pmatrix} 0 & 1 & 7 & 0 \\ 0 & 2 & 0 & 11 \\ 0 & 0 & 4 & 0 \end{pmatrix}$$

1	•	→	2	1	•	→	3	7	•	→	1
2	•	→	2	2	•	→	4	11	•	→	2
3	•	→	3	4	•	→	3	4	•	→	3

Der benötigte Speicherplatz ist $\Theta(n + \alpha)$, wobei α die Anzahl an Matrixeinträgen $\neq 0$ ist. Für $\alpha \ll n \cdot m$ (also Matrizen mit sehr vielen Nullen) ist dies vorteilhaft gegenüber der Darstellung durch ein zweidimensionales Array mit $n \cdot m$ Komponenten. Die Addition von zwei $(n \times m)$ -Matrizen

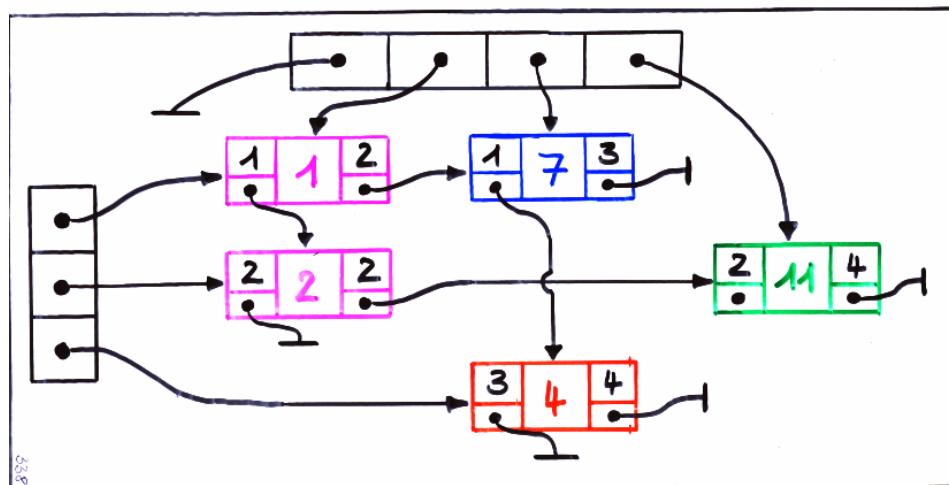
$$(a_{i,j})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq m}} + (b_{i,j})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq m}} = (c_{i,j})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq m}}$$

lässt sich mit dieser Variante durch geeignete Navigation durch die Listen der beiden Matrizen in Zeit $\Theta(n + \alpha + \beta)$ realisieren, wobei α die Anzahl an von Null verschiedenen Einträgen in der ersten Matrix $\mathbf{A} = (a_{i,j})$ und β die Anzahl an von Null verschiedenen Einträgen in der zweiten Matrix $\mathbf{B} = (b_{i,j})$ ist. Man beachte, daß hier der Summand n in der Kostenfunktion nicht weggelassen werden kann, da manche Zeilen aus lauter Nullen bestehen können (dargestellt durch die leere Liste). Die Matrizen-Multiplikation lässt sich mit dieser Darstellungsform jedoch nur schwerlich realisieren, da nun Zeilen der ersten Matrix mit Spalten der zweiten Matrix zu „kombinieren“ sind und Zugriffe auf die Elemente einer Spalte aufwendiges Suchen in den Zeilen-Listen erfordern.

Zur Unterstützung der Matrizenmultiplikation kann zusätzlich ein Array $B[1..m]$ von Listen(-köpfen) verwendet werden, welche die von Null verschiedenen Matrixeinträge spaltenweise verketten. Der Speicherbedarf für Variante 2 ist $\Theta(n + m + \alpha)$, wobei n die Anzahl an Zeilen, m die Spaltenanzahl und α wie oben die Anzahl der von Null verschiedenen Matrixeinträge ist.

Listendarstellung für unbesetzter Matrizen (Variante 2)

z.B. $\begin{pmatrix} 0 & 1 & 7 & 0 \\ 0 & 2 & 0 & 11 \\ 0 & 0 & 4 & 0 \end{pmatrix}$ wird dargestellt durch



(* Sorry. Fehler auf der Folie. Bei dem Listenglied für den Matrixeintrag 4 *)
(* in Zeile 3 und Spalte 3 ist der Spaltenindex falsch. *)

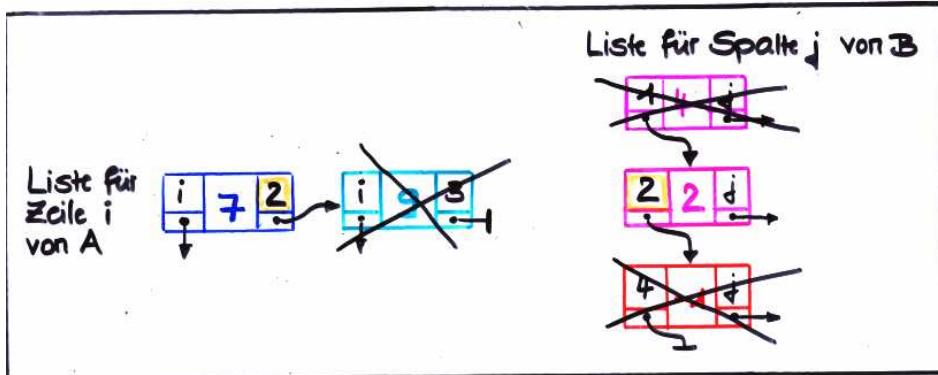
Seien $\mathbf{A} = (a_{i,k})$ eine $(n \times m)$ -Matrix und $\mathbf{B} = (b_{k,j})$ eine $(m \times l)$ -Matrix. Die Grundidee zur Berechnung der Listendarstellung der Produktmatrix $\mathbf{D} = (d_{i,j})$ basiert darauf, die Koeffizienten

$$d_{i,j} = \sum_{k=1}^m a_{i,k} \cdot b_{k,j}$$

durch ein verzahntes Durchlaufen der Listen für Zeile i von \mathbf{A} und Spalte j von \mathbf{B} zu bestimmen. Dabei ist stets zu beachten, daß der Fall $d_{i,j} = 0$ möglich ist, auch wenn es „zueinander passende“ Einträge $a_{i,k}$ und $b_{k,j}$ in den genannten Listen gibt.

Matrizenmultiplikation für dünnbesetzte Matrizen (Variante 2)

$$A = \begin{pmatrix} 0 & \cdots & \\ 0 & 7 & 3 & 0 & 0 \\ \cdots & & & & \end{pmatrix} \quad B = \begin{pmatrix} \cdots & & 4 & \\ & & 2 & \\ \cdots & & 0 & \\ & & -1 & \\ & & 0 & \cdots \end{pmatrix}$$



$$d_{ij} = \sum_{k=1}^5 a_{ik} \cdot b_{kj} = 7 \cdot 2 = 14$$

Für Variante 2 kann die Laufzeit der Matrizenmultiplikation durch $\Theta(n+l+R)$ angegeben werden, wobei

$$\begin{aligned} R &= \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq l} (\alpha_i + \beta_j) \\ &= \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq l} \alpha_i + \sum_{1 \leq j \leq l} \sum_{1 \leq i \leq n} \beta_j \\ &= \sum_{1 \leq i \leq n} l \cdot \alpha_i + \sum_{1 \leq j \leq l} n \cdot \beta_j \\ &= l\alpha + n\beta \end{aligned}$$

Dabei sind $\alpha = |\{(i, k) : a_{i,k} \neq 0\}|$, $\beta = |\{(k, j) : b_{k,j} \neq 0\}|$ wie zuvor und $\alpha_i = |\{k : a_{i,k} \neq 0\}|$ die Anzahl der Einträge $\neq 0$ in Zeile i von A und $\beta_j = |\{k : b_{k,j} \neq 0\}|$ die Anzahl an von Null verschiedenen Einträgen der j -ten Spalte von B . Der Summand $\alpha_i + \beta_j$ steht für die Kosten zur Berechnung des Eintrags $d_{i,j}$ in der Produktmatrix, der Summand $n + l$ für die Initialisierung der Listenköpfe in der Produktmatrix.

3.1.2 Stacks und Queues

Stacks und Queues sind Listen mit eingeschränkten Zugriffsmöglichkeiten. Stacks arbeiten nach dem LIFO-Prinzip (last-in-first-out), während die Zugriffsmöglichkeiten von Queues durch das FIFO-Prinzip (first-in-first-out) gegeben sind:

- Queue (Warteschlange): FIFO-Liste Q mit den Operationen

- $Front(Q)$: liefert das vorderste Queue-Element (lesender Zugriff)
 - $Remove(Q)$: entfernt das vorderste Queue-Element (falls $Q \neq \emptyset$)
 - $Add(Q, x)$: hängt x an das Ende der Queue Q an.
- Stack (Keller, Stapel): LIFO-Liste S mit den Operationen
 - $Top(S)$: liefert das oberste Stack-Element (lesender Zugriff)
 - $Pop(S)$: entfernt das oberste Stack-Element (falls $S \neq \emptyset$)
 - $Push(S, x)$: legt x auf den Stack S .

Zusätzlich ist jeweils der Test, ob der Stack bzw. die Queue leer ist, zulässig. Eine Implementierung ist mit Zeigern oder Arrays möglich. Alle zulässigen Operationen lassen sich so implementieren, daß sie in konstanter Zeit ausführbar sind.

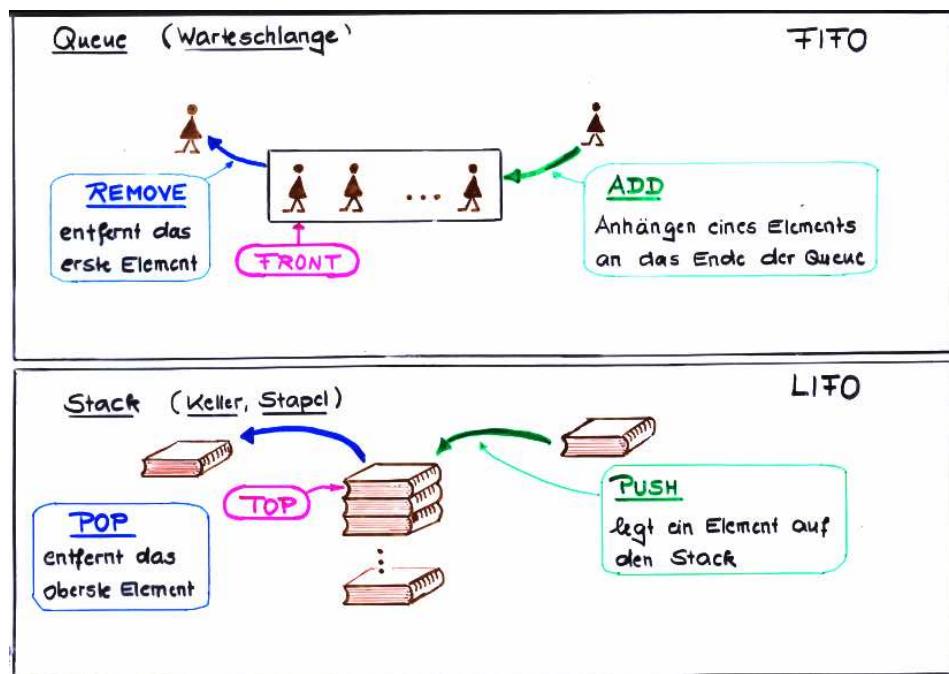


Abbildung 9: Stacks und Queues

Als Anwendungsbeispiel für Stacks betrachten wir die Umwandlung eines arithmetischen Ausdrucks in Infix-Darstellung in einen entsprechenden (klammerlosen) Ausdruck in Postfix-Darstellung. Zur Vereinfachung gehen wir von einem syntaktisch korrekten, vollständig geklammerten Ausdruck A aus, in dem nur Variablen (etwa a, b, c, \dots), aber keine Konstanten vorkommen, und der nur die beiden Operatoren $+$ und $*$ verwendet. Siehe Algorithmus 21.

Algorithmus 21 Transformation eines arithmetischen Infix-Ausdrucks in einen entsprechenden Postfix-Ausdruck

$A :=$ erstes Symbol des gegebenen Infix-Ausdrucks;
 $K := \emptyset;$ (* initialisiere K als leeren Keller *)

REPEAT

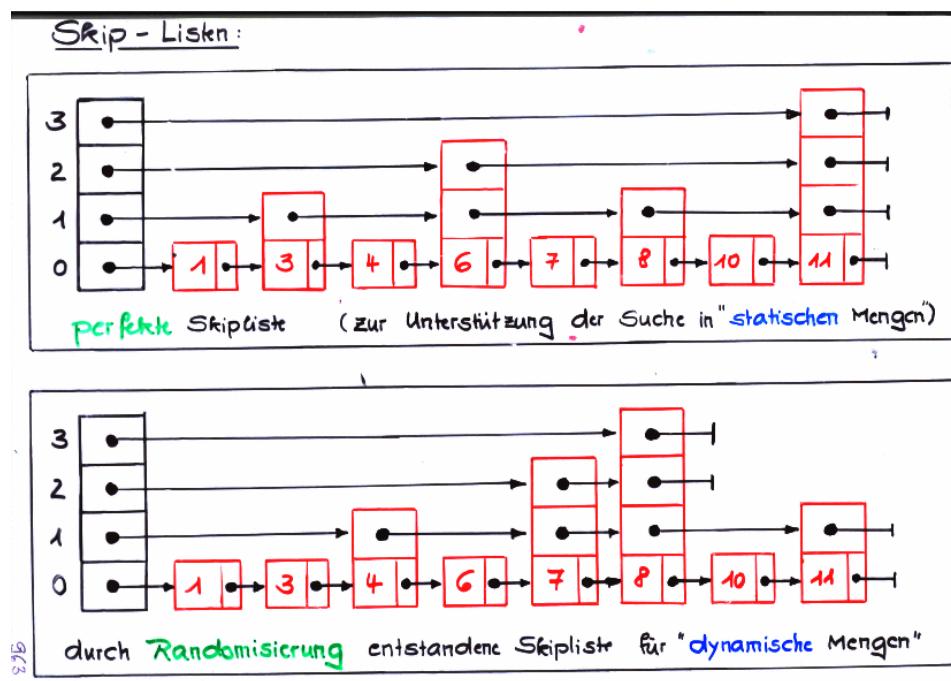
IF A ist eine Variable **THEN**
 gib A aus
ELSE
 IF $A \in \{+, *\}$ **THEN**
 $Push(K, A);$
 ELSE
 IF $A = „,“$ **THEN**
 gib $Top(K)$ aus;
 $Pop(K)$
 FI
 FI
FI
IF der Ausdruck ist noch nicht zu Ende gelesen **THEN**
 $A :=$ nächstes Symbol
FI

UNTIL Ende des Ausdrucks erreicht.

3.1.3 Skiplisten

Skiplisten sind eine Variante von linearen sortierten Listen, welche die Operationen Einfügen, Löschen und Suchen unterstützt. Um die Suchprozedur zu erleichtern werden neben der eigentlichen (wie üblich organisierten) sortierten Liste zusätzliche Verkettungen auf verschiedenen Ebenen verwendet. Die Ebenen sind hierarchisch angeordnet: jedes Listenglied, das in Ebene i vertreten ist, ist auch in den Ebenen $0, 1, \dots, i-1$ vertreten. Die Verkettung auf Ebene 0 entspricht einer gewöhnlichen sortierten Liste. Die Verkettungen der Ebenen $1, 2, \dots$ können als Hilfslisten angesehen werden, in der manche der auf den darunterliegenden Ebenen repräsentierten Elemente miteinander verkettet sind; andere werden „übersprungen“. Unter der *Höhe* eines Listenelements versteht man die Anzahl an Ebenen, in denen das betreffende Listenelement vertreten ist. Im Folgenden sprechen wir von der i -ten Liste (oder Liste i) und meinen damit diejenige Hilfsliste, die sich aus den Verkettungen der i -ten Ebene ergibt.

Die folgende Skizze (oberer Teil) illustriert das Konzept von Skiplisten.



Zur Vereinfachung nehmen wir an, daß $n = 2^k$ Datensätze dargestellt werden sollen. Für $2^{k-1} < n < 2^k$ kann ein spezielles Listenglied verwendet werden, welches das Ende der Skipliste markiert und dieselbe Höhe wie der Listenkopf hat. Alternativ kann man (wie auf der Folie im unteren Teil) auf ein spezielles Listenende verzichten und die Verkettung auf den einzelnen Ebenen „individuell“ beenden. Der unten angegebene Suchalgorithmus setzt jedoch ein für alle Ebenen gemeinsames Listenende voraus.

Perfekte Skiplisten. In perfekten Skiplisten verkettet Liste i genau die Elemente, die an den Positionen $2^i, 2 \cdot 2^i, 3 \cdot 2^i, \dots, 2^{k-i} \cdot 2^i$ stehen, $i = 1, \dots, k$. Die Suche in einer

perfekten Skipliste nach einem Schlüsselwert x kann in Zeit $\mathcal{O}(\log n)$ durchgeführt werden. Sei x der gesuchte Schlüsselwert. Wir starten die Suche mit dem Listenkopf der k -ten Ebene und suchen nun x (bzw. den zu x gehörenden Datensatz), indem wir sukzessive zur Ebene 0 „hinabsteigen“.

Um von Ebene i zu Ebene $i - 1$ zu gelangen, bestimmen wir dasjenige Listenglied v von Liste i , dessen Schlüsselwert $key(v)$ gerade noch kleiner oder gleich x ist, während für das auf der gerade aktuellen Ebene i nachfolgende Listenglied $Next_{i-1}(v)$ der Schlüsselwert $> x$ ist. Ist $key(v) = x$, dann kann die Suche beendet werden. Der zu x gehörende Datensatz ist auf Ebene 0 zu finden. Andernfalls (d.h. wenn $key(v) < x$) gehen wir zur Ebene $i - 1$ über, vorausgesetzt v hat einen Nachfolger und $i \geq 1$. Falls v keinen Nachfolger hat oder $i = 0$, so beenden wir die Suche mit dem Ergebnis, daß es keinen Datensatz mit Schlüsselwert x in der Skipliste gibt.

Man beachte, daß beim Übergang von Ebene i zu Ebene $i - 1$ ein Vergleich ausreichend ist. Lediglich die Schlüsselwerte von v mit dem Nachfolger von v der $(i - 1)$ -ten Ebene müssen verglichen werden. Diese Ideen sind in Algorithmus 125 zusammengefasst. Hier sowie im Folgenden wird das Symbol \perp für „undefiniert“ verwendet. Falls v auf Ebene i das letzte Element ist, so ist also $Next_i(v) = \perp$.

Algorithmus 22 Suche nach Schlüsselwert x in einer perfekten Skipliste der Höhe k

```

i := k;                                     (* wir starten mit der obersten Ebene  $i = k$  *)
v := Kopfelement der Skipliste;          (* Pseudoschlüsselwert  $key(v) = -\infty$  an *)
WHILE  $i \geq 0$  DO                                (* Schleifeninvariante:  $key(v) \leq x$  *)
    w := Nexti(v);                      (* w ist Nachfolger von v in Liste i oder  $\perp$  (undefiniert) *)
    IF  $w \neq \perp$  und  $key(w) \leq x$  THEN
        v := w
    FI                                         (* v ist „letztes“ Listenglied von Ebene i mit  $key(v) \leq x$  *)
    IF  $key(v) = x$  THEN
        gib den in Listenglied  $v$  gespeicherten Datensatz aus und halte an
    ELSE
        IF  $v$  ist letztes Listenelement THEN
            gib „Schlüsselwert  $x$  nicht gefunden“ aus und halte an
        ELSE
            i := i - 1                         (* gehe zur nächst tieferen Ebene *)
        FI
    FI
OD                                         (* Jetzt ist  $i = -1$ . Abstieg von Ebene 0 zu Ebene -1 nicht möglich, also x nicht gefunden. *)
    gib „Schlüsselwert  $x$  nicht vorhanden“ aus.

```

Die durch die Suche verursachten Kosten sind linear in der Höhe der Skipliste; also logarithmisch in der Anzahl n an gespeicherten Datensätzen. Man beachte, daß auf Ebene

i der Vergleich „ $key(v) = x$ “ insgesamt nur einmal durchgeführt wird. Summation über alle Ebenen liefert somit die obere Schranke $\mathcal{O}(\log n)$ für die Anzahl an Vergleichen. Einfügen und Löschen in perfekten Skipisten erfordern jedoch sehr hohen Restrukturierungsaufwand. Perfekte Skipisten sind daher nur für *statische* Datenmengen, für die kaum Einfüge- und Löschoperationen zu erwarten sind, empfehlenswert.

Randomisierte Skipisten. In randomisierten Skipisten verzichtet man auf die Forderung, daß genau die Elemente $j \cdot 2^i$, $j = 1, \dots, 2^{k-i}$ in Liste i dargestellt werden. Stattdessen wird beim Einfügen eines Elements *zufällig* die Höhe des neuen Listenglieds gewählt. Zur Vereinfachung lassen wir beliebige natürliche Zahlen als Werte für die Höhe h zu. In der Praxis wird man sicherlich mit einer Höhenbeschränkung arbeiten. Um der Struktur perfekter Skipisten nahe zu kommen, wählt man eine Zufallsfunktion $random(0, 1, \dots)$, so daß

$$Prob[random(0, 1, \dots) = h] = \left(\frac{1}{2}\right)^{h+1}, \quad h = 0, 1, 2, \dots$$

Eine derartige Zufallsfunktion kann durch das Experiment

- Werfe solange eine Münze bis „Zahl“ oben liegt.
- Gib $h = \text{Anzahl der Münzwürfe minus 1}$ zurück.

simuliert werden.

Erwartete Listenhöhe und erwarteter Platzbedarf. Wir schätzen den Platzbedarf mit der Anzahl an benötigten Zeigern ab. Im Folgenden sei n die Anzahl an Elementen (Schlüsselwerten), die in der Skipiste dargestellt werden sollen also die in Ebene 0 repräsentiert sind. Weiter sei $H(n)$ die Höhe der Skipiste (d.h. die maximale Höhe der Listenglieder) und $Z(n)$ die Anzahl der verwendeten Zeiger. Offenbar ist

$$Z(n) = \sum_{i=1}^n (h_i + 1) + (H(n) + 1),$$

wobei h_i die Höhe des i -ten Listenglieds ist. Der Summand $H(n) + 1$ steht für die Anzahl an Zeigern, mit denen der Kopf der Skipiste ausgestattet ist.

Wir berechnen nun die erwartete Höhe der Listenglieder und der gesamten Skipiste. Die erwartete Höhe jedes Listenglieds (der Erwartungswert der Funktion $random(0, 1, \dots)$) ist

$$\begin{aligned} \mathbb{E}[random(0, 1, \dots)] &= \sum_{h=0}^{\infty} h \cdot Prob[random(0, 1, \dots) = h] \\ &= \sum_{h=0}^{\infty} h \cdot \left(\frac{1}{2}\right)^{h+1} = 1. \end{aligned}$$

Damit gilt:

$$\begin{aligned}
\text{Prob}[H(n) \geq h] &\leq n \cdot \text{Prob}[random(0, 1, \dots) \geq h] \\
&= n \cdot \left(1 - \sum_{k=0}^{h-1} \text{Prob}[random(0, 1, \dots) = k] \right) \\
&= n \cdot \left(1 - \sum_{k=0}^{h-1} \left(\frac{1}{2}\right)^{k+1} \right) \\
&= n \cdot \left(1 - \left(1 - \left(\frac{1}{2}\right)^h\right) \right) = n \cdot \left(\frac{1}{2}\right)^h
\end{aligned}$$

Insbesondere ist:

$$\text{Prob}[H(n) \geq h] \leq \min \left\{ 1, n \cdot \left(\frac{1}{2}\right)^h \right\}$$

Der Erwartungswerte $\mathbb{E}[H(n)]$ (die erwartete Höhe der Skipliste) ergibt sich wie folgt:

$$\begin{aligned}
\mathbb{E}[H(n)] &= \sum_{h \geq 0} h \cdot \text{Prob}[H(n) = h] \\
&= \text{Prob}[H(n) = 1] + \text{Prob}[H(n) = 2] + \text{Prob}[H(n) = 3] + \dots \\
&\quad + \text{Prob}[H(n) = 2] + \text{Prob}[H(n) = 3] + \dots \\
&\quad + \text{Prob}[H(n) = 3] + \dots \\
&\quad \vdots \\
&= \sum_{h \geq 1} \sum_{i \geq h} \text{Prob}[H(n) = i] \\
&= \sum_{h \geq 1} \text{Prob}[H(n) \geq h].
\end{aligned}$$

Aus obiger Abschätzung für die Wahrscheinlichkeiten $\text{Prob}[H(n) \geq h]$ folgt mit $k =$

$\lfloor \log n \rfloor + 1$:

$$\begin{aligned}
\mathbb{E}[H(n)] &= \sum_{h=1}^k \underbrace{\text{Prob}[H(n) \geq h]}_{\leq 1} + \sum_{h \geq k+1} \underbrace{\text{Prob}[H(n) \geq h]}_{\leq n \cdot \frac{1}{2}^h} \\
&\leq k + \underbrace{\sum_{h \geq k+1} n \cdot \left(\frac{1}{2}\right)^h}_{\leq n \cdot \left(\frac{1}{2}\right)^k < n \cdot \frac{1}{n} = 1} \\
&\leq \lfloor \log n \rfloor + 2 = \mathcal{O}(\log n)
\end{aligned}$$

Man beachte, daß $2^k > n$ und somit

$$\sum_{h=k+1} (1/2)^h = (1/2)^{k+1} \sum_{j=0} (1/2)^j = (1/2)^{k+1} \cdot 2 = (1/2)^k = 1/2^k < 1/n.$$

Die erwartete Anzahl $\mathbb{E}[Z(n)]$ an Zeigern ergibt sich wie folgt. Für den Listenkopf werden erwartungsgemäß $\mathbb{E}[H(n)] + 1$ Zeiger benötigt. Für jedes der n Listenglieder ist die erwartete Höhe gleich 1 (siehe oben). Also werden im Mittel zwei Zeiger für jedes Listenglied benötigt (jeweils ein Zeiger für die Ebenen 0 und 1). Damit ergibt sich

$$\mathbb{E}[Z(n)] = 2n + \mathbb{E}[H(n)] + 1 = \mathcal{O}(n).$$

Damit ist gezeigt, daß der erwartete Platzbedarf linear ist.

Für den Suchvorgang nach einem Datensatz mit vorgegebenem Schlüsselwert x in einer randomisierten Skipliste verwenden wir dieselben Ideen wie für perfekte Skiplisten; jedoch benötigen wir eine sequentielle Suche innerhalb der einzelnen Ebenen. Siehe Algorithmus 126.

Kostenanalyse der Suche in randomisierten Skiplisten. Im Folgenden bezeichnet $H(v)$ die Höhe des Listenglieds v . Die einfachste Argumentation für eine Abschätzung der mittleren Kosten des Suchvorgangs ist etwas informell und beruht auf einer Rückverfolgung der unternommenen Suchschritte. Da der Suchvorgang stets von „links oben nach rechts unten“ durch jeweils einen Schritt nach rechts auf derselben Ebene oder mittels eines Schritts nach unten bei demselben Listenelement (Ebenenwechsel) stattfindet, argumentieren wir nun mit *Rückwärtsschritten*, die entweder nach *links* oder *oben* durchgeführt werden.

Es ist klar, daß bei der Suche jedes (besuchte) Listenglieds v über seine höchste Ebene, also Ebene $h = H(v)$, durch einen (Vorwärts-)Schritt von seinem linken Nachbarn u auf Ebene h erreicht wird. Daher müssen bei der Rückverfolgung der Suchschritte bei jedem Listenglied solange Rückwärtsschritte nach oben durchgeführt werden, bis die oberste Ebene erreicht ist.

Algorithmus 23 Suche nach Schlüsselwert x in randomisierter Skipiste der Höhe k

```

i :=  $k$ ;                                (* wir starten mit der obersten Ebene  $i = k$  *)
v := Kopfelement der Skipiste;          (* Pseudoschlüsselwert  $key(v) = -\infty$  an *)
WHILE  $i \geq 0$  DO                    (* Schleifeninvariante:  $key(v) \leq x$  *)
    w :=  $Next_i(v)$ ;                  (* sequentielle Suche in Liste  $i$  gestartet mit  $v$  *)
    REPEAT
        IF  $w \neq \perp$  und  $key(w) \leq x$  THEN
            v := w;
            w :=  $Next_i(w)$ ;
        FI
        UNTIL  $w = \perp$  oder  $key(w) > x$ ;
                (*  $v$  ist „letztes“ Listenglied von Ebene  $i$  mit  $key(v) \leq x$  *)
        IF  $key(v) = x$  THEN
            gib den in Listenglied  $v$  gespeicherten Datensatz aus und halte an
        ELSE
            i :=  $i - 1$                       (* gehe zur nächst tieferen Ebene *)
        FI
    OD
    gib „Schlüsselwert  $x$  nicht vorhanden“ aus.          (* jetzt ist  $i = -1$  *)

```

Nehmen wir nun an, wir sind auf der obersten Ebene $h = H(v)$ von Listenglied v angelangt. Der linke Nachbar u von v auf Ebene h hat also die Höhe mindestens $h = H(v)$. Gemäß Konstruktion (Wahl der Höhen für eingefügte Elemente mittels des Experiments $random(0, 1, \dots)$) hat u mit Wahrscheinlichkeit $1/2$ die Höhe $H(u) = h$ und mit Wahrscheinlichkeit $1/2$ eine Höhe $H(u) > h$. Dasselbe trifft auf jedes Listenglied v zu, bei dem wir uns während des Suchprozesses auf Ebene h mit $h \leq H(v)$ befinden. Mit Wahrscheinlichkeit $1/2$ hat v die Höhe $H(v) = h$ und mit Wahrscheinlichkeit $1/2$ ist $H(v) > h$. Diese Gleichverteilung für die Ereignisse „ $H(v) = h$ “ und „ $H(v) > h$ “ induziert, daß bei der Rückverfolgung der Suchschritte mit derselben Wahrscheinlichkeit (nämlich jeweils $1/2$) ein Rückwärtsschritt nach links oder ein Rückwärtsschritt nach oben durchzuführen ist. Also:

$$\begin{aligned} &\text{mittlere Anzahl an Rückwärtsschritten nach oben} \\ &= \text{mittlere Anzahl an Rückwärtsschritten nach links} \end{aligned}$$

Daraus resultiert:

$$\begin{aligned} & \text{mittlere Gesamtanzahl an Rückwärtsschritten} \\ = & \quad \text{mittlere Anzahl an Rückwärtsschritten nach } oben \\ + & \quad \text{mittlere Anzahl an Rückwärtsschritten nach } links \\ = & \quad 2 * \text{mittlere Anzahl an Rückwärtsschritten nach } oben \\ \leq & \quad 2 \cdot \mathbb{E}[H(n)] = \mathcal{O}(\log n) \end{aligned}$$

Einfügen und Löschen kann mit Hilfe eines geeignet umformulierten Suchalgorithmus ebenfalls im Mittel in logarithmischer Zeit vorgenommen werden. Wir fassen die Ergebnisse zusammen, wobei wir von der Annahme ausgehen, daß die Darstellung der zu einem Schlüsselwert gehörenden Information sowie die Darstellung der Schlüsselwerte selbst lediglich eine konstante Anzahl an Platzeinheiten benötigt:

Satz 3.1.1 (Mittlere Kosten für randomisierte Skipisten). Die erwartete Laufzeit für Suchen, Löschen und Einfügen in randomisierten Skipisten ist $\mathcal{O}(\log n)$. Der erwartete Platzbedarf ist $\mathcal{O}(n)$. Dabei ist n die Anzahl der darzustellenden Schlüsselwerte.

3.2 Graphen und Bäume

Zahlreiche Fragestellungen der Informatik lassen sich mit Graphen bzw. Bäumen formalisieren und mithilfe wohlbekannter Graphalgorithmen lösen. Beispiele für Graphstrukturen sind u.a. Rechnernetze, Flussdiagramme, Syntaxdiagramme, endliche Automaten, Schaltbilder, Petri Netze, u.v.m.

3.2.1 Grundbegriffe

Im Folgenden werden die wichtigsten Grundbegriffe.

Definition 3.2.1 (Graph). Ein Graph ist ein Paar $G = (V, E)$ bestehend aus einer Menge V von *Knoten* und einer Menge E von *Kanten*. Die Kanten stehen für Verbindungen zwischen den Knoten und können gerichtet oder ungerichtet sein.

- Für *gerichtete* Graphen (englisch „directed graph“), im folgenden auch kurz *Digraph* genannt, ist $E \subseteq V \times V$. Ist $e = \langle v, w \rangle \in E$, dann heißt v der Anfangsknoten von e und w der Endknoten von e . Kanten der Form $\langle v, v \rangle$ werden auch *Schleifen* genannt.
- Für *ungerichtete* Graphen ist E eine Menge von ungeordneten Paaren (v, w) mit $v, w \in V$ und $v \neq w$.

Im ungerichteten Fall können die Kanten auch als zweielementige Knotenmengen aufgefasst werden. Üblich ist jedoch die Tupelschreibweise (v, w) anstelle von $\{v, w\}$. \square

Man beachte, daß $(v, w) = (w, v)$, falls (v, w) für eine Kante in einem ungerichteten Graphen steht. Dies gilt jedoch *nicht* für gerichtete Graphen.

In dieser Vorlesung verwenden wir ausschließlich *endliche Graphen*, d.h. Graphen $G = (V, E)$, für welche die Knotenmenge V (und damit auch die Kantenmenge E) endlich ist. Im Folgenden sprechen wir kurz von einem Graphen, wenn wir einen endlichen (gerichteten oder ungerichteten) Graphen meinen. Für den gerichteten Graphen in Abbildung 10 ist die Kantenmenge

$$E_{directed} = \{(v, w), (v, x), (w, x), (x, y), (y, x)\}.$$

Im ungerichteten Fall ist $E_{undirected} = \{\{v, w\}, \{v, x\}, \{w, x\}, \{x, y\}\}$, wobei wir auch hier künftig die Tupelschreibweise (v, w) anstelle von $\{v, w\}$ benutzen.

Die maximale Anzahl an Kanten in gerichteten Graphen ist n^2 , während es im ungerichteten Fall maximal $n(n - 1)/2$ Kanten gibt. Dabei ist jeweils $n = |V|$, die Knotenzahl. Da die Kantenrelation E jedoch sehr viel kleiner als die angegebenen Werte n^2 bzw. $n(n - 1)/2$ sein kann, ist es üblich, die Effizienz von Graphalgorithmen an beiden Parametern $n = |V|$ und $m = |E|$ zu messen und mit zweistelligen Kostenfunktionen $T(n, m)$ zu arbeiten.

Bemerkung 3.2.2 (Alternative Definitionen von Graphen). Die angegebene Art von Kanten in gerichteten Graphen als geordnete Zustandspaare ist für unsere Zwecke

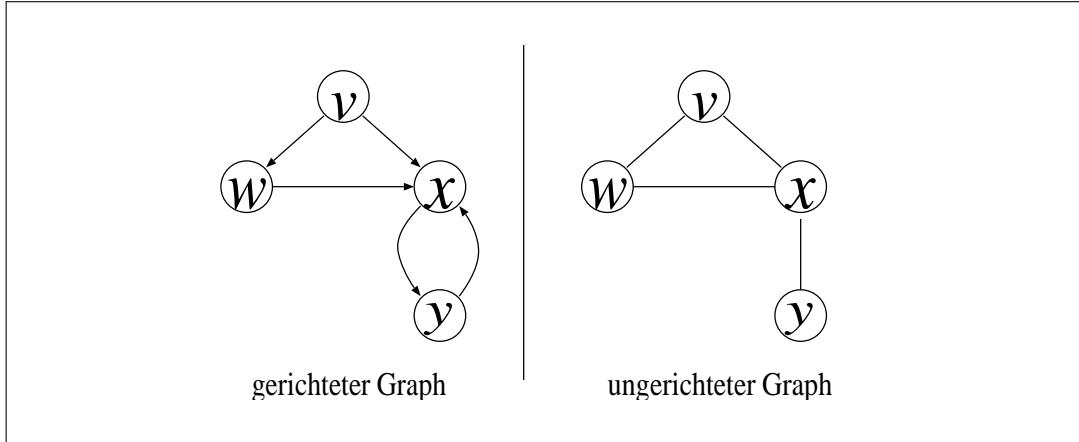


Abbildung 10: Beispiel zu Graphen

ausreichend. Tatsächlich werden aber in vielen Anwendungen auch andere Kantenmengen, etwa $E \subseteq V \times \Sigma \times V$, benötigt, wobei Σ ein Alphabet für die Beschriftung der Kanten ist. Z.B. wird dieser Kantentyp in endlichen Automaten eingesetzt. Kanten $\langle v, a, w \rangle$ und $\langle v, b, w \rangle$ mit demselben Anfangsknoten v , demselben Endknoten w , aber unterschiedlichen Beschriftungen $a, b \in \Sigma$, werden *parallele Kanten* genannt.

In der Literatur werden viele äquivalente Definitionen für ungerichtete Graphen verwendet. Z.B. kann man E auch als symmetrische antireflexive Relation $V \times V$ definieren. Eine (in dieser Vorlesung nicht benötigte) Variante sind ungerichtete Graphen, in denen zusätzlich Schleifen, d.h. Kanten der Form (v, v) , zugelassen sind. Wie für gerichtete Graphen können auch ungerichtete Graphen mit Kantenbeschriftungen versehen werden, die parallele Kanten möglich machen. \square

Definition 3.2.3 (Pfad, Weg). Sei $G = (V, E)$ ein (gerichteter oder ungerichteter) Graph. Ein *Pfad* oder *Weg* in G ist eine Knotenfolge $\pi = v_0, v_1, \dots, v_r$ mit $r \geq 0$ und $(v_i, v_{i+1}) \in E$, $i = 0, 1, \dots, r - 1$. Wir sprechen von einem Pfad, der von v nach w führt, wenn der Anfangsknoten $v_0 = v$ und der Endknoten $v_r = w$ ist. Der Pfad π heißt *einfach*, falls gilt: die Knoten v_0, \dots, v_r paarweise verschieden sind. Die Länge von π ist r , also die Anzahl an Kanten, die in π durchlaufen werden. (Beachte, daß $r = 0$ zugelassen ist.) Diese wird auch mit $length(\pi)$ oder $|\pi|$ bezeichnet. Ein Knoten w heißt von Knoten v *erreichbar* (in G), wenn es einen Pfad $\pi = v_0, v_1, \dots, v_r$ in G gibt, so daß $v_0 = v$ und $v_r = w$. \square

Beispiel 3.2.4. Beispiele für Pfade in den beiden Graphen in Abbildung 10 sind:

$$\pi_1 = v \text{ (Länge 0)}, \pi_2 = v, w, x \text{ (Länge 2)} \text{ und } \pi_3 = v, w, x, y, x, y \text{ (Länge 5)}.$$

Im ungerichteten Fall ist zusätzlich $\pi_4 = v, w, x, v$ ein Pfad. \square

Bezeichnung 3.2.5 (Nachfolgermengen, Vorgängermengen). Wir verwenden folgende Bezeichnungen:

- $Post(v) = \{w \in V : (v, w) \in E\}$ bezeichnet die Menge aller (direkten) Nachfolger von v ,
- $Pre(v) = \{w \in V : (w, v) \in E\} = \{w \in V : v \in Post(w)\}$ die Menge aller (direkten) Vorgänger von v ,
- $Post^*(v) = \{w \in V : w \text{ ist von } v \text{ in } G \text{ erreichbar}\}$ die Menge aller von v erreichbaren Knoten,
- $Pre^*(v) = \{w \in V : v \in Post^*(w)\}$ die Menge aller Knoten, die v erreichen können.

In ungerichteten Graphen werden zwei Knoten v, w , die über eine Kante miteinander verbunden sind (d.h. $(v, w) \in E$), auch *benachbart* genannt. Wir erweitern die Bezeichnungen auf Knotenmengen. Z.B. für $W \subseteq V$ steht $Post(W)$ für $\bigcup_{w \in W} Post(w)$. Analog ist $Pre(W)$ definiert. \square

Beispiel 3.2.6. Für den gerichteten Graphen in Abbildung 10 ist z.B.

$$Post(v) = \{w, x\}, \quad Post^*(v) = \{v, w, x, y\}, \quad Pre(v) = Pre^*(v) = \emptyset,$$

und $Post(x) = \{y\}$, $Post^*(x) = \{x, y\}$. Im ungerichteten Fall ist jedoch $Post(x) = \{y, v, w\}$ und $Post^*(x) = \{v, w, x, y\}$. \square

Für ungerichtete Graphen ist $Post(v) = Pre(v)$ und $Post^*(v) = Pre^*(v)$. Insbesondere gilt im ungerichteten Fall

$$\sum_{v \in V} |Post(v)| = \sum_{v \in V} |Pre(v)| = 2 \cdot |E|$$

Im gerichteten Fall können $Post(v)$ und $Pre(v)$ zwar völlig verschieden sein, jedoch gilt

$$\sum_{v \in V} |Post(v)| = \sum_{v \in V} |Pre(v)| = |E|.$$

Definition 3.2.7 (Zusammenhängend, Zusammenhangskomponente). Sei $G = (V, E)$ ein ungerichteter Graph und $C \subseteq V$. C heißt zusammenhängend, falls es für je zwei Knoten $v, w \in C$ einen Pfad von v nach w gibt, der nur durch Knoten in C läuft, also einen Pfad der Form v_0, v_1, \dots, v_r mit $\{v_0, \dots, v_r\} \subseteq C$ und $v_0 = v, v_r = w$. C heißt Zusammenhangskomponente von G , falls C eine nicht leere maximale zusammenhängende Knotenmenge ist. Maximalität bedeutet dabei, daß C in keiner anderen zusammenhängenden Teilmenge C' von V enthalten ist. G heißt zusammenhängend, falls V zusammenhängend ist. \square

Offenbar sind die Zusammenhangskomponenten eines ungerichteten Graphen G die Äquivalenzklassen der Knotenmenge V unter der Erreichbarkeits-Äquivalenzrelation \equiv , wobei $v \equiv w$ genau dann, wenn $Post^*(v) = Post^*(w)$. Insbesondere zerfällt G in paarweise disjunkte Zusammenhangskomponenten. D.h., sind C_1, \dots, C_r die paarweise verschiedenen Äquivalenzklassen bzgl. \equiv , so ist

$$V = \bigcup_{1 \leq i \leq r} C_i, \quad E = \bigcup_{1 \leq i \leq r} E_i,$$

wobei $E_i = E \cap (C_i \times C_i)$.

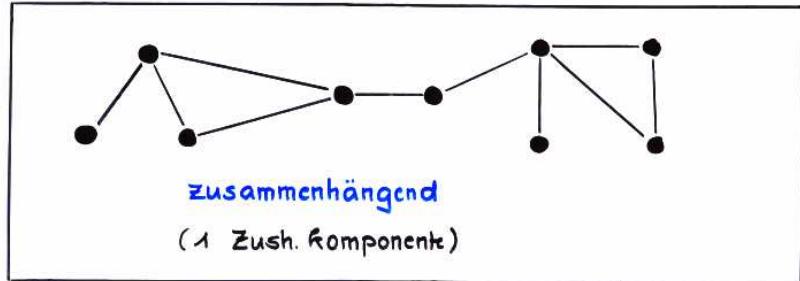
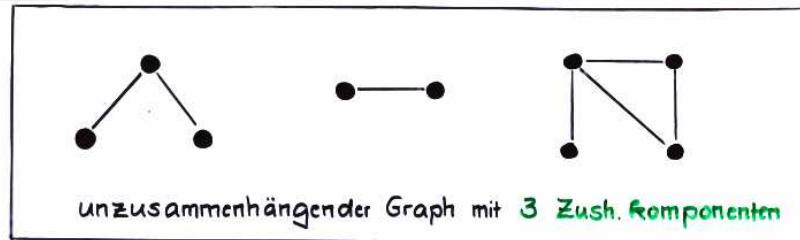


Abbildung 11: Unzusammenhängender und zusammenhängender Graph

Lemma 3.2.8 (Mindestanzahl an Kanten in zusammenhängenden Graphen). Ist $G = (V, E)$ ein ungerichteter Graph mit $n = |V| \geq 1$ Knoten und $m = |E|$ Kanten, so gilt: Ist G zusammenhängend, dann ist $m \geq n - 1$.

Beweis. Wir weisen die Aussage durch Induktion nach n nach. Ist $n = 1$, so hat G keine Kanten, also ist $m = 0 = 1 - 1 = n - 1$. Ebenso ist die Behauptung für $n = 2$ unmittelbar klar, da jeder zusammenhängende Graph mit zwei Knoten genau eine Kante hat. In diesem Fall gilt also $m = 1 = 2 - 1 = n - 1$.

Wir nehmen nun $n \geq 3$ an und setzen voraus, daß G zusammenhängend ist. Zunächst wählen wir einen Knoten, für welchen die Anzahl an ausgehenden Kanten minimal ist. Sei also v ein Knoten, für den $|Post(v)| = k$ minimal ist. Der Fall $k = 0$ ist nicht möglich, da sonst v ein isolierter Knoten wäre (was im Widerspruch zur Annahme steht, daß G zusammenhängend ist). Ist $k \geq 2$, so gilt:

$$2m = 2|E| = \sum_{w \in V} \underbrace{|Post(w)|}_{\geq k} \geq nk \geq 2n$$

und somit $m \geq n > n - 1$. Für $k = 1$ argumentieren wir wie folgt. Sei $G \setminus \{v\}$ derjenige Graph, welcher aus G durch Entfernen von v sowie der ausgehenden Kante von v entsteht. Mit G ist offenbar auch $G \setminus \{v\}$ zusammenhängend. Die Induktionsvoraussetzung auf $G \setminus \{v\}$ angewandt, liefert

$$m - 1 = \text{Anzahl an Kanten in } G \setminus \{v\} \geq \text{Anzahl an Knoten in } G \setminus \{v\} - 1 = n - 2$$

Also ist $m \geq n - 1$. □

Einen analogen Begriff des Zusammenhangs für den gerichteten Fall werden wir später betrachten.

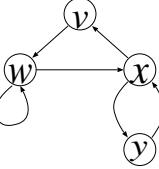
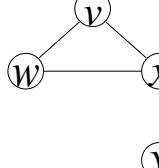
	einfache Zyklen	nicht-einfache Zyklen
	w, w	w, w, w
	v, w, x, v	v, w, x, v, w, x, v
	x, y, x	v, w, x, y, x, v
	Zyklus v, w, x, v (einfach)	keine Zyklen
	x, y, x	v, w, x, w, v

Abbildung 12: Zyklen in gerichteten und ungerichteten Graphen

Definition 3.2.9 (Zyklus bzw. Kreis). Ein *einfacher Zyklus* (oder einfacher Kreis) in einem

- Digraphen ist ein einfacher Pfad $\pi = v_0, v_1, \dots, v_r$ der Länge $r \geq 1$ mit $v_0 = v_r$
- ungerichteten Graphen ist ein einfacher Pfad $\pi = v_0, v_1, \dots, v_r$ mit $v_0 = v_r$ und $r \geq 3$.

Ein *Zyklus* ist ein Pfad v_0, v_1, \dots, v_r , der sich aus einfachen Zyklen zusammensetzt. Formal bedeutet dies:

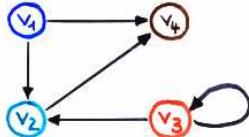
- (1) Jeder einfache Zyklus ist ein Zyklus.
- (2) Sind $\pi_1 = v_0, \dots, v_{i-1}, v_i, v_{i+1}, \dots, v_r$ und $\pi_2 = w_0, w_1, \dots, w_\ell$ Zyklen, wobei $w_0 = w_\ell = v_i$, so ist auch $v_0, \dots, v_{i-1}, w_0, w_1, \dots, w_\ell, v_{i+1}, \dots, v_r$ ein Zyklus.
- (3) Nur die durch (1) und (2) generierbaren Pfade sind Zyklen.

Abbildung 12 illustriert den Begriff von Zyklen in gerichteten und ungerichteten Graphen. Ein Graph G heißt azyklisch (oder zyklenfrei oder kreisfrei), falls es keine Zyklen in G gibt. Wir verwenden die Abkürzung DAG für „directed acyclic graphs“ (also zyklenfreie Digraphen). Häufig spricht man auch von der Zyklenfreiheit einer Kantenmenge $E' \subseteq E$ und meint damit die Zyklenfreiheit des Teilgraphen $G' = (V, E')$. □

3.2.2 Implementierung von Graphen

Prinzipiell gibt es zwei Möglichkeiten zur computer-internen Darstellung von Graphen. Die einfachste Alternative besteht in der Verwendung von *Adjazenzmatrizen*, welche – unabhängig von der Größe der Kantenrelation E den Platzbedarf $\Theta(|V|^2)$ implizieren. Platzeffizienter ist die Darstellung von Graphen mit *Adjazenzlisten*, da diese lediglich $\Theta(|V| + |E|)$ Platz beanspruchen. Der Summand $\Theta(|V|)$ steht für die Listenköpfe, der Summand $\Theta(|E|)$ für die Gesamtlänge der Adjazenzlisten.

Beispiel zur Darstellung von Graphen



Adjazenzmatrix

	v_1	v_2	v_3	v_4
v_1	0	1	0	1
v_2	0	0	0	1
v_3	0	1	1	0
v_4	0	0	0	0

Adjazenzlisten

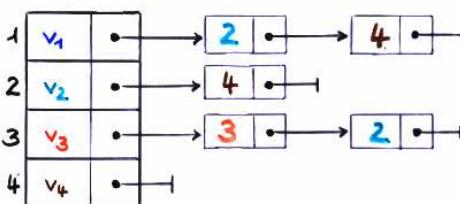


Abbildung 13: Adjazenzmatrizen und Adjazenzlisten

Bei der Adjazenzlistendarstellung werden die direkten Nachfolgermengen $Post(v)$ jeweils durch eine Liste dargestellt. Dabei wird oftmals mit Knotennummern gearbeitet, so daß in der zu Knoten v gehörenden Adjazenzliste genau die Knotennummern j verkettet sind, für die $v_j \in Post(v)$ gilt. (In den folgenden Skizzen der Adjazenzlisten weichen wir oftmals von der oben genannten Darstellungsform ab und schreiben die Knotennamen (anstelle von Knotennummern) in die Adjazenzlisten.) Die gängigste Implementierung mit Adjazenzlisten verwaltet die Listenköpfe (mit eventueller Zusatzinformation über die Knoten) in einem Array, wobei der Eintrag des Knotens v_i an der Array-Position i steht. Alternativ kann man eine Liste verwenden, in der sämtliche Informationen über die Knoten (einschließlich der Listenköpfe für die Adjazenzlisten) verkettet sind. Diese Implementierungsform ist dann sinnvoll, wenn die Knotenzahl flexibel ist. Jedoch sind Zugriffe auf die Adjazenzlisten erschwert.

Beschriftungen der Kanten können als Werte der Adjazenzmatrix gespeichert werden (wobei ein geeigneter Spezialeintrag für Knotenpaare (v, w) mit $(v, w) \notin E$ zu wählen ist)

oder mit Hilfe einer zusätzlichen Komponente für die Listenglieder der Adjazenzlisten dargestellt werden.

Diverse Varianten von Graphen (z.B. endliche Automaten) arbeiten mit Kantenbeschriftungen, die parallele Kanten ermöglichen. Auch diese können mit Adjazenzlisten dargestellt werden. Eine Adjazenzlistendarstellung eines endlichen Automaten ist in Abbildung 14 angegeben.

Varianten von Graphen: z.B. Digraphen mit Kantenrelation

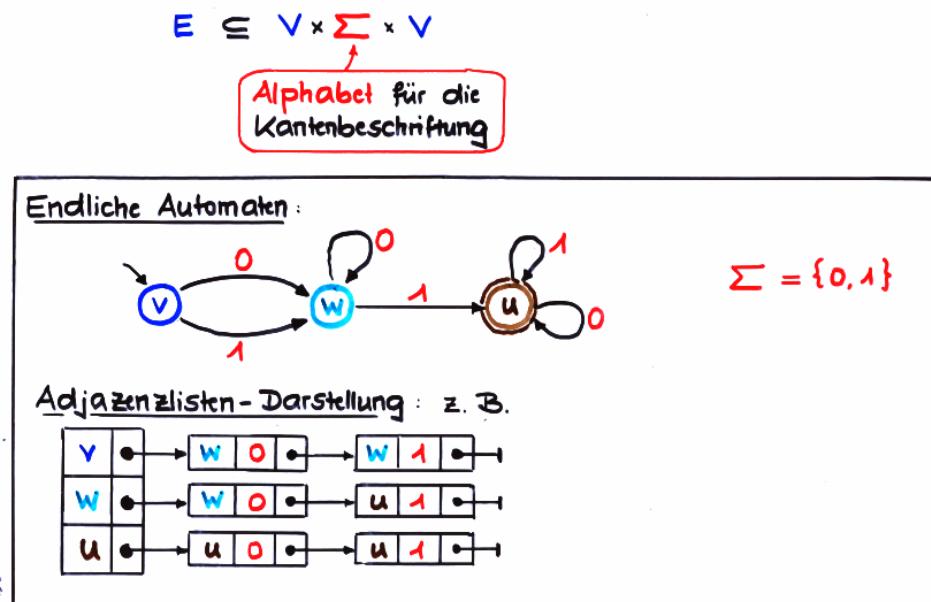


Abbildung 14: Adjazenzlistendarstellung eines endlichen Automaten

3.2.3 Traversierungsmethoden: Tiefen- und Breitensuche

Das generelle Schema (siehe Algorithmus 24) verwendet zwei Mengen von Knoten:

- eine Menge *Visited*, die sich z.B. durch einen Bitvektor darstellen lässt, und welche die Menge aller bereits besuchten Knoten verwaltet,
- eine Teilmenge *Shell* von *Visited*, welche den „Rand“ der bereits besuchten Knoten verwaltet.

Die Randmenge *Shell* charakterisiert solche Knoten $w \in Visited$, für die noch nicht alle ausgehenden Kanten (w, u) untersucht wurden.

Ein Knoten gilt als *besucht*, sobald er in die Menge *Visited* eingefügt wird. Eine Kante gilt als *benutzt*, sobald sie in dem ELSE-Zweig des unten angegebenen Traversierungs-

Algorithmus 24 Schema für Graph-Traversierungsalgorithmen

Visited := \emptyset ; (* verwaltet die bereits besuchten Knoten *)

(* Initial sind alle Kanten „unbenutzt“. *)

(* Schleifeninvariante: $\text{Post}^*(x) \subseteq \text{Visited}$ für alle Knoten $x \in \text{Visited}$ *)

FOR ALL Knoten v DO
IF $v \notin Visited$ THEN

(* Besuche v und alle von v erreichbaren, noch nicht besuchten Zustände *)

Visited := *Visited* $\cup \{v\}$;
Traverse(*v*)

FI
OD

(* Nun gilt $Visited = V$. *)

Traverse(v):

(* $\text{Traverse}(v)$ besucht alle Knoten $w \in \text{Post}^*(v) \setminus \text{Visited}^*$ *)

$Shell := \{v\};$ (* $Shell$ verwaltet die noch nicht vollständig expandierten Knoten *)
(* $Shell =$ „Randmenge von $Visited$ “ *)

WHILE $Shell \neq \emptyset$ DO

wähle einen Knoten $w \in Shell$;

IF es gibt keine von w ausgehende, unbenutzte Kante **THEN**

$$Shell := Shell \setminus \{w\}$$

ELSE

wähle eine unbenutzte Kante (w, u) und markiere diese als benutzt;

IF $u \notin Visited$ THEN

füge u in $Visited$ und $Shell$ ein

FI

I —

1

1

Algorithmus betrachtet wird.²² Wir sagen, daß ein Knoten u über die Kante (w, u) besucht wird, wenn bei der Wahl der Kante (w, u) im ELSE-Zweig des angegebenen Algorithmus der Knoten u noch nicht besucht war.

Durch den Aufruf von $\text{Traverse}(v)$, gestartet mit einer Besuchsmenge Visited , welche die Eigenschaft $\text{Visited} \cap \text{Post}^*(v) = \{v\}$ besitzt, werden genau die von v erreichbaren Knoten besucht. Genauer gilt:

Satz 3.2.10 (Korrekttheit des Traversierungsschemas). Durch den Aufruf von $\text{Traverse}(v)$ in Algorithmus 24 werden genau diejenigen Knoten besucht, die von v erreichbar sind, nicht jedoch von den Knoten v' , für welche $\text{Traverse}(v')$ vor $\text{Traverse}(v)$ in Algorithmus 24 aufgerufen wurde.

Beweis. Wir weisen folgende allgemeinere Aussage nach. Ist v ein Knoten in G und $V_0 \subseteq V \setminus \{v\}$ eine Knotenmenge, so daß:

- (i) für alle Knoten $x \in V_0$ gilt $\text{Post}^*(x) \subseteq V_0$,
- (ii) ist (x, y) eine als benutzt markierte Kante, so ist $\{x, y\} \subseteq V_0$,

dann werden durch $\text{Traverse}(v)$ gestartet mit der Besuchsmenge $\text{Visited} = V_0 \cup \{v\}$ genau diejenigen Knoten w besucht, welche in $\text{Post}^*(v) \setminus V_0$ liegen. Beachte, daß initial $V_0 = \emptyset$ und alle Kanten unbenutzt sind. Daher sind zu Beginn von Algorithmus 24 beide Bedingungen (i) und (ii) für $V_0 = \text{Visited} = \emptyset$ erfüllt. Nach der Ausführung von $\text{Traverse}(v)$ sind die Bedingung (i) und (ii) für die resultierende Besuchsmenge $V'_0 = V_0 \cup \text{Post}^*(v)$ erfüllt, so daß dieselbe Argumentation für den nächsten Knoten v' , für den in Algorithmus 24 $\text{Traverse}(v')$ aufgerufen wird, greift.

Wir nehmen nun an, daß unmittelbar vor dem Aufruf von $\text{Traverse}(v)$ die Bedingungen (i) und (ii) für die Besuchsmenge $V_0 = \text{Visited}$ erfüllt sind und zeigen, daß in $\text{Traverse}(v)$ genau diejenigen Knoten w besucht werden, welche in $\text{Post}^*(v) \setminus V_0$ liegen. Dies ergibt sich aus folgender *Schleifeninvariante*:²³

- (1) $\text{Shell} \subseteq \text{Post}^*(v)$
- (2) für jeden Knoten $x \in \text{Post}^*(v) \setminus \text{Visited}$ gibt es einen Pfad

$$v, v_1, \dots, v_{r-1}, \underbrace{v_r, v_{r+1}, \dots, v_k, x}_{\text{nur unbenutzte Kanten}} \quad \in \text{Shell}$$

von v nach x , der ein nichtleeres Suffix v_r, v_{r+1}, \dots, x beginnend in einem Knoten $v_r \in \text{Shell}$ und gebildet aus unbenutzten Kanten besitzt.

²²Im Falle eines ungerichteten Graphen kann man zulassen, daß jede Kante $(w, u) = (u, w)$ zweimal benutzt wird; nämlich einmal als von w ausgehende Kante und einmal als von u ausgehende Kante. Dies ist jedoch für die Korrektheit und asymptotische Laufzeit irrelevant. Sofern die Kante (w, u) in dieser Orientierung bereits benutzt wurde, bleibt die spätere Benutzung der Kante (u, w) (als von u ausgehende Kante) ohne Effekt, da der Zielknoten w dann bereits in Visited liegt.

²³Mit einer Schleifeninvariante ist eine Bedingung gemeint, die unmittelbar vor und nach jedem Schleifendurchlauf erfüllt ist, und somit durch die Ausführung des Schleifenrumpfs unverändert also invariant bleibt.

Nach der Terminierung von $\text{Traverse}(v)$ ist $\text{Shell} = \emptyset$. Aufgrund von (2) kann es keinen Knoten $x \in \text{Post}^*(v) \setminus \text{Visited}$ geben. Also ist $\text{Post}^*(v) \subseteq \text{Visited}$ nach der Terminierung von $\text{Traverse}(v)$. Zusammen mit (1) folgt hieraus, daß unter den Voraussetzungen (i) und (ii) während der Ausführung von $\text{Traverse}(v)$ genau diejenigen Knoten aus $\text{Post}^*(v)$ besucht werden, die vor Aufruf von $\text{Traverse}(v)$ noch nicht in Visited lagen.

Wir zeigen nun die Gültigkeit der genannten Schleifeninvariante. Bedingung (1) ist klar, da initial $\text{Shell} = \{v\} \subseteq \text{Post}^*(v)$ und da ein Knoten u nur dann in Shell eingefügt wird, wenn u über eine Kante (w, u) erreichbar ist, wobei w ein Knoten ist, der zuvor in Shell lag. Für jeden Knoten $u \in \text{Shell}$ gibt es also einen Pfad v, w_1, \dots, w_r, u bestehend aus Knoten, die während $\text{Traverse}(v)$ besucht wurden. Insbesondere gilt $u \in \text{Post}^*(v)$.

Bedingung (2) ist initial erfüllt, da für jeden Knoten $x \in \text{Post}^*(v) \setminus V_0$ und jeden einfachen Pfad $v = w_0, w_1, \dots, w_r, w_{r+1} = x$ keiner der Knoten w_1, \dots, w_r in V_0 liegen kann. (Andernfalls wäre $x \in \text{Post}^*(w_i) \subseteq V_0$ für ein $i \in \{1, \dots, r\}$, wegen Voraussetzung (i).) Voraussetzung (ii) induziert, daß keine der Kanten (w_i, w_{i+1}) vor dem ersten Schleifendurchlauf als benutzt markiert ist. Wir nehmen nun an, daß Bedingung (2) unmittelbar vor dem j -ten Schleifendurchlauf erfüllt ist und zeigen, daß (2) auch unmittelbar nach dem j -ten Schleifendurchlauf gilt. Falls im j -ten Schleifendurchlauf das gewählte Element $w \in \text{Shell}$ aus Shell entfernt wird, so sind alle aus w hinausführenden Kanten als benutzt markiert. Bedingung (2) ist von dieser Veränderung der Randmenge Shell offenbar nicht betroffen (da $w = v_r$ nicht möglich ist). Wir nehmen nun an, daß eine aus w hinausführende Kante (w, u) im j -ten Schleifendurchlauf als benutzt markiert wird. Sei $x \in \text{Post}^*(v)$, so daß $x \notin \text{Visited}$ unmittelbar nach dem j -ten Schleifendurchlauf. Dann ist $x \neq u$. Sei

$$v, v_1, \dots, v_{r-1}, \overbrace{v_r}^{\in \text{Shell}}, v_{r+1}, \dots, v_k, x$$

ein einfacher Pfad von v nach x , dessen Suffix $v_r, v_{r+1}, \dots, v_k, x$ aus Kanten gebildet ist, die vor dem j -ten Schleifendurchlauf nicht benutzt wurden. Falls die Kante (w, u) nicht im Suffix $v_r, v_{r+1}, \dots, v_k, x$ beteiligt ist, gilt Bedingung (2) für Knoten x auch nach dem j -ten Schleifendurchlauf. Falls $(w, u) = (v_i, v_{i+1})$ für ein $i \in \{r, r+1, \dots, k-1\}$, so betrachten wir den Pfad

$$v, v_1, \dots, v_{r-1}, v_r, v_{r+1}, \dots, \overbrace{v_i}^w, \underbrace{v_{i+1}, \dots, v_k, x}_{\text{nur unbenutzte Kanten}}^{\in \text{Shell}}$$

Unmittelbar nach dem j -ten Schleifendurchlauf ist keine der Kanten in dem Suffix $u = v_{i+1}, \dots, v_k, v_{k+1} = x$ als besucht markiert. Ferner liegt u in Shell , da andernfalls die aus u hinausführende Kante (v_{i+1}, v_{i+2}) als benutzt markiert wäre. \square

Satz 3.2.11 (Terminierung und Anzahl an Schleifendurchläufen des Traversierungsschemas). Die Anzahl an Durchläufen der WHILE-Schleife, die durch die Ausführung des Traversierungsschemas (Algorithmus 24) innerhalb $\text{Traverse}(\cdot)$ ausgeführt werden, ist $\Theta(n + m)$, wobei n für die Anzahl an Knoten und m für die Anzahl an Kanten steht.

Beweis. In jeder Iteration der WHILE-Schleife wird entweder ein Knoten aus Shell entfernt oder eine Kante als benutzt markiert. Da jeder Knoten höchstens einmal in

$Shell$ eingefügt (und somit auch nur einmal aus $Shell$ gelöscht werden kann) und die Benutzt-Markierung einer Kante niemals rückgängig gemacht wird, sind höchstens $n + m$ Schleifendurchläufe möglich. Andererseits wird jeder Knoten genau einmal in $Shell$ aufgenommen und aus $Shell$ entfernt und jede Kante genau einmal als benutzt markiert. \square

Corollary 3.2.12. $Traverse(v)$ gestartet mit der Besuchsmenge $Visited = \{v\}$ durchläuft den von v erreichbaren Teil des Graphen in Zeit $\Theta(n_v + m_v)$, wobei $n_v = |Post^*(v)|$ und $m_v = |\{(v, w) \in E : v, w \in Post^*(v)\}|$.

Die gängigsten Instanzen des skizzierten Traversierungsschemas sind die sog. Tiefen- und Breitensuche.

- **Depth-First-Search (DFS, Tiefensuche):** Verwaltung der Randmenge $Shell$ durch einen Stack
- **Breadth-First-Search (BFS, Breitensuche):** Verwaltung von $Shell$ durch eine Queue.

Im Folgenden gehen wir von einer Adjazenzlistendarstellung aus. Diese erlaubt es, auf die Benutzt-Markierungen der Kanten zu verzichten und stattdessen die Adjazenzlisten „von links nach rechts“ zu durchlaufen, um auf die jeweils nächste noch unbenutzte ausgehende Kante eines Knotens zuzugreifen. Werden die Listenköpfe in einem Array verwaltet, dann lässt sich die äußere FOR-Schleife (in der alle Knoten v durchlaufen werden) durch einen Durchlauf aller Array-Positionen implementieren.

Breitensuche (BFS). Für die Breitensuche (siehe Algorithmus 25) kann man das Entfernen eines Knotens w aus der Randmenge $Shell$ sofort nach der „Wahl“ des Knotens w durchführen. Die innere FOR-Schleife, die alle direkten Nachfolger u des gewählten Knotens w durchläuft, entspricht einem Durchlauf durch die Adjazenzliste von w .

Tiefensuche (DFS). In ähnlicher Weise lässt sich die Tiefensuche implementieren, die anstelle einer Warteschlange einen Keller benutzt. Siehe Algorithmus 26. Für die Tiefensuche kann eine rekursive Formulierung angegeben werden. Siehe Algorithmus 27. Diese macht die explizite Verwaltung der Randmenge $Shell$ durch einen Stack überflüssig. Die Randmenge $Shell$ ergibt sich dann aus dem Rekursionsstack.

Wie für das allgemeine Traversierungsschema werden für $Visited \cap Post^*(v) = \{v\}$ durch die Anweisungen ab der mit dem Kommentar $Traverse(v)$ versehenen Zeile in der Breiten- und Tiefensuche alle Nachfolger von v besucht. Andere Knoten $w \notin Visited \cup Post^*(v)$ werden erst in einem späteren Durchlauf der äußeren FOR-Schleife besucht.

Algorithmus 25 Breitensuche (BFS)

Visited := \emptyset ;
Initialisiere *Shell* als leere Warteschlange;
FOR ALL Knoten v **DO**
 IF $v \notin \text{Visited}$ **THEN**
 füge v in *Visited* ein; (* *Traverse*(v) *)
 Add(*Shell*, v);
 WHILE *Shell* $\neq \emptyset$ **DO**
 $w := \text{Front}(\text{Shell})$;
 Remove(*Shell*);
 (* Durchlaufe die Adjazenzliste von w und *)
 (* füge alle noch nicht besuchten Söhne von w in *Shell* ein *)
 FOR ALL $u \in \text{Post}(w)$ **DO**
 IF $u \notin \text{Visited}$ **THEN** füge u in *Visited* ein; *Add*(*Shell*, u) **FI**
 OD
 OD
 FI
 OD

Algorithmus 26 Tiefensuche (iterative Formulierung mit Keller)

Visited := \emptyset ;
Initialisiere *Shell* als leeren Keller;
FOR ALL Knoten v **DO**
 IF $v \notin \text{Visited}$ **THEN**
 füge v in *Visited* ein; (* *Traverse*(v) *)
 Push(*Shell*, v);
 WHILE *Shell* $\neq \emptyset$ **DO**
 $w := \text{Top}(\text{Shell})$;
 IF das Ende der Adjazenzliste von w ist erreicht **THEN**
 Pop(*Shell*);
 ELSE
 $u :=$ nächster Knoten in der Adjazenzliste von w
 IF $u \notin \text{Visited}$ **THEN** füge u in *Visited* ein; *Push*(*Shell*, u) **FI**
 FI
 OD
 FI
 OD

Algorithmus 27 Tiefensuche (rekursive Formulierung)

Visited := \emptyset ;
FOR ALL Knoten v **DO**
 IF $v \notin \text{Visited}$ **THEN**
 $DFS(v)$
 FI
OD

Rekursiver Algorithmus $DFS(w)$:

füge w in *Visited* ein;
 (* Durchlaufe die Adjazenzliste von w und führe $DFS(u)$ *)
 (* für alle noch nicht besuchten Söhne u von w aus *)
FOR ALL $u \in \text{Post}(w)$ **DO**
 IF $u \notin \text{Visited}$ **THEN**
 $DFS(u)$
 FI
OD

DFS- und BFS-Besuchsreihenfolgen. Betrachtet man die Besuchsreihenfolgen, die sich durch die BFS oder DFS ergeben, dann ist stets zu beachten, daß diese wesentlich davon abhängen, in welcher Reihenfolge die Knoten in den Adjazenzlisten stehen und in welcher Reihenfolge die Knoten v , für welche die Traversierung gestartet wird, betrachtet werden.²⁴ Als Beispiel betrachten wir die Graphen in Abbildung 10. Steht w vor x in der Adjazenzliste von v , so liefern die Tiefen- und Breitensuche gestartet mit Knoten v die Besuchsreihenfolge v, w, x, y . Steht jedoch x vor w in der Adjazenzliste von v , so ist die DFS-Besuchsreihenfolge (für die DFS gestartet mit Knoten v) gleich v, x, y, w , während mit der Breitensuche gestartet mit v die Knoten in der Reihenfolge v, x, w, y besucht werden.

Kosten. Wie zuvor gehen wir von einer Darstellung mit Adjazenzlisten aus. Da die DFS und BFS die Adjazenzlisten jedes Knotens genau einmal durchlaufen, wird jede Kante genau einmal im gerichteten Fall und genau zweimal im ungerichteten Fall betrachtet. Weiter wird das Knoten-Array komplett durchlaufen. Man beachte, daß die Verwendung der Menge *Visited* sicherstellt, daß jeder Knoten genau einmal in die Warteschlange Q bzw. in den Stack *Shell* aufgenommen wird. Weiter verursachen die Operationen auf Warteschlangen und Stacks jeweils nur konstante Kosten. Setzt man voraus, daß die Menge *Visited* durch einen Bitvektor repräsentiert wird, so erhält man in beiden Fällen lineare Kosten:

²⁴Letzteres hängt bei einer Verwaltung der Listenköpfe in einem Array von der Reihenfolge ab, in der die Knoten und zugehörigen Listenköpfe in dem Array abgelegt sind.

Satz 3.2.13 (Kosten der BFS und DFS). Mit der Breiten- oder Tiefensuche in einem (gerichteten oder ungerichteten) Graphen $G = (V, E)$, der durch Adjazenzlisten dargestellt ist, können alle Knoten $v \in V$ in Zeit $\Theta(n + m)$ besucht werden. Der zusätzliche Platzbedarf ist jeweils $\Theta(n)$. Dabei ist $n = |V|$ die Knoten- und $m = |E|$ die Kantenanzahl. Mit einer Darstellung durch Adjazenzmatrizen anstelle von Adjazenzlisten ergibt sich dagegen die Laufzeit $\Theta(n^2)$.

Die Einsatzmöglichkeiten der Tiefen- und Breitensuche sind vielfältig. Wir werden später (Kapitel 6) einige Anwendungen kennenlernen. Ein sehr einfaches Beispiel ist die Berechnung der Zusammenhangskomponenten eines ungerichteten Graphen. Wahlweise kann eine Tiefen- oder Breitensuche eingesetzt werden. Für beide Traversierungsstrategien werden in jeder Iteration der äußeren FOR-Schleife, in der ein neuer, zuvor noch nicht besuchter Startknoten v gewählt wird, genau die Zustände einer Zusammenhangskomponente besucht. Mit einer Adjazenzlistendarstellung ergibt sich damit die Laufzeit $\Theta(|V| + |E|)$.

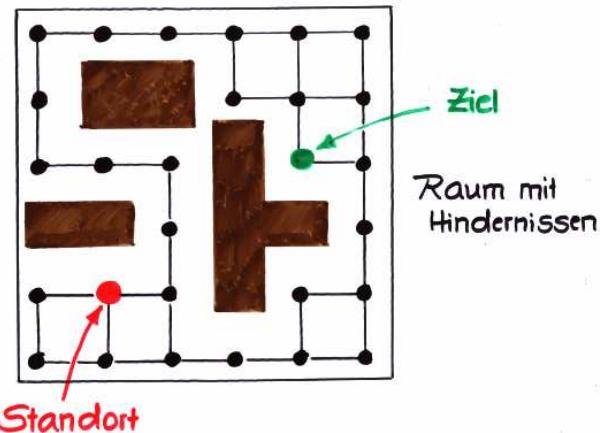
Ein weiteres einfaches Beispiel für eine Anwendung der Breitensuche wird in folgendem Unterabschnitt diskutiert.

3.2.4 Kürzeste Wege (single source, Einheitskosten)

In zahlreichen Anwendungen spielen Wegeprobleme (die Frage nach einem Weg mit gewissen Eigenschaften zwischen zwei Knoten) eine große Rolle. Zu den Standardbeispielen zählen ein Bahnhofschaftssystem, das auf Benutzeranfragen wie „Bestimme die schnellste (oder kürzeste oder billigste) Zugverbindung zwischen München und Kiel“ in Echtzeit reagieren muß. Weitere Standardbeispiele sind Fragestellungen aus den Bereich Jobscheduling oder der Robotik wie „Bestimme eine Ausführungsreihenfolge von Jobs, so daß die Gesamtausführungszeit minimal ist“.²⁵ oder „Gegeben ist ein Raum mit Hindernissen. Finde eine Folge von Roboterschritten, die den Roboter in kürzester Zeit vom Standard A zum Zielort Z bringt.“

²⁵Dabei werden Relationen zugrundegelegt, die Aufschluß darüber geben, welche Jobs parallel ausgeführt werden können und welche Jobs erst bei Vorliegen der Resultate von anderen Jobs gestartet werden können.

Wege-Probleme, z.B. Kürzester Weg (Einheitskostenfkt.)



Gesucht: minimale Anzahl an Roboterschritten, die vom Standort zum Ziel führen

©

Kürzeste Wege-Probleme. Der Ausgangspunkt ist ein Graph $G = (V, E)$. Man unterscheidet zwischen folgenden Fragestellungen:

- **single-source:** Gegeben ist ein „Startknoten“ v . Bestimme kürzeste Wege von v zu allen Knoten w .
- **all-pairs:** Bestimme kürzeste Wege für alle Knotenpaare (v, w) .

Es gibt zusätzlich die Variante „single source-single target“, bei der Start- und Zielknoten festgelegt sind. Jedoch besteht die beste bekannte Lösung hierfür in der Berechnung kürzester Wege von dem Startknoten zu allen anderen Knoten.

Für die genannten Varianten ist stets zu unterscheiden, ob die kürzesten Wege hinsichtlich der Einheitskostenfunktion, die jeder Kante eine Kosteneinheit zuordnet, zu bestimmen sind oder ob eine Kostenfunktion vorliegt, die den Kanten unterschiedliche Werte zuordnen kann. Im letzten Fall müsste es korrekter Weise „billigste Wege“ statt „kürzeste Wege“ heißen. Dennoch ist die Sprechweise „kürzeste Wege“ gebräuchlich. In diesem Abschnitt betrachten wir nur das „single-source-Wegeproblem“ mit Einheitskosten. Andere Kürzeste-Wege-Probleme werden in Kapitel 6 besprochen.

Die Frage nach kürzesten Wegen von einem festen Startknoten v (single-source, Einheitskosten) ausgehend kann leicht mit Hilfe der Breitensuche beantwortet werden. Im Folgenden sei $G = (V, E)$ ein ungerichteter oder gerichteter Graph und $v \in V$ ein Knoten, den wir als Startknoten auszeichnen. Unser Ziel ist die Berechnung von

$$\ell(w) = \min\{\text{length}(\pi) : \pi \text{ ist Pfad von } v \text{ nach } w\}$$

und eines zugehörigen kürzesten Pfads von v nach w , falls es einen solchen gibt. (Wir setzen $\ell(w) = \infty$, falls w nicht von v erreichbar ist.) Die Idee eines BFS-basierten

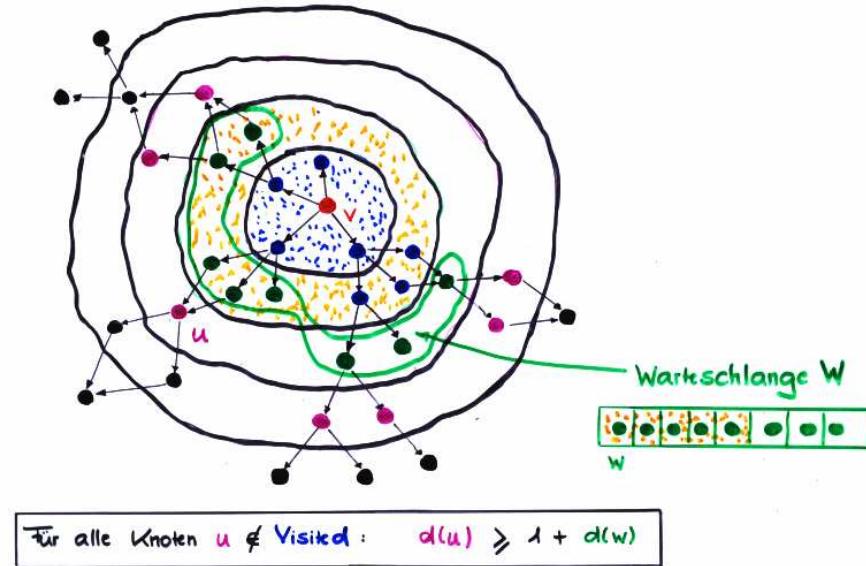
Algorithmus besteht darin, mit den Werten $\ell(v) = 0$ und $\ell(w) = \infty$ zu beginnen, eine Breitensuche mit Knoten v zu starten und jeweils $\ell(u) = \ell(w) + 1$ zu setzen, sobald Knoten u über die Kante (w, u) besucht wird. Tatsächlich erhält man auf diese Weise die gewünschten Werte $\ell(w)$.

Die Korrektheit des Verfahrens folgt aus der Beobachtung, daß zu jedem Zeitpunkt der Breitensuche gestartet mit v die Warteschlange Q die Form $w_1, \dots, w_k, w_{k+1}, \dots, w_r$ hat, wobei $\ell(w_1) = \dots = \ell(w_k) = \ell$ und $\ell(w_{k+1}) = \dots = \ell(w_r) = \ell + 1$ für ein $\ell \in \mathbb{N}$. Der Fall $r = k$ ist dabei möglich. Weiter gilt für eine derartige Konfiguration der Warteschlange, daß

- alle Knoten x mit $\ell(x) < \ell$ bereits abgearbeitet wurden,
- alle Knoten x mit $\ell(x) = \ell$ entweder in Q liegen oder bereits abgearbeitet wurden,
- noch keiner der Knoten x mit $\ell(x) > \ell + 1$ besucht wurde.

Mit der Formulierung „abgearbeiteter Knoten“ meinen wir einen Knoten w mit $\ell(w) < \infty$, welcher der Warteschlange Q bereits entnommen wurde. (Dies entspricht der Bedingung $w \in Visited \setminus Shell$ in der ursprünglichen Formulierung der Breitensuche). Insbesondere haben alle Knoten $u \in V \setminus Visited$ den Abstand $\ell(u) \geq \ell + 1$ zu v . Für $w = w_1$ resultiert hieraus, daß $\ell(u) = \ell + 1$ für jeden Knoten $u \in Post(w) \setminus Visited$.

Abstandszonen von Knoten v



(* Auf der Folie muss es $\ell(u)$ statt $d(u)$ heißen. *)

Auf kürzeste Wege kann durch die Angabe der letzten Kante eines kürzesten Wegs geschlossen werden. Wird u über die Kante (w, u) während der Ausführung der Breitensuche besucht, dann genügt es, sich den Knoten w als direkten Vorgänger von u auf einem kürzesten Weg von v nach u zu merken.

Algorithmus 28 BFS-basierte Kürzeste-Wege-Berechnung mit Startknoten v

FOR ALL $w \in V \setminus \{v\}$ **DO**

$\ell(w) := \infty$

OD

$\ell(v) := 0;$

(* $\ell(w) < \infty$ entspricht der Bedingung $w \in Visited$ *)

$Shell :=$ leere Warteschlange;

$Add(Shell, v);$

WHILE $Shell \neq \emptyset$ **DO**

$w := Front(Shell);$

$Remove(Shell);$

(* expandiere w *)

FOR ALL $u \in Post(w)$ **DO**

IF $\ell(u) = \infty$ **THEN**

$\ell(u) := \ell(w) + 1;$

$father(u) := w;$ (* (w, u) ist letzte Kante eines kürzesten Wegs von v nach u *)

$Add(Shell, u)$

FI

OD

OD

(* Für alle Knoten w ist $\ell(w)$ die Länge eines kürzesten Wegs von v nach w . *)

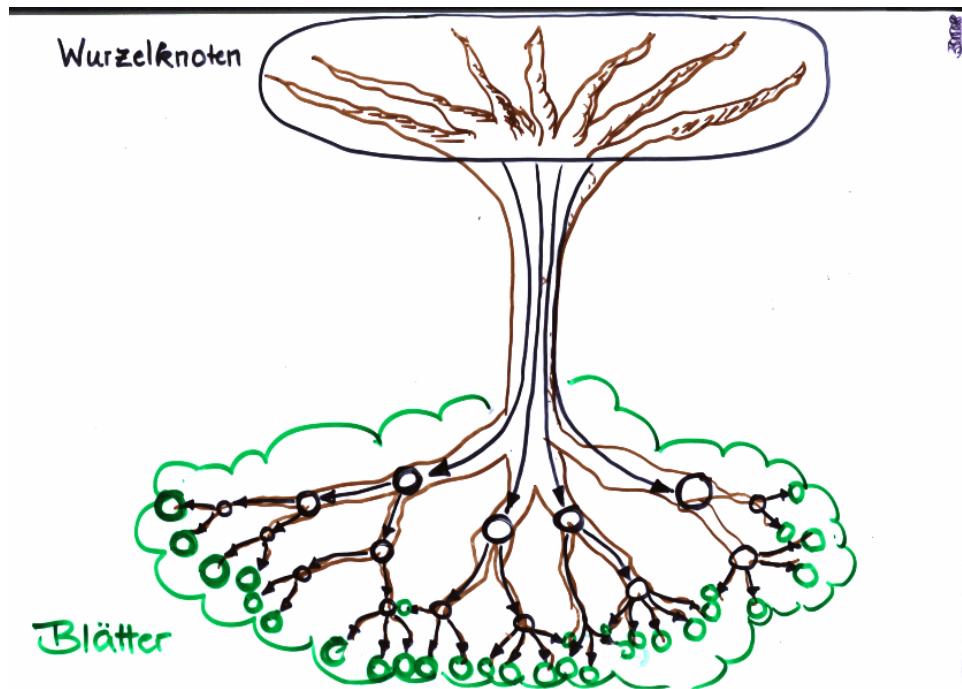
(* Falls $\ell(w) = \infty$, so ist $w \notin Post^*(v)$. Es gibt also keinen Pfad von v nach w . *)

Diese Ideen führen zu Algorithmus 28. Das *father*-Feld dient der Rekonstruktion kürzester Wege. Ist nämlich $\ell(w) < \infty$, so ist v_0, \dots, v_r ein kürzester Pfad von $v_0 = v$ zu $v_r = w$, wobei $v_{i-1} = \text{father}(v_i)$, $i = r, r-1, \dots, 1$.

Die Laufzeit von Algorithmus 28 kann durch $\mathcal{O}(n + m)$ nach oben abgeschätzt werden (wobei wie zuvor $n = |V|$ und $m = |E|$). Präzise kann die Laufzeit durch $\Theta(n_v + m_v)$ angegeben werden, wobei $n_v = |\text{Post}^*(v)|$ die Anzahl an von v erreichbaren Knoten ist und $m_v = |E \cap (\text{Post}^*(v) \times \text{Post}^*(v))|$.

3.2.5 Gerichtete Bäume

Naja, fast wie in der Botanik:



Definition 3.2.14 (Gerichteter Baum). Ein gerichteter Baum (kurz Baum genannt) ist ein Digraph $\mathcal{T} = (V, E)$, für den gilt:

- entweder ist $V = \emptyset$ (in diesem Fall wird \mathcal{T} der *leere Baum* genannt)
- oder es existiert ein Knoten $v_0 \in V$, von dem jeder Knoten $v \in V$ über genau einen Pfad von v_0 erreichbar ist.

Der Knoten v_0 ist eindeutig bestimmt (siehe Satz 3.2.15 unten) und wird *Wurzel* oder *Wurzelknoten* von \mathcal{T} genannt. Jeder Knoten $v \in V$ mit $\text{Post}(v) \neq \emptyset$ heißt *innerer Knoten* von \mathcal{T} . Die Knoten $w \in \text{Post}(v)$ werden *Söhne* von v genannt. Entsprechend wird v als *Vater* der Knoten $w \in \text{Post}(v)$ bezeichnet. Alle Knoten $v \in V$ mit $\text{Post}(v) = \emptyset$ heißen *Blätter* von \mathcal{T} . \square

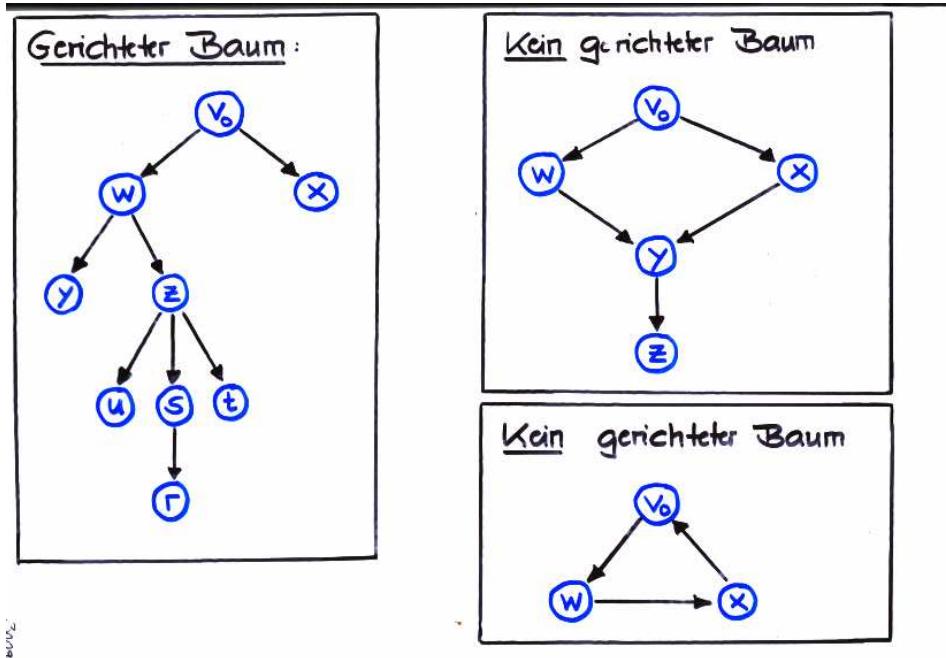


Abbildung 15: Gerichteter Baum

Der folgende Satz zeigt, daß jeder nicht-leere Baum genau eine Wurzel hat und daß jeder Knoten (außer der Wurzel) genau einen Vater hat.

Satz 3.2.15 (Alternative Charakterisierung von gerichteten Bäumen). Sei $\mathcal{T} = (V, E)$ ein Digraph mit $V \neq \emptyset$. Dann sind folgende Aussagen äquivalent.

- (a) \mathcal{T} ist ein nicht-leerer Baum.
- (b) Es gibt genau einen Knoten $v_0 \in V$ mit $Pre(v_0) = \emptyset$ und für alle Knoten $v \in V \setminus \{v_0\}$ gilt $v \in Post^*(v_0)$ und $|Pre(v)| = 1$.

Beweis. Sei \mathcal{T} ein nicht-leerer Baum mit Wurzelknoten v_0 .

(a) \implies (b): Zu zeigen sind drei Eigenschaften:

1. $Pre(v_0) = \emptyset$
2. $Pre(v) \neq \emptyset$ für alle Knoten $v \neq v_0$
3. $|Pre(v)| \leq 1$ für alle Knoten $v \neq v_0$

ad 1. Wir nehmen an, daß $w \in Pre(v_0)$. Dann sind $\pi_1 = v_0$ (Pfad der Länge 0) und $\pi_2 = v_0, \dots, w, v_0$ Pfade, die von v_0 zu v_0 führen. Das Präfix v_0, \dots, w steht dabei für den eindeutig bestimmten Pfad, der von v_0 nach w führt. Widerspruch.

ad 2. Jeder Knoten $v \in V \setminus \{v_0\}$ muß mindestens einen Vorgänger $w \in Pre(v)$ haben, da es andernfalls keinen Pfad von v_0 nach v geben könnte.

ad 3. Wir nehmen an, daß es einen Knoten $v \in V$ gibt, welcher zwei Vorgänger $w_1, w_2 \in Pre(v)$ (mit $w_1 \neq w_2$) hat. Seien $\pi_1 = v_0, \dots, w_1, v$, $\pi_2 = v_0, \dots, w_2, v$ Pfade von der Wurzel v_0 zu w_1 bzw. w_2 . Dann sind $\pi_1 = v_0, \dots, w_1, v$ und $\pi_2 = v_0, \dots, w_2, v$ zwei verschiedene Wege von v_0 zu v . Widerspruch.

(b) \implies (a): Zu zeigen ist die Eindeutigkeit von Pfaden, die von v_0 zu gegebenem Knoten $v \in V$ führen. Angenommen es gibt zwei (verschiedene) Wege π_1 und π_2 , welche von v_0 zu v führen. Wir betrachten das längste gemeinsame Suffix x_k, x_{k-1}, \dots, x_0 der beiden Pfade von π_1 und π_2 und die Knoten w_1, w_2 , welche diesem Suffix in π_1 bzw. π_2 vorangehen, also:

$$\begin{aligned}\pi_1 &= v_0, \dots, w_1, x_k, x_{k-1}, \dots, x_0, \\ \pi_2 &= v_0, \dots, w_2, x_k, x_{k-1}, \dots, x_0 \\ w_1 &\neq w_2\end{aligned}$$

(Dabei ist $x_0 = v$.) Dann sind w_1 und w_2 zwei verschiedene Vorgänger von Knoten x_k . Also $|Pre(x_k)| \geq 2$. Widerspruch. \square

Im Folgenden schreiben wir $Wurzel(\mathcal{T})$ um den eindeutig bestimmten Wurzelknoten eines nicht-leeren Baums \mathcal{T} zu bezeichnen. Wir setzen $Wurzel(\mathcal{T}) = \perp$ für den leeren Baum \mathcal{T} . Entsprechend schreiben wir $father(v)$ für den eindeutig bestimmten direkten Vorgänger eines Knotens $v \in V$, der nicht der Wurzelknoten ist, und $father(v_0) = \perp$ für den Wurzelknoten v_0 . Zwei Knoten w_1, w_2 mit $w_1 \neq w_2$ und $father(w_1) = father(w_2)$ werden auch *Brüder* genannt.

Da in einem nicht-leeren Baum jeder Knoten außer der Wurzel genau eine einführende Kante hat, erhalten wir:

Corollar 3.2.16. Jeder nicht-leere Baum mit n Knoten hat $n - 1$ Kanten.

Definition 3.2.17 (Verzweigungsgrad, Binärbaum, entarteter Baum). Sei $\mathcal{T} = (V, E)$ ein Baum, $k \in \mathbb{N}$, $k \geq 1$. \mathcal{T} hat den Verzweigungsgrad k (oder kurz Grad k), falls $|Post(v)| \leq k$ für alle $v \in V$. Bäume vom Grad 2 werden auch Binärbäume genannt. Bäume vom Grad 1 heißen entartete Bäume. \square

Definition 3.2.18 (Höhe, Tiefe, Teilbaum, Vollständigkeit). Die Höhe eines nicht-leeren Baums $\mathcal{T} = (V, E)$ ist die Länge eines längsten Pfads in \mathcal{T} und wird mit $Höhe(\mathcal{T})$ oder $H(\mathcal{T})$ bezeichnet. Für den leeren Baum \mathcal{T} definieren wir $Höhe(\mathcal{T}) = -1$. Für jeden Knoten $v \in V$ bezeichnet

$$\mathcal{T}(v) = \mathcal{T}_v = (Post^*(v), E_v), \quad E_v = E \cap (Post^*(v) \times Post^*(v))$$

den zu v gehörenden *Teilbaum*. Die Tiefe von v (in \mathcal{T}) ist die Länge des (eindeutig bestimmten) Pfads vom Wurzelknoten v_0 zu v . Die Höhe von v ist die Höhe des Baums Teilbaums $\mathcal{T}(v)$. Diese wird auch mit $Höhe(v)$ oder $H(v)$ bezeichnet. Ist \mathcal{T} ein Baum, dann heißt \mathcal{T} vollständig vom Grad k , falls $|Post(v)| = k$ für alle inneren Knoten $v \in V$ und alle Blätter dieselbe Tiefe haben. \square

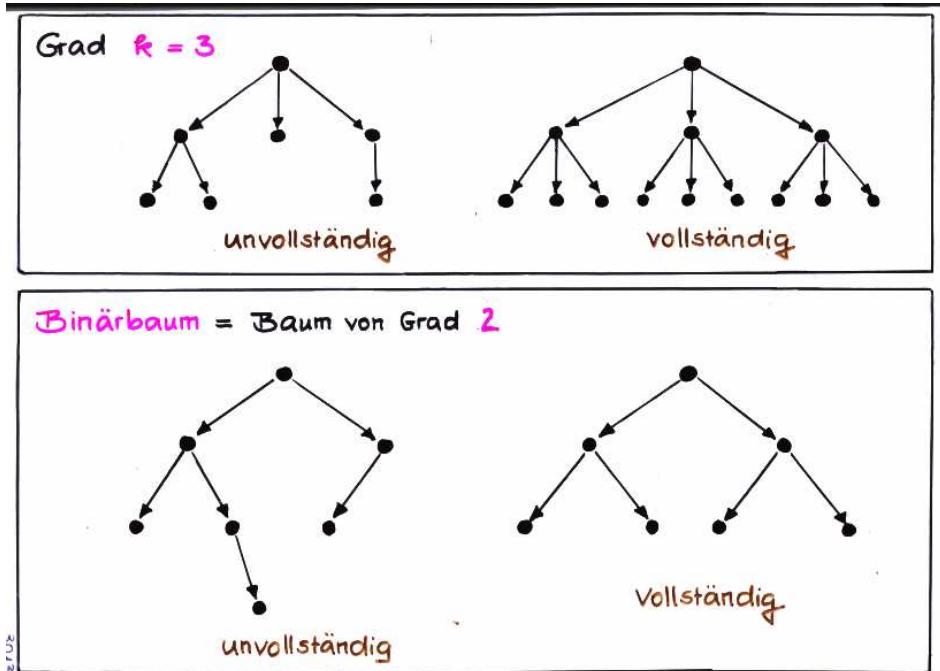


Abbildung 16: Beispiele zu Definition 3.2.17

Satz 3.2.19 (Knotenanzahl versus Höhe). Sei $\mathcal{T} = (V, E)$ ein Baum. Weiter sei $n = |V|$ die Knotenanzahl und $h = \text{Höhe}(\mathcal{T})$ die Höhe von \mathcal{T} . Dann gilt: Hat \mathcal{T} den Grad k , so ist

$$n \leq \frac{k^{h+1} - 1}{k - 1}.$$

Insbesondere gilt für $k \geq 2$: $\text{Höhe}(\mathcal{T}) \geq \log_k(n \cdot (k - 1) + 1) - 1$. Für $k = 2$ (Binärbäume) gilt

$$n \leq 2^{h+1} - 1 \text{ und } \text{Höhe}(\mathcal{T}) \geq \log_2(n + 1) - 1.$$

Beweis. Sei $h = \text{Höhe}(\mathcal{T})$. Ist $h = -1$, so ist \mathcal{T} der leere Baum. Dieser hat

$$n = 0 = \frac{k^0 - 1}{k - 1}$$

Knoten. Ist $h \geq 0$, so kann die Anzahl an Knoten durch

$$\sum_{i=1}^h k^i = \frac{k^{h+1} - 1}{k - 1}$$

nach oben abgeschätzt werden, da es maximal k^i Knoten der Tiefe i geben kann. \square

Definition 3.2.20 (Gerichteter Wald). Ein gerichteter Wald (kurz Wald) ist ein Graph $G = (V, E)$, der sich aus gerichteten Bäumen zusammensetzt, d.h. für den es gerichtete

$\text{Höhe}(\mathcal{T}) = H(\mathcal{T}) = \text{Länge eines längsten Pfads in } \mathcal{T}$

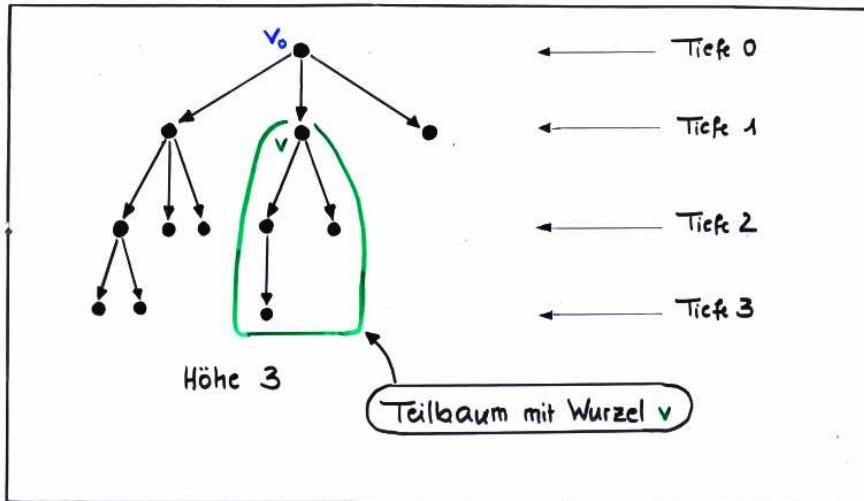
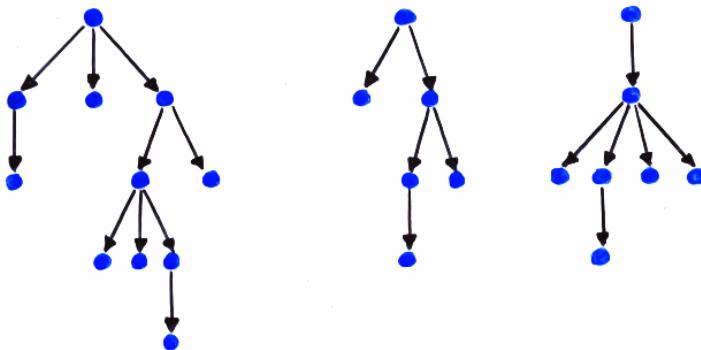


Abbildung 17: Höhe, Tiefe, Teilbaum

Bäume $\mathcal{T}_i = (V_i, E_i)$, $i = 1, \dots, k$, gibt, so daß sich V als „Vereinigung“ der Bäume $\mathcal{T}_1, \dots, \mathcal{T}_k$ darstellen läßt. Formal heißt das, daß $V = V_1 \cup \dots \cup V_k$ und $E = E_1 \cup \dots \cup E_k$, wobei V_1, \dots, V_k paarweise disjunkt sind, $E_i = E \cap (V_i \times V_i)$, $i = 1, \dots, k$, und wobei (V_i, E_i) gerichtete Bäume sind. \square

Gerichteter Wald



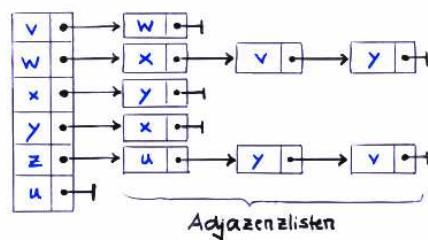
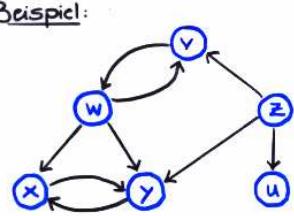
3026a

DFS- und BFS-Wälder. Abschließend erwähnen wir, daß die Tiefen- und Breitensuche in einem Digraphen $G = (V, E)$ jeweils einen Wald induzieren, dessen Knoten die Knoten von G sind. Die Kante $(v, w) \in E$ ist genau dann eine Kante im DFS- bzw. BFS-Wald, wenn Knoten w über die Kante (v, w) bei der Ausführung der Tiefen- bzw. Breitensuche besucht wurde.

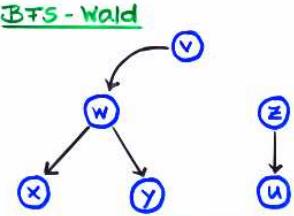
Man beachte, daß ein Graph mehrere BFS- und DFS-Wälder haben kann, die sich durch unterschiedliche Anordnungen der Knoten in den Adjazenzlisten und dem Array (bzw. der Liste), in dem die Köpfe der Adjazenzlisten verwaltet werden, ergeben.

BFS und **DFS** in Digraphen induzieren einen Wald von ger. Bäumen.

Beispiel:



BFS - Wald



DFS - Wald

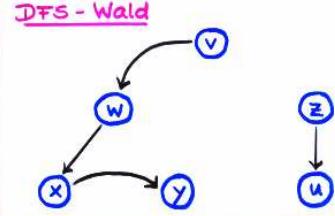


Abbildung 18: BFS- und DFS-Wälder

Implementierungsmöglichkeiten für Bäume. Welche Implementierungsart geeignet ist, hängt wesentlich von der konkreten Anwendung ab. Zu den prinzipiellen Möglichkeiten zählen neben Adjazenzlisten (wie für gewöhnliche Graphen):

- **Pointer-Darstellung (Variante 1)** Ist der Verzweigungsgrad k bekannt, so können die Knoten durch ein Array von Zeigern auf die Söhne dargestellt werden. Häufig wird zusätzlich ein Zeiger auf den Vater verwendet. Siehe Abbildung 19. (In einigen Anwendungen, z.B. UNION-FIND-Wälder (vgl. Kapitel 6), werden sogar ausschließlich „Vater-Verweise“ verwendet.)

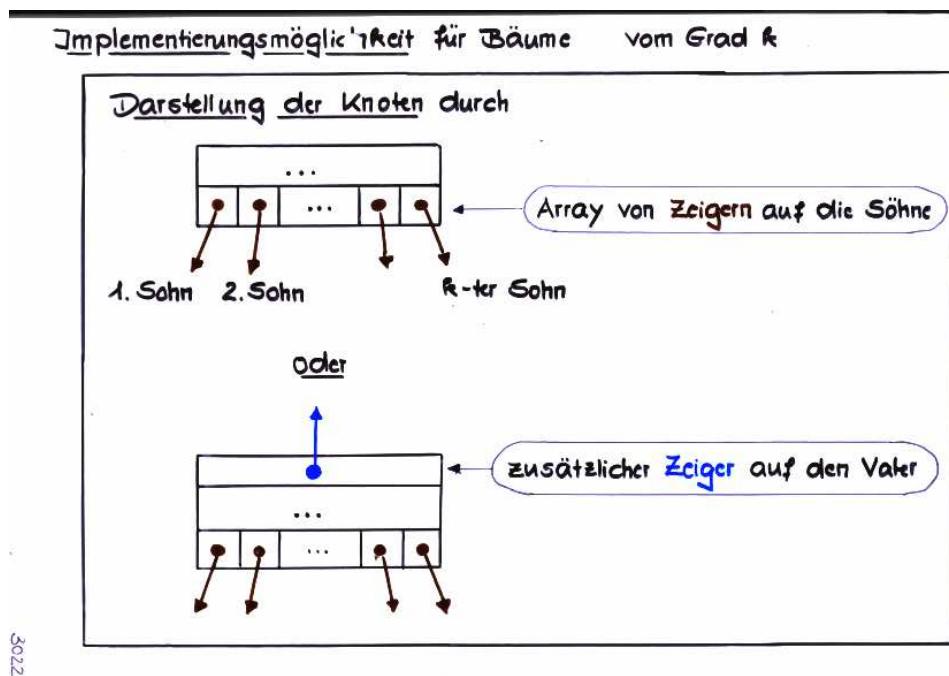


Abbildung 19: Pointer-Darstellung (Variante 1)

- **Pointerdarstellung (Variante 2)** Jeder Knoten wird durch zwei Zeiger dargestellt, wobei der erste Zeiger auf den ersten Sohn des Knotens verweist und der zweite Zeiger auf den rechten Bruder. Siehe Abbildung 20.

Die beiden auf den Pointerdarstellungen Implementierungsalternativen setzen eine *Ordnung* der Söhne eines Knotens voraus, die es erlaubt, vom ersten, zweiten, dritten, ..., k -ten Sohn eines Knotens zu sprechen. Oftmals liegt eine solche Ordnung in natürlicher Weise vor. Sie kann aber auch willkürlich gewählt werden. Der „nächste Bruder“ (manchmal auch „rechter Bruder“ genannt) des i -ten Sohns von v ist dann der $(i + 1)$ -te Sohn von v , soweit existent.

Implementierungsmöglichkeit für Bäume (bei Verzweigungsgrad)

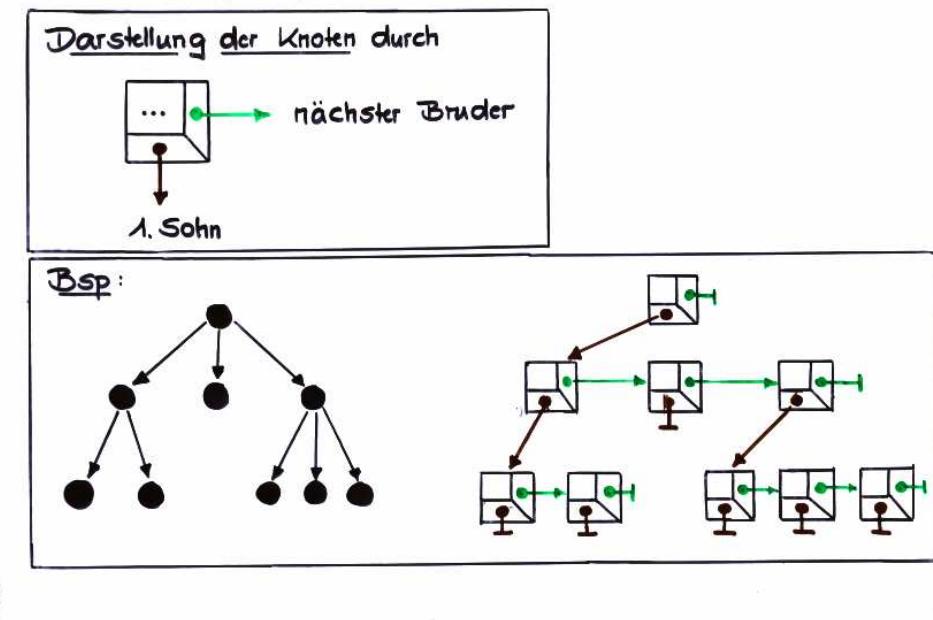


Abbildung 20: Pointer-Darstellung (Variante 2)

Traversierungsmethoden. Die üblichen Traversierungsmethoden für Bäume, in denen von jedem inneren Knoten v auf die Söhne von v zugegriffen werden kann, sind Pre- und Postorder, die beide auf einer Tiefensuche beruhen:

- **Preorder:** erst wird die Wurzel, dann die Söhne der Wurzel (und deren Teilbäume) besucht.
- **Postorder:** erst werden die Söhne (und deren Teilbäume) besucht, dann die Wurzel.

Gegebenenfalls sollten die Söhne in der Ordnung entsprechenden Reihenfolge besucht werden. Man beachte, daß die Preorder-Besuchsreihenfolge genau mit der DFS-Besuchsreihenfolge übereinstimmt. Die Postorder-Besuchsreihenfolge ergibt sich aus der Tiefensuche, wenn den Knoten ihre Besuchsnummer erst bei Entnahme aus dem (Rekursions-)Stack zugeteilt wird.

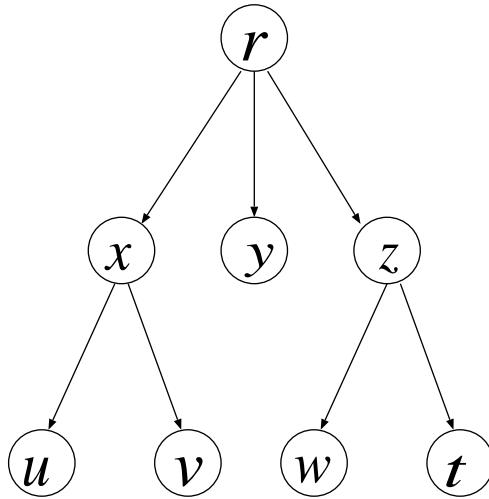
Algorithmus 29 Preorder-Durchlauf $\text{Preorder}(v)$

markiere Knoten v als besucht;
FOR ALL Söhne w von v **DO**
 $\text{Preorder}(w)$
OD

Algorithmus 30 Postorder-Durchlauf $\text{Postorder}(v)$

FOR ALL Söhne w von v **DO**
 $\text{Postorder}(w)$
OD
markiere Knoten v als besucht;

Für den Baum in der Skizze unten nehmen wir an, daß die Reihenfolge der Söhne jedes Knotens von links nach rechts abgelesen werden kann. Dann ist r, x, u, v, y, z, w, t die Preorder-Besuchsreihenfolge. Diese entspricht genau der DFS-Besuchsreihenfolge, wenn die Tiefensuche mit dem Wurzelknoten r gestartet wird. Die Postorder-Besuchsreihenfolge u, v, x, y, w, t, z, r entspricht der Reihenfolge, in der die Rekursionsaufrufe $\text{DFS}(\cdot)$ abgeschlossen werden.



Offenbar ist die Laufzeit eines Preorder- oder Postorderdurchlaufs linear in der Knotenzahl, also $\Theta(n)$, falls der Baum n Knoten hat.

3.2.6 Binärbäume

Bei Binärbäumen spricht man von dem *linken* bzw. *rechten Sohn* eines inneren Knotens anstelle des ersten bzw. zweiten Sohns. In zahlreichen Anwendungen hat man es mit Binärbäumen zu tun, für die für jeden Sohn w eines inneren Knotens v eindeutig feststeht, ob w der linke oder rechte Sohn von v ist. Oftmals kann ein innerer Knoten einen rechten Sohn haben, während kein linker Sohn existiert. Wir verwenden folgende Schreibweisen. Ist v ein innerer Knoten eines Binärbaums \mathcal{T} , so schreiben wir $left(v)$ bzw. $right(v)$ für den linken bzw. rechten Sohn von v (falls existent). Falls v keinen linken Sohn hat, schreiben wir $left(v) = \perp$. Entsprechend steht $right(v) = \perp$ dafür, daß v keinen rechten Sohn hat. Der *linke Teilbaum* eines Knotens v ist der Teilbaum mit der Wurzel $left(v)$. Ist $left(v) = \perp$, so ist der linke Teilbaum von v der leere Baum. In analoger Weise ist der *rechte Teilbaum* von v definiert.

Implementierungsmöglichkeiten. Die für beliebige Bäume erwähnte Pointerdarstellung (Variante 1) ist die gängigste Darstellungsvariante für Binärbäume (vgl. Abbildung 21). Wie für Listen kann diese Darstellungsform auch mithilfe von Arrays (statt Zeigern) realisiert werden. Hierzu verwendet man drei Arrays, die jeweils die Schlüsselwerte nebst eventueller Zusatzinformationen über die Knoten sowie die Positionen des linken bzw. rechten Sohns im Array angeben. Dazu gleichwertig ist eine Darstellung durch ein Array, dessen Elemente aus drei Komponenten „key“, „LSohn“ und „RSohn“ bestehen. Siehe Abbildung 22. Zusätzlich muß eine Variable bereitgestellt werden, welche die Position der Wurzel angibt.

Gängige Implementierung von Binärbaum n:

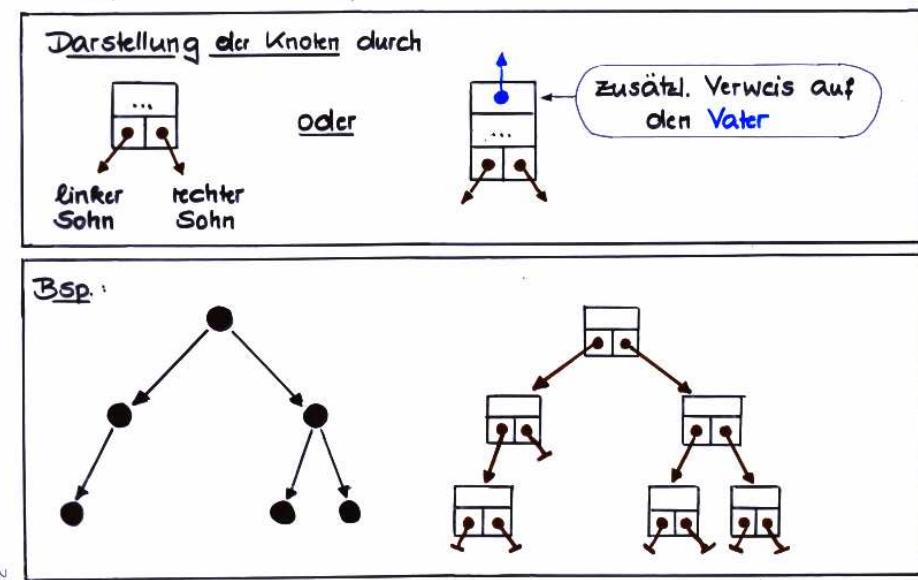


Abbildung 21: Gängige Implementierungsmöglichkeit für Binäräume

Darstellung von Binäräumen durch 3 Arrays:

	1	2	3	4	5	6
Key	15	20	17	2	34	9
LSohn	⊥	⊥	6	⊥	⊥	4
RSohn	⊥	5	2	⊥	⊥	1

Wurzel = 3

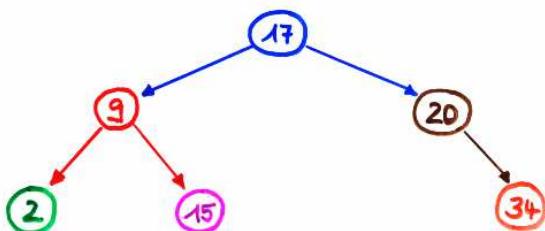


Abbildung 22: Beispiel zur Darstellung eines Binärbaums durch drei Arrays

Eine weitere Möglichkeit zur Darstellung eines Binärbaums besteht in der Verwendung eines Arrays ohne explizite Darstellung der Vater-Sohn-Verweise. Die Idee besteht darin, die Informationen zu dem linken bzw. rechten Sohn des an der i -ten Array-Komponente stehenden Knotens in der $2i$ -ten bzw. $(2i + 1)$ -ten Array-Komponente abzulegen (sofern existent). Ein Beispiel ist in Abbildung 23 zu finden. Diese Implementierungsmöglichkeit ist für weitgehend vollständige Binäräume, deren Knotenzahl a priori feststeht, besonders platzeffizient, da sie ohne explizite Speicherung von Verweisen auf die Söhne eines Knotens auskommt.

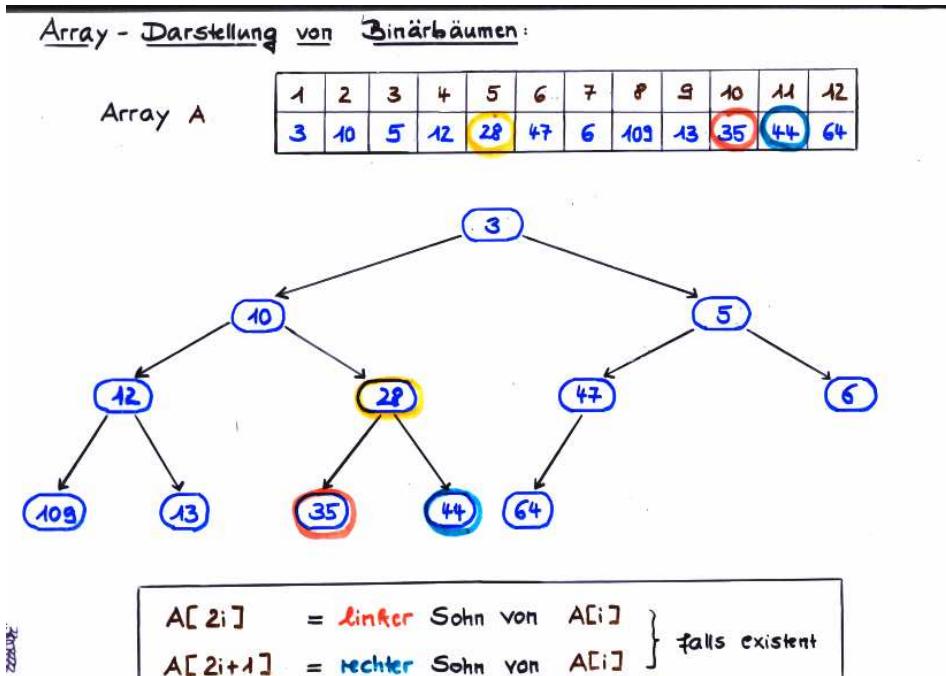


Abbildung 23: Beispiel zur Darstellung eines Binärbaums mit nur einem Array

Traversierungsmöglichkeiten für Binäräume :

- **Preorder (WLR):** besuche erst die Wurzel, dann den linken Teilbaum, dann den rechten Teilbaum. Siehe Algorithmus 31.
- **Inorder (LWR):** besuche erst den linken Teilbaum, dann die Wurzel, dann den rechten Teilbaum. Siehe Algorithmus 32.
- **Postorder (LRW):** besuche erst den linken Teilbaum, dann den rechten Teilbaum, dann die Wurzel. Siehe Algorithmus 33.

In manchen Anwendungen kann auch eine auf „rechts vor links“ beruhende Durchlaufstrategie (WRL, RLW oder RWL) nützlich sein.

Algorithmus 31 *Preorder(v)*

Markiere v als besucht.

IF $\text{left}(v) \neq \perp$ **THEN** *Preorder(left(v)) FI;*
IF $\text{right}(v) \neq \perp$ **THEN** *Preorder(right(v)) FI*

Algorithmus 32 *Inorder(v)*

IF $\text{left}(v) \neq \perp$ **THEN** *Inorder(left(v)) FI;*

Markiere v als besucht.

IF $\text{right}(v) \neq \perp$ **THEN** *Inorder(right(v)) FI*

Algorithmus 33 *Postorder(v)*

IF $\text{left}(v) \neq \perp$ **THEN** *Postorder(left(v)) FI;*

IF $\text{right}(v) \neq \perp$ **THEN** *Postorder(right(v)) FI*

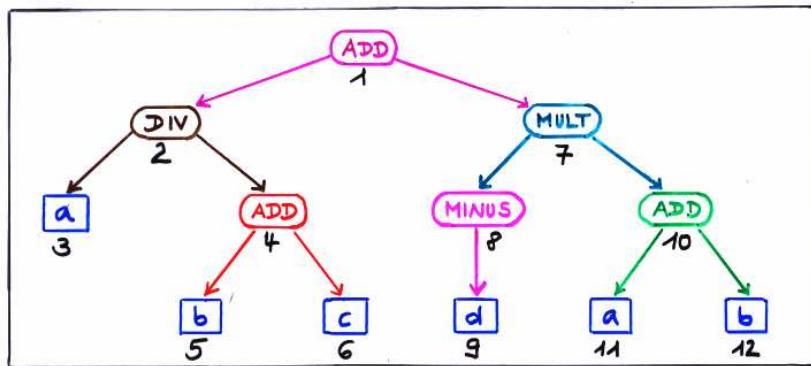
Markiere v als besucht.

Als Anwendungsbeispiel für die Preorder-Durchlaufstrategie erwähnen wir die Umwandlung eines arithmetischen Ausdrucks in Präfix-Darstellung (vgl. Abbildung 24). Der Ausgangspunkt ist der Syntaxbaum des betreffenden Ausdrucks (dessen Knoten mit den Operatoren markiert, die Blätter mit den Operanden beschriftet sind). Traversiert man diesen nach dem Preorder-Prinzip und gibt jeweils die Knotenbeschriftung aus, so ergibt sich die Präfix-Darstellung des Ausdrucks. Analoges gilt für die Inorder, welche die Infix-Darstellung generiert, und die Postorder, welche den Ausdruck in Postfix-Notation ausgibt.

Als weiteres Beispiel für den Einsatz von Binärbäumen erwähnen wir Syntaxbäume für aussagenlogische Formeln. Diese entsprechen der Schaltbilddarstellung.

Syntaxbäume für arithmetische Ausdrücke

z.B. $(a / (b + c)) + (-d) * (a + b)$



Preorder-Durchlauf ergibt Präfix-Darstellung:

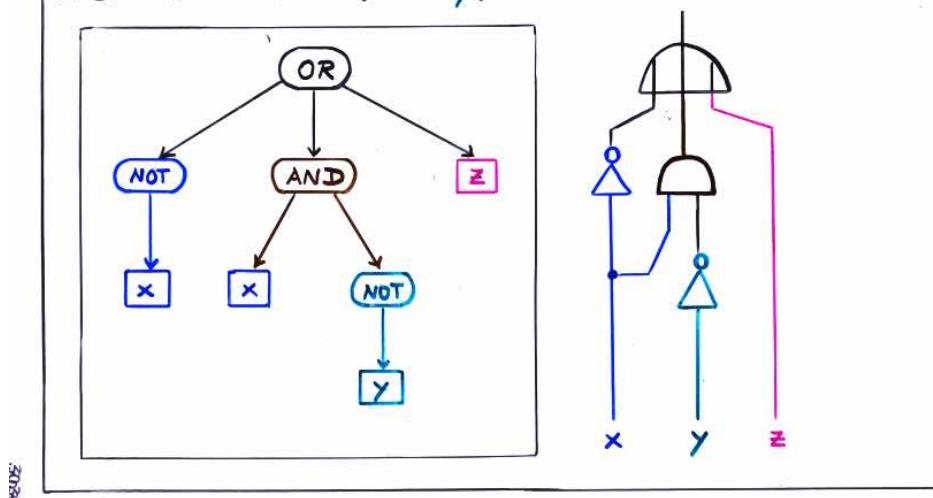
$+ / a + b c * - d + a b$

Abbildung 24: Präfixdarstellung eines arithmetischen Ausdrucks

Syntaxbäume für aussagenlogische Formeln

... sind Schaltbilder

z.B. $\alpha = \neg x \vee (x \wedge \neg y) \vee z$

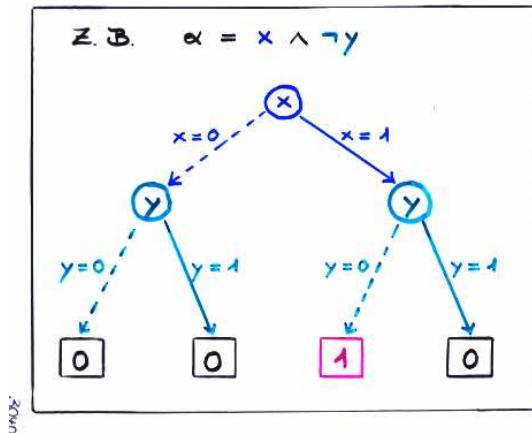


In zahlreichen Varianten spielen Entscheidungsbäume in der Informatik eine große Rolle. Zunächst erwähnen wir Entscheidungsbäume für aussagenlogische Formeln. Diese lassen sich zu binären Entscheidungsgraphen (kurz BDDs für „binary decision diagrams“)

verallgemeinern, die sich insbesondere für die Analyse von VLSI-Schaltkreisen bewährt haben.

Entscheidungsbäume für aussagenlogische Formeln

- * vollständige Binäräbäume
- * Pfade stehen für die möglichen Belegungen
- * Blätter stehen für die Wahrheitswerte

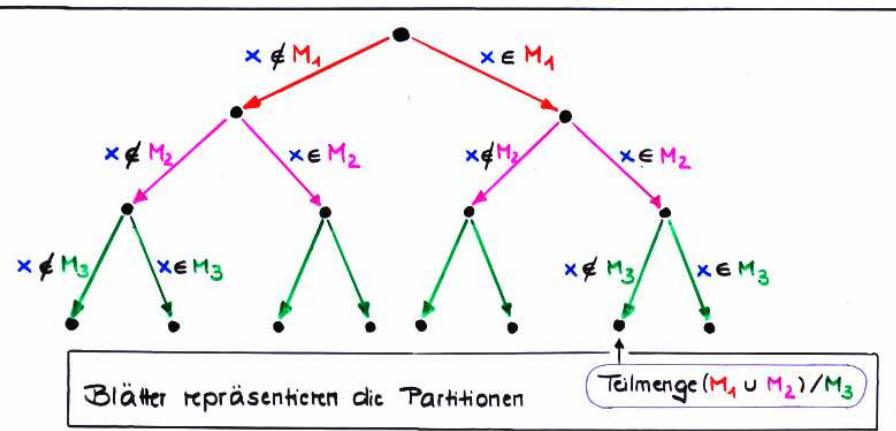
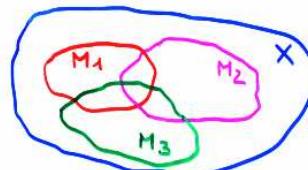


Ein weiteres Beispiel für die Einsatzmöglichkeiten von Entscheidungsbäumen sind Binärbaum, deren Blätter für Äquivalenzklassen einer festen Grundmenge stehen.

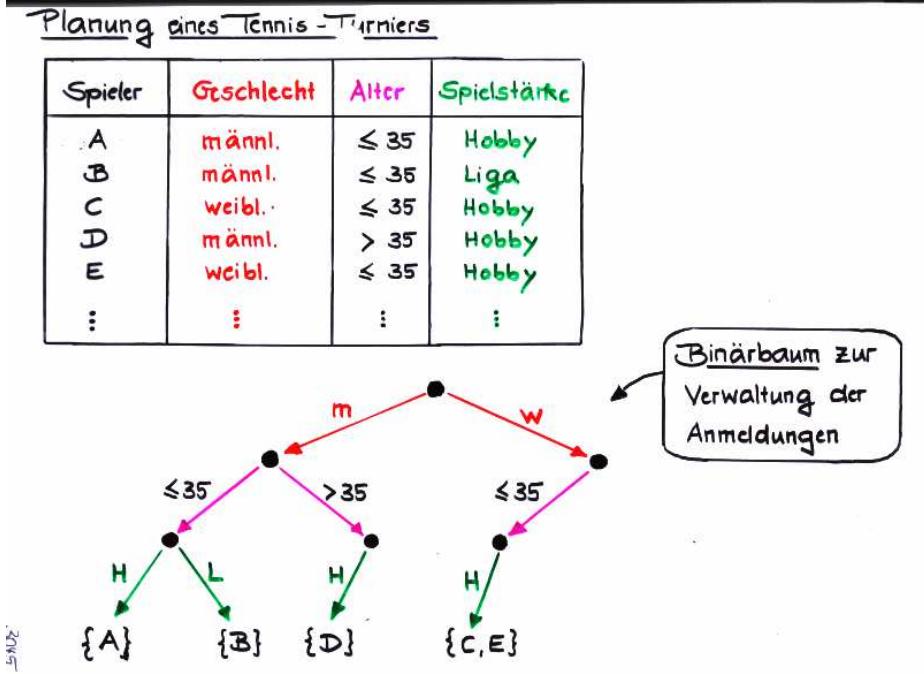
Entscheidungsbäume zum Bilden von Äquivalenzklassen

Grundmenge X

Teilmengen M_1, M_2, M_3

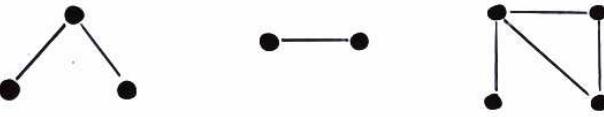


Derartige Entscheidungsbäume können als Implementierungsgrundlage zum Auffinden der Äquivalenzklassen dienen. Als Beispiel erwähnen wir ein Turnierplanungssystem, in dem die gemeldeten Spieler und Spielerinnen gemäß diverser Kriterien (z.B. Geschlecht, Spielstärke) in Spielklassen eingeteilt werden sollen.

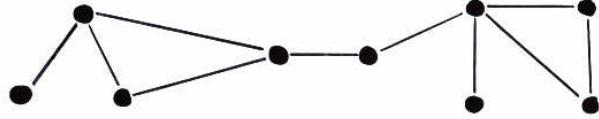


3.2.7 Ungerichtete Bäume

Definition 3.2.21 (Zusammenhängend, Zusammenhangskomponente). Sei $G = (V, E)$ ein ungerichteter Graph und $C \subseteq V$. C heißt zusammenhängend, falls je zwei $v, w \in C$ voneinander erreichbar sind (d.h. $w \in Post^*(v)$ und $v \in Post^*(w)$). C heißt Zusammenhangskomponente von G , falls C eine nicht leere maximale zusammenhängende Knotenmenge ist. Maximalität bedeutet dabei, daß C in keiner anderen zusammenhängenden Teilmenge C' von V enthalten ist. G heißt zusammenhängend, falls V zusammenhängend ist. \square



unzusammenhängender Graph mit 3 Zush. Komponenten



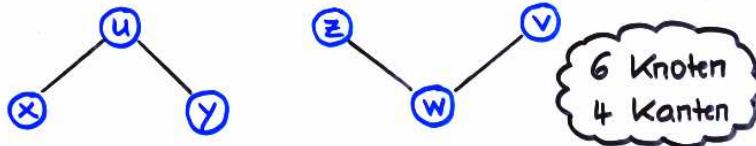
zusammenhängend
(1 Zush. Komponente)

Offenbar gilt: Die Zusammenhangskomponenten von G sind die Äquivalenzklassen von V bzgl. der Erreichbarkeits-Äquivalenzrelation $v \equiv w$ genau dann, wenn $\text{Post}^*(v) = \text{Post}^*(w)$. Insbesondere zerfällt G in paarweise disjunkte Zusammenhangskomponenten. D.h. sind C_1, \dots, C_r die paarweise verschiedenen Äquivalenzklassen bzgl. \equiv , so ist

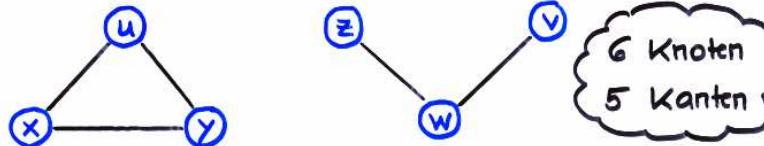
$$V = \bigcup_{1 \leq i \leq r} C_i, \quad E = \bigcup_{1 \leq i \leq r} E_i,$$

wobei $E_i = E \cap (C_i \times C_i)$. Weiter gilt (für $V \neq \emptyset$): G ist genau dann zusammenhängend, wenn es genau eine Zusammenhangskomponente gibt.

Bsp.: unzust. , azyklischer Graph



Bsp.: unzust., zyklischer Graph



N
G
v
w
z
x
y
u

Lemma 3.2.22. Sei $G = (V, E)$ ein ungerichteter Graph mit $|V| = n$. Dann gilt:

- (a) $n \geq 2$ und $|E| < n - 1 \implies G$ ist nicht zusammenhängend.
- (b) $E \geq n \geq 3 \implies G$ ist zyklisch.

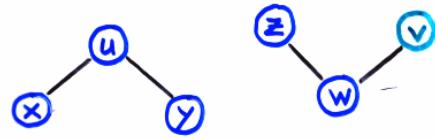
Insbesondere gilt: Ist G zusammenhängend und azyklisch, so ist $|E| = n - 1$.

Beweis. siehe Vorlesung.

Zum Beweis von Lemma 6.2.1, Teil (a):

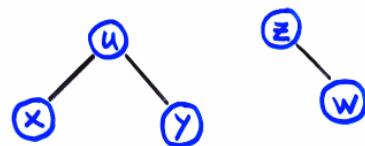
Entfernen eines Knotens (samt der zugehörigen Kanten)

Bsp.: Graph G



$$u \notin \text{Post}_G^*(w) \\ \text{Post}_G^*(v)$$

Graph G'



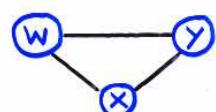
$$u \notin \text{Post}_{G'}^*(w)$$

Zum Beweis von Lemma 6.2.1, Teil (b):

Bsp. zum Entfernen von Knoten (Bew. von Lemma 3.2. (b))

1. Fall: $\text{Post}(v) = \emptyset$

Graph G



Graph G'



2. Fall: $|\text{Post}(v)| = 1$

Graph G

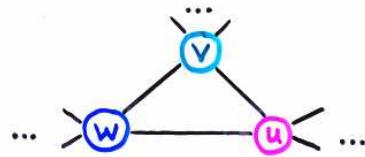


Graph G'



Zum Beweis von Lemma 3.2. (b):

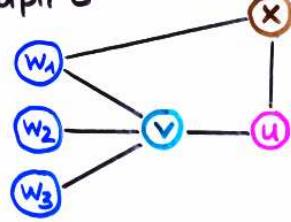
Fall 3.1: es gibt $w, u \in \text{Post}(v)$ mit $(w, u) \in E$



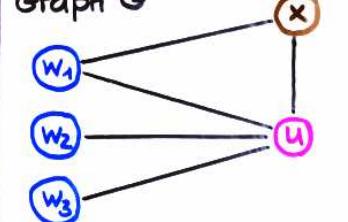
Zyklus v, w, u, v

Fall 3.2:

Graph G



Graph G'



□

Ist $C \subseteq V$, dann heißt $G_C = (C, E_C)$ mit $E_C = E \cap (C \times C)$ der durch C induzierte Teilgraph von G . Offenbar gilt für jede Zusammenhangskomponente C von G : G_C ist zusammenhängend. Somit ist $|C| - 1 \leq |E_C|$. Mit Worten: Jede Zusammenhangskomponente enthält mindestens $|C| - 1$ Kanten.

Satz 3.2.23. Sei $G = (V, E)$ ein ungerichteter Graph mit $|V| = n \geq 3$. Dann sind folgende Aussagen äquivalent:

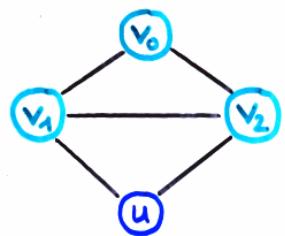
- (a) G ist zusammenhängend und zyklenfrei
- (b) G ist azyklisch und $|E| = n - 1$
- (c) G ist zusammenhängend und $|E| = n - 1$
- (d) Zu jedem Knotenpaar (v, w) gibt es genau einen einfachen Pfad von v nach w .

Beweis. siehe Vorlesung.

Zum Beweis von Satz 6.2.5, Teil (c) \Rightarrow (a):

Identifikation von Knoten auf einer Zyklus

Bsp.: Graph G

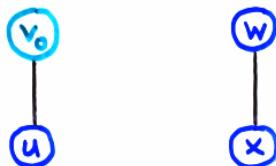


Zyklus v_0, v_1, v_2, v_0

$$u \in \text{Post}_G^*(v_0)$$

$$w \notin \text{Post}_G^*(v_0)$$

Graph G'



$$u \in \text{Post}_{G'}^*(v_0)$$

$$w \notin \text{Post}_{G'}^*(v_0)$$

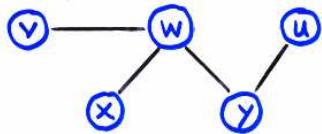
Skizz.

□

Definition 3.2.24 (Ungerichteter Baum). Jeder ungerichtete Graph, für den die (oder eine der) Aussagen (a)-(d) zutreffen, wird *ungerichteter Baum* genannt.²⁶ □

²⁶In Satz 6.2.5 wir vorausgesetzt, daß die Knotenzahl $n \geq 3$ ist. Aus formalen Gründen ist es oftmals sinnvoll, jeden azyklischen und zusammenhängenden ungerichteten Graphen (also auch den leeren Graphen, jeden ungerichteten Graphen mit einem Knoten sowie jeden ungerichteten Graphen mit zwei Knoten und einer Kante) als ungerichteten Baum zu bezeichnen.

Bsp.: Ungerichteter Baum

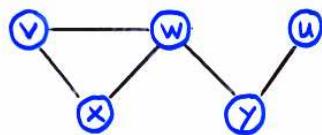


azyklisch & zush.

5 Knoten

4 Kanten

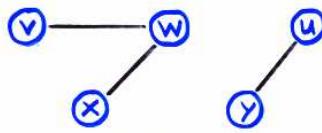
Bsp.: Keine ungerichteten Bäume



zyklisch & zush.

5 Knoten

5 Kanten



azyklisch & unzsh.

5 Knoten

3 Kanten

Satz 3.2.25. Ist die Adjazenzlisten-Darstellung eines ungerichteten Graphen $G = (V, E)$ gegeben, so lässt sich in $\mathcal{O}(n)$ Schritten prüfen, ob G azyklisch ist. Entsprechendes gilt für die Frage, ob G ein ungerichteter Baum ist.

Zyklen test in ungerichteten Graphen. Wir verwenden eine Modifikation der üblichen Travierungsmethoden (DFS oder BFS), wobei wir für jeden besuchten Knoten u die Information, über welche Kante besucht wird, speichern. Etwa

$$\text{father}(u) = w \text{ gdw. } u \text{ wird über die Kante } (w, u) \text{ besucht}$$

(wobei wir für diejenigen Knoten v , für die die Traversierung gestartet wird, einen Spezialwert verwenden müssen, z.B. $\text{father}(v) = \perp$) Sobald wir bei der Expansion von v auf einen bereits besuchten direkten Nachfolger w von v stoßen, so daß $\text{father}(v) \neq w$, dann ist der vorliegende Graph zyklisch. Je nach dem ob w bei der Tiefen suche vor oder nach v besucht ist, ist entweder der Pfad w_0, w_1, \dots, w_r, v mit $w_0 = v$, $w_{i+1} = \text{father}(w_i)$, $i = 0, 1, \dots, r$ und $w_r = v$ oder der Pfad w_0, w_1, \dots, w_r, w mit $w_0 = w$, $w_r = v$ und $\text{father}(w_{i+1}) = w_i$, $i = 0, 1, \dots, r - 1$, ein einfacher Zyklus.

Die Funktion Zyklus(v)

(* Tiefensuche (gestartet mit v) mit Zyklustest *)

```

Visited := Visited ∪ {v},
FOR ALL w ∈ Post(v) DO
  IF father(v) ≠ w ∈ Visited THEN return "true"   FI
  IF w ∉ Visited
    THEN father(w) := v,
    IF zyklus(w) THEN return "true"   FI
  FI
  (* Kantenzähler := Kantenzähler + 1 *)
  (* es gilt stets Kantenzähler ≤ 2 * Knotenzahl *)
OD
Return "false".

```

Zyklustest in ungerichteten Graphen, z.B. mittels DFS

```

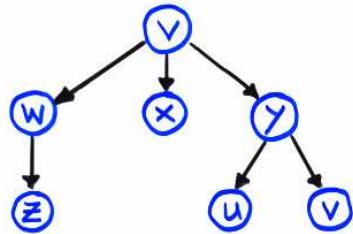
Visited := ∅,          (* Kantenzähler := 0 *)
FOR ALL v ∈ V DO
  IF v ∉ Visited
    THEN father(v) := ⊥
    IF zyklus(v)
      THEN return "Zyklus gefunden"
    FI
  FI
OD
Return "der Graph ist azyklisch."

```

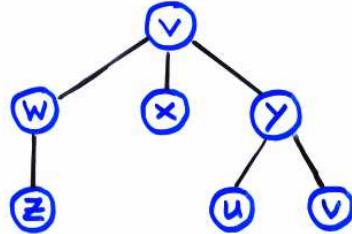
DFS mit Zyklustest

Der Zusammenhang zwischen gerichteten und ungerichteten Bäumen ist wie folgt. Jedem gerichteten Baum liegt ein ungerichteter Baum zugrunde.

Jedem gerichteten Baum liegt ein ungerichteter Baum zugrunde.



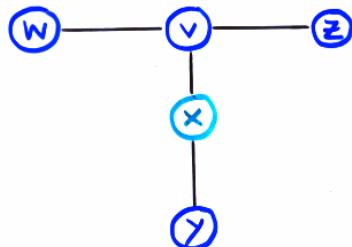
gerichteter Baum



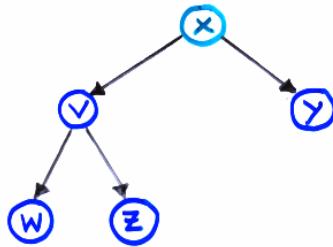
ungerichteter Baum

Umgekehrt kann man in jedem ungerichteten Baum die Kanten so richten, daß ein gerichteter Baum entsteht. Dies ist sogar dann möglich, wenn man den Wurzelknoten fest vorgibt.

In jedem ungerichteten Baum können die Kanten so gerichtet werden, daß ein gerichteter Baum entsteht.



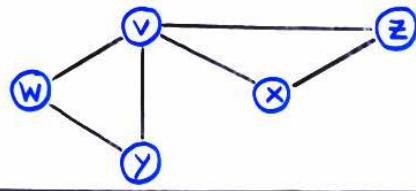
ungerichteter Baum



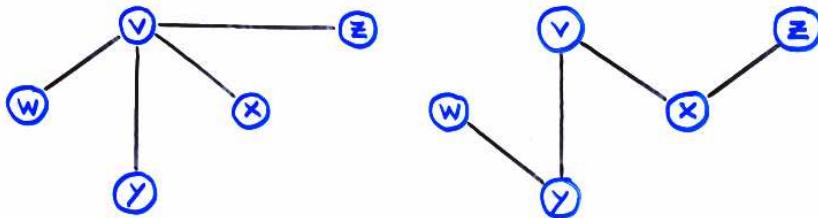
gerichteter Baum

Definition 3.2.26 (Aufspannender Baum). Sei $G = (V, E)$ ein ungerichteter Graph. Ein *aufspannender Baum* für G ist ein ungerichteter Baum $\mathcal{T} = (V, E_{\mathcal{T}})$, wobei $E_{\mathcal{T}} \subseteq E$. \square

Ungerichteter Graph



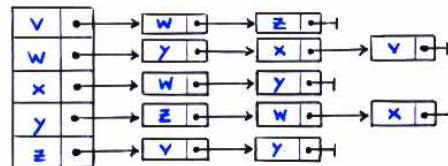
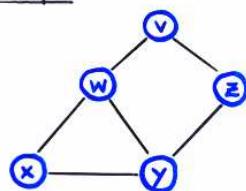
Aufspannende Bäume, z.B.



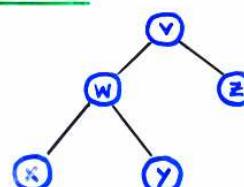
Offenbar gilt: Es gibt genau dann einen aufspannenden Baum für G , wenn G zusammenhängend ist. Ein solcher lässt sich mit DFS oder BFS bestimmen.

BFS und **DFS** in ungerichteten zusammenhängenden Graphen liefern aufspannende Bäume.

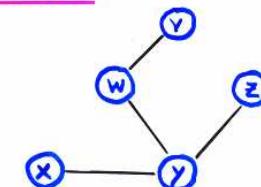
Beispiel:



BFS - Baum

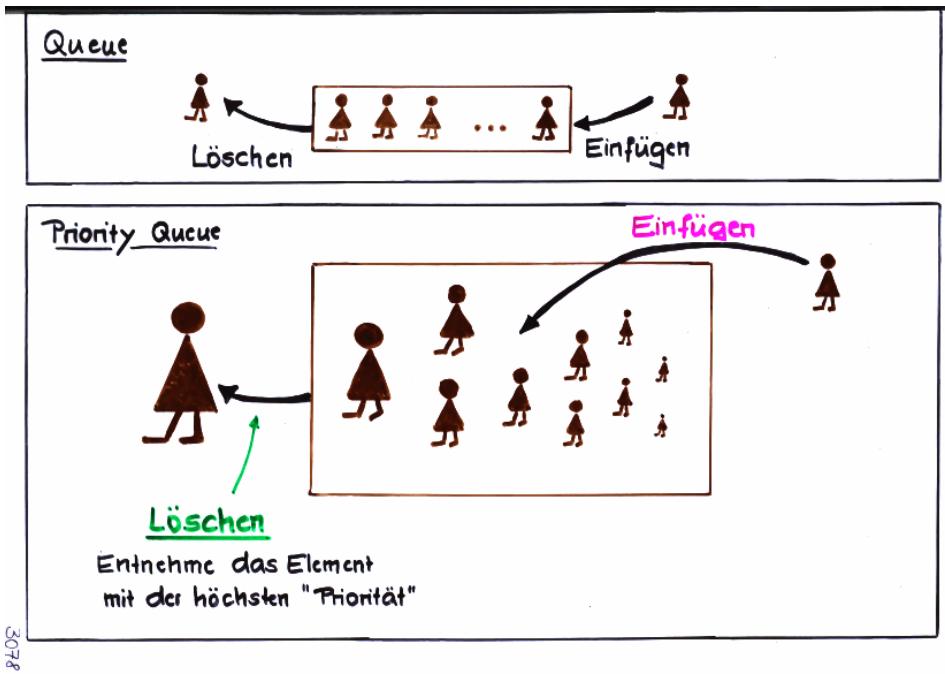


DFS - Baum



3.3 Priority Queues (Heaps)

Priority Queues, auch *Heaps* oder *Halden* genannt, sind eine Datenstruktur zur Verwaltung von (Multi-)Mengen, in denen Elemente unterschiedlicher Priorität dargestellt werden. Zugriffe sind stets nur auf Elemente (bzw. das Element) der höchsten Priorität möglich.



Die „Prioritäten“ werden anhand gewisser Schlüsselwerte eines geordneten Bereichs (z.B. Zahlen mit der natürlichen Ordnung) vergeben. Wir erläutern hier das Konzept von *Minimumsheaps*, in denen das Element mit dem kleinsten Schlüsselwert als das Element mit der höchsten Priorität angesehen wird. Zur Vereinfachung nehmen wir an, daß die Schlüsselwerte (reelle oder ganze) Zahlen sind und identifizieren die Elemente der Priority Queue mit ihren Schlüsselwerten. In analoger Weise können Maximumsheaps oder andere Arten von Schlüsselwerten behandelt werden.

3.3.1 Minimumsheaps

Unterstützt werden folgende Operationen:

- $Insert(Q, x)$: Einfügen eines Elements mit Schlüsselwert x in Q .
- $ExtractMin(Q)$: liefert das (ein) Element mit dem kleinsten Schlüsselwert in Q und entfernt dieses aus Q .

Ferner wird die Abfrage, ob $Q = \emptyset$ unterstützt. Die Operation $ExtractMin(Q)$ ist nur für $Q \neq \emptyset$ zulässig. Für Maximumsheaps ist $ExtractMin(Q)$ durch $ExtractMax(Q)$ zu ersetzen, welches ein Element mit maximalem Schlüsselwert zurückgibt und dieses entfernt.

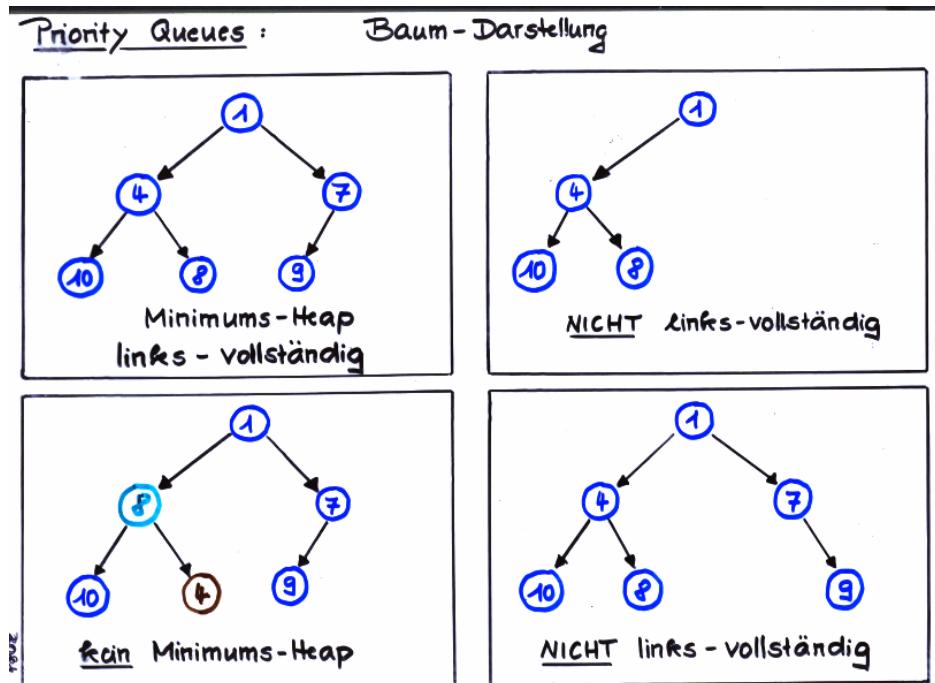
Baumdarstellung von Minimumsheaps. Sei \mathcal{T} ein Binärbaum, in dem jeder Knoten v mit einem Schlüsselwert $key(v)$ versehen ist. \mathcal{T} hat die *Minimumsheap-Eigenschaft*, falls folgende beiden Bedingungen erfüllt sind:

- (1) Ist v ein innerer Knoten v von \mathcal{T} und w ein Sohn von v , so ist $key(v) \leq key(w)$.
- (2) \mathcal{T} ist *linksvollständig*.

Linksvollständigkeit eines Binärbaums \mathcal{T} ist eine etwas abgeschwächte Form der Vollständigkeit, die für Bäume mit beliebiger Knotenzahl zutreffend sein kann. Formal bedeutet Linksvollständigkeit eines Binärbaums, daß folgende beiden Eigenschaften erfüllt sind. Ist h die Höhe von \mathcal{T} , so gilt:

- \mathcal{T} ist bis zur Tiefe $h - 1$ vollständig, d.h. jeder Knoten der Tiefe $i < h - 1$ ist ein innerer Knoten mit genau zwei Söhnen.
- Ist v ein Knoten der Tiefe $h - 1$ mit höchstens einem Sohn, dann hat v keinen rechten Sohn und alle Knoten der Tiefe $h - 1$, die rechts von v liegen, sind Blätter.

Insbesondere gibt es kein Blatt der Tiefe h , das rechts von v liegt. Die Rechts-Links-Beziehung der Knoten von \mathcal{T} ist wie folgt definiert: Seien v, w Knoten mit $v \neq w$. w liegt *rechts von v* , wenn es einen Knoten u gibt, so daß v im linken Teilbaum von u und w im rechten Teilbaum von u liegt.



Man beachte, daß die Heapbedingung (1) durch die Forderung

$$key(v_0) \leq key(v_1) \leq \dots \leq key(v_r)$$

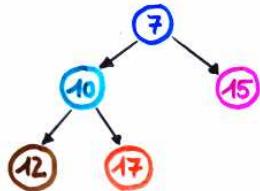
für jeden Pfad v_0, v_1, \dots, v_r in \mathcal{T} ersetzt werden kann.

Ein linksvollständiger Minimumsheap mit k Knoten kann – wie bereits in Abschnitt 3.2.6 beschrieben – mit einem Array $Q[1 \dots n]$ der Länge $n \geq k$ implementiert werden.

- $Q[1]$: Wurzel
- $Q[2i]$: linker Sohn des an Position i gespeicherten Knotens (falls $2i \leq k$)
- $Q[2i + 1]$: rechter Sohn des an Position i gespeicherten Knotens (falls $2i + 1 \leq k$)

Diese Darstellungsform ist aufgrund der Linksvollständigkeit recht kompakt und daher für Heaps üblich.

