Algorithmen und Datenstrukturen

Benjamin Blankertz und Vera Röhr

TECHNISCHE UNIVERSITÄT BERLIN

VORLESUNGSSKRIPT

Algorithmen und Datenstrukturen

Benjamin Blankertz und Vera Röhr

[Fachgebiet Neurotechnologie, MAR 4.3]

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Gedankenexperiment zur Laufzeit 1
1.2 Algorithmische Konzepte 2

2 Laufzeit und Komplexität **4**
2.1 Einschub: Wiederholung von Wachstumsordnungen 4
2.2 Wachstumsordnungen von Laufzeit und Speicherbedarf 8
2.2.1 Empirische Laufzeitanalyse von `TwoSum` 9
2.2.2 Analytische Laufzeitanalyse von `TwoSum` 11
2.3 Amortisierte Laufzeitanalyse 12
2.4 Hinweise für die Praxis mit vielen Beispielen 13
2.4.1 Abschätzung der Laufzeit von Funktionen ohne Rekursion 14
2.4.2 Abschätzung der Laufzeit von rekursiven Funktionen . . . 18
2.4.3 Laufzeit von Funktionen mit komplexeren Datenstrukturen 20
2.4.4 Laufzeit von Pseudocode 21
2.5 Laufzeiten bei einfachen Java Datenstrukturen 23
2.6 P, NP und NP-Vollständigkeit 24

3 Datenstrukturen **27**
3.1 Vorrangwarteschlange 27
3.1.1 Implementation 30
3.1.2 Laufzeiten 32
3.2 Indizierte Vorrangwarteschlangen 32
3.2.1 Implementation 34
3.2.2 Laufzeitbetrachtung 36
3.3 Union-Find: Dynamische Zusammenhangskomponenten 38
3.3.1 Laufzeiten der Union-Find Varianten 45

4 Backtracking **46**
4.1 Beispiel: Aufzählung aller Permutationen 47
4.2 Beispiel: Rösselsprung 50
4.3 Generelles Backtrack-Prinzip 54

4.4 Beispiel: Sudoku 55
4.5 Einschub: Bäume und Baumdurchsuchungen 57
4.6 *Backtracking* als Suche im Baum der Teillösungen 59
4.7 Das Wichtigste in Kürze 63

5 Gierige Algorithmen (*Greedy Algorithms*) 64
5.1 Einführung zu Greedy-Algorithmen 64
5.2 Generelles Prinzip von Greedy-Algorithmen 66
5.3 Greedy Beispiel: Intervallauswahl 67
5.3.1 Optimalität von "Frühestes-Ende" 69
5.3.2 Implementation 70
5.4 Greedy Beispiel: Intervallverteilung 71
5.4.1 Optimalität von "Frühester-Start" 73
5.4.2 Implementation 75
5.5 Greedy Beispiel: Planung mit minimalen Verspätungen 77
5.5.1 Optimalität von "Früheste-Deadline" 78
5.6 Strategien für Optimalitätsbeweise von gierigen Algorithmen . . 82
5.7 Greedy-Algorithmus für das teilbare Rucksackproblem 83
5.7.1 Optimalität von "Greedy-Knapsack" 84
5.7.2 Einschränkung des Greedy-Knapsack Algorithmus 87
5.8 Weitere Anwendungen von gierigen Algorithmen 87
5.9 Das Wichtigste in Kürze 87

6 Minimax Verfahren und Alpha-Beta Suche 89
6.1 Minimax-Verfahren 89
6.2 Motivation der Alpha-Beta-Suche 94
6.2.1 Negamax Variante der Alpha-Beta Implementierung . . . 98

7 Branch-and-Bound 101
7.1 Das Konzept von *Branch-and-Bound* 101
7.2 Beispiel: *Branch-and-Bound* für das 0/1-Rucksackproblem 104
7.3 *Beyond Branch-and-Bound* 110
7.4 Das Wichtigste in Kürze 113

8 Dynamisches Programmieren 114
8.1 Einführung in die Dynamische Programmierung 115
8.2 Beispiel 1: Gewichtete Intervallauswahl 122
8.3 Beispiel 2: Das 0/1-Rucksack Problem 126
8.4 Beispiel 3: Editierdistanz 129

9 Ungewichtete Graphen 134
9.1 Graphen - Einführung der Grundbegriffe 135

9.2 Durchsuchen von Graphen 143
9.3 Tiefensuche (DFS) in Graphen 147
9.4 Breitensuche in Graphen 155
9.5 Gerichtete Graphen (Digraphen) 160
9.5.1 Zeitstempel bei der Tiefensuche 162
9.6 Topologische Sortierung 164
9.7 Kategorisierung von Kanten in Digraphen 168
9.8 Das Wichtigste in Kürze 170

10 Minimale Spannbäume 173
10.1 Gewichtete Graphen 173
10.2 Minimale Spannbäume 174
10.3 Prims Algorithmus für minimale Spannbäume 179
10.4 Der unbekanntere Kruskal Algorithmus: *Reverse-Delete* 184
10.5 Kruskals Algorithmus für minimale Spannbäume 184
10.6 Das Wichtigste in Kürze 187

11 Kürzeste Wege in gewichteten Graphen 189
11.1 Einleitung zu kürzesten Wegen 189
11.1.1 Kantengewichtete Digraphen 190
11.2 Single-Source Shortest Paths 192
11.3 Kürzeste Wege in gewichteten azyklischen Digraphen 197
11.4 Dijkstra Algorithmus für kürzeste Wege 201
11.5 Bellman-Ford: Kürzeste Wege in Digraphen mit beliebigen Gewichten 207
11.5.1 Effiziente Version des Bellman-Ford Algorithmus 209
11.6 Unterschiede zwischen Dijkstra und Bellman-Ford 210
11.7 Das Wichtigste in Kürze 213

12 Flussgraphen 214
12.1 Flussgraphen 214
12.1.1 Minimale Schnitte und Maximaler Fluss 221
12.2 Allgemeine Methode zum Identifizieren des maximalen Flusses . 227
12.2.1 Überlegungen zur Konkretisierung der Ford-Fulkerson Methode 231
12.3 Der Edmonds-Karp Algorithmus 232
12.4 Der Kapazitätskontrolle Algorithmus (*Capacity Scaling*) 236
12.5 Laufzeiten von *Maxflow* Algorithmen 239

13 Heuristische Algorithmen 240
13.1 Motivation für die Verwendung von Heuristiken 240
13.2 Der A* Algorithmus für kürzeste Wege 244

13.3 A* jenseits von direkten Graphenproblemen 252
13.4 Heuristiken für das Problem des Handlungsreisenden 253
13.5 Das Wichtigste in Kürze 254

14 Approximative Algorithmen 256
14.1 Approximation für das metrische TSP 256
14.2 Nicht-Approximierbarkeit des allgemeinen TSP 258
14.3 Approximationsschema für das 0/1-Rucksackproblem 259
14.4 Das Wichtigste in Kürze 263

15 Randomisierte Algorithmen 264
15.1 Beispiel: Maximal azyklischer Untergraph 265
15.2 Das Wichtigste in Kürze 266

16 Hashtabellen 267
16.1 Hashfunktionen 269
16.1.1 Kollisionen und perfektes Hashing 270
16.1.2 Hashfunktion: Multiplikationsmethode 271
16.1.3 Hashfunktion: Divisions-Rest-Methode 273
16.2 Die Idee des Universellen Hashing 274
16.3 Strategien zur Kollisionsauflösung 275
16.3.1 Hashverfahren mit Verkettung (*Separate Chaining*) 275
16.3.2 Effizienz von Hashing mit Verkettung 277
16.3.3 Hashverfahren mit Sondierung (*probing*) 278
16.3.4 Quadratisches Sondieren 280
16.3.5 Effizienz von Hashverfahren mit Sondierung 282
16.4 Hashing mit Java Objekten 283
16.4.1 Hashtabellen in Java Collections 284
16.5 Universelles Hashing 285
16.6 Das Wichtigste in Kürze 286

A Appendix 287
A.1 Branch-and-Bound Implementierung für das Rucksackproblem . 288
A.2 Implementierungen von Symboltabellen 291

Abbildungsverzeichnis 293

Listings 295

Index 298

Literaturverzeichnis 304

KAPITEL 1

Einleitung

Dieses Skript soll die Leserinnen und Leser in die wunderbare Welt der Algorithmen und Datenstrukturen einführen. Unter **Algorithmen** versteht man Verfahren als schrittweise Anleitung zum Lösen von quantitativen Problemen. **Datenstrukturen** dienen der Organisation von Daten zur effizienten Verarbeitung. In der TU Lehrveranstaltung "Algorithmen und Datenstrukturen" wird die Implementation dieser Bausteine durch die Objekt-orientierte Programmierung in Java praktisch umgesetzt.
Die Kenntnis effizienter Algorithmen und die Fähigkeit solche Algorithmen für spezielle Anwendungen auszuwählen und anzupassen, spielt in vielen Bereichen der Informatik eine entscheidende Rolle. Dabei geht es weniger um auswendig gelernte Verfahren, sondern vielmehr um die Fähigkeit, algorithmische Lösungsstrategien zu unbekannten Problemvarianten zu entwickeln.

1.1 Gedankenexperiment zur Laufzeit

Auch angesichts des Wettbewerbs um immer schneller werdende Computer und immer schneller laufende Software, ist es essenziell zu wissen, auf welche Weise komplexe Probleme effizient gelöst werden können. Daher ist es wichtig, sich über Wachstumsverhalten von Laufzeit Gedanken zu machen und effiziente Implementationsmöglichkeiten zu kennen.
Dies wird an dem folgenden Gedankenexperiment illustriert. Wir nehmen an, dass Sie ein Programm zur Lösung eines komplexeren Problems geschrieben haben. Durch geschicktes Programmieren ist es Ihnen gelungen, eine Laufzeit von einer Minute bei einer Eingabegröße von $N = 1000$ zu erreichen. Was passiert, wenn Ihr Programm in der Praxis eingesetzt wird, und die Datenmenge wächst, die das Programm verarbeiten muss?

Wachstum	W-Ordnung	$N = 1000$	$10N$	$100N$	$1000N$
konstant	1	1 Minute	1 Minute	1 Minute	1 Minute
logarithmisch	$\log N$	1 Minute	$1,\bar{3}$ Minuten	$1,\bar{6}$ Minuten	2 Minuten
linear	N	1 Minute	10 Minuten	2 Stunden	1 Tag
leicht überlinear	$N \log N$	1 Minute	13 Minuten	3 Stunden	$1\frac{1}{2}$ Tage
quadratisch	N^2	1 Minute	2 Stunden	5 Tage	23 Wochen
kubisch	N^3	1 Minute	1 Tag	2 Jahre	2000 Jahre
exponenziell	2^N	1 Minute	`inf`	`inf`	`inf`

Tabelle 1.1 Laufzeiten in Abhängigkeit von Wachstumsordnung: Für ein Programm wird eine Laufzeit für eine Eingabegröße von $N = 1000$ von einer Minute angenommen. Die Tabelle zeigt die berechnete Laufzeit für andere Eingabegrößen in Abhängigkeit von der Wachstumsordnung der Laufzeit. Konkret wurde für die Berechnung der konstante Faktor in den Laufzeitfunktionen so gewählt, dass sich für $N = 1000$ eine Minute ergibt. Die angegebenen Laufzeiten für $10N$, $100N$, $1000N$ sind gerundet.

Hierfür ist die Kenntnis der Wachstumsordnung der Laufzeit der verwendeten Algorithmen relevant. Dieses Konzept wird in Kapitel 2 genauer erläutert. Hier ist ein intuitives Verständnis ausreichend.

- Annahme: Die Laufzeit für eine Eingabe der Größe $N = 1000$ ist eine Minute.
- Als mögliche Wachstumsordnung der Laufzeit Ihres Programmes, betrachten wir die typischen Fälle: konstant, logarithmisch, linear, leicht überlinear, quadratisch, kubisch und exponenziell.

Tabelle 1.1 zeigt, wie sich die Laufzeit für Eingabegrößen von $10N$, $100N$ und $1000N$ in Abhängigkeit der Wachstumsordnung verhält. Es hängt offensichtlich an der Wachstumsordnung, ob und für welche Anwendungen (und auf welcher Hardware) das Programm sinnvoll eingesetzt werden kann. Für viele Probleme hat eine offensichtliche und einfach zu implementierende Lösung eine exponentielle Laufzeit, z. B. bei einer kompletten Durchsuchung eines Lösungsraums mit Backtracking. Mit mehr Fachkenntnis und oft nur geringfügig größerem Implementationsaufwand lassen sich viele Probleme deutlich effizienter lösen. Manchmal ist es dabei erforderlich, Abstriche bei der Optimalität der Lösung hinzunehmen und Approximative Algorithmen zu verwenden, oder nur bestimmte Problemkonstellationen zu betrachten, um diese gezielter lösen zu können, wie bei der Verwendung Heuristischer Algorithmen.

1.2 Algorithmische Konzepte

Dieser Abschnitt gibt einen kurzen Überblick für die unterschiedlichen algorithmischen Konzepte, die in diesem Skript dargestellt werden. bezeichnen?)

Wir unterscheiden zwischen Kategorien von Verfahren. Die erste Kategorie enthält grundlegende Verfahren zur Suche von Lösung gewisser Probleme. Dieser Prozess kann als eine Suche in einem Baum (Baum der Teillösungen) dargestellt werden. Diese Kategorie umfasst die Paradigmen Backtracking, *Branch-and-Bound*, *Greedy*-Algorithmen, Dynamisches Progarmmieren, sowie *Divide-and-Conquer*. **(Letztere Technik wird nicht in diesem Skript behandelt, da sie Teil der Vorlesung "Einführung in die Programmierung" (IntroProg) ist.)**
Die zweite Kategorie umfasst Variationen von Algorithmenarten. Dies sind Heuristische Algorithmen, Approximative Algorithmen und Randomisierte Algorithmen.

Konzepte von Algorithmen, die als Suche im Baum von Teillösungen aufgefasst werden können

- **Backtracking** generiert schrittweise Lösungskandidaten und versucht sie zu einer (optimalen) Lösung zu erweitern (bzw. alle Lösungen zu finden). Bei Misserfolg wird zu einem früheren Kandidaten zurückgekehrt und weitergesucht.
- **Branch-and-Bound** erkundet den Lösungsraum in einem Entscheidungsbaum, wobei nicht aussichtsreiche Zweige abgeschnitten werden.
- **Greedy-Algorithmen:** Schrittweise Auswahl, die auf direktem Weg zu einer optimalen Lösung führt
- **Dynamische Programmierung:** Aufteilung in überlappende Teilprobleme; Speicherung von Zwischenlösungen, um größere Probleme unter Rückgriff auf gespeicherte Zwischenlösungen effizient zu lösen
- **Divide-and-Conquer:** Aufteilung in kleinere Teilprobleme, deren Lösungen zu Lösungen größerer Probleme zusammengesetzt werden

Variationen von Algorithmen

- **Randomisierte Algorithmen** verwenden Zufallsentscheidungen, um bessere Laufzeiten zu erzielen.
- **Heuristische Algorithmen** verwenden eine Heuristik, um den Lösungsraum zunächst in aussichtsreichere Richtungen zu explorieren.
- **Approximative Algorithmen** nehmen Abweichungen von der optimalen Lösungen in Kauf, um bessere Laufzeiten zu erzielen. Für die relative Größe der Abweichung können garantierte Grenzen angegeben werden.

KAPITEL 2

Laufzeit und Komplexität

2.1 Einschub: Wiederholung von Wachstumsordnungen

> Da das Thema *Wachstumsordnungen* in der Vorlesung IntroProg behandelt wird, ist die Darstellung hier kurz gehalten und nur als Wiederholung gedacht.

Zur Charakterisierung von Rechenzeit und Speicherbedarf von Algorithmen werden häufig die folgenden Funktionsklassen von Wachstumsordnungen verwendet: $\boldsymbol{o}(f)$, $\boldsymbol{O}(f)$, $\boldsymbol{\Theta}(f)$, $\boldsymbol{\Omega}(f)$, $\boldsymbol{\omega}(f)$.
Die Wichtigste im Rahmen dieses Skriptes ist

$$\boldsymbol{O}(f) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c > 0\ \exists n_0\ \forall n > n_0\ \ g(n) < c \cdot f(n)\}$$

Die wichtigsten Wachstumsordnungen sind

- $\boldsymbol{O}(1)$ konstant
- $\boldsymbol{O}(\log N)$ logarithmisch
- $\boldsymbol{O}(N)$ linear
- $\boldsymbol{O}(N \log N)$ 'leicht überlinear'
- $\boldsymbol{O}(N^2)$ quadratisch
- $\boldsymbol{O}(N^3)$ kubisch
- $\boldsymbol{O}(2^N)$ exponentiell zur Basis 2

Um die Begriffe konkreter zu besprechen, wird zunächst geklärt, was unter der *Eingabegröße* verstanden wird. Laufzeiten und Speicherbedarf werden nämlich in ihrer Abhängigkeit von der Eingabegröße in dem Algorithmus betrachtet.

Definition: Eingabegröße

Die Eingabegröße bei einer Anwendung eines Algorithmus auf eine konkrete Eingabe ist die Anzahl der Bits, die benötigt werden, um alle Eingabedaten zu kodieren.

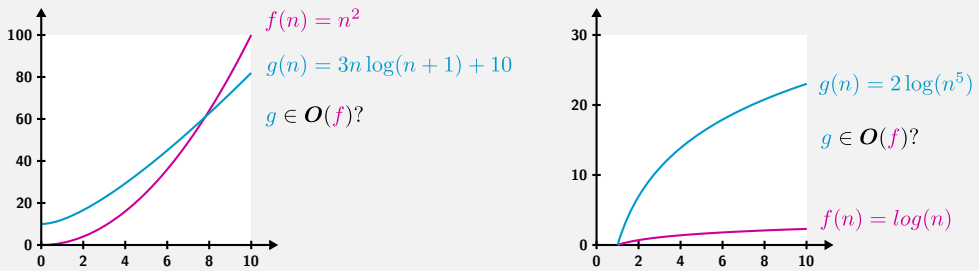
Wie wir weiter unten sehen werden, spielen konstante Faktoren bei der (asymptotischen) Laufzeitbetrachtung, und damit auch bei der Eingabegröße, keine Rolle. Daher reicht es in vielen Fällen aus, sich unter der Eingabegröße die Anzahl der Eingabewerte vorzustellen. Für einen Sortieralgorithmus ist dies z. B. die Anzahl der Zahlen, die sortiert werden soll. Wenn diese Zahlen einen Datentyp fester Größe (wie `int` oder `double`) haben, unterscheidet sich die Anzahl der benötigten Bits nur um einen konstanten Faktor von der Anzahl der Zahlen (also der Länge des Arrays oder der Liste). Bei einem Datentyp mit variabler Länge wie `String` muss die Länge der Zeichenketten mit berücksichtigt werden. Ein anderer Spezialfall bei der Betrachtung von Laufzeit in Abhängigkeit der Eingabe liegt vor, wenn die Abschätzung abhängig vom konkreten *Wert* eines Eingabeparameters ist. Dieser Fall tritt in Abschnitt 8.3 auf.

Die mathematische Definition der Funktionenklasse $\boldsymbol{O}(\cdot)$ sieht kompliziert aus, trägt aber in der Anwendung zur Vereinfachung bei. Die exakte Laufzeit eines Algorithmus in Abhängigkeit von der Eingabegröße ist ein äußerst komplexer Term. Mit Hilfe der Funktionenklasse $\boldsymbol{O}(\cdot)$ kann die Wachstumsordnung der Laufzeit auf einen einfachen Term reduziert werden. So wird die Laufzeit von Algorithmen mit Ausdrücken wie $\boldsymbol{O}(N^2)$ oder $\boldsymbol{O}(N \log N)$ angegeben, obwohl die exakte Laufzeit eine viele kompliziertere Beschreibung hat. Durch diese starke Vereinfachung mit Hilfe von Wachstumsordnungen geht natürlich auch Information verloren. Für die meisten Entscheidungen hat dies glücklicherweise keine Nachteile.

Als ersten Schritt im Verständnis von Wachstumsordnungen sollte man die Funktionenklasse $\boldsymbol{O}(\cdot)$ verstehen. Wir schreiben, wie üblich, $\boldsymbol{O}(N^2)$ für $\boldsymbol{O}(f)$ mit der Funktion $f(N) = N^2$ und analog $\boldsymbol{O}(1)$, $\boldsymbol{O}(N)$, $\boldsymbol{O}(N \log N)$, $\boldsymbol{O}(N^3)$, $\boldsymbol{O}(2^N)$. Die Funktionenklasse $\boldsymbol{O}(N^2)$ umfasst gemäß der Definition

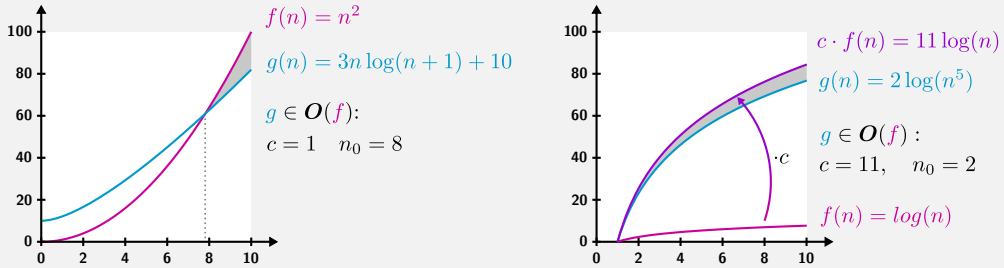
$$\boldsymbol{O}(f) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c > 0\ \exists n_0\ \forall n > n_0 \quad g(n) < c \cdot f(n)\}$$

alle Funktionen, die 'nicht stärker als quadratisch' ansteigen. Genauer gesagt gehört eine Funktion g zu der Funktionenklasse $\boldsymbol{O}(N^2)$, wenn es eine Konstante c gibt, so dass g ab einem gewissen (und möglicherweise sehr großen) Wert n_0 immer kleiner als c mal f ist. Es ist also nur das asymptotische Verhalten entscheidend (für $n_0 = 1.000.000$ spielen die ersten eine Million Funktionswerte keine Rolle) und konstante Faktoren werden durch die Multiplikation mit c ignoriert. In Abbildung 2.1 sind zwei Aufgaben zu Wachstumsordnungen dargestellt und die Antworten sind in Abbildung 2.2.



(a) (b)

Abbildung 2.1 Illustration von Wachstumsordnungen: Fragen: (a) Gehört die Funktion $g(n) = 3n\log(n+1) + 10$ zu der Funktionenklasse $O(f)$, also zu $O(N^2)$? **(b)** Gehört die Funktion $g(n) = 2\log(n^5)$ zu der Funktionenklasse $O(f)$, also zu $O(\log n)$?



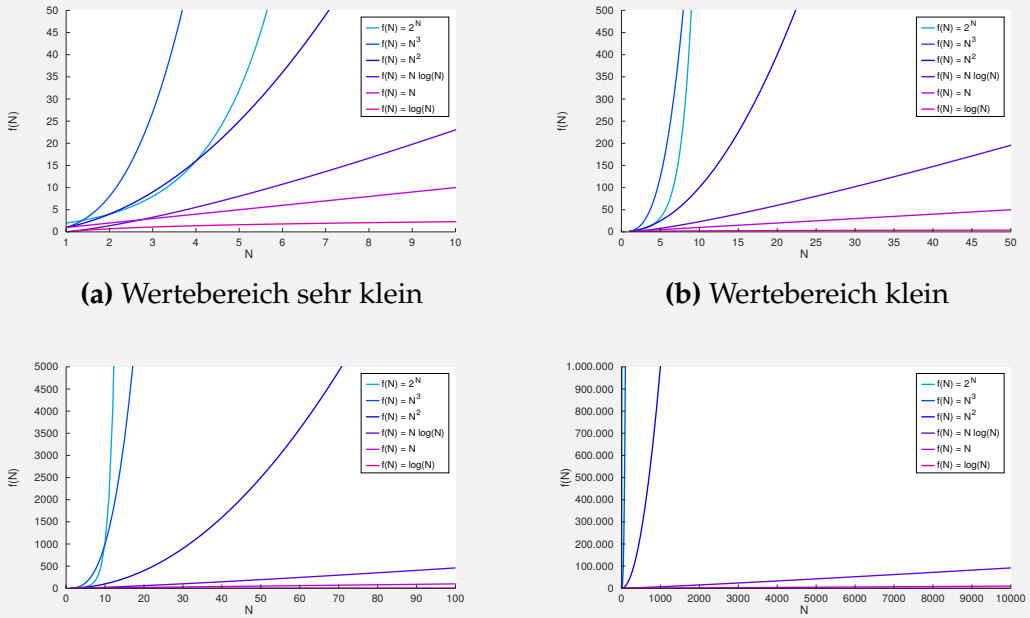
(a) (b)

Abbildung 2.2 Illustration von Wachstumsordnungen: Antworten: (a) $g \in O(n^2)$ gemäß Definition mit $c = 1$ und $n_0 = 8$. **(b)** $g \in O(\log n)$ gemäß Definition mit $c = 11$ und $n_0 = 2$.

Der Einfluss von *Offset* und konstanten Faktoren

Die Definition der Wachstumsordnungen ignoriert konstante Faktoren und Offset (Verschiebung auf der y-Achse). Dies bedeutet, dass eine Beurteilung, die nur auf Wachstumsordnungen beruht, immer kritisch hinterfragt werden muss. Praktisch ist fast nur die Wachstumsordnung relevant, wenn es um '*große* Eingaben' geht, z. B. um das Sortieren von > 10.000 Werten. Sehr große Offsets oder konstante Faktoren sind in der Rechenzeit von Algorithmen äußerst selten. Als Beispiel für den Einfluss zeigt Abbildung 2.4 den Vergleich der Funktionen $100N + 1000$ (kleinere Wachstumsordnung, großer Faktor und Offset) und N^2 (große Wachstumsordnung, kleiner Faktor und Offset).

Eine Folgerung aus der Irrelevanz konstanter Faktoren ist, dass bei Wachstumsordnungen die Basis des Logrithmus keine Rolle spielt: $O(\log_2 N) = O(\log_{10} N)$,



(a) Wertebereich sehr klein

(b) Wertebereich klein

(c) Wertebereich etwas größer

(d) Wertebereich groß

Abbildung 2.3 Wachstumsordnungen im Vergleich für unterschiedliche Wertebereiche für N. Der Eindruck hängt auch stark vom gewählten Bereich auf der y-Achse ab.

da sich Logarithmen zu verschiedenen Basen $\log_a N$ und $\log_b N$ nur um einen von N unabhängigen Faktor unterscheiden.

Weitere Klassen von Wachstumsordnungen

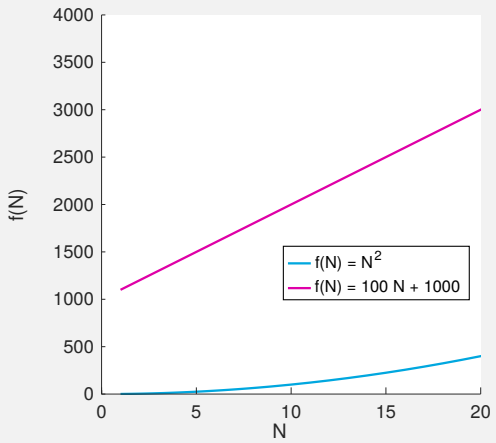
Das Gegenstück zu der Klasse $\boldsymbol{O}(f)$ die $\leq$ entspricht, ist die Entsprechung von $\geq$ bzgl. Wachstumsordnungen:

$$\boldsymbol{\Omega}(f) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c > 0\ \exists n_0\ \forall n > n_0 \quad g(n) > c \cdot f(n)\}$$

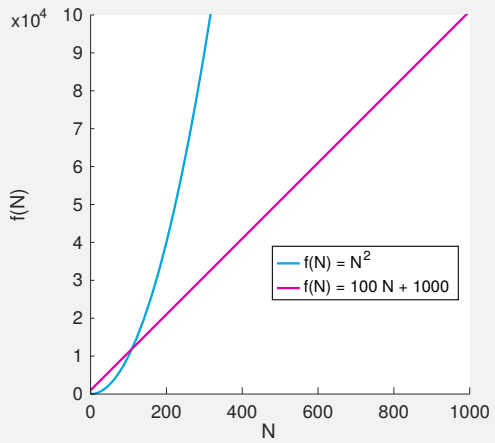
Funktionen g, die sowohl in $\boldsymbol{O}(f)$ als auch in $\boldsymbol{\Omega}(f)$ sind, haben dieselbe Wachstumsordnung wie f. Für die Gleichheit von Wachstumsordnungen wird die Klasse $\boldsymbol{\Theta}(f)$ definiert:

$$\boldsymbol{\Theta}(f) = \boldsymbol{O}(f) \cap \boldsymbol{\Omega}(f)$$

Für die Funktion $T(N) = 10N^2 + 5N + 27$ gilt also $T \in \boldsymbol{\Theta}(N^2)$. Dies wird auch kurz $10N^2 + 5N + 27 \in \boldsymbol{\Theta}(N^2)$ geschrieben.



(a) Im kleinen Wertebereich haben Offset und konstanter Faktor einen starken Einfluß.



(b) Bei größeren N zählt im Wesentlichen die Wachstumsordnung.

Abbildung 2.4 Der Einfluss von Offset und konstanten Faktoren: Es werden die Funktionen $100N + 1000$ (kleinere Wachstumsordnung, großer Faktor und Offset) und N^2 (kleinere Wachstumsordnung, großer Faktor und Offset) verglichen.

2.2 Wachstumsordnungen von Laufzeit und Speicherbedarf

Die im vorigen Abschnitt eingeführten Wachstumsordnungen $\boldsymbol{O}(f)$, $\boldsymbol{\Theta}(f)$ und $\boldsymbol{\Omega}(f)$ werden benutzt, um die Laufzeit und den Speicherbedarf von Programmen und Algorithmen zu charakterisieren. Lässt sich z. B. die Laufzeit eines Programmes bei einer Eingabegröße N durch $T(N) = 15\,N^2 + 3\,N + 10$ Sekunden abschätzen, dann spricht man von einer **Laufzeit in $\boldsymbol{O}(N^2)$**, da $T(N) \in \boldsymbol{O}(N^2)$. Dabei spielt die Einheit der Zeitmessung keine Rolle (ob Millisekunden, Minuten oder Tage), da dies durch den konstanten Faktor c in der Definition von $\boldsymbol{O}(f)$ ausgeglichen wird. Analog kann man den Speicherbedarf als Wachstumsordnung ausdrücken.

Für eine Laufzeitanalyse stellt man zunächst die Abhängigkeit von der Eingabe fest, z. B. die Länge N einer Liste, die sortiert werden soll. Es können auch mehrere Eingabegrößen relevant sein, z.B. die Anzahl der Knoten V und die Anzahl der Kanten E eines Graphes. Bei manchen Algorithmen spielt nicht nur die Anzahl der Eingabedaten, sondern auch die konkreten Werte eine Rolle. Manche Sortieralgorithmen laufen z.B. schneller, wenn die Daten schon halbwegs vorsortiert sind. Dies führt zu einer Unterscheidung von Laufzeit im Durchschnitt und im *Worst Case*. Ein anderer Eingabedaten-abhängiger Aspekt wird in der amortisierten Laufzeitanalyse in Abschnitt 2.3 berücksichtigt.

Wir führen als Beispiel eine Laufzeitanalyse der Methode `TwoSumCount(int[] a)` durch, und zwar zunächst empirisch (durch Ausprobieren), dann analytisch (durch Berechnung). Diese Funktion zählt in dem `int` Array `a` die Paare `a[i]`, `a[j]` (für beliebige `i` und `j`), deren Summe `0` ergibt.

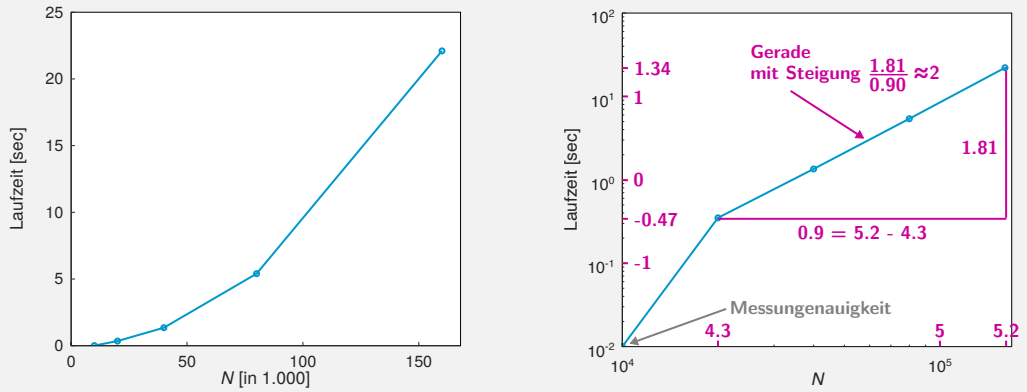
2.2.1 Empirische Laufzeitanalyse von `TwoSum`

Listing 2.1 `TwoSum` als Beispielprogramm für eine empirische Laufzeitanalyse

```
1  public class TwoSum
2  {
3    public static int count(int[] a) {
4      int N = a.length;
5      int counter = 0;
6      for (int i = 0; i < N; i++) {
7        for (int j = i+1; j < N; j++) {
8          if (a[i] + a[j] == 0) {
9            counter++;
10         }
11       }
12     }
13     return counter;
14   }
15
16   public static void main(String[] args) {
17     int N = Integer.parseInt(args[0]);
18     int[] a = new int[N];
19     for (int i = 0; i < N ; i++) {
20       a[i]= -10000 + (int)(20000*Math.random());
21     }
22
23     long start = System.currentTimeMillis();
24     int counter = count(a);
25     long stop = System.currentTimeMillis();
26     System.out.println("Count: " + counter + " in " +
27         (stop-start)/1000.0 + "s");
28   }
29 }
```

Für Felder der Länge $N = 10\text{k}, 20\text{k}, 40\text{k}, 80\text{k}$ und 160k wurden folgende Laufzeiten gemessen: 0.01 s, 0.34 s, 1.35 s, 5.40 s, 22.11 s. Der genaue funktionale Zusammenhang zwischen Eingabegröße und Laufzeit ist aus den Datenpunkten nicht ersichtlich (ob quadratisch, kubisch, ...).
Hier hilft manchmal der Trick, eine logarithmische Skala auf beiden Achsen zu verwenden. Wenn die Laufzeit durch eine Potenzfunktion bestimmt ist, hat der

log-log Plot einen linearen Verlauf, wobei die Steigung der Ordnung der Potenzfunktion entspricht. In Fall von `TwoSum` zeigt die Steigung 2 in Abbildung 2.5 (b) einen quadratischen Zusammenhang der Laufzeit in Abbildung 2.5 (a) an.



(a) Laufzeit auf normalen (linearen) Achsen

(b) Laufzeit im log-log Plot

Abbildung 2.5 Empirische Laufzeitanalysis von `TwoSum`: Der lineare Verlauf der Laufzeit im log-log Plot in **(b)** hat die Steigung 2. Dies korrespondiert zu einem quadratischen Zusammenhang (zweite Potenz) in **(a)**.

Um einzusehen, warum der log-log Trick klappt, nehmen wir eine unbekannte Potenzfunktion $f(N)$ als gegeben an. Die Steigung einer Funktion $g(N)$ zwischen den Stellen N_1 und N_2 ist:

$$\frac{g(N_2) - g(N_1)}{N_2 - N_1}$$

Um die Steigung der Funktion f im log-log Plot zu bestimmen, wird der Logarithmus sowohl auf die Funktionswerte $f(N_i)$ (y-Achse) als auch auf die Stellen N_i (x-Achse) angewendet. Die Steigung ist also

$$\frac{\log(f(N_2)) - \log(f(N_1))}{\log(N_2) - \log(N_1)}$$

Nun gehen wir davon aus, dass die Funktion f eine Potenzfunktion $f(N) = N^k$ (mit unbekanntem k) ist und rechnen die Steigung im log-log Plot aus:

$$\frac{\log(f(N_2)) - \log(f(N_1))}{\log(N_2) - \log(N_1)} = \frac{\log(N_2^k) - \log(N_1^k)}{\log(N_2) - \log(N_1)} = \frac{k\log(N_2) - k\log(N_1)}{\log(N_2) - \log(N_1)}$$

$$= k\,\frac{\log(N_2) - \log(N_1)}{\log(N_2) - \log(N_1)} = k$$

Das Ergebnis ist k. Also lässt sich tatsächlich der Exponent k einer Potenzfunktion f aus der Steigung von f im log-log Plot bestimmen.

2.2.2 Analytische Laufzeitanalyse von `TwoSum`

Wir führen zunächst eine detaillierte analytische Laufzeitanalyse durch, bei der die Ausführungshäufigkeit aller Zeilen berücksichtigt wird. Danach leiten wir einfachere Verfahrensweisen für Situationen ab, wenn nur die *Wachstumsordnung* der Laufzeit bestimmt werden soll. In Abschnitt 2.4 werden viele Beispiele besprochen.

Listing 2.2 Analytische Laufzeitanalyse von **`TwoSum`**. Genau genommen müsste bei den `for`-Schleifen zwischen Initialisierung, Vergleich und Inkrement differenziert werden.

```
1  public static int TwoSumCount(int[] a)
2  {                                  // Häufigkeit der Ausführung jeder
3                                     // Code Zeile:
4    int N = a.length;                // 1
5    int counter = 0;                 // 1
6
7    for (int i = 0; i < N; i++)      // N
8      for (int j = i+1; j < N; j++)  // N(N-1)/2
9        if (a[i] + a[j] == 0)        // N(N-1)/2
10         counter++;                 // 0 bis N(N-1)/2
11                                    // (abhängig von den Daten)
12   return counter;                  // 1
13 }
```

Um die Laufzeit zu bestimmen, müssen diese Häufigkeiten mit der benötigten Ausführungsdauer jeder Codezeile multipliziert und dann aufsummiert werden. Wir interessieren uns hier nur für Wachstumsordnung. Daher können wir die unterschiedlichen Ausführungsdauern der Zeile ignorieren und zählen nur die Zeilen. In Zeile 10 zählen wir den *worst case*. Somit erhalten wir

$$1 + 1 + N + 3 \cdot N(N-1)/2 + 1 = \tfrac{3}{2}N^2 - \tfrac{1}{2}N + 3,$$

also die Wachstumsordnung N^2. Man sagt "die Laufzeit von `TwoSum` ist in $\Theta(N^2)$".

Fazit für eine Laufzeitanalyse in Wachstumsordnungen

Für eine Laufzeitanalyse in Wachstumsordnungen spielen konstante Faktoren keine Rolle. Daher kommt es hier nur darauf an, wie oft die innerste Schleife durchlaufen wird. Dagegen spielt die Anzahl der Befehle in einer Schleife oder anderswo keine Rolle, und ebenso wenig, wie aufwändig die einzelnen Befehle sind. Letzteres gilt natürlich nur, wenn die Ausführungsdauer der Befehle nicht von der Eingabe abhängt. Bei der Benutzung von Bibliotheksfunktionen z. B. aus den Java Collections muss die Laufzeit in der Dokumentation recherchiert und entsprechend berücksichtigt werden, siehe auch Abschnitt 2.5. Bei Programmabschnitten, die hintereinander ausgeführt werden, braucht nur das Maximum der Laufzeiten der einzelnen Abschnitte berücksichtigt zu werden.
In Abschnitt 2.4 wird die Vorgehensweise in ein paar Regeln zusammengefasst und an Beispielen erläutert.

2.3 Amortisierte Laufzeitanalyse

Bei manchen Algorithmen kann die Laufzeit von Fall zu Fall stark schwanken. Dies ist z. B. der Fall, wenn statische Datenstrukturen für dynamische Prozesse benutzt werden. In diesen Fällen wird häufig eine **amortisierte Laufzeitanalyse** verwendet, um eine mittlere Laufzeit zu erhalten.
Dies lässt sich am besten an einem konkreten Beispiel erklären. Wir betrachten den Fall, dass die Datenstruktur Stapel mit Hilfe eines Arrays (an Stelle einer verketteten Liste) implementiert wird. Die Größe des Arrays ist statisch und muss angepasst werden, falls ein Element beim Hinzufügen nicht mehr in das Array hineinpasst. Beim Entnehmen von Elementen aus dem Stapel ist eine dynamische Verkleinerung des Arrays nicht zwingend notwendig, aber im Sinne der Speichereffizienz sinnvoll.
Es ergeben sich dann beim schrittweisen Hinzufügen von Elementen, jeweils in den Fällen der Array Anpassung deutlich längere Laufzeiten. Es wird Speicher für das neue Array alloziert und die Daten werden von dem alten in das neue Array kopiert.

Amortisierte Laufzeitanalyse

Falls ein Verfahren bei der gleichen Operation in unterschiedlichen Situationen unterschiedliche Laufzeiten aufweist, bietet sich eine *amortisierte Laufzeit-*

> *analyse* an. Dabei wird die Laufzeit über die unterschiedlichen Situationen gemittelt.

Im Fall des Arrays ist es eine häufige Strategie, die Größe zu verdoppeln (bzw. zu halbieren), wenn eine Anpassung erforderlich ist. Die `push()` Operation eines Stacks hat im Normalfall eine konstante Laufzeit, siehe Abschnitt 2.5. Wird das Array von Größe N auf $2N$ erweitert, ergeben sich N Schleifendurchläufe zum Kopieren der Daten. Die amortisierte Laufzeitanalyse sieht daher wie folgt aus. Nehmen wir N als Zweierpotenz an, so ist beim schrittweisen Hinzufügen von N Elementen die Gesamtanzahl der Schleifendurchläufe

$$1 + 2 + 4 + 8 + \cdots + N = 2N - 1$$

Es ergibt sich also eine durchschnittliche Anzahl von $\frac{2N-1}{N}$, also knapp 2 Schleifendurchläufen pro `push()` Operation. Die amortisierten Kosten sind also in $\boldsymbol{O}(1)$.

2.4 Hinweise für die Praxis mit vielen Beispielen

Laufzeitanalysen können manchmal selbst bei kurzen und einfachen Algorithmen und Programmen sehr komplex sein und gehen dann weit über den Inhalt dieses Skriptes hinaus. Ein solches Beispiel sind die Verfahren mit Pfadkompression für die `UnionFind` Struktur. In diesen Fällen werden in diesem Skript die Ergebnisse aus der Literatur zitiert und nicht die Laufzeiten hergeleitet.
In vielen Fällen lässt sich die Größenordnung der Laufzeit recht einfach bestimmen. Hier wird erklärt, wie das praktisch geht. Dabei wird immer eine *worst-case* Analyse gemacht.
Zunächst muss immer beachtet werden, wovon die Laufzeit abhängig, bzw. welche Abhängigkeit von Interesse ist. Im einfachsten Fall ist es der Wert der Eingabe. Bei einer Funktionen `f1(int N)` hängt die Laufzeit von dem Wert des Parameters `N` ab.
Die Algorithmen, deren Laufzeit in diesem Skript bestimmt werden, bekommen eine komplexere Datenstruktur als Eingabe, wie z. B. eine Liste oder einen gewichteten Graphen. In diesen Fällen betrachtet man die Laufzeit in Abhängigkeit der Länge der Liste, bzw. der Anzahl der Knoten und Anzahl der Kanten des Graphen. Das Grundprinzip bei der Bestimmung der Laufzeit ist dasselbe wie bei der einfachen Variablen. Deswegen konzentrieren wir uns in diesem Ratgeber auf den einfachen Fall.
Am liebsten möchte man die genaue Abhängigkeit der Laufzeit kennen, also eine Funktion, die zu jeder Eingabe `N` angibt, wieviele Rechenoperationen benötigt werden. Oft begnügt man sich jedoch mit der viel gröberen Angabe, welcher Wachstumsordnung diese Funktion angehört, siehe Abschnitt 2.1. Wie im Fazit

von Abschnitt 2.2.2 dargelegt, genügt dabei die Betrachtung der am häufigsten ausgeführten Codezeile.

2.4.1 Abschätzung der Laufzeit von Funktionen ohne Rekursion

Nun zu dem konkreten Vorgehen. Es geht im Prinzip nur darum festzustellen, wie häufig die am häufigsten ausgeführte Codezeile ausgeführt wird. Dabei müssen geschachtelte Schleifen und rekursive Aufrufe analysiert werden.
Wenn es keine Rekursion gibt, kann mit den folgenden Regeln die Komplexität der Funktion, also die Wachstumsordnung der Laufzeit bestimmt werden. Um die Regeln einfach formulieren zu können, benutzen wir hier den Begriff 'Komplexität' sowohl für Wachstumsordnungen $\Theta(f)$ / $O(f)$ als auch für Stellvertreterfunktion der Wachstumsklasse (also f an Stelle von $\Theta(f)$ / $O(f)$). In welchen Fällen die Regel eine scharfe Abschätzung mittels Θ und wann nur eine obere Grenze mit O liefern, wird in den Hinweisen erläutert.

Laufzeitabschätzung von Funktionen ohne Rekursion

1. Bei Codeabschnitten, die hintereinander ausgeführt werden, wird das Maximum der Komplexitäten genommen.
2. Die Komplexität einer Schleife ist die maximale Anzahl der Schleifendurchläufe (multipliziert mit der Komplexität des Schleifenrumpfs).
3. Spezialfall: Wird die Laufvariable in jedem Schleifendurchlauf halbiert (bis sie 0 wird), so ist die Komplexität der Logarithmus des Anfangswertes. Entsprechendes gilt, wenn die Variable von 1 angefangen bis zu einem Grenzwert verdoppelt wird.
4. Bei geschachtelten Schleifen werden die Komplexitäten multipliziert.
5. In jedem Schritt kann der einfachste Vertreter einer Wachstumsordnung genommen werden, insbesondere also Konstanten weggelassen werden.

Hinweise:

- Falls bei inneren Schleifen die Anzahl der Durchläufe von der Laufvariable einer äußeren Schleife abhängt und man in Regel 2 einen Maximalwert nimmt, bekommt man nur eine obere Abschätzung der Komplexität, also in der O-Hierarchie. Dies wird unten in den Beispielen `f5` und `f6` verdeutlicht.
- Falls obiger Fall nicht vorliegt, ist die resultierende Komplexität in der Θ-Hierarchie.

- Da die Basis des Logarithmus bei Wachstumsordnungen keine Rolle spielt, gilt Regel 3 auch, wenn die Laufvariable durch einen anderen konstanten Wert als 2 dividiert, bzw. mit ihm multipliziert wird.

Nun werden die Regeln zur Bestimmung der Wachstumsordnungen von Laufzeiten an einigen Beispielfunktionen angewendet. Dabei werden in Kommentaren im Code bei Schleifen die Einzelkomplexitäten und in der am häufigsten ausgeführten Zeile die Gesamtkomplexität (mit Präfix `-->`) notiert.

```
1 public static int f1(int N) {
2   int x = 0;
3   for (int i = 0; i < N; i++)     // Θ(N)
4     x+= i*i;                      // --> Θ(N)
5   return x;
6 }
```

`f1(N)`: In der Funktion `f1` gibt es nur eine Schleife und sie wird `N`-mal durchlaufen. Zeile 4 wird `N`-mal ausgeführt. Daher ist die Wachstumsordnung der Laufzeit nach Regel 2 in $\Theta(N)$. An dieser Laufzeit ändert sich nach Regel 5 nichts, wenn die Schleife bis `N/2` oder `2*N` oder gar `100*N + 1000` läuft. Wenn die Grenze der Schleife allerdings `N*N` ist, ändert sich die Wachstumsordnung auf $\Theta(N^2)$.

```
1 public static int f2(int N) {
2   int x = 0;
3   for (int i = 0; i < N; i++)       // Θ(N)
4     for (int j = 0; j < N; j++)     // Θ(N)
5       x+= i*j;                      // --> Θ(N^2)
6   return x;
7 }
```

`f2(N)`: Nun haben wir zwei geschachtelte Schleifen. Die äußere Schleife läuft für `i` über `N` unterschiedliche Werte. Für jeden Wert läuft die innere Schleife über `j` ebenfalls über `N` Werte. Nach den Regeln 2 und 4 erhält man die Komplexität $\Theta(N^2)$. Wie im ersten Beispiel ändert sich daran gemäß Regel 5 nichts, wenn die Grenzen der Schleifen nicht `N`, sondern eine lineare Funktion von `N` sind. Aber wie sieht es aus, wenn die Schleifengrenze gar nicht von `N` abhängt:

```
1 public static int f3(int N) {
2   int x = 0;
3   for (int i = 0; i < N; i++)       // Θ(N)
4     for (int j = 0; j < 10; j++)    // Θ(1)
5       x+= i*j;                      // --> Θ(N)
```

```
6   return x;
7 }
```

f3(N): Wenn die Anzahl der Schleifendurchläufe nicht von N abhängt, liegt eine konstante Komplexität $\Theta(1)$ vor. Durch Regel 4 erhalten wir als Gesamtkomplexität $\Theta(N)$.

```
1 public static int f4(int N) {
2   int x = 0;
3   for (int i = 0; i < N; i++)          // Θ(N)
4     for (int j = 0; j <= i; j++)       // O(N)
5       x+= i*j;                         // --> O(N^2), auch Θ(N^2) siehe Text
6   return x;
7 }
```

f4(N): Nun wird es etwas schwieriger. Die Grenze der inneren Schleife hängt vom Wert der Laufvariablen der äußeren Schleife ab. Mit der Abschätzungsregel 2 erhalten wir die Komplexität $O(N)$für beide Schleifen, da die innere Schleife im 'schlimmsten' Fall `N-1` mal ausgeführt wird (größtmöglicher Wert von `i`). Wir erhalten $O(N^2)$ mit Regel 4 als Abschätzung für die Gesamtkomplexität. Nun stellt sich die Frage, ob die Komplexität auch tatsächlich $\Theta(N^2)$ ist, oder die Grenze niedriger ist. Hierfür muss man sich von den einfachen Abschätzungsregeln abwenden und schrittweise überlegen: Für `i=0` wird die innere Schleife 1 mal ausgeführt, für `i=1` läuft `j` von 0 bis 1, also 2 mal, für `i=2` wird sie 3 mal ausgeführt und so weiter. Insgesamt ergibt sich

$$1 + 2 + 3 + \cdot + N = N(N+1)/2 = {}^1\!/_2 N^2 + {}^1\!/_2 N,$$

also stimmt die Wachstumsordnung $\Theta(N^2)$.

```
1 public static int f5(int N) {
2   int x = 0;
3   for (int i = 1; i < N; i*=2)         // Θ(log N)
4     x+= i*i;                           // --> Θ(log N)
5   return x;
6 }
```

f5(N): Bei diesem Beispiel ist das Besondere, dass die Laufvariable nicht um 1 erhöht, sondern verdoppelt wird. Nach Regel 3 ist die Komplexität $\Theta(\log N)$. Die Gültigkeit dieser Regel wird durch die folgenden Überlegungen veranschaulicht: Für $N = 32$ läuft die Variable `i` über die Werte 1, 2, 4, 8, 16, bzw. 2^0, 2^1, 2^2, 2^3, 2^4. Also wird Zeile 4 dabei 5-mal ausgeführt. Die Laufvariable durchläuft die Zweierpotenzen kleiner N, also ist die Anzahl der Schleifendurchläufe der Zweierlogarithmus von N. Damit wird bestätigt, dass die Laufzeit von `f5` in

$\Theta(\log N)$ ist. Da es uns nur auf die Größenordnung ankommt, muss nicht überlegt werden, was hier genau passiert, wenn `N` keine Zweierpotenz ist. Es ist klar, dass sich hierbei nur eine Änderung um ± 1 im Vergleich zur nächstgrößeren oder nächstkleineren Zweierpotenz ergibt.

```
1 public static int f6(int N) {
2   int x = 0;
3   for (int i = 1; i < N; i*=2)          // Θ(log N)
4     for (int j = 0; j < i; j++)         // O(N)
5       x+= i*j;                          // --> O(N log N), allerdings Θ(N)
6   return x;                             // siehe Text
7 }
```

`f6(N)`: Dies ist nun das kniffeligste Beispiel, jedenfalls wenn die genaue Komplexität bestimmt werden soll. Eine Abschätzung erhält man so: Die äußere Schleife besitzt gemäß 3 die Komplexität $\Theta(\log N)$ und für die innere liefert 2 die Abschätzung $O(N)$, wenn nur der schlimmste Fall berücksichtigt wird. Die Gesamtkomplexität kann laut Regel 4 durch $O(N \log N)$ abgeschätzt werden. Wieder stellt sich die Frage, ob die Laufzeit von `f6(N)` auch in $\Theta(N \log N)$ ist, oder ob die Grenze schärfer ist. Dazu muss die Anzahl der Schleifendurchläufe der beiden geschaltelten Schleifen gemeinsam betrachtet werden. Die Variable `i` durchläuft die Werte 1, 2, 4, 8, 16, ... bzw. $2^0, 2^1, 2^2, 2^3, 2^4, \ldots$ und die innere Schleife wird i-mal ausgeführt. Insgesamt ist die Anzahl der Ausführung von Zeile 5 somit 1+2+4+8+16+... Diese Summe aufeinander folgender Zweierpotenzen ist gleich der nächsten Zweierpotenz minus 1. Dies kann man sich gut in der Binärdarstellung vorstellen: $1_2 + 10_2 + 100_2 + 1000_2 + 10000_2 = 11111_2 = 100000_2 - 1_2$. Wenn N also den Wert einer Zweierpotenz 2^n hat, ist die Anzahl der Ausführungen von Zeile 5 gleich $2^n - 1 = N - 1$. Die Laufzeit von `f6(N)` ist damit in $\Theta(N)$.

```
1 public static int f7(int N) {
2   int x = 0;
3   for (int i = N; i > 0; i \= 2)        // Θ(log N)
4       x += i*i;
5   for (int j = 0; j < N; j++)           // Θ(N)
6       x += j*j;                         // --> Θ(N)
7   return x;
8 }
```

`f7(N)`: Die erste Schleife besitzt Komplexität $\Theta(\log N)$ gemäß 3 und die zweite $\Theta(N)$ gemäß 2. Diese Schleifen sind nicht geschachtelt, sondern werden hintereinander ausgeführt. Daher erhalten wir mit Regel 1 die Gesamtkomplexität $\Theta(N)$.

2.4.2 Abschätzung der Laufzeit von rekursiven Funktionen

Zur Laufzeitabschätzung von rekursiven Funktionen stellt man sich den Rekursionsbaum vor und schätzt die Laufzeit wie folgt ab:

Laufzeitabschätzung von Funktionen mit Rekursion

1. Man multipliziert die Anzahl der Knoten des Rekursionsbaumes mit der Komplexität des restlichen Codes.
2. Die Größenordnung der Knotenanzahl ist $\Theta(h)$ für einen *unverzweigten* Baum der Höhe h und sie ist $\Theta(k^h)$, wenn der Baum k Verzweigungen besitzt.
3. Rekursionsaufrufe, bei denen das Argument um einen konstanten Wert verringert wird, führen auf eine lineare Höhe des Rekursionsbaums.
4. Rekursionsaufrufe, bei denen das Argument durch einen konstanten Wert dividiert wird, führen auf eine logarithmische Höhe des Rekursionsbaums.

Hinweis:

- Falls der Baum nicht an allen Knoten (außer den Blättern) k-viele, sondern eine variable Anzahl an Verzweigungen $\leq k$ hat, erhält man nun eine obere Abschätzung mit O, siehe Beispiel `r3(N)`.
- Die Regeln stellen eine vereinfachte Vorgehensweise dar, die für die Beispiele im Skript **(und zur Bearbeitung der Klausuraufgaben)** ausreichend ist.

Die Anzahl der Verzweigungen k entspricht der Anzahl paralleler rekursiver Aufrufe in dem Funktionscode. Von dem 'restlichen Code' ist häufig nur der Basisfall relevant (Code, der bei Rekursionsabbruch ausgeführt wird). In einfachen Beispielen hat er konstante Laufzeit, kann also ignoriert werden.

> Bei komplexeren Fällen rekursiver Funktionen kann der sogenannte Hauptsatz der Laufzeitfunktionen (bzw. das *master theorem for the analysis of algorithms*) verwendet werden, siehe [Cormen et al., 2013].

```
1 public static int r1(int N) {
2   if (N == 0)
3     return 1;
4   return r1(N-1) + 1;      // keine Verzweigung -> k=1; lineare Höhe
5 }
```

r1(N): Die Funktion ruft sich selbst mit dem um 1 verminderten Argument auf, also liefert Regel 3 $\Theta(N)$ für die Baumhöhe. Es gibt keine Verzweigungen ($k = 1$) und alle Zeilen für sich haben konstante Laufzeit, also ist die Komplexität $\Theta(N)$ gemäß Regeln 1 und 2.

```
1 public static int r2(int N) {
2   if (N <= 1)
3     return 1;
4   return r2(N/2) + 1;     // -> k=1, logarithmische Baumhöhe
5 }
```

r2(N): Die Funktion ruft sich selbst mit halbiertem Argument auf, also liegt laut 4 die Baumhöhe in $\Theta(\log N)$. Ansonsten ist die Lage wie oben: keine Verzweigung und konstante Laufzeit der einzelnen Zeilen, also ist die Gesamtkomplexität $\Theta(\log N)$.

```
1 public static int r3(int N) {
2   if (N <= 1)
3     return 1;
4   return r3(N-1) + r3(N-2) + r3(N-3);     // drei Verzweigungen -> k=3
5 }
```

r3(N): Die Funktion besitzt $k = 3$ parallele rekursive Aufrufe und die Höhe des Rekursionsbaumes linear. Teile des Baumes besitzen durch die rekursiven Aufrufe mit `N-2` und `N-3` kürzere Pfade, daher können wir nur eine obere Abschätzung mit der O-Komplexität erzielen. Wir erhalten die Abschätzung der Laufzeit durch $O(3^N)$ gemäß Regel 2.

```
1 public static int r4(int N) {
2   if (N <= 0)
3     return 1;
4   return r4(N/2) + r4(N/2);
5 }
```

r4(N): Die Funktion besitzt eine zweifache Verzweigung des Rekursionsbaumes ($k = 2$), sowie eine logarithmische Baumhöhe $\log_2 N$ wegen der Halbierung des Argumentes, siehe Regel 4. Die Anzahl der Knoten ist folglich in $\Theta(2^{\log_2 N}) = \Theta(N)$. Die Laufzeit von `r4(N)` ist also in $\Theta(N)$.
Wem diese Rechnung Unbehagen bereitet, kann die Bestimmung per Hand zum Vergleich heranziehen: Der Aufruf von `r4(N)` führt zu 2 Aufrufen von `r4(N/2)` und in nächster Ebene zu 4 Aufrufen von `r4(N/4)`, dann zu 8 Aufrufen von `r4(N/8)` usw. Wenn N eine Zweierpotenz ist, wird Zeile 3 genau N-mal ausgeführt. (Wenn N keine Zweierpotenz ist, ist die Anzahl gleich der nächst

kleineren Zweierpotenz.) Auch auf diese Weise erhält man die Komplexität $\Theta(N)$ für `r4(N)`.

2.4.3 Laufzeit von Funktionen mit komplexeren Datenstrukturen

Schauen wir uns zunächst den Fall an, dass die Laufzeit der Funktion von zwei Variablen abhängt:

```
1 public static int g1(int V, int E) {
2   int x = 0;
3   for (int v = 0; v < V; v++)
4     for (int e = 0; e < E; e++)
5       x+= v*e;
6   return x;
7 }
```

`g1(V, E)`: Bei Funktionen mit zwei Variablen muss nichts Besonderes beachtet werden. Die äußere Schleife wird V-mal durchlaufen und bei jedem Durchgang wird die innere Schleife E-mal durchlaufen. Die Laufzeit ist folglich in $\Theta(V * E)$.

Bisher haben wir nur Fälle betrachtet, bei denen die einfache Ausführung einer einzelnen Zeile eine konstante Laufzeit hat. Falls in einer Zeile Funktionen aufgerufen werden, deren Laufzeit nicht konstant ist, muss dies natürlich berücksichtigt werden.

```
1 public static PriorityQueue<Integer> g2(int E) {
2   PriorityQueue<Integer> pq = new PriorityQueue<>(E);
3   for (int e = 0; e < E; e++) {
4     pq.add(e);                    // log. Laufzeit in der Länge der Queue
5   }
6   return pq;
7 }
```

`g2(E)`: Zeile 4 wird E Mal aufgerufen. Nun ist noch die Laufzeit von Zeile 4 zu beachten. Die Methode `add()` einer `PriorityQueue` hat eine Laufzeit in $\Theta(\log N)$, wenn die Warteschlange N Elemente besitzt, siehe Abschnitt 3.1.2. In unserer Funktion `g2(E)` wächst die Warteschlange auf Größe E an, also ist die Laufzeit von Zeile 4 durch $\Theta(\log(E))$ beschränkt. Damit haben wir insgesamt $O(E \log E)$ als Abschätzung für die Laufzeit von `g2(E)`. Die Laufzeit von Zeile 4 wächst mit dem Längerwerden der Warteschlange an. Es ist möglich, dass die Gesamtlaufzeit von `g2(E)` nicht in $\Theta(E \log E)$, sondern in einer niedrigeren Wachstumsklasse ist. Da dies hier nicht zu sehen ist, begnügen wir uns in diesem Fall mit der O-Abschätzung.

Es folgt noch ein Beispiel, das dem Skript inhaltlich vorausgreift. Es benutzt die Datenstruktur von gerichteten Graphen. Der Aspekt, der hier verdeutlicht werden soll, ist auch ohne genauere Kenntnis jener Datenstruktur zu verstehen.

```
1 public static Queue<DirectedEdge> g3(Digraph G) {
2   Queue<DirectedEdge> q = new LinkedList<>();
3   for (int v = 0; v < G.V(); v++) {
4     for (DirectedEdge e : G.incident(v)) {
5       q.add(e);                                // konstante Laufzeit
6     }
7   }
8   return pq;
9 }
```

g3(G): Für einen `Digraph G` gibt `G.V()` die Anzahl der Knoten zurück. Für einen Knoten `v` liefert `incident(v)` alle von `v` ausgehende Kanten. Wir benutzen V und E als Variable für die Anzahl von Knoten und Kanten und interessieren uns für die Laufzeit in Abhängigkeit von diesen beiden Größen. Die äußere Schleife läuft über alle V Knoten. Die innere Schleife läuft über alle vom aktuellen Knoten `v` ausgehenden Kanten. Das könnten im schlimmsten Fall alle Kanten sein. Dadurch hätten wir eine Abschätzung nach oben von jeweils E inneren Schleifendurchläufen und damit insgesamt für `g3(G)` eine Laufzeit in $\boldsymbol{O}(VE)$ (da die Methode `add()` von `LinkedList` eine konstante Laufzeit besitzt). Lässt sich diese Abschätzung verbessern? Für einen beliebigen Knoten `v` kann man keine bessere Abschätzung für die Anzahl der inneren Schleifendurchläufe bekommen. In diesem Fall hilfe eine gesamtheitliche Betrachtung. Jede Kante `e` hat einen eindeutigen Startknoten. Daher ist klar, dass Zeile 5 für jede Kante des Graphen genau einmal ausgeführt wird. Also ist die Laufzeit in $\boldsymbol{\Theta}(E)$. Etwas versteckte Methodenaufrufe wurden allerdings noch nicht erwähnt. In Zeile 3 wird die Methode `G.V()` und in Zeile 4 `incident()` von `Digraph` aufgerufen. Sie haben beide eine konstante Laufzeit und beeinflussen daher die Laufzeit nicht.

Schließlich sollte man bei Laufzeitbetrachtungen die Initialisierung der Datenstrukturen nicht vergessen. Normalerweise ist ihre Laufzeit nicht größer als diejenige des eigentlichen Programms und sie fällt dann nicht ins Gewicht. Allerdings ist eine Laufzeitanalyse nur dann vollständig, wenn dies geprüft wurde. Vor allem bei Pseudocode, siehe nächster Abschnitt, kann sich in der Initialisierung manchmal die Hauptkomplexität verbergen, z. B. wenn dort eine Sortierung eines Datenarrays implizit versteckt ist.

2.4.4 Laufzeit von Pseudocode

Zur Bestimmung der Laufzeit von Pseudocode gelten natürlich dieselben Regeln wie oben. Hier kann allerdings die Besonderheit auftreten, dass komplexere Konzepte verwendet werden.

Wir betrachten als ein Beispiel einen Pseudocode, der zu einem gegebenen Array die maximale Summe einer zusammenhängenden Teilfolge bestimmt.

```
function p1(int[] a)
max ← −inf
for all zusammenhängende Teilfolgen t von a
  sum(t) ← sum of t
  if sum(t) > max
    max ← sum(t)
  end
end
return max
```

p1(a): Sei N die Länge des Arrays a. Hier wird über alle zusammenhängenden Teilfolgen iteriert. Die Anzahl solcher Teilfolgen liegt in $\Theta(N^2)$: Man kann die Teilfolgen durch zwei geschachtelte Schleifen der Länge N für Anfangs- und Endindex erzeugen. Das Berechnen der Summe *sum(t)* kann durch Komplexität $O(N)$ abgeschätzt werden. Dies ergibt insgesamt die Laufzeitkomplexität $O(N^3)$ für den obenstehenden Pseudocode. Mit Dynamischer Programmierung kann diese Aufgabe in linearer Laufzeit gelöst werden, was einen drastischen Laufzeitvorteil bedeutet. Dies zeigt wieder wie wichtig es ist, sich mit effizienten Implementierungen auszukennen.
Eine andere Situation ergibt sich, falls die maximale Summe *aller* Teilmengen bestimmte werden soll, also ohne dass sie zusammenhängend sein müssen. In diesem Fall liegt die Laufzeit, wie die Anzahl aller Teilfolgen in $\Theta(2^N)$. Zur Codierung könnte eine Binärzahl der Länge N dienen, bei der jede Ziffer anzeigt, ob das entsprechende Element in der Teilfolge ist oder nicht.

Abschlussbemerkung zu Θ und O

Die Wachstumsordnung $\Theta(N^2)$ ist die Klasse aller Funktionen mit quadratischem Wachstum. Oft werden Wachstumsordnungen auch als $O(\cdot)$ angegeben. Die Klasse $O(N^2)$ umfasst alle Funktionen, die *höchstens* quadratisch ansteigen. Diese Klasse umfasst also z. B. auch die linearen, logarithmischen und die konstanten Funktionen. Die Angabe mit $\Theta(\cdot)$ ist stärker eingegrenzt. Wenn Laufzeiten mit der ungenaueren $O(\cdot)$ Klasse angegeben werden, ist dies entweder eine Gewohnheit, oder es liegt daran, dass man nicht genau weiß, ob die Laufzeit nicht vielleicht schneller als die angegebene Schranke ist. Wenn also von einer Laufzeit in $O(N^2)$ die Rede ist, wird keine Aussage darüber gemacht, ob die Laufzeit nicht vielleicht schneller, z. B. in $O(N \log N)$ ist. Zwei solche Fälle wurden in den Beispielen `f5` und `f6` in Abschnitt 2.4.1 besprochen.
Selbst bei wichtigen Algorithmen ist es möglich, dass die best mögliche (*worst-case*) Laufzeit nicht bekannt ist, oder zumindest, dass die Laufzeit von der

Implementation abhängt. Im ersten Fall muss die Laufzeit in der O-Hierarchie angegeben werden. Im zweiten Fall kann eine Implementations-unabhängige Abschätzung mit O angegeben werden. Falls für eine spezielle Implementation gezeigt werden kann, dass die Wachstumsordnung scharf ist, gibt es eine Implementations-abhängige Abschätzung mit Θ. Bei Algorithmen, die eine Priority-Queue (Vorrangwarteschlange) verwenden, wird oft zunächst die Laufzeit für eine einfache Implementation der PQ mit binärem Heap angegeben als O-Abschätzung. Wird stattdessen ein effizienterer *Fibonacci Heap* verwendet, kann häufig eine bessere Laufzeit erzielt werden.

2.5 Laufzeiten bei einfachen Java Datenstrukturen

Bei der Implementierung von Algorithmen ist bei der Auswahl der Datenstrukturen die Laufzeit der Operationen zu beachten. In diesem Kapitel werden die Laufzeiten für einige häufig benutzte Datenstrukturen / Methoden aufgelistet. Für andere Datenstrukturen oder Methoden ist die Java Dokumentation zu konsultieren.
Leider sind die Laufzeiten in der Java Dokumentation nicht immer angegeben. In diesem Fall sollte man in den Quellcode schauen. Die weiter unten angegebenen Laufzeiten von Methoden der `LinkedList` sind der folgenden Seite entnommen: `http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/LinkedList.java`.

Laufzeiten von `Stack` und `Queue`

Laufzeiten für die in der Java Einführung angegebenen Implementationen von `Stack`, `Queue` und `Bag`:

Stack (*worst case*)	
`push()`	$O(1)$
`pop()`	$O(1)$

Queue (*worst case*)	
`enqueue()`	$O(1)$
`dequeue()`	$O(1)$

Bag (*worst case*)	
`add()`	$O(1)$

Der Zusatz *worst case* bei der Laufzeit ist insbesondere eine Abgrenzung zu einer *amortisierten Laufzeit* (siehe Seite 12 des AlgoDat Skriptes) bei der die Operationen manchmal eine längere Laufzeit haben, z. B. wenn nach einer gewissen Anzahl von Einfügungen ein Array vergrößert und der Inhalt kopiert werden muss.
Bei den Java Klassen `Stack` und `Queue` ist zu beachten, dass diese auch Methoden implementieren, die keine konstante sondern eine lineare Laufzeit besitzen, wie `remove()`.

Laufzeiten für ausgewählte Java *Collections*

Die `LinkedList` ist eine beidseitige Warteschlange (*double ended queue*, kurz deque). Daher unterstützt sie die Funktionalität einer Warteschlange und eines Stacks. Zu jedem `xxxFirst()` gibt es ein entsprechendes `xxxLast()` mit derselben Laufzeit. Weitere Methoden, die hier nicht aufgeführt sind, können in der Java Dokumentation recherchiert werden.

LinkedList (*worst case*)	
`addFirst(E e)`	$O(1)$
`contains(Object o)`	$O(N)$
`get(int index)`	$O(N)$
`indexOf(Object o)`	$O(N)$
`peekFirst()`	$O(1)$
`pollFirst()`	$O(1)$
`removeFirst()`	$O(1)$
`remove(int index)`	$O(N)$
`remove(Object o)`	$O(N)$
`set(int index, E e)`	$O(N)$

Altertnativ gibt es noch die Varianten `ArrayList` und `ArrayDeque`, die ähnliche Funktionalität mit Laufzeiten in derselben Wachstumsordnung zur Verfügung stellen. `ArrayList` bietet den Vorteil der direkten Indizierung und `ArrayDeque` ist ansonsten die schnellste Variante innerhalb der jeweiligen Wachstumsordnung. Allerdings haben Einfügungen bei beiden *Array* Varianten nur *amortisiert* konstante Laufzeit, während `LinkedList` *worst case* konstante Laufzeit besitzt. `ArrayList` hat für Einfügung außer am Ende sogar nur eine lineare, also langsamere, Laufzeit.

2.6 P, NP und NP-Vollständigkeit

> Die Darstellung der Komplexitätsklassen P und NP in diesem Abschnitt ist informell. Exakte Definitionen sind im Zusammenhang dieses Skriptes nicht relevant und können anderswo nachgelesen werden, z. B. in [Cormen et al., 2013].

Die **Komplexitätsklasse P** umfasst diejenigen Probleme, die von einem Algorithmus mit Laufzeit in $O(p(n))$ für ein Polynom $p(n)$ in der Eingabegröße n gelöst werden können. In einer ganz groben Einteilung könnte man sagen, das Probleme in P halbwegs effizient lösbar sind. Andererseits ist es nicht angebracht für Laufzeiten mit höheren Potenzen von Effizenz zu sprechen. Daher formuliert

man es besser anders herum: Probleme, die nicht in P enthalten sind, sind für größere Eingaben n meist praktisch nicht lösbar.

Dazu ein Beispiel: Ein Algorithmus mit einer Laufzeit von 2^n Rechenoperationen (also außerhalb der Komplexitätsklasse P; $f(n) = 2^n$ ist kein Polynom!) benötigt bei einer Eingabegröße von $n = 100$ selbst auf einem Computer mit 100 Millionen Rechenoperationen pro Sekunde mehr als 10^{14} Jahre (ca. 10.000 mal das Alter unseres Universums). Für viele Anwendungen ist eine Eingabegröße von $n = 100$ klein!

Die **Komplexitätsklasse NP** umfasst die Probleme, bei denen in polynomieller Laufzeit festgestellt werden kann, ob ein Lösungskandidat tatsächlich korrekt ist. Wie lange es dauert, eine korrekte Lösung zu finden, spielt keine Rolle, nur die Dauer der Verifikation eines gegebenen Lösungskandidaten.

Ein Problem $\mathcal{X}$ heißt **NP-schwer** (*NP-hard*), wenn jedes beliebiges Problem aus NP in polynomieller Zeit auf eine Lösung von $\mathcal{X}$ zurückgeführt werden kann. Diese Polynomzeitreduktion von einem NP-Problem $\mathcal{Y}$ auf ein Problem $\mathcal{X}$ bedeutet: Es gibt einen Algorithmus, der eine Eingabe für $\mathcal{Y}$ in polynomieller Zeit in eine Eingabe für $\mathcal{X}$ umwandelt, und einen Algorithmus, der eine $\mathcal{X}$-Ausgabe in polynomieller Zeit in eine $\mathcal{Y}$-Ausgabe umwandelt. Durch diese Umwandlung hat man also die Möglichkeit, ein $\mathcal{Y}$-Problem, durch einen $\mathcal{X}$-Algorithmus lösen zu lassen, wobei die Laufzeit eines solchen Verfahrens zur $\mathcal{Y}$-Lösung 'nur' um eine polynomielle Wachstumordnung über der Laufzeit des $\mathcal{X}$-Algorithmus liegt. Sollte also $\mathcal{X}$ in polynomieller Laufzeit zu lösen sein, so gilt dies auf für $\mathcal{Y}$.

Ein Problem heißt **NP-vollständig**, wenn es zu NP gehört und NP-schwer ist.

Nach der Definition von NP gilt offensichtlich P$\subseteq$NP. Es wird von den allermeisten Wissenschaftlern vermutet, dass P$\neq$NP gilt. Aber dies konnte bisher nicht bewiesen werden (Millenium Problem, 1.000.000 $ Preisgeld).

NP-vollständige Probleme stellen Prüfsteine für die P=NP Hypothese dar. Denn wenn für ein einizges NP-vollständiges Problem ein polynomieller Algorithmus gefunden wird, ist P=NP gezeigt und viele relevante Problemstellungen, die zzt. praktisch nicht lösbar sind, könnten dadurch lösbar werden.

Daher sind NP-vollständige Probleme interessante Herausforderungen. Dies gilt auch unabhängig von der P=NP Frage. Insbesondere werden für solche Probleme oft ›Ersatzansätze‹ gesucht:

- Ansätze, die in bestimmten praktischen Fällen eine effiziente Lösung finden, auch wenn der *worst-case* exponentiell bleibt. Dies wird beispielweise in manchen Fällen durch heuristische Verfahren erreicht.

- Ansätze, die in effizienter Laufzeit suboptimale, approximative Lösungen bestimmen, siehe approximative Algorithmen

Beispiele für NP-vollständige Probleme

- **Problem des Handlungsreisenden:** Finde in einem vollständigen, gewichteten Graphen einen Zyklus mit minimalem Gewicht, der jeden Knoten genau einmal enthält (*Traveling Salesman Problem*; TSP).
- **Hamiltonpfad:** Finde einen Pfad, der jeden Knoten eines gegebenen Graphen genau einmal besucht. (Ebenso 'Hamiltonzyklus')
- **Maximaler Schnitt:** Bestimme in einem gewichteten Graphen einen Schnitt mit maximalem Gewicht.
- **0/1-Rucksack Problem**, auch bei Beschränkung auf ganzzahlige Gewichte

KAPITEL 3

Datenstrukturen

3.1 Vorrangwarteschlange

> Da das Thema *Vorrangwarteschlange* und die zugrundliegenden binären Heaps in der Vorlesung IntroProg behandelt werden, ist die Darstellung hier kurz gehalten und nur als Wiederholung gedacht.

Die Datenstruktur Vorrangwarteschlange (*priority queue*, Prioritätenschlange) ist eine spezielle Form der Warteschlange (*queue*), bei dem jedem Element eine 'Priorität' für die Reihenfolge des Abrufes mitgegeben wird. Eine effiziente Implementation beruht auf binären Heaps (*binary heaps*).

Wie bei einer Warteschlange, können Werte elementweise der Vorrangwarteschlange zugefügt werden. Der Abruf erfolgt allerdings nicht chronologisch, sondern nach einer gegebenen Ordnung. Das Kriterium der Abrufreihenfolge (die Priorität) wird über einen `Comparator` bzw. ein `Comparable` realisiert. Im einfachsten Fall sind die Elemente der Vorrangwarteschlange vom Typ `Integer` oder `Double` und stellen gleichzeitig die Priorität dar. So kann z. B. immer das 'größte Element' abgerufen und entfernt werden. Man erhält so eine Sortierung der Eingabeelemente.

Auf komplexere Weise wird diese Datenstruktur in wichtigen Algorithmen benutzt. Bei Graphenalgorithmen können z. B. die entdeckten Knoten in der Reihenfolge ihres Abstandes von einem Startknoten abgerufen werden.

Zur Verdeutlichung der Nützlichkeit einer Vorrangwarteschlange betrachten wir hier eine einfache Aufgabe: Von einem sehr großen Eingabestrom sollen die Werte mit den M größten Schlüsseln herausgesucht werden.

- **Möglichkeit 1:** Unsortierte Liste mit den Werten der aktuell M größten Schlüssel speichern und neue Elemente mit all diesen Werten vergleichen: uneffizient.
- **Möglichkeit 2:** Mit einer sortierten Liste geht es schneller, bedeutet aber größeren Aufwand, um sie sortiert zu halten.
- **Möglichkeit 3:** Implementation durch eine Vorrangwarteschlange erlaubt einen Mittelweg mit einer teilweisen Sortierung, die die benötigte Funktionalität bei großer Effizienz gewährleistet.

Wir definieren die Funktionalität einer Vorrangwarteschlange über eine API:

API für eine Vorrangwarteschlange

public class MaxPQ<K extends Comparable<K>>

	`public MaxPQ(int capacity)`	Erzeugt Vorrangwarteschlange mit Kapazität `capacity`.
void	`add(K item)`	Fügt ein Element hinzu.
K	`poll()`	Entfernt den größten Schlüssel und gibt ihn zurück.
boolean	`isEmpty()`	Prüft, ob die Warteschlange leer ist.
int	`size()`	Gibt Anzahl der Elemente zurück.

Ebenso kann eine `MinPQ` Vorrangwarteschlange definiert werden, bei der `poll()` den kleinsten Schlüssel entfernt und zurückgibt. Unsere Implementation ist eine allgemeine `PriorityQueue`, bei der man im Konstruktor angibt, ob eine aufsteigende oder absteigende Ordnung realisiert werden soll (Argument `int sign` mit `-1` für `MinPQ` und `1` für `MaxPQ`).

Eine Vorrangwarteschlange implementieren

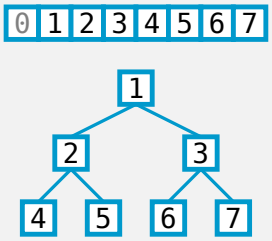
Auch, wenn die Standardbibliothek von Java eine `PriorityQueue` zur Verfügung stellt, gehen wir der Frage nach, wie eine Vorrangwarteschlange implementiert werden kann.

- Eine Möglichkeit wäre die Implementation als einfache Warteschlange, wobei die Methode `poll()` den minimalen Schlüssel in der unsortierten Schlange suchen müsste. Dies hätte eine Laufzeit von $\boldsymbol{O}(N)$.
- Wenn man ein geordnetes Feld benutzt, könnte das größte oder kleinste Element zwar in $\boldsymbol{O}(1)$ gefunden werden, aber das Einfügen eines neuen Elementes hätte eine *worst case* Laufzeit von $\boldsymbol{O}(N)$.
- Mit einem binären Heap kann eine Vorrangwarteschlange mit einer Laufzeit von $\boldsymbol{O}(\log N)$ für `poll()` und `add()` realisiert werden.

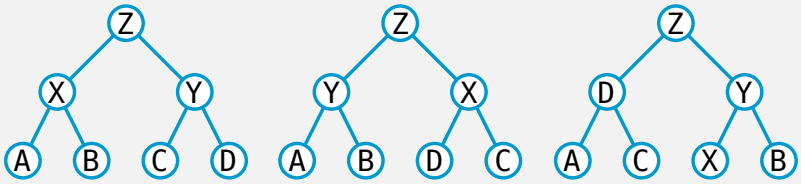
Definition: Binärer Heap

Ein binärer Heap (auch 'binäre Halde') ist ein Feld `A`, das als vollständiger (linksvoller) Binärbaum interpretiert wird und für den eine Ordnungsbedingung gilt.

- Die Wurzel ist `A[1]`. (Index `0` wird nicht benutzt.)
- Für Knoten k ist $\lfloor k/2 \rfloor$ der übergeordnete Knoten und $2k$ und $2k+1$ die beiden untergeordneten Knoten.
- Die Heap Ordnung besagt, dass jeder Schlüssel größer (bzw. kleiner für `MinPQ`) ist als die Schlüssel an den beiden untergeordneten Knoten.



Dabei ist zu beachten, dass die Positionen der Schlüssel durch die Heapordnung nicht vollständig festgelegt ist:



Eine Vorrangwarteschlange beruht im Wesentlichen auf einem binären Heap mit zwei Methoden, die der Aufrechterhaltung der Heap-Ordnung dienen. Das Hinzufügen und priorisierte Entnehmen von Elementen geschieht mit diesen beiden Methoden `swim()` und `sink()`. Das Verfahren wird hier am Beispiel einer MaxPQ erklärt.
Ist *ein* Schlüssel im Heap am falschen Platz, z. B. durch Einfügen oder das Ändern seiner Priorität, so kann die Heap Ordnung durch die beiden folgenden Operationen wiederhergestellt werden.

- **`swim()`**: Ist der Schlüssel größer als der Schlüssel des übergeordneten Knoten, wird er solange nach oben getauscht, bis er an der richtigen Stelle ist ("nach oben schwimmen").

- **`sink()`**: Ist der Schlüssel kleiner als ein (oder beide) Schlüssel der untergeordneten Knoten, so wird er mit dem größeren jener beiden vertauscht - iterativ bis er richtig platziert ist ("nach unten absinken").

Mit diesen beiden Methoden kann die Funktionalität einer Vorrangwarteschlange realisiert werden. Zum Einfügen eines Elementes fügt man es am Ende des Heaps hinzu und lässt es nach oben schwimmen (Methode `swim()`). Das größte Element befindet sich immer an der Wurzel des Heaps. Es kann also ohne Suche abgerufen werden. Um es zu Entfernen, holt man das letzte Element des Heaps an die Wurzel und lässt es absinken (Methode `sink()`).
Die Laufzeit $O(\log N)$ ergibt sich aus der Beobachtung, dass bei `swim()` und `sink()` die Anzahl der Iterationen durch die Höhe des Binärbaums begrenzt ist. Die Höhe eines Binärbaums mit N Elementen ist $\log(N)$.

3.1.1 Implementation einer `PriorityQueue`

Listing 3.1 Implementation einer Vorrangwarteschlange, leicht adaptiert aus [Sedgewick and Wayne, 2014, S. 345].

```
public class PriorityQueue<E extends Comparable<E>>
{
  private E[] pq;
  private int N;
  private int sign;        // +1 MaxPQ, -1 MinPQ

  public PriorityQueue(int capacity, int sign) {
    pq = (E[]) new Comparable[capacity+1];
    this.sign = sign;
  }
  public PriorityQueue(int capacity) {
    this(capacity, 1);     // Default ist MaxPQ
  }

  public int size()        { return N; }
  public boolean isEmpty() { return N == 0; }

  public void add(E e) {
    pq[++N] = e;
    swim(N);
  }

  public E poll() {
    E head = pq[1];
    swap(1, N);
    pq[N--] = null;
    sink(1);
    return head;
  }

  private boolean order(int i, int j) {  // '>=' für MaxPQ, '<=' für MinPQ
```

```
    return sign * pq[i].compareTo(pq[j]) >= 0;
  }

  private void swap(int i, int j) {
    E e = pq[i];  pq[i] = pq[j];  pq[j] = e;
  }

  private void swim(int k) {
    while (k > 1 && !order(k/2, k)) {
      swap(k/2, k);
      k = k/2;
    }
  }

  private void sink(int k) {
    while (2*k <= N) {
      int j = 2*k;
      if (j < N && !order(j, j+1))
        j++;
      if (order(k, j)) break;
      swap(k, j);
      k = j;
    }
  }
```

Wir kommen auf die Aufgabe aus der Einleitung von Vorrangwarteschlangen zurück. Das folgende Client Programm extrahiert aus dem Eingabefeld `testArray` die $M = 4$ größten Zahlen unter Benutzung einer `MinPQ`:

```
public static void main(String[] args) {
    int[] testArray = {4, 2, -17, 5, 23, 45, 0, 34, -7, 2, 0, 34};
    int M = 4;

    PriorityQueue<Integer> pq = new PriorityQueue<>(M+1, -1);
    for (int k : testArray) {
      pq.add(k);
      if (pq.size() > M) {
        pq.poll();
      }
    }
    // Ausgabe der extrahierten Zahlen:
    while (pq.size() > 0) {
      int k = pq.poll();
      System.out.println("Element: " + k);
    }
  }
```

3.1.2 Laufzeiten von Methoden der `PriorityQueue`

Die Laufzeit der Methoden der `PriorityQueue` aus den *Java Collections* ist der Java Dokumentation entnommen und in der folgenden Tabelle wiedergegeben.

PriorityQueue (*worst case*)	
`add(E e)`	$O(\log N)$
`contains(Object o)`	$O(N)$
`peek()`	$O(1)$
`poll()`	$O(\log N)$
`remove()`	$O(\log N)$
`remove(Object o)`	$O(N)$

Bei der `PriorityQueue` ist zu beachten, dass ein Ändern der Priorität durch Entfernen (`remove(Object o)`) und wieder Einfügen (`add(E e)`) realisiert werden muss, was in einer linearen Laufzeit resultiert. Im folgenden Abschnitt wird die `IndexPriorityQueue` eingeführt, die dies in logarithmischer Zeit erlaubt, sofern Indizes für die Elemente verfügbar sind.

3.2 Indizierte Vorrangwarteschlangen

Vorrangwarteschlangen haben den Nachteil, dass das Löschen eines Schlüssels nicht effizient realisiert werden kann. Die Laufzeit dieser Operation ist in $O(N)$, wobei eine logarithmische Laufzeit wünschenswert wäre. Außerdem haben die Schlüssel in vielen Anwendungen einen Index und es ist praktisch, auch über den Index auf die Schlüssel zugreifen zu können, zusätzlich zu der Möglichkeit mit `poll()`, z. B. auch um einen Schlüssel zu verändern.

Diese Möglichkeit bieten **Indizierte Vorrangwarteschlangen**. Über den Index können dann Elemente in $O(\log N)$ gelöscht werden.

Die Funktionalität einer indizierten Vorrangwarteschlange wird über die folgende API festgelegt:

API für eine indizierte Vorrangwarteschlange

public class IndexMaxPQ<K extends Comparable<K>>

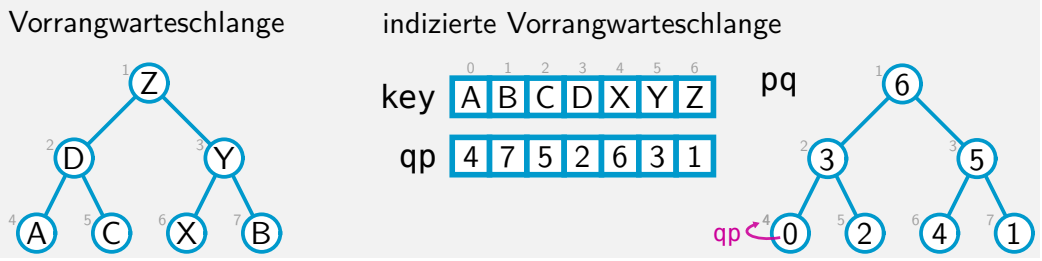
	`public IndexMaxPQ(int c)`	Erzeugt indizierte Vorrangwarteschlange mit Kapazität `c`.
void	`add(int i, K key)`	Fügt Schlüssel `key` mit Index `i` hinzu.
void	`change(int i, K key)`	Ändert den Schlüssel mit Index `i` in `key`.
void	`remove(int i)`	Löscht den Eintrag mit Index `i`.
int	`poll()`	Entfernt den Schlüssel vom Kopf und liefert seinen Index.
K	`peek()`	Liefert den Schlüssel vom Kopf der Schlange.
K	`peekIndex()`	Liefert den Index des Schlüssels vom Kopf der Schlange.
K	`keyOf(int i)`	Gibt den Schlüssel zu Index `i` zurück.
boolean	`contains(int i)`	Gibt es einen Schlüssel mit Index `i`?
boolean	`isEmpty()`	Prüft, ob die Warteschlange leer ist.
int	`size()`	Gibt Anzahl der Elemente zurück.

Durch die Index-basierten Methoden `add()`, `change()` und `keyOf()` beinhaltet eine indizierte Vorrangwarteschlange u. a. die Funktionalität eines Arrays.

Wie bei normalen Vorrangwarteschlangen, basiert die Implementation von indizierten Vorrangwarteschlangen auf einer binären Halde. Allerdings werden hier nicht die Schlüssel selbst, sondern ihre Indizes in der Halde gespeichert. Zwei zusätzliche Arrays gewährleisten den Zugriff über den Schlüssel-Index auf den Schlüssel und die Position in der Halde.

Insgesamt verwenden indizierte Vorrangwarteschlangen also drei Felder:

- Die Schlüssel werden unter ihrem angegeben Index in einem Feld `key` gespeichert.
- In einem binären Heap `pq` werden die Schlüssel-Indizes in der Heap Ordnung bezüglich der Schlüssel gespeichert.
- Ein weiteres Feld `qp` speichert die inverse Abbildung, vom Schlüssel-Index zum Index im Heap.



3.2.1 Implementation einer indizierten Vorrangwarteschlange

Listing 3.2 Implementation einer indizierten Vorrangwarteschlange, leicht adaptiert von `https://algs4.cs.princeton.edu/24pq` (Sedgewick und Wayne).

```
public class IndexPQ<K extends Comparable<K>>
{
  private K[] keys;
  private int[] pq;
  private int[] qp;          // pq[qp[i]] = qp[pq[i]] = i
  private int N;
  private int sign;          // +1 IndexMaxPQ, -1 IndexMinPQ

  public IndexPQ(int capacity, int sign) {
    keys = (K[]) new Comparable[capacity];
    pq = new int[capacity + 1];
    qp = new int[capacity];
    for (int i = 0; i < capacity; i++) {
      qp[i] = -1;            // qp[i]==-1 <=> es gibt keinen
    }                        //               Schlüssel zu Index i
    this.sign = sign;
  }

  public int size()              { return N; }
  public boolean isEmpty()       { return N == 0; }
  public boolean contains(int i) { return qp[i] != -1; }

  public void add(int i, K key) {
    keys[i] = key;
    qp[i] = ++N;
    pq[N] = i;
    swim(N);
  }

  public void change(int i, K key) {
    keys[i] = key;
    swim(qp[i]);
    sink(qp[i]);
  }

  public int peekIndex()
      { return pq[1]; }
  public K peek() {
      { return keys[pq[1]]; }
  public K keyOf(int i)
      { return keys[i]; }

  public int poll() {
```

```
    int head = pq[1];
    swap(1, N--);
    sink(1);
    qp[head] = -1;
    keys[head] = null;
    pq[N+1] = -1;
    return head;
  }

  public void remove(int i) {
    int index = qp[i];
    swap(index, N--);
    swap(index, N--);
    swim(index);
    sink(index);
    keys[i] = null;
    qp[i] = -1;
  }

// Methoden swim und sink wie zuvor
// Bei order und swap kleine Änderungen

  private boolean order(int i, int j) {
    return sign *
      keys[pq[i]].compareTo(keys[pq[j]])
      >= 0;
  }

  private void swap(int i, int j) {
    int tmp = pq[i];
    pq[i] = pq[j];
    pq[j] = tmp;
    qp[pq[i]] = i;
    qp[pq[j]] = j;
  }

  private void swim(int k) {
    while (k > 1 && !order(k/2, k)) {
      swap(k/2, k);
      k = k/2;
    }
  }

  private void sink(int k) {
    while (2*k <= N) {
      int j = 2*k;
      if (j < N && !order(j, j+1)) {
        j++;
      }
      if (order(k, j)) { break; }
      swap(k, j);
```

```
            k = j;
        }
    }
}
```

Wie schon bei den anderen Datenstrukturen ist dies eine Minimal Implementierung der Kernfunktionalität, bei der viele wichtige Überprüfungen weggelassen wurden.

