Diskrete und Geometrische Algorithmen

Stefan Hetzl
stefan.hetzl@tuwien.ac.at

TU Wien

WS 2018/19

Inhaltsverzeichnis

Vorwort

Dieses Skriptum begleitet die im Wintersemester 2018/19 an der TU Wien gehaltene Vorlesung *Diskrete und geometrische Algorithmen*. Bis auf kleinere Korrekturen ist es identisch mit der während des Semesters Einheit für Einheit auf TISS publizierten Version. Als ergänzende Literatur können zum Thema der Vorlesung die englischsprachigen Lehrbücher [2, 5, 3] sowie die deutschsprachigen Lehrbücher [6, 7] empfohlen werden. Zur vertiefenden Lektüre kann zur Analyse von Algorithmen [8], zur Kryptographie [1], zur linearen Optimierung [9] sowie zu geometrischen Algorithmen [4] empfohlen werden.

Kapitel 1

Einführung

1.1 Berechnungsprobleme und Algorithmen

Definition 1.1. Ein *Berechnungsproblem* ist eine Relation der Form $P \subseteq X \times Y$ so dass für alle $x \in X$ ein $y \in Y$ existiert mit $(x, y) \in P$.

Dabei stellen wir uns X als Menge der möglichen Eingaben vor, Y als Menge der möglichen Ausgaben und $(x, y) \in P$ als die Aussage "bei Eingabe x ist y eine korrekte Ausgabe". Oft werden wir statt Berechnungsproblem einfach nur Problem sagen. Ein Berechnungsproblem ist aber zu unterscheiden von einem mathematischen Problem, bei welchem es sich (üblicherweise) um eine Frage der Form "Ist die Aussage ... wahr?" handelt. Beispiele für Berechnungsprobleme sind:

Bestimmung des ggT
Eingabe: positive ganze Zahlen n_1 und n_2
Ausgabe: der größte gemeinsame Teiler von n_1 und n_2

Sortierproblem
Eingabe: eine endliche Folge ganzer Zahlen $(a_1, \ldots, a_n)$
Ausgabe: eine Permutation $(a_{\pi(1)}, \ldots, a_{\pi(n)})$ so dass $a_{\pi(1)} \leq \cdots \leq a_{\pi(n)}$

Linearisierung
Eingabe: eine endliche partiell geordnete Menge $(A, \leq)$
Ausgabe: eine totale Ordnung $a_1, \ldots, a_n$ der Elemente von A so dass $a_i \leq a_j \Rightarrow i \leq j$

Wie man am dritten der obigen Beispiele sehen kann, muss ein Berechnungsproblem nicht unbedingt eine eindeutige Lösung haben. Falls $P \subseteq X \times Y$ ein Problem ist, dann heißt jedes $x \in X$ *Instanz von* P, beispielsweise ist $(3, 7, 2, 5, 8)$ eine Instanz des Sortierproblems.

Definition 1.2. Ein *Algorithmus* ist eine wohldefinierte Rechenvorschrift.

Wir geben hier keine präzisere Definition des Begriffs Algorithmus. Dies zu tun würde im Wesentlichen auf die Definition einer Programmiersprache hinauslaufen und damit am Thema dieser Vorlesung vorbeigehen. Wir werden konkrete Algorithmen in *Pseudocode* angeben, d.h. in einer der Situation angepassten Mischung aus natürlicher Sprache und üblichen Anweisungen und Kontrollstrukturen einer Programmiersprache wie Schleifen, Verzweigungen, usw. Wir verlangen auch nicht formell dass jeder Algorithmus *terminiert*, d.h. dass er für jede Eingabe nach endlicher Zeit stoppt. Allerdings werden wir in dieser Vorlesung fast ausschließlich terminierende Algorithmen betrachten.

Wir kennen bereits viele Algorithmen, zum Beispiel Algorithmen zur Addition und Multiplikation zweier natürlicher Zahlen in Dezimaldarstellung, wie sie in der Volksschule gelehrt werden, den Algorithmus zur Division mit Rest von Polynomen, das gaußsche Eliminationsverfahren zur Lösung linearer Gleichungssysteme, usw. Algorithmen sind aus der Mathematik nicht wegzudenken. Anders als bisher werden wir in dieser Vorlesung ein systematisches Studium von Algorithmen betreiben. Dieses beschränkt sich nicht auf die bloße Definition und Verwendung von Algorithmen, sondern untersucht Fragen wie z.B. "Wie effizient ist dieser Algorithmus?" "Wie ist das Verhältnis zwischen zwei Algorithmen?" "Welche Ansätze zur Entwicklung guter Algorithmen gibt es?" "Ist dieser Algorithmus der beste zur Lösung dieses Problems? In welchem Sinn ist er das?" usw. Algorithmen werden also von einem Mittel zur Lösung von Problemen zu einem Objekt unserer Untersuchungen.

Algorithmen werden verwendet um Berechnungsprobleme zu lösen. Damit meint man Folgendes:

Definition 1.3. Sei $P \subseteq X \times Y$ ein Berechnungsproblem und $\mathcal{A}$ ein Algorithmus. Wir sagen dass $\mathcal{A}$ *das Problem* P löst falls für jede Instanz $x \in X$ gilt dass $y = \mathcal{A}(x)$ die Eigenschaft $(x, y) \in P$ hat.

Oft wird aus dem Kontext heraus klar sein, welches Berechnungsproblem wir lösen wollen, dann sprechen wir einfach von der *Korrektheit* eines Algorithmus. Ein bekannter Algorithmus ist der euklidische Algorithmus. Dieser bestimmt den größten gemeinsamen Teiler zweier positiver ganzer Zahlen.

Beispiel 1.1. Wir bestimmten $\mathrm{ggT}(24, 90)$ mit dem euklidischen Algorithmus:

90	24	18	6
24	18	6	0

Also $\mathrm{ggT}(24, 90) = 6$.

Der euklidische Algorithmus kann wie folgt in Pseudocode aufgeschrieben werden.

Algorithmus 1 Der euklidische Algorithmus

```
Prozedur EUKLID(a, b)
    Solange b > 0
        t := a
        a := b
        b := t mod b
    Ende Solange
    Antworte a
Ende Prozedur
```

Die Korrektheit des euklidischen Algorithmus, d.h. die Tatsache dass dieser Algorithmus für alle positiven ganzen Zahlen a und b als Antwort $\mathrm{ggT}(a, b)$ liefert ist auf den ersten Blick nicht offensichtlich. Der Korrektheitsbeweis erfordert ein wenig Arbeit.

Satz 1.1. *Der euklidische Algorithmus ist korrekt, d.h. er löst das Problem der Bestimmung des größten gemeinsamen Teilers zweier positiver ganzer Zahlen.*

Beweis. Zunächst behaupten wir:

$$\text{für alle } a \geq b \geq 1 \text{ gilt: } \mathrm{ggT}(a, b) = \mathrm{ggT}(a - b, b). \qquad (*)$$

Falls nämlich $c \mid a$ und $c \mid b$, dann $c \mid a - b$. Umgekehrt gilt: falls $c \mid a - b$ und $c \mid b$, dann $c \mid a$. Also haben $\{a, b\}$ und $\{a - b, b\}$ die selben gemeinsamen Teiler, insbesondere gilt $\mathrm{ggT}(a, b) = \mathrm{ggT}(a - b, b)$.

Seien nun $a, b \geq 1$, dann gilt $\mathrm{ggT}(a, b) = \mathrm{ggT}(a \bmod b, b)$. Falls nämlich $a < b$, dann ist ja $a \bmod b = a$ und $\mathrm{ggT}(a, b) = \mathrm{ggT}(a \bmod b, b)$ gilt trivialerweise. Falls $a \geq b$, dann erhalten wir $\mathrm{ggT}(a, b) = \mathrm{ggT}(a \bmod b, b)$ aus wiederholter Anwendung von (*).

Der euklidische Algorithmus induziert zwei endliche Folgen $(a_i)_{i=0}^n$ und $(b_i)_{i=0}^n$ mit $a_0 = a$, $b_0 = b$, $a_{i+1} = b_i$, $b_{i+1} = a_i \bmod b_i$, $b_n = 0$ und $a_i, b_i > 0$ für $i = 0, \ldots n - 1$. Einerseits gilt nun wegen obiger Beobachtung $\mathrm{ggT}(a_{n-1}, b_{n-1}) = \mathrm{ggT}(a, b)$ und andererseits gilt $b_{n-1} \mid a_{n-1}$ da ja $b_n = 0$. Damit ist also der Rückgabewert $a_n = b_{n-1} = \mathrm{ggT}(a, b)$. □

Ein und dasselbe (Berechnungs-)problem erlaubt oft unterschiedliche Lösungsalgorithmen. Beispielsweise kann der größte gemeinsame Teiler zweier Zahlen auch durch den folgenden, auf erschöpfender Suche (engl. *brute force*) basierenden, Algorithmus berechnet werden.

Algorithmus 2 Suche nach ggT

```
Prozedur GGT-Suche(a, b)
    Für i = min{a, b}, ..., 1
        Falls i dividiert a und i dividiert b dann
            Antworte i
        Ende Falls
    Ende Für
Ende Prozedur
```

Unterschiedliche Algorithmen für das selbe Problem können unterschiedliche Vor- und Nachteile haben. So ist die Korrektheit des obigen Algorithmus, anders als beim euklidischen, trivial. Allerdings wird man wohl erwarten, dass der euklidische Algorithmus effizienter ist. Bevor wir genauer darauf eingehen, wie die Effizienz von Algorithmen beurteilt werden kann, wollen wir noch ein weiteres Beispiel betrachten.

Oft operieren Algorithmen auf umfangreicheren Daten als etwa zwei ganzen Zahlen. Ein wichtiger Aspekt ist dann die Frage in welcher Form bzw. Struktur die Daten gespeichert werden. Man spricht in diesem Zusammenhang auch von *Datenstrukturen*. Eine der einfachsten und gleichzeitig wichtigsten Datenstrukturen ist das *Datenfeld* (engl. *array*). Mathematisch handelt es sich dabei um eine endliche Folge. Die Elemente der Folge werden in aufsteigender Reihenfolge im Speicher abgelegt. Wir schreiben $A, B, \ldots$ für Datenfelder, $A[i]$ für das i-te Element, der kleinste Index eines Datenfelds ist 1, wir schreiben $A.\mathit{Länge}$ für die Länge des Datenfelds (d.h. für die Anzahl von Elementen, die es enthält). Die Notation $A.\mathit{Länge}$ wird in der Informatik verwendet, um das Attribut *Länge* des Objekts A zu notieren. Manche Attribute (wie dieses)

können nur gelesen werden (engl. *read-only*), manche können auch geschrieben werden, d.h. in Zuweisungen verwendet werden. Was davon der Fall ist wird üblicherweise aus dem Kontext heraus klar sein.

Eines der am häufigsten auftretenden Berechnungsprobleme ist das Sortieren eines Datenfelds. Folglich sind Sortieralgorithmen sehr gründlich untersucht worden. Wir wollen hier einen ersten einfachen Sortieralgorithmus betrachten: Einfügesortieren (engl. *insertion sort*). Die Grundidee besteht darin, ein Datenfeld zu unterteilen in einen sortierten Bereich (links) und einen noch nicht sortierten Bereich (rechts). Der sortierte Bereich wird dann sukzessive vergrößert, indem erste Element des unsortierten Bereichs in den sortierten Bereich (an der richtigen Stelle) eingefügt wird. Am Ende ist der unsortierte Bereich leer und das Datenfeld damit vollständig sortiert. Die Idee kann wie folgt als Pseudocode notiert werden:

Algorithmus 3 Einfügesortieren

```
1: Prozedur EINFÜGESORTIEREN(A)
2:     Für j := 2, ..., A.Länge
3:         x := A[j]
4:         i := j − 1
5:         Solange i ≥ 1 und A[i] > x
6:             A[i + 1] := A[i]
7:             i := i − 1
8:         Ende Solange
9:         A[i + 1] := x
10:    Ende Für
11: Ende Prozedur
```

Beispiel 1.2. siehe `bsp.einfuegesortieren.pdf`.

Für den Beweis der Korrektheit von Einfügesortieren wollen wir kurz einige Begriffe besprechen, die für Korrektheitsbeweise von Algorithmen von Bedeutung sind: Eine *Vorbedingung* eines Algorithmus ist eine Bedingung von der wir annehmen, dass sie beim Start des Algorithmus erfüllt ist. Für Einfügesortieren ist die Vorbedingung leer: A kann ein beliebiges Datenfeld sein. Eine *Nachbedingung* ist eine Aussage von der wir zeigen wollen, dass sie nach Ausführung des Algorithmus erfüllt ist falls die Eingabe die Vorbedingung erfüllt hat. Im Fall von Einfügesortieren ist die Nachbedingung: "A ist aufsteigend sortiert". Typischerweise wird die Korrektheit einer Schleife mit einem Induktionsbeweis behandelt. Eine *Schleifeninvariante* ist eine Aussage, die 1. unmittelbar vor Beginn des ersten Durchlaufs der Schleife wahr ist und 2. von der Schleife erhalten wird (d.h. wenn sie vor einem Durchlauf wahr ist, dann ist sie auch nach dem Durchlauf wahr). Damit ist eine Schleifeninvariante auch nach dem letzten Durchlauf der Schleife wahr.

Satz 1.2. *Einfügesortieren ist korrekt.*

Beweis. Sei A_0 das Eingabedatenfeld und $n = A.Länge$. Eine geeignete Invariante der äußeren Schleife ist (*): $A[1, \ldots, j-1]$ enthält eine sortiere Permutation von $A_0[1, \ldots, j-1]$ und $A[j, \ldots, n] = A_0[j, \ldots, n]$. Die Bedingung (*) ist unmittelbar vor dem ersten Durchlauf wahr, da $j = 2$ und ein ein-elementiges Datenfeld immer sortiert ist. Die Bedingung (*) wird durch die Schleife erhalten, da die Elemente von $A[j, \ldots, n]$ nicht verändert werden und der Schleifenkörper $x = A[j]$ in den sortierten Bereich $A[1, \ldots, j-1]$ an korrekter Stelle einsortiert und so den sortierten Bereich auf $A[1, \ldots, j]$ erweitert. Nach dem letzten Schleifendurchlauf ist $j = n+1$ und damit folgt aus der Invariante dass $A[1, \ldots, n]$ sortiert ist. □

1.2 Aufwandsabschätzung

Der wichtigste Aspekt eines Algorithmus, neben seiner Korrektheit, ist typischerweise sein Ressourcenverbrauch, und hier vor allem: seine Laufzeit. Auch der Verbrauch anderer Ressourcen, zum Beispiel Speicherplatz, kann von Bedeutung sein, zentral ist aber in den allermeisten Situationen die Frage wie viel Zeit ein Algorithmus benötigt. Eine erste Antwort auf diese Frage bestünde darin, eine konkrete Implementierung des Algorithmus in einer bestimmten Programmiersprache anzufertigen, diese auf einem bestimmten Computer auf verschiedene Eingaben anzuwenden und die verbrauchte Zeit zu protokollieren. Derartige empirische Studien haben in der Informatik durchaus ihren Nutzen, allerdings haben sie die folgenden Nachteile:

1. ist es unmöglich alle, typischerweise unendlich vielen, Eingaben auszuprobieren und
2. hängen die Ergebnisse von der Implementierung, der Programmiersprache und vom Computer ab.

Eine wichtige Beobachtung ist in diesem Zusammenhang, dass in der Praxis typischerweise der Ressourcenverbrauch (oft sogar: monoton) mit der Größe der Eingabe wächst. Anstatt also die Laufzeit als eine Funktion der Menge erlaubter Eingaben zu betrachten, will man sie als eine Funktion der Eingabegröße betrachten. Jetzt gibt es natürlich für eine feste Eingabegröße womöglich solche Eingaben, die mehr Zeit benötigen und solche die weniger Zeit benötigen. Man unterscheidet deshalb zwischen dem Zeitbedarf im besten Fall (engl. *best case*) und dem im schlechtesten Fall (engl. *worst case*). Zur Illustration werden wir jetzt eine Analyse der Komplexität von Einfügesortieren im besten sowie im schlechtesten Fall durchführen. Um die Analyse von Algorithmen zu vereinfachen und von Details der Implementierung und der Hardware zu abstrahieren (sh. oben) trifft man üblicherweise die folgenden Annahmen:

1. Die Instruktionen werden sequentiell ausgeführt (was auf Mehrprozessor-Systemen nicht immer der Fall sein muss).
2. Der Zeitbedarf jeder Zeile ist konstant, also insbesondere ist er unabhängig von den Werten der Programmvariablen und wird nicht beeinflusst von Speicherhierachiekonzepten wie z.B. caching.
3. Wir arbeiten mit unbeschränkten Datentypen, also z.B. den natürlichen Zahlen und nicht, wie das in konkreten Implementierung typischerweise der Fall ist mit, z.B., $\{0, \ldots, 2^{64}-1\}$.

Für $i = 2, \ldots, 10$ sei c_i die Zeit, die zur Ausführung von Zeile i benötigt wird. Wir nehmen an dass für die Ende-Statements $c_8 = c_{10} = 0$ gilt. Sei A das Eingabedatenfeld und n die Länge von A. Dann belaufen sich die Kosten der Anwendung von Einfügesortieren auf das Datenfeld A auf

$$T(A) = c_2 n + (c_3 + c_4 + c_9)(n-1) + c_5 \sum_{j=2}^{n} t_j + (c_6 + c_7) \sum_{j=2}^{n} (t_j - 1)$$

wobei t_j angibt, wie oft die Bedingung der inneren Schleife überprüft wird, wenn der äußere Schleifenzähler j ist. Die t_j hängen offensichtlich von A ab. Was sind nun mögliche Werte für die t_j ?

Im besten Fall ist das Eingabedatenfeld A bereits sortiert. Dann gilt nämlich $t_j = 1$ für $j = 2, \ldots, n$ und wir erhalten

$$T_{\mathrm{b}}(n) = c_2 n + (c_3 + c_4 + c_5 + c_9)(n-1),$$

d.h. also $T_{\mathrm{b}}(n) = kn + l$ für geeignete Konstanten k und l. Mit anderen Worten: die Laufzeit hängt linear von der Größe der Eingabe ab.

Im schlechtesten Fall muss jedes $A[j]$ mit *allen* Elementen in $A[1], \ldots, A[j-1]$ verglichen werden. Das geschieht dann wenn das Eingabedatenfeld absteigend sortiert ist. Dann ist also $t_j = j$ für $j = 2, \ldots, n$ und wir erhalten $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$ sowie $\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$ aus der gaußschen Summenformel und damit

$$T_{\mathrm{s}}(n) = (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 + (c_2 + c_3 + c_4 + c_9 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2})n - c_3 - c_4 - c_5 - c_9,$$

d.h. also $T_{\mathrm{s}}(n) = kn^2 + ln + m$ für geeignete Konstanten k, l und m. Im schlechtesten Fall hängt die Laufzeit also quadratisch von der Größe der Eingabe ab.

Wir sehen also, dass es einen deutlichen Unterschied zwischen dem besten Fall und dem schlechtesten Fall geben kann. Motiviert u.a. dadurch betrachtet man oft auch die Laufzeit im durchschnittlichen Fall (engl. *average case*). Hier eröffnet sich allerdings ein neues Problem: es ist a priori nicht klar, welche Wahrscheinlichkeitsverteilung den Eingabedaten zugrunde liegt. Diese kann auch je nach Anwendung recht unterschiedlich ausfallen. Der Einfachheit halber arbeitet man oft mit einer (in geeignetem Sinn) uniformen Wahrscheinlichkeitsverteilung.

Wir gehen für die Analyse der mittleren Laufzeit von Einfügesortieren von der folgenden Wahrscheinlichkeitsverteilung aus: Gegeben das Eingabedatenfeld $A[1], \ldots, A[n]$ und seine Sortierung $A[\pi(1)], \ldots, A[\pi(n)]$ nehmen wir an dass jede Permutation $\pi \in S_n$ gleich wahrscheinlich ist. D.h. jedes $\pi \in S_n$ tritt mit Wahrscheinlichkeit $\frac{1}{n!}$ auf. Wir verwenden die folgende Eigenschaft einer zufällig gewählten $\pi \in S_n$: Sei $1 \leq i \leq j \leq n$, dann ist

$$\mathrm{W}(\pi(j) \text{ ist } i\text{-t-größte Element in } \{\pi(1), \ldots, \pi(j)\}) = \frac{1}{j}.$$

Diese Aussage werden wir im nächsten Kapitel beweisen. Die mittlere Anzahl von Aufrufen des inneren Schleifenkopfs ist dann also

$$\bar{t}_j = \frac{1}{j}1 + \frac{1}{j}2 + \cdots \frac{1}{j}j = \frac{1}{j}\frac{j(j+1)}{2} = \frac{j+1}{2}.$$

Damit erhalten wir

$$\sum_{j=2}^{n} \bar{t}_j = \frac{1}{2}\sum_{j=3}^{n+1} j = \frac{(n+1)(n+2)}{4} - \frac{3}{2} = \frac{n^2+3n}{4} - 1,$$

$$\sum_{j=2}^{n}(\bar{t}_j - 1) = \frac{1}{2}\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{4} \text{ und}$$

$$T_{\mathrm{d}}(n) = (\frac{c_5}{4} + \frac{c_6}{4} + \frac{c_7}{4})n^2 + (c_2 + c_3 + c_4 + c_9 + \frac{3c_5}{4} - \frac{c_6}{4} - \frac{c_7}{4})n - c_3 - c_4 - c_9 - c_5,$$

d.h. also $T_{\mathrm{d}}(n) = kn^2 + ln + m$ für geeignete Konstanten k, l und m.

Mit dieser Analyse von Einfügesortieren haben wir also in einem gewissen Sinn eine Darstellung der Laufzeit auf *allen* Eingaben erhalten, womit wir fürs Erste das oben angesprochene erste Problem als behandelt betrachten wollen. Was das zweite Problem angeht bleibt uns noch die Abstraktion von den konkreten Konstanten c_i. Das wird durch Angabe der Laufzeit mit den Landau-Symbolen erreicht, auch "Groß-O-Notation" genannt ($O(f)$ steht für "Ordnung von f").

Definition 1.4. Sei $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ eine Funktion. Wir definieren

$$O(f) = \{g : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \exists c > 0, n_0 \in \mathbb{N} \text{ so dass } \forall n \geq n_0 \colon g(n) \leq c \cdot f(n)\}.$$

Falls $g \in O(f)$ sagen wir dass f eine *asymptotisch obere Schranke* von g ist. Sei weiters

$$\Omega(f) = \{g : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \exists c > 0, n_0 \in \mathbb{N} \text{ so dass } \forall n \geq n_0 \colon c \cdot f(n) \leq g(n)\}.$$

Falls $g \in \Omega(f)$ sagen wir dass f eine *asymptotisch untere Schranke* von g ist. Schließlich definieren wir noch

$$\Theta(f) = \{g : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \exists c > 0, d > 0, n_0 \in \mathbb{N} \text{ so dass } \forall n \geq n_0 \colon c \cdot f(n) \leq g(n) \leq d \cdot f(n)\}.$$

Falls $g \in \Theta(f)$ sagen wir dass f eine *asymptotisch scharfe Schranke* von g ist.

Beispiel 1.3. Seien $c, d, e > 0$, dann ist $cn^2+dn+e = \Theta(n^2)$. Einerseits ist nämlich $cn^2+dn+e \leq (c+d+e)n^2$ für $n \geq 1$. Andererseits ist auch $cn^2 \leq cn^2 + dn + e$ für $n \geq 0$.

Oft wird diese Notation sinngemäß auch für $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ oder Funktionen anderen Typs verwendet. Deshalb wird in der Literatur häufig der Typ von f gar nicht erst angegeben. Zu dieser Definition lassen sich unmittelbar die folgenden Beobachtungen machen.

Lemma 1.1. *Für alle Funktionen f, g gilt:*

1. $\Theta(f) = O(f) \cap \Omega(f)$
2. $g \in O(f)$ *genau dann wenn* $f \in \Omega(g)$
3. $g \in O(f)$ *genau dann wenn* $\limsup_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
4. $g \in \Omega(f)$ *genau dann wenn* $\liminf_{n\to\infty} \frac{g(n)}{f(n)} > 0$

wobei wir voraussetzen dass f und g nur endlich viele Nullstellen haben.

Beweis. 1. folgt direkt aus der Definition. Für 2. sei $c > 0$, $n_0 \in \mathbb{N}$ so dass $\forall n \geq n_0$: $g(n) \leq c \cdot f(n)$. Sei $d = \frac{1}{c}$, dann gilt $\forall n \geq n_0$: $d \cdot g(n) \leq f(n)$, d.h. also $f \in \Omega(g)$. Für die Gegenrichtung setzen wir $c = \frac{1}{d}$.

Für 3. sei wieder $c > 0, n_0 \in \mathbb{N}$ so dass $\forall n \geq n_0$: $g(n) \leq c \cdot f(n)$, d.h. $\frac{g(n)}{f(n)} \leq c$. Also ist $\{\frac{g(n)}{f(n)} \mid n \geq n_0\}$ im kompakten Intervall $[0, c]$ enthalten, besitzt also einen größten Häufungspunkt in $[0, c]$. Für die Gegenrichtung sei $n_0 - 1$ die größte Nullstelle von f. Dann ist $\left(\frac{g(n)}{f(n)}\right)_{n\geq n_0}$ beschränkt: falls $\left(\frac{g(n)}{f(n)}\right)_{n\geq n_0}$ nämlich unbeschränkt wäre, dann wäre $\limsup_{n\to\infty} \frac{g(n)}{f(n)} = \infty$. Also $\exists c > 0$ so dass $\frac{g(n)}{f(n)} \leq c$, d.h. $g(n) \leq c \cdot f(n)$ für $n \geq n_0$. 4. folgt aus 2. und 3. □

Oft schreiben wir $g(n) = O(f(n))$ statt $g \in O(f)$ um das Rechnen mit Termen wie z.B. $n^2+O(n)$ zu ermöglichen.

Satz 1.3. *Sei $f(n) = \sum_{i=0}^{k} a_i n^i$, $a_k > 0$, dann $f(n) = \Theta(n^k)$.*

Beweis. Man beachte dass

$$\frac{\sum_{i=0}^{k} a_i n^i}{n^k} = a_k + \frac{a_{k-1}}{n} + \frac{a_{k-2}}{n^2} + \cdots + \frac{a_0}{n^k} \longrightarrow a_k \text{ für } n \to \infty.$$

Daraus folgt mit Lemma 1.1 das Resultat. □

Wir sagen dass eine Funktion f von polynomialem Wachstum ist falls ein $k \geq 1$ existiert so dass $f(n) = O(n^k)$.

Wir können nun die Laufzeit von Einfügesortieren im besten Fall angeben als $\Theta(n)$, jene im schlechtesten Fall als $\Theta(n^2)$ und jene im (uniform verteilten) Durchschnittsfall ebenfalls als $\Theta(n^2)$. Oft werden wir uns bei Angabe der Laufzeit im schlechtesten Fall auf die obere Schranke, d.h. $O(\cdot)$ beschränken und umgekehrt bei der Laufzeit im besten Fall auf die untere Schranke $\Omega(\cdot)$. Wenn wir also sagen dass der Algorithmus $\mathcal{A}$ Laufzeit $O(f)$ hat ist damit gemeint, dass er im schlechtesten Fall Laufzeit $O(f)$ hat und analog für Ω und den besten Fall.

Kommen wir nun wieder zurück zur Bestimmung des ggT zweier positiver ganzer Zahlen a und b. Für die Analyse wollen wir davon ausgehen, dass a und b n-Bit Zahlen sind, d.h. $a, b \leq 2^n - 1$. Dann ist die Laufzeit der erschöpfenden ggT-Suche im schlechtesten Fall, d.h. für ähnlich große a und b, $O(2^n)$. Für den euklidischen Algorithmus überlegen wir uns zunächst:

Lemma 1.2. *Falls $a \geq b$ dann $a \bmod b < \frac{a}{2}$.*

Beweis. Falls $b \leq \frac{a}{2}$ dann $a \bmod b < b \leq \frac{a}{2}$ und wir sind fertig. Falls $b > \frac{a}{2}$, dann ist ja $a \geq b > \frac{a}{2}$ und damit $a \bmod b = a - b < \frac{a}{2}$. □

Also ist $a_{i+1} < \frac{a_i}{2}$ und damit gibt es höchstens $\log \max\{a, b\} + 1$, also $O(n)$ viele Schleifendurchläufe[1]. Wir sehen dass der euklidische Algorithmus polynomial, sogar linear, ist während die erschöpfende Suche exponentiell ist.

Eng in Zusammenhang mit dieser asymptotischen Notation steht auch die Folgende:

Definition 1.5. Sei $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$ eine Funktion. Wir definieren

$$o(f) = \{g : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \forall c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 \colon g(n) \leq c \cdot f(n)\}.$$

Falls $g \in o(f)$ sagen wir dass g *asymptotisch kleiner als* f ist.

$$\omega(f) = \{g : \mathbb{N} \to \mathbb{R}_{\geq 0} \mid \forall c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 \colon c \cdot f(n) \leq g(n)\}.$$

Falls $g \in \omega(f)$ sagen wir dass g *asymptotisch größer als* f ist.

Lemma 1.3. *Für alle Funktionen f, g gilt:*

1. $f \notin o(f)$
2. $f \notin \omega(f)$
3. $g \in o(f)$ *genau dann wenn* $f \in \omega(g)$
4. $o(f) \subseteq O(f)$
5. $\omega(f) \subseteq \Omega(f)$
6. $o(f) \cap \omega(f) = \emptyset$
7. $g \in o(f)$ *genau dann wenn* $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$
8. $g \in \omega(f)$ *genau dann wenn* $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$

[1] In dieser Vorlesung werden wir mit log immer den Logarithmus zur Basis 2 notieren

wobei wir wieder annehmen, dass f und g nur endlich viele Nullstellen haben.

Beweis. Für 1. setzen wir $c = \frac{1}{2}$ und für 2. setzen wir $c = 2$. 3. kann analog zu Lemma 1.1 gelöst werden indem c auf $\frac{1}{d}$ und d auf $\frac{1}{c}$ gesetzt wird. 4. und 5. folgen unmittelbar aus der Definition. 6. folgt durch Wahl geeigneter Konstanten ebenfalls direkt aus der Definition. Für 7. beachte man, dass die Definition von $g \in o(f)$ geschrieben werden kann als $\forall \varepsilon > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 \colon \frac{g(n)}{f(n)} \leq \varepsilon$. Für 8. schreibt man die Definition von $g \in \omega(f)$ als $\forall \varepsilon > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 \colon \frac{f(n)}{g(n)} \leq \varepsilon$, d.h. $\frac{g(n)}{f(n)} \geq \frac{1}{\varepsilon}$. □

Definition 1.6. Für $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ schreiben wir $f \sim g$ genau dann wenn $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$. Falls $f \sim g$ sagen wir dass f und g *asymptotisch äquivalent sind.*

Kapitel 2

Elementare Kombinatorik

Abzählprobleme sind klassische kombinatorische Fragestellungen. Bei einem Abzählproblem fragt man sich wie viele Elemente eine bestimmte endliche Menge hat. Dabei finden, für endliche Mengen A und B oft die folgenden Überlegungen Anwendung:

1. Summenregel: Falls $A \cap B = \emptyset$, dann $|A \cup B| = |A| + |B|$
2. Produktregel: $|A \times B| = |A| \cdot |B|$
3. Gleichheitsregel: Falls eine Bijektion $f : A \to B$ existiert, dann ist $|A| = |B|$.

Für eine endliche Menge A wird eine bijektive Abbildung $\pi : A \to A$ auch als *Permutation* bezeichnet. Zur Erleichterung der Notation, geht man oft davon aus, dass $A = \{1, \ldots, n\}$. Solche Permutationen können in einer *zweizeiligen Darstellung* als

$$\pi = \begin{pmatrix} 1 & 2 & \cdots & n \\ \pi(1) & \pi(2) & \cdots & \pi(n) \end{pmatrix}$$

geschrieben werden. Eine alternative Darstellung ist die *Zyklendarstellung* als

$$\pi = (a_1\ \pi(a_1)\ \cdots\ \pi^{l_1-1}(a_1))(a_2\ \pi(a_2)\ \cdots\ \pi^{l_2-1}(a_2)) \cdots (a_k\ \pi(a_k)\ \cdots\ \pi^{l_k-1}(a_k))$$

wobei l_i definiert ist als das kleinste l so dass $\pi^l(a_i) = a_i$ und a_i so aus $\{1, \ldots, n\}$ gewählt wird, dass es in keinem der vorherigen Zyklen auftritt. Damit kommt also jedes $a \in \{1, \ldots, n\}$ genau ein Mal in der Zyklendarstellung vor.

Beispiel 2.1. Die Permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 2 & 5 & 1 & 3 & 4 \end{pmatrix}$$

wird in Zyklendarstellung als $(164)(2)(35)$ geschrieben.

Die Anzahl von Permutation von $\{1, \ldots, n\}$ ist $n! = n \cdot (n-1) \cdot (n-2) \cdots 1$. Die Funktion $n \mapsto n!$ kann auch rekursiv definiert werden durch $0! = 1$ und $(n+1)! = (n+1)n!$. Die Menge aller Permutationen von $\{1, \ldots, n\}$ wird auch als S_n bezeichnet. Man kann sich leicht davon überzeugen, dass S_n mit der Komposition von Funktionen eine Gruppe bildet.

Eine *Variation ohne Wiederholung* besteht aus der k-fachen Auswahl eines von n Objekten wobei das Objekt nach seiner Auswahl nicht mehr für spätere Auswahlen zur Verfügung steht. Die Anzahl der Variationen ohne Wiederholung ist $n \cdot n - 1 \cdot \cdots \cdot (n-k+1) = \frac{n!}{(n-k)!}$

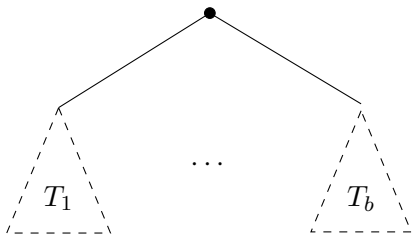
Beispiel 2.2. Für die ersten k Stellen einer Permutation $\pi \in S_n$ werden k aus n Objekten ohne Wiederholung ausgewählt, es gibt also $\frac{n!}{(n-k)!}$ Möglichkeiten. Für $k = n$ ergibt sich dann wie gehabt $\frac{n!}{(n-n)!} = n!$.

Eine *Variation mit Wiederholung* besteht aus der k-fachen Auswahl eines von n Objekten wobei das Objekt nach seiner Auswahl für spätere Auswahlen zur Verfügung steht. Die Anzahl der Variationen mit Wiederholung ist $n \cdot n \cdots n$, d.h. n^k.

Beispiel 2.3. Wenn ein Münzwurf entweder Kopf oder Zahl ergibt und eine Münze k mal hintereinander geworfen wird, gibt es insgesamt 2^k verschiedene Versuchsausgänge. Die Menge der Versuchsausgänge, d.h., der k-fachen Wiederholung von "Kopf" oder "Zahl" steht in Bijektion zu der Menge der Zeichenketten der Länge k die nur aus 0 und 1 bestehen. Diese wiederum steht in Bijektion zu den Teilmengen einer k-elementigen Menge. Somit gibt es auch davon jeweils 2^k.

Beispiel 2.4. Sei $b \geq 2$. Ein *vollständiger Baum mit Arität* b und Tiefe 0 ist ein einzelner Knoten

Ein solcher Knoten heißt *Blatt*. Ein vollständiger Baum mit Arität b und Tiefe $d+1$ ist ein Graph[1] der Form



wobei $T_1, \ldots, T_b$ vollständige Bäume mit Arität b und Tiefe d sind. Jeder Pfad in einem vollständigen Baum kann identifiziert werden mit einem Tupel $(p_1, \ldots, p_d) \in \{1, \ldots, b\}^d$. Also hat ein vollständiger Baum b^d Blätter sowie

$$b^0 + b^1 + \cdots + b^d = \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1}$$

Knoten. Ein vollständiger Binärbaum der Tiefe 3 hat also $2^3 = 8$ Blätter und $2^4 - 1 = 15$ Knoten.

Eine *Kombination ohne Wiederholung* besteht aus der k-fachen Auswahl eines von n Objekten wobei das Objekt nach seiner Auswahl nicht mehr für spätere Auswahlen zur Verfügung steht und die Reihenfolge der Wahl der Objekte irrelevant ist. Wir wählen also eine k-elementige Teilmenge einer n-elementigen Menge. Die Anzahl der Kombinationen ohne Wiederholung ist $\frac{n!}{(n-k)!k!} = \binom{n}{k}$, sie ergibt sich durch die Anzahl $\frac{n!}{(n-k)!}$ der Variationen ohne Wiederholung und der Überlegung, dass jeder Kombination $k!$ Variationen entsprechen.

Beispiel 2.5. Bei der Multiplikation von $(x+y)\cdots(x+y) = (x+y)^n$ müssen zur Bestimmung des Koeffizienten von $x^k y^{n-k}$ alle Möglichkeiten in Betracht gezogen werden in n Faktoren k mal x zu wählen und die anderen $n-k$ Mal y. Es gibt $\binom{n}{k}$ solche Möglichkeiten, also ergibt sich der binomische Lehrsatz

$$(x+y)^n = \sum_{k=0}^{n} \binom{n}{k} x^k y^{n-k}.$$

[1] Ein Graph ist ein Paar $G = (V, E)$ wobei V die Menge von *Knoten* ist und $E \subseteq V \times V$ die Menge von *Kanten*.

Das erklärt auch warum $\binom{n}{k}$ als *Binomialkoeffizient* bezeichnet wird.

Bei der Lottoziehung "6 aus 45" werden aus 45 Kugeln 6 Stück gezogen wobei die Reihenfolge für die Ermittlung des Gewinners egal ist. Es gibt also $\binom{45}{6} = 8145060$ Möglichkeiten für den Ausgang der Ziehung.

Eine *Kombination mit Wiederholung* besteht aus der k-fachen Auswahl eines von n Objekten wobei das Objekt nach seiner Auswahl für spätere Auswahlen zur Verfügung steht und die Reihenfolge der Wahl der Objekte irrelevant ist. Wir wählen also eine k-elementige Multimenge[2] mit Träger $\{1, \dots, n\}$. Eine solche Multimenge kann geschrieben werden als Tupel $(a_1, \dots, a_k)$ wobei $1 \leq a_1 \leq \cdots \leq a_k \leq n$. Nun gibt es eine Bijektion von der Menge der k-elementigen Multimengen mit Träger $\{1, \dots, n\}$ auf die k-elementigen Teilmengen von $\{1, \dots, n+k-1\}$, die durch

$$(a_1, \dots, a_k) \mapsto (a_1, a_2 + 1, \dots, a_k + k - 1)$$

gegeben ist. Dann ist nämlich $1 \leq a_1 < a_2 + 1 < \cdots < a_k + k - 1 \leq n + k - 1$. Die Anzahl k-elementiger Teilmengen von $\{1, \dots, n+k-1\}$ kennen wir bereits: $\binom{n+k-1}{k}$.

Beispiel 2.6. Bei einem Brettspiel würfelt ein Spieler mit zwei Würfeln gleichzeitig. Bei den möglichen Würfen handelt es sich um eine Kombination mit Wiederholung mit $n = 6$ und $k = 2$. Es gibt also $\binom{7}{2} = 21$ verschiedene Würfe.