Array - Darstellung eines Minimums - Heaps:



i	1	2	3	4	5	6	7
Q[i]	7	10	15	12	17	—	—

Knotenzahl $k = 5$

z.B. linker Sohn von $10 = Q[2]$ ist $Q[4] = 12$

rechter Sohn von $10 = Q[2]$ ist $Q[5] = 17$

tipp

Abbildung 25: Beispiel zur üblichen Array-Darstellung eines Heaps

Einfügen in einem Minimumsheap. Die Operation $Insert(Q, x)$ arbeitet nach folgendem Schema: Füge x als rechtestes Blatt ein und überprüfe bzw. korrigiere in Bottom-Up-Manier die Heap-Bedingung. Hierzu läuft man rückwärts auf dem Pfad von dem neu eingefügten Blatt zur Wurzel und vertauscht gegebenenfalls die Schlüsselwerte des aktuellen Knotens v und des Vaters von v . Sobald ein Knoten v erreicht ist, für den keine Vertauschung notwendig ist, kann das Verfahren abgebrochen werden. Siehe Algorithmus 34 sowie die Folie in Abbildung 26.

Entnahme des kleinsten Elements. Die Ausführung der Operation $ExtractMin(Q)$ (siehe Algorithmen 35 sowie die Folie in Abbildung 27) verläuft wie folgt: Im ersten Schritt wird die Markierung der Wurzel ausgegeben. Wir markieren dann (vorläufig) die

Algorithmus 34 *INSERT(Q,x)*

(* Sei $k = |Q|$ die Anzahl an Elementen in Q . *)

$i := k + 1;$

IF $i > n$ **THEN**

 return „Sorry. Keine Plätze frei.“.

ELSE

WHILE $i > 1$ und $Q[i \text{ div } 2] > x$ **DO**

$Q[i] := Q[i \text{ div } 2];$

$i := i \text{ div } 2$

OD

$Q[i] := x;$

$k := k + 1$

FI

Wurzel mit dem Schlüsselwert y des letzten Blatts v und löschen v . Nun lässt man y an die korrekte Position „sinken“. Der Sinkvorgang ist in Algorithmus 36 mit der Prozedur *Heapify* beschrieben.

Algorithmus 35 *EXTRACT_MIN(Q)*

(* Sei $k = |Q|$ die Anzahl an Elementen in Q . *)

IF $k = 0$ **THEN**

 return „Sorry. Keine Elemente in Q .“

ELSE

$min := Q[1];$

$Q[1] := Q[k];$

$k := k - 1;$

 Lasse das neue Wurzelement an die richtige Position sinken. (* *Heapify(1)* *)

 return min

FI

Kosten der Heapoperationen. Die oben skizzierten Verfahren zum Einfügen (Algorithmus 34) und Löschen (Algorithmus 35) haben offenbar die Laufzeit $\Theta(\log k)$, wobei k die Anzahl an Elementen im Heap ist. Die Begründung der logarithmischen Laufzeit resultiert aus der Beobachtung, dass aufgrund der Linksvollständigkeit die Elementanzahl k zwischen 2^h und $2^{h+1} - 1$ liegt, wobei h die Höhe des Heaps ist. Die obere Schranke $2^{h+1} - 1$ gilt in jedem Binärbaum der Höhe h . Die untere Schranke 2^h erklärt sich aus der Vollständigkeit bis zur Tiefe $h - 1$. Also ist $h = \Theta(\log k)$. Ferner beruhen der Einfüge- und Löschalgorithmus jeweils auf einer Traversierung des Baums entlang *eines* Pfads beginnend in der Wurzel bzw. auf einer Rückverfolgung eines Pfads von einem Blatt in Richtung Wurzel. In beiden Fällen sind die Kosten durch die Höhe des Heaps beschränkt.

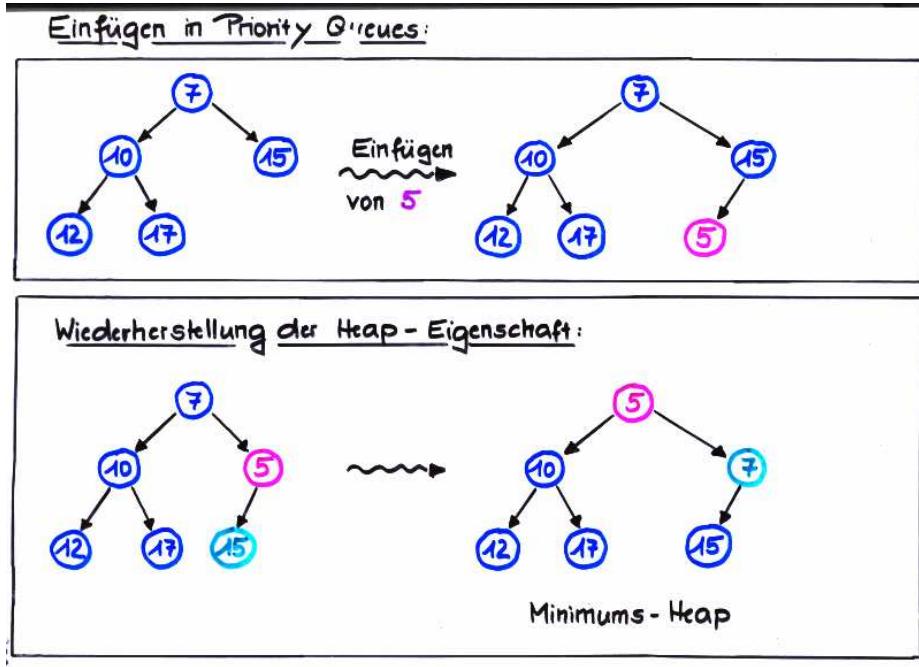


Abbildung 26: Beispiel zum Einfügen in einen Heap

3.3.2 Heapsort

Heapsort ist ein internes Sortierverfahren mit Vergleichen, das wie Mergesort die Laufzeit $\Theta(n \log n)$ im schlimmsten und mittleren Fall hat. Die Grundidee basiert auf dem folgenden Schema. Sei x_1, \dots, x_n die zu sortierende Zahlenfolge.

1. Erstelle einen Minimumsheap für die Zahlen x_1, \dots, x_n .
2. Gib sukzessive das kleinste Element aus.

Die Grundidee des Verfahrens ist in Algorithmus 37 formalisiert.

Der erste Schritt kann zwar durch sukzessives Einfügen der Elemente x_1, \dots, x_n in den anfangs leeren Heap vollzogen werden, jedoch gibt es eine effizientere Methode für den Heapaufbau, die lediglich die Laufzeit $\Theta(n)$ anstelle von $\Theta(n \log n)$ für das n -fache Anwenden der Operation $Insert(Q, x_i)$ hat. Dieses Verfahren beruht auf folgendem Schema:

1. Setze $m = \lfloor n/2 \rfloor$ und geniere Blätter x_{m+1}, \dots, x_n .
2. Für $i = m, m-1, \dots, 1$:
 - Füge x_i als (vorläufigen) Vater von x_{2i} und x_{2i+1} ein.
 - Lasse x_i an die richtige Position sinken (Algorithmus 36).

Algorithmus 36 *Heapify(i)*

(* Sei $k = |Q|$ die Anzahl an Elementen in Q . *)

IF $2i \leq k$ **THEN**

IF $2i + 1 \leq k$ **THEN**

$$x := \min\{Q[i], Q[2i], Q[2i+1]\}; j := \begin{cases} i & : \text{falls } Q[i] = x \\ 2i & : \text{falls } Q[2i] = x < Q[i] \\ 2i+1 & : \text{sonst} \end{cases}$$

ELSE

$$x := \min\{Q[i], Q[2i]\}; j := \begin{cases} i & : \text{falls } Q[i] = x \\ 2i & : \text{sonst} \end{cases}$$

FI

IF $i \neq j$ **THEN**

vertausche $Q[i]$ und $Q[j]$;

Heapify(j)

FI

FI

Algorithmus 37 Heapsort (Idee)

(* Eingabe: Zahlenfolge x_1, \dots, x_n *)

Erstelle einen Minimumsheap Q für x_1, \dots, x_n ;

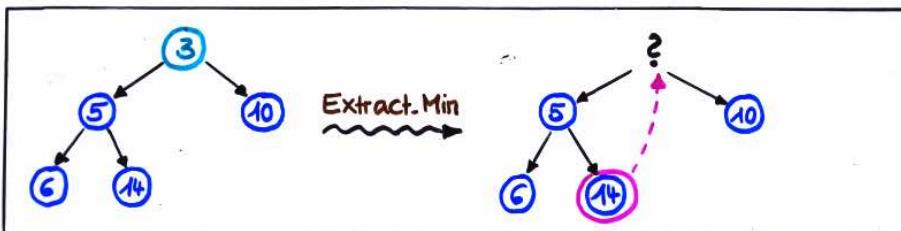
WHILE $Q \neq \emptyset$ **DO**

$x := ExtractMin(Q)$;

gib x aus

OD

"Löschen" in Minimums-Heaps : Entnahme des Kleinsten Elements



Wiederherstellen eines Minimums-Heaps:

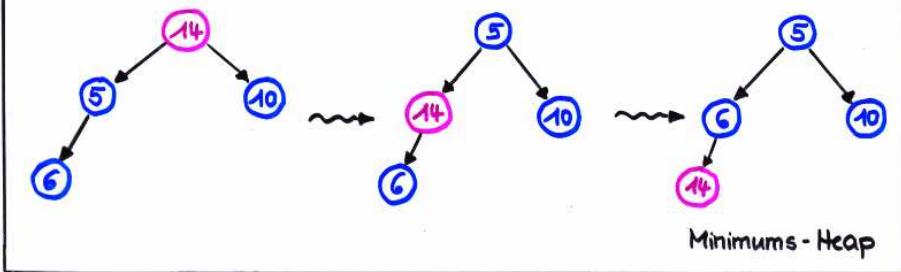


Abbildung 27: Beispiel zur Operation *ExtractMin*

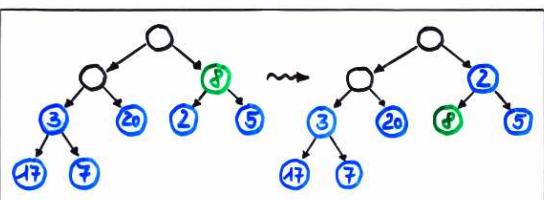
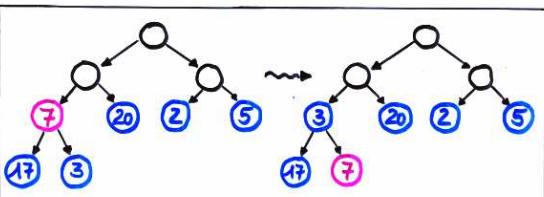
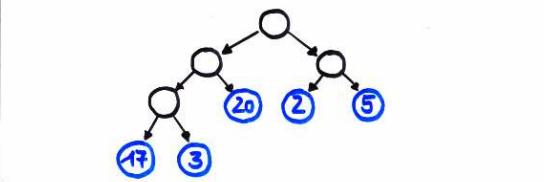
Man beachte, daß von der Folgenlänge n auf die Struktur des Heaps und somit auf Anzahl $n - m$ an Blättern des Baums geschlossen werden kann. Für die genannte Array-Darstellung des Heaps reduziert sich der erste Schritt auf die Zuweisungen $Q[j] := x_j$ für $j = m + 1, \dots, n$.

Folgende Folien skizzieren die Arbeitsweise von Heapsort:

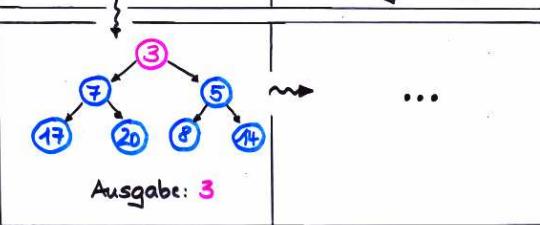
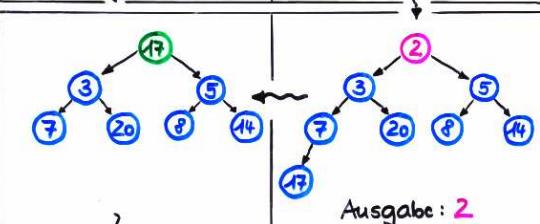
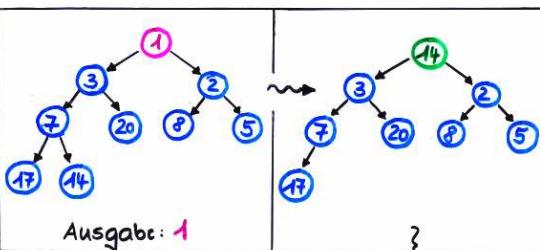
Heapsort: der Heap-Aufbau

Zahlenfolge 1 14 8 7 20 2 5 17 3

Generieren der Blätter:



Heapsort: der Heap-Abbau



...

Analyse des Heapaufbaus. Sei n die Anzahl der zu sortierenden Datensätze. Dann ist $h = \lfloor \log n \rfloor$ die Höhe des Baums, der zur Darstellung des betreffenden Minimumsheaps eingesetzt wird. Zunächst stellen wir fest, daß für jedes $i \in \{0, 1, \dots, h-1\}$ höchstens 2^i Elemente auf Tiefe i eingefügt werden. Jeder solche Schlüsselwert, welcher auf der i -ten Ebene eingefügt wird, kann um maximal $h-i$ Ebenen sinken. Der Rechenaufwand für den Aufbau des Heaps ist daher durch $\mathcal{O}(\frac{n}{2} + K)$ beschränkt, wobei der Summand $\frac{n}{2}$ für die Behandlung der letzten $n/2$ Elemente steht und

$$\begin{aligned}
 K &= \sum_{i=0}^{h-1} 2^i \cdot (h-i) \\
 &= \sum_{j=1}^h 2^{h-j} \cdot j \\
 &= 2^h \cdot \sum_{j=1}^h \frac{j}{2^j} \\
 &\leq 2^h \cdot \underbrace{\sum_{j=0}^{\infty} \frac{j}{2^j}}_{=2} \\
 &= \mathcal{O}(2^h) = \mathcal{O}(n)
 \end{aligned}$$

Beachte, daß $\sum_{j=0}^{\infty} \frac{j}{2^j} = \frac{1}{2} \sum_{j=1}^{\infty} j \cdot \left(\frac{1}{2}\right)^{j-1} = \frac{1}{2} \cdot f' \left(\frac{1}{2}\right)$, wobei für $|x| < 1$:

$$f(x) = \sum_{j=0}^{\infty} x^j = \frac{1}{1-x}, \text{ also } f'(x) = \sum_{j=1}^{\infty} j \cdot x^{j-1} = \frac{1}{(1-x)^2}$$

Umgekehrt ist klar, daß der Heapaufbau für n Elemente mindestens $\Omega(n)$ Schritte erfordert. Daher werden für den Heapaufbau $\Theta(n)$ Schritte durchgeführt (im schlimmsten und mittleren Fall). Wir erhalten die worst-case Laufzeit $\Theta(n \log n)$. Da $\Omega(n \log n)$ eine untere Schranke für die mittlere Laufzeit jedes Sortierverfahrens mit Vergleichen ist (Satz 2.1.7, ist $\Theta(n \log n)$ zugleich die asymptotische mittlere Laufzeit. Wir halten diese Ergebnisse in folgendem Satz fest:

Satz 3.3.1 (Kosten für Heapsort). Mit Heapsort kann eine Zahlenfolge der Länge n in Zeit $\Theta(n \log n)$ sortiert werden (worst case und average case).

4 Verwaltung dynamischer Mengen

Wir betrachten Datenstrukturen zur Verwaltung von „Dictionaries“, d.h. Datenmengen, welche sich dynamisch durch Einfügen und Löschen von Datensätzen verändern und für welche die Zugriffe (lesend oder schreibend) auf gespeicherte Datensätze unterstützt werden soll. Es wird dabei davon ausgegangen, daß jeder Datensatz ein Verbund

$$\langle \text{Schlüsselwert}, \text{Information} \rangle$$

ist. Wir gehen hier von sogenannten *Primärschlüsseln* aus, welche die Datensätze eindeutig charakterisieren. D.h. je zwei Datensätze haben unterschiedliche Schlüsselwerte. Zur Vereinfachung abstrahieren wir von der eigentlichen Information der Datensätze und identifizieren die Datensätze mit ihren Schlüsselwerten. Typische Schlüsselwerte sind Zahlen, die für irgendwelche Identifikationsnummern stehen (z.B. Matrikelnummer, Personalnummer, Steuernummer, Paßnummer, etc.). Die genannten Beispiele sind typisch für extern gespeicherte Datenmengen. Die hier vorgestellten Verfahren können jedoch zur Verwaltung nur temporär (zur Laufzeit eines Programmabschnitts) benötigten Datenmengen eingesetzt werden.

Die generelle Annahme, die den Betrachtungen in diesem Kapitel unterliegt, ist, daß die Menge der möglichen Schlüsselwerte, das so genannte *Schlüsseluniversum* \mathcal{U} , sehr viel größer als die Menge der Schlüsselwerte der aktuellen Datenmenge \mathcal{D} ist. Einfache Datenstrukturen wie Bitvektoren, sortierte Arrays oder Listen sind zwar (bedingt) einsetzbar, jedoch sind sie im allgemeinen als nicht effizient einzustufen. So sind z.B. Bitvektoren nur für kleine Schlüsseluniversen geeignet. Für Array-Darstellungen sowie geordnete Listen sind Einfügen und Löschen kostspielig. Für ungeordnete Listen ist das Suchen aufwendig.

	Suchen	Einfügen	Löschen	Platzbedarf
Bitvektoren	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(\mathcal{U})$
sortierte Arrays	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n_{max})$
ungeordnete Listen	$\Theta(n)$	$\Theta(1)$ oder $\Theta(n)$	$\Theta(1)$	$\Theta(n)$
geordnete Listen	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Optimal wäre eine Datenstruktur, die es ermöglicht, eine Datenmenge mit n Datensätzen in Platz $\mathcal{O}(n)$ darzustellen und die Operationen Suchen, Einfügen und Löschen in konstanter Zeit auszuführen.²⁷ Dies ist leider nicht machbar, jedoch gibt es Techniken, die diesen Forderungen recht nahekommen.

- **Hashing:** *schlechtes worst-case Verhalten, aber gutes durchschnittliches Verhalten.* Unter gewissen Voraussetzungen ist der Platzbedarf $\mathcal{O}(n)$ und die durchschnittliche Zugriffszeit konstant.
- **Balancierte Suchbäume:** *kontrolliertes worst-case Verhalten.* Der Platzbedarf ist stets $\mathcal{O}(n)$. Die Zugriffszeiten sind logarithmisch.

²⁷Bei der Angabe des Platzbedarfs machen wir die vereinfachende Annahme, daß jeder Datensatz nur $\mathcal{O}(1)$ Platz benötigt.

Eine weitere Datenstruktur mit gutem durchschnittlichen Verhalten haben wir bereits kennengelernt: die Skiplisten.

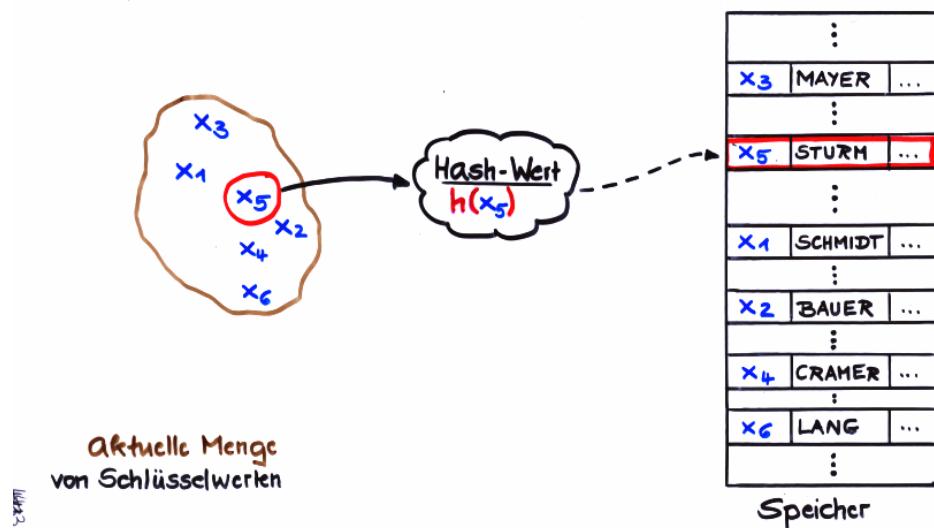
4.1 Hashing

In diesem Abschnitt nehmen wir an, daß das Schlüsseluniversum \mathcal{U} eine endliche Teilmenge von \mathbb{N} ist. Dies ist keine echte Einschränkung, sofern \mathcal{U} endlich ist, da man dann \mathcal{U} bijektiv auf eine endliche Teilmenge von \mathbb{N} abbilden kann, was einer Codierung der Schlüsselwerte entspricht.

Hashing:

Gegeben: Menge von Datensätzen mit **Schlüsselwerten**

Ziel: Ermittle durch **Berechnung** die Adresse eines Datensatzes



Hashing basiert auf dem naiven Grundgedanken, für jeden Schlüsselwert $x \in \mathcal{U}$, die „Adresse“ $h(x)$, an der ein Datensatz d mit Schlüsselwert $x = \text{key}(d)$ gespeichert ist bzw. gespeichert werden müßte, durch *Berechnung* zu ermitteln. Der Begriff „Adresse“ ist jedoch im weiten Sinn zu verstehen. Der zur Darstellung der Datenmenge zur Verfügung stehende Speicherbereich wird in sogenannten *Buckets* eingeteilt, etwa B_0, B_1, \dots, B_{m-1} .²⁸ Dabei ist im allgemeinen $|\mathcal{U}| \gg m$. (Das Symbol \gg ist als „sehr viel größer als“ zu lesen.) Eine *Hashfunktion* (für das Schlüsseluniversum \mathcal{U}) ist eine Abbildung

$$h : \mathcal{U} \rightarrow \{0, 1, \dots, m - 1\},$$

die jedem möglichen Schlüsselwert $x \in \mathcal{U}$ einen Hashwert $h(x) \in \{0, 1, \dots, m - 1\}$ zuordnet. Dabei setzen wir voraus, daß m eine natürliche Zahl mit $m \geq 2$ ist. Ein

²⁸Zunächst kann man sich vorstellen, daß die Buckets paarweise disjunkt sind und den kompletten zur Verfügung stehenden Speicherbereich abdecken. Einige Varianten von Hashing weichen jedoch von der ersten Bedingung ab und lassen zu, daß sich die Buckets überlappen.

Datensatz d mit $key(d) = x$ und $h(x) = i$ wird dann irgendwo in Bucket B_i gespeichert. Anstelle eindeutig lokalisierbarer Speicheradressen gibt $h(x)$ also lediglich über den betreffenden Speicherbereich (Bucket) Aufschluß.

Zur Wahl der Hashfunktion. Die Hashfunktion h sollte leicht zu berechnen sein und möglichst gut streuen (d.h. die Schlüsselwerte weitgehend gleichmäßig auf die Buckets verteilen.). Beispiele für häufig verwendete Hashfunktionen sind:

- die Divisions-Rest-Methode: Modulo-Funktionen
- die Mittel-Quadrat-Methode
- die Multiplikationsmethode

Siehe Abbildung 28.

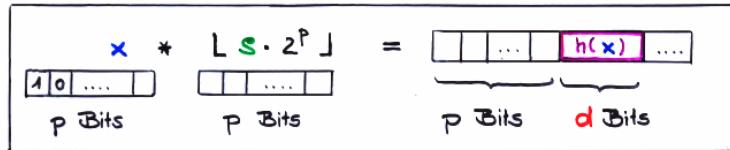
Beispiele für Hash-Funktionen:

- * Modulo - Funktionen $h(x) = x \bmod m$
- * Mittel - Quadrat - Methode : $h(x) = \text{"mittlerer Ziffernblock" von } x^2$

x	$x \bmod 100$	x^2	Mittel - Quadrat
128	28	16 384	38
129	29	16 641	64
130	30	16 900	90

- * Multiplikations - Methode:

s irrationale Zahl in $[0, 1[$, $d \in \mathbb{N}_{\geq 1}$



↗

Abbildung 28: Beispiele für Hashfunktionen

Kollisionen. Man spricht von einer *Kollision*, wenn zwei gespeicherte Datensätze d_1 , d_2 denselben Hashwert haben, d.h.

$$h(\text{key}(d_1)) = h(\text{key}(d_2)).$$

Die Behandlung von Kollisionen erfordert eine zusätzliche Organisation der Datensätze mit demselben Hashwert. Dazu werden listen-ähnliche Strukturen verwendet (siehe unten).

Wie wahrscheinlich sind Kollisionen? Selbst bei einer perfekten Hashfunktion h , welche die Schlüsselwerte optimal auf die Buckets (die Werte $0, 1, \dots, m-1$) verteilt und unter der Annahme einer Gleichverteilung für die Schlüsselwerte des Universums \mathcal{U} , Kollisionen zu erwarten. Dies ergibt sich aus folgender Beobachtung:

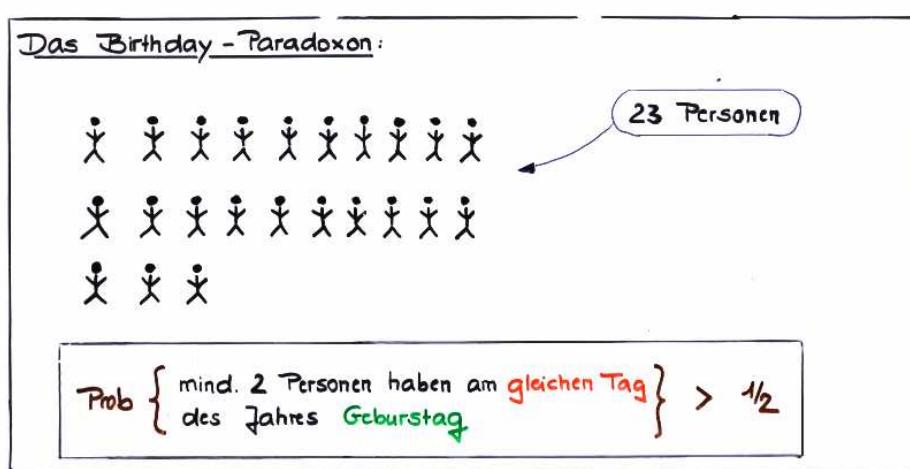
Die Datenmenge \mathcal{D} werde durch sukzessives Einfügen der Datensätze d_1, \dots, d_n erstellt. Sei $x_j = \text{key}(d_j)$, $j = 1, \dots, n$, der zu Datensatz d_j gehörende Schlüsselwert. Die Wahrscheinlichkeit, daß keine Kollision auftritt, also daß $h(x_1), \dots, h(x_n)$ paarweise verschieden sind, ist

$$1 \cdot \frac{m-1}{m} \cdot \frac{m-2}{m} \cdot \dots \cdot \frac{m-n+1}{m} = \frac{(m-1) \cdot (m-2) \cdot \dots \cdot (m-n+1)}{m^{n-1}},$$

wobei $n \leq m$ vorausgesetzt wird, da für $m < n$ zwangsweise eine Kollision auftritt. Der i -te Faktor $\frac{m-i+1}{m}$ steht für die Wahrscheinlichkeit, daß die i -te Einfügeoperation keine Kollision auslöst, unter der Bedingung, daß auch zuvor keine Kollision eingetreten ist. Wir erhalten:

$$\begin{aligned} \text{W'keit für eine Kollision} &= 1 - \text{W'keit, daß keine Kollision auftritt} \\ &= 1 - \frac{(m-1) \cdot (m-2) \cdot \dots \cdot (m-n+1)}{m^{n-1}} \end{aligned}$$

Für $m = 365$ und $n = 23$ ergibt sich die Wahrscheinlichkeit 0.507 für eine Kollision. Diese Aussage ist auch als das *Birthday-Paradoxon* bekannt. Siehe Abbildung 29.



... als Indiz für die "hohe Wahrscheinlichkeit" von Kollisionen

BRUNNEN

Abbildung 29: Birthday-Paradoxon

Kollisionsbehandlung. Zwei prinzipielle Strategien zur Auflösung von Kollisionen:

- **Hashing mit Verkettung der Überläufer:** Für jeden Hashwert $j \in \{0, 1, \dots, m - 1\}$ wird eine *Kollisionsliste* erstellt, in der sämtliche Datensätze d mit dem Hashwert $h(key(d)) = j$ verkettet werden. Die Listenköpfe werden in der Hashtabelle verwaltet.
- **Hashing mit offener Adressierung:** Die Datensätze werden *in* der Hashtabelle abgelegt, wobei eine implizite Verkettung von kollidierenden Datensätzen mit Hilfe einer Folge von Hashfunktionen eingesetzt wird.

4.1.1 Hashing mit Kollisionslisten

Hashing mit Kollisionslisten speichert alle Datensätze mit demselben Hashwert j in einer linearen, unsortierten Liste, die so genannten Kollisionsliste. Die eigentlichen Datensätze werden also außerhalb der Hashtabelle gespeichert. Die Listenköpfe der Kollisionslisten sind in der Hashtabelle abgelegt. Siehe Abbildung 30.

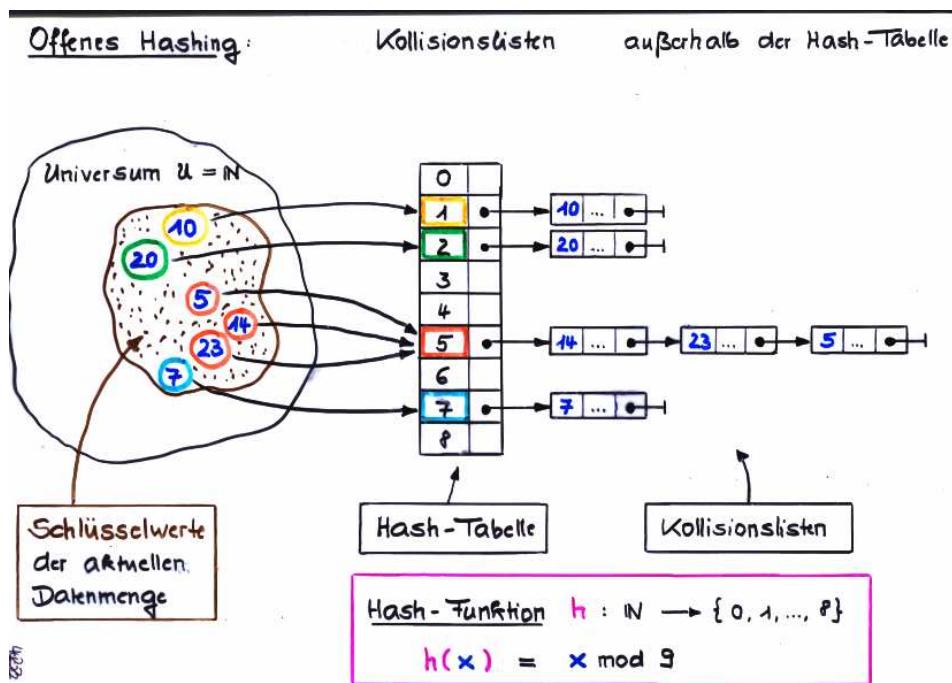


Abbildung 30: Hashing mit Kollisionslisten

Die Operationen Suchen, Einfügen und Löschen lassen sich in der offensichtlichen Weise mit den betreffenden Listenoperationen durchführen. (Die Einfügeoperationen finden jeweils am Listenende statt.)

In dem folgenden Satz analysieren wir die Kosten, wobei wir annehmen, daß die Hashwerte in konstanter Zeit berechnet werden können. Für die Durchschnittsanalyse wird eine

perfekte Hashfunktion und *Gleichverteilung* der Schlüsselwerte $x \in \mathcal{U}$ vorausgesetzt. D.h. wir gehen davon aus, daß für jeden Hashwert $j \in \{0, 1, \dots, m - 1\}$ die Wahrscheinlichkeit für das Ereignis „ $h(x) = j$ “ gleich $1/m$ ist, wobei x ein zufällig gewählter Schlüsselwert aus \mathcal{U} ist.

Diese Annahme garantiert, daß die erwartete Länge der Kollisionslisten gleich n/m ist, wenn n die Anzahl der gespeicherten Datensätze ist. Dies erklärt sich daraus, daß unter der Annahme einer perfekten Hashfunktion und Gleichverteilung der Schlüsselwerte die erwartete Länge aller Kollisionslisten übereinstimmt. Da die Gesamtlänge der m Kollisionslisten gleich n ist, ist $\frac{n}{m}$ die erwartete Länge jeder der Kollisionslisten. (Eine andere Argumentationsmöglichkeit wurde im Kontext der Analyse von Bucketsort für gleichverteilte reelle Schlüsselwerte und $n = m$ angegeben. Das hier genannte „intuitive“ Argument wäre dort ebenso einsetzbar, um die erwartete Listenlänge 1 aller Buckets zu begründen.)

Satz 4.1.1 (Komplexität von Hashing mit Kollisionslisten). Für Hashing mit Kollisionslisten benötigt die Suche nach einem Schlüsselwert

- im schlimmsten Fall $n + 1$ Schritte
- im Durchschnitt $1 + \frac{n}{m}$ Schritte für eine erfolglose Suche
- im Durchschnitt höchstens $1 + \frac{n}{2m}$ Schritte für eine erfolgreiche Suche.

Die Platzkomplexität ist $\mathcal{O}(n + m)$. Dabei ist n die Anzahl der gespeicherten Datensätze und m die Größe der Hashtabelle.

Beweis. In allen drei Fällen bezeichnet der Summand 1 die Kosten für das Berechnen des Hashwerts und damit verbunden den Zugriff auf den Listenkopf der betreffenden Kollisionsliste.

Der schlimmste Fall tritt ein, wenn alle Datensätze auf denselben Hashwert abgebildet werden, und somit die betreffende Kollisionsliste aus n Elementen besteht. Wir erläutern nun die angegebenen Kosten für den durchschnittlichen Fall.

Für die *erfolglose* Suche steht der Summand $\frac{n}{m}$ für den Suchvorgang in der betreffenden Kollisionsliste. Man beachte, daß die mittlere Länge der Kollisionslisten gleich $\frac{n}{m}$ ist (siehe oben.)

Für eine *erfolgreiche* Suche ergibt sich die mittlere Schrittanzahl, indem die Werte für eine erfolglose Suche mit Belegungsfaktoren $\frac{i}{m}$, $i = 0, 1, \dots, n - 1$, mit Wahrscheinlichkeiten gewichtet aufsummiert werden. Man beachte, daß bei der erfolglosen Suche nach dem $(i + 1)$ -sten eingefügten Element unmittelbar nach Einfügen des i -ten Elements dieselben Schritte durchgeführt werden wie bei der erfolgreichen Suche nach dem $(i + 1)$ -sten eingefügten Element nach Einfügen aller n Elementen. Etwaige Löschoperationen, die zwischendurch ausgeführt werden, können die Laufzeit nur verringern und wurden in obiger Argumentation daher ignoriert.

Die mittlere Anzahl an Schritten für eine erfolgreiche Suche ist also durch K beschränkt, wobei

$$\begin{aligned}
K &= \frac{1}{n} \sum_{i=1}^n \left(1 + \frac{i-1}{m} \right) \\
&= n \cdot \frac{1}{n} + \frac{1}{nm} \cdot \sum_{i=1}^{n-1} i \\
&= 1 + \frac{1}{nm} + \frac{n(n-1)}{2} \\
&= 1 + \frac{n-1}{2m} \leq 1 + \frac{n}{2m}
\end{aligned}$$

□

Die Kosten für Einfüge- und Löschoperationen werden durch die vorangegangene Suche dominiert. Sie ergeben sich somit aus Satz 4.1.1 und können durch $\Theta(1 + \frac{n}{m})$ angegeben werden.

Für den Fall $n \approx m$, oder allgemeiner $n \approx Cm$ für eine Konstante C , ergeben sich also *konstante mittlere Ausführungszeiten* für die Operationen Suchen, Löschen und Einfügen.

4.1.2 Hashing mit offener Adressierung

Die Idee für Hashing mit offener Adressierung besteht darin, die Datensätze *in* der Hashtabelle abzulegen und anstelle von Kollisionslisten eine implizite Verkettung der Datensätze durch den Einsatz mehrerer Hashfunktionen vorzunehmen.

Wir beschränken uns hier auf den Fall, in dem pro Zelle in der Hashtabelle höchstens ein Datensatz abgelegt wird. Dieselben Techniken können eingesetzt werden, wenn für in jeder der m Zellen der Hashtabelle bis zu b Datensätze eingetragen werden können. Dabei ist b eine im allgemeinen sehr kleine, ganze Zahl b , die a-priori festzulegen ist. Insgesamt können dann $n \leq m \cdot b$ Datensätze gespeichert werden. Typische Werte für b sind $b \in \{1, 2, 3, 4\}$. Wir betrachten hier nur den Fall $b = 1$. Dies macht selbstverständlich nur für $n \leq m$ Sinn, etwa $m \approx 2n$. Offene Adressierung ist daher nur dann sinnvoll einsetzbar, wenn die Anzahl n an darzustellenden Datensätzen in etwa a-priori bekannt ist.

Sondierungsfolgen. Zur Kollisionsbehandlung verwendet man eine *offene Adressierung* mit Hilfe einer Folge $h = h_0, h_1, h_2, \dots, h_{m-1}$ von m Hashfunktionen, so daß

$$\{0, 1, \dots, m-1\} = \{h_j(x) : j = 0, 1, \dots, m-1\} \quad \text{für alle } x \in \mathcal{U}.$$

h wird auch die primäre Hashfunktion genannt. Die *Sondierungsfolge* $h_0(x), h_1(x), h_2(x), \dots$ wird für die Zugriffe auf einen Datensatz mit Schlüsselwert x (also zur Durchführung der Operationen Suchen, Löschen und Einfügen) verwendet.

Suchen, Löschen, Einfügen. Die Grundidee für die Suche nach einem Datensatz mit Schlüsselwert x besteht darin, sukzessive die Einträge in der Hashtabelle, die sich an den Positionen $h_0(x), h_1(x), h_2(x), \dots$ in der Hashtabelle befinden, zu betrachten, bis der Schlüsselwert x gefunden ist oder kein Eintrag in der Zelle für den betreffenden Hashwert $h_i(x)$ vorliegt oder alle m Zellen der Hashtabelle erfolglos betrachtet wurden. Diese Vorgehensweise ist adäquat, falls eine statische Datenmenge vorliegt, die durch sukzessives Einfügen erstellt wird, jedoch keine Löschoperationen durchgeführt werden. Die Behandlung dynamischer Datenmengen, welche sich dynamisch durch Einfügen und Löschen von Datensätzen verändern, ist bei offener Adressierung etwas aufwendiger. Wird ein Element gelöscht, so muß verhindert werden, daß der Sondierungsvorgang eines anschließenden Suchprozesses nicht an der Position des gelöschten Elements abbricht, obwohl weitere Datensätze zu dem betreffenden (primären) Hashwert gespeichert sind. Wir machen uns die Problematik am Beispiel der linearen Sondierung klar, welche Hashfunktionen des Typs $h_i(x) = (x + i) \bmod m$, $i = 0, 1, \dots, m - 1$, einsetzt. Sind z.B. d_1, d_2, d_3 Datensätze mit dem Hashwert $h(key(d_i)) = j$, wobei d_1, d_2, d_3 an den Positionen $j, j + 1$ und $j + 2$ eingetragen sind, und wird der Datensatz d_1 gelöscht, so ist es für anschließende Zugriffe auf d_2 und d_3 wichtig, daß Position j mit einem geeigneten Vermerk versehen wird, aus welchem ersichtlich wird, daß weitere Elemente in der Sondierungsfolge vorkommen.

Um alle drei Operationen Suchen, Löschen und Einfügen zu unterstützen, wird jede Zelle der Hashtabelle mit zwei Booleschen Werten markiert. Einer dieser beiden Booleschen Werte gibt an, ob die betreffende Zelle *frei* ist, während der andere Boolesche Wert Aufschluss darüber gibt, ob der Inhalt der betreffenden Zelle *gelöscht* wurde. Die Markierungen *gelöscht* sind wesentlich für den Suchprozeß.

- **Suchen** eines Datensatzes mit Schlüsselwert x : Betrachte der Reihe nach die Zellen $h(x) = h_0(x), h_1(x), h_2(x), \dots$ bis entweder x gefunden wurde oder die betrachtete Zelle $h_j(x)$ *frei*, aber *nicht gelöscht*, ist oder $j = m - 1$.
- **Einfügen** eines Datensatzes d mit Schlüsselwert x : Bestimme den kleinsten Index j , so daß die Zelle $h_j(x)$ in der Hashtabelle *frei* ist. Füge dort den Datensatz d ein und markiere die betreffende Zelle als *nicht frei*.
- **Löschen** eines Datensatzes: Markiere die betreffende Zelle in der Hashtabelle als *frei* und *gelöscht*.

Da die Markierungen *gelöscht* niemals zurückgenommen werden, ist offene Adressierung für sehr dynamische Datenmengen wenig geeignet. Sobald nämlich m verschiedene Datensätze betrachtet (eingefügt und eventuell später gelöscht) wurden, so ist jede Zelle der Hashtabelle mit der Markierung *gelöscht* versehen und jede erfolglose Suche erfordert $\Theta(m)$ Suchschritte.

Hashing mit offener Adressierung ist zwar oftmals platzeffizienter als Hashing mit Kollisionslisten; vorteilhaft ist die offene Adressierung jedoch nur für Datenmengen beschränkter Größe (Anzahl der Datensätze $\leq m$ bzw. $\leq m \cdot b$, falls bis zu b Datensätze pro Zelle ablegt werden können), die durch sukzessives Einfügen entstehen, aber keine oder höchstens sehr wenig Löschoperationen stattfinden.

Kosten für Hashing mit offener Adressierung. Wie zuvor sei n die Anzahl an gespeicherten Datensätzen und m die Anzahl an Zellen in der Hashtabelle, wobei $n < m$ vorausgesetzt wird. Wie bereits erwähnt, ist das Konzept der offenen Adressierung für Datenmengen mit häufigen Löschoperationen kritisch zu sehen. Wurde in jede Hashzelle mindestens einmal ein Datensatz eingefügt und später eventuell gelöscht, dann entstehen für eine erfolglose Suche zwangsläufige Sondierungsfolgen der Länge m .

Die folgenden Betrachtungen zur Effizienz der offenen Adressierung beziehen sich auf den Fall, daß keine Löschoperationen vorgenommen wurden. Eine hierzu gleichwertige Sicht ist, daß die Zahl n die Anzahl an Datensätzen angibt, die irgendwann eingefügt wurden, also auch die gelöschten Datensätze erfasst. Die Forderung $n < m$ bleibt jedoch auch hier bestehen.

Ausgangspunkt für die Analyse der durchschnittlichen Kosten ist die Annahme von *idealen Sondierungsfolgen*, in denen jede der Permutationen von $0, 1, 2, \dots, m - 1$ mit derselben Wahrscheinlichkeit als Sondierungsfolge $h_0(x), h_1(x), \dots, h_{m-1}(x)$ für alle $x \in \mathcal{U}$ auftritt. Dies entspricht der Annahme der Gleichverteilung aller Schlüsselwerte und einer perfekten Sondierungsstrategie.

Satz 4.1.2 (Kosten von Hashing mit offener Adressierung (ohne Löschoperationen)). Unter den oben genannten Voraussetzungen werden

- im schlimmsten Fall n Sondierungsschritte
- im Mittel höchstens $\frac{1}{1-\frac{n}{m}}$ Sondierungsschritte für eine erfolglose Suche
- im Mittel höchstens $\frac{m}{n} \ln \frac{1}{1-\frac{n}{m}} + \frac{m}{n}$ Sondierungsschritte für eine erfolgreiche Suche

durchgeführt. Der Platzbedarf ist $\Theta(n + m) = \Theta(m)$.

Obige Formeln sind leichter lesbar, wenn man sie in Abhängigkeit des Belegungsfaktors $\alpha = \frac{n}{m}$ hinschreibt. Dann ist die mittlere Anzahl an Sondierungsschritten durch $\frac{1}{1-\alpha}$ für eine erfolglose Suche und durch $\frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha}$ für eine erfolgreiche Suche nach oben beschränkt.

Beweis. Im schlimmsten Fall werden alle n Schlüsselwerte auf denselben Hashwert abgebildet, womit sich die maximale Anzahl n an Sondierungsschritten ergibt. Wir betrachten nun den mittleren Fall.

Für eine *erfolglose* Suche schätzen wir die mittlere Anzahl an Sondierungsschritten wie folgt ab. Die Wahrscheinlichkeit, daß der erste Sondierungsschritt auf eine Zelle trifft, die bereits (durch einen anderen Schlüsselwert) belegt ist, beträgt $\frac{n}{m}$. Für den zweiten Sondierungsschritt stehen nun nur noch $m - 1$ Zellen zur Verfügung und auf diese entfallen $n - 1$ Schlüsselwerte. Die Wahrscheinlichkeit, daß auch der zweite Sondierungsschritt auf eine bereits belegte Zelle trifft, ist daher $\frac{n-1}{m-1}$. Mit demselben Argument erhalten wir, daß mit Wahrscheinlichkeit $\frac{n-i+1}{m-i+1}$ auch der i -te Sondierungsschritt auf eine belegte Zelle

trifft. Dies führt zu folgender Rechnung:

erwartete Anzahl an Sondierungsschritten
für eine erfolglose Suche

$$\begin{aligned}
&= \sum_{i=1}^n i \cdot \text{Prob}(\text{genau } i \text{ Sondierungsschritte}) \\
&= \sum_{i=1}^n \text{Prob}(\text{Anzahl Sondierungsschritte } \geq i) \\
&= \sum_{i=1}^n \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+1}{m-i+1} \\
&\leq \sum_{i=1}^n \left(\frac{n}{m}\right)^i \\
&\leq \sum_{i=0}^{\infty} \left(\frac{n}{m}\right)^i \\
&= \frac{1}{1 - \frac{n}{m}}
\end{aligned}$$

Wir betrachten nun die *erfolgreiche* Suche. Im Folgenden nehmen wir an, daß die Hashtabelle durch sukzessives Einfügen aus Datensätzen mit den Schlüsselwerten x_1, \dots, x_n (in dieser Reihenfolge) belegt wird. Wir argumentieren nun wie in der Analyse von Hashing mit Kollisionslisten und stellen fest, daß die mittlere Anzahl an Sondierungsschritten für die erfolgreiche Suche nach dem $(i+1)$ -sten eingefügten Schlüsselwert x_{i+1} gleich der mittleren Anzahl an Sondierungsschritten für die erfolglose Suche nach x_{i+1} unmittelbar nach Einfügen der ersten i Elemente x_1, \dots, x_i ist. Obige Rechnung zeigt, daß dieser Wert durch

$$\frac{1}{1 - \frac{i}{m}} = \frac{m}{m-i}$$

nach oben beschränkt ist.

Wir erhalten:

erwartete Anzahl an Sondierungsschritten
für eine erfolgreiche Suche

$$\begin{aligned}
 &\leq \frac{1}{n} \cdot \sum_{i=0}^{n-1} \frac{m}{m-i} \\
 &= \frac{m}{n} \cdot \sum_{i=0}^{n-1} \frac{1}{m-i} = \frac{m}{n} \cdot \sum_{j=m-n+1}^m \frac{1}{j} \\
 &= \frac{m}{n} \cdot \left(\underbrace{\sum_{j=1}^m \frac{1}{j}}_{=H(m)} - \underbrace{\sum_{j=1}^{m-n} \frac{1}{j}}_{=H(m-n)} \right) = \frac{m}{n} \cdot (H(m) - H(m-n))
 \end{aligned}$$

Dabei ist $H(k) = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{k}$ die k -te Partialsumme der harmonischen Reihe. Im Kontext der Average-Case-Analyse von Quicksort (siehe Seite 36 ff) haben wir gesehen, daß

$$\ln k \leq \ln(k+1) \leq H(k) \leq \ln k + 1.$$

Wir setzen dies für $k = m$ bzw. $k = m - n$ in obige Formel ein und erhalten:

erwartete Anzahl an Sondierungsschritten
für eine erfolgreiche Suche

$$\begin{aligned}
 &\leq \frac{m}{n} \cdot (H(m) - H(m-n)) \\
 &\leq \frac{m}{n} \cdot (\ln m + 1 - \ln(m-n)) \\
 &= \frac{m}{n} \cdot \left(\ln \frac{m}{m-n} + 1 \right) \\
 &= \frac{m}{n} \cdot \ln \frac{m}{m-n} + \frac{m}{n}
 \end{aligned}$$

□

Die Kosten für Einfügen (evtl. Löschen) ergeben sich aus den Kosten der vorangehenden Suche und können somit aus Satz 4.1.2 hergeleitet werden.

Für eine zu 90% gefüllte Hashtabelle (also den Belegungsfaktor $\frac{n}{m} = \frac{9}{10}$) werden im Mittel höchstens

$$\frac{10}{9} \ln \frac{1}{1-\frac{9}{10}} + \frac{10}{9} \approx 3.67$$

Sondierungsschritte im Falle einer erfolgreichen Suche ausgeführt, während für eine erfolglose Suche im Mittel mit maximal $\frac{1}{1-\frac{9}{10}} = 10$ Sondierungsschritten zu rechnen ist. Allgemein gilt: Ist der Belegungsfaktor $\frac{n}{m}$ durch eine Konstante C mit $0 < C < 1$ beschränkt, also $n \leq Cm$, so ergeben sich konstante mittlere Zugriffszeiten.

Im Folgenden stellen wir drei Instanzen für Hashing mit offener Adressierung vor, die sich durch die Art und Weise, wie sich die Sondierungsfolgen ergeben, unterscheiden.

Lineare Sondierung. Die einfachste Form der offenen Adressierung ist die *lineare Sondierung*, welcher die Hashfunktionen

$$h_i(x) = (h(x) + i) \bmod m, \quad i = 0, 1, \dots, m - 1$$

zugrunde liegen. Wird etwa ein Datum d mit dem Hashwert $j = h(key(d))$ gesucht, so betrachtet man der Reihe nach die Positionen $j, j + 1, j + 2, \dots$ (modulo m), bis entweder alle Positionen betrachtet wurden oder der gesuchte Datensatz gefunden wurde.

Quadratische Sondierung. Ein Nachteil der linearen Sondierung ist die *Clusterbildung*. Hierunter versteht man die Tendenz, lange zusammenhängende Abschnitte in der Hashtabelle mit Datensätzen zu belegen, während andere zusammenhängende Abschnitte frei bleiben. Derartige Cluster treten auf, wenn mehrere Schlüsselwerte auf denselben Hashwert abgebildet werden, sind aber auch leicht möglich, wenn die Hashwerte nur wenig differieren. Liegen z.B. k Einträge mit denselben Hashwerten $h(x_1) = \dots = h(x_k) = 1$ vor und wird ein Element mit dem Hashwert $h(x_{k+1}) = 2$ hinzugefügt, so wird dieses in Zelle $k + 1$ eingefügt, das nächste Element mit dem Hashwert $h(x_{k+2}) = 2$ in Zelle $k + 2$, usw.

Clusterbildung hat schlechten Einfluss auf die mittlere Suchzeit und ist einer Hashtabelle mit demselben Belegungsfaktor n/m , in welcher die Elemente gut gestreut sind, unterlegen. Eine etwas mildere Form möglicher Clusterbildungen wird durch das quadratische Sondieren mit den Hashfunktionen

$$\begin{aligned} h_0(x) &= h(x) \\ h_1(x) &= (h(x) + 1) \bmod m, \\ h_2(x) &= (h(x) - 1) \bmod m, \\ h_3(x) &= (h(x) + 4) \bmod m, \\ h_4(x) &= (h(x) - 4) \bmod m, \\ &\vdots \\ h_{2j-1}(x) &= (h(x) + j^2) \bmod m, \\ h_{2j}(x) &= (h(x) - j^2) \bmod m, \\ &\vdots \end{aligned}$$

(für $j = 1, \dots, \frac{m-1}{2}$) erreicht. Um sicherzustellen, daß für festes x die $h_i(x)$'s alle Elemente $0, 1, \dots, m - 1$ erfassen, muß m eine Primzahl mit $m \bmod 4 = 3$ sein. Auch wenn die Clusterbildung weniger extrem als bei der linearen Sondierung ist,

so liegt dennoch allen Schlüsselwerten $x, y \in \mathcal{U}$ mit denselben primären Hashwerten $h(x) = h(y)$ dieselbe Sondierungsfolge zugrunde. Man spricht auch von *sekundärer* Clusterbildung.

Double Hashing. Ein möglicher Ausweg aus dem Problem der Clusterbildung besteht in der Verwendung des Double Hashing, in welchem neben der primären Hashfunktion h eine weitere Hashfunktion h' eingesetzt wird. Betrachtet wird nun die Sondierungsfolge

$$\begin{aligned} h_0(x) &= h(x), \\ h_1(x) &= (h(x) + h'(x)) \bmod m, \\ h_2(x) &= (h(x) + 2h'(x)) \bmod m, \\ h_3(x) &= (h(x) + 3h'(x)) \bmod m, \\ &\vdots \\ h_{m-1}(x) &= (h(x) + (m-1)h'(x)) \bmod m, \end{aligned}$$

Wichtig dabei ist, daß die Tabellengröße m eine Primzahl ist und h' Werte zwischen 1 und $m-1$ annimmt, da andernfalls $\{h_j(x) : j = 0, 1, \dots, m-1\} = \{0, 1, \dots, m-1\}$ nicht garantiert werden kann. Beispielsweise ist

$$h(x) = x \bmod m \quad \text{und} \quad h'(x) = 1 + (x \bmod m - 1)$$

für eine Primzahl m eine gebräuchliche Wahl, da h und h' dann weitgehend „unabhängig“ sind.

Die Gefahr der Clusterbildung ist für das doppelte Hashing vergleichsweise gering, da für zwei Schlüsselwerte $x, y \in \mathcal{U}$ nur dann dieselbe Sondierungsfolge entsteht, wenn $h(x) = h(y)$ und $h'(x) = h'(y)$.

4.1.3 Universelles Hashing

Die zuvor vorgestellten Hashverfahren haben unter gewissen Voraussetzungen konstante durchschnittliche Zugriffszeit, jedoch sind die Kosten im schlimmsten Fall linear, also genauso schlecht wie für eine simple Listendarstellung. Die für die Durchschnittsanalyse geforderten Voraussetzungen für Hashing mit Kollisionslisten (Satz 4.1.1) waren perfekte Hashfunktionen und Gleichverteilung aller Schlüsselwerte. Dies ist jedoch eine kritische Annahme, da sie stark von der darzustellenden Datenmenge abhängt und in vielen Fällen als unrealistisch anzusehen ist. Im Extremfall können alle Datensätze auf denselben Hashwert abgebildet werden, was zu den linearen worst-case Kosten führt. Ob dieser Extremfall unwahrscheinlich ist oder nicht, hängt von den konkreten Eingaben (genauer den Schlüsselwerten der darzustellenden Datensätze) ab.

Wir erläutern nun, wie das Konzept der Randomisierung im Kontext von Hashing eingesetzt werden kann, um ein gutes durchschnittliches Verhalten für beliebige Datenmengen zu garantieren. Der Grundgedanke dabei ist sehr einfach und besteht darin, eine Hashfunktion *zufällig* aus einer geeigneten Menge \mathcal{H} zu wählen, wobei Gleichverteilung

der Hashfunktionen in \mathcal{H} angenommen wird, also jede der Hashfunktionen $h \in \mathcal{H}$ mit derselben Wahrscheinlichkeit (also $1/|\mathcal{H}|$) zum Zug kommt. Um ein gutes durchschnittliches Verhalten zu erreichen, wird die Forderung gestellt, daß die Wahrscheinlichkeit für eine Kollision für zwei beliebige Schlüsselwerte x und y (also die relative Anzahl an Hashfunktionen h , die x und y denselben Hashwert zuordnen) gleich $1/m$, also gleich der Kollisionswahrscheinlichkeit für zufällig gewählte Hashwerte $h(x), h(y) \in \{0, 1, \dots, m-1\}$, ist.

Definition 4.1.3 (Universelle Hashfunktionsklassen). Sei $m \geq 2$, \mathcal{U} das Schlüsseluniversum und \mathcal{H} eine nichtleere (und endliche) Menge von Hashfunktionen $h : \mathcal{U} \rightarrow \{0, 1, \dots, m-1\}$. \mathcal{H} heißt *universell*, falls für alle $x, y \in \mathcal{U}$ mit $x \neq y$ gilt:

$$\frac{|\{h \in \mathcal{H} : h(x) = h(y)\}|}{|\mathcal{H}|} = \frac{1}{m}$$

□

Der Ausgangspunkt für universelles Hashing ist eine universelle Menge von Hashfunktionen, aus der zufällig eine Hashfunktion h gewählt wird. Die weitere Vorgehensweise für die von nun an feste Hashfunktion h ist dann wie in den vorangegangenen Abschnitten, wobei wir hier nur den Fall von Kollisionslisten diskutieren.²⁹

Man beachte, daß die angegebene Bedingung für universelle Hashfunktionsklassen alle Schlüsselwerte gleich behandelt werden. Durch den Trick der Randomisierung kommt universelles Hashing der Annahme von perfekten Hashfunktionen und Gleichverteilung aller Schlüsselwerte tatsächlich nahe, was durch den folgenden Satz belegt wird. Dieser besagt, daß unter Einsatz von universellem Hashing die mittlere Anzahl an Kollisionen für alle Schlüsselwerte x gleich dem Belegungsfaktor $\frac{n}{m}$ ist, also gleich der erwarteten Länge der Kollisionslisten für eine n -elementige Datenmenge, wenn perfekte Hashfunktionen eingesetzt und Gleichverteilung der Schlüsselwerte vorausgesetzt werden.

Satz 4.1.4 (Erwartete Anzahl an Kollisionen für universelles Hashing). Sei \mathcal{H} eine universelle Menge von Hashfunktionen $h : \mathcal{U} \rightarrow \{0, 1, \dots, m-1\}$ und sei $\mathcal{D} \subseteq \mathcal{U}$ die Schlüsselmenge der aktuell dargestellten Datenmenge. Weiter sei $x \in \mathcal{U}$ und

$$\mathbb{E}\left[|Kollision(x, \mathcal{D})| \right]$$

der Erwartungswert für die Anzahl an Schlüsselwerten $y \in \mathcal{D}$ mit $h(x) = h(y)$ für eine zufällig gewählte Hashfunktion $h \in \mathcal{H}$. (D.h. $|Kollision(x, \mathcal{D})| = |\{y \in \mathcal{D} : h(x) = h(y)\}|$ für die zufällig gewählte Hashfunktion h .) Dann gilt:

$$\mathbb{E}\left[|Kollision(x, \mathcal{D})| \right] = \frac{n}{m} = \frac{|\mathcal{D}|}{m},$$

wobei $n = |\mathcal{D}|$ die Anzahl an aktuell dargestellten Datensätzen bezeichnet.

²⁹Für den Einsatz von universellem Hashing im Kontext der offenen Adressierung sind die Hashfunktionen h_0, h_1, h_2, \dots der Sondierungsfolgen unabhängig, randomisiert zu wählen. Die in Definition 4.1.3 angegebene Bedingung ist entsprechend zu modifizieren. Damit kann die Perfektion der Sondierungsfolgen im Erwartungswert erzielt werden.

Beweis. Für $x, y \in \mathcal{U}$ und $h \in \mathcal{H}$ sei

$$\delta(x, y, h) = \begin{cases} 1 & : \text{falls } h(x) = h(y) \\ 0 & : \text{sonst} \end{cases}$$

Dann gilt:

$$\begin{aligned} \mathbb{E}\left[|Kollision(x, \mathcal{D})| \right] &= \mathbb{E}\left[|\{y \in \mathcal{D} : h(x) = h(y)\}| \right] \\ &= \frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} |\{y \in \mathcal{D} : h(x) = h(y)\}| \\ &= \frac{1}{|\mathcal{H}|} \sum_{h \in \mathcal{H}} \sum_{y \in \mathcal{D}} \delta(x, y, h) \\ &= \frac{1}{|\mathcal{H}|} \sum_{y \in \mathcal{D}} \sum_{h \in \mathcal{H}} \delta(x, y, h) \\ &= \frac{1}{|\mathcal{H}|} \sum_{y \in \mathcal{D}} |\{h \in \mathcal{H} : h(x) = h(y)\}| \\ &= \sum_{y \in \mathcal{D}} \frac{|\{h \in \mathcal{H} : h(x) = h(y)\}|}{|\mathcal{H}|} \\ &= \sum_{y \in \mathcal{D}} \frac{1}{m} = \frac{|\mathcal{D}|}{m} \end{aligned}$$

□

Universelles Hashing verhält sich also im Mittel wie gewöhnliches Hashing unter optimalen Bedingungen. Aus dem obigen Satz folgt nämlich, dass sich die durchschnittliche Länge $\frac{n}{m}$ für die Kollisionslisten ergibt, wobei n die Anzahl an Datensätzen ist, welche durch Hashing mit Kollisionslisten verwaltet wird. Betrachtet man nämlich einen beliebigen Hashwert $i \in \{0, 1, \dots, m - 1\}$, so folgt aus Satz 4.1.4, dass die erwartete Anzahl an Datensätzen der dargestellten Datenmenge, welche den Hashwert i haben, also in die Kollisionsliste für i eingefügt werden, gleich $\frac{n}{m}$ ist.

Wir betonen nochmals, daß für die Analyse des universellen Hashings keinerlei Annahmen über die Verteilung der Schlüsselwerte gemacht werden und dennoch die Gleichwertigkeit aller Schlüsselwerte im durchschnittlichen Fall erreicht wird. Die Annahme der Perfektion der Hashfunktion ist in gewisser Weise in die Definition universeller Hashfunktionsklassen eingebaut. Ein ähnliches Phänomen haben wir bereits für die randomisierte Version von Quicksort vernommen. Man spricht auch vom *Robin Hood Effekt*, um eine derartige Strategie zu bezeichnen, welche durch Randomisierung eine Gleichbehandlung aller Eingaben und somit dasselbe Durchschnittsverhalten für alle Eingaben erzielt.

Beispiel für eine universelle Hashfunktionsklasse. Es bleibt die Frage zu klären, welche universellen Hashfunktionsklassen eingesetzt werden können. Im Folgenden nehmen wir $\mathcal{U} \subseteq \{0, 1, \dots, m^r - 1\}$ an, wobei m eine Primzahl ist. Wir können uns nun die Schlüsselwerte $x \in \mathcal{U}$ als r -stellige Zahlen über der Basis m vorstellen, also

$$x = \sum_{j=0}^{r-1} x_j m^j, \quad \text{wobei } x_0, \dots, x_{r-1} \in \{0, 1, \dots, m-1\}.$$

Sei $a = \sum_{j=0}^{r-1} a_j m^j$, wobei $a_0, \dots, a_{r-1} \in \{0, 1, \dots, m-1\}$. Dann ist $h_a : \mathcal{U} \rightarrow \{0, 1, \dots, m-1\}$ folgende Hashfunktion:

$$h_a(x) = \left(\sum_{j=0}^{r-1} a_j x_j \right) \bmod m.$$

Satz 4.1.5. Die Hashfunktionsklasse $\mathcal{H} = \{h_a : 0 \leq a < m^r\}$ ist universell.

Beweis. Zunächst stellen wir fest, daß die m^r Hashfunktionen h_a für $a \in \{0, 1, \dots, m^r - 1\}$ paarweise verschieden sind. Gilt nämlich $h_a = h_b$, wobei

$$a = \sum_{j=0}^{r-1} a_j m^j, \quad b = \sum_{j=0}^{r-1} b_j m^j,$$

so gilt

$$a_j = h_a(m^j) = h_b(m^j) = b_j, \quad j = 0, 1, \dots, r-1,$$

und somit $a = b$. Also ist $|\mathcal{H}| = m^r$.

Seien $x, y \in \mathcal{U}$ mit $x \neq y$, wobei

$$x = \sum_{j=0}^{r-1} x_j m^j, \quad y = \sum_{j=0}^{r-1} y_j m^j$$

und $x_0, \dots, x_{r-1}, y_0, \dots, y_{r-1} \in \{0, 1, \dots, m-1\}$. Zur Vereinfachung nehmen wir an, daß $x_0 \neq y_0$. Die Argumentation für andere Ziffern $x_j \neq y_j$ ist analog.

Für jedes $a = \sum_{j=0}^{r-1} a_j m^j$ gilt:

$$\begin{aligned} h_a(x) = h_a(y) \quad \text{gdw.} \quad \left(\sum_{j=0}^{r-1} a_j y_j \right) \bmod m &= \left(\sum_{j=0}^{r-1} a_j x_j \right) \bmod m \\ \text{gdw.} \quad a_0 \underbrace{(x_0 - y_0)}_{\neq 0} \bmod m &= - \left(\sum_{j=1}^{r-1} a_j (x_j - y_j) \right) \bmod m \end{aligned}$$

Da m ein Primzahl ist, ist $\mathbb{Z}/m\mathbb{Z}$ ein Körper und besitzt somit multiplikative Inverse für Elemente $\neq 0$. Daher gilt:

$$h_a(x) = h_a(y) \quad \text{gdw.} \quad a_0 = -(x_0 - y_0)^{-1} \left(\sum_{j=1}^{r-1} a_j(x_j - y_j) \right) \pmod{m}$$

Für jedes der m^{r-1} Tupel $(a_1, \dots, a_{r-1}) \in \{0, 1, \dots, m-1\}^{r-1}$ gibt es also genau ein $a_0 \in \{0, 1, \dots, m-1\}$, so daß $h_a(x) = h_a(y)$. Die Anzahl an Werten $a \in \{0, 1, \dots, m^r-1\}$, für welche $h_a(x)$ und $h_a(y)$ übereinstimmen, ist daher m^{r-1} . Hieraus folgt:

$$\frac{|\{h_a \in \mathcal{H} : h_a(x) = h_a(y)\}|}{|\mathcal{H}|} = \frac{m^{r-1}}{m^r} = \frac{1}{m}$$

Also ist \mathcal{H} universell. □

Ein weiteres Beispiel für eine universelle Menge von Hashfunktionen ist

$$\mathcal{H}_p = \{h_{a,b} : a, b \in \mathbb{N}, 1 \leq a < p, 0 \leq b < p\},$$

wobei $|\mathcal{U}| = p$ eine Primzahl ist. O.E. sei $\mathcal{U} = \{0, 1, \dots, p-1\}$. Für $a, b \in \mathbb{N}$ mit $a \geq 1$ ist die Hashfunktion $h_{a,b} : \mathbb{N} \rightarrow \{0, 1, \dots, m-1\}$ wie folgt definiert:

$$h_{a,b}(x) = ((ax + b) \pmod{p}) \pmod{m}.$$

Der Nachweis, daß \mathcal{H}_p tatsächlich universell ist, findet sich z.B. den in dem in der Fußnote³⁰ zitierten Artikel. (Vgl. Proposition 7 auf Seite 7).

4.1.4 Kollisionslisten beschränkter Länge

Abschließend erwähnen wir eine Variante von Hashing, welche Kollisionslisten beschränkter Länge b verwendet. Diese Technik eignet sich zur Effizienzsteigerung gewisser Berechnungen, in der die Hashtabelle als so genannte *Computed Table* verwendet wird. Zur Kollisionsauflösung wird folgende einfache Strategie verwendet: Liegen bereits b Einträge für den Hashwert $h(x) = j$ vor und soll ein Datensatz mit Schlüsselwert x eingefügt werden, dann wird – gemäß einer gewissen Heuristik – einer der alten Datensätze überschrieben. Diese Strategie erweist sich in vielen Anwendungen als sehr effizient, setzt jedoch voraus, daß das Löschen alter Datensätze keine Auswirkungen auf die Korrektheit der zu berechnenden Werte hat.

Als Beispiel für eine solche Situation betrachten wir die Berechnung der Binomialkoeffizienten:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

für $1 \leq k < n$. Ein naives rekursives Schema ist in Algorithmus 38 angegeben. Dieses ist

³⁰J.Lawrence Carter, Mark N. Wegman, Universal classes of hash functions, Journal of Computer and System Sciences, Volume 18, Issue 2, 1979, Pages 143-154.

Algorithmus 38 *BinKoeff(n, k)*

```

IF  $k = 0$  oder  $n = k$  THEN
    return 1
ELSE
     $a := \text{BinKoeff}(n - 1, k);$ 
     $b := \text{BinKoeff}(n - 1, k - 1);$ 
    return  $a + b$ 
FI

```

ungeschickt formuliert, da im allgemeinen viele Rekursionsaufrufe mehrfach stattfinden. Tatsächlich lässt sich ein sehr viel effizienteres Verfahren zur Berechnung der Binomialkoeffizienten mit der Methode des dynamischen Programmierens (siehe Abschnitt 5.5) formulieren. Eine Alternative ist in eine hashbasierte Methode, die in Algorithmus 39 angegeben ist. Diese ist dann empfehlenswert, wenn man nicht nur an einem Binomialkoeffizienten interessiert ist, sondern viele Binomialkoeffizienten benötigt werden, die erst während der Ausführung eines komplexen Programms bekannt werden. Hier verwenden wir \perp als Symbol für „undefiniert“, also $\text{Result}(n, k) = \perp$, falls kein Eintrag in der Computed Table für das Paar (n, k) vorliegt.

Algorithmus 39 *BinKoeff(n, k)*

```

(* Berechnung der Binomialkoeffizienten mit Hilfe einer Computed Table *)
IF Result(n, k) = r ≠ ⊥ THEN
    return r                                (* gebe den Eintrag aus der Computed Table zurück *)
ELSE
    IF k = 0 oder n = k THEN
        Result := 1
    ELSE
        a := BinKoeff(n - 1, k);
        b := BinKoeff(n - 1, k - 1);
        Result := a + b
    FI
    Result(n, k) := Result;           (* Eintrag in die Computed Table *)
    return Result
FI

```

In Algorithmus 39 bietet sich der Einsatz einer zweistelligen Hashfunktion an. Z.B. kann man $h(n, k) = (n + k) \bmod m$ für eine geeignete Konstante m wählen. In der Tat können hier Kollisionsliste beschränkter Länge verwendet werden, da es für die Korrektheit des Verfahrens unerheblich ist, wenn bereits berechnete Ergebnisse gelöscht werden. Dieselbe Idee kann auch für die offene Adressierung eingesetzt werden, indem man die maximal betrachtete Längen b der Sondierungsfolgen vorgibt und nach spätestens nach b Sondierungsschritten abbricht. Noch einfacher ist es, auf Sondierungsfolgen völlig zu verzichten und bis zu b Einträgen pro Zelle vorzunehmen.

4.2 Suchbäume

Die im vorangegangenen Abschnitt diskutierten Hashtechniken ermöglichen ein gutes durchschnittliches Verhalten, unter gewissen Annahmen sogar konstante mittlere Zugriffszeiten, haben jedoch ein schlechtes worst-case Verhalten. Im Gegensatz hierzu zielen die nun vorgestellten Konzepte zu Suchbäumen auf ein kontrolliertes worst-case Verhalten ab. Wie zuvor ist die Ausgangssituation eine Menge von Datensätzen der Form

$$\langle \text{Schlüsselwert}, \text{Information} \rangle,$$

wobei die Schlüsselwerte die Datensätze eindeutig charakterisieren und total geordnet sind. Zur Vereinfachung nehmen wir wieder an, daß die Schlüsselwerte Zahlen mit der natürlichen Ordnung sind. Ferner abstrahieren wir von der Zusatzinformation der Datensätze.

Wir unterscheiden zwei Arten von Suchbäumen:

- binäre Suchbäume: zur Organisation von Datenmengen im Hauptspeicher
- viel verzweigte Suchbäume: zur Organisation von externen Dateien.

In beiden Fällen ist die Höhe des Suchbaums der relevante Faktor für die Laufzeit der Operationen Suchen, Löschen und Einfügen. Deshalb verwendet man gewisse *Balancierungskriterien*, welche die Höhen der Suchbäume kontrollieren.

4.2.1 Binäre Suchbäume und AVL-Bäume

Definition 4.2.1 (Suchbaum). Ein *binärer Suchbaum*, kurz „Suchbaum“, ist ein Binärbaum, dessen Knoten v jeweils mit einem Schlüsselwert $\text{key}(v)$ versehen sind, so daß für alle inneren Knoten v gilt:

- Ist w ein Knoten im linken Teilbaum von v , dann ist $\text{key}(w) < \text{key}(v)$.
- Ist u ein Knoten im rechten Teilbaum von v , dann ist $\text{key}(u) > \text{key}(v)$. □

Um die Datensätze, die in einem Suchbaum dargestellt sind, in aufsteigend sortierter Reihenfolge auszugeben, kann man die Inorder-Durchlaufstrategie (LWR) verwenden.

Implementierung von Suchbäumen. Üblich ist die in Abschnitt 3.2.5 beschriebene Implementierung mit der Darstellung der Knoten durch eine Komponente für den Schlüsselwert, jeweils einen Verweis auf die Söhne und evtl. einen Verweis auf den Vater. Zur Unterstützung von anderen Operationen können zusätzliche Informationen über die Knoten gespeichert werden.

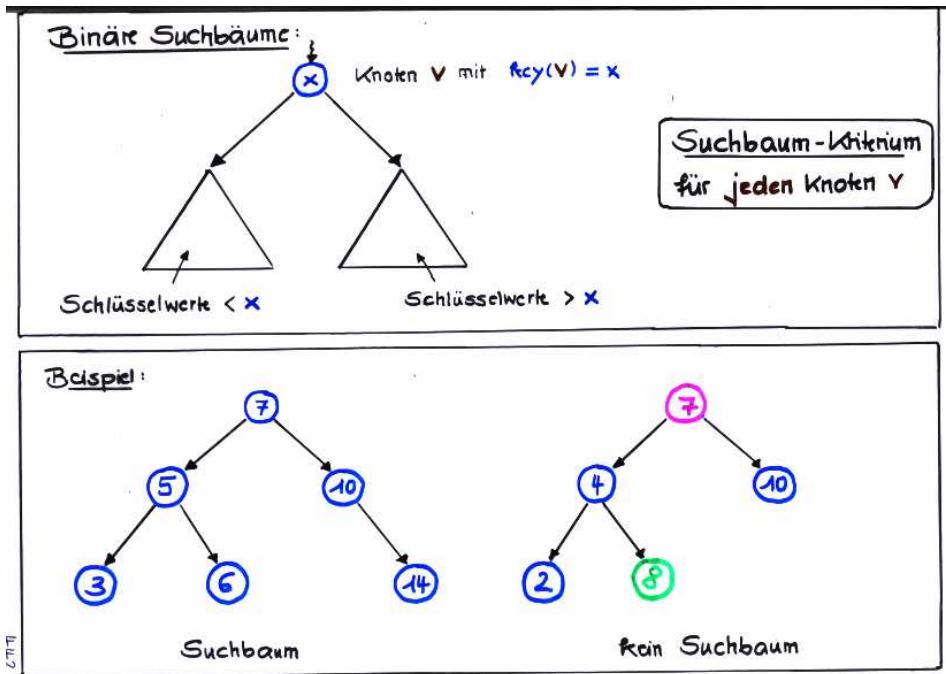


Abbildung 31: Binäre Suchbäume

Operationen auf Suchbäumen. Die Suche eines Schlüsselwerts x erfolgt durch den Funktionsaufruf $\text{Suche}(x, v_0)$, wobei v_0 der Wurzelknoten ist. (Für den leeren Baum ist die Suche sofort „erfolglos“ zu beenden.) Diese basiert auf dem Schema:

- Starte in der Wurzel $v = v_0$.
- Sei v der aktuelle Knoten.
 - Ist $\text{key}(v) > x$, dann suche x im linken Teilbaum von x (falls existent).
 - Ist $\text{key}(v) < x$, dann suche x im rechten Teilbaum von x (falls existent).
 - Ist $\text{key}(v) = x$, dann beende die Suche.

Eine rekursive Formulierung des Verfahrens $\text{Suche}(x, v)$ ist in Algorithmus 40 auf Seite 150 angegeben. Der Algorithmus $\text{Suche}(x, v)$ ist so formuliert, daß stets ein Knoten w zurückgegeben wird.

Für eine erfolgreiche Suche ist $\text{key}(w) = x$. Für eine erfolglose Suche ist w ein Knoten mit höchstens einem Sohn. (Ist z.B. $\text{key}(w) < x$, dann hat w keinen rechten Sohn.) Die Rückgabe des Knotens w wird für das Einfügen benötigt.

Einfügen. Soll der Schlüsselwert x eingefügt werden und ist v derjenige Knoten, in dem die erfolglose Suche nach x geendet hat, dann wird ein neues Blatt w mit dem Schlüsselwert $\text{key}(w) = x$ und dessen Vater gleich v ist, eingefügt. w ist der linke Sohn von v , falls $x < \text{key}(v)$. Andernfalls ist w der rechte Sohn von v . Man beachte, daß v vor

Algorithmus 40 Suche(x, v)

$w := v;$

WHILE die Suche ist noch nicht abgeschlossen **DO**

IF $key(w) = x$ **THEN**

 return „ x liegt auf Knoten w “

FI

IF $key(w) > x$ **THEN**

IF w hat einen linken Sohn **THEN**

$w :=$ linker Sohn von w

ELSE

 return „erfolglose Suche endet in Knoten w “

FI

FI

IF $key(w) < x$ **THEN**

IF w hat einen rechten Sohn **THEN**

$w :=$ rechter Sohn von w

ELSE

 return „erfolglose Suche endet in Knoten w “

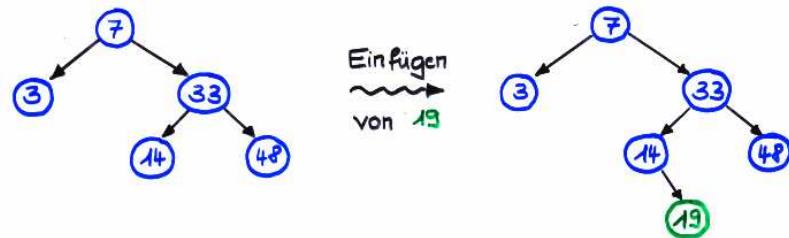
FI

FI

OD

dem Einfügen den betreffenden Sohn nicht hatte, da sonst die Suche fortgesetzt worden wäre. Ferner ist der Spezialfall zu berücksichtigen, bei dem x in den leeren Suchbaum eingefügt wird. In diesem Fall wird ein Blatt w generiert, das mit dem Schlüsselwert x markiert ist und das als Wurzel ausgezeichnet wird. Ein Beispiel zum Einfügen in einen binären Suchbaum ist in Abbildung 32 auf Seite 151 angegeben.

Einfügen in Suchbäumen:



844

Abbildung 32: Beispiel zum Einfügen in einen Suchbaum

Löschen. Beim Löschen eines Schlüsselwerts x , der als Markierung eines Knotens v gefunden wurde, sind drei Fälle zu unterscheiden:

1. Ist v ein Blatt, dann wird v einfach gelöscht.
2. Wenn v genau einen Sohn w hat, dann kann man wie folgt vorgehen:
 - Ist v die Wurzel, dann wird v gelöscht und w zur Wurzel gemacht.
 - Ist u der Vater von v , dann kann man v löschen und die Kante (u, v) durch die Kante (u, w) ersetzen.
3. Wir nehmen an, daß v zwei Söhne hat. Wir bestimmen zunächst denjenigen Knoten w im linken Teilbaum von v , der mit dem nächst kleineren Schlüsselwert markiert ist. Dieser ist der größte Schlüsselwert im linken Teilbaum von v und läßt sich durch die Suche des Schlüsselwerts $x = \text{key}(v)$ im linken Teilbaum von v bestimmen.³¹ Die Markierung $\text{key}(w)$ wird dann zur Markierung von v gemacht und der Knoten w gelöscht. Beachte: der Knoten w hat keinen rechten Sohn, kann also gemäß Fall 1 oder 2 gelöscht werden. Tatsächlich gelöscht werden also stets nur solche Knoten, die höchstens einen Sohn haben.

Abbildung 33 zeigt je ein Beispiel für das Löschen gemäß erstem und zweitem Fall.

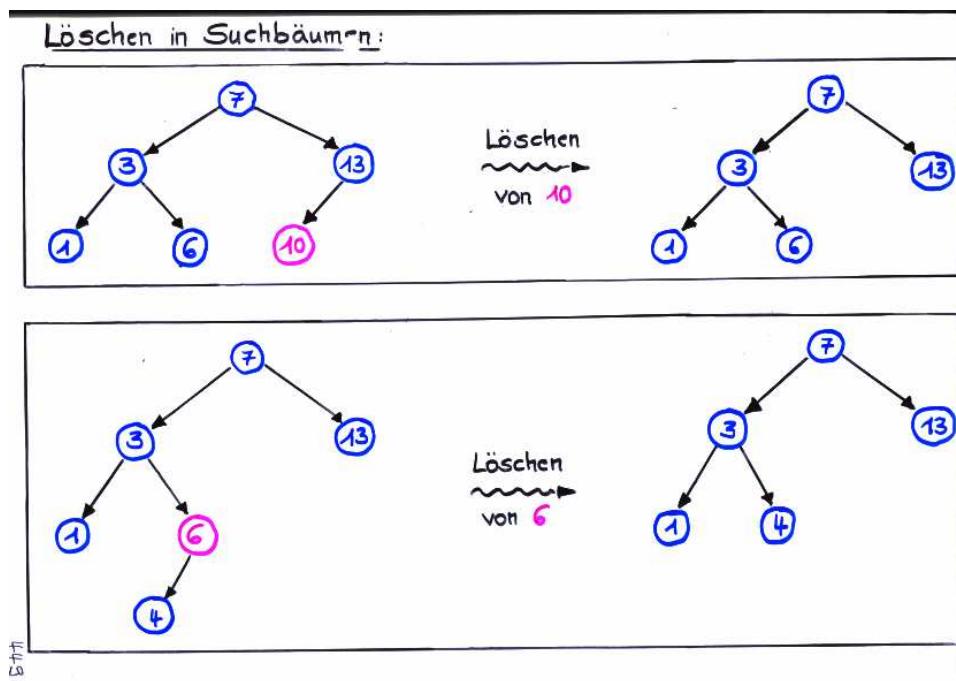


Abbildung 33: Beispiele zum Löschen in einem binären Suchbaum (1. und 2. Fall)

³¹Alternativ kann der Knoten w' mit dem kleinsten Schlüsselwert x' im rechten Teilbaum von v ermittelt werden.

Folgende Beispiele illustrieren den dritten Fall, in dem jeweils der Schlüsselwert 7 der Wurzel gelöscht wird. Im ersten Beispiel (Abbildung 34) wird der Schlüsselwert der Wurzel durch den Schlüsselwert eines Blatts (nämlich 6) ersetzt. Das betreffende Blatt wird also gemäß Fall 1 gelöscht. In Abbildung 35 wird die 7 ebenfalls durch die 6 ersetzt, jedoch liegt der Schlüsselwert 6 auf einem inneren Knoten. Dieser ist gemäß Fall 2 zu löschen.

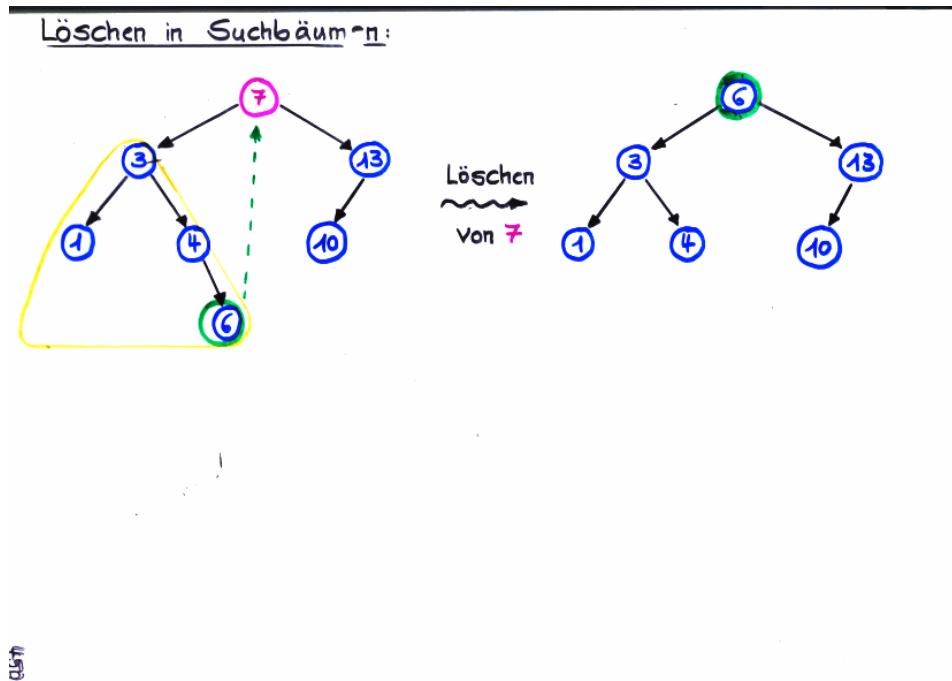
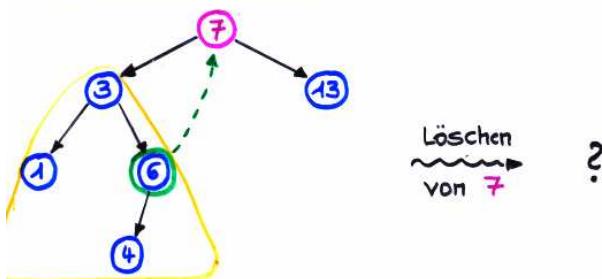


Abbildung 34: Beispiel zum Löschen in einem binären Suchbaum (3. Fall)

Löschen in Suchbäumen



Zwischenschritt:

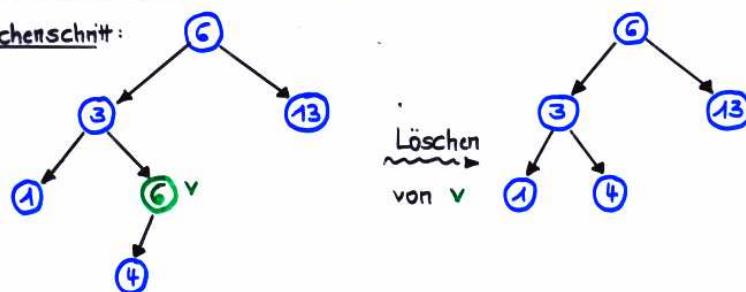


Abbildung 35: Beispiel zum Löschen in einem binären Suchbaum (3. Fall)

Im Wesentlichen wird für alle drei Operationen der Suchbaum auf einem Pfad von der Wurzel bis zu einem Blatt durchlaufen. Damit ergeben sich folgende Kosten:

Satz 4.2.2 (Kosten für Suchen, Einfügen, Löschen in binären Suchbäumen). Die Operationen Suchen, Löschen und Einfügen in einen binären Suchbaum \mathcal{T} lassen sich in Zeit

$$\Theta(\text{Höhe}(\mathcal{T})) = \mathcal{O}(n)$$

durchführen. Dabei ist n die Anzahl der Schlüsselwerte (Knoten) in \mathcal{T} .

Zur Erinnerung: Aus Satz 3.2.19 auf Seite 98 folgt:

$$\lfloor \log n \rfloor \leq \text{Höhe}(\mathcal{T}) \leq n - 1.$$

Extrembeispiel: Der Suchbaum \mathcal{T} , der durch sukzessives Einfügen der Schlüsselwerte $1, 2, \dots, n$ entsteht, ist ein entarteter Baum der Höhe $n - 1$. Die Laufzeit, die zur Konstruktion von \mathcal{T} benötigt wird, ist

$$\Theta\left(\sum_{i=1}^n i\right) = \Theta(n^2).$$

Um derartige Extremfälle, in denen der Suchbaum zu einer Liste entartet, zu verhindern, wendet man Balancierungskriterien an, die garantieren, daß selbst im schlimmsten Fall die Höhe des Suchbaums logarithmisch in der Knoten-/Schlüsselanzahl ist. Wir betrachten hier höhenbalancierte Suchbäume, die nach ihren Entdeckern Adelson, Velskij und Landis benannt sind und kurz als AVL-Bäume bezeichnet werden.

Definition 4.2.3 (Balance). Sei \mathcal{T} ein binärer Suchbaum und v ein Knoten in \mathcal{T} . Weiter sei \mathcal{T}_L der linke und \mathcal{T}_R der rechte Teilbaum des Knotens v . Die Balance in Knoten v ist der Wert

$$bal(v) = \text{Höhe}(\mathcal{T}_L) - \text{Höhe}(\mathcal{T}_R).$$

□

Zur Erinnerung: Die Höhe des leeren Baums wurde als -1 definiert. Falls v ein innerer Knoten ohne linken Sohn ist, so ist \mathcal{T}_L der leere Baum und es gilt $bal(v) = -1 - \text{Höhe}(\mathcal{T}_R)$. Ist v ein Blatt, dann sind die Bäume \mathcal{T}_L und \mathcal{T}_R beide leer und es gilt daher $bal(v) = 0$.

Definition 4.2.4 (AVL-Baum). Ein AVL-Baum ist ein binärer Suchbaum \mathcal{T} , so daß $bal(v) \in \{-1, 0, 1\}$ für jeden Knoten v in \mathcal{T} . □

AVL-Bäume werden wie Suchbäume implementiert, wobei für jeden Knoten v entweder die Höhe des betreffenden Teilbaums oder die Balance in Knoten v als zusätzliche Komponente gespeichert wird.

Satz 4.2.5 (Maximale Höhe von AVL-Bäumen). Sei \mathcal{T} ein AVL-Baum mit n Knoten. Dann gilt:

$$\text{Höhe}(\mathcal{T}) = \Theta(\log n).$$

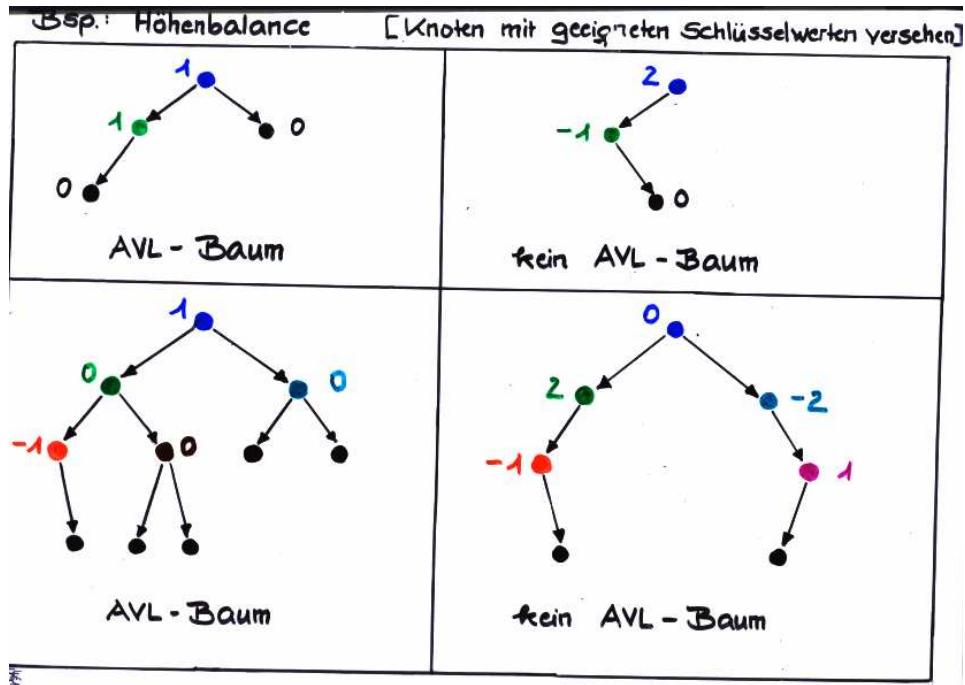


Abbildung 36: Zur AVL-Bedingung

Beweis. Die untere Schranke $\Omega(\log n)$ folgt aus der Tatsache, daß jeder Binärbaum der Höhe h höchstens $1 + 2 + \dots + 2^{h-1} + 2^h = 2^{h+1} - 1$ Knoten hat. (Siehe oben oder Satz 3.2.19.)

Nun zum Nachweis der oberen Schranke $\mathcal{O}(\log n)$ für die Höhe eines AVL-Baums. Hierzu berechnen wir eine untere Schranke für die minimale Anzahl n_h an Knoten, die ein AVL-Baum der Höhe h haben kann.

- Ist $h = -1$, so ist $n_h = n_{-1} = 0$, da nur der leere Baum die Höhe -1 hat.
- Ist $h = 0$, so ist $n_h = n_0 = 1$, da jeder (Binär-)Baum der Höhe 0 aus der Wurzel besteht.
- Ist $h = 1$, so ist offenbar $n_1 = 2$.
- Sei nun \mathcal{T} ein AVL-Baum der Höhe $h \geq 2$. Im Folgenden bezeichne \mathcal{T}_L den linken und \mathcal{T}_R den rechten Teilbaum der Wurzel v_0 von \mathcal{T} .
 - Ist $bal(v_0) = 0$, so haben \mathcal{T}_L und \mathcal{T}_R die Höhe $h - 1$ und \mathcal{T} hat mindestens $2n_{h-1}$ Knoten.
 - Ist $bal(v_0) = -1$, so hat \mathcal{T}_L die Höhe $h - 1$ und \mathcal{T}_R die Höhe $h - 2$. In diesem Fall hat \mathcal{T} mindestens $1 + n_{h-1} + n_{h-2}$ Knoten.
 - Ist $bal(v_0) = 1$, so hat \mathcal{T}_L die Höhe $h - 2$ und \mathcal{T}_R die Höhe $h - 1$. In diesem Fall hat \mathcal{T} mindestens $1 + n_{h-1} + n_{h-2}$ Knoten.

Diese Überlegungen zeigen, daß

$$n_h = 1 + n_{h-1} + n_{h-2}.$$

Durch Induktion nach h kann nun gezeigt werden, daß

$$n_h \geq F_{h+1} \text{ für } h \geq 0,$$

wobei F_h die h -te Fibonacci-Zahl ist. Zur Erinnerung: $F_0 = 0, F_1 = 1, F_h = F_{h-1} + F_{h-2}$ für $h \geq 2$ (siehe Seite 25). Also ist $n_0 = 1 = F_1, n_1 = 2 = F_2$ und für $h \geq 2$:

$$n_h = 1 + n_{h-1} + n_{h-2} \geq 1 + F_h + n_{h-1} > F_{h+1}.$$

Weiter wissen wir, daß die F_h 's exponentiell anwachsen (siehe Lemma 1.3.4 auf Seite 26). Hieraus folgt die Behauptung.

□

Bemerkung 4.2.6 (Fibonacci-Bäume als Vertreter schlechtester AVL-Bäume). Im Beweis von Satz 4.2.5 genügte die Abschätzung $n_h \geq F_{h+1}$. Tatsächlich gilt:

$$n_h = F_{h+3} - 1$$

für alle $h \geq -1$. Beachte, dass $F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2$ und $F_4 = 3$, und somit:

$$\begin{aligned} n_{-1} &= 0 = 1 - 1 = F_2 - 1, \\ n_0 &= 1 = 2 - 1 = F_3 - 1, \\ n_1 &= 2 = 3 - 1 = F_4 - 1 \end{aligned}$$

und im Induktionsschritt:

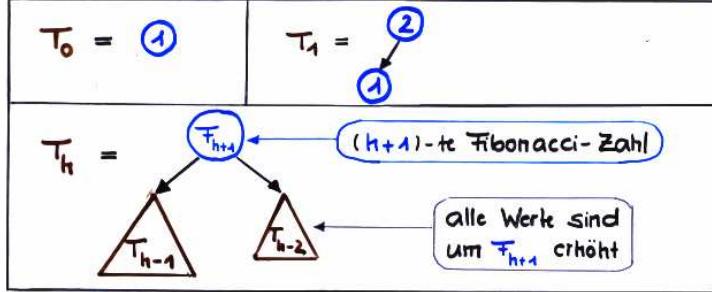
$$n_h = 1 + n_{h-1} + n_{h-2} = 1 + (F_{h+2} - 1) + (F_{h+1} - 1) = F_{h+2} + F_{h+1} - 1 = F_{h+3} - 1$$

Mit der auf Seite 26 angegebenen Formel für F_h erhält man

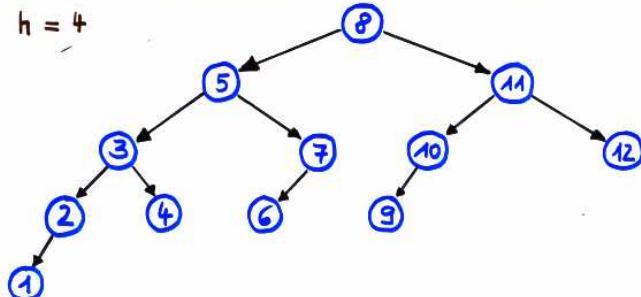
$$\text{Höhe}(\mathcal{T}) \leq 1.44 \cdot \log(n + 2).$$

Die minimale Anzahl n_h an Knoten, die jeder AVL-Baum der Höhe h mindestens haben muß, wird für die so genannten *Fibonacci-Bäume* erreicht. Die Fibonacci-Bäume sind rekursiv definiert wie auf der folgenden Folie angedeutet:

Fibonacci - Bäume : typische Vertreter für die "schlechtesten" AVL - Bäume



z. B. $h = 4$



(* Auf der Folie muß es F_{h+2} statt F_{h+1} heißen. *)

Der Fibonacci-Baum der Höhe $h \geq 2$ ergibt sich also aus einer mit F_{h+2} markierten Wurzel, deren linker Teilbaum der Fibonacci-Baum der Höhe $h - 1$ ist und deren rechter Teilbaum aus dem Fibonacci-Baum der Höhe $h - 2$ entsteht, indem alle Schlüsselwerte um F_{h+2} erhöht werden.

Ist N_h die Anzahl an Knoten im Fibonacci-Baum der Höhe h , so gilt $N_0 = 1$, $N_1 = 2$ und $N_h = 1 + N_{h-1} + N_{h-2}$ für $h \geq 2$. Also erfüllen die N_h 's dieselbe Rekursionsformel wie die n_h 's. Der Fibonacci-Baum der Höhe h hat also genau $N_h = n_h = F_{h+3} - 1$ Knoten, ist also ein „schlechtester“ AVL-Baum der Höhe h . □

Man mache sich an dieser Stelle noch einmal klar, daß lineare (worst case) Laufzeit $\Theta(n)$ für die Suche in einem gewöhnlichen Suchbaum wesentlich dramatischer ist als logarithmische Suchzeit in einem AVL-Baum. Liegen 1 Mio. Datensätze vor, die in einem Suchbaum gespeichert werden, dann erfordert die Suche in einem nicht-balancierten Suchbaum im schlimmsten Fall 1.000.000 Schritte, während in einem AVL-Baum selbst im schlimmsten Fall ca. 30 Schritte ausreichend sind. (Als „Schritt“ bezeichnen wir dabei den Vergleich des gesuchten Schlüsselwerts mit dem Schlüsselwert eines Knotens im Suchbaum.)

Operationen auf AVL-Bäumen. Zunächst können Suchen, Löschen und Einfügen wie in gewöhnlichen Suchbäumen durchgeführt werden, jedoch kann dadurch die Balancebedingung verletzt werden. Derartige Beispiele sind in Abbildung 37 zu sehen.

Nach dem Einfügen bzw. Löschen muß daher geprüft werden, ob Rebalancierungsmaßnahmen notwendig sind. Anschließend muß die Balanceinformation berichtet werden.

Einfügen/Löschen in AVL-Bäume kann die Balance zerstören!

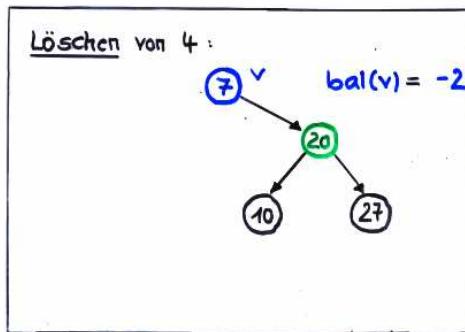
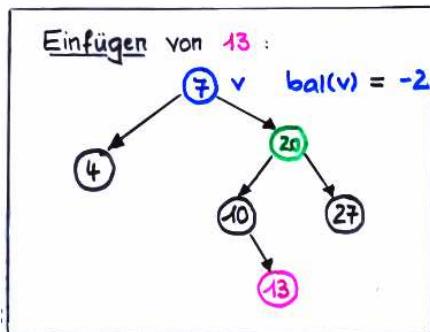
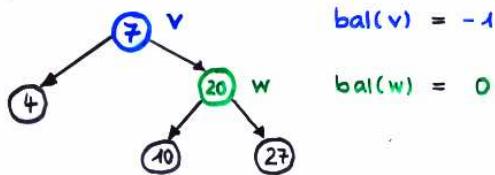


Abbildung 37: Einfügen und Löschen kann die AVL-Balance zerstören

Zur Rebalancierung werden die *einfache Rotation* und die *Doppelrotation* verwendet. Beide beruhen auf lokalen Veränderungen an einem Knoten, an dem nach dem Einfügen oder Löschen die Balance gestört ist, und können in konstanter Zeit durchgeführt werden. Die einfache Rotation und die Doppelrotation können nach links oder nach rechts ausgeführt werden. Abbildung 38 zeigt den Fall einer einfachen Rotation bzw. Doppelrotation nach links, die nach dem Einfügen im rechten Teilbaum oder Löschen im linken Teilbaum erforderlich sein kann. Man beachte, daß der rotierte Baum tatsächlich der Suchbaumbedingung genügt, da z.B. im Falle der einfachen Rotation alle Schlüsselwerte in B zwischen $\text{key}(v)$ und $\text{key}(w)$ liegen, während die Schlüsselwerte in A (bzw. C) kleiner $\text{key}(v)$ (bzw. größer $\text{key}(w)$) sind.

Bottom-Up-Verfahren zur Wiederherstellung der AVL-Balance. Einfache Rotationen oder Doppelrotationen werden stets nur an solchen Knoten v durchgeführt, für welche die Teilbäume von v der AVL-Bedingung genügen. Dies kann dadurch sichergestellt werden, indem die AVL-Bedingung in *Bottom-Up*-Manier wiederhergestellt wird.

Sei v_0, v_1, \dots, v_r der Pfad von der Wurzel v_0 zu dem eingefügten Knoten oder Vater des gelöschten Knotens v_r . Wir laufen nun von v_r rückwärts in Richtung Wurzel v_0 , überprüfen für jeden Knoten v_i die Balanceinformation und führen gegebenenfalls eine geeignete Rotation an Knoten v_i durch. Nachdem die AVL-Balance in dem Teilbaum mit Wurzel v_i wiederhergestellt ist, betrachten wir den Vater v_{i-1} von v_i (vor der Rebalancierung) und führen gegebenenfalls Rebalancierungsmaßnahmen an dem Teilbaum mit Wurzel v_{i-1} durch.

Rebalancierung von AVL - Bäumen: im Bottom - Up - Verfahren

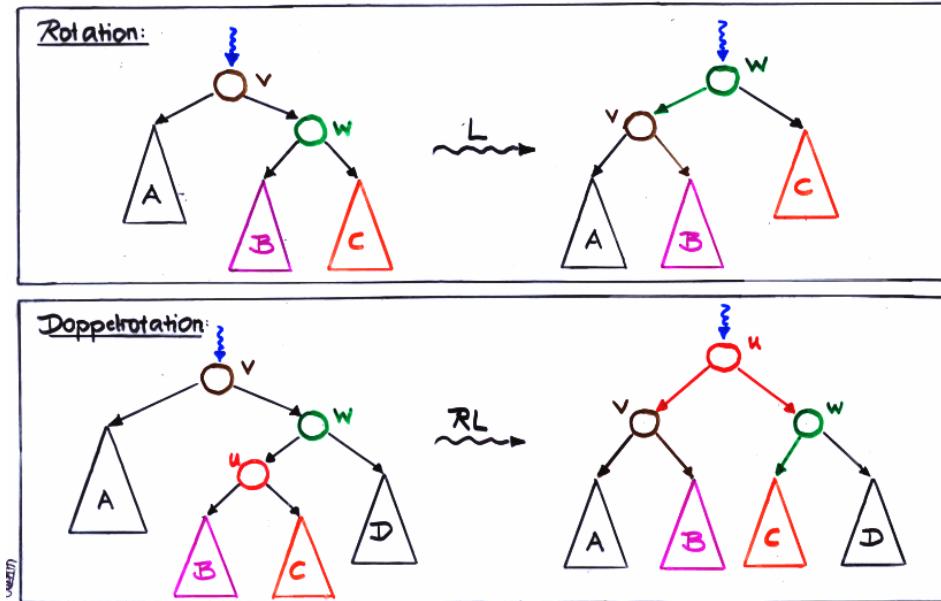
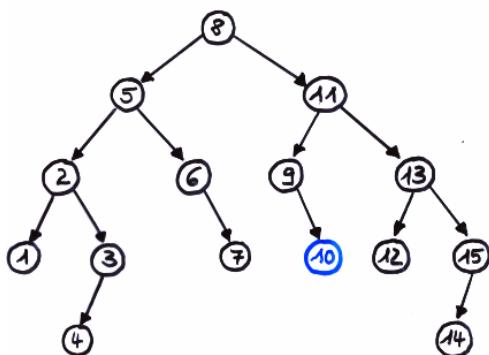


Abbildung 38: Schema der Rotation und Doppelrotation

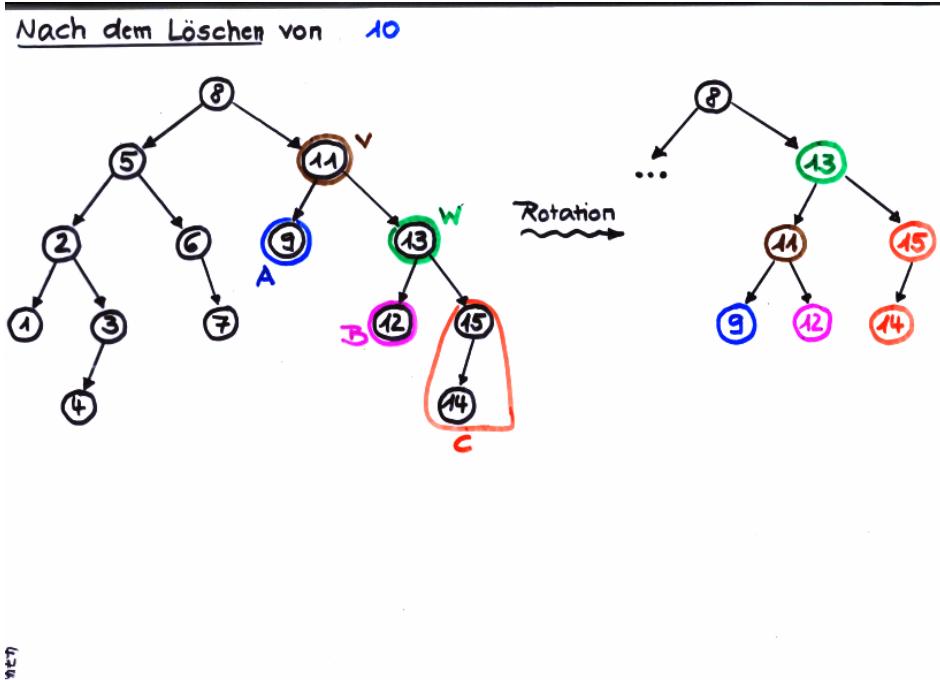
Beispiel 4.2.7 (Rebalancierungen nach dem Löschen). Wir beginnen mit einem Beispiel, welches die Rebalancierungsstufen nach einer Löschoperation illustriert. In folgendem AVL-Baum wird der Knoten mit dem Schlüsselwert 10 gelöscht.

Beispiel: Löschen in AVL - Bäumen



Löschen von 10 zerstört die AVL - Balance am Knoten 11

Wir laufen nun von dem Knoten mit Schlüsselwert 9 (als Vater des gelöschten Knotens) rückwärts zur Wurzel und überprüfen an jedem Knoten die Balance. Der erste Knoten, an dem die Balance gestört ist, ist der Knoten v mit Schlüsselwert 11. Hier führen wir eine einfache Rotation durch:



Da die Balance an dem Wurzelknoten nicht gestört ist, ist die Rebalancierungsmassnahme abgeschlossen. Es ergibt sich also der AVL-Baum in Abbildung 39. \square

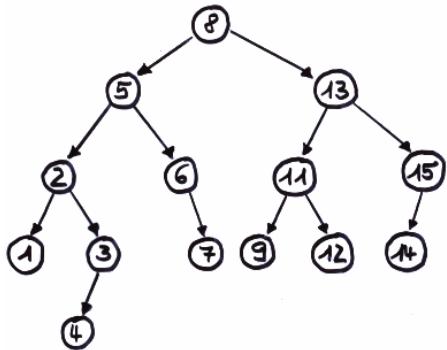
In obigem Beispiel reichte eine Rotation, um die AVL-Bedingung nach dem Löschen wiederherzustellen. Tatsächlich können jedoch mehrere Rebalancierungen notwendig sein. Wir erörtern dies im Beweis des folgenden Satzes:

Satz 4.2.8 (Löschen in AVL-Bäumen). Nach dem Löschen in einem AVL-Baum \mathcal{T} sind mit der skizzierten Bottom-Up-Methode im schlimmsten Fall $\Theta(\text{Höhe}(\mathcal{T}))$ Rotationen notwendig, um die Balance wiederherzustellen.

Beweis. Wie zuvor erläutert, können wir annehmen, daß ein Knoten v vorliegt, an dem die AVL-Balance gestört ist und dessen Teilbäume in AVL-Balance sind. Aus Symmetriegründen genügt es sich auf den Fall zu beschränken, daß im linken Teilbaum von v gelöscht wurde.

Wir diskutieren nun die möglichen Fälle, welche sich aus den Höhen der Teilbäume des rechten Sohns w von v ergeben, und zeigen, daß in allen Fällen eine Rotation ausreicht, um die AVL-Balance im Teilbaum von v wiederherzustellen. Diese Überlegungen begründen die obere Schranke $\mathcal{O}(\text{Höhe}(\mathcal{T}))$ für die Anzahl an durchzuführenden Rotationen. Die untere Schranke $\Omega(\text{Höhe}(\mathcal{T}))$ resultiert aus der zusätzlichen Beobachtung, daß im schlimmsten Fall an jedem Knoten auf dem Rückwärtsweg von dem gelöschten Knoten zur Wurzel rotiert werden muss (siehe unten).

Nach dem Löschen von 10 und der Rotation bei 11



AVL - Balance ist wiederhergestellt !

Für

Abbildung 39: AVL-Baum aus Beispiel 4.2.7 nach der Löschoperation und Rotation

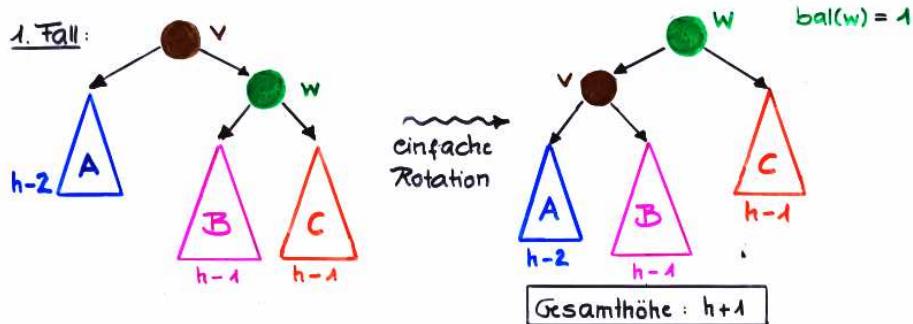
1. Fall: Die Teilbäume von w haben dieselbe Höhe.
2. Fall: Der rechte Teilbaum von w ist höher als der linke Teilbaum.
3. Fall/4. Fall: Der linke Teilbaum von w ist höher als der rechte Teilbaum, wobei sich der dritte und vierte Fall aus den Höhen der Teilbäume des linken Sohns von w ergeben.

In den ersten beiden Fällen führen wir eine einfache Rotation (siehe Abbildungen 40 und 41), im dritten und vierten Fall eine Doppelrotation (siehe Abbildungen 42 und 43) durch.

In dem in Abbildung 40 skizzierten ersten Fall, in welchem die beiden Teilbäume des rechten Sohns w von v dieselbe Höhe haben, ergibt sich also, dass der rotierte Teilbaum dieselbe Höhe hat wie der Teilbaum vor dem Löschen. Da vor dem Löschen die AVL-Bedingung erfüllt war, bleibt die Balance des Vaters von v und dessen Vorgänger unverändert.

In den verbleibenden drei Fällen (Fall 2-4) hat der rebalancierte Teilbaum jeweils die Höhe h , wobei $h+1$ die ursprüngliche Höhe des Teilbaums vor dem Löschen ist. In diesen Fällen ändert sich die Balance des Vaters, so dass an dessen Vater eventuell eine weitere Rebalancierung notwendig ist. Die anfallenden Rebalancierungen können sich also bis zur Wurzel fortsetzen. Im schlimmsten Fall sind daher $\Theta(\text{Höhe}(\mathcal{T}))$ Rebalancierungen notwendig.

Rebalancierung nach dem Löschen:



Situation vor dem Löschen:

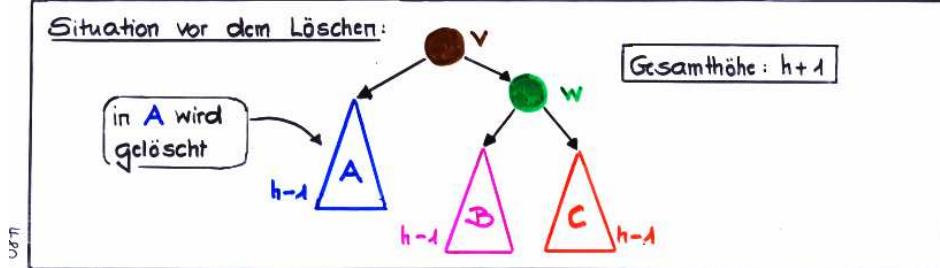
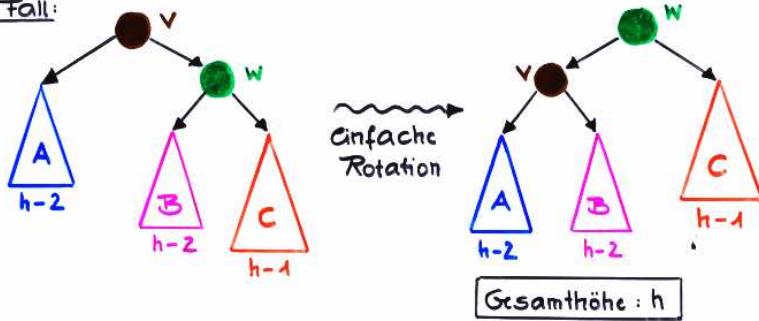


Abbildung 40: 1. Fall im Beweis von Satz 4.2.8

In jedem der vier Fälle kann man den Skizzen entnehmen, daß durch die angegebene Rotation die AVL-Bedingung des betreffenden Teilbaums wiederhergestellt ist. \square

Rebalancierung nach dem Löschen:

2. Fall:



Situation vor dem Löschen:

in A wird gelöscht

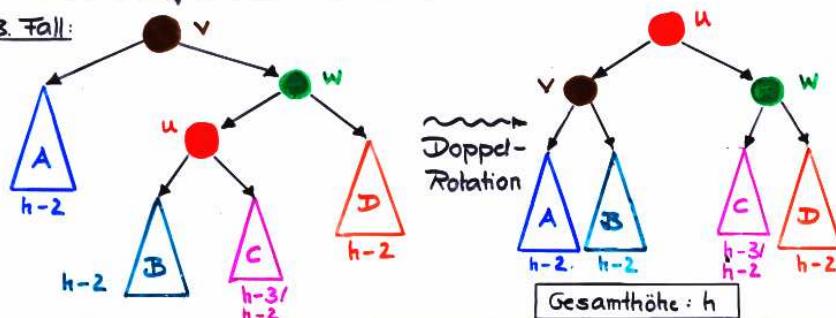
Gesamthöhe : h+1

Höhenreduktion!

Abbildung 41: 2. Fall im Beweis von Satz 4.2.8

Rebalancierung nach dem Löschen:

3. Fall:



Situation vor dem Löschen:

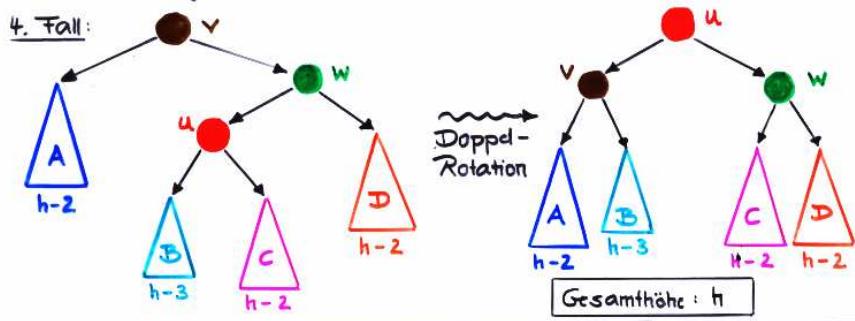
Gesamthöhe : h+1

Höhenreduktion!

Abbildung 42: 3. Fall im Beweis von Satz 4.2.8

Rebalancierung nach dem Löschen

4. Fall:



Situation vor dem Löschen:

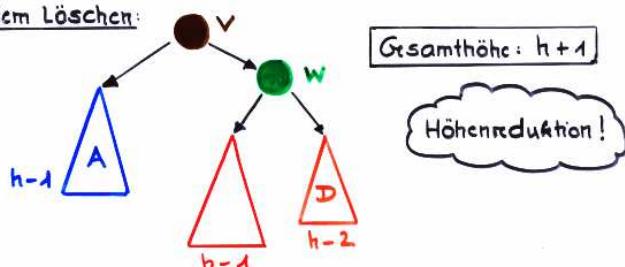


Abbildung 43: 4. Fall im Beweis von Satz 4.2.8

Ein Beispiel für einen AVL-Baum, bei dem nach dem Löschen an mehr als einem Knoten Rebalancierungsmaßnahmen notwendig sind, ist ein Fibonacci-Baum der Höhe $h \geq 4$, in dem das rechteste Element (der größte Schlüsselwert) gelöscht wird.

Nun zu den Rebalancierungen nach dem Einfügen eines Schlüsselwerts. Auch hier laufen wir von dem neu eingefügten Blatt in Richtung Wurzel und überprüfen an jedem Knoten, ob die Balance gestört ist. Im Gegensatz zum Löschen ist hier jedoch höchstens eine Rebalancierung auszuführen:

Satz 4.2.9 (Einfügen in AVL-Bäumen). Nach dem Einfügen in einen AVL-Baum \mathcal{T} ist stets eine einfache Rotation oder Doppelrotation ausreichend, um die Balance wiederherzustellen.

Beweis. Wir betrachten die möglichen Fälle, wobei wir uns auf den Fall beschränken, daß in den rechten Teilbaum des Knotens v eingefügt wurde. Der Knoten v ist der tiefste Knoten, an dem die Balance gestört ist. Wir zeigen nun, daß stets eine Rebalancierung genügt, die den Teilbaum mit Wurzel v in einen AVL-Baum überführt, dessen Höhe mit der ursprünglichen Höhe des Teilbaums übereinstimmt. Für die echten Vorgänger des Knotens v bleibt die Balance also unverändert. Sei w der rechte Sohn von v .

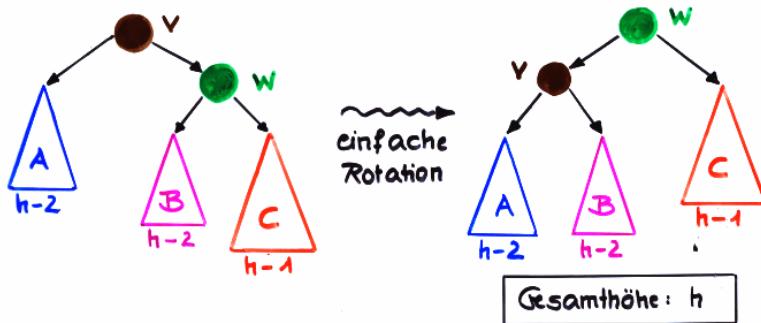
1. Fall: Der rechte Teilbaum von w ist höher als der linke Teilbaum von w . In diesem Fall reicht eine einfache Rotation an v , siehe Abbildung 44 auf Seite 167.

2./3. Fall: Der linke Teilbaum von w ist höher als der rechte Teilbaum von w . In diesem Fall führe wir eine Doppelrotation aus. Um uns klar zu machen, dass der resultierende Teilbaum von v seine ursprüngliche Höhe hat, betrachten wir den linken Sohn u von w und unterscheiden, ob im linken oder rechten Teilbaum von u eingefügt wurde. Siehe Abbildungen 45 und 46.

Den Skizzen ist zu entnehmen, dass in allen drei Fällen der Teilbaum mit Wurzel v nach der (Doppel-)Rotation wieder dieselbe Höhe wie vor dem Einfügen hat, so dass an den Vorgänger von v keine weiteren Rebalancierungen (oder Korrektur der Balanceinformation) erforderlich sind.

Man beachte, dass diese drei Fälle ausreichend sind, da wir hier davon ausgehen, dass die AVL-Balance erst durch das Einfügen eines Elements gestört wurde. Daher ist es nicht möglich, dass beide Teilbäume von w dieselbe Höhe haben. \square

Rebalancierung nach dem Einfügen: (1. Fall)



Situation vor dem Einfügen:

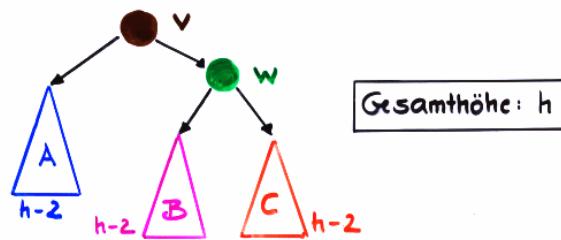
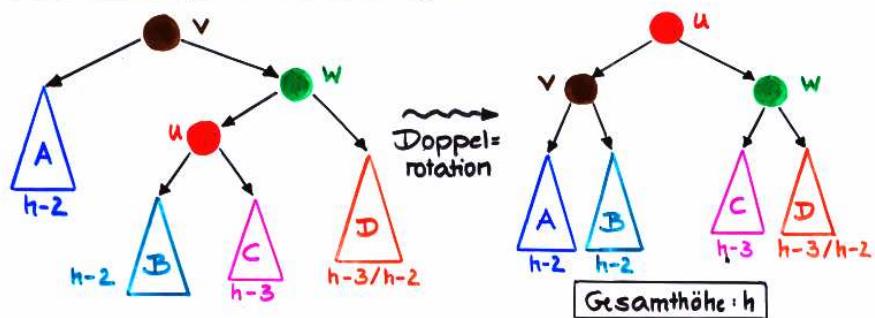


Abbildung 44: 1. Fall im Beweis von Satz 4.2.9

Rebalancierung nach dem Einfügen: (2. Fall)



vor dem Einfügen

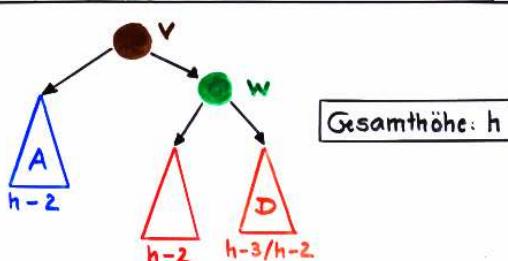


Abbildung 45: 2. Fall im Beweis von Satz 4.2.9

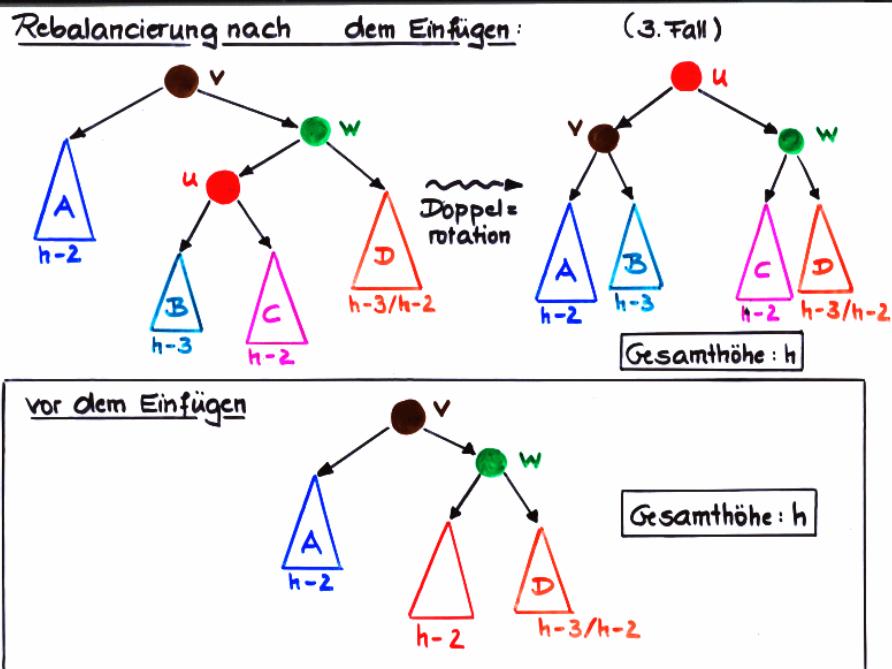
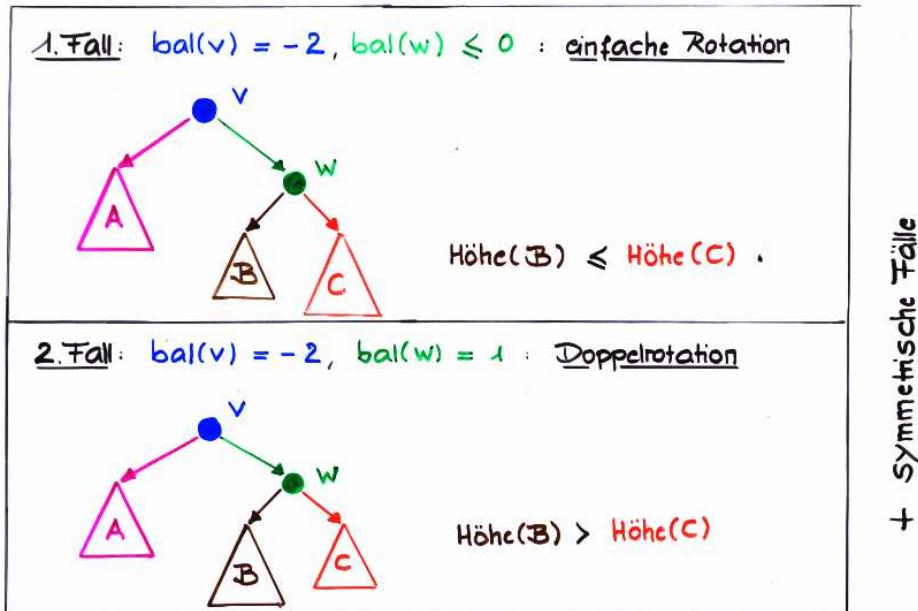


Abbildung 46: 3. Fall im Beweis von Satz 4.2.9

Rebalancierungskriterien. Die in den Beweisen zu Satz 4.2.8 und 4.2.9 erörterten Fälle zeigen, welche Strategie beim Rebalancieren angewandt werden muß. Sei v derjenige Knoten, an dem rebalanciert werden muß.

- Ist $\text{bal}(v) = -2$ und $\text{bal}(\text{right}(v)) \in \{-1, 0\}$, dann ist eine einfache Rotation nach links auszuführen.
- Ist $\text{bal}(v) = 2$ und $\text{bal}(\text{left}(v)) \in \{1, 0\}$, dann ist eine einfache Rotation nach rechts auszuführen.
- Ist $\text{bal}(v) = -2$ und $\text{bal}(\text{right}(v)) = 1$, dann ist eine Doppelrotation nach links auszuführen.
- Ist $\text{bal}(v) = 2$ und $\text{bal}(\text{left}(v)) = -1$, dann ist eine Doppelrotation nach rechts auszuführen.

„Strategie“ für die Auswahl der Rebalancierungsart:



Wir fassen die Ergebnisse nochmals zusammen. Nach dem eigentlichen Einfügen bzw. Löschen (wie in gewöhnlichen Suchbäumen) muß die Balanceinformation berichtigt werden und eventuell rebalanciert werden.³² Hierzu sind die Knoten $v_0, v_1, \dots, v_r = v$, auf dem Weg von der Wurzel v_0 zu dem Knoten v , der eingefügt oder dessen Sohn gelöscht wurde, in umgekehrter Reihenfolge zu durchlaufen. Sobald für einen Knoten v_j die Höhe des betreffenden Teilbaums unverändert blieb, können die Vorgängerknoten v_{j-1}, \dots, v_0 vernachlässigt werden, da sich deren Balance nicht geändert hat. Beim Einfügen ist dies stets nach der Ausführung einer Rotation der Fall.

³²Selbstverständlich kann sich die Balance auch an solchen Knoten ändern, an denen nach dem Einfügen oder Löschen keine Rebalancierung nötig ist.

Es bleibt zu klären, wie man den Rückwärtsweg von dem eingefügten Knoten oder Vater des gelöschten Knotens findet. Die einfachste Möglichkeit besteht in der Verwendung von zusätzlichen Verweisen auf den jeweiligen Vater. Eine platzeffizientere Implementierungsmöglichkeit besteht darin, bei der vorangegangenen Suche die Verweise von v_{j-1} zu dem (linken oder rechten) Sohn v_j umzudrehen. Ist etwa v_{j+1} der rechte Sohn von v_j , so kann man den „rechten“ Zeiger von v_j temporär auf v_{j-1} (statt v_{j+1}) umlenken. Zusätzlich benötigt man ein Bit, aus welchem abgelesen werden kann, welcher der beiden Zeiger auf den Vater verweist.

Satz 4.2.10 (Kosten für Suchen, Löschen, Einfügen in AVL-Bäumen). Suchen, Löschen und Einfügen in einen AVL-Baum mit n Knoten können in Zeit $\mathcal{O}(\log n)$ durchgeführt werden.

Blattsuchbäume

Für die bisher betrachteten Suchbäume gingen wir davon aus, daß jeder Knoten einen Datensatz bestehend aus einem Schlüsselwert und gewisser Zusatzinformation repräsentiert. Eine Variante von Suchbäumen sind sogenannte Blattsuchbäume, bei denen die Schlüsselwerte mit den zugehörigen Informationen über die Datensätze nur in den *Blättern* gespeichert werden, während die inneren Knoten mit Werten versehen sind, die lediglich als „Pfadinformation“ dienen. Üblich ist es, die inneren Knoten mit dem kleinsten Schlüsselwert eines Blatts des rechten Teilbaums zu beschriften. Anhand dieser Information kann der Suchalgorithmus in Analogie zu gewöhnlichen Suchbäumen vorgenommen werden.

Blattsuchbäume eignen sich vor allem, wenn *Bereichsanfragen*, etwa

„Bestimme alle Schlüsselwerte x mit $a \leq x \leq b$ “

durchgeführt werden sollen. Hierzu wird eine Doppelverkettung der Blätter verwendet.

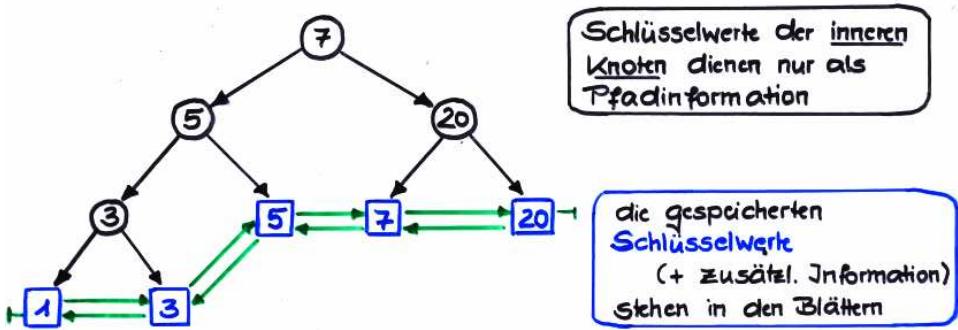
Definition 4.2.11 (Binärer Blattsuchbaum). Ein (binärer) Blattsuchbaum ist ein binärer Baum \mathcal{T} , in dessen Blätter Datensätze (mit Schlüsselwerten) abgelegt sind und in dem jeder Knoten mit einem Schlüsselwert markiert ist, so daß für jeden inneren Knoten v gilt:

- v hat genau zwei Söhne.
- $key(v_L) < key(v) \leq key(v_R)$ für alle Knoten v_L im linken Teilbaum von v und alle Knoten v_R im rechten Teilbaum von v .
- $key(v) = \min \{key(w) : w \text{ ist ein Blatt im rechten Teilbaum von } v\}$.

□

Die Forderung, daß alle inneren Knoten genau zwei Söhne haben, verhindert redundante innere Knoten und stellt linearen Platzbedarf sicher. Liegt nämlich ein Blattsuchbaum \mathcal{T} zur Repräsentation von n Datensätzen vor, so hat \mathcal{T} genau n Blätter und genau $n - 1$ innere Knoten. Die Gesamtanzahl an Knoten ist daher $2n - 1$.

Blatt - Suchbäume



Implementierung: ähnlich wie für gewöhnl. Suchbäume
mit **Doppelverkettung** der Blätter

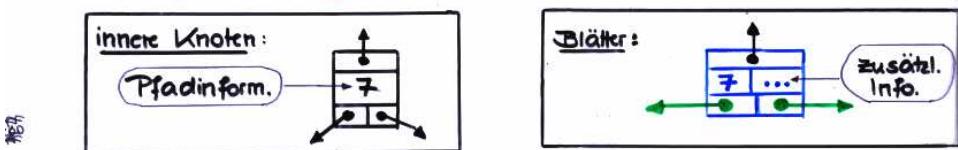


Abbildung 47: Blattsuchbäume

Suchen, Einfügen und Löschen können mit ähnlichen Algorithmen wie für gewöhnliche Suchbäume realisiert werden. Eine Ausformulierung der Verfahren wird als Übungsaufgabe gestellt. Beispiele sind in Abbildungen 48 und 49 angegeben. Im Gegensatz zu gewöhnlichen Suchbäumen können in Blattsuchbäumen stets nur Blätter gelöscht werden, da nur diese „echte“ Datensätze enthalten. Ferner ist beim Einfügen und Löschen zu beachten, daß die Pfadinformation auf dem Pfad von der Wurzel zu dem eingefügten Knoten bzw. Vater des gelöschten Knotens aktualisiert werden muss.

Um die Kosten für Suchen, Einfügen und Löschen logarithmisch zu beschränken, können Balancierungskriterien wie für gewöhnliche Suchbäume eingesetzt werden.

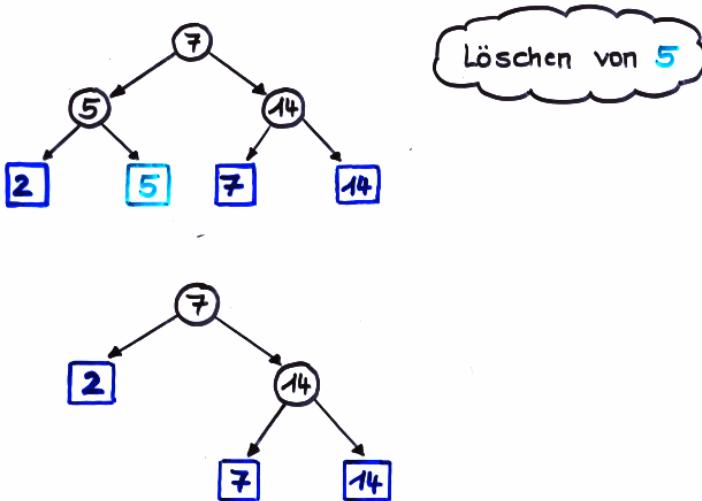
Definition 4.2.12 (AVL-Blattsuchbaum). Ein binärer Blattsuchbaum heißt AVL-Blattsuchbaum (kurz AVL-Blattbaum), falls für alle inneren Knoten v gilt:

$$Höhe(\mathcal{T}_L) - Höhe(\mathcal{T}_R) \in \{0, 1, -1\},$$

wobei \mathcal{T}_L der linke Teilbaum von v und \mathcal{T}_R der rechte Teilbaum von v ist. \square

Die Operationen Suchen, Einfügen und Löschen in AVL-Blattsuchbäumen lassen sich mit ähnlichen Algorithmen wie in gewöhnlichen AVL-Bäumen realisieren und beruhen auf einer Traversierung des Baums entlang eines Pfads sowie eventuellen Rebalancierungen mit einfachen Rotationen sowie Doppelrotationen in Bottom-Up-Manier. Die Höhenabschätzung $Höhe(\mathcal{T}) = \Theta(\log n)$ gilt gleichermaßen für AVL-Bäume und AVL-Blattsuchbäume. Wir erhalten daher:

Löschen in Blatt-Suchbäumen



264

Abbildung 48: Einfügen in einem Blattsuchbaum

Satz 4.2.13 (Kosten für AVL-Blattsuchbäume). Die Operationen Suchen, Löschen und Einfügen können in einen AVL-Blattbaum mit n Schlüsselwerten in Zeit $\mathcal{O}(\log n)$ durchgeführt werden.

Bereichsanfragen können in Zeit $\mathcal{O}(\log n + k)$ beantwortet werden, wobei k die Anzahl an Blättern ist, deren Schlüsselwert in dem betreffenden Bereich liegt.

Einfügen in Blatt-Suchbäume

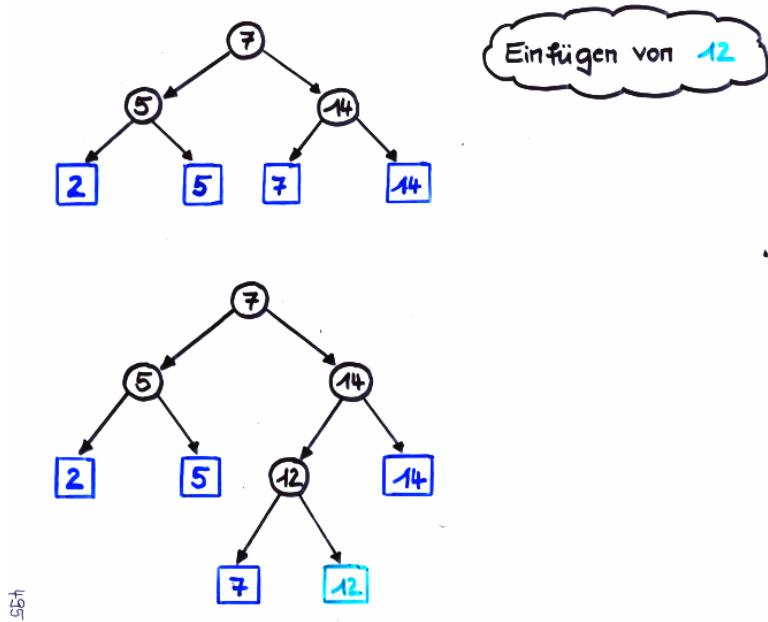


Abbildung 49: Löschen aus einem Blattsuchbaum

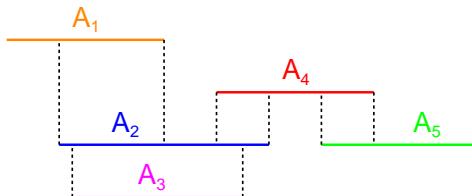
Beispiel: Das Sichtbarkeitsproblem für achsenparallele Liniensegmente

Sichtbarkeitsprobleme befassen sich mit der Frage, welche Objekte von anderen Objekten sichtbar sind. Ein typisches Sichtbarkeitsproblem der Computer-Graphik stellt sich bei der Erstellung einer wirklichkeitsgetreuen Ansicht eines Raums. Diese muß berücksichtigen, welche Objekte vom Standort sichtbar sind und welche Objekte durch andere verdeckt sind. Eine andere klassische Instanz von Sichtbarkeitsproblemen tritt beim Chip-Entwurf auf, bei dem einander sichtbare Schaltelemente möglichst kompakt (aber unter Einhaltung gewisser Abstandsbedingungen) platziert werden müssen.

Wir beschränken uns hier auf eine sehr einfache Variante von Sichtbarkeitsproblemen. Gegeben seien n horizontale Liniensegmente

$$A_i = [a_i, b_i] \times \{y_i\} = \{(x, y_i) : a_i \leq x \leq b_i\}, \quad i = 1, \dots, n.$$

Wir nehmen zur Vereinfachung an, daß die Werte $a_1, \dots, a_n, b_1, \dots, b_n$ paarweise verschieden sind. Gesucht sind alle Segmentpaare (A_i, A_j) , die sich gegenseitig sehen können, d.h. alle Paare (A_i, A_j) mit $i \neq j$ und für die es Punkte $s = (x, y_i) \in A_i$, $s' = (x, y_j) \in A_j$ gibt, so daß die Verbindungsstrecke zwischen s und s' kein anderes Segment schneidet.



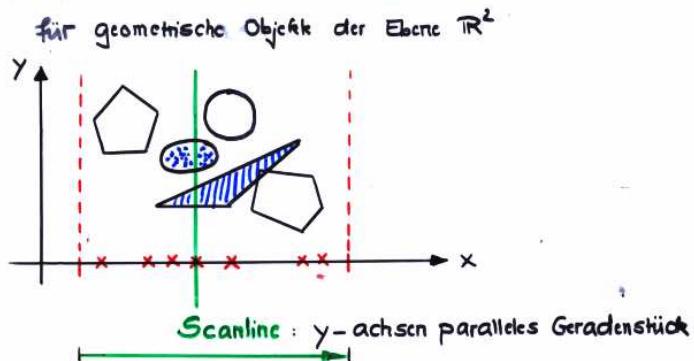
In dem Beispiel oben sind (A_1, A_2) , (A_2, A_3) , (A_3, A_4) und (A_4, A_5) Sichtbarkeitspaare. Im Folgenden behandeln wir die Paare (A_i, A_j) als ungeordnete Paare, d.h. wir identifizieren (A_i, A_j) mit (A_j, A_i) . Der Aufwand des naiven Verfahrens, das für alle Paare (A_i, A_j) prüft, ob A_i von A_j sichtbar ist, hat die Laufzeit $\Omega(n^2)$. Wir stellen jetzt ein besseres auf dem so genannten *Scanline-Prinzip* beruhendes Verfahren mit der Laufzeit $\mathcal{O}(n \log n)$ vor.

Scanline-Algorithmen (häufig auch Plane-Sweep-Algorithmen genannt) basieren auf einer Art „Sortierung“ der zu untersuchenden geometrischen Objekte. Diese Sortierung ergibt sich durch die Reihenfolge, in welcher die Objekte von einer sogenannte *Scanline*, die z.B. von links nach rechts über die Ebene wandert, geschnitten werden.³³

Objekte, die von der Scanline getroffen werden, werden auch *aktiven Objekte* genannt. Zur Verwaltung der jeweils aktiven Objekte wird oftmals eine sortierte Liste oder ein Suchbaum (in unserem Fall ein AVL-Blattsuchbaum) verwendet. Die Dynamik der Scanline wird algorithmisch durch eine Auswahl von *Haltepunkten* modelliert, die für die Momente stehen, an denen aktive Objekte stillgelegt oder nichtaktivierte Objekte aktiviert werden.

³³In dem hier vorgestellten Algorithmus ist die Scanline eine zur y -Achse parallele Gerade, die von links nach rechts wandert. Selbstverständlich kann man auch x -achsenparallele Geraden oder Geraden mit anderer Steigung verwenden. In einigen Fällen ist auch ein um einen Punkt rotierender Strahl als Scanline sinnvoll.

Das Scanline-Prinzip:



Grundidee:

Analysiere die **aktiven Objekte** (d.h. Objekte, die von der Scanline getroffen werden)

wobei die Scanline nur an ausgewählten **Haltepunkten** betrachtet wird.

Abbildung 50: Zur Idee des Scanline-Prinzips

Zurück zum Sichtbarkeitsproblem. Die Scanline ist eine y -achsenparallele Gerade mit den Haltepunkten

$$Q = \{a_1, b_1, \dots, a_n, b_n\},$$

die in einer aufsteigend sortierten Liste verwaltet werden. Die jeweils aktiven Liniensegmente A_i werden durch einen AVL-Blattsuchbaum \mathcal{T} dargestellt, wobei die y -Koordinate als Schlüsselwert für A_i angesehen wird, d.h. $key(A_i) = y_i$. Die Segmente A_i sind also durch die Blätter in \mathcal{T} repräsentiert.

Wir nennen A_j den *unteren Nachbar* von A_i in \mathcal{T} , falls gilt:

- A_j und A_i sind in \mathcal{T} repräsentiert
- $y_j < y_i$
- es gibt kein in \mathcal{T} repräsentiertes Segment A_k mit $y_j < y_k < y_i$.

In diesem Fall wird A_i der *obere Nachbar* von A_j in \mathcal{T} genannt. Für jeden Haltepunkt $q \in Q$ wird ein Segment A_i in \mathcal{T} eingefügt oder aus \mathcal{T} gelöscht.

- Wird das Segment A_i am Haltepunkt $q = a_i$ in \mathcal{T} eingefügt und hat A_i in \mathcal{T} einen oberen Nachbarn A^+ , dann sind A_i und A^+ voneinander sichtbar. Entsprechendes gilt für den unteren Nachbarn A^- von A_i (falls existent).
- Wird das Segment A_i am Haltepunkt $q = b_i$ aus \mathcal{T} gelöscht und hat A_i einen oberen und einen unteren Nachbarn A^+ bzw. A^- in \mathcal{T} unmittelbar vor dem Löschen, dann sind A^+ und A^- (an einem Punkte $y > b_i$) voneinander sichtbar.

Der Algorithmus für das Sichtbarkeitsproblem ist in Algorithmus 108 auf Seite 444 formuliert. Ein Beispiel ist in Abbildung 51 angegeben.

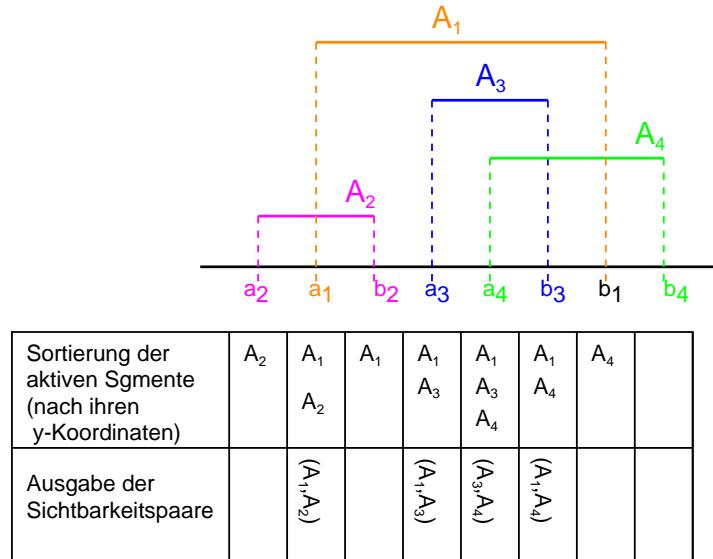


Abbildung 51: Beispiel zu Algorithmus 108

Laufzeit. Wie zuvor sei n die Anzahl an gegebenen Liniensegmenten. Die Sortierung der $2n$ Haltepunkte der Scanline kann in $\Theta(n \log n)$ Schritten vorgenommen werden. Die Zugriffe auf den AVL-Blattsuchbaum \mathcal{T} verursachen jeweils die Kosten $\mathcal{O}(\log n)$. Da insgesamt jeder der $2n$ Haltepunkte betrachtet wird, ergeben sich die Kosten $\mathcal{O}(n \log n)$ für alle Zugriffe (Einfügen, Löschen) auf \mathcal{T} . Die Suche nach unteren bzw. oberen Nachbarn kann bei einer Doppelverkettung der Blätter in konstanter Zeit ausgeführt werden. Damit ergeben sich die Gesamtkosten $\Theta(n \log n)$.

Algorithmus 41 Scanline-Algorithmus zum Bestimmen der Sichtbarkeitspaare

$Q :=$ sortierte Liste der Haltepunkte a_i, b_i der Scanline;

(* Die aktiven Liniensegmente werden in einem AVL-Blattsuchbaum verwaltet *)
(* (sortiert nach den y -Koordinaten). *)

$\mathcal{T} := \emptyset;$ (* Initialisierung des leeren AVL-Blattsuchbaums *)

WHILE $Q \neq \emptyset$ **DO**

$q :=$ nächster Haltepunkt der Scanline;

IF $q = a_i$ **THEN**

füge A_i in \mathcal{T} ein; (* A_i wird aktiviert *)

IF A_i hat einen oberen Nachbarn A^+ in \mathcal{T} **THEN**

gib (A_i, A^+) als Sichtbarkeitspaar aus

FI

IF A_i hat einen unteren Nachbarn A^- in \mathcal{T} **THEN**

gib (A_i, A^-) als Sichtbarkeitspaar aus

FI

FI

IF $q = b_i$ **THEN**

IF A_i hat einen oberen und unteren Nachbarn A^+ bzw. A^- in L **THEN**

gib das Sichtbarkeitspaar (A^+, A^-) aus

FI

entferne A_i aus \mathcal{T} ;

(* A_i wird stillgelegt *)

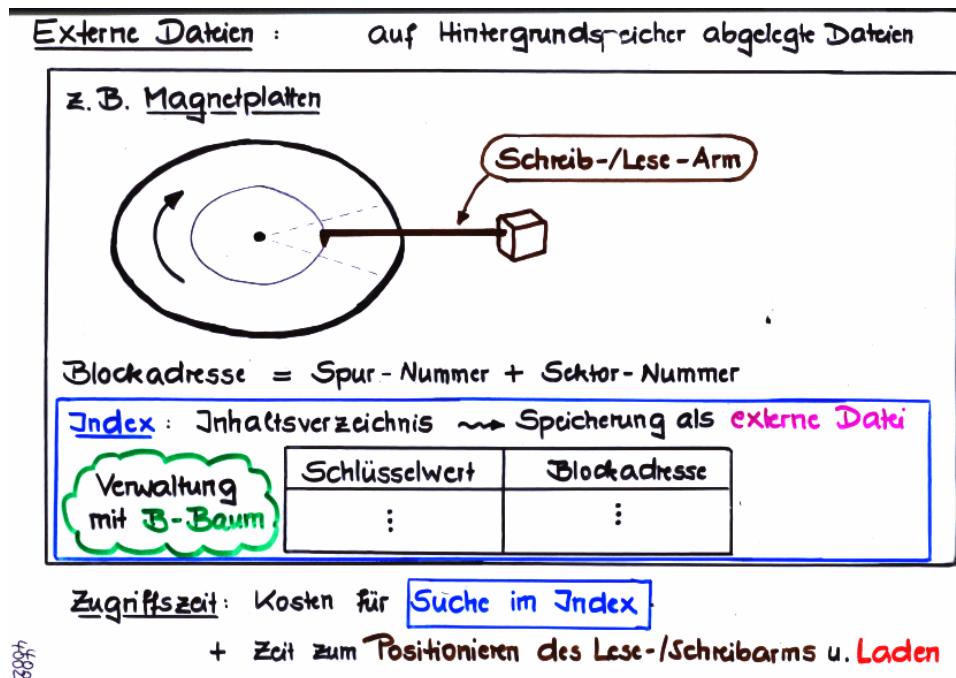
FI

OD

4.2.2 B-Bäume

B-Bäume (Abkürzung für die nach ihrem Erfinder benannten Bayer-Bäume) sind eine Datenstruktur zur Verwaltung externer Dateien, die auf einem Hintergrundspeicher mit wahlfreiem Zugriff (Magnetplatten oder Disketten) abgelegt sind. Wie zuvor nehmen wir an, daß die Datensätze mit Schlüsselwerten versehen sind, welche die Datensätze eindeutig charakterisieren.

Typischerweise werden B-Bäume zur Darstellung von *Indexen* verwendet. Ein Index für eine Datei kann als Stichwortverzeichnis interpretiert werden, in dem alle in der Datei vorkommenden Schlüsselwerte zusammen mit einem Vermerk, an welcher Stelle sich der Datensatz mit dem betreffenden Schlüsselwert befindet, aufgelistet sind. Der Index enthält für jeden Datensatz der Datei einen Eintrag *(Schlüsselwert, Adresse)*.



