Algorithmen und Datenstrukturen

Studentisches Skript

Kim Wittenburg
5wittenb@informatik.uni-hamburg.de

Version vom 20. März 2017

Auf Basis der Vorlesung von Prof. Dr. Petra Berenbrink und Prof. Dr. Chris Biemann im Wintersemester 2016/2017

Disclaimer

Diese Aufzeichnungen sind nicht in Absprache mit Prof. Dr. Berenbrink oder Prof. Dr. Chris Biemann entstanden. Ich habe mir zwar Mühe gegeben, kann aber nicht dafür garantieren, dass die Aufzeichnungen vollständig oder korrekt sind.

Inhaltsverzeichnis

1. Asymptotische Notation **8**

Asymptotisch scharfe Schranke . . . 9
Asymptotisch obere Schranke . . . 10
Asymptotisch untere Schranke . . . 11
Asymptotisch echte obere Schranke . . . 12
Asymptotisch echte untere Schranke . . . 13
Mengenbeziehungen . . . 14
Zusammenfassung . . . 14
Grenzen der Asymptotischen Notation . . . 15

2. Lösen von Rekurrenzen **16**

Die Substitutionsmethode . . . 17
Variablensubstitution . . . 21
Die Master Methode . . . 22

3. Sortieralgorithmen **25**

Merge-Sort . . . 25
Quicksort . . . 27
Randomisiertes Quicksort . . . 32
Vergleichsbasierte Sortieralgorithmen . . . 36
Finden des k-t kleinsten Elementes in einer ungeordneten Liste . . . 37

4. Heaps und Heapsort **40**

Arten von Heaps . . . 40
Max-Heapify . . . 41
Build-Max-Heap . . . 43
Heap-Extract-Max . . . 45
Heapsort . . . 45

5. Binäre Suchbäume **47**

Definitionen . . . 47
Inorder-Tree-Walk . . . 48
Tree-Search . . . 50
Tree-Minimum und Tree-Maximum . . . 51
Tree-Successor . . . 52
Tree-Insert . . . 53
Tree-Delete . . . 54

6. Rot-Schwarz-Bäume **57**
Motivation 57
Definition 57
Balanciertheit eines Rot-Schwarz-Baumes 58
Rotationen 60
RB-Tree-Insert 61
Ausblick 67

7. Hashtabellen **68**
Motivation 68
Definition 68
Hashtabellen in Java 70
Operationen auf Hashtabellen 70
Hashfunktionen 74

8. Greedy-Algorithmen **76**
Huffman-Codes 76
Allgemeine Greedy-Probleme 81
Scheduling 82

9. Dynamische Programmierung **85**
Kettenmultiplikation von Matrizen 85
Das Prinzip der dynamischen Programmierung 86
Der Floyd-Warshall-Algorithmus 90

10. Elementare Graphalgorithmen **95**
Die Datenstruktur Graph 95
Adjazenzlisten 95
Adjazenzmatrizen 96
Breadth-First-Search 97
Depth-First-Search 100
Wälder 101

11. Minimale Spannbäume **102**
Grundsätzlicher Algorithmus 102
Sichere Kanten 103
Algorithmus von Kruskal 104
Algorithmus von Prim 106

12. Kürzeste Pfade in Graphen **109**
Problemstellung 109
Kürzeste Pfad-Bäume 110
Der Bellman-Ford-Algorithmus 110
Algorithmus von Dijkstra 116

13. Maximale Flüsse in Netzwerken **120**
Die Ford-Fulkerson-Methode 122
Residuennetzwerke 123
Flussverbessernde Pfade 124
Korrektheit der Ford-Fulkerson-Methode 125
Optimalität der Ford-Fulkerson-Methode (Schnitte) 126
Algorithmus von Edmonds und Karp 129

14. Komplexitätstheorie **132**
Komplexitätsklassen 133
Codierungen für Probleminstanzen 133
Turing-Maschinen 135
Entscheidungs- und Optimierungsprobleme 137
Die Komplexitätsklasse $\mathcal{P}$ 138
Die Komplexitätsklasse $\mathcal{NP}$ 139
$\mathcal{P}$ vs. $\mathcal{NP}$ 140
$\mathcal{NP}$-Vollständigkeit 141
Beispiel für eine Reduktion: Matchings in bipartiten Graphen 142
$\mathcal{NP}$-Vollständigkeit anwenden 143
Das erste $\mathcal{NP}$-vollständige Problem: SAT 144
Weitere $\mathcal{NP}$-vollständige Probleme 148

A. Schleifeninvarianten **152**

B. Grundlagen zur Wahrscheinlichkeitsrechnung **154**
Elementare Kombinatorik 154
Das Urnenmodell 157
Wahrscheinlichkeit 157
Wahrscheinlichkeitsverteilung 158
Diskrete Wahrscheinlichkeitsverteilungen 159
Bedingte Wahrscheinlichkeiten 159
Diskrete Zufallsvariablen 161
Erwartungswert 162
Varianz und Standardabweichung 162

1. Asymptotische Notation

Algorithmen sind ein wichtiger Teil der theoretischen Informatik. Meistens geht es aber nicht darum, nur einen Algorithmus zu finden, sondern auch darum zu zeigen, dass ein bestimmter Algorithmus ein gegebenes Problem besser löst, als ein anderer. Dazu betrachtet man in den meisten Fällen die Laufzeit des Algorithmus.

Allerdings kann man die spezifische Laufzeit eines Allgorithmus nicht im Allgemeinen bestimmen, da diese von Faktoren wie der verwendeten Hardware oder der Programmiersprache abhängen kann. Deshalb betrachten wir die asymptotische Laufzeit. Bei der asymptotischen Laufzeit analysieren wir einen Algorithmus (der meistens als Pseudocode aufgeschrieben ist) darauf, wie sich die Laufzeit im Verhältnis zur Größe der Eingabe verändert und ignorieren dann alle additiven und multiplikativen Konstanten, da diese davon abhängig sind, wie der Algorithmus am Ende ausgeführt wird.

In der Realität sieht dies so aus: Gegeben ist ein Algorithmus (bzw. eine Funktion), deren asymptotische Laufzeit bestimmt werden soll. Dann suchen wir nach bestimmten Kriterien eine andere Funktion, die die gegebene Funktion von oben, von unten oder beides beschränkt.

Die asymptotische Notation wird auch Landau-Notation oder Big-O-Notation genannt.

Zur Notation

Für eine Funktion g werden mit $o(g)$, $\mathcal{O}(g)$, $\Theta(g)$, $\Omega(g)$ und $\omega(g)$ **Mengen von Funktionen** definiert. Aus Gründen der Bequemlichkeit schreibt man meistens allerdings ein $=$ anstelle von $\in$. Wenn wir also sagen wollen, dass die Funktion $f(n)$ in der Menge $\Theta(n^2)$ enthalten ist, sind die folgenden Schreibweisen äquivalent:

$$f(n) \in \Theta(n^2)$$
$$f(n) = \Theta(n^2)$$

Man darf sich hiervon allerdings nicht verwirren lassen: Es handelt sich hier nicht um eine Äquivalenz! $\Theta(n^2) = f(n)$ ist **falsch**. Es ist lediglich als alternatives Symbol zu verstehen.

Asymptotisch scharfe Schranke

Definition 1.1. *Seien $f(n)$ und $g(n)$ Funktionen. Dann ist die asymptotisch scharfe Schranke*

$$\Theta(g) = \{f \mid \exists c_1, c_2 > 0 : \exists n_0 > 0 : \forall n > n_0 : 0 \leq c_1 \cdot |g(n)| \leq |f(n)| \leq c_2 \cdot |g(n)|\}$$

die Menge der Funktionen, die asymptotisch gleich schnell wachsen wie g.

Die Idee bei dieser Definition ist, dass man g mit Konstanten multiplizieren kann und so die Funktion f von oben **und** von unten beschränken kann (also „scharf“ beschränken kann). Der englische Begriff dafür ist *tight bound*. $\Theta(g)$ ist damit die **Menge von Funktionen**, die asymptotisch genauso schnell wachsen wie g. Wenn $f(n)$ und $g(n)$ asymptotisch gleich schnell wachsen, können wir also $f(n) \in \Theta(g(n))$ schreiben. Meistens schreibt man stattdessen aber $f(n) = \Theta(g(n))$.

Beispiele

1. $n^2 = \Theta(n^2)$
2. $n^3 + 20n^2 = \Theta(n^3)$
3. $90n^5 + 10n^3 + n + 20 = \Theta(n^5)$

Erklärung

1. Im ersten Beispiel ist offensichtlich $f(n) = g(n) = n^2$. Hier ist klar, dass f und g gleich schnell wachsen.
2. Im zweiten Beispiel ist n^3 der Term, der für große n das stärkste Gewicht hat. Nach der Definition interessieren uns nur sehr große Werte für n, sodass der Term $20n^2$ nicht so stark ins Gewicht fällt, wie n^3. Das heißt, dass für große n gilt: $n^3 + 20n^2 \approx n^3$. Damit wächst die Funktion asymptotisch genauso schnell wie n^3.
3. Dieses Beispiel funktioniert analog zu Beispiel 2. Es gilt hier für große n: $90n^5 + 10n^3 + n + 20 \approx 90n^5$. Das ergibt, dass $90n^5 + 10n^3 + n + 20 \approx 90n^5 = \Theta(90n^5)$ ist. Da wir aber nach der Definition multiplikative Konstanten ignorieren können, gilt $\Theta(90n^5) = \Theta(n^5)$ (d.h. die beiden Mengen sind äquivalent).

Im Allgemeinen gilt für ein beliebiges Polynom $p(n)$:

$$p(n) = \Theta(n^{\text{grad}(p)})$$

Ausführlicheres Beispiel

Gegeben sei die Funktion $f : \mathbb{N} \to \mathbb{N}$ mit $f(n) = 2n^2$. Zu bestimmen ist die scharfe Schranke.

Wir suchen also $g(n)$, $c_1 > 0$ und $c_2 > 0$, sodass ab einem bestimmten n_0 gilt, dass für alle größeren n $c_1 \cdot g(n) \leq f(n)$ und $c_2 \cdot g(n) \geq f(n)$ ist. Wir wählen also $g(n) = n^2$. Jetzt wollen wir diese Vermutung beweisen:

Wir wählen dafür $c_1 = 1$. Dann ist $c_1 \cdot g(n) = n^2$. Jetzt ist offensichtlich, dass $n^2 \leq 2n^2$ ist, für alle $n > 0$. Wir wählen dann $c_2 = 3$. Dann ist $c_2 \cdot g(n) = 3n^2$. Auch hier ist offensichtlich, dass $3n^2 \geq 2n^2$ für alle $n > 0$ ist. Damit haben wir gezeigt, dass $f(n) = \Theta(n^2)$ ist.

Hätten wir für $g(n)$ ein Polynom mit einem Grad $\neq 2$ gewählt, hätte der Beweis nicht geklappt. Beispielhaft zeigen wir das für $g(n) = n$:

Für $c_1 = 1$ ist $c_1 \cdot g(n) = n$. Da $n < 2n^2$ für alle $n > 0$ ist, haben wir ein passendes c_1 gefunden. Angenommen es gibt nun auch einen gültigen Wert $c_2 = c$, dann wäre $c_2 \cdot g(n) = cn$. Allerdings ist jetzt für alle $n > c$, dass $n^2 > cn$ ist. Also kann es keinen gültigen Wert für c geben. Damit muss die Laufzeitschranke $g(n) = n$ falsch sein.

Asymptotisch obere Schranke

Die asymptotisch obere Schranke ist für die (theoretische) Informatik die wichtigste Schranke. Sie gibt an, wie groß die Laufzeit eines Algorithmus **höchstens** ist. Es kommt häufig vor, dass man für einen Algorithmus nur eine obere Schranke angibt. Gelegentlich wird auch die scharfe Schranke angegeben, um zu betonen, dass der Algorithmus auch nicht wesentlich schneller ist, als die angegebene Schranke. Die obere Schranke ist aber normalerweise wichtiger, da diese garantiert, dass der Algorithmus höchstens eine gewisse Zeit braucht.

Definition 1.2. *Seien wieder $f(n)$ und $g(n)$ Funktionen. Dann ist die asymptotisch obere Schranke*

$$\mathcal{O}(g) = \{f \mid \exists c > 0 : \exists n_0 > 0 : \forall n > n_0 : 0 \leq |f(n)| \leq c \cdot |g(n)|\}$$

die Menge der Funktionen, die asymptotisch langsamer wachsen als g. Alternativ kann die obere Schranke auch folgendermaßen definiert werden:

$$\mathcal{O}(g) = \left\{ f \;\middle|\; \lim_{n \to \infty} \frac{f(n)}{g(n)} < \infty \right\}$$

Da die obere Schranke in der Informatik so wichtig ist, spricht man umgangssprachlich oft statt der oberen Schranke einfach von der Laufzeit des Algorithmus. Mit „Quicksort hat eine Laufzeit von n^2" meint man also, dass Quicksort in $\mathcal{O}(n^2)$ ist.

Aus der Definition von $\mathcal{O}$ ergibt sich, dass $\Theta(g(n)) \subset \mathcal{O}(g(n))$ ist.

Beispiele

1. Alle Beispiele der scharfen Schranke gelten auch für die obere Schranke.
2. $n \in \mathcal{O}(n^2)$
3. $24n^3 + 13n^2 + 5n \in \mathcal{O}(n^3)$

Erklärung

1. Da $\Theta(g(n)) \subset \mathcal{O}(g(n))$ ist, ist jedes Element in $\Theta(g(n))$ auch in $\mathcal{O}(n)$. Daher gelten die Beispiele auch für $\mathcal{O}$
2. $\mathcal{O}$ gibt nur eine obere Schranke an, nicht aber notwendigerweise die beste obere Schranke. Es gilt auch $n = \mathcal{O}(n)$. Allerdings ist $n = \mathcal{O}(n^2)$ auch korrekt.
3. Analog zu Θ können wir multiplikative und additive Konstanten ignorieren. Da uns außerdem nur große Werte von n interessieren, ist $24n^3 + 13n^2 + 5n \in \Theta(n^3)$ und somit auch in $\mathcal{O}(n^3)$.

Asymptotisch untere Schranke

Die untere Schranke ist in der angewandten Informatik normalerweise eher uninteressant, da diese im Prinzip nur aussagt, dass der Algorithmus langsamer als die angegebene Schranke ist. In der theoretischen Informatik ist vor allem die kleinste untere Schranke interessant. Mit dieser kann man angeben, was die **geringste Laufzeit** zum Lösen eines Problems ist. So kann man Aussagen beweisen wie „Man kann eine Liste nicht schneller sortieren als in linearer Zeit“.

Definition 1.3. *Seien wieder $f(n)$ und $g(n)$ Funktionen. Dann ist die asymptotisch untere Schranke*

$$\Omega(g) = \{f \mid \exists c > 0 : \exists n_0 > 0 : \forall n > n_0 : |f(n)| \geq c \cdot |g(n)| \geq 0\}$$

die Menge der Funktionen, die asymptotisch schneller wachsen als g. Alternativ kann die untere Schranke auch folgendermaßen definiert werden:

$$\Omega(g) = \left\{ f \,\middle|\, \lim_{n \to \infty} \frac{f(n)}{g(n)} > 0 \right\}$$

Auch für die untere Schranke gilt $\Theta(g(n)) \subset \Omega(g(n))$. Die Funktionen in $\Theta(g(n))$ können also sowohl eine obere als auch eine untere Schranke sein. Dies passt genau zu der Definition von oben.

Beispiele

1. $5n^2 + 3 = \Omega(1)$
2. $n^{500} = \Omega(n^{300})$
3. $2n = \Omega(3n)$

Erklärung

1. Da Ω eine untere Schranke ist, sollte dieses Beispiel relativ eindeutig sein. Eine Konstante wächst definitiv langsamer (mit n) als $5n^2 + 3$.
2. Analog zu Beispiel 1 ist n^{500} für große n asymptotisch größer als n^{300}.
3. Dieses Beispiel mag zunächst falsch erscheinen. Allerdings sind sowohl $2n$ als auch $3n$ in $\Theta(n)$. Damit wachsen $2n$ und $3n$ asymptotisch gleich und es gilt $2n = \Omega(3n)$.

Asymptotisch echte obere Schranke

Die echte obere Schranke ist sehr ähnlich definiert wie die *normale* obere Schranke. Allerdings enthält sie keine Funktionen, die asymptotisch gleich wachsen wie die angegebene Schranke. Die echte obere Schranke ist für theoretische Betrachtungen interessant, wird aber in der Praxis kaum verwendet.

Definition 1.4. *Seien wieder $f(n)$ und $g(n)$ Funktionen. Dann ist die asymptotisch echt obere Schranke*

$$o(g) = \{f \mid \forall c > 0 : \exists n_0 > 0 : \forall n > n_0 : 0 \leq |f(n)| < c \cdot |g(n)|\}$$

die Menge der Funktionen, die asymptotisch echt langsamer wachsen als g. Alternativ kann die echte obere Schranke auch folgendermaßen definiert werden:

$$o(g) = \left\{ f \,\middle|\, \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \right\}$$

Für die echte obere Schranke gilt $o(g(n)) \cap \Theta(g(n)) = \emptyset$.

Beispiele

1. $5n^2 + 3n = o(n^3)$
2. $n \log n = o(n^2)$
3. $n^2 \neq o(n^2)$

Erklärung

1. Wir können wie immer additive und multiplikative Konstanten weglassen. Außerdem ist für große n der Term $5n^2$ deutlich größer als $3n$. Also ist $5n^2 + 3n = \Theta(n^2)$. Offensichtlich wächst n^3 asymptotisch schneller als n^2. Also ist $5n^2 + 3n = o(n^3)$.
2. Es ist gilt für alle $n > 2$, dass $\log_2 n < n$ ist. Daher ist $n \cdot \log_2 n$ asymptotisch echt kleiner als $n^2 = n \cdot n$. Die Basis des Logarithmus ist effektiv eine multiplikative Konstante, sodass gilt: $n \log n = o(n^2)$.
3. Offenbar gilt $n^2 = \Theta(n^2)$. Da aber die Schnittmenge von Θ und o leer ist, kann n^2 nicht die echte obere Schranke von sich selbst sein.

Asymptotisch echte untere Schranke

Wir definieren die asymptotisch echte untere Schranke analog zur echten oberen Schranke. Auch die echte untere Schranke ist für theoretische Betrachtungen interessant, wird aber in der Praxis kaum verwendet.

Definition 1.5. *Seien wieder $f(n)$ und $g(n)$ Funktionen. Dann ist die asymptotisch echt untere Schranke*

$$\omega(g) = \{f \mid \forall c > 0 : \exists n_0 > 0 : \forall n > n_0 : |f(n)| > c \cdot |g(n)| \geq 0\}$$

die Menge der Funktionen, die asymptotisch echt schneller wachsen als g. Alternativ kann die echte untere Schranke auch folgendermaßen definiert werden:

$$\omega(g) = \left\{ f \,\middle|\, \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \right\}$$

Für die echte untere Schranke gilt $\omega(g(n)) \cap \Theta(g(n)) = \emptyset$.

Beispiele

1. $5n^3 + 2n = \omega(n^2)$
2. $n^2 = \omega(n \log n)$
3. $n^2 \neq \omega(n^2)$

Erklärung

1. Es ist $5n^3 + 2n = \Theta(n^3)$. Offensichtlich wächst n^3 asymptotisch schneller als n^2. Also ist $5n^3 + 2n = \omega(n^2)$.

2. Dieses Beispiel ist analog zu Beispiel 2 bei der asymptotisch echten oberen Schranke.
3. Dieses Beispiel ist analog zu Beispiel 3 bei der asymptotisch echten oberen Schranke.

Mengenbeziehungen

Zwischen den verschiedenen asymptotischen Schranken gelten verschiedene Beziehungen. Zum Beispiel:

$$\begin{aligned}\mathcal{O}(g) \cap \Omega(g) &= \Theta(g)\\ \omega(g) &\subseteq \Omega(g)\\ o(g) &\subseteq O(g)\\ o(g) \cap \omega(g) &= \emptyset\end{aligned}$$

Gerade die erste Mengenbeziehung kann besonders hilfreich sein, wenn es darum geht, eine asymptotisch scharfe Schranke für eine Funktion zu finden. Aus der ersten Äquivalenz folgt nämlich folgende alternative Definition für $\Theta(g)$:

$$\Theta(g) = \left\{ f \,\middle|\, \lim_{n\to\infty} 0 < \frac{f(n)}{g(n)} < \infty \right\}$$

Dies lässt sich häufig einfacher überprüfen, als die obige Definition.

Zusammenfassung

Wir haben jetzt die folgenden Notationen kennen gelernt. Die Spalte Pseudo-Notation soll ein intuitives Gefühl für die Bedeutung der Notation geben. Offiziell ist die Notation falsch.

Symbol	Bedeutung	Pseudo-Notation
$f = o(g)$	f wächst asymptotisch echt langsamer als g.	$f < g$
$f = O(g)$	f wächst asymptotisch langsamer als g.	$f \leq g$
$f = \Theta(g)$	f und g wachsen asymptotisch gleich schnell.	$f = g$
$f = \Omega(g)$	f wächst asymptotisch schneller als g.	$f \geq g$
$f = \omega(g)$	f wächst asymptotisch echt schneller als g.	$f > g$

Bei Wikipedia gibt es einige Beispiele für Algorithmen mit bestimmten Laufzeiten.

Grenzen der Asymptotischen Notation

Die Asymptotische Notation vereinfacht das Vergleichen von Algorithmen ungemein. Man kann quasi sofort sehen, welcher Algorithmus schneller ist. Allerdings klappt dies nicht immer. Eines der bekanntesten Beispiele ist MERGE-SORT und QUICKSORT. Beides sind sehr effiziente Sortieralgorithmen. MERGE-SORT ist in $\Theta(n \log n)$ und Quicksort im *worst-case* in $O(n^2)$. In randomisierter Form liegt die erwartete Laufzeit von Quicksort bei $\Theta(n \log n)$. Scheinbar ist also MERGE-SORT deutlich schneller als Quicksort.

In der Realität wird trotzdem fast immer QUICKSORT eingesetzt, weil es in der echten Welt schneller ist als MERGE-SORT. Das liegt daran, dass die Konstanten, die durch die Notation versteckt werden, bei MERGE-SORT deutlich größer sind als bei QUICKSORT. Außerdem kann man zeigen, dass die Wahrscheinlichkeit für eine n^2-Laufzeit bei QUICKSORT sehr gering ist. Dadurch schlägt der QUICKSORT-Algorithmus MERGE-SORT in der Realität fast immer.

2. Lösen von Rekurrenzen

Viele Algorithmen, die wir uns ansehen werden, sind rekursiv. Das hat vor allem den Grund, dass man bei iterativen Algorithmen oft quasi sofort die Laufzeit ablesen kann (man guckt einfach, wie oft die Schleifen durchlaufen werden). Bei rekursiven Algorithmen ist die Analyse normalerweise nicht so einfach. Viele der Algorithmen, die wir uns in dieser Veranstaltung ansehen werden, können der Klasse der **Divide and Conquer** (*deutsch: Teile und Herrsche*) Algorithmen zugeordnet werden. Diese Algorithmen haben ein sehr typisches Schema:

1. Der Algorithmus erhält ein Problem der Größe n.
2. **Divide**: Das Problem wird in a Teilprobleme unterteilt. Jedes der Teilprobleme hat die Größe n/b.
3. **Conquer**: Die Teilprobleme werden rekursiv gelöst.
4. Der Algorithmus integriert die Lösungen der Teilprobleme zu einer Gesamtlösung.

Dadurch ergeben sich folgende Laufzeiten:

- $D(n)$: Die Zeit, die die Aufteilung in Subprobleme benötigt.
- $C(n)$: Die Zeit, die die Rekombination der Lösungen der Subprobleme benötigt.

Damit kann dann die Zeit, die der Algorithmus zum Lösen eines Problems der Größe n braucht, durch folgende Formel dargestellt werden:

$$T(n) = \begin{cases} \Theta(1) & \text{für } n < c \\ a \cdot T(n/b) + D(n) + C(n) & \text{sonst} \end{cases}$$

Die Konstante c hängt vom konkreten Algorithmus ab. Dieser Fall entspricht dem Rekursionsabbruch und muss für die formale Korrektheit berücksichtigt werden. Dass der Algorithmus in diesem Fall in $\Theta(1)$ ist, lässt sich dadurch erklären, dass in diesem Fall keine rekursiven Aufrufe gemacht werden (sonst wäre es ja nicht der Rekursionsabbruch). Dann muss das Problem auch nicht in Teilprobleme unterteilt und wieder integriert werden. Übrigt bleibt also nur eine konstante Zeit.

Konstante Laufzeit

Eine Laufzeit von 0 ist zwar in der Theorie möglich, kommt aber in der Realität nicht vor. Auch wenn in einem Algorithmus buchstäblich *nichts* passiert, ist dies

„nur“ eine konstante Laufzeit, also $\Theta(1)$.

Zur Notation

Durch die asymptotische Notation ist es häufig möglich, Terme zu vereinfachen, indem man sie durch ein asymptotisches Symbol ersetzt. Da die asymptotischen Symbole eigentlich Mengen sind, ist auch diese Schreibweise technisch gesehen nicht korrekt. Zum Beispiel:

$$T(n) = n^2 + \Theta(1)$$

Dies wird gelesen als „$T(n) = n^2$ plus eine konstante Zeit“. $\Theta(1)$ steht hier also repräsentativ für **irgendeine Funktion**, die selbst in $\Theta(1)$ ist. Welche Funktion das genau ist, lässt sich nicht sagen, da die konkreten Zahlen auf jeder Hardware unterschiedlich sind.

Durch obige Definition erhalten wir also eine rekursive Formel (eine **Rekurrenz**). Wir wollen aber die Laufzeit nicht als rekursive, sondern als geschlossene Formel angeben. Dazu müssen wir die Rekurrenz auflösen. Das Auflösen von Rekurrenzen kann unter Umständen sehr kompliziert werden. In dieser Veranstaltung lernen wir zwei Methoden:

1. Die Substitutionsmethode
2. Die Master Methode

Neben diesen beiden gibt es noch viele weitere Methoden (z.B. Generating Functions).

Die Substitutionsmethode

Die Substitutionsmethode ist eine Methode zum Lösen von Rekurrenzen. Die Idee für das Vorgehen ist dabei:

1. Wir raten eine Lösung für eine geschlossene Form der Rekurrenz
2. Wir beweisen mit vollständiger Induktion, dass unsere geratene Lösung korrekt ist.

Raten einer Lösung

Für das Raten einer Lösung gibt es kein allgemeines Verfahren. Nach einiger Zeit entwickelt man für viele Rekurrenzen ein *Gefühl* für deren Laufzeit und kann darauf aufbauen. Ansonsten muss man gut raten.

Es gibt allerdings einige Tricks, die man benutzen kann, um besonders gut zu raten:

Einsetzen der Rekurrenz

Meistens reicht es, die Rekurrenz, die man gefunden hat, einige Male in sich selbst einzusetzen. Dann versucht man eventuelle Klammern aufzulösen und ein Muster in der erhaltenen Formel zu erkennen. Ein Beispiel hierfür findet sich in Skriptteil 3: Sortieralgorithmen im Abschnitt Merge-Sort.

Baum-Methode

Die Baum-Methode funktioniert technisch genauso wie das Einsetzen. Allerdings schreibt man hier nicht nur formeln, sondern zeichnet ein Diagramm, aus dem man dann eine Laufzeit erraten kann. An der Wurzel des Baumes steht $T(n)$, also die Laufzeit für das gesamte Problem. Die Kinder dieses Knotens sind dann $T(n/b)$. Von diesen Kindern gibt es a Stück. So kann man (je nachdem wie groß a und b sind) einige Ebenen des Baumes zeichnen. Dann muss man sich überlegen, welche Höhe der Baum hat und wie viele Knoten auf welcher Ebene des Baumes vorhanden sind. Wenn wir so die Anzahl und Größe der Teilprobleme finden, erhalten wir dasselbe Ergebnis wie beim Einsetzen.

Ähnliche Rekurrenzen

Manchmal begegnet man einer Rekurrenz, die einer bekannten Rekurrenz sehr ähnlich sieht. Weiter unten werden wir zum Beispiel zeigen, dass

$$T(n) = 2T(\lfloor n/2 \rfloor) + n = \mathcal{O}(n \log n)$$

ist. Betrachten wir nun die Rekurrenz

$$T(n) = 2T(\lfloor n/2 \rfloor + 25) + n$$

können wir sehen, dass der additive Unterschied von 25 für große n nicht besonders stark ins Gewicht fällt. Daher könnte man auch für dieses Problem raten, dass

$$T(n) = \mathcal{O}(n \log n)$$

ist.

Schrittweise Verbesserung

Oft ist es einfach sowohl eine obere als auch eine untere Grenze für ein Problem zu erraten. Für die Rekurrenz

$$T(n) = 2T(\lfloor n/2 \rfloor) + n$$

können wir aufgrund des $+n$ sofort sehen, dass $T(n) = \Omega(n)$ gilt. Wir können außerdem schnell beweisen, dass $T(n) = \mathcal{O}(n^2)$ gilt. Von dort aus könnten wir die obere und untere Schranke aneinander annähern und so bei einer möglichen Lösung landen.

Achtung

Man sollte nicht versuchen, auf numerischem Weg eine Lösung zu erraten (z.B. durch Plotten der Funktion für $0 \leq n \leq 1\,000\,000$), da auch häufig Laufzeiten wie $\mathcal{O}(\log \log n)$ auftreten. Da zum Beispiel $\log 2^{32} = 32$ ist und $\log \log 2^{32} = \log 32 = 5$ ist, kann es selbst für sehr große n unmöglich sein, den Unterschied zwischen einem konstanten Faktor und $\log \log$ zu erkennen.

Korrektheit der geratenen Lösung beweisen

Wir können die Korrektheit einer geratenen Lösung durch vollständige Induktion beweisen. Wie üblich besteht diese aus Induktionsanfang und Induktionsschritt. Es ist allerdings empfehlenswert, mit dem Induktionsschritt zu beginnen. Falls die geratene Lösung sich als nicht korrekt herausstellt, erkennt man dies in den meisten Fällen im Induktionsschritt, sodass man sich unnötige Arbeit ersparen kann, wenn man mit dem Induktionsschritt beginnt.

Als Beispiel betrachten wir die Rekurrenz

$$T(n) = 2T(\lfloor n/2 \rfloor) + n.$$

Angenommen wir haben die Laufzeit $T(n) = \mathcal{O}(n \log n)$ geraten und wollen dies nun beweisen. Dann müssen wir also zeigen, dass gilt:

$$\exists c > 0 : \exists n_0 : \forall n > n_0 : T(n) \leq c \cdot (n \log n)$$

Induktionschritt Wir beginnen mit dem Induktionsschritt. Dafür stellen wir die folgende Induktionsannahme auf: Angenommen es gilt

$$T(\lfloor n/2 \rfloor) \leq c \cdot \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor).$$

Nun ist

$$\begin{aligned} T(n) &= 2T(\lfloor n/2 \rfloor) + n \\ &\overset{I.A.}{\leq} 2(c \cdot \lfloor n/2 \rfloor \log(\lfloor n/2 \rfloor)) + n \\ &\leq cn \log(n/2) + n \\ &= cn \log n - cn \log 2 + n \\ &= cn \log n - cn + n \\ &= cn \log n - (c-1) \cdot n \\ &\leq cn \log n \quad \text{für } c \geq 1 \end{aligned}$$

Anmerkung

In der Informatik ist der Logarithmus zur Basis 2 sehr häufig. Daher schreibt man normalerweise anstelle von $\log_2(x)$ einfach nur $\log(x)$. Das ist auch bei der Umformung oben der Fall.

Damit haben wir also gezeigt, dass

$$T(n) \leq cn \log n$$

ist, was den Induktionsschritt erfolgreich abschließt.

Induktion ohne $+1$

Wie vielleicht aufgefallen ist, haben wir bei dem Rekursionsschritt nirgendwo das typische $+1$ gebraucht, sondern stattdessen $n/2$ benutzt. **Das ist nicht allgemeingültig.** Wir können diese Vorgehensweise hier verwenden, weil der rekursive Aufruf $T(\lfloor n/2 \rfloor)$ ist. Würden wir nicht abrunden, würden wir mit dieser Vorgehensweise viele mögliche Werte für n auslassen. So allerdings klappt alles.

Induktionsanfang Der Induktionsanfang muss den Basisfall (*engl.: base case / boundary conditions*) abdecken. Normalerweise ist dies das kleinste n, das als Eingabe für den Algorithmus erlaubt ist.

Wir probieren also $T(n) \leq cn \log n$ für $n = 1$ zu zeigen. Dazu nehmen wir an, dass $T(1) = 1$ ist[1].

Dann setzen wir ein:

$$\begin{aligned} T(1) = 1 &\leq c \cdot 1 \cdot \log 1 = c \cdot 1 \cdot 0 = 0 \\ \Rightarrow 1 &\leq 0 \end{aligned}$$

Hier haben wir also einen Widerspruch. Wir können unsere Behauptung also nicht für $T(1)$ beweisen. Das macht aber nichts. Wir wählen stattdessen einfach einen anderen Basisfall und berücksichtigen $n = 1$ nicht in dem induktiven Beweis.

Hier müssen wir nun aufpassen: Da die Rekurrenz bei jedem rekursiven Aufruf abrundet, würde sowohl für $n = 2$ als auch für $n = 3$ ein rekursiver Aufruf $T(1)$ gemacht werden. Da wir aber für diesen Aufruf die Laufzeit nicht induktiv beweisen können, müssen wir für den Beweis nun als Basisfall sowohl $n = 2$ als auch $n = 3$ berücksichtigen (alle größeren Werte für n werden immer entweder auf $n = 2$ oder $n = 3$ zurückgeführt, der Beweis ist hier nicht aufgeführt).

[1] Diese Annahme basiert auf dem Algorithmus, aus dem wir die Rekurrenz erhalten haben. In diesem Beispiel ist der Algorithmus nicht vorgegeben, daher nehmen wir $T(1) = 1$ an.

Anmerkung: Wir lassen $T(1)$ in dem Beweis außer Acht. Für die Rekurrenz bleibt aber $n = 1$ der Basisfall.

Wir wollen also unsere Behauptung $T(n) = \mathcal{O}(n \log n)$ für $n = 2$ und $n = 3$ beweisen. Wir setzen ein:

$$T(2) = 2\underbrace{T(1)}_{1} + 2 = 2 + 2 = 4 \leq c \cdot 2 \log 2$$
$$T(3) = 2\underbrace{T(1)}_{1} + 3 = 2 + 3 = 5 \leq c \cdot 3 \log 3$$

Wir können also $c \geq 2$ für diese beiden Fälle wählen. Da der Induktionsschritt nur $c \geq 1$ gefordert hat, reicht also $c \geq 2$ aus, um unsere Behauptung zu beweisen. Dies schließt den Induktionsschritt ab und es gilt

$$T(n) = 2T(\lfloor n/2 \rfloor) + n = \mathcal{O}(n \log n)$$

für genügend große Werte von n (nämlich $n \geq 2$).

Variablensubstitution

Variablensubstitution ist eigentlich kein alternatives Verfahren zum Lösen von Rekurrenzen. Allerdings kann man sich dies als Trick zunutze machen, um sich Rekurrenzen zu vereinfachen.

Als Beispiel betrachten wir

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$$

Zuerst können wir die Gaußklammern weglassen, da das Abrunden einen Beweis nur komplizierter macht und für eine obere Schranke egal ist. Nun können wir eine neue Variable einführen:

$$m = \log n \Leftrightarrow 2^m = n$$
$$\Rightarrow \sqrt{n} = n^{1/2} = (2^m)^{1/2} = 2^{m/2}$$

Dann können wir n für m substituieren und erhalten

$$T(2^{m/2}) = 2T(2^{m/2}) + m.$$

Jetzt setzen wir $S(m) = T(2^m)$ und erhalten damit

$$S(m) = 2S(m/2) + m.$$

Für diese Rekurrenz kennen wir bereits die Lösung. Es ist

$$S(m) = \mathcal{O}(m \log m).$$

Jetzt reicht es, zurück zu substituieren und wir erhalten

$$T(n) = T(2^m) = S(m) = \mathcal{O}(m \log m) = \mathcal{O}(\log n \log \log n).$$

Die Master Methode

Es gibt viele Algorithmen, deren Laufzeit durch sehr unterschiedliche Rekurrenzen dargestellt wird. Es gibt aber auch viele Algorithmen, bei denen die Rekurrenzen sehr ähnlich sind. Die Master Methode verallgemeinert einige der häufigsten Fälle von Rekurrenzen und liefert für diese sofort die Lösungen.

Als Ausgangssituation haben wir eine Rekurrenz der Form

$$T(n) = a \cdot T(n/b) + f(n)$$

mit Konstanten $a \geq 1$ und $b > 1$ sowie einem asymptotisch positiven $f(n)$. Die Bedeutung von a und b bleibt wie bisher. In $f(n)$ steckt der Aufwand für das Aufteilen eines Problems in a Teilprobleme sowie der Aufwand die Lösungen der Teilprobleme zu integrieren.

Die Master Methode hat dann drei Fälle:

Definition 2.1 (Master Method)**.** *Seien $a \geq 1$ und $b > 1$ konstant. Sei $f(n)$ eine Funktion und $T : \mathbb{N} \to \mathbb{R}$ mit*

$$T(n) = a \cdot T(n/b) + f(n)$$

wobei n/b entweder $\lfloor n/b \rfloor$ oder $\lceil n/b \rceil$ bedeuten kann (oder n/b, wenn wir wissen, dass n eine 2er Potenz ist). Dann kann $T(n)$ folgendermaßen asymptotisch beschränkt werden:

1. *Gilt $f(n) = \mathcal{O}\left(n^{(\log_b a)-\epsilon}\right)$ für ein $\epsilon > 0$, dann ist*

   $$T(n) = \Theta\left(n^{\log_b a}\right).$$

   Alternativdefinition: Für $f(n) = o(n^{\log_b a})$.

2. *Gilt $f(n) = \Theta\left(n^{\log_b a}\right)$, dann ist*

   $$T(n) = \Theta\left(n^{\log_b a} \cdot \log n\right).$$

3. *Gilt $f(n) = \Omega\left(n^{(\log_b a)+\epsilon}\right)$ für ein $\epsilon > 0$ mit $a \cdot f(n/b) \leq c \cdot f(n)$ für ein $c < 1$ und für genügend große n, dann ist*

   $$T(n) = \Theta\left(f(n)\right).$$

   Alternativdefinition: Für $f(n) = \omega(n^{\log_b a})$.

Anwendung der Master Methode

Das Anwenden der Master-Methode ist relativ einfach:

1. Zerlege eine gegebene Rekurrenz so, dass man a, b und $f(n)$ erhält.
2. Tipp: Berechne an dieser Stelle $\log_b(a)$.
3. Vergleiche $f(n)$ mit $n^{\log_b a}$.
4. Wähle den passenden Fall aus und setze ein.

Grenzen der Master Methode

Die Master Methode ist nicht für alle Rekurrenzen anwendbar, sondern nur for Rekurrenzen der Form

$$T(n) = a \cdot T(n/b) + f(n)$$

Beweis der Master Methode

Wir werden den Beweis hier nicht ausführen. Die Beweisidee ist allerdings denkbar einfach: Wir generalisieren das Baum-Verfahren zum Raten einer möglichen Lösung für Rekurrenzen der gegebenen Form. Im Detail lässt sich dies in den gängigen Lehrbüchern nachlesen.

Beispiel 1

Gegeben sei die Rekurrenz

$$T(n) = 9T(n/3) + n$$

Hier ist also $a = 9$, $b = 3$ und $f(n) = n$. Also ist $\log_3 9 = 2$ und $n^{\log_3 9} = n^2$. Offensichtlich ist $f(n) = n = \mathcal{O}(n^{\log_3 9-\epsilon}) = \mathcal{O}(n^{2-\epsilon})$ für $\epsilon = 1$. Alternativ lässt sich argumentieren, dass $f(n) = n = o(n^2))$ ist. Also befinden wir uns in Fall 1 der Master Methode. Als Ergebnis erhalten wir

$$T(n) = \Theta(n^2)$$

Beispiel 2

Gegeben sei die Rekurrenz

$$T(n) = T(2n/3) + 1$$

Hier ist nun $a = 1$, $b = 3/2$ und $f(n) = 1$. Damit ist $n^{\log_b a} = n^{\log_{2/3} 1} = n^0 = 1$. Wir befinden uns also in Fall 2 der Master Methode, da $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$ ist. Das Ergebnis ist damit

$$T(n) = \Theta(\log n)$$

Beispiel 3

Gegeben sei die Rekurrenz

$$T(n) = 3T(n/4) + n \log n$$

Es ist $a = 3$, $b = 4$ und $f(n) = n \log n$. Damit ist $n^{\log_b a} = n^{\log_4 3} = \Theta(n^{0.793})$. Offensichtlich ist nun $f(n) = n \log n = \Omega(n)$. Daher muss auch $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ für $\epsilon \approx 0.2$ sein.

Da außerdem $a \cdot f(n) = 3\frac{n}{4} \log\left(\frac{n}{4}\right) \leq \frac{3}{4} n \log n = c \cdot f(n)$ ist (für $c = \frac{3}{4}$), können wir Fall 3 der Master Methode anwenden und erhalten

$$T(n) = \Theta(n \log n)$$

3. Sortieralgorithmen

In diesem Kapitel beschäftigen wir uns mit einigen Algorithmen, die auf unterschiedliche Weise eine Liste von Zahlen sortieren. Als Vorwissen wird vorausgesetzt, dass man mindestens Bubble-Sort, Insertion-Sort und Selection-Sort kennt. Alle drei Algorithmen haben eine Laufzeit von $\mathcal{O}(n^2)$.

Merge-Sort

Merge-Sort ist einer der einfachsten rekursiven Sortieralgorithmen mit einer ziemlich guten schlechtesten Laufzeit. Technisch gesehen ist Merge-Sort immer in $\Theta(n \log n)$. Andere Algorithmen (z.B. Bubble-Sort oder Insertion-Sort) können zwar in einigen Spezialfällen in n sortieren, brauchen aber in den meisten Fällen $\mathcal{O}(n^2)$ und sind damit tendentiell langsamer.

Idee

1. Wir teilen die Liste von Zahlen in zwei gleich große Hälften.
2. Wir sortieren jede Hälfte rekursiv.
3. Wir *mergen* die beiden sortierten Teillisten zusammen.

Pseudocode

```
Merge-Sort(A, p, r)
1  if p < r
2      q = p + ⌊(r − p)/2⌋
3      Merge-Sort(A, p, q)
4      Merge-Sort(A, q + 1, r)
5      Merge(A, p, q, r)
```

Der Algorithmus teilt ein gegebenes Array der Länge l in zwei Hälften der Länge $\frac{l}{2}$ und sortiert diese dann rekursiv. Anschließend werden die beiden sortierten Teile wieder zusammengefügt. Um mit Merge-Sort eine Liste $A = A[1], A[2], ..., A[n]$ zu sortieren, ruft man Merge-Sort$(A, 1, n)$ auf.

Die Merge-Funktion lässt sich folgendermaßen in $\Theta(n)$ implementieren:

```
MERGE(left, right)
 1  list result
 2  while length(left) > 0 or length(right) > 0
 3      if length(left) > 0 & length(right) > 0
 4          if first(left) ≤ first(right)
 5              append first(left) to result
 6              left = rest(left)
 7          else
 8              append first(right) to result
 9              right = rest(right)
10      elseif length(left) > 0
11          append left to result
12          left = EMPTY LIST
13      elseif length(right) > 0
14          append right to result
15          right = EMPTY LIST
16  return result
```

Laufzeitanalyse

Zur Analyse der Laufzeit von Merge-Sort stellen wir zunächst fest, dass es sich hier um einen Divide and Conquer Algorithmus handelt. In diesem Fall ist

- $a = b = 2$
- $D(n) = \Theta(1)$
- $C(n) = \Theta(n)$

Daraus ergibt sich die folgende Formel für die Laufzeit:

$$T(n) = 2T(n/2) + \Theta(n)$$

Das letzte $\Theta(1)$ kann weggelassen werden, da $\Theta(n)$ asymptotisch größer ist und für große n deutlich größer ist als $\Theta(1)$. Diese rekursive Formel wollen wir nun zu einer geschlossenen Formel auflösen.