Lemma 2.1. *Sei $1 \leq i \leq j \leq n$, sei $\pi \in S_n$ uniform gewählt, dann ist*

$$\mathrm{W}(\pi(j) \textit{ ist } i\textit{-t-größtes Element in } \{\pi(1), \dots, \pi(j)\}) = \frac{1}{j}.$$

Beweis. 1. Die Anzahl der Tupel $(\pi(1), \dots, \pi(j))$ ist $\frac{n!}{(n-j)!}$. 2. Die Anzahl der Tupel $(\pi(1), \dots, \pi(j))$ in denen $\pi(j)$ am i-t-größten ist ergibt sich a) durch Auswahl der Menge $\{\pi(1), \dots, \pi(j)\} \subseteq \{1, \dots, n\}$, dafür gibt es $\binom{n}{j} = \frac{n!}{(n-j)!j!}$ Möglichkeiten, b) durch Fixierung von $\pi(j)$ als das i-t-größte Element und c) durch Auswahl einer Reihenfolge für $\{\pi(1), \dots, \pi(j-1)\}$, dafür gibt es $(j-1)!$ Möglichkeiten, d.h. also insgesamt $\frac{n!(j-1)!}{(n-j)!j!} = \frac{n!}{(n-j)!j}$. Wir erhalten also

$$\frac{\text{günstige}}{\text{mögliche}} = \frac{n! \cdot (n-j)!}{(n-j)! \cdot j \cdot n!} = \frac{1}{j}.$$

□

Für zwei endliche Mengen A und B mit $A \cap B = \emptyset$ gilt wie erwähnt $|A \cup B| = |A| + |B|$. Falls $A \cap B \neq \emptyset$ werden durch $|A|+|B|$ die Element im Durchschnitt doppelt gezählt. Durch Korrektur dieser Doppelzählung erhalten wir $|A \cup B| = |A|+|B|-|A \cap B|$. Im Fall von drei endlichen Mengen A, B und C werden durch $|A|+|B|+|C|$ alle Elemente die in zwei Mengen liegen zu oft gezählt. Eine erste Korrektur ergibt $|A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C|$. Nun werden aber die Elemente im Schnitt von drei Mengen gar nicht gezählt. Durch einen weiteren Korrekturschritt erhalten wir also $|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cup B \cup C|$. Im Allgemeinen gilt:

Satz 2.1 (Prinzip von Inklusion und Exklusion)**.** *Seien $A_1, \dots, A_n$ endliche Mengen, dann ist*

$$|\bigcup_{i=1}^{n} A_i| = \sum_{\emptyset \neq I \{1,\dots,n\}} (-1)^{|I|+1} |\bigcap_{i \in I} A_i|$$

[2]Sei X eine Menge. Eine Multimenge M mit Träger X ist gegeben durch ihre charakteristische Funktion $\chi_M : X \to \mathbb{N}$. Eine Multimenge kann also, anders als eine Menge, ein Element mehrfach enthalten. Man definiert $|M| = \sum_{x \in X} \chi_M(x)$.

Beweis. Sei $x \in \bigcup_{i=1}^{n} A_i$, $S = \{i \in \{1 \ldots, n\} \mid x \in A_i\}$ und $s = |S|$. Auf der rechten Seite wird x jetzt genau dann in einem Durchschnitt gezählt wenn $I \subseteq S$ und zwar, je nach Kardinalität von I entweder positiv oder negativ. Das Element x wird also

$$\binom{s}{1} - \binom{s}{2} + \binom{s}{3} - \cdots + (-1)^{s+1}\binom{s}{s} = \sum_{i=1}^{s} \binom{s}{i}(-1)^{i+1}$$

mal gezählt. Nun ist aber nach dem binomischen Lehrsatz $\sum_{i=0}^{s} \binom{s}{i}(-1)^i = (1-1)^s = 0$, sowie, nach Multiplikation mit -1, auch $\sum_{i=0}^{s} \binom{s}{i}(-1)^{i+1} = 0$. Weiters ist $\sum_{i=0}^{s} \binom{s}{i}(-1)^{i+1} = -1 + \sum_{i=1}^{s} \binom{s}{i}(-1)^{i+1}$. Damit wird x also $\sum_{i=1}^{s} \binom{s}{i}(-1)^{i+1} = 1$ mal gezählt. □

Das *Schubfachprinzip* (engl. *pigeonhole principle*) besagt Folgendes: Seien A und B endliche Mengen mit $|A| > |B|$, dann existiert keine injektive Funktion $f : A \to B$. Es kann mit einem einfachen Induktionsargument bewiesen werden. Ein Beispiel für seine Anwendung liefert der folgende Satz.

Satz 2.2. *Für jedes ungerade $q \in \mathbb{N}$ existiert ein $i \geq 1$ so dass $q \mid 2^i - 1$.*

Beweis. Wir kürzen $2^i - 1$ als a_i ab. Wir betrachten $a_1, \ldots, a_q \pmod q$. Falls es darunter ein a_i gibt mit $a_i \equiv 0 \pmod q$ dann sind wir fertig. Falls nicht, dann gibt es nach dem Schubfachprinzip $i < j$ mit $a_i \equiv a_j \pmod q$, d.h. $q \mid a_j - a_i$. Nun ist aber $a_j - a_i = 2^j - 1 - 2^i + 1 = 2^j - 2^i = 2^i(2^{j-i} - 1)$ und da q ungerade muss $q \mid a_{j-i}$, Widerspruch. □

Kapitel 3

Teile und Herrsche

In der Praxis auftretende Berechnungsprobleme haben oft die Eigenschaft dass ihre Instanzen zerlegt werden können und Lösungen der kleineren Instanzen zur Lösung der ursprünglichen Instanz hilfreich sind. Diese Vorgehensweise zur Lösung eines Berechnungsproblems ist so weit verbreitet, dass sich dafür ein eigener Begriff eingebürgert hat: die *teile-und-herrsche Strategie*. Algorithmen, die der teile-und-herrsche Strategie folgen bestehen üblicherweise aus drei Phasen:

1. *Aufteilung* der Eingabeinstanz in mehrere Instanzen kleiner Größe
2. *Lösung* der kleineren Instanzen durch rekursiven Aufruf des Algorithmus
3. *Kombination* der Lösungen der kleineren Instanzen zu einer Lösung der ursprünglichen Instanz

Die Basis dieser Rekursion wird normalerweise dadurch gebildet, dass Instanzen deren Größe eine Konstante ist trivial gelöst werden können.

3.1 Sortieren durch Verschmelzen

Das Sortierproblem kann durch einen teile-und-herrsche Algorithmus gelöst werden. Die Grundidee dazu ist, 1. das Eingabedatenfeld in zwei (zirka) gleich große Teile zu teilen, 2. jeden dieser beiden Teile unabhängig vom anderen durch einen rekursiven Aufruf zu sortieren, und 3. die beiden sortierten Datenfelder zu einem einzigen sortierten Datenfeld zu verschmelzen. Deshalb wird dieser Algorithmus auch als "Sortieren durch Verschmelzen" (engl. *merge sort*) bezeichnet. Von den beiden Schritten 1 und 2 sollte klar sein wie sie umgesetzt werden können. Der Ansatz zur Realisierung des 3. Schritts besteht darin zwei Indizes zu verwenden, jeweils einen für jedes der Eingabedatenfelder, und sie so durch die Eingabedatenfelder laufen zu lassen, dass das aktuelle Minimum immer unter einem der beiden steht. Dieses aktuelle Minimum wird in das Ausgabedatenfeld übertragen. In Algorithmus 4 ist diese Idee als Pseudocode ausformuliert.

Basierend auf der Prozedur Verschmelzen kann nun Sortieren durch Verschmelzen wie in Algorithmus 5 als Pseudocode formuliert werden. Für $i \leq j$ ist die Notation $A := B[i, \ldots, j]$ eine Abkürzung für die Herstellung eines Datenfeldes A dessen Inhalt genau den Einträgen i bis j des Datenfeldes B entspricht. Eine solche Kopie kann durch eine einfache Schleife in Laufzeit $\Theta(j - i)$ hergestellt werden.

Wir wollen nun die Laufzeit von Sortieren durch Verschmelzen analysieren. Die Prozedur "Verschmelzen" benötigt Laufzeit $\Theta(A.Länge + B.Länge)$. Sei $T(n)$ die Laufzeit von Sortieren durch

Algorithmus 4 Verschmelzen

Prozedur VERSCHMELZEN(A, B)
  Sei C ein neues Datenfeld der Länge $A.Länge + B.Länge$
  $i := 1$
  $j := 1$
  Für $k := 1, \ldots, A.Länge + B.Länge$
    Falls $j > B.Länge$ oder ($i \leq A.Länge$ und $A[i] \leq B[j]$) **dann**
      $C[k] := A[i]$
      $i := i + 1$
    sonst
      $C[k] := B[j]$
      $j := j + 1$
    Ende Falls
  Ende Für
  Antworte C
Ende Prozedur

Algorithmus 5 Sortieren durch Verschmelzen (engl. *merge sort*)

Prozedur VSORTIEREN(A)
  Falls $A.Länge = 1$ **dann**
    Antworte A
  sonst
    $m := \left\lceil \frac{A.Länge}{2} \right\rceil$
    $L := A[1, \ldots, m]$
    $R := A[m + 1, \ldots, A.Länge]$
    $L' :=$ VSORTIEREN(L)
    $R' :=$ VSORTIEREN(R)
    Antworte VERSCHMELZEN(L', R')
  Ende Falls
Ende Prozedur

Verschmelzen wenn das Eingabedatenfeld die Länge n hat. Klar ist bereits dass $T(1) = \Theta(1)$. Für $n \geq 2$ beobachten wir: Die Aufteilung in Teilprobleme benötigt Laufzeit $\Theta(n)$, das Lösen der Teilprobleme benötigt $T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$, das Verschmelzen der beiden Lösungen, die ja Größe m und $n-m$ haben, benötigt $\Theta(n)$. Insbesondere können wir hier beobachten, dass die Laufzeit von Sortieren durch Verschmelzen, anders als die von Einfügesortieren, nur von $n = A.Länge$ abhängt, nicht aber vom Inhalt von A. Somit ist die Laufzeit im besten Fall gleich der Laufzeit im schlechtesten Fall und gleich der Laufzeit im Durchschnittsfall.

Insgesamt erhalten wir für die Laufzeit

$$T(n) = \begin{cases} \Theta(1) & \text{für } n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{für } n \geq 2 \end{cases}.$$

Bei dieser Darstellung der Laufzeit handelt es sich um eine *Rekursionsgleichung*. Die Analyse eines teile-und-herrsche Algorithmus führt typischerweise auf eine Rekursionsgleichung einer solchen Form. In Kapitel 4 werden wir zeigen, dass für die obige Rekursionsgleichung $T(n) = \Theta(n \log n)$ gilt. Für den Augenblick wollen wir uns damit begnügen die folgende einfache Beobachtung zu machen, die den Kern der obigen Rekursionsgleichung enthält.

Satz 3.1. *Sei* $S : \{2^k \mid k \geq 1\} \to \mathbb{N}$ *definiert durch* $S(n) = \begin{cases} 2 & \text{falls } n = 2 \\ 2S(\frac{n}{2}) + n & \text{falls } n > 2 \end{cases}$, *dann ist* $S(n) = n \log n$.

Beweis. Zu zeigen ist also, für alle $k \geq 1$, dass $S(2^k) = k \cdot 2^k$. Für $k = 1$ ist das per definitionem erfüllt. Für $k \geq 2$ haben wir $S(2^k) = 2 \cdot S(2^{k-1}) + 2^k =^{\text{IH}} 2 \cdot (k-1) \cdot 2^{k-1} + 2^k = k \cdot 2^k$. □

3.2 Dichtestes Punktepaar

Wir betrachten jetzt ein erstes fundamentales geometrisches Berechnungsproblem. Für Punkte $p_1, p_2 \in \mathbb{R}^2$ sei $\mathrm{d}(p_1, p_2)$ die übliche euklidische Distanz, d.h.

$$\mathrm{d}(\begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

Wir wollen das folgende Problem lösen:

Dichtestes Punktepaar
Eingabe: Eine endliche Menge $P \subseteq \mathbb{R}^2$
Ausgabe: $p, q \in P$ so dass $p \neq q$ und $\mathrm{d}(p, q)$ minimal

Klar ist, dass ein auf erschöpfender Suche basierender Algorithmus existiert, der einfach alle Paare von Punkten ausprobiert, siehe Algorithmus 6. Es gibt $\binom{n}{2}$ Paare die alle in jedem Fall durchlaufen werden, damit hat dieser Algorithmus Laufzeit $\Theta(n^2)$.

Wir werden nun sehen, dass es möglich ist mit einem Algorithmus der dem teile-und-herrsche Prinzip folgt (selbst im schlechtesten Fall) eine Laufzeit von $O(n \log n)$ zu erreichen. Sei $P \subseteq \mathbb{R}^2$ eine endliche Menge von Punkten. Die Grundidee des Verfahrens ist wie folgt:

1. *Aufteilung*: Wir teilen das Problem entlang einer vertikalen Gerade, genauer: Sei $x_m \in \mathbb{R}$ und $P = Q \uplus R$ so dass $|Q| = \left\lceil \frac{|P|}{2} \right\rceil$, $|R| = \left\lfloor \frac{|P|}{2} \right\rfloor$, für alle $\begin{pmatrix} x \\ y \end{pmatrix} \in Q$: $x \leq x_m$ und für alle $\begin{pmatrix} x \\ y \end{pmatrix} \in R$: $x \geq x_m$. Man beachte, dass sowohl P als auch Q Punkte mit x-Koordinate x_m enthalten können.

Algorithmus 6 Erschöpfende Suche nach dichtestem Punktepaar

```
Prozedur DPP-Suche(P)
    d_min := ∞
    Für i := 1, ..., P.Länge
        Für j := i + 1, ..., P.Länge
            Falls d(P[i], P[j]) < d_min dann
                d_min := d(P[i], P[j])
                a := (P[i], P[j])
            Ende Falls
        Ende Für
    Ende Für
    Antworte a
Ende Prozedur
```

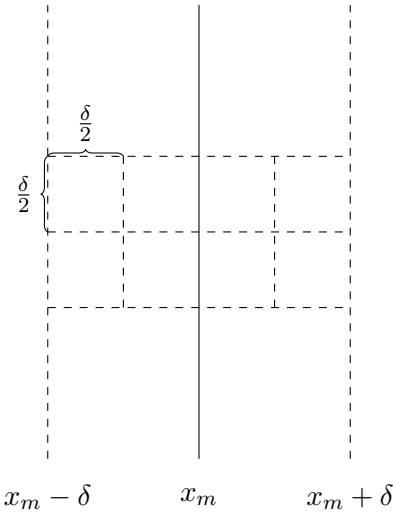


Abbildung 3.1: Kombination der Lösungen der Teilprobleme

2. *Lösung*: Mit Hilfe rekursiver Aufrufe bestimmen wir das dichteste Punktepaar $q_1, q_2 \in Q$ sowie das dichteste Punktepaar $r_1, r_2 \in R$. Sei $\delta = \min\{\mathrm{d}(q_1, q_2), \mathrm{d}(r_1, r_2)\}$.

3. *Kombination*: Das dichteste Punktepaar von P ist nun entweder q_1, q_2 oder r_1, r_2 oder ein Paar q, r wobei $q \in Q$ und $r \in R$. Natürlich können wir jetzt nicht alle (quadratisch vielen) Paare in $Q \times R$ durchsuchen wenn wir nur Laufzeit $O(n \log n)$ verwenden wollen. Mit einer kurzen Überlegung kann man aber zeigen, dass es ausreicht linear viele Paare zu überprüfen. Der Schlüssel dazu ist, dass wir δ bereits kennen. Falls nämlich ein Paar $q = \begin{pmatrix} q_x \\ q_y \end{pmatrix} \in Q$, $r = \begin{pmatrix} r_x \\ r_y \end{pmatrix} \in R$ das dichteste Punktepaar von P ist, muss nämlich $q_x, r_x \in [x_m - \delta, x_m + \delta]$ sein, d.h. dass q und r in einem Schlauch der Breite 2δ um x_m liegen. Weiters muss natürlich auch $|q_y - r_y| < \delta$ sein. Also liegen q und r in einem $2\delta \times \delta$ großem Rechteck das um die Gerade $x = x_m$ zentriert ist. Dieses Rechteck stellen wir uns nun als unterteilt in 8 Zellen der Größe $\frac{\delta}{2} \times \frac{\delta}{2}$ vor, siehe Abbildung 3.1. Jeder der Zellen (als Produkt geschlossener Intervalle) auf der linken Seite (von $x = x_m$) enthält höchstens einen Punkt von Q: würde eine Zelle nämlich zwei Punkte $q_1', q_2' \in Q$ enthalten, dann wäre $\mathrm{d}(q_1', q_2') \leq \sqrt{\left(\frac{\delta}{2}\right)^2 + \left(\frac{\delta}{2}\right)^2} = \frac{\delta}{\sqrt{2}} < \delta$ was ein Widerspruch ist.

Analog dazu enthält auch jede Zelle auf der rechten Seite höchstens einen Punkt von R. In diesem Rechteck gibt es also höchstens 8 Punkte von P. Um alle für das dichteste in Frage kommenden Punktepaare $q \in Q, r \in R$ zu überprüfen reicht es also, die Punkte im Schlauch rund um die Gerade $x = x_m$ nach y-Koordinate zu sortieren und für jeden Punkt den Abstand zu seinen 7 nächsten in der sortierten Liste zu überprüfen. Falls auf diese Weise ein Punktepaar $q \in Q, r \in R$ mit $\mathrm{d}(q, r) < \delta$ gefunden wird, so ist dieses das dichteste in P, falls nicht, dann ist das dichteste Punktepaar in P je nachdem entweder q_1, q_2 oder r_1, r_2. Damit haben wir uns also geometrisch davon überzeugt, dass diese Idee für einen teile-und-herrsche-Algorithmus sinnvoll ist und wir können uns an die konkrete Realisierung machen.

Ein Aspekt der durch die obige Diskussion noch nicht vollständig festgelegt ist, ist was passieren soll wenn mehrere Punkte in P auf der Gerade $x = x_m$ liegen. In solchen nicht-deterministischen Situationen ist es häufig nützlich, eine einfach zu berechnende Determinisierung festzulegen. Dazu definieren wir:

Definition 3.1. Die *lexikographische Ordnung* $<_{\text{lex}}$ auf $\mathbb{R}^2$ ist festgelegt durch:

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} <_{\text{lex}} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} \text{ genau dann wenn } 1.\ x_1 < x_2 \text{ oder}$$
$$2.\ x_1 = x_2 \text{ und } y_1 < y_2.$$

Wir schreiben $\leq_{\text{lex}}$ für die reflexive Hülle von $<_{\text{lex}}$ und beobachten, dass $<_{\text{lex}}$ eine totale Ordnung auf $\mathbb{R}^2$ ist. Wir können die Aufteilung des Problems für die Menge P nun also deterministisch spezifizieren als: Sei $p_m \in P$, $Q = \{p \in P \mid p \leq_{\text{lex}} p_m\}$, $R = \{p \in P \mid p >_{\text{lex}} p_m\}$ so dass $|Q| = \left\lceil \frac{|P|}{2} \right\rceil$ und $R = \left\lfloor \frac{|P|}{2} \right\rfloor$. Dann ist x_m die x-Koordinate von p_m. Falls also mehrere Punkte in P die x-Koordinate x_m haben gibt es auf der Gerade $x = x_m$ einen Punkt so dass alle darunter liegenden Punkte in Q landen und alle darüber liegenden in R.

Wir benötigen also eine Darstellung der Punkte sortiert nach $\leq_{\text{lex}}$ sowie eine in der die selben Punkte nach y-Koordinate sortiert sind. Das können wir erreichen, indem wir zwei Datenfelder verwenden die die selben Punkte enthalten: eines wird nach $\leq_{\text{lex}}$ sortiert und eines nach der y-Koordinate. Ein wichtiger Punkt ist nun dass wir es uns nicht leisten können, nach der Teilung erneut zu sortieren. Das würde in jedem Schritt Zeit $O(n \log n)$ verbrauchen und um eine Gesamtlaufzeit von $\Theta(n \log n)$ zu erreichen müssen wir in jedem Schritt mit Zeit $O(n)$ auskommen. Dieses Hindernis kann überwunden werden, indem wir diese beiden Sortierung einmal zu Beginn des Algorithmus (in Zeit $O(n \log n)$) berechnen und dann sicherstellen, dass die Sortierung durch den gesamten Algorithmus hindurch beibehalten wird.

Zur Realisierung dieses Ansatzes benötigen wir noch eine weitere Operation. Falls wir aus einer Menge X alle Elemente x auswählen wollen, die eine Eigenschaft $P(x)$ haben, dann schreiben wir diese Menge als $\{x \in X \mid P(x)\}$. Eine analoge Operation auf Datenfeldern kann wie folgt definiert werden: Sei A ein Datenfeld, dann ist $\langle x \in A \mid P(x) \rangle$ ein Datenfeld das alle Elemente von A enthält, die die Eigenschaft P erfüllen und zwar *in der Reihenfolge, in der sie in A vorkommen*. Algorithmus 7 führt diese Auswahl durch. Falls die Überprüfung der Eigenschaft P konstante Zeit erfordert (was üblicherweise bei uns der Fall sein wird), dann läuft dieser Algorithmus in Zeit $\Theta(n)$ wobei $n = A.\mathit{Länge}$.

Der gesamte Algorithmus zur Bestimmung des dichtesten Punktepaares kann als Pseudocode wie in Algorithmus 8 geschrieben werden.

Für die Laufzeitanalyse von DPP-TH sei $n = |P|$. Die Laufzeit von DPP-TH ist $\Theta(n \log n)$ für das Sortieren plus der Laufzeit von DPP-TH-Rek. Für die Laufzeitkomplexität von DPP-TH-

Algorithmus 7 Auswahl aus einem Datenfeld

Prozedur AUSWAHL$_P(A)$
  Sei B ein neues Datenfeld
  $j := 1$
  Für $i = 1, \ldots, A.Länge$
    Falls $P(A[i])$ **dann**
      $B[j] := A[i]$
      $j := j + 1$
    Ende Falls
  Ende Für
  Antworte B
Ende Prozedur

Rek erhalten wir

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 3 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{if } n > 3 \end{cases}$$

Diese Rekursionsgleichung ist beinahe identisch zu jener die wir aus Sortieren durch Verschmelzen erhalten haben. In der Tat gilt auch hier $T(n) = \Theta(n \log n)$ wie wir in Kapitel 4 zeigen werden. Somit ist auch die Laufzeit des Gesamtalgorithmus DPP-TH $\Theta(n \log n)$.

Wir sehen also, dass die Kombinationsphase in einem teile-und-herrsche-Algorithmus durchaus auch trickreicher sein kann als das beim Sortieren durch Verschmelzen der Fall ist.

3.3 Matrixmultiplikation

Wir werden annehmen, dass eine Matrix im Speicher abgelegt wird als ein Datenfeld, dessen Elemente Datenfelder einer festen Länge sind. Falls also A eine Matrix ist, so ist $A[i]$ die i-te Zeile von A und damit $A[i][j]$ das j-te Element der i-ten Zeile. Um die Notation etwas abzukürzen, schreiben wir stattdessen auch $A[i, j]$. Wir betrachten das folgende Problem:

Matrixmultiplikation
Eingabe: eine $n \times m$-Matrix A und eine $m \times l$-Matrix B
Ausgabe: $A \cdot B$

Seien $n, m, l \geq 1$ und $A = (a_{i,j})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq m}}$ und $B = (b_{i,j})_{\substack{1 \leq i \leq m \\ 1 \leq j \leq l}}$ Matrizen. Dann ist deren Produkt $A \cdot B = C = (c_{i,j})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq l}}$ bekannterweise durch

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

definiert. Diese Formel induziert sofort Algorithmus 9. Um die Analyse zu vereinfachen, werden wir von nun an annehmen, dass $n = m = l$ ist sowie dass n eine Zweierpotenz ist. Dann kann die Laufzeit des obigen Algorithmus aufgrund der drei verschachtelten Schleifen sofort als $\Theta(n^3)$ angegeben werden.

Für $n = 2n'$ kann eine $n \times n$-Matrix in vier $n' \times n'$-Matrizen geteilt werden.

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \quad B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \quad C = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Algorithmus 8 Teile-und-herrsche Algorithmus für dichtestes Punktepaar

Prozedur DPP-TH(P)
  $P := P$ aufsteigend nach $\leq_{\text{lex}}$ sortiert
  $P_y := P$ aufsteigend nach y-Koordinate sortiert
  Antworte DPP-TH-Rek(P, P_y)
Ende Prozedur

Prozedur DPP-TH-Rek(P, P_y)
  Falls $P.Länge \leq 3$ **dann**
    Antworte DPP-Suche(P)
  Ende Falls

  $m := \left\lceil \frac{P.Länge}{2} \right\rceil$
  $Q := P[1, \ldots, m]$
  $R := P[m+1, \ldots, P_x.Länge]$
  $Q_y := \langle p \in P_y \mid p \leq_{\text{lex}} P[m] \rangle$
  $R_y := \langle p \in P_y \mid p >_{\text{lex}} P[m] \rangle$

  $(q_1, q_2) :=$ DPP-TH-Rek(Q, Q_y)
  $(r_1, r_2) :=$ DPP-TH-Rek(R, R_y)
  $\delta := \min\{\mathrm{d}(q_1, q_2), \mathrm{d}(r_1, r_2)\}$

  $S_y := \langle p \in P_y \mid P[m].x - \delta \leq p.x \leq P[m].x + \delta \rangle$
  $s_1 := (0, 0)$
  $s_2 := (\infty, \infty)$
  Für $i := 1, \ldots, S_y.Länge - 1$
    Für $j := i+1, \ldots, \min\{i+7, S_y.Länge\}$
      Falls $\mathrm{d}(S_y[i], S_y[j]) < \mathrm{d}(s_1, s_2)$ **dann**
        $s_1 := S_y[i]$
        $s_2 := S_y[j]$
      Ende Falls
    Ende Für
  Ende Für

  Falls $\mathrm{d}(s_1, s_2) < \delta$ **dann**
    Antworte (s_1, s_2)
  sonst falls $\mathrm{d}(r_1, r_2) < \mathrm{d}(q_1, q_2)$ **dann**
    Antworte (r_1, r_2)
  sonst
    Antworte (q_1, q_2)
  Ende Falls
Ende Prozedur

Algorithmus 9 Matrixmultiplikation (direkt)

Vorbedingung: $n, m, l \geq 1$, A ist $n \times m$-Matrix, B ist $m \times l$-Matrix

```
Prozedur MATMULT(A, B)
    Sei C eine neue n × l-Matrix
    Für i := 1, ..., n
        Für j := 1, ..., l
            C[i, j] := 0
            Für k := 1, ..., m
                C[i, j] := C[i, j] + A[i, k] · B[k, j]
            Ende Für
        Ende Für
    Ende Für
    Antworte C
Ende Prozedur
```

Damit kann eine $n \times n$-Matrix mit Elementen aus einer Menge R identifiziert werden mit einer 2×2-Matrix deren Elemente $n' \times n'$-Matrizen mit Elementen aus R sind (Diese Beobachtung kann präzisiert werden, z.B. für einen Ring R, indem man zeigt, dass die Matrizenringe $R^{n \times n}$ und $(R^{n' \times n'})^{2 \times 2}$ isomorph sind). Für $C = A \cdot B$ erhalten wir damit die Darstellung

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

der $n \times n$-Multiplikation durch $n' \times n'$-Multiplikation. Diese Beobachtung induziert das in Algorithmus 10 angegebene einfache teile-und-herrsche Verfahren. Hier schreiben wir im Sinne der soeben diskutierten Teilung $A_{1,1}$ für $A[1, \ldots, n'; 1, \ldots, n']$, $A_{1,2}$ für $A[1, \ldots, n'; n' + 1, \ldots, n]$, usw. auch für B und C. Die Laufzeit dieses Algorithmus erfüllt also:

Algorithmus 10 Matrixmultiplikation (rekursiv)

Vorbedingung: A, B sind $n \times n$ Matrizen, n ist Zweierpotenz

```
Prozedur MATMULT2POT(A, B)
    Sei C eine neue n × n-Matrix
    Falls n = 1 dann
        C[1, 1] := A[1, 1] · B[1, 1]
    sonst
        C_{1,1} := MATMULT2POT(A_{1,1}, B_{1,1}) + MATMULT2POT(A_{1,2}, B_{2,1})
        C_{1,2} := MATMULT2POT(A_{1,1}, B_{1,2}) + MATMULT2POT(A_{1,2}, B_{2,2})
        C_{2,1} := MATMULT2POT(A_{2,1}, B_{1,1}) + MATMULT2POT(A_{2,2}, B_{2,1})
        C_{2,2} := MATMULT2POT(A_{2,1}, B_{1,2}) + MATMULT2POT(A_{2,2}, B_{2,2})
    Ende Falls
    Antworte C
Ende Prozedur
```

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1 \\ 8T(\frac{n}{2}) + \Theta(n^2) & \text{falls } n \geq 2 \end{cases}$$

Die Rekursionsgleichung hat die asymptotische Lösung $T(n) = \Theta(n^3)$ wie wir im Kapitel 4 sehen werden, dieser teile-und-herrsche Algorithmus ist also asymptotisch nicht effizienter als das direkte Verfahren. An dieser Stelle könnte man nun glauben, dass der teile-und-herrsche-Ansatz für die Matrixmultiplikation keine Verbesserung des direkten Verfahren erlaubt, das wäre allerdings ein Irrtum wie der Algorithmus von Strassen zeigt.

Der Algorithmus von Strassen folgt ebenso wie der obige der teile-und-herrsche-Methode. Er unterscheidet sich vom obigen dadurch, dass er eine geschicktere Darstellung der $C_{i,j}$ findet, die mit sieben Multiplikationen (von $n' \times n'$-Matrizen) auskommt. Seien nämlich

$$\begin{aligned}
P_1 &= A_{1,1} \cdot (B_{1,2} - B_{2,2}),\\
P_2 &= (A_{1,1} + A_{1,2}) \cdot B_{2,2},\\
P_3 &= (A_{2,1} + A_{2,2}) \cdot B_{1,1},\\
P_4 &= A_{2,2} \cdot (B_{2,1} - B_{1,1}),\\
P_5 &= (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2}),\\
P_6 &= (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2}), \text{ und}\\
P_7 &= (A_{1,1} - A_{2,1}) \cdot (B_{1,1} + B_{1,2}).
\end{aligned}$$

Dann gilt

$$\begin{aligned}
C_{1,1} &= P_5 + P_4 - P_2 + P_6,\\
C_{1,2} &= P_1 + P_2,\\
C_{2,1} &= P_3 + P_4, \text{ und}\\
C_{2,2} &= P_5 + P_1 - P_3 - P_7.
\end{aligned}$$

wie man sich durch eine kurze Rechnung überzeugen kann. Der Strassen-Algorithmus ist dann wie folgt wobei wir (wie oben) die Abkürzungen $A_{i,j}$, $B_{i,j}$ und $C_{i,j}$ auch im Pseudocode verwenden.

Algorithmus 11 Strassen-Algorithmus

Vorbedingung: A, B sind $n \times n$ Matrizen, n ist Zweierpotenz
  Prozedur STRASSEN2POT(A, B)
    Sei C eine neue $n \times n$-Matrix
    Falls $n = 1$ **dann**
      $C[1,1] := A[1,1] \cdot B[1,1]$
    sonst
      $P_1 :=$ STRASSEN$(A_{1,1}, B_{1,2} - B_{2,2})$
      $P_2 :=$ STRASSEN$(A_{1,1} + A_{1,2}, B_{2,2})$
      $P_3 :=$ STRASSEN$(A_{2,1} + A_{2,2}, B_{1,1})$
      $P_4 :=$ STRASSEN$(A_{2,2}, B_{2,1} - B_{1,1})$
      $P_5 :=$ STRASSEN$(A_{1,1} + A_{2,2}, B_{1,1} + B_{2,2})$
      $P_6 :=$ STRASSEN$(A_{1,2} - A_{2,2}, B_{2,1} + B_{2,2})$
      $P_7 :=$ STRASSEN$(A_{1,1} - A_{2,1}, B_{1,1} + B_{1,2})$
      $C_{1,1} := P_5 + P_4 - P_2 + P_6$
      $C_{1,2} := P_1 + P_2$
      $C_{2,1} := P_3 + P_4$
      $C_{2,2} := P_5 + P_1 - P_3 - P_7$
    Ende Falls
    Antworte C
  Ende Prozedur

Die Laufzeit vom Strassen-Algorithmus erfüllt also:

$$T(n) = \begin{cases} \Theta(1) & \text{falls } n = 1 \\ 7T(\frac{n}{2}) + \Theta(n^2) & \text{falls } n > 1 \end{cases}$$

Im nächsten Kapitel werden wir zeigen dass die asymptotische Lösung dieser Rekursionsgleichung $T(n) = \Theta(n^{\log 7})$ ist. Da $\log 7 \approx 2{,}807$ wird dadurch eine Verbesserung des direkten sowie des rekursiven Verfahrens erreicht. Da die Konstanten des Strassen-Algorithmus aber größer sind als bei den beiden anderen Verfahren, wird in der Praxis meist eine Kombination verwendet: für hinreichend große Matrizen wird der Strassen-Algorithmus eingesetzt, für kleinere ein direkteres Verfahren.

Der Algorithmus von Strassen ist ein gutes Beispiel für einen teile-und-herrsche Algorithmus mit einer trickreichen Aufteilungs- und damit auch Kombinationsphase.

Bemerkung 3.1. Die Einschränkung auf quadratische Matrizen und n einer Zweierpotenz ist weder für den einfachen teile-und-herrsche Algorithmus, noch für den Strassen-Algorithmus notwendig. Diese Einschränkung dient lediglich der einfacheren Darstellung da durch sie einige Sonderfälle nicht behandelt werden müssen. In der Tat kann für beliebige n, m, l die Multiplikation einer $n \times m$- mit einer $m \times l$-Matrix durch Multiplikationen von Matrizen kleinerer Größe dargestellt werden, durch eine Zerlegung der Form $n = n_1 + n_2$, $m = m_1 + m_2$, $l = l_1 + l_2$. Setzt man zum Beispiel jeweils $x_1 = \lceil \frac{x}{2} \rceil$ und $x_2 = \lfloor \frac{x}{2} \rfloor$ erhält man einen allgemeineren teile-und-herrsche Algorithmus. Ähnliches gilt für den Strassen-Algorithmus. Für diesen ist allerdings notwendig dass alle $A_{i,j}$ die selbe Größe haben und dass alle $B_{i,j}$ die selbe Größe haben, d.h. also dass n, m und l gerade sind. Das kann erreicht werden durch Anfügen einer Nullzeile oder Nullspalte im Bedarfsfall.

Bemerkung 3.2. Der Algorithmus von Strassen wurde im Jahr 1969 publiziert und hat, wie beschrieben, eine Laufzeitkomplexität von $O(n^{2{,}807\cdots})$. Seitdem konnten weitere Verbesserungen der asymptotischen Laufzeit der Matrixmultiplikation erreicht werden. Ein signifikanter Schritt vorwärts war der Algorithmus von Coppersmith-Winograd aus dem Jahr 1990 mit einer Laufzeitkomplexität von $O(n^{2{,}376})$. Über diesen hinaus konnten bis heute nur geringe Verbesserungen erreicht werden, so z.B. $O(n^{2{,}374})$ von Stothers 2010 und $O(n^{2{,}373})$ von Williams 2011. Diese weiteren Algorithmen haben allerdings so große Konstanten, dass sie in der Praxis keine Bedeutung haben.

Untere Schranken zur Matrixmultiplikation sind kaum bekannt. Klar ist, dass $\Omega(n^2)$ eine triviale untere Schranke ist, da die Eingabe der Größe $2n^2$ ja gelesen und die Ausgabe der Größe n^2 geschrieben werden muss. Eine untere Schranke von $\Omega(n^2 \log n)$ auf der Größe einer gewissen, eingeschränkten, Klasse von Schaltkreisen wurde von Raz 2003 bewiesen.

Kapitel 4

Rekursionsgleichungen

Eine Rekursionsgleichung definiert eine Folge $(x_n)_{n\geq 0}$ durch eine Gleichung, die x_n basierend auf n und den x_i mit $i < n$ definiert sowie hinreichend vielen Anfangswerten. Typischerweise ist man daran interessiert, einen geschlossenen Ausdruck für x_n zu finden, d.h. einen, in dem keine x_i mehr vorkommen.

Beispiel 4.1. Die bekannte Rekursionsgleichung

$$F_n = F_{n-1} + F_{n-2} \text{ für } n \geq 2 \text{ mit } F_0 = 0 \text{ und } F_1 = 1$$

definiert die Folge der Fibonacci-Zahlen $0, 1, 1, 2, 3, 5, 8, 13, \ldots$.

4.1 Lineare Rekursionsgleichungen erster Ordnung

Definition 4.1. Die *Ordnung* einer Rekursionsgleichung ist das kleinste k so dass die Definition von x_n von $\{x_{n-1}, \ldots, x_{n-k}\}$ abhängt.

So hat zum Beispiel die Fibonacci-Gleichung die Ordnung 2.

Definition 4.2. Eine *lineare Rekursionsgleichung erster Ordnung* ist eine Rekursionsgleichung von der Form

$$x_n = a_n x_{n-1} + b_n \text{ für } n \geq 1$$

mit festgelegten Anfangswert x_0.

Beispiel 4.2. Die Rekursionsgleichung

$$x_n = x_{n-1} + b_n \text{ für } n \geq 1 \text{ mit } x_0 = 0$$

hat als Lösung $x_n = \sum_{i=1}^{n} b_n$.

Die Rekursionsgleichung

$$x_n = a_n x_{n-1} \text{ für } n \geq 1 \text{ mit } x_0 = 1$$

hat als Lösung $x_n = \prod_{i=1}^{n} a_n$.

Diese beiden Beispiele können zu dem folgenden Resultat verallgemeinert werden:

Satz 4.1. *Die Rekursionsgleichung $x_n = a_n x_{n-1} + b_n$ für $n \geq 1$ mit festgelegten Anfangswert $x_0 = b_0$ hat die Lösung*

$$x_n = \sum_{i=0}^{n} b_i \prod_{j=i+1}^{n} a_j.$$

Beweis. Mit Induktion: Für $n = 0$ gilt $x_0 = b_0$. Für $n > 0$ haben wir

$$x_n = a_n x_{n-1} + b_n = a_n \left(\sum_{i=0}^{n-1} b_i \prod_{j=i+1}^{n-1} a_j \right) + b_n = \left(\sum_{i=0}^{n-1} b_i \prod_{j=i+1}^{n} a_j \right) + b_n = \sum_{i=0}^{n} b_i \prod_{j=i+1}^{n} a_j.$$

□

4.2 Lineare Rekursionsgleichungen k-ter Ordnung

Wir wollen nun lineare Rekursionsgleichungen beliebiger Ordnung betrachten, schränken uns dabei aber auf den Fall konstanter Koeffizienten ein

Definition 4.3. Eine *homogene lineare Rekursionsgleichung mit konstanten Koeffizienten k-ter Ordnung* ist von der Form

$$x_n = c_{k-1} x_{n-1} + \cdots + c_0 x_{n-k} \text{ für } n \geq k \tag{4.1}$$

Falls die Konstanten $c_0, \ldots, c_{k-1}$ Elemente eines Körpers K sind, können wir die Rekursionsgleichung (4.1) über diesem Körper K auffassen. Wir sagen dass $(a_n)_{n\geq 0}$ eine Lösung von (4.1) in K ist falls alle $a_n \in K$ sind und $a_n = c_{k-1} a_{n-1} + \cdots + c_0 a_{n-k}$ für alle $n \geq k$. Wir definieren $K^\omega = \{(a_n)_{n\geq 0} \mid a_n \in K \text{ für alle } n \geq 0\}$ und stellen fest, dass K^ω ein Vektorraum unendlicher Dimension über K ist.