Die „Adresse“ bezieht sich auf den Block, in dem der Datensatz gespeichert ist (also der Nummer der Spur und der Nummer des Sektors). In vielen Anwendungen ist selbst der Index zu groß, um als ganzes im Hauptspeicher verwaltet werden zu können. Der Index wird dann – wie die Datei selbst – extern gespeichert. Die Suche nach einem zu einem gegebenen Schlüsselwert gehörenden Datensatz basiert auf folgendem Schema:

- Ein möglichst großer Ausschnitt des Index wird in den Hauptspeicher geladen.
- Der Schlüsselwert wird mit einem Hauptspeicherverfahren in dem geladenen Indexausschnitt gesucht.
- Falls der Schlüsselwert gefunden wurde, wird der betreffende Block geladen und auf den Datensatz zugegriffen. Andernfalls wird der nächste Ausschnitt des Index geladen, etc.

Die Indexausschnitte entsprechen den Knoten des B-Baums. Welcher Indexausschnitt als nächstes geladen werden muß, ergibt sich aus der Suche im zweiten Schritt des soeben skizzierten Schemas.

Der Zeitaufwand für die Hauptspeicheraktionen ist verglichen mit den Zugriffen auf die Magnetplatte oder Diskette (Positionieren des Lese-/Schreibkopfes und Laden des Blocks in den Hauptspeicher) marginal. Die interne Suche im Indexausschnitt kann bei geeigneter Verwaltung der Seiten (z.B. AVL-Bäume) bis zu 10.000-mal schneller sein als die Zugriffe auf den Hintergrundspeicher.

Definition 4.2.14 (B-Baum der Ordnung m). Sei m eine ganze Zahl ≥ 1 . Ein *B-Baum* der Ordnung m ist ein gerichteter Baum \mathcal{T} vom Grad $2m+1$ mit folgenden Eigenschaften. Ist \mathcal{T} nicht leer, dann gilt:

1. Jeder Knoten außer eventuell der Wurzel hat zwischen m und $2m$ Schlüssel.
2. Der Wurzelknoten v_0 hat mindestens einen und höchstens $2m$ Schlüssel.
3. Ist v ein innerer Knoten mit k Schlüsseln, dann hat v genau $k+1$ Söhne. Sind x_1, \dots, x_k die Schlüssel eines inneren Knotens v mit $x_1 < \dots < x_k$, dann sind die Söhne v_0, \dots, v_k von v so geordnet, daß gilt:

$$x_i < x < x_{i+1} \text{ für alle Schlüsselwerte } x \text{ im Teilbaum } \mathcal{T}_{v_i}, i = 0, 1, \dots, k.$$

Dabei setzen wir $x_0 = -\infty$ und $x_{k+1} = +\infty$.
4. Alle Blätter haben dieselbe Tiefe, d.h. sind gleich weit von der Wurzel entfernt.

□

Die Knoten eines B-Baums entsprechen den Seiten/Blöcken des Index, können also im ganzen in den Hauptspeicher geladen werden. Die obere Schranke $2m$ sollte so gewählt sein, daß der zur Verfügung stehende Hauptspeicherplatz annähernd genutzt wird. Die untere Schranke m garantiert dann eine (in etwa) 50% Speicherplatzausnutzung. Die Sonderrolle der Wurzel, die weniger als m Schlüsselwerte haben darf, erklärt sich aus den Einfüge- und Löschoperationen, die wir später erläutern.

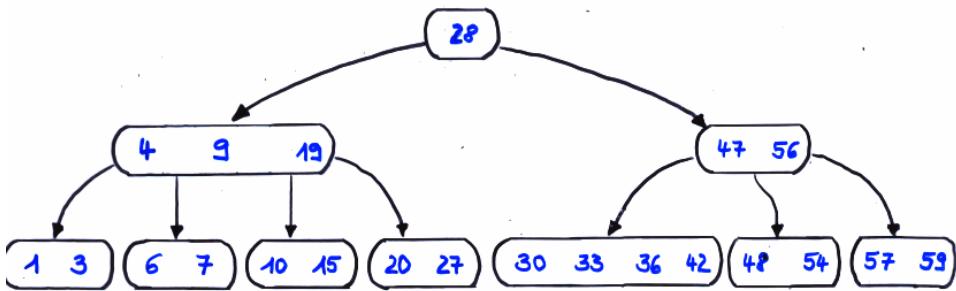
Die Ordnung m ist in Abhängigkeit der Schlüsselwerte und Seitengröße zu wählen. Typische Werte für m sind $50 \leq m \leq 2.000$.

Das Konzept von B-Bäumen ist nicht auf die Repräsentation von Indexen beschränkt, auch wenn dies die Hauptanwendung ist. Tatsächlich können auch die Datensätze selbst (anstelle deren Adressen) mit den Schlüsselwerten in den Knoten abgelegt werden. Im allgemeinen sind jedoch die Adressen wesentlich kürzer als die Datensätze, so dass durch den Einsatz eines Index eine größere Ordnung m möglich wird.

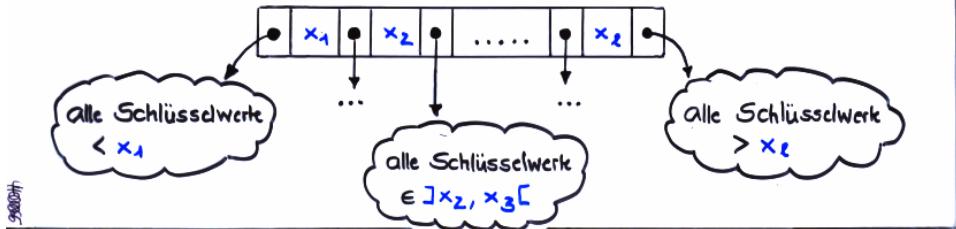
Ein Beispiel für einen B-Baum der Ordnung $m = 2$ ist auf folgender Folie angegeben. Hier sowie im Folgenden stellen wir nur die Schlüsselwerte, nicht aber die zugehörige Information, dar:

B-Bäume

(Ordnung $m = 2$)



Beachte: die **Ordnung** der Söhne der inneren Knoten ist wichtig!



Höhe von B-Bäumen. In B-Bäumen stellt die letzte Bedingung (alle Blätter haben dieselbe Tiefe) eine gewisse Ausgeglichenheit des B-Baums sicher und garantiert logarithmische Höhe.

Ist \mathcal{T} ein B-Baum der Ordnung m mit N Schlüsselwerten und der Höhe h , dann gilt:

$$2(m+1)^h - 1 \leq N \leq (2m+1)^{h+1} - 1$$

und somit

$$\log_{(2m+1)}(N+1) - 1 \leq h \leq \log_{(m+1)}\left(\frac{N+1}{2}\right).$$

Die obere Schranke $(2m+1)^{h+1} - 1$ für die Anzahl an Schlüsselwerten wird einem B-Baum \mathcal{T} der Ordnung m und Höhe h erreicht, falls jeder Knoten in \mathcal{T} genau $2m$ Schlüsselwerte hat. Es liegt dann ein vollständiger Baum der Höhe h vom Verzweigungsgrad $2m+1$ vor. Dieser hat

$$1 + (2m+1) + \dots + (2m+1)^h = \frac{(2m+1)^{h+1} - 1}{(2m+1) - 1} = \frac{(2m+1)^{h+1} - 1}{2m}$$

Knoten (siehe Satz 3.2.19 auf Seite 98). Multipliziert man die Knotenanzahl mit dem Faktor $2m$ (Anzahl an Schlüsselwerten pro Knoten), so erhält man die maximale Anzahl $(2m+1)^{h+1} - 1$ an Schlüsselwerten.

Die untere Schranke $2(m+1)^h - 1$ an Schlüsselwerten wird für einen minimal ausgelasteten B-Baum der Höhe h erreicht. Dessen Wurzel hat nur einen Schlüsselwert und somit zwei Söhne. Jeder andere Knoten hat genau m Schlüsselwerte. Die beiden Teilbäume der

Wurzel sind also vollständige Bäume der Höhe $h - 1$ und vom Verzweigungsgrad $m + 1$. Sie haben also jeweils genau

$$1 + (m + 1) + \dots + (m + 1)^{h-1} = \frac{(m + 1)^h - 1}{(m + 1) - 1} = \frac{(m + 1)^h - 1}{m}$$

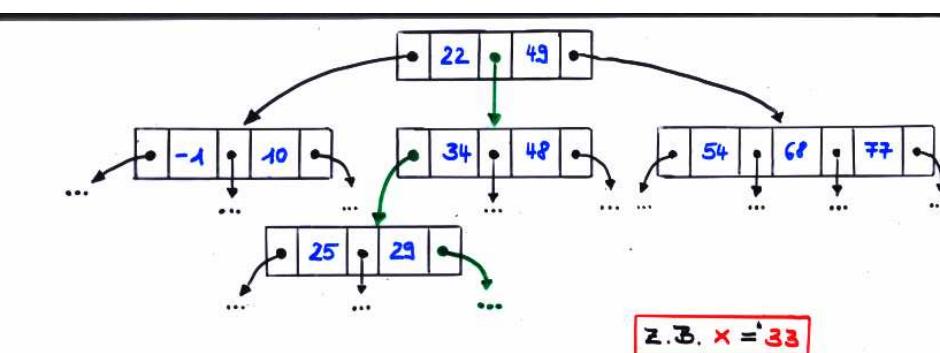
Knoten und somit genau $(m+1)^h - 1$ Schlüsselwerte. Die Gesamtanzahl an Schlüsselwerten in diesem minimal ausgelasteten B-Baum beträgt also $1 + 2((m+1)^h - 1) = 2(m+1)^h - 1$. Zahlenbeispiel: Für $m = 1.000$ und $h = 2$ ist die Anzahl an Schlüsselwerten

$$\geq 2(m+1)^h - 1 = 2 \cdot 1.001^2 - 1 > 2.000.000.$$

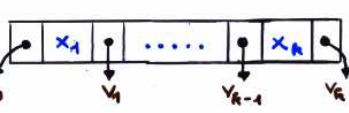
Operationen auf B-Bäumen. Suchen, Einfügen und Löschen können ähnlich wie in binären Suchbäumen ausgeführt werden. Anstelle von Rotationen werden hier jedoch andere Techniken eingesetzt:

- Einfügen mit eventueller Overflowbehandlung (Splitting)
- Löschen mit eventueller Underflowbehandlung (Balance oder Concatenation)

Suchen. Der Suchprozess für Schlüsselwert x ist auf der Folie unten skizziert. Wir beginnen in der Wurzel und suchen den Schlüsselwert zunächst innerhalb des Wurzelknotens. Ist die Suche nicht erfolgreich, so gibt sie Aufschluß in welchem Teilbaum die Suche fortzuführen ist. Wir laden den betreffenden Sohn in den Hauptspeicher und suchen wieder intern nach dem gegebenen Schlüsselwert. Das Verfahren wird solange fortgesetzt bis der Schlüsselwert entweder gefunden oder ein Blatt erreicht ist.



Suche nach einem Schlüsselwert x

- * starte in der Wurzel
- * für jeden besuchten Knoten $v =$ 
 - falls $x \in \{x_1, \dots, x_k\}$: STOP
 - falls der Knoten v ein Blatt ist, dann bestimme i mit $x_i < x < x_{i+1}$ und gehe zu dem i -ten Sohn des Knotens

Für den Suchvorgang ist die Anzahl an Seitenzugriffen durch

$$Höhe(\mathcal{T}) + 1 \leq \log_{(m+1)}((N+1)/2) + 1$$

nach oben geschränkt. Werden die Seiten als AVL-Bäume organisiert, dann kann der interne Aufwand pro Knoten, welcher durch das Suchen innerhalb der Seite (Knoten) entsteht, durch $\mathcal{O}(\log(2m)) = \mathcal{O}(\log m)$ abgeschätzt werden.

Algorithmus 42 Einfügen eines Datums d mit Schlüsselwert x in einen B-Baum \mathcal{T}

IF der Baum ist leer **THEN**

generiere eine Wurzel mit dem Datum d

ELSE

suche x in \mathcal{T} ;

IF x nicht gefunden **THEN**

sei v dasjenige Blatt von \mathcal{T} , in dem die Suche endet;

füge d auf v ein;

IF v hat $2m+1$ Schlüsselwerte **THEN**

$OVERFLOW(v)$

FI

ELSE

return „Datum mit Schlüsselwert x schon vorhanden“

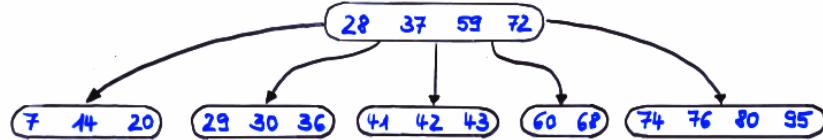
FI

FI

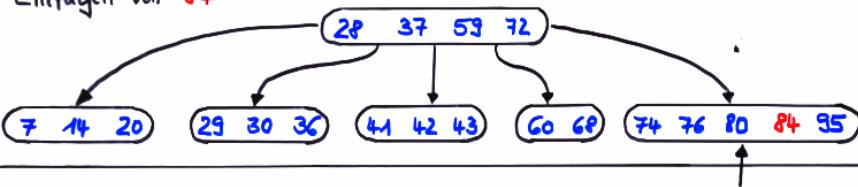
Einfügen mit Overflowbehandlung. Dem Einfügen (siehe Algorithmus 42) geht stets eine Suche nach dem betreffenden Schlüsselwert voraus. Diese endet stets in einem Blatt, sofern der betreffende Schlüsselwert noch nicht vorkommt. Der neue Datensätze wird zunächst auf dem durch die Suche erreichten Blatt eingefügt. Falls dadurch ein Blatt mit $2m+1$ Schlüsselwerten entsteht, ist eine Reorganisation (Overflowbehandlung) notwendig. Dies wird durch ein Splitting realisiert, in dem der betreffende Knoten v in zwei Knoten mit je m Schlüsseln zerlegt wird und der mittlere Schlüsselwert in dem Vater von v eingefügt wird. Auf diese Weise kann sich das Splitting bis zur Wurzel fortsetzen.

Wir betrachten ein Beispiel für das Einfügen, das eine Overflowbehandlung erfordert:

Einfügen in einen B-Baum um der Ordnung $m = 2$



Einfügen von 84



OVERFLOW

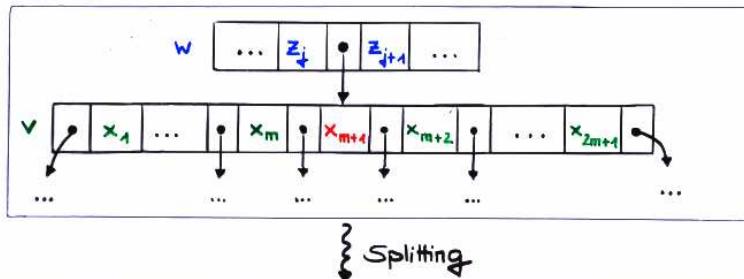
jetzt $2m+1 = 5$ Schlüssele

tRÖ

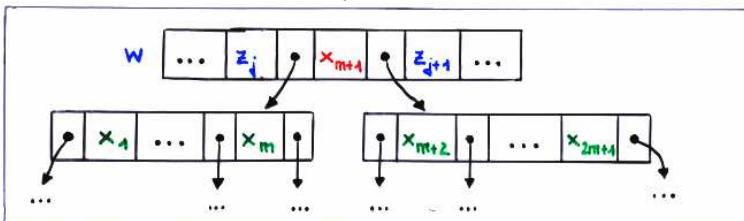
Wir erläutern nun die Overflowbehandlung, deren Aufgabe es ist, nach dem Einfügen den entstandenen Baum so zu reorganisieren, daß ein B-Baum entsteht. Die Ausgangssituation ist ein Knoten v , der genau $2m + 1$ Schlüsselwerte hat. Falls v nicht die Wurzel ist, dann zerlegen wir v in zwei neue Knoten mit jeweils m Schlüsselwerten und fügen den „mittleren“ Schlüsselwert von v in den Vater von v ein.

OVERFLOW(v)

1. Fall: v ist nicht die Wurzel. Sei w der Vater von v .



Splitting



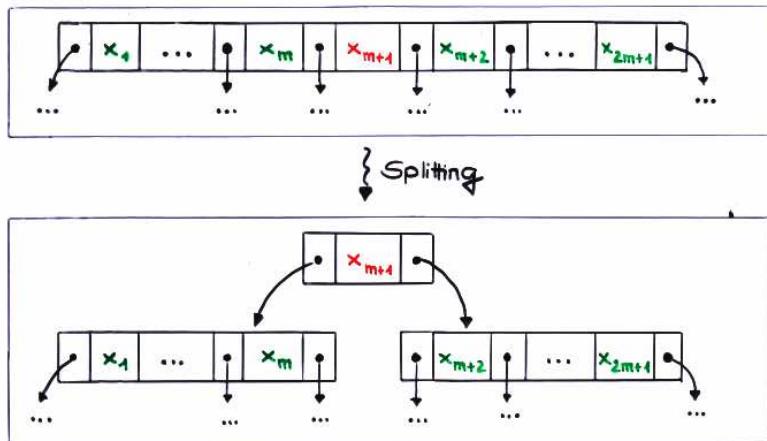
Falls w jetzt $2m+1$ Schlüssele hat, dann OVERFLOW(w).

tRÖ

Der zweite Fall betrifft das Splitting der Wurzel. In diesem Fall erhöht sich die Höhe des Baums. Es entsteht eine neue Wurzel mit nur einem Schlüsselwert.

OVERTLOW(v)

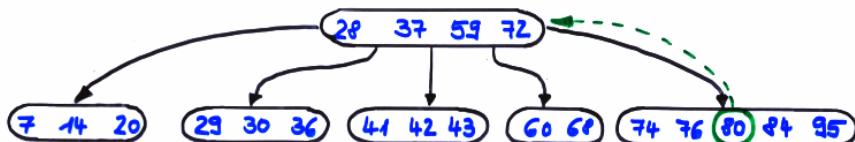
2. Fall: v ist die Wurzel



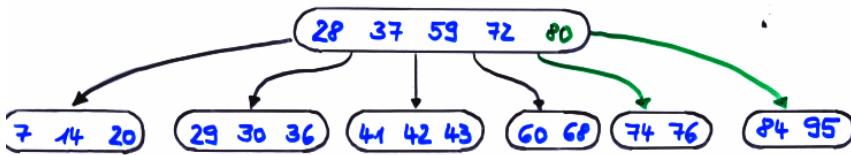
STRÜPF

Folgende Folien zeigen ein Beispiel zur Overflow-Behandlung:

Nach dem Einfügen von 84



Splitting: Restrukturierungsmaßnahme bei "Overflow"

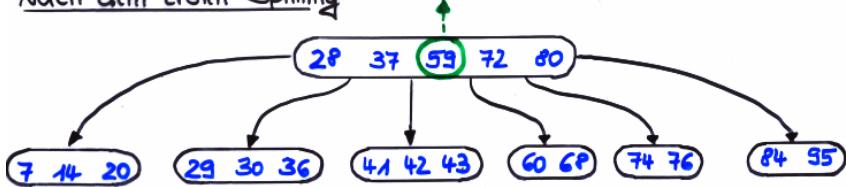


Overflow am Wurzelnoden!

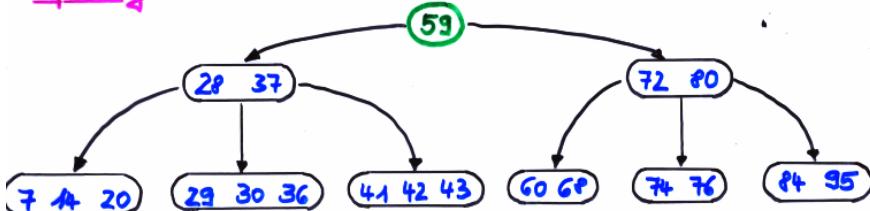
Also: Splitting des Wurzelnodens

STRÜPF

Nach dem ersten Splitting



Splitting der Wurzel:



Restrukturierungsmaßnahmen abgeschlossen.

✓

Löschen mit Underflowbehandlung. Das Löschen eines Datensatzes mit vorgegebenem Schlüsselwert x (siehe Algorithmus 43) geschieht ähnlich wie in binären Suchbäumen. Befindet sich der zu löschende Datensatz d auf einem Blatt, so kann er einfach gelöscht werden. Andernfalls wird der Datensatz d durch denjenigen Datensatz d' ersetzt, der mit dem nächstgrößeren Schlüsselwert x' versehen ist. Dieser Datensatz d' liegt auf einem Blatt und kann durch eine Suche nach x gestartet mit der Wurzel desjenigen Teilbaums, der sich unmittelbar rechts von d befindet, gefunden werden. Anschließend löscht man d' aus dem Blatt. Den folgenden Löschalgorithmus haben wir so formuliert, daß im zweiten Fall die Datensätze d und d' vertauscht werden. (Nach diesem Vertauschungsschritt liegt d auf einem Blatt v' . Der Vertauschungsschritt entfällt, falls sich der betreffende Datensatz d auf einem Blatt $v = v'$ befindet.) Anschließend wird d aus dem Blatt v' entfernt.

Durch das Entfernen eines Datensatzes aus dem Blatt v' , kann ein Knoten mit $m - 1$ Schlüsselwerten entstehen. Ist v' nicht der Wurzelknoten, dann ist eine Underflowbehandlung notwendig, welche die B-Baum-Bedingung wiederherstellt. Weiter gibt es noch den Sonderfall, daß v' der Wurzelknoten ist, der vor dem Löschen nur den Datensatz d (Schlüsselwert x enthielt) und nach dem Löschen keinen Schlüsselwert mehr hat. In diesem Fall ist der resultierende B-Baum der leere Baum (d.h. der Wurzelknoten v' ist zu eliminieren.)

Ein Beispiel für das Löschen:

Algorithmus 43 Löschen eines Datums d mit Schlüsselwert x in einen B-Baum \mathcal{T}

Suche denjenigen Knoten v , der x enthält;

IF v ist ein Blatt **THEN**

$v' := v$

ELSE

bestimme den nächstgrößeren gespeicherten Schlüsselwert x' sowie das Blatt v' , welches x' enthält;

vertausche die zu x und x' gehörenden Datensätze

FI

(* Nun liegt der zu dem Schlüsselwert x gehörende Datensatz d auf dem Blatt v' . *)

lösche Datensatz d aus v' ;

IF $v' \neq$ Wurzelknoten und v' hat $m - 1$ Schlüsselwerte **THEN**

$UNDERFLOW(v')$

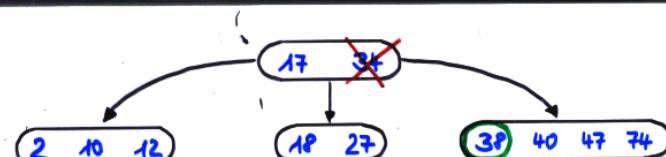
ELSE

IF $v' =$ Wurzelknoten und v' hat keinen Datensatz mehr **THEN**

vermerke, dass \mathcal{T} leer ist

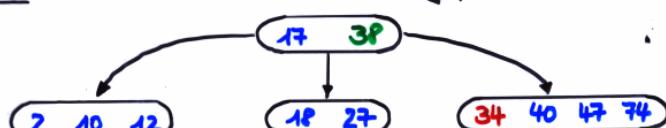
FI

FI

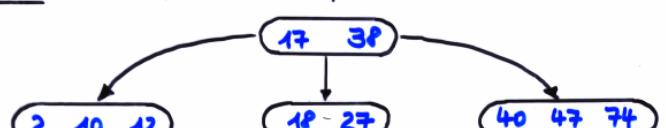


Löschen von 34:

1. Schritt: Vertausche 34 mit dem nächstgrößeren Wert. Hier: 38



2. Schritt: Löschen von 34 auf dem Blatt

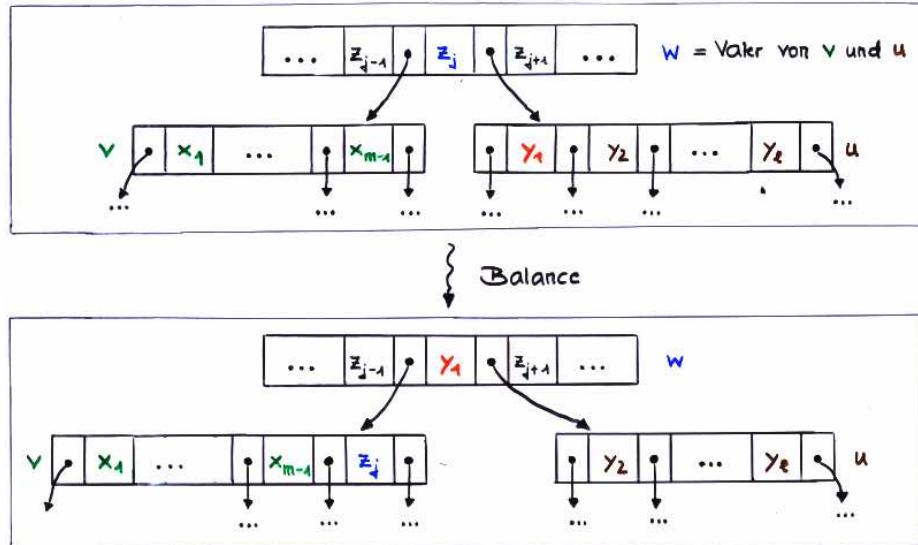


Zur Underflowbehandlung gibt es zwei Möglichkeiten, die unter den englischen Stichworten *Balance* (Ausgleich mit einem Bruder) oder *Concatenation* (Zusammenlegen von Seiten) bekannt sind. Der Ausgleich mit einem Bruderknoten (Balance) hat den Vorteil, daß

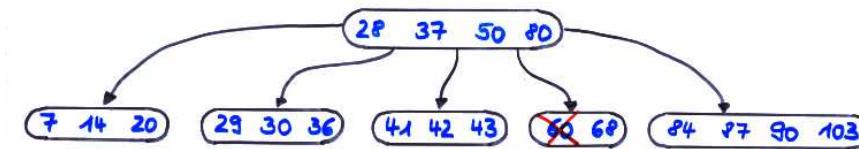
sich die Schlüsselanzahl des Vaterknotens nicht verändert.³⁴

UNDERFLOW(v)

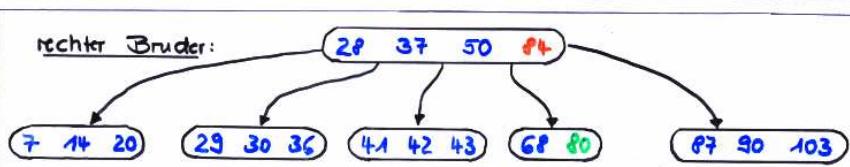
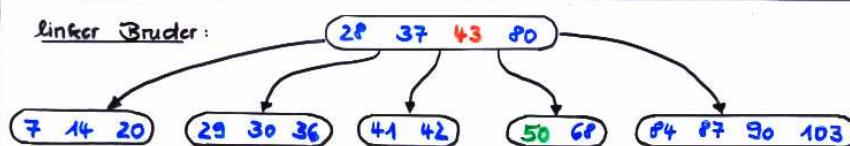
1. Fall: v hat einen benachbarten Bruder u mit $\ell \geq m+1$ Schlüsseln, z.B.



Wir betrachten ein Beispiel für den Ausgleich mit einem Bruder.



Löschen von 60: Ausgleich mit linkem und rechten Bruder möglich

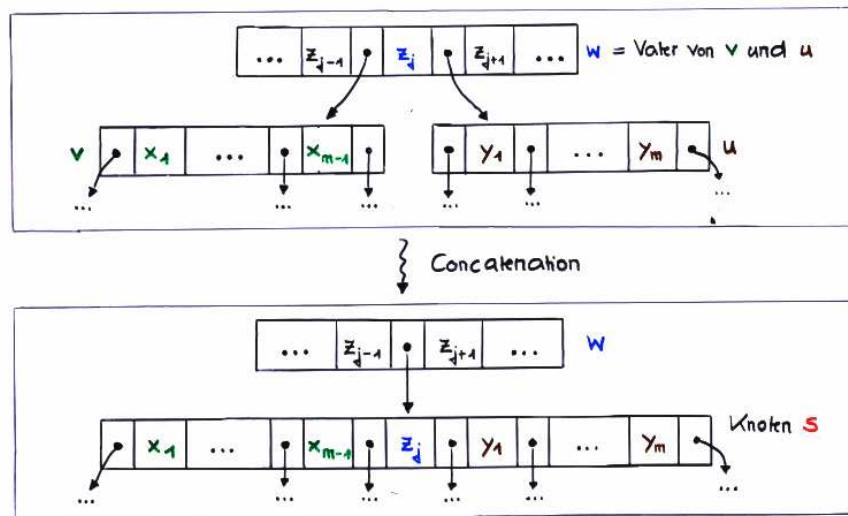


³⁴Der Ausgleich kann auch als Overflowbehandlung eingesetzt werden, vorausgesetzt der Knoten mit $2m + 1$ Schlüsselwerten hat einen Bruder, der weniger als $2m$ Schlüsselwerte hat.

Das Zusammenlegen von Seiten/Knoten (Concatenation) ist stets dann möglich, wenn der Ausgleich nicht stattfinden kann.

UNDERFLOW(v)

2. Fall: v hat keinen Bruder mit $> m$ Schlüssel. Sei u ein benachbarter Bruder von v.

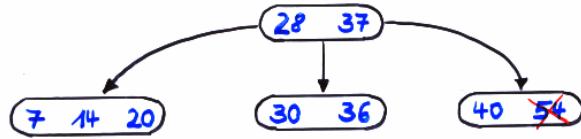


Falls w nicht die Wurzel ist und jetzt $m-1$ Schlüssel hat, dann UNDERFLOW(w).

Falls w die Wurzel ist und jetzt keine Schlüssel mehr hat, dann Wurzel := s.

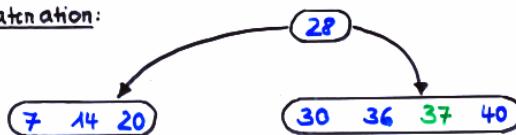
Man beachte, daß sich durch das Zusammenlegen von Seiten/Knoten die Höhe des Baums ändern kann. Nämlich dann, wenn Söhne der Wurzel zusammengelegt werden und die Wurzel nur einen Schlüsselwert hatte.

In dem folgenden Beispiel ist kein Ausgleich möglich.



Löschen von 54: Kein Ausgleich mit benachbartem Bruder möglich

Concatenation:



2704

Im Gegensatz zu binären Suchbäumen wachsen B-Bäume also „nach oben“ durch den zweiten Fall der Overflow-Behandlung. Entsprechend schrumpfen B-Bäume „von oben“ durch den zweiten Fall der Underflow-Behandlung für den Sonderfall, daß die Wurzel der Vater der zusammengelegten Knoten ist und lediglich einen Schlüsselwert enthält. Ferner machen diese Sonderfälle des Einfügens/Löschen klar, warum es für den Wurzelknoten zugelassen ist, weniger als m Schlüsselwerte zu haben, auch dann, wenn insgesamt mehr als m Datensätze darzustellen sind.

Kosten. Die Rebalancierungsmaßnahmen können im schlimmsten Fall an jedem Knoten auf dem (umgekehrten) Pfad von dem modifizierten Blatt bis zur Wurzel nötig sein. An jedem Knoten entstehen jedoch nur konstante Kosten. Der gesamte Aufwand für das Einfügen und Löschen kann daher durch die maximale Anzahl an notwendigen Blocktransporten

$$2 \cdot \text{Höhe}(\mathcal{T}) + 1 \leq 2 \cdot \log_{(m+1)}\left(\frac{N+1}{2}\right) + 1$$

abgeschätzt werden.

Zahlenbeispiel: Ist eine Datei mit 2 Mio Datensätzen gegeben und ist $m = 1.000$, dann hat der B-Baum die Höhe 2. Für das Einfügen und Löschen eines Datensatzes sind – selbst im schlimmsten Fall – lediglich 6 Blocktransporte erforderlich. Für die Suche reichen stets 3 Blocktransporte. Wir fassen die Ergebnisse zusammen:

Satz 4.2.15 (Kosten der Operationen auf B-Bäumen). Suchen, Einfügen und Löschen in einen B-Baum der Ordnung m mit N Schlüsselwerten lassen sich so implementieren, daß die Anzahl an Seitenzugriffen durch

$$\mathcal{O}(\text{Höhe}(\mathcal{T})) = \mathcal{O}(\log_{(m+1)}(N))$$

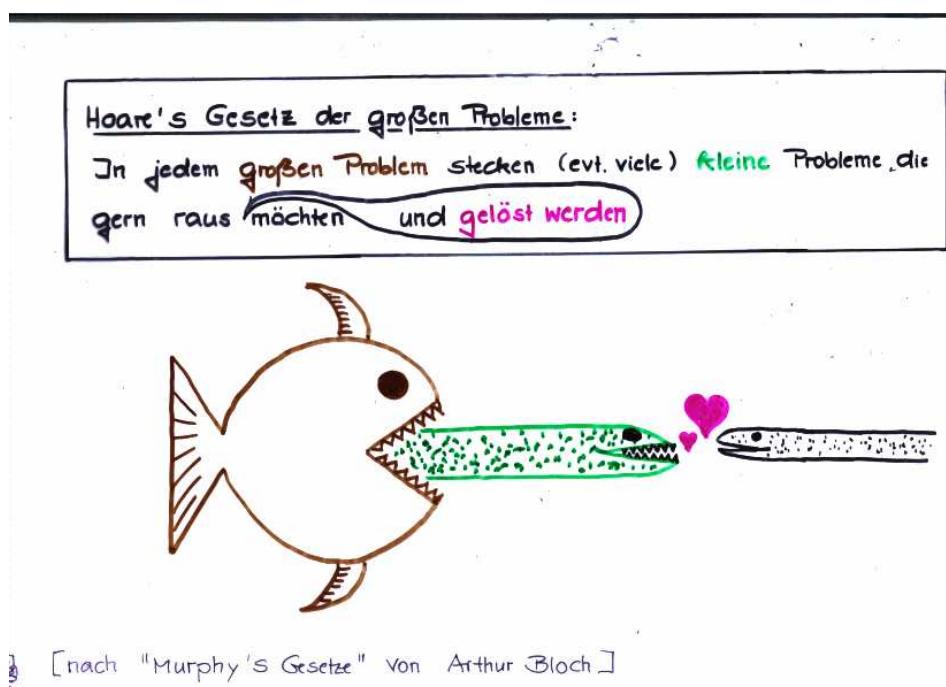
beschränkt ist.

5 Entwurfsmethoden für Algorithmen

Ein Patentrezept für den Entwurf effizienter Algorithmen gibt es leider nicht. Dennoch gibt es prinzipielle Lösungsstrategien, die bei dem Entwurf von Algorithmen oftmals hilfreich sein können. Wir erläutern hier die wichtigsten Konzepte anhand von Beispielen. Teilweise geben wir ein allgemeines Schema an, das zum Verständnis der zugrundeliegenden Idee beitragen sollen. Selbstverständlich können in konkreten Anwendungen Abweichungen von dem skizzierten Schema notwendig (und/oder sinnvoll) sein.

5.1 Divide & Conquer

Einige der Entwurfskonzepte wie Divide & Conquer (oder dynamisches Programmieren) basieren auf einer Zerlegung des gegebenen Problems in „kleine“ Teilprobleme, deren Lösungen zu einer Lösung des Gesamtproblems zusammengesetzt werden können.



Typisch für Divide & Conquer Algorithmen ist die Zerlegung des Problems in (oftmals disjunkte) Teilprobleme, die separat gelöst werden. Ist die Eingabegröße der Teilprobleme hinreichend klein, dann werden die Teilprobleme direkt gelöst. Andernfalls wird dasselbe Zerlegungsschema auf die Teilprobleme angewandt. Die Lösungen der Teilprobleme werden dann zur Gesamtlösung zusammengesetzt. Bereits behandelte Beispiele sind die binäre Suche, Mergesort, Quicksort und der hier diskutierte Linearzeit-Median-Algorithmus. Folgende Tabelle demonstriert die drei Phasen am Beispiel von Merge- und Quicksort:

	Mergesort	Quicksort
DIVIDE	Zerlege die Eingabefolge in zwei gleichgroße Teilfolgen	Zerlege die Eingabefolge gemäß eines Pivot-Elements
CONQUER		Sortiere die Teilfolgen (rekursiv)
COMBINE	Mische die sortierten Teilfolgen	Hänge die sortierten Teilfolgen aneinander

5.1.1 Das Master-Theorem

Häufiger Spezialfall von Divide & Conquer Algorithmen ist die Zerlegung in ungefähr gleichgroße Teilprobleme und lineare Zerlegungs- und Zusammensetzungskosten. Dies führt zu einer Kostenfunktion

$$T(n) = \begin{cases} d & : \text{falls } n \leq n_0 \\ a \cdot T(\lceil n/b \rceil) + cn & : \text{falls } n > n_0 \end{cases}$$

wobei wir annehmen, daß das Problem für Eingabegröße $n \leq n_0$ direkt (ohne Rekursionsaufruf) gelöst wird. Die Konstanten a, b sind ganze Zahlen ($a \geq 1, b \geq 2$) mit folgender Bedeutung:

- a ist die Anzahl an Teilprobleme,
- $\lceil n/b \rceil$ ist die Größe der Teilprobleme.

Für die in der Kostenfunktion auftretenden Konstanten c und d setzen wir $c, d \geq 1$ voraus. Für die asymptotischen Kosten kann man o.E. annehmen, daß $c = d$.

Weiter genügt es, sich auf den Fall $n = b^k$ für eine natürliche Zahl k zu beschränken, wenn man nur an der Größenordnung von T interessiert ist.

Satz 5.1.1 (Master-Theorem). Die Kostenfunktion T sei wie oben.³⁵

- Für $a < b$ (d.h. Gesamtgröße der Teilprobleme $< n$) ist $T(n) = \Theta(n)$.
- Für $a = b$ (d.h. Gesamtgröße der Teilprobleme $= n$) ist $T(n) = \Theta(n \log n)$.
- Für $a > b$ (d.h. Gesamtgröße der Teilprobleme $> n$) ist $T(n) = \Theta(n^{\log_b a})$.

Satz 5.5.14 ist ein Spezialfall der unten angegebenen verallgemeinerten Fassung (siehe Satz 5.5.15).

Z.B. die Kosten für Mergesort ergeben sich aus dem Fall $a = b = 2$. Weiter erhalten wir $\Theta(n)$ als Lösung für $T(n) = \Theta(n) + 3T(\frac{n}{4})$ und $\Theta(n^2)$ als Lösung für $T(n) = \Theta(n) + 9T(\frac{n}{3})$ oder auch $T(n) = \Theta(n) + 4T(\frac{n}{2})$.

In vielen Fällen, in denen Satz 5.5.14 nicht anwendbar ist, z.B. weil das Problem nicht in gleichgroße Teilprobleme unterteilt wird, kann das Master-Theorem hilfreich sein, die Lösung der Rekurrenz für die Kostenfunktion zu erraten und diese dann durch Induktion zu beweisen. Z.B. betrachten wir einen Divide & Conquer Algorithmus wie

³⁵Die Angabe des Verhältnis zwischen der Gesamtgröße der Teilprobleme und der Eingabegröße n bezieht sich auf den Fall, daß n ein ganzahliges Vielfaches von b ist.

den Median-Algorithmus, der ein Problem der Größe $n > n_0$ in zwei Teilprobleme, eines der Eingabegröße $\lfloor n/5 \rfloor$ und eines der Eingabegröße $\lfloor 7/10 \cdot n \rfloor$, unterteilt und dessen Zerlegungs- und Zusammensetzungskosten linear sind. D.h. die Kostenfunktion hat die Gestalt

$$T(n) = T(\lfloor n/5 \rfloor) + T(\lfloor 7/10 \cdot n \rfloor) + c \cdot n$$

für $n > n_0$ und $T(n) = d$ für $n \leq n_0$ (wobei c und d Konstanten > 0 sind). Das Master-Theorem ist nicht anwendbar. Dennoch legt es die Vermutung nahe, daß $T(n) = \Theta(n)$, da die Gesamtgröße der Teilprobleme $\leq n/5 + 7/10n = 9/10n < n$ ist. Eine entsprechende Vorgehensweise führt zum Auffinden der Lösung $F(n) = \Theta(n \log n)$ für die Rekurrenz

$$F(n) = F(\lfloor n/5 \rfloor) + F(\lfloor 4/5n \rfloor) + c \cdot n.$$

Tatsächlich gilt sogar folgende Verallgemeinerung des Master-Theorems, die neben dem Sonderfall gleichgrosser Teilprobleme und lineare Zerlegungs- und Zusammensetzungskosten auch die eben genannten Fälle umfasst:

Satz 5.1.2 (Master-Theorem (allgemeine Fassung)). Gegeben ist eine Rekursionsgleichung der Form

$$T(n) = \Theta(n^k) + \sum_{i=1}^m T(d_i n)$$

wobei $m \in \mathbb{N}_{\geq 1}$, $k \geq 0$ eine reelle Zahl und d_1, \dots, d_m reelle Zahlen im offenen Intervall $]0, 1[$. Dann gilt:

$$T(n) = \begin{cases} \Theta(n^k) & : \text{falls } \sum_{i=1}^m d_i^k < 1 \\ \Theta(n^k \log n) & : \text{falls } \sum_{i=1}^m d_i^k = 1 \\ \Theta(n^r) & : \text{falls } \sum_{i=1}^m d_i^k > 1 \text{ und } \sum_{i=1}^m d_i^r = 1 \end{cases}$$

Satz 5.5.14 ergibt sich als ein Spezialfall von Satz 5.5.15, indem wir $k = 1$, $m = a$ und $d_1 = \dots = d_a = \frac{1}{b}$ betrachten. Dann ist $d_1^k + \dots + d_m^k = a \cdot \frac{1}{b}$. Im dritten Fall ergibt sich aus Satz 5.5.15, daß $T(n) = \Theta(n^r)$, wobei r so gewählt ist, dass $1 = d_1^r + \dots + d_m^r = a \cdot (\frac{1}{b})^r = \frac{a}{b^r}$, also $a = b^r$ und somit $r = \log_b a$.

Beweis. Bevor wir den Beweis des Master-Theorems bringen, erläutern wir kurz, was mit der angegebenen Rekurrenz genau gemeint ist. Die angegebene Rekurrenz ist so zu lesen, dass es eine natürliche Zahl n_0 und reelle Konstanten a, A mit $0 < a < A$ gibt, so dass

$$an^k + \sum_{i=1}^m T(\lceil d_i n \rceil) \leq T(n) \leq An^k + \sum_{i=1}^m T(\lceil d_i n \rceil)$$

für alle $n \geq n_0$, wobei $\lceil d_i n_0 \rceil < n_0$ vorausgesetzt wird. (Es gilt also $\lceil d_i n \rceil \leq n - 1$ für alle $n \geq n_0$.) Zur Vereinfachung schreiben wir $T(d_i n)$ statt $T(\lceil d_i n \rceil)$ bzw. $T(\lfloor d_i n \rfloor)$. Da wir uns lediglich für das asymptotische Verhalten von T interessieren, ist es keine

Einschränkung anzunehmen, dass T monoton ist und dass, $T(0) = 0$ und $T(n) \geq 1$ für alle $n \geq 1$ gilt.³⁶ Ist nun $f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ eine monotone Funktion mit $f(0) = 0$ und $f(n) \geq 1$ für $n \geq 1$, so können die Konstanten a, A klein bzw. groß genug gewählt werden, um $af(m) \leq T(m) \leq Af(m)$ für alle $m < n_0$ zu erzwingen. Hiervon machen wir Gebrauch, indem wir $f(n) = n^k$ im ersten Fall setzen, $f(n) = n^k \log n$ im zweiten Fall und $f(n) = n^r$ im dritten Fall.

Wie bereits erwähnt gehen wir lässig mit den Zahlen $d_i n$ um und behandeln diese wie ganze Zahlen, wobei $d_i n < n$ vorausgesetzt wird.

1. Fall. Wir beginnen mit dem Fall $\sum_{i=1}^m d_i < 1$.

Die Tatsache, daß $T(n) = \Omega(n^k)$ folgt sofort aus der Beziehung $T(n) = \Theta(n^k) + \sum_i T(\dots)$. Wir weisen nun die obere Schranke $\mathcal{O}(n^k)$ nach. Zu zeigen ist, daß es eine Konstante $C > 0$ gibt, so daß $T(n) \leq Cn^k$ für hinreichend großes n . Dies weisen wir mit dem Prinzip der induktiven Ersetzung nach und nehmen dabei an, daß $T(m) \leq Cm^k$ für $m < n$.

$$\begin{aligned} T(n) &\leq \sum_{i=1}^m T(d_i n) + An^k \\ &\leq \sum_{i=1}^m Cd_i^k n^k + An^k \\ &= n^k \cdot \left(A + C \underbrace{\sum_{i=1}^m d_i^k}_{=:D} \right) \\ &= n^k \cdot (A + C \cdot D) \end{aligned}$$

Man beachte, daß $0 < D < 1$. Es gilt:

$$A + CD \leq C \text{ gdw } A \leq C(1 - D) \text{ gdw } C \geq \frac{A}{1-D}$$

Also können wir $C \geq \max \{A, \frac{A}{1-D}\} = \frac{A}{1-D}$ wählen und erhalten $T(n) \leq Cn^k$ für alle $n \geq 0$ und somit $T(n) = \mathcal{O}(n^k)$.

2. Fall. Wir diskutieren nun den Fall $\sum_{i=1}^m d_i^k = 1$.

Für die obere Schranke ist der Nachweis zu führen, daß $T(n) \leq Cn^k \log n$ für eine geeignete wählende positive Konstante C . Auch hier arbeiten wir zunächst heuristisch und

³⁶Dies kann damit erklärt werden, dass (1) T durch $T'(n) = \max_{0 \leq i \leq n} T(i)$ ersetzt werden kann, womit die Monotonie erzwungen wird, und (2) T' durch T'' ersetzt werden kann, wobei $T''(0) = 0$ und $T''(n) = \max\{1, T'(n)\}$ für $n \geq 1$. Es gilt dann $T(n) = \Theta(T''(n))$.

nehmen $T(m) \leq Cm^k \log m$ für alle $m < n$ an. Für $m \geq n_0$ gilt dann:

$$\begin{aligned}
T(n) &\leq \sum_{i=1}^m T(d_i n) + An^k \\
&\leq \sum_{i=1}^m Cd_i^k n^k \log(d_i n) + An^k \\
&= n^k \cdot \left(A + C \sum_{i=1}^m d_i^k (\log d_i + \log n) \right) \\
&= n^k \cdot \left(A + C \sum_{i=1}^m d_i^k \log d_i + \underbrace{C \sum_{i=1}^m d_i^k \log n}_{=\log n} \right) \\
&= n^k \cdot \left(A + \underbrace{C \sum_{i=1}^m d_i^k \log d_i}_{=:E} + C \log n \right) \\
&= n^k \cdot (A + CE + C \log n)
\end{aligned}$$

Beachte, daß $E < 0$, da die d_i 's alle zwischen 0 und 1 liegen und daher $\log d_i < 0$. Wir wählen $C \geq \max\{A, -\frac{A}{E}\}$ und erhalten

$$n^k(A + CE + C \log n) \leq n^k(A - \frac{A}{E} \cdot E + C \log n) = n^k(A - A + C \log n) = Cn^k \log n$$

und somit $T(n) = \mathcal{O}(n^k \log n)$.

Die Rechnung für die untere Schranke ist analog, wobei wir von der Rekurrenz

$$T(n) \geq \sum_{i=1}^m T(d_i n) + an^k$$

ausgehen und $T(m) \geq cm^k \log m$ für $m < n$ annehmen:

$$\begin{aligned}
T(n) &\geq \sum_{i=1}^m T(d_i n) + an^k \\
&\geq \sum_{i=1}^m cd_i^k n^k \log(d_i n) + an^k \\
&= n^k \cdot \left(a + c \sum_{i=1}^m d_i^k (\log d_i + \log n) \right) \\
&= n^k \cdot \left(a + \underbrace{c \sum_{i=1}^m d_i^k \log d_i}_{=E} + \underbrace{c \sum_{i=1}^m d_i^k \log n}_{=\log n} \right) \\
&= n^k \cdot (a + cE + c \log n)
\end{aligned}$$

Hier wählen wir c so, daß $0 < c \leq \min\{a, -\frac{a}{E}\}$ und erhalten $T(n) \geq cn^k \log n$.

3. Fall. Es bleibt der Fall $\sum_{i=1}^m d_i^k > 1$ und $\sum_{i=1}^m d_i^r = 1$.

Offenbar ist dann $k < r$, da die d_i 's zwischen 0 und 1 liegen. Zunächst betrachten wir die untere Schranke, die wir mit $T(n) \geq cn^r$ ansetzen:

$$\begin{aligned}
T(n) &\geq \sum_{i=1}^m T(d_i n) + an^k \\
&\geq \sum_{i=1}^m c(d_i n)^r + an^k \\
&\geq \sum_{i=1}^m cd_i^r n^r + an^k \\
&\geq cn^r \underbrace{\sum_{i=1}^m d_i^r}_{=1} = cn^r
\end{aligned}$$

Hier muß die Konstante c also nur so gewählt werden, daß sie auf den Induktionsanfang passt, d.h. wir können $c = a$ setzen.

Nun zur oberen Schranke. Wir wählen ein beliebiges $s \in]k, r[$ und zeigen, dass es eine Konstante $C > 0$ und ein $n_1 \in \mathbb{N}$ gibt, so dass $T(n) \leq Cn^r - n^s$.

Im Folgenden sei $F_s = \sum_{1 \leq i \leq m} d_i^s - 1$. Dann ist $F_s > 0$.

Wir arbeiten wieder heuristisch und nehmen $T(m) \leq Cm^r - m^s$ für $m < n$ an. Dann gilt

für $n \geq n_0$:

$$\begin{aligned}
T(n) &\leq \sum_{i=1}^m T(d_i n) + An^k \\
&\leq \sum_{i=1}^m Cd_i^r n^r - \sum_{i=1}^m d_i^s n^s + An^k \\
&= Cn^r - (1 + F_s)n^s + An^k \\
&= (Cn^r - n^s) + (-F_s n^s + An^k)
\end{aligned}$$

Wegen $F_s > 0$ und $s > k$ gilt $-F_s n^s + An^k \leq 0$ für hinreichend grosses n , etwa $n \geq n_1$. Also kann auch hier die Konstante C gemäß des Induktionsanfangs, also passend für die Fälle $n < \max\{n_0, n_1\}$, gewählt werden. \square

Mit $k = 0$, $m = 1$ und $d_1 = \frac{1}{2}$ erhält man die Rekurrenz für die binäre Suche $T(n) = \Theta(1) + T(\frac{n}{2})$ und damit die Lösung $\Theta(\log n)$ gemäß Fall 2. Mit $k = 1$, $m = 2$ und $d_1 = d_2 = \frac{1}{2}$ erhalten wir die von Mergesort bekannte Rekurrenz $T(n) = \Theta(n) + 2T(\frac{n}{2})$ und deren Lösung $\Theta(n \log n)$, ebenfalls gemäß Fall 2.

Eine Instanz des ersten Falls mit $k = 1$, $m = 2$ und $d_1 = \frac{1}{5}$, $d_2 = \frac{7}{10}$ ist die Rekurrenz

$$T(n) = \Theta(n) + T(\frac{n}{5}) + T(\frac{7}{10}n),$$

die wir für den Linearzeit-Algorithmus zur Mediansuche erhalten hatten. Ein Beispiel für den dritten Fall wird in folgendem Unterabschnitt angegeben.

5.1.2 Matrizenmultiplikation nach Strassen

Gegeben sind zwei $(n \times n)$ -Matrizen \mathbf{A} und \mathbf{B} , wobei $n = 2^k$ eine Zweierpotenz ist (und k eine ganze Zahl ≥ 1). Gesucht ist das Matrizenprodukt $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$. Die Schulmethode (siehe Algorithmus 70) benötigt offenbar $\Theta(n^3)$ Schritte.

Algorithmus 44 Matrizenmultiplikation für zwei $(n \times n)$ -Matrizen (Schulmethode)

(* Eingabe sind zwei Matrizen $\mathbf{A} = (a_{i,j})_{1 \leq i,j \leq n}$ und $\mathbf{B} = (b_{j,k})_{1 \leq j,k \leq n}$ *)

FOR $i = 1, \dots, n$ **DO**

FOR $k = 1, \dots, n$ **DO**

$c_{i,k} := 0$;

FOR $j = 1, \dots, n$ **DO**

$c_{i,k} := c_{i,k} + a_{i,j} * b_{j,k}$

OD

OD

OD

Gib die Produktmatrix $\mathbf{C} = (c_{i,k})_{1 \leq i,k \leq n}$ aus.

Naives Divide & Conquer. Wir stellen nun einen Divide & Conquer Algorithmus, welcher die drei Matrizen \mathbf{A} , \mathbf{B} und \mathbf{C} in jeweils $4 \left(\frac{n}{2} \times \frac{n}{2}\right)$ -Matrizen zerlegt:

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{pmatrix} \quad \mathbf{C} = \begin{pmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{pmatrix}$$

Die Produktmatrix \mathbf{C} ergibt sich nun durch:

$$\begin{aligned} \mathbf{C}_{1,1} &= \mathbf{A}_{1,1} \cdot \mathbf{B}_{1,1} + \mathbf{A}_{1,2} \cdot \mathbf{B}_{2,1} \\ \mathbf{C}_{1,2} &= \mathbf{A}_{1,1} \cdot \mathbf{B}_{1,2} + \mathbf{A}_{1,2} \cdot \mathbf{B}_{2,2} \\ \mathbf{C}_{2,1} &= \mathbf{A}_{2,1} \cdot \mathbf{B}_{1,1} + \mathbf{A}_{2,2} \cdot \mathbf{B}_{2,1} \\ \mathbf{C}_{2,2} &= \mathbf{A}_{2,1} \cdot \mathbf{B}_{1,2} + \mathbf{A}_{2,2} \cdot \mathbf{B}_{2,2} \end{aligned}$$

Für die Berechnung von \mathbf{C} durch die angegebenen Formeln erhält man die Kostenfunktion $T(1) = 1$ und

$$T(n) = 8 \cdot T\left(\frac{n}{2}\right) + \Theta(n^2).$$

Dies ergibt sich aus der Beobachtung, daß 8 Multiplikationen und 4 Additionen von $\left(\frac{n}{2} \times \frac{n}{2}\right)$ -Matrizen nötig sind.

Mit $k = 2$, $m = 8$, $d_1, \dots, d_m = \frac{1}{2}$ erhalten wir aus Satz 5.5.15:

$$T(n) = 8 \cdot T(n/2) + \Theta(n^2) = \Theta(n^3).$$

Beachte: $8(\frac{1}{2})^2 = \frac{8}{4} = 2 > 1$. Daher ist der dritte Teil von Satz 5.5.15 relevant. Weiter gilt

$$8(\frac{1}{2})^3 = \frac{8}{8} = 1,$$

also ist $r = 3$. Mit dem naiven Divide & Conquer ist also gegenüber der „Schulmethode“ keine asymptotische Zeitersparnis zu verzeichnen.

Matrizenmultiplikation nach Strassen. Wir betrachten nun die von Strassen vorgeschlagene Methode. Diese benutzt ebenfalls die Zerlegung der beiden Matrizen \mathbf{A} und \mathbf{B} in die Untermatrizen $\mathbf{A}_{i,j}$ und $\mathbf{B}_{i,j}$. Die Berechnung der Untermatrizen $\mathbf{C}_{i,j}$ der Produktmatrix \mathbf{C} beruht auf dem folgenden Schema.³⁷

$$\begin{aligned} \mathbf{M}_1 &:= (\mathbf{A}_{1,1} - \mathbf{A}_{2,2}) \cdot (\mathbf{B}_{2,1} + \mathbf{B}_{2,2}) \\ \mathbf{M}_2 &:= (\mathbf{A}_{1,2} + \mathbf{A}_{2,2}) \cdot (\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\ \mathbf{M}_3 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1}) \cdot (\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\ \mathbf{M}_4 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2}) \cdot \mathbf{B}_{2,2} \\ \mathbf{M}_5 &:= \mathbf{A}_{1,1} \cdot (\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\ \mathbf{M}_6 &:= \mathbf{A}_{2,2} \cdot (\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\ \mathbf{M}_7 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{1,2}) \cdot \mathbf{B}_{1,1} \end{aligned}$$

³⁷Die Autorin übernimmt keine Garantie, daß jeder Index in den angegebenen Formeln stimmt.

und

$$\begin{aligned}\mathbf{C}_{1,1} &:= \mathbf{M}_1 + \mathbf{M}_2 - \mathbf{M}_4 + \mathbf{M}_6 \\ \mathbf{C}_{1,2} &:= \mathbf{M}_4 + \mathbf{M}_5 \\ \mathbf{C}_{2,1} &:= \mathbf{M}_6 + \mathbf{M}_7 \\ \mathbf{C}_{2,2} &:= \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_5 - \mathbf{M}_7\end{aligned}$$

Wir verzichten auf den Korrektheitsnachweis und konzentrieren uns stattdessen auf die Kostenanalyse. Die Methode von Strassen benötigt 7 Multiplikationen und 18 Additionen/Subtraktionen von $(\frac{n}{2} \times \frac{n}{2})$ -Matrizen. Dies führt zur Rekurrenz

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + \Theta(n^2).$$

Wiederum wenden wir den dritten Teil von Satz 5.5.15 an, da $7\left(\frac{1}{2}\right)^2 = \frac{7}{4} > 1$. Mit $r = \log 7$ erhalten wir $7\left(\frac{1}{2}\right)^{\log 7} = \frac{7}{7} = 1$ und somit:

$$T(n) = \Theta\left(n^{\log 7}\right) \approx \Theta\left(n^{2.8}\right)$$

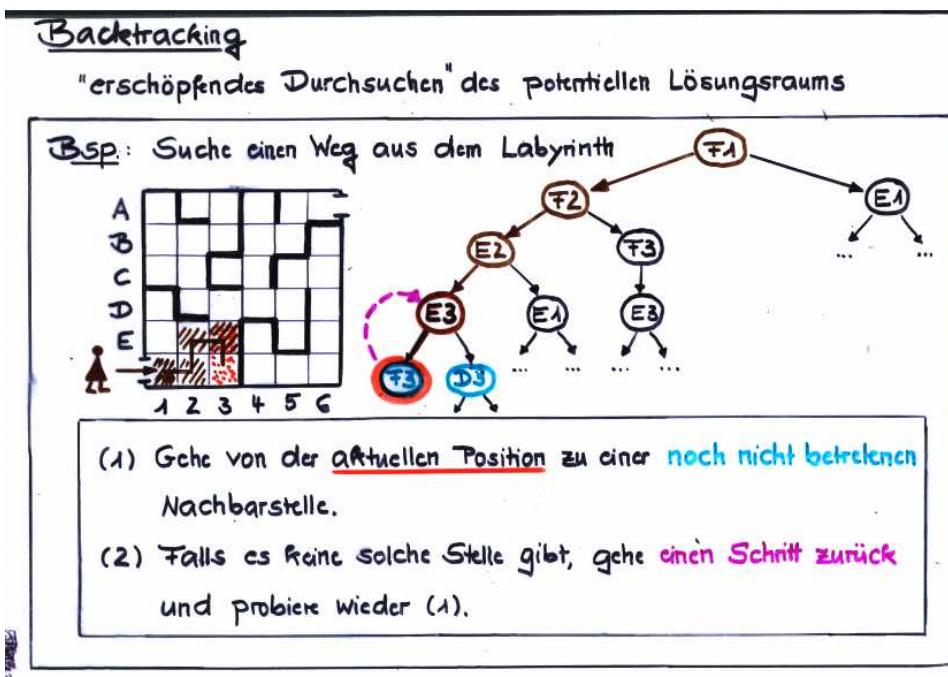
Wir fassen die Ergebnisse zusammen:

Satz 5.1.3 (Komplexität der Strassen-Methode). Mit der Methode von Strassen können zwei $n \times n$ -Matrizen in Zeit $\Theta\left(n^{\log 7}\right)$ multipliziert werden.

Experimentelle Ergebnisse haben gezeigt, daß die Strassen-Methode erst für großes n (ca. $n \geq 500$) besser als die Schulmethode ist. In der Praxis empfiehlt sich daher eine gemischte Strategie, die für $n \geq 500$ eine Zerlegung à la Strassen vornimmt. Sobald die Zeilenanzahl auf einen Wert < 500 reduziert ist, wird die Schulmethode angewandt, um die Teilprodukte zu ermitteln.

5.2 Backtracking und Branch & Bound

Backtracking und Branch & Bound zählen zu den sogenannte Aufzählungsmethoden. Beide Strategien gehen davon aus, daß der potentielle Lösungsraum des vorliegenden Problems baumartige Gestalt hat. Der Grundgedanke von Backtracking ist die schrittweise Konstruktion einer Lösung, wobei Schritte rückgängig gemacht werden, sobald sie sich als falsch erwiesen haben. Beschränkungsfunktionen versuchen anhand unterer und oberer Schranken „Irrwege“ frühzeitig zu erkennen und so den potentiellen Lösungsraum zu beschneiden. Branch & Bound Strategien arbeiten zusätzlich mit einer Verzweigungsheuristik, die den „besten“ nächsten Schritt ausfindig zu machen versucht. Wir erläutern die Grundidee am Beispiel eines Labyrinths (siehe Folie). Dieses kann als ungerichteter Graph aufgefaßt werden, in dem ein Pfad von dem Anfangsknoten (Eingang des Labyrinths) zu dem Zielknoten (Ausgang des Labyrinths) gesucht wird.



Das Grundprinzip von Backtracking basiert auf einer DFS-ähnlichen Graphtraversierung des zugrundeliegenden Aufzählungsbaums, welcher den Lösungsraum repräsentiert. Von der aktuellen Position geht man zu einer noch nicht besuchten Nachbarstelle. Wenn es keine solche gibt, geht man einen Schritt zurück und sucht dort nach einer noch nicht besuchten Nachbarstelle.

Backtracking beruht auf einer Darstellung des potentiellen Lösungsraums durch einen Baum, den so genannten *Aufzählungsbaum*, und wird daher auch oftmals Aufzählungsmethode genannt. Die Knoten des Aufzählungsbaums können wir uns als Tupel $\bar{x} = \langle x_1, \dots, x_n \rangle$ vorstellen, wobei die Werte x_i für „vorläufig“ getroffene Entscheidungen stehen. Die Söhne eines Knotens $\bar{x} = \langle x_1, \dots, x_n \rangle$ sind genau die Knoten der Form $\langle \bar{x}, y \rangle = \langle x_1, \dots, x_n, y \rangle$. Die Blätter des Aufzählungsbaums repräsentieren die eigentlichen Lösungskandidaten.

Für das oben skizzierte Labyrinth entspricht der Aufzählungsbaum dem rechts skizzierten Baum, welcher die Pfade von der Startposition F1 repräsentiert.

Beispiel 5.2.1 (Aufzählungsbaum für das 8er Puzzle). Als weiteres Beispiel für die Darstellung des Lösungsraums eines Problems durch einen Aufzählungsbaum betrachten wir das 8er Puzzle:



Wir starten mit einer Anfangskonfiguration, in dem 8 Plättchen, die von 1 bis 8 nummeriert sind auf einem 3×3 -Spielfeld angeordnet sind. Zugmöglichkeiten bestehen immer nur darin, eines der Plättchen, welches dem leeren Feld benachbart ist, auf das leere Feld zu verschieben. Gesucht ist eine Zugfolge, welche die Anfangskonfiguration in die Zielkonfiguration, in der die Plättchen – wie oben skizziert – sortiert sind, überführt. Zunächst handelt es sich hierbei um ein Erreichbarkeitsproblem in einem Graphen, dessen Knoten die Konfigurationen und Kanten die jeweiligen Zugmöglichkeiten sind. Für das 8er Puzzle hat man es mit einem Graphen mit rund $9! = 362.880$ Knoten zu tun. Wesentlich dramatischer ist die Situation beim 15-Puzzle, bei dem ein Graph mit ca. $16! \approx 20.9 * 10^{12}$ Knoten zu untersuchen ist. Geht man davon aus, daß 10^8 Knoten pro Sekunde erzeugt werden können, muß man sich immer noch ca. 40 Tage gedulden bis sämtliche Knoten generiert wurden. Offenbar benötigt man hierzu (sowie für das $(n^2 - 1)$ er Puzzle mit $n \geq 6$) einige zusätzlichen Tricks, welche die Suche nach einem Pfad zur Zielkonfiguration beschleunigen. Mehr dazu später.

Der Aufzählungsbaum für das $(n^2 - 1)$ er Puzzle ergibt sich durch „Abwickeln“ des Konfigurationsgraphen in einen Baum. Die Wurzel steht für die Anfangskonfiguration, jeder andere Knoten für eine Zugfolge, die intuitiv mit der letzten Konfiguration identifiziert werden kann. Die Blätter stehen für die (Pfade zur) Zielkonfiguration und stellen somit die Lösungen dar. Man beachte, dass der Aufzählungsbaum unendlich ist, sofern nicht die Einschränkung gemacht wird, daß jede Konfiguration höchstens einmal „besucht“ werden darf. \square

Das Konzept von Aufzählungsbäumen dient lediglich der Anschauung. Für ein gegebenes Problem kann es sehr viele völlig verschiedene Aufzählungsbäume geben. Diese dienen nur als gedankliche Hilfestellung für den Entwurf (und die Analyse) eines Backtracking oder Branch & Bound Algorithmus. Letztendlich wird der Aufzählungsbaum nicht explizit konstruiert. Die prinzipielle Vorgehensweise von Backtracking besteht darin, die Knoten *dynamisch* im Preorder-Prinzip zu generieren und nur solche Knoten zu expandieren (d.h. deren Teilbäume zu erzeugen), für die gewisse Randbedingungen erfüllt sind. Die komplette Analyse des Aufzählungsbaums würde dagegen einem naiven Algorithmus, der „alle Möglichkeiten ausprobiert“, entsprechen.

Auch das „Generieren“ der Knoten des Aufzählungsbaums ist bildlich zu verstehen. Tatsächlich genügt es oftmals, nur die letzte Komponente x_n eines Knotens $\bar{x} = \langle x_1, \dots, x_n \rangle$

darzustellen. Die vorangegangenen Komponenten x_i , $i = n - 1, n - 2, \dots, 1$, ergeben sich aus dem „inversen Weg“ von dem Knoten \bar{x} zur Wurzel. Branch & Bound zielt darauf ab, große Teile des Aufzählungsbaums einzusparen (d.h. gar nicht erst zu generieren). Dies wird durch den Einsatz von Heuristiken erreicht, welche die Reihenfolge festlegen, in welcher die Knoten expandiert werden.

Bemerkung 5.2.2 (Aufzählungsbäume und nichtdeterministische Algorithmen). Aufzählungsbäume können oftmals als Pendant zu den Berechnungsbäumen nichtdeterministischer Algorithmen angesehen werden. Z.B. kann der Aufzählungsbaum des 8er-Puzzles mit dem Berechnungsbaum des nichtdeterministischen Verfahrens

„Rate eine Zugfolge und überprüfe, ob diese zur Zielkonfiguration führt“

identifiziert werden. Diese Beobachtung erklärt, warum Backtracking oder Branch & Bound Algorithmen für viele NP-harte Probleme geeignet sind, insbesondere dann, wenn vergleichsweise einfache, nichtdeterministische „Guess & Check“ Algorithmen für die betreffende Problemstellung angegeben werden können. Andererseits sind Backtracking Algorithmen meist nicht effizient, da sie – wenn auch mit einigen Zusatztricks – auf einer „alle-Lösungskandidaten-ausprobieren“-Idee beruhen, und sollten daher nur dann eingesetzt werden, wenn keine effizienten Verfahren zu erwarten sind. \square

5.2.1 Backtracking

Wir erläutern die Vorgehensweise von Backtracking an drei Beispielen, in denen man es mit einem endlichen Aufzählungsbaum zu tun hat.

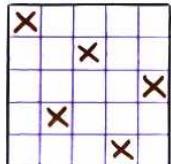
Das n -Damen-Problem

Gesucht ist eine Plazierung von n Damen auf dem $n \times n$ Schachbrett, so daß keine der Damen eine andere bedroht.

Das n-Damen Problem:

Plaziere n Damen auf dem $n \times n$ -Schachbrett, so daß sich die Damen nicht gegenseitig bedrohen.

$n = 5$



Darstellung des Lösungsraums:

Tupel (x_1, x_2, \dots, x_n) mit $x_i \in \{1, \dots, n\}$, wobei

$x_i = j$ gdw Dame aus Zeile i steht in Spalte j

Beachte: der Lösungsraum hat n^n Elemente!

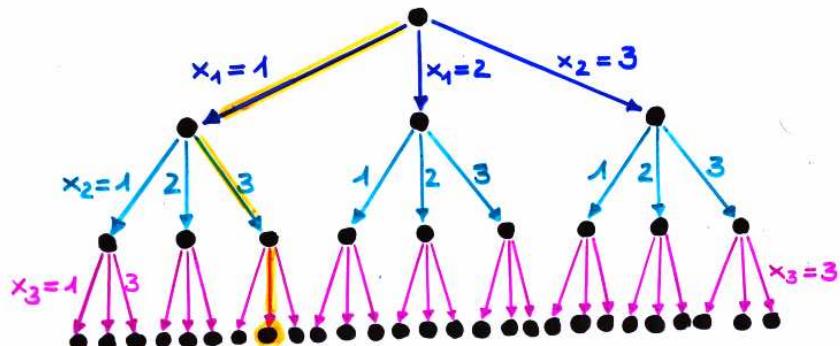
Mit der trivialen „Alle möglichen Konfigurationen ausprobieren“-Idee versagt man hier kläglich, da es

$$\binom{n^2}{n} \geq n^n \text{ Konfigurationen}$$

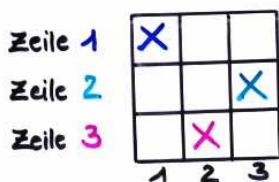
gibt. Da in jeder Zeile genau eine Dame stehen muß, können wir die Lösungskonfigurationen als n -Tupel $\langle x_1, \dots, x_n \rangle$ beschreiben, so daß $x_i = j$ genau dann, wenn die Dame der i -ten Zeile in Spalte j steht.

Der Aufzählungsbaum für das n -Damen-Problem hat die Höhe n . Die von Knoten der Tiefe $i - 1$ ausgehenden Kanten stehen für die Positionen der Dame in der i -ten Zeile (also die möglichen Werte von x_i). Jeder Knoten v der Tiefe i kann daher mit der Konfiguration $\langle x_1, \dots, x_i \rangle$, welche die Plazierungen der ersten i Damen angibt und welche sich aus dem Pfad von der Wurzel zu v ergibt, identifiziert werden. Die Blätter repräsentieren die n^n Positionierungen.

Aufzählungsbaum für das 3-Damen - Problem (Variante 1)



Die Blätter repräsentieren die $3^3 = 27$ möglichen Konfigurationen!

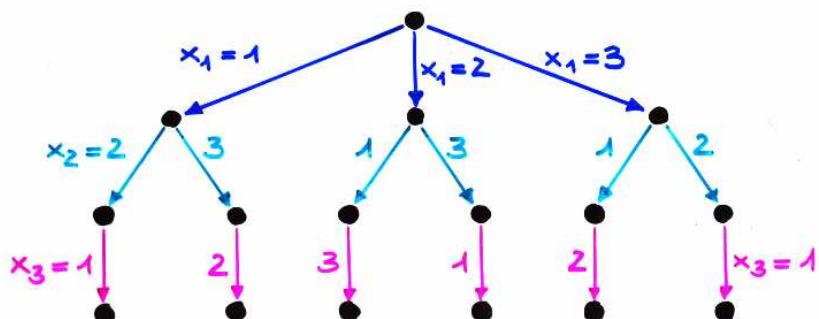


Damen 2 und 3 bedrohen sich

2029

Dieser Baum hat $1+n+n^2+\dots+n^n$ Knoten. Beschneidungen des Baums sind aufgrund der Randbedingung, daß sich keine Damen gegenseitig bedrohen dürfen, möglich. Zunächst kann man die Tatsache ausnutzen, daß die Damen in verschiedenen Spalten stehen müssen, also daß die Werte x_1, \dots, x_n paarweise verschieden sein müssen. Dadurch erhält man einen Baum mit $n!$ Blättern.

Aufzählungsbaum für das 3-Damer - Problem (Variante 2)



Die Blätter repräsentieren die $3! = 6$ mögl. Konfigurationen, in denen in jeder Zeile und Spalte genau eine Dame steht.

2029

Mit dieser Variante des Aufzählungsbaums erhält man eine recht einfache Funktion, die für jeden inneren Knoten $\bar{x} = \langle x_1, \dots, x_{i-1} \rangle$ der Tiefe i die Söhne $\langle x_1, \dots, x_{i-1}, y \rangle$ generiert. Die Söhne von \bar{x} sind alle Knoten der Form $\langle x_1, \dots, x_{i-1}, y \rangle$ mit

$$y \in \{1, \dots, n\} \setminus \{x_1, \dots, x_{i-1}\}.$$

Für jeden dieser Werte $x_i = y$ prüft man nun, ob durch die Plazierung der i -ten Dame in Spalte y eine der bereits plazierten Damen bedroht wird, also ob eine der bereits plazierten Damen auf derselben Diagonalen steht. Die Bedrohung der j -ten Dame durch die i -te Dame in Spalte y ist gleichwertig zur Bedingung, dass $|x_j - y| = |j - i|$.

- Wenn nein, dann wird eine Plazierung für die $(i + 1)$ -te Dame gesucht ($i < n$ vorausgesetzt).
- Wenn ja, dann wird der Lösungskandidat $x_i = y$ verworfen und der nächste Wert für y betrachtet (vorausgesetzt es gibt noch nicht betrachtete Werte für y).
- Wurden alle möglichen Werte $y \in \{1, \dots, n\} \setminus \{x_1, \dots, x_{i-1}\}$ erfolglos ausprobiert, dann hat sich die Plazierung der $(i - 1)$ -ten Dame in Spalte x_{i-1} als unmöglich herausgestellt, sofern die ersten $(i - 2)$ Damen in den Spalten x_1, \dots, x_{i-2} stehen. Es findet Backtracking statt.

Diese Schritte sind in Algorithmus 45 auf Seite 205 zusammengefasst.

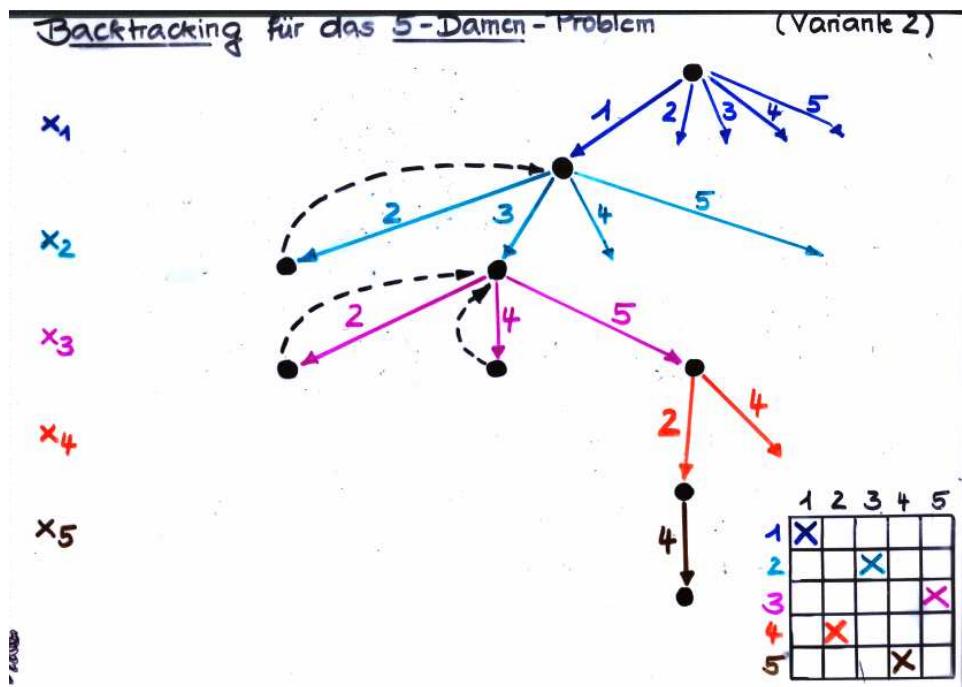
Algorithmus 45 <i>nDamen($i, \langle x_1, \dots, x_{i-1} \rangle$)</i>	(Backtracking)
--	----------------

```

FOR  $y = 1, \dots, n$  DO
  IF  $y \notin \{x_1, \dots, x_{i-1}\}$  THEN
    IF es gibt kein  $j \in \{1, \dots, i - 1\}$ , so dass  $|i - j| = |y - x_j|$  THEN
      IF  $i = n$  THEN
        gib  $\langle x_1, \dots, x_{n-1}, y \rangle$  als Lösung aus
      ELSE
        nDamen( $i + 1, \langle x_1, \dots, x_{i-1}, y \rangle$ )
      FI
    FI
  OD

```

Als Beispiel betrachten wir den Fall $n = 5$. Die gestrichelten Pfeile auf der Folie deuten das Backtracking an.



Mit dieser Vorgehensweise werden für das 5-Damen-Problem lediglich 9 Knoten generiert bis die erste Lösung gefunden wird. Der gesamte Aufzählungsbaum hat dagegen $5! = 120$ Blätter und 326 Knoten.

Das allgemeine Schema für Backtracking

Das allgemeine Schema für Backtracking benutzt einen rekursiven Algorithmus, welcher die Knoten des Aufzählungsbaums im Preorder-Prinzip erzeugt. In dem in Algorithmus 46 angegebenen Schema gehen wir zur Vereinfachung von einem Aufzählungsbaum aus, dessen „Lösungsblätter“ allesamt dieselbe Tiefe n haben. Das Schema ist entsprechend umzuformulieren, wenn ein Aufzählungsbaum vorliegt, in dem die Lösungen durch Blätter unterschiedlicher Tiefe repräsentiert sind. Für unendliche Aufzählungsbäume sind zusätzliche Tricks nötig.

Die Knoten des Aufzählungsbaums kann man sich als Tupel $\langle x_1, \dots, x_{i-1} \rangle$ vorstellen, die für die bisher probeweise durchgeführten „Schritte“ stehen. Zusätzlich können natürlich noch andere Werte übergeben werden. Die Hilfsfunktion $PossibleValues(x_1, \dots, x_{i-1})$ generiert alle potentiellen Werte y für x_i . Dies sind genau diejenigen Werte y , so dass $\langle x_1, \dots, x_{i-1}, y \rangle$ ein Knoten im Aufzählungsbaum ist. Oftmals wird die Funktion $PossibleValues(\dots)$ so gewählt, dass sie lediglich den zulässigen Wertebereich für x_i berücksichtigt. Die Randbedingungen, die eine Forderung an die Beziehung der Werte x_1, \dots, x_i untereinander stellen, werden in einem zweiten Schritt geprüft. Soweit möglich ist es jedoch sinnvoll,

Algorithmus 46 $\text{Backtracking}(i, \langle x_1, \dots, x_{i-1} \rangle)$ (allgemeines Schema)

```

FOR ALL  $y \in \text{PossibleValues}(x_1, \dots, x_{i-1})$  DO
    IF  $\langle x_1, \dots, x_{i-1}, y \rangle$  erfüllt die Randbedingungen THEN
        IF  $i = n$  THEN
            ...
        ELSE
             $\text{Backtracking}(i + 1, \langle x_1, \dots, x_{i-1}, y \rangle)$ 
        FI
    FI
OD
```

auch die Randbedingungen in die Funktion $\text{PossibleValues}(\dots)$ zu integrieren, um den Aufzählungsbaum zu verkleinern.

Die angegebene Parametrisierung dient der Anschauung. In vielen Fällen ist es unnötig, die kompletten Werte x_1, \dots, x_{i-1} zu übergeben, statt dessen können andere Parameter sinnvoll sein. Z.B. reicht es beim n -Damen Problem den Wert i zu übergeben und die Variablen x_1, \dots, x_n als global zu deklarieren.

Wird Backtracking zum Lösen von Optimierungsaufgaben eingesetzt, dann können die Randbedingungen die zu optimierende Zielfunktion berücksichtigen. Wir erläutern dies am Ende von diesem Abschnitt am Beispiel des Rucksackproblems.

Graphfärben

Eine klassische Instanz des Graphfärbe-Problems ist die Frage nach der Färbung einer Landkarte, in der benachbarte Länder unterschiedlich gefärbt werden sollen und die Gesamtanzahl an verwendeten Farben minimal ist.

Graph-Färben:

Für planare Graphen (z.B. Landkarten) sind stets 4 Farben ausreichend.



Die allgemeine Formulierung des Graphfärbe-Problems ist wie folgt: Gegeben ist ein ungerichteter Graph $G = (V, E)$, dessen Knoten so zu färben sind, daß benachbarte Knoten unterschiedlich gefärbt sind und daß die Anzahl der verwendeten Farben minimal ist. Die Färbung der Knoten lässt sich durch eine surjektive Abbildung $F : V \rightarrow \{1, 2, \dots, m\}$ formalisieren, so daß $F(v) \neq F(w)$ für $(v, w) \in E$. Minimalität einer Färbung steht für die Minimalität von m .

Das Graphfärbe-Problem stellt sich in zahlreichen Anwendungen. Das Turnierplanungs-Problem wurde bereits in der ersten Vorlesung (Beispiel 1.1.3) genannt. Weitere Beispiele sind die Registerzuordnung in Compilern, Maschinen-Auftragszuteilungs-Probleme oder die Optimierung von Ampelschaltungen.

Graphfärbe-Heuristik. Wir kehren nun zur eigentlichen Fragestellung zurück. Gegeben ist ein ungerichteteter Graph $G = (V, E)$, $V = \{v_1, \dots, v_n\}$, für den eine optimale Färbung der Knoten gesucht ist. Man beachte, daß es stets eine Färbung mit $n = |V|$ Farben gibt. Naheliegend ist die in Algorithmus 47 auf Seite 210 angegebene Vorgehensweise, die im Wesentlichen darauf beruht, eine Farbe m zu wählen und sukzessive einen noch nicht gefärbten Knoten, der (noch) keinen m -farbigen Nachbarn hat, mit Farbe m zu colorieren, bis es keinen solchen Knoten mehr gibt. In diesem Fall wird die nächste Farbe $m + 1$ betrachtet.

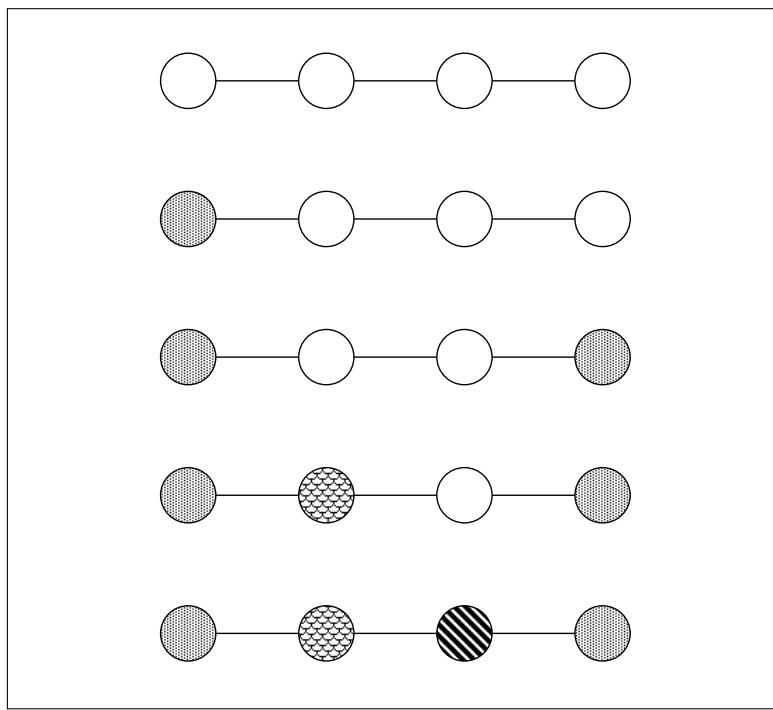
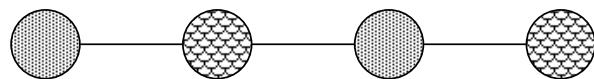


Abbildung 52: Beispiel zur Färbeheuristik

Algorithmus 47 generiert zwar stets eine zulässige Färbung (d.h. je zwei benachbarte Knoten sind unterschiedlich gefärbt), jedoch ist die verwendete Anzahl an Farben nicht notwendig minimal. Abbildung 52 auf Seite 209 zeigt schrittweise eine mögliche Färbung der Knoten eines Graphen mit vier Knoten. Diese verwendet drei Farben, optimal wären jedoch zwei Farben:



Algorithmus 47 Ein einfacher Graphfärbe Algorithmus

markiere alle Knoten als ungefärbt;

$m := 1$;

WHILE es gibt einen noch ungefärbten Knoten **DO**

FOR ALL noch nicht gefärbte Knoten v **DO**

(* Prüfe, ob v mit Farbe m gefärbt werden kann *)

IF v hat keinen Nachbarn w , der mit m gefärbt ist **THEN**

 färbt v mit Farbe m

FI

OD

$m := m + 1$

(* nächste Farbe *)

OD

Tatsächlich können wir keinen derart einfachen und effizienten Algorithmus für das Graphfärbe-Problem erwarten, da bereits die Entscheidungsvariante

- **Gegeben:** ungerichteter Graph $G = (V, E)$ und natürliche Zahl $m \geq 1$
- **Gefragt:** Gibt es eine Färbung für G mit (höchstens) m Farben?

zu den „schwierigen“ Problemen zählt:

Satz 5.2.3 (NP-Vollständigkeit des Graphfärbe-Problems). Die Entscheidungsvariante des Graphfärbe-Problems ist NP-vollständig.

Beweis. Einen nichtdeterministischen Polynomialzeit-Algorithmus für die Entscheidungsvariante des Graphfärbe-Problems erhalten wir mit der Guess & Check Methode:

- wähle nichtdeterministisch eine Zuordnung der Knoten zu den Farben $1, \dots, m$
- prüfe, ob die geratene Zuordnung eine Färbung ist.

Hierzu kann man für jeden Knoten v die Adjazenzliste von v durchlaufen und prüfen, ob jedem Nachbar von v eine andere Farbe als v zugewiesen wurde.

Es bleibt die NP-Härte nachzuweisen. Wir verwenden eine polynomielle Reduktion von 3SAT auf die Entscheidungsvariante des Graphfärbe-Problems. Der Ausgangspunkt ist eine 3KNF-Formel

$$\alpha = \bigwedge_{1 \leq j \leq k} \underbrace{(L_{j,1} \vee L_{j,2} \vee L_{j,3})}_{j\text{-te Klausel}},$$

wobei die $L_{j,r}$'s Literale sind, etwa $L_{j,r} \in \{x_1, \dots, x_n, \neg x_1, \dots, \neg x_n\}$. Zunächst streichen wir aus α alle tautologischen Klauseln, also Klauseln, welche zueinander komplementäre Literale x_i und $\neg x_i$ enthalten. Wir können also im Folgenden annehmen, daß die Literale $L_{j,1}, L_{j,2}, L_{j,3}$ jeder Klausel paarweise nicht komplementär sind. Möglicherweise stimmen zwei (oder alle drei) der Literale überein. Alternativ können wir auch Klauseln mit nur einem oder mit zwei Literalen betrachten. Das ist hier unerheblich.

Unser Ziel ist es, eine in polynomieller Zeit durchführbare Transformation anzugeben, welche α in ein Paar (G, m) bestehend aus einem ungerichteten Graphen G und einer natürlichen Zahl m überführt, so daß G genau dann eine Färbung mit m Farben besitzt, wenn α erfüllbar ist.

Wir wählen die Farbanzahl $m = n + 1$, wobei n die Anzahl an Aussagensymbolen (Variablen x_i) der Formel α ist. Die Knotenmenge von G besteht aus der Literalmenge, k Klauselknoten $\kappa_1, \dots, \kappa_k$ sowie $n + 1$ Farbknoten c_0, \dots, c_n :

$$V = \underbrace{\{x_1, \dots, x_n, \neg x_1, \dots, \neg x_n\}}_{\text{Literale}} \cup \underbrace{\{\kappa_1, \dots, \kappa_k\}}_{\text{Klauselknoten}} \cup \underbrace{\{c_0, c_1, \dots, c_n\}}_{\text{Farbknoten}}.$$

Die Kantenrelation E ist $E_0 \cup E_\alpha$, wobei die Kantenrelation E_0 von α unabhängig ist und dazu dient, jede $(n + 1)$ -Färbung als Belegung interpretieren zu können. E_α wird in Abhängigkeit von α gewählt.

Die Kantenrelation E_0 verbindet zueinander komplementäre Literale x_i mit $\neg x_i$ und sowie die Literale x_i und $\neg x_i$ mit den Farbknoten c_ℓ für $\ell \in \{1, \dots, n\} \setminus \{i\}$. Ferner enthält E_0 je eine Kante zwischen zwei Farbknoten c_h, c_ℓ für $h \neq \ell$. Also:

$$\begin{aligned} E_0 = & \{(x_i, \neg x_i) : 1 \leq i \leq n\} \cup \{(c_h, c_\ell) : 0 \leq h < \ell \leq n\} \\ & \cup \{(x_i, c_\ell), (\neg x_i, c_\ell) : 1 \leq i \leq n, 1 \leq \ell \leq n, i \neq \ell\} \end{aligned}$$

Die Kanten zwischen den Farbknoten stellen sicher, daß in jeder Färbung von G die c_ℓ 's paarweise verschiedene Farben haben. (Die c -Knoten bilden eine Clique.) Wir können gedanklich also die c 's mit Farben identifizieren.

Da x_i zu allen Farbknoten, ausser c_i und c_0 , benachbart ist, hat x_i die Farbe c_0 oder c_i , falls eine Färbung mit $n + 1$ Farben vorliegt. Dasselbe gilt für das negative Literal $\neg x_i$, wobei die Kante zwischen x_i und $\neg x_i$ sicherstellt, daß einer der Literalknoten x_i oder $\neg x_i$ mit c_0 , der andere mit c_i gefärbt ist.

Dies erlaubt es, Farbe c_0 mit dem Wahrheitswert true zu identifizieren und c_i als false zu lesen. In diesem Sinn hat jede $(n + 1)$ -Färbung für V mit der Kantenrelation E_0 eine Interpretation als Belegung der Literale mit Wahrheitswerten. Auch umgekehrt kann jeder Belegung eine $(n + 1)$ -Färbung zugeordnet werden, nämlich Farbe ℓ für den Farbknoten c_ℓ und Farbe 0 für jedes mit true belegte Literal. Ist x_i (bzw. das negative Literal $\neg x_i$) mit false belegt, so färben wir x_i (bzw. $\neg x_i$) mit der Farbe i .

Die Klauselknoten κ_j bleiben in der Transformation „Färbung \leftrightarrow Belegung“ unberücksichtigt. Sie werden in der Kantenrelation E_α eingesetzt und sollen erzwingen, dass die m -Färbungen den erfüllenden Belegungen für α entsprechen.

Die Kantenrelation E_α verbindet jeden Klauselknoten κ_j mit

- dem Farbknoten c_0 ,
- den Farbknoten c_i , wobei weder x_i noch $\neg x_i$ in der j -ten Klausel vorkommen ($1 \leq i \leq n$),
- den Literalknoten $L_{j,1}, L_{j,2}, L_{j,3}$, für die in der j -ten Klausel vorkommenden Literale.

Die einzige Chance für eine $(n + 1)$ -Färbung von G ist also κ_j mit einer der drei Farben $c_{j,1}, c_{j,2}, c_{j,3}$ zu versehen, wobei $c_{j,r} = c_i$, falls $L_{j,r} \in \{x_i, \neg x_i\}$. Dies ist jedoch nur dann möglich, wenn einer der drei Literalknoten $L_{j,1}, L_{j,2}, L_{j,3}$ mit c_0 gefärbt ist, also gedanklich den Wahrheitswert true hat.

Korrektheit der Konstruktion. Zunächst ist klar, daß G in polynomieller Zeit aus α konstruiert werden kann. Liegt eine $(n + 1)$ -Färbung $F : V \rightarrow \{0, 1, \dots, n\}$ für $G = (V, E_0 \cup E_\alpha)$ vor, so betrachten wir die Belegung

$$x_i \mapsto \text{true gdw } F(x_i) = F(c_0).$$

Man verifiziert nun leicht, daß α unter dieser Belegung den Wahrheitswert true hat.

Ist umgekehrt eine erfüllende Belegung für α gegeben, so definieren wir eine Färbung $F : V \rightarrow \{0, 1, \dots, n\}$ wie folgt: Wir setzen $F(c_\ell) = \ell$ und

$$F(L) = \begin{cases} i & : \text{falls } L \in \{x_i, \neg x_i\} \text{ und } L \text{ ist mit false belegt} \\ 0 & : \text{sonst} \end{cases}$$

In der j -ten Klausel wählen wir eines der Literale $L_{j,r}$, welches unter der gegebenen Belegung den Wahrheitswert true hat. Ist $L_{j,r} \in \{x_i, \neg x_i\}$, so setzen wir

$$F(\kappa_j) = i.$$

Man überlegt sich auch hier leicht, daß F tatsächlich eine Färbung von G ist. \square

Graphfärbe-Optimierungs-Algorithmus. Für das Graphfärbe-Problem benutzen wir nun eine simple Strategie, welche darauf beruht, sukzessive die Werte $m = 1, 2, \dots, n$ (wobei $n = |V|$ die Knotenanzahl ist) zu betrachten und jeweils prüfen, ob es eine m -Färbung, d.h. Färbung mit m Farben, für G gibt. Siehe Algorithmus 48. Asymptotisch besser ist die Verwendung einer binären Suche nach dem kleinsten Wert m , so dass G mit m Farben gefärbt werden kann.

Algorithmus 48 Graphfärbe-Optimierungs-Algorithmus

```

 $m := 0;$ 
REPEAT
 $m := m + 1;$ 
 $Färben(m, 1, \langle \cdot \rangle)$  (* Backtracking-Algorithmus (s.u.) *)
UNTIL Färbung gefunden

```

Algorithmus 48 arbeitet mit einem Backtracking-Algorithmus $Färben(m, i, \langle x_1, \dots, x_{i-1} \rangle)$, welcher in Algorithmus 49 angegeben ist. Dieser setzt eine beliebige, aber feste Knotennumerierung v_1, \dots, v_n voraus und hat als Eingabe

- die Farbanzahl m ,
- den Index i des Knotens v_i , der gefärbt werden soll,
- eine zulässige m -Färbung $\langle x_1, \dots, x_{i-1} \rangle$ für die Knoten v_1, \dots, v_{i-1} .

Gesucht ist nun eine Farbe $x_i = y \in \{1, \dots, m\}$ für Knoten v_i , so daß $\langle x_1, \dots, x_i \rangle$ eine zulässige m -Färbung für die Knoten v_1, \dots, v_i ist. Hierzu wird jeder Wert $y \in \{1, \dots, m\}$ als potentieller Lösungskandidat angesehen, für den die Randbedingung

„Ist $(v_j, v_i) \in E$, $1 \leq j < i$, so ist $x_j \neq y$, d.h. die Farbe x_j von v_j stimmt nicht mit der Farbe y von v_i überein.“

geprüft wird.

Algorithmus 49 *Färben*($m, i, \langle x_1, \dots, x_{i-1} \rangle$)

y := 1;
WHILE $y \leq m$ **DO**
 IF $\langle x_1, \dots, x_{i-1}, y \rangle$ ist eine zulässige Färbung der ersten i Knoten **THEN**
 IF $i = n$ **THEN**
 return „ m -Färbung gefunden“
 ELSE
 Färben($m, i + 1, \langle x_1, \dots, x_{i-1}, y \rangle$)
 FI
 $y := y + 1$;
 FI
OD

Die Laufzeit des skizzierten Algorithmus ist leider hoffnungslos schlecht. Unser Algorithmus verwendet einen Aufzählungsbaum vom Verzweigungsgrad m . Dieser hat bis zu

$$\sum_{i=0}^n m^i = \frac{m^{n+1} - 1}{m - 1}$$

Knoten ($m > 1$ vorausgesetzt). An jedem Knoten entstehen die Kosten $\mathcal{O}(mn)$. Die gesamte worst-case Laufzeit kann daher durch $\mathcal{O}(nm^{n+2})$ und $\Omega(m^n)$ abgeschätzt werden. Die maximale Rekursionstiefe ist n . Daher ist die Platzkomplexität linear in der Knotenzahl, also $\Theta(n)$, vorausgesetzt man verwendet globale Variablen x_1, \dots, x_n anstelle der Parametrisierung mit den x_i 's.

Das Rucksackproblem

Gegeben sind n Objekte mit Gewichten $w_1, \dots, w_n \in \mathbb{N}_{>0}$, Gewinnen $p_1, \dots, p_n \in \mathbb{N}_{>0}$ sowie ein Gewichtslimit (Rucksackkapazität) $K \in \mathbb{N}_{>0}$. Gesucht ist eine optimale Füllung des Rucksacks, d.h. eine Auswahl der Objekte, so daß das Gewichtslimit K nicht überschritten wird und der erzielte Gewinn optimal ist.

Wir unterscheiden zwei Varianten von RP. Das *rationale Rucksack-Problem* erlaubt es, Bruchteile der Objekte auszuwählen. Gesucht ist also ein n Tupel (x_1, \dots, x_n) mit rationalen Zahlen $x_i \in [0, 1]$, so daß $\text{Gewicht}(x_1, \dots, x_n) \leq K$ und $\text{Gewinn}(x_1, \dots, x_n)$ maximal ist.

Bei dem $\{0, 1\}$ -Rucksack-Problem dagegen dürfen die Objekte nur ganz oder gar nicht mitgenommen werden. Gesucht ist also ein Bittupel (x_1, \dots, x_n) mit Werten $x_i \in \{0, 1\}$, so daß $\text{Gewicht}(x_1, \dots, x_n) \leq K$ und $\text{Gewinn}(x_1, \dots, x_n)$ maximal ist. Dabei ist für $r \in \{1, \dots, n\}$ und $x_1, \dots, x_r \in [0, 1]$:

$$\begin{aligned}\text{Gewicht}(x_1, \dots, x_r) &= x_1 \cdot w_1 + \dots + x_r \cdot w_r \\ \text{Gewinn}(x_1, \dots, x_r) &= x_1 \cdot p_1 + \dots + x_r \cdot p_r.\end{aligned}$$

Das $\{0, 1\}$ -Rucksackproblem ist NP-vollständig. Wir betrachten nun zwei sehr einfache, effiziente Algorithmen. Der erste Algorithmus löst das rationale Rucksackproblem und liefert damit eine obere Schranke für den maximalen Gewinn, welcher durch das $\{0, 1\}$ -Rucksackproblem erzielt werden kann. Der zweite Algorithmus liefert eine untere Schranke für den optimalen Gewinn des $\{0, 1\}$ -Rucksackproblems und kann als heuristische (eventuell nicht-optimale) Methode eingesetzt werden. Beide Algorithmen arbeiten jeweils mit einer Sortierung der Objekte nach den *relativen Gewinnen*. Wir nehmen o.E. an, daß

$$p_1/w_1 \geq p_2/w_2 \geq \dots \geq p_n/w_n.$$

Algorithmus 50 packt die ersten Objekte $1, \dots, i$ vollständig ein und den maximal möglichen Anteil des $(i+1)$ -ten Objekts, vorausgesetzt $w_1 + \dots + w_i \leq K < w_1 + \dots + w_i + w_{i+1}$. Algorithmus 51 auf Seite 217 nimmt dagegen nur die ersten i Objekte mit. Die Laufzeiten der beiden Algorithmen sind jeweils $\Theta(n \log n)$, wobei n die Anzahl an Objekten ist. Diese entsteht durch die Sortierung der Objekte nach ihren relativen Gewinnen. Die Rechenzeit für die restlichen Schritte, in denen die Werte x_1, \dots, x_n berechnet werden, ist jeweils linear in n . Der zusätzliche Platzbedarf ist jeweils konstant, da – neben den Eingabewerten p_i, w_i, K – nur die Variablen w, p und i benötigt werden.

Beispiel 5.2.4. Für die in der Tabelle angegebenen Instanz des Rucksackproblems

Objekt i	1	2	3
Gewinn p_i	30	50	60
Gewicht w_i	1	2	3
relativer Gewinn p_i/w_i	30	25	20

mit der Rucksackkapazität $K = 5$ ergeben sich durch die beiden Algorithmen folgende Lösungen. Für das rationale Rucksackproblem (Algorithmus 50) erhalten wir die Lösung $(x_1, x_2, x_3) = (1, 1, 2/3)$ mit dem erzielten Gewinn

Algorithmus 50 Algorithmus für das rationale Rucksackproblem

sortiere die Objekte absteigend nach ihren relativen Gewinnen, etwa $p_1/w_1 \geq \dots \geq p_n/w_n$

```
w := 0;                                (* Gewicht der aktuellen Rucksackfüllung *)
p := 0;                                (* Gewinn der aktuellen Rucksackfüllung *)
i := 1

WHILE  $i \leq n$  DO
  IF  $w < K$  THEN
    IF  $w + w_i \leq K$  THEN
       $x_i := 1$ ;                      (* packe Objekt  $i$  vollständig in den Rucksack *)
       $w := w + w_i$ ;
       $p := p + p_i$ ;
    ELSE
       $x_i := (K - w)/w_i$             (* packe größtmöglichen Anteil von Objekt  $i$  in den
                                         Rucksack *)
    FI
     $w := w + x_i w_i$ ;             (*  $w := K$  *)
     $p := p + x_i p_i$ 
  ELSE
     $x_i := 0$                       (*  $w = K$ , d.h. Rucksack ist bereits vollständig gefüllt *)
  FI
   $i := i + 1$ 

OD
Gib  $(x_1, \dots, x_n)$  und  $p$  aus.
```

Algorithmus 51 Heuristik für das $\{0, 1\}$ -Rucksackproblem

sortiere die Objekte absteigend nach ihren relativen Gewinnen, etwa $p_1/w_1 \geq \dots \geq p_n/w_n$

```

 $w := 0;$                                 (* Gewicht der aktuellen Rucksackfüllung *)
 $p := 0;$                                 (* Gewinn der aktuellen Rucksackfüllung *)
 $i := 1$ 

WHILE  $i \leq n$  DO
  IF  $w + w_i \leq K$  THEN
     $x_i := 1;$                             (* packe Objekt  $i$  in den Rucksack *)
     $w := w + w_i;$ 
     $p := p + p_i$ 
  ELSE
     $x_i := 0$                             (* Mitnahme von Objekt  $i$  würde Kapazität übersteigen *)
  FI
   $i := i + 1$ 

OD
Gib  $(x_1, \dots, x_n)$  und  $p$  aus.

```

$$\text{Gewinn} \left(1, 1, \frac{2}{3}\right) = 30 + 50 + \frac{2}{3} \cdot 60 = 120.$$

Für das $\{0, 1\}$ -Rucksackproblem liefert Algorithmus 51 die Lösung $(x_1, x_2, x_3) = (1, 1, 0)$, für welche der Gewinn $30 + 50 = 80$ erreicht wird. Für das $\{0, 1\}$ -Rucksackproblem ist dies zwar eine zulässige Rucksackfüllung, jedoch ist diese *nicht optimal*, da durch $(x_1, x_2, x_3) = (0, 1, 1)$ der Gewinn $50 + 60 = 110$ erreicht wird. \square

Wir zeigen nun die Korrektheit von Algorithmus 50 für das rationale Rucksackproblem.

Satz 5.2.5 (Optimale Lösung des rationalen Rucksackproblems). Algorithmus 50 liefert eine optimale Lösung für das rationale Rucksackproblem.

Beweis. Wir können uns auf den Fall $w_1 + \dots + w_n > K$ beschränken, da andernfalls $(x_1, \dots, x_n) = (1, \dots, 1)$ die triviale optimale Lösung ist und diese offenbar durch den angegebenen Algorithmus berechnet wird.

Sei also $w_1 + \dots + w_n > K$. Weiter nehmen wir $p_1/w_1 \geq p_2/w_2 \geq \dots \geq p_n/w_n$ an. Die durch Algorithmus 50 berechnete Lösung hat die Form

$$\underbrace{(x_1, \dots, x_i)}_{\text{Einsen}}, x_{i+1}, \underbrace{x_{i+2}, \dots, x_n}_{\text{Nullen}} = (1, 1, \dots, 1, b, 0, \dots, 0),$$

beginnend mit i Einsen, einem Bruchteil $x_{i+1} = b \in [0, 1[$ des $(i+1)$ -ten Objekts und Nullen für die Objekte $i+2, \dots, n$. Wir zeigen nun, daß keine Modifikation der berechneten Rucksackfüllung, welche ein Fragment eines mitgenommenen Objekts ℓ durch ein nicht mitgenommenen Fragments eines anderen Objekts j ersetzt, einen höheren Gewinn erzielen kann. Es muß dann gelten

$$\ell \leq i + 1 \leq j \text{ und } \ell < j. \quad (*)$$

Weiter nehmen wir an, daß die veränderte Rucksackfüllung das x -Fragment von Objekt ℓ und das y -Fragment von Objekt j nimmt. Intuitiv ersetzen wir also das $(x_\ell - x)$ -Fragment von Objekt ℓ durch das $(y - x_j)$ -Fragment von Objekt j . Insbesondere gilt dann $0 \leq x < x_\ell$, $1 \geq y > x_j$ und $(y - x_j)w_j = (x_\ell - x)w_\ell$, also

$$y - x_j = (x_\ell - x) \cdot \frac{w_\ell}{w_j}. \quad (**)$$

Der erzielte Gewinn der modifizierten Rucksackfüllung ist

$$\underbrace{\sum_{k=1}^n x_k p_k}_{\text{Gewinn der berechneten Rucksackfüllung}} - (x_\ell - x)p_\ell + (y - x_j)p_j$$

Wegen $\ell < j$ (siehe $(*)$) gilt: $p_\ell/w_\ell \geq p_j/w_j$. Daher gilt wegen $(**)$:

$$(y - x_j)p_j - (x_\ell - x)p_\ell = (x_\ell - x)w_\ell \frac{p_j}{w_j} - (x_\ell - x)p_\ell = w_\ell \cdot \underbrace{(x_\ell - x)}_{>0} \left(\underbrace{\frac{p_j}{w_j} - \frac{p_\ell}{w_\ell}}_{\leq 0} \right) \leq 0$$

Also ist der Gewinn der modifizierten Rucksackfüllung nicht größer als der unter der berechneten Rucksackfüllung. Da sich jede andere Rucksackfüllung, welche die Kapazität ausschöpft, durch mehrere derartige Modifikationen ergibt, in denen ein Fragment $z = x_\ell - x$ von Objekt ℓ durch ein Fragment $z' = y - x_j$ eines anderen Objekts j mit $\ell < j$ ersetzt wird, ist die berechnete Lösung optimal. Wegen der Annahme $w_1 + \dots + w_n > K$ und da alle Gewinne positiv sind, kommen nur Rucksackfüllungen, welche die Kapazität ausschöpfen, als optimale Lösungskandidatinnen in Frage. \square

Weiter halten folgende offensichtliche Aussage fest:

Lemma 5.2.6 (Obere und untere Schranken des $\{0, 1\}$ -Rucksackproblems). *Der Gewinn einer optimalen Lösung des rationalen Rucksackproblems (Algorithmus 50) ist eine obere Schranke für den Gewinn einer optimalen Lösung für das zugehörige $\{0, 1\}$ -Rucksackproblem. Algorithmus 51 liefert eine untere Schranke für den Gewinn einer optimalen Lösung des $\{0, 1\}$ -Rucksackproblems.*

Im Folgenden werden wir Algorithmen vorstellen, die eine optimale Rucksackfüllung für das $\{0, 1\}$ -Rucksack-Problem ermitteln.

Backtracking-Algorithmus für das $\{0, 1\}$ -Rucksackproblem. Wir lösen das $\{0, 1\}$ -Rucksackproblem mit einem Backtracking-Algorithmus, dem ein binärer Aufzählungsbaum zugrundeliegt. In jedem inneren Knoten der Tiefe $i - 1$ findet die Entscheidung $x_i = 0$ (i -tes Objekt wird nicht mitgenommen) oder $x_i = 1$ (i -tes Objekt wird mitgenommen) statt. Dies entspricht dem Berechnungsbaum der nicht-deterministischen Guess & Check Methode „rate n Bits x_1, \dots, x_n und prüfe, ob die Mitnahme der Objekte i mit“.

$x_i = 1$ eine zulässige Rucksackfüllung ergibt“. In diesem Aufzählungsbaum suchen wir nun nach einer Lösung, welche den Gewinn maximiert.

Der Backtracking-Algorithmus (siehe Algorithmus 52 auf Seite 219) benutzt einen rekursiven Algorithmus $Rucksack(i, \langle x_1, \dots, x_{i-1} \rangle)$, welcher initial mit $Rucksack(1, \langle \cdot \rangle)$ aufgerufen wird und welcher Teile des Aufzählungsbaums durchläuft. Dieser verwendet eine globale Variable $optgewinn$, die initial den Wert $-\infty$ hat und die jeweils den Gewinn der besten bisher gefundenen Lösung angibt. Das n -Tupel $optlsg$ stellt am Ende eine optimale Rucksackfüllung dar.

Algorithmus 52 Backtracking Algorithmus $Rucksack(i, \langle x_1, \dots, x_{i-1} \rangle)$ (Variante 1)

```

IF  $i \leq n$  THEN
     $Rucksack(i + 1, \langle x_1, \dots, x_{i-1}, 0 \rangle)$ 
    IF  $Gewicht(x_1, \dots, x_{i-1}, 1) \leq K$  THEN
         $Rucksack(i + 1, \langle x_1, \dots, x_{i-1}, 1 \rangle)$ 
    FI
ELSE
    IF  $optgewinn < Gewinn(x_1, \dots, x_n)$  THEN
         $optgewinn := Gewinn(x_1, \dots, x_n);$ 
         $optlsg := (x_1, \dots, x_n)$ 
    FI
FI

```

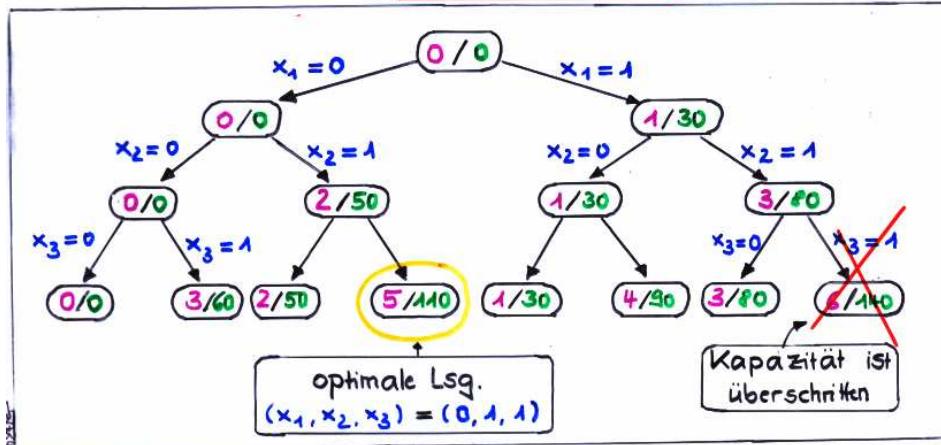
Aus Effizienzgründen sollten der jeweils aktuelle Gewinn und das aktuelle Gewicht (also die Zahlen $Gewinn(x_1, \dots, x_{i-1})$ und $Gewicht(x_1, \dots, x_{i-1})$) als weitere Parameter in $Rucksack(i, \dots)$ übergeben werden. Dies entspricht der Beschriftung der Knoten im Aufzählungsbaum mit dem jeweiligen Gewinn und Gewicht.

Aufzählungsbaum für das $\{0,1\}$ -Rucksackproblem

j	1	2	3
T_j	30	50	60
w_j	1	2	3

Beschriftung der Knoten mit dem aktuellen Gewicht und Gewinn

Kapazität $K = 5$



Das skizzierte Verfahren zum Lösen des $\{0,1\}$ -Rucksackproblems hat die worst-case Laufzeit $\Theta(2^n)$, da der Aufzählungsbaum

$$1 + 2 + 4 + \dots + 2^n = 2^{n+1} - 1$$

Knoten hat und pro Knoten nur konstante Kosten entstehen; vorausgesetzt die aktuellen Gewichte und Gewinne werden als Parameter übergeben. Der Platzbedarf ist $\Theta(n)$, da die Rekursionstiefe gleich n ist. Dies setzt jedoch voraus, dass auf die Parametrisierung mit den x_i 's verzichtet wird. Tatsächlich sind die x_i 's überflüssig, wenn man nur an dem maximalen Gewinn interessiert ist, nicht aber an einer optimalen Rucksackfüllung. Zur Berechnung einer optimalen Lösung (wie im Algorithmus angedeutet) können die x_i 's als globale Variablen deklariert werden.

Das folgende Verfahren verbessert zwar nicht die worst-case Laufzeit, kann jedoch in vielen Fällen zu wesentlich kürzeren (tatsächlichen) Rechenzeiten führen. Die Idee ist, die Randbedingungen zu verschärfen, um so möglichst viele Äste des Aufzählungsbaums zu beschneiden. Hierzu verwenden wir eine Beschränkungsfunktion, die neben den eigentlichen Randbedingungen (der Forderung, daß $Gewicht(\bar{x}) \leq K$) auch die zu optimierende Zielfunktion $Gewinn(\bar{x})$ berücksichtigt.

Die Grobidee besteht darin, obere Schranken für die beste Lösung in den Teilbäumen zu benutzen. Falls ein Knoten vorliegt, dessen obere Schranke schlechter (oder höchstens genauso gut) wie die beste bisher gefundene Lösung ist, dann kann auf die Expansion des betreffenden Knotens verzichtet werden.

Berechnung der oberen Schranken. Sei $\bar{x} = \langle x_1, \dots, x_{i-1} \rangle$ ein Knoten im Aufzählungsbaum für das $\{0,1\}$ -Rucksackproblem mit n Objekten. Es sei $opt(\bar{x})$ der maximale

Gewinn eines Blatts $\langle x_1, \dots, x_{i-1}, \dots \rangle$ in dem Teilbaum von \bar{x} . Offenbar gilt:

$$opt(\bar{x}) = Gewinn(\bar{x}) + \gamma(\bar{x}),$$

wobei

$\gamma(\bar{x}) =$ optimaler Gewinn des $\{0, 1\}$ -Rucksackproblems
mit den Objekten $i, i+1, \dots, n$ und der
Restkapazität $K' = K - Gewicht(\bar{x})$.

Die obere Schranke $ub(\bar{x}) = ub(x_1, \dots, x_{i-1})$ für eine $\{0, 1\}$ -Rucksackfüllung, in der die Mitnahme der ersten $i-1$ Objekte durch x_1, \dots, x_{i-1} festgelegt ist, berechnen wir mit der in Algorithmus 50 auf Seite 216 beschriebenen Methode zum Lösen des rationalen Rucksackproblems. (Die Abkürzung $ub(\dots)$ steht für „upper bound“.)

Algorithmus 50 angewandt auf das rationale Rucksackproblem mit $n-i+1$ Objekten und den Gewichten w_i, \dots, w_n , den Gewinnen p_i, \dots, p_n und der Restkapazität $K' = K - Gewicht(\bar{x})$ liefert eine obere Schranke $\gamma^+(\bar{x})$ für $\gamma(\bar{x})$. Eine obere Schranke $ub(\bar{x})$ für die beste Lösung im Teilbaum mit Wurzel \bar{x} erhalten wir durch

$$ub(\bar{x}) = Gewinn(\bar{x}) + \gamma^+(\bar{x}).$$

Offenbar gilt $opt(\bar{x}) \leq ub(\bar{x})$.

Wir verwenden diese Beobachtung, um irrelevante Teile des Aufzählungsbaums zu ignorieren: Ist $\bar{x} = \langle x_1, \dots, x_{i-1} \rangle$ der aktuelle Knoten und ist $ub(\bar{x}) \leq optgewinn$ der Wert der bisher besten gefundenen Lösung, dann wird Knoten \bar{x} und dessen Teilbaum ignoriert. Diese Ideen sind in Algorithmus 53 auf Seite 221 zusammengefasst.

Algorithmus 53 Backtracking Algorithmus $Rucksack(i, \langle x_1, \dots, x_{i-1} \rangle)$ (Variante 2)

```

IF  $i \leq n$  THEN
  IF  $ub(x_1, \dots, x_{i-1}) > optgewinn$  THEN
     $Rucksack(i+1, \langle x_1, \dots, x_{i-1}, 0 \rangle)$ 
    IF  $Gewicht(x_1, \dots, x_{i-1}, 1) \leq K$  THEN
       $Rucksack(i+1, \langle x_1, \dots, x_{i-1}, 1 \rangle)$ 
    FI
  FI
ELSE
  IF  $optgewinn < Gewinn(x_1, \dots, x_n)$  THEN
     $optgewinn := Gewinn(x_1, \dots, x_n)$ 
  FI
FI

```

Ein Beispiel ist auf folgender Folie angegeben. Hier liefert die Analyse des linken Teilbaums der Wurzel eine Lösung mit dem Gesamtgewinn 5. Also wird $optgewinn = 5$ gesetzt. Die obere Schranke $opt(1)$ für den Knoten $\bar{x} = \langle 1 \rangle$ ist $ub(1) = p_1 + \gamma^+(1) = 1 + 3 = 4 < 5$, also wird auf die Analyse des rechten Teilbaums verzichtet.

Rucksackproblem

j	1	2	3
P_j	1	3	2
w_j	1	1	1

Kapazität $K = 2$

Restkapazität 1

j	2	3
P_j	3	2
w_j	1	1

$x_1 = 0$

$x_1 = 1$

$$\text{gewinn}(0, 1, 1) = 5$$

$$\begin{aligned} \text{Gewinn der} \\ \text{besten Lsg.} \\ \leq 4 \\ = 1 + 3 \end{aligned}$$

Bemerkung 5.2.7 (Effiziente Berechnung der oberen Schranken). Mit einer geschickten Vorverarbeitung der Objekte und geeigneter Parametrisierung können die oberen Schranken $\gamma^+(\bar{x})$ in linearer Zeit berechnet werden. Hierzu verwendet man eine aufsteigende Sortierung der Objekte nach ihren relativen Gewinnen, etwa $p_1/w_1 \leq \dots \leq p_n/w_n$. Der Algorithmus für das rationale Rucksackproblem angewandt für die letzten $n - i + 1$ Objekte und der Kapazität $K' = K - \text{Gewicht}(x_1, \dots, x_{i-1})$ liefert dann eine Lösung der Form $(x_i, \dots, x_n) = (0, \dots, 0, b, 1, \dots, 1)$. Es gilt dann

$$\gamma^+(\langle x_1, \dots, x_{i-1}, 0 \rangle) = \gamma^+(\langle x_1, \dots, x_{i-1} \rangle) - x_i p_i.$$

Der Wert $\gamma^+(\langle x_1, \dots, x_{i-1}, 1 \rangle)$ ergibt sich, indem das Tupel $(0, \dots, 0, b, 1, \dots, 1)$ zu einem Tupel der Form $(0, \dots, 0, 0, 0, \dots, 0, c, 1, \dots, 1)$ durch sukzessives Entfernen von Objekten überführt wird bis sich die Restkapazität $K'' = K' - x_i = K - \text{Gewicht}(x_1, \dots, x_{i-1}, 1)$ ergibt. \square

Das skizzierte Verfahren hat im schlimmsten Fall dieselbe Laufzeit wie die einfache Version. Jedoch liegt die tatsächliche Rechenzeit oftmals erheblich unter der der einfachen Version. Im folgenden Abschnitt stellen wir weitere Verbesserungsmöglichkeiten vor. Diese zielen darauf ab, möglichst schnell zu einer optimalen Lösung zu gelangen, so daß durch die Beschränkungstechnik möglichst große Teile des Aufzählungsbaums eingespart werden können. An der exponentiellen worst-case Laufzeit können jedoch auch diese zusätzlichen Tricks nichts ändern.

5.2.2 Branch & Bound

Im Wesentlichen beruhen die Backtracking-Algorithmen auf einem Preorder-Durchlauf des dynamisch erzeugten (Teils des) Aufzählungsbaums. Die Generierung der Söhne eines Knotens $\bar{x} = \langle x_1, \dots, x_{i-1} \rangle$ erfolgt nach einem strengen Schema. Z.B. generiert (und expandiert) der für das Rucksackproblem vorgestellte Backtracking Algorithmus stets zuerst den Knoten $\langle \bar{x}, 0 \rangle$, dann den Sohn $\langle \bar{x}, 1 \rangle$, vorausgesetzt, dass $Gewicht(\bar{x}, 1) \leq K$. Die Grundidee von Branch & Bound Techniken besteht in der Verwendung von Heuristiken, die ein zielgerichtetes Suchen im Aufzählungsbaums ermöglichen.³⁸

- Branch: expandiere zuerst „vielversprechende“ Knoten
- Bound: ignoriere Teilbäume, in denen nachweislich keine (optimale) Lösung gefunden werden kann.

Branch & Bound-Techniken werden häufig für Optimierungsprobleme verwendet, die sich nicht optimal mit der Greedy-Methode (siehe den folgenden Abschnitt 5.4 auf Seite 249 ff) lösen lassen. Es gibt sehr viele Varianten von Branch & Bound Algorithmen, die sich in der Art und Weise, wie die Schranken eingesetzt werden, unterscheiden. Wir erläutern mögliche Vorgehensweisen für Maximierungsprobleme. Wie für die Backtracking Algorithmen identifizieren wir die Knoten im Aufzählungsbaum mit Tupeln $\bar{x} = \langle x_1, \dots, x_{i-1} \rangle$. Sei opt eine Variable, die jeweils den Wert der besten bisher gefundenen Lösung speichert. $opt(\bar{x})$ bezeichne den Wert einer besten Lösung im Teilbaum mit Wurzel \bar{x} . Untere und obere Schranken $ub(\bar{x})$, $lb(\bar{x})$ für $opt(\bar{x})$ können für die Beschränkungstechnik wie folgt eingesetzt werden.

- Obere Schranken („upper bounds“): Ist die obere Schranke $ub(\bar{x})$ höchstens so gut wie der Wert der besten bisher gefundenen Lösung (also $ub(\bar{x}) \leq opt$), dann wird der Knoten \bar{x} samt seiner Nachfolger ignoriert.
- Untere Schranken („lower bounds“): Stimmt die untere Schranke $lb(\bar{x})$ mit der oberen Schranke $ub(\bar{x})$ überein, dann gilt $lb(\bar{x}) = opt(\bar{x}) = ub(\bar{x})$. D.h. in diesem Fall kann man ohne explizite Analyse des Teilbaums mit Wurzel \bar{x} auf den Wert einer optimalen Lösung im Teilbaum mit Wurzel \bar{x} schließen.

Wir setzen obere und untere Schranke für die Beschränkungsstrategie ein. In solchen Fällen, in denen „hilfreiche“ untere Schranken nicht effizient berechnet werden können (bzw. keine effizienten Verfahren bekannt sind), kann man für Maximierungsprobleme auf die unteren Schranken verzichten. Dies entspricht der Verwendung der trivialen unteren Schranken $lb(\bar{x}) = -\infty$. Liegt ein Minimierungsproblem vor, dann sind die Rollen der unteren und oberen Schranken zu vertauschen.

³⁸Der Begriff „Branch & Bound-Algorithmus“ wird in der Literatur mit unterschiedlicher Bedeutung verwendet. Tatsächlich kann man mehrere Entwurfsstrategien durch Branch & Bound Techniken erweitern. Die hier vorgestellte Methode ist eine Erweiterung des Backtracking und wird von manchen Autoren als „Backtracking mit Branch & Bound“ bezeichnet.

Bemerkung 5.2.8 (Wert einer optimalen Lösung versus eine/alle Lösungen). Die genannte Vorgehensweise ist zunächst nur dann gerechtfertigt, wenn man nur an dem Wert einer optimalen Lösung interessiert ist, nicht aber an der Lösung selbst. Will man eine optimale Lösung (und nicht nur deren Wert) berechnen, dann muß der vollständige Pfad zu einer optimalen Lösung durchlaufen werden, es sei denn, optimale Lösungen werden bereits mit der Berechnung der oberen/unteren Schranken mitgeliefert. Sind alle optimalen Lösungen gesucht (anstelle einer einzigen Lösung), dann dürfen die Knoten $\langle x_1, \dots, x_{i-1} \rangle$ mit $ub(x_1, \dots, x_{i-1}) = opt$ nicht ignoriert werden. \square

Die im Branchschritt verwendete Strategie zur Auswahl des nächsten zu expandierenden Knoten setzt voraus, daß zuerst *alle* Söhne des aktuellen Knotens \bar{x} generiert werden bevor der nächste Knoten und dessen Teilbaum expandiert wird. Dies ist ein wesentliches Unterscheidungsmerkmal zu Backtracking-Algorithmen, in denen der nächste Sohn des expandierenden Knotens erst dann generiert wird, wenn der Teilbaum des zuvor ausgewählten Sohns vollständig analysiert wurde. (Mit der *Expansion* eines Knotens meinen wir die Generierung der Söhne.)

Dieser Sachverhalt lässt sich salopp wie folgt formulieren: Während Backtracking-Algorithmen auf einer DFS-ähnlichen Traversierung des Aufzählungsbaums beruhen, verwenden Branch & Bound-Algorithmen eine andere Durchlaufstrategie, in der der nächste Knoten erst dann expandiert wird, wenn sämtliche direkten Nachfolger des aktuellen Knotens erzeugt wurden. Eine mögliche Vorgehensweise für Branch & Bound-Algorithmen ist ein BFS-Durchlauf des Aufzählungsbaums. Dies entspricht der Organisation der erzeugten noch nicht expandierten Knoten im FIFO-Prinzip. Diese Vorgehensweise ist jedoch oftmals weniger effizient als Methoden, die auch für den Branchschritt heuristische Verfahren (basierend auf oberen und/oder unteren Schranken) verwenden und darauf abzielen, möglichst schnell eine „gute“ Nährungslösungen oder gar eine optimale Lösung zu finden, um so mit der Beschränkungsstrategie große Teile des Aufzählungsbaums einsparen zu können.

Wir stellen zwei Varianten von Branch & Bound Algorithmen vor, die sich durch ihren Branchschritt unterscheiden. Die erste Methode (die sogenannte LIFO-Methode) basiert auf dem LIFO-Prinzip zur Verwaltung der generierten noch nicht expandierten Knoten. Die zweite Methode (die sogenannte LC-Methode) organisiert die Menge aller generierten noch nicht expandierten Knoten in einer Prioritätswarteschlange.

Die LIFO-Methode

Die LIFO-Methode beruht auf einer Durchlaufstrategie des Aufzählungsbaums, die als Mischung aus Tiefen- und Breitensuche angesehen werden kann. Wir verwenden einen DFS-ähnlichen rekursiven Durchlaufalgorithmus, wobei sich die Reihenfolge, in der die Söhne des aktuellen expandierenden Knotens auf den LIFO-Speicher (Rekursionsstack) gelegt werden, aus den oberen Schranken ergibt.

Als Beispiel betrachten wir erneut das $\{0, 1\}$ -Rucksackproblem und erläutern die Vorgehensweise der LIFO-Variante von Branch & Bound. Gegeben sind n Objekte mit positiven Gewinnen p_1, \dots, p_n und Gewichten w_1, \dots, w_n und die Rucksackkapazität K . Ziel ist

es, die Laufzeit des Backtracking-Algorithmus (2. Variante) zu verkürzen, indem eine Heuristik eingesetzt wird, um den nächsten zu expandierenden Knoten zu bestimmen.

Obere und untere Schranken für das Rucksackproblem ergeben sich aus den beiden in Abschnitt 5.2.1 beschriebenen Greedy-Methoden (Algorithmen 50 und 51 auf Seite 216 ff). Sei $\bar{x} = \langle x_1, \dots, x_{i-1} \rangle$ ein Knoten und $opt(\bar{x})$ der maximale Gewinn eines Blatts $\langle x_1, \dots, x_{i-1}, \dots \rangle$ in dem Teilbaum von \bar{x} . Mit den auf Seite 220 eingeführten Bezeichnungen gilt

$$opt(\bar{x}) = \text{Gewinn}(\bar{x}) + \gamma(\bar{x}),$$

wobei $\gamma(\bar{x})$ für den optimalen Gewinn des $\{0, 1\}$ -Rucksackproblems mit den Objekten $i, i+1, \dots, n$ und der Restkapazität $K' = K - \text{Gewicht}(\bar{x})$ steht. Algorithmen 50 und 51 angewandt auf das Rucksackproblem mit $n - i + 1$ Objekten und den Gewichten w_i, \dots, w_n , den Gewinnen p_i, \dots, p_n und der Kapazität $K' = K - \text{Gewicht}(\bar{x})$ liefern Werte $\gamma^-(\bar{x})$ und $\gamma^+(\bar{x})$, so daß

$$\underbrace{\gamma^-(\bar{x})}_{\text{Algorithmus 51}} \leq \gamma(\bar{x}) \leq \underbrace{\gamma^+(\bar{x})}_{\text{Algorithmus 50}}.$$

Die unteren und oberen Schranken für den Knoten \bar{x} ergeben sich wie folgt:

$$ub(\bar{x}) = \text{Gewinn}(\bar{x}) + \gamma^+(\bar{x}), \quad lb(\bar{x}) = \text{Gewinn}(\bar{x}) + \gamma^-(\bar{x}).$$

Offenbar gilt

$$lb(\bar{x}) \leq opt(\bar{x}) \leq ub(\bar{x}).$$

Wir verwenden die oben skizzierte Beschränkungsstrategie an Knoten \bar{x} . Ist $ub(\bar{x}) \leq optgewinn$, wobei $optgewinn$ der Wert der bisher besten gefundenen Lösung ist, dann wird Knoten \bar{x} und dessen Teilbaum ignoriert. Ist $ub(\bar{x}) = lb(\bar{x})$, dann ist $opt(\bar{x}) = ub(\bar{x})$ und eine zugehörige optimale Lösung des betreffenden Teilbaums wurde durch die Heuristik berechnet. Es besteht also in diesem Fall keine Notwendigkeit, den Teilbaum von \bar{x} zu analysieren, sofern man nur an einer (statt allen) optimalen Lösung interessiert ist. Ein Beispiel ist auf folgender Folie angegeben:

Rucksackproblem: Einsatz von oberen und unteren Schranken

j	1	2	3
P _j	3	3	2
w _j	3	2	2

Kapazität K = 4

Restkapazität
K' = 4

maximaler
Gewinn
= 5

5 ≤ gewinn ≤ 5

3 ≤ gewinn ≤ 4.5

Folgende Folie zeigt ein Beispiel, in dem zuerst der rechte Teilbaum der Wurzel analysiert wird, da dessen obere Schranke 10 besser als die des linken Teilbaums ist. Die Analyse der rechten Teilbaums findet eine Lösung mit dem Gewinn 9. Da dies die obere Schranke des linken Teilbaums ist, kann der linke Teilbaum ignoriert werden.

Branch & Bound für das Rucksack-Problem

j	1	2	3
P _j	3	5	6
w _j	1	2	3

Kapazität K = 4

Restkapazität 3

untere Schranke

$$= 3 + 5 = 8$$

obere Schranke

$$= 3 + 5 + \frac{1}{3}6 = 10$$

5 ≤ gewinn ≤ 9

8 ≤ gewinn ≤ 10

Algorithmus 54 Branch & Bound Algorithmus $Rucksack(i, \langle x_1, \dots, x_{i-1} \rangle)$

```
IF  $i \leq n$  THEN
  IF  $ub(x_1, \dots, x_{i-1}) > optgewinn$  THEN
    IF  $ub(x_1, \dots, x_{i-1}) = lb(x_1, \dots, x_{i-1})$  THEN
       $optgewinn := ub(x_1, \dots, x_{i-1})$ 
    ELSE
      ... verzweige an Knoten  $\langle x_1, \dots, x_{i-1} \rangle \dots$  (* Algorithmus 55 *)
    FI
  FI
ELSE
  IF  $optgewinn < Gewinn(x_1, \dots, x_n)$  THEN
     $optgewinn := Gewinn(x_1, \dots, x_n)$ 
  FI
FI
```

Der Branchschritt kann z.B. mit dem in Algorithmus 55 angegebenen Verfahren vorgenommen werden. Die Auswahl des nächsten zu expandierenden Knotens berücksichtigt die oberen Schranken. Derjenige Sohn des aktuellen Knotens wird zuerst expandiert, dessen obere Schranke besser ist. Ist $ub(\bar{x}, 0) > ub(\bar{x}, 1)$, dann wird zuerst der Knoten $\langle \bar{x}, 0 \rangle$ untersucht. Andernfalls umgekehrt. Dabei setzen wir $Gewicht(\bar{x}, 1) \leq K$ voraus. Wie zuvor erwähnt sollte das skizzierte Verfahren in einer Implementierung dahingehend modifiziert werden, daß die Werte $Gewinn(\bar{x})$, $Gewicht(\bar{x})$ sowie die unteren und oberen Schranken $lb(\bar{x})$ und $ub(\bar{x})$ als Parameter übergeben werden.

Folgende Folie zeigt ein Beispiel, in dem nur drei Knoten des Aufzählungsbaums generiert werden, um eine optimale Lösung zu bestimmen. Die untere und obere Schranke des linken Teilbaums der Wurzel stimmen überein und sind größer als die obere Schranke des rechten Teilbaums. Die Lösung $(x_1, x_2, x_3) = (0, 1, 1)$, unter der das Maximum angenommen wird, ergibt sich aus der berechneten Lösung $\gamma^+(0) = 8$ für das rationale Rest-Rucksackproblem für den linken Teilbaum.

Algorithmus 55 Verzweigen am Knoten $\langle x_1, \dots, x_{i-1} \rangle$

IF $Gewicht(x_1, \dots, x_{i-1}, 1) \leq K$ **THEN**

(* Nachfolger mit besserer oberen Schranke zuerst *)

IF $ub(x_1, \dots, x_{i-1}, 1) > ub(x_1, \dots, x_{i-1}, 0)$ **THEN**

Rucksack(i + 1, $\langle x_1, \dots, x_{i-1}, 1 \rangle$);

Rucksack(i + 1, $\langle x_1, \dots, x_{i-1}, 0 \rangle$)

ELSE

Rucksack(i + 1, $\langle x_1, \dots, x_{i-1}, 0 \rangle$);

Rucksack(i + 1, $\langle x_1, \dots, x_{i-1}, 1 \rangle$)

FI

ELSE

(*
 $Gewicht(x_1, \dots, x_{i-1}, 1) > K \geq Gewicht(x_1, \dots, x_{i-1}) = Gewicht(x_1, \dots, x_{i-1}, 0)$
*)

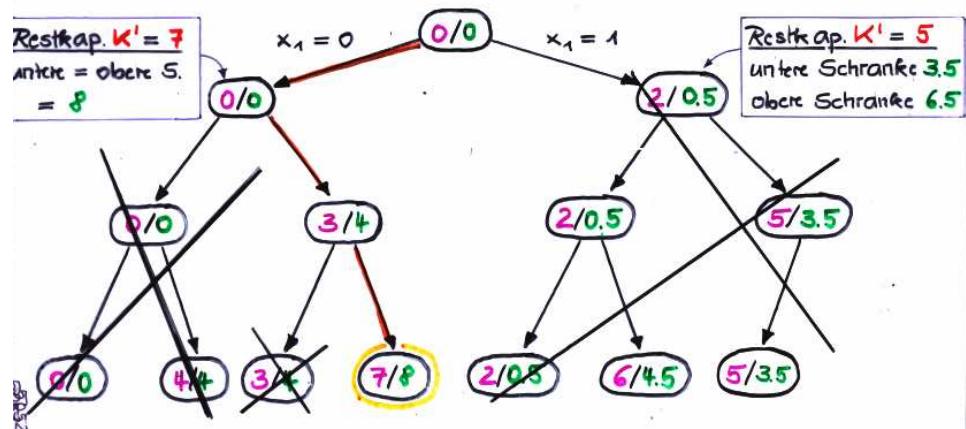
Rucksack(i + 1, $\langle x_1, \dots, x_{i-1}, 0 \rangle$); (* nur 0-Nachfolger betrachten *)

FI

Branch & Bound - Technik für das {0,1}-Rucksackproblem

i	1	2	3
P _j	1/2	4	4
w _j	2	3	4

Kapazität $K = 7$



Bemerkung 5.2.9 (Alternative Heuristiken für den Branch-Schritt). Es gibt für den Branchschritt diverse andere Möglichkeiten, die Schranken einzusetzen. Beispielsweise

kann man denjenigen der beiden Söhne mit der besseren unteren Schranke zuerst expandieren. Dies ist dann sinnvoll, wenn zu erwarten ist, dass die unteren Schranken näher an den Werten $opt(\bar{x})$ liegen als die oberen Schranken. \square

Die Least Cost Methode

Eine alternative Vorgehensweise wählt unter allen erzeugten, noch nicht expandierten Knoten denjenigen mit der besten oberen Schranke. Hierzu verwenden wir eine Prioritätswarteschlange, die sämtliche bereits generierten, noch zu expandierenden Knoten verwaltet. Als Ordnungskriterium wird auf die oberen Schranken $ub(\bar{x})$ zurückgegriffen, wobei wir voraussetzen, dass die oberen Schranken aller Blätter mit dem exakten Wert übereinstimmen.

Algorithmus 56 Branch & Bound Algorithmus für das Rucksackproblem (Least Cost Methode)

Initialisiere Q als Priority Queue (Maximumsheap), welche die Wurzel des Aufzählungsbaums als einziges Element enthält;

WHILE $Q \neq \emptyset$ **DO**

(* wähle einen Knoten \bar{x} aus Q , für den $ub(\bar{x})$ maximal ist, und entferne \bar{x} aus Q . *)

$\bar{x} := EXTRACT_MAX(Q);$

IF $Tiefe(\bar{x}) = n$ **THEN**

(* \bar{x} ist das erste gefundene Blatt und stellt somit eine optimale Rucksackfüllung dar *)

gib \bar{x} als Lösung aus und halte an

ELSE

(* expandiere v *)

berechne $ub(\bar{x}, 0)$ und füge $\langle \bar{x}, 0 \rangle$ in Q ein;

IF $Gewicht(\bar{x}, 1) \leq K$ **THEN**

berechne $ub(\bar{x}, 1)$ und füge $\langle \bar{x}, 1 \rangle$ in Q ein

FI

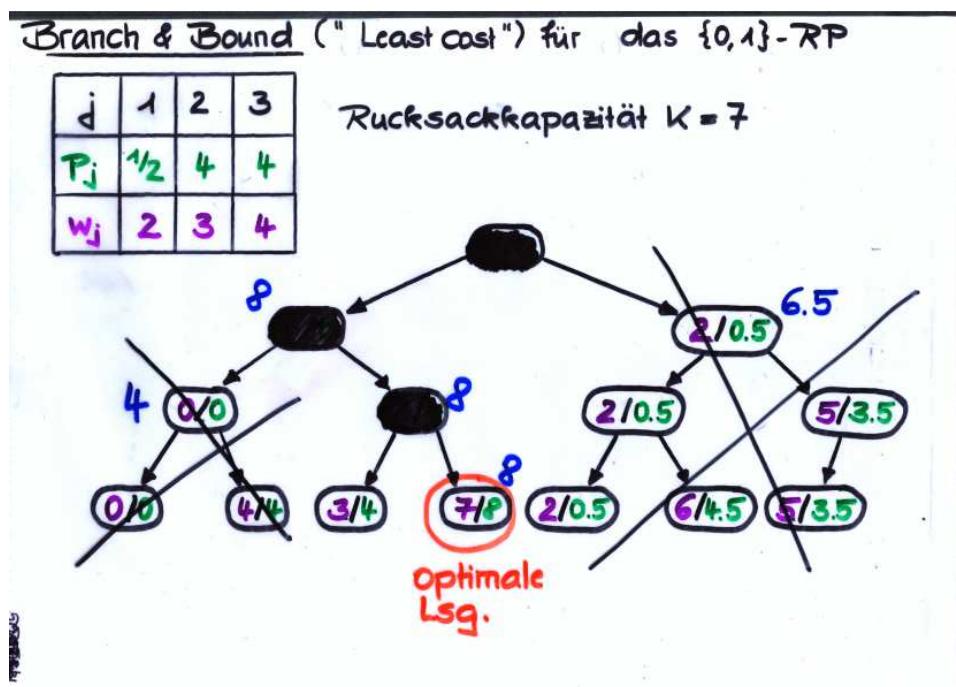
FI

OD

Wir skizzieren die prinzipielle Idee der LC-Methode am Beispiel des $\{0, 1\}$ -Rucksackproblems, wobei wir uns auf den Einsatz oberer Schranken beschränken. Hier ist $ub(\bar{x}) = Gewinn(\bar{x})$, falls \bar{x} ein Blatt ist, also wenn $Tiefe(\bar{x}) = n$ ist. Ist man lediglich an einer optimalen

Rucksackfüllung (oder nur an dem Gewinn einer optimalen Rucksackfüllung) interessiert, so kann man wie in Algorithmus 56 beschrieben vorgehen. Hier verwalten wir die generierten, aber noch nicht expandierten Knoten in einem Maximumsheap, dessen Ordnung durch die oberen Schranken $ub(\bar{x}) = \text{Gewinn}(\bar{x}) + \gamma^+(\bar{x})$ gegeben ist, die wie zuvor unter Einsatz des Algorithmus für das rationale Rucksackproblem berechnet werden können. Der nächste zu expandierende Knoten ist jeweils derjenige Knoten mit der besten oberen Schranke. Die Verwendung der Hilfsvariablen $optgewinn$ ist hier überflüssig, da das mit der LC-Methode zuerst besuchte Blatt stets eine optimale Lösung repräsentiert. Diese Aussage weisen wir in Satz 5.2.10 unten nach.

Auf folgender Folie ist ein Beispiel angegeben. Die mit Werten beschrifteten Knoten stellen den durchlaufenen Teil des Aufzählungsbaums dar. Die expandierten Knoten sind schwarz markiert.



Algorithmus 57 auf Seite 231 zeigt das allgemeine Schema der LC-Methode für Maximierungsprobleme unter Einsatz oberer Schranken. Vorauszusetzen ist lediglich, dass folgende Eigenschaften (1) und (2) erfüllt sind:

- (1) Für alle Knoten v im Aufzählungsbaum gilt $opt(v) \leq ub(v)$.
- (2) Für alle Blätter z im Aufzählungsbaum gilt $value(z) = ub(z)$.

Hier ist $value(\cdot)$ die zu maximierende Bewertung der Blätter. $opt(v)$ steht für den maximalen Wert eines Blatts im Teilbaum von x , also

$$opt(v) = \max\{value(z) : z \text{ ist ein Blatt im Teilbaum von } v\}.$$

Insbesondere gilt $opt(z) = value(z)$ für jedes Blatt z .

Algorithmus 57 Schema der Least Cost-Methode für Maximierungsprobleme

(* \mathcal{T} sei der Aufzählungsbaum für ein Maximierungsproblem *)

Initialisiere eine Priority Queue (Maximumsheap) Q , die den Wurzelknoten von \mathcal{T} als einziges Element enthält;

(* Ordnung der Elemente von Q ist durch die oberen Schranken $ub(\cdot)$ gegeben *)

WHILE $Q \neq \emptyset$ **DO**

$v := EXTRACT_MAX(Q);$

IF v ist ein Blatt **THEN**

(* v stellt eine optimale Lösung dar *)

gib $ub(v)$ aus und halte an

FI

(* Expandiere Knoten v , d.h. *)

(* generiere die Söhne von v *)

FOR ALL Söhne w von v in \mathcal{T} **DO**

berechne $ub(w)$ und füge w in Q ein

OD

OD

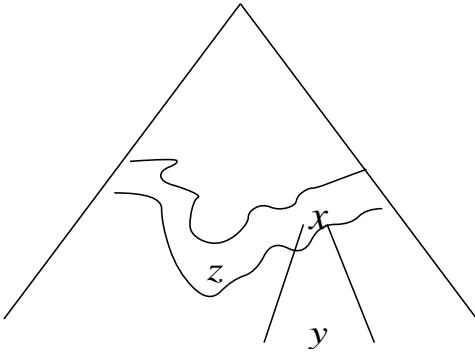
Es spielt hier keine Rolle, ob die Blätter dieselbe Tiefe haben. Sogar unendliche (aber endlich verzweigte) Aufzählungsbäume können mit der LC-Methode behandelt werden, jedoch kann für diese nur die partielle Korrektheit, nicht aber die Terminierung garantiert werden.

Für den Fall, daß die oberen Schranken $ub(x)$ exakt mit den Werten $opt(x)$ übereinstimmen und genau eine optimale Lösung existiert, wird bei der LC-Methode tatsächlich nur der Pfad von der Wurzel des Aufzählungsbaums zu einem optimalen Blatt durchlaufen. Alle anderen Äste des Aufzählungsbaums werden mit der Beschränkungsfunktion beschnitten. Daher nennen manche Autoren diese Durchlaufstrategie auch *Best-First-Search*.

Satz 5.2.10 (Partielle Korrektheit der LC-Methode). Unter den obigen Voraussetzungen (1) und (2) stellt das erste Blatt, welches die LC-Methode der Priority Queue entnimmt, eine optimale Lösung dar.

Beweis. Sei z das erste Blatt, welches die LC-Methode findet und y ein beliebiges anderes Blatt im Aufzählungsbaum. Zu zeigen ist, dass $value(y) \leq value(z)$.

Die BFS-ähnliche Vorgehensweise der LC-Methode, mit der jeder innere Knoten v , welcher der Priority Queue Q entnommen wird, durch seine Söhne ersetzt wird, stellt sicher, dass es zu jedem noch nicht besuchten Knoten w einen Knoten x in Q gibt, so dass sich w im Teilbaum von x befindet. Wir wenden diese Aussage auf das Blatt $w = y$ an.



Nach Definition von $opt(x)$ ist

$$opt(x) = \max\{value(y') : y' \text{ ist Blatt im Teilbaum von } x\} \geq value(y).$$

Ferner gilt $ub(z) \geq ub(x)$, da z durch *EXTRACT_MAX* aus Q entnommen wird. Hieraus folgt:

$$value(z) \stackrel{(2)}{=} ub(z) \geq ub(x) \stackrel{(1)}{\geq} opt(x) \geq value(y)$$

□

Analoges gilt für Minimierungsprobleme und unteren Schranken. Hier sind die Bedingungen (1) und (2) durch

- (1') Für alle Knoten v im Aufzählungsbaum gilt $opt(v) \geq lb(v)$.
- (2') Für alle Blätter z im Aufzählungsbaum gilt $value(z) = lb(z)$.

zu ersetzen. Die Priority Queue ist als Minimismheap hinsichtlich der unteren Schranken zu implementieren.

Das 15er Puzzle

Wir erläutern den Einsatz der LC-Methode für das 15er-Puzzle, das wir als Minimierungsproblem formulieren. Vgl. Beispiel 5.2.1 auf Seite 201. Gegeben ist eine Anfangskonfiguration x_0 der 15 Plättchen auf einem 4×4 -Spielfeld. Gesucht ist ein kürzester Weg (Folge von Spielzügen minimaler Länge) von x_0 zur sortierten Zielkonfiguration.

15er-Puzzle

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

Startkonfiguration x_0



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

Zielkonfiguration z

Minimierungsproblem: gesucht ist ein **kürzester Weg**
von der Startkonfiguration zur Zielkonfiguration.

Um die LC-Methode anwenden zu können, benötigen wir geeignete Werte $opt(\bar{x})$ und zugehörige (leicht zu berechnende) untere Schranken $lb(\bar{x})$ für jeden Knoten \bar{x} im Aufzählungsbaum. Sei $\bar{x} = \langle x_0, \dots, x_n \rangle$ ein Knoten im Aufzählungsbaum. Ist \bar{x} ein Blatt, d.h. stimmt x_n mit der Zielkonfiguration überein, so setzen wir $value(\bar{x}) = opt(\bar{x}) = Tiefe(\bar{x}) = n$. Dies entspricht der Bewertung der Blätter hinsichtlich der Anzahl an Spielzügen, die vorzunehmen sind, um das entsprechende Blatt von der Startkonfiguration zu erreichen. Wir nehmen nun an, dass \bar{x} ein innerer Knoten im Aufzählungsbaum ist, also dass x_n von der Zielkonfiguration verschieden ist. Der Wert $opt(\bar{x})$ einer besten Lösung im Teilbaum mit Wurzel \bar{x} ist die Länge eines kürzesten Pfads von x_0 zur Zielkonfiguration, bei dem die ersten Spielzüge durch die Konfigurationsfolge x_0, \dots, x_n gegeben sind. Offenbar gilt:

$$opt(\bar{x}) = Tiefe(\bar{x}) + \Delta(x_n) = n + \Delta(x_n),$$

wobei $\Delta(x)$ die minimale Anzahl an Spielzügen ist, die benötigt werden, um Konfiguration x in die Zielkonfiguration überzuführen. Da das 15er-Puzzle nicht für alle Anfangskonfigurationen lösbar ist, kann $\Delta(x)$ gleich ∞ sein. Mehr dazu später.

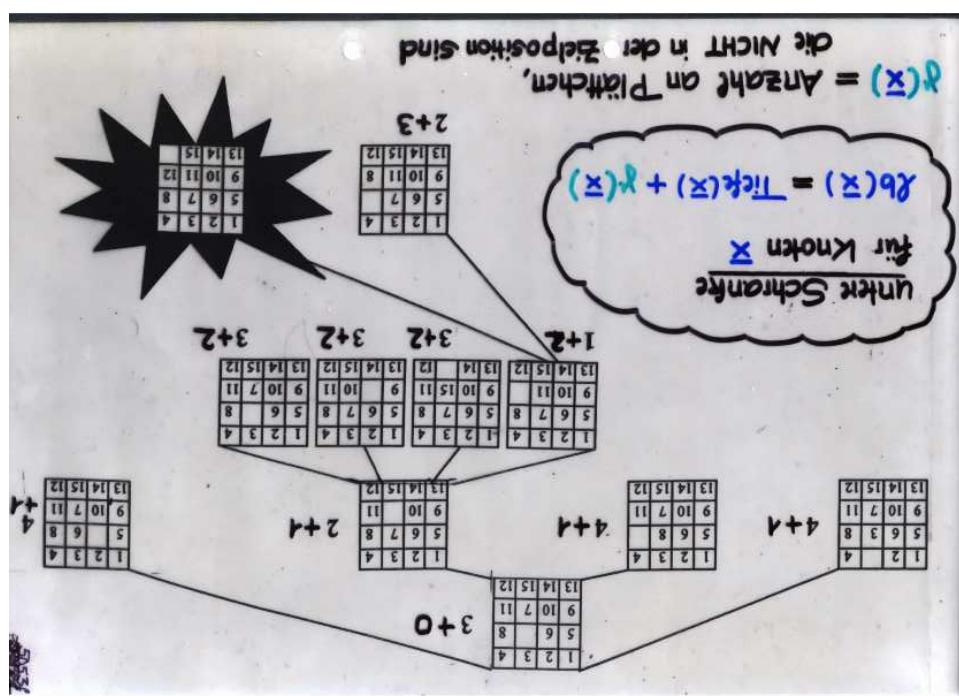
Es gibt viele Möglichkeiten, geeignete untere Schranken mit den oben genannten Eigenschaften (1') und (2') zu definieren. Wir stellen hier zwei Varianten vor, welche beide untere Schranken für die Werte $\Delta(\cdot)$ verwenden, also von der Form

$$lb(\bar{x}) = Tiefe(\bar{x}) + \gamma(x_n)$$

sind, wobei $\gamma(x) \leq \Delta(x)$. Eine Möglichkeit besteht darin, $\gamma(x)$ als die Anzahl der Plättchen in einer Konfiguration x zu definieren, welche nicht in ihrer Zielposition plaziert sind. Offenbar ist $\gamma(x)$ eine untere Schranke für die Anzahl $\Delta(x)$ der notwendigen

Schritte, um von der aktuellen Konfiguration x zur Zielkonfiguration zu gelangen. Z.B. gilt $\gamma(x_0) = 3$ für die auf der Folie angegebene Startkonfiguration x_0 , da hier genau die Plättchen mit der Aufschrift 7, 11 und 12 nicht auf ihren Zielpunkten sind. Eine bessere untere Schranke erhält man durch die Summe aller Entfernungen der einzelnen Plättchen von ihren Zielpositionen. Für obige Konfiguration x_0 ist dies ebenfalls der Wert $1 + 1 + 1 = 3$, da alle drei Plättchen 7, 11 und 12 nur ein Feld von ihrer Zielposition entfernt sind.