Jetzt wenden wir die *Guess and Verify*-Methode an. Dazu setzen wir einfach $T(n)$ einige Male rekursiv ein:

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= 2(2T(n/4) + \Theta(n/2)) + \Theta(n) \\ &= 2(2(2T(n/8) + \Theta(n/4)) + \Theta(n/2)) + \Theta(n) \\ &= 2^3T(n/2^3) + 2^2\Theta(n/2^2) + 2^1\Theta(n/2^1) + 2^0\Theta(n/2^0) \end{aligned}$$

Da es sich bei n um eine positive natürliche Zahl handelt, können wir die Rekursion nicht unendlich oft wiederholen, sondern nur $\log_2 n$ Mal. Damit erhalten wir:

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

Für einen Summanden $2^i \cdot \Theta\left(\frac{n}{2^i}\right)$ ist $i < \log n$. Uns soll hier die obere Schranke für Merge-Sort reichen. Dazu tun wir so, als ob $i = \log n$ in jedem Summanden wäre. So erhalten wir für einen Summanden $2^{\log n} \cdot \mathcal{O}\left(\frac{n}{2^{\log n}}\right) = n \cdot \mathcal{O}(1)$. Da es $\log n$ Summanden gibt, ist also die Laufzeit von Merge-Sort von oben durch

$$n \cdot T(1) + \log n \cdot n \cdot \mathcal{O}(1)$$

beschränkt. Das legt nahe, dass wir für die Laufzeit

$$T(n) = \mathcal{O}(n \log n)$$

raten.

Wichtig: Bis zu diesem Punkt ist die Laufzeit nur eine Behauptung. Im nächsten Schritt muss diese Behauptung durch vollständige Induktion bewiesen werden.

Eigentlich hätten wir uns vieles dieser Arbeit sparen können. Merge-Sort kann nämlich auch durch die Master-Methode gelöst werden. In der Form

$$T(n) = 2T(n/2) + \Theta(n)$$

ist dann $a = b = 2$ und $f(n) = n$. Da außerdem $n^{\log_b a} = n^{\log_2 2} = n^1 = n$ ist, befinden wir uns in Fall 2 der Master-Methode und können direkt einsetzen:

$$\Theta(n^{\log_b a} \cdot \log n) = \Theta(n \log n)$$

Quicksort

Quicksort ist wie Merge-Sort ein Sortieralgorithmus der Divide and Conquer Klasse.

Idee

1. Suche ein Pivotelement in der Liste aus
2. **Divide**: Teile die Liste so, dass alle Elemente, die kleiner sind als das Pivotelement links und alle größeren rechts vom Pivotelement stehen.

3. Dann steht das Pivotelement bereits an der richtigen Stelle.
4. **Conquer**: Sortiere die Teillisten links und rechts vom Pivotelement rekursiv.
5. Da wir in-place sortieren, sind nun alle Elemente an der richtigen Stelle.

Pseudocode

Der Pseudocode ist denkbar einfach und setzt direkt unsere Idee um.

```
QUICKSORT(A, l, r)
1  if l < r
2        q = PARTITION(A, l, r)
3        QUICKSORT(A, l, q − 1)
4        QUICKSORT(A, q + 1, r)
```

Zum Sortieren eines Arrays ruft man dann QUICKSORT(A, 1, $length(A)$) auf.

Die Magie des Algorithmus liegt aber offenbar in der PARTITION-Funktion:

```
PARTITION(A, l, r)
1  x = A[r]
2  i = l − 1
3  for j = l to r − 1
4        if A[j] ≤ x
5              i = i + 1
6              exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

Wir wählen ein Pivotelement x (in diesem Fall immer das Element am rechten Rand). Der Zähler i zeigt immer auf das letzte bekannte Element, das kleiner (oder gleich) dem Pivotelement ist. Am Anfang kennen wir noch keine solche Elemente, also zeigt i *vor* die Liste ($l - 1$). Immer wenn wir nun beim Durchgehen der Elemente ein Element finden, dass kleiner ist als das Pivotelement, inkrementieren wir i und tauschen das Element mit der Stelle. So sammeln sich die Elemente, die kleiner oder gleich dem Pivotelement sind, links im Array. Am Ende tauschen wir noch das Pivotelement an die richtige Stelle und sind dann fertig.

Diese Implementierung von PARTITION ist nicht besonders *gut*, da wir immer das rechte Element der Liste als Pivotelement wählen. Wenn wir *Glück* haben und das Pivotelement immer der Median der Liste ist, arbeitet der Algorithmus analog zu MERGE-SORT. Im schlimmsten Fall ist aber das Pivotelement immer das größte Element der Liste, wodurch der Algorithmus analog zu Selection-Sort arbeitet (und damit in $\mathcal{O}(n^2)$ ist). Es gibt viele Möglichkeiten, QUICKSORT durch die Wahl des Pivotelements zu optimieren. Zum Beispiel

- Wähle (z.B.) 5 Elemente und wähle deren Median als Pivotelement.
- Wähle ein zufälliges Pivotelement.
- Wähle (z.B.) 5 zufällige Elemente und wähle deren Median als Pivotelement.
- Viele mehr...

Für unbekannte Eingaben ist es normalerweise am effizientesten, ein zufälliges Pivotelement zu wählen.

Laufzeitanalyse

Die Laufzeit von QUICKSORT hängt offensichtlich davon ab, wie gut unser Pivotelement gewählt ist. Bei einem schlechten Pivotelement bekommen wir ein schlechtes **Partitionierungsverhältnis** und damit eine hohe Laufzeit.

Worst Case

Im Worst Case ergibt sich damit die folgende Laufzeit:

$$T(n) = \max_{0 \leq q \leq n} \{T(q) + T(n-q-1)\} + \Theta(n)$$

In jeder Iteration wählen wir das *schlechteste* Pivotelement q. Das *schlechteste* Pivotelement ist so definiert, dass dadurch die Laufzeit maximal wird. Wir maximieren also die Laufzeit über alle möglichen Partitionierungsverhältnisse q und $n-q-1$.

Das $\Theta(n)$ am Ende repräsentiert den Aufwand, der für das Partitionieren selbst nötig ist. Man sieht leicht an der obigen PARTITION-Funktion, dass dies in linearer Zeit möglich ist. Alternativ hätte man auch $d \cdot n$ schreiben können (für eine Konstante d).

Wir raten $T(n) \leq cn^2$ (für eine Konstante c) als worst-case Laufzeit von QUICKSORT. Dann können wir einsetzen:

$$\begin{aligned} T(n) &= \max_{0 \leq q < n} \{T(q) + T(n-q-1)\} + \Theta(n) \\ &= \max_{0 \leq q < n} \{cq^2 + c(n-q-1)^2\} + \Theta(n) \\ &= c \cdot \max_{0 \leq q < n} \{q^2 + (n-q-1)^2\} + \Theta(n) \end{aligned}$$

Man sieht schnell, dass der Ausdruck

$$q^2 + (n-q-1)^2$$

für $0 \leq q < n$ bei $q = 0$ oder $q = n - 1$ maximal ist. Also setzen wir $q = n - 1$ ein (dies ist gleichbedeutend damit, dass wir immer das größte Element als Pivotelement wählen). Damit erhalten wir

$$q^2 + (n - q - 1)^2 \leq (n - 1)^2 + (n - (n - 1) - 1)^2 = (n - 1)^2$$

und können dies in die Rekurrenz einsetzen:

$$\begin{aligned} T(n) &\leq c \cdot \max_{0 \leq q < n} \{q^2 + (n - q - 1)^2\} + \Theta(n) \\ &\leq c \cdot (n - 1)^2 + \Theta(n) \\ &\leq cn^2 - 2cn + c + dn \\ &= cn^2 - (2cn - dn - c) \\ &\leq cn^2 \end{aligned}$$

Der letzte Schritt geht nur, wenn $2cn \geq dn + c$ ist. Das ist aber offensichtlich möglich. Damit haben wir eine obere Schranke für unseren QUICKSORT-Algorithmus gefunden:

$$T(n) = \mathcal{O}(n^2)$$

Da wir außerdem ein Beispiel konstruieren können, für welches der Algorithmus tatsächlich diese Zeit benötigt (eine bereits sortierte Liste), können wir die worst-case Performance von QUICKSORT mit $\Theta(n^2)$ bestimmen.

Best-Case

QUICKSORT muss nicht in $\Theta(n^2)$ laufen. Wenn wir in jedem Rekursionsdurchgang ein nahezu perfektes Partitionierungsverhältnis erreichen (also das Pivotelement der Median der Liste ist), dann hat QUICKSORT dieselbe Laufzeit wie Merge-Sort:

$$\begin{aligned} & T(n) \leq 2T(n/2) + \Theta(n) \\ \implies & T(n) = \Theta(n \log n) \end{aligned}$$

Ausgewogenes Partitionierungsverhältnis

Offenbar haben wir bei QUICKSORT nun eine besondere Situation: Wenn wir das Pivotelement schlecht wählen, ist die Laufzeit in $\Theta(n^2)$. Wenn wir ein gutes Pivotelement wählen, ist sie in $\Theta(n \log n)$. Was passiert nun, wenn wir ein Partitionierungsverhältnis irgendwo dazwischen wählen? Als Beispiel betrachten wir ein Verhaltnis von 13 zu 2. Damit ergibt sich folgende Rekurrenz:

$$T(n) \leq T(13n/15) + T(2n/15) + \underbrace{cn}_{\text{Laufzeit der Partitionierung}}$$

Anmerkung: Hier steht nun cn statt $\Theta(n)$. Beide Schreibweisen sind äquivalent.

In einem Entscheidungsbaum der Quicksort-Funktion würde es nun also einen sehr kurzen Pfad geben (bei dem immer ein rekursiver Aufruf mit $2n/15$ gemacht wird) und einen sehr langen Pfad (wenn immer mit $13n/15$ aufgerufen wird). Es gibt außerdem noch viele Pfade, die eine Länge irgendwo dazwischen haben. Wir wollen nun eine obere Zeitschranke für dieses Partitionierungsverhältnis finden.

Dazu tun wir so, als ob jeder Pfad so lang wäre wie der längste Pfad. Da wir nur eine obere Zeitschranke finden wollen, ist dies in Ordnung, da wir dadurch die Laufzeit schlimmstenfalls zu hoch abschätzen, aber immernoch eine gültige obere Zeitschranke erhalten. Auf dem längsten Pfad schrumpft die Eingabe bei jedem rekursiven Aufruf um den Faktor $\frac{13}{15}$. Also ist

$$\left(\frac{13}{15}\right)^x \cdot n \leq 1$$

für ein x. Dieses x wollen wir nun finden:

$$\begin{aligned}
&\left(\frac{13}{15}\right)^x \cdot n \leq 1 \\
\iff &\left(\frac{13}{15}\right)^x \leq \frac{1}{n} \\
\iff &\left(\frac{15}{13}\right)^x \geq n \\
\iff &x \log(15/13) \geq \log n \\
\iff &x \geq \log_{\frac{15}{13}} n = \underbrace{\frac{1}{\log \frac{15}{13}}}_{\text{konstant}} \cdot \log n \\
&= \Theta(\log n)
\end{aligned}$$

Also hat der längste Pfad eine Länge in $\Theta(\log n)$. Da wir außerdem beim Partitionieren jedes Element genau ein Mal mit dem Pivotelement vergleichen, führen wir auf jeder Ebene des Baumes eine Arbeit von höchstens $\mathcal{O}(n)$ durch. Damit können wir die Gesamtlaufzeit auf

$$T(n) = \mathcal{O}(n \log n)$$

abschätzen.

Überraschenderweise ist also ein Teilungsverhältnis von 13 zu 2 immer noch in $\Theta(n \log n)$. Tatsächlich gilt diese Laufzeitschranke bei Quicksort, wenn wir das Pivotelement immer so wählen, dass die Eingabe in einem konstanten Verhältnis proportional partioniert wird.

Anmerkung

Normalerweise interessiert uns die Average-Case Laufzeit von Algorithmen nur wenig. Für die Informatik ist normalerweise nur die obere Schranke interessant. Für Klassen von Algorithmen ist es normalerweise außerdem interessant, eine untere

Schranke anzugeben. So ist es beispielsweise nicht möglich, mit vergleichsbasierten Sortierverfahren besser als in $\Theta(n \log n)$ zu sortien.

Randomisiertes Quicksort

Trotz obiger Anmerkung wollen wir nun eine average-case Analyse von QUICKSORT machen. Die naive Methode dafür wäre, jede mögliche Eingabe des Algorithmus zu betrachten und dann durch die Anzahl der möglichen Eingaben zu teilen. Im Falle von QUICKSORT sind das alle Permutationen von n (geschrieben $[n]$):

$$T_{avg}(n) = \frac{1}{n!} \cdot \sum_{p \in [n]} T(p)$$

Idee

Allerdings stört uns, dass es im Prinzip möglich ist, absichtlich eine schlechte Eingabe zu konstruieren. Wir wollen also unsere Strategie zum Auswählen des Pivotelementes so verändern, dass es nicht mehr möglich ist, absichtlich einen *schlechten* Input zu geben. Dazu haben wir zwei Möglichkeiten:

1. Wir permutieren die Eingabe zufällig und lassen den Algorithmus wie er ist.
2. Wir lassen die Eingabe wie sie ist und wählen in jedem Durchgang ein zufälliges Pivotelement.

Tatsächlich unterscheiden sich die beiden Möglichkeiten nur unwesentlich voneinander. Wir entscheiden uns hier für Version 1.

Pseudocode

```
RANDOMIZED-QUICKSORT(A, l, r)
1  if l < r
2        q = RANDOMIZED-PARTITION(A, l, r)
3        RANDOMIZED-QUICKSORT(A, l, q − 1)
4        RANDOMIZED-QUICKSORT(A, q + 1, r)
```

```
RANDOMIZED-PARTITION(A, l, r)
1  z = Random(l, r)
2  swap A[z] ↔ A[r]
3  return PARTITION(A, l, r)
```

Diese Implementierung ist auch optimal. Wir könnten eine Version von RANDOMIZED-PARTITION schreiben, die für randomisierte Werte optimiert ist. Allerdings soll uns diese Version für jetzt reichen.

Laufzeitanalyse

Wir wollen nun die Laufzeit vom randomisierten Quicksort analysieren. Offensichtlich gelten die oberen und unteren Schranken von der nicht randomisierten Version auch hier. Allerdings können wir bei dieser Version die erwartete Laufzeit (stochastisch) bestimmen. Diese Laufzeit tritt im Erwartungsfall für eine beliebige Eingabe ein.

Intuitiv erwarten wir für die Laufzeit vom randomisierten Quicksort $\mathcal{O}(n \log n)$, da wir erwarten, dass ein zufälliges Pivotelement die Eingabe mehr oder weniger in zwei gleich große Teile aufteilt.

Wir bemerken:

- Die Laufzeit vom randomisierten Quicksort wird durch PARTITION dominiert.
- Jedes Mal wenn PARTITION aufgerufen wird, wählen wir ein Pivotelement.
- Ein Pivotelement wird in **keinem** folgenden Aufruf eingeschlossen.
- Also kann es insgesamt höchstens n Aufrufe von PARTITION geben.
- Ein Aufruf von PARTITION braucht $\mathcal{O}(1)$ sowie eine Laufzeit, die proportional zur Anzahl der Durchläufe der Schleife ist.
- In jeder Iteration wird das Pivotelement mit einem anderen Element verglichen.
- Also müssen wir eine Schranke für die Gesamtanzahl der Vergleiche finden, die gemacht werden. Diese dominiert die Laufzeit von Quicksort.

Da wir zusätzlich zu den Vergleichen auch n Aufrufe von PARTITION haben, können wir die Laufzeit von Quicksort schon mal durch $\mathcal{O}(n + X)$ beschränken, wobei X die Anzahl der Vergleiche ist, die während des gesamten Algorithmus durchgeführt werden. Wir suchen nun eine Beschränkung für X.

Dazu definieren wir auf dem Ergebnis von Quicksort (also dem sortierten Array) die Sequenz $z_1, z_2, \ldots, z_n$, wobei z_i das i-t kleinste Element ist. Nun definieren wir

$$Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$$

und stellen fest, dass jedes Paar von Elementen **höchstens ein Mal** miteinander verglichen wird und dass in dem Fall genau eines der beiden Elemente das Pivotelement sein muss. Daher definieren wir die Zufallsvariable

$$X_{ij} = \begin{cases} 1 & z_i \text{ wird irgendwann mit } z_j \text{ verglichen} \\ 0 & \text{sonst} \end{cases}$$

Da jedes Paar von Elementen höchstens ein Mal miteinander verglichen wird, ist die Gesamtzahl an Vergleichen also definiert durch

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

Da wir die erwartete Laufzeit bestimmen wollen, bestimmen wir zunächst die erwartete Anzahl von Vergleichen:

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(z_i \text{ wird mit } z_j \text{ verglichen}) \end{aligned}$$

Die erste Umformung ist aufgrund der Linearität des Erwartungswertes möglich. Für die zweite Umformung bemerken wir, dass der Erwartungswert i.A. definiert ist als $E[X_{ij}] = \sum_x (x \cdot P(X_{ij} = x))$. Unser X_{ij} kann nur die Werte 0 und 1 annehmen, sodass gilt:

$$\begin{aligned} E[X_{ij}] &= 0 \cdot P(X_{ij} = 0) + 1 \cdot P(X_{ij} = 1) \\ &= P(X_{ij} = 1) \end{aligned}$$

$X_{ij} = 1$ ist nach Definition der Fall, wenn z_i irgendwann mit z_j verglichen wird. Da man an der Doppelsumme sofort sieht, dass jede Kombination i, j nur ein Mal vorkommt, können wir das „irgendwann“ weglassen.

Es bleibt $P(z_i \text{ wird mit } z_j \text{ verglichen})$ zu bestimmen. Einfacher ist es, sich die Gegenwahrscheinlichkeit anzusehen, also die Wahrscheinlichkeit, dass die beiden Elemente **nicht** miteinander verglichen werden.

- Nachdem ein Pivotelement x mit $z_i < x < z_j$ ausgewählt ist, können z_i und z_j nicht mehr miteinander verglichen werden, da sie in unterschiedlichen Zweigen des Rekursionsbaumes sind.
- Wenn z_i als erstes Pivotelement in Z_{ij} gewählt wird, dann wird z_i mit jedem anderen Element in Z_{ij} verglichen.
- Gleiches gilt für z_j

Also muss entweder z_i oder z_j als erstes Pivotelement aus Z_{ij} gewählt werden, damit die beiden miteinander verglichen werden. Bevor irgendein Element aus Z_{ij} als Pivotelement gewählt wird, ist die ganze Menge in derselben Partition. Damit ist auch klar, dass jedes

Element in Z_{ij} dieselbe Wahrscheinlichkeit hat, als Pivotelement ausgewählt zu werden. Da $|Z_{ij}| = j - i + 1$ ist, ist diese Wahrscheinlichkeit $\frac{1}{j-i+1}$ für jedes Element (*Anmerkung*: Dies ist nicht die Wahrscheinlichkeit, dass ein Element innerhalb der Partition als Pivot gewählt wird. Die Wahrscheinlichkeit gilt nur bei der Wahl in Z_{ij}).

Also ist

$$\begin{aligned}
&P(z_i \text{ wird mit } z_j \text{ verglichen})\\
=&P(z_i \text{ oder } z_j \text{ wird als erstes Pivotelement in } Z_{ij} \text{ gewählt})\\
\stackrel{*}{=}&P(z_i \text{ wird als erstes Pivotelement in } Z_{ij} \text{ gewählt})+\\
&P(z_j \text{ wird als erstes Pivotelement in } Z_{ij} \text{ gewählt})\\
=&\frac{1}{j-i+1} + \frac{1}{j-i+1}\\
=&\frac{2}{j-i+1}
\end{aligned}$$

* *gilt, weil die beiden Ereignisse sich gegenseitig ausschließen.*

Damit können wir in die Doppelsumme von oben einsetzen:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(z_i \text{ wird mit } z_j \text{ verglichen}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

Zur besseren Übersicht setzen wir nun $k = j - i$.

$$\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}\\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} = 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k}\\
&< 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k} \stackrel{*}{=} 2 \sum_{i=1}^{n-1} \mathcal{O}(\log n)\\
&= \mathcal{O}(n \log n)
\end{aligned}$$

* folgt daraus, dass Der vorletzte Schritt ist möglich, da die harmonische Summe durch $\mathcal{O}(\log n)$ beschränkt ist[1].

Damit haben wir gezeigt, dass die erwartete Laufzeit vom randomisierten Quicksort durch

$$\mathcal{O}(n \log n)$$

beschränkt ist.

[1] http://stackoverflow.com/a/25905474

Das schnellste Quicksort

Quicksort ist einer der schnellsten allgemeinen Sortierverfahren. Durch die Randomisierung haben wir in der Theorie fast alle negativen Effekte beseitigt. In der Realität ist aber nicht jedes $\Theta(n \log n)$ gleich. Die Konstanten, die durch die Notation versteckt werden, können immer noch unterschiedlich sein.

Für Quicksort gibt es sogar Wettbewerbe, bei denen versucht wird, eine möglichst schnelle Implementierung von Quicksort zu machen (also mit möglichst kleinen Konstanten). Aber wie wir bewiesen haben, wird auch der größte Aufwand nicht dafür sorgen, dass Quicksort schneller als $\Omega(n \log n)$ wird, selbst bei noch so kleinen Konstanten.

Vergleichsbasierte Sortieralgorithmen

Bisher haben wir verschiedene vergleichsbasierte Sortieralgorithmen kennen gelernt. Die beste Laufzeit, die wir dabei erreicht haben ist $\Theta(n \log n)$. Das wirft die Frage auf, ob wir überhaupt schneller sortieren können.

Die Antwort ist im Allgemeinen ja. Es gibt Sortieralgorithmen (z.B. BucketSort), die in linearer Zeit sortieren. Diese sind allerdings nicht in allen Fällen anwendbar. Für vergleichsbasierte Sortieralgorithmen (also solche, die Elemente miteinander vergleichen) gilt eine untere Schranke von $\Theta(n \log n)$.

Um dies zu beweisen betrachten wir Entscheidungsbäume. Für jeden vergleichsbasierten Algorithmus kann man einen Entscheidungsbaum zeichnen. Dieser wird wie folgt konstruiert:

1. Beginnend bei der Wurzel zeichnen wir Knoten und beschriften diese mit $i : j$ mit $1 \leq i, j \leq n$. Ein solcher Knoten bedeutet, dass die Elemente i und j miteinander verglichen werden.
2. Jeder dieser Knoten hat zwei Kinder. Die Kanten werden entsprechend mit $\leq$ und $>$ beschriftet und repräsentieren den Ausgang des Vergleiches.
3. Die Blätter des Baumes sind Permutationen der Eingabe.

In so einem Entscheidungsbaum beschreibt ein Pfad von der Wurzel zu einem Blatt die Folge von Vergleichen, die vom Algorithmus gemacht werden, um festzustellen, dass die Eingabe der Permutation des Blattes entspricht. Die Länge eines Pfades von der Wurzel zu einem Blatt ist genau die Anzahl der Vergleiche, die der Algorithmus in dem Fall macht.

Jeder vollständige Sortierungsalgorithmus muss jede Permutation erzeugen können. Kann ein Algorithmus das nicht, bedeutet dies im Umkehrschluss, dass es mindestens eine

Permutation gibt, die der Algorithmus nicht korrekt sortieren kann. Das heißt also, dass es in so einem Entscheidungsbaum mindestens $n!$ Blätter geben muss.

Wir nehmen an, dass wir für jeden möglichen Entscheidungsbaum auch einen Algorithmus entwickeln können. Damit ist dann jede untere Schranke der Höhe aller Entscheidungsbäume mit $n!$ Blättern auch gleichzeitig eine untere Schranke für jeden beliebigen vergleichsbasierten Sortieralgorithmus.

Wir wissen, dass jeder binäre Baum (und damit jeder Entscheidungsbaum) der Höhe h höchstens 2^h Blätter haben kann (in dem Fall ist jede Ebene des Baumes vollständig). Das können wir umformen zu $h \geq \log(n!)$. Es gilt nun $\log(n!) = \Omega(n \log n)$[2]. Die mathematische Grundlage dafür ist die Stirlingformel[3]. Damit gilt

$$h = \Omega(n \log n)$$

Damit ist eine untere Schranke für jedes vergleichsbasierte Sortierverfahren

$$\Omega(n \log n).$$

Finden des k-t kleinsten Elementes in einer ungeordneten Liste

Oftmals ist es für eine Anwendung gar nicht nötig, eine ganze Liste zu sortieren. Manchmal brauchen wir nur das k-t kleinste Element aus einer Liste. In diesem Fall wäre es unnötig, die ganze Liste zu sortieren, da das deutlich länger dauern würde.

Zunächst stellen wir fest, dass es sehr einfach ist, das kleinste Element zu finden. Auch das zweitkleinste oder größte Element kann offensichtlich in linearer Zeit gefunden werden. Dies geht allerdings nicht mehr so einfach, wenn wir das $\sqrt{n}$-t kleinste Element finden wollen.

Wir stehen also vor folgendem Problem: Gegeben ist eine Liste A mit n verschiedenen Zahlen sowie eine Zahl $i \in \{1, \ldots, n\}$. Wir suchen das Element $x \in A$, welches größer als genau $i - 1$ andere Elemente in A ist.

Idee

Der naive Ansatz für dieses Problem ist, die gegebene Liste zuerst zu zu sortieren und dann das k-te Element abzulesen. Dies wäre offensichtlich in $\Theta(n \log n)$ möglich. Beim Sortieren tun wir aber deutlich mehr als wir eigentlich müssten. Daher entwickeln wir eine Funktion $\textsc{Select}(A, i)$ wie folgt:

[2] http://stackoverflow.com/a/8118257
[3] https://de.wikipedia.org/wiki/Stirlingformel

1. Teile A in $\lfloor n/5 \rfloor$ Gruppen mit jeweils 5 Elementen auf. Dazu kommt noch maximal eine Gruppe mit genau $n \mod 5 < 5$ Elementen (7 statt 5 zu verwenden wäre genauso möglich. Wichtig ist vor allem, dass wir hier eine Konstante wählen).
2. Sortiere jede der Gruppen und finde den Median (das mittlere Element nach dem Sortieren).
3. Rufe Select rekursiv auf den Medianen der $\lceil n/5 \rceil$ Gruppen auf. Wir erhalten so den Median der Mediane x.
4. Partitioniere die Eingabe um x. Dann gibt es $k-1$ Elemente, die kleiner sind als x und $n-k$ Elemente, die größer sind.
5. Wenn $i = k$ ist, dann haben wir das gesuchte Element gefunden. Sonst suchen wir rekursiv in den kleineren bzw. größeren Elementen weiter (abhängig davon ob $i < k$ oder $i > k$ ist. Im zweiten Fall suchen wir nach dem $i-k$-t kleinsten Element).

Betrachtet man diesen Algorithmus, sieht man schnell, dass wir hier in $\mathcal{O}(n)$ sind, also für ein beliebiges k nur linear viel Zeit benötigen, um das k-te Element zu finden.

Laufzeitanalyse

Bei dem Select-Algorithmus fällt auf, dass wir in Schritt 4 bereits den Suchraum stark einschränken können. Wenn wir die Mediane der 5-er Gruppen sowie den Median der Mediane kennen, können wir über einige Elemente in A bereits etwas aussagen:

- Der Median der Mediane ist x.
- Alle Elemente, die größer als ein Median sind, der wiederum größer als x ist, sind auch größer als x.
- Alle Elemente, die kleiner als ein Median sind, der wiederum kleiner als x ist, sind auch kleiner als x.

Da wir außerdem wissen, dass mindestens die Hälfte der Mediane aus Schritt 2 größer oder gleich x sind, wissen wir auch, dass aus mindestens der Hälfte der $\lceil n/5 \rceil$ Gruppen 3 Elemente größer als x sind (einen Sonderfall stellen nur die Gruppe von x selbst sowie die potentiell unvollständige letzte Gruppe dar). Damit können wir bereits sagen, dass für die Anzahl der Elemente, die größer sind als x gilt:

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

Für die Anzahl der Elemente, die kleiner sind als x können wir dieselbe Aussage treffen.

Damit können wir folgern, dass Select in Schritt 5 im schlimmsten Fall rekursiv mit

$$n - \left(\frac{3n}{10} - 6\right) = \frac{7n}{10} + 6$$

Elementen aufgerufen wird (da wir ja die kleineren bzw. größeren Elemente verwerfen können).

Insgesamt ergeben sich damit folgende Laufzeiten:

- Schritt 1, 2 und 4 laufen jeweils in $\mathcal{O}(n)$.
- Schritt 3 läuft in $T(\lceil n/5 \rceil)$.
- Schritt 5 läuft in $T(7n/10 + 6)$.

Also ist

$$T(n) \leq \begin{cases} \Theta(1) & n < 140 \\ T(\lceil n/5 \rceil) + T(7n/10 + 6) + \mathcal{O}(n) & n \geq 140 \end{cases}$$

Bleibt zu klären, wo die 140 herkommt.

Dazu müssen wir mit der Substitutionsmethode die Laufzeit bestimmen. Die Master-Methode können wir hier nicht anwenden. Wir raten also die Laufzeit $\mathcal{O}(n)$. Dann ist $T(n) \leq cn$ für ein konstantes c. Wir erhalten:

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + an \\ &\overset{*}{\leq} cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) \end{aligned}$$

* Für diese Ungleichung schätzen wir ab: $\lceil n/5 \rceil \leq n/5 + 1$.

Damit $T(n) \leq cn$ ist, muss

$$-cn/10 + 7c + an \leq 0$$

sein, also

$$c \geq 10a \cdot \frac{n}{n - 70}$$

Dies ist nur für $n > 70$ der Fall. Welches $n > 70$ wir wählen, ist allerdings egal. Wir entscheiden uns daher für $n = 140$. Damit ist dann $T(n) = \mathcal{O}(n)$.

Asymptotisches Verhalten

Asymptotisch ist jede Laufzeit für eine konstante Eingabe ebenfalls konstant. Daher können wir wie oben eine beliebige **konstante** Grenze für n wählen, ab der die Laufzeit asymptotisch linear ist. So lange die Eingabe kleiner oder gleich einer Konstanten ist, ist auch die Laufzeit konstant.

4. Heaps und HEAPSORT

Die Datenstruktur Heap (*engl. Haufen*) ist ein lineares Array, welches wir als binären Baum interpretieren. Im Gegensatz zu allgemeinen binären Bäumen, die im nächsten Kapitel behandelt werden, können Heaps nur spezielle binäre Bäume repräsentieren, nämlich solche, die bis auf die unterste Ebene vollständig sind. Die unterste Ebene des Baumes kann unvollständig sein, muss aber von links nach rechts befüllt werden.

Dann kann der Baum folgendermaßen im Array repräsentiert werden:

- Die Wurzel des Baumes ist an der Stelle $A[1]$.
- Für einen Knoten i ist $\mathit{Parent}(i) = \lfloor i/2 \rfloor$ der Vaterknoten.
- Für einen Knoten i sind $\mathit{Left}(i) = 2i$ und $\mathit{Right}(i) = 2i + 1$ die Indizes der Kinderknoten.

Wir definieren außerdem

- length(A) als Anzahl der Elemente im Array A.
- heap-size(A) als Anzahl der Elemente im Heap A.
- Die **Höhe** (*height*) eines Knotens im Baum als die Anzahl der Kanten auf dem längsten direkten Weg zu einem Blatt von dem Knoten aus.

Heaps sind in der Informatik besonders interessant, da alle Rechenoperationen auf ihnen **sehr** effizient durch Bitshift-Operationen ausgeführt werden können. Für allgemeine Bäume müssen dazu erst Referenzen in Speicher aufgelöst werden, was potentiell deutlich länger dauert (vgl. Kapitel 4).

Arten von Heaps

Wir unterscheiden zwei Arten von Heaps: **Min-Heaps** und **Max-Heaps**. Diese sind folgendermaßen definiert:

Max-Heap

$$\forall i : A[\mathit{Parent}(i)] \geq A[i]$$

Beim max-heap steht damit das größte Element an der Wurzel des Heap-Baumes.

Min-Heap

$$\forall i : A[Parent(i)] \leq A[i]$$

Beim min-heap steht also das kleinste Element an der Wurzel des Heap-Baumes.

Dies sind die einzigen Beschränkungen für min- und max-heaps. Daraus geht hervor, dass eine Menge von Werten (Zahlen) mehrere gültige Repräsentationen als Heap haben kann (sowohl als min-heap als auch als max-heap). Einzig der Wert der Wurzel ist eindeutig bestimmt. Insbesondere gibt es keine definierte Ordnung zwischen Geschwisterknoten, sodass wir z.B. beim min-heap nicht sicher sagen können, welcher Knoten das größte Element beinhaltet. Wir können nur sagen, dass es **irgendein** Blatt sein muss.

Es gibt drei wichtige Basisprozeduren, die auf einem Heap arbeiten. Wir werden uns hier nur die Prozeduren für den max-heap ansehen, allerdings funktionieren die Operationen für den min-heap ganz analog.

- MAX-HEAPIFY: Diese Prozedur sorgt dafür, dass der (potentiell ungültige) Heap nach Einfügen eines neuen Elementes wieder zu einem gültigen Heap umgebaut wird.
- BUILD-MAX-HEAP: Erstellt einen max-heap aus unsortierten Daten.
- HEAP-EXTRACT-MAX: Entfernt das größte Element aus dem Heap und sorgt dafür, dass danach wieder ein gültiger Heap entsteht.

MAX-HEAPIFY

MAX-HEAPIFY erhält ein Array A und einen Index i. Wir gehen davon aus, dass die Teilbäume an $Left(i)$ und $Right(i)$ bereits den Heap-Kriterien entsprechen.

Idee

MAX-HEAPIFY wird aufgerufen, wenn das Element an der Stelle i geändert/hinzugefügt wurde. Das Problem, was hier entstehen könnte ist, dass das neue Element zu klein ist und eigentlich weiter unten im Heap stehen müsste. Die Idee ist nun, dass MAX-HEAPIFY das neue Element so lange in Richtung Blätter des Baumes bewegt, bis die Heap-Eigenschaften erfüllt sind.

Pseudocode

```
MAX-HEAPIFY(A, i)
 1  ℓ = Left(i)
 2  r = Right(i)
 3  if l ≤ heap-size(A)  &  A[ℓ] > A[i]
 4        largest = ℓ
 5  else
 6        largest = i
 7  if r ≤ heap-size(A)  &  A[r] > A[largest]
 8        largest = r
 9  if largest ≠ i
10        exchange A[i] ↔ A[largest]
11        MAX-HEAPIFY(A, largest)
```

- In den Zeilen 1-8 suchen wir das größte der Elemente $A[i]$, $A[l]$ und $A[r]$.
- In Zeile 9 prüfen wir dann, ob eine Verletzung der Heap-Bedingungen vorliegt.
- Wenn das nicht so ist, dann sind wir hier schon fertig.
- Sonst bewegen wir das Element $A[i]$ eine Position nach unten (indem wir es mit den größeren der beiden Elemente an der Wurzel der linken und rechten Teilbäume vertauschen) und rufen MAX-HEAPIFY rekursiv auf, um eventuell den Teilbaum zu *reparieren*, den wir durch das Vertauschen zerstört haben.

Laufzeit

Die Laufzeit von MAX-HEAPIFY setzt sich zusammen aus:

1. Dem Finden des größten Elements sowie dem Vertauschen (beides in $\Theta(1)$).
2. Die Zeit, die die Ausführung von MAX-HEAPIFY auf dem linken oder rechten Teilbaum braucht.

Für 2. stellen wir fest, dass ein Teilbaum ungefähr $\frac{n-1}{2}$ Knoten hat, wenn der Baum vollständig ist. Da der Baum eines Heaps nur fast vollständig ist, ist die Größe eines Teilbaums aber immer noch höchstens $\lceil 2n/3 \rceil$. Das passiert genau dann, wenn das unterste Level des Baumes zur Hälfte gefüllt ist.

Damit ergibt sich für die Laufzeit:

$$T(n) \leq T(2n/3) + \Theta(1)$$

Hier können wir Fall 2 der Master-Methode anwenden und erhalten sofort

$$T(n) = \Theta(\log n)$$

Das kann man sich intuitiv auch erklären, wenn man merkt, dass das neue Element im schlimmsten Fall bis zu einem Blatt wandern muss, allerdings auf genau einem Pfad im Baum. Da die Höhe eines fast vollständigen binären Baumes etwa $\log n$ ist, ist auch die Laufzeit in $\Theta(\log n)$.

Build-Max-Heap

Diese Prozedur soll aus unsortierten Daten einen kompletten Heap erstellen.

Idee

Man kann relativ einfach zeigen, dass die Elemente $A[\lfloor n/2 \rfloor + 1], A[\lfloor n/2 \rfloor + 2], \ldots, A[n]$ in einem Heap die Blätter sind. Den Beweis sparen wir uns hier.

Von hier aus ist die Idee denkbar einfach: Als erstes definieren wir die letzten $\lfloor n/2 \rfloor$ Elemente unserer Eingabe als Blätter des Heaps. Dann benutzen wir Max-Heapify und fügen einfach ein unsortiertes Element nach dem anderen in den Heap ein.

Pseudocode

```
BUILD-MAX-HEAP(A)
1  heap-size(A) = length(A)
2  for i = ⌊length(A)/2⌋ downto 1
3          MAX-HEAPIFY(A, i)
```

Korrektheit

Satz 4.1. *Die Prozedur* Build-Max-Heap *erstellt einen korrekten Max-Heap.*

Beweis. Wir beweisen die Korrektheit von Build-Max-Heap mit einer Schleifeninvariante: Wir wollen zeigen, dass zu Beginn jeder Iteration alle Knoten $i+1, i+2, \ldots n$ die Wurzel eines gültigen Heaps sind.

Initialisierung Zu Begin ist $i = \lfloor n/2 \ \rfloor$. Dass alle folgenden Knoten Blätter im Heap (und damit selbst ein korrekter Heap) sind, haben wir bereits unter Idee festgestellt.

Aufrechterhaltung Für ein i haben alle Kinder von i einen höheren Index und befinden sich demnach im Array rechts von i. Diese sind nach Annahme bereits ein korrekter Heap. Also können wir problemlos Max-Heapify$(A,\ i)$ aufrufen. Dass dies korrekt funktioniert wissen wir bereits. Da wir außerdem i dekrementieren, stellen wir die Invariante für die nächste Iteration her.

Terminierung Offensichtlich wird i nur dekrementiert und landet irgendwann bei 1.

□

Laufzeit

Der intuitive Ansatz ist, festzustellen, dass wir für $n/2$ Elemente Max-Heapify aufrufen, welches in $\mathcal{O}(\log n)$ ist. Also sind wir insgesamt in $\mathcal{O}(n \log n)$. Diese Aussage stimmt auch.

Allerdings können wir eine bessere obere Schranke finden. Dafür stellen wir fest:

1. Die Laufzeit von Max-Heapify hängt von der Höhe des Knotens ab.
2. Ein n-elementiger Heap hat die Höhe $\lfloor \log n \rfloor$.
3. Es gibt höchstens $\left\lceil n/2^{h+1} \right\rceil$ Knoten für jede Höhe h.
4. Wir habe also viele Knoten mit geringer Höhe und wenige Knoten mit großer Höhe
5. Die Laufzeit von Max-Heapify bei einem Knoten der Höhe h ist $\mathcal{O}(h)$.

Dann können wir für Build-Max-Heap sehen, dass

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot \mathcal{O}(h) \leq \mathcal{O}\left(n \cdot \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$$

ist. Es gilt nun[1]:

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

Für unsere Formel ist $x = \frac{1}{2}$. Also setzen wir ein:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{\frac{1}{2}}{\left(1 - \frac{1}{2}\right)^2} = 2$$

und erhalten

$$T(n) \leq \mathcal{O}\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) \leq \mathcal{O}\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = \mathcal{O}(n)$$

Also ist die Laufzeit von Build-Max-Heap in $\mathcal{O}(n)$.

[1] Dies geht aus der geometrischen Reihe hervor. Für die Herleitung siehe https://de.wikipedia.org/wiki/Geometrische_Reihe#Herleitungen.

Heap-Extract-Max

Idee

Die Idee ist relativ simpel: Wir entfernen das Element an der Wurzel des max-heaps und fügen dann das letzte Element des Heap-Arrays an dessen Stelle ein. Zuletzt benutzen wir Max-Heapify, um die Struktur im Heap wiederherzustellen.

Laufzeit

Das Entfernen und Ersetzen des ersten Elements benötigt eine konstante Zeit. Max-Heapify braucht $\mathcal{O}(\log n)$. Also braucht auch Heap-Extract-Max auch $\mathcal{O}(\log n)$. Eine ausführliche Analyse sparen wir uns hier.

Heapsort

Idee

Heapsort ist ein Sortieralgorithmus, der auf der Datenstruktur Heap basiert. Die Idee ist folgende:

1. Wir bauen aus einem Array einen Heap mit Build-Max-Heap.
2. Wir extrahieren das größte Element (die Wurzel) und tauschen es mit dem letzten Element des Arrays.
3. Jetzt haben wird das kleinste bis zum zweitgrößten Element im Array.
4. Möglicherweise erfüllt der verbleibende Baum nicht die Heap-Anforderungen. Also bauen wir den Heap mit Max-Heapify wieder auf.
5. Wiederhole den Prozess bei Schritt 2.

Pseudocode

```
Heapsort(A)
1  Build-Max-Heap(A)
2  for i = length(A) downto 2
3          exchange A[1] ↔ A[i]
4          heap-size(A) = heap-size(A) − 1
5          Max-Heapify(A, 1)
```

Laufzeit

Die Laufzeit kann man quasi sofort ablesen: BUILD-MAX-HEAP braucht $\mathcal{O}(n)$, danach machen wir $\mathcal{O}(n)$ Schleifendurchläufe, wobei jeder davon durch das MAX-HEAPIFY in $\mathcal{O}(\log n)$ liegt. Damit ergibt sich eine Gesamtlaufzeit von $\mathcal{O}(n \log n)$.

Wie schon bei MERGE-SORT hat HEAPSORT eine worst-case Laufzeit von $\mathcal{O}(n \log n)$. Allerdings sind die Konstanten bei Quicksort weiterhin günstiger, sodass die meisten Anwendungsfälle in der echten Welt durch Quicksort besser lösbar sind.

5. Binäre Suchbäume

Binäre Suchbäume sind eine Datenstruktur, die für viele verschiedene Operationen effektiv benutzt werden kann:

- Suchen
- Minima finden
- Maxima finden
- Objekte einfügen
- Objekte entfernen
- ...

Binäre Suchbäume (*engl. binary search tree, BST*) können dadurch sehr vielseitig eingesetzt werden, zum Beispiel als Warteschlange oder als Map. Das schöne an binären Suchbäumen ist, dass viele Operationen abhängig von der Höhe des Baumes sind. Für einen vollständigen Baum haben wir daher potentiell sehr schnelle Vorgänge (logarithmisch zu n). Allerdings gilt das nicht für jeden BST, da diese auch degenerieren können.

Wir werden in diesem Abschnitt sehen, dass das Suchen in einem BST für vollständige Bäume in $\Theta(\log n)$ liegt und in $\Theta(n)$ wenn der Baum aus einem einzigen Pfad besteht. Für zufällige Bäume ist die erwartete Zeit in $\Theta(\log n)$.

Wir werden im Folgenden einige Algorithmen kennen lernen und analysieren, die mit binären Suchbäumen arbeiten.

Bäume haben zwei für uns relevante Kenngrößen: Die Anzahl an Knoten und die Höhe. Erstere wird normalerweise mit n bezeichnet, letztere mit h. Für vollständige Bäume gilt $h = \log_2(n+1)$, bzw. $n = 2^h - 1$, für degenerierte Bäume gilt im schlimmsten Fall $h = n$.

Definitionen

Wir repräsentieren binäre Bäume durch eine verkettete Datenstruktur. Da wir beliebige Bäume darstellen wollen, können wir hier keine Arrays mehr benutzen, ohne sehr ineffizient zu sein. Wir repräsentieren stattdessen jeden Knoten im Baum durch ein Objekt mit den folgenden Feldern:

- Ein Schlüssel (*key*)

- Beliebige Daten, die mit dem Schlüssel assoziiert sind.
- Ein Pointer zum linken Kind (*left*)
- Ein Pointer zum rechten Kind (*right*)
- Ein Pointer zum Elternknoten (*parent*)

Jeder der Pointer kann NULL sein, wenn es keinen entsprechenden Wert gibt. Die Wurzel des Baumes ist der einzige Knoten, für den *parent* = NULL ist. Der *key* kann nicht NULL sein. Alle *key*s müssel durch eine totale Ordnung sortierbar sein.

Damit ein binärer Baum nun ein binärer Suchbaum ist, müssen für alle Knoten x im Baum folgende Eigenschaften erfüllt sein:

- Für alle Knoten y im **linken** Unterbaum von x gilt: $key[y] \leq key[x]$.
- Für alle Knoten y im **rechten** Unterbaum von x gilt: $key[y] \geq key[x]$.

Heaps und binäre Suchbäume

Ein Heap erfüllt die Eigenschaften eines binären Suchbaumes nicht.

INORDER-TREE-WALK

Unsere erste Prozedur, die mit binären Suchbäumen arbeitet, ist INORDER-TREE-WALK. Wir wollen hier alle Knoten im Baum genau ein Mal besuchen. In unserem Fall heißt dies, dass wir jeden Knoten genau ein Mal ausgeben (print) möchten, aber natürlich kann dieselbe Prozedur auch für andere Operationen benutzt werden.

Idee

Wir definieren eine rekursive Funktion, die alle Knoten des Baumes mit der Wurzel x besucht. Dazu müssen wir

1. den linken Unterbaum von x rekursiv besuchen.
2. x selbst besuchen (also in diesem Fall ausgeben).
3. den rechten Unterbaum von x rekursiv besuchen.

Pseudocode