Lemma 4.1. *Sei K ein Körper, seien $c_0, \ldots, c_{k-1} \in K$, dann ist die Lösungsmenge von (4.1) in K ein Untervektorraum von K^ω mit Dimension k.*

Beweis. Klar ist, dass die Lösungsmenge eine Teilmenge von K^ω ist. Seien nun $(a_n)_{n\geq 0}, (b_n)_{n\geq 0}$ Lösungen von 4.1 und $\lambda, \mu \in K$, dann ist für $n \geq k$

$$\begin{aligned} a_n &= c_{k-1} a_{n-1} + \ldots + c_0 a_{n-k}, \\ b_n &= c_{k-1} b_{n-1} + \ldots + c_0 b_{n-k}, \end{aligned}$$

und damit auch

$$\lambda a_n + \mu b_n = c_{k-1}(\lambda a_{n-1} + \mu b_{n-1}) + \ldots + c_0(\lambda a_{n-1} + \mu b_{n-1}),$$

d.h. also auch $(\lambda a_n + \mu b_n)_{n\geq 0}$ ist eine Lösung von (4.1) und damit ist die Lösungsmenge ein Unterraum von $\mathbb{R}^\omega$. Seien $a_0, \ldots, a_{k-1}$ beliebig, dann ist $(a_n)_{n\geq 0}$ eindeutig bestimmt, und kann geschrieben werden als $(a_n)_{n\geq 0} = a_0 e_0 + \cdots + a_{k-1} e_{k-1}$ wobei $e_i \in K^\omega$ jene Lösung von (4.1) ist, die durch $e_{i,j} = \delta_{i,j}$ für $j = 0, \ldots, k-1$ bestimmt wird. Also ist $e_0, \ldots, e_{k-1}$ Basis des Lösungsraums und damit ist seine Dimension k. □

Für den Spezialfall $k = 1$ und $x_0 = 1$ von (4.1) haben wir in Beispiel 4.2 bereits festgestellt, dass die Lösung $x_n = c_0^n$ ist. Es liegt also nahe, Lösungen von der Form $(\alpha^n)_{n\geq 0}$ zu betrachten. Für eine solche Lösung gilt

$$\alpha^n - c_{k-1}\alpha^{n-1} - \cdots - c_0\alpha^{n-k} = 0.$$

für alle $n \geq k$ und insbesondere für $n = k$:

$$\alpha^k - c_{k-1}\alpha^{k-1} - \cdots - c_1\alpha - c_0 = 0.$$

Diese Beobachtung motiviert die folgende Definition:

Definition 4.4. Das *charakteristische Polynom* der Rekursionsgleichung (4.1) ist

$$\chi(z) = z^k - c_{k-1}z^{k-1} - \cdots - c_1 z - c_0$$

Das heißt also: falls $(\alpha^n)_{n\geq 0}$ Lösung von (4.1) ist, dann ist α Nullstelle von $\chi(z)$. Es gilt auch eine (stärkere) Implikation in die andere Richtung so dass wir das folgende Resultat erhalten:

Satz 4.2. *Sei $\chi(z)$ das charakteristische Polynom von (4.1) mit Nullstellen $\alpha_1, \ldots, \alpha_r$ und Vielfachheiten $m_1, \ldots, m_r$. Dann ist $\{(n^j\alpha_i^n)_{n\geq 0} \mid 1 \leq i \leq r, 0 \leq j < m_i\}$ eine Basis des Lösungsraums von (4.1).*

Beweis. Sei α eine Nullstelle von $\chi(z)$, dann ist $\chi(\alpha) = 0$, d.h. also

$$\alpha^k = c_{k-1}\alpha^{k-1} + \cdots + c_1\alpha + c_0$$

und nach Multiplikation mit α^{n-k}

$$\alpha^n = c_{k-1}\alpha^{n-1} + \cdots + c_1\alpha^{n-k+1} + c_0\alpha^{n-k}.$$

Somit ist $(\alpha^n)_{n\geq 0}$ eine Lösung.
Sei α Doppelnullstelle von $\chi(z)$, dann ist α auch Nullstelle von $\chi'(z)$ und damit vom Polynom $\psi(z) = (n-k)\chi(z) + z\chi'(z)$. Nun ist aber

$$\begin{aligned}\alpha\chi'(\alpha) &= k\alpha^k - (k-1)c_{k-1}\alpha^{k-1} - \cdots - 2c_2\alpha^2 - c_1\alpha \text{ und}\\ (n-k)\chi(\alpha) &= (n-k)\alpha^k - (n-k)c_{k-1}\alpha^{k-1} - \cdots - (n-k)c_1\alpha - (n-k)c_0\end{aligned}$$

und damit

$$\psi(\alpha) = n\alpha^k - (n-1)c_{k-1}\alpha^{k-1} - \cdots - (n-k+1)c_1\alpha - (n-k)c_0.$$

Nach Multiplikation mit α^{n-k} erhalten wir

$$n\alpha^n = (n-1)c_{k-1}\alpha^{n-1} - \cdots - (n-k+1)c_1\alpha^{n-k+1} - (n-k)c_0\alpha^{n-k}$$

also ist auch $(n\alpha^n)_{n\geq 0}$ eine Lösung.
Falls α eine dreifache Nullstelle von $\chi(z)$ ist, ist α eine Doppelnullstelle von $\psi(z)$ und wir wenden die Abbildung $\chi \mapsto \psi$ ein zweites Mal, diesmal auf ψ, an. Induktiv folgt damit: falls α eine m-fache Nullstelle von $\chi(z)$ ist, dann sind $(\alpha^n)_{n\geq 0}, (n\alpha^n)_{n\geq 0}, \ldots, (n^m\alpha^n)_{n\geq 0}$ Lösungen.
Diese k Lösungen sind linear unabhängig, also handelt es sich wegen Lemma 4.1 um eine Basis des Lösungsraums. □

Die allgemeine Lösung von (4.1) hat also die Form

$$x_n = \sum_{i=1}^{r}\sum_{j=0}^{m_i-1} c_{i,j}n^j\alpha_i^n$$

wobei die $c_{i,j}$ Konstanten sind, die von den Anfangswerten abhängen.

Beispiel 4.3. Mit Hilfe des obigen Satzes lässt sich leicht eine explizite Darstellung der Fibonacci-Zahlen berechnen. Das charakteristische Polynom von $F_n = F_{n-1} + F_{n-2}$ ist $\chi(z) = z^2 - z - 1$. Dieses hat die Nullstellen

$$z_{1,2} = \frac{1}{2} \pm \sqrt{\frac{1}{4} + 1} = \frac{1 \pm \sqrt{5}}{2},$$

woraus sich die allgemeine Lösung

$$F_n = c_1 \left(\frac{1+\sqrt{5}}{2} \right)^n + c_2 \left(\frac{1-\sqrt{5}}{2} \right)^n$$

ergibt, wobei c_1 und c_2 von den Anfangsbedingungen abhängen. Für $F_0 = 0$ und $F_1 = 1$ ergibt sich das lineare Gleichungssystem

$$\begin{aligned} c_1 + c_2 &= 0 \\ c_1 \left(\frac{1+\sqrt{5}}{2} \right) + c_2 \left(\frac{1-\sqrt{5}}{2} \right) &= 1 \end{aligned}$$

für c_1 und c_2, das die Lösung $c_1 = \frac{1}{\sqrt{5}}, c_2 = -\frac{1}{\sqrt{5}}$ hat. Wir erhalten also die Darstellung

$$F_n = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n.$$

Bemerkung 4.1. Die obige Darstellung der Fibonacci-Zahlen ist nicht nur von rein theoretischem Interesse, sie bietet auch einen effizienteren Algorithmus zur Berechnung von F_n. Der direkt auf der Definition $F_n = F_{n-1} + F_{n-2}$ basierende Algorithmus benötigt $\Theta(n)$ Zeit zur Berechnung von F_n. Zur Verwendung der obigen Darstellung muss im Wesentlichen a^n (für $a = \frac{1\pm\sqrt{5}}{2}$) berechnet werden. Das ist in Zeit $O(\log n)$ wie folgt möglich: zunächst berechnen wir alle a^n für Zweierpotenz n, also $a^0, a^1, a^2, a^4, a^8, \ldots$. Aus a^i kann $(a^i)^2 = a^{2i}$ mit einer einzigen Multiplikation berechnet werden. Wir schreiben dann n binär und multiplizieren die entsprechenden Potenzen, also z.B. $(37)_\text{b} = 100101$ und damit $a^{37} = a^{32}a^4a^1$.

4.3 Die Substitutionsmethode

Als Substitutionsmethode bezeichnet man die aus den folgenden beiden Schritten bestehende Vorgehensweise zum Auflösen einer Rekursionsgleichung:

1. Errate die Form der Lösung.
2. Beweise dass die Form korrekt ist.

Üblicherweise wird der zweite Schritt mit Induktion gemacht wobei die Verwendung der Induktionshypothese die Form einer *Substitution* eines T-Terms durch die erratene rechte Seite hat. Man verwendet dabei einen Lösungsansatz mit unbekannten Konstanten und bestimmt diese Konstanten erst im Lauf des zweiten Schritts (bzw. deren Existenz zu zeigen). Eine ähnliche Vorgehensweise (Beweisansatz mit unbekannten Konstanten, die erst nach Abschluss des Beweises bestimmt werden) ist in vielen Situationen zum Beweis von Abschätzungen nützlich.

Beispiel 4.4. Eine etwas vereinfachte Variante der Rekursionsgleichung von Sortieren durch Verschmelzen ist

$$T(n) = T(\left\lceil \frac{n}{2} \right\rceil) + T(\left\lfloor \frac{n}{2} \right\rfloor) + n.$$

Wir wollen uns jetzt nicht mehr für die exakte Lösung interessieren, sondern nur noch für eine asymptotische Lösung. Deswegen sind die konkreten Zahlenwerte eines endlichen Anfangs von T irrelevant und werden gar nicht mehr angegeben. Zunächst erraten wir die obere Schranke $T(n) = O(n \log n)$. Wir müssen also zeigen dass $\exists c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : T(n) \leq cn \log n$. Die Konstanten c und n_0 lassen wir noch offen und setzen zu einem Induktionsbeweis an.

Bezüglich des Induktionsanfangs bemerken wir, dass die Funktion $n \mapsto n \log n$ die Nullstellen 0 und 1 hat, danach ist sie positiv. Nachdem wir nicht garantieren können dass $T(0) = T(1) = 0$ müssen wir uns auf $n_0 \geq 2$ einschränken. Dann muss als Induktionsbasis ein hinreichend großes Intervall $[n_0, b[$ gewählt werden, so dass sowohl $n \mapsto \lceil \frac{n}{2} \rceil$ als auch $n \mapsto \lfloor \frac{n}{2} \rfloor$, angewandt auf $n \geq b$ immer noch größer gleich n_0 ist. Offensichtlich reicht dafür $b = 2n_0$. Damit schränken wir uns ein auf c mit $T(m) \leq cm \log m$ für $m \in [n_0, 2n_0[$.

Für den Induktionsschritt sei $n \geq 2n_0$. Wir nehmen an dass $T(m) \leq cm \log m$ für alle $m \in [n_0, n[$ und setzen an mit

$$T(n) \leq c \left\lceil \frac{n}{2} \right\rceil \log \left\lceil \frac{n}{2} \right\rceil + c \left\lfloor \frac{n}{2} \right\rfloor \log \left\lfloor \frac{n}{2} \right\rfloor + n = (*).$$

Falls $n = 2k$, dann

$$\begin{aligned}(*) &= cn \log \frac{n}{2} + n = cn \log n - cn \log 2 + n. \\ &= cn \log n - cn + n\end{aligned}$$

Nun schränken wir uns ein auf $c \geq 1$ und erhalten damit

$$\leq cn \log n.$$

Falls $n = 2k + 1$, dann

$$(*) = c(k+1) \log(k+1) + ck \log k + n \leq cn \log(k+1) + n$$

und da $k = \frac{n-1}{2}$ ist $k + 1 = \frac{n+1}{2}$ und damit

$$= cn \log \frac{n+1}{2} + n = cn \log(n+1) - cn + n.$$

Wir wollen $(*) \leq cn \log n$ erreichen und müssen uns dazu wieder einschränken auf $c \geq 1$. Das allein reicht aber noch nicht, der Term $-cn$ muss zusätzlich zu n auch $cn(\log(n+1) - \log n)$ entfernen. Wir brauchen also ein n_0 so dass für alle $n \geq n_0$ gilt

$$\begin{aligned} cn \log(n+1) - cn + n &\leq cn \log n, \text{ d.h.} \\ \log(n+1) - 1 + \frac{1}{c} &\leq \log n, \text{ d.h.} \\ \log(n+1) - \log n &\leq 1 - \frac{1}{c}, \text{ d.h.} \\ \log \frac{n+1}{n} &\leq 1 - \frac{1}{c}. \end{aligned}$$

Wir müssen also c, n_0 finden so dass die folgenden Bedingungen erfüllt sind:

1. $n_0 \geq 2$
2. $\forall m \in [n_0, 2n_0[: T(m) \leq cm \log m$
3. $c \geq 1$
4. $\forall n \geq 2n_0 : \log \frac{n+1}{n} \leq 1 - \frac{1}{c}$

Dazu wählen wir zuerst ein $c > 1$, dann gibt es n_0 das 4. wahr macht da $c > 1$ impliziert dass $1 - \frac{1}{c} > 0$ und $\frac{n+1}{n} \searrow 1$ und damit $\log \frac{n+1}{n} \searrow 0$ für $n \to \infty$. Falls dieses n_0 zu klein war erhöhen wir es um $n_0 \geq 2$ zu erfüllen. Es bleibt noch, 2. zu erfüllen. Das geschieht durch hinreichende Erhöhung von c bei gleichbleibendem n_0 und die Beobachtung dass dadurch 1., 3. und 4. beibehalten werden.

Damit haben wir also gezeigt dass $T(n) = O(n \log n)$.

Beispiel 4.5. Sei

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

Eine Vermutung ist $T(n) = O(n)$, d.h. $\exists c > 0 \exists n_0 \in \mathbb{N} \forall n \geq n_0 : T(n) \leq cn$. Auf direkte Weise würde das zum folgenden Ansatz für den Induktionsschritt führen:

$$T(n) \leq c \left\lfloor \frac{n}{2} \right\rfloor + c \left\lceil \frac{n}{2} \right\rceil + 1 = cn + 1$$

Wir wollen $T(n) \leq cn$ erreichen was offensichtlich unmöglich ist. Jetzt ist aber $T(n) = O(n)$ trotzdem wahr was durch den folgenden, verbesserten, Ansatz auch gezeigt werden kann: $\exists c > 0, d > 0, n_0 \in \mathbb{N} \forall n \geq n_0 : T(n) \leq cn - d$. Dann erhalten wir

$$T(n) \leq c \left\lfloor \frac{n}{2} \right\rfloor - d + c \left\lceil \frac{n}{2} \right\rceil - d + 1 = cn - 2d + 1 \leq cn - d$$

unter der Voraussetzung $d \geq 1$. Der Induktionsanfang wird wie oben behandelt und so erhält man $T(n) = O(cn - d) = O(n)$.

4.4 Die Rekursionsbaummethode

Die Rekursionsbaummethode besteht im Aufzeichnen des durch eine Rekursionsgleichung induzierten Rekursionsbaums sowie in der anschließenden Summierung der im gesamten Baum anfallenden Kosten. Sie wird oft verwendet, um zu einer Vermutung zu gelangen, die danach mit Substitutionsmethode bewiesen wird. Deshalb wird mit Rekursionsbäumen oft recht ungenau gearbeitet, z.B. lässt man die Rundungsoperatoren $\lfloor \cdot \rfloor$ und $\lceil \cdot \rceil$ weg oder man ersetzt Ausdrücke wie $\Theta(f(n))$ durch $cf(n)$.

Beispiel 4.6. Die Rekursion von Sortieren durch Verschmelzen war

$$T(n) = \begin{cases} \Theta(1) & \text{für } n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{für } n \geq 2 \end{cases}.$$

Wir vereinfachen diese zu $T(n) = 2T(\frac{n}{2}) + cn$ und zeichnen den Rekursionsbaum auf, siehe Abbildung 4.1.

In der ersten Ebene, d.h. am Wurzelknoten, treten Kosten von cn auf. In der zweiten Ebene treten an jedem Knoten Kosten von $c\frac{n}{2}$ auf. Da es in der zweiten Ebene zwei Knoten gibt, treten insgesamt in der zweiten Ebene Kosten von cn auf. In der dritten Ebene gibt es vier Konten

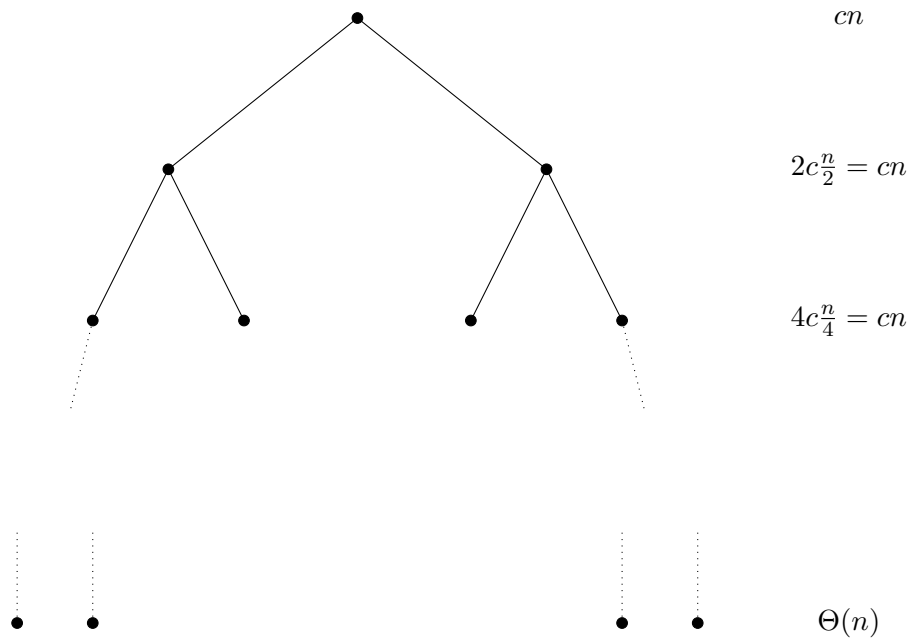


Abbildung 4.1: Rekursionsbaum von $T(n) = 2T(\frac{n}{2}) + cn$

mit Kosten jeweils $c\frac{n}{4}$, also hat auch diese Ebene Kosten von cn, usw. Jede Ebene hat also Kosten cn. Wie viele Ebenen gibt es? Bis n auf eine Konstante (z.B. konkret auf 1) reduziert wurde benötigen wir $\log n$ Ebenen. Der Rekursionsbaum ist also ein vollständiger Binärbaum der Tiefe $\log n$ wobei wir auch hier, im Sinne einer vereinfachenden Annahme, die Tatsache vernachlässigen dass $\log n$ nicht notwendigerweise eine natürliche Zahl ist und nicht klar ist was mit einem Baum reeller Tiefe gemeint sein soll. Ein solcher Baum hat $\Theta(n)$ Blätter. So kommen wir also zur Vermutung $T(n) = cn \log n + \Theta(n) = \Theta(n \log n)$.

Beispiel 4.7. Betrachten wir nun die vereinfachte Rekursionsgleichung

$$T(n) = 3T(\frac{n}{4}) + cn^2$$

Der Rekursionsbaum für diese Gleichung ist in Abbildung 4.2 zu finden. Die erste Ebene hat Kosten cn^2, die zweite $\frac{3}{16}cn^2$, die dritte $\left(\frac{3}{16}\right)^2 cn^2$ usw. Dieser Rekursionsbaum ist ein vollständiger Baum der Arität 3 mit Tiefe $\log_4 n$, er hat also $3^{\log_4 n} = n^{\log_4 3}$ Blätter. Nun beobachten wir dass

$$\sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + n^{\log_4 3} < cn^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i + n^{\log_4 3} = cn^2 \frac{1}{1 - \frac{3}{16}} + n^{\log_4 3} = n^2$$

und erhalten somit die Vermutung $T(n) = O(n^2)$.

4.5 Die Mastermethode

Summenbildungen über Rekursionsbäume, wie sie in den beiden Beispielen in der vorherigen Sektion auftreten, folgen immer dem selben Muster. Wir werden nun einen allgemeinen Satz über solche Summen beweisen, der dann erlaubt die Lösungen der allermeisten Teile-und-herrsche

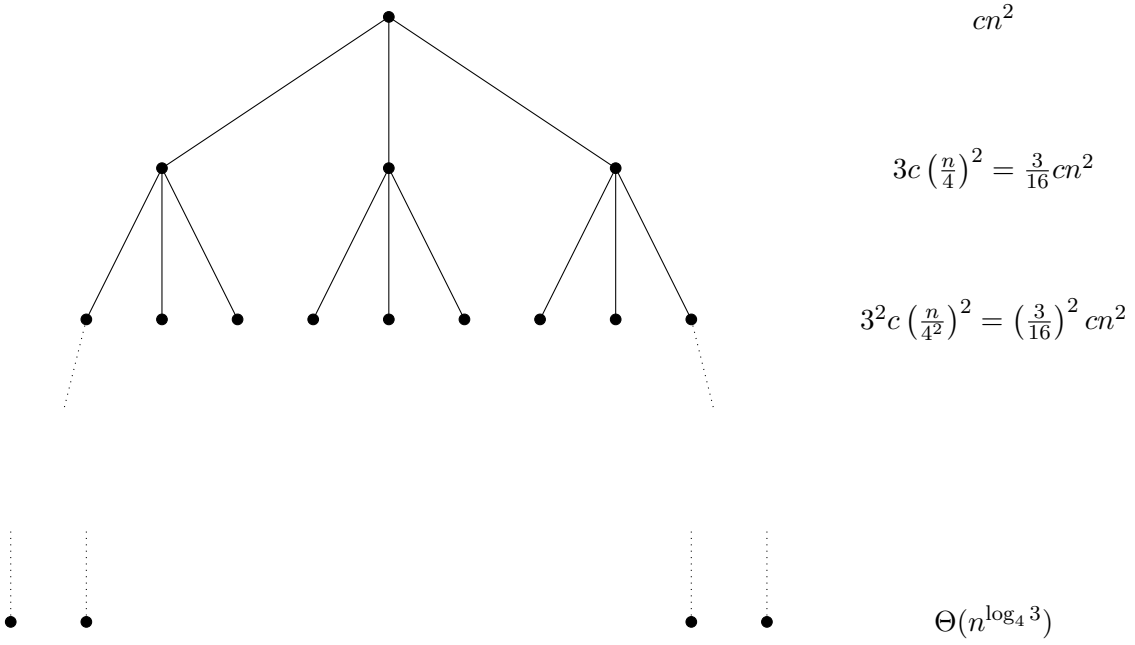


Abbildung 4.2: Rekursionsbaum von $T(n) = 3T(\frac{n}{4}) + cn^2$

Rekursionsgleichungen als Korollare zu erhalten. Dazu betrachten wir $a_0, a_1 \in \mathbb{N}$ mit $a = a_0 + a_1 \geq 1$, $b > 1$ und eine Rekursionsgleichung der Form

$$T(n) = a_0 T(\left\lfloor \frac{n}{b} \right\rfloor) + a_1 T(\left\lceil \frac{n}{b} \right\rceil) + f(n) \tag{4.2}$$

Da wir uns nur für asymptotische Lösungen interessieren, entfällt die Angabe von Anfangswerten. Der Anfang des Rekursionsbaums der Gleichung (4.2) ist in Abbildung 4.3 dargestellt. Um das iterierte auf- und abrunden darzustellen verwenden wir Zeichenketten im Alphabet $\{0, 1\}$ wobei 0 für Abrunden und 1 für Aufrunden steht.

Definition 4.5. $\{0,1\}^*$ ist die Menge aller endlich langen aus 0 und 1 bestehenden Zeichenketten (Wörtern) inklusive dem Leerwort ε. Die Länge $|w|$ eines Wortes $w \in \{0,1\}^*$ ist die Anzahl der Zeichen aus denen w besteht. Die Länge des Leerwortes ist $|\varepsilon| = 0$. Für $w \in \{0,1\}^*$ bezeichnet $n_0(w)$ die Anzahl von Nullern in w und $n_1(w)$ die Anzahl von Einsern in w.

Definition 4.6. Sei $b > 1$. Wir definieren für $n \in \mathbb{N}$ und $w \in \{0,1\}^*$ die Zahl $n_w \in \mathbb{N}$ als

$$n_w = \begin{cases} n & \text{falls } w = \varepsilon \\ \lfloor \frac{n_v}{b} \rfloor & \text{falls } w = 0v \\ \lceil \frac{n_v}{b} \rceil & \text{falls } w = 1v \end{cases}$$

Jeder Knoten im Baum entspricht also T angewandt auf ein bestimmtes n_w. Ein w kommt im Baum an $a_0^{n_0(w)} a_1^{n_1(w)}$ vielen Stellen vor Die lokalen Kosten an einem Knoten der $T(n_w)$ entspricht sind $f(n_w)$. Wir wollen diese lokalen Kosten über alle Knoten im Baum summieren.

Wie tief ist dieser Baum? Falls n eine Potenz von b ist, und somit niemals gerundet wird, ist klar: nach $\log_b n$ Schritten in die Tiefe ist $T(1)$ erreicht. Aber auch der Fall wo n keine Potenz von b ist, ist nicht wesentlich komplizierter: n_w verhält sich bis auf eine additive Konstante so wie $\frac{n}{b^{|w|}}$, siehe Abbildung 4.4.

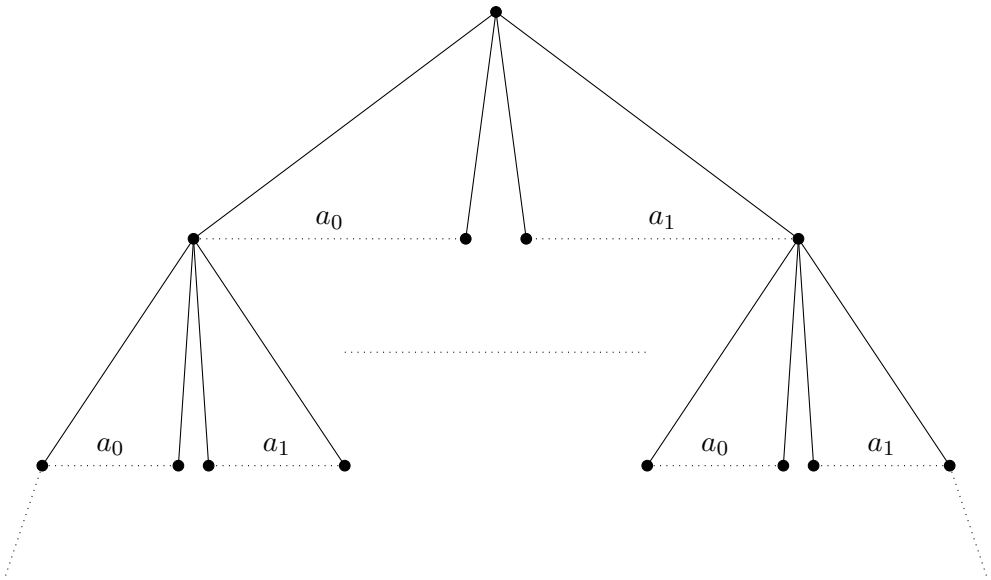


Abbildung 4.3: Anfang des Rekursionsbaums von Gleichung (4.2)

Lemma 4.2. *Sei $b > 1$, dann existiert ein $c > 0$ so dass für alle $n \in \mathbb{N}$ und $w \in \{0,1\}^*$:* $\frac{n}{b^{|w|}} - c < n_w < \frac{n}{b^{|w|}} + c$.

Beweis. Sei $|w| = j$, dann ist[1] $n_{0^j} \leq n_w \leq n_{1^j}$. Wir behaupten weiters dass

$$n_{0^j} \geq \frac{n}{b^j} - \sum_{i=0}^{j-1} \frac{1}{b^i} \quad \text{und} \quad n_{1^j} \leq \frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i}.$$

Wir zeigen die Ungleichung für n_{1^j} mit Induktion nach j. Falls $j = 0$, dann ist $n_\varepsilon = n$. Für den Induktionsschritt gilt

$$n_{1^{j+1}} = \left\lceil \frac{n_{1^j}}{b} \right\rceil \leq \frac{n_{1^j}}{b} + 1 \leq^{\text{IH}} \frac{1}{b}\left(\frac{n}{b^j} + \sum_{i=0}^{j-1} \frac{1}{b^i}\right) + 1 = \frac{n}{b^{j+1}} + \sum_{i=0}^{j} \frac{1}{b^i}.$$

Analog dazu wird die Ungleichung für n_{0^j} gezeigt. Das Resultat folgt nun aus

$$\sum_{i=0}^{j-1} \frac{1}{b^i} < \sum_{i=0}^{\infty} \left(\frac{1}{b}\right)^i = \frac{1}{1-\frac{1}{b}} = \frac{b}{b-1} = c.$$

□

Für die Tiefe $d(n)$ des Baums können wir nun eine beliebige Funktion wählen, die $\frac{n}{b^{d(n)}} = \Theta(1)$ erfüllt. Dann ist nämlich nach Lemma 4.2 auch für alle $w \in \{0,1\}^{d(n)}$ erreicht dass $n_w = \Theta(1)$ und damit auch für alle $w \in \{0,1\}^{d(n)}$ dass $T(n_w) = \Theta(1)$. Wir setzen $d(n) = \lfloor \log_b n \rfloor$ und beobachten dass $\frac{n}{b^{d(n)}} = \Theta(1)$. Dementsprechend definieren wir die Indizes der inneren Knoten und die Indizes der Blattknoten als:

Definition 4.7. $X_I(n) = \{0,1\}^{\leq d(n)-1}$ und $X_B(n) = \{0,1\}^{d(n)}$.

[1] Für einen Buchstaben x und ein $i \in \mathbb{N}$ bezeichnet x^i das Wort das aus i Vorkommen des Buchstaben x besteht.

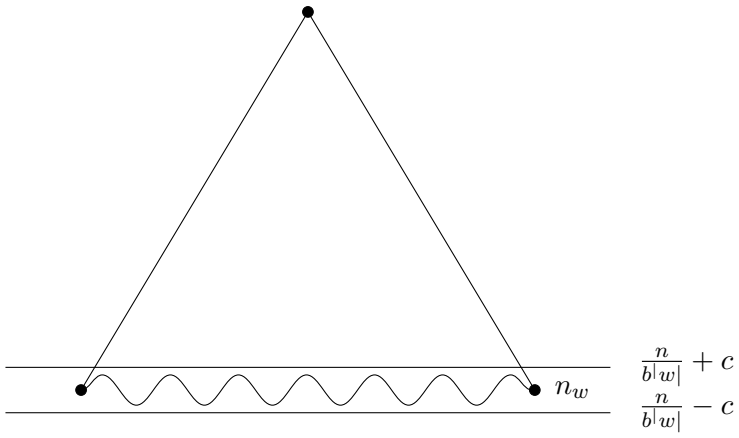


Abbildung 4.4: Illustration von Lemma 4.2 für $w \in \{0,1\}^j$

Die Kosten des gesamten Baums setzen sich zusammen aus den Kosten an den inneren Knoten $\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w)$ und den Kosten an den Blättern. Jedes Blatt verursacht konstante Kosten. Um die Anzahl der Blätter zu bestimmen, können wir den Unterschied zwischen a_0 und a_1 vernachlässigen und den Baum als vollständigen Baum der Arität a mit Tiefe $d(n)$ betrachten. Dieser hat $a^{d(n)} = a^{\lfloor \log_b n \rfloor} = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$ Blätter. Also erhalten wir

$$T(n) = \Theta(n^{\log_b a}) + \sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w).$$

für die Summe über den gesamten Baum.

Eine weitere nützliche Beobachtung ist: Falls wir eine Funktion φ schichtweise summieren wollen die nur von der Länge von w, nicht aber von w selbst abhängt, dann können wir den Unterschied zwischen a_0 und a_1 vernachlässigen und somit den Baum als einen vollständigen Baum mit Arität a betrachten und entsprechend aufsummieren, d.h.

$$\sum_{w \in \{0,1\}^j} a_0^{n_0(w)} a_1^{n_1(w)} \varphi(|w|) = a^j \varphi(j) \text{ und damit}$$

$$\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} \varphi(|w|) = \sum_{j=0}^{d(n)-1} a^j \varphi(j).$$

Satz 4.3 (Master-Theorem für Teile-und-herrsche-Rekursionsgleichungen)**.** *Seien $a_0, a_1 \in \mathbb{N}$ mit $a = a_0 + a_1 \geq 1$, sei $b > 1$ und sei $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$. Dann hat die Rekursionsgleichung*

$$T(n) = a_0 T(\left\lfloor \frac{n}{b} \right\rfloor) + a_1 T(\left\lceil \frac{n}{b} \right\rceil) + f(n)$$

die folgende asymptotischen Lösung.

1. *Falls $f(n) = O(n^{\log_b a - \varepsilon})$ für ein $\varepsilon > 0$, dann $T(n) = \Theta(n^{\log_b a})$.*
2. *Falls $f(n) = \Theta(n^{\log_b a})$, dann $T(n) = \Theta(n^{\log_b a} \log n)$.*
3. *Falls es ein $c < 1$ gibt, so dass für alle bis auf endlich viele $n \in \mathbb{N}$ die Ungleichung $a_0 f(\lfloor \frac{n}{b} \rfloor) + a_1 f(\lceil \frac{n}{b} \rceil) \leq c f(n)$ gilt, dann ist $f(n) = \Omega(n^{\log_b a + \varepsilon})$ für ein $\varepsilon > 0$ und $T(n) = \Theta(f(n))$.*

Wie man sehen kann ist für dieses Resultat vor allem a wesentlich, a_0 und a_1 spielen eine untergeordnete Rolle. Deshalb werden solche Rekursionsgleichungen oft auch geschrieben als $T(n) = aT(\frac{n}{b}) + f(n)$ wobei jedes der a Vorkommen von $\frac{n}{b}$ für $\lceil \frac{n}{b} \rceil$ oder $\lfloor \frac{n}{b} \rfloor$ steht.

Beweis. Sei $g(n) = \sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w)$. In Fall 1 haben wir

$$g(n) = O\left(\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} n_w^{\log_b a - \varepsilon} \right) = O\left(\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} \left(\frac{n}{b^{|w|}} \right)^{\log_b a - \varepsilon} \right)$$

weil $n_w = \Theta(\frac{n}{b^{|w|}})$ und weiters

$$\begin{aligned}
&= O\left(\sum_{j=0}^{d(n)-1} a^j \left(\frac{n}{b^j} \right)^{\log_b a - \varepsilon} \right) = O\left(n^{\log_b a - \varepsilon} \sum_{j=0}^{d(n)-1} \left(\frac{a}{b^{\log_b a - \varepsilon}} \right)^j \right) \\
&= O\left(n^{\log_b a - \varepsilon} \sum_{j=0}^{d(n)-1} (b^\varepsilon)^j \right) = O\left(n^{\log_b a - \varepsilon} \frac{b^{\varepsilon d(n)} - 1}{b^\varepsilon - 1} \right) \\
&= O(n^{\log_b a - \varepsilon} n^\varepsilon) = O(n^{\log_b a})
\end{aligned}$$

weil $b^{d(n)} = \Theta(n)$. Insgesamt erhalten wir also $T(n) = \Theta(n^{\log_b a}) + O(n^{\log_b a}) = \Theta(n^{\log_b a})$.
In Fall 2 haben wir

$$g(n) = \Theta\left(\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} n_w^{\log_b a} \right) = \Theta\left(\sum_{w \in X_I(n)} a_0^{n_0(w)} a_1^{n_1(w)} \left(\frac{n}{b^{|w|}} \right)^{\log_b a} \right)$$

weil $n_w = \Theta(\frac{n}{b^{|w|}})$ und weiters

$$\begin{aligned}
&= \Theta\left(\sum_{j=0}^{d(n)-1} a^j \left(\frac{n}{b^j} \right)^{\log_b a} \right) = \Theta\left(n^{\log_b a} \sum_{j=0}^{d(n)-1} \left(\frac{a}{b^{\log_b a}} \right)^j \right) \\
&= \Theta\left(n^{\log_b a} d(n) \right) = \Theta(n^{\log_b a} \log_b n) = \Theta(n^{\log_b a} \log n).
\end{aligned}$$

Insgesamt erhalten wir also $T(n) = \Theta(n^{\log_b a}) + \Theta(n^{\log_b a} \log n) = \Theta(n^{\log_b a} \log n)$.
In Fall 3 ist $cf(n) \geq a_0 f(\lfloor \frac{n}{b} \rfloor) + a_1 f(\lceil \frac{n}{b} \rceil)$ für alle $n \geq n_0$. Sei nun $d_0(n) = d(n) - p$ wobei $p \in \mathbb{N}$ so gewählt ist, dass $\frac{n}{b^{d_0(n)}} \geq n_0$ für alle $n \geq n_0$. Außerdem ist $\frac{n}{b^{d_0(n)}} = \Theta(1)$. Sei $n \geq n_0$. Wir zeigen zunächst

$$c^j f(n) \geq \sum_{w \in \{0,1\}^j} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w) \text{ für } j = 0, \ldots, d_0(n) \qquad (*)$$

mit Induktion nach j. Falls $j = 0$, dann ist $f(n) = f(n_\varepsilon)$. Für den Induktionsschritt haben wir

$$\begin{aligned}
c^{j+1} f(n) &\geq c \sum_{w \in \{0,1\}^j} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w) \geq \sum_{w \in \{0,1\}^j} a_0^{n_0(w)} a_1^{n_1(w)} (a_0 f(n_{0w}) + a_1 f(n_{1w})) \\
&= \sum_{v \in \{0,1\}^{j+1}} a_0^{n_0(v)} a_1^{n_1(v)} f(n_v).
\end{aligned}$$

Dann ist $c^{d_0(n)} f(n) \geq \sum_{w \in \{0,1\}^{d_0(n)}} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w)$, da aber $\frac{n}{b^{d_0(n)}} = \Theta(1)$ ist wegen Lemma 4.2 auch $\min\{f(n_w) \mid w \in \{0,1\}^{d_0(n)}\}$ eine Konstante q und somit $c^{d_0(n)} f(n) \geq q a^{d_0(n)}$. Damit erhalten wir

$$f(n) \geq q \left(\frac{a}{c}\right)^{\lfloor \log_b n \rfloor - p} = \Theta\left(\left(\frac{a}{c}\right)^{\log_b n}\right) = \Theta(n^{\log_b \frac{a}{c}}).$$

Nun ist $n^{\log_b \frac{a}{c}} = n^{\log_b a - \log_b c}$ und da $c < 1$ war ist $\log_b c < 0$, also ist $f(n) = \Omega(n^{\log_b a + \varepsilon})$ für $\varepsilon = -\log_b a > 0$.