Wendet man die LC-Methode an, d.h. expandiert man jeweils denjenigen Knoten des bereits erzeugten Teils des Aufzählungsbaums, dessen Söhne noch nicht generiert sind und für den die untere Schranke $lb(\bar{x})$ minimal ist, dann ist sichergestellt, daß im Falle der Terminierung das gefundene Blatt eine optimale Lösung darstellt (Satz 5.2.10 für Minimierungsprobleme). Die folgende Folie zeigt den mit der LC-Methode generierten Teil des Aufzählungsbaums für die Anfangskonfiguration x_0 wie auf der Folie oben.



Die LC-Methode terminiert jedoch nur dann, wenn die Anfangskonfiguration tatsächlich in die Zielkonfiguration überführt werden kann. Der dem 15er Puzzle zugrundeliegende ungerichtete Graph hat zwei gleichgroße Zusammenhangskomponenten. Die Fragestellung des 15er Puzzles ist also nur für die Hälfte aller möglichen Anfangskonfigurationen lösbar. Es gibt jedoch ein einfaches Kriterium, mit dem man prüfen kann, ob die Zielkonfiguration von der Anfangskonfiguration erreichbar ist.

Wir numerieren die 16 Felder zeilenweise von 1 bis 16 durch. Z.B. das Feld in Zeile 1 und Spalte 2 hat die Nummer 2, das Feld in Zeile 4 und Spalte 3 hat die Nummer 15. Sei x eine beliebige Konfiguration der 15 Plättchen. Wir schreiben $Pos(i, x) = j$, wenn in der Konfiguration x das Plättchen mit der Aufschrift i auf dem Feld mit Nummer j liegt. Die Schreibweise $Pos(16, x) = j$ bedeutet, daß in der Konfiguration x das Feld mit Nummer

j leer ist. Intuitiv legen wir also auf das leere Feld ein Plättchen mit der Aufschrift 16. Jeder Spielzug entspricht dem Vertauschen des Plättchens mit der Aufschrift 16 mit einem auf einem benachbarten Feld liegenden Plättchen.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Numerierung der Felder
im 15er Puzzle

Ein *Fehlstand* in x ist ein Paar (i, j) mit $i, j \in \{1, 2, \dots, 16\}$, so daß $i < j$ und $Pos(i, x) > Pos(j, x)$. $F(x)$ bezeichne die Anzahl an Fehlständen in x . Weiter sei $sign(x) \in \{0, 1\}$ folgender Wert. $sign(x)$ ist 1, falls $Pos(16, x) \in \{2, 4, 5, 7, 10, 12, 13, 15\}$. Andernfalls ist $sign(x) = 0$.

	2		4
5		7	
	10		12
13		15	

$sign(x) = 1$

Wir zitieren folgendes Resultat ohne Beweis:

Satz 5.2.11 (Lösbarkeit des 15er Puzzles). Die Zielkonfiguration ist genau dann von der Anfangskonfiguration x erreichbar, wenn der Wert $F(x) + sign(x)$ gerade ist. (ohne Beweis)

Nun zur Terminierung der LC-Methode für eine Startkonfiguration, von welcher die Zielkonfiguration z erreichbar ist, also $F(x_0) + sign(x_0)$ gerade. Sei $\Delta(x_0) = k$, d.h. die Länge eines kürzesten Pfads von x_0 zu z ist k . Weiter sei $\bar{y} = \langle x_0, \dots, z \rangle$ ein Blatt im Aufzählungsbaum der Tiefe k . Für alle Knoten \bar{x} im Aufzählungsbaum der Form $\bar{x} = \langle x_0, \dots, x_n \rangle$ mit $n > k$ gilt

$$lb(\bar{x}) \geq Tiefe(\bar{x}) = n > k = lb(\bar{y}).$$

Keiner dieser Knoten kann daher dem Minimumsheap Q entnommen werden, bevor \bar{y} oder ein anderes Blatt der Tiefe k gefunden wird. Man beachte, dass Q für alle noch nicht besuchten Knoten v einen Knoten w enthält, so dass sich v im Teilbaum von w befindet. Solange also $v = \bar{y}$ noch nicht besucht wurde, gibt es einen Knoten w in Q , so dass \bar{y} im Teilbaum von w liegt. Weiter ist $lb(w) \leq opt(w) = k$.

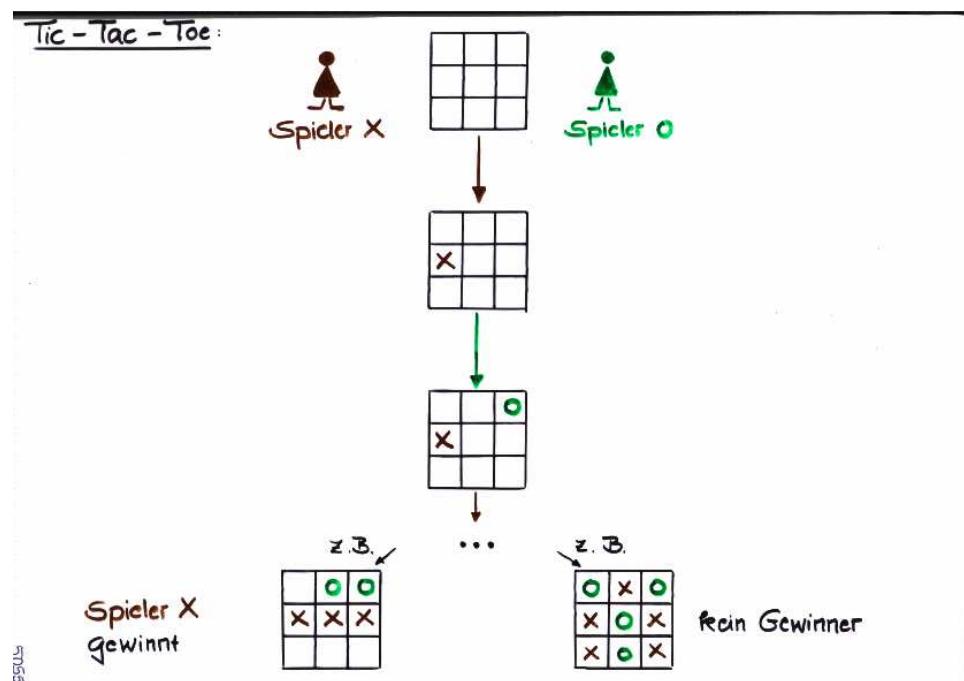
5.3 Spielbäume

Ausgangspunkt ist ein Zwei-Personen-Strategiespiel ohne Zufallskomponente, z.B. Brettspiele wie Dame, Mühle, Schach, Reversi, Go, etc., bei dem zwei Spieler „Spieler X“ (Computer) und „Spieler O“ (Gegner) abwechselnd am Zug sind.

Unter einer *Konfiguration* versteht man den aktuellen Spielstand (z.B. Aufstellung der Spielfiguren, Angabe des Spielers, der den nächsten Zug machen muß, etc.). Das Spiel ist gegeben durch die Festlegung

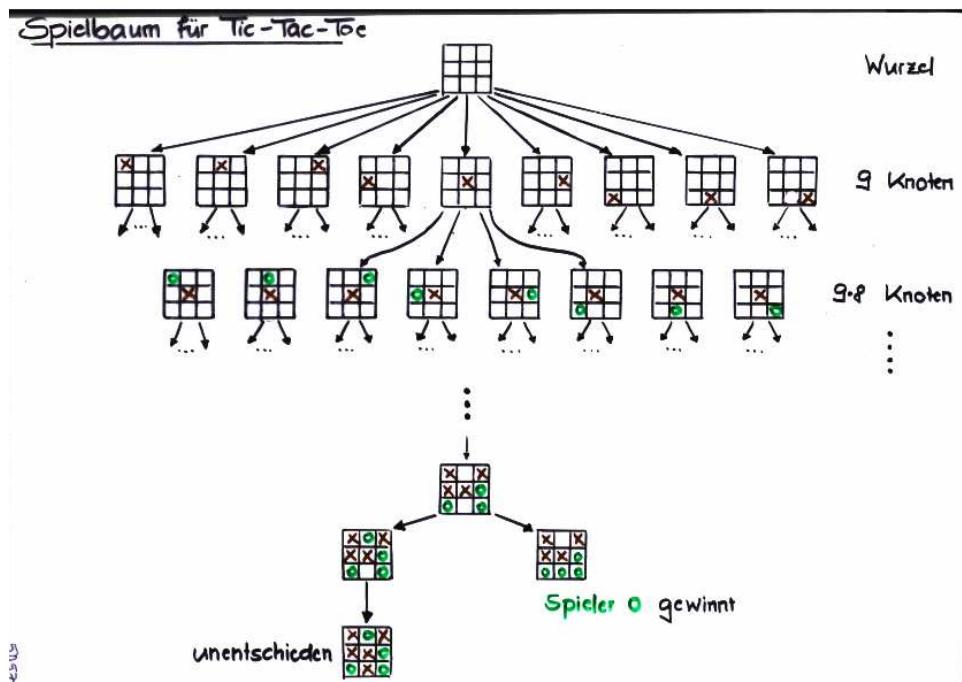
- einer Anfangskonfiguration,
- der legalen Spielzügen,
- der Endkonfigurationen (Festlegung, wann das Spiel beendet ist)
- und die Angabe einer Bewertung der Endkonfigurationen, die angibt, wer gewonnen hat und/oder wieviel Sieg- bzw. Verlustpunkte erzielt wurden.

Sei v eine Konfiguration. Wir nennen w eine *Folgekonfiguration* von v , wenn es einen legalen Spielzug gibt, der v nach w überführt. Eine formale Darstellung kann man durch die Angabe eines gerichteten Graphen mit einem als Wurzel ausgezeichneten Knoten erhalten. Die Knotenmenge ist die Menge der Konfigurationen. Die Wurzel ist die Anfangskonfiguration, die Endkonfigurationen sind Terminalknoten. Die Kanten entsprechen den legalen Spielzügen. Für Spiele wie Schach oder Mühle liegt ein zyklischer Graph vor. Für andere Spiele, etwa Tic Tac Toe, liegt ein azyklischer Graph vor. Als Beispiel betrachten wir einen Ausschnitt des Tic Tac Toe Spielgraphen.



Diese formale Sicht eines Spiels als Graph dient lediglich der Anschauung. Der Spielgraph ist in den meisten Fällen extrem groß und kann (aus Zeit- und Platzgründen) nicht explizit dargestellt werden.

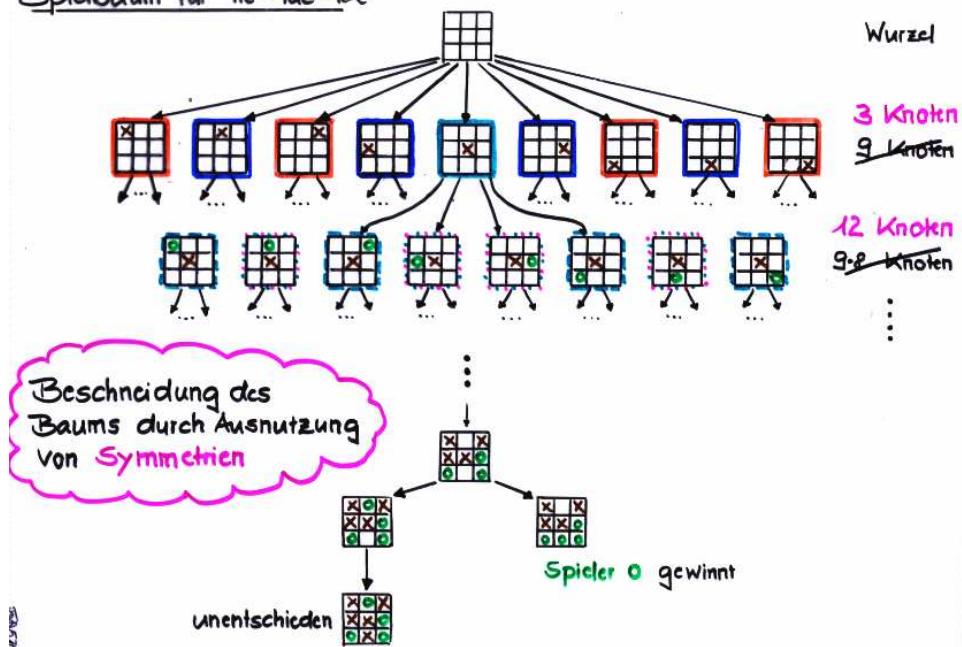
Gedanklich arbeiten wir mit dem durch Abwickeln des Spielgraphen entstehenden Spielbaum. Ein innerer Knoten v im Spielbaum heißt *Maxknoten*, falls er eine Konfiguration repräsentiert, in der Spieler X am Zug ist. Andernfalls heißt v *Minknoten*. Die Bezeichnungen Max- bzw. Minknoten suggerieren, daß Maxknoten für solche Knoten stehen, an denen der Gewinn von Spieler X zu maximieren ist; während die Minknoten für Spielzüge des Gegners (Spieler O) stehen, der versuchen wird, den Gewinn von Spieler X zu minimieren.



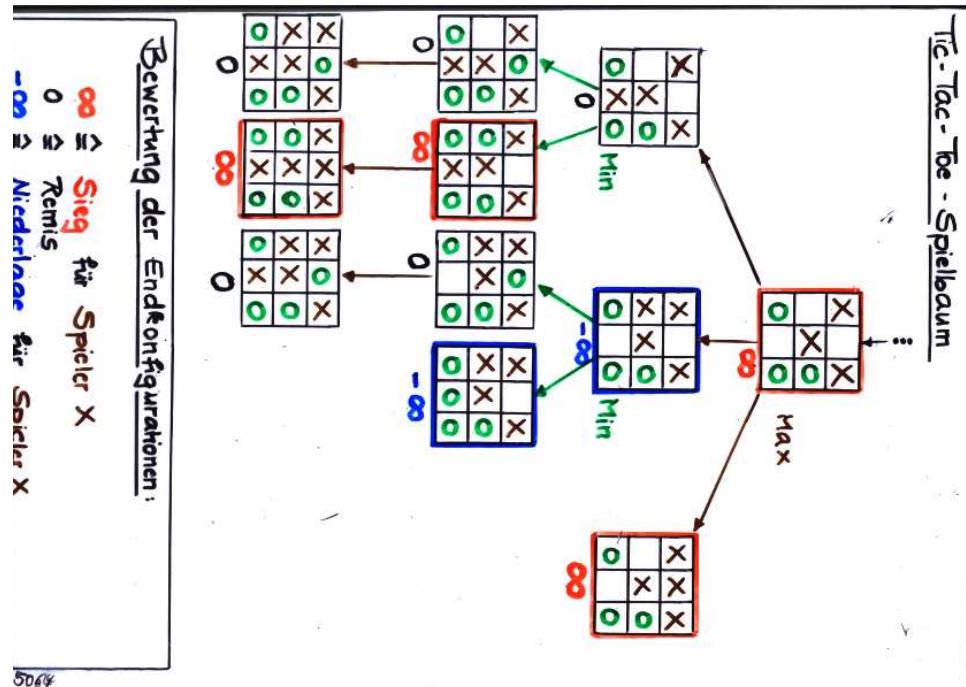
Zuggenerator. Gegeben ist eine Spielkonfiguration v_0 , in der Spieler X (Computer) am Zug ist. Also ist v_0 ein Maxknoten. Gesucht ist eine optimale Spielstrategie für Spieler X, die garantiert, daß Spieler X einen möglichst hohen Gewinn erzielt, unabhängig davon, wie sich Spieler O im Folgenden verhält. Falls es keine solche Gewinnstrategie gibt, ist eine Strategie gesucht, die die Verlustpunkte für Spieler X minimiert. Dabei wird stets angenommen, daß der Gegenspieler in der Lage ist, optimal zu spielen. Annahmen wie „der Gegenspieler übersieht (hoffentlich) einen gewinnbringenden Zug“ sollen nicht gemacht werden.

Die prinzipielle Lösungsmöglichkeit besteht in einer Graphanalyse. Für einfache Spiele ist eine Analyse des vollständigen Graphen in zumutbarer Echtzeit möglich, wenn zusätzliche Tricks angewandt werden. Z.B. liegt für Tic Tac Toe ein azyklischer Graph vor, den wir uns als Baum mit rund $9! = 362\,880$ Knoten vorstellen können. Durch Ausnutzung von Symmetrien ist dieser recht effizient zu analysieren.

Spielbaum für Tic-Tac-Toe



In solchen Fällen, in denen der Spielgraph oder -baum hinreichend klein ist, hilft eine Rückwärtssuche, die in den Endkonfigurationen beginnt und in Bottom-Up-Manier die Spielergebnisse der Endkonfigurationen auf die inneren Knoten überträgt. Jeder Maxknoten wird mit dem besten Spielergebnis seiner Söhne, ein Minknoten mit dem schlechtesten Spielergebnis seiner Söhne beschriftet. Eine Spielstrategie ergibt sich wie folgt. Wir wählen denjenigen Spielzug $v_0 \rightarrow w$, so daß die Bewertung von w maximal ist unter allen Söhnen von v_0 .



Die beschriebene Technik lässt sich für Spiele, für die der zugrundeliegende Graph zyklisch ist, modifizieren. Jedoch ergibt sich nur dann eine in der Praxis anwendbare Methode, wenn der Spielgraph hinreichend klein ist. Für komplexe Spiele ist eine vollständige Analyse des Spielgraphen zum Scheitern verurteilt, da der Spielgraph viel zu groß ist. Z.B. hat man es beim Schach mit einem zyklischen Graphen mit ca. 10^{120} Knoten zu tun. Die Analyse des Graphen würde ca. 10^{21} Jahrhunderte dauern, wenn die Zugmöglichkeiten jeder Konfiguration in 1/3 Nanosekunde berechnet werden können.

Anstelle den kompletten Graphen zu analysieren, legt man eine *Suchtiefe* d fest. Die Suchtiefe steht für die maximale Anzahl an aufeinanderfolgenden Spielzügen, welche der Zuggenerator berücksichtigt. Diese wird meist anhand der vom Benutzer gewählten Spielstärke festgelegt und ist in Abhängigkeit von Echt-Zeit-Bedingungen („welchen Zeitbeschränkungen unterliegt der Zuggenerator?“) zu wählen. Die Suchtiefe kann während des Spiels geändert werden. Z.B. hat man es in der Endphase eines Schachspiels mit einem wesentlich geringeren durchschnittlichen Verweigungsgrad zu tun als in der Mittelphase und kann daher in der Endphase die Suchtiefe erhöhen.

Der Spielbaum zu vorgegebener Suchtiefe. Konzeptionell ähneln Spielbäume den in Abschnitt 5.3 behandelten Aufzählungsbäumen. Für feste Suchtiefe d ergibt sich durch „Abwickeln“ des Spielgraphen ein Baum der Höhe (höchstens) d , dessen Wurzel für die aktuelle Spielkonfiguration v_0 steht.³⁹ Diesen nennen wir den Spielbaum bzgl. der

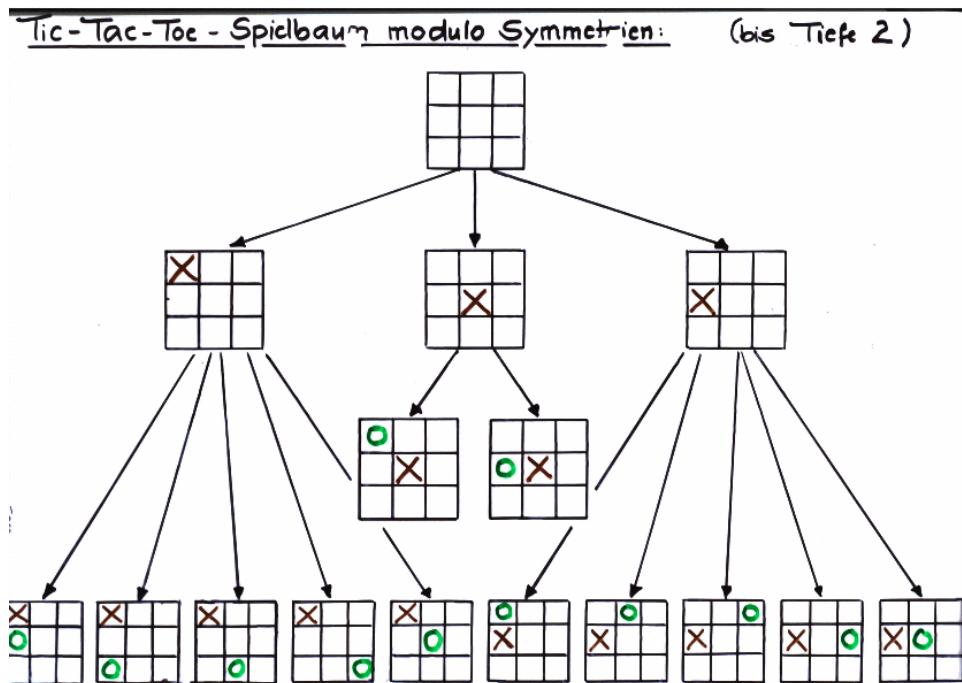
³⁹In Spielen wie Tic Tac Toe, in denen der Spielgraph ein zyklischer Graph mit maximaler Weglänge 9 ist, kann der Spielbaum die Höhe $< d$ haben. Ein Beispiel: Wir nehmen an, daß die ersten 4 Spielzüge bereits durchgeführt sind und suchen nun einen „besten Zug“ für Spieler X. Ist die vorgegebene Suchtiefe $d = 7$, dann entsteht ein Spielbaum der Höhe 5.

Suchtiefe d , wobei wir im folgenden auf den Zusatz „bzgl. der Suchtiefe d “ verzichten und kurz vom Spielbaum sprechen.

- Die Knoten sind Tupel $\langle v_0, v_1, \dots, v_n \rangle$ der Länge $n \leq d$, für welche die Konfigurationsfolge v_0, v_1, \dots, v_n aus legalen Spielzügen entsteht.
- Die Kante von $\langle v_0, v_1, \dots, v_{n-1} \rangle$ zu $\langle v_0, v_1, \dots, v_{n-1}, v_n \rangle$ steht für den Spielzug, der die Konfiguration v_{n-1} in v_n überführt.
- Die Blätter sind Knoten $\langle v_0, v_1, \dots, v_n \rangle$, so daß entweder $n = d$ oder v_n eine Endkonfiguration ist.

Gedanklich (und in den Skizzen) identifiziert man jeden Knoten $\langle v_0, v_1, \dots, v_n \rangle$ des Spielbaums mit der letzten Konfiguration v_n . Man beachte jedoch, daß ein Knoten des Spielgraphen (also eine Spielkonfiguration) durch mehrere Knoten im Spielbaum repräsentiert sein kann. Nämlich dann, wenn es mehrere Zugfolgen gibt, die von der Anfangskonfiguration zu der betreffenden Konfiguration führen.

Folgende Skizze zeigt den Spielbaum für Tic-Tac-Toe mit der Suchtiefe 2, wobei zur Vereinfachung symmetrische Konfiguration zusammengefasst sind:



Die Payoff-Funktion. Jedem Knoten im Spielbaum wird eine ganze Zahl zugewiesen, welche die Güte der Konfiguration aus Sicht von Spieler X mißt. Dazu werden zunächst die Blätter des Spielbaums bewertet (mit Hilfe einer Payoff-Funktion); die Werte werden in Bottom-Up-Manier „von unten nach oben“ durch den Baum propagiert, wie zuvor am Beispiel von Tic-Tac-Toe erläutert. Der Spielbaum wird zwar nicht explizit erstellt, jedoch wird der vermeintlich beste Zug mit einem Verfahren ermittelt, das auf einem

Postorder-Durchlauf des Spielbaums beruht. Die verschiedenen Repräsentationen (Knoten im Spielbaum) einer Konfiguration (Knoten im Spielgraph) können mit unterschiedlichen Bewertungen versehen sein, da die verbleibende Suchtiefe im Spielbaum, also die Höhe der betreffenden Teilbäume des Spielbaums, unterschiedlich sein kann.

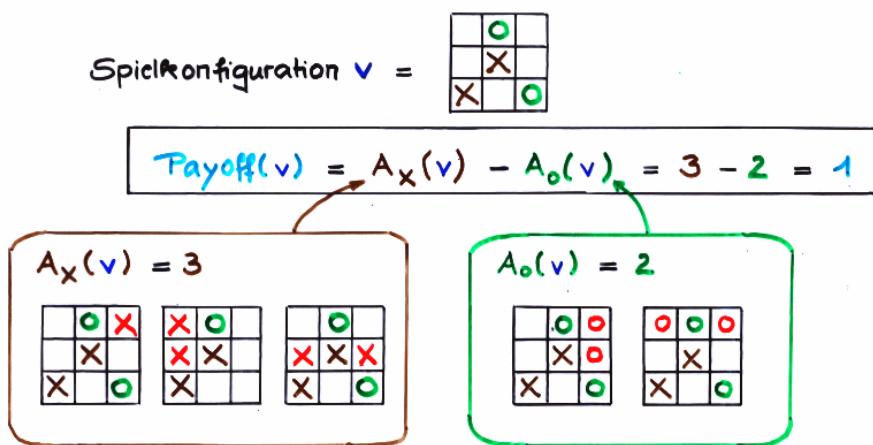
Wir setzen im Folgenden voraus, daß eine Funktion vorliegt, die die Konfigurationen hinsichtlich der Gewinnchancen von Spieler X bewertet.

- Die Endkonfigurationen können gemäß dem Spielausgang bewertet werden. Z.B. $+\infty$ für den Sieg von Spieler X, $-\infty$ für den Sieg von Spieler O und 0 für ein Remis.
- Für Blätter der Tiefe d , für welche die zugehörige Konfiguration keine Endkonfiguration ist, wird die Bewertung anhand einer Heuristik vorgenommen.

Im Folgenden sei $\text{Payoff}(v)$ der Wert, welchen die Payoff-Funktion der Konfiguration v zuordnet.

Für Brettspiele wie Schach orientiert sich die Payoff-Funktion im Wesentlichen an dem Materialwert und der Plazierung der Spielfiguren. Wir erläutern hier kurz eine (sinnvolle) Payoff-Funktion für Tic-Tac-Toe. Sei v eine Spielkonfiguration. Wir definieren $A_X(v)$ als die Anzahl an Zeilen, Spalten und Diagonalen in der Konfiguration v , die zu vollständigen „X-Linien“ ergänzt werden können. Entsprechend definieren wir $A_O(v)$ und setzen $\text{Payoff}(v) = A_X(v) - A_O(v)$.

Beispiel zur Payoff - Funktion für Tic-Tac-Toe:



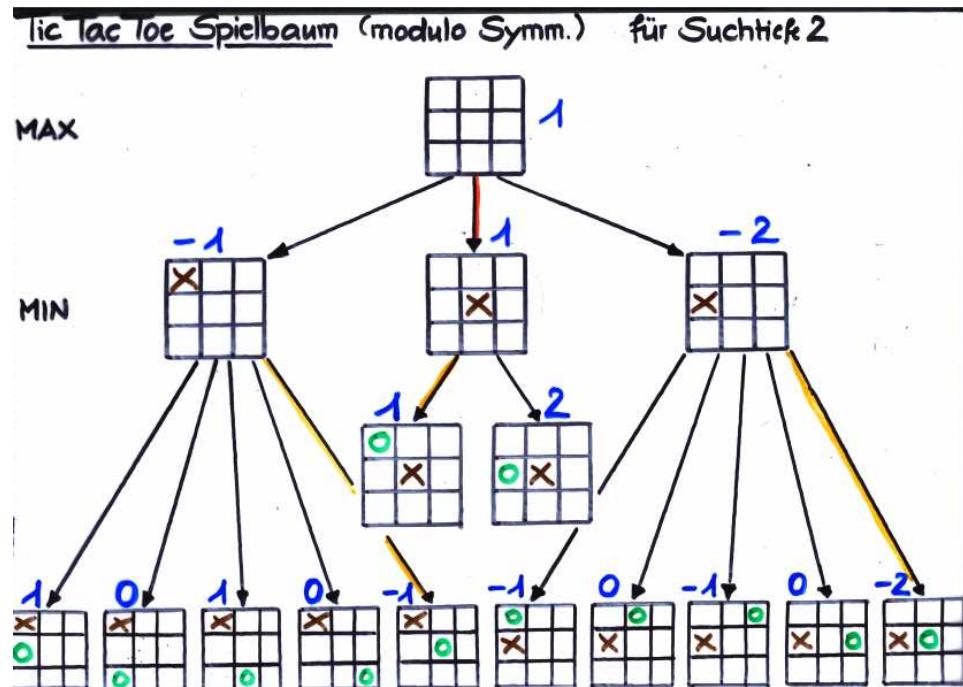
5

5.3.1 Das Minimax-Verfahren

Mit der Minimax-Technik wird für jeden Knoten des Spielbaums die Bewertung festgelegt. Die Blätter des Spielbaums werden mit Hilfe der Payoff-Funktion bewertet. Für jeden

inneren Knoten $\langle v_0, v_1, \dots, v_{n-1}, v_n \rangle$, für den n gerade ist (d.h. v_n ist ein Maxknoten und stellt eine Konfiguration dar, in der Spieler X am Zug ist) nehmen wir die maximale Bewertung der Söhne von v . Ist n ungerade (d.h. v_n ist ein Minknoten), so wird dem Knoten $\langle v_0, v_1, \dots, v_n \rangle$ die minimale Bewertung seiner Söhne zugeordnet. Einen optimalen Zug für die aktuelle Konfiguration v_0 erhält man nun durch den Konfigurationswechsel $v_0 \rightarrow v_1$, für den die Bewertungen des Knotens $\langle v_0, v_1 \rangle$ und der Wurzel $\langle v_0 \rangle$ übereinstimmen.

Als Beispiel betrachten wir Tic Tac Toe mit der Suchtiefe $d = 2$. Folgende Skizze zeigt den Spielbaum und die Bewertungen der Knoten, die sich mit der oben genannten Payoff-Funktion und dem Minimax-Verfahren ergeben.



Wir realisieren die Minimax-Bewertung mit einem Preorder-Durchlauf zur Generierung der Knoten und bewerten die Knoten nach dem Postorder-Prinzip. Die Parameter des in Algorithmus 58 auf Seite 243 skizzierten Verfahrens sind eine Konfiguration v sowie ein Wert $\ell \in \{0, 1, \dots, d\}$. (Zur Erinnerung: d ist die Suchtiefe.) Intuitiv steht die Zahl ℓ für die verbleibende Suchtiefe. Durch den Aufruf von $\text{Minimax}(v, \ell)$ wird die Bewertung eines Knotens $\langle v_0, v_1, \dots, v_{d-\ell} \rangle$ mit $v_{d-\ell} = v$ berechnet.⁴⁰ Dies ist der wie folgt definierte Wert $c(v, \ell)$. Ist v eine Endkonfiguration oder $\ell = 0$, dann ist $c(v, \ell) = \text{Payoff}(v)$. Ist v keine Endkonfiguration und $\ell \geq 1$:

- Ist v ein Maxknoten (Spieler X am Zug), dann ist

$$c(v, \ell) = \max \{c(w, \ell - 1) : w \text{ ist Folgekonfiguration von } v\}.$$

⁴⁰Man beachte, dass die Minimax-Bewertung aller Knoten $\langle v_0, v_1, \dots, v_n \rangle$ nur von der letzten Konfiguration $v = v_n$ und der verbleibenden Suchtiefe, also dem Wert $\ell = d - n$, abhängt. Daher genügt eine Parametrisierung mit dem Paar (v, ℓ) . Die Zugfolge $v_0 \rightarrow v_1, v_1 \rightarrow v_2, \dots, v_{n-1} \rightarrow v_n = v$, die zur Konfiguration v führte, ist für die Bewertung irrelevant.

- Ist v ein Minknoten (Spieler O am Zug), dann ist

$$c(v, \ell) = \min \{c(w, \ell - 1) : w \text{ ist Folgekonfiguration von } v\}.$$

Algorithmus 58 *Minimax(v, ℓ)*

IF v ist ein Blatt oder $\ell = 0$ **THEN**

return *Payoff*(v)

ELSE

IF v ist ein Maxknoten **THEN**

$max := -\infty$

FOR ALL Söhne w von v **DO**

$max := \max\{max, Minimax(w, \ell - 1)\}$

OD

return max

ELSE

$min := +\infty$

FOR ALL Söhne w von v **DO**

$min := \min\{min, Minimax(w, \ell - 1)\}$

OD

return min

FI

FI

Die Bewertung der aktuellen Spielkonfiguration v_0 erhält man durch den Aufruf von $Minimax(v_0, d)$, welcher den Wert $c(v_0, d)$ zurückgibt. Die Laufzeit des Minimax-Verfahrens ist offenbar linear in der Größe des Spielbaums und somit durch $\mathcal{O}(b^d)$ beschränkt, wobei b der maximale Verzweigungsgrad der Knoten im Spielbaum ist und wobei die Kosten zur Berechnung der Payoff-Funktionswerte als konstant angenommen werden.

5.3.2 α - β -Pruning

α - β -Pruning ist eine Beschränkungstechnik, welche die Laufzeit des Minimax-Verfahrens verbessern soll. Die Grundidee ist die Verwendung von vorläufigen Bewertungen der Max- und Minknoten:

- untere Schranken $\alpha(v, \ell) \leq c(v, \ell)$ für jeden Maxknoten v ,
- obere Schranken $\beta(w, \ell) \geq c(w, \ell)$ für jeden Minknoten w .

Ist $\ell = 0$ oder v bzw. w eine Endkonfiguration, dann werden die α - bzw. β -Werte gemäß der Payoff-Funktion bestimmt. Intuitiv kann man sich eine Initialisierung $\alpha(v, \ell) = -\infty$, $\beta(w, \ell) = +\infty$ denken. Der aktuelle α - bzw. β -Wert ist

$$\begin{aligned}\alpha(v, \ell) &= \text{höchste Bewertung für } \langle v, \ell \rangle, \text{ die bisher gefunden wurde,} \\ \beta(w, \ell) &= \text{kleinste Bewertung für } \langle w, \ell \rangle, \text{ die bisher gefunden wurde.}\end{aligned}$$

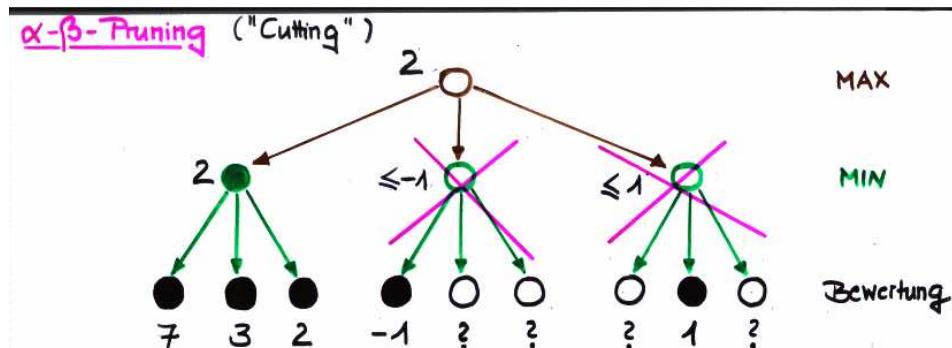
Diese Schranken ändern sich dynamisch während der Ausführung des Minimax-Verfahrens und werden wie folgt bei der Berechnung von $c(v, \ell)$ eingesetzt:

- α -Cut: Ist v ein Maxknoten, w eine Folgekonfiguration von v mit $\beta(w, \ell - 1) \leq \alpha(v, \ell)$, dann kann der Teilbaum von w ignoriert werden.
- β -Cut: Ist w ein Miniknoten, v eine Folgekonfiguration von w mit $\alpha(v, \ell - 1) \geq \beta(w, \ell)$, dann kann der Teilbaum von v ignoriert werden.

Der α -Cut kann wie folgt erläutert werden. Liegt eine Folgekonfiguration w von v vor mit $\beta(w, \ell - 1) \leq \alpha(v, \ell)$, dann ist

$$c(w, \ell - 1) \leq \beta(w, \ell - 1) \leq \alpha(v, \ell) \leq c(v, \ell).$$

D.h. der Spielzug $v \rightarrow w$ führt zu einer Bewertung, die höchstens so gut wie die der bereits gefundenen besten Zugmöglichkeit ist. Der Knoten w und dessen Teilbaum kann daher vernachlässigt werden. Eine entsprechende Argumentation rechtfertigt den β -Cut. Folgende Folie zeigt ein Beispiel für das Cutting. Wir nehmen an, dass die Söhne der Wurzel in der Reihenfolge „von links nach rechts“ generiert werden. Nach der Analyse des linken Teilbaums ist der α -Wert der Wurzel gleich 2. Wird nun der erste (linkste) Sohn des mittleren Teilbaums der Wurzel generiert und liefert die Payoff-Funktion die Bewertung -1 , so erhält der mittlere Sohn der Wurzel den β -Wert -1 . Da dieser schlechter ist als der α -Wert der Wurzel wird die Analyse des mittleren Teilbaums abgebrochen. Analoges gilt für den rechten Teilbaum, für den wir annehmen, dass zuerst der mittlere (mit 1 bewertete) Sohn erzeugt wird.



α -Werte für Max-Knoten: untere Schranken für die Bewertung

β -Werte für Min-Knoten: obere Schranken für die Bewertung

α -Cut: Ein Sohn w eines Max-Knotens v kann gestrichen werden, falls $\beta(w) \leq \alpha(v)$.

Der folgende Satz zeigt die Korrektheit des α - β -Prunings, das wir durch zwei sich gegenseitig aufrufende Algorithmen $MaxValue(v, \ell, \beta)$ und $MinValue(w, \ell, \alpha)$ formuliert haben. Siehe Algorithmus 59 auf Seite 245. Im Aufruf $MaxValue(v, \ell, \beta)$ steht der Wert β für den aktuellen β -Wert $\beta(w, \ell + 1)$ des „Vaters“ von w . Entsprechende Bedeutung hat der Wert α bei Aufruf von $MinValue(w, \ell, \alpha)$.

Algorithmus 59 α - β -Pruning

$MaxValue(v, \ell, \beta)$

IF v ist ein Blatt oder $\ell = 0$ **THEN**
 return $Payoff(v)$
ELSE
 $\alpha := -\infty;$ (* $\alpha = \alpha(v, \ell)$ *)
FOR ALL Söhne w von v **DO**
 $\alpha := \max\{\alpha, MinValue(w, \ell - 1, \alpha)\};$
IF $\alpha \geq \beta$ **THEN**
 return β (* Cutting *)
FI
OD
FI
return α

$MinValue(w, \ell, \alpha)$

IF w ist ein Blatt oder $\ell = 0$ **THEN**
 return $Payoff(w)$
ELSE
 $\beta := +\infty;$ (* $\beta = \beta(w, \ell)$ *)
FOR ALL Söhne v von w **DO**
 $\beta := \min\{\beta, MaxValue(v, \ell - 1, \beta)\};$
IF $\alpha \geq \beta$ **THEN**
 return α (* Cutting *)
FI
OD
FI
return β

In dem Algorithmus nicht eingebaut ist der Fall, dass eine Gewinnstrategie gefunden wurde. Ist z.B. in $MaxValue(v, \ell, \beta)$ der aktuelle Wert für α gleich ∞ , so wurde ein Gewinn bringender Spielzug gefunden und die Betrachtung weiterer Folgekonfigurationen von v ist überflüssig.

Satz 5.3.1 (Korrektheit von Algorithmus 59). Die Bewertung $c(v_0, d)$ eines Maxknotens v_0 bzgl. der Suchtiefe d wird durch den Funktionsaufruf $MaxValue(v_0, d, +\infty)$ berechnet.

Beweis. Der Beweis von Satz 5.3.1 beruht auf folgenden beiden allgemeineren Aussagen:

- (i) Ist v eine Endkonfiguration oder $\ell = 0$, so gilt für alle α und β :

$$\text{MaxValue}(v, \ell, \beta) = \text{MinValue}(v, \ell, \alpha) = \text{Payoff}(v).$$

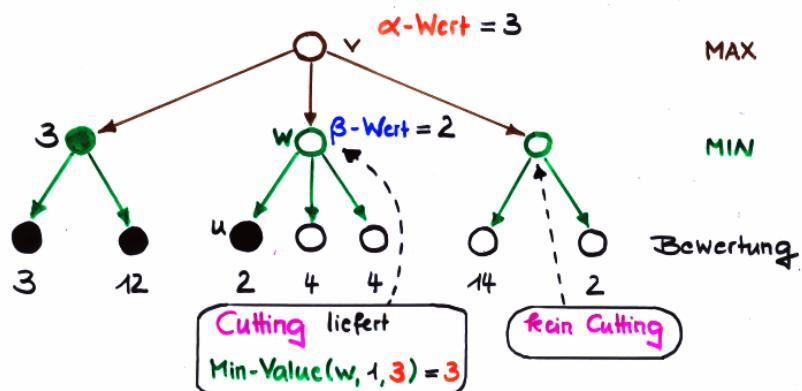
- (ii) Seien v, w zwei Konfigurationen, die keine Endkonfigurationen sind und $\ell \geq 1$. Dann gilt:

- Ist v ein Maxknoten, so ist $\text{MaxValue}(v, \ell, \beta) = \min\{c(v, \ell), \beta\}$.
- Ist w ein Minknoten, so ist $\text{MinValue}(w, \ell, \alpha) = \max\{c(w, \ell), \alpha\}$.

Aussage (i) ist klar. Aussage (ii) lässt sich durch Induktion nach ℓ beweisen. Mit $\beta = +\infty$ ergibt sich die Aussage von Satz 5.3.1. \square

Die worst-case Laufzeit der Minimax-Methode (mit oder ohne α - β -Pruning) ist linear in der Größe des Spielbaums. Die Effizienz von α - β -Pruning hängt wesentlich von der Reihenfolge ab, in der die Söhne des zu expandierenden Knotens betrachtet werden.

Die Effizienz der Minimax-Strategie mit α - β -Pruning ist abhängig von der Reihenfolge, in der die Teilbäume betrachtet werden.



Reihenfolge: "von links nach rechts"

Obige Folie demonstriert, dass die Reihenfolge, in welcher die Söhne eines Knoten (also die Zugmöglichkeiten in gegebener Konfiguration) entscheidend für das Cutting ist. Sowohl im mittleren und rechten Teilbaum ist ein Cutting möglich, wenn die Söhne des mittleren bzw. rechten Minknotens in geeigneter Reihenfolge betrachtet werden, nämlich jeweils der schlechtere Zug aus Sicht von Spieler X.

Deshalb wird α - β -Pruning häufig mit Heuristiken kombiniert, die zu vielen Cuts führen sollen. (Derartige Techniken ergeben einen Branch & Bound-Algorithmus für die Minimax-Technik.) Anzustreben sind dabei Heuristiken, die möglichst schnell zu vorläufigen

Bewertungen $\alpha(v, \ell)$ bzw. $\beta(w, \ell)$ führen, welche nahe bei den exakten Werten $c(v, \ell)$ bzw. $c(w, \ell)$ liegen. Salopp formuliert bedeutet dies, dass vielversprechende Spielzüge (etwa Schlagzüge für Spiele wie Mühle, Dame oder Schach) höhere Priorität haben sollten.

Der angegebene Minimax-Algorithmus sowie das α - β -Pruning arbeiten nur gedanklich mit dem Spielbaum. Die Knoten v, w , welche als Parameter übergeben werden, sind Spielkonfigurationen (also Knoten im Spielgraph). Für jeden Maxknoten v und jede Restsuchtiefe ℓ können mehrere Aufrufe von $\text{MaxValue}(v, \ell, \dots)$ mit demselben β -Wert oder unterschiedlichen β -Werten stattfinden. Durch den Einsatz geeigneter Hashtechniken können derartige Mehrfachrechnungen teilweise umgangen werden.

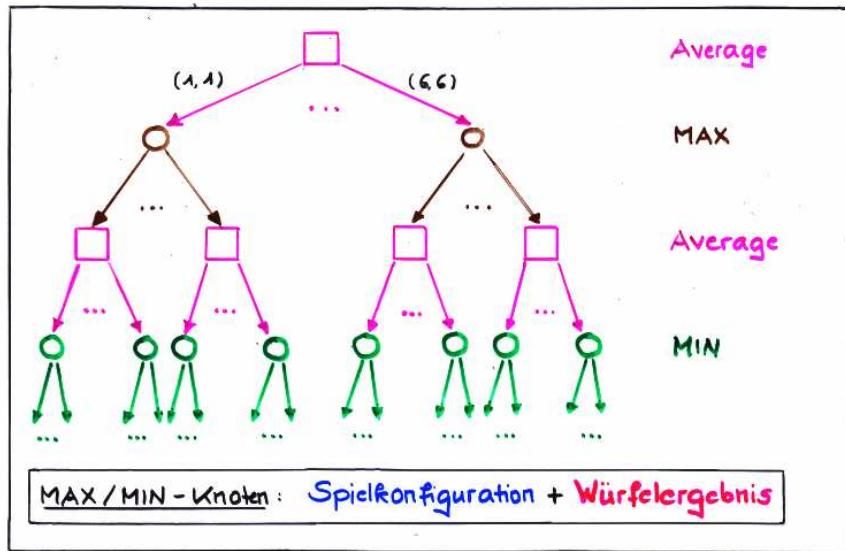
Ein weiterer Ansatz zur Effizienzsteigerung besteht darin, das Suchfenster $[-\infty, \beta]$ für den Wert $\alpha = \alpha(v, \ell)$ in $\text{MaxValue}(v, \ell, \beta)$ eines Maxknotens v auf einen Bereich $[\alpha^-, \beta]$ einzuschränken. Dies entspricht der Initialisierung „ $\alpha := \alpha^-$ “ anstelle von „ $\alpha := -\infty$ “ in $\text{MaxValue}(v, \ell, \beta)$. Hierzu kann eine beliebige untere Schranke $\alpha^- < c(v, \ell)$ eingesetzt werden. Analoges gilt für Minknoten, für die β in $\text{MinValue}(w, \ell, \alpha)$ mit einer beliebigen oberen Schranke $\beta^+ > c(w, \ell)$ initialisiert werden kann. Beispielsweise kann man für die zuletzt betrachtete Folgekonfiguration w von Maxknoten v das Suchfenster $[\alpha, +\infty]$ in $\text{Minvalue}(w, \ell, \alpha)$ auf $[\alpha, \alpha + 1]$ verkleinern, da für den letzten Sohn w von v lediglich die Frage, ob die Bewertung von w besser als α ist, bedeutsam ist, nicht aber der exakte Wert $c(w, \ell)$. (Hierbei setzen wir eine ganzzahlige Payoff-Funktion voraus.) Derartige untere bzw. obere Schranken (α^- bzw. β^+) können sich eventuell auch durch spielabhängige Heuristiken ergeben.

Experimentelle Ergebnisse haben gezeigt, daß unter der Verwendung guter Heuristiken zum Erzeugen vieler Cuts mit α - β -Pruning ca. $b^{3/4 \cdot d}$ Konfigurationen (statt b^d) untersucht werden. Dabei ist b der durchschnittliche Verzweigungsgrad der inneren Knoten des Spielbaums und d die Suchtiefe.

Spiele mit Zufallsfaktoren z.B. Backgammon. Prinzipiell sind ähnliche Strategien anwendbar. Der wesentliche Unterschied ist, daß drei Arten von Knoten im Spielgraphen/Spielbaum betrachtet werden: Minknoten, Maxknoten (wie zuvor) und *Zufallsknoten*.

Spielbäume für Zwei-Personen-Spiele mit Würfeln

z.B. Backgammon



Z.B. beim Backgammon stehen die aus einem Zufallsknoten hinausführenden Kanten für die möglichen Würfelergebnisse. Die Min-/Maxknoten stehen für eine Spielkonfiguration und ein konkretes Würfereignis. Zur Bewertung der Zufallsknoten wird der Erwartungswert der Bewertungen der Söhne verwendet.

5.4 Die Greedy-Methode

Greedy-Algorithmen verfolgen eine „gierige“ Strategie, bei der – anhand einer gewissen Auswahlfunktion – vielversprechende Objekte zuerst betrachtet und getroffene Entscheidungen nicht revidiert werden. Greedy-Algorithmen werden häufig zum Lösen von Optimierungsproblemen eingesetzt.

Für Probleme, in denen eine gewisse Menge von Objekten auszuwählen ist, wird für jedes analysierte Objekt x entschieden, ob x verworfen oder in die Lösungsmenge aufgenommen wird. Für Problemstellungen, in denen gewisse Funktionswerte $f(x)$ zu berechnen sind, wird der endgültige Wert $f(x)$ berechnet. Das allgemeine Schema, von dem konkrete Algorithmen mehr oder weniger abweichen, ist in Algorithmus 60 angegeben. Wir gehen dort von einer Darstellung der gesuchten Lösung durch eine Funktion f aus. Z.B. kann f für die charakteristische Funktion einer zu berechnenden Lösungsmenge stehen. Oftmals wählt die Auswahlfunktion ein Objekt, welches hinsichtlich einer in der Vorverarbeitung erstellten Sortierung minimal oder maximal ist.

Algorithmus 60 Allgemeines Schema für Greedy-Algorithmen

(* Sei S die zu analysierende Menge. *)

Vorverarbeitung von S (z.B. Sortierung);

$A := S;$

WHILE $A \neq \emptyset$ **DO**

(* Auswahlfunktion *)

berechne $f(x)$; (* eventuell unter Verwendung der Werte $f(y)$ für $y \in S \setminus A$ *)

$A := A \setminus \{x\}$ (* Entscheidung für die Wahl von $f(x)$ ist endgültig *)

OD

Beispielsweise sind die in Abschnitt 5.2.1 auf Seite 215 ff angegebenen Verfahren für das Rucksackproblem (Algorithmen 50 und 51) Instanzen des angegebenen Schemas für Greedy-Algorithmen. Für das rationale Rucksackproblem ist $f(i) = x_i \in [0, 1]$ der Bruchteil, der von Objekt i mitgenommen wird. Für das $\{0, 1\}$ -Rucksackproblem kann die durch $f(i) = x_i \in \{0, 1\}$ definierte Funktion als charakteristische Funktion der berechneten Rucksackfüllung angesehen werden. Auch der in Algorithmus 47 auf Seite 210 angegebenen Graphfarbe-Heuristik liegt das Greedy-Schema zugrunde, allerdings in etwas abweichender Form.

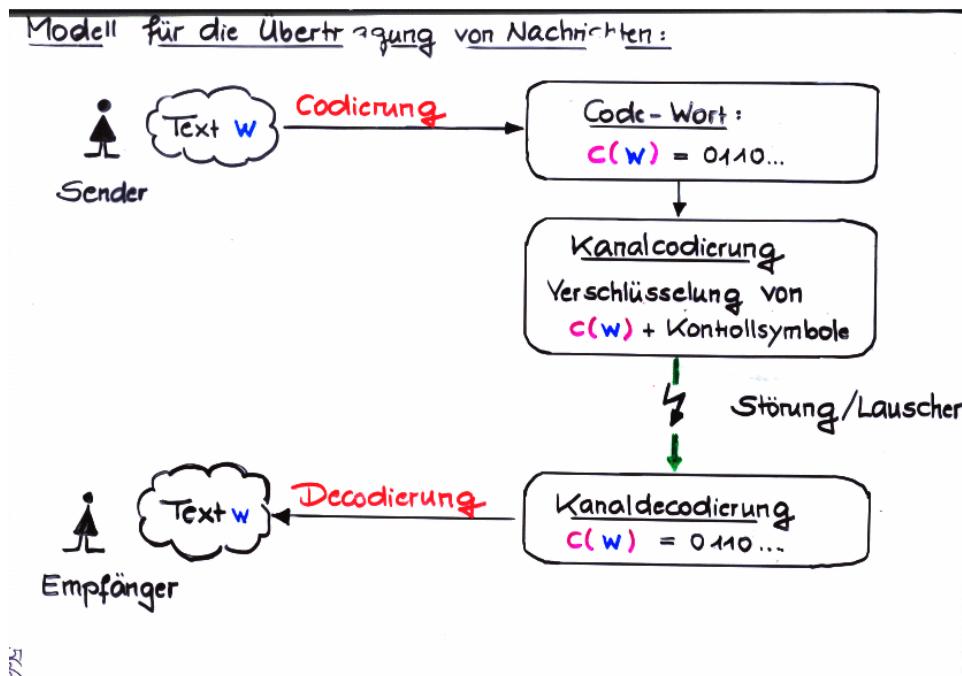
Neben dem $\{0, 1\}$ -Rucksack- und Graphfarbe-Problem kann die Greedy-Methode für viele andere NP-harte Probleme eingesetzt werden, um effiziente heuristische Verfahren zu entwerfen. Da Greedy-Algorithmen meist polynomielle Laufzeit haben, ist es nicht verwunderlich, daß diese – angesetzt auf NP-harte Probleme – keine korrekte oder optimale Lösung liefern.

Am Beispiel des rationalen Rucksackproblems sieht man jedoch, daß diese Beobachtung keinesfalls ausschließt, daß (nicht-NP-harte) Optimierungsprobleme mit der Greedy-Methode effizient gelöst werden können. Siehe Satz 5.2.5 auf Seite 217. Weitere Beispiele für Greedy-Algorithmen, welche wir im Verlauf der Vorlesung kennenlernen werden,

sind die Konstruktion optimaler Präfixcodes (siehe Abschnitt 5.4.1 unten), kürzeste Wege-Probleme sowie minimale aufspannende Bäume (siehe Kapitel 6).

5.4.1 Huffman-Codes

Bei der Nachrichtenübertragung wird der zu versendende Text w zunächst *codiert*. Das Codewort ist üblicherweise eine 0-1-Folge (und kann auch als Folge von Morsezeichen aufgefaßt werden). Nach dieser sogenannte Quellencodierung wird das Codewort über einen Kanal an den Empfänger weitergeleitet. Dieser ist der eigentlich heikle Schritt, da der Kanal Störungen ausgesetzt sein kann (z.B. atmosphärische Störungen bei der Satellitenübertragung) und da die Nachricht vor unbefugten Lauschern geschützt werden soll.



Wir betrachten hier nur den ersten Schritt, in dem der ursprüngliche Text durch eine Bitfolge codiert wird. Ziel ist es, ein möglichst kurzes Codewort zu generieren.

Definition 5.4.1 (Zur Erinnerung: Alphabet, Wörter und co). Sei Σ ein Alphabet, d.h. eine endliche Menge von Zeichen oder Symbolen. Ein *endliches Wort* (im Folgenden kurz *Wort* genannt) über Σ ist eine endliche eventuell leere Folge $w = x_1 x_2 \dots x_r$ von Symbolen $x_i \in \Sigma$.⁴¹ r heißt die Länge des Worts w und wird auch mit $|w|$ bezeichnet. Das *leere Wort* ist das Wort der Länge 0 und wird mit dem griechischen Buchstaben ε („epsilon“) bezeichnet. Sind $w = x_1 \dots x_r$ und $w' = y_1 \dots y_l$ zwei Wörter über Σ , dann steht $w \circ w'$

⁴¹Für die Wörter eines Alphabets sind beide Schreibweisen „mit Kommas“ oder „ohne Kommas“ gebräuchlich. Etwa für $\Sigma = \{0, 1\}$ sind die Schreibweisen 00110 und 0, 0, 1, 1, 0 gleichwertig.

oder kurz ww' für das Wort $x_1 \dots x_r y_1 \dots y_l$, das sich durch Hintereinanderhängen der Wörter w und w' ergibt.⁴² Ein Wort w' heißt

- *Präfix* eines Worts w_0 , falls es ein Wort w gibt, so daß $w_0 = w'w$.
- *Suffix* eines Worts w_0 , falls es ein Wort w gibt, so daß $w_0 = ww'$.
- *Teilwort* eines Worts w_0 , falls es ein Wörter w_1, w_2 gibt, so daß $w_0 = w_1 w' w_2$.

Σ^* bezeichnet die Menge aller endlichen Wörter über Σ , $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ die Menge aller endlichen, nicht leeren Wörter. \square

Codierungen. Der zu codierende Text w wird als nicht-leeres Wort über einem Alphabet Σ interpretiert. Beispielsweise kann man für deutschsprachige Texte das Alphabet Σ aller Klein- und Großbuchstaben und Sonderzeichen (Leerzeichen, Kommas, Punkt, etc.) verwenden. Ein *Binärcode* für Σ ist eine injektive Abbildung

$$c : \Sigma \rightarrow \{0, 1\}^+.$$

Das *Codewort* (oder die *Codierung*) eines Worts $w = x_1 \dots x_r \in \Sigma^*$ bzgl. c ist das Wort

$$c(w) = c(x_1)c(x_2) \dots c(x_r).$$

(Dabei nehmen wir an, daß $x_1, \dots, x_r \in \Sigma$.) Die *Länge* der Codierung $c(w)$ ist

$$|c(w)| = \sum_{i=1}^r |c(x_i)| = \sum_{x \in \Sigma} |c(x)| \cdot \text{Anzahl der Vorkommen von } x \text{ in } w.$$

Für Binärcodes fester Länge (etwa $|c(x)| = k$ für alle $x \in \Sigma$), ist $|c(w)| = k \cdot |w|$ nur von der Textlänge abhängig. Kürzere Codewörter kann man erhalten, indem man Binärcodes verwendet, in denen die Wörter $c(x)$, $x \in \Sigma$, unterschiedliche Länge haben dürfen.

Liegt z.B. ein Text w der Länge 100.000 über dem Alphabet $\Sigma = \{A, B, C, D, E, F\}$ vor, so kann ein Binärcode fester Länge mit je drei Bits pro Zeichen verwendet werden, etwa

x	A	B	C	D	E	F
$c(x)$	000	001	010	011	100	101

Man erhält ein Codewort der Länge $|c(w)| = 300.000$. Ein kürzeres Codewort erhält man, indem man sich an den Häufigkeiten der Zeichen im Text orientiert und die Codierungen $c(x)$ in Abhängigkeit der Anzahl $h(x)$ an Vorkommen des Zeichens x in w wählt. Etwa

x	A	B	C	D	E	F
$h(x)$	50.000	40.000	7.000	1.000	1.000	1.000
$c(x)$	0	100	101	111	1100	1101

⁴²Beachte, daß $\varepsilon w = w\varepsilon = w$.

Nun beträgt die Codierungslänge nur noch

$$50.000 + 3 * 40.000 + 3 * 7.000 + 3 * 1.000 + 4 * 1.000 + 4 * 1.000 = 202.000.$$

Der Empfänger erhält die verschlüsselte Nachricht, d.h. das Codewort $c(w)$. Um auf den ursprünglichen Text w zu schließen, muß er das Codewort entschlüsseln. Wir setzen voraus, daß dem Empfänger das Alphabet Σ sowie der Code c bekannt ist. Für Binärcodes fester Länge (d.h. die Codewörter aller Zeichen $x \in \Sigma$ haben dieselbe Länge) ist der Decodervorgang offensichtlich.

Für Binärcodes variabler Länge sind jedoch zusätzliche Voraussetzung nötig, die eine eindeutige Entschlüsselung möglich machen, d.h. welche die Injektivität der erweiterten Abbildung $c : \Sigma^* \rightarrow \{0, 1\}^*$ garantieren. Ist z.B. $\Sigma = \{A, B, C\}$, so läßt der Binärcode $c(A) = 0$, $c(B) = 1$ und $c(C) = 00$ keine eindeutige Entschlüsselung zu, da der Empfänger des Codesworts $c(w) = 00$ nicht wissen kann, ob die gesendete Nachricht $w = AA$ oder $w = C$ ist.

Definition 5.4.2 (Präfixcode). Sei Σ ein Alphabet. Ein *Präfixcode* für Σ ist ein Binärcode $c : \Sigma \rightarrow \{0, 1\}^+$, so dass für alle $x \in \Sigma$ gilt:

Das Wort $c(x)$ ist kein Präfix der Codierung $c(y)$ eines anderen Zeichens $y \in \Sigma$.

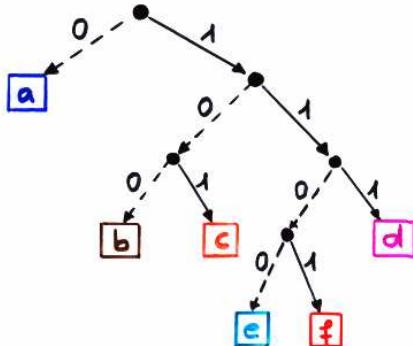
□

Offenbar gilt für jeden Präfixcode c : Aus $c(w_1) = c(w_2)$ folgt $w_1 = w_2$. D.h. Präfixcodes erlauben eine eindeutige Entschlüsselung. (Man beachte, daß jeder Binärcode mit fester Codelänge ein Präfixcode ist.)

Baumdarstellung von Präfixcodes. Präfixcodes können durch einen Binärbaum, dessen Kanten mit 0 oder 1 beschriftet sind, dargestellt werden. Die Kante von v zum linken Sohn $left(v)$ ist stets mit 0, die Kante von v zum rechten Sohn $right(v)$ mit 1 beschriftet. Die Blätter sind mit den Symbolen $x \in \Sigma$ markiert. (Genauer: Jedes Symbol $x \in \Sigma$ ist durch genau ein Blatt repräsentiert.) Das Bitwort, welches sich aus den Pfaden von der Wurzel zu dem Blatt mit der Markierung x ergibt, entspricht der Codierung $c(x)$.

Darstellung eines Präfix-Codes als Binärbaum

a	b	c	d	e	f
0	100	101	111	1100	1101



Decodierung durch wiederholte Baumtraversierung

G

Dieser Baum dient als Implementierungshilfe für den Decodervorgang. Liegt dem Empfänger das Codewort $\beta = b_1 b_2 \dots b_l \in \{0, 1\}^+$ vor, dann kann durch wiederholte Baumtraversierungen auf das codierte Wort w geschlossen werden.

Huffman-Codes. Eine naheliegende Strategie zum Auffinden eines Präfixcodes c , für den das Codewort $c(w)$ des zu versendenden Texts w möglichst kurz ist, besteht darin, häufig vorkommende Zeichen $x \in \Sigma$ möglichst kurz zu codieren. Anstelle der absoluten Häufigkeiten für die Vorkommen der Zeichen $x \in \Sigma$ in dem Text w verwenden wir eine Wahrscheinlichkeitsverteilung $p : \Sigma \rightarrow [0, 1]$. Beispielsweise kann $p(x)$ für die durchschnittliche relative Häufigkeit von x in Texten $w \in \Sigma^+$ eines gewissen Typs stehen. Z.B. ist für Texte der deutschen Sprache bekannt, dass die Buchstaben „E“ und „N“ am häufigsten vorkommen, während die Buchstaben „X“ und „Y“ eher selten erscheinen. Liegt ein konkreter Text w vor, der optimal zu codieren ist, so kann man $p(x) = h(x)/|w|$ setzen, wobei $h(x)$ die Anzahl an Vorkommen von x in w bezeichnet.

Definition 5.4.3 (Mittlere Codelänge, optimaler Präfixcode). Die mittlere Codelänge der Zeichen $x \in \Sigma$ von c bzgl. der Wahrscheinlichkeitsverteilung $p : \Sigma \rightarrow [0, 1]$ ist

$$L_{c,p} = \sum_{x \in \Sigma} p(x) \cdot |c(x)|.$$

Im Folgenden schreiben wir häufig L_c statt $L_{c,p}$. Ein Präfixcode c für Σ heißt *optimal* für (Σ, p) (oder *Huffman-Code* für (Σ, p)), wenn die mittlere Codelänge $L_{c,p}$ minimal ist unter allen Präfixcodes für Σ . \square

Als Beispiel betrachten wir folgenden Präfixcode c für das Alphabet $\Sigma = \{A, B, C, D, E, F\}$ mit Wahrscheinlichkeitsverteilung p :

x	A	B	C	D	E	F
$c(x)$	0	100	101	111	1100	1101
$p(x)$	$1/2$	$1/4$	$3/32$	$1/16$	$1/16$	$1/32$

Die mittlere Codelänge $L_{c,p}$ ergibt sich durch die gewichtete Summe

$$1 \cdot \frac{1}{2} + 3 \cdot \frac{1}{4} + 3 \cdot \frac{3}{32} + 3 \cdot \frac{1}{16} + 4 \cdot \frac{1}{16} + 4 \cdot \frac{1}{16} = 2 + \frac{7}{32}.$$

Stimmen die Werte $p(x)$ mit der relativen Häufigkeit $h(x)/|w|$ von x in dem Text w überein, dann ist $|c(w)| = L_{c,p} \cdot |w|$.

Algorithmus von Huffman. Gegeben ist das Alphabet Σ sowie eine Wahrscheinlichkeitsverteilung p für Σ , wobei $|\Sigma| = n \geq 2$ vorausgesetzt wird. Gesucht ist ein Präfixcode für Σ , so daß die mittlere Codelänge von c bzgl. p minimal ist.

Algorithmus 61 Algorithmus zur Erstellung eines optimalen Präfixcodes

Erstelle eine Priority Queue Q (Minimumsheap) bestehend aus den Symbolen $x \in \Sigma$, sortiert hinsichtlich der Wahrscheinlichkeiten $p(x)$.

(* Die Symbole $x \in \Sigma$ werden als Blätter des zu konstruierenden Baums angesehen. *)

FOR $i = 1, \dots, n - 1$ **DO**

$v := EXTRACT_MIN(Q);$

$w := EXTRACT_MIN(Q);$

generiere einen neuen Knoten u mit $p(u) = p(v) + p(w)$, $left(u) = v$ und $right(u) = w$;

füge u in Q ein;

OD

Der Greedy-Algorithmus (siehe Algorithmus 61) zum Erstellen des optimalen Präfixcodes baut in Bottom-Up-Manier den Baum für den Code auf. Initial werden die Symbole $x \in \Sigma$ als Blätter mit der Wahrscheinlichkeitsmarkierung $p(x)$ dargestellt. Die Auswahlfunktion sucht unter allen (noch) vaterlosen Knoten des bereits erstellten Teils des Baums zwei Knoten v, w , die eine minimale Wahrscheinlichkeitsmarkierung haben. Diese werden zu den Söhnen eines neuen Knotens u mit der Wahrscheinlichkeitsmarkierung $p(u) = p(v) + p(w)$ gemacht.

Die Menge der vaterlosen Knoten kann in einer Priority Queue verwaltet werden (siehe Abschnitt 3.3, Seite 121 ff). Damit ergeben sich pro Schleifendurchlauf logarithmische Kosten. Man erhält die Gesamtkosten $\Theta(n \log n)$.

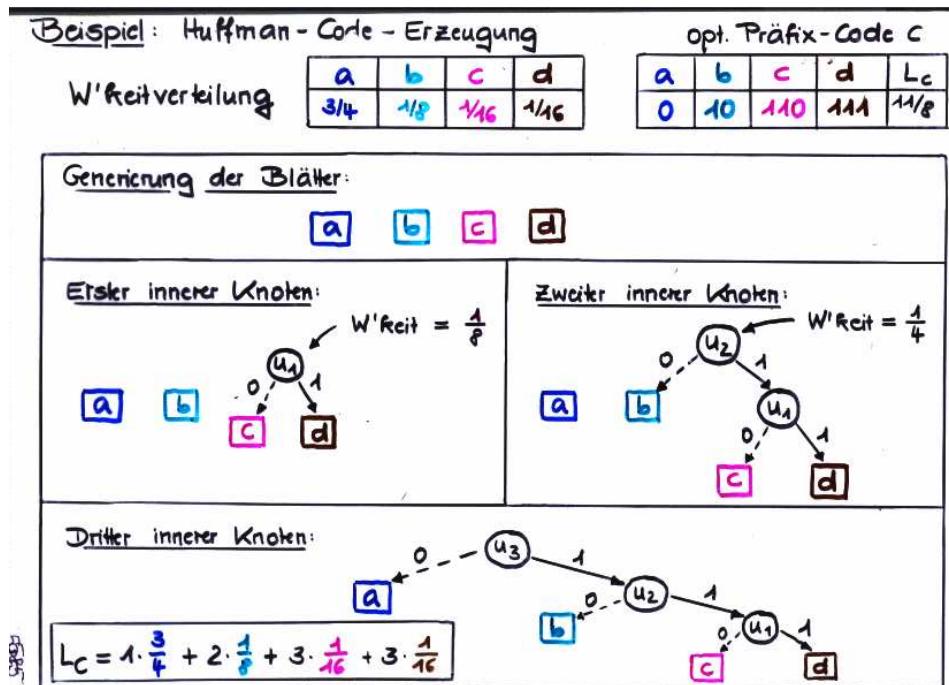
Bemerkung 5.4.4 (Anzahl an Iterationen im Algorithmus von Huffman). Ist $|\Sigma| = n$, dann ist nach $n - 1$ Schritten ein Binärbaum entstanden, dessen Blätter in Eins-zu-Eins-Beziehung zu den Symbolen des Alphabets stehen und somit einen Präfixcode für Σ repräsentiert. Man beachte, dass nach $n - 1$ Iterationen tatsächlich ein Binärbaum

entstanden ist. Dies erklärt sich aus der Beobachtung, dass ein gerichteter Wald konstruiert wird, dessen innere Knoten jeweils zwei Söhne haben. Jeder Baum mit dieser Eigenschaft und m Blättern hat $m - 1$ innere Knoten. Hieraus ergibt sich folgende Schleifeninvariante für den in Algorithmus 61 konstruierten Wald:

$$n - \text{Anzahl an inneren Knoten} = \text{Anzahl an Bäumen}$$

Da in jeder Iteration ein innerer Knoten erzeugt wird, liegt nach $n - 1$ Schritten ein Baum (genauer: ein Wald bestehend aus einem Baum) vor. \square

Folgende Folie illustriert den Algorithmus an einem Beispiel:



Satz 5.4.5 (Korrektheit und Laufzeit von Algorithmus 61). Huffman-Codes können mit der Greedy-Methode in Zeit $\Theta(n \log n)$ berechnet werden.

Die angegebene Laufzeit $\Theta(n \log n)$ folgt aus den Überlegungen zu Priority Queues. Für den Nachweis der Korrektheit des Algorithmus ist zu zeigen, daß der im Algorithmus konstruierte Präfixcode optimal ist. Hierzu weisen wir nach (vgl. Lemma 5.4.9 unten), daß ein optimaler Präfixcode für ein mindestens dreielementiges Alphabet Σ bestimmt werden kann, indem zwei Zeichen des Alphabets mit den geringsten Zugriffswahrscheinlichkeiten wie im Algorithmus zu einem Zeichen zusammengefasst werden.

Im Folgenden sei x_1, \dots, x_n eine Aufzählung der Zeichen des Alphabets Σ mit $x_i \neq x_j$ für $i \neq j$ und

$$p(x_n) \geq p(x_{n-1}) \geq \dots \geq p(x_2) \geq p(x_1).$$

Ferner setzen wir $n \geq 3$ voraus, da für $n = 2$ die Optimalität des berechneten Präfixcodes offensichtlich ist. (Für $n = 2$ wird lediglich ein Wurzelknoten generiert, dessen beide

Söhne die Blätter mit den beiden Zeichen von Σ sind. Es wird also ein Präfixcode c der mittleren Codelänge 1 erzeugt. Dieser ist offenbar optimal.)

Wir beschreiben zunächst eine Transformation $c \mapsto \bar{c}$, welche einen gegebenen Präfixcode c zu einem Präfixcode \bar{c} modifiziert, dessen mittlere Codelänge kürzer oder gleich der mittleren Codelänge von c ist und in dessen Baumdarstellung die Elemente x_1, x_2 einen gemeinsamen Vater haben.

Lemma 5.4.6 (Transformation $c \mapsto \bar{c}$). *Für jeden Präfixcode c für Σ gibt es einen Präfixcode \bar{c} für Σ mit*

- (1) $L_{c,p} \geq L_{\bar{c},p}$
- (2) $\bar{c}(x_1) = \gamma 0$ und $\bar{c}(x_2) = \gamma 1$ für ein Wort $\gamma \in \{0, 1\}^*$.

Beweis. Wir beschreiben den Übergang $c \mapsto \bar{c}$ durch Transformationen auf dem Baum \mathcal{T} , der c zugeordnet ist.

1. Schritt: Falls \mathcal{T} einen Teilbaum $\mathcal{T}(x)$ der Höhe ≥ 1 enthält, der nur genau Blatt für das Zeichen x enthält, so kann der komplette Teilbaum $\mathcal{T}(x)$ durch das Blatt x ersetzt werden. Offenbar wird hierdurch die mittlere Codelänge nicht verlängert, da $c(x)$ durch ein kürzeres oder gleichlanges Codewort ersetzt wird und die Codierungen der anderen Zeichen erhalten bleibt. Diese Transformation wird solange angewandt bis jeder innere Knoten in \mathcal{T} einen Teilbaum mit mindestens zwei Blättern induziert.

2. Schritt: Wir wählen nun einen inneren Knoten v maximaler Tiefe in \mathcal{T} . Die Söhne von v sind dann Blätter, etwa y_1, y_2 . Wir vertauschen nun (y_1, y_2) und (x_1, x_2) , gehen also zum Baum für den Präfixcode \bar{c} mit

$$\bar{c}(z) = \begin{cases} c(z) & : \text{falls } z \notin \{x_1, x_2, y_1, y_2\} \\ c(x_i) & : \text{falls } z = y_i \\ c(y_i) & : \text{falls } z = x_i \end{cases}$$

über. Es gilt dann

$$|\bar{c}(y_i)| = |c(x_i)| \leq |c(y_i)| = |\bar{c}(x_i)| = Tiefe(v) + 1,$$

da v ein Knoten maximaler Tiefe ist. Insbesondere ist $|\bar{c}(y_1)| - |c(y_1)| \leq 0$ für

$i = 1, 2$. Hieraus folgt:

$$\begin{aligned}
L_{\bar{c},p} - L_{c,p} &= \underbrace{p(y_1) \cdot (\underbrace{|\bar{c}(y_1)| - |c(y_1)|}_{\leq 0})}_{\geq p(x_1)} + p(x_1) \cdot (|\bar{c}(x_1)| - |c(x_1)|) \\
&\quad + \underbrace{p(y_2) \cdot (\underbrace{|\bar{c}(y_2)| - |c(y_2)|}_{\leq 0})}_{\geq p(x_2)} + p(x_2) \cdot (|\bar{c}(x_2)| - |c(x_2)|) \\
&\leq p(x_1) \cdot (|\bar{c}(y_1)| - |c(y_1)| + \underbrace{|\bar{c}(x_1)| - |c(x_1)|}_{=|c(y_1)|}) \\
&\quad + p(x_2) \cdot (|\bar{c}(y_2)| - |c(y_2)| + \underbrace{|\bar{c}(x_2)| - |c(x_2)|}_{=|c(y_2)|}) \\
&= 0
\end{aligned}$$

Also ist $L_{\bar{c},p} \leq L_{c,p}$.

Offenbar erfüllt die so konstruierte Präfixcode \bar{c} die gewünschten Bedingungen (1) und (2). \square

Corollar 5.4.7. Es gibt einen optimalen Präfixcode \bar{c} mit $\bar{c}(x_1) = \gamma 0$ und $\bar{c}(x_2) = \gamma 1$ für ein Wort $\gamma \in \{0, 1\}^*$.

Transformation $\bar{c} \leftrightarrow c'$. Sei c ein Präfixcode mit $c(x_1) = \gamma 0$ und $c(x_2) = \gamma 1$ für ein Wort $\gamma \in \{0, 1\}^*$. Ist Σ zweielementig, also $\Sigma = \{x_1, x_2\}$, dann ist c offenbar ein optimaler Präfixcode für Σ . Im Folgenden nehmen wir an, daß Σ mindestens drei Zeichen enthält. Weiter sei

$$\Sigma' = (\Sigma \setminus \{x_1, x_2\}) \cup \{x'\},$$

wobei $x' \notin \Sigma$. Die Präfixcodes für Σ' stehen in Eins-zu-Eins-Beziehung zu den Präfixcodes \bar{c} für Σ , welche die Eigenschaft (2) in Lemma 5.4.6 haben. Ist nämlich \bar{c} ein solcher Präfixcode für Σ , dann können wir \bar{c} mit dem Präfixcode $c' : \Sigma' \rightarrow \{0, 1\}^+$,

$$c'(y) = \begin{cases} \bar{c}(y) & : \text{falls } y \in \Sigma' \setminus \{x'\} \\ \gamma & : \text{falls } y = x' \end{cases}$$

identifizieren. Umgekehrt kann jeder Präfixcode c' für Σ' durch

$$\bar{c}(y) = \begin{cases} c'(y) & : \text{falls } y \in \Sigma \setminus \{x_1, x_2\} \\ c'(x')0 & : \text{falls } y = x_1 \\ c'(x')1 & : \text{falls } y = x_2 \end{cases}$$

zu einem Präfixcode \bar{c} für Σ mit der Eigenschaft (2) modifiziert werden.

Im Folgenden versehen wir das modifizierte Alphabet Σ' mit der Wahrscheinlichkeitsverteilung $p' : \Sigma' \rightarrow [0, 1]$, welche durch

$$p'(y) = \begin{cases} p(y) & : \text{falls } y \in \Sigma' \setminus \{x'\} \\ p(x_1) + p(x_2) & : \text{falls } y = x' \end{cases}$$

gegeben ist.

Lemma 5.4.8 (Mittlere Codelänge von \bar{c} und c'). *Für die oben beschriebene Eins-zu-Eins-Beziehung $\bar{c} \leftrightarrow c'$ zwischen den Präfixcodes \bar{c} für Σ mit der Eigenschaft (2) und Präfixcodes c' für Σ' gilt:*

$$L_{\bar{c},p} = L_{c',p'} + p'(x')$$

Beweis. Es gilt:

$$\begin{aligned} p(x_1)|\bar{c}(x_1)| + p(x_2)|\bar{c}(x_2)| &= p(x_1)|c'(x')0| + p(x_2)|c'(x')1| \\ &= p(x_1)(|c'(x')| + 1) + p(x_2)(|c'(x')| + 1) \\ &= (p(x_1) + p(x_2)) \cdot (|c'(x')| + 1) \\ &= p'(x')|c'(x')| + p'(x') \end{aligned}$$

und somit:

$$\begin{aligned} L_{\bar{c},p} &= \sum_{x \in \Sigma} p(x) \cdot |\bar{c}(x)| \\ &= \sum_{y \in \Sigma \setminus \{x_1, x_2\}} \underbrace{p(y)}_{=p'(y)} \cdot \underbrace{|\bar{c}(y)|}_{=c'(y)} + \underbrace{p(x_1)|\bar{c}(x_1)| + p(x_2)|\bar{c}(x_2)|}_{=p(x')|c'(x')|+p'(x')} \\ &= \sum_{y \in \Sigma \setminus \{x_1, x_2\}} p'(y) \cdot |c'(y)| + p(x')|c'(x')| + p'(x') \\ &= L_{c',p'} + p'(x') \end{aligned}$$

□

Lemma 5.4.9 (Optimale Präfixcodes für Σ und Σ'). *Sei \bar{c} ein Präfixcode für Σ mit der Eigenschaft (2) aus Lemma 5.4.6 und sei c' der induzierte Präfixcode für Σ' . Dann gilt:*

\bar{c} ist genau dann optimal für (Σ, p) , wenn c' optimal für (Σ', p') ist.

Beweis. Ist c' optimal für (Σ', p') und ist c_0 ein beliebiger Präfixcode für Σ , dann gilt aufgrund der Aussage von Lemma 5.4.8:

$$L_{c_0,p} \geq L_{\bar{c}_0,p} = L_{c'_0,p'} + p'(x') \geq L_{c',p'} + p'(x') = L_{\bar{c},p}$$

Also ist \bar{c} optimal für (Σ, p) . Hier bezeichnet $c_0 \mapsto \bar{c}_0$ die in Lemma 5.4.6 auf Seite 256 beschriebene Transformation, welche c_0 auf einen mindestens ebenso guten Präfixcode \bar{c}_0 für Σ mit der Eigenschaft (2) abbildet. c'_0 ist der durch \bar{c}_0 induzierte Präfixcode für Σ' , der sich durch Zusammenfassen von x_1 und x_2 zu x' ergibt.

Ist umgekehrt \bar{c} optimal für (Σ, p) und c'_0 ein beliebiger Präfixcode für Σ' , so gilt:

$$L_{c'_0, p'} = L_{\bar{c}_0, p} - p'(x') \geq L_{\bar{c}, p} - p'(x') = L_{c', p'}$$

Also ist c' optimal für (Σ', p') . \square

Kombiniert man die Aussage von Corollar 5.4.7 auf Seite 257 mit der Aussage von Lemma 5.4.9, so ergibt sich die Optimalität des durch den Huffman-Algorithmus berechneten Präfixcodes durch Induktion nach $n = |\Sigma|$.

5.4.2 Matroide

Bevor wir weitere Beispiele zu Greedy-Algorithmen geben, befassern wir uns mit einer Theorie, welche die Korrektheit von Greedy-Algorithmen als Lösungsverfahren für Optimierungsprobleme garantieren kann. Dabei konzentrieren wir uns auf Problemstellungen, in denen eine Auswahl an Objekten vorzunehmen ist, die eine gewisse Zielfunktion maximiert.

Definition 5.4.10 (Teilmengensystem, Gewichtsfunktion). Ein Teilmengensystem ist ein Paar (S, \mathcal{U}) bestehend aus einer endlichen Menge S und einer Menge $\mathcal{U} \subseteq 2^S$ (wobei 2^S die Potenzmenge von S bezeichnet), so daß folgende zwei Eigenschaften erfüllt sind:

- (1) $\emptyset \in \mathcal{U}$
- (2) Ist $A \subseteq B \subseteq S$ und $B \in \mathcal{U}$, so gilt $A \in \mathcal{U}$.

Ein Teilmengensystem mit Gewichtsfunktion ist ein Tripel (S, \mathcal{U}, w) bestehend aus einem Teilmengensystem (S, \mathcal{U}) und einer Abbildung $w : S \rightarrow \mathbb{R}_{\geq 0}$, welche jedem Element $x \in S$ ein nicht-negatives Gewicht $w(x)$ zuordnet. Die erweiterte Gewichtsfunktion $w : 2^S \rightarrow \mathbb{R}_{\geq 0}$ ist durch

$$w(A) = \sum_{x \in A} w(x)$$

gegeben. Dabei ist $w(\emptyset) = 0$. Das zu (S, \mathcal{U}, w) gehörende Maximierungsproblem fragt nach einer Menge $L \in \mathcal{U}$, für welche $w(L)$ maximal ist unter allen Mengen $B \in \mathcal{U}$. \square

Für die zu Teilmengensystemen (S, \mathcal{U}, w) gehörenden Maximierungsprobleme verwenden wir die in Algorithmus 62 angegebene Formulierung der Greedy-Methode. Für diese ist klar, daß die ausgegebene Lösung L zu \mathcal{U} gehört. Die Maximalität von $w(L)$ ist jedoch nicht klar. Tatsächlich ist sie nicht zwangsläufig erfüllt; z.B. wenn wir das durch das $\{0, 1\}$ -Rucksackproblem induzierte Teilmengensystem betrachten. Hier ist $S = \{x_1, \dots, x_n\}$ die Menge der fraglichen Objekte und \mathcal{U} die Menge aller Teilmengen A von S , so daß $Gewicht(A) \leq K$ (wobei K die vorgeschriebene Rucksackkapazität ist). Die Funktion w

Algorithmus 62 Greedy-Algorithmus für das zu (S, \mathcal{U}, w) gehörende Maximierungsproblem

Bestimme eine absteigende Sortierung x_1, \dots, x_n der Objekte aus S nach ihren Gewichten.

(* Es gilt nun $w(x_1) \geq w(x_2) \geq \dots \geq w(x_n)$. *)

```
L := ∅;  
FOR k := 1, ..., n DO  
  IF L ∪ {xk} ∈ U THEN  
    L := L ∪ {xk}  
  FI  
OD
```

Gib L aus Lösung aus.

nimmt hier die Rolle der Gewinnfunktion ein. Wie wir bereits gesehen haben, kann die Greedy-Methode für das $\{0, 1\}$ -Rucksackproblem jedoch keine optimale Rucksackfüllung garantieren, selbst dann, wenn die Sortierung nach den relativen Gewinnen (anstelle der absoluten Gewinne) erfolgt.

Tatsächlich fehlt im Rucksackproblem-Teilmengensystem eine Voraussetzung, die wir für den Nachweis der Optimalität der Greedy-Methode benötigen. Dies ist die so genannte Austauscheigenschaft:

Definition 5.4.11 (Matroid). Ein Matroid ist ein Teilmengensystem $\mathcal{M} = (S, \mathcal{U})$, welches die sogenannte Austauscheigenschaft, erfüllt:

(3) Sind $A, B \in \mathcal{U}$ mit $|A| < |B|$, so gibt es ein $x \in B \setminus A$, so daß $A \cup \{x\} \in \mathcal{U}$.

□

Bevor wir in folgendem Satz die Korrektheit von Algorithmus 62 für Matroide nachweisen, machen wir uns klar, warum Greedy-Algorithmen oftmals effizient sind. Für die Sortierphase können wir die Kosten $\mathcal{O}(n \log n)$ veranschlagen. Die FOR-Schleife wird n -mal durchlaufen. Ihre Kosten werden dominiert durch den Test, ob $L \cup \{x_k\} \in \mathcal{U}$. Wir nehmen an, daß hierfür die Kosten $T(k)$ entstehen. (Beachte, daß $|L| \leq k - 1$ zu Beginn der k -ten Iteration.) Insgesamt erhalten wir damit die Laufzeit

$$\mathcal{O}\left(n \log n + \sum_{i=1}^k T(i)\right).$$

Setzt man die Monotonie von T voraus (was für eine Kostenfunktion eine natürliche Eigenschaft ist), so können die Kosten durch $\mathcal{O}(n \log n + nT(n))$ abgeschätzt werden. Im günstigsten Fall ist $T(n) = \Theta(1)$ und man erhält die Gesamtkosten $\mathcal{O}(n \log n)$. Für $T(n) = \mathcal{O}(n)$ erhält man die Kosten $\mathcal{O}(n^2)$.

Satz 5.4.12 (Korrekttheit der Greedy-Methode für Matroide (Maximierungsprobleme)). Sei (S, \mathcal{U}, w) ein Teilmengensystem mit Gewichtsfunktion, so daß (S, \mathcal{U}) ein Matriod ist. Dann wird mit der Greedy-Methode (Algorithmus 62) eine optimale Lösung für das zu (S, \mathcal{U}, w) gehörende Maximierungsproblem bestimmt.

Beweis. Sei (S, \mathcal{U}) ein Matroid, wobei $S = \{x_1, \dots, x_n\}$ und $w(x_1) \geq w(x_2) \geq \dots \geq w(x_n)$. Sei $L = \{x_{i_1}, \dots, x_{i_k}\}$ die von Algorithmus 62 gefundene Lösung, wobei wir $i_1 < i_2 < \dots < i_k$ voraussetzen. Offenbar ist $L \in \mathcal{U}$.

Wir nehmen an, daß L keine Lösung des zu (S, \mathcal{U}, w) gehörenden Maximierungsproblems ist, also dass $w(L)$ nicht maximal ist. Dann gibt es ein $L' = \{x_{j_1}, \dots, x_{j_r}\} \in \mathcal{U}$ mit $w(L') > w(L)$. Wir können o.E. annehmen, daß $j_1 < j_2 < \dots < j_r$. Wegen $w(x) \geq 0$ und $w(L') > w(L)$ ist $L' \subseteq L$ nicht möglich. Ebenso kann der Fall $L \subseteq L'$, also $x_{i_1} = x_{j_1}, \dots, x_{i_k} = x_{j_k}$ und $k < r$, ausgeschlossen werden, da dann wenigstens eines der Elemente x_μ mit $i_k < \mu \leq j_{k+1}$ zu L hinzugefügt worden wäre. Mit denselben Argumenten kann ausgeschlossen werden, dass $w(x_{i_\mu}) \leq w(x_{j_\mu})$, $\mu = 1, \dots, \min\{k, r\}$, da dann für $k < r$ ein weiteres Element in L eingefügt worden wäre bzw. $w(L') \leq w(L)$ für $k \geq r$. Daher gibt es einen Index ℓ mit

$$\begin{aligned} w(x_{i_1}) &\geq w(x_{j_1}), \\ w(x_{i_2}) &\geq w(x_{j_2}), \\ &\vdots \\ w(x_{i_{\ell-1}}) &\geq w(x_{j_{\ell-1}}) \\ w(x_{i_\ell}) &< w(x_{j_\ell}) \end{aligned}$$

Da die Elemente x_1, \dots, x_n nach ihren Gewichten absteigend sortiert sind, gilt:

$$j_\ell < i_\ell$$

Wir betrachten nun die Teillösungen

$$A = \{x_{i_1}, \dots, x_{i_{\ell-1}}\} \text{ und } B = \{x_{j_1}, \dots, x_{j_{\ell-1}}, x_{j_\ell}\}.$$

Man beachte, daß A diejenige Teillösung von L ist, die nach den ersten $i_\ell - 1$ Durchläufen der FOR-Schleife vorliegt. Wegen $A \subseteq L \in \mathcal{U}$ und $B \subseteq L' \in \mathcal{U}$ gilt $A, B \in \mathcal{U}$. Wegen $|A| < |B|$ garantiert die Austauscheigenschaft die Existenz eines Elements $x_{j_\rho} \in B \setminus A$, so daß $A \cup \{x_{j_\rho}\} \in \mathcal{U}$. Es gilt dann $j_\rho \leq j_\ell < i_\ell$. Wegen

$$w(x_{j_\rho}) \geq w(x_{j_\ell}) > w(x_{i_\ell})$$

hätte der Greedy-Algorithmus das Element x_{j_ρ} vor x_{i_ℓ} auswählen müssen. Widerspruch!

□

Für die Korrektheit von Greedy-Algorithmen für Maximierungsprobleme genügt es also zu zeigen, dass die betreffende Fragestellung als Matroid und der fragliche Algorithmus als Instanz von Algorithmus 62 aufgefasst werden kann.

Der folgende Satz ist eher von theoretischen Interesse. Er belegt, dass die Austauscheigenschaft wesentlich ist, um die Korrektheit der Greedy-Methode für *jede* Gewichtsfunktion sicherzustellen.

Satz 5.4.13. Ist (S, \mathcal{U}) ein Teilmengensystem, welches die Austauscheigenschaft nicht besitzt, so gibt es eine Gewichtsfunktion $w : S \rightarrow \mathbb{R}_{\geq 0}$, so daß der Greedy-Algorithmus (Algorithmus 62) eine nicht-optimale Lösungsmenge L berechnet.

Beweis. Da die Austauscheigenschaft für (S, \mathcal{U}) nicht erfüllt ist, gibt es $A, B \in \mathcal{U}$ mit $|A| < |B|$, so daß $A \cup \{x\} \notin \mathcal{U}$ für alle $x \in B \setminus A$. Wir definieren nun die Gewichtsfunktion w wie folgt:

$$w(x) = \begin{cases} |B| + 1 & : \text{falls } x \in A \\ |B| & : \text{falls } x \in B \setminus A \\ 0 & : \text{sonst.} \end{cases}$$

Der Greedyalgorithmus gibt eine Lösungsmenge $L \in \mathcal{U}$ aus, so daß $A \subseteq L$ und $L \cap (B \setminus A) = \emptyset$. Dann ist

$$w(L) = (|B| + 1) \cdot \underbrace{|A|}_{\leq |B|-1} \leq (|B| + 1) \cdot (|B| - 1) \leq |B|^2 - 1 < |B|^2 \leq w(B)$$

Beachte, dass $w(B) = |B \cap A| \cdot (|B| + 1) + |B \setminus A| \cdot |B| \geq |B \cap A| \cdot |B| + |B \setminus A| \cdot |B| = |B|^2$. Also ist L nicht optimal. \square

Man beachte, daß Satz 5.4.13 keine Aussage über die Korrektheit der Greedy-Methode (im Sinne von Algorithmus 60) macht, sofern der vorliegenden Problemstellungen keine Teilmengenstruktur zugrundeliegt oder der fragliche Greedy-Algorithmus nicht als Instanz von Algorithmus 60 angesehen werden kann. Z.B. ist dies für das rationale Rucksackproblem der Fall.

Minimierungsprobleme. Die bisherigen Betrachtungen zu Matroiden haben sich auf Maximierungsprobleme beschränkt. Wir diskutieren nun den Fall von Minimierungsproblemen. Wie zuvor ist unser Ausgangspunkt ein Teilmengensystem (S, \mathcal{U}) mit einer Gewichtsfunktion $w : S \rightarrow \mathbb{R}_{\geq 0}$. Eine Menge $A \subseteq S$ wird \mathcal{U} -maximal genannt, falls $A \in \mathcal{U}$ und A in keiner anderen Menge $B \in \mathcal{U}$ enthalten ist. Also:

$$A \text{ } \mathcal{U}\text{-maximal gdw (1) } A \in \mathcal{U} \text{ und (2) Aus } A \subseteq B \in \mathcal{U} \text{ folgt } A = B.$$

Das zu (S, \mathcal{U}, w) gehörende Minimierungsproblem fragt nach einer \mathcal{U} -maximalen Menge $L \in \mathcal{U}$, so dass

$$w(L) = \min \{w(A) : A \text{ ist } \mathcal{U}\text{-maximal}\}.$$

Der Greedy-Algorithmus für das durch (S, \mathcal{U}, w) induzierte Minimierungsproblem ist in Algorithmus 63 angegeben. Dieser unterscheidet sich von dem entsprechenden Algorithmus für Maximierungsprobleme (Algorithmus 62) nur dadurch, dass die Elemente von S in der Sortierphase *aufsteigend* sortiert werden.

In Analogie zu Satz 5.4.12 erhalten wir:

Satz 5.4.14 (Korrektheit der Greedy-Methode für Matroide (Minimierungsprobleme)). Sei (S, \mathcal{U}, w) ein Teilmengensystem mit Gewichtsfunktion, so daß (S, \mathcal{U}) ein Matroid ist. Dann wird mit der Greedy-Methode (Algorithmus 63) eine optimale Lösung für das zu (S, \mathcal{U}, w) gehörende Minimierungsproblem bestimmt.

Algorithmus 63 Greedy-Algorithmus für das zu (S, \mathcal{U}, w) gehörende Minimierungsproblem

Bestimme eine aufsteigende Sortierung x_1, \dots, x_n der Objekte aus S nach ihren Gewichten.

(* Es gilt nun $w(x_1) \leq w(x_2) \leq \dots \leq w(x_n)$. *)

```
L := ∅;  
FOR k := 1, ..., n DO  
  IF L ∪ {x_k} ∈ U THEN  
    L := L ∪ {x_k}  
  FI  
OD
```

Gib L aus Lösung aus.

Beweis. Wir führen den Beweis, indem wir die Dualität von Minimierungs- und Maximierungsproblemen ausnutzen. Zunächst jedoch folgende Vorbemerkung:

Ist $A \in \mathcal{U}$, dann gilt: A ist genau dann \mathcal{U} -maximal, wenn $|A| = \max\{|B| : B \in \mathcal{U}\}$.
(*)

Beweis von (*): Ist $A \in \mathcal{U}$ und $|A| = \max\{|B| : B \in \mathcal{U}\}$, so ist A in keiner anderen Menge von \mathcal{U} enthalten, da diese grösse Kardinalität als A hätte. Ist andererseits A eine \mathcal{U} -maximale Menge und ist $C \in \mathcal{U}$, so dass $|C| = \max\{|B| : B \in \mathcal{U}\}$, so gilt $|A| \leq |C|$. Die Annahme $|A| < |C|$ kann mit der Austauscheigenschaft zum Widerspruch geführt werden, da es dann ein Element $x \in C \setminus A$ geben müsste, so dass $A \cup \{x\} \in \mathcal{U}$. Dies würde der \mathcal{U} -Maximalität von A widersprechen.

Aufgrund von (*) gibt es eine Konstante K , so dass alle \mathcal{U} -maximalen Mengen A die Kardinalität K haben, also $|A| = K$ für alle \mathcal{U} -maximalen Mengen A . Weiter sei

$$w_{\max} = \max_{x \in S} w(x).$$

Wir betrachten nun die Gewichtsfunktion $w' : S \rightarrow \mathbb{R}_{\geq 0}$, die wie folgt definiert ist:

$$w'(x) = w_{\max} - w(x).$$

Für jede Teilmenge A von S ist $w'(A) = |A|w_{\max} - w(A)$. Insbesondere gilt:

$$w'(A) = K \cdot w_{\max} - w(A) \text{ für alle } \mathcal{U}\text{-maximalen Mengen } A \quad (**)$$

Offenbar gilt $w'(x) \geq w'(y)$ genau dann, wenn $w(x) \leq w(y)$. Daher stimmt die durch Algorithmus 62 berechnete Lösungsmenge L für das zu (S, \mathcal{U}, w') gehörende Maximierungsproblem mit der durch Algorithmus 63 berechneten Lösungsmenge für das zu (S, \mathcal{U}, w) gehörende Minimierungsproblem überein. Zu zeigen ist nun, dass L \mathcal{U} -maximal und dass $w(L)$ minimal unter allen \mathcal{U} -maximalen Mengen ist.

Die \mathcal{U} -Maximalität der durch Algorithmus 62 berechneten Lösungsmenge L ist (aufgrund von Eigenschaft (2) von Teilmengensystemen) offensichtlich. Aus Satz 5.4.12 folgt:

$$w'(L) = \max\{w'(B) : B \in \mathcal{U}\}$$

Wegen $w' \geq 0$ (nach Definition von w') gilt $w'(B) \leq w'(C)$, falls $B \subseteq C$. Daher wird das Maximum der Werte $w'(B)$ durch (wenigstens) eine \mathcal{U} -maximale Menge angenommen. Daher gilt:

$$w'(L) = \max\{w'(B) : B \text{ ist } \mathcal{U}\text{-maximal}\} \quad (***)$$

Durch Kombination von $(**)$ und $(***)$ erhalten wir

$$\begin{aligned} w(L) &= Kw_{\max} - w'(L) \\ &= Kw_{\max} - \max\{w'(B) : B \text{ ist } \mathcal{U}\text{-maximal}\} \\ &= Kw_{\max} - \max\{Kw_{\max} - w(B) : B \text{ ist } \mathcal{U}\text{-maximal}\} \\ &= Kw_{\max} - (Kw_{\max} - \min\{w(B) : B \text{ ist } \mathcal{U}\text{-maximal}\}) \\ &= \min\{w(B) : B \text{ ist } \mathcal{U}\text{-maximal}\} \end{aligned}$$

Also ist die durch Algorithmus 63 berechnete Menge L eine Lösung des zu (S, \mathcal{U}, w) gehörenden Minimierungsproblems. \square

Matroide mit positiven und negativen Gewichten. Die vorangegangenen Überlegungen zur Greedy-Methode für Matroide haben sich auf nicht-negative Gewichtsfunktionen bezogen. Auch für Gewichtsfunktionen $w : S \rightarrow \mathbb{R}$, die möglicherweise negative Funktionswerte (also $w(x) < 0$) annehmen, kann die Greedy-Methode (Algorithmus 62 für Maximierungsprobleme und Algorithmus 63 für Minimierungsprobleme) eingesetzt werden, jedoch wird dann jeweils eine \mathcal{U} -maximale Menge $L \in \mathcal{U}$ mit extremem Gewicht berechnet, also

- $w(L) = \max\{w(A) : A \text{ ist } \mathcal{U}\text{-maximal}\}$, falls die Objekte *absteigend* nach ihren Gewichten sortiert werden (Algorithmus 62),
- $w(L) = \min\{w(A) : A \text{ ist } \mathcal{U}\text{-maximal}\}$, falls die Objekte *aufsteigend* nach ihren Gewichten sortiert werden (Algorithmus 63).

5.4.3 Auftragsplanung mit Schlussterminen

Als Beispiel für den Einsatz der Matroid-Theorie betrachten wir folgende Fragestellung des Task Scheduling. Gegeben sind n Aufträge $\mathcal{A}_1, \dots, \mathcal{A}_n$ mit Deadlines $t_1, \dots, t_n \in \mathbb{N}$ und Gewinne $p_1, \dots, p_n \in \mathbb{N}$, wobei $1 \leq t_i \leq n$ für alle i . Intuitiv ist p_i der Gewinn, der durch

die fristgerechte Fertigstellung von Auftrag \mathcal{A}_i erzielt wird. Es wird davon ausgegangen, daß eine Maschine zur Verfügung steht, welche jeden Auftrag in einer Zeiteinheit erledigen kann, wobei die Aufträge nur sequentiell abgearbeitet werden können, aber in beliebiger Reihenfolge. Wir nehmen hier also an, daß keinerlei Prioritäten der Jobs vorliegen, und lassen keine parallele Ausführung mehrerer Jobs zu. Weiter darf jeder Auftrag höchstens einmal ausgeführt werden. Das Ziel ist es nun, einen Arbeitsplan zu erstellen, der – mit Zeitpunkt 0 beginnend – einen Zeitplan zur Ausführung ausgewählter Aufträge erstellt, so daß der erzielte Gewinn maximiert wird.

Wir zeigen nun, daß die Greedy-Strategie „*höchster fristgerechter Gewinn zuerst*“ zur optimalen Lösung führt. Dies ist in Algorithmus 64 formuliert. In der IF-Abfrage ist zu prüfen, ob die Auftragsmenge $U = L \cup \{x_k\}$ fristgerecht abgearbeitet werden kann. Damit ist gemeint, dass es eine Numerierung $\mathcal{A}_{j_1}, \dots, \mathcal{A}_{j_k}$ der Aufträge in U gibt, so dass der Auftragsplan

führen \mathcal{A}_{j_1} zum Zeitpunkt 0 aus,
führen \mathcal{A}_{j_2} zum Zeitpunkt 1 aus,
 \vdots
führen \mathcal{A}_{j_k} zum Zeitpunkt $k - 1$ aus

alle U -Aufträge spätestens zum Zeitpunkt ihres Schlusstermins erledigt. Letzteres ist äquivalent zu $t_{j_1} \geq 1, \dots, t_{j_k} \geq k$.

Lemma 5.4.15 (Charakterisierungen fristgerecht ausführbarer Auftragsmengen). *Sei $U \subseteq \{\mathcal{A}_1, \dots, \mathcal{A}_n\}$. Dann sind folgende Aussagen äquivalent:*

- (a) *Es gibt einen Auftragsplan für U , der alle Aufträge fristgerecht beendet, also eine Numerierung $\mathcal{A}_{j_1}, \dots, \mathcal{A}_{j_k}$ der Aufträge in U , so dass $t_{j_1} \geq 1, \dots, t_{j_k} \geq k$.*
- (b) *Für alle $t \in \{1, \dots, n\}$ gilt $N_t(U) \leq t$, wobei $N_t(U) = |\{\mathcal{A}_i \in U : t_i \leq t\}|$ die Anzahl an Aufträgen in U ist, die bis Zeitpunkt t erledigt sein müssen.*
- (c) *Gilt $U = \{\mathcal{A}_{j_1}, \dots, \mathcal{A}_{j_k}\}$, wobei $\mathcal{A}_{j_1}, \dots, \mathcal{A}_{j_k}$ paarweise verschiedene Aufträge sind und $t_{j_1} \leq \dots \leq t_{j_k}$, so ist $t_{j_1} \geq 1, t_{j_2} \geq 2, \dots, t_{j_k} \geq k$.*

Beweis. (c) \Rightarrow (a): klar.

(a) \Rightarrow (b): folgt aus der Tatsache, daß bis Zeitpunkt t höchstens t Aufträge ausgeführt werden können. Da U termingerecht realisierbar ist (Voraussetzung (a)), können höchstens t Aufträge in U eine Deadline $\leq t$ haben.

(b) \Rightarrow (c): Angenommen es gilt $t_{j_t} < t$ für ein $t \in \{1, \dots, k\}$. Dann gilt

$$t_{j_1} \leq t_{j_2} \leq \dots \leq t_{j_t} \leq t - 1$$

und somit

$$\{\mathcal{A}_i \in U : t_i \leq t - 1\} \supseteq \{\mathcal{A}_{j_1}, \dots, \mathcal{A}_{j_t}\}.$$

Daher ist $N_{t-1}(U) = |\{\mathcal{A}_i \in U : t_i \leq t - 1\}| \geq t$. Widerspruch zu Voraussetzung (b). \square

Die Äquivalenz von (a) und (c) in Lemma 5.4.15 zeigt, dass eine termingerechte Realisierung einer Auftragsmenge U genau dann möglich ist, wenn der auf der Schedulingstrategie „*earliest deadlines first*“ beruhende Auftragsplan eingesetzt werden kann.

Algorithmus 64 Greedy-Algorithmus für das Auftrag-Deadline-Maximierungsproblem

Bestimme eine absteigende Sortierung der Aufträge nach ihren Gewinnen.

(* O.E. gelte $p_1 \geq p_2 \geq \dots \geq p_n$. *)

$L := \emptyset$; (* Lösungsmenge L ist eine Menge von Aufträgen *)

FOR $i = 1, \dots, n$ **DO**

IF es gibt einen Auftragsplan, der alle Aufträge in $L \cup \{\mathcal{A}_i\}$ fristgerecht abarbeitet

THEN

$L := L \cup \{\mathcal{A}_i\}$

FI

OD

Gib L aus Lösung aus.

Laufzeit. Für die Sortierphase können wir die Kosten $\Theta(n \log n)$ veranschlagen. Die Frage, ob $L \cup \{\mathcal{A}_i\}$ einen fristgerechten Auftragsplan besitzt, kann – aufgrund der Aussage von Lemma 5.4.15 – dadurch beantwortet werden, indem L als sortierte Liste (aufsteigend sortiert nach den Schlussterminen) realisiert wird. Für den fraglichen Auftrag \mathcal{A}_i ist (1) die entsprechende Stelle zu suchen, an welcher \mathcal{A}_i einzufügen ist und (2) dann zu prüfen, ob durch die um eine Zeiteinheit verzögerten Aufträge $\mathcal{A}_j \in L$ mit größerem Schlusstermin (also $t_j > t_i$) noch immer fristgerecht beendet werden. Dieser Vorgang erfordert eine sequentielle Suche in der Liste für L und verursacht im schlimmsten Fall die Kosten $\Theta(i)$. Aufsummieren über alle i ergibt quadratische Laufzeit.

Korrektheit. Zum Nachweis der Korrektheit der durch Algorithmus 64 ausgegebenen Lösung bedienen wir uns des in Satz 5.4.12 angegebenen Kriteriums und zeigen, daß (1) die Problemstellung auf das durch einen Matroiden mit Gewichtsfunktion induzierte Maximierungsproblem abgebildet werden kann und (2) Algorithmus 64 als entsprechende Instanz des Greedy-Schemas für Matroide (Algorithmus 62) interpretiert werden kann. Zunächst bilden wir die Problemstellung auf ein Teilmengensystem ab: Als Grundmenge S verwenden wir die Menge aller Aufträge, also $S = \{\mathcal{A}_1, \dots, \mathcal{A}_n\}$. Die Menge $\mathcal{U} \subseteq 2^S$ definieren wir als die Menge aller Auftragsmengen U , die auf der vorliegenden Maschine fristgerecht realisierbar sind, also

$$\mathcal{U} = \{U \subseteq S : U \text{ erfüllt die Eigenschaften (a)-(c) aus Lemma 5.4.15}\}.$$

Wir verwenden die offensichtliche Gewichtsfunktion $w : S \rightarrow \mathbb{N}$, welche jedem Auftrag \mathcal{A}_i den Gewinn $w(\mathcal{A}_i) = p_i$ zuordnet. Es ist nun leicht einsichtig, daß (S, \mathcal{U}) ein Teilmengensystem im Sinn von Definition 5.4.10 und Algorithmus 64 eine Instanz von Algorithmus 62 ist. Die Korrektheit ergibt sich daher aus Satz 5.4.12 auf Seite 261 und folgendem Lemma:

Lemma 5.4.16 (Matroid der Aufträge). *Das wie oben definierte Auftrags-Teilmengensystem (S, \mathcal{U}) hat die Austauscheigenschaft (ist also ein Matroid).*

Beweis. Zum Nachweis der Austauscheigenschaft nehmen wir an, daß W, V Auftragsmengen in \mathcal{U} sind mit $|W| < |V|$. Für $U \in \mathcal{U}$ schreiben wir $N_t(U)$ für die Anzahl an Aufträgen in U mit einem Schlusstermin $\leq t$.

Sei $s \in \{0, 1, \dots, n\}$ der größte Index, so daß $N_s(V) \leq N_s(W)$. Wegen $N_n(V) = |V| > |W| = N_n(W)$ ist $s < n$ und somit

$$N_t(V) > N_t(W) \text{ für alle } s+1 \leq t \leq n. \quad (*)$$

Wegen $N_s(V) \leq N_s(W)$ und $N_{s+1}(V) > N_{s+1}(W)$ gilt:

$$|\{\mathcal{A}_i \in V : t_i = s+1\}| = N_{s+1}(V) - N_s(V) > N_{s+1}(W) - N_s(W) = |\{\mathcal{A}_i \in W : t_i = s+1\}|$$

Also enthält V mehr Aufträge mit der Deadline $s+1$ als W . Sei nun $\mathcal{A}_i \in V \setminus W$ ein Auftrag mit der Deadline $t_i = s+1$. Wir zeigen nun, daß $W \cup \{\mathcal{A}_i\} \in \mathcal{U}$.

Hierzu weisen wir Eigenschaft (b) in Lemma 5.4.15 nach und zeigen, daß $N_t(W \cup \{\mathcal{A}_i\}) \leq t$ für alle t .

- Für $1 \leq t \leq s$ gilt $N_t(W \cup \{\mathcal{A}_i\}) = N_t(W) \leq t$, da $W \in \mathcal{U}$ und $t_i = s+1 > t$.
- Für $s+1 \leq t \leq n$ gilt $N_t(W \cup \{\mathcal{A}_i\}) = N_t(W) + 1 \leq N_t(V) \leq t$, da $V \in \mathcal{U}$ und nach Wahl von s (siehe (*)).

Aus Lemma 5.4.15 folgt $U \in \mathcal{U}$. □

Wir erhalten:

Satz 5.4.17. Mit Algorithmus 64 kann das Auftrags-Schlusstermin-Problem in Zeit $\Theta(n^2)$ gelöst werden.

Weitere Beispiele zur Greedy-Methode und Matroide folgen in Kapitel 6.

5.5 Dynamisches Programmieren

Dynamisches Programmieren ist eine Algorithmenentwurfsstrategie, die auf einer Zerlegung des gegebenen Problems in Teilprobleme und einer Folge von lokalen Entscheidungen, die letztendlich zu einer globalen Entscheidung zusammengebaut werden, beruht. Im Gegensatz zur Greedy-Methode sind die lokalen Entscheidungen nicht notwendig endgültig und werden durch den Zusammenbau der Teillösungen zur globalen Lösung eventuell revidiert.

Die oben genannten Konzepte (das Zerlegen in Teilprobleme sowie das Zusammensetzung der Teillösungen) treten auch bei manchen Divide & Conquer Algorithmen und den Aufzählungsmethoden auf. Wir erläutern die wesentlichen Unterscheidungsmerkmale. Im Gegensatz zu Divide & Conquer Algorithmen, die auf einem Top-Down Ansatz beruhen, liegt dem dynamischen Programmieren eine Bottom-Up Strategie zugrunde, in der die Lösungen für kleinere, im allgemeinen nicht disjunkte Teilprobleme zu einer Lösung für ein größeres Teilproblem zusammengesetzt werden. Oftmals ist es sinnvoll, die Lösungen der Teilprobleme in tabellarischer Form zu speichern, um konstante Zugriffszeiten auf die Teillösungen zu ermöglichen.

Zur Erläuterung der Grundidee des dynamischen Programmierens betrachten wir das Beispiel der *Binomialkoeffizienten* und deren Berechnung anhand der Rekursionsformel

$$\binom{n}{i} = \binom{n-1}{i} + \binom{n-1}{i-1} \quad \text{für } 1 \leq i < n.$$

In Abschnitt 4.1.4 (Seite 146) machten wir die Feststellung, daß bei der naiven rekursiven Berechnung der Binomialkoeffizienten basierend auf der genannten Rekursionsformel viele Rekursionsaufrufe mehrfach stattfinden. Der auf dem dynamischen Programmieren beruhende Algorithmus umgeht die Mehrfachberechnungen mit einem Bottom-Up-Ansatz, welcher die relevanten Werte „ ℓ über j “ für wachsende Differenz $k = \ell - j$ der Reihe nach berechnet. Dies ist in Algorithmus 65 skizziert.

Algorithmus 65 Berechnung der Binomialkoeffizienten (dynamisches Programmieren)

FOR $j = 1, \dots, n$ **DO**
 $bin(j, j) := 1;$
 $bin(j, 0) := 1$
OD
FOR $k = 1, \dots, n - 1$ **DO**
 FOR $j = 1, \dots, \min\{i, n - k\}$ **DO**
 $bin(j + k, j) := bin(j + k - 1, j) + bin(j + k - 1, j - 1)$
 OD
OD
Gib den Wert $bin(n, i)$ aus.

Die Bottom-Up Strategie ist vor allem dann sinnvoll, wenn ein Problem vorliegt, von dem man zwar weiß, daß sich dessen Lösung durch Kombination der Lösungen von gewissen Teilproblemen ergibt, für die jedoch a priori nicht bekannt ist, welche Teilprobleme

letztendlich relevant sind. Eine ähnliche Ausgangssituation liegt für die Backtracking und Branch & Bound Algorithmen vor, für die die Kenntnis, welche Teilprobleme (Fragmente des Aufzählungsbaums) für die Gesamtlösung bedeutsam sind, erst während der Ausführung gewonnen wird. Tatsächlich gibt es eine Reihe von Problemen, die sowohl mit der Methode des dynamischen Programmierens als auch mit Backtracking (oder Branch & Bound) gelöst werden können. Das wesentliche Unterscheidungsmerkmal zwischen dem dynamischen Programmieren und den Aufzählungsmethoden ist die Bottom Up Strategie des dynamischen Programmierens, die im Gegensatz zur Top-Down Analyse des Aufzählungsbaums steht.

Während Algorithmen, die auf den Aufzählungsmethoden beruhen, meist exponentielle worst-case Laufzeit haben, und daher meist nur für „schwierige“ Probleme (für die keine effizientere Algorithmen bekannt sind) angewandt werden, lässt sich keine allgemeine Aussage über die Effizienz von Algorithmen machen, denen das dynamische Programmieren zugrundeliegt. Die Zeit- und Platzkomplexität hängt wesentlich von der Anzahl der zur Diskussion stehenden Teilprobleme ab. In solchen Fällen, in denen sowohl die Aufzählungsmethoden als auch dynamisches Programmieren (sinnvoll) anwendbar sind, ergibt sich häufig kein wesentlicher Unterschied in der Zeitkomplexität. Durch die tabellarische Speicherung der Teillösungen sind die auf dem dynamischen Programmieren beruhenden Algorithmen jedoch oftmals weniger speicherplatzeffizient.

DIVIDE & CONQ.	Greedy-Meth.	Backtracking	dyn. Programmieren
<p>Top-Down Zerlegung in Teilprobleme</p> <p>Laufzeit-Analyse durch Lösen einer Rekurrenz (s. Master-Theorem)</p> <p>häufige Anwendung: Optimierungsprobleme</p> <ul style="list-style-type: none"> * exakte Lsg. für spez. Problemklassen * Näherungslsg. 	<p>Folge lokaler endgültiger Entscheidungen</p> <p>meist sehr effizient</p>	<p>Top-Down Konstruktion des Aufz. baums</p> <p>meist expon. Laufzeit</p> <p>Varianten: Branch & Bound α-β-Pruning für Spielbäume</p>	<p>Bottom-Up Konstruktion der Lösungen für die Teilprobleme (tabellarische Darstellung)</p> <p>Vorauss. für Opt.-probleme: Optimalitätsprinzip</p>

5.5.1 Erstellung regulärer Ausdrücke für endliche Automaten

Wir beginnen mit zwei Beispielen der dynamischen Programmierung aus dem Bereich formaler Sprachen.

Wir betrachten zunächst das nachfolgende Beispiel. Gegeben ist ein deterministischer endlicher Automat \mathcal{M} . Gesucht ist ein regulärer Ausdruck α , welcher die von \mathcal{M} akzeptierte Sprache beschreibt.

Zur Erinnerung: endliche Automaten und reguläre Ausdrücke. Ein nicht-deterministischer endlicher Automat $\mathcal{M} = (Q, \Sigma, \delta, Q_0, F)$ besteht aus einem endlichen Zustandsraum Q , einer Anfangszustandsmenge $Q_0 \subseteq Q$, einer Endzustandsmenge $F \subseteq Q$, einem Alphabet Σ sowie einer Übergangsfunktion

$$\delta : Q \times \Sigma \rightarrow 2^Q.$$

Die akzeptierte Sprache $\mathcal{L}(\mathcal{M})$ besteht allen endlichen Wörtern $w = a_1 a_2 \dots a_m$, die einen initialen, akzeptierenden Lauf in \mathcal{M} haben. Dabei versteht man unter einem *Lauf* für w eine Zustandsfolge $\pi = p_0, p_1, \dots, p_m$, so daß $p_i \in \delta(p_{i-1}, a_i)$, $i = 1, \dots, m$. π heißt akzeptierend, falls $p_n \in F$, und initial, falls $p_0 \in Q_0$.

Reguläre Ausdrücke über dem Alphabet Σ (hier mit griechischen Buchstaben α, β, γ bezeichnet) sind aus den Atomen

- \emptyset als Ausdruck für die leere Sprache,
- ε als Ausdruck für die einelementige Sprache $\{\varepsilon\}$ bestehend aus dem leeren Wort ε ,
- den Symbolen $a \in \Sigma$ als Ausdruck für die einelementige Sprache $\{a\}$ bestehend aus dem Terminalzeichen a aufgefasst als Wort der Länge 1,

sowie den Verknüpfungsoperatoren Kleene-Abschluß α^* , Konkatenation $\alpha_1 \circ \alpha_2$ und Vereinigung $\alpha_1 + \alpha_2$ gebildet.⁴³ Ist $A = \{\alpha_1, \dots, \alpha_m\}$ eine endliche Menge regulärer Ausdrücke, so verwenden wir die Summenschreibweise

$$\sum_{\alpha \in A} \alpha$$

anstelle von $\alpha_1 + \dots + \alpha_m$ (in irgendeiner Klammerung). Ferner verzichten wir auf das Konkatenationssymbol \circ und schreiben kurz $\alpha\beta$ statt $\alpha \circ \beta$.

$\mathcal{L}(\alpha)$ bezeichnet die durch α dargestellte Sprache, also $\mathcal{L}(\alpha) \subseteq \Sigma^*$. Ist z.B. $\alpha_{Buchstabe} = a + b + \dots + z$ und $\alpha_{Ziffer} = 0 + 1 + \dots + 9$, so beschreibt

$$\alpha_{Buchstabe} (\alpha_{Ziffer} + \alpha_{Buchstabe})^*$$

die Menge aller Wörter, die mit einem Kleinbuchstaben beginnen und danach ein beliebig langes (eventuell leeres) Wort gebildet aus Kleinbuchstaben und Ziffern folgt. Ferner bezeichnet \equiv die semantische Äquivalenz regulärer Ausdrücke, also $\alpha_1 \equiv \alpha_2$ genau dann, wenn $\mathcal{L}(\alpha_1) = \mathcal{L}(\alpha_2)$.

⁴³In der Literatur sind die Schreibweisen nicht einheitlich. Manche Autoren verwenden das Symbol „ \cup “ oder „ $|$ “ anstelle von „ $+$ “. Manchmal wird auch ein Semikolon „;“ oder ein Punkt „.“ für das Konkatenationssymbol verwendet.

Konstruktion eines regulären Ausdrucks für gegebenen Automaten. Endliche Automaten und reguläre Ausdrücke sind äquivalente Formalismen für reguläre Sprachen. Wir betrachten hier die Fragestellung, bei der ein endlicher Automat \mathcal{M} gegeben und ein regulärer Ausdruck α mit $\mathcal{L}(\alpha) = \mathcal{L}(\mathcal{M})$ gesucht ist.

Zunächst können wir einige Vereinfachungen an dem Automaten vornehmen. Ist z.B. $F = \emptyset$, so ist $\mathcal{L}(\mathcal{M}) = \emptyset$ und wir können den regulären Ausdruck \emptyset verwenden. Weiter können alle Zustände q eliminiert werden, die keinen Endzustand erreichen können. Diese Vereinfachung des Automaten kann vorgenommen werden, indem eine Tiefen- oder Breiten-Rückwärtssuche mit den F -Zuständen gestartet wird. Mit Rückwärtssuche ist hier gemeint, daß wir die Kanten umdrehen, d.h. den Graphen $G^{-1} = (Q, E^{-1})$ betrachten, wobei

$$E^{-1} = \{(q, p) : (p, q) \in E\} = \{(q, p) : q = \delta(p, a) \text{ für ein } a \in \Sigma\}.$$

Im Folgenden können wir also davon ausgehen, daß die Endzustandsmenge F von jedem Zustand erreichbar ist. Das Verfahren der dynamischen Programmierung wird eingesetzt, um reguläre Ausdrücke $\alpha[q, p]$ zu konstruieren, welche die Sprache

$$L_{q,p} = \{w \in \Sigma^* : \text{es gibt einen Lauf für } w \text{ von } q \text{ nach } p \text{ in } \mathcal{M}\}$$

erzeugen. Liegen die Ausdrücke $\alpha[q, p]$ vor, so ergibt sich der gesuchte Ausdruck α durch

$$\alpha = \sum_{q \in Q_0} \sum_{p \in F} \alpha[q, p].$$

Dies erklärt sich daraus, dass jeder initiale, akzeptierende Lauf in einem Anfangszustand $q \in Q_0$ beginnt und in einem der Endzustände $p \in F$ endet.

Berechnung der Ausdrücke $\alpha[q, p]$ mit dynamischer Programmierung. Es bleibt zu klären, wie die Ausdrücke $\alpha[q, p]$ berechnet werden. Hierzu verwenden wir eine beliebige, aber feste Numerierung der Zustände in \mathcal{M} , etwa

$$Q = \{q_1, \dots, q_n\}.$$

Zur Vereinfachung schreiben wir $\alpha[i, j]$ anstelle von $\alpha[q_i, q_j]$. Die Berechnung der $\alpha[i, j]$'s erfolgt nun iterativ durch die Konstruktion regulärer Ausdrücke $\alpha^k[i, j]$ für $k = 0, 1, 2, \dots, n$ mit folgender Eigenschaft:

$$\mathcal{L}(\alpha^k[i, j]) = \left\{ \begin{array}{l} \text{Menge aller } a_1 \dots a_m \in \Sigma^*, \text{ für die} \\ \text{es einen Lauf } p_0, p_1, \dots, p_m \text{ mit} \\ \text{folgenden Eigenschaften gibt:} \\ \quad (1) p_0 = q_i \\ \quad (2) p_m = q_j \\ \quad (3) p_1, \dots, p_{m-1} \in \{q_1, \dots, q_k\} \end{array} \right.$$

Sind diese Ausdrücke $\alpha^k[i, j]$, $k = 0, 1, \dots, n$, generiert, so können wir $\alpha[i, j] = \alpha^n[i, j]$ setzen. Man beachte, daß für $k = n$ die oben genannte Eigenschaft (3) irrelevant ist, da sie Läufe über beliebige Zustände zuläßt.

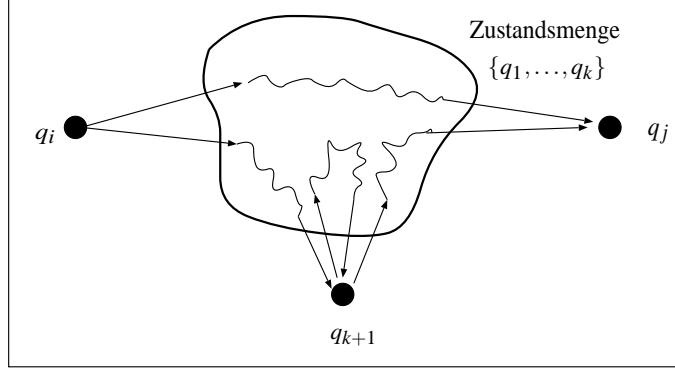


Abbildung 53: Zulässige Läufe für die regulären Ausdrücke $\alpha^{k+1}[i, j]$

Die Berechnung der Ausdrücke $\alpha^k[i, j]$ erfolgt durch dynamisches Programmieren (siehe Algorithmus 66 auf Seite 273). Für den Fall $k = 0$ sind nur Läufe der Länge 0 oder 1 zugelassen. Dies erklärt die angegebenen Ausdrücke

$$\alpha^0[i, j] = a_{\ell_1} + \dots + a_{\ell_r}, \text{ falls } \{a \in \Sigma : q_j \in \delta(q_i, a)\} = \{a_{\ell_1}, \dots, a_{\ell_r}\} \text{ und } i \neq j.$$

Für $\alpha^0[i, i]$ ist zusätzlich das leere Wort zu berücksichtigen, da es stets den Lauf der Länge 0 von q_i nach q_i gibt. Dabei ist

$$\sum_{\substack{a \in \Sigma \\ q_j \in \delta(q_i, a)}} a = \emptyset,$$

falls es kein a mit $q_j \in \delta(q_i, a)$ gibt. Für den Fall $i = j$ und $\{a : q_i \in \delta(q_i, a)\} = \emptyset$ kann $\alpha^0[i, i] = \varepsilon$ anstelle von $\alpha^0[i, i] = \emptyset + \varepsilon$ verwendet werden. Die Definition von $\alpha^{k+1}[i, j]$ ergibt sich aus der Beobachtung, daß die in $\alpha^{k+1}[i, j]$ zugelassenen Läufe *entweder* nicht durch Zustand q_{k+1} führen (Teilausdruck $\alpha^k[i, j]$) *oder* von der Form

$$\underbrace{q_i \xrightarrow[a]{\leq k} q_{k+1}}_{\alpha^k[i, k+1]} \underbrace{\xrightarrow[a \in \Sigma]{\leq k} q_{k+1} \xrightarrow[a \in \Sigma]{\leq k} \dots \xrightarrow[a \in \Sigma]{\leq k} q_{k+1}}_{(\alpha^k[k+1, k+1])^*} \underbrace{q_{k+1} \xrightarrow[a]{\leq k} q_j}_{\alpha^k[k+1, j]}$$

sind, wobei durch $\xrightarrow[a]{\leq k}$ angedeutet wird, daß die durchlaufenen Zustände in $\{q_1, \dots, q_k\}$ liegen. Letzteres wird durch den Teilausdruck $\alpha^k[i, k+1] (\alpha^k[k+1, k+1])^* \alpha^k[k+1, j]$ formalisiert.

Algorithmus 66 Berechnung der regulären Ausdrücke $\alpha^k[i, j]$

```
FOR  $i = 1, \dots, n$  DO
  FOR  $j = 1, \dots, n$  DO
    IF  $j \neq i$  THEN
       $\alpha^0[i, j] := \sum_{\substack{a \in \Sigma \\ q_j \in \delta(q_i, a)}} a$ 
    ELSE
       $\alpha^0[i, i] := \varepsilon + \sum_{\substack{a \in \Sigma \\ q_i \in \delta(q_i, a)}} a$ 
    FI
    OD
  OD
FOR  $k = 0, 1, \dots, n - 1$  DO
  FOR  $i = 1, \dots, n$  DO
    FOR  $j = 1, \dots, n$  DO
       $\alpha^{k+1}[i, j] := \alpha^k[i, j] + \alpha^k[i, k+1] (\alpha^k[k+1, k+1])^* \alpha^k[k+1, j]$ 
    OD
  OD
OD
```

Auf den ersten Blick könnte man meinen, daß die Laufzeit kubisch ist. Dies ist jedoch ein Trugschluss, da das Hinschreiben der Ausdrücke aufwendig ist und nicht mit einer Zeiteinheit bewertet werden kann. Stattdessen müssen wir die Länge der entstehenden Ausdrücke berücksichtigen. Für die maximale Länge der Ausdrücke $\alpha^k[i, j]$ können wir die Rekurrenz $T(k + 1) = 4T(k) + 4$ und $T(0) = \Theta(|\Sigma|)$ verwenden. Betrachtet man das Eingabealphabet Σ als fest, so ist $|\Sigma| = \Theta(1)$ und somit $T(k) = \Theta(4^k)$. Aufsummieren über alle k und Knotenindizes i und j liefert exponentielle worst-case Laufzeit

$$\sum_{k=1}^n \sum_{i=1}^n \sum_{j=1}^n 4^k = n^2 \cdot \left(\frac{4^{n+1} - 1}{4 - 1} - 1 \right) = \Theta(n^2 \cdot 4^n).$$

Dasselbe gilt für den Platzbedarf, wobei hier zu beachten ist, daß zur Berechnung der Ausdrücke $\alpha^{k+1}[\dots]$ nur die Ausdrücke $\alpha^k[\dots]$ benötigt werden. Es genügen daher zwei $(n \times n)$ -Matrizen zur Verwaltung der berechneten regulären Ausdrücke.

Die Rutten-Methode. Eine effiziente Vorgehensweise zur Erstellung regulärer Ausdrücke für endliche Automaten beruht auf der Idee, jedem Zustand q einen regulären Ausdruck α_q zuzuordnen, welcher für die Sprache aller Wörter $x \in \Sigma^*$ steht, die einen in Zustand q beginnenden, akzeptierenden Lauf im Automaten haben. Der gesuchte Ausdruck für den Automaten ist dann die Summe der Ausdrücke α_q für alle Anfangszustände.

Zunächst kann man ein Gleichungssystem für die α_q 's hinschreiben, das sich sofort aus der Übergangsrelation δ ergibt:

$$\alpha_q \equiv \sum_{p \in Q} \sum_{\substack{a \in \Sigma \\ p \in \delta(q, a)}} a \alpha_p$$

Ist q ein Endzustand, so ist der angegebene Ausdruck um „ $+\varepsilon$ “ zu erweitern. Das Gleichungssystem für die α_q 's kann nun gelöst werden, indem man sukzessive die beiden Äquivalenzgesetze

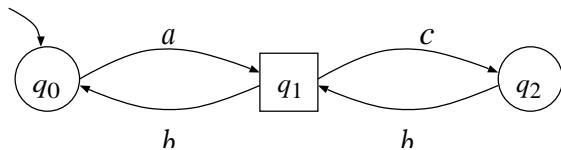
$$\beta_1 \gamma + \dots + \beta_k \gamma \equiv (\beta_1 + \dots + \beta_k) \gamma \quad \text{und} \quad \gamma \beta_1 + \dots + \gamma \beta_k \equiv \gamma (\beta_1 + \dots + \beta_k)$$

und Äquivalenzgesetze für ε und \emptyset sowie die Regel

$$\text{„Aus } \alpha \equiv \beta \alpha + \gamma \text{ und } \varepsilon \notin \mathcal{L}(\beta) \text{ folgt } \alpha \equiv \beta^* \gamma.“}$$

anwendet. Man beachte den Sonderfall $\gamma \equiv \emptyset$, für den sich die angegebene Regel durch „Aus $\alpha \equiv \beta \alpha$ und $\varepsilon \notin \mathcal{L}(\beta)$ folgt $\alpha \equiv \emptyset$ “ ersetzen lässt (da $\beta^* \emptyset \equiv \emptyset$).

Wir verzichten hier auf eine Ausformulierung des Verfahrens als Algorithmus und geben stattdessen nur ein einfaches Beispiel.



Zunächst erstellen wir ein Gleichungssystem:

$$\begin{aligned}\alpha_0 &\equiv a\alpha_1 \\ \alpha_1 &\equiv b\alpha_0 + c\alpha_2 + \varepsilon \\ \alpha_2 &\equiv b\alpha_1\end{aligned}$$

Wir setzen nun die Gleichung für α_0 in die für α_1 ein und erhalten:

$$\alpha_1 \equiv \underbrace{ba}_{\beta} \alpha_1 + \underbrace{c\alpha_2 + \varepsilon}_{\gamma}$$

Anwenden der oben genannten Regel ergibt:

$$\alpha_1 \equiv (ba)^*(c\alpha_2 + \varepsilon)$$

Einsetzen in die Gleichung für α_2 liefert:

$$\alpha_2 \equiv b(ba)^*(c\alpha_2 + \varepsilon) \equiv b(ba)^*c\alpha_2 + b(ba)^*$$

Durch Anwenden der genannten Regel mit $\beta = b(ba)^*c$ und $\gamma = b(ba)^*$ erhalten wir:

$$\alpha_2 \equiv (b(ba)^*c)^* b(ba)^*$$

Einsetzen in die Gleichung für α_0 liefert einen gesuchten regulären Ausdruck.
Selbstverständlich gibt es hier für das Einsetzen viele Möglichkeiten, die letztendlich zu verschiedenen (aber äquivalenten) Ausdrücken führen können.

5.5.2 Das Wortproblem für kontextfreie Sprachen

In diesem Abschnitt stellen wir einen auf dem dynamischen Programmieren beruhenden Algorithmus vor, welcher für eine gegebene kontextfreie Grammatik \mathcal{G} und gegebenes Wort w entscheidet, ob w durch \mathcal{G} generiert werden kann. Wir erinnern zunächst kurz an die benötigten Grundbegriffe.

Kontextfreie Grammatiken. Eine kontextfreie Grammatik \mathcal{G} besteht aus einer endlichen Menge V von Nichtterminalen (auch Variablen genannt), einer endlichen Menge Σ von Terminalzeichen, einem Startsymbol $S \in V$ sowie einer endlichen Menge \mathcal{P} von Regeln (auch Produktionen genannt). Im Folgenden verwenden wir Großbuchstaben S, A, B, C, \dots für die Nichtterminalen. Kleinbuchstaben am Anfang des Alphabets (a, b, c, \dots) stehen für Terminalzeichen. Wörter bezeichnen wir mit Kleinbuchstaben am Ende des Alphabets (etwa u, v, w, x, y, z).

Formal sind die Regeln Paare (A, w) bestehend aus einem Nichtterminal A und einem Wort $w \in (V \cup \Sigma)^*$. Üblich ist die Schreibweise $A \rightarrow w$ statt $(A, w) \in \mathcal{P}$ sowie

$$A \rightarrow w_1 \mid w_2 \mid \dots \mid w_m,$$

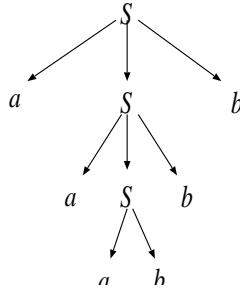
um anzudeuten, daß $(A, w_1), \dots, (A, w_m)$ die in der Grammatik vorkommenden Regeln mit dem Nichtterminal A auf der linken Seite sind. Intuitiv bedeutet diese Schreibweise, daß das Nichtterminal A in beliebigem Kontext (daher der Name „kontextfrei“) durch eines der Wörter w_1, \dots, w_m ersetzt werden kann. Für die Ersetzung eines Nichtterminal in einem Wort ist eine Doppelpfeil-Schreibweise üblich: $uAv \Rightarrow uwv$, falls $A \rightarrow w$. Dabei sind u, v beliebige Wörter über $V \cup \Sigma$. \Rightarrow^* steht für die Ableitungsrelation, die sich durch Hintereinanderhängen von Einzelschritt-Ableitungen mittels \Rightarrow ergibt. Also

$$u \Rightarrow^* w$$

genau dann, wenn es eine Folge u_0, u_1, \dots, u_n mit $n \geq 0$, $u_0 = u$, $u_n = w$ und $u_{i-1} \Rightarrow u_i$, $i = 1, \dots, n$, gibt. Für den Sonderfall $n = 0$ ergibt sich $u \Rightarrow^* u$ für alle $u \in (V \cup \Sigma)^*$. Ableitungen können durch so genannte Ableitungsbäume dargestellt werden. Deren innere Knoten stehen für die Nichtterminalen, die in einem Wort der Ableitung vorkommen. Die Söhne eines inneren Knotens entsprechen den angewandten Regeln. Z.B. kann die folgende Ableitung

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$$

für $aaabbb$ aus der Grammatik mit dem Startsymbol S , dem Alphabet $\Sigma = \{a, b\}$ und den zwei Regeln $S \rightarrow ab \mid aSb$ durch folgenden Ableitungsbau dargestellt werden:



Die durch eine Grammatik \mathcal{G} erzeugte Sprache ist

$$\mathcal{L}(\mathcal{G}) = \{w \in \Sigma^* : S \Rightarrow^* w\},$$

also die Menge aller Wörter über dem Terminalalphabet Σ , welche aus dem Startsymbol herleitbar sind. Z.B. ist die durch die oben erwähnte Grammatik mit $V = \{S\}$, $\Sigma = \{a, b\}$ und den zwei Regeln $S \rightarrow ab|aSb$ erzeugte Sprache gleich $\{a^n b^n : n \geq 1\}$. Eine kontextfreie Grammatik für korrekt geklammerte arithmetische Ausdrücke ist durch $V = \{S, A, B\}$, $\Sigma = \{a, +, *, (,)\}$ und dem Produktionssystem

$$S \rightarrow A \mid S + A \quad A \rightarrow B \mid A * B \quad B \rightarrow a \mid (S)$$

gegeben. Das Terminalzeichen a steht hier stellvertretend für Variablen oder Konstanten. Die durch \mathcal{G} erzeugte Sprache ist die Menge aller geklammerten arithmetischen Ausdrücke gebildet aus den zweistelligen Operatoren $+$ und $*$ („modulo“ der Assoziativ- und Kommutativgesetze für $+$ und $*$), mit der redundante Klammern eingespart werden können. Z.B. gilt:

$$S \Rightarrow A \Rightarrow A * B \Rightarrow A * (S) \Rightarrow A * (S + A) \Rightarrow \dots \Rightarrow a * (a + a)$$

Chomsky Normalform. Zur Vereinfachung nehmen wir im Folgenden an, daß die vorliegende Grammatik weder ε -Regeln noch Kettenregeln enthält, also daß alle Regeln die Form $A \rightarrow w$ mit $|w| \geq 2$ oder $A \rightarrow a$ für ein Terminalsymbol $a \in \Sigma$ haben. Zu jeder solchen kontextfreien Grammatik kann sehr leicht eine äquivalente Grammatik in Chomsky Normalform (kurz CNF) konstruiert werden. Damit ist eine kontextfreie Grammatik gemeint, in der alle Regeln die Form $A \rightarrow a$ für ein $a \in \Sigma$ oder $A \rightarrow BC$ für Nichtterminale B, C haben. Z.B. ist die kontextfreie Grammatik mit den Regeln

$$S \rightarrow AC \mid AB, \quad C \rightarrow SC \mid c, \quad A \rightarrow a, \quad B \rightarrow b$$

in CNF; nicht aber die Grammatik mit den Regeln $S \rightarrow aSb \mid ab$. Offenbar sind die Ableitungsbäume von CNF-Grammatiken stets Binärbäume.

Der Algorithmus zur Erstellung einer äquivalenten CNF-Grammatik für gegebene kontextfreie Grammatik \mathcal{G} führt folgende Schritte aus. (Beachte, wir setzen hier voraus, daß \mathcal{G} weder ε -Regeln $A \rightarrow \varepsilon$ noch Kettenregeln $A \rightarrow B$ enthält.)

1. Für jedes Terminalzeichen a füge ein neues Nichtterminal A_a und die Regel $A_a \rightarrow a$ ein.

2. Für jede Regel $A \rightarrow w$ mit $|w| \geq 2$ ersetze jedes Vorkommen eines Terminalzeichen a durch das Nichtterminal A_a .
3. Ersetze jede Regel $A \rightarrow B_1B_2 \dots B_k$ mit $k \geq 3$ durch die Regeln

$$\begin{array}{rcl} A & \rightarrow & B_1C_1 \\ C_1 & \rightarrow & B_2C_2 \\ C_2 & \rightarrow & B_3C_3 \\ & \vdots & \\ C_{k-3} & \rightarrow & B_{k-2}C_{k-2} \\ C_{k-2} & \rightarrow & B_{k-1}B_k \end{array}$$

wobei C_1, \dots, C_{k-2} paarweise verscheidene neue Nichtterminale sind.

Die so generierte Grammatik \mathcal{G}' ist nun in CNF. Die Größe $\text{size}(\mathcal{G}')$ von \mathcal{G}' gemessen an der Anzahl an Nichtterminalen und Gesamtlänge aller Regeln ist linear in der Größe von \mathcal{G} . Beachte, daß im dritten Schritt eine Regel der Länge $k + 1$ durch Regeln der Gesamtlänge $3(k - 1)$ ersetzt werden. Die Anzahl an zusätzlichen Nichtterminalen ist durch $\mathcal{O}(\text{size}(\mathcal{G}))$ beschränkt.

Beispiel 5.5.1 (Erstellung einer CNF). Die beschriebene Transformation arbeitet für die Grammatik mit den Regeln

$$S \rightarrow aSb \quad \text{und} \quad S \rightarrow ab$$

wie folgt. Im ersten Schritt werden neue Nichtterminale $A_a = A$ und $A_b = B$ mit den Regeln $A \rightarrow a$ und $B \rightarrow b$ hinzugefügt. Anschließend werden sämtliche Vorkommen von a bzw. b auf der rechten Seite einer Regel durch A bzw. B ersetzt. Wir erhalten das Produktionssystem

$$S \rightarrow ASB \mid AB, \quad A \rightarrow a, \quad B \rightarrow b$$

Im letzten Schritt wird die Regel $S \rightarrow ASB$ durch die beiden Regeln $S \rightarrow AC$ und $C \rightarrow SB$ ersetzt. Für die Regel $S \rightarrow AB$ ist nichts zu tun, da diese bereits die geforderte Form hat. \square

Der Cocke-Younger-Kasami Algorithmus. Um das Wortproblem für kontextfreie Sprachen zu lösen, gehen wir von einer Darstellung der Sprache durch eine CNF-Grammatik $\mathcal{G} = (V, \Sigma, \mathcal{P}, S)$ sowie einem Eingabewort $w = a_1a_2 \dots a_n \in \Sigma^*$ aus. Für den Test, ob $w \in \mathcal{L}(\mathcal{G})$ gilt, wenden wir die Methode des dynamischen Programmierens zur Bestimmung der Variablenmengen

$$V[i, j] = \{A \in V : A \Rightarrow^* w_{i,j}\}$$

an. Dabei ist

$$w_{i,j} = a_i a_{i+1} \dots a_{i+j-1}$$

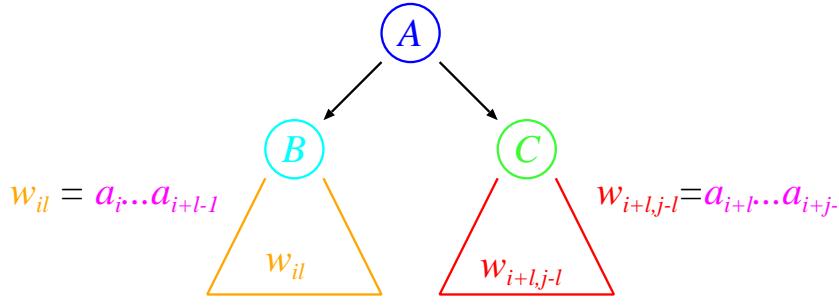


Abbildung 54:

dasjenige Teilwort der Länge j von w , welches mit dem i -ten Symbol beginnt. Dabei ist vorauszusetzen, daß $1 \leq i \leq n$ und $1 \leq j \leq n + 1 - i$. Offenbar liegt w genau dann in $\mathcal{L}(\mathcal{G})$, wenn $S \in V[1, n]$. Da \mathcal{G} in CNF ist, gilt

$$V[i, 1] = \{A \in V : A \rightarrow a_i\}.$$

Für $j \geq 2$ und $A \in V$ gilt $A \Rightarrow^* w_{i,j}$ genau dann, wenn es eine Regel $A \rightarrow BC$ und einen Index $l \in \{1, \dots, j-1\}$ gibt, so daß

$$B \Rightarrow^* w_{i,l} = a_i \dots a_{i+l-1} \text{ und } C \Rightarrow^* w_{i+l,j-l} = a_{i+l} \dots a_{i+j-1}.$$

Abbildung 54 illustriert diese Aussage mit Hilfe des Ableitungsbaums. Daher haben wir folgende rekursive Charakterisierung der Mengen $V[i, j]$.

$$V[i, j] = \bigcup_{l=1}^{j-1} \{A \in V : \exists A \rightarrow BC \text{ mit } B \in V[i, l] \wedge C \in V[i+l, j-l]\}$$

Beachte, dass l für die Länge des aus B hergeleiteten Präfix $a_i \dots a_{i+l-1}$ steht. Also läuft der Index l von 1 bis $j-1$.

Diese Beobachtung wird im Algorithmus von Cocke-Younger-Kasami (kurz CYK-Algorithmus genannt) angewandt, siehe Algorithmus 67 auf Seite 280. Dieser berechnet in Bottom-Up-Manier die Zeilen der folgenden Tabelle:

$V[1, n]$						
$V[1, n-1]$	$V[2, n-1]$					
$V[1, n-2]$	$V[2, n-2]$	$V[3, n-2]$				
⋮						
$V[1, 2]$	$V[2, 2]$	$V[3, 2]$	$V[4, 2]$...	$V[n-1, 2]$	
$V[1, 1]$	$V[2, 1]$	$V[3, 1]$	$V[4, 1]$...	$V[n-1, 1]$	$V[n, 1]$
a_1	a_2	a_3	a_4	...	a_{n-1}	a_n

Algorithmus 67 Der CYK-Algorithmus

(* Das Eingabewort sei $w = a_1 a_2 \dots a_n$.*)

```
FOR  $i = 1, \dots, n$  DO
     $V[i, 1] := \{A \in V : A \rightarrow a_i\}$ ;
OD
FOR  $j = 2, \dots, n$  DO
    FOR  $i = 1, \dots, n + 1 - j$  DO
         $V[i, j] := \emptyset$ ;
        FOR  $l = 1, \dots, j - 1$  DO
             $V[i, j] := V[i, j] \cup \{A \in V : \exists A \rightarrow BC \text{ mit } B \in V[i, l] \wedge C \in V[i + l, j - l]\}$ ;
        OD
    OD
IF  $S \in V[1, n]$  THEN
    Return „JA.  $w \in \mathcal{L}(G)$ .“
ELSE
    Return „NEIN.  $w \notin \mathcal{L}(G)$ .“
FI
```

Es ist offensichtlich, daß die Laufzeit des CYK-Algorithmus kubisch in der Länge des Eingabeworts und der Platzbedarf quadratisch ist, wenn die Grammatik als fest angesehen wird. Daher erhalten wir folgenden Satz:

Satz 5.5.2 (Kosten des CYK-Algorithmus). Das Wortproblem für kontextfreie Sprachen, dargestellt durch CNF-Grammatiken, läßt sich mit dem CYK-Algorithmus in Zeit $\mathcal{O}(n^3)$ und Platz $\mathcal{O}(n^2)$ lösen. Dabei ist n die Länge des Eingabeworts.

Beispiel 5.5.3. [CYK-Algorithmus] Die Arbeitsweise des CYK-Algorithmus für die Sprache $L = \{a^n b^n : n \geq 1\}$, dargestellt durch die CNF-Grammatik

$$S \rightarrow AC \mid AB \quad C \rightarrow SB \quad A \rightarrow a \quad B \rightarrow b,$$

und das Wort $w = aabb$ ist wie folgt. Im Initialisierungsschritt wird $V[1, 1] = V[2, 1] = \{A\}$ und $V[3, 1] = V[4, 1] = \{B\}$ gesetzt. Daraus ergibt sich

$$V[1, 2] = V[3, 2] = \emptyset \text{ und } V[2, 2] = \{S\}.$$

Im zweiten Schleifendurchlauf erhalten wir $V[1, 3] = \emptyset$ und $V[2, 3] = \{C\}$, da $S \in V[2, 2]$, $B \in V[3, 1]$ und $C \rightarrow SB$. Wegen $A \in V[1, 1]$ und $C \in V[2, 3]$ ergibt sich $V[1, 4] = \{S\}$. Die oben erwähnte Tabelle hat also die Gestalt:

{S}			
-- {C}			
-- {S}		--	
{A}	{A}	{B}	{B}
a	a	b	b

Der Algorithmus terminiert also mit der Antwort „JA“. □

5.5.3 Optimalitätsprinzip

Ein weites Anwendungsfeld des dynamischen Programmierens sind Optimierungsprobleme, die sich nicht mit der Greedy-Methode lösen lassen. Die Voraussetzung für die Anwendbarkeit des dynamischen Programmierens auf Optimierungsprobleme ist das *Optimalitätsprinzip*. Hierunter versteht man die Eigenschaft, daß sich die (oder eine) optimale Lösung für die Eingabeobjekte x_1, \dots, x_n durch geeignete Kombination der optimalen Lösungen für gewisse Teilprobleme ergibt. Wir erläutern die prinzipielle Vorgehensweise unter der Annahme, daß die optimalen Lösungen der beiden Teilprobleme mit den Eingabeobjekten x_1, \dots, x_{n-1} bzw. x_2, \dots, x_n , zu einer optimalen Lösung für die Eingabeobjekte x_1, \dots, x_n kombiniert werden können, etwa

$$OptLsg(x_1, \dots, x_n) = Comb(OptLsg(x_1, \dots, x_{n-1}), OptLsg(x_2, \dots, x_n)).$$

Dabei steht $Comb(\dots)$ für eine Funktion, welche die Teillösungen in geeigneter Art und Weise kombiniert. Die naive Ausnutzung der Rekursionsformel und Übersetzung in einen rekursiven Algorithmus führt zur Laufzeit

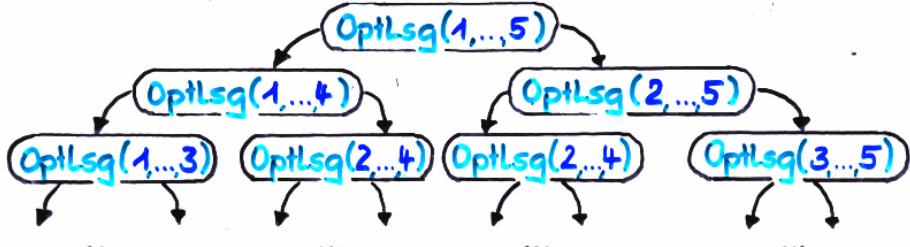
$$\begin{aligned} T(n) &= 2 \cdot T(n-1) + C(n) \\ &\geq 2 \cdot T(n-1) \\ &\geq 4 \cdot T(n-2) \\ &\quad \vdots \\ &\geq 2^n \cdot T(1). \end{aligned}$$

In obiger Abschätzung bezeichnet $C(n)$ die Zerlegungs-/Kombinationskosten. Insgesamt erhalten wir also mindestens exponentielle Laufzeit. Diese lässt sich zwar nicht immer vermeiden, aber in diesem Fall ist die naive Vorgehensweise unsinnig, da einige Rekursionsaufrufe wiederholt stattfinden.

"Naive Rekursion"

$$\text{OptLsg}(x_1, \dots, x_n) = \text{Comb}(\text{OptLsg}(x_1, \dots, x_{n-1}), \text{OptLsg}(x_2, \dots, x_n))$$

Z.B. $x_i = i, n = 5$



Viele Rekursionsaufüfe finden mehrfach statt!

Tatsächlich werden nur $n(n + 1)/2$ Teilprobleme benötigt. Nämlich die Teilprobleme mit den Eingabeobjekten x_i, x_{i+1}, \dots, x_j , $1 \leq i \leq j \leq n$. Dynamisches Programmieren nutzt diese Beobachtung aus und arbeitet für obiges Beispielschema nach dem Prinzip:

```

FOR i = 1, ..., n DO
  berechne OptLsg(x_i)
OD
FOR k = 1, ..., n - 1 DO
  (* Berechne die Lösungen für k + 1 Objekte *)
  FOR i = 1, ..., n - k DO
    L1 := OptLsg(xi, ..., xi+k-1);
    L2 := OptLsg(xi+1, ..., xi+k);
    OptLsg(xi, ..., xi+k) := Comb(L1, L2);
  OD
OD
  
```

Eine analoge Vorgehensweise ist für andere Zerlegungen möglich, z.B. wenn sich die optimale Lösung durch Kombination der optimalen Lösungen von mehr als zwei Teilproblemen ergibt.