```
INORDER-TREE-WALK(x)
1  if x ≠ NULL
2        INORDER-TREE-WALK(left[x])
3        print key[x]
4        INORDER-TREE-WALK(right[x])
```

Eine bemerkenswerte Eigenschaft von binären Suchbäumen ist, dass die INORDER-TREE-WALK-Prozedur alle Schlüssel in sortierter Reihenfolge ausgibt.

Laufzeit

Satz 5.1. *Wenn x die Wurzel eines binären Suchbaumes mit n Knoten ist, dann braucht* INORDER-TREE-WALK *eine Laufzeit von* $\Theta(n)$.

Beweis. Zuerst stellen wir fest, dass sich die Laufzeit von INORDER-TREE-WALK durch folgende Rekurrenz beschreiben lässt:

$$T(n) = T(k) + T(n-k-1) + d$$

Dabei ist k die Größe des linken (rechten) Unterbaumes und d eine Konstante. $n-k-1$ ist die Größe des anderen Unterbaumes und ergibt sich daraus, dass wir insgesamt genau n Knoten (inklusive der Wurzel) haben.

Sei $T(n)$ die Laufzeit von INORDER-TREE-WALK. Wir beweisen die Annahme induktiv:

Induktionsanfang Offensichtlich ist $T(1) = \Theta(1) = \Theta(n)$, da in diesem Fall gar kein rekursiver Aufruf stattfindet.

Induktionsannahme Angenommen es gilt

$$T(m) = (c+d)m + c$$

für alle $m < n$.

Induktionsschritt : Wir wollen nun zeigen, dass $T(n) = \Theta(n)$ ist:

$$T(n) = T(k) + T(n-k-1) + d$$

Wir wissen nun, dass sowohl k als auch $n-k-1$ kleiner als n sind. Also können wir unsere Induktionsannahme anwenden:

$$\begin{aligned}
T(n) &= T(k) + T(n-k-1) + d \\
&= [(c+d)\cdot k + c] + [(c+d)(n-k-1) + c] + d \\
&= (c+d)k + c + (c+d)n - (c+d)k - (c+d) + c + d \\
&= (c+d)n + c - (c+d) + c + d \\
&= (c+d)n + c
\end{aligned}$$

Das zeigt den Induktionsschritt. Damit gilt

$$T(n) = \Theta(n)$$

□

Tree-Search

Als nächstes wollen wir nach einem gegebenen Schlüssel k suchen. Wenn der Schlüssel im BST vorhanden ist, soll eine Referenz zum entsprechenden Knoten zurückgegeben werden, ansonsten NULL.

Idee

- Wir fangen bei der Wurzel des Baumes an.
- Wir folgen einem bestimmten Pfad in Richtung der Blätter.
- Für jeden Knoten x auf dem Pfad, vergleichen wir $key[x]$ mit k.
- Wenn $key[x] = k$ ist, sind wir fertig.
- Wenn $k < key[x]$, durchsuche den linken Unterbaum rekursiv.
- Wenn $k > key[x]$, durchsuche den rechten Unterbaum rekursiv.

Psueodcode

```
TREE-SEARCH(x, k)
1  if x = NULL or k = key[x]
2      return x
3  elseif k < key[x]
4      return TREE-SEARCH(left[x], k)
5  else
6      return TREE-SEARCH(right[x], k)
```

Laufzeit

Die Laufzeit für eine Suche im Baum kann offensichtlich sehr schnell zu Ende sein. Im besten Fall ist eine Suche also in $\Theta(1)$ zu Ende.

Im schlechtesten Fall allerdings müssen wir einen Pfad im Baum von der Wurzel bis zu einem Blatt durchlaufen. Damit ist TREE-SEARCH durch $\mathcal{O}(h)$ beschränkt, wobei h die Höhe des BST ist.

TREE-MINIMUM und TREE-MAXIMUM

Wir wollen den größten bzw. kleinsten Knoten in einem binären Suchbaum finden.

Idee

Bei einem BST muss für alle Knoten y im linken Unterbaum eines Knotens x gelten, dass $key[y] \leq key[x]$ ist. Also können wir, ausgehend von der Wurzel des BST, immer den *left*-Referenzen folgen, bis $left =$ NULL ist. Der Knoten, bei dem wir dann ankommen ist nicht notwendigerweise ein Blatt, allerdings gibt es definitiv keine kleineren Knoten mehr. Also ist das unser Minimum.

Um das Maximum zu finden, gehen wir analog vor und folgen den *right*-Referenzen, bis es keine mehr gibt.

Pseudocode

```
TREE-MINIMUM(x)
1  while left[x] ≠ NULL
2       x = left[x]
3  return x
```

```
TREE-MAXIMUM(x)
1  while right[x] ≠ NULL
2       x = right[x]
3  return x
```

Bemerke

TREE-MINIMUM und TREE-MAXIMUM sind iterative Algorithmen.

Laufzeit

So wie wir die Algorithmen konzipiert haben, bewegen wir uns im Baum in jedem Schleifendurchlauf ein Level abwärts. Das können wir höchstens h Mal machen (wobei h die Höhe des Baumes ist). Also sind beide Algorithmen in $\mathcal{O}(h)$.

Im besten Fall stehen Minimum bzw. Maximum genau an der Wurzel. Dann sind wir in $\Omega(1)$.

Tree-Successor

Wir suchen den Knoten, der in einer sortierten Liste der Nachfolger des Knoten x wäre. Anders gesagt suchen wir den Knoten, der bei Inorder-Tree-Walk nach x ausgegeben wird.

Idee

- Wenn der rechte Unterbaum von x nicht leer ist, dann ist der Nachfolger von x das Minimum des rechten Unterbaumes.
- Falls der rechte Unterbaum leer ist, dann ist der Nachfolger von x der kleinste Vorfahre von x, dessen linkes Kind ebenfalls ein Vorfahre von x ist.

Den zweiten Punkt kann man sich so vorstellen: Wir laufen im Baum aufwärts. Dabei haben wir immer zwei Möglichkeiten: Nach rechts oben oder nach links oben laufen. Wenn wir nach links oben laufen, dann kommen wir aus dem Unterbaum, der nur größere Schlüssel enthält. Wenn wir aber nach rechts oben laufen, dann kommen wir aus dem Unterbaum, der nur kleinere Schlüssel enthält. Dann haben wir den Nachfolger gefunden, da dieser ja größer sein muss[1].

Wir wollen nun exemplarisch die Korrektheit dieser Beobachtung beweisen:

Beweis. Angenommen es gibt in dem Baum von x einen Nachfolger von x namens y.

Fall 1 x hat einen rechten Unterbaum. Dann enthält dieser nur Elemente mit größeren (oder gleichen) Schlüsseln. Im Baum gibt es außerdem keinen anderen Teilbaum, der den Nachfolger von x enthalten könnte. Die einzigen Teilbäume, die dafür in Frage kämen, wären rechte Unterbäume von einem Vorfahren von x. Da x dann aber im linken Unterbaum dieses Vorfahren wäre und selbst einen rechten Unterbaum hat, müssen die Elemente darin zwar größer als x aber kleiner als der entsprechende Vorfahre sein. Also

[1] Eigentlich kann der so gefundene Knoten auch gleichgroß sein. Das macht aber nichts, da Inorder-Tree-Walk in dem Fall trotzdem zuerst die Unterbäume berücksichtigt und es sich dann trotzdem um einen Nachfolger handelt.

ist in diesem Fall der Nachfolger von x im rechten Unterbaum von x und wird auch vom Algorithmus gefunden.

Fall 2 x hat keinen rechten Unterbaum. Dann wandern wir im Baum aufwärts. Wenn wir bei einem Schritt aufwärts festellen, dass x im rechten Unterbaum des Vorfahren ist, bedeutet das, dass der Schlüssel von x größer oder gleich dem des Vorfahren sein muss und es sich damit nicht um einen Nachfolger von x handeln kann. Stellen wir aber fest, dass sich x im linken Unterbaum eines Vorfahren befunden hat, dann ist der Schlüssel des Vorfahren größer oder gleich dem vom x. Damit haben wir einen Kandidaten für unser y gefunden. Es kommen nun auch keine anderen Kandidaten in Frage, da alle Elemente im rechten Unterbaum des Vorfahren noch größer sind. Für die Vorfahren des Vorfahren können wir analog zu oben argumentieren, dass diese auch zu groß bzw. zu klein sind.

Gibt es keinen Vorfahren von x, bei dem x im linken Unterbaum ist, handelt es sich bei x um das Maximum des Baumes. □

Pseudocode

```
TREE-SUCCESSOR(x)
1  if right[x] ≠ NULL
2        return TREE-MINIMUM(right[x])
3  y = parent[x]
4  while y ≠ NULL & x = right[y]
5        x = y
6        y = parent[y]
7  return y
```

Die Prozedur TREE-PREDECESSOR ist analog zu TREE-SUCCESSOR.

Laufzeit

Von TREE-MINIMUM wissen wir bereits, dass die Laufzeit in $\mathcal{O}(h)$ ist. Da wir in der Schleife (die Fall 2 im obigen Beweis abdeckt) im Baum nur aufwärts wandern, ist auch diese in $\mathcal{O}(h)$. Also ist auch der ganze Algorithmus in $\mathcal{O}(h)$.

TREE-INSERT

Wir wollen nun auch in der Lage sein, neue Objekte in den BST einzufügen, ohne die BST-Eigenschaften zu verletzen.

Idee

Wir fügen einen neuen Knoten immer als Blatt im Baum hinzu. Dazu gehen wir wie bei TREE-SEARCH vor. So bekommen wir direkt die Position, an der der neue Schlüssel eingefügt werden muss.

Pseudocode

```
TREE-INSERT(T, z)
 1  y = NULL, x = root[T]
 2  while x ≠ NULL
 3      y = x
 4      if key[z] < key[x]
 5          x = left[x]
 6      else
 7          x = right[x]
 8  parent[z] = y
 9  if y = NULL
10      root[T] = z // T was empty
11  elseif key[z] < key[y]
12      left[y] = z
13  else
14      right[y] = z
```

Laufzeit

Dieser Algorithmus geht wie TREE-SEARCH vor und führt dann einige Referenzänderungen (in konstanter Zeit) durch. Also ist die Laufzeit in $\mathcal{O}(h)$.

TREE-DELETE

Idee

Wenn wir ein Element z aus dem BST löschen, gibt es drei zu berücksichtigende Fälle:

1. z hat keine Kinder. Dann können wir z einfach löschen und bei $parent[z]$ den Zeiger auf z auf NULL setzen.

2. z hat ein Kind. Dann verknüpfen wir den Elternknoten von z direkt mit dem Kind.

3. z hat zwei Kinder. Dann suchen wir den Nachfolger y von z (welcher kein linkes Kind hat) und ersetzen den Schlüssel und die Daten von z mit denen von y. Dann löschen wir y.

Satz 5.2. *Der Nachfolger eines Knotens x hat kein linkes Kind, wenn wir uns oben in Fall 3 befinden.*

Beweis. Angenommen der Nachfolger (genannt y) von x hätte ein linkes Kind.

Fall 1 Der Nachfolger von x befindet sich im rechten Unterbaum von x. Dann gilt für alle Elemente in dem Unterbaum, dass sie größer (oder gleich) x sind. Wenn dort also y ein linkes Kind (nennen wir es z) hätte, würde $x \leq z \leq y$ gelten. Dann wäre y also nicht der Nachfolger. Das kann also nicht sein. In Fall 1 hat also y kein linkes Kind.

Fall 2 Der Nachfolger ist der erste Vorgänger von x, bei dem x im linken Unterbaum ist. Dann hat x keinen rechten Unterbaum (sonst wären wir in Fall 1). Allerdings hat dann x nur ein Kind und wir befinden uns nicht in Fall 3.

Damit ist klar, dass die Idee für den Algorithmus korrekt ist. □

Pseudocode

```
TREE-DELETE(T, z)
 1  if left[z] = NULL or right[z] = NULL
 2      y = z
 3  else
 4      y = TREE-SUCCESSOR(z)
 5  if left[y] ≠ NULL
 6      x = left[y]
 7  else
 8      x = right[y]
 9  if x ≠ NULL
10      parent[x] = parent[y]
11  if parent[y] = NULL
12      root[T] = x
13  elseif y = left[parent[y]]
14      left[parent[y]] = x
15  else
16      right[parent[y]] = x
17  if y = z
18      key[z] = key[y]
19      copy y's data into z
20  return y
```

Laufzeit

In Fall 1 und 2 ist die Laufzeit offensichtlich in $\Theta(1)$. In Fall 3 müssen wir den Nachfolger von z finden, daher ist die Laufzeit in $\mathcal{O}(h)$.

Insgesamt ist damit die Laufzeit von Tree-Delete in $\mathcal{O}(h)$.

6. Rot-Schwarz-Bäume

Motivation

Binäre Suchbäume sind eine vergleichsweise simple Datenstruktur, die viele Operationen sehr effizient ausführen kann. Die Laufzeiten der meisten Operationen sind dabei abhängig von der Höhe des Baumes.

Für eine gegebene Menge Schlüssel gibt es nun mehrere Möglichkeiten, diese in einen binären Baum einzusortieren. Eine davon ist immer, genau einen Pfad (und somit effektiv eine verkettete Liste) zu haben. Das ist aber normalerweise unerwünscht, da dann die Höhe des Baumes vergleichsweise groß ist. Stattdessen wollen wir eigentlich einen binären Suchbaum, der immer so balanciert wie möglich ist. Bei einem balancierten Suchbaum wächst die Höhe logarithmisch zur Anzahl der Knoten. Dasselbe gilt für die Laufzeit der für uns interessanten Algorithmen.

Offensichtlich sind die problematischen Operationen das Einfügen und Entfernen von Elementen in oder aus dem Baum. Rot-Schwarz-Bäume sind spezielle binäre Suchbäume, die gewährleisten, dass der Baum immer mehr oder weniger balanciert bleibt.

Definition

Wir erweitern die Definition von binären Suchbäumen so, dass jeder Knoten ein zusätzliches Bit erhält: Seine Farbe. Diese kann entweder rot oder schwarz sein. Dieses zusätzliche Bit wollen wir dazu verwenden, bestimmte Struktureigenschaften des Baumes zu repräsentieren, was am Ende dafür sorgen wird, dass wir immer einen relativ balancierten Baum erhalten.

Ein Unterschied zu normalen BSTs ist, dass man bei RBTs (Red-Black-Trees) normalerweise die NULL-Knoten mit modelliert. Dann sind alle Blätter des Baumes NULL-Knoten und die inneren Knoten des Baumes genau die Knoten, die unsere Werte repräsentieren und alle NULL-Knoten genau die äußeren Knoten.

Grafische Notation

Wenn man RBTs zeichnet, lässt man meistens die NULL-Knoten weg. Es ist wichtig zu verstehen, dass diese im Gegensatz zu BSTs trotzdem **als Knoten** vorhanden

sind. Bei BSTs gibt es zwar NULL-Referenzen, aber keine NULL-Knoten. Bei RBTs ist das anders.

Mit dieser Vorarbeit können wir nun endlich Rot-Schwarz-Bäume definieren:

Definition 6.1 (Rot-Schwarz-Baum)**.** *Ein gefärbter binärer Suchbaum ist genau dann ein Rot-Schwarz-Baum, wenn folgende Punkte erfüllt sind:*

1. *Jeder Knoten ist entweder Rot oder Schwarz.*
2. *Die Wurzel ist schwarz.*
3. *Jedes Blatt (*NULL*) ist schwarz.*
4. *Jeder rote Knoten hat* ***nur*** *schwarze Kinder.*
5. *Alle Pfade zu den Blätter, die von demselben Knoten ausgehen, haben dieselbe Anzahl schwarzer Knoten.*

Definition 6.2. *Die* ***Schwarz-Höhe*** *eines Knotens v in einem Rot-Schwarz-Baum ist die Anzahl schwarzer Knoten in einem beliebigen Pfad von v zu einem Blatt (exklusive v selbst). Wir schreiben* $\mathrm{bh}(v)$ *für die Schwarz-Höhe.*

Balanciertheit eines Rot-Schwarz-Baumes

Bevor wir Knoten in den RBT einfügen können, wollen wir beweisen, dass ein RBT auch tatsächlich einem beliebigen binären Suchbaum überlegen ist (vorausgesetzt, dass wir in angemessener Laufzeit einfügen können). Dazu beweisen wir folgendes Lemma:

Lemma 6.1. *Ein Rot-Schwarz-Baum mit n inneren Knoten hat höchstens die Höhe* $2\log(n+1)$*.*

Um dieses Lemma zu beweisen, zeigen wir zunächst das folgende:

Lemma 6.2. *Ein Rot-Schwarz-Teilbaum mit der Wurzel x enthält* ***mindestens*** $2^{\mathrm{bh}(x)}-1$ *innere Knoten.*

Beweis. Wir beweisen Lemma 6.2 durch Induktion:

Induktionsanfang Für einen Baum der Höhe 0 ist x ein Blatt (NULL). Dann enthält der Teilbaum mit der Wurzel x

$$2^{\mathrm{bh}(x)} - 1 = 2^0 - 1 = 1 - 1 = 0$$

innere Knoten.

Induktionsannahme Angenommen Lemma 6.2 gilt für alle Rot-Schwarz-Bäume bis zur Höhe h.

Induktionsschritt Sei x die Wurzel eines Baumes der Höhe $h+1$. Wir nehmen außerdem an, dass x zwei Kinder hat. Die Schwarz-Höhe eines Kindes ist dann

1. $\text{bh}(x)$, wenn das Kind rot ist.
2. $\text{bh}(x) - 1$, wenn das Kind schwarz ist.

Fall 1 Die Kinder von x sind beide rot. Dann ist die Schwarz-Höhe der Kinder genau gleich wie die Schwarz-Höhe von x. Da Unterbäume von x aber eine kleinere Höhe haben, können wir direkt die Induktionsannahme auf die Kinder von x anwenden. Dann erhalten wir für die Anzahl der inneren Knoten des Baumes mit der Wurzel x:

$$\text{Innere Knoten von } x \overset{I.A.}{\geq} 2 \cdot \left(2^{\text{bh}(x)} - 1\right) + 1 \geq 2^{\text{bh}(x)} - 1$$

Hier ist also das Lemma erfüllt.

Fall 2 Die Kinder von x sind beide schwarz. Wir können auch hier die Induktionsannahme auf die Kinder von x anwenden. Da aber hier die Kinder selbst schwarz sind, wissen wir außerdem, dass jeder der Teilbäume der Kinder mindestens $2^{\text{bh}(x)-1} - 1$ innere Knoten enthält. Daher ist die Anzahl innerer Knoten von x:

$$\text{Innere Knoten von } x \overset{I.A.}{\geq} 2 \cdot (2^{\text{bh}(x)-1} - 1) + 1 = 2^{\text{bh}(x)} - 2 + 1 = 2^{\text{bh}(x)} - 1$$

Fall 3 Die Kinder von x sind unterschiedlich gefärbt. Wieder können wir die Induktionsannahme auf die Kinder von x anwenden. Hierfür ist wichtig zu bemerken, dass die Schwarz-Höhe jedes (inneren) Knotens mindestens 1 ist:

$$\text{Innere Knoten von } x \overset{I.A.}{\geq} (2^{\text{bh}(x)-1} - 1) + (2^{\text{bh}(x)} - 1) + 1 = \underbrace{2^{\text{bh}(x)-1}}_{\geq 1} + 2^{bh(x)} - 1 \geq 2^{\text{bh}(x)} - 1$$

□

Nun können wir Lemma 6.1 beweisen:

Beweis. Sei h die Höhe des Rot-Schwarz-Baumes. Durch Eigenschaft 4 von RBTs wissen wir, dass mindestens die Hälfte der Knoten auf jedem Pfad von der Wurzel zu einem Blatt schwarz sein müssen (exklusive der Wurzel selbst). Also ist die Schwarz-Höhe der

Wurzel mindestens $h/2$. Nach Lemma 6.2 wissen wir außerdem, dass die Anzahl der inneren Knoten mindestens $2^{\text{bh(Wurzel)}} - 1$ ist. Also setzen wir ein:

$$\begin{aligned} n &\geq 2^{h/2} - 1 \\ \Leftrightarrow n + 1 &\geq 2^{h/2} \\ \Leftrightarrow \log(n+1) &\geq h/2 \\ \Leftrightarrow h &\leq 2\log(n+1) \end{aligned}$$

□

Korollar 6.1. *Da auch alle Formeln für normale binäre Suchbäume auch für Rot-Schwarz-Bäume gelten, wissen wir nun sofort, dass* TREE-SEARCH*,* TREE-MINIMUM*,* TREE-MAXIMUM*,* TREE-SUCCESSOR *und* TREE-PREDECESSOR *für RBTs immer in einer Laufzeit von $O(\log n)$ durchgeführt werden können.*

Rotationen

Bevor wir einen Algorithmus kennen lernen, mit dem wir Elemente in einen Rot-Schwarz-Baum einfügen können, ohne die Rot-Schwarz-Eigenschaften zu verletzen, müssen wir zunächst den Begriff der Rotation definieren. Wir unterscheiden zwischen Links- und Rechtsrotationen. Eine Rotation nur unter bestimmten Bedingungen durchgeführt werden:

- Eine Linksrotation um x kann nur durchgeführt werden, wenn das rechte Kind von x nicht NULL ist.
- Eine Rechtsrotation um x kann nur durchgeführt werden, wenn das linke Kind von x nicht NULL ist.

Eine Rotation sieht dann so aus:

```
LEFT-ROTATE(T, x)
 1  z = parent[x]
 2  y = right[x]
 3  right[x] = left[y]
 4  left[y] = x
 5  parent[x] = y
 6  parent[y] = z
 7  if z = NULL
 8        root[T] = y
 9  elseif right[z] = x
10        right[z] = y
11  else
12        left[z] = y
```

```
Right-Rotate(T, x)
 1  z = parent[x]
 2  y = left[x]
 3  left[x] = right[y]
 4  right[y] = x
 5  parent[x] = y
 6  parent[y] = z
 7  if z = NULL
 8        root[T] = y
 9  elseif right[z] = x
10        right[z] = y
11  else
12        left[z] = y
```

Wir rotieren also um ein Element x herum, sodass danach das linke bzw. rechte Kind von x an dessen Stelle im Baum steht, ohne dass der binäre Baum zerstört wird. Offensichtlich laufen beide Operationen in $\Theta(1)$ ab.

Im Folgenden werden wir die Rotation dazu benutzen, die Höhe des Baumes zu verändern, um diesen zu balancieren. Da wir versuchen wollen, einen möglichst schnellen Algorithmus zum Einfügen neuer Elemente zu erhalten, kommt uns sehr gelegen, dass wir das Rotieren in konstanter Zeit tun können.

RB-Tree-Insert

Jetzt haben wir alle Voraussetzungen kennen gelernt, um ein neues Element in einen exitierenden RBT einzufügen. Wir nennen den einzufügenden Knoten z.

Idee

1. Als erstes fügen wir z wie in einen normalen BST mit Tree-Insert ein (s. letzter Teil 5).
2. Dann färben wir z rot.
3. Jetzt rufen wir RB-Insert-Fixup auf, um die Eigenschaften des Rot-Schwarz-Baumes wiederherzustellen.

Bei 3. stellt sich die Frage, ob die Rot-Schwarz Eigenschaften überhaupt verletzt werden können. Zur Erinnerung: Die Eigenschaften sind

1. Jeder Knoten ist entweder rot oder schwarz.
2. Die Wurzel ist schwarz.

3. Jedes Blatt (NULL) ist schwarz.

4. Jeder rote Knoten hat **nur** schwarze Kinder.

5. Alle Pfade zu den Blätter, die von demselben Knoten ausgehen, haben dieselbe Anzahl schwarzer Knoten.

Wir stellen fest:

- 1. und 3. werden definitiv nicht verletzt.
- 5. wird auch nicht verletzt, da wir z rot gefärbt haben. Somit hat z keinen Einfluss auf die Schwarz-Höhe.
- Da z aber rot ist, könnten die Eigenschaften 2. und 4. verletzt sein und zwar genau dann, wenn z die Wurzel des Baumes ist (2. ist verletzt) oder wenn der Elternknoten von z rot ist (4. ist verletzt).

Pseudocode

```
RB-INSERT-FIXUP(T, z)
 1  while color[parent[z]] = RED
 2      if parent[z] = left[parent[parent[z]]]
 3          y = right[parent[parent[z]]]
 4          if color[y] = RED
 5              color[parent[z]] = BLACK
 6              color[y] = BLACK
 7              color[parent[parent[z]]] = RED
 8              z = parent[parent[z]]
 9          else
10              if z = right[parent[z]]
11                  z = parent[z]
12                  LEFT-ROTATE(T, z)
13              color[parent[z]] = BLACK
14              color[parent[parent[z]]] = RED
15              RIGHT-ROTATE(T, parent[parent[z]])
16      else
17          same as if-case with left and right exchanged
18  color[root[T]] = BLACK
```

Bemerke: Wenn der Elternknoten von z schwarz ist, überspringen wir die Schleife sofort. In dem Fall muss nichts weiter getan werden.

Korrektheit

Wir wollen die Korrektheit von RB-INSERT-FIXUP beweisen. Dazu benutzen wir die Technik der Schleifeninvariante (siehe Anhang A).

Satz 6.1. *Am Anfang jeder Iteration in* RB-INSERT-FIXUP *gilt:*

1. *Der Knoten z ist rot.*

2. *Wenn $parent[z]$ die Wurzel des Baumes ist, ist $parent[z]$ schwarz.*

3. *Es gibt höchstens eine Verletzung der Bedingungen für Rot-Schwarz-Bäume und zwar entweder 2. oder 4. (aus den gleichen Gründen wie oben).*

Beweis. Wir beweisen die Schleifeninvariante wie folgt

Initialisierung

Vor der ersten Iteration (bzw. vor dem Einfügen von z) hatten wir einen gültigen Rot-Schwarz-Baum vorliegen. Also gilt vor der ersten Iteration (nach dem Einfügen von z):

1. Wenn RB-INSERT-FIXUP aufgerufen wird, ist z der rote Knoten, der hinzugefügt wurde.

2. Wenn $parent[z]$ die Wurzel des Baumes ist, bleibt diese schwarz und wird nicht verändert.

3. Bedingung 1., 3. und 5. sind offensichtlich wahr. Bedingung 2 könnte verletzt sein, wenn z die neue Wurzel des Baumes ist. Dann ist aber auch z der einzige innere Knoten des ganzen Baumes. Da in dem Fall beide Kinder von z (nämlich beide NULL) schwarz sind, ist hier aber 4. nicht verletzt. Wenn Bedingung 4 verletzt ist, dann müssen z und $parent[z]$ rot sein. Da aber $parent[z]$ rot ist, kann das nicht die Wurzel des (vorher korrekten) RBTs sein. Also ist in diesem Fall 2. nicht verletzt.

Terminierung

Die Terminierung tritt ein, wenn $parent[z]$ schwarz ist. Dass dies auch tatsächlich passiert, wird sich in der Aufrechterhaltung zeigen. In diesem Fall kann Bedingung 4 für RBTs offensichtlich nicht verletzt sein. Durch Zeile 18 wird außerdem Bedingung 2 definitiv erfüllt. Das heißt am Ende sind tatsächlich alle Bedingungen für RBTs erfüllt. Wenn wir also die Aufrechterhaltung der Invariante zeigen können, haben wir die Korrektheit von RB-INERT-FIXUP bewiesen und sind fertig.

Aufrechterhaltung

Bei der Aufrechterhaltung müssen wir eigentlich 6 Fälle unterscheiden. Drei davon sind allerdings symmetrisch zu den anderen drei (siehe Zeile 17). Also reicht es, drei Fälle zu unterscheiden. Im Pseudocode sind dies die Zeilen 5-8, 11-12 und 13-15 (die letzten beiden Fälle sind kombiniert. Warum sehen wir später).

Um einfacher argumentieren zu können, führen wir ein paar Hilfsbegriffe ein:

- $\mathit{parent}[z]$ ist der *Vater* von z.
- $\mathit{parent}[\mathit{parent}[z]]$ ist der *Großvater* von z. Bemerke, dass dieser tatsächlich immer existiert. Der einzige Fall, in dem dies ein Problem wäre ist, wenn z die Wurzel des Baumes ist. Dann ist aber der Vater von z schwarz und wir betreten die Schleife nicht. Wir wissen außerdem, dass der Großvater von z immer schwarz ist, da der Vater von z rot ist, und Bedingung 4 für RBTs bereits zwischen z und dessen Vater verletzt ist.
- $\mathit{right}[\mathit{parent}[\mathit{parent}[z]]]$ ist der *Onkel* von z.

Fall 1 Der Onkel von z ist rot. Wir wissen durch die Schleifenbedingung, dass der Vater von z ebenfalls rot ist. Wir wissen außerdem, dass der Großvater von z schwarz ist. Also können wir problemlos den Vater und Onkel von z schwarz färben. Dadurch erhöht sich die Schwarz-Tiefe in den entsprechenden Teilbäumen allerdings um 1. Um dieses Problem zu lösen, färben wir den Großvater von z nun rot, um Bedingung 5 für RBTs zu erhalten.

Jetzt haben wir alle Verwandten von z relativ zu z richtig gefärbt. Allerdings könnte nun der Großvater von z (der jetzt rot ist), das Kind eines anderen roten Knotens sein. Um dieses Problem zu beheben, setzen wir $z = \mathit{parent}[\mathit{parent}[z]]$ und gehen in die nächste Iteration.

Jetzt müssen wir beweisen, dass dieses Vorgehen tatsächlich die Schleifeninvariante aufrecht erhält. Sei dazu z der aktuelle Knoten und z' der Großvater von z (unser neues z für die nächste Iteration).

1. Wir färben den Großvater von z rot, also ist z' am Anfang der nächsten Iteration rot.
2. In dieser Iteration ändert sich die Farbe vom Urgroßvater von z ($\mathit{parent}[z'] = \mathit{parent}[\mathit{parent}[\mathit{parent}[z]]]$) nicht. Wenn das die Wurzel des Baumes ist und vor der Iteration schwarz war, ist er das nun immer noch. Selbiges gilt für alle Vorfahren von $\mathit{parent}[z']$.
3. Wir wissen nun, dass in Fall 1 Bedingung 5. für RBTs erhalten wird. Es ist klar, dass auch 1. und 3. nicht verletzt werden.

   Wenn in z' der nächsten Iteration die Wurzel des Baumes ist, haben wir in Fall 1 die Verletzung von Bedingung 4. korrigiert. Da aber z' nun rot und die Wurzel ist,

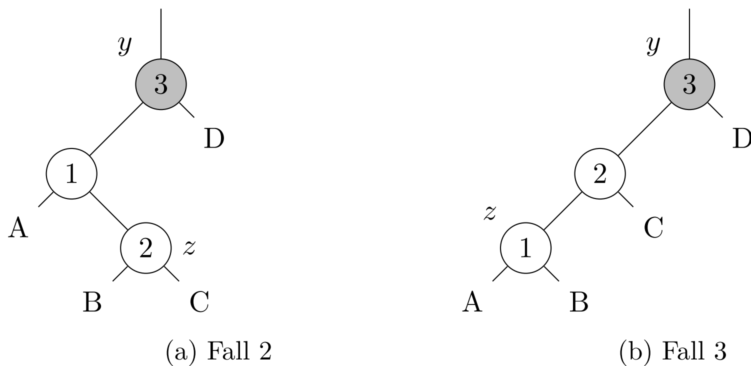
ist offenbar Bedingung 2. durch z' verletzt.

Wenn z' nicht die Wurzel des Baumes ist, dann wird Bedingung 2. auch nicht verletzt. Wir haben außerdem die Verletzung von Bedingung 4. korrigiert, die am Anfang des Schleifendurchgangs herrschte, indem wir z' rot gefärbt haben. Wenn nun der Vater von z' schwarz ist, gibt es keine Verletzungen der Bedingungen für RBTs mehr und wir sind fertig. Wenn aber der Vater von z' rot war, ist nun Bedingung 4. zwischen z' und dessen Vater verletzt.

Also gibt es am Ende von Fall 1 höchstens eine Verletzung der Bedingungen für RBTs.

Das zeigt die Schleifeninvariante für Fall 1.

Fall 2 Der Onkel von z ist schwarz und z ist das rechte Kind. Dieser Fall unterscheidet sich von Fall 3 nur dadurch, dass bei Fall 3 z das linke Kind ist. Durch eine einfache Linksrotation können wir die Seite des Kindes vertauschen (s. Abbildung). Da in diesem Fall z und $parent[z]$ beide rot sind, ändert sich durch die Rotation weder die Schwarz-Höhe noch wird dadurch Bedingung 5. für RBTs verletzt.



(a) Fall 2

(b) Fall 3

Abbildung 6.1.: Die Rotation von Fall 2 zu Fall 3

Fall 3 Der Onkel von z ist schwarz und z ist das linke Kind. In diesem Fall ändern wir nur die Farben von dem Vater und dem Großvater von z und rotieren wie es in der folgenden Abbildung dargestellt ist:

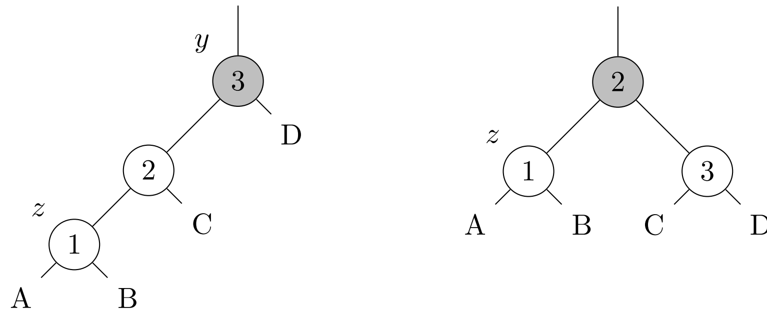


Abbildung 6.2.: Rotation in Fall 3

Hier wird offensichtlich Bedingung 5 für RBTs erfüllt. Durch diese Rotation haben wir außerdem erreicht, dass es keine zwei benachbarten roten Knoten mehr gibt. Das heißt, dass wir hier nach dem aktuellen Schleifendurchlauf fertig sind. Auch hier beweisen wir die Aufrechterhaltung der Invariante:

1. Fall 2 wird in Fall 3 umgewandelt. Danach ist z offensichtlich rot (siehe Abbildungen).
2. Fall 3 färbt den Vater von z schwarz. Sollte der Vater von z nun die Wurzel des Baumes sein, tritt kein Problem auf. Ansonsten war die Wurzel schon vorher schwarz.
3. Wie in Fall 1 sind die Bedingungen 1, 3 und 5 erfüllt. Da z in Fall 2 und 3 nicht die Wurzel sein kann, ist außerdem Bedingung 2 erfüllt. Fall 2 und 3 führen auch zu keiner neuen Verletzung von Bedingung 2, da der einzige Knoten, der rot gefärbt wird, durch die Rotation in Fall 3 das Kind eines schwarzen Kindes wird. Fall 2 und 3 korrigieren eine Verletzung von Bedingung 4, führen aber keine neue Verletzung ein (s. Abbildungen).

Das zeigt die Korrektheit von Fall 3 und damit auch von Fall 2. Da Fall 4, 5 und 6 völlig symmetrisch sind, beendet dies den Beweis. Damit ist unsere Schleifeninvariante korrekt. Da wir außerdem am Ende von RB-INSERT-FIXUP eine eventuell verbleibende Verletzung von Bedingung 2 korrigieren, und Bedingung 4 beim Verlassen der Schleife definitiv erfüllt ist, haben wir am Ende der Prozedur mit Sicherheit einen korrekten Rot-Schwarz-Baum. □

Laufzeit

Die Laufzeitanalyse für RB-TREE-INSERT ist sehr einfach: Wir wissen bereits, dass das Einfügen durch TREE-INSERT in $\mathcal{O}(\log n)$ liegt. Die RB-INSERT-FIXUP-Prozedur führt nur in Fall 1 tatsächlich mehrere Iterationen aus. In Fall 2 und 3 ist die Laufzeit in $\Theta(1)$. In Fall 1 wird unser z in jedem Iterationsschritt um zwei Plätze aufwärts bewegt. Im

schlimmsten Fall muss es bis zur Wurzel durchwandern. Das benötigt höchstens $\mathcal{O}(\log n)$ Schritte.

Damit liegt die Laufzeit für RB-INSERT-FIXUP und damit auch für RB-TREE-INSERT in $\mathcal{O}(\log n)$.

Ausblick

Rot-Schwarz-Bäume sind eine sehr effiziente und vergleichsweise einfach zu implementiere Variante von binären Suchbäumen. Allerdings gibt es noch effizientere Varianten wie zum Beispiel AVL-Bäume. Bei diesen ist der konstante Faktor, der durch die asymptotische Notation versteckt wird nochmal deutlich kleiner als bei Rot-Schwarz-Bäumen (1,44 im Vergleich zu 2). Also sind AVL-Bäume deutlich besser balanciert.

Allerdings werden RBTs trotzdem noch sehr oft verwendet. Dies liegt einerseits daran, dass sie einfacher zu implementieren sind und andererseits daran, dass AVL-Bäume absolut gesehen deutlich mehr Operationen für das Fixup brauchen (asymptotisch allerdings weiterhin $\mathcal{O}(\log n)$ viele). Das kann je nach Rechnerarchitektur und Implementierung deutlich länger dauern als bei RBTs (z.B. beim Schreiben auf einer Festplatte oder über ein Netzwerk). Aus diesem Grund werden für viele allgemeine Implementierungen von binären Suchbäumen gerne Rot-Schwarz-Bäume verwendet.

7. Hashtabellen

Motivation

Es gibt viele Situationen, in denen man verschiedene Daten als eine Einheit speichern möchte. Jede der Datenstrukturen, die wir bisher betrachtet haben (Arrays, Listen, Heaps, Binäre Suchbäume) hatte eigene Stärken und Schwächen. Hashtabellen (oft auch als *Dictionary* bezeichnet) sind nun eine weitere Datenstruktur, die eine Sammlung von Daten darstellt.

Allerdings eignet sich eine Hashtabelle nicht für jeden Anwendungszweck gleich gut. So lange wir aber nur eine Menge von Daten brauchen, in die wir **einfügen**, **löschen** und in der wir **suchen** können, sind Hashtabellen sehr effizient[1].

Definition

Idee

Eine Hashtabelle kann man sich als eine Art *intelligentes Array* vorstellen, bei dem bestimmte Objekte an bestimmte Plätze sortiert werden, um diese schnell wiederzufinden. Dazu assoziieren wir alle Daten mit einem ganzzahligen Schlüssel und berechnen aus dieser Zahl den Index im Array, an den das Objekt gehört.

Der naive Ansatz für so eine Datenstruktur wäre einfach ein Array zu nehmen, dass genau so viele Felder hat, wie wir mögliche Schlüssel haben. Dieser Ansatz wäre zwar theoretisch machbar, allerdings ist es normalerweise nicht praktikabel ein so großes Array zu erzeugen, insbesondere wenn es nur wenige Daten gibt, die tatsächlich in die Tabelle eingefügt werden. Wir wollen also ein Array haben, das kleiner ist, als die Menge der möglichen Schlüssel. Dann brauchen wir aber auch eine Funktion (Hashfunktion), die für einen Schlüssel den entsprechenden Index im Array berechnet.

Da die Hashfunktion im allgemeinen nicht injektiv sein muss, kann es passieren, dass mehrere Objekte an denselben Platz im Array sortiert werden. Daher soll jeder Platz nicht nur ein Element, sondern eine beliebig lange Liste von Elementen aufnehmen können.

[1] Es gibt viele verschiedene Arten von Hashtabellen (bzw. Hash-Funktionen). Die Qualität der Hashfunktion bestimmt maßgeblich die Qualität der Hashtabelle.

Formale Definition

Gegeben sei ein endliches **Universum** $U \subseteq \mathbb{N}$ von Schlüsseln. Jeder Schlüssel ist eine natürliche Zahl:

$$U = \{0, 1, \ldots, u - 1\}$$

u ist meistens sehr groß. Für 32-bit Zahlen wäre zum Beispiel $u > 4.000.000$.

Definition 7.1 (Element). *Ein Element ist ein Wertepaar $(key, data)$, wobei $key \in U$ ist. data ist beliebig. Wir nehmen an, dass keine zwei verschiedenen Elemente den gleichen Schlüssel haben. Wir bezeichnen für ein Element x den Schlüssel von x mit $key[x]$.*

Definition 7.2 (Hashtabelle). *Eine Hashtabelle ist dann eine Datenstruktur zum Speichern von Elementen. Wir fordern dabei, dass jedes Element höchstens ein Mal in der gesamten Hashtabelle vorkommt. Eine solche Tabelle unterstützt die folgenden Operationen:*

Einfügen *Ein neues Element wird irgendwo in die Hashtabelle eingefügt.*

Suchen *Sucht nach den Daten für einen gegebenen Schlüssel und gibt entweder die entsprechenden Daten oder einen Fehler zurück.*

Löschen *Entfernt ein gegebenes Element aus der Hashtabelle.*

Genauer ist eine Hashtabelle ein Array mit m Feldern. Jedes der Felder beinhaltet eine (zunächst leere) doppelt verkettete Liste von Elementen.

Definition 7.3. *Nun definieren wir noch den Begriff der Hashfunktion. Eine **Hashfunktion** ist eine Abbildung*

$$h : U \to \{0, 1, \ldots, m - 1\}$$

*für ein $m \leq |U|$. Eine Hashfunktion bildet also einen Schlüssel auf einen Index in der Hashtabelle ab. Man sagt, dass $h(k)$ der **Hashwert** des Schlüssels k ist. Im Normalfall nehmen wir an, dass $h(k)$ in konstanter Zeit berechnet werden kann.*

Da h nicht notwendigerweise injektiv ist, kann es sein, dass mehrere Schlüssel auf denselben Index abgebildet werden. Dies bezeichnen wir als **Kollision**. Um Kollisionen zu lösen, benutzen wir verkettete Listen[2]. So können wir mehrere Elemente in demselben Index der Hashtabelle speichern.

Zuletzt definieren wir noch den **Füllgrad** einer Hashtabelle (engl.: *load factor*) als

$$\alpha = \frac{n}{m}$$

und bemerken zunächst nur, dass der Füllgrad weder eine natürliche Zahl noch größer als 1 sein muss. Der Füllgrad entspricht genau der durchschnittlichen Länge der Listen in der Hashtabelle.

[2] Es gibt auch andere Methoden, mit Kollisionen umzugehen. Siehe Wikipedia.

Hashtabellen in Java

Aus SE-1 und SE-2 ist unter Umständen das Konzept von Hashtabellen bereits bekannt. Man muss allerdings ein bisschen mit den Begrifflichkeiten aufpassen.

In Java sind Hashtabellen mit der Klasse `java.util.HashMap` implementiert. Diese Klasse unterstützt alle Operationen, die wir uns auf Hashtabellen ansehen werden. Beim Einfügen in eine `HashMap` übergibt man zwei Objekte: Den Schlüssel und den Wert. In unserem Setting entspricht der Wert dem *data*-Feld eines Elements.

Allerdings ist der *key* nicht das Schlüsselobjekt, sondern der `hashCode()` des Schlüsselobjekts. Insbesondere ist der `hashCode()` **nicht der Hashwert**. Der Hashwert wird erst intern in der `HashMap`-Klasse durch die Hashfunktion berechnet (Die `hashCode()`-Funktion ist auch nicht die Hashfunktion).

Bei Java `HashMap`s gilt außerdem nicht die Einschränkung, dass die Schlüssel paarweise verschieden sein müssen. Es gilt allerdings die Einschränkung, dass die Schlüsselobjekte für verschiedene Elemente paarweise verschieden sein müssen.

Operationen auf Hashtabellen

Wir definieren drei Operationen auf Hashtabellen:

Chained-Hash-Insert(T, x)
1 insert x at head of list $T[\mathrm{h}(key[x])]$

Angenommen, dass x noch nicht in der Hashtabelle enthalten ist, ist die Laufzeit offensichtlich in $\mathcal{O}(1)$ (wir müssen nur ein paar Referenzen umbiegen, um ein neues Element am Anfang einer verketteten Liste einzufügen).

Wenn wir nicht wissen, ob x bereits in der Tabelle enthalten ist, müssen wir zuerst danach suchen.

Chained-Hash-Search(T, k)
1 search for element with key k in list $T[\mathrm{h}(k)]$

Im schlimmsten Fall müssen wir die ganze verkettete Liste im Index $h(k)$ durchsuchen. Das ist offenbar proportional zur Listenlänge. Im Folgenden werden wir uns dies noch genauer ansehen.

Chained-Hash-Delete(T, x)
1 delete x from list $T[\mathrm{h}(key[x])]$

Achtung Das Argument für diese Operation ist x, also ein Element und nicht nur ein Schlüssel.

Die Laufzeit ist hier ebenfalls $\mathcal{O}(1)$, zumindest wenn wir doppelt verkettete Listen benutzen. Dann müssen wieder nur einige Referenzen geändert werden, sodass x aus der Liste entfernt wird.

Falls wir statt x einen Schlüssel übergeben würden oder einfach verkettete Listen verwenden würden, müssten wir zuerst eine Suche durchführen. Im ersten Fall, um x zu finden, im zweiten Fall, um den Vorgänger von x in der Liste zu finden.

Laufzeit der Suche

Wir wollen nun die Laufzeit der Suche in einer Hashtabelle möglichst genau bestimmen. Anstatt uns dabei die Anzahl der Elemente in der Liste im Index $h(k)$ anzusehen, führen wir eine Analyse über den Füllgrad $\alpha = n/m$ durch, wobei n die Gesamtzahl der Elemente in der Hashtabelle ist und m die Anzahl der Indizes (Slots) in dem Array, das unsere Hashtabelle repräsentiert.

Als erstes stellen wir fest, dass die worst-case Laufzeit der Suche $\mathcal{O}(n)$ ist. Das passiert, wenn alle Schlüssel durch $h(k)$ in denselben Slot gehasht werden und wir einen Schlüssel suchen, das noch nicht in der Tabelle vorhanden ist. An dieser Laufzeit können wir im allgemeinen Fall auch nichts verbessern.

Wir wollen hier aber nicht aufhören und uns noch die durchschnittliche Laufzeit einer Suche ansehen. Offensichtlich hängt diese von der Hashfunktion h ab (die wir hier aber noch nicht kennen). Daher treffen wir nun eine (zugegebenermaßen sehr starke) Annahme über h: Wir nehmen an, dass jedes Element mit gleicher Wahrscheinlichkeit in einen beliebigen der m Slots gehasht wird. Wir nehmen außerdem an, dass die Elemente unabhängig voneinander gehasht werden. Diese Annahme nennt man **Einfaches uniformes Hashing** (engl.: *simple uniform hashing*).

Nun wollen wir eine Hashtabelle mit einer solchen Hashfunktion analysieren. Um einfaches uniformes Hashing umzusetzen, helfen uns zwei Intuitionen:

1. Wir haben eine zufällige Eingabe und eine konstante Hashfunktion.
2. Wir haben eine feste Eingabe und randomisieren die Hashfunktion irgendwie (z.B. wählen wir beim Initialisieren der Hashtabelle eine zufällige Funktion aus einer Menge von Hashfunktionen aus).

Der erste Fall ist aber deutlich einfacher zu betrachten, auch wenn er in der Realität nur bedingt vorkommt. Wir beschränken uns hier trotzdem darauf.

Zur Analyse definieren wir zunächst

$$n_j = length(T[j]) \qquad j \in \{0, 1, \ldots, m-1\}$$

als die Länge der Liste im Slot j. Dann gilt offensichtlich $n_0 + n_1 + ... + n_{m-1} = n$.

Durch unsere Annahme des einfachen uniformen Hashings wissen wir außerdem, dass der Erwartungswert von n_j dessen Durchschnitt entspricht:

$$E[n_j] = \alpha = \frac{n}{m}$$

Da wir außerdem davon ausgehen, dass h in $\mathcal{O}(1)$ berechnet wird, hängt die Laufzeit der Suche nach dem Element mit dem Schlüssel k linear von der Länge der Liste im Slot $h(k)$ ab (also linear von $n_{h(k)}$).

Wir betrachten im Folgenden die erfolglose und erfolgreiche Suche getrennt.

Satz 7.1. *Unter der Annahme des einfachen uniformen Hashings ist die erwartete Laufzeit einer* erfolglosen *Suche bei einer Hashtabelle mit verketteten Listen in* $\Theta(1 + \alpha)$.

Beweis. Wir stellen fest:

- Jeder Schlüssel k wird mit gleicher Wahrscheinlichkeit in einen der m Slots gehasht. Also gibt es keinen Unterschied zwischen den verschiedenen Slots.
- Die erwartete Laufzeit für eine erfolglose Suche für k ist die erwartete Laufzeit, die benötigt wird, die gesamte Liste in $T[h(k)]$ zu durchsuchen.
- Die erwartete Länge von $T[h(k)]$ ist $E[n_{h(k)}] = \alpha$.
- Also müssen wir im erwarteten Fall α Elemente untersuchen.
- Die Ausführung der Hashfunktion ist außerdem konstant.

Damit erhalten wir die Laufzeit $\Theta(1 + \alpha)$ im erwarteten Fall. Bemerke, dass wir diese Schreibweise nicht weiter verkürzen können, ohne weitere Annahmen zu treffen. Es könnte je nach Hashfunktion sein, dass α ungefähr 1 oder kleiner oder größer ist. □

Satz 7.2. *Unter der Annahme des einfachen uniformen Hashings ist die erwartete Laufzeit einer* erfolgreichen *Suche bei einer Hashtabelle mit verketteten Listen in* $\Theta(1+\alpha)$.

Beweis. Da es hier genau einen Slot in der Hashtabelle gibt, in der der gesuchte Schlüssel ist, können wir hier nicht alle Slots gleich behandeln. Wir können aber annehmen, dass das gesuchte Element mit gleicher Wahrscheinlichkeit jedes der n Elemente in der Hashtabelle sein könnte. Dann ist die Wahrscheinlichkeit, dass das Element in einem Slot j ist, anhängig von der Länge der Liste in dem Slot.

Sei daher x_i das i-te Element, das in die Hashtabelle eingefügt wurde. Dann ist $1 \leq i \leq n$ und $k_i = key(x_i)$. Nun definieren wir eine Bernoulli-Zufallsvariable mit

$$X_{ij} = \begin{cases} 1 & h(k_i) = h(k_j) \\ 0 & \text{sonst} \end{cases}$$

Unter unserer Annahme des einfachen uniformen Hashings ist dann

$$\begin{aligned} P(X_{ij} = 1) &= \sum_{z=1}^{m} P(h(k_i) = z) \cdot P(h(k_j) = z) \\ &= \sum_{z=1}^{m} \left(\frac{1}{m}\right)^2 \\ &= \frac{1}{m} \end{aligned}$$

und damit auch $E[X_{ij}] = P(X_{ij} = 1) = \frac{1}{m}$.

Dann ist unsere erwartete Laufzeit

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right]$$

Wir summieren also für jedes Element (äußere Summe) die Anzahl der Elemente, die später hinzugefügt wurden (innere Summe), falls diese in demselben Slot landen (sonst ist X_{ij} nach Definition 0). Dazu addieren wir die Laufzeit der Hashfunktion. Da wir am Ende nur die Laufzeit für eine Suche brauchen, teilen wir am Ende durch die Anzahl der Elemente n und berechnen davon den Erwartungswert.

Nun ist

$$\begin{aligned} E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right] &= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} E[X_{ij}]\right) \\ &= \frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} \frac{1}{m}\right) \\ &= \left(\frac{1}{n}\sum_{i=1}^{n} 1\right) + \left(\frac{1}{n}\sum_{i=1}^{n}\sum_{j=i+1}^{n} \frac{1}{m}\right) \\ &= 1 + \frac{1}{nm}\sum_{i=1}^{n}\sum_{j=i+1}^{n} 1 = 1 + \frac{1}{nm}\left(n^2 - \frac{n(n+1)}{2}\right) \\ &= 1 + \frac{n}{m} - \frac{n+1}{2m} = 1 + \alpha - \frac{n+1}{2n}\alpha \\ &= 1 + \alpha\left(1 + \frac{n+1}{2n}\right) = \Theta(1+\alpha) \end{aligned}$$

□

Das heißt also, dass wenn m mindestens proportional zur Anzahl der Elemente n ist, dann ist $n = \mathcal{O}(m)$ und damit $\alpha = \mathcal{O}(1)$, wodurch eine Suche im erwarteten Fall konstante Zeit braucht (allerdings nur unter der Annahme des einfachen uniformen Hashings).

Hashfunktionen

Als nächstes wollen wir einige Hashfunktionen betrachten. Dazu überlegen wir uns zuerst, was eigentlich eine gute Hashfunktion ausmacht. Da ist schnell klar, dass wir versuchen wollen, möglichst nahe an das einfache uniforme Hashing heranzukommen. Allerdings ist das in den meisten Fällen nicht möglich, da es für jede Hashfunktion immer auch eine Eingabe gibt, für die alle Schlüssel in denselben Slot gehasht werden.

Manchmal kennen wir aber auch die Verteilung der Schlüssel im Voraus. Wenn wir zum beispiel wissen, dass die Schlüssel zufällige reelle Zahlen $k \in [0, 1)$ sind, würde eine Hashfunktion $h(k) = \lfloor k \cdot m \rfloor$ die Annahme des einfachen uniformen Hashings erfüllen.

Naives Hashing

Wir betrachten ein Array der Größe $|U|$. Ein Element mit Schlüssel k wird am Index k gespeichert. Das gibt uns für alle Operationen eine konstante Laufzeit. Das Problem: Normalerweise ist unsere Menge von tatsächlichen Schlüsseln K deutlich kleiner als U. Wir verschwenden also unnötig viel Speicherplatz

Was macht eine gute Hashfunktion aus?

- Sie erfüllt (mehr oder weniger) die Annahme des einfachen uniformen Hashings. Dies ist aber meistens unmöglich, bzw. hängt definitiv von der Wahl der Schlüssel ab (es gibt für alle Hashfunktionen eine Möglichkeit die Schlüssel so zu wählen, dass wir im worst-case landen, dass also h(k) immer gleich ist). Manchmal kennen wir allerdings die Verteilung der Schlüssel. Zum Beispiel könnten die Schlüssel zufällige reelle Zahlen aus k \in [0, 1) sein. wenn die Schlüssel unabhängig und uniform gewählt werden, dann reicht $h(k) = \lfloor k \cdot m \rfloor$ als Hash-Funktion aus (und erfüllt die Bedingung des Einfachen Uniformen Hashings genau).

Meistens ist aber das Universum von Schlüsseln eine Teilmenge der natürlichen Zahlen. Wir schauen uns zwei einfache Hashfunktionen an.

Divisionsmethode

Die Divisionsmethode definiert die Hashfunktion als

$$h(k) = k \bmod m$$

Diese Methode ist relativ schnell, hat aber einige Nachteile:

Bestimmte Werte für m können sehr ungünstig sein. Zum Beispiel 2er Potenzen. Wenn wir nämlich $m = 2^p$ für ein $p \in \mathbb{N}$ wählen, betrachten wir bei $k \bmod 2^p$ genau die p niedrigstwertigen Bits von k und ignorieren den Rest. Es wäre aber besser, wenn wir alle Bits berücksichtigen, um k zu hashen.

Meistens ist es für die Divisionsmethode ganz gut, für m eine Primzahl zu wählen, die nicht zu nahe an einer 2er Potenz liegt.

Multiplikationsmethode

Die Multiplikationsmethode definiert die Hashfunktion als

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

mit

- A ist eine Konstante mit $0 < A < 1$.
- kA ist eine reelle Zahl mit $0 \leq kA < k$
- $kA \bmod 1$ ist der Nachkommateil von kA, also ist $kA \bmod 1 \in [0, 1)$.
- Dann ist $m(kA \bmod 1) \in [0, m)$ und $\lfloor m(kA \bmod 1) \rfloor \in [0, 1, \ldots, m-1]$.

Der Vorteil dieser Methode ist, dass der Wert von m egal ist. Tatsächlich wählt man meistens gerade eine 2er Potenz, da dann die Rechenoperationen auf der Hardware besonders effizient durchgeführt werden können.

8. Greedy-Algorithmen

Die **Greedy-Strategie** ist (wie Divide & Conquer oder Dynamische Programmierung) ein Design-Paradigma. Die Idee ist dabei: Ein Greedy-Algorithmus (greedy = *gierig*) trifft immer die Entscheidung, die zum aktuellen Zeitpunkt am besten aussieht, bzw. am zielführensten erscheint. Im Allgemeinen werden so nicht immer optimale Ergebnisse erzielt, allerdings gibt es viele Probleme, die durch einen Greedy-Algorithmus gelöst werden können. Es gibt außerdem noch eine Reihe weiterer Probleme, die zwar nicht optimal gelöst werden können, bei denen aber zumindest eine sehr gute Lösung erzielt wird. Man sollte aber auch bemerken, dass es Probleme oder Lösungsansätze gibt, bei denen sich eine Greedy-Strategie als fatal erweist. Das Schöne an Greedy-Algorithmen ist im Normalfall, dass man sie relativ leicht implementieren kann.

Huffman-Codes

Huffman-Codes sind ein Anwendungsbereich, in dem Greedy-Algorithmen erfolgreich angewendet werden können.

Definitionen

Die Idee beim Huffman-Code ist die folgende: Wir zählen, wie oft jeder Buchstabe in der Eingabe vorkommt. Dann weisen wir jedem Eingabezeichen eine Folge von Bits zu, die dieses Zeichen kodiert. Dadurch können wir bei einer geschickten Wahl der Kodierung eine Kompression von 20 bis 90% erreichen.

Damit wir Mehrdeutigkeiten bei Codes verhindern, definieren wir als erstes die sogenannte Präfix-Eigenschaft.

Definition 8.1. *Ein Code erfüllt die* ***Präfix-Eigenschaft****, wenn kein Codewort das Präfix eines anderen Codewortes ist.*

Bei Huffman-Codes fordern wir, dass die Präfix-Eigenschaft erfüllt ist.

Beispiel

Gegeben sei ein Text mit 100.000 Zeichen. Wir betrachten der Einfachheit halber nur die Zeichen a, b, c, d, e und f. Angenommen die Buchstaben sind folgendermaßen verteilt:

a	b	c	d	e	f
45%	13%	12%	16%	9%	5%

Wenn wir eine Codierung wählen, die jedem Zeichen die gleiche Anzahl an Bits zuweist (hier mindestens 3), dann brauchen wir mindestens 300.000 Bits zum Codieren der Eingabe (so einen Code nennt man auch Block-Code).

Wenn wir allerdings eine variable Codewortlänge erlauben, können wir den folgenden Code wählen (der die Präfix-Eigenschaft erfüllt):

a	b	c	d	e	f
0	101	100	111	1101	1100

Das ergibt dann einen Speicherplatz von

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1.000 = 224.000$$

Baumrepräsentation

Für Codes, die die Präfix-Eigenschaft erfüllen gibt es eine sehr bequeme Darstellung in Baumform. Wir konstruieren für einen gegebenen Code einen Baum nach folgendem Schema:

- Jedes Blatt repräsentiert einen Buchstaben der Eingabe.
- Wir interpretieren den Pfad von der der Wurzel zu einem Blatt als ein binäres Codewort. 0 bedeutet „links“, 1 bedeutet „rechts“.
- Oft schreibt man in die inneren Knoten des Baumes zusätzlich die Häufigkeiten der entsprechenden Unterbäume.

Optimale Codes

Man kann zeigen, dass ein optimaler Code immer durch einen vollständigen binären Baum repräsentiert werden, also durch einen Baum, bei dem jeder innere Knoten genau zwei Kinder hat (der Baum muss nicht notwendigerweise balanciert sein). Der Beweis wird in der Übung gemacht.

Man kann außerdem zeigen, dass für vollständige binäre Bäume, die einen Code mit $|C|$ Zeichen repräsentieren die Anzahl der Blätter $|C|$ ist und die Anzahl der inneren Knoten $|C| - 1$. Auch dieser Beweis wird in der Übung gemacht.

Kosten

Die Kosten sollen die Anzahl an Bits sein, die zum Kodieren nötig sind. Wir definieren:

- Für jedes $c \in C$ ist $f(c)$ die Häufigkeit (*frequency*) des Buchstaben.
- $d_T(c)$ ist die Tiefe des Blattes von c in der Baumrepräsentation (und damit die Länge des Codeworts).

Dann berechnen sich die Kosten für einen Baum T durch:

$$B(T) = \sum_{c \in C} f(c) \cdot d_T(c)$$

Idee

Wir benutzen eine Min-Priority-Queue für den Algorithmus. Dies ist ein Min-Heap mit bestimmten Eigenschaften und den folgenden Operationen:

- Build-Min-Heap: Konstruiert einen Heap aus n Elementen in $\mathcal{O}(n)$.
- Extract-Min: Findet das kleinste Element und entfernt es in $\mathcal{O}(\log n)$.
- Insert: Fügt ein neues Element ein. Dies geht in $\mathcal{O}(\log n)$.

Dann funktioniert der Huffman-Algorithmus wie folgt:

- Wir konstruieren den Code-Baum von den Blättern zur Wurzel
- Wir beginnen mit den $|C|$ Blättern des Baumes und machen dann $|C| - 1$ Merge-Operationen, um einen optimalen Baum zu erzeugen.
- In jedem Merge extrahieren wir die beiden Zeichen, die am seltensten vorkommen und fügen stattdessen einen Knoten ein, der als Häufigkeit die Summe der Häufigkeiten der beiden zusammengeführten Zeichen hat.

Pseudocode