Es bleibt zu zeigen dass $g(n) = \Theta(f(n))$. Zunächst einmal ist klar dass $g(n) = \Omega(f(n))$ da $f(n) = f(n_\varepsilon)$ ja ein Summand von $g(n)$ ist. Für die andere Richtung betrachten wir den Rekursionsbaum bis zur Tiefe $d_0(n)$. Die Anzahl von Blättern ist $a^{d_0(n)} = a^{\lfloor \log_b n \rfloor - p} = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$. Für $j \in \{d_0(n), \ldots, d(n) - 1\}$ haben wir: $\forall w \in \{0,1\}^j$: $\frac{n}{b^{|w|}} = \Theta(1)$, also $\forall w \in \{0,1\}^j$: $n_w = \Theta(1)$, also $\forall w \in \{0,1\}^j$: $f(n_w) = \Theta(1)$. Weiters ist die Anzahl der $f(n_w)$-Summanden an einem Blatt der Tiefe $d_0(n)$ konstant. Also sind die Kosten an einem Blatt der Tiefe $d_0(n)$ konstant und wir erhalten:

$$g(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{d_0(n)-1} \sum_{w \in \{0,1\}^j} a_0^{n_0(w)} a_1^{n_1(w)} f(n_w) \leq \Theta(n^{\log_b a}) + \sum_{j=0}^{d_0(n)-1} c^j f(n)$$

$$< \Theta(n^{\log_b a}) + f(n) \sum_{j=0}^{\infty} c^j = \Theta(n^{\log_b a}) + \frac{1}{1-c} f(n) = \Theta(f(n))$$

da $f(n) = \Omega(n^{\log_b a + \varepsilon})$ für ein $\varepsilon > 0$. Also ist $g(n) = O(f(n))$. Insgesamt erhalten wir also $T(n) = \Theta(n^{\log_b a}) + \Theta(f(n)) = \Theta(f(n))$. □

Korollar 4.1. *Die Rekursionsgleichung*

$$T(n) = T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil) + \Theta(n)$$

die sowohl die Laufzeit von Sortieren durch Verschmelzen als auch jene von unserem Teile-und-herrsche Algorithmus für das dichteste Punktepaar beschreibt hat die Lösung $T(n) = \Theta(n \log n)$.

Beweis. In der Notation des master-Theorems gilt $a = b = 2$ und es trifft der zweite Fall zu. Die Lösung ist also $\Theta(n^{\log_b a} \log n) = \Theta(n \log n)$. □

Korollar 4.2. *Die Rekursionsgleichung*

$$T(n) = 8T(\frac{n}{2}) + \Theta(n^2)$$

die die Laufzeit des einfachen rekursiven Algorithmus zur Matrixmultiplikation beschreibt hat die Lösung $T(n) = \Theta(n^3)$.

Beweis. In der Notation des master-Theorems ist $a = 8, b = 2$ und es trifft der erste Fall zu da $n^2 = O(n^{3-\varepsilon})$ für ein $\varepsilon > 0$. Die Lösung ist also $\Theta(n^{\log_b a}) = \Theta(n^3)$. □

Korollar 4.3. *Die Rekursionsgleichung*

$$T(n) = 7T(\frac{n}{2}) + \Theta(n^2)$$

die die Laufzeit des Strassen-Algorithmus zur Matrixmultiplikation beschreibt hat die Lösung $T(n) = \Theta(n^{2{,}807\ldots})$.

Beweis. In der Notation des master-Theorems ist $a = 7, b = 2$. Es ist $\log_2 7 = 2{,}807\ldots$. Da $n^2 = O(n^{2{,}807\ldots-\varepsilon})$ für ein $\varepsilon > 0$ trifft der erste Fall zu. Die Lösung ist also $\Theta(n^{\log_b a}) = \Theta(n^{2{,}807})$. □

Beispiel 4.8. Auf die Rekursionsgleichung

$$T(n) = T(\left\lfloor \frac{n}{2} \right\rfloor) + T(\left\lceil \frac{n}{2} \right\rceil) + n \log n$$

ist das master-Theorem in dieser Form nicht anwendbar. Es ist nämlich $a = b = 2$, so dass $n \log n$ verglichen werden muss mit n. Nun ist aber $n \log n \neq \Theta(n)$, $n \log n \neq \Omega(n^{1+\varepsilon})$ sowie $n \log n \neq O(n^{1-\varepsilon})$.

Kapitel 5

Datenstrukturen

In diesem Kapitel werden wir uns mit Datenstrukturen beschäftigen. Eine Datenstruktur ist eine Struktur zur Speicherung und Organisation von Daten, die typischerweise gewisse Operationen (auf effiziente Weise) zur Verfügung stellt. Eine einfache Datenstruktur, das Datenfeld, haben wir bereits kennengelernt. Es erlaubt Lese- und Schreibzugriff auf ein beliebiges Element dessen Index bekannt ist in konstanter Zeit. Algorithmen und Datenstrukturen gehen Hand in Hand: jede Datenstruktur benötigt Algorithmen, welche die gewünschten Operationen zur Verfügung stellen und umgekehrt: oft sind gewisse Datenstrukturen notwendig, damit ein Algorithmus ein bestimmte Laufzeit erreicht. So haben wir zum Beispiel im teile-und-herrsche Algorithmus zur Lösung des dichtesten Punktepaarproblems (Algorithmus 8) eine Menge auf eine Weise repräsentiert, die das Durchlaufen anhand zweier unterschiedlicher Ordnungen erlaubt hat. Auch diese Repräsentation kann als Datenstruktur betrachtet werden.

Datenstrukturen sind darüber hinaus auch deswegen von großer Bedeutung in der Informatik, da die Betrachtung von Berechnungsproblemen und Algorithmen als Lösungen dieser zwar für viele aber bei weitem nicht für alle Anwendungen adäquat ist. Oft befindet man sich in der Praxis in einer Situation die dynamisch ist, also nicht durch eine Eingabe-Ausgabe-Relation vollständig beschrieben werden kann.

5.1 Das Wörterbuchproblem

In diesem Abschnitt wollen wir das *Wörterbuchproblem* betrachten. Das *Wörterbuchproblem* besteht darin eine möglichst effiziente Organisation einer endlichen Menge von Datensätzen zu finden die abgefragt und verändert(!) werden kann. Wir nehmen dabei an, dass jeder Datensatz D durch einen eindeutigen Schlüssel $D.x$ identifiziert wird. Wir wollen, dass unser Wörterbuch M zumindest die folgenden Operationen unterstützt:

1. $M.Suche(x)$ gibt jenes D aus M zurück dessen Schlüssel x ist falls es existiert.
2. $M.Einfügen(D)$ fügt den neuen Datensatz D zu M hinzu.
3. $M.Löschen(x)$ entfernt den Datensatz mit Schlüssel x aus M.

Zum Beispiel könnte man an einer Repräsentation aller Studenten dieser Universität interessiert sein. Diese Menge verändert sich im Laufe der Zeit. Die Matrikelnummer kann als eindeutiger Schlüssel dienen. Als Lösung für das Wörterbuchproblem erwarten wir eine geeignete Datenstruktur gemeinsam mit zumindest drei Algorithmen, die die oben beschriebenen Operationen (möglichst effizient) durchführen.

Die einfachste Art ein solches Wörterbuch zu realisieren besteht darin, M als Datenfeld aufzufassen. Die Laufzeitkomplexität der Operationen (wobei $n = |M|$) ist dann wie folgt:

1. $M.\mathit{Suche}(x)$ benötigt Zeit $O(n)$, d.h. genauer: $\Theta(n)$ im schlechtesten Fall und $\Theta(1)$ im besten Fall.
2. $M.\mathit{Einfügen}(D)$ benötigt Zeit $O(n)$ da sichergestellt werden muss, dass kein Duplikat erzeugt wird und dafür im schlechtesten Fall ($D.x$ existiert noch nicht) das gesamte Datenfeld durchlaufen werden muss.
3. $M.\mathit{Löschen}(x)$ benötigt Zeit $\Theta(n)$ da D mit $D.x = x$ zunächst gefunden werden muss und dann das Datenfeld verkürzt werden muss.

Eine bessere Darstellung von M besteht darin, ein nach Schlüssel (aufsteigend) sortiertes Datenfeld zu verwenden. Dann haben wir die folgenden Operationen:

1. $M.\mathit{Suche}(x)$ in Zeit $O(\log n)$ mit binärer Suche (engl. *binary search*), siehe Algorithmus 12.
2. $M.\mathit{Einfügen}(D)$ in Zeit $\Theta(n)$ da das Datenfeld verlängert werden muss.
3. $M.\mathit{Löschen}(x)$ in Zeit $\Theta(n)$ da das Datenfeld verkürzt werden muss.

Algorithmus 12 Binäre Suche

```
Prozedur BSuche(A, x)
    Antworte BSucheRek(A, x, 1, A.Länge)
Ende Prozedur

Prozedur BSucheRek(A, x, l, r)
    Falls l > r dann
        Antworte "Nicht gefunden"
    sonst
        m := ⌈(l+r)/2⌉
        Falls A[m] = x dann
            Antworte m
        sonst falls A[m] < x dann
            Antworte BSucheRek(A, x, m + 1, r)
        sonst                                          ▷ A[m] > x
            Antworte BSucheRek(A, x, l, m − 1)
        Ende Falls
    Ende Falls
Ende Prozedur
```

Mit einem sortierten Datenfeld ist die Suche also effizient, Änderung hingegen nicht.

Eine Datenstruktur in der lokale Änderung auf effiziente Weise gemacht werden können sind verkettete Listen. Hier unterscheidet man zwischen einfach verketteten und doppelt verketteten Listen. Die Grundidee ist, dass die Elemente einer Liste jeweils einzeln an einer beliebigen Stelle im Speicher abgelegt werden und jedes Element auf das nächste (einfache Verkettung) bzw. auf das nächste und das vorherige verweist (doppelte Verkettung). Diese Verweise auf Speicherpositionen werden als *Zeiger* (engl. *pointer*) bezeichnet. Ein Zeiger verweist entweder auf einen bestimmten Ort im Speicher oder er hat den Wert Nil, den Nullzeiger. Realisiert wird ein

einzelner Knoten durch ein Tupel v (für Vertex), dessen Elemente im Fall einer einfach verketteten Liste $v.D$ und $v.nächster$ sind, im Fall einer doppelt verketteten Liste $v.D$, $v.nächster$ und $v.vorheriger$ sind. Der Vorteil einer doppelt verketteten Liste gegenüber einer einfach verketteten besteht darin, dass sie in beide Richtungen durchlaufen werden kann, ihr Nachteil darin, dass sie (konstant) mehr Speicherplatz benötigt. Doppelt verkettete Listen erlauben Einfügen und löschen in konstanter Zeit durch Umsetzen der pointer Eine doppelt verkette Liste als Darstellung von M erlaubt die folgenden Operationen:

1. $M.Suche(x)$ benötigt Zeit $O(n)$ genauer: $\Theta(n)$ im schlechtesten Fall, $\Theta(1)$ im besten Fall und $\Theta(n)$ im Durchschnittsfall.
2. $M.Einfügen(D)$ in Zeit $\Theta(n)$ da sichergestellt werden muss, dass kein Duplikat erzeugt wird
3. $M.Löschen(x)$ wie die Suche in Zeit $O(n)$ da D mit $D.x = x$ gefunden werden muss.

Doppelt verkettete Listen erlauben zusätzlich noch die folgenden effizienten Operationen:

1'. $M.Einfügen(D)$ in Zeit $\Theta(1)$ unter der Annahme dass D noch nicht in M vorkommt.

2'. $M.Löschen(v)$ in Zeit $\Theta(1)$ durch Umsetzen der Zeiger unter der Voraussetzung dass v ein Element von M ist.

Bei verketteten Listen ist zwar im Vergleich zum unsortierten Datenfeld nichts gewonnen was die Suche angeht, allerdings erhalten wir bei den dynamischen Operationen Einfügen und Löschen neue Möglichkeiten.

5.2 Suchbäume

Eine gute Lösung für das Wörterbuchproblem besteht in der Verwendung von *Suchbäumen*. Ein binärer Baum wird im Speicher, ähnlich einer verketteten Liste, so abgelegt, dass jeder Knoten, zusätzlich zu seinem eigentlichen Inhalt dem Datensatz, noch zwei Zeiger enthält: einen auf das linke Kind und einen auf das rechte Kind. Wir schreiben dafür $v.D$, $v.links$ und $v.rechts$. Für Blätter werden diese beiden Zeiger auf Nil gesetzt. Ein Suchbaum ist nun ein auf diese Weise im Speicher abgelegter binärer Baum der die folgende *Suchbaumeigenschaft* hat:

Definition 5.1. Ein Suchbaum ist ein Baum für den die folgende *Suchbaumeigenschaft* gilt: für alle Knoten v gilt:

1. Für jeden Knoten w im linken Teilbaum von v gilt: $w.D.x < v.D.x$.
2. Für jeden Knoten w im rechten Teilbaum von v gilt: $w.D.x > v.D.x$.

Die Suche in einem Suchbaum funktioniert dann wie in Algorithmus 13 beschrieben. Man beachte die Ähnlichkeit dieses Algorithmus zur binären Suche in einem sortierten Datenfeld. Algorithmus 13 hat Laufzeitkomplexität $O(h)$ wobei h die Höhe, d.h. die Länge des längsten Pfades, des Baums ist.

Die Liste aller Elemente die der Suchbaum enthält kann in aufsteigender Reihenfolge ausgegeben werden, indem er in *symmetrischer Reihenfolge* (engl. *inorder*) durchlaufen wird, also zuerst der linke Teilbaum, dann der aktuelle Knoten, dann der rechte Teilbaum, siehe Algorithmus 14.

Algorithmus 13 Suche in einem Suchbaum

```
Prozedur SUCHE(v, x)
    Falls v = NIL dann
        Antworte "Nicht gefunden"
    sonst falls v.D.x = x dann
        Antworte D
    sonst falls v.D.x < x dann
        Antworte SUCHE(v.rechts, x)
    sonst                                          ▷ v.D.x > x
        Antworte SUCHE(v.links, x)
    Ende Falls
Ende Prozedur
```

Algorithmus 14 Durchlaufen eines Suchbaums in symmetrischer Reihenfolge

```
Prozedur SBAUSGABE(v)
    Falls v ≠ NIL dann
        SBAUSGABE(v.links)
        AUSGABE(v.D)
        SBAUSGABE(v.rechts)
    Ende Falls
Ende Prozedur
```

Ein einfacher Algorithmus zum Einfügen eines Elements D ist

Algorithmus 15 Einfügen in einen Suchbaum

```
Prozedur EINFÜGEN(v, D)
    Falls v = NIL dann                  ▷ Leerer Suchbaum wird initialisiert
        Sei v_0 neuer Knoten
        v_0.D := D
        v_0.links := NIL
        v_0.rechts := NIL
        Antworte v_0
    sonst falls v.D.x = D.x dann
        Antworte "Schlüssel existiert bereits"
    sonst falls v.D.x < D.x dann
        v.rechts := EINFÜGEN(v.rechts, D_0)
        Antworte v
    sonst                                          ▷ v.D.x > D.x
        v.links := EINFÜGEN(v.links, D_0)
        Antworte v
    Ende Falls
Ende Prozedur
```

Die Vorgehensweise zum Löschen eines Elements wird am besten zunächst an einem Beispiel illustriert. Angenommen wir wollen aus dem in Abbildung 5.1 angegebenen Suchbaum das Element mit dem Schlüssel 3 löschen. Dann wird dadurch zunächst einmal die Baumstruktur zerstört und wir erhalten den in Abbildung 5.2 angegebenen Suchbaum. Eine konservative Methode zur Wiederherstellung eines Suchbaums, d.h. eine die möglichst wenig verändert, besteht darin,

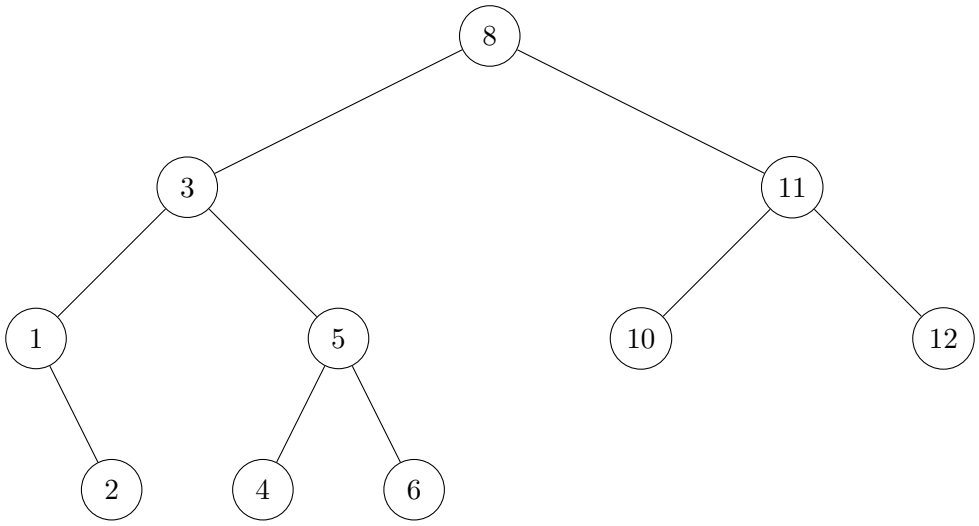


Abbildung 5.1: Suchbaum

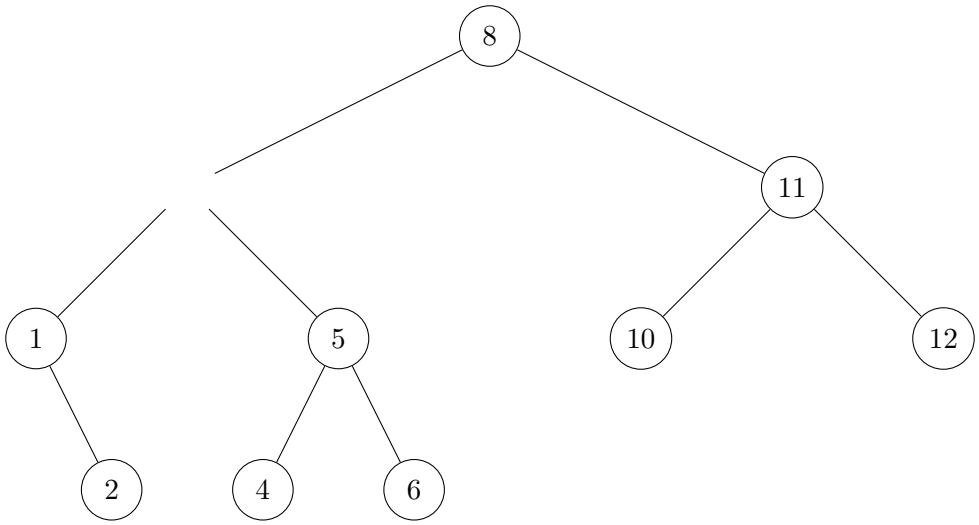


Abbildung 5.2: Suchbaum nach Löschung von 3

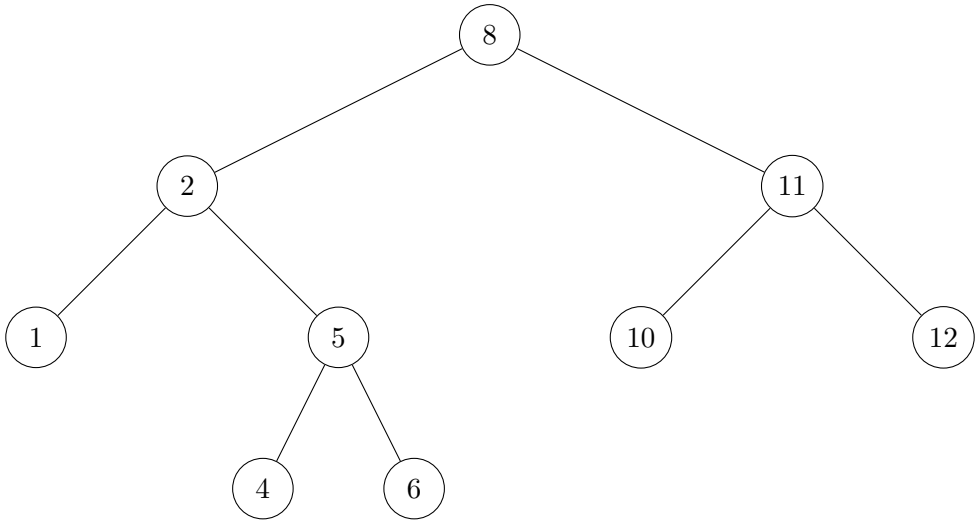


Abbildung 5.3: Suchbaum nach Ersetzung von 3 durch 2

ein geeignetes Element an die entstandene Lücke zu verschieben. Dafür geeignete Element sind 1. das Maximum im linken Teilbaum von 3 sowie 2. das Minimum im rechten Teilbaum von 3. Wenn wir uns für das Maximum m des linken Teilbaums entscheiden und an die Stelle des gelöschten Elements setzen sind danach alle Schlüssel im neuen linken Teilbaum kleiner als m und alle Schlüssel im rechten Teilbaum sind größer als m da sie ja größer als das gelöschte Element waren und dieses wiederum größer als m. Das Argument für das Minimum des rechten Teilbaums ist analog. In unserem Beispiel erhalten wir also den in Abbildung 5.3 angegebenen Suchbaum. Als Pseudocode ist sieht dieser Algorithmus wie folgt aus:

Algorithmus 16 Löschen aus einem Suchbaum

```
Prozedur LÖSCHEN(v, x)
    Falls v = NIL dann
        Antworte "Nicht gefunden"
    sonst falls v.D.x = x dann
        Antworte LÖSCHEWURZEL(v)
    sonst falls v.D.x < x dann
        v.rechts := LÖSCHEN(v.rechts, x)
        Antworte v
    sonst                                          ▷ v.D.x > x
        v.links := LÖSCHEN(v.links, x)
        Antworte v
    Ende Falls
Ende Prozedur
```

Algorithmus 17 Löschen der Wurzel aus einem Suchbaum

Prozedur LöscheWurzel(v)
  Falls $v.links \neq$ Nil **dann**
    $(D_{\max}, v_l) :=$ ExtrahiereMax($v.links$)
    Sei v' neuer Knoten
    $v'.D := D_{\max}$
    $v'.links := v_l$
    $v'.rechts := v.rechts$
    Antworte v'
  sonst falls $v.rechts \neq$ Nil **dann**
    $(D_{\min}, v_r) :=$ ExtrahiereMin($v.rechts$)
    Sei v' neuer Knoten
    $v'.D := D_{\min}$
    $v'.links := v.links$
    $v'.rechts := v_r$
    Antworte v'
  sonst ▷ v ist ein Blattknoten
    Antworte Nil
  Ende Falls
Ende Prozedur

Algorithmus 18 Finde und lösche Minimum aus Suchbaum

Prozedur ExtrahiereMin(v)
  Falls $v =$ Nil **dann**
    Antworte "Leerer Baum, minimum nicht definiert"
  sonst falls $v.links =$ Nil **dann** ▷ v ist Minimum
    Antworte $(v.D, v.rechts)$
  sonst
    $(D_{\min}, v'_l) :=$ ExtrahiereMin($v.links$)
    $v.links := v'_l$
    Antworte $(D_{\min}, v)$
  Ende Falls
Ende Prozedur

Insgesamt erhalten wir also die folgenden Laufzeitkomplexitäten für die Implementierung eines Wörterbuchs als Suchbaum:

1. $M.$*Einfügen*(D) in Zeit $O(h)$
2. $M.$*Löschen*(x) in Zeit $O(h)$
3. $M.$*Suche*(x) in Zeit $O(h)$

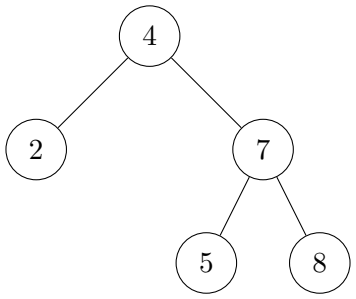
wobei h die Höhe des Baums ist. Das führt zur Frage: Was ist die Höhe des Baums?

Im schlechtesten Fall hat der Baum eine lineare Struktur. Das geschieht wenn die Elemente von M in (aufsteigend oder absteigend) sortierter Reihenfolge eingefügt werden. Dann ist $h = n = |M|$. Im besten Fall ist der Baum ein fast vollständiger Binärbaum, d.h. alle Schichten bis auf die letzte sind vollständig. In diesem Fall ist $h = \Theta(\log n)$.

Wir wollen jetzt auch (einen) Durchschnittsfall analysieren. Auch hier ist a priori nicht klar welche Wahrscheinlichkeitsverteilung betrachtet werden soll. Wir entscheiden uns für die folgende Anwendung: ein Suchbaum wird aus einer Liste $D_1, \ldots, D_n$ mit paarweise verschiedenen Schlüsseln mittels der Einfüge-Operation aufgebaut. Danach verändert er sich nicht mehr und wir suchen nach Elementen aus der Eingabeliste. Wir interessieren uns also für die durchschnittlichen Kosten einer Such-Operation. Nachdem es für diese Anwendung nur auf die Reihenfolge der Schlüssel ankommt, reicht es darauf eine Wahrscheinlichkeitsverteilung zu definieren. Wir nehmen an, dass jede Permutation der Schlüssel die gleiche Wahrscheinlichkeit $\frac{1}{n!}$ hat. Diese Annahme wird in der Literatur als *Permutationsmodell* bezeichnet. Sie hat übrigens *nicht* zur Folge dass jeder Baum mit n Knoten gleich wahrscheinlich ist, da verschiedene Permutationen den selben Baum erzeugen.

Bevor wir mit der Analyse beginnen ist es nützlich noch einige Begriffe für Bäume einzuführen: Die *Tiefe* $\mathrm{d}(v)$ eines Knoten v in einem Baum T ist die Anzahl der Kanten auf dem eindeutigen Pfad von v zur Wurzel von T. Die Anzahl der Schlüsselvergleiche die zum Auffinden eines Knotens v notwendig sind ist $\mathrm{d}(v)+1$. Für einen Baum $T = (V, E)$ definieren wir die *Pfadlänge* $\mathrm{L}(T)$ von T durch $\mathrm{L}(T) = \sum_{v \in V}(\mathrm{d}(v) + 1)$. Die durchschnittlichen Kosten für die Suche eines in T vorhandenen Schlüssels in T belaufen sich auf $\bar{\mathrm{L}}(T) = \frac{\mathrm{L}(T)}{|T|}$.

Beispiel 5.1. Der Baum



hat zum Beispiel $d(2) = 1$, $d(5) = 2, L(T) = 1 + 2 + 2 + 3 + 3 = 11$ und $\bar{\mathrm{L}}(T) = \frac{11}{5} = 2{,}2$.

Satz 5.1. *Im Permutationsmodell ist der Erwartungswert der Kosten einer Such-Operation* $O(\log n)$.

Beweis. Sei T ein Baum. Falls T leer ist, dann ist $\mathrm{L}(T) = 0$. Falls T nicht leer ist, sei T_l dessen linker Teilbaum und T_r dessen rechter Teilbaum. Dann ist

$$\mathrm{L}(T) = \mathrm{L}(T_\mathrm{l}) + |T_\mathrm{l}| + \mathrm{L}(T_\mathrm{r}) + |T_\mathrm{r}| + 1 = \mathrm{L}(T_\mathrm{l}) + \mathrm{L}(T_\mathrm{r}) + |T|. \qquad (*)$$

Dem Permutationsmodell entsprechend wählen wir uniform verteilt eine Eingabepermutation π von n Datensätzen mit paarweise unterschiedlichen Schlüsseln. Diese Eingabepermutation induziert einen Suchbaum T_π mit Pfadlänge $L(T_\pi)$. Wir schreiben $\mathrm{EL}(n)$ für den Erwartungswert der Pfadlänge von T_π.

Das erste Element der Eingabepermutation ist die Wurzel des Baums. Der linke Teilbaum der Wurzel wird so viele Elemente enthalten wie es D_i in der Eingabepermutation gibt deren Schlüssel kleiner als der der Wurzel ist, der rechte Teilbaum so viele Elemente wie es D_i gibt deren Schlüssel größer ist. Nachdem die Eingabepermutation zufällig war, ist für alle $k \in \{1, \ldots, n\}$ die Wahrscheinlichkeit dass das erste Element den k-t-größten Schlüssel in $D_1, \ldots, D_n$ hat $\frac{1}{n}$.

Mit (*) erhalten wir also $\mathrm{EL}(0) = 0$ und, für $n \geq 1$:

$$\begin{aligned}\mathrm{EL}(n) &= \frac{1}{n}\sum_{k=1}^{n}(\mathrm{EL}(k-1) + \mathrm{EL}(n-k) + n) \\ &= n + \frac{1}{n}\left(\sum_{k=1}^{n}\mathrm{EL}(k-1) + \sum_{k=1}^{n}\mathrm{EL}(n-k)\right) \\ &= n + \frac{2}{n}\sum_{k=0}^{n-1}\mathrm{EL}(k).\end{aligned}$$

Wir haben also

$$\begin{aligned}n\mathrm{EL}(n) &= n^2 + 2\sum_{k=0}^{n-1}\mathrm{EL}(k), \\ (n-1)\mathrm{EL}(n-1) &= (n-1)^2 + 2\sum_{k=0}^{n-2}\mathrm{EL}(k), \\ n\mathrm{EL}(n) - (n-1)\mathrm{EL}(n-1) &= 2n - 1 + 2\mathrm{EL}(n-1) \text{ und} \\ \mathrm{EL}(n) &= \frac{2n-1}{n} + \frac{n+1}{n}\mathrm{EL}(n-1).\end{aligned}$$

Somit ist $\mathrm{EL}(n)$ durch eine lineare Rekursionsgleichung erster Ordnung gegeben. Also

$$\begin{aligned}\mathrm{EL}(n) &=^{\text{Satz 4.1}} \sum_{i=1}^{n}\frac{2i-1}{i}\prod_{j=i+1}^{n}\frac{j+1}{j} = (n+1)\sum_{i=1}^{n}\frac{2i-1}{i(i+1)} \\ &= (n+1)\left(\sum_{i=1}^{n}\frac{3}{i+1} - \sum_{i=1}^{n}\frac{1}{i}\right) = (n+1)\left(\frac{3}{n+1} + \sum_{i=2}^{n}\frac{2}{i} - 1\right) \\ &= 2(n+1)H_n - 3n\end{aligned}$$

wobei $H_n = \sum_{i=1}^{n}\frac{1}{i}$ die n-te harmonische Zahl ist. Die n-te harmonische Zahl kann abgeschätzt werden durch $H_n = \ln n + \gamma + O(\frac{1}{n})$ wobei γ die Euler-Mascheroni Konstante ist und einen Wert von 0,577… hat. Wir erhalten also $\mathrm{EL}(n) = O(n \ln n) = O(n \log n)$ und damit $\frac{\mathrm{EL}(n)}{n} = O(\log n)$. □

Suchbäume verhalten sich also (im Permutationsmodell) im Durchschnittsfall so wie im besten Fall (anders als das beim Einfügesortieren, sh. Kapitel 1, der Fall war). Trotzdem ist diese Lösung noch nicht völlig zufriedenstellend. Wir würde das Wörterbuchproblem gerne mit Operationen lösen die auch im schlechtesten Fall logarithmisch sind, umso mehr als der schlechteste Fall eine sortierte Eingabe ist (was nicht in allen Anwendungen nur mit Wahrscheinlichkeit $\frac{1}{n!}$ auftritt).

5.3 AVL-Bäume

Die *Höhe* $\mathrm{h}(v)$ eines Knotens v ist die maximale Anzahl von Knoten auf einem Pfad von v zu einem Blatt. Die Höhe des leeren Baums ist 0. Die Höhe eines nicht-leeren Baums ist die Höhe seiner Wurzel und damit ≥ 1.

Definition 5.2. Sei T ein Suchbaum, v ein Knoten in T, T_l der linke Teilbaum von v und T_r der rechte Teilbaum von v. Dann ist der *Balancegrad* von v definiert als $\mathrm{bal}(v) = \mathrm{h}(T_\mathrm{r}) - \mathrm{h}(T_\mathrm{l})$. Ein Suchbaum T heißt *balanciert* falls für jeden Knoten v von T gilt: $|\mathrm{bal}(v)| \leq 1$.

Satz 5.2. *Ein balancierter Baum mit n Knoten hat Höhe $\Theta(\log n)$.*

Beweis. Ein balancierter Baum mit Höhe h und maximaler Anzahl von Knoten ist der vollständige Binärbaum. Dieser hat $n = \Theta(2^h)$ Knoten. Für einen beliebigen balancierten Baum ist also $h = \Omega(\log n)$.

Ein balancierter Baum mit Höhe h und minimaler Anzahl von Knoten hat für $h = 1$ einen Knoten und für $h = 2$ zwei Knoten. Für $h \geq 3$ hat er zwei Teilbäume, einen davon ein balancierter Baum mit Höhe $h - 1$ und minimaler Anzahl von Knoten, den anderen ein balancierter Baum mit Höhe $h - 2$ und minimaler Anzahl von Knoten. Wir erhalten also für die minimale Anzahl von Knoten n_h in einem balancierten Baum mit Höhe h die Rekursionsgleichung

$$n_h = n_{h-1} + n_{h-2} + 1 \text{ mit } n_1 = 1 \text{ und } n_2 = 2.$$

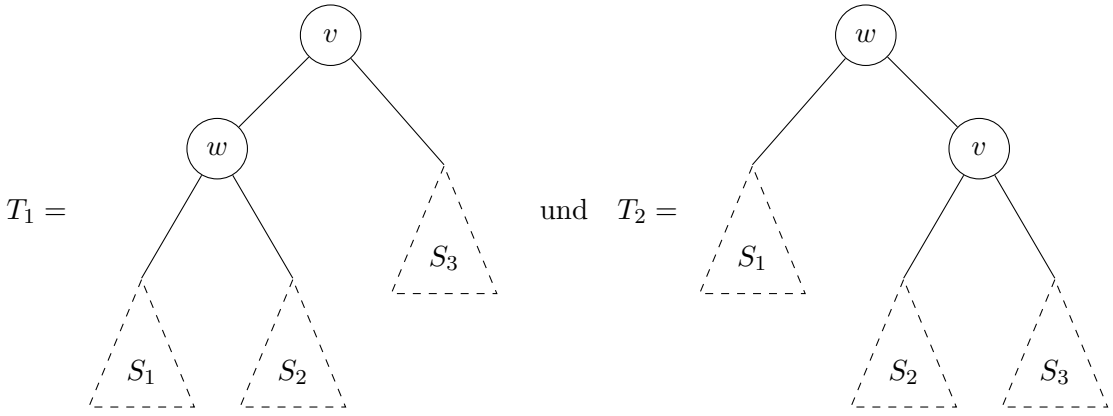
Man sieht sofort dass $n_h \geq F_h$. Wir bissen bereits dass

$$F_h = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^h - \left(\frac{1-\sqrt{5}}{2} \right)^h \right)$$

Nun geht $\left(\frac{1-\sqrt{5}}{2}\right)^h \to 0$ für $h \to \infty$ und damit $F_h = \Theta(\Phi^h)$ für den goldenen Schnitt $\Phi = \frac{1+\sqrt{5}}{2}$. Also $n_h = \Omega(\Phi^h)$. Für einen beliebigen balancierten Baum gilt also $h = O(\log_\Phi n) = O(\log n)$. Insgesamt gilt also für einen beliebigen balancierten Baum $h = \Theta(\log n)$. □

In diesem Abschnitt werden wir AVL-Bäume betrachten, die auf Adelson-Velskij und Landis zurückgehen. Der Ansatz von AVL-Bäumen besteht darin, die Einfüge- und Lösch-Operationen so zu modifizieren dass die Balanciertheit des Baums beibehalten wird. Das wird durch Korrekturtransformationen erreicht, die sich auf die folgenden Rotationsoperationen stützen.

Definition 5.3. Seien



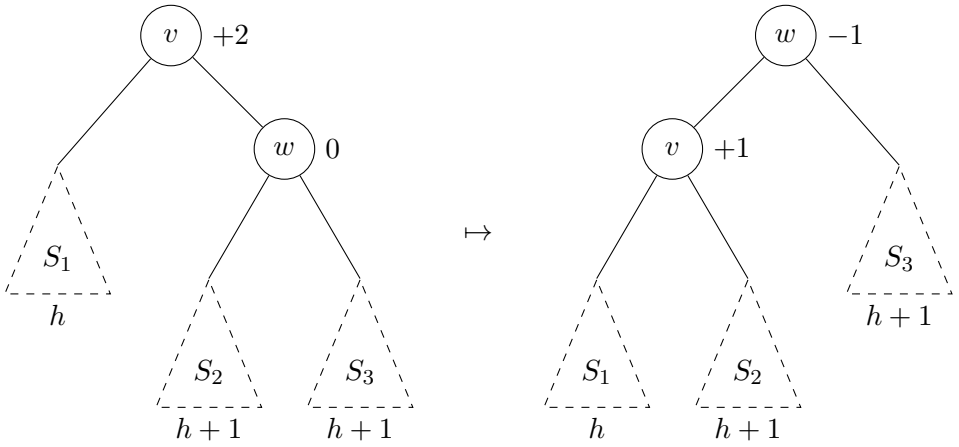
wobei v und w Knoten sind und S_1, S_2 und S_3 Bäume. Dann bezeichnen wir die Abbildung von T_1 auf T_2 als Rechtsrotation (an der Stelle v) und die Abbildung von T_2 nach T_1 als Linksrotation (an der Stelle w).

Die Suchbaumeigenschaft wird durch diese Rotationen beibehalten. Seien nämlich x_1, x_2 und x_3 Schlüssel in S_1, S_2 bzw. S_3, dann gilt $x_1 < w.x < x_2 < v.x < x_3$ sowohl in T_1 als auch in T_2.

Konkret entwickeln wir die Korrekturtransformationen wie folgt: Zunächst einmal stellen wir fest, dass die Eigenschaft der Balanciertheit unabhängig von den Schlüssel der Datensätze ist,

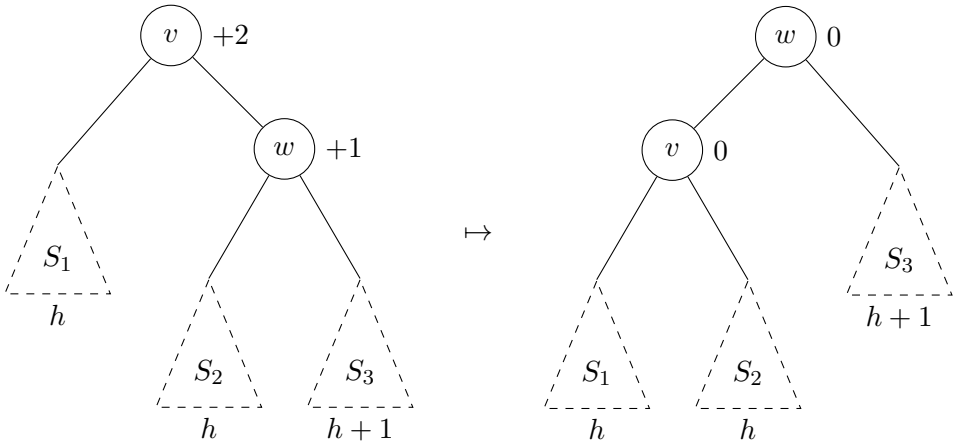
sondern nur von der Form des Baums abhängt. Die Einfügeoperation verändert die Form des Baums indem sie ein neues Blatt hinzufügt, die Löschoperation indem sie ein bestehendes Blatt entfernt. Der Balancegrad aller Knoten die nicht auf dem Pfad von der Wurzel zur Stelle der Änderung liegen bleibt also unverändert. Der Balancegrad der Knoten auf diesem Pfad verändert sich um höchstens 1. Insgesamt haben wir bei der Korrektur der Form des Baums also mit $O(h)$ Knoten zu tun deren Balancegrad in $\{-2, -1, 0, 1, 2\}$ liegt. Wir skizzieren eine Prozedur BALANCIEREN(v). Die Prozedur erhält einen Baum T über dessen Wurzel v als Eingabe. Von T wird angenommen, dass $|\text{bal}(v)| \leq 2$ und für alle Knoten $w \in T \setminus \{v\}$ gilt: $|\text{bal}(w)| \leq 1$. Die Prozedur gibt einen balancierten Suchbaum mit den Elementen von T zurück. BALANCIEREN(v) geht wie folgt vor:

1. Falls $|\text{bal}(v)| \leq 1$, dann antworte mit v.
2. Falls $\text{bal}(v) = +2$, dann existiert ein Knoten $w = v.\textit{rechts}$.
   (a) Falls $\text{bal}(w) = 0$, dann führen wir die folgende Linksrotation durch



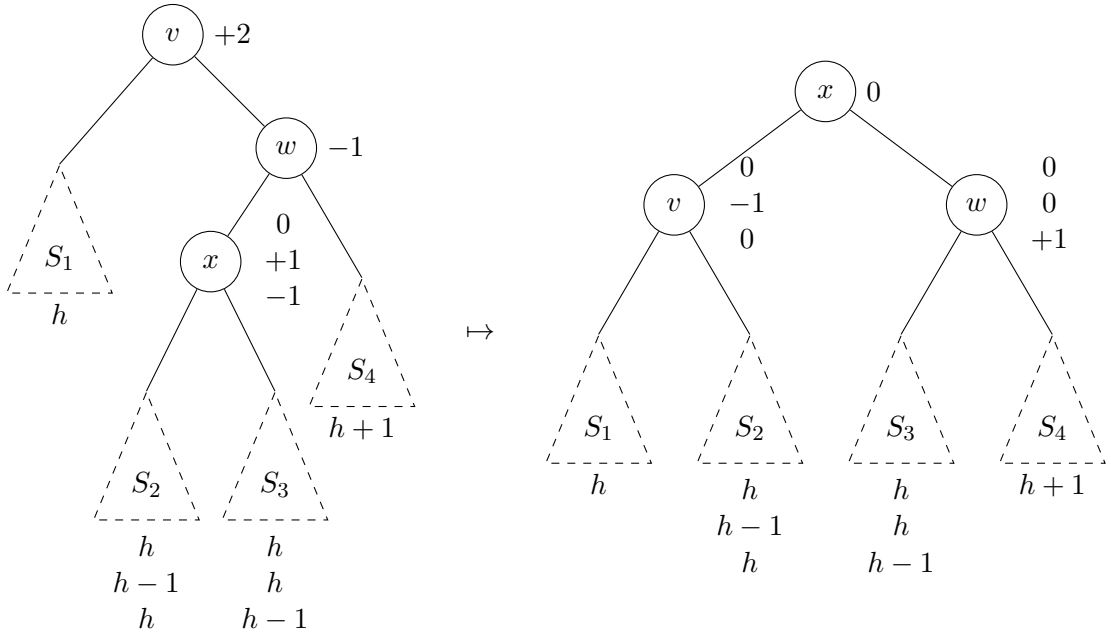
   und antworten mit w. Der Eingabebaum hat Tiefe $h + 3$. Der Ausgabebaum hat ebenfalls Tiefe $h + 3$.