3.2.2 Laufzeitbetrachtung einer indizierten Vorrangwarteschlange

Wie bei der Vorrangwarteschlange sind auch bei der indizierten Version `swim()` und `sink()` die einzigen Methoden, die Schleifen enthalten, abgesehen von der Initialisierung von `qp` im Konstruktor. Diese beiden Methoden haben auch hier eine Laufzeit in $\boldsymbol{O}(\log N)$, wobei N für die Anzahl der Elemente in der Schlage steht. Entsprechend haben alle Methoden, die `swim()` oder `sink()` aufrufen, eine Laufzeit in $\boldsymbol{O}(\log N)$. Dabei ist zu beachten, dass auch der Aufruf beider Funktionen die Wachstumsordnung nicht ändert (konstanter Faktor 2). Somit ist die Laufzeit von `add()`, `change()`, `poll()` und `remove()` in $\boldsymbol{O}(\log N)$, während alle anderen Methoden eine konstante Laufzeit haben.
Die Laufzeiten der indizierten Vorrangwarteschlange mit binärem Heap sind in folgender Tabelle zusammengefasst:

IndexPQ *(worst case)*	
`IndexPQ(int N, int sign)`	$\boldsymbol{O}(N)$
`add(int i, K key)`	$\boldsymbol{O}(\log N)$
`change(int i, K key)`	$\boldsymbol{O}(\log N)$
`contains(int i)`	$\boldsymbol{O}(1)$
`keyOf(int i)`	$\boldsymbol{O}(1)$
`peek()`	$\boldsymbol{O}(1)$
`peekIndex()`	$\boldsymbol{O}(1)$
`poll()`	$\boldsymbol{O}(\log N)$
`remove(int i)`	$\boldsymbol{O}(\log N)$

Verbesserung der Laufzeit

Folgende Maßnahmen können den konstanten Faktor in der Laufzeit verbessern:

- `swim()` und `sink()` können effizienter implementiert werden: In dem Heap wird nach oben bzw. nach unten die Zielposition gesucht und dann nur ein Tausch durchgeführt.
- Anstelle von `change()` können die Methoden `decreaseKey()` und `increaseKey()` implementiert werden, die jeweils nur `swim()` bzw. nur `sink()` aufrufen.

Zudem gibt es alternative Implementationen von (indizierten) Vorrangwarteschlangen, die eine bessere asymptotische Laufzeit haben, als die mit binären Heaps, z. B. mit Fibonacci Heap, Strict Fibonacci Heap oder Brodal Heap. Allerdings sind die Implementationen deutlich aufwändiger.

3.3 Union-Find: Dynamische Zusammenhangskomponenten

Dieser Abschnitt führt die Datenstruktur *Union-Find* zur dynamischen Verwaltung von Zusammenhangskomponenten ein. In diesem Skript wird die Datenstruktur in Abschnitt 10.5 für eine effiziente Implementierung des Kruskal Algorithmus für Minimale Spannbäume verwendet.

Die Darstellung ist etwas umfangreicher als notwendig, da sie einen Entwicklungsweg aufzeigt. Ausgehend von zwei naheliegenden Ansätzen werden über mehrere Schritte immer effizientere Varianten entwickelt. Der Vorgang zeigt, wie wichtig eine Analyse des Laufzeitverhaltens ist, insbesondere als Motivation um schnellere Lösungen zu finden. An diesem Beispiel ist auch interessant zu sehen, dass die spektakulär effizientere Version keine komplexere Implementierung benötigt als die ursprünglichen Implementierungen.

Bei der Benutzung der Union-Find Struktur startet man mit einer festen Anzahl von einzelnen Objekten. Die `union` Methode wird mit zwei Objekten aufgerufen und bewirkt, dass die beiden Mengen, die jene Objekte enthalten, vereinigt werden. (Falls sich die beiden Objekte schon in derselben Menge befinden, passiert nichts.) Mit der Methode `connected` kann abgefragt werden, ob zwei Objekte in derselben Menge liegen. Dies ist schon im Wesentlichen die ganze Funktionalität der Union-Find Struktur:

API für Union-Find (dynamische Verwaltung von Zusammenhangskomponenten)

public class UnionFind

	`UnionFind(int V)`	Initialisiert `V` Komponenten, jeweils mit einem Knoten `0`,...,`V-1`
`void`	`union(int v, int w)`	Verbindet die Komponenten, die `v` und `w` enthalten
`int`	`find(int v)`	ID der Komponente, die `v` enthält
`boolean`	`connected(int v, int w)`	Sind `v` und `w` in derselben Komponente?
`int`	`count()`	Anzahl der Komponenten

Wenn man die Union-Find Struktur im Zusammenhang mit Graphen sieht, ist folgende Interpretation naheliegend. Man startet mit einem Graphen mit V Knoten und ohne Kanten. Dann bildet jeder Knoten eine eigene Zusammenhangskomponente. Durch die `union(v, w)` Methode werden die Knoten `v` und `w` durch eine Kante verbunden. Dadurch werden ihre Zusammenhangskomponenten vereint. Mit `connected(v, w)` kann geprüft werden, ob es einen Weg zwischen `v` und `w` gibt, also ob sie in derselben Zusammenhangskomponente liegen. Mit dieser Vorstellung lässt sich die Funktionalität der Union-Find Struktur verstehen. Man muss dabei allerdings beachten, dass die Struktur nicht die Information speichert, welche Knotenpaare durch eine Kante verbunden sind, sondern nur, welche in derselben Komponente liegen. Die beiden Operationen `union(0, 2)` + `union(2, 5)` führen also zu exakt demselben Ergebnis wie

`union(0, 5)` + `union(2, 5)`. Abbildung 3.1 illustriert die Funktionsweise der Union-Find Struktur.

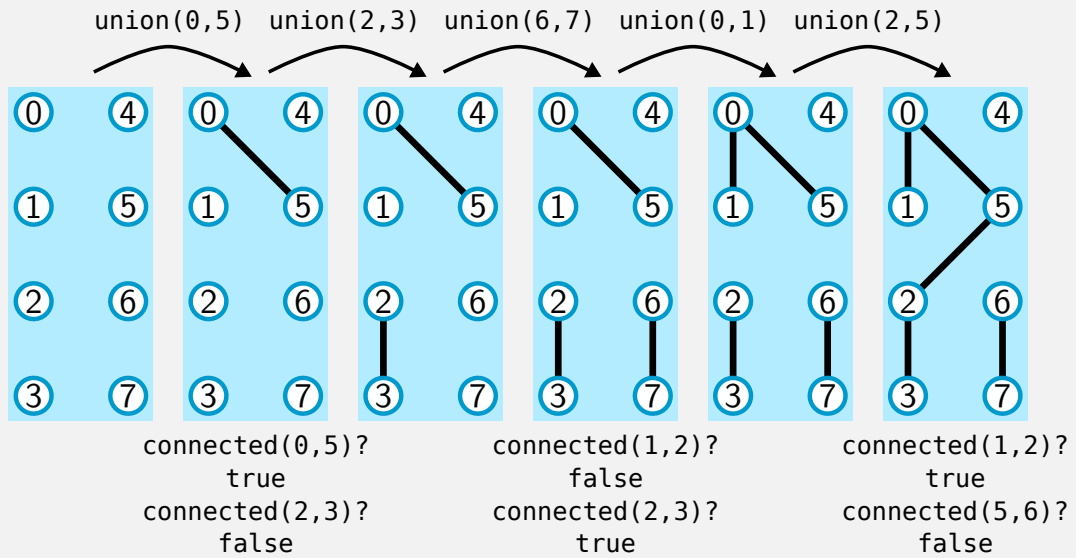


Abbildung 3.1 Illustration der Union-Find Struktur: Die *union* Operation verbindet die Komponenten, in denen die zwei angegebenen Knoten enthalten sind und die *connected* fragt ab, ob zwei Knoten in derselben Komponente liegen. Bei dieser Darstellung ist zu beachten, dass die Union-Find Struktur *keine Kanten* speichert, sondern nur die Information, welche Knoten in einer Komponente liegen.

Wir werden nun eine Folge von Implementierungen mit schrittweisen Verbesserungen durchgehen. Zunächst wählen wir eine Variante, bei der `find()`, bzw. `connected()` möglichst schnell ist. Wir verwenden ein Knoten-indiziertes Array `id`, das die Nummer einer Zusammenhangskomponente speichert. Eine `connected(v,w)` Anfrage entspricht dann dem Vergleich `id[v] == id[w]`.

Listing 3.3 Union-Find Implementation mit quick-find

```
public class UnionFind { // quick-find
  private int[] id;
  private int count;

  public UnionFind(int V) {
    count = V;
    id = new int[V];
    for (int v = 0; v < V; v++) {
      id[v] = v;
    }
```

```
    }

    public int count()
    { return count; }

    public int find(int v)
    { return id[v]; }

    public boolean connected(int v, int w)
    { return id[v] == id[w]; }

    public void union(int v, int w) {
      if (connected(v, w))
        return;
      int wid = id[w];
      for (int u = 0; u < id.length; u++) {
        if (id[u] == wid) {
          id[u] = id[v];
        }
      }
      count--;
    }
  }
```

Die Methoden `find()` und `connected()` sind sehr schnell. Sie haben eine konstante Laufzeit. Aber `union()` muss das ganze Feld durchsuchen, also liegt die Laufzeit in $\boldsymbol{O}(V)$. Daher ist dieser Ansatz nicht sehr effizient.

Um eine schnellere `union()` Operation zu realisieren, können wir eine Baumstruktur für jede Komponente nutzen. Der Knoten `w` an der Wurzel hat seinen eigenen Index als `id`, also `id[w]==w`. Andere Knoten `v` haben den Index ihres Elternknoten als `id[v]`. Daher benennen wir die Variable `id` ab jetzt in `parent` um. *Ein* Knoten jeder Komponente verweist per `parent` auf sich selbst. Dieser wird *Wurzel* genannt.

Bei diesem Ansatz ist `union(v,w)` sehr einfach zu realisieren: Man lässt die Wurzel der `w`-Komponente auf *die Wurzel* der `v`-Komponente verweisen. Man könnte auch die Wurzel der `w`-Komponente auf `v` verweisen lassen, aber das führt zu hohen Bäumen und damit zu einer schlechteren Laufzeit. Im Gegenzug ist nun `find()` aufwändiger: Man muss über das Elternknoten-Array `parent` die Wurzel suchen:

Listing 3.4 Union-Find Implementation mit (lazy) quick-union

```
1 // UnionFind mit lazy quick-union
2 //   gebenüber der quick-find Version
3 //   sind nur folgende Methoden anders:
```

```
4   public int find(int v) {
5     while (v != parent[v])
6       v = parent[v];
7     return v;
8   }
9
10  public void union(int v, int w) {
11    int vRoot = find(v);
12    int wRoot = find(w);
13    if (vRoot == wRoot)
14      return;
15
16    parent[wRoot] = vRoot;
17    count--;
18  }
```

Die Methode `find()` hat im schlechtesten Fall eine Laufzeit in $O(V)$. Und auch wenn das eigentliche Zusammenlegen zweier Komponenten schnell geht (Zeile 16), bringt es im *worst case* keinen Vorteil, da zunächst per langsamem `find()` die Wurzeln der Komponenten gesucht werden müssen. Also geht die Suche nach einem geeigneten Ansatz weiter.

Die Fälle mit schlechter Laufzeit bei dem *quick-union* Ansatz treten auf, wenn größere Komponenten an kleinere angehängt werden, weil dabei der entstehende Baum höher wird. Dem Höhenwachstum der Bäume kann dadurch begegnet werden, dass immer der kleinere an den größeren Baum angehängt wird. Dazu speichern wir die Höhe jedes Baums in dem Knoten-indizierten Array `ht`.

Listing 3.5 Implementation von eager Union-Find

```
1  public class UnionFind { // flache Bäume
2    private int[] parent;
3    private int[] ht;
4    private int count;
5
6    public UnionFind(int V) {
7      count = V;
8      parent = new int[V];
9      ht = new int[V];
10     for (int v = 0; v < V; v++) {
11       parent[v] = v;
12       ht[v] = 0;
13     }
14   }
15
16   // count, connected und find wie oben
17
```

```
18   public void union(int v, int w) {
19     int vRoot = find(v);
20     int wRoot = find(w);
21     if (vRoot == wRoot)
22       return;
23
24     if (ht[wRoot] < ht[vRoot]) {
25       parent[wRoot] = vRoot;
26     } else {
27       parent[vRoot] = wRoot;
28       if (ht[vRoot] == ht[wRoot]) {
29         ht[wRoot]++;
30       }
31     }
32     count--;
33   }
34 }
```

Laufzeit von Union-Find mit flachen Bäumen

Maximale Höhe der Bäume in *eager* Union-Find

Die Bäume in *eager* Union-Find in Listing 3.5 haben maximal die Höhe $\lg V$. Dabei bezeichnet lg den Logarithmus zur Basis 2.

Beweis. Wir zeigen durch Induktion nach n, dass ein Baum mit n Knoten in *eager* Union-Find höchstens die Höhe $\lg n$ hat. Werden zwei Bäume mit *unterschiedlichen* Höhen zusammengelegt, so hat der entstehende Baum dieselbe Höhe, wie der größere der beiden ursprünglichen Bäume. Daher folgt die Induktionsbehauptung (IB) direkt aus der Induktionsvoraussetzung (IV).

Betrachten wir also den Fall, dass zwei Bäume mit k_0 und k_1 Knoten und Höhe h zu einem Baum mit $k_0 + k_1$ Knoten vereint werden. Nach IV gilt $h \leq \lg k$ für $k = \min(k_0, k_1)$. Durch das Anhängen des einen Baumes an die Wurzel des anderen Baumes steigt die Höhe um eins. Also ist für den neuen Baum die Höhe maximal $1 + h \leq 1 + \lg k = \lg(2k) \leq \lg(k_0 + k_1)$. □

Mit *eager* Union-Find haben wir einen Algorithmus für dynamische Zusammenhangskomponenten mit guter Laufzeit auch im *worst case*. Diese Laufzeitgarantie ist für die Praxis ausreichend, aber es geht noch besser! Man kann bei der `find()` Operation die Baumstruktur nebenbei verbessern. 'Verbessern' heißt in unserem Fall, die Bäume möglichst flach zu halten, da dann die Wurzel von beliebigen Knoten in der Komponente schnell erreicht wird.

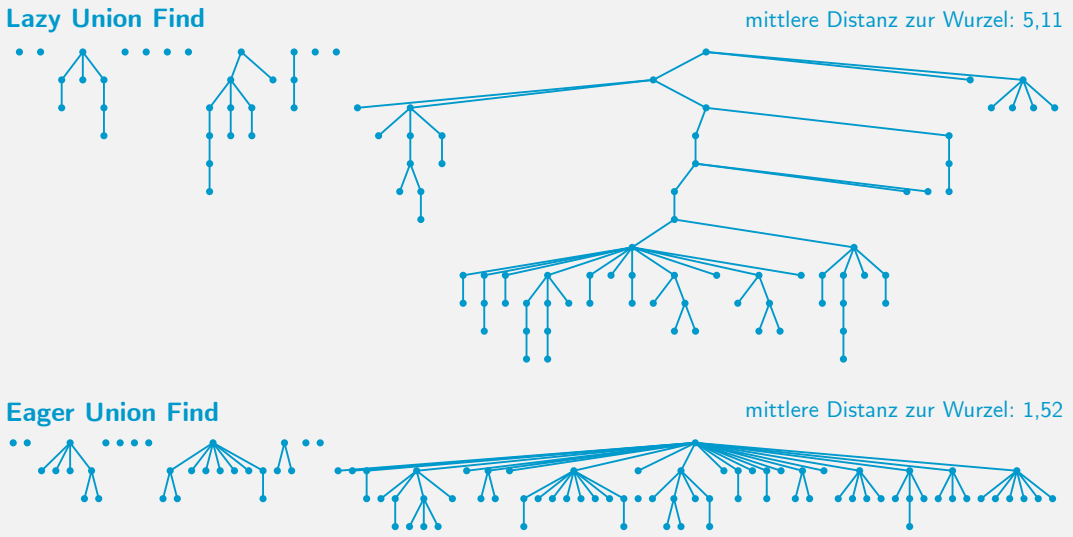


Abbildung 3.2 Die Abbildung stellt die Bäume gegenüber, die mit den beiden Union-Find Implementationen in Listing 3.4 und 3.5 erstellt wurden. Dabei wurden auf 100 Knoten 88 zufällige `union()` Operationen ausgeführt. Die Grafik wurde der Abbildung aus [Sedgewick and Wayne, 2014, S. 254] nachempfunden.

Bei der sogenannten **Pfadkompression** nutzt man das Iterieren zu der Wurzel eines Baumes während `find()` dazu, alle durchlaufenen Knoten näher an die Wurzel zu bringen. Dies kann auf zwei Arten realisiert werden:

- Zweistufig: Bestimme im ersten Durchlauf die Wurzel wie bisher und hänge im zweiten Durchlauf alle Knoten direkt an die Wurzel.
- Einstufig: Hänge jeden Knoten in der Iteration an seinen Vorvorgänger (Halbierung der Pfadtiefe).

Listing 3.6 Union-Find Implementation mit Pfadkompression

```
public class UnionFind {    //Pfadkompression
  // ... wie eager UnionFind
  // es ändert sich nur find()

  public int find(int v) {
    while (v != parent[v]) {
      parent[v] = parent[parent[v]];
      v = parent[v];
    }
    return v;
```

```
    }
  }
```

Die Laufzeitanalyse dieser Union-Find Variante mit Pfadkompression ist sehr komplex. Daher begnügen wir uns in dieser Vorlesung mit dem Ergebnis.

Wachstumsordnung für V Knoten (*worst case*)

Algorithmus	init	union	find
Quick-find	$O(V)$	$O(V)$	$O(1)$
Lazy quick-union	$O(V)$	$O(V)$	$O(V)$
Eager quick-union	$O(V)$	$O(\lg V)$	$O(\lg V)$
Union-Find mit Pfadkompression	$O(V)$	$O(\lg^* V)$	$O(\lg^* V)$

Was ist $\lg^*(V)$?? Die vereinfachte, für die Praxis ausreichende Antwort ist: $\lg^*(V) < 5$. Das stimmt allerdings nicht immer. Aber zumindest für $V <$ # Atome im Universum, also für alle in der Praxis vorkommenden Fälle. Die Definition des **iterierten Logarithmus** lautet:

$$\lg^*(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \lg^*(\lg(n)) & \text{otherwise} \end{cases}$$

In dieser Tabelle sieht man wie unglaublich langsam die Funktion wächst:

V	$\lg^* V$
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

Daher ist es umso erstaunlicher, dass es für die Laufzeit von Union-Find mit Pfadkompression eine noch schärfere Abschätzung mit der inversen Ackermann Funktion gibt. Diese Funktion wächst noch langsamer als der iterierte Logarithmus.

3.3.1 Laufzeiten der Union-Find Varianten

Laufzeiten für N `union/find` Operationen bei V Knoten (*worst case*)	
Algorithmus	**Laufzeit**
Quick-find	$\boldsymbol{O}(NV)$
Lazy quick-union	$\boldsymbol{O}(NV)$
Eager quick-union	$\boldsymbol{O}(V + N \lg V)$
Union-Find mit Pfadkompression	$\boldsymbol{O}((V + N) \lg^* V)$

Um die Steigerung der Effizienz durch geeignete Algorithmen zu quantifizieren, nehmen wir als Beispiel $V = 10^9$ Knoten (passt gerade in heutige Speicher) und $N = 10^9$ `union/find` Operationen. Dann bedeutet der Wechsel von den ersten Union-Find Varianten zu Union-Find mit Pfadkompression eine Reduktion der Laufzeit **von 30 Jahren auf 6 Sekunden**.
Die Laufzeit des Algorithmus mit Pfadkompression ist sehr, sehr nah an linearem Wachstum. Daher ist es ein interessantes Ergebnis von [Fredman and Saks, 1989], dass es keinen linearen Algorithmus geben kann.

Quellenverweise

Dieses Kapitel orientiert sich stark an dem Buch [Sedgewick and Wayne, 2014]. Abbildung 3.2 ist aus [Sedgewick and Wayne, 2014, S. 254].

KAPITEL 4

Backtracking

Das **Backtrack-Prinzip** (*Backtracking*) ist ein systematischer Ansatz, um einen Lösungsraum vollständig zu durchsuchen. Bei vielen Aufgabenstellungen ist man daran interessiert, eine bestimmte Lösung zu finden, z. B. eine, die ein Optimalitätskriterium erfüllt (*kürzester* Weg, *bester* Zug). In diesen Fällen versucht man, eine vollständige Durchsuchung des Lösungsraums zu vermeiden. Wir werden in den folgenden Kapiteln viele Ansätze für eine solche gezielte Lösungssuche besprechen.
Im Gegensatz dazu wird Backtracking verwendet,

- wenn alle Lösungen eines Problems gefunden werden sollen, z. B. Gewinnstellungen bei einem Strategiespiel und
- wenn kein 'intelligenterer' Ansatz bekannt ist.

Des Weiteren hat *Backtracking* den Vorteil, dass es sich auch bei komplexeren Strukturen relativ leicht implementieren lässt. Durch den geringen Aufwand bei der Implementation ist daher der Ansatz für Probleme mit kleinem Suchraum attraktiv.
Zudem lässt sich die Effizienz von Backtracking zum Finden einer bestimmen ('optimalen') Lösung in manchen Fällen dadurch steigern, dass viele Äste des Suchbaums nicht berücksichtigt werden müssen. Dies ist der Fall, wenn bei gewissen Teillösungen ausgeschlossen werden kann, dass sie zu einer (optimalen) Lösung fortgesetzt werden können. Backtracking ist somit die Grundlage für komplexere Algorithmen wie *Branch-and-Bound*
Im Folgenden werden wir zunächst zwei konkrete Beispiele für Backtracking betrachten, das Auflisten aller Permutationen der Zahlen $0, 1, \ldots, N-1$ in Abschnitt 4.1 und den Rösselsprung in Abschnitt 4.2. Im Anschluss wird das generelle Prinzip des Backtracking erläutert. Dann werden das Lösen

von Sudoku Rätseln mit Backtracking sowie einige Verbesserungen skizziert. Abschließend wird das Verfahren des Backtracking als Suche im Baum der Teillösungen dargestellt und Varianten zur Steigerung der Effizenz besprochen.

4.1 Beispiel: Aufzählung aller Permutationen

Wir betrachten die Aufgabe, alle Permutationen der Zahlen $0, 1, \ldots, N-1$ aufzulisten. Dazu bilden wir Sequenzen dieser Zahlen, die jede Zahl nur einmal enthalten. Diese Sequenzen sind die Teillösungen des Problems. Eine Sequenz der Länge N ist eine (Voll-)Lösung. In diesem Beispiel kann jede Teillösung zu (mindestens) einer vollständigen Lösung erweitert werden.
Den Ausgangspunkt bildet die leeren Liste, d.h. es wurde noch keine Zahl gewählt. Diese Liste stellt die aktuelle Teillösung dar. Nun werden alle möglichen Erweiterungen generiert:

- Für die erste Stelle kommen alle Zahlen aus $\{0, \ldots, N-1\}$ als Kandidaten in Frage.
- Es wird ein Kandidat für die erste Stelle gewählt.
- Für die zweite Stelle kommen alle Zahlen in Frage, die noch nicht in der Teillösung verwendet wurden.
- Dieser Prozess wird solange durchlaufen, bis alle Plätze besetzt (bzw. alle Zahlen ausgewählt) sind. Dann ist eine Lösung gefunden.
- Sobald es für eine Stelle keine neuen Kandidaten mehr gibt, geht man eine Stelle zurück (*backtracking!*) und setzt dort mit dem nächsten Kandidaten fort. (Dies bedeutet im Allgemeinen, dass der Codeablauf mit dem nächsten Schleifenaufruf auf der Ebene, deren Schleife noch nicht zu Ende gelaufen ist, fortgesetzt wird.)

Der Ablauf ist in Abbildung 4.1 für die Zahlen 0, 1, 2 illustriert.

Java Implementation: Permutationen mit *Backtracking*

Listing 4.1 Erzeugung aller Permutationen mit Backtracking

```
public class Permutations {
  private int N;
  private int[] a;

  public Permutation(int N) {
    this.N = N;
```

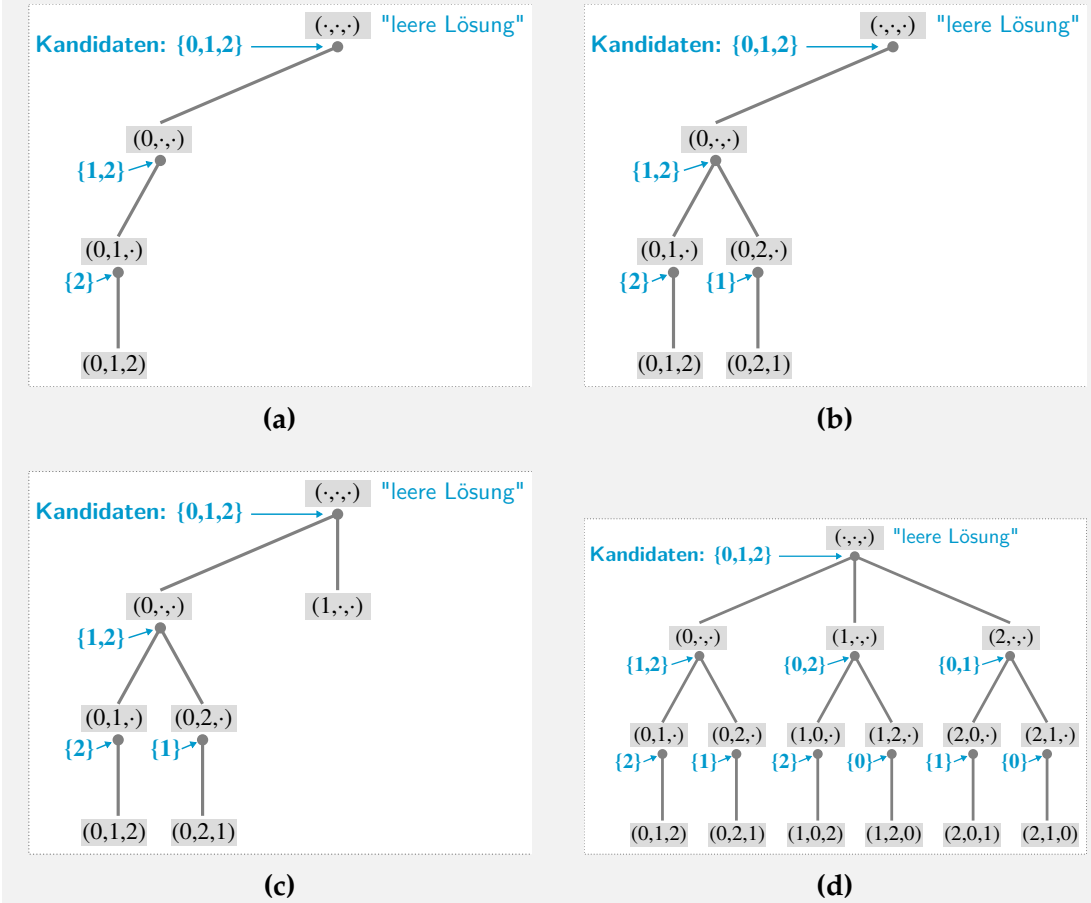


Abbildung 4.1 Alle Permutation der Zahlen 0, 1, 2 werden per Backtracking erzeugt: (a) Ausgehend von der leeren Teillösung, wird in jedem Schritt ein möglicher Kandidat an die Liste angehängt, bis die Liste die Länge 3 hat, also eine vollständige Lösung ist. **(b)** Nun setzt das Backtracking ein. Nachdem ein Schritt rückgängig gemacht wird, ist die Teillösung $(0,1)$. An dieser Stelle gibt es keine weiteren Möglichkeiten. Der einzige Kandidat, die Zahl 2, wurde schon benutzt. Daher geht das Backtracking noch einen weiteren Schritt zurück zu der Teillösung (0). Nachdem an dieser Stelle die 1 schon genutzt wurde, wird nun die 2 gewählt. Man erhält die Teillösung $(0,2)$ und in einem weiteren Schritt die Lösung $(0,2,1)$. **(c)** Es setzt wieder das Backtracking ein. Nun muss es drei Schritte zurückgehen, um von der leeren Teillösung die nächste Möglichkeit zu wählen. So erhält man die Teillösung (1). **(d)** Es wird der mittlere Teilbaum erzeugt und schließlich der rechte. Damit sind alle Permutation als Lösungen an den Blättern erzeugt worden.

```
      a = new int[N];
      backtracking(0);
    }

    // candidateList(): findet unbenutzte Zahlen
    public Iterable<Integer> candidateList(int k) {
      LinkedList<Integer> c =
            new LinkedList<>();
      boolean[] used = new boolean[N];

      for (int i = 0; i < k; i++)
        used[a[i]] = true;
      for (int i = 0; i < N; i++)
        if (!used[i])
          c.add(i);

      return c;
    }

    public void backtracking(int k) {
      if (k == N) printSolution();
      else {
        Iterable<Integer> candidates = candidateList(k);
        for (Integer c : candidates) {
          a[k] = c;
          backtracking(k+1);
        }
      }
    }

    public void printSolution() {
      for (int i : a)
        System.out.print(i + " ");
      System.out.println();
    }
  }
```

Zwischenresümee *Backtracking*

Es gibt immer nur eine aktuelle Teillösung (Variable a), die gespeichert werden muss. Das stellt einen großen Vorteil gegenüber anderen Verfahren da. Wenn der Baum der Teillösungen in der Reihenfolge einer Breitensuche (s. u.) durchsucht werden würde, müssten viele Teillösungen gleichzeitig im Speicher gehalten werden.
Zur Erzeugung der Permutationen haben wir folgende Elemente benutzt:

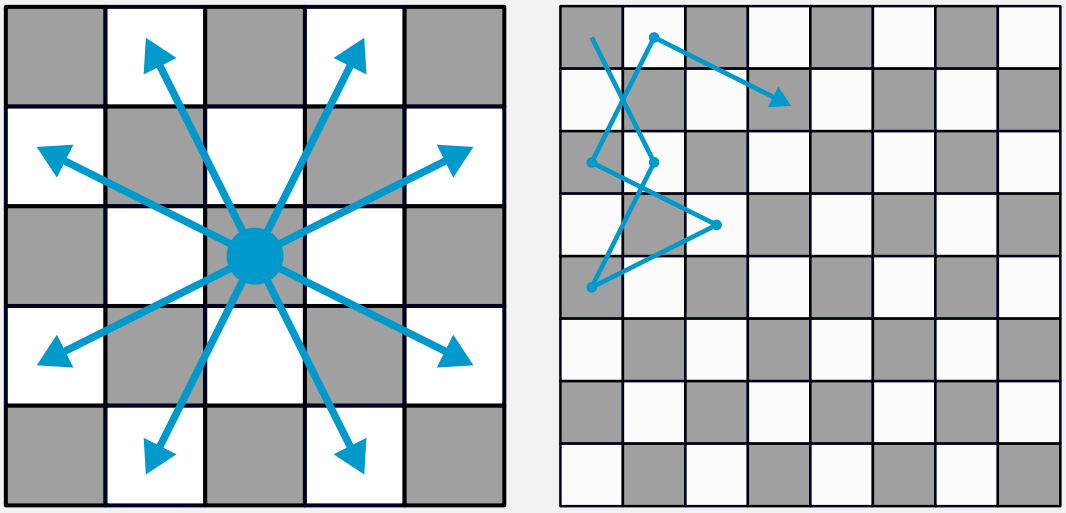
- Test, ob Lösung gefunden wurde

- Menge der Kandidaten für Auswahlschritt erzeugen
- Schleife über alle diese Kandidaten
- Auswahl durchführen
- rekursiver Aufruf

Bevor wir zu dem generellen Prinzip kommen, besprechen wir noch ein zweites Beispiel.

4.2 Beispiel: Rösselsprung

Wir betrachten das Problem des *Rösselsprungs*. Gegeben sei ein Schachbrett mit einem Springer, dabei soll eine Zugfolge gefunden werden, bei der jedes Feld genau einmal besucht wird, siehe Abbildung 4.2. Beim *geschlossenen Rösselsprung* muss die Endposition mit der Ausgangsposition übereinstimmen. Allgemeiner wird das Problem für beliebige Brettgröße $n \times n$ betrachtet. Für alle $n \geq 5$ gibt es Lösungen, für $n < 5$ nicht.



(a) **(b)**

Abbildung 4.2 Der Rösselsprung: Die möglichen Züge eines Springer (Rössel) beim Schachspiel zeigt **(a)**. Ein Rösselsprung ist eine Zugfolge mit einem Springer, bei der alle Felder genau einmal besucht werden. **(b)** zeigt eine Anfangssequenz eines Rösselsprungs (ohne Garantie, dass sie zu einer Lösung vervollständigt werden kann).

Die Aufgabe ist strukturell ähnlich zu dem vorigen Beispiel der Permutationen. Eine Lösung kann schrittweise durch Hinzufügen eines Teilschrittes erzeugt werden.
Anders als bei den Permutationen kann man hier in eine Sackgasse gelangen. Bei vielen Teillösungen gibt es keine Zugmöglichkeit, da alle erreichbaren Felder von der Zugfolge schon benutzt wurden. Eine solche Teillösung kann also nicht zu einer Lösung erweitert werden. In diesem Beispiel kann es also notwendig sein, schon vor dem Erreichen einer Lösung das Backtracking auszulösen. Ein weiterer Unterschied ist, dass wir beim Rösselsprung schon mit einer einzigen Lösung zufrieden sind. Alle Lösungen zu finden ist auch ein interessantes Ziel, allerdings (abhängig von der Brettgröße) sehr rechenintensiv.

Backtracking Ansatz für den Rösselsprung

Wir brauchen zunächst Strukturen, um Brett und Spielzüge zu realisieren. Diese implementieren wir in den Klassen `Position` (Position auf Spielbrett), `Move` (Spielzug) und `Board` (Spielbrett). Die Klasse `Board` muss insbesondere Methoden für die Elemente des *Backtracking* bereitstellen:

- `validMoves()` mögliche Züge generieren
- `isSolved()` ist Lösung erreicht?
- `doMove(move)` Zug ausführen
- `undoMove(move)` Zug zurücknehmen

Damit kann die rekursive *Backtracking* Methode leicht implementiert werden, siehe Methode `backtracking()` der Klasse `ClosedKnightsTour` am Ende von Code 4.2.

Listing 4.2 Finden eines geschlossenen Rösselsprungs mit Backtracking. Die Klassen `Position`, `Move` und `Board` implementieren die Spiellogik und die Klasse `ClosedKnightTour` realisiert das eigentliche Backtracking. Die Methoden, die wesentliche Elemente des Backtracking darstellen, sind farbig hervorgehoben.

```
public class Position {
  protected int x, y;

  public Position(int x, int y) {
    this.x = x;
    this.y = y;
  }

  public Position(Position that) {
    this(that.x, that.y);
  }
```

```
    public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null || getClass()
          != o.getClass()) return false;
      Position that = (Position) o;
      return this.x == that.x
          && this.y == that.y;
    }
  }

  public class Move {
    protected Position pos;
    private int dir;
    public static final int nDirs = 8;
    private static final int[] xd = {1, 2, 2, 1, -1, -2, -2, -1};
    private static final int[] yd = {-2, -1, 1, 2, 2, 1, -1, -2};

    public Move(Position pos, int dir) {
      this.pos = new Position(pos);
      this.dir = dir;
    }

    public Position endPosition() {
      int x = pos.x + xd[dir];
      int y = pos.y + yd[dir];
      return new Position(x, y);
    }

  }

  public class Board {
    public static final int OFF = -1;
    private Position startPos;
    private int nx;
    private int ny;
    private boolean[][] field;
    private Position knightPos;
    private int nFree;

    public Board(int nx, int ny) {
      this.startPos = new Position(0, 0);
      this.nx = nx;
      this.ny = ny;
      this.field = new boolean[nx][ny];
      this.knightPos = startPos;
      this.nFree = nx * ny - 1;
    }

    public int getField(Position pos) {
```

```
      if (pos.x < 0 || pos.x >= nx ||
          pos.y < 0 || pos.y >= ny) {
        return OFF;
      } else {
        return field[pos.x][pos.y] ? 1: 0;
      }
    }

    public Stack<Move> validMoves() {
      Stack<Move> moves = new Stack<>();
      for (int dir = 0; dir < Move.nDirs; dir++) {
        Move move = new Move(knightPos, dir);
        if (getField(move.endPosition()) == 0) {
          moves.push(move);
        }
      }
      return moves;
    }

    public boolean isSolved() {
      return nFree == 0 && knightPos.equals(startPos);
    }

    public void doMove(Move move) {
      knightPos = move.endPosition();
      field[knightPos.x][knightPos.y] = true;
      nFree--;
    }

    public void undoMove(Move move) {
      field[knightPos.x][knightPos.y] = false;
      knightPos = move.pos;
      nFree++;
    }
  }

  public class ClosedKnightsTour {

    public static boolean backtracking(Board board) {
      Stack<Move> moves = board.validMoves();
      for (Move move : moves) {
        board.doMove(move);
        if (board.isSolved()) {
          return true;
        }
        boolean solutionFound = backtracking(board);
        if (solutionFound) return true;
        board.undoMove(move);
      }
      return false;
```

```
    }

    public static void main(String[] args) {
      Board board = new Board(5, 5);
      boolean solvable = backtracking(board);
      System.out.println("Lösbar? " + solvable);
    }
  }
```

Bemerkungen zur *Backtracking* Lösung für den Rösselsprung

Der Code 4.2 entscheidet nur, ob es eine Lösung gibt. Um diese auch auszugeben, bedarf es einer kleinen Ergänzung. Es gibt zur Laufzeit immer nur ein `Board` Objekt. Bei dem rekursiven Aufruf wird eine Referenz übergeben.
Backtracking ist eine systematische Durchsuchung des Lösungsraums, die meist nicht effizient zu einzelnen Lösungen führt. Lösungen für den Rösselsprung können mit *Backtracking* nur für die Brettgröße 5×5 in akzeptabler Zeit gefunden werden. Bei größeren Spielfeldern braucht man andere Methoden, z.B. die Regel von Warnsdorff als Heuristik [von Warnsdorf, 1823] oder *Divide-and-Conquer*. Eine intensive Auseinandersetzung mit dem Problem und eine Lösung in quadratischer Laufzeit bietet [Conrad et al., 1994].

4.3 Generelles Backtrack-Prinzip

Die gezeigten Beispiele motivieren das folgende allgemeine Backtrack-Prinzip zum Generieren aller Lösungen für ein Problem:

```
1 Starte mit leerer Lösung
2 Rekursion:
3   Generiere Kandidaten für nächsten Schritt in aktueller Teillösung // [1]
4   Für jeden Kandidaten:
5     Führe Schritt aus // [2]
6     Falls dies eine Lösung ergibt: ausgeben bzw. speichern // [3]
7     Gehe in die Rekursion für den nächsten Schritt
8     Mache Schritt rückgängig // [4]
```

Elemente des *Backtracking*:

1 Menge an möglichen Lösungskandidaten generieren

2 Lösungsschritt ausführen

3 Prüfen, ob eine Teillösung eine vollständige Lösung ist

4 Lösungsschritt rückgängig machen

Bei jeder speziellen Anwendung können gewisse Änderungen an dem generellen Schema sinnvoll bzw. notwendig sein. Der Punkt *Schritt rückgängig machen* [4] ist in manchen Fällen überflüssig, wie im Permutationen Beispiel. Das *Prüfen auf Lösung* [3] wird manchmal am Anfang der *Backtracking* Methode durchgeführt. Falls nur eine Lösung gesucht wird (und nicht alle) wie bei unserem Rösselsprung Beispiel, wird die Suche in Zeile 6 abgebrochen, falls eine Lösung gefunden wurde. Damit der Abbruch beim Rücklauf durch die Rekursionsstufen weitergegeben wird, setzt man eine Variable oder einen Rückgabewert und prüft nach Zeile 7, ob beim Rekursionsaufruf eine Lösung gefunden wurde. Wenn dies der Fall ist, springt man aus der Schleife, ohne weitere Möglichkeiten zu probieren. Im `backtracking()` der `ClosedKnightTour` in Code 4.2 wurde dazu der Rückgabewert `true` benutzt.
Eine mögliche Verbesserung des generellen Vorgehens ist die folgende. Die möglichen Lösungsschritte werden nicht in beliebiger Reihenfolge durchprobiert, sondern besonders aussichtsreiche zuerst. Dieser Ansatz führt bei dem folgenden Sudoku Beispiel zu einer deutlichen Steigerung der Effizienz. Diese und weitere Verbesserungen werden in Abschnitt 4.6 als Suche im Baum der Teillösungen veranschaulicht.

4.4 Beispiel: Sudoku

Als ein konkretes Beispiel für Backtracking mit Erweiterungen betrachten wir das Lösen von **Sudoku** Rätseln. Die Regeln sind wie folgt. In die freien Kästchen werden die Ziffern 1–9 eingetragen. Dabei muss jede Ziffer in jeder Zeile, in jeder Spalte und in jeder umrandeten 3×3 Submatrix *genau einmal* auftreten.
In Bezug auf das Generieren von Lösungen gibt es hier eine Neuerung gegenüber dem zuvor betrachteten Beispiel. Bei dem Rösselsprung müssen die Züge in der vorgegebenen Reihenfolge ausgeführt werden. Bei Sudoku kann in jedem Schritt ausgewählt werden, welches Kästchen als nächstes ausgefüllt wird. Dies gibt spezielle Möglichkeiten, um die Effizienz zu steigern.

				5			8	
		6				1		
	8				3			
2	3							
			8		1	2		
8		1			4			
3		5		7				2
							4	
9			3		5			8

Abbildung 4.3 Ein Sudoku Rätsel.

Elemente des *Backtracking* zum Sudoku Lösen

Gemäß dem generellen Schema des Backtracking in Abschnitt 4.3, legen wir die vier Elemente des Backtracking fest:

- Lösungskandidaten: Alle Ziffern, die nach den Regeln passen

- Teillösung vollständig? Wenn alle Felder gefüllt sind
- Schritt ausführen: Ziffer in ein Kästchen speichern
- Schritt rückgängig machen: Ziffer aus dem Kästchen löschen

Wie oben bemerkt, kommt beim Sudoku Lösen ein weiteres Element dazu, die Auswahl des nächsten Kästchen, das gefüllt wird. Hier wählen wir zunächst die einfachste Variante:

- Auswahl: Nimm das erste leere Kästchen!

Mit diesen Festlegungen im Rahmen des generellen Backtrack-Prinzips lässt sich leicht ein Programm zum Lösen von Sudoku Rätseln implementieren. Die Effizienz ist allerdings nicht so gut, insbesondere bei Rätseln der schwierigen Kategorie.
Daher betrachten wir zwei Verbesserungen, die dazu führen, dass aussichtsreiche Lösungsschritte zuerst verfolgt werden und hoffnungslose Teillösungen frühzeitig erkannt und nicht weiter expandiert werden:

- Vorausschauende Kandidatenmenge: Falls für irgendein Kästchen keine Lösungskandidaten mehr existieren, gib für jeden Auswahlschritt eine leere Auswahl zurück. Die Teillösung kann nicht vervollständigt werden, daher sollte dieser Ast des Baumes nicht expandiert werden.
- Eingegrenzte Auswahl: Nimm als nächsten Auswahlschritt dasjenige Kästchen, für das am wenigsten passende Kandidaten existieren (ähnlich der Warnsdorff Regel beim Rösselsprung).

Mit diesen kleinen Verbesserungen erhält man ein effizientes Programm zum Lösen von Sudoku Rätseln, siehe Tabelle 4.1.

Auswahlkriterium	Sudoku Schwierigkeitsgrad		
	leicht	mittel	schwer
der Reihe nach	1.904.832	863.305	(term. nicht)
vorausschauend	127	142	12.507.212
eingegrenzt & vorausschauend	48	65	10.374

Tabelle 4.1 Die Tabelle zeigt die Anzahl der Lösungsschritte beim Lösen von drei Sudoku Rätseln verschiedener Schwierigkeitsstufen im Vergleich der unterschiedlichen Auswahlmethoden in einer Simulation von [Skiena, 2008].

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

Abbildung 4.4 Das schwierige Sudoku Rätsel aus [Skiena, 2008], siehe Tabelle 4.1.

4.5 Einschub: Bäume und Baumdurchsuchungen

> Die Datenstruktur *Bäume* und Verfahren zum Durchlaufen (Traversieren) von Bäumen ist kein explizites Thema der Vorlesung AlgoDat (und dieses Skriptes). In der Vorlesung IntroProg wurden allgemeine Bäume ganz kurz eingeführt und nur die speziellen binären Suchbäume ausführlich besprochen. In Kapitel 9 wird die allgemeinere Datenstruktur der Graphen eingeführt, die Bäume als Spezialfälle umfasst. Da wir schon vorher Bäume und Verfahren zur Baumdurchsuchung abstrakt zur Veranschaulichung und Diskussion anderer Verfahren verwenden, werden die dabei verwendeten Begriffe hier kurz eingeführt.

Ein **Baum** besteht aus Knoten und einer Nachfolgerrelation auf den Knoten. Wir betrachten nur Bäume, bei denen ein Knoten als Wurzel ausgezeichnet ist. Knoten ohne Nachfolger heißen Blätter. Ein Knoten v_1 ist ein Vorgänger eines Knotens v_2, wenn v_2 ein Nachfolger von v_1 ist. Die Wurzel besitzt keinen, alle anderen Knoten genau einen Vorgänger. Ein Pfad in einem Baum ist eine Folge von Knoten $v_0, \ldots, v_K$, bei der jeder Knoten v_{k+1} Nachfolger von v_k für $k < K$ ist. Bei einem Baum muss jeder Knoten durch genau einen Pfad mit der Wurzel verbunden sein, d. h. ein Baum ist zusammenhängend und zyklenfrei.

Wir betrachten auch Bäume, die den Anschein erwecken könnten, Zyklen zu haben. Nimmt man bei einem Spiel die Anfangsstellung als Wurzel und die Spielzüge als Nachfolgerrelation, so kann es passieren, dass unterschiedliche Zugfolgen (= Pfade im Baum) zu derselben Spielstellung führen. Falls man die Spielstellung als Knoten auffasst, käme es hier zu einem Zyklus. Bei der Betrachtung von Suchprozessen würden wir diese Spielstellungen allerdings auf Grund der unterschiedlichen Zugfolgen als unterschiedliche Knoten auffassen und somit die Zyklenfreiheit erhalten. (Nichtsdestotrotz ist es aus Effizienzgründen oft wichtig, Zugfolgen die zu gleichen Spielstellungen führen, entsprechend zu berücksichtigen, um Redundanz bei der Suche zu vermeiden.)

Falls Lösungen zu einem Problem durch eine Folge von Auswahlschritten bzw. Entscheidungen konstruiert werden können, lässt sich die algorithmische Suche nach Lösungen als Baum darstellen. Dies wird im nächsten Abschnitt genauer ausgeführt. Dabei ist es entscheidend, in welcher Reihenfolge der Baum durchlaufen wird. Daher werden im Folgenden die beiden wichtigsten Verfahren zur **Baumtraversierung** (Durchlaufen aller Knoten eines Baumes) vorgestellt.

Diese Traversierung kann in einem als Datenstruktur gegebenen Baum stattfinden. Sie kann allerdings auch stattfinden, wenn nur die Regel zum Aufbau des Baumes gegeben ist. Bei einem Strategiespiel hat man selten den ganzen Baum der Zugkombinationen und Spielstellungen gegeben. Man weiß aber für jede Spielstellung ($\sim$ Knoten), welche Spielstellungen ($\sim$ Nachfolger) durch

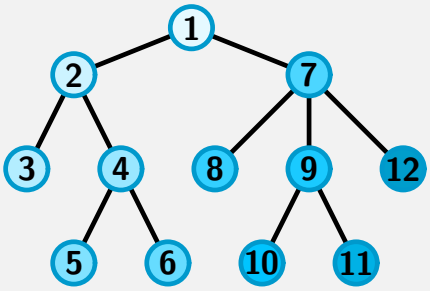
einen Zug erreicht werden können. Wir veranschaulichen die Traversierung hier bei gegebenen Bäumen.

Tiefensuche und Breitensuche in Bäumen

> Die Tiefensuche wurde in IntroProg in binären Suchbäumen als *Inorder-Tree-Walk* besprochen.

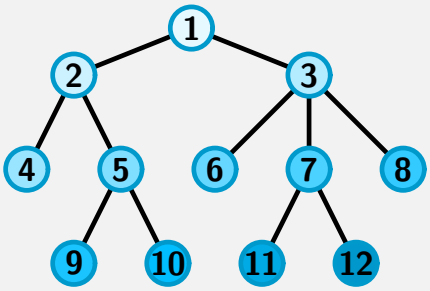
Tiefensuche in Bäumen

Bei der Tiefensuche (*depth-first search*) geht die Suche zunächst in die Tiefe und erst mit niedrigerer Präferenz in die Breite. Man geht von einem Knoten zu dem ersten Nachfolger, von dort wieder zu dessen erstem Nachfolger usw. bis man an ein Blatt stößt. Dann geht man einen Schritt zurück und besucht den nächsten Nachfolger.



Breitensuche in Bäumen

Bei der Breitensuche (*breadth-first search*) geht die Suche zunächst in die Breite und erst mit niedriger Präferenz in die Tiefe. Man geht von einem Knoten der Reihe nach zu allen direkten Nachfolgern und erst dann zu deren Nachfolgern.



Bei Bäumen (aber nicht bei Graphen!) lassen sich Tiefen- und Breitensuche gleich implementieren, wobei der einzige Unterschied in der Datenstruktur liegt, die

verwendet wird (Stapel für Tiefensuche und Warteschlange für Breitensuche), siehe Listing 4.3.

Listing 4.3 Pseudocode für Tiefen- und Breitensuche in Bäumen. Der einzige Unterschied besteht darin, welche Datenstruktur verwendet wird, ein Stapel für Tiefensuche und eine Warteschlange für Breitensuche. Achtung: Diese direkte Korrespondenz gilt nur für Tiefen- und Breitensuche *in Bäumen*. In Graphen ist eine nicht-rekursive Implementation der Tiefensuche nicht so leicht möglich.

```
1  // Tiefensuche
2  D : ein Stapel (Stack)
3  D ← {Wurzel}
4  while D ≠ ∅
5    v ← D.get()          // pop()
6    besuche Knoten v
7    for all w Nachfolger von v
8      D.put(w)           // push()
9    end
10 end
```

```
1  // Breitensuche
2  D : eine Warteschlange (Queue)
3  D ← {Wurzel}
4  while D ≠ ∅
5    v ← D.get()          // dequeue()
6    besuche Knoten v
7    for all w Nachfolger von v
8      D.put(w)           // enqueue()
9    end
10 end
```

Die Tiefensuche lässt sich auch sehr einfach rekursiv implementieren:

```
1  procedure DFS(v)
2    besuche Knoten v
3    for all w Nachfolger von v
4      DFS(w)
5    end
6  end
7
8  Aufruf durch: DFS( Wurzel )
```

4.6 *Backtracking* als Suche im Baum der Teillösungen

Wir veranschaulichen Backtracking als eine Suche im Baum der Teillösungen. Dieser Suchbaum ist nicht als Datenstruktur vorhanden. Er wird beim Backtracking (normalerweise) auch nicht explizit erzeugt, sondern nur durchlaufen. Zur Illustration nehmen wir den Suchbaum als bekannt an und schauen in welcher Reihenfolge er beim Backtracking durchlaufen wird. Wir betrachten nicht nur unterschiedliche Reihenfolgen des Durchlaufens, sondern auch unterschiedlich angeordnete Suchbäume.

Wir gehen für die Betrachtung von den folgenden Voraussetzungen aus, die beim Backtracking immer erfüllt sein müssen:

- Lösungen können als Lösungsbaum schrittweise entwickelt werden.
- Bei jeder Verzweigung ist eine Auswahl aus einer relativ begrenzten Auswahl an Möglichkeiten zu treffen.
- Die leere Lösung ist die Teillösung, bei der noch keine Entscheidung getroffen wurde. Alle Lösungen lassen sich aus der leeren Lösung durch eine endliche Anzahl von Lösungsschritten entwickeln.

In diesem Fall können wir die Suche als einen Baum der Teillösungen darstellen. In diesem Baum steht an der Wurzel die leere Teillösung. Jeder Knoten steht für die Entscheidung in einem bestimmten Lösungsschritt und jede Auswahlmöglichkeit entspricht einer Kante.
Hier kann man sich die Frage stellen, ob dies Vorgehen immer zu einem Baum führt (also einem Graphen ohne Zyklen). Dies muss nicht der Fall sein. Bei einem Spiel kann man z. B. mit unterschiedlichen Zugfolgen zu derselben Spielstellung gelangen. Wenn wir trotzdem von einem Such*baum* ausgehen, heißt dies, dass wir diese Situationen nicht miteinander identifizieren, sondern als unterschiedlich behandeln. In vielen Anwendungsfällen kann dies äußert ungünstig sein, da gleichwertige Teillösungen unabhängig voneinander behandelt werden, also dieselben Teilbäume vielfach durchsucht werden. In diesen Fällen kann es vorteilhaft sein, alle erzeugten Teillösungen zu speichern und bei einer neuen Teillösung zu prüfen, ob sie schon bekannt ist. Um die Gleichwertigkeit zweier Teillösungen effizient zu prüfen, wird oft Hashing eingesetzt, siehe Abschnitt 16. An dieser Stelle lassen wir das Problem gleichwertiger Teillösungen an unterschiedlichen Knoten des Suchbaums außer acht.

Beyond Backtracking

Es gibt unterschiedliche Möglichkeiten, das einfache Backtracking Verfahren für bestimmte Anwendungen effizienter zu gestalten. Einige davon haben wir in dem Sudoku Beispiel kennengelernt. In diesem Abschnitt werden sie als allgemeine Prinzipien der Suche im Baum der Teillösungen dargestellt und erweitert.
Die erste Erweiterungsmöglichkeit betrifft die Struktur des Suchbaumes – welcher Knoten welche Entscheidung repräsentiert. Hier unterscheiden wir die Varianten

- A-priori: Es steht vorher fest, welcher Knoten welche Entscheidung repräsentiert. Beispiele: Permutationen, Rösselsprung und Sudoku in der einfachen Variante (wähle nächstes freie Kästchen)

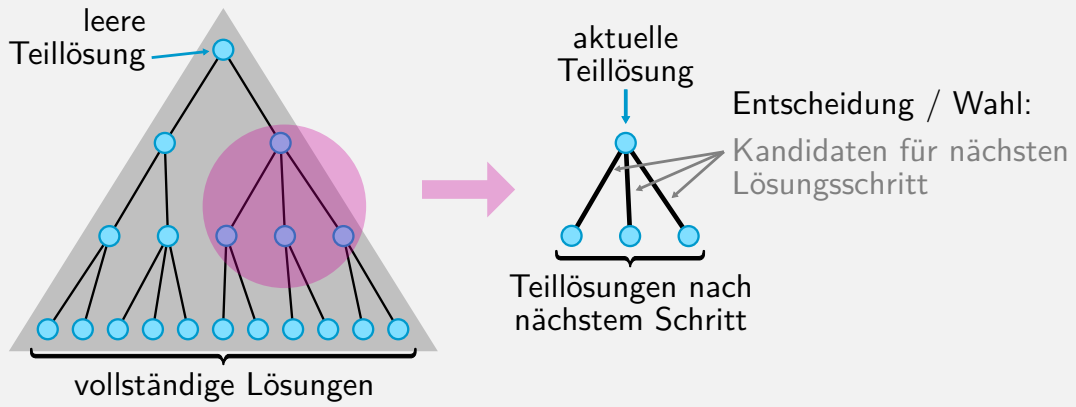


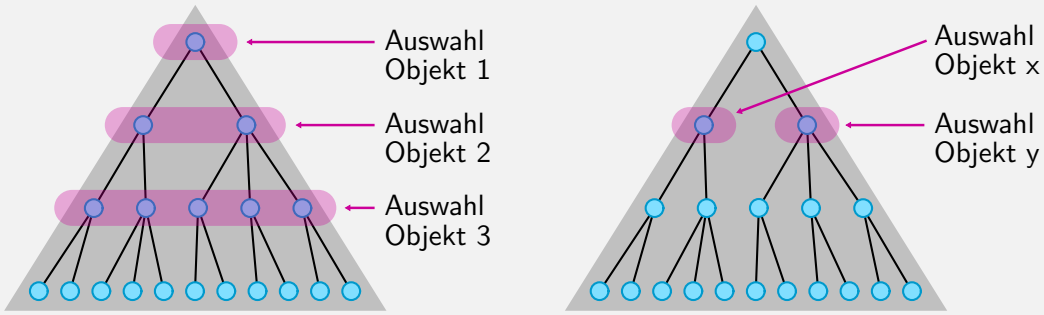
Abbildung 4.5 Baum der Teillösungen: Die Suche nach Lösungen wird als Suche im Baum der Teillösungen illustriert. Jeder Knoten entspricht einer Teillösung. Von dort verlaufen nach unten als Kanten die unterschiedlichen Möglichkeiten, die jeweilige Teillösung um einen Schritt zu verlängern. Die Blätter repräsentieren diejenigen Teillösungen, bei denen kein weiterer Auswahlschritt möglich ist. Die Blätter müssen nicht, wie hier dargestellt, auf einer Höhe liegen. Beim Rösselsprung enden z. B. die meisten Äste in Blättern, die eine Sackgasse und keine Lösung darstellen, nach weniger als 63 Zügen.

- On-the-fly: Während der Suche wird für jeden Knoten bestimmt, welche Entscheidung dort getroffen wird. Beispiel: Sudoku in der Erweiterung mit 'eingegrenzter Auswahl'

Üblicherweise versteht man unter *Backtracking* nur die Variante mit Tiefensuche und statischem Suchbaum. Breitensuche ist aufwändiger, weil parallele Teillösungen gespeichert werden müssen. Letzteres ist allerdings keine Einschränkung, wenn ein Baum oder Graph zur Durchsuchung gegeben ist.
Eine Dynamische Suche ist hilfreich, wenn nur bestimmte Lösungen gesucht werden. Dazu benötigt man ein Kriterium bzw. eine Heuristik, die angibt, in welcher Richtung die Lösungen (wahrscheinlich) zu finden sind. Diese Idee wird systematisch in Verfahren wie Branch-and-Bound und heuristischen Algorithmen verfolgt.

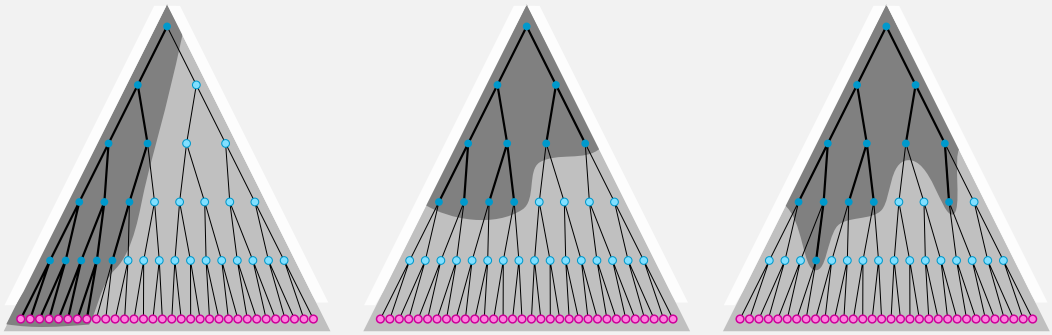
Ausblick auf Verbesserungen

Mit Backtracking lässt sich für viele Problemstellungen eine systematische Durchsuchung des Lösungsraumes leicht implementieren. Da der Lösungsraum oft zu groß ist, um in akzeptabler Laufzeit komplett durchsucht werden zu können, ist es angebracht zu überlegen, ob er sich beschränken lässt, ohne dass dadurch Lösungen verpasst werden. Ein typisches Beispiel ist das Ausnutzen



(a) A-priori Struktur **(b)** On-the-fly Struktur

Abbildung 4.6 Struktur des Suchbaums: Meist liegt bei einem Suchbaum im voraus fest, welche Entscheidung an welchem Knoten getroffen wird. Dies ist die in **(a)** gezeigte A-priori Struktur. Hier entscheidet allein die Tiefe eines Knoten darüber, welche Auswahl dort getroffen wird. Es kann aber geschickter sein, wie in **(b)** dargestellt, erst on-the-fly bei einem Knoten festzulegen, welche Auswahl getroffen wird. In dem Sudoku Beispiel wird in der Erweiterung 'eingegrenzte Auswahl' aus der Situation heraus entschieden, welches Kästchen als nächstes gefüllt wird.



(a) Tiefensuche **(b)** Breitensuche **(c)** 'Dynamische Suche'

Abbildung 4.7 Reihenfolge des Expandierens: (a) Die übliche Reihenfolge des Besuchens der Knoten ist beim Backtracking die Tiefensuche. Hierzu gab es verschiedene Beispiele in diesem Kapitel, wie die Wahl des nächsten freien Kästchens bei Sudoku-Lösen. **(b)** Man kann eine vollständige Durchsuchung des Baums der Teillösungen auch in der Reihenfolge der Breitensuche durchführen. Dies wäre eine naheliegender Ansatz, wenn die kürzeste Zugfolge zum Schachmatt in einem Schachrätsel gesucht ist. **(c)** Oft können Suchreihenfolgen günstiger sein, wenn der nächste Knoten, der expandiert wird, dynamisch ausgewählt wird, wie dies bei der 'vorausschauenden und eingegrenzten Auswahl' im Sudoku Beispiel gemacht wurde.

von 'Symmetrien'. Wie in dem Sudoku Beispiel können manchmal bestimmte Möglichkeiten ausgelassen werden, da sie zu keiner Lösung führen können.
Falls nur eine und nicht alle Lösungen gesucht werden, ist es sinnvoll, aussichtsreiche Kandidaten bevorzugt weiterzuverfolgen, damit früher eine Lösung gefunden wird.
Bei Optimierungsproblemen gibt es einen weiteren wichtigen Ansatz. Hier können Kandidaten weggelassen werden, die zu keiner *optimalen* Lösung führen können. Dazu stellt sich die Frage, wie man wissen kann, dass eine Teillösung nicht zu einer optimalen Lösung fortgesetzt werden kann. Dies kann erreicht werden, indem man Schranken für bestenfalls erreichbare Lösungswerte bestimmt. Diese Grundidee führt zum Branch-and-Bound Konzept.

4.7 Das Wichtigste in Kürze

- ***Backtracking*** kann bei Problemen angewendet werden, deren Lösung(en) in einer Folge von Einzelschritten erzeugt werden kann.
- Bei jeder Entscheidung für einen Einzelschritt werden alle Möglichkeiten mit Hilfe von Rekursion in einer Tiefensuche ausprobiert. So wird der Lösungsraum strukturiert und erschöpfend durchsucht.
- Die Grundelemente des Backtracking sind: (1) Kandidaten für den nächsten Schritt generieren, (2) Schritt ausführen, (3) Schritt rückgängig machen, (4) Festellen, ob eine Teillösung eine vollständige Lösung ist.
- Manchmal kann der Suchraum eingegrenzt werden, z. B. durch Ausnutzung von Symmetrien und durch Auslassen von Teillösungen, die zu keiner Lösung führen können.

Quellenverweise

Teile dieses Kapitels orientieren sich an der Darstellung in [Skiena, 2008], und zwar Abschnitt 4.1 an S. 234ff und 4.4 an S. 239ff.

KAPITEL 5

Gierige Algorithmen (*Greedy Algorithms*)

5.1 Einführung zu Greedy-Algorithmen

Bei dem Backtracking Konzept ging es vorwiegend um den Fall, dass man an vielen bzw. allen Lösungen eines Problems interessiert ist oder dass kein Ansatz für eine gezielte Suche, die ohne die Durchsuchung des gesamten Lösungsraums auskommt, bekannt ist. Den in diesem Kapitel besprochenen Gierigen Algorithmen (*Greedy Algorithms*) geht es um das komplementäre Anwendungsszenario. Bei ihnen geht es darum, nur eine einzige Lösung für spezielle Probleme zu finden, mit Hilfe einer besonders gezielten Suche. Ein Greedy Algorithmus betrachtet bei der Durchsuchung des Baumes der Teillösungen nur eine einzige Möglichkeit. Es wird also nur ein Pfad von der leeren zur vollständigen Lösung beschritten.
Bei Optimierungsproblemen ist man an Lösungen interessiert, die ein Optimalitätskriterium erfüllen (z. B. an einem Weg mit minimalen Kosten von s nach t). Im Idealfall findet ein Greedy Algorithmus die optimale Lösung. An jeder Stelle der Suche wird eine Entscheidung getroffen, die noch zu (mindestens) einer optimalen Lösung führen kann. Wenn dies immer garantiert werden kann, endet die Suche bei einer optimalen Lösung. Hierfür ist es erforderlich, dass eine Entscheidung, die auf Grund von *lokaler* Information getroffen wird, eine *globale* Bedeutung hat. Bei der mit "**?**" markierten Entscheidung in Abbildung 5.1 sind der bisher gewählte Pfad und die Abzweigungen (lokale Information) bekannt, während der Rest des Suchbaums und die Position der optimalen Lösungen (globale Information) unbekannt ist. Dennoch muss eine Entscheidung getroffen werden, die sicherstellt, dass sich im darunter liegenden Teilbaum noch eine optimale Lösung befindet. In der Abbildung ist dies nur für die rechte Abzwei-

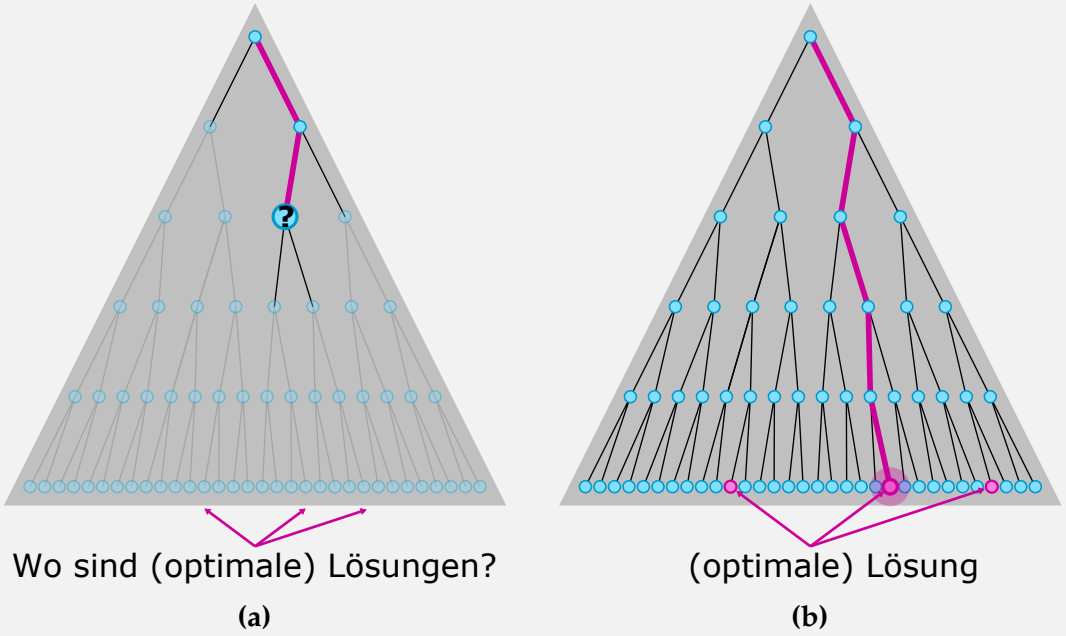


Abbildung 5.1 Greedy Suche im Suchbaum illustriert: (a) Auf der Suche nach einer optimalen Lösung muss an der mit "**?**" die dritte Entscheidung getroffen werden. Bekannt ist zu dieser Zeit nur der bisherige Pfad (in violett) und die Entscheidungsmöglichkeiten (schwarze Kanten). Dies nennt man 'lokale Information'. Unbekannt ist die Struktur des restlichen Baums (ausgegraute Kanten und Knoten) und die Lage der optimalen Lösungen, genannt "globale Information". In diesem Beispiel führt nur eine Entscheidung für die rechte Kante zu einer optimalen Lösung. **(b)** Ein Greedy Algorithmus wählt immer nur *eine* der Entscheidungsmöglichkeiten. Von dem Suchbaum wird also nur ein Pfad betrachtet. Im idealen Anwendungsfall führt die Greedy Auswahl trotz der Beschränkung zu einer optimalen Lösung.

gung der Fall, während die linke Abzweigung zu keiner optimalen Lösung führt. Es ist also der 'unwahrscheinliche' Fall, dass eine greedy Auswahl garantiert zu einer optimalen Lösung führt. Stellen Sie sich diese Situation aus der Ego-Sicht in einem Labyrinth vor. Mr. Greedy kennt nur seinen Pfad und die vor ihm liegende Abzweigung. Aus dieser sehr begrenzten lokalen Information muss er eine Entscheidung mit globaler Reichweite treffen - ein Wunder, wenn das klappt! Erfreulicherweise ist dieser Fall für einige besonders wichtige Anwendungen gegeben. In den Kapiteln über Graphen werden wir folgenden gierigen Algorithmen begegnen:

- Dijkstra Algorithmus für kürzeste Pfade
- Prim Algorithmus für minimale Spannbäume

- Kruskal Algorithmus für minimale Spannbäume

Wenn die Greedy Auswahl klappt (zu einer optimalen Lösung führt), ist es eine feine Sache: Ein Suchbaum mit exponentieller Größe (in der Anzahl der Entscheidungen = Höhe des Suchbaums) wird durch ein Greedy Verfahren auf lineare Größe (ein Pfad) beschränkt. Daher können Greedy Algorithmen eine enorme Steigerung der Effizienz bewirken. (Dies setzt voraus, dass die Entscheidung effizient getroffen werden kann.)
Manchmal werden auch Greedy Algorithmen betrachtet, die (im Normalfall) nicht zu einer optimalen Lösung führen. Man kann sich z. B. aus Effizienzgründen dafür entscheiden, eine suboptimale Lösung zu akzeptieren, wie in dem Kapitel 14 über Approximative Algorithmen. Außerdem werden suboptimale Greedy Algorithmen benutzt, um einen ersten Lösungskandidaten zu erzeugen, der durch ein weiteres Verfahren verbessert wird (Approximative Algorithmen und Randomisierte Algorithmen). Bei manchen Branch-and-Bound Algorithmen werden initiale, suboptimale Lösungen mit einem Greedy Verfahren erzeugt, um Schranken zu bestimmen, die die Suche nach optimalen Lösungen effizienter machen.
Diese Motivation wird im nächsten Abschnitt zu einem generellen Prinzip zusammengefasst. Danach werden einige Beispiele für Greedy Algorithmen besprochen. Um die Suche nach einem geeigneten Auswahlverfahren zu illustrieren, werden jeweils unterschiedliche Ansätze durchgesprochen und darauf geprüft, ob sie zweckdienlich sind. Dabei ist die Formulierung von jedem Ansatz recht einfach. Die Schwierigkeit bei Greedy Algorithmen besteht darin zu zeigen, dass die Auswahl tatsächlich zu einer optimalen Lösung führt.

5.2 Generelles Prinzip von Greedy-Algorithmen

Hier sind die Kerngedanken des vorhergehenden Abschnittes zusammengefasst. Ein **Greedy Algorithmus** geht folgendermaßen vor:

- Das Problem wird durch eine schrittweise Auswahl gelöst.
- In jedem Schritt wird eine Auswahl getroffen, die nach einem lokalen Kriterium aussichtsreich ist, und einer optimalen Lösung nicht im Wege steht.

Da immer nur eine einzige Auswahl in jedem Schritt berücksichtigt wird, bewirkt die *Greedy* Suche eine immense Einschränkung des Suchraums gegenüber *Backtracking*. Teillösungen jenseits des eingeschlagenen Pfades werden ignoriert.
Allerdings gibt es bei weitem nicht für jedes Problem einen Greedy-Algorithmus. Und nicht jeder Greedy-Algorithmus führt zur optimalen Lösung (die wenigsten tun es). Wenn es zu einem Problem einen Greedy-Algorithmus gibt, der optimal ist, dann ist dies oft eine besonders effiziente Lösung. Dem könnte

entgegenstehen, dass die Auswahl selbst sehr komplex ist. Dies ist ein seltener Fall. Greedy-Algorithmen sind meist einfach zu implementieren. Der Beweis der Optimalität ist oft schwierig.

5.3 Greedy Beispiel: Intervallauswahl

Als erstes Beispiel betrachten wir eine Variante der Ablaufplanung, die **Intervallauswahl** (*interval scheduling*). Dabei gibt es Anfragen $1, \ldots, n$ zur Nutzung einer Ressource (z. B. Raum, Prozessor, Messgerät) mit Anfangszeiten s_k und Endzeiten f_k entsprechend dem Zeitintervall $[s_k, f_k)$. Nun besteht die Aufgabe darin, eine Belegung für die Ressource zu erstellen, die möglichst viele Anfragen (ohne Doppelbelegungen) erfüllt.

Definition: Kompatibilität von Intervallen

Wir nennen eine Menge von Intervallen **kompatibel**, wenn es keine Konflikte (Überlappungen) zwischen den Intervallen gibt. Formale Details: Wir haben die Intervalle als rechts-offen definiert. Wenn die Endzeit eines Intervalles mit der Anfangszeit des nächsten Intervalles übereinstimmt, sind sie kompatibel.

Damit kann die Aufgabe auch so formuliert werden: Wähle eine möglichst große Menge von kompatiblen Intervallen aus.
Ein Beispiel für eine Intervallauswahl ist in Abbildung 5.2 dargestellt.

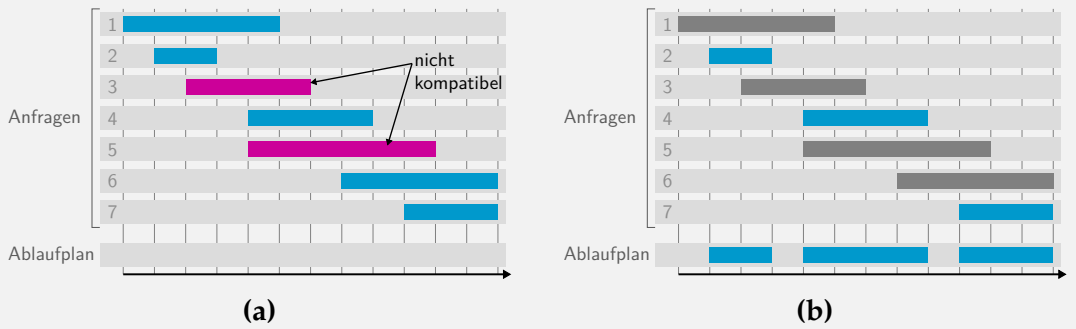


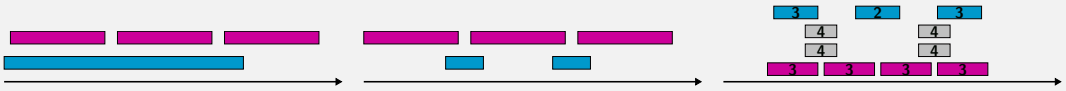
Abbildung 5.2 Die Aufgabe der Intervallauswahl: (a) Es gibt sieben 'Anfragen', die jeweils durch ein Zeitintervall charakterisiert sind. Der Ablaufplan soll möglichst viele Anfragen erfüllen. Dabei darf es keine Doppelbelegungen geben. Von zwei nicht-kompatiblen Intervallen (violett markiert) kann also höchstens eins übernommen werden. **(b)** zeigt einen möglichen Ablauf, der drei Anfragen erfüllt.

Nun soll für die Aufgabe ein Greedy Algorithmus gefunden werden. Um das Erstellen eines Ablaufplanes in einzelne Auswahlschritte zu zerlegen, können

die Anfragen in einer gewissen Reihenfolge durchlaufen werden. Es wird jede Anfrage akzeptiert, falls sie zu den bisher gewählten kompatibel ist. Bei diesem Vorgehen ist die Reihenfolge des Durchlaufens der Anfragen entscheidend. Wir betrachten unterschiedliche Sortierungen und diskutieren ihre Tauglichkeit.

- **[Frühester-Start]** Anfragen aufsteigend nach s_k sortiert
- **[Frühestes-Ende]** Anfragen aufsteigend nach f_k sortiert
- **[Kürzestes-Intervall]** Anfragen aufsteigend nach $f_k - s_k$ sortiert
- **[Wenigste-Konflikte]** Anfragen aufsteigend sortiert nach der Anzahl von anderen Anfragen, mit denen sie nicht kompatibel sind.

Welche Sortierungen resultieren in einem geeigneten Greedy-Algorithmus? In diesem Fall ist bei drei Strategien leicht an einem Gegenbeispiel zu sehen, dass sie nicht optimal sind, siehe Abbildung 5.3
Die Gegenbeispiele in Abbildung 5.5 zeigen, dass die drei vorgeschlagenen Strategien *Frühestes-Start*, *Kürzestes-Intervall* und *Wenigste-Konflikte* offensichtlich nicht optimal sind.



(a) Frühester-Start **(b)** Kürzestes-Intervall **(c)** Wenigste-Konflikte

Abbildung 5.3 Gegenbeispiele zur Optimalität von Ansätzen zur Intervallauswahl: Die Strategie 'Frühster-Start' scheitert in Beispiel **(a)**, da das Intervall mit der frühsten Anfangszeit besonders lang ist und somit viele andere, kürzere blockiert. **(b)** Die Bevorzugung kurzer Intervalle ist allerdings auch nicht zielführend. Zwei kurze Intervalle blockieren hier drei lange. **(c)** Es ist etwas schwieriger zu sehen, dass auch die Präferenz für Intervalle mit den wenigsten Konflikten nicht unbedingt zu optimalen Lösungen führt. Hier zeigt die Zahl in jedem Intervall an, mit wievielen der anderen es in Konflikt steht. Wenn zunächst das Intervall mit den wenigsten Konflikten (2) gewählt wird, können danach nur noch zwei weitere, kompatible Intervalle ausgewählt werden, also drei insgesamt (blaue Markierung). Die optimale Auswahl beinhaltet vier Intervalle (violette Markierung).

5.3.1 Optimalität des Greedy-Algorithmus "Frühestes-Ende"

Wie steht es nun mit dem Greedy-Algorithmus *Frühestes-Ende*? Auch wenn man kein einfaches Gegenbeispiel zur Optimalität findet, ist dies natürlich kein Beweis für die Optimalität. Ein solcher Beweis ist nicht ganz leicht zu führen. Wir führen zunächst ein paar Notationen und Konventionen ein.

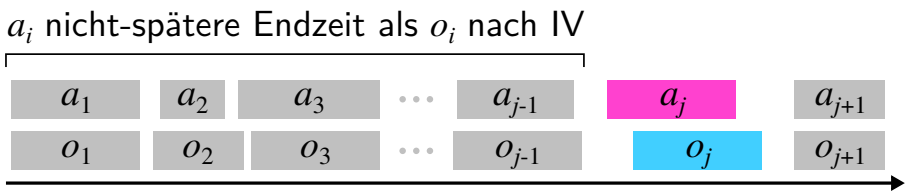
- Wir identifizieren ein Intervall mit seinem Index k.
- Für ein Intervall (bzw. Index) a schreiben wir $s(a)$ für seine Start- und $f(a)$ für seine Endzeit.

Diese Notation erleichtert es, unterschiedliche Intervallauswahlen und -reihenfolgen gleichzeitig zu betrachten.

Optimalität des Greedy-Algorithmus Frühestes-Ende

Durchlaufe die n Intervalle nach der aufsteigenden Reihenfolge ihrer Endzeitpunkte und wähle jedes aus, das kompatibel mit den bisher gewählten ist.
Dieser Algorithmus liefert eine optimale Intervallplanung in einer Laufzeit in $\boldsymbol{O}(n \log n)$.

Beweis. Seien $a_1, \ldots, a_k$ die ausgewählten Intervalle in der Reihenfolge der Auswahl und $o_1, \ldots, o_m$ die Intervalle einer optimalen Intervallplanung, ebenfalls nach Endzeiten sortiert. Es gilt also $f(a_i) < f(a_j)$ und $f(o_i) < f(o_j)$ für $i < j$.
Wir zeigen zunächst durch Induktion nach j, dass $f(a_j) \leq f(o_j)$ gilt:
Induktionsanfang: Für $j = 1$ gilt dies nach unserem Auswahlkriterium 'frühste Endzeit'.
Induktionsschritt: Nach IV können wir voraussetzen, dass $f(a_i) \leq f(o_i)$ für alle $i < j$ gilt. Anfrage o_j ist kompatibel mit $a_1, \ldots, a_{j-1}$, denn es gilt $s(o_j) \geq f(o_{j-1}) \geq^{\text{IV}} f(a_{j-1})$. Daher wählt der Algorithmus eine Anfrage a_j, deren Endzeit nicht später als $f(o_j)$ ist. Es gilt also $f(a_j) \leq f(o_j)$, was zu zeigen war. □



Daraus folgt $k = m$: Für $k < m$ wäre Intervall o_{k+1} kompatibel mit $a_1, \ldots, a_k$: $s(o_{k+1}) \geq f(o_k) \geq f(a_k)$. Also hätte der Algorithmus nach a_k das Intervall o_{k+1} auswählen können und hätte nicht nach k Intervallen gestoppt. Die

Planung gemäß Algorithmus Frühestes-Ende beinhaltet also genauso viele Intervalle, wie die optimale Planung, ist somit auch optimal. ■

Dieser optimale Algorithmus lässt sich wie folgt in Pseudocode formulieren:

Listing 5.1 Pseudocode des Greedy Algorithmus für *Interval Scheduling*

```
1 // Gegeben: n Intervalle
2 sortiere Intervalle nach Endzeiten → I_1, ..., I_n
3 R ←∅
4 for j = 1 to n
5   if Intervall I_j ist kompatibel mit R
6     füge I_j zu R hinzu
7   end
8 end
```

Für die Kompatibilitätsprüfung kann einfach der Endzeitpunkt des letzten Intervals von R gespeichert werden. Das Durchlaufen der Intervalle nach Endzeitpunkten kann über eine Vorrangwarteschlange PQ implementiert werden. Dadurch ergibt sich die Laufzeit $\boldsymbol{O}(n \log n)$. Für das sortierte Durchlaufen der Intervalle definieren wir eine Klasse `Interval` und eine Vergleichsoperation.

5.3.2 Implementation des Greedy Algorithmus für *Interval Scheduling*

Zunächst implementieren wir eine Klasse `Interval`, die Anfangs- und Endzeit speichern und zurückgeben kann. Dann definieren wir eine Methode, die zwei Intervalle gemäß ihrer Endzeitpunkte vergleicht, als `Comparator` Schnittstelle.

Listing 5.2 Implementation des Greedy Algorithmus für *Interval Scheduling*

```
public class Interval {
  private double s;
  private double f;

  public Interval(double s, double f) {
    this.s = s;
    this.f = f;
  }

  public double start()  { return s; }
  public double finish() { return f; }
}
```

```
import java.util.Comparator;
public class IntervalCompareFinishTime implements Comparator<Interval> {
  @Override
  public int compare(Interval i1, Interval i2) {
    return Double.compare(i1.finish(), i2.finish());
  }
}


public abstract class IntervalScheduling {

  public static Iterable<Interval>
          intervalScheduling(Iterable<Interval> intervals) {
    PriorityQueue<Interval> pq =
        new PriorityQueue<>(new IntervalCompareFinishTime());
    for (Interval i : intervals) {
      pq.add(i);
    }

    Queue<Interval> queue = new LinkedList<>();
    double finishTime = 0;    // Endzeit des letzten gewählten Intervalls
    while (!pq.isEmpty()) {
      Interval iv = pq.poll();           // Intervall mit Frühestem Ende
      if (iv.start() >= finishTime) {    // Falls kompatibel
        queue.add(iv);                   // wähle es aus und
        finishTime = iv.finish();        // aktualisiere Endzeit
      }
    }
    return queue;
  }
}
```

5.4 Greedy Beispiel: Intervallverteilung

Dieses Beispiel baut auf dem vorherigen in Abschnitt 5.3 auf. Nun sind die Anfragen als Anforderungen zu interpretieren, die *alle* erfüllt werden müssen. Dabei können die angeforderten Intervalle auf parallele Ressourcen verteilt werden, mit dem Ziel möglichst wenige Ressourcen zu benutzen (*Interval Partitioning Problem* oder *Interval Coloring Problem*).

Genauer: Die Intervalle $1, \ldots, n$ mit Anfangszeiten s_k und Endzeiten f_k sollen auf möglichst wenige Ressourcen verteilt werden, wobei die Intervalle innerhalb jeder Ressource kompatibel sein müssen, siehe Abbildung 5.4.

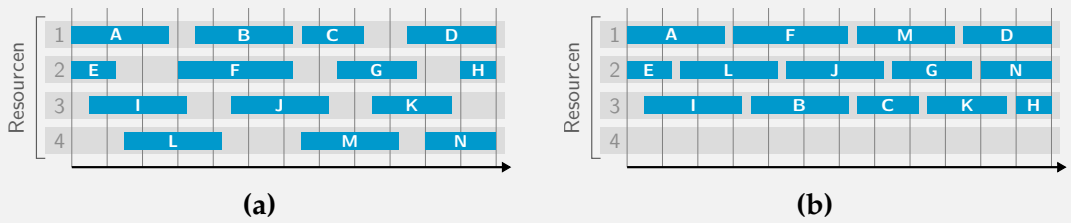


Abbildung 5.4 Die Aufgabe der Intervallverteilung: Die Intervalle **A** bis **N** sollen auf möglichst wenige Ressourcen verteilt werden. **(a)** zeigt eine Verteilung auf vier Ressourcen und **(b)** zeigt eine optimale Lösung, die nur drei Ressourcen nutzt.

Gierige Auswahl bei der Intervallverteilung

Wie bei dem Beispiel der Intervallauswahl im vorherigen Abschnitt betrachten wir die Intervalle in einer gewissen Sortierung. Die jeweilige Anforderung wird dann der ersten Ressource zugewiesen, zu deren bisherigen Intervallen sie kompatibel ist.
Wir betrachten dieselben vier Sortierungen wie bei der Intervallauswahl und überlegen ob einer der resultierenden Greedy-Algorithmen optimal ist.

- **[Frühester-Start]** Intervalle aufsteigend nach s_k sortiert
- **[Frühestes-Ende]** Intervalle aufsteigend nach f_k sortiert
- **[Kürzestes-Intervall]** Intervalle aufsteigend nach $f_k - s_k$ sortiert
- **[Wenigste-Konflikte]** Intervalle aufsteigend sortiert nach der Anzahl von anderen Intervallen, mit denen sie in Konflikt stehen.

Die Gegenbeispiele in Abbildung 5.5 zeigen, dass die drei vorgeschlagenen Strategien *Frühestes-Ende*, *Kürzestes-Intervall* und *Wenigste-Konflikte* offensichtlich nicht optimal sind.

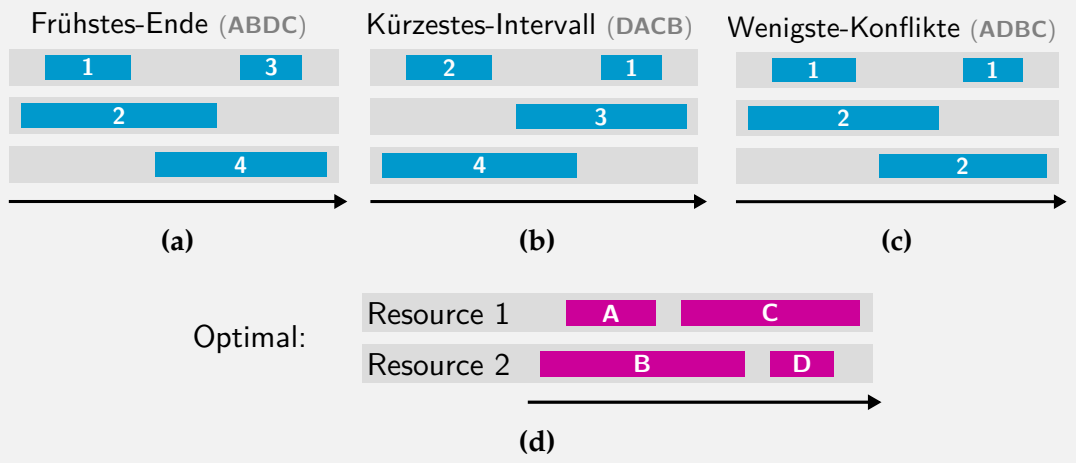


Abbildung 5.5 Gegenbeispiele zur Optimalität: Die drei Greedy-Algorithmen **(a)** *Frühestes-Ende*, **(c)** *Kürzestes-Intervall* und **(c)** *Wenigste-Konflikte* benötigen drei an Stelle der optimalen zwei Ressourcen **(d)**. Sie sind also nicht optimal.

5.4.1 Optimalität der Gierigen Strategie "Frühester-Start"

Es bleibt die Variante mit der vierten Sortierung *Frühester-Start* und in der Tat führt sie zu einem optimalen Greedy Verfahren. Wir definieren den Algorithmus zunächst noch einmal exakt und machen uns dann an den Beweis seiner Optimalität.

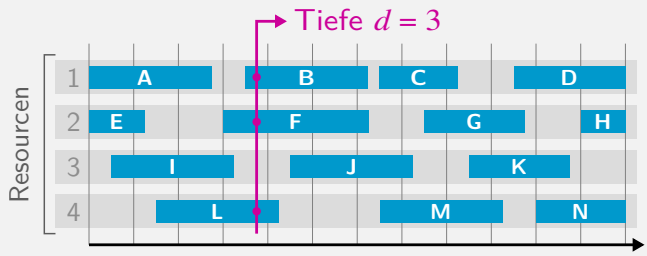
Greedy-Algorithmus *Frühester-Start* zur Intervallverteilung

Durchlaufe die Intervalle $I_1, \ldots, I_n$ nach der aufsteigenden Reihenfolge ihrer Startzeitpunkte und weise jedes einer Ressource zu, in der das Intervall keinen Konflikt verursacht. Nehme eine neue Ressource, falls das Intervall in keine der existierenden Ressourcen passt.

Zur Vorbereitung des Beweises wird die Tiefe einer Intervallmenge definiert. Diese Tiefe liefert ein Strukturargument, mit dem sich die Optimalität von *Frühester-Start* zeigen lässt.

Definition: Tiefe einer Intervallmenge

Zu einer Menge von Intervallen $S = \{I_1, \ldots, I_n\}$ definieren wir die **Tiefe** (*depth*) $d(S)$ als die größte Anzahl von Intervallen, die einen gemeinsamen Punkt umfassen.



Strukturelle Grenze bei der Intervallverteilung

Offensichtlich gilt die folgende Aussage:

Tiefe als untere Grenze bei der Intervallverteilung

Die Anzahl der Ressourcen, die bei einer Intervallverteilung benötigt werden, ist mindestens die Tiefe der Menge der zu verteilenden Intervalle.

Beweis: Seien $I_1, \ldots, I_d$ die Intervalle, die einen gemeinsamen Punkt umfassen. Diese Intervalle müssen auf d unterschiedliche Ressourcen verteilt werden, da es andernfalls zu einem Konflikt in dem gemeinsamen Punkt kommt. □

Erstaunlich ist hingegen, dass dies auch die obere Grenze ist. Es kann also immer eine Intervallverteilung gefunden werden, deren Anzahl an Ressourcen der Tiefe der vorgegebenen Intervallmenge entspricht. Und unser einfacher Greedy-Algorithmus *Frühester-Start* findet sie! Diese Tatsache wird in dem Optimalitätsbeweis gezeigt. Dabei wird Bezug auf den folgenden Pseudocode genommen.