```
Huffman(C)
1  n = |C|
2  Q = Build-Min-Heap(C)
3  for i = 1 to n − 1
4        allocate new node z
5        left[z] = x = Extract-Min(Q)
6        right[z] = y = Extract-Min(Q)
7        f[z] = f[x] + f[y]
8        Insert(Q, z)
```

Zur Übung: Zeigen Sie, dass ein Code, der durch den oben stehenden Algorithmus erzeugt wurde, die Präfix-Eigenschaft erfüllt.

Laufzeit

Die Initialisierung durch BUILD-MIN-HEAP läuft in $\mathcal{O}(n)$. Danach haben wir n Schleifendurchläufe, von denen jeder in $\mathcal{O}(\log n)$ liegt. Die Laufzeit ist also insgesamt in

$$\mathcal{O}(n \log n)$$

Korrektheit

Um die Korrektheit des Huffman-Algorithmus zu beweisen, beweisen wir zunächst ein Lemma.

Lemma 8.1. *Sei C das Alphabet für die Codewörter und $c \in C$ ein Zeichen mit der Freqzenz $f[c]$. Seien außerdem $x, y \in C$ die beiden Zeichen in C mit den niedrigsten Frequenzen. Dann gibt es einen optimalen (Huffman-)Code für C, in dem die Codewörter für x und y dieselbe Länge haben und sich nur durch ein Bit unterscheiden.*

Beweis. Seien T ein optimaler Baum für C und a und b zwei Geschwisterblätter maximaler Tiefe in T. Wir nehmen oBdA an, dass $f[a] \leq f[b]$ und $f[x] \leq f[y]$ gilt. Da $f[x]$ und $f[y]$ die beiden niedrigsten Frequenzen sind, gilt dann auch $f[x] \leq f[a]$ und $f[y] \leq f[b]$.

Nun konstruieren wir einen neuen Baum T', indem wir die Knoten a und x tauschen. Aus T' konstruieren wir dann einen weiteren Baum T'', indem wir die Knoten b und y tauschen.

Wir betrachten nun die Kosten von T im Vergleich zu T':

$$\begin{aligned}
&B(T) - B(T') \\
=&\sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\
=&f[x] d_T(x) + f[a] d_T[a] - f[x] d_{T'}(x) - f[a] d_{T'}(a) \\
=&f[x] d_T(x) + f[a] d_T[a] - f[x] d_T(a) - f[a] d_T(x) \\
=&(f[a] - f[x]) \cdot (d_T(a) - d_T(x)) \\
\geq&0
\end{aligned}$$

Die dritte Zeile kommt zustande, da sich die Kosten für alle $c \neq a, b, x, y$ von T zu T' nicht ändern. Daher fallen diese durch die Subtraktion weg. Die letzte Ungleichung gilt, da nach unserer oBdA-Annahme $f[a] - f[x] \geq 0$ ist und außerdem $d_T(a) - d_T(x) \geq 0$ ist.

Die obige Rechnung sagt aus, dass $B(T) - B(T') \geq 0$ ist, dass also die Kosten von T größer oder gleich den Kosten des neuen T' sind. Eine analoge Rechnung lässt sich auch

für T' und T'' durchführen. Damit gilt also

$$B(T'') \leq B(T') \leq B(T)$$

Da aber T bereits ein optimaler Baum ist, kann es keine anderen Bäume geben, die niedrigere Kosten haben. Damit müssen also T'' und T' auch optimale Bäume sein, was den Beweis für Lemma 8.1 abschließt. □

Aus dem Beweis können wir zwei Schlussfolgerungen ziehen:

1. Wenn wir in jedem optimalen Baum die beiden Knoten mit minimaler Frequenz nach unten „tauschen" können, dann können wir auch in unserem Algorithmus damit anfangen, die beiden Knoten mit der niedrigsten Frequenz zusammenzufassen.

2. Die Elemente mit der niedrigsten Frequenz können ganz unten im Baum stehen, müssen das aber nicht. Es gibt auch optimale Bäume, bei denen die Zeichen mit der niedrigsten Frequenz nicht die längsten Codewörter haben (auch wenn der Huffman-Algorithmus solche Bäume niemals erzeugt).

Um nun den Korrektheitsbeweis für den Huffman-Algorithmus zu beenden, müssen wir noch zeigen, dass wir tatsächlich zwei Elemente zusammenfassen dürfen, um einen optimalen Baum zu erzeugen. Dies tun wir in Lemma 8.2:

Lemma 8.2. *Seien $x, y \in C$ die beiden Zeichen mit der niedrigsten Frequenz. Sei außerdem $C' = C - \{x, y\} \cup \{z\}$ mit $f[z] = f[x] + f[y]$ und T' ein Baum, der einen optimalen Präfixcode für C' repräsentiert (bemerke: z ist in T' ein Blatt). Dann repräsentiert der Baum T, der entsteht, wenn wir in T' das Blatt z durch einen inneren Knoten mit den Kindern x und y ersetzen, ebenfalls einen optimalen Präfixcode für C.*

Beweis. Zuerst stellen wir fest, dass

$$\forall c \in C - \{x, y\} : d_T(c) = d_{T'}(c)$$

gilt und damit auch $f[c]d_T(c) = f[c]d_{T'}(c)$ ist. Alle „anderen" Zeichen in c bleiben also unverändert. Als wollen wir zeigen, dass für die Kosten von T und T' gilt:

$$B(T') = B(T) - f[x] - f[y]$$

Dazu stellen wir zuerst fest, dass $d_T(x) = d_T(y) = d_{T'}(z) + 1$ ist (in Worten: Das Codewort für z ist ein Zeichen kürzer als die Codewörter für x und y). Dies gilt nach Lemma 8.1, da wir für die beiden Zeichen mit der geringsten Frequenz Codewörter wählen können, die gleich lang sind und sich nur durch ein Zeichen unterscheiden. Da wir außerdem T so konstruieren, dass z der Elternknoten von x und y ist, ist das Codewort von z entsprechend ein Bit kürzer.

Damit gilt dann

$$\begin{aligned}
&f[x]d_T(x) + f[y]d_T(y) \\
=&f[x] \cdot (d_{T'}(z) + 1) + f[y] \cdot (d_{T'}(z) + 1) \\
=&(f[x] + f[y]) \cdot (d_{T'}(z) + 1) \\
=&f[z] \cdot (d_{T'}(z) + 1) \\
=&f[z]d_{T'}(z) + f[z] \\
=&f[z]d_{T'}(z) + (f[x] + f[y])
\end{aligned}$$

Da nun $f[x]d_T(x) + f[y]d_T(y) = f[z]d_{T'}(z) + (f[x] + f[y])$ ist, gilt

$$\begin{aligned}
&B(T) = B(T') + f[x] + f[y] \\
\iff &B(T') = B(T) - f[x] - f[y]
\end{aligned}$$

Der Beweis von Lemma 8.2 erfolgt nun durch einen Widerspruch: Angenommen T würde keinen optimalen Präfixcode für C repräsentieren. Dann gäbe es einen anderen Baum T'' mit $B(T'') < B(T)$.

Nach Lemma 8.1 sind x und y Geschwisterknoten (oder können zu welchen gemacht werden). Dann können wir in T'' x und y durch ein neues Blatt z ersetzen, für das $f[z] = f[x] + f[y]$ gilt. Dann ist

$$\begin{aligned}
B(T''') &= B(T'') - f[x] - f[y] \\
&< B(T) - f[x] - f[y] \\
&= B(T') \\
\implies B(T''') &< B(T')
\end{aligned}$$

Das ist aber ein Widerspruch, da wir am Anfang des Lemmas angenommen haben, dass T' bereits optimal ist. Damit haben wir Lemma 8.2 bewisen. □

Allgemeine Greedy-Probleme

Greedy-Algorithmen können eine sehr bequeme und effiziente Art zum Lösen von Optimierungsproblemen sein. Allerdings kann nicht jedes Problem durch einen Greedy-Algorithmus gelöst werden. Es ist sogar noch schlimmer: Es gibt keinen allgemeinen Weg, um zu beurteilen, ob ein Problem durch einen Greedy-Algorithmus gelöst werden kann. Es gibt aber einige wichtige Eigenschaften, die ein Problem haben muss, damit ein Greedy-Algorithmus existieren kann. Welche Eigenschaften das sind, unterscheidet sich von Autor zu Autor. Cormen schreibt:

> How can one tell if a greedy algorithm will solve a particular optimization problem? There is no way in general, but the greedy-choice property and optimal substructure are the two key ingredients.

Greedy-Eigenschaft

Die Greedy-Eigenschaft (*greedy-choice* property) ist die Eigenschaft eines Problems, nach der *greedy* (also gierig) entschieden wird. Beim Huffman-Algorithmus sind das die Frequenzen der einzelnen Zeichen.

Für einen Greedy-Algorithmus muss es eine solche Eigenschaft geben, damit wir durch lokal optimale Entscheidungen bei einer global optimalen Lösung ankommen können. Zur Erinnerung: Wir dürfen dabei nicht die Lösungen von kleineren Teilproblemen berücksichtigen, sondern müssen mit den Daten der bisherigen (Schleifen)durchläufe auskommen.

Optimale Substruktur

Die optimale Lösung eines Problems muss in sich die optimalen Lösungen der Teilprobleme enthalten. Dass das beim Huffman-Algorithmus der Fall ist, haben wir in Lemma 8.2 bewiesen, indem wir gezeigt haben, dass die optimale Lösung für C (mit x und y) die optimale Lösung für C' (mit z anstelle von x und y) enthält.

Scheduling

Beim Scheduling-Problem versuchen wir mehrere Aufgaben so hintereinander auszuführen, dass die durchschnittliche Wartezeit minimal wird. Dabei ist zu berücksichtigen, dass die Aufgaben (Jobs), die später ausgeführt werden, erst so lange warten müssen, wie die früheren Jobs ausgeführt werden. Man sollte außerdem beachten, dass die durchschnittliche **Wartezeit** minimiert werden soll, nicht die durchschnittliche Laufzeit.

Tatsächlich betrachten wir ein Problem, dass sich leicht von dem obigen Scheduling-Problem unterscheidet, dessen Lösung aber äquivalent ist. Statt der durchschnittlichen Wartezeit (die oben minimiert werden soll) minimieren wir die Gesamtwartezeit (die ja n Mal die durchschnittliche Wartezeit ist). Die Aufgabe ist also, den Wert der folgenden Formel zu minimieren:

$$T = \sum_{i=1}^{n} (\text{Zeit für Aufgabe } i)$$

Das Problem des Scheduling findet sich im Alltag an vielen Stellen. Zum Beispiel muss ein Prozessor entscheiden, in welcher Reihenfolge anstehende Prozesse bearbeitet werden sollen. Es findet sich auch bei Aufzügen, die entscheiden müssen, in welcher Reihenfolge die Etagen angefahren werden sollen, um möglichst wenig Wartezeiten zu haben.

Beispiel

Gegeben seinen 3 Kunden mit den jeweiligen Bearbeitungszeiten:

$$t_1 = 5 \qquad 5_2 = 10 \qquad t_3 = 3$$

Es gibt nun $3! = 6$ Möglichkeiten, in welcher Reihenfolge diese Kunden bedient werden.

Reihenfolge	T	
1 2 3	$5 + (5 + 10) + (5 + 10 + 3) = 38$	
1 3 2	$5 + (5 + 3) + (5 + 3 + 10) = 31$	
2 1 3	$10 + (10 + 5) + (10 + 5 + 3) = 43$	
2 3 1	$10 + (10 + 3) + (10 + 3 + 5) = 41$	
3 1 2	$3 + (3 + 5) + (3 + 5 + 10) = 29$	optimal
3 2 1	$3 + (3 + 10) + (3 + 10 + 5) = 34$	

An diesem Beispiel lässt sich leicht erkennen, dass die optimale Lösung erreicht ist, wenn wir die Aufgaben in aufsteigender Reihenfolge der Bearbeitungszeiten abarbeiten. Dies müssen wir nun in einen Greedy-Algorithmus übersetzen und die Korrektheit dieser Idee beweisen.

Idee

Die Idee für einen Greedy-Algorithmus für das Scheduling ist einfach: Wir wählen zu jedem Zeitpunkt die anscheinend beste Option. In diesem Fall ist das die kleinste Laufzeit, die wir finden können.

Pseudocode

Als Übung für den Leser.

Korrektheit

Wir beweisen die Korrektheit der Idee (nicht die Korrektheit der Implementation).

Satz 8.1. *Der Greedy-Algorithmus, der in jedem Durchgang den Job mit der kürzesten Laufzeit wählt, löst das Scheduling-Problem optimal.*

Beweis. Sei $P = p_1 p_2 \dots p_n$ eine Permutation von $\{1, \dots, n\}$. Sei $s_i = t_{p_i}$ die Laufzeit

des i-ten Jobs. Dann ist

$$\begin{aligned} T(P) &= s_1 + (s_1 + s_2) + (s_1 + s_2 + s_3) + \dots \\ &= ns_1 + (n-1)s_2 + (n-2)s_3 + \dots \\ &= \sum_{k=1}^{n} (n-k+1) \cdot s_k \end{aligned}$$

Angenommen P ist nicht nach aufsteigenden Bearbeitungszeiten geordnet. Dann gibt es a, b mit $a < b$ und $s_a > s_b$. Dann können wir die Positionen des a-ten und b-ten Jobs vertauschen und erhalten damit als Laufzeit

$$T(P') = (n-a+1)s_b + (n-b+1)s_a + \sum_{k \in \{1,n\} - \{a,b\}} (n-k+1)s_k$$

Nun vergleichen wir die Laufzeit von P mit P':

$$\begin{aligned} & T(P) - T(P') \\ =& (n-a+1)s_a + (n-b+1)s_b - (n-a+1)s_b - (n-b+1)s_a \\ =& (n+1)(s_a + s_b - s_b - s_a) + a(s_b - s_a) + b(s_a - s_b) \\ =& b(s_a - s_b) - a(s_a - s_b) \\ =& (b-a) \cdot (s_a - s_b) \\ >& 0 \end{aligned}$$

Die letzte Ungleichung gilt, da wir $b > a$ und $s_a > s_b$ angenommen haben. Da aber so $T(P) > T(P')$ ist, wissen wir nun, dass wir jeden Zeitplan P verbessern können, indem wir zwei Jobs nach der Regel der kürzesten Bearbeitungszeit (*shortest-service-time-first*) tauschen. Das beendet den Beweis. □

9. Dynamische Programmierung

Ähnlich wie auch Greedy-Algorithmen ist die dynamische Programmierung eine Methode zum algorithmischen Lösen von Optimierungsproblemen. Wir wollen das Prinzip der dynamischen Programmierung anhand eines Beispiels kennenlernen.

Kettenmultiplikation von Matrizen

Gegeben ist eine Kette von Matrizen $A_1, A_2, \ldots, A_n$, die in der gegebenen Reihenfolge miteinander multipliziert werden sollen. Dabei nehmen wir an, dass die Matrix A_i die Dimension $(p_{i-1} \times p_i)$ hat (dass die Matrizen also auch wirklich miteinander multipliziert werden können).

Da die Matrizenmultiplikation assoziativ ist, gibt es nun aber viele Möglichkeiten die n Matrizen miteinander zu multiplizieren. Dass sich dies tatsächlich auf die Laufzeit auswirkt, zeigt das folgende Beispiel:

Angenommen wir wollen die Matrizen mit folgenden Dimensionen multiplizieren:

$$A_1 \text{ mit } (10 \times 100) \quad A_2 \text{ mit } (100 \times 5) \quad A_3 \text{ mit } (5 \times 50) \quad A_4 \text{ mit } (50 \times 10)$$

Nun gibt es die folgenden Klammerungsvarianten:

$$\begin{gathered} (A_1(A_2(A_3A_4))) \\ (A_1((A_2A_3)A_4)) \\ ((A_1A_2)(A_3A_4)) \\ ((A_1(A_2A_3))A_4) \\ (((A_1A_2)A_3)A_4) \end{gathered}$$

Die Multiplikation von zwei Matrizen der Dimensionen $(p \times q)$ und $(q \times r)$ kann mit $p \cdot q \cdot r$ Operationen durchgeführt werden. Nun vergleichen wir die Klammerungsvarianten 2 und 5:

$$\begin{array}{ll} \text{Variante 2} & 100 \cdot 5 \cdot 50 + 100 \cdot 50 \cdot 10 + 10 \cdot 100 \cdot 10 = 85.000 \\ \text{Variante 5} & 10 \cdot 100 \cdot 5 + 10 \cdot 5 \cdot 50 + 10 \cdot 50 \cdot 10 = 12.500 \end{array}$$

Offensichtlich macht es also einen gewaltigen Unterschied, wie die Klammerung gewählt wird. Allerdings ist die Anzahl möglicher Klammerungen exponentiell zur Anzahl der Faktoren. Das ist ein Problem. Wir wollen nun das Prinzip der dynamischen Programmierung anwenden, um doch noch einen Algorithmus zu bekommen, der eine optimale Klammerung in polynomieller Zeit findet.

Das Prinzip der dynamischen Programmierung

Um einen Algorithmus nach dem Prinzip der dynamischen Programmierung zu entwickeln, geht man in vier Schritten vor:

1. **Beschreibung der Struktur** einer optimalen Lösung.
2. Rekursive Definition des **Wertes** einer optimalen Lösung.
3. **Berechnung des Wertes** einer optimalen Lösung im Bottom-Up-Verfahren.
4. **Konstruktion einer optimalen Lösung** durch die vorherigen Schritte (wenn gefordert).

Im ersten Moment mag es so scheinen, als ob das Vorgehen sehr ähnlich zu Greedy-Algorithmen ist. Und tatsächlich gibt es gewisse Ähnlichkeiten. Wir berechnen auch hier eine optimale Teillösung, die am Ende für die Gesamtlösung verwendet wird. Allerdings berechnen wir bei der dynamischen Programmierung nur die Werte der optimalen Teillösungen und nicht deren Struktur. Die Struktur ergibt sich dann aus den berechneten Werten.

Wir wollen jetzt das Prinzip der dynamischen Programmierung auf die Kettenmultiplikation von Matrizen anwenden.

1. Beschreibung der Struktur einer (optimalen) Lösung

Sei $A_{i,j} = A_i \cdots A_j$ für $i \leq j$. Wenn $i < j$ ist, dann muss jede Lösung (jede Klammerung) die Kette von Matrizen an einem k zerteilen, mit $i \leq k < j$. Das Endergebnis ergibt sich dann aus den Multiplikationen $A_{i,k}$, $A_{k+1,j}$ und $A_{i,k} \cdot A_{k+1,j}$. Wir berechnen also die „linke Klammer“ $A_{i,k}$ und die „rechte Klammer“$A_{k+1,j}$ und multiplizieren diese dann miteinander.

Dies erfordert als Rechenaufwand

- die Berechnung von $A_{i,k}$ plus
- die Berechnung von $A_{k+1,j}$ plus
- die Multiplikation von $A_{i,k}$ mit $A_{k+1,j}$.

Eine optimale Lösung wird dadurch charakterisiert, dass die Wahl von k optimal ist (dass es also kein anderes k gibt, welches besser geeignet ist).

Das k bleibt zunächst unbekannt. Wir werden später sehen, wie wir mit diesem Problem umgehen.

2. Rekursive Definition des Wertes einer optimalen Lösung

Sei $m[i, j]$ die Anzahl der Skalarprodukte, die zum Berechnen von $A_{i,j} = A_i \cdot A_{i+1} \cdots A_j$ benötigt werden (dann berechnet sich das Endergebnis durch $m[1, n]$). Wir definieren den Wert von $m[i, j]$ rekursiv:

Fall 1 $i = j$: ist „multiplizieren“ wir eine einzelne Matrix mit nichts anderem. Das Ergebnis ist die Matrix selbst. Wir müssen also gar nicht multiplizieren. Also ist

$$m[i, j] = m[i, i] = 0$$

Fall 2 $i < j$: Angenommen die optimale Klammerung erfolgt am Index k mit $i \leq k < j$. Dann ist die Matrix $A_{i,k}$ eine $p_{i-1} \times p_k$-Matrix und $A_{k+1,j}$ eine $p_k \times p_j$-Matrix. Also ist

$$m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} \cdot p_k \cdot p_j$$

Da wir aber den Wert für k immer noch nicht kennen, müssen wir den optimalen Wert zunächst über alle möglichen k definieren:

$$m[i, j] = \begin{cases} 0 & i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + p_{i-1} \cdot p_k \cdot p_j\} & i < j \end{cases}$$

Bemerke

Bis zu diesem Zeitpunkt kennen wir den Wert von k nicht. Das wird sich auch nicht wirklich ändern. Stattdessen müssen wir alle möglichen Werte von k ausprobieren und das beste wählen. Wie sich das auswirkt sehen wir unten noch.

Wenn wir einen Weg kennen würden, das optimale k zu berechnen, ohne alle Möglichkeiten auszuprobieren, könnten wir dieses Problem auch durch einen Greedy-Algorithmus lösen. Da ein Greedy-Algorithmus aber nicht die Ergebnisse der Teilprobleme berücksichtigen darf, brauchen wir hier die dynamische Programmierung.

3. Berechnung des Wertes einer optimalen Lösung

Wir wollen nun $m[1, n]$ berechnen. Nach unserer Definition oben ist dies die minimale Anzahl an Multiplikationen, um $A_1 \cdot A_2 \cdots A_n$ zu berechnen.

Im ersten Moment mag es so aussehen, als ob wir hier nichts gewonnen haben. Wenn wir m so wie in der Definition rekursiv berechnen, bräuchte dies mindestens eine Laufzeit von $\Omega(2^n)$, da wir die Teilprobleme wieder und wieder berechnen müssen. Das wollen wir nun verbessern, indem wir uns bereits berechnete Teilergebnisse in m merken und wiederverwenden. Dies nennt man eine **bottom-up-Berechnung**. Wenn wir die Kettenmultiplikation von Matrizen mit einem bottom-up-Verfahren berechnen, können wir tatsächlich eine polynomielle Laufzeit erhalten.

Für einen solchen Algorithmus stellen wir fest, dass die Berechnung von $m[i, j]$ nur von **kleineren Teilproblemen** für $k = 1, 2, \ldots, j - 1$ abhängt. Formal:

- $A_{i,k}$ ist das Produkt von $k - i + 1 < j - i + 1$ Matrizen.
- $A_{k+1,j}$ ist das Produkt von $j - k < j - i + 1$ Matrizen.

Wenn wir also bereits bekannte Ergebnisse wiederverwenden, sollte die Tabelle (Matrix) m in aufsteigender Reihenfolge der Kettenlänge befüllt werden.

Dynamische Programmierung

Die Wiederverwendung von Teilergebnissen ist ein wichtiger Charakterzug der dynamischen Programmierung. Im Normalfall merkt man sich die bekannten Ergebnisse in einer Tabelle bzw. Matrix und kann dann in späteren Schritten auf diese bekannten Ergebnisse zurückgreifen.

4. Konstruktion einer optimalen Lösung

Um am Ende des Algorithmus nicht nur den optimalen Wert zu kennen, sondern auch die optimale Lösung reproduzieren zu können, merken wir uns in jedem Schritt auch das k, das wir gewählt haben in einer Matrix s.

$$s[i, j] = k \iff m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} \cdot p_k \cdot p_j$$

Pseudocode