   (b) Falls $\text{bal}(w) = +1$, dann führen wir die folgende Linksrotation durch



   und antworten mit w. Der Eingabebaum hat Tiefe $h+3$. Der Ausgabebaum hat Tiefe $h + 2$.

(c) Falls $\text{bal}(w) = -1$, dann existiert ein Knoten $x = w.\textit{links}$. Wir führen die folgende Doppelrotation durch

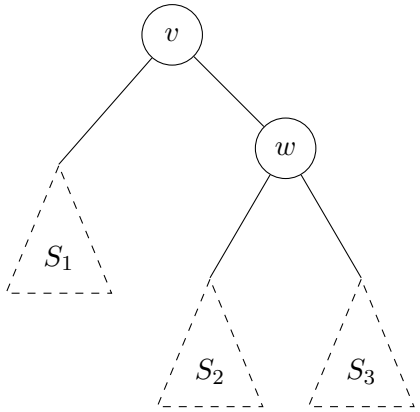


und antworten mit x. Der Eingabebaum hat Tiefe $h+3$. Der Ausgabebaum hat Tiefe $h+2$.

3. Der Fall $\text{bal}(v) = -2$ ist symmetrisch zum vorherigen.

Ein AVL-Baum erlaubt nun eine effiziente Implementierung dieser Prozedur indem jeder Knoten v, zusätzlich zu $v.D$, $v.\textit{links}$, $v.\textit{rechts}$ noch ein Feld $v.\textit{bal}$ für den Balancegrad von v hat. Der Balancegrad wird im Knoten gespeichert um die (linearen) Kosten zu seiner Bestimmung einzusparen. Damit muss der in den Knoten gespeicherte Balancegrad natürlich auch durch alle ändernden Operationen aktualisiert werden. Erreicht ist dadurch dass die Prozedur BALANCIEREN eine Laufzeitkomplexität von $\Theta(1)$ hat.

Die Operationen in AVL-Bäumen können dann folgendermaßen implementiert werden. Die Suchoperation in einem AVL-Baum ist wie die in einem allgemeinen Suchbaum. Die Einfüge- und Lösch-operation müssen an jeder Stelle v den neues Balancegrad $\text{bal}(v)$ berechnen, und, zumindest falls $|\text{bal}(v)| = 2$, die Prozedur BALANCIEREN(v) aufrufen. Den neuen Balancegrad auf die naive Weise zu berechnen, d.h. durch Bestimmung der Höhen der Teilbäume, ist keine Option, da dies lineare Zeit benötigen würde. Um den neuen Balancegrad in konstanter Zeit zu berechnen, muss er aus dem alten Balancegrad mit Hilfe von Informationen berechnet werden, die den Einfüge- und Löschoperationen zur Verfügung stehen. Konkret wird dazu die Veränderung der Höhe der Teilbäume benötigt. Diese beiden Operationen müssen also im rekursiven Aufstieg die Information mitführen, ob der aktuelle Teilbaum seine Höhe verändert hat, d.h. ob sie um 1 gestiegen ist (im Fall von Einfügen) oder um 1 gefallen (im Fall von Löschen). Zur Illustration geben wir hier ein Beispiel. Wir betrachten einen Baum der Form



in dem EINFÜGEN(v, D) den Aufruf EINFÜGEN(w, D) gemacht hat (d.h. der Schlüssel x des einzufügenden Elements war größer als $v.D.x$). Wir gehen davon aus, dass die Prozedur EINFÜGEN so modifiziert wurde, dass sie mit einem Paar $(v, \Delta h)$ antwortet wobei v, wie gehabt, der neue Baum ist und Δh die Differenz zwischen der Höhe des neuen Baums und der Höhe des alten Baums. Ebenso wird die Prozedur BALANCIEREN so modifiziert, dass sie mit dem Paar $(v, \Delta h)$ antwortet wobei v wiederum der neue Baum ist und Δh die Höhenänderung vom alten zum neuen Baum. Sei nun (w, Δ_h) die Antwort von EINFÜGEN(w, D). Der Balancegrad von v sowie Δh_v berechnen sich dann wie folgt:

$v.bal := v.bal + \Delta h_w$
$(\Delta_b, v) :=$ BALANCIEREN(v)
$\Delta h_v := \Delta h_w + \Delta_b$

An den anderen Stellen des Pseudocodes von Einfügen und Löschen werden ähnliche Änderungen durchgeführt. Diese Operationen haben damit eine Laufzeit von $O(\log n)$ wobei n die Anzahl der Datensätze ist.

Logarithmische Komplexität ist in der Praxis sehr gering. Zum Beispiel kann ein Suchbaum der alle ca. 8 Mrd. Menschen die auf der Erde leben enthält mit einer Tiefe von ca. 33 erstellt werden. Mit AVL-Bäumen haben wir also das Wörterbuchproblem auf zufriedenstellende Weise gelöst.

Suchbäume erlauben auch das folgende Sortierverfahren: ein gegebenes Datenfeld A kann sortiert werden durch 1. einen Aufbau eines Suchbaums für A durch wiederholten Einfügen in den leeren Baum sowie 2. den Durchlauf dieses Suchbaums in symmetrischer Reihenfolge, siehe Algorithmus 14. Der zweite Schritt benötigt Laufzeit $\Theta(n)$. Bei Verwendung von AVL-Bäumen benötigt der erste Schritt Zeit $O(n \log n)$ und der gesamt Algorithmus damit (auch im schlechtesten Fall) $O(n \log n)$.

5.4 Stapel und Warteschlangen

Ein Stapel (engl. *stack*) ist eine Datenstruktur welche die folgenden Operationen zur Verfügung stellt:

1. Hinzufügen(S,D) legt den Datensatz D auf den Stapel S (engl. *push*).
2. Entfernen(S) gibt das oberste Element vom Stapel zurück und entfernt es vom Stapel (engl. *pop*).

Dieses Prinzip bezeichnet man auch als LIFO "last in first out". Ein Stapel kann als einfach

verkettete Liste S mit einem Startzeiger $S.Start$ implementiert werden, wobei der Stapel leer ist genau dann wenn $S.Start = \text{NIL}$, siehe Algorithmus 19. Diese Operationen benötigen jeweils Zeit $\Theta(1)$.

Algorithmus 19 Stapel

```
Prozedur HINZUFÜGEN(S, D)
    Sei v neuer Knoten
    v.D := D
    v.nächster := S.Start
    S.Start := v
Ende Prozedur

Prozedur ENTFERNEN(S)
    Falls W.Start = NIL dann
        Antworte "Stapel leer"
    sonst
        D := S.Start.D
        S.Start := S.Start.nächster
        Antworte D
    Ende Falls
Ende Prozedur
```

Wir wollen nun eine Warteschlange (ähnlich wie an einer Supermarktkassa) implementieren. Es soll möglich sein, das erste Element aus einer Warteschlange zu entfernen (z.B. um es abzuarbeiten) sowie ein neues Element an das Ende der Warteschlange zu stellen. Die Elemente werden also in der Reihenfolge ihres Einlangen abgearbeitet. Dieses Prinzip bezeichnet man auch als FIFO "first in first out", die gewünschte Datenstruktur als Warteschlange (engl. *FIFO queue*). Wir wollen die folgenden Operationen

1. Hinzufügen(W,D) stellt D an das Ende der Warteschlange W.
2. Entfernen(W) gibt das Element vom Anfang der Warteschlange zurück und entfernt es.

Eine Möglichkeit eine solche Warteschlange W zu implementieren besteht darin ein einfach verlinkte Liste zu verwenden mit dem Startzeiger $W.Start$ und einem zusätzlichen Endezeiger $W.Ende$. Die Implementierung in Algorithmus 20 geht davon aus, dass die Warteschlange leer ist genau dann wenn der Startzeiger NIL ist. Wie man leicht sehen kann benötigen diese Operationen jeweils $\Theta(1)$ Zeit.

5.5 Prioritätswarteschlangen

Oft reicht ein so einfaches Konzept einer Warteschlange aber nicht aus. Von einer Prioritätswarteschlange spricht man, wenn die Elemente nach Priorität sortiert sind. Für einen Datensatz D schreiben wir jetzt $D.x$ für die Priorität von D. Die Priorität übernimmt also die Funktion des Schlüssels als Sortierkriterium. Wie nehmen allerdings nicht mehr an, dass die Prioritäten paarweise verschieden sind. Für Prioritätswarteschlange Q wollen wir zumindest die folgenden Operationen zur Verfügung stellen:

1. *Einfügen*(Q, D) fügt das Element D in Q (an geeigneter Stelle) ein.

Algorithmus 20 Warteschlange

```
Prozedur HINZUFÜGEN(W, D)
    Sei v neuer Knoten
    v.D := D
    v.nächster := NIL
    Falls W.Start = NIL dann
        W.Start := v
    sonst
        W.Ende.nächster := v
    Ende Falls
    W.Ende := v
Ende Prozedur

Prozedur ENTFERNEN(W)
    Falls W.Start = NIL dann
        Antworte "Warteschlange leer"
    sonst
        D := W.Start.D
        W.Start := W.Start.nächster
        Antworte D
    Ende Falls
Ende Prozedur
```

2. *Maximum*(Q) gibt das Element maximaler Priorität zurück.

3. *ExtrahiereMaximum*(Q) löscht das Element maximaler Priorität aus Q und gibt es zurück.

4. *ErhöhePriorität*(Q, D, x) erhöht die Priorität des Elements D auf x wobei angenommen wird dass $x \geq D.x$.

Eine Prioritätswarteschlangen wird üblicherweise basierend auf der Datenstruktur der *Halde* (engl. *heap*) implementiert.

Definition 5.4. Eine Halde ist ein binärer Baum mit den folgenden Eigenschaften:

1. Alle bis auf das unterste Level sind vollständig aufgefüllt.

2. Das unterste Level ist von links bis zu einem bestimmten Punkt vollständig aufgefüllt.

3. Falls w ein Kind von v ist, dann ist Priorität$(w) \leq$ Priorität(v).

Daraus folgt unmittelbar dass die Wurzel des Baums maximale Priorität hat. Eine Halde kann (ähnlich wie ein Suchbaum) im Speicher mit Hilfe von Knoten abgelegt werden, die jeweils einen Zeiger *links* und einen Zeiger *rechts* haben. Wie wollen hier aber eine andere Darstellung angeben, die, je nach Anwendung, gewisse Vor- und Nachteile hat. Da alle Level bis auf das letzte vollständig ausgefüllt sind, können wir die Halde auf effiziente Weise in einem Datenfeld speichern das Level für Level von oben nach unten und, pro Level, von links nach rechts befüllt wird.

Beispiel 5.2. Die in Abbildung 5.4 als Baum dargestellte Halde hat als Datenfeld die folgende Form:

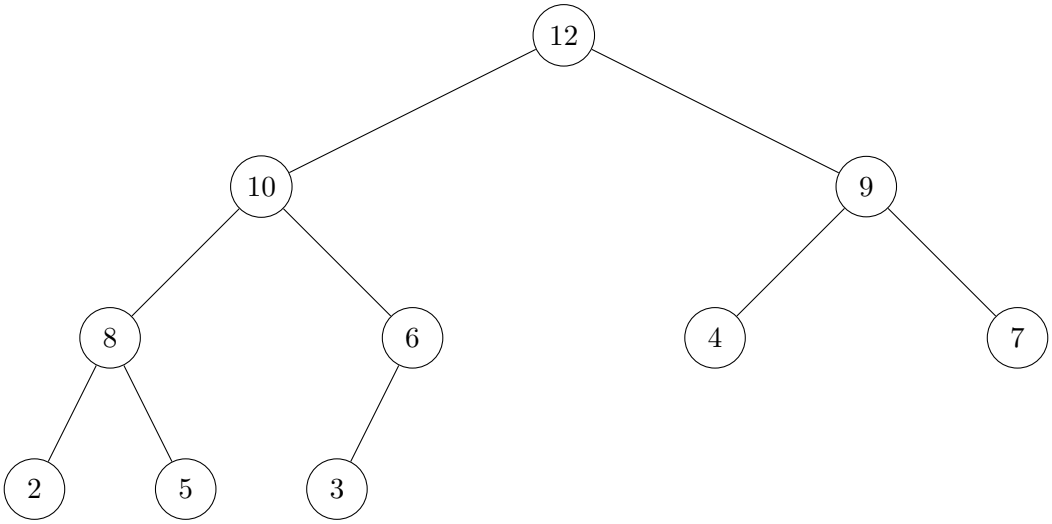


Abbildung 5.4: Eine Halde in Baumdarstellung

12	10	9	8	6	4	7	2	5	3

Um diese Darstellung einer Halde Q zu realisieren, speichern wir zusätzlich zu $Q.Länge$ auch noch die Haldenlänge $Q.HLänge$ die angibt bis zu welchem Punkt das Datenfeld befüllt ist. Wir werden bei allen Operationen annehmen dass $Q.Länge$ hinreichend groß ist. In der Praxis ist das z.B. dann erfüllt wenn im Vorhinein bekannt ist, wie viele Elemente Q höchstens enthalten wird. Falls diese Anzahl nicht im Vorhinein bekannt ist, muss bei einer Einfügeoperation gegebenenfalls die Länge des zugrundeliegenden Datenfelds erhöht werden, was mit zusätzlicher Komplexität verbunden sein kann.

Wir werden Elemente der Halde mit ihrem Index im zugrundeliegenden Datenfeld referenzieren. Dann hat das linke Kind des Elements mit Index i den Index $2i$, das rechte Kind den Index $2i+1$ und der Vater den Index $\lfloor \frac{i}{2} \rfloor$. Dementsprechend definieren wir die Prozeduren LINKS$(i) := 2i$, RECHTS$(i) := 2i + 1$ und VATER$(i) := \lfloor \frac{i}{2} \rfloor$.

Das Einfügen eines neuen Elements geschieht dadurch, dass es zunächst an die letzte Stelle geschrieben wird und danach aufwärts an die richtige Stelle verschoben wird, siehe Algorithmus 21. Auf ähnliche Weise kann die Erhöhung der Priorität eines Elements implementiert werden, siehe Algorithmus 22. AUFWÄRTSKORRIGIEREN und damit auch EINFÜGEN und ERHÖHEPRIORITÄT benötigt in einer Halde mit n Elementen $O(\log n)$ Zeit.

Zur Extraktion des Maximums geht man dual dazu vor: zunächst wird das Maximum (das sich ja an der Wurzel $Q[1]$ befindet) entfernt und durch das Element am Ende der Prioritätswarteschlange ersetzt. Danach wird dieses Element, das ja jetzt zu niedrige Priorität für seine Position hat, Stück für Stück durch Vertauschungen an eine korrekte Stelle (abwärts) verschoben, siehe Algorithmus 23. ABWÄRTSKORRIGIEREN und damit auch EXTRAHIEREMAXIMUM benötigt ebenfalls $O(\log n)$ Zeit.

Algorithmus 21 Einfügen in eine Prioritätswarteschlange

```
Prozedur EinfügenMax(Q, D)
    Q.HLänge := Q.HLänge + 1                ▷ Annahme: Q.Länge ist hinreichend groß
    Q[Q.HLänge] := D
    AufwärtsKorrigieren(Q, Q.HLänge)
Ende Prozedur

Prozedur AufwärtsKorrigieren(Q, i)
    Solange i > 1 und Q[Vater(i)].x < Q[i].x
        Vertausche Q[i] und Q[Vater(i)]
        i := Vater(i)
    Ende Solange
Ende Prozedur
```

Algorithmus 22 Erhöhung der Priorität in einer Prioritätswarteschlange

Vorbedingung: $x \geq Q[i].x$

```
Prozedur ErhöhePriorität(Q, i, x)
    Q[i].x := x
    AufwärtsKorrigieren(Q, i)
Ende Prozedur
```

Algorithmus 23 Extraktion des Maximums aus Prioritätswarteschlange

```
Prozedur ExtrahiereMaximum(Q)
    Falls Q.HLänge = 0 dann
        Antworte "Q ist leer."
    sonst
        max := Q[1]
        Q[1] := Q[Q.HLänge]
        Q.HLänge := Q.HLänge − 1
        AbwärtsKorrigieren(Q, 1)
        Antworte max
    Ende Falls
Ende Prozedur

Prozedur AbwärtsKorrigieren(Q, i)
    Solange i ≤ Q.HLänge
        Sei m ∈ {i, links(i), rechts(i)} ∩ [1, ..., Q.HLänge] so dass Q[m].x maximal ist
        Falls m = i dann                    ▷ Abwärts-Korrektur beendet
            i := Q.HLänge + 1
        sonst
            Vertausche Q[i] und Q[m]
            i := m
        Ende Falls
    Ende Solange
Ende Prozedur
```

Anders als bei den Operationen für AVL-Bäume, die funktional und rekursiv implementiert waren, haben wir uns bei diesen Korrekturoperationen für eine imperative Implementierung

entschieden. Der funktionale Programmierstil harmoniert üblicherweise gut mit Bäumen (und sonstigen rekursiv definierten Datenstrukturen), der imperative mit Datenfeldern (und anderen Datenstrukturen mit globalem Zugriff).

Auf Basis von Halden lässt sich auch ein Sortierverfahren angeben: *Haldensortieren* (engl. *heapsort*). Die Grundidee dieses Verfahrens besteht darin aus dem Eingabendatenfeld eine Halde aufzubauen und dann Schritt für Schritt jeweils das Maximum der Halde zu entfernen und an das Ende des Ausgabedatenfelds zu schreiben. Aufgrund der gewählten Implementierung einer Halde als Datenfeld können wir in diesem Verfahren für das Eingabedatenfeld, die Halde und das Ausgabedatenfeld den selben Speicherbereich benutzen, siehe Algorithmus 24. Man sieht

Algorithmus 24 Haldensortieren (engl. *heapsort*)

```
Prozedur HaldensortierenHalde(A)
    ErzeugeHalde(A)
    Für i := A.Länge, ..., 2
        Vertausche A[1] und A[i]
        A.HLänge := A.HLänge − 1
        AbwärtsKorrigieren(A, 1)
    Ende Für
Ende Prozedur

Prozedur ErzeugeHalde(A)
    A.HLänge := A.Länge
    Für i := Vater(A.HLänge), ..., 1
        AbwärtsKorrigieren(A, i)
    Ende Für
Ende Prozedur
```

sofort ein, dass ErzeugeHalde in Zeit $O(n \log n)$ läuft. Tatsächlich benötigt diese Prozedur sogar nur Zeit $O(n)$: Eine Halde mit n Elementen hat Höhe $\lfloor \log n \rfloor$ und höchstens $\lceil \frac{n}{2^{h+1}} \rceil$ Knoten der Höhe h. AbwärtsKorrigieren(A, i) benötigt Zeit $O(h)$ wobei h die Höhe von $A[i]$ ist. Insgesamt erhalten wir also für die Laufzeit von ErzeugeHalde:

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

Aus $\sum_{k=0}^{\infty} = \frac{1}{1-x}$ erhält man durch Differenzieren und Multiplikation mit x die Gleichung $\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$ und damit für $x = \frac{1}{2}$:

$$= O(n).$$

Insgesamt benötigt Haldensortieren dann Zeit $O(n \log n)$.

Statt einer Sortierung nach maximaler Priorität kann natürlich auch eine Prioritätswarteschlange mit Sortierung nach minimaler Priorität auf symmetrische Weise implementiert werden. Die hier beschriebenen Datenstrukturen heißen in der Literatur auch *max-priority queue* basierend auf *max-heaps*, die symmetrischen *min-priority queue* basierend auf *min-heaps*.

Kapitel 6

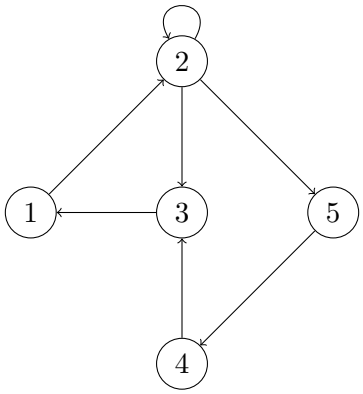
Suchen und Sortieren in Graphen

6.1 Graphen

Definition 6.1. Ein Graph ist ein Paar $G = (V, E)$ mit $E \subseteq V \times V$.

Die Elemente von V heißen *Knoten*, die Elemente von E *Kanten*. Graphen werden oft aufgezeichnet indem die Kanten als Pfeile zwischen den Knoten dargestellt werden.

Beispiel 6.1. Der Graph $G = (V, E)$ mit $V = \{1, 2, 3, 4, 5\}$ und $E = \{(1, 2), (2, 2), (2, 3), (2, 5), (3, 1), (4, 3), (5, 4)\}$ kann z.B. folgendermaßen gezeichnet werden.



Ein Graph $G = (V, E)$ heißt *ungerichtet* falls für alle $(x, y) \in E$ auch $(y, x) \in E$ ist. Beim Zeichnen eines ungerichteten Graphen wird eine Kante nicht als Pfeil sondern als Linie dargestellt. Zwei Knoten $x, y \in V$ heißen *adjazent* falls $(x, y) \in E$ oder $(y, x) \in E$. Der *Ausgangsgrad* von $v \in V$ ist $\mathrm{d}^+(v) = |\{w \in V \mid (v, w) \in E\}|$. Der *Eingangsgrad* von $v \in V$ ist $\mathrm{d}^-(v) = |\{u \in V \mid (u, v) \in E\}|$. Ein Pfad ist eine Liste $v_1, \ldots, v_n \in V$ so dass für alle $i \in \{1, \ldots, n-1\}$ gilt: $(v_i, v_{i+1}) \in E$.

Graphen treten in einer Unzahl von Anwendungskontexten und Berechnungsproblemen auf, wobei man es in der Informatik naturgemäß üblicherweise mit endlichen Graphen zu tun hat. Wir haben bereits Suchbäume kennengelernt die als Graphen betrachtet werden können. Andere Beispiele für Situationen die durch Graphen modelliert werden können sind: Verkehrsnetze wobei die Knoten z.B. Städten entsprechen und die Kanten Zugverbindungen, das WWW wobei die Knoten Webseiten und die Kanten Hyperlinks entsprechen oder auch Landkarten wobei jedem Land ein Knoten entspricht und zwei Konten durch eine ungerichtete Kante verbunden sind

falls sie aneinander grenzen. Beispiele für Berechnungsprobleme aus der Graphentheorie sind:

Kürzester Pfad

Eingabe: endlicher Graph $G = (V, E)$, Gewichtsfunktion $g : E \to \mathbb{R}_{\geq 0}$, $s, t \in V$

Ausgabe: Pfad $v_1, \ldots, v_n$ mit $v_1 = s$, $v_n = t$ so dass $\sum_{i=1}^{n-1} g(v_i, v_{i+1})$ minimal ist

Problem des Handlungsreisenden (engl. *Travelling Salesman Problem (TSP)*)

Eingabe: endlicher Graph $G = (V, E)$, Gewichtsfunktion $g : E \to \mathbb{R}_{\geq 0}$

Ausgabe: Pfad $v_1, \ldots, v_n, v_{n+1} = v_1$ mit $\{v_1, \ldots, v_n\} = V$ so dass $\sum_{i=1}^{n} g(v_i, v_{i+1})$ minimal ist

Knotenfärbung

Eingabe: endlicher ungerichteter Graph $G = (V, E)$

Ausgabe: Abbildung $f : V \to \{1, \ldots, k\}$ so dass $(x, y) \in E \Rightarrow f(x) \neq f(y)$ und k minimal ist

Es gibt verschiedene Datenstrukturen zur Repräsentation eines Graphen.

Definition 6.2. Die *Adjazenzmatrix* eines Graphen $G = (\{1, \ldots, n\}, E)$ ist die Matrix $M = (m_{i,j})_{1 \leq i,j \leq n}$ wobei

$$m_{i,j} = \begin{cases} 1 & \text{falls } (i,j) \in E \\ 0 & \text{falls } (i,j) \notin E \end{cases}$$

Beispiel 6.2. Die Adjazenzmatrix des in Beispiel 6.1 angegebenen Graphen ist

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Eine Adjazenzmatrix wird im Speicher als ein Datenfeld von Datenfeldern abgelegt. Der für eine Adjazenzmatrix benötigte Speicherplatz ist $\Theta(|V|^2)$. Auf Basis einer Adjazenzmatrix kann von gegebenen Knoten v und w in Zeit $\Theta(1)$ festgestellt werden ob der Graph die Kante (v, w) enthält.

Definition 6.3. Die *Adjazenzliste* eines Graphen $G = (\{1, \ldots, n\}, E)$ ist ein Datenfeld L der Länge n wobei $L[i]$ die (einfach verkettete) Liste jener $j \in \{1, \ldots, n\}$ ist für die $(i, j) \in E$ gilt.

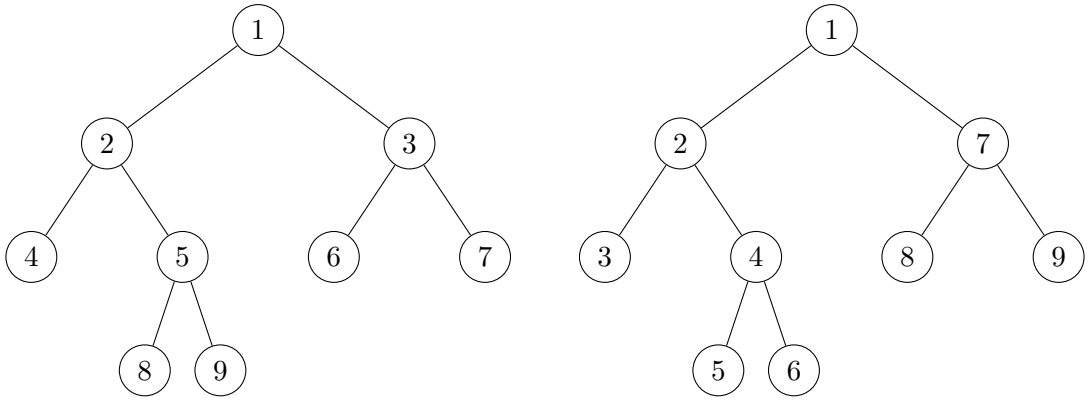


Abbildung 6.1: Breitensuche (links) und Tiefensuche (rechts) in einem Baum

Beispiel 6.3. Die Adjazenzliste des in Beispiel 6.1 angegebenen Graphen ist:

1: 2

2: 2,3,5

3: 1

4: 3

5: 4

Der für eine Adjazenzliste benötigte Speicherplatz ist $\Theta(|E|)$ was im schlechtesten Fall auch $\Theta(|V|^2)$ ist. Für dünn besetzte Graphen, d.h. also solche die wesentlich weniger als $|V|^2$ viele Kanten haben, stellt die Verwendung einer Adjazenzliste aber eine Speicherersparnis dar. Auf Basis einer Adjazenzliste kann, gegeben Knoten v und w, in Zeit $O(\mathrm{d}^+(v))$ festgestellt werden, ob der Graph die Kante (v, w) enthält.

Beispiel 6.4. Sei $G = (V, E)$ ein Graph wobei V die Menge der Kreuzungen in einer Großstadt ist und $(v, w) \in E$ falls eine Straße von v nach w führt ohne eine andere Kreuzung zu passieren. Der Graph G ist dünn besetzt, die Verwendung einer Adjazenzliste ist also empfehlenswert.

6.2 Breitensuche und Tiefensuche

Wenn wir einen Baum vollständig durchlaufen wollen, dann kann das unter anderem auf die folgenden beiden Arten bewerkstelligt werden: 1. Schicht für Schicht, indem wir also möglichst spät weiter absteigen und 2. indem wir möglichst früh weiter absteigen. Die erste Vorgehensweise wird auch als Breitensuche (engl. *breadth-first search (BFS)*) bezeichnet, die zweite als Tiefensuche (engl. *depth-first search (DFS)*), siehe Abbildung 6.1. Analoge Vorgehensweisen sind auch in beliebigen Graphen möglich.

Der Ansatz zur Breitensuche in einem beliebigen Graphen besteht darin, von einem Startknoten u ausgehend den Graphen in alle Richtungen zu durchlaufen wobei die Knoten Ebene für Ebene abgearbeitet werden. Alle neu entdeckten Knoten werden in einer Warteschlange zur späteren Bearbeitung eingereiht. Dabei führen wir eine Prozedur P auf allen gefundenen Knoten aus, siehe Algorithmus 25.

Wie bereits in Abschnitt 5.4 festgestellt, benötigen Einfügen und Entfernen jeweils $\Theta(1)$ Zeit. Jeder Knoten der von u aus erreichbar ist wird genau ein Mal in die Warteschlange eingereiht und genau ein Mal aus ihr entnommen, also finden insgesamt $O(|V|)$ viele Warteschlangen-

Algorithmus 25 Breitensuche (engl. *breadth-first search*)

```
Prozedur BREITENSUCHE(V, E, u)
    Sei bekannt neues Datenfeld der Länge |V|, überall mit falsch initialisiert
    Sei Q eine neue Warteschlange, leer initialisiert
    bekannt[u] := wahr
    HINZUFÜGEN(Q, u)
    Solange Q nicht leer
        v := ENTFERNEN(Q)
        P(v)
        Für jede Kante (v, w) ∈ E
            Falls bekannt(w) = falsch dann
                bekannt(v) = wahr
                HINZUFÜGEN(Q, w)
            Ende Falls
        Ende Für
    Ende Solange
Ende Prozedur
```

Operationen statt. Für jeden erreichten Knoten wird genau ein Mal die Liste seiner ausgehenden Kanten durchlaufen, somit wird also die innerste Schleife $O(|E|)$ mal durchlaufen. Insgesamt erhalten wir also die Laufzeit $O(|V| + |E|)$. Dieser Algorithmus ruft die Prozedur P auf allen von u aus erreichbaren Knoten auf. Knoten die von u aus nicht erreicht werden können werden nicht besucht.

Die Tiefensuche kann nun, bis auf die Reihenfolge der Knoten, auf die gleiche Weise organisiert werden. Die für die Tiefensuche gewünschte Reihenfolge erhalten wir, indem wir die Warteschlange (FIFO) durch einen Stapel (LIFO) ersetzen. Wie vorhin führen wir auch jetzt eine Prozedur P auf genau den von u aus erreichbaren Knoten aus, siehe Algorithmus 26.

Algorithmus 26 Tiefensuche (engl. *depth-first search*)

```
Prozedur TIEFENSUCHE(V, E, u)
    Sei bekannt neues Datenfeld der Länge |V|, überall mit falsch initialisiert
    Sei S ein neuer Stapel, leer initialisiert
    bekannt[u] := wahr
    HINZUFÜGEN(S, u)
    Solange S nicht leer
        v := ENTFERNEN(S)
        P(v)
        Für jede Kante (v, w) ∈ E
            Falls bekannt(w) = falsch dann
                bekannt(v) = wahr
                HINZUFÜGEN(S, w)
            Ende Falls
        Ende Für
    Ende Solange
Ende Prozedur
```

Alles was wir vorhin über die Laufzeit gesagt haben gilt auch hier. Die Tiefensuche hat also ebenfalls Laufzeit $O(|V| + |E|)$.

Beispiel 6.5. siehe `bsp.breitensuche.tiefensuche.pdf`.

Breitensuche und Tiefensuche kann nun für eine bestimmte Prozedur P, wie in Algorithmen 25 und 26 angegeben, direkt verwendet werden. Größere Bedeutung als diese Prozeduren im Wortlaut hat aber das Designprinzip, einen Graphen entlang seiner Kanten zu durchlaufen wobei problemspezifische zusätzliche Information (wie z.B. das *bekannt*-Datenfeld) mitgeführt wird. In diesem Sinn bilden die Breitensuche und Tiefensuche eine wichtige Grundlage für viele ähnliche Algorithmen. So kann zum Beispiel durch eine ganz einfache Modifikation etwa der Tiefensuche ein Algorithmus zur Detektion von Zyklen angegeben werden.

Definition 6.4. Ein ungerichteter Graph $G = (V, E)$ heißt *zusammenhängend* wenn es für alle $v, w \in V$ einen Pfad von v nach w gibt.

Definition 6.5. Ein gerichteter Zyklus ist ein Pfad der Form $v_1, v_2, \ldots, v_{k-1}, v_k = v_1$ mit $k \geq 2$ so dass für alle $i, j \in \{1, \ldots, k-1\}$ mit $i \neq j$ auch $v_i \neq v_j$ ist. Ein ungerichteter Zyklus ist ein gerichteter Zyklus $v_1, \ldots, v_k$ mit $k \geq 4$ so dass $v_k, \ldots, v_1$ ebenfalls ein gerichteter Zyklus ist.

In ungerichteten Graphen werden wir von nun an nur ungerichtete Zyklen betrachten. Nun kann man sich leicht überlegen dass ein zusammenhängender ungerichteter Graph G genau dann einen Zyklus enthält wenn die Breitensuche oder Tiefensuche einen Knoten erreicht, der bereits bekannt ist. Einen Algorithmus zur Detektion eines Zyklus in einem ungerichteten Graphen erhält man also durch Ersetzung des Körpers der innersten Schleife durch:

Falls *bekannt*(w) = **wahr dann**
  Antworte "Graph enthält Zyklus!"
sonst
  bekannt(v) = **wahr**
  HINZUFÜGEN(S, w)
Ende Falls

6.3 Topologisches Sortieren

Ein weiteres Beispiel für ein Problem das durch geeignetes Durchlaufen eines Graphen gelöst werden kann ist die Erstellung einer Linearisierung eines gerichteten Graphen (in der Literatur oft auch als topologisches Sortieren bezeichnet).

Definition 6.6. Sei $G = (V, E)$ und $n = |V|$. Eine *Linearisierung* von G ist eine Liste $v_1, \ldots, v_n$ so dass $i \neq j \Rightarrow v_i \neq v_j$ und $(v_i, v_j) \in E \Rightarrow i < j$.

Beispiel 6.6. Wenn wir in einem Binärbaum als gerichteten Graphen auffassen, indem die Kanten von oben nach unten orientiert werden, dann ist die Datenfeld-Darstellung einer Halde eine topologische Sortierung der Baumdarstellung der Halde.

Satz 6.1. *Sei $G = (V, E)$ endlich. Dann hat G eine Linearisierung genau dann wenn G zyklenfrei ist.*

Beweis. Angenommen G hat eine Linearisierung $v_1, \ldots, v_n$ und einen Zyklus. Dann muss der Zyklus die Form $v_{i_1}, \ldots, v_{i_{k-1}}, v_{i_k} = v_{i_1}$ haben. Damit gilt $i_1 < i_2 < \ldots i_k < i_1$, Widerspruch. Jeder Graph der eine Linearisierung hat muss also zyklenfrei sein.

Bevor wir die Gegenrichtung beweisen, beobachten wir dass ein endlicher zyklenfreier Graph einen Knoten v mit $\mathrm{d}^-(v) = 0$ enthält: dieser kann gefunden werden, indem wir mit einem beliebigen Knoten v_0 starten und eine beliebige eingehende Kante rückwärts gehen. Dieser

Schritt wird solange wiederholt bis wir an einem Knoten ohne eingehende Kanten angelangt sind. Dieser Prozess terminiert, da der Graph endlich und zyklenfrei ist.

Nun zeigen wir mit Induktion nach $|V|$ dass ein zyklenfreier Graph $G = (V, E)$ eine Linearisierung hat. Die Induktionsbasis $|V| = 0$ ist trivial. Für den Induktionsschritt sei $v \in V$ mit $\mathrm{d}^-(v) = 0$. Sei $G' = G - v$, also der Graph G aus dem v und alle Kanten die v enthalten entfernt worden sind. Dann hat G' nach Induktionshypothese eine Linearisierung $v_2, \ldots, v_n$. Nachdem es kein $u \in V \setminus \{v\}$ gibt mit $(u, v) \in E$, ist $v, v_2, \ldots, v_n$ eine Linearisierung von G. □

Wir wollen nun das folgende Berechnungsproblem betrachten:

Linearisierung (Topologisches Sortieren)
Eingabe: Ein endlicher zyklenfreier Graph $G = (V, E)$
Ausgabe: Eine Linearisierung von G

Als Anwendung kann man sich z.B. Folgendes vorstellen: die Knoten repräsentieren die Teilaufgaben eines Projekts, von Aufgabe A nach Aufgabe B wird eine Kante gesetzt falls A vor B erledigt werden muss. Gesucht ist dann eine Reihenfolge in der diese Aufgaben unter Beachtung ihrer Abhängigkeiten abgearbeitet werden können.

Der Beweis von Satz 6.1 suggeriert ja bereits die folgende algorithmische Vorgehensweise:

Wiederhole $|V|$ mal:

bestimme einen Knoten v mit $\mathrm{d}^-(v) = 0$, gib v aus und entferne v aus dem Graphen.

Die Bestimmung eines Knoten v mit $\mathrm{d}^-(v) = 0$ kann, wie im Beweis von Satz 6.1, so gemacht werden, dass von einem beliebigen Knoten u ausgehend die Kanten zurück verfolgt werden bis ein solcher Knoten v gefunden werden wird. Wie im Beweis von Satz 6.1 garantiert die Zyklenfreiheit des Graphen dass dieser Algorithmus terminiert. Diese Methode zur Bestimmung eines $v \in V$ mit $\mathrm{d}^-(v) = 0$ benötigt Zeit $O(|V|)$. Da sie $|V|$ mal wiederholt wird ist der Zeitbedarf dieses Algorithmus nur mit $O(|V|^2)$ abschätzbar.