Listing 5.3 Pseudocode des Greedy-Algorithmus *Frühester-Start*

```
1  // Gegeben: n Intervalle mit Start- und Endzeiten s_i und f_i
2  sortiere Intervalle nach Anfangszeiten → I_1, ..., I_n
3  K = 0
4  for j = 1 to n
5     if Intervall I_j ist kompatibel zu einer Ressource R(k) für ein k < K
6        füge I_j zu R(k) hinzu
7     else
8        initialisiere neue Ressource R(K)
9        füge I_j zu R(K) hinzu
10       K ←K + 1
11    end
```

12 **end**

Optimalität des Greedy Algorithmus für *Interval Partitioning*

Der Algorithmus *Frühester-Start* in Listing 5.3 liefert eine optimale Intervallverteilung und kann mit einer Laufzeit in $\boldsymbol{O}(n \log n)$ implementiert werden.

Beweis. Die Abfrage in Zeile 5 stellt sicher, dass alle Ressourcen kompatibel, also ohne Intervallkonflikte, sind. Nun zeigen wir noch, dass *Frühester-Start* nicht mehr als d Ressourcen benötigt, wobei d die Tiefe der gegebenen Menge von Intervallen ist.
Wenn eine neue Ressource in Zeile 8 für j eröffnet wird, müssen die letzten Intervalle in allen vorhandenen K Ressourcen den Anfangspunkt von I_j umfassen: Wegen der Sortierung nach Anfangszeitpunkten müssen jene Intervalle früher starten, und wegen der Inkompatibilität enden sie später als s_j.
Es gibt also $K + 1$ Intervalle, die einen gemeinsamen Punkt umfassen. Somit ist die neu eröffnete Ressource innerhalb der Optimalitätsgrenze: $K + 1 \leq d$.
□

Die Laufzeit wird anhand einer Java Implementation untersucht.

5.4.2 Implementation des Greedy Algorithmus für *Interval Partitioning*

Wir implementieren eine Klasse `Ressource`, die die Intervalle einer Ressource speichert. Außerdem wird der späteste Endzeitpunkt der enthaltenen Intervalle gespeichert. Und diesmal wird ein `Comparator` benötigt, der zwei Intervalle gemäß ihrer Startzeitpunkte vergleicht.

```
public class Ressource implements Comparable<Ressource> {

  private double f;
  public Queue<Interval> intervals;

  public Ressource(Interval iv) {
    f = iv.finish();
    intervals = new LinkedList<>();
    intervals.add(iv);
  }

  public double getFinish()        { return f;  }
  public void setFinish(double f)  { this.f= f; }

  @override
```

```
  public int compareTo(Ressource that) {
    return Double.compare(this.f, that.f);
  }
}


import java.util.Comparator;
public class IntervalCompareStartTime implements Comparator<Interval> {
  @override
  public int compare(Interval i1, Interval i2) {
    return Double.compare(i1.start(), i2.start());
  }
}
```

Das Durchlaufen der Intervalle nach Anfangszeitpunkten wird wie bei der Intervallauswahl über eine Vorrangwarteschlange PQ implementiert. Als Vergleichsoperation für die PQ wird die Startzeit benutzt.

Listing 5.4 Implementation des Greedy-Algorithmus Frühester-Start

```
1  public static Iterable<Ressource> earliestStart(Iterable<Interval> intervals) {
2    PriorityQueue<Interval> pq =
3        new PriorityQueue<>(new IntervalCompareStartTime());
4    for (Interval iv : intervals) {
5      pq.add(iv);
6    }
7
8    PriorityQueue<Ressource> ressources = new PriorityQueue<>();
9    while (!pq.isEmpty()) {
10     Interval iv = pq.poll();
11     Ressource res = ressources.peek();   // null at first iteration
12     if (res != null && iv.start() >= res.getFinish()) {
13       res.intervals.enqueue(iv);
14       ressources.poll();                 // remove res
15       res.setFinish(iv.finish());        // update finish time
16       ressources.add(res);               // and re-add
17     } else {
18       ressources.add(new Ressource(iv));
19     }
20   }
21   return ressources;
22 }
```

Untersuchung der Laufzeit von Frühester-Start

Laufzeit des Greedy-Algorithmus für *Interval Partitioning*

Der Implementation von *Frühester-Start* in Listing 5.4 zur optimalen Intervallverteilung hat eine Laufzeit in $\boldsymbol{O}(n \log n)$ für n Intervalle.

Beweis. Zur Sortierung wird jedes Interval per `add()` in eine PQ eingefügt. Für n Intervalle ergibt dies eine Laufzeit in $\boldsymbol{O}(n \log n)$.
Die Schleife (Zeilen 8–18) wird für jedes der n Intervalle einmal durchlaufen.

- Die verwendeten PQ Methoden `poll()` und `add()` haben eine Laufzeit in $\boldsymbol{O}(\log n)$, da die PQs höchstens n Elemente beinhalten.
- Die Methode `peek()` (Zeile 10) hat konstante Laufzeit.
- Die Methode `enqueue()` von `Queue` (Zeile 12) hat konstante Laufzeit.

Damit hat diese Schleife und der gesamte Algorithmus eine Laufzeit in $\boldsymbol{O}(n \log n)$. □

5.5 Greedy Beispiel: Planung mit minimalen Verspätungen

Bei der **Planung mit minimalen Verspätungen** (*Schedule to Minimize Lateness*) geht es wieder um eine Ablaufplanung mit einer einzigen Zeitschiene. In diesem Fall sind als Anforderungen Intervalllängen t_j und ein spätester Endzeitpunkt d_j (*deadline*) vorgegeben. Durch Festsetzen des Anfangszeitpunktes s_j der Anforderung j entsteht ggf. eine Verspätung (*lateness*) $l_j = \max\{0, s_j + t_j - d_j\}$. Das Ziel bei der "Planung mit minimalen Verspätungen" ist es, einen Ablauf zu erstellen, der die Maximalverspätung $L = \max_j l_j$ minimiert, siehe Abbildung 5.6.

Gierige Auswahl bei Planung mit minimalen Verspätungen

Die Anforderungen werden in einer gewissen Ordnung (siehe Liste) ohne Leerlaufzeiten arrangiert. Diesmal sind keine Start- und Endzeiten vorgegeben, sondern Dauer t_i und Deadline d_i. Daher gibt es andere Optionen für die Sortierungen, die der Greedy-Auswahl zugrunde liegen können.

- **[Kürzeste-Dauer]** Anforderungen aufsteigend nach t_j sortiert
- **[Kleinste-Pufferzeit]** Anforderungen aufsteigend nach $d_j - t_j$ sortiert
- **[Früheste-Deadline]** Anforderungen aufsteigend nach d_j sortiert

Wieder stellt sich die Frage, ob eine dieser Sortierungen zu einem gierigen Algorithmus führt, der die optimale Lösung findet und wenn ja, welcher. Die

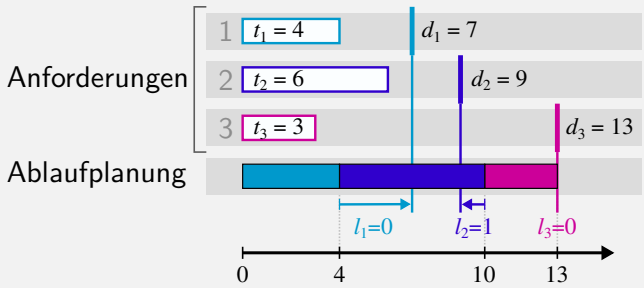


Abbildung 5.6 Planung mit minimalen Verspätungen: Der gezeigte Ablauf (1-2-3) verursacht eine Verspätung von einer Zeiteinheit, da das Ende des zweiten Intervalls bei 10 liegt, die Deadline in der Anforderung ist jedoch $d_2 = 9$. Das dritte Zeitintervall endet genau zu seiner Deadline, und das erste 3 Zeiteinheiten vor seiner Deadline.

zwei Sortierungen *Kürzeste-Dauer* und *Kleinste-Pufferzeit* fallen durch, wie die Gegenbeispiele in Abbildung 5.7 zeigen.

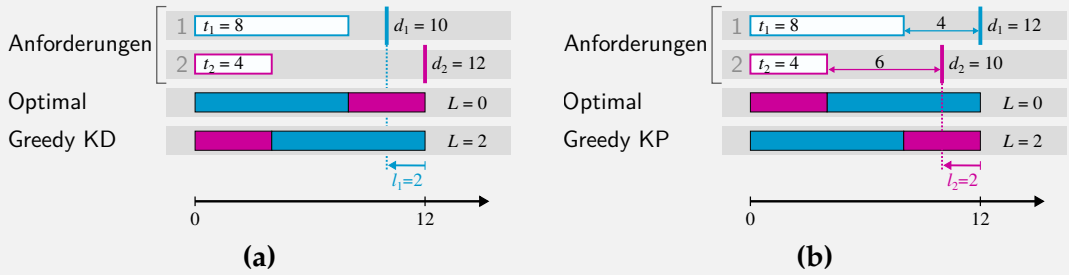


Abbildung 5.7 Gegenbeispiele zur Optimalität: Die Ablaufplanung der zwei Greedy-Algorithmen **(a)** *Kürzeste-Dauer* und **(b)** *Kleinste-Pufferzeit* führen beide zu einer Verspätung von zwei Zeiteinheiten, während die optimale Planung ohne Verspätung auskommt. Also sind diese beiden Strategien nicht optimal.

5.5.1 Optimalität der Gierigen Strategie "Früheste-Deadline"

Greedy-Algorithmus *Früheste-Deadline* zur Planung mit minimaler Verspätung

Plane alle Anfragen in der aufsteigenden Sortierung ihrer *Deadlines*.

Der Beweis der Optimalität dieses Greedy Algorithmus erfolgt durch ein sogenanntes **Austauschargument**: Wir nehmen einen beliebigen optimalen Ablauf als

gegeben an. Dieser wird schrittweise in das Ergebnis des Greedy Algorithmus umgeformt, ohne dass dabei die Optimalität verloren geht.
Dafür nehmen wir im Folgenden an, dass die Aufgaben $1, \ldots, n$ gemäß ihrer Deadline aufsteigend sortiert sind. Bei dem Beweis spielen Vertauschungen eine wichtige Rolle.

> Wir sprechen von einer **Inversion**, wenn in einem Ablauf eine Aufgabe i vor einer Aufgabe j eingeplant ist, obwohl die Deadline von j vor der Deadline von i liegt, also $d_j < d_i$.

Der von *Früheste-Deadline* erzeugte Ablauf hat offensichtlich keine Inversionen.

Optimalität von *Früheste-Deadline*

Die gierige Strategie *Früheste-Deadline* ergibt einen optimalen Ablauf, d. h. das Maximum der Verspätungen ist so klein wie möglich.

Auf den folgenden Seiten werden folgende Aussagen bewiesen:

- **Existenzbeweis:** Es gibt einen optimalen Ablauf ohne Inversionen und Leerlauf.
- **Gleichwertigkeit:** Alle Abläufe ohne Inversionen und Leerlauf haben dieselbe maximale Verspätung, also sind sie im Sinne der Optimalität gleichwertig.

Da der Ablauf, der durch *Früheste-Deadline* erzeugt wird, keine Inversionen und keinen Leerlauf hat, folgt aus den beiden Aussagen die Optimalität. □

Existenzbeweis

Es gibt einen optimalen Ablauf ohne Inversionen und Leerlauf.

Es werden zunächst Teilaussagen **1** bis **4** bewiesen und diese am Ende zum Beweis der Existenzaussage zusammengeführt.

1 Es gibt einen optimalen Ablaufplan ohne Leerlauf.

- **Beweis.** Wir nehmen einen optimalen Ablaufplan mit Leerlauf als gegeben an. Dann können alle auf den Leerlauf folgenden Intervalle entsprechend vorgezogen werden. Dadurch kann sich keine Verspätung vergrößern. Also ist der resultierende Ablaufplan ohne Leerlauf auch optimal.
- Wenn es also überhaupt einen optimalen Ablaufplan gibt, dann gibt es auch einen ohne Leerlauf. Da es nur endlich viele Ablaufpläne ohne

Leerlauf gibt (Permutationen der Aufgaben), muss (mindestens) einer von ihnen optimal sein.

Sei nun ein optimaler Ablauf $\boldsymbol{O}$ ohne Leerlauf gegeben. Wir zeigen durch die Beweise von drei Aussagen, dass Inversionen von $\boldsymbol{O}$, sofern überhaupt vorhanden, durch Vertauschung schrittweise beseitigt werden können, ohne die Optimalität zu verlieren.
Sei $\boldsymbol{O}$ ein optimaler Ablauf ohne Leerlauf aus **1** mit Inversion.

2 Es gibt Aufgaben $i < j$ mit $d_j < d_i$, die im Ablauf $\boldsymbol{O}$ direkt aufeinander folgen.

▶ **Beweis.** Wir betrachten eine Inversion $h < k$ mit $d_k < d_h$. Wenn man im Ablauf $\boldsymbol{O}$ von Aufgabe h schrittweise die nachfolgenden Aufgaben betrachtet, muss irgendwann (spätestens bei Index k) die erste Aufgabe kommen, deren Deadline kleiner (bzw. früher) ist, als die Dealine der vorherigen Aufgabe.

▶ Dieser Übergang ist das aufeinanderfolgende Paar $i < j$ mit $d_j < d_i$.

3 Wenn in der Situation von **2** die Aufgaben i und j vertauscht werden, erhalten wir einen Ablauf mit einer Inversionen weniger.

▶ **Beweis.** Das Paar (i, j) war eine Inversion im ursprünglichen Ablauf, die im neuen Ablauf nicht mehr existiert.

▶ Da i und j direkt aufeinander folgen, werden keine neuen Inversionen erzeugt.

4 Die *maximale* Verspätung wird durch die Vertauschung nicht vergrößert.

▶ **Beweis.** Da zwei direkt aufeinander folgenden Aufgaben i und j lokal vertauscht werden, bleiben die Verspätungen von allen anderen Aufgaben gleich. Daher brauchen nur die Verspätungen von i und j untersucht werden.

▶ Wir bezeichnen Anfangszeit und Verspätung des urspünglichen Ablaufs mit s_i, s_j und l_i, l_j und die Werte nach Vertauschung mit s'_i, s'_j und l'_i und l'_j. Offensichtlich gibt $l_i < l_j$. Wenn wir $l'_i < l_j$ und $l'_j < l_j$ gezeigt haben, wissen wir, dass die maximale Verspätung durch die Inversion nicht vergrößert wird.

▶ Die Aussage $l'_j < l_j$ ist klar, da Aufgabe j vorgezogen wird (und die Deadline gleichbleibt). Es bleibt $l'_i < l'_j$ zu zeigen:

$$
\begin{aligned}
l'_i &= s'_i + t_i - d_i && \text{Definition von Verspätung} \\
&= s'_j + t_j + t_i - d_i && \text{wegen } s'_i = s'_j + t_j \\
&= s_i + t_j + t_i - d_i && \text{wegen } s'_j = s_i \\
&= s_j + t_j - d_i && \text{wegen } s_j = s_i + t_i \\
&\leq s_j + t_j - d_j = l_j && \text{wegen } d_j < d_i
\end{aligned}
$$

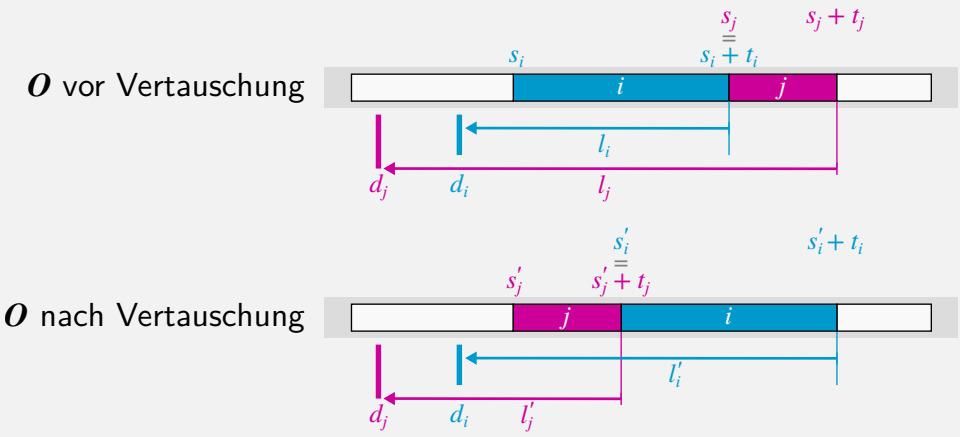


Abbildung 5.8 Illustration des Beweises von Aussage 4

Aus den Aussagen 1 – 4 folgt die behauptete Existenz:

- Laut 1 gibt es einen optimalen Ablauf ohne Leerlauf. Dieser kann nur endlich viele Inversionen, nämlich maximal $\binom{n}{2}$, enthalten.
- Alle direkt aufeinander folgenden Inversionen können gemäß 3 durch Vertauschung beseitigt werden.
- Dabei bleibt die optimale Verspätung nach 4 erhalten.
- Wenn es keine direkt aufeinander folgenden Inversionen mehr gibt, kann es laut 2 überhaupt keine Inversionen mehr geben.
- Wir haben also einen optimalen Ablauf ohne Inversionen und ohne Leerlauf. ■

Nachdem die Existenz eines Ablaufes ohne Inversionen und Leerlauf bewiesen wurde, wird gezeigt, dass alle Abläufe dieser Art äquivalent in Bezug auf ihre maximale Verspätung sind.

Äquivalenzbeweis

Alle Abläufe ohne Inversionen und Leerlauf haben dieselbe maximale Verspätung.

Beweis.

- Zwei Abläufe ohne Inversionen und Leerlauf können sich nur in der Reihenfolge von Intervallen unterscheiden, die dieselbe Deadline haben.
- Diese Intervalle müssen aufgrund der Inversionsfreiheit in beiden Abläufen in jeweils einem Block hintereinander liegen.
- Da die beiden Blöcke dieselben Intervalle umfassen und es keine Lücken gibt, ist die Länge der Blöcke dieselbe.
- Somit stimmen die Endzeiten der jeweils letzten Intervalle überein.
- Da die maximale Verspätung dieser Intervalle gerade durch diesen Endzeitpunkt bestimmt wird, ist sie in beiden Abläufen gleich. □

Abbildung 5.9 Illustration des Äquivalenzbeweises: (a) Intervalle i, j, und k bilden einen Block von Intervallen, die alle dieselbe Deadline $d_i = d_j = d_k$ haben. Die maximale Verspätung erzeugt das letzte Intervall in dem Block. **(b)** Da das Ende des letzten Intervalles bei allen Vertauschungen gleich ist, ändert sich die maximale Verspätung durch eine solche Vertauschung nicht.

5.6 Strategien für Optimalitätsbeweise von gierigen Algorithmen

Die drei Beispiele in den Abschnitten 5.3, 5.4 und 5.5 waren von der Problemstellung und von dem Ansatz recht ähnlich. Es waren aber in den drei Fällen unterschiedliche Sortierungen, die zu einer optimalen Lösung führten und die Beweise der Optimalität hatten unterschiedliche Strategien:

- Der Optimalitätsbeweis bei der Intervallauswahl folgt der Strategie "der gierige Algorithmus fällt nie zurück" (gegenüber anderen, insbesondere optimalen Verfahren). Dies war in dem folgenden Sinn erfüllt:

- Der Endzeitpunkt des von Frühestes-Ende in Schritt k ausgewählten Intervalls ist nie später als das Ende des Intervalls, das von irgendeinem anderen Algorithmus in Schritt k gewählt wurde.
- Bei anderen Optimierungsproblemen sucht man eine andere Interpretation für das *nicht Zurückfallen*.

- Bei der Intervallverteilung kam ein "Strukturargument" zum Tragen: Man leitet eine Grenze für den Optimalwert her und zeigt, dass der Greedy-Algorithmus diese erreicht.
- In der Ablaufplanung mit minimalen Verzögerungen wurde ein "Austauschargument" benutzt: Eine beliebige optimale Lösung kann durch äquivalentes Ändern der Auswahl in die Lösung des Greedy-Algorithmus umgewandelt werden.

Es folgt die Besprechung eines Greedy-Algorithmus für eine einfache Variante des bekannten Rucksackproblems. Dieser Algorithmus wird später in dem Kapitel Branch-and-Bound benutzt, um einen praktikablen Ansatz für das schwierige 0/1-Rucksackproblem zu erhalten. Auch hier wird eine Art Austauschargument in dem Optimalitätsbeweis verwendet. Er ist allerdings deutlich einfacher, als der Beweis in dem vorigen Kapitel.

5.7 Greedy-Algorithmus für das teilbare Rucksackproblem

Als weiteres Beispiel für einen Greedy-Algorithmus dient das teilbare Rucksackproblem. Es entsteht durch eine Lockerung einer Anforderung aus dem bekannten 0/1-Rucksackproblem, welches NP-vollständig ist.

Rucksackproblem

Es sind n Objekte mit Gewicht w_i und Wert v_i (für $1 \leq i \leq n$) sowie ein Rucksack (*knapsack*) mit einer maximalen Kapazität W gegeben. Wähle Objekte, so dass ihr Gesamtwert maximal ist und ihr Gesamtgewicht die Kapazität nicht überschreitet.

Das Ziel des 0/1-Rucksackproblems kann als diejenige Teilmenge $S \subseteq \{1, \ldots, n\}$ definiert werden, die die folgende Optimierung erfüllt:

$$S \text{ maximiert } \sum_{i \in S} v_i \quad \text{unter der Bedingung } \sum_{i \in S} w_i \leq W$$

Die Auswahl kann auch über einen Vektor $\mathbf{a} \in \{0,1\}^n$ formalisiert werden, wobei $a_i = 1$ anzeigt, dass Objekt i eingepackt wird.
Diese Formulierung kann in der Form generalisiert werden, dass $a_i \in [0,1]$ erlaubt wird, es können also Objekte auch teilweise (mit Anteil a_i) eingepackt werden. Dies beschreibt das teilbare Rucksackproblem, welches in diesem Abschnitt besprochen wird.

Teilbares Rucksackproblem

In derselben Situation wie oben sollen Anteile $a_i \in [0,1]$ (für $1 \leq i \leq n$) so gewählt werden, dass das anteilige Gesamtgewicht die Kapazität nicht überschreitet und der anteilige Gesamtwert maximal ist.

Es kann wie folgt formalisiert werden:

$$\mathbf{a} = (a_1, \ldots, a_n) \in [0,1]^n \text{ maximiert } \sum_{i=1}^{n} a_i v_i \quad \text{unter der Bedingung } \sum_{i=1}^{n} a_i w_i \leq W$$

Als Auswahlpriorität für einen Greedy-Algorithmus könnten *Größter-Wert* oder *Kleinstes-Gewicht* in Betracht gezogen werden. Sie sind aber nicht optimal, wie man an einfachen Gegenbeispielen erkennt.
Für die Ausnutzung des Rucksacks sind Objekte mit möglichst großem relativem Wert (Wert-pro-Gewicht), $d_i = \frac{v_i}{w_i}$, am besten geeignet. Daher geht der Greedy-Algorithmus folgendermaßen vor:

Listing 5.5 Greedy-Algorithmus für das teilbare Rucksackproblem

```
1 sortiere Objekte absteigend nach v_i/w_i
2 r ← W          // Restkapazität
3 for i = 1 to n
4   a_i ←min(1, r / w_i)
5   r ← r − a_i w_i
6 end
```

5.7.1 Beweis der Optimalität des Greedy-Knapsack Algorithmus

Optimalität des Greedy-Knapsack Algorithmus

Der Greedy-Knapsack Algorithmus von Listing 5.5 bestimmt die optimale Lösung des teilbaren Rucksackproblems.

Beweis. Der Lösungsvektor hat die Form $\mathbf{a} = (1, \ldots, 1, a_{k+1}, 0, \ldots, 0)$ für ein $k < n$. Wir zeigen, dass das Verschieben von Anteilen nach hinten die Lösung nicht verbessern kann. Sei $i < k$ und $j \geq k$. Wir reduzieren a_i von 1 auf $\alpha < 1$ und erhöhen a_j entsprechend. Durch die Reduktion von Objekt i sinkt das Gewicht um $(1-\alpha)w_i$. Also kann von Objekt j dieses Gewicht zusätzlich genommen werden, entsprechend einer Erhöhung des Koeffizienten um $\beta = (1-\alpha)w_i/w_j$. Der Gesamtwert ändert sich durch die Verschiebung um

$$\begin{aligned}
\beta v_j - (1-\alpha)v_i &= (1-\alpha)\frac{w_i}{w_j}v_j - (1-\alpha)v_i \\
&= (1-\alpha)\left(\frac{w_i v_j}{w_j} - v_i\right) \\
&= \frac{1-\alpha}{w_j}\left(w_i v_j - w_j v_i\right)
\end{aligned}$$

Diese Änderung ist wegen $\frac{v_i}{w_i} \geq \frac{v_j}{w_j}$ (Sortierung nach relativem Wert) nicht positiv. Die Lösung kann durch Verschieben von Anteilen nach hinten also nicht verbessert werden. Für $i = k+1$ geht die Beweisführung analog. Also ist der Greedy-Algorithmus optimal. Die Idee dieses Beweises ist in Abbildung 5.10 illustriert. □

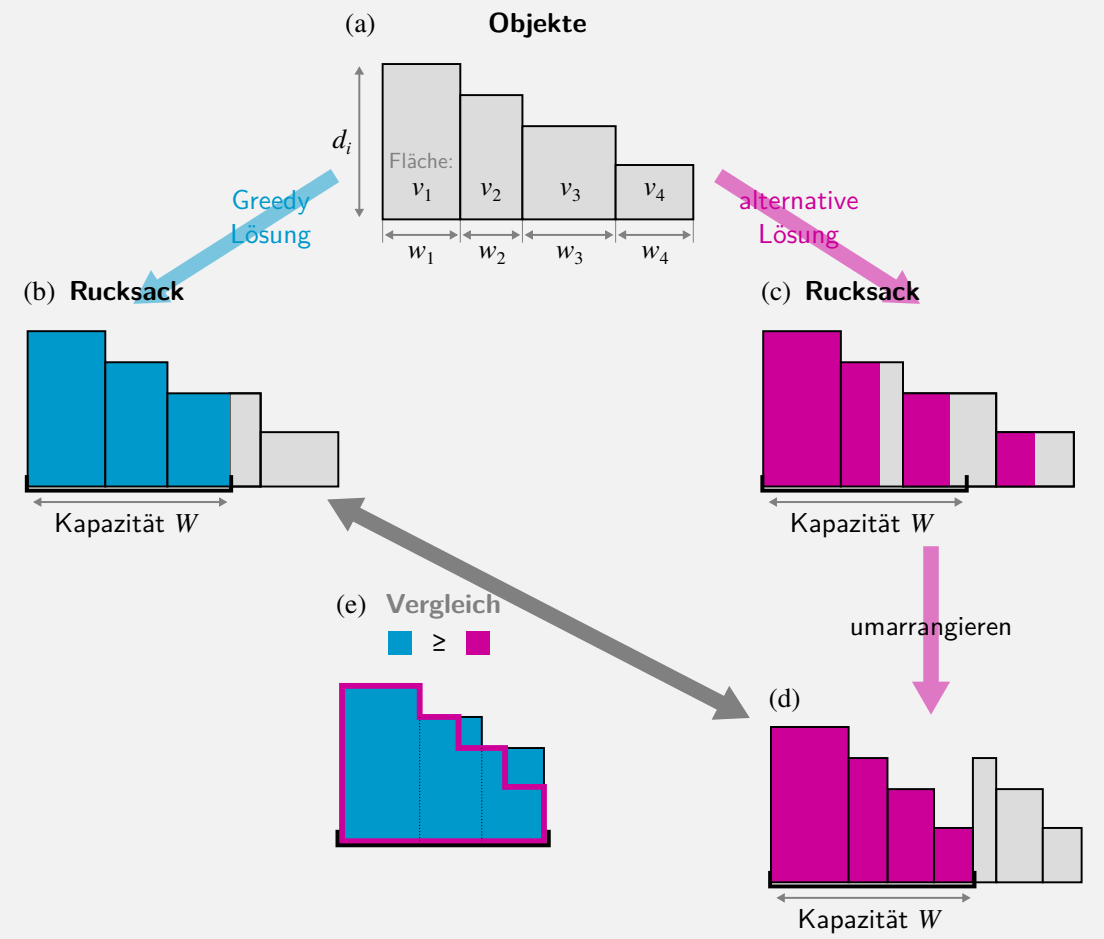
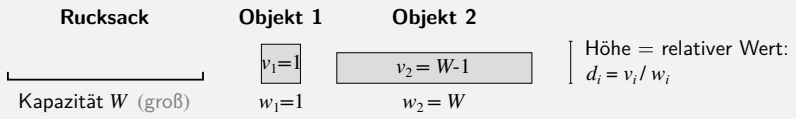


Abbildung 5.10 Illustration des Optimalitätsbeweises an einem Beispiel: (a) Die Objekte sind als Rechtecke dargestellt. Die Breite entspricht ihrem Gewicht und die Höhe dem relativen Wert. Daher ist der Flächeninhalt proportional zu ihrem (absoluten) Wert. Die Objekte sind entsprechend dem Greedy-Knapsack Verfahren nach relativem Wert sortiert. **(b)** Das Greedy Verfahren wählt die beide Objekte mit größerem relativen Wert komplett und das folgende Objekt teilweise aus, so dass die Kapazitätsgrenze ausgeschöpft wird. **(c)** Es wird eine alternative Lösung betrachtet, um zu sehen, ob sie zu einem höheren Wert als die Greedy Lösung führen kann. Diese Lösung wählt Objekte ganz oder teilweise aus. Die ausgewählten Anteile (violett markiert) werden nach links verschoben. **(d)** Auf Grund der Kapazitätsgrenze gehen die ausgewählten Objekte nicht über W hinaus. Daher nehmen die ausgewählten Objekte in den Lösungen (b) und (d) auf der x-Achse dieselbe Strecke ein. **(e)** Der Vergleich der Flächen zeigt, dass die blaue Fläche auf Grund der Sortierung nach der Höhe der Rechtecke nicht kleiner als die violette Fläche sein kann. Also ist der Wert der Greedy-Knapsack Lösung in blau maximal.

5.7.2 Einschränkung des Greedy-Knapsack Algorithmus

Der vorgeschlagene Algorithmus findet die optimale Lösung für das teilbare Rucksackproblem, aber (bis auf in Spezialfällen) nicht für das 0/1-Rucksackproblem. Für das 0/1-Problem kann die Lösung sogar beliebig stark vom Optimum abweichen: Für eine vorgegebene Zahl K kann ein Beispiel mit zwei Objekten konstruiert werden, bei dem der Greedy-Knapsack Algorithmus nur $\frac{1}{K}$-tel des möglichen Wertes erzielt.

Sei K eine (große) Zahl. Wir betrachten einen Rucksack mit Kapazität $W = K + 1$ sowie zwei Objekte mit $v_1 = w_1 = 1$ und $v_2 = W - 1$, $w_2 = W$. Dann sind die relativen Werte $d_1 = 1$ und $d_2 = \frac{W-1}{W} < 1$. Also wählt der Greedy-Algorithmus Objekt 1 mit einem Wert von 1, während die optimale Wahl Objekt 2 mit Wert $W - 1 = K$ ist.
Durch Wahl von K kann ein beliebig großer Verlust (Quotient von optimaler zu Greedy-Lösung) erzeugt werden.



5.8 Weitere Anwendungen von gierigen Algorithmen

Weitere bekannte Anwendungen von Greedy-Algorithmen, die in diesem Skript nicht besprochen werden, sind

- Minimale, aufspannende Wurzelbäume (*minimum-cost arborescence*), siehe z. B. [Kleinberg and Tardos, 2013], S. 219ff
- Clustering, siehe z. B. [Kleinberg and Tardos, 2013], S. 199ff
- Optimales Caching, siehe z. B. [Kleinberg and Tardos, 2013], S. 173ff
- Huffman Code und Komprimierung, siehe z. B. [Kleinberg and Tardos, 2013], S. 203ff

5.9 Das Wichtigste in Kürze

- In besonders glücklichen Fällen gibt es für ein Problem eine lokale Regel, die stets zu einer optimalen Lösung führt. Diese kann von einem **Greedy Algorithmus** verwendet werden.
- Dazu ist es meist wichtig, die Entscheidungen in einer geeigneten Reihenfolge zu bearbeiten. Dies ist bei allen Beispielen in diesem Kapitel der Fall. In

anderen Kapiteln werden wir Beispielen begegnen, bei denen die Reihenfolge der Entscheidungen durch das Problem vorgegeben, also unveränderbar ist, und es trotzdem Greedy Algorithmen gibt.

- Für das teilbare Rucksack-Problem gibt es einen optimalen Greedy-Algorithmus. Für den 0/1 Rucksack kann seine Lösung beliebig schlecht sein.

Quellenverweise

Bei der Darstellung der Gierigen Algorithmen habe ich viele Ideen von den großartigen Folien von Kevin Wayne zu seiner Vorlesung *Theory of Algorithms* (COS 423, Princeton University 2013) aufgenommen. (Seine Vorlesung orientiert sich seinerseits an [Kleinberg and Tardos, 2013].) Der Abschnitt über das teilbare Rucksackproblem hat die Seiten 186f von [Schöning, 2001] als Grundlage. Die Idee zur Illustration des Optimalitätsbeweises ist [Dietzfelbinger, 2012] entnommen.

KAPITEL 6

Minimax Verfahren und Alpha-Beta Suche

6.1 Minimax-Verfahren

In diesem Skript werden viele unterschiedliche algorithmische Konzepte vorgestellt, mit denen Lösungen für Optimierungsprobleme effizient gefunden werden können. Dabei gibt es eine Funktion, die minimiert oder maximiert werden soll. Ein Beispiel dafür wäre die Länge eines Weges von Punkt A nach B (Problem der Kürzesten Pfade) oder der mögliche Informationsfluss durch ein Netzwerk (Maxflow Problem). In diesem Kapitel geht es um Lösungen in einem *kompetitiven* Setting. Dies liegt z. B. in einem Spiel mehrerer Personen vor, die unterschiedliche Ziele verfolgen. Jede Person hat eine eigene Funktion, die sie maximieren möchte, oder jede möchte den eigenen Anteil einer Ressource maximieren.
Wir betrachten Zwei-Personen Nullsummen-Spiele, d. h. der Gewinn der einen Person ist der Verlust der anderen. Sie haben also gegensätzliche Ziele. Wir benutzen die Konvention, dass **A** die Bewertung maximieren und **B** die Bewertung minimieren soll.

Das Minimax-Verfahren

Das Minimax-Verfahren durchläuft den Lösungsraum rekursiv nach *Backtracking* Manier und trifft eine Auswahl mit abwechselnden gegensätzlichen Kriterien. Bei einem Spiel zwischen **A** und **B**: Wähle das Maximum der Bewertungen für Züge von **A** und das Minimum für **B**.
Bei der vollständigen Minimax-Suche werden solange Züge ausgeführt, bis eine Endstellung erreicht ist und die Bewertung feststeht. Bei einer

> unvollständigen Suche wird nach einer gewissen Zugzahl gestoppt und die erreichte Stellung bewertet.

Dieses Grundprinzip liegt vielen Programmen für Strategiespiele zu Grunde. Einige Erweiterungen werden weiter unten in diesem Abschnitt besprochen. Wir illustrieren das Minimax-Verfahren an einem einfachen Spiel: *BestDifference*. Es sei eine Zahlenfolge gegeben, z. B. **2, 8, 3, 5, 4, 1**. A und B ziehen abwechselnd eine der beiden Zahlen, die am Rand (Anfang und Ende der Sequenz) stehen. Sobald nur noch zwei Zahlen übrig sind, wird die Differenz berechnet. Ziel von A ist, dass die Differenz groß und von B, dass sie klein ist.

Eine mögliche Spielfolge für die Sequenz **2, 8, 3, 5, 4, 1** ist die folgende:

A (↑): **2** **3**

Ergebnis: **5 - 4 = 1**

B (↓): **8** **1**

Kann A mehr Punkte erreichen? Gibt es eine Strategie für B, die ein kleineres Ergebnis erzielt? Diese Fragen können beantwortet werden, indem das Minimax-Verfahren auf das Spiel angewendet wird. Dabei werden alle möglichen Kombinationen von Zügen durchlaufen und an jeder Stelle der beste Zug für A bzw. B ausgewählt. Dabei gilt derjenige Zug als der beste, der aus Sicht des/der Ziehenden das beste Endresultat erzielt, unter der Voraussetzung, dass beide optimal spielen. Veranschaulicht wird die Minimax-Suche mit dem Beispiel dieser Zahlenfolge in Abbildung 6.1.

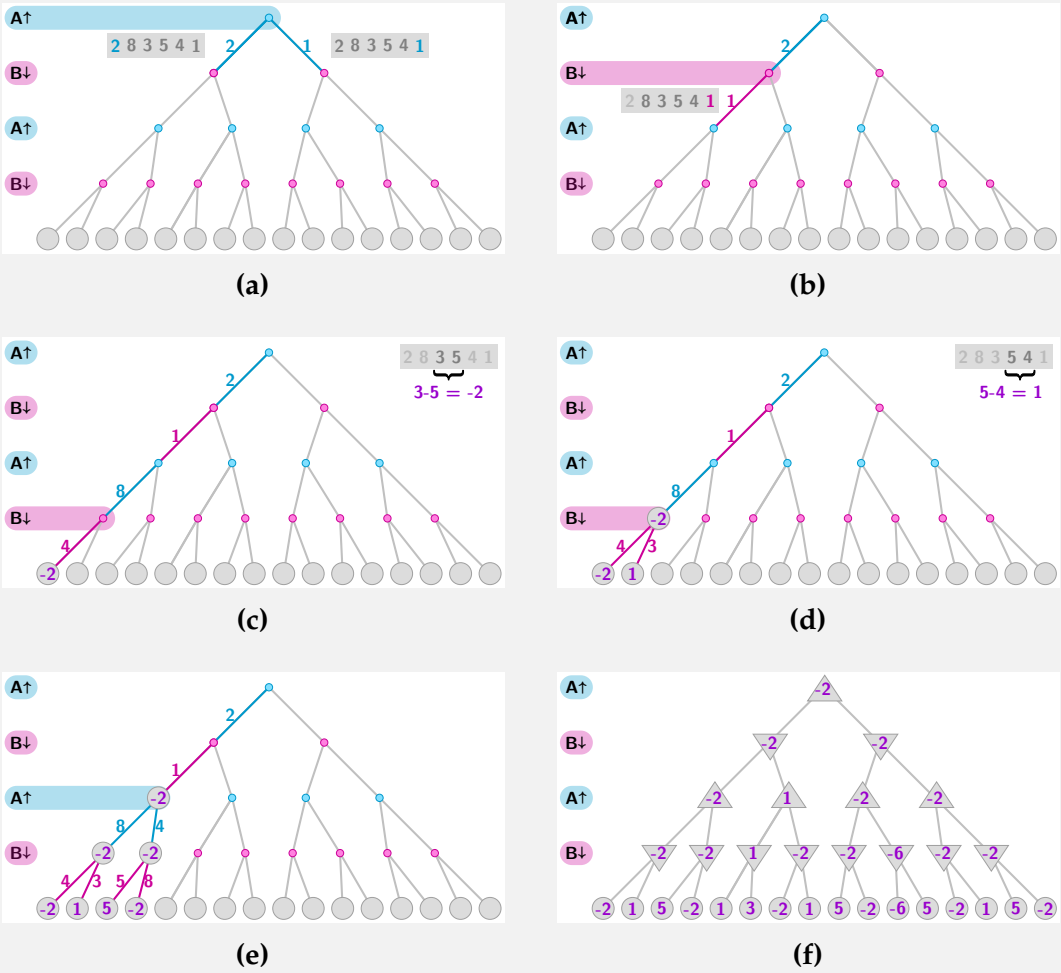


Abbildung 6.1 **Minimax Suchbaum für *BestDifference*:** Gespielt wird in diesem Beispiel mit der Zahlenfolge 2, 8, 3, 5, 4, 1. **(a)** **A** kann die 2 (von der linken Seite) oder die 1 (von der rechten Seite) ziehen. Im Folgenden betrachten wir zunächst immer nur eine Zugmöglichkeit. Aus Darstellungsgründen wählen wir für **A** zunächst die linke und für **B** zuerst die rechte Zugmöglichkeit. **(b)** **B** zieht die 1. **(c)** Nachdem **A** die 8 und **B** die 4 gezogen hat, verbleiben die Zahlen 3 und 5. Das Ergebnis ist die Differenz $3 - 5 = -2$. **(d)** Die anderen Zugmöglichkeiten von **B** führt zu der Differenz 1. Da **B** das Ergebnis *minimieren* will, wäre die erste Möglichkeit besser. Als Bewertung für den hervorgehobenen Knoten wird das Minimum aller Nachfolgerknoten genommen, da **B** an der Reihe ist. In diesem Fall bekommt der Knoten den Wert -2, das Minimum von -2 und 1. **(e)** Nachdem die zweite Zugmöglichkeit für den letzten Zug von **A** auf dieselbe Art durchgespielt wurde, wird nun der Wert für einen **A**-Knoten bestimmt. Da **A** das Resultat maximieren will, wird das Maximum der untergeordneten Knoten gewählt, hier also -2, als Maximum von -2 und -2. **(f)** Auf diese Weise werden alle Zugmöglichkeiten durchlaufen. Nun steht an jedem Knoten, welches Ergebnis die jeweilig ziehende Person in der entsprechenden Situation mindestens (im Sinne der Güte des Ergebnisses) erzielen kann. Die Richtung des Dreiecks verdeutlicht, ob an einem Knoten maximiert oder minimiert wird.

Implementation des *BestDifference* Spiels

Listing 6.1 Implementation des Minimax-Verfahrens für das Spiel *BestDifference*

```
public class Game {
  private int[] values;
  public static final int[] moves = {-1, 1};
  private int first, last;

  Game(int[] values) {
    this.values = values;
    first = 0;
    last = values.length-1;
  }

  public boolean isFinal() {
    return last - first == 1;
  }

  public int score() {
    return values[first] - values[last];
  }

  public void doMove(int move) {
    if (move < 0) {
      first++;
    } else {
      last--;
    }
  }

  public void undoMove(int move) {
    if (move < 0) {
      first--;
    } else {
      last++;
    }
  }
}

public class MinMaxBestDifference {
  Game game;

  MinMaxBestDifference(int[] values) {
    game = new Game(values);
  }

  public int scorePlayerA() {         // Kernfunktionalität von Minimax
```

```
      if (game.isFinal())                // Lösung (Ende) erreicht?
        return game.score();
      int maxScore = Integer.MIN_VALUE;
      for (int move : Game.moves) {      // alle Möglichkeiten durchlaufen
        game.doMove(move);               // Zug ausführen
        int score = scorePlayerB();      // alternierend rekursiver Aufruf
        game.undoMove(move);             // Zug rückgängig machen
        if (score > maxScore)            // Maximum der Bewertungen
          maxScore = score;
      }
      return maxScore;
    }

    public int scorePlayerB() {          // Kernfunktionalität von Minimax
      if (game.isFinal())                // analog zu scorePlayerA()
        return game.score();
      int minScore = Integer.MAX_VALUE;
      for (int move : Game.moves) {
        game.doMove(move);
        int score = scorePlayerA();
        game.undoMove(move);
        if (score < minScore)
          minScore = score;
      }
      return minScore;
    }

    public static void main(String[] args) {
      int[] values = {2, 8, 3, 5, 4, 1};
      MinMaxBestDifference bd = new MinMaxBestDifference(values);
      int bestScoreA = bd.scorePlayerA());
    }
  }
```

Das Minimax-Suchverfahren im Kontrast zu *Backtracking*

Wie die Implementation zeigt, ist das Minimax-Suchverfahren sehr ähnlich zum *Backtracking*. Als zusätzliches Grundelement kommt eine Bewertungsfunktion hinzu, hier `game.score()`. An Stelle des einfach rekursiven Aufrufs beim *Backtracking* steht beim Minimax der rekursiv alternierende Aufruf der gegenläufigen Auswahlverfahren. Die Minimax Methoden geben die Bewertung der jeweiligen Spielposition zurück, während es beim *Backtracking* die Lösbarkeit ist.

Erweiterungen des Minimax-Suchverfahrens

Bei komplexeren Spielen, wie Schach, können (außer in der letzten Phase des Endspiels) die Zugkombinationen nicht bis zum Spielende durchgerechnet wer-

den. Daher wird Minimax nur bis zu einer bestimmten Zugtiefe durchgeführt (unvollständige Minimax-Suche). Dann wird die Güte der erreichten Spielposition bewertet. Häufig wird dabei mit einer dynamischen Zugtiefe gearbeitet. Bei der Ruhesuche (*Quiescence search*) wird z. B. eine Zugfolge über die vorgegebene Zugtiefe hinaus weiterfolgt, solange die Bewertung nach jedem Zug stark schwankt. Dies ist besonders bei Zugkombinationen wichtig, bei denen abwechselnd Figuren geschlagen werden. Hier würde die Ruhesuche die Zugkombinationen so lange durchführen, bis der Schlagabtausch beendet ist und eine realistische Bewertung möglich ist.
Allerdings werden bei Minimax viele überflüssige Auswertungen gemacht. Diese Beobachtung führt zu dem deutlich effizienteren Verfahren der Alpha-Beta-Suche, das im nächsten Abschnitt vorgestellt wird.

6.2 Motivation der Alpha-Beta-Suche

Die **Alpha-Beta-Suche** (*alpha-beta pruning*) basiert auf der einfachen Überlegung, dass in bestimmten Situationen gewisse Züge des Gegenspielers überhaupt nicht in Erwägung gezogen werden müssen. Ein Beispiel: Spieler **A** hat für eine Zugmöglichkeit alle Erwiderungen von **B** durchdacht und kommt auf eine Bewertung von 3. Bei dem nächsten Zug, den **A** durchdenkt, stößt er direkt auf eine gute Erwiderung von **B** mit einer Bewertung von -5. In dieser Situation braucht **A** gar keine anderen Zugmöglichkeiten von **B** zu durchdenken. Auf Grund der *einen* guten Erwiderungsmöglichkeit, sollte **A** die zweite Zugmöglichkeit sowieso verwerfen. Bei diesen Überlegungen wird immer davon ausgegangen, dass beide Spieler optimal spielen.

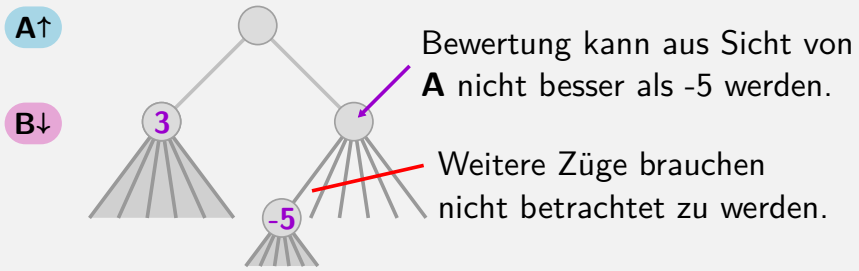


Abbildung 6.2 Mit der ersten Zugmöglichkeit erzielt **A** den Wert 3. Für die zweite Zugmöglichkeit ergibt schon der erste Gegenzug von **B** die (für **A**) schlechte Bewertung von -5. Daher braucht **A** die anderen Zugmöglichkeiten von **B** nicht zu durchdenken, da dies höchstens dazu führen kann, dass eine noch bessere Erwiderung von **B** entdeckt wird. **A** erzielt aber mit der ersten Zugmöglichkeit (mindestens) den besseren Wert 3, wird also nicht die zweite Zugmöglichkeit wählen, die bei einer geschickten Antwort von **B** nicht besser als der Wert -5 werden kann.

Dieses Beispiel soll nun zu einem allgemeinen Prinzip generalisiert werden. Dafür werden zwei Schranken, α und β, verwendet, die für die Alpha-Beta-Suche namensgebend sind.
In dem obigen Beispiel stellt die Bewertung 3 eine untere Schranke in dem folgenden Sinn dar: **A** hat schon eine Zugmöglichkeit durchdacht, die mindestens die Bewertung 3 sichert. (Es kann besser werden, falls **B** nicht optimal spielt.) Züge von **A**, für die es auch nur *eine* Erwiderung von **B** gibt, die zu einer Bewertung *unterhalb der Grenze* führt, können verworfen werden.
Für diese untere Schranke wird die Variable α verwendet und für die obere Schranke in den analogen Überlegungen aus Sicht von **B** die Variable β.

Die Alpha-Beta Suche

Bei der Alpha-Beta-Suche wird zusätzlich zum Minimax-Verfahren bei jedem Knoten ein Intervall $[\alpha\ \beta]$ gespeichert, welches während der Suche aktualisiert wird. Es bedeutet:

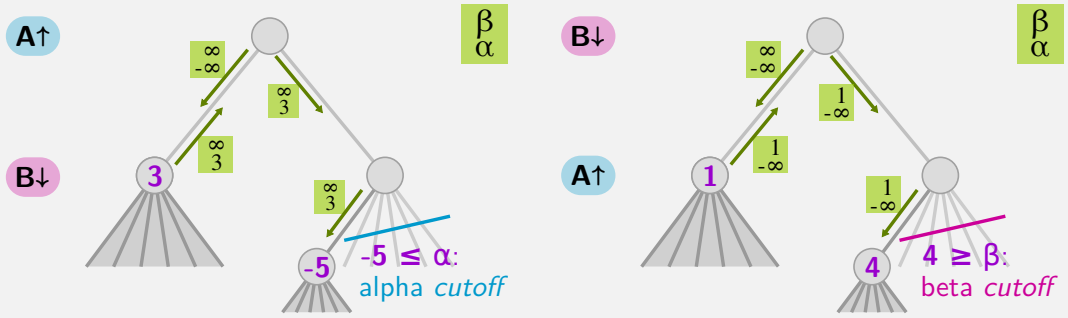
- Für **A** wurde eine Zugvariante gefunden, die zu einer Bewertung $\geq \alpha$ führt.
- Für **B** wurde eine Zugvariante gefunden, die zu einer Bewertung $\leq \beta$ führt.

Dies wird folgendermaßen erreicht: $[\alpha\ \ \beta]$ wird am Wurzelknoten mit $[-\inf\ \inf]$ initialisiert. Das $[\alpha\ \beta]$-Intervall wird an untergeordnete Knoten weitergegeben. Wird bei einem **A**-Knoten eine Zugmöglichkeit (für **A**) mit einer Bewertung $> \alpha$ gefunden, wird α angepasst. Wird bei einem **B**-Knoten eine Zugmöglichkeit (für **B**) mit einer Bewertung $< \beta$ gefunden, wird β angepasst.
Auf diese Weise kann die Suche begrenzt werden:

- Wenn an einem **A**-Knoten eine Zugmöglichkeit (für **A**) mit Bewertung $\geq \beta$ entdeckt wird, brauchen keine weiteren Zugmöglichkeiten an diesem Knoten in Betracht gezogen zu werden (*beta-cutoff*).
- Wenn an einem **B**-Knoten eine Zugmöglichkeit (für **B**) mit Bewertung $\leq \alpha$ entdeckt wird, brauchen keine weiteren Zugmöglichkeiten an diesem Knoten in Betracht gezogen zu werden (*alpha-cutoff*).

Die Abbruchkriterien alpha- und beta-cutoff realisieren genau die Überlegung, die oben in einem speziellen Beispiel diskutiert und in Abbildung 6.2 illustriert wurde: **A** geht Zugmöglichkeiten durch, erwägt Erwiderungen von **B** und sucht das Maximum. Falls dabei eine Zugmöglichkeit von **B** mit $> \beta$ bewertet wurde, brauchen keine weiteren Züge an diesem Knoten berücksichtigt zu werden: **B** würde es nicht zu dieser Spielsituation kommen lassen, denn die β Schranke

zeigt, dass es für **B** eine günstigere Variante gibt. Bei Gleichheit kann auch abgebrochen werden, da eine gleichwertige Variante schon gefunden wurde. (Nur wenn *alle* Zugfolgen, bei denen beide Spieler optimal spielen, gefunden werden sollen, wird bei Gleichheit nicht abgebrochen.) Diese Gedanken zum alpha- und beta-Abbruch werden in Abbildung 6.3 dargestellt.



(a) Für die abgeschnittenen Äste gilt: mit Bewertung ≥ -5 würde **B** sie nicht wählen; mit Bewertung < 3 würde **A** nicht in diesen Zweig gehen

(b) Für die abgeschnittenen Äste gilt: mit Bewertung ≤ 4 würde **A** sie nicht wählen; mit Bewertung > 1 würde **B** nicht in diesen Zweig gehen.

Abbildung 6.3 Illustration zum Alpha-Beta Intervall

Die Alpha-Beta-Suche bietet also einen Vorteil gegenüber dem Minimax Verfahren, da Zugmöglichkeiten, die für das Ergebnis bei optimaler Spielweise irrelevant sind, nicht berücksichtigt werden. Für die Beispielfolge des *BestDifference* Spiels ist der (beschnittene) Baum der Alpha-Beta-Suche in Abbildung 6.4 dargestellt.

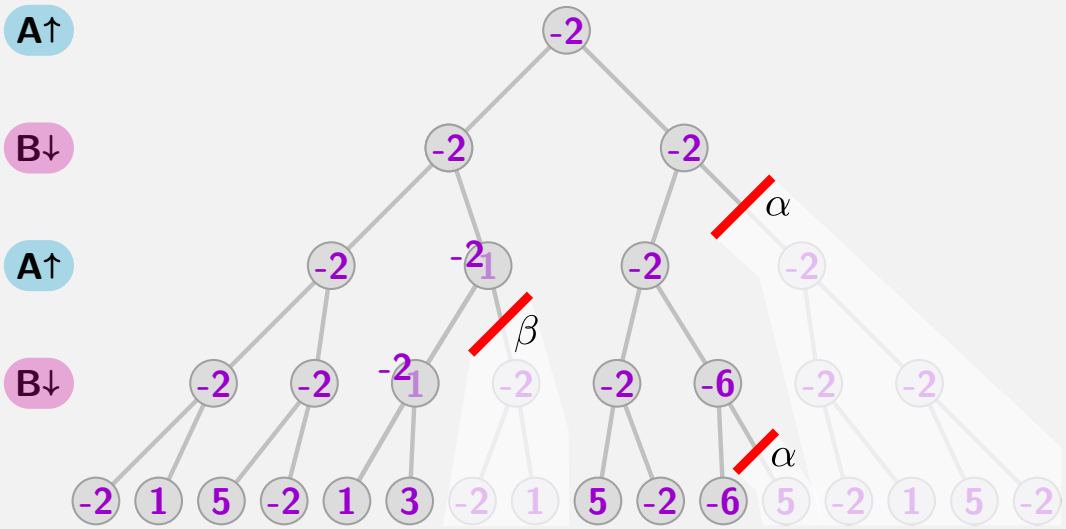


Abbildung 6.4 **Baum der Alpha-Beta-Suche** für die gleiche Beispielsequenz des *BestDifference* Spiels, die bei dem Minimax-Verfahren verwendet wurde. Die Bewertungen der Endstellungen sind Abbildung 6.1 entnommen. Bei der Alpha-Beta-Suche wären die ausgegrauten Zweige nicht berechnet worden. Der Wert an einem Knoten gibt hier nicht unbedingt das für den ziehenden Spieler *von der jeweiligen Situation* bestmögliche Ergebnis wieder. Hier hängt der Wert auch von den zuvor gefundenen Zugvarianten ab. Der Wert kann auch bedeuten, dass das entsprechende Ergebnis von dem Spieler durch eine früher gefundene Zugstrategie erreicht werden kann.

Implementierung der Alpha-Beta-Suche für das *BestDifference* Spiel

Listing 6.2 Implementation der Alpha-Beta-Suche für das Spiel *BestDifference*

```
public int scorePlayerA(int alpha, int beta) {
  if (game.isFinal()) {
    return game.score();
  }
  for (int move : Game.moves) {
    game.doMove(move);
    int score = scorePlayerB(alpha, beta);
    game.undoMove(move);
    if (score > alpha) {
      alpha = score;
      if (alpha >= beta) break;
    }
  }
  return alpha;
}

public int scorePlayerB(int alpha, int beta) {
  if (game.isFinal()) {
    return game.score();
  }
  for (int move : Game.moves) {
    game.doMove(move);
    int score = scorePlayerA(alpha, beta);
    game.undoMove(move);
    if (score < beta) {
      beta = score;
      if (beta <= alpha) break;
    }
  }
  return beta;
}
```

6.2.1 Negamax Variante der Alpha-Beta Implementierung

Der Code der Methoden `scorePlayerA()` und `scorePlayerB()` ist fast gleich. Die Methoden rufen sich wechselseitig auf. Die Methode für **A** maximiert und verändert α, während die Methode für **B** minimiert und β aktualisiert. Diese Beobachtung legt es nahe, dass die Implementation verkürzt werden kann, indem für beide Spieler eine einheitliche Methode benutzt wird.
Genau dies leistet die **Negamax-Variante**. Dabei wird die Bewertung so verändert, dass beide Spieler maximieren, jeweils aus ihrer Sicht. Dies muss natürlich

bei der Bewertungsfunktion berücksichtigt werden. In dem Code Beispiel 6.3 wird dies durch die Multiplikation von `game.score()` mit der Variable `player` erreicht, wobei `player` die Werte `1` und `-1` annimmt. Die negamax-Methode (`scorePlayer()` in Code Beispiel 6.3) ruft sich selbst auf und führt dabei eine Negation der Spielbewertung durch und vertauscht die Rollen von α und β.

Listing 6.3 Negamax Implementation des Spiels *BestDifference*

```
1  public int alphaBetaNegamax() {
2    return scorePlayer(1, -Integer.MAX_VALUE, Integer.MAX_VALUE);
3  }
4
5  public int scorePlayer(int player, int alpha, int beta) {
6    if (game.isFinal()) {
7      return player * game.score();
8    }
9    for (int move : Game.moves) {
10     game.doMove(move);
11     int score = - scorePlayer(-player, -beta, -alpha);
12     game.undoMove(move);
13     if (score > alpha) {
14       alpha = score;
15       if (alpha >= beta) break;
16     }
17   }
18   return alpha;
19 }
```

Preisfrage: Warum `-Integer.MAX_VALUE` an Stelle von `Integer.MIN_VALUE` in Zeile 3?

Bemerkungen und Erweiterungen der Alpha-Beta-Suche

Die Alpha-Beta-Suche kann eine erhebliche Einschränkung des Suchraums bewirken. Wie groß die Einsparung ist, hängt stark von der Reihenfolge ab, in der die Zugmöglichkeiten behandelt werden. Im ungünstigsten Fall werden die Züge in einer Reihenfolge durchlaufen, in der überhaupt keine alpha- und beta-Abbrüche zustande kommen. Dann ist der Ablauf der Alpha-Beta-Suche exakt derselbe wie bei dem Minimax-Verfahren. Falls es h Spielzüge gibt (= Höhe des Suchbaums) und bei jedem Zug k-viele Zugmöglichkeiten, ist die Laufzeit in $\Theta(k^h)$. Je früher ein günstiger Zug gefunden wird, desto effektiver kann beschnitten werden. Im besten Fall werden von einem Spieler alle k Möglichkeiten und von dem anderen immer nur eine Möglichkeit betrachtet. Dann ist die Laufzeit in $\Theta(k^{h/2})$.

Daher ist es eine gute Strategie, vielversprechende Spielzüge innerhalb der Alpha-Beta-Suche zuerst auszuwählen. Dies kann mit einer sogenannten Killer-Heuristik bewerkstelligt werden. Diese Übertragung von Spielzügen ist allerdings nur bei manchen Spielen anwendbar.
Wie bei Minimax, wird auch bei der Alpha-Beta-Suche oft mit einer dynamischen Zugtiefe gearbeitet (Ruhesuche, *Quiescence search*).
Bemerkung: Für das Beispiel *BestDifference* kann mit Dynamischer Programmierung die Lösung deutlich effizienter gefunden werden.

KAPITEL 7

Branch-and-Bound

Der ***Branch-and-Bound*** Ansatz generiert, wie *Backtracking*, Teillösungen in einer Baumstruktur. Das Ziel ist es, eine optimale Lösung in einem *Optimierungsproblem* zu finden. Bei der folgenden Darstellung gehen wir von einem Maximierungsproblem aus. Minimierungsprobleme können mit entsprechenden Anpassungen analog behandelt werden. Es gibt also eine Funktion, die jeder (vollständigen) Lösung einen Wert zuordnet. Eine **optimale Lösung** ist eine Lösung, die den maximalen Wert erzielt. Wie zuvor bezeichnen wir mit Teillösung einen unvollständigen Zwischenzustand bei der schrittweisen Konstruktion einer Lösung. Als Spezialfall ist auch eine vollständige Lösung eine Teillösung.

7.1 Das Konzept von *Branch-and-Bound*

Zusätzlich zur *Backtracking* Strategie stoppt *Branch-and-Bound* die Suche bei Teillösungen, die zu keiner optimalen Lösung führen können. Um dies zu entscheiden, benötigt man die Information, wie gut sich eine Teillösung bestenfalls entwickeln kann. Wenn dies nicht besser ist, als die beste schon bekannte Lösung, braucht man die Teillösung nicht weiterzuverfolgen.
Wesentlich (und namensgebend) für den Ansatz sind also zwei Schranken (*bounds*). Eine gefundene Lösung stellt eine untere Schranke (*lower bound*) für die weitere Suche dar. Es werden nur Teillösungen weiterverfolgt, die zu einem besseren Ergebnis führen können als diese untere Schranke. Um dies zu beurteilen, wird für jede Teillösung eine obere Schranke (*upper bound*) benötigt, die das maximale Potenzial angibt. Je enger die obere Grenze geschätzt wird, desto effizienter kann *Branch-and-Bound* sein, denn dadurch kann der Suchraum stärker eingeschränkt werden.

Branch-and-Bound und die Schranken

Für das *Branch-and-Bound* Verfahren benötigt man eine Methode, die zu jeder Teillösungen eine obere Schranke (***upper bound***) bestimmt. Diese Schranke besagt, dass keine Lösung, die eine Fortsetzung dieser Teillösung ist, einen größeren Wert erzielen kann. Der *Upper Bound* ist also eine obere Schranke für das Potenzial.
Sobald irgendeine Lösung mit Wert c_0 gefunden wurde, können alle Teillösungen mit einem *Upper Bound* kleiner gleich c_0 verworfen werden. Sie können nicht zu besseren Lösungen führen. Der Wert einer gefundenen Lösung stellt somit eine untere Schranke (***lower bound***) dar.
Es werden beim Branch-and-Bound nur solche Teillösungen weiterverfolgt, deren *upper bound* über dem aktuellen *lower bound* liegt. Mit jeder gefundenen Lösung wird der *lower bound* angepasst.

Branch-and-Bound mit Initiallösung

Der *Branch-and-Bound* Mechanismus greift also erst, nachdem die erste Lösung gefunden wurde. Bis dahin läuft alles wie beim Backtracking. Da es bei manchen Problemen recht lange dauern kann, bis die Tiefensuche des Backtrackings die erste Lösung liefert, kann dies den Vorteil von *Branch-and-Bound* einschränken. Daher ist es vorteilhaft, wenn man schnell eine (meist suboptimale) Lösung generieren kann, z. B. mit einem Greedy-Algorithmus. Dies ist nicht bei allen Problemen möglich, aber wenn, dann kann man es als Schritt vor dem eigentlichen *Branch-and-Bound* ausführen und den Wert dieser Initiallösung als *Lower Bound* nutzen.
Wichtig: Der *Branch-and-Bound* Ablauf fängt immer mit der leeren Teillösung an. Die Initiallösung wird nur für den Startwert des *Lower Bound* verwendet.
Demnach kann das *Branch-and-Bound* Verfahren in dem folgenden Pseudocode zusammengefasst werden:

```
1  Bestimme eine Initiallösung
2  Setze Lower Bound auf den Wert dieser Lösung (oder –Inf)
3  Rekursion
4    Falls Teillösung eine Lösung darstellt:
5      Falls Lösung den bisher höchsten Wert hat: speichern
6      Lösungswert ist neuer Lower Bound
7    Generiere Kandidaten für nächsten Schritt in aktueller Teillösung
8    Für jeden Kandidaten:
9      Führe Schritt aus
10     Berechne Upper Bound der erweiterten Teillösung
11     Falls dieser über dem Lower Bound liegt
```

```
12        Gehe in die Rekursion für den nächsten Schritt
13      Mache Schritt rückgängig
```

Zusätzlich zu den Grundbausteinen des *Backtracking* benötigt *Branch-and-Bound* die folgenden Elemente:

1. ***Branching:*** Verzweigung im Baum der Teillösungen, als Teil davon:
   - Reihenfolge der Entscheidungen beim *branching*
2. ***Bounding:*** Bestimmen von oberen Schranken für die aktuelle Teillösung: Wie gut können resultierende Lösungen bestenfalls sein.
3. ***Lower Bound:*** Wird mit `-Inf` oder dem Wert einer Initiallösung initialisiert, und während des Ablaufs aktualisiert, wenn eine Lösung mit höherem Wert gefunden wurde.

Das *branching* ist auch Teil des *Backtracking*, aber beim *Branch-and-Bound* kommt dem Punkt eine größere Bedeutung zu. Die Reihenfolge, in der die Auswahlmöglichkeiten getestet werden, ist entscheidend für die Effektivität der *Bounds*. Um einen *Branch-and-Bound* Ansatz für ein konkretes Problem implementieren zu können, müssen die folgenden Fragen beantwortet werden:

- ***Branching:*** Welche Entscheidung wird zur Verzweigung im Baum der Teillösungen verwendet?
  - Typische Entscheidungen sind "Objekt X wird ausgewählt oder nicht".
- **Auswahlreihenfolge:** In welcher Reihenfolge werden diese Entscheidungen getroffen?
  - Diese sollte so gewählt werden, dass die Wahrscheinlichkeit höher ist, früh gute Lösungen zu finden.
- ***Bounding:*** Wie werden die Schranken für die Lösungswerte (*Bounds*) bestimmt?
  - *Bounds* kann man häufig dadurch erhalten, dass man Lösungen unter vereinfachten Bedingungen betrachtet.
- **Initiallösung:** Wie kann schnell (meist Greedy) ein korrekte, wenngleich suboptimale Lösung generiert werden?

7.2 Beispiel: *Branch-and-Bound* für das 0/1-Rucksackproblem

Das 0/1-Rucksackproblem ist NP-vollständig, siehe hier. Daher gibt es keine realistische Hoffnung, ein Verfahren zu finden, das für alle Eingaben effizient optimale Lösungen bestimmt. Hier entwickeln wir einen *Branch-and-Bound* Ansatz, der bei günstiger Ausgangslage viele Teile des Suchbaumes als irrelevant erkennt und ignoriert. So gibt es zumindest die Hoffnung, dass der Ansatz viele konkrete Probleme in akzeptabler Zeit löst.
Hier werden die Entscheidungen bezüglich der Grundelemente von *Branch-and-Bound* gemäß der Leitfragen am Ende des letzten Abschnittes festgelegt:

- Der Baum der Teillösungen wird so konstruiert, dass bei jedem Knoten in der k-ten Ebene die Entscheidung getroffen wird, ob der k-te Gegenstand ausgewählt wird.
- Der *Upper Bound* einer Teillösung wird als Greedy Lösung des teilbaren Rucksackproblems für die verbleibenden Gegenstände bestimmt.
  - Genauer: Wurden in der Teillösung schon Entscheidungen für die ersten k Gegenstände getroffen, so wird das teilbare Rucksackproblem für die Gegenstände ab $k+1$ betrachtet. Dabei wird als Kapazität die Restkapazität der Teillösung gesetzt.
- Damit der Greedy-Algorithmus sinnvoll funktioniert, müssen die Gegenstände also im *Branch-and-Bound* Verfahren absteigend nach ihrem relativen Wert sortiert werden.
- Als Initiallösung für den *Lower Bound* nehmen wir die Greedy-Lösung, wobei der geteilte Gegenstand weggelassen wird.

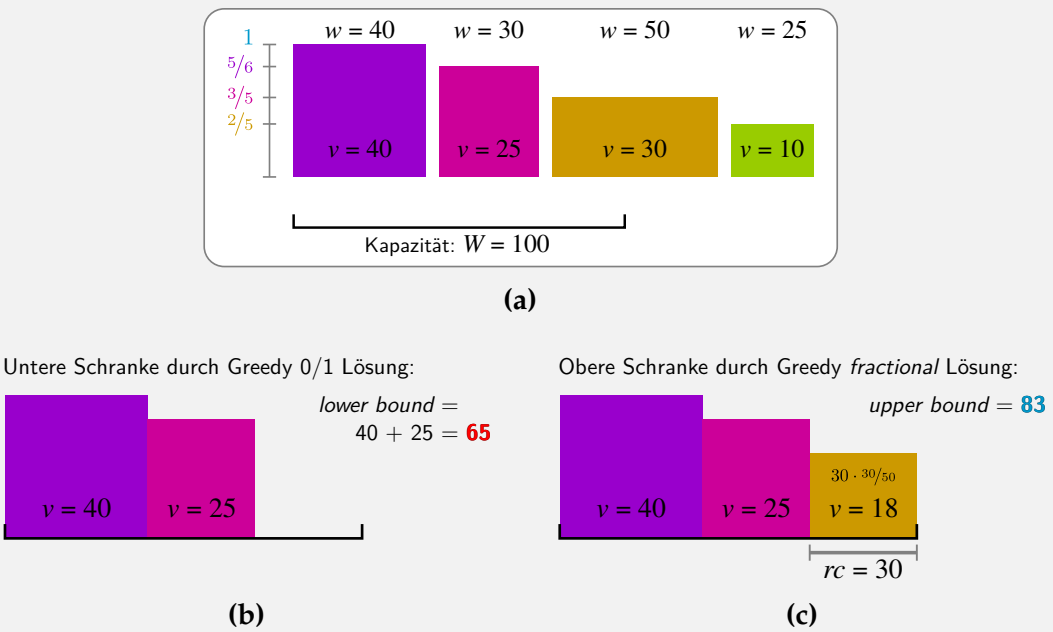


Abbildung 7.1 **Bestimmung der Bounds für das 0/1 Rucksackproblem:** Die Bestimmung der Schranken wird an dem Beispiel Rucksack **(a)** mit vier Objekten demonstriert. Das Gewicht eines Objektes wird durch seine Breite, der relative Wert durch seine Höhe und der absolute Wert durch seine Fläche dargestellt. Wichtig ist, dass die Objekte nach ihrem relativen Wert absteigend sortiert sind. **(b)** Die untere Schranke wird durch die folgende Initiallösung bestimmt: Objekte werden der Reihe nach ausgewählt, bis das erste nicht mehr passt (Kapazitätsüberschreitung). Das Gewicht der eingepackten Objekte ist die untere Schranke. **(c)** Die obere Schranke kann für jede Teillösung bestimmt werden. Hier wird es für die leere Teillösung (noch kein Objekt ausgewählt) exemplarisch gezeigt. Die noch vorhandenen Objekte werden der Reihe nach ausgewählt. Wenn das erste nicht mehr passt, wird es anteilig ausgewählt. Im dargestellten Fall ist bei der Auswahl des dritten Objektes die Restkapazität $100 - 40 - 30 = 30$. Das dritte Objekt mit Gewicht 50 passt nur noch zu dem Anteil $30/50$. Der anteilige Wert beträgt $30 \cdot 30/50 = 18$. Somit ist das Gesamtgewicht der eingepackten Objekte $40 + 25 + 18 = 83$. Dies ist die obere Schranke der leeren Teillösung.

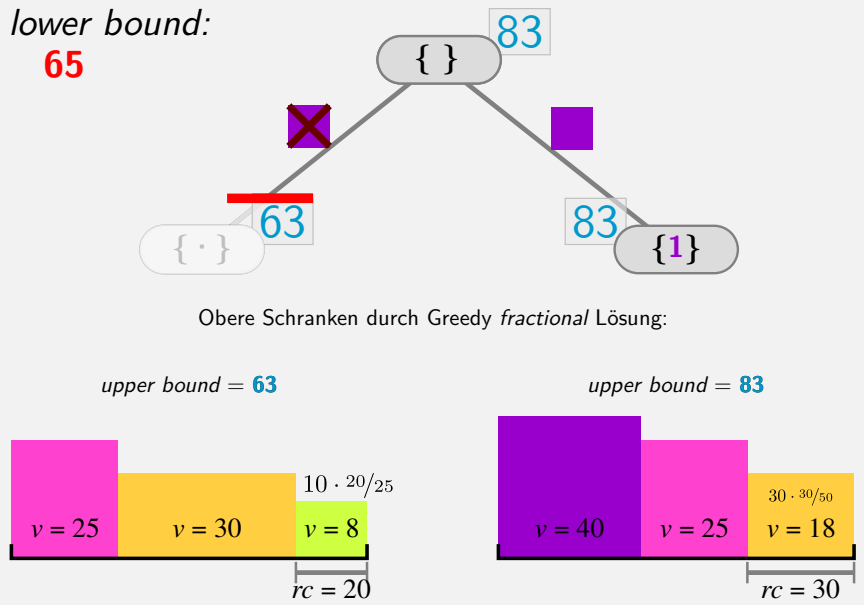


Abbildung 7.2 Der erste Schritt im *Branch-and-Bound* für das 0/1-Rucksackproblem: Es wird verzweigt gemäß der Entscheidung, ob das erste Objekt (blauviolett) eingepackt wird (rechter Zweig) oder nicht (linker Zweig). Falls es nicht eingepackt wird, erhält man eine obere Schranke von 63. Eine bessere Lösung kann in diesem Zweig also nicht gefunden werden, egal wie die Entscheidung für die restlichen Objekte getroffen wird. Da dieser Wert unter der unteren Schranke von 65 liegt, wird dieser Zweig des Suchbaums nicht weiterverfolgt. Wenn das erste Objekt ausgewählt wird (rechter Zweig), erhält man die obere Schranke 83. Hier wird also weitergesucht.

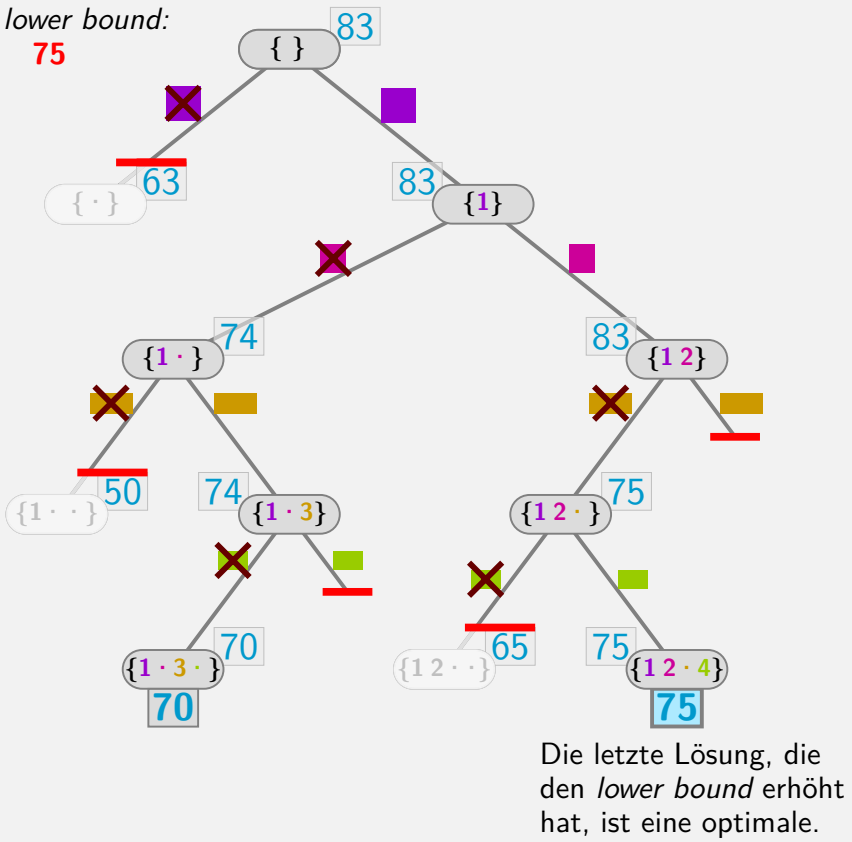


Abbildung 7.3 Vollständiger *Branch-and-Bound* Suchbaum für das 0/1-Rucksackproblem: An jedem Knoten im Suchbaum wird vorgegangen, wie in Abbildung 7.2 erläutert. Ein spannender Moment ist, wenn unten links die erste vollständige Lösung erreicht wird. Objekte **1** und **3** wurden eingepackt. Dies ergibt den Wert 70, also wurde zu dem Zeitpunkt der *lower bound* von 65 auf 70 erhöht. Später wird eine Lösung mit der 75 gefunden. Die Suche geht weiter, allerdings ist der danach noch zu prüfende Fall nicht zulässig. Bei der Teillösung mit Objekten **1** und **2** kann Objekt **3** nicht ausgewählt werden, da dies zu einer Kapazitätsüberschreitung führen würde (abgeschnittener Ast rechts auf mittlerer Höhe). Nun wird diejenige Lösung ausgewählt, die als letztes den *lower bound* erhöht hat.

Listing 7.1 *Branch-and-Bound* Implementation für das 0/1-Rucksackproblem

```
public class Item {
  public double value;
  public double weight;

  Item(double value, double weight) {
    this.value = value;
    this.weight = weight;
  }
}

// Comparator for sorting items suitable for the Greedy algorithm
public class ItemCompareRelativeValue implements Comparator<Item> {
  @Override public int compare(Item item1, Item item2)
  {
    return Double.compare(item1.value/item1.weight,
                          item2.value/item2.weight);
  }
}

public class Knapsack {
  private LinkedList<Item> inventory;
  protected double capacity;
  protected double residualCapacity;

  public Knapsack(double capacity) {
    this.inventory = new LinkedList<>();
    this.capacity = capacity;
    this.residualCapacity = capacity;
  }

  // copy constructor (omitted here)
  public Knapsack(Knapsack that) {...}

  public double value() {
    double knapsackValue = 0.0;
    for (Item item : inventory) {
      knapsackValue += item.value;
    }
    return knapsackValue;
  }

  public double weight() {
    double knapsackWeight = 0.0;
    for (Item item : inventory) {
      knapsackWeight += item.weight;
    }
    return knapsackWeight;
```

```
    }

    public void addItem(Item item) {
      inventory.push(item);
      residualCapacity -= item.weight;
    }

    public void removeItem() {
      Item item = inventory.pop();
      residualCapacity += item.weight;
    }
  }

  public class BranchAndBoundKnapsack {
    private Item[] items;
    private double capacity;
    Knapsack knapsack;
    Knapsack optKnapsack;

    public BranchAndBoundKnapsack(Item[] items, double capacity) {
      this.items = items;  // Side effect: changing input object!
      Arrays.sort(items, Collections.reverseOrder(new ItemCompareRelativeValue()));
      this.capacity = capacity;
    }

    public Knapsack initialSolution() {
      Knapsack initialKnapsack = new Knapsack(capacity);
      for (Item item : items) {
        if (item.weight < initialKnapsack.residualCapacity) {
          initialKnapsack.addItem(item);
        }
      }
      return initialKnapsack;
    }

    public double upperBound(Knapsack knapsack, int startIndex) {
      double residualCapacity = knapsack.residualCapacity;
      double addedValue = 0;
      for (int k = startIndex; k < items.length; k++) {
        double weight = items[k].weight;
        if (weight <= residualCapacity) {
          addedValue += items[k].value;
          residualCapacity -= weight;
        } else {
          addedValue += items[k].value * residualCapacity / weight;
          break;
        }
      }
      return knapsack.value() + addedValue;
    }
```

```
  public double branchAndBound() {
    Knapsack initKnapsack = initialSolution();
    optKnapsack = initKnapsack;
    knapsack = new Knapsack(capacity);
    return pack(initKnapsack.value(), 0);
  }

  public double pack(double lowerBound, int level) {
    if (level == items.length) {
      if (knapsack.value() > optKnapsack.value()) {
        optKnapsack = new Knapsack(knapsack);
// copy constructor!!
      }
      return knapsack.value();
    }
    if (upperBound(knapsack, level + 1) > lowerBound) {  // exclude item
      double value = pack(lowerBound, level + 1);
      if (value > lowerBound) lowerBound = value;
    }
    Item newItem = items[level];                          // include item
    if (newItem.weight <= knapsack.residualCapacity) {   // if it fits
      knapsack.addItem(newItem);
      if (upperBound(knapsack, level + 1) > lowerBound) {
        double value = pack(lowerBound, level + 1);
        if (value > lowerBound) lowerBound = value;
      }
      knapsack.removeItem();
    }
    return lowerBound;
  }
```

7.3 *Beyond Branch-and-Bound*

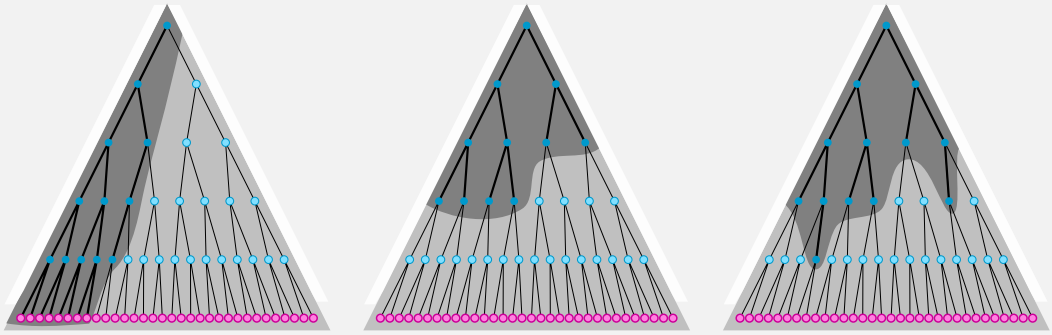
Entscheidend für die Effektivität der *Bounds* ist es, dass frühzeitig möglichst gute Lösungen gefunden werden. Erst dann können die *Bounds* greifen. Der *upper bound* einer Teillösung gibt nur eine Beschränkung nach oben. Bei sinnvollen *Bounds* gilt meist auch:

> Je höher der *Bound* einer Teillösung ist, desto besser ist die Chance, eine gute Lösung als Fortsetzung dieser Teillösung zu finden.

Dies motiviert die Strategie von der starren Suchreihenfolge der Tiefensuche Abstand zu nehmen, und immer diejenige Teillösung zuerst weiterzuverfolgen, die unter allen bekannten den größten *Bound* besitzt. Bei dem Suchbaum in

Abbildung 7.3 wäre mit diesem Ansatz bei der Verzweigung für das zweite Objekt zunächst der rechte Zweig weiterfolgt worden (*upper bound* 83 > 74). Auf diese Weise wäre die Lösung mit Wert 75 zuerst gefunden und der *lower bound* entsprechend erhöht worden. Das hätte zur Folge gehabt, dass dann der linke Zweig für das zweite Objekt abgeschnitten worden wäre (74 < 75).
Das ist ein Ansatz, der bei den Heuristischen Algorithmen in Abschnitt 13 systematisiert wird und z. B. zu dem A*-Algorithmus führt. In der Fachliteratur ist der Begriff des *Branch-and-Bound* Verfahrens nicht scharf abgegrenzt. In [Nau et al., 1984] wird eine Verallgemeinerung (*General Branch-and-Bound*) vorgeschlagen, die insbesondere den heuristischen A*-Algorithmus mit umfasst.

Im Folgenden werden Überlegungen zum Einfluss der Suchreihenfolge diskutiert und unterschiedliche Varianten auch in ihren Implementationen verglichen. Dazu wiederholen wir an dieser Stelle die Abbildung 4.7 aus dem Kapitel 4 (Backtracking) mit dem Vergleich unterschiedlicher Suchreihenfolgen hier als Abbildung 7.4.



(a) Tiefensuche **(b)** Breitensuche **(c)** 'Dynamische Suche'

Abbildung 7.4 Vergleich unterschiedlicher Suchreihenfolgen: Die Reihenfolge, in der Zweige des Suchbaums exploriert werden, ist entscheidend für die Effektivität der Schranken im *Branch-and-Bound* Verfahren. Besonders effizient ist die dynamische Suche, bei der diejenige Teillösung zuerst weiter verfolgt wird, die die größte obere Schranke, also das meiste Potenzial für eine gute Lösung hat. Dies entspricht dem A*-Algorithmus, siehe Kapitel 13.

Es folgen Implementierung mit unterschiedlichen Datenstrukturen für die Menge der Teillösungen, die die verschiedenen Suchreihenfolgen realisieren. Zunächst einige Überlegungen dazu:
Bei der Tiefensuche bewegt man sich immer nur zwischen Eltern und Kindknoten, entlang der Baumkanten. Sobald eine Teillösung entdeckt wurde, wird sie direkt 'besucht'. Daher reicht es, ein Objekt für die aktuelle Teillösung zu

speichern. Der Übergang geht mit "Schritt ausführen" und "Schritt zurücknehmen", wie beim Backtracking. Dies wurde in Listing 7.1 durch Rekursion implementiert.

Bei der Breitensuche und der dynamischen Suche werden die Teillösungen auch quer zur Baumstruktur (nicht entlang der Kanten) durchlaufen. Man muss also die entdeckten Teillösungen speichern. So können sie später besucht werden, wenn sie nach der Suchreihenfolge an der Reihe sind. Listing A.1 zeigt eine Implementation der Klasse `PartialSolution` für Teillösungen beim Rucksackproblem.

Um einen direkteren Vergleich der Implementation der drei Suchstrategien zu haben, beginnen wir in Listing A.2 mit einer nicht-rekursiven Implementation der Tiefensuche. Sie speichert die Teillösungen in einem Stack. Die Implementation mit Breitensuche in Listing A.3 ist quasi identisch, wobei nur der `Stack` durch eine `Queue` ausgetauscht wird, siehe auch den Abschnitt über Baum Traversierungen auf Seite 58.

Darauf aufbauend, kann eine *Branch-and-Bound* Variante implementiert werden, bei der jeweils diejenige Teillösung bearbeitet wird, die den größten *Upper Bound* hat. In der Implementation in Listing A.5 wird im wesentlichen die `Queue` durch eine `PriorityQueue` ersetzt. Dafür muss die Klasse der Teillösungen den `bound` als Attribut haben und `Comparable` gemäß dieses Bounds implementieren. Diese Funktionalität wird durch die Klasse `PSolBound` in Listing A.4 gewährleistet. In dieser Variante geschiet das Beschneiden des Suchbaumes indirekt. Teillösungen mit schlechtem *Bound* werden nicht aus der PQ geholt. Die zuerst gefundene Lösung ist immer eine optimale. Mehr dazu in Kapitel 13 über heuristische Algorithmen.

Die besprochenen Implementationsvarianten sind im Anhang A.1.

Bemerkungen zu den Implementationen

- Die Implementation mit `Queue` ist nur aus Gründen der Vollständigkeit und für den Übergang zur `PriorityQueue` gegeben. Die Breitensuche ergibt in den wenigsten Fällen, wenn überhaupt, eine sinnvolle Suchreihenfolge für *Branch-and-Bound*, da dann erst sehr spät die ersten vollständigen Lösungen gefunden werden und der *Bound* greifen kann.
- Die Implementation mit `Stack` ist in der Suchreihenfolge der rekursiven Lösung sehr ähnlich. Die rekursive Variante ist meist schneller. Allerdings ist die Größe des Rekursionsstacks beschränkt. Daher muss für größere Probleme die `Stack` Variante genutzt werden.
- Die Implementation mit `PriorityQueue` kann zu der Klasse der heuristischen Algorithmen gezählt werden, die in Kapitel 13 besprochen werden.

7.4 Das Wichtigste in Kürze

- ***Branch-and-Bound*** verwendet einen speziellen Ansatz zur Beschneidung des Suchraums.
- Zu jeder Teillösung wird eine Schranke der bestenfalls erreichbaren Lösung (*Bound*) bestimmt. Teillösungen deren *Bound* kleiner oder gleich dem Wert schon gefundener Lösungen ist, brauchen nicht weiterverfolgt zu werden.
- Als Erweiterung kann der *Bound* zusätzlich dazu verwendet werden, aussichtsreiche Teillösungen bevorzugt weiter zu verfolgen.

KAPITEL 8

Dynamisches Programmieren

Dieses Kapitel beschäftigt sich mit dem Lösungsparadigma **Dynamisches Programmieren**. Die Bezeichnung ›dynamisches Programmieren‹ ist recht unspezifisch und ist auf *Planung über die Zeit* zurückzuführen. Richard Bellman, der Namensgeber, hat den Ausdruck wohl aus strategischen Gründen gewählt.
Der Ansatz des Dynamischen Programmierens, um eine effiziente Lösung zu erreichen, unterscheidet sich deutlich von den bisher diskutierten Ansätzen wie Greedy Algorithmen und Branch-and-Bound. Jene Verfahren erzielen Effizienz durch das Ignorieren von Teillösungen, die nicht aussichtsreich sind. Der Suchbaum wird beschnitten. Bei der Dynamischen Programmierung werden 'gleichwertige' Teilprobleme in rekursiven Berechnungsbäumen miteinander identifiziert. Lösungen für solche Teilprobleme werden nur ein einziges Mal berechnet und gespeichert und können dann an den anderen Stellen im Berechnungsbaum abgerufen werden, wie später in Abbildung 8.4 gezeigt wird.

> Zur Abgrenzung wird hier kurz das Grundprinzip von **Divide-and-Conquer**, das in IntroProg eingeführt wurde, wiederholt:
>
> - Zerlege das Problem in Teilprobleme.
> - Wenn ein Teilproblem klein genug ist, löse es direkt;
> - Andernfalls benutze Rekursion, um eine Lösung zu erhalten.
> - Kombiniere Lösungen von Teilproblemen schrittweise zu Lösungen der jeweils übergeordneten Probleme.
>
> Beispiele für die Verwendung des *Divide-and-Conquer* Paradigmas sind: Binäre Suche, *Mergesort*, Strassen-Algorithmus für Matrixmultiplikation

8.1 Einführung in die Dynamische Programmierung

Das Paradigma der **Dynamischen Programmierung** (*dynamic programming*) geht folgendermaßen vor:

- Für die Bestimmung des Lösungswertes wird eine rekursive Formel aufgestellt, die eine optimale Lösung auf eine optimale Lösung von Teilproblemen zurückführt (*optimal substructure*).
- Dabei ist wichtig, dass viele der Teilprobleme gleichwertig sind (*overlapping subproblems*).
- Die Bestimmung der optimalen Lösung wird jeweils nur einmal ausgeführt und gespeichert. Für die wiederkehrenden Teilprobleme wird die gespeicherte Lösung abgerufen (*memoization* oder *tabulation*).

Die Dynamische Programmierung teilt mit dem *Divide-and-Conquer* Ansatz die Idee, Probleme rekursiv in Teilprobleme zu unterteilen bis sie klein genug zur direkten Angabe der Lösung sind. Die Lösungen der Teilprobleme werden zu einer Lösung des ursprünglichen Problems kombiniert. Damit die Dynamischen Programmierung angewendet werden kann, muss die Voraussetzung erfüllt sein, dass viele der Teilprobleme gleichartig sind. Dann kommt die spezifische Speicherung der Teillösungen zum Zuge und führt meistens zu einer deutlichen Steigerung der Effizienz. Hervorzuheben ist, dass die rekursive Formel für den Wert einer optimalen Lösung nur in der Konzeption benutzt wird. Eine direkte Umsetzung dieser rekursiven Formel mit einer rekursiven Berechnung hätte keinen Effizienzvorteil. Dieser kommt erst durch die für dynamische Programmierung typische Speicherung der Teillösungen. Der sogenannte *bottom-up* Ansatz kommt ganz ohne Rekursion aus, während beim *top-down* Ansatz zwar rekursive Aufrufe stattfinden, aber die Größe des Berechnungsbaums auf die Größe des Arrays zur Zwischenspeicherung begrenzt ist.