5.5.4 Optimale Suchbäume

Unser erstes Beispiel zum Einsatz des dynamischen Programmierens für Optimierungsprobleme beschäftigt sich mit Suchbaumdarstellungen von statischen Mengen, also Mengen, für die nur die Suche unterstützt werden soll, aber keine oder kaum Einfüge- oder

Löschoperationen vorgenommen werden. Beispielsweise benötigt der Parser einer höheren Programmiersprache schnellen Zugang zu allen Schlüsselwörtern der betreffenden Programmiersprache. Diese können als statisch angesehen werden.

Legt man die betreffenden Datensätze sortiert nach ihren Schlüsselwerten in einen AVL-Baum ab, so ist die Zugriffszeit – gemessen an der Anzahl an Vergleichen des gesuchten Worts mit den Schlüsselwörtern im Baum – logarithmisch beschränkt. Dies bezieht sich auf alle Schlüsselwerte, unabhängig davon, wie oft nach ihnen gesucht wird. Andererseits ist es plausibel, anzunehmen, dass für statische Mengen (etwa die Menge der Schlüsselwörter einer Programmiersprache) statistische Aussagen über die Zugriffshäufigkeiten vorliegen. Beispielsweise kommen in vielen Programmiersprachen Schlüsselwörter wie „begin“ und „end“ häufiger als andere vor. Um für die sehr häufig benötigten Schlüsselwerte die logarithmische Suchzeit unterbieten zu können, liegt es nahe, auf eine AVL-Balancierung zu verzichten und statt dessen häufig vorkommende Schlüsselwerte möglichst weit oben im Baum zu plazieren.

Im Folgenden gehen wir davon aus, dass eine statische Datenmenge bestehend aus n Datensätzen mit den Schlüsselwerten x_1, \dots, x_n und positiven Zugriffswahrscheinlichkeiten p_1, \dots, p_n vorliegen. Unser Ziel ist die Konstruktion eines Suchbaums mit *minimaler mittlerer Suchzeit* hinsichtlich der gegebenen Zugriffsverteilung.

Definition 5.5.4 (Mittlere Suchzeit). Sei x_1, \dots, x_n eine Folge von Schlüsselwerten mit Zugriffswahrscheinlichkeiten p_1, \dots, p_n . D.h. mit Wahrscheinlichkeit p_i wird auf x_i zugegriffen, wobei wir $p_1, \dots, p_n > 0$ und $p_1 + \dots + p_n = 1$ voraussetzen. Weiter sei \mathcal{T} ein binärer Suchbaum für x_1, \dots, x_n . Die mittlere Suchzeit von \mathcal{T} ist durch

$$F(\mathcal{T}) = \sum_{i=1}^n p_i \cdot (\text{Tiefe}_{\mathcal{T}}(x_i) + 1)$$

definiert. Dabei wird jedes x_i mit dem Knoten, auf dem x_i liegt, identifiziert. \square

Beachte, daß $\text{Tiefe}_{\mathcal{T}}(x_i) + 1$ die Anzahl an Vergleichen ist, die benötigt werden, wenn in \mathcal{T} nach dem Schlüsselwert x_i gesucht wird. Also ist $F(\mathcal{T})$ die erwartete Anzahl an Vergleichen, die bei einem (erfolgreichen) Suchvorgang nach einem im Baum gespeicherten Schlüsselwert vorkommen.⁴⁴

Im Folgenden nehmen wir an, daß n Schlüsselwerte x_1, \dots, x_n mit Wahrscheinlichkeiten p_1, \dots, p_n vorliegen. Weiter setzen wir die Sortierung $x_1 \leq \dots \leq x_n$ voraus. Unser Ziel ist die Konstruktion eines Suchbaums \mathcal{T} , so daß $F(\mathcal{T})$ minimal ist unter allen Suchbäumen für x_1, \dots, x_n . Ein solcher Suchbaum wird auch *optimaler Suchbaum* genannt.

Beispiel 5.5.5 (Höhenbalancierte versus optimale Suchbäume). Als Spielzeugbeispiel nehmen wir folgende Wahrscheinlichkeitsverteilung an:

⁴⁴Es gibt Varianten der angegebenen Definition der mittleren Suchzeit, die auch die erfolglose Suche in Betracht ziehen. Wir gehen hierauf nicht ein und beschränken uns auf die angegebene, etwas einfachere Definition.

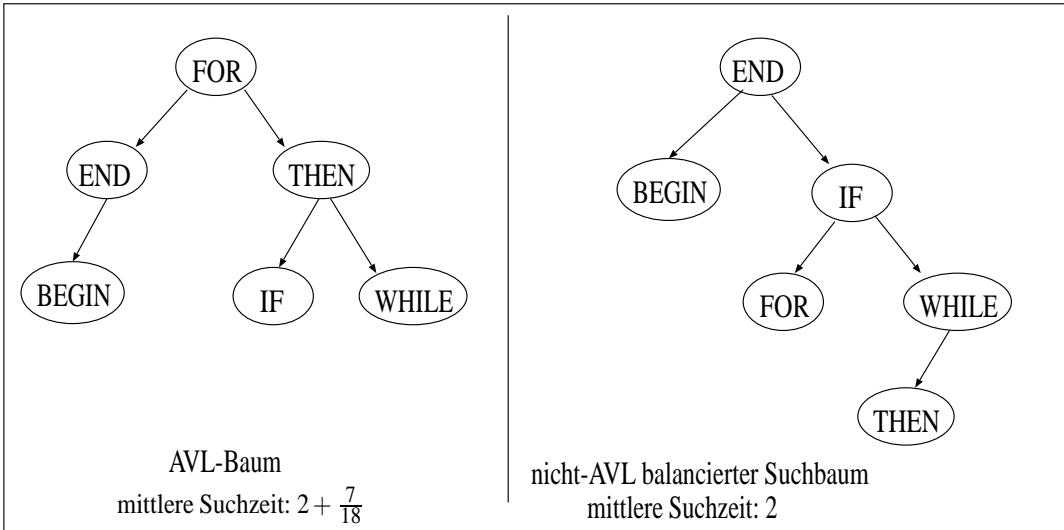


Abbildung 55: AVL-Baum versus optimaler Suchbaum

BEGIN	END	FOR	IF	THEN	WHILE
$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{9}$	$\frac{1}{18}$	$\frac{1}{18}$	$\frac{1}{9}$

Abbildung 55 zeigt links einen AVL-Baum, rechts einen nicht-balancierten Suchbaum. Die mittlere Suchzeit der beiden Bäume errechnet sich wie folgt:

$$F(\mathcal{T}_{AVL}) = 1 \cdot \frac{1}{9} + 2 \cdot \frac{1}{3} + 2 \cdot \frac{1}{18} + 3 \cdot \frac{1}{3} + 3 \cdot \frac{1}{18} + 3 \cdot \frac{1}{9} = \frac{2 + 12 + 2 + 18 + 3 + 6}{18} = 2 + \frac{7}{18}$$

$$F(\mathcal{T}) = 1 \cdot \frac{1}{3} + 2 \cdot \frac{1}{3} + 2 \cdot \frac{1}{18} + 3 \cdot \frac{1}{9} + 3 \cdot \frac{1}{9} + 4 \cdot \frac{1}{18} = \frac{6 + 12 + 2 + 6 + 6 + 4}{18} = 2$$

Tatsächlich ist also der nicht-AVL-balancierte Suchbaum hinsichtlich der mittleren Suchzeit dem AVL-Baum vorzuziehen. \square

Wir stellen nun ein auf dem dynamischen Programmieren beruhenden Algorithmus vor, dessen Idee darin besteht, sukzessive optimale Suchbäume für x_i, \dots, x_j zu erstellen, wobei für die Konstruktion eines optimalen Suchbaums für ℓ Schlüsselwerte auf die optimalen Suchbäume für $\ell - 1$ oder weniger Schlüsselwerte zurückgegriffen wird. Wir benötigen einige zusätzliche Bezeichnungen.

Bezeichnung 5.5.6 (Relativierte mittlere Suchzeiten). Für $1 \leq i \leq j \leq n$ sei

$$p(i, j) = p_i + \dots + p_j$$

$$\begin{aligned} F(i, j) &= \min \left\{ F(\mathcal{T}) : \mathcal{T} \text{ ist optimaler Suchbaum für } x_i, \dots, x_j \right. \\ &\quad \left. \text{mit den bedingten Wahrscheinlichkeiten } \frac{p_i}{p(i, j)}, \dots, \frac{p_j}{p(i, j)} \right\} \end{aligned}$$

$$\tilde{F}(i, j) = p(i, j) \cdot F(i, j)$$

Für $i > j$ setzen wir $F(i, j) = \tilde{F}(i, j) = 0$. Im Folgenden sagen wir „ \mathcal{T} ist optimaler Suchbaum für x_i, \dots, x_j “ und legen dabei die Wahrscheinlichkeitsverteilung $\frac{p_i}{p(i, j)}, \dots, \frac{p_j}{p(i, j)}$ zugrunde. Ist $i \leq j$ und \mathcal{T} ein Suchbaum für x_i, \dots, x_j , so ist

$$\tilde{F}(\mathcal{T}) = p(i, j) \cdot F(\mathcal{T}).$$

Aus technischen Gründen benötigen wir auch den leeren Baum. Für diesen setzen wir $\tilde{F}(\mathcal{T}) = F(\mathcal{T}) = 0$. \square

Wegen $p(1, n) = p_1 + \dots + p_n = 1$ gilt:

$$F(1, n) = \tilde{F}(1, n) = \min \{ F(\mathcal{T}) : \mathcal{T} \text{ ist optimaler Suchbaum für } x_1, \dots, x_n \}$$

Offenbar gilt für $i \leq j$:

$$\tilde{F}(\mathcal{T}) = p(i, j) \cdot F(\mathcal{T}) = \sum_{i \leq k \leq j} p_\ell \cdot (\text{Tiefe}_{\mathcal{T}}(x_\ell) + 1)$$

Ferner gilt für $i < j$:

$$\begin{aligned} \tilde{F}(i, j) &= p(i, j) \cdot F(i, j) \\ &= \min \{ p(i, j) F(\mathcal{T}) : \mathcal{T} \text{ ist optimaler Suchbaum für } x_i, \dots, x_j \} \\ &= \min \{ \tilde{F}(\mathcal{T}) : \mathcal{T} \text{ ist optimaler Suchbaum für } x_i, \dots, x_j \} \end{aligned}$$

und $\tilde{F}(\mathcal{T}) = \tilde{F}(i, i) = p(i, i) = p_i$, falls \mathcal{T} ein (der) Suchbaum für x_i ist, da dieser lediglich aus der Wurzel besteht.

Lemma 5.5.7 (Rekursionsformel für die relativierten mittleren Suchzeiten). Sei $i < j$ und \mathcal{T} ein Suchbaum für x_i, \dots, x_j mit der Wurzel x_k (wobei $i \leq k \leq j$). Weiter sei \mathcal{T}_L der linke Teilbaum von \mathcal{T} und \mathcal{T}_R der rechte Teilbaum von \mathcal{T} . Dann gilt:

$$\tilde{F}(\mathcal{T}) = p(i, j) + \tilde{F}(\mathcal{T}_L) + \tilde{F}(\mathcal{T}_R)$$

Man beachte, daß die Fälle $i = k$ (und somit \mathcal{T}_L leer) sowie $j = k$ (und somit \mathcal{T}_R leer) möglich sind. In diesem Fall ist der betreffende Summand gleich 0.

Beweis. Offenbar ist \mathcal{T}_L ein Suchbaum für x_i, \dots, x_{k-1} und es gilt

$$\text{Tiefe}_{\mathcal{T}_L}(x_\ell) = \text{Tiefe}_{\mathcal{T}}(x_\ell) - 1$$

für jeden Index $\ell \in \{i, \dots, k-1\}$. Entsprechend ist \mathcal{T}_R ein Suchbaum für x_{k+1}, \dots, x_j und es gilt $\text{Tiefe}_{\mathcal{T}_R}(x_\ell) = \text{Tiefe}_{\mathcal{T}}(x_\ell) - 1$ für jeden Index $\ell \in \{k+1, \dots, j\}$.

Wegen $\text{Tiefe}_{\mathcal{T}}(x_k) = 0$ folgt hieraus:

$$\begin{aligned} & \tilde{F}(\mathcal{T}) \\ &= p_k + \sum_{i \leq \ell \leq k-1} p_\ell \cdot (\text{Tiefe}_{\mathcal{T}}(x_\ell) + 1) + \sum_{k+1 \leq \ell \leq j} p_\ell \cdot (\text{Tiefe}_{\mathcal{T}}(x_\ell) + 1) \\ &= p_k + \sum_{i \leq \ell \leq k-1} p_\ell \cdot (\text{Tiefe}_{\mathcal{T}_L}(x_\ell) + 1 + 1) + \sum_{k+1 \leq \ell \leq j} p_\ell \cdot (\text{Tiefe}_{\mathcal{T}_R}(x_\ell) + 1 + 1) \\ &= \underbrace{p_k + \sum_{i \leq \ell \leq k-1} p_\ell}_{=p_i + \dots + p_{k-1} + p_k + p_{k+1} + \dots + p_j = p(i,j)} + \underbrace{\sum_{k+1 \leq \ell \leq j} p_\ell}_{=p(j,k+1)} \\ &\quad + \underbrace{\sum_{i \leq \ell \leq k-1} p_\ell \cdot (\text{Tiefe}_{\mathcal{T}_L}(x_\ell) + 1)}_{=\tilde{F}(\mathcal{T}_L)} + \underbrace{\sum_{k+1 \leq \ell \leq j} p_\ell \cdot (\text{Tiefe}_{\mathcal{T}_R}(x_\ell) + 1)}_{=\tilde{F}(\mathcal{T}_R)} \\ &= p(i,j) + \tilde{F}(\mathcal{T}_L) + \tilde{F}(\mathcal{T}_R) \end{aligned}$$

□

Aus der Rekursionsformel für die relativierten mittleren Suchzeiten erhalten wir die Aussage, dass die Teilbäume optimaler Suchbäume selbst wieder optimal sind. Ist also \mathcal{T} ein optimaler Suchbaum für x_1, \dots, x_n mit Wurzel x_k , so hat \mathcal{T} die in Abbildung 56 auf Seite 289 angedeutete Struktur, wobei der Wurzelknoten x_k so gewählt ist, daß $\tilde{F}(1, k-1) + \tilde{F}(k+1, n)$ minimal ist.

Genauer gilt:

Corollar 5.5.8 (Optimalitätsprinzip).

$$\tilde{F}(i, j) = p(i, j) + \min_{i \leq k \leq j} (\tilde{F}(i, k-1) + \tilde{F}(k+1, j))$$

Diese Beobachtung versetzt uns in die Lage, einen optimalen Suchbaum für x_1, \dots, x_n wie folgt zu bestimmen: Wir betrachten probeweise jeden Knoten x_k als potentiellen Wurzelknoten mit einem optimalen Suchbaum für x_1, \dots, x_{k-1} als linken Teilbaum und einem optimalen Suchbaum für x_{k+1}, \dots, x_n als rechten Teilbaum und bestimmen denjenigen mit der kleinsten mittleren Suchzeit. Dies klingt zunächst nach einem rekursiven Top-Down-Ansatz, der allerdings dazu führen würde, daß mehrere Berechnungen wiederholt stattfinden. Stattdessen arbeiten wir Bottom-Up, siehe Algorithmus 68 auf Seite 288.

Algorithmus 68 Konstruktion eines optimalen Suchbaums

FOR $i = 1, \dots, n$ **DO**

$\tilde{F}(i, i) := p_i;$ (* optimaler Suchbaum für x_i besteht aus *)
 $Wurzel(i, i) := i;$ (* der Wurzel mit dem Schlüsselwert x_i *)
 $p(i, i) := p_i;$

$\tilde{F}(i, i - 1) := 0$ (* leerer Suchbaum *)

OD

$\tilde{F}(n + 1, n) := 0$

FOR $\ell = 1, \dots, n - 1$ **DO**

(* Konstruiere optimale Suchbäume für $x_i, \dots, x_{i+\ell}$ *)

FOR $i = 1, \dots, n - \ell$ **DO**

$p(i, i + \ell) := p(i, i + \ell - 1) + p_{i+\ell};$

(* Berechne $\tilde{F}(i, i + \ell)$ gemäss Corollar 5.5.8 durch *)
(* $\tilde{F}(i, i + \ell) = p(i, i + \ell) + \min_{i \leq k \leq i+\ell} \tilde{F}(i, k - 1) + \tilde{F}(k + 1, i + \ell)$ *)

$min := +\infty;$

FOR $k = i, \dots, i + \ell$ **DO**

IF $min > \tilde{F}(i, k - 1) + \tilde{F}(k + 1, i + \ell)$ **THEN**

$min := \tilde{F}(i, k - 1) + \tilde{F}(k + 1, i + \ell);$

$Wurzel(i, i + \ell) := k$

FI

OD

$\tilde{F}(i, i + \ell) := p(i, i + \ell) + min$

OD

OD

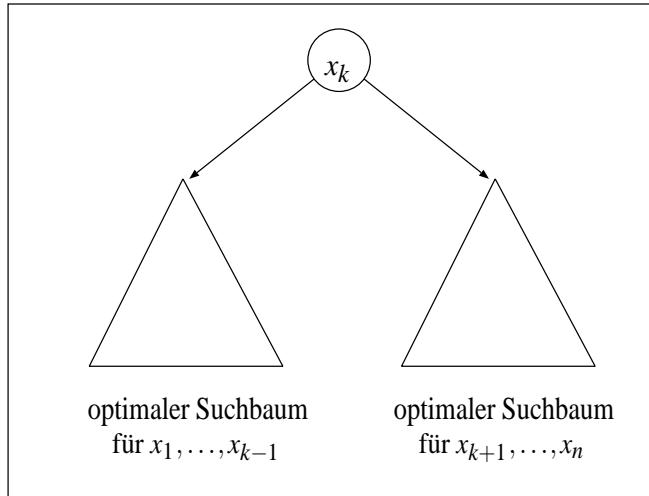


Abbildung 56: Struktur eines optimalen Suchbaums für x_1, \dots, x_n

Zwecks Lesbarkeit wurden einige Anweisungen vereinfacht hingeschrieben. Z.B. genügt es für jedes i einen Zähler $p(i)$ statt $p(i, i + \ell)$ zu verwenden.

Die Laufzeit von Algorithmus 68 ist offenbar kubisch, der Platzbedarf quadratisch. Die Korrektheit ergibt sich aus dem oben genannten Optimalitätsprinzip:

Satz 5.5.9 (Kosten für die Konstruktion eines optimalen Suchbaums). Mit Algorithmus 68 kann ein optimaler Suchbaum in Zeit $\Theta(n^3)$ erstellt werden. Der Platzbedarf ist $\Theta(n^2)$.

Eine Laufzeitverbesserung kann durch folgende Beobachtung erzielt werden. Ist $i < j$, so gilt:

$$\min_{i \leq k \leq j} (\tilde{F}(i, k - 1) + \tilde{F}(k + 1, j)) = \min_{Wurzel(i, j - 1) \leq k \leq Wurzel(i + 1, j)} (\tilde{F}(i, k - 1) + \tilde{F}(k + 1, j))$$

Daher kann die Wurzel eines optimalen Suchbaums für x_i, \dots, x_j , so gewählt werden, daß:

$$Wurzel(i, j - 1) \leq Wurzel(i, j) \leq Wurzel(i + 1, j)$$

Also genügt es, in der inneren FOR-Schleife in Algorithmus 68 lediglich die Werte k zwischen $Wurzel(i, i + \ell - 1)$ und $Wurzel(i + 1, i + \ell)$ zu betrachten. Wir verzichten hier auf den Korrektheitsnachweis der Modifikation und führen nur die Kostenanalyse durch:

Satz 5.5.10 (Laufzeitverbesserung von Algorithmus 68). Mit der genannten Modifikation ist die Laufzeit von Algorithmus 68 durch $\mathcal{O}(n^2)$ beschränkt.

Beweis. Seien $T(i, i + \ell)$ die Kosten zur Bestimmung des Wurzelknotens k eines optimalen Suchbaums für $x_i, \dots, x_{i + \ell}$, die in der genannten Modifikation von Algorithmus 68

durch die innere FOR-Schleife entstehen. Weiter schreiben wir kurz $W(i, i + \ell)$ statt $Wurzel(i, i + \ell)$. Dann gilt:

$$T(i, i + \ell) = W(i + 1, i + \ell) - W(i, i + \ell - 1) + 1$$

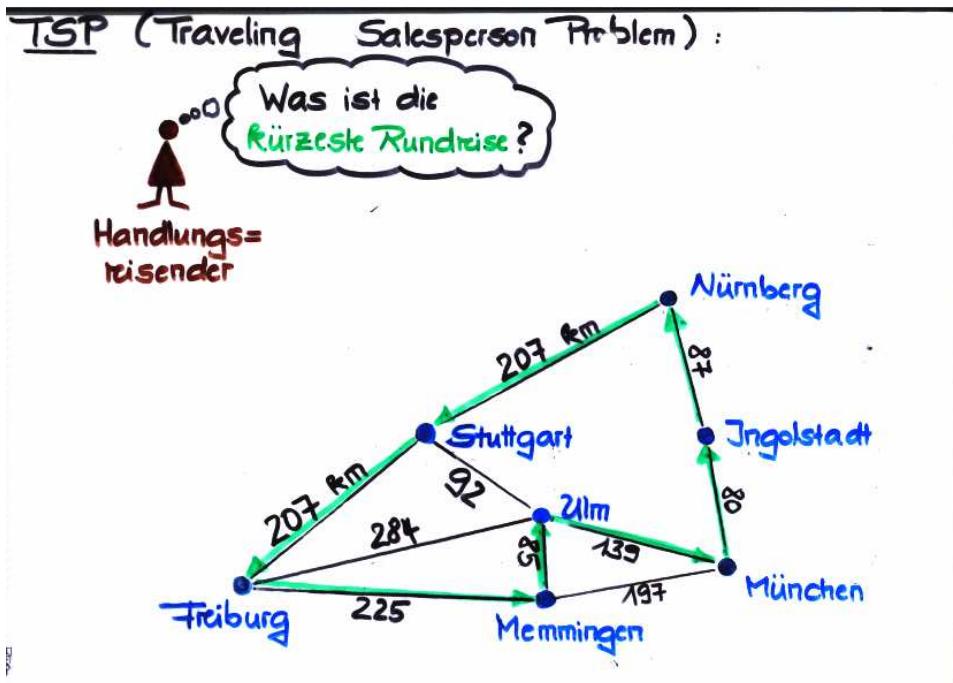
Die Gesamtkosten errechnen sich nun wie folgt:

$$\begin{aligned} & \sum_{\ell=1}^{n-1} \sum_{i=1}^{n-\ell} T(i, i + \ell) \\ &= \sum_{\ell=1}^{n-1} \sum_{i=1}^{n-\ell} (W(i + 1, i + \ell) - W(i, i + \ell - 1) + 1) \\ &= \sum_{\ell=1}^{n-1} \sum_{i=1}^{n-\ell} W(i + 1, i + \ell) - \sum_{\ell=1}^{n-1} \sum_{i=1}^{n-\ell} W(i, i + \ell - 1) + \sum_{\ell=1}^{n-1} \sum_{i=1}^{n-\ell} 1 \\ &= \sum_{\ell=1}^{n-1} \sum_{j=2}^{n-\ell+1} W(j, j + \ell - 1) - \sum_{\ell=1}^{n-1} \sum_{i=1}^{n-\ell} W(i, i + \ell - 1) + \sum_{\ell=1}^{n-1} (n - \ell) \\ &= \sum_{\ell=1}^{n-1} (\underbrace{W(n - \ell + 1, n)}_{\leq n} - \underbrace{W(1, \ell)}_{\geq 1}) + \frac{n(n - 1)}{2} \\ &\leq \sum_{\ell=1}^{n-1} (n - 1) + \frac{n(n - 1)}{2} = \Theta(n^2) \end{aligned}$$

□

5.5.5 Das Problem des Handlungsreisenden.

Die Problemstellung des Handlungsreisenden, welches häufig mit TSP für die englische Bezeichnung *Traveling Salesperson Problem* abgekürzt wird, ist wie folgt. Gegeben sind $n + 1$ Städte v_0, \dots, v_n mit einer Abstandsfunktion δ , die für jedes Städtepaar (v_i, v_j) die Kosten $\delta(v_i, v_j) \in \mathbb{N}$ für die Wegstrecke von v_i nach v_j angibt. Dem Handlungsreisenden stellt sich die Frage nach einer *kürzesten Rundreise*, d.h. eine Tour, in der jede Stadt genau einmal besucht wird.



Formalisierte Fragestellung des TSP. Formalisieren lässt sich das TSP durch einen Digraphen (V, E) mit V und einer Abstandsfunktion $\delta : E \rightarrow \mathbb{N}$.⁴⁵ Gesucht ist ein einfacher Zyklus $\pi = v_0, v_1, \dots, v_n$ in G der Länge $|V|$, für welchen die Kosten

$$\delta(\pi) = \sum_{i=0}^{n-1} \delta(v_i, v_{i+1})$$

minimal sind. (Zur Erinnerung: v_0, v_1, \dots, v_n ist genau dann ein einfacher Zyklus, wenn die ersten n Knoten v_0, v_1, \dots, v_{n-1} paarweise verschieden sind und $v_0 = v_n$.) In diesem Fall ist π also ein Hamiltonkreis, d.h. ein Zyklus, der jeden Knoten von G genau einmal besucht (abgesehen vom Start- und Endknoten). Im Kontext des TSP ist die Bezeichnung Rundreise anstelle von Hamiltonkreis gebräuchlich. Ferner ist es üblich, von einer kürzesten Rundreise zu sprechen, um eine Rundreise mit minimalen Kosten zu bezeichnen.

⁴⁵ Man kann ebenso die reellen Zahlen als Wertebereich für δ zulassen. Dies ist hier – abgesehen von den komplexitätstheoretischen Betrachtungen – unerheblich.

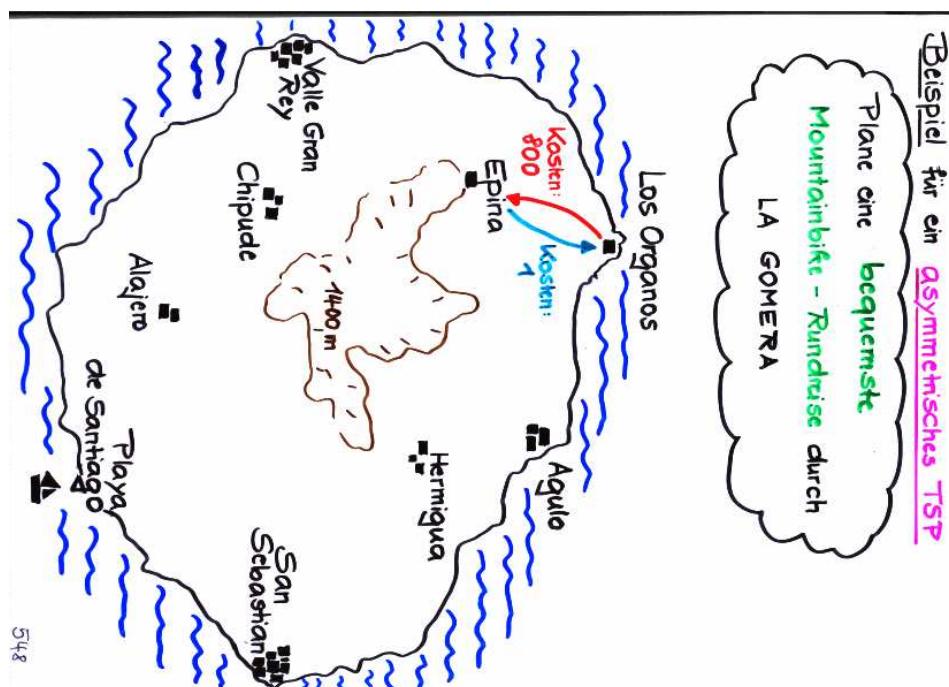
Beispiele für mögliche Instanzen des TSP. Das TSP hat zahlreiche praxisrelevante Anwendungen. Wir nennen ein Beispiel aus der Robotik. Ein Roboter soll in vorgegebenen Zeitintervallen n Schrauben einer Maschine anziehen. Gesucht ist eine Reihenfolge, in der die Schrauben angezogen werden, so daß die Gesamtdauer der notwendigen Roboterarm-Positionierungen minimal ist. Eine weitere Instanz des TSP ist die Frage nach einem optimalen Produktionszyklus. Eine Fabrik ist mit einer Maschine ausgerüstet, die je nach Einstellung eines von n Produkten herstellen kann. Gesucht ist ein kostengünstigster Produktionszyklus. Die Kosten werden an dem Aufwand, der durch die Umstellung der Maschine von Produkt i zu Produkt j entsteht, gemessen.

Man unterscheidet zwei Varianten des TSP:

- symmetrische Variante: die Kantenrelation E ist symmetrisch und es gilt $\delta(v, w) = \delta(w, v)$
- asymmetrische Variante: $\delta(v, w) \neq \delta(w, v)$ ist möglich.

Die symmetrische Variante kann auch so verstanden werden, dass als Eingabe ein ungerichteter Graph mit einer Kantenbeschriftung $\delta : E \rightarrow \mathbb{N}$ vorliegt.

Klassische Instanzen für das asymmetrische TSP sind Wegeprobleme, in denen die Abstandsfunktion nicht die Weglänge sondern die Schwierigkeit einer Strecke mißt. Als Beispiel nennen wir die Frage nach einer bequemsten Mountainbike-Rundreise durch eine Landschaft mit extremen Steigungen bzw. Gefällen (wie die kanarische Insel „La Gomera“).



Obwohl die Problemstellung des TSP zunächst sehr ähnlich der Frage nach kürzesten Wegen (welche in polynomieller Zeit gelöst werden können, siehe Kapitel 6) erscheint, zählt das TSP zu den algorithmisch „schwierigeren“ Problemen:

Satz 5.5.11 (NP-Vollständigkeit des TSP). Die Entscheidungsvariante des TSP, welche für gegebenen Graph mit Abstandsfunktion und gegebenes K fragt, ob es eine Rundreise der Kosten $\leq K$ gibt, ist NP-vollständig.

Beweis. Die Mitgliedschaft in NP ergibt sich aus der Tatsache, dass die Entscheidungsvariante des TSP mit der Guess & Check Methode

- rate eine Knotenfolge v_0, v_1, \dots, v_{n-1} bestehend aus n Knoten, wobei n die Knotenzahl des gegebenen Digraphen ist,
- prüfe, ob v_0, \dots, v_{n-1}, v_0 ein einfacher Zyklus mit den Kosten $\leq K$ ist

nichtdeterministisch, in polynomieller Zeit gelöst werden kann. Die NP-Härte resultiert aus der Tatsache, dass das Hamiltonkreis-Problem als Spezialfall des TSP angesehen werden kann und dieses bereits NP-hart ist. \square

Entscheidungs- versus Optimierungsvarianten des TSP. Bevor wir uns mit Algorithmen für das TSP befassen, machen wir uns den Zusammenhang zwischen der Entscheidungsvariante des TSP und den folgenden beiden Optimierungsvarianten klar, in denen jeweils ein Digraph $G = (V, E)$ mit einer Abstandsfunktion $\delta : E \rightarrow \mathbb{N}$ gegeben ist.

- **Optimierungsvariante 1 des TSP:** Gesucht sind die Kosten einer kürzesten Rundreise.
- **Optimierungsvariante 2 des TSP:** Gesucht ist eine kürzeste Rundreise.

Zunächst ist klar, dass jeder Algorithmus für Optimierungsvariante 2 zugleich als Algorithmus für Optimierungsvariante 1 und dieser wiederum als Lösungsverfahren für die Entscheidungsvariante des TSP eingesetzt werden kann. Andererseits gelten in gewisser Hinsicht auch die Umkehrungen, da mit nur polynomiellem Mehraufwand ein Algorithmus für die Entscheidungsvariante des TSP in einem Algorithmus für Optimierungsvariante 1 eingesetzt werden kann und analoges für die beiden Optimierungsvarianten gilt.

Löse Optimierungsvariante 1 mit einem Algorithmus für die Entscheidungsvariante. Gegeben sind G und δ . Mit dem Prinzip der binären Suche und unter Einsatz eines Algorithmus für die Entscheidungsvariante ermitteln wir den kleinsten Wert K , so dass G eine Rundreise mit den Kosten $\leq K$ besitzt. Der Wertebereich, auf dem die binäre Suche durchzuführen ist, ist $1, 2, \dots, K_{max}$, wobei

$$K_{max} = \sum_{(v,w) \in E} \delta(v, w).$$

Offenbar ist K dann die korrekte Antwort für die Optimierungsvariante 1, vorausgesetzt ein solches K wurde gefunden. Andernfalls besitzt G keine Rundreise und der Wert

$K = \infty$ (oder eine andere entsprechende Ausgabe) kann zurückgegeben werden. Man beachte, dass $\log K_{\max}$ durch die Eingabegröße beschränkt ist. Diese kann nämlich durch

$$N = |V| + |E| + \sum_{(v,w) \in E} \log(\delta(v,w) + 1)$$

angegeben werden. Die Gesamtkosten sind daher $\mathcal{O}(N \cdot T(N))$, wobei $T(N)$ für die Kosten des verwendeten Entscheidungsverfahrens für das TSP stehen.

Löse Optimierungsvariante 2 mit einem Algorithmus für die Optimierungsvariante 1. Liegt andererseits ein Algorithmus für Optimierungsvariante 1 vor, so kann man diesen wie folgt in einem Algorithmus für Optimierungsvariante 2 einsetzen. Zunächst berechnen wir die Kosten K einer kürzesten Rundreise in G , falls es eine solche gibt. Wenn nein, so brechen wir die Berechnung mit einer entsprechenden Ausgabe ab. Wenn ja, so arbeiten wir nach folgendem Schema:

```

 $E_R := \emptyset;$  (* Kantenmenge der Rundreise *)
FOR ALL Kanten  $e \in E$  DO
    berechne die Kosten  $K'$  einer kürzesten Rundreise in  $G \setminus \{e\}$ ;
    IF  $K' > K$  THEN
         $E_R := E_R \cup \{e\};$  (* Kante  $e$  wird für kürzeste Rundreise benötigt *)
    ELSE
         $G := G \setminus \{e\};$  (* Kante  $e$  wird für kürzeste Rundreise nicht benötigt *)
    FI
OD
```

Gib die durch E_R induzierte Rundreise aus.

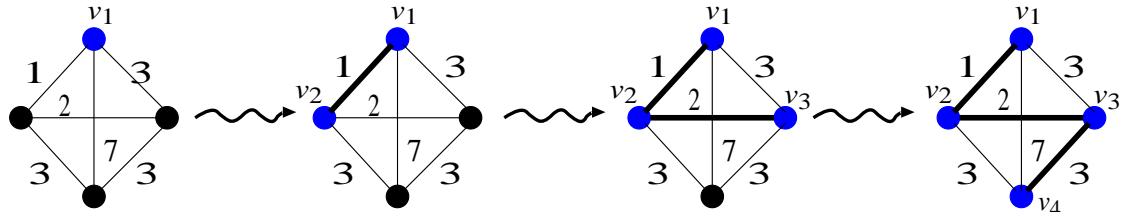
$G \setminus \{e\}$ bezeichnet hier den Graphen, der aus G durch Streichen der Kante e entsteht. Man macht sich leicht klar, dass die Kantenmenge E_R tatsächlich eine kürzeste Rundreise induziert. Ferner verifiziert man auch hier leicht, dass nur polynomieller Mehraufwand – verglichen mit den Kosten, welche der Algorithmus für Optimierungsvariante 1 verursacht – entsteht.

Diese Überlegungen weisen nach, dass alle drei Varianten des TSP denselben komplexitätstheoretischen Schwierigkeitsgrad haben. Liegt nämlich für eine der drei Varianten ein Polynomialzeit-Algorithmus vor, so sind auch die anderen beiden Varianten in polynomieller Zeit lösbar. Ähnliche Aussagen gelten für viele andere Problemstellungen, in denen die Entscheidungsvariante NP-hart ist.

Erweiterte Gewichtsfunktion. Wir diskutieren nun Lösungsverfahren für das TSP (Optimierungsvariante 1 bzw. 2). Da nicht in jedem Graph ein Hamiltonkreis existiert, vereinfachen wir die Formalismen wie folgt. Wir erweitern G zu einem vollständigen Digraphen mit der Kantenmenge $V \times V$, wobei wir $\delta(v,w) = \infty$ für $(v,w) \notin E$ setzen. Im Folgenden nehmen wir also an, daß G ein Digraph mit der Kantenmenge $V \times V$ ist und daß eine Gewichtsfunktion des Typs $\delta : V \times V \rightarrow \mathbb{N} \cup \{\infty\}$ vorliegt. Damit ist sichergestellt, daß jede Knotenfolge ein Weg in G ist. Die Kosten können jedoch ∞ sein.

Heuristische Verfahren für das TSP. Es gibt eine Reihe einfacher und effizienter Greedy-Algorithmen für das TSP (Optimierungsvarianten 2), die zwar im allgemeinen keine optimale Rundreise berechnen, aber dennoch als heuristisches Verfahren zur Berechnung von Näherungswerten eingesetzt werden können. Derartige Verfahren können – wie am Beispiel des $\{0, 1\}$ -Rucksackproblems demonstriert wurde – für den Entwurf von Backtracking und Branch & Bound Algorithmen eingesetzt werden. Im Folgenden setzen wir einen vollständigen Digraphen voraus. Ist dies a-priori nicht erfüllt, so kann man Pseudo-Kanten (v, w) mit $\delta(v, w) = \infty$ hinzufügen.

Erstes heuristisches Verfahren. Eine Möglichkeit besteht darin, mit einem beliebigen Startknoten $s = v_0$ zu beginnen und durch sukzessives Anhängen von Kanten einen einfachen Pfad v_0, v_1, \dots, v_i aufzubauen. Liegt der Pfad v_0, v_1, \dots, v_i bereits vor und ist $i < |V|$, so wählen wir die billigste Kante (v_i, w) , die von dem letzten Knoten v_i des gegebenen Pfads zu einem noch nicht besuchten Knoten $w \in V \setminus \{v_0, \dots, v_i\}$ führt und hängen die Kante (v_i, w) an den Pfad an. Im letzten Schritt liegt ein Hamiltonpfad v_0, \dots, v_{n-1} vor, der alle Knoten des Graphen genau einmal besucht. Dieser wird durch Anhängen der Kante (v_{n-1}, v_0) zu einer Rundreise vervollständigt. Folgende Skizze zeigt ein Beispiel für die beschriebene Methode angewandt auf einen ungerichteten Graphen (symmetrisches TSP):



Letztendlich entsteht also die Rundreise $\pi = v_1, v_2, v_3, v_4, v_1$ mit den Kosten $1+2+3+7=13$. Optimal wäre jedoch die Rundreise v_1, v_2, v_4, v_3, v_1 mit den Kosten $1+3+3+3=10$. Siehe Abbildung 57.

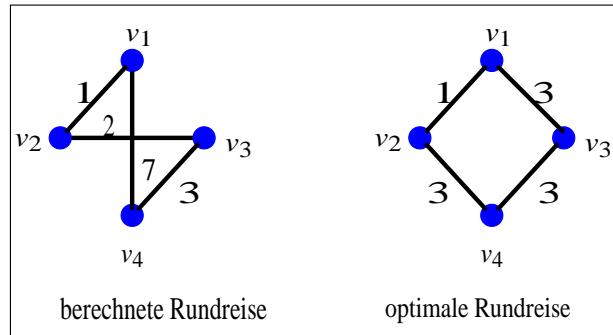


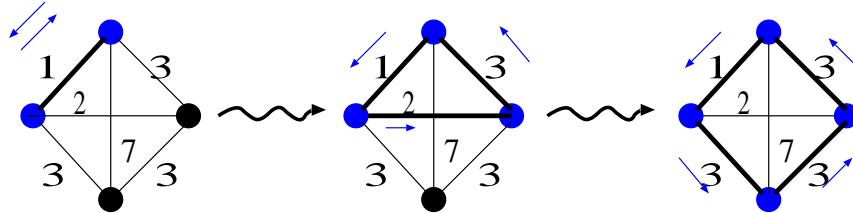
Abbildung 57: Durch die erste Heuristik berechnete Rundreise versus optimale Rundreise

Zweites heuristisches Verfahren. Das eben skizzierte Verfahren liefert in den meisten Fällen nur eine grobe Näherung und kann weit von einer optimalen Rundreise entfernt

sein. Eine Verbesserung des Verfahrens kann durch folgende Modifikation, welche mit einfachen Zyklen anstelle einfacher Pfade arbeitet, erzielt werden. Begonnen wird mit einem Pseudo-Zyklus v_0, v_1, v_0 , etwa mit der Nebenbedingung, dass die Kosten $\delta(v_0, v_1)$ der Kante von v_0 zu v_1 minimal sind.⁴⁶ Alternativ kann man mit dem kürzesten 2er-Pseudo-Zyklus beginnen und v_0 und v_1 so wählen, dass $\delta(v_0, v_1) + \delta(v_1, v_0)$ minimal ist. Der jeweils aktuelle Zyklus v_0, \dots, v_i, v_{i+1} mit $v_{i+1} = v_0$ wird nun durch Ersetzen einer Kante (v_j, v_{j+1}) durch zwei Kanten $(v_j, w), (w, v_{j+1})$ erweitert. Dabei ist w ein Knoten, der nicht in $\{v_0, \dots, v_i\}$ vorkommt. Als heuristisches Kriterium kann die Forderung gestellt werden, dass

$$\delta(v_j, w) + \delta(w, v_{j+1}) - \delta(v_j, v_{j+1})$$

minimal ist unter allen Paaren (j, w) , wobei $0 \leq j \leq i$ und $w \notin \{v_0, \dots, v_i\}$. Ein Beispiel für diese Vorgehensweise für den ungerichteten Graphen von oben ist in folgender Skizze angegeben:



Es ergibt sich hier zufällig eine optimale Rundreise. Im allgemeinen liefert die beschriebene Methode jedoch eine suboptimale Lösung.

Drittes heuristisches Verfahren. Ein andere Methode, die sich in der Praxis als eines der besten heuristischen Verfahren herausgestellt hat, startet mit einem Pfad $\pi = s, v_1, s, v_2, s, \dots, s, v_{n-1}, s$, der alle Knoten in G besucht. Dabei ist s ein beliebig gewählter Startknoten und $V \setminus \{s\} = \{v_1, \dots, v_{n-1}\}$. Zu jedem Zeitpunkt des Verfahrens ist π ein Zyklus, der alle Knoten durchläuft und der aus mehreren Teilzyklen besteht, die – abgesehen von Knoten s – knotendisjunkt sind. Pfad π wird nun sukzessive modifiziert, indem zwei Teilzyklen

$$\pi_1 = \underbrace{s, \dots, v}_{\sigma_1}, s, \quad \pi_2 = s, \underbrace{w, \dots, s}_{\sigma_2}, s$$

von π zusammengefasst und durch den Zyklus

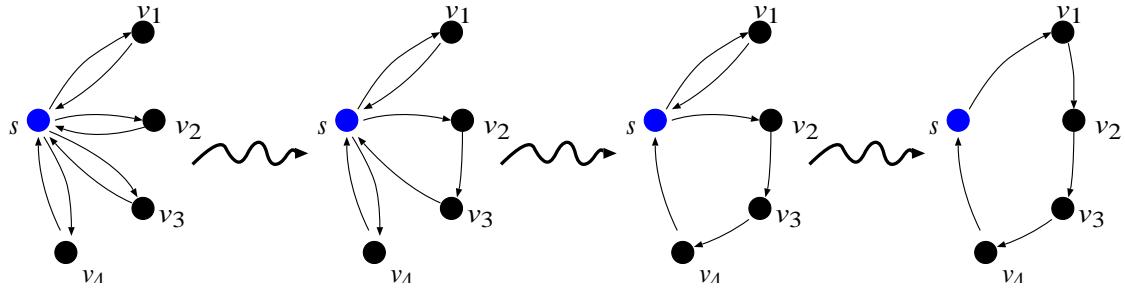
$$\pi' = \underbrace{s, \dots, v}_{\sigma_1}, \underbrace{w, \dots, s}_{\sigma_2},$$

ersetzt werden. Dabei ist die Kante (v, w) so zu wählen, dass die Kostenersparnis

$$\delta(v, s) + \delta(s, w) - \delta(v, w)$$

maximal ist. Folgende Skizze zeigt schematisch für einen Digraphen mit fünf Knoten, wie sich der Pfad π zu einer Rundreise verändern kann:

⁴⁶Wir sprechen hier von einem Pseudo-Zyklus, da das Hin-und-Herlaufen auf einer Kante nicht als Zyklus zählt. Dies ist hier jedoch irrelevant.



Auch hier können Beispiele gegeben werden, in denen das Verfahren eine suboptimale Rundreise bestimmt.

Neben den drei beschriebenen Methoden gibt es eine Reihe weiterer heuristischer Ansätze für das TSP, auf die wir hier nicht eingehen. Im Folgenden diskutieren wir, wie das TSP exakt gelöst werden kann, also wie eine Rundreise minimaler Kosten algorithmisch bestimmt werden kann.

Die naive Lösung für das TSP. Zunächst stellen wir fest, daß die triviale Lösung „alle Permutationen der Knoten ausprobieren“ zu einer miserablen Laufzeit $\Omega(n!)$ führt, wobei n die Anzahl der Knoten ist. Ein kleines Rechenbeispiel: Wir nehmen an, daß pro Sekunde 10^8 Permutationen „ausprobiert“ werden können. Das TSP mit $n = 12$ ist dann in ca. 5 Sekunden gelöst. Für $n = 20$ sind ca. $2.4 * 10^{18}$ Permutationen zu generieren. Wir erhalten nach ca. 800 Jahren eine Antwort. Für $n = 50$ müssen wir noch etwas länger warten: nach ca. 10^{49} Jahren sind schließlich alle $50! \approx 3 * 10^{64}$ Permutationen generiert.

Lösung des TSP mit dynamischen Programmieren. Zunächst fixieren wir einen Startknoten $s = v_0$, in welchem die Rundreise beginnen soll. Die Kosten einer kürzesten Rundreise sind

$$\min cost = \min_{v \in V \setminus \{s\}} (Cost(V \setminus \{s\}, v) + \delta(v, s)).$$

Dabei steht $Cost(V \setminus \{s\}, v)$ für die Kosten eines kürzesten einfachen Wegs von s nach v , der alle Knoten in G durchläuft:

$$s, \underbrace{\dots, v}_{\text{einfacher Pfad durch alle Knoten in } G}, s$$

Obige Formel quantifiziert also über alle möglichen Knoten, die als letztes in einer Rundreise besucht werden. Wir wenden nun die Methode des dynamischen Programmierens zur Berechnung der Werte $Cost(V \setminus \{s\}, v)$ an. Die Idee besteht darin, die wie folgt definierten Werte $Cost(W, v)$ in Bottom-Up Manier, also für immer grösser werdende Knotenmengen W , zu berechnen.

Sei $W \subseteq V \setminus \{s\}$ und $v \in W$. $Cost(W, v)$ bezeichne die Kosten für eine kürzeste Reise von s zu v , wobei jeder Knoten $w \in W$ genau einmal besucht wird und sonst keine

anderen Knoten traversiert werden, also Pfade der Form

$$s, \underbrace{\dots, w}_{\substack{\text{einfacher Pfad durch} \\ \text{alle Knoten in } W \setminus \{v\}}}, v$$

Eine formale Definition der Werte $\text{Cost}(W, v)$ ist wie folgt. Sei $\text{Paths}(W, v)$ die Menge aller Pfade der Form $\pi = s, w_1, \dots, w_k, v$, wobei w_1, \dots, w_k, v paarweise verschieden sind und $W = \{w_1, \dots, w_k, v\}$. Dann ist $\text{Cost}(W, v) = \min \{ \delta(\pi) : \pi \in \text{Paths}(W, v) \}$. Offenbar gilt

$$\text{Cost}(\{v\}, v) = \delta(s, v),$$

da nur der Pfad s, v bestehend aus der Kante von s nach v zur Debatte steht. Ferner haben wir:

Lemma 5.5.12 (Optimalitätsprinzip für das TSP). *Sei $W \subseteq V \setminus \{s\}$ mit $|W| \geq 2$. Dann gilt:*

$$\text{Cost}(W, v) = \min \{ \text{Cost}(W \setminus \{v\}, w) + \delta(w, v) : w \in W \setminus \{v\} \}$$

Beweis. Die Aussage ist klar, da einer der Knoten $w \in W \setminus \{v\}$ auf einem kürzesten einfachen Pfad $\pi \in \text{Paths}(W, v)$ der vorletzte Knoten und das Präfix bis w ein kürzester Pfad in $\text{Paths}(W \setminus \{v\}, w)$ sein muss. \square

Diese Überlegungen führen zu dem in Algorithmus 69 angegebenen Verfahren, welches die Kosten einer kürzesten Rundreise bestimmt. Hierzu werden die Werte $\text{Cost}(W, v)$ zunächst für alle einelementigen Knotenmengen $W \subseteq V \setminus \{s\}$, dann für alle zweielementigen Knotenmengen, dann für alle dreielementigen Knotenmengen, usw. berechnet. In der letzten Iteration erhält man die Werte $\text{Cost}(V \setminus \{s\}, v)$ für alle Knoten $v \in V \setminus \{s\}$, aus denen sich der gesuchte Wert mincost mit der oben genannten Formel ergibt.

Berechnung einer kürzesten Rundreise. Algorithmus 69 wurde so formuliert, dass nur die Kosten einer kürzesten Rundreise ermittelt werden, nicht aber eine kürzeste Rundreise selbst. Zur Berechnung einer kürzesten Rundreise kann wie folgt verfahren werden. Für jedes Paar (W, v) bestehend aus einer Teilmenge W von $V \setminus \{s\}$ und einem Knoten $v \in W$ bestimmen wir einen Knoten $\text{Last}(W, v) = w$, für den (w, v) die letzte Kante eines kürzesten Pfads

$$\pi(W, v) \in \text{Paths}(W, v)$$

ist. Ist also $\text{Last}(W, v) = w$, so gibt es eine kürzeste Reise $\pi(W, v) \in \text{Paths}(W, v)$ von der Form $\pi(W, v) = s, \dots, w, v$. Die Berechnung derartiger Knoten $\text{Last}(W, v)$ kann wie folgt in Algorithmus 69 eingebaut werden:

- Ist $W = \{v\}$ einelementig, dann ist $\text{Last}(\{v\}, v) = s$.
- Ist W mindestens zweielementig, dann kann man $\text{Last}(W, v) = w$ setzen, wobei w ein Knoten in $W \setminus \{v\}$ ein Knoten ist, für den $\text{Cost}(W, v) = \text{Cost}(W \setminus \{v\}, w) + \delta(w, v)$ gilt.

Algorithmus 69 Algorithmus für das TSP

(dynamisches Programmieren)

(* s ist ein fest gewählter Startknoten, in dem die Rundreise beginnt. *)**FOR ALL** Knoten v **DO**

$$\text{Cost}(\{v\}, v) := \delta(s, v)$$

OD**FOR ALL** $\ell = 2, \dots, |V| - 1$ **DO****FOR ALL** ℓ -elementige Teilmengen W von $V \setminus \{s\}$ **DO****FOR ALL** $v \in W$ **DO**

$$\text{Cost}(W, v) := \min_{w \in W \setminus \{v\}} (\text{Cost}(W \setminus \{v\}, w) + \delta(w, v)); \quad (* \text{ siehe Lemma 5.5.12 } *)$$

OD**OD****OD**

$$\text{mincost} := \min_{v \in V \setminus \{s\}} (\text{Cost}(V \setminus \{s\}, v) + \delta(v, s)); \quad (* \text{ Kosten einer kürzesten Rundreise } *)$$

Mit Hilfe dieser Knoten $\text{Last}(\dots)$ können wir kürzeste Reisen $\pi(W, v) \in \text{Paths}(W, v)$ rekursiv konstruieren. Sei W mindestens zweielementig und $w = \text{Last}(W, v)$. Dann ist

$$\pi(W, v) = \underbrace{s, \dots, w}_{\pi(W \setminus \{v\}, w)}, v = \pi(W \setminus \{v\}, w), v$$

eine gesuchte Reise von s nach v . Eine kürzeste Rundreise ist von der Gestalt $\pi(V \setminus \{s\}, v), s$, wobei v ein Knoten in $V \setminus \{s\}$ ist, für den mincost mit dem Wert $\text{Cost}(V \setminus \{s\}, v) + \delta(v, s)$ übereinstimmt.

Im Beweis des folgenden Satzes diskutieren wir die Kosten von Algorithmus 69:

Satz 5.5.13 (Kosten des DP-Algorithmus für das TSP). Das TSP kann mit dynamischem Programmieren in Zeit $\Theta(n^2 2^n)$ und Platz $\Theta(n 2^n)$ gelöst werden. Dabei ist n die Anzahl an Knoten.

Beweis. Um die folgende Rechnung etwas zu vereinfachen gehen wir von einem Graphen mit $|V| = n + 1$ Knoten aus. Dann ist $V \setminus \{s\}$ n -elementig.

Die Initialisierung erfordert offenbar $\Theta(n)$ Schritte. Für $\emptyset \neq W \subseteq V \setminus \{s\}$ können wir die Kosten zur Berechnung der Werte $\text{Cost}(W, v)$ mit $v \in W$ durch $\mathcal{O}(n^2)$ abschätzen. Aufsummieren über alle W ergibt die obere Schranke $\mathcal{O}(n^2 2^n)$.

Wir geben nun eine etwas präzisere Kostenanalyse an, um die untere Schranke $\Omega(n^2 2^n)$ nachzuweisen. Für festes $\ell \in \{2, \dots, n\}$ verursacht jede ℓ -elementige Teilmenge W von

$V \setminus \{s\}$ die Kosten $\Theta(N_W)$, wobei

$$N_W = \sum_{v \in W} \sum_{w \in W \setminus \{v\}} 1 = \ell \cdot (\ell - 1).$$

Aufsummieren über alle Teilmengen ergibt die Kosten $\Theta(N)$, wobei

$$N = \sum_{\ell=2}^n \binom{n}{\ell} \cdot \ell \cdot (\ell - 1).$$

Wir weisen nun nach, dass $N = \Omega(n^2 2^n)$. Für $\ell \in \{2, \dots, n-2\}$ gilt:

$$\binom{n}{\ell} \cdot \ell \cdot (\ell - 1) = \binom{n-2}{\ell-2} \cdot n \cdot (n-1)$$

Somit ist

$$\begin{aligned} N &= \sum_{\ell=2}^{n-2} \binom{n}{\ell} \cdot \ell \cdot (\ell - 1) + \overbrace{n(n-1)}^{\ell=n} + \overbrace{n(n-1)(n-2)}^{\ell=n-1} \\ &= \sum_{\ell=2}^{n-2} \binom{n-2}{\ell-2} \cdot n \cdot (n-1) + \underbrace{n(n-1) + n(n-1)(n-2)}_{=\Theta(n^3)} \\ &\stackrel{\text{Indexverschiebung}}{=} n \cdot (n-1) \cdot \sum_{\ell=0}^{n-4} \binom{n-2}{\ell} + \Theta(n^3) \\ &= n \cdot (n-1) \cdot \left(\sum_{\ell=0}^{n-2} \binom{n-2}{\ell} - \underbrace{(n-2)}_{\ell=n-3} - \underbrace{1}_{\ell=n-2} \right) + \Theta(n^3) \end{aligned}$$

Aus der binomischen Formel

$$(a+b)^m = \sum_{\ell=0}^m \binom{m}{\ell} a^\ell b^{m-\ell}$$

ergibt sich mit $a = b = 1$ und $m = n-2$:

$$\sum_{\ell=0}^{n-2} \binom{n-2}{\ell} = (1+1)^{n-2} = 2^{n-2}$$

Daher gilt:

$$\begin{aligned} N &= n \cdot (n-1) \cdot \left(\sum_{\ell=0}^{n-2} \binom{n-2}{\ell} - (n-2) - 1 \right) + \Theta(n^3) \\ &= n \cdot (n-1) \cdot (2^{n-2} - (n-2) - 1) + \Theta(n^3) \\ &= \Theta(n^2 2^n) \end{aligned}$$

Wir erhalten $N = \Theta(n^2 2^n)$. Die Aussage über den Speicherplatzbedarf folgt mit ähnlichen Argumenten. \square

Die Laufzeit $\Theta(n^2 2^n)$ ist zwar schlecht, aber eine erhebliche Verbesserung gegenüber der naiven Lösung, die alle Knotenpermutationen ausprobiert und die Laufzeit $\Omega(n!)$ hat. Tatsächlich kritisch ist jedoch der enorme Platzbedarf des angegebenen Verfahrens. Eine Verbesserung des Platzbedarfs kann erzielt werden, indem man die Tatsache ausnutzt, dass zur Behandlung der ℓ -elementigen Teilmengen W von $V \setminus \{s\}$ nur die Werte der $(\ell - 1)$ -elementigen Teilmengen benötigt werden, nicht aber die Werte $Cost(W', v)$ für $|W'| \leq \ell - 2$.

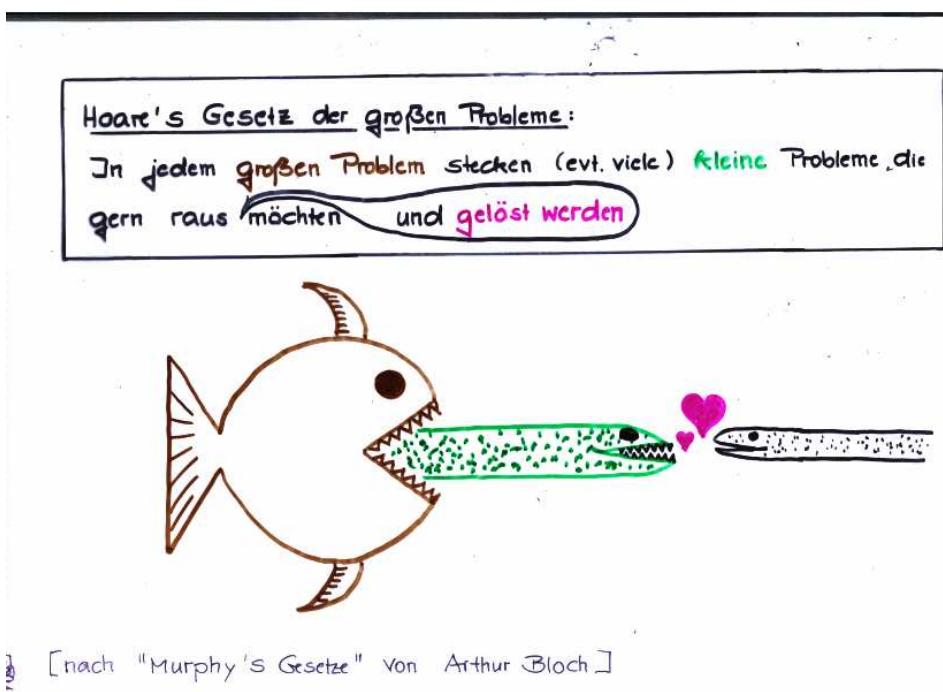
Eine andere Lösungsmöglichkeit für das TSP, die ebenfalls zu exponentieller Laufzeit führt, jedoch weniger platzintensiv ist, besteht in der Verwendung von Branch & Bound Algorithmen. Auch hier gibt es diverse Varianten. Eine Möglichkeit besteht darin, einen binären Aufzählungsbaum zu verwenden, welcher für jede (relevante) Kante e gemäß der Fälle „Kante e gehört zur konstruierten Rundreise“ oder „Kante e gehört nicht zur konstruierten Rundreise“ verzweigt. Als untere Schranke für eine Rundreise in G kann der Wert

$$\frac{1}{2} \sum_{v \in V} \left(\min_{w \in V \setminus \{v\}} \delta(v, w) + \min_{w \in V \setminus \{v\}} \delta(w, v) \right)$$

eingesetzt werden. Dieser ergibt sich aus der Beobachtung, dass in jeder Rundreise für jeden Knoten eine Kante (w, v) in v hineinführt und eine Kante (v, w) aus v hinausführt. Da auf diese Weise $2n$ Kanten betrachtet werden, wird zusätzlich der Faktor $\frac{1}{2}$ benötigt. Liegt nun eine Reihe bereits getroffener Entscheidungen vor, also ein Knoten $\bar{x} = (x_1, \dots, x_i) \in \{0, 1\}^i$ im Aufzählungsbaum, der sich aus den getroffenen Entscheidungen ergibt, etwa $x_j = 0$, falls die j -te Kante nicht in der jeweiligen Rundreise durchlaufen wird ($j = 1, \dots, i$), so ist die genannte Formel für die untere Schranke geeignet zu modifizieren, um eine untere Schranke für die kürzeste Rundreise im Teilbaum von \bar{x} zu berechnen.

Einschub: Allgemeine Bemerkung zur Kostenanalyse von DIVIDE & CONQUER Algorithmen

Einige der Entwurfskonzepte wie DIVIDE & CONQUER (oder dynamisches Programmieren) basieren auf einer Zerlegung des gegebenen Problems in „kleine“ Teilprobleme, deren Lösungen zu einer Lösung des Gesamtproblems zusammengesetzt werden können.



Typisch für DIVIDE & CONQUER-Algorithmen ist die Zerlegung des Problems in (oftmals disjunkte) Teilprobleme, die separat gelöst werden. Ist die Eingabegröße der Teilprobleme hinreichend klein, dann werden die Teilprobleme direkt gelöst. Andernfalls wird dasselbe Zerlegungsschema auf die Teilprobleme angewandt. Die Lösungen der Teilprobleme werden dann zur Gesamtlösung zusammengesetzt. Bereits behandelte Beispiele sind die binäre Suche, Mergesort, Quicksort und der hier diskutierte Linearzeit-Median-Algorithmus. Folgende Tabelle demonstriert die drei Phasen am Beispiel von Merge- und Quicksort:

	Mergesort	Quicksort
DIVIDE	Zerlege die Eingabefolge in zwei gleichgroße Teilstücke	Zerlege die Eingabefolge gemäß eines Pivot-Elements
CONQUER	Sortiere die Teilstücke (rekursiv)	
COMBINE	Mische die sortierten Teilstücke	Hänge die sortierten Teilstücke aneinander

Häufiger Spezialfall ist die Zerlegung in ungefähr gleichgroße Teilprobleme und lineare Zerlegungs- und Zusammensetzungskosten. Dies führt zu einer Kostenfunktion

$$T(n) = \begin{cases} d & : \text{falls } n \leq n_0 \\ a \cdot T(\lceil n/b \rceil) + cn & : \text{falls } n > n_0 \end{cases}$$

wobei wir annehmen, daß das Problem für Eingabegröße $n \leq n_0$ direkt (ohne Rekursionsaufruf) gelöst wird. Die Konstanten a, b sind ganze Zahlen ($a \geq 1, b \geq 2$) mit folgender Bedeutung:

- a ist die Anzahl an Teilprobleme,
- $\lceil n/b \rceil$ ist die Größe der Teilprobleme.

Für die in der Kostenfunktion auftretenden Konstanten c und d setzen wir $c, d \geq 1$ voraus. Für die asymptotischen Kosten kann man o.E. annehmen, daß $c = d$.

Weiter genügt es, sich auf den Fall $n = b^k$ für eine natürliche Zahl k zu beschränken, wenn man nur an der Größenordnung von T interessiert ist.

Satz 5.5.14 (Master-Theorem). *Die Kostenfunktion T sei wie oben.⁴⁷*

- Für $a < b$ (d.h. Gesamtgröße der Teilprobleme $< n$) ist $T(n) = \Theta(n)$.
- Für $a = b$ (d.h. Gesamtgröße der Teilprobleme $= n$) ist $T(n) = \Theta(n \log n)$.
- Für $a > b$ (d.h. Gesamtgröße der Teilprobleme $> n$) ist $T(n) = \Theta(n^{\log_b a})$.

Satz 5.5.14 ist ein Spezialfall der unten angegebenen verallgemeinerten Fassung (siehe Satz 5.5.15).

Z.B. die Kosten für Mergesort ergeben sich aus dem Fall $a = b = 2$. Weiter erhalten wir $\Theta(n)$ als Lösung für $T(n) = \Theta(n) + 3T(\frac{n}{4})$ und $\Theta(n^2)$ als Lösung für $T(n) = \Theta(n) + 9T(\frac{n}{3})$ oder auch $T(n) = \Theta(n) + 4T(\frac{n}{2})$.

In vielen Fällen, in denen Satz 5.5.14 nicht anwendbar ist (z.B. weil das Problem nicht in gleichgroße Teilprobleme unterteilt wird) kann das Master-Theorem hilfreich sein, die Lösung der Rekurrenz für die Kostenfunktion zu erraten und diese dann durch Induktion zu beweisen. Z.B. betrachten wir einen DIVIDE & CONQUER Algorithmus wie den Median-Algorithmus, der ein Problem der Größe $n > n_0$ in zwei Teilprobleme, eines der Eingabegröße $\lfloor n/5 \rfloor$ und eines der Eingabegröße $\lfloor 7/10 \cdot n \rfloor$, unterteilt und dessen Zerlegungs- und Zusammensetzungskosten linear sind. D.h. die Kostenfunktion hat die Gestalt

$$T(n) = T(\lfloor n/5 \rfloor) + T(\lfloor 7/10 \cdot n \rfloor) + c \cdot n$$

für $n > n_0$ und $T(n) = d$ für $n \leq n_0$ (wobei c und d Konstanten > 0 sind). Das Master-Theorem ist nicht anwendbar. Dennoch legt es die Vermutung nahe, daß $T(n) = \Theta(n)$, da die Gesamtgröße der Teilprobleme $\leq n/5 + 7/10n = 9/10n < n$ ist. Eine entsprechende Vorgehensweise führt zum Auffinden der Lösung $F(n) = \Theta(n \log n)$ für die Rekurrenz

$$F(n) = F(\lfloor n/5 \rfloor) + F(\lfloor 4/5n \rfloor) + c \cdot n.$$

Tatsächlich gilt sogar folgende Verallgemeinerung des Master-Theorem, die neben dem Sonderfall gleichgrosser Teilprobleme und lineare Zerlegungs- und Zusammensetzungskosten auch die eben genannten Fälle umfasst:

⁴⁷Die Angabe des Verhältnis zwischen der Gesamtgröße der Teilprobleme und der Eingabegröße n bezieht sich auf den Fall, daß n ein ganzahliges Vielfaches von b ist.

Satz 5.5.15 (Master-Theorem (allgemeine Fassung)). *Gegeben ist eine Rekursionsgleichung der Form*

$$T(n) = \Theta(n^k) + \sum_{i=1}^m T(d_i n) \text{ für } n \geq n_0,$$

wobei $n_0, m \in \mathbb{N}_{\geq 1}$, $k \in \mathbb{N}$ und d_1, \dots, d_m reelle Zahlen im offenen Intervall $]0, 1[$. Dann gilt:

$$T(n) = \begin{cases} \Theta(n^k) & : \text{falls } \sum_{i=1}^m d_i^k < 1 \\ \Theta(n^k \log n) & : \text{falls } \sum_{i=1}^m d_i^k = 1 \\ \Theta(n^r) & : \text{falls } \sum_{i=1}^m d_i^k > 1 \text{ und } \sum_{i=1}^m d_i^r = 1 \end{cases}$$

Zur Vereinfachung schreiben wir $T(d_i n)$ statt $T(\lceil d_i n \rceil)$. Die Werte $T(n)$ für $0 \leq n < n_0$ sind beliebig.

Satz 5.5.14 ergibt sich als ein Spezialfall von Satz 5.5.15, indem wir $k = 1$, $m = a$ und $d_1 = \dots = d_a = \frac{n}{b}$ betrachten.

Beweis. Zur Vereinfachung nehmen wir an, daß $T(0) = 0$, was im Kontext von DIVIDE & CONQUER Algorithmen angemessen ist, wenn man von der naheliegenden Annahme ausgeht, daß für Eingaben der Größe 0 keine Rechenschritte durchgeführt werden.

Zum Nachweis der oberen Schranken (Operator \mathcal{O}) nehmen wir an, daß

$$T(n) \leq \sum_{i=1}^m T(d_i n) + An^k,$$

für alle $n \geq n_0$ und $T(n) \leq An^k$ für $0 \leq n \leq n_0$. Beachte, daß ein solches A existiert, auch wenn aufgrund der Definition von Θ , genauer $f(n) = \mathcal{O}(n^k)$, zunächst $f(n) \leq An$ nur für hinreichend großes n , etwa $n \geq n_1$, zu gelten scheint. Jedoch ist man mit $A \geq \max\{f(0), f(1), \dots, f(n_1 - 1)\}$ auf der sicheren Seite. Für den Nachweis der unteren Schranken (Operator Ω) wird

$$T(n) \geq \sum_{i=1}^m T(d_i n) + an^k$$

für alle $n \geq n_0$ und $T(n) \geq an^k$ für $0 \leq n < n_0$ vorausgesetzt.

Wie bereits erwähnt gehen wir lässig mit den Zahlen $d_i n$ um und behandeln diese wie ganze Zahlen.

1. Fall. Wir beginnen mit dem Fall $\sum_{i=1}^m d_i < 1$.

Zu zeigen ist, daß es eine Konstante $C > 0$ gibt, so daß $T(n) \leq Cn^k$ für hinreichend großes n . Dies weisen wir mit dem Prinzip der induktiven Ersetzung nach und nehmen

dabei an, daß $T(m) \leq Cm$ für $m < n$.

$$\begin{aligned}
T(n) &\leq \sum_{i=1}^m T(d_i n) + An^k \\
&\leq \sum_{i=1}^m Cd_i^k n^k + An^k \\
&= n^k \cdot \left(A + C \underbrace{\sum_{i=1}^m d_i^k}_{=:D} \right) \\
&= n^k \cdot (A + C \cdot D)
\end{aligned}$$

Man beachte, daß $0 < D < 1$. Es gilt:

$$A + CD \leq C \text{ gdw } A \leq C(1 - D) \text{ gdw } C \geq \frac{A}{1-D}$$

Also können wir $C \geq \max \{A, \frac{A}{1-D}\} = \frac{A}{1-D}$ wählen und erhalten $T(n) \leq Cn^k$ für alle $n \geq 0$ und somit $T(n) = \mathcal{O}(n^k)$. Die Tatsache, daß $T(n) = \Omega(n^k)$ folgt sofort aus der Beziehung $T(n) = \Theta(n^k) + \sum_i T(\dots)$.

2. Fall. Wir diskutieren nun den Fall $\sum_{i=1}^m d_i^k = 1$.

Für die obere Schranke ist der Nachweis zu führen, daß $T(n) \leq Cn^k \log n$ für eine geeignete wählende positive Konstante C . Auch hier arbeiten wir zunächst heuristisch:

$$\begin{aligned}
T(n) &\leq \sum_{i=1}^m T(d_i n) + An^k \\
&\leq \sum_{i=1}^m Cd_i^k n^k \log(d_i n) + An^k \\
&= n^k \cdot \left(A + C \sum_{i=1}^m d_i^k (\log d_i + \log n) \right) \\
&= n^k \cdot \left(A + C \sum_{i=1}^m d_i^k \log d_i + C \underbrace{\sum_{i=1}^m d_i^k \log n}_{=\log n} \right) \\
&= n^k \cdot \left(A + \underbrace{C \sum_{i=1}^m d_i^k \log d_i}_{=:E} + C \log n \right) \\
&= n^k \cdot (A + CE + C \log n)
\end{aligned}$$

Beachte, daß $E < 0$, da die d_i 's alle zwischen 0 und 1 liegen und daher $\log d_i < 0$. Wir wählen $C \geq \max\{A, -\frac{A}{E}\}$ und erhalten

$$n^k(A + CE + C \log n) \leq n^k(A - \frac{A}{E} \cdot E + C \log n) = n^k(A - A + C \log n) = Cn^k \log n$$

und somit $T(n) = \mathcal{O}(n^k \log n)$. Die Rechnung für die untere Schranke ist analog, wobei wir von der Rekurrenz

$$T(n) \geq \sum_{i=1}^m T(d_i n) + an^k$$

ausgehen und $T(m) \geq cm^k \log m$ für $m < n$ annehmen:

$$\begin{aligned} T(n) &\geq \sum_{i=1}^m T(d_i n) + an^k \\ &\geq \sum_{i=1}^m cd_i^k n^k \log(d_i n) + an^k \\ &= n^k \cdot \left(a + c \sum_{i=1}^m d_i^k (\log d_i + \log n) \right) \\ &= n^k \cdot \left(a + \underbrace{c \sum_{i=1}^m d_i^k \log d_i}_{=E} + \underbrace{c \sum_{i=1}^m d_i^k \log n}_{=\log n} \right) \\ &= n^k \cdot (a + cE + c \log n) \end{aligned}$$

Hier wählen wir c so, daß $0 < c \leq \min\{a, -\frac{a}{E}\}$ und erhalten $T(n) \geq cn$.

3. Fall. Es bleibt der Fall $\sum_{i=1}^m d_i^k > 1$ und $\sum_{i=1}^m d_i^r = 1$.

Offenbar ist dann $k < r$, da die d_i 's zwischen 0 und 1 liegen. Zunächst betrachten wir die untere Schranke, die wir mit $T(n) \geq cn^r$ ansetzen:

$$\begin{aligned} T(n) &\geq \sum_{i=1}^m T(d_i n) + an^k \\ &\geq \sum_{i=1}^m c(d_i n)^r + an^k \\ &\geq \sum_{i=1}^m cd_i^r n^r + an^k \\ &\geq cn^r \sum_{i=1}^m d_i^r = cn^r \end{aligned}$$

Hier muß die Konstante c also nur so gewählt werden, daß sie auf den Induktionsanfang passt, d.h. wir können $c = a$ setzen.

Nun zur oberen Schranke. Unser Ansatz ist nun $T(n) \leq Cn^r - n^s$ für eine Konstante s mit $k < s < r$. Im folgenden sei $F = \sum_{1 \leq i \leq m} d_i^s - 1$. Dann ist $F > 0$.

$$\begin{aligned} T(n) &\leq \sum_{i=1}^m T(d_i n) + An^k \\ &\leq \sum_{i=1}^m Cd_i^r n^r - \sum_{i=1}^m d_i^s n^s + An^k \\ &= Cn^r - (1 + F)n^s + An^k \end{aligned}$$

Wegen $F > 0$ und $s > k$ gilt $-Fn^s + An^k \leq 0$ für hinreichend grosses n (etwa $n \geq n_1$). Also kann auch hier die Konstante C gemäß des Induktionsanfangs (also passend für die Fälle $n < \max\{n_0, n_1\}$) gewählt werden. \square

Mit $k = 0$, $m = 1$ und $d_1 = \frac{1}{2}$ erhält man die Rekurrenz für die binäre Suche $T(n) = \Theta(1) + T(\frac{n}{2})$ und damit die Lösung $\Theta(\log n)$ gemäß Fall 2. Mit $k = 1$, $m = 2$ und $d_1 = d_2 = \frac{1}{2}$ erhalten wir die von Mergesort bekannte Rekurrenz $T(n) = \Theta(n) + 2T(\frac{n}{2})$ und deren Lösung $\Theta(n \log n)$, ebenfalls gemäß Fall 2.

Eine Instanz des ersten Falls mit $k = 1$, $m = 2$ und $d_1 = \frac{1}{5}$, $d_2 = \frac{7}{10}$ ist die Rekurrenz

$$T(n) = \Theta(n) + T(\frac{n}{5}) + T(\frac{7}{10}n),$$

die wir für den Linearzeit-Algorithmus zur Mediansuche erhalten hatten. Ein Beispiel für den dritten Fall wird in folgendem Unterabschnitt angegeben.

Matrizenmultiplikation nach Strassen

Gegeben sind zwei $(n \times n)$ -Matrizen \mathbf{A} und \mathbf{B} , wobei $n = 2^k$ eine Zweierpotenz ist (und k eine ganze Zahl ≥ 1). Gesucht ist das Matrizenprodukt $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$. Die Schlemethode (siehe Algorithmus 70) benötigt offenbar $\Theta(n^3)$ Schritte.

Naives DIVIDE & CONQUER. Wir stellen nun einen DIVIDE & CONQUER-Algorithmus, welcher die drei Matrizen \mathbf{A} , \mathbf{B} und \mathbf{C} in jeweils 4 $(\frac{n}{2} \times \frac{n}{2})$ -Matrizen zerlegt:

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{1,1} & \mathbf{A}_{1,2} \\ \mathbf{A}_{2,1} & \mathbf{A}_{2,2} \end{pmatrix} \quad \mathbf{B} = \begin{pmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} \end{pmatrix} \quad \mathbf{C} = \begin{pmatrix} \mathbf{C}_{1,1} & \mathbf{C}_{1,2} \\ \mathbf{C}_{2,1} & \mathbf{C}_{2,2} \end{pmatrix}$$

Die Produktmatrix \mathbf{C} ergibt sich nun durch:

$$\begin{aligned} \mathbf{C}_{1,1} &= \mathbf{A}_{1,1} \cdot \mathbf{B}_{1,1} + \mathbf{A}_{1,2} \cdot \mathbf{B}_{2,1} \\ \mathbf{C}_{1,2} &= \mathbf{A}_{1,1} \cdot \mathbf{B}_{1,2} + \mathbf{A}_{1,2} \cdot \mathbf{B}_{2,2} \\ \mathbf{C}_{2,1} &= \mathbf{A}_{2,1} \cdot \mathbf{B}_{1,1} + \mathbf{A}_{2,2} \cdot \mathbf{B}_{2,1} \\ \mathbf{C}_{2,2} &= \mathbf{A}_{2,1} \cdot \mathbf{B}_{1,2} + \mathbf{A}_{2,2} \cdot \mathbf{B}_{2,2} \end{aligned}$$

Algorithmus 70 Matrizenmultiplikation für zwei $(n \times n)$ -Matrizen (Schulmethode)

Eingabe sind zwei Matrizen $\mathbf{A} = (a_{i,j})_{1 \leq i,j \leq n}$ und $\mathbf{B} = (b_{j,k})_{1 \leq j,k \leq n}$ *)

FOR $i = 1, \dots, n$ **DO**

FOR $k = 1, \dots, n$ **DO**

$c_{i,k} := 0;$

FOR $j = 1, \dots, n$ **DO**

$c_{i,k} := c_{i,k} + a_{i,j} * b_{j,k}$

OD

OD

OD

Gib die Produktmatrix $\mathbf{C} = (c_{i,k})_{1 \leq i,k \leq n}$ aus.

Für die Berechnung von \mathbf{C} durch die angegebenen Formeln erhält man die Kostenfunktion $T(1) = 1$ und

$$T(n) = 8 \cdot T\left(\frac{n}{2}\right) + \Theta(n^2).$$

Dies ergibt sich aus der Beobachtung, daß 8 Multiplikationen und 4 Additionen von $(\frac{n}{2} \times \frac{n}{2})$ -Matrizen nötig sind.

Mit $k = 2, m = 8, d_1, \dots, d_m = \frac{1}{2}$ erhalten wir aus Satz 5.5.15:

$$T(n) = 8 \cdot T(n/2) + \Theta(n^2) = \Theta(n^3).$$

Beachte: $8(\frac{1}{2})^2 = \frac{8}{4} = 2 > 1$. Daher ist der dritte Teil von Satz 5.5.15 relevant. Weiter gilt

$$8(\frac{1}{2})^3 = \frac{8}{8} = 1,$$

also ist $r = 3$. Mit dem naiven Divide & Conquer ist also gegenüber der „Schulmethode“ keine asymptotische Zeitersparnis zu verzeichnen.

Matrizenmultiplikation nach Strassen. Wir betrachten nun die von Strassen vorgeschlagene Methode. Diese benutzt ebenfalls die Zerlegung der beiden Matrizen \mathbf{A} und \mathbf{B} in die Untermatrizen $\mathbf{A}_{i,j}$ und $\mathbf{B}_{i,j}$. Die Berechnung der Untermatrizen $\mathbf{C}_{i,j}$ der Produktmatrix \mathbf{C} beruht auf dem folgenden Schema.⁴⁸

$$\begin{aligned} \mathbf{M}_1 &:= (\mathbf{A}_{1,1} - \mathbf{A}_{2,2}) \cdot (\mathbf{B}_{2,1} + \mathbf{B}_{2,2}) \\ \mathbf{M}_2 &:= (\mathbf{A}_{1,2} + \mathbf{A}_{2,2}) \cdot (\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\ \mathbf{M}_3 &:= (\mathbf{A}_{2,1} - \mathbf{A}_{1,1}) \cdot (\mathbf{B}_{1,1} + \mathbf{B}_{1,2}) \\ \mathbf{M}_4 &:= (\mathbf{A}_{1,1} + \mathbf{A}_{1,2}) \cdot \mathbf{B}_{2,2} \\ \mathbf{M}_5 &:= \mathbf{A}_{1,1} \cdot (\mathbf{B}_{1,2} - \mathbf{B}_{2,2}) \\ \mathbf{M}_6 &:= \mathbf{A}_{2,2} \cdot (\mathbf{B}_{2,1} - \mathbf{B}_{1,1}) \\ \mathbf{M}_7 &:= (\mathbf{A}_{2,1} + \mathbf{A}_{1,2}) \cdot \mathbf{B}_{1,1} \end{aligned}$$

⁴⁸Die Autorin übernimmt keine Garantie, daß jeder Index in den angegebenen Formeln stimmt.

und

$$\begin{aligned}\mathbf{C}_{1,1} &:= \mathbf{M}_1 + \mathbf{M}_2 - \mathbf{M}_4 + \mathbf{M}_6 \\ \mathbf{C}_{1,2} &:= \mathbf{M}_4 + \mathbf{M}_5 \\ \mathbf{C}_{2,1} &:= \mathbf{M}_6 + \mathbf{M}_7 \\ \mathbf{C}_{2,2} &:= \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_5 - \mathbf{M}_7\end{aligned}$$

Wir verzichten auf den Korrektheitsnachweis und konzentrieren uns stattdessen auf die Kostenanalyse. Die Methode von Strassen benötigt 7 Multiplikationen und 18 Additionen/Subtraktionen von $(\frac{n}{2} \times \frac{n}{2})$ -Matrizen. Dies führt zur Rekurrenz

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + \Theta(n^2).$$

Wiederum wenden wir den dritten Teil von Satz 5.5.15 an, da $7\left(\frac{1}{2}\right)^2 = \frac{7}{4} > 1$. Mit $r = \log 7$ erhalten wir $7\left(\frac{1}{2}\right)^{\log 7} = \frac{7}{7} = 1$ und somit:

$$T(n) = \Theta\left(n^{\log 7}\right) \approx \Theta\left(n^{2.8}\right)$$

Wir fassen die Ergebnisse zusammen:

Satz 5.5.16 (Komplexität der Strassen-Methode). *Mit der Methode von Strassen können zwei $n \times n$ -Matrizen in Zeit $\Theta(n^{\log 7})$ multipliziert werden.*

Experimentelle Ergebnisse haben gezeigt, daß die Strassen-Methode erst für großes n (ca. $n \geq 500$) besser als die Schulmethode ist. In der Praxis empfiehlt sich daher eine gemischte Strategie, die für $n \geq 500$ eine Zerlegung à la Strassen vornimmt. Sobald die Zeilenanzahl auf einen Wert < 500 reduziert ist, wird die Schulmethode angewandt, um die Teilprodukte zu ermitteln.

6 Algorithmen auf Graphen

Graphenprobleme spielen in fast allen Teilbereichen der Informatik eine wichtige Rolle. Grundlegendes zu Graphen sowie die wichtigsten Traversierungsalgorithmen (Tiefen- und Breitensuche) wurde bereits in Abschnitt 3.2 besprochen. Weiter sind aus dieser Vorlesung (Kapitel 5) einige algorithmisch schwierige Graphprobleme, wie Graphfärbungen, das Cliquesproblem, das Problem des Handlungsreisenden oder das Hamiltonkreisproblem bekannt. Dieses Kapitel beschäftigt sich mit einigen zentralen, algorithmischen Fragestellungen der Graphtheorie, für die es effiziente Lösungen gibt.

6.1 Wegeprobleme mit gewichteten Kanten

Wir beginnen mit Kürzeste-Wegeproblemen in Digraphen, bei denen es darum geht, kürzeste Pfade zwischen zwei Knoten zu bestimmen. In Abschnitt 3.2.4 (Seite 91 ff) haben bereits kurz über Wegeprobleme in Graphen gesprochen. Dort ging es jedoch nur um die single source Variante mit Einheitskosten, während wir hier von einer Bewertung der Kanten mit reellen Gewichten ausgehen. Zunächst scheint auch die Fragestellung des TSP (siehe Abschnitt 5.5.5) verwandt zu sein, jedoch ging es dort um die Konstruktion eines Pfads, der alle Knoten genau einmal durchläuft, während wir hier keine derartige Randbedingung stellen und beliebige Pfade, die zwei Knoten verbinden, zulassen. Wir werden sehen, dass sich Kürzeste-Wege-Probleme sehr viel einfacher und effizienter lösen lassen als das TSP.

Wir betrachten nun den gewichteten Fall, in dem ein gerichteter Graphen, dessen Kanten mit reellen oder ganzzahligen Bewertungen beschriftet sind, als Ausgangspunkt dient. Die Bewertungen der Kanten (oft auch Gewichte genannt) sind häufig nicht-negative Zahlen, die als Kosten für das Durchlaufen der Kante interpretiert werden können. Die Kosten können sich aus diversen Kriterien ergeben. Beispielsweise können die Kosten für die Entfernung der Knoten stehen oder für den Preis, der für das Durchfahren der Kante erhoben wird (z.B. Mautgebühren oder Bahnpreise) oder für die Beschwerlichkeit einer Strecke. Z.B. ist für Radfahrer ein geteerter Radweg wesentlich bequemer als eine Off-Road-Strecke. Der Begriff „kürzester Pfad“ ist dem Kontext entsprechend etwa als „kürzester Pfad im geographischen Sinn“, „billigster Pfad“ oder „bequemster Pfad“ zu verstehen.

Der Begriff „Kosten“ ist in der Literatur auch für negative Kantenbewertungen üblich. Stehen die Kantengewichte für den Gewinn bzw. Verlust, der durch das Benutzen einer Kante erzielt wird, dann erhält man einen Graphen, dessen Kanten sowohl positive als auch negative Bewertungen haben können.

Definition 6.1.1 (Gewichtsfunktion). Sei $G = (V, E)$ ein Digraph. Eine Gewichtsfunktion, manchmal auch Kostenfunktion! δ genannt, für (die Kanten von) G ist eine Abbildung

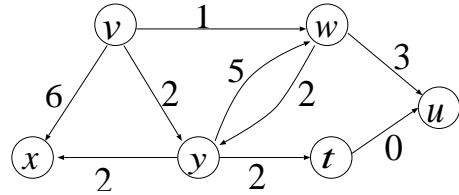
$\delta : E \rightarrow \mathbb{R}$.⁴⁹ Ist $\pi = v_0, v_1, \dots, v_r$ ein Pfad in G , dann wird der Wert

$$\delta(\pi) = \sum_{i=0}^{r-1} \delta(v_i, v_{i+1})$$

die *Kosten* für π bezüglich δ genannt. Ist δ nicht-negativ, also $\delta(v, w) \geq 0$ für alle Kanten $(v, w) \in E$, so spricht man auch von einer *Abstandsfunktion*. \square

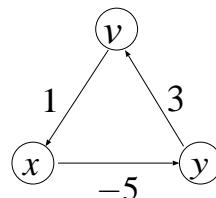
Definition 6.1.2 (Kürzester Pfad/Weg). Seien $G = (V, E)$ ein Digraph, $v, w \in V$ und δ eine Gewichtsfunktion für G . Ein kürzester Pfad (kürzester Weg) von v nach w in G bezüglich δ ist ein Pfad π von v nach w mit $\delta(\pi) \leq \delta(\pi')$ für jeden Pfad π' von v nach w in G . Ergeben sich G und δ aus dem Kontext, dann ist es üblich, den Zusatz „in G bezüglich δ “ wegzulassen.⁵⁰ \square

Beispiel 6.1.3. Auf folgenden Folie ist ein Beispiel zur Illustration der Kosten von Pfaden angegeben.



Ein kürzester Weg von v nach x ist $\pi = v, y, x$. Dieser hat die Kosten $\delta(\pi) = 2 + 2 = 4$ und ist somit günstiger als der Pfad bestehend aus der Kante von v nach x , da $\delta(v, x) = 6$. Der kürzeste, in v startende Weg nach w benutzt lediglich die Kante (v, w) und hat die Kosten 1. Für Knoten u gibt es zwei kürzeste Wege von v nach u , nämlich $\pi_1 = v, w, u$ und $\pi_2 = v, y, t, u$. Diese haben die Kosten $\delta(\pi_1) = 1 + 3 = 4 = 2 + 2 + 0 = \delta(\pi_2)$. \square

Folgende Skizze weist auf die Problematik hin, daß die Lösbarkeit kürzester Wegeprobleme in Graphen mit negativen Kantengewichten nicht allgemein gewährleistet ist.



⁴⁹Für die Kürzeste-Wege-Algorithmen ist es unerheblich, ob die Kantengewichte ganzzahlig oder reell sind. Für die jeweils angegebene Kostenanalyse setzen wir jedoch voraus, dass das Rechnen mit den Kantengewichten (im Wesentlichen nur Addition) exakt und in konstanter Zeit durchgeführt werden kann.

⁵⁰In der Literatur wird häufig auch der Begriff *Länge* (anstelle von Kosten) benutzt. Bei der Verwendung des Begriffs „Länge“ ist der Verzicht auf den Zusatz „bezüglich δ “ jedoch etwas irreführend, da die Länge eines Wegs als die Anzahl der durchlaufenen Kanten definiert ist (siehe Definition 3.2.3 auf Seite 79).

In dem abgebildeten Digraphen gibt es keine kürzeste Wege. Der Grund ist, daß der Zyklus v, x, y, v negative Kosten, nämlich $1 - 5 + 3 = -1$, hat. Um z.B. von v nach y zu gelangen, kann dieser Zyklus beliebig oft durchlaufen werden. Mit jedem Durchlauf des Zyklus reduzieren sich die Kosten:

$$\begin{aligned}\delta(v, x, y) &= -4 \\ \delta(v, x, y, v, x, y) &= -5 \\ \delta(v, x, y, v, x, y, v, x, y) &= -6 \\ &\vdots\end{aligned}$$

Also ist $\inf\{\delta(\pi) : \pi \text{ Pfad von } v \text{ nach } y\} = -\infty$ und es gibt keinen kürzesten Pfad von v nach y . Dasselbe gilt natürlich für Knoten x oder auch v anstelle von y .

Bezeichnung 6.1.4 (Kosten kürzester Wege). Für festes G und δ schreiben wir $\Delta(v, w)$, um die Kosten eines kürzesten Wegs von v nach w (falls existent) zu bezeichnen. Falls es keinen Weg von v nach w gibt, so setzen wir $\Delta(v, w) = \infty$. Falls es beliebig „kurze“ Wege von v nach w gibt, so schreiben wir $\Delta(v, w) = -\infty$. Ist also G ein Digraph, so ist

$$\Delta(v, w) = \inf\{\delta(\pi) : \pi \text{ ist ein Pfad von } v \text{ nach } w\} \in \mathbb{R} \cup \{\infty, -\infty\},$$

wobei $\inf \emptyset = \infty$ gesetzt wird. \square

Existenz kürzester Wege. Liegt ein Digraph mit einer Gewichtsfunktion vor, die nur nicht-negative Werte annimmt, so ist die Existenz eines kürzesten Pfads von Knoten v zu Knoten w gesichert, vorausgesetzt w ist von v erreichbar. Dies erklärt sich aus der Beobachtung, dass kürzeste Wege unter den einfachen Pfaden gesucht werden können, also Pfaden, in denen kein Knoten zweimal oder öfter besucht wird. (Zur Erinnerung: Einfache Pfade sind solche Pfade, in denen jeder Knoten höchstens einmal vorkommt. Da wir uns auf endliche Graphen beschränken, ist die Anzahl an einfachen Pfaden endlich.) Durch die Elimination der Zyklen in einem nicht-einfachen Pfad $\pi = v, \dots, y, x, \dots, x, z, \dots, w$ ergibt sich nämlich ein einfacher Pfad $v, \dots, y, x, z, \dots, w$ mit demselben Anfangs- und Endknoten und dessen Kosten $\leq \delta(\pi)$ sind:

$$\begin{aligned}&\delta(v, \underbrace{\dots, y}_{\pi_1}, \underbrace{x, \dots, x}_{\sigma}, \underbrace{z, \dots, w}_{\pi_2}) \\&= \delta(v, \underbrace{\dots, y}_{\pi_1}, x) + \underbrace{\delta(x, \dots, x)}_{\sigma \geq 0} + \delta(z, \dots, w) \\&\geq \delta(v, \underbrace{\dots, y}_{\pi_1}, x) + \delta(z, \dots, w) \\&= \delta(v, \underbrace{\dots, y}_{\pi_1}, x, \underbrace{z, \dots, w}_{\pi_2})\end{aligned}$$

Offenbar bleibt diese Argumentation korrekt, wenn zwar auch negative Kantengewichte zugelassen sind, aber keine Zyklen negativer Kosten existieren. Wir halten diese Beobachtung in folgendem Lemma fest:

Lemma 6.1.5 (Existenz einfacher kürzester Pfade). Seien $G = (V, E)$ ein Digraph und $\delta : E \rightarrow \mathbb{R}$ eine Gewichtsfunktion. Falls es in G keinen Zyklus mit negativen Kosten gibt, dann gibt für je zwei Knoten $v, w \in V$ mit $w \in Post^*(v)$ einen einfachen Pfad π von v nach w mit $\delta(\pi) = \Delta(v, w)$.

Schließt man also den Fall von Zyklen mit negativen Kosten aus, so gibt es stets einen einfachen, kürzesten Pfad von v nach w , vorausgesetzt w ist von v erreichbar. Genauer gilt:

Lemma 6.1.6. $\Delta(v, w) > -\infty$ gilt genau dann, wenn es keinen Zyklus negativer Kosten gibt, der von v erreichbar und von welchem w erreichbar ist.

Beweis. Die Teilaussage „dann, wenn“ ergibt sich aus der Tatsache, dass

$$\Delta(v, w) = \inf \{ \delta(\pi) : \pi \text{ ist ein einfacher Pfad von } v \text{ nach } w \}$$

(vgl. Lemma 6.1.5 oben). Da die Menge der einfachen Pfade endlich ist, ist $\Delta(v, w) > -\infty$. Den Nachweis der Teilaussage „genau dann“ führen wir durch Kontraposition. Wir nehmen also an, dass $\Delta(v, w) > -\infty$ und dass es einen Pfad der Form

$$\pi = \underbrace{v, \dots, y}_{\lambda_1} \underbrace{x, \dots, x}_{\sigma} \underbrace{z, \dots, w}_{\lambda_2}$$

gibt, wobei der Teilpfad $\sigma = x, \dots, x$ ein Zyklus mit negativen Kosten ist. Also ist $\delta(\sigma) < 0$. Wir betrachten nun die Pfade

$$\pi_i = \underbrace{v, \dots, y}_{\lambda_1} \underbrace{x, \dots, x}_{i\text{-mal } \sigma} \underbrace{z, \dots, w}_{\lambda_2},$$

welche den Zyklus σ i -mal durchlaufen. Es gilt also $\pi = \pi_1$ und

$$\delta(\pi_i) = \underbrace{\delta(v, \dots, y)}_{=\delta(\lambda_1)} + i \cdot \delta(\sigma) + \underbrace{\delta(z, \dots, w)}_{=\delta(\lambda_2)}.$$

Wegen $\delta(\sigma) < 0$ gilt $\lim_{i \rightarrow \infty} \delta(\pi_i) = -\infty$ und somit $\Delta(v, w) = -\infty$. Widerspruch. \square

Lemma 6.1.7. Seien $G = (V, E)$ ein Digraph, $w \in V$ und $\delta : E \rightarrow \mathbb{R}$ eine Gewichtsfunktion. Folgende Aussagen sind äquivalent:

- (a) $\Delta(w, w) = 0$
- (b) $\Delta(w, w) > -\infty$
- (c) w liegt auf keinem Zyklus negativer Kosten.

Beweis. (a) \Rightarrow (b): klar.

(b) \Rightarrow (c) folgt aus der Tatsache, dass im Falle der Existenz eines Zyklus negativer Kosten dieser beliebig oft durchlaufen werden kann und die Kosten somit beliebig klein werden.

(c) \Rightarrow (a): Wenn w auf keinem Zyklus negativer Kosten liegt, so ist $\Delta(w, w) \geq 0$. Andererseits ist klar, dass stets $\Delta(w, w) \leq 0$ gilt, da der Pfad w der Länge 0 die Kosten 0 hat und an der Infimumsbildung $\Delta(w, w) = \inf\{\dots\}$ beteiligt ist. \square

Wir erhalten folgenden Satz:

Satz 6.1.8 (Existenz kürzester Wege und Zyklen negativer Kosten). Seien $G = (V, E)$ ein Digraph und $\delta : E \rightarrow \mathbb{R}$ eine Gewichtsfunktion. Folgende Aussagen sind äquivalent:

- (a) G hat keine Zyklen negativer Kosten.
- (b) $\Delta(v, w) > -\infty$ für alle Knotenpaare (v, w)
- (c) $\Delta(w, w) = 0$ für alle Knoten w
- (d) Für alle Knoten v, w mit $w \in Post^*(v)$ gibt es einen einfachen kürzesten Pfad von v nach w .

Bemerkung 6.1.9 (Wegeprobleme in ungerichteten Graphen). Wir beschränken uns in diesem Abschnitt auf den gerichteten Fall. Selbstverständlich können Wegeprobleme auch für ungerichtete Graphen betrachtet werden. Für ungerichtete Graphen machen jedoch nur nicht-negative Gewichtsfunktionen Sinn. In der angegebenen Definition von Pfaden wurden nämlich auch Pfade der Form v, x, v zugelassen, in denen hintereinander eine Kante in beide Richtungen durchlaufen wird. Hat nun aber die Kante (v, x) ein negatives Gewicht, dann ist die Frage nach kürzesten Wegen witzlos, da die Kosten der Pfade v, x, v, x, v, \dots mit wachsender Pfadlänge weniger werden und ein derartiges Hin- und Herlaufen auf der Kante $(v, x) = (x, v)$ in jeden über x laufenden Pfad eingebaut werden kann.

Die in den folgenden Unterabschnitten angegebenen Algorithmen können auch für ungerichtete Graphen eingesetzt werden, jedoch muss dann vorausgesetzt werden, dass alle Kantenbeschriftungen ≥ 0 sind. Möchte man auch negative Kantenbewertungen zulassen, so muss man sich (aus dem oben genannten Grund) auf kürzeste *einfache* Pfade beschränken. Die angegebenen Algorithmen können dann jedoch nicht ohne weiteres eingesetzt werden. \square

Relaxation. Den drei in den folgenden Unterabschnitten erläuterten Algorithmen liegt folgende offensichtliche Beobachtung zugrunde, die als Dreiecksungleichung für kürzeste Wege angesehen werden kann:

$$\Delta(v, u) \leq \Delta(v, w) + \Delta(w, u)$$

Ist nämlich $\Delta(v, w) < \infty$ und $\Delta(w, u) < \infty$ und sind π_1, π_2 Pfade von v nach w bzw. von w nach u mit $\delta(\pi_1) = \Delta(v, w)$ und $\delta(\pi_2) = \Delta(w, u)$, so ergibt sich durch

Hintereinanderhängen von π_1 und π_2 ein Pfad von v nach u dessen Kosten gleich $\Delta(v, w) + \Delta(w, u)$ betragen. Wegen $\delta(w, u) \geq \Delta(w, u)$ gilt daher auch:

$$\Delta(v, u) \leq \Delta(v, w) + \delta(w, u), \text{ falls } u \in Post(w).$$

Die wesentliche Idee der nun folgenden Algorithmen besteht darin, mit oberen oberen Schranken $d(v, u)$ für $\Delta(v, u)$ zu arbeiten und diese sukzessive durch so genannte *Relaxationsschritte*

$$d(v, u) := \min\{d(v, u), d(v, w) + \delta(w, u)\}$$

bzw. $d(v, u) := \min\{d(v, u), d(v, w) + d(w, u)\}$ zu verbessern.

6.1.1 Algorithmus von Dijkstra

In diesem Abschnitt behandeln wir das kürzeste Wegeproblem in Digraphen mit *nicht-negativen* Kantengewichten und einem als Startknoten ausgezeichneten Knoten v , also eine single-source Variante für nicht-negative Kantengewichte. Das Ziel ist die Bestimmung kürzester Wege von v zu allen anderen Knoten.

Im Folgenden sei also $G = (V, E)$ ein Digraph mit einer Gewichtsfunktion $\delta : E \rightarrow \mathbb{R}_{\geq 0}$ mit nicht-negativen Werten.⁵¹ Weiter nehmen wir an, daß ein festgewählter Startknoten $v \in V$ vorliegt. Da die Kantengewichte ≥ 0 sind, gibt es für alle $w \in Post^*(v)$ wenigstens einen kürzesten Pfad von v nach w . Wie oben erwähnt kann man sich bei der Suche nach kürzesten Pfaden auf einfache Pfade beschränken (Satz 6.1.8).

Bezeichnung 6.1.10. Im Folgenden bezeichne $\Delta(w)$ die Kosten eines kürzesten Pfads von v nach w , falls existent. Genauer:

$$\Delta(w) = \Delta(v, w) = \min\{\delta(\pi) : \pi \text{ ist Pfad von } v \text{ nach } w\},$$

wobei $\min \emptyset = \infty$ gesetzt wird. □

Unser Ziel ist die Berechnung der Werte $\Delta(w)$ für alle Knoten w . Auf die Berechnung kürzester Pfade gehen wir später ein.

Der Dijkstra-Algorithmus. Die Grundidee ist ein Greedy-Algorithmus, der konzeptiell ähnlich wie die auf der Breitensuche beruhenden Methode für das „single-source-unit-weights“-Problem verfährt (siehe Algorithmus 28 auf Seite 94). Wesentlicher Unterschied sind die Zugriffe auf die Knoten der Randmenge W , die nicht durch eine Queue organisiert wird, sondern der stets ein solcher Knoten w entnommen wird, für welchen die bereits berechnete obere Schranke $d(w)$ für $\Delta(w)$ minimal ist.⁵² Zu jedem Zeitpunkt während der Ausführung des Dijkstra-Algorithmus steht der im Algorithmus berechnete Wert

⁵¹Der Algorithmus von Dijkstra kann für gerichtete oder ungerichtete Graphen eingesetzt werden. Wir beschränken uns jedoch in der Diskussion auf den gerichteten Fall.

⁵²Hierzu bietet sich offenbar eine Priority Queue an. Wir werden jedoch auch eine andere Implementierungsmöglichkeit diskutieren. Mehr dazu später.

$d(w)$ für die Kosten eines kürzesten Wegs von v nach w , der – abgesehen vom Endknoten w – nur durch Knoten läuft, für welche die Kosten kürzester Pfade bereits bestimmt sind. Diese Knoten verwaltet der Dijkstra-Algorithmus in einer Menge $D = Done$. Intuitiv besteht die Randmenge aus allen Knoten $w \in V \setminus Done$, so daß $d(w) < \infty$.

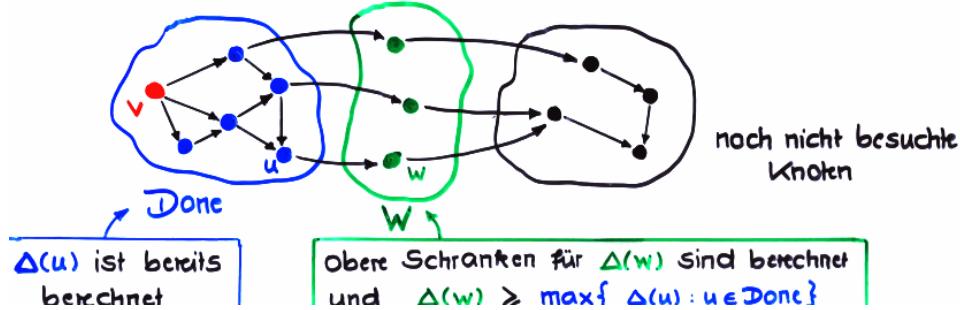
Kürzeste Wege in Graphen mit Abstandsft.

Eingabe: Graph $G = (V, E)$ mit Abstandsft. $\delta : E \rightarrow \mathbb{R}_{\geq 0}$, Startknoten v

Gesucht: kürzeste Wege von v zu allen anderen Knoten und

$$\Delta(w) = \text{Länge eines kürzesten Wegs von } v \text{ zu } w$$

Algorithmus von Dijkstra:



In Algorithmus 71 sind die wesentlichen Schritte zusammengefasst. Man startet mit der Menge $Done = \emptyset$ und $d(v) = 0$, $d(w) = \infty$ für alle $w \in V \setminus \{v\}$. Dann wird sukzessive ein Knoten $w \in V \setminus D$ gewählt, für den $d(w)$ minimal ist. Der gewählte Knoten w wird in $Done$ eingefügt und die oberen Schranken $d(u)$ für $\Delta(u)$ werden für die Knoten $u \in Post(w) \cap (V \setminus Done)$ durch den Relaxationsschritt $d(u) := \min\{d(u), d(w) + \delta(w, u)\}$ zu verbessern versucht. Insbesondere wird stets im ersten Schritt $d(w) = \delta(v, w)$ für alle $w \in Post(v) \setminus \{v\}$ gesetzt.

In der angegebenen Formulierung fügt der Algorithmus alle Knoten w irgendwann in $Done$ ein, auch dann, wenn w nicht von v erreichbar ist. Selbstverständlich kann die WHILE-Schleife abgebrochen werden, sobald es keinen Knoten $w \notin Done$ mit $d(w) < \infty$ mehr gibt (also wenn die Randmenge leer ist), da dann $d(w) = \Delta(w) = \infty$ für alle Knoten $w \in V \setminus Done$.

Beispiel 6.1.11. Folgende Folie zeigt ein Beispiel für die Arbeitsweise des Dijkstra-Algorithmus. Die blauen Knoten bilden die jeweilige $Done$ -Menge. Die Randmenge ist durch die grünen Knoten gegeben.

Algorithmus 71 Algorithmus von Dijkstra

```

FOR ALL Knoten  $w \in V$  DO
  IF  $w \neq v$  THEN
     $d(w) := +\infty$ 
  ELSE
     $d(v) := 0$ 
  FI
OD

```

Done := \emptyset ;

WHILE $Done \neq V$ **DO**

wähle einen Knoten $w \in V \setminus Done$ mit minimalem $d(w)$;
 füge w in $Done$ ein;

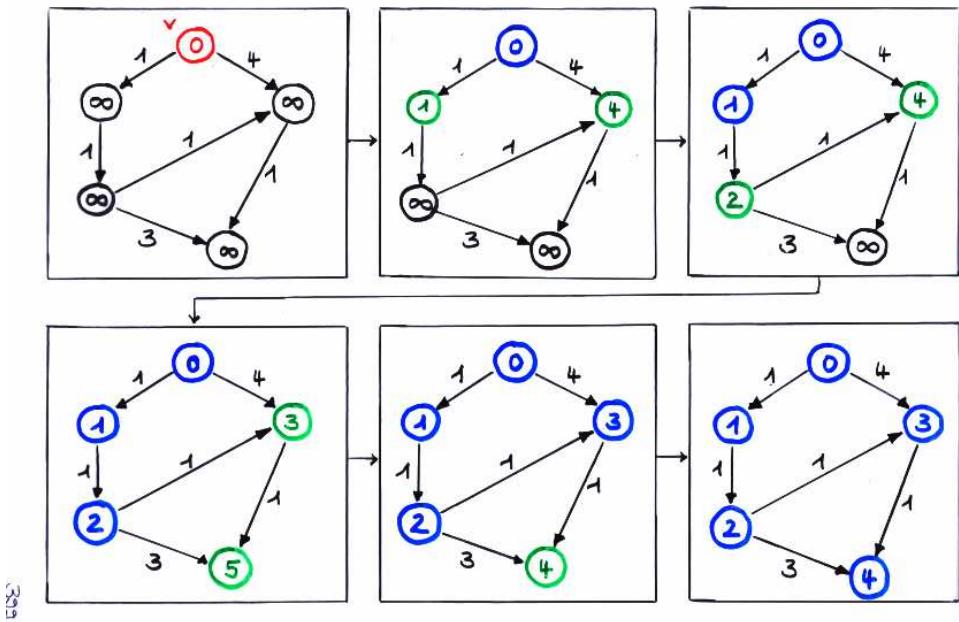
```

FOR ALL  $u \in Post(w) \setminus Done$  DO
     $d(u) := \min\{d(u), d(w) + \delta(w, u)\}$  (* Relaxationsschritt *)
OD

```

OD

Beispiel zum Algorithmus von Dijkstra



□

Korrektheit des Dijkstra-Algorithmus. Folgendes Lemma bildet die Grundlage des Dijkstra-Algorithmus. Intuitiv ist D die Menge derjenigen Knoten, für welche die kürzesten Wege bereits berechnet sind, also $D = Done$. Die Werte $d(u, D)$ in Lemma 6.1.12 stehen für die im Algorithmus berechneten oberen Schranken $\Delta(u)$ für $\Delta(u)$.

Lemma 6.1.12 (Zur Korrektheit des Dijkstra-Algorithmus). Seien G, δ, Δ, v wie oben und $D \subseteq V$, so dass $v \in D$ und

$$\Delta(x) = d(x, D) \leq \Delta(u) \text{ für alle } x \in D \text{ und } u \in V \setminus D.$$

Dabei ist

$$d(y, D) = \min \left\{ \begin{array}{l} \delta(\pi) : \pi = v, x_1, \dots, x_k, y \text{ ist ein Pfad von } v \text{ nach } y \\ \text{mit } v, x_1, \dots, x_k \in D \text{ und } k \geq 0 \end{array} \right\},$$

wobei $\min \emptyset = \infty$. Sei w ein Knoten in $V \setminus D$, für den $d(w, D)$ minimal ist, also $d(w, D) = \min_{u \in V \setminus D} d(u, D)$. Dann gilt:

- (a) $\Delta(w) = d(w, D)$
- (b) Für alle $u \in V \setminus D$ ist $\Delta(u) \geq \Delta(w)$.
- (c) Für alle $u \in V \setminus D$, $w \neq u$ gilt:
 - Ist $u \notin Post(w)$, so ist $d(u, D \cup \{w\}) = d(u, D)$.

- Ist $u \in Post(w)$, so ist $d(u, D \cup \{w\}) = \min\{d(u, D), \Delta(w) + \delta(w, u)\}$.

Beweis. Zunächst ist klar, dass $d(w, D) \geq \Delta(w)$.

ad (a) und (b). Sei w ein Knoten in $V \setminus D$, für den $d(w, D)$ minimal ist und u ein beliebiger Knoten in $V \setminus D$. Weiter sei

$$\pi = \underbrace{v, \dots, x}_{\in D}, \underbrace{y, \dots, u}_{\notin D}$$

ein kürzester Pfad von v nach u , wobei v, \dots, x das längste Präfix von π ist, welches vollständig in D verläuft. Insbesondere gilt $\delta(\pi) = \Delta(u)$ und y ist der erste Knoten in π , der nicht in D liegt. Nach Definition von $d(y, D)$ und aufgrund der Minimalität von $d(w, D)$ ist

$$\delta(v, \dots, x, y) \geq d(y, D) \geq d(w, D) \geq \Delta(w)$$

Da $\delta \geq 0$ sind die Kosten des Suffix y, \dots, w von π nicht-negativ. Also gilt:

$$\Delta(u) = \delta(\pi) = \delta(v, \dots, x, y) + \underbrace{\delta(y, \dots, w)}_{\geq 0} \geq \delta(v, \dots, x, y)$$

Wir kombinieren die beiden Ungleichungen und erhalten Aussage (b):

$$\Delta(u) \geq d(w, D) \geq \Delta(w)$$

Aussage (a) ergibt sich, indem wir $w = u$ betrachten:

$$\Delta(w) \geq d(w, D) \geq \Delta(w)$$

und somit $\Delta(w) = d(w, D)$.

ad (c). Sei u ein Knoten in $V \setminus D$ mit $w \neq u$. Ferner sei

$$\pi = \underbrace{v, \dots, w}_{\in D}, \underbrace{\dots, x}_{\in D} u,$$

ein Pfad von v nach u , welcher durch Knoten w führt und dessen vorletzter Knoten x in D liegt. Dann gilt:

$$\delta(\pi) \geq \Delta(x) + \delta(x, u) = d(x, D) + \delta(x, u) \geq d(u, D)$$

Aufgrund dieser Beobachtung sind für den Übergang von $d(u, D)$ zu $d(u, D \cup \{w\})$ nur solche Pfade von Interesse, welche die Form v, \dots, w, u haben, also solche Pfade, deren vorletzter Knoten w ist. Alle anderen Pfade, die an der Minimumsbildung für $d(u, D \cup \{w\})$ beteiligt sind, werden entweder schon in $d(u, D)$ berücksichtigt oder haben die Kosten $\geq d(u, D)$. Hieraus resultieren die angegebenen Formeln zur Berechnung von $d(u, D \cup \{w\})$ aus $d(u, D)$. \square

Der Algorithmus von Dijkstra basiert auf Lemma 6.1.12, wobei die Menge $D = Done$ alle Knoten $x \in V$ verwaltet, für welche der Wert $\Delta(x)$ bereits berechnet ist, also solche Knoten, für welche $d(x) = d(x, Done) = \Delta(x)$ gilt. Es kann zusätzlich weitere Knoten x mit $d(x) = \Delta(x)$ und $x \notin Done$ geben. Diese werden dann (ohne Veränderung von $d(x)$) in den nächsten Iterationen in $Done$ aufgenommen. Für den in jeder Iteration gewählten Knoten w gilt $w \notin Done$ und $d(w) = d(w, Done)$ ist minimal. Teilaussage (b) von Lemma 6.1.12 induziert daher $\Delta(w) = d(w)$. Teilaussage (c) belegt, dass durch die vorgenommenen Relaxationsschritte die Werte $d(u, Done)$ für die um w erweiterte $Done$ -Menge berechnet werden.

In Algorithmus 72 wurde der Zusammenhang zwischen dem Dijkstra-Algorithmus und den Teilaussagen von Lemma 6.1.12 durch Kommentare verdeutlicht.

Berechnung kürzester Wege. Die angegebene Formulierung von Algorithmus 71 auf Seite 317 berechnet lediglich die Kosten kürzester Wege. Zur Berechnung kürzester Wege kann ein Feld verwendet werden, das für jeden Knoten $w \in Post^*(v)$ einen Knoten $father(w)$ speichert, welcher der durch den Dijkstra-Algorithmus gefundene, direkte Vorgänger von w auf einem kürzesten Pfad von v nach w ist. Siehe Algorithmus 72 auf Seite 317. Derselbe Trick wurde bei der BFS-basierten Kürzeste-Wege-Berechnung eingesetzt. Ist also in der inneren FOR-Schleife (Relaxationsschritt)

$$d(w) + \delta(w, u) < d(u),$$

so wird $father(u)$ zu w gesetzt. Am Ende des Algorithmus ist dann u_0, u_1, \dots, u_k ein kürzester Weg von v nach w , wobei $u_k = w$, $u_{i-1} = father(u_i)$, $i = k, k-1, \dots, 1$, und $u_0 = v$. Selbstverständlich ist dabei $\Delta(w) < \infty$ vorauszusetzen. Für die Knoten w mit $d(w) = \infty$ bei Verlassen der WHILE-Schleife ist $w \notin Post^*(v)$ und somit $d(w) = \Delta(w) = \infty$. In diesem Fall gibt es keinen Weg – und somit auch keinen kürzesten Weg – von v nach w .

Mögliche Implementierungen und Kosten. Wir gehen im Folgenden von einer Adjazenzlisten-Darstellung des Graphen mit den Kantengewichten und einer Bitvektor-Darstellung der $Done$ -Menge aus. Ferner sei $n = |V|$ die Knotenanzahl und $m = |E|$ die Kantenanzahl. Da in jedem Durchlauf der While-Schleife ein Knoten in die $Done$ -Menge eingefügt wird und dieser nicht mehr entnommen wird, wird die While-Schleife genau n -mal durchlaufen wird.

Wir diskutieren nun mehrere Implementierungsvarianten, die sich in der Repräsentation der Randmenge

$$W = \{w \in V : w \notin Done, d(w) < \infty\}$$

unterscheiden.

Keine explizite Darstellung der Randmenge. Die einfachste Möglichkeit besteht darin, auf eine explizite Darstellung von W zu verzichten. Die Bestimmung eines Knotens $w \notin Done$, für den $d(w)$ minimal ist, kann in Zeit $\Theta(n)$ durchgeführt werden. Zusätzlich sind die Kosten für die For-Schleife innerhalb der While-Schleife zu berücksichtigen. Diese

Algorithmus 72 Algorithmus von Dijkstra

```

FOR ALL Knoten  $w \in V$  DO
  IF  $w \neq v$  THEN
     $d(w) := +\infty$ 
  ELSE
     $d(v) := 0;$ 
  FI
OD

```

Done := \emptyset ;

- (*) Schleifeninvariante: $\Delta(v) = \Delta(u)$ für alle $u \in V \setminus \text{Done}$. $\Delta(v) = \Delta(u)$ für alle $u \in \text{Done}$.
- (*) Für $x \in \text{Done}$ ist $d(x) = \Delta(x)$ = Kosten eines kürzesten Pfads von v nach x .
- (*) Für $u \in V \setminus \text{Done}$ ist $d(u) = d(u, \text{Done}) \geq \Delta(u)$. Siehe Teil (b) von Lemma 6.1.12.

WHILE es gibt einen Knoten $w \in V \setminus Done$ mit $d(w) < \infty$ DO

wähle einen Knoten $w \in V \setminus Done$ mit minimalem $d(w)$;

füge w in $Done$ ein; (* $d(w) = \Delta(w)$ *)

(* vgl. Teil (a) von Lemma 6.1.12 *)

FOR ALL $u \in Post(w) \setminus Done$ **DO**

IF $d(u) > d(w) + \delta(w, u)$ **THEN**
 $d(u) := d(w) + \delta(w, u)$ (* Relaxationsschritt *)
 $father(u) := w;$
FI
(* jetzt gilt $d(u) = d(u, Done)$ für die um w erweiterte Done-Menge *)
(* vgl. Teil (c) von Lemma 6.1.12 *)

OD

(* Jetzt gilt $d(w) \equiv \Delta(w)$ für alle Knoten w . *)

(* Ist $d(w) < \infty$ und $u_k = w$, $u_{i-1} = \text{father}(u_i)$, $i = k, k-1, \dots, 1$, und *)
 (* $\text{father}(u_1) = v$, so ist $v, u_1, \dots, u_{k-1}, w$ ein kürzester Pfad von v nach w . *)

betragen $\Theta(K)$, wobei

$$K = \sum_{w \in V} |Post(w)| = |E| \leq n^2$$

und wobei wir über alle Durchläufe der While-Schleife summiert haben. Damit ergeben sich quadratische Gesamtkosten $\Theta(n^2)$.

Darstellung der Randmenge durch eine Priority Queue. Alternativ kann eine Priority Queue (Minimumsheap) zur Verwaltung der Randmenge W eingesetzt werden. Die Elemente der Priority Queue haben die Gestalt von Paaren $\langle w, d(w) \rangle$ und sind hinsichtlich der d -Werte (im Sinn von Heaps) geordnet.

- Eine Implementierungsmöglichkeit besteht darin, für jeden Knoten w maximal einen Eintrag der Form $\langle w, d(w) \rangle$ in der Priority Queue zuzulassen. Durch die Relaxationsschritte kann eine Anpassung der Einträge $\langle u, d(u) \rangle$ für die direkten Nachfolger u des gewählten Knotens w und somit eine Reorganisation der Priority Queue erforderlich werden. Hierzu muss $\langle u, d(u) \rangle$ – wie im Einfüge-Algorithmus für Heaps – an die richtige Position gebracht werden. Die Kosten hierfür sind durch die Höhe des Heaps, also $\mathcal{O}(\log n)$, beschränkt, da die Priority Queue zu jedem Zeitpunkt maximal n Elemente enthält. Zur Unterstützung der Umstrukturierung der Priority Queue können Verweise von den Knoten $w \in W$ zu den betreffenden Einträgen in der Priority Queue eingesetzt werden, um so die zu modifizierenden Einträge in konstanter Zeit zu finden.

Die Reorganisation der Priority Queue, welche durch die Relaxationsschritte erforderlich werden können, verursacht für jeden Knoten $w \in W$, der W als „minimales Element“ entnommen wird, die Kosten $\mathcal{O}(|Post(w)| \cdot \log n)$. Man beachte, dass im schlimmsten Fall für jeden Knoten u der Adjazenzliste von w der Eintrag $\langle u, d(u) \rangle$ geändert oder ein Eintrag $\langle u, d(u) \rangle$ eingefügt werden muss.

Für diese Implementierungsvariante ergeben sich also die Gesamtkosten⁵³

$$\Theta\left(n + \sum_{w \in V} |Post(w)| \cdot \log n\right) = \Theta(n + m \cdot \log n),$$

wobei $n = |V|$ die Knotenanzahl und $m = |E|$ die Kantenanzahl ist.

- Eine weitere Implementierungsmöglichkeit der Priority Queue besteht darin, zuzulassen, daß für einen Knoten u mehrere Einträge in der Priority Queue stehen. Sobald $d(u)$ „verbessert“ wird, wird ein neuer Eintrag in die Priority Queue eingefügt. Bei der Entnahme des minimalen Elements ist dann stets zu prüfen, ob der betreffende Knoten bereits in *Done* liegt oder nicht. Diese Implementierungsmöglichkeit hat den Vorzug, daß sich die Reorganisation der Priority Queue auf das

⁵³Wir haben oben zwar nur mit der oberen Schranke $\mathcal{O}(\cdot)$ argumentiert, jedoch ist die Schranke scharf, da die Priority Queue im schlimmsten Fall $n - 1$ Elemente enthalten kann, nämlich dann, wenn $|Post(v)| \geq n - 1$.

Entfernen und das Einfügen von Elementen beschränkt. Die Priority Queue enthält zu jedem Zeitpunkt höchstens $m = |E|$ Einträge (wobei wir $m \geq 1$ annehmen). Es ergeben sich dieselben asymptotischen Kosten

$$\Theta(n + m \log m) = \Theta(n + m \log n)$$

wie für die zuerst genannte Implementierungsmöglichkeit mit Priority Queues. Beachte, daß $m \leq n^2$ und somit $\log m \leq \log(n^2) = 2 \log n = \mathcal{O}(\log n)$.

Oftmals nimmt man an, daß $m \geq n$. In diesem Fall kann der Summand $\Theta(n)$ weggelassen und die Kostenfunktion durch $\Theta(m \log n)$ abgeschätzt werden. Falls $m \ll n^2$ ist die Implementierung der Randmenge W mit einer Priority Queue der zuvor vorgeschlagenen Variante ohne explizite Darstellung der Randmenge vorzuziehen,⁵⁴ also etwa wenn sehr grosse Graphen betrachtet werden, die nur einen kleinen Verzweigungsgrad haben.
Wir fassen die Resultate zusammen:

Satz 6.1.13 (Komplexität des Algorithmus von Dijkstra). Mit dem Algorithmus von Dijkstra lassen sich kürzeste Wege in Digraphen mit nicht-negativen Kantengewichten von einem Startknoten v zu allen anderen Knoten w bestimmen. Die Zeitkomplexität ist

- $\Theta(n^2)$, falls keine explizite Darstellung der Randmenge W vorgenommen wird,
- $\Theta(m \log n)$, falls die Randmenge W durch eine Priority Queue repräsentiert wird.

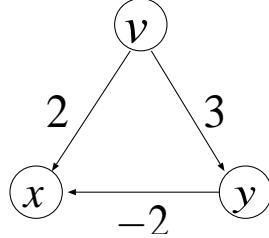
Dabei ist $n = |V|$ die Anzahl an Knoten und $m = |E|$ die Anzahl an Kanten und es wird $m \geq n$ vorausgesetzt.

Bemerkung 6.1.14 (Matroid-Eigenschaft). Die Korrektheit des Dijkstra-Algorithmus haben wir mit den in Lemma 6.1.12 angegebenen Eigenschaften begründet, die zugleich als Schleifeninvariante in Algorithmus 72 dienen und somit plausibel machen, was der Algorithmus eigentlich berechnet. Alternativ hätte die Korrektheit auch mit der Matroid-Theorie (siehe Abschnitt 5.4.2, Seite 259 ff) nachgewiesen werden können.

Das zugrundeliegende Teilmengensystem besteht hier aus der Menge S aller einfachen, in v beginnenden Pfade und der Menge $\mathcal{U} \subseteq 2^S$, die sich aus solchen Pfadmengen $U \subseteq S$ zusammensetzt, so daß je zwei Pfade in U unterschiedliche Endknoten haben. Es ist nun leicht einzusehen, daß (S, \mathcal{U}) ein Matroid ist. Die dem Matroid (S, \mathcal{U}) zugrundegelegte Gewichtsfunktion $w : S \rightarrow \mathbb{R}_{\geq 0}$ ordnet jedem einfachen Pfad π dessen Kosten zu, also $w(\pi) = \delta(\pi)$. Zu betrachten ist hier das zu (S, \mathcal{U}, w) gehörende Minimierungsproblem, welches nach einer \mathcal{U} -maximalen Pfadmenge A fragt, so dass $w(A)$ minimal ist. Der Dijkstra-Algorithmus (Algorithmus 72) weicht zwar von dem in Algorithmus 63 auf Seite 263 angegebenen Greedy-Schema für Matroide ab. Bei genauem Hinsehen stellt man jedoch fest, daß der Dijkstra-Algorithmus nur eine effizientere Formulierung wählt, die nicht über alle Pfade quantifiziert. □

⁵⁴Die Schreibweise „ $x \ll y$ “ ist üblich, um anzudeuten, daß x wesentlich kleiner als y ist.

Bemerkung 6.1.15 (Dijkstra versagt für negative Kantengewichte). Für die Korrektheit des Algorithmus von Dijkstra ist entscheidend, daß die Funktion δ nicht-negativ ist, also der vorliegende Graph keine Kanten mit negativen Gewichten hat. Andernfalls kann die Aussage von Lemma 6.1.12 falsch sein und der Algorithmus von Dijkstra ein falsches Ergebnis liefern.



In obigem Beispiel würde die Greedy-Strategie des Dijkstra-Algorithmus zuerst den Wert $d(x) = 2$ für Knoten x bestimmen und dann $d(y) = 3$ setzen und anhalten. Tatsächlich ist der Pfad v, y, x mit den Kosten $3 - 2 = 1$ jedoch günstiger als der Pfad bestehend aus der Kante von v nach x . Hier versagt die Greedy-Strategie also, da die Entscheidung für x nicht eingefroren werden darf. \square

6.1.2 Algorithmus von Bellman und Ford

Wir gleichen nun das Defizit des Dijkstra-Algorithmus aus und erläutern, wie kürzeste Wege in Graphen mit eventuellen negativen Kantengewichten berechnet werden können. Wie bereits erwähnt, stellt sich dabei die Problematik, daß die Existenz kürzester Wege nicht a-priori gewährleistet ist.

Im Folgenden sei $G = (V, E)$ ein gerichteter Graph und $\delta : E \rightarrow \mathbb{R}$ eine Gewichtsfunktion, wobei $\delta(w, u) < 0$ zugelassen ist. Wir behandeln zunächst den Bellman-Ford-Algorithmus, welcher das single-source Problem, bei dem ein Startknoten v vorgegeben ist, löst. Die all-pairs Variante wird im nächsten Abschnitt (Seite 329 ff) besprochen. Unser Ziel ist es, algorithmisch herauszufinden, ob G keinen Zyklus negativer Kosten enthält und gegebenfalls die Kosten kürzester Wege von v zu allen Knoten $w \in Post^*(v)$ sowie kürzeste Wege selbst zu berechnen. Wie zuvor schreiben wir $\Delta(w)$ statt $\Delta(v, w)$ für die Kosten eines kürzesten Wegs von v nach w (falls existent).

Die prinzipielle Vorgehensweise des Bellman-Ford-Algorithmus ist zum Dijkstra-Algorithmus ähnlich, jedoch liegt dem Bellman-Ford-Algorithmus nicht die Greedy-Strategie zugrunde. Der Bellman-Ford-Algorithmus arbeitet zwar ebenfalls mit oberen Schranken $d(w)$ für $\Delta(w)$, die durch sukzessive Relaxationsschritte verbessert werden. Jedoch werden diese Relaxationsschritte solange wie möglich (bzw. nötig) ausgeführt, beruhend auf dem Schema:

```

FOR ALL Knoten  $w \in V$  DO
    IF  $w \neq v$    THEN  $d(w) := +\infty$    ELSE  $d(v) := 0$    FI
OD
WHILE es gibt  $(w, u) \in E$ , so daß  $d(u) > d(w) + \delta(w, u)$  DO
   $d(u) := d(w) + \delta(w, u)$ 
  
```

OD

Die Terminierung des Schemas ist jedoch nicht gewährleistet, wenn Zyklen negativer Kosten vorliegen. Tatsächlich reicht es aber $n - 1$ Iterationen auszuführen, in denen jeweils alle Kanten für den Relaxationsschritt in Betracht gezogen werden. Dabei ist $n = |V|$ die Knotenanzahl. Intuitiv erklärt sich die Iterationsanzahl $n - 1$ daraus, daß wir nur nach einfachen Pfaden Ausschau halten müssen. Eine formale Erklärung ist in Lemma 6.1.19 zu finden. Das entsprechende Verfahren ist in Algorithmus 73 auf Seite 325 angegeben.

Algorithmus 73 Algorithmus von Bellman und Ford

FOR ALL Knoten $w \in V$ **DO**

IF $w \neq v$ **THEN**

$d(w) := +\infty$

ELSE

$d(v) := 0$

FI

OD

(* sei $n = |V|$ die Knotenanzahl *)

FOR $i = 1, \dots, n - 1$ **DO**

FOR ALL Kanten $(w, u) \in E$ **DO**

$d(u) := \min\{d(u), d(w) + \delta(w, u)\}$

(* Relaxationsschritt *)

OD

OD

(* Prüfe die Existenz kürzester Wege. *)

FOR ALL Kanten $(w, u) \in E$ **DO**

IF $d(u) > d(w) + \delta(w, u)$ **THEN**

 return „Es gibt Zyklen negativer Kosten.“

(* Abbruch *)

FI

OD

(* Jetzt ist $d(w) = \Delta(w)$ für alle $w \in Post^*(v)$. *)

Beispiel 6.1.16. Für den Digraphen aus Bemerkung 6.1.15 auf Seite 324 arbeitet der Bellman-Ford Algorithmus zunächst wie der Dijkstra-Algorithmus und setzt $d(v) = 0$, $d(x) = 2$ und $d(y) = 3$, was sich aus der Betrachtung der Kanten (v, x) und (v, y) in der ersten Iteration ergibt. Nach Betrachtung der Kante (v, y) wird entweder in der ersten oder der zweiten Iteration der mittleren FOR-Schleife der Relaxationsschritt

$$d(x) := \min\left\{\underbrace{d(x)}_{=2}, \underbrace{d(y) + \delta(y, x)}_{=3} \right\} = -2$$

ausgeführt, was zu dem Wert $d(x) = 1 = \Delta(x)$ führt. □

Korrektheit des Bellman-Ford Algorithmus. Wir machen uns nun die Korrektheit des Bellman-Ford Algorithmus klar. Zunächst ist klar, daß zu jedem Zeitpunkt während der Ausführung des Bellman-Ford Algorithmus der Wert $d(w)$ für die Kosten eines Pfads von v nach w stehen. Dabei gehen wir gedanklich von einem vollständigen Graphen mit „Pseudokanten“ (w, u) mit dem Gewicht $\delta(w, u) = +\infty$, falls (w, u) keine Kante des eigentlichen Graphen G ist. Weiter ist offensichtlich, daß die Werte $d(w)$ niemals verschlechtert werden. Wir halten diese Eigenschaften fest:

Lemma 6.1.17. Zu jedem Zeitpunkt während der Ausführung von Algorithmus 73 gilt für alle Knoten w :

- (a) $d(w) \geq \Delta(w)$
- (b) Sobald $d(w) = \Delta(w)$ gesetzt wird, so wird $d(w)$ in keiner der folgenden Iterationen verändert.

Zur Formulierung einer Schleifeninvariante des Bellman-Ford Algorithmus benötigen wir eine weitere Bezeichnung.

Bezeichnung 6.1.18 (Minimale Länge kürzester Wege). Ist w ein Knoten, für den es einen kürzesten Weg von v nach w gibt, so schreiben wir $\ell(w)$ für die minimale Länge kürzester Wege von v nach w . Der Begriff „Länge“ bezieht sich hier wie üblich auf die Kantenanzahl (und nicht auf die Gewichte der Kanten). Für $\Delta(w) \in \{+\infty, -\infty\}$ setzen wir $\ell(w) = +\infty$. \square

In Beispiel 6.1.3 auf Seite 311 liegen zwei kürzeste Pfade $\pi_1 = v, w, u$ und $\pi_2 = v, y, t, u$ von v nach u vor. Hier ist also $\ell(u) = |\pi_1| = 2$. (Dabei ist $|\pi_1|$ die Länge von π_1 , also die Anzahl an Kanten, die in π_1 durchlaufen werden.)

Lemma 6.1.19 (Schleifeninvariante für den Bellman-Ford Algorithmus). Unmittelbar nach dem i -ten Durchlauf der mittleren FOR-Schleife in Algorithmus 73 gilt:

$$d(w) = \Delta(w)$$

für alle Knoten w mit $\ell(w) \leq i$. Dabei ist $i \in \{0, 1, \dots, n-1\}$.

Beweis. Wir weisen die Aussage durch Induktion nach i nach. Für $i = 0$ ist nichts zu zeigen, da $\ell(w) = 0$ nur für $w = v$ möglich ist und da bereits im Initialisierungsschritt (erste FOR-Schleife) $d(v) = 0$ gesetzt wird. Andererseits ist auch $\Delta(v) = 0$, falls $\ell(v) = 0$, also falls v auf keinem Zyklus negativer Kosten liegt.

Wir nehmen nun an, daß $i \geq 1$. Aufgrund von Lemma 6.1.17 sowie der Induktionsvoraussetzung ist die Behauptung für $\ell(w) \leq i-1$ klar. Sei nun u ein Knoten mit $\ell(u) = i$ und sei $\pi = v, \dots, w, u$ ein kürzester Weg von v nach u der Länge i . Dann ist das Präfix $\pi' = v, \dots, w$ von π ein kürzester Weg von v nach w . π' hat die Länge $i-1$. Daher ist $\ell(w) = i-1$. Nach Induktionsvoraussetzung gilt: $d(w) = \Delta(w)$ nach der $(i-1)$ -ten Iteration in Algorithmus 73. Aufgrund von Lemma 6.1.17 wird dieser

Wert in der i -ten Iteration nicht verändert. Spätestens durch den Relaxationsschritt „ $d(u) := \min\{d(u), d(w) + \delta(w, u)\}$ “ erhält man

$$d(u) = d(w) + \delta(w, u) = \Delta(w) + \delta(w, u) = \delta(\pi) = \Delta(u).$$

Es ist auch möglich, daß $d(u) = \Delta(u)$ zu Beginn der i -ten Iteration gilt oder daß $d(u)$ durch einen anderen Relaxationsschritt „ $d(u) := \min\{d(u), d(x) + \delta(x, u)\}$ “ in der i -ten Iteration den Wert $\Delta(u)$ erhält. Für die Behauptung ist dies jedoch irrelevant. \square

Satz 6.1.20 (Korrekttheit und Kosten des Bellman-Ford Algorithmus). Nach der $(n - 1)$ -ten Iteration der äusseren FOR-Schleife in Algorithmus 73 gilt:

- (a) $d(w) = \Delta(w)$ für alle Knoten w , für die es einen kürzesten Pfad von v nach w gibt.
- (b) Es gibt genau dann eine Kante (w, u) in G , so daß $d(u) > d(w) + \delta(w, u)$, wenn es einen Zyklus negativer Kosten in G gibt, der von v erreichbar ist.

Die Laufzeit des Bellman-Ford Algorithmus ist $\Theta(n(n + m)) = \Theta(nm)$, wobei $n = |V|$ und $m = |E|$ und $m \geq n$ sowie eine Adjazenzlisten-Darstellung von G vorausgesetzt wird.

Beweis. Die Aussage über die Laufzeit ist offensichtlich.

Aussage (a) folgt sofort aus Lemma 6.1.19 mit $i = n - 1$ und der Beobachtung, daß $\ell(w) \leq n - 1$ für alle Knoten w , für die es einen kürzesten Pfad von v nach w gibt. Dies resultiert aus der Tatsache, daß es *einfache* kürzeste Wege zu jedem Knoten w mit $\Delta(w) \in \mathbb{R}$ gibt (Lemma 6.1.5).

Nun zum Nachweis von Aussage (b).

- Zunächst nehmen wir an, dass eine Kante (w, u) vorliegt, so dass $d(u) > d(w) + \delta(w, u)$ nach der $(n - 1)$ -ten Iteration. Zu zeigen ist die Existenz eines von v erreichbaren Zyklus mit negativen Kosten.

Die Beziehung $d(u) > d(w) + \delta(w, u)$ ist nur möglich, wenn $d(w) + \delta(w, u) < \infty$. Daher gilt

$$\Delta(u) \leq d(w) + \delta(w, u) < \infty.$$

Wäre $\Delta(u) > -\infty$, so wäre $-\infty < \Delta(u) < +\infty$ und somit $\ell(u) \leq n - 1$. Aus Aussage (a) würde dann folgen, dass $\Delta(u) = d(u) \leq d(w) + \delta(w, u)$ nach der $(n - 1)$ -ten Iteration. Widerspruch. Also ist $\Delta(u) = -\infty$. Aus Lemma 6.1.6 auf Seite 313 folgt die Existenz eines von v erreichbaren Zyklus in G , welcher negative Kosten hat.

- Wir nehmen nun an, dass ein Zyklus $\pi = v_0, \dots, v_k$ negativer Kosten vorliegt, wobei $v_0 \in Post^*(v)$. Wir zeigen, dass es einen Index $i \in \{1, \dots, k\}$ gibt, so dass $d(v_i) > d(v_{i-1}) + \delta(v_{i-1}, v_i)$ nach der $(n - 1)$ -ten Iteration. Hierzu nehmen wir an, dass

$$d(v_i) \leq d(v_{i-1}) + \delta(v_{i-1}, v_i) \text{ für } i = 1, \dots, k.$$

Dann ist

$$d(v_i) - d(v_{i-1}) \leq \delta(v_{i-1}, v_i), \quad i = 1, \dots, k.$$

Aufsummieren über alle i liefert:

$$\sum_{i=1}^k (d(v_i) - d(v_{i-1})) \leq \sum_{i=1}^k \delta(v_{i-1}, v_i) = \delta(\pi) < 0$$

Andererseits ist $v_k = v_0$ (da π ein Zyklus ist) und somit

$$\begin{aligned} \sum_{i=1}^k (d(v_i) - d(v_{i-1})) &= \sum_{i=1}^k d(v_i) - \sum_{i=1}^k d(v_{i-1}) \\ &= \sum_{i=1}^k d(v_i) - \sum_{i=0}^{k-1} d(v_i) \\ &= d(v_k) - d(v_0) = 0. \end{aligned}$$

Widerspruch. □

Die Berechnung kürzester Wege kann ähnlich wie im Dijkstra-Algorithmus erfolgen, indem der Relaxationsschritt um eine entsprechende Zuweisung „ $\text{father}(u) := w$ “ erweitert wird, sofern $d(u)$ gleich $d(w) + \delta(w, u)$ gesetzt wird.

Bemerkung 6.1.21 (Optimalitätsprinzip des Bellman-Ford Algorithmus). In der in Kapitel 5 behandelten Klassifikation von Entwurfsstrategien kann der Bellman-Ford Algorithmus dem dynamischen Programmieren zugeordnet werden. Dies erklärt sich daraus, dass unmittelbar nach der i -ten Iteration der mittleren FOR-Schleife die berechnete obere Schranken $d(w)$ durch die Werte

$$\Delta^i(w) = \min\{\delta(\pi) : \pi \text{ ist ein Pfad von } v \text{ nach } w \text{ der Länge } \leq i\} \in \mathbb{N} \cup \{+\infty\}$$

nach oben beschränkt sind. Also $\Delta(w) \leq d(w) \leq \Delta^i(w)$ für alle Knoten w nach der i -ten Iteration. Dies ist eine Verallgemeinerung von Lemma 6.1.19, die sich ebenfalls durch Induktion nach i nachweisen lässt. Das zugrundeliegende Optimalitätsprinzip ist

$$\Delta^{i+1}(u) = \min\{\Delta^i(u), \Delta^i(w) + \delta(w, u) : w \in \text{Pre}(u)\},$$

welches in der $(i+1)$ -ten Iteration durch die vorgenommenen Relaxationsschritte realisiert wird. Da in der angegebenen Formulierung des Bellman-Ford Algorithmus nicht mit den indizierten Δ 's gearbeitet wird, können die Werte $d(w)$ besser als $\Delta^i(w)$ sein. Wird nämlich Knoten w vor Knoten u in jedem Schleifendurchlauf betrachtet, so wird der Relaxationsschritt mit dem besseren Wert $\Delta^{i+1}(w)$ (oder einem der Werte $\Delta^j(w)$ für ein $j > i+1$) anstelle von $\Delta^i(w)$ ausgeführt. □

6.1.3 Algorithmus von Floyd

Bisher haben wir uns auf das single-source Problem konzentriert. In diesem Abschnitt geht es um die all-pairs Variante von Kürzeste-Wegeproblemen, wobei wir uns auf den gerichteten Fall beschränken. Gegeben ist also $G = (V, E)$ ein Digraph mit einer Gewichtsfunktion $\delta : E \rightarrow \mathbb{R}$, die eventuell manchen Kanten ein negatives Gewicht zuweist. Gesucht ist für alle Knotenpaare (v, w) ein kürzester Weg von v nach w , falls existent.

Mehrfaches Anwenden von single-source Algorithmen. Selbstverständlich kann der Bellman-Ford Algorithmus n -mal angewandt werden, um für jeden Knoten v kürzeste Wege zu allen anderen Knoten zu bestimmen. Die resultierenden Kosten sind $\Theta(n^2m)$, wobei $n = |V|$ und $m = |E|$. Sind die Kantengewichte stets nicht-negativ, dann können kürzeste Wege und deren Kosten durch mehrfaches Anwenden des Dijkstra-Algorithmus in kubischer Zeit $\Theta(n^3)$ oder $\Theta(mn \log n)$ berechnet werden.

All-pairs-Algorithmus (Floyd). Wir stellen nun den nach seinem Erfinder Floyd benannten Algorithmus vor, der beliebige Gewichtsfunktionen behandeln kann und die all-pairs Fragestellung explizit mit Hilfe der Methode des dynamischen Programmierens löst. Aus technischen Gründen ist es hilfreich, die gegebene Gewichtsfunktion $\delta : E \rightarrow \mathbb{R}$ zu einer Abbildung $\delta : V \times V \rightarrow \mathbb{R} \cup \{\infty\}$ zu erweitern, die jedem Knotenpaar ein Gewicht zuordnet. Dazu setzen wir

$$\delta(w, u) = \infty, \text{ falls } (w, u) \notin E.$$

Kantengewicht ∞ steht also gedanklich für eine unendliche teure Kante. Durch die Erweiterung von δ kann G als vollständiger Graph angesehen werden. Die Existenz von Pfaden zwischen je zwei Knoten ist damit gesichert, jedoch können deren Kosten ∞ sein, nämlich dann, wenn es keinen echten Pfad in dem ursprünglichen Graphen gibt, welcher die betreffenden Knoten verbindet.

Zur Vereinfachung betrachten wir zunächst nur die Kosten kürzester Wege. Das Ziel ist also die Berechnung der Werte

$$\Delta(v, w) = \inf \{ \delta(\pi) : \pi \text{ ist Pfad von } v \text{ nach } w \}$$

für alle $v, w \in V$, vorausgesetzt diese Werte sind $> -\infty$. Andernfalls hat G Zyklen negativer Kosten (vgl. Satz 6.1.8 auf Seite 314) und der Algorithmus soll mit einer entsprechenden „Fehlermeldung“ anhalten.

Die Grundidee des Algorithmus von Floyd besteht darin, eine Knotennumerierung v_1, \dots, v_n festzulegen und sukzessive die Werte

$$\begin{aligned} \Delta^i(v, w) &= \text{Kosten eines kürzesten einfachen Pfads von } v \text{ nach } w \text{ der Form} \\ &\quad \pi = v, u_1, \dots, u_k, w, \text{ wobei } k \geq 0 \text{ und } u_1, \dots, u_k \in D_i \end{aligned}$$

zu berechnen, wobei $D_i = \{v_1, \dots, v_i\}$.⁵⁵ Für $v = w$, also $\Delta^i(v, v)$, sind – neben dem trivialen Pfad v der Länge 0 (mit den Kosten 0) – einfache Zyklen der Form v, u_1, \dots, u_k, v mit $u_1, \dots, u_k \in D_i$ und $k \geq 0$, zu betrachten. Diese sind jedoch nur dann für das Minimum relevant, wenn deren Kosten negativ sind. Für beliebige Knoten v, w gilt stets $\Delta^i(v, w) > -\infty$, da wir über die einfachen Pfade von v nach w quantifizieren und es nur endlich viele einfache Pfade gibt. Offenbar sind

$$\Delta^n(v, w) = \Delta(v, w)$$

die Kosten eines kürzesten Wegs von v nach w , falls G keine Zyklen negativer Kosten besitzt.

Wie bereits in Abschnitt 5.5 (Seite 268 ff) erläutert, beruht das Konzept des dynamischen Programmierens für Optimierungsprobleme auf dem Optimalitätsprinzip. In unserem Fall ist zu zeigen, wie sich die Werte $\Delta^{i+1}(\dots)$ aus den Werten $\Delta^i(\dots)$ ergeben.

Lemma 6.1.22 (Optimalitätsprinzip für den Floyd-Algorithmus). Für alle $1 \leq i < n$ und $v, w \in V$:

$$\Delta^{i+1}(v, w) = \min \{ \Delta^i(v, w), \Delta^i(v, v_{i+1}) + \Delta^i(v_{i+1}, w) \}.$$

Abbildung 58 illustriert die Aussage von Lemma 6.1.22.

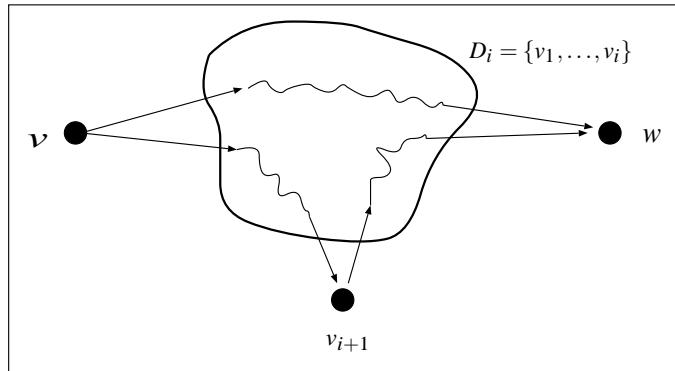


Abbildung 58: $\Delta^{i+1}(v, w) = \min \{ \Delta^i(v, w), \Delta^i(v, v_{i+1}) + \Delta^i(v_{i+1}, w) \}$

Beweis. Salopp formuliert beruht die Aussage von Lemma 6.1.22 auf der banalen Tatsache, dass ein kürzester Pfad v, u_1, \dots, u_k, w mit $u_1, \dots, u_k \in D_i$ entweder durch v_{i+1} als Zwischenknoten führt oder eben nicht. (Dabei bezeichnen wir die u_j 's als Zwischenknoten.) Im ersten Fall kann nur einer der Knoten u_j mit v_{i+1} übereinstimmen, da wir uns auf einfache Pfade konzentrieren.

⁵⁵Alle Hörerinnen und Hörer sollten nun ein déjà-vu Erlebnis haben: dieselbe Idee wurde bei der Konstruktion regulärer Ausdrücke aus endlichen Automaten angewandt. Jedoch reicht es hier, einfache Pfade zu betrachten während für die regulären Ausdrücke auch Zyklen berücksichtigt werden mussten.

Eine etwas formalere Argumentation könnte wie folgt aussehen. Sei $Pfade^i(v, w)$ die Menge aller einfachen Pfade (einfache Zyklen, falls $v = w$) von v nach w der Form

$$\pi = v, \underbrace{u_1, \dots, u_k}_{\in D_i}, w,$$

wobei $k \geq 0$ und $u_1, \dots, u_k \in D_i$. (Zur Erinnerung: G wurde gedanklich zu einem vollständigen Graphen erweitert. Daher sind die Pfadmengen $Pfade^i(v, w)$ nicht leer, auch dann, wenn $\Delta^i(v, w) = \infty$.) Dann ist

$$\Delta^i(v, w) = \min \left\{ \delta(\pi) : \pi \in Pfade^i(v, w) \right\}.$$

Die Pfadmenge $Pfade^{i+1}(v, w)$ zerfällt in die disjunkten Teilmengen $Pfade^i(v, w)$ und die Menge aller Pfade $\pi \in Pfade^{i+1}(v, w)$, welche v_{i+1} als Zwischenknoten passieren. Da wir uns auf *einfache* Pfade beschränken, können alle diese (einfachen) Pfade π in ein Anfangsstück $\pi_1 \in Pfade^i(v, v_{i+1})$ und ein Endstück $\pi_2 \in Pfade^i(v_{i+1}, w)$ zerlegt werden. Daher gilt:

$$\begin{aligned} & \Delta^{i+1}(v, w) \\ &= \min \left(\left\{ \delta(\pi) : \pi \in Pfade^i(v, w) \right\} \cup \right. \\ &\quad \left. \left\{ \delta(\pi_1) + \delta(\pi_2) : \pi_1 \in Pfade^i(v, v_{i+1}), \pi_2 \in Pfade^i(v_{i+1}, w) \right\} \right) \\ &= \min \left\{ \min \left\{ \delta(\pi) : \pi \in Pfade^i(v, w) \right\}, \right. \\ &\quad \left. \min \left\{ \delta(\pi_1) + \delta(\pi_2) : \pi_1 \in Pfade^i(v, v_{i+1}), \pi_2 \in Pfade^i(v_{i+1}, w) \right\} \right\} \\ &= \min \{ \Delta^i(v, w), \Delta^i(v, v_{i+1}) + \Delta^i(v_{i+1}, w) \} \end{aligned}$$

□

Lemma 6.1.22 bildet die Basis für die Anwendbarkeit des dynamischen Programmierens, da es sicherstellt, daß sich eine optimale Lösung für die „zulässigen“ Zwischenknoten v_1, \dots, v_{i+1} aus einer optimalen Lösung für die „zulässigen“ Zwischenknoten v_1, \dots, v_i ergibt. Die optimalen Teillösungen werden in Bottom-Up-Manier generiert: Beginnend mit der Matrix Δ^0 ergibt sich Δ^{i+1} aus der Matrix Δ^i .

Algorithmus 74 zeigt eine Formulierung, welche zunächst nur die Kosten kürzester Wege anstelle der kürzesten Wege selbst berechnet. Die finalen Werte $\Delta^n(v, w)$ stimmen jedoch nur dann mit $\Delta(v, w)$ überein, wenn G keine Zyklen negativer Kosten hat.

Beispiel 6.1.23. Wir veranschaulichen die Arbeitsweise des Floyd-Algorithmus an einem Beispiel. Die Funktionen $\Delta^i(\dots)$ stellen wir durch Matrizen dar.