Für dünn besetzte Graphen ist das nicht besonders gut. Eine bessere Vorgehensweise besteht darin, inspiriert von Breitensuche und Tiefensuche, den Graphen entlang seiner Kanten zu durchlaufen, wobei als nächste Knoten nur solche mit Eingangsgrad 0 (bezüglich des noch nicht durchlaufenen Teils) in Frage kommen. Um diese Knoten zu kennen wird statt dem *bekannt*-Datenfeld eine Datenfeld *Grad* mitgeführt in dem der Eingangsgrad aller Knoten bezüglich des noch nicht durchlaufenen Teils abgelegt (und auf dem aktuellen Stand gehalten) wird, siehe Algorithmus 27. Analog zur Laufzeitabschätzung der Breitensuche und Tiefensuche kann man sich auch hier leicht überlegen, dass die Laufzeit dieses Algorithmus $\Theta(|V| + |E|)$ ist.

Algorithmus 27 Linearisieren (Topologisches Sortieren)

```
Prozedur LINEARISIEREN(V, E)
    Sei Grad ein neues Datenfeld der Länge |V|, überall mit 0 initialisiert
    Für alle (v, w) ∈ E
        Grad[w] := Grad[w] + 1
    Ende Für
    Sei M neue Warteschlange oder Stapel, leer initialisiert
    Für alle v ∈ V
        Falls Grad[v] = 0 dann
            HINZUFÜGEN(M, v)
        Ende Falls
    Ende Für
    Solange M nicht leer
        v := ENTFERNEN(M)
        AUSGABE(v)
        Für alle (v, w) ∈ E
            Grad[w] := Grad[w] − 1
            Falls Grad[w] = 0 dann
                HINZUFÜGEN(M, w)
            Ende Falls
        Ende Für
    Ende Solange
Ende Prozedur
```

Kapitel 7

Gierige Algorithmen

Ein gieriger Algorithmus stellt die Lösung für ein Problem dadurch zusammen dass er zu jedem Zeitpunkt der Lösung jenen Teil hinzufügt der zu diesem Zeitpunkt am vielversprechendsten ist. Einmal gemachte Entscheidungen werden dabei nicht mehr revidiert. Auf diese Weise werden lokal optimale Bausteine zu einer Lösung zusammengefügt. Für gewisse Probleme wird dadurch eine global optimale Lösung erreicht. Wenn gierige Algorithmen auch nicht immer optimale Lösungen liefern, so haben sie doch den Vorteil dass sie schnell (im Sinne der Laufzeit) und einfach (im Sinne des Implementierungsaufwands) sind.

7.1 Minimale Spannbäume

Bevor wir ein erstes Beispiel für einen gierigen Algorithmus untersuchen, stellen wir noch einige graphentheoretische Vorüberlegungen an. In einem ungerichteten Graphen $G = (V, E)$ werden wir eine Kante zwischen zwei Knoten v und w einfach als $\{v, w\}$ notieren. Eine Kante von einem Knoten zu sich selbst wird dann einfach als $\{v\}$ geschrieben. Damit bezeichnet auch $|E|$ auf die Anzahl ungerichteter Kanten. Weiters definieren wir für $v \in V$ den Grad von v als $\mathrm{d}(v) = |\{w \in V \mid \{v, w\} \in E\}|$, es gibt als keinen getrennten Eingangs- und Ausgangsgrad mehr. Von nun an werden wir auch nur noch nicht-leere[1] endliche ungerichtete Graphen betrachten und diese Eigenschaften nicht mehr explizit erwähnen.

Definition 7.1. Sei $G = (V, E)$ ein Graph und $V' \subseteq V$. Der *von V' in G induzierte Graph* ist $G' = (V', E')$ wobei $E' = \{\{v, w\} \in E \mid v, w \in V'\}$.

Man beachte dass die obige Definition auch alle Kanten der Form $\{v\}$ einschließt indem $v = w$.

Definition 7.2. Sei $G = (V, E)$ ein Graph. $G' = (V', E')$ heißt *Zusammenhangskomponente* von G falls

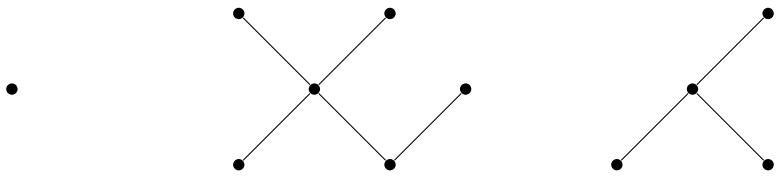
1. $V' \subseteq V$,
2. G' wird durch V' in G induziert,
3. G' ist zusammenhängend und
4. Für alle V'' mit $V \subset V'' \subseteq V$ gilt: der durch V'' in G induzierte Graph ist nicht zusammenhängend.

[1] d.h. solche mit mindestens einem Knoten

Aus dieser Definition folgt unmittelbar dass jeder ungerichtete Graph als disjunkte Vereinigung seiner Zusammenhangskomponenten dargestellt werden kann.

Definition 7.3. Ein Graph $G = (V, E)$ heißt *Baum* falls er zusammenhängend und zyklenfrei ist. Ein Graph $G = (V, E)$ heißt *Wald* falls er zyklenfrei ist.

Beispiel 7.1. Ein Wald der aus 3 Bäumen besteht:



In dieser Definition eines Baums ist keine Wurzel ausgezeichnet, jeder Knoten kann als Wurzel designiert werden (je nachdem ändert sich dann die Form des Baums). Bäume mit einem als Wurzel ausgezeichnetem Knoten bezeichnet man in diesem Kontext auch als *Wurzelbäume*. Zunächst machen wir einige Beobachtungen über Bäume.

Lemma 7.1. *Sei $G = (V, E)$ ein zyklenfreier Graph. Dann ist $|E| \leq |V| - 1$.*

Beweis. Wir gehen mit Induktion nach $|V|$ vor. Falls $|V| = 1$, dann muss $|E| = 0$ sein. Sei nun $|V| \geq 2$. Falls $E = \emptyset$ sind wir fertig. Falls $E \neq \emptyset$, dann existiert ein $v \in V$ mit $\mathrm{d}(v) \geq 1$. Seien $\{v, w_1\}, \ldots, \{v, w_k\}$ alle zu v inzidenten Kanten, dann gilt für alle $i, j \in \{1, \ldots, k\}$ mit $i \neq j$: jeder Pfad von w_i nach w_j enthält v, sonst würde nämlich G einen Zyklus enthalten. Für $i = 1, \ldots, k$ sei nun $G_i = (V_i, E_i)$ die Zusammenhangskomponente von w_i im Graphen der aus G entsteht wenn wir v sowie die Kanten $\{v, w_1\}, \ldots, \{v, w_k\}$ entfernen. Dann sind alle G_i zyklenfrei und mit der Induktionshypothese ist also

$$|E| = k + \sum_{i=1}^{k} |E_i| \leq^{\mathrm{IH}} k + \sum_{i=1}^{k} (|V_i| - 1) = \sum_{i=1}^{k} |V_i| = |V| - 1.$$

□

Lemma 7.2. *Sei $G = (V, E)$ ein zusammenhängender Graph. Dann ist $|E| \geq |V| - 1$.*

Beweis. Wir gehen mit Induktion nach $|V|$ vor. Falls $|V| = 1$ ist das Resultat trivial. Sei nun $|V| \geq 2$. Da G zusammenhängend ist existiert ein Knoten v mit $\mathrm{d}(v) \geq 1$. Seien $\{v, w_1\}, \ldots, \{v, w_k\}$ alle zu v inzidenten Kanten. Sei $G' = (V', E')$ der Graph der aus G entsteht wenn wir v und sowie $\{v, w_1\}, \ldots, \{v, w_k\}$ löschen. Dann zerfällt G' in $l \leq k$ Zusammenhangskomponenten $(V_1, E_1), \ldots, (V_l, E_l)$ und mit der Induktionshypothese gilt

$$|E| = k + \sum_{i=1}^{l} |E_i| \geq k + \sum_{i=1}^{l} (|V_i| - 1) \geq \sum_{i=1}^{l} |V_i| = |V| - 1.$$

□

Definition 7.4. Ein Pfad $v_1, \ldots, v_n$ heißt *einfach* falls $i \neq j \Rightarrow v_i \neq v_j$.

Definition 7.5. Ein Graph $G = (V, E)$ heißt *minimal zusammenhängend* falls er zusammenhängend ist und für alle $E' \subset E$ gilt: (V, E') ist nicht zusammenhängend.

Definition 7.6. Ein Graph $G = (V, E)$ heißt *maximal zyklenfrei* falls er zyklenfrei ist und für alle $E' \supset E$ gilt: (V, E') enthält einen Zyklus.

Satz 7.1. *Sei $G = (V, E)$ ein Graph. Dann sind äquivalent:*

1. *G ist ein Baum.*
2. *Je zwei Knoten von G sind durch genau einen einfachen Pfad verbunden.*
3. *G ist minimal zusammenhängend.*
4. *G ist zusammenhängend und $|E| = |V| - 1$.*
5. *G ist zyklenfrei und $|E| = |V| - 1$.*
6. *G ist maximal zyklenfrei.*

Beweis. 1. $\Rightarrow$ 2: Da G zusammenhängend ist, gibt es von u nach v einen Pfad und damit o.B.d.A. einen einfachen Pfad. Angenommen es gibt zwei unterschiedliche einfache Pfade von u nach v. Dann gibt es einen ersten Knoten w_1 an dem sie sich unterscheiden und einen letzten Knoten w_2 an dem sie sich unterschieden. Die beiden einfachen Pfade von w_1 nach w_2 bilden dann einen Zyklus.

2. $\Rightarrow$ 3.: Da je zwei Konten durch einen Pfad verbunden sind ist G zusammenhängend. Sei nun $(u, v) \in E$, dann ist u, v der einzige einfache Pfad von u nach v und damit ist $(V, E \setminus \{\{u, v\}\})$ nicht zusammenhängend.

3. $\Rightarrow$ 4.: G ist zusammenhängend und damit ist $|E| \geq |V| - 1$ wegen Lemma 7.2. Es bleibt zu zeigen dass $|E| \leq |V| - 1$ ist. Angenommen $|E| > |V| - 1$ dann würde G wegen Lemma 7.1 einen Zyklus enthalten. Aus diesem könnte eine beliebige Kante gelöscht werden ohne den Zusammenhang zu zerstören, G wäre also nicht minimal zusammenhängend.

4. $\Rightarrow$ 5.: Angenommen G enthält einen o.B.d.A. einfachen Zyklus $v_1, \ldots, v_k$. Für $l \in \{k, \ldots, |V|\}$ definieren wir einen Teilgraphen $G_l = (V_l, E_l)$ von G mit $|V_l| = |E_l| = l$ wie folgt: Falls $l = k$, dann ist besteht G_k aus dem Zyklus $v_1, \ldots, v_k$. Sei $l > k$. Da G zusammenhängend ist, existiert ein $\{v, w\} \in E$ so dass $v \in V_{l-1}$ und $w \notin V_{l-1}$. Wir definieren $V_l = V_{l-1} \cup \{w\}$ und $E_l = E_{l-1} \cup \{\{v, w\}\}$. Dann gilt $|V_l| = |E_l|$. Für $l = |V|$ erhalten wir also einen Teilgraphen $(V, E_{|V|})$ von (V, E). Damit ist $|E_V| = |V| \leq |E|$ was aber $|E| = |V| - 1$ widerspricht.

5.$\Rightarrow$ 6.: Angenommen es gäbe $E' \supset E$ so dass (V, E') zyklenfrei wäre, dann wäre wegen Lemma 7.1 ja $|E'| \leq |V| - 1$ was $|E'| > |E| = |V| - 1$ widerspricht.

6.$\Rightarrow$ 1.: Es reicht zu zeigen dass G zusammenhängend ist. Angenommen G wäre nicht zusammenhängend, seien dann v und w Knoten aus verschiedenen Zusammenhangskomponenten und $G' = (V, E \cup \{\{v, w\}\})$. Dann ist G' immer noch zyklenfrei, denn jeder Zyklus in G' wäre entweder Zyklus in G (Widerspruch) oder er würde die Kante $\{v, w\}$ mindestens zwei Mal enthalten woraus sich ein Zyklus in jeder der beiden Zusammenhangskomponenten bilden ließe (Widerspruch). G ist also nicht maximal zyklenfrei, Widerspruch. □

Korollar 7.1. *Ein Wald (V, E) besteht aus $|V| - |E|$ Bäumen.*

Beweis. Angenommen (V, E) besteht aus den m Bäumen $(V_1, E_1), \ldots, (V_m, E_m)$. Dann ist

$$|E| = \sum_{i=1}^{m} |E_i| =^{\text{Satz 7.1}} \sum_{i=1}^{m} (|V_i| - 1) = \sum_{i=1}^{m} |V_i| - m = |V| - m$$

und damit $m = |V| - |E|$. □

Wir kommen jetzt zu einer Anwendung von Bäumen. Angenommen wir wollen eine Menge V von Orten (z.B. Computern in einem Gebäude oder Pins auf einer Leiterplatte) verbinden. Die Knoten V bilden gemeinsam mit möglichen Verbindungen $E \subseteq V \times V$ einen ungerichteten Graphen. Zusätzlich sind die Kosten des Legens einer Verbindung bekannt, d.h. eine Kostenfunktion $c : E \to \mathbb{R}_{\geq 0}$ ist gegeben. Der Graph der Verbindungen soll also $G' = (V, E')$ sein wobei $E' \subseteq E$ ist, G' zusammenhängend ist und $\sum_{e \in E} c(e)$ minimal sein soll.

Korollar 7.2. *Sei $G = (V, E)$ ein zusammenhängender Graph und $E' \subseteq E$ so dass $G' = (V, E')$ minimal zusammenhängend ist. Dann ist G' ein Baum.*

Beweis. Folgt unmittelbar aus Satz 7.1. □

Der gesuchte Verbindungsgraph ist also ein Baum. Das motiviert die folgenden Definitionen und das folgende Berechnungsproblem.

Definition 7.7. Sei $G = (V, E)$ ein zusammenhängender Graph. Ein *Spannbaum* von G ist ein Baum $B = (V, E')$ mit $E' \subseteq E$.

Der Baum S spannt also G auf.

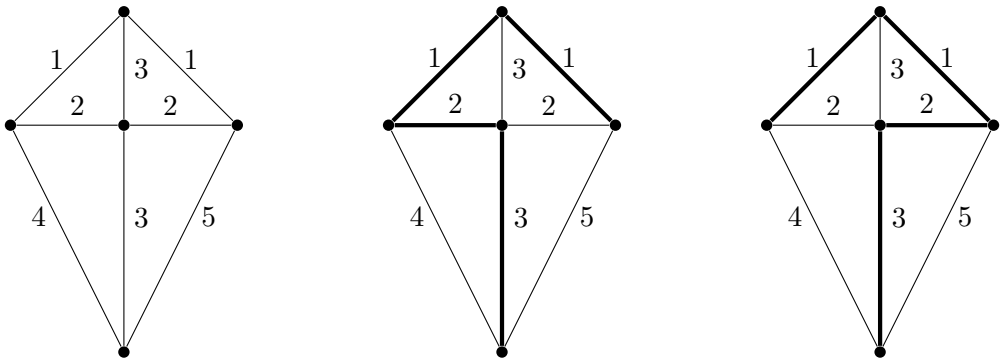
Definition 7.8. Sei $G = (V, E)$ ein zusammenhängender Graph und sei $c : E \to \mathbb{R}_{\geq 0}$. Ein *minimaler Spannbaum* von G bezüglich c ist ein Spannbaum $B = (V, E')$ von G so dass $\sum_{e \in E'} c(e)$ minimal ist unter aller Spannbäumen von G.

Minimaler Spannbaum
Eingabe: Ein zusammenhängender Graph G und eine Kostenfunktion $c : E \to \mathbb{R}_{\geq 0}$
Ausgabe: Ein minimaler Spannbaum von G bezüglich c

7.2 Der Algorithmus von Kruskal

Wir werden jetzt den Algorithmus von Kruskal zur Berechnung eines minimalen Spannbaums betrachten. Dieser Algorithmus ist leicht erklärt: Starte mit einer leeren Menge von Kanten B und wiederhole den folgenden Schritt so oft wie möglich: füge von den noch nicht betrachteten Kanten von G eine mit minimalen Kosten zu B hinzu falls dadurch kein Zyklus entsteht. Dieser Algorithmus vereinigt also sukzessive Bäume zu Bäumen, beginnen bei Knoten-Singletons bis daraus ein Spannbaum entsteht. Es handelt sich um einen gierigen Algorithmus da in jedem Schritt ein lokal optimales Element der Lösung (eine Kante minimalen Gewichts) hinzugefügt wird ohne sich darum zu kümmern, ob dadurch eine global optimale Lösung entsteht (ein minimaler Spannbaum).

Beispiel 7.2. Ein Graph mit Kantenkosten die zwei minimale Spannbäume erlauben:



Satz 7.2. *Der Algorithmus von Kruskal ist korrekt.*

Bevor wir diesen Satz zeigen können überlegen wir uns noch einige Eigenschaften von minimalen Spannbäumen.

Lemma 7.3. *Sei $G = (V, E)$ ein zusammenhängender Graph und $c : E \to \mathbb{R}_{\geq 0}$ injektiv. Sei $\emptyset \subset S \subset V$ und sei $e = (v, w)$ die Kante mit minimalen Kosten so dass $v \in S$ und $w \in V \setminus S$. Dann enthält jeder minimale Spannbaum von G die Kante e.*

Beweis. Sei B ein Spannbaum der e nicht enthält. Dann existiert ein einfacher Pfad $v_1, \ldots, v_k$ von v nach w in B. Da $v \in S$ und $w \in V \setminus S$ gibt es ein $i \in \{1, \ldots, k-1\}$ so dass $v_i \in S$ und $v_{i+1} \in V \setminus S$. Sei $e_0 = (v_i, v_{i+1})$, dann ist $c(e_0) > c(e)$. Sei $B' = (B \setminus e_0) \cup e$. Dann ist B' zusammenhängend, da B zusammenhängend ist und in jedem Pfad in B die Kante e_0 ersetzt werden kann durch $v_i, v_{i-1}, \ldots, v_1 = v, w = v_k, v_{k-1}, \ldots, v_{i+1}$. Außerdem hat B' genau $|V| - 1$ Knoten, ist also nach Satz 7.1 ein Baum und damit ein Spannbaum von G. Weiters ist $c(B') < c(B)$, also ist B kein minimaler Spannbaum. □

Beweis von Satz 7.2. Sei die Eingabe $G = (V, E)$ und $c : E \to \mathbb{R}_{\geq 0}$ injektiv. Der Algorithmus von Kruskal erzeugt einen zyklenfreien und zusammenhängenden Graphen $B = (V, E')$ mit $E' \subseteq E$ wobei die Zyklenfreiheit direkt überprüft wird und der Zusammenhang aus dem Vorgehen des Algorithmus folgt.

Es bleibt zu zeigen dass B ein minimaler Spannbaum ist. Sei dazu $e = (v, w)$ eine vom Algorithmus von Kruskal hinzugefügte Kante. Sei S die Zusammenhangskomponente von v unmittelbar vor dem Hinzufügen dieser Kante. Dann ist $w \notin S$ da sonst ein Zyklus entstehen würde. Weiters ist in der sortierten Kantenliste keine Kante zwischen S und $V \setminus S$ vor e vorgekommen, da sonst $w \in S$ wäre. Also ist e die kürzeste Kante zwischen S und $V \setminus S$ und muss deshalb nach Lemma 7.3 in einem minimalen Spannbaum vorkommen. Der Algorithmus von Kruskal fügt also nur solche Kanten zu seiner Ausgabe hinzu die in einem minimalen Spannbaum vorkommen müssen.

Falls $c : E \to \mathbb{R}_{\geq 0}$ nicht injektiv ist, sei $e_1, \ldots, e_n$ die im Algorithmus Verwendung findende Sortierung der Kanten in nichtfallender Reihenfolge. Wir betrachten eine geringfügig modifizierte Kostenfunktion $\hat{c} : E \to \mathbb{R}_{\geq 0}, e \mapsto c(e) + \delta(e)$ wobei δ so gewählt wird dass 1. $\hat{c}(e_i) < \hat{c}(e_{i+1})$ für $i = 1, \ldots, n-1$ und 2. $\delta(e)$ so klein ist dass jeder minimale Spannbaum von G bezüglich $\hat{c}$ auch minimaler Spannbaum von G bezüglich c ist. Dann liefert der Algorithmus von Kruskal einen minimalen Spannbaum bezüglich $\hat{c}$ und damit einen bezüglich c. □

Wir wollen jetzt noch einen zweiten, etwas eleganteren, Korrektheitsbeweis des Algorithmus von Kruskal betrachten. Der Algorithmus geht wie folgt vor: sei $e_1, \ldots, e_n$ eine Sortierung von

E so dass $c(e_1) \leq \cdots \leq c(e_n)$, sei $E_0 = \emptyset$ und

$$E_{i+1} = \begin{cases} E_i \cup \{e_{i+1}\} & \text{falls } (V, E_i \cup \{e_{i+1}\}) \text{ zyklenfrei ist und} \\ E_i & \text{sonst.} \end{cases}$$

Der Algorithmus antwortet mit (V, E_n).

2. Beweis von Satz 7.2. Zunächst ist klar dass $B = (V, E_n)$ zyklenfrei ist. B ist aber auch zusammenhängend: Angenommen B wäre nicht zusammenhängend, seien dann $v, w \in V$ in unterschiedlichen Zusammenhangskomponenten von B. Dann gibt es einen Pfad p von v nach w in G und eine Kante e_i auf p mit der die Zusammenhangskomponente von v verlassen wird und damit auch $e_i \notin E_n$. Dann ist aber $(V, E_n \cup \{e_i\})$ zyklenfrei und damit auch $(V, E_i \cup \{e_i\})$ zyklenfrei, also $e_i \in E_n$, Widerspruch. B ist also ein Spannbaum von G.

Für die Minimalität von B reicht es zu zeigen dass für alle $i \in \{0, \ldots, n\}$ ein minimaler Spannbaum von G mit Kantenmenge M_i existiert so dass $E_i \subseteq M_i$. Dann ist nämlich $E_n = M_n$ und damit B minimal. Wir gehen mit Induktion nach i vor. Der Fall $i = 0$ ist trivial. Für den Induktionsschritt definieren wir $M_{i+1} = M_i$ falls keine Kante hinzugefügt wird oder die hinzugefügte Kante $e_{i+1} \in M_i$ ist. Sei nun also $(V, E_i \cup \{e_{i+1}\})$ zyklenfrei und $e_{i+1} \notin M_i$. Dann enthält $(V, M_i \cup \{e_{i+1}\})$ einen Zyklus. Da aber (V, E_{i+1}) zyklenfrei ist, existiert ein e_j auf dem Zyklus mit $e_j \notin E_{i+1}$. Sei nun $M_{i+1} = (M_i \setminus \{e_j\}) \cup \{e_{i+1}\}$. Dann ist (V, M_{i+1}) zusammenhängend da (V, M_i) zusammenhängend ist und in jedem M_i-Pfad die Kante e_j durch e_{i+1} und den Rest des Zyklus ersetzt werden kann. Weiters ist $|M_{i+1}| = |M_i| = |V| - 1$ und (V, M_{i+1}) mit Satz 7.1 also ein Spannbaum von G. Außerdem ist $c(e_{i+1}) \leq c(e_j)$ denn $c(e_j) < c(e_{i+1})$ impliziert $j < i + 1$ sowie $(V, E_j \cup \{e_j\})$ zyklenfrei und damit $e_j \in E_i$. Also ist $c(M_{i+1}) \leq c(M_i)$ und, da M_i minimal ist, $c(M_{i+1}) = c(M_i)$ und M_{i+1} also ebenfalls ein minimaler Spannbaum. □

Um die Laufzeit des Algorithmus von Kruskal zu analysieren formulieren wir ihn zunächst im Detail in Form von Pseudocode aus. Es ist geschickt explizite Tests auf Zyklenfreiheit zu unterlassen da diese viel Zeit benötigen. Stattdessen werden wir die Zusammenhangskomponenten des Waldes mitführen. Der Algorithmus von Kruskal erzeugt einen Spannbaum durch sukzessive Vereinigung zweier Bäume in einem Wald zu einem neuen Baum durch Hinzufügen einer Kante. Er benötigt also eine Datenstruktur zur Darstellung einer Partition einer endlichen Menge, die Partition der Knoten in Zusammenhangskomponenten. Eine solche Datenstruktur zur Darstellung einer Partition einer endlichen Menge wird auch als *union-find*-Datenstruktur bezeichnet. Sie stellt (zumindest) die folgenden Operationen zur Verfügung:

1. INITIALISIEREN(M) erzeugt Datenstruktur für die Partition von M die aus Singleton-Klassen besteht.
2. VEREINIGEN(P, x_1, x_2) für zwei Elemente x_1 und x_2 von M.
3. FINDEN(P, x) gibt für $x \in M$ die Klasse C zurück die x enthält.

Auf Basis einer solchen Datenstruktur für eine Partition kann der Algorithmus von Kruskal dann wie in Algorithmus 28 ausformuliert werden. Ein einfache Implementierung der Partitionsdatenstruktur kann wie folgt erreicht werden. Sei $M = \{1, \ldots, n\}$. Wir verwenden auch zur Bezeichnung der Äquivalenzklassen Zahlen von 1 bis n. Wir arbeiten mit einem Datenfeld *ElementInKlasse* der Länge n wobei *ElementInKlasse*$[i]$ die Klasse ist in der sich das Element i befindet. Wir benötigen weiters ein Datenfeld *Kardinalität* der Länge n wobei *Kardinalität*$[i]$ die Kardinalität der Klasse i ist. Und schließlich verwenden wir noch ein weiteres Datenfeld

Algorithmus 28 Algorithmus von Kruskal

Prozedur KRUSKAL(V, E, c)
  $B := \emptyset$
  $P :=$ INITIALISIEREN(V)
  Für alle $(v, w) \in E$ in nichtfallender Reihenfolge
    Falls FINDEN$(P, v) \neq$ FINDEN(P, w) **dann**
      $B := B \cup \{\{v, w\}\}$
      VEREINIGEN(v, w)
    Ende Falls
  Ende Für
  Antworte (V, B)
Ende Prozedur

Elemente der Länge m wobei *Elemente*$[i]$ eine einfach verkettete Liste der Elemente der Klasse i ist.

Beispiel 7.3. Sei $M = \{1, \ldots, 8\}$. Nach Initialisierung und den Aufrufen VEREINIGEN$(P, 2, 4)$, VEREINIGEN$(P, 6, 8)$ und VEREINIGEN$(P, 1, 4)$ haben die Datenfelder den folgenden Inhalt:

ElementInKlasse:	$2, 2, 3, 2, 5, 6, 7, 6$
Kardinalität:	$0, 3, 1, 0, 1, 2, 1, 0$
Elemente:	NIL; 2, 4, 1; 3; NIL; 5; 6, 8; 7; NIL

Die Initialisierung benötigt Zeit $\Theta(n)$. Das Finden eines Elements geschieht direkt durch Nachsehen im Datenfeld *ElementInKlasse* und benötigt daher nur Zeit $\Theta(1)$. Für das Vereinigen der Klassen von x_1 und x_2 gehen wir wie folgt vor: zunächst seien C_1 die Klasse von x_1 und C_2 die Klasse von x_2. Dann kann über *Kardinalität* festgestellt werden welche der beiden Klassen die größere ist, sei o.B.d.A. C_1 die größere und C_2 die kleinere. Wir entfernen dann die Klasse C_2 und füge alle ihre Elemente der Klasse C_1 hinzu. Die dementsprechende Aktualisierung der Datenfelder benötigt Zeit $\Theta(|C_2|)$ da die zu ändernden Indices im Datenfeld *Elemente* zu finden sind.

Der Algorithmus von Kruskal führt $|V| - 1$ Vereinigungen aus. Jeder der Klassen hat höchstens die Größe $|V|$. Eine naive obere Schranke ist also $O(|V|^2)$ für die Laufzeit der Vereinigungsoperationen. Diese Schranke kann durch eine genauere Analyse allerdings verbessert werden. Sie ist nämlich insofern nicht realistisch als sie zwei miteinander inkompatible Annahmen über den schlechtesten Fall kombiniert, nämlich 1. dass viele Vereinigungen stattfinden und 2. dass diese Vereinigungen großer Mengen sind.

Lemma 7.4. *In der Datenfeld-Implementierung der Partitionsdatenstruktur benötigen k sukzessive Vereinigungsoperationen beginnend bei der Singleton-Partition höchstens $O(k \log k)$ Zeit.*

Beweis. In k (binären) Vereinigungsoperationen sind höchstens $2k$ Elemente von M involviert. Sei $i \in M$. Wir schreiben $C_j(i)$ für den Index der Klasse von i nach der j-ten Vereinigungsoperation und $|C_j(i)|$ für die Kardinalität dieser Klasse. Dann ist $|C_k(i)| \leq 2k$.[2] Außerdem gilt $C_{j+1}(i) \neq C_j(i) \Rightarrow |C_{j+1}(i)| \geq 2|C_j(i)|$ da ja bei jeder Vereinigungsoperation der Index der größeren Klasse behalten wird. Deshalb kann sich also die Klassenzugehörigkeit von i höchstens $\log 2k$ mal ändern. Da in k Vereinigungen höchstens $2k$ Elemente involviert sind, erhalten wir also für die gesamte Laufzeit $O(2k \log 2k) = O(k \log k)$. □

[2] Tatsächlich könnte man das noch genauer abschätzen. Das ist aber für dieses Resultat nicht nötig.

In einer Situation wie der des obigen Lemmas sagen wir auch dass eine einzelne von k Vereinigungsoperation *amortisierte Kosten* $\frac{1}{k}O(k \log k) = O(\log k)$ hat. Wir sprechen dann auch von *amortisierter Laufzeitanalyse*. Eine solche Vorgehensweise zur Abschätzung des schlechtesten Falls ist immer dann sinnvoll wenn mehrere Operationen durchgeführt werden von denen zwar eine einzelne recht teuer sein kann, die aber insgesamt im Durchschnitt nicht sehr teuer sind. In einer ähnlichen Situation waren wir auch bei der Abschätzung der Laufzeit der Erzeugung einer Halde aus einem Datenfeld im Kontext von Haldensortieren.

Die Gesamtlaufzeit des Algorithmus von Kruskal setzt sich also wie folgt zusammen: Wir benötigen $O(|V|)$ Zeit für die Initialisierung der Partitionsdatenstruktur, $O(|E| \log |E|)$ Zeit für das Sortieren der Kanten, $O(|E|)$ Schleifendurchläufe sowie $O(|V| \log |V|)$ für die $|V| - 1$ Vereinigungsoperationen. Da in einem zusammenhängenden Graphen $|E| \geq |V| - 1$ ist, erhalten wir insgesamt also $O(|E| \log |E|)$. Weiters ist ja $|E| \leq |V|^2$, also $\log |E| \leq 2 \log |V|$ und damit $\log |E| = O(\log |V|)$. Somit kann die Laufzeit des Algorithmus von Kruskal auch geschrieben werden als $O(|E| \log |V|)$.

7.3 Der Algorithmus von Prim

Wir betrachten jetzt noch einen zweiten, ebenfalls gierigen, Algorithmus zur Berechnung eines minimalen Spannbaums: den Algorithmus von Prim.

Definition 7.9. Sei $G = (V, E)$ und $c : E \to \mathbb{R}_{\geq 0}$, sei $G' = (V', E')$ ein Teilgraph von G, d.h. also $V' \subseteq V$ und $E' \subseteq E$, und sei $v \in V \setminus V'$. Dann sind die *Anschlusskosten* von v an G' bezüglich c

$$\min\{c(\{u, v\}) \mid \{u, v\} \in E, u \in V'\}$$

bzw. $+\infty$ falls diese Menge leer ist.

Der Algorithmus von Prim geht wie folgt vor: gegeben einen Graphen $G = (V, E)$ mit $|V| = n$ und eine Kostenfunktion $c : E \to \mathbb{R}_{\geq 0}$ setzen wir zu Beginn $V_1 = \{s\}$, $E_1 = \emptyset$. Für $i = 2, \ldots, n$ sei $v_i \in V \setminus V_i$ der Knoten mit minimalen Anschlusskosten an (V_i, E_i) und $\{u_i, v_i\}$ eine Kante die diese Anschlusskosten realisiert. Dann setzen wir $V_{i+1} = V_i \cup \{v_i\}$ und $E_{i+1} = E_i \cup \{\{u_i, v_i\}\}$. Der Algorithmus antwortet mit (V_n, E_n). Auf diese Weise verwalten wir einen wachsenden Baum der Teilgraph von G.

Satz 7.3. *Der Algorithmus von Prim ist korrekt.*

Beweis. Zunächst ist leicht zu beobachten dass alle $B_i = (V_i, E_i)$ Bäume sind. Da $V_n = V$ ist B_n ein Spannbaum von G. Für die Minimalität von B_n reicht es zu zeigen dass für alle $i \in \{1, \ldots, n\}$ ein minimaler Spannbaum von G mit Kantenmenge M_i existiert so dass $E_i \subseteq M_i$. Dann ist nämlich $M_n = E_n$ und damit B_n minimal. Wir gehen mit Induktion nach i vor. Der Fall $i = 1$ ist trivial. Für den Induktionsschritt machen wir eine Fallunterscheidung: Falls $\{u_i, v_i\} \in M_i$, sei $M_{i+1} = M_i$. Falls $\{u_i, v_i\} \notin M_i$, dann muss M_i eine Kante $\{w, w'\}$ enthalten mit $w \in V_i$ und $w' \notin V_i$ da M_i zusammenhängend ist. Dann ist $\{w, w'\} \neq \{u_i, v_i\}$ und da v_i minimale Anschlusskosten hat ist $c(\{u_i, v_i\}) \leq c(\{w, w'\})$. Wir definieren $M_{i+1} = (M_i \setminus \{\{w, w'\}\}) \cup \{\{u_i, v_i\}\}$. Nun ist (V_i, E_i) und G zusammenhängend, also gibt es einen Pfad p von w nach u_i und einen Pfad q von v_i nach w'. Somit kann also in jedem M_i-Pfad die Kante $\{w, w'\}$ durch $p, \{u_i, v_i\}, q$ ersetzt werden um daraus einen M_{i+1}-Pfad zu erhalten. M_{i+1} ist also zusammenhängend und da $|M_{i+1}| = |M_i| = |V| - 1$ ist M_{i+1} mit Satz 7.1 ein Spannbaum von G. Außerdem ist $c(M_{i+1}) \leq c(M_i)$ und, da M_i minimal ist, ist $c(M_{i+1}) = c(M_i)$ und damit (V, M_{i+1}) ein minimaler Spannbaum von G. □

Zur Implementierung des Algorithmus von Prim legen wir die Knoten in einer nach Minimum sortierten Prioritätswarteschlange ab. Für einen Knoten v ist $v.x$ die Priorität, also die minimalen Anschlusskosten bezüglich des aktuellen Baums. Wir führen ein Datenfeld B mit so dass $B[v] =$ **wahr** ist genau dann wenn der Knoten v im aktuellen Baum enthalten ist. Die Kanten des Baums werden implizit konstruiert, indem wir mit jedem Knoten v das Feld $v.\mathit{Vorgänger}$ mitführen das auf jenen Knoten zeigt mit dem die minimalen Anschlusskosten erreicht werden können. Bei Bedarf kann der Baum dann aus diesen Feldern explizit gemacht werden. Die Laufzeit des Algorithmus von Prim setzt sich wie folgt zusammen: Die Initialisierung in-

Algorithmus 29 Algorithmus von Prim

Prozedur PRIM(V, E, c, s)
  Sei B ein neues Datenfeld der Länge $|V|$, überall mit **falsch** initialisiert
  Für alle $v \in V$
    $v.x := \infty$
    $v.\mathit{Vorgänger} :=$ NIL
  Ende Für
  $s.x := 0$
  $Q :=$ MIN-PRIORITÄTSWARTESCHLANGE(V)
  Solange Q nicht leer
    $v :=$ EXTRAHIEREMINIMUM(Q)
    $B[v] :=$ **wahr**
    Für alle $(v, w) \in E$
      Falls $c(v, w) < w.x$ und $B[w] =$ **falsch dann**
        $w.\mathit{Vorgänger} := v$
        REDUZIEREPRIORITÄT$(Q, w, c(v, w))$
      Ende Falls
    Ende Für
  Ende Solange
  Antworte $(V, \{\{u, v\} \mid u = v.\mathit{Vorgänger}\})$
Ende Prozedur

klusive Aufbau der Prioritätswarteschlange benötigt Zeit $O(|V|)$, ebenso die Berechnung der Antwort. Die äußere Schleife wird $O(|V|)$ mal durchlaufen wobei die Extraktion des Minimums jeweils $O(\log |V|)$ Zeit benötigt. Die innere Schleife wird $|E|$ mal durchlaufen wobei bei jedem Durchlauf durch die Änderung der Priorität Kosten von $O(\log |V|)$ anfallen können. In einem zusammenhängenden Graphen ist $|E| = \Omega(|V|)$, insgesamt erhalten wir also eine Laufzeit von $O(|E| \log |V|)$.

7.4 Der Algorithmus von Dijkstra

Wir werden jetzt einen eng mit dem Algorithmus von Prim verwandten Algorithmus betrachten: den Algorithmus von Dijkstra. Er löst das Problem der Bestimmung des kürzesten Pfades.

Kürzester Pfad

Eingabe: zusammenhängender Graph $G = (V, E)$, Kostenfunktion $c : E \to \mathbb{R}_{\geq 0}$, $s, t \in V$

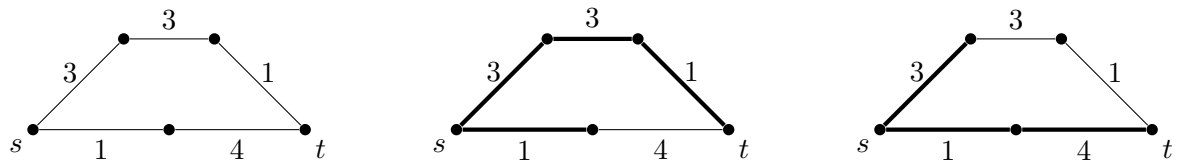
Ausgabe: Pfad $v_1, \ldots, v_n$ mit $v_1 = s$, $v_n = t$ so dass $\sum_{i=1}^{n-1} c(\{v_i, v_{i+1}\})$ minimal ist

Dieses Problem hat viele Anwendungen, z.B. die offensichtliche in Navigationssystemen. Wie beim Algorithmus von Prim werden wir sukzessive einen Teilgraphen B von G aufbauen der aus den minimalen Pfaden von s zu den Knoten von B besteht. Zur Verwaltung der Knoten verwenden wir eine Minimum-Prioritätswarteschlange. Der wesentliche Unterschied besteht darin, dass die Priorität $v.x$ eines Knoten v nicht mehr seine Anschlusskosten sind sondern seine B-Distanz von s, d.h. die Länge des kürzesten Pfades von s nach v der bis auf die letzte Kante nur aus Knoten in B besteht. Ebenso wie beim Algorithmus von Prim repräsentieren wir durch $v.\mathit{Vorgänger}$ implizit alle konstruierten Pfade. Der Algorithmus von Dijkstra berechnet die kürzesten Pfade von s zu jedem Knoten $v \in V$. Die Antwort für ein bestimmtes $t \in V$ kann daraus leicht in Zeit $O(|V|)$ bestimmt werden.