Die Zwischenspeicherung von Teilergebnissen ist im Prinzip ein sehr naheliegender Ansatz, auf den man in einfachen Fällen auch kommt, ohne den Ansatz des Dynamischen Programmierens zu kennen. Dies wird an dem einleitenden Beispiel der Fibonacci Zahlen verdeutlicht. Diese Grundidee des Zwischenspeicherns von Teillösungen kann aber auch bei einigen komplexen Problemen angewendet werden, bei denen es nicht direkt ersichtlich ist. In diesen Fällen braucht man Übung und Ideenreichtum, um eine geeignete Rekursionsformel für die Lösungswerte des Problems zu entwickeln. Und das lohnt sich: In vielen Fällen kann ein enormer Effizienzgewinn durch das Dynamische Programmieren erzielt werden.

Einführendes Beispiel: Fibonacci Zahlen

Zur Einführung der Techniken des dynamischen Programmierens betrachten wir die effiziente Bestimmung der n-ten **Fibonacci Zahl** F_n. Die Fibonacci Zahlen sind rekursiv definiert:

$$F_n = \begin{cases} 0 & \text{für } n = 0 \\ 1 & \text{für } n = 1 \\ F_{n-1} + F_{n-2} & \text{für } n \geq 2 \end{cases}$$

Abbildung 8.1 zeigt den Rekursionsbaum für die Berechnung von F_6 und das

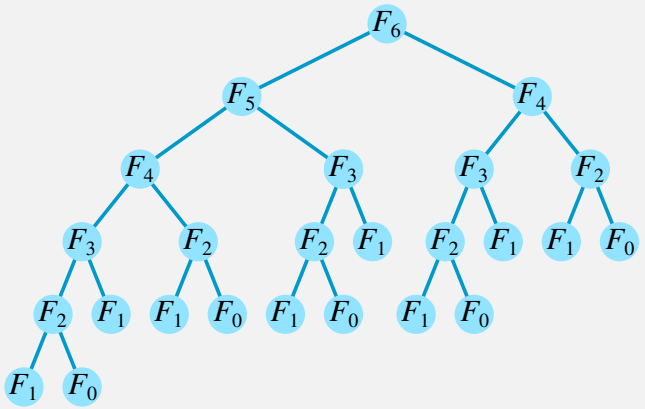


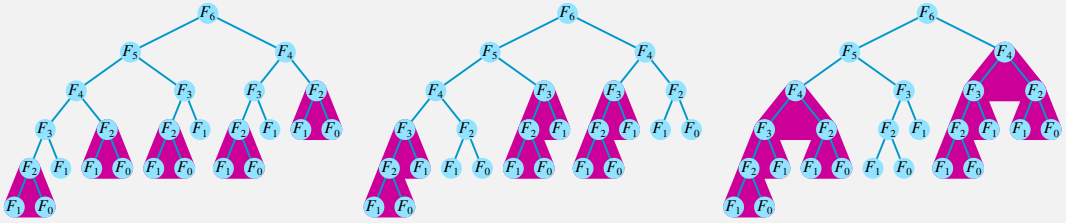
Abbildung 8.1 Rekursionsbaum zur Berechnung einer Fibonaccizahl: Die Berechnung von F_6 benötigt gemäß der Rekursionsformel die Werte F_5 und F_4. Diese hängen wiederum von ihren beiden Vorgängern ab. Die rekursive Berechnung von F_6 kann also durch den abgebildeten Baum dargestellt werden. Die Blätter mit den Werten F_0 und F_1 sind die Anfangsfälle, und alle inneren Knoten stehen für die Addition im Rekursionsfall.

folgende Listing enthält die Implementation in Java.

```
public static long fibonacci(int n) {
  if (n < 2) {
    return n;
  } else {
    return fibonacci(n-1) + fibonacci(n-2);
  }
}
```

Die Berechnung von F_n über die Rekursionsformel ist nicht effizient, da Teillösungen (F_k für $k < n$) vielfach berechnet werden, siehe Abbildung 8.2. Schon die Berechnung von F_{45} benötigt über 367 Millionen Aufrufe:

```
F(45) = 1134903170  calculated with 3672623805 recursive calls.
```



(a) (b) (c)

Abbildung 8.2 Redundanz in der rekursiven Berechnung der Fibonaccizahl F_6: Der Wert von F_2 wird fünfmal **(a)**, von F_3 dreimal **(b)** und von F_4 zweimal berechnet **(c)**.

Der Berechnungsbaum ist maximal binär verzweigt, also ist die Laufzeit in $O(2^n)$. Eine genauere Abschätzung erhält man mit folgenden Überlegungen: Es gilt $\lim_{n\to\infty} \frac{F_{n+1}}{F_n} = \phi = \frac{\sqrt{5}+1}{2} \approx 1.6180$. Die Abschätzung $\frac{F_{n+1}}{F_n} \approx 1.6$ gilt schon ab $n = 6$ recht gut. Daher ergibt sich als Wachstumsverhalten $F_n \approx 1.6^n$. Da bei dem Rekursionsbaum an den Blättern die Werte 0 und 1 (und keine größeren Werte) stehen, muss die Größenordnung der Anzahl der rekursiven Aufrufe bei 1.6^n liegen, (denn, um F_n zu berechnen, müssen F_n-viele Blätter mit dem Wert 1 aufsummiert werden.) Die rekursive Berechnung der Fibonacci Zahlen hat also eine exponentielle Laufzeit.
Das Problem ist, dass dieselben Teillösungen vielfach berechnet werden. Gemäß dem Paradigma der Dynamischen Programmierung können die Doppeltberechnungen durch Speicherung der Zwischenlösungen vermieden werden. Diese Strategie wird auch memoization (ohne 'r'!) oder tabulation genannt, je nachdem welche Strategie bei der Speicherung verwendet wird, *top-down* oder *bottom-up*.

Berechnung mit Zwischenspeichern nach Bedarf: *top-down*

Wir können ein Feld zur Speicherung der Teillösung verwenden, und Teillösungen nur nach Bedarf berechnen, so dass keine Doppeltberechnungen stattfinden. Diese Strategie wird *memoization* genannt.

```
public class FibonacciTopDown {    // memoization
  private static long[] F;

  public static long fibonacci(int n) {
    F = new long[n+1];
    F[0] = 0;
```

```
    F[1] = 1;
    for (int k = 2; k <= n; k++) {
      F[k] = -1;
    }
    return fibo(n);
  }

  public static long fibo(int n) {
    if (F[n] < 0) {
      F[n] = fibo(n-1) + fibo(n-2);
    }
    return F[n];
  }

  public static void main(String[] args) {
    int n = 12;
    long Fn = fibonacci(n);
    System.out.println("F(" + n + ") = " + Fn);
  }
}
```

Das Feld `F` wird benutzt, um Teillösungen zwischenzuspeichern. Der Wert `-1` zeigt an, dass der Wert noch nicht berechnet wurde.

Bei dem *top-down* Ansatz gehen die Berechnungsanfragen von oben nach unten. Die Anfangswerte sind definiert. Von dort werden die Werte *bottom-up* berechnet und gespeichert, siehe Abbildung 8.3.

Berechnungen mit Zwischenspeichern auf Vorrat: *bottom-up*

Wenn sowieso alle Werte des Feldes berechnet werden müssen, kann die Berechnung auch gleich *bottom-up* durchgeführt werden. Diese Strategie zur Füllung des Zwischenspeichers wird *tabulation* genannt.

Bei den Fibonaccizahlen ist dies der Fall: Zur Berechnung von F_n müssen alle vorherigen Werte, also alle F_k für $k < n$ berechnet werden. Also kann dies direkt *bottom-up* erledigt werden, ohne auf den 'Bedarf' zu warten.

```
public static long fibonacci(int n) {    // tabulation
    F = new long[n+1];
    F[0] = 0;
    F[1] = 1;
    for (int k = 2; k <= n; k++) {
      F[k] = F[k-1] + F[k-2];
    }
    return F[n];
  }
```

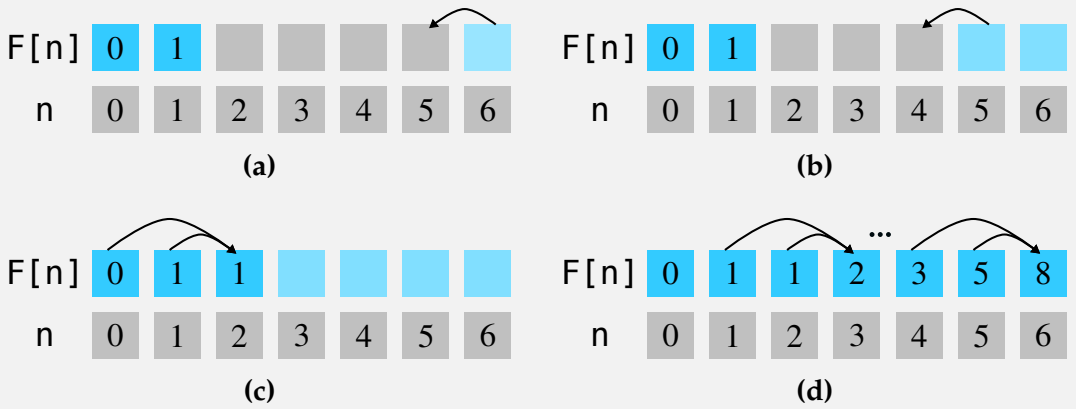


Abbildung 8.3 Top-down Ansatz zur Zwischenspeicherung (*memoization*): Bei der ersten Anfrage an einen Wert des Zwischenspeichers wird die entsprechende Berechnung gestartet und gespeichert. Bei allen zukünftigen Anfragen kann der Wert aus dem Zwischenspeicher zurückgegeben werden. **(a)** Die Anfrage des Wertes von F_6 resultiert in einer Anfrage an F_5 (und später auch an F_4). **(b)** Die Anfrage an F_5 löst eine Anfrage an F_4 aus usw. **(c)** Schließlich landet die Anfrage bei F_2. Dort kann sie direkt aus den Anfangswerten von F_0 und F_1 berechnet werden. **(d)** Dies führt dazu, dass nun von unten kommend alle angefragten Werte berechnet werden. Nun lösen doppelte Anfragen keine doppelten Berechnungen aus, da das Ergebnis aus dem Speicher abgerufen wird.

Die Zwischenspeicherung führt dazu, dass große Teile des Rekursionsbaums nicht durchlaufen werden müssen. Die entsprechenden Rechnungen wurden an andere Stelle schon ausgeführt und gespeichert, siehe Abbildung 8.4.

Speichereffizienz?

Die Implementationen mit *memoization* hat linearen Speicherbedarf für das Feld der Zwischenlösungen. Es stellt sich die Frage, ob man den Speicherbedarf reduzieren kann und in diesem Fall sogar mit konstantem Speicherbedarf auskommt. Dies ist eher ungewöhnlich, aber bei der Berechnung der Fibonaccizahlen ist es tatsächlich möglich. Bei dem *bottom-up* Ansatz benötigt man nämlich immer nur die letzten beiden Werte! Es reicht also ein Feld der Länge 2 zur Zwischenspeicherung:

```
public static long fibonacci(int n)
  {
    F = new long[2];
    F[0] = 0;
    F[1] = 1;
    for (int k = 2; k <= n; k++)
```

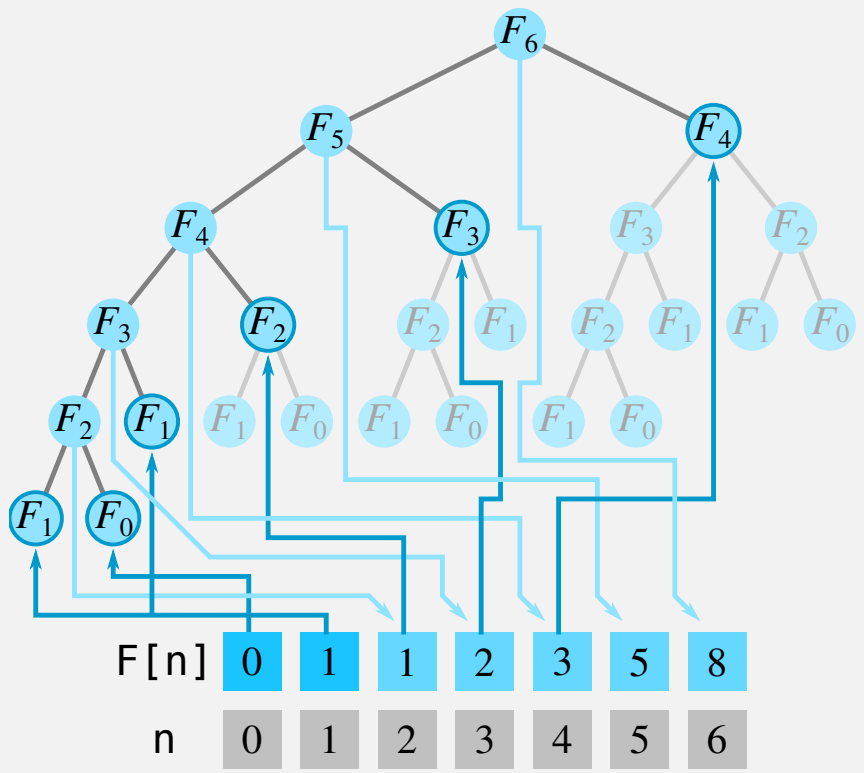


Abbildung 8.4 Einsparung durch Zwischenspeicherung bei Fibonacci Rekursion: Sobald der Wert einer Fibonaccizahl berechnet wurde, wird er in dem Array `F[n]` gespeichert und kann dann bei späteren Anfragen ohne Neuberechnung abgerufen werden. Dadurch müssen große Teile des Rekursionsbaums nicht berechnet werden.

```
        F[k%2] = F[(k-1)%2] + F[(k-2)%2];
    return F[n%2];
}
```

Wir werden sehen, dass auch bei komplexeren Problemen machmal der Speicherbedarf der Dynamischen Programmierung reduziert werden kann.

Nebenbemerkung zur Laufzeit

Es sieht auf den ersten Blick so aus, als ob die Implementationen zur Berechnung der Fibonaccizahlen mit Dynamischer Programmierung eine lineare Laufzeit haben. Dies ist aber nicht der Fall. Die Zahlen, die bei der Berechnung addiert werden, wachsen exponentiell schnell. Dadurch kann nicht mehr angenommen werden, dass die Rechenoperationen in konstanter Zeit durchgeführt werden kann. Die Laufzeit der Addition ist linear in der Anzahl der Ziffern der

Summanden. So kommt es, dass die tatsächliche Laufzeit unserer `fibonacci()` Methode mit Dynamischer Programmierung in $\boldsymbol{O}(n^2)$ liegt. Dies ist immer noch ein immenser Vorteil gegenüber der ursprünglichen rekursiven Berechnung mit exponentieller Laufzeit. Ein Verfahren zur Berechnung der Fibonaccizahlen in $\boldsymbol{O}(n \log n)$ Zeit kann in [Erickson, 2019, S. 103] nachgelesen werden.

Von Fibonacci zum Dynamischen Programmieren

Das Fibonacci Beispiel hat ein paar wichtige Aspekte des dynamischen Programmierens beleuchtet: In einer rekursiven Formel wird eine Lösung auf die Lösungen von Teilproblemen zurückgeführt. Wenn dieselben Teilprobleme vielfach gebraucht werden, kann die Effizienz durch eine Speicherung der Lösungswerte erhöht werden. Die Berechnung kann *top-down* bei Bedarf oder *bottom-up* auf Vorrat durchgeführt werden. Der *top-down* (*memoization*) Ansatz kann günstig sein, wenn nur manche Teillösungen gebraucht werden und im Voraus nicht klar ist, welche das sein werden. Der *bottom-up* (*tabulation*) Ansatz kann die Speichereffizienz steigern, wenn die Teillösungen nur 'schichtweise' gebraucht werden, also Teillösungen überschrieben werden können.

Das Grundprinzip der dynamischen Programmierung lautet: Redundante Berechnung durch Speicherung von Teillösungen vermeiden! Um ein Optimierungsproblem per Dynamischer Programmierung zu lösen, sucht man zunächst eine rekursive Formel. Dafür nimmt man eine optimale Lösung als gegeben an und versucht den Wert der Lösung in Abhängigkeit von Teillösungen darzustellen (Substrukturanalyse, *optimal substructure*). Dabei wird oft eine Fallunterscheidung gemacht. Bei komplexeren Problemen werden viele Teillösungen benötigt. Sind darunter sehr viele gleiche Teilprobleme (*overlapping subproblems*), kommt der Vorteil der dynamischen Programmierung zum Tragen. Falls sehr viele unterschiedliche Teilprobleme vorkommen, benötigt der Ansatz des Dynamischen Programmierens oft zu viel Speicher und der Laufzeitgewinn ist nicht so groß.

Eine Einschränkung bei der Dynamischen Programmierung für Optimierungsprobleme ist, dass mit der Rekursionsformel nur der optimale Wert definiert wird, nicht die zugehörige Lösung selbst. Bei dem einführenden Beispiel der Fibonaccizahl kam dieser Aspekt nicht zum Tragen, da es dabei sowieso nur um die Berechnung eines Wertes geht und nicht um das Finden einer Lösung mit optimalem Wert.

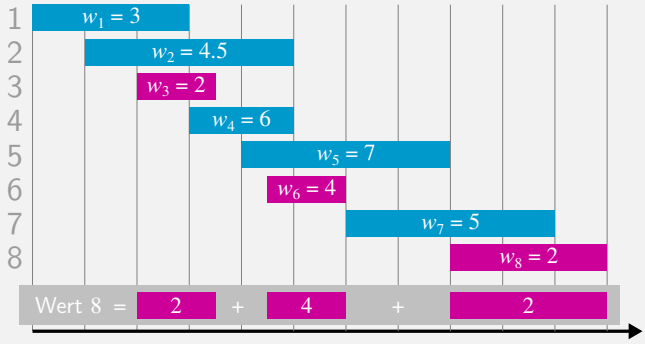
Wir werden sehen, dass man bei Optimierungsproblemen oft zwei Durchläufe benötigt: den ersten zur Bestimmung des optimalen Wertes und den zweiten zur Rekonstruktion der zugehörigen Lösung. Manchmal kann die Lösung bei der Bestimmung des optimalen Wertes mit gespeichert werden und der zweite Durchlauf ist nicht notwendig.

8.2 Beispiel 1: Gewichtete Intervallauswahl

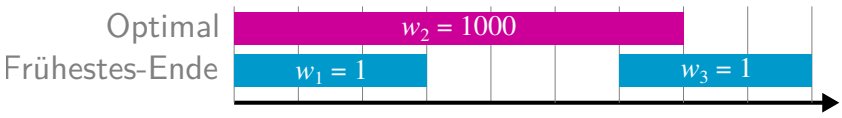
Als Erweiterung unseres ersten Beispiels in dem Kapitel zu Greedy Algorithmen betrachten wir nun die Auswahl *gewichteter* Intervalle.

Gewichtete Intervallauswahl

Es gibt Anfragen $1, \ldots, n$ zur Nutzung einer Ressource (z.B. Raum, Prozessor, Messgerät) in dem Zeitintervall $I_k = [s_k, f_k)$ und mit dem Gewicht w_k.
Ziel: Wähle kompatible Intervalle $S \subseteq \{1, \ldots, n\}$, so dass das Gewicht der gewählten Intervalle $\sum_{k \in S} w_k$ (= Wert der Lösung) maximal ist.



Bei der *ungewichteten* Intervallauswahl (entsprechend gleichen Gewichten w_k für alle k) ist der Greedy-Algorithmus Frühestes-Ende optimal. Bei der *gewichteten* Intervallauswahl kann er grandios scheitern, wie das folgende Beispiel zeigt:



Wir benötigen also einen neuen Ansatz. Da wir uns in dem Kapitel über Dynamische Programmierung befinden, wird es nicht verwundern, dass dieser Ansatz hier zum Tragen kommt.
Dafür überlegt man, wie man eine optimale Lösung aus optimalen Lösungen von Teilproblemen ableiten kann. Im Folgenden setzen wir voraus, dass die Intervalle *aufsteigend nach Endzeiten* sortiert sind, also in der Reihenfolge, die beim Greedy Algorithmus erfolgreich war.
Wir definieren zunächst eine Funktion, die zu jedem Intervall das Vorgängerintervall angibt, das am wenigsten Lücke lässt (oder 0, falls jenes nicht exisitert):

$$p(j) = \begin{cases} \max\{i \in \mathbb{N} \mid i < j \text{ und } i \text{ kompatibel mit } j\} & \text{falls existent} \\ 0 & \text{sonst} \end{cases}$$

Die Definition wird in Abbildung 8.5 veranschaulicht. Intervall j ist mit Intervall $p(j)$ kompatibel und auch mit allen Intervallen vor $p(j)$ (durch die Sortierung nach Endzeitpunkten).

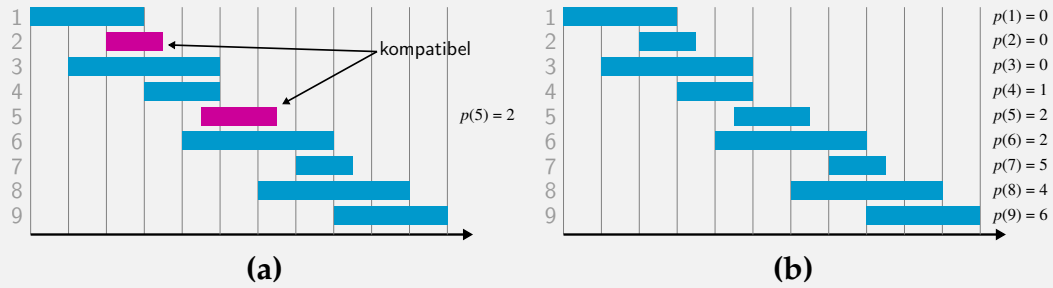


Abbildung 8.5

Mit dieser Definition können wir den Ansatz mit Dynamischer Programmierung formulieren. Fassen wir zunächst das generelle Vorgehen zusammen. Es wird eine Funktion $\textsc{Opt}(k)$ definiert, die den Wert einer optimalen Lösung zu der gegebenen Eingabe k zurückgibt (hier: maximales Gewicht für Anfragen $1, \ldots, k$). Durch Überlegungen, wie der optimale Wert durch den optimalen Wert von Teilproblemen dargestellt werden kann (Substrukturanalyse), wird eine Rekursionsgleichung für $\textsc{Opt}$ aufgestellt. Diese bildet dann den Ansatzpunkt für die Lösung per Dynamischer Programmierung. Wie für Rekursionsgleichungen üblich, hat die $\textsc{Opt}$ Funktion einen oder mehrere Anfangsfälle, sowie einen Rekursionsfall. Um die Formel für den Rekursionsfall aufzustellen, ist es oft hilfreich, sich vorzustellen, dass eine optimale Lösung gegeben ist. Dann macht man eine Fallunterscheidung je nachdem, wie diese Lösung beschaffen ist und führt in jedem einzelnen Fall den optimalen Wert auf den optimalen Wert eines Teilproblems zurück. In der Rekursionsformel wird die Fallunterscheidung durch eine Maximumsbildung (bzw. Minimumsbildung für Minimierungsprobleme) ausgedrückt. Sehen wir uns an, wie das in unserem ersten Beispiel funktioniert.

Definition: Sei $\textsc{Opt}(k)$ der optimale Wert für Intervalle $1, \ldots, k$.

- **1. Fall:** In der optimalen Lösung ist Intervall k enthalten.

  Nun muss der Wert der optimalen Lösung auf ein Teilproblem zurückgeführt werden. Von Intervall k wissen wir durch die Fallunterscheidung, dass es ausgewählt ist. Nach der Definition von $p(k)$ sind die Intervalle $p(k)+1, \ldots, k-1$ mit der Lösung nicht kompatibel. Wir greifen daher auf die optimale Lösung für $1, \ldots, p(k)$ zurück und fügen ihr das Intervall k hinzu. Der optimale Wert $\textsc{Opt}(k)$ ist der optimale Wert des Teilproblems, also $\textsc{Opt}(p(k))$ plus das Gewicht w_k des hinzugefügten Intervalls k.

- **2. Fall:** In der optimalen Lösung ist Intervall k nicht enthalten.

Die optimale Lösung ist identisch mit der optimalen Lösung des Teilproblems für die Intervalle $1, \ldots, k-1$. Da Intervall k nicht ausgewählt wurde, stimmt der optimale Wert $\textsc{Opt}(k)$ mit dem optimalen Wert des Teilproblems, also $\textsc{Opt}(k-1)$ überein.

Der optimale Wert wird also durch folgende Rekursionsgleichung beschrieben:

$$\textsc{Opt}(k) = \begin{cases} 0 & \text{falls } k = 0 \\ \max(\underbrace{w_k + \textsc{Opt}(p(k))}_{\text{Fall 1}}, \underbrace{\textsc{Opt}(k-1)}_{\text{Fall 2}}) & \text{sonst} \end{cases}$$

Die Rekursionsgleichung kann wie folgt implementiert werden:

```
1  Sortiere Anfragen nach Endzeit        // f_1 ≤ ··· ≤ f_n
2  Berechne p_1, ..., p_n
3  opt(n)
4
5  procedure opt(k)
6  if k = 0
7    return 0
8  else
9    return max( w_k + opt(p_k), opt(k − 1) )
10 end
```

Diese Implementierung entspricht der ursprünglichen Implementierung zur Berechnung der Fibonaccizahlen. Wie dort ist die Laufzeit sehr schlecht, da die Lösungen für Teilprobleme vielfach neuberechnet werden. Die Rekursionsformel für $\textsc{Opt}$ liefert nur die Basis für den Ansatz der dynamischen Programmierung. Nun muss noch die Speicherung der Zwischenlösungen hinzu kommen. Dazu hatten wir die beiden Ansätze *top-down* und *bottom-up* kennengelernt. Wie sollte in diesem Fall vorgegangen werden? Aus der Rekursionsformel für $\textsc{Opt}$ sehen wir, dass *alle* Teillösungen benötigt werden, da $\textsc{Opt}(k)$ von $\textsc{Opt}(k-1)$ abhängt. Daher ist der *bottom-up* Ansatz vorzuziehen. Die Werte der Teillösungen *m[k]* werden *bottom-up* von 0 bis zu dem gesuchten Wert n berechnet werden:

Listing 8.1 Algorithmus zur Bestimmung der Intervallauswahl mit maximalem Gewicht über dynamische Programmierung

```
1  Sortiere Anfragen nach Endzeit        // f_1 ≤ ··· ≤ f_n
2  Berechne p_1, ..., p_n
3
4  m[0] ← 0
5  for k = 1 to n
```

```
6    m[k] ← max( w_k + m[p_k], m[k−1] )
7  return m[n]
```

Laufzeit für gewichtete Intervallauswahl mit dynamischer Programmierung

Der Algorithmus in Listing 8.1 bestimmt die Auswahl aus n gewichteten Intervallen mit maximalem Gewicht in einer Laufzeit in $\Theta(n \log n)$ und mit Speicherbedarf in $\Theta(n)$.

Beweis. Das Sortieren der Aufträge nach Endzeiten in Zeile 1 benötigt eine Laufzeit in $\Theta(n \log n)$. Das gleiche gilt für die Berechnung von p_k durch Sortierung nach Anfangszeiten in Zeile 2. Initialisierung (nicht im Pseudocode enthalten) und die Berechnung des Feldes $m[]$ (Zeilen 5 und 6) benötigen beide eine Laufzeit in $\Theta(n)$.
Die Vorbereitung hat also insgesamt eine Laufzeit in $\Theta(n \log n)$ und die eigentliche Intervallauswahl bei gegebenen Sortiertungen $\Theta(n)$. Insgesamt ergibt sich also, wie behauptet, die Laufzeit in $\Theta(n \log n)$.
Der Speicherbedarf für Feld $m[]$ lässt sich nicht wie im Fibonacci Beispiel reduzieren, da die Rekursionsformel mit $p(k)$ auf unterschiedliche Vorlösungen zurückgreift. □

Die optimale Lösung feststellen

Der bisherige Ansatz bestimmt den *Wert* der optimalen Lösung. Wie bekommen wir die *Lösung* selbst, also in unserem Beispiel die ausgewählten Intervalle? Dies kann in einem zweiten Durchlauf geschehen:

```
1 procedure findSolution(k)
2 if  k = 0
3   return ∅
4 else  if  w_k + m[p_k] > m[k−1]
5   return {k} ∪ findSolution(p_k)
6 else
7   return  findSolution (k−1)
8 end
```

Bemerkung: Die Lösung muss nicht eindeutig sein. In dem Fall $w_k + m[p_k] = m[k-1]$ ist die Maximumsbildung in OPT ambivalent. Daher kann Intervall k ausgewählt werden, muss es aber nicht. Beim Rekonstruieren der Lösung kann man sich entscheiden, welche Variante man präferiert (striktes $>$ oder ein $\geq$ in Zeile 4).

8.3 Beispiel 2: Das 0/1-Rucksack Problem

Im Gegensatz zu dem teilbaren Rucksackproblem konnte das 0/1-Rucksackproblem *nicht effizient* per Greedy Ansatz gelöst werden, siehe Abschnitt 5.7.2. Aus Abschnitt 2.6 wissen wir, dass das 0/1-Rucksack Problem (auch bei der Beschränkung auf ganzzahlige Gewichte) NP-Vollständig ist. Es gibt also keine Hoffnung für einen effizienten Ansatz mit dynamischer Programmierung. Wir probieren es trotzdem! (Und beschränken uns dabei auf ganzzahlige Gewichte.)

Erster Ansatz für das 0/1-Rucksack Problem

Definition: Sei $\text{OPT}(k)$ der Wert einer optimalen Lösung für Objekte $1, \ldots, k$.

- **1. Fall** Objekt k ist in der Lösung nicht ausgewählt.

  Dann besteht die Lösung in der optimalen Lösung für Teilproblem $1, \ldots, k-1$ und der Wert ändert sich nicht.

- **2. Fall** Objekt k ist ausgewählt.

  Ohne weitere Information lässt sich die Lösung nicht auf Teillösungen zurückführen!

Dieser Ansatz ist im zweiten Fall zum Scheitern verurteilt, da wir nicht wissen, wieviel freie Kapazität vorhanden ist, um weitere Objekte auszuwählen (und auch nicht, ob Objekt k überhaupt ausgewählt werden konnte). Wir müssen also die Restkapazität als weitere Variable in OPT mitberücksichtigen.

Richtiger Ansatz für das 0/1-Rucksack Problem

Definition: Sei $\text{OPT}(k, W)$ der Wert einer optimalen Lösung **O** für Objekte $1, \ldots, k$ mit Maximalgewicht W. Die Reihenfolge der Objekte in **O** ist beliebig.

- **1. Fall:** Es gilt $w_k > W$.

  Dann kann Objekt k nicht ausgewählt werden (Kapazitätsüberschreitung). Die Lösung **O** stimmt mit der optimalen Lösung für Teilproblem $1, \ldots, k-1$ mit Maximalgewicht W überein. Es gilt also $\text{OPT}(k, W) = \text{OPT}(k-1, W)$.

- **2. Fall:** Es gilt $w_k \leq W$ und Objekt k ist in der Lösung **O** nicht ausgewählt.

  Auch in diesem Fall stimmt die Lösung **O** mit der optimalen Lösung für Teilproblem $1, \ldots, k-1$ mit Maximalgewicht W überein. Es gilt also $\text{OPT}(k, W) = \text{OPT}(k-1, W)$.

- **3. Fall:** Es gilt $w_k \leq W$ und Objekt k ist in der Lösung **O** ausgewählt[1].

[1] Wenn Objekt k in der Lösung ausgewählt ist, muss $w_k \leq W$ aufgrund der Gewichtsbeschränkung gelten.

Die Lösung **O** besteht in der optimalen Lösung für Teilproblem $1, \ldots, k-1$ mit Maximalgewicht $W - w_k$, wobei der Lösungswert um v_k, den Wert von Objekt k größer ist. Es gilt also $\textsc{Opt}(k, W) = v_k + \textsc{Opt}(k-1, W - w_k)$.

Der erste Fall (Einschränkung durch die Problemstellung) muss mit einer Fallunterscheidung abgefangen werden, während die anderen beiden Fälle (Wahlmöglchkeit in der Lösung) mit einer Maximumsbildung verknüpft werden:

$$\textsc{Opt}(k, W) = \begin{cases} 0 & \text{falls } k = 0 \\ \textsc{Opt}(k-1, W) & \text{falls } w_k > W \\ \max(\underbrace{v_k + \textsc{Opt}(k-1, W - w_k)}_{k \text{ ausgewählt}}, \underbrace{\textsc{Opt}\,(k-1, W)}_{k \text{ nicht ausgewählt}}) & \text{sonst} \end{cases}$$

Weitere Entscheidungen zur Implementation

Die Funktion Opt greift in der ersten Dimension nur auf Vorgänger zu, d.h. $\textsc{Opt}(k, \cdot)$ hängt nur von $\textsc{Opt}(k-1, \cdot)$ ab. Daher könnte bei dem *bottom-up* Ansatz der Speicherbedarf zur Bestimmung des optimalen Wertes auf zwei Spalten der Matrix M beschränkt werden, analog zu dem Fibonacci Beispiel auf Seite 119. Allerdings ist dann unklar, wie die eigentliche Lösung (Liste der angewählten Objekte) ausgelesen werden kann. Dies geschieht normalerweise in einem zweiten Durchlauf, wobei das Feld / die Matrix der gespeicherten Teillösungen durchlaufen wird, siehe Seite 125. Dies ist aber bei der Speicher-effizienten Variante nicht möglich.
Bei der Rekursionsformel für Opt werden nicht unbedingt alle Teillösungen benötigt. Daher benutzen wir hier den *top-down* Ansatz. Der *bottom-up* Ansatz ist leichter zu implementieren, aber u. U. etwas weniger effizient.

Knapsack Top-Down Implementation

Listing 8.2 Lösung des 0/1-Rucksackproblems mit Dynamischer Programmierung (top-down)

```
1  public class Knapsack {
2
3    public int W, K;
4    public int[] weight;
5    public double[] value;
6    private double[][] M;
7    private Queue<Integer> inventory = new LinkedList<>();
8
9    public Knapsack(int[] weight, double[] value, int W) {
10     this.W = W;
```

```
11      this.K = weight.length - 1;
12      this.weight = weight;
13      this.value = value;
14      M = new double[K+1][W+1];
15      for (int w = 0; w <= W; w++) {
16        M[0][w] = 0.0;
17        for (int k = 1; k <= K; k++)
18          M[k][w] = -1.0;
19      }
20    }
21
22    public double opt(int k, int w)
23    {
24      if (M[k][w] < 0)
25        if (weight[k] > w)
26          M[k][w] = opt(k-1, w);
27        else
28          M[k][w] = Math.max( value[k] + opt(k-1, w-weight[k]),
29                              opt(k-1, w) );
30      return M[k][w];
31    }
32
33    public void findSolution(int k, int w)
34    {
35      if (k == 0) return;
36      else if (weight[k] > w)
37        findSolution(k-1, w);
38      else if (value[k] + M[k-1][w-weight[k]] > M[k-1][w]) {
39        findSolution(k-1, w-weight[k]);
40        inventory.add(k);
41      } else
42        findSolution(k-1, w);
43    }
44
45    public static void main(String[] args)
46    {
47      double[] value = {0, 2, 3, 1, 5, 7, 3, 6};
48      int[] weight =   {0, 3, 4, 2, 4, 7, 3, 5};
49      int maxWeight = 14;
50
51      Knapsack knapsack = new Knapsack(weight, value, maxWeight);
52      double optValue = knapsack.opt(knapsack.K, knapsack.W);
53      knapsack.findSolution(knapsack.K, knapsack.W);
54
55      System.out.println("Optimal load: ");
56      for (int k : knapsack.inventory) {
57        System.out.println(knapsack.weight[k] + " - " + knapsack.value[k]);
58      }
59    }
60  }
```

Laufzeit und Speicherbedarf der Knapsack Implementierung

Laufzeit der Knapsack Implementierung

Die 0/1-Knapsack Implementierung für ganzzahlige Gewichte basierend auf dynamischer Programmierung hat eine Laufzeit in $\boldsymbol{O}(KW)$, wobei K die Anzahl der Objekte und W die Kapazität des Rucksacks ist. Der Speicherbedarf ist ebenfalls in $\boldsymbol{O}(KW)$.

Beweis. Die Methode `opt()` geht durch die Abfrage in Zeile 23 für jedes Paar (k, w) nur einmal in die Rekursion. Somit ist die Laufzeit von `opt()` und von der Initialisierung in $\boldsymbol{O}(KW)$. □

Dieses Resultat wirft eine wichtige Frage auf:

0/1-Knapsack ist NP-vollständig und die Laufzeit ist in $\boldsymbol{O}(KW)$??

Auflösung: Diese Abschätzung 'zählt nicht', da sie nicht nur von der *Anzahl* der Eingabeobjekte, sondern auch von einem *Eingabewert* abhängt. Um diesen Aspekt genauer zu diskutieren, benötigen wir die folgende Definition:

Pseudo-polynomieller Algorithmus

Ein Algorithmus zur Lösung eines Problems, das als Eingaben ganze Zahlen hat, heißt **pseudopolynomiell**, wenn seine Laufzeit durch ein Polynom in der Eingabegröße und dem größten Absolutwert der Eingabezahlen beschränkt ist.

Die Knapsack Implementierung ist also pseudopolynomiell. Nach der formalen Definition darf eine Laufzeit-Komplexität nur von der Bitanzahl abhängen, die benötigt wird, um die Eingabedaten zu kodieren. Wenn die Kapazitätsgrenze W mit n Bits kodiert wird, kann die Grenze bis zu $W = 2^n - 1$ betragen. Die Laufzeit in Abhängigkeit von der Eingabelänge in Bits ist daher $\boldsymbol{O}(K2^n)$, also exponentiell! Genau genommen müssten noch die Bits berücksichtigt werden, die benötigt werden, um die K Objekte zu kodieren. Dies macht allerdings nur einen konstanten Faktor aus, wenn nur W variiert wird.

8.4 Beispiel 3: Editierdistanz

Die Editierdistanz (*edit distance*, auch Levenshtein-Distanz) ist ein Maß für die Ähnlichkeit zwischen zwei Zeichenketten. Sie dient als Grundlage für den *diff* Befehl und approximative *String-Matching* Algorithmen. Das Verfahren findet

auch Anwendung in der Bioinformatik zur Analyse von DNA- und RNA-Sequenzen. Dort wird das Distanzmaß noch etwas angepasst, um die biologischen Gegebenheiten besser zu modellieren.

Editierdistanz (*edit distance*)

Die Editierdistanz zwischen zwei Strings a und b ist die minimale Anzahl von elementaren Buchstabenoperationen, die notwendig sind, um a in b umzuwandeln. Die elementaren Buchstabenoperationen sind:

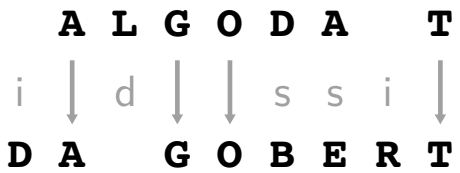
- **D** Einen Buchstaben löschen (*delete*)
- **I** Einen Buchstaben einfügen (*insert*)
- **S** Einen Buchstaben ersetzen (*substitute*)

Das Editieren kann buchstabenweise von vorne entlang des Strings a durchgeführt werden. Dann wird noch folgende Aktion verwendet:

- **–** Einen Buchstaben unverändert übernehmen

Das Übernehmen eines Buchstaben zählt *nicht* als Operation im Sinne der Editierdistanz.

Schauen wir uns ein Beispiel an. Wie kann man ALGODAT in DAGOBERT verwandeln? Ein Möglichkeit ist:



Diese Umwandlung ergibt eine Distanz von 5 (zweimal Einfügen, einmal Löschen, zweimal Ersetzen). Die Editierdistanz ist die *kleinste* Anzahl von Buchstabenoperationen. Geht es noch kürzer als in dem Beispiel? Wir entwerfen einen Ansatz mit Dynamischer Programmierung für dieses Problem.
Gemäß dem oben beschriebenen Ansatz der dynamischen Programmierung definieren wir eine Funktion OPT. Zunächst führen wir einige Schreibweisen ein.

- Für einen String a bezeichnet $a[1:i]$ den String, der aus den ersten i Zeichen von a besteht.
- $a[1:0]$ ist der leere String.
- Des Weiteren bezeichnet $a[i]$ (für $i > 0$) den i-ten Buchstaben von a und

$$a[i] \neq b[j] \quad = \quad \begin{cases} 0 & \text{falls die Zeichen } a[i] \text{ und } b[j] \text{ gleich sind} \\ 1 & \text{sonst} \end{cases}$$

Wir suchen eine rekursive Definition $\text{OPT}(i, j)$ für die Editierdistanz zwischen den Substrings $a[1 : i]$ und $b[1 : j]$. Dazu betrachten wir zunächst die Randfälle.

- Für $j = 0$ soll String $a[1 : i]$ in einen leeren String umgewandelt werden. Dazu müssen alle Zeichen von $a[1 : i]$ gelöscht werden (also i-mal die Operation **D** ausgeführen): $\text{OPT}(i, 0) = i$.
- Für $i = 0$ soll ein leerer String in den String $b[1 : j]$ umgewandelt werden. Dazu müssen alle Zeichen des Strings $b[1 : j]$ eingefügt werden (also j-mal die Operation **I** ausgeführen): $\text{OPT}(0, j) = j$.
- Für $i, j > 0$ betrachten wir alle möglichen Operationen, addieren die Editierkosten (=1 für **D**, **I** und **S**) zu der jeweiligen Teillösung und wählen das Minimum:

$$
\begin{aligned}
\text{OPT}(i,j) = \min(\ & \text{OPT}(i, j-1) + 1, && \text{einfügen von } b[j] \\
& \text{OPT}(i-1, j) + 1, && \text{löschen von } a[i] \\
& \text{OPT}(i-1, j-1) + (a[i] \neq b[j]) && \text{ersetzen oder übernehmen} \\
&) &&
\end{aligned}
$$

Der Aufbau der Matrix, deren Werte durch OPT definiert werden, ist in Abbildung 8.6 für a = `ALGODAT` und b = `DAGOBERT` dargestellt. Die Randfälle sind

Abbildung 8.6 Matrix der Teillösungen für die Editierdistanz: In jedem Feld der Matrix wird die Editiertdistanz für alle möglichen Anfangsteile der beiden Zeichenketten gespeichert **(a)** . **(b)** Die Randfälle können unabhängig von den Zeichenketten bestimmt werden. Die Editierdistanz der kompletten Zeichenketten 'ALGODAT' und 'DAGOBERT' wird in dem Eintrag unten rechts stehen. Gemäß der rekursiven Definition $\text{OPT}(i, j)$ werden für den Wert jedes Feldes die Werte der Nachbarfelder links, oberhalb und links-oberhalb benötigt. Es müssen also die Werte der ganzen Matrix bestimmt werden.

unabhängig von den Zeichenketten. Die gesuchte Editierdistanz der vollständigen Strings wird in Eintrag (7,8) unten rechts in der Matrix stehen. Um den Wert zu bestimmen, werden alle Teillösungen benötigt. Daher bringt die "Berechnung bei Bedarf" des *top-down* Ansatzes keinen Vorteil. Es ist also günstiger, direkt alle Werte der Matrix *bottom-up* zu bestimmen.
Wie bei bei der Dynamischen Programmierung üblich, wird zunächst nur der Wert der Lösung bestimmt, also die Editierdistanz. Um die Editiersequenz auszugeben, müsste wieder ein zweiter Durchlauf erfolgen, der anhand der gespeicherten `D[][]` Werte die optimale Sequenz rekonstruiert.
Falls bei einer Anwendung sehr lange Zeichenfolgen verarbeitet werden müssen und Speicherplatz eingespart werden soll, kann die Matrix auf zwei Zeilen, oder alternativ auf zwei Spalten, beschränkt werden. Dann funktioniert allerdings, wie bei dem Rucksack Beispiel besprochen, die Rekonstruktion der Lösung, also der Editiersequenz, nicht.

Listing 8.3 Implementation der Editierdistanz mit Dynamischer Progammierung

```
public class EditDistance {
  private String a, b;
  private int an, bn;
  private int D[][];

  public EditDistance(String a, String b) {
    this.a = a;
    this.b = b;
    an = a.length();
    bn = b.length();
    D = new int[an + 1][bn + 1];
    for (int i = 0; i <= an; i++) {
      D[i][0] = i;
    }
    for (int j = 0; j <= bn; j++) {
      D[0][j] = j;
    }
  }

  public int distance() {
    for (int i = 1; i <= an; i++) {
      for (int j = 1; j <= bn; j++) {
        int d1 = D[i][j-1] + 1;
        int d2 = D[i-1][j] + 1;
        int d3 = D[i-1][j-1] + (a.charAt(i-1) == b.charAt(j-1) ? 0 : 1);
        D[i][j] = Math.min(Math.min(d1, d2), d3);
      }
    }
    return D[an][bn];
```

```
    }
  }
```

Bemerkung: In Java gibt `a.charAt(i-1)` das i-te Zeichen des Strings `a` zurück. Dadurch ergibt sich eine Diskrepanz zu der OPT-Formel.

Die volle Matrix `D` aller Zwischenergebnisse für unser Beispiel sieht folgendermaßen aus:

		D	**A**	**G**	**O**	**B**	**E**	**R**	**T**
	0	1	2	3	4	5	6	7	8
A	1	1	1	2	3	4	5	6	7
L	2	2	2	2	3	4	5	6	7
G	3	3	3	2	3	4	5	6	7
O	4	4	4	3	2	3	4	5	6
D	5	4	5	4	3	3	4	5	6
A	6	5	4	5	4	4	4	5	6
T	7	6	5	5	5	5	5	5	**5**

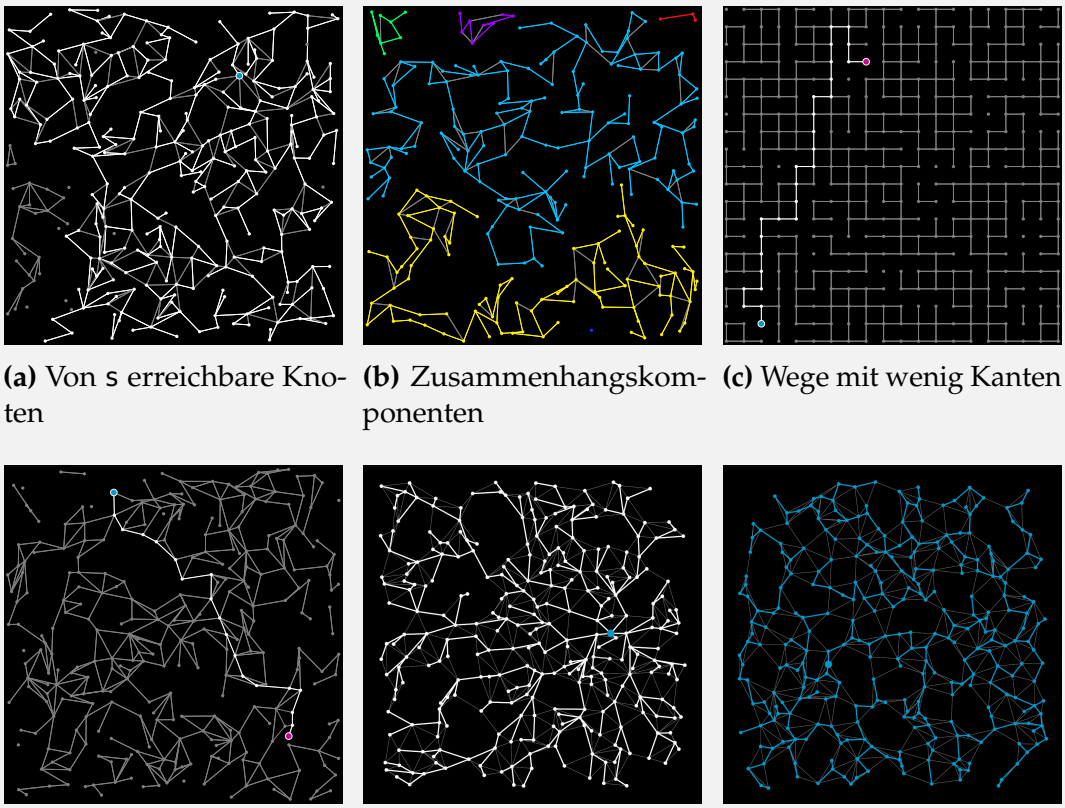
Quellenverweise

Bei der Darstellung vom *weighted interval scheduling* und dem Rucksack Problem habe ich einige Ideen von den großartigen Folien von Kevin Wayne zu seiner Vorlesung *Theory of Algorithms* (COS 423, Princeton University 2013) aufgenommen. Seine Vorlesung orientiert sich seinerseits an dem Buch von Kleinberg & Tardos. Die Besprechung der pseudopolynomiellen Laufzeit hatte [Röglin, 2011, S. 21f] und die Editierdistanz [Schöning, 2001, S. 165f] als Vorlage. Die Idee zu Abbildung 8.4 stammt aus [Erickson, 2019].

KAPITEL 9

Ungewichtete Graphen

Während die vorangegangen Kapitel allgemeine algorithmische Konzepte besprochen haben, geht es in diesem und den drei folgenden Kapiteln um konkrete Algorithmen im Zusammenhang mit Graphen. Graphen sind eine fundamentale Datenstruktur. Sie werden in sehr vielen Anwendungsbereichen der Informatik zur Repräsentation verwendet, z. B. von Streckennetzen in der Navigation, von Bildern in der Bildverarbeitung, sowie für kausale und zeitliche Abläufe. Algorithmen, die typische Aufgaben im Zusammenhang mit Graphen effizient lösen, haben einen großen Verbreitungsgrad. Ein paar der Graphenaufgaben, die in diesem Skript besprochen werden, sind in Abbildung 9.1 dargestellt.



(a) Von s erreichbare Knoten **(b)** Zusammenhangskomponenten **(c)** Wege mit wenig Kanten

(d) kürzester Weg **(e)** kürzeste Wege **(f)** Minimaler Spannbaum

Abbildung 9.1 Typische Aufgaben für Graphenalgorithmen: (a) bis **(c)** beziehen sich auf ungewichtete Graphen und **(d)** bis **(f)** auf Kanten-gewichtete Graphen. In **(d)** wird der kürzeste Weg zwischen zwei Knoten gesucht und in **(e)** die kürzesten Wege von einem Startknoten zu allen andere (*single source shortest paths*).

9.1 Graphen - Einführung der Grundbegriffe

Ein **Graph** bezeichnet eine Vernetzung von Knotenpunkten durch Verbindungen. Die meisten Begriffe rund um Graphen lassen sich intuitiv verstehen. Sie sind in Abbildung 9.2 dargestellt. Es folgen die formalen Definitionen.

Ein **Graph** ist ein Paar (Tupel) $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$. Dabei ist $\boldsymbol{V}$ eine endliche Menge von **Knoten** (*nodes*), bzw. **Ecken** (*vertices*) und $\boldsymbol{E}$ eine Menge von Kanten (*edges*). Um die Verwechslungsgefahr bei den Variablen zu reduzieren, verwenden wir nicht den Begriff Ecken (die nicht(!) $\boldsymbol{E}$ sind), sondern Knoten. Jede **Kante** in $\boldsymbol{E}$ verbindet zwei Knoten $e = (v, w)$ mit $v, w \in \boldsymbol{V}$. Man nennt dann die Knoten v und w **adjazent** (*adjacent*), wenn sie durch eine Kante verbunden sind. Der Knoten v heißt **inzident** (*incident*) mit der Kante e und ebenso heißt e inzident mit v.

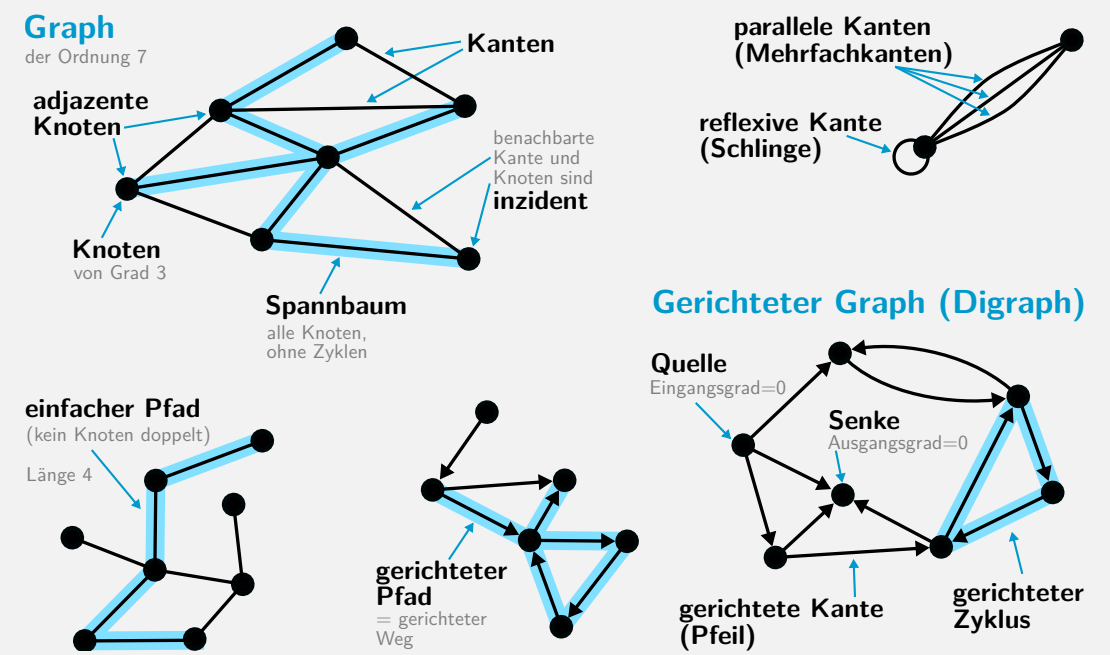


Abbildung 9.2 Grundbegriffe rund um ungewichtete Graphen

Ebenso ist natürlich w inzident mit e. Wir verwenden $V = |\boldsymbol{V}|$ und $E = |\boldsymbol{E}|$ (ohne Fettdruck) für die Anzahl der Knoten bzw. Kanten. In den Implementierungen verwenden wir natürliche Zahlen als Knoten, normalerweise: $\boldsymbol{V} = \{0, \ldots, V - 1\}$.
Ein Graph $\boldsymbol{G}' = (\boldsymbol{V}', \boldsymbol{E}')$ ist ein **Teilgraph** von $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, wenn seine Knoten und Kanten Teilmenge von den Knoten und Kanten von $\boldsymbol{G}$ sind, also $\boldsymbol{V}' \subseteq \boldsymbol{V}$ und $\boldsymbol{E}' \subseteq \boldsymbol{E}$ gilt. In einem 'normalen' (ungerichteten) Graphen haben die Kanten keine Richtung: $(v, w) \equiv (w, v)$. Reflexive Kanten (v, v) sind zugelassen. Sie werden auch **Schlinge** oder **Schleife** (*self-loop*) genannt.

Gerichtete Graphen (Digraphen)

Bei einer Kante $e = (v, w)$ in einem **gerichteten Graphen** (*directed graph*), auch **Digraphen** (*digraph*), heißt v Anfangknoten (*head*) und w Endknoten (*tail*). Auch hier sind reflexive Kanten (v, v) zugelassen. (Es gibt allerdings auch Definitionen von Digraphen, die reflexive Kanten ausschließen.) In einem gerichteten Graphen werden die Kanten **gerichtete Kanten** oder **Pfeile** (*arcs*) genannt. Wir schreiben $v \to w$ für die gerichtete Kante von v nach w, wenn hervorgehoben werden soll, dass es eine *gerichtete* Kante ist.
Ein gerichteter Digraph, der keine Zyklen (siehe nächster Abschnitt) besitzt, wird **zyklenfreier Digraph** (*directed acyclic graph*, DAG) genannt.

Pfade (Wege) und Zyklen

Ein **Pfad** (*path*) oder **Weg** in einem Graphen ist eine Folge von Knoten, die durch Kanten verbunden sind. In einem **einfachen** Pfad kommt jeder Knoten nur einmal vor. Ein **Zyklus** (*cycle*) ist ein Pfad, bei dem Anfangs- und Endknoten übereinstimmen. Bei gerichteten Graphen benutzt man auch die Begriffe **gerichteter Pfad** (*directed path*) und **gerichteter Zyklus**. In einem **einfachen Zyklus** kommen alle Knoten nur einmal vor, bis auf die Übereinstimmung von Anfangs- und Endknoten. Ein Graph, der keine Zyklen enthält, heißt **azyklisch**. Die **Länge** eines Pfads bzw. Zyklus' ist die Anzahl der Kanten. Ein Pfad in $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ von einem Startknoten s zu einem Zielknoten t ist eine Sequenz $v_0, \ldots, v_K$ mit $s = v_0$, $t = v_K$ und $(v_k, v_{k+1}) \in \boldsymbol{E}$ für alle $k < K$.

Weitere Definitionen

Mit der **Ordnung** eines Graphen bezeichnet man die Anzahl seiner Knoten. Der **Grad eines Knotens** in einem Graphen ist die Anzahl der Kanten, die ihn verbinden. Bei einem Digraphen unterscheidet man **Eingangsgrad** (Anzahl der ankommenden Pfeile) und **Ausgangsgrad** (Anzahl der ausgehenden Pfeile). Eine **Quelle** ist ein Knoten mit Eingangsgrad 0 und Ausgangsgrad > 0. Eine **Senke** ist ein Knoten mit Ausgangsgrad 0 und Eingangsgrad > 0. Zwei Kanten heißen **parallel**, wenn sie dieselben beiden Knoten verbinden. Solche Kanten werden auch **Mehrfachkanten** genannt. Bei einem Digraphen muss auch die Richtung übereinstimmen.

Zusammenhang

Ein Graph heißt **zusammenhängend** (*connected*), wenn es zwischen zwei beliebigen Knoten einen Weg gibt. Ein nicht-zusammenhängender Graph besteht aus mehreren **Zusammenhangskomponenten** (*connected components*). Knoten s und t eines Digraphen heißen **stark verbunden**, wenn es zwischen zwei beliebigen Knoten Pfade in beiden Richtungen gibt. Der Pfad von s nach t und der von t nach s darf über dieselben Knoten laufen, wenn es Pfeile in der entsprechenden Richtung gibt. Einen *Digraphen* nennt man **stark zusammenhängend** (*strong connected*), wenn alle seine Knoten stark verbunden sind. Die Eigenschaft *stark verbunden* ist eine Äquivalenzrelation. Daraus folgt: Ein gerichteter Graph, der nicht stark zusammenhängend ist, besteht aus mehreren **starken Zusammenhangskomponenten**.

Graphen und Bäume

Ein azyklischer, zusammenhängender Graph ist ein **Baum**, wie in Abschnitt 4.5 definiert. Eine Menge von Bäumen, die keine gemeinsamen Knoten haben, erge-

ben einen **Wald**. Ein **Spannbaum** eines Graph $\boldsymbol{G}$ ist ein Teilgraph, der ein Baum ist und alle Knoten enthält. Falls $\boldsymbol{G}$ selbst schon ein Baum ist, so ist er selbst sein einziger Spannbaum. Andernfalls hat jeder Spannbaum von $\boldsymbol{G}$ weniger Kanten als $\boldsymbol{G}$.

Dick und dünn

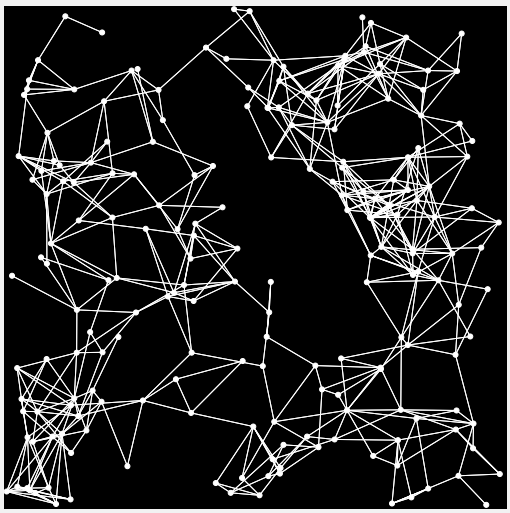
Die maximale Anzahl von Kanten E in einem Graphen $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ ohne parallele Kanten ist in der Größenordnung[1] V^2. Man nennt einen Graphen **dünn** (*sparse*), wenn $E \ll V^2$, andernfalls **dicht** (*dense*), siehe Abbildung 9.3. Diese Einteilung hat offensichtlich eine Grauzone. Es ist trotzdem wichtig, sich über die Dichte von Graphen Gedanken zu machen, aus dem folgenden Grund: Die zu erwartende Dichte ist relevant für die Auswahl von Algorithmen und Datenstrukturen in Bezug auf Laufzeit und Speicherbedarf. Für dünne Graphen ist die Wachstumsordnung $\boldsymbol{O}(E)$ deutlich besser als $\boldsymbol{O}(V^2)$ und dies gilt auch in komplexeren Wachstumsordnungen, die später bei Graphenalgorithmen vorkommen werden. Ein Beispiel: In einer konkreten Anwendung ist zu erwarten, dass jeder Knoten mit maximal 5 anderen Knoten verbunden ist (dünner Graph, und konstanter Verzweigungsfaktor). Dann gilt $E \leq 5V$, also $\boldsymbol{O}(V) = \boldsymbol{O}(E)$, da konstante Faktoren bei der Wachstumsordnung irrelevant sind. In diesem Fall ist ein Algorithmus mit einer Laufzeit in $\boldsymbol{O}(E)$ einem Algorithmus mit Laufzeit in $\boldsymbol{O}(V \log V)$ vorzuziehen ($\boldsymbol{O}(V)$ ist besser als $\boldsymbol{O}(V \log V)$). Für eine Anwendung, in der dichte Graphen erwartet werden, also $\boldsymbol{O}(E) = \boldsymbol{O}(V^2)$ gilt, ist allerdings der zweite Algorithmus effizienter, da $\boldsymbol{O}(V^2)$ schlechter ist als $\boldsymbol{O}(V \log V)$.

APIs für Graphen

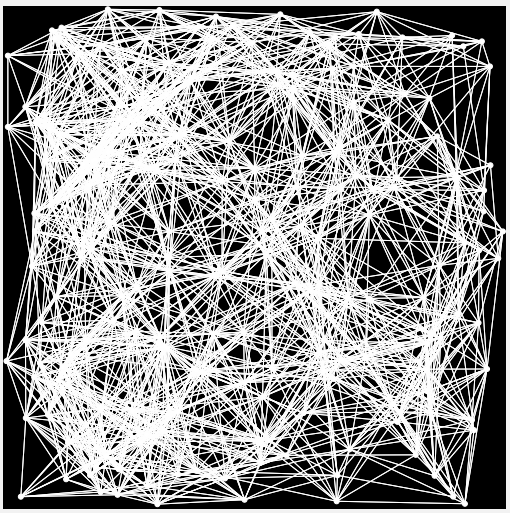
Bei der Implementation von Graphen übernehmen wir die Grundelemente (und mehr) aus [Sedgewick and Wayne, 2014]. Die Datenstrukturen sind nicht sehr flexibel insofern die Knoten statisch sind. Es können keine Knoten gelöscht oder eingefügt werden. Diese Funktionalität wird bei den hier besprochenen Algorithmen nicht benötigt und die hier gewählten Wege ermöglichen besonders kurzen und verständlichen Code. Im Allgemeinen sollte man eine Implementation verwenden, in der Knoten durch eine eigene Datenstruktur repräsentiert werden und die Speicherung im Graphen dynamische Änderungen zulässt.

Für Graphen und Digraphen verwenden wir die folgenden APIs:

[1] Bei gerichteten Graphen ist die Anzahl inklusive reflexiver Kanten genau V^2, bei ungerichteten Graphen $V(V+1)/2$ und ohne reflexive Kanten jeweils V weniger.



(a) Dünner Graph, $V = 200$, $E = 500$



(b) Dichter Graph, $V = 200$, $E = 2000$

Abbildung 9.3 Ein dünner und ein dichter Graph

API für einen ungerichteten Graphen

public class Graph

	`Graph(int V)`	Erzeugt leeren Graphen mit `V` Knoten
`int`	`V()`	Anzahl der Knoten
`int`	`E()`	Anzahl der Kanten
`void`	`addEdge(int v, int w)`	Füge Kante `(v,w)` zum Graphen hinzu
`Iterable<Integer>`	`adj(int v)`	Knoten adjazent zu `v`

API für einen Digraphen

public class Digraph

	`Digraph(int V)`	Erzeugt leeren Digraphen mit `V` Knoten
`int`	`V()`	Anzahl der Knoten
`int`	`E()`	Anzahl der Kanten
`void`	`addEdge(int v, int w)`	Füge Kante `v->w` zum Graphen hinzu
`Iterable<Integer>`	`adj(int v)`	Knoten adjazent zu `v`

Wie in der Java Einführung erwähnt, bringt die Kapselung durch Datenabstraktion unter anderem den Vorteil, dass die Anwendungs- und die Entwicklungssicht getrennt sind. Die obigen APIs liefern die vollständige Information, die für die Anwenderin oder den Anwender relevant ist. Sie/er braucht die Details der Datenstrukturen und Algorithmen nicht zu kennen. Wir befinden uns hier in der Rolle der Entwickler und müssen abwägen, welche Datenstrukturen zur Repräsentation von Graphen geeignet sind.

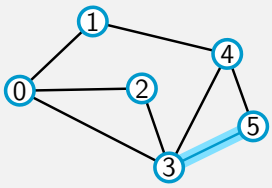
Bei der Implementierung von Graphen ist eine effiziente Speicherung der Daten, insbesondere der Kanten E, wichtig. Hier bezieht sich Effizienz auf Speicherbedarf und Laufzeit der Graphenalgorithmen.
Es gibt unterschiedliche Möglichkeiten, Graphen als Datenstruktur zu implementieren. Welche Variante die sinnvollste ist, hängt von dem Anwendungsfall ab, also von der Art des Graphen und ggf. von den Algorithmen, die zum Einsatz kommen sollen. Wir besprechen die Hauptvarianten:

- **Adjazenzmatrix**
- **Adjazenzlisten**
- **Doppelt verkettete Pfeillisten**

Für Graphen mit anderen Objekten als `int` an den Knoten verwendet man zusätzlich ein knotenindiziertes Array (Symboltabelle).

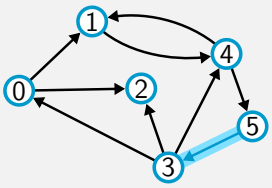
Datenstruktur Adjazenzmatrix

Bei Adjanzenzmatrizen wird die Menge der Kanten E in einer Matrix der Größe $V \times V$ mit Boole'schen Werten gespeichert, siehe Abbildung 9.4.



	0	1	2	3	4	5
0	0	1	1	1	0	0
1	1	0	0	0	1	0
2	1	0	0	1	0	0
3	1	0	1	0	1	1
4	0	1	0	1	0	1
5	0	0	0	1	1	0

(a) Ungerichteter Graph



	0	1	2	3	4	5
0	0	1	1	0	0	0
1	0	0	0	0	1	0
2	0	0	0	0	0	0
3	1	0	1	0	1	0
4	0	1	0	0	0	1
5	0	0	0	1	0	0

(b) Gerichteter Graph

Abbildung 9.4 Adjazenzmatrix zur Speicherung der Kanten: In jeder Zeile stehen die ausgehenden Kanten von dem entsprechenden Knoten. **(a)** Bei ungerichteten Graphen ist jede Kante doppelt repräsentiert, bei gerichteten Graphen **(b)** nur einfach, und zwar in der Zeile, die zu dem Startknoten korrespondiert.

Bei ungerichteten Graphen ist jede (nicht-reflexive) Kante doppelt gespeichert. Die Matrix ist in diesem Fall also symmetrisch.

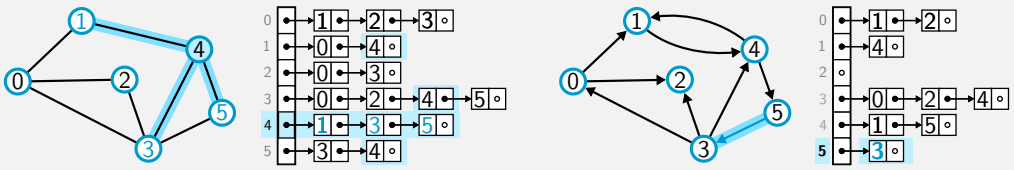
Adjazenzmatrix zur Speicherung der Kanten eines Graphen

Bei Adjazenzmatrizen ist der **Speicherbedarf** $\Theta(V^2)$, unabhängig von der Anzahl der Kanten. Die **Laufzeit** beträgt $O(1)$ zum Prüfen, ob v und w adjazent sind und $O(V)$ zum Iterieren über die Nachbarknoten von v.

Daher ist eine Implementierung von Graphen mit einer Adjazenzmatrix für dichte Graphen geeignet, für dünne jedoch ineffizient im Speicherbedarf, und meist auch in Bezug auf Laufzeit.

Datenstruktur Adjazenzlisten

Bei Adjazenzlisten werden für jeden Knoten $v \in V$ alle adjazenten Knoten in einer Liste `adj[v]` gespeichert, siehe Abbildung 9.5. Bei gerichteten Graphen werden nur die ausgehenden Pfeile berücksichtigt.



(a) Ungerichteter Graph **(b)** Gerichteter Graph

Abbildung 9.5 Adjazenzlisten zur Speicherung von Kanten: In einem Array der Länge V werden Listen mit den Nachbarknoten gespeichert. **(a)** Die Liste mit Index 4 zeigt, dass es Kanten von Knoten **4** zu den Knoten **1**, **3** und **5** gibt. Diese Kanten sind allerdings auch in den Adjazenzlisten jener Knoten gespeichert. **(b)** Bei gerichteten Graphen wird jede Kante nur einmal repräsentiert und zwar in der Liste, die zu dem Startknoten korrespondiert.

Jeder Eintrag in der Liste enthält den Zielknoten und einen Zeiger auf den folgenden Listeneintrag. Der letzte Eintrag enthält als Zeiger `null`. Die Reihenfolge in der Liste ist beliebig. Bei ungerichteten Graphen ist jede (nicht-reflexive) Kante doppelt gespeichert.

Adjazenzlisten zur Speicherung der Kanten eines Graphen

Der **Speicherbedarf** von Adjazenzlisten ist in $\boldsymbol{\Theta}(V + E)$. Die **Laufzeit** $\boldsymbol{O}(grad(v))$ zum Prüfen, ob v und w adjazent sind und $\boldsymbol{O}(grad(v))$ zum Iterieren über die Nachbarknoten von v.

Für Graphen mit *wenigen Kanten* sind Adjazenzlisten für viele Aufgaben effizient, da die Iteration über Nachbarknoten schnell ist. Weniger effizient für das Entfernen und Hinzufügen von **Knoten**.

Datenstruktur: doppelt verkettete Pfeillisten

Für Graphen, bei denen dynamisch Knoten hinzugefügt und entfernt werden können, bieten sich **doppelt verkettete Pfeillisten** (*doubly connected arc list; DCAL*) an. Dabei werden die Knoten als doppelt verkettete Listen gespeichert. Jedes Listenelement enthält einen Zeiger auf seinen Vorgänger, einen auf seinen Nachfolger sowie einen Zeiger auf eine Pfeilliste.
Jedes Element der Pfeilliste enthält drei Einträge:

- Rückwärtsverkettung: Zeiger auf die vorige Kante, bzw. auf das Knotenelement
- Zeiger zum Endknoten des Pfeils
- Vorwärtsverkettung: Zeiger auf die nächste Kante, bzw. auf`null`

Übersicht der Datenstrukturen für Graphen

- **Adjazenzmatrix**: ggf. geeignet für dichte statische Graphen
- **Adjazenzlisten**: gut geeignet für dünne statische Graphen
- **Doppelt verkettete Pfeillisten (DCAL)**: gut geeignet für dünne dynamische Graphen

Hier ist mit dynamischem Graph gemeint, dass auch nach der Initialisierung Knoten hinzugefügt oder entfernt werden, im Gegensatz zu statischen Graphen. In diesem Skript werden Adjazenzlisten verwendet, da sie für die hier besprochenen Algorithmen die einfachste und effizienteste Variante darstellen. Es folgen die Implementierungen der Klassen `Graph` und `Digraph` aus [Sedgewick and Wayne, 2014], die für Graphenalgorithmen in diesem Skript verwendet werden.

Listing 9.1 Implementation der Klasse `Graph` aus [Sedgewick and Wayne, 2014, S. 562]. Sie verwendet eine Multimenge (`Bag`) als Datenstruktur für die Adjazenzliste. Dies soll die minimale Anforderung an Funktionalität verdeutlichen. Es könnte ebenfalls eine `LinkedList` oder andere der *Java Collections* verwendet werden.

```
public class Graph {
  private final int V;
  private int E;
  private Bag<Integer>[] adj;

  public Graph(int V) {
    this.V = V;
    adj = (Bag<Integer>[]) new Bag[V];
    for (int v = 0; v < V; v++)
      adj[v] = new Bag<Integer>();
  }
```

```
  public int V()  { return V; }
  public int E()  { return E; }

  public void addEdge(int v, int w) {
    adj[v].add(w);
    adj[w].add(v);
    E++;
  }

  public Iterable<Integer> adj(int v) {
    return adj[v];
  }
}
```

Listing 9.2 Implementation der Klasse `Digraph` aus [Sedgewick and Wayne, 2014, S. 611]

```
public class Digraph {
  private final int V;
  private int E;
  private Bag<Integer>[] adj;

  // Alles exakt wie bei Graph (außer Namen des Konstruktors und)
  // die Methode addEdge:

  public void addEdge(int v, int w) {
    adj[v].add(w);              // Kante nur in einer Richtung hinzufügen
    E++;
  }

  public Digraph reverse() {    // neue Methode nur für Digraph
    Digraph R = new Digraph(V);
    for (int v = 0 ; v < V; v++)
      for (int w : adj[v])
        R.addEdge(w, v);
    return R;
  }
}
```

9.2 Durchsuchen von Graphen

Wenn es um Graphenalgorithmen geht, ist der wichtigste Grundbaustein, den Graphen zu explorieren, also angefangen bei einem Knoten, alle anderen Knoten entlang der Kanten zu entdecken. Dieser Vorgang wird auch *Durchsuchen*

oder *Durchlaufen*, bzw. *Traversierung* (der Knoten) eines Graphen genannt. Daher nähern wir uns den Graphenalgorithmen mit der Traversierung, insbesondere der Tiefensuche und der Breitensuche. Als Erweiterungen der einfachen Traversierung von Graphen betrachten wir folgende Aufgaben in gegebenen Graphen:

- Gebe zu jedem erreichbaren Knoten einen Weg von s zurück.
- Suche den kürzesten Weg von s zu allen möglichen Zielknoten.
- Identifiziere alle Zusammenhangskomponenten von $\boldsymbol{G}$.
- Teste, ob $\boldsymbol{G}$ einen Zyklus enthält.
- Gebe die erreichbaren Knoten in speziellen Reihenfolgen zurück.

Durchsuchen (Traversierung) eines Graphen

Zu einem gegebenen Knoten s eines Graphen oder Digraphen G sollen alle von s in $\boldsymbol{G}$ erreichbaren Knoten markiert werden.

Siehe hierzu auch den Abschnitt 4.5 über Bäume und Baumdurchsuchungen.

Allgemeiner Ansatz zur Graphendurchsuchung

Wir starten zunächst mit einem ganz allgemeinen Ansatz zur Durchsuchung eines Graphen $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ ausgehend von einem Knoten s. Wir benutzen eine Menge M zum Markieren der entdeckten Knoten. Anfänglich ist nur der Startknoten s markiert. Dann werden neue Knoten markiert, die von einem bereits markierten Knoten über eine Kante des Graphen erreichbar sind. Dies wird fortgeführt, bis keine unmarkierten Knoten mehr auf diese Weise erreicht werden können:

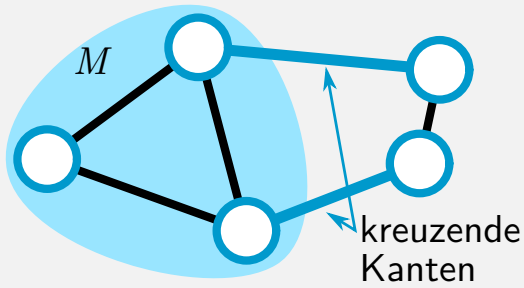
Listing 9.3 Allgemeine Graphendurchsuchung (Traversierung)

```
1 M ←{s}          // M: markierte Knoten (marked)
2 while there is (v,w) ∈ E with v ∈ M and w ∉ M do
3   add w to M
4 end
```

Die Menge der markierten Knoten M wird von $\{s\}$ also schrittweise erweitert. Dabei wird immer eine neue *kreuzende* Kante (v, w) benutzt.

Kreuzende Kante

Eine Kante (v, w) wird *kreuzende Kante* (*crossing edge*) genannt, wenn sie einen Bereich transzendiert. Hier bezieht sich der Begriff auf den markierten Bereich, d. h. auf die Menge M in Listing 9.3. Bei einer kreuzenden Kante (v, w) ist also der Startknoten v in M und der Zielknoten w außerhalb von M.



Zeile 3 wird für jeden erreichbaren Knoten einmal ausgeführt. Abgesehen von der Auswahl von v und w zur Erfüllung der **while** Bedingung ist die Laufzeit in $\boldsymbol{O}(V)$. Als kritischer Punkt bleibt also eine effiziente Wahl der nächsten Kante (v, w) zu finden. In diesem Zusammenhang ist auch eine geeignete Datenstruktur für die Menge M auszuwählen.
Schon ohne diese konkreteren Festlegungen bekommen wir das erste interessante und folgenreiche Ergebnis: Die bei der Traversierung benutzten Kanten ergeben einen Baum.

Suchbaum

Die von dem Algorithmus in Listing 9.3 benutzen Kanten (v, w), also diejenigen, die die Bedingung in Zeile 2 erfüllen, ergeben einen Baum. Er wird *Suchbaum* genannt.

Beweis. Wir betrachten den Teilgraphen von $\boldsymbol{G}$, der durch die vom Algorithmus benutzen Kanten definiert ist. Ein Baum ist ein azyklischer, zusammenhängender Graph. Da immer nur Knoten markiert werden, die von bereits markierten Knoten erreichbar sind, ist der Teilgraph offensichtlich zusammenhängend. Es bleibt zu zeigen, dass er keine Zyklen besitzt. Dazu nehmen wir an, dass es vom Algorithmus benutzte Kanten (v_k, v_{k+1}) für $k < K$ gibt, die einen Zyklus ergeben ($v_0 = v_K$) und führen diese Annahme zu einem Widerspruch.
Sei (v_i, v_{i+1}) diejenige Kante des vermeintlichen Zyklus, die bei der Graphdurchsuchung als letzte hinzugefügt wurde. Damit die Bedingung aus Zeile 2 erfüllt ist, muss $v_i \in M$ und $v_{i+1} \notin M$ gelten. Dann kann aber die von v_{i+1}

ausgehende Kante nicht vorher vom Algorithmus benutzt worden sein. (Im Fall $i = K - 1$ benutzt man die Identität $v_0 = v_K$, betrachtet also die von v_0 ausgehende Kante). □

Wenn der Graph zusammenhängend ist, also alle Knoten von s erreichbar sind, dann ist der Suchbaum ein *Spannbaum* des Graphen.
Um uns der Implementierung einer Graphendurchsuchung zu nähern, müssen wir die Strategien zur Auswahl des nächsten Knotens, der markiert werden soll, festlegen. Wir werden zwei grundlegende Ansätze bezüglich der Reihenfolge betrachten:

- Gehe rekursiv in die Tiefe. Wenn es nicht weiter geht, gehe einen Rekursionsschritt zurück und mache dasselbe für die anderen Nachbarknoten, also wie beim Backtracking. Dies führt zur **Tiefensuche**.
- Besuche zunächst alle Knoten mit Abstand 1 vom Startknoten, dann die mit Abstand 2 usw. Dies führt zur **Breitensuche**.

Zunächst überlegen wir, was überhaupt von der Suchstrategie beeinflusst wird. Die Menge der Knoten M, die der Algorithmus in Listing 9.3 zurückgibt, ist *unabhängig* von der Suchstrategie. Es sind immer alle von s in $\boldsymbol{G}$ erreichbaren Knoten.
Dagegen hängt die *Reihenfolge*, in der die Knoten in M eingefügt werden, sehr wohl von der Suchstrategie ab (und auch von der Reihenfolge in den Adjazenzlisten).
Daraus ergibt sich dieselbe Abhängigkeit für den Suchbaum. Die Menge seiner Knoten ist fix (siehe erster Punkt), aber seine Struktur (welche Knoten durch Kanten verbunden sind) ist es nicht, siehe Abbildung 9.6.

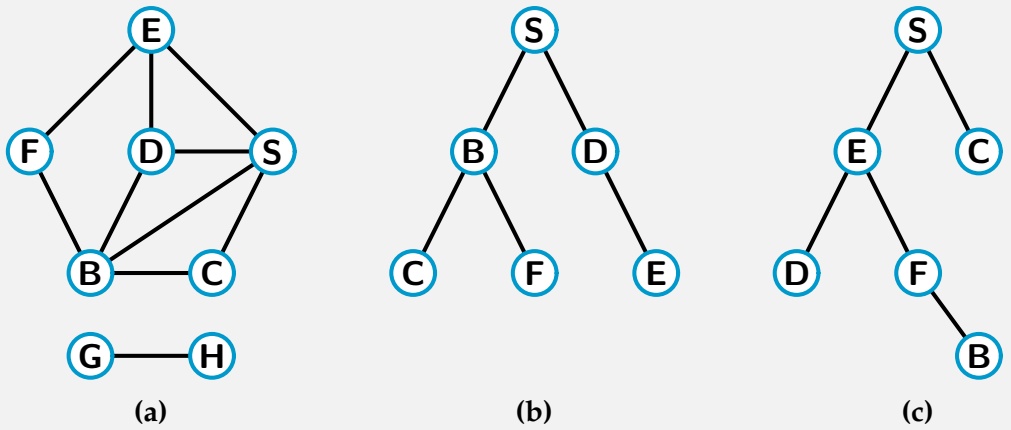


Abbildung 9.6 Die Struktur des Suchbaums: Gegeben ist der Graph in **(a)** und er wird ausgehend von Knoten **S** durchsucht. Die Struktur des Suchbaums hängt von der Suchstrategie ab. Der Suchbaum **(b)** ergibt sich, wenn die Kanten in aufsteigend alphabetischer Reihenfolge (Startknoten/Endknoten) ausgewählt werden und **(c)** bei absteigender Reihenfolge. Auch viele andere Ergebnisse sind möglich.

9.3 Tiefensuche (DFS) in Graphen

Um den Prozess der Traversierung eines Graphen zu beschreiben, werden die Begriffe *einen Knoten entdecken*, *einen Knoten besuchen* und *einen Knoten fertig bearbeitet haben* verwendet.

- Man sagt ein "Knoten wurde entdeckt", sobald der Algorithmus das erste mal auf ihn zugreift. Formal ist dies für einen Knoten w der Fall, wenn eine kreuzende Kante mit w als Zielknoten gefunden wurde. Wir sprechen hier bei kreuzenden Kanten von Start- und Zielknoten, auch wenn es um ungerichtete Graphen geht. Dann wird derjenige Knoten Startknoten genannt, der markiert ist.

- Ein "Knoten wird besucht", wenn von ihm aus die Nachbarknoten exploriert werden. Es ist also der Vorgang des Iterierens über die zu ihm inzidenten Kanten (bzw. adjazenten Knoten). Bei der Tiefensuche wird ein Knoten besucht, direkt nach dem er entdeckt wurde. Daher wird hier der Unterschied der Begriffe nicht so deutlich.

- Ein "Knoten ist fertig bearbeitet", wenn er im weiteren Verlauf des Algorithmus nicht mehr besucht wird. Bei der Tiefensuche ist dies der Fall, wenn von ihm erreichbaren Knoten besucht und alle von ihm aus entdeckten Knoten auch fertig bearbeitet sind.

Diese Begriffe sind in dem folgenden Pseudocode für die Tiefensuche mit Rekursion annotiert:

Listing 9.4 Pseudocode für Tiefensuche (DFS) in Graphen mit Rekursion

```
1 M ←∅
2 dfs(G, 0)                                  // Tiefensuche ausgehend von Knoten 0
3
4 procedure dfs(G, v)                        // besuche Knoten v
5 add v to M                                 // markiere v als entdeckt
6 for each w with (v,w) ∈ E and w ∉ M        // iteriere über kreuzende Kanten
7   dfs(G, w)                                // w ist entdeckt, folge Kante (v,w)
8 end                                        // v fertig bearbeitet
```

Der schrittweise Ablauf der Tiefensuche ist in einem kleinen Beispielgraphen in Abbildung 9.7 dargestellt.

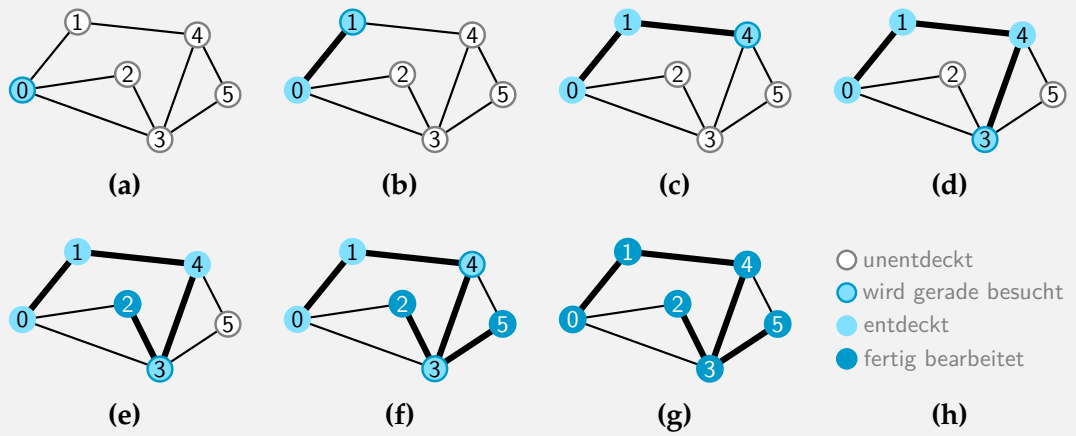


Abbildung 9.7 Ablauf der Tiefensuche in einem kleinen Beispielgraphen: Startend bei Knoten **0 (a)** wird als erstes Knoten **1** entdeckt und besucht. Von dort wird Knoten **4 (b)**, und dann Knoten **3 (d)** entdeckt. Im nächsten Schritt wird Knoten **2** besucht. Da er keine unentdeckten Nachbarn hat, ist der Knoten fertig bearbeitet **(e)**. Es geht à la Backtracking zurück zu Knoten **3**. Von dort wird Knoten **5** entdeckt und besucht. Auch er ist ohne weitere Entdeckungen fertig bearbeitet **(f)**. Nun geht es über die Knoten **3**, **4** und **1** zurück zur **0** wo dann die Tiefensuche beendet ist **(g)**.

Um nun vom Pseudocode zu einer Java Implementation zu gelangen, muss überlegt werden, welche Datenstruktur für die Markierung M am besten geeignet ist. Die folgende Funktionalität wird für die Markierung benötigt:

- Leer initialisieren (Zeile 1)
- Elemente hinzufügen (Zeile 5)
- Abfragen, ob ein Element zu M gehört (Zeile 6)

Für die Laufzeit ist besonders das Abfragen kritisch, da diese Operation am häufigsten aufgerufen wird. Ein Boole'sches Array (der Größe V) ist eine gute Wahl. So hat das Hinzufügen und das Abfragen konstante Laufzeit und die Laufzeit der Initialisierung ist in $\boldsymbol{O}(V)$. Damit kann der Pseudocode sehr einfach in eine Java Implementierung übertragen werden:

Listing 9.5 Tiefensuche: markiert die von `s` erreichbaren Knoten

```
1  public class DepthFirstTraversal {
2    private boolean[] marked;
3
4    public DepthFirstTraversal(Graph G, int s) {
5      marked = new boolean[G.V()];
6      dfs(G, s);
7    }
8
9    public void dfs(Graph G, int v) {
10     marked[v] = true;
11     for (int w : G.adj(v)) {
12       if (!marked[w]) {
13         dfs(G, w);
14       }
15     }
16   }
17 }
```

Korrektheit der Tiefensuche

Korrektheit des Programms zur Tiefensuche

Das Programm in Listing 9.5 markiert genau die von `s` aus erreichbaren Knoten.

Beweis. Es werden nur Knoten über Pfade ausgehend von `s` markiert. Also werden keine Knoten markiert, die nicht von `s` erreichbar sind.
Es bleibt zu zeigen: Wenn es einen Pfad von `s` zu einem Knoten `t` gibt, so wird `t` markiert. Nehmen wir an, `t` würde nicht markiert werden. Da `s` als Startknoten markiert ist, muss der Pfad irgendwo von dem markierten in den unmarkierten Bereich übergehen. Auf dem Pfad von `s` nach `t` muss

es somit (mindestens) eine kreuzende Kante geben, also `(v,w)`, für die `v` markiert ist, `w` aber nicht. In diesem Fall hätte die Tiefensuche nachdem `v` (in Zeile 10) markiert wurde auch `w` markiert: `w` ist in der Adjazenzliste von `v`, daher kommt unser `w` bei der Iteration in Zeile 11 tatsächlich vor. Da `w` nach unserer Annahme nicht markiert ist, ist die Bedingung in Zeile 12 erfüllt. Folglich wird in Zeile 13 `dfs(G, w)` aufgerufen und in diesem Aufruf wird `w` in Zeile 10 markiert.
So führt die Annahme zu einem Widerspruch und die Behauptung ist bewiesen. □

Die Terminierung wird von der folgenden Laufzeitbetrachtung impliziert.

Es folgt eine Betrachtung der Laufzeit der Tiefensuche. Sie wird üblicherweise in Abhängigkeit von V und E, der Anzahl der Knoten und Anzahl der Kanten angegeben. Beim Abwägen der Laufzeit für konkrete Anwendungsfälle sollte die Dichte des Graphen mit berücksichtigt werden, siehe unten.

Laufzeit und Speicherbedarf der Tiefensuche

Die Laufzeit der Tiefensuche ist für gerichtete und ungerichtete Graphen in $\boldsymbol{O}(V + E)$. Der Speicherbedarf ist in $\boldsymbol{O}(V)$.

Beweis anhand der Implementierung in Listing 9.5. Da die Methode `dfs` nur für unmarkierte Knoten aufgerufen wird, wird jeder Knoten höchstens einmal markiert. Die `for` Schleife in Zeile 11 wird folglich höchstens für jeden Knoten `v` einmal aufgerufen und läuft dann über alle seine Nachbarn `w`. Der Rumpf (Zeilen 12-13) wird also höchstens einmal für jede Kante $(v, w) \in \boldsymbol{E}$ aufgerufen.
Für ungerichtete Graphen kann jede Kante zweimal betrachtet werden, von jeder Seite einmal. Der Faktor 2 spielt allerdings für die Wachstumsordnung keine Rolle. Somit ist die Laufzeit der Methode `dfs` in $\boldsymbol{O}(E)$.
Die Initialisierung von `marked` in Zeile 5 benötigt eine Laufzeit in $\boldsymbol{O}(V)$. Insgesamt ist die Laufzeit der Tiefensuche also in $\boldsymbol{O}(V + E)$. □

Der explizit belegte Speicher ist das Feld `marked` der Größe V. Außerdem wird der Speicherplatz für den Rekursionsstack benötigt. Dabei sind die maximale Rekursionstiefe, sowie die lokalen Variablen zu beachten. Die Rekursionstiefe ist durch V begrenzt, da in der Rekursion jeweils einem einfachen Pfad gefolgt wird. Der Speicherbedarf der lokalen Variablen ist konstant. Damit ergibt sich insgesamt die Größenordnung $\boldsymbol{O}(V)$ für den Speicherbedarf. □

Die Laufzeit $\boldsymbol{O}(V+E)$ könnte man auch als $\boldsymbol{O}(\max\{V,E\})$ ausdrücken. Wenn man den Begriff dünner Graph so auffasst, dass die Anzahl der Kanten in $\boldsymbol{O}(V)$ liegt, dann ist die Wachstumsordnung der Laufzeit bei ihnen $\boldsymbol{O}(V)$. Bei beliebigen (bzw. dichten) Graphen ist die Wachstumsordnung der Laufzeit $\boldsymbol{O}(V^2)$.