```
CHAIN-MULTIPLY(p = (p_1, p_2, ...))
 1  n = length[p] - 1
 2  for i = 1 to n
 3        m[i, i] = 0
 4  for ℓ = 2 to n
 5        for i = 1 to n - ℓ + 1
 6              j = i + ℓ - 1
 7              m[i, j] = ∞
 8              for k = i to j - 1
 9                    q = m[i, k] + m[k + 1, j] + p_{i-1} · p_k · p_j
10                    if q < m[i, j]
11                          m[i, j] = q
12                          s[i, j] = k
```

Einige Erklärungen zum Code:

- In Zeile 4 beginnen wir mit Ketten der Länge 2, da wir für Ketten der Länge 1 schon wissen, dass diese ohne Multiplikation lösbar sind (das ist im Pseudocode in Zeile 2-3 behandelt).
- Die Schleife in Zeile 5 berechnet die „nächsten“ Felder in m.
- In Zeile 7 wird das zu berechnende Feld zunächst mit einem Wert initialisiert, der größer als jeder andere Wert ist. Dies ist nur eine bequeme Schreibweise, damit wir nicht gesondert mit leeren Feldern umgehen müssen.
- Die Schleife in Zeile 8 berechnet dann letztlich die Anzahl der Rechenoperationen für eine bestimmte Klammerung (für ein bestimmtest k). Die Schleife probiert einfach alle möglichen Werte für k aus.
- In der Schleife in Zeile 8 kommen nur kleinere Indizes von m vor. Wir kennen dort also bereits die Werte von $m[i, k]$ und $m[k + 1, j]$.

Beispiel

Wir betrachten die Matrizen

$$\begin{array}{ll} A_1 & (30 \times 35) \\ A_2 & (35 \times 15) \\ A_3 & (15 \times 5) \\ A_4 & (5 \times 10) \\ A_5 & (10 \times 20) \\ A_6 & (20 \times 25) \end{array}$$

Nach dem Durchlauf des Algorithmus erhalten wir folgende Tabelle (Matrix) für m:

j \ i	1	2	3	4	5	6
6	15.125	10.500	5.375	3.500	5.000	0
5	11.875	7.125	2.500	1.000	0	
4	9.375	4.375	750	0		
3	7.875	2.625	0			
2	15.750	0				
1	0					

Tabelle 9.1.: Ergebnis des Algorithmus

Laufzeit

Der Algorithmus besteht aus drei verschachtelten Schleifen und einer separaten Schleife ganz am Anfang.

1. Die Schleife am Anfang macht $\mathcal{O}(n)$ Iterationen mit einer jeweils konstanten Laufzeit.
2. Die ℓ-Schleife macht $\mathcal{O}(n)$ Iterationen.
3. Die i-Schleife macht $\mathcal{O}(n)$ Iterationen.
4. Die k-Schleife macht $\mathcal{O}(n)$ Iterationen.

Da bis auf die geschachtelten Schleifen alle anderen Anweisungen in konstanter Zeit ausgeführt werden können ist die Laufzeit der verketteten Multiplikation von Matrizen in

$$\mathcal{O}(n^3)$$

Der Floyd-Warshall-Algorithmus

Im folgenden lernen wir einen Algorithmus kennen, der ähnlich wie der Algorithmus von Bellman und Ford bzw. der Algorithmus von Dijkstra kürzeste Pfade in Graphen findet (vgl. Kapitel 12). Allerdings beschränkt sich dieser Algorithmus nicht auf eine einzige Quelle, sondern berechnet alle kürzesten Pfade von allen Knoten zu allen Knoten

(*all-pairs-shortest-paths*). Es empfiehlt sich, zuerst das Kapitel 12 und Kapitel 10 zu lesen, da einige Argumentationen in diesem Kapitel dann einfacher nachzuvollziehen sind.

Formal lässt sich das Problem folgendermaßen formulieren:

Gegeben ist ein gerichteter, gewichteter Graph $G = (V, E)$ mit einer Gewichtungsfunktion $w : E \to \mathbb{R}$. Wir nehmen an, dass der Graph keine negativen Zyklen enthält. Außerdem gehen wir davon aus, dass die Knotenmenge $V = \{1, 2, \ldots, n\}$ ist. Dann ist das Ergebnis eines *All-Pairs-Shortest-Paths*-Algorithmus eine $n \times n$ Matrix $W = (w_{ij})$ mit den kürzesten Pfadlängen zwischen allen Knoten:

$$w_{ij} = \delta(i, j)$$

Zur Erinnerung: Das Gewicht eines Pfades $p = (v_1, v_2, \ldots, v_k)$ berechnet sich folgendermaßen:

$$w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1}).$$

Als ersten Ansatz zum Lösen dieses Problems könnte man einen *single-source-shortest-paths*-Algorithmus für jeden Knoten im Graphen ausführen. Allerdings würde dies bei dichten Graphen eine Laufzeit von $\mathcal{O}(n^4)$ benötigen. Wir werden im Folgenden daher einen Algorithmus entwickeln, der dieses Problem schneller lösen kann.

Adjazenzmatrizen

Der Floyd-Warshall-Algorithmus bekommt als Eingabe eine Adjazenzmatrix des Graphen G. Dies ist eine $n \times n$-Matrix $W = (w_{ij})$ mit

$$w_{ij} = \begin{cases} 0 & i = j \\ w(i, j) & i \neq j \land (i, j) \in E \\ \infty & i \neq j \land (i, j) \notin E \end{cases}$$

Entwicklung des Algorithmus

Beim Floyd-Warshall-Algorithmus handelt es sich um einen DP-Algorithmus (Dynamische Programmierung). Wir folgen also den üblichen vier Schritten:

1. **Beschreibung der Struktur** einer optimalen Lösung.
2. Rekursive Definition des **Wertes** einer optimalen Lösung.
3. **Berechnung des Wertes** einer optimalen Lösung im Bottom-Up-Verfahren.
4. **Konstruktion einer optimalen Lösung** durch die vorherigen Schritte (wenn gefordert).

1. Struktur eines kürzesten Pfades

Wir wissen bereits vom Bellman-Ford-Algorithmus, bzw. dem Algorithmus von Dijkstra, dass ein kürzester Pfad von u nach v, der über den Knoten w geht, die kürzesten Pfade von u nach w und von w nach v enthält. Dies ist auch für den Floyd-Warshall-Algorithmus der Ansatzpunkt des Algorithmus.

2. Rekursive Definition des Wertes einer optimalen Lösung

Wir definieren $d_{ij}^{(m)}$ als das Gewicht des kürzesten Pfades von i nach j, der höchstens m Kanten verwendet. Dann ist

$$d_{ij}^{(0)} = \begin{cases} 0 & i = j \\ \infty & i \neq j \end{cases}$$
$$d_{ij}^{(m)} = \min_k \left\{ d_{ik}^{(m-1)} + w_{kj} \right\}$$

In Worten: Der kürzeste Pfad von i nach j mit höchstens m Kanten ergibt sich aus dem kürzesten Pfad mit höchstens $m-1$ Kanten, an den eine weitere Kante angehängt wird. Welche weitere Kante angehängt wird, steht nicht sofort fest, sondern muss über die Minimierung über alle k herausgefunden werden. Bemerke außerdem, dass in der zweiten Zeile auch $k = i$ oder $k = j$ gelten kann.

3. Berechnung des Wertes einer optimalen Lösung

Wir suchen nun alle kürzesten Pfade. Da es insgesamt genau n Knoten gibt, wissen wir, dass alle zyklenfreien Pfade aus höchstens $n-1$ Kanten bestehen. Also können wir sicher sein, dass der kürzeste Pfad zwischen zwei beliebeigen Knoten i und j nach höchstens $n-1$ Iterationen gefunden wurde. Es gilt:

$$\delta(i, j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \dots$$

Wir müssen also $D^{(n-1)}$ berechnen. Ein erster Ansatz dafür wäre in insgesamt $n-1$ Iterationen alle n^2 Matrixeinträge zu berechnen. Für jeden müssen alle möglichen Zwischenknoten k geprüft werden. Dies würde eine Laufzeit von $\mathcal{O}(n^4)$ ergeben. Hier haben wir also nichts gewonnen.

Stattdessen berechnen wir die Lösung in einem Bottom-Up-Verfahren. Wie dies funktionieren könnte, zeigt ein Vergleich mit der Kettenmultiplikation von Matrizen. Wir erinnen uns, dass wir die optimale Verkettung zweier Matrizen durch

$$c_{ij} = \sum_k a_{ik} \cdot b_{kj}$$

berechnet haben. Wenn wir hier nun die Summe durch die min-Funktion ersetzen und $\cdot$ durch $+$, erhalten wir:

$$c_{ij} = \min_k \{a_{ik} + b_{ik}\}$$

Dies sieht der Formel, die wir oben definiert haben, erstaunlich ähnlich:

$$d_{ij}^{(m)} = \min_k \left\{ d_{ik}^{(m-1)} + w_{kj} \right\}$$

Deshalb wollen wir auch ein sehr ähnliches Berechnungsverfahren durchführen:

Wir definieren dazu $c_{ij}^{(m)}$ als das Gewicht des kürzesten Pfades von i nach j, der nur Zwischenknoten aus der Menge $\{1, 2, \ldots, m\}$ verwendet. Nach dieser Definition ist dann

$$\delta(i, j) = c_{ij}^{(n)}.$$

Die entscheidende Beobachtung ist nun, dass $c_{ij}^{(n)}$ in Abhängigkeit von $c_{ij}^{(<n)}$ berechnet werden kann. Die letzte Pfaderweiterung hängt also immer nur von den vorhergehenden Erweiterungen ab. Dasselbe gilt auch für alle anderen Pfaderweiterungen. Diese Beobachtung gibt uns eine direkte Berechnungsvorschrift:

$$\begin{aligned} c_{ij}^{(0)} &= w_{ij} \\ c_{ij}^{(m)} &= \min\left(c_{ij}^{(m-1)}, c_{im}^{(m-1)} + c_{mj}^{(m-1)} \right) \end{aligned}$$

Die min-Funktion hat hier zwei Argumente: Das erste entspricht dem Fall, dass keine Verbesserung des Pfades erzielt werden kann, wenn der Zwischenknoten m hinzugefügt wird. In dem Fall bleibt der kürzeste Pfad mit den Zwischenknoten aus $\{1, 2, \ldots m\}$ derselbe wie wenn der Zwischenknoten m nicht verwendet werden darf (vorherige Iteration). Andernfalls wird der Pfad entsprechend über den Zwischenknoten m verbessert.

Dies ergibt den folgenden Pseudocode (die Eingabe ist eine Adjazenzmatrix des Graphen G):

```
Floyd-Warshall(G)
1  for m = 1 to n
2      for i = 1 to n
3          for j = 1 to n
4              if c_ij > c_im + c_mj
5                  c_ij = c_im + c_mj
```

Dieser Code enthält tatsächlich noch eine weitere Optimierung, da bei den Zugriffen auf die Matrix c das $^{(m)}$ weggelassen wurde. Man könnte dieses nun wieder ergänzen, dadurch wäre der Algorithmus auch weiterhin korrekt. Tatsächlich wird aber in einer bestimmten Reihenfolge auf die Matrix zugegriffen, sodass in einem Iterationsschritt von m einmal berechnete Werte nicht wieder für andere Berechnungen gebraucht werden. Diese Tatsache geben wir hier ohne Beweis an.

Laufzeit

Man sieht sofort, dass die Laufzeit des Floyd-Warshall-Algorithmus in $\mathcal{O}(n^3)$ ist. Ergänzt man im obigen Code die $^{(m)}$ wieder, ist auch die Platzkomplexität in $\mathcal{O}(n^3)$ (es wird n Mal eine $n \times n$ Matrix erzeugt). Ohne die $^{(m)}$ ist die Platzkomplexität in $\mathcal{O}(n^2)$.

Es ist außerdem erwähnenswert, dass es sich hier um ein *besonders gutes* $\mathcal{O}(n^3)$ handelt (die Konstanten, die durch die Notation versteckt sind, sind sehr klein). Es wird in jedem Schritt nur ein Vergleich, 1-2 Additionen und eine Zuweisung gemacht. Dies macht den Floyd-Warshall-Algorithmus zu einem sehr beliebten und sehr effizienten Algorithmus zur Lösung des *All-Pairs-Shortest-Paths*-Problems.

Als letztes sollte man sich klar machen, dass die Berechnung von allen Pfaden im Graphen im Vergleich zu den Pfaden von einer einzigen Quelle aus unter Umständen nur unwesentlich aufwändiger ist. Der Bellman-Ford Algorithmus hat eine Laufzeit von $\Theta(V \cdot E)$. Wenn wir nun Pfade in einem sehr dichten Graphen suchen ($|E| \approx |V|^2$), dann ist die (asymptotische) Laufzeit des Floyd-Warshall-Algorithmus annähernd gleich der des Bellman-Ford-Algorithmus. Hier können wir also ohne zusätzlichen Zeitaufwand ein deutlich aussagekräftigeres Ergebnis erhalten. In der Realität muss man je nach Anwendungsfall entscheiden, welcher Algorithmus besser geeignet ist. Wenn man bereits weiß, dass ein Graph sehr dicht ist, lohnt es sich meistens, nur den Floyd-Warshall-Algorithmus zu verwenden. Enthält ein Graph aber viele Knoten und nur wenige Kanten, kann man durch die Verwendung des Bellman-Ford-Algorithmus (oder dem Algorithmus von Dijkstra) enorm viel Zeit einsparen.

10. Elementare Graphalgorithmen

Die Datenstruktur Graph

Definition 10.1. *Ein **Graph** ist eine Datenstruktur, die sehr vielseitig eingesetzt werden kann. Formal ist ein Graph $G = (V, E)$ ein 2-Tupel aus*

- *einer Knotenmenge V*
- *einer Kantenmenge E*

Die Knotenmenge ist im Normalfall eine Menge aus beliebigen Elementen. Bei der Kantenmenge unterscheiden wir *gerichtete* und *ungerichtete* Graphen.

- Bei gerichteten Graphen ist $E \subset V \times V$. Ein Element $e \in E$ nennt man dann eine gerichtete Kante. Bei gerichteten Graphen berücksichtigen wir die Richtung von Kanten. Das heißt, dass es sein kann, dass eine Kante $(u, v) \in E$ existiert, aber keine Kante (v, u).
- Bei ungerichteten Graphen ist uns die Richtung einer Kante egal. Formal könnte man $E \subset \{\{u, v\} : u, v \in V\}$ definieren. Häufig schreibt man aber wie bei gerichteten Graphen Tupel statt zweielementige Mengen. Bei ungerichteten Graphen ist für eine Kante $(u, v) \in E$ immer auch $(v, u) \in E$.

Um Graphen zur Implementierung intern zu repräsentieren, haben sich zwei Varianten durchgesetzt:

- Adjazenzlisten
- Adjazenzmatrizen

Diese betrachten wir nun genauer.

Adjazenzlisten

Man kann einen Graphen durch Adjazenzlisten repräsentieren. Die Idee bei dieser Repräsentation ist, dass alle ausgehenden Kanten von Knoten in einer Liste aus Listen gespeichert werden. Der Graph wird also als eine Liste mit $|V|$ Elementen repräsentiert. Jedes dieser Elemente ist eine Liste $Adj[u]$, in der die direkten Nachbarn des Knoten u stehen. Üblicherweise verwendet man hier verkettete Listen.

In gerichteten Graphen ist damit die Summe der Adjazenzlisten

$$\sum_{v \in V} Adj[v] = |E|$$

In ungerichteten Graphen speichern wir im Prinzip jede Kante zwei Mal (da jeder der verbundenen Knoten mit dem jeweils anderen verbunden ist). Damit ist hier

$$\sum_{v \in V} Adj[v] = 2 \cdot |E|$$

In beiden Fällen ist damit der Speicherbedarf für einen beliebigen Graphen in $\Theta(|V|+|E|)$.

In dieser Repräsentation müssen wir irgendwie verifizieren können, ob eine Kante existiert oder nicht (das müssen wir auch in jeder anderen Repräsentation). Wir wollen also wissen, ob für zwei Knoten u und v $(u, v) \in E$ ist. Dazu müssen wir prüfen, ob $v \in Adj[u]$ ist. Das kann in $O(|Adj[u]|)$ geschehen. Die Laufzeit dieses Problems wächst also mit der Anzahl der Nachbarn von u. Diese Laufzeitschranke gilt aber nur für direkte Nachbarn. Für Pfade im Graphen wird der Algorithmus etwas komplexer.

Möchte man mit der Adjazenzlistenrepräsentation einen gewichteten Graphen repräsentieren, braucht man ein zusätzliches Feld in jedem Listeneintrag, in dem das Kantengewicht gespeichert wird.

Adjazenzmatrizen

Eine Adjazenzmatrix eines Graphen ist eine $|V| \times |V|$-Matrix $A = (a_{ij})$ mit

$$a_{ij} = \begin{cases} 1 & \text{wenn } (i, j) \in E \text{ ist} \\ 0 & \text{sonst} \end{cases}$$

Damit wir eine solche Matrix verwenden können, müssen den Knoten Zahlen zwischen 1 und $|V|$ zugewiesen werden können.

Die Adjazenzmatrix hat einige interessante Eigenschaften:

- Der Speicherbedarf einer solchen Matrix ist in $\Theta(|V|^2)$, ist also unabhängig von E.
- Für ungerichtete Graphen ist die Adjazenzmatrix symmetrisch.
- Für gewichtete Graphen können wir 1-Einträge in der Matrix einfach durch das Kantengewicht ersetzen und können damit auch gewichtete Graphen quasi ohne Mehraufwand repräsentieren.
- Wir können in $\mathcal{O}(1)$ prüfen, ob zwei Knoten durch eine direkte Kante verbunden sind. Man kann aber nicht sofort sehen, ob ein Pfad zwischen zwei Knoten existiert.

Zur Notation

Die Laufzeiten von Graphalgorithmen hängen oft von der Anzahl der Knoten, Kanten, oder beidem im Graphen ab. Oft lässt man daher in der asymptotischen Notation die Betragsstriche um V bzw. E weg. Zum Beispiel sehen wir folgende Schreibweisen als gleichwertig an:

$$f(n) = \Theta(|V|^2 + E)$$
$$f(n) = \Theta(V^2 + E)$$

An dieser Stelle stellt sich die Frage, ob man diese Repräsentationen für Graphen nicht auch für die bereits bekannten Algorithmen anwenden könnte (z.B. Suchbäume oder Heaps). Tatsächlich wäre das möglich, ist aber hochgradig ineffizient. Insbesondere mit Heaps können wir in der Array-Repräsentation sehr effizient arbeiten. Diese Vorteile würden wir durch die allgemeinere Repräsentation zerstören.

Wann eine Adjazenzliste oder eine Adjazenzmatrix besser geeignet ist, kann man im Allgemeinen nicht sagen. Adjazenzmatrixen können besonders für Graphen mit vielen Knoten und wenig Kanten unnötig groß werden, haben aber eine bessere Zugriffszeit als Adjazenzlisten.

Breitensuche (Breadth-First-Search)

Die Breitensuche (BFS) in einem Graphen zählt mit zu den einfachsten Algorithmen zum Durchsuchen eines Graphen und ist ein wichtiger Bausteil für viele andere Graphenalgorithmen. Die Breitensuche *durchsucht* eine Zusammenhangskomponente eines Graphen $G = (V, E)$ und besucht dabei jeden Knoten in der Zusammenhangskomponente (bzw. alle Knoten im Graphen, falls es nur eine Zusammenhangskomponente gibt).

Idee

Wir beginnen bei einem Knoten s und besuchen dann nach und nach die Nachbarn und Nachbars-Nachbarn von s. Dabei merken wir uns in einer Queue, welche Knoten wir schon besucht haben. Wir besuchen bei jedem Knoten erst alle direkten Nachbarn und besuchen dann wiederum die Nachbarn der Nachbarn usw. Wenn wir außerdem die Knoten und Kanten, die wir besucht haben, in einem neuen Graphen speichern, erhalten wir einen Baum, in dem jeder Pfad von der Wurzel s zu einem anderen Knoten dem kürzesten Pfad im ursprünglichen Graphen entspricht (Achtung: Wenn es Kantengewichte gibt, ist der kürzeste Pfad nicht immer der günstigste Pfad). Den resultierenden Baum nennt man **Breitensuchbaum**.

Eine QUEUE (Warteschlange) ist eine listenähnliche Datenstruktur mit zwei relevanten Operationen:

- ENQUEUE fügt ein neues Element am Ende der Warteschlange ein.
- DEQUEUE entfernt das erste Element aus der Warteschlange.

Beide Operationen können in $\mathcal{O}(1)$ implementiert werden (zum Beispiel durch eine doppelt verkettete Liste).

Außerdem führen wir zwei Attribute für jeden Knoten ein:

- d ist die Pfadlänge von s zum Knoten.
- π ist der Vorgänger eines Knoten im Breitensuchbaum

Pseudocode

```
BREADTH-FIRST-SEARCH(G, s)
 1  for each vertex u ∈ A.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  ENQUEUE(Q, s)
 9  while Q ≠ ∅
10      u = DEQUEUE(Q)
11      for each v ∈ G.Adj[u]
12          if v.color = WHITE
13              v.color = GRAY
14              v.d = u.d + 1
15              v.π = u
16              ENQUEUE(Q, v)
17      u.color = BLACK
```

Der obige Algorithmus färbt die Knoten nach der folgenden Regel:

- Jeder Knoten, der noch nicht vom Algorithmus besucht wurde, ist weiß.
- Jeder Knoten, der vom Algorithmus gefunden (aber noch nicht untersucht) wurde ist grau.
- Jeder Knoten, den der Algorithmus fertig besucht hat ist schwarz.

Diese Färbungen finden wir auch im Algorithmus wieder:

- In den Zeilen 1-8 wird der Ausgangszustand hergestellt (alle Knoten sind weiß, nur s ist in der QUEUE).
- In den Zeilen 11-16 werden die Nachbarn des Knoten u festgestellt und zur QUEUE hinzugefügt.
- In Zeile 17 haben wir den Knoten u fertig besucht und färben ihn daher schwarz.

Man sollte an dieser Stelle bemerken, dass bei der Breitensuche zwar immer alle Knoten besucht werden, dass das aber nicht für die Kanten im Graphen gilt. In Zeile 11 gibt es außerdem mehrere Möglichkeiten, in welcher Reihenfolge die Nachbarn von u besucht werden. Dieser Unterschied führt dazu, dass Breitensuchbäume nicht eindeutig sind.

Korrektheit

Satz 10.1. *Der* BREADTH-FIRST-SEARCH-*Algorithmus erhält folgende Schleifeninvariante aufrecht: Wenn der Test in Zeile 9 ausgeführt wird, befiden sich nur graue Knoten in der Warteschlange.*

Die Beweistechnik der Schleifeninvarianten ist im Anhang A erläutert

Beweis. Wir beweisen die Schleifeninvariante wie üblich.

Initialisierung Zu Beginn des Algorithmus wird jeder Knoten weiß gefärbt. Dann wird der Knoten s grau gefärbt und zur QUEUE hinzugefügt. Damit ist s der einzige graue Knoten und befindet sich in der QUEUE.

Aufrechterhaltung Immer wenn ein Knoten in Zeile 13 grau gefärbt wird, wird er auch in Zeile 16 zur QUEUE hinzugefügt. Immer wenn ein Knoten in Zeile 17 schwarz gefärbt wird, wurde er vorher in Zeile 10 aus der QUEUE entfernt. Da sich in der Queue nur graue Knoten befinden, ist auch u in Zeile 10 grau. Somit kann in Zeile 12 niemals $u = v$ gelten und u wird kein zweites Mal zur QUEUE hinzugefügt.

Terminierung Jeder Knoten wird nur ein Mal besucht. Da es nur endlich viele Knoten gibt, terminiert der Algorithmus. □

Laufzeitanalyse

Jeder Knoten wird höchstens ein Mal zur QUEUE hinzugefügt bzw. daraus entfernt. Beide Operationen können in konstanter Zeit ausgeführt werden. Also benötigt das Färben $\mathcal{O}(V)$ Operationen. Außerdem wird jede Adjazenzliste $Adj.[u]$ höchstens ein Mal durchlaufen. Das ist insgesamt $\Theta(E)$ Mal. Damit ist die Laufzeit der Breitensuche in $\Theta(V + E)$.

Tiefensuche (Depth-First-Search)

Die Tiefensuche (DFS) ist ein Suchalgorithmus in Graphen, der der Breitensuche sehr ähnlich ist. Tatsächlich könnte man bei der Breitensuche die Queue durch einen Stack ersetzen und hätte damit die Tiefensuche fertig implementiert. Wir wählen hier jedoch eine leicht andere Technik.

Idee

Die Tiefensuche unterscheidet sich von der Breitensuche dadurch, dass wir nun von Anfang an versuchen, möglichst lange Pfade zu finden. Wir gucken uns also nicht erst alle Nachbarn eines Knoten an, sondern versuchen als erstes einen möglichst langen Pfad zu finden. Erst wenn wir einen Pfad nicht mehr verlängern können, gehen wir im *Backtracking* zurück, bis wir einen alternativen Pfad entlanggehen können.

Auch bei diesem Algorithmus konstruieren wir durch die Suche einen Baum, den wir Tiefensuchbaum nennen. Wir färben außerdem die Knoten wie bei der Breitensuche. Zusätzlich zur Färbung erhält jeder Knoten drei weitere Attribute:

- Einen Zeitstempel d, in dem der Zeitpunkt steht, zu dem der Knoten vom Algorithmus entdeckt wurde
- Ein Zeitstempel f, in dem der Zeitpunkt steht, zu dem der Algorithmus den Knoten fertig bearbeitet hat.
- Den Vorgängerknoten π im Tiefensuchbaum.

Pseudocode

```
Depth-First-Search(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color = WHITE
7          DFS-Visit(G, u)
```

Die Unterprozedur DFS-Visit ist dann so implementiert:

```
DFS-VISIT(G, u)
 1  time = time + 1
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]
 5      if v.color = WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK
 9  time = time + 1
10  u.f = time
```

Laufzeit

Die Initialisierung in den Zeilen 1-4 besucht jeden Knoten genau ein Mal und läuft damit in $\Theta(V)$. Jeder Aufruf von DFS-Visit färbt einen weiteren Knoten schwarz, also kann es höchstens $\mathcal{O}(V)$ Aufrufe von DFS-Visit geben. Außerdem wird auch hier jede Adjazenzliste $Adj[u]$ ein Mal durlaufen, was eine summierte Laufzeit von $\Theta(E)$ ergibt. Insgesamt ist damit die Laufzeit der Tiefensuche ebenfalls $\Theta(V + E)$.

Wälder

Sowohl die Breitensuche als auch die Tiefensuche können erweitert werden, um mehrere Zusammenhangskomponenten zu finden. Die Tiefensuche tut das in der obigen Implemenntierung sogar schon. Das Ergebnis des Algorithmus ist dann ein Teilgraph des ursprünglichen Graphen. Die Zusammenhangskomponenten des Teilgraphen sind jeweils Bäume, weswegen man den Teilgraphen auch als **Wald** bezeichnet. Im Kontext der Breiten- und Tiefensuche spricht man auch von Breitensuchwäldern und Tiefensuchwäldern.

11. Minimale Spannbäume

Minimale Spannbäume sind eine der bekanntesten Klassen von Greedy-Algorithmen. Die Problemstellung ist die folgende:

Definition 11.1. *Gegeben ist ein ungerichteter, gewichteter Graph* $G = (V, E, w)$ *mit* $w : E \to \mathbb{R}$. *Der Einfachheit halber nehmen wir an, dass es keine doppelten Kantengewichte gibt (dass* w *also injektiv ist). dann ist ein Spannbaum ein Teilgraph* $T = (V_T, E_T, w_T)$ *von* G*, der folgende Eigenschaften erfüllt:*

- T *ist ein Baum.*
- T *enthält alle Knoten von* G *(*$V = V_T$*).*
- T *enthält* $V - 1 = V_T - 1$ *Kanten (*$|E_T| = |V_T| - 1$*).*

Für einen minimalen Spannbaum gilt zusätzlich noch, dass

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

minimal ist.

Grundsätzlicher Algorithmus

Idee

Grundsätzlich lässt sich das Problem lösen, indem wir nach und nach eine Kantenmenge A aufbauen, die am Ende die Kantenmenge des minimalen Spannbaumes ist. Wir implementieren den Algorithmus dann so, dass die folgende Schleifeninvariante gilt: A **ist in jeder Iteration eine Teilmenge der Kanten des minimalen Spannbaumes.**

In jeder Iteration fügen wir dann eine weitere Kante $(u, v) \in E$ zu A hinzu, ohne dabei die Invariante zu verletzen. Eine solche Kante nennen wir **sichere Kante**. Es ist offensichtlich, dass die Invariante nicht verletzt wird, wenn wir eine sichere Kante hinzufügen (da diese ja genau so definiert ist, dass die Invariante erhalten bleibt).

Pseudocode

Durch das Grundgerüst, das oben beschrieben wurde, können wir bereits einen Teil des Algorithmus implementieren:

SPANNING-TREE($G = (V, E)$)
1 $A = \emptyset$
2 **while** A does not form a spanning tree
3 find an edge (u, v) that is safe for A
4 $A = A \cup \{(u, v)\}$

Am Pseudocode kann man schnell sehen, dass die Invariante aufrecht erhalten wird. Einen formalen Beweis sparen wir uns hier.

Damit ist das Ergebnis auf jedem Fall ein gültiger MST. Wir wissen aber noch nicht, wie wir eine sichere Kante für A finden können.

Sichere Kanten

Im Folgenden werden wir zwei Algorithmen sehen, mit denen wir sichere Kanten finden können und so einen minimalen Spannbaum berechnen können.

Um die Korrektheit der beiden folgenden Algorithmen beweisen zu können, brauchen wir zuerst ein paar Definitionen und einen Hilfssatz.

Definition 11.2. *Ein Schnitt $(S, V - S)$ eines ungerichteten Graphen $G = (V, E)$ ist eine Partition von V.*

Definition 11.3. *Eine Kante (u, v)* schneidet *einen Schnitt $(S, V - S)$, wenn entweder $(u, v) \in S \times (V - S)$ oder $(u, v) \in (V - S) \times S$ gilt.*

Definition 11.4. *Ein Schnitt respektiert eine Kantenmenge A, wenn keine Kante aus A den Schnitt schneidet.*

Definition 11.5. *Eine Kante heißt* minimale Kante *(auch:* leichte Kante*,* light edge*) bzgl. eines Schnitts, wenn sie den Schnitt schneidet und unter allen Kanten, die den Schnitt schneiden, das kleinste Kantengewicht hat.*

Satz 11.1. *Sei $G = (V, E)$ ein gewichteter, ungerichteter Graph und T ein minimaler Spannbaum für G. Sei A eine Teilmenge von E, für die $A \subseteq T$ gilt. Sei außerdem $(S, V - S)$ ein Schnitt von G, der A respektiert und (u, v) eine minimale Kante bezüglich des Schnitts $(S, V - S)$. Dann ist (u, v) sicher für A.*

Beweis. Wenn $(u, v) \in T$ ist, sind wir direkt fertig. Angenommen $(u, v) \notin T$. Das Ziel ist es nun, einen minimalen Spannbaum T' zu konstruieren, der $A \cup \{(u, v)\}$ enthält. Wenn es einen solchen Baum gibt, dann ist (u, v) nach Definition sicher.

Da $(u, v) \notin T$ ist, muss es einen eindeutigen Pfad

$$p = (u = w_1 \to w_2 \to \cdots \to w_k = v)$$

in T geben, wobei $(w_i, w_{i+1}) \in T$ für $1 \leq i < k$ ist. Wäre das nicht der Fall, wäre entweder T kein Baum oder würde nicht alle Knoten aus G enthalten. Da (u, v) eine minimale

Kante bezüglich des Schnitts $(S, V - S)$ ist, liegen u und v auf unterschiedlichen Seiten des Schnitts. Also muss es eine Kante $(x, y) \in T$ geben, die den Schnitt schneidet. Da aber A den Schnitt respektiert, ist $(x, y) \notin A$.

Da der Pfad von u zu v in T eindeutig sein muss, führt das Entfernen von (x, y) dazu, dass T in zwei Komponenten unterteilt wird. Wenn wir nun aber (u, v) hinzufügen, erhalten wir einen neuen Spannbaum $T' = T - \{(x, y)\} \cup \{(u, v)\}$. Da (u, v) eine minimale Kante bezüglich $(S, V - S)$ ist und (x, y) ebenfalls den Schnitt schneidet, ist $w(u, v) \leq w(x, y)$ und damit

$$W(T') = w(T) - w(x, y) + w(u, v) \leq W(T)$$

Damit ist T' ein minimaler Spannbaum. Da $A \subseteq T$ und $(x, y) \notin A$ ist, ist $A \subseteq T'$. Da außerdem auch $(u, v) \in T'$ ist, ist $A \cup \{(u, v)\} \subseteq T'$ und (u, v) ist damit sicher für A. □

Dieser Beweis bestätigt uns, dass es tatsächlich zu einem optimalen Ergebnis führt, wenn wir eine Greedy-Vorgehensweise zum Lösen des Problems wählen. Wir können außerdem einige Schlussfolgerungen für unseren Algorithmus ziehen:

- Der Graph $G_A = (V, A)$ ist ein *Wald*. Jede Zusammenhangskomponente von G_A ist ein Baum.
- Jede minimale Kante $(u, v) \in A$ verbindet verschiedene Zusammenhangskomponenten von G_A, da $A \cup \{(u, v)\}$ azyklisch sein muss.
- Die äußere Schleife im allgemeinen Algorithmus wird $|V| - 1$ Mal ausgeführt, da bei jeder Iteration genau eine Kante zum MST hinzugefügt wird und der MST insgesamt $|V| - 1$ Kanten hat.

Bevor wir die Algorithmen zum Finden von sicheren Kanten kennenlernen, machen wir noch folgende Feststellung:

Korollar 11.1. *Sei $G = (V, E)$ ein ungerichteter, gewichteter Graph und T ein minimaler Spannbaum für G. Sei $A \subseteq E$ mit $A \subseteq T$ und $C = (V_C, E_C)$ eine Zusammenhangskomponente des Waldes $G_A = (V, A)$. Sei zuletzt (u, v) eine minimale Kante bezüglich $(V_C, V - V_C)$. Dann ist (u, v) für A sicher.*

Beweis. Der Schnitt $(V_C, V - V_C)$ respektiert A (A definiert die Zusammenhangskomponenten von G_A) und (u, v) ist eine minimale Kante für diesen Schnitt. Damit ist (u, v) für A sicher. □

Algorithmus von Kruskal

Im Algorithmus von Kruskal ist A (im allgemeinen Fall) ein Wald.

Idee

Im Algorithmus von Kruskal fügen wir in jeder Iteration eine Kante zu A hinzu, die zwei Zusammenhangskomponenten von A verbindet und ein minimales Gewicht hat. Dass diese Vorgehensweise zu einem korrekten Ergebnis führt, zeigt die folgende Überlegung:

Seien C_1, C_2 zwei Bäume, die durch eine Kante (u, v) verbunden sind. Da dann (u, v) eine minimale Kante sein muss, die C_1 mit einem anderen Baum verbindet, ist (u, v) für C_1 sicher.

Zur Implementierung brauchen wir allerdings noch die Datenstruktur Disjoint-Set. Ein Disjoint-Set ist eine Menge von Knotenmengen mit den folgenden Operationen:

- Make-Set(u) initialisiert eine Menge, die nur den Knoten u enthält.
- Find-Set(u) gibt ein Objekt zurück, das die Menge von u repräsentiert. So kann durch Find-Set(u) = Find-Set(v) verglichen werden, ob u und v zu derselben Menge gehören.
- Union(u, v) vereinigt die Knotenmenge, die u enthält mit der Knotenmenge, die v enthält.

Pseudocode

```
Kruskal(G = (V, E, w))
1  A = ∅
2  for each vertex v ∈ V[G]
3      Make-Set(v)
4  sort edges of E into nondecreasing order by weight w
5  for each edge (u, v) ∈ E in nondecreasing order by weight w
6      if Find-Set(u) ≠ Find-Set(v)
7          A = A ∪ {(u, v)}
8          Union(u, v)
9  return A
```

Korrektheit

Die Korrektheit des Algorithmus folgt direkt aus dem obigen Satz.

Laufzeit

- Die Initialisierung in Zeile 1 hat eine Laufzeit von $\mathcal{O}(1)$.
- Das Sortieren der Kanten braucht $\mathcal{O}(E \log E)$.

- Die äußere Schleife führt $\mathcal{O}(E)$ Find-Set- und Union-Operationen und $|V|$ Make-Set-Operationen aus. Das braucht insgesamt

$$\mathcal{O}(V+E)\cdot\mathcal{O}(\log E)=\mathcal{O}(E\log V)$$

Damit ist die Gesamtlaufzeit des Algorithmus von Kruskal in $\mathcal{O}(E\log V)$.

Algorithmus von Prim

Im Algorithmus von Prim ist A nun kein Wald mehr, sondern immer ein einziger Baum.

Idee

Wir beginnen bei einer Wurzel r und vergrößern dann in jedem Schritt den Baum A um eine minimale Kante, die A mit einem isolierten Knoten aus $G_A=(V,A)$ verbindet. Nach dem obigem Korollar werden so nur sichere Kanten hinzugefügt und wir erhalten einen minimalen Spannbaum.

Um in jedem Schritt eine minimale Kante zu finden, brauchen wir einen kleinen Trick. Wir könnten auch einfach in jedem Durchgang jede einzelne Kante überprüfen, das wäre aber sehr ineffizient. Daher erstellen wir eine Min-Priority-Queue (ein Min-Heap), die die Knoten enthält, die noch *nicht* im MST enthalten sind. In der Queue wird die Priorität der Knoten durch einen Schlüssel bestimmt:

Für $v\in V$ sei $key[v]$ das minimale Kantengewicht der Kanten, die v mit einem anderen Knoten in A verbinden. Gibt es keine solche Kante, sei $key[v]=\infty$.

Sei außerdem $\pi[v]$ der Elternknoten von v im Baum. Während der Ausführung des Algorithmus konstruieren wir A implizit durch

$$A=\{(v,\pi[v]):v\in V-\{r\}-Q\}$$

Nach der Terminierung des Algorithmus wird die Min-Priority-Queue leer sein und wir erhalten durch diese Definition einen minimalen Spannbaum.

Pseudocode

```
PRIM(G = (V, E, w), r)
 1  for each u ∈ V
 2        key[u] = ∞
 3        π[u] = NIL
 4  key[r] = 0
 5  Q = MAKE-MIN-PRIORITY-QUEUE(V)
 6  while Q ≠ ∅
 7        u = EXTRACT-MIN(Q) // Element mit kleinstem Schlüssel
 8        for each v ∈ Adj[u]
 9              if v ∈ Q  &  w(u, v) < key[v]
10                    π[v] = u
11                    key[v] = w(u, v)
```

Durch die Min-Priority-Queue müssen wir nicht jedesmal alle Knoten überprüfen, sondern können direkt denjenigen wählen, der das geringste Kantengewicht hat.

Korrektheit

Wir beweisen die Korrektheit über eine dreiteilige Schleifeninvariante:

1. $A = \{(v, \pi[v]) : v \in V - \{r\} - Q\}$
2. Die Kanten, die bereits zum MST hinzugefügt wurden, sind $V - Q$.
3. Für alle $v \in Q$ gilt: Wenn $\pi[v] \neq \text{NIL}$ ist, dann ist $key[v] < \infty$ und $key[v]$ bezeichnet das Gewicht einer minimalen Kante $(v, \pi[v])$, die v mit einem Knoten im MST zu verbindet.

Initialisierung

1. Zu Beginn ist $A = \emptyset = V - \{r\} - Q$.
2. Es ist $A = \emptyset = V - Q$.
3. Zu Beginn ist $\pi[v] = \text{NIL}$ für alle $v \in V$.

Damit gilt die Invariante am Anfang.

Aufrechterhaltung

1. In Zeile 7 finden wir immer außer in der ersten Iteration ein $u \in Q$, welches den Schnitt $(V - Q, Q)$ schneidet. Nachdem wir u aus Q entfernen, fügen wir es zu der Knotenmenge $V - Q$ im Baum hinzu. Damit wird $(u, \pi[u])$ implizit zu A hinzugefügt.

2. Siehe 1.
3. Die Schleife in den Zeilen 8-11 ändert den Schlüssel und das π-Feld jedes Nachbarn von u, nicht aber von den Knoten, die schon im Baum sind.

Damit gelten alle drei Teile der Invariante auch im nächsten Schleifendurchlauf.

Laufzeitanalyse

Die Laufzeit des Algorithmus von Prim hängt stark von der Implementierung der Min-Priority-Queue ab. Wenn wir einen Binary-Min-Heap verwenden, dann ergibt sich für die Laufzeit:

- Build-Min-Heap, bzw. Make-Min-Priority-Queue dominiert die Initialisierung in den Zeilen 1-5 und läuft in $\mathcal{O}(V)$.
- Der Rumpf der **while** -Schleife wird $\mathcal{O}(V)$ Mal ausgeführt. In jedem Durchlauf wird Extract-Min in $\mathcal{O}(\log V)$ ausgeführt, was eine Gesamtlaufzeit der Schleife von $\mathcal{O}(V \log V)$ ergibt.
- Die **for** -Schleife in den Zeilen 8-11 wird **insgesamt** in $\mathcal{O}(E)$ ausgeführt, da die Summe der Adjazenzlisten $2 \cdot |E|$ ist. Da allerdings in jedem Schleifendurchlauf der Schlüssel des Knoten v geändert wird, muss auch in jedem Durchlauf nochmal Min-Heapify aufgerufen werden, dessen Laufzeit in $\mathcal{O}(\log n)$ liegt.

Das ergibt eine Gesamtlaufzeit von $\mathcal{O}(E \log V + V \log V)$. Durch eine andere Implementierung der Warteschlange kann man die Laufzeit aber noch verbessern. Verwendet man beispielsweise Fibonacci-Heaps, lässt sich die Laufzeit auf $\mathcal{O}(E + V \log V)$ verbessern.

12. Kürzeste Pfade in Graphen

Problemstellung

In diesem Kapitel beschäftigen wir uns mit dem Problem, in gerichteten, gewichteten Graphen den optimalen Pfad von einem Knoten zu einem anderen Knoten zu berechnen. Formal definieren wir dieses Problem folgendermaßen:

Definition 12.1. *Sei $G = (V, E)$ ein gerichteter Graph. Sei außerdem $w : E \to \mathbb{R}$ eine Gewichtungsfunktion der Kanten. Gegeben sind zwei Knoten $s, d \in V$. Dann suchen wir den gerichteten Pfad mit dem niedrigsten Gesamtgewicht von s nach d.*

Bei Pfaden mit minimalem Gesamtgewicht spricht man oft auch von *kürzesten* Pfaden. Damit ist **nicht** ein Pfad mit minimal vielen Kanten gemeint.

Dazu machen wir zunächst einige Feststellungen:

- Wir bezeichnen den kürzesten Pfad von s nach d mit $\delta(s, d)$.
- Auf dem kürzesten Pfad von s nach d sind die Teilpfade zwischen zwei Knoten bereits minimal (Beweis siehe unten).
- Enthält ein Graph Zyklen mit negativem Gesamtgewicht, gibt es möglicherweise keinen kürzesten Pfad von s nach d. Ist dies der Fall, wird der Algorithmus erkennen, dass es negative Zyklen im Graphen gab.
- Ein kürzester Pfad kann auch keine positiv gewichteten Zyklen enthalten (dann wäre es kein kürzester Pfad). Daher kann ein solcher Pfad höchstens $n - 1$ Knoten enthalten.

Lemma 12.1. *Sei $p = (v_1, v_2, \ldots, v_k)$ ein kürzester Pfad von v_1 nach v_k. Sei weiterhin $p_{ij} = (v_i, v_{i+1}, \ldots, v_j)$ für $1 \leq i \leq j \leq k$ ein Teilpfad von v_i nach v_j. Dann ist p_{ij} der kürzeste Pfad von v_i nach v_j.*

Beweis. Wir zerlegen p in

$$v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k.$$

Dann ist das Gesamtgewicht des Pfades $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. Angenommen es gäbe einen Teilpfad p'_{ij} von v_i nach v_j mit $w(p'_{ij}) < w(p_{ij})$. Dann gäbe es einen Pfad

$$v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k.$$

von v_1 nach v_k mit dem Gesamtgewicht $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk}) < w(p)$. Dies ist ein Widerspruch zu der Annahme, dass p bereits optimal ist. Also muss auch der Teilpfad v_i nach v_j optimal sein. □

Kürzeste Pfad-Bäume

Bevor wir einen Algorithmus kennen lernen, der die kürzesten Pfade von einem Knoten zu anderen Knoten berechnet, wollen wir zunächst noch definieren, wie das Ergebnis des Algorithmus aussehen soll.

Definition 12.2. *Ein kürzester Pfad-Baum $G' = (V', E')$ ausgehend von einem Knoten s ist ein gerichteter Teilgraph von G mit $V' \subseteq V$ und $E' \subseteq E$, für den gilt:*

- *V' ist eine Menge von Knoten, die von s aus in G erreichbar sind.*
- *G' ist ein Baum mit der Wurzel s.*
- *Für alle $v \in V'$ ist der Pfad von der Wurzel s zu v in G' der kürzeste Pfad von s zu v in G.*

Da wir normalerweise nicht nur die Länge eines kürzesten Pfades, sondern auch dessen Struktur wissen wollen, wird der Algorithmus einen kürzesten Pfad-Baum generieren.

Wir stellen an dieser Stelle schon einmal fest, dass kürzeste Pfad-Bäume nicht eindeutig sind.

Der Bellman-Ford-Algorithmus

Idee

Das grundsätzliche Konzept des Bellman-Ford-Algorithmus wendet direkt die obigen Feststellungen zu kürzesten Pfaden an. Der Algorithmus soll Schritt für Schritt die kürzesten Pfade aus einem Graphen G finden. Tatsächlich löst dieser Algorithmus mehr als das ursprüngliche Problem fordert. Wir wollen eigentlich nur den kürzesten Pfad von einem Knoten zu einem anderen. Stattdessen berechnet dieser Algorithmus die kürzesten Pfade von einem Knoten s zu allen von dort aus erreichbaren Knoten. In jedem Schritt tut der Algorithmus dann folgendes:

- Für jeden Knoten $v \in V$ speichern wir einen Wert $d[v]$, der eine obere Schranke für das Gesamtgewicht des kürzesten Pfades von s nach v darstellt. Dieser Wert wird als Schätzung für den Pfad bezeichnet (engl. *shortest path estimate*).
- Am Anfang ist $d[v] = \infty$ für $v \neq s$ und $d[s] = 0$.
- Wir speichern außerdem in $\pi[v]$ den Vorgänger von v im kürzesten Pfad.

- In jedem Schritt wird versucht, die Schätzung zu verbessern und die Vorgänger entsprechend aktualisiert.

Einen solchen Schritt im Algorithmus bezeichnen wir als **Relaxierung**.

Die Idee hinter diesem Aufbau ist, dass der Algorithmus eine Invariante aufrecht erhält, dass in $d[v]$ immer das Gesamtgewicht des kürzesten Weges von s nach v enthält (Beweis s.u.).

Pseudocode

Zuerst betrachten wir den Pseudocode für einen Relaxierungsschritt:

```
RELAX(u, v, w)
1  if d[v] > d[u] + w(u, v)
2      d[v] = d[u] + w(u, v)
3      π[v] = u
```

Die Bedeutung des Codes lässt sich leicht zusammenfassen: Wenn wir den Pfad nach v verbesern können, indem wir zuerst nach u und dann erst nach v gehen, dann tun wir das auch.

Mit diesen Voraussetzungen können wir nun den Bellman-Ford-Algorithmus aufschreiben:

```
BELLMAN-FORD(G, w, s)
1  INITIALISE-SINGLE-SOURCE(G, s)
2  for i = 1 to |V[G]| − 1
3      for each edge (u, v) ∈ E[G]
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ E[G]
6      if d[v] > d[u] + w(u, v)
7          return FALSE
8  return TRUE
```

Der Algorithmus besteht aus zwei Teilen. In Zeile 2-4 werden die kürzesten Pfade gefunden. Die Schleife in den Zeilen 5-7 ist die Prüfung, ob es negative Zyklen im Graphen gibt. Dass so negative Zyklen erkannt werden, kann man sich intuitiv leicht erklären: Wir haben vorher alle Pfade gefunden, die i Kanten benutzen. Da i bis $|V| - 1$ hochzählt, müssen alle kürzesten Pfade dabei gefunden worden. Wenn es jetzt trotzdem noch möglich ist, einen Pfad zu verbessern, dann muss es einen negativen Zyklus geben. Diese Beobachtung reicht aber noch nicht für einen formalen Beweis (s.u.).

Durch die Formulierung des Pseudocodes ist an dieser Stelle aber schon relativ klar, wie unsere Invariante aussehen wird: Nach jedem Schleifendurchlauf haben wir alle kürzesten Pfade mit maximal i Kanten gefunden.

Der Algorithmus ist in der oben formulierten Variante noch nicht eindeutig. In Zeile 3 gibt es mehrere Möglichkeiten, in welcher Reihenfolge die Kanten abgearbeitet werden. Diese Reihenfolge beeinflusst sehr stark, wie der minimale Pfad-Baum am Ende aussieht und wie schnell kürzeste Pfade gefunden werden. Dies ist eine Schwäche des Bellman-Ford-Algorithmus, die im allgemeinen nicht korrigiert werden kann.

Laufzeitanalyse

Die Laufzeitanalyse des Bellman-Ford-Algorithmus ist sehr einfach: Offensichtlich wird die Laufzeit durch die beiden geschachtelten Schleifen in Zeile 2 und 3 dominiert. Die Äußere Schleife macht dabei $\mathcal{O}(|V|)$ Durchläufe, die innere jeweils $\mathcal{O}(|E|)$. Insgesamt ergibt dies eine Laufzeit von $\mathcal{O}(|V| \cdot |E|)$.

Korrektheit

Wir werden die Korrektheit des Bellman-Ford-Algorithmus beweisen. Dazu brauchen wir zuerst einige Hilfssätze:

Lemma 12.2 (Upper Bound Property)**.** *Sei $G = (V, E)$ ein gerichteter, gewichteter Graph mit der Gewichtungsfunktion $w : E \to \mathbb{R}$. Sei weiterhin $s \in V$ ein Knoten. Dann gilt im Bellman-Ford-Algorithmus die Invariante $d[v] \geq \delta(s, v)$ für alle $v \in V$. Des Weiteren wird $d[v]$ nicht mehr verändert, nachdem es $\delta(s, v)$ erreicht hat.*

Beweis. Wir führen den Beweis durch Induktion über die Anzahl der Relaxierungen durch:

Induktionsanfang Nach der Initialisierung gilt $d[v] = \infty \geq \delta(s, v)$.

Induktionsannahme Angenommen es gilt $d[x] \geq \delta(s, x)$ für alle $x \in V$, die vor einer Relaxierung verarbeitet wurden. Wir wollen nun zeigen, dass die Aussage auch nach der Relaxierung der nächsten Kante $(u, v) \in V$ gilt.

Induktionsschritt Wir unterscheiden zwei Fälle:

Fall 1 $d[v]$ wird verändert. Dann gilt

$$d[v] = d[u] + w(u, v) \overset{\text{I.A.}}{\geq} \delta(s, u) + w(u, v) \geq \delta(s, v)$$

Fall 2 $d[v]$ wird nicht verändert. Dann gilt nach I.A. direkt $d[v] \geq \delta(s, v)$.

Der zweite Teil von Lemma 12.2 ergibt sich direkt aus der Definition der Relaxion. Da wir gerade bereits gezeigt haben, dass $d[v] \geq \delta(s, v)$ gilt, müssen wir hier nur noch feststellen, dass bei der Relaxierung der Wert von $d[v]$ nicht vergrößert wird. □

Lemma 12.3 (Konvergenzeigenschaft)**.** *Sei $G = (V, E)$ ein gerichteter, gewichteter Graph mit der Gewichtungsfunktion $w : E \to \mathbb{R}$. Sei außerdem $s \in V$ ein Knoten. Sei nun $s \rightsquigarrow u \to v$ ein kürzester Pfad in G für zwei Knoten u und v. Nachdem dann im Bellman-Ford-Algorithmus eine Folge von Relaxierungen ausgeführt wurde, die* RELAX(u, v, w) *enthält, gilt: Wenn vor der Relaxierung $d[u] = \delta(s, u)$ galt, gilt danach auch $d[v] = \delta(s, v)$.*

Beweis. Durch Lemma 12.2 wissen wir bereits, dass $d[u] = \delta(s, u)$ auch nach der Relaxierung der Kante (u, v) gilt. Es gilt außerdem

$$d[v] \leq d[u] + w(u, v) = \delta(s, u) + w(u, v) = \delta(s, v)$$

Die erste Ungleichheit ergibt sich aus der Definition der Relaxierung. Die letzte Gleichung geht aus der Annahme hervor, dass $s \rightsquigarrow u \to v$ ein kürzester Pfad ist. Da wir nun $d[v] \leq \delta(s, v)$ gezeigt haben und nach Lemma 12.2 auch $d[v] \geq \delta(s, v)$ gilt, ist nach der Relaxierung $d[v] = \delta(s, v)$. □

Lemma 12.4 (Pfadrelaxierungseigenschaft)**.** *Sei $G = (V, E)$ ein gerichteter, gewichteter Graph mit der Gewichtungsfunktion $w : E \to \mathbb{R}$. Sei außerdem $s \in V$ ein Knoten und $p = (v_1, v_2, \ldots, v_k)$ ein kürzester Pfad von $s = v_0$ nach v_k. Wenn dann im Bellman-Ford-Algorithmus eine Folge von Relaxierungen durchgeführt wird, die die Kanten $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ in dieser Reihenfolge relaxiert, gilt danach $d[v_k] = \delta(s, v_k)$. Dies gilt unabhängig von anderen Relaxierungen.*

Beweis. Wir beweisen Lemma 12.4 durch Induktion über die Anzahl der relaxierten Kanten in p. Nachdem die i-te Kante von p relaxiert wurde, gilt $d[v_i] = \delta(s, v_i)$.

Induktionsanfang : Für $i = 0$ gilt

$$d[v_0] = d[s] = 0 = \delta(s, s)$$

Nach Lemma 12.2 ändert sich dieser Wert im folgenden nicht mehr.

Induktionsannahme Angenommen es gilt $d[v_{i-1}] = \delta(s, v_{i-1})$. Wir relaxieren nun irgendwann die Kante (v_{i-1}, v_i).

Induktionsschritt : Nach der Konvergenzeigenschaft gilt nach der Relaxierung $d[v_i] = \delta(s, v_i)$. Nach Lemma 12.2 ändert sich auch dieser Wert nicht mehr. □

Jetzt sind wir so weit, dass wir die Korrektheit des Algorithmus beweisen können.

Lemma 12.5. *Sei $G = (V, E)$ ein gerichteter, gewichteter Graph mit der Gewichtungsfunktion $w : E \to \mathbb{R}$. Sei weiterhin $s \in V$ ein Knoten. Wir nehmen an, dass in G keine negativen Zyklen existieren, die von s aus erreicht werden können. Dann ist nach $|V| - 1$ Iterationen des Bellman-Ford-Algorithmus $d[v] = \delta(s, v)$ für alle Knoten $v \in V$, die von s aus erreichbar sind.*

Beweis. Sei $p = (v_0, v_1, \ldots, v_k)$ ein kürzester, azyklischer Pfad von $v_0 = s$ nach $v_k = v$. Dieser Pfad hat höchstens $|V| - 1$ Kanten, also ist $k \leq |V| - 1$. In jeder der $|V| - 1$ Iterationen des Algorithmus werden alle Kanten von E relaxiert. Darunter ist also auch in der i-ten Iteration die Kante (v_{i-1}, v_i). Aufgrund der Pfadrelaxierungseigenschaft (Lemma 12.4) gilt also am Ende sicher

$$d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v).$$

□

Damit wissen wir schon mal, dass die Pfadlänge, die berechnet wird, stimmt. Nun müssen wir noch zeigen, dass auch der kürzeste Pfad-Baum korrekt ist.

Lemma 12.6 (Vorgänger-Eigenschaft)**.** *Sei $G = (V, E)$ ein gerichteter, gewichteter Graph mit der Gewichtungsfunktion $w : E \to \mathbb{R}$. Sei weiterhin $s \in V$ ein Knoten. Wir nehmen an, dass G keine negativen Zyklen enthält, die von s aus erreichbar sind. Nachdem dann Relaxierungen ausgeführt wurden, die in $d[v] = \delta(s, v)$ für alle $v \in V$ resultieren, ist G_π der kürzeste Pfad-Baum mit der Wurzel s.*

Beweis. Für den Beweis reicht es zu zeigen, dass die Eigenschaften eines kürzesten Pfad-Baumes für G_π gelten:

- Die Knoten, die von s aus erreichbar sind, haben am Ende ein $d[v] < \infty$. Ein Knoten $v \in V - \{s\}$ hat ein $d[v] < \infty$ genau dann, wenn $\pi[v] \neq \text{nil}$ ist (durch die Relaxierung). Daher sind alle Knoten in V_π von s aus erreichbar.

- Wir müssen noch zeigen, dass ein Pfad $p = (s = v_0, v_1, \ldots, v_k = v)$ in G_π ein kürzester Pfad in G ist. Dazu stellen wir fest, dass für $i = 1, 2, \ldots, k$ sowohl $d[v_i] = \delta(s, v_i)$ als auch $d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i)$ gilt. Es ist also

$$w(v_{i-1}, v_i) \leq \delta(s, v_i) - \delta(s, v_{i-1})$$

Wenn wir nun die Kantengewichte des Pfades p aufsummieren erhalten wir

$$\begin{aligned} w(p) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) \\ &\leq \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) \\ &= \delta(s, v_k) - \delta(s, v_0) = \delta(s, v_k) \end{aligned}$$

Es ist also $w(p) \leq \delta(s, v_k)$. Da wir wissen, dass $\delta(s, v_k)$ eine untere Schranke für das Gesamtgewicht von p ist, wissen wir, dass für p $w(p) = \delta(s, v_k)$ gilt. Das beweist Lemma 12.6.

□

Als letztes müssen wir für die Korrektheit noch den Fall behandeln, dass ein Graph negative Zyklen enthält. Dies tun wir mit Lemma 12.7:

Lemma 12.7. *Sei $G = (V, E)$ ein gerichteter, gewichter Graph mit der Gewichtungsfunktion $w : E \to \mathbb{R}$. Sei außerdem $s \in V$ ein Knoten. Wir betrachten einen kürzesten Pfad $p = (v_1, v_2, \ldots, v_k)$ von $s = v_0$ zu v_k.*

- *Wenn G keine negativen Zyklen enthält, die von s aus erreicht werden können, gibt der Algorithmus* TRUE *zurück und es gilt $d[v] = \delta(s, v)$ für alle Knoten $v \in V$, die von s aus erreichbar sind. Dann berechnet der Bellman-Ford-Algorithmus einen kürzesten Pfad-Baum mit der Wurzel s.*
- *Wenn G einen negativ gewichteten Zyklus enthält, der von s aus erreichbar ist, gibt der Algorithmus* FALSE *zurück.*

Beweis. Wir unterscheiden zwei Fälle:

Fall 1 Angenommen der Graph enthält keinen negativ gewichteten Zyklus, der von s aus erreichbar ist. Dann gilt Lemma 12.5 und es gilt $d[v] = \delta(s, v)$ für alle von s aus erreichbaren Knoten v. Für alle nicht erreichbaren Knoten bleibt $d[v] = \delta(s, v) = \infty$. Dies folgt aus Lemma 12.2, da immer $\infty = \delta(i, v) \leq d[v]$ gilt. Des Weiteren gilt durch die Vorgängergraph-Eigenschaft (Lemma 12.6), dass der Algorithmus einen kürzesten Pfad-Baum G_π produziert. Da außerdem

$$d[v] = \delta(s, v) \leq \delta(s, u) + w(u, v) = d[u] + w(u, v)$$

gilt, trifft die Bedingung in Zeile 6 niemals zu und der Algorithmus gibt TRUE aus.

Fall 2 Angenommen G enthält einen von s aus erreichbaren negativen Zyklus. Sei $c = (v_0, v_1, \ldots, v_k)$ mit $v_0 = v_k$ ein solcher Zyklus. Angenommen der Bellman-Ford-Algorithmus würde trotzdem TRUE ausgeben. Dann wäre $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_1)$ für $0 \leq i \leq k$. Da es sich um einen negativen Kreis handelt, ist die Summe der Kantengewichte negativ:

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$$

Es ist außerdem

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k}(d[v_{i-1}] + w(v_{i-1}, v_i)) = \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Da jeder Knoten im Zyklus c genau ein Mal auftritt, ist $\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}]$. Wir können also oben ersetzen:

$$\begin{aligned} \sum_{i=1}^{k} d[v_i] &\leq \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i) \\ \Rightarrow \sum_{i=1}^{k} d[v_i] &\leq \sum_{i=1}^{k} d[v_i] + \sum_{i=1}^{k} w(v_{i-1}, v_i) \\ \Rightarrow 0 &\leq 0 + \sum_{i=1}^{k} w(v_{i-1}, v_i) \end{aligned}$$

Die letzte Ungleichung sagt aber gerade aus, dass die Summe der Kantengewichte im Zyklus nicht negativ ist. Das ist ein Widerspruch. Also kann der Algorithmus in diesem Fall nicht TRUE ausgeben. □

Algorithmus von Dijkstra

Der Bellman-Ford Algorithmus ist ein vergleichsweise einfacher Algorithmus zum Finden von kürzesten Pfaden. Er hat allerdings einen kleinen Schönheitsfehler: Es werden sehr viele Relaxierungen ausgeführt, die eigentlich nicht nötig wären. Der Algorithmus von Dijkstra versucht dies zu verbessern und möglichst nur die Relaxierungen auszuführen, die auch nötig sind.

Idee

Der Algorithmus von Dijkstra hat eine Einschränkung: Er funktioniert nur, wenn alle Kantengewichte positiv sind. Wir werden gleich sehen, wieso das so ist. Im Algorithmus gehen wir folgendermaßen vor:

- Wir merken uns die Knoten S, zu denen wir bereits den kürzesten Pfad kennen.
- Wir fügen in jeder Iteration einen neue Knoten $u \in V - S$ zu S, und führen dann Relaxierungen auf allen Kanten aus, die aus u herausführen.
- Wir fügen in jedem Schritt den *günstigsten* Knoten hinzu (den Knoten mit dem kleinsten Kantengewicht, der eine Kante von der Menge S in die Menge $V - S$ hat. Man spricht auch vom *minimum shortest path estimate*). Dazu verwenden wir eine Min-Priority-Queue Q, in der die Knoten gespeichert werden, die noch nicht in S sind.
- Immer wenn wir Relaxierungen durchführen, müssen auch die *shortest path estimates* (d-Werte) aktualisiert werden.

Pseudocode

```
Dijkstra(G, w, s)
1  Initialise-Single-Source(G, s)
2  S = ∅
3  Q = Make-Min-Priority-Queue(V[G])
4  while Q ≠ ∅
5        u = Extract-Min(Q)
6        S = S ∪ {u}
7        for each vertex v adjacent to u
8              Relax(u, v, w)
```

Korrektheit

Bevor wir tatsächlich die Korrektheit beweisen, betrachten wir zunächst die folgende Schleifeninvariante: Am Anfang der **while** -Schleife in Zeile 4 gilt $Q = V - S$.

Beweis. Wir beweisen die Invariante wie üblich:

Initialisierung Die Invariante gilt offensichtlich vor dem ersten Durchlauf der Schleife.

Aufrechterhaltung In jedem Schleifendurchlauf entfernen wir mit Extract-Min ein Element aus $Q = V - S$ und fügen es zu S hinzu. Aufgrund unserer Definition der Min-Priority-Queue ist u der Knoten mit der kleinsten Schätzung $d[u]$. Danach werden noch alle Kanten (u, v) relaxiert. Dabei werden weder Q noch S verändert.

Terminierung Dass der Algorithmus terminiert ist offensichtlich, da $|Q|$ zu Beginn endlich ist.

□

Satz 12.1. *Wird der Algorithmus von Dijkstra auf einem gerichteten, gewichteten Graphen $G = (V, E)$ mit der nicht negativen Gewichtungsfunktion $w : E \to \mathbb{R}$ und einem Quellknoten s ausgeführt, terminiert er mit $d[u] = \delta(s, u)$ für alle $u \in V$.*

Beweis. Zuerst stellen wir fest, dass der Algorithmus sicher terminiert. Dies geht daraus hervor, dass wir nur mit endlichen Größen arbeiten.

Wir müssen dann noch zeigen, dass jedes Mal, wenn ein Knoten u zur Menge S hinzugefügt wird, $d[u] = \delta(s, u)$ gilt. Dies zeigen wir, indem wir die Behauptung wie eine Schleifeninvariante beweisen:

Initialisierung Am Anfang gilt $S = \emptyset$. Daher ist hier die Aussage trivial und gilt.

Aufrechterhaltung Angenommen die Behauptung gilt nicht. Dann sei u der erste Knoten, der mit $d[u] \neq \delta(s, u)$ zu S hinzugefügt wird (der erste Knoten, bei dem der Algorithmus einen Fehler gemacht hat). Wir betrachten nun den Zeitpunkt, zu dem u zu S hinzugefügt werden soll. Wir wissen zu diesem Zeitpunkt, dass $S \neq \emptyset$ ist und es definitiv einen Pfad von s zu u gibt (sonst wäre $d[u] = \infty = \delta(s, u)$). Damit gibt es auch einen kürzesten Pfad p von s nach u. Bevor u zu S hinzugefügt wird, verbindet der Pfad p einen Knoten in S mit einem Knoten in $V - S$.

Sei nun y ein Knoten im Pfad p, sodass $y \in V - S$ ist und $x \in S$ der Vorgänger von y in p ist. Dann wissen wir dadurch, dass u nach Annahme der erste Knoten mit $d[u] \neq \delta(s, u)$ ist und aufgrund der Konvergenzeigenschaft, dass $d[y] = \delta(s, y)$ ist. Da es keine negativen Kanten im Graphen gibt, gilt außerdem

$$\delta(s, y) \leq \delta(s, u)$$

und durch die *upper bound property* (Lemma 12.2):

$$d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u].$$

Da aber sowohl u als auch y vor dem Hinzufügen von u in $V - S$ waren, muss $d[u] \leq d[y]$ gelten (wir wählen ja immer den Knoten mit kleinstem d und wählen im aktuellen Schritt nach Annahme gerade den Knoten u). Damit muss die obige Ungleichung mit Gleichheit erfüllt sein:

$$d[y] = \delta(s, y) = \delta(s, u) = d[u]$$

Dies ist aber ein Widerspruch zu unserer Wahl von u. Das beweist die Korrektheit des Algorithmus. □

Laufzeit

Die Laufzeit vom Algorithmus von Dijkstra hängt sehr stark von der Wahl der Datenstruktur für die MIN-PRIORITY-QUEUE ab, da sich in jedem Durchlauf die Schlüssel

für die Queue ändern. Wir können aber bereits sagen, dass die **while** -Schleife $\Theta(V)$ mal durchlaufen wird. Die innere **for** -Schleife wird außerdem **insgesamt** $\Theta(E)$ mal durchlaufen (jede Kante (u, v) genau ein Mal, nämlich dann, wenn u in der **while** -Schleife untersucht wird). Wir können also an dieser Stelle bereits sagen, dass die Laufzeit des Algorithmus von Dijkstra von unten durch $\Omega(|V| + |E|)$ beschränkt ist. Diese Laufzeit wird aber nur erreicht, wenn Extract-Min in konstanter Zeit ausgeführt werden kann.

13. Maximale Flüsse in Netzwerken

Es gibt viele Problemstellungen der echten Welt, die auf eine relativ einfache Klasse von Graphen zurückgeführt werden können. Dazu gehören zum Beispiel:

- Wie viel Wasser kann durch ein Rohrsystem mit unterschiedlich großen Rohren geleitet werden?
- Wie viele Daten können in einer gegebenen Zeit höchstens über ein Netzwerk versendet werden?
- Wie viele Fahrzeuge können in einer gegebenen Zeit höchstens von A nach B fahren?
- ...

All diese Probleme haben bestimmte Eigenschaften gemeinsam: Es geht darum, möglichst viel von etwas von einem Ort A zu einem anderen Ort B zu leiten, wobei aber jeder Teil der Leitung (einzelne Rohre, eine Straße) nur eine begrenzte Kapazität hat (unterschiedlich große Rohre, unterschiedlich breite Straßen). Dies können wir abstrahieren und auf Graphen interpretieren, indem wir Kanten als Rohre, Straßen oder Leitungen in Netzwerken interpretieren und Knoten als Verbindungsstücke oder Kreuzungen. Dann können wir uns für so einen gerichteten, gewichteten Graphen fragen: Was ist der Maximalwert, der durch dieses Netzwerk gesendet werden kann. Dieses Problem nennt man das **Maximum-Flow Problem**.

Definition

Um von dem konkreten Anwendungsfall (Rohre, Straßen) zu abstrahieren, brauchen wir zunächst eine geeignete Terminologie, die ein gleichermaßen ein Rohrsystem oder ein Straßennetzwerk beschreiben kann.
Definition 13.1 (Flussnetzwerk). *Ein Flussnetzwerk ist ein gerichteter, gewichteter Graph $G = (V, E)$ mit einer Gewichtungsfunktion $w : E \to \mathbb{R}^+$ und einer Kapazitätsfunktion $c : V \times V \to \mathbb{R}^+$ mit*

$$c(u, v) = \begin{cases} w(u, v) & (u, v) \in E \\ 0 & \textit{sonst} \end{cases}$$

Ein Flussnetzwerk muss folgende Eigenschaften erfüllen:

- *Es gibt eine **Quelle** s, die keine eingehenden Kanten hat.*

- *Es gibt eine **Senke** t, die keine ausgehenden Kanten hat.*
- *Für alle Knoten $v \in V - \{s,t\}$ gibt es einen Pfad $s \rightsquigarrow v \rightsquigarrow t$. Daraus folgt, dass es nur genau eine starke Zusammenhangskomponente in G gibt.*

Definition 13.2 (Fluss)**.** *Ein Fluss ist eine Funktion $f : V \times V \to \mathbb{R}$, welche die folgenden Eigenschaften erfüllt:*

- *Der Fluss ist **kapazitätsbeschränkt** (*capacity constraint*). Es gilt:*

$$\forall u,v \in V : f(u,v) \leq c(u,v)$$

Der Fluss respektiert also die Kapazitäten des Netzwerkes.

- *Es gilt eine **inverse Symmetrie** (*skew symmetry*):*

$$\forall u,v \in V : f(u,v) = -f(v,u)$$

Wir werden sehen, dass dies bedeutet, dass die Flussstärke entgegen der Flussrichtung negativ ist.

- *Die **Flusserhaltung** (*flow conservation*) muss gewährleistet sein:*

$$\forall u \in V - \{s,t\} : \sum_{u \in V} f(u,v) = 0$$

Umgangssprachlich heißt dies, dass aus jedem Knoten (außer s und t) genauso viel ausfließen muss, wie in den Knoten einfließt. Daraus folgt, dass genau der Betrag, der die Quelle verlässt auch in der Senke ankommt.

Definition 13.3. *Der Betrag (auch Größe, (Fluss)Wert,* value*) eines Flusses $|f|$ ist der Gesamtwert des Flusses, der die Quelle verlässt, bzw. die Senke erreicht und berechnet sich durch*

$$|f| = \sum_{v \in V} f(s,v) = \sum_{v \in V} f(v,t).$$

Dies wäre beispielsweise die Wassermenge, die bei einem gegebenen Fluss insgesamt in einer Zeiteinheit durch ein Netzwerk geschickt werden kann.

Definition 13.4. *Der **positiv eingehende Fluss** eines Knotens v ist*

$$\sum_{u \in V : f(u,v) > 0} f(u,v).$$

*Der **positiv ausgehende Fluss** eines Knotens v ist*

$$\sum_{u \in V : f(v,u) > 0} f(v,u).$$

*Die **Flussstärke** an einem Knoten v berechnet sich durch*

$$\text{positiv ausgehender Fluss in } v - \text{positiv eingehender Fluss in } v$$

Die Ford-Fulkerson-Methode

Die Ford-Fulkerson-Methode ist ein allgemeines Verfahren, um den maximalen Fluss in einem Netzwerk zu bestimmen. Die Idee basiert auf der Feststellung, dass wir durch eine Greedy-Vorgehensweise tatsächlich einen maximalen Fluss finden. Wir suchen einfach so lange nach flussvergrößernden Pfaden (genaue Definition folgt unten), wie wir noch welche finden können.

Um zu verstehen, wie das funktioniert, müssen wir das Konzept der **Auslöschung** kennenlernen. Auslöschung tritt ein, wenn zwei Knoten durch zwei Kanten verbunden sind. Es gilt also für zwei Knoten $u, v \in V$, dass $c(u, v) > 0$ und $c(v, u) > 0$ ist. Wichtig ist, dass hier auch keine Beziehung zwischen den Kapazitäten besteht (bei der Flussstärke gibt es eine Beziehung zwischen $f(v, u)$ und $f(u, v)$). Eine Interpretation einer solchen Konstellation ist, dass es z.B. zwei verschiedene Straßen von A nach B, bzw. von B nach A gibt. Wenn wir nun einen Fluss haben, der beide Kanten ausnutzt, dann findet Auslöschung statt. Eine Anzahl von k von A nach B sowie eine Anzahl von $k + l$ von B nach A ergibt insgesamt eine Anzahl von l von B nach A.

Im Algorithmus werden wir diese Situationen nicht explizit beachten. Aus unserer Interpretation von Kapazitäten und Flüssen wird sich implizit ergeben, dass solche Konstellationen berücksichtigt werden. An dieser Stelle schon einmal eine kleine (unvollständige) Vorschau darauf, wie eine inkrementelle Vergrößerung des Flusses ablaufen wird:

- Wir suchen einen Pfad von s nach t, der noch verbessert werden kann.
- Wir erhöhen den Fluss um einen Wert d auf allen Kanten (v, u) des Pfades.
- Um weiterhin einen korrekten Fluss zu erhalten, müssen wir dann den Flusswert $f(u, v)$ um d verringern.
- Dies entspricht genau der Auslöschung. Wenn der Flusswert in eine Richtung vergrößert wird, wird er in die andere verringert.

Idee

Die Idee der Ford-Fulkerson-Methode ist denkbar einfach:

1. Wir beginnen mit einem Fluss $f(u, v) = 0$ für alle $u, v \in V$.
2. In jeder Iteration erhöhen wir den Betrag des Flusses, indem wir einen flussvergrößernden Pfad finden (genaue Definition folgt).
3. Wir wiederholen Schritt 1 und 2 so lange, bis wir keinen flussvergrößernden Pfad mehr finden.

Anmerkung

Im englischen heißen flussvergrößernde Pfade „augmenting paths“.

Damit ergibt sich die folgende Struktur des Algorithmus:

```
FORD-FULKERSON(G)
1  initialize flow f to 0
2  while there exists an augmenting path p
3          augment flow f along p
```

Dieser Code verrät offensichtlich noch nicht viel über eine konkrete Implementierung der Ford-Fulkerson-Methode. Das ist auch beabsichtigt, da es verschiedene Implementierungen dieser Methode gibt, von der wir auch eine kennenlernen werden.

Residuennetzwerke

Definition 13.5 (Residuennetzwerk)**.** *Sei $G = (V, E)$ ein Flussnetzwerk mit der Kapazitätsfunktion c und f ein Fluss auf diesem Netzwerk. Ein **Residuennetzwerk** (*residual network*) ist dann ein Netzwerk $G_f = (V, E_f)$ mit der Kapazitätsfunktion*

$$c_f(u, v) = c(u, v) - f(u, v)$$

und

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}.$$

Intuitiv kann man sich ein Residuennetzwerk folgendermaßen vorstellen:

- Wenn es einen Fluss der Größe x von u zu v gibt, dann gibt es aufgrund der Symmetrie auch einen Fluss von $-x$ von v nach u.
- Indem x Einheiten von u nach v fließen, können die $-x$ ausgelöscht werden.
- Dann können sogar noch $c(u, v) \geq 0$ weitere Einheiten von u nach v fließen.
- Insgesamt können also $x + c(u, v) \geq c(u, v)$ Einheiten von u nach v fließen.

Wir bemerken an dieser Stelle, dass die Kapazität im Residuennetzwerk größer sein kann, als die Kapazität der entsprechenden Kante im ursprünglichen Netzwerk, nämlich genau dann, wenn $f(u, v) < 0$ ist.

Lemma 13.1. *Sei $G = (V, E)$ ein Flussnetzwerk und f ein Fluss in diesem Netzwerk. Sei dann G_f das durch f entstandene Residuennetzwerk und f' ein Fluss in dem Residuennetzwerk G_f. Dann ist*

$$(f + f')(u, v) = f(u, v) + f'(u, v)$$

ein Fluss in G mit dem Flusswert

$$|f + f'| = |f| + |f'|.$$

Beweis. Wir müssen nach der Definition die drei Eigenschaften (Kapazitätsbeschränkung, inverse Symmetrie, Flusserhaltung) nachweisen:

Inverse Symmetrie

$$\begin{aligned}(f+f')(u,v) &= f(u,v)+f'(u,v)\\ &= -f(v,u)-f'(v,u)\\ &= -(f(v,u)+f'(v,u))\\ &= -(f+f')(v,u)\end{aligned}$$

Kapazitätsbeschränkung Bemerke hierzu, dass f' ein Fluss in G_f ist, sodass $f'(u,v) \leq c_f(u,v)$ ist. Nach Definition von $c_f(u,v) = c(u,v) - f(u,v)$ ist dann

$$\begin{aligned}(f+f')(u,v) &= f(u,v)+f'(u,v)\\ &\leq f(u,v)+c_f(u,v)\\ &= f(u,v)+(c(u,v)-f(u,v))\\ &= c(u,v)\end{aligned}$$

Flusserhaltung Für alle $u \in V - \{s,t\}$ gilt:

$$\begin{aligned}\sum_{v\in V}(f+f')(u,v) &= \sum_{v\in V}(f(u,v)+f'(u,v))\\ &= \sum_{v\in V} f(u,v) + \sum_{v\in V} f'(u,v)\\ &= 0+0\\ &= 0\end{aligned}$$

Als letzten Teil des Beweises zeigen wir noch, dass die Gleichung für den Flusswert gilt:

$$\begin{aligned}|f+f'| &= \sum_{v\in V}(f+f')(s,v)\\ &= \sum_{v\in V}(f(s,v)+f'(s,v))\\ &= \sum_{v\in V} f(s,v) + \sum_{v\in V} f'(s,v)\\ &= |f|+|f'|\end{aligned}$$

□

Flussverbessernde Pfade

Das zweite Konzept, das nun bereits einige Male angesprochen wurde, ist das eines flussverbessernden Pfades.

Definition 13.6 (Flussverbessernde Pfade)**.** *Sei $G = (V, E)$ ein Flussnetzwerk und f ein Fluss in dem Netzwerk. Dann ist ein flussverbessernder Pfad (*augmenting path*) p ein Pfad von s nach t im Residuennetzwerk G_f.*

Der deutsche Begriff *flussverbessernder Pfad* beschreibt bereits relativ gut die Bedeutung eines solchen Pfades. Dies können wir nun noch genauer präzisieren. Da im Residuennetzwerk jede Kante eine Kapazität hat, die genau der möglichen Verbesserung des Flusses im Netzwerk entlang dieser Kante entspricht, ist also das Kantengewicht bei einem flussverbessernden Pfad ebenfalls die mögliche Verbesserung der entsprechenden Kante. Da das Residuennetzwerk außerdem nach Definition nur positive Kanten enthält, bedeutet allein die Existenz eines Pfades bereits, dass eine Verbesserung des Flusses erzielt werden kann.

Nun bleibt noch zu zeigen, wie wir einen Fluss im Residuennetzwerk zu einem korrekten Flow im ursprünglichen Flussnetzwerk erweitern. Dafür suchen wir einen flussverbessernden Pfad im Residuennetzwerk, berechnen die Flussverbesserung auf dem Pfad und setzen alle anderen Werte für den Fluss im Residuennetzwerk auf 0. Am Ende verbessern wir dann den Ausgangsfluss durch diesen neuen Fluss. Formal ist dies in folgendem Lemma beschrieben:

Lemma 13.2. *Sei $G = (V, E)$ ein Flussnetzwerk. Sei f ein Fluss auf G und p ein flussverbessernder Pfad in G_f. Dann ist $f_p : V \times V \to \mathbb{R}$ mit*

$$f_p(u, v) = \begin{cases} c_f(p) & \textit{Der Pfad } p \textit{ enthält } (u, v). \\ -c_f(p) & \textit{Der Pfad } p \textit{ enthält } (v, u). \\ 0 & \textit{sonst} \end{cases}$$

ein Fluss in G_f mit einem Flusswert $|f_p| = c_f(p) > 0$.

Der Beweis von Lemma 13.2 bleibt als Übung für den Leser.

Korollar 13.1 (Flussverbesserungs-Korollar)**.** *Seien G, f, p, f_p definiert wie oben. Dann ist $f' : V \times V \to \mathbb{R}$ mit $f' = f + f_p$ ein Fluss in G mit dem Flusswert $|f'| = |f| + |f_p| > |f|$ (also ein* besserer *Fluss).*

Beweis. Folgt aus den vorhergegangenen beiden Lemmata. Ein genauer Beweis bleibt als Übung für den Leser. □

Korrektheit der Ford-Fulkerson-Methode

Aus Lemma 13 und 13.2 folgt direkt die Korrektheit der Algorithmus: Da ein flussverbessernder Pfad auch gefunden wird (zum Beispiel durch Breitensuche oder Tiefensuche) ist klar, dass in jeder Iteration der Ford-Fulkerson-Methode ein korrekter Fluss entsteht.

Optimalität der Ford-Fulkerson-Methode

Als nächstes wollen wir zeigen, dass der Fluss, der durch die Ford-Fulkerson-Methode entsteht auch **optimal** ist. Dafür erweitern wir zunächst die Definition eines Flusses auf Mengen von Knoten:

Definition 13.7. *Sei $G = (V, E)$ ein Netzwerk und f ein Fluss auf diesem Netzwerk. Seien weiterhin $X, Y \subseteq V$. Dann ist*

$$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y).$$

Enthalten X oder Y nur ein einzelnes Element, lässt man manchmal auch die Mengenklammern weg, sodass beispielsweise gilt:

$$\forall u \in V - \{s, t\} : \sum_{v \in V} f(u, v) = 0 \iff \forall u \in V - \{s, t\} : f(u, V) = 0$$

Korollar 13.2. *Es gelten die folgenden Äquivalenzen:*

1. *Aufgrund der inversen Symmetrie:*

$$\forall X \subseteq V : f(X, X) = 0$$

2. *Allgemeiner gilt ebenso:*

$$\forall X, Y \subseteq V : f(X, Y) = -f(Y, X)$$

3. *Vereinigung: Für alle $X, Y, Z \subseteq V$ mit $X \cap Y = \emptyset$ gilt*

$$f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$$
$$f(Z, X \cup Y) = f(Z, X) + f(Z, Y)$$

Schnitte

Definition 13.8 (Schnitt)**.** *Ein Schnitt (S, T) eines Netzwerks $G = (V, E)$ ist eine Knotenpartition von V in zwei Mengen S und $T = V - S$, wobei $s \in S$ und $t \in T$ ist.*

Definition 13.9 (Netzfluss)**.** *Für einen Fluss f in einem Netzwerk $G = (V, E)$ ist der **Netzfluss** eines Schnittes (S, T) definiert als $f(S, T)$. Analog ist die Kapazität des Schnittes $c(S, T)$ definiert als die Summe der Kapazitäten aller Kanten (u, v), für die $u \in S$ und $v \in T$ ist.*

Bemerke. Der Netzfluss eines Schnittes kann negative Summanden enthalten (es können durch die *skew symmetry* des Flusses *Rückwärtskanten* von S nach T vorkommen). Bei der Kapazität des Schnittes kann das nicht passieren.

Definition 13.10 (Minimaler Schnitt)**.** *Ein* ***minimaler Schnitt*** *ist ein Schnitt mit minimaler Kapazität (ein Schnitt für den es keinen anderen Schnitt im Netzwerk gibt, der eine niedrigere Kapazität hat).*

Der Netzfluss für jeden Schnitt ist gleich. Formal beweist dies das folgende Lemma:

Lemma 13.3. *Sei f ein Fluss im Netzwerk $G = (V, E)$ mit der Quelle s und der Senke t. Sei außerdem (S, T) ein Schnitt von G. Dann ist der Netzfluss von (S, T) genau $f(S, T) = |f|$.*

Beweis. Wir können die Gleichheit einfach für einen beliebigen Schnitt (S, T) nachrechnen:

$$\begin{aligned} f(S,T) &= f(S,V) - f(S,S) \\ &= f(S,V) - 0 \\ &= f(s,V) + f(S-s,V) \\ &= f(s,V) + 0 \qquad \text{(Aufgrund der Flusserhaltung)} \\ &= |f| \qquad \text{(Nach Definition)} \end{aligned}$$

□

Korollar 13.3. *Die Kapazität eines beliebigen Schnittes (S, T) eines Netzwerkes G ist eine obere Schranke für den Flusswert eines beliebigen Flusses f in dem Netzwerk.*

Beweis. Wir rechnen aus:

$$\begin{aligned} |f| &= f(S,T) \\ &= \sum_{u \in S} \sum_{v \in T} f(u,v) \\ &\leq \sum_{u \in S} \sum_{v \in T} c(u,v) \\ &= c(S,T) \end{aligned}$$

□

Dieses Korollar sagt u.A. aus, dass auch der maximale Fluss in einem Netzwerk durch die Kapazität eines beliebigen Schnittes beschränkt ist. Tatsächlich wird sich zeigen, dass sogar gilt, dass der maximale Flusswert und die Kapazität des minimalen Schnittes genau gleich sind. Das nennt man das **Max-flow min-cut Theorem**:

Satz 13.1 (Max-Flow Min-Cut Theorem)**.** *Sei f ein Fluss in dem Netzwerk $G = (V, E)$ mit der Quelle s und der Senke t. Dann sind die folgenden drei Aussagen äquivalent:*

1. *f ist ein maximaler Fluss in G.*
2. *Das Residuennetzwerk G_f enthält keinen flussverbessernden Pfad.*
3. *Es gibt einen Schnitt (S, T) von G, für den $|f| = c(S, T)$ gilt.*

Bemerke. Das letzte Korollar hat bereits gezeigt, dass der Flusswert jedes Flusses kleiner oder gleich der Kapazität jedes Schnittes ist. Wenn wir nun also einen Schnitt und einen Fluss finden, bei dem hier sogar Gleichheit vorliegt, muss es sich definitiv um einen minimalen Schnitt handeln.

Beweis. Wir beweisen die Implikationen (1) $\implies$ (2), (2) $\implies$ (3) und (3) $\implies$ (1). Damit haben wir dann die paarweise Biimplikation von (1), (2) und (3) gezeigt.

(1) $\implies$ (2) Sei f ein maximaler Fluss in G, aber G_f enthält noch mindestens einen flussverbessernden Pfad p. Dann folgt aus dem Flussverbesserungs-Korollar, dass es einen Fluss $f + f_p$ in G gibt, dessen Flusswert echt größer ist als $|f|$. Dies ist ein Widerspruch zu der Annahme, dass f maximal ist. Also kann es keinen flussverbessernden Pfad in G_f geben.

(2) $\implies$ (3) Angenommen G_f enthält keinen flussverbessernden Pfad mehr. Dann definieren wir

$$S = \{v \in V : \exists \text{path from } s \text{ to } v \text{ in } G_f\}$$
$$T = V - S$$

(S, T) ist also ein Schnitt, da $s \in S$ und $t \in T$ ist. Es gilt nun aber, dass für alle $u \in S$ und $v \in T$ der Flusswert $f(u, v) = c(u, v)$ ist, da sonst $(u, v) \in E_f$ wäre und dann $v \in S$ sein müsste. Nach dem obigen Lemma gilt nun $|f| = f(S, T)$ und nun auch $f(S, T) = c(S, T)$, also ist $|f| = c(S, T)$.

(3) $\implies$ (1) Nach dem vorhergegangenen Korollar ist für jeden Schnitt (S, T) $|f| \leq c(S, T)$. Gibt es also einen Schnitt mit $|f| = c(S, T)$ muss dies ein maximaler Fluss sein.

□

Pseudocode