FOR ALL Knoten v, w **DO**

IF $v \neq w$ **THEN**
 $\Delta^0(v, w) := \delta(v, w)$

ELSE
 $\Delta^0(v, v) := \min\{0, \delta(v, v)\}$

FI

OD

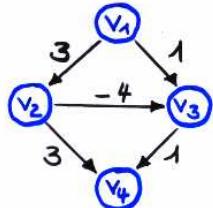
FOR ALL $i = 1, 2, \dots, n$ **DO**

FOR ALL Knoten v, w **DO**
 $\Delta^i(v, w) := \min\{\Delta^{i-1}(v, w), \Delta^{i-1}(v, v_i) + \Delta^{i-1}(v_i, w)\}$ (* Relaxationsschritt *)

OD

OD

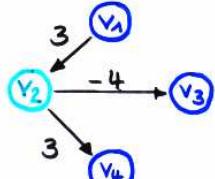
(* $\Delta^n(v, w) = \Delta(v, w)$, falls G keine Zyklen negativer Kosten hat. *)

Algorithmus von Floyd:

	v_1	v_2	v_3	v_4
v_1	0	3	1	∞
v_2	∞	0	-4	3
v_3	∞	∞	0	1
v_4	∞	∞	∞	0

$D_1 = \{v_1\}$: keine Veränderung

$D_2 = \{v_1, v_2\}$:



	v_1	v_2	v_3	v_4
v_1	0	3	-1	6
v_2	∞	0	-4	3
v_3	∞	∞	0	1
v_4	∞	∞	∞	0

Man beachte, dass $\Delta^0 = \Delta^1$, da v_1 keine einführende Kante hat. Bei der Berechnung von Δ^2 wird

$$\Delta^2(v_1, v_3) = \min\left\{\underbrace{\Delta^1(v_1, v_3)}_{=1}, \underbrace{\Delta^1(v_1, v_2) + \Delta^1(v_2, v_3)}_{=3-4}\right\} = -1$$

$$\Delta^2(v_1, v_4) = \min\left\{\underbrace{\Delta^1(v_1, v_4)}_{=\infty}, \underbrace{\Delta^1(v_1, v_2) + \Delta^1(v_2, v_4)}_{=3+3}\right\} = 6$$

gesetzt. In der dritten Iteration erhalten wir:

Matrix nach der
2. Iteration

	v_1	v_2	v_3	v_4
v_1	0	3	-1	6
v_2	∞	0	-4	3
v_3	∞	∞	0	1
v_4	∞	∞	∞	0

$D_3 = \{v_1, v_2, v_3\}$

	v_1	v_2	v_3	v_4
v_1	0	3	-1	0
v_2	∞	0	-4	-3
v_3	∞	∞	0	1
v_4	∞	∞	∞	0

$D_4 = \{v_1, v_2, v_3, v_4\} : \text{Keine Veränderung}$

Im letzten Schritt ergibt sich $\Delta^3 = \Delta^4$, da v_4 keine ausgehenden Kanten hat. \square

Folgendes Lemma klärt, wann der Algorithmus abzubrechen ist, weil ein Zyklus negativer Kosten vorliegt:

Lemma 6.1.24 (Abbruchkriterium). G hat genau dann einen Zyklus negativer Kosten, wenn es einen Index $i \in \{0, 1, \dots, n\}$ und einen Knoten v mit $\Delta^i(v, v) < 0$ gibt.

Beweis. Da die Werte $\Delta^i(v, w)$ stets für die Kosten eines Pfads von v nach w (hinsichtlich der Erweiterung von G zu einem vollständigen Digraphen) stehen, ist klar, dass gilt: Aus $\Delta^i(v, v) < 0$ folgt $\delta(\sigma) < 0$ für einen Zyklus der Form $\sigma = v, \dots, v$.

Liegt andererseits ein einfacher Zyklus $\sigma = w_0, w_1, \dots, w_k$ vor, wobei $\delta(\sigma) < 0$, $w_k = w_0$, und ist i der kleinste Index mit $\{w_0, \dots, w_{k-1}\} \subseteq D_i$, so ist $\Delta^i(w_j, w_j) < 0$ für alle Knoten w_j , die auf σ liegen. \square

Berechnung kürzester Wege. Wir berechnen Pfade $\pi^i(v, w) \in \text{Pfade}^i(v, w)$ mit den Kosten $\delta(\pi^i(v, w)) = \Delta^i(v, w)$ wie folgt:

- $\pi^0(v, v) = v$ (Pfad der Länge 0)
- $\pi^0(v, w) = v, w$ (Pfad der Länge 1), falls $(v, w) \in E$, $v \neq w$.

- Sei $i \geq 0$. Ist $\Delta^{i+1}(v, w) = \Delta^i(v, w)$, so ist $\pi^i(v, w)$ ein kürzester Pfad von v nach w , der nur durch Knoten in D_{i+1} führt, abgesehen vom Anfangs- und Endknoten v bzw. w . Wir können also $\pi^{i+1}(v, w) = \pi^i(v, w)$ setzen. Andernfalls, also wenn

$$\Delta^{i+1}(v, w) = \Delta^i(v, v_{i+1}) + \Delta^i(v_{i+1}, w) < \Delta^i(v, w),$$

so führt jeder kürzeste Pfad $\pi \in \text{Pfade}^{i+1}(v, w)$ durch den Knoten v_{i+1} . Wir können

$$\pi^{i+1}(v, w) = \pi^i(v, v_{i+1}) \circ \pi^i(v_{i+1}, w)$$

setzen. Dabei steht das Symbol „ \circ “ für das Aneinanderhängen von Pfaden.

Im Algorithmus genügt es, sich für jedes Knotenpaar (v, w) den letzten Knoten $P(v, w) = v_{i+1}$ zu merken, für welchen der Wert $\Delta^{i+1}(v, w)$ zu $\Delta^i(v, v_{i+1}) + \Delta^i(v_{i+1}, w)$ gesetzt wurde. Siehe Algorithmus 75 auf Seite 335. Ein kürzester Wege $\pi^n(v, w)$ von v nach w kann dann rekursiv konstruiert werden. Nach dem n -ten Durchlauf der FOR-Schleife ist dann $P(v, w) = v_j$, wobei j der grösste Index ist, so dass ein kürzester Pfad von v nach w durch Knoten v_j als Zwischenknoten führt. Ist dann z.B. $P(v, v_j) = v_k$, $P(v_j, w) = v_\ell$ und $P(v, v_k) = v$, so hat der gesuchte Pfad die Struktur

$$v, v_k, \dots, v_j, \dots, v_\ell, \dots, w.$$

Ferner kann auf die Indizierung Δ^i verzichtet werden, da man im Relaxationsschritt

$$\Delta^i(v, w) := \min\{\Delta^{i-1}(v, w), \Delta^{i-1}(v, v_i) + \Delta^{i-1}(v_i, w)\}$$

ebenso mit den neuen, besseren Werten $\Delta^i(\cdot)$ (sofern vorliegend) rechnen kann.

Die Laufzeit des Verfahrens ist offenbar kubisch in der Knotenzahl. Der Speicherplatzbedarf ist quadratisch in der Knotenzahl. Wir fassen die Ergebnisse zusammen:

Satz 6.1.25 (Kosten des Floyd Algorithmus). Der Algorithmus von Floyd berechnet kürzeste Wege und deren Kosten bzw. erkennt die Existenz von Zyklen negativer Kosten in Zeit $\Theta(n^3)$ und Platz $\Theta(n^2)$.

Vergleich der Algorithmen von Dijkstra, Bellman-Ford und Floyd. Zunächst stellen wir fest, dass die beiden single source Algorithmen sich vom Floyd-Algorithmus in der Art des angewandten Relaxationsschritts unterscheiden, denen die Ungleichungen

- $\Delta(v, u) \leq \Delta(v, w) + \delta(w, u)$ für den Dijkstra- und Bellman-Ford-Algorithmus,
- $\Delta(v, u) \leq \Delta(v, w) + \Delta(w, u)$ für den Floyd-Algorithmus

unterliegt. Damit verbunden ist auch die Auswahl der Knotenpaare für den Relaxationsschritt ein Unterscheidungsmerkmal.

Während der Bellman-Ford- sowie der Floyd-Algorithmus auf dem Konzept des dynamischen Programmierens beruhen, verwendet der Dijkstra-Algorithmus eine Greedy-Strategie. Die Gegenüberstellung Dijkstra–Floyd ist daher ähnlich wie der Vergleich des Dijkstra- mit dem Bellman-Ford-Algorithmus. Der Dijkstra-Algorithmus arbeitet

Algorithmus 75 Algorithmus von Floyd

FOR ALL Knoten v, w **DO**

IF $v \neq w$ **THEN**

$\Delta(v, w) := \delta(v, w);$

$P(v, w) := v$

ELSE

IF $\delta(v, v) < 0$ **THEN**

return „es existiert ein Zyklus negativer Kosten“ (* vorzeitiger Abbruch *)

ELSE

$\Delta(v, v) := 0$

FI

FI
OD

FOR ALL $i = 1, 2, \dots, n$ **DO**

FOR ALL Knoten v **DO**

FOR ALL Knoten w **DO**

IF $\Delta(v, w) < \Delta(v, v_i) + \Delta(v_i, w)$ **THEN**

$\Delta(v, w) := \Delta(v, v_i) + \Delta(v_i, w);$ (*) kürzester bisher gefundener *)

$P(v, w) := v_i$ (*) Pfad von v nach w führt durch v_i *)

FI

OD

IF $\Delta(v, v) < 0$ **THEN**

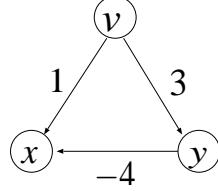
return „es existiert ein Zyklus negativer Kosten“ (* vorzeitiger Abbruch *)

FI

OD

OD

mit einer vom Startknoten v abhängigen Knotenmenge $D_i = Done$, von der garantiert werden konnte, daß die korrekten Werte $\Delta(w) = \Delta^i(v, w)$ für $w \in D_i$ bereits vorliegen. Im Gegensatz hierzu ist der Floyd-Algorithmus ein klassischer Repräsentant des dynamischen Programmierens und erlaubt die Korrektur getroffener Entscheidungen, indem Δ^i zu Δ^{i+1} „verbessert“ wird. Wir demonstrieren dies am Beispiel eines Graphen mit negativen Kantengewichten, für welchen die Greedy-Methode fehlschlägt:



Für Startknoten v berechnet der Dijkstra-Algorithmus $d(x) = 1$ und revidiert diese Entscheidung nicht mehr. Der Floyd-Algorithmus berechnet für die Knotennummerierung $v = v_1, x = v_2, y = v_3$ folgende Werte: $\Delta^1(v, y) = 3$, $\Delta^1(v, x) = 1$, $\Delta^2 = \Delta^1$ und $\Delta^3(v, x) = \min\{1, 3 - 4\} = -1 = \Delta(v, x)$.

6.1.4 Algorithmus von Warshall

Wir betrachten eine Variante des oben beschriebenen Wegeproblems und Floyd-Algorithmus, in welcher lediglich nach der Existenz von Wegen, nicht aber nach den kürzesten Wegen oder deren Kosten, gefragt ist.

Formal haben wir es mit folgender Fragestellung zu tun. Gegeben ist eine endliche Menge V und eine binäre Relation R auf V , also $R \subseteq V \times V$. (Man kann R als gerichtete Kantenmenge auffassen.) Gesucht ist die reflexive Hülle R^* von R .

Bemerkung 6.1.26 (Zur Erinnerung: Reflexive, transitive Hülle). Sei V eine Menge und $R \subseteq V \times V$. R heißt *reflexiv*, falls $(v, v) \in R$ für alle $i \in \{1, \dots, n\}$. R heißt *transitiv*, falls aus $(v, w) \in R$ und $(w, u) \in R$ folgt $(v, u) \in R$. Die reflexive transitive Hülle von R ist die kleinste binäre Relation R^* , welche reflexiv und transitiv ist und R enthält. Beispielsweise entspricht die Erreichbarkeitsrelation in einem Digraphen der reflexiven transitiven Hülle der Kantenrelation. \square

Ist z.B. $V = \{1, 2, 3, 4\}$ und $R = \{(1, 2), (2, 3), (1, 4), (4, 1)\}$, so ist

$$R^* = R \cup \{(1, 1), (2, 2), (3, 3), (4, 4)\} \cup \{(1, 3), (4, 2), (4, 3)\}.$$

Im Folgenden nehmen wir $V = \{1, \dots, n\}$ an und betrachten den Digraphen $G = (V, R)$. Die reflexive Hülle R^* stimmt mit der Menge aller Erreichbarkeitspaare (i, j) mit $j \in Post^*(i)$ überein. Wendet man den Algorithmus von Floyd auf G an, wobei wir die Einheitsgewichtsfunktion $\delta(i, j) = 1$ für alle $(i, j) \in R$ zugrundelegen, dann erhält man $R^* = R^n$, wobei

$$R^k = \{(i, j) : \Delta^k(i, j) < \infty\}, \quad k = 0, 1, \dots, n.$$

Diese Vorgehensweise ist jedoch unnötig aufwendig, da es genügt, die Bitvektor-Darstellung der Relationen R^k zu berechnen. Weder die Werte $\Delta^k(i, j)$ noch die kürzesten Wege

werden benötigt. Im Folgenden fassen wir die Werte $R^k(i, j)$ als Boolesche Ausdrücke auf, die genau dann den Wahrheitswert true haben, wenn $(i, j) \in R^k$. Offenbar gilt: $R^0(i, i) = \text{true}$ und $R^0(i, j) = R(i, j)$ für $i \neq j$. Für $1 \leq k \leq n$:

$$R^k(i, j) = R^{k-1}(i, j) \vee (R^{k-1}(i, k) \wedge R^{k-1}(k, j)).$$

Diese Formel wird im Algorithmus von Warshall (siehe Algorithmus 76 auf Seite 337) verwendet.

Algorithmus 76 Algorithmus von Warshall

```

FOR  $i = 1, \dots, n$  DO
  FOR  $j = 1, \dots, n$  DO
    IF  $i \neq j$  THEN
       $R^*(i, j) := R(i, j);$ 
    ELSE
       $R^*(i, i) := \text{true};$ 
    FI
  OD
OD
FOR ALL  $k = 1, 2, \dots, n$  DO
  FOR  $i = 1, \dots, n$  DO
    FOR  $j = 1, \dots, n$  DO
      IF  $\neg R^*(i, j)$  THEN
         $R^*(i, j) := R^*(i, k) \wedge R^*(k, j)$ 
      FI
    OD
  OD
OD
```

Satz 6.1.27 (Kosten des Warshall-Algorithmus). Die Laufzeit des Algorithmus von Warshall ist (unter dem uniformen und logarithmischen Kostenmaß) $\Theta(n^3)$, der Platzbedarf ist $\Theta(n^2)$.

6.2 Algorithmen für ungerichtete Graphen

In diesem Abschnitt beschäftigen wir uns mit einigen Algorithmen (u.a. Zyklentest, aufspannende Bäume) für ungerichtete Graphen.

6.2.1 Ungerichtete Bäume

Zunächst erinnern wir nochmal an den Begriff des Zusammenhangs. Eine Knotenmenge C eines ungerichteten Graphs $G = (V, E)$ heißt **zusammenhängend**, falls je zwei $v, w \in C$ voneinander erreichbar sind (d.h. $w \in \text{Post}^*(v)$ und $v \in \text{Post}^*(w)$). C heißt **Zusammenhangskomponente** von G , falls C eine nicht leere, maximale, zusammenhängende

Knotenmenge ist. G heißt *zusammenhängend*, falls V zusammenhängend ist, also wenn es genau eine Zusammenhangskomponente gibt. In Lemma 3.2.8 auf Seite 81 hatten wir gesehen, daß ein ungerichteter zusammenhängender Graph mit n Knoten mindestens $n - 1$ Kanten hat. Insbesondere enthält jede Zusammenhangskomponente C (genauer der zugehörige Teilgraph) mindestens $|C| - 1$ Kanten.⁵⁶ In folgendem Lemma weisen wir $n - 1$ als obere Schranke für die Kantenanzahl in zyklischen ungerichteten Graphen nach.

Lemma 6.2.1. Sei $G = (V, E)$ ein ungerichteter Graph mit $|V| = n$ und $|E| = m$. Dann gilt: Ist $m \geq n \geq 3$, so ist G zyklisch.

Beweis. Wir beweisen die Aussage durch Induktion nach $n = |V|$. Der Induktionsanfang ist $n = 3$. Ist $m \geq n = 3$, so besteht der vorliegende Graph offenbar aus einem Dreieck und ist somit zyklisch.

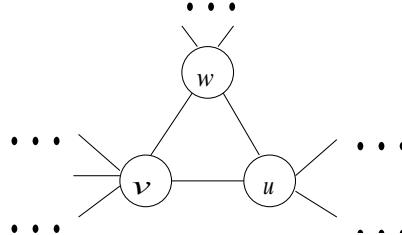
Sei nun $m \geq n \geq 4$. Ferner sei v ein beliebiger Knoten in G .

1. Fall: $|Post(v)| \leq 1$, also entweder $Post(v) = \emptyset$ oder $|Post(v)| = 1$.

Wir betrachten den Graphen $G' \setminus \{v\}$, der aus v entsteht, in dem v sowie eventuell die von v ausgehende Kante gestrichen wird. Der Graph $G' \setminus \{v\}$ besteht aus genau $n - 1$ Knoten und m oder $m - 1$ Kanten, je nachdem, ob $Post(v)$ leer oder eelementig ist. Da $m \geq n$ vorausgesetzt wird, ist die Kantenanzahl in $G' \setminus \{v\}$ größer oder gleich der Knotenanzahl. Nach Induktionsvoraussetzung ist $G' \setminus \{v\}$ und somit auch G zyklisch.

2. Fall: $|Post(v)| = k \geq 2$. Wir unterscheiden nun zwei Unterfälle:

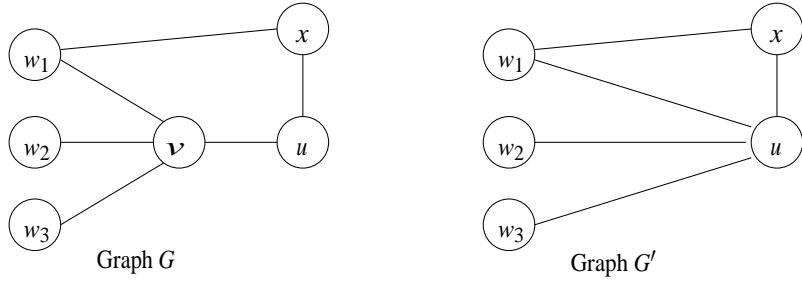
Fall 2.1 Es existieren Knoten $w, u \in Post(v)$ mit $(w, u) \in E$. Dann ist v, w, u, v ein Zyklus in G .



Fall 2.2 Für je zwei Knoten $w, u \in Post(v)$ gilt $(w, u) \notin E$.

Im Folgenden sei $u \in Post(v)$ ein fest gewählter Nachfolger von v und $Post(v) \setminus \{u\} = \{w_1, \dots, w_{k-1}\}$. Wir entfernen nun Knoten v und ersetzen die Kante $(v, u) \in E$ durch $k - 1$ Kanten (w_i, u) , $i = 1, \dots, k - 1$. Ferner entfernen wir die Kanten (w_i, v) , $i = 1, \dots, k - 1$. Sei G' der resultierende Graph. Folgende Skizze illustriert den Übergang von G zu G' an einem Beispiel:

⁵⁶Ist $G = (V, E)$ ein ungerichteter Graph und $C \subseteq V$, dann wird $G_C = (C, E_C)$ mit $E_C = E \cap (C \times C)$ der durch C induzierte Teilgraph von G genannt. Offenbar ist G_C zusammenhängend, falls C eine Zusammenhangskomponente von G ist. Somit ist $|C| - 1 \leq |E_C|$.



Die Knotenanzahl in G' ist $n - 1$, die Kantenanzahl in G' ist

$$m - \underbrace{1}_{\text{Kante } (v, u)} - \underbrace{(k - 1)}_{\text{Kanten } (w_i, v)} + \underbrace{(k - 1)}_{\text{Kanten } (w_i, u)} = m - 1.$$

Nach Induktionsvoraussetzung ist G' zyklisch. Falls G' einen Zyklus enthält, der durch keine der neuen Kanten (w_i, u) führt, so ist dies zugleich ein Zyklus in G . Andernfalls betrachten wir einen einfachen Zyklus in G' , der eine oder mehrere der Kanten (w_i, u) enthält. Wir ersetzen jede dieser Kanten (w_i, u) durch die beiden Kanten (w_i, v) und (v, u) und erhalten somit einen Zyklus in G . (Wird die Kante zwischen w_i und u in der Orientierung „zuerst u , dann w_i “ durchlaufen, so ist diese durch $(u, v), (v, w_i)$ zu ersetzen.)

□

Aus Lemma 6.2.1 (siehe oben) und Lemma 3.2.8 auf Seite 81 folgt:

Corollar 6.2.2 (Kantenanzahl zusammenhängender, azyklischer Graphen). Sei G ein ungerichteter Graph mit n Knoten, wobei $n \geq 1$. Dann gilt:

Ist G zusammenhängend und azyklisch, so ist $|E| = n - 1$.

Da die Zusammenhangskomponenten eines azyklischen Graphen selbst wieder azyklisch sind⁵⁷ und somit die Voraussetzung von Corollar 6.2.2 erfüllen, erhalten wir:

Corollar 6.2.3 (Kantenanzahl azyklischer Graphen). Jeder ungerichtete azyklische Graph mit n Knoten und k Zusammenhangskomponenten hat $n - k$ Kanten.

Beweis. Für $n = 0$ ist $k = 0$ und $|E| = 0 = 0 - 0 = n - k$. Für $n \geq 1$ ist die Argumentation wie folgt. Seien C_1, \dots, C_k die Zusammenhangskomponenten des gegebenen ungerichteten azyklischen Graphen G . Weiter sei $n_i = |C_i|$ die Knotenanzahl von C_i und m_i die Anzahl an Kanten, die zwischen den C_i -Knoten verlaufen. Dann gilt $n = n_1 + \dots + n_k$ und $m_i = n_i - 1$, da C_i (genauer: der durch C_i induzierte Teilgraph von G) zusammenhängend und azyklisch ist. Also ist die Anzahl an Kanten in G gleich

$$m_1 + \dots + m_k = (n_1 - 1) + \dots + (n_k - 1) = \underbrace{(n_1 + \dots + n_k)}_{=n} - k = n - k.$$

□

⁵⁷Wir identifizieren hier jede Zusammenhangskomponente C mit dem induzierten Teilgraphen $G_C = (C, E_C)$, wobei $E_C = E \cap (C \times C)$.

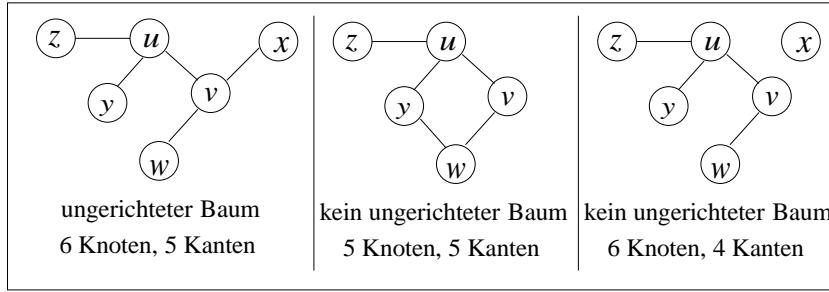


Abbildung 59: Beispiele zu ungerichteten Bäumen

Wir kommen nun zu dem ungerichteten Pendant von gerichteten Bäumen. Ungerichtete Bäume haben keine Wurzel, stattdessen sind sie durch die in Satz 6.2.5 (siehe unten) angegebenen Eigenschaften charakterisiert.

Definition 6.2.4 (Ungerichteter Baum). Jeder ungerichtete Graph, der zusammenhängend und zyklenfrei ist, wird ungerichteter Baum genannt. \square

Abbildung 59 gibt einige Beispiele. In Analogie zum gerichteten Fall wird insbesondere auch der leere Graphen als ungerichteter Baum zugelassen.

Satz 6.2.5 (Charakterisierungen ungerichteter Bäume). Sei $G = (V, E)$ ein ungerichteter Graph mit $|V| = n \geq 1$. Dann sind folgende Aussagen äquivalent:

- (a) G ist ein ungerichteter Baum (d.h. zusammenhängend und zyklenfrei).
- (b) G ist zyklenfrei und $|E| = n - 1$.
- (c) G ist zusammenhängend und $|E| = n - 1$.
- (d) Zu jedem Knotenpaar (v, w) gibt es genau einen einfachen Pfad von v nach w .

Beweis. (a) \Rightarrow (b) und (a) \Rightarrow (c) ergeben sich aus Corollar 6.2.2.

(b) \Rightarrow (a) folgt aus Corollar 6.2.3. Ist nämlich k die Anzahl an Zusammenhangskomponenten in G , so ist

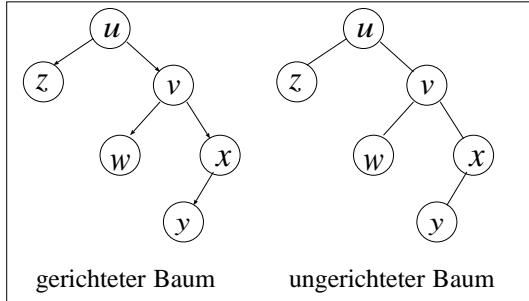
$$n - 1 \stackrel{(b)}{=} |E| \stackrel{\text{Corollar 6.2.3}}{=} n - k$$

und somit $k = 1$. Daher ist G zusammenhängend.

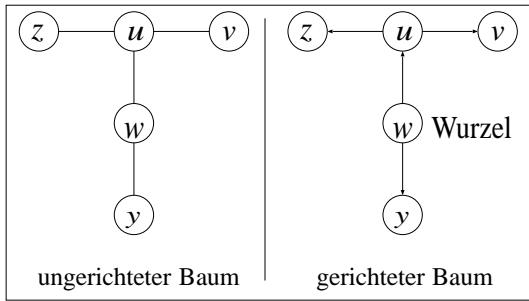
(c) \Rightarrow (a): Sei G zusammenhängend und $|E| = n - 1$. Wäre G zyklisch, so könnte eine beliebige Kante eines Zyklus entfernt werden, ohne den Zusammenhang zu zerstören. Der resultierende, zusammenhängende Graph hätte n Knoten und $n - 2$ Kanten, was aufgrund der Aussage von Lemma 3.2.8 auf Seite 81 nicht möglich ist.

Der Nachweis der Äquivalenz von (d) und (a)-(c) wird als Übungsaufgabe gestellt. \square

Gerichtete versus ungerichtete Bäume. Der Zusammenhang zwischen gerichteten und ungerichteten Bäumen ist wie folgt. Jedem gerichteten Baum liegt ein ungerichteter Baum zugrunde.



Umgekehrt kann man in jedem ungerichteten Baum die Kanten so richten, daß ein gerichteter Baum entsteht. Dies ist sogar dann möglich, wenn man den Wurzelknoten fest vorgibt.



6.2.2 Zyklentest in ungerichteten Graphen

Wir stellen nun einen Algorithmus vor, welcher für gegebenen ungerichteten Graphen in Zeit $\Theta(n)$ die Zykliefreiheit prüft. Wir arbeiten mit einer Modifikation der Tiefensuche, wobei wir für jeden besuchten Knoten u die Information, über welche Kante besucht wird, speichern. Etwa

$$\text{father}(u) = w, \text{ falls } u \text{ wird über die Kante } (w, u) \text{ besucht}$$

wobei wir für diejenigen Knoten v , für welche die Traversierung gestartet wird, einen Spezialwert verwenden müssen, z.B. $\text{father}(v) = \perp$. Sobald wir bei der Expansion von v auf einen bereits besuchten direkten Nachfolger w von v stoßen, so daß $\text{father}(v) \neq w$, dann ist der vorliegende Graph zyklisch. Diese Ideen sind in Algorithmus 78 zusammengefasst. Die Initialisierung ist in Algorithmus 77 angegeben.

Korrektheit. Wir nehmen zunächst an, dass G zyklisch ist und zeigen, dass Algorithmus 77 „Zyklus gefunden“ ausgibt.

Sei $\pi = v_0, v_1, \dots, v_r$ ein einfacher Zyklus in G , so dass v_0 in der Tiefensuche nach den Knoten v_1, \dots, v_{r-1} besucht wird. Es gilt also $v_0 = v_r$, $r \geq 3$ und v_0, v_1, \dots, v_{r-1} sind

Algorithmus 77 Zyklentest in ungerichteten Graphen

Visited := \emptyset ; (* Kantenzähler := 0 *)
FOR ALL $v \in V$ **DO**
 IF $v \notin \text{Visited}$ **THEN**
 $\text{father}(v) := \perp$; (* v ist DFS-Startknoten *)
 IF $\text{Zyklus}(v)$ **THEN**
 return „Zyklus gefunden“
 FI
 FI
 OD
 return „der Graph ist zyklenfrei“

Algorithmus 78 $\text{Zyklus}(v)$

Visited := $\text{Visited} \cup V$;
FOR ALL $w \in \text{Post}(v)$ **DO** (* Schleifeninvariante: Kantenzähler $\leq 2n$ *)
 IF $\text{father}(v) \neq w$ and $w \in \text{Visited}$ **THEN**
 return „true“
 ELSE
 IF $w \notin \text{Visited}$ **THEN**
 $\text{father}(w) := v$;
 IF $\text{Zyklus}(w)$ **THEN** return „true“ **FI**
 FI
 FI
 (* Kantenzähler := Kantenzähler + 1 *)
OD
 return „false“ (* Kein Zyklus gefunden *)

paarweise verschieden. Weiter nehmen wir an, dass Algorithmus 77 bis zur Ausführung von $Zyklus(v_0)$ gelangt und nicht vorher einen anderen Zyklus findet. Wir betrachten nun die Knoten v_{r-1} und v_1 . Dann gilt $v_{r-1}, v_1 \in Post(v_0)$ und $v_{r-1} \neq v_1$. Aus Symmetriegründen können wir annehmen, dass $father(v_0) \neq v_{r-1}$. Andernfalls betrachten wir den Zyklus v_r, v_{r-1}, \dots, v_0 anstelle von v_0, v_1, \dots, v_r . Bei der Ausführung von $Zyklus(v_0)$ ist die Bedingung

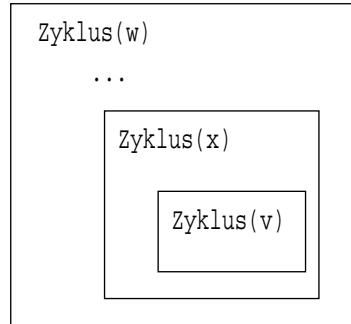
$$\text{„}father(v_0) \neq v_{r-1} \text{ und } v_{r-1} \in Visited\text{“}$$

erfüllt. Also gibt $Zyklus(v_0)$ „true“ zurück, was letztendlich zur Ausgabe „Zyklus gefunden“ in Algorithmus 77 führt.

Wir nehmen nun umgekehrt an, daß Algorithmus 77 mit der Antwort „Zyklus gefunden“ terminiert und zeigen, dass G zyklisch ist. Dann gibt es Knoten w und v , so dass die Inspektion der Kante (v, w) in $Zyklus(v_0)$ zur Ausgabe von „Zyklus gefunden“ führte, also $w \in Post(v)$, $father(v) \neq w$ und $w \in Visited$ während der Inspektion der Kante (v, w) innerhalb der Ausführung von $Zyklus(v)$.

1. Fall: $Zyklus(w)$ wurde vor $Zyklus(v)$ aufgerufen.

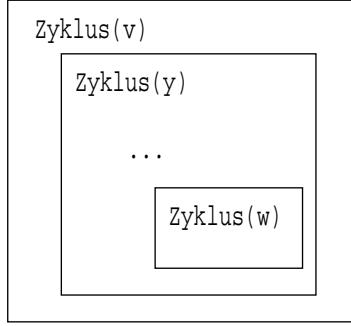
Da v von w erreichbar ist und da innerhalb $Zyklus(w)$ alle von w erreichbaren Knoten besucht werden, die zuvor noch nicht besucht wurden, findet der Aufruf von $Zyklus(v)$ innerhalb $Zyklus(w)$ statt. Sei $x = father(v)$ derjenige Knoten, so dass v über die Kante (x, v) besucht wird. Dann ist $x \in Post^*(w)$.



Da $father(v) \neq w$ ist $x \neq w$. Insbesondere gibt es einen Zyklus der Form w, \dots, x, v, w .

2. Fall: $Zyklus(w)$ wurde nach $Zyklus(v)$ aufgerufen.

Da w von v erreichbar ist, wird $Zyklus(w)$ innerhalb von $Zyklus(v)$ aufgerufen. Dann gibt es einen Knoten $y \in Post(v)$, so dass $father(y) = v$ und $Zyklus(w)$ innerhalb $Zyklus(y)$ aufgerufen wird. Beachte, dass w nicht über die Kante (v, w) besucht werden kann, also $father(w) \neq v$, da andernfalls die Bedingung $w \in Visited$ während der Inspektion der Kante (v, w) in $Zyklus(v)$ nicht erfüllt sein könnte.



In diesem Fall gibt es einen Zyklus der Form v, y, \dots, w, v .

Laufzeit. Wir diskutieren nun die Kosten des beschriebenen DFS-basierten Zyklentests, wobei $n = |V|$ die Anzahl an Knoten und $m = |E|$ die Kantenanzahl ist. Zunächst ist klar, daß $\mathcal{O}(n + m)$ eine obere Schranke ist, da $\Theta(n + m)$ die Laufzeit einer vollständig durchgeführten Tiefensuche ist. Tatsächlich bricht der Algorithmus aber bereits nach $\mathcal{O}(n)$ Schritten ab. Der Grund hierfür ist, daß spätestens bei der $2n$ -ten Ausführung der äusseren IF-Abfrage in $Zyklus(\cdot)$ ein Zyklus erkannt wird. Dies liegt an der Tatsache, daß bis zu diesem Zeitpunkt mindestens n Kanten inspiziert wurden (möglicherweise einige davon doppelt, da jede Kante (v, w) in den Adjazenzlisten von v und w eingetragen ist) und jeder ungerichtete Graph mit n Kanten und Knoten einen Zyklus enthält. Folgender Satz fasst die Ergebnisse zusammen:

Satz 6.2.6 (Kosten des Zyklentests in ungerichteten Graphen). Ist die Adjazenzlisten-Darstellung eines ungerichteten Graphen G mit n Knoten gegeben, so läßt sich in $\mathcal{O}(n)$ Schritten prüfen, ob G azyklisch ist.

Entsprechendes gilt für die Frage, ob G ein ungerichteter Baum ist. Mit der Tiefensuche kann in Zeit $\mathcal{O}(n)$ Bedingung (b) oder (c) aus Satz 6.2.5 (Seite 340) geprüft werden.

6.2.3 Minimale aufspannende Bäume

Unter einem aufspannenden Baum für einen ungerichteten Graphen G versteht man einen ungerichteten Baum, der durch Streichen von Kanten aus G entsteht.

Definition 6.2.7 (Aufspannender Baum). Sei $G = (V, E)$ ein ungerichteter Graph. Ein aufspannender Baum für G ist ein ungerichteter Baum $\mathcal{T} = (V, E_{\mathcal{T}})$, wobei $E_{\mathcal{T}} \subseteq E$. \square

Offenbar gilt: Es gibt genau dann einen aufspannenden Baum für G , wenn G zusammenhängend ist. Ein solcher läßt sich mit einer Tiefen- oder Breitensuche bestimmen. Man spricht auch von DFS- bzw. BFS-Bäumen.⁵⁸ Die Baumkanten sind jeweils solche Kanten (v, w) , so dass w in der Tiefen- bzw. Breitensuche über die Kante (v, w) besucht wird. Z.B. ergeben sich die aufspannenden Bäume in Abbildung 60 auf Seite 345 durch eine

⁵⁸Für nicht-zusammenhängende, ungerichtete Graphen ergeben sich durch die Tiefen- und Breitensuche Wälder (sogenannte DFS- bzw. BFS-Wälder) bestehend aus ungerichteten Bäumen, die jeweils die Knoten einer Zusammenhangskomponente erfassen.

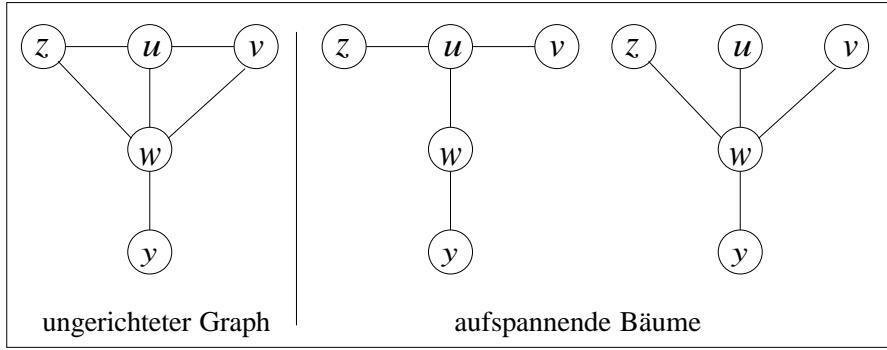


Abbildung 60: Beispiele zu aufspannenden Bäumen

Breitensuche gestartet mit Knoten u (links) bzw. w (rechts). Der linke aufspannende Baum kann ebenfalls durch eine Tiefensuche gestartet mit Knoten y entstanden sein, nämlich dann, wenn u der erste Knoten in der Adjazenzliste von w ist. Steht w vor z und v in der Adjazenzliste von u , so ergibt sich der rechte aufspannende Baum durch eine Tiefensuche gestartet mit u .

Wir betrachten nun aufspannende Bäume für zusammenhängende Graphen, deren Kanten mit Kosten (Gewichten) assoziiert sind. Diese sind z.B. im Kontext der Frage nach einem günstigsten Versorgungsnetz (Telefon, Gasleitungen, Bahnlinien, etc.) relevant. Tatsächlich greifen manche Telefongesellschaften auf das unten vorgestellte Verfahren für die Berechnung minimaler aufspannender Bäume zur Berechnung der Telefongebühren zurück.

Definition 6.2.8 (Minimaler aufspannender Baum (MST)). Sei $G = (V, E)$ ein ungerichteter zusammenhängender Graph und $\delta : E \rightarrow \mathbb{R}$ eine Gewichtsfunktion. Weiter sei $\mathcal{T} = (V, E_{\mathcal{T}})$ ein aufspannender Baum für G . Die Kosten von \mathcal{T} bzgl. δ sind

$$cost(\mathcal{T}) = \sum_{e \in E_{\mathcal{T}}} \delta(e).$$

\mathcal{T} heißt minimaler aufspannender Baum für G bezüglich δ , falls $cost(\mathcal{T})$ minimal ist unter allen aufspannenden Bäumen für G . Häufig verwenden wir die gebräuchliche Abkürzung MST für die englische Bezeichnung „minimal spanning tree“. \square

Abbildung 61 zeigt links einen ungerichteten, zusammenhängenden Graphen mit einer Kostenfunktion und rechts zwei aufspannende Bäume. Für den linken aufspannenden Baum betragen die Kosten $3+1+2+3+4 = 13$, für den rechten $3+1+2+5+7=18$.

6.2.4 Der Algorithmus von Prim

Wir stellen nun zwei Greedy-Algorithmen zur MST-Bestimmung vor. Der Ausgangspunkt ist jeweils ein ungerichteter, zusammenhängender Graph $G = (V, E)$ mit einer Kostenfunktion $\delta : E \rightarrow \mathbb{R}$. Wir beginnen mit dem Algorithmus von Prim, der mit einer

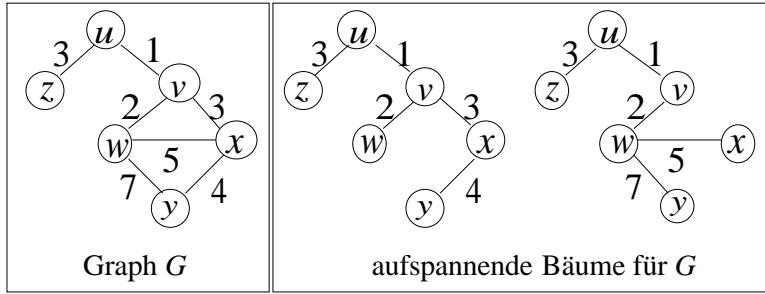


Abbildung 61: Beispiel für einen Graphen und aufspannende Bäume

Kantenmenge E' arbeitet, die zu jedem Zeitpunkt Baumstruktur (für einen Teilgraphen von G) hat. Ist die aktuelle Kantenmenge E' und ist U die Menge der Knoten, welche auf mindestens einer Kante in E' liegen, so wird E' um eine Kante (u, v) erweitert, die einen Knoten $u \in U$ mit einem Knoten in $v \in V \setminus U$ verbindet und die minimale Kosten unter all diesen Kanten hat, also

$$\delta(u, v) = \min_{(u', v') \in E \cap (U \times (V \setminus U))} \delta(u', v').$$

Die wesentlichen Schritte dieser Vorgehensweise zur Konstruktion eines MST sind in Algorithmus 79 auf Seite 346 angegeben.

Algorithmus 79 Algorithmus von Prim zur MST-Konstruktion

wähle einen beliebigen Knoten $u_0 \in V$ und setze $U := \{u_0\}$;
 $E' := \emptyset$;

(* Schleifeninvariante: (U, E') ist ein ungerichteter Baum, welcher *)
 (* zu einem MST ergänzt werden kann (siehe Corollar 6.2.11 auf Seite 348) *)

WHILE $U \neq V$ **DO**

bestimme eine Kante $(u, v) \in E$, die U und $V \setminus U$ verbindet mit minimalen Kosten;
 füge v in U ein;
 füge (u, v) in E' ein;

OD

Gib E' zurück (* (V, E') ist ein MST *)

Beispiel 6.2.9. Abbildung 62 demonstriert die Arbeitsweise des Prim-Algorithmus für folgenden Graphen:

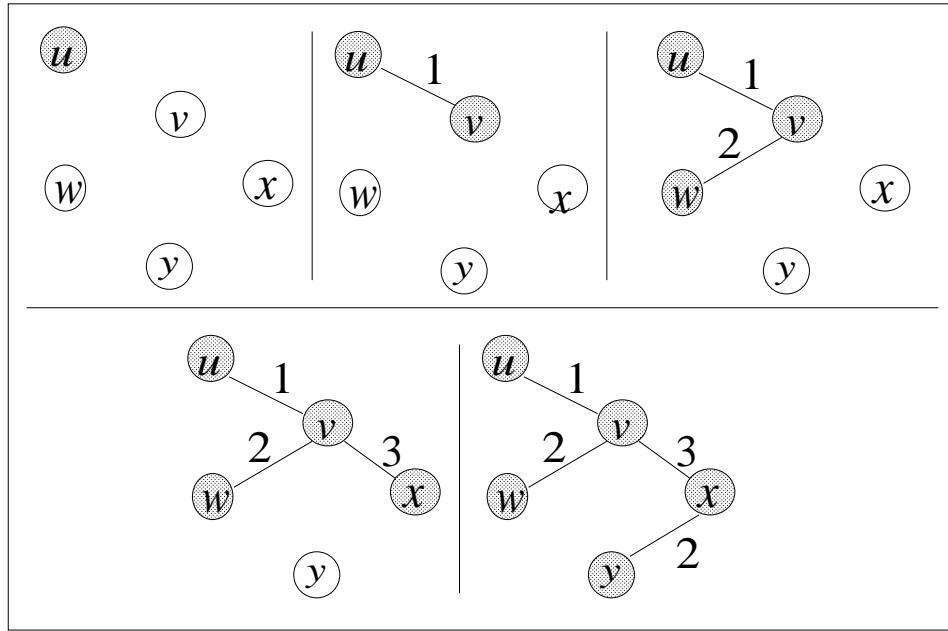
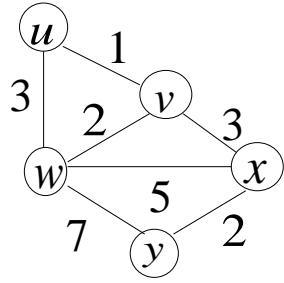


Abbildung 62: Beispiel zum Prim-Algorithmus

Die grau schattierten Knoten bilden jeweils die Menge U . □

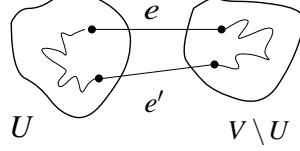
Lemma 6.2.10 (Zur Korrektheit des Prim-Algorithmus). Sei $G = (V, E)$ ein ungerichteter, zusammenhänger Graph und $U \subseteq V$ und $E' \subseteq E$ eine Kantenmenge, so dass (U, E') ein ungerichteter Baum ist. Weiter sei $e = (u, v) \in E \setminus E'$ eine Kante in G , so dass $u \in U$, $v \in V \setminus U$ und

$$\delta(e) = \min\{\delta(e') : e' \in E \text{ verbindet } U \text{ mit } V \setminus U\}.$$

Dann gibt es einen aufspannenden Baum $\mathcal{T} = (V, E_{\mathcal{T}})$ für G mit $E' \cup \{e\} \subseteq E_{\mathcal{T}}$, für den $\text{cost}(\mathcal{T})$ minimal ist unter allen aufspannenden Bäumen $\mathcal{T}' = (V, E_{\mathcal{T}'})$ für G mit $E' \subseteq E_{\mathcal{T}'}$.

Beweis. Sei $\mathcal{T}' = (V, E_{\mathcal{T}'})$ ein aufspannender Baum für G mit $e \notin E_{\mathcal{T}'}$ und $E' \subseteq E_{\mathcal{T}'}$. Wir zeigen, dass es einen aufspannenden Baum $\mathcal{T} = (V, E_{\mathcal{T}})$ für G mit $E' \cup \{e\} \subseteq E_{\mathcal{T}}$ und $\text{cost}(\mathcal{T}) \leq \text{cost}(\mathcal{T}')$ gibt.

Da $|E_{\mathcal{T}'}| = n - 1$, besteht die Kantenmenge $E_{\mathcal{T}'} \cup \{e\}$ aus n Kanten. Daher gibt es einen einfachen Zyklus $\pi = v_0, v_1, \dots, v_k$, der sich aus Kanten in $E_{\mathcal{T}'} \cup \{e\}$ zusammensetzt (Lemma 6.2.1 auf Seite 338). Da \mathcal{T}' zyklenfrei ist, kommt Kante e auf diesem Zyklus vor. Da e die Knotenmengen U und $V \setminus U$ verbindet, muss es eine weitere Kante e' auf diesem Zyklus geben, welche U mit $V \setminus U$ verbindet.



Dann gilt $e' \in E_{\mathcal{T}'}$ und nach Wahl von e :

$$\delta(e) \leq \delta(e').$$

Sei nun $E_{\mathcal{T}} = (E_{\mathcal{T}'} \setminus \{e'\}) \cup \{e\}$. Dann ist $\mathcal{T} = (V, E_{\mathcal{T}})$ ein Baum, welcher aus \mathcal{T}' entsteht, indem Kante e' durch e ersetzt wird. Die Baumeigenschaft von \mathcal{T} ergibt sich aus der Beobachtung, dass \mathcal{T} zusammenhängend ist (da e und e' auf einem Zyklus liegen) und $n - 1$ Kanten hat (Satz 6.2.5 auf Seite 340). Also ist \mathcal{T} ein aufspannender Baum für G . Ferner gilt:

$$cost(\mathcal{T}) = cost(\mathcal{T}') - \delta(e') + \underbrace{\delta(e)}_{\leq \delta(e')} \leq cost(\mathcal{T}')$$

Weiter gilt $E' \subseteq E_{\mathcal{T}'}$ und $e' \notin E'$ (da E' nur Knoten in U miteinander verbindet). Also ist $E' \cup \{e\} \subseteq E_{\mathcal{T}}$. \square

Corollar 6.2.11 (Schleifeninvariante des Prim-Algorithmus). Sind U , E' und e wie in Lemma 6.2.10 und gilt $E' \subseteq E_{\mathcal{T}}$ für die Kantenmenge eines minimalen aufspannenden Baums \mathcal{T} für G , so gibt es einen minimalen aufspannenden Baum, der die Kantenmenge $E' \cup \{e\}$ enthält.

Die Korrektheit des Prim-Algorithmus ergibt sich nun wie folgt. Durch Induktion nach i kann – mit Hilfe von Corollar 6.2.11 – gezeigt werden, dass zu Beginn des i -ten Durchlaufs der WHILE-Schleife die aktuelle Kantenmenge E' zu einer Kantenmenge $E_{\mathcal{T}}$ ergänzt werden kann, so dass $(V, E_{\mathcal{T}})$ ein minimaler aufspannender Baum ist. Sobald also $U = V$ ist, ist (U, E') ein minimaler aufspannender Baum. Man beachte, dass der Eingabograph als zusammenhängend vorausgesetzt wird. Daher ist die Abbruchbedingung $U = V$ tatsächlich nach n Iterationen erfüllt.

Laufzeit. Wir diskutieren verschiedene Implementierungsvarianten und deren Laufzeit. Prinzipiell stehen hier dieselben Möglichkeiten offen, die für den Dijkstra-Algorithmus vorgeschlagen wurden. Tatsächlich sind die beiden Algorithmen ähnlich, wenn man das Auffinden einer billigsten Kante, die U und $V \setminus U$ verbindet, mit Hilfe von Werten

$$d(v) = \min \{ \delta(u, v) : (u, v) \in E, u \in U, v \in V \setminus U \}$$

realisiert. Im Prim-Algorithmus ist dann in jeder Iteration ein Knoten $v \in V \setminus U$ mit minimalem $d(v)$ und eine Kante (u, v) mit $u \in U$ und $\delta(u, v) = d(v)$ zu wählen. Diese Überlegung führt zu Algorithmus 80, welcher für jeden Knoten $v \in V \setminus U$, der über eine Kante mit U verbunden ist, einen Knoten $father(v) = u \in U$ speichert, so dass $\delta(v, u) = d(v)$.

Algorithmus 80 Algorithmus von Prim zur MST-Konstruktion

wähle einen beliebigen Knoten $u_0 \in V$ und setze $U := \{u_0\}$;

FOR ALL Knoten $v \in Post(u_0)$ **DO**

$d(v) := \delta(u_0, v)$;

$father(v) := u_0$;

OD

$E' := \emptyset$;

WHILE $U \neq V$ **DO**

bestimme einen Knoten $v \in V \setminus U$, für den $d(v)$ minimal ist;

füge v in U ein;

füge die Kante $(father(v), v)$ in E' ein;

FOR ALL Knoten $w \in Post(v) \setminus U$ **DO**

IF $d(w)$ ist undefiniert oder $d(w) > \delta(v, w)$ **THEN**

$d(w) := \delta(v, w)$;

$father(w) := v$

FI

OD

OD

Gib E' zurück

(* (V, E') ist ein MST *)

Ähnlich wie für den Dijkstra-Algorithmus ergeben sich nun unterschiedliche Möglichkeiten zur Repräsentation der Randmenge

$$W = \{v \in V \setminus U : d(v) \text{ ist definiert}\}.$$

Keine explizite Darstellung von W . Die einfachste Variante verzichtet auf eine explizite Darstellung von W und initialisiert $d(w)$ mit ∞ , sofern w ein Knoten in $V \setminus \{u_0\}$ ist, welcher kein direkter Nachfolger von u_0 ist, also $w \in V \setminus (Post(u_0) \cup \{u_0\})$. Stellt man U durch einen Bitvektor dar, so kann ein Knoten $v \in V \setminus U$ mit minimalem $d(v)$ in Zeit $\Theta(n)$ gefunden werden. In diesem Fall erhält man quadratische Gesamtkosten

$$\Theta(n + n^2 + m) = \Theta(n^2).$$

Der Summand n steht für die Initialisierung, der Summand n^2 für die Gesamtzeit der Minimumssuche in den $n - 1$ Iterationen und der Summand $m = |E|$ für die Gesamtausführungszeit der inneren FOR-Schleife (dort werden alle Adjazenzlisten durchlaufen).

Darstellung von W durch eine Priority Queue. Weitere Implementierungsvarianten ergeben sich durch die Verwaltung der Knoten in der Randmenge W durch einen Minimumsheap mit Einträgen der Form $\langle w, d(w) \rangle$. Initial besteht der Heap aus den Einträgen $\langle w, \delta(w, u_0) \rangle$ für $w \in Post(u_0)$. Nimmt man den Restrukturierungsaufwand des Heaps in Kauf, welcher durch die innere FOR-Schleife verursacht wird, so kommt man für jeden Knoten mit höchstens einem Eintrag in der Priority Queue aus. Es ergeben sich höchstens logarithmische Kosten für jede in der inneren FOR-Schleife betrachtete Kante (v, w) und somit die Gesamtkosten

$$\Theta\left(\underbrace{n}_{\text{Init.}} + \underbrace{n \log n}_{(n-1)\text{-mal EXTRACT_MIN}} + \underbrace{m \log n}_{\substack{\text{Einfügen/Reorganisation} \\ \text{des Minimumsheaps}}}\right) = \Theta(m \log n)$$

im schlimmsten Fall. Beachte, dass $m \geq n - 1$, da G zusammenhängend ist.

Alternativ kann man pro Knoten mehrere Einträge zulassen. Die Anzahl der Iterationen der WHILE-Schleife kann sich dadurch auf $\Theta(m)$ erhöhen, wobei in jeder Iteration der WHILE-Schleife zu prüfen ist, ob der der Priority Queue entnommene Knoten bereits in U liegt. In diesem Fall ist der betreffende Knoten zu ignorieren. Die Anzahl an Elementen des Heaps kann auf $\Theta(m)$ anwachsen. Die Gesamtkosten betragen dann $\Theta(n + m \log m) = \Theta(m \log n)$, da $n - 1 \leq m < n^2$ und $\log(n^2) = 2 \log n$.

Satz 6.2.12 (Kosten des Prim-Algorithmus). Mit dem Algorithmus von Prim lässt sich ein minimaler aufspannender Baum je nach Implementierung in Zeit $\Theta(n^2)$ oder $\Theta(m \log n)$ bestimmen. Dabei ist n die Anzahl an Knoten in dem zugrundeliegenden Graphen und m die Kantenanzahl.

6.2.5 Der Kruskal-Algorithmus und UNION-FIND Wälder

Wir stellen nun einen weiteren Algorithmus zur MST-Bestimmung vor, der auf Kruskal zurückgeht. Dieser arbeitet ebenfalls mit einer Greedy-Strategie und betrachtet die Kanten in aufsteigender Sortierung hinsichtlich ihrer Kosten. Im Gegensatz zum Prim-Algorithmus ist die jeweilige Kantenmenge nicht zwangsläufig zusammenhängend, sondern kann aus mehreren Bäumen bestehen. Sie bildet jedoch stets einen azyklischen Graphen. Die Grundidee des Kruskal-Algorithmus für die MST-Konstruktion besteht darin, mit $E_T = \emptyset$ beginnend, sukzessive eine noch nicht betrachtete Kante $e \in E$ mit minimalen Kosten (unter allen noch nicht betrachteten Kanten) auszuwählen und zu prüfen, ob diese zur aktuellen Kantenmenge E_T hinzugefügt werden kann. Letzteres besteht in dem Test, ob die Hinzunahme von e zu E_T einen Zyklus auslöst. Wenn nein, so wird e in E_T eingefügt; andernfalls wird e verworfen. Das Abbruchkriterium greift auf Satz 6.2.5 (Seite 340) zurück: sobald die Kantenmenge E_T genau $|V| - 1$ Kanten enthält, ist (V, E_T) ein aufspannender Baum und der Algorithmus hält an. Die wesentlichen Schritte sind in Algorithmus 81 auf Seite 352 zusammengefasst.

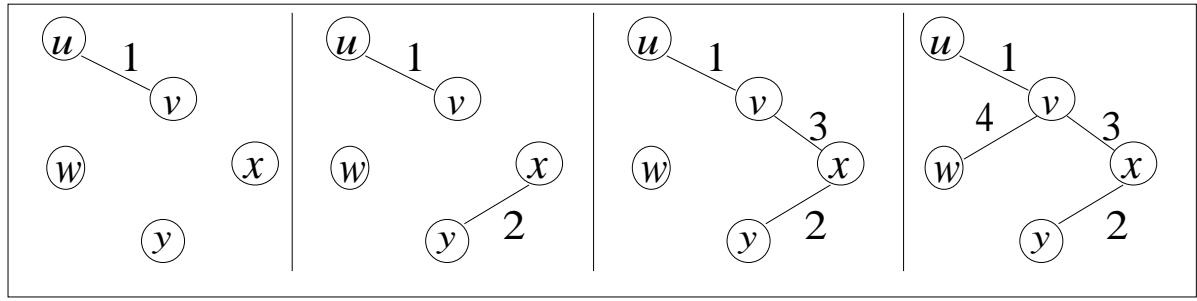
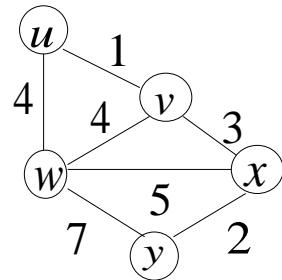


Abbildung 63: Beispiel zum Kruskal-Algorithmus

Beispiel 6.2.13. Abbildung 63 demonstriert die Arbeitsweise des Kruskal-Algorithmus für den folgenden Graphen:



Im Gegensatz zum Prim-Algorithmus wird also hier stets die nächst günstige, noch nicht betrachtete Kante inspiert und in E_T eingefügt, sofern diese keinen Zyklus in E_T auslöst. Es kann daher temporär – wie im zweiten Schritt in Abbildung 63 – ein Wald mit mehreren Teilbäumen entstehen. \square

Im Gegensatz zum Prim-Algorithmus umgeht der Kruskal-Algorithmus einen aufwendigen Auswahlschritt für die nächste Kante, nimmt dafür jedoch eine initiale Sortierphase in Kauf.

Korrektheit des Kruskal-Algorithmus. Für den Nachweis der Korrektheit des Kruskal-Algorithmus bedienen wir uns der Theorie zu Matroiden, die in Abschnitt 5.4.2 auf Seite 259 ff behandelt wurde. Zunächst weisen wir die Matroid-Eigenschaft für das Teilmengensystem der zyklenfreien Teilmengen von E nach. Mit einer zyklenfreien Kantenmenge ist eine Kantenmenge $E' \subseteq E$ gemeint, für welche der Graph (V, E') azyklisch ist.

Lemma 6.2.14 (Matroid der zyklenfreien Kantenmengen). Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph. Ferner sei \mathcal{U} die Menge aller zyklenfreien Kantenmengen $E' \subseteq E$. Dann ist (E, \mathcal{U}) ein Matroid.

Beweis. Offenbar ist $\emptyset \in \mathcal{U}$. Ist $E'' \subseteq E' \subseteq E$ und E' zyklenfrei, so ist auch E'' zyklenfrei. Also ist (E, \mathcal{U}) ein Teilmengensystem.

Algorithmus 81 Algorithmus von Kruskal zur MST-Konstruktion

bestimme eine Sortierung e_1, e_2, \dots, e_m der Kanten aufsteigend nach ihren Gewichten;

$E_{\mathcal{T}} := \emptyset$; (* $E_{\mathcal{T}}$ ist zyklenfreie Kantenmenge, die zu einem MST ergänzt wird *)
 $i := 1$;

WHILE $|E_{\mathcal{T}}| < |V| - 1$ **DO**

IF $E_{\mathcal{T}} \cup \{e_i\}$ ist zyklenfrei **THEN**

$E_{\mathcal{T}} := E_{\mathcal{T}} \cup \{e_i\}$

FI

$i := i + 1$

OD

Gib $E_{\mathcal{T}}$ aus.

(* $(V, E_{\mathcal{T}})$ ist ein MST *)

Nun zum Nachweis der Austauscheigenschaft. Seien E'', E' zyklenfreie Kantenmengen und $|E''| < |E'|$. Zu zeigen ist die Existenz einer Kante $e \in E' \setminus E''$, so dass $E'' \cup \{e\}$ zyklenfrei ist. Nehmen wir an, dass jede Kante $e \in E' \setminus E''$ einen Zyklus in E'' auslöst. Dann verbindet jede solche Kante $e \in E' \setminus E''$ zwei Knoten, welche derselben Zusammenhangskomponente in dem Graphen (V, E'') angehören. Wegen

$$E' = (E' \setminus E'') \cup (E' \cap E'')$$

liegen die Knoten v und w jeder Kante $(v, w) \in E'$ in derselben Zusammenhangskomponente unter der Kantenrelation E'' . Ist also C' eine Zusammenhangskomponente von (V, E') , so ist C' auch in (V, E'') zusammenhängend. Diese Überlegungen zeigen, dass jede Zusammenhangskomponente C'' von (V, E'') als disjunkte Vereinigung von Zusammenhangskomponenten C' von (V, E') dargestellt werden kann. Insbesondere gilt:

Anzahl an Zusammenhangskomponenten von (V, E'')

(*)

\leq Anzahl an Zusammenhangskomponenten von (V, E')

Andererseits ist für jeden azyklischen ungerichteten Graphen mit r Zusammenhangskomponenten und N Knoten die Kantenanzahl durch $N - r$ gegeben. Siehe Corollar 6.2.3 auf Seite 339. Aus (*) folgt daher, dass $|E'| \leq |E''|$. Widerspruch. \square

\mathcal{U} -Maximalität einer Kantenmenge $E' \subseteq E$ entspricht hier der Forderung, dass E' zyklenfrei ist und $|E'| = n - 1$. Die \mathcal{U} -maximalen Kantenmengen sind also genau diejenigen Kantenmengen, welche einen aufspannenden Baum bilden. Offenbar ist der Kruskal-Algorithmus eine Instanz des Greedy-Schemas für das zu (E, \mathcal{U}, δ) gehörende

Minimierungsproblem (Algorithmus 63 auf Seite 263). Die zugrundeliegende Gewichtsfunktion ist also $w = \delta$. Satz 5.4.14 auf Seite 262 induziert daher die Korrektheit des Kruskal-Algorithmus.

Exkurs: UNION-FIND-Datenstrukturen

Für eine effiziente Implementierung des Zyklentests „Ist $E_{\mathcal{T}} \cup \{e_i\}$ zyklenfrei?“ im Kruskal-Algorithmus kann eine so genannte UNION-FIND-Datenstruktur eingesetzt werden. UNION-FIND-Datenstrukturen dienen der Darstellung einer Partition einer festen endlichen Grundmenge V , d.h. einer Zerlegung von V in paarweise disjunkte, nichtleere Teilmengen $V = V_1 \cup V_2 \cup \dots \cup V_r$, wobei folgende beiden Operationen unterstützt werden:

- Suchen eines Elements $v \in V$: $FIND(v)$ liefert diejenige Menge V_i , die v enthält.
- Bilden der Vereinigung $V_i \cup V_j$ durch Anwenden der Operation $UNION(V_i, V_j)$.

Ist z.B. $G = (V, E)$ ein ungerichteter Graph wie im Kruskal-Algorithmus, wobei $V = \{v_1, \dots, v_n\}$, so können die oben erwähnten Operationen $FIND$ und $UNION$ zum Bestimmen der Zusammenhangskomponenten oder für einen Zyklentest eingesetzt werden.

Berechnung der Zusammenhangskomponenten mit UNION-FIND. Hierzu verwenden wir Partitionen $V = V_1 \cup \dots \cup V_r$, deren Komponenten V_i jeweils für zusammenhängende Knotenmengen stehen. Wir starten mit der „trivialen“ Partition $V = \{v_1\} \cup \dots \cup \{v_n\}$. Die folgenden Partitionen entstehen durch sukzessives Bearbeiten der Kanten: Je zwei Mengen V_i und V_j werden (mittels $UNION$) zusammengefaßt, sobald eine Kante, die V_i und V_j verbindet, entdeckt wird. Welche Mengen zu vereinigen sind, kann mittels der FIND-Operation herausgefunden werden. Die V_i 's stehen dann jeweils für zusammenhängende Knotenmengen. Sobald alle Kanten betrachtet wurden, stellen die V_i 's die Zusammenhangskomponenten dar. Siehe Algorithmus 82 auf Seite 353.

Algorithmus 82 UNION-FIND-Algorithmus zur Bestimmung der Zusammenhangskomponenten eines ungerichteten Graphen

Bilde einelementige Mengen $\{v\}$ für jeden Knoten v .

FOR ALL Knoten v **DO**

FOR ALL $w \in Post(v)$ **DO**

$v' := FIND(v);$

$w' := FIND(w);$

IF $v' \neq w'$ **THEN**

$UNION(v', w')$

FI

OD

OD

(* Die konstruierten Mengen bilden jeweils eine Zusammenhangskomponente. *)

Zyklentest mit UNION-FIND. Eine ähnliche Vorgehensweise kann für den Zyklentest in ungerichteten Graphen eingesetzt werden. Siehe Algorithmus 83. Hierbei ist es wichtig, dass jede Kante genau einmal betrachtet wird, da andernfalls das Hin- und Herlaufen auf einer Kante, also eine Knotenfolge der Form v, w, v , als Zyklus angesehen werden würde. Beispielsweise können die Kanten als Paare (v_i, v_j) mit $i < j$ aus einer Datei eingelesen werden, wobei eine beliebige Indizierung v_1, \dots, v_n der Knoten vorausgesetzt wird.

Algorithmus 83 UNION-FIND-Zyklentest in ungerichteten Graphen

Bilde einelementige Mengen $\{v\}$ für jeden Knoten v .

FOR ALL Kanten (v, w) **DO**

$v' := FIND(v);$

$w' := FIND(w);$

IF $v' = w'$ **THEN**

 return „Zyklus gefunden“

ELSE

$UNION(v', w')$

FI

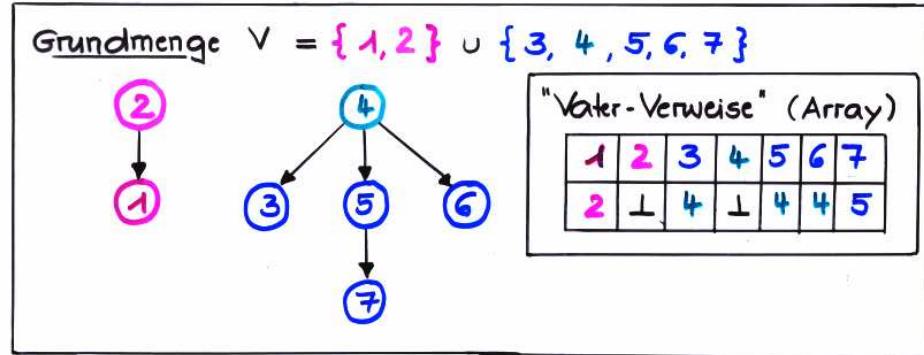
OD

return „Der Graph ist zyklenfrei.“

Implementierung von UNION-FIND-Wäldern. Eine der gängigen Methoden zur Realisierung der Operationen $FIND$ und $UNION$ ist die Darstellung der Partition $V = V_1 \cup \dots \cup V_r$ durch einen *gerichteten Wald*, in dem jedem Baum genau eine der Mengen V_i entspricht. Bei der $FIND$ -Operation wird zur Benennung der Mengen V_i auf die Namen der betreffenden Wurzelknoten (also Elemente von V) zurückgegriffen.

Für die Elemente der Grundmenge $V = \{v_1, \dots, v_n\}$ wird ein Array der Länge n verwendet, in das (neben eventueller Zusatzinformationen) für jeden Knoten v_i ein Verweis auf den Vater von v_i eingetragen wird. Der Verweis besteht üblicherweise lediglich in dem Arrayindex des Vaters. Für Wurzelknoten ist ein Spezialeintrag, etwa \perp , zu verwenden.

Bsp.: UNION-FIND Wald



z.B. $\text{FIND}(7) = \text{FIND}(5) = 4$
 $\text{FIND}(1) = 2$

(Array-Index $\hat{=}$ Element der Grundmenge)

Der Name einer Menge V_i der aktuellen Partition ergibt sich aus dem betreffenden Wurzelknoten.

1. Die Operation $\text{FIND}(v)$ kann mit Hilfe der Vaterverweise realisiert werden.
2. Aus Effizienzgründen wird zur Ausführung der Operation UNION zu jedem Wurzelknoten v die Anzahl an Knoten in dem zu v gehörenden Baum gespeichert. Die Operation $\text{UNION}(v, w)$ (mit Wurzelknoten v, w) macht den Baum mit der kleineren Knotenzahl zum Teilbaum des anderen. Haben beide Bäume dieselbe Knotenzahl, dann ist es irrelevant, welcher Baum zum Teilbaum des anderen gemacht wird.

In Algorithmus 84 auf Seite 356 sind die drei Grundoperationen (Initialisierung, FIND und UNION) angegeben. Zur Unterstützung der UNION-Operation verwenden wir einen Zähler $\text{mynumber}(v)$, welcher Aufschluss über die Anzahl an Elementen in dem Teilbaum von v gibt. (Tatsächlich wird $\text{mynumber}(v)$ nur für Wurzelknoten benötigt.) Ein Beispiel für die UNION-Operation ist in Abbildung 64 auf Seite 357 angegeben.

Die beschriebene Vorgehensweise zur Durchführung von UNION stellt sicher, daß jeder Baum höchstens logarithmische Höhe hat. Dies ist in folgendem Satz formuliert:

Satz 6.2.15 (Obere Schranke für die Höhe von UNION-FIND-Bäumen). Für jeden Baum \mathcal{T} eines UNION-FIND-Walds gilt:

$$\text{Höhe}(\mathcal{T}) \leq \log N,$$

wobei N die Anzahl der Knoten in \mathcal{T} ist.

Algorithmus 84 Operationen auf UNION-FIND-Wäldern

Initialisierung (* generiere Wald bestehend aus $|V|$ einelementigen Bäumen *)

FOR ALL $v \in V$ **DO**

$father(v) := \perp;$
 $mynumber(v) := 1;$

OD

FIND(v) (* laufe von v zur Wurzel und gib diese aus *)

$w := v;$
 WHILE $father(w) \neq \perp$ **DO**
 $w := father(w)$
 OD
 return w

UNION(v, w) (* hänge den kleineren Baum an den grösseren als Teilbaum an *)

IF $mynumber(v) \geq mynumber(w)$ **THEN**
 $father(w) := v;$
 $mynumber(v) := mynumber(v) + mynumber(w)$
ELSE
 $father(v) := w;$
 $mynumber(w) := mynumber(v) + mynumber(w)$
FI

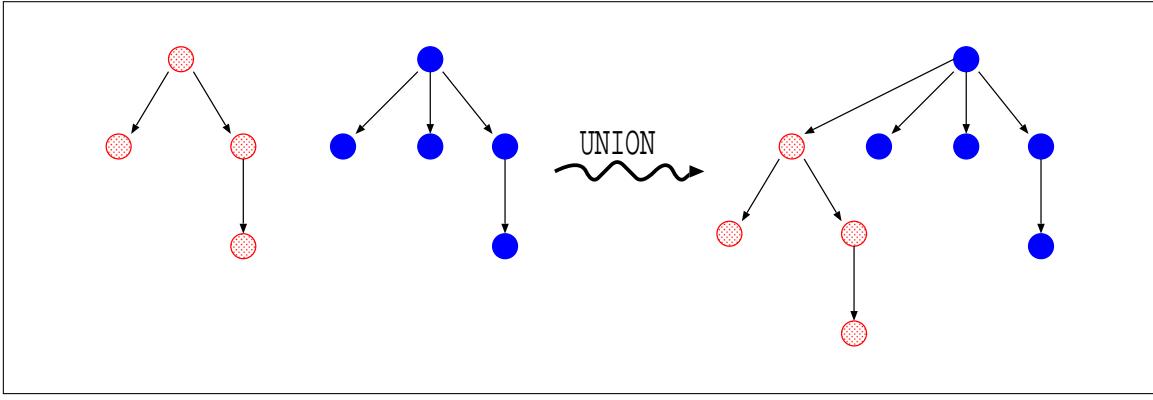
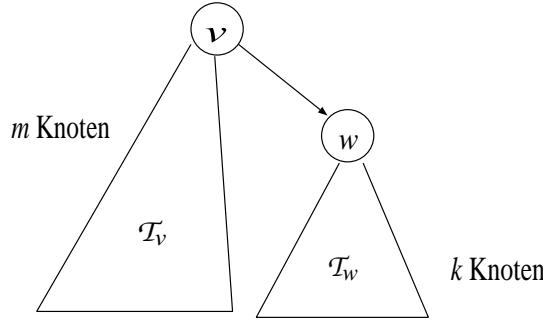


Abbildung 64: Beispiel zur UNION-Operation

Beweis. Wir weisen die Aussage durch Induktion über die Anzahl ℓ an durchgeführten UNION-Operationen nach. Für $\ell = 0$ ist $N = 1$ und es liegt ein Baum der Höhe 0 vor. Sei nun $\ell \geq 1$. Der vorliegende Baum \mathcal{T} ist durch eine UNION-Operation angewandt auf zwei Bäume $\mathcal{T}_v, \mathcal{T}_w$ mit den Wurzelknoten v bzw. w entstanden. Sei m die Knotenzahl in \mathcal{T}_v und k die Knotenzahl in \mathcal{T}_w . O.E. $m \geq k$. Dann hat \mathcal{T} die Form



und es gilt $N = m + k$, also $k \leq \frac{1}{2}N$, da $k \leq m$. Nach Induktionsvoraussetzung gilt:

$$\text{Höhe}(\mathcal{T}_v) \leq \log m \leq \log N$$

$$\text{Höhe}(\mathcal{T}_w) \leq \log k \leq \log(\frac{1}{2}N) = \log N - 1$$

Hieraus folgt:

$$\text{Höhe}(\mathcal{T}) = \max \left\{ \underbrace{\text{Höhe}(\mathcal{T}_v)}_{\leq \log N}, \underbrace{\text{Höhe}(\mathcal{T}_w) + 1}_{\leq \log N} \right\} \leq \log N$$

□

Die Kosten für die Durchführung einer FIND-Operation sind durch die Höhe des betreffenden Baums gegeben. Aufgrund von Satz 6.2.15 verursacht jede FIND-Operation höchstens die Kosten $\mathcal{O}(\log n)$, wenn n die Gesamtanzahl an betrachteten Elementen ist. Die UNION-Operationen verursachen jeweils lediglich konstante Kosten. Wir erhalten:

Corollar 6.2.16 (Kosten für Folgen von UNION-FIND-Operationen). Ist σ eine Folge von $\mathcal{O}(m)$ UNION-FIND-Operationen, dann kann σ in Zeit $\mathcal{O}(m \log m)$ ausgeführt werden.⁵⁹

Wir erwähnen hier ohne Beweis, dass eine zusätzliche Laufzeitverbesserung der UNION-FIND-Operationen durch die so genannte *Pfadkompression* möglich ist. Diese wird in Zusammenhang mit jeder FIND-Operation durchgeführt und macht alle Knoten w_r, w_{r-1}, \dots, w_1 , die auf dem durchlaufenen Rückwärtspfad liegen, zu Söhnen der Wurzel w_0 . Dabei ist $v = w_r$ der betreffende Knoten, für welchen die FIND-Operation aufgerufen wird, w_{r-1} der Vater von v , w_{r-2} der Vater von w_{r-1} , usw. Mit der Pfadkompression wird erreicht, dass eine gegebene Folge von $\mathcal{O}(m)$ UNION-FIND-Operationen in Zeit $\mathcal{O}(mG(m))$ durchgeführt werden kann, wobei $G(m)$ eine extrem langsam wachsende Funktion ist (fast konstant).

Einsatz der UNION-FIND-Datenstruktur für den Kruskal-Algorithmus

Wir setzen nun die UNION-FIND-Datenstruktur im Algorithmus von Kruskal ein, um den Test, ob durch die Hinzunahme der Kante e ein Zyklus ausgelöst wird, zu unterstützen. Hierzu starten wir mit den einelementigen Knotenmengen $\{v\}$, $v \in V$. Die Bäume des UNION-FIND-Walds entsprechen den Zusammenhangskomponenten des Graphen (V, E_τ) der jeweils aktuellen Kantenmenge E_τ . Wird die Kante $e = (v, w)$ betrachtet, dann liefern die Operationen $FIND(v)$ und $FIND(w)$ die Wurzeln v' bzw. w' derjenigen Zusammenhangskomponenten V_i, V_j mit $v \in V_i$ und $w \in V_j$. Die Kante e löst genau dann keinen Zyklus aus, wenn $V_i \neq V_j$, also wenn $v' \neq w'$. In diesem Fall wird e in E_τ eingefügt und die Mengen V_i und V_j bilden zusammen eine neue Zusammenhangskomponente. Dieser Beobachtung wird durch die Ausführung der Operation $UNION(v', w')$ Rechnung getragen. Siehe Algorithmus 85 auf Seite 359.

Die Laufzeit von Algorithmus 85 kann wie folgt abgeschätzt werden. Für die Sortierung benötigen wir $\Theta(m \log m) = \Theta(m \log n)$ Rechenschritte, wobei $m = |E|$ und $n = |V|$. Beachte, dass $\log m \leq \log n^2 = 2 \log n = \Theta(\log n)$. Die WHILE-Schleife wird höchstens m -mal durchlaufen. Pro Schleifendurchlauf werden zwei FIND-Operationen und höchstens eine UNION-Operation ausgeführt. Insgesamt werden also $\mathcal{O}(m)$ UNION-FIND-Operationen ausgeführt. Mit Corollar 6.2.16 auf Seite 358 ergeben sich die Kosten $\mathcal{O}(m \log n)$ für die WHILE-Schleife. Wir erhalten:

Satz 6.2.17 (Kosten des Kruskal Algorithmus). Mit der beschriebenen UNION-FIND-Implementierung des Kruskal-Algorithmus lässt sich ein MST in Zeit $\Theta(m \log m) = \Theta(m \log n)$ bestimmen. Dabei ist n die Anzahl an Knoten und m die Anzahl an Kanten in dem zugrundeliegenden Graphen.

⁵⁹Ist $m < n$, so fallen für den Initialisierungsschritt zusätzlich die Kosten $\Theta(n)$ an. Der Wert n steht dabei für die Kardinalität der Grundmenge V .

Algorithmus 85 Der Kruskal Algorithmus

(UNION-FIND-Formulierung)

$E_{\mathcal{T}} := \emptyset$; (* $E_{\mathcal{T}}$ ist zyklenfreie Kantenmenge, die zu einem MST ergänzt wird *)

(* Initialisiere den UNION-FIND-Wald *)

FOR ALL Knoten v **DO**

$\text{father}(v) := \perp$; (* jeder Knoten v ist Wurzel eines UNION-FIND-Walds *)

mynumber(v) := 1; (* Anzahl der Knoten im *)

(* UNION-FIND-Baum mit Wurzel v ist 1 *)

OD

Bestimme eine Sortierung e_1, \dots, e_m der Kanten von G aufsteigend nach den Gewichten.
 $i := 1;$

WHILE $|E_{\mathcal{T}}| < |V| - 1$ **DO**

Sei $e_i = (v, w)$.

$v' := FIND(v);$

$w' := FIND(w);$

IF $v' \neq w'$ THEN

(* v und w liegen in unterschiedlichen *)

(* Zusammenhangskomponenten von E_T *)

$$E_{\mathcal{T}} := E_{\mathcal{T}} \cup \{(v, w)\};$$

$UNION(v', w');$

FI

$i := i + 1;$

OD

Gib E_T zurück.

(* $(V, E_{\mathcal{T}})$ ist ein MST *)

6.3 DFS-Kantenklassifizierung und Zyklentest in Digraphen

Die Tiefen- und Breitensuche bilden die Basis vieler Graphalgorithmen. Wir betrachten hier nochmals die Tiefensuche in Digraphen, wobei wir mit einer Variante arbeiten, die Besuchsnummern für die Knoten vergibt und eine Klassifizierung der Kanten vornimmt. Diese Erweiterung der Tiefensuche ermöglicht eine Einsicht in die Struktur des Digraphen und kann zum Lösen vieler Graphprobleme verwendet werden. Wir gehen hier jedoch nur auf den Zyklentest ein.

Im Folgenden sei $G = (V, E)$ ein Digraph in Adjazenzlistendarstellung. Als Basis verwenden wir die rekursive Formulierung der Tiefensuche mit Hilfe des Algorithmus $DFS(v)$.

DFS-Besuchsnummern. Während der Tiefensuche kann jedem Knoten v die Besuchsnummer $dfsNr(v)$ zugeordnet werden (siehe Algorithmus 86 auf Seite 361). Ist also v_1, \dots, v_n eine Anordnung der Knoten von G , so daß $dfsNr(v_j) = j$, dann werden die Knoten von G genau in der Reihenfolge v_1, \dots, v_n besucht. Durch die Verwendung von Besuchsnummern wird die explizite Darstellung der Menge *Visited* aller bereits besuchten Knoten überflüssig. Diese ergibt sich aus der Menge aller Knoten, denen eine Besuchsnummer ≥ 1 zugewiesen wurde. Für alle noch nicht besuchten Knoten v stimmt $dfsNr(v)$ mit dem initialen Wert (den wir als 0 definieren) überein.

DFS-Wälder. In Abschnitt 3.2.5 (Seite 95 ff) haben wir bereits gesehen, daß die Tiefensuche einen Wald induziert, dessen Kanten genau diejenigen Kanten (v, w) sind, so daß der Knoten w „über die Kante (v, w) “ bei der Ausführung der DFS besucht wird. Solche Kanten werden auch *DFS-Baumkanten* genannt. DFS-Baumkanten sind also genau diejenigen Kanten $(v, w) \in E$, so dass Knoten w bei der Inspektion der Kante (v, w) während der Ausführung von $DFS(v)$ noch nicht besucht war. Der Aufruf von $DFS(w)$ findet also innerhalb $DFS(v)$ statt.

Als *DFS-Wurzeln* bezeichnen wir alle Knoten v , für die ein Aufruf $DFS(v)$ vom Hauptalgorithmus aus stattfindet. Die Ausführung von $DFS(v)$ für eine DFS-Wurzel v führt zu einem gerichteten Baum mit Wurzel v . Dieser wird auch DFS-Baum mit Wurzel v genannt.

Man beachte, daß die DFS-Besuchsnummern und die Struktur des DFS-Walds (also die DFS-Wurzeln und DFS-Baumkanten) von der konkreten Darstellung des Graphen abhängen. Bei einer Adjazenzlistenimplementierung ist die Reihenfolge, in der die Knoten in dem Array abgelegt, und der Aufbau der Adjazenzlisten ausschlaggebend.

DFS-Klassifizierung der Kanten. Durch die Ausführung der Tiefensuche können alle Kanten in G , die keine DFS-Baumkanten sind, wie folgt klassifiziert werden:

- **DFS-Rückwärtskanten:** Kanten $(v, w) \in E$, so daß w ein Vorgänger von v im DFS-Wald ist, also v von w im DFS-Wald erreichbar ist.
- **DFS-Vorwärtskanten:** Kanten $(v, w) \in E$, so daß w von v im DFS-Wald über einen Pfad der Länge ≥ 2 erreichbar ist.

Algorithmus 86 Tiefensuche (DFS) mit expliziten Besuchszählern und Knotenfärbung

```
FOR ALL Knoten  $v$  DO
     $dfsNr(v) := 0$                                      (*  $color(v) := blue$  *)
OD
```

```

FOR ALL Knoten  $v$  DO
     $lfdNr := 0;$                                      (* laufende Nummer *)
    IF  $dfsNr(v) = 0$  THEN
         $DFS(v)$                                      (*  $v$  wird DFS-Wurzel *)
    FI
OD

```

Rekursiver Algorithmus $DFS(v)$:

$dfsNr(v) := lfdNr;$ $(^* color(v) := green ^*)$
 $lfdNr := lfdNr + 1;$

```

FOR ALL  $w \in Post(v)$  DO
  IF  $dfsNr(w) = 0$  THEN
     $DFS(w)$                                 (*  $(v, w)$  wird Baumkante im DFS-Wald *)
  FI

```

- (* Falls $\text{color}(w) = \text{green}$, dann ist (v, w) Rückwärtskante. *)
- (* Falls $\text{color}(w) = \text{black}$ und $\text{dfsNr}(w) < \text{dfsNr}(v)$, dann ist (v, w) Seitwärtskante. *)
- (* Falls $\text{color}(w) = \text{black}$ und $\text{dfsNr}(w) > \text{dfsNr}(v)$, dann ist (v, w) Vorwärtskante. *)

OD

(* $\text{color}(v) := \text{black}$ *)

- DFS-Seitwärtskanten: alle anderen Kanten in G .

DFS-Seitwärtskanten sind also genau solche Kanten $(v, w) \in E$, so daß weder v von w noch w von v im DFS-Wald erreichbar sind. Dabei ist es möglich, daß v und w in demselben DFS-Baum liegen. v und w können aber auch in unterschiedlichen DFS-Bäumen liegen. Abbildung 65 auf Seite 362 zeigt ein Beispiel, wobei links der Digraph und in der Mitte ein möglicher DFS-Wald zu sehen ist. Im rechten Bild sind die Rückwärtskante mit R, die beiden Seitwärtskanten mit S und die Vorwärtskante mit V markiert.

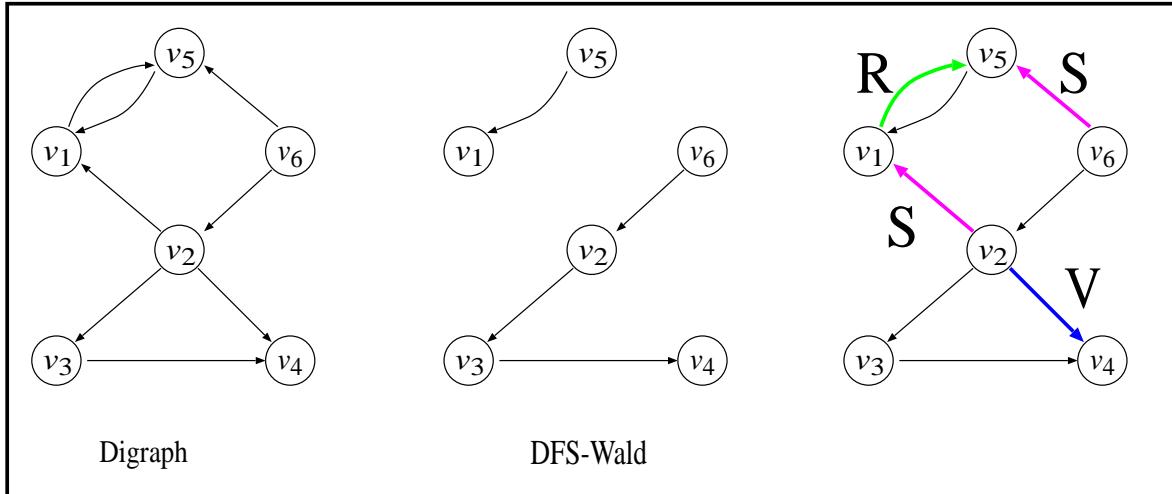


Abbildung 65: Beispiel zur DFS-Kantenklassifizierung

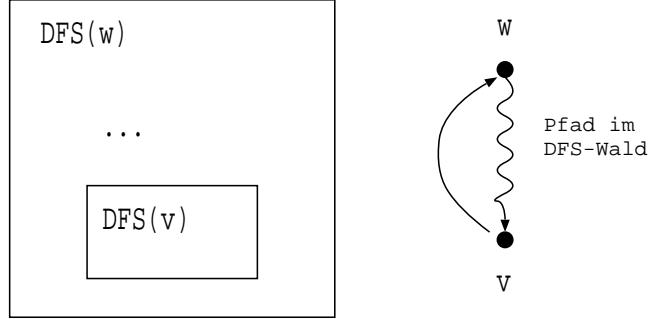
DFS-Färbung der Knoten. Wir verwenden eine dynamische Knotenfärbung mit den Farben *blau*, *grün* und *schwarz*, die sich während der Tiefensuche ändert.

- $\text{color}(v) = \text{blue}$ gdw v wurde noch nicht besucht.
- $\text{color}(v) = \text{green}$ gdw die Ausführung von $\text{DFS}(v)$ läuft
(ist aber noch nicht abgeschlossen).
- $\text{color}(v) = \text{black}$ gdw die Ausführung von $\text{DFS}(v)$ ist bereits abgeschlossen.

Die durch die DFS-Besuchszahlen gegebene Knotenreihenfolge stimmt also genau mit der Reihenfolge überein, in der die Knoten grün gefärbt werden. Die Schwarzfärbungsreihenfolge ist genau die Reihenfolge, in der die Aufrufe $\text{DFS}(v)$ abgeschlossen werden. Initial sind alle Knoten blau gefärbt, was der üblichen Initialisierung „ $\text{Visited} := \emptyset$ “ entspricht.

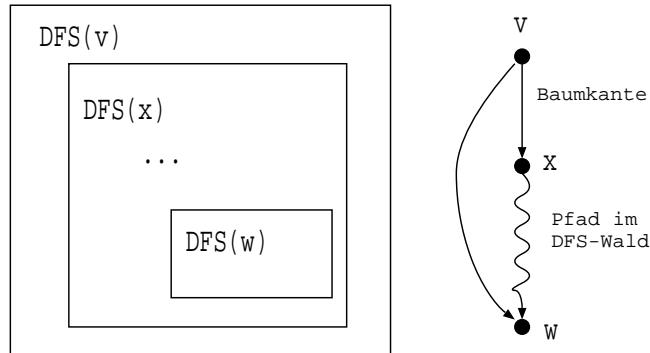
Die DFS-Klassifizierung der Kanten kann man an der Färbung der Knoten ablesen. Sei (v, w) eine Kante in G .

- (v, w) ist genau dann eine DFS-Rückwärtskante, wenn $DFS(v)$ während der Ausführung von $DFS(w)$ aufgerufen wird; also wenn die Färbung von Knoten w während der Ausführung von $DFS(v)$ grün ist.



(v, w) wird genau dann zur DFS-Vorwärts- oder DFS-Seitwärtskante, wenn $DFS(v)$ nicht während der Ausführung von $DFS(w)$ aufgerufen wird; also wenn die Färbung von Knoten w zu Beginn der Ausführung von $DFS(v)$ blau oder schwarz ist.

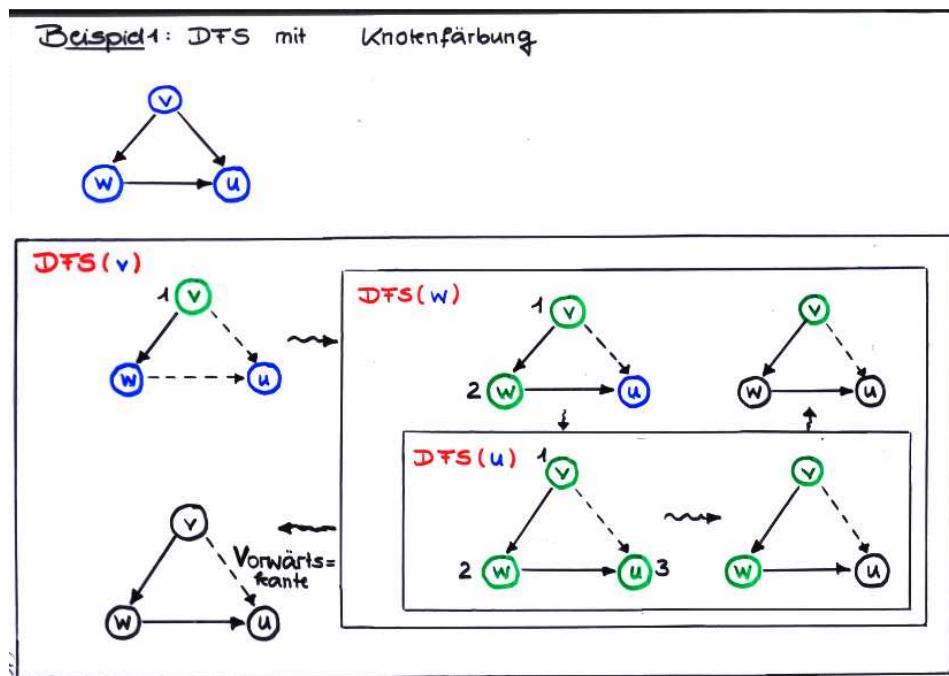
- Um eine *Vorwärtskante* handelt es sich genau dann, wenn w als Nachfolgers eines Sohns x von v in der DFS besucht wird, wobei $x \neq v$ und x vor w in der Adjazenzliste von v steht. D.h. der Aufruf von $DFS(w)$ findet innerhalb der Ausführung von $DFS(x)$ statt. In dem Moment, in dem die Kante (v, w) während der Ausführung von $DFS(v)$ inspiziert wird, ist die Färbung von Knoten w schwarz. Da v vor w besucht wird, ist die DFS-Besuchsnummer von w größer als die DFS-Besuchsnummer von v .



- Die Kante (v, w) wird genau dann zur *Seitwärtskante*, wenn die Ausführung von $DFS(w)$ zu Beginn der Ausführung von $DFS(v)$ bereits abgeschlossen ist; also wenn die DFS-Besuchsnummer von w kleiner als die DFS-Besuchsnummer von v ist und die Färbung von w bei der Inspektion der Kante (v, w) während der Ausführung von $DFS(v)$ schwarz ist.

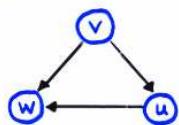
Beispiel 6.3.1 (Vorwärts- und Seitwärtskanten). Wir betrachten zunächst zwei Beispiele. Im ersten Beispiel (siehe Folie unten) wird (v, u) zur DFS-Vorwärtskante, wenn wir annehmen, dass w vor u in der Adjazenzliste von v steht und dass die Tiefensuche mit

Knoten v gestartet wird. Mit diesen Annahmen wird u über die Kante (w, u) bei der Tiefensuche besucht. $DFS(u)$ ist daher bereits abgeschlossen, wenn die Kante (v, u) innerhalb $DFS(v)$ betrachtet wird.

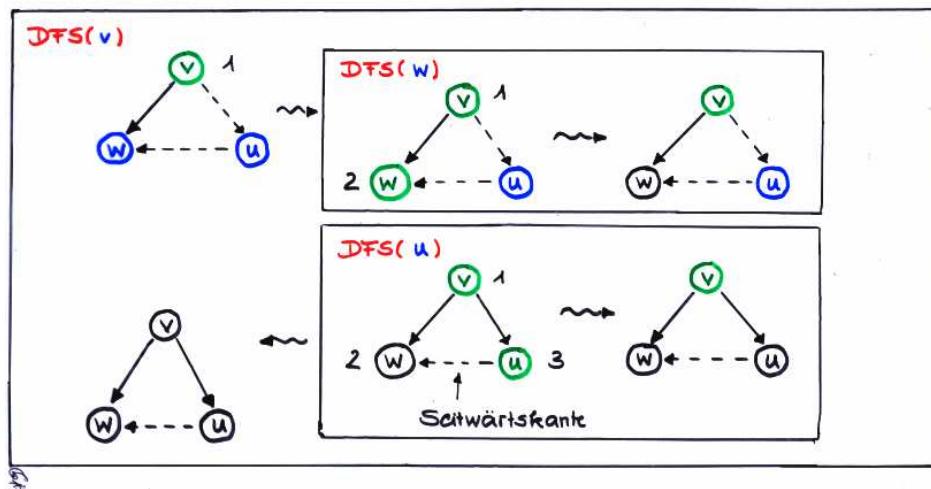
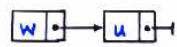


Im zweiten Beispiel (siehe Folie unten) wird die Kante (u, w) zur DFS-Seitwärtskante. Auch hier nehmen wir an, daß v zur DFS-Wurzel wird und daß w vor u in der Adjazenzliste von v steht. Dann ist $DFS(w)$ bei Aufruf $DFS(u)$ bereits abgeschlossen (d.h. w ist schwarz gefärbt).

Beispiel 2: DFS mit Knotenfärbung



Adjazenzliste von v :



□

Lemma 6.3.2 (Erreichbarkeit im DFS-Wald). Seien $G = (V, E)$ ein Digraph und $v, w \in V$, $v \neq w$. Folgende Aussagen sind äquivalent:

- (a) w ist von v im DFS-Wald erreichbar.
- (b) $dfsNr(v) < dfsNr(w)$ und w ist von v in G über einen Pfad erreichbar, der abgesehen von v aus Knoten besteht, die zu Beginn von $DFS(v)$ noch nicht besucht sind.
- (c) $dfsNr(v) < dfsNr(w)$ und $DFS(w)$ wird vor $DFS(v)$ beendet.

Ist v eine DFS-Wurzel, so sind folgende Aussagen äquivalent:

- (a) w ist von v im DFS-Wald erreichbar.
- (b') $dfsNr(v) < dfsNr(w)$ und $w \in Post^*(v)$.

Beweis. (a) \implies (b): Ist v, u_1, \dots, u_k, w ein Pfad im DFS-Wald, so ist v, u_1, \dots, u_k, w zugleich ein Pfad in G und es gilt

$$dfsNr(v) < dfsNr(u_1) < \dots < dfsNr(u_k) < dfsNr(w),$$

was die Aussage in (b) belegt.

(b) \implies (c) folgt aus der Tatsache, dass unter Voraussetzung (b) $DFS(w)$ innerhalb $DFS(v)$ aufgerufen wird, und somit $DFS(w)$ nach $DFS(v)$ aufgerufen, aber vor $DFS(v)$ beendet wird.

(c) \implies (a) ergibt sich aus der Beobachtung, dass in $DFS(v)$ nur von v im DFS-Wald erreichbare Knoten besucht werden und dass Bedingung (c) dazu äquivalent ist, dass der Aufruf von $DFS(w)$ innerhalb $DFS(v)$ stattfindet. \square

Folgender Satz liefert die Basis für einen DFS-basierten Zyklustest in gerichteten Graphen:

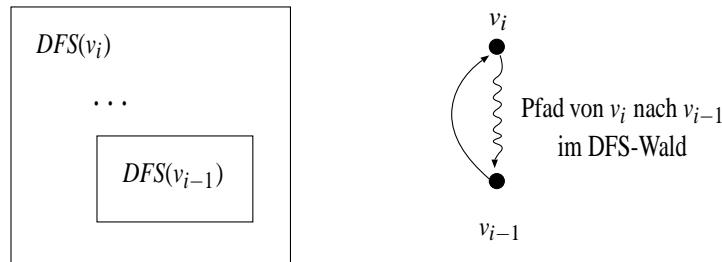
Satz 6.3.3 (Zyklen und Rückwärtskanten). Sei G ein Digraph. Dann gilt:

G ist genau dann azyklisch, wenn es keine DFS-Rückwärtskanten gibt.

Beweis. Es ist klar, daß jede DFS-Rückwärtskante einen Zyklus in G „schließt“. Liegt andererseits ein Zyklus v_0, v_1, \dots, v_r in G vor und ist v_i ($i \geq 1$) derjenige Knoten, der bei der Ausführung der DFS zuerst besucht wird, dann gilt offenbar

$$dfsNr(v_{i-1}) > dfsNr(v_i).$$

Da v_{i-1} über den Pfad $v_i, v_{i+1}, \dots, v_r, v_1, \dots, v_{i-1}$ von v_i in G erreichbar ist, wird v_{i-1} während der Ausführung von $DFS(v_i)$ besucht.



v_{i-1} ist also von v_i im DFS-Wald erreichbar. Also ist (v_{i-1}, v_i) eine DFS-Rückwärtskante.

□

Bemerkung 6.3.4. Die Aussage von Satz 6.3.3 ist so zu lesen, dass es im Falle eines zyklischen Digraphen G hinsichtlich *jedes* DFS-Walds für G wenigstens eine Rückwärtskante gibt. Umgekehrt genügt die Existenz einer Rückwärtskante hinsichtlich *eines* DFS-Walds, um auf die Existenz eines Zyklus in G schliessen zu können. □

Die Kantenklassifizierung kann – basierend auf Satz 6.3.3 – benutzt werden, um G auf Zyklenfreiheit zu testen. Hierzu ist lediglich eine Umformulierung von der in Algorithmus 86 gewählten Formulierung der Tiefensuche nötig, welche die Existenz von Rückwärtskanten prüft. Das Verfahren kann leicht erweitert werden, um im Falle eines zyklischen Graphen einen Zyklus als Antwort auszugeben. Nämlich wie?

Der so konzipierte DFS-basierte Zyklentest hat offenbar dieselbe Laufzeit wie die Tiefensuche. Wir erhalten:

Satz 6.3.5 (Kosten der DFS-basierten Zyklensuche in Digraphen). Mit der Tiefensuche lässt sich in Zeit $\Theta(n + m)$ prüfen, ob G zyklenfrei ist. Dabei ist $n = |V|$ die Anzahl an Knoten und $m = |E|$ die Anzahl an Kanten.

Zur Erinnerung: In ungerichteten Graphen kann der Zyklentest sogar in Zeit $\Theta(n)$ durchgeführt werden (siehe Satz 6.2.6 auf Seite 344). Diese Aussage gilt jedoch nicht in Digraphen, da es zyklenfreie gerichtete Graphen mit $\Theta(n^2)$ Kanten gibt. (Beispiel?)

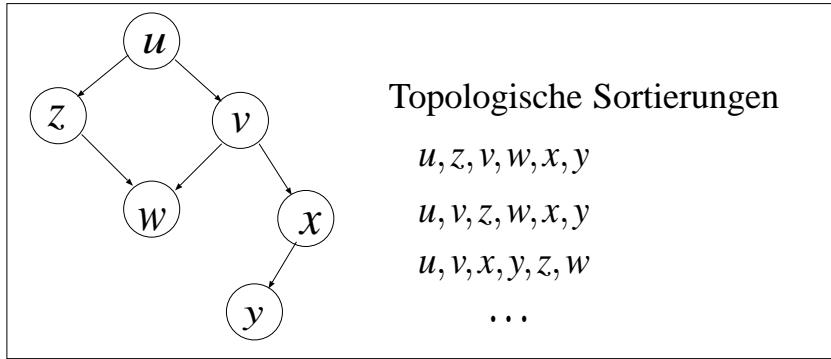


Abbildung 66: Beispiel für topologische Sortierungen

6.4 Topologisches Sortieren

Eine topologische Sortierung für einen Digraphen ist eine Aufzählung der Knoten, in der jeder Knoten v vor all seinen (direkten) Nachfolgern vorkommt. Topologische Sortierungen spielen in zahlreichen Anwendungen eine große Rolle, in denen die Kanten für einen gewissen „Informationsfluss“ stehen. Salopp formuliert: Die „vollständige Information“ an Knoten v liegt erst dann vor, wenn die Information für alle echten Vorgänger von v bekannt ist.

Definition 6.4.1 (Topologische Sortierung). Sei $G = (V, E)$ ein Digraph mit $|V| = n$. Eine topologische Sortierung für G ist eine Knotenpermutation v_1, \dots, v_n (also Knotenfolge, in der jeder Knoten genau einmal vorkommt), so dass gilt:

$$\text{Ist } (v_i, v_j) \in E, \text{ so ist } i < j.$$

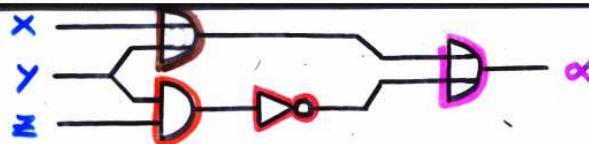
Die induzierte Funktion $ord : V \rightarrow \{1, 2, \dots, n\}$, $ord(v_i) = i$, welche jedem Knoten seine Ordnungsnummer zuweist, wird auch topologische Ordnungsfunktion für G genannt.⁶⁰ \square

Ein Digraph kann mehrere topologische Sortierungen haben wie das Beispiel in Abbildung 66 zeigt.

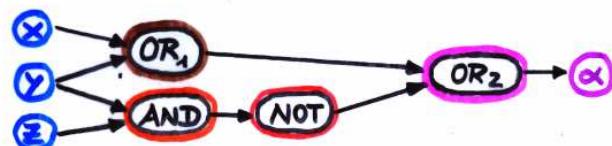
Liegt beispielsweise der Abhängigkeitsgraph von auszuführenden Jobs vor, dessen Knoten die Jobs sind und der genau dann eine Kante von J nach J' enthält, wenn Auftrag J' erst nach Fertigstellung von J gestartet werden kann, so stimmen die topologischen Sortierungen des Abhängigkeitsgraphen mit den zulässigen Abarbeitungsfolgen der Jobs überein. Als weiteres Anwendungsbeispiel erwähnen wir die Berechnung der durch ein Schaltbild dargestellten Schaltfunktion (kombinatorischer Schaltkreis). Wir fassen das Schaltbild als gerichteten azyklischen Graphen auf und bestimmen eine topologische Sortierung der Gatter, etwa g_1, \dots, g_r . Liegen konkrete Werte für die Eingabeveriablen vor, so können die Gatter in der Reihenfolge g_1, \dots, g_r ausgewertet werden.

⁶⁰ $ord : V \rightarrow \{1, 2, \dots, n\}$ ist also eine bijektive Abbildung, so daß $ord(v) < ord(v')$ für alle Kanten $(v, v') \in E$.

Bsp.:



Schaltbild als gerichteter Graph (DAG)



Topologische Sortierung:

$x, y, z, OR_1, AND, NOT, OR_2, \alpha$

Berechnung der Schaltfunktionen gemäß top. Sortierung:

$$f_{OR_1} = x \vee y, \quad f_{AND} = y \wedge z, \quad f_{NOT} = \neg f_{AND} = \dots$$

...

Satz 6.4.2 (Topologische Sortierungen und Zyklenfreiheit). Sei G ein Digraph. Es gibt genau dann eine topologische Sortierung für G , wenn G azyklisch ist.

Beweis. Die Aussage „genau dann“ ist klar. Ist nämlich ord eine topologische Ordnungsfunktion für G und ist $\pi = w_0, w_1, \dots, w_r$ ein Pfad in G , dann ist

$$ord(w_0) < ord(w_1) < \dots < ord(w_r).$$

Damit ist $w_0 = w_r$ und $r \geq 1$ unmöglich.

Der Beweis des Teils „dann, wenn“ benutzt folgende Hilfsaussage:

Hilfsaussage: In jedem nichtleeren DAG gibt es einen Knoten v mit $\text{Pre}(v) = \emptyset$.

Zum Beweis der Hilfsaussage nehmen wir an, daß $\text{Pre}(v) \neq \emptyset$ für alle Knoten v eines nichtleeren DAGs G . Zu jedem Knoten v wählen wir einen beliebigen Knoten $\text{pre}(v) \in \text{Pre}(v)$. Sei v_0 ein beliebiger Knoten in G . Wir definieren induktiv

$$v_{i+1} = \text{pre}(v_i), \quad i = 0, 1, 2, \dots$$

Sei n die Kardinalität der Knotenmenge V von G (also $n = |V|$). Dann ist

$$\pi = v_n, v_{n-1}, \dots, v_1, v_0$$

ein Pfad in G . Es gibt Indizes $0 \leq j < i \leq n$, so daß $v_i = v_j$. Dann ist

$$v_i, v_{i-1}, \dots, v_{j+1}, v_j$$

ein Zyklus in G . Dies widerspricht der Annahme, daß G azyklisch ist. Damit ist die Hilfsaussage bewiesen.

Die „dann, wenn“ Aussage von Satz 6.4.2 beweisen wir nun mit vollständiger Induktion nach $n = |V|$ unter Verwendung der Hilfsaussage.

- Im Induktionsanfang $n = 0$ ist nichts zu zeigen, da die leere Knotenfolge eine topologische Sortierung der Knoten des leeren Digraphen (also einem Digraphen ohne Knoten) ist.⁶¹
- Im Induktionsschritt $n \implies n + 1$ nehmen wir an, daß G ein DAG mit $n + 1$ Knoten ist. Wir wählen einen Knoten v , so daß $\text{Pre}(v) = \emptyset$. Sei G' derjenige Graph, der aus G entsteht, wenn v (und die aus v hinausführenden Kanten) entfernt wird. Diesem Knoten v kann die Ordnungsnummer $\text{ord}(v) = 1$ zugewiesen werden. G' ist offenbar ein DAG mit n Knoten. Nach Induktionsvoraussetzung gibt es eine topologische Ordnungsfunktion ord' für G' . Wir setzen

$$\text{ord}(w) = \text{ord}'(w) + 1$$

für jeden Knoten w in G' und erhalten somit eine topologische Ordnungsfunktion für G .

□

Der Induktionsbeweis von Satz 6.4.2 enthält zugleich ein rekursives Verfahren zum Bestimmen einer topologischen Sortierung für einen DAG G . Siehe Algorithmus 87 auf Seite 371. Wir wählen sukzessive einen Knoten v , der keine echten Vorgänger hat. Dieser erhält die Ordnungsnummer 1. Entfernt man v aus G , dann entsteht wiederum ein DAG, auf den sich dieselbe Vorgehensweise anwenden läßt. Somit erhält man einen Knoten

⁶¹Wem der leere Graph nicht geheuer ist, kann den Induktionsanfang auch mit $n = 1$ führen. Auch hier ist die Aussage klar, da ein DAG mit einem Knoten keine Kanten hat.

mit der Ordnungsnummer 2. So fortlaufend erhält man eine topologische Sortierung für G . Dieses Verfahren kann zugleich zu einem Zyklentest erweitert werden. Sobald ein Teilgraph entsteht, in dem es keinen Knoten v ohne einführende Kanten gibt, liegt ein zyklischer Graph vor (siehe Hilfsaussage im Beweis von Satz 6.4.2).

Algorithmus 87 Erstellung einer topologischen Sortierung mit eingebautem Zyklentest

$lfdNr := 0;$

WHILE es gibt einen Knoten v mit $Pre(v) = \emptyset$ **DO**

wähle einen solchen Knoten v ;

$lfdNr := lfdNr + 1;$

$ord(v) := lfdNr;$

entferne v aus G (samt der aus v hinausführenden Kanten);

OD

IF $V \neq \emptyset$ **THEN**

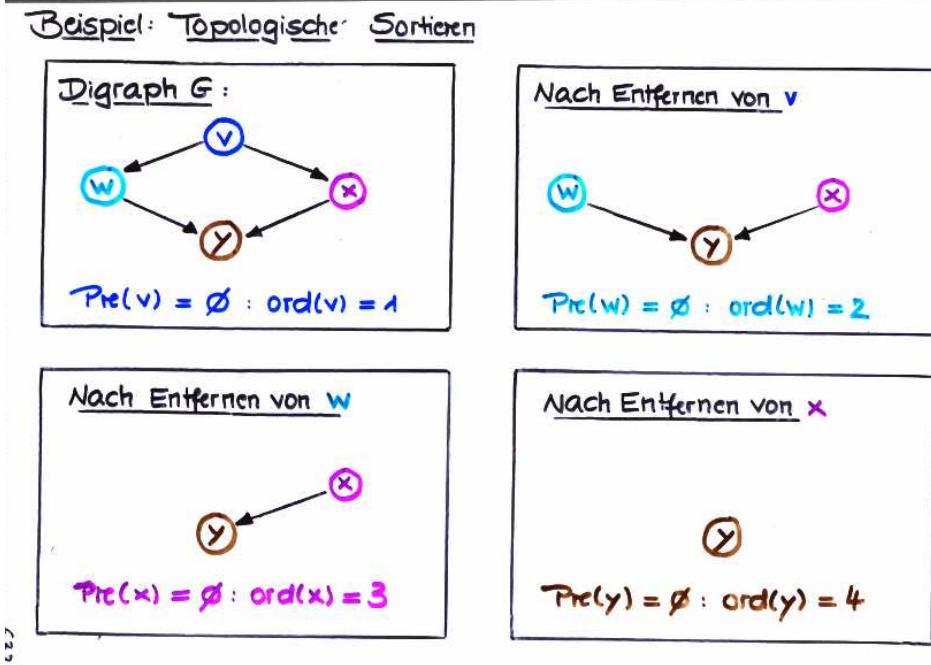
gib „ G ist zyklisch“ aus

ELSE

gib die berechnete Ordnungsfunktion aus

FI

Beispiel 6.4.3. Folgende Folie zeigt ein Beispiel:



Wir erhalten also die topologische Sortierung v, w, x, y . □

Das soeben skizzierten Verfahren benutzt die direkten Vorgängermengen $\text{Pre}(\cdot)$. Diese können zwar recht leicht aus der Adjazenzlisten-Darstellung generiert werden, jedoch gibt es eine einfachere Implementierungsmöglichkeit, welche die Mengen $\text{Pre}(v)$ nicht explizit repräsentiert. Anstelle dessen wird für jeden Knoten v ein Zähler $\text{indegree}(v)$ verwendet, welcher die Anzahl der in v einführenden Kanten zählt. Das „Entfernen“ eines Knotens v entspricht dann der Dekrementierung der Zähler $\text{indegree}(w)$ für alle Knoten $w \in \text{Post}(v)$. Die Menge aller Knoten mit Eingangsgrad 0 werden in einer Menge V_0 verwaltet. Diese kann z.B. als Warteschlange oder als lineare Liste implementiert werden. Diese Vorgehensweise ist in Algorithmus 88 auf Seite 373 zusammengefasst.

Die worst-case Laufzeit des Verfahrens ist offenbar $\Theta(|V| + |E|)$, da jeder Knoten v höchstens einmal in V_0 eingefügt und aus V_0 entfernt wird. Damit ist sichergestellt, daß jede Kante (v, w) in der Schleife „FOR ALL $w \in \text{Post}(v)$ DO ... OD“ höchstens einmal inspiziert wird. Die Berechnung der Werte $|\text{Pre}(v)|$ kann durch einen Durchlauf durch alle Adjazenzlisten vorgenommen werden, wobei für jeden Knoten v die Anzahl an Adjazenzlisten mitgezählt wird, in denen v enthalten ist. Daher kann auch der Initialisierungsschritt in Zeit $\Theta(|V| + |E|)$ realisiert werden. Wir erhalten folgenden Satz:

Satz 6.4.4 (Kosten für die Erstellung einer topologischen Sortierung). Sei $G = (V, E)$ ein Digraph. Der Test auf Zyklenfreiheit sowie gegebenenfalls die Bestimmung einer topologischen Sortierung sind in Zeit $\Theta(n + m)$ möglich, wobei $n = |V|$ und $m = |E|$.

Algorithmus 88 Erstellung einer topologischen Sortierung mit eingebautem Zyklentest

FOR ALL Knoten v **DO**

$\text{indegree}(v) := |\text{Pre}(v)|$

OD

$V_0 := \{v \in V : \text{indegree}(v) = 0\};$

(* V_0 verwaltet die Knoten, *)

(* die keinen Vorgänger haben *)

$lfdNr := 0;$

WHILE $V_0 \neq \emptyset$ **DO**

wähle einen Knoten $v \in V_0$;

(* v ist Knoten ohne Vorgänger *)

entferne v aus V_0 ;

$lfdNr := lfdNr + 1;$

$ord(v) := lfdNr;$

(* entferne v aus G *)

FOR ALL $w \in \text{Post}(v)$ **DO**

$\text{indegree}(w) := \text{indegree}(w) - 1;$

IF $\text{indegree}(w) = 0$ **THEN**

füge w in V_0 ein (* w hat nach Entfernen von v keinen Vorgänger *)

FI

OD

OD

IF $lfdNr \neq |V|$ **THEN**

gib „ G ist zyklisch“ aus

(* es gibt noch nicht entfernte Knoten und diese *)

(* haben jeweils mindestens einen Vorgänger *)

ELSE

gib die berechnete Ordnungsfunktion aus

FI

Kürzeste-Wege für azyklische Digraphen. Die Frage nach kürzesten Wegen wurde in Abschnitt 6.1 für beliebige Digraphen diskutiert. Für azyklische Graphen gibt es eine sehr einfache Methode für das single-source Problem, welche die Relaxationsschritte in der Reihenfolge einer topologischen Sortierung ausführt.

Im Folgenden sei $G = (V, E)$ ein zyklenfreier Digraph und $\delta : E \rightarrow \mathbb{R}$ eine Gewichtsfunktion, welche positive und negative Werte annehmen kann. Ferner sei v der Startknoten, von dem kürzeste Pfade zu allen anderen Knoten gesucht sind. Wie in den Algorithmen in Abschnitt 6.1 arbeiten wir mit oberen Schranken

$$d(w) \geq \Delta(w) = \min\{\delta(\pi) : \pi \text{ ist ein Pfad von } v \text{ nach } w\} \in \mathbb{R} \cup \{\infty\},$$

die durch sukzessive Relaxationsschritte „ $d(w) := \min\{d(w), d(u) + \delta(u, w)\}$ “ verbessert werden bis $d(w) = \Delta(w)$ für alle Knoten w . Siehe Algorithmus 89 auf Seite 374. Die Laufzeit ist offenbar $\Theta(n + m)$, wenn eine Adjazenzlistendarstellung des Graphen vorausgesetzt wird. Dabei ist $n = |V|$ und $m = |E|$.

Algorithmus 89 Kürzeste-Wege Algorithmus für azyklische Digraphen

Bestimme eine topologische Sortierung v_1, \dots, v_n für G .

(* Initialisierung wie im Dijkstra oder Bellman-Ford Algorithmus *)

```

FOR ALL Knoten  $w \in V$  DO
  IF  $w \neq v$  THEN
     $d(w) := +\infty$ 
  ELSE
     $d(v) := 0$ 
  FI
OD
```

Sei $v = v_i$.

(* $\Delta(v_j) = d(v_j) = \infty$ für $1 \leq j < i$ *)

```

FOR  $j = i, \dots, n$  DO
  FOR ALL  $w \in Post(v_j)$  DO
     $d(w) := \min\{d(w), d(v_j) + \delta(v_j, w)\}$  (* Relaxation *)
  OD
OD
```

(* $d(w) = \Delta(w)$ für alle Knoten w *)

Lemma 6.4.5 (Korrektheit von Algorithmus 89). Ist G ein azyklischer Digraph mit Gewichtsfunktion, so berechnet Algorithmus 89 die Kosten kürzester Pfade von v zu allen anderen Knoten, d.h. $d(w) = \Delta(w)$ zum Zeitpunkt der Terminierung.

Beweis. Offenbar ist keiner der Knoten v_j mit $j < i$, also Knoten, welche in der topologischen Sortierung vor Knoten $v = v_i$ stehen ist, von v erreichbar. Daher ist

$\Delta(v_j) = \infty = d(v_j)$, falls $j < i$. Ferner ist klar, dass $d(w) < \infty$ nur dann möglich ist, wenn es einen (kürzesten) Pfad von v nach w gibt, da die Werte $d(w)$ nur durch Anwenden der Relaxationsschritte geändert werden. Also ist die Aussage auch für solche Knoten $w = v_j$ mit $j > i$ und $\Delta(w) = \infty$ richtig.

Für den Startknoten $v = v_i$, wird offenbar der korrekte Wert $\Delta(v) = 0 = d(v)$ berechnet, da sich $d(v)$ in keinem der Relaxationsschritte verändern kann. Beachte, $v = v_i \notin Post(v_j)$ für alle $j \geq i$.

Sei nun $j > i$. Falls v_j von v_i erreichbar ist, so betrachten wir einen kürzesten Pfad w_0, w_1, \dots, w_r , der von $v = w_0$ nach $v_j = w_r$ führt und welcher minimale Länge (gemessen an der Kantenanzahl) unter allen kürzesten Pfaden von v nach w hat. Die Knoten w_0, w_1, \dots, w_r werden in dieser Reihenfolge in der FOR-Schleife betrachtet. Zwischen w_k und w_{k+1} werden möglicherweise noch andere Knoten betrachtet, das ist jedoch unerheblich.

Durch Induktion nach $k \in \{0, 1, \dots, r - 1\}$ kann nun gezeigt werden, dass $d(w_{k+1}) = \Delta(w_{k+1})$ spätestens bei der Inspektion des Knotens w_k (und dessen Adjazenzliste) in der FOR-Schleife gesetzt wird und dass dieser Wert in den darauf folgenden Iterationen nicht mehr verändert wird. \square

Bemerkung 6.4.6. Die Aussage von Lemma 6.4.5 kann durch folgende Schleifeninvariante für die zweite FOR-Schleife verschärft werden:

- $\Delta(v_k) = d(v_k)$ für alle Indizes k mit $i \leq k \leq j$,
- $\Delta(v_k) \leq d(v_k)$ für alle Indizes k mit $n \geq k > j$.

Diese Aussage kann durch Induktion nach j nachgewiesen werden. \square

6.5 Starke Zusammenhangskomponenten

In Abschnitt 6.2 haben wir über die Zusammenhangskomponenten ungerichteter Graphen gesprochen. Wir betrachten nun das gerichtete Gegenstück, sogenannte starke Zusammenhangskomponenten.⁶²

Definition 6.5.1 (Starke Zusammenhangskomponente, SCC). Sei $G = (V, E)$ ein Digraph und $C \subseteq V$. C heißt starke Zusammenhangskomponente, falls folgende Eigenschaften gelten:

- (i) Je zwei Knoten in C sind voneinander erreichbar, also $v_2 \in Post^*(v_1)$ für alle $v_1, v_2 \in C$.
- (ii) C ist eine maximale Knotenmenge, welche Eigenschaft (i) besitzt, d.h. ist $C \subseteq C' \subseteq V$, so dass $v_2 \in Post^*(v_1)$ für alle Knoten $v_1, v_2 \in C'$, so gilt $C = C'$.

Im Folgenden verwenden wir häufig die Abkürzung SCC, welche für die englische Bezeichnung „strongly connected component“ steht. G heißt stark zusammenhängend, falls die Knotenmenge V eine starke Zusammenhangskomponente ist, also falls je zwei Knoten in G voneinander erreichbar sind. \square

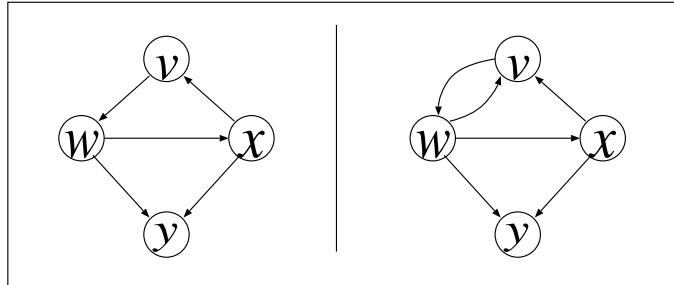


Abbildung 67: $\{v, w, x\}$ und $\{y\}$ sind die SCCs beider Digraphen

In beiden Digraphen aus Abbildung 67 erfüllt die Knotenmenge $\{v, w\}$ zwar die Eigenschaft (i), jedoch ist $\{v, w\}$ keine starke Zusammenhangskomponente, da die Maximalität (Eigenschaft (ii)) nicht erfüllt ist. In beiden Fällen sind $\{v, w, x\}$ und $\{y\}$ die starken Zusammenhangskomponenten.

Lemma 6.5.2. Sei G ein Digraph und v_1, v_2 zwei Knoten in G . Dann gilt:

v_1, v_2 liegen genau dann in derselben starken Zusammenhangskomponente,
wenn $Post^*(v_1) = Post^*(v_2)$.

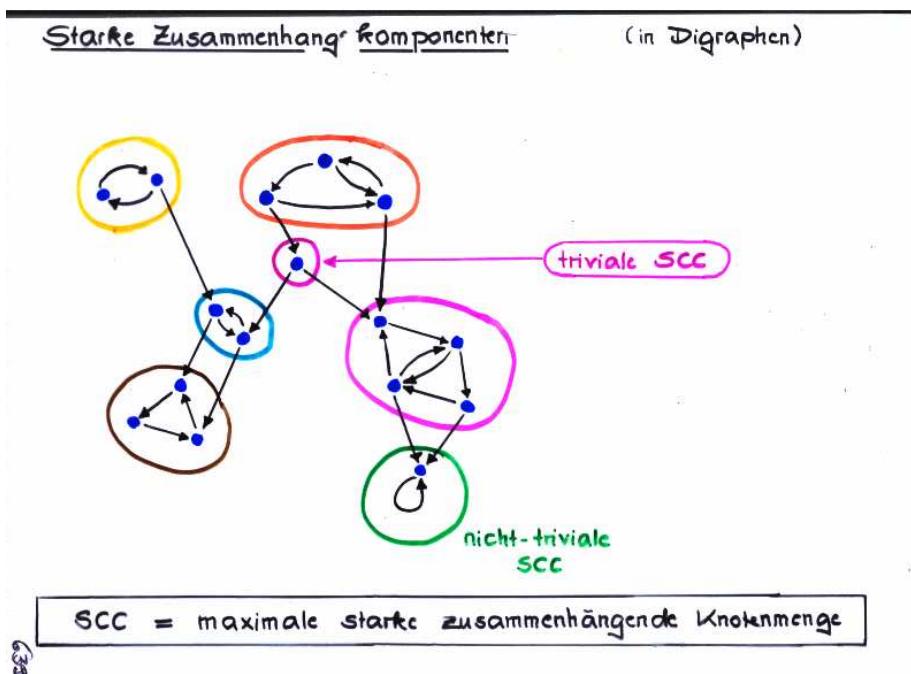
⁶²Im ungerichteten Fall gibt es zusätzlich den Begriff des *zweifachen Zusammenhangs*, der ebenfalls als ungerichtetes Analog zu starkem Zusammenhang angesehen werden kann. Diesen werden wir nicht besprechen.

Beweis. Falls v_1, v_2 in derselben starken Zusammenhangskomponente liegen und $w \in Post^*(v_2)$, dann ist w von v_1 über einen Pfad der Form $v_1, \dots, v_2, \dots, w$ erreichbar. Also ist $Post^*(v_1) \subseteq Post^*(v_2)$. Aus Symmetriegründen folgt $Post^*(v_1) = Post^*(v_2)$.

Falls umgekehrt $\text{Post}^*(v_1) = \text{Post}^*(v_2)$, so gilt $v_1 \in \text{Post}^*(v_1) = \text{Post}^*(v_2)$ und $v_2 \in \text{Post}^*(v_2) = \text{Post}^*(v_1)$. (Man beachte, dass stets $w \in \text{Post}^*(w)$, da w über den Pfad der Länge 0 von sich selbst erreichbar ist.) Also liegen v_1, v_2 in derselben starken Zusammenhangskomponente. \square

Die starken Zusammenhangskomponenten bilden also die Äquivalenzklassen der Erreichbarkeitsrelation, welche genau solche Knoten v_1, v_2 identifiziert, für die $\text{Post}^*(v_1) = \text{Post}^*(v_2)$ gilt. Insbesondere lässt sich die Knotenmenge V als disjunkte Vereinigung der starken Zusammenhangskomponenten schreiben. Jeder Knoten v liegt also in *genau* einer starken Zusammenhangskomponente. Diese ist $\{w \in \text{Post}^*(v) : v \in \text{Post}^*(w)\}$.

Bezeichnung 6.5.3 (Nicht-triviale SCC). Eine starke Zusammenhangskomponente C heißt nicht-trivial, falls der induzierte Teilgraph $G_C = (C, E_C)$ mit $E_C = E \cap (C \times C)$ wenigstens eine Kante enthält. Andernfalls heißt C trivial. \square



Offenbar ist eine starke Zusammenhangskomponente C genau dann nicht-trivial, wenn entweder $|C| \geq 2$ oder $C = \{v\}$ für einen Knoten v mit $(v, v) \in E$. Triviale starke Zusammenhangskomponenten sind also genau solche starken Zusammenhangskomponenten, die aus *genau einem* Knoten und *keiner* Kante bestehen.

Lemma 6.5.4 (Einfache Eigenschaften von SCCs).

- (a) Jeder Knoten ist in genau einer starken Zusammenhangskomponente enthalten.

- (b) Jeder Zyklus in G verläuft vollständig in einer nicht-trivialen starken Zusammenhangskomponente.
- (c) In jeder nicht-trivialen starken Zusammenhangskomponente gibt es einen Zyklus.
- (d) Für jeden Knoten v in G und jede starke Zusammenhangskomponente C gilt:

$$Post^*(v) \cap C \neq \emptyset \text{ genau dann, wenn } C \subseteq Post^*(v).$$

Insbesondere gilt: G ist genau dann azyklisch, wenn G keine nicht-trivialen starken Zusammenhangskomponenten besitzt. Die Berechnung und Analyse der starken Zusammenhangskomponenten ist also ein weiteres Verfahren zum Testen auf Zyklenfreiheit. Bevor wir auf Methoden zur Berechnung der SCCs eingehen, fassen wir unsere Ergebnisse zur Zyklenfreiheit zusammen:

Corollar 6.5.5 (Charakterisierungen zyklenfreier Digraphen). Sei G ein Digraph. Dann sind folgende Aussagen äquivalent:

- (a) G ist zyklenfrei.
- (b) Es gibt keine DFS-Rückwärtskanten in G .
- (c) G hat topologische Sortierungen.
- (d) G hat keine nicht-trivialen starken Zusammenhangskomponenten.

6.5.1 Algorithmus von Aho, Hopcroft und Ullman

Im Folgenden beschreiben wir einen Algorithmus zum Berechnen der starken Zusammenhangskomponenten. Sei $G = (V, E)$ ein Digraph wie zuvor. Wir betrachten den *inversen Graph*

$$G^{-1} = (V, E^{-1}), \quad \text{wobei } E^{-1} = \{(w, v) : (v, w) \in E\},$$

der aus G entsteht, indem alle Kanten „umgedreht“ werden. Offenbar gilt:

- (1) G und G^{-1} haben dieselben starken Zusammenhangskomponenten.

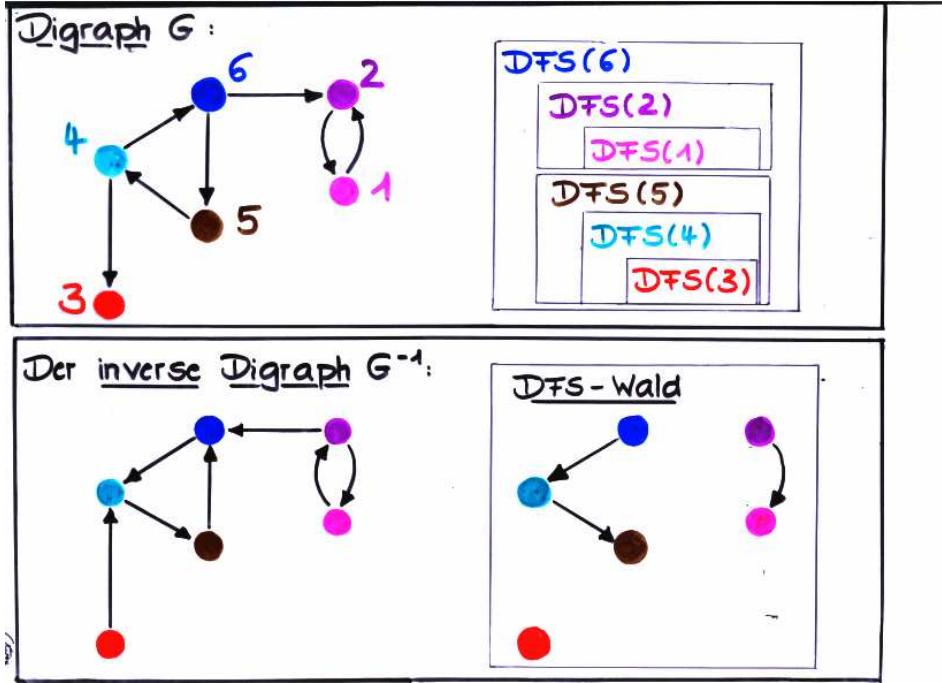
Ferner gilt für jede SCC C von G (und G^{-1}):

- (2) Ist w derjenige Knoten aus C , welcher in der DFS in G zuerst besucht wird, d.h.,

$$dfsNr(w) = \min\{dfsNr(v) : v \in C\},$$

so werden alle Knoten $v \in C \setminus \{w\}$ innerhalb $DFS(w)$ besucht. Insbesondere wird $DFS(v)$ für alle Knoten $v \in C \setminus \{w\}$ nach $DFS(w)$ gestartet und vor $DFS(w)$ abgeschlossen. Mit der zuvor besprochenen Knotenfärbung ist w also derjenige Knoten aus C , der zuletzt schwarz gefärbt wird.

Die Idee des AHU-Algorithmus zur Berechnung der SCCs ist wie folgt. Zunächst führen wir eine Tiefensuche in G durch. Diese dient dazu, die Reihenfolge v_1, \dots, v_n der Knoten zu bestimmen, in welcher die Knoten in G schwarz gefärbt werden, also die Knotenreihenfolge, die sich aus den Zeitpunkten ergibt, an denen die Knoten aus dem (Rekursions-)Stack genommen werden. Die Nummer j des Knotens v_j bezeichnen wir als *Schwarzfärbungsindex*. Wir führen nun eine Tiefensuche in G^{-1} durch, wobei als Startknoten jeweils derjenige noch nicht besuchte Knoten mit dem höchsten Schwarzfärbungsindex gewählt wird. Sei $F = (V, E_F)$ der resultierende DFS-Wald von G^{-1} .



Die Bäume in F repräsentieren die starken Zusammenhangskomponenten von G . (Genauer: Die Knotenmengen der Bäume von F bilden jeweils eine starke Zusammenhangskomponente von G .) Die beschriebene Vorgehensweise ist in Algorithmus 90 auf Seite 380 zusammengefasst.

Satz 6.5.6 (Zur Korrektheit des AHU-Algorithmus). Seien G, G^{-1} wie oben und F der DFS-Wald, welcher sich durch die Tiefensuche in G^{-1} im AHU-Algorithmus ergibt. Dann gilt für alle $v, w \in V$:

v und w liegen genau dann in derselben starken Zusammenhangskomponente von G , wenn v und w in demselben DFS-Baum von F liegen.

Beweis. Wir können o.E. annehmen, dass w eine DFS-Wurzel im DFS-Wald von G^{-1} ist. Im Folgenden verwenden wir G und G^{-1} als Index für die DFS-Aufrufe, DFS-Besuchsnummern und Nachfolgermengen, um zu verdeutlichen, daß ob wir uns auf G oder G^{-1} beziehen.

Algorithmus 90 Berechnung der starken Zusammenhangskomponenten (AHU-Algorithmus)

Führe eine Tiefensuche in G durch und ermittle die Knotenreihenfolge v_1, \dots, v_n , in welcher die Rekursionsaufrufe $DFS(v_i)$ beendet werden (also die Knoten schwarz gefärbt werden).

Konstruiere den Digraphen G^{-1} .

(* Führe eine DFS in G^{-1} durch. *)

Visited := \emptyset ;

FOR $i = n, n - 1, \dots, 1$ **DO**

IF $v_i \notin Visited$ **THEN**

$DFS_{G^{-1}}(v_i)$

FI

OD

Gib die Knotenmengen der entstandenen DFS-Bäume von G^{-1} aus.

1. Liegen v und w in derselben SCC C von G , so liegen v und w auch in derselben SCC von G^{-1} . Wie oben erwähnt (siehe (1)), ist w der erste Knoten von C der in der DFS in G^{-1} besucht wird. Da bei Aufruf von $DFS_{G^{-1}}(w)$ die anderen Knoten noch nicht besucht sind, wird jeder Knoten $x \in C \setminus \{w\}$ innerhalb $DFS_{G^{-1}}(w)$ besucht. Alle diese Knoten, und somit auch v , liegen daher im DFS-Baum von G^{-1} mit Wurzel w .
2. Sei \mathcal{T} derjenige DFS-Baum des entstandenen DFS-Walds von G^{-1} , dessen Wurzel gleich w ist. Ferner sei C diejenige starke Zusammenhangskomponente von G , welche w enthält. Wir zeigen, daß die Knotenmenge von \mathcal{T} in C enthalten ist. Hierzu ist zu zeigen, daß für jeden Knoten v in \mathcal{T} gilt:

$$w \in Post_G^*(v) \text{ und } v \in Post_G^*(w).$$

Da v in \mathcal{T} von w erreichbar ist, gilt $v \in Post_{G^{-1}}^*(w)$. Hieraus folgt:

$$w \in Post_G^*(v).$$

Es bleibt zu zeigen, daß v in G von w erreichbar ist. Für $v = w$ ist nichts zu zeigen. Sei nun $v \neq w$. Dann gilt:

(*) w wird nach v schwarz gefärbt (also $DFS_G(x)$ nach $DFS_G(v)$ beendet),

da w die Wurzel des DFS-Baums \mathcal{T} ist und v in \mathcal{T} liegt.

Falls $dfsNr_G(w) < dfsNr_G(v)$, so wird $DFS_G(v)$ innerhalb $DFS_G(w)$ aufgerufen. In diesem Fall ist v von w in G erreichbar.

Nehmen wir nun an, dass $dfsNr_G(w) > dfsNr_G(v)$. Dann betrachten wir denjenigen Knoten x in \mathcal{T} , für den $dfsNr_G(x)$ minimal ist. Offenbar gilt dann

$dfsNr_G(w) > dfsNr_G(x)$. Der Pfad von w nach x in \mathcal{T} induziert einen Pfad von x nach w in G , der – abgesehen von x – nur aus solchen Knoten besteht, die bei Aufruf von $DFS_G(x)$ noch nicht besucht sind. Daher wird w innerhalb $DFS_G(x)$ besucht. Letzteres impliziert, dass $DFS_G(w)$ vor $DFS_G(x)$ abgeschlossen wird. Dies ist nicht möglich, da in der DFS in G^{-1} Knoten w als DFS-Wurzel gewählt wird (vgl. (*)).

□

Die Laufzeit des in Algorithmus 90 skizzierten Verfahrens wird wesentlich von den durchzuführenden Tiefensuchen in G und G^{-1} dominiert. Dies ist jeweils in Zeit $\Theta(|V| + |E|)$ möglich. Damit erhalten wir folgenden Satz:

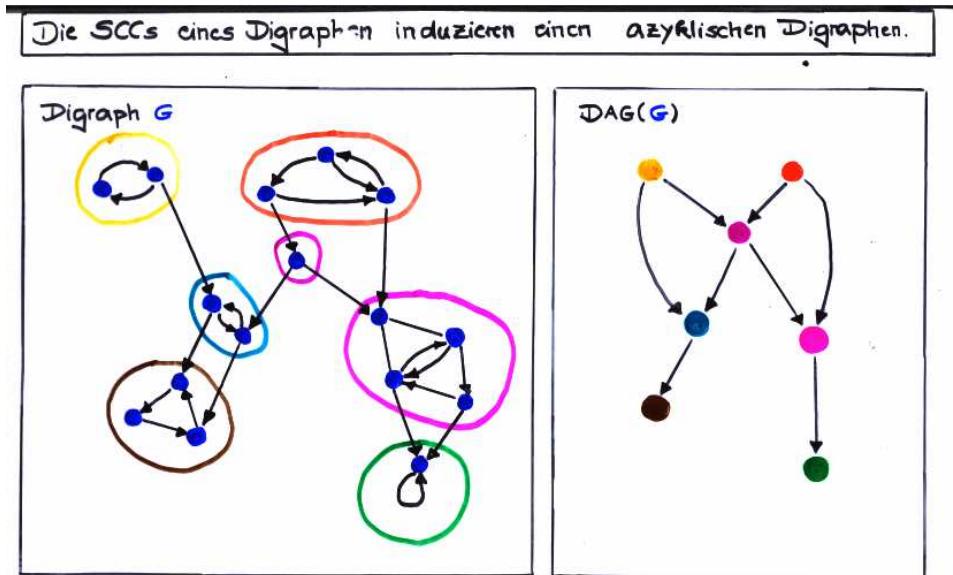
Satz 6.5.7 (Kosten des AHU-Algorithmus). Sei G ein Digraph mit n Knoten und m Kanten. Die starken Zusammenhangskomponenten von G lassen sich mit Algorithmus 90 in Zeit $\Theta(n + m)$ berechnen.

6.5.2 Der Digraph der starken Zusammenhangskomponenten

Der durch die starken Zusammenhangskomponenten induzierte azyklische Digraph ist der Digraph $DAG(G)$, dessen Knoten die starken Zusammenhangskomponenten von G sind und dessen Kantenmenge E' gegeben ist durch:

$$(C, C') \in E' \text{ gdw } C \neq C' \text{ und es gibt } v \in C, v' \in C' \text{ mit } (v, v') \in E.$$

Die Zyklenfreiheit des Digraphen der SCCs weisen wir in Lemma 6.5.8 (siehe unten) nach. Zunächst ein Beispiel:



(* Fehler auf der Folie: die gelbe SCC (links oben) hat nur eine ausgehende Kante. *)

Lemma 6.5.8. Für jeden Digraphen G ist $DAG(G)$ azyklisch.

Beweis. Wir nehmen an, dass $DAG(G)$ zyklisch ist. Sei etwa C_0, C_1, \dots, C_k ein Zyklus in $DAG(G)$, also $k \geq 1$ und $C_0 = C_k$, $(C_i, C_{i+1}) \in E'$ für $i = 0, 1, \dots, k-1$. Dann gibt es Knoten

$$v'_0, v_1, v'_1, \dots, v'_{k-1} v_{k-1}, v_k \text{ mit } v'_i, v_i \in C_i \text{ und } (v_i, v'_{i+1}) \in E'.$$

Sei $v_0 = v_k$. Da v_i, v'_i in derselben starken Zusammenhangskomponente C_i von G liegen, gibt es einen Pfad in G der Form

$$\pi = v_0, \dots, v'_0, v_1, \dots, v'_1, \dots, v_{k-1}, \dots, v'_{k-1}, \underbrace{v_0}_{=v_k}.$$

Da π ein Zyklus ist, liegen alle Knoten auf π in derselben starken Zusammenhangskomponente von G . Insbesondere gilt $C_0 = C_1 = \dots = C_{k-1} = C_k$. Da aber $(C_0, C_0) \notin E'$ ist dies nicht möglich. \square

$DAG(G)$ spiegelt exakt die Erreichbarkeitsrelation in G wider. Knoten w ist nämlich genau dann in G von v erreichbar, wenn die starke Zusammenhangskomponente C_w , welche w enthält, in $DAG(G)$ von der starken Zusammenhangskomponente C_v , welche v enthält, erreichbar ist.

Man beachte, dass G und $DAG(G)$ isomorph (also bis auf die Namen der Knoten identisch) sind, falls G ein azyklischer Graph ist. In diesem Fall sind nämlich alle SCCs trivial.

6.5.3 Elimination von Kettenregeln in kontextfreien Grammatiken

Als Anwendungsbeispiel für den durch die starken Zusammenhangskomponenten induzierten Digraphen nennen wir die Elimination von Kettenregel in kontextfreien Grammatiken.

Wir erinnern an die Abschnitt 5.5.2 auf Seite 276 eingeführten Bezeichnungen für kontextfreie Grammatiken. Unter einer Kettenregel versteht man eine Regel der Form $A \rightarrow B$, welche die Ersetzung eines Nichtterminals A durch ein anderes Nichtterminal B ermöglicht. Liegt eine kontextfreie Grammatik \mathcal{G} vor, so bestehen die ersten Schritte zur Erstellung einer äquivalenten Grammatik in Chomsky Normalform in der Elimination von ε -Regeln und Kettenregeln.

Wir betrachten hier nur die Eliminierung von Kettenregeln. Zunächst erstellen wir einen Digraphen $G = (V, E)$, dessen Knotenmenge V die Menge der Nichtterminale von \mathcal{G} ist und dessen Kantenrelation E genau durch die in \mathcal{G} vorkommenden Kettenregeln gegeben ist, also

$$E = \{(A, B) : A, B \text{ Nichtterminale in } \mathcal{G} \text{ mit } A \rightarrow B\}.$$

Wir konstruieren dann den induzierten Digraphen $DAG(G)$ und erstellen für diesen eine topologische Sortierung. Wir nehmen nun an, dass C_1, \dots, C_k die starken Zusammenhangskomponenten von G sind, so dass $i \leq j$, falls $A \rightarrow B$ eine Kettenregel in \mathcal{G} ist, wobei $A \in C_i$ und $B \in C_j$. Wir verfahren nun wie folgt:

- In jeder SCC C_i wählen wir ein Nichtterminal $A_i \in C_i$ als Repräsentanten aus.
- In jeder Regel von \mathcal{G} ersetzen wir alle Vorkommen eines Nichtterminals $A \in C_i \setminus \{A_i\}$ durch A_i ($i = 1, \dots, k$).
- Anschliessend streichen wir alle Kettenregeln der Form $A_i \rightarrow A_i$.

Nun haben alle verbleibenden Kettenregeln die Form $A_i \rightarrow A_j$, wobei $i < j$. Diese entfernen wir, indem wir in absteigender Reihenfolge hinsichtlich der topologischen Sortierung (also mit $i = k-1$ beginnend) die Kettenregel $A_i \rightarrow A_j$ durch die Regeln $A_i \rightarrow x$ ersetzen, wobei x über alle Wörter quantifiziert, für die es eine Regel der Form $A_j \rightarrow x$ gibt. Man beachte, dass durch die Betrachtung der A_i 's in absteigender topologischer Sortierung sichergestellt ist, dass keine dieser Regeln $A_j \rightarrow x$ eine Kettenregel ist. Die so entstandene Grammatik ist also tatsächlich kettenregelfrei. Die durchzuführenden Schritte sind in Algorithmus 91 auf Seite 384 zusammengefasst.

Die Größe der resultierenden Grammatik ist durch $\mathcal{O}(\text{size}(\mathcal{G})^2)$ beschränkt, sofern keine Duplikate der Regeln eingefügt werden. Dabei ist $\text{size}(\mathcal{G})$ die Anzahl der Nichtterminals in \mathcal{G} plus die Gesamtlänge aller Regeln in \mathcal{G} . Ist nämlich n_i die Gesamtlänge aller Regeln $B \rightarrow x$ in der ursprünglichen Grammatik \mathcal{G} , wobei $B \in C_i$, so ist die Gesamtlänge der Regeln $A_i \rightarrow x$ in der resultierenden Grammatik durch

$$n_i + n_{i+1} + \dots + n_k$$

beschränkt. Die Gesamtlänge aller Regeln in der generierten kettenregelfreien Grammatik ist daher durch

$$\sum_{i=1}^k i \cdot n_i \leq \underbrace{k}_{\leq \text{size}(\mathcal{G})} \cdot \underbrace{(n_1 + \dots + n_k)}_{\leq \text{size}(\mathcal{G})} = \mathcal{O}(\text{size}(\mathcal{G})^2)$$

beschränkt.

Die Korrektheit der Transformation ergibt sich aus folgenden Beobachtungen. Ersstens sind alle Kettenregeln der Form $A \rightarrow A$ effektlos und können daher ersatzlos gestrichen werden. Falls B in dem Digraphen der Kettenregeln von A erreichbar, also falls es es eine Folge von Kettenregeln der Form $A \rightarrow B_1, B_1 \rightarrow B_2, \dots, B_k \rightarrow B$ gibt, so gilt:

$$\{w \in (V \cup \Sigma)^*: A \Rightarrow^* w\} \supseteq \{w \in (V \cup \Sigma)^*: B \Rightarrow^* w\}$$

Daher können alle Regeln $B \rightarrow x$ auf A übertragen werden (in dem Sinn, dass die Regeln $A \rightarrow x$ eingefügt werden), ohne die erzeugte Sprache der Grammatik zu ändern. Liegen A

Algorithmus 91 Elimination von Kettenregeln aus kontextfreien Grammatiken

Konstruiere den durch \mathcal{G} induzierten Digraphen G der Kettenregeln.

Berechne die SCCs von G und konstruiere den induzierten Digraphen $DAG(G)$.

Erstelle eine topologische Sortierung C_1, \dots, C_k für $DAG(G)$.

Wähle in jeder SCC C_i ein Nichtterminal A_i , wobei $A_i = S$ für diejenige SCC C_i , welche das Startsymbol S von \mathcal{G} enthält.

Ersetze in allen Regeln von \mathcal{G} jedes Vorkommen eines Nichtterminals $B \notin \{A_1, \dots, A_k\}$ durch A_i , wobei i der Index derjenigen SCC ist, welche B enthält (also $B \in C_i$).

(* Nur noch A_1, \dots, A_k kommen in den Regeln von \mathcal{G} als Nichtterminale vor. *)

Streiche alle Kettenregeln der Form $A_i \rightarrow A_i$.

(* Alle verbleibenden Kettenregeln haben die Form $A_i \rightarrow A_j$ mit $i < j$. *)

FOR $i = k - 1, k - 2, \dots, 1$ **DO**

(* eliminiere Kettenregeln der Form $A_i \rightarrow B$ *)

FOR $j = i + 1, \dots, k$ **DO**

IF \mathcal{G} enthält die Kettenregel $A_i \rightarrow A_j$ **THEN**

streiche $A_i \rightarrow A_j$;

FOR ALL Regeln $A_j \rightarrow x$ in \mathcal{G} **DO**

füge $A_i \rightarrow x$ in \mathcal{G} ein

OD

FI

OD

OD

und B in derselben starken Zusammenhangskomponente des Digraphen der Kettenregeln, so gilt

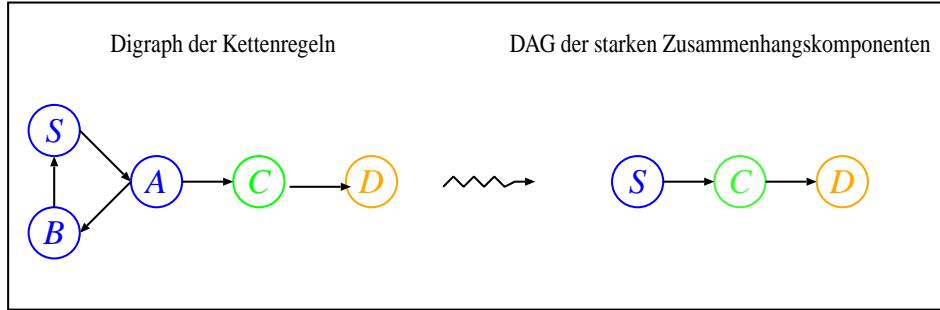
$$\{w \in (V \cup \Sigma)^*: A \Rightarrow^* w\} = \{w \in (V \cup \Sigma)^*: B \Rightarrow^* w\}.$$

Daher können A und B (und somit alle Nichtterminale einer starken Zusammenhangskomponente) identifiziert werden. Mit diesen Überlegungen ergibt sich, dass die erzeugte Sprache der Grammatik in den durchgeführten Schritten von Algorithmus 91 unverändert bleibt.

Beispiel 6.5.9 (Elimination von Kettenregeln). Wir betrachten die Grammatik \mathcal{G} mit den Nichtterminalen S, A, B, C, D , den Terminalzeichen a, c, d und den folgenden Regeln.

$$\begin{array}{lll} S \rightarrow A \mid aB & B \rightarrow S \mid Ba & D \rightarrow d \mid dDD \\ A \rightarrow B \mid C & C \rightarrow D \mid c & \end{array}$$

Zu entfernen sind hier die fünf Kettenregeln $S \rightarrow A$, $A \rightarrow B$, $A \rightarrow C$, $B \rightarrow S$ und $C \rightarrow D$. Hierzu erstellen wir den Digraphen der Kettenregeln (links im Bild) und den assoziierten DAG der SCCs (rechts), wobei wir in der Skizze rechts die SCCs mit dem jeweils ausgewählten Nichtterminal identifizieren.



Die Variablen S, A, B bilden eine starke Zusammenhangskomponente. Zusätzlich bilden jeweils C und D einelementige starke Zusammenhangskomponenten. Wir ersetzen A und B durch S und streichen die Regel $S \rightarrow S$. Man erhält die Grammatik mit folgenden Regeln:

$$S \rightarrow aS \mid aC \mid C \mid cSd \mid Sa \quad C \rightarrow D \mid c \quad D \rightarrow d \mid dDD$$

Im letzten Schritt wird zuerst $C \rightarrow D$ durch $C \rightarrow d$ und $C \rightarrow dDD$, dann $S \rightarrow C$ durch $S \rightarrow d$, $S \rightarrow dDD$ und $S \rightarrow c$ ersetzt. Die resultierende kettenregelfreie Grammatik besteht also aus dem Produktionssystem

$$S \rightarrow aS \mid aC \mid d \mid dDD \mid c \mid cSd \mid Sa \quad C \rightarrow d \mid dDD \mid c \quad D \rightarrow d \mid dDD$$

□

Weitere Anwendungsbeispiele für SCCs sind DATALOG-Programme⁶³, welche mit Hilfe des durch die starken Zusammenhangskomponenten induzierten DAGs für den Abhängigkeitsgraphen ausgewertet werden, oder die Verifikation reaktiver Systeme.

⁶³DATALOG ist eine für Datenbankanwendungen konzipierte Logikprogrammiersprache.

6.6 Das Netzwerkflussproblem

Eine Reihe praxisrelevanter Problemstellungen lassen sich als Netzwerkflussprobleme formalisieren, bei denen es darum geht, eine möglichst große Masse von einer Quelle s zu einer Senke t zu befördern.⁶⁴ Typische Instanzen sind die Frage nach dem maximalen Verkehrsfluss (Fahrzeuge pro Minute), der durch das Strassennetz einer Stadt geleitet werden kann, oder die Frage nach der maximalen Wassermenge, die durch einen Kanal fließen kann. Es gibt jedoch auch zahlreiche Anwendungen in der Informatik wie z.B. die Datenübertragung im Internet (die Frage nach dem maximalen Datenfluss von einem Rechner s zu einem Rechner t) oder die Simulation von Markovprozessen.

6.6.1 Grundbegriffe

Definition 6.6.1 (Netzwerk, Flussfunktion). Ein Netzwerk ist ein Tupel $N = (V, E, c, s, t)$ bestehend aus

- einem Digraphen $G_N = (V, E)$
- einer Kapazitätsfunktion $c : E \rightarrow \mathbb{N}_{\geq 1}$
- Knoten $s, t \in V$ mit $s \neq t$ und $Pre(s) = Post(t) = \emptyset$.