Pfade von einem Ausgangsknoten zu allen anderen mit Tiefensuche finden

Das Programm markiert alle erreichbaren Knoten. Dabei benutzt es implizit Pfade von dem Startknoten zu allen anderen Knoten. Wir wollen das Programm um die Funktionalität erweitern, diese Pfade zurückzugeben:

Alle Pfade von einem Startknoten *single-source paths*

Zu einem gegeben Graphen $\boldsymbol{G}$ und Startknoten s, soll für alle Knoten v entschieden werden, ob sie von s erreichbar sind und falls ja, soll ein Pfad zurückgegeben werden.

Zunächst macht es den Eindruck, dass die Speicherung aller Pfade aufwändig ist. Dem ist glücklicherweise nicht so. Hier kommt uns das Ergebnis zu Gute, dass alle benutzten Pfade einen Baum ergeben, siehe S. 145. Daher können sie alle gemeinsam in einem Knoten-indizierten Array gespeichert werden, das von jedem Knoten auf den Elternknoten verweist.
Um daraus einen Pfad abzulesen, muss man sich vom Zielknoten zum Startknoten bewegen und die Knoten 'rückwärts' in eine Liste speichern. Diese Funktionalität soll von der Methode `pathTo()` zur Verfügung gestellt werden.

API zur Pfadsuche

interface Paths

	`Paths(Graph G, int s)`	Finde Pfade in `G` ausgehend von `s`
`boolean`	`hasPathTo(int v)`	Gibt es einen Pfad von `s` zu `v`
`Iterable<Integer>`	`pathTo(int v)`	Pfad von `s` zu `v` (oder `null`)

Listing 9.6 Pfade mit Tiefensuche finden aus [Sedgewick and Wayne, 2014, S. 567]

```
public class DepthFirstPaths {
  private boolean[] marked;
  private int[] parent;
  private final int s;

  public DepthFirstPaths(Graph G, int s) {
    marked = new boolean[G.V()];
    parent = new int[G.V()];
    this.s = s;
```

```
      dfs(G, s);
    }

    public void dfs(Graph G, int v) {
      marked[v] = true;
      for (int w : G.adj(v)) {
        if (!marked[w]) {
          parent[w] = v;
          dfs(G, w);
        }
      }
    }

    public boolean hasPathTo(int v) {
      return marked[v];
    }

    // Um den Pfad von s nach v zurückzugeben, geht man von v solange zum
    // Vorgänger bis man bei s ankommt. Durch Benutzung eines Stapels gibt
    // ein Abruf mit pop() den Pfad in richtiger Reihenfolge zurück.
    public Iterable<Integer> pathTo(int v) {
      if (!hasPathTo(v))
        return null;
      Stack<Integer> path = new Stack<>();
      for (int w = v; w != s; w = parent[w]) {
        path.push(w);
      }
      path.push(s);
      return path;
    }
  }
```

In Abbildung 9.1 (a) sind die mit Tiefensuche entdeckten Pfade in einem Zufallsgraphen ausgehend von einem Startknoten abgebildet.

Zusammenhangskomponenten

Mit einer anderen kleinen Erweiterung kann mit Tiefensuche die folgende Aufgabe gelöst werden:

Graphen in Zusammenhangskomponenten zerlegen

Ein gegebener Graph soll in seine Zusammenhangskomponenten zerlegt werden. Dazu soll jedem Knoten die Nummer seiner Komponente zugeordnet werden.

Diese Aufgabe kann z. B. folgendermaßen implementiert werden. Durch Tiefensuche mit einem beliebigen Startknoten werden alle erreichbaren Knoten

mit ID #1 gekennzeichnet. Dann wird iterativ die ID erhöht und der Vorgang wiederholt, ausgehend von dem nächsten Knoten, der noch ohne ID ist. In Abbildung 9.1 (b) sind die Zusammenhangskomponenten in einem Zufallsgraphen mit unterschiedlichen Farben markiert.
Die Zerlegung in *starke Zusammenhangskomponenten* in einem *gerichteten* Graphen ist deutlich schwieriger.

Zusammenhangskomponenten mit Tiefensuche identifizieren

In der folgenden Implementierung übernimmt `id` auch die Rolle von `marked`. Der `id` Wert 0 zeigt an, dass der Knoten 0 nicht markiert ist. Die Nummerierung der Zusammenhangskomponenten `id` beginnt also ab 1.

Listing 9.7 Zusammenhangskomponenten mit Tiefensuche identifizieren aus [Sedgewick and Wayne, 2014, S. 584]

```
public class ConnectedComponents {
  private int count;
  private int[] id;

  public ConnectedComponents(Graph G) {
    id = new int[G.V()];
    for (int v = 0; v < G.V(); v++) {
      if (id[v] == 0) {
        count++;
        dfs(G, v);
      }
    }
  }

  public void dfs(Graph G, int v) {
    id[v] = count;
    for (int w : G.adj(v)) {
      if (id[w] == 0)
        dfs(G, w);
    }
  }

  public int count() { return count; }

  public int id(int v) { return id[v]; }

  public boolean connected(int v, int w)
  { return id[v] == id[w]; }
}
```

Graph auf Zyklen prüfen

Mit einer anderen kleinen Änderung kann man mit Tiefensuche prüfen, ob ein ungerichteter Graph Zyklen enthält. Diese Funktion wird später in anderen Algorithmen verwendet.

Zykluserkennung

Zu einem gegebenen Graphen (ohne Schleifen und Mehrfachkanten) soll entschieden werden, ob er einen Zyklus enthält.

Im Prinzip erkennt man einen Zyklus daran, dass die Tiefensuche auf einen schon markierten Knoten trifft. Allerdings sind bei einem ungerichteten Graphen alle Kanten doppelt vorhanden. Daher muss man prüfen, ob der markierte Knoten nicht derjenige ist, von dem man gekommen ist. Für einen gerichteten Graphen ist die Prüfung auf gerichtete Zyklen etwas aufwändiger.

Listing 9.8 Zyklen in ungerichteten Graphen mit Tiefensuche erkennen, aus [Sedgewick and Wayne, 2014, S. 584]

```
1  public class Cycle {
2    private boolean[] marked;
3    private int[] parent;
4    public boolean hasCycle;
5
6    public Cycle(Graph G) {
7      marked = new boolean[G.V()];
8      parent = new int[G.V()];
9      for (int v = 0; v < G.V(); v++) {
10       if (!marked[v]) {
11         dfs(G, v);
12       }
13     }
14   }
15
16   public void dfs(Graph G, int v) {
17     marked[v] = true;
18     for (int w : G.adj(v))
19       if (!marked[w]) {
20         parent[w] = v;
21         dfs(G, w);
22       } else if (parent[v] != w) {
23         hasCycle = true;
24       }
25   }
26 }
```

Um den gefundenen Zyklus zurückzugeben, kann der folgende Code die Zeile 23 in dem `else if` Zweig von `dfs()` ersetzen:

```
cycle = new Stack<Integer>();
  cycle.push(w);
  for (int u = v; u != w; u = parent[u]) {
    cycle.push(u);
  }
  cycle.push(w);
```

In diesem Fall sollte die Bedingung der `if` Anweisung in Zeile 19 durch `&& cycle != null` ergänzt werden und `cycle` als Objektvariable deklariert werden. Die Variable `hasCycle` wird überflüssig, da dies durch `cycle != null` geprüft werden kann.

9.4 Breitensuche in Graphen

Die Breitensuche durchläuft den Graphen schichtweise. Zunächst werden alle Knoten besucht, die den Abstand 1 zum Startknoten s haben, dann die Knoten mit Abstand 2 usw. So lassen sich alle erreichbaren Knoten in Bezug auf ihren Abstand zu s klassifizieren. Dabei ist der Abstand zweier Knoten definiert als die geringste Anzahl von Kanten eines Weges zwischen den Knoten.
Die Implementierung geht folgendermaßen vor: Ausgehend von s werden alle Nachbarknoten, die noch nicht markiert sind, markiert und sie in eine Warteschlange geschoben. Dieser Prozess wird wiederholt mit den Knoten, die der Warteschlange entnommen werden, bis sie leer ist.

Listing 9.9 Breitensuche in Pseudocode

```
1  // Q: Queue, M: Array der Größe V
2  s ← 0
3  M ←{s}
4  Enqueue(Q, s)
5  while Q ≠ ∅ do
6    v ← Dequeue(Q)          // v wird besucht
7    for each w with (v,w) ∈ E and w ∉ M
8      add w to M            // w wurde entdeckt
9      Enqueue(Q, w)
10   end                     // v fertig bearbeitet
11 end
```

Die Begriffe 'entdecken', 'besuchen', und 'fertig bearbeiten' wurden bei der Tiefensuche eingeführt. Wenn bei der Breitensuche ein Knoten besucht wird,

werden von ihm aus zunächst alle Nachbarknoten entdeckt (die vorher noch nicht entdeckt waren) und damit ist der Knoten fertig bearbeitet. Dann geht es bei dem nächsten Knoten weiter. Der schrittweise Ablauf der Breitensuche ist in einem kleinen Beispielgraphen in Abbildung 9.7 dargestellt.

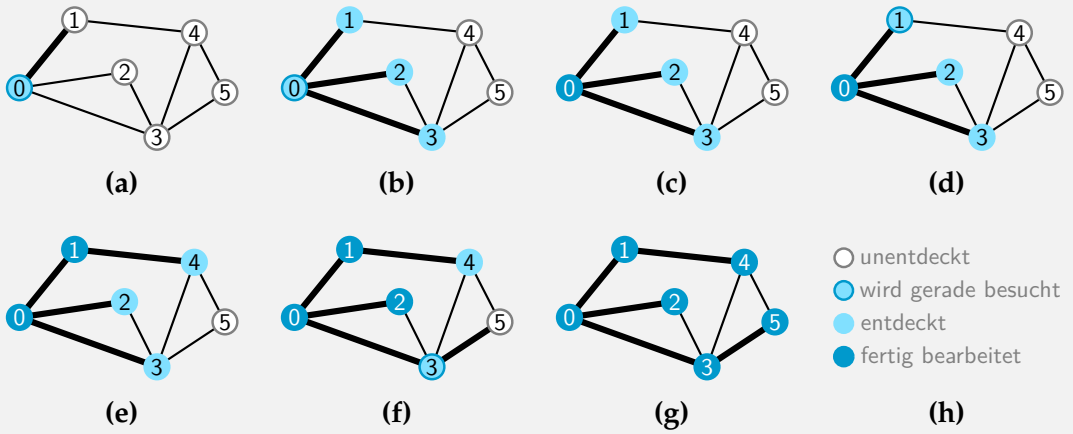


Abbildung 9.8 Ablauf der Breitensuche in einem kleinen Beispielgraphen: Zunächst wird der Startknoten (hier **0**) besucht. Von ihm aus wird Knoten **1** entdeckt **(a)**. Nun wird allerdings nicht wie bei der Tiefensuche Knoten **1** besucht, sondern es geht weiter mit dem Entdecken von Knoten **0** aus. Es werden Knoten **2** und dann **3** entdeckt **(b)**. Im Programmablauf werden sie in die Warteschlange eingefügt. Damit ist Knoten **0** fertig bearbeitet**(c)**. Als nächstes wird Knoten **1** besucht **(d)**, da es derjenige Knoten ist, der am längsten in der Warteschlage ist. Von ihm aus wird Knoten **4** entdeckt **(e)**. Als nächstes wird Knoten **2** besucht. Da er kein unentdeckten Nachbarn besitzt, ist **2** damit fertig bearbeitet **(f)**. Es folgt Knoten **3**. Von ihm aus wird der letzte Knoten **5** entdeckt. Anschließend werden die Knoten **4** und **5** besucht und sind ohne weitere Entdeckungen fertig bearbeitet **(g)**.

Terminierung des Algorithmus zur Breitensuche

Der Algorithmus in Listing 9.9 terminiert in endlicher Laufzeit.

Die *for* Schleife in Zeile 7 iteriert über eine feste, endliche Menge von Knoten. Die *while* Schleife läuft, solange die Warteschlange nicht leer ist. In der Schleife wird als erstes ein Element entfernt. Da jeder Knoten höchstens einmal markiert wird und der Algorithmus nur bei der Markierung Knoten in die Warteschlange einfügt, terminiert der Algorithmus. □

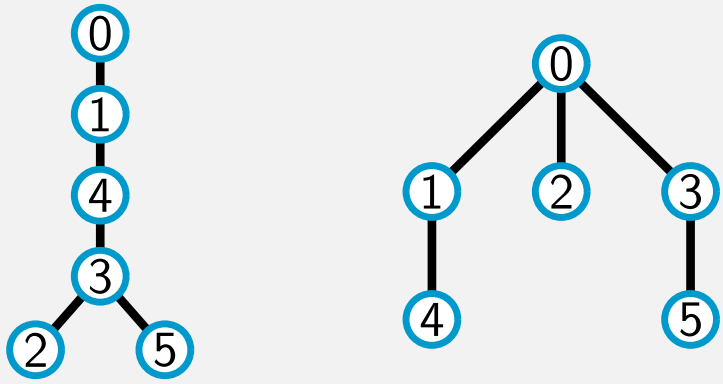
Laufzeit und Speicherbedarf der Breitensuche

Die Laufzeit der Breitensuche ist für gerichtete und ungerichtete Graphen in $\boldsymbol{O}(V + E)$. Der Speicherbedarf ist in $\boldsymbol{O}(V)$.

Beweis anhand des Algorithmus in Listing 9.9. Jeder Knoten kann nur einmal zu M hinzugefügt werden, denn Zeile 8 wird nur ausgeführt, wenn $w \notin M$ erfüllt ist. Also wird jeder Knoten höchstens einmal in die Schlange Q eingefügt und einmal herausgeholt. Die *for* Schleife in Zeile 7 wird also für jeden Knoten v höchstens einmal aufgerufen. Für jeden Knoten v werden alle benachbarten Knoten w durchlaufen und auf $w \notin M$ geprüft. Dies geschieht für jede Kante nur einmal, also ist die Laufzeit dieses Programmteils in $\boldsymbol{O}(E)$. Für ungerichtete Graphen kann jede Kante zweimal betrachtet werden, von jeder Seite einmal. Der Faktor 2 spielt allerdings für die Wachstumsordnung keine Rolle.
Die Initialisierung des Arrays M hat eine Laufzeit in $\boldsymbol{O}(V)$. Insgesamt ist die Laufzeit der Breitensuche also in $\boldsymbol{O}(V + E)$. □

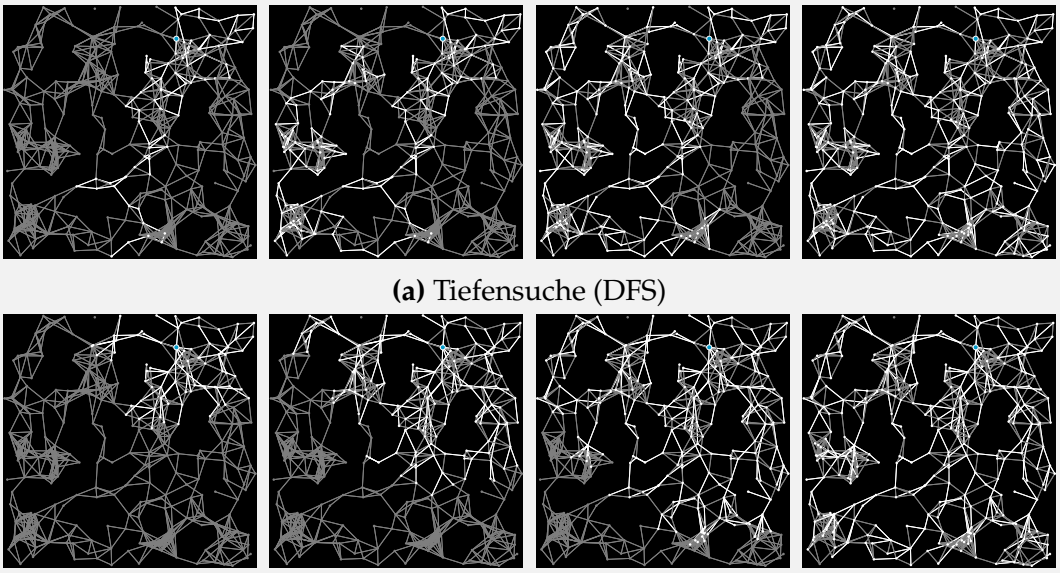
Der einzige nicht-konstante Speicherbedarf bei der Breitensuche wird von der Warteschlange benötigt. Da keine Knoten doppelt in die Warteschlange eingefügt werden, ist ihre maximale Länge V. Somit ist der Speicherbedarf der Warteschlange und des ganzen Algorithmus in $\boldsymbol{O}(V)$. □

Analog zu Listing 9.6 kann `BreadthFirstPaths` implemeniert werden, um Pfade zu allen erreichbaren Knoten von einem Startknoten `s` mit Breitensuche zurückzugeben. Wie die Reihenfolge des Entdeckens, so unterscheiden sich auch die resultierenden Suchbäume. Für den Graphen, der in den Illustrationen 9.7 und 9.8 verwendet wurden, ist der DFS- und der BFS-Suchbaum in Abbildung 9.9 dargestellt.



(a) DFS-Suchbaum **(b)** BFS-Suchbaum

Abbildung 9.9 Suchbäume mit Tiefen- und Breitensuche: (a) Suchbaum aus der Tiefensuche in Abbildung 9.7. **(b)** Suchbaum aus der Breitensuche in Abbildung 9.8.



(a) Tiefensuche (DFS)

(b) Breitensuche (BFS)

Abbildung 9.10 Vergleich von Tiefen-und Breitensuche

Pfade mit den wenigsten Kanten mit Breitensuche finden

Die Breitensuche in Algorithmus 9.9 markiert alle erreichbaren Knoten. Auf Grund der schichtweisen Durchsuchung gemäß Abstand von s, eignet sich die Breitensuche für die folgende Aufgabe:

Pfade mit minimaler Kantenanzahl von einem Startknoten

Zu einem gegebenen Graphen G und Startknoten s soll für alle erreichbaren Knoten ein Pfad mit minimaler Kantenanzahl zurückgegeben werden.

> Bei dem Begriff *kürzeste* Pfade ist Vorsicht angebracht. Dieser Begriff bekommt bei den später betrachteten *gewichteten* Graphen eine spezielle Bedeutung. Daher muss bei dem Begriff immer auf den Kontext geachtet werden (ungewichteter oder gewichteter Graph). Hier wird für ungewichtete Graphen meist der eindeutige Begriff 'Pfade mit minimaler Kantenanzahl' anstelle von 'kürzeste Pfade' verwendet.

Zum Speichern der Pfade benutzen wir dieselbe Strategie wie bei der Tiefensuche.

Listing 9.10 BFS zum Finden von Pfaden mit den wenigsten Kanten

```
1  procedure bfs(G, s)
2  //Q: Queue
3  //parent: Array of size V
4  M ←{s}
5  dist[s] ← 0
6  Enqueue(Q, s)
7  while Q ≠ ∅ do
8    v ← Dequeue(Q)
9    for each w with (v,w) ∈ E and w ∉ M
10     add w to M
11     parent[w] ← v
12     dist[w] ← dist[v] + 1
13     Enqueue(Q, w)
14   end
15 end
```

Der Pfad zu einem gegebenen Zielknoten kann genau wie bei den Pfaden des DFS-Suchbaums aus `parent` konstruiert werden, Methode `pathTo`.

Die Distanz, also die Anzahl der Kanten im kürzesten Weg von `s` zu `v` wird in `dist[v]` gespeichert.

Korrektheit der Breitensuche

Korrektheit der Breitensuche für Pfade mit wenigsten Kanten

Der Algorithmus zur Breitensuche in Listing 9.10 bestimmt zu allen erreichbaren Knoten den Pfad von `s` mit den wenigsten Kanten.

Beweis. In die Warteschlange kommen zunächst alle Knoten mit Abstand 1 vom Startknoten. Der Beweis erfolgt durch Induktion. Wenn in der Warteschlange alle Knoten mit Abstand n von `s` sind, werden bei der Abarbeitung der Warteschlange alle Knoten mit Abstand $n+1$ in die Schlange eingereiht. Durch die Entdeckung der Knoten in aufsteigender Reihenfolge ihres Abstands von `s`, kann kein Knoten über einen längeren Weg entdeckt werden. □

Die Terminierung wird von der Laufzeitbetrachtung für Breitensuche impliziert.

Gegenüberstellung von Tiefensuche und Breitensuche

Sowohl bei Tiefen- als auch bei Breitensuche werden vom Startknoten ausgehend erreichbare Knoten markiert. Dabei wird immer ein Knoten über eine kreuzende Kante ausgewählt, also über eine Kante, die den markierten Bereich transzendiert.
Tiefensuche fährt bei frisch entdeckten Knoten fort, während Breitensuche die schon am längsten entdeckten Knoten auswählt. Dasselbe Grundverfahren mit anderen Auswahlvarianten für die kreuzende Kante wird uns in den nächsten Kapiteln begegnen.
Beide Verfahren markieren alle erreichbare Knoten mit einer Laufzeit in $\boldsymbol{O}(V+E)$ ud Speicherbedarf in $\boldsymbol{O}(V)$. Nur die Breitensuche eignet sich zum Finden von Wegen mit den wenigsten Kanten. Mit beiden Verfahren können Zusammenhangskomponenten identifizert werden, allerdings eignet sich nur die Tiefensuche für *starke* Zusammenhangskomponenten in Digraphen (nicht Teil dieses Skriptes).

9.5 Gerichtete Graphen (Digraphen)

Die bisher besprochenen Algorithmen sind gleichermaßen für ungerichtete und gerichtete Graphen geeignet. Im Folgenden beschäftigen wir uns mit Eigenschaften und Algorithmen, die sich speziell auf *gerichtete* Graphen (Digraphen)

beziehen. Dabei eignet sich insbesondere Tiefensuche dazu, Strukturinformation zu gewinnen. Insbesondere geht es darum, spezielle Sortierungen der Knoten zu bestimmen, die in späteren Algorithmen hilfreich sein werden:

- Nebenreihenfolge (*preorder*) in der Knoten entdeckt werden
- Hauptreihenfolge (*postorder*) in der Knoten fertig abgearbeitet sind

Mit dieser Strukturinformation können *gerichtete Zyklen* erkannt und Topologische Sortierungen von Digraphen hergestellt werden.

Besonderheit in gerichteten Graphen

Um einen Digraphen mit Tiefensuche zu untersuchen, werden Tiefensuchdurchläufe mit unterschiedlichen Startknoten gestartet, bis alle Knoten erreicht wurden, wie bei der Identifikation von Zusammenhangskomponenten.
Da man bei jedem Durchlauf einen Suchbaum erhält, und alle Suchbäume disjunkt sind, ist das Gesamtergebnis aller Durchläufe ein *Tiefensuchwald*. Dies ist noch wie bei ungerichteten Graphen. Bei gerichteten Graphen kann aber das Ergebnis, also der Tiefensuchwald, je nach Reihenfolge der Startknoten unterschiedlich sein, siehe Abbildung 9.11. Dies liegt daran, dass bei einem Durchlauf der Tiefensuche in gerichteten Graphen, anders als bei ungerichteten Graphen, nicht unbedingt die ganze Zusammenhangskomponente gefunden wird, sondern nur der vom Startknoten in Kantenrichtung erreichbare Teil. Nur bei *stark* zusammenhängenden Digraphen wird garantiert eine ganze Zusammenhangskomponente erreicht.

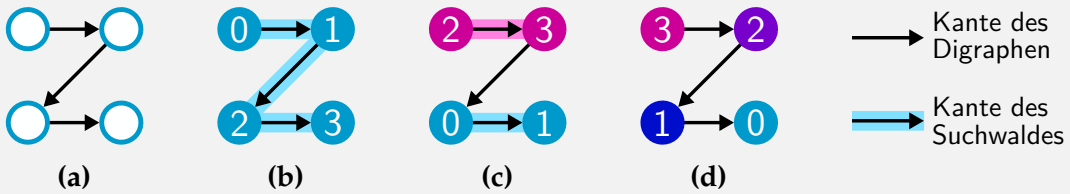


Abbildung 9.11 Variabilität beim Tiefensuchwald: Der gerichtete Graph **(a)** soll mit Tiefensuche traversiert werden. Welcher Tiefensuchwald dabei entsteht, hängt von der Reihenfolge der Knoten mit der die Durchläufe gestartet werden, ab. Hier wird die Suche immer in aufsteigender Sortierung begonnen, und wir betrachten unterschiedliche Indizierungen. **(b)** Start bei Knoten **0** erschließt den ganzen Digraphen in einem Anlauf. **(c)** In dieser Konfiguration wird von **0** aus nur die untere Hälfte des Digraphen entdeckt. Ein zweiter Durchgang startend bei Knoten **2** erschließt die obere Hälfte. **(d)** Bei einer unglücklichen Reihenfolge wird jeweils von dem Startknoten ausgehend gar kein anderer Knoten entdeckt. Hier entsteht ein Suchwald aus vier Einzelknoten.

9.5.1 Zeitstempel bei der Tiefensuche

Wir definieren bei der Tiefensuche die folgenden Zeitstempel:

- Eingangsstempel wenn der Knoten entdeckt wird (Färbung hellblau)
- Ausgangsstempel wenn der Knoten fertig abgearbeitet ist (Färbung dunkelblau)

Anhand dieser Zeitstempel werden drei Reihenfolgen der Knoten eines Digraphen definiert. Dabei zählen wir die Zeit gemeinsam für Ein- und Ausgangsstempel hoch, siehe Variable `time` in Listing 9.11.

- **Nebenreihenfolge** (*preorder*) nennt man die Sortierung der Knoten aufsteigend nach ihrem Eingangsstempel.
- **Hauptreihenfolge** (*postorder*) nennt man die Sortierung der Knoten aufsteigend nach ihrem Ausgangsstempel.
- **Umgekehrte Hauptreihenfolge** (*reverse postorder*) nennt man die Sortierung der Knoten absteigend nach ihrem Ausgangsstempel.

Listing 9.11 Java Code zur Vergabe von Zeitstempeln für alle Knoten in Digraphen bei der Tiefensuche

```
public class NodeOrder {
  public int[] dt; // discovery time (=0 for not discovered yet, i.e. unmarked)
  public int[] ft; // finalizing time
  private int time;

  public NodeOrder(Digraph G) {
    dt = new int[G.V()];
    ft = new int[G.V()];
    for (int v = 0; v < G.V(); v++) {
      if (dt[v] == 0) {        // use 'dt[v] > 0' as 'marked[v]'
        dfs(G, v);
      }
    }
  }

  public void dfs(Digraph G, int v) {
    dt[v] = ++time;
    for (int w : G.adj(v)) {
      if (dt[w] == 0) {
        dfs(G, w);
      }
    }
    ft[v] = ++time;
  }
```

```
}
```

Abbildung 9.12 stellt den Ablauf beim Generieren der Zeitstempel dar.

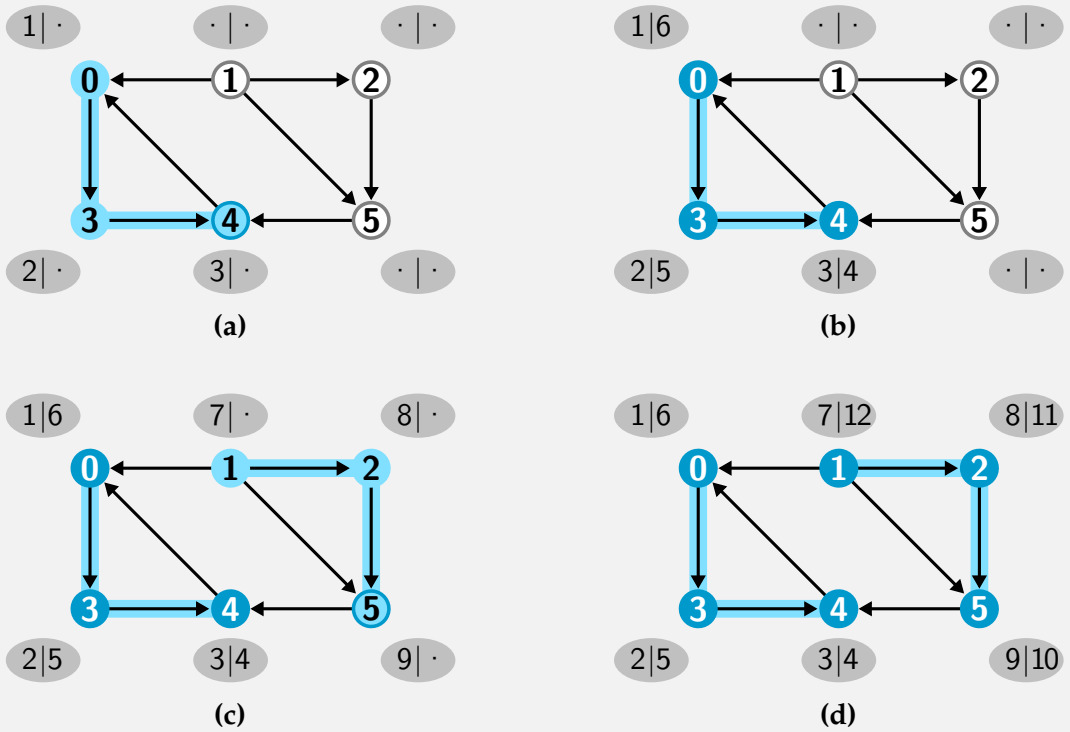


Abbildung 9.12 Beispieldurchlauf DFS mit Zeitstempeln: Die Knoten werden in aufsteigender Reihenfolge durchlaufen. **(a)** Die Knoten **0**, **3** und **4** werden entdeckt. Sie bekommen der Reihe nach die Eingangsstempel mit den Zeiten 1, 2 und 3. Bei Knoten **4** geht es nicht weiter. **(b)** Daher ist er fertig bearbeitet und erhält den Ausgangsstempel mit Zeit 4. Dann folgen Knoten **3** und **0** mit Ausgangszeiten 5 und 6. Es beginnt der zweite DFS Durchlauf bei Knoten **1**. **(c)** Er bekommt den Eingangsstempel 7 und es folgen die Knoten **2** und **5** mit den Eingangzeiten 8 und 9. Damit ist Knoten **5** fertig bearbeitet. **(d)** Er erhält den Ausgangsstempel mit Zeit 10. Dann folgen Knoten **2** und **1** mit Ausgangszeiten 11 und 12. Es ergibt sich die Nebenordnung: **0**, **3**, **4**, **1**, **2**, **5** und die Hauptordnung: **4**, **3**, **0**, **5**, **2**, **1**.

Die Zeitstempel können mit einer Laufzeit in $\boldsymbol{O}(V + E)$ (Tiefensuche) vergeben werden. Offensichtlich liegt das Entdecken immer vor dem Fertig-Bearbeitet-Haben. Für jeden Knoten v gilt also $dt(v) < ft(v)$. Für das Verhältnis zwischen den Zeitstempeln zweier verschiedener Knoten gilt der folgende Satz:.

Erreichbarkeit im Suchwald der Tiefensuche

Es gibt einen gerichteten Pfad von v nach w im Suchwald, genau dann, wenn $dt(v) \leq dt(w)$ und $ft(w) \leq ft(v)$ gilt.

Diese Eigenschaft wird bei der Topologischen Sortierung im folgenden Abschnitt eine entscheidende Rolle spielen. Des Weiteren kann mit Hilfe der Zeitstempel eine Methode `isDAG()` implementiert werden, die in einer Laufzeit in $\boldsymbol{O}(V + E)$ entscheidet, ob ein gegebener Digraph ein **zyklenfreier Digraph** (*directed acyclic graph*, DAG) ist.

9.6 Topologische Sortierung

Unter einer **topologischen Sortierung** eines Digraphen versteht man eine Reihenfolge der Knoten, die mit seinen Pfeilen verträglich ist: Durchläuft man die Knoten in dieser Reihenfolge, so zeigen alle Pfeile auf zukünftige Knoten.

Als Folgerung aus der Definition ergibt sich: Graphen mit topologischer Sortierung haben mindestens eine *Quelle* (Knoten am Anfang der Ordnung) und mindestens eine *Senke* (Knoten am Ende der Ordnung).

Wenn die Pfeile Vorrangbedingungen (*precedence constraints*) darstellen (v muss vor w geschehen), dann liefert die topologische Sortierung eine Ablaufreihenfolge. Die Aufgabe des topologischen Sortierens kommt in vielen Anwendungsbereichen vor, z. B. Vererbungshierarchien, Ablaufplanung, Tabellenkalkulation. Bei der Tabellenkalkulation stehen in den Zellen Formeln, die die Werte von anderen Zellen benutzen können. Die Werte der verwendeten Zellen müssen also zuerst berechnet werden (Vorrangbedingung). In Zelle `G10` kann z. B. die Formel `=SUM(A10:F10)` stehen. Bevor Zelle `G10` berechnet werden kann, müssen also die Werte der Zellen `A10`, `B10`, ..., `F10` berechnet werden. Diese können aber wieder von anderen Zellen abhängen. In `A10` könnte beispielsweise die Formel `=AVERAGE(A1:A9)` stehen. Bei großen Tabellenkalkulationen können komplexe Abhängigkeiten entstehen. Die Vorrangbedingungen können leicht in einem gerichteten Graphen modelliert werden. Dort würde zunächst auf Zyklenfreiheit getestet werden und ein vorhandener Zyklus als Inkonsistent angezeigt werden. Ist der Digraph zyklenfrei, kann mit Hilfe der Topologischen Sortierung eine Reihenfolge identifiert werden, in der die Werte der Zellen berechnet werden können.

Topologische Sortierungen können durch eine Rangordnung definiert werden.

Definition: Topologische Ordnung

Eine injektive (rechtseindeutige) Funktion

$$r : \boldsymbol{V} \to \mathbb{R}$$

definiert eine topologische Ordnung eines Digraphen $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, wenn für alle Kanten $v{\to}w \in \boldsymbol{E}$ gilt: $r(v) < r(w)$.
Man kann den Wertebereich der Funktionen auch auf $\{0, \ldots, V-1\}$ (Knotenindizes) einschränken. Hier wird ganz $\mathbb{R}$ zugelassen, da dies die Definitionen von topologischen Ordnungen vereinfacht. Mit einer nachgeschalteten Rangfunktion kann man eine Abbildung auf die Knotenindizes erreichen. Dies nennen wir Topologische Rangfolge.

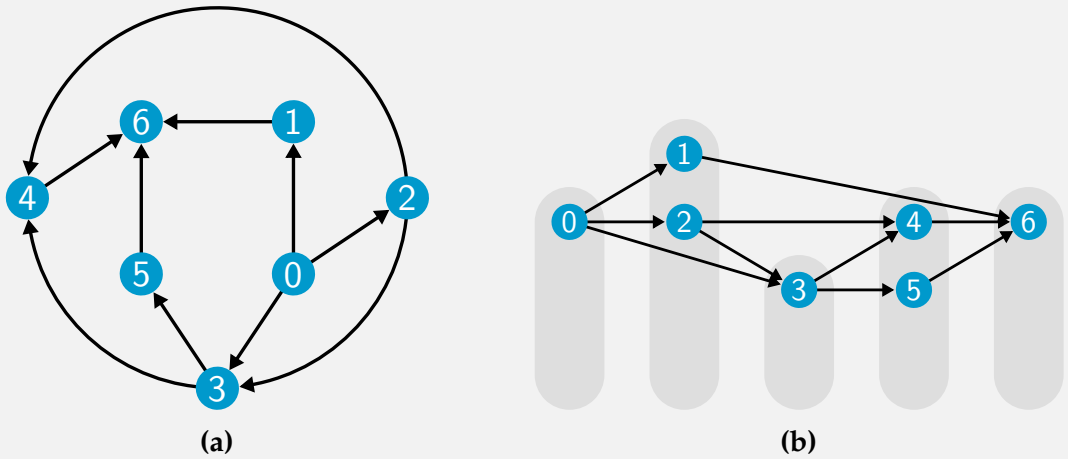


Abbildung 9.13 **Topologische Sortierung eines zyklenfreien Digraphen:** Es ist ein Digraph mit beliebiger Knotenindizierung gegeben (nicht dargestellt). Mit einer topologischen Sortierung bestimmt man eine Rangfolge der Knoten, die mit den Pfeilen als Vorrangbedinung verträglich ist. In **(a)** wird diese Rangfolge als Knotenindizes verwendet. Durch die geänderten Knotenpositionen (bei gleicher Struktur) in **(b)** wird die Struktur der Topologischen Sortierung deutlich: alle Pfeile zeigen von links nach rechts. Im schattierten Bereich ganz links befinden sich die Quellen, und auf der rechten Seite die Senken. Die Reihenfolge innerhalb der schattierten Bereiche ist nicht festgelegt.

Existenz einer Topologischen Ordnung

Ein Digraph kann genau dann topologisch geordnet werden, wenn er zyklenfrei ist.

Offensichtlich kann es bei Existenz eines Zyklus keine topologische Ordnung geben. Dass jeder DAG eine topolische Sortierung besitzt, zeigen wir an Hand eines Algorithmus, der diese Aufgabe löst. Die Reihenfolge des Entdeckens bei der Tiefensuche liefert eine gute Ausgangsbasis. Für Baumkanten und Vorwärtskanten ist die Vorrangbedingung erfüllt. Da es keine Rückwärtskanten gibt (Zyklenfreiheit), bleiben die Seitenkanten als einziger kritischer Punkt. Die umgekehrte Reihenfolge des fertig Bearbeitens (**umgekehrte Hauptordnung**, *reverse postorder*) erfüllt in allen Fällen die topologische Sortierung.

Beweis der Existenz einer Topologischen Sortierung

Definition einer Topologischen Ordnung durch umgekehrte Hauptordnung

Die umgekehrte Hauptordnung, die durch die Rangordnung $r(v) = -ft(v)$ definiert ist, ergibt eine topologische Ordnung für jeden zyklenfreien Digraphen.

Zum Beweis sei $v \to w$ eine Kante des Digraphen G. Wir betrachten die Situation, in der `dfs(G,v)` aufgerufen wird und zeigen $ft(w) < ft(v)$. Es gibt folgende Möglichkeiten:

- `w` ist noch nicht markiert.

  Dann liegen `v` und `w` in demselben Pfad des Suchwaldes und `v` wird zuerst entdeckt. Folglich gilt $ft(w) < ft(v)$, siehe die Eigenschaften der Zeitstempel.

- w ist markiert.

  Dann muss w schon vollständig abgearbeitet sein, also gilt $ft(w) < dt(v) < ft(v)$. (Wäre w noch nicht abgearbeitet, dann müsste es einen Pfad von w nach v geben, was zusammen mit der Kante $v \to w$ einen Zyklus ergäbe.)

  Falls w aus einem anderen Pfad als v stammt (SK), ist dieser bereits vollständig abgearbeitet, also $ft(w) < dt(v) < ft(v)$.

  Falls `w` aus demselben Pfad stammt, muss $v \to w$ eine VK sein, da RK auf Grund der Zyklenfreiheit nicht existieren. Für VK gilt $ft(w) < ft(v)$.

Für jeden Pfeil $v \to w$ gilt also $ft(w) < ft(v)$. □

Für die Implementierung der Topologischen Sortierung benutzen wir einen `Stack` für die *umgekehrte* Hauptordnung. Um die Knoten in der korrekten Reihenfolge abzurufen, muss man bei dem `Stack` aus `java.util` die Elemente per `pop()` abrufen. Bei der Benutzung des `Stack` als `Iterable` funktioniert es nur mit anderen `Stack` Implementationen, z. B. mit der in Listing **??** in der Java Einführung.

Listing 9.12 Implementierung der Topologischen Sortierung. Sie funktioniert nur für DAG. Es sollte also zunächst auf Zyklenfreiheit getestet werden.

```
public class Topological {
  private boolean[] marked;
  private Stack<Integer> reversePost;

  public Topological(Digraph G) {           // expects that G is a DAG
    reversePost = new Stack<>();
    marked = new boolean[G.V()];
    for (int v = 0; v < G.V(); v++) {
      if (!marked[v]) {
        dfs(G, v);
      }
    }
  }

  public void dfs(Digraph G, int v) {
    marked[v] = true;
    for (int w : G.adj(v)) {
      if (!marked[w]) {
        dfs(G, w);
      }
    }
    reversePost.push(v);
  }

  public Stack<Integer> order() {
     return reversePost;
  }
}
```

Laufzeit und Speicherbedarf der Topologischen Sortierung

Die Topologische Sortierung eines zyklenfreien Digraphen hat eine Laufzeit in $\boldsymbol{O}(V+E)$ und Speicherbedarf in $\boldsymbol{O}(V)$. Die Zyklenfreiheit kann ebenfalls mit einer Laufzeit in $\boldsymbol{O}(V+E)$ überprüft werden.

Die Laufzeit ergibt sich, wie bei der Tiefensuche, aus der Tatsache, dass jeder Knoten nur einmal besucht (markiert) wird und daher die Zeilen maximal für jede Kante einmal ausgeführt werden. Der Speicherbedarf ist ebenfalls wie bei der Tiefensuche.

9.7 Kategorisierung von Kanten in Digraphen

Mit Hilfe der Tiefensuche kann man die Kanten des Suchbaums, bzw. des Suchwaldes, eines gerichteten Graphen kategorisieren. Da das Ergebnis der Tiefensuche nicht nur von der Struktur des Graphen abhängt, sondern durch andere Faktoren stark variieren kann, bezieht sich diese Kategorisierung nur auf einen konkreten Durchlauf der Tiefensuche, nicht generell auf den Graphen.

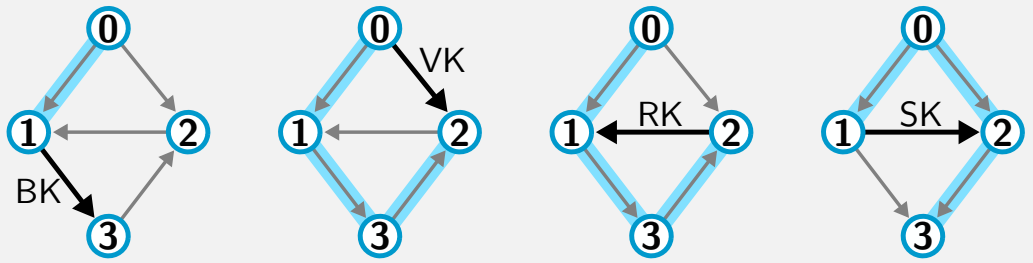
Kategorisierung der Kanten eines Digraphen bei Tiefensuche

Es wird über alle Knoten eines Digraphen iteriert und falls der Knoten vorher noch nicht entdeckt wurde, wird eine Tiefensuche mit ihm als Anfangsknoten gestartet. Während eines Durchlaufs der Tiefensuche werden Kanten $v \to w$ untersucht, ausgehend von einem bereits markierten Knoten v. An diesem Punkt des Algorithmus wird die folgende Einteilung der Kanten in Baumkanten, Vorwärtskanten, Rückwärtskanten und Seitenkanten vorgenommen:

- Ist w noch nicht markiert, so wird $v \to w$ eine **Baumkante**, also Teil des Suchwalds.
- Sonst, wenn also w auch schon markiert ist:
- Wenn es einen gerichteten Pfad im Suchwald von v nach w gibt: **Vorwärtskante** (VK).
- Wenn es einen gerichteten Pfad im Suchwald von w nach v gibt: **Rückwärtskante** (RK).
- Andernfalls ist es eine **Seitenkante** (SK).

Diese Einteilung ist abhängig von der Reihenfolge, in der die Knoten zum Start der Tiefensuche durchlaufen werden.

- Die Unterscheidung zwischen Vorwärts- und Rückwärtskanten ist eindeutig: Jeder Knoten gehört zu genau einem Baum des Suchwaldes. Einen Pfad kann es nur geben, wenn v und w in demselben Suchbaum liegen. In einem Baum kann es nicht gerichtete Pfade in beiden Richtungen geben, also entweder Vorwärts- oder Rückwärtskante.
- Vorwärtskanten sind Abkürzungen: Eine Vorwärtskante ist nicht Teil des Suchwaldes. Sie ist ein Pfad von v nach w der Länge 1. Der im Suchbaum existierende Pfad muss mindestens die Länge 2 haben.
- Rückwärtskanten schließen einen gerichteten Zyklus: Im Suchbaum existiert ein gerichteter Pfad von w nach v. Die gerichtete Kante $v \to w$ schließt also den Zyklus (ist aber nicht Teil des Suchwaldes).
- Seitenkanten verbinden zwei markierte Knoten. Also liegen v und w auf Pfaden des Suchwaldes. Da es aber weder einen Pfad von v nach w noch einen Pfad von w nach v im Suchwald gibt, sind diese Pfade getrennt, ggf. sogar in unterschiedlichen Bäumen.



(a) Baumkante **(b)** Vorwärtskante **(c)** Rückwärtskante **(d)** Seitenkante

Abbildung 9.14 Kategorien von Kanten eines Digraphen bei Tiefensuche: Die Kanten werden während der Tiefensuche kategorisiert. Die hellblaue Hinterlegung zeigt, welche Kanten bereits Teil des Suchbaums sind. Die markierte Kante wird gerade inspiziert. Bei Kante **1**→**3** in **(a)** wird der Zielknoten neu entdeckt. Sie ist eine Baumkante. Kante **0**→**2** in **(b)** ist eine Vorwärtskante, da es einen Pfad von **0** nach **2** im Suchbaum gibt, nämlich **0**→**1**→**3**→**2**. Kante **2**→**1** in **(c)** ist eine Rückwärtskante, da es einen Pfad **1** ⇝**2** im Suchbaum gibt, nämlich **1**→**3**→**2**, der durch **2**→**1** zu einem Zyklus geschlossen würde. Die Kante **1**→**2** in **(d)** ist eine Seitenkante. Sie verbindet die Pfade **0**→**2** und **0**→**1** des Suchbaums.

Abbildung 9.15 demonstriert, wie die Kategorisierung von der Reihenfolge der Startknoten und von der Reihenfolge der Adjazenzlisten abhängt.

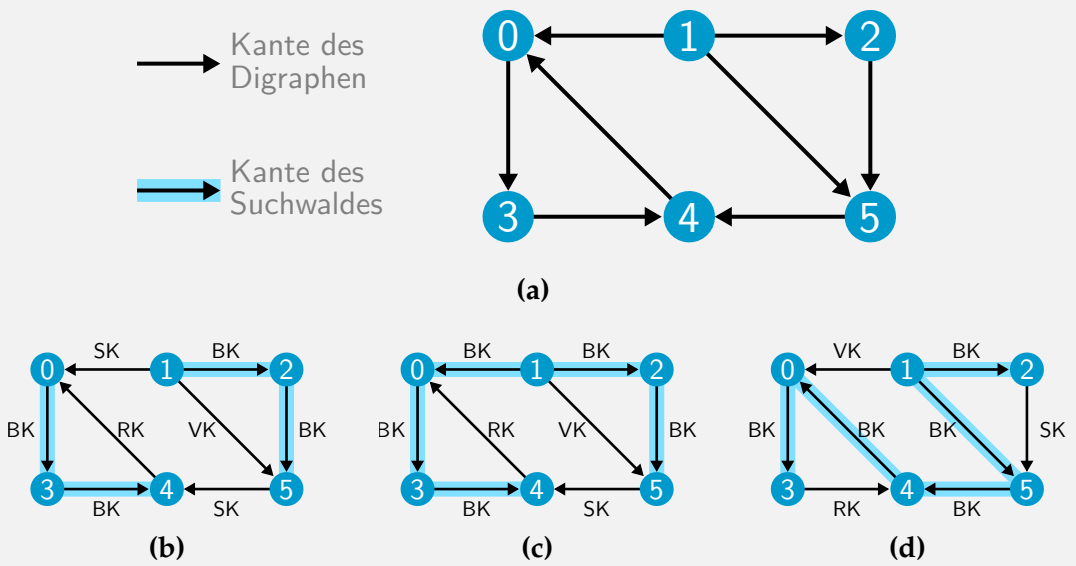


Abbildung 9.15 Beispiele für Kategorisierung der Kanten: Die Kanten des gerichteten Graphen in **(a)** werden mit iterierter Tiefensuche kategorisiert. Das Ergebnis hängt von der Reihenfolge der Knoten und Kanten ab. In **(b)** werden die Knoten in aufsteigender Reihenfolge durchlaufen. **(c)** ergibt sich, wenn die Tiefensuche bei Knoten **1** gestartet wird, und **(d)** bei Start von Knoten **1** in Richtung von Knoten **5**.

Effiziente Zuordnung der Kategorien

Mit unserer Definition sind die Kategorien der Kanten nicht effizient festzustellen. Die Überprüfung, ob ein Pfad von v nach w im Suchbaum existiert, erfordert eine Iteration entlang der Baumkanten. Da diese Überprüfung im *worst-case* für alle Kanten durchgeführt werden muss, landen wir auf diese Weise bei einer mindestens quadratischen Laufzeit. Hier kommen die Zeitstempel ins Spiel. Mit dem Satz über die Erreichbarkeit im Suchwald der Tiefensuche lassen sich die Bedingungen zur Unterscheidung der Knotentypen VK, RK und SK in konstanter Zeit überprüfen.
Eine effiziente Kategorisierung mit Hilfe der Zeitstempel ist somit auf die folgende Weise möglich. In einem ersten Durchgang werden die Zeitstempel in $\boldsymbol{O}(V+E)$ vergeben und anschließend die Kanten per DFS kategorisiert. Da das Kriterium der "Erreichbarkeit im Suchwald" aus Abschnitt 9.5 eine konstante Laufzeit besitzt, hat das gesamte Verfahren eine Laufzeit in $\boldsymbol{O}(V+E)$.

9.8 Das Wichtigste in Kürze

- Dünne Graphen werden am besten mit Adjazenzlisten implementiert.

- Graphen können mit Breiten- und Tiefensuche in $\boldsymbol{O}(V + E)$ durchsucht / traversiert werden.
- Ausgehend von s wird ein Suchbaum erstellt, der alle von s erreichbaren Knoten enthält (Spannbaum).
- Falls der Graph nicht zusammenhängend ist, erhält man mit iterierten Suchdurchläufen einen Suchwald.
- Breitensuche erstellt Wege, die über möglichst wenige Kanten zum Ziel führen.
- Tiefensuche liefert Strukturinformation, u. a. eine topologische Sortierung für zyklenfreie Digraphen (DAG).

Herausforderungen für Graphenalgorithmen

Es gibt viele Problemstellungen rund um Graphen. Manche von ihnen sind mit den hier vermittelten Methoden leicht zu lösen, andere sind schwerer, sehr schwer, oder bisher noch gar nicht zufriedenstellend gelöst. Hier ist eine kleine Auswahl:

- **Bipartit:** Können die Knoten mit zwei Farben eingefärbt werden, so dass keine Kante Knoten derselben Farbe verbindet?
  Einfach mit Kenntnissen dieses Kapitels
- **Euler Zyklus:** Finde einen Zyklus, der jede Kante genau einmal benutzt. Gibt es immer, wenn alle Knoten eine grade Ordnung haben.
  Relativ einfach mit Kenntnissen dieses Kapitels
- **Hamilton Zyklus:** Finde einen Zyklus, der jeden Knoten genau einmal benutzt.
  NP-schweres Problem
- **Isomorphismus:** Prüfe, ob zwei Graphen bis auf Nummerierung der Knoten übereinstimmen.
  Offenes Problem, ob dies in P lösbar ist
- **Ebenes Layout:** Finde eine Anordnung der Knoten, so dass sich keine Kanten kreuzen.
  Komplexer Algorithmus mit DFS und linearer Laufzeit [Hopcroft & Tarjan 1974]

Quellenverweise

Die Implementationen in diesem Kapitel stammen aus [Sedgewick and Wayne, 2014]. Bei der Darstellung der Algorithmen wurden zusätzlich [Ottmann and Widmayer, 2011] und [Cormen et al., 2013] zu Rate gezogen.

KAPITEL 10

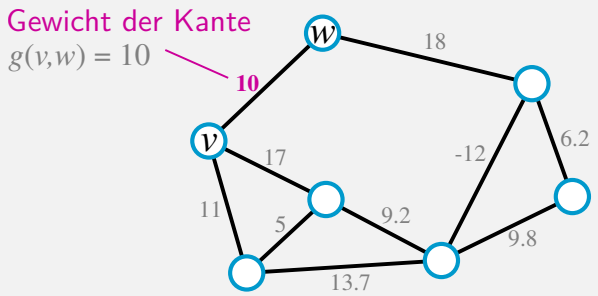
Minimale Spannbäume

10.1 Gewichtete Graphen

Für die Modellierung vieler praktischer Probleme ist es wichtig, den Kanten eines Graphen Gewichte zuordnen zu können.[1] Diese Gewichte können z. B. die Längen der Streckenabschnitte oder die Fahrtdauer sein.
In diesem Kapitel verwenden wir **gewichtete Graphen** und in dem nächsten gewichtete Digraphen. Dabei wird jeder Kante des Graphen ein Gewicht zugeordnet. Man kann auch Knoten-gewichtete Graphen betrachten, was wir in diesem Skript jedoch nicht tun.

Kantengewichtete Graphen

Ein **gewichteter Graph** ist ein Tupel $G = (V, E, g)$, wobei (V, E) ein Graph und $g : E \to \mathbb{R}$ eine Gewichtsfunktion ist. Wir schreiben $g(v, w)$, bzw. *weight(v,w)* für das Gewicht der Kante (v, w). Negative Gewichte sind erlaubt, wenn sie nicht explizit ausgeschlossen werden.



[1] In manchen Anwendungen / Algorithmen ist es üblich von *Kosten* zu sprechen. Wir verwenden die Begriffe synonym und benutzen jeweils den im gegebenen Kontext gebräuchlicheren.

Da zu jeder Kante ein Gewicht gespeichert werden soll, führen wir für gewichtete Graphen eine Klasse für gewichtete Kanten ein, die die beiden Knoten sowie das Gewicht enthält. Dies zieht weitere Änderungen in der Implementation mit sich. Anstelle von Adjazenzlisten gibt es Inzidenzlisten, die für jeden Knoten die *inzidenten Kanten* enthalten. Bei ungerichteten Graphen ist jede Kante doppelt gespeichert, und zwar in den Inzidenzlisten *beider* Knoten.

API für eine gewichtete Kante

public class Edge implements Comparable<Edge>

	`Edge(int v, int w, double weight)`	Kante `v-w` mit Gewicht `weight`
`double`	`weight()`	Gewicht der Kante
`int`	`either()`	Einer der beiden Knoten dieser Kante
`int`	`other(int v)`	Der Knoten, der nicht `v` ist
`int`	`compareTo(Edge that)`	Vergleich nach Gewicht

API für einen kantengewichteten Graphen

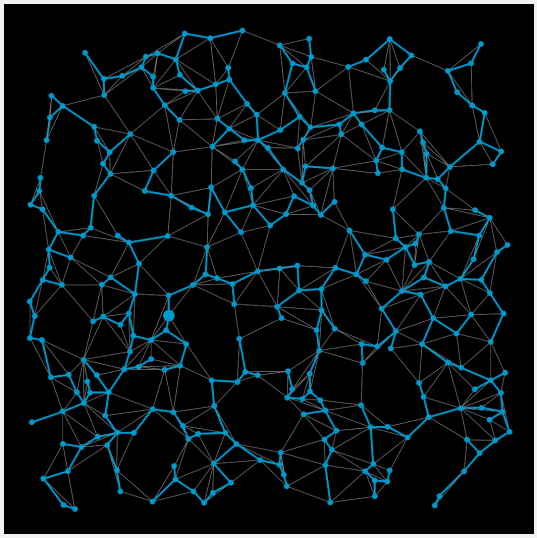
public class EdgeWeightedGraph

	`EdgeWeightedGraph(int V)`	Erzeugt leeren Graphen mit `V` Knoten
`int`	`V()`	Anzahl der Knoten
`int`	`E()`	Anzahl der Kanten
`void`	`addEdge(Edge e)`	Füge Kante `e` zum Graphen hinzu
`Iterable<Edge>`	`incident(int v)`	Kanten inzident zu `v`
`Iterable<Edge>`	`edges()`	Alle Kanten dieses Graphen

10.2 Minimale Spannbäume

Spannbaum und Minimaler Spannbaum

Der **Spannbaum** eines Graphen G ist ein Teilgraph von G, der alle Knoten von G enthält und ein Baum (also zusammenhängend und zyklenfrei) ist. Das Gewicht eines Spannbaumes eines gewichteten Graphen ist die Summe seiner Kantengewichte. Ein **Minimaler Spannbaum** (MST) eines gewichteten Graphen ist ein Spannbaum des Graphen, der unter allen Spannbäumen des Graphen minimales Gewicht hat.



Jeder Spannbaum mit V Knoten hat $V - 1$ Kanten. Aber das Gewicht unterschiedlicher Spannbäume gewichteter Graphen kann sehr unterschiedlich sein, siehe Abbildung 10.1. Die Bestimmung minimaler Spannbäume ist für viele Anwendungen in der Praxis wichtig. Und eine algorithmische Herausforderung: Ein vollständig verbundener Graph mit V Knoten hat V^{V-2} Spannbäume (Cayleys Formel, siehe [Ariannejad and Emami, 2014]). Wir brauchen also einen effizienten Algorithmus, kein *brute force*.

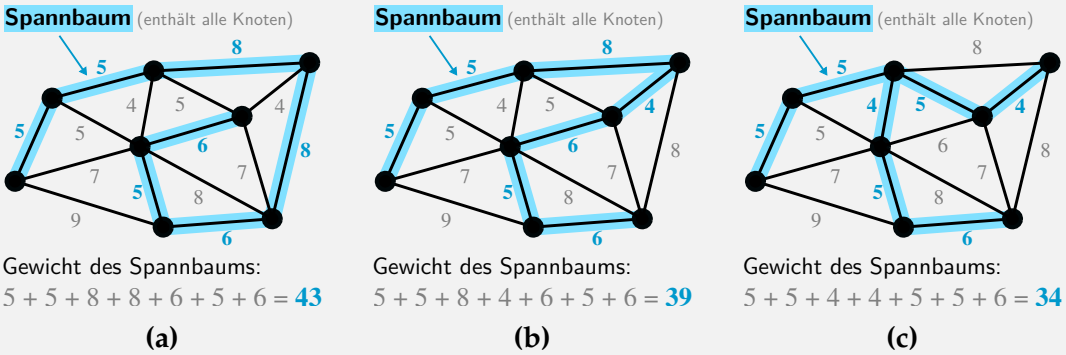


Abbildung 10.1 Minimaler Spannbaum: Ein Graph kann verschiedene Spannbäume haben. Der Spannbaum in **(a)** hat das Gewicht 43 und der in **(b)** das Gewicht 39. **(c)** zeigt einen minimalen Spannbaum des Graphen. Er hat das Gewicht 34. Es gibt auch andere Spannbäume dieses Graphen mit demselben Gewicht.

Der erste Algorithmus zur Bestimmung eines minimalen Spannbaums geht auf eine Arbeit des tschechischen Mathematikers Borůvka von 1926 zurück. Er ent-

wickelte das Verfahren zur optimierten Planung des elektrischen Netzwerkes von Mähren (südöstlicher Teil der Tschechischen Republik). Heute werden minimale Spannbäume in vielen Bereichen eingesetzt, z. B. in der Clusteranalyse, Gesichtserkennung, Bildbearbeitung und im allgemeinen Netzwerkdesign. Darüber hinaus werden die Algorithmen zur approximativen Lösung von NP-harten Problemen benutzt.

Spannendes über Spannbäume

Sei $T = (V, E')$ ein Subgraph von $G(V, E)$. Dann ist T genau dann ein *Spannbaum* von G, wenn eine (und damit alle) der folgenden, äquivalenten Bedingungen erfüllt ist:

- Jedes Knotenpaar von T ist durch genau einen einfachen Pfad verbunden.
- T hat $V - 1$ Kanten und ist zusammenhängend.
- T hat $V - 1$ Kanten und ist azyklisch.
- T ist azyklisch und das Hinzufügen einer beliebigen Kante erzeugt einen Zyklus.
- T ist zusammenhängend und das Entfernen einer beliebigen Kante macht ihn unzusammenhängend.
- T ist azyklisch und zusammenhängend.

Allgemeines Vorgehen zur Bestimmung eines Minimalen Spannbaums

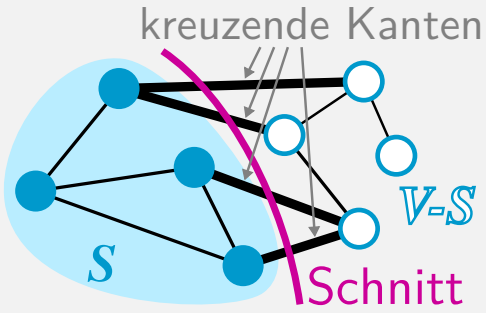
Das *globale* Optimierungsproblem kann durch iterierte Sicherstellung einer *lokalen Eigenschaft* mit einem Greedy Algorithmus gelöst werden. Wir werden zunächst einen allgemeinen Ansatz formulieren und dann zu konkreten Algorithmen für eine geeignete Kantenauswahl fortschreiten. Wir setzen im Folgenden voraus, dass der gegebene Graph *zusammenhängend* ist. Andernfalls wendet man die Algorithmen auf seine Zusammenhangskomponenten an und erhält einen minimalen Spannwald.

Schnitte durch Graphen und kreuzende Kanten

Ein **Schnitt** durch einen Graphen $G = (V, E)$ teilt seine Knoten in zwei nicht-leere Teilgraphen. Konkret beschreiben wir einen Schnitt durch eine Teilmenge $S \subseteq V$ mit der Eigenschaft, dass (S, E_S) und $(V - S, E_{V-S})$ nicht-leere Graphen sind. Dabei bezeichnet E_S die Menge aller derjenigen Kanten von G, die in S liegen:

$$E_S := \{(v, w) \in E \mid v \in S \ \& \ w \in S\}$$

Kanten mit einem Knoten innerhalb und einem Knoten außerhalb von S heißen **kreuzende Kanten**. Diejenigen der kreuzenden Kanten, die minimales Gewicht haben, heißen **minimal kreuzende Kanten**. Es kann mehr als eine minimal kreuzende Kante geben.



Mit der Definition von Schnitten und minimal kreuzenden Kanten können wir die *Schnitteigenschaft* formulieren und beweisen, die eine zentrale Rolle in der Bestimmung eines Minimalen Spannbaumes einnimmt.

Schnitteigenschaft

Sei ein beliebiger Schnitt durch einen Graphen gegeben. Jeder minimale Spannbaum des Graphen muss eine der minimal kreuzenden Kanten enthalten.

Beweis. Wir nehmen an, dass es einen MST gibt, der zu dem gegebenen Schnitt keine der minimal kreuzenden Kanten enthält. Sei (v, w) eine minimal kreuzende Kante und B der MST erweitert um die Kante (v, w). Da der MST schon alle Knoten verbunden hat, entsteht so ein Zyklus und es muss eine weitere kreuzende Kante des MST geben, die an dem Zyklus beteiligt ist, sagen wir (t, u).
Wenn (t, u) aus B entfernt wird, bleibt B zusammenhängend und ist also ein Spannbaum. Da (v, w) ein kleineres Gewicht als (t, u) hat, ist der neue Spannbaum leichter als der ursprüngliche, was der Annahme widerspricht, dass es ein MST war. □

Aus der Schnitteigenschaft folgt unmittelbar: Wenn alle Gewichte des Graphen unterschiedlich sind, ist für jeden Schnitt die minimal kreuzende Kante eindeutig. In diesem Fall ist auch der MST eindeutig. Des Weiteren kann aus der Schnitteigenschaft der allgemeine Ansatz zur Bestimmung minimaler Spannbäume abgeleitet werden:

Listing 10.1 Generischer Greedy Ansatz zur Bestimmung eines MST

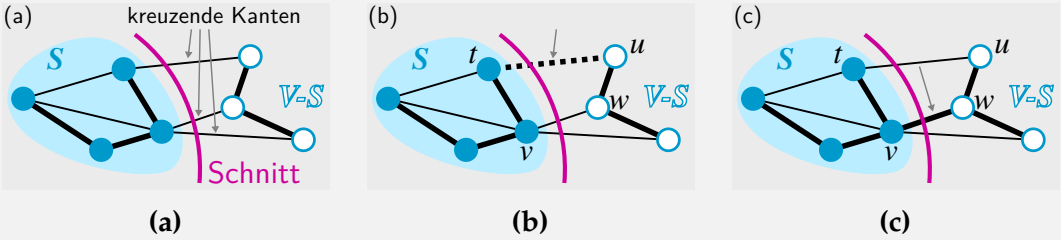


Abbildung 10.2 **Beweis der Schnitteigenschaft:** **(a)** Der MST muss eine der kreuzende Kanten enthalten. **(b)** Wir nehmen an, es ist (t, u) und nicht die minimale Kante (v, w). **(c)** Weglassen von (t, u) und Hinzufügen von (v, w) erhält Spannbaum-Eigenschaft und verringert das Gewicht. Also muss (v, w) im MST enthalten sein. Diese Darstellung ist etwas vereinfachend und ist nur als Ergänzung zu dem formalen Beweis zu sehen.

```
1 ME ←∅
2 while ME ist kein Spannbaum
3   wähle Schnitt S, dessen kreuzende Kanten alle nicht in ME liegen
4   füge eine minimal kreuzende Kante (v, w) zu ME hinzu
5 end
6 // ME ist ein minimaler Spannbaum
```

Der Ansatz ist ein Greedy Verfahren, da in jedem Schritt eine Kante ausgewählt und der Lösung *ME* zugefügt wird, ohne dass einer dieser Auswahlschritte später rückgängig gemacht wird.
Bei diesem allgemeinen Ansatz bleibt offen, wie der Schnitt S zu wählen ist. Ähnlich wie bei der Graphendurchsuchung, beweisen wir zunächst die Korrektheit des allgemeinen Ansatzes und widmen uns dann konkreten Algorithmen.

Korrektheit des allgemeinen Ansatzes zur Bestimmung eines MST

Der MST Algorithmus in Listing 10.1 bestimmt einen minimalen Spannbaum.

Beweis. Der folgende Beweis gilt für den Fall, dass alle Gewichte des Graphen unterschiedlich sind, also der MST eindeutig ist. Der Beweis kann auf den Fall ohne diese Einschränkung verallgemeinert werden.
Nach der Schnitteigenschaft ist jede hinzugefügte Kante Teil des MST. Zu zeigen bleibt, dass alle Kanten des MST ausfindig gemacht werden. Nehmen wir an, dies sei nicht der Fall. Dann stellen die Kanten *ME* keinen Spannbaum von G dar. Es gibt also mindestens eine nicht leere Zusammen-

hangskomponente von ME, die nicht ganz G ist. Wir wählen eine dieser Zusammenhangskomponenten als S. Für den durch S definierten Schnitt sind keine der kreuzenden Kanten in ME. Also würde der Pseudocode eine weitere Kante auswählen. □

Der Rest dieses Kapitels stellt drei konkrete Algorithmen für minimale Spannbäume vor:

- **Prims Algorithmus**: Bilde einen Baum ausgehend von einem Startknoten s. Wähle unter den kreuzenden Kanten eine mit geringstem Gewicht.
- ***Reverse-Delete* Algorithmus**: Durchlaufe die Kanten nach absteigendem Gewicht. Lösche eine Kante, falls sie den Graphen nicht unzusammenhängend macht.
- **Kruskals Algorithmus**: Durchlaufe die Kanten nach aufsteigendem Gewicht. Füge eine Kante hinzu, wenn sie keinen Zyklus mit den bisher gewählten Kanten bildet.

10.3 Prims Algorithmus für minimale Spannbäume

Aus dem allgemeinen Ansatz kennen wir die Strategie zur Erzeugung eines Minimalen Spannbaumes immer minimal kreuzende Kanten auszuwählen. Dabei bleibt offen bezüglich welchen Schnittes die kreuzenden Kanten gewählt werden. Ein naheliegender Ansatz ist, einen beliebigen Startknoten zu markieren und dann immer eine minimal kreuzende Kante bezüglich des markierten Bereiches auszuwählen. Dies ist der **Prim Algorithmus**:

- Markiere den zufällig gewählten Startknoten.
- Wähle eine minimal kreuzende Kante bezüglich des Schnittes, der durch den markierten Bereich definiert wird (Startknoten markiert, Endknoten nicht)
- Iteriere dies, bis $V - 1$ Kanten ausgewählt wurden.

Für eine effiziente Implementierung stellt sich noch die Frage, auf welche Weise die *minimale* unter den kreuzenden Kanten ausgewählt wird. Alle Kanten zu durchsuchen, hätte eine (worst-case) Laufzeit in $O(E)$. Effizienter ist es, alle in Frage kommenden Kanten in eine Vorrangwarteschlange einzufügen und daraus jeweils diejenige mit geringstem Gewicht zu entnehmen: Die Laufzeit dafür ist in $O(\log E)$. Mit dieser Idee gelingt die erste Implementation von Prims Algorithmus. Sie speichert den MST des Graphen `G` in dem Vorgängerarray `parent`:

Listing 10.2 Implementierung des Prim Algorithmus mit Vorrangwarteschlange orientiert an [Sedgewick and Wayne, 2014, S. 662].

```
1  public class MSTPrim {
2    protected PriorityQueue<Edge> pq;
3
4    public MSTPrim(WeightedGraph G) {
5      marked = new boolean[G.V()];
6      parent = new int[G.V()];
7      pq = new PriorityQueue<Edge>(G.V());
8
9      visit(G, 0);                  // Startknoten 0
10     while (!pq.isEmpty()) {       // hole minimal kreuzende Kante aus pq
11       Edge e = pq.poll();
12       int v = e.either();
13       int w = e.other(v);
14       if (!marked[w]) {           // prüfe, ob w immer noch unmarkiert,
15         parent[w] = v;            // also e eine kreuzende Kante ist
16         visit(G, w);
17       } else if (!marked[v]) {    // dito für v
18         parent[v] = w;
19         visit(G, v);
20       }
21     }
22   }
23
24   public void visit(WeightedGraph G, int v) {
25     marked[v] = true;
26     for (Edge e : G.incident(v)) {
27       int w = e.other(v);
28       if (!marked[w]) {           // nur kreuzende Kanten in pq speichern
29         pq.add(e);
30       }
31     }
32   }
33 }
```

Korrektheit und Laufzeit des Prim Algorithmus mit Priority Queue

Die Implementation des Prim Algorithmus mit Priority Queue bestimmt den minimalen Spannbaum in einer Laufzeit in $\boldsymbol{O}(E \log E)$ und Speicherbedarf in $\boldsymbol{O}(V + E)$. (Laufzeit und Speicherbedarf können mit einer verbesserten Implementation reduziert werden.)

Beweis der Korrektheit. Die Korrektheit folgt aus dem Satz zu dem allgemeinen Ansatz auf Seite 178. Um diesen Satz anwenden zu können, muss gezeigt werden, dass Prim ein Spezialfall des allgemeinen Ansatzes ist, also, dass Prim immer eine *minimal kreuzende* Kante bezüglich eines Schnittes auswählt.
Die Menge der markierten Knoten definiert einen Schnitt. In die Warteschlange werden nur *kreuzende* Kanten eingefügt (Zeile 28-29), und von diesen wird per PQ diejenige mit geringstem Gewicht ausgewählt (Zeile 11). Bei der Auswahl werden diejenigen Kanten verworfen, deren zweiter Knoten mittlerweile markiert wurde (Zeilen 14 und 17<). Daher wird immer eine minimal kreuzende Kante ausgewählt. □

Beweis der Laufzeit.

- Die Initialisierung hat eine Laufzeit in $\boldsymbol{O}(V)$.
- Die Methode `visit()` wird für jeden Knoten maximal einmal aufgerufen, denn der Aufruf geschieht nur für *unmarkierte* Knoten, die sodann in der Methode markiert werden. Daher gelangt jede *Kante* höchstens einmal in die Warteschlange.
- Die Methoden `isEmpty()`, `poll()` und `add()` der Priority Queue werden also höchstens E-mal aufgerufen.
- Die Methoden `isEmpty()` und `poll()` haben konstante Laufzeit.
- Da höchstens E Elemente (Kanten) in die Warteschlange eingefügt werden, hat `add()` eine Laufzeit in $\boldsymbol{O}(\log E)$.
- Insgesamt ergibt sich eine Laufzeit in $\boldsymbol{O}(E \log E)$. (Da wir den Graphen als zusammenhängend angenommen haben, gilt $E \geq V - 1$.) ■

Verbesserte Implementierung mit indizierter Vorrangwarteschlange

Der Prim Algorithmus kann durch Verwendung einer *indizierten* Vorrangwarteschlange beschleunigt werden. Bei der bisherigen Implementierung gelangen alle kreuzenden Kanten in die Warteschlange und verbleiben dort, bis sie abgerufen werden. Dies ist ineffizient: Sobald eine kürzere kreuzende Kante zu einem Knoten gefunden wurde, sind alle längeren kreuzenden Kanten irrelevant. Sie kommen gemäß der Schnitteigenschaft für die Auswahl nicht in Frage. Daher reicht es, für jeden *Knoten* die *kürzeste* ankommende Kante zu speichern. Dadurch reduziert sich die Länge der Warteschlange von E auf V. Außer dem Gewinn für die Laufzeit wird auch der Speicherbedarf von $\boldsymbol{O}(E)$ auf $\boldsymbol{O}(V)$ reduziert.

In der folgenden Implementierung wird ein knoten-indiziertes Array `weightOfCE` (CE = *crossing edge*) zu Speicherung des Gewichtes einer minimalen Kante verwendet.

Listing 10.3 Implementierung des Prim Algorithmus mit indizierter PQ gemäß [Sedgewick and Wayne, 2014, S. 664]. Die indizierte PQ ist in Listing 3.2 implementiert.

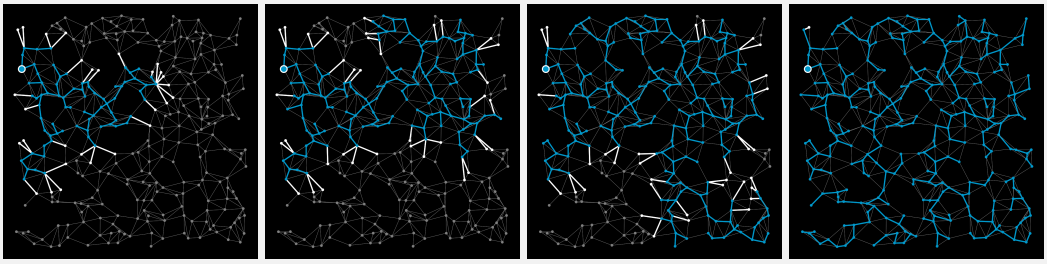
```
1  public class MSTPrim {
2    protected double[] weightOfCE;       // Gewicht einer minimal kreuzenden
3                                         // Kante zum Knoten
4    protected IndexPQ<Double> pq;
5
6    public MSTPrim(WeightedGraph G) {
7      marked = new boolean[G.V()];
8      parent = new int[G.V()];
9      weightOfCE = new double[G.V()];
10     for (int v = 0; v < G.V(); v++) {
11       weightOfCE[v] = Double.POSITIVE_INFINITY;
12     }
13     pq = new IndexPQ<Double>(G.V(), -1);
14     weightOfCE[0] = 0;                 // Start bei Knoten 0
15     pq.add(0, weightOfCE[0]);
16
17     while (!pq.isEmpty()) {
18       int v = pq.poll();
19       visit(G, v);
20     }
21   }
22
23   public void visit(WeightedGraph G, int v) {
24     marked[v] = true;
25     for (Edge e : G.incident(v)) {
26       int w = e.other(v);
27       if (marked[w])
28         continue;
29       if (e.weight() < weightOfCE[w]) {
30         parent[w] = v;
31         weightOfCE[w] = e.weight();
32         if (pq.contains(w)) {
33           pq.change(w, weightOfCE[w]);
34         } else {
35           pq.add(w, weightOfCE[w]);
36         }
37       }
38     }
39   }
40 }
```

Korrektheit und Laufzeit des Prim Algorithmus mit IndexPQ

Die Implementation des Prim Algorithmus mit *indizierter* Vorrangwarteschlange bestimmt den minimalen Spannbaum in einer Laufzeit in $\boldsymbol{O}(E \log V)$ und mit Speicherbedarf in $\boldsymbol{O}(V)$.

Beweis. Die Korrektheit ergibt sich wie zuvor. Die Laufzeit ergibt sich wie folgt: Die Methode `visit()` wird für jeden Knoten maximal einmal aufgerufen. Die `for` Schleife in `visit()` wird also höchstens E-mal durchlaufen. Die Warteschlange enthält jeden Knoten höchstens einmal. (Für schon vorhandene Knoten wird nur ggf. das Gewicht aktualisiert.) Daher ist die Laufzeit jedes Aufrufs der PQ-Methoden `change()` und `add()` in $\boldsymbol{O}(\log V)$. Bei einer *indizierten* PQ hat `contains()` konstante Laufzeit, siehe Abschnitt 3.2.2. Insgesamt erhalten wir also eine Laufzeit in $\boldsymbol{O}(E \log V)$.

Der Ablauf von Prims Algorithmus bei dem Aufbau eines Minimalen Spannbaums ist in Abbildung 10.3 dargestellt.



(a) **(b)** **(c)** **(d)**

Abbildung 10.3 Ablauf von Prims Algorithmus zur Erstellung eines Minimalen Spannbaums: Ausgehend von einem beliebig ausgewählten Startknoten, werden schrittweise Kanten ausgewählt und die Zielknoten markiert (blau eingefärbt). In jedem Schritt wird eine der kreuzenden Kanten (weiß eingefärbt) ausgewählt und zwar eine mit dem geringsten Gewicht unter allen kreuzenden Kanten. In diesem Beispiel wurde die (euklidische) Länge einer Kante als ihr Gewicht definiert.

10.4 Der unbekanntere Kruskal Algorithmus: *Reverse-Delete*

Im nächsten Abschnitt wird der weitläufig bekannte Kruskal Algorithmus betrachtet. Es existiert auch eine weniger bekannte Variante, die im selben Artikel [Kruskal, 1956] vorgeschlagen wurde: Der *Reverse-Delete* Algorithmus. Sie funktioniert so:

Reverse-Delete Algorithmus für minimale Spannbäume

- Beginne mit dem gegebenen Graphen.
- Durchlaufe die Kanten in der Reihenfolge nach *absteigendem* Gewicht.
- Überprüfe, ob das Entfernen der Kante den Graphen *unzusammenhängend* macht, wenn nicht, entferne sie.
- Andernfalls erhalte sie und fahre mit der nächsten Kante fort bis alle Kanten durchlaufen wurden.

Eine effiziente Implementierung, insbesondere des Tests auf Zusammenhang, ist komplex. Daher wird der Algorithmus in diesem Skript nicht weiter besprochen. Die geschickte Implementierung in [Thorup, 2000] erreicht eine Laufzeit in $\boldsymbol{O}(E \log V (\log \log V)^3)$.

10.5 Kruskals Algorithmus für minimale Spannbäume

Der prominente Kruskal Algorithmus geht umgekehrt vor. Die Kanten werden in der Reihenfolge nach *aufsteigendem* Gewicht durchlaufen. In jedem Schritt wird die aktuelle Kante ggf. zu einer Menge *MST* hinzugefügt, die am Ende den minimalen Spannbaum bildet. Es wird geprüft, ob das Hinzufügen der Kante zu einem Zyklus in *MST* führt. Nur wenn dies nicht der Fall ist, wird sie ausgewählt.
Der Ablauf ist völlig anders als bei dem Prim Algorithmus. Die ausgewählten Kanten bilden zunächst keinen Baum, der schrittweise vergrößert wird. Daher können die Kanten beim Ablauf nicht, wie bisher, in einem Knoten-indizierten `parent` Array gespeichert werden. Bei der Auswahl einer Kante ist noch nicht klar, welcher Knoten später in dem Spannbaum Eltern- und welcher Kindknoten sein wird.
Obwohl der Algorithmus einfach zu implementieren aussieht, gibt es einen Haken für eine halbwegs effiziente Lösung. Das Problem ist der Test, ob eine Kante (v, w) einen Zyklus erzeugen würde, der in jedem Schritt durchgeführt werden muss. Die Prüfung auf Zyklen mit Tiefensuche hat eine Laufzeit in $\boldsymbol{O}(V + E)$. Hier ginge der Test einfacher als in dem allgemeinen Fall, da wir nur

mit Tiefensuchen prüfen müssten, ob w von v aus erreichbar ist. Dies hat eine Laufzeit in $\boldsymbol{O}(V)$.

Da bei Kruskal dieser Test für *jede Kante*, also E-mal, durchgeführt werden muss, ergäbe sich eine Gesamtlaufzeit von $\boldsymbol{O}(VE)$: zu langsam! Glücklicherweise liegt hier eine spezielle Situation vor: Der Graph, der während des Kruskal Algorithmus auf Zyklen geprüft wird, ändert sich bei jedem Schritt nur um eine Kante. Anstatt die Zyklenprüfung jedes Mal naiv neu zu starten, können wir schrittweise eine Struktur aufbauen, die eine schnelle Prüfung ermöglicht.

Die Frage, ob durch Hinzufügung der Kante (v, w) ein Zyklus entsteht, ist äquivalent zu der Frage, ob v und w bereits in derselben Zusammenhangskomponente liegen. Bei Benutzung der Identifikation von Zusammenhangskomponenten mit Tiefensuche wäre in Bezug auf Laufzeit nichts gewonnen. Wir benutzen daher die Datenstruktur `UnionFind`, die in Abschnitt 3.3 eingeführt wurde. Sie ist sehr effizient für die dynamische Verwaltung von Zusammenhangskomponenten.

Mit Hilfe von `UnionFind` kann der Algorithmus von Kruskal auf die folgende Weise formuliert werden:

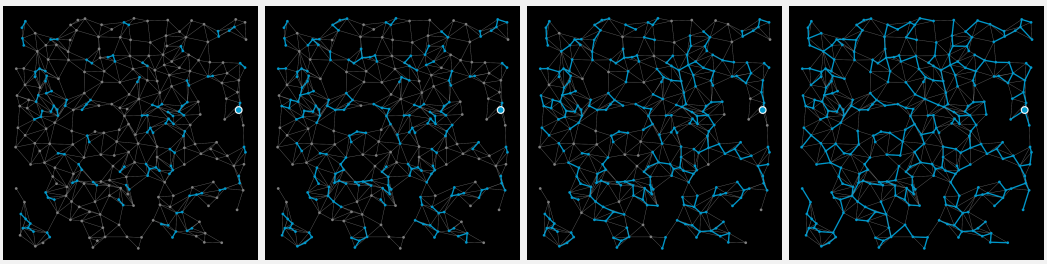
Listing 10.4 Algorithmus von Kruskal mit `UnionFind` zur Bestimmung eines MST des gewichteten Graphen $(\boldsymbol{V}, \boldsymbol{E}, g)$.

```
1  UF : UnionFind(V)    // Initialisierung mit einzelnen Komponenten
2                       // für jeden Knoten
3  MST ←∅               // Kanten des minimalen Spannbaums (z.B. als Queue)
4   sortiere  Kanten E aufsteigend nach Gewicht
5  for (v,w) ∈ E in dieser Reihenfolge
6    if UF.find(v) ≠ UF.find(w) then
7      UF.union(v, w)
8      add (v,w) to MST
9    end
10 end
```

Das Durchlaufen der Kanten aufsteigend nach Gewicht kann z.B. über eine `minPQ` implementiert werden. Der Ablauf der Kantenauswahl mit Kruskals Algorithmus ist in Abbildung 10.4 dargestellt und sollte mit dem Ablauf bei Prim in Abbildung 10.3 verglichen werden.

Korrektheit des Kruskal Algorithmus

Der Algorithmus von Kruskal in Listing 10.4 bestimmt den minimalen Spannbaum des gegebenen Graphen.



(a) (b) (c) (d)

Abbildung 10.4 Ablauf von Kruskals Algorithmus zur Erstellung eines Minimalen Spannbaums: Die Kanten werden aufsteigend nach ihrem Gewicht (hier = Länge) durchlaufen. Sie werden markiert (blau eingefärbt), falls sie mit den zuvor markierten Kanten keinen Zyklus bilden. Am Ende bilden die markierten Kanten den Minimalen Spannbaum. (Die Markierung eines Startknotens in den Abbildungen ist für den Kruskal Algorithmus irrelevant.)

Beweis. Die Korrektheit folgt aus dem Beweis zum allgemeinen Ansatz, sobald wir gezeigt haben, dass Kruskals Algorithmus dem allgemeinen Ansatz folgt, also immer eine minimale kreuzende Kante auswählt.

Jede Kante (v, w), die hinzugefügt wird, ist eine kreuzende Kante des Schnittes, der durch alle mit v verbundenen Knoten, also die v beinhaltende Zusammenhangskomponente von Union-Find, definiert wird. Diese Kante (v, w) ist tatsächliche eine *kreuzende* Kante, da v zu der Menge gehört und w nicht, da *UF.find(v)* $\neq$ *UF.find(w)* durch die Bedingung in Zeile 5 sichergestellt ist. Unter den kreuzenden Kanten hat (v, w) minimales Gewicht, da alle anderen kreuzenden Kanten später in der Sortierung vorkommen.

Durch die sehr effiziente Laufzeit der Operationen der Union-Find Struktur wird die Laufzeit des Kruskal Algorithmus nur von dem sortierten Durchlaufen der Kanten dominiert:

Laufzeit und Speicherbedarf des Kruskal Algorithmus

Der Algorithmus von Kruskal in Listing 10.4 bestimmt den minimalen Spannbaum in einer *worst-case* Laufzeit in $\boldsymbol{O}(E \log E)$ und einem Speicherbedarf in $\boldsymbol{O}(E)$.

Beweis. Wir gehen von einer Implementation mittels einer Vorrangewarteschlange (`minPQ`) für das nach Gewicht sortierte Durchlaufen der Kanten aus.

- Die Initialisierung der Union-Find Struktur in Zeile 1 benötigt $\boldsymbol{O}(V)$ und das Einfügen der Kanten in eine PQ benötigt $\boldsymbol{O}(E)$.
- Die *for*-Schleife wird für jede Kante einmal durchlaufen. Also benötigen die `poll()` Operationen der `minPQ` insgesamt eine Laufzeit in $\boldsymbol{O}(E \log E)$.
- Durch die effiziente Union-Find Implementation fallen die `union()` und `find()` Operationen nicht ins Gewicht.
- Der Speicherbedarf ist proportional zu V für Union-Find und zum Speichern des Spannbaums in MST, und für die PQ ist er proportional zu E.

Da eine effiziente Bestimmung von Minimalen Spannbäumen für viele praktische Anwendungen äußerst wichtig ist, wurde einige Forschung in dieser Richtung betrieben. Hier sind ein paar der Weiterentwicklungen aufgeführt:

- [Fredman and Tarjan, 1987]: Prim mit Fibonacchi Heap: $\boldsymbol{O}(E + V \log V)$
- [Chazelle, 1997,Chazelle, 2000]: Laufzeit sehr nah an $\boldsymbol{O}(E)$, aber für die Praxis zu kompliziert.
- [Karger et al., 1995]: Randomisierter Algorithmus mit linearer Laufzeit im Erwartungswert.

Bemerkung zur Historie

Borůvka hat in seinem Aufsatz wesentliche Ideen für die Algorithmen von Prim und Kruskal bereits 1929 in seinem Artikel erwähnt. Weitere Grundlagen für den Prim Algorithmus wurden von Jarnik 1939 und von Kruskal 1956 veröffentlicht. Prims Veröffentlichung ist von 1961. Bei der Einschätzung der frühen Ansätze ist zu berücksichtigen, dass damals viele der effizienten Datentypen wie Vorrangwarteschlangen noch nicht bekannt waren. Daher war sowohl der Entwurf der Algorithmen, als auch eine effiziente Implementierung deutlich schwieriger.

10.6 Das Wichtigste in Kürze

- Ein Spannbaum eines Graphen ist ein Teilgraph $\boldsymbol{T}$, der alle Knoten des Graphen umfasst und die folgenden äquivalenten Bedingungen erfüllt:

  1. $\boldsymbol{T}$ ist maximal azyklisch: Das Hinzufügen einer beliebigen Kante erzeugt einen Zyklus.
  2. $\boldsymbol{T}$ ist minimal zusammenhängend: Das Entfernen einer beliebigen Kante macht ihn unzusammenhängend.

- In diesem Kapitel geht es um die Bestimmung eines Spannbaumes eines gewichteten (ungerichteten) Graphen mit minimalem Gewicht.
- Kruskal Algorithmus ist an Bedingung 1 orientiert. Die Kanten werden nach aufsteigendem Gewicht durchlaufen und ausgewählt, wenn sie keinen Zyklus erzeugen.
- Der Reverse-Delete Algorithmus ist an Bedingung 2 orientiert. Er durchläuft die Kanten nach absteigendem Gewicht und löscht sie, falls dadurch der Graph nicht unzusammenhängend wird.
- Der Algorithmus von Prim hat eine leichte Ähnlichkeit mit der Breitensuche. Ausgehend von einem Startknoten wird ein markierter Bereich über eine kreuzende Kante immer weiter ausgedehnt.
  - Bei der Breitensuche wird derjenige Zielknoten als nächstes besucht, der sich schon am längsten in der Warteschlange befindet.
  - Bei Prim wird derjenige Knoten als nächstes besucht, der über die Kante mit geringstem Gewicht zu erreichen ist.
- Obwohl Prims und Kruskals Algorithmus in der Reihenfolge der Kantenwahl sehr unterschiedlich vorgehen (vergleiche Abbildungen 10.3 und 10.4), sind sie beide Spezialfälle eines allgemeinen Greedy Ansatzes.
- Bei der Implementierung von Prim führt eine einfache Verwendung der PQ auf eine Laufzeit in $\boldsymbol{O}(E \log E)$. Mit Hilfe einer indizierten PQ kann die Laufzeit auf $\boldsymbol{O}(E \log V)$ reduziert werden.
- Für die effiziente Implementierung des Kruskal Algorithmus ist die Benutzung der Union-Find zur dynamischen Verwaltung von Zusammenhangskomponenten essentiell.

Quellenverweise

Große Teile dieses Kapitels orientieren sich an [Sedgewick and Wayne, 2014], insbesondere die Java Implementationen. Bei der Darstellung des Kruskal Algorithmus wurde zusätzlich [Cormen et al., 2013] zu Rate gezogen.

KAPITEL 11

Kürzeste Wege in gewichteten Graphen

11.1 Einleitung zu kürzesten Wegen

In Abschnitt 9.4 haben wir mit Breitensuche Wege von einem Startknoten zu allen erreichbaren Knoten bestimmt, die die *minimale Anzahl von Kanten* benutzen. Dies ist eine Art von Aufgabe, die in vielen Anwendungsbereichen vorkommt. Allerdings wird meist eine allgemeinere Form benötigt. Für die Navigation verwendet man z. B. Graphen, um Straßenkarten zu modellieren. Dabei stellen die Knoten Kreuzungspunkte und die Kanten Streckenabschnitte dar. Eine typische Aufgabe in der Navigation ist es, *kürzeste Wege* im Sinne der Gesamtstrecke oder Gesamtfahrzeit zu bestimmen. Mit der Breitensuche finden wir Wege, die die geringste Anzahl von Streckenabschnitten benutzt. Durch eine Modifikation des Graphen könnte man eine ungefähre Lösung mit Breitensuche erzielen. Dazu legt man eine Auflösungsgenauigkeit fest, sagen wir 1 Meter, und ersetzt dann jede Kante durch viele einfach verbundene Kanten, wobei die Anzahl der Kanten der Länge des Streckenabschnittes in Metern entspricht. Die Lösung mit Breitensuche in diesem modifizierten Graphen findet die Wege mit der geringsten Anzahl von 1 Meter Abschnitten, was bis auf Rundungsfehler den kürzesten Wegen entspricht. Diese Art das Problem anzugehen, hat allerdings einen Haken. Die Anzahl der Kanten des Graphen vergrößert sich enorm, besonders bei höherer Auflösungsgenauigkeit, und darunter leidet die Laufzeit.
Daher wird üblicherweise ein anderer Weg beschritten, der eleganter und effizient ist und immer das exakte Ergebnis liefert. Dazu wird jeder Kante des Graphen ein Gewicht zugeordnet, wie in dem vorigen Kapitel über minimale Spannbäume. Die Aufgabe besteht dann darin, den Weg mit dem kleinsten Gesamtgewicht (= Summe der Gewichte aller benutzten Kanten) zu finden. Auf

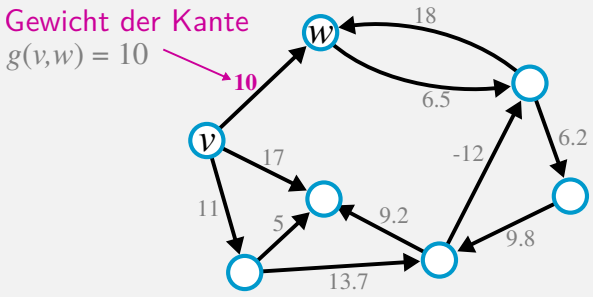
diese Weise können z. B. die Längen der Kanten oder die Kosten für Wegstrecken (etwa bei Flugreisen) berücksichtigt werden.
Die Algorithmen für kürzeste Wege werden üblicher Weise für *gerichtete* Graphen formuliert, einige der Algorithmen funktionieren auch gar nicht auf ungerichteten Graphen. Daher führen wir nun die *gewichteten, gerichteten* Graphen, also die gewichteten Digraphen ein.