```
FORD-FULKERSON(G, s, t)
1  for each edge (u, v) ∈ E
2        f[u, v] = 0
3        f[v, u] = 0
4  while there is a path p = s → t in the residual network G[f]
5        c_f(p) = min {c_f(u, v) : (u, v) ∈ p}
6        for each edge (u, v) ∈ p
7              f[u, v] = f[u, v] + c_f(p)
8              f[v, u] = −f[u, v]
```

Wir haben nun den ersten Pseudocode verbessert, indem wir den Algorithmus zum Finden von flussverbessernden Pfaden ergänzt haben. Aus diesem Code geht nun also klar hervor, wie eine Flussverbesserung erfolgen kann.

Laufzeit

Die Laufzeit des obigen Ford-Fulkerson-Algorithmus ist nicht ideal. Das zeigt die folgende Analyse: Wir nehmen dafür an, dass alle Kanten ganzzahlige Kapazitäten haben. Sei nun $|f^*|$ der Flusswert eines optimalen Flusses. Dann ergibt sich die Laufzeit folgendermaßen:

- Die Initialisierung in Zeile 1-3 ist in $\mathcal{O}(E)$.
- Die **while** -Schleife wird insgesamt höchstens $|f^*|$ Mal ausgeführt.
- Innerhalb der **while** -Schleife braucht das Finden eines flussverbessernden Pfades und das Durchführen der Verbesserung eine Laufzeit von $\mathcal{O}(V + E)$.

Insgesamt ergibt sich so eine Laufzeit von

$$\mathcal{O}(E \cdot |f^*|)$$

Diese Laufzeit ist **nicht** polynomiell. Stellt man sich die Frage, ob es nicht möglich ist, die Laufzeitschranke zu verringern, stellt man schnell fest, dass es durchaus Beispiele gibt, bei der eine unglückliche Wahl der flussverbessernden Pfade dazu führt, dass diese Laufzeit auch erreicht wird.

Algorithmus von Edmonds und Karp

Der Algorithmus von Edmonds ist eine Verbesserung des Ford-Fulkerson-Algorithmus, der tatsächlich einen maximalen Fluss in polynomieller Zeit findet. Die grundsätzliche Idee dabei ist, in jedem Iterationsschritt immer den **kürzesten** flussverbessernden Pfad zu wählen. Dieser lässt sich leicht durch eine Breitensuche finden.

Um die Korrektheit dieser Idee zu beweisen, definieren wir $\delta_f(u, v)$ als die Länge des kürzesten Pfades von u nach v in G_f (Mit dem kürzesten Pfad meinen wir hier tatsächlich den Pfad mit der geringsten Anzahl an Kanten. Dies hat nichts mit der Kapazität oder dem Flusswert einer Kante zu tun). Zunächst beweisen wir ein Lemma:

Lemma 13.4. *Wenn der Edmonds-Karp-Algorithmus auf einem Netzwerk $G = (V, E)$ mit der Quelle s und der Senke t ausgeführt wird, dann wächst $\delta_f(s, v)$ für alle $v \in V - \{s, t\}$ in jeder Iteration* ***streng monoton****.*

Beweis. Angenommen es gibt ein erstes Mal, dass die Länge des kürzesten Pfades für eine Flussverbesserung absinkt. Dann sei f der Fluss vor dieser Flussverbesserung und f' der Fluss nach der Verbesserung. Sei außerdem v der Knoten mit minimalem $\delta_{f'}(s, v)$, für den $\delta_{f'}(s, v) < \delta_f(s, v)$ gilt (der Knoten auf dem verkürzten Pfad, der am nächsten an s liegt). Sei zuletzt noch $p = s \rightsquigarrow u \to v$ ein kürzester Pfad von s nach v in $G_{f'}$, sodass $(u, v) \in E_{f'}$ ist und $\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1$ ist.

So wie wir die Knoten u und v definiert haben, wissen wir, dass sich die Entfernung $\delta(s, u)$ nicht verringert hat (da wir ja ein minimales δ gewählt haben). Es gilt also $\delta_{f'}(s, u) \geq \delta_f(s, u)$. Dann kann es aber die Kante $(u, v) \in E_f$ nicht geben, wie die folgende Rechnung zeigt:

$$\delta_f(s, v) \overset{*}{\leq} \delta_f(s, u) + 1 \leq \delta_{f'}(s, u) + 1 = \delta_{f'}(s, v)$$

* Wir wissen, dass es mindestens den Pfad $s \rightsquigarrow u \to v$ in f gibt. Also ist die Länge dieses Pfades eine obere Schranke für $\delta_f(s, v)$.

Wenn es die Kante also gibt, hat sich die Länge von $\delta(s, v)$ nicht erhöht. Wie kann es nun sein, dass es zwar die Kante $(u, v) \in E_{f'}$ gibt, nicht aber $(u, v) \notin E_f$? Die einzige Möglichkeit ist, dass der Algorithmus den Flusswert der Kante (v, u) erhöht hat. Da der Algorithmus aber immer den kürzesten Pfad im Residuennetzwerk verbessert, hat der kürzeste Pfad von s nach u in G_f die Kante (v, u) als letzte Kante und es gilt:

$$\begin{aligned} \delta_f(s, v) &= \delta_f(s, u) - 1 \\ &\leq \delta_{f'}(s, u) - 1 \\ &= \delta_{f'}(s, v) - 2 \end{aligned}$$

Das kann nicht sein, da wir zu Anfang angenommen haben, dass $\delta_{f'}(s, v) < \delta_f(s, v)$ ist. □

Nun haben wir alle Hilfsmittel, um zu beweisen, dass die Laufzeit des Edmonds-Karp-Algorithmus polynomiell ist:

Satz 13.2. *Wenn der Edmonds-Karp-Algorithmus auf einem Netzwerk $G = (V, E)$ mit Quelle s und Senke t ausgeführt wird, führt er höchstens $\mathcal{O}(V \cdot E)$ Flussverbesserungen durch, bevor er terminiert.*

Beweis. Wir nennen eine Kante (u, v) in G_f **kritisch** auf einem flussverbessernden Pfad p, wenn $c_f(p) = c_f(u, v)$ ist. Zur Erinnerung:

- Bei einer Flussverbesserung verschwinden alle kritischen Kanten aus dem Residuennetzwerk.
- Mindestens eine Kante muss kritisch sein.

Wir können nun zeigen, dass jede Kante höchstens $|V|/2 - 1$ Mal kritisch sein kann. Dazu sei $(u, v) \in E$. Wir wissen, das alle flussverbessernde Pfade kürzeste Pfade im Residuennetzwerk sind. Daher gilt das erste Mal, wenn (u, v) kritisch wird:

$$\delta_f(s, v) = \delta_f(s, u) + 1.$$

Nachdem die Flussverbesserung durchgeführt wurde, verschwindet die Kante aus dem Residuennetzwerk. Die Kante kann erst wieder im Residuennetzwerk auftauchen, wenn der

Fluss von u nach v verringert wird (wenn also die Kante (v, u) in einem Flussverbessernden Pfad auftaucht). In dem Moment gilt, dass für den Fluss f' in G zu diesem Zeitpunkt $\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1$ ist. Jetzt gilt nach dem vorhergegangenen Lemma, dass $\delta_f(s, v) \leq \delta_{f'}(s, v)$ ist. Zusammengenommen ergibt dies:

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) + 1 \geq \delta_f(s, v) + 1 = \delta_f(s, u) + 2$$

Wenn also die Kante (u, v) ein weiteres mal kritisch wird, hat sich die Entfernung von u zu s im flussverbessernden Pfad um mindestens 2 vergrößert. Daher ist die Entfernung von s zu u von oben durch $|V| - 2$ beschränkt. Die -2 kommen dadurch zustande, dass einerseits die Kantenzahl eines Pfades um 1 kleiner ist als die Knotenzahl auf einem Pfad und andererseits t niemals Teil eines flussverbessernden Pfades bis u ist. Da nun die Pfadlänge um 2 wächst, wenn (u, v) das nächste mal kritisch wird, kann diese Kante insgesamt höchstens $\frac{|V|-2}{2} = \frac{|V|}{2} - 1 = \mathcal{O}(V)$ Mal kritisch werden. Im Graphen gibt es $\mathcal{O}(E)$ Knotenpaare, die im Residuennetzwerk durch eine Kante verbunden sind. Damit ist die Gesamtzahl kritischer Kanten durch $\mathcal{O}(V \cdot E)$ beschränkt. $\square$

Für die Laufzeit des Edmonds-Karp-Algorithmus stellen wir nun nur noch fest, dass es $\mathcal{O}(E)$ Iterationen gibt und somit die Laufzeit in $\mathcal{O}(V \cdot E^2)$ ist. Andere Algorithmen (z.B. die Push-Relabel Methode) erzielen sogar noch bessere Ergebnisse wie $\mathcal{O}(V^2 E)$ oder sogar $\mathcal{O}(V^3)$.

14. Komplexitätstheorie

Anmerkung

Große Teile dieses Abschnittes sind für viele vermutlich bereits aus FGI-1 bekannt. Dieses Kapitel dient zur Wiederholung und als Nachschlagewerk zu den wichtigsten Begriffen der Komplexitätstheorie.

Wir haben bisher schon einige verschiedene Probleme kennengelernt. Von einfachen Sortieralgorithmen (Kapitel 3) über Graphenalgorithmen (Kapitel 10 und 12) bis hin zur Dynamischen Programmierung (Kapitel 9). Dabei sind auch die asymptotischen Laufzeiten immer größer geworden. Wir haben sogar ein paar Probleme gesehen, bei denen wir nicht sofort einen Algorithmus gefunden haben, der das Problem in polynomieller Zeit löst. Ein Beispiel für einen solchen Algorithmus ist der Ford-Fulkerson-Algorithmus zum Berechnen eines maximalen Flusses, dessen Laufzeit abhängig vom Wert $|f^*|$ des maximalen Flusses ist. Dies wollen wir im folgenden noch weiter ausreizen.

Das Forschungsfeld der Komplexitätstheorie wurde begründet, bevor es die ersten Computer gab. Damals fragten sich Mathematiker nach den Grenzen der Berechenbarkeit: „Welche Probleme lassen sich überhaupt algorithmisch lösen?“ In der ersten Hälfte des zwanzigsten Jahrhunderts entwickelte dann Alan Turing ein Rechenmodell (die Turing-Maschine), das einerseits sehr einfach zu verstehen ist, andererseits aber alle algorithmisch lösbaren Probleme lösen kann. Turing definierte den Begriff der Berechenbarkeit durch die Turing-Maschine selbst. Diese Definition ist auch heute noch gültig. Wir werden im Folgenden das Rechenmodell der Turing-Maschine kennen lernen und sehen wie wir argumentieren können, ob eine Funktion effizient berechenbar ist oder nicht. Wir werden außerdem einige Probleme kennen lernen, die im Allgemeinen unlösbar sind, egal wie viel Zeit man dafür aufwendet.

Eines der bekanntesten unlösbaren[1] Probleme ist das Halteproblem. Umgangssprachlich formuliert lautet das Halteproblem:

> Es ist im Allgemeinen nicht möglich, für einen Algorithmus festzustellen, ob dieser terminiert oder nicht.

[1] Unlösbar meint hier nicht, dass bisher keine Lösung für dieses Problem gefunden wurde. Viel mehr heißt es, dass bewiesen wurde, dass es keine allgemeine Lösung für ein Problem geben kann.

Komplexitätsklassen

Uns interessieren zunächst nicht die unlösbaren Probleme. Wir wollen einen Formalismus einführen, mit dem wir feststellen können, ob ein Problem *effizient* lösbar ist. Informell können wir bereits die für uns wichtigen Problemklassen beschreiben. Die Problemklasse $\mathcal{P}$ enthält alle Probleme, für die es einen Algorithmus gibt, der das Problem in $\mathcal{O}(n^k)$ löst (für ein k). Die Problemklasse $\mathcal{NP}$ enthält alle Probleme, für die es Algorithmen gibt, die zumindest eine Lösung in polynomieller Zeit ($\mathcal{O}(n^k)$ für ein k) verifizieren können. Neben $\mathcal{P}$ und $\mathcal{NP}$ gibt es noch viele weitere Komplexitätsklassen, die uns hier aber nicht weiter interessieren.

Intuitiv ist relativ offensichtlich, dass es schwieriger sein sollte, Probleme zu lösen anstatt nur eine gegebene Lösung zu verifizieren. Dass aber jedes Problem, das effizient gelöst werden kann, auch effizient verifiziert werden kann, ist klar (man berechnet die Lösung und vergleicht dann mit dem Zertifikat). Es gilt also offensichtlich $\mathcal{P} \subseteq \mathcal{NP}$. Eine der größten ungeklärten Fragen der modernen Informatik ist, ob $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ gilt. Diese Frage konnte bis heute nicht beantwortet werden.

Im Folgenden wollen wir darauf hinarbeiten, die beiden Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$ formal zu definieren.

Codierungen für Probleminstanzen

Wir haben bei den bisherigen Algorithmen schon gesehen, dass wir im Normalfall die Laufzeit eines Algorithmus in Abhängigkeit von der Eingabegröße angeben möchten. Normalerweise ist diese Eingabegröße n. Bisher haben wir dieser Eingabegröße keine weitere Beachtung geschenkt. Da wir aber im nächsten Abschnitt den Formalismus der Turing-Maschine kennen lernen werden, der eine klar definierte Eingabe braucht, beschäftigen wir uns jetzt mit der Frage, wie eine Eingabe für ein Problem *sinnvoll* codiert werden kann.

Um uns das Leben einfacher zu machen, erlauben wir als Eingabezeichen für einen Algorithmus (für Turing-Maschinen) nur Zeichenketten aus 0 und 1. Formal ist dies eine Codierung $e : \mathbb{N} \to \{0, 1\}^*$. Als Beispiel betrachten wir die natürlichen Zahlen als Eingabe für ein Problem. Der intuitive Weg eine natürliche Zahl zu codieren, ist das Binärsystem zu verwenden. Hier ist also $e(i) = (i)_2$. Wir sagen nun, dass ein Algorithmus ein Problem in der Zeit $\mathcal{O}(T(n))$ löst, wenn der Algorithmus für eine beliebige Probleminstanz i mit $n = |i|$ Zeichen in der Zeit $\mathcal{O}(T(n))$ eine Lösung produziert. Ein Problem ist in polynomieller Zeit lösbar, wenn es einen Algorithmus gibt, der das Problem in $\mathcal{O}(n^k)$ für ein konstantes k löst. Mit diesen Voraussetzungen können wir die Komplexitätsklasse $\mathcal{P}$ als all diejenigen Probleme definieren, die in polynomieller Zeit gelöst werden können.

Man muss allerdings bei der Codierung für die Eingabe aufpassen, da man durch eine ineffiziente Codierung die Komplexität eines Algorithmus verstecken kann. Da wir das

nicht möchten, gehen wir im Folgenden immer davon aus, dass die Codierung der Eingabe *effizient genug* ist. Im nächsten Teilabschnitt werden wir genauer sehen, was *effizient genug* bedeutet. Für die meisten Fälle muss man sich darüber keine weiteren Gedanken machen. Wir schreiben $< \dots >$, um zu kennzeichnen, dass es sich um eine angemessene Codierung handelt. So ist beispielsweise $< G >$ die optimale Codierung für einen Graphen G.

Wie eine ineffiziente Codierung die Komplexität eines Algorithmus verstecken kann, zeigt das folgende Beispiel: Wir betrachten ein Problem, das als Eingabe eine Zahl $k \in \mathbb{N}$ erhält und eine Laufzeit von $\Theta(k)$ hat. Wenn wir nun die Zahl k in einer **unären** Codierung der Länge n angeben, dann ist die Laufzeit entsprechend $\mathcal{O}(n)$, also polynomial. Wenn k in einer **binären** Codierung gegeben ist, dann ist die Eingabelänge plötzlich nur noch $n = \lfloor \log k \rfloor + 1$. Hier haben wir es also mit einem exponentiellen Unterschied der Laufzeit zu tun: $\Theta(k) = \Theta(2^n)$.

Formale Sprachen

Formale Sprachen sind ein Weg, um die syntaktische Struktur von Texten zu beschreiben. Wir werden formale Sprachen für die Spezifikation der Eingabe einer Turingmaschine benutzen.

Definition 14.1 (Alphabet)**.** *Eine Menge Σ von Symbolen nennen wir ein Alphabet.*

Definition 14.2 (Wort)**.** *Ein Wort ist eine Folge von Zeichen aus Σ. Wir bezeichnen mit ϵ das leere Wort.*

Definition 14.3. *Mit Σ^2, Σ^3 usw. wird die Konkatenation der Menge Σ mit sich selbst bezeichnet. Dann definieren wir $\Sigma^+ = \Sigma \cup \Sigma^2 \cup \Sigma^3 \cup \dots$ und $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$. Σ^* wird auch als* ***Abschluss*** *von Σ bezeichnet.*

Definition 14.4 (Sprache)**.** *Eine Sprache über einem Alphabet Σ ist eine Teilmenge $L \subseteq \Sigma^*$.*

Bemerke. Ein Entscheidungsproblem kann formal als Teilmenge von Σ^* gesehen werden.

Formale Sprachen lassen sich weiter analysieren. Für uns reicht es festzustellen, dass man formale Sprachen auch miteinander zu neuen formalen Sprachen verknüpfen kann. Dafür kann man die bereits bekannten Mengenoperation (Vereinigung, Schnitt, Komplement) verwenden. Als eine neue Operation definieren wir noch die Konkatenation:

Definition 14.5 (Konkratenation)**.** *Für zwei formale Sprachen L_1 und L_2 ist die Konkatenation $L_1 \cdot L_2 = \{xy : x \in L_1, y \in L_2\}$ die Menge der Wörter xy mit $x \in L_1$ und $y \in L_2$.*

Wir wollen jetzt formale Sprachen dazu benutzen, um genau zu definieren, was eine Eingabe ist, die *effizient genug* codiert wurde.

Definition 14.6. *Eine Funktion $f : \{0,1\}^* \to \{0,1\}^*$ ist **polynomial berechenbar**, wenn es eine Turing-Maschine M gibt, die für ein $x \in \{0,1\}^*$ in Polynomialzeit $f(x)$ berechnet. Zwei Codierungen e_1 und e_2 sind polynomial verwandt (*polynomially related*), wenn es zwei polynomielle Funktionen f_{12} und f_{21} gibt, sodass für alle Probleminstanzen i gilt:*

$$f_{12}(e_1(i)) = e_2(i) \qquad f_{21}(e_2(i)) = e_1(i)$$

Die Wahl zwischen zwei Codierungen hat keinen Einfluss darauf, ob ein Problem in $\mathcal{P}$ ist oder nicht, so lange die beiden Codierungen polynomiell verwandt sind. Für alle „Standardeingaben“ (Adjazenzlisten, Adjazenzmatrizen, ...) kann man zeigen, dass diese miteinander polynomial verwandt sind.

Lemma 14.1. *Sei Q ein Entscheidungsproblem für die Probleminstanzen in der Menge I. Seien e_1 und e_2 zwei polynomial verwandte Codierungen auf I. Dann ist $e_1(Q) \in \mathcal{P}$ gdw. $e_2(Q) \in \mathcal{P}$ ist.*

Beweis. Wir zeigen hier nur eine Richtung des Beweises (nämlich $e_1(Q) \in \mathcal{P} \implies e_2(Q) \in \mathcal{P}$). Die andere Richtung ist entsprechend symmetrisch.

Wir wissen durch $e_1(Q) \in \mathcal{P}$, dass $e_1(Q)$ in polynomieller Zeit gelöst werden kann. Wir wissen außerdem, dass für jede Probleminstanz i in polynomieller Zeit $e_1(i)$ in $e_2(i)$ überführt werden kann. Um nun $e_2(Q)$ für eine Eingabe $e_2(i)$ zu lösen, berechnen wir also einfach zuerst $e_1(i)$ und führen dann den Algorithmus für $e_1(Q)$ auf $e_1(i)$ aus. Hier braucht offensichtlich jeder Schritt nur polynomielle Zeit, also ist $e_2(Q) \in \mathcal{P}$. □

Turing-Maschinen

Bisher ist es sehr einfach gewesen, zu zeigen, dass ein Algorithmus ein Problem in polynomieller Zeit löst. Wir geben einfach den Pseudocode an und verdeutlichen, dass nur eine bestimmte Anzahl an Schritten ausgeführt werden muss. Was bisher noch nicht möglich ist, ist zu zeigen, dass ein Problem **nicht** in polynomieller Zeit gelöst werden kann. In diesem Kapitel lernen wir den Formalismus der Turing-Maschinen kennen, mit dem genau das möglich sein wird.

Die Motivation, Turing-Maschinen anstelle von Pseudocode zu benutzen ist vor allem, dass Pseudocode sehr schlecht formalisiert werden kann. Es ist zwar jedem klar, was eine Zuweisung bedeutet, allerdings ist nicht notwendigerweise offensichtlich, dass eine *Operation* auch in konstanter Zeit ausgeführt werden kann. Das Rechenmodell der Turing-Maschinen ist einfach genug, um eine formale Definition von Laufzeiten zu ermöglichen. Da man außerdem zeigen kann, dass eine Turing-Maschine alle Probleme, die ein moderner Computer in polynomieller Zeit lösen kann, ebenfalls in polynomieller Zeit lösen kann (und umgekehrt), ist es ausreichend zu zeigen, dass eine Turing-Maschine ein Problem in polynomieller Zeit (nicht) lösen kann.

Anmerkung

Eine Turing-Maschine ist tatsächlich gleich mächtig wie jedes andere Rechenmodell (Church-Turing-These). Das bedeutet, dass es egal ist, ob ein Algorithmus auf einen hochparallelisierten Rechner-Cluster oder auf einer Turing-Maschine ausgeführt wird, es wird nicht passieren, dass nur ein Rechenmodell ein Problem in polynomieller Zeit lösen kann. Was allerdings passieren kann, ist dass das Polynom für einen Parallelrechner deutlich kleiner ist und die Berechnung entsprechend auch viel schneller ist.

Eine Turing-Maschine (TM) besteht aus einem unendlichen Band, das in einzelne Zellen unterteilt ist, und einem Zeiger (oder auch Lese-Schreib-Kopf), der auf eine Zelle auf dem Band zeigt. Jede Zelle auf dem Band kann genau ein Symbol speichern. Zusätzlich dazu hat eine Turing-Maschine eine endliche **Steuerung**, deren Größe unabhängig von der Eingabe ist. Die Steuerung ist ein Automat, der aus einer endlichen Anzahl von Zuständen besteht. In jedem Zustand führt die TM die folgenden Schritte aus:

1. Die TM liest das Symbol an der Stelle des Zeigers vom Band.
2. Die TM schreibt abhängig vom Zustand ein neues Symbol an dieselbe Stelle. Das neue Symbol kann auch das gleiche sein, dass sie zuvor gelesen hat.
3. Der Zeiger wird einen Schritt nach links, nach rechts oder gar nicht bewegt.

Die Steuerung der TM ist das, was wir als **Algorithmus** bezeichnen. Formal lässt sich eine TM folgendermaßen definieren:

Definition 14.7 (Turing-Maschine)**.** *Eine Turing-Maschine ist ein 7-Tupel* $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$*. Dabei ist:*

- Q *eine endliche Menge von Zuständen.*
- Γ *eine endliche Menge von Bandsymbolen.*
- Σ *eine endliche Menge von Eingabesymbolen mit* $\Sigma \subset \Gamma$*.*
- $B = \{L, R, N\}$ *ist die Menge der Bewegungsrichtungen des Pointers (*left*,* right*,* none*).*
- $\delta : (Q \times \Gamma) \to (Q \times \Gamma \times B)$ *ist die Überführungsfunktion, die einen Zustand auf den nächsten überführt.*
- $q_0 \in Q$ *ist der Startzustand.*
- $F \subset Q$ *ist die eine Menge von Endzuständen.*

Der Zustand einer TM ist normalerweise implizit aus dem Bandinhalt und dem aktuellen Zustand $q \in Q$ *klar. Deswegen verwenden wir die folgende Kurzschreibweise für den Zustand einer TM:*

$$\alpha_1 q \alpha_2 \qquad (q \in Q, \alpha_1, \alpha_2 \in \Gamma^*)$$

Einen Übergang von einem Zustand zum nächsten kennzeichnen wir durch einen Doppelpfeil

$$\alpha_1 q \alpha_2 \Rightarrow \alpha_1' q' \alpha_2'$$

wenn der Übergang durch einen einzigen Schritt erfolgt (einmaliges Anwenden der Überführungsfunktion) oder durch einen Doppelpfeil mit Stern

$$\alpha_1 q \alpha_2 \underset{*}{\Rightarrow} \alpha_1' q' \alpha_2'$$

wenn der Übergang durch mehrmaliges Anwenden der Überführungsfunktion erfolgt.

Definition 14.8 (Ausgabe)**.** *Die Ausgabe einer Turing-Maschine ist der Inhalt des Bandes nachden die TM einen Endzustand erreicht hat.*

Definition 14.9. *Eine TM M* ***akzeptiert*** *eine Sprache L, wenn M für jedes Wort in der Sprache einen Zustand $f \in F$ erreicht. Wir sagen manchmal auch, dass M in dem Fall 1 ausgibt. Analog wird eine Eingabe von M* ***abgelehnt****, wenn in einem Zustand $q \notin F$ gehalten wird. Wir sagen in dem Fall auch, dass die TM 0 ausgibt.*

Bemerke. Akzeptieren und Ablehnen sind nicht die beiden einzigen Ereignisse, die bei einer TM auftreten können. Es kann auch passieren, dass die TM in eine Dauerschleife geht. Dies ist ein völlig anderes Verhalten und darf nicht mit dem Ablehnen einer Eingabe verwechselt werden.

Anmerkung

Um das Arbeiten mit TMs einfacher zu machen, erlaubt man meistens einige zusätzliche Zeichen im Eingabe- und/oder Bandalphabet. Wir werden die Formalitäten hier nicht weiter betrachten, sondern nur feststellen, dass eine TM, die als Eingabe eine endliche Anzahl von Zeichen erlaubt, äquivalent zu einer TM ist, die nur Zeichenketten aus 1 und 0 erlaubt.

Es sollte aber angemerkt werden, dass die Konventionen hier anders sind als in FGI-1: Wir bezeichnen mit dem Zeichen # hier ein Trennzeichen. In FGI-1 haben wir meistens $ als Trennzeichen verwendet und mit # eine leere Bandzelle markiert.

Definition 14.10 (Berechenbarkeit)**.** *Die Menge der berechenbaren Probleme ist definiert als die Menge der Probleme, die von einer TM in endlicher Zeit gelöst werden können.*

Entscheidungs- und Optimierungsprobleme

Viele Probleme aus der echten Welt sind Optimierungsprobleme: Kürzeste Pfade, Größtes Matching, Größte Clique, ... Wir wollen in diesem Kapitel einen Formalismus definieren, durch den wir belegen können, dass ein Problem nicht effizient gelöst werden kann. Dabei ist es störend, dass die Ausgabe des Algorithmus variabel ist. Wir hätten lieber eine

Problemstellung, auf die nur mit „Ja“ oder „Nein“ geantwortet werden muss. Ein solches Problem nennt man **Entscheidungsproblem**.

Auch wenn es im ersten Moment so scheinen mag, als ob Entscheidungsprobleme eine völlig andere Art von Problem sind, gibt es eine enge Beziehung zwischen Optimierungs- und Entscheidungsproblemen. Wir können nämlich einen Algorithmus für ein Optimierungsproblem ganz einfach zu einem Algorithmus für ein Entscheidungsproblem abändern: Dazu führen wir einfach den Optimierungsalgorithmus aus und prüfen am Ende, ob das produzierte Ergebnis dem erwarteten Ergebnis entspricht. In vielen Fällen (aber nicht immer) lässt sich auch das Entscheidungsproblem auf das Optimierungsproblem zurückführen.

Die Idee für uns ist nun, dass wir nur zeigen müssen, dass ein bestimmtes Entscheidungsproblem sehr schwer zu lösen ist. Dann haben wir automatisch gezeigt, dass auch alle verwandten Optimierungsalgorithmen mindestens genauso schwer sind.

Ein Beispiel ist das Max-Flow-Problem. Die Formulierung des Optimierungsproblems könnte sein: „Was ist der größte Fluss im Netzwerk G“. Das entsprechende Entscheidungsproblem wäre dann: „Gibt es einen Fluss der Größe k“.

Die Komplexitätsklasse $\mathcal{P}$

Wir können nun formal die Komplexitätsklasse $\mathcal{P}$ definieren. Dafür wiederholen wir zunächst ein paar bekannte Begriff. Wir betrachten eine Turing-Maschine M:

- M **akzeptiert** eine Zeichenkette $x \in \{0,1\}^*$, wenn M mit der Eingabe x nach endlicher Zeit $M(x) = 1$ ausgibt.
- Die von M akzeptierte Sprache ist $L = \{x \in \Sigma^* : M(x) = 1\}$.
- M lehnt ein Wort x ab, wenn $M(x) = 0$ ist.
- Es kann Wörter geben, die von M weder akzeptiert noch abgelehnt werden.
- M **entscheidet** eine Sprache, wenn M auf jeder Eingabe anhält (also jedes Wort entweder akzeptiert oder abgelehnt wird).
- M entscheidet/akzeptiert eine Sprache in **Polynomialzeit**, wenn es ein k gibt, sodass die Anzahl der Zustände, die für das Entscheiden der Sprache durch $\mathcal{O}(n^k)$ beschränkt ist.

Mit diesen Voraussetzungen können wir endlich $\mathcal{P}$ formal definieren:
Definition 14.11. *Die Komplexitätsklasse $\mathcal{P}$ ist definiert als*

$$\mathcal{P} = \{L \subseteq \{0,1\}^* : \textit{Es gibt eine TM } M\textit{, die } L \textit{ in Polynomialzeit entscheidet.}\}$$

Satz 14.1. *Eine Äquivalente Definition ist*

$$\mathcal{P} = \{L : L \text{ wird von einer TM in Polynomialzeit akzeptiert}\}.$$

Beweis. Offensichtlich wird eine Sprache von einer TM akzeptiert, wenn diese die Sprache entscheidet. Um zu zeigen, dass die beiden Definitionen von $\mathcal{P}$ äquivalent sind, müssen wir also nur noch zeigen, dass eine Sprache von einer TM entschieden wird, wenn diese die Sprache akzeptiert.

Sei dazu L eine von M in Polynomialzeit akzeptierte Sprache. Es gibt also eine Konstante c, sodass M die Sprache in höchstens $T = cn^k$ Schritten akzeptiert. Jetzt können wir eine andere TM M' konstruieren, die für jede Eingabe x die Ausführung von M für T Schritte simuliert. Wenn M nach höchstens T Schritten akzeptiert hat, gibt M 1 aus und sonst 0. Das entscheidet die Sprache. □

Bemerke. Dieser Beweis ist nicht konstruktiv, d.h. wir erhalten aus dem Beweis nicht sofort ein Konstruktionsverfahren. Wir kennen nämlich ggf. nicht die Laufzeit T. Wir wissen nur, dass diese Laufzeit existieren muss.

Die Komplexitätsklasse $\mathcal{NP}$

Als nächstes wollen wir noch die Komplexitätsklasse $\mathcal{NP}$ formalisieren. Wir erinnern uns, dass die Komplexitätsklasse die Menge der Probleme war, die in Polynomialzeit verifiziert werden können. Dazu müssen wir nun erstmal erklären, was eigentlich Verifikation bedeutet. Als Beispiel betrachten wir das Problem des Hamilton-Kreises. Ein Hamilton-Kreis ist ein Kreis in einem Graphen, der jeden Knoten exakt ein Mal enthält. Das Hamiltonkreisproblem beantwortet die Frage, ob es in einem gegebenen Graphen einen Hamilton-Kreis gibt. Offensichtlich lässt sich dieses Problem nicht ohne weiteres lösen. Wenn wir aber ein **Zertifikat** bekommen, also einen Lösungsvorschlag, ist es sehr einfach (= in polynomieller Zeit), zu **verifizieren**, dass diese Lösung korrekt oder inkorrekt ist. Dazu prüfen wir einfach, ob die Definition eines Hamilton-Kreises eingehalten wurde. Für das Hamiltonkreisproblem wäre ein Zertifikat eine Folge

$$v_1, v_2, \ldots, v_{|V|}$$

von Knoten, die möglicherweise einen Hamilton-Kreis bilden.

Definition 14.12 (Verifikationsmaschine)**.** *Eine Verifikationsmaschine ist eine Turing-Maschine M, die zwei Argumente bekommt: Das erste Argument x ist die normale Eingabe, für die eine Lösung gefunden werden soll. Das zweite Argument y ist ein **Zertifikat,** eine vorgeschlagene Lösung für die Eingabe x. M **verifiziert** die Eingabe x, wenn es ein Zertifikat y gibt, sodass $M(x, y) = 1$ ist. Die von einer Verifikationsmaschine **verifizierte Sprache** ist*

$$L = \{x \in \{0,1\}^* : \exists y \in \{0,1\}^* : M(x, y) = 1\}.$$

Anders herum ***verifiziert*** M *eine Sprache* L*, wenn gilt:*

$$\forall x \in L : \exists y : M(x,y) = 1$$
$$\forall x \notin L : \forall y : M(x,y) = 0$$

Nun können wir tatsächlich $\mathcal{NP}$ definieren:
Definition 14.13. *Eine Sprache* L *ist in der Komplexitätsklasse* $\mathcal{NP}$ *genau dann wenn es eine Verifikationsmaschine* M *und eine Konstante* c *gibt, sodass gilt:*

$$\forall x \in L : \textit{Es gibt ein Zertifikat } y \textit{ mit } |y| = \mathcal{O}(|x|^c)\textit{, sodass } M(x,y) = 1 \textit{ ist.}$$

Anmerkung

Historisch definiert man die Klasse $\mathcal{NP}$ als die Menge der Probleme, die in Polynomialzeit von einer nichtdeterministischen TM gelöst werden können. Daher kommt auch der Name ($\mathcal{N}$ichtdeterministische $\mathcal{P}$olynomialzeit). Eine solche TM arbeitet in zwei Phasen:

1. Es wird nichtdeterministisch eine Lösung (Zertifikat) für das Problem geraten.
2. Das Zertifikat wird deterministisch verifiziert.

Man sieht sofort die Ähnlichkeiten zur Verifikation und tatsächlich kann man leicht zeigen, dass beide Definitionen äquivalent sind.

Was aber bei beiden Definitionen wichtig ist: Es sind nicht alle Probleme in $\mathcal{NP}$, die nicht in $\mathcal{P}$ sind. Es gibt auch Probleme, die weder in $\mathcal{P}$ noch in $\mathcal{NP}$ liegen.

$\mathcal{P}$ vs. $\mathcal{NP}$

Es gehört zu einer der größten ungeklärten Fragen, ob die beiden Komplexitätsklassen $\mathcal{P}$ und $\mathcal{NP}$ gleich sind oder nicht. Was man allerdings weiß ist folgendes:

- Die Komplexitätsklasse $\mathcal{P}$ ist gegenüber dem Komplement abgeschlossen. Für eine Sprache $L \in \mathcal{P}$ ist also auch $\overline{L} = \{x : x \notin L\} \in \mathcal{P}$.

Es ist allerdings unklar, ob co-$\mathcal{NP} = \{L : \overline{L} \in \mathcal{NP}\} = \mathcal{NP}$ ist. Es bleiben vier mögliche Beziehungen:

$$\mathcal{NP} = \text{co-}\mathcal{NP} \text{ und } \mathcal{P} = \mathcal{NP} \qquad \text{Sehr unwahrscheinlich}$$
$$\mathcal{NP} = \text{co-}\mathcal{NP} \text{ und } \mathcal{P} \neq \mathcal{NP}$$
$$\mathcal{NP} \neq \text{co-}\mathcal{NP} \text{ und } \mathcal{P} = \mathcal{NP} \cap \text{co-}\mathcal{NP}$$
$$\mathcal{NP} \neq \text{co-}\mathcal{NP} \text{ und } \mathcal{P} \neq \mathcal{NP} \cap \text{co-}\mathcal{NP} \qquad \text{Am wahrscheinlichsten}$$

$\mathcal{NP}$-Vollständigkeit

Da es offenbar leichtere und schwierigere Probleme zu geben scheint, wollen wir nun einen Weg finden, um zu beweisen, das ein Problem *zu den schwierigsten Problemen seiner Art* gehört. Intuitiv könnte diese Definition sein, dass ein Problem schwierig ist, wenn es in $\mathcal{NP}$ ist. Das ist aber keine gute Definition, da $\mathcal{P}$ eine Teilmenge von $\mathcal{NP}$ ist. Deswegen wollen wir den Begriff der Reduktion einführen, der uns dabei helfen wird, schwierige Probleme zu identifizieren.

Definition 14.14 (Reduktion)**.** *Seien L_1 und L_2 formale Sprachen. Wir sagen, dass L_1 in Polynomialzeit auf L_2 reduziert werden kann, wenn es eine Funktion $f : \{0,1\}^* \to \{0,1\}^*$ gibt, die in Polynomialzeit berechnet werden kann und für die gilt:*

$$\forall x \in \{0,1\}^* : x \in L_1 \text{ gdw. } f(x) \in L_2$$

Wenn es eine solche Funktion gibt, schreiben wir $L_1 \leq_p L_2$. Die Funktion nennt man eine Reduktionsfunktion.

Die Intuition für eine Reduktion sollte die folgende sein: Wenn wir ein Problem A, für das wir keine effizente Lösung kennen, aber eine effiziente Lösung für ein Problem B kennen und wissen, wie wir eine Probleminstanz von A effizient in eine Instanz von B umformen können, dann können wir auch Problen A effizient lösen. Konkret ist f also eine Abbildung von Probleminstanzen:

- Wenn $x \in L_1$, dann ist auch $f(x) \in L_2$.
- Wenn $x \notin L_1$, dann ist auch $f(x) \notin L_2$.

Es wird explizit **nicht** gefordert, dass f injektiv oder surjektiv sein muss. Dadurch wissen wir, dass Problem A höchstens so schwer zu berechnen ist wie Problem B. Dadurch wird es in der Theorie sehr einfach, zu beweisen, dass ein bestimmtes Problem in $\mathcal{P}$ ist:

Lemma 14.2. *Seien $L_1, L_2 \subseteq \{0,1\}^*$ mit $L_1 \leq_p L_2$, Dann gilt $L_2 \in \mathcal{P} \implies L_1 \in \mathcal{P}$.*

Beweis. Sei M_2 eine TM, die L_2 in Polynomialzeit entscheidet. Sei F eine TM, die (in Polynomialzeit) die Reduktionsfunktion f berechnet. Dann können wir eine TM M_1 konstruieren, die die folgenden Schritte ausführt:

1. Die Eingabe ist ein $x \in L_1$.
2. Berechne $f(x)$ mit der TM F.
3. Berechne $f(x) \in L_2$ durch M_2.
4. Gib das Ergebnis von M_2 als Ergebnis von M_1 aus.

Man sieht sofort, dass M_1 die Sprache L_1 entscheidet. Dass dies auch in Polynomialzeit passiert, kann man folgendermaßen sehen: Da f in Polynomialzeit berechnet werden

kann, lässt sich die Laufzeit durch $|f(x)| \leq |x|^k$ abschätzen. Da auch M_2 in Polynomialzeit terminiert, lässt sich auch deren Laufzeit entsprechend durch ein m^ℓ abschätzen. Zusammen ist also die Laufzeit durch $(|x|^k)^\ell = |x|^{k\cdot\ell}$ beschränkt und demnach polynomial. Offensichtlich funktioniert dies aber nur, wenn die Reduktion auch tatsächlich in Polynomialzeit erfolgen kann. □

Auf ganz ähnliche Weise wollen wir nun diese Erkenntnisse als Grundlage für eine Definition von besonders schwierigen Problemen benutzen:

Definition 14.15 ($\mathcal{NP}$-schwer)**.** *Eine Sprache (ein Problem) $L \subseteq \{0,1\}^*$ ist $\mathcal{NP}$-schwer, wenn gilt:*

$$\forall L' \in \mathcal{NP} : L' \leq_p L$$

Definition 14.16 ($\mathcal{NP}$-vollständig)**.** *Eine Sprache (ein Problem) $L \subseteq \{0,1\}^*$ ist $\mathcal{NP}$-vollständig, wenn $L \in \mathcal{NP}$ und L $\mathcal{NP}$-schwer ist. Die Menge der $\mathcal{NP}$-vollständigen Probleme bezeichnen wir mit $\mathcal{NPC}$.*

Die Definition von $\mathcal{NP}$-Vollständigkeit im Vergleich zu $\mathcal{NP}$-schwer mag einem wenig sinnvoll erscheinen. Es gibt aber durchaus $\mathcal{NP}$-schwere Probleme, die selbst nicht in $\mathcal{NP}$ sind. Dazu gehört zum Beispiel eine gewinnende Strategie beim Schach.

Wir werden sehen, dass es diese Definition von $\mathcal{NP}$-Vollständigkeit ermöglicht, mit relativ wenig Aufwand zu zeigen, dass ein Problem „besonders schwer“ ist. Zuvor machen wir aber noch die folgende Feststellung:

Satz 14.2. *Wenn es ein $\mathcal{NP}$-vollständiges Problem gibt, welches in $\mathcal{P}$ ist, dann gilt $\mathcal{P} = \mathcal{NP}$. Eine äquivalente Aussage ist: Wenn es ein $\mathcal{NP}$-vollständiges Problem gibt, das nicht in $\mathcal{P}$ ist, dann ist kein $\mathcal{NP}$-vollständiges Problem in $\mathcal{P}$ und es gilt $\mathcal{P} \neq \mathcal{NP}$.*

Beweis. Angenommen es gäbe ein $L \in \mathcal{NPC}$ mit $L \in \mathcal{P}$. Dann gilt für jedes $L' \in \mathcal{NP}$, dass $L' \leq_p L$ ist. Nach dem obigen Lemma folgt aus $L \in \mathcal{P}$, dass auch $L' \in \mathcal{P}$ ist und damit gilt $\mathcal{P} = \mathcal{NP}$. □

Matchings in bipartiten Graphen

Wir wollen nun beispielhaft sehen, wie man eine Reduktion anwenden kann. Dafür betrachten wir das Problem von Matchings in bipartiten Graphen. Zuerst einige Definitionen:

Definition 14.17 (Bipartiter Graph)**.** *Ein ungerichteter Graph $G = (V, E)$ heißt bipartit, wenn es eine Knotenpartition von V in A und $B = V - A$ gibt, sodass $E \subseteq A \times B$ ist. Alle Kanten müssen also Knoten aus den beiden Knotenmengen A und B verbinden. Innerhalb von A und B sind keine Knoten miteinander verbunden.*

Definition 14.18 (Matching)**.** *Ein Matching in einem ungerichteten Graphen $G = (V, E)$ ist eine Teilmenge $M \subseteq E$, für die jeder Knoten aus V mit höchstens eine Kante aus M inzidiert. Ein **maximales Matching** (*engl.: maximum Matching*) ist ein Matching, für das $|M|$ maximal ist. Ein **nicht erweiterbares Matching** (*engl. maximal Matching*) ist ein Matching, bei dem keine Kante zu M hinzugenommen werden kann, ohne die Matching-Eigenschaft zu verletzen.*

Bemerke. Ein *maximum Matching* und ein *maximal Matching* sind nicht das gleiche. Jedes *maximum Matching* ist auch ein *maximal Matching*, aber nicht umgekehrt.

Im ersten Moment sieht das Matching-Problem sehr schwer lösbar aus. Und tatsächlich ist das Matching-Problem in beliebigen Graphen $\mathcal{NP}$-vollständig. In bipartiten Graphen jedoch können wir uns mit Reduktion weiterhelfen. Wir reduzieren nun das Matching-Problem auf ein Max-Flow Problem:

Zuerst konstruieren wir folgendermaßen ein Flussnetzwerk:

1. Füge zum bipartiten Graphen zwei Knoten s, t hinzu.
2. Verbinde jeden Knoten aus der Menge A mit einer ungerichteten Kante mit s und jeden Knoten aus der Menge B entsprechend mit t.
3. Weise jeder Kante in diesem Graphen die Kapazität 1 zu.

Auf dieses Flussnetzwerk wenden wir nun den Algorithmus von Edmonds und Karp an. Wir wissen, dass dieser ganzzahlige Ergebnisse produziert, wenn alle Kapazitäten ganzzahlig sind. Also wird der Flusswert jeder Kante in unserem Netzwerk entweder 1 oder 0 sein. Wir nehmen nun alle Kanten mit dem Flusswert 1 in unser Matching auf und erhalten so ein maximales Matching (*maximum Matching*).

$\mathcal{NP}$-Vollständigkeit anwenden

Nachdem wir nun den Begriff der $\mathcal{NP}$-Vollständigkeit definiert haben, wollen wir sehen, wie man diese Definition anwenden kann, um zu zeigen, dass ein Problem $\mathcal{NP}$-vollständig ist. Dazu haben wir grundsätzlich zwei Möglichkeiten:

- Wir zeigen für ein Problem L, dass $L \in \mathcal{NP}$ ist und zeigen dann, dass für **jedes** $L' \in \mathcal{NP}$ gilt: $L' \leq_p L$. Dies ist normalerweise sehr schwierig. Der erste dieser Beweise wurde von Cook erbracht und zeigte, dass $\textsc{Sat} \in \mathcal{NPC}$ ist.
- Alternativ kennen wir bereits ein $\mathcal{NP}$-vollständiges Problem und können dieses auf unser Problem reduzieren.

Dass die zweite Möglichkeit tatsächlich ausreichend ist, ergibt sich aus der Transitivität der $\leq_p$-Relation, die wir im folgenden Lemma nachweisen:

Lemma 14.3 (Transitivität von $\leq_p$)**.** *Die Relation $\leq_p$ ist transitiv, es gilt also: $L_1 \leq_p L_2 \wedge L_2 \leq_p L_3 \implies L_1 \leq_p L_3$.*

Beweis. Nach der Definition von $\leq_p$ gibt es zwei Polynomialzeitfunktionen f und g mit

$$x \in L_1 \iff f(x) \in L_2 \qquad y \in L_2 \iff g(y) \in L_3$$

Zusammen gilt also

$$x \in L_1 \iff f(x) \in L_2 \iff g(f(x)) \in L_3$$

Die Funktion $g \circ f$ ist also eine Polynomialzeitreduktion von L_1 auf L_3. Dass diese auch in polynomieller Zeit läuft ist durch die Definition von f und g klar. □

Nun wollen wir noch formal beweisen, dass es tatsächlich reicht, eine einzige Reduktion zu machen, um zu zeigen, dass ein Problem in $\mathcal{NPC}$ ist:

Satz 14.3. *Seien L, L' Sprachen mit $L' \in \mathcal{NPC}$, sodass $L' \leq_p L$ gilt. Dann ist L $\mathcal{NP}$-schwer. Wenn außerdem $L \in \mathcal{NP}$ ist, dann ist L $\mathcal{NP}$-vollständig.*