Der Knoten s wird Quelle (engl. source), der Knoten t auch Zielknoten (engl. target) genannt. Eine *Flussfunktion* für N ist eine Abbildung $f : E \rightarrow \mathbb{R}_{\geq 0}$ mit den folgenden Eigenschaften (1) und (2).

(1) $f(e) \leq c(e)$ für alle Kanten $e \in E$.

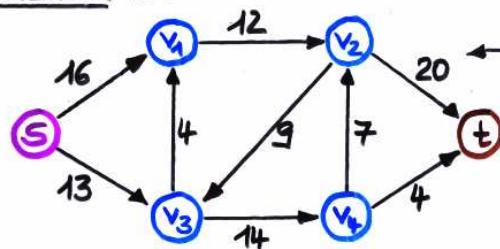
(2) Für alle Knoten $v \in V \setminus \{s, t\}$ gilt:

$$\sum_{u \in Pre(v)} f(u, v) = \sum_{w \in Post(v)} f(v, w).$$

Eigenschaft (2) wird auch *Flusserhaltungsgesetz* garantiert. Diese kann auch so gelesen werden, daß die Masse, die in einen Knoten v hineinfließt, gleich der Masse ist, die aus v hinausfließt. \square

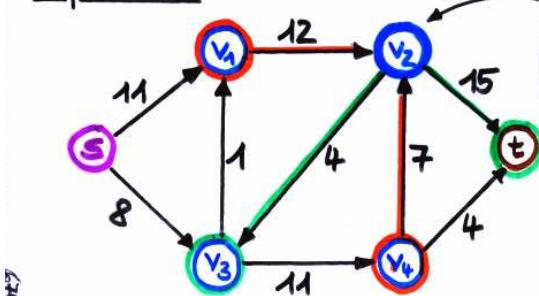
⁶⁴Die Abkürzungen s und t stehen für die englischen Bezeichnungen „source“ und „target“.

Netzwerk N :



Kapazitätsfkt., z.B.
 $c(v_2, t) = 20$

Flussfunktion für N



Flussverhaltungsgesetz für v_2
 $12 + 7 = 19 = 15 + 4$

Definition 6.6.2 (Flusswert). Der Flusswert von f ist gegeben durch

$$\text{Flow}(f) = \sum_{w \in \text{Post}(s)} f(s, w).$$

□

Intuitiv ist der Flusswert von f die Masse, die aus dem Knoten s hinausfliesst. In dem Beispiel auf der Folie ist $\text{Flow}(f) = 11 + 8 = 19$. Dies entspricht genau dem Wert $15 + 4$, der in t hineinfliest. Dies ist kein Zufall. Aufgrund des Flusserhaltungsgesetzes gilt stets (siehe Lemma 6.6.8 unten):

$$\text{Flow}(f) = \sum_{v \in \text{Pre}(t)} f(v, t)$$

Definition 6.6.3 (Maximaler Fluss, optimale Flussfunktion). Der maximale Fluss von N ist gegeben durch:

$$\text{MaxFlow}(N) = \max \{ \text{Flow}(f) : f \text{ ist eine Flussfunktion für } N \}$$

Eine Flussfunktion f für N wird *optimal* genannt, wenn $\text{MaxFlow}(N) = \text{Flow}(f)$.⁶⁵ □

Unser Ziel ist nun ein algorithmisches Verfahren zur Bestimmung einer optimalen Flussfunktion.

⁶⁵Wir werden sehen, dass es stets eine Flussfunktion f mit $\text{Flow}(f) \geq \text{Flow}(f')$ für alle anderen Flussfunktionen f' für N gibt. Damit ist sichergestellt, dass das Maximum der Flusswerte existiert.

Bemerkung 6.6.4. Etwas abweichend von der einschlägigen Literatur machen wir die vereinfachende Annahme, daß s bzw. t eine graphische Quelle bzw. Senke (d.h. Knoten ohne direkten Vorgänger bzw. Nachfolger) ist. Diese Annahme ist unwesentlich. Sie dient nur der Anschauung. Man kann auf diese Forderung verzichten; muß dann allerdings die Definition des Flusswerts durch

$$Flow(f) = \sum_{w \in Post(s)} f(s, w) - \sum_{v \in Pre(s)} f(v, s)$$

ersetzen. Später werden wir zusätzlich die Forderung stellen, daß für jede Kante $(v, w) \in E$ das Knotenpaar (v, w) nicht in E liegt. Auch diese Forderung ist unnötig. Sie dient lediglich der gedanklichen Veranschaulichung und zur Vereinfachung der Formalismen. \square

6.6.2 Das MaxFlow-MinCut-Theorem

Definition 6.6.5 (Schnitt, Schnittkapazität). Ein Schnitt für N ist eine Teilmenge S von V mit $s \in S$ und $t \notin S$. Die Kapazität eines Schnitts ist gegeben durch:⁶⁶

$$cap(S) = \sum_{\substack{v \in S \\ w \in Post(v) \setminus S}} c(v, w)$$

Die *minimale Schnittkapazität* von N ist

$$MinCut(N) = \min \{ cap(S) : S \text{ ist ein Schnitt für } N \}.$$

\square

Für das Beispiel von oben ist die Kapazität des Schnitts $S = \{s, v_1, v_4\}$ gleich

$$cap(S) = c(s, v_3) + c(v_1, v_2) + c(v_4, v_2) + c(v_4, t) = 8 + 12 + 7 + 4 = 31.$$

Eine zentrale Rolle spielt der folgenden Satz.

Satz 6.6.6 (Max-Flow-Min-Cut-Theorem). Sei N ein Netzwerk. Dann gilt:

$$MaxFlow(N) = MinCut(N).$$

Der Beweis von Satz 6.6.6 gliedert sich in zwei Teile. Im ersten Teil (siehe Lemma 6.6.9 unten) zeigen wir

(*) $Flow(f) \leq cap(S)$ für jede Flussfunktion f und jeden Schnitt S für N .

Im zweiten Teil zeigen wir, daß es eine Flussfunktion f und einen Schnitt S gibt, so dass $Flow(f) = cap(S)$; siehe Corollar 6.6.19 auf Seite 398. Diese Flussfunktion f ist dann (wegen (*)) optimal.

⁶⁶„cap“ steht für die englische Bezeichnung „capacity“.

Bezeichnung 6.6.7 (Erweiterte Kapazitätsfunktion). Wir erweitern die Kapazitätsfunktion c sowie jede Flussfunktion f für N zu Abbildungen $c : V \times V \rightarrow \mathbb{N}$ bzw. $f : V \times V \rightarrow \mathbb{R}_{\geq 0}$, indem wir $c(v, w) = f(v, w) = 0$ für $(v, w) \notin E$ setzen. Dies ermöglicht einfache Schreibweisen wie $\sum_{v \in V} c(v, w)$ statt $\sum_{v \in \text{Pre}(w)} c(v, w)$. \square

Das folgende Lemma zeigt, daß sich das Flusserhaltungsgesetz auf Schnitte überträgt: unter jeder Flussfunktion f ist die in S hineinfließende Masse gleich der aus S hinausfließenden Masse.

Lemma 6.6.8 (Nettofluss von Flussfunktionen über Schnitten). Sei N ein Netzwerk, f eine Flussfunktion für N und S ein Schnitt für N . Dann gilt:

$$\text{Flow}(f, S) = \text{Flow}(f),$$

wobei

$$\text{Flow}(f, S) = \sum_{\substack{v \in S \\ w \notin S}} f(v, w) - \sum_{\substack{v \in S \\ u \notin S}} f(u, v)$$

der so genannte Nettofluß von f über S ist.

Beweis. Für festen Knoten $v \in S \setminus \{s\}$ gilt aufgrund des Flusserhaltungsgesetzes:

$$\begin{aligned} & \sum_{w \notin S} f(v, w) - \sum_{u \notin S} f(u, v) \\ &= \sum_{w \in V} f(v, w) - \sum_{u \in V} f(u, v) - \sum_{w \in S} f(v, w) + \sum_{u \in S} f(u, v) \\ &= - \sum_{w \in S} f(v, w) + \sum_{u \in S} f(u, v) \end{aligned}$$

Wegen $\text{Pre}(s) = \emptyset$ ist $\sum_{u \in S} f(u, s) = \sum_{u \notin S} f(u, s) = 0$. Ferner ist

$$\sum_{w \notin S} f(s, w) = \text{Flow}(f) - \sum_{w \in S} f(s, w)$$

Hieraus folgt:

$$\begin{aligned}
Flow(f, S) &= \sum_{\substack{v \in S \\ w \notin S}} f(v, w) - \sum_{\substack{v \in S \\ u \notin S}} f(u, v) \\
&= \sum_{w \notin S} f(s, w) + \sum_{\substack{v \in S \setminus \{s\} \\ w \notin S}} f(v, w) - \sum_{\substack{u \in S \setminus \{s\} \\ u \notin S}} f(u, v) - \underbrace{\sum_{u \notin S} f(u, s)}_{=0} \\
&= \sum_{w \notin S} f(s, w) - \sum_{\substack{v \in S \setminus \{s\} \\ w \in S}} f(v, w) + \sum_{\substack{u \in S \setminus \{s\} \\ u \in S}} f(u, v) + \underbrace{\sum_{u \in S} f(u, s)}_{=0} \\
&= Flow(f) - \sum_{w \in S} f(s, w) - \sum_{\substack{v \in S \setminus \{s\} \\ w \in S}} f(v, w) + \sum_{\substack{u \in S \setminus \{s\} \\ u \in S}} f(u, v) + \sum_{u \in S} f(u, s) \\
&= Flow(f) - \sum_{v, w \in S} f(v, w) + \sum_{u, v \in S} f(u, v) \\
&= Flow(f)
\end{aligned}$$

□

Mit $S = \{s\}$ stimmt die Aussage von Lemma 6.6.8 genau mit dem Flusserhaltungsgesetz überein. Für $S = V \setminus \{t\}$ erhalten wir die zuvor angesprochene Formel

$$Flow(f) = \sum_{v \in Pre(t)} f(v, t).$$

Aus Lemma 6.6.8 folgt die oben erwähnte Aussage (*), nämlich:

Lemma 6.6.9 (Flusswerte sind stets durch Schnittkapazitäten nach oben beschränkt). Für jede Flussfunktion f und jeden Schnitt S für N gilt:

$$Flow(f) \leq cap(S)$$

Beweis.

$$\begin{aligned}
Flow(f) &= Flow(f, S) \\
&= \sum_{\substack{v \in S \\ w \notin S}} \underbrace{f(v, w)}_{\leq c(v, w)} - \underbrace{\sum_{\substack{v \in S \\ u \notin S}} f(u, v)}_{\geq 0} \\
&\leq \sum_{\substack{v \in S \\ w \notin S}} c(v, w) \\
&= cap(S)
\end{aligned}$$

□

Insbesondere folgt hieraus, dass die minimale Schnittkapazität eine obere Schranke für den maximalen Fluss ist. Dies ist der erste Teil von Satz 6.6.6.

Corollar 6.6.10 (Max-Flow-Min-Cut-Theorem, 1. Teil). $\text{MaxFlow}(N) \leq \text{MinCut}(N)$

6.6.3 Der Algorithmus von Ford & Fulkerson

Wir geben nun einen konstruktiven Beweis für die Existenz einer Flussfunktion f und eines Schnitts S , so daß der Flußwert von f mit der Kapazität von S übereinstimmt. Die Konstruktion einer solchen Flussfunktion f ist letztendlich ein Algorithmus zum Berechnen einer optimalen Flussfunktion.

Um eine optimale Flussfunktion zu berechnen, arbeitet man mit Knotenfolgen, die von s nach t führen, und die als Wege in dem zugrundeliegenden ungerichteten Graphen angesehen werden können. D.h. die Orientierung der Kanten kann für die Berechnung einer optimalen Flussfunktion „umgedreht“ werden. Im Folgenden machen wir die Annahme, daß für alle $v, w \in V$ gilt:

$$\text{Ist } (v, w) \in E, \text{ so ist } (w, v) \notin E.$$

Diese Annahme wird in dieser Vorlesung nur zur Vereinfachung der Formalismen gemacht. Sie erlaubt eine einfache Handhabung von Vorwärts- und Rückwärtskanten (siehe Definition 6.6.11 unten). Der Algorithmus arbeitet in derselben Weise, wenn $(v, w), (w, v) \in E$ zugelassen ist. Der einzige Unterschied ist, dass ohne unsere zusätzliche Annahme ein Knotenpaar (v, w) sowohl Vorwärts- als auch Rückwärtskante sein kann. Dies ist bei den nun folgenden Bezeichnungen zu berücksichtigen, wenn man auf obige Forderung verzichten möchte.

Definition 6.6.11 (Ungesättigte Vorwärts-, Rückwärtskanten, Restgraph). Sei f eine Flussfunktion für N und seien v, w zwei Knoten in G .

- (v, w) heißt *ungesättigte Vorwärtskante* für f , falls $f(v, w) < c(v, w)$.
- (v, w) heißt *ungesättigte Rückwärtskante* für f , falls $f(w, v) > 0$.

Im Folgenden sprechen wir meist kurz von Vorwärts- und Rückwärtskanten ohne den Zusatz „ungesättigt“.

Die Restkapazitäten sind gegeben durch die Funktion $c_f : V \times V \rightarrow \mathbb{R}_{\geq 0}$, die wie folgt definiert ist:

$$c_f(v, w) = \max\{c(v, w) - f(v, w), f(w, v)\}$$

Der Restgraph für f ist der Digraph $\text{Rest}(f) = (V, E_f)$, wobei

$$E_f = \{(v, w) \in V \times V : c_f(v, w) > 0\}.$$

□

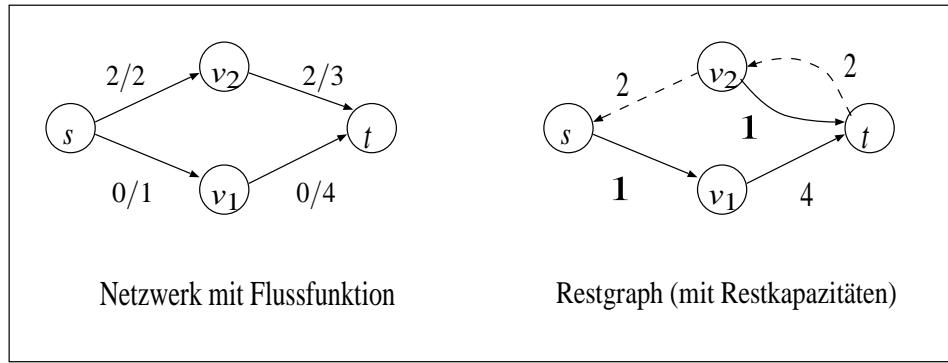
Da wir $(v, w) \notin E$ oder $(w, v) \notin E$ voraussetzen, gilt:

$$c_f(v, w) = \begin{cases} 0 & : \text{falls } (v, w) \notin E \text{ und } (w, v) \notin E \\ c(v, w) - f(v, w) & : \text{falls } (v, w) \in E \\ f(w, v) & : \text{falls } (w, v) \in E. \end{cases}$$

Also $c_f(v, w) = c(v, w) - f(v, w)$, falls (v, w) eine Vorwärtskante ist, und $c_f(v, w) = f(w, v)$, falls (v, w) eine Rückwärtskante ist.⁶⁷

Der Restgraph setzt sich also genau aus den ungesättigten Kanten zusammen. Dies sind entweder Kanten in G oder „umgedrehte“ Kanten. In den Skizzen verwenden wir durchgezogene Pfeile für ungesättigte Vorwärtskanten und gestrichelte Pfeile für ungesättigte Rückwärtskanten.

Beispiel 6.6.12 (Restgraph, Restkapazitäten). Wir betrachten ein einfaches Beispiel mit vier Knoten. Die Kantenbeschriftungen a/b sind so zu lesen, dass der erste Wert a für $f(v, w)$, der zweite Wert b für die Kapazität $c(v, w)$ steht.



In diesem Beispiel ist $0 < f(v_2, t) = 2 < 3 = c(v_2, t)$. Daher sind (v_2, t) und (t, v_2) Kanten im Restgraphen. Die Restkapazität der Vorwärtskante (v_2, t) ist

$$c_f(v_2, t) = c(v_2, t) - f(v_2, t) = 3 - 1 = 1.$$

Die Restkapazität der Rückwärtskante (t, v_2) ist $c_f(t, v_2) = f(v_2, t) = 2$. \square

Die Grobidee für die Konstruktion einer optimalen Flussfunktion f ist die folgende. Wir starten mit der Flussfunktion $f(v, w) = 0$ für alle $(v, w) \in V \times V$. Dann wird der Flußwert von f sukzessive entlang eines Pfads von s nach t im Restgraphen erhöht, indem der Wert von ungesättigten Vorwärtskanten erhöht und der Wert von ungesättigten Rückwärtskanten verkleinert wird.

Definition 6.6.13 (Zunehmender Pfad). Jeder einfache Pfad $\pi = v_0, v_1, \dots, v_{r-1}, v_r$ im Restgraphen von f mit $s = v_0$, $t = v_r$ wird zunehmender Pfad (englisch *augmenting path*) für f genannt. \square

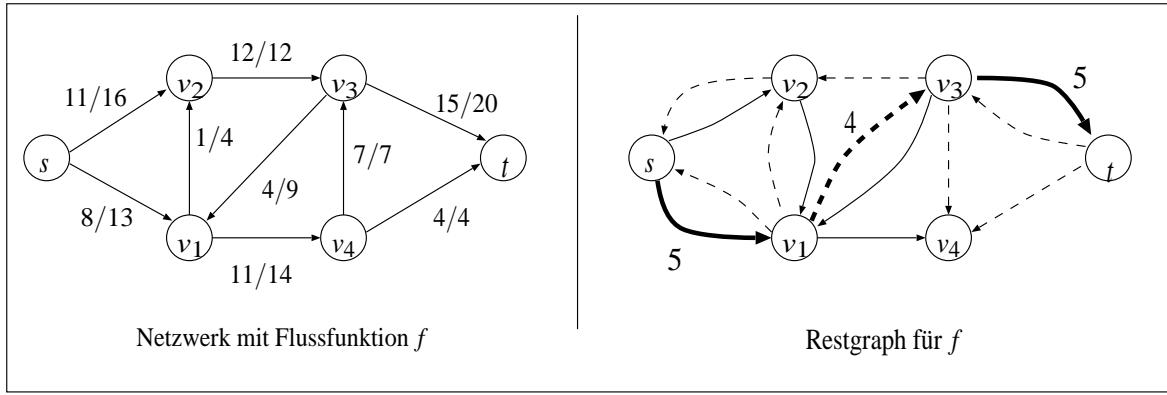


Abbildung 68: Beispiel für ein Netzwerk mit Flussfunktion und zunehmenden Pfad

Beispiel 6.6.14 (Zunehmender Pfad). In Beispiel 6.6.12 gibt es nur einen zunehmenden Pfad. Dieser ist s, v_1, t und besteht aus zwei Vorwärtskanten.

Ein weiteres Beispiel zu einem Netzwerk mit Flussfunktion und dem zugehörigen Restgraphen ist in Abbildung 68 angegeben. Dick eingezzeichnet ist der zunehmende Pfad s, v_1, v_3, t , der sich aus den beiden Vorwärtskanten (s, v_1) und (v_3, t) mit jeweils der Restkapazität $5 = 13 - 8 = 20 - 15$ und der Rückwärtskante (v_1, v_3) mit der Restkapazität $4 = f(v_3, v_1)$ zusammensetzt. \square

Die Erhöhung des Flusswerts entlang eines zunehmenden Pfads erfolgt durch Ändern des Flusses auf allen Kanten des zunehmenden Pfads um denselben Wert ε . Der Wert ε ist die minimale Restkapazität der Kanten auf dem zunehmenden Pfad. Der Fluss auf Vorwärtskanten wird um ε erhöht. Für Rückwärtskanten wird ε von dem aktuellen Fluss der betreffenden (inversen) Kante abgezogen. Die aktuelle Flussfunktion f wird also durch eine neue Flussfunktion f^* ersetzt, die sich von f nur auf den Kanten des zunehmenden Pfads unterscheidet.

Das folgende Lemma bietet die theoretische Grundlage für den oben skizzierten Flusserhöhungsschritt.

Lemma 6.6.15 (Flusserhöhung). Seien f eine Flussfunktion und $\pi = v_0, v_1, \dots, v_r$ ein zunehmender Pfad für f . Sei

$$\varepsilon = \min \{ c_f(v_j, v_{j+1}) : j = 0, 1, \dots, r-1 \}$$

die minimale Restkapazität in π und $f^* : V \times V \rightarrow \mathbb{R}_{\geq 0}$ folgende Funktion:

- $f^*(v_j, v_{j+1}) = f(v_j, v_{j+1}) + \varepsilon$, falls $(v_j, v_{j+1}) \in E$,
also falls (v_j, v_{j+1}) eine Vorwärtskante für f ist,

⁶⁷Die Formel $c_f(v, w) = \max\{c(v, w) - f(v, w), f(w, v)\}$ ist dann zu verwenden, wenn $(v, w) \in E$ und $(w, v) \in E$ zugelassen ist, da dann für $f(w, v) > 0$ und $f(v, w) < c(v, w)$ die Kante (v, w) wahlweise als Vorwärts- oder Rückwärtskante angesehen werden kann.

- $f^*(v_{j+1}, v_j) = f(v_{j+1}, v_j) - \varepsilon$, falls $(v_j, v_{j+1}) \notin E$,
also falls (v_j, v_{j+1}) eine Rückwärtskante für f ist,
- $f^*(v, w) = f(v, w)$ in allen verbleibenden Fällen.

Dann ist f^* eine Flussfunktion für N und es gilt:

$$\text{Flow}(f^*) = \text{Flow}(f) + \varepsilon.$$

Beweis. Zunächst zeigen wir, daß f^* tatsächlich eine Flussfunktion für N ist. Es ist klar, daß $0 \leq f^*(v, w) \leq c(v, w)$ für alle Kanten $(v, w) \in E$. Da sich f und f^* nur auf den Kanten des zunehmenden Pfads unterscheiden, ist das Flusserhaltungsgesetz für alle Knoten, die nicht auf π liegen, erfüllt.

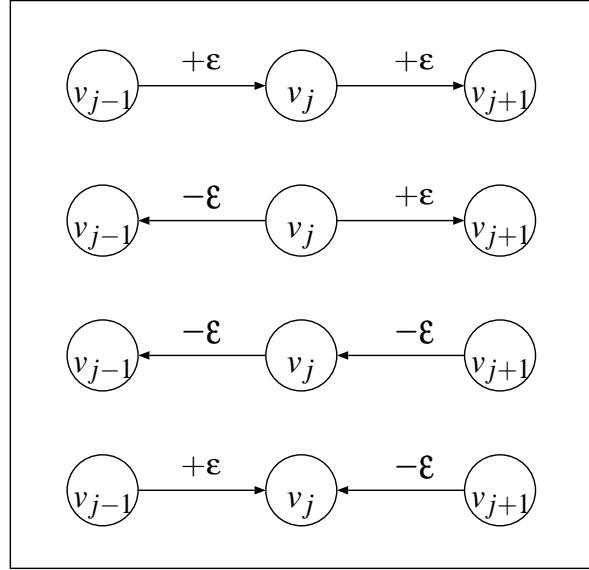


Abbildung 69: Flusserhaltungsgesetz für f^*

Für die Knoten v_j auf dem zunehmenden Pfad π (wobei $1 \leq j < r$) machen wir uns mit der in Abbildung 69 gezeigten Skizze klar, dass die Flusserhöhung keinen Einfluss auf das Flusserhaltungsgesetz an Knoten v_j hat. Im ersten Fall sind (v_{j-1}, v_j) und (v_j, v_{j+1}) Vorwärtskanten für f . Also $v_{j-1} \in \text{Pre}(v_j)$ und $v_{j+1} \in \text{Post}(v_j)$. Im ersten Fall fliesst also (hinsichtlich der neuen Flussfunktion f^*) in Knoten v_j zusätzlich die Masse „ $+\varepsilon$ “ über die Kante (v_{j-1}, v_j) hinein und derselbe Wert „ $+\varepsilon$ “ aus v_j über die Kante (v_j, v_{j+1}) hinaus. Wir erhalten:

$$\sum_{w \in \text{Post}(v_j)} f^*(v_j, w) = \sum_{w \in \text{Post}(v_j)} f(v_j, w) + \varepsilon = \sum_{u \in \text{Pre}(v_j)} f(u, v_j) + \varepsilon = \sum_{u \in \text{Pre}(v_j)} f^*(u, v_j)$$

Eine analoge Argumentation trifft auf den dritten Fall zu. Hier sind (v_{j-1}, v_j) und (v_j, v_{j+1}) Rückwärtskanten für f und der Fluss durch Knoten v_j verringert sich um ε . Im zweiten

Fall ist (v_{j-1}, v_j) eine Rückwärtskante und (v_j, v_{j+1}) eine Vorwärtskante für f . Also gilt $v_{j-1}, v_{j+1} \in Post(v_j)$. Hier bleibt die Masse, welche durch Knoten v_j fliesst, unverändert, da lediglich eine Umverteilung der aus v_j hinausfliessenden Masse stattfindet.

$$f^*(v_j, v_{j-1}) = f(v_j, v_{j-1}) - \epsilon, \quad f^*(v_j, v_{j+1}) = f(v_j, v_{j+1}) + \epsilon$$

Analoges trifft auf den vierten Fall zu, bei dem die in v_j hineinfliessenden Masse umverteilt wurde.

Wir berechnen nun den Flusswert von f^* . Da π ein Pfad von s nach t ist, gilt $v_0 = s$ und $(s, v_1) \in E$. Da s keine echten Vorgänger hat (oder auch, da π einfach ist), gilt $s \notin \{v_1, \dots, v_r\}$. Also ist (s, v_1) die einzige Kante, an welcher s beteiligt ist und welche in π als Vorwärts- oder Rückwärtskante benutzt wird. Da $(v_1, s) \notin E$, kann (s, v_1) nur eine Vorwärtskante sein. Somit ist

$$f^*(s, v_1) = f(v_0, v_1) + \epsilon$$

und $f^*(s, w) = f(s, w)$ für alle Knoten $w \in S \setminus \{v_1\}$. Wir erhalten:

$$\begin{aligned} Flow(f^*) &= \sum_{w \in S} f^*(s, w) \\ &= \sum_{w \in V \setminus \{v_1\}} f(s, w) + \underbrace{f(s, v_1) + \epsilon}_{=f^*(s, v_1)} \\ &= \sum_{w \in V} f(s, w) + \epsilon \\ &= Flow(f) + \epsilon. \end{aligned}$$

Damit erhalten wir, daß f^* alle in Lemma 6.6.15 angegebenen Bedingungen erfüllt. \square

Für die Flussfunktion aus Beispiel 6.6.14 (Seite 392) und den zunehmenden Pfad $\pi = s, v_1, v_3, t$ ist der Flusserhöhungsschritt $f \rightsquigarrow f^*$ in Abbildung 70 auf Seite 396 angegeben. Der Algorithmus von Ford & Fulkerson (siehe Algorithmus 92 auf Seite 396) benutzt Lemma 6.6.15, um sukzessive Flusserhöhungen durchzuführen. Ist f die aktuelle Flussfunktion, dann wird ein zunehmender Pfad für f gesucht. Wenn es keinen solchen gibt, ist f – wie wir in Corollar 6.6.20 auf Seite 399 sehen werden – optimal. Andernfalls wird f durch die in Lemma 6.6.15 angegebene Flussfunktion f^* ersetzt.

Die Suche nach einem zunehmenden Pfad kann ohne explizite Konstruktion des Restgraphen durchgeführt werden. Liegen Matrixdarstellungen von c und f vor, dann können diese für die Erreichbarkeitsanalyse im Restgraphen (z.B. Breitensuche gestartet mit s) verwendet werden. Die Kosten für jeden Flusserhöhungsschritt können durch $\mathcal{O}(n^2)$ abgeschätzt werden, wenn man Matrizendarstellungen für f und c verwendet. (Dabei ist $n = |V|$ die Knotenanzahl.) Eine bessere Implementierungsmöglichkeit verwendet Listendarstellungen der direkten Vorgänger- und Nachfolgermengen $Pre(\cdot)$ und $Post(\cdot)$. Diese ermöglichen die Laufzeit $\mathcal{O}(n + m) = \mathcal{O}(m)$ für die Suche im Restgraphen und den betreffenden Flusserhöhungsschritt, wobei $m = |E| \geq n$ vorausgesetzt wird.

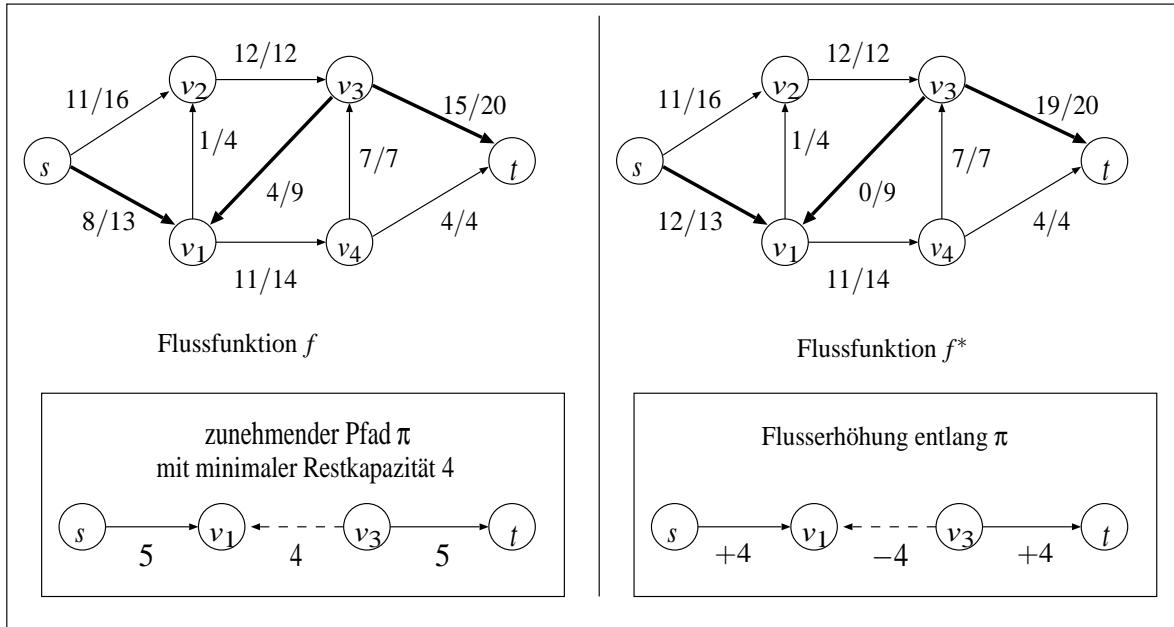


Abbildung 70: Beispiel zur Flusserhöhung entlang eines zunehmenden Pfads

Algorithmus 92 Algorithmus von Ford und Fulkerson

(* Beginne mit der trivialen Flussfunktion $f = 0$. *)

FOR ALL Knoten $v, w \in V$ **DO**
 $f(v, w) := 0$
OD

(* sukzessive Flusserhöhung entlang zunehmender Pfade *)

REPEAT

Suche einen Pfad π von der Quelle s zur Senke t im Restgraphen von f ;

IF ein solcher zunehmender Pfad π wurde gefunden **THEN**
erhöhe f entlang von π gemäß Lemma 6.6.15
FI

UNTIL es wurde kein zunehmender Pfad gefunden;

Gib die Flussfunktion f zurück.

(* $Flow(f) = MaxFlow(N)$ *)

Da wir es mit ganzzahligen Kapazitäten zu tun haben, ist auch die Flussfunktion f stets ganzzahlig. Der Wert ϵ aus dem Flusserhöhungslemma (Lemma 6.6.15 auf Seite 393) ist stets ein ganze Zahl ≥ 1 . Wegen

$$\text{Flow}(f) \leq \sum_{v \in \text{Post}(s)} c(s, v)$$

liegt nach spätestens $\sum_{v \in \text{Post}(s)} c(s, v)$ Flusserhöhungsschritten eine Flussfunktion f vor, deren Flusswert nicht weiter erhöht werden kann. Wir erhalten den folgenden Satz:

Satz 6.6.16 (Obere Schranke für die Anzahl an Flusserhöhungsschritten). Der Algorithmus von Ford & Fulkerson terminiert nach spätestens $\sum_{v \in \text{Post}(s)} c(s, v)$ Flusserhöhungsschritten.

Zunächst ist noch nicht klar, dass sich der Ford & Fulkerson Algorithmus nicht in einem lokalen Maximum verfängt. Als nächstes zeigen wir, daß jede Flussfunktion f , für die keine Flusserhöhung im obigen Sinn möglich ist, optimal ist. Daraus folgt dann die Korrektheit des Ford & Fulkerson Algorithmus.

Lemma 6.6.17 (Korrektheit des Abbruchkriteriums des Ford-Fulkerson Algorithmus). Sei f eine Flussfunktion für N , so dass es keinen zunehmenden Pfad für f gibt. Ferner sei S_f die Menge aller von der Quelle s erreichbaren Knoten im Restgraphen von f . Dann ist S_f ein Schnitt für N mit

$$\text{MaxFlow}(N) = \text{Flow}(f) = \text{cap}(S_f) = \text{MinCut}(N).$$

Beweis. Offenbar ist $s \in S_f$ und $t \notin S_f$. Also ist S_f ein Schnitt.

Wegen $\text{Flow}(g) \leq \text{cap}(S)$ für jede Flussfunktion g und jeden Schnitt S (siehe Lemma 6.6.9 auf Seite 390) genügt es zu zeigen, dass $\text{Flow}(f) = \text{cap}(S_f)$ ist.

Es gilt: Ist $v \in S_f$ und $w \in V \setminus S_f$, dann ist (v, w) keine Kante im Restgraphen. Andernfalls wäre nämlich w von s im Restgraphen von f erreichbar. Daher gilt

$$c_f(v, w) = 0 \text{ für alle Knotenpaare } (v, w) \in S_f \times (V \setminus S_f).$$

Hieraus folgt:

$$f(v, w) = c(v, w) \quad \text{und} \quad f(w, v) = 0$$

für alle Knotenpaare $(v, w) \in S_f \times (V \setminus S_f)$. Somit:

$$\text{cap}(S) = \sum_{\substack{v \in S_f \\ w \notin S_f}} c(v, w) = \sum_{\substack{v \in S_f \\ w \notin S_f}} f(v, w)$$

Andererseits (wegen Lemma 6.6.8 auf Seite 389):

$$\text{Flow}(f) = \text{Flow}(f, S_f) = \sum_{\substack{v \in S_f \\ w \notin S_f}} \underbrace{f(v, w)}_{=c(v, w)} - \underbrace{\sum_{\substack{v \in S_f \\ w \notin S_f}} f(w, v)}_{=0} = \sum_{\substack{v \in S_f \\ w \notin S_f}} c(v, w)$$

Es folgt: $\text{Flow}(f) = \text{cap}(S_f)$. □

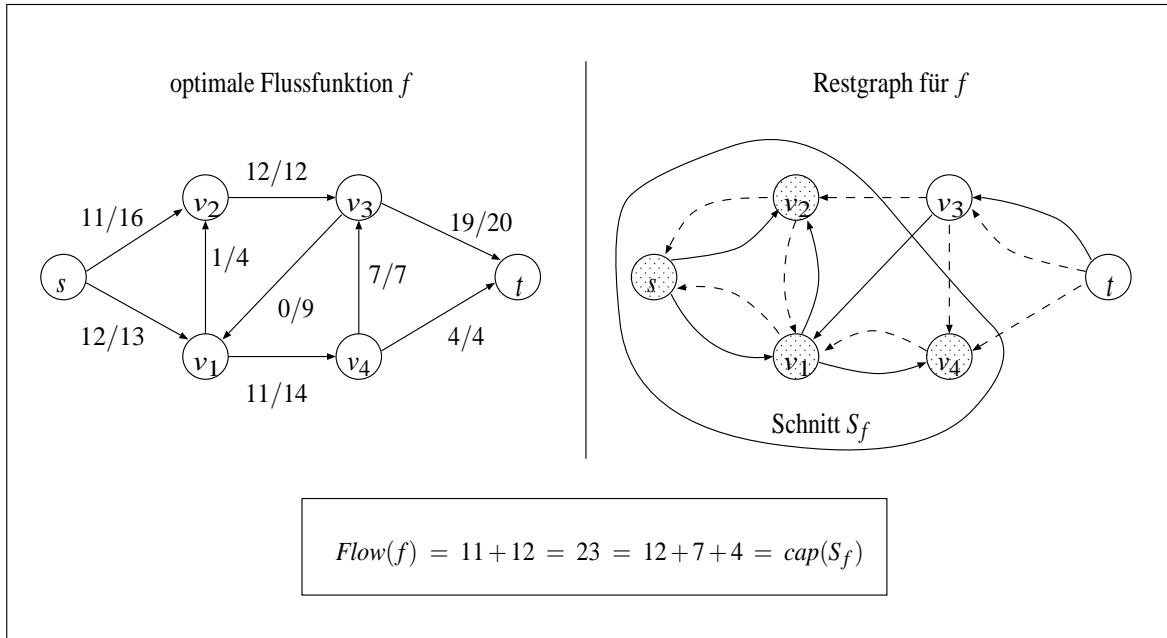


Abbildung 71: Optimale Flussfunktion f und minimaler Schnitt S_f

Beispiel 6.6.18 (Optimale Flussfunktion und minimaler Schnitt). Abbildung 71 auf Seite 398 illustriert die Aussage von Lemma 6.6.17 an einem Beispiel. Für die skizzierten Flussfunktion f ist der induzierte Schnitt bestehend aus den von der Quelle s erreichbaren Knoten im Restgraphen gleich

$$S_f = \{s, v_1, v_2, v_4\}.$$

Die Schnittkapazität von S_f

$$cap(S_f) = c(v_2, v_3) + c(v_4, v_3) + c(v_4, t) = 12 + 7 + 4 = 23$$

stimmt genau mit dem Flusswert von f überein. Daher ist f optimal (d.h. der Flusswert von f maximal) und die Kapazität von S_f minimal. \square

Die vorangegangenen Betrachtungen liefern nun die Existenz einer Flussfunktion f , für die ein Schnitt S angegeben werden kann, dessen Schnittkapazität mit dem Flusswert von f übereinstimmt (nämlich die durch den Ford und Fulkerson Algorithmus berechnete Flussfunktion). Damit ist der zweite Teil des Max-Flow-Min-Cut-Theorems (Satz 6.6.6 auf Seite 388) bewiesen:

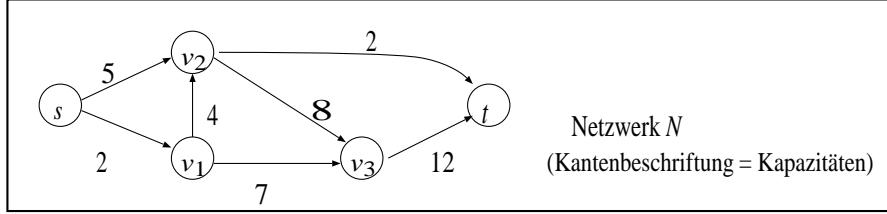
Corollar 6.6.19 (Max-Flow-Min-Cut-Theorem, 2. Teil).

Es gibt eine Flussfunktion f und einen Schnitt S mit $Flow(f) = cap(S)$.

Eine weitere Folgerung aus Lemma 6.6.17 ist die Korrektheit des Ford & Fulkerson Algorithmus:

Corollar 6.6.20 (Korrekttheit des Ford & Fulkerson Algorithmus). Der Ford & Fulkerson Algorithmus gibt eine Flussfunktion f mit $\text{MaxFlow}(N) = \text{Flow}(f)$ zurück.

Beispiel 6.6.21 (Ford & Fulkerson Algorithmus). Wir betrachten das folgende Netzwerk N . Abbildung 72 auf Seite 401 zeigt die schrittweise Erhöhung der Flussfunktion durch den Ford & Fulkerson Algorithmus.



Im ersten Schritt (Flussfunktion $f_0 = 0$) wählen wir den zunehmenden Pfad $\pi_0 = s, v_1, v_2, t$. Dieser besteht nur aus Vorwärtskanten mit den Restkapazitäten

$$c_{f_0}(s, v_1) = c(s, v_1) = 2, \quad c_{f_0}(v_1, v_2) = c(v_1, v_2) = 4, \quad c_{f_0}(v_2, t) = c(v_2, t) = 2.$$

Also wird f_0 entlang π_0 um 2 erhöht. Wir erhalten die Flussfunktion f_1 im zweiten Bild. Für diese wählen wir den zunehmenden Pfad $\pi_1 = s, v_2, v_1, v_3, t$ mit den Restkapazitäten

$$\begin{aligned} c_{f_1}(s, v_2) &= c(s, v_2) - f_1(s, v_1) &= 5-0 &= 5 \\ c_{f_1}(v_2, v_1) &= f_1(v_1, v_2) &= & 2 \\ c_{f_1}(v_1, v_3) &= c(v_1, v_3) - f_1(v_1, v_3) &= 7-0 &= 7 \\ c_{f_1}(v_3, t) &= c(v_3, t) - f_1(v_3, t) &= 12-0 &= 12 \end{aligned}$$

und führen den zugehörigen Flusserhöhungsschritt $f_1 \rightsquigarrow f_2$ durch. Da hier (v_2, v_1) eine Rückwärtskante mit minimaler Restkapazität (nämlich 2) ist, wird der Wert für (v_1, v_2) auf 0 zurückgesetzt. Im Restgraphen für f_2 gibt es nur noch einen einzigen zunehmenden Pfad, nämlich s, v_2, v_3, t , mit den Restkapazitäten

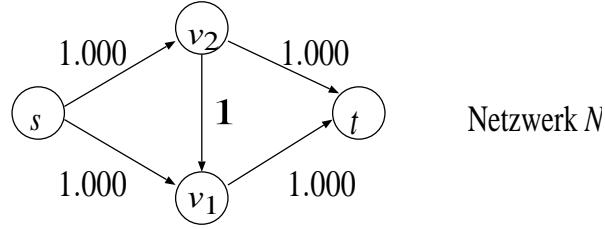
$$\begin{aligned} c_{f_2}(s, v_2) &= c(s, v_2) - f_2(s, v_1) &= 5-2 &= 3 \\ c_{f_2}(v_2, v_3) &= c(v_2, v_3) - f_2(v_2, v_3) &= 8-0 &= 8 \\ c_{f_2}(v_3, t) &= c(v_3, t) - f_2(v_3, t) &= 12-2 &= 10. \end{aligned}$$

Die minimale Restkapazität ist also 3 und wir erhalten die Flussfunktion f_3 im unteren Teil der Skizze. In dem induzierten Restgraphen ist t von s nicht erreichbar, also ist f_3 eine optimale Flussfunktion. Der maximale Fluss ist also

$$\text{MaxFlow}(N) = \text{Flow}(f_3) = 5 + 2 = 7.$$

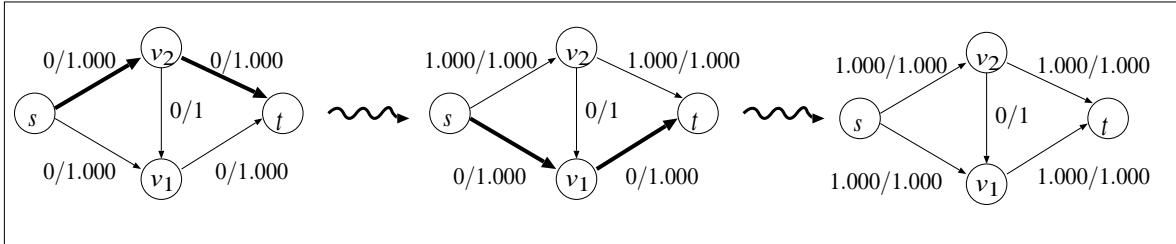
Tatsächlich stimmt dies mit der Kapazität des Schnitts $S_{f_3} = \{s\}$ der von s im Restgraphen von f_3 erreichbaren Knoten überein. \square

Zur Laufzeit des Ford und Fulkerson Algorithmus. Die Laufzeit des Algorithmus (Anzahl an Flusserhöhungsschritten) ist abhängig von der Wahl der zunehmenden Pfade. Wir demonstrieren diese Aussage an folgendem Extrembeispiel, in dem ein Netzwerk mit vier Knoten gegeben ist:



Werden abwechselnd die zunehmenden Pfade $\pi_1 = s, v_2, v_1, t$ (mit Vorwärtskante (v_2, v_1)) und $\pi_2 = s, v_1, v_2, t$ (mit Rückwärtskante (v_1, v_2)) und jeweils der Restkapazität 1 gewählt, so sind insgesamt 2.000 Flusserhöhungsschritte durchzuführen. Siehe Abbildung 73 auf Seite 402.

Tatsächlich genügen jedoch zwei Flusserhöhungsschritte, wenn man zuerst den zunehmenden Pfad s, v_1, t , dann den zunehmenden Pfad s, v_2, t betrachtet. Der Flusswert wird in beiden Schritten um 1.000 erhöht.



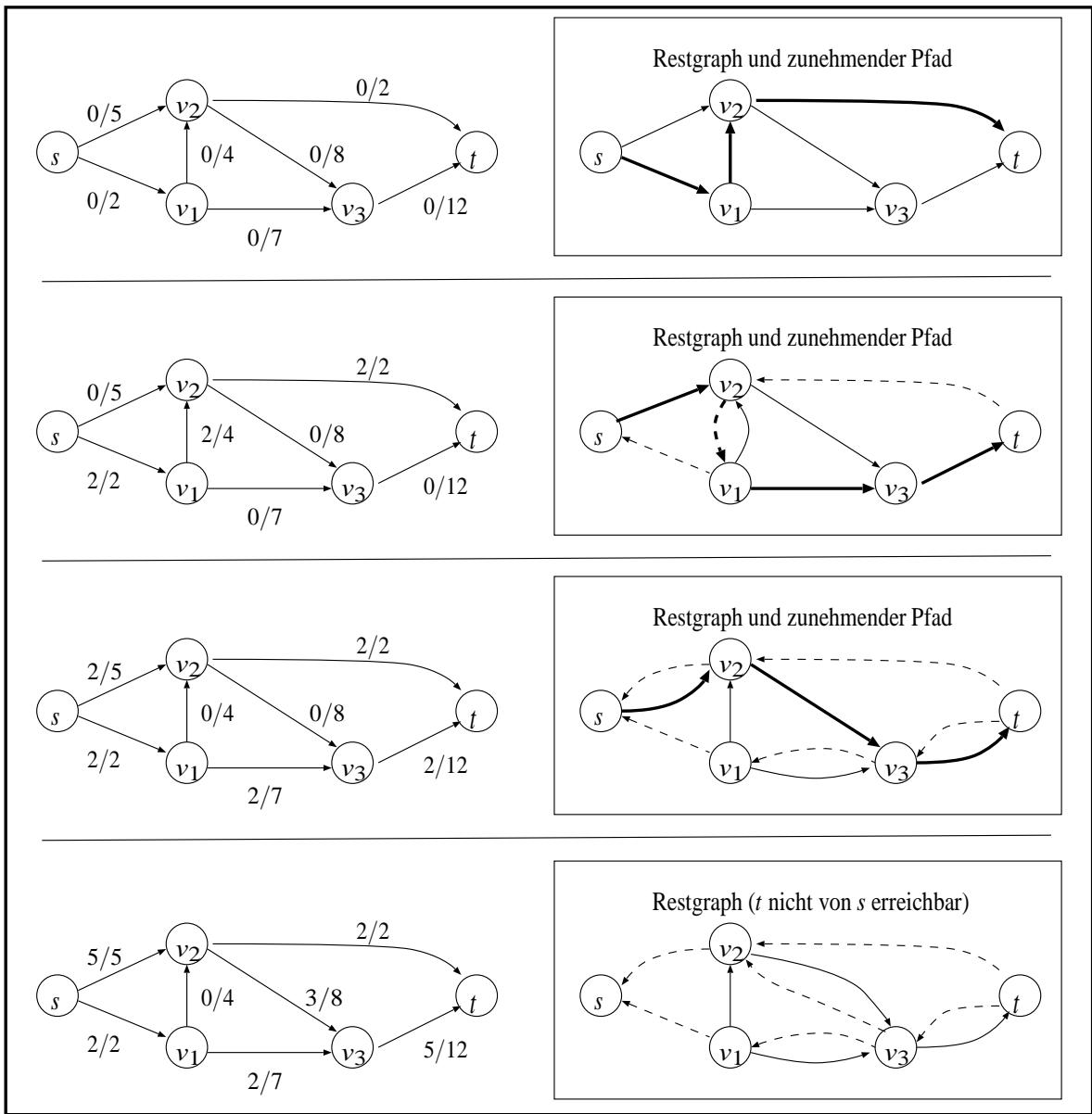


Abbildung 72: Beispiel zum Ford & Fulkerson Algorithmus

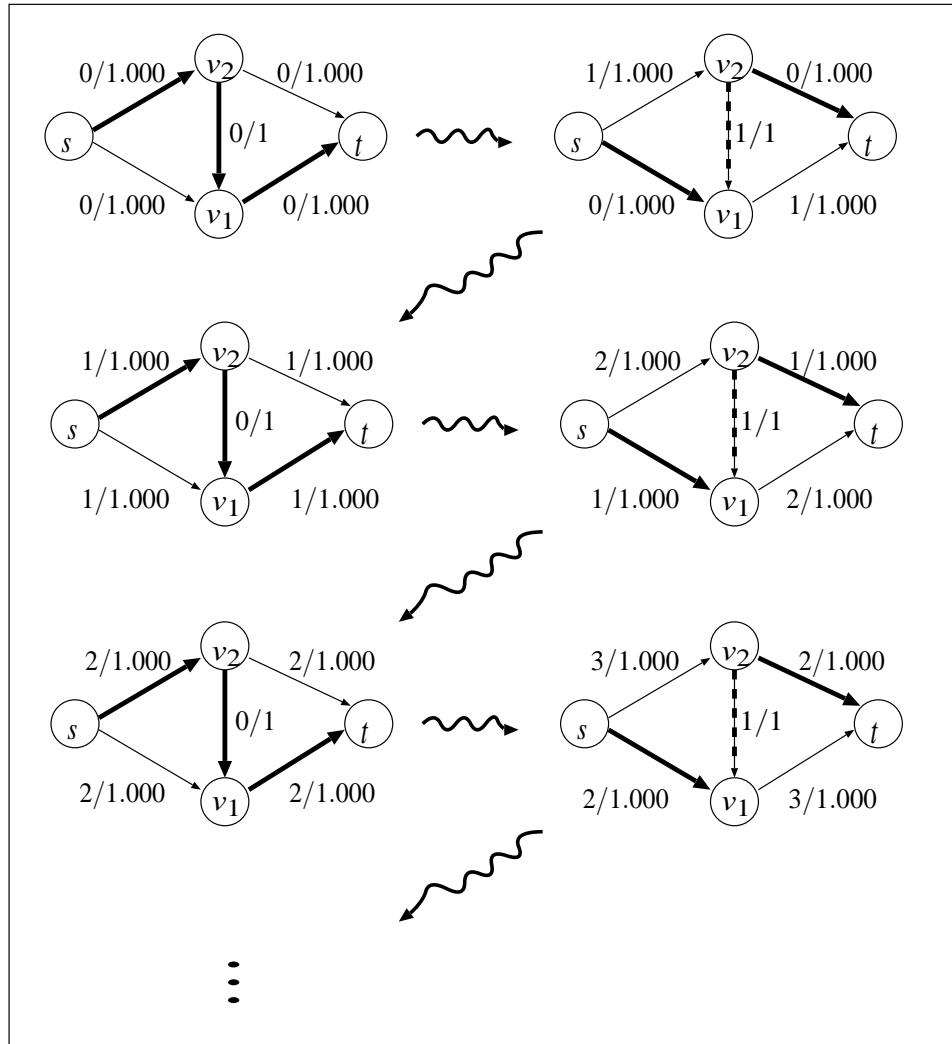


Abbildung 73: Extrembeispiel zum FF-Algorithmus mit 2.000 Flusserhöhungsschritten

6.6.4 Die Edmond-Karp Strategie für die Suche nach zunehmenden Pfaden

Obiges Extrembeispiel zeigt, dass bei willkürlicher Wahl von zunehmenden Pfaden (also Pfaden von s nach t im betreffenden Restgraphen) die Anzahl an Flusserhöhungsschritten von den Kapazitäten abhängen und daher sehr gross sein kann. Polynomielle Laufzeitbeschränkung kann jedoch durch den Einsatz einer sehr einfachen Strategie bei der Suche nach zunehmenden Pfaden erreicht werden:

„wähle stets einen *kürzesten* zunehmenden Pfad“

Dabei bezieht sich der Begriff eines *kürzesten* Pfads auf die Länge der Pfade (also die Kantenanzahl). Die Suche nach einem *kürzesten* zunehmenden Pfad kann mit einer *Breitensuche* im Restgraphen gestartet mit Knoten s realisiert werden (siehe Abschnitt 3.2.4 auf Seite 91 ff).

Abbildung 74 auf Seite 407 zeigt die schrittweise Erhöhung der Flussfunktion durch den Edmond-Karp Algorithmus angewandt auf das Netzwerk aus Beispiel 6.6.21 (Seite 399). Wir analysieren nun die Laufzeit von Algorithmus 92 mit der Edmond-Karp Strategie zur Wahl zunehmender Pfade. Hier sowie im Folgenden ist $n = |V|$ die Knotenanzahl, $m = |E|$ die Kantenanzahl und es wird $m \geq n$ vorausgesetzt.

Satz 6.6.22 (Anzahl an Flusserhöhungsschritten mit der Edmond-Karp Strategie). Wird im Ford und Fulkerson Algorithmus (Algorithmus 92) stets ein *kürzester* zunehmender Pfad gewählt, so ist die Anzahl an Flusserhöhungsschritten durch $\mathcal{O}(nm)$ beschränkt.

Beweis. Im Folgenden sei $f_0, \pi_0, f_1, \pi_1, f_2, \pi_2, \dots, \pi_r, f_{r+1}$ die alternierende Folge von Flussfunktionen f_i und zugehörigen *kürzesten* zunehmenden Pfaden π_i im Restgraphen von f_i , die durch den Ford und Fulkerson Algorithmus unter Einsatz der Edmond-Karp Strategie berechnet wird. Es ist also $f_0 = 0$ und f_{r+1} eine optimale Flussfunktion. Zu zeigen ist, daß die Anzahl $r + 1$ an durchgeführten Flusserhöhungsschritten durch $\mathcal{O}(nm)$ nach oben beschränkt ist. Wir zeigen nun:

(*) Ist $e = (v, w) \in E \cup E^{-1}$ eine Kante, so daß $e = (v, w)$ in π_i und $e^{-1} = (w, v)$ in π_j vorkommt, wobei $0 \leq i < j \leq r$, dann gilt:

$$|\pi_j| \geq |\pi_i| + 2$$

Dabei ist $|\pi| = \text{length}(\pi)$ die Länge von π , also die Anzahl an Kanten in π .

Wenn (*) nachgewiesen ist, ist die weitere Argumentation wie folgt.

In jedem Flusserhöhungsschritt $f_i \rightsquigarrow f_{i+1}$ wird mindestens eine Kante $e = (v, w)$ des betreffenden zunehmenden Pfads π_i „voll ausgeschöpft“, womit gemeint ist, daß eine Flusserhöhung um die Restkapazität $c_{f_i}(e)$ stattfindet.

- Handelt es sich bei e um eine Vorwärtskante für f_i , so ist $f_{i+1}(e) = c(e)$.
- Ist e eine Rückwärtskante für f_i , so ist $f_{i+1}(e) = 0$.

In keinem der beiden Fälle ist e eine Kante im Restgraphen von f_{i+1} . Bevor dieselbe Kante e also in einem späteren Flusserhöhungsschritt $f_h \rightsquigarrow f_{h+1}$ entlang π_h eingesetzt werden kann und auch dort voll ausgeschöpft wird (d.h. die minimale Restkapazität $c_{f_h}(e)$ unter allen in π_h vorkommenden Kanten hat), muss die inverse Kante $e^{-1} = (w, v)$ in einem der zunehmenden Pfade π_j mit $i < j < h$ vorkommen. Aus (*) folgt:

$$|\pi_i| \leq |\pi_j| - 2 \leq |\pi_h| - 4$$

Wird also Kante $e = (v, w)$ in den Pfaden $\pi_{i_0}, \pi_{i_1}, \dots, \pi_{i_\rho}$ voll ausgeschöpft (in obigem Sinn), dann gibt es eine Indexfolge $j_0, j_1, \dots, j_{\rho-1}$, so daß

- $i_0 < j_0 < i_1 < j_1 < \dots < i_{\rho-1} < j_{\rho-1} < i_\rho$,
- $e^{-1} = (w, v)$ kommt in $\pi_{j_0}, \pi_{j_1}, \dots, \pi_{j_{\rho-1}}$ vor,
- $1 \leq |\pi_{i_0}| \leq |\pi_{j_0}| - 2 \leq |\pi_{i_1}| - 4 \leq |\pi_{j_1}| - 6 \leq \dots \leq |\pi_{i_\rho}| - 4\rho$.

Da die π_i 's jeweils kürzeste Pfade von s nach t im Restgraphen von f_i sind, sind die π_i 's einfache Pfade und es gilt somit:

$$|\pi_i| \leq n - 1 \quad \text{für alle } i \in \{i_0, i_1, \dots, i_\rho\}.$$

Hieraus folgt jedoch, daß jedes $e \in E \cup E^{-1}$ höchstens ca. $\frac{n}{4}$ -mal in einem Flusserhöhungsschritt voll ausgeschöpft werden kann. Genauer gilt:

$$1 \leq |\pi_{i_0}| \leq |\pi_{i_\rho}| - 4\rho \leq n - 1 - 4\rho$$

und $1 \leq n - 1 - 4\rho$ genau dann, wenn $\rho \leq \frac{n-2}{4} = \frac{n}{4} - \frac{1}{2}$. Also ist $\frac{n}{4} + \frac{1}{2}$ die maximale Anzahl an Flusserhöhungsschritten, an denen die gegebene Kante e beteiligt sein und dort die minimale Restkapazität haben kann.

Hieraus folgt:

$$\begin{aligned} & \text{Anzahl an Flusserhöhungsschritten} \\ & \leq \underbrace{|E \cup E^{-1}|}_{=2m} \cdot \left(\frac{n}{4} + \frac{1}{2}\right) \\ & \leq 2m \cdot \left(\frac{n}{4} + \frac{1}{2}\right) = \frac{nm}{2} + m \\ & = \mathcal{O}(nm) \end{aligned}$$

Beweis von (*). Im Folgenden bezeichne $\ell_i(x, y)$ die Länge eines kürzesten Pfads von Knoten x nach Knoten y im Restgraphen von f_i . Es gilt also $\ell_i(s, t) = |\pi_i|$. Wir zeigen nun, daß für alle i mit $0 \leq i \leq r-1$ und für alle Knoten v gilt:

$$(**) \quad \ell_{i+1}(s, v) \geq \ell_i(s, v) \quad \text{und} \quad \ell_{i+1}(v, t) \geq \ell_i(v, t)$$

Wir weisen nur die erste Aussage $\ell_{i+1}(s, v) \geq \ell_i(s, v)$ nach. Die Argumentation für $\ell_{i+1}(v, t) \geq \ell_i(v, t)$ ist analog.

Falls $\ell_{i+1}(s, v) = \infty$, dann die Ungleichung trivialerweise erfüllt. Wir nehmen nun an, daß v von s im Restgraphen von f_{i+1} erreichbar ist, also daß $\ell_{i+1}(s, v) < \infty$. Sei s, v_1, \dots, v_k ein kürzester Pfad im Restgraphen von f_{i+1} , der von $v_0 = s$ nach $v_k = v$ führt (also $k = \ell_{i+1}(s, v)$). Dann gilt:

$$(\ast\ast\ast) \quad \ell_i(s, v_{j+1}) \leq \ell_i(s, v_j) + 1$$

Dies ist wie folgt einsichtig.

- Ist (v_j, v_{j+1}) eine Kante im Restgraphen von f_i , so gilt $\ell_i(s, v_{j+1}) \leq \ell_i(s, v_j) + 1$ trivialerweise.
- Wenn (v_j, v_{j+1}) keine Kante im Restgraphen von f_i ist, dann muß sich der Flusswert der inversen Kante (v_{j+1}, v_j) im Flusserhöhungsschritt $f_i \rightsquigarrow f_{i+1}$ verändert haben. Andernfalls könnte (v_j, v_{j+1}) im Restgraphen für f_{i+1} nicht vertreten sein.

Also liegt die Kante (v_{j+1}, v_j) auf dem zunehmenden Pfad π_i . Da π_i ein kürzester Pfad von s nach t ist und da v_{j+1} unmittelbar vor v_j in π_i vorkommt, folgt $\ell_i(s, v_{j+1}) = \ell_i(s, v_j) - 1$. Also ist auch in diesem Fall $(\ast\ast\ast)$ erfüllt.

Mit $(\ast\ast\ast)$ erhalten wir:

$$\begin{aligned} \ell_i(s, v) &= \ell_i(s, v_k) \\ &\leq \ell_i(s, v_{k-1}) + 1 \\ &\leq \ell_i(s, v_{k-2}) + 2 \\ &\quad \vdots \\ &\leq \ell_i(s, \underbrace{v_0}_{=s}) + k = \underbrace{\ell_i(s, s)}_{=0} + k \\ &= k = \ell_{i+1}(s, v) \end{aligned}$$

Damit ist der Beweis von $(\ast\ast)$ abgeschlossen. Wir benutzen nun $(\ast\ast)$ zum Nachweis von (\ast) . Sei also $e = (v, w) \in E \cup E^{-1}$ ein Kante, für die e in π_i und $e^{-1} = (w, v)$ in π_j vorkommt und wobei $i < j$ gilt. π_i und π_j haben also die Form

$$\pi_i = s, \dots, v, w, \dots, t, \quad \pi_j = s, \dots, w, v, \dots, t.$$

Da beide kürzeste Pfade von s nach t in den Restgraphen für f_i bzw. f_j sind, gilt:

- (i) $|\pi_i| = \ell_i(s, v) + \ell_i(v, t)$
- (ii) $|\pi_j| = \ell_j(s, w) + 1 + \ell_j(v, t)$

$$(iii) \quad \ell_i(s, w) = \ell_i(s, v) + 1$$

Aus (**) folgt wegen $i < j$:

$$(iv) \quad \ell_j(s, w) \geq \ell_i(s, w), \quad \ell_j(v, t) \geq \ell_i(v, t)$$

Wir kombinieren diese Aussagen und erhalten:

$$\begin{aligned} |\pi_j| &\stackrel{(ii)}{=} \ell_j(s, w) + 1 + \ell_j(v, t) \\ &\stackrel{(iv)}{\geq} \ell_i(s, w) + 1 + \ell_i(v, t) \\ &\stackrel{(iii)}{=} \ell_i(s, v) + 1 + 1 + \ell_i(v, t) \\ &\stackrel{(i)}{=} |\pi_i| + 2 \end{aligned}$$

□

Da kürzeste Wege von s nach t in den jeweiligen Restgraphen mit der Breitensuche in Zeit $\mathcal{O}(n+m) = \mathcal{O}(m)$ (falls $m = |E| \geq n = |V|$ vorausgesetzt wird) bestimmt werden können, erhalten wir:

Corollar 6.6.23 (Kosten des FF-Algorithmus mit der Edmond-Karp Strategie). Wird im Ford und Fulkerson Algorithmus die Edmond-Karp Strategie zur Bestimmung zunehmender Pfade eingesetzt, so ist die Laufzeit durch $\mathcal{O}(nm^2)$ beschränkt. Dabei wird $n \leq m$ vorausgesetzt, wobei n die Knoten- und m die Kantenanzahl ist.

Bemerkung 6.6.24. Die Terminierung des Ford & Fulkerson Algorithmus ist zunächst nur für natürliche Kapazitäten gesichert, da sich dann in jedem Flusserhöhungsschritt der Flusswert um mindestens 1 vergrößert. Für positive, reelle Kapazitäten kann der Ford & Fulkerson Algorithmus jedoch endlos laufen. (Wir verzichten auf die Angabe eines Beispiels hierfür.) Mit der Edmond-Karp Strategie spielt es jedoch keine Rolle, ob die Kapazitäten ganzzahlig oder reellwertig sind. □

In der Literatur wurden eine Reihe von anderen Algorithmen für das Netzwerkflußproblem vorgeschlagen, die eine bessere Laufzeit haben. Der bekannteste unter diesen ist der Algorithmus von Dinic, dessen Laufzeit durch $\mathcal{O}(n^3)$ beschränkt ist. Wir verzichten auf Erläuterung hierzu.

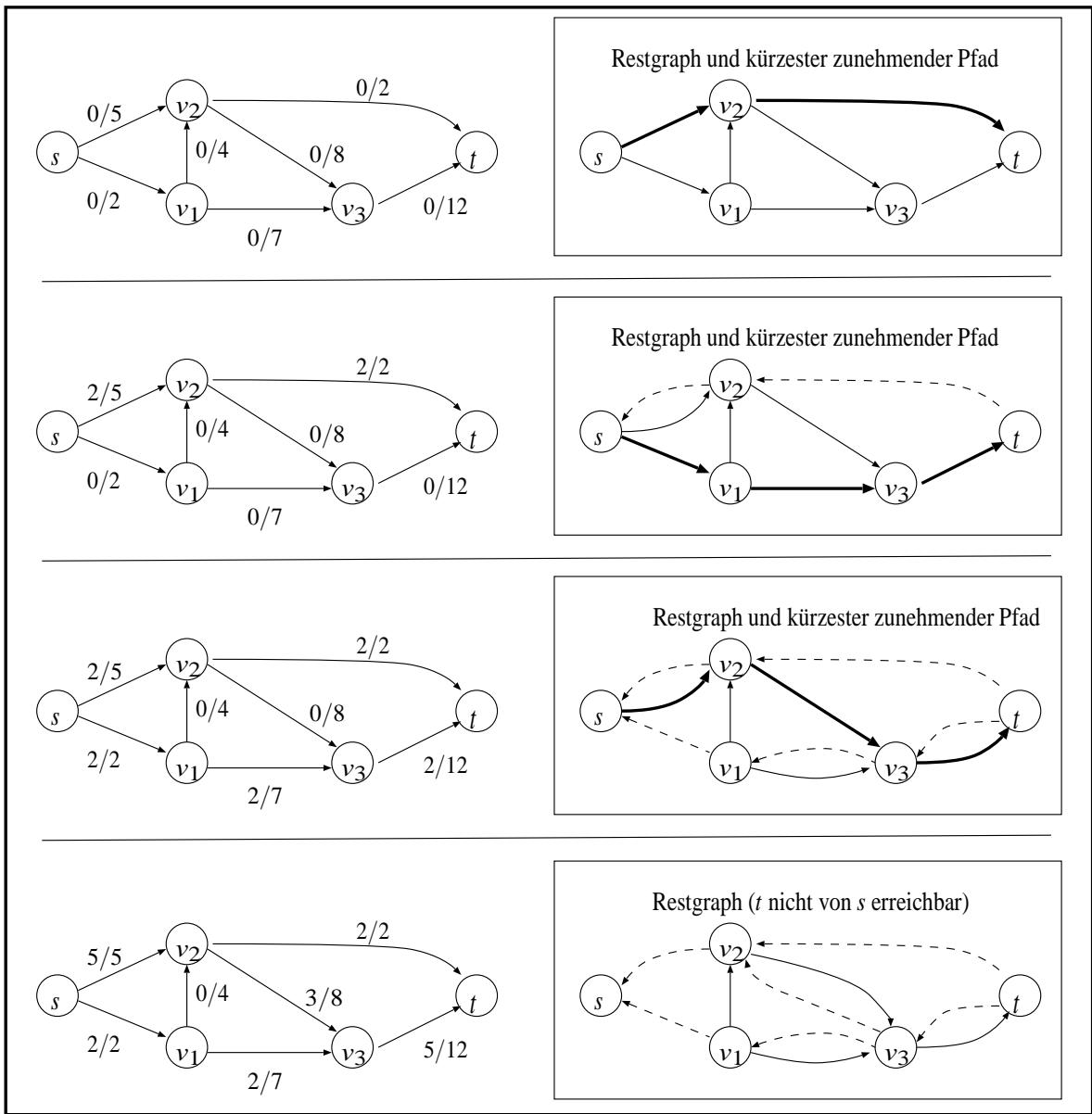
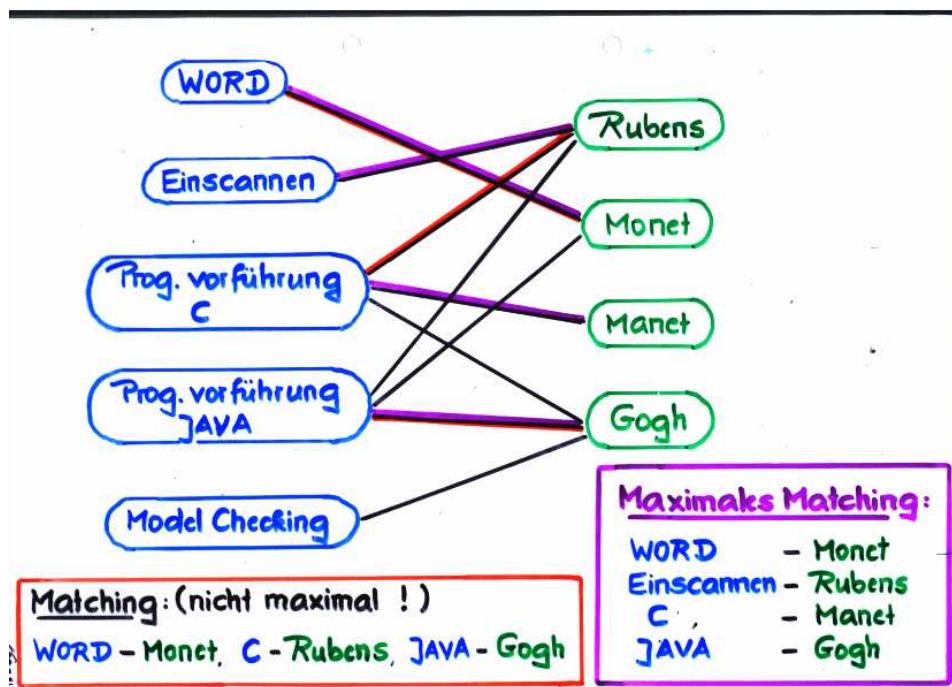


Abbildung 74: Beispiel zum Edmond-Karp Algorithmus

6.7 Bipartites Matching

Matching-Probleme suchen nach einer Auswahl an Kanten, so daß jeder Knoten höchstens auf einer der ausgewählten Kanten liegt. Typische Instanzen sind Zuordnungsprobleme, die sich z.B. bei der Zuteilung von (nur exklusiv nutzbaren) Ressourcen an Prozesse ergeben.

Als Beispiel betrachten wir ein Matchingproblem aus dem Universitätsalltag. In einem Raum stehen vier (nach Malern benannte) Rechner, die völlig unterschiedlich ausgestattet sind. Weiter liegt eine Liste von auszuführenden Jobs vor, die allesamt die exklusive Nutzung eines Rechners (mit gewisser Ausstattung) erfordern. Gefragt ist nach einer Zuordnung Jobs-Rechner, welche die Ausführung maximal vieler Jobs ermöglicht. Wir formalisieren das Problem durch einen ungerichteten Graphen. Wir stellen die Jobs und Rechner als Knoten dar. Eine Kante führt genau dann von Job J zu Rechner R , wenn J auf R ausführbar ist (d.h. R verfügt über die für Job J notwendige Ausstattung). Gesucht ist eine Auswahl an Kanten, so daß keine zwei ausgewählten Kanten gemeinsame Knoten haben und so daß die Anzahl der Kanten maximal ist.



Es gibt sehr viele Varianten von Matchingproblemen. Wir betrachten hier nur eine einfache Variante, in der ein bipartiter Graph zugrundegelegt und nach einem Matching maximaler Kardinalität gefragt wird. Bipartite Graphen sind ungerichtete Graphen, deren Knotenmenge in zwei disjunkte Teilmengen zerfällt, so daß jede Kante zwischen diesen beiden Teilmengen verläuft.

Definition 6.7.1 (Bipartiter Graph, (maximales) Matching). Sei $G = (V, E)$ ein ungerichteter Graph. G heißt *bipartit* (zweigeteilt), falls es nichtleere Knotenmengen V_L und V_R gibt, so daß

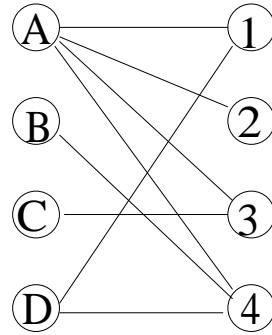
- (1) $V = V_L \cup V_R$, $V_L \cap V_R = \emptyset$
- (2) für jede Kante $(v, w) \in E$ ist $\{v, w\} \cap V_L \neq \emptyset$ und $\{v, w\} \cap V_R \neq \emptyset$.

Ein *Matching* für G ist eine Kantenmenge $M \subseteq E$, so dass jeder Knoten von G auf höchstens einer Kante von M liegt, also wenn für alle Kanten $(v, w), (v', w') \in M$ gilt:

Aus $(v, w) \neq (v', w')$ folgt $\{v, w\} \cap \{v', w'\} = \emptyset$.

Ein Matching M heißt *maximal*, wenn $|M| \geq |M'|$ für alle Matchings M' von G . \square

Beispiel 6.7.2. Viele Autoren motivieren Matchingprobleme mit „Heiratsproblemen“. Wir wollen uns dem nicht verschließen und geben ebenfalls ein derartiges, hochgradig praxisrelevantes Beispiel. Vier Personen (A, B, C und D) haben unter vier weiteren Personen (1,2,3,4) diejenigen ausgewählt, die sie sich als Ehepartner(in) wünschen, und umgekehrt. Anhand dieser Information bildet eine Heiratsagentur nun solche potentiellen Pärchen, die sich gegenseitig das JA-Wort zur Eheschließung geben möchten. Gesucht ist eine Pärchenbildung, bei der nur Wunschpaare zulässig sind und die Anzahl an Heiratsvermittlungen maximal ist.



Hier ist $V_L = \{A, B, C, D\}$ und $V_R = \{1, 2, 3, 4\}$. Die Kanten stehen für die Wunschpaare. Z.B. ist

$$M = \{(A, 2), (B, 4), (C, 3), (D, 1)\}$$

ein maximales Matching. \square

6.7.1 Reduktion des Matchingproblems auf das Netzwerkflussproblem

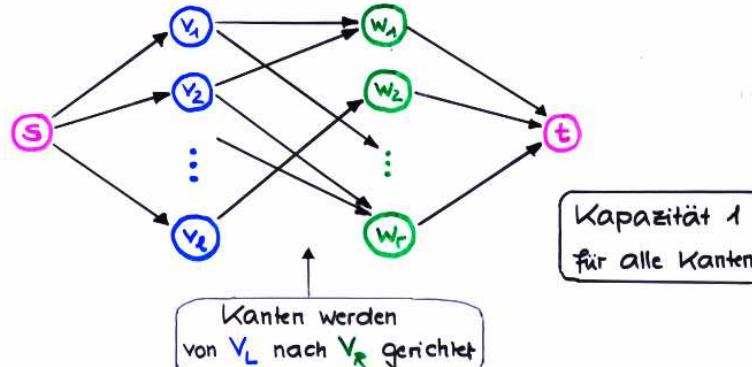
Wir beschreiben nun ein Verfahren, das zu einem gegebenen bipartiten Graphen $G = (V, E)$ mit der Zerlegung $V = V_L \cup V_R$ ein maximales Matching bestimmt. Die Lösungsidee ist, G mit einem Netzwerk N_G zu assoziieren. N_G entsteht aus G , indem eine Quelle s und ein Zielknoten t zu G hinzugefügt werden. Wir verbinden s mit den Knoten $v \in V_L$, alle Knoten $w \in V_R$ mit t und richten die Kanten von G so, daß stets der V_L -Knoten der Anfangsknoten ist. Weiter legen wir die Einheitskapazitätsfunktion zugrunde.

Das Matching Problem:

Gegeben: bipartiter ungerichteter Graph

Gesucht: maximales Matching

Lösungsmethode: Berechne den maximalen Fluss in dem Netzwerk



Definition 6.7.3 (Netzwerk N_G). Sei $G = (V, E)$ ein bipartiter Graph, wobei $V = V_L \cup V_R$ wie oben. Das zu G gehörende Netzwerk N_G ist

$$N_G = (V \cup \{s, t\}, E', c, s, t),$$

wobei $s, t \notin V$, $s \neq t$. Die Kantenrelation E' von N_G ist definiert durch:

$$E' = \{(s, v) : v \in V_L\} \cup \{(w, t) : w \in V_R\} \cup \{(v, w) \in E : v \in V_L, w \in V_R\}.$$

Die Kapazitätsfunktion c ist gegeben durch $c(e) = 1$ für alle $e \in E'$. Eine 0-1-Flußfunktion für N_G ist eine Flußfunktion f für N_G mit $f(e) \in \{0, 1\}$ für alle $e \in E'$. \square

Folgender Satz erläutert die Eins-zu-Eins-Beziehung zwischen den Matchings in G und den 0-1-Flußfunktionen in N_G .

Satz 6.7.4 (Matchings für G versus Flußfunktionen für N_G). Sei $G = (V, E)$ ein bipartiter Graph mit der Partitionierung $V = V_L \cup V_R$.

- (a) Zu jedem Matching M gibt es eine 0-1-Flußfunktion f_M für N_G mit $\text{Flow}(f_M) = |M|$.
- (b) Zu jeder 0-1-Flußfunktion f für N_G gibt es ein Matching M_f für G mit $\text{Flow}(f) = |M_f|$.

Beweis. In Teil (a) definieren wir die 0-1-Flußfunktion f_M wie folgt:

$$f_M(s, v) = f_M(w, t) = f_M(v, w) = 1, \quad \text{falls } v \in V_L, w \in V_R \text{ und } (v, w) \in M$$

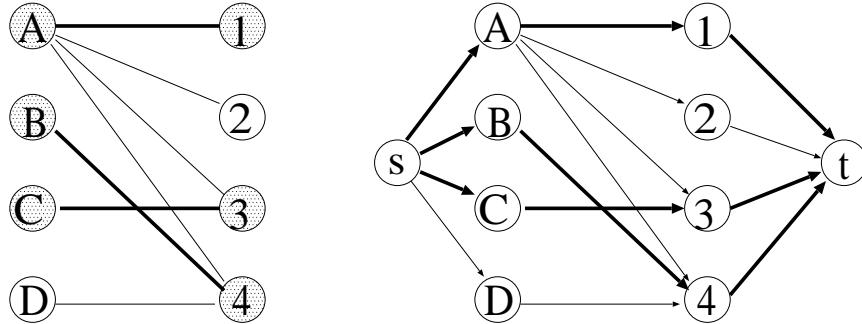
und $f_M(v, w) = 0$ in allen anderen Fällen. Dann ist f_M eine 0-1-Flußfunktion mit dem Flußwert $|M|$. Man beachte, dass das Flusserhaltungsgesetz für f_M aufgrund der Matching-Eigenschaft „jeder Knoten liegt auf höchstens einer Kante“ erfüllt ist.

In Teil (b) verwenden wir das Matching

$$M_f = \{(v, w) \in V_L \times V_R : f(v, w) = 1\}$$

Auch hier entspricht das Flusserhaltungsgesetz für f der Matching-Eigenschaft von M_f . Offenbar gilt $\text{Flow}(f) = |M_f|$. \square

Beispiel 6.7.5 (Matching für G und induzierte Flussfunktion für N_G). Zur Illustration der Aussage von Teil (a) in Satz 6.7.4 betrachten wir ein Beispiel. In folgender Skizze ist links ein bipartiter Graph gezeichnet, in dem durch die dicken Kanten ein Matching angedeutet wird. Auf der rechten Seite der Skizze ist das zugehörige Netzwerk abgebildet. Die durch das Matching induzierte Flussfunktion wird hier durch die dicken Kanten angedeutet, welchen der Wert 1 zugeordnet ist, während alle dünnen Kanten den Wert 0 haben.



Offenbar stimmt der Flußwert (nämlich 3) mit der Kardinalität des gegebenen Matchings überein. \square

Aus Satz 6.7.4 folgern wir, dass die Kardinalität $|M|$ eines maximalen Matchings mit dem maximalen Fluß in N_G übereinstimmt und dass ein maximales Matching mit einem Algorithmus für das Netzwerkflussproblem berechnet werden kann. Genauer gilt:

Satz 6.7.6 (Reduktion des Matchingproblems auf das Netzwerkflussproblem). Sei G wie zuvor. Dann gilt:

$$\text{MaxFlow}(N_G) = \max\{|M| : M \text{ ist ein Matching für } G\}$$

Weiter gilt: Falls f eine optimale Flußfunktion für N_G ist, die mit der Ford & Fulkerson-Methode bestimmt wurde, so ist f eine 0-1-Flussfunktion und das zu f gehörende Matching M_f ein maximales Matching für G . f kann mit der Ford & Fulkerson-Methode in Zeit $\mathcal{O}(nm)$ bestimmt werden. Dabei ist $m = |E|$ die Kantenanzahl, $|V| = n$ die Knotenanzahl und es wird $m \geq n/2$ vorausgesetzt.⁶⁸

⁶⁸Die Voraussetzung $m \geq n/2$ ist insofern für ungerichtete Graphen natürlich, da andernfalls isolierte Knoten vorliegen.

Beweis. Die ersten Aussagen folgen aus Satz 6.7.4. Die Aussage über die Laufzeit resultiert aus der Beobachtung, daß die maximale Anzahl an Flußerhöhungsschritten gleich

$$\sum_{v \in Post(s)} c(s, v) = \sum_{v \in V_L} c(s, v) = |V_L| = \mathcal{O}(n)$$

ist. Die Suche nach zunehmenden Pfaden kann mit einer Tiefen- oder Breitensuche in Zeit $\mathcal{O}(n+m) = \mathcal{O}(m)$ durchgeführt werden.⁶⁹ Also ist die Laufzeit des Ford & Fulkerson Algorithmus durch $\mathcal{O}(nm)$ beschränkt. \square

6.7.2 Perfektes Matching und der Satz von Hall

Abschließend diskutieren wir die Frage, wann ein perfektes Matching existiert. Damit ist ein Matching gemeint, welches allen Knoten $v \in V_L$ einen „Partner“ in V_R zuordnet. Dies ist selbstverständlich nur für $|V_L| \leq |V_R|$ möglich.

Definition 6.7.7 (Perfektes Matching). Sei $G = (V, E)$ ein bipartiter Graph mit $V = V_L \cup V_R$ wie in Definition 6.7.1 und M ein Matching für G . M wird perfekt genannt, falls $|M| = |V_L|$. \square

Folgender Satz gibt ein hinreichendes und notwendiges Kriterium für die Existenz eines perfekten Matchings. Dabei verwenden wir folgende Bezeichnung. Für $W \subseteq V_L$ ist

$$Post(W) = \bigcup_{w \in W} Post(w)$$

die Menge aller Knoten $u \in V_R$, die über wenigstens eine Kante mit einem W -Knoten verbunden sind.

Satz 6.7.8 (Satz von Hall). Sei $G = (V, E)$ ein bipartiter Graph mit $V = V_L \cup V_R$ wie zuvor. Dann sind äquivalent:

- (a) G hat ein perfektes Matching.
- (b) $|Post(W)| \geq |W|$ für jede Knotenmenge $W \subseteq V_L$.

Beweis. (a) \implies (b): Liegt ein perfektes Matching M vor und ist $W \subseteq V_L$, so enthält M für jeden Knoten $w \in W$ genau eine Kante (w, w_M) . Die Knoten w_M liegen also in $Post(W)$ und sind paarweise verschieden. Also ist $|Post(W)| \geq |W|$.

(b) \implies (a): Im Folgenden setzen wir $|Post(W)| \geq |W|$ für alle $W \subseteq V_L$ voraus. Zu zeigen ist die Existenz eines perfekten Matchings. Wir betrachten das zugehörige Netzwerk N_G sowie eine $\{0, 1\}$ -Flussfunktion f in N_G mit maximalem Flusswert. Unser Ziel ist nun der Nachweis, dass der maximale Fluss $Flow(f)$ gleich $|V_L|$ ist (vgl. Satz 6.7.6 auf Seite 411). Sei S_f die Menge aller von der Quelle s erreichbaren Knoten im Restgraphen von f . In Lemma 6.6.17 auf Seite 397 wurde gezeigt, dass S_f ein Schnitt ist und der Flusswert von f mit der Kapazität von S_f übereinstimmt, also:

⁶⁹Es ist hier für die angegebene Laufzeit $\mathcal{O}(nm)$ unerheblich, ob die Edmond-Karp Strategie eingesetzt wird oder nicht.

$$Flow(f) = cap(S_f) \quad (1)$$

Wir zeigen nun zunächst, dass

$$Post(V_L \cap S_f) \subseteq S_f \quad (2)$$

Beweis von (2). Sei v ein beliebiger Knoten in $S_f \cap V_L$ und $w \in Post(v)$. Wir zeigen, dass $w \in S_f$. Hierzu nehmen wir an, dass $w \notin S_f$. Dann ist (v, w) keine Kante im Restgraphen. Also ist die Kante (v, w) gesättigt, d.h.

$$f(v, w) = 1.$$

Aufgrund des Flusserhaltungsgesetzes und da $Pre(v) = \{s\}$, ist $f(s, v) = 1$. Also ist auch (s, v) keine Kante im Restgraphen. Knoten v kann also im Restgraphen nur über eine Rückwärtskante von der Quelle s erreicht werden. Daher gibt es ein $w' \in Post(v) \cap S_f$ mit

$$f(v, w') = 1.$$

Also sind w und w' zwei direkte Nachfolger von v mit $f(v, w) = f(v, w') = 1$. Dies ist aufgrund des Flusserhaltungsgesetzes nicht möglich, da (s, v) die einzige in v einführende Kante ist. Diese Überlegungen zeigen, dass $Post(v) \subseteq S_f$ für alle $v \in S_f \cap V_L$. Damit ist der Beweis von (2) abgeschlossen.

Aus (2) ergibt sich:

(*) Für alle Kanten (u, v) in N_G mit $u \in S_f$ und $v \in V \setminus S_f$ gilt $u = s$ oder $v = t$.

Alle anderen Kanten in N_G haben nämlich die Form (u, v) , wobei $u \in V_L$ und $v \in V_R$. Mit der Annahme, dass $u \in S_f$ würde $v \in S_f$ gelten (siehe (2)). Beachte ferner, dass $s \in S_f$ und $t \notin S_f$.

Hieraus folgt:

$$\begin{aligned} cap(S_f) &\stackrel{\text{Def. 6.6.5}}{=} \sum_{\substack{(u, v) \text{ Kante in } N_G \\ u \in S_f, v \notin S_f}} \underbrace{c(u, v)}_{=1} \\ &\stackrel{(*)}{=} \underbrace{|\{(s, v) : (s, v) \text{ Kante in } N_G \text{ und } v \notin S_f\}|}_{\text{gdw } v \in V_L \setminus S_f} \\ &\quad + \underbrace{|\{(u, t) : (u, t) \text{ Kante in } N_G \text{ und } u \in S_f\}|}_{\text{gdw } u \in S_f \cap V_R} \\ &= |V_L \setminus S_f| + |S_f \cap V_R| \end{aligned}$$

Ferner gilt $Post(S_f \cap V_L) \subseteq S_f \cap V_R$, ebenfalls wegen (2) und aufgrund der Tatsache, dass alle Kanten in G zwischen V_L und V_R verlaufen. Wir erhalten:

$$cap(S_f) = |V_L \setminus S_f| + |S_f \cap V_R| \geq |V_L \setminus S_f| + |Post(S_f \cap V_L)| \quad (3)$$

Da wir $|Post(W)| \geq |W|$ für alle $W \subseteq V_L$ voraussetzen, erhalten wir mit $W = S_f \cap V_L$:

$$|Post(S_f \cap V_L)| \geq |S_f \cap V_L| \quad (4)$$

Wir kombinieren nun (1), (3) und (4):

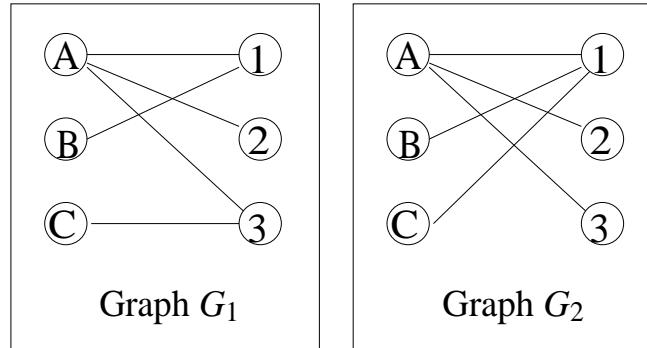
$$\begin{aligned} Flow(f) &\stackrel{(1)}{=} cap(S_f) \\ &\stackrel{(3)}{\geq} |V_L \setminus S_f| + |Post(S_f \cap V_L)| \\ &\stackrel{(4)}{\geq} |V_L \setminus S_f| + |S_f \cap V_L| \\ &= |V_L| \end{aligned}$$

Andererseits gilt $Flow(f) \leq |V_L|$ für alle Flussfunktionen f in N_G . (Da die Kanten (s, u) mit $u \in V_L$ die einzigen von s ausgehenden Kanten in N_G sind.) Also gilt:

$$Flow(f) \geq |V_L| \geq Flow(f)$$

und somit $Flow(f) = |V_L|$. □

Beispiel 6.7.9 (Perfektes Matching). Betrachten wir den bipartiten Graphen G_1 der folgenden Skizze, so stellen wir fest, dass das Hall'sche Kriterium erfüllt ist, da nämlich A , B , und C jeweils mindestens einen Nachfolger haben, die Nachfolgermengen von $\{A, B\}$, $\{B, C\}$ und $\{A, C\}$ jeweils mindestens zweielementig sind und $\{A, B, C\}$ drei Nachfolger hat. Tatsächlich ist $M = \{(A, 2), (B, 1), (C, 3)\}$ ein perfektes Matching.



Der Graph G_2 besitzt jedoch kein perfektes Matching, da die zweielementige Knotenmenge $\{B, C\} \subseteq V_L$ nur einen Nachfolgeknoten, nämlich Knoten 1, hat. □

Der Satz von Hall liefert zwar kein zufriedenstellendes algorithmisches Kriterium für die Existenz eines perfekten Matchings (was die Betrachtung aller Teilmengen W von V_L bedeuten würde). Jedoch ermöglicht er einen einfachen Nachweis der Existenz eines perfekten Matchings in Graphen, in denen alle Knoten denselben Verzweigungsgrad haben. Solche Graphen werden auch regulär genannt.

Satz 6.7.10 (Existenz eines perfekten Matchings in regulären Graphen). Sei $G = (V, E)$ ein bipartiter Graph und $k \geq 1$, so dass $|Post(v)| = k$ für alle $v \in V$. Dann besitzt G ein perfektes Matching.

Beweis. Sei $V = V_L \cup V_R$ wie zuvor. Wir verwenden das „Hall’sche Kriterium“ (siehe Satz 6.7.8 oben) und zeigen, dass $|Post(W)| \geq |W|$ für alle Teilmengen W von V_L .

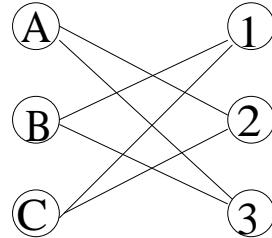
Sei $W \subseteq V_L$. Ferner sei E_1 die Menge aller Kanten $(w, v) \in E$ mit $w \in W$ und E_2 die Menge aller Kanten $(u, v) \in E$ mit $v \in Post(W)$. Für $(w, v) \in E_1$ gilt $w \in W$ und somit $v \in Post(w) \subseteq Post(W)$, also $(w, v) \in E_2$. Wir erhalten $E_1 \subseteq E_2$. Hieraus folgt

$$k|W| \stackrel{(*)}{=} |E_1| \leq |E_2| \stackrel{(*)}{=} k|Post(W)|$$

und somit $|W| \leq |Post(W)|$.

Die Gleichungen $(*)$ ergeben sich aus der Voraussetzung, dass jeder Knoten genau k Nachfolger hat. Ferner sind je zwei von zwei verschiedenen Knoten $w_1, w_2 \in W$ ausgehenden Kanten verschieden, da $w_1 \neq w_2$ aufgrund der Voraussetzung, dass G bipartit ist und dass $W \subseteq V_L$, ausgeschlossen ist. Analoges gilt für $Post(W) \subseteq V_R$. \square

Beispiel 6.7.11. Folgende Skizze zeigt einen regulären bipartiten Graphen mit $|Post(v)| = 2$ für alle Knoten v .



Hier ist $M = \{(A, 3), (B, 1), (C, 2)\}$ ein perfektes Matching. \square

So, das war's. Mehr haben wir leider nicht mehr geschafft.

Ich wünsche allen Hörerinnen und Hörern schöne Semesterferien und viel Erfolg in der Klausur und in den Vordiplomsprüfungen.

The End

7 Mustererkennung

Mustererkennungsprobleme treten z.B. beim Editieren von Texten, in der Genetik (DNA-Analyse) oder Spracherkennung auf. Wir beschäftigen uns hier mit der einfachsten Variante, in der ein Vorkommen eines Musters in einem Text gesucht ist.

Wir erinnern an die in Definition 5.4.1 auf Seite 250 eingeführten Bezeichnungen. Sei Σ ein beliebiges Alphabet. Σ^* bezeichnet die Menge aller Wörter (Strings) gebildet aus den Zeichen von Σ . Σ^+ steht für die Menge der nichtleeren Wörter über Σ . Ein Suffix (Endstück) eines Wortes w' ist ein Wort w , so daß $w' = xw$ für ein Wort $x \in \Sigma^*$. Entsprechend ist ein Präfix (Anfangsstück) von w' ein Wort w , so daß $w' = wx$ für ein $x \in \Sigma^*$.

Das Mustererkennungsproblem. Gegeben ist ein Text $T = t_1t_2\dots t_n \in \Sigma^+$ und ein Muster $M = m_1m_2\dots m_k \in \Sigma^+$. Im Folgenden nehmen wir $n > k$ und $t_1, \dots, t_n, m_1, \dots, m_k \in \Sigma$ an. $T[i]$ bezeichnet das an Position i beginnende Teilwort von T , welches die Länge k (Länge des Musters) hat, also

$$T[i] = t_i t_{i+1} \dots t_{i+k-1}.$$

Gefragt ist, ob das Muster M im Text T vorkommt, also ob es eine Textposition $i \in \{1, 2, \dots, n - k + 1\}$ gibt, so daß $T[i] = M$. Z.B. kommt das Muster $M = \text{PASS}$ im Text $T = \text{SPASSVOGEL}$ vor; genauer gilt $M = T[2]$. Ein Muster M kann natürlich auch mehrere Vorkommen in einem Text haben. Z.B. kommt das Muster $M = \text{ana}$ an zwei Stellen im Text $T = \text{banana}$ vor, nämlich $M = T[2] = T[4]$.

Das naive Verfahren prüft für jede Textposition $i \in \{1, \dots, n - k + 1\}$, ob $T[i] = M$ ist, wobei der Vergleich von $T[i]$ und M zeichenweise vollzogen wird. Siehe Algorithmus 93 auf Seite 416. Die Laufzeit ist offenbar linear in der Text- und Musterlänge, also $\Theta(n \cdot k)$.

Algorithmus 93 Naives Verfahren zur Mustererkennung

```
i := 0;  
WHILE  $i < n - k + 1$  DO  
    i := i + 1;  
    vergleiche  $M$  und  $T[i]$  zeichenweise; (*  $\Theta(k)$  Vergleiche *)  
    IF  $T[i] = M$  THEN  
        return „JA. Das Muster steht im Text ab Position  $i$ .“  
    FI  
OD  
(* kein vorzeitiger Abbruch der WHILE-Schleife, also ist  $M$  kein Teilwort von  $T$  *)  
return „NEIN. Das Muster kommt im Text nicht vor.“
```

In den folgenden Abschnitten stellen wir bessere Verfahren vor, die auf einer „Vorabanalyse“ des Musters beruhen, in der gewisse Hilfsfunktionen für das Muster erstellt werden. Beide Algorithmen basieren auf dem naiven Verfahren; jedoch setzen sie die Hilfsfunktionen ein, um größere Verschiebungen des Musters zu ermöglichen.

7.1 Das Verfahren von Knuth, Morris und Pratt

Der Algorithmus von Knuth, Morris & Pratt (kurz KMP-Algorithmus genannt) basiert auf dem naiven Verfahren, wobei Muster und die Textteilwörter $T[r]$ von links nach rechts verglichen werden. Er benutzt eine Hilfsfunktion h für das Muster, die es ermöglicht beim Vergleich von $T[r]$ und M nicht ab Position 1 zu beginnen, sondern an einer Position $j + 1$, wobei (durch die Vergleiche von $T[1], T[2], \dots, T[r - 1]$ mit dem Muster) bereits feststeht, daß die ersten j Zeichen von $T[r]$ und M übereinstimmen.

m_1	\dots	m_j	m_{j+1}	\dots	m_j				
=		=	?						
t_1	\dots	t_r	\dots	t_{r+j-1}	t_{r+j}	\dots	t_{r+k-1}	\dots	t_n

Im Folgenden bezeichne i den „Textzeiger“ (d.h. die aktuelle Textposition ist $i = r + j \in \{1, \dots, n\}$).

Die Hilfsfunktion h . Die Hilfsfunktion h ordnet jeder Musterposition $j \in \{1, \dots, m\}$ den größten Index l zu, so daß $m_1 \dots m_l$ ein echtes Suffix von $m_1 \dots m_l \dots m_j$ ist, d.h. $m_r = m_{j-l+r}$, $r = 1, 2, \dots, l$.

$$\begin{array}{ccccccccc} m_1 & \dots & m_{j-l} & m_{j-l+1} & m_{j-l+2} & \dots & m_j \\ & & m_1 & m_2 & \dots & m_l \end{array}$$

Wir definieren $h : \{1, \dots, k\} \rightarrow \{0, 1, \dots, k - 1\}$ wie folgt. Für $j \in \{1, \dots, k\}$ sei

$$h(j) = \max \{l \in \{1, \dots, j - 1\} : m_1 \dots m_l \text{ ist Suffix von } m_1 \dots m_j\}$$

wobei $\max \emptyset = 0$ gesetzt wird. Insbesondere ist $h(1) = 0$. Zunächst ein Beispiel: Für das Muster $M = \text{aabaaab}$ sind die Werte der Hilfsfunktion h wie folgt:

i	1	2	3	4	5	6
$h(i)$	0	1	0	1	2	3

Einsatz der Hilfsfunktion h im KMP-Algorithmus. h wird im KMP-Algorithmus wie folgt eingesetzt. Die Musterpositionen $j \in \{1, \dots, k\}$ werden als Zustände eines Automaten (dem sogenannte *Skelett-Automaten*) angesehen (wobei der zusätzliche Zustand 0 verwendet wird). Nach der Bearbeitung des i -ten Textzeichens haben die Automatenzustände folgende Bedeutung. Der Automat ist genau dann in Zustand j , wenn $m_1 \dots m_j$ das längste Präfix des Musters ist, das Suffix des bereits gelesenen Textteils (also des Worts $t_1 t_2 \dots t_i$) ist. Insbesondere ist

$$m_1 \dots m_j = t_{i-j+1} \dots t_i.$$

Der Endzustand $j = k$ wird also genau dann erreicht, wenn $M = T[i-k+1] = t_{i-k+1} \dots t_i$, d.h. wenn ein Vorkommen des Musters gefunden ist. Der initiale Zustand ist $j = 0$.

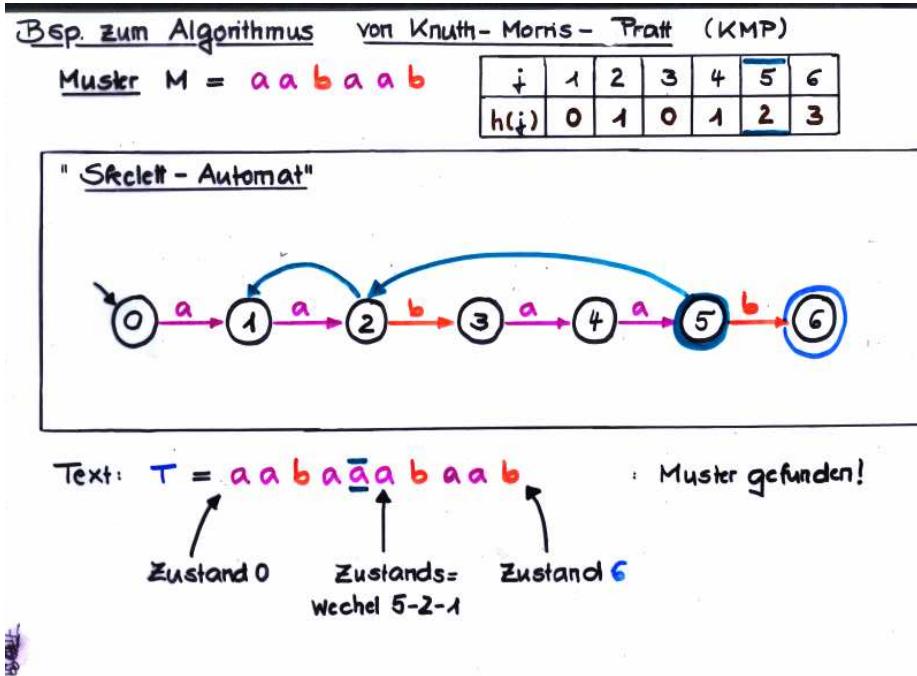
Die Folge der Wörter w_0, w_1, w_2, \dots mit

$$\begin{aligned} w_0 &= m_1 \dots m_{h(h(j))} \dots m_{h(j)} \dots m_j \\ w_1 &= m_1 \dots m_{h(h(j))} \dots m_{h(j)} \\ w_2 &= m_1 \dots m_{h(h(j))} \\ &\vdots \end{aligned}$$

durchläuft alle Suffixe von $m_1 \dots m_j$, die Präfixe des Musters sind. Sei j der aktuelle Zustand des Skelett-Automaten und sei i die aktuelle Textposition, wobei wir $i < n$ und $j < k$ annehmen. Das nächste Textzeichen t_{i+1} wird wie folgt bearbeitet. Sei

$$j_0 = j, \quad j_1 = h(j), \quad j_2 = h(j_1), \quad j_3 = h(j_2), \dots$$

Es werden sukzessive die Musterpositionen (Automatenzustände) $l = j_0, j_1, j_2, \dots$ betrachtet bis entweder $m_{l+1} = t_{i+1}$ oder $l = 0$. Da $j = j_0 > j_1 > j_2 > \dots$, entspricht dies einem schrittweisen „Zurückgehen“ im Automaten. Sobald $m_{l+1} = t_{i+1}$, dann wechselt der Skelett-Automat in den Zustand $l + 1$. In diesem Fall ist $m_1 \dots m_l m_{l+1}$ das längste Suffix von $t_1 \dots t_{i+1}$ mit $l \leq j$. Wird der Zustand $l = 0$ erreicht (d.h. $l = 0 = j_{\nu+1} = h(j_{\nu})$, wobei $j_0 > j_1 > \dots > j_{\nu} > 0$) und gilt $m_1 \neq t_{j+1}$, dann bleibt der Skelett-Automat im Zustand 0. Der Fall $l = 0$ und $m_1 = t_{j+1}$ wird wie zuvor beschrieben behandelt, d.h. der Skelett-Automat wechselt in den Zustand 1.



Wir formulieren den Algorithmus ohne explizite Benutzung des Automaten (siehe Algorithmus 94, Seite 419). Wir verwenden zwei ganzzahlige Variablen j und i , die als Automatenzustand (Musterposition) bzw. Textzeiger interpretiert werden können.

Algorithmus 94 KMP-Algorithmus zur Mustererkennung

Erstelle die Hilfsfunktion h für das Muster M . (* siehe Algorithmus 96 auf Seite 421 *)

$i := 0$; (* i ist die aktuelle Textposition *)
 $j := 0$; (* j ist der aktuelle Zustand (Musterposition) *)

WHILE $j < k$ und $i \leq n$ **DO**

(* berechne den maximalen Index $l \leq j$ mit $m_1 \dots m_{l+1} = t_{i-l+1} \dots t_{i+1}$ *)

WHILE $j > 0$ und $m_{j+1} \neq t_{i+1}$ **DO**

$j := h(j)$;

OD

(* Ist $m_{j+1} \neq t_{i+1}$, dann ist $j = 0$ der neue Zustand. *)

IF $m_{j+1} = t_{i+1}$ **THEN**

$j := j + 1$;

FI

$i := i + 1$;

(* nächstes Textzeichen *)

OD

IF $j = k$ **THEN**

gib „das Muster steht im Text ab Position $i - k + 1$ “ aus

ELSE

gib „das Muster kommt im Text nicht vor“ aus

FI

Anzahl an Zustandswechsel. Sei N_i die Anzahl an Ausführungen der Anweisung $j := h(j)$, die innerhalb der WHILE-Schleife für die Textposition $i + 1$ ausgeführt werden. In Automatenterminologie: N_i ist die Anzahl an Zustandswechsel, mit der der Skelett-Automat bei der Bearbeitung der $(i + 1)$ -ten Textposition mit h „zurückgeht“. Sei

$$N = \sum_i N_i$$

die Gesamtanzahl an Zustandswechsel mit der Hilfsfunktion h . Der Index i läuft über alle bearbeiteten Textpositionen (minus 1). Weiter sei z_i der Zustand, der nach der Bearbeitung des i -ten Textzeichens angenommen wird. Dann gilt

$$z_{i+1} \in \{ h^{N_i}(z_i), h^{N_i}(z_i) + 1 \},$$

wobei wir $z_0 = 0$ setzen. Insbesondere ist $z_{i+1} \leq z_i - N_i + 1$.

Die Gesamtanzahl der Zustandswechsel ist höchstens $N + n$, wobei der zweite Summand für die Anzahl an Textpositionen i mit $z_{i+1} = h^{N_i}(z_i) + 1$ steht; also die Anzahl an Textpositionen i , für die die Anweisung $j := j + 1$ in der WHILE-Schleife ausgeführt wird. Wir schätzen N ab. (Siehe auch Algorithmus 95, Seite 421.)

$$\begin{aligned} N &= \sum_i N_i = \sum_i (N_i + z_{i+1}) - \sum_i z_{i+1} - z_0 && (\text{beachte: } z_0 = 0) \\ &\leq \sum_i (N_i + z_{i+1}) - \sum_i z_i && (\text{Indexverschiebung}) \\ &= \sum_i (N_i + z_{i+1} - z_i) \leq \sum_i 1 \leq n. \end{aligned}$$

Dabei läuft der Index i jeweils über alle bearbeiteten Textpositionen (minus 1), also einer Teilmenge von $\{0, 1, \dots, n - 1\}$.

Berechnung der Hilfsfunktion h . Die Berechnung der Hilfsfunktion h beruht auf einem Schema, das ähnlich zur Mustersuche im Text verfährt. Die Werte $h(j)$, $j = 1, \dots, k$, werden sukzessive anhand folgender Formeln berechnet. Es gilt stets $h(1) = 0$. Sei $j \in \{1, \dots, n - 1\}$ und $l \in \{j, h(j), h^2(j), h^3(j), \dots\}$ der größte Index, so daß entweder $l = 0$ oder $m_{l+1} = m_{j+1}$. Dann ist

$$h(j + 1) = \begin{cases} l + 1 & : \text{falls } m_{l+1} = m_{j+1} \\ 0 & : \text{sonst (d.h. falls } l = 0\text{)}. \end{cases}$$

Siehe Algorithmus 96, Seite 421.

Die Anzahl an „Zustandswechsel“ ergibt sich wie oben. Sie ist beschränkt durch $2k$. Wir fassen die Ergebnisse zusammen:

Satz 7.1.1 (Kosten des KMP-Algorithmus). *Das Mustererkennungsproblem lässt sich mit dem KMP-Algorithmus in Zeit (und Platz) $\Theta(n + k)$ lösen. Dabei ist n die Textlänge und k die Musterlänge.*

Algorithmus 95 Zur Laufzeitanalyse des KMP-Algorithmus

Erstelle die Hilfsfunktion h für das Muster M .

$i := 0; \quad j := 0;$ (* Der initiale Zustand ist $z_0 = 0$. *)

(* Initialisiere den Zustandswechselzähler N durch $N := 0$. *)
WHILE $j < k$ und $i \leq n$ **DO**
 (* Der aktuelle Zustand ist $z_i = j$. Setze $N_i := 0$ *)

WHILE $j > 0$ und $m_{j+1} \neq t_{i+1}$ **DO**
 $j := h(j);$ (* setze $N_i := N_i + 1$ *)
OD

IF $m_{j+1} = t_{i+1}$ **THEN**
 $j := j + 1;$ (* wird höchstens n -mal ausgeführt *)
FI

(* aktualisiere den Zustandswechselzähler durch $N := N + N_i$ *)
 $i := i + 1;$
OD

: (* jetzt gilt $N \leq n$ *)

Algorithmus 96 Algorithmus zum Erstellen der KMP-Hilfsfunktion h

$h(1) := 0;$

FOR $j = 1, 2, \dots, k - 1$ **DO**

(* berechne den maximalen Index $l < j$ mit $m_1 \dots m_l m_{l+1} = m_{j-l+1} \dots m_j m_{j+1}$ *)
 $l := h(j);$

$x := m_{j+1};$

WHILE $l > 0$ und $x \neq m_{l+1}$ **DO**
 $l := h(l);$

OD

IF $x = m_{l+1}$ **THEN**

$h(j+1) := l + 1;$ (* **ELSE** $h(j+1) := 0$ *)

FI

OD

7.2 Das Verfahren von Boyer-Moore

Wir stellen nun einen weiteren Algorithmus zur Mustererkennung vor, der im schlimmsten Fall wie das naive Verfahren $\Theta(n \cdot k)$ Vergleiche benötigt. Für realistische Texte ist dieser von Boyer und Moore entwickelte Algorithmus jedoch meist sehr viel effizienter (auch als Verfahren mit linearer worst case Laufzeit). Er wird daher in vielen Textverarbeitungssystemen eingesetzt.

Die Grobidee des Verfahrens von Boyer-Moore (kurz BM-Verfahren genannt) ist eine Variante des naiven Verfahrens, bei der das Muster M mit den Teilworten $T[i]$ des Texts von rechts nach links verglichen wird.

t_1	\dots	t_i	\dots	t_s	t_{s+1}	\dots	t_{i+k-1}	t_{i+k}	\dots
?		=		=					
m_1	\dots	m_j	m_{j+1}	\dots	m_k				

Eine Verbesserung der Laufzeit des naiven Verfahrens wird dadurch versucht zu erzielen, in dem – mit Hilfe zweier Hilfsfunktionen δ_1 und δ_2 für das Muster – im Falle eines Mismatches das Muster möglichst weit nach rechts geschoben wird.

Algorithmus 97 Grundschema für das Boyer-Moore-Verfahren

Erstelle die Hilfsfunktionen für das Muster M .

$i := 1;$

(* i ist die aktuelle Textposition *)

WHILE $i \leq n - k + 1$ **DO**

Vergleiche M und $T[i]$ zeichenweise von rechts nach links;

IF $M = T[i]$ **THEN**

 return „das Muster steht im Text ab Position i “

ELSE

 berechne (unter Verwendung der Hilfsfunktionen) die Anzahl r an Positionen, um die das Muster verschoben werden kann;

 verschiebe das Muster nach rechts, d.h. setze $i := i + r$;

FI

OD

Gib „das Muster kommt im Text nicht vor“ aus.

Im Folgenden bezeichne j die aktuelle Musterposition. i ist eine Hilfsvariable, die für den Text benutzt wird. Der Wert $i \in \{1, 2, \dots, n - k + 1\}$ gibt Aufschluß über das Teilwort $T[i]$, für das der Vergleich mit M durchgeführt wird. Es gilt dann stets, daß $T[i]$ und M an den Positionen $j + 1, \dots, k$ übereinstimmen. Als nächstes werden die j -ten Zeichen von $T[i]$ und M (also die Zeichen t_{i+j-1} und m_j) verglichen. Die aktuelle Textposition

```

WHICH_FINALY_HALTS._AT_THAT.POINT
AT_THAT

WHICH_FINALY_HALTS._AT_THAT.POINT
AT_THAT

WHICH_FINALY_HALTS._AT_THAT.POINT
AT_THAT

WHICH_FINALY_HALTS._AT_THAT.POINT
AT_THAT

WHICH_FINALY_HALTS._AT_THAT.POINT
AT_THAT

WHICH_FINALY_HALTS._AT_THAT.POINT
AT_THAT

```

Abbildung 75: Beispiel zum Boyer-Moore-Algorithmus (nur Vorkommensheuristik)

ist also $s = i + j - 1$. Das Verschieben des Musters wird durch folgende Anweisungen durchgeführt:

- Zurücksetzen von j auf k (Ende des Musters).
- Erhöhen von i um einen Wert r , der sich aus δ_1 und δ_2 ergibt.

Die Vorkommenheuristik. Die Vorkommenheuristik δ_1 ist eine Funktion $\delta_1 : \Sigma \rightarrow \{0, 1, \dots, k\}$, deren Werte gegeben sind durch

$$\delta_1(x) = \max \{j : m_j = x\},$$

wobei $\max \emptyset = 0$. D.h. der Wert $\delta_1(x)$ gibt die rechteste Position j an, an der das Zeichen x im Muster vorkommt. Ist $\delta_1(x) = j > 0$, so gilt $m_j = x$ und $x \notin \{m_{j+1}, \dots, m_k\}$. Kommt x in M überhaupt nicht vor, so ist $\delta_1(x) = 0$.

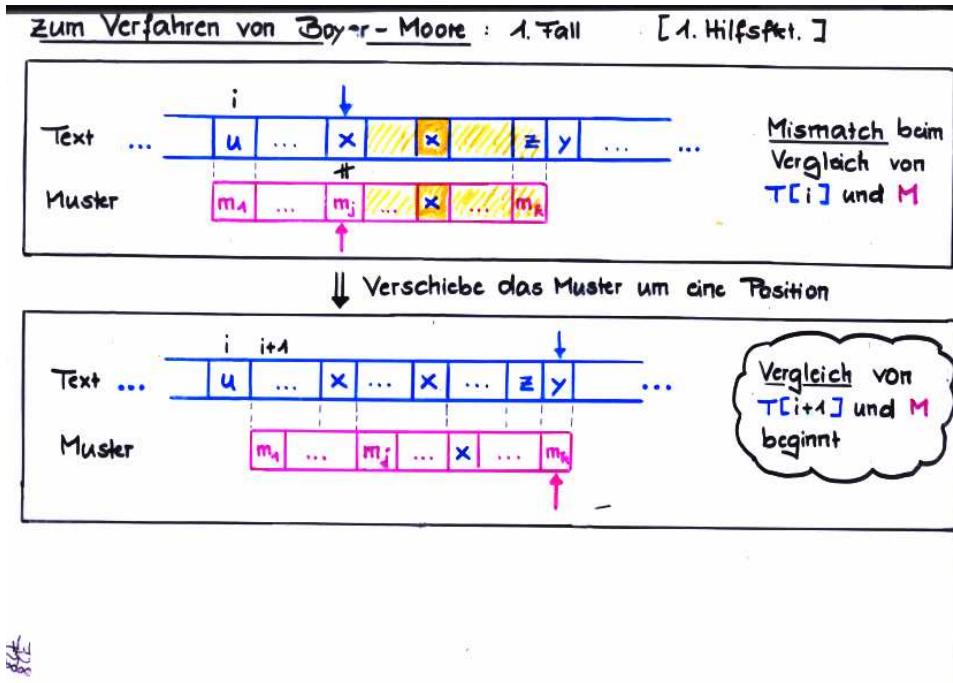
Beispiel 7.2.1 (Einsatz der Vorkommensheuristik). Wir machen uns den Einsatz der Funktion δ_1 an dem in Abbildung 75 gegebenem Beispiel klar. In diesem Beispiel hat der Text die Länge $n = 34$, das Muster die Länge $k = 7$. Mit dem Boyer-Moore-Verfahren wird das Muster nach nur 13 Vergleichen gefunden. \square

Die Funktion δ_1 wird im BM-Verfahren wie folgt eingesetzt. Sei j die erste (rechteste) Musterposition, für die ein Mismatch beim Vergleich von $T[i]$ und M stattfindet. Also $m_j \neq x = t_{i+j-1}$.

1. Fall: $x \in \{m_{j+1}, \dots, m_k\}$. Dann ist $\delta_1(x) > j$. Der Vergleich von $T[i+1]$ und M wird gestartet, d.h. das Muster wird um eine Position nach rechts verschoben.
2. Fall: $x = m_r$ und $x \notin \{m_{r+1}, \dots, m_k\}$, wobei $r = \delta_1(x) < j$. Dann wird der Vergleich von $T[i+j-r]$ und M gestartet, d.h. das Muster um $j-r$ Positionen nach rechts verschoben.
3. Fall: x kommt im Muster nicht vor. Dann ist $\delta_1(x) = 0$. Der Vergleich von $T[i+j]$ beginnt, d.h. das Muster wird um j Positionen nach rechts verschoben.

Der zweite und dritte Fall können zu „ $\delta_1(x) < j$ “ zusammengefaßt werden. Das Muster wird in beiden Fällen um $j - \delta_1(x)$ Positionen nach rechts verschoben.

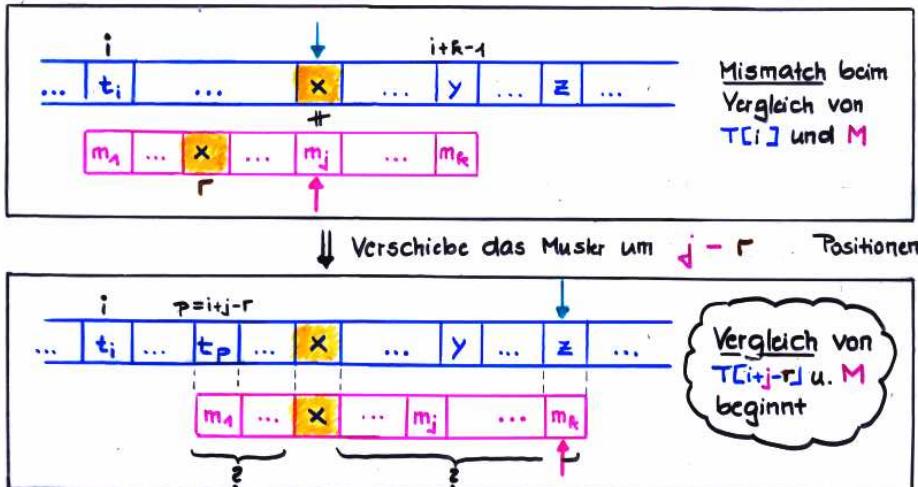
Ein Beispiel zum ersten Fall ist der Text $T = \text{bxaxaax} \dots$, in dem das Muster $M = \text{xaax}$ gesucht wird. Beim Vergleich von $T[1] = \text{bxax}$ mit dem Muster M tritt an der Musterposition $j = 2$ das erste Mismatch auf. Es gilt $\delta_1(\text{x}) = 4 > 2 = j$.



Das Schema für den zweiten Fall ist wie folgt:

zum Algorithmus von Boyer-Moore : 2. Fall

$x \notin \{m_{j+1}, \dots, m_k\}$



Entsprechend: Falls x in M nicht vorkommt, dann verschiebe das Muster um $i - r$ Positionen.

Zunächst formulieren wir das BM-Verfahren nur mit der Vorkommensheuristik δ_1 . Siehe Algorithmus 98, Seite 426.

Tatsächlich sind für viele Texte bereits mit der Vorkommensheuristik große Sprünge möglich; sie kann jedoch völlig versagen. Extrembeispiel: Für den Text $T = \text{aaa...a} = \text{a}^n$ und das Muster $M = \text{baa...a} = \text{ba}^{k-1}$ (mit $n > k > 1$) tritt beim Vergleich von $T[i] = \text{a}^i$ und M erst im $(k-1)$ -ten Schritt ein Mismatch auf ($\text{b} = m_1 \neq t_i = \text{a}$). Wegen $\delta_1(\text{a}) = k > 1$ findet lediglich eine Verschiebung des Musters um eine Position statt (1. Fall). Insgesamt werden also $\Theta(n \cdot k)$ Vergleiche durchgeführt.

Die Matchheuristik. Die Matchheuristik δ_2 versucht in dem oben genannten 1. Fall größere Verschiebungen des Musters zu ermöglichen. δ_2 setzt sich aus zwei Hilfsfunktionen Δ und $\bar{\Delta}$ zusammen.

Die Hilfsfunktion Δ : Tritt beim Vergleich von $T[i]$ und M ein Mismatch $m_j \neq t_s$ (mit $s = i + j - 1$) auf, so kann das Muster stets um $j - \Delta(j) + 1$ Positionen nach rechts verschoben werden, wobei

$$\Delta(j) = \max \{l \leq j : m_{j+1} \dots m_k = m_l \dots m_{l+k-j-1}\}$$

und $\max \emptyset = 0$. $\Delta(j)$ ist also der größte Index l , für den das bereits gelesene Teilwort $m_{j+1} \dots m_k = t_{i+j} \dots t_{i+k-1}$ des Texts ein wiederholtes Vorkommen im Muster hat. Insbesondere ist $l = 0$, wenn $m_{j+1} \dots m_k = t_{i+j} \dots t_{i+k-1}$ keine weiteren Vorkommen in M hat. Es kann mit dem Vergleich von $T[i + j - \Delta(j) + 1]$ und M begonnen werden. Dies entspricht einer Verschiebung des Musters um $j - \Delta(j) + 1$ Positionen.

Algorithmus 98 Das BM-Verfahren (einfache Version; nur mit der Vorkommensheuristik)

Berechne die Vorkommensheuristik δ_1 für das Muster M .

$i := 1; \quad j := k;$ (* Vergleiche $T[1]$ und M von rechts nach links *)

REPEAT

$x := t_{i+j-1};$ (* es gilt $m_{j+1} \dots m_k = t_{i+j} \dots t_{i+k-1}$ *)
(* x ist das aktuelle Textzeichen *)

IF $x = m_j$ **THEN**

$j := j - 1;$

ELSE

(* verschiebe das Muster mit Hilfe der Vorkommensheuristik δ_1 *)

IF $\delta_1(x) > j$ **THEN**

$i := i + 1$

(* 1. Fall *)

ELSE

$i := i + j - \delta_1(x)$

(* 2. oder 3. Fall *)

FI

$j := k;$

(* Vergleiche $T[i]$ und M von rechts nach links *)

FI

UNTIL $j = 0$ oder $i > n - k + 1$;

IF $j = 0$ **THEN**

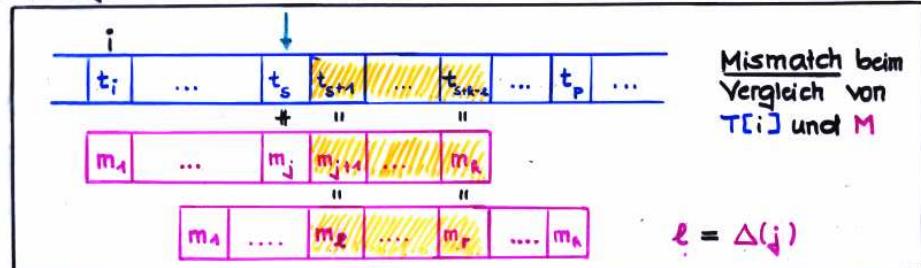
gib „das Muster steht im Text ab Position i “ aus.

ELSE

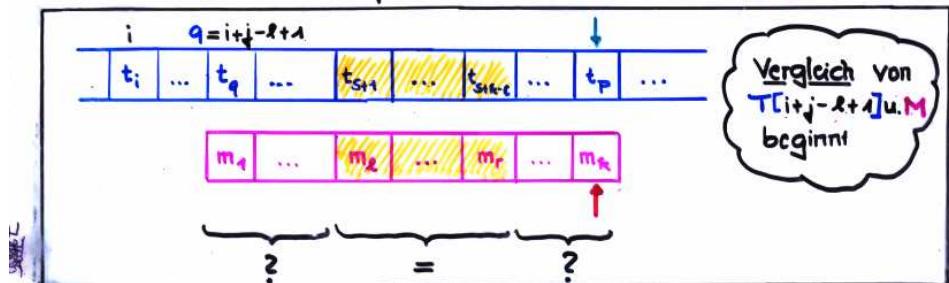
gib „das Muster kommt im Text nicht vor“ aus.

FI

zum Algorithmus von Boyer-Moore: die Match-Heuristik Δ



↓ Verschiebe das Muster um $j - \ell + 1$ Positionen



Beispiel 7.2.2 (Hilfsfunktion Δ). Wir betrachten ein Beispiel:

Text T ... o r a n g e _ a a n a n a s _ b a n a n a ...
Muster M b a n a n a

Offenbar ist $\delta_1(a) = 6$, da das Zeichen a in dem Suffix ana des Musters vorkommt. Die Vorkommensheuristik δ_1 liefert für das Mismatch $m_3 = n \neq a = t_s$ also lediglich eine Verschiebung des Musters um eine Position (1. Fall). Für das Mismatch $a \neq n$ an der Musterposition $j = 3$ liefert die Funktion Δ den Wert $\Delta(3) = 2$, da

$$m_{j+1} \dots m_k = m_4 m_5 m_6 = ana = m_2 m_3 m_4.$$

Also kann das Muster um $j - \Delta(j) + 1 = 3 - 2 + 1 = 2$ Positionen nach rechts verschoben werden.

Text T ... o r a n g e _ a a n a n a s _ b a n a n a ...
Muster M b a n a n a

Das nächste Mismatch $a \neq b$ tritt jetzt an der Musterposition $j = 1$ auf. Wieder liefert die Vorkommensheuristik nur eine Verschiebung um eine Position nach rechts (da $\delta_1(a) = 6 > j = 1$). Da das bereits gelesene Teilwort $anana$ kein weiteres Vorkommen im Muster hat, ist $\Delta(1) = 0$. Somit ist mit der Δ -Heuristik ein Sprung um

$$j - \Delta(j) + 1 = 1 - 0 + 1 = 2$$

Positionen möglich. □

Die Hilfsfunktion $\overline{\Delta}$: Eine weitere Verbesserung der Verschiebeheuristik sucht nach dem längsten echten Suffix $m_1 \dots m_l$ des bereits gelesenen Teilworts $m_{j+1} \dots m_k$ und verschiebt das Muster dann um $k - l$ Positionen. Diese soll die Matchheuristik in den Fällen verbessern, in denen $\Delta(j) = 0$ ist, d.h. in denen das bereits gelesene Teilwort $m_{j+1} \dots m_k = t_{i+j} \dots t_{i+k-1}$ keine weiteren Vorkommen im Muster hat.

Beispiel 7.2.3 (Hilfsfunktion $\overline{\Delta}$). Wir betrachten das Beispiel von zuvor.

Text T	... o r a n g e _ a a n a n a s _ b a n a n a ...
Muster M	b a n a n a

Beim Vergleich der Teilworts *aanana* des Texts mit dem Muster $M = banana$ gibt es kein nichtleeres Präfix $m_1 \dots m_l$, das zugleich Suffix von $m_{j+1} \dots m_k = anana$ ist. Also kann das Muster sogar um $k = 6$ Positionen verschoben werden.

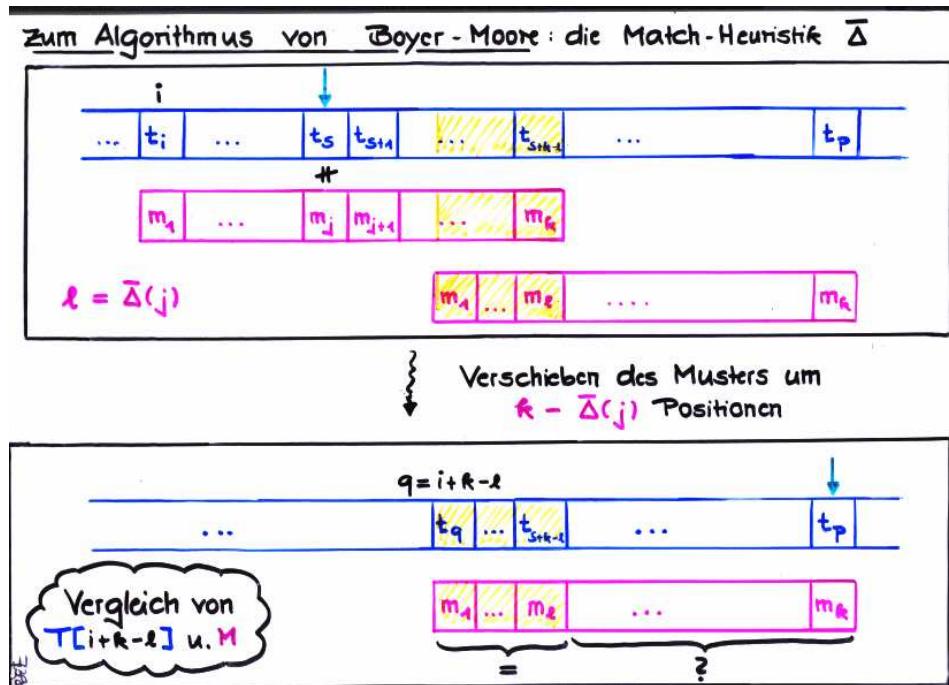
Text T	... o r a n g e _ a a n a n a s _ b a n a n a ...
Muster M	b a n a n a

□

Die Matchheuristik verwendet – neben Δ – auch die Funktion

$$\overline{\Delta}(j) = \max \{l \in \{0, 1, \dots, k\} : m_1 \dots m_l \text{ ist echtes Suffix von } m_{j+1} \dots m_k\},$$

die dann eingesetzt wird, wenn $\Delta(j) = 0$ ist. In diesem Fall kann das Muster um $k - \overline{\Delta}(j)$ Positionen nach rechts verschoben werden. Man beachte, daß für $\Delta(j) = 0$ stets $k - \overline{\Delta}(j) \geq j + 1$ ist.



Die Hilfsfunktion δ_2 (auch Matchheuristik genannt) ist gegeben durch

$$\delta_2(j) = \begin{cases} j - \Delta(j) + 1 & : \text{falls } \Delta(j) > 0 \\ k - \bar{\Delta}(j) & : \text{falls } \Delta(j) = 0. \end{cases}$$

Beispiel 7.2.4 (Matchheuristik). Wie zuvor betrachten wir das Extrembeispiel $T = \text{a}^n$, $M = \text{ba}^{k-1}$ (wobei $n > k > 1$). Dann ist

$$\Delta(1) = \bar{\Delta}(1) = 0$$

und somit $\delta_2(1) = k$. Die Matchheuristik liefert also nach dem ersten Mismatch beim Vergleich von $T[1]$ und M einen Sprung um k Positionen. Insgesamt werden bei der Suche von M in T lediglich die Teilwörter

$$T[1], T[k+1], T[2k+1], \dots, T[\rho k+1]$$

mit M verglichen (wobei $\rho = \lfloor n/k \rfloor$). □

Das Boyer-Moore-Verfahren setzt die Vorkommens- und Matchheuristik ein, um im Falle eines Mismatches einen möglichst großen Sprung zu machen. Die exakte Anzahl an Positionen, um die das Muster im Falle eines Mismatches nach rechts verschoben wird, ist

$$r = \max \{j - \delta_1(x), \delta_2(j)\},$$

wobei $x = t_s = t_{i+j-1}$ dasjenige Textzeichen ist, das das Mismatch ausgelöst hat. Die Bedeutung von i und j ist wie zuvor.

Berechnung der Hilfsfunktionen δ_1 und δ_2 . Die Vorkommensheuristik δ_1 ergibt sich mit der offensichtlichen Methode in Zeit $\Theta(k + |\Sigma|)$. (Die Laufzeit kann durch $\Theta(k)$ abgeschätzt werden, falls das Alphabet Σ als fest betrachtet wird.)

Die Hilfsfunktion δ_2 ergibt sich aus den Funktionen Δ und $\bar{\Delta}$, die jeweils mit einer Methode berechnet werden können, die auf das Verfahren zur Berechnung der Hilfsfunktion

SK | hier auch

h des KMP-Algorithmus (siehe Absatz 7.1) zurückgreift.

Berechnung von Δ : Sei $\bar{M} = m_k m_{k-1} \dots m_2 m_1$ und $\bar{h} : \{1, \dots, k\} \rightarrow \{0, 1, \dots, k-1\}$ die Hilfsfunktion für \bar{M} aus dem KMP-Algorithmus

SK | hier auch

(siehe Abschnitt 7.1). Dann gilt für $l \leq k$:

$$\bar{h}(k-l+1) = \min \{i \geq k-l : m_k m_{k-1} \dots m_{i+1} \text{ ist Suffix von } m_k m_{k-1} \dots m_l\}.$$

Lemma 7.2.5. Es gilt:

$$\{r \in \{1, \dots, k\} : \bar{h}(r) = k-j\} \neq \emptyset \text{ genau dann, wenn } \Delta(j) \geq 1.$$

Algorithmus 99 Das BM-Verfahren (mit der Vorkommens- und Matchheuristik)

Berechne die Hilfsfunktionen δ_1 und δ_2 für das Muster M .

$i := 1; \quad j := k;$ (* Vergleiche $T[1]$ und M von rechts nach links *)

REPEAT

$x := t_{i+j-1};$ (* x ist aktuelle Textzeichen *)

IF $x = m_j$ **THEN**

$j := j - 1$

ELSE

(* berechne die Anzahl r an Positionen, um die das Muster verschoben wird *)

$r := \max\{j - \delta_1(x), \delta - 2(j)\};$

(* Vergleiche $T[i + r]$ und M von rechts nach links *)

$j := k; \quad i := i + r;$

FI

UNTIL $j = 0$ oder $i > n - k + 1;$

IF $j = 0$ **THEN**

gib „das Muster steht im Text ab Position i “ aus.

ELSE

gib „das Muster kommt im Text nicht vor“ aus.

FI

Algorithmus 100 Berechnung der Vorkommensheuristik δ_1

FOR ALL $x \in \Sigma$ **DO**

$\delta_1(x) := 0;$

OD

FOR $j = 1, \dots, k$ **DO**

$\delta_1(m_j) := j;$

OD

In diesem Fall gilt:

$$\Delta(j) = k - \min\{r : \bar{h}(r) = k - j\} + 1.$$

Beispiel 7.2.6. Als Beispiel betrachten wir das Muster $M = \text{aabaaabaab}$. Für $j = 7$ ist $\Delta(j) = \Delta(7)$ der maximale Index $l \leq 6$, so daß aab ein Präfix von $m_l \dots m_k$ ist. Also ist $\Delta(7) = 5$. Wir überzeugen uns von der Aussage von Lemma 7.2.1. Offenbar ist $\bar{M} = \text{baabaaabaa}$. Die Werte der Hilfsfunktion \bar{h} sind:

r	1	2	3	4	5	6	7	8	9	10
$\bar{h}(r)$	0	0	0	1	2	3	0	1	2	3

Also ist $\{r : \bar{h}(r) = k - j\} = \{r : \bar{h}(r) = 3\} = \{6, 10\}$. Somit erhalten wir aus Lemma 7.2.1:

$$\Delta(7) = 10 - \min\{6, 10\} + 1 = 10 - 6 + 1 = 5.$$

□

Mit Lemma 7.2.5 erhalten wir folgende Berechnungsmöglichkeit für die Funktion Δ .

Algorithmus 101 Berechnung der Hilfsfunktion Δ

Berechne die KMP-Hilfsfunktion \bar{h} für \bar{M} .

FOR ALL $j = 1, \dots, k$ **DO**

$\Delta(x) := 0$;

OD

FOR $r = k, k-1, \dots, 1$ **DO**

$j := k - \bar{h}(r)$; $\Delta(j) := k - r + 1$;

OD

Nun zur Berechnung von $\bar{\Delta}$. Offenbar ist $\bar{\Delta}(j)$ der maximale Wert $l \in \{0, 1, \dots, k\}$, so daß $m_1 \dots m_l$ ein echtes Suffix von $M = m_1 \dots m_k$ und $l \leq k - j - 1$ ist. Die Menge aller Suffixe von $m_1 \dots m_k$, die von der Form $m_1 \dots m_l$ (d.h. Präfixe von M) sind, lassen sich durch die Hilfsfunktion h des KMP-Algorithmus für M berechnen. Hierzu betrachtet man die Folge

$$l_1 = h(k), l_2 = h(h(k)), l_3 = h(h(h(k))), \dots$$

Alle Werte j mit $l_i \leq k - j - 1 < l_{i-1}$ haben dann den Wert $\bar{\Delta}(j) = l_i$.

Die skizzierten Verfahren zur Berechnung von Δ und $\bar{\Delta}$ lassen sich in der Berechnung von δ_2 kombinieren, so daß sich die mehrfache Anwendung der Hilfsfunktion h in der Berechnung von $\bar{\Delta}$ einsparen läßt. Dies basiert auf folgendem Lemma.

Lemma 7.2.7. Ist $\Delta(j) = 0$, dann ist $\bar{\Delta}(j) = h(k)$.

Beispiel 7.2.8. Als Beispiel betrachten wir $M = \text{aabaaabaab}$. Offenbar ist

$$\Delta(6) = 0 \text{ und } \{r : \bar{h}(r) = 4\} = \emptyset.$$

$\overline{\Delta}(j)$ ist der maximale Index l , so daß $m_1 \dots m_l$ ein echtes Suffix von $m_{j+1} \dots m_k$ ist. Für $j = 6$ ist $m_{j+1} \dots m_k = \text{baab}$; also $\overline{\Delta}(6) = 3 = h(10)$. \square

Lemma 7.2.9. Für $j \geq k - h(k)$ gilt $\Delta(j) \geq 1$.

Die Aussagen von Lemma 7.2.5, 7.2.7 und 7.2.9 ergeben folgende Formel:

$$\delta_2(j) = \begin{cases} j - \Delta(j) + 1 & : \text{falls } j \geq k - h(k) \text{ oder } \delta(j) = 0 \\ k - h(k) & : \text{sonst} \end{cases}$$

die wir zur Formulierung des Algorithmus zur Berechnung von $\overline{\Delta}$ verwenden. Siehe Algorithmus 102.

Algorithmus 102 Berechnung der Matchheuristik δ_2

Berechne die KMP-Hilfsfunktion h für M .

Berechne die Hilfsfunktion Δ für M .

$r := h(k)$;

FOR ALL $j = k - r, \dots, k$ **DO**

$\delta_2(j) := j - \Delta(j) + 1$; (* für $j \geq k - h(k)$ ist $\Delta(j) \geq 1$ *)

OD

FOR $j = 1, \dots, k - r - 1$ **DO**

IF $\Delta(j) = 0$ **THEN**

$\delta_2(j) := k - r$

ELSE

$\delta_2(j) := j - \Delta(j) + 1$;

FI

OD

Die asymptotische Laufzeit zur Berechnung von δ_2 ist also durch die Kosten für die Berechnung der Hilfsfunktionen h und \overline{h} für die Muster M bzw. \overline{M} gegeben. h und \overline{h} können jeweils in Zeit

SK | hier auch

$\Theta(k)$ berechnet werden (siehe Abschnitt 7.1).

Zur Laufzeit des BM-Verfahrens. Die worst-case Laufzeit des BM-Verfahrens ist $\Theta(n \cdot k)$ (bzw. $\Theta(n \cdot k + |\Sigma| \cdot k)$, wenn das Alphabet als variabel angesehen wird). Dennoch ist die tatsächliche Laufzeit des BM-Verfahrens oftmals sehr viel besser. Die Situationen, in denen beide Heuristiken versagen, sind für Texte und Muster einer reellen Sprache (mit dem gewöhnlichen Alphabet aller Klein- und Großbuchstaben, Leerzeichen und Sonderzeichen wie Punkt, Komma, etc.) unrealistisch. In der Tat hat sich bereits die einfache BM-Verfahren, das nur mit der Hilfsfunktion δ_1 arbeitet, als sehr effizientes Verfahren für Mustererkennungsprobleme der Textverarbeitung herausgestellt.

Eine intuitive Erklärung für das schlechte worst-case Verhalten des BM-Algorithmus ergibt sich daraus, daß die Wörter $T[i]$ und M von rechts nach links verglichen werden. Beim Vergleich von $T[i+r]$ und M wird auf keinerlei Information zurückgegriffen, die sich aus dem Vergleich von $T[i]$ und M ergeben hat.

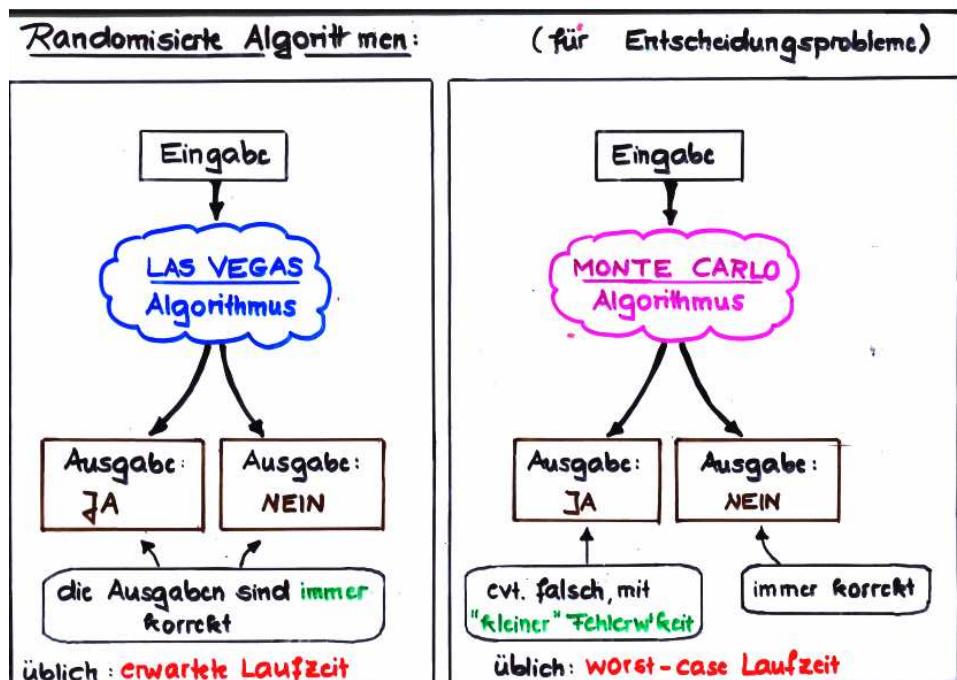
7.3 Der Algorithmus von Karp & Rabin

Der Algorithmus von Karp und Rabin zur Mustererkennung ist eine Alternative zu den beiden zuvor behandelten deterministischen Mustererkennungsalgorithmen und setzt das Konzept der Randomisierung ein, um durchschnittliche lineare Laufzeit zu erreichen. Der Karp-Rabin-Algorithmus ist konzeptionell sehr viel einfacher als die deterministischen Linearzeit-Algorithmen, da es keine Hilfsfunktionen für das Muster oder den Text benötigt. Leider betrifft dies nur die algorithmische Einfachheit; nicht die Analyse.

Einschub: Randomisierte Algorithmen. Bevor wir fortfahren ein paar allgemeine Bemerkungen zu randomisierten Algorithmen. Man unterscheidet zwei Arten von randomisierten Algorithmen.

- LAS VEGAS Algorithmen geben immer die korrekte Antwort (d.h. sind total korrekt im Sinne der in Abschnitt 1.1 gegebenen Definition).
- Für MONTE CARLO Algorithmen sind falsche Antworten möglich; jedoch wird vorausgesetzt, daß die Fehlerwahrscheinlichkeit „hinreichend klein“ ist.

Welche Fehlerwahrscheinlichkeit als genügend klein anzusehen ist, ist selbstverständlich Geschmackssache. Zunächst erscheint es irritierend, daß man einen Algorithmus als korrekt ansieht, selbst wenn er ein falsches Ergebnis liefern kann. Man mache sich jedoch klar, daß wir stets mit Wahrscheinlichkeit für das Auftreten eines Hardwarefehlers leben müssen und daß diese in vielen Fällen weit über der Fehlerwahrscheinlichkeit eines randomisierten Verfahrens liegt.



Während für MONTE CARLO Algorithmen die Effizienz meist an der Laufzeit im schlimmsten Fall gemessen wird, ist für LAS VEGAS Algorithmen die erwartete Laufzeit das übliche Kriterium für die „Güte“ der Ausführungszeit. Entsprechendes gilt für den Platzbedarf. Für Ja/Nein-Probleme unterscheidet man zwischen MONTE CARLO Algorithmen mit *einseitigem Fehler* (bei denen die Antwort „Nein“ immer korrekt ist, während die Antwort „Ja“ falsch sein kann, oder umgekehrt) und MONTE CARLO Algorithmen mit *beidseitigem Fehler* (bei denen sowohl die Antwort „Ja“ als auch die Antwort „Nein“ falsch sein kann). (Auf der Folie ist das Schema für MONTE CARLO Algorithmen mit einseitigem Fehler skizziert.) Liegt ein MONTE CARLO Algorithmus mit einseitigem Fehler und Fehlerwahrscheinlichkeit < 1 vor, dessen Fehlerwahrscheinlichkeit als nicht annehmbar groß angesehen wird, dann kann man durch mehrfaches Ausführen des Algorithmus eine beliebig kleine Fehlerwahrscheinlichkeit garantieren.

Der Karp-Rabin Algorithmus. Die Problemstellung ist wie zuvor. Gegeben ist ein Text $T = t_1 \dots t_n$ und ein Muster $M = m_1 \dots m_k$, wobei $n > k$ und $t_1, \dots, t_n, m_1, \dots, m_k$ Zeichen eines Alphabets Σ sind. Gefragt ist, ob $T[i] = M$ für einen Index i , wobei $T[i] = t_i t_{i+1} \dots t_{i+k-1}$. Zur Vereinfachung nehmen wir an, daß das Alphabet $\Sigma = \{0, 1, \dots, 9\}$ zugrundeliegt. Dies ermöglicht es, die Wörter $T[i]$ und M als Dezimalzahlen (mit jeweils k Dezimalstellen und eventuellen führenden Nullen) aufzufassen:

$$T[i] = \sum_{j=0}^{k-1} 10^{k-1-j} \cdot t_{i+j}, \quad M = \sum_{j=0}^{k-1} 10^{k-1-j} \cdot m_{j+1}.$$

Liegt ein anderes Alphabet Σ der Kardinalität d vor, dann kann man die Elemente von Σ mit den Zahlen $0, 1, \dots, d - 1$ durchnumerieren und mit dem Zahlensystem zur Basis d (anstelle des Dezimalsystems) arbeiten. Der Algorithmus von Karp & Rabin basiert auf der sogenannten *Fingerprint-Technik*. Grob gesprochen werden lediglich zufällige „Fingerabdrücke“ des Musters und der Textfragmente $T[i]$ verglichen. Die „Fingerabdrücke“ sind im Wesentlichen Hashwerte, die für das Muster und Textfragmente erstellt und anschliessend auf Gleichheit verglichen werden.

Wir stellen zuerst eine MONTE CARLO Version des Karp-Rabin Algorithmus vor, die wir dann zu einem LAS VEGAS Verfahren modifizieren.

Die MONTE CARLO Version des Karp-Rabin Algorithmus ist eine einfache Variation des naiven Mustererkennungsalgorithmus, der $T[i]$ und M für alle Textpositionen $i \in \{1, \dots, n - k + 1\}$ vergleicht. Anstelle $T[i]$ und M zeichenweise zu vergleichen, verwenden wir eine zufällig gewählte Primzahl p und vergleichen die Zahlen $F_p(T[i])$ und $F_p(M)$ für die zugehörige Fingerprint-Funktion

$$F_p : \mathbb{N} \rightarrow \{0, 1, \dots, p - 1\}, \quad F_p(x) = x \bmod p.$$

Ist z.B. $T = 34217$ und $M = 511$ und wird die Zahl $p = 3$ gewählt, dann ist

$$F_p(M) = 511 \bmod 3 = 1 = 421 \bmod p = F_p(T[2]).$$

Der Algorithmus würde also fälschlicherweise die Antwort „Ja“ liefern. Wird jedoch $p = 17$ gewählt, dann ist

$$F_p(M) = 1 \text{ und } F_p(T[1]) = 2, F_p(T[2]) = F_p(T[3]) = 13.$$

Also liefert der Algorithmus die korrekte Antwort „Nein“. Die Kosten für den Zah-

Algorithmus 103 Mustererkennung: das Verfahren von Karp & Rabin (Grobschema)

wähle zufällig eine Primzahl p ;
 $i := 1$;

REPEAT

(* Vergleiche die Fingerprints von $T[i]$ und M *)

IF $F_p(T[i]) = F_p(M)$ **THEN**
 return „Ja. M kommt (vermutlich) in T vor.“
ELSE
 $i := i + 1$;
FI

UNTIL $i > n - k + 1$;

Return „Nein. M ist nicht in T enthalten.“.

lenvergleich können als konstant angenommen werden, falls p „hinreichend klein“ ist, d.h. mindestens $\lceil \log p \rceil$ Bits zur Darstellung natürlicher Zahlen zur Verfügung stehen (uniformes Kostenmaß).

Berechnung der Fingerprints. Wir gehen davon aus, daß $T[i]$ und M zunächst nur als Ziffernfolgen vorliegen und erläutern, wie die Werte $F_p(T[i])$ und $F_p(M)$ berechnet werden können. Die zu $T[i]$ und M gehörenden Dezimalzahlen ergeben sich durch Aufsummieren in $\Theta(k)$ Schritten. Diese berechnen wir für M und $T[1]$. Damit ergeben sich auch die Zahlen $F_p(M)$ und $F_p(T[1])$ in Zeit $\Theta(k)$. Die Berechnung von $F_p(T[i+1])$ kann mit Hilfe der Zahl $F_p(T[i])$ in konstanter Zeit durchgeführt werden. Hierzu verwenden wir folgende Beobachtung:

$$\begin{aligned} T[i+1] &= (t_{i+1} \dots t_{i+k-1} t_{i+k})_{10} \\ &= 10 \cdot (t_{i+1} \dots t_{i+k-1})_{10} + t_{i+k} \\ &= 10 \cdot (T[i] - t_i \cdot 10^{k-1}) + t_{i+k}. \end{aligned}$$

Die Schreibweise $(\dots)_{10}$ soll andeuten, daß wir die Ziffernfolge \dots als Dezimalzahl auffassen. Wir benutzen die Rechengesetze

$$\begin{aligned} F_p(x+y) &= (F_p(x) + F_p(y)) \bmod p, \\ F_p(xy) &= (F_p(x) \cdot F_p(y)) \bmod p \end{aligned}$$

und erhalten

$$F_p(T[i+1]) = (10 \cdot (F_p(T[i]) - t_i \cdot F_p(10^{k-1})) + t_{i+k}) \bmod p.$$

Beispielsweise ist für $T = 34217$ und $k = 3$:

$$T[2] = 421 \text{ und } T[3] = 217 = 10 \cdot (421 - 4 \cdot 10^2) + 7.$$

Algorithmus 104 MONTE CARLO Algorithmus zur Mustererkennung

wähle zufällig eine Primzahl p ;

$$Q := 10^{k-1} \bmod p;$$

$$x := M \bmod p;$$

(* x ist Fingerprint von M *)

$$y := T[1] \bmod p;$$

(* y ist Fingerprint von $T[1]$ *)

$$i := 1;$$

WHILE $i \leq n - k + 1$ **DO**

(* Vergleiche die Fingerprints von $T[i]$ und M *)

IF $x = y$ **THEN**

return „Ja. M kommt (vermutlich) im Text T vor.“

FI

$$i := i + 1;$$

(* berechne den Fingerprint $F_p(T[i])$ *)

IF $i \leq n - k + 1$ **THEN**

$$y := (10 * (y - t_{i-1} * Q) + t_{i+k-1}) \bmod p;$$

FI

OD

Return „Nein. M ist nicht in T enthalten“.

Insgesamt ergibt sich die Laufzeit $\mathcal{O}(n + k)$ für die skizzierte Vorgehensweise. Dabei wird – wie zuvor erläutert – vorausgesetzt, daß der Zahlenvergleich von $F_p(T[i])$ und $F_p(M)$ sowie das Rechnen mit k -stelligen Dezimalzahlen nur konstante Kosten verursacht.