11.1.1 Kantengewichtete Digraphen

Nach den gerichteten und den gewichteten Graphen werden nun die gewichteten, gerichteten Graphen eingeführt. Bei gewichteten Graphen können sowohl die Kanten als auch die Knoten mit Gewichten versehen werden. Wir verwenden *kanten*gewichtete Digraphen. Bei Problemen, die für knotengewichtete Digraphen formuliert sind, können die Kantengewichte so definiert werden, dass sich die Lösungen für knoten- und kantengewichtete Digraphen gleichen. Die formale Definition gewichteter Digraphen vereint die Aspekte von gerichteten und gewichteten Graphen:

Kantengewichtete Digraphen

Ein **gewichteter Digraph** ist ein Tupel $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E}, g)$, wobei $(\boldsymbol{V}, \boldsymbol{E})$ ein Digraph und $g : \boldsymbol{E} \to \mathbb{R}$ eine Gewichtsfunktion ist. Wir schreiben $g(v{\to}w)$, bzw. $weight(v{\to}w)$ für das Gewicht des Pfeils $v{\to}w$. Negative Gewichte sind erlaubt, wenn sie nicht explizit ausgeschlossen werden.



Für die Implementation `EdgeWeightedDigraph` der kantengewichteten Graphen wird zunächst die Klasse `DirectedEdge` für gewichtete, gerichtete Kanten[1] eingeführt. Objekte vom Typ `EdgeWeightedDigraph` enthalten Startknoten, Endknoten, sowie das Gewicht. Die Adjazenzlisten enthalten für jeden Knoten die ausgehenden Kanten. Dabei ist die Information des Startknotens redundant. Dies wird in Kauf genommen, da es die Implementierung mancher Algorithmen vereinfacht.

[1] Wir übernehmen den Klassennamen aus [Sedgewick and Wayne, 2014]. Da für ungewichtete, gerichtete Kanten keine eigene Klasse eingeführt wurde, gibt es keine Kollision von Klassennamen.

API für einen kantengewichteten Digraphen

In der API des Paares `DirectedEdge` / `EdgeWeightedDigraph` liegt der einzige Unterschied zu dem Paar `Edge` / `EdgeWeightedGraph` darin, dass bei der `DirectedEdge` zwischen Start- und Endknoten unterschieden wird. Man erhält sie durch die Methoden `from()` und `to()`. Dagegen hat `Edge` nur die unspezifischen Methoden `either()` und `other(v)`, mit denen man einen beliebigen der beiden Knoten bzw. den anderen der beiden Knoten erhält.

API für eine gewichtete gerichtete Kante

public class DirectedEdge implements Comparable<DirectedEdge>

	`DirectedEdge(int v, int w, double weight)`	Kante `v->w` mit Gewicht `weight`
`double`	`weight()`	Gewicht der Kante
`int`	`from()`	Startknoten dieser Kante
`int`	`to()`	Endknoten dieser Kante
`int`	`compareTo(DirectedEdge that)`	Vergleicht die Kantengewichte

API für einen kantengewichteten Digraphen

public class EdgeWeightedDigraph

	`EdgeWeightedDigraph(int V)`	Erzeugt leeren Digraphen mit `V` Knoten
`int`	`V()`	Anzahl der Knoten
`int`	`E()`	Anzahl der Kanten
`void`	`addEdge(DirectedEdge e)`	Füge Kante `e` zum Digraphen hinzu
`Iterable<DirectedEdge>`	`incident(int v)`	Kanten inzident zu `v`
`Iterable<DirectedEdge>`	`edges()`	Alle Kanten dieses Digraphen

Kürzeste Wege in gewichteten Digraphen

Nach diesen Vorbereitungen formulieren wir die Aufgabe, der sich dieses Kapitel widmet:

Kürzeste Wege von einem Startknoten (*single-source shortest paths*; SSSP)

Finde in einem kantengewichteten Digraphen von einem Startknoten zu allen anderen Knoten jeweils einen Weg mit geringstem Gewicht.

Dabei ist das Gewicht eines Weges als die Summe der Gewichte seiner Kanten definiert. Die Begriffe 'Länge' und 'Gewicht' werden im Zusammenhang von Wegen und gewichteten (gerichteten) Graphen synonym verwendet, also *kürzester Weg = Weg mit geringstem Gewicht*.

Von diesem Problem gibt es mehrere Varianten, wie zum Beispiel

- Finde einen kürzesten Weg von einem Startknoten s zu einem Zielknoten t.
- Finde zwischen allen Paaren von Knoten jeweils einen kürzesten Weg (*all-pairs shortest paths*; APSP).

Beide Aufgaben können zwar durch leichte Abwandlungen mit den in diesem Kapitel vorgestellten SSSP Algorithmen gelöst werden, jedoch nicht sonderlich effizient. Das erste Problem mit gegebenem Start- und Zielknoten wird in dem Kapitel über Heuristische Algorithmen ausführlich besprochen. Für das APSP Problem gibt es eine effiziente Lösung mit Dynamischer Programmierung (Floyd-Warshall Algorithmus), die allerdings in diesem Skript nicht vorkommt.
Für das single-source shortest paths Problem gibt es unterschiedliche Algorithmen, welche davon geeignet und effizient für ein gegebenes Problem sind, hängt davon ab, wie der Graph beschaffen ist. Wir betrachten die folgenden Fälle:

- Der Graph ist azyklisch (DAG).
- Die Kantengewichte sind nicht-negativ.
- Die Gewichte sind beliebig und der Graph besitzt keine negativen Zyklen.

Die Frage nach kürzesten Wegen ist nur dann sinnvoll, wenn man auf dem Weg von Start- zu Endknoten keine *negativen Zyklen* (= Zyklen mit negativem Gesamtgewicht) passieren kann. Dann würde jede zusätzliche Runde durch diesen Zyklus das Gewicht des Weges verringern, es gäbe also keinen kürzesten Weg. Wir setzen im Folgenden generell voraus, dass die **Graphen keine negativen Zyklen** besitzen.

11.2 Single-Source Shortest Paths

Baum der kürzesten Wege (*Shortest Paths Tree, SPT*)

Zunächst überlegen wir in welcher Datenstruktur die Information für die kürzesten Wege gespeichert werden soll. Das Gesamtgewicht bzw. Länge jedes kürzesten Weges zu einem Zielknoten t kann in einem knotenindizierten Array `dist` unter Index t gespeichert werden. Für die Speicherung der Wege hilft uns der folgende Satz.

Die SSSP Wege können als Baum dargestellt werden

Die kürzesten Wege von einem Startknoten können als ein Baum repräsentiert werden. Falls es zu einem Zielknoten mehrere kürzeste Wege gibt, wird nur einer repräsentiert.

Dieser Baum wird **Baum der kürzesten Wege** (*Shortest Paths Tree*, **SPT**) genannt.

Beweis. Es ist zu zeigen, dass im Untergraphen der kürzesten Wege Zyklen vermieden werden können. Nehmen wir an, dass es einen Zyklus im Untergraphen gibt. Dies kann kein gerichteter Zyklus sein, da darin ein Pfeil keinen Beitrag zu einem kürzesten Weg haben kann, siehe Abbildung 11.1 (a).
In dem ungerichteten Zyklus muss es einen Knoten geben, auf den zwei Pfeile zeigen. Beide Pfeile müssen dasselbe Gewicht haben. Einer der beiden Pfeile kann entfernt werden. Damit ist auch der Zyklus entfernt, siehe Abbildung 11.1 (b).

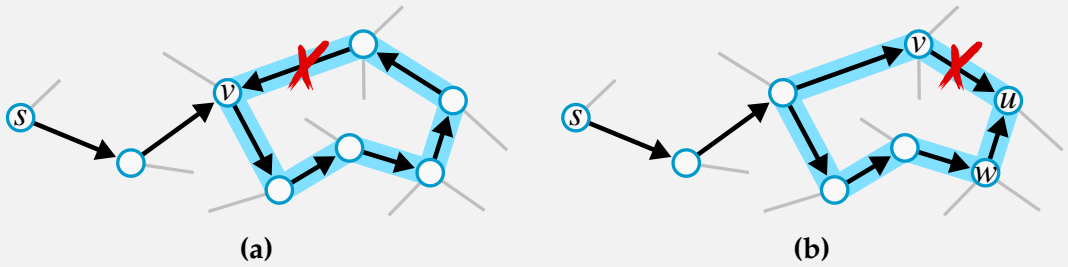


Abbildung 11.1 Kürzeste Wege (SSSP) können als Baum repräsentiert werden: Es ist zu zeigen, dass Zyklen nicht auftreten, bzw. vermieden werden können. **(a)** Ein gerichteter Zyklus kann nicht auftreten: Ein Pfeil hätte keinen Beitrag zu kürzesten Wegen, und zwar derjenige, der auf einen Knoten v zeigt, zu dem ein Pfad vom Startknoten s aus führt. **(b)** Ungerichtete Zyklen können vermieden werden: Wenn der Zyklus nicht gerichtet ist, muss es (mindestens) zwei Pfeile geben, die auf denselben Knoten u zeigen. Sie müssen dasselbe Gewicht besitzen. (Sonst könnte derjenige mit größerem Gewicht keinen Beitrag zu kürzesten Wegen haben.) Dann reicht es einen der beiden Pfeile für kürzeste Wege zu benutzen. Alle kürzesten Wege, die über $w \to u$ laufen, können über $v \to u$ mit demselben Gewicht erreicht werden. **Bemerkung:** Die grauen Striche sollen weitere Kanten andeuten, die hier keine Rolle spielen.

Die gerade gezeigte Tatsache erleichtert die Implementation der SSSP Algorithmen: Wir können die kürzesten Wege speicher-effizient als Baumstruktur in einem knotenindizierten Array `parent` speichern, wie bei der Tiefen- und Breitensuche in dem Kapitel über ungewichtete Graphen.
Außer den kürzesten Wegen, muss zu jedem Knoten auch die Länge des kürzesten Weges gespeichert werden. Dafür verwenden wir das Array `dist`. Bei manchen Algorithmen wird nicht jeder Knoten direkt auf dem kürzesten Wege erreicht. Es können im Laufe der Suche kürzere Wege gefunden werden. Daher ist es präziser, bei dem Ablauf des Algorithmus von einem kürzesten der aktuell bekannten Wege zu sprechen. Die Bedeutung der Variablen ist also: In `dist[v]`

wird die Länge eines kürzesten, aktuell bekannten Weges von `s` nach `v` gespeichert und in `parent[v]` der Vorgänger auf diesem Weg.
Damit haben wir das Grundgerüst für alle SSSP Algorithmen:

```
public class SingleSourceShortestPaths {
  private double[] dist;   // Länge eines aktuell kürzesten Weges, oder
                           // POSITIVE_INFINITY, falls keiner bekannt
  private int[] parent;    // Vorgänger im Baum der aktuell kürzesten Wege

  public double dist(int v)   { return dist[v]; }
  public boolean hasPathTo(int v) {
    return dist[v] < Double.POSITIVE_INFINITY;
  }
  public Iterable<int> pathTo(int v)  // wie zuvor in Listing 9.6
}
```

Relaxation

Die meisten Algorithmen benutzen eine einfache Grundregel, die *Relaxation*: Wenn eine Kante einen kürzeren Weg ermöglicht, nutze sie! Zunächst wird diese Relaxierungsregel als *lokaler* Grundbaustein für die Suche kürzester Wege unter die Lupe genommen. Danach folgen die unterschiedlichen Algorithmen für das SSSP Problem, die im Prinzip aus der iterierten Anwendung der Regel bestehen. Wir schreiben $d(v)$ für die Länge des aktuell kürzesten Weg zum Knoten v (bzw. `dist[v]` im Java und Pseudocode) und $g(v \to w)$ oder $weight(v \to w)$ für das Gewicht der Kante $v \to w$ (bzw. `e.weight()` im Java Code, wenn `e` die Kante $v \to w$ ist). Für einen Pfad $v \rightsquigarrow w$ schreiben wir $weight(v \rightsquigarrow w)$ für das Gesamtgewicht des Pfades, also für die Summe der Kantengewichte.

Relaxation einer Kante

Falls auf der Suche nach kürzesten Wegen für eine gewichtete Kante $v \to w$ die **Relaxierungsbedingung**

$$d(w) > d(v) + weight(v \to w)$$

erfüllt ist, wird $d(w)$ auf $d(v) + weight(v \to w)$ aktualisiert und v als neuer Vorgänger von w in dem Baum kürzester Wege gespeichert. Der kürzeste Weg nach w ist nun also der kürzeste Weg nach v plus die Kante $v \to w$. Diesen Vorgang nennt man die **Relaxierung** der Kante $v \to w$. Er ist in Abbildung 11.2 illustriert.
Inhaltlich bedeutet die Relaxierungsbedingung, dass der neu entdeckte Weg über v nach w kürzer ist, als der kürzeste zuvor bekannte Weg nach w.

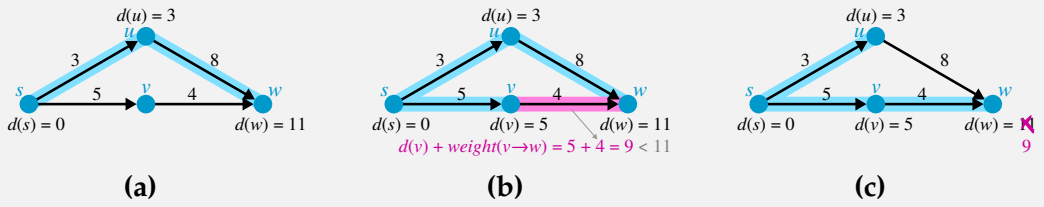


Abbildung 11.2 Relaxation einer Kante: Es ist ein kleiner gewichteter Digraph gegeben, in dem die kürzesten Wege ausgehend von s gesucht sind. **(a)** Zunächst wurden die Knoten u und w besucht und dadurch ein Weg der Länge 11 nach w gefunden. **(b)** Anschließend wird von s aus v besucht und von dort aus w entdeckt. Der Knoten w wurde zwar schon vorher entdeckt, aber der neue Weg ist kürzer: Es ist die Relaxationsbedingung $d(w) > d(v) + weight(v{\to}w)$ erfüllt. **(c)** Daher wird nun v als neuer Vorgänger von w in dem Baum der kürzesten Wege gespeichert, und der `dist` Wert von w entsprechend auf 9 aktualisiert.

Der Ursprung des Begriffs *Relaxation* (Entspannung) liegt in der Vorstellung eines gespannten Gummis von `s` über `u` nach `w`, das *entspannt* wird, wenn es den kürzeren Weg über `v` anstatt über `u` geleitet wird (siehe Abbildung 11.2). Es ist also eher eine *Entspannung über die* Kante $v{\to}w$, als eine *Entspannung der Kante* $v{\to}w$.
Mit der Erfüllung der Relaxationsbedingung für jede *einzelne Kante* kann die Kürzeste-Wege-Eigenschaft für den *ganzen Digraphen* sichergestellt werden. Dies ebnet den Weg für Greedy Algorithmen, die immer einen Schritt mit lokaler Verbesserung auswählen und dadurch am Ende ein globales Optimum erreichen.

Bedingung für kürzeste Wege

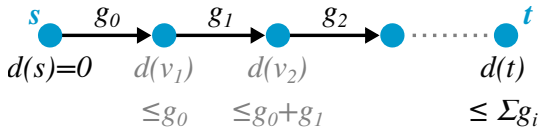
Für jeden Knoten v eines kantengewichteten Digraphen sei $d(v)$ die Länge eines Weges von s nach v. Dann gilt die folgende Äquivalenz:
Die Werte $d(v)$ sind die Längen kürzester Wege von Knoten s, genau dann wenn $d(v) + weight(v{\to}w) \geq d(w)$ für jede Kante $v{\to}w$ gilt.

Beweis. "$\Rightarrow$"

Wir nehmen an, dass die letzte Bedingung *nicht* stimmt und zeigen, dass dies zu einem Widerspruch mit der Voraussetzung an $d(v)$ führt. Es gibt also eine Kante $v{\to}w$ mit Relaxierungsbedarf: $d(v) + weight(v{\to}w) > d(w)$. Dann hat der Weg von s über v und $v{\to}w$ nach w eine Länge $d(v) + weight(v{\to}w)$, die kleiner als $d(w)$ ist. Dies steht im Widerspruch dazu, dass die Werte $d(\cdot)$ die Längen der kürzesten Wege sind. □

Beweis. "$\Leftarrow$"

Wir betrachten einen kürzesten Weg von s nach t: $v_0\ (= s), v_1, \ldots, v_K\ (= t)$. Aus $d(w) \leq d(v) + weight(v \to w)$ für alle Kanten des Weges folgt: $d(t) \leq \sum_{i<K} g_i$, siehe Skizze, wobei g_i als $weight(v_i, v_{i+1})$ definiert ist. Die Summe über g_i ist die Länge des kürzesten Weges. Also muss $d(t)$ gleich dieser Länge sein, was zu zeigen war. □



Die gerade bewiesene Bedingung wird in den Korrektheitsbeweisen in diesem Kapitel benutzt, zunächst für einen allgemeinen Ansatz und dann für drei wichtige SSSP Algorithmen, die Spezialfälle des allgemeinen Ansatzes sind.

Listing 11.1 Allgemeiner Ansatz für kürzeste Wege durch Relaxierung. In jedem Schritt ist $dist[v]$ die Länge eines einfachen Weges von s nach v, oder inf, falls v (noch) nicht erreicht wurde. Für erreichte Knoten ist $parent[v]$ der Vorgänger von v auf jenem Weg.

```
1  // dist: Array der Größe V
2  // parent: Array der Größe V
3  for all nodes v ∈ V
4     dist[v] ← ∞
5  end
6  dist[s] ← 0
7  while exists edge v→w ∈ E with dist[w] > dist[v] + weight(v→w) do
8     parent[w] = v                          // Relaxierung der
9     dist[w] = dist[v] + weight(v→w)        // Kante v→w
10 end
```

Nun stellt sich die Frage, ob mit diesem Ansatz immer zu jedem erreichbaren Knoten des Digraphen kürzeste Wege gefunden werden. Die Antwort ist ja und nein. Sie werden immer gefunden, wenn es sie überhaupt gibt, also wenn die Frage nach kürzesten Wegen überhaupt sinnvoll ist. Schwierigkeiten machen *negative Zyklen*, also Zyklen bei denen die Summe der Kantengewichte negativ ist. Wenn der Digraph negative Zyklen hat, die von s erreichbar sind, ergibt jeder weitere Durchlauf des Zyklus ergibt einen Weg mit geringerem Gewicht. Wir beschränken uns im Folgenden auf Digraphen ohne negative Zyklen. In diesem Fall sind kürzeste Wege immer *einfache* Wege, die also keine Knoten mehr als einmal besuchen.

Korrektheit und Terminierung des allgemeinen SSSP Ansatzes

Der allgemeine Ansatz zur Bestimmung aller kürzesten Wege ausgehend von einem Startknoten s in einem gewichteten Digraphen ohne negative Zyklen in Listing 11.1 terminiert mit dem korrekten Ergebnis.

Beweis der Terminierung: In jedem Schritt wird ein d Wert verkleinert. Es kann nur endlich viele Schritte geben, da ein d-Wert immer der Länge eines einfachen Pfades entspricht (siehe oben), und es nur endlich viele verschiedene einfache Pfade in einem Graphen gibt.

Beweis der Korrektheit: Nach Terminierung der `while` Schleife in Zeile 6 gilt für alle Kanten $v{\to}w$ die Bedingung $d(w) \leq d(v) + weight(v{\to}w)$. Somit folgt die Korrektheit aus der zuvor bewiesenen Bedingung für kürzeste Wege. □

Je nachdem welche weiteren Einschränkungen des Graphen bekannt sind, sind unterschiedliche Algorithmen geeignet, um kürzeste Wege zu finden:

- Azyklische Digraphen (DAG): mit Topologischer Sortierung
- Digraphen ohne negative Gewichte: Dijkstra Algorithmus
- Digraphen ohne negative Zyklen: Bellman-Ford Algorithmus

Bei Bellman-Ford sind also negative Kantengewichte zugelassen. Die Einschränkung auf Digraphen ohne negative Zyklen ergibt sich aus der Tatsache, dass ansonsten keine kürzesten Wege existieren, siehe oben. Der Dijkstra Algorithmus kann auch für Digraphen mit negativen Gewichten verwendet werden, allerdings ist dann die worst-case Laufzeit schlechter als bei Bellman-Ford.

11.3 Kürzeste Wege in gewichteten azyklischen Digraphen

Für die Aufgabe kürzeste Wegen in azyklischen Digraphen (DAG) zu bestimmen sind wir durch die Topologische Sortierung bestens vorbereitet. Das Verfahren geht so: Man nimmt eine topologische Sortierung des DAG und geht in der Reihenfolge dieser Sortierung bis zum Startknoten s. Von dort werden alle ausgehenden Kanten relaxiert. Dieser Prozess wird iterativ mit dem folgenden Knoten in der topologischen Reihenfolge fortgesetzt. Aufgrund der topologischen Sortierung gibt es keine Kanten, die zu einem Knoten zurückführen, dessen Kanten schon relaxiert wurden. Daher muss jede Kante nur einmal relaxiert werden, sodass das Verfahren sehr effizient ist.

Listing 11.2 Pseudocode zur SSSP mit Topologischer Sortierung

```
for each node v
  dist[v] ← ∞
end
dist[s] ← 0
determine a topological sorting
for each node v in topological order
  if node s was not encountered so far
    continue
  for each w with v→w ∈ E
    relax(v→w)
  end
end
```

```
procedure relax(v→w)
if dist[w] > dist[v] + weight(v→w)
  parent[w] = v;
  dist[w] = dist[v] + weight(v→w)
end
```

Korrektheit und Laufzeit des SSSP Algorithmus mit Topologischer Sortierung

Der SSSP Algorithmus mit topologischer Sortierung bestimmt kürzeste Wege in jedem gewichteten DAG von Startknoten s in einer Laufzeit in $\boldsymbol{O}(V + E)$

Beweis der Korrektheit:

Jede Kante $e = v \to w$ wird relaxiert, wenn der Knoten v gemäß der Sortierung an der Reihe ist. Die bei der Relaxation hergestellte Bedingung $dist[w] \leq dist[v] + g(v, w)$ bleibt im weiteren Verlauf gültig:

- $dist[v]$ bleibt unverändert, da alle Kanten mit v als Endknoten nach der topologischen Sortierung schon vorher bearbeitet worden sein müssen.
- $dist[w]$ kann durch weitere Relaxierungen nur kleiner werden.

Die Bedingung für kürzeste Wege ist also erfüllt und impliziert die Korrektheit.

Laufzeit:

- Jede Kante wird höchstens[a] einmal relaxiert, und *relax()* hat konstante Laufzeit.
- Also hat die Schleife eine Laufzeit in $\boldsymbol{O}(E)$.

- Die Initialisierung der Arrays benötigt $\boldsymbol{O}(V)$ und die topologische Sortierung $\boldsymbol{O}(V+E)$.
- Da diese drei Schritte nacheinander ausgeführt werden, ergibt sich für die Gesamtlaufzeit ebenfalls $\boldsymbol{O}(V+E)$.

[a] Kanten, die in der Sortierung vor s kommen, werden gar nicht relaxiert. Sie sind nicht von s erreichbar und daher irrelevant.

Anwendung des topologischen Algorithmus für kürzeste Wege

Mit Hilfe des gerade vorgestellten Algorithmus kann eine interessante Anwendung aus der Bildbearbeitung realisiert werden: Inhaltsabhängige Bildskalierung (*content-aware image resizing*) durch *seam carving*. Dies bezeichnet die Aufgabe ein Bild horizontal oder vertikal zu stauchen, ohne dass dadurch Verzerrungen entstehen oder Inhalt verloren geht, siehe Abbildung 11.3.

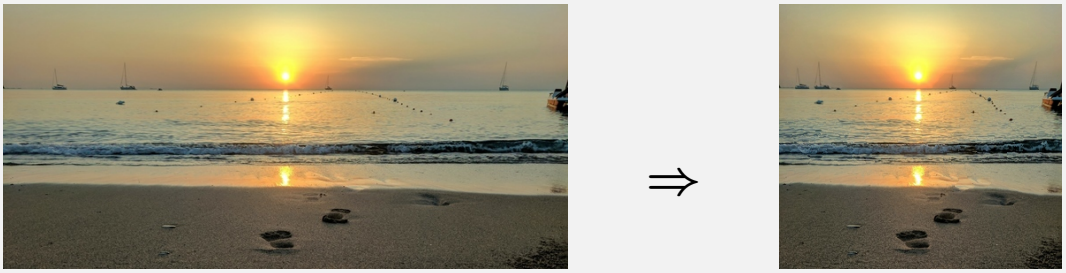


Abbildung 11.3 Inhaltsabhängige Bildskalierung: Das Bild auf der linken Seite wird gestaucht, ohne dass dabei Verzerrungen entstehen. Die Objekte werden näher zusammengerückt, indem Pfade von oberen zum unteren Rand mit möglichst wenig Kontrastübergang gelöscht werden.

Ein besonders effizienter Ansatz zur inhaltsabhängigen Bildskalierung kann mit Dynamischer Programmierung realisiert werden [Avidan and Shamir, 2007]. Hier wird die Aufgabe Graphen-basiert über den SSSP Algorithmus mit Topologischer Sortierung gelöst.
Jeder Pixel wird als Knoten in einem gewichteten Digraphen dargestellt, siehe Abbildung 11.4. Für vertikale Nähte gehen von jedem Knoten drei Kanten zu den unteren Nachbarn. Als Gewicht wird der Kontrast zwischen den verbundenen Pixeln genommen. Die Naht ist der kürzeste Pfad von oben nach unten. Die Pixel entlang der Naht werden gelöscht. Damit der Pfad bei einem beliebigen Knoten der obersten Zeile beginnen und bei einem beliebigen Knoten der untersten Zeile enden kann, fügt man zu dem Graphen zwei Knoten hinzu. Den

Startknoten verbindet man mit allen Knoten der obersten Zeile und zu dem Endknoten kommt man über Pfeile von allen Knoten der untersten Zeile. Mit dieser einfachen Methode wurde das in Abbildung 11.3 gezeigte Ergebnis erzielt. Für bessere Ergebnisse sollte auch die Nachbarschaft jedes Pixels berücksichtigt und mit Knotengewichten gearbeitet werden.

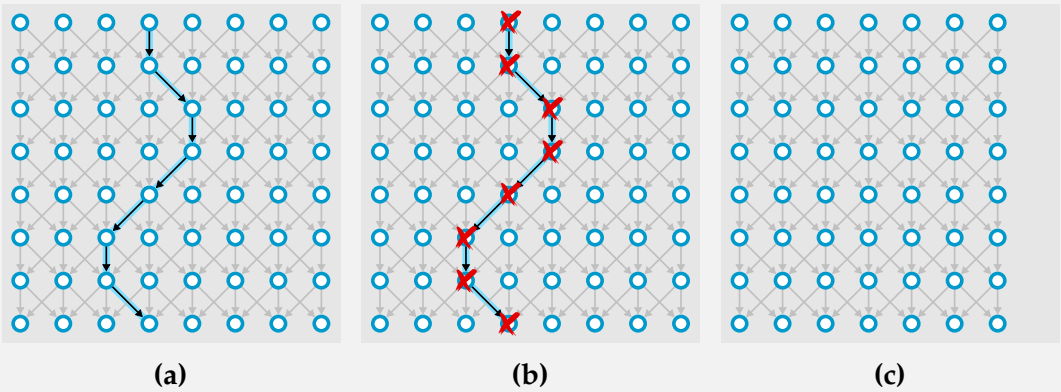


Abbildung 11.4 Graphen-basierter Ansatz für inhaltsabhängige Bildskalierung: Das Bild wird als ein DAG modelliert, bei dem jeder Knoten ein Pixel darstellt und mit den drei Nachbarknoten unter ihm (gerade und diagonal) verbunden ist. Die Kantengewichte werden z. B. als Kontrastunterschiede zwischen den Pixeln definiert. **(a)** Mit Topologischer Sortierung wird ein kürzester Weg von der oberen zu der unteren Bildkante gesucht. **(b)** Alle Pixel aus dem gefunden Pfad werden gelöscht. **(c)** So erhält man ein Bild, dass in horizontaler Richtung um ein Pixel schmaler ist als das Original.

11.4 Dijkstra Algorithmus für kürzeste Wege

Die Beschränkung auf azyklische Digraphen ist wesentlich für den Algorithmus mit topologischer Sortierung und stellt eine starke Einschränkung dar. Als nächste Stufe erlauben wir Zyklen, schränken aber alle Gewichte auf nicht-negative Werte ein. In diesem Fall liefert der **Dijkstra Algorithmus** eine effiziente Implementierung. Auch er folgt dem allgemeinen Ansatz, wählt aber eine andere Reihenfolge der Relaxierung.

Der Dijkstra Algorithmus baut einen SPT Suchbaum auf. Am Anfang ist nur der Startknoten s in dem Suchbaum. Wie bei der Tiefen- und Breitensuche wird in jedem Schritt ein Knoten über eine kreuzende Kante ausgewählt, also einem Pfeil, der von einem markierten in den unmarkierten Bereich zeigt.

Bei Dijkstra wird derjenige Knoten ausgewählt, der die kleinste Distanz von s hat. Die benutzte Kante wird relaxiert und der Knoten markiert. Der markierte Bereich muss nicht explizit gespeichert werden. Im folgenden Pseudocode ist M die Menge der Zielknoten von kreuzenden Kanten. M ist sozusagen der Rand des markierten Bereiches.

Listing 11.3 Pseudocode für den Dijkstra SSSP Algorithmus

```
1  // initialisiere 'dist' wie im allgemeinen Ansatz
2  M ←{s}
3  while M ≠∅
4    v ← element of M with minimum dist
5     delete v from M
6    for each w with v→w ∈ E
7      relax (v→w)
8    end                    // v fertig bearbeitet
9  end
10 procedure relax(v→w)
11 if dist[w] > dist[v] + weight(v→w)
12   parent[w] = v;
13   dist[w] = dist[v] + weight(v→w)
14   add w to M                    // w ist Zielknoten einer kreuzenden Kante
15 end
```

Es ist wichtig zu verstehen, wie bei dem Dijkstra Algorithmus die Kanten ausgewählt werden, die dem Baum der kürzesten Wege zugefügt werden. Diese Auswahl passiert in Zeile 4, wenn derjenige Knoten aus M mit dem kleinsten *dist* Wert selektiert wird. Da M die Zielknoten kreuzender Kanten enthält, wird

bei Dijkstra also diejenige kreuzende Kante ausgewählt, die zu dem Knoten mit der geringsten Distanz von s führt, siehe Abbildung 11.5.

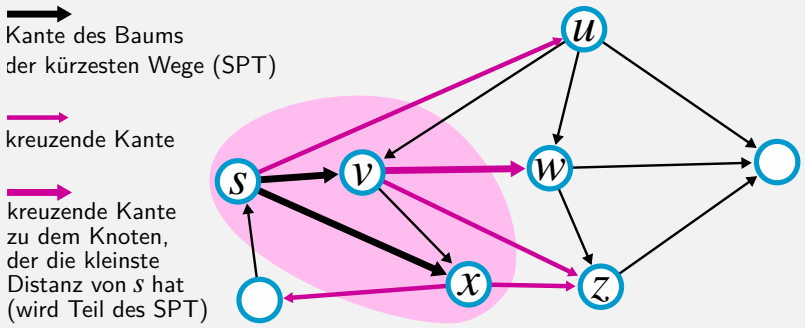


Abbildung 11.5 Auswahl der kreuzenden Kante im Dijkstra Algorithmus: Beim Dijkstra Algorithmus wird schrittweise dem Baum der kürzesten Wege (SPT) ein Knoten zugefügt (Zeile 4). Die Menge der Knoten, die schon im SPT sind, ist hier durch den pinken Bereich markiert. Als nächstes wird derjenige Knoten ausgewählt, der über eine kreuzende Kante (bezüglich des pinken Bereiches) erreicht werden kann, und der den kleinsten *dist* Wert hat. Hier wird also der Knoten w ausgewählt, der von allen Knoten außerhalb des pinken Bereiches den geringsten Abstand von s hat. Es wird *nicht* der Knoten z ausgewählt, der über die kürzeste der kreuzenden Kanten erreichbar ist. Dies ist das Auswahlkriterium bei dem Prim Algorithmus für Minimale Spannbäume. Es wird auch *nicht* der Knoten u ausgewählt, der über die geringste Anzahl von Kanten erreicht wird. Dies ist das Auswahlkriterium der Breitensuche.

Korrektheit des Dijkstra Algorithmus

Der Algorithmus von Dijkstra bestimmt kürzeste Wege ausgehend von Startknoten s in jedem gewichteten Graphen ohne negative Zyklen.

Beweis der Terminierung. Immer, wenn der Menge M ein Knoten hinzugefügt wird, verkleinert sich der *dist* Wert. Dies kann aus demselben Grund, wie in dem Beweis der Terminierung beim allgemeinen SSSP Ansatz, nur endlich oft passieren. Es werden also nur endlich oft Knoten M hinzugefügt. Da bei jedem Durchlauf der *while*-Schleife ein Knoten der Menge M entnommen wird, ist M irgendwann leer und der Algorithmus terminiert. □

Beweis der Korrektheit. Für jeden Knoten v, der von s erreichbar ist, werden alle ausgehenden Kanten $v \to w$ relaxiert. Dabei wird die Bedingung $dist[w] \leq$

$dist[v] + weight(v \to w)$ hergestellt. Sie ist auch bei der Terminierung gültig, da

- $dist[w]$ im Weiteren höchstens verringert wird und
- falls $dist[v]$ verringert wird (Zeile 13), wird v in M eingefügt (Zeile 14), also wird die Kante $v \to w$ nochmals relaxiert und die Bedingung wird wiederhergestellt. □

Effizienz des Dijkstra Algorithmus bei nicht-negativen Gewichten

Bei Graphen $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E}, g)$ mit nicht-negativen Gewichten wird jede Kante höchstens einmal relaxiert und jeder Knoten nur einmal (in Zeile 4) ausgewählt.

Beweis. Wir zeigen drei Behauptungen:

1 Die Distanzen, über die v in Zeile 4 ausgewählt wird, sind monoton steigend. Begründung:

- $dist[v]$ wurde als kleinster $dist$-Wert ausgewählt (Zeile 4).
- Alle Knoten u, die zu der Zeit in M sind, haben also einen größeren $dist$-Wert: $dist[u] \geq dist[v]$.
- Für Knoten w, die später in M eingefügt werden, gilt für ein solches u: $dist[w] = dist[u] + weight(u \rightsquigarrow w) \geq dist[v]$, da die Kantengewichte als nicht-negativ vorausgesetzt wurden.

2 Nachdem v in Zeile 4 ausgewählt wurde, wird es weder wieder in M eingefügt, noch wird sein $dist$ Wert geändert, da dies nur unter der Bedingung $dist[v] > dist[u] + weight(u \to v)$ für ein später ausgewähltes u passieren würde, was 1 widerspricht.

3 Jeder Pfeil $v \to w$ wird folglich nur einmal relaxiert. Die dabei hergestellte Bedingung $dist[w] \leq dist[v] + weight(v \to w)$ bleibt bis zur Terminierung gültig:

- $dist[w]$ kann durch weitere Relaxierungen nur kleiner werden.
- $dist[v]$ bleibt nach 2 unverändert.

Aus 3 folgt die erste und aus 2 folgt die zweite Aussage. □

Für die Implementation des Dijkstra Algorithmus bleibt zu entscheiden, welche Datenstruktur für M verwendet wird. Die kritische Operation ist das Entnehmen des kleinsten Elements bezüglich eines Schlüssels ($dist$). Daher wäre

eine Implementation mit einer `minPQ` (Vorrangwarteschlange) möglich. Die Schlüssel der PQ sind die Knoten, wobei die zugehörigen *dist* Werte die Ordnung (`comparable`) definieren. Allerdings muss berücksichtigt werden, dass die Distanzwerte (in *relax*) *verändert* werden. Das Ändern von Schlüsseln bzw. ihrer Ordnung ist, wie das Löschen, *nicht effizient* in Vorrangwarteschlangen. Daher hatten wir diese Methoden nicht in der API.

Für dieses Anforderungsprofil bietet sich die indizierte Vorrangwarteschlange an. Die *dist* Werte (Schlüssel) werden über Knoten indiziert.

Listing 11.4 Pseudocode für den Dijkstra Algorithmus mit Indizierter PQ

```
1  Q : IndexMinPQ
2  for each node v
3     dist[v] ← ∞
4  end
5  dist[s] ← 0
6  add(Q, s, 0)
7  while Q ≠ ∅
8     v ← poll(Q)
9     for each w with v→w ∈ E
10       relaxIndexPQ(v→w)
11    end
12 end
13 procedure relaxIndexPQ(v→w)
14 if dist[w] > dist[v] + weight(v→w)
15    parent[w] ← v
16    dist[w] ← dist[v] + weight(v→w)
17    if contains(Q, w)
18       decreaseKey(Q, w, dist[w])
19    else
20       add(Q, w, dist[w])
21    end
22 end
```

Laufzeit des Dijkstra Algorithmus mit IndexMinPQ

Die Implementation des Dijkstra Algorithmus mit einer indizierten Priority Queue hat eine Laufzeit in $\boldsymbol{O}(E \log V)$ für Graphen mit nicht-negativen Gewichten.

Beweis. Die maximale Länge der IndexPQ ist V. Daher haben *poll* (Zeile 8), *decreaseKey* (Zeile 18) und *add* (Zeile 20) eine Laufzeit in $\boldsymbol{O}(\log V)$.
Nach dem Satz zur Effizienz wird die *while* Schleife höchstens V-mal und die Relaxierung höchstens E-mal ausgeführt. Da die Relaxierung innerhalb der *while* Schleife ausgeführt wird, ergibt sich eine Gesamtlaufzeit in $\boldsymbol{O}(E \log V)$.

Eine noch höhere Effizienz kann mit einem *Fibonacci-Heap* erzielt werden. Damit lässt sich die Laufzeit des Dijkstra Algorithmus sogar im *worst-case* auf $\boldsymbol{O}(E + V \log V)$ senken. Allerdings ist die Implementierung eines Fibonacci-Heaps aufwändig, siehe [Cormen et al., 2013, S. 511ff].

Dijkstra Algorithmus bei negativen Gewichten

Der Korrektheitsbeweis wurde für den allgemeinen Fall, ohne die Beschränkung auf nicht-negative Gewichte geführt, d. h. der Dijkstra Algorithmus liefert auch in diesem allgemeinen Fall das richtige Ergebnis. Dennoch steht an vielen Stellen, dass er ein Algorithmus nur für Digraphen mit *nicht-negativen* Gewichten ist. Dies liegt an der Effizienz. Der Ablauf des Dijkstra Algorithmus ist in dem Fall von nicht-negativen Gewichten ein Greedy Algorithmus und hat eine garantiert gute Laufzeit. Jedes Mal, wenn Algorithmus 11.4 in Zeile 8 einen Knoten v aus der PQ holt, wird er dem Baum der kürzesten Pfade (mit der Kante, die in *parent[v]* gespeichert ist) hinzugefügt. Die Auswahl ist bei Digraphen mit nicht-negativen Gewichten gemäß dem Satz über die Effizienz des Dijkstra Algorithmus endgültig. Wenn es negative Gewichte gibt, gilt die Aussage im Allgemeinen nicht mehr, siehe Abbildung 11.6. Die Garantie der Effizienz des Dijkstra Algorithmus geht also verloren, wenn man negative Gewichte zulässt. Die *worst-case* Laufzeit ist dann sogar exponentiell [Erickson, 2019]!

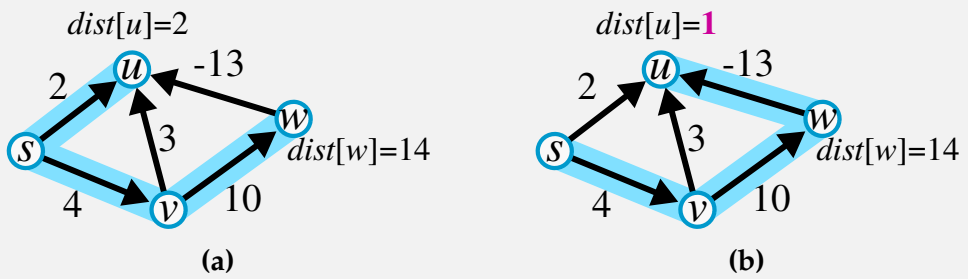


Abbildung 11.6 **Dijkstra ist bei negativen Gewichten nicht Greedy: (a)** Der Dijkstra Algorithmus besucht die Knoten zunächst in der Reihenfolge u, v, w. Von w aus wird ein neuer Weg nach u entdeckt, der kürzer ist, als der anfänglich benutzte. **(b)** Die Kante $s \to u$ muss aus dem SPT entfernt und $w \to u$ neu hinzugefügt werden. Dieses Beispiel zeigt, dass der Dijkstra Algorithmus bei Digraphen mit negativen Gewichten kein Greedy Verfahren ist.

11.5 Bellman-Ford: Kürzeste Wege in Digraphen mit beliebigen Gewichten

Die Beschränkung auf *nicht-negative* Gewichte ist wesentlich für die Effizienz des Dijkstra Algorithmus. Diese kommen allerdings bei einigen Problemstellungen vor. So werden z. B. bei der Ablaufplanung über negative Gewichte Vorbedingungen modelliert: Teilprojekt X muss 2 Monate vor Beginn von Teilprojekt Y beendet sein. Der kürzeste Weg ergibt die minimale Projektlaufzeit.) Wenn auch negative Gewichte erlaubt werden, ist es wichtig zu beachten, dass keine negativen Zyklen auftreten dürfen, siehe die Diskussion hier.

Für den Fall von Digraphen mit negativen Gewichten ist der **Bellman-Ford Algorithmus** geeignet. Auch er folgt dem allgemeinen Ansatz von S. 196.

Die Beschreibung des Algorithmus ist denkbar einfach:

Bellman-Ford Algorithmus für SSSP in Digraphen mit beliebigen Gewichten

Relaxiere $V - 1$ mal alle Kanten in beliebiger Reihenfolge.
Da nur relaxiert werden kann, wenn für den Anfangsknoten des Pfeils schon eine Distanz bestimmt wurde ($dist[v] < \infty$), wächst auch hier ein SPT schrittweise, ausgehend vom Startknoten s.

Listing 11.5 Pseudocode für den allgemeinen Bellman-Ford SSSP Algorithmus

```
1  // initialisiere 'dist' wie zuvor
2  for k ← 1 to V−1
3    for each node v
4      for each node w with v→w ∈ E
5        relax (v→w)
6      end
7    end
8  end
9
10 procedure relax(v→w)
11 if dist[w] > dist[v] + weight(v→w)
12   parent[w] = v;
13   dist[w] = dist[v] + weight(v→w)
14 end
```

Korrektheit des Bellman-Ford Algorithmus

Bei Anwendung des Bellman-Ford Algorithmus auf einen Digraphen ohne negative Zyklen gilt: Nach k Durchläufen der äußersten Schleife in Algorithmus 11.5 sind die kürzesten Wege für alle Knoten v bestimmt und in $dist[v]$ gespeichert, die maximal k Kanten von dem Startknoten s entfernt sind.

Beweis. durch Induktion nach k. Für $k = 0$ ist die Behauptung offensichtlich erfüllt.

Sei v_0 $(= s)$, $\ldots$, v_k $(= t)$ ein kürzester Weg von s nach t mit $k > 0$ Kanten. Gemäß Induktionsvoraussetzung (IV) ist nach Durchlauf $k-1$ der Wert $dist[v_{k-1}]$ die Länge des kürzesten Weges von s nach v_{k-1}. Es ist die Induktionsbehauptung (IB) zu zeigen: Nach einem weiteren Durchlauf ist $dist[v_k]$ die Länge des kürzesten Weges von s nach v_k.
Die Relaxierung der Kante $v_{k-1} \to v_k$ in Schritt k ergibt:

$$dist[v_k] \leq dist[v_{k-1}] + weight(v_{k-1} \to v_k)$$

Der Wert von $dist[v_k]$ kann nicht echt kleiner sein als die rechte Seite, da der gegebene Pfad als *kürzester* Weg angenommen wurde. Somit gilt Gleichheit und die IB wurde für Durchlauf k gezeigt. □

Laufzeit des allgemeinen Bellman-Ford Algorithmus

Der Bellman-Ford Algorithmus bestimmt die kürzesten Wege in einem Digraphen ohne negative Zyklen von einem Startknoten s in einer Laufzeit in $\boldsymbol{O}(EV)$.

Beweis: Alle E Kanten werden $V-1$ mal relaxiert. Die Relaxierung hat konstante Laufzeit. Die Schachtelung der beiden `for`-Schleifen in Zeilen 3 und 4 von Listing 11.5 kann als eine Schleife über alle E Kanten angesehen werden. □

Mit Hilfe des Bellman-Ford Algorithmus können negative Zyklen erkannt werden:

Erkennung von negativen Zyklen

In einem gewichteten Digraphen können negative Zyklen in einer Laufzeit in $\boldsymbol{O}(EV)$ erkannt werden.

Beweis: Man wendet den Bellman-Ford Algorithmus an. Wenn die Bedingung $dist[w] \leq dist[v] + weight(v{\to}w)$ für mindestens eine Kante *nicht* erfüllt ist, muss ein negativer Zyklus vorliegen, denn wenn der Digraph keine negativen Zyklen hätte, müsste nach dem Korrektheitssatz obige Bedingung für alle Kanten $v{\to}w$ erfüllt sein. Der Zyklus kann dann ausgehend von w über die in *parent* gespeicherten Kanten zurückverfolgt werden.
Die Überprüfung der Bedingung, sowie die Zurückverfolgung des Zyklus erfolgt in einer Laufzeit in $\boldsymbol{O}(E)$, daher ist die Gesamtlaufzeit in $\boldsymbol{O}(EV)$.

11.5.1 Effiziente Version des Bellman-Ford Algorithmus

Der Bellman-Ford Algorithmus hat eine Laufzeit in $\boldsymbol{O}(EV)$. Für den *worst case* gibt es keine schnellere Variante. Mit einer relativ kleinen und naheliegenden Änderung erhält man einen Algorithmus, der in der Praxis meist eine wesentlich kürzere Laufzeit besitzt.[2]
Die Änderung ist durch die einfache Beobachtung motiviert, dass nur diejenigen Kanten $v{\to}w$ relaxiert werden müssen, deren Anfangsknoten v im letzten Schritt eine Änderung des Distanzwertes $dist[v]$ erfahren hat. Konkret kann dies über eine Warteschlange implementiert werden, in die alle Knoten w gespeist werden, deren Distanzwert durch eine Relaxierung vermindert wurde. Allerdings sollten keine Knoten mehrfach in die Schlange gelangen. Um dies auszuschließen, verwenden wir ein boole'sches Array *onQ*, das speichert, welche Knoten sich in der Warteschlange befinden.

Listing 11.6 Pseudocode für den Bellman-Ford mit Warteschlange

```
1  Q : Queue
2  onQ : boolean array of size V
3  // initialisiere 'dist' wie zuvor
4  enqueue(Q, s)
5  onQ[s] ← true
6  while Q ≠ ∅
7    v = dequeue(Q)
8    onQ[v] ← false
9    for each node w with v→w ∈ E
10     relaxBF(v→w)
11   end
12 end
```

[2] Laut [Sedgewick and Wayne, 2014] liegt die Laufzeit in der Praxis meist in $\boldsymbol{O}(E+V)$, also wie in dem Algorithmus mit topologischem Sortieren in DAG.

```
13 procedure relaxBF(v→w)
14 if  dist[w] > dist[v] + weight(v→w)
15   parent[w] = v;
16   dist[w] = dist[v] + weight(v→w)
17   if !onQ[w]
18     enqueue(Q, w)
19     onQ[w] ←true
20   end
21 end
```

11.6 Unterschiede zwischen Dijkstra und Bellman-Ford

Die Algorithmen von Dijkstra und Bellman-Ford sind beide Konkretisierungen des Allgemeinen Ansatzes. Sie unterscheiden sich in der Reihenfolge, in der die Knoten des Digraphen besucht werden, siehe Abbildung 11.7:

- Bei dem Dijkstra Algorithmus werden die Knoten in der Reihenfolge ihrer Gewichtsdistanz (= Länge des kürzesten Weges definiert durch Kantengewichte) von s besucht.
- Bei dem Bellman-Ford Algorithmus werden die Knoten in der Reihenfolge ihrer Kantendistanz (= Anzahl der Kanten eines Weges zu ihnen) von s besucht.

Bei Bellman-Ford werden die Knoten zunächst nur über die Wege mit geringster Kantenanzahl besucht. Sobald allerdings zu einem Knoten ein Weg mit geringerem Gewicht über eine höhere Anzahl von Kanten gefunden wird, wird er und ggf. von ihm erreichbare Knoten erneut besucht. Dieses Wiedereintauchen in den schon bearbeiteten Bereich bedeutet, dass bei Bellman-Ford nicht immer kreuzende Kanten zu Auswahl neuer Knoten benutzt werden. Bei Dijkstra kann dieser Fall nur bei negativen Kantengewichten auftreten.
Abbildung 11.8 zeigt Schnappschüsse des Ablaufes der beiden Algorithmen bei der Anwendung auf zwei zufällige euklidische Graphen (Gewichte entsprechen den Längen der Kanten in der Ebene). Dabei tritt der Unterschied allerdings nicht so klar zum Vorschein. Um diesen besser hervorzuheben, wird in Abbildung 11.9 ein spezieller Graph verwendet.

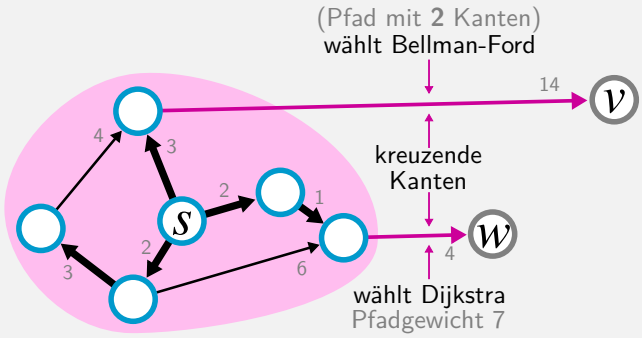
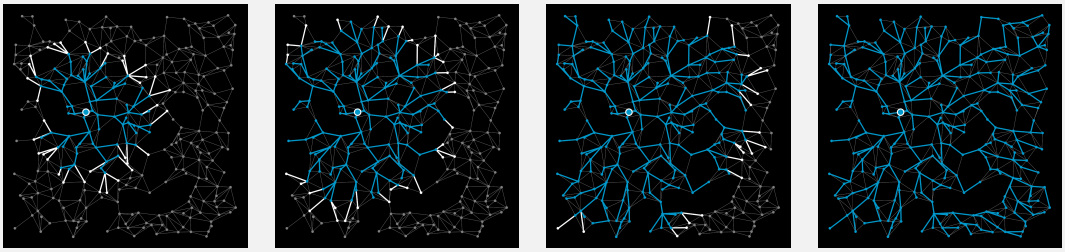
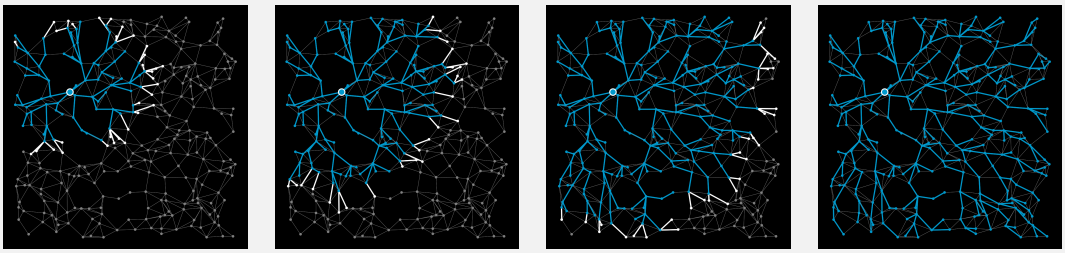


Abbildung 11.7 Reihenfolge der Kantenbearbeitung bei Dijkstra und Bellman-Ford: Beide Algorithmen wählen einen Knoten aus, der von den schon bearbeiteten über eine Kante erreichbar ist. Der Dijkstra Algorithmus wählt jeweils denjenigen Knoten aus, der die geringste Gewichtsdistanz von s hat, hier also Knoten w. Bei Bellman-Ford wird ein Knoten ausgewählt, der über die geringste Anzahl von Kanten zu erreichen ist[a]

[a] Exakt ausgedrückt: BF wählt einen Knoten, dessen zur Zeit kürzester bekannte Weg von s die geringste Anzahl von Kanten besitzt.

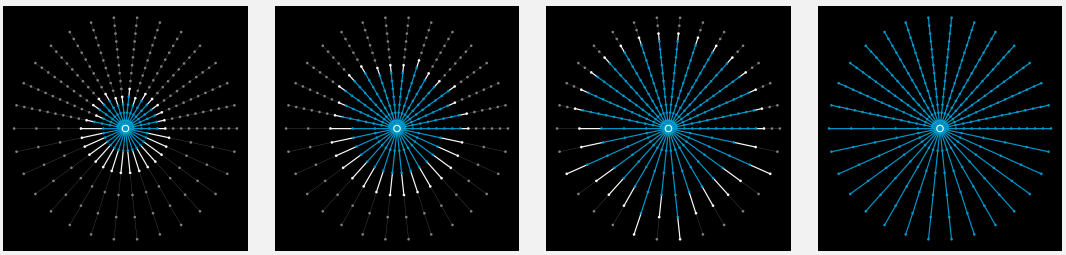


(a) Dijkstra Algorithmus

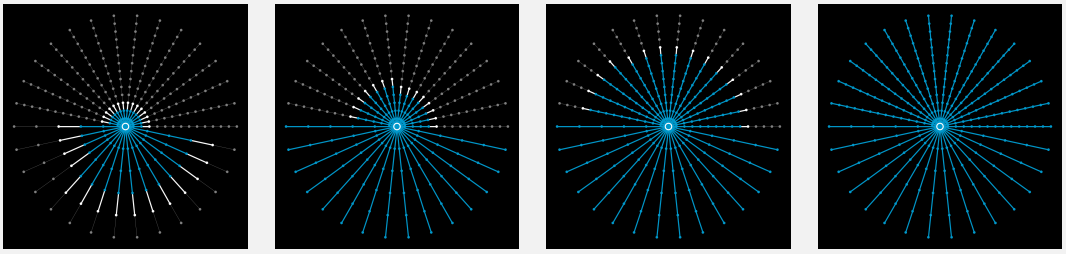


(b) Bellman-Ford Algorithmus

Abbildung 11.8 Ablauf der Algorithmen von Dijkstra und Bellman-Ford auf einem euklidischen Graphen. Kanten in der Warteschlange sind weiß dargestellt.



(a) Dijkstra Algorithmus



(b) Bellman-Ford Algorithmus

Abbildung 11.9 Ablauf der Algorithmen von Dijkstra und Bellman-Ford auf einem speziellen Sterngraphen. Die Kantengewichte sind als die euklidischen Abstände der Knoten definiert. In der oberen Hälfte sind in den Strahlen die Abstände zwischen den Knoten kürzer, als in der unteren Hälfte. Der Punkt in der Mitte ist als Startknoten gewählt. **(a)** Beim Dijkstra Algorithmus werden die Knoten gleichmäßig gemäß ihres Abstandes vom Mittelpunkt besucht. **(b)** Bellman-Ford geht nach Kantenanzahl vor und beendet daher die untere Hälfte schon viel früher, als die obere Hälfte.

11.7 Das Wichtigste in Kürze

In diesem Kapitel wurden drei Algorithmen für das Finden kürzester Pfade von einem Startknoten (SSSP) vorgestellt. Der Algorithmus mit topologischer Sortierung funktioniert nur für azyklische Digraphen (DAG), Dijkstra ist in erster Linie für Digraphen (und Graphen) mit nicht-negativen Gewichten und Bellman-Ford ist für alle Digraphen[3] geeignet. Die Kerneigenschften der drei Algorithmen sind in der folgenden Tabelle zusammengetragen:

	Topo. Sortierung	Dijkstra	Bellman-Ford
geeignet für	DAG	Graphen mit $g \geq 0$	Digraphen
wählt Knoten nach	topo. Sortierung	Gewichtsdistanz	Kantendistanz
Auswahl ist	Greedy	Greedy	nicht Greedy
besucht Knoten	einmal	einmal	ggf. mehrfach
Laufzeit Relax	$\boldsymbol{O}(1)$	$\boldsymbol{O}(\log V)$	$\boldsymbol{O}(1)$
Gesamtlaufzeit	$\boldsymbol{O}(V+E)$	$\boldsymbol{O}(E \log V)$	$\boldsymbol{O}(EV)$

In der Praxis ist der Bellman-Ford Algorithmus bei vielen Graphen deutlich effizienter, als die *worst-case* Laufzeit suggeriert. Der Dijkstra Algorithmus funktioniert auch für Digraphen mit negativen Kantengewichten, dann aber im *worst-case* deutlich ineffizienter als Bellman-Ford.

Quellenverweise

Weite Teile dieses Kaptiels orientieren sich an [Sedgewick and Wayne, 2014]. Die Idee inhaltsabhängige Bildskalierung Graphen-basiert zu lösen, stammt aus den Folien des Kurses *COS 226* (2012) von Kevin Wayne. Der Originalansatz stammt aus [Avidan and Shamir, 2007], siehe auch das Video unter `https://www.youtube.com/watch?v=vIFCV2spKtg`.

[3] Da die Aufgabenstellung nur für Digraphen ohne negative Zyklen sinnvoll gestellt ist, haben wir uns bei den SSSP Algorithmen generell auf solche beschränkt. Mit Hilfe des Bellman-Ford Algorithmus kann überprüft werden, ob es negative Zyklen gibt.

KAPITEL 12

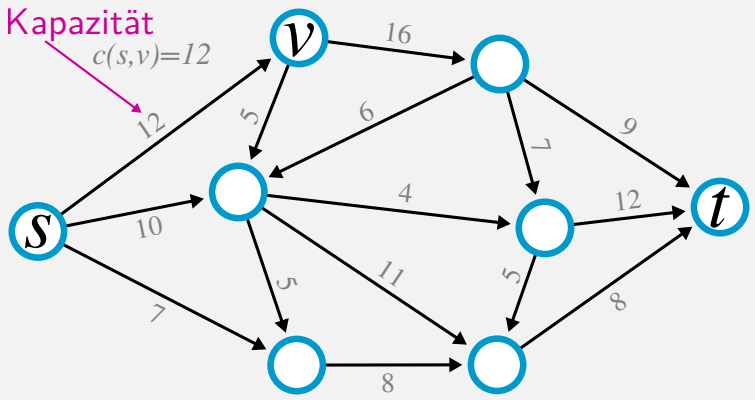
Flussgraphen

12.1 Flussgraphen

Mit Flussgraphen wird eine Dynamik in einem Netzwerk modelliert. Man stellt sich die Kanten am besten als Leitungen vor, durch die eine Flüssigkeit fließt. Ebenso kann z. B. der Fluss von Informationen durch Netzwerke modelliert werden. Das Hauptergebnis, das in diesem Kapitel erzielt wird, ist allerdings auch für 'normale' gewichtete Graphen relevant. Es werden effiziente Algorithmen vorgestellt, die bestimmen, auf welchen Wegen der größte Fluss von einem zu einem anderen Knoten hergestellt werden kann. Dieses Problem ist äquivalent zu dem Problem, den minimalen Schnitt in einem gewichteten Graphen zu finden. Daher lösen die *Maxflow* Algorithmen auch das *mincut* Problem.

Flussgraph

Ein **Flussgraph** (oder Netzwerk; *network*) ist ein gewichteter Digraph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$. Die Gewichte werden als Kapazitäten bezeichnet und sind *positiv*. Wir schreiben $c(v, w)$ für die Kapazität der Kante $v \to w$ und definieren $c(v, w) = 0$ für $v{\to}w \notin \boldsymbol{E}$. Des Weiteren nehmen wir an, dass es eine ausgezeichnete **Quelle** s (Knoten mit Eingangsgrad 0) und eine ausgezeichnete **Senke** t (Knoten mit Ausgangsgrad 0) gibt.



Die Definition der Flussgraphen an sich bringt eigentlich nichts Neues. Es ist nur eine andere Sichtweise auf gewichtete Digraphen. Das neue Element kommt nun:

Definition Fluss

Ein **Fluss** (*flow*) ordnet jeder Kante des Digraphen einen Fluss zu, mittels einer Funktion

$$f : \boldsymbol{V} \times \boldsymbol{V} \to \mathbb{R}$$

wobei $f(v,w) = 0$ für $v \to w \notin \boldsymbol{E}$ gesetzt wird, und die folgenden beiden Bedingungen erfüllt sind:

- **Kapazitätsbeschränkung:** (*capacity constraint*) Der Fluss jeder Kante ist *positiv* und höchstens gleich der Kapazität der Kante:

$$\forall v, w \in \boldsymbol{V} : \quad 0 \leq f(v,w) \leq c(v,w)$$

- **Flusserhaltung:** (*local equilibrium*) Für jeden Knoten außer Quelle und Senke ist der *Zufluss* (Summe vom Fluss der Kanten nach v) gleich dem *Abfluss*:

$$\forall v \in \boldsymbol{V} - \{s,t\} : \quad \sum_{w \in \boldsymbol{V}} f(w,v) = \sum_{w \in \boldsymbol{V}} f(v,w)$$

Der **Wert des Flusses** ist definiert als der Zufluss zur Senke: $|f| = \sum_{v \in \boldsymbol{V}} f(v,t)$.

Inhaltlich bedeutet die Flusserhaltung, dass nichts verloren gehen darf. Alles was bei einem Knoten als Zufluss ankommt, muss irgendwohin wieder abfließen.

Das Hauptziel dieses Kapitels ist die Entwicklung von Algorithmen, die bestimmen, auf welchen Wegen der größte Fluss von der Quelle zur Senke erreicht werden kann. Es soll also der maximale Fluss (*max-flow*) erreicht werden. Der

Wert eines Flusses wurde als Zufluss zur Senke definiert. Auf Grund der Flusserhaltung könnte er genauso gut als Abfluss von der Quelle definiert werden. Der Fluss kann sich bei seinem Weg von der Quelle zur Senke auf beliebig viele Wege aufteilen, genau so, wie man es sich bei dem Fluss von Wasser durch ein Rohrsystem vorstellt. Abbildung 12.1 zeigt einen Flussgraphen mit einem Fluss und illustriert den Wert des Flusses.

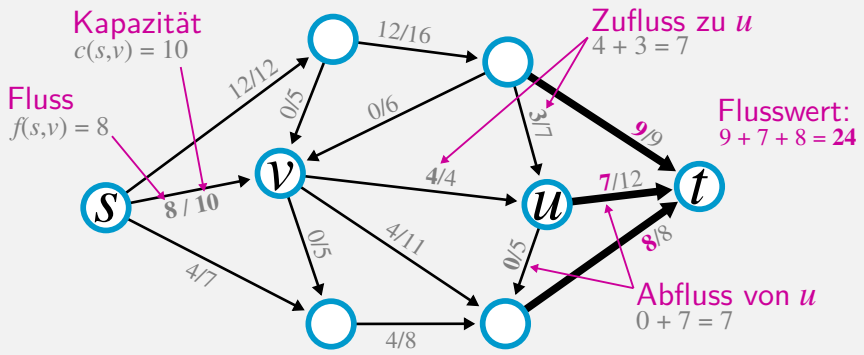


Abbildung 12.1 Darstellung eines Flussgraphen mit einem Fluss: Die Kanten werden sowohl mit dem aktuellen Fluss, als auch mit der maximalen Kapazität beschriftet. Die Kapazitäten sind bei einem spezifischen Problem fest vorgegeben und unveränderlich. Der Fluss kann variieren. In diesem Kapitel betrachten wir unterschiedliche Flüsse mit dem Ziel, denjenigen mit maximalem Flusswert zu finden. Im abgebildeten Fluss ist der Wert 24, die Summe der Flüsse der drei Kanten, die in der Senke t münden.

Maximaler Fluss (*Maxflow* Problem)

Das Problem des maximalen Flusses bezeichnet die Aufgabe, zu einem gegebenen Flussgraphen einen Fluss mit maximalem Wert zu finden.

Strategie zur Bestimmung des maximalen Flusses

Das Maxflow Problem ist komplex. Der maximale Fluss kann nicht Kante für Kante in einem Durchlauf mit geeigneter Kantenreihenfolge bestimmt werden, wie es in den meisten der bisher besprochenen Algorithmen der Fall war. Das Vorgehen ist im Grundprinzip folgendes:

- Man startet mit einem 0-Fluss, d. h. $f(v, w) = 0$ für alle v, w).
- Dann sucht man iterativ Pfade von s nach t, entlang derer der Fluss erhöht werden kann, bis keine Erhöhung mehr möglich ist.

Wenn man die Pfade für die Erhöhung des Flusses geschickt wählt, führt dieses Vorgehen zu dem maximalen Fluss. Wir führen zunächst die augmentierenden, bzw. Fluss vergrößernden Pfade ein.

Fluss vergrößernde Pfade und ihr kritischer Wert

Ein **(Fluss) vergrößernder Pfad** (*augmenting path*) ist ein *ungerichteter* Pfad von s nach t, durch den der Flusswert vergrößert werden kann.
Dabei bedeutet *ungerichtet*, dass eine gerichtete Flusskante auch in *Gegenrichtung* benutzt werden kann. Man denkt den Pfad aber in Richtung von s nach t, unabhängig von der eigentlichen Richtung der Kanten. Der Sinn hinter dieser zunächst unnatürlich wirkenden Definition ist, dass ein Fluss umgeplant werden kann – eine ursprünglich geplante Menge kann später wieder reduziert werden, um den Fluss auf einen anderen Weg zu schicken. Dieser Aspekt wird in Abbildung 12.2 verdeutlicht.
Damit der Flusswert anhand des Pfades vergrößert werden kann, müssen zwei Bedingungen erfüllt sein:

1. Bei Kanten *in Pfadrichtung* ist die Kapazität *nicht ausgeschöpft*. Hier kann der Fluss vergrößert werden.
2. Bei Kanten *gegen Pfadrichtung* ist der *Fluss größer als* 0. Hier kann der Fluss reduziert werden.

Der **kritische Wert** ist der kleinste Wert, um den der Fluss entlang des Pfades

- auf Kanten in Pfadrichtung *erhöht* und
- auf Kanten gegen Pfadrichtung *reduziert* werden kann.

Die letzte Kante zu t geht *in Pfadrichtung*, da t eine Senke ist. Also wird der Flusswert durch einen vergrößernden Pfad um den kritischen Wert des Pfades erhöht.

Die folgende Abbildung 12.3 zeigt beispielhaft, wie für einen gegebenen Graphen der maximale Fluss gefunden werden kann. Ausgehend von einem leeren Fluss wird der Fluss in vier Schritten entlang Fluss vergrößernden Pfaden erhöht.

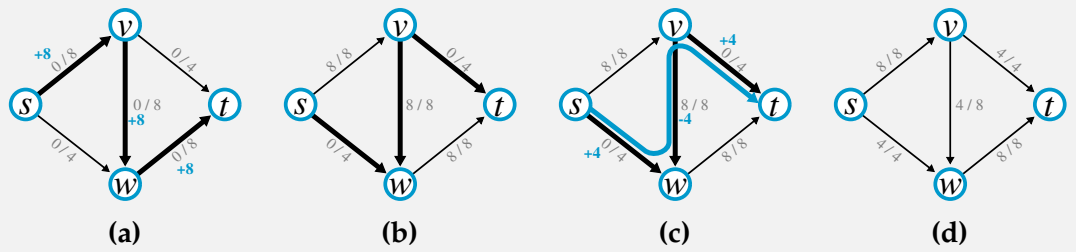
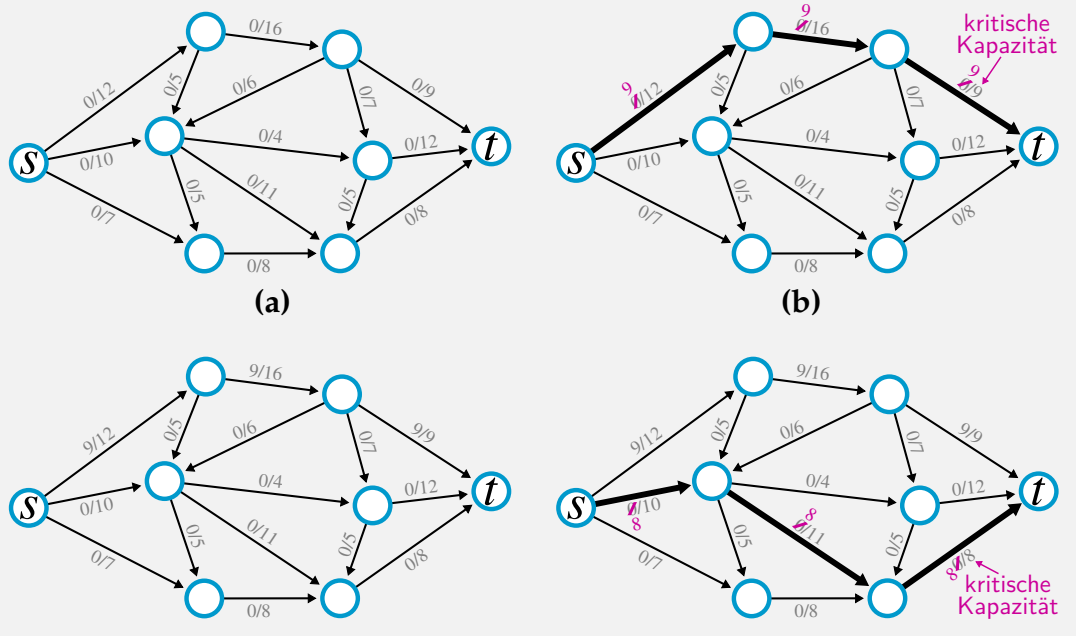


Abbildung 12.2 Kleines Beispiel zur Flussumplanung: Startend mit einem 0-Fluss wird der Fluss entlang des Pfades $s \to v \to w \to t$ erhöht **(a)**. Der kritische Wert ist 8 (freie Kapazität aller Kanten), also kann der Fluss um 8 Einheiten vergrößert werden. Das Ergebnis sieht man in **(b)**. Nun soll noch die freie Kapazität der Kante $s \to w$ ausgenutzt werden. Ohne eine Umplanung ginge es allerdings bei Knoten w nicht weiter. Die einzige von w ausgehende Kante ist $w \to t$. Deren Kapazität ist allerdings schon voll ausgeschöpft. Andererseits ist der von v kommende Zufluss nach t noch nicht ausgeschöpft. Daher wird der in **(c)** dargestellte Pfad für die Vergrößerung gewählt. Das Durchlaufen der Kante $v \to w$ in Gegenrichtung steht für die Umplanung des Flusses. Von den 8 bei v ankommenden Einheiten werden nun nur 4 durch die Kante $v \to w$ und weitere 4 durch $v \to t$ geschickt. Auf diese Weise wird die Kante $w \to t$ um 4 Einheiten entlastet. Diese können nun von $s \to w$ kommend durch $w \to t$ fließen. Das Ergebnis ist der in **(d)** gezeigte Fluss. Insgesamt entspricht der zweite Schritt einer Erhöhung des Flusses entlang des Pfades s–w–v–t um 4 Einheiten.



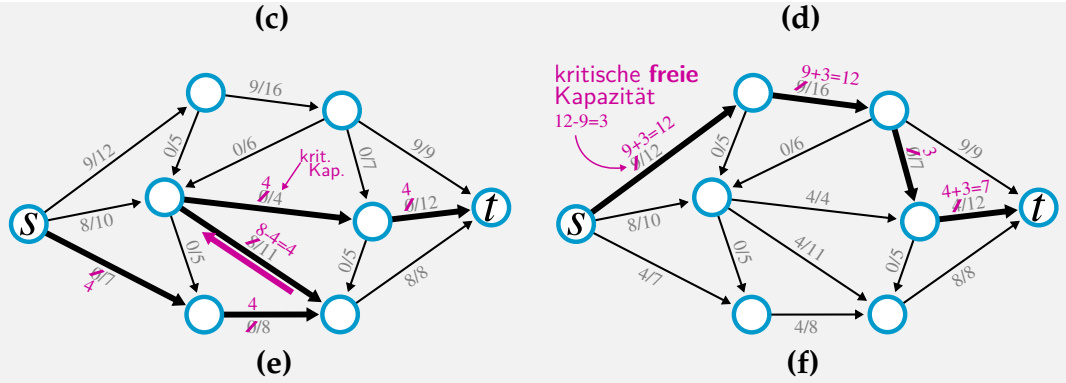


Abbildung 12.3 **Fluss vergrößernde Pfade:** **(a)** Flussgraph mit leerem Fluss. Es gibt unterschiedliche Möglichkeiten vergrößernde Pfade zu wählen. **(b)** Wir wählen zunächst den Pfad oben lang. Die kleinste Kapazität der Kanten dieses Pfades ist 9. Es ist also der kritische Wert. Der Fluss entlang dieses Pfades wird um 9 erhöht und wir erhalten **(c)**. In **(d)** wird ein weiterer Pfad gewählt. Sein kritischer Wert ist 8 und der Fluss entlang des Pfades wird entsprechend erhöht. Im unteren Bereich ist kein direkter Pfad von s nach t frei. Daher wird in **(e)** ein Pfad gewählt, der eine Kante in Gegenrichtung durchläuft (Umplanung, siehe Abbildung 12.2). Die relevante Kapazität einer Kante, die in Gegenrichtung durchlaufen wird, ist der momentane Fluss, also 8 in diesem Fall. Da der Pfad eine andere Kante mit Kapazität 4 beinhaltet, ist dies der kritische Wert, um den der Fluss erhöht wird. Bei dem letzten vergrößernden Pfad in **(f)** wird deutlich, dass für den kritischen Wert nicht die Kapazität, sondern die *freie* Kapazität relevant ist, also Kapazität einer Kante minus aktuellem Fluss durch die Kante. Nun gibt es keine vergrößernden Pfade mehr.

Wenn kein vergrößernder Pfad mehr existiert, wie am Ende von Abbildung 12.3, stellt sich die Frage, ob damit zwingend der maximale Fluss gefunden ist. Ist es auch möglich, dass man durch eine ungünstige Wahl vergrößernder Pfade in eine Sackgasse gerät und auf anderem Wege ein größerer Fluss hergestellt werden kann? Weiter unten wird bewiesen, dass letzteres nicht der Fall ist, und man immer beim maximalen Fluss angelangt ist, wenn es keine vergrößernden Pfade mehr gibt.

Diese Methode, den maximalen Fluss über die iterierte Auswahl vergrößernder Pfade herzustellen, ist nach Ford-Fulkerson benannt. Bevor wir sie detaillierter besprechen, werden die sogenannten **Restgraphen** (*residual graphs*) eingeführt. Dieses Konzept integriert die beiden unterschiedlichen Anforderungen bezüglich Pfeilen *in* und *gegen* Pfadrichtung und beschreibt somit die Möglichkeiten zur Flussvergrößerung. Restgraphen sind nur ein Konstrukt, das zur einfacheren algorithmischen Behandlung von vergrößernden Pfaden eingeführt wird. Restgraphen sind auch für die Implementation von Flussalgorithmen wichtig.

Restgraphen

Zu einem Flussgraph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ und Fluss f definieren wir einen gewichteten Digraphen $\boldsymbol{G}_f = (\boldsymbol{V}, \boldsymbol{E}_f)$, genannt Restgraph zu $\boldsymbol{G}$ und f, wobei die Gewichte die **Restkapazität** (*residual capacity*) sind:

$$rc(v,w) = \begin{cases} c(v,w) - f(v,w) & \text{falls } v{\to}w \in \boldsymbol{E} \\ f(v,w) & \text{falls } w{\to}v \in \boldsymbol{E} \end{cases}$$

Es werden nur Kanten $v{\to}w$ berücksichtigt, deren Restkapazität $rc(v,w) > 0$ ist, also

$$\boldsymbol{E}_f = \{v{\to}w \mid v,w \in \boldsymbol{V} \ \ \& \ \ rc(v,w) > 0\}.$$

Der Zusammenhang zwischen Restgraphen und Flussvergrößerungen

Für einen Pfeil $v{\to}w \in \boldsymbol{G}$ des ursprünglichen Flussgraphen, gibt die Restkapazität

- des Pfeils $v{\to}w \in \boldsymbol{G}_f$ an, um wieviel der Fluss f vergrößert und
- des Pfeils $w{\to}v \in \boldsymbol{G}_f$ an, um wieviel der Fluss f verringert werden kann.

Der kritische Wert eines Pfads im Restgraphen ist die kleinste Restkapazität seiner Kanten. Da der Restgraph nur Kanten $v \to w$ mit $rc(v,w)>0$ enthält, ist der kritischer Wert immer > 0. Wenn es in dem Restgraphen $\boldsymbol{G}_f$ einen Pfad von s nach t gibt, kann der Fluss f entlang des Pfades um den kritischen Wert vergrößert werden:

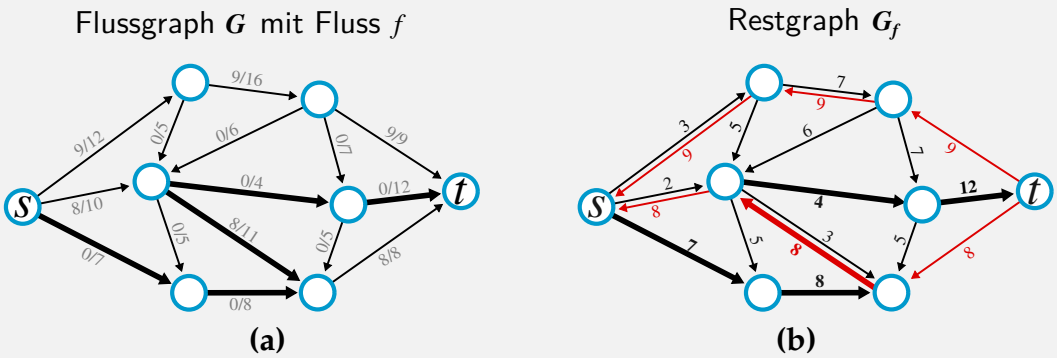


Abbildung 12.4 Restgraph zu dem Fluss eines Flussgraphen: (a) Bei dem Flussgraph G mit Fluss f ist ein Fluss vergrößernder Pfad eingezeichnet. **(b)** zeigt den Restgraphen. Die Darstellung der Kanten in unterschiedlichen Farben dient nur der Verdeutlichung, wie die Kanten entstanden sind (Kanten von G schwarz, und invertierte Kanten von G in rot). Der vergrößernde Pfad aus (a) ist ein (normaler) gerichteter Pfad im Restgraphen. So wird das Durchlaufen einer Kante in Gegenrichtung, die in (a) zur Flussumplanung notwendig ist, im Restflussgraphen in das Durchlaufen in Kantenrichtung umgewandelt.

12.1.1 Minimale Schnitte und Maximaler Fluss

Es gibt einen engen Zusammenhang zwischen minimalen Schnitten in einem Graphen und maximalen Schnitten in einem Flussgraphen. Dieser Zusammenhang kann zweifach genutzt werden. Zum einen kann das Zusammenspiel zum Beweis der Korrektheit des allgemeinen Verfahrens zur Bestimmung des maximalen Flusses verwendet werden. Außerdem bietet der Zusammenhang die Möglichkeit, die im Folgenden vorgestellten MaxFlow Algorithmen auch zu verwenden, um minimale Schnitte in gewichteten Graphen zu finden.

Schnitte und ihre Kapazitäten

Ein s von t trennender **Schnitt** (*cut*) teilt die Knoten eines Flussgraphen in zwei zusammenhängende, nicht-leere Teilmengen S und $T = V - S$, wobei die Quelle in S und die Senke in T ist: $s \in S,\ t \in T$.
Die **Kapazität eines Schnittes** in einem Digraphen ist die Summe der Kapazitäten der kreuzenden Kanten, die S verlassen. Kanten, die nach S hereinführen werden *nicht* gezählt.

Schnitte werden auch für ungerichtete und für ungewichtete Graphen betrachtet. Diese Definitionen werden im diesem Skript nicht verwendet, seien aber der Vollständigkeit halber hier genannt. Für ungerichtete Graphen ist die Kapazität eines Schnittes die Summe der Gewichte *aller* kreuzenden Kanten. Bei ungewichteten Graphen ist die Kapazität eines Schnittes die *Anzahl* der kreuzenden Kanten, also entsprechend der Definition für gewichtete Graphen, wenn man das Gewicht 1 für alle Kanten annimmt.

Abbildung 12.5 zeigt Beispiele für vier unterschiedliche Schnitte in einem Graphen und ihre Kapazitäten:

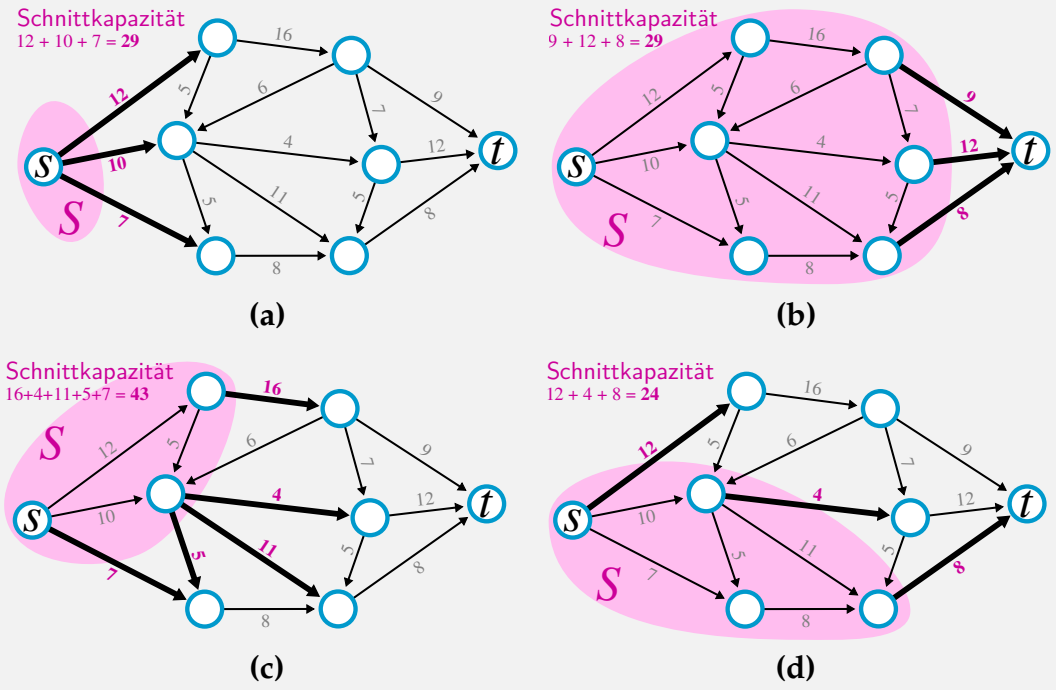


Abbildung 12.5 **Schnitte durch einen Graphen**

Minimaler Schnitt (*mincut problem*)

Das Mincut Problem besteht darin, einen Schnitt mit minimaler Kapazität (minimaler Schnitt; *mincut*) zu finden.

(Minimale) Schnitte werden nicht nur für Flussgraphen (also gewichtete Digraphen) betrachtet, sondern auch für ungerichtete Graphen, sowohl mit als auch ohne Kantengewichte.

Fluss über einen Schnitt

Wir definieren als **Fluss über einen Schnitt** $f(S)$ zu gegebenem Fluss f und Schnitt S die Summe der Flusswerte aller *kreuzenden Kanten*. Dabei werden Kanten *aus* S positiv und Kanten *nach* S negativ gerechnet.

Zur einfacheren Schreibweise in der Formel schreiben wir auch $f(e)$ für den Fluss der Kante $e = v{\to}w$ und definieren

- die Menge der kreuzenden Kanten, die *nach* S *herein* zeigen

$$E_S^{\leftarrow} = \{v{\to}w \in E \mid v \notin S \;\&\; w \in S\}$$

- und die Menge der kreuzenden Kanten, die *aus* S *heraus* zeigen

$$E_S^{\rightarrow} = \{v{\to}w \in E \mid v \in S \;\&\; w \notin S\}$$

Mit diesen Definitionen erhalten wir die folgende Formel für den Fluss über den Schnitt S:

$$f(S) = \sum_{\substack{v\to w\in E\\ v\in S,\, w\notin S}} f(v,w) - \sum_{\substack{w\to v\in E\\ v\in S,\, w\notin S}} f(w,v) = \sum_{e\in E_S^{\rightarrow}} f(e) - \sum_{e\in E_S^{\leftarrow}} f(e)$$

Schnitttheorem: Zusammenhang von Flüssen und Schnitten

Der Fluss ist über alle Schnitte derselbe. Er entspricht immer dem Flusswert.

Bevor wir die Aussage beweisen, betrachten wir ein paar Beispiele:

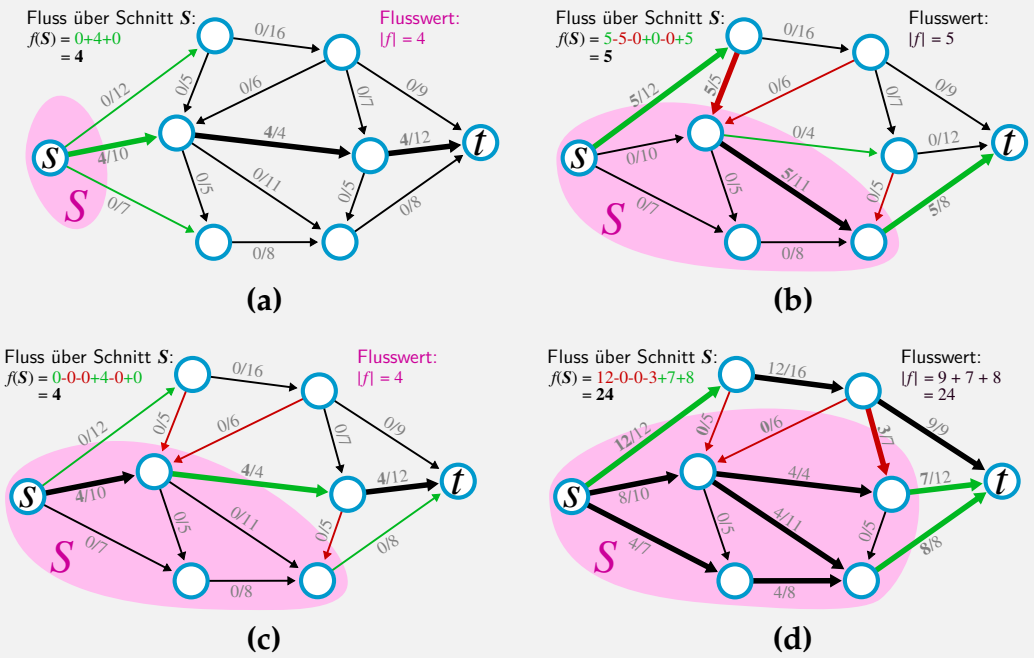


Abbildung 12.6 Beispiele für die Aussage des Schnitttheorems: Der Fluss über alle Schnitte eines Flussgraphen entspricht dem Flusswert. Die Beispiele **(a)** bis **(c)** zeigen einfache Fälle, bei dem der Fluss nur entlang eines Pfades verläuft. Der Fluss in **(d)** ist der maximale Fluss.

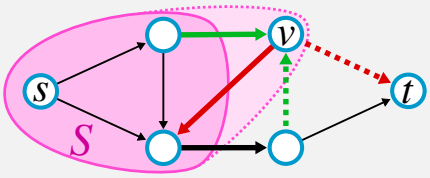
Beweis des Schnitttheorems

Wir beweisen durch Induktion nach $|T|$, dass für alle Schnitte (S, T) gilt: $f(S, T) = |f|$.

Induktionsanfang. Für $|T| = 1$ ist $T = \{t\}$ und $S = V - \{t\}$. In diesem Fall entspricht der Flusswert per Definition dem Wert des Flusses über den Schnitt (S, T). Beides ist der Zufluss zu t, da t als Senke keine ausgehenden Kanten besitzt.

Induktionsschritt. Per IV wissen wir, dass $f(S + \{v\}, T) = |f|$ gilt und müssen zeigen, dass dies auch für $f(S, T + \{v\})$ gilt. Bei dem Übergang von Schnitt $(S + \{v\}, T)$ zu $(S, T + \{v\})$ passiert folgendes:

- Es fällt weg (in der Skizze rot/grün gestrichelt): Fluss der Kanten von v nach T und der negativ gewichtete Fluss der Kanten von T nach v.
- Es kommt hinzu (in der Skizze rot/grün durchgezogen): Fluss der Kanten von S nach v und der negativ gewichtete Fluss der Kanten von v nach S.



In Summe kommt also der Zufluss nach v dazu (grün), und es wird der Abfluss von v abgezogen (rot). Nach der Flusserhaltungsbedingung ergibt dies 0. □

Alternativer rechnerischer Beweis der Schnitttheorems

Wir nehmen die Definition des Fluss eines Schnittes und addieren den folgenden Term, der 0 ergibt, da beide Summen über alle Kanten innerhalb T laufen:

$$\sum_{\substack{v \to w \in E \\ v \in T,\, w \in T}} f(v, w) - \sum_{\substack{w \to v \in E \\ v \in T,\, w \in T}} f(w, v)$$

Auf diese Weise erhalten wir nach Umsortieren der Summanden:

$$f(S,T) = \sum_{\substack{v \to w \in E \\ v \in S,\, w \in T}} f(v,w) - \sum_{\substack{w \to v \in E \\ v \in S,\, w \in T}} f(w,v)$$ Definition Fluss über Schnitt

$$= \sum_{\substack{v \to w \in E \\ v \in V,\, w \in T}} f(v,w) - \sum_{\substack{w \to v \in E \\ v \in V,\, w \in T}} f(w,v)$$ Addition von obigem Term, $V = S \cup T$ und Umsortieren

$$= \sum_{w \in T} (\text{Zufluss zu } w - \text{Abfluss von } w) = |f|$$ Wegen Flusserhaltung bleibt nur der Term für t.

□

Wir kommen zu den beiden Hauptaussagen dieses Kapitels. Sie stellen den Zusammenhang zwischen minimalem Schnitt, maximalem Fluss sowie vergrößernden Pfaden her und werden gemeinsam über eine Äquivalenz dreier Aussagen bewiesen.

Vergrößernde Pfade und maximaler Fluss

Ein Fluss f ist genau dann maximal, wenn es keine vergrößernden Pfade gibt.

Maximaler Fluss und minimaler Schnitt

Der Wert des maximalen Flusses entspricht der Kapazität des minimalen Schnittes.

Beide Aussagen sind Teil der folgenden Äquivalenz:

Minimaler Schnitt, maximaler Fluss und vergrößernde Pfade

Für einen Flussgraphen G mit Fluss f sind die folgenden drei Aussagen äquivalent:

1. Es gibt einen Schnitt, dessen Kapazität mit dem Wert von f übereinstimmt.
2. f ist ein maximaler Fluss.
3. Es gibt keinen vergrößernden Pfad für f.