Algorithmus 30 Algorithmus von Dijkstra

Prozedur DIJKSTRA(V, E, c, s, t)
  Sei B ein neues Datenfeld der Länge $|V|$, überall mit **falsch** initialisiert
  Für alle $v \in V$
    $v.x := \infty$
    $v.\mathit{Vorgänger} :=$ NIL
  Ende Für
  $s.x := 0$
  $Q :=$ MIN-PRIORITÄTSWARTESCHLANGE(V)
  Solange Q nicht leer
    $v :=$ EXTRAHIEREMINIMUM(Q)
    $B[v] :=$ **wahr**
    Für alle $(v, w) \in E$
      Falls $v.x + c(v, w) < w.x$ und $B[w] =$ **falsch dann**
        $w.\mathit{Vorgänger} := v$
        REDUZIEREPRIORITÄT$(Q, w, v.x + c(v, w))$
      Ende Falls
    Ende Für
  Ende Solange
  Antworte $(t, t.\mathit{Vorgänger}, t.\mathit{Vorgänger}.\mathit{Vorgänger}, \ldots, s)^{-1}$
Ende Prozedur

Beispiel 7.4. Ein Graph G mit Kantenkosten (links), der vom Algorithmus von Prim bei Eingabe G erzeugte Spannbaum (Mitte) und die vom Algorithmus von Dijkstra erzeugten Pfade bei Besuch von t (rechts).



Satz 7.4. *Der Algorithmus von Dijkstra ist korrekt.*

Beweis. Sei $G = (V, E)$ sowie $c : E \to \mathbb{R}_{\geq 0}$ die Eingabe, sei $|V| = n$. Für $i = 1, \ldots, n$ sei S_i die Menge der nach dem i-ten Schleifendurchlauf abgearbeiteten, d.h. aus Q extrahierten, Knoten und für $v \in S_i$ sei p_v der durch die Vorgänger-Felder codierte Pfad von s nach v. Wir behaupten dass für alle $i = 1, \ldots, n$ und alle $v \in S_i$ gilt dass p_v ein kürzester Pfad von s nach v ist. Wir gehen mit Induktion nach i vor. Für $i = 1$ ist das trivial. Für den Induktionsschritt sei v der

Knoten in $S_{i+1} \setminus S_i$ und sei u der Vorgänger von v. Dann ist nach Induktionshypothese p_u ein kürzester Pfad von s nach u und $p_v = p_u, v$. Sei p ein anderer Pfad von s nach v, dann hat p die Form p_1, x, y, p_2 wobei p_1, x in S_i ist und y außerhalb von S_i. Da aber nach Definition des Algorithmus v unter allen mit einer Kante aus S_i erreichbaren Knoten in $V \setminus S_i$ minimale Distanz zu s hat, ist $c(p_1, x, y) \geq c(p_v)$ und damit $c(p) \geq c(p_v)$. □

Die Laufzeit vom Algorithmus von Dijkstra ist, wie jene vom Algorithmus von Prim, $O(|E| \log |V|)$.

7.5 Matroide

Definition 7.10. Sei E eine endliche Menge und $\mathcal{U} \subseteq \mathfrak{P}(E)$. $M = (E,\mathcal{U})$ heißt *Matroid* falls:

1. $\emptyset \in \mathcal{U}$,
2. $A \subseteq B \in \mathcal{U}$ impliziert $A \in \mathcal{U}$ und
3. es gilt die folgende *Austauscheigenschaft*:
   Ist $A, B \in \mathcal{U}$ mit $|A| > |B|$, dann gibt es $x \in A \setminus B$ so dass $B \cup \{x\} \in \mathcal{U}$.

Die Bezeichnung Austauscheigenschaft rührt daher, dass in der Menge $A = \{x\} \cup A'$ die Teilmenge A' durch B ausgetauscht werden kann ohne die Eigenschaft der Unabhängigkeit zu verlieren.

Beispiel 7.5. Sei M eine Matrix über einem Körper, E die Menge der Spaltenvektoren von M und $\mathcal{U}$ die Menge der linear unabhängigen Mengen von Spaltenvektoren von M. Dann bildet $(E,\mathcal{U})$ ein Matroid:

Klar ist dass $\emptyset$ linear unabhängig ist und dass jede Teilmenge einer linear unabhängigen Menge auch linear unabhängig ist, $(E,\mathcal{U})$ ist also ein Unabhängigkeitssystem. Für die Austauscheigenschaft seien A und B linear unabhängig und $|A| > |B|$, dann ist $\dim\langle A\rangle = |A| > |B| = \dim\langle B\rangle$. Angenommen für alle $x \in A$ wäre $B \cup \{x\}$ linear abhängig, dann wäre ja $\langle A \cup B\rangle = \langle B\rangle$ und damit $\dim\langle A \cup B\rangle = \dim\langle B\rangle < \dim\langle A\rangle$. Das widerspricht aber $\dim\langle A \cup B\rangle \geq \dim\langle A\rangle$.

Dieser Spezialfall aus der linearen Algebra hat auch historisch den Begriff des Matroids motiviert. Die Austauscheigenschaft ist eine Verallgemeinerung des Austauschsatzes von Steinitz. Der Terminologie der linearen Algebra folgend können wir nun definieren und beweisen:

Definition 7.11. Sei $(E,\mathcal{U})$ ein Matroid. Eine Menge $A \in \mathcal{U}$ heißt *Basis* falls kein A' existiert mit $A \subset A' \in \mathcal{U}$.

Da E endlich und $\mathcal{U}$ nicht leer ist hat jedes Matroid mindestens eine Basis.

Satz 7.5. *Sei $M = (E,\mathcal{U})$ ein Matroid, seien B_1, B_2 Basen von M, dann ist $|B_1| = |B_2|$.*

Beweis. Angenommen B_1 und B_2 wären Basen von M mit $|B_1| > |B_2|$, dann gäbe es nach der Austauscheigenschaft ein $x \in B_1 \setminus B_2$ so dass $B_2 \cup \{x\} \in \mathcal{U}$, also wäre B_2 nicht maximal unabhängig. □

Definition 7.12. Der *Rang* eines Matroids M ist die Kardinalität seiner Basen.

Beispiel 7.6. Sei $G = (V, E)$ ein zusammenhängender Graph. Wir bezeichnen $A \subseteq E$ als unabhängig falls (V, A) zyklenfrei, d.h. ein Wald, ist. Sei $\mathcal{U}$ die Menge aller unabhängigen Kantenmengen, dann ist $M_G = (E,\mathcal{U})$ ein Matroid:

Es ist nämlich $(V, \emptyset)$ zyklenfrei und falls (V, A) zyklenfrei ist und $B \subseteq A$, dann ist auch (V, B) zyklenfrei. Für die Austauscheigenschaft seien $A, B \in \mathcal{U}$ mit $|A| > |B|$. Nach Korollar 7.1 besteht der Wald (V, A) aus $|V| - |A|$ Bäumen und der Wald (V, B) aus $|V| - |B|$ Bäumen. Nach dem Schubfachprinzip gibt es also in (V, A) einen Baum T dessen Knoten zu mindestens zwei Bäumen in (V, B) gehören. Da T zusammenhängend ist, gibt es eine Kante $\{v, w\}$ in T so dass v und w zu verschiedenen Bäumen in (V, B) gehören. Damit ist $B \cup \{\{v, w\}\} \in \mathcal{U}$.

Die Basen von M_G sind dann die maximalen unabhängigen Mengen, d.h. die maximalen zyklenfreien Teilgraphen, d.h. nach Satz 7.1 die Spannbäume von V.

Definition 7.13. Sei $(E, \mathcal{M})$ ein Matroid und $g : E \to \mathbb{R}_{\geq 0}$, dann heißt $(E, \mathcal{M}, g)$ *gewichtetes Matroid*.

Wir können also das Problem der Bestimmung eines minimalen Spannbaums abstrahieren zur Bestimmung einer Basis minimalen Gewichts in einem gewichteten Matroid:

Basis minimalen Gewichts
Eingabe: gewichtetes Matroid $(E, \mathcal{M}, g)$
Ausgabe: Basis $B \in \mathcal{M}$ so dass $\sum_{b \in B} g(b)$ minimal

Dann besteht die korrespondiere Abstraktion des Algorithmus von Kruskal in dem folgenden generischen gierigen Algorithmus, siehe Algorithmus 31.

Algorithmus 31 Generischer gieriger Algorithmus

```
Prozedur GIERIG(E,U,g)
    Sortiere E[1,...,n] so dass g(E[1]) ≤ ··· ≤ g(E[n])
    A := ∅
    Für i := 1,...,n
        Falls A ∪ {E[i]} ∈ U dann
            A := A ∪ {E[i]}
        Ende Falls
    Ende Für
    Antworte A
Ende Prozedur
```

Beispiel 7.7. Der generische gierige Algorithmus angewandt auf das in Beispiel 7.6 diskutierte Matroid M_G entspricht genau dem Algorithmus von Kruskal.

Satz 7.6. *Sei $M = (E, \mathcal{U}, g)$ ein gewichtetes Matroid. Dann berechnet* GIERIG$(E, \mathcal{U}, g)$ *eine Basis minimalen Gewichts von M.*

Beweis. Sei $(E, \mathcal{U}, g)$ ein gewichtetes Matroid mit Rang r und seien $a_1, \ldots, a_r \in E$ die vom generischen gierigen Algorithmus ausgewählten Elemente in dieser Reihenfolge. Dann ist $g(a_1) \leq \cdots \leq g(a_r)$. Sei $\{b_1, \ldots, b_r\}$ eine Basis mit $g(b_1) \leq \cdots \leq g(b_r)$. Es reicht zu zeigen dass $g(a_i) \leq g(b_i)$ für alle $i \in \{1, \ldots, r\}$. Angenommen es gäbe ein $k \in \{1, \ldots, r\}$ so dass $g(a_k) > g(b_k)$, dann ist $k \geq 2$ da $a_1 \in E$ minimales Gewicht hat. Sei $A = \{a_1, \ldots, a_{k-1}\}$ und $B = \{b_1, \ldots, b_k\}$. Dann ist $|B| > |A|$ und damit gibt es nach der Austauscheigenschaft ein $b_j \in B \setminus A$ so dass $A \cup \{b_j\} \in \mathcal{U}$. Dann ist aber $g(b_j) \leq g(b_k) < g(a_k)$ und damit steht b_j strikt vor a_k in der Sortierung $e_1, \ldots, e_n$ von E. Der gierige Algorithmus hätte also b_j zu A hinzugefügt bevor er a_k betrachtet hätte, Widerspruch. □

Beispiel 7.8. Sei $G = (V, E)$ ein zusammenhängender Graph, $g : E \to \mathbb{R}_{\geq 0}$ und $s \in V$. Wir definieren dann P als die Menge aller bei s beginnenden zyklenfreien Pfade und erweitern g auf P durch $g(p) = \sum_{e \text{ auf } p} g(e)$. Eine Menge $A \subseteq P$ heißt unabhängig wenn die Pfade in A paarweise verschiedene Endknoten haben. Sei $\mathcal{U}$ die Menge der unabhängigen Pfadmengen, dann ist $M = (P, \mathcal{U}, g)$ ein gewichtetes Matroid.

Klar ist $\emptyset \in \mathcal{U}$ und $A \subseteq B \in \mathcal{U}$ impliziert $A \in \mathcal{U}$. Die Austauscheigenschaft gilt, da jede unabhängige Menge von $k + 1$ Pfaden einen Endknoten enthält der nicht Endknoten in einer unabhängigen Menge von k Pfaden ist.

Eine Basis von M ist eine Menge von Pfaden die alle Knoten erreichen, eine Basis minimalen Gewichts enthält für jeden Knoten v nur einen kürzesten Pfade von s nach v. Der Rang von M ist $|V|$. Der generische gierige Algorithmus entspricht dann in Ablauf und Ergebnis dem Algorithmus von Dijkstra.

7.6 Huffman-Codes

Definition 7.14. Ein Alphabet ist eine endliche Menge von Zeichen.

Oft bezeichnen wir Alphabete mit A. Ein Wort über A ist eine endliche Folge von Zeichen aus A inklusive des leeren Wortes ε. Wir schreiben A^* für die Menge der Wörter über A. Es ist leicht zu sehen, dass A^* mit der Verkettung von Wörtern und ε ein Monoid bildet, genauer: das von A frei erzeugte Monoid.

Definition 7.15. Sei A ein Alphabet. Ein *Code*[3] ist ein Monoidhomomorphismus $\varphi : A^* \to \{0, 1\}^*$ so dass für alle $x_1, \ldots, x_n, y_1, \ldots, y_m \in \varphi(A)$ gilt:

$$x_1, \ldots, x_n = y_1, \ldots, y_m \text{ impliziert } n = m \text{ und } x_i = y_i \text{ für } i = 1, \ldots, n.$$

Da, bis auf Permutation der Buchstaben des Alphabets, ein Code eindeutig durch $\varphi(A) = X \subseteq \{0, 1\}^*$ gegeben ist, wird in der Literatur oft auch einfach X als Code bezeichnet. Oft werden auch andere Codierungsalphabete als lediglich $\{0, 1\}$ zugelassen. Für unsere Zwecke sind die hier eingeführten Begriffe allerdings ausreichend.

Beispiel 7.9. Sei $A = \{a, b, c\}$, dann ist der Monoidhomomorphismus $\varphi : A^* \to \{0, 1\}^*$ eindeutig definiert durch die Angabe von $\varphi(a) = 00$, $\varphi(b) = 100$, $\varphi(c) = 10$. Wir wollen nun zeigen, dass φ ein Code ist. Angenommen es gäbe $x_1, \ldots, x_n, y_1, \ldots, y_m \in \varphi(A)$ mit $x_1 \cdots x_n = y_1 \cdots y_m$, dann ist o.B.d.A. $x_1 = 10$ und $y_1 = 100$, also beginnt x_2 mit 0, also $x_2 = 00$, also beginnt y_2 mit 0, also $y_2 = 00$, usw. Damit ist also für alle $k \geq 1$: $x_1 \cdots x_k 0 = y_1 \cdots y_k$. Das widerspricht der Darstellung $x_1 \cdots x_n = y_1 \cdots y_m$.

Auch wenn ein Code wie der obige eine eindeutige Darstellung des codierten Wortes bietet, so ist er doch für die Praxis denkbar ungeeignet, da bei Übertragung einer Bitfolge der Form $10000 \cdots$ bis zum nächsten 1er oder bis zum Ende der Folge gewartet werden muss um zu entscheiden ob das Wort $baa \cdots$ oder $caa \cdots$ lautet.

In der Praxis will man es also typischerweise erkennen wenn ein Codewort zu Ende ist. Eine einfache Lösung dazu besteht in der Verwendung eines Codes dessen Wörter konstante Länge haben, d.h. eines $\varphi : A^* \to \{0, 1\}^*$ so dass ein $k \geq 1$ existiert so dass für alle $x \in A$: $\varphi(x) = k$. Klassische Beispiele dafür sind etwa ASCII (American Standard Code for Information Interchange) mit $k = 7$ oder die Erweiterung ISO 8859-1 von ASCII mit $k = 8$.

[3] Achtung, der Begriff Code hat nichts zu tun mit der Verschlüsselung von Daten.

Codes mit Wörtern fester Länge haben allerdings den Nachteil, dass sie zur Kompression von Daten nicht geeignet sind, da jedes Wort der Länge l durch genau $k \cdot l$ Bits codiert wird. Bei der Datenkompression will man die Redundanz von Wörtern ausnutzen um sie auf möglichst kompakte Weise darzustellen. Das kann geschehen indem häufig vorkommende Zeichen durch kurze Codewörter dargestellt werden und weniger häufig vorkommende durch längere. Für ein zu codierendes Wort $w \in A^*$ gilt ja

$$|\varphi(w)| = \sum_{x \in A} n_x(w)|\varphi(x)|.$$

Wenn also die $\varphi(x)$ geschickt gewählt werden, kann $|\varphi(w)|$ reduziert bzw. minimiert werden. Eine nützliche Vorgehensweise um das zu ermöglichen und gleichzeitig die Erkennbarkeit des Endes eines Codesworts zu garantieren besteht in der Verwendung von (sogenannten) Präfixcodes.

Definition 7.16. $u \in \{0,1\}^*$ heißt *Präfix* von $v \in \{0,1\}^*$ falls ein $w \in \{0,1\}^*$ existiert mit $uw = v$. Wir schreiben dann auch $u \leq v$.

Es ist leicht zu sehen dass $(\{0,1\}^*, \leq)$ eine Partialordnung ist.

Definition 7.17. Sei A ein Alphabet. Ein *Präfixcode* ist ein Code $\varphi : A^* \to \{0,1\}^*$ so dass für alle $x, y \in A$ gilt: $\varphi(x) \not\leq \varphi(y)$.

Der in Beispiel 7.9 behandelte Code ist kein Präfixcode. In einem Präfixcode kann, aufgrund der Präfixfreiheit, jedes Zeichen des ursprünglichen Alphabets erkannt werden sobald sein Codewort vollständig übertragen wird.

Beispiel 7.10. Sei $A = \{a, b, c, d, e\}$ und $w \in A^*$ mit

$$n_a(w) = 3, n_b(w) = 12, n_c(w) = 10, n_d(w) = 5, n_e(w) = 14.$$

Ein Code φ mit Codewörtern fester Länge benötigt 3 Bits zur Codierung von 5 Buchstaben und hat damit $|\varphi(w)| = (3 + 12 + 10 + 5 + 14) \cdot 3 = 132$. Der Code

$$\psi(a) = 000, \psi(b) = 10, \psi(c) = 01, \psi(d) = 001, \psi(e) = 11$$

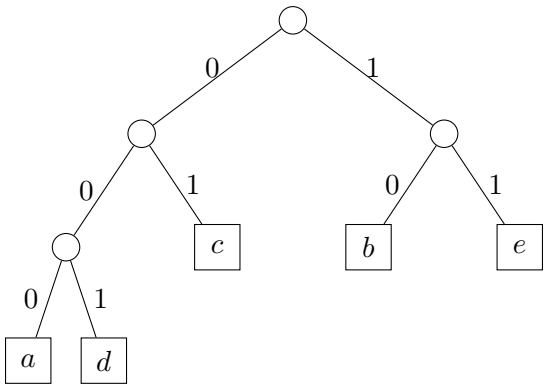
ist ein Präfixcode und benötigt zur Codierung von w lediglich $|\psi(w)| = 3(3+5) + 2 \cdot (10 + 12 + 14) = 96$ Bits.

Wir wollen also das folgende Problem lösen:

Optimaler Präfixcode
Eingabe: ein Alphabet A und ein Wort $w \in A^*$
Ausgabe: Präfixcode $\varphi : A^* \to \{0,1\}^*$ so dass $\|\varphi(w)\|$ minimal ist

Ein Präfixcode kann als binärer Baum dargestellt werden in dem die Blätter die Buchstaben enthalten und der Pfad von der Wurzel zu einem Blatt das diesem Buchstaben zugeordnete Codewort. Das ist auch zur Decodierung praktisch, da jedes empfangene Bit einem Abwärtsschritt im Baum entspricht und bei Erreichen eines Blatts ein Buchstabe ausgegeben und zur Wurzel zurückgesprungen wird.

Beispiel 7.11. Der in Beispiel 7.10 angegebenen Präfixcode ψ wird als Baum wie folgt dargestellt:



Über den Baum eines optimalen Präfixcodes kann man nun leicht die folgenden Beobachtungen machen:

1. Es handelt sich um einen vollständigen binären Baum (sonst könnte nämlich die Länge mind. eines Codeworts strikt verkürzt werden).
2. Innerhalb einer einzelnen Schicht des binären Baums können beliebige Permutationen durchgeführt werden können ohne die Optimalitätseigenschaft zu verlieren.
3. Die Buchstaben müssen in aufsteigender Häufigkeit von unten nach oben im Baum eingetragen werden.

D.h. es bleibt lediglich die Form des Baums zu bestimmen. Eine gierige Vorgehensweise dazu besteht darin, bei Eingabealphabet A und Eingabewort w wie folgt vorzugehen: zunächst werden die beiden Buchstaben $x, y \in A$ mit geringster Häufigkeit in w bestimmt. Diese werden als Blätter fixiert. Danach wird $A' = (A \setminus \{x, y\}) \cup \{[xy]\}$ gesetzt wobei $[xy]$ ein neuer Buchstabe ist. Weiters wird w' aus w erzeugt indem sowohl x als auch y durch $[xy]$ ersetzt wird. Der Buchstabe $[xy]$ wird mit dem inneren Knoten identifiziert der die Blätter x und y verbindet. Die Prozedur wird nun rekursiv für A' und w' aufgerufen. Auf diese Weise wird ein Baum von den Blättern zur Wurzel hin aufgebaut der einen Präfixcode repräsentiert. Die explizite Berechnung von A' und w' kann man sich dabei ersparen, wesentlich ist nur die Häufigkeit von $[xy]$ in w' die ja die Summe der Häufigkeiten von x und y in w ist. Diese Vorgehensweise wird auch als Algorithmus von Huffman bezeichnet, die dabei entstehenden Präfixcodes als Huffman-codes. Zur Verwaltung des sich ändernden Alphabets verwenden wir eine Minimum-Prioritätswarteschlange die nach der Häufigkeit der Buchstaben sortiert ist. Der entsprechende Pseudocode ist in Algorithmus 32 zu finden. Wie man sich durch schnelles Nachrechnen leicht überzeugen kann, erzeugt der Algorithmus von Huffman für A und w wie in Beispiel 7.10 den ebenda angegebenen Präfixcode ψ. Die Laufzeit dieser Prozedur ist $O(|w| + |A| \log |A|)$.

Lemma 7.5. *Sei A ein Alphabet, $w \in A^*$, seien $a, b \in A$ die beiden Buchstaben mit minimaler Häufigkeit in w, sei $A' = (A \setminus \{a, b\}) \cup \{[ab]\}$ und w' gleich w nach Ersetzung von a und b durch $[ab]$. Sei $\varphi : A^* \to \{0, 1\}^*$ ein Präfixcode mit $\varphi(a) = w0$ und $\varphi(b) = w1$ und sei*

$$\varphi' : A'^* \to \{0, 1\}, x \mapsto \begin{cases} w & \text{falls } x = [ab] \\ \varphi(x) & \text{sonst} \end{cases}.$$

Dann ist $|\varphi'(w')| = |\varphi(w)| - n_{[ab]}(w')$.

Beweis. Wir haben

$$|\varphi(w)| = \sum_{x \in A \setminus \{a,b\}} n_x(w)|\varphi(x)| + n_a(w)|\varphi(a)| + n_b(w)|\varphi(b)|,$$

Algorithmus 32 Algorithmus von Huffman

Vorbedingung: $|A| \geq 2$

Prozedur HUFFMAN(A, w)
  Für alle $a \in A$ sei v_a ein neuer Baumknoten mit $v_a.x = n_a(w)$
  $Q :=$ erzeuge min-Prioritätswarteschlange mit Inhalt $\{v_a \mid a \in A\}$
  Solange $|Q| \geq 2$
    Sei v ein neuer Baumknoten
    $v.links :=$ EXTRAHIEREMIN(Q)
    $v.rechts :=$ EXTRAHIEREMIN(Q)
    $v.x := v.links.x + v.rechts.x$
    HINZUFÜGEN(Q, v)
  Ende Solange
  Antworte MINIMUM(Q)
Ende Prozedur

da aber $|\varphi(a)| = |\varphi(b)| = |\varphi'([ab])| + 1$ und $n_a(w) + n_b(w) = n_{[ab]}(w')$ ist

$$\begin{aligned} &= \sum_{x \in A \setminus \{a,b\}} n_x(w)|\varphi(x)| + n_{[ab]}(w')(|\varphi'([ab])| + 1) \\ &= \sum_{x \in A'} n_x(w')|\varphi'(x)| + n_{[ab]}(w') \\ &= |\varphi'(w')| + n_{[ab]}(w'). \end{aligned}$$

□

Satz 7.7. *Der Algorithmus von Huffman erzeugt einen optimalen Präfixcode.*

Beweis. Wir gehen mit Induktion auf der Größe des Alphabets vor. Der Fall $|A| = 2$ ist trivial. Für den Induktionsschritt sei φ der vom Algorithmus erzeugte Präfixcode und sei ψ ein optimaler Präfixcode. Dann erfüllen φ und o.B.d.A. auch ψ die Bedingungen von Lemma 7.5 und damit erfüllen die ebenda definierten φ' und ψ'

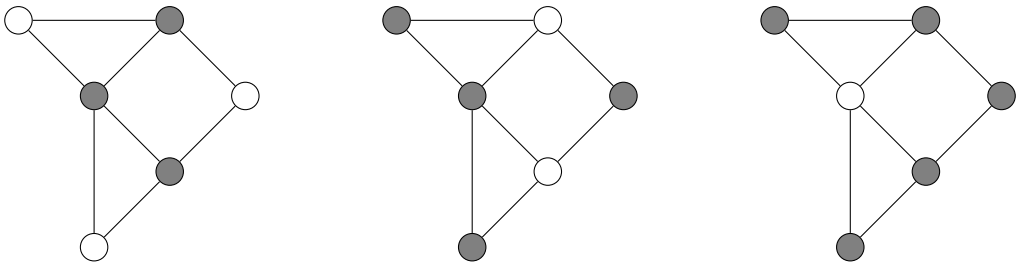
$$\begin{aligned} |\varphi'(w')| &= |\varphi(w)| - n_{[ab]}(w') \text{ und} \\ |\psi'(w')| &= |\psi(w)| - n_{[ab]}(w'). \end{aligned}$$

Da nach Induktionshypothese $|\varphi'(w)| = |\psi'(w)|$, ist also $|\varphi(w)| = |\psi(w)|$. □

7.7 Das Knotenüberdeckungsproblem

Definition 7.18. Sei $G = (V, E)$ ein Graph. Eine *Knotenüberdeckung* von V ist eine Menge $V' \subseteq V$ so dass $\{u, v\} \in E \Rightarrow u \in V'$ oder $v \in V'$.

Beispiel 7.12. Ein Graph und drei seiner Knotenüberdeckungen (in grau):



Wie man sich leicht überlegen kann hat dieser Graph keine Knotenüberdeckung der Größe 2.

Jeder Graph hat die triviale Knotenüberdeckung $V = V'$. Wenn wir nach einer optimalen Knotenüberdeckung fragen, d.h. eine minimaler Kardinalität, dann erhalten wir das folgende Berechnungsproblem:

Optimale Knotenüberdeckung
Eingabe: ungerichteter Graph $G = (V, E)$
Ausgabe: Knotenüberdeckung V' von G so dass $\|V'\|$ minimal ist

Ein Berechnungsproblem bei dem, wie oben, die gewünschte Ausgabe aus einer größeren Menge zulässiger Lösungen, z.B. der Knotenüberdeckungen, durch eine Minimalitäts- oder eine Maximalitätsbedingung definiert wird, bezeichnet man auch als *Optimierungsproblem*. Für viele Anwendungsfälle ist es allerdings nicht unbedingt notwendig eine optimale Lösung zu finden, stattdessen reicht oft auch eine die nahe an die optimale Lösung herankommt. Algorithmen die solche Lösungen berechnen bezeichnet man als *Approximationsalgorithmen*. Natürlich gibt es oft triviale Approximationsalgorithmen, z.B. im Fall der Knotenüberdeckung einfach V zu verwenden. Deshalb interessiert man sich im Kontext von Approximationsalgorithmen oft für Approximationsgarantien.

Definition 7.19. Wir sagen dass ein Algorithmus für ein Minimierungsproblem *Approximationsrate* $\rho(n)$ hat falls für jede Eingabe x der Größe n, für die vom Algorithmus gelieferte Ausgabe mit Kosten c und die optimalen Kosten c^* bei Eingabe x gilt dass $\frac{c}{c^*} \leq \rho(n)$.

Mit Kosten in der obigen Definition ist die Güte der Lösung gemeint, also z.B. im Fall der Knotenüberdeckung die Anzahl der Knoten in der Überdeckung. Die Frage der Laufzeit eines Algorithmus ist unabhängig von der Frage nach seiner Approximationsrate. Ein Algorithmus mit Approximationsrate $\rho(n)$ wird oft auch als $\rho(n)$-*Approximationsalgorithmus* bezeichnet. Oft ist es möglich, für Optimierungsprobleme, auch wenn kein polynomialer Algorithmus zu ihrer exakten Lösung bekannt ist, einen polynomialen Approximationsalgorithmus zu finden, gelegentlich sogar mit konstanter, d.h. nicht von n abhängiger, Approximationsrate. Gierige Algorithmen eignen sich dafür besonders gut, da sie immer eine geringe Laufzeit haben. Betrachten wir zum Beispiel Algorithmus 33 zur Berechnung einer Knotenüberdeckung. Dieser Algorithmus berechnet keine optimale Knotenüberdeckung. Das sieht man allein schon daran, dass er nur Knotenüberdeckungen gerader Kardinalität berechnet, es gibt aber Graphen deren optimale Knotenüberdeckung ungerade Kardinalität hat, siehe Beispiel 7.12. Dieser Algorithmus hat Laufzeit $O(|E|)$.

Satz 7.8. *Die Prozedur* GierigeKnotenüberdeckung *ist ein* 2*-Approximationsalgorithmus.*

Beweis. Sei $G = (V, E)$ der Eingabegraph, sei $V^* \subseteq V$ eine optimale Knotenüberdeckung, sei V' die Ausgabe des Algorithmus und sei E' die Menge der aus A gewählten Kanten, dann ist

Algorithmus 33 Gierige Knotenüberdeckung

Prozedur GierigeKnotenüberdeckung(V, E)
  $U := \emptyset$
  $A := E$
  Solange A nicht leer
    Sei $\{u, v\} \in A$
    $U := U \cup \{u, v\}$
    Entferne alle e aus A die u oder v enthalten
  Ende Solange
  Antworte U
Ende Prozedur

$|V'| = 2|E'|$. Nun gibt es nach der Definition des Algorithmus keine zwei Kanten in E' die einen Knoten teilen. Also muss jede Knotenüberdeckung für jede Kante $e \in E'$ mindestens einen Konten enthalten, um e zu überdecken. Also ist $|V^*| \geq |E'| = \frac{1}{2}|V'|$ und damit $\frac{|V'|}{|V^*|} \leq 2$. □

Es ist nicht bekannt ob das Problem der optimalen Knotenüberdeckung durch einen Algorithmus mit polynomialer Laufzeit (exakt) gelöst werden kann. An sich kann das schon Grund genug sein, sich einen Approximationsalgorithmus zu überlegen. In diesem (und vielen vergleichbaren Fällen) weiß man allerdings sogar dass die Existenz eines solchen Algorithmus implizieren würde dass $\mathbf{P} = \mathbf{NP}$. Die Frage ob $\mathbf{P} = \mathbf{NP}$ ist eines der schwierigsten offenen Probleme der Mathematik.

Um das genauer zu erklären, muss kurz ausgeholt werden. Ein Entscheidungsproblem ist ein Berechnungsproblem dessen Ausgabe entweder "ja" oder "nein" ist. Berechnungsprobleme und insbesondere auch Optimierungsprobleme können üblicherweise recht direkt in Entscheidungsprobleme übertragen werden, z.B.

Knotenüberdeckung
Eingabe: ungerichteter Graph $G = (V, E)$, $k \in \mathbb{N}$
Ausgabe: "ja" wenn G eine Knotenüberdeckung V' hat mit $\lvert V'\rvert \leq k$ und "nein" sonst

Ein Entscheidungsproblem wird als Menge betrachtet indem es mit der Menge seiner "ja"-Instanzen identifiziert wird. Wir schränken unsere Betrachtung nun ein auf Entscheidungsprobleme die Teilmengen von $\{0, 1\}^*$ sind. Graphen, natürliche Zahlen, Tupel von Graphen und natürlichen Zahlen, etc. können als Elemente eines Entscheidungsproblems betrachtet werden indem eine Kodierung in Wörter über $\{0, 1\}$ fixiert wird und der Graph, die Zahl, etc. mit seiner/ihrer Codierung identifiziert wird.

P bezeichnet nun die Menge aller Entscheidungsprobleme für die ein Algorithmus existiert dessen Laufzeit polynomial in der Größe der Eingabe ist[4]. Ein Entscheidungsproblem S ist[5] in **NP** genau dann wenn es ein Entscheidungsproblem $R \in \mathbf{P}$ sowie ein $c \in \mathbb{N}$ gibt so dass

$$x \in S \Leftrightarrow \exists y \in \{0, 1\}^* \text{ mit } |y| \leq |x|^c \text{ und } (x, y) \in R.$$

[4]Diese Definition bleibt insofern informell als wir im Rahmen dieser Vorlesung nicht formell definieren was ein Algorithmus ist. Für die Definition von **P** hat das aber kaum Bedeutung da sich unterschiedliche derartige formale Definition modulo polynomialer Berechenbarkeit nicht unterscheiden.

[5]**NP** steht für "lösbar in **n**icht-deterministisch **p**polynomialer Zeit" nach einer anderen Definition von **NP** über nicht-deterministische Berechnung

An der Definition des Entscheidungsproblems der Knotenüberdeckung ist jetzt unmittelbar sichtbar, dass es in **NP** ist indem wir für R die folgende Relation wählen:

$$((G,k),V') \in R \Leftrightarrow V' \text{ ist Knotenüberdeckung von } G \text{ mit } |V'| \leq k$$

und beobachten dass $R \in \mathbf{P}$. Überdies kann vom Entscheidungsproblem der Knotenüberdeckung (mit etwas mehr technischem Aufwand) gezeigt werden, dass es **NP**-vollständig ist. **NP**-vollständige Probleme sind in einem gewissen technischen Sinn die schwierigsten Probleme in **NP**. Für ein beliebiges **NP**-vollständiges Problem S gilt dass $S \in \mathbf{P} \Leftrightarrow \mathbf{P} = \mathbf{NP}$.

Wenn nun das Optimierungsproblem der optimalen Knotenüberdeckung einen polynomialen Algorithmus besitzt, dann hat auch das Entscheidungsproblem einen polynomialen Algorithmus und, da dieses **NP**-vollständig ist, wäre dann $\mathbf{P} = \mathbf{NP}$.

Es gibt tausende **NP**-vollständige Probleme von denen viele praxisrelevant sind. Die Bedeutung der Frage ob $\mathbf{P} = \mathbf{NP}$ rührt daher dass die (meisten) Probleme in **P**in der Praxis effizient lösbar sind und viele praxisrelevante Probleme **NP**-vollständig sind.

Kapitel 8

Dynamische Programmierung

In Kapitel 3 haben wir Teile-und-herrsche Algorithmen kennengelernt. Diese gehen so vor, dass eine Instanz eines Problems in kleinere, disjunkte, Teilinstanzen zerlegen, diese unabhängig voneinander lösen und deren Lösungen dann zu einer der ursprünglichen Instanz kombinieren. Solche Algorithmen sind nicht effizient, wenn im Laufe der Berechnung die selben Teilinstanzen oft auftreten, dann werden diese nämlich unabhängig voneinander mehrfach gelöst. *Dynamische Programmierung* ist ein Designprinzip für Algorithmen, das solche mehrfach auftretenden Teilinstanzen auf eine Weise organisiert die doppelte Berechnungen vermeidet. Die Teilinstanzen werden dann typischerweise beginnend bei den kleineren gelöst und so wird eine Lösung für die ursprüngliche Instanz berechnet. Damit das effizient möglich ist, ist es notwendig dass die Anzahl der zur Bestimmung einer Lösung notwendigen Teilinstanzen nicht allzu groß ist (also z.B. nur polynomial).

Dieses Prinzip kann an der Berechnung der Fibonacci-Zahlen gut illustriert werden. Diese sind ja bekannterweise definiert als

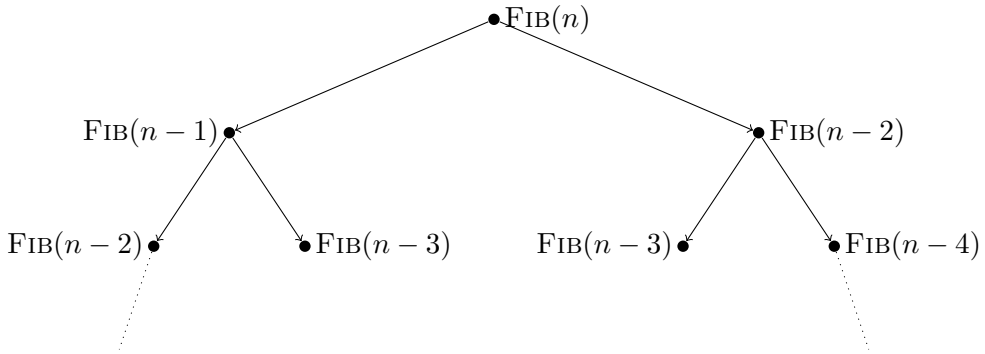
$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}.$$

Wir haben zwar in Bemerkung 4.1 schon gesehen, dass ein Algorithmus existiert, der F_n in Zeit $O(\log n)$ berechnet. Wir wollen aber zur Illustration des Prinzips der dynamischen Programmierung noch andere, ineffizientere, Algorithmen zur Berechnung von F_n betrachten. Wenn die obige rekursive Definition auf naive Weise algorithmisch umgesetzt wird erhalten wir Algorithmus 34. Diese Prozedur hat Laufzeit $\Theta(2^n)$ da sie für jedes n zwei unabhängige Aufrufe von

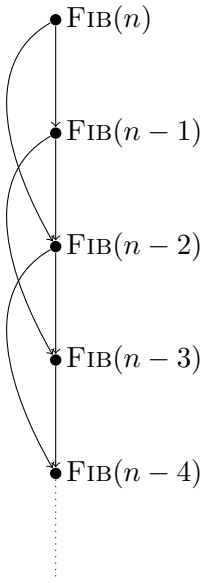
Algorithmus 34 Berechnung der n-ten Fibonacci-Zahl (exponentiell)

```
Prozedur FIB(n)
    Falls n = 0 dann
        Antworte 0
    sonst falls n = 1 dann
        Antworte 1
    sonst
        Antworte FIB(n − 1) + FIB(n − 2)
    Ende Falls
Ende Prozedur
```

sich selbst startet. Das ist offensichtlich nicht geschickt, da dadurch F_i für viele $i < n$ sehr oft berechnet wird. Der Rekursionsbaum des Aufrufs FIB(n) hat die Form



Die entscheidende Beobachtung ist nun dass eine mehrfache Berechnung des selben Wertes vermieden werden kann wenn wir statt einem Baum der Berechnung einen gerichteten azyklischen Graphen zugrunde legen. Diesen bezeichnet man auch als *Teilproblemgraph* und er hat für Fib(n) die Form



Eine solche Struktur kann nun auf zwei Arten realisiert werden: 1. von oben nach unten wobei bereits berechnete Werte zwischengespeichert werden (top-down mit Memoisierung) oder 2. von unten nach oben (bottom up), dann entfällt die Notwendigkeit der expliziten Zwischenspeicherung. Die erste ist in Algorithmus 35 angegeben, die zweite in Algorithmus 36. Beide dieser Prozeduren benötigen Laufzeit $\Theta(n)$. Der Vorteil der top-down Prozedur ist dass sie dem ursprünglichen Algorithmus am ähnlichsten ist (der ja auch von oben nach unten vorgeht). Der Vorteil der bottom-up Prozedur ist, dass sie einfacher ist. Allerdings müssen für die bottom-up Prozedur die notwendigen Teil-Instanzen zur Berechnung der ursprünglichen Instanz von vornherein bekannt sein.

8.1 Das Stabzerlegungsproblem

Wir betrachten nun das folgende Optimierungsproblem: gegeben ist ein Eisenstab der Länge $n \in \mathbb{N}$ sowie eine Liste von Preisen $p_1, \ldots, p_n$ wobei p_i der erzielbare Preis für einen Eisenstab

Algorithmus 35 Berechnung der n-ten Fibonacci-Zahl (top-down mit Memoisierung)

```
Prozedur FIB(n)
    Sei F[0,...,n] ein neues Datenfeld, überall mit −∞ initialisiert
    Antworte FIBREK(F, n)
Ende Prozedur

Prozedur FIBREK(F, n)
    Falls F[n] = −∞ dann
        Falls n = 0 dann
            F[n] := 0
        sonst falls n = 1 dann
            F[n] := 1
        sonst
            F[n] := FIBREK(F, n − 1) + FIBREK(F, n − 2)
        Ende Falls
    Ende Falls
    Antworte F[n]
Ende Prozedur
```

Algorithmus 36 Berechnung der n-ten Fibonacci-Zahl (bottom-up)

```
Prozedur FIB(n)
    Sei F[0,...,n] ein neues Datenfeld
    F[0] := 0
    F[1] := 1
    Für i = 2,...,n
        F[n] := F[n − 1] + F[n − 2]
    Ende Für
    Antworte F[n]
Ende Prozedur
```

der Länge i ist. Der gegebene Stab der Länge n soll so zerschnitten werden, dass der durch Verkauf der Einzelteile erzielbare Gewinn (d.h. die Summe der Preise) maximal ist.