Fehlerwahrscheinlichkeit der MONTE CARLO Version. Wir schätzen nun die Wahrscheinlichkeit ab, daß das Verfahren fälschlicherweise die Antwort „Ja“ gibt. Seien $x, y \in \mathbb{N}$, so daß $x \neq y$ und $x, y < 10^k$. (Zur Erinnerung: $k = |M|$ ist die Länge des Musters.) Dann gilt:

$$\begin{aligned} F_p(x) &= F_p(y) \\ \text{gdw } |x - y| &\equiv 0 \bmod p \\ \text{gdw } p &\text{ ist ein Primfaktor von } |x - y|. \end{aligned}$$

Sei r die Anzahl an Bits, die zur Darstellung von p (im Binärsystem) zur Verfügung stehen. D.h. die Primzahl p wird zufällig aus dem Zahlenbereich $\{2, 3, \dots, 2^r - 1\}$ gewählt. Weiter sei $R = 2^r - 1$ und $\pi(R)$ die Anzahl an Primzahlen, die $\leq R$ sind. Somit gilt folgende Formel:

$$\begin{aligned} & \text{Prob} \left[F_p(x) = F_p(y) \mid p \text{ ist zufällige Primzahl } \leq R \right] \\ &= \frac{\text{Anzahl der Primfaktoren von } |x - y|}{\pi(R)}. \end{aligned}$$

Aus der Zahlentheorie ist bekannt, daß es „sehr viele“ Primzahlen gibt. Wir zitieren das fundamentale Ergebnis ohne Beweis.

Satz 7.3.1 (Primzahlsatz).

$$\lim_{R \rightarrow \infty} \frac{\pi(R)}{R/\ln R} = 1.$$

Dabei steht $\ln R$ für den natürlichen Logarithmus, der als Basis die Eulersche Zahl $e \approx 2,7\dots$ hat. Insbesondere ist

$$\pi(R) = \Theta\left(\frac{R}{\ln R}\right) = \Theta\left(\frac{R}{\log R}\right).$$

(ohne Beweis)

Weiter gilt $|x - y| < 10^k = 2^{k \cdot \log(10)}$. Daher ist die Anzahl der Primfaktoren von $|x - y|$ beschränkt durch $k \cdot \log 10$. (Beachte: Jede Primzahl ist ≥ 2 .) Wir erhalten folgende Abschätzung:

$$\begin{aligned} & \text{Prob} \left[F_p(x) = F_p(y) \mid p \text{ ist zufällige Primzahl } \leq R \right] \\ &< \frac{k \cdot \log(10)}{\pi(R)} = \Theta\left(\frac{k \cdot \log R}{R}\right) \end{aligned}$$

Verbesserung der Fehlerwahrscheinlichkeit. Die Fehlerwahrscheinlichkeit kann durch mehrfaches Anwenden des Verfahrens verbessert werden. Wir nehmen an, daß die gewünschte Fehlerwahrscheinlichkeit höchstens ε ist (für eine „kleine“ positive reelle Zahl ε). Weiter nehmen wir an, daß R (bzw. r) groß genug gewählt war, so daß $(k \log R)/R < 1$ ist. Wir wählen nun eine ganze Zahl $L \geq 1$, so daß

$$\left(\frac{k \cdot \log R}{R}\right)^L < \varepsilon$$

und führen das Verfahren mit L zufällig gewählten Primzahlen aus dem Bereich $\{2, 3, \dots, R\}$ durch. Da es sich hier um einen MONTE CARLO Algorithmus mit einseitigem Fehler handelt, genügt es selbstverständlich, das Verfahren nur für die Antwort „Ja“ wiederholt auszuführen.

Diese Vorgehensweise ist in Algorithmus 117 skizziert. Die Fehlerwahrscheinlichkeit ist dann $< \varepsilon$. Die worst-case Laufzeit ist $\mathcal{O}(L \cdot (n + k))$.

Algorithmus 105 MONTE CARLO Algorithmus mit verbesserter Fehlerw'keit

$l := 1;$

(*) wähle zufällig eine Primzahl p ;

$Q := 10^{k-1} \bmod p$;
 $x := M \bmod p$;
 $y := T[1] \bmod p$;
 $i := 1$;

WHILE $i \leq n - k + 1$ **DO**

(* Vergleiche die Fingerprints von $T[i]$ und M *)

IF $x = y$ **THEN**

IF $l = L$ **THEN**

return „Ja. M kommt (vermutlich) im Text T vor.“

ELSE

$l := l + 1$; goto (*);

(* wiederhole das Verfahren *)

FI

FI

$i := i + 1$;

IF $i \leq n - k + 1$ **THEN**

$y := (10 * (y - t_{i-1} * Q) + t_{i+k-1}) \bmod p$;

FI

OD

Return „Nein. M kommt nicht in T vor.“

Die LAS VEGAS Version des Karp-Rabin Verfahrens. Wir modifizieren nun das MONTE CARLO Verfahren (mit nur einer zufällig gewählten Primzahl, also $L = 1$), so daß ein fehlerfreier LAS VEGAS Algorithmus entsteht (siehe Algorithmus 118). Dazu verändern wir den Algorithmus einfach dahingehend, daß – sobald $F_p(T[i]) = F_p(M)$ – der zeichenweise Vergleich von $T[i]$ und M durchgeführt wird. Die erwartete Laufzeit der

Algorithmus 106 LAS VEGAS Algorithmus zur Mustererkennung

wähle zufällig eine Primzahl p ;

$$Q := 10^{k-1} \bmod p; \quad x := M \bmod p; \quad y := T[1] \bmod p; \quad i := 1;$$

WHILE $i \leq n - k + 1$ **DO**

(* Vergleiche die Fingerprints von $T[i]$ und M *)

IF $x = y$ **THEN**

(* Vergleiche M und $T[i]$ zeichenweise *)

IF $M = T[i]$ **THEN**

return „Ja. M kommt in T vor.“

FI

FI

$$i := i + 1;$$

IF $i \leq n - k + 1$ **THEN**

$$y := (10 * (y - t_{i-1} * Q) + t_{i+k-1}) \bmod p;$$

FI

OD

Return „Nein. M kommt nicht in T vor“.

LAS VEGAS Methode ist durch die Kostenfunktion

$$T(n, k) = \mathcal{O}(n + k) + \mathcal{O}\left(k \cdot \frac{nk \log R}{R}\right)$$

gegeben. Diese erklärt sich wie folgt. Der zeichenweise Vergleich von $T[i]$ und M erfordert $\mathcal{O}(k)$ Zeiteinheiten. Der erste Summand $\mathcal{O}(n + k)$ steht für sämtliche Schritte, die bereits bei der MONTE CARLO Version auszuführen sind und dem höchstens einmal ausgeführten erfolgreichen Vergleich von $T[i]$ und M . Die erwartete Anzahl an Textpositionen i , für die $T[i]$ und M zeichenweise ohne Erfolg verglichen werden, ist $\mathcal{O}(n(k \log R)/R)$. Diese ergibt sich aus den obigen Überlegungen zur Fehlerwahrscheinlichkeit der MONTE CARLO Version. Für feste Textposition i mit $T[i] \neq M$ ist die Wahrscheinlichkeit für $F_p(T[i]) = F_p(M)$ beschränkt durch $\mathcal{O}((k \log R)/R)$ (siehe oben). Aufsummieren über alle Textpositionen i mit $T[i] \neq M$ ergibt die obere Schranke $\mathcal{O}(n(k \log R)/R)$.

Wir fassen die Ergebnisse in dem folgenden Satz zusammen, wobei wir die Größenordnung von R so wählen, daß sich lineare Laufzeit ergibt.

Satz 7.3.2 (Kosten des Karp-Rabin Algorithmus). Für $R = \Theta(n^2 k \log(n^2 k))$ gilt:

- Die MONTE CARLO Version (Algorithmus 116) hat die Fehlerwahrscheinlichkeit $\mathcal{O}(1/n)$ und lineare worst-case Laufzeit $\mathcal{O}(n + k)$.
- Die LAS VEGAS Version hat die worst-case Laufzeit $\mathcal{O}(n \cdot k)$ und die erwartete Laufzeit $\mathcal{O}(n + k)$.

Dabei vernachlässigen wir die Kosten (und Fehlerwahrscheinlichkeit) für das Generieren zufälliger Primzahlen. Obige Kostenanalyse setzt voraus, daß der Vergleich von Zahlen $\leq R$, also Zahlen mit $\mathcal{O}(\log n + \log k)$ Binärstellen, als elementare Operation, welche in konstanter Zeit ausgeführt werden kann, angesehen wird. Für enorm großes n und k ist dies nicht realistisch. In diesem Fall sind zusätzliche Kosten für den Zahlenvergleich zu berücksichtigen.

Es bleibt zu klären, wie zufällige Primzahlen generiert werden können.

Generierung zufälliger Primzahlen. Viele randomisierte Verfahren (z.B. der soeben behandelte Mustererkennungsalgorithmus, oder diverse Verfahren, die in der Kryptographie eingesetzt werden) benutzen einen Generator zum Erzeugen zufälliger Primzahlen. Das Grundschema eines solchen ZufallsPrimzahlgenerators ist wie folgt. Man legt eine Zahl $r \in \mathbb{N}$ fest, so daß eine r -stellige Zahl $p = \langle x_0, x_1, \dots, x_{r-1} \rangle$ (Binärzahldarstellung mit eventuellen führenden Nullen) ermittelt wird, deren Ziffern x_i zufällig gewählt werden (Münzwurf). Es wird geprüft, ob p eine Primzahl ist. Wenn nein, dann wird das Verfahren wiederholt. Siehe Algorithmus 119. Aus dem Primzahlsatz (Satz 7.3.1 auf Seite 437) ergibt sich, daß im Mittel nach nur $\Theta(r)$ Versuchen eine Primzahl gefunden wird. Beachte, dass aus dem Primzahlsatz folgt, daß mit einer Wahrscheinlichkeit von ca. $1/\ln R$ die zufällig gewählte Zahl $x \in \{1, 2, \dots, R\}$ eine Primzahl ist. Daher wird im Mittel spätestens nach ca. $\ln R = \Theta(r)$ Iterationen eine Primzahl gefunden.

Algorithmus 107 Generieren zufälliger Primzahlen mit maximal r Bits

REPEAT

wähle zufällig r Bits x_0, x_1, \dots, x_{r-1} ; (* werfe r -mal eine faire Münze *)

$$x := x_0 + 2 * x_1 + 4 * x_2 + \dots + 2^{r-2} * x_{r-2} + 2^{r-1} * x_{r-1};$$

Prüfe, ob x eine Primzahl ist; (* Primzahltest *)

UNTIL x ist eine Primzahl.

Gib x aus.

8 Geometrische Algorithmen

Geometrische Algorithmen befassen sich mit klassischen geometrischen Fragestellungen wie „Gegeben ist eine Menge P von Punkten im \mathbb{R}^3 . Bestimme zu jedem Punkt $p \in P$ den nächsten Nachbarn in P “ oder „Berechne die Schnittmenge zweier Polygone“ oder „Berechne die konvexe Hülle einer endlichen Punktmenge $P \subseteq \mathbb{R}^d$ “. Derartige Problemstellungen spielen eine wesentliche Rolle in der Computer-Graphik, beim VLSI-Design oder in der Robotik. Sie können aber auch in vielen anderen Gebieten Anwendung finden. Beispielsweise verwendet ein amerikanischer Jeanshersteller geometrische Algorithmen, um den Abfall zu minimieren, der beim Zuschnitt von Einzelteilen aus den Stoffbahnen entsteht. Eine mögliche Instanz des Problems der konvexen Hülle ist eine Anfrage an eine Datenbank, in der Datensätze mit diversen Attributen (z.B. Name, Adresse, Geburtsdatum, Gehalt, etc. für eine Personaldatei) gespeichert sind und in der sämtliche Datensätze, für die die zugehörigen Attributwerte innerhalb gewissen Grenzen liegen, gesucht sind. Z.B. ergibt sich aus den Bedingungen „ $40 \leq \text{Alter} \leq 50$ “ und „ $5\,000 \leq \text{Gehalt} \leq 10\,000$ “ ein Rechteck. Komplexere Bedingungen können zu anderen konvexen Gebieten führen.

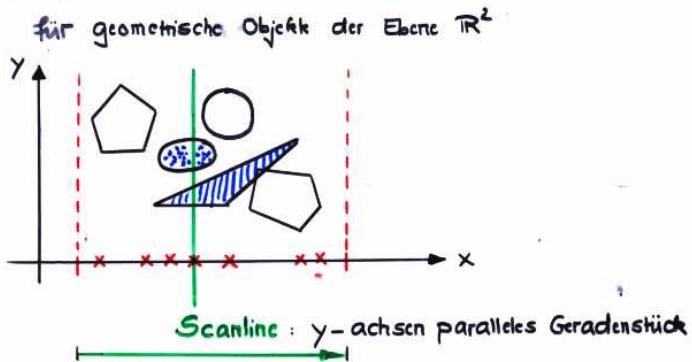
Wir stellen hier zwei grundlegende Prinzipien (das sogenannte Scanline-Prinzip und geometrisches DIVIDE & CONQUER) vor und erläutern diese an sehr einfachen Beispielen, wobei wir uns auf geometrische Probleme in der Ebene beschränken.

8.1 Scanline-Algorithmen

Scanline-Algorithmen (häufig auch Plane-Sweep-Algorithmen genannt) basieren auf einer Art „Sortierung“ der zu untersuchenden geometrischen Objekte. Diese Sortierung ergibt sich durch die Reihenfolge, in der die Objekte von einer sogenannten *Scanline*, die z.B. von links nach rechts über die Ebene wandert, geschnitten werden.⁷⁰ Zur Verwaltung der jeweils *aktiven Objekte* (Objekte, die von der Scanline getroffen werden) wird oftmals eine sortierte Liste oder ein Suchbaum verwendet. Die Dynamik der Scanline wird algorithmisch durch eine Auswahl von *Haltepunkten* modelliert, die für die Momente stehen, an denen aktive Objekte stillgelegt oder nichtaktivierte Objekte aktiviert werden.

⁷⁰In den hier vorgestellten Algorithmen ist die Scanline eine zur y -Achse parallele Gerade, die von links nach rechts wandert. Selbstverständlich kann man auch x -achsenparallele Geraden oder Geraden mit anderer Steigung verwenden. In einigen Fällen ist auch ein um einen Punkt rotierender Strahl als Scanline sinnvoll.

Das Scanline-Prinzip:



Grundidee:

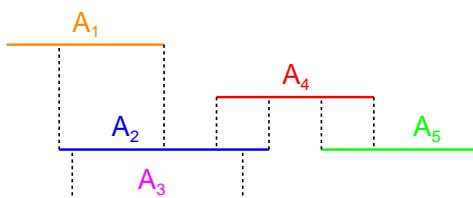
Analyse die **aktiven Objekte** (d.h. Objekte, die von der Scanline getroffen werden)

wobei die Scanline nur an ausgewählten **Haltpunkten** betrachtet wird.

8.1.1 Das Sichtbarkeitsproblem

Sichtbarkeitsprobleme befassen sich mit der Frage, welche Objekte von anderen Objekten sichtbar sind. Ein typisches Sichtbarkeitsproblem der Computer-Grafik stellt sich bei der Erstellung einer wirklichkeitsgetreuen Ansicht eines Raums. Diese muß berücksichtigen, welche Objekte vom Standort sichtbar sind und welche Objekte durch andere verdeckt sind. Eine andere klassische Instanz von Sichtbarkeitsproblemen tritt beim Chip-Entwurf auf, bei dem einander sichtbare Schaltelemente möglichst kompakt (aber unter Einhaltung gewisser Abstandsbedingungen) plaziert werden müssen.

Wir beschränken uns hier auf eine sehr einfache Variante von Sichtbarkeitsproblemen. Gegeben seien n horizontale Liniensegmente $A_i = [a_i, b_i] \times \{y_i\} = \{(x, y_i) : a_i \leq x \leq b_i\}$, $i = 1, \dots, n$. Wir nehmen zur Vereinfachung an, daß die Werte $a_1, \dots, a_n, b_1, \dots, b_n$ p.v. sind. Gesucht sind alle Segmentpaare (A_i, A_j) , die sich gegenseitig sehen können, d.h. alle Paare (A_i, A_j) mit $i \neq j$ und für die es Punkte $s = (x, y_i) \in A_i$, $s' = (x, y_j) \in A_j$ gibt, so daß die Verbindungsstrecke zwischen s und s' kein anderes Segment schneidet.



In dem Beispiel oben sind (A_1, A_2) , (A_2, A_3) , (A_2, A_4) und (A_4, A_5) Sichtbarkeitspaare. Im Folgenden behandeln wir die Paare (A_i, A_j) als ungeordnete Paare, d.h. wir identifizieren (A_i, A_j) mit (A_j, A_i) . Der Aufwand des naiven Verfahrens, das für alle Paare

(A_i, A_j) , prüft, ob A_i von A_j sichtbar ist, hat die Laufzeit $\Omega(n^2)$. Wir stellen jetzt ein besseres auf dem Scanline-Prinzip beruhendes Verfahren mit der Laufzeit $\mathcal{O}(n \log n)$ vor. Die Scanline ist eine y -achsenparallele Gerade mit den Haltepunkten

$$Q = \{a_1, b_1, \dots, a_n, b_n\},$$

die in einer aufsteigend sortierten Liste verwaltet werden. Die jeweils aktiven Liniensegmente A_i werden durch einen

AVL-Blattsuchbaum L dargestellt (vgl. Abschnitt 4.2.1; dabei wird die y -Koordinate als Schlüsselwert für A_i angesehen, d.h. $\text{key}(A_i) = y_i$). Die Segmente A_i sind also durch die Blätter in L repräsentiert.

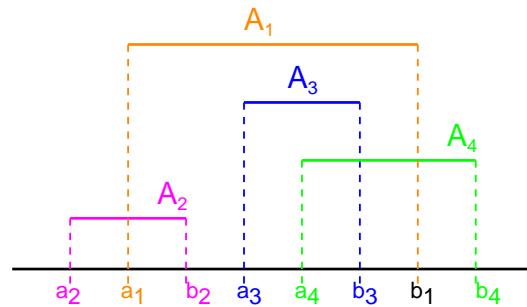
Wir nennen A_j den *unteren Nachbarn* von A_i in L , falls gilt:

- A_j und A_i sind in L repräsentiert
- $y_j < y_i$
- es gibt kein in L repräsentiertes Segment A_k mit $y_j < y_k < y_i$.

In diesem Fall wird A_i der *obere Nachbar* von A_j in L genannt. Für jeden Haltepunkt $q \in Q$ wird ein Segment A_i in L eingefügt oder aus L gelöscht.

- Wird das Segment A_i am Haltepunkt $q = a_i$ in L eingefügt und hat A_i in L einen oberen Nachbarn A^+ , dann sind A_i und A^+ voneinander sichtbar. Entsprechendes gilt für den unteren Nachbarn A^- von A_i (falls existent).
- Wird das Segment A_i am Haltepunkt $q = b_i$ aus L gelöscht und hat A_i einen oberen und einen unteren Nachbarn A^+ bzw. A^- in L unmittelbar vor dem Löschen, dann sind A^+ und A^- (an einem Punkte $y > b_i$) voneinander sichtbar.

Zunächst ein Beispiel:



Sortierung der aktiven Segmente (nach ihren y -Koordinaten)	A_2	A_1 A_2	A_1	A_1 A_3	A_1 A_3 A_4	A_1 A_4	A_4	
Ausgabe der Sichtbarkeitspaare		(A_1, A_2)		(A_1, A_3)	(A_3, A_4)	(A_1, A_4)		

Algorithmus 108 Scanline-Algorithmus zum Bestimmen der Sichtbarkeitspaare

$Q :=$ sortierte Liste der Haltepunkte a_i, b_i der Scanline;

(* Die aktiven Liniensegmente werden in einem AVL-Blattsuchbaum verwaltet *)
(* (sortiert nach den y -Koordinaten). *)
 $L := \emptyset$; (* Initialisierung des leeren AVL-Blattsuchbaums *)

WHILE $Q \neq \emptyset$ **DO**

$q :=$ nächster Haltepunkt der Scanline;
IF $q = a_i$ **THEN**
 füge A_i in L ein; (* A_i wird aktiviert *)
 IF A_i hat einen oberen Nachbarn A^+ in L **THEN**
 gib (A_i, A^+) als Sichtbarkeitspaar aus
FI
 IF A_i hat einen unteren Nachbarn A^- in L **THEN**
 gib (A_i, A^-) als Sichtbarkeitspaar aus
FI
FI
IF $q = b_i$ **THEN**
 IF A_i hat einen oberen und unteren Nachbarn A^+ bzw. A^- in L **THEN**
 gib das Sichtbarkeitspaar (A^+, A^-) aus
FI
 entferne A_i aus L ; (* A_i wird stillgelegt *)
FI

OD

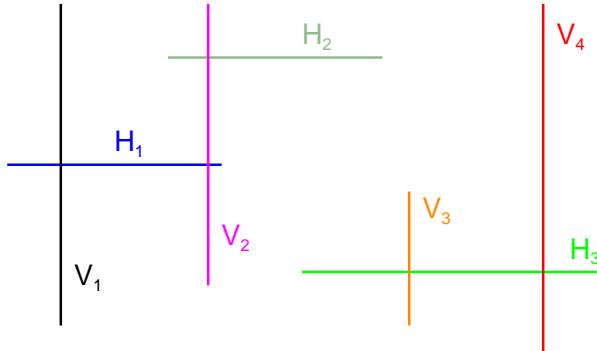
Der Algorithmus für das Sichtbarkeitsproblem ist in Algorithmus 108 formuliert.

Laufzeit: Die Sortierung der $2n$ Haltepunkte der Scanline kann in $\Theta(n \log n)$ Schritten durchgeführt werden. Das Bestimmen der oberen und unteren Nachbarn eines Segments A_i in L lässt sich mit AVL-Blattsuchbäumen, die mit einer doppelverketteten Liste für die Blätter ausgerüstet sind, in konstanter Zeit ausführen. Einfügen und Löschen erfordert jeweils $\mathcal{O}(\log n)$ Schritte. Damit ergibt sich die Gesamlaufzeit $\mathcal{O}(n \log n)$ für den skizzierten Algorithmus.

8.1.2 Das Schnittproblem für achsenparallele Segmente

Schnittprobleme für geometrische Objekte treten beispielsweise beim VLSI-Entwurf auf. Die verschiedenen Schichten eines Chips werden durch Rechtecke spezifiziert. Ein Entwurf ist nur dann korrekt, wenn die Rechtecke einer Schicht sich nicht schneiden.

Wir betrachten hier nur den einfacheren Fall von Liniensegmenten in der Ebene, wobei wir zunächst von achsenparallelen Segmenten ausgehen und in Abschnitt 8.1.3 den allgemeinen Fall (fast) beliebiger Liniensegmente der Ebene betrachten.



Gegeben seien n vertikale und horizontale Liniensegmente

$$\begin{aligned} V_i &= \{x_i\} \times [c_i, d_i], \quad i = 1, \dots, l \\ H_j &= [a_j, b_j] \times \{y_j\}, \quad j = 1, \dots, h. \end{aligned}$$

(Also ist $n = l + h$.) Wir nehmen o.B.d.A an, daß die Werte $a_j, b_j, x_i, j = 1, \dots, h$, $i = 1, \dots, l$, paarweise verschieden sind. Gesucht ist die Menge aller Schnittpaare, d.h. Paare (V_i, H_j) mit $V_i \cap H_j \neq \emptyset$.

Zunächst stellen wir fest, daß für ein gegebenes Paar (V_i, H_j) der Schnittest sehr einfach vollzogen werden kann. Es gilt nämlich:

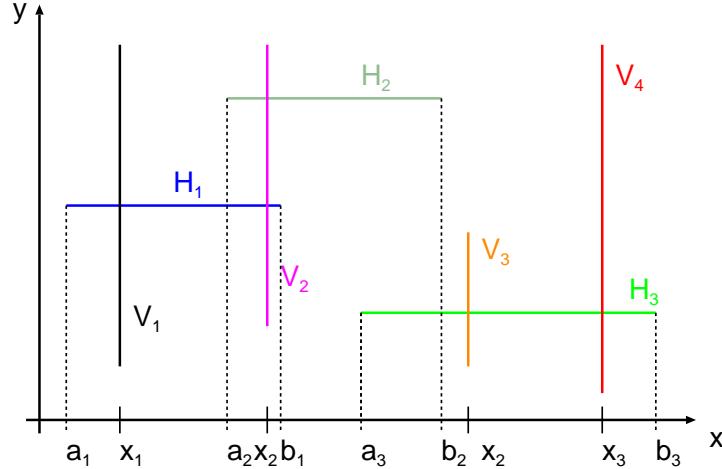
$$V_i \cap H_j \neq \emptyset \quad \text{gdw} \quad a_j \leq x_i \leq b_j \text{ und } c_i \leq y_j \leq d_i$$

Der naive Algorithmus, der für alle Paare (V_i, H_j) prüft, ob $H_j \cap V_i \neq \emptyset$ hat die worst-case Laufzeit $\Theta(l \cdot h) = \Theta(n^2)$. In der Tat ist diese worst-case Komplexität optimal in dem Sinn, daß es $n^2/4$ Schnittpaare geben kann und somit bereits die Ausgabe aller Schnittpaare die Kosten $\Theta(n^2)$ verursacht.

Wir geben nun ein Scanline-Verfahren an, das effizienter als das naive Verfahren ist, falls die Gesamtzahl k aller Schnittpaare wesentlich geringer als $n^2/4$ ist. Die Haltepunkte sind gegeben durch die Menge

$$Q = \{a_j, b_j : j = 1, \dots, h\} \cup \{x_i : i = 1, \dots, l\}$$

der relevanten x -Koordinaten. Wir organisieren Q als sortierte Liste. Am Haltepunkt $q \in Q$ sind genau die horizontalen Segmente H_j aktiv, für die gilt $a_j \leq q \leq b_j$.



In dem Beispiel auf der Folie sind am Haltepunkt x_2 die horizontalen Segmente H_1 und H_2 aktiv. Wir verwenden einen AVL-Blattsuchbaum, der die jeweils aktiven horizontalen Segmente verwaltet (Sortierung bzgl. der y -Koordinaten). Am Haltepunkt $q = x_i$ können durch die Bereichsanfrage in L

„bestimme alle Segmente H_j mit $c_i \leq y_j \leq d_i$ “

alle horizontalen Segmente H_j ermittelt werden, die einen nichtleeren Schnitt mit dem vertikalen Segment V_i haben.

Das skizzierten Verfahren kann (mit AVL-Blattsuchbäumen und Doppelverkettung der Blätter) in Zeit $\mathcal{O}(n \log n + k)$ durchgeführt werden, wobei k die Anzahl an Schnittpaaren und n die Anzahl an vertikalen und horizontalen Liniensegmenten ist.

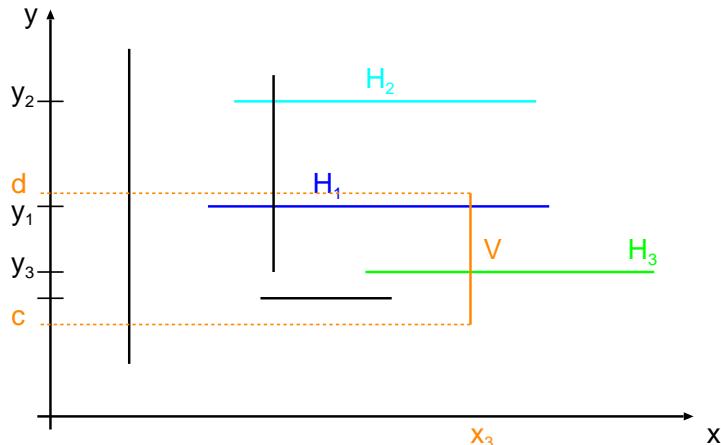
Algorithmus 109 Scanline-Algorithmus für das achsenparallele Schnittproblem

$Q :=$ sortierte Liste der Haltepunkte a_i, b_i der Scanline;

```

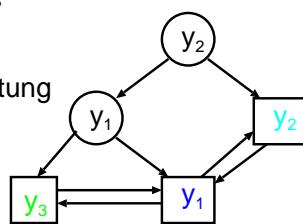
(* die aktiven horizontalen Segmente werden in einem AVL-Blattsuchbaum verwaltet
*)
(* (sortiert nach den y-Koordinaten) *)
 $L := \emptyset;$  (* Initialisierung des leeren AVL-Blattsuchbaums *)
WHILE  $Q \neq \emptyset$  DO
     $q :=$  nächster Haltepunkt der Scanline;
    IF  $q = a_j$  THEN
        füge  $H_j$  in  $L$  ein; (*  $H_j$  wird aktiviert *)
    FI
    IF  $q = b_j$  THEN
        entferne  $A_i$  aus  $L$ ; (*  $H_j$  wird stillgelegt *)
    FI
    IF  $q = x_i$  THEN
        bestimme alle horizontalen Segmente  $H_j \in L$  mit  $c_i \leq y_j \leq d_i$  und gib  $(V_i, H_j)$  aus;
    FI
OD

```



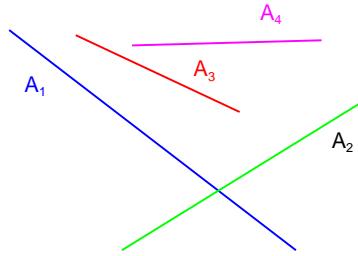
AVL-Blattbaum für die aktiven horizontalen Segmente z.B. am Haltepunkt x_3

Bereichsanfrage:
verwendet die Verkettung
der Blätter



8.1.3 Das allgemeine Liniensegment-Schnittproblem

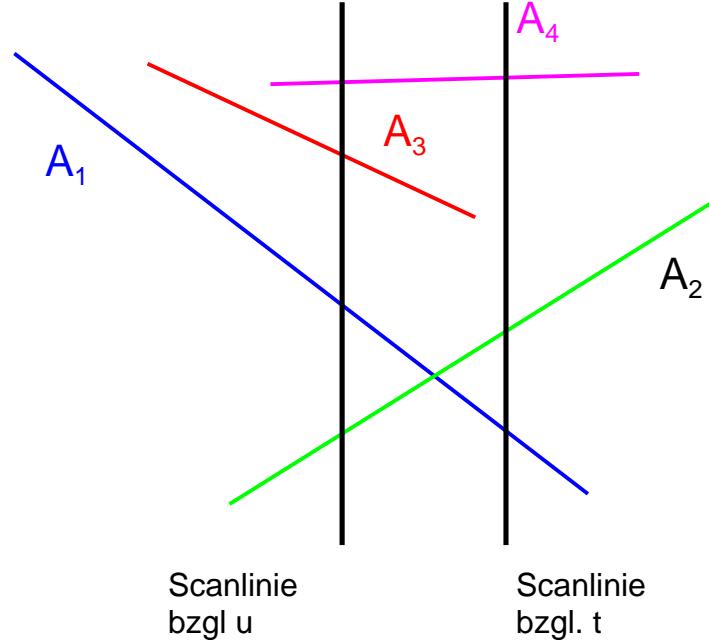
Gegeben seien n Liniensegmente A_1, \dots, A_n der Ebene (nicht notwendig horizontal oder vertikal). Gefragt ist, ob es zwei sich schneidende Segmente gibt.



Wir machen hier einige vereinfachende Annahmen:

- Keines der Segmente verläuft parallel zur y -Achse.
- Die x -Koordinaten der Anfangs- und Endpunkte der Segmente sind p.v..
- In jedem Punkt der Ebene schneiden sich höchstens zwei Segmente.

Wir lösen das allgemeine Liniensegment-Schnittproblem mit einem Scanline-Algorithmus, der ähnlich arbeitet wie der in Abschnitt 8.1.2 beschriebene Algorithmus für das Schnittproblem achsenparalleler Segmente. Der wesentliche Unterschied ist, daß die Sortierung der aktiven Liniensegmente bzgl. der y -Koordinaten sich dynamisch ändern kann. Die Menge Q der Haltepunkte besteht aus den x -Koordinaten aller Anfangs- und Endpunkte der gegebenen Segmente. L bezeichne die Menge aller aktiven Segmente. Für jeden Haltepunkt t der Scanline definieren wir eine Ordnung \leq_t der aktiven Liniensegmente. Seien A_i, A_k zwei Liniensegmente, die an einem Haltepunkt t der Scanline aktiv sind. Wir definieren: $A_i \leq_t A_k$ gilt genau dann, wenn für die eindeutig bestimmten Punkte $(t, y_i) \in A_i$ und $(t, y_k) \in A_k$ gilt $y_i < y_k$. Beachte: Ist $A_i \leq_u A_k$, so ist $A_k \leq_t A_i$ für einen späteren Haltepunkt t (d.h. $t > u$) möglich.



In der obigen Skizze gilt $A_1 \leq_u A_2$, aber $A_2 \leq_t A_1$. Es gilt:

- Sind A_i und A_k am Haltepunkt t aktiv und galt $A_i \leq_u A_k$ für einen Haltepunkt $u < t$, dann ist entweder $A_i \leq_t A_k$ oder $A_i \cap A_k \neq \emptyset$.
- $A_i \cap A_k \neq \emptyset$ impliziert, daß A_i und A_k unmittelbare Nachbarn in L für einen Haltepunkt t sind.

Diese Beobachtungen werden in dem Algorithmus für das allgemeine Schnittproblem ausgenutzt. L wird als AVL-Blattsuchbaum organisiert. Beim Einfügen eines Segments in L am Haltepunkt t wird die jeweils aktuelle Ordnung \leq_t zugrundegelegt.

- Sobald ein Segment A_j in L eingefügt wird, wird für jeden unmittelbaren Nachbarn A_i geprüft, ob A_i einen nichtleeren Schnitt mit A_j hat.
- Beim Löschen von A_j aus L wird geprüft, ob A_j zwei unmittelbare Nachbarn A_i und A_k in L hatte und ob diese einen nichtleeren Schnitt haben.

Der eigentliche Schnittest kann durch Lösen eines linearen Gleichungssystems mit zwei Unbekannten durchgeführt werden. Sei $\alpha_i = (x_{i,1}, y_{i,1})$ der Anfangspunkt von A_i und $\beta_i = (x_{i,2}, y_{i,2})$ der Endpunkt von A_i . D.h. es gilt $x_{i,1} < x_{i,2}$ und

$$A_i = \{\alpha_i + \lambda(\beta_i - \alpha_i) : 0 \leq \lambda \leq 1\}.$$

Dann ist $A_i \cap A_j \neq \emptyset$ genau dann, wenn es reelle Zahlen $\lambda, \mu \in [0, 1]$ mit

$$\begin{aligned} x_{i,1} + \lambda(x_{i,2} - x_{i,1}) &= x_{j,1} + \mu(x_{j,2} - x_{j,1}) \\ y_{i,1} + \lambda(y_{i,2} - y_{i,1}) &= y_{j,1} + \mu(y_{j,2} - y_{j,1}) \end{aligned}$$

Algorithmus 110 Scanline-Algorithmus für das allgemeine Schnittproblem

$Q :=$ sortierte Liste der x -Koordinaten der Anfangs-/Endpunkte von A_1, \dots, A_n ;

(* die aktiven Liniensegmente werden in einem AVL-Blattsuchbaum verwaltet *)
(* (sortiert nach den y -Koordinaten) *)
 $L := \emptyset$; (* Initialisierung des leeren AVL-Blattsuchbaums *)

WHILE $Q \neq \emptyset$ **DO**

$t :=$ nächster Haltepunkt der Scanline;
IF $t = x$ -Koordinate des Anfangspunkts von A_j **THEN**
 füge A_j in L ein; (* A_j wird aktiviert *)
 IF A_j hat einen unmittelbaren Nachbarn A_i in L mit $A_j \cap A_i \neq \emptyset$ **THEN**
 return „Schnittpunkt entdeckt“
FI
FI
IF $t = x$ -Koordinate des Endpunkts von A_j **THEN**
 IF A_j hat zwei unmittelbare Nachbarn A_i, A_k in L mit $A_i \cap A_k \neq \emptyset$ **THEN**
 return „Schnittpunkt entdeckt“
FI
 entferne A_j aus L ; (* A_j wird stillgelegt *)
FI

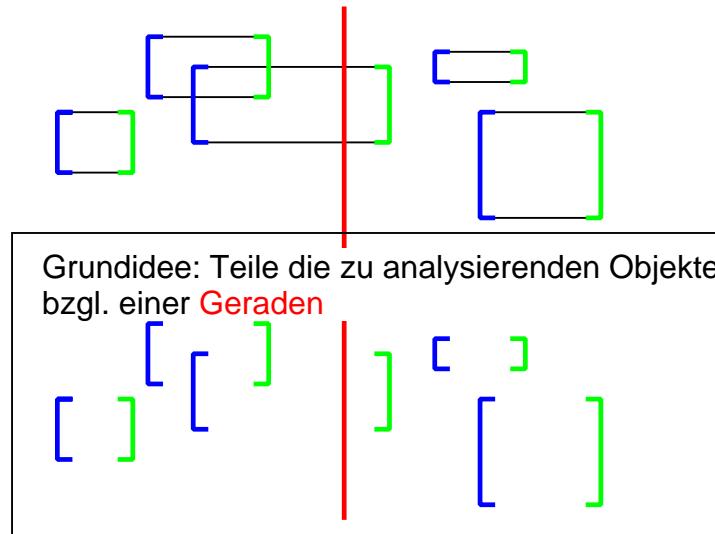
OD

gibt. Für jedes Segmentpaar (A_i, A_j) kann der Schnittest also in konstanter Zeit durchgeführt werden. Für jeden der $2n$ Haltepunkte der Scanline sind eine Einfüge- oder Löschoperation sowie ein oder zwei Schnittests auszuführen. Der Aufwand pro Haltepunkt ist also konstant. Die Sortierung der Haltepunkte erfordert $\mathcal{O}(n \log n)$ Zeiteinheiten. Die gesamte Laufzeit des beschriebenen Verfahrens ist $\mathcal{O}(n \log n)$.

Das zuletzt beschriebene Verfahren löst lediglich die existentielle Version des allgemeinen Schnittproblems (d.h. gibt lediglich einen Schnittpunkt aus, falls ein solcher existiert). Das Verfahren kann zum Berechnen aller Schnittpunkte erweitert werden. Dazu ist es erforderlich, daß für jeden gefundenen Schnittpunkt, die betreffenden Segmente in L umgeordnet werden. Ist k die Anzahl der Schnittpunkte, dann ist eine Implementierung mit der Rechenzeit $\mathcal{O}((n+k) \log n)$ möglich. Wir verzichten auf Details dieser Erweiterung.

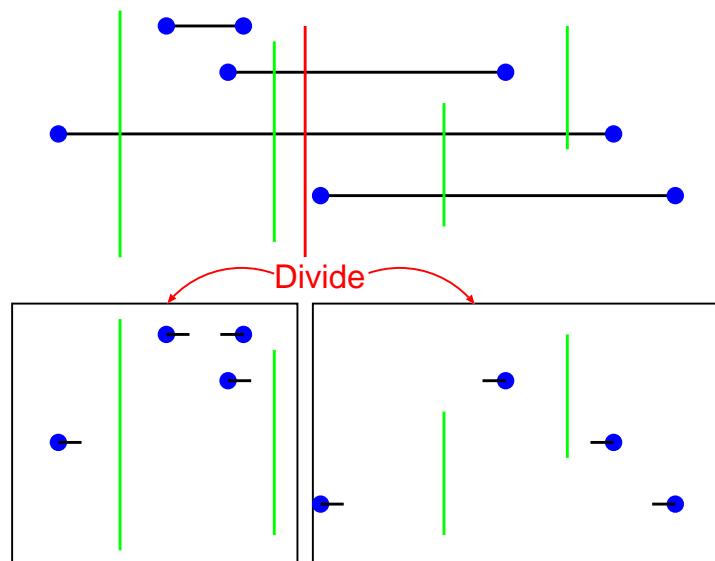
8.2 Geometrisches DIVIDE & CONQUER

Die DIVIDE & CONQUER-Technik (Vgl. Abschnitt 5.1 lässt sich auch für einige geometrische Probleme anwenden. In der Zerlegungsphase findet beim geometrischen DIVIDE & CONQUER eine Aufteilung der zu analysierenden Objekte mit Hilfe einer (oftmals achsenparallelen) Geraden statt. In manchen Fällen müssen auch die Objekte selbst zweigeteilt werden. Dies setzt gewisse problemabhängige „Tricks“ zur Darstellung der Objekte voraus.



Schnittproblems für achsenparallele Liniensegmente Wir erläutern die Grundideen am Beispiel des Schnittproblems für achsenparallele Liniensegmente (vgl. Abschnitt 8.1.2). Wie zuvor seien n vertikale und horizontale Segmente V_1, \dots, V_l und H_1, \dots, H_h gegeben, wobei wir zur Vereinfachung annehmen, daß die Anfangs- und Endpunkte der horizontalen Segmente paarweise verschieden sind. In der Zerlegungsphase wird eine zur x -Achse parallele Gerade verwendet, die kein vertikales Segment und keinen Anfangs- oder Endpunkt eines horizontalen Segments enthält.

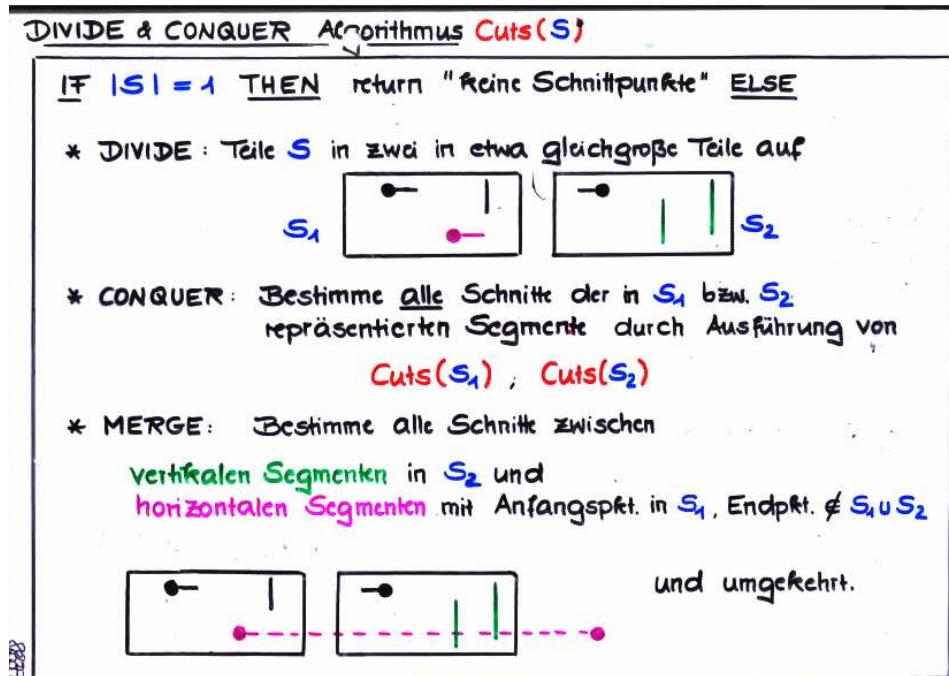
- Jedes vertikale Segment kann eindeutig der linken oder rechten Halbebene zugeordnet werden.
- Die horizontalen Segmente werden jeweils durch ihren Anfangs- und Endpunkt dargestellt. Diese können eindeutig der linken oder rechten Ebene zugeordnet werden.



Wir formulieren einen rekursiven DIVIDE & CONQUER-Algorithmus $Cuts(S)$, der als Eingabe eine Menge $S \subseteq \{V_1, \dots, V_l\} \cup A \cup E$ hat. A bzw. E stehen für die Mengen der Anfangs- bzw. Endpunkte der horizontalen Segmente. Ist $H_j = [a_j, b_j] \times \{y_j\}$, dann ist $A = \{(a_j, y_j) : j = 1, \dots, h\}$ und $E = \{(b_j, y_j) : j = 1, \dots, h\}$. Wir können uns S als einen y -achsenparallelen Streifen vorstellen. Ein horizontales Segment heißt in S repräsentiert, falls der Anfangs- oder Endpunkt (oder beide) in S liegen. Für $|S| \geq 2$ ist die Vorgehensweise wie folgt:

- (1) S wird entlang einer zur x -Achse parallelen Gerade in zwei in etwa gleichgroße Teilmengen S_1 und S_2 zerlegt.
- (2) Alle Schnittpunkte von in S_1 bzw. S_2 repräsentierten Segmenten werden (rekursiv) ermittelt.⁷¹
- (3) Alle in S liegenden Schnittpunkte, die nicht in Schritt (2) gefunden wurden, werden ermittelt.

Der Fall $|S| = 1$ ist trivial, da dann kein Schnittpunkt in S liegt.



Die Korrektheit des Verfahrens lässt sich mit Induktion nach $|S|$ beweisen. Wir zeigen, daß $Cuts(S)$ sämtliche Schnittpaare (H, V) mit $H \cap V \neq \emptyset$ ausgibt, wobei V ein vertikales Segment in S und H ein horizontales Segment in S ist, so daß der Anfangs- oder Endpunkt von H in S liegt. Der Induktionsanfang $|S| \in \{0, 1\}$ ist klar, da es keine Schnittpunkte in S gibt. Im Induktionsschritt nehmen wir an, daß $|S| \geq 2$ ist und in die Mengen S_1 ,

⁷¹In Schritt (2) werden nur solche Schnittpaare (H_j, V_i) erfaßt, so daß $V_i \in S$ und $\{(a_j, y_j), (b_j, y_j)\} \in S$. Insbesondere liegt dann der Schnittpunkt selbst, in dem durch S gegebenen „Streifen“.

S_2 zerlegt wird. Aus Symmetriegründen reicht es, Paare (H, V) zu betrachten, so daß V ein vertikales Segment in S_2 ist und H ein horizontales Segment, dessen Anfangs- oder Endpunkt von H in S_1 liegt. Für die Lage von H gibt es vier Fälle.

Zum DIVIDE & CONQUER Algorithmus:

Sei H ein horizontales Segment, das in S_1 repräsentiert ist.

1. Fall:



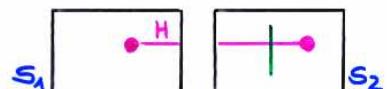
Kein vertikales Segment in S_2 schneidet H .

2. Fall:



Kein vertikales Segment in S_2 schneidet H .

3. Fall:



Alle Schnittpunkte von H mit vertikalen Segmenten in S_2 werden in $Cuts(S_2)$ gefunden.

4. Fall:

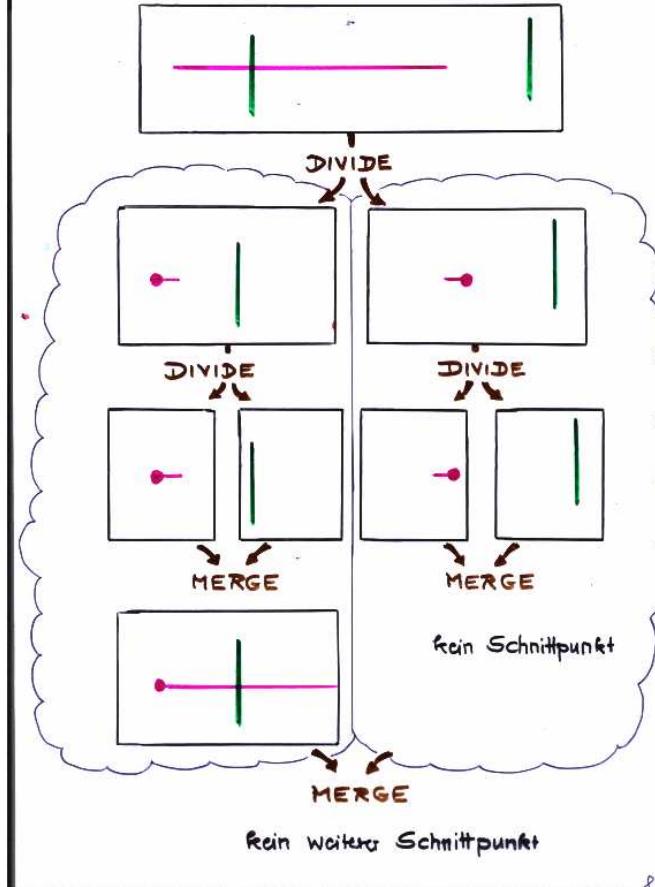


Die Schnittpunkte von H mit vertikalen Segmenten in S_2 werden im MERGE-Schritt gefunden.

Alle vier Fälle ergeben, daß – sofern $V \cap H \neq \emptyset$ – der Schnittpunkt von V und H in $Cuts(S)$ gefunden wird.

Wir betrachten ein Beispiel:

Beispiel: Divide and Conquer für das Schnittproblem



PL2

Schritt (1) kann mit einer Sortierung der Anfangs- und Endpunkte und vertikalen Segmenten nach den x -Koordinaten in Zeit $\mathcal{O}(n \log n)$ durchgeführt werden. Zur Erleichterung von Schritt (2) empfiehlt sich eine Numerierung der Objekte. Die Menge S kann dann einfach durch die Angabe der Nummern i und j des ersten bzw. letzten Objekts von S beschrieben werden. (D.h. die Parameter des skizzierten Algorithmus $Cuts(\dots)$ sind die Nummern i und j des ersten und letzten Objekts von S .) Zur Unterstützung von Schritt (3) können drei zusätzliche sortierte Listen verwendet werden. Sei $\alpha_\nu = (a_\nu, y_\nu)$ der Anfangs- und $\beta_\nu = (b_\nu, y_\nu)$ der Endpunkt des horizontalen Segments $H_\nu = [a_\nu, b_\nu] \times \{y_\nu\}$. Wir benutzen:

- eine Liste für die Anfangspunkte $\alpha_\nu \in S$, so daß $\beta_\nu \notin S$ (Liste 1)
- eine Liste für die Endpunkte $\beta_\nu \in S$ mit $\alpha_\nu \notin S$ (Liste 2)
- eine Liste für die vertikalen Segmente in S (Liste 3).

Die Listen für die Anfangs- und Endpunkte werden bzgl. der y -Koordinaten sortiert. Die dritte Liste (für die vertikalen Segmente) wird bzgl. der unteren Intervallgrenze c_μ der vertikalen Segmente $V_\mu = \{x_\mu\} \times [c_\mu, d_\mu]$ sortiert. Besteht S aus den Objekten mit

den Nummern $i, i+1, \dots, j$, dann besteht S_1 aus den Objekten i, \dots, m und S_2 aus den Objekten $m+1, \dots, j$. Dabei ist $m = \lfloor (i+j)/2 \rfloor$. Durch ein „verzahntes“ Durchlaufen von Liste 1 und 3 kann man alle Paare (H_ν, V_μ) ausfindig machen, so daß $\alpha_\nu \in S_1, \beta_\nu \notin S$ und $V_\mu \in S_2$. Entsprechend erhalten wir alle Schnittpaare (H_ν, V_μ) mit $\alpha_\nu \notin S, \beta_\nu \in S_2$ und $V_\mu \in S_1$, in dem wir Liste 2 und 3 verzahnt durchlaufen. Für diese Paare muß geprüft werden, ob $H_\nu \cap V_\mu = \emptyset$. Dies ist aber gleichbedeutend damit, daß $c_\mu \leq y_\nu \leq d_\mu$.

Laufzeit: Mit der oben erwähnten Listendarstellung können wir die Laufzeit durch die Rekurrenz

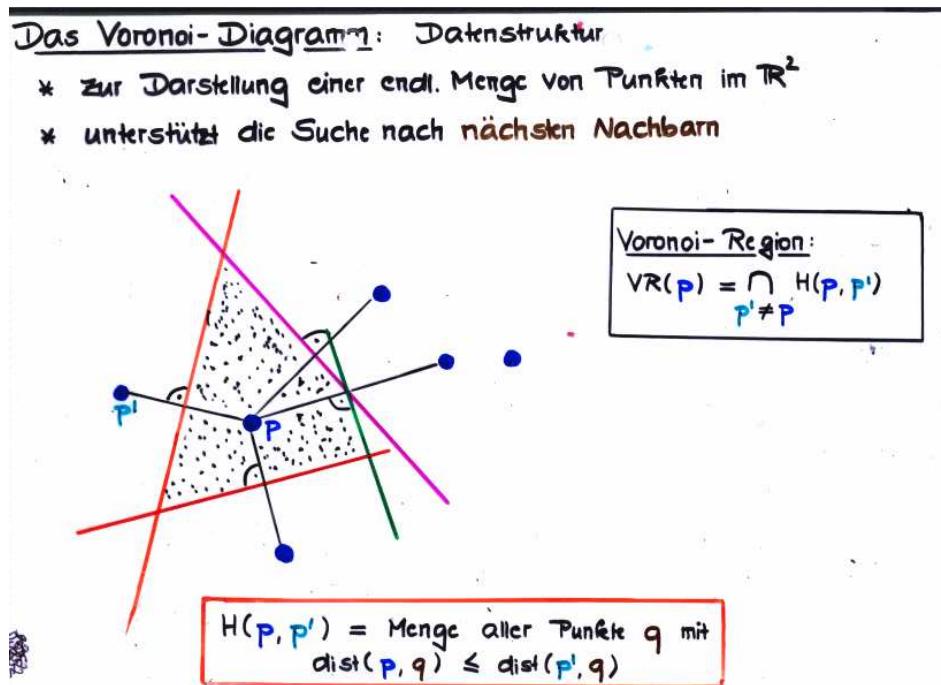
$$T(n) = \begin{cases} 1 & : \text{falls } n = 1 \\ 2T(\lceil n/2 \rceil) + cn & : \text{falls } n \geq 2 \end{cases}$$

abschätzen, wobei die Ausgabe der Schnittpunkte zunächst nicht berücksichtigt ist.⁷² Diese hat die Lösung $\mathcal{O}(n \log n)$ (siehe Satz 5.5.14). Damit ergibt sich die Gesamlaufzeit $\mathcal{O}(n \log n + k)$, wenn k die Anzahl an Schnittpunkten ist.

Das Schnittproblem für achsenparallele Liniensegmente lässt sich also sowohl mit dem Scanline-Prinzip als auch der Methode des geometrischen DIVIDE & CONQUER in Zeit $\mathcal{O}(n \log n + k)$ lösen. Dabei ist n die Anzahl der Liniensegmente und k die Anzahl der Schnittpunkte.

8.3 Das Voronoi-Diagramm

Zahlreiche Distanzprobleme lassen sich leicht mit dem sogenannte Voronoi-Diagramm lösen. Wir geben hier lediglich eine sehr kurze informelle Erläuterung der Grundideen an.



⁷²Der „Knackpunkt“ an der genannten Rekurrenz sind die linearen Zusammensetzungskosten. Diese kann man durch die oben erwähnte Listendarstellung gewährleisten. Diese ermöglicht es, nur linear viele Segmentpaare (H, V) mit leerem Schnitt zu betrachten.

Gegeben sei eine endliche Menge P von Punkten in der Ebene. Typische Distanzprobleme, die sich mit dem Voronoi-Diagramm lösen lassen, sind die Frage nach einem dichtesten Punktpaar (d.h. verschiedene Punkte $p_1, p_2 \in P$ mit minimalem Abstand), das Problem aller nächsten Nachbarn (d.h. zu jedem Punkt $p \in P$ ist ein Punkt $p' \in P \setminus \{p\}$ mit minimalem Abstand zu p gesucht), das sogenannte Postmann-Problem (bei dem zu einem Punkt $z \in \mathbb{R}^2$ ein Punkt $p \in P$ mit minimalem Abstand gesucht ist) oder die Frage nach der konvexen Hülle von P . Dabei wird stets der Euklidische Abstand

$$dist(p, p') = \sqrt{(x - x')^2 + (y - y')^2} \quad \text{für } p = (x, y), p' = (x', y')$$

zugrundegelegt. Das Voronoi-Diagramm ist eine Datenstruktur, die die Suche nach nächsten Nachbarn unterstützt.

Für $p, p' \in \mathbb{R}^2$ sei $H(p|p')$ die Halbebene

$$H(p|p') = \{q \in \mathbb{R}^2 : dist(p, q) < dist(p', q)\}$$

aller Punkte q , die näher an p als an p' liegen. Sei $P = \{p_1, \dots, p_n\}$, wobei $p_i = (x_i, y_i)$, $i = 1, \dots, n$. Die *Voronoi-Region* von p_i bzgl. P ist

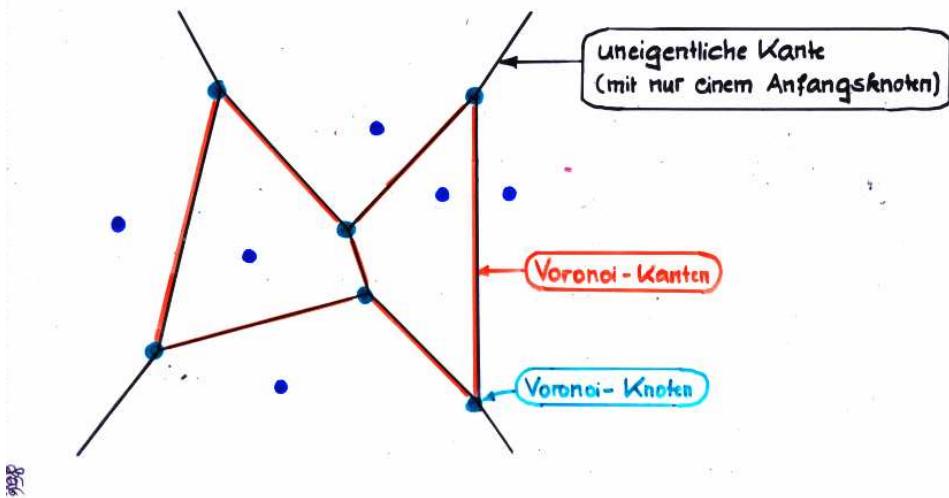
$$VR(p_i) = \bigcap_{p' \in P \setminus \{p_i\}} H(p_i|p'),$$

die Menge aller Punkte, für die p der nächstgelegene Punkt in P ist. Ist $P = \{p\}$ einelementig, dann wird $VR(p) = \mathbb{R}^2$ gesetzt. $VR(p_i)$ kann beschränkt oder unbeschränkt sein. Wir bezeichnen mit *Rand*(p_i) die Menge aller Randpunkte der Voronoi-Region von p_i . Für $|P| = n \geq 2$ bestehen die Mengen *Rand*(p_i) aus einem zusammenhängenden Zug von (beschränkten oder unbeschränkten) Geradenstücken, die die Voronoi-Region von p umranden. Die Geradenstücke ergeben sich genau als diejenigen Schnittmengen $\text{Rand}(p_i) \cap \text{Rand}(p_j)$, für die p_i und p_j benachbarte Voronoi-Regionen haben, d.h. für die

$$\text{Abschluß von } VR(p_i) \cap \text{Abschluß von } VR(p_j) \neq \emptyset.$$

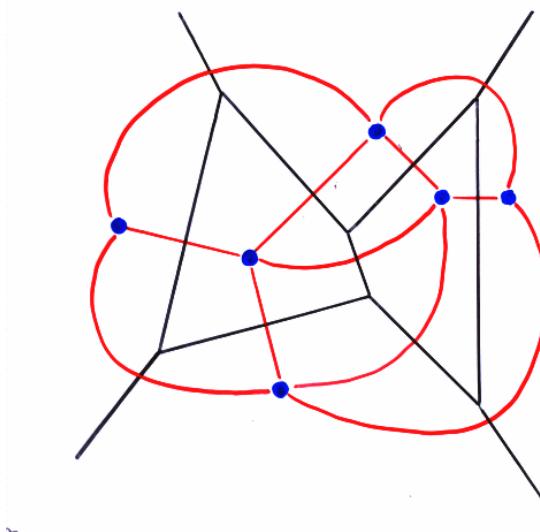
Dabei besteht der Abschluß von $VR(p_i)$ gerade aus der Voronoi-Region von p_i und der Randmenge *Rand*(p_i). Zwei Punkte $p_i, p_j \in P$, $i \neq j$, sind also genau dann benachbart, wenn deren Randmengen einen nichtleeren Schnitt haben. Die Schnittpunkte der abgeschlossenen Voronoi-Regionen werden auch *Voronoi-Knoten* genannt. Die *Voronoi-Kanten* sind die Geradenstücke, die sich als Schnittmengen der Abschlüsse zweier Voronoi-Regionen (bzw. der Randmengen) ergeben.

Das Voronoi-Diagramm: Computer-interne Darstellung
als ungerichteter Graph (mit doppelt verketteter Kantenliste)



Der zugehörige *duale Graph* besteht aus der Knotenmenge $V = P$ und der Kantenrelation E mit $(v, w) \in E$ genau dann, wenn p und q benachbarte Voronoi-Regionen haben.

Der duale Graph eines Voronoi-Diagramms

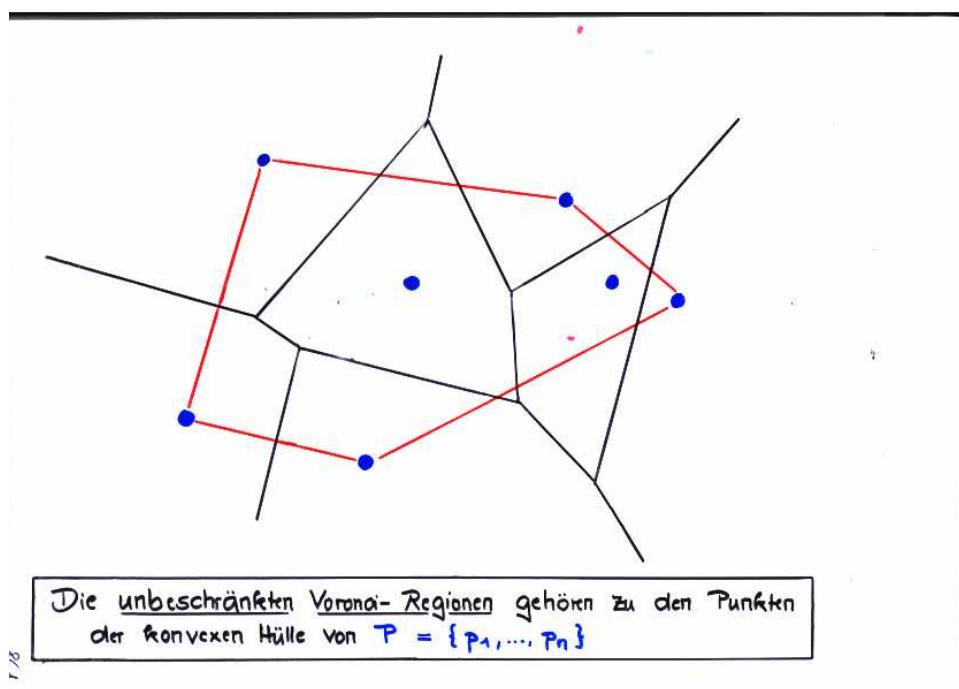


Wir geben drei fundamentale Eigenschaften des Voronoi-Diagramms an, die als Basis für Algorithmen zum Lösen von „Nächste-Nachbar-Problemen“ oder zum Bestimmen der konvexen Hülle von P dienen können.

- (a) Für jeden Punkt $q \in \mathbb{R}^2$ gibt es (mindestens) einen Punkt p in P mit $q \in VR(p)$.
Für jeden solchen Punkt $p \in P$ gilt:

$$dist(p, q) = \min_{p' \in P} dist(p', q).$$

- (b) Sind p_i und p_j nächste Nachbarn, dann hat der duale Graph eine Kante, die p_i und p_j verbindet, d.h. dann sind die Voronoi-Regionen von p_i und p_j benachbart.
(c) Die Punkte $p \in P$, für die $VR(p)$ unbeschränkt ist, sind genau diejenigen Punkte in P , die zur reflexiven Hülle von P gehören.



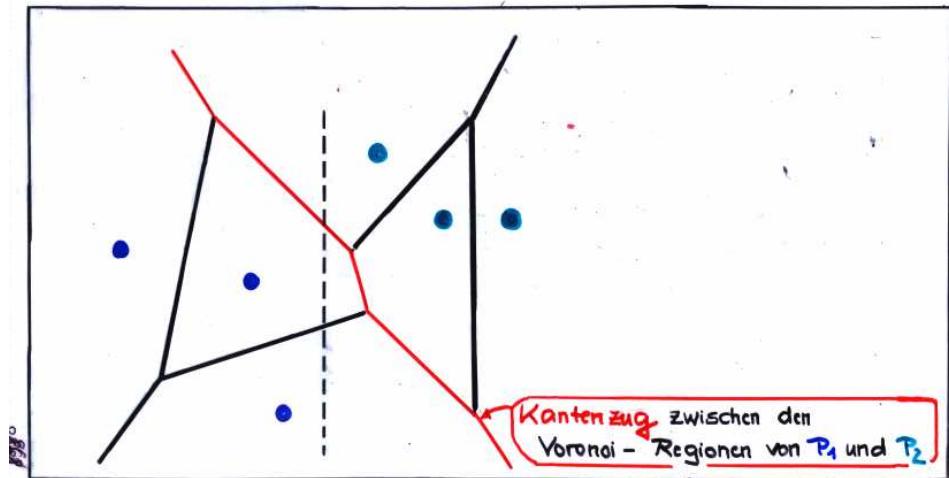
Das Voronoi-Diagramm lässt sich mit einem Algorithmus, der auf dem geometrischen DIVIDE & CONQUER beruht, in Zeit $\mathcal{O}(n \log n)$ konstruiert werden.

Divide & Conquer - Verfahren zum Erstellen des Voronoi-Diagramms

DIVIDE : Teile die Punktmenge in etwa gleichgroße Teilmengen P_1 und P_2

CONQUER : Konstruiere die Voronoi-Diagramme für die Teilmengen P_1 und P_2

MERGE : setze die Voronoi-Diagramme zusammen



Die wesentliche Grundidee besteht darin, die Punktmenge P in zwei in etwa gleichgroße Teilmengen P_1 und P_2 aufzuteilen, für die das Voronoi-Diagramm (rekursiv) erstellt wird. Schließlich wird ein Kantenzug ermittelt, der es ermöglicht, die Voronoi-Diagramme für P_1 und P_2 zu dem Voronoi-Diagramm für P zusammenzusetzen. Wir verzichten an dieser Stelle auf Details.

9 Algebraische und arithmetische Probleme

9.0.1 Matrizenmultiplikation nach Strassen

Gegeben sind zwei $n \times n$ Matrizen A und B , wobei $n = 2^k$ eine Zweierpotenz ist (und k eine ganze Zahl ≥ 1). Gesucht ist das Matrizenprodukt $C = A \cdot B$.

Wir zerlegen die drei Matrizen in jeweils 4 ($\frac{n}{2} \times \frac{n}{2}$)-Matrizen.

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \quad B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \quad C = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Matrizenmultiplikation :

Gegaben: ($n \times n$) - Matrizen A, B (wobei $n = 2^k$ Zweierpotenz)

Gesucht: $C = A \cdot B$

Schulmethode:

Laufzeit: $\Theta(n^3)$

FOR $i = 1, \dots, n$ DO
 FOR $j = 1, \dots, n$ DO berechne $C[i, j] := \sum_{k=1}^n A[i, k] \cdot B[k, j]$ OD
OD

Naives DIVIDE & CONQUER:

Zerlege A, B in $(\frac{n}{2} \times \frac{n}{2})$ -Matrizen und berechne

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

durch

$C_{11} := A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$	$C_{12} := A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$
$C_{21} := A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$	$C_{22} := A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$

Für die Berechnung von C durch die Formeln

$$C_{i,j} = A_{i,1} \cdot B_{1,j} + A_{i,2} \cdot B_{2,j}$$

erhält man die Kostenfunktion $T(1) = 1$ und

$$T(n^2) = 8 \cdot T\left(\left(\frac{n}{2}\right)^2\right) + c \cdot n^2.$$

Dabei ist c eine reelle positive Konstante, deren exakter Wert für die asymptotischen Kosten unerheblich ist. Diese ergibt sich aus der Beobachtung, daß 8 Multiplikationen und 4 Additionen von $(\frac{n}{2} \times \frac{n}{2})$ -Matrizen nötig sind.⁷³ Mit $N = n^2$ folgt aus Satz 5.1.1:

$$T(N) = 8 \cdot T(N/4) + c \cdot N = \Theta(N^{\log_4 8}) = \Theta(n^3).$$

⁷³Tatsächlich wird man als „Eingabegröße“ nicht die Anzahl n^2 der Matrixelemente sondern die Anzahl n der Zeilen bzw. Spalten für die Kostenfunktion verwenden. D.h. letztendlich ist nicht T sondern die Funktion $T'(n) = T(n^2)$ die Kostenfunktion.

Beachte:

$$N^{\log_4 8} = n^{2 \cdot \log_4 8} = n^{2 \cdot 3/2} = n^3.$$

Hiermit ist also gegenüber der „Schulmethode“ keine Zeitersparnis zu verzeichnen.

Matrizenmultiplikation nach Strassen: Wir betrachten nun die von Strassen vorgeschlagene Methode. Diese benutzt ebenfalls die Zerlegung der beiden Matrizen A und B in die Untermatrizen $A_{i,j}$ und $B_{i,j}$. Die Berechnung der Untermatrizen $C_{i,j}$ der Produktmatrix C beruht auf dem folgenden Schema.

Matrizenmultiplikation nach Strassen: ($n = 2^k \geq 2$)

Berechne das Matrizenprodukt $\mathbf{A} \cdot \mathbf{B} = \mathbf{C}$ für $n \times n$ -Matrizen

gemäß der Zerlegung in $(\frac{n}{2} \times \frac{n}{2})$ -Matrizen

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad \mathbf{C} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

und der Berechnungsvorschrift:

$M_1 := (A_{11} - A_{22}) \cdot (B_{21} + B_{22})$	$M_2 := (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$	
$M_3 := (A_{21} - A_{12}) \cdot (B_{11} + B_{12})$	$M_4 := (A_{11} + A_{12}) \cdot B_{22}$	
$M_5 := A_{11} \cdot (B_{12} - B_{22})$	$M_6 := A_{22} \cdot (B_{21} - B_{11})$	$M_7 := (A_{21} + A_{22}) \cdot B_{11}$

7 Multiplikationen für $(\frac{n}{2} \times \frac{n}{2})$ -Matrizen

$C_{11} := M_1 + M_2 - M_4 + M_6$	$C_{12} := M_4 + M_5$
$C_{21} := M_6 + M_7$	$C_{22} := M_2 + M_3 + M_5 - M_7$

5

Die Methode von Strassen benötigt nur 7 Multiplikationen und 18 Additionen/Subtraktionen von $(\frac{n}{2} \times \frac{n}{2})$ -Matrizen. Dies führt zur Rekurrenz

$$T(n^2) = 7 \cdot T\left(\left(\frac{n}{2}\right)^2\right) + c \cdot n^2.$$

Mit $N = n^2$ folgt aus Satz 5.1.1:

$$T(N) = 7 \cdot T(N/4) + c \cdot N = \Theta(N^{\log_4 7}) = \Theta(n^{\log 7}) \approx \Theta(n^{2.8}).$$

Wir fassen die Ergebnisse zusammen:

Satz 9.0.1. Mit der Methode von Strassen können zwei $n \times n$ -Matrizen in Zeit $\Theta(n^{\log 7})$ multipliziert werden.

Experimentelle Ergebnisse haben gezeigt, daß die Strassen-Methode erst für großes n (ca. $n \geq 500$) besser als die Schulmethode ist. In der Praxis empfiehlt sich daher eine gemischte Strategie, die für $n \geq 500$ eine Zerlegung à la Strassen vornimmt. Sobald die Zeilenanzahl auf einen Wert < 500 reduziert ist, wird die Schulmethode angewandt, um die Teilprodukte zu ermitteln.

10 Randomisierung

Die bisher besprochenen Algorithmen waren so formuliert, daß der jeweils nächste auszuführende Schritt eindeutig festgelegt war. Solche Algorithmen werden auch *deterministisch* genannt. In einigen Fällen haben wir auch *nicht-deterministische* Formulierungen wie „wähle ein Element mit gewissen Eigenschaften“ verwendet, die mehrere Alternativen zur Auswahl stellen. Z.B. haben wir in der Formulierung von Quicksort (vgl. Abschnitt 2.1.3) die Auswahl des Pivotelements offengelassen. Ein anderes Beispiel war der Algorithmus von Ford & Fulkerson (vlg. Abschnitt 6.6.3), bei dem wir die Auswahl des zunehmenden Pfads nicht weiter spezifiziert haben. In all diesen Fällen ist der Nichtdeterminismus als Implementierungsfreiheit zu interpretieren. Formulierungen wie „wähle eine Element x , so daß ...“ standen stellvertretend für alle deterministischen Formulierungen $x := \dots$, die sich durch die Fixierung einer der möglichen Alternativen ergeben.

In diesem Abschnitt erweitern wir Algorithmen (und Datenstrukturen) um das Konzept von *Randomisierung*. In randomisierten Algorithmen können Zufallsfunktionen eingesetzt werden, die letztendlich für die Entscheidungen, welche Schritte als nächstes ausgeführt werden, verantwortlich sind. Typische Zufallsfunktionen ergeben sich durch das Werfen einer fairen Münze.

⋮

$x := \text{random}(\text{head}, \text{tail});$

IF $x = \text{head}$ **THEN** ... **ELSE** ... **FI**

⋮

Anweisungen wie $x := \text{random}(x_1, \dots, x_n)$ können als das Werfen eines n -seitigen Würfels interpretiert werden.

Randomisierte Algorithmen haben gegenüber den konventionellen deterministischen Algorithmen zwei wesentliche Vorteile: Sie sind oftmals konzeptionell einfacher und/oder effizienter als die beste bekannte deterministische Lösung. Andererseits ist die Analyse randomisierter Algorithmen häufig wesentlich aufwendiger als die Bestimmung der worst-case Laufzeit eines deterministischen Verfahrens. Dies liegt im Wesentlichen daran, daß es bei fester Eingabe zu unterschiedlichen Laufzeiten sowie unterschiedlichem Speicherplatzbedarf kommen kann. Zudem sind sogar unterschiedlichen Ergebnisse (Antworten des Algorithmus) bei fester Eingabe möglich. Alle drei Größen sind *Zufallsgrößen*. Wir machen uns diese Aussage an sehr einfachen Beispielen klar.

Erwartete Ausgabe. Liegt ein Algorithmus mit ganzzahliger Ausgabe vor, dann ist die erwartete Ausgabe durch die Formel

$$\sum_{i=-\infty}^{+\infty} \text{Prob}(\text{die Ausgabe hat den Wert } i) * i$$

gegeben. Eine entsprechende Formel gilt für Algorithmen mit reeller Ausgabe eines abzählbaren Bereichs. Dies ist für uns ohne Belang. Als Beispiel betrachten wir Algorithmus 111. Die Ausgabe ist entweder -10 oder 24 . Nimmt man Gleichverteilung (eine faire

Algorithmus 111 Beispiel für einen randomisierten Algorithmus mit zufälligem Ergebnis

```

 $x := \text{random}(\text{head}, \text{tail});$ 
IF  $x = \text{head}$  THEN
    gib  $-10$  aus
ELSE
    gib  $24$  aus
FI
```

Münze) an, dann ist die erwartete Ausgabe $1/2 * (-10) + 1/2 * 24 = -5 + 12 = 7$.⁷⁴ Man beachte, daß hier

$$\text{Prob}(\text{die Ausgabe hat den Wert } -10) = \frac{1}{2} = \text{Prob}(\text{die Ausgabe hat den Wert } 24)$$

und $\text{Prob}(\text{die Ausgabe hat den Wert } i) = 0$ für alle $i \notin \{-10, 24\}$.

Erwartete Laufzeit. Die generelle Formel, die die erwartete Laufzeit eines randomisierten Algorithmus angibt, ist

$$T_{avg}(n) = \sum_{k=0}^{\infty} \text{Prob}(t(n) = k) * k,$$

wobei n für die Eingabegröße und $t(n)$ für die Anzahl an tatsächlich ausgeführten Rechenschritten steht. Für eine Laufzeitanalyse wird man jedoch oftmals nicht in diese Formel einsetzen, sondern intuitiv argumentieren. Dazu benutzt man eine „Formel“ des Typs

$$T_{avg}(n) = \sum_{i=0}^{\infty} \text{Prob}(\text{Ereignis } i) * \text{Rechenzeit bei Eintreten von Ereignis } i.$$

In dieser „Formel“ wird über alle Ereignisse summiert, die sich durch die randomisierten Anweisungen während der Ausführung des Algorithmus ergeben können. Wir verzichten hier auf eine mathematisch präzise Formulierung und betrachten zwei Beispiele.

In dem folgenden Beispiel (Algorithmus 112) ist (neben dem Ergebnis auch) die Laufzeit abhängig vom Ausgang des Würfelexperiments. Die Anzahl der Additionen, die mit der Anweisung $a := a + 1$ ausgeführt werden, ist mit Wahrscheinlichkeit $1/3$ gleich n und mit Wahrscheinlichkeit $2/3$ gleich n^3 . Damit ergibt sich die erwartete Anzahl an Additionen

$$\text{Prob}(x \in \{1, 5\}) * n + \text{Prob}(x \in \{2, 3, 4, 6\}) * n^3 = \frac{1}{3} * n + \frac{2}{3} * n^3.$$

⁷⁴Natürlich *erwarten* wir nicht die Ausgabe „7“ sondern „10“ oder „24“. Der Begriff „Erwartungswert“ steht hier für Mittelwert. Dieser entspricht dem Wert, den wir *erwarten*, wenn wir den Algorithmus sehr oft ausführen und den Mittelwert der Ausgaben bilden.

Algorithmus 112 Beispiel für einen randomisierten Algorithmus mit zufälliger Laufzeit

```
a := 0;  
x := random(1, 2, ..., 6);  
(* werfe einen Würfel *)  
IF x = 1 oder x = 5 THEN  
  FOR i = 1, ..., n DO  
    (* mit W'keit 2/6 = 1/3 *)  
    a := a + 1  
    (* n Additionen *)  
  OD  
ELSE  
  (* W'keit 2/6 = 1/3 *)  
  FOR i = 1, ..., n DO  
    FOR j = 1, ..., n DO  
      FOR k = 1, ..., n DO  
        a := a + 1  
        (* n3 Additionen *)  
      OD  
    OD  
  OD  
FI
```

In diesem Fall stimmt die erwartete Laufzeit größenordnungsmäßig mit der worst-case Laufzeit überein.

Wir betrachten ein weiteres Beispiel (Algorithmus 113), in dem die Anzahl an randomisierten Schritten variabel ist. Hier ist die erwartete Laufzeit konstant, während im worst-case noch nicht einmal Terminierung garantiert ist (also die worst-case Laufzeit ∞ ist). Die erwartete Anzahl an Additionen in der Anweisung $a := a + 1$ ist

Algorithmus 113 Beispiel für einen randomisierten Algorithmus mit zufälliger Laufzeit

```
a := 0;  
REPEAT  
  x := random(head, tail);  
  (* werfe eine Münze *)  
  a := a + 1;  
UNTIL x = head;
```

$$\sum_{i=0}^{\infty} \text{Prob}(\text{genau } i\text{-mal Zahl}) * (i + 1) = \sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i \cdot (i + 1) = 2.$$

Input-Sampling. Es gibt unterschiedliche Arten, in denen das Konzept der Randomisierung eingesetzt werden kann. Die einfachste Art ist das sogenannte *Input-Sampling*, das sich wie folgt motivieren lässt. Einige deterministische Algorithmen haben ein sehr viel besseres durchschnittliche Verhalten als das worst-case Verhalten (z.B. Quicksort, Hashing). Die Durchschnittsanalyse basiert in vielen Fällen auf der idealisierten Annahme, daß die Eingabewerte gleichverteilt sind. Beispielsweise wird in der Durchschnittsanalyse

von Quicksort davon ausgegangen, daß jede Sortierung $x_{i_1} < \dots < x_{i_n}$ der Eingabewerte x_1, \dots, x_n gleichwahrscheinlich ist. Anstelle auf die Gleichverteilung der Eingabewerte zu vertrauen, können in randomisierten Algorithmen Zufallsfunktionen verwendet werden, die eine Gleichverteilung garantieren.⁷⁵ Tatsächlich sind die *erwarteten Kosten* für die randomisierte Version von Quicksort (in der das Pivotelement jeweils zufällig gewählt wird) $\Theta(n \log n)$. Man beachte, daß hierzu keinerlei stochastischen Annahmen über die Eingabefolge x_1, \dots, x_n gemacht werden müssen. Die erwartete Laufzeit der

Algorithmus 114 Randomisiertes Quicksort (Grobschema)

(*Eingabe: Zahlenmenge $\{x_1, x_2, \dots, x_n\}$ (wobei x_1, \dots, x_n p.v. sind) *)

IF $n = 0$ THEN

gib die leere Folge als sortierte Zahlenfolge zurück

ELSE

IF $n = 1$ **THEN**

gib x_1 als sortierte Zahlenfolge zurück

ELSE

$x := \text{random}(x_1, \dots, x_n);$ (* das Pivotelement x wird zufällig gewählt *)

(* Zerlegung in zwei Teilmengen A und B (durch $n - 1$ wesentliche Vergleiche) *)

$$A := \{x_i : x_i < x\}; \quad B := \{x_i : x_i > x\};$$

sortiere A und B mit randomisiertem Quicksort;

setze die sortierten Teilfolgen zusammen und gib die sortierte Gesamtfolge zurück;

FI [View Details](#) [Edit](#) [Delete](#)

I

1

randomisierten Version von Quicksort ist durch die Rekurrenz

$$T_{avg}(n) = \sum_{i=1}^n Prob(\text{Pivotelement ist das } i\text{-kleinste Element}) * t_i(n),$$

$$t_i(n) = T_{avg}(i-1) + T_{avg}(n-i) + n - 1$$

bestimmt. Dabei nehmen wir $n \geq 2$ an. Weiter setzen wir $T_{avg}(0) = T_{avg}(1) = 0$. $T_{avg}(n)$ gibt die erwartete Anzahl an wesentlichen Vergleichen an, die benötigt werden, um eine n -elementige Eingabemenge in die Teilmengen A und B zu zerlegen.⁷⁶ Wir

⁷⁵ Das ist das nicht die volle Wahrheit. Tatsächlich arbeitet man mit sogenannte Pseudozufallsfunktionen, die von echten Zufallsfunktionen nur schwer zu unterscheiden sind.

⁷⁶Als wesentlichen Vergleich bezeichnen wir eine Operation, die für jedes Element $x_i \neq x$ entscheidet, ob x_i zu A oder B hinzugefügt wird. Tatsächlich wird man zwischen n und $2n$ Vergleiche benötigen, in denen man der Reihe nach alle Elemente x_i mit x vergleicht. Etwa gemäß folgender Abfragen: Ist $x_i = x$, so wird x_i ignoriert. Ist $x_i < x$, so wird x_i zu A hinzugefügt. Andernfalls wird x_i zu B hinzugefügt.

erhalten die Rekurrenz

$$T_{avg}(n) = n - 1 + \frac{1}{n} \sum_{i=1}^n (T_{avg}(i-1) + T_{avg}(n-i)),$$

die wir wie in Abschnitt 2.1.3 (mit Hilfe einer Integralabschätzung der Partialsummen der harmonischen Reihe) lösen können.

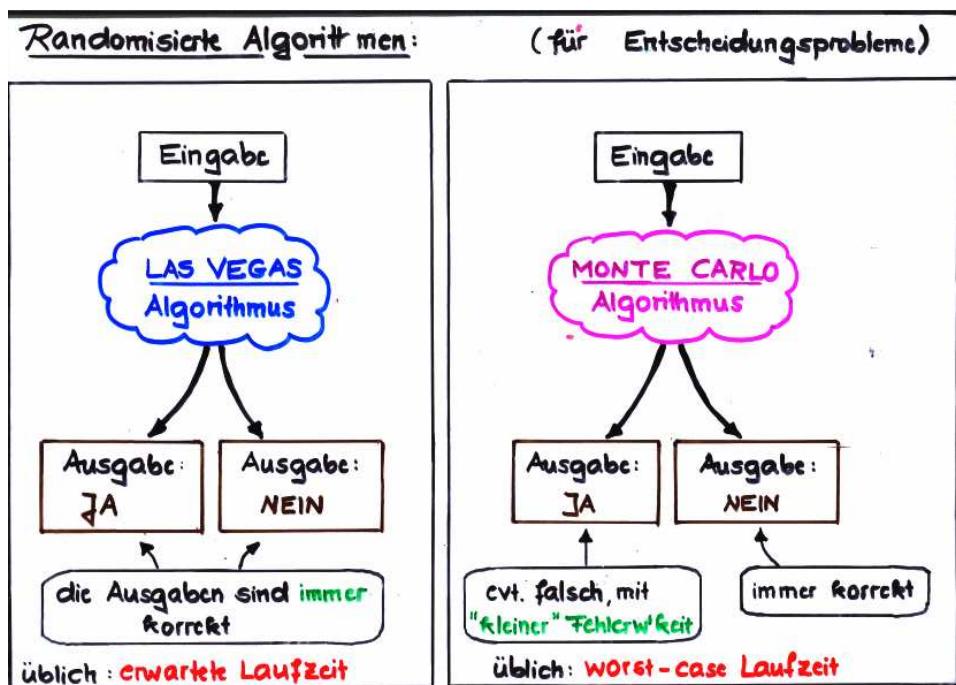
In den Abschnitten 10.2 und 10.3 erläutern wir zwei weitere Grundprinzipien für randomisierte Algorithmen: die sogenannte *Fingerprint-Technik* (am Beispiel des Mustererkennungs-Algorithmus von Karp & Rabin) und das Prinzip der *randomisierten Suche* (am Beispiel des Primzahltests von Miller & Rabin).

10.1 MONTE CARLO und LAS VEGAS Algorithmen

Man unterscheidet zwei Arten von randomisierten Algorithmen.

- LAS VEGAS Algorithmen geben immer die korrekte Antwort (d.h. sind total korrekt im Sinne der in Abschnitt 1.1 gegebenen Definition).
- Für MONTE CARLO Algorithmen sind falsche Antworten möglich; jedoch wird vorausgesetzt, daß die Fehlerwahrscheinlichkeit „hinreichend klein“ ist.

Welche Fehlerwahrscheinlichkeit als genügend klein anzusehen ist, ist selbstverständlich Geschmackssache. Zunächst erscheint es irritierend, daß man einen Algorithmus als korrekt ansieht, selbst wenn er ein falsches Ergebnis liefern kann. Man mache sich jedoch klar, daß wir stets mit Wahrscheinlichkeit für das Auftreten eines Hardwarefehlers leben müssen und daß diese in vielen Fällen weit über der Fehlerwahrscheinlichkeit eines randomisierten Verfahrens liegt.



Während für MONTE CARLO Algorithmen die Effizienz meist an der Laufzeit im schlimmsten Fall gemessen wird, ist für LAS VEGAS Algorithmen die erwartete Laufzeit das übliche Kriterium für die „Güte“ der Ausführungszeit. Entsprechendes gilt für den Platzbedarf. Für Ja/Nein-Probleme unterscheidet man zwischen MONTE CARLO Algorithmen mit *einseitigem Fehler* (bei denen die Antwort „Nein“ immer korrekt ist, während die Antwort „Ja“ falsch sein kann, oder umgekehrt) und MONTE CARLO Algorithmen mit *beidseitigem Fehler* (bei denen sowohl die Antwort „Ja“ als auch die Antwort „Nein“ falsch sein kann).⁷⁷ Liegt ein MONTE CARLO Algorithmus mit einseitigem Fehler und Fehlerwahrscheinlichkeit < 1 vor, dessen Fehlerwahrscheinlichkeit als nicht annehmbar groß angesehen wird, dann kann man durch mehrfaches Ausführen des Algorithmus eine beliebig kleine Fehlerwahrscheinlichkeit garantieren.

10.2 Mustererkennung: der Algorithmus von Karp & Rabin

Der Algorithmus von Karp & Rabin zur Mustererkennung ist eine Alternative zu den in Abschnitt 7 behandelten deterministischen Mustererkennungsalgorithmen. Das Verfahren von Karp & Rabin hat (unter gewissen Voraussetzungen) lineare Laufzeit; es ist jedoch konzeptionell sehr viel einfacher als die deterministischen Linearzeit-Algorithmen (da es keine Hilfsfunktionen für das Muster oder den Text benötigt).⁷⁸

Die Problemstellung ist wie in Abschnitt 7. Gegeben ist ein Text $T = t_1 \dots t_n$ und ein Muster $M = m_1 \dots m_k$, wobei $n > k$ und $t_1, \dots, t_n, m_1, \dots, m_k$ Zeichen eines Alphabets Σ sind. Gefragt ist, ob $T[i] = M$ für einen Index i , wobei $T[i] = t_i t_{i+1} \dots t_{i+k-1}$. Zur Vereinfachung nehmen wir an, daß das Alphabet $\Sigma = \{0, 1, \dots, 9\}$ zugrundeliegt. Dies ermöglicht es, die Wörter $T[i]$ und M als Dezimalzahlen (mit jeweils k Dezimalstellen und eventuellen führenden Nullen) aufzufassen:

$$T[i] = \sum_{j=0}^{k-1} 10^{k-1-j} \cdot t_{i+j}, \quad M = \sum_{j=0}^{k-1} 10^{k-1-j} \cdot m_{j+1}.$$

Liegt ein anderes Alphabet Σ der Kardinalität d vor, dann kann man die Elemente von Σ mit den Zahlen $0, 1, \dots, d - 1$ durchnumerieren und mit dem Zahlensystem zur Basis d (anstelle des Dezimalsystems) arbeiten.

Der Algorithmus von Karp & Rabin basiert auf der Fingerprint-Technik. Grob gesprochen werden lediglich zufällige „Fingerabdrücke“ des Musters und der Textfragmente $T[i]$ verglichen. Wir stellen zuerst einen MONTE CARLO Algorithmus vor, den wir dann zu einem LAS VEGAS Verfahren modifizieren.

Die MONTE CARLO Methode ist eine einfache Variation des naiven Mustererkennungsalgorithmen, der $T[i]$ und M für alle Textpositionen $i \in \{1, \dots, n - k + 1\}$ vergleicht. Anstelle $T[i]$ und M zeichenweise zu vergleichen, verwenden wir eine zufällig

⁷⁷Auf der Folie ist das Schema für MONTE CARLO Algorithmen mit einseitigem Fehler skizziert.

⁷⁸Leider betrifft das nur die algorithmische Einfachheit; nicht die Analyse.

gewählte Primzahl p und vergleichen die Zahlen $F_p(T[i])$ und $F_p(M)$ für die zugehörige Fingerprint-Funktion

$$F_p : \mathbb{N} \rightarrow \{0, 1, \dots, p-1\}, \quad F_p(x) = x \bmod p.$$

Ist z.B. $T = 34217$ und $M = 511$ und wird die Zahl $p = 3$ gewählt, dann ist

$$F_p(M) = 511 \bmod 3 = 1 = 421 \bmod p = F_p(T[2]).$$

Der Algorithmus würde also fälschlicherweise die Antwort „Ja“ liefern. Wird jedoch $p = 17$ gewählt, dann ist

$$F_p(M) = 1 \text{ und } F_p(T[1]) = 2, F_p(T[2]) = F_p(T[3]) = 13.$$

Also liefert der Algorithmus die korrekte Antwort „Nein“. Die Kosten für den Zah-

Algorithmus 115 Mustererkennung: das Verfahren von Karp & Rabin (Grobschema)

wähle zufällig eine Primzahl p ;

$i := 1$;

REPEAT

(* Vergleiche die Fingerprints von $T[i]$ und M *)

IF $F_p(T[i]) = F_p(M)$ **THEN**

 return „Ja. M kommt (vermutlich) in T vor.“

ELSE

$i := i + 1$;

FI

UNTIL $i > n - k + 1$;

Return „Nein. M ist nicht in T enthalten.“.

lenvergleich können als konstant angenommen werden, falls p „hinreichend klein“ ist, d.h. mindestens $\lceil \log p \rceil$ Bits zur Darstellung natürlicher Zahlen zur Verfügung stehen (uniformes Kostenmaß).

Berechnung der Werte $F_p(T[i])$ und $F_p(M)$: Wir gehen davon aus, daß $T[i]$ und M zunächst nur als Ziffernfolgen vorliegen. Die zugehörige Dezimalzahl ergibt sich durch Aufsummieren in $\Theta(k)$ Schritten. Diese berechnen wir für M und $T[1]$. Damit ergeben sich auch die Zahlen $F_p(M)$ und $F_p(T[1])$ in Zeit $\Theta(k)$. Die Berechnung von $F_p(T[i+1])$ kann mit Hilfe der Zahl $F_p(T[i])$ in konstanter Zeit durchgeführt werden. Hierzu verwenden wir folgende Beobachtung:

$$\begin{aligned} T[i+1] &= (t_{i+1} \dots t_{i+k-1} t_{i+k})_{10} \\ &= 10 \cdot (t_{i+1} \dots t_{i+k-1})_{10} + t_{i+k} \\ &= 10 \cdot (T[i] - t_i \cdot 10^{k-1}) + t_{i+k}. \end{aligned}$$

Die Schreibweise $(\dots)_{10}$ soll andeuten, daß wir die Ziffernfolge \dots als Dezimalzahl auffassen. Wir benutzen die Rechengesetze

$$F_p(x + y) = (F_p(x) + F_p(y)) \bmod p,$$

$$F_p(xy) = (F_p(x) \cdot F_p(y)) \bmod p$$

und erhalten

$$F_p(T[i+1]) = (10 \cdot (F_p(T[i]) - t_i \cdot F_p(10^{k-1})) + t_{i+k}) \bmod p.$$

Beispielsweise ist für $T = 34217$ und $k = 3$:

$$T[2] = 421 \text{ und } T[3] = 217 = 10 \cdot (421 - 4 \cdot 10^2) + 7.$$

Algorithmus 116 MONTE CARLO Algorithmus zur Mustererkennung

```

wähle zufällig eine Primzahl  $p$ ;  

 $Q := 10^{k-1} \bmod p$ ;  

 $x := M \bmod p$ ; (*  $x$  ist Fingerprint von  $M$  *)  

 $y := T[1] \bmod p$ ; (*  $y$  ist Fingerprint von  $T[i]$  *)  

 $i := 1$ ;  

WHILE  $i \leq n - k + 1$  DO  

  (* Vergleiche die Fingerprints von  $T[i]$  und  $M$  *)  

  IF  $x = y$  THEN  

    return „Ja.  $M$  kommt (vermutlich) im Text  $T$  vor.“  

  FI  

   $i := i + 1$ ;  

  (* berechne den Fingerprint  $F_p(T[i])$  *)  

  IF  $i \leq n - k + 1$  THEN  

     $y := (10 * (y - t_{i-1} * Q) + t_{i+k-1}) \bmod p$ ;  

  FI  

OD  

Return „Nein.  $M$  ist nicht in  $T$  enthalten“.

```

Insgesamt ergibt sich die Laufzeit $\mathcal{O}(n + k)$ für die skizzierte Vorgehensweise. Dabei wird – wie zuvor erläutert – vorausgesetzt, daß der Zahlenvergleich von $F_p(T[i])$ und $F_p(M)$ sowie das Rechnen mit k -steligen Dezimalzahlen nur konstante Kosten verursacht.

Fehlerwahrscheinlichkeit. Wir schätzen nun die Wahrscheinlichkeit ab, daß das Verfahren fälschlicherweise die Antwort „Ja“ gibt. Seien $x, y \in \mathbb{N}$, so daß $x \neq y$ und $x, y < 10^k$. (Zur Erinnerung: $k = |M|$ ist die Länge des Musters.) Dann gilt:

$$F_p(x) = F_p(y) \text{ gdw. } |x - y| \equiv 0 \pmod{p}$$

gdw. p ist ein Primfaktor von $|x - y|$.

Sei r die Anzahl an Bits, die zur Darstellung von p (im Binärsystem) zur Verfügung stehen. D.h. die Primzahl p wird zufällig aus dem Zahlenbereich $\{2, 3, \dots, 2^r - 1\}$ gewählt. Weiter sei $R = 2^r - 1$ und $\pi(R)$ die Anzahl an Primzahlen, die $\leq R$ sind. Somit gilt folgende Formel:

$$\text{Prob} [F_p(x) = F_p(y) \mid p \text{ zufällige Primzahl } \leq R] = \frac{\text{Anzahl Primfaktoren von } |x - y|}{\pi(R)}.$$

Aus der Zahlentheorie ist bekannt, daß es „sehr viele“ Primzahlen gibt. Wir zitieren das fundamentale Ergebnis ohne Beweis.

Satz 10.2.1 (Primzahl-Satz).

$$\lim_{R \rightarrow \infty} \frac{\pi(R)}{R/\ln R} = 1.$$

Dabei steht $\ln R$ für den natürlichen Logarithmus, der als Basis die Eulersche Zahl $e \approx 2,7\dots$ hat. Insbesondere ist

$$\pi(R) = \Theta\left(\frac{R}{\ln R}\right) = \Theta\left(\frac{R}{\log R}\right).$$

Weiter gilt $|x - y| < 10^k = 2^{k \cdot \log(10)}$. Daher ist die Anzahl der Primfaktoren von $|x - y|$ beschränkt durch $k \cdot \log 10$. (Beachte: Jede Primzahl ist ≥ 2 .) Wir erhalten folgende Abschätzung:

$$\begin{aligned} \text{Prob} [F_p(x) = F_p(y) \mid p \text{ zufällige Primzahl } \leq R] \\ &< \frac{k \cdot \log(10)}{\pi(R)} = \Theta\left(\frac{k \cdot \log R}{R}\right). \end{aligned}$$

Verbesserung der Fehlerwahrscheinlichkeit. Die Fehlerwahrscheinlichkeit kann durch mehrfaches Anwenden des Verfahrens verbessert werden. Wir nehmen an, daß die gewünschte Fehlerwahrscheinlichkeit höchstens ε ist (für eine „kleine“ positive reelle Zahl ε). Weiter nehmen wir an, daß R (bzw. r) groß genug gewählt war, so daß $(k \log R)/R < 1$ ist. Wir wählen nun eine ganze Zahl $L \geq 1$, so daß

$$\left(\frac{k \cdot \log R}{R}\right)^L < \varepsilon$$

und führen das Verfahren mit L zufällig gewählten Primzahlen aus dem Bereich $\{2, 3, \dots, R\}$ durch.⁷⁹ Diese Vorgehensweise ist in Algorithmus 117 skizziert. Die Fehlerwahrscheinlichkeit ist dann $< \varepsilon$. Die worst-case Laufzeit ist $\mathcal{O}(L \cdot (n + k))$.

Algorithmus 117 MONTE CARLO Algorithmus mit verbesserter Fehlerw'keit

```

 $l := 1;$ 
(*) wähle zufällig eine Primzahl  $p$ ;
 $Q := 10^{k-1} \bmod p; \quad x := M \bmod p; \quad y := T[1] \bmod p; \quad i := 1;$ 
WHILE  $i \leq n - k + 1$  DO
    (* Vergleiche die Fingerprints von  $T[i]$  und  $M$  *)
    IF  $x = y$  THEN
        IF  $l = L$  THEN
            return „Ja.  $M$  kommt (vermutlich) im Text  $T$  vor.“
        ELSE
             $l := l + 1;$  goto (*);                                (* wiederhole das Verfahren *)
        FI
    FI
     $i := i + 1;$ 
    IF  $i \leq n - k + 1$  THEN
         $y := (10 * (y - t_{i-1} * Q) + t_{i+k-1}) \bmod p;$ 
    FI
OD
Return „Nein.  $M$  kommt nicht in  $T$  vor.“

```

Die LAS VEGAS Version. Wir modifizieren nun das MONTE CARLO Verfahren (mit nur einer zufällig gewählten Primzahl, also $L = 1$), so daß ein fehlerfreier LAS VEGAS Algorithmus entsteht (siehe Algorithmus 118). Dazu verändern wir den Algorithmus einfach dahingehend, daß – sobald $F_p(T[i]) = F_p(M)$ – der zeichenweise Vergleich von $T[i]$ und M durchgeführt wird. Die erwartete Laufzeit der LAS VEGAS Methode ist durch die Kostenfunktion

$$T(n, k) = \mathcal{O}(n + k) + \mathcal{O}\left(k \cdot \frac{nk \log R}{R}\right)$$

gegeben. Diese erklärt sich wie folgt. Der zeichenweise Vergleich von $T[i]$ und M erfordert $\mathcal{O}(k)$ Zeiteinheiten. Der erste Summand $\mathcal{O}(n + k)$ steht für sämtliche Schritte, die bereits bei der MONTE CARLO Version auszuführen sind und dem höchstens einmal ausgeführten erfolgreichen Vergleich von $T[i]$ und M . Die erwartete Anzahl an Textpositionen i , für die $T[i]$ und M zeichenweise *ohne Erfolg* verglichen werden, ist $\mathcal{O}(n(k \log R)/R)$. Diese ergibt sich aus den obigen Überlegungen zur Fehlerwahrscheinlichkeit der MONTE CARLO Version. Für feste Textposition i mit $T[i] \neq M$ ist die Wahrscheinlichkeit

⁷⁹Da es sich hier um einen MONTE CARLO Algorithmus mit einseitigem Fehler handelt, genügt es selbstverständlich, das Verfahren nur für die Antwort „Ja“ wiederholt auszuführen.

Algorithmus 118 LAS VEGAS Algorithmus zur Mustererkennung

wähle zufällig eine Primzahl p ;
 $Q := 10^{k-1} \bmod p$; $x := M \bmod p$; $y := T[1] \bmod p$; $i := 1$;

WHILE $i \leq n - k + 1$ **DO**

(* Vergleiche die Fingerprints von $T[i]$ und M *)

IF $x = y$ **THEN**

(* Vergleiche M und $T[i]$ zeichenweise *)

IF $M = T[i]$ **THEN**

return „Ja. M kommt in T vor.“

FI

FI

$i := i + 1$;

IF $i \leq n - k + 1$ **THEN**

$y := (10 * (y - t_{i-1} * Q) + t_{i+k-1}) \bmod p$;

FI

OD

Return „Nein. M kommt nicht in T vor.“

für $F_p(T[i]) = F_p(M)$ beschränkt durch $\mathcal{O}((k \log R)/R)$ (s.o.). Aufsummieren über alle Textpositionen i mit $T[i] \neq M$ ergibt die obere Schranke $\mathcal{O}(n(k \log R)/R)$.

Wir fassen die Ergebnisse in dem folgenden Satz zusammen, wobei wir die Größenordnung von R so wählen, daß sich lineare Laufzeit ergibt.

Satz 10.2.2. Bezeichnungen wie zuvor. Für $R = \Theta(n^2 k \log(n^2 k))$ gilt:

- Die MONTE CARLO Version (Algorithmus 116) hat die Fehlerwahrscheinlichkeit $\mathcal{O}(1/n)$ und lineare worst-case Laufzeit $\mathcal{O}(n + k)$.
- Die LAS VEGAS Version hat die worst-case Laufzeit $\mathcal{O}(n \cdot k)$ und die erwartete Laufzeit $\mathcal{O}(n + k)$.

Dabei vernachlässigen wir die Kosten (und Fehlerwahrscheinlichkeit) für das Generieren zufälliger Primzahlen. Wie zufällige Primzahlen generiert werden können, wird in Abschnitt 10.3 erläutert. Wir weisen nochmals darauf hin, daß obige Kostenanalyse voraussetzt, daß der Vergleich von Zahlen $\leq R$ (also Zahlen mit $\mathcal{O}(\log n + \log k)$ Binärstellen) als elementare Operation (die in konstanter Zeit ausgeführt werden kann) angesehen wird. Für enorm großes n und k ist dies nicht realistisch. In diesem Fall sind zusätzliche Kosten für den Zahlenvergleich zu berücksichtigen.

10.3 Primzahltest: der Algorithmus von Miller & Rabin

Viele randomisierte Verfahren (z.B. der soeben behandelte Mustererkennungsalgorithmus, oder diverse Verfahren, die in der Kryptographie eingesetzt werden) benutzen einen Generator zum Erzeugen zufälliger Primzahlen. Das Grundschema eines solchen Zufalls-Primzahlgenerators ist wie folgt. Man legt eine Zahl $r \in \mathbb{N}$ fest, so daß eine r -stellige Zahl

$p = \langle x_0, x_1, \dots, x_{r-1} \rangle$ (Binärzahldarstellung mit eventuellen führenden Nullen) ermittelt wird, deren Ziffern x_i zufällig gewählt werden (Münzwurf). Es wird geprüft, ob p eine Primzahl ist. Wenn nein, dann wird das Verfahren wiederholt. Siehe Algorithmus 119. Aus dem Primzahl-Satz (Satz 7.3.1) ergibt sich, daß im Mittel nach nur $\Theta(r)$ Versuchen eine Primzahl gefunden wird. Beachte, dass aus dem Primzahlsatz folgt, daß mit einer Wahrscheinlichkeit von ca. $1/\ln R$ die zufällig gewählte Zahl $x \in \{1, 2, \dots, R\}$ eine Primzahl ist. Daher wird im Mittel spätestens nach ca. $\ln R = \Theta(r)$ Iterationen eine Primzahl gefunden.

Algorithmus 119 Generieren zufälliger Primzahlen mit maximal r Bits

REPEAT

wähle zufällig r Bits x_0, x_1, \dots, x_{r-1} ; (* werfe r -mal eine faire Münze *)
 $x := x_0 + 2 * x_1 + 4 * x_2 + \dots + 2^{r-2} * x_{r-2} + 2^{r-1} * x_{r-1}$;
Prüfe, ob x eine Primzahl ist;

UNTIL x ist eine Primzahl.

Gib x aus.

Primzahltest. Gegeben ist eine ganze Zahl $p \geq 3$, für die geprüft werden soll, ob p eine Primzahl ist. In der Praxis (insbesondere für die in der Kryptographie eingesetzten Methoden) werden häufig sehr große zufällige Primzahlen (mit ca. 100 Dezimalstellen) benötigt, für die nur die maximale Stellenzahl im Binärsystem festgelegt ist. Die Laufzeit des Primzahltests wird daher üblicherweise an der Stellenzahl von p als Binärzahl gemessen.⁸⁰ Im Folgenden sei r die Länge der Binärzahldarstellung von p .

Die meisten Primzahlalgorithmen versuchen, die Primalität von p mit Hilfe eines *Zeugen* für die Zusammengesetztheit von p zu widerlegen. Beispielsweise das naive Verfahren, das für alle ganzen Zahlen x mit $2 \leq x < p$ prüft, ob x ein Teiler von p ist, sucht nach einem Teiler von p (also einer Zahl, die die Zusammengesetztheit von p „bezeugt“). Die Laufzeit des naiven Verfahrens ist $\Theta(p) = \Theta(2^{\log p})$, also exponentiell in der Darstellungsgröße von p .⁸¹

Randomisierte Primzahltests. Wir stellen hier den MONTE CARLO Primzahl-Algorithmus von Miller & Rabin vor. Dieser (sowie fast alle anderen MONTE CARLO Primzahltests) benutzt ein hinreichendes Kriterium für die Nichtprimalität von p , das durch ein Prädikat

$$Witness : \mathbb{N} \times \mathbb{N} \rightarrow \{true, false\}$$

mit folgenden Eigenschaften beschrieben wird.

(I) Aus $Witness(x, p) = true$ folgt $2 \leq x < p$ und p ist *keine* Primzahl.

⁸⁰Größenordnungsmäßig ist es völlig irrelevant, ob das Dezimal- oder das Binärsystem oder ein anderes Zahlensystem verwendet wird.

⁸¹Die Laufzeit des naiven Verfahrens lässt sich zwar verbessern, in dem man nur alle Zahlen x mit $2 \leq x \leq \sqrt{p}$ betrachtet oder in dem man sämtliche ganzzähligen Vielfachen bereits betrachteter Zahlen ignoriert; dennoch ergibt sich dadurch kein Polynomialzeitalgorithmus.

(II) Ist p keine Primzahl und $p \geq 3$, dann sind mindestens die Hälfte aller Zahlen $x \in \{2, 3, \dots, p-1\}$ Zeugen für die Zusammengesetztheit von p , d.h. es gilt:

$$\left| \{x \in \{2, 3, \dots, p-1\} : \text{Witness}(x, p) = \text{true}\} \right| \geq \frac{p-1}{2}.$$

Die einfachste Form eines Zeugen x für die Zusammengesetztheit einer Zahl p ist die Bedingung „ x ist Teiler von p “. Diese erfüllt allerdings nicht die Forderung (II).

Jedes Zeugen-Prädikat mit den Eigenschaften (I) und (II) induziert einen randomisierten Primzahltest, der auf folgenden Schema beruht. Wir wählen zufällig eine Zahl x aus dem Bereich $\{2, 3, \dots, p-1\}$ und prüfen, ob $\text{Witness}(x, p) = 1$. Wenn ja, dann ist (wegen (I)) p keine Primzahl. Andernfalls wird das Verfahren so lange wiederholt bis eine Maximalzahl L an Versuchen, einen Zeugen für die Nichtprimärtät von p zu finden, erreicht ist. Bedingung (II) stellt sicher, daß – sofern p keine Primzahl ist – in jedem Versuch mit einer Wahrscheinlichkeit $\geq 1/2$, die zufällig gewählte Zahl x ein Zeuge für die Nichtprimärtät von p ist. Die Wahrscheinlichkeit, in L Versuchen stets ein Element x mit $\text{Witness}(x, p) = 0$ zu „erwischen“, ist also $\leq (1/2)^L$. Dieses Schema ist in Algorithmus 120 skizziert.

Algorithmus 120 Primzahltest: Grundschema der MONTE CARLO Algorithmen

(* Eingabe: ganze Zahl p mit $p \geq 3$ *)

FOR $l = 1, 2, \dots, L$ **DO**

wähle zufällig eine ganze Zahl x mit $2 \leq x < p$;

IF $\text{Witness}(x, p)$ **THEN**

(* x ist Zeuge für die Nichtprimärtät von p *)

return „Nein. p ist keine Primzahl.“

FI

OD

Return „Ja. p ist (vermutlich) eine Primzahl.“

(* Fehlerw'keit $\leq (1/2)^L$ *)

Im Folgenden erläutern wir das Zeugenkriterium, das zunächst von Miller (1976) entdeckt wurde. Rabin (1981) hat dieses Zeugenkriterium zur Formulierung eines randomisierten Primzahltests vorgeschlagen.

Einschub: Rechnen mit Kongruenzen. Wir benötigen zunächst einige Grundlagen über das Rechnen mit Kongruenzen. Seien x, y, p ganze Zahlen, $p \geq 2$. $x \bmod p$ bezeichnet die eindeutig bestimmte ganze Zahl r mit $0 \leq r < p$, für die $x - r$ ein ganzzahliges Vielfaches von p ist. x und y heißen *kongruent modulo p*, i.Z.

$$y \equiv x \bmod p,$$

falls $y \bmod p = x \bmod p$. Eine äquivalente Formulierung ist $y \equiv x \bmod p$ genau dann, wenn p ein Teiler von $x - y$ ist. Bereits in Abschnitt 10.2 haben wir die Rechenregeln

$$\begin{aligned} (x + y) \bmod p &= (x \bmod p + y \bmod p) \bmod p \\ (x * y) \bmod p &= (x \bmod p * y \bmod p) \bmod p \end{aligned}$$

benutzt. Aus diesen folgt: $y \equiv x \pmod{p}$ gilt genau dann, wenn p ein Teiler von $y - x$ ist (d.h. wenn $y - x = c \cdot p$ für eine ganze Zahl c).

Satz 10.3.1 (Satz von Fermat). Ist p eine Primzahl und $x \in \{1, \dots, p-1\}$, dann gilt:

$$x^{p-1} \equiv 1 \pmod{p}.$$

Beweis. Sei p eine Primzahl. Wir zeigen zunächst, daß $x^p \equiv x$. Mit der binomischen Formel gilt:

$$\begin{aligned} x^p &= ((x-1)+1)^p = \sum_{j=0}^p \binom{p}{j} (x-1)^j \\ &= 1 + \sum_{j=1}^{p-1} \binom{p}{j} (x-1)^j + (x-1)^p \end{aligned}$$

Da p stets ein Teiler der Binomialkoeffizienten $\binom{p}{j}$, $j = 1, \dots, p-1$, ist, folgt:

$$x^p \equiv ((x-1)^p + 1) \pmod{p}.$$

Durch Induktion nach $x \in \{0, 1, \dots, p-1\}$ kann man zeigen, daß hieraus

$$x^p \equiv x \text{ für } x \in \{0, 1, \dots, p-1\}$$

folgt. Wir haben also gezeigt, daß p ein Teiler von $x^p - x = x(x^{p-1} - 1)$ ist. Da p eine Primzahl ist, ist p Teiler von x oder ein Teiler von $x^{p-1} - 1$. Da wir $x \in \{1, \dots, p-1\}$ voraussetzen, kann p kein Teiler von x sein. Daher ist p ein Teiler von $x^{p-1} - 1$. Hieraus folgt, daß $x^{p-1} \equiv 1 \pmod{p}$. \square

Satz 10.3.1 legt folgendes Zeugenkriterium nahe. Ist $x \in \{2, \dots, p-1\}$, so daß $x^{p-1} \not\equiv 1 \pmod{p}$, dann ist x ein Zeuge für die Zusammengesetztheit von p .

Beispielsweise sind für $p = 8$ alle Werte $x \in \{2, 3, \dots, 7\}$ Zeugen für die Nichtprimalität von p .

x	2	3	4	5	6	7
x^7	128	2.187	16.384	78.125	279.936	823.543
$x^7 \pmod{8}$	0	3	0	5	0	7

Die Umkehrung von Satz 10.3.1 gilt jedoch nicht. Z.B. erhält man für $p = 15$ und $x = 11$:

$$x^{p-1} = 11^{14} = 121^7 = (8 \cdot 15 + 1)^7 \equiv 1 \pmod{15}.$$

Darüberhinaus gibt es Zahlen (die sogenannten *Carmichael Zahlen*)⁸², die keine Primzahlen sind, für die jedoch gilt:

$$x^{p-1} \equiv 1 \pmod{p} \quad \text{für alle } x \in \{2, 3, \dots, p-1\}.$$

⁸²Solche Zahlen kommen extrem selten vor. Die ersten drei Carmichael Zahlen sind 561, 1105, 1729. Insgesamt gibt es nur 255 Carmichael Zahlen, die $\leq 10^9$ sind.

Satz 10.3.1 als alleinige Basis für das Zeugenprädikat $\text{Witness}(\cdot)$ ist nicht ausreichend, da Eigenschaft (II) nicht erfüllt ist. Der folgende Satz liefert ein weiteres notwendiges Kriterium für Primzahlen.

Satz 10.3.2. Ist p eine Primzahl ≥ 3 , dann gilt für alle ganzen Zahlen z :

Aus $z^2 \equiv 1 \pmod p$ folgt $z \equiv 1 \pmod p$ oder $z \equiv -1 \pmod p$.

Das Miller-Rabin-Zeugenprädikat benutzt Satz 10.3.1 und Satz 10.3.2. Dieses ist gegeben durch (siehe Algorithmus 121): $\text{Witness}(x, p) = 1$ genau dann, wenn $x \in \{2, 3, \dots, p-1\}$ und

- entweder $x^{p-1} \not\equiv 1 \pmod p$
- oder es existiert eine Zahl $z \in Q(x, p)$ mit $z^2 \equiv 1 \pmod p$ und $z \not\equiv \pm 1 \pmod p$.

Dabei ist $Q(x, p) = \{x^{\langle b_{r-1}, \dots, b_1, b_0 \rangle} : j = 1, \dots, r-1\}$, wobei

$$p-1 = \langle b_{r-1}, \dots, b_1, b_0 \rangle = \text{Binärzahldarstellung von } p.$$

Also $b_0, \dots, b_{r-1} \in \{0, 1\}$ und $p-1 = b_0 + 2 \cdot b_1 + \dots + 2^{r-1} \cdot b_{r-1}$. Z.B. für $p=11$ ist

$$p-1 = 10 = 8 + 2 = 2^3 + 2^1 = \langle 1, 0, 1, 0 \rangle.$$

Sei x eine ganze Zahl zwischen 2 und 10. Dann ist

$$Q(x, 11) = \{x^{\langle 1, 0, 1 \rangle}, x^{\langle 1, 0 \rangle}, x^{\langle 1 \rangle}\} = \{x^5, x^2, x\}.$$

Algorithmus 121 Das Miller-Rabin Zeugenprädikat $\text{Witness}(x, p)$

```

IF es gibt  $z \in Q(x, p)$  mit  $z^2 \equiv 1 \pmod p$  und  $z \not\equiv \pm 1 \pmod p$  THEN
    return „true“
FI
IF  $x^{p-1} \equiv 1 \pmod p$  THEN
    return „true“
FI
Return „false“.

```

Der folgende Satz zeigt, daß für jede zusammengesetzte Zahl p mindestens die Hälfte aller Elemente $x \in \{2, 3, \dots, p-1\}$ Zeugen für p sind. Wir zitieren dieses Ergebnis ohne Beweis.

Satz 10.3.3. Sei $p \geq 3$ eine ganze Zahl, die keine Primzahl ist. Dann gibt es mindestens $(p-1)/2$ Elemente $x \in \{2, 3, \dots, p-1\}$ mit $\text{Witness}(x, p) = 1$. (ohne Beweis)

Der Primzahltest von Miller & Rabin ist in Algorithmus 122 formuliert.

Algorithmus 122 Primzahltest: Algorithmus von Miller-Rabin

FOR $l = 1, \dots, L$ **DO**

wähle zufällig eine ganze Zahl x mit $2 \leq x < p$;

IF $\text{Witness}(x, p)$ **THEN**

(* x ist Zeuge für die Zusammengesetztheit von p *)

return „Nein. p ist keine Primzahl.“

FI

OD

Return „Ja. p ist (vermutlich) eine Primzahl.“

(* Fehlerw'keit $\leq (1/2)^L$ *)

Berechnung der Potenzen $x^{p-1} \bmod p$. Wir skizzieren nun, wie sich das Miller-Rabin Witness Prädikat effizient berechnen lässt. Hierzu verwenden wir die Methode des *Repeated Squaring*, um die Potenzen modulo p zu berechnen. Sei

$$p - 1 = \sum_{j=0}^{r-1} b_j \cdot 2^j.$$

Dann gilt:

$$x^{p-1} = x^{b_{r-1} \cdot 2^{r-1}} \cdot x^{b_{r-2} \cdot 2^{r-2}} \cdots x^{b_1 \cdot 2} \cdot x^{b_0}.$$

Sei

$$\begin{aligned} d_{r-1} &= x^{b_{r-1}} \\ d_{r-2} &= x^{\langle b_{r-1}, b_{r-2} \rangle} = x^{\langle b_{r-1}, 0 \rangle} \cdot x^{b_{r-2}} = d_{r-1}^2 \cdot x^{b_{r-2}} \\ d_{r-3} &= x^{\langle b_{r-1}, b_{r-2}, b_{r-3} \rangle} = x^{\langle b_{r-1}, b_{r-2}, 0 \rangle} \cdot x^{b_{r-3}} = d_{r-2}^2 \cdot x^{b_{r-3}} \\ &\vdots \\ d_j &= x^{\langle b_{r-1}, b_{r-2}, \dots, b_j \rangle} = d_{j+1}^2 \cdot x^{b_j} \\ &\vdots \\ d_0 &= x^{p-1} = x^{\langle b_{r-1}, \dots, b_1, 0 \rangle} \cdot x^{b_0} = d_1^2 \cdot x^{b_0}. \end{aligned}$$

Entsprechendes gilt, wenn wir modulo p rechnen. Sei $z_{r-1} = x^{b_{r-1}} \bmod p$ und

$$z_j = (z_{j+1}^2 \cdot x^{b_j}) \bmod p, \quad j = r-2, r-3, \dots, 1, 0.$$

Dann gilt $z_j = d_j \bmod p$, $j = 0, 1, \dots, r-1$, und somit $z_0 = x^{p-1} \bmod p$. Wir verwenden die oben angegebene Charakterisierung von z_j mittels z_{j+1} als rekursive Auswertungsvorschrift für $z_{r-1}, z_{r-2}, \dots, z_1, z_0$ (siehe Algorithmus 123). Diese erfordert lediglich $\mathcal{O}(r) = \mathcal{O}(\log p)$ Schritte, in denen jeweils mit Zahlen der Größenordnung von p (also Zahlen der Stelligkeit $\mathcal{O}(r)$) gerechnet wird. Zu dem hat sie den Vorzug, daß die Berechnung der Werte $z^2 \bmod p$ für $z \in Q(x, p)$ in die Berechnung von $x^{p-1} \bmod p$ integriert werden können. Siehe Algorithmus 124. Die Laufzeit der so erhaltenen Auswertungsvorschrift für das Witness-Prädikat ist $\mathcal{O}(r^2) = \mathcal{O}((\log p)^2)$, wenn wir das logarithmische Kostenmaß zugrundelegen. Wir fassen die Ergebnisse zusammen:

Satz 10.3.4. Der MONTE CARLO Algorithmus von Miller-Rabin für den Primzahltest hat die Fehlerwahrscheinlichkeit $\mathcal{O}(1/2^L)$ und die worst-case Laufzeit $\mathcal{O}(L \cdot r^2) = \mathcal{O}(L \cdot (\log p)^2)$ (bzgl. dem logarithmischen Kostenmaß). Dabei ist L die festgelegte maximale Anzahl an Versuchen, einen Zeugen für die Nichtprimärtät von p zu finden, und r die Länge der Binärzahldarstellung von p .

Für die meisten zusammengesetzten Zahlen p ist die tatsächliche Fehlerwahrscheinlichkeit sogar sehr viel kleiner (da die Anzahl an Zeugen weit größer als $(p - 1)/2$ ist). Ein praktischer Erfahrungswert ist, daß man bereits für $L = 3$ hinreichend zuverlässige Ergebnisse bekommt.

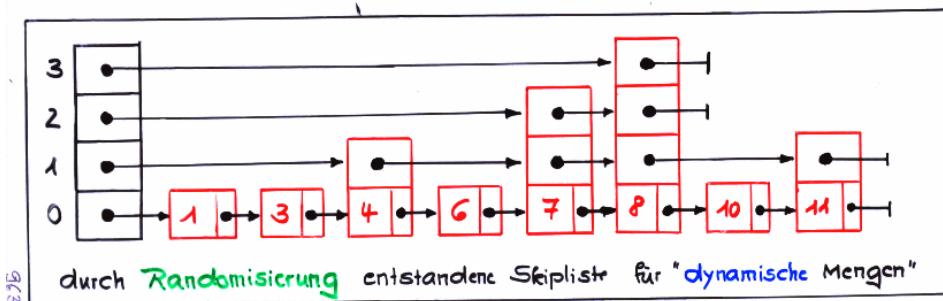
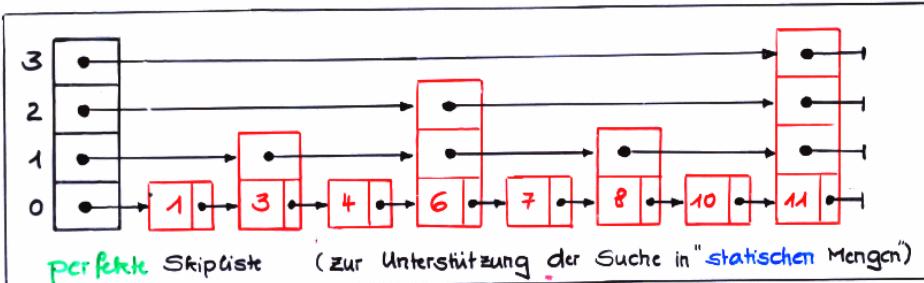
10.4 Randomisierte Datenstrukturen

Das Prinzip der Randomisierung läßt sich ebenfalls auf die zu einem abstrakten Datentyp gehörenden Methoden anwenden.

10.4.1 Skiplisten

Skiplisten sind eine Variante von linearen sortierten Listen, die die Operationen Einfügen, Löschen und Suchen unterstützen. Um die Suchprozedur zu erleichtern werden neben der eigentlichen (wie üblich organisierten) sortierten Liste zusätzliche Verkettungen auf verschiedenen Ebenen verwendet. Die Ebenen sind hierarchisch angeordnet: jedes Listenglied, das in Ebene i vertreten ist, ist auch in den Ebenen $0, 1, \dots, i-1$ vertreten. Die Verkettung auf Ebene 0 entspricht einer gewöhnlichen sortierten Liste. Die Verkettungen der Ebenen $1, 2, \dots$ können als Hilfslisten angesehen werden, in der manche der auf den darunterliegenden Ebenen repräsentierten Elemente miteinander verkettet sind. Unter der *Höhe* eines Listenelements versteht man die Anzahl an Ebenen, in denen das betreffende Listenelement vertreten ist. Im Folgenden sprechen wir von der i -ten Liste (oder Liste i) und meinen damit diejenige Hilfsliste, die sich aus den Verkettungen der i -ten Ebene ergibt. Die folgende Skizze sagt vermutlich mehr als tausend Worte:

Skip - Listen:



Zur Vereinfachung nehmen wir an, daß $n = 2^k$ Datensätze dargestellt werden sollen.

Perfekte Skiplisten. In perfekten Skiplisten verkettet Liste i genau die Elemente, die an den Positionen $2^i, 2 \cdot 2^i, 3 \cdot 2^i, \dots, 2^{k-i} \cdot 2^i$ stehen, $i = 1, \dots, k$. Die Suche in einer perfekten Skipliste nach einem Schlüsselwert x kann in Zeit $\mathcal{O}(\log n)$ durchgeführt werden. Sei x der gesuchte Schlüsselwert. Wir starten die Suche mit dem Listenkopf der der k -ten Ebene und suchen nun x (bzw. den zu x gehörenden Datensatz), in dem wir sukzessive zur Ebene 0 „hinabsteigen“. Um von Ebene i zu Ebene $i - 1$ zu gelangen, bestimmen wir dasjenige Listenglied v von Liste i , dessen Schlüsselwert $key(v)$ gerade noch kleiner oder gleich x ist (während für das auf der gerade aktuellen Ebene i nachfolgende Listenglied $Next_{i-1}(v)$ der Schlüsselwert $> x$ ist). Ist $key(v) = x$, dann kann die Suche beendet werden. Der zu x gehörende Datensatz ist auf Ebene 0 zu finden. Andernfalls (d.h. wenn $key(v) < x$) gehen wir entweder zur Ebene $i - 2$ über oder – falls v keinen Nachfolger hat oder $i = 1$ – beenden die Suche mit dem Ergebnis, daß es keinen Datensatz mit Schlüsselwert x in der Skipliste gibt. Man beachte, daß beim Übergang von Ebene i zu Ebene $i - 1$ ein Vergleich ausreichend ist. Lediglich die Schlüsselwerte von v mit dem Nachfolger von v der ($i - 1$)-ten Ebene müssen verglichen werden. Diese Ideen sind in Algorithmus 125 zusammengefaßt. Die durch die Suche verursachten Kosten sind somit linear in der Höhe der Skipliste; also logarithmisch in der Anzahl n an gespeicherten Datensätzen. Einfügen und Löschen in perfekten Skiplisten erfordern jedoch sehr hohen Restrukturierungsaufwand. Sie sind daher nur für *statische* Datenmengen, für die kaum Einfüge- und Löschoperationen zu erwarten sind, empfehlenswert.

Algorithmus 125 Suche nach Schlüsselwert x in einer perfekten Skipiste

$i := k$; (* wir starten mit der obersten Ebene $i = k$ *)
 $v :=$ Kopfelement der Skipiste; (* wir nehmen den Pseudoschlüsselwert $key(v) = -\infty$ an *)

WHILE $i > 0$ und $key(v) \neq x$ **DO**

$w := Next_i(v)$; (* w ist Nachfolger von v in Liste i *)

IF $w \neq \perp$ and $key(w) \leq x$ **THEN**

$v := w$

FI

(* v ist „letztes“ Listenglied von Ebene i mit $key(v) \leq x$ *)

IF $key(v) = x$ **THEN**

gib den in Listenglied v gespeicherten Datensatz aus und halte an

ELSE

$i := i - 1$ (* gehe zur nächsttieferen Ebene *)

FI

OD (* entweder ist Ebene 0 erreicht oder x gefunden *)

IF $key(v) = x$ **THEN**

gib den in Listenglied v gespeicherten Datensatz aus

ELSE

gib „Schlüsselwert x nicht vorhanden“ aus.

FI

Randomisierte Skipisten. In randomisierten Skipisten verzichtet man auf die Forderung, daß genau die Elemente $j \cdot 2^i$, $j = 1, \dots, 2^{k-i}$ in Liste i dargestellt werden. Statt dessen wird beim Einfügen eines Elements *zufällig* die Höhe des neuen Listenglieds gewählt. Zur Vereinfachung lassen wir beliebige natürliche Zahlen als Werte für die Höhe h zu. In der Praxis wird man freilich mit einer Höhenbeschränkung arbeiten. Um der Struktur perfekter Skipisten nahezukommen, wählt man eine Zufallsfunktion $random(0, 1, \dots)$, so daß

$$Prob[random(0, 1, \dots) = h] = \frac{1}{2^{h+1}}, \quad h = 0, 1, 2, \dots$$

Eine derartige Zufallsfunktion kann durch das Experiment

- Werfe solange eine Münze bis „Zahl“ oben liegt.
- Gib $h = \text{Anzahl der Münzwürfe minus 1}$ zurück.

simuliert werden.

Erwarteter Platzbedarf. Wir schätzen den Platzbedarf mit der Anzahl an benötigten Zeigern ab. Im Folgenden sei n die Anzahl an Elementen (Schlüsselwerten), die in der Skipiste dargestellt werden sollen (also die in Ebene 0 repräsentiert sind). Weiter sei $H(n)$ die Höhe der Skipiste (d.h. die maximale Höhe der Listenglieder) und $Z(n)$ die Anzahl der verwendeten Zeiger. Offenbar ist

$$Z(n) = \sum_{i=1}^n (h_i + 1) + (H(n) + 1),$$

wobei h_i die Höhe des i -ten Listenglieds ist. Der Summand $H(n) + 1$ steht für die Anzahl an Zeigern, mit denen der Kopf der Skipiste ausgestattet ist.

Wir berechnen nun die erwartete Höhe der Listenglieder und der gesamten Skipiste. Die erwartete Höhe jedes Listenglieds (der Erwartungswert der Funktion $random(0, 1, \dots)$) ist

$$\begin{aligned} \mathbb{E}[random(0, 1, \dots)] &= \sum_{h=0}^{\infty} h \cdot Prob[random(0, 1, \dots) = h] \\ &= \sum_{h=0}^{\infty} h \cdot \left(\frac{1}{2}\right)^{h+1} = 1. \end{aligned}$$

Dann gilt:

$$\begin{aligned}
\text{Prob}[H(n) \geq h] &\leq n \cdot \text{Prob}[\text{random}(0, 1, \dots) \geq h] \\
&= n \cdot \left(1 - \sum_{k=0}^{h-1} \text{Prob}[\text{random}(0, 1, \dots) = k] \right) \\
&= n \cdot \left(1 - \sum_{k=0}^{h-1} \frac{1}{2^{k+1}} \right) \\
&= n \cdot \left(1 - \left(1 - \left(\frac{1}{2} \right)^h \right) \right) \\
&= n \cdot \left(\frac{1}{2} \right)^h.
\end{aligned}$$

Insbesondere ist

$$\text{Prob}[H(n) \geq h] \leq \min \left\{ 1, n \cdot \left(\frac{1}{2} \right)^h \right\}.$$

Die Erwartungswerte $\mathbb{E}[H(n)]$ und $\mathbb{E}[Z(n)]$ ergeben sich wie folgt:

$$\begin{aligned}
\mathbb{E}[H(n)] &= \sum_{h \geq 0} h \cdot \text{Prob}[H(n) = h] \\
&= \text{Prob}[H(n) = 1] + \text{Prob}[H(n) = 2] + \text{Prob}[H(n) = 3] + \dots \\
&\quad + \text{Prob}[H(n) = 2] + \text{Prob}[H(n) = 3] + \dots \\
&\quad + \text{Prob}[H(n) = 3] + \dots \\
&\quad \vdots \\
&= \sum_{h \geq 1} \sum_{i \geq h} \text{Prob}[H(n) = i] \\
&= \sum_{h \geq 1} \text{Prob}[H(n) \geq h].
\end{aligned}$$

Aus obiger Abschätzung für die Wahrscheinlichkeiten $\text{Prob}[H(n) \geq h]$ folgt:

$$\begin{aligned}
\mathbb{E}[H(n)] &= \sum_{h=1}^{\lfloor \log n \rfloor + 2} \underbrace{\text{Prob}[H(n) \geq h]}_{\leq 1} + \sum_{h > \lfloor \log n \rfloor + 2} \underbrace{\text{Prob}[H(n) \geq h]}_{\leq n \cdot (1/2)^h} \\
&= \mathcal{O}(\log n) + \underbrace{\sum_{\substack{h > \lfloor \log n \rfloor + 2 \\ \leq n \cdot (1/2)^{\lfloor \log n \rfloor + 1}}} n \cdot (1/2)^h}_{< n \cdot 1/n = 1} \\
&= \mathcal{O}(\log n).
\end{aligned}$$

Die erwartete Anzahl an Zeigern ergibt sich wie folgt. Erwartungsgemäß werden $\mathbb{E}[H(n)] + 1$ Zeiger für den Listenkopf benötigt. Für jedes der n Listenglieder ist die erwartete Höhe gleich 1 (s.o.). Also werden im Mittel zwei Zeiger für jedes Listenglied benötigt (jeweils ein Zeiger für die Ebenen 0 und 1). Damit ergibt sich

$$\mathbb{E}[Z(n)] = 2n + \mathbb{E}[H(n)] + 1 = \mathcal{O}(n).$$

Damit ist gezeigt, daß der erwartete Platzbedarf linear ist.

Wir erläutern nun kurz, wie man einen Datensatz mit vorgegebenem Schlüsselwert x in einer randomisierten Skipliste sucht. Dazu verwenden wir dieselben Ideen wie für perfekte Skiplisten; jedoch benötigen wir eine sequentielle Suche innerhalb der einzelnen Ebenen. Siehe Algorithmus 126. Die Kostenanalyse der Suche in randomisierten Skiplisten ist

Algorithmus 126 Suche nach Schlüsselwert x in einer randomisierten Skipliste

```

 $i := k;$                                      (* wir starten mit der obersten Ebene  $i = k$  *)
 $v :=$  Kopfelement der Skipliste;           (* wir nehmen den Pseudoschlüsselwert  $key(v) = -\infty$  an *)
WHILE  $i > 0$  und  $key(v) \neq x$  DO

    (* sequentielle Suche in Liste  $i$  gestartet mit  $v$  *)

     $w := Next_i(v);$ 
REPEAT
    IF  $w \neq \perp$  and  $key(w) \leq x$  THEN
         $v := w;$ 
         $w := Next_i(w);$ 
    FI
    UNTIL  $w = \perp$  oder  $key(w) > x$ ;
            (*  $v$  ist „letztes“ Listenglied von Ebene  $i$  mit  $key(v) \leq x$  *)
    IF  $key(v) = x$  THEN
        gib den in Listenglied  $v$  gespeicherten Datensatz aus und halte an
    ELSE
         $i := i - 1$                                 (* gehe zur nächsttieferen Ebene *)
    FI
OD                                         (* Ebene 0 ist erreicht oder  $x$  gefunden *)
IF  $key(v) = x$  THEN
    gib den in Listenglied  $v$  gespeicherten Datensatz aus
ELSE
    gib „Schlüsselwert  $x$  nicht vorhanden“ aus.
FI
```

leider sehr viel schwieriger als in perfekten Skiplisten. Wir verzichten auf die Ausführung der Analyse und zitieren das Resultat, daß die erwartete Laufzeit der Suche nach einem Schlüsselwert logarithmisch in der Anzahl n an gespeicherten Datensätzen ist. Einfügen und Löschen kann mit Hilfe des Suchalgorithmus ebenfalls in logarithmischer Zeit vorgenommen werden.

Wir fassen die Ergebnisse zusammen:

Satz 10.4.1. Die *erwartete Laufzeit* für Suchen, Löschen und Einfügen in randomisierten Skiplisten ist $\mathcal{O}(\log n)$. Der *erwartete Platzbedarf* ist $\mathcal{O}(n)$. Dabei ist n die Anzahl der

darzustellenden Schlüsselwerte.⁸³

⁸³Wie in Abschnitt 4 wird davon ausgegangen, daß die Darstellung der zu einem Schlüsselwert gehörenden Information sowie die Darstellung der Schlüsselwerte selbst lediglich eine konstante Anzahl an Platzeinheiten benötigt.

Literatur

Fast alle in der Vorlesung besprochenen Themen können in

[CRL] T. Cormen, C. Leiserson, R. Rivest, C. Stein: *Introduction to Algorithms*, MIT Press, 2006 (3. Auflage).

und dem “Klassiker”

[AHU83] A. Aho, J. Hopcroft, J. Ullman: *Data Structures and Algorithms*, Addison-Wesley, 1983.

nachgelesen werden. Weitere Literatur:

[AHU74] A. Aho, J. Hopcroft, J. Ullman: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[Blum] N. Blum: *Theoretische Informatik - Eine anwendungsorientierte Einführung* Oldenburg Verlag, 1998.

[Güt] R. Güting: *Datenstrukturen und Algorithmen*, Teubner Verlag, 1992.

[HS] E. Horowitz, S. Sahni: *Computer Algorithms*, Computer Science Press, 1978.

[OW] T. Ottmann, P. Widmayer: *Algorithmen und Datenstrukturen*, Spektrum Akademischer Verlag, 1996 (3. Auflage).

Index

- α - β -Pruning, 243
- α -Cut, 243
- β -Cut, 243
- $\gamma(\bar{x})$, 221
- 15er Puzzle, 232
- Ableitung, 276
- Ableitungsbaum, 276
- Abstandsfunktion, 291, 311
- Abstraktion, 10
- Add, 68
- Adjazenzliste, 83
- Adjazenzmatrix, 83
- Aho,Hopcroft,Ullman (SCC-Berechnung), 378
- AHU-Algorithmus, 378
- aktives Objekt, 174
- all-pairs, 92, 329
- Alphabet, 250, 270, 276
- Array, 63
 - asymptotische Notationen, 12
 - aufspannender Baum, 119, 344
 - Auftragsplanung mit Schlussterminen, 264
 - Aufzählungsbaum, 200
 - augmenting path, 392
 - ausgeglichenes Mischen, 55
 - Austauscheigenschaft, 260, 267, 352
 - Average-Case-Analyse, 15
 - AVL-Baum, 155
 - AVL-Blattsuchbaum, 171
 - azyklischer Graph, 82
- B-Baum, 178
- Backtracking, 200, 202
 - n -Damen, 202
 - Graphfärbungen, 213
 - Rucksackproblem, 218
- Balance
 - B-Bäume, 186
- Baum
 - aufspannend, 119, 344
- Binär-, 97
- entartet, 97
- Fibonacci-, 157
- gerichtet, 95
- leerer, 95
- Spiel-, 236
- Teil-, 97
- ungerichtet, 116, 340
- vollständig, 97
- Baumkante, 344, 360
- Bayer-Baum, 178
- Bellman & Ford, 324
- benachbart, 80
- Bereichsanfrage, 170
- Best-First-Search, 231
- BFS, 88
- Binärbaum, 97
- Binärkode, 251
- Binäre Suche, 22
- Binärer Suchbaum, 148
- Binomialkoeffizient, 146, 268
- bipartiter Graph, 408
- bipartites Matching, 408
- Bitvektor, 63
- Blatt, 41, 95
- Blattsuchbaum, 170
- Boyer-Moore, 422
- Brüder, 97
- Branch & Bound, 200, 223
 - 15er Puzzle, 232
- Rucksackproblem, 224
- TSP, 301
- Breadth First Search, 88
- Breitensuche, 88, 403
- Bubblesort, 28
- Bucketsort, 49
- $c(v,w)$, 242, 386
- $\text{cap}(S)$, 388
- Chomsky Normalform, 277, 382
- Clusterbildung, 141

- CNF, 277
 Cocke, Younger, Kasami, 279
 Codelänge
 mittlere, 253
 Codewort, 251
 Codierung, 251
 Computed Table, 146
 Concatenation
 B-Bäume, 186
 $\text{Cost}(W, v)$, 297
 Cutting, 243
 CYK-Algorithmus, 279

 dünnbesetzte Matrizen, 65
 DAG, 82
 DATALOG, 385
 Decodierung, 252
 Depth First Search, 88
 DFS, 88
 Baumkante, 360
 Besuchsnummern, 360
 Kantenklassifizierung, 360
 Knotenfärbung, 362
 Rückwärtskante, 360
 Seitwärtskante, 362
 Vorwärtskante, 360
 Wald, 360
 Wurzel, 360
 $\text{DFS}(v)$, 88, 360
 dfsNr, 360
 Digraph, 78
 Dijkstra, 315
 directed acyclic graph (DAG), 82
 DIVIDE & CONQUER, 302
 Divide & Conquer, 191
 Binäre Suche, 22
 binäre Suche, 191
 Matrizenmultiplikation, 197
 Mediansuche, 57, 191
 Mergesort, 30, 191
 Quicksort, 34, 191
 Doppelrotation, 159
 Double Hashing, 142
 dynamisches Programmieren, 268, 328

 Bellman & Ford, 328
 Binomialkoeffizienten, 268
 CYK-Algorithmus, 278
 Floyd, 329
 optimale Suchbäume, 287
 reguläre Ausdrücke von NEA, 269
 TSP, 297

 Ebene
 Skipliste, 71
 Edmond-Karp Algorithmus, 403
 einfacher Pfad, 79
 einfacher Zyklus, 82
 Eingabegröße, 12, 15
 endlicher Automat, 270
 entarteter Baum, 97
 Entscheidungsbaum, 41
 Entscheidungsvariante
 Graphfarben, 211
 TSP, 293
 Entwurfsmethoden, 191
 erreichbar, 79
 Erwartungswert, 17
 erweiterte Gewichtsfunktion, 259
 erweiterte Kapazitätsfunktion, 389
 Euklid-Algorithmus, 24
 Exponentielle Suche, 24
 externes Sortieren, 55
 ExtractMax, 121, 123
 ExtractMin, 121, 123

 $F(\mathcal{T})$, 285
 $F(i, j)$, 285
 Fehlstand, 235
 Fibonacci
 -Baum, 157
 Zahlen, 25, 157
 FIND, 350
 Fingerprint-Technik, 434
 $\text{Flow}(f)$, 387
 $\text{Flow}(f, S)$, 389
 Floyd, 329
 Flusserhaltungsgesetz, 386
 Flussfunktion, 386

Flusswert, 387
 Flusserhöhung, 393
 Ford & Fulkerson, 391
 Front, 68
 gerichteter Graph, 78
 gerichteter Wald, 98
 Gewichtsfunktion
 δ , 310
 w , 259
 erweitert, 259, 329
 ggT, 24
 größter gemeinsamer Teiler, 24
 Grammatik, 276, 382
 Graph, 78, 310
 regulär, 414
 Spiel-, 236
 Graphfärben, 208
 Greedy-Algorithmus, 223, 249
 Auftragsplanung, 264
 Dijkstra, 315
 Graphfarbe-Heuristik, 208
 Huffman, 250
 Kruskal, 351
 Prim, 345
 Rucksackproblem, 215
 Guess & Check Methode, 202, 211, 218, 293
 Höhe, 41
 AVL-Bäume, 155
 B-Baum, 180
 Heaps, 124
 mittlere, 43
 Skipliste, 71
 UNION-FIND-Bäume, 355
 von Bäumen, 97
 Halde, 121
 Hall
 Satz von, 412
 Haltepunkt, 174
 Hamilton-Weg-Problem, 293
 Hamiltonkreis, 291
 harmonischen Reihe, 37, 140
 Hashfunktion, 131
 Hashing, 131
 Double, 142
 mit Kollisionslisten, 134
 offene Adressierung, 136
 universell, 143
 Heap, 121
 Heapsort, 125
 Heuristik
 Graphfärben, 208
 Rucksackproblem, 215
 TSP, 295
 Hilfsfunktion h , 417
 Huffman-Codes, 250
 Index, 178
 Inorder, 107
 Kürzeste Rundreise, 291
 Kürzeste Wege-Probleme, 92
 Kürzeste-Wege, 91, 310, 374
 Kantengewichte, 311
 Kapazität
 Rest-, 391
 Schnitt-, 388
 Kapazitätsfunktion, 386
 erweitert, 389
 Karp-Rabin, 433
 Keller, 68
 Kettenregel, 382
 KMP-Algorithmus, 417
 Knotenfärbung (DFS-), 362
 Knuth, Morris und Pratt, 417
 Kollision, 132
 Kollisionslisten, 134
 Konfiguration, 236
 kontextfreie Grammatik, 276, 382
 kontextfreie Sprache, 276, 382
 Korrektheit
 partiell, 8
 total, 8
 Kosten
 $\Delta(v, w)$, 312
 $\Delta(w)$, 326

$cost(\mathcal{T})$, 345
 $\delta(\pi)$, 311
einer Rundreise, 291
Kostenfunktion
für Algorithmen, 12
in Graphen, 310
Kostenmaß
logarithmisch, 15
uniform, 14
Kreis, 82
kreisfrei, 82
Kruskal, 350
Länge
von Pfaden, 79
LAS VEGAS Algorithmus, 433
Lauf, 55, 270
LC-Methode, 229
Least Cost Methode, 229
leerer Baum, 95
left(v), 252
length(π), 79
LIFO-Methode, 224
lineare Sondierung, 141
linksvollständig, 122
Liste, 63
Listenhöhe
Skipliste, 73
logarithmisches Kostenmaß, 15
LRW, 107
LWR, 107
Master-Theorem, 192, 302
Matching
bipartit, 408
maximal, 408
perfekt, 412
Matrizenmultiplikation
nach Strassen, 197, 307
dünnbesetzte Matrizen, 66
Matroid, 260
der Aufträge, 266
der zyklenfreien Kantenmengen, 351
Dijkstra, 323
MaxFlow(N), 387
MaxFlow-MinCut-Theorem, 388
maximaler Fluss, 387
maximales Matching, 408
Maximalität
für Matroide, 262, 352
Maximumsheap, 229
Maximumsheaps, 121
Maximumssuche, 57
Maxknoten, 237
Maxsummenproblem, 19
MaxValue(v, ℓ), 245
Median, 57, 193
Median-Algorithmus
deterministisch, 60
randomisiert, 58
Mergesort, 30, 191, 192
mincost, 297
MinCut(N), 388
minimal spanning tree, 345
minimale Schnittkapazität, 388
minimaler aufspannender Baum, 345
Minimax-Verfahren, 241
Minimumsheap, 254, 322, 350
Minimumssuche, 28, 57
Minknoten, 237
MinValue(w, ℓ), 245
Mischen, 32
ausgeglichenes, 55
extern, 55
mittlere Codelänge, 253
mittlere Höhe, 43
mittlere Suchzeit, 284
MONTE CARLO Algorithmus, 434
MST, 345
Mustererkennung, 416
n-Damen-Problem, 202
Nachfolger, 79
Nettofluss, 389
Netzwerk, 386
Netzwerkflussproblem, 386
Nichtterminal, 276
NP-Vollständigkeit, 211, 215, 292

offene Adressierung, 136
 optimaler Präfixcode, 253
 optimaler Suchbaum, 283
 Optimalitätsprinzip, 282

- Bellman-Ford Algorithmus, 328
- Floyd Algorithmus, 330
- optimale Suchbäume, 287
- TSP, 298

 Optimierungsvarianten

- TSP, 293

 Ordnung

- eines B-Baums, 179

 Overflow, 182

 $p(i, j)$, 285
 partielle Korrektheit, 8
 Payoff-Funktion, 240
 perfekte Skipliste, 71
 perfektes Matching, 412
 Pfad, 79

- einfach, 79
- zunehmend, 392

 Pfadkompression, 358
 Pivotelement, 34
 polynomiell-zeitbeschränkt, 16
 Pop, 68
 Post, 79
 Post*, 79
 Postfix, 69
 Postorder, 104, 107
 Präfix, 251
 Präfixcode, 252

- mittlere Codelänge, 253
- optimal, 253

 Pre, 79
 Pre*, 79
 Preorder, 104, 107
 Prim, 345
 Primzahlsatz, 437
 Priority Queue, 121, 229, 254, 322, 350
 Problem des Handlungsreisenden, 291
 Produktion, 276
 Pruning

- $\alpha\text{-}\beta$, 243

 Push, 68
 quadratische Sondierung, 141
 Quelle, 386
 Queue, 68

- Priority, 121

 Quicksort, 34, 62, 191
 Rückwärtskante, 360, 391
 Radixsort, 52
 $\text{random}(0, 1, \dots)$, 73
 randomisierte Skipliste, 73
 randomisiertes Quicksort, 41
 Rebalancierung, 159
 Reduktion, 10, 211, 411
 reflexive, transitive Hülle, 336
 Regel, 276

- Ketten-, 382

 regulärer Ausdruck, 270
 Rekurrenz, 22, 192
 Relaxationsschritt, 315
 Remove, 68
 Restgraph, 391
 Restkapazität, 391
 $\text{right}(v)$, 252
 Robin Hood Effekt, 144
 Rotation, 159
 Rucksackproblem, 215
 Rundreise, 291
 Rutten-Methode, 274
 Scanline-Prinzip, 174
 SCC, 376
 Schluesseluniversum, 130
 Schleifeninvariante, 86
 Schnitt, 388
 Schnittkapazität, 388
 Schwarzfärbungsindex, 379
 Seitwärtskante, 362
 Selektieren, 57
 semantische Äquivalenz \equiv , 270
 Sequentielle Suche, 21
 Sichtbarkeitsproblem, 174
 $\text{sign}(x)$, 235
 single-source, 92, 315

Skelett-Automat, 417
 Skipliste, 71

- perfekt, 71
- randomisiert, 73

 Sohn, 95
 Sondierung

- linear, 141
- quadratisch, 141

 Sondierungsfolge, 136
 Sortieren, 28

- durch Einfügen, 28
- durch Minimumssuche, 28
- extern, 55
- in linearer Zeit, 46
- intern, 28
- topologisch, 368

 sparse matrices, 65
 Spezifikation, 7
 Spielbaum, 236
 Spielgraph, 236
 Splitting, 182
 Stabilität, 53
 Stack, 68
 Stapel, 68
 stark zusammenhängend, 376
 starke Zusammenhangskomponente, 376
 Strassen

- Matrizenmultiplikation, 197, 307
- strongly connected component, 376

 Suchbaum, 148

- AVL-, 155
- binär, 148
- Blatt-, 170
- optimal, 283

 Suche

- AVL-Baum, 158
- B-Baum, 178, 181
- binär, 22
- binärer Suchbaum, 149
- Blattsuchbaum, 170
- Exponentielle, 24
- Hashing, 134, 138
- in Skiplisten, 71, 75
- Sequentiell, 21

 Suchtiefe, 239
 Suffix, 251
 tail-recursion, 39
 Teilbaum, 97
 Teilgraph, 115, 338
 Teilmengensystem, 259
 Teilwort, 251
 Terminal, 276
 Tic Tac Toe, 237
 Tiefe

- von Knoten in Bäumen, 97

 Tiefensuche, 88, 360
 Top, 68
 topologische Sortierung, 368, 383
 totale Korrektheit, 8
 Traveling Salesperson Problem, 291
 Traverse(v), 86
 Traversierung

- Graph-, 84

 TSP, 291
 U-Maximalität, 262, 352
 ungerichteter Baum, 340
 ungerichteter Graph, 78
 ungesättigte

- Rückwärtskante, 391
- Vorwärtskante, 391

 uniformes Kostenmaß, 14
 UNION-FIND, 350
 universelles Hashing, 142
 Vater, 95
 Verzweigungsgrad, 97
 Visited, 84
 vollständig, 97

- links-, 122

 Vorgänger, 79
 Vorwärtskante, 360, 391
 Wahrscheinlichkeitsverteilung, 17
 Wald

- BFS-, 100
- DFS-, 100, 360

 gerichtet, 98

Warshall, 336
Warteschlange, 68
Weg, 79
Wegeprobleme, 310
WLR, 107
Worst-Case-Analyse, 15
Wort, 250
Wortproblem
 kontextfreie Sprachen, 276
Wurzel, 95
 DFS, 360
Wurzelknoten, 95

Zerlegungskosten, 193
Zielknoten, 386
zunehmender Pfad, 392
zusammenhängend, 80, 111, 337
 stark, 376
Zusammenhangskomponente, 80, 111, 337
 stark, 376
Zusammensetzungskosten, 193
Zyklentest
 in Digraphen, 360, 371, 378
 in ungerichteten Graphen, 117, 341,
 354
zyklischer Graph, 82
Zyklus, 82
 einfach, 82
 negativer Kosten, 313, 327, 333
Zyklus(v), 341