Beweis von 1 nach 2.
1 Es gibt einen Schnitt mit $c(S,T) = |f| \Rightarrow$ 2 f ist maximaler Fluss

Für jeden Fluss f' gilt: $|f'| = f'(S,T) \leq c(S,T) = |f|$. Also ist der Flusswert $|f|$ maximal. □

Beweis von 2 nach 3.
2 f ist maximaler Fluss $\Rightarrow$ 3 Es gibt keinen vergrößernden Pfad für f

Wir nehmen an, dass es einen vergrößernden Pfad für f gibt. Dann kann der Fluss entlang dieses Pfads um den kritischen Wert vergrößert werden, im Widerspruch zur Annahme, dass der Fluss f maximal ist. □

Beweis von 3 nach 1.
3 Kein vergrößernder Pfad für $f \Rightarrow$ 1 Es gibt Schnitt mit $c(S,T) = |f|$

Wenn es keinen vergrößernden Pfad für f gibt, definiert die Menge S aller Knoten v, die von s im Restgraphen erreicht werden können, einen Schnitt. Nach dieser Definition gilt $rc(v,w) = 0$ für alle kreuzenden Kanten $v \to w$, also gilt $c(S,T) = f(S,T)$. Weiter folgt mit dem Schnitttheorem: $c(S,T) = f(S,T) = |f|$. □

Durch diese drei Implikationen ist die Äquivalenz der drei Aussagen bewiesen. ■
Der Kernteil des Beweises (3 $\Rightarrow$ 1) ist in Abbildung 12.7 illustriert.

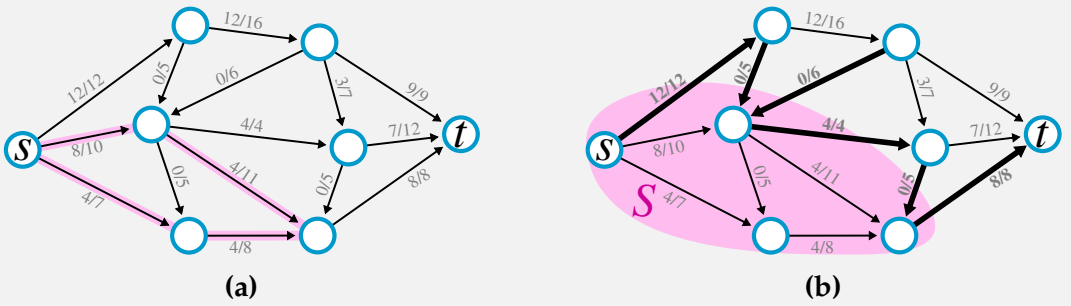


Abbildung 12.7 Illustration zum Maxflow - Mincut Theorem: Wenn es keine vergrößernden Pfade gibt, kann es keinen Pfad von s nach t im Restgraphen geben. Die von s im Restgraphen erreichbaren Pfade definieren also einen s von t trennenden Schnitt S, für den folgendes gilt: Alle S verlassenden kreuzenden Kanten müssen voll gefüllt sein und alle S betretenden kreuzenden Kanten müssen leer sein. Daher stimmt die Kapazität von S mit dem Fluss über S überein.

12.2 Allgemeine Methode zum Identifizieren des maximalen Flusses

Von **Ford-Fulkerson** wurde die Technik der vergrößernden Pfade entwickelt, um eine allgemeine Methode zum Identifizieren des maximalen Flusses in Flussgraphen anzugeben:

Listing 12.1 Pseudocode zur Bestimmung des maximalen Flusses eines Flussgraphen gemäß der Ford-Fulkerson Methode

```
1  for each e in E
2    f(e) ←0
3  end
4  G_f ←Restgraph von G
5  while es gibt einen Pfad p in G_f do
6    cv ←min { rc(e) | e liegt auf Pfad p in G_f }
7    vergrößere f entlang p um cv
8    aktualisiere G_f
9  end
10 return f // maximaler Fluss
```

Mit "*vergrößere f entlang p um cv*" ist das Verfahren der vergrößernden Pfade gemeint:

- Auf Kanten in Richtung des Pfades p wird der Fluss um cv erhöht und
- auf Kanten gegen Richtung des Pfades p wird der Fluss um cv reduziert.

Die Ford-Fulkerson Methode ist ein allgemeiner Ansatz. Sie legt nicht fest, auf welche Weise der (vergrößernde) Pfad p in Zeile 5 ausgewählt wird. Daher bezeichnen wir das Verfahren hier als *Methode* und nicht als *Algorithmus*.

Vorbemerkungen zu Korrektheit und Laufzeit der Ford-Fulkerson Methode

Korrektheit: Wenn die allgemeine Ford-Fulkerson Methode terminiert, wissen wir nach dem Satz über vergrößernde Pfade, dass das Ergebnis ein maximaler Fluss ist. Es ist nur die Frage, ob die Ford-Fulkerson Methode garantiert terminiert.
Bevor wir die Laufzeit diskutieren, die die Terminierung impliziert, besprechen wir Beispiele, die der Methode Schwierigkeiten bereiten. Dabei ist zu beachten, dass bisher keine Strategie zur Auswahl der vergrößernden Pfade spezifiziert wurde. Die Beispiele beruhen auf einer 'unglücklichen' Reihenfolge.

Kleiner Flussgraph mit potenziell langer Laufzeit

Der in Abbildung 12.8 dargestellte Flussgraph hat den maximalen Fluss von $|f^*| = 2.000$, der durch die Kombination der beiden Pfade $s \to v \to t$ und $s \to w \to t$ mit jeweils 1.000 Einheiten erreicht wird. Eine unglückliche Wahl der vergrößernden Pfade ist ein Wechsel von $s \to v \to w \to t$ und $s \to w \to v \to t$. Diese Pfade haben beide den kritischen Wert 1, so dass insgesamt 2.000 Iterationen nötig sind, um den maximalen Fluss $|f^*|$ zu erzeugen.

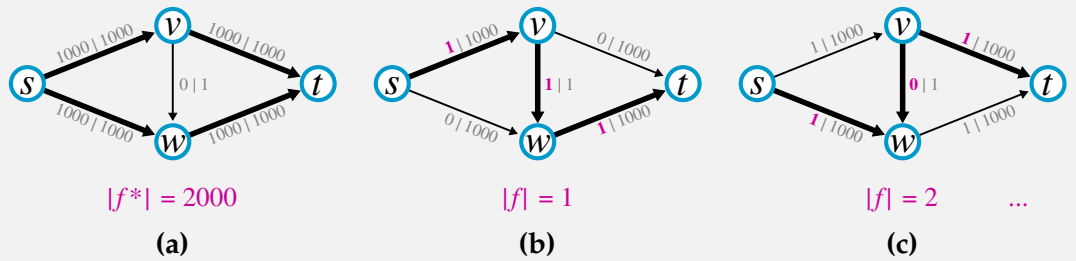


Abbildung 12.8 Flussgraph mit potenziell langer Laufzeit bei Ford-Fulkerson: Der dargestelle Flussgraph hat einen maximalen Fluss von $|f^*| = 2000$. **(a)** Dieser kann durch zwei vergrößerende Pfade erreicht werden. Bei einer ungeschickten Wahl der Pfade werden sehr viele Schritte benötigt: **(b)** Entlang des Pfades $s \to v \to t$ kann der Fluss um 1 erhöht werden (kritischer Wert 1 durch $v \to w$). **(c)** Ebenso ist entlang $s \to w \to t$ nur eine Erhöhung um 1 möglich. Um den maximalen Fluss $|f^*| = 2000$ zu erreichen, sind 2000 Flussvergrößerungen notwendig.

Dieses Beispiel zeigt auch, dass die Laufzeit der Ford-Fulkerson Methode, sofern kein geeignetes Verfahren zur Pfadauswahl angegeben wird, nicht nur von der *Struktur des Graphen* (V, E), sondern auch von *seinen Kapazitäten* abhängen kann.

Kleiner Graph ohne Terminierung

Es kommt noch schlimmer. Es gibt *keine Garantie*, dass die Ford-Fulkerson Methode *überhaupt terminiert*. Diese ungünstigen Fälle können allerdings nur auftreten, wenn es unter den Kapazitäten des Flussgraphen *irrationale* Zahlen gibt und die vergrößernden Pfade ungünstig gewählt werden. Abbildung 12.9 zeigt ein solches Beispiel:

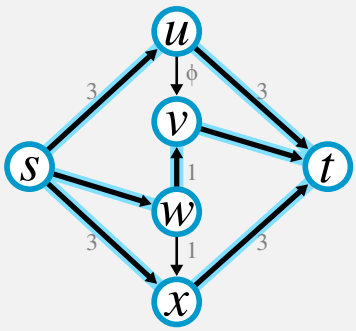


Abbildung 12.9 Graph ohne garantierte Terminierung bei Ford-Fulkerson. Die Kapazität der Kante $u \to v$ ist irrational: $\phi = (\sqrt{5}-1)/2$.

Mangelnde Korrektheit der Ford-Fulkerson Methode

Bei Ford-Fulkerson Methode gibt es für Flussgraphen mit irrationalen Kapazitäten keine Garantie für die Terminierung.

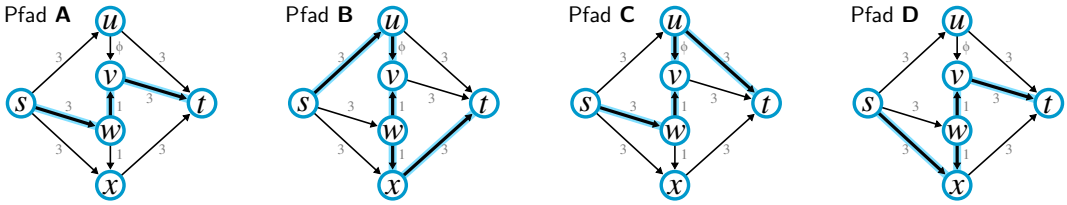
Beweis. Das Beispiel in Abbildung 12.9 beruht auf dem Verhältnis des goldenen Schnittes $\phi = (\sqrt{5}-1)/2$ und den Eigenschaften

$$\phi^2 = \left(\frac{\sqrt{5}-1}{2}\right)^2 = \frac{5-2\sqrt{5}+1}{4} = \frac{3-\sqrt{5}}{2} = 1-\phi$$

und

$$\phi - \phi^2 = \phi(1-\phi) = \phi \cdot \phi^2 = \phi^3.$$

Der abgebildete Graph hat einen maximalen Fluss von mindestens 7: Die Pfade $s-u-t$ und $s-x-t$ bringen jeweils 3 und $s-w-v-t$ bringt 1. Eine ungeschickte Wahl, die zu einer nicht-terminierenden Sequenz von Flussvergrößerungen führt, nutzt die folgenden vier Pfade:



Folgende Wahl vergrößernder Pfade führt zu einer *endlosen* Sequenz:

Pfad	Fluss	Restkapazitäten $u\to v$	$v\to w$	$w\to x$				
A	1	ϕ	0	1	$\to$	ϕ^{k+1}	0	ϕ^k (für $k=0$)
B	ϕ	0	ϕ	ϕ^2	da $1-\phi=\phi^2$			
C	ϕ	ϕ	0	ϕ^2				
B	ϕ^2	ϕ^3	ϕ^2	0	da $\phi-\phi^2=\phi^3$			
D	ϕ^2	ϕ^3	0	ϕ^2	$\to$	ϕ^{k+3}	0	ϕ^{k+2} (für $k=0$)
A + $K\cdot$ BCBD	$1+\sum_{k=1}^{2K} 2\phi^k$	ϕ^{2K+1}	0	ϕ^{2K}				

Die Pfad Sequenz *A und dann immer wiederholend B, C, B, D* konvergiert nicht zum maximalen Fluss:

$$1+2\sum_{k=1}^{\infty}\phi^k = 1+\frac{2}{1-\phi} = 4+\sqrt{5} < 7$$

Zum vollständigen Beweis fehlen noch folgende (einfache) Punkte:

- Zeige die Eigenschaften der Sequenz (siehe vorige Seite) durch Induktion nach k.
- Zeige insbesondere $1-\phi^k = \phi^{k+1}$ und $\phi^k - \phi^{k+1} = \phi^{k+2}$.
- Prüfe bei der Induktion, dass die Kanten mit Kapazität 3 nicht über ihre Kapazitätsgrenzen gefüllt werden.

Laufzeit der Allgemeinen Ford-Fulkerson Methode

Um die Laufzeit der allgemeinen Ford-Fulkerson Methode zu bestimmen, beschränken wir uns daher auf *rationale* Kapazitäten. Wir setzen sogar voraus, dass alle Kapazitäten des Flussgraphen in $\mathbb{N}^{>0}$ sind. Beliebige rationale Zahlen können mit dem KGV aller Nenner multipliziert werden, um einen äquivalenten Flussgraphen mit Kapazitäten in $\mathbb{N}^{>0}$ zu definieren. Insofern stellt die Beschränkung auf natürliche Zahlen keine weitere Beschränkung dar.
Das Beispiel in Abbildung 12.8 zeigt, dass die Laufzeit nicht nur von der Größe des Graphen (V, E), sondern *auch von den Kapazitäten* abhängen kann. In der Tat erhalten wir:

Laufzeit der Ford-Fulkerson Methode

Die Ford-Fulkerson Methode benötigt für einen Flussgraphen, dessen Kapazitäten natürliche Zahlen sind, eine Laufzeit in $\boldsymbol{O}(E|f^*|)$, wobei f^* der maxi-

male Fluss ist. (Wenn konkrete Verfahren zur Auswahl der vergrößernden Pfade benutzt werden, wie bei dem Edmonds-Karp Algorithmus kann eine bessere Laufzeit garantiert werden.)

Beweis. Die Restkapazitäten sind immer ganze Zahlen, also auch der kritische Wert jedes vergrößernden Pfades.

- Der Fluss wird bei jeder Iteration der *while*-Schleife um den kritischen Wert erhöht, also um mindestens 1. Daher wird der maximale Fluss nach höchstens $|f^*|$ Durchläufen der *while*-Schleife erreicht.
- In jeder Iteration werden $\boldsymbol{O}(V + E_{\boldsymbol{R}}) = \boldsymbol{O}(E)$ Schritte benötigt, um einen Pfad im Restgraphen zu finden, z. B. durch Tiefensuche. Die Zeit, um die Restkapazität in Zeile 6 zu bestimmen und den Fluss in Zeile 7 zu vergrößern ist linear in $\boldsymbol{E}$.
- Insgesamt ergibt sich eine Laufzeit in $\boldsymbol{O}(E|f^*|)$. □

12.2.1 Überlegungen zur Konkretisierung der Ford-Fulkerson Methode

Wir haben gesehen, dass die allgemeine Ford-Fulkerson Methode, die keine spezifische Auswahl der vergrößernden Pfade beinhaltet, problematisch ist. Daher wurden Algorithmen vorgeschlagen, die eine geschicktere Reihenfolge der Pfadvergrößerungen verwenden. Dadurch kann man eine garantierte Terminierung und eine bessere (worst-case) Laufzeit erzielen.

Eine spezifische Wahl der vergrößernden Pfade

Die bewiesene worst-case Laufzeit der Ford-Fulkerson Methode $\boldsymbol{O}(E|f^*|)$ hängt von dem maximalen Fluss ab. Diese Angabe ist nur begrenzt nützlich, da der maximale Fluss noch nicht bekannt ist, wenn man sich für eine Methode zur Bestimmung des maximalen Flusses entscheiden möchte. Alternativ kann als Wachstumsordnung der Laufzeit $\boldsymbol{O}(EVC)$ angegeben werden, wobei C eine obere Schranke für die Kapazitäten ist, da $|f^*| \leq VC$.
Um eine Laufzeitschranke zu erzielen, die nur von der Größe des Graphen abhängt, muss man eine spezielle Strategie zur Auswahl des vergrößernden Pfades (= nicht-leerer Pfad im Restgraphen) verwenden. Folgende Strategien scheinen plausibel:

- Wähle einen vergrößernden Pfad mit wenigen Kanten.
- Wähle einen vergrößernden Pfad mit großem Fluss (großem kritischen Wert).

In dem folgenden Abschnitt wird der Algorithmus von Edmonds-Karp besprochen, der die ersten Strategie implementiert. Im übernächsten Abschnitt folgt der *Capacity-Scaling* Algorithmus, der auf der zweiten Strategie beruht.

12.3 Der Edmonds-Karp Algorithmus (Pfad mit wenigen Kanten)

Der **Edmonds-Karp Algorithmus** wählt als vergrößernden Pfad in der Ford-Fulkerson Methode einen Pfad, der die wenigsten Kanten hat:

- Man fängt mit einem leeren Fluss f an. Der Fluss wird iterativ vergrößert.
- Unter allen Pfaden von s nach t im Restgraphen $\boldsymbol{G}_f$ wird ein Pfad p mit der geringsten Anzahl von Kanten ausgesucht.
- Dies kann z. B. durch Breitensuche im Restgraphen geschehen.
- Dann wird der kritische Wert des Pfades p bestimmt. Dies ist die kleinste Restkapazität (Gewichte im Restgraphen $\boldsymbol{G}_f$) der Kanten des Pfades.
- Alle Kantengewichte des Restgraphen entlang p werden um diesen kritischen Wert verringert. Dann geht es weiter mit dem nächsten Iterationsschritt.

Der Fluss f braucht dabei nicht explizit gespeichert zu werden. Alle benötigte Information ist in dem Restgraphen.

Listing 12.2 Pseudocode zur Bestimmung des maximalen Flusses eines Flussgraphen $\boldsymbol{G}$ von Quelle s zur Senke t mit dem Edmonds-Karp Algorithmus

```
1  G_f ←Restgraph von G für leeren Fluss f ≡ 0
2  while es gibt einen Pfad p von s nach t in G_f do
3    wähle Pfad p in G_f mit den wenigsten Kanten
4    cv ←min { rc(e) | e liegt auf Pfad p }
5    // aktualisiere G_f entlang p:
6    for all Knoten v, w mit v→w auf Pfad p in G_f
7      rc(v,w) ←rc(v,w) – cv
8      rc(w,v) ←rc(w,v) + cv
9    end
10 end
```

Für die Implementierung bietet es sich an, alle Kanten in dem Restgraphen $\boldsymbol{G}_f$ zu speichern und belassen, auch wenn die Restkapazität einer Kante 0 ist. Dann dürfen bei der Wahl des Pfades in Zeilen 2 und 3 nur Kanten $v \to w$ mit $rc(v, w) > 0$ berücksichtigt werden.

Laufzeit des Edmonds-Karp Algorithmus

Der Edmonds Karp Algorithmus bestimmt den maximalen Fluss eines Flussgraphen in einer Laufzeit von $\boldsymbol{O}(E^2V)$.

Beweis. Die Korrektheit gilt als Spezialfall der Ford-Fulkerson Methode. Die Laufzeit folgt aus den beiden Hilfssätzen, die auf den folgenden Seiten bewiesen werden.

1 Die Längen der kürzesten vergrößernden Pfade ist monoton steigend.

2 Spätestens nach E Iterationen steigt die Länge *streng* an.

Da die Länge des kürzesten Pfades höchstens $V-1$ beträgt, kann es nach den Aussagen **1** und **2** maximal $E(V-1)$-viele Iterationen (= Flussvergrößerungen) geben. Ein kürzester vergrößernder Pfad wird jeweils im Restgraphen mit Breitensuche in $\boldsymbol{O}(E)$ gefunden (beachte $E \geq V-1$). Ebenso erfolgt die Aktualisierung des Restgraphen in $\boldsymbol{O}(E)$. Die Gesamtlaufzeit ist somit in $\boldsymbol{O}(E^2V)$. □

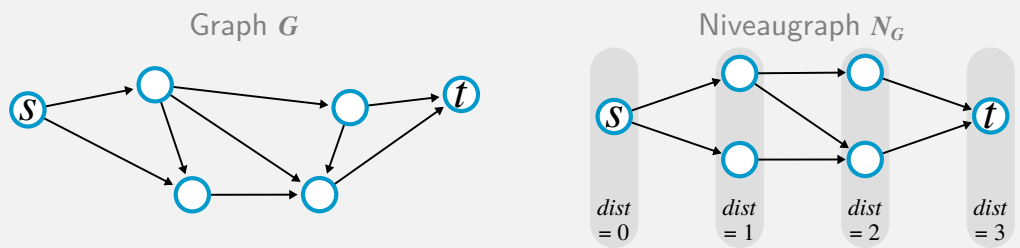
Um den ersten Hilfssatz anschaulich zu beweisen, führen wir **Niveaugraphen** ein.

Niveaugraph

Zu einem gegebenen Flussgraphen $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ definieren wir den Niveaugraphen $\boldsymbol{N_G} = (\boldsymbol{V}, \boldsymbol{E_N})$ als Untergraph von $\boldsymbol{G}$ mit folgenden Kanten:

$$\boldsymbol{E_N} = \{v \to w \mid dist(w) = dist(v) + 1\}$$

wobei *dist(v)* die Anzahl der Kanten des kürzesten Weges von der Quelle s nach v in $\boldsymbol{G}$ ist. Der Niveaugraph enthält genau dann einen Pfad s⇝v, wenn s⇝v ein kürzester Pfad in $\boldsymbol{G}$ ist.



Hilfssatz 1

Die Längen der kürzesten vergrößernden Pfade, die im Edmonds-Karp Algorithmus ausgewählt werden, sind schwach monoton steigend.

Beweis. Sei f der Fluss *vor* und f' der Fluss *nach* einer Flussvergrößerung. Wir vergleichen die Niveaugraphen zu den Restgraphen $\boldsymbol{G}_f$ und $\boldsymbol{G}_{f'}$. Der vergrößernde Pfad p muss ein Pfad in $\boldsymbol{N}_{\boldsymbol{G}_f}$ sein. Durch die Flussvergrößerung können nur Kanten verändert werden, die in Pfad p benachbarte Knoten verbinden:

- Kanten in Pfadrichtung können höchstens wegfallen. Dies passiert, wenn eine Kante $v \to w$ voll gefüllt wird: $f'(v,w) = c(v,w)$.
- Kanten gegen Pfadrichtung $w \to v$ können neu entstehen. Dies passiert, wenn eine Kante $v \to w$ angefüllt wird, die vorher leer war: $f'(v,w) > f(v,w) = 0$.

Weder wegfallende Kanten, noch neue Kanten, die im Niveaugraphen rückwärts verlaufen würden, können zu kürzeren Wegen im Restgraphen $\boldsymbol{G}_{f'}$ führen. □

Hilfssatz 2

Spätestens nach E Iterationen steigt die Länge *streng* an (also um einen Wert > 0)

Beweis. Der kürzeste vergrößerende Pfad p im Restpgrahen $\boldsymbol{G}_f$ ist im Niveaugraphen $\boldsymbol{N}_{\boldsymbol{G}_f}$ enthalten. Bei jeder Flussvergrößerung wird mindestens eine Kante des Pfades aus dem Restgraphen gelöscht. Dies ist diejenige Kante, bei der der kritische Wert erreicht wird (Kapazität erschöpft bei Kanten in Pfadrichtung im Flussgraphen, bzw. Fluss auf 0 reduziert bei Kanten gegen Pfadrichtung im Flussgraphen).
Nach spätestens E-vielen Flussvergrößerungen sind also alle Kanten aus dem Niveaugraphen gelöscht. Dies bedeutet, dass es keine vergrößernden Pfade dieser Länge mehr gibt. Daher muss die Länge um mindestens 1 ansteigen. □

Verbesserungen der Laufzeit von Edmonds-Karp

Es stellt sich die Frage, ob die Wachstumsordnung der Laufzeit $\boldsymbol{O}(E^2V)$ verbessert werden kann. Diese Abschätzung ergibt sich gemäß dem Beweis der Laufzeit aus

- der Anzahl der Flussvergrößerungen: $\boldsymbol{O}(VE)$ und
- der Laufzeit der Flussvergrößerung: $\boldsymbol{O}(E)$

Es gibt Beispiele für Flussgraphen, bei denen die *Anzahl der notwendigen Flussvergrößerungen* tatsächlich in $\boldsymbol{\Theta}(VE)$ liegt, wenn immer ein kürzester vergrößernder Pfad gewählt wird. An dieser Schranke ist also in Edmonds-Karp nichts zu verbessern.
Es kann aber die *benötigte Zeit für Flussvergrößerungen* reduziert werden. Der *blocking-flow* Algorithmus wurde in [Dinic, 1970] vorgeschlagen, also *vor* der Veröffentlichung von Edmonds-Karp. Dabei werden Pfade in dem Niveaugraphen schrittweise aktualisiert, um jeweils den nächsten vergrößernden Pfad effizienter zu finden. Auf diese Weise lässt sich eine Laufzeit in $\boldsymbol{O}(EV^2)$ erreichen.
Mit dynamischen Bäumen [Sleator and Tarjan, 1983] kann sogar eine Laufzeit in $\boldsymbol{O}(EV \log V)$ erzielt werden.

12.4 Der Kapazitätskontrolle Algorithmus (Pfad mit großem Fluss)

Bei der Überlegung, wie die Auswahl der vergrößernden Pfade in der allgemeinen Ford-Fulkerson Methode konkretisiert werden könnte, wurden zwei Strategien vorgeschlagen. Der Strategie der kürzesten Pfade wurde im Edmonds-Karp Algorithmus nachgegangen. Noch zielführender scheint die Strategie:

- Wähle einen Pfad, der den *Fluss maximal vergrößert*.

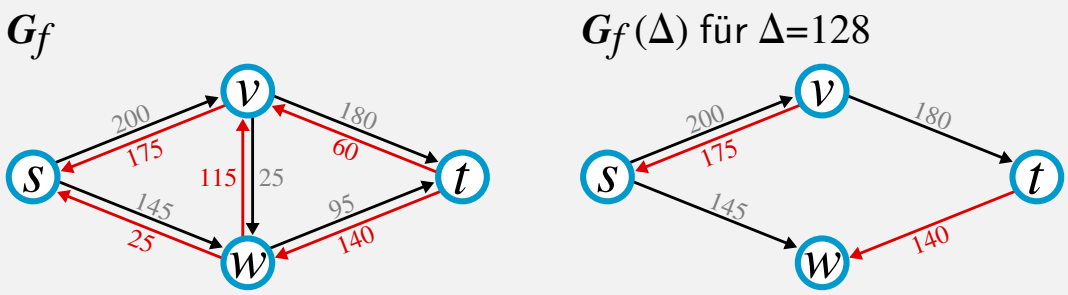
Eine entsprechende Pfadauswahl wurde auch von Edmonds und Karp vorgeschlagen, ist aber weniger effizient zu implementieren.
Geben wir uns also mit weniger zufrieden: Der vergrößernde Pfad erhöht den Fluss *nicht maximal*, aber *relativ stark*. Wir benutzen einen Parameter Δ zur Kapazitätskontrolle: Es werden nur Pfade mit einem Fluss $\geq \Delta$ gewählt. Wenn es keine solchen Pfade mehr gibt, wird Δ halbiert, und das Spiel wird fortgesetzt. Dieses Verfahren wird ***Capacity Scaling*** genannt.

Der *Capacity Scaling* Algorithmus

Zu einem gegebenen Flussgraphen $G = (V, E)$ und $\Delta > 0$ definieren wir den Subgraphen $G_f(\Delta)$ des Restgraphen G_f, der nur Kanten mit einer Restkapazität von mindestens Δ enthält:

$$E_f(\Delta) = \{v{\to}w \in E \mid rc(v,w) \geq \Delta\}$$

Mit jedem Pfad in $G_f(\Delta)$ kann der Fluss f *um mindestens* Δ *vergrößert* werden.



Listing 12.3 Bestimme den maximalen Fluss eines Flussgraphen G von Quelle s zur Senke t mit *Capacity Scaling*

```
1 C ←maximale Kapazität von Kanten in G
2 Δ ←2^⌊log_2(C)⌋ // kleinste 2-er Potenz ≤ C
3 G_f(Δ) ←Δ–Restgraph von G_f für leeren Fluss f ≡ 0
```

```
4  while Δ ≥ 1
5    while es gibt einen Pfad p von s nach t in G_f(Δ) do
6      cv ←min { rc(e) | e liegt auf Pfad p in G_f(Δ) }
7      // aktualisiere G_f(Δ) entlang p
8      for all Knoten v, w mit v→w auf Pfad p in G_f(Δ)
9        rc(v,w) ←rc(v,w) − cv
10       rc(w,v) ←rc(w,v) + cv
11     end
12   end
13   Δ ←Δ/2
14   aktualisiere G_f(Δ), indem neue Kanten hinzugefügt werden
15 end
```

Bei der Aktualisierung von $\boldsymbol{G}_f(\Delta)$ in Zeile 14 werden alle Kanten mit einer Restkapazität im Intervall $[\Delta,\ 2\Delta[$ zu dem vorigen Graphen $\boldsymbol{G}_f(2\Delta)$ hinzugefügt. Es muss natürlich nur *ein* Restgraph gespeichert werden. Die Abhängigkeit von Δ in dem Code dient nur der inhaltlichen Einordnung.

Korrektheit des *Capacity Scaling* Algorithmus

Für einen Flussgraphen $\boldsymbol{G}$ mit ganzzahligen Kapazitäten bestimmt der *Capacity Scaling* Algorithmus 12.3 den maximalen Fluss.

Beweis. Wenn alle Kapazitäten ganzzahlig sind, dann gilt dies auch für alle Restkapazitäten. Für $\Delta = 1$ gilt also $\boldsymbol{G}_f(\Delta) = \boldsymbol{G}_f$. Daher folgt aus dem Satz über vergrößernde Pfade, dass der Fluss maximal ist, wenn es keine Pfade mehr von s nach t in $\boldsymbol{G}_f$ gibt. □

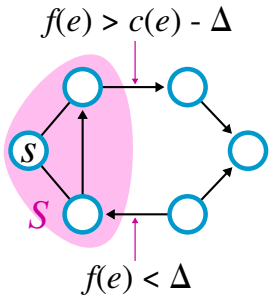
Um die Laufzeit des *Capacity Scaling* Algorithmus zu bestimmen, beweisen wir zunächst den folgenden Hilfssatz:

Hilfssatz für die Laufzeit von *Capacity Scaling*

Am Ende einer Skalierungsiteration (Zeile 12) gilt: $|f^*| \leq |f| + E\Delta$.

Beweis. Wir betrachten den Schnitt $\boldsymbol{S}$ der durch alle von s in $\boldsymbol{G}_f(\Delta)$ erreichbaren Knoten definiert ist und zeigen $c(\boldsymbol{S}, \boldsymbol{V} - \boldsymbol{S}) \leq |f| + E\Delta$, siehe Schnittheorem.
Nach Definition ist $s \in \boldsymbol{S}$ und da am Ende einer Skalierungsiteration die *while*-Bedingung in Zeile 5 nicht erfüllt ist, ist t nicht in $\boldsymbol{G}_f(\Delta)$ erreichbar, also $t \notin \boldsymbol{S}$.

$$
\begin{aligned}
|f| &= \sum_{e \in E_S^{\rightarrow}} f(e) - \sum_{e \in E_S^{\leftarrow}} f(e) && \text{Schnitttheorem, S. 223} \\
&\geq \sum_{e \in E_S^{\rightarrow}} (c(e) - \Delta) - \sum_{e \in E_S^{\leftarrow}} \Delta && \text{Definition von } G_f(\Delta)\text{:} \\
&= \sum_{e \in E_S^{\rightarrow}} c(e) - \sum_{e \in E_S^{\rightarrow}} \Delta - \sum_{e \in E_S^{\leftarrow}} \Delta && \text{Umsortieren der Summen} \\
&\geq c(S, V - S) - E\Delta && \text{Defintion von } c(S, T)
\end{aligned}
$$



Laufzeit des *Capacity Scaling* Algorithmus

Der *Capacity Scaling* Algorithmus (Listing 12.3) bestimmt den maximalen Fluss in einer Laufzeit in $O(E^2 \log C)$.

Beweis. Wir beweisen zunächst die folgenden Teilaussagen **1** – **3** und leiten dann daraus die Laufzeit her.

1 Es gibt höchstens $1 + \lceil \log_2 C \rceil$ Skalierungsphasen

(= Durchläufe der *while*-Schleife, Zeile 4–15).

Beweis. Dies ist der Fall, da Δ anfänglich in $]C/2,\ C]$ gewählt und dann immer halbiert wird. □

2 In jeder Skalierungsphase gibt es höchstens $2E$-viele Flussvergrößerungen

(= Durchläufe der *while*-Schleife, Zeilen 5–12).

Beweis. Am Anfang einer Skalierungsphase gilt nach dem Hilfssatz $|f^*| \leq |f| + E2\Delta$ als Resultat der vorgegangenen Phase für 2Δ. Da jede Flussvergrößerung den Flusswert um Δ erhöht, kann die Anzahl $2E$ nicht überschreiten. (Sonst würde der Flusswert $|f|$ über den Maximalwert $|f^*|$ steigen.) □

3 Jede Flussvergrößerung (Zeile 5–11) benötigt eine Laufzeit in $O(E)$

Suchen eines Pfades p im Restgraphen und Erhöhung von $G_f(\Delta)$ entlang p.

Beweis. Suchen eines Pfades p im Restgraphen benötigt $O(E)$ und Erhöhung von $G_f(\Delta)$ entlang p ebenfalls. □

Nach 1 und 2 werden insgesamt maximal $2E(1 + \lceil \log_2 C \rceil)$ Flussvergrößerungen ausgeführt und laut 3 haben sie jeweils eine Laufzeit in $O(E)$. Die Aktualisierung in Zeile 14 kann ebenfalls in $O(E)$ erfolgen. Die Gesamtlaufzeit ist somit in $O(E^2 \log C)$. ■

12.5 Laufzeiten von *Maxflow* Algorithmen

Die folgende Tabelle zeigt eine Übersicht über die Laufzeiten von *Maxflow* Algorithmen für Flussgraphen mit ganzzahligen Kapazitäten.

Algorithmen zum Finden des Maximalen Flusses		
Algorithmus	***worst-case***	**alternativ**
Ford-Fulkerson	$O(E\|f^*\|)$	$O(EVC)$
Edmonds-Karp	$O(E^2V)$	
blocking-flow	$O(EV^2)$	
blocking-flow mit dynamischen Bäumen	$O(EV \log V)$	
Capacity scaling	$O(E^2 \log C)$	

Dabei ist C die maximale Kapazität, $|f^*|$ der maximale Fluss.

Quellenverweise

Bei der Darstellung wurden viele Ideen von den großartigen Folien von Kevin Wayne zu seiner Vorlesung *Theory of Algorithms* [Wayne, 2013] aufgenommen. Seine Vorlesung orientiert sich seinerseits an den Büchern [Kleinberg and Tardos, 2013] und [Kozen, 2012]. Weiterhin waren [Ottmann and Widmayer, 2011, 639ff] und [Erickson, 2019], sowie [Cormen et al., 2013] wichtige Quellen für dieses Kapitel.

KAPITEL 13

Heuristische Algorithmen

13.1 Motivation für die Verwendung von Heuristiken

Heuristiken nutzen problem-spezifische Informationen, die es erlauben, eine Lösungssuche für eine bestimmte Problemklasse zielgerichteter durchzuführen, als z. B. durch eine uninformierte Durchsuchung des Lösungsraums. **Heuristische Algorithmen** finden eine optimale Lösung (oder zumindest eine 'gute' Lösung) in einem (meist exponentiell) großen Lösungsraum unter Verwendung einer Heuristik. Oft gibt es keine Garantien für eine schnellere Laufzeit im Vergleich zu Ansätzen ohne Heuristik. Manchmal gibt es auch keine Garantie, dass eine optimale Lösung, oder eine Lösung mit vorgegebener maximaler Abweichung vom Optimum, gefunden wird. Dennoch sind einige heuristische Algorithmen in der Praxis sehr nützlich. Dies wird dann anhand von systematischen empirischen Algorithmenanalysen belegt.

- Heuristischen Verfahren folgen meist einem der folgenden Ansätze:

1. Verfahren, die eine Lösung sukzessiv von null einer Heuristik folgend aufbauen
2. Verfahren, die eine schnell hergestellte, suboptimale Lösung schrittweise mit einer Heuristik verbessern

- In der zweiten Variante kommen oft nicht optimale Greedy Algorithmen für die Erstellung einer Ausgangslösung zum Einsatz.

Wir besprechen heuristische Algorithmen zunächst im Rahmen des Problems kürzester Wege, weil sich dabei die Funktionsweise von Heuristiken besonders gut illustrieren lässt. Dabei wird die Heuristik explizit durch eine Funktion auf den Knoten des Graphen repräsentiert. Heuristische Verfahren sind aber auch in vielen anderen Problemfeldern hilfreich. Dies wird am Ende dieses Kapitels

zunächst für den A*-Algorithmus und anschließend an Hand des Problems des Handlungsreisenden diskutiert. In jenem letzten Beispiel ist die Heuristik nicht explizit formuliert. Dort bezeichnet der Begriff Heuristik eher unscharf einen plausiblen Ansatz.

Noch einmal kürzeste Wege

In vielen Anwendungen stellt sich die Frage nach kürzesten Wegen von einem Startpunkt zu *einem Zielpunkt* (z. B. in der Navigation, der Robotik, und in Computerspielen). Zur Modellierung können Graphen verwendet werden. Dabei geht es auch um Wege auf 'freien Flächen'. Die Knoten sind also nicht unbedingt Kreuzungspunkte, sondern können eine Diskretisierung der Fläche sein. In diesen Fällen werden oft Gitternetze als Graphen verwendet. Gleichlange Wegstrecken können unterschiedliche Kosten haben, je nach Begebenheit der Landschaft. Da die Graphen sehr groß sein können, ist Effizienz wichtig.
Entsprechend dem üblichen Gebrauch, benutzen wir bei den folgenden Algorithmen den Begriff *Kosten* eines Pfades für die Summe der Kantengewichte, die wir bisher mit *Gewicht* eines Pfades bezeichnet haben.

Definition: Kosten eines Pfades

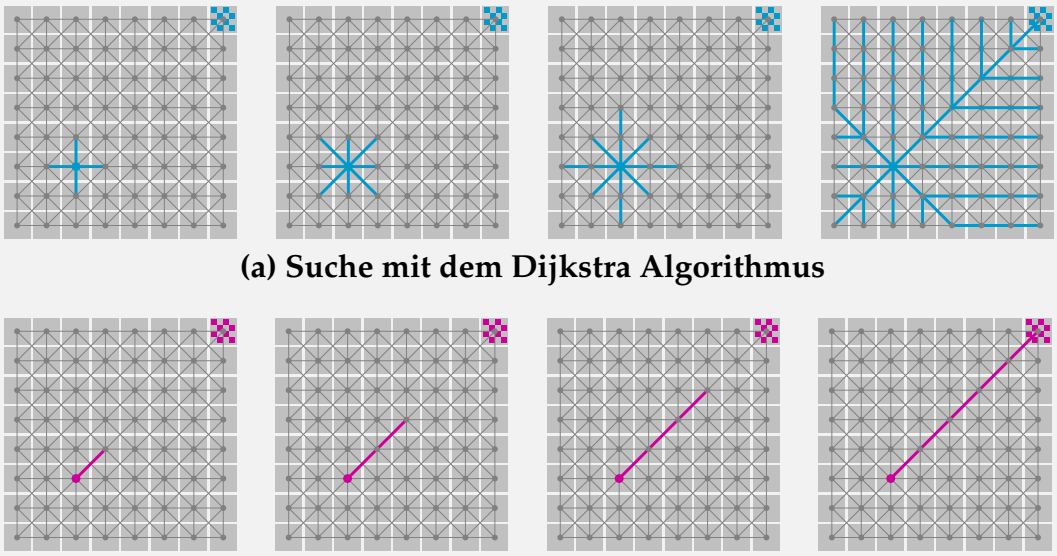
Für einen Pfad p bezeichnen wir mit $c(p)$ die **Kosten des Pfades** (bzw. die Pfadlänge), also die Summe der Gewichte seiner Kanten.

Der kürzeste Weg von s nach t

Finde den kürzesten Weg (Pfad mit geringsten Kosten) von einem Startknoten s zu einem Zielknoten t in einem gewichteten Graphen.

Ansätze mit schon besprochenen Algorithmen

Der Dijkstra Algorithmus findet die kürzesten Wege von s zu *allen anderen* Knoten. Wir können den Algorithmus auch bei gegebenem Ziel benutzen: Sobald der Zielknoten aus der Warteschlange kommt, wird die Suche gestoppt. Dijkstra baut den Suchbaum nach aufsteigender Länge der kürzesten Pfade auf. Die Suche geht gleichermaßen in alle Richtungen – es gibt es keine Ausrichtung auf den Zielknoten. Das ist bei gegebenem Zielknoten *nicht* sonderlich *effizient*. Abbildung 13.1 vergleicht den Dijkstra Algorithmus und eine ideale Suchstrategie für die Suche des kürzesten s-t-Pfades.



(a) Suche mit dem Dijkstra Algorithmus

(b) Ideal (mit welcher Heuristik?)

Abbildung 13.1 Dijkstra durch Heuristik bei s-t Suche verbessern: Es soll der kürzeste Pfad von einem Startknoten zu einem Zielknoten (rechte obere Ecke) gefunden werden. **(a)** Der Dijkstra Algorithmus sucht in alle Richtungen. **(b)** Ideal wäre eine Suchrichtung, die auf das Ziel ausgerichtet ist. Hierbei können Heuristiken helfen. Natürlich soll das Verfahren auch in komplexeren Problemen als dem hier abgebildeten funktionieren.

Mit einer Heuristik $h(v)$, die den Abstand jedes Knotens zum Ziel *schätzt*, können wir vielversprechende Knoten zuerst explorieren. Für kürzeste Wege auf Landkarten ist z. B. die Luftliniendistanz eine solche Schätzung.

Der kürzeste Weg von s nach t einer Heuristik folgend

Als erste Alternative betrachten wir einen Algorithmus, der sich vollständig auf eine Heuristik verlässt. Wir werden sehen, dass dies in etwas komplexeren Graphen zu sehr ineffizienten Suchverläufen führen kann. Daher wird im Anschluss eine Synthese aus dem Dijkstra Algorithmus und dem folgenden BEST-FIRST Algorithmus vorgestellt.

BEST-FIRST - Greedy Algorithmus

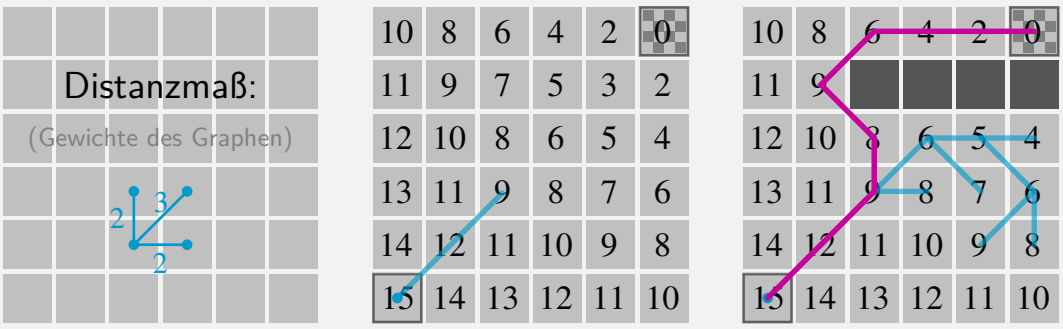
Der **BEST-FIRST Algorithmus** funktioniert ähnlich wie Breitensuche. Er basiert auf einer Heuristik $h(v)$, die für jeden Knoten v eine *Schätzung seines Abstands zum Ziel* angibt. Ausgehend von s wird immer ein neuer Knoten besucht, der von einem der schon besuchten Knoten über eine Kante erreichbar ist. Unter diesen erreichbaren Knoten wird derjenige ausgewählt, dessen

Abstand zum Ziel gemäß der Heuristik am kleinsten ist. Besuchte Knoten werden später nicht noch einmal besucht (Greedy Auswahl).

Listing 13.1 Pseudocode für den BEST-FIRST Algorithmus

```
1  // Gegeben G, Startknoten s, Ziel t
2  Q: PQ mit Priorisierung der Knoten durch Heuristik h
3  M: boolean Array der Größe V
4  parent: int Array der Größe V
5
6  add(Q, s, h(s))
7  M ←{s}
8  while Q ≠ ∅
9    v ← poll(Q)                  // v wird besucht
10   if v = t return true         // Pfad von s nach t gefunden
11   for each w with v→w ∈ E and w ∉ M
12     add w to M                 // w wurde entdeckt
13     parent[w] = v
14     add(Q, w, h(w))
15   end                          // v fertig bearbeitet
16 end
17 return false                   // kein Pfad gefunden
```

BEST-FIRST scheitert bei schwierigeren Beispielen, d. h. der Algorithmus findet nicht die *optimale* Lösung, da er sich zu stark auf die Heuristik verlässt, siehe Abbildung 13.2.



(a) (b) (c)

Abbildung 13.2 **BEST-FIRST in Aktion:** **(a)** zeigt das Distanzmaß, das für alle Abbildungen mit Grid Graphen verwendet wird. **(b)** Bei einem Grid Graphen ohne Blockaden findet der BEST-FIRST Algorithmus in gerader Linie zum Zielknoten. **(c)** Mit Blockaden kann die Heuristik allerdings zu Umwegen und in Sackgassen führen. Der kürzeste Weg zum Zielknoten (magenta) wird erst nach einigen Fehlversuchen (Suchbaum in blau) gefunden.

13.2 Der A* Algorithmus für kürzeste Wege

Der **A* Algorithmus** ("A-Stern", *A-star*) kombiniert

- Vorwärtskosten (*forward cost*): von einem Knoten zum Ziel, geschätzt duch eine Heuristik $h(v)$ mit den

- Rückwärtskosten (*backward cost*): vom Start zu einem Knoten, schrittweise beim Programmablauf aktualisiert.

Genauer: Die Rückwärtskosten entsprechen der Länge des kürzesten bislang bekannten Weges von s nach v, $dist[v]$, die während der Suche wie bei Dijkstra aktualisiert wird. Ausgehend von s wird immer ein neuer Knoten besucht, der von einem der schon besuchten Knoten über eine Kante erreichbar ist. Unter diesen erreichbaren Knoten wird derjenige Knoten v ausgewählt, für den die geschätzte Länge des Weges von s nach t über v

$$f[v] = dist[v] + h(v)$$

am kürzesten ist. Einige Details des Algorithmus und insbesondere die Anforderungen an die Heuristik $h(v)$ werden auf den nächsten Seiten geklärt. In Abbildung 13.3 wird der A* Algorithmus mit Dijkstra und BEST-FIRST in Bezug auf die Reihenfolge der Kantenauswahl verglichen.

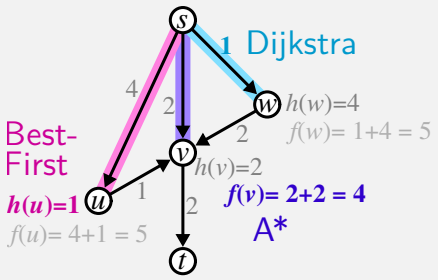


Abbildung 13.3 Vergleich der Kantenauswahl zwischen Dijkstra, BEST-FIRST und A*: In dieser Abbildung wird die Auswahl der ersten Kanten bei den drei Algorithmen für einen gegebenen Graphen und eine gegebene Heuristik verglichen. (Die Heuristik orientiert sich an der Luftliniendistanz und ist ansonsten beliebig festgelegt.) BEST-FIRST wählt den Knoten u, da dieser sich (gemäß Heuristik) am nächsten an dem Zielknoten befindet. Dieser Weg stellt aber einen Umweg zum Ziel dar. BEST-FIRST findet in diesem Beiispiel nicht den kürzesten Weg. Der Dijkstra Algorithmus entscheidet sich für Knoten w, da dieser die kleinste Distanz vom Startknoten besitzt. Dijkstra findet den kürzesten Weg von s nach t später. Der A* Algorithmus berücksichtigt den Abstand in beide Richtungen und entscheidet sich direkt für den richtigen Weg über Knoten v. Für diesen Knoten ist die geschätzte Weglänge $f(v)$ für den Weg von s nach t über v am kürzesten.

Heuristik für A* konkretisieren

Offensichtlich sind nur bestimmte Funktionen als Heuristik hilfreich. Wir diskutieren im Folgenden an Beispielen, welche Probleme bei schlecht gewählten Heuristiken auftreten können und leiten daraus Anforderungen an eine gute Heuristik ab.
Im folgenden Beispiel führt eine ungünstige Heuristik die A* Suche in die Irre, so dass nicht der kürzeste Weg gefunden wird:

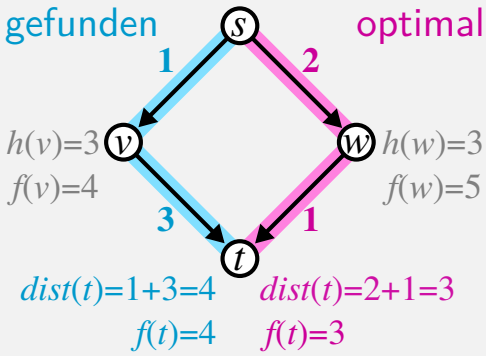


Abbildung 13.4 Mit einer unzulässigen Heuristik kann A* den kürzesten Weg verfehlen. A* wählt den blauen Pfad auf der linken Seite. Kürzer ist der rechte Pfad auf der rechten Seite.

Mit der Heuristik in Abbildung 13.4 wählt A* von s aus zunächst Knoten v mit $f(v) = 4$ und dann Knoten t mit $f(t) = 4$. Wegen $f(t) < f(w)$ wird w nicht besucht und die Suche endet mit dem Pfad $s - v - t$.
A* findet hier also nicht den kürzesten Weg $s \to w \to t$. Das Problem ist der zu hohe h-Wert von w. Da die Heuristik angibt, dass Knoten w weit vom Ziel entfernt ist (3 an Stelle von 1 Einheit), hält A* den Besuch von Knoten w für nicht aussichtsreich. Damit A* keine kürzesten Wege verpasst, treffen wir die folgende Definition:

Definition: Zulässige Heuristik

Wir nennen eine Heuristik h **zulässig** , wenn $h(w)$ die Weglänge von w nach t nicht überschätzt:

Für jeden Pfad p von w nach t muss $h(w) \leq c(p)$ gelten.

Insbesondere gilt $h(t) = 0$, da die Folge von Knoten, die nur aus dem Knoten t besteht, ein Pfad von t nach t ohne Kanten, also mit Kosten 0 ist.

Wir werden später sehen, dass A* mit einer zulässigen Heuristik immer den kürzesten Weg findet. Zunächst wird allerdings noch eine weitere Forderung an die Heuristik besprochen, die für die Effizienz von A* benötigt wird.
A* soll ein effizienter Algorithmus sein. Daher soll jeder Knoten, wie bei Dijkstra, *nur einmal* besucht werden. Selbst bei zulässigen Heuristiken kann es aber erforderlich sein, manche Knoten mehrfach zu besuchen, um eine optimale Lösung zu finden:

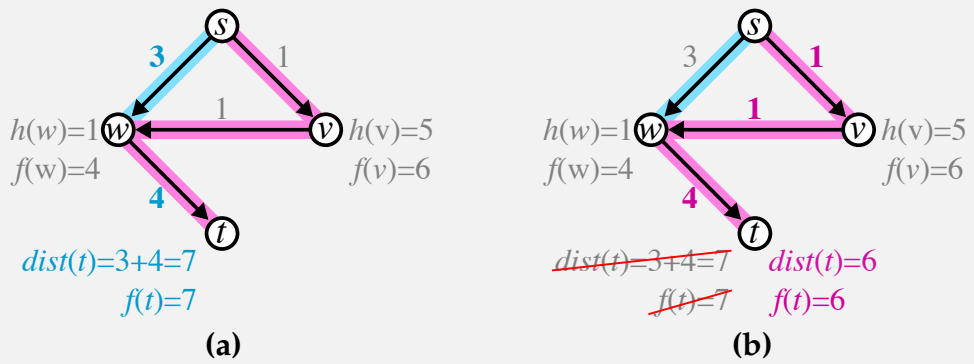


Abbildung 13.5 Inkonsistente Heuristik in A*: (a) A* besucht im ersten Schritt Knoten v wegen $f(v) < f(w)$. Die Kante $s \to v$ wird zu diesem Zeitpunkt als kürzester Weg von s nach v angegeben. **(b)** .In den nächsten Schritten geht der Weg von s über w nach v und schließlich zu t. Der Knoten v wird doppelt besucht.

In dem Beispiel in Abbildung 13.5 wird der Knoten v zweimal besucht. Das Problem ist, dass der h-Wert von w viel höher ist (4 Einheiten) als bei v, obwohl sie nur eine Einheit voneinander entfernt sind. Die Heuristik ist nicht konsistent. Um eine garantiert effiziente Laufzeit zu erzielen, werden *konsistente* Heuristiken vorausgesetzt.

Definition: Konsistente (monotone) Heuristik

Wir nennen eine Heuristik h **konsistent** (*consistent*) oder monoton, wenn $h(t) = 0$ gilt und h die Dreiecksungleichung erfüllt, also

$$h(v) \;\leq\; weight(v \to w) + h(w)$$

für alle Kanten $v \to w$ des Graphen gilt.

Aus Konsistenz folgt Zulässigkeit

Wenn eine Heuristik h konsistent ist, dann ist sie auch zulässig.

Beweis. Wir bezeichnen die Länge bzw. die Kosten (*cost*) eines Pfades p mit $c(p)$. Es ist zu zeigen, dass der Wert der Heuristik an einem Knoten v_0 nicht größer ist, als die Länge eines Pfades von v_0 zum Zielknoten t. Sei also ein Pfad $p = v_0, \ldots, v_k$ mit $v_k = t$ gegeben und wir zeigen $h(v_0) \leq c(p)$:

$$\begin{aligned}
h(v_0) &\leq weight(v_0 \to v_1) + h(v_1) && \text{Def. von Konsistenz}\\
&\leq weight(v_0 \to v_1) + weight(v_1 \to v_2) + h(v_2) && \text{Def. von Konsistenz}\\
&\leq \sum_{i=0}^{k-1} weight(v_i \to v_{i+1}) + h(v_k) && \text{Def. von Konsistenz}\\
&= c(p) && \text{Def. von } c \text{ und } h(v_k) = h(t) = 0
\end{aligned}$$

□

Geschätzte Pfadlängen sind bei konsistenter Heuristik monoton steigend

Bei Verwendung einer konsistenten Heuristik wird die geschätzte Gesamtpfadlänge zum Zielknoten durch Erweiterung eines Pfades nie kürzer.

Beweis. Sei p ein Pfad von s zu einem Knoten v. Zu zeigen: Die geschätzte Pfadlänge nach t kann nicht kleiner werden, wenn der nächste Schritt im Pfad festgelegt wird. Wir betrachten $p + w$, also den Pfad p verlängert um die Kante $v \to w$. Die geschätzte Pfadlänge für p ist $c(p) + h(v)$ und für den verlängerten Pfad $c(p + w) + h(w)$. Aus der Konsistenz von h folgt:

$$\begin{aligned} c(p) + h(v) &\leq c(p) + weight(v \to w) + h(w) && \text{Konsistenz von } h \\ &= c(p + w) + h(w) && \text{Definition von } c \end{aligned}$$

□

Anforderungen an die Heuristik in A*

Auch wenn in manchen, speziellen Fällen A* gezielt mit inkonsistenten Heuristiken eingesetzt wird, ist die Verwendung mit konsistenten Heuristiken der typische Fall. Dann muss kein Knoten mehrfach besucht werden, analog zu dem Fall nicht-negativer Gewichte bei Dijkstra. Viele in der Praxis benutzten Heuristikfunktionen sind konsistent, wie z. B. die Luftlinien-Distanz für Wege auf Landkarten.

Bei Verwendung einer konsistenten Heuristik funktioniert A* wie Dijkstra, mit dem Unterschied, dass die Knoten in der Reihenfolge der f Schätzung exploriert werden. Bei der Implementierung von A* ist der einzige Unterschied zum Dijkstra Algorithmus, dass bei A* zur Priorisierung bei der Knotenwahl `dist[v] + h(v)` an Stelle von `dist[v]` verwendet wird:

Listing 13.2 Pseudocode für den A* Algorithmus

```
1  Q : IndexMinPQ
2  for each node v
3     dist[v] ← ∞
4  end
5  dist[s] ← 0
6  add(Q, s, h(s))          // f(s) = 0 + h(s)
7  while Q ≠ ∅
8     v ← poll(Q)
9     if v = t
10       return dist[t]
11    end
12    for each w with v→w ∈ E
13       relaxAStar(v→w)
14    end
15 end
```

```
16 return inf            // no path from s to t
17 procedure relaxAStar(v→w)
18 if  dist[w] > dist[v] + weight(v→w)
19   parent[w] = v
20   dist[w] = dist[v] + weight(v→w)
21   if contains(Q, w)
22     decreaseKey(Q, w, dist[w] + h(w))
23   else
24     add(Q, w, dist[w] + h(w))
25   end
26 end
```

Korrektheit und Laufzeit von A* mit konsistenter Heuristik

Der A* Algorithmus mit *zulässiger* Heuristik findet den kürzesten Weg zwischen gegebenen Start- und Zielknoten in jedem Graphen $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$. Wenn der Graph keine negativen Kantengewichte hat und eine *konsistente* Heuristik verwendet wird, ist die Laufzeit in $\boldsymbol{O}(E_0 \log V_0)$. Dabei ist V_0 die Anzahl der besuchten Knoten und E_0 die Anzahl der relaxierten Kanten.

Im Prinzip hat A* dieselbe Laufzeit wie Dijkstra. Bei A* heben wir die Abhängigkeit von den *tatsächlich* besuchten Knoten und den *tatsächlich* relaxierten Kanten hervor. Durch die Heuristik ist in vielen Anwendungsfällen $V_0 \ll V$ und $E_0 \ll E$, also A* deutlich schneller als Dijkstra. Aber dafür gibt es keine Garantie. Es gibt auch Fälle in denen $V_0 \approx V$ und $E_0 \approx E$ gilt und A* keinen Vorteil bringt.

Beweis der Laufzeit von A* Mit konsistenten Heuristiken sind wir nach der Monotonie-Eigenschaft in Hinblick auf Laufzeit in derselben Situation wie bei Dijkstra:

- Die Monotonie der *dist*-Werte, die wir bezüglich der Effizienz des Dijkstra Algorithmus bewiesen haben, überträgt sich wegen $f[v] = dist[v] + h(v)$ auf die f-Werte, da sich $h(v)$ beim Programmablauf nicht ändert.
- Die f-Werte, über die v in Zeile 8 ausgewählt wird, sind also *monoton steigend*.
- Die *while*-Schleife wird höchstens V_0-mal ausgeführt.

Also wird jede Kante nur einmal relaxiert und wir erhalten die Laufzeit in $\boldsymbol{O}(E_0 \log V_0)$ wie bei Dijkstra. □

Um die Korrektheit von A* zu beweisen, zeigen wir zunächst den folgenden Hilfssatz.

Hilfssatz für die Korrektheit von A*

Sei p ein Pfad s⇝v dessen Knoten fertig bearbeitet sind, wobei v auch in der PQ sein darf (also noch auf die Bearbeitung wartet). In diesem Fall gilt: $dist[v] \leq c(p)$.

Beweis durch Induktion nach der Anzahl der Kanten von p. Für den Induktionsanfang betrachten wir den Pfad der Länge 0, der nur aus dem Startknoten s besteht. In diesem Fall gilt die Behauptung, da $dist[s] = 0$ ist und auch die Kosten des Pfades 0 betragen.

Sei u der vorletzte Knoten in p und p_0 der Subpfad s⇝u von p. Da u fertig bearbeitet ist, gilt für den Nachbarknoten v die Relaxierungsbedingung

$$dist[v] \leq dist[u] + weight(u \to v)$$

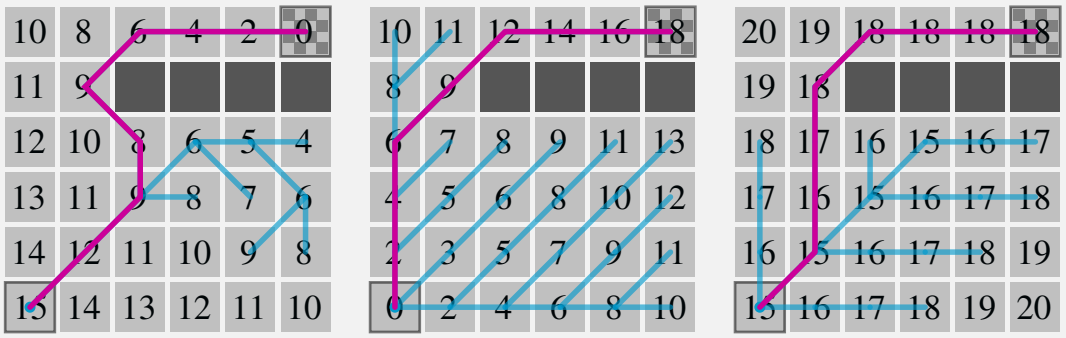
Nach IV gilt $dist[u] \leq c(p_0)$. Damit erhalten wir

$$dist[v] \leq c(p_0) + weight(u \to v) = c(p) \qquad \square$$

Beweis der Korrektheit von A* Wenn A* mit einem gefundenen Pfad terminiert, dann ist $f(t)$ kleiner als die f-Werte aller Knoten in der PQ. In der PQ sind alle Knoten, die über kreuzende Kanten von dem Bereich der bearbeiteten Knoten erreicht werden können. Sei p ein anderer Pfad von s nach t. Dieser Pfad muss durch einen Knoten v laufen, der in der PQ ist. Sei p_0 der Subpfad s⇝v und p_1 der Subpfad v⇝t.

$$\begin{aligned}
dist[t] &= f[t] && \text{da } h(t) = 0 \text{ für zulässiges } h \text{ ist} \\
&\leq f[v] && \text{wegen Priorität der PQ} \\
&= dist[v] + h(v) && \text{Definition von } f \\
&\leq dist[v] + c(p_1) && \text{da } h \text{ zulässig ist} \\
&\leq c(p_0) + c(p_1) && \text{nach dem Lemma} \\
&= c(p)
\end{aligned}$$

Also kann es keinen kürzeren Pfad von s nach t geben. □



(a) BEST-FIRST **(b)** Dijkstra **(c)** A*

Abbildung 13.6 A* im Vergleich mit BEST-FIRST und Dijkstra: Es gelten dieselben Kantengewichte wie in Abbildung 13.2, 2 für horizontale und 3 für vertikale Nachbarn. Die Zahlen in den Quadraten geben die für den jeweiligen Algorithmus relevante Priorität wieder, bei **(a)** BEST-FIRST die Heuristik $h()$, bei **(b)** Dijkstra die Distanz von Startknoten $dist[]$ und bei **(c)** A* die geschätzte Pfadlänge $f[]$. Die Prioritäten $dist[]$ von Dijkstra und $f[]$ von A* werden erst beim Durchlauf für die besuchten Knoten bestimmt und aktualisiert.
A* findet die optimale Lösung und besucht dabei weniger Kanten als Dijkstra. Bei größeren und komplexeren Graphen ist die Einsparung oft sehr viel größer.

13.3 A* jenseits von direkten Graphenproblemen

Die Darstellung von A* war stark an die Suche in *gegebenen* Graphen (mit indizierten Knoten) orientiert. Bei anderen Aufgaben, wie solchen, die im Konext von Backtracking und Branch-and-Bound besprochen wurden, wird der Suchbaum erst bei der Suche erstellt und die Knoten entsprechen Teillösungen, die nicht auf einfache Indizes reduziert werden können. Auch in diesen Fällen kann A* angewendet werden. Die Kosten sind hier die Anzahl der Lösungsschritte zur Lösung. Dafür wird eine Heuristik benötigt, die für jede Teillösung die *Anzahl der Lösungsschritte zum Ziel* schätzt. Damit A* effizient ist, wird *Konsistenz* benötigt: Durch einen Lösungsschritt darf der Wert der Heuristik höchstens um 1 sinken. Mit Zulässigkeit und *ohne Konsistenz* ist man im Prinzip bei Branch-and-Bound.

A* im Baum der Teillösungen: Pseudocode

Für eine Teillösung *psol* sei *psol.f* der f-Wert von A*, also

- Anzahl der Schritte bis Erreichen der Teillösung *psol* plus
- Wert der Heuristik für Teillösung *psol*.

Listing 13.3 Pseudocode für A* im Baum der Teillösungen

```
1  Q : PriorityQueue of partial solutions
2  esol : empty partial solution
3  add(Q, esol , esol .f)
4  while Q ≠ ∅
5    psol ← poll (Q)
6    if psol is solution
7      return psol
8    end
9    for each move possible in psol
10     perform move in psol
11     add(Q, psol , psol .f)
12   end
13 end
14 return null
```

Unterschied zu A* im Graphen mit indizierten Knoten

Warum wird hier kein *relax* verwendet? Bei Graphen wird in *relax* geprüft, ob der Endknoten über die neue Kante schneller erreicht wird, als vorher (falls er

schon vorher entdeckt wurde). In diesem Fall wird der Weg über die neue Kante berücksichtigt.
Anders als im Graphen mit indizierten Knoten, kann hier nicht leicht festgestellt werden, ob die neue Teillösung einer schon gefundenen Teillösung entspricht. Dadurch können äquivalente Teillösungen in die Queue gelangen. Da jeweils die Teillösung mit den wenigsten Schritten der Queue zuerst entnommen wird, ist dies kein Problem für die Korrektheit des Verfahrens. Allerdings kann die Effizienz deutlich leiden, da die Laufzeit der Queue Methoden von der Länge der Queue abhängt. Abhilfe schafft hier die Verwendung von Hash Mengen.

13.4 Heuristiken für das Problem des Handlungsreisenden

Als zweite Herausforderung betrachten wir das

Problem des Handlungsreisenden (*Travelling Salesman Problem*, TSP)

Zu einem vollständigen, gewichteten Graphen soll ein Zyklus mit minimalem Gewicht bestimmt werden, der jeden Knoten genau einmal besucht (TSP-Tour).
Ein Graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ heißt **vollständig**, wenn er alle (nicht-reflexiven) Kanten enthält, also $\boldsymbol{E} = \{v \to w \mid v, w \in \boldsymbol{V} \text{ mit } v \neq w\}$ gilt.

Das TSP ist NP-vollständig und daher eine interessante Herausforderung für die Algorithmenentwicklung. Hier betrachten wir zwei heuristische Algorithmen, die *schnell* Lösungen bestimmen, aber diese Lösungen können weit vom Optimum entfernt sein.

Heuristische Algorithmen für das TSP

Die hier kurz angerissenen heuristischen Ansätze sind ganz anderer Art als der A* Algorithmus. Hier wird die Heuristik nicht explizit formuliert, sondern ist eher vage eine plausible Idee, wie man in die Richtung einer optimalen Lösung gelangen kann.

- **Nearest Insertion:** Fange mit einer Tour an, die nur die beiden am *dichtesten zusammen liegenden* Knoten verbindet.
- Wähle dann iterativ immer denjenigen Knoten aus, der den *kleinsten Abstand* zu einem der Knoten der Tour hat.
- Füge diesen Knoten so in die Tour ein, dass die Zunahme der Tourlänge möglichst klein ist. Der Knoten wird bei der Kante eingefügt, zu der er den geringsten Abstand hat.

- **Farthest Insertion:** Fange mit einer Tour an, die nur die beiden am *weitesten auseinander* liegenden Knoten verbindet.

  - Wähle dann iterativ immer denjenigen Knoten aus, dessen minimaler Abstand zu einem der Knoten der Tour *maximal* ist.
  - Füge diesen Knoten wie bei *Nearest Insertion* in die Tour ein.

Beide Varianten können mit einer Laufzeit in $O(n^2)$ implementiert werden. In der Praxis findet *Farthest Insertion* laut [Schöning, 2001, S. 326] meist bessere Lösungen als *Nearest Insertion*.

13.5 Das Wichtigste in Kürze

Generelles:

- *Heuristische Algorithmen* bezeichnen eine unscharfe Klasse von Algorithmen.
- Bei dem prominentesten Vertreter, A*, wird die Heuristik explizit als eine Funktion formalisiert.
- Bei anderen Algorithmen bezeichnet *heuristisch* nur vage eine Suchstrategie, von der anzunehmen ist, dass sie schnell Richtung (optimale) Lösung führt.
- Generell lässt sich sagen, dass heuristische Algorithmen den Suchraum zunächst in eine Richtung explorieren, in der sich erwartungsgemäß die Lösung befindet.
- Dadurch ergibt sich eine schnelle Laufzeit bei passender Heuristik, aber keine Verbesserung im *worst-case*.
- Viele heuristische Verfahren liefern nur suboptimale Lösungen, ohne eine Garantie, wie nah die gefundene Lösung am Optimum ist.

A* Algorithmus:

- Der A* Algorithmus benutzt eine Heuristik zur Schätzung, wie lang der Weg von einem beliebigen Knoten zum Zielknoten *mindestens* ist. (Für die Navigation ist die Luftliniendistanz eine solche Schätzung.)
- A* nutzt diese Heuristik als *Vorwärtskosten* und verbindetet sie mit den *Rückwärtskosten* (Distanz vom Startknoten aus, wie bei Dijkstra) zu einer Schätzung $f[v]$ der Weglänge vom Start zum Ziel bei Benutzung des Knotens v als Zwischenstopp.

- Diese geschätzte Weglänge $f[v]$ wird als Priorität genutzt für die Auswahl des nächsten Knotens, der bei der Suche benutzt wird. Außer dieser Nutzung von $f[v]$ an Stelle von $dist[v]$ ist die Implementation von A* zur Suche kürzester Wege identisch mit der Implementation des Dijkstra Algorithmus.
- Die Effizienz von A* ist garantiert, wenn die Heuristik *konsistent* ist.

Quellenverweise

Eine sehr schöne, interaktive Webseite zu den Algorithmen Dijkstra, BEST-FIRST und A* zum Finden kürzester Wege zwischen zwei Knoten ist `https://www.redblobgames.com/pathfinding/a-star/introduction.html`. Sie war auch eine Quelle der Inspiration zu Darstellungen in diesem Skript. Als weitere Quelle diente [Schöning, 2001, 319ff].

KAPITEL 14

Approximative Algorithmen

Heuristische Verfahren erlauben es für manche Probleme schnelle Lösungen zu generieren. Diese Lösungen sind oft nicht optimal, und man bekommt keine Garantie, wie stark die Abweichung vom Optimum maximal ist. Eine positive Ausnahme ist der A* Algorithmus, der zumindest bei Verwendung einer konsistenten Heuristik effizient eine *optimale* Lösung findet.
Approximative Algorithmen zielen auf suboptimale Lösungen ab, um schnellere Laufzeiten zu erzielen, und geben eine Schranke für die maximale Abweichung vom Optimum an.

Definition: ρ-Approximationsalgorithmus

Ein Algorithmus wird als **ρ-Approximationsalgorithmus** bezeichnet, wenn sich für alle Probleme sein Lösungswert C höchstens um den Faktor $\rho > 1$ von dem optimalen Wert C^* unterscheidet, also

$$C \geq C^*/\rho \qquad \text{für Maximierungsprobleme}$$

bzw.

$$C \leq \rho C^* \qquad \text{für Minimierungsprobleme}$$

gilt.

14.1 Approximation für das metrische TSP

Wir betrachten wieder das Travelling Salesman Problem (TSP). Bei dem im letzten Kapitel besprochenen heuristischen Ansatz bleibt unklar, wie weit die Lösung vom Optimum entfernt sein kann. Unter der *Voraussetzung*, dass die

Entfernungen zwischen den Knoten die *Dreiecksungleichung* erfüllen (metrisches TSP), gibt es effiziente approximative Algorithmen für das TSP:
Der Christofides Algorithmus bestimmt eine Lösung in $\boldsymbol{O}(V^4)$ mit höchstens 1.5 mal den Kosten des Optimums. Dies ist die beste bekannte Approximation. Allerdings kann es unter der Annahme P≠NP *keine beliebig guten* Approximationen geben. Bislang ist bekannt, dass $\frac{123}{122}$ eine obere Schranke ist [Karpinski et al., 2015]. Wir besprechen hier eine einfachere Variante, die eine Lösung bestimmt, deren Tourlänge höchstens das doppelte des Optimums ist. Es ist also ein 2-Approximationsalgorithmus.
Dieser Ansatz nimmt einen minimalem Spannbaum des gegebenen Graphen als Ausgangsbasis für die approximative Lösung. Der Spannbaum wird in einen Zyklus verwandelt, der jeden Knoten genau einmal besucht.

Listing 14.1 Für Graphen G, dessen Kosten die Dreiecksungleichungen erfüllen, bestimmt dieser Pseudocode eine TSP-Tour, die maximal die doppelten Kosten der optimalen TSP-Tour hat.

```
1 procedure APPROX-TSP-TOUR(G)
2 bestimme einen minimalen Spannbaum T von G mit bel. Startknoten s
3 sei H die Liste der Knoten von T in Nebenreihenfolge
4 H stellt als Zyklus eine TSP−Tour dar
```

Zur Erinnerung: Die Nebenreihenfolge ist die Sortierung der Knoten eines Graphen in der Reihenfolge des Entdecktwerdens bei der Tiefensuche, d. h. aufsteigend nach Eingangsstempel.

Korrektheit und Approximation von APPROX-TSP-TOUR

Der Pseudocode APPROX-TSP-TOUR liefert eine korrekte TSP-Tour (jeder Knoten wird genau einmal besucht) in einer Laufzeit in $\boldsymbol{O}(V^2 \log V)$. Die Tour hat maximal die zweifache Länge der optimalen Tour.

Beweis. Die Laufzeit von Prim für den MST ist in $\boldsymbol{O}(E \log V)$. Die anschließende Tiefensuche zur Sortierung der Knoten ist in $\boldsymbol{O}(V^2)$. Da in der vorliegenden Situation $E \in \boldsymbol{O}(V^2)$ gilt, ist die Laufzeit wie behauptet.
Es bleibt, die 2-Approximationseigenschaft zu zeigen. Eine TSP-Tour wird durch Entfernung einer beliebigen Kante zu einem Spannbaum, also ist die Länge des MST eine untere Schranke für die Länge der optimalen TSP-Tour TSP^*:

$$c(MST) \leq c(TSP^*)$$

Außerdem kann der MST in einen Zyklus umgewandelt werden, der fast eine (suboptimale) TSP-Tour darstellt. Dazu betrachten wir die vollständige Traversierung des MST. In diese Liste werden die Knoten bei einer Tiefensuche sowohl bei dem Besuch eines Knotens (Eingangsstempel), als auch beim Verlassen eines Knotens (Ausgangsstempel) eingetragen. Diese Traversierung W passiert jede Kante des MST zweimal, einmal in jeder Richtung. W bildet einen Zyklus, da die Tiefensuche am Ende wieder zu der Wurzel zurückkehrt und es gilt

$$c(W) \;=\; 2c(MST) \;\leq\; 2c(TSP^*)$$

Allerdings ist der Zyklus W noch keine zulässige TSP Tour, da die Knoten mehrfach, genauer gesagt doppelt, besucht werden. Dies kann durch einen kleinen Umbau des Zyklus behoben werden:
Bei dem Durchlaufen von W wird jeder Knoten, der vorher schon besucht wurde, ausgelassen. Die resultierende Tour nennen wir T. Sie besucht jeden Knoten nur einmal. Durch das Überspringen eines Knotens wird die Länge des Zyklus nicht länger:
Sei $u - v - w$ eine Knotenfolge des ursprünglichen Zyklus, aus der der Knoten v ausgelassen wird. Wegen der Dreiecks-Ungleichung gilt:

$$c(u,w) \;\leq\; c(u,v) + c(v,w)$$

Daher erhalten wir insgesamt die Abschätzung

$$c(T) \;\leq\; c(W) \;=\; 2c(MST) \;\leq\; 2c(TSP^*)$$

Da die Tour T genau der Rückgabe von APPROX-TSP-TOUR entspricht, ist damit die Behauptung bewiesen. □

14.2 Nicht-Approximierbarkeit des allgemeinen TSP

Nun folgt das ernüchternde Resultat, dass es für das allgemeine Handlungsreisendenproblem (also ohne die Voraussetzung der Dreiecksungleichung) *keine* Approximation geben kann, nicht einmal eine schlechte.

Das allgemeine Handlungsreisendenproblem ist nicht approximierbar

Unter der Voraussetzung P≠NP gibt es für kein $\rho \geq 1$ einen ρ-Approximationsalgorithmus mit polynomieller Laufzeit für das allgemeine Handlungsreisendenproblem.

Beweis. Wir nehmen an, dass es für ein $\rho \geq 1$ einen ρ-Approximationsalgorithmus $\mathcal{X}$ mit polynomieller Laufzeit für das TSP gibt. Da die Approximation dann auch für alle größeren Zahlen gilt, können wir ρ als *ganzzahlig* voraussetzen (z. B. durch Aufrunden). Wir zeigen, dass sich mit $\mathcal{X}$ auch das Hamiltonzyklus Problem lösen lässt.

Sei ein Graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ gegeben, für den die Existenz eines Hamiltonzyklus festgestellt werden soll. Wir definieren den vollständigen und gewichteten Graphen $\boldsymbol{G}' = (\boldsymbol{V}, \boldsymbol{E}', c)$ durch

$$\boldsymbol{E}' = \{(v, w) \in \boldsymbol{V} \times \boldsymbol{V} \mid v \neq w\}$$

$$c(v, w) = \begin{cases} 1 & \text{falls } (v, w) \in \boldsymbol{E} \\ \rho V + 1 & \text{sonst} \end{cases}$$

Offensichtlich kann $\boldsymbol{G}'$ in polynomieller Zeit in V und E aus $\boldsymbol{G}$ erstellt werden. $\boldsymbol{G}'$ ist so definiert, dass der ρ-Approximationsalgorithmus $\mathcal{X}$ für das TSP in $\boldsymbol{G}'$ die Frage nach dem Hamilton-Zyklus in $\boldsymbol{G}$ beantwortet!
Nach Konstruktion unterscheiden sich die Kosten für die optimale TSP-Tour um mehr als den Faktor ρ, abhängig davon, ob ein Hamilton-Zyklus in $\boldsymbol{G}$ exisitert oder nicht. Daher könnte der ρ-Approximationsalgorithmus $\mathcal{X}$ die Hamilton-Frage entscheiden: Wir wenden $\mathcal{X}$ auf das TSP $\boldsymbol{G}'$ an und unterscheiden nach den Kosten der gefundenen Tour T:

1. $c(T) \leq \rho V$: Die Tour enthält nur Kanten, die zu $\boldsymbol{G}$ gehören, da alle anderen Kanten Kosten $\rho V + 1$ haben. Daher stellt die TSP-Tour T auch einen Hamilton-Zyklus in $\boldsymbol{G}$ dar.

2. $c(T) > \rho V$: Die optimale Tour kann sich maximal um den Faktor ρ unterscheiden, hat also Kosten $> V$. Da ein Hamilton-Zyklus in $\boldsymbol{G}$ eine TSP Tour mit Kosten V wäre, kann es diesen nicht geben, wenn die optimale TSP Tour Kosten $> V$ hat.

14.3 Approximationsschema für das 0/1-Rucksackproblem

Mittels dynamischer Programmierung konnte ein Algorithmus für das 0/1-Rucksackproblem mit pseudopolynomieller Laufzeit formuliert werden. Die Laufzeit in $\boldsymbol{O}(KW)$ kann für große Kapazitätsgrenzen W sehr schlecht sein. Mit einem approximativen Algorithmus kann man effizient Lösungen finden, die beliebig nah am Optimum sind. Wenn im Rahmen von dynamischer Programmierung die Laufzeit verbessert werden soll, muss die Tabelle verkleinert

werden. Im Sinne einer approximativen Lösung ist dies im Prinzip durch eine Skalierung einer Variable der OPT Funktion möglich. Die Anzahl der Objekte kann natürlich nicht skaliert werden. Aber auch eine Skalierung der Kapazitätsgrenze ist nicht zielführend, da die Kapazitätsgrenze exakt eingehalten werden muss. Im Gegensatz dazu können die Werte der Objekte skaliert werden. Daher erstellen wir nun einen neuen dynamischen Programmier-Ansatz, bei dem die OPT Funktion von dem zu erreichenden Wert abhängt.

0/1-Rucksack Problem – Duales Dynamisches Programm

Wir definieren eine rekursive Funktion $\text{OPT}(k, v)$, die angibt wieviel Gewicht mindestens notwendig ist, um mit einer Auswahl aus Objekten $1, \ldots, k$ einen vorgegebenen Wert v (oder mehr) zu erreichen, bzw. ∞, falls der Wert v gar nicht mit den Objekten erreicht werden kann.

- V sei die Summe aller Werte: $V = \sum_{k=1}^{K} v_k$.
- Der Definitionsbereich von OPT ist $0 \leq k \leq K$ und $0 \leq v \leq V$.
- Den Wert $v = 0$ erreicht man ohne Objekte, also ist die Gewichtsgrenze $\text{OPT}(k, 0) = 0$ für alle k.
- Mit $k = 0$ Objekten, kann man keinen Zielwert $v > 0$ erreichen, also ist der Funktionswert ∞.
- Für alle anderen Werte von k von v werden die beiden Fälle unterschieden, ob Objekt k ausgewählt wird, oder nicht.
- Also definieren wir $\text{OPT}(k, v)$ auf die folgende Weise:

$$\text{OPT}(k, v) = \begin{cases} 0 & \text{falls } v = 0 \\ \infty & \text{falls } k = 0 \,\&\, v > 0 \\ \max(\underbrace{w_k + \text{OPT}(k-1, \max(0, v - v_k))}_{k \text{ ausgewählt}}, \underbrace{\text{OPT}(k-1, v)}_{k \text{ nicht ausgewählt}}) & \text{sonst} \end{cases}$$

Korrektheit und Laufzeit der dualen 0/1-Rucksack Lösung

Der Algorithmus basierend auf dynamischer Programmierung mit obiger OPT Funktion findet die optimale Lösung des 0/1-Rucksack Problems mit ganzzahligen Werten und Gewichten in einer Laufzeit in $\boldsymbol{O}(KV)$, wobei V die Summe der Werte aller K Objekte ist: $V = \sum_{k=1}^{K} v_k$.

Beweis. Die Laufzeit der dynamischen Programmierung ergibt sich aus der Größe der Tabelle und der Laufzeit der Bestimmung eines Tabelleneintrages, sowie der Zeit die benötigt wird, um den Lösungswert aus der Tabelle zu suchen.

Der maximal erreichbare Wert ist offensichtlich V. Daher wissen wir für die Tabelle:

- Die Tabelle hat die Größe $(K+1)(V+1)$.
- Das Berechnen jedes Tabelleneintrags kann gemäß der OPT Funktion in konstanter Zeit ausgeführt werden.

Insgesamt kann also die Tabelle in einer Laufzeit in $\boldsymbol{O}(KV)$ bestimmt werden.

Der Lösungswert steht nicht unbedingt am Ende der Tabelle, in dem Eintrag (K, V), sondern er muss aus der untersten Zeile der Tabelle herausgesucht werden: Man sucht den größten Wert v, für den $OPT(K, v) \leq W$ gilt. Dies kann in einer Laufzeit in $\boldsymbol{O}(V)$ erledigt werden. Daher bleibt es für die Gesamtlaufzeit bei der Wachstumsordnung $\boldsymbol{O}(KV)$. □

Vergleich der Lösungsvarianten für das Rucksackproblem:

- Laufzeit von Variante 1 mit $\text{OPT}(k, W)$ ist in $\boldsymbol{O}(KW)$.
- Laufzeit von Variante 2 mit $\text{OPT}(k, v)$ ist in $\boldsymbol{O}(KV)$.
  Für den maximalen Wert der Objekte $\bar{v} = \max_k(v_k)$ erhalten wir die gröbere Abschätzung $\boldsymbol{O}(K^2\bar{v})$.

Welche Variante effizienter ist, hängt von der Kapazitätsgrenze und der Größe der Werte ab. Die zweite Variante hat den Vorteil, dass sie die Grundlage für ein Approximationsschema liefert.

Approximationsschema

Ein **Approximationsschema** für ein Optimierungsproblem ist ein Algorithmus, der zu jeder Eingabe und jedem $\varepsilon > 0$ eine $(1+\varepsilon)$-Approximation liefert.

Nun ist nicht nur interessant, wie sich die Laufzeit in Abhängigkeit von der Eingabe verhält, sondern auch, wie die *Laufzeit von ε abhängt*. Man nennt ein Approximationsschema ein **Approximationsschema mit vollständig polynomieller Laufzeit**, falls seine Laufzeit polynomiell in der Größe der Eingabe und in $\frac{1}{\varepsilon}$ ist.

Approximationsschema für das 0/1-Rucksack Problem

Seien K Objekte mit Gewichten w_k und Werten v_k sowie eine Kapazitätsgrenze W gegeben. Sei $\bar{v} = \max_k(v_k)$ der maximale Wert eines Objektes. Zu der gegebenen Approximationsgüte $\varepsilon > 0$ definieren wir $E = \varepsilon\frac{\bar{v}}{K}$. Wir gehen davon aus, dass es keine Objekte mit einem Gewicht $w_k > W$ gibt. Dies ist keine Einschränkung, weil solche Objekte sowieso nicht ausgewählt werden könnten.
Wir benutzen E als Skalierungsfaktor und berechnen neue Werte $v'_k = \lfloor\frac{v_k}{E}\rfloor = \lfloor\frac{K}{\varepsilon}\frac{v_k}{\bar{v}}\rfloor$. Die skalierten Werte liegen also zwischen 0 und $\frac{K}{\varepsilon}$. Nun wird der zuvor skizzierte Algorithmus auf das Problem mit den skalierten Werten v'_k angewendet.

Approximationsschema für den 0/1-Rucksack

Der oben beschriebene Algorithmus ist ein Approximationsschema mit vollständig polynomieller Laufzeit.

Beweis. Approximationseigenschaft: Sei S eine optimale Lösung für das Originalproblem, die Wert $V^* = \sum_{k\in S} v_k$ erzielt und S' eine optimale Lösung für die skalierten Werte v'_k, die von dem Approximations-Algorithmus verwendet wird. Da S' optimal für die skalierten Werte v'_k ist, erhalten wir:

$$\sum_{k\in S'} v'_k \;\geq\; \sum_{k\in S} v'_k \;=\; \sum_{k\in S} \lfloor\frac{v_k}{E}\rfloor \;\geq\; \sum_{k\in S}(\frac{v_k}{E} - 1) \;\geq\; \frac{V^*}{E} - K$$

Nun können wir die Werte der Lösung des Approximations-Algorithmus bezogen auf die Originalwerte abschätzen:

$$\sum_{k\in S'} v_k \;\geq\; \sum_{k\in S'} v'_k E \;\geq\; (\frac{V^*}{E} - K)E \;=\; V^* - KE \;\geq\; V^*(1-\varepsilon)$$

wobei die letzte Ungleichung mit $\bar{v} \leq V^*$ aus $E \;=\; \varepsilon\frac{\bar{v}}{K} \;\leq\; \varepsilon\frac{V^*}{K}$ folgt. □

Laufzeit: Die Originaltabelle hat die Größe $K \times V \;\leq\; K \times K\bar{v}$, also ist die skalierte Tabelle kleiner oder gleich $K \times \frac{K\bar{v}}{E} \;=\; K \times \frac{K\bar{v}K}{\varepsilon\bar{v}} \;=\; K \times \frac{K^2}{\varepsilon}$. Somit ist die Laufzeit in $O(\frac{K^3}{\varepsilon})$. □

Approximative Variante von A*

Man kann den A* Algorithmus beschleunigen, wenn man suboptimale Lösungen in Kauf nimmt. Eine einfache Variante ist der **Gewichtete A*** Algorithmus (*weighted A**). Dabei wird eine gegebene konsistente Heuristik h mit einem

konstanten Faktor $\varepsilon > 0$ multipliziert. Dadurch werden wenig Knoten besucht, also terminiert der Algorithmus schneller. Der gefundene Pfad ist maximal um den Faktor ε länger als der optimale Pfad. Durch die Skalierung verliert die Heuristik meist ihre Konsistenz. Daher kann der oben genannte Pseudocode nicht verwendet werden.

14.4 Das Wichtigste in Kürze

- Approximative Algorithmen geben eine Garantie, wie nah sie an die optimale Lösung kommen.
- Sie werden als effizienter Ansatz für NP-vollständige Probleme genutzt.
- Bei der Approximation von Optimierungsproblemen kann folgendes passieren:

1. Das Problem lässt sich *überhaupt nicht* approximieren, egal wie lax die Approximationsgüte angesetzt wird, siehe das Beispiel TSP ohne Dreiecksungleichung
2. Das Problem lässt sich approximieren, aber *nur bis zu einer gewissen Grenze*, siehe das Beispiel TSP mit Dreiecksungleichung
3. Das Problem kann *beliebig gut* approximiert werden, siehe das Beispiel 0/1-Rucksack

- Bei dem letzten Punkt unterscheidet man noch danach, wie stark die Approximationsgüte die Laufzeit beeinträchtigt.
- Im guten Fall ist die Laufzeit polynomiell in Eingabegröße und $\frac{1}{\varepsilon}$. In diesem Fall sprich man von einem Approximationsschema mit vollständig polynomieller Laufzeit.

Quellenverweise

Sehr gute Darstellungen Approximativer Algorithmen finden sich in [Cormen et al., 2013, S. 1117ff] und [Dasgupta et al., 2008, S. 279ff]. Insbesondere der Abschnitt zur Nicht-Approximierbarkeit des allgemeinen TSP orientiert sich an diesen beiden Büchern. Der duale Ansatz mit Dynamischer Programmierung für das 0/1-Rucksackproblem ist in [Röglin, 2011, S. 19] beschrieben und [Ottmann and Widmayer, 2011, S. 454ff] beschreibt das Approximationsschema.

KAPITEL 15

Randomisierte Algorithmen

Algorithmen, die Zufallsentscheidungen einbeziehen, heißen **Randomisierte Algorithmen** (*randomized algorithms*). Das universelle Hashing, welches in Abschnitt 16.5 noch besprochen wird, ist ein randomisierter Algorithmus. Dabei wurde die Randomisierung insbesondere eingesetzt, um gezielte Sabotage durch hervorgerufene Kollisionshäufungen zu vermeiden.
Bei Optimierungsproblemen unterscheidet man zwischen zwei unterschiedlichen Arten von randomisierten Algorithmen:

Las-Vegas-Algorithmen

Las-Vegas-Algorithmen bestimmen immer die *optimale Lösung* und im Erwartungswert ist die Laufzeit sehr effizient[a]. Allerdings kann die Laufzeit selbst bei der gleichen Eingabe stark variieren.

[a] *Effizient* ist hier im Vergleich mit Laufzeiten nicht-randomisierter Algorithmen für dasselbe Problem gemeint.

Ein Beispiel ist Randomisiertes Quicksort. Es hat als Erwartungswert eine Laufzeit in $\boldsymbol{O}(n \log n)$ und im *worst-case* in $\boldsymbol{O}(n^2)$.

Bei einer Variante der Las-Vegas-Algorithmen wird auch erlaubt, dass sie keine Lösung liefern. Aber wenn sie eine Lösung zurückgeben, muss es eine optimale sein.

Monte-Carlo-Algorithmen

Monte-Carlo-Algorithmen terminieren immer in *effizienter Laufzeit*[a], aber das Ergebnis ist nur mit einer gewissen Wahrscheinlichkeit optimal.

[a] *Effizient* ist hier im Vergleich mit Laufzeiten nicht-randomisierter Algorithmen für dasselbe Problem gemeint.

Ein Beispiel ist der *Monte Carlo Tree Search* Algorithmus. Er wird z. B. erfolgreich bei Spielen eingesetzt (u. a. in Alpha Go). Im nächsten Abschnitt besprechen wir ein anderes Beispiel für einen Monte-Carlo-Algorithmus.

Jeder Las-Vegas-Algorithmus kann in einen Monte-Carlo-Algorithmus umgewandelt werden (Abbruchkriterium nach polynomieller Laufzeit und Ausgabe der bis dahin besten Lösung).