Beweis. Da L' $\mathcal{NP}$-vollständig ist, gibt es für alle $L'' \in \mathcal{NP}$ eine Reduktion $L'' \leq_p L'$. Da außerdem $L' \leq_p L$ ist, können wir mit der Transitivität schließen, dass $L'' \leq_p L$ ebenfalls für alle $L'' \in \mathcal{NP}$ gilt. Das zeigt, dass L $\mathcal{NP}$-schwer ist. Ist außerdem $L \in \mathcal{NP}$, ist L nach Definition $\mathcal{NP}$-vollständig. □

Aus diesen Beweisen ergibt sich sofort eine Methode, um $L \in \mathcal{NPC}$ für ein L zu beweisen:

1. Zeige, dass $L \in \mathcal{NP}$ ist.
2. Wähle ein bekanntes $\mathcal{NP}$-vollständiges L'.
3. Beschreibe einen Algorithmus F, der die Reduktionsfunktion f berechnet, die jede Probleminstanz von L' auf eine von L abbildet.
4. Beweise, dass gilt: $\forall x \in \{0,1\}^* : x \in L' \iff f(x) \in L$.
5. Beweise, dass f in Polynomialzeit berechenbar ist.

Man muss bei diesem Vorgehen gut aufpassen, dass man die Reduktion in die *richtige* Richtung durchführt. Es werden Probleminstanzen von L' auf Probleminstanzen von L abgebildet, nicht anders herum. Außerdem sollte man sich gut überlegen, welches Problem L' man wählt. Nach Möglichkeit sollte man sich für eines entscheiden, dass dem urpsrünglichen Problem L sehr ähnlich ist, da sonst die Reduktion sehr komplex wird.

Das erste $\mathcal{NP}$-vollständige Problem: SAT

Das Problem an dem eben beschriebenen Vorgehen ist, dass man zur Reduktion eines $\mathcal{NP}$-vollständigen Problems zuerst ein Problem braucht, von dem man schon weiß, dass es $\mathcal{NP}$-vollständig ist. Es muss also irgendein *erstes* $\mathcal{NP}$-vollständiges Problem geben. Das ist historisch gesehen das SAT-Problem. Für dieses Problem wurde tatsächlich gezeigt,

dass es möglich ist, **jedes** Problem in $\mathcal{NP}$ auf SAT zu reduzieren. Man spricht hier von einer *Master-Reduktion*.

Problembeschreibung

SAT ist eines der wichtigsten Probleme in der theoretischen Informatik. Obwohl die Problemstellung sehr einfach ist, ist das Lösen des Problems vermutlich nicht in polynomieller Zeit möglich. SAT ist eines der wichtigsten Probleme für Reduktionen, um $\mathcal{NP}$-Vollständigkeit zu zeigen. SAT ist ein Problem aus der Aussagenlogik.

Wie wir wissen[2], ist eine aussagenlogische Formel eine Verknüpfung von n boolschen Variablen (wir nennen sie hier $x_1, x_2, \ldots, x_n$) durch m Operatoren aus der Menge $\{\wedge, \vee, \neg, \rightarrow, \leftrightarrow\}$ und einige Klammern. Wir nehmen hier an, dass es keine redundanten Klammern gibt.

Definition 14.19. *Eine **Belegung** ist eine Funktion, die jeder der Variablen einen boolschen Wert (1 oder 0) zuweist. Eine **erfüllende Belegung** einer Formel F ist eine Belegung, für die F zu 1 auswertet. Eine Formel heißt **erfüllbar**, wenn es eine erfüllende Belegung gibt.*

Das SAT-Problem kann dann folgendermaßen formuliert werden:

$$\text{SAT} = \{< F >: F \text{ ist eine erfüllbare aussagenlogische Formel.}\}$$

Dieses Problem scheint nicht besonders schwierig zu sein. Der folgende Satz besagt allerdings das Gegenteil. Vorher stellen wir aber noch ohne Beweis fest, dass es möglich ist, eine aussagenlogische Formel mit n Variablen und m Operatoren in polynomieller Länge ($\mathcal{O}(n+m)$) zu codieren.

Satz 14.4. SAT *ist $\mathcal{NP}$-vollständig.*

Beweis. Wir zeigen hier nur, dass SAT $\in \mathcal{NP}$ ist. Durch eine nichtdeterministische Turing-Maschine kann man das schnell sehen. Diese muss nur eine Belegung für die gegebene Formel erraten und diese dann deterministisch verifizieren. Das Verifizieren einer Belegung ist offensichtlich in polynomieller Zeit möglich (man wertet die Operatoren der Reihenfolge nach aus). Als nächstes muss gezeigt werden, dass SAT auch $\mathcal{NP}$-schwer ist. Der Beweis ist sehr umfangreich und komplex und ist daher hier nicht aufgeführt. □

Das SAT-Problem wird schon seit einiger Zeit untersucht. Bisher ist es allerdings nicht gelungen zu zeigen, dass SAT in polynomieller Zeit deterministisch lösbar ist. Es wurden aber einige andere interessante Feststellungen gemacht:

[2] In dieser Veranstaltung gehen wir davon aus, dass Aussagenlogik bereits bekannt ist. Die Definitionen zu diesem Themengebiet sind formal nicht ausreichend und sind nur als Gedächtnisstütze zu verstehen.

- Es gibt einige große Klassen von Formeln, für die SAT tatsächlich in polynomieller Zeit lösbar ist. Allerdings gibt es immer noch Klassen von Formeln, für die das nicht geht.
- Die Dichte einer Formel bezeichnet das Verhältnis von der Anzahl der Literale zur Anzahl der Variablen. Es fällt auf, dass Formeln mit einer niedrigeren Dichte tendenziell einfacher zu lösen sind, als Formeln mit einer hohen Dichte.
- Tatsächlich gibt es einen ganz bestimmten Schwellwert. Überschreitet die Dichte diesen Wert, werden die entsprechenden Formeln auf einmal extrem schwer zu lösen. Unterhalb dieses Schwellwertes sind die meisten Formeln effizient lösbar.

CNF-SAT und 3-CNF-SAT

Wir betrachten nun noch zwei Probleme, die eng mit SAT verwandt sind. Für eines davon wollen wir dann die $\mathcal{NP}$-Vollständigkeit beweisen.

Definition 14.20 (Konjunktive Normalform, KNF)**.** *Ein **Literal** in einer Aussagenlogische Formel ist entweder eine Variable x_i oder deren Negation $\neg x_i$. Eine **Klausel** ist eine Verknüpfung von Literalen durch logische Oder-Operationen. Eine Formel ist in **konjunktiver Normalform (KNF)**, wenn sie nur aus Klauseln besteht, die durch Und-Operationen verknüpft sind. Eine Formel ist in **3-KNF**, wenn jede Klausel genau drei Literale enthält.*

Bemerke. Zu jeder Formel existiert eine äquivalente Formel in konjunktive Normalform. Zu jeder Formel existiert eine erfüllbarkeitsäquivalente Formel in 3-KNF.

Wir definieren nun zwei $\mathcal{NP}$-vollständige Probleme:

$$\text{CNF-SAT} = \{< F >: F \text{ ist in CNF und erfüllbar.}\}$$
$$\text{3-CNF-SAT} = \{< F >: F \text{ ist in 3-CNF und erfüllbar.}\}$$

Satz 14.5. CNF-SAT *ist $\mathcal{NP}$-vollständig.*

Beweis. Da SAT $\in \mathcal{NP}$ ist, ist auch CNF-SAT $\in \mathcal{NP}$. Dass für alle $L \in \mathcal{NP}$ $L \leq_p$ CNF-SAT ist, lässt sich beispielsweise durch eine Reduktion von SAT auf CNF-SAT zeigen. Den Beweis gehen wir hier nicht durch. □

Stattdessen betrachten wir die noch weiter vereinfachte Version 3-CNF-SAT. Gerade dieses Problem scheint intuitiv nicht extrem schwer zu lösen zu sein, da ja jede Klausel nur genau drei Literale enthält. Vor allem ist das verwandte Problem 2-CNF-SAT **nicht** $\mathcal{NP}$-vollständig. Allerdings werden wir nun beweisen, dass 3-CNF-SAT nicht so einfach ist, wie es intuitiv erscheinen mag.

Satz 14.6. 3-CNF-SAT *ist $\mathcal{NP}$-vollständig.*

Beweis. Da SAT $\in \mathcal{NP}$ ist, gilt dies offensichtlich auch für 3-CNF-SAT. Wir müssen nun nur noch zeigen, dass sich auch jedes Problem aus $\mathcal{NP}$ auf 3-CNF-SAT reduzieren lässt. Dies tun wir, indem wir CNF-SAT auf 3-CNF-SAT reduzieren. Zur Erinnerung: Dies bedeutet, dass wir eine Probleminstanz von CNF-SAT in eine Probleminstanz von 3-CNF-SAT umformen müssen.

Gegeben sei also eine aussagenlogische Formel $F = F_1 \wedge F_2 \wedge \cdots \wedge F_k$ in konjunktiver Normalform mit den Klauseln $F_1, F_2, \ldots, L_k$. Die Klauseln können noch mehr oder weniger als drei Literale enthalten. Wir wollen nun eine erfüllbarkeitsäquivalente Formel in 3-CNF konstruieren (das ist gerade die Reduktion). Wir unterscheiden einige Fälle für jede Klausel $F_i = \alpha_1 \vee \alpha_2 \vee \cdots \vee \alpha_\ell$. Dabei bezeichnet jedes α ein Literal, keine Variable.

Fall 1 F_i hat genau drei Literale. In dem Fall übernehmen wir die Klausel für F'.

Fall 2 F_i hat mehr als drei Literale. Dann führen wir neue Variablen in F' ein, die wir $y_1, y_2, \ldots, y_{\ell-3}$ nennen und ersetzen die Klausel F_i in F' durch:

$$\begin{aligned} F_i' = &(\alpha_1 \vee \alpha_2 \vee y_1)\wedge \\ &(\alpha_3 \vee \neg y_1 \vee y_2)\wedge \\ &(\alpha_4 \vee \neg y_2 \vee y_3) \wedge \cdots \wedge \\ &(\alpha_{\ell-2} \vee \neg y_{\ell-4} \vee y_{\ell-3})\wedge \\ &(\alpha_{\ell-1} \vee \alpha_\ell \vee \neg y_{\ell-3}) \end{aligned}$$

Wir beweisen nun, dass diese neue Formel F_i' auch tatsächlich erfüllbarkeitsäquivalent zu F_i ist.

$\Rightarrow$ Angenommen es gibt eine erfüllende Belegung für F_i. Dann gibt es ein α_j, dass in der erfüllenden Belegung zu 1 ausgewertet wird. Angenommen es ist $\alpha_1 = \cdots = \alpha_{j-1} = 0$ und $\alpha_j = 1$. Dann ist die Erweiterung dieser Belegung durch

$$\begin{aligned} y_1 = \cdots = y_{j-2} = 1 \\ y_{j-1} = \cdots = y_{\ell-3} = 0 \end{aligned}$$

eine erfüllende Belegung für F_i'.

$\Leftarrow$ Angenommen es gibt keine erfüllende Belegung für F_i, aber eine erfüllende Belegung für F_i'. Dann werden definitiv alle α zu 0 ausgewertet. Das bedeutet aber, dass in der ersten Klausel von F_i' y_1 zu 1 ausgewertet werden muss. Das heißt dann, dass auch y_2 in der zweiten Klausel zu 1 ausgewertet werden muss. Durch Induktion lässt sich zeigen, dass alle y_m zu 1 ausgewertet werden müssen. Dann ist allerdings die letzte Klausel von F_i' nicht erfüllt. Das ist ein Widerspruch.

Damit sind F_i und F_i' erfüllbarkeitsäquivalent.

Fall 3 F_i hat ein Literal. Dann führen wir neue Variablen y_1 und y_2 ein und ersetzen F_i in F' durch

$$\begin{aligned}(\alpha_1 \vee y_1 \vee y_2)\wedge \\ (\alpha_1 \vee y_1 \vee \neg y_2)\wedge \\ (\alpha_1 \vee \neg y_1 \vee y_2)\wedge \\ (\alpha_1 \vee \neg y_1 \vee \neg y_2)\end{aligned}$$

Man sieht sofort, dass diese Formel in 3-CNF erfüllbarkeitsäquivalent zu F_i ist.

Fall 4 F_i hat zwei Literale. Dann führen wir eine neue Variable y ein und ersetzen F_i in F' durch

$$F_i' = (\alpha_1 \vee \alpha_2 \vee y) \wedge (\alpha_1 \vee \alpha_2 \vee \neg y)$$

Auch hier sieht man, dass F_i und F_i' erfüllbarkeitsäquivalent sind.

Da die Anzahl neuer Klauseln in jedem der Fälle höchstens polynomial von der Anzahl der Variablen abhängt, lässt sich diese Reduktion in Polynomialzeit durchführen. Damit gilt CNF-SAT $\leq_p$ 3-CNF-SAT. □

Weitere $\mathcal{NP}$-vollständige Probleme

Clique

Definition 14.21 (Clique). *Eine Clique in einem ungerichteten Graphen $G = (V, E)$ ist eine Teilmenge $V' \subseteq V$, wobei jeder Knoten in V' mit jedem anderen Knoten in V' durch eine Kante in E verbunden ist. Der Teilgraph mit der Knotenmenge V soll also vollständig sein. Eine k-Clique ist eine Clique mit k Knoten.*

Wir fragen uns nun, was die größte mögliche Clique in einem gegebenen Graphen ist (Optimierungsproblem), bzw. ob es eine Clique der Größe k gibt (Entscheidungsproblem). Der naive Ansatz zum Lösen des Entscheidungsproblems ist, alle Teilmengen von V der Größe k aufzulisten und jede auf ihre Vollständigkeit zu überprüfen. Die Laufzeit dieses Algorithmus wäre allerdings mindestens

$$\Omega\left(k^2 \cdot \binom{|V|}{k}\right).$$

Diese Laufzeit ist im Allgemeinen nicht polynomiell. Wir zeigen daher jetzt, dass CLIQUE $\mathcal{NP}$-vollständig ist.

Satz 14.7. CLIQUE *ist $\mathcal{NP}$-vollständig.*

Beweis. Zuerst zeigen wir, dass CLIQUE $\in \mathcal{NP}$ ist. Dazu betrachten wir den folgenden Zertifizierungsalgorithmus:

1. Gegeben den Graphen $G = (V, E)$ sowie eine Teilmenge von Knoten von V' als Zertifikat.
2. Prüfe für jedes Knotenpaar $u, v \in V'$, ob $(u, v) \in E$ ist.
3. Dies lässt sich in $\mathcal{O}(|V|^2)$ bewerkstelligen.

Bleibt zu zeigen, dass CLIQUE auch $\mathcal{NP}$-schwer ist. Wir nehmen dafür eine Reduktion von 3-CNF-SAT vor. Sei also $F = C_1 \wedge C_2 \wedge \cdots \wedge C_k$ eine Formel in 3-CNF. Jede Klausel enthält dann drei Literale: $C_i = \alpha_1^i \vee \alpha_2^i \vee \alpha_3^i$. Daraus konstruieren wir nun einen Graphen, der genau dann eine k-Clique enthält, wenn F erfüllbar ist (bemerke: k ist die Anzahl der Klauseln in F). Diesen erstellen wir so:

1. Wir erstellen für jede Klausel $C_r = \alpha_1^r \vee \alpha_2^r \vee \alpha_3^r$ drei Knoten v_1^r, v_2^r, v_3^r im Graphen. Es ist also

$$V = \{v_i^r : \exists C_r \in F : \alpha_i^r \in C_r\}$$

2. Wir verbinden zwei Knoten v_i^r und v_j^s, nur dann, wenn die beiden Knoten aus unterschiedlichen Klauseln kommen (also $r \neq s$ ist) und a_i^r nicht die Negation von α_j^s ist.

Die Idee hinter dieser Konstruktion ist einfach: Wir erstellen einen Knoten für jedes Literal. Dann verbinden wir all diejenigen Literale aus unterschiedlichen Klauseln, die gleichzeitig erfüllt werden können. Wenn wir damit alle Klauseln erwischen, dann ist die Formel erfüllbar und es gibt eine Clique, sonst nicht. Offensichtlich ist dies in polynomieller Zeit möglich. Nun zeigen wir noch die Korrektheit dieser Reduktion formal:

$\Rightarrow$ Angenommen es gibt eine erfüllende Belegung für F. Dann gibt es in jeder Klausel C_r ein Literal α_i^r, das zu 1 ausgewertet wird. Für alle k Klauseln gibt es dann also entsprechend k Knoten im Graphen. Da nun in jeder Klausel ein entsprechendes Literal existiert, können die Literale nicht gegenseitig komplementär sein. Das heißt, dass es auch eine Kante im Graphen gibt und der Graph damit auch eine k-Clique enthält.

$\Leftarrow$ Angenommen F ist unerfüllbar, G enthält aber trotzdem eine k-Clique V'. Nach Konstruktion gibt es in G keine Kanten zwischen den Knoten, die die Literale einer Klausel repräsentieren. Das heißt, dass eine k-Clique genau ein Literal aus jeder Klausel erfüllen muss. Wenn es nun eine k-Clique gibt, heißt das, dass F erfüllbar wäre. Das ist ein Widerspruch.

Damit ist CLIQUE $\mathcal{NP}$-vollständig. $\square$

Independent-Set

Definition 14.22 (Independent-Set)**.** *Ein Independent-Set eines ungerichteten Graphen $G = (V, E)$ ist eine Teilmenge $V' \subseteq V$, wobei keine zwei Knoten in V' durch eine Kante aus E miteinander verbunden sind. Ein k-Independent-Set ist ein Independent-Set mit k Knoten.*

Wie schon bei CLIQUE ist der naive Ansatz, einfach alle Teilmengen von V der Größe k zu überprüfen. Die Laufzeit davon ist ebenfalls

$$\Omega\left(k^2 \cdot \binom{|V|}{k}\right).$$

Mithilfe die Stirling-Formel können wir zeigen, dass dies keine polynomielle Laufzeit ist. Dafür formen wir um:

$$\begin{aligned} k^2\binom{|V|}{k} &= \left(\frac{n}{2}\right)^2 \cdot \binom{n}{n/2} \\ &= \left(\frac{n^2}{4}\right) \cdot \frac{n!}{(\frac{n}{2})! \cdot (n - \frac{n}{2})!} \\ &= \left(\frac{n^2}{4}\right) \cdot \frac{n!}{((\frac{n}{2})!)^2} \end{aligned}$$

Hier können wir nun die Fakultät durch die Stirling-Formel abschätzen:

$$\begin{aligned} k^2 \cdot \binom{|V|}{k} &\approx \left(\frac{n^2}{4}\right) \cdot \frac{\sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n}{\left(\sqrt{2\pi\left(\frac{n}{2}\right)} \cdot \left(\frac{n}{2e}\right)^{\frac{n}{2}}\right)^2} \\ &= \left(\frac{n^2}{4}\right) \cdot \frac{\sqrt{2\pi n} \cdot \left(\frac{n}{e}\right)^n}{2\pi\left(\frac{n}{2}\right) \cdot \left(\frac{n}{2e}\right)^n} \\ &= \left(\frac{n^2}{4}\right) \cdot \frac{\sqrt{2\pi n}}{2\pi\left(\frac{n}{2}\right)} \cdot 2^n \\ &= \left(\frac{n^2}{4}\right) \cdot \frac{2\sqrt{2\pi n}}{2\pi n} \cdot 2^n = \left(\frac{n^2}{4}\right) \cdot \frac{2}{\sqrt{2\pi n}} \cdot 2^n \\ &= \frac{2n^2}{4\sqrt{2\pi} \cdot \sqrt{n}} \cdot 2^n = \frac{n^{\frac{3}{2}}}{2\sqrt{2\pi}} \cdot 2^n \\ &= \Omega(2^n) \end{aligned}$$

Diese Berechnung gilt genauso für CLIQUE. Wir wollen zuletzt noch zeigen, dass auch INDEPENDENT-SET $\mathcal{NP}$-vollständig ist.

Satz 14.8. INDEPENDENT-SET *ist $\mathcal{NP}$-vollständig.*

Beweis. Dass INDEPENDENT-SET $\in \mathcal{NP}$ ist, ist klar. Dies kann wie bei CLIQUE-geprüft werden. Wir wollen jetzt zeigen, dass CLIQUE $\leq_p$ INDEPENDENT-SET ist. Gegeben ist eine Probleminstanz (G, k). Wir konstruieren dazu einen Graphen $G' = (V, E')$ mit

$$E' = V \times V - \{(v, v) : v \in V\} - E$$

E' enthält also alle Kanten, die E nicht enthält (abgesehen von Kanten von einem Knoten zu sich selbst). Der Beweis, dass dieser Graph G' genau dann ein k-Independent-Set enthält, wenn G eine k-Clique enthält ist, trivial, da die beiden Probleme genau komplementär zueinander sind. Ein formaler Beweis bleibt daher als Übung für den Leser.

Da die Konstruktion von G' offensichtlich in polynomieller Zeit möglich ist, ist INDEPENDENT-SET $\in \mathcal{NPC}$. □

A. Schleifeninvarianten

Der Begriff **Invariante** bezeichnet eine Aussage (normalerweise im aussagenlogischen Sinne), die in einem Teil eines Programms oder Algorithmus gültig ist. Da sich der Wahrheitswert der Aussage an verschiedenen Stellen im Programm nicht ändert, ist sie unveränderlich, also invariant.

Eine Schleifeninvariante ist eine Invariante, die über eine Schleife in einem Algorithmus gilt. Das besondere an einer Schleifeninvariante ist, dass es eine sehr klare Vorgehensweise gibt, mit der man die Gültigkeit von Schleifeninvarianten beweisen kann. Diese Beweistechnik ist der vollständigen Induktion sehr ähnlich.

Als erstes muss eine Aussage definiert werden, die bewiesen werden soll (das ist unsere Invariante). Bei der vollständigen Induktion ist so eine Aussage beispielsweise ein Satz oder eine Formel, die bewiesen werden sollen. Wie man so eine Invariante findet, wird unten noch kurz diskutiert. Der Beweis der Gültigkeit der Schleifeninvariante besteht dann aus drei Teilen: Initialisierung, Aufrechterhaltung und Terminierung.

Initialisierung Die Initialisierung entspricht dem Induktionsanfang. Hier müssen wir zeigen, dass die Aussage, die wir beweisen wollen, erfüllt ist, **bevor** die Schleife das erste Mal betreten wird.

Aufrechterhaltung Die Aufrechterhaltung entspricht dem Induktionsschritt. Analog zur Induktionsannahme nehmen wir hier an, dass unsere Aussage für die ersten n Schleifendurchläufe erfüllt ist. Normalerweise schreibt man diese Annahme nicht explizit auf, da sie für alle Schleifeninvarianten gleich ist. In der Aufrechterhaltung müssen wir dann zeigen, dass die Aussage nach dem $n+1$-ten Durchlauf weiterhin gültig ist.

Terminierung Die Terminierung hat keine Analogie bei der Induktion. Für Algorithmen ist wichtig, dass diese auch immer terminieren. Das gilt auch für Schleifen. Wenn unsere Annahme immer erfüllt ist, aber die Schleife nie verlassen wird, dann ist der Algorithmus trotzdem falsch. Daher müssen wir bei der Terminierung noch zeigen, dass die Schleife auch in jedem Fall irgendwann verlassen wird. Dann sind wir fertig und haben durch die Invariante die Korrektheit der Behauptung bewiesen, die wir aufgestellt haben.

Bei der Induktion ist es nicht nötig die Terminierung zu zeigen. Tatsächlich wäre es sogar von Nachteil, wenn es eine Terminierung bei Induktion gäbe, da das bedeuten würde, dass eine Formel nur für $n < c$ gilt, wobe $c \in \mathbb{N}$ eine Konstante ist.

Finden von Invarianten

Es gibt keine allgemeine Methode, durch die man *die richtige* Invariante für eine Schleife findet. Hier muss man sich also anders behelfen. Dabei sollte man folgende Punkte berücksichtigen:

- Die Schleifeninvariante muss in (bzw. vor) jedem Schleifendurchlauf gelten. Eine Aussage wie „Der Algorithmus liefert das richtige Ergebnis“ erfüllt diese Anforderung normalerweise nicht.
- Es ist hilfreich, wenn man eine Aussage über eine Variable im Programm trifft, die sich in der Schleife verändert. Sonst wäre die Invariante völlig unabhängig von der Schleife und wir könnten entweder die Schleife im Programm weglassen, oder (was wahrscheinlicher ist) die Invariante bringt uns nicht weiter.
- Die Aussage sollte in irgendeiner Form zielbringend sein. Im Idealfall nähert sich ein Wert in jedem Durchlauf einem optimalen Wert an. Dann könnte eine Invariante zum Beispiel sein, dass der aktuelle Wert der „bisher beste Wert“ ist.
- Wenn man möchte, dass eine Aussage nur in bestimmten Schleifendurchläufen gilt, kann man diese als Implikation formulieren. Zum Beispiel: „Wenn i gerade ist, gilt...“. Dann macht man zwar über ungerade i keine Aussage, das ist aber unter Umständen auch kein Problem (zum Beispiel wenn man schon weiß, dass i im letzten Durchlauf gerade sein wird).
- Man kann auch mehrere Aussagen kombinieren und gleichzeitig in einem Beweis bearbeiten. Manchmal ist das nötig, um gewisse Randbedingungen nicht zu verletzen.

B. Grundlagen zur Wahrscheinlichkeitsrechnung

In diesem Teil werden kurz die Grundlagen zur Wahrscheinlichkeitsrechnung erläutert. Dies ist in der Algorithmik für bestimmte randomisierte Algorithmen und deren Beweise von elementarer Bedeutung.

Elementare Kombinatorik

In dem ersten Abschnitt dieses Kapitels geht es darum, zu zählen. Dies ist in einfachen Fällen bereits zum Berechnen der Wahrscheinlichkeit eines Ereignisses ausreichend. Außerdem vermittelt die Anzahl der Möglichkeiten, die es in einem Szenario gibt, oft ein deutlich besseres Gefühl darüber, wie die Wahrscheinlichkeit eines Ereignisses sein müsste.

Wir wollen nun also einige Grundlagen darüber kennen lernen, wie man die Anzahl der Möglichkeiten (Kombinationen) von Ereignissen berechnen kann. Dies nennt man in der Mathematik *Kombinatorik*. Als Beispiel betrachten wir SWIFT-Codes. SWIFT-Codes sind 11-stellige alphanumerische Folgen, die nach dem Format `AAAA BB CC DDDD` gebildet werden. Dabei sind:

- `AAAA` 4 Buchstaben, die die Bank identifizieren.
- `BB` zwei Buchstaben, die als Landescode zu interpretieren sind (nach ISO 3166-1-alpha-2).
- `CC` Zwei Buchstaben oder Ziffern, die einen Ort codieren.
- `DDD` Drei Buchstaben oder Ziffern, die den Zweig identifizieren.

Die Frage ist nun: Wie viele mögliche SWIFT-Codes gibt es?

Um diese Frage zu beantworten, schreiben wir nicht einfach der Reihe nach alle SWIFT-Codes auf. Das würde viel zu lange dauern. Stattdessen wollen wir berechnen, wie viele Kombinationen es gibt. In der Kombinatorik gilt:

1. Eine Verkettung von Möglichkeiten (sprachlich „und“) entspricht einer Multiplikation.
2. Eine Alternative von Möglichkeiten (sprachlich „oder“) entspricht einer Addition.

Für die SWIFT-Codes bedeutet dies:

- Es gibt 26 Buchstaben, also $26 \cdot 26 \cdot 26 \cdot 26$ Möglichkeiten `AAAA` zu bilden.
- Es gibt $26 \cdot 26$ Möglichkeiten für `BB`.
- Für ein `C` sind Buchstaben **oder** Ziffern erlaubt. Also gibt es $(26+10) \cdot (26+10)$ Möglichkeiten.
- Analog zu `C`: $(26+10)^3$

Insgesamt gibt es also $456976 \cdot 249 \cdot 1296 \cdot 46656 \approx 4.57 \cdot 10^{17}$ verschiedene SWIFT-Codes.

Permutationen

Eine Permutation ist eine beliebige Anordnung einer Menge von Elementen, wobei jedes Element nur ein Mal verwendet werden darf. Zum Beispiel hat die Menge $\{a, b, c\}$ die 6 Permutationen *abc*, *acb*, *bac*, *bca*, *cab*, *cba*. Viele stochastische Probleme lassen sich auf den Formalismus von Permutationen zurückführen, daher werden wir nun einige grundlegende Rechnungen mit Permutationen kennen lernen.

Gegeben ist eine Menge mit n Elementen. Wir suchen zunächst die Anzahl der Permutationen. Diese berechnet sich durch $n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$. Intuitiv lässt sich das leicht veranschaulichen: Wir wissen, dass wir als Ergebnis eine Folge von n Elementen erhalten. Wenn wir nun nach und nach die Elemente der Menge auf diese Folge zuordnen, haben wir für das erste Element n Möglichkeiten. Für das zweite Element verbleiben dann nur noch $n-1$ mögliche Plätze usw. Nach der Regel „Verkettung entspricht Multiplikation“ ist also die Gesamtzahl der Möglichkeiten $n!$.

k-Permutationen

k-Permutationen sind eine besondere Art von Permutationen. Definiert sind k-Permutationen als die k-elementigen Teillisten der Permutationen. Auch hier gilt, dass keine Duplikate erlaubt sind.

Man kann sich bereits intuitiv erklären, wie sich die Anzahl der k-Permutationen berechnet: Wir erhalten als Ergebnis eine k elementige Liste. Für das erste Element können wir aus allen n Elementen der Menge wählen. Für das zweite nur noch aus $n-1$ usw. Also ist die Anzahl der k-Permutationen

$$n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-k+1) = \frac{n!}{(n-k)!}$$

k-Kombinationen

k-Kombinationen sind eng verwandt mit den k-Permutationen. Im Gegensatz zu Permutationen interessiert uns hier allerdings die Reihenfolge der gewählten Elemente nicht mehr. Während also die beiden 2-Permutationen ab und ba unterschiedlich sind, sind die beiden 2-Kombinationen $\{a, b\}$ und $\{b, a\}$ gleich. Dies verringert die Anzahl möglicher Kombinationen im Gegensatz zu den Permutationen nochmal um die Anzahl der möglichen Permutationen von k Elementen (Erinnerung: Da es sich hier um Verknüpfungen handelt, ist eine Verringerung keine Subtraktion sondern eine Division). Die Anzahl der möglichen Kombinationen ist dann

$$\frac{n!}{k!(n-k)!} = \binom{n}{k}$$

Dies wird auch als **Binomialkoeffizient** bezeichnet.

Der Binomialkoeffizient ist für die grundlegende Wahrscheinlichkeitsrechnung von elementarer Bedeutung. Damit lassen sich viele einfach Probleme lösen wie zum Beispiel die Frage „Wie wahrscheinlich ist es, im Lotto zu gewinnen?“ Dies lässt sich direkt berechnen. Es gewinnt genau eine 6er Kombination von Zahlen. Die Wahrscheinlichkeit ist:

$$P(\text{„Im Lotto gewinnen“}) = \frac{1}{\binom{49}{6}}$$

Im Rahmen der Algorithmik ist der genaue Wert des Binomialkoeffizienten häufig gar nicht so wichtig. Viel wichtiger ist sein asymptotisches Verhalten (da der Umgang mit Fakultäten oft nicht sehr schön ist). Eine untere Schranke ergibt sich schnell:

$$\begin{aligned}\binom{n}{k} &= \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{k \cdot (k-1) \cdot \ldots \cdot 2 \cdot 1} \\ &= \left(\frac{n}{k}\right)\left(\frac{n-1}{k-1}\right) \cdots \left(\frac{n-k+1}{1}\right) \\ &\geq \left(\frac{n}{k}\right)^k\end{aligned}$$

Um eine obere Schranke zu erhalten, brauchen wir die sog. Stirling-Formel:

Definition B.1 (Stirlingformel)**.** *Eine untere Schranke für die Fakultätsfunktion ergibt sich durch*

$$k! \geq \left(\frac{k}{e}\right)^k$$

Dabei ist e die Euler'sche Zahl.

Damit lässt sich für den Binomialkoeffizienten eine obere Schranke bestimmen:

$$\begin{aligned}\binom{n}{k} &= \frac{n \cdot (n-1) \cdot \ldots \cdot (n-k+1)}{k \cdot (k-1) \cdot \ldots \cdot 2 \cdot 1} \\ &\leq \frac{n^k}{k!} \\ &\leq \left(\frac{en}{k}\right)^k\end{aligned}$$

Das Urnenmodell

Das Urnenmodell ist eine verallgemeinerte mathematische Beschreibung für den Versuchsaufbau von bestimmten Wahrscheinlichkeitsrechnungen. Man unterscheidet meistens vier Varianten:

1. Ohne Reihenfolge, mit Zurücklegen
2. Ohne Reihenfolge, ohne Zurücklegen
3. Mit Reihenfolge, mit Zurücklegen
4. Mit Reihenfolge, ohne Zurücklegen

Die Berücksichtigung der Reihenfolge entspricht genau dem Unterschied zwischen Permutationen und Kombinationen (s.o.). Mit oder ohne Zurücklegen entspricht dem Erlauben oder Verbieten von Duplikaten. Bisher haben wir Duplikate immer verboten. Wenn wir nun Duplikate erlauben, dann ist die Anzahl der möglichen Anordnungen von k Elementen (aus einer Auswahl von n) genau n^k. Ohne das Erlauben von Duplikaten ist es $n!$. Diese Fälle sind aber für unsere Anwendungsfälle weniger relevant.

Anmerkung: Hier wurde das Urnenmodell nicht vollständig beschrieben, sondern nur ein kurzer Überblick gegeben.

Wahrscheinlichkeit

Nachdem wir nun wissen, wie man Permutationen und Kombinationen berechnet, wollen wir auch das Berechnen von Wahrscheinlichkeiten einführen. Dies tun wir folgendermaßen:

Definition B.2. *Ein **Ereignisraum** S ist eine Menge von sogenannten **Elementarereignissen**. Ein Elementarereignis ist ein möglicher Ausgang eines Experimentes (z.B. ein bestimmter SWIFT-Code). Der Ereignisraum S wird auch **Grundgesamtheit** genannt.*

Beispiel B.1. *Für das zweimalige Werfen einer Münze besteht der Ereignisraum aus vier Elementarereignissen: $S = \{KK, KZ, ZK, ZZ\}$.*

Definition B.3. *Ein **Ereignis** ist eine Teilmenge des Ereignisraums.*

Meistens wird ein Ereignis in natürlichsprachiger Form angegeben. Eine Schwierigkeit besteht dann darin, herauszufinden, welche Elementarereignisse zu diesem Ereignis gehören. Ein Beispielereignis beim zweimaligen Münzwurf könnte beispiel sein „Einmal wird Kopf geworfen“. Dieses Ereignis besteht aus den Elementarereignissen KZ und ZK.

Wir unterscheiden bestimmte Ereignisarten:

- *Das Ereignis $E = S$ (Ereignisraum) ist das **sichere Ereignis**.*
- *Das Ereignis $E = \emptyset$ ist das **Nullereignis**.*
- *Zwei Ereignisse A und B schließen sich gegenseitig aus, wenn $A \cap B = \emptyset$ ist.*
- *Elementare Ereignisse s werden oft als Ereignisse $\{s\}$ behandelt.*

Nach Definition befinden sich alle Elementarereignisse im gegenseitigen Ausschluss.

Wahrscheinlichkeitsverteilung

Definition B.4. *Eine Wahrscheinlichkeitsverteilung* $\Pr\{.\} : \mathcal{P}(S) \to \mathbb{R}$ *ist eine Abbildung von Ereignissen in die reellen Zahlen, sodass gilt:*

1. $\forall A \subseteq S : Pr\{A\} \geq 0$
2. $Pr\{S\} = 1$
3. $A \cap B = \emptyset \implies Pr\{A \cup B\} = Pr\{A\} + Pr\{B\}$

Wir nennen $\Pr\{A\}$ *die **Wahrscheinlichkeit** des Ereignisses A.*

Schreibweise

Wir schreiben hier die Funktion Pr mit Mengenklammern, um zu verdeutlichen, dass die Funktion **Mengen** auf Zahlen abbildet.

Durch eine solche Abbildung lässt sich die Wahrscheinlichkeit beliebiger Ereignisse berechnen. Aus den obigen Axiomen folgen direkt einige Rechenregeln:

- Für zwei Ereignisse A, B mit $A \subseteq B$ gilt $\Pr\{A\} \leq \Pr\{B\}$.
- Für das Gegenereignis $\overline{A} = S \setminus A$ gilt $\Pr \overline{A} = 1 - \Pr\{A\}$.
- Für zwei Ereignisse A, B gilt $\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\} - \Pr\{A \cap B\} \leq \Pr\{A\} + \Pr\{B\}$.

Im Beispiel des zweimaligen Münzwurfes nehmen wir an, dass jedes Elementarereignis die Wahrscheinlichkeit $\frac{1}{4}$ hat. Dann ist die Wahrscheinlichkeit des Ereignisses „Mindestens ein Mal wird Kopf geworfen“

$$\Pr\{KK, KZ, ZK\} = \Pr\{KK\} + \Pr\{KZ\} + \Pr\{ZK\} = \frac{3}{4}.$$

Alternativ können wir auch über das Gegenereignis argumentieren: Die Wahrscheinlichkeit, kein Mal Kopf zu werfen ist $\Pr\{ZZ\} = \frac{1}{4}$, also ist die Gegenwahrscheinlichkeit $1 - \frac{1}{4} = \frac{3}{4}$.

Diskrete Wahrscheinlichkeitsverteilungen

Wir betrachten in dieser Veranstaltung nur diskrete Wahrscheinlichkeitsverteilungen. Das sind solche, die einen endlichen oder abzählbar unendlichen Ereignisraum haben. In solchen Verteilungen berechnet sich die Wahrscheinlichkeit eines Ereignisses A durch

$$\Pr\{A\} = \sum_{s \in A} \Pr\{s\}$$

Außerdem kommt es sehr häufig vor, dass alle Elementarereignisse dieselbe Wahrscheinlichkeit haben, nämlich $\frac{1}{|S|}$. Dies nennen wir eine **Gleichverteilung** (*uniform probability distribution*). Das ist zwar nicht immer der Fall, vereinfacht aber viele Rechnungen erheblich, da es dann möglich ist, einfach über Permutationen und Kombinationen zu argumentieren, ohne besondere Bedingungen berücksichtigen zu müssen. Für ein Ereignis A in einer Gleichverteilung ist

$$\Pr\{A\} = \frac{|A|}{|S|}.$$

Beispiel B.2. *Wir betrachten nun einen n fachen Münzwurf. Wir nehmen an, dass die Münze fair ist, dass also Kopf und Zahl gleich wahrscheinlich sind. Der Ereignisraum ist dann jede Kombination aus n Kopf- oder Zahl-Zeichen: $S = \{K, Z\}^n$. Es ist $|S| = 2^n$. Die Wahrscheinlichkeit eines Elementarereignisses (eine Kombination aus n Zeichen) ist dann $\frac{1}{2^n}$. Wir betrachten das Beispiel $A = \{$„Es wird genau k Mal Kopf geworfen“$\}$. Es ist hier also $|A| = \binom{n}{k}$, da es genau so viele Möglichkeiten gibt, Zeichenketten der Länge n zu konstruieren, die genau k Ks enthalten. Damit ist die Wahrscheinlichkeit*

$$\Pr\{A\} = \frac{\binom{n}{k}}{2^n}.$$

Bedingte Wahrscheinlichkeiten

Bedingte Wahrscheinlichkeiten treten auf, wenn wir bereits einen Teil über den Ausgang eines Experimentes wissen. Angenommen wir wissen beim zweimaligen Münzwurf

bereits, dass *mindestens* ein Wurf Kopf ergeben wird. Die Frage ist nun, wie hoch die Wahrscheinlichkeit ist, dass beide Male Kopf geworfen wird?

In diesem Beispiel ist die Rechnung relativ einfach. Wir wissen, dass das Elementarereignis ZZ nicht mehr möglich ist. Damit bleiben noch drei Ereignisse, von denen nur eines günstig ist. Das entspricht einer Wahrscheinlichkeit von $\frac{1}{3}$.

Im Allgemeinen haben wir also zwei Ereignisse: Ein Ereignis A von dem wir die Wahrscheinlichkeit berechnen wollen und ein Ereignis B von dem wir bereits wissen wie hoch die Wahrscheinlichkeit ist, dass es eintritt. Man schreibt:

$$\Pr\{A|B\} = \frac{\Pr\{A \cap B\}}{\Pr\{B\}}$$

Dies geht natürlich nur, wenn $\Pr\{B\} \neq 0$ ist. Man spricht dies „A gegeben B".

So können wir nun das obige Beispiel auch ausrechnen: Hier ist nun A das Ereignis, dass beide Münzen Kopf zeigen und B das Ereignis, dass mindestens eine Münze Kopf zeigt. Wir rechnen:

$$\Pr\{A|B\} = \frac{\frac{1}{4}}{\frac{3}{4}} = \frac{1}{3}.$$

Unabhängigkeit

Definition B.5 (Unabhängigkeit)**.** *Die Ereignisse A und B sind* ***unabhängig****, wenn* $\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\}$ *ist. Dies ist äquivalent zu* $\Pr\{A|B\} = \Pr\{A\}$*.*

Beispiel B.3. *Wir werfen einen Würfel* 100 *Mal. Wie hoch ist die Wahrscheinlichkeit, dass wir beim* 100*-ten Wurf eine* 6 *werfen, wenn wir wissen, dass die* 99 *Würfe davor keine einzige* 6 *enthalten haben?*

$$\Pr\{\text{„Eine 6 werfen"}|\text{„99 Mal keine 6 werfen"}\} = \frac{1}{6}$$

Im ersten Moment mag dies überraschend sein, allerdings muss man sich klar machen, dass einzelne Würfe eines Würfels unabhängig voneinander sind. Der „Fehler" der Intuition liegt hier darin, dass bereits das Ereignis, 99 Mal keine 6 zu werfen extrem niedrig ist. Da wir das aber als gegeben annehmen, bleibt nur ein einziger Würfelwurf.

Satz von Bayes

Wir können die Formel aus dem obigen Abschnitt folgendermaßen umformen:

$$\Pr\{A \cap B\} = \Pr\{B\} \cdot \Pr\{A|B\} = \Pr\{A\} \cdot \Pr\{B|A\}$$

Eine weitere Umformung ergibt

$$\Pr\{A|B\} = \frac{\Pr\{A\} \cdot \Pr\{B|A\}}{\Pr\{B\}}$$

Dies ist bekannt als der **Satz von Bayes**. Durch ähnliche Umformungen lässt sich auch noch $\Pr\{B\}$ ersetzen:

$$\Pr\{B\} = \Pr\{B \cap A\} + \Pr\{B \cap \overline{A}\} = \Pr\{A\} \cdot \Pr\{B|A\} + \Pr\{\overline{A}\} \cdot \Pr\{B|\overline{A}\}$$

Diskrete Zufallsvariablen

Definition B.6 (Zufallsvariable)**.** *Eine* ***Zufallsvariable*** $X : S \to \mathbb{R}$ *ist eine Funktion aus dem endlichen Ereignisraum S in die reellen Zahlen. Sie weist jedem Ausgang eines Zufallsexperimentes eine Zahl zu, was uns ermöglicht, mit der Wahrscheinlichkeitsverteilung der resultierenden Zahlen zu arbeiten.*

Definition B.7. *Für eine Zufallsvariable X und eine Zahl $x \in \mathbb{R}$ definieren wir das Ereignis $X = x$ als die Menge $\{s \in S : X(s) = x\}$. Damit gilt*

$$\Pr\{X = x\} = \sum_{s \in S : X(s) = x} \Pr\{s\}.$$

Definition B.8 (Wahrscheinlichkeitsdichtefunktion)**.** *Die Funktion $f(x) = \Pr\{X = x\}$ ist die* ***Wahrscheinlichkeitsdichtefunktion*** *der Zufallsvariable X. Aus den obigen Axiomen der Wahrscheinlichkeitsrechnung ergibt sich, dass* $\Pr\{X = x\} \geq 0$ *und* $\sum_x \Pr\{X = x\} = 1$ *ist.*

Multivariate Wahrscheinlichkeitsdichtefunktion

Oft haben wir es nicht nur mit einer, sondern mit mehreren Zufallsvariablen in demselben Ereignisraum zu tun. Wir nennen diese hier X und Y.

Definition B.9. *Für zwei Zufallsvariablen X und Y ist die* ***multivariate Wahrscheinlichkeitsdichtefunktion*** *die Funktion $f(x, y) = \Pr\{X = x \text{ und } Y = y\}$.*

Es gelten folgende Äquivalenzen:

$$\Pr\{Y = y\} = \sum_x \Pr\{X = x \text{ und } Y = y\}$$
$$\Pr\{X = x\} = \sum_y \Pr\{X = x \text{ und } Y = y\}$$
$$\Pr\{X = x | Y = y\} = \frac{\Pr\{X = x \text{ und } Y = y\}}{\Pr\{Y = y\}}$$

Definition B.10 (Unabhängigkeit)**.** *Wir nennen zwei Zufallsvariablen X und Y* ***unabhängig****, wenn für alle $x, y \in S$ die Ereignisse $X = x$ und $Y = y$ unabhängig sind. Eine äquivalente Definition ist*

$$\forall x, y \in S : \Pr\{X = x \textit{ and } Y = y\} = \Pr\{X = x\} \cdot \Pr\{Y = y\}$$

Erwartungswert

In der Algorithmik ist es normalerweise gar nicht so wichtig, welche konkreten Werte eine Zufallsvariable tatsächlich annehmen kann. Viel wichtiger ist, was wir im Durchschnitt von einer Zufallsvariablen erwarten können. Genau das liefert uns der Erwartungswert.

Definition B.11 (Erwartungswert)**.** *Der **Erwartungswert** einer diskreten Zufallsvariablen X ist*

$$E[X] = \sum_x x \cdot \Pr\{X = x\}$$

Manchmal schreibt man statt $E[X]$ auch μ.

Beispiel B.4. *Wir betrachten ein Spiel, in dem zwei faire Münzen geworfen werden. Jedes Mal wenn ein Kopf auftritt, gewinnt man 3€. Für jede auftretende Zahl verliert man 2€. Dann ist der Erwartungswert der Zufallsvariablen X (der erwartete Gewinn):*

$$\begin{aligned} E[X] &= 6 \cdot \Pr\{\text{„2 Mal Kopf“}\} \\ &\quad + (3-2) \cdot \Pr\{\text{„1 Mal Kopf, 1 Mal Zahl“}\} \\ &\quad - 4 \cdot \Pr\{\text{„2 Mal Zahl“}\} \\ &= 6 \cdot \frac{1}{4} + 1 \cdot \frac{1}{2} - 4 \cdot 14 = 1 \end{aligned}$$

Wichtig hierbei ist, dass wir nicht wissen, ob der Erwartungswert bei einem konkreten Experiment tatsächlich eintritt. Wir wissen nur, dass wenn man das Experiment sehr oft wiederholt, zu erwarten ist, dass man im Durchschnitt 1€ pro Spiel gewinnt.

Varianz und Standardabweichung

Ein Problem beim Erwartungswert ist noch, dass wir nicht wissen, wie stark die tatsächlichen Werte darum herum gestreut sind. Dafür ist die Varianz da. Das Spiel ist deutlich gefährlicher, wenn es nicht um 3€ und −2€ geht, sondern um 3000€ und −2999€. Der Erwartungswert ist aber in beiden Fällen gleich.

Definition B.12 (Varianz)**.** *Die Varianz berechnet sich durch*

$$\begin{aligned} Var[X] &= E\left[(X - E[X])^2\right] \\ &= E\left[X^2 - 2XE[X[+E^2[X]\right] \\ &= E\left[X^2\right] - 2E[XE[X]] + E^2[X] \\ &= E\left[X^2\right] - 2E^2[X] + E^2[X] \\ &= E\left[X^2\right] - E^2[X] \end{aligned}$$

Definition B.13 (Standardabweichung)**.** *Standardabweichung ist als die Wurzel aus der Varianz $\sigma = \sqrt{\mu}$ definiert. Umgekehrt schreibt man manchmal für die Varianz σ^2.*

Normalverteilung

Manchmal möchte man für eine Zufallsvariable wissen, ob die Abweichung eines konkreten Experimentes vom Erwartungswert signifikant ist, bzw. man möchte wissen, in welchem Bereich der Großteil der erwarteten Werte liegt. Dann kann man eine Normalverteilung verwenden. Diese ist für eine Zufallsvariable definiert mit der Wahrscheinlichkeitsdichtefunktion

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}.$$

In so einer Normalverteilung wird normalerweise eine Abweichung von mehr als $\mu \pm 3\sigma$ als signifikant angesehen.