15.1 Beispiel: Maximal azyklischer Untergraph

Gerichtete azyklische Graphen (DAGs) sind aus algorithmischer Sicht sehr angenehme Objekte. Viele Operationen lassen sich für DAGs sehr effizient durchführen. Daher wandelt man bei manchen Anwendungen Digraphen durch das Entfernen von Kanten, die Zyklen verursachen, in DAGs um.
Dafür muss zu einem gegebenen, gerichteten Graphen ein *maximal azyklischer Untergraph* (*maximum acyclic subgraph*) gefunden werden. Das ist ein NP-vollständiges Problem. Wir besprechen eine randomisierte Methode als Beispiel für einen Monte-Carlo-Algorithmus. Das Ergebnis ist also nicht unbedingt eine optimale Lösung.

Monte-Carlo-Algorithmus für einen maximal azyklischen Untergraphen

Es sei ein gerichteter Graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ gegeben. Die Knoten seien in einer beliebigen, *zufälligen* Reihenfolge sortiert. Wir definieren zwei Untergraphen $\boldsymbol{G}_1$ und $\boldsymbol{G}_2$ durch die Kantenmengen

$$\boldsymbol{E}_1 = \{v \to w \in \boldsymbol{E} \mid v < w\}$$

$$\boldsymbol{E}_2 = \{v \to w \in \boldsymbol{E} \mid v > w\}$$

Auf Grund der Definition sind beide Subgraphen zyklenfrei. Da $\boldsymbol{G}_1$ und $\boldsymbol{G}_2$ zusammen $\boldsymbol{G}$ ergeben, muss einer der beiden Graphen mindestens halb so groß sein, wie der ursprüngliche Graph. Wir haben also einen Subgraphen von $\boldsymbol{G}$ gefunden, der zyklenfrei und mindestens halb so groß ist wie $\boldsymbol{G}$. Er ist also 'relativ' groß, aber vermutlich nicht maximal.

Der Algorithmus ist sehr simpel, und die gefundene Lösung ist wahrscheinlich sehr weit von der Optimalen entfernt. Warum verdient er dennoch eine Erwähnung? Der Algorithmus ist sehr schnell und bietet daher eine gute Ausgangsbasis für Verbesserungen. Man kann den Algorithmus z. B. häufig mit

zufälligen Permutationen der Knotenreihenfolge starten und am Ende den größten aller erzeugten Subgraphen auswählen. Hierbei kommt der Aspekt des Randomisierens stärker zur Geltung, als bei der einmaligen Ausführung.
Ein anderer Ansatz ist es, die Lösungen schrittweise zu verbessern, z. B. durch Vertauschen von Kanten in der Permutation und Beibehalten von Vertauschungen, die das Ergebnis verbessern.

15.2 Das Wichtigste in Kürze

- Randomisierte Algorithmen liefern optimale Lösungen, haben aber keine *worst-case* Garantie für die Laufzeit (Las-Vegas) oder
- sie haben eine garantiert schnelle Laufzeit, liefern aber *suboptimale* Lösungen, ohne eine Schranke für die Approximationsgüte (Monte-Carlo).

Quellenverweise

Schöne Darstellungen der randomisierten Algorithmen bieten [Ottmann and Widmayer, 2011, S. 710ff] und [Röglin, 2011]. Die Beispielanwendung für das Problem eines maximalen azyklischen Untergraphen ist [Skiena, 2012, S. 348] entnommen.

KAPITEL 16

Hashtabellen

Hashing ist eine Methode zur dynamischen Verwaltung von Daten. In Hashtabellen können Informationen sehr schnell gespeichert und wieder abgerufen werden. Dabei werden oft Schlüssel-Wert (*key-value*) Paare verwendet. Die Information (der Wert) wird unter Verwendung eines Schlüssels gespeichert und abgerufen. Diese Funktionalität wird *Symboltabelle* (*symbol table*; auch *associative array*, *map*) oder *Wörterbuch* (*dictionary*) genannt. Für ein normales Lexikon wäre das Stichwort der Schlüssel und die Definition der Wert.

Die minimalen Anforderungen an die Funktionen einer Symboltabelle werden durch das folgende Interface spezifiziert:

```
interface SymbolTable<K,V> {
  V get(K key);
  void put(K key, V value);
  void delete(K key);
}
```

Symboltabellen können leicht über *Arrays* oder *verkettete Listen* implementiert werden, siehe Abschnitt A.2 auf Seite 291 im Anhang.

Die Implementierung mit einem Array ist sehr einfach und die Operationen (*get*, *put* und *delete*) sind mit ihrer Laufzeit in $\boldsymbol{O}(1)$ sehr schnell. Im Gegensatz dazu ist die Laufzeit aller Methoden bei der Verwendung verketteter Listen in $\boldsymbol{O}(N)$ (wobei N die Anzahl der bereits gespeicherten Elemente ist).

Die unterschiedlichen Implementierungen erzielen also die folgenden Laufzeiten (N = Anzahl der bereits gespeicherten Elemente):

Algorithmus / Datenstruktur	durchschnittlicher Fall		schlimmster Fall	
	get	put	get	put
Schlüssel-indiziertes Array	$O(1)$	$O(1)$	$O(1)$	$O(1)$
balancierter Suchbaum	$O(\lg N)$	$O(\lg N)$	$O(\lg N)$	$O(\lg N)$
Binärer Suchbaum	$O(\lg N)$	$O(\lg N)$	$O(N)$	$O(N)$
verkettete Liste	$O(N)$	$O(N)$	$O(N)$	$O(N)$

Nach der Laufzeit schneidet also Array Implementierung eindeutig am besten ab. Allerdings ist sie nur möglich, falls die Schlüssel nicht-negative, ganzzahlige Werte in einem relativ engen Bereich sind. Sind die Schlüssel aber komplexere Objekte (z. B. Zeichenketten, Spielstellungen etc) oder Zahlen aus einem großen Zahlenbereich (z. B. Matrikelnummern), so ist eine direkte Umsetzung einer Symboltabelle mit einem Array nicht möglich, oder nur mit einer großen Speicherverschwendung. Die Implementierung über eine verkettete Liste hätte damit kein Problem.

Beispiel zur Motivation von Hashtabellen

Ein kleines Beispiel: Von einer gegebenen Zeichenkette soll für alle vorkommenden *Buchstaben* ausgegeben werden, wie oft sie vorkommen. Dies könnte einfach mit einem Array der Größe 256 (UFT-8 Kodierung vorausgesetzt) umgesetzt werden. Die ungenutzten Einträge des Array (nicht-vorkommende Zeichen) stören nicht besonders.

Aber wie sollte die Aufgabe, die Häufigkeit vorkommender *Wörter* in einem Buch zu zählen, gelöst werden? Die numerische Darstellung der Zeichenketten hat viel zu hohe Werte, um sie als Indizierung in einem Array zu nutzen. Von den besprochenen Möglichkeiten bleibt nur die Implementierung über eine verkettete Liste, die für die geforderten Aufgaben allerdings keine gute Laufzeit hat.

Die in diesem Abschnitt besprochenen Hashtabellen erlauben einen Mittelweg zwischen den Implementationen mit Arrays und verketteten Listen. Die Speicherung der Information ist deutlich effizienter als bei Arrays und die Laufzeit ist deutlich kürzer als bei verketteten Listen. In den meisten Fällen haben die Operationen in Hashtabellen eine Laufzeit nah an $O(1)$. Innerhalb von Hashing kann durch die Wahl von Parametern der Trade-off zwischen Speicher/Rechenzeit beeinflusst werden.

Hashtabellen für Schlüsselmengen oder Schlüssel-Werte Paare

Obwohl die Werte einer Symboltabelle große Objekte sein können, spielen sie bei unseren Betrachtungen keine besondere Rolle. Sie liegen so effizient wie möglich im Speicher und die Symboltabelle verwaltet nur Referenzen auf die Objekte. Es geht hier also 'nur' um das effiziente Speichern und Verwalten von Schlüsseln (Eintragen, Suchen, Löschen). Der Aspekt der Werte wird im Folgenden daher meist ausgespart. Bei einigen Anwendungen hat man *gar keine Werte* und es geht nur um die Verwaltung von Schlüsseln in einer **Hashmenge** (*hashset*). In den Java Collections gibt es hierfür die Klasse `HashSet<K>` und für Schlüssel-Werte Paare `HashMap<K,V>`. Siehe auch die praktischen Hinweise auf Seite 284.

16.1 Hashfunktionen

Die Menge aller möglichen Schlüssel wird als Universum, $\boldsymbol{U}$ bezeichnet. Diese Menge kann sehr, sehr groß sein (z. B. die Menge aller Zeichenketten, oder die Menge aller Stellungen des Schachspiels). Die Menge der in einer Anwendung tatsächlich verwendeten Schlüssel $\boldsymbol{K}$ ist im Vergleich zu $\boldsymbol{U}$ klein: $|\boldsymbol{K}| \ll |\boldsymbol{U}|$. Bei einer direkten Indizierung eines Arrays mit Schlüsseln wird viel Speicher verschwendet, siehe Abbildung 16.1 (a), und in vielen Fällen überschreitet sie den Speicherumfang. (Zeichenketten als Integer ergeben schnell Werte, die viel größer als Speicheradressen sind.)

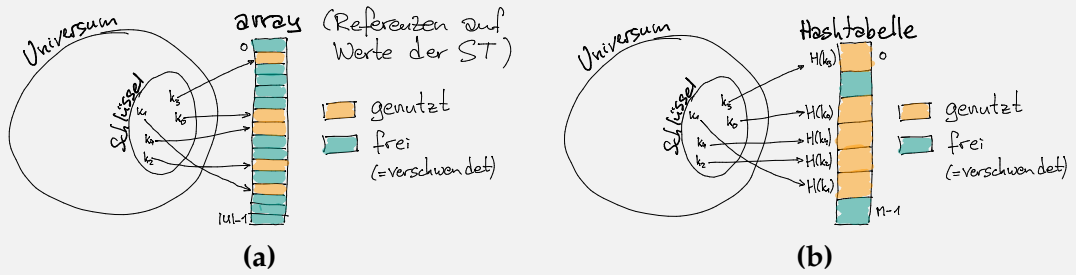


Abbildung 16.1 Speicherung von Schlüsseln in einer Hashtabelle: Eine kleine, aber im voraus nicht bekannte, Menge von Schlüsseln soll gespeichert werden. Die Schlüssel stammen aus einem großen Universum $\boldsymbol{U} \subseteq \mathbb{N}_0$. **(a)** Bei einer Speicherung in einem Array mit den Schlüssel als Indizes bleiben große Lücken. Sehr viel Speicher bleibt ungenutzt. **(b)** Bei Hashtabellen wird eine Hashfunktion $h()$ verwendet, die alle Schlüssel des Universums auf einen kleinen Wertebereich $0, \ldots, M-1$ mit $M \ll |\boldsymbol{U}|$ abbildet. (ST = Symboltabelle)

Definition Hashfunktion

Hashtabellen beruhen darauf, dass die Schlüssel, die aus einem Universum $\boldsymbol{U}$ stammen, mit einer **Hashfunktion**

$$H : \boldsymbol{U} \to \{0, \ldots, M-1\}; \quad k \mapsto H(k)$$

auf Werte $< M$ mit $M \ll |\boldsymbol{U}|$ abgebildet werden (*hashing*). Wir gehen im Folgenden davon aus, dass $\boldsymbol{U} \subseteq \mathbb{N}_0$ gilt und schreiben $h()$ für Hashfunktionen, die auf den natürlichen Zahlen definiert sind.

In Java wird die Methode `hashCode()` verwendet, um beliebige Objekte in ganze Zahlen umzuwandeln. Auf diese Hashcodes kann eine Hashfunktion $h()$ angewendet werden, die auf den natürlichen Zahlen definiert ist.

Im Folgenden gehen wir davon aus, dass die Schlüssel natürliche Zahlen sind, also $\boldsymbol{U} \subseteq \mathbb{N}_0$ gilt. Die Umwandlung von beliebigen Java Objekten in natürliche Zahlen wird in dem Abschnitt über Hashcodes genauer dargestellt.
Mit den oben definierten Hashfunktionen sollen die möglichen Schlüssel in einen kleinen Wertebereich abgebildet werden. Da $M \ll |\boldsymbol{U}|$ ist, können dabei **Kollisionen** auftreten. Dies bezeichnet die Situation, dass zwei unterschiedliche Schlüssel $k_0, k_1 \in \boldsymbol{U}$ $(k_0 \neq k_1)$ auf gleiche Werte abgebildet werden: $H(k_0) = H(k_1)$. Kollisionen verursachen ein Problem bei der Speicherung in einer Hashtabelle. Sie müssen also *aufgelöst* werden.
Hashverfahren besitzen die beiden folgenden Elemente:

1. **Hashfunktion:** Sie sollte schnell zu berechnen sein, und die Schlüssel möglichst gleichmäßig auf die Hashadressen verteilen.

2. **Verfahren zur Auflösung von Kollisionen:** Dies sollte effizient sein und möglichst nicht Kollisionen an anderer Stelle provozieren.

Durch die Wahl des Hashverfahrens (und seiner Parameter) kann der *Trade-off* zwischen Speicher- und Laufzeit-Effizienz gewählt werden.

16.1.1 Kollisionen und perfektes Hashing

Eine Kollision ist das Ereignis, dass zwei unterschiedliche Schlüssel per Hashfunktion auf dieselbe Hashadresse abgebildet werden. Sie treten eher auf, wenn die Schlüssel nicht gleichmäßig verteilt sind. Dies ist meistens der Fall. (Compilerbeispiel: Variablennamen sind nicht zufällig, sondern oft ähnlich.)

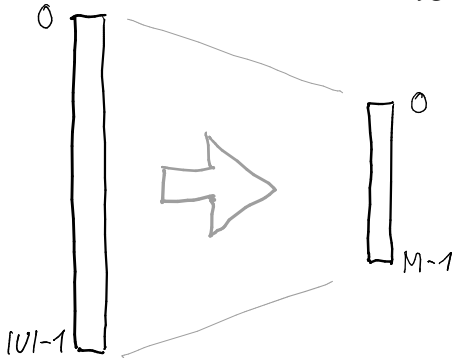
Ist die Menge der Schlüssel im voraus bekannt, so lässt sich eine kollisionsfreie Hashfunktion definieren: **perfektes Hashing**. Dazu können z. B. die Schlüssel sortiert werden, und jeder Schlüssel wird auf die Ordnungszahl in der Sortierung

abgebildet. Dieser Fall, das die Schlüsselmenge im voraus bekannt ist, ist in der Anwendung eher untypisch.

Kollisionen und Hashfunktionen

Gehen wir also davon aus, dass die Voraussetzung für perfektes Hashing nicht erfüllt ist. Kollisionen lassen sich nicht vermeiden, selbst wenn die Hashfunktion geschickt gewählt ist. Die Wahrscheinlichkeit für Kollisionen ist selbst dann groß, wenn M mit 'Sicherheitsabstand' gewählt ist, d. h., wenn der **Belegungsfaktor** (*load factor*) $\alpha = \frac{N}{M}$ klein ist:

Bei einer Hashtabelle der Größe $M = 10.000$ tritt im Durchschnitt eine Kollision bei dem 125. Schlüssel auf. Allgemein gilt dies bei $\sim \sqrt{\pi M/2}$ vielen Schlüsseln. Dies verdeutlich die Notwendigkeit einer guten Kollisionsauflösung, selbst wenn man bereit ist, viel Speicherplatz zu investieren.
Im Folgenden werden einige Hashfunktionen vorgestellt. Alle haben das Ziel, die Hashadressen möglichst *gleichmäßig* zu verteilen, auch wenn die Hashcodes es nicht sind, d. h. *Cluster* in den Hashcodes sollen *aufgelöst* werden.



Um den großen Zahlenbereich $\{0, \ldots, |\boldsymbol{U}| - 1\}$ auf einen kleinen Zahlenbereich $\{0, \ldots, M - 1\}$ abzubilden, gibt es zwei einfache Methoden:

- Multiplikation mit dem Faktor $M/|\boldsymbol{U}|$ und Abrunden
- Rest bei der Division durch M bestimmen.

16.1.2 Hashfunktion: Multiplikationsmethode

Betrachten wir zunächst den einfachen Versuch einer multiplikativen Hashfunktion

$$h : k \mapsto \lfloor k \frac{M}{|\boldsymbol{U}|} \rfloor$$

Diese Funktion bildet zwar die Bereiche korrekt ab, erfüllt aber nicht die gewünschte Eigenschaft der Clusterauflösung: Aufeinander folgende Schlüssel bzw. Hashcodes werden auf dieselbe Hashadresse abgebildet und verursachen

Kollisionen, siehe Abbildung 16.2. Ein weiterer Nachteil ist, dass $|\boldsymbol{U}|$ bekannt sein muss.

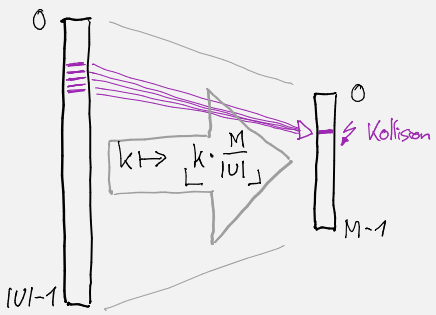


Abbildung 16.2 Naive Multiplikation ergibt eine schlechte Hashfunktion: Schlüssel, die nah beieinander liegen, werden häufig auf dieselbe Hashadresse abgebildet und verursachen dadurch Kollisionen.

Eine geeignete Hashfunktion mit Multiplikationsmethode ist durch den folgenden Satz motiviert:

Satz von Vera Turán Sós (*three-gap theorem*)

Sei γ eine irrationale Zahl. Wir betrachten die Zahlenfolge $c(n) = n\gamma - \lfloor n\gamma \rfloor \in [0,1]$. Die von den ersten n Zahlen gebildeten n Intervalle (bei Betrachtung des Intervalls $[0,1]$ als Ring, wobei 0 und 1 zusammengeklebt werden) haben höchstens drei unterschiedliche Längen. Der nächste Punkt c_{n+1} fällt immer in ein Intervall der größten Länge.

Definition Multiplikationsmethode für Hashfunktionen

Bei der Multiplikationsmethode wird der Schlüssel k zunächst mit einer irrationalen Zahl multipliziert. Von dem Ergebnis wird nur der Nachkommateil genommen, mit M multipliziert und abgerundet.
Laut [Knuth, 1973] führt der goldene Schnitt $\phi = (\sqrt{5}-1)/2$ bei der Wahl der irrationalen Zahl zu einer besonders gleichmäßigen Verteilung:

$$h(k) = \lfloor M(k\phi - \lfloor k\phi \rfloor) \rfloor \qquad \text{(Fibonacci-Hash)}$$

Für Details sowie weitere Tricks bei der Implementation siehe [Knuth, 1973].

16.1.3 Hashfunktion: Divisions-Rest-Methode

Eine einfache Methode, um beliebige ganze Zahlen auf Zahlen $< M$ (Hashadressen) abzubilden, ist es, den *Rest bei der Division durch M* zu berechnen. Entsprechend können Hashfunktionen definiert werden:

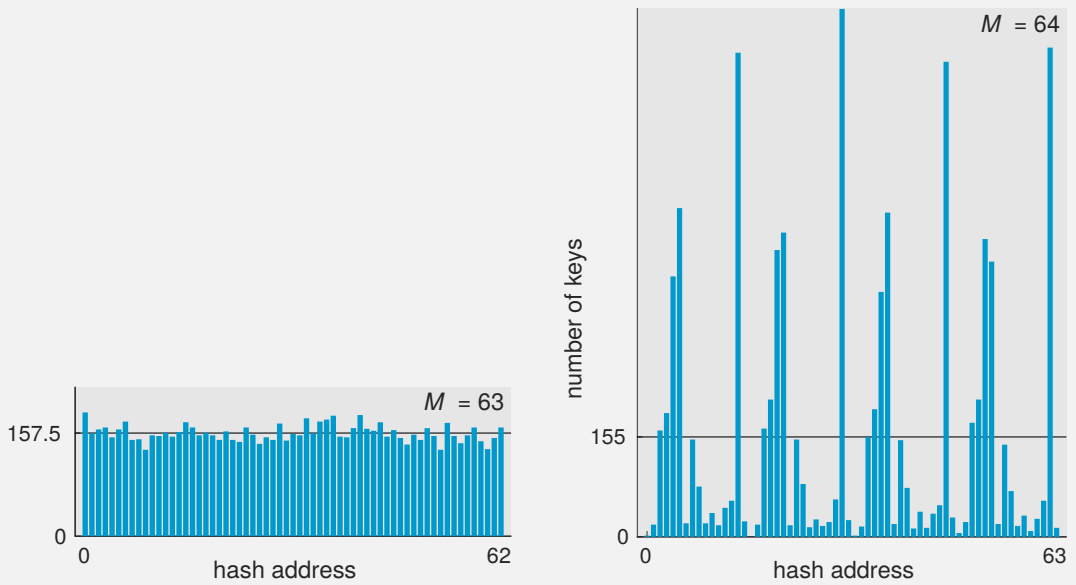
Definition Divisions-Rest-Methode für Hashfunktionen

Bei der Divisions-Rest-Methode wird eine Hashfunktion für eine Hashtabelle der Größe M durch den Rest bei der Division durch M definiert:

$$h : \mathbb{N}_0 \to \{0, \ldots, M-1\}; \quad k \mapsto k \bmod M$$

Bei der Divisions-Rest-Methode ist die genaue Wahl von M wichtig, nicht nur die Größenordnung: Wenn M z. B. eine Zweierpotenz 2^i ist, werden nur die niedrigwertigsten Bits des Schlüssels berücksichtigt. Dies kann bei einer ungleichmäßigen Schlüsselverteilung zu vermehrten Kollisionen führen.

Gute Werte für M sind *Primzahlen*, besonders solche, die *nicht nah an einer Zweierpotenz* liegen. Speziellere Empfehlungen können in [Knuth, 1973] nachgelesen werden. In der empirischen Vergleichsstudie vieler Hashfunktionen in [Lum et al., 1971] lieferte die Divisions-Rest-Methode bei guter Wahl von M durchschnittlich die *besten* Ergebnisse.
Zur Illustration der Divisions-Rest-Methode nehmen wir absichtlich *ungleichmäßig* verteilte Schlüssel, da dann die Hashfunktion besonders gefordert ist. In `hashCode()` von `String` (siehe Abschnitt 16.4) werden die vier niedrigwertigsten Bits durch die entsprechenden Bits des letzten Buchstaben ersetzt. Das Resultat für die Hashfunktionen mit Divisions-Rest-Methode für $M = 63$ und $M = 64$ ist in Abbildung 16.3 abgebildet.
Bei der originalen `hashCode()` Funktion von `String` sind die Hashcodes bei normalen Texten meist in allen Bits recht gleichmäßig verteilt, da in der Hashcode Methode das Prinzip der Divisons-Rest-Methode mit einem geeigneten Divisor angewendet wird. In diesem Fall spielt die exakte Wahl von M keine große Rolle. Das Histogramm für $M = 64$ sieht dann sehr ähnlich wie das Histrogramm für $M = 63$ in Abbildung 16.3 (a) aus.



(a) $M = 63$ **(b)** $M = 64$

Abbildung 16.3 **Histogram der resultierenden Hashadressen** aller Wörter des Werkes *Kritik der reinen Vernunft* bei Anwendung der Divisions-Rest-Methode.

16.2 Die Idee des Universellen Hashing

Auch wenn man eine Hashfunktion aus den beiden vorgestellten Ansätzen auswählt, kann es bei einer *ungünstigen Schlüsselverteilung* zu überproportional vielen Kollisionen kommen. Eine fest gewählte Hashfunktion kann nicht für alle Situationen geeignet sein. Ein Ansatz, diesem Dilemma zu entkommen, ist **Universelles Hashing**. Dabei wird für jede neue Anwendung die Hashfunktion *zufällig* aus einer geeigneten Familie von Funktionen ausgewählt. Bis zum Ende der Anwendung wird die Hashfunktion natürlich konstant gelassen.

Durch die Randomisierung kann man im Erwartungswert die kleinstmögliche Anzahl an Kollisionen, unabhängig von der Verteilung der Schlüssel, erzielen. Im Folgenden werden die wichtigsten Eigenschaften des Universellen Hashing zusammengefasst. Mehr Details sind in Abschnitt 16.5 zu finden.

Unabhängig von der Folge von Schlüsseln, die in eine Hashtabelle eingetragen werden, ist die Anzahl der zu erwartenden Kollisionen jeweils gleich dem Füllgrad der Tabelle. Dies entspricht dem Anteil bei einer zufälligen Wahl von Hashadressen, also der bestmöglichen Durchmischung. Bemerkenswert ist hier, dass es für eine *beliebig (gemein) gewählte* Folge von Schlüsseln gilt. Zu beachten ist, dass sich die Aussage auf den *Erwartungswert* bezüglich der universellen

Menge $\mathcal{H}$ bezieht. Für eine einzelne Anwendung, bei der die gewählte Hashfunktion fix bleibt, erhält man *keine Garantie*.
Was ist der praktische Nutzen? Mit universellem Hashing kann *denial-of-service* Angriffen vorgebeugt werden, siehe [Crosby and Wallach, 2003]. Falls ein Service eine feste Hashfunktion verwendet, die den Angreifern bekannt ist, können sie gezielt Schlüssel einfügen, die alle auf dieselbe Hashadresse abgebildet werden. Dadurch wird durch jede Einfügung eine Kollision erzeugt, und somit der Service immer weiter verlangsamt, bis er funktionsunfähig ist.

16.3 Strategien zur Kollisionsauflösung

Nun widmen wir uns der Frage, wie man mit den beim Hashing auftretenden Kollisionen umgehen kann. Ein Schlüssel k, der beim Hashing nicht unter seiner Hashadresse $h(k)$ in die Tabelle eingetragen werden kann, weil dort besetzt ist, wird **Überläufer** genannt. Generell gibt es zwei Arten mit Überläufern umzugehen:

1. **Hashverfahren mit Verkettung** (*separate chaining*): Überläufer werden *außerhalb* der Hashtabelle in einer zusätzlichen Datenstruktur gespeichert.

2. **Hashverfahren mit Sondierung** (Offene Hashverfahren, *open addressing*): Überläufer werden *innerhalb* der Hashtabelle an einem anderen, freien Ort gespeichert.

Achtung bei den englischen Begriffen: Offene Hashverfahren heißen *open addressing* und *closed hashing*, während Hashverfahren mit Verkettung zu *closed addressing* bzw. *open hashing* zählen.

16.3.1 Hashverfahren mit Verkettung (*Separate Chaining*)

Die Hashtabelle wird als Feld von *Referenzen auf verkettete Listen* angelegt. Wenn Kollisionen auftreten, werden die Überläufer an die jeweilige Liste angehängt. In jedem Element der verketteten Liste ist der abgelegte Schlüssel, sowie eine Referenz auf den zugehörigen Wert gespeichert. Und, wie immer bei einer verketteten Liste, gibt es eine Referenz auf den nächsten Listeneintrag, bzw. `null` am Listenende, siehe Abbildung 16.4.
Bei allen Operationen (put, get, delete) wird zunächst die Hashfunktion angewendet, um die Hashadresse festzustellen. Unter der Hashadresse ist in der Hashtabelle die Referenz auf eine verkettete Liste gespeichert. Dann wird die Operation auf die verkettete Liste angewendet. Je länger die Ketten sind, desto länger dauern die Operationen. Eine erfolgreiche Suche kann auch schnell abbrechen. Die Implementation dieses Verfahrens ist sehr einfach:

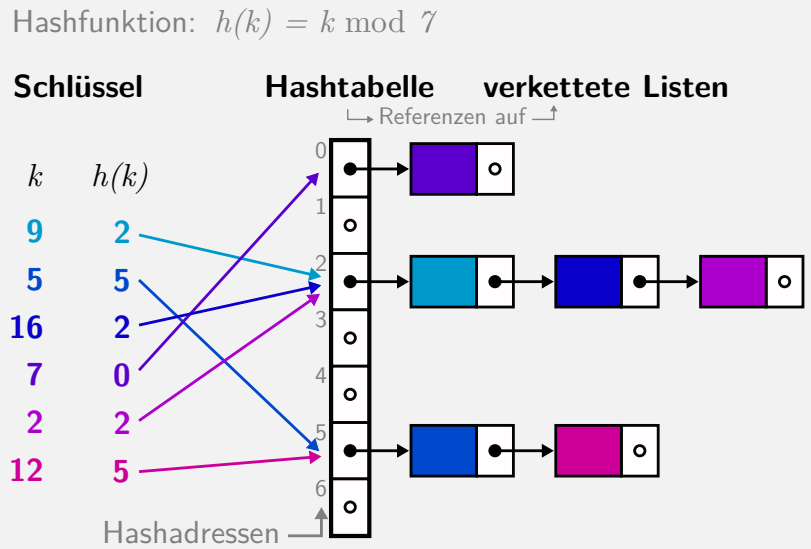


Abbildung 16.4 Hashverfahren mit Verkettung: Zum Einfügen werden Schlüssel an die Liste, die unter der entsprechenden Hashadresse gespeichert ist, angehängt. Nach einer Kollision enthält die Liste mehrere Schlüssel. Zum Suchen eines Schlüssels wird die Liste an der Hashadresse des Schlüssels durchsucht.

Listing 16.1 Implementation von Hashing mit Verkettung

```
public class SeparateChainingHash<K, V> implements SymbolTable<K, V> {
  private int M;
  private LinkedListST<K, V>[] st;

  public SeparateChainingHash(int M) {
    this.M = M;
    st = (LinkedListST<K, V>[]) new LinkedListST[M];
    for (int m=0; m<M; m++) {
      st[m] = (LinkedListST<K, V>) new LinkedListST();
    }
  }

  private int hash(K key) {
    return Math.floorMod(key.hashCode(), M);
  }

  public V get(K key)           { return st[hash(key)].get(key); }
  public void put(K key, V val) { st[hash(key)].put(key, val); }
  public void delete(K key)     { st[hash(key)].delete(key); }
}
```

Bemerkung: Bei der Implementation von `hash()` wird `Math.floorDiv` an Stelle des Modulo Operators `%` verwendet, da `key.hashCode()` negativ sein kann, die Hashadresse, die `hash()` zurückgibt positiv sein muss. Naheliegend wäre es, `Math.abs(key.hashCode() % M)` zu verwenden. Dies führt aber in seltenen Fällen auf Grund einer Besonderheit von `Math.abs()` zu einem Fehler. `Math.abs(MIN_VALUE)` liefert `MIN_VALUE`, also eine negative Zahl zurück. Dies ist ein Problem der Zweierkomplementdarstellung, bei der es eine negative Zahl mehr als positive Zahlen gibt. Aus diesem Grund ist z.B. der Wert von `Math.abs("polygenelubricants".hashCode())` negativ. Durch die Modulo Operation bleibt der Wert negativ, d. h. eine Implementation von `hash()` unter Verwendung von `Math.abs()` könnte eine ungültige Hashadresse liefern, was zu einem Fehler führen würde.
Alternativ zu der oben gewählten Lösung mit `Math.floorMod()` kann man, wie in [Sedgewick and Wayne, 2014, S. 512] vorgeschlagen, das oberste Bit maskieren, um das negative Vorzeichen loszuwerden. Dies ergibt im Gegensatz zu `Math.abs()` immer eine positive Zahl:

```
private int hash(K key) {
    return (key.hashCode() & 0x7fffffff) % M;
  }
```

16.3.2 Effizienz von Hashing mit Verkettung

Die *Effizienz* von Hashing mit Verkettung hängt von der allgemein schwer abzuschätzenden *Verteilung der Hashadressen* ab. Die hängt wiederum von der Verteilung der Schlüssel bzw. Hashcodes und der verwendeten Hashfunktion ab. Die praktische Erfahrung zeigt allerdings, dass bei einer sinnvoll gewählten Hashfunktion die Verteilung der Listenlängen ähnlich zu einer Poisson-Verteilung ist. Dies ist nur eine geringe Abweichung vom Idealfall mit gleichmäßig verteilten Schlüsseln, bei dem sich eine Binomialverteilung ergibt.

Effizienz von Hashing mit Separate Chaining

Bei einer sinnvollen Wahl von Hashcodes und Hashfunktion ist die Anzahl der Schlüsselvergleiche beim Einfügen und (erfolglosen) Suchen in einer Hashtabelle mit M verketteten Listen und N Schlüsseln $\sim N/M$.

Das Hashverfahren *separate chaining* funktioniert auch dann noch, wenn der Belegungsfaktor größer als 1 ist. Es werden bewusst viele Kollisionen in Kauf genommen, um Speicherplatz zu sparen. Nehmen wir an, dass eine Hashtabelle der Größe $M = 1.000$ mit $N = 10.000$ Schlüssel gleichmäßig belegt ist. An jedem Tabelleneintrag hängt also eine verkettete Liste der Länge 10 (Belegungsfaktor

N/M). Dann ist `get()` im Schnitt 5- und im schlimmsten Fall 10-mal langsamer als bei einem Schlüssel-indizierten Array, siehe Seite 268. Andererseits sind sie 1.000 mal schneller als bei einer verketteten Liste (ohne Hashtabelle). Eine typische Wahl für die Größe der Hashtabelle ist $M \sim \frac{N}{5}$. Dann ist also die Anzahl der notwendigen Schlüsselvergleiche im Schnitt 5, siehe Kasten Effizienz von Separate Chaining.

16.3.3 Hashverfahren mit Sondierung (*probing*)

Im Gegensatz zur Speicherung von Schlüsseln *außerhalb* der Hashtabelle in Listen (*separate chaining*) kann eine Kollision auch dadurch aufgelöst werden, dass ein freier Platz *innerhalb* der Hashtabelle gesucht und für den Überläufer verwendet wird. Die Suche nach einen freien Platz zum Einfügen eines Schlüssel-Wert Paares muss nach einer *festen Regel* erfolgen, die später bei der Suche nach einem Schlüssel repliziert wird.

Hashing mit Sondierung (*open addressing*)

Beim **Hashing mit Sondierung** (*open addressing*) verwendet man eine **Sondierungsfunktion**. Bei der **linearen Sondierung** ist dies die Funktion $s(n) = n$. (Weiter unten wird Quadratische Sondierung besprochen.) Falls beim Einfügen eines Schlüssels k eine Kollision auftritt, werden die Plätze entlang der **Sondierungsfolge**

$$(h(k) + s(n)) \bmod M \quad \text{für } n = 0, \ldots, M-1$$

geprüft und der Überläufer in dem ersten freien Platz gespeichert. Beim Suchen eines Schlüssels k wird dieselbe Sondierungsfolge durchlaufen und jeweils der dort gespeicherte Schlüssel mit dem gesuchten Schlüssel k verglichen. Kommt man mit der Sondierungsfolge bei einem freien Platz in der Hashtabelle an, bevor der Schlüssel gefunden wurde, endet die Suche erfolglos.

Das **Suchen** geschieht entlang der Sondierungsfolge. Der Schlüssel, der unter der jeweiligen Adresse in der Hashtabelle gespeichert ist, wird untersucht:

- Ist er gleich dem gesuchten Schlüssel: *erfolgreiche Suche*
- Ist er gleich `null`: *erfolglose Suche*
- Andernfalls geht es in der Sondierungsfolge weiter.

Beim **Einfügen** folgt man gleichermaßen der Sondierungsfolge, bis ein freier Platz (Schlüssel `null`) gefunden wird.

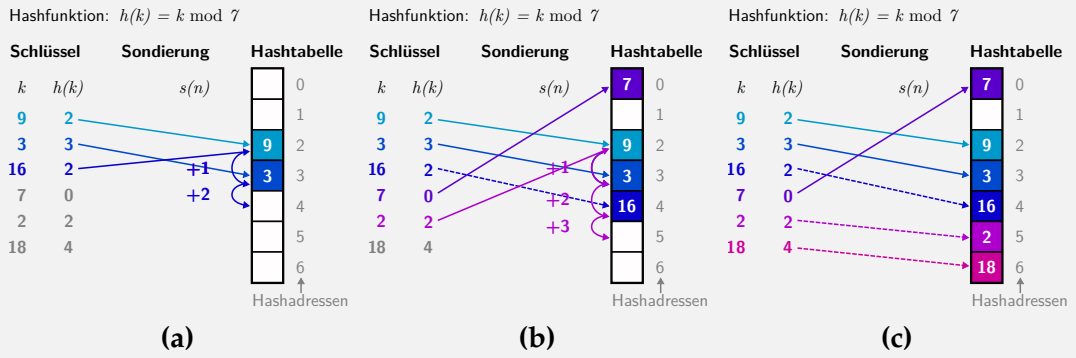


Abbildung 16.5 Hashing mit linearer Sondierung: Die Schlüssel von der linken Spalte werden mit der Hashfunktion $h(k) = k \bmod 7$ auf ihre Hashadresse abgebildet. Kollisionen werden mit linearer Sondierung aufgelöst. **(a)** Nach dem Einfügen der Schlüssel **9** und **3** gibt es bei dem Schlüssel 16 eine Kollision, da er auch unter der Adresse 2 gespeichert werden soll. Mit einem Sondierungsschritt wird Adresse 3 geprüft, die aber auch besetzt ist. Im zweiten Schritt wird 4 als freie Adresse für den Schlüssel **16** gefunden. **(b)** Der Schlüssel **2** soll ebenfalls unter der Hashadresse 2 gespeichert werden. Hier wird erst im vierten Sondierungsschritt 5 als Speicherposition gefunden. **(c)** Auch für Schlüssel **18** muss sondiert werden. Es gab zwar vorher keinen Schlüssel mit 4 als Hashadresse, aber die Speicherposition wurde für den Überläufer **16** benutzt.

Das **Löschen** benötigt eine besondere Aufmerksamkeit. Spielen wir zunächst die Variante durch, dass der zu löschende Schlüssel einfach aus der Hashabelle entfernt wird, also die entsprechende Adresse frei wird. Falls dies eine Adresse ist, welche in der Sondierungsfolge eines später eingefügten Schlüssels k' übersprungen werden musste, dann könnte jener Schlüssel k' nicht mehr wiedergefunden werden. Die Suche entlang der Sondierungsfolge würde an dem frei gewordenen Eintrag abbrechen. (Bei freien Einträgen die Suche *nicht* zu stoppen, ist keine Alternative, da sonst u. U. die komplette Hashtabelle durchsucht wird, was äußerst ineffizient wäre.) Konkret wäre dies der Fall, wenn Schlüssel **16** aus der Hashtabelle in Abbildung 16.5 gelöscht würde. Dann würde bei der Suche nach Schlüssel **18** die Adresse $h(18) = 4$ untersucht. Da sie leer ist, würde als Resultat zurückgegeben, dass Schlüssel **18** nicht vorhanden ist. Es gibt zwei Möglichkeiten für die korrekte Implementierung des **Löschens**:

Verzögertes Löschen (*lazy deletion*): Der Eintrag wird nur als gelöscht markiert und die Methoden werden angepasst:

- Einfügen (*put*) überschreibt einen als gelöscht markierten Eintrag.
- Suche (*get*) wird bei als gelöscht markierten Einträgen fortgesetzt (so als ob der Eintrag noch existieren würde).

Sofortiges Löschen (*eager deletion*): Der Eintrag wird tatsächlich gelöscht (Schlüssel auf `null` setzen). Dabei müssen allerdings die folgenden Einträge in dem Cluster verschoben werden. Dies ist komplizierter, als bei einer verketteten Liste, da sich in dem Cluster sowohl Überläufer von unterschiedlichen Hashadressen befinden können als auch reguläre Einträge. Daher werden alle folgenden Einträge in dem Cluster gelöscht und per `put()` wieder neu in die Hashtabelle eingetragen.

Primäre und sekundäre Häufungen

Bei Kollisionsauflösung durch Sondierung entstehen oft **Häufungen** (*clusters*) in der Belegung der Hashtabellen. Häufungen führen zu langen Sondierungsfolgen (beim Einfügen und bei der Suche) und sind daher schlecht für die Performanz. Sobald eine Häufung entstanden ist, wächst die Wahrscheinlichkeit, dass weitere Kollisionen irgendwo mit dem Cluster passieren und er dadurch immer weiter wächst. Dies nennt man **primäre Häufung** (*primary clustering*).
Sekundäre Häufungen bezeichnen speziell die Häufungen, die dadurch entstehen, dass viele Schlüssel auf dieselbe Hashadresse abgebildet werden und daher derselben Hashsequenz folgen.

16.3.4 Quadratisches Sondieren

Um primäre Häufungen zu vermeiden, kann eine quadratische Sondierungsfunktion benutzt werden. Durch Alternieren des Vorzeichens wird die Streuung noch weiter verbessert.

Quadratisches Sondieren (*quadratic probing*)

Quadratisches Sondieren ein offenes Hashing mit der Sondierungsfunktion

$$s(n) = (-1)^n \cdot \lceil n/2 \rceil^2$$

Wenn die Größe der Hashtabelle M als eine Primzahl der Form $4k + 3$ gewählt wird, dann durchläuft die Sondierungsfolge alle M Hashadressen [Radke, 1970].

Quadratisches Sondieren vermeidet im Gegensatz zum linearen Sondieren die primären Häufungen. Das Problem der sekundären Häufungen bleibt bestehen, da die Sondierungsfolge für alle Schlüssel mit gleichem Hashcode dieselbe ist. Sekundäre Häufungen lassen sich nur dann vermeiden, wenn die Sondierungsfolge vom Schlüssel selbst abhängt und nicht nur von seinem Hashcode. Genau dies wird beim Double Hashing gemacht.

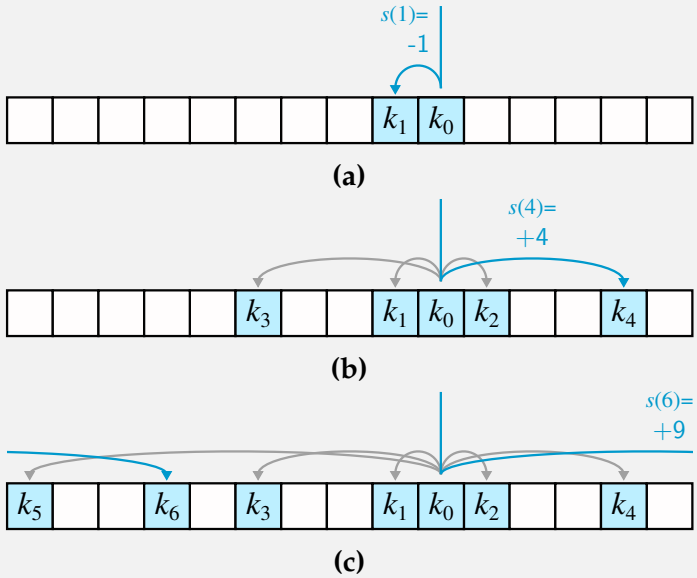


Abbildung 16.6 Quadratische Sondierungsfolge: Die Hashtabelle hat die Größe $M = 15$ und es werden die Schlüssel $k_n = 9 + 15n$ in die Tabelle eingefügt. Sie haben bei Hashfunktion $h(k) = k \bmod 15$ alle die Adresse 9. Die Kollisionen werden mit der quadratischen Sondierungsfunktion aufgelöst.

Double Hashing

Als ein Verfahren, das sekundäre Häufungen durch eine Schlüssel-abhängige Sondierungsfunktion vermeidet, betrachten wird das **Double Hashing**. Beim Double Hashing hängt die Sondierungsfunktion von dem Schlüssel k ab. Dazu wird eine zweite, von der ersten verschiedene Hashfunktion $h'(k)$ verwendet:

$$s(n,k) = n \cdot h'(k)$$

Es sollte $h'(k) \neq 0$ für alle k sein. Wenn M eine Primzahl ist, dann durchläuft die Sondierungsfolge alle M Hashadressen. Um sekundäre Häufungen möglichst effektiv zu vermeiden, sollte $h'(k)$ von $h(k)$ unabhängig sein. Genauer: Für alle Schlüssel k und k' sollte das Ereignis, dass die Schlüssel eine Kollision bei h verursachen ($h(k) = h(k')$) unabhängig von dem Ereignis sein, dass sie eine Kollision bei h' verursachen ($h'(k) = h'(k')$), d.h. es sollte gelten:

$$P(\{(k,k') \mid h(k) = h(k') \;\&\; h'(k) = h'(k')\}) = \\ P(\{(k,k') \mid h(k) = h(k')\}) \cdot P(\{(k,k') \mid h'(k) = h'(k')\})$$

Diese Bedingung wird von $h(k) = k \bmod M$ und $h'(k) = 1 + k \bmod (M-2)$ erfüllt. Das Löschen von Schlüsseln beim Double Hashing ist durch die Verwendung von zwei Hashfunktionen etwas komplizierter.

Weitere, noch effizientere, aber auch komplexere Verfahren sind uniformes und randomisiertes Sondieren. Diese Verfahren, sowie weitere Ansätze für Hashing werden in [Ottmann and Widmayer, 2011, S. 208ff] beschrieben.

16.3.5 Effizienz von Hashverfahren mit Sondierung

Die Effizienz der offenen Hashverfahren nimmt sehr stark ab, wenn sich der Belegungsfaktor 1 nähert. Daher versucht man, den Belegungsfaktor nicht weit über 0.5 steigen zu lassen. Zur Not wird die Hashtabelle vergrößert. Dies erfordert eine Änderung der Hashfunktion und ein Rehashing, ist also sehr rechenintensiv. Bei einem niedrigen Belegungsfaktor sind die offenen Hashverfahren wenig speichereffizient.
Die Effizenz hängt stark von der Verteilung der Schlüssel ab. Unter der Annahme einer gleichmäßigen Verteilung, die durch eine geeignete Wahl der Hashfunktion halbwegs erreicht wird, gelten die folgenden Sätze ([Knuth, 1973]).

Effizienz von Hashing mit linearer Sondierung

Die durchschnittliche Anzahl von Sondierungen in einer Hashtabelle der Größe M mit N Schlüsseln und $\alpha = N/M$ ist

$$\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad \text{für erfolgreiche Suche und}$$

$$\sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right) \qquad \text{für erfolglose Suche und Einfügungen.}$$

Bei einem Belegungsfaktor von $\alpha = 1/2$, der als Tipp für die Praxis die obere Grenze darstellt, ergeben sich die ungefähren Werte $3/2$ und $5/2$.

Effizienz von Hashing mit quadratischer Sondierung

Die durchschnittliche Anzahl von Sondierungen in einer Hashtabelle der Größe M mit N Schlüsseln und $\alpha = N/M$ ist

$$\sim 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2} \qquad \text{für erfolgreiche Suche und}$$

$$\sim \frac{1}{1-\alpha} - \alpha + \ln\left(\frac{1}{1-\alpha}\right) \qquad \text{für erfolglose Suche und Einfügungen.}$$

Bei einem Belegungsfaktor von $\alpha = 1/2$ ergeben sich die ungefähren Werte 1.44 und 2.19. Für höhere Werte von α wird der Unterschied zur linearen Sondierung immer größer.

16.4 Hashing mit Java Objekten

Bisher haben wir angenommen, dass die Schlüssel natürliche Zahlen sind. In der Praxis ist dies meist nicht der Fall. Es sollen beliebige Java Objekte als Schlüssel verwendet werden können. Dazu hat Java das Konzept der **Hashcodes** eingebaut. Die Java Klasse `Object`, von der alle anderen Klassen abgeleitet sind, deklariert die Methode **`hashCode()`**, die zu jedem Objekt einen `int` Wert zurückgibt. Für alle Klassen in den Java Bibliotheken ist die Methode `hashCode` sinnvoll implementiert. Für jede eigene Klasse muss eine eigene Methode implementiert werden, falls sie die Hashing Funktionalität unterstützen soll. Wenn die `hashCode()` Methode nicht überschrieben wird, erbt die Klasse die Standardimplementation, die die Speicheradresse des Objektes zurückgibt, also nicht sinnvoll ist.
In Java 8 wird z. B. die Methode `hashCode()` für `String` Objekte im Prinzip folgendermaßen implementiert, (allerdings als Objektmethode):

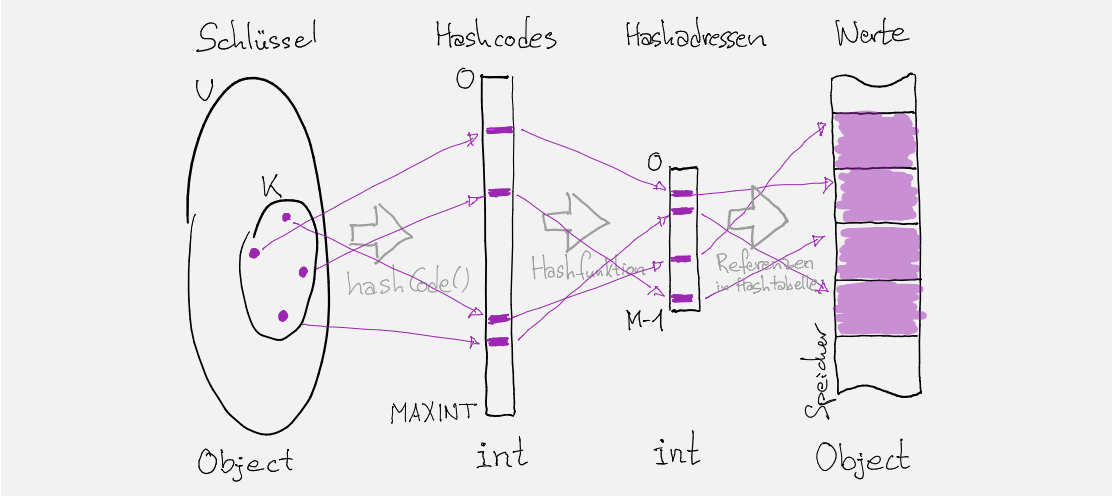
```
public static int hashCode(String str) {
  int hash = 0;
  for (int i = 0; i < str.length(); i++)
    hash = 31 * hash + str.charAt(i);
  return hash;
}
```

Die tatsächliche Implementierung speichert berechnete Hashcodes in einer Variablen, um sie bei erneutem Aufruf nicht neu berechnen zu müssen.
Für Datentypen aus den offiziellen Java Bibliotheken sind jeweils `hashCode()` Methoden implementiert. Für primitive Datentypen kann über den entsprechenden Wrappertyp eine vorhandene `hashCode()` Methode verwendet werden. Für eigene Datentypen läuft man über alle relevanten Datenfelder und verbindet deren Hashcodes wie im `String` Beispiel. Für die Klasse `FlowEdge`, die bei Flussgraphen zur Speicherung einer Kante mit ihrer Kapazität und dem aktuellen Fluss genutzt wird, könnte `hashCode()` folgendermaßen definiert werden:

```
public class FlowEdge {
  private int v, w;
  private double flow, capacity;
  ...
  public int hashCode() {
    int hash = v;
    hash = 17 * hash + w;
    hash = 17 * hash + ((Double) flow).hashCode();
    hash = 17 * hash + ((Double) capacity).hashCode();
    return hash;
  }
}
```

Bei der Definition von Hashfunktionen haben wir als Definitionsbereich die nicht-negativen ganzen Zahl $\mathbb{N}_0$ vorausgesetzt. Damit sie auf Hashcodes, die vom Typ `int` sind, angewendet werden können, sollte also das Vorzeichenbit gelöscht werden, siehe die Diskussion über `Math.abs()`.

Die Abbildung der Schlüssel auf Hashadressen wird in zwei Schritte unterteilt. Da die Schlüssel beliebige Objekte sein können, werden diese zunächst in ganze Zahlen vom Typ `int`, genannt *Hashcodes*, umgewandelt. Die Hashcodes werden dann von einer *Hashfunktion* h auf einen kleinen Wertebereich $\{0, \ldots, M-1\}$ abgebildet. Diese Werte dienen als Index in einer Hashtabelle und werden daher *Hashadressen* genannt.



16.4.1 Hashtabellen in Java Collections

Die Java Collections stellen die Klassen `HashSet` und `HashMap` zur Verfügung. Zur Benutzung mit eigenen Klassen müssen die Methoden `hashCode()` und `equals()` sinnvoll überschrieben werden: Die Ausgabe von `hashCode()` muss **konsistent** sein, d. h. die Hashcodes müssen verträglich mit `equals()` sein:

- Gilt `a.equals(b)`, dann müssen `a.hashCode()` und `b.hashCode()` die gleichen Werte liefern.

Die Berechnung von `hashCode()` sollte möglichst *effizient* sein. Bei aufwändigeren Berechnungen sollte der Hashcode in einer Objektvariablen gespeichert werden, um Mehrfachberechnungen zu vermeiden. Die Schlüssel sollten durch die Hashcodes möglichst *gleichmäßig verteilt* werden. Der letzte Punkt kann z. B. empirisch überprüft werden, wie in Abbildung 16.3.

16.5 Universelles Hashing

In Abschnitt 16.2 wurde Universelles Hashing motiviert und die wesentlichen Eigenschaften zusammengefasst. Hier folgt eine detailliertere Darstellung.
Wir betrachten ein festes Universum $\boldsymbol{U}$ und ein festes M (Größe der Hashtabelle). Sei $\mathcal{H}$ eine Familie von Hashfunktionen. $\mathcal{H}$ heißt **universell**, falls für zwei beliebige unterschiedliche Schlüssel $k, l \in \boldsymbol{U}$ gilt:

$$\frac{|\{h \in \mathcal{H} \mid h(k) = h(l)\}|}{|\mathcal{H}|} \leq \frac{1}{M}$$

Dies bedeutet, dass im Durchschnitt über alle Hashfunktionen aus $\mathcal{H}$ die Wahrscheinlichkeit einer Kollision nicht größer ist, als wenn die zwei Hashadressen zufällig aus $\{0, \ldots, M-1\}$ ausgewählt werden. Im Schnitt über $\mathcal{H}$ betrachtet, kann es also keine systematischen Häufungen geben, die bei einzelnen Hashfunktionen durch spezielle Verteilungen der Schlüssel verursacht werden können:

Universelles Hashing - Eigenschaft

Eigenschaft einer universellen Menge von Hashfunktionen

Sei $\mathcal{H}$ eine universelle Menge von Hashfunktionen. Wir nehmen an, dass in einer Hashtabelle bereits N Schlüssel gespeichert sind. Dann ist für einen gegeben Schlüssel k der Anteil der Hashfunktionen $h \in \mathcal{H}$, für den es eine Kollision gibt N/M.

Beweis siehe [Ottmann and Widmayer, 2011, S. 196].

Der folgende Satz gibt eine konkrete universelle Familie von Hashfunktionen an, die für universelles Hashing benutzt werden kann.

Universelle Familie von Hashfunktionen

Zu einem gegebenen Universum $\boldsymbol{U}$ sei $p \geq |\boldsymbol{U}|$ eine Primzahl. Für ganze Zahlen a, b definieren wir

$$h_{a,b} : \boldsymbol{U} \to \{0, \ldots, M-1\}; \quad k \mapsto ((ak + b) \bmod p) \bmod M$$

Dann ist die Menge $\mathcal{H} = \{h_{a,b} \mid 1 \leq a < p \text{ und } 0 \leq b < p\}$ ist eine universelle Familie von Hashfunktionen.

Beweis siehe [Ottmann and Widmayer, 2011, S. 197].

16.6 Das Wichtigste in Kürze

Hashtabellen implementieren die Funktionalität von Symboltabellen für Schlüssel-Wert Paare (einfügen, suchen und löschen über den Schlüssel). Im Gegensatz zu Speicher-effizienten, aber langsamen Listen und schnellen, aber Speicher-intensiven Arrays bilden Hashtabellen einen guten Mittelweg. Die wesentlichen Elemente von Hashverfahren sind:

1. **Hashfunktion:** Sie sollte schnell zu berechnen sein, und die Schlüssel möglichst gleichmäßig auf die Hashadressen verteilen. Hier wird meist die Divisions-Rest-Methode verwendet, wobei als Tabellengröße M eine Primzahl verwendet wird.
2. **Zur Auflösung von Kollisionen** gibt es Verfahren mit Verkettung und Verfahren mit Sondierung.

Vergleich der Verfahren zur Kollisionsauflösung

Hashing mit Verkettung

- Nahezu konstante Laufzeit unter guten Bedingungen
- Laufzeit steigt langsam mit Füllgrad α.
- Overhead im Speicherbedarf durch externe Listen
- Weniger sensibel gegenüber ungünstigen Hashfunktionen

Hashing mit Sondierung

- Nahezu konstante Laufzeit unter guten Bedingungen
- Laufzeit steigt stark ab $\alpha > 0.5$.
- Speichereffizient bei guter Wahl der Tabellengröße
- Aufwändigere Implementation von *eager delete*

Quellenverweise

Die Darstellung der Hashtabellen ist an [Ottmann and Widmayer, 2011, S. 193ff] angelehnt. Bei den Hashfunktionen sind auch einige Aspekte [Cormen et al., 2013, S. 265ff] entnommen. Die Darstellung der Effizenz von Hashing mit Verkettung und mit Sondierung stammt aus [Sedgewick and Wayne, 2014].

ANHANG A

Appendix

A.1 Branch-and-Bound Implementierung für das Rucksackproblem

Listing A.1 Implementation einer Klasse zur Speicherung der Teillösungen

```
public class PartialSolution {
  public Knapsack knapsack;
  public int level;

  PartialSolution(Knapsack knapsack, int level) {
    this.knapsack = knapsack;
    this.level = level;
  }
}
```

Listing A.2 *Branch-and-Bound* Implementation mit einem `Stack` (dfs)

```
  public Knapsack pack() {
    Knapsack initKnapsack = initialSolution();
    double maxValue = initKnapsack.value();
    Knapsack optKnapsack = initKnapsack;

    Stack<PartialSolution> stack = new Stack<>();
    stack.push(new PartialSolution(new Knapsack(capacity), 0));
    while (!stack.isEmpty()) {
      PartialSolution psol = stack.pop();
      if (psol.knapsack.value() > maxValue) {
        maxValue = psol.knapsack.value();
        optKnapsack = psol.knapsack;
      }
      if (psol.level < items.length) {
        if (bound(psol.knapsack, psol.level+1) > maxValue)
          stack.push(new PartialSolution(psol.knapsack, psol.level+1));
        Item newItem = items[psol.level];
        if (newItem.weight <= psol.knapsack.residualCapacity) {
          Knapsack knapsack = new Knapsack(psol.knapsack);
          knapsack.addItem(newItem);
          if (bound(knapsack, psol.level+1) > maxValue)
            stack.push(new PartialSolution(knapsack, psol.level+1));
        }
      }
    }
    return optKnapsack;
  }
```

Listing A.3 *Branch-and-Bound* Implementation mit einer `Queue` (bfs)

```
public Knapsack pack() {
  Knapsack initKnapsack = initialSolution();
  double maxValue = initKnapsack.value();
  Knapsack optKnapsack = initKnapsack;

  Queue<PartialSolution> queue = new LinkedList<>();
  queue.add(new PartialSolution(new Knapsack(capacity), 0));
  while (!queue.isEmpty()) {
    PartialSolution psol = queue.poll();
    if (psol.knapsack.value() > maxValue) {
      maxValue = psol.knapsack.value();
      optKnapsack = psol.knapsack;
    }
    if (psol.level < items.length) {
      if (bound(psol.knapsack, psol.level+1) > maxValue)
        queue.add(new PartialSolution(psol.knapsack, psol.level+1));
      Item newItem = items[psol.level];
      if (newItem.weight <= psol.knapsack.residualCapacity) {
        Knapsack knapsack = new Knapsack(psol.knapsack);
        knapsack.addItem(newItem);
        if (bound(knapsack, psol.level+1) > maxValue)
          queue.add(new PartialSolution(knapsack, psol.level+1));
      }
    }
  }
  return optKnapsack;
}
```

Listing A.4 Implementation der Klasse Teillösung mit *Bound*

```
public class PSolBound implements Comparable<PSolBound> {
  public Knapsack knapsack;
  public int level;
  private double bound;

  public PSolBound(Knapsack knapsack, Item[] items, int level) {
    this.knapsack = knapsack;
    this.bound = bound(knapsack, items, level);
    this.level = level;
  }

  public double bound(Knapsack knapsack, Item[] items, int startIndex) {
     // ... as before as method of BranchAndBoundKnapsack
  }
```

```
  @Override
  public int compareTo(PSolBound that) {
    return - Double.compare(this.bound, that.bound);
  }
}
```

Listing A.5 *Branch-and-Bound* Implementation mit einer `PriorityQueue`

```
public Knapsack pack() {
  PriorityQueue<PSolBound> queue = new PriorityQueue<>();
  queue.add(new PSolBound(new Knapsack(capacity), items, 0));
  while (!queue.isEmpty()) {
    PSolBound psol = queue.poll();
    if (psol.level == items.length) {
      return psol.knapsack;
    } else {
      queue.add(new PSolBound(psol.knapsack, items, psol.level+1));
      Item newItem = items[psol.level];
      if (newItem.weight <= psol.knapsack.residualCapacity) {
        Knapsack knapsack = new Knapsack(psol.knapsack);
        knapsack.addItem(newItem);
        queue.add(new PSolBound(knapsack, items, psol.level+1));
      }
    }
  }
  return null;
}
```

A.2 Implementierungen von Symboltabellen

In dem Abschnitt über Hashtabellen wurden in der Einleitung die Implementation einer Symboltabelle über ein Array bzw. über eine verkettete Liste als naheliegende Ansätze erwähnt. Hier sind entsprechende Java Implementation abgedruckt. Die Vor- und Nachteile dieser Varianten und der Vorteil von Hashtabellen wird in dem entsprechenden Abschnitt diskutiert.

Listing A.6 Java Implementierung einer Symboltabelle mit einem Array

```
public class ArrayST<V> implements SymbolTable<Integer, V> {
    private V[] valuearray;

    public ArrayST(Integer maxKey) {       // Konstruktor: Schlüssel 0 ... maxKey-1
        valuearray= (V[]) new Object[maxKey];
    }

    public void put(Integer key, V value) { valuearray[key]= value; }
    public void delete(Integer key)       { valuearray[key]= null; }
    public V get(Integer key)             { return valuearray[key]; }
}
```

Listing A.7 Java Implementierung einer Symboltabelle mit einer verketteten Liste

```
public class LinkedListST<K,V> implements SymbolTable<K,V> {
  private Node head;

  private class Node {
    K key;
    V val;
    Node next;

    Node(K key, V val, Node next) {
      this.key = key;
      this.val = val;
      this.next = next;
    }
  }

  public V get(K key) {
    for (Node node = head ; node != null ; node = node.next)
      if (node.key == key)
        return node.val;
    return null;
  }
```

```
  public void put(K key, V val) {
    for (Node node = head ; node != null ; node = node.next)
      if (node.key == key) {
        node.val = val;               // erfolgreiche Suche: überschreiben
        return;
      }
    head = new Node(key, val, head);  // erfolglose Suche: hinzufügen
  }

  //public void deleteLazy(K key)  { put(key, null); }

  public void delete(K key) {
    Node prev = null;
    Node node = head;
    while (node != null && node.key != key) {
      prev = node;
      node = node.next;
    }
    if (node != null) {                          // erfolgreiche Suche
      if (prev == null) head = node.next;        //
ersten Knoten löschen
      else              prev.next = node.next;   //
anderen Knoten löschen
    }
  }
}
```

Die auskommentierte Methode `deleteLazy()` implementiert ein verzögertes Löschen (*lazy deletion*). Das Listenelement wird nicht entfernt, sondern der zugehörige Wert wird lediglich auf `null` gesetzt. Für die Funktionalität ist das ausreichend, aber es ist nicht effizient. Daher ist die etwas komplizierte Variante des sofortigen Löschens (*eager deletion*) vorzuziehen.

Abbildungsverzeichnis

2.1 Illustration von Wachstumsordnungen: Fragen 6
2.2 Illustration von Wachstumsordnungen: Antworten 6
2.3 Wachstumsordnungen im Vergleich für unterschiedliche Wertebereiche 7
2.4 Der Einfluss von Offset und konstanten Faktoren 8
2.5 Empirische Laufzeitanalysis von `TwoSum` 10

3.1 Illustration der Union-Find Struktur 39
3.2 Gegenüberstellung der Bäume von lazy und eager Union-Find 43

4.1 Permutation der Zahlen $0, 1, 2$ per Backtracking 48
4.2 Rösselsprung 50
4.3 Ein Sudoku Rätsel. 55
4.4 Ein schwieriges Sudoku Rätsel 56
4.5 Baum der Teillösungen 61
4.6 Struktur des Suchbaums 62
4.7 Reihenfolge des Expandierens im Suchbaum 62

5.1 Greedy Suche im Suchbaum illustriert 65
5.2 Aufgabe für einen Greedy Algorithmus: Intervallauswahl 67
5.3 Gegenbeispiele zur Optimalität von Ansätzen zur Intervallauswahl . 68
5.4 Die Aufgabe der Intervallverteilung 72
5.5 Gegenbeispiele zur Optimalität von Strategien zur Intervallplanung . 73
5.6 Planung mit minimalen Verspätungen 78
5.7 Gegenbeispiele zur Optimalität von Strategien zur Ablaufplanung mit minimalen Verspätungen 78
5.8 Illustration des Beweises von Teilaussage 4 des Existenzbeweises . . 81
5.9 Illustration des Äquivalenzbeweises 82
5.10 Illustration des Optimalitätsbeweises von Greedy-Knapsack 86

6.1 Minimax Suchbaum für *BestDifference* 91
6.2 Motivation der Alpha-Beta-Suche 94
6.3 Illustration zum Alpha-Beta Intervall 96
6.4 Alpha-Beta Suchbaum 97

7.1 Bestimmung der Bounds für das 0/1 Rucksackproblem 105
7.2 Schritt eins beim Branch-and-Bound für das 0/1-Rucksackproblem . 106
7.3 Branch-and-Bound Suchbaum für das 0/1-Rucksackproblem 107
7.4 Vergleich unterschiedlicher Suchreihenfolgen 111

8.1 Rekursionsbaum zur Berechnung einer Fibonaccizahl 116
8.2 Redundanz in der rekursiven Berechnung der Fibonaccizahlen 117
8.3 Top-down Ansatz zur Zwischenspeicherung (*memoization*) 119
8.4 Einsparung durch Zwischenspeicherung bei Fibonacci Rekursion . . 120
8.5 Bester Vorgänger 123
8.6 Matrix der Teillösungen für die Editierdistanz 131

9.1 Aufgaben für Graphenalgorithmen 135
9.2 Grundbegriffe zu Graphen 136
9.3 Ein dünner und ein dichter Graph 139
9.4 Adjazenzmatrix zur Speicherung von Kanten 140
9.5 Adjazenzlisten zur Speicherung von Kanten 141
9.6 Unterschiedliche Suchbäume zu einem Graphen 147
9.7 Ablauf der Tiefensuche 148
9.8 Ablauf der Breitensuche 156
9.9 Suchbäume mit Tiefen- und Breitensuche 158
9.10 Vergleich von Tiefen-und Breitensuche 158
9.11 Diversität im Tiefensuchwald 161
9.12 Beispieldurchlauf DFS mit Zeitstempeln 163
9.13 Topologische Sortierung eines DAG 165
9.14 Kategorien von Kanten eines Digraphen bei Tiefensuche 169
9.15 Beispiele für Kategorisierung der Kanten 170

10.1 Minimaler Spannbaum 175
10.2 Beweis der Schnitteigenschaft 178
10.3 Ablauf von Prims Algorithmus zur Erstellung eines MST 183
10.4 Ablauf von Kruskals Algorithmus zur Erstellung eines MST 186

11.1 Kürzeste Wege (SSSP) können als Baum repräsentiert werden 193
11.2 Relaxation einer Kante 195
11.3 Inhaltsabhängige Bildskalierung 199
11.4 Graphen-basierter Ansatz für inhaltsabhängige Bildskalierung 200
11.5 Auswahl der kreuzenden Kante im Dijkstra Algorithmus 202
11.6 Dijkstra ist bei negativen Gewichten nicht Greedy 206
11.7 Reihenfolge der Kantenbearbeitung bei Dijkstra und Bellman-Ford . 211
11.8 Ablauf von Dijkstra und Bellman-Ford 211
11.9 Ablauf von Dijkstra und Bellman-Ford 212

12.1 Darstellung eines Flussgraphen mit einem Fluss 216
12.2 Kleines Beispiel zur Flussumplanung 218
12.3 Fluss vergrößernde Pfade 219
12.4 Flussgraph mit Fluss und zugehöriger Restgraph 220
12.5 Schnitte durch einen Graphen 222
12.6 Fluss über Schnitte und Flusswert 223
12.7 Illustration zum Maxflow - Mincut Theorem 226
12.8 Flussgraph mit langer Laufzeit bei Ford-Fulkerson 228
12.9 Graph ohne garantierte Terminierung bei Ford-Fulkerson 229

13.1 Dijkstra durch Heuristik verbessern 242
13.2 BEST-FIRST in Aktion 244
13.3 Vergleich der Kantenauswahl zwischen Dijkstra, Best-First und A* . 245
13.4 Unzulässige Heuristik in A* 245
13.5 Inkonsistente Heuristik in A* 246
13.6 A* im Vergleich mit BEST-FIRST und Dijkstra 251

16.1 Speicherung von Schlüsseln in einer Hashtabelle 269
16.2 Naive Multiplikation ergibt schlechte Hashfunktion 272
16.3 Anwendung der Divisions-Rest-Methode mit unterschiedlichen M Werten 274
16.4 Hashverfahren mit Verkettung 276
16.5 Hashing mit linearer Sondierung 279
16.6 Quadratische Sondierungsfolge 281

Listings

2.1 `TwoSum` als Beispielprogramm für eine empirische Laufzeitanalyse 9
2.2 Analytische Laufzeitanalyse von `TwoSum` 11
3.1 Implementation einer Vorrangwarteschlange 30
3.2 Implementation einer indizierten Vorrangwarteschlange 34
3.3 Union-Find Implementation mit quick-find 39
3.4 Union-Find Implementation mit (lazy) quick-union 40
3.5 Implementation von eager Union-Find 41
3.6 Union-Find Implementation mit Pfadkompression 43
4.1 Erzeugung aller Permutationen mit Backtracking 47

4.2 Finden eines geschlossenen Rösselsprungs mit Backtracking . . . 51
4.3 Pseudocode für Tiefen- und Breitensuche in Bäumen 59
4.4 Rekursiver Pseudocode für Tiefensuche in Bäumen 59
5.1 Pseudocode des Greedy Algorithmus für *Interval Scheduling* . . . 70
5.2 Implementation des Greedy Algorithmus für *Interval Scheduling* . 70
5.3 Pseudocode des Greedy-Algorithmus *Frühester-Start* 74
5.4 Implementation des Greedy-Algorithmus Frühester-Start 76
5.5 Greedy-Algorithmus für das teilbare Rucksackproblem 84
6.1 Implementation des Minimax-Verfahrens für das Spiel *BestDifference* 92
6.2 Implementation der Alpha-Beta-Suche für das Spiel *BestDifference* 98
6.3 Negamax Implementation des Spiels *BestDifference* 99
7.1 *Branch-and-Bound* Implementation für das 0/1-Rucksackproblem 108
8.1 Algorithmus zur Bestimmung der Intervallauswahl mit maximalem Gewicht über dynamische Programmierung 124
8.2 Lösung des 0/1-Rucksackproblems mit Dynamischer Programmierung (top-down) 127
8.3 Implementation der Editierdistanz mit Dynamischer Progammierung 132
9.1 Implementation der Klasse `Graph` 142
9.2 Implementation der Klasse `Digraph` 143
9.3 Allgemeine Graphendurchsuchung (Traversierung) 144
9.4 Pseudocode für Tiefensuche (DFS) in Graphen mit Rekursion . . 148
9.5 Tiefensuche: markiert die von `s` erreichbaren Knoten 149
9.6 Pfade mit Tiefensuche finden 151
9.7 Zusammenhangskomponenten mit Tiefensuche identifizieren . . 153
9.8 Zyklen mit Tiefensuche erkennen 154
9.9 Breitensuche in Pseudocode 155
9.10 BFS zum Finden von Pfaden mit den wenigsten Kanten 159
9.11 Java Code zur Vergabe von Zeitstempeln für alle Knoten in Digraphen bei der Tiefensuche 162
9.12 Implementierung der Topologischen Sortierung 167
10.1 Generischer Greedy Ansatz zur Bestimmung eines MST 177
10.2 Implementierung des Prim Algorithmus mit Vorrangwarteschlange 180
10.3 Implementierung des Prim Algorithmus mit indizierter PQ 182
10.4 Algorithmus von Kruskal mit UnionFind 185
11.1 Allgemeiner Ansatz für kürzeste Wege durch Relaxierung 196
11.2 Pseudocode zur SSSP mit Topologischer Sortierung 197
11.3 Pseudocode für den Dijkstra SSSP Algorithmus 201
11.4 Pseudocode für den Dijkstra Algorithmus mit Indizierter PQ . . . 204
11.5 Pseudocode für den allgemeinen Bellman-Ford SSSP Algorithmus 207
11.6 Pseudocode für den Bellman-Ford mit Warteschlange 209

12.1 Pseudocode zur Bestimmung des maximalen Flusses eines Flussgraphen gemäß der Ford-Fulkerson Methode 227
12.2 Pseudocode zur Bestimmung des maximalen Flusses eines Flussgraphen G von Quelle s zur Senke t mit dem Edmonds-Karp Algorithmus 232
12.3 Bestimme den maximalen Fluss eines Flussgraphen G von Quelle s zur Senke t mit *Capacity Scaling* 236
13.1 Pseudocode für den BEST-FIRST Algorithmus 243
13.2 Pseudocode für den A* Algorithmus 248
13.3 Pseudocode für A* im Baum der Teillösungen 252
14.1 Pseudocode eines 2-Approximationsalgorithmus für das metrische TSP 257
16.1 Implementation von Hashing mit Verkettung 276
A.1 Implementation einer Klasse zur Speicherung der Teillösungen . 288
A.2 *Branch-and-Bound* Implementation mit einem `Stack` (dfs) 288
A.3 *Branch-and-Bound* Implementation mit einer `Queue` (bfs) 289
A.4 Implementation der Klasse Teillösung mit *Bound* 289
A.5 *Branch-and-Bound* Implementation mit einer `PriorityQueue` . . . 290
A.6 Java Implementierung einer Symboltabelle mit einem Array . . . 291
A.7 Java Implementierung einer Symboltabelle mit einer verketteten Liste 291

Index

0/1-Rucksackproblem, 83

A* Algorithmus, 244
A-star, 244
A-Stern, 244
adjacent, 135
adjazent, 135
Adjazenzliste, 141
Adjazenzmatrix, 140
all-pairs shortest paths, 192
alpha-beta pruning, 94
Alpha-Beta-Suche, 94
alpha-cutoff, 95
amortisierte Laufzeitanalyse, 12
API
  Digraph, 139
  gerichtete gewichtete Kante, 191
  gewichtete Kante, 174
  Graph, 138
  indizierte Vorrangwarteschlange, 32
  kantengewichteter Digraph, 191
  kantengewichteter Graph, 174
  MaxPQ, *siehe* API, Vorrangwarteschlange
  MinPQ, *siehe* API, Vorrangwarteschlange
  Priority Queue, *siehe* API, Vorrangwarteschlange
  Vorrangwarteschlange, 28
Approximationsschema, 261
  mit vollständig polynomieller Laufzeit, 261
Approximativer Algorithmus, 256
arcs, 136
`ArrayDeque`, 24
`ArrayList`, 24
associative array, 267
Auflistung aller Permutationen
  Implementation, 47
Ausgangsgrad, 137
azyklisch, 137

Backtracking, 46
  Rösselsprung, 50
Backtracking
  allgemeines Prinzip, 54
  Pseudocode, 54
backward cost, 244
Baum, 57, 137
Baum der kürzesten Wege, 192
Baum der Teillösungen, 59
Baumkante, 168
Baumtraversierung, 57
Belegungsfaktor, 271
Bellman-Ford
  Korrektheitsbeweis, 207
  Laufzeit, 208
Bellman-Ford Algorithmus, 207
Best-First Algorithmus, 242
beta-cutoff, 95
binary heaps, 27
Branch-and-Bound, 101
  Initiallösung, 102
Branch-and-bound
  Grundelemente, 103
breadth-first search, 58

Breitensuche, 58
  Laufzeit, 157
  Speicherbedarf, 157

Capacity Scaling, 236
  Korrektheit, 237
  Laufzeit, 238
Capacity Scaling Algorithmus
  Pseudocode, 236
closed addressing, 275
closed hashing, 275
cluster, 280
clustering
  primary, 280
  secondary, 280
connected, 137
connected components, 137
content-aware image resizing, 199
crossing edge, 145
cut, 221
cycle, 137

DAG, 136, 164
DCAL, 142
depth-first search, 58
dictionary, 267
Digraph, 136
`Digraph`, 139
digraph, 136
Digraphen, 160
Dijkstra
  Greedy Algorithmus, 205
  mit negativen Gewichten, 205
Dijkstra Algorithmus, 201
  Korrektheitsbeweis, 202
  Laufzeit, 204
directed acyclic graph, 136, 164
directed graph, 136
directed path, 137
`DirectedEdge`, 191
Divide-and-Conquer, 114
Divisions-Rest-Methode, 273
Double Hashing, 281
Durchlaufen, 144
Durchsuchen, 143
Dynamic Programming, 115
Dynamische Programmierung, 115

eager deletion, 280, 292
Ecken, 135
`Edge`, 174
edge
  crossing, 145
edges, 135
`EdgeWeightedDigraph`, 191
`EdgeWeightedGraph`, 174
edit distance, 129
Editierdistanz, 129
Edmonds-Karp
  Laufzeit, 232
Edmonds-Karp Algorithmus, 232
  Pseudocode, 232
Effizienz
  lineare Sondierung, 282
  quadratische Sondierung, 282
  separate chaining, 277
Eingangsgrad, 137

Farthest Insertion:, 254
Fibonacci Zahl, 116
Fibonacci-Hash, 272
Fibonacci-Heap, 205
flow, 215
Fluss, 215
  kritischer Wert, 217
Fluss über einen Schnitt, 222
Flusserhaltung, 215
Flussgraph, 214
Ford-Fulkerson
  Laufzeit, 230
  Pseudocode, 227
forward cost, 244

gerichteter Pfad, 137
gerichteter Zyklus, 137

gewichteter Digraph, 190
gewichteter Graph, 173
Gieriger Algorithmus, 64
Goldener Schnitt, 272
Grad eines Knotens, 137
Graph, 135
  dichter, 138
  dünner, 138
  gerichteter, 136
  Suchbaum, 145
  Traversierung, 143
`Graph`, 138
Greedy Algorithm, 64
Greedy Algorithms, 64
Greedy Algorithmus
  Dijkstra, 205
Greedy-Algorithmus, 64

Hamiltonpfad, 26
Hamiltonzyklus, 259
Hamiltonzyklus', 26
`hashCode()`, 283
Hashcodes, 283
Hashfunktion, 270
  Divisions-Rest-Methode, 273
  Multiplikationsethode, 272
  perfekte, 270
Hashing
  mit Sondierung, 278
  offenes, 278
  perfektes, 270
Hashmenge, 269
hashset, 269
Hashtabelle, 270
Hashverfahren mit Sondierung, 275
Hashverfahren mit Verkettung, 275
Hauptreihenfolge, 162
Heap Ordnung, 29
Heuristik, 240
  konsistent, 247
  zulässig, 246
Heuristische Algorithmen, 240
Häufung, 280
  primäre, 280
  sekundäre, 280

ieterierter Logarithmus, 44
Implementation
  Auflistung aller Permutationen, 47
  Rösselsprung, 51
Implementierung
  Prim Algorithmus mit IndexPQ, 182
incident, 135
`IndexPQ`
  Laufzeit, 36
Indizierte Vorrangwarteschlangen, 32
Inhaltsabhängige Bildskalierung, 199
Initiallösung, 102
interval scheduling, 67
Intervallauswahl, 67
Inversion, 79
inzident, 135

Kante, 135
  gerichtete, 136
  kreuzende, 145
  reflexiv, 136
Kantengewicht, 173
Kantengewichte, 190
Kapazität eines Schnittes, 221
Kapazitätsbeschränkung, 215
Kapazitätskontroll Algorithmus, 236
Knoten, 135
Kollision, 270
Komplexitätsklasse NP, 25
Komplexitätsklasse P, 24
Konsistente Heuristik, 247
Korrektheit
  Allgemeiner MST Ansatz, 178
  Capacity Scaling, 237
  Prim Algorithmus, 180
Kosten, 173
kreuzende Kante, 145
kreuzende Kanten, 177

Kruskal Algorithmus
  Korrektheit, 185
  Laufzeit, 186
Kruskals Algorithmus
  Pseudocode, 185
Kürzeste Wege
  Bellman-Ford, 207
  Dijkstra, 201
  in gewichteten DAG, 197
  mit topologischer Sortierung, 197
Kürzeste Wege in DAG
  Korrektheitsbeweis, 198

Las-Vegas-Algorithmus, 264
Laufzeit
  Algorithmen für maximalen Fluss, 239
  ArrayDeque, 24
  ArrayList, 24
  BFS, 157
  Breitensuche, 157
  Capacity Scaling, 238
  DFS, 150
  indizierte Vorrangwarteschlange, 36
  Kruskal Algorithmus, 186
  LinkedList, 24
  Prim Algorithmus, 181
  Prim Algorithmus mit IndexPQ, 183
  Queue, 23
  Stack, 23
  Tiefensuche, 150
  Union-Find, 45
  Vorrangwarteschlange, 30, 32
Laufzeitanalyse, 8
  amortisierte, 12
  analytische, 11
  empirische, 9
lazy deletion, 279, 292
Levenshtein-Distanz, 129
linear probing, 278
Lineare Sondierung, 278
`LinkedList`, 24

load factor, 271
lower bound, 101, 102

map, 267
max-flow, 215
Maximal azyklischer Untergraph, 265
Maximum acyclic subgraph, 265
Mehrfachkanten, 137
memoization, 117
memoization, 117
minimal kreuzende Kanten, 177
Minimaler Schnitt, 221
Minimaler Spannbaum, 174
  reverse delete, 184
Monte-Carlo-Algorithmus, 264
`MSTPrim`, 179, 182
Multiplikatiionsmethode, 272

Nearest Insertion:, 253
Nebenreihenfolge, 162
Negamax-Variante, 98
negative Zyklen
  Erkennung, 208
negativer Zyklus, 192
network, 214
Niveaugraph, 233
nodes, 135
NP-hard, 25
NP-schwer, 25
NP-vollständig, 83
NP-Vollständigkeit, 24

open addressing, 275, 278
open hashing, 275
optimale Lösung, 101
Ordnung, 137

parallel, 137
path, 137
perfektes Hashing, 270
Pfad, 137
  einfacher, 137
  gerichteter, 137

Pfadkompression, 43
Pfeile, 136
Pfeilliste
  doppelt verkettet, 142
Planung mit minimalen Verspätungen, 77
postorder, 162
preorder, 162
Prim Algorithmus, 179
  Korrektheit, 180
  Laufzeit, 181
Prim Algorithmus mit IndexPQ
  Implementierung, 182
  Korrektheit, 183
  Laufzeit, 183
primary clustering, 280
primäre Häufung, 280
Priority Queue
  Laufzeit, 28
priority queue, 27
`PriorityQueue`, 32
probing
  probing
    linear, 278
  quadratic, 280

quadratic probing, 280
Quadratisches Sondieren, 280
Quelle, 137, 214

Randomisierter Algorithmus, 264
Randomisierung
  Universelles Hashing, 274
Randomized algorithm, 264
Rechenzeit, 4
Relaxierung, 194
Relaxierungsbedingung, 194
residual graphs, 219
Restgraphen, 219
Restkapazität, 220
reverse delete, 184
reverse postorder, 162, 166
ρ-Approximationsalgorithmus, 256
Rucksackproblem, 83
Rösselsprung, 50
  Implementation, 51
Rückwärtskante, 168
Rückwärtskosten, 244

Schedule to Minimize Lateness, 77
Schnitt, 176, 221
  Kapazität, 221
  minimaler, 221
Schnitteigenschaft, 177
seam carving, 199
secondary clustering, 280
Seitenkante, 168
Sekundäre Häufung, 280
Senke, 137, 214
separate chaining, 275, 277
shortest path, 192
Shortest Paths Tree, 192
single-source shortest paths, 191
`sink()`, 29
Sofortiges Löschen, 280
Sondieren
  randomisiertes, 282
  uniformes, 282
Sondierung
  lineare, 278
  quadratische, 280
Sondierungsfolge, 278
Sondierungsfunktion, 278
Spannbaum, 138, 174
  Eigenschaften, 176
Speicherbedarf, 4
  Breitensuche, 157
  Tiefensuche, 150
SPT, 192
stark verbunden, 137
stark zusammenhängend, 137
strong connected, 137
Substrukturanalyse, 123
Suchbaum, 145

Sudoku, 55
`swim()`, 29
symbol table, 267
Symboltabelle, 267

tabulation, 117
tabulation, 118
teilbare Rucksackproblem, 84
Tiefe einer Intervallmenge, 73
Tiefensuche, 58
  Laufzeit, 150
  rekursive Implementation, 149
  Speicherbedarf, 150
Topologische Sortierung, 164
Traveling Salesman Problem, 26
Travelling Salesman Problem
  approximativer Ansatz, 256
  heuristischer Ansatz, 253
Travelling Salesman Problem, 253
Traversierung, 144

umgekehrte Hauptordnung, 166
Umgekehrte Hauptreihenfolge, 162
Union-Find
  Laufzeiten, 44, 45
  quick-union, 40
`UnionFind`, 39–41, 43
universell, 285
Universelles Hashing, 274
Universum, 269
upper bound, 101, 102

vertices, 135
Verzögertes Löschen, 279
Vorrangwarteschlange, 27
  indizierte, 32
  Laufzeit, 28, 30, 32
Vorwärtskante, 168
Vorwärtskosten, 244

Wachstumsordnung, 4
Wald, 138
Weg, 137
*weighted A**, 262
Wert des Flusses, 215
Wörterbuch, 267

Zulässige Heuristik, 246
Zusammenhangskomponente, 137
  starke, 137
Zusammenhangskomponenten
  identifizieren, 153
zusammenhängend, 137
zyklenfreier Digraph, 136, 164
Zyklus, 137
  Erkennung mit Tiefensuche, 154

Überläufer, 275

Literaturverzeichnis

[Ariannejad and Emami, 2014] Ariannejad, M. and Emami, M. (2014). A new proof of cayley's formula for labeled spanning trees. *Electronic Notes in Discrete Mathematics*, 45:99–102.

[Avidan and Shamir, 2007] Avidan, S. and Shamir, A. (2007). Seam carving for content-aware image resizing. In *ACM SIGGRAPH 2007 papers*, pages 10–es.

[Chazelle, 1997] Chazelle, B. (1997). A faster deterministic algorithm for minimum spanning trees. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 22–31. IEEE.

[Chazelle, 2000] Chazelle, B. (2000). A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM (JACM)*, 47(6):1028–1047.

[Conrad et al., 1994] Conrad, A., Hindrichs, T., Morsy, H., and Wegener, I. (1994). Solution of the knight's hamiltonian path problem on chessboards. *Discrete Applied Mathematics*, 50(2):125–134.

[Cormen et al., 2013] Cormen, T. H., Leiserson, C. E., Rivest, R., and Stein, C. (2013). *Algorithmen-Eine Einführung*. De Gruyter Oldenbourg.

[Crosby and Wallach, 2003] Crosby, S. A. and Wallach, D. S. (2003). Denial of service via algorithmic complexity attacks. In *USENIX Security Symposium*, pages 29–44.

[Dasgupta et al., 2008] Dasgupta, S., Papadimitriou, C. H., and Vazirani, U. V. (2008). *Algorithms*. McGraw-Hill Higher Education New York.

[Dietzfelbinger, 2012] Dietzfelbinger, M. (2012). Effiziente Algorithmen. https://www.tu-ilmenau.de/en/institute-of-theoretical-computer-science/lehre/lehre-ss-2017/aud. TU Ilmenau.

[Dijkstra et al., 1959] Dijkstra, E. W. et al. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271.

[Dinic, 1970] Dinic, E. A. (1970). Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, volume 11, pages 1277–1280.

[Erickson, 2019] Erickson, J. (2019). *Algorithms*. Independently published.

[Fredman and Saks, 1989] Fredman, M. and Saks, M. (1989). The cell probe complexity of dynamic data structures. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 345–354.

[Fredman and Tarjan, 1987] Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615.

[Hopcroft and Tarjan, 1974] Hopcroft, J. and Tarjan, R. (1974). Efficient planarity testing. *Journal of the ACM (JACM)*, 21(4):549–568.

[Karger et al., 1995] Karger, D. R., Klein, P. N., and Tarjan, R. E. (1995). A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM (JACM)*, 42(2):321–328.

[Karpinski et al., 2015] Karpinski, M., Lampis, M., and Schmied, R. (2015). New inapproximability bounds for TSP. *Journal of Computer and System Sciences*, 81(8):1665–1677.

[Kleinberg and Tardos, 2013] Kleinberg, J. and Tardos, E. (2013). *Algorithm Design: Pearson New International Edition*. Pearson Education Limited.

[Knuth, 1973] Knuth, D. E. (1973). The art of computer programming, vol. 3: Searching and sorting. *Reading MA: Addison-Wisley*.

[Kozen, 2012] Kozen, D. C. (2012). *The design and analysis of algorithms*. Springer Science & Business Media.

[Kruskal, 1956] Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50.

[Lum et al., 1971] Lum, V. Y., Yuen, P., and Dodd, M. (1971). Key-to-address transform techniques: a fundamental performance study on large existing formatted files. *Communications of the ACM*, 14(4):228–239.

[Nau et al., 1984] Nau, D. S., Kumar, V., and Kanal, L. (1984). General branch and bound, and its relation to a* and ao*. *Artificial Intelligence*, 23(1):29–58.

[Nešetřil et al., 2001] Nešetřil, J., Milková, E., and Nešetřilová, H. (2001). Otakar borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete mathematics*, 233(1-3):3–36.

[Orlin, 2013] Orlin, J. B. (2013). Max flows in o (nm) time, or better. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 765–774.

[Ottmann and Widmayer, 2011] Ottmann, T. and Widmayer, P. (2011). *Algorithmen und datenstrukturen*. Springer.

[Pettie and Ramachandran, 2002a] Pettie, S. and Ramachandran, V. (2002a). An optimal minimum spanning tree algorithm. *Journal of the ACM (JACM)*, 49(1):16–34.

[Pettie and Ramachandran, 2002b] Pettie, S. and Ramachandran, V. (2002b). A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM Journal on Computing*, 31(6):1879–1895.

[Prim, 1957] Prim, R. C. (1957). Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401.

[Radke, 1970] Radke, C. E. (1970). The use of quadratic residue research. *Communications of the ACM*, 13(2):103–105.

[Röglin, 2011] Röglin, H. (2011). Randomisierte und approximative Algorithmen. *Vorlesungsskript, Universität Bonn, Wintersemester*, 12.

[Röglin, 2012] Röglin, H. (2012). Skript zur Algorithmen und Berechnungskomplexität I. http://www.roeglin.org/teaching/Skripte/AlgoI.pdf. Universität Bonn.

[Schöning, 2001] Schöning, U. (2001). *Algorithmik*. Spektrum Akademischer Verlag.

[Sedgewick and Wayne, 2014] Sedgewick, R. and Wayne, K. (2014). *Algorithmen: Algorithmen und Datenstrukturen (4. aktualisierte Auflage)*. Pearson Studium.

[Skiena, 2012] Skiena, S. (2012). Algorithms lecture #16 (CSE 373/548) at State University of New York, Stony Brook. https://www3.cs.stonybrook.edu/~algorith/video-lectures.

[Skiena, 2008] Skiena, S. S. (2008). *The Algorithm Design Manual*. Springer Science & Business Media, 2 edition.

[Sleator and Tarjan, 1983] Sleator, D. D. and Tarjan, R. E. (1983). A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391.

[Tarjan, 1975] Tarjan, R. E. (1975). Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225.

[Thorup, 2000] Thorup, M. (2000). Near-optimal fully-dynamic graph connectivity. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 343–350.

[von Warnsdorf, 1823] von Warnsdorf, H. (1823). *Des Rösselsprunges einfachste und allgemeinste Lösung*. Verhagen.

[Wayne, 2013] Wayne, K. (2013). Theory of Algorithms (COS 423) at Princeton University. https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures.php.

[Zwick, 1995] Zwick, U. (1995). The smallest networks on which the ford-fulkerson maximum flow procedure may fail to terminate. *Theoretical computer science*, 148(1):165–170.