Stabzerlegungsproblem

Eingabe: $n \in \mathbb{N}$, $p_1, \ldots, p_n \in \mathbb{N}$

Ausgabe: $i_1, \ldots, i_k \in \mathbb{N}$ so dass $n = i_1 + \cdots + i_k$ und $p_{i_1} + \cdots + p_{i_k}$ maximal

Für einen Stab der Länge n gibt es 2^{n-1} verschiedene Möglichkeiten eine Zerteilung zu wählen, da an jedem ganzzahligen Punkt entweder geschnitten oder nicht geschnitten werden kann. Alle diese Zerteilungen durchzuprobieren und davon jene mit maximalem Gewinn zu wählen würde also Zeit $\Omega(2^n)$ benötigen. Mit dynamischer Programmierung ist das auf effizientere Weise lösbar.

Beispiel 8.1. Gegeben $n = 8$ und $p_1, \ldots, p_n$ wie folgt:

i	1	2	3	4	5	6	7	8
p_i	1	3	6	9	13	15	16	18

Dann ist die optimale Zerlegung des Stabs der Länge 8 in einen Stab der Länge 3 und einen der Länge 5. Damit wird ein Gewinn von $p_3 + p_5 = 19$ realisiert.

Sei, für $1 \leq i \leq n$, r_i der für einen Stab der Länge i mit Preisen $p_1, \ldots, p_n$ maximal erzielbare Gewinn. Dann ist

$$r_i = \max\{p_1 + r_{i-1}, p_2 + r_{i-2} \ldots, p_{i-1} + r_1, p_i\}$$

da wir unterscheiden können ob nicht geschnitten wird (p_i) oder mindestens einmal geschnitten wird und dann nach Länge des ersten Stücks. Wir sehen also dass die optimale Lösung (die r_i bestimmt) zusammengesetzt ist durch optimale Lösungen für kleinere Instanzen des selben Problems (jene die die r_j bestimmen für $1 \leq j < i$). Das ist entscheidend für die Anwendbarkeit des dynamischen Programmierens. Außerdem ist die Gesamtanzahl der kleineren Instanzen, d.h. der r_j, die rekursiv zur Bestimmung von r_i benötigt werden und damit die Anzahl der Knoten im Teilproblemgraph klein (hier gleich i). Wenn wir $r_0 = 0$ setzen können wir r_i auch schreiben als

$$r_i = \max\{p_j + r_{i-j} \mid 1 \leq j \leq i\}.$$

Wir können jetzt, ähnlich wie bei den Fibonacci-Zahlen, eine bottom-up Berechnung der r_i in einem Datenfeld R durchführen. Um nicht nur den maximal erzielbaren Gewinn r_n sondern auch die Teilung $i_1 + \cdots + i_k = n$ des Stabs zu berechnen, verwenden wir das zusätzliche Datenfeld S das, für $i = 1, \ldots, n$, in $S[i]$ speichert wie lange das erste Stück der optimalen Zerteilung eines Stabs der Länge i ist. Die globale Zerteilung kann dann leicht aus S in Zeit $O(n)$ berechnet werden. Diese Vorgehensweise ist in Algorithmus 37 als Pseudocode formuliert. Die Laufzeit dieser Prozedur ist aufgrund der beiden verschachtelten Schleifen $\Theta(n^2)$.

8.2 Segmentierte Methode der kleinsten Quadrate

Bei der einfachen linearen Regression in der Statistik sind Punkte $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n) \in \mathbb{R}^2$ gegeben die z.B. aus einem Experiment stammen und wir wollen die (Ausgabe-)werte y_i durch die (Eingabe-)werte x_i erklären wobei wir annehmen, dass ein linearer Zusammenhang besteht. Wir wollen also eine Gerade $y = ax + b$ bestimmen welche die Punkte

Algorithmus 37 Stabzerlegung (bottom-up)

Prozedur STABZERLEGUNG(P, n)
  Sei $R[0, \ldots, n]$ neues Datenfeld
  Sei $S[1, \ldots, n]$ neues Datenfeld
  $R[0] := 0$
  Für $i = 1, \ldots, n$
    $m := 0$
    Für $j = 1, \ldots, i$
      Falls $P[j] + R[i - j] \geq m$ **dann**
        $m := P[j] + R[i - j]$
        $S[i] := j$
      Ende Falls
    Ende Für
    $R[i] := m$
  Ende Für
  Antworte $(R[n], S)$
Ende Prozedur

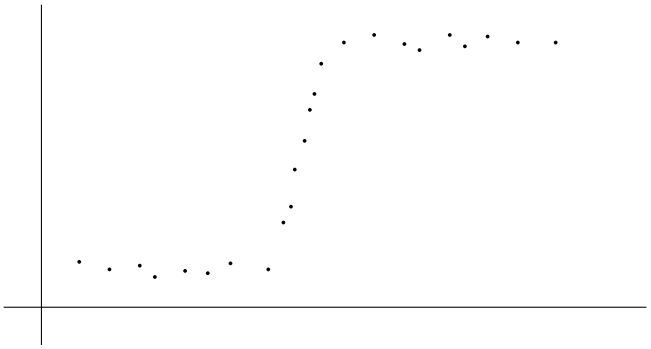


Abbildung 8.1: Eingabedaten für segmentierte lineare Regression

$p_1, \ldots, p_n$ bestmöglich im Sinne der kleinsten Fehlerquadrate annähert, d.h. es sollen $a, b \in \mathbb{R}$ bestimmt werden so dass die *Fehlerquadratsumme*

$$\sum_{i=1}^{n} (y_i - ax_i - b)^2$$

minimiert wird. Man kann zeigen dass diese Lösung eindeutig ist und gegeben ist durch

$$a = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})} \quad \text{und} \quad b = \bar{y} - a\bar{x}.$$

wobei $\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n}$ und $\bar{y} = \frac{\sum_{i=1}^{n} y_i}{n}$.

Oft können die gegebenen Daten aber durch eine einzelne Gerade nicht sehr gut angenähert werden, siehe Abbildung 8.1. Eine Reaktion darauf, die wir hier verfolgen wollen, besteht darin, die Daten stückweise durch Geraden anzunähern. Dabei muss nun natürlich spezifiziert werden was minimiert werden soll, d.h. wie die Anzahl der Geraden gegen die Fehlerquadratsumme gewichtet werden soll. Eine Möglichkeit dies zu tun besteht darin, eine Konstante $A > 0$ festzulegen, welche die Kosten der Verwendung einer (zusätzlichen) Gerade darstellt. Dann werden die

Punkte $p_1, \ldots, p_n$ entlang der x-Achse in Segmente geteilt von denen jedes unabhängig durch eine Gerade approximiert wird. Je nach Wert von A fällt die Anzahl der Segmente mehr oder weniger stark ins Gewicht. Wir kommen so zum folgenden Optimierungsproblem

Segmentierte einfache lineare Regression

Eingabe: $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n) \in \mathbb{R}^2$ mit $x_1 < \ldots < x_n$ und $A > 0$

Ausgabe: $0 = n_0 < n_1 < \cdots < n_{k-1} < n_k = n$ und für alle $i = 1, \ldots, k$ eine Gerade $y = a_i x + b_i$ so dass

$$\sum_{i=1}^{k} \left(A + \sum_{j=n_{i-1}+1}^{n_i} (y_j - a_i x_j - b_i)^2 \right)$$

minimal ist

Auch dieses Problem eignet sich für einen Ansatz mit dynamischem Programmieren. Für $j = 1, \ldots, n$ seien c_j die Kosten der optimalen Lösung des Teilproblems $p_1, \ldots, p_j$. Dann setzt sich c_j zusammen aus den Kosten des letzten Segments $p_i, \ldots, p_j$ sowie den Kosten der optimalen Lösung des Teilproblems $p_1, \ldots, p_{i-1}$ und den Kosten der Verwendung einer Geraden. Wir erhalten also

$$c_j = \min\{e_{i,j} + A + c_{i-1} \mid 1 \leq i \leq j\}$$

wobei $e_{i,j}$ die Fehlerquadratsumme des Segments $p_i, \ldots, p_j$ ist. Außerdem sei $c_0 = 0$, womit der Fall dass $p_1, \ldots, p_j$ nicht mehr unterteilt wird abgedeckt wird durch $c_j = e_{1,j} + A + c_0 = e_{1,j} + A$. Ähnlich wie bei der Stabzerlegung bildet diese Beobachtung nun die Basis für einen Algorithmus der die mehrfache Betrachtung von Teilproblemen vermeidet, siehe Algorithmus 38. Für ein Segment von k Punkten kann die Gerade mit minimaler Fehlerquadratsumme in Zeit $O(k)$ über die oben angegebenen Formeln bestimmt werden. Damit kann die kleinste Fehlerquadratsumme dieser Gerade in Zeit $O(k)$ bestimmt werden. Auf diese Weise alle $E[i, j]$ mit $i \leq j \leq n$ zu berechnen benötigt also Zeit $O(n^3)$. Die beiden verschachtelten Schleifen im zweiten Teil des Algorithmus benötigen Zeit $O(n^2)$. Wir erhalten also insgesamt eine Laufzeit von $O(n^3)$. Die Laufzeit dieses Algorithmus kann durch eine geschicktere Behandlung der ersten Phase auf $O(n^2)$ reduziert werden. Da dies aber nicht mehr zur Diskussion der dynamischen Programmierung beiträgt wollen wir hier darauf verzichten.

Algorithmus 38 Segmentierte Methode der kleinsten Quadrate

Prozedur SegmentierteMKQ(P, n, A)
  Sei $E[1, \ldots, n; 1, \ldots, n]$ ein neues Datenfeld
  Für alle $1 \leq i \leq j \leq n$
    $E[i, j] :=$ kleinste Fehlerquadratsumme für das Segment $p_i, \ldots, p_j$
  Ende Für

  Sei $C[0, \ldots, n]$ ein neues Datenfeld
  Sei $S[1, \ldots, n]$ ein neues Datenfeld
  $C[0] := 0$
  Für $j = 1, \ldots, n$
    $m := +\infty$
    Für $i = 1, \ldots, j$
      Falls $E[i, j] + A + C[i-1] < m$ **dann**
        $m := E[i, j] + A + C[i-1]$
        $S[j] := i$
      Ende Falls
    Ende Für
    $C[j] := m$
  Ende Für
  Antworte $(C[n], S)$
Ende Prozedur

Kapitel 9

Randomisierung

Unter einem *randomisierten Algorithmus* versteht man einen Algorithmus der gewisse Entscheidung basierend auf (aus externer Quelle erhaltenen) Zufallszahlen macht[1]. Das kann zu verschiedenen Zwecken nützlich sein. Einerseits kann man die Eingabedaten oder das Verhalten des Algorithmus randomisieren und so sicherstellen, dass ein Algorithmus auch im schlechtesten Fall nur die Komplexität des durchschnittlichen Falls hat. Je nach Laufzeit im schlechtesten und durchschnittlichen Fall und je nach Verteilung der Eingabedaten ist das mehr oder weniger nützlich. Andererseits kann Randomisierung in einem Algorithmus auch verwendet werden, um schnell ein Resultat zu liefern, das aber nur mit gewisser Wahrscheinlichkeit < 1 korrekt ist. Eine Wiederholung (mit unabhängigen Zufallszahlen) erlaubt dann eine beliebige Annäherung der Irrtumswahrscheinlichkeit an 0.

9.1 Randomisierung der Eingabe

In Hinblick auf ein einfaches Beispiel für die Randomisierung der Eingabe wollen wir das folgende *Bewerberproblem* betrachten. Eine Abfolge von Bewerbern absolviert Vorstellungsgespräche für eine Stelle. Einer hire&fire-Mentalität folgend soll zu jedem Zeitpunkt, d.h. auch zwischen den Bewerbungsgesprächen, der beste Bewerber die Stelle halten. Wenn ein besserer Bewerber als der aktuelle Stelleninhaber gefunden wird, wird dieser durch jenen ersetzt. Wir gehen davon aus dass die Bewerber mit $k_1, \ldots, k_n \in \mathbb{R}_{>0}$ beurteilt werden. Die natürliche Vorgehensweise ist in Algorithmus 39 als Pseudocode ausformuliert. Wir gehen dabei davon aus dass $K[i]$ den Wert k_i enthält und dass $K[0] = 0$ ist. Für die Analyse wollen wir davon ausgehen, dass die

Algorithmus 39 Direkte Lösung des Bewerberproblems

```
1: Prozedur BewerbersuChe(K, n)
2:     m := 0
3:     Für i := 1, ... n
4:         Falls K[i] > K[m] dann          ▷ Vorstellungsgespräch für Kandidat i
5:             m := i                      ▷ Einstellen von Kandidat i
6:         Ende Falls
7:     Ende Für
8:     Antworte m
9: Ende Prozedur
```

[1] Das ist nicht zu verwechseln mit einem nicht-deterministischen Algorithmus der, in gewissem Sinn, verschiedene deterministische Berechnungen gleichzeitig macht.

Einstellung eines Kandidaten hohe Kosten verursacht. Wir interessieren uns also dafür, wie oft die Zeile 5 in dieser Prozedur ausgeführt wird. Eine Vorgehensweise wie in Algorithmus 39 zur Suche eines Maximums oder Minimums kommt oft als Teil anderer Algorithmen vor.

Lemma 9.1. *Die Anzahl der Neuanstellungen ist im schlechtesten Fall n und im durchschnittlichen Fall $O(\log n)$.*

Beweis. Der schlechteste Fall tritt ein wenn $k_1 < k_2 < \cdots < k_n$. Dann werden n Neuanstellungen durchgeführt, eine für jeden Kandidaten.

Für den durchschnittlichen Fall gehen wir davon aus, dass jede Eingabepermutation der k_i gleich wahrscheinlich ist. Sei X die Anzahl der Neuanstellungen und sei

$$X_i = \begin{cases} 1 & \text{falls der } i\text{-te Kandidat eingestellt wird und} \\ 0 & \text{falls der } i\text{-te Kandidat nicht eingestellt wird.} \end{cases}$$

Dann ist $\mathrm{E}X_i = \mathrm{W}(X_i = 1)$. Da der i-te Kandidat eingestellt wird genau dann wenn $k_i = \max\{k_1, \ldots, k_i\}$, ist nach Lemma 2.1 die Wahrscheinlichkeit $\mathrm{W}(X_i = 1) = \frac{1}{i}$. Somit erhalten wir

$$\mathrm{E}X = \mathrm{E}\sum_{i=1}^{n} X_i = \sum_{i=1}^{n} \mathrm{E}X_i = \sum_{i=1}^{n} \frac{1}{i} = H_n = \ln n + O(1).$$

□

Wir haben also einen Algorithmus der im schlechtesten Fall deutlich höhere Kosten verursacht als im durchschnittlichen Fall. Um die Laufzeit des durchschnittlichen Falls zu garantieren, können wir die Eingabedaten vorverarbeiten indem wir auf sie eine (uniform verteilte) Zufallspermutation anwenden. Sei ZufallspermutationRandomisiert(n) eine solche uniform verteilte Zufallspermutation von n Elementen. Wir erhalten dann den in Algorithmus 40 angegebenen Pseudocode. Der Erwartungswert der Anzahl der Neuanstellungen wird hier über die Verteilung der Eingabe

Algorithmus 40 Randomisierte Lösung des Bewerberproblems

```
Prozedur BewerbersucheRandomisiert(K, n)
    K := Zufallspermutation(K)
    Antworte Bewerbersuche(K, n)
Ende Prozedur
```

und der Zufallspermutation und der Eingabe gebildet und ist damit unabhängig von der Eingabe $O(\log n)$. Die systematische Konstruktion einer Eingabe mit schlechtem Verhalten ist also nicht mehr möglich. In diesem Sinn gibt es keine worst-case Eingabe mehr.

Eine vergleichbare Situation trat auch in Abschnitt 5.2 auf. Das auf (nicht-balancierten) Suchbäumen basierende Sortierverfahren hatte eine Laufzeit von $O(n^2)$ im schlechtesten Fall aber $O(n \log n)$ im Durchschnittsfall. Durch eine Zufallspermutation der Eingabe kann dieses in ein randomisiertes Verfahren umgewandelt werden dass auf beliebiger Eingabe eine erwartete Laufzeit von $O(n \log n)$ hat.

Wir wollen uns jetzt damit beschäftigen wie man eine Zufallspermutation erzeugen kann. Klar ist, dass wir irgendeine Art von Zufallsquelle voraussetzen müssen. Wir nehmen also die Verfügbarkeit einer Prozedur Zufall(i, j) an, die eine uniform verteilte Zufallszahl im Intervall $[i, j] \subseteq \mathbb{N}$ liefert. Viele Programmiersprachen bzw. Betriebsysteme stellen tatsächlich solche Funktionen zur Verfügung, die als Zufallsquelle verschiedene Umgebungsdaten oder einen Pseudozufallszahlen-Generator verwenden.

Ein geeigneter Ansatz zur Berechnung einer zufälligen Permutation eines Datenfelds A der Länge n besteht darin, für $i = 1, \ldots, n$ das Element $A[i]$ mit einem zufälligen Element in $A[i, \ldots, n]$ zu vertauschen. Das ist der Algorithmus von Fisher-Yates, als Pseudocode ausformuliert in Algorithmus 41. Dieser Algorithmus hat Laufzeit $\Theta(n)$ (unter der Annahme dass ZUFALL(i, j)

Algorithmus 41 Der Fisher-Yates Algorithmus

```
Prozedur FisherYates(A)
    Für i = 1, ..., A.Länge
        Vertausche A[i] und A[Zufall(i, n)]
    Ende Für
    Antworte A
Ende Prozedur
```

konstante Zeit benötigt). Klar ist, dass der Algorithmus von Fisher-Yates eine Permutation des Eingabdatenfelds berechnet, da er nur Vertauschungen durchführt. Für die Analyse der Wahrscheinlichkeitsverteilung der erzeugten Permutation ist noch etwas Arbeit nötig.

Satz 9.1. *Der Fisher-Yates Algorithmus erzeugt eine uniform verteilte Zufallspermutation.*

Beweis. Sei $n = A.Länge$ und für $i = 0, 1, \ldots, n$ sei A_i das Datenfeld nach dem i-ten Durchlauf der Schleife. Dann ist A_0 das Eingabedatenfeld. Sei o.B.d.A. $A[i] = i$ für $i = 1, \ldots, n$. Sei V_k^n die Menge aller Variationen von k aus n Elementen ohne Wiederholung. Dann ist $|V_k^n| = \frac{n!}{(n-k)!}$. Wir zeigen mit Induktion nach $i = 1, \ldots, n$ dass für alle $v \in V_i^n$: $\mathrm{W}(A_i[1, \ldots, i] = v) = \frac{1}{|V_i^n|} = \frac{(n-i)!}{n!}$. Für $i = n$ folgt daraus dass das Ausgabedatenfeld A_n eine uniform verteilte Permutation ist. Die Induktionsbasis $i = 1$ folgt direkt aus Definition des Algorithmus. Für den Induktionsschritt sei $w \in V_{i+1}^n$, dann ist $w = \langle v, x \rangle$ wobei $v \in V_i^n$ und $x \in \{1, \ldots, n\} \setminus v$. Dann ist

$$\mathrm{W}(A_{i+1}[1, \ldots, i+1] = w) = \mathrm{W}(A_i[1, \ldots, i] = v \text{ und } A_{i+1}[i+1] = x)$$

und da $\{1, \ldots, n\} \setminus v$ genau die Elemente von $A_i[i+1, \ldots, n]$ sind ist $\mathrm{W}(A_{i+1}[i+1] = x) = \frac{1}{n-i}$

$$= \frac{(n-i)!}{n!} \frac{1}{n-i} = \frac{(n-(i+1))!}{n!}.$$

□

9.2 Quicksort

Quicksort ist einer der schnellsten Sortieralgorithmen und wird deshalb in der Praxis oft verwendet. Der Algorithmus folgt dem Teile-und-Herrsche Prinzip. Das Eingabedatenfeld $A[1, \ldots, n]$ wird zunächst durch Vertauschungen von Elementen in zwei Teile geteilt: $A[1, \ldots, m-1]$ und $A[m+1, \ldots, n]$ so dass alle Elemente im linken Teil kleiner gleich $A[m]$ sind und alle Elemente im rechten Teil größer gleich $A[m]$ sind. Dann wird die Prozedur rekursiv auf den beiden Teilen aufgerufen. Insgesamt wird so das Datenfeld durch Vertauschungen sortiert.

Die Aufteilung des Datenfelds $A[l, \ldots, r]$ wird so realisiert, dass zunächst ein *Pivotelement* aus dem Datenfeld gewählt wird, z.B. $A[r]$. Danach wird das Datenfeld von links nach rechts durchlaufen wobei der jeweils bisher durchlaufene Teil aus zwei Teilen besteht: jene Elemente die größer sind als $A[r]$ und jene die kleiner sind als $A[r]$. Am Ende wird das Pivotelement $A[r]$ an die richtige Stelle permutiert. Diese Vorgehensweise ist in Algorithmus 42 ausformuliert.

Algorithmus 42 Quicksort

```
Prozedur QUICKSORT(A, l, r)
    Falls l < r dann
        m := TEILEN(A, l, r)
        QUICKSORT(A, l, m − 1)
        QUICKSORT(A, m + 1, r)
    Ende Falls
Ende Prozedur

Prozedur TEILEN(A, l, r)
    x := A[r]
    i := l − 1
    Für j = l, . . . , r − 1
        Falls A[j] ≤ x dann
            i := i + 1
            Vertausche A[i] mit A[j]
        Ende Falls
    Ende Für
    Vertausche A[i + 1] mit A[r]
    Antworte i + 1
Ende Prozedur
```

Die Prozedur TEILEN wählt zunächst als Pivotelement $x = A[r]$. Zu Beginn jedes Schleifendurchlaufs gilt für das Datenfeld A: falls $k \in \{l, \ldots, i\}$ ist $A[k] \leq x$, falls $k \in \{i+1, \ldots, j-1\}$ ist $A[k] > x$ und falls $k \in \{j, \ldots, r\}$ wurde $A[k]$ noch nicht mit x verglichen. In jedem Schleifendurchlauf wird j um eins erhöht und gegebenenfalls durch eine Vertauschung diese Invariante beibehalten.

Beispiel 9.1. Die Prozedur TEILEN angewandt auf das Datenfeld

3	6	2	7	4	1	8	5

transformiert dieses in

3	2	4	1	5	7	8	6

und antwortet mit $i + 1 = 5$.

Die Laufzeit der Prozedur TEILEN ist $\Theta(r - l)$. Die Gesamtlaufzeit von Quicksort hängt also davon ab, wie die Aufteilungen des Datenfeldes stattfinden. Intuitiv ist eine Aufteilung umso besser je schneller die Größe der Instanzen schrumpft. Tatsächlich kann man sich leicht überlegen, dass im Fall einer aufsteigend sortierten Eingabe ein Datenfeld der Länge n aufgespalten wird in eines der Länge $n - 1$, in das Pivotelement, sowie eines der Größe 0. Für diesen Fall ist die Laufzeit von Quicksort also gegeben durch

$$T(n) = T(n-1) + \Theta(n).$$

Durch die Substitutionsmethode kann leicht gezeigt werden dass $T(n) = \Theta(n^2)$. Die Laufzeit von Quicksort im schlechtesten Fall ist also $\Omega(n^2)$.

Der obigen Intuition folgend kann man sich vorstellen, dass die beste Aufteilung eines Datenfelds der Länge n jene ist die zwei Datenfelder der selben Länge erzeugt. Die Laufzeit wird dann (modulo Rundung) beschrieben durch die Rekursionsgleichung

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

Nach den zweiten Fall des Master-Theorems ergibt sich also $T(n) = \Theta(n \log n)$.

Die Laufzeit ergibt sich aber selbst dann zu $O(n \log n)$ wenn eine unbalancierte Aufteilung gewählt wird, so lange die Längen der Teildatenfelder durch eine multiplikative Konstante und die ursprüngliche Länge bestimmt werden. Wird zum Beispiel in jedem Schritt eine 1-zu-9 Aufteilung gewählt, dann erhalten wir (modulo Rundung) die Rekursionsgleichung

$$T(n) = T(\frac{1}{10}n) + T(\frac{9}{10}n) + cn.$$

Im Rekursionsbaum dann jede Ebene Kosten $\leq cn$ und es gibt $O(\log_{\frac{9}{10}} n)$ Ebenen, insgesamt haben wir also Kosten von $O(n \log_{\frac{9}{10}} n) = O(n \log n)$.

Wir analysieren nun den durchschnittlichen Fall. Im durchschnittlichen Fall treten bessere und schlechtere Aufteilungen gemischt auf. Wir gehen, wie immer bisher, davon aus dass alle Permutationen des Eingabedatenfelds gleich wahrscheinlich sind. Weiters nehmen wir zur Vereinfachung der Analyse an, dass die Schlüssel paarweise unterschiedlich sind. Dann können wir zeigen:

Satz 9.2. *Die Laufzeit von Quicksort im durchschnittlichen Fall ist $O(n \log n)$.*

Beweis. Ein Aufruf von Quicksort auf einem Datenfeld der Länge n führt zu höchstens n Aufrufen der Prozedur Teilen. Der Aufwand eines Aufrufs von Teilen ist $\Theta(v)$ wobei v die Anzahl der durchgeführten Vergleiche ("$A[j] \leq x$") ist. Weiters wird ja immer mit dem Pivotelement verglichen das in späteren Aufrufen von Teilen keine Rolle mehr spielt, also wird kein Paar zwei Mal verglichen. Sei X die Anzahl der Vergleiche die über alle Aufrufe von Teilen hinweg insgesamt durchgeführt werden. Dann ist die Gesamtlaufzeit von Quicksort $O(X + n)$. Wir wollen nun EX bestimmen.

Seien dazu $\{z_1, \ldots, z_n\}$ die Elemente des Eingabedatenfeldes mit $z_1 < \cdots < z_n$. Für $i \leq j$ sei weiters $Z_{i,j} = \{z_i, \ldots, z_j\}$. Der Algorithmus beginnt mit der Auswahl eines Pivotelements z_k aus $Z_{1,n}$ das mit $z_1, \ldots, z_{k-1}, z_{k+1}, \ldots, z_n$ verglichen wird um die Teilung des Datenfelds zu berechnen. Nach Teilung des Datenfelds werden $z_1, \ldots, z_{k-1}$ sowie $z_{k+1}, \ldots, z_n$ nur noch untereinander verglichen, nicht aber miteinander. Dieser Prozess wiederholt sich dann.

Allgemein können wir also feststellen, dass z_i mit z_j verglichen wird genau dann wenn z_i oder z_j als erstes in $Z_{i,j}$ als Pivotelement gewählt wird und damit also kein (früher gewähltes) Pivotelement zwischen z_i und z_j steht. Sei

$$X_{i,j} = \begin{cases} 1 & \text{falls } z_i \text{ mit } z_j \text{ verglichen wird} \\ 0 & \text{sonst} \end{cases}$$

Dann ist $X = \sum_{1 \le i < j \le n} X_{i,j}$ und wir erhalten

$$\begin{aligned}
\mathrm{E}X &= \sum_{1 \le i < j \le n} \mathrm{E}X_{i,j} = \sum_{1 \le i < j \le n} \mathrm{W}(z_i \text{ wird mit } z_j \text{ verglichen}) \\
&= \sum_{1 \le i < j \le n} \mathrm{W}(z_i \text{ oder } z_j \text{ ist erstes Pivotelement aus } Z_{i,j}) \\
&= \sum_{1 \le i < j \le n} \left(\frac{1}{j-i+1} + \frac{1}{j-i+1} \right) \\
&= 2 \sum_{1 \le i < j \le n} \frac{1}{j-i+1} \\
&= 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k+1} \\
&< 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k} \\
&= 2 \sum_{i=1}^{n-1} O(\ln n) \\
&= O(n \log n).
\end{aligned}$$

Somit ist auch die Gesamtlaufzeit im durchschnittlichen Fall $O(n \log n)$. □

Um diese durchschnittliche Laufzeit als erwartete Laufzeit auch auf ungünstigen Eingaben zu garantieren bietet sich die in Algorithmus 43 angegebene Randomisierung an. Hier wird die

Algorithmus 43 Randomisiertes Quicksort

```
Prozedur QuicksortRandomisiert(A, l, r)
    Falls l < r dann
        i := Zufall(l, r)
        Vertausche A[i] und A[r]
        m := Teilen(A, l, r)
        Quicksort(A, l, m − 1)
        Quicksort(A, m + 1, r)
    Ende Falls
Ende Prozedur
```

Auswahl eines zufälligen Elements als Pivotelement dadurch realisiert, dass das zufällig gewählte Element mit $A[r]$ vertauscht wird bevor die Teilung des Datenfelds durchgeführt wird. Ein zufällig gewähltes Pivotelement teilt das aktuelle Teildatenfeld mit großer Wahrscheinlichkeit gut auf. Diese randomisierte Variante von Quicksort hat, unabhängig vom Eingabedatenfeld, die erwartete Laufzeit $O(n \log n)$.

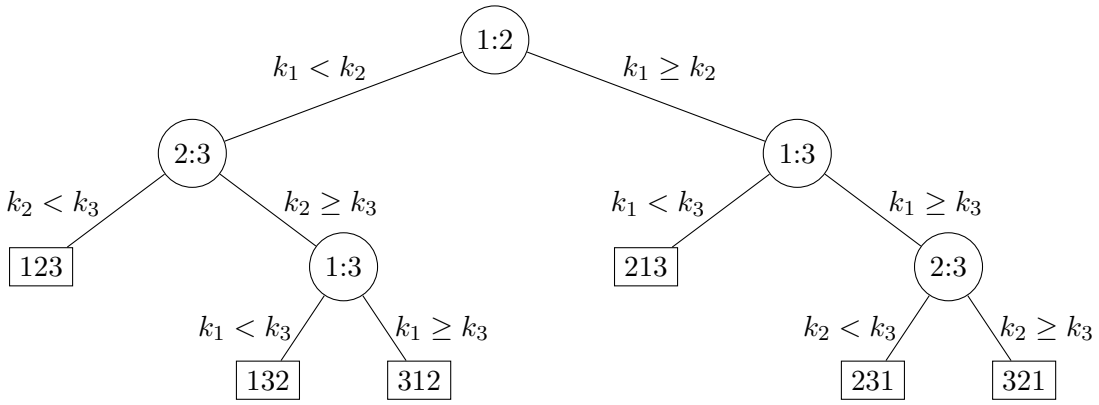
9.3 Eine untere Schranke auf Sortieralgorithmen

Wir haben nun einige Sortierverfahren gesehen deren Laufzeit immer mindestens $n \log n$ war. Es ist natürlich zu fragen, ob es auch besser möglich ist. Eine positive Antwort auf diese Frage bestünde in einem schnelleren Sortierverfahren (das in gewissen Spezialfällen tatsächlich

existiert, siehe unten). Eine negative Antwort darauf zu geben fällt aber viel schwerer da ein Beweis über alle Sortierverfahren notwendig wäre. Wir werden hier eine untere Schranke für ein bestimme *Klasse von Sortierverfahren* beweisen: für vergleichsbasierte Sortierverfahren. Diese zeichnen sich dadurch aus, dass sie lediglich Schlüsselvergleiche verwenden können, um die korrekte Reihenfolge der Eingabedaten zu bestimmen (nicht aber z.B. Schlüssel addieren können). Alle bisher in dieser Vorlesung betrachteten Sortierverfahren sind von dieser Art.

Die Entscheidungen die ein vergleichsbasiertes Sortierverfahren trifft können durch einen Entscheidungsbaum dargestellt werden. Bei einer Eingabe von Schlüssel $k_1, \dots k_n$ stellt ein Knoten mit Beschriftung $i; j$ einen Vergleich von k_i mit k_j dar. Ein Blatt stellt dann die sortierende Permutation der Eingabe dar. Auf diese Weise induziert ein vergleichsbasiertes Sortierverfahren eine Folge $(T_n)_{n\geq 2}$ von Berechnungsbäumen. Der Entscheidungsbaum T_n hat dann $n!$ Blätter, denn einerseits muss ja jede Permutation vorkommen, da sie als Eingabe möglich ist. Andererseits unterscheiden sich die Permutationen an zwei verschiedenen Blättern und so kann keine Permutation zwei Mal vorkommen.

Beispiel 9.2. Ein Entscheidungsbaum eines vergleichsbasierten Sortierverfahrens für $n = 3$ ist:



Wir können dann zeigen:

Satz 9.3. *Jedes vergleichsbasierte Sortierverfahren benötigt im schlechtesten Fall Laufzeit* $\Omega(n \log n)$.

Beweis. Wir betrachten ein Eingabedatenfeld der Länge n und ein Sortierverfahren $\mathcal{A}$. Sei b die Anzahl der Blätter des Entscheidungsbaums von $\mathcal{A}$ für Eingabe der Länge n, dann ist $b = n!$. Sei h die maximale Höhe eines Blattes, dann ist $b \leq 2^h$ und so erhalten wir $n! \leq 2^h$, d.h. $\log(n!) \leq h$. Da $n! \geq (\frac{n}{2})^{\frac{n}{2}}$, ist $\log(n!) \geq \frac{n}{2} \log \frac{n}{2} = \Omega(n \log n)$. □

Um den Durchschnittsfall zu behandeln beweisen wir zunächst:

Lemma 9.2. *Ein binärer Baum T mit $b \geq 2$ Blättern hat mittlere Tiefe $\bar{d}(T) \geq \log b$.*

Beweis. Wir gehen mit Induktion auf b vor. Der Fall $b = 2$ ist trivial. Für den Induktionsschritt sei T_1 der linke Teilbaum von T mit b_1 Blättern und T_2 der rechte Teilbaum von T mit b_2 Blättern. Dann ist $b = b_1 + b_2$ und

$$\begin{aligned}
\bar{d}(T) &= \frac{1}{b}(b_1(\bar{d}(T_1) + 1) + b_2(\bar{d}(T_2) + 1)) \\
&\geq^{\text{IH}} \frac{1}{b}(b_1(\log b_1 + 1) + b_2(\log b_2 + 1)) \\
&= \frac{1}{b_1 + b_2}(b_1 \log 2b_1 + b_2 \log 2b_2).
\end{aligned}$$

Diese Funktion erreicht ihr Minimum bei $b_1 = b_2 = \frac{b}{2}$, also

$$\begin{aligned} &\geq \frac{1}{b}(\frac{b}{2}\log b + \frac{b}{2}\log b) \\ &= \log b. \end{aligned}$$

□

Satz 9.4. *Jedes vergleichsbasierte Sortierverfahren benötigt im durchschnittlichen Fall Laufzeit* $\Omega(n \log n)$.

Beweis. Sei $\mathcal{A}$ ein vergleichsbasiertes Sortierverfahren und sei T_n der Entscheidungsbaum von $\mathcal{A}$ für Eingaben der Länge n, dann hat T_n genau $n!$ Blätter und damit ist, nach Lemma 9.2, $\bar{d}(T_n) \geq \log n!$. Wie oben gezeigt ist $\log n! = \Omega(n \log n)$. □

Man kann durchaus auch sinnvolle Sortierverfahren angeben, die keine vergleichsbasierten Sortierverfahren sind und, zumindest für gewisse Spezialfälle, eine bessere Laufzeit haben. Zählsortieren geht davon aus, dass die Schlüssel in einen bestimmten Bereich $\{1, \ldots, k\}$ fallen und erlaubt dann eine Sortierung in Zeit $\Theta(n + k)$ wobei n die Länge des Eingabedatenfelds ist. Für hinreichend kleine k ist damit also ein Sortierverfahren mit Laufzeit $\Theta(n)$ angegeben. Ein Einsatz dieses Verfahrens ist nur sinnvoll wenn k nicht größer als zirka $n \log n$ ist.

Beispiel 9.3. Bei dem Eingabedatenfeld

$A =$

a	b	c	d	e	f	g	h
2	4	3	2	1	4	3	3

sind alle Schlüssel in $\{1, \ldots, 4\}$, d.h. also $k = 4$. Zunächst zählen wir, für alle $j \in \{1, \ldots, k\}$ wie viele Elemente den Schlüssel j haben und erhalten.

1	2	3	2

Daraus berechnen wir, für $j \in \{1, \ldots, k\}$ den letzten Index mit Schlüssel j im sortierten Datenfeld.

1	3	6	8

Das induziert eine Unterteilung

1	2	2	3	3	3	4	4

des Ausgabedatenfelds. Dieses wird dann, dieser Unterteilung folgend, mit den Elementen von A befüllt und wir erhalten.

e	a	d	c	g	h	b	f
1	2	2	3	3	3	4	4

Zählsortieren ist in Algorithmus 44 als Pseudocode ausformuliert.

Algorithmus 44 Zählsortieren

Prozedur ZÄHLSORTIEREN(A, k)
  Sei $B[1, \ldots, n]$ ein neues Datenfeld
  Sei $C[1, \ldots, k]$ ein neues Datenfeld, überall mit 0 initialisiert.
  Für $i := 1, \ldots, A.Länge$
    $C[A[i].x] := C[A[i].x] + 1$
  Ende Für
  Für $i := 2, \ldots, k$
    $C[i] := C[i] + C[i-1]$
  Ende Für
  Für $i := A.Länge - 1$
    $B[C[A[i].x]] := A[i]$
    $C[A[i].x] := C[A[i].x] - 1$
  Ende Für
  Antworte B
Ende Prozedur

9.4 Der Miller-Rabin Primzahltest

Wir wollen uns nun mit dem folgenden Entscheidungsproblem beschäftigen:

Primzahlen
Eingabe: $n \in \mathbb{N}$
Ausgabe: "ja" falls $n \in \mathbb{P}$, "nein" sonst

Dieses Problem hat wichtige Anwendungen in der Kryptographie, zum Beispiel bildet es die Grundlage der Schlüsselerzeugung beim RSA-Verfahren das wir in Abschnitt 9.5 genauer besprechen werden. Man kann sich leicht überlegen, dass eine Zahl $n \in \mathbb{N}$ zusammengesetzt ist genau dann wenn sie einen Teiler $\leq \sqrt{n}$ hat, so dass es für einen Primzahltest ausreichend ist, Probedivisionen von 1 bis $\sqrt{n}$ durchzuführen. Dieses auf Probedivisionen basierende Verfahren hat also Laufzeit $O(\sqrt{n})$. In der Praxis hat man es aber oft mit Primzahlen zu tun die etwa 1000 bis 3000 Bits haben. Nun ist z.B. $\sqrt{2^{2000}} = 2^{1000}$ was zu hoch ist, um zu einem praktisch einsetzbaren Algorithmus zu führen.

Seit 2002 ist ein Algorithmus bekannt der in Zeit $O((\log n)^k)$ für ein $k \in \mathbb{N}$ entscheidet ob ein gegebenen $n \in \mathbb{N}$ eine Primzahl ist (AKS-Verfahren). Das derzeit kleinste bekannt solche k ist $k = 7$. Dieses Resultat hat bisher nur theoretische Bedeutung, für die praktische Anwendung ist diese Klasse von Algorithmen nicht effizient genug. In der Praxis wendet man stattdessen probabilistische Primzahltests an die effizient sind und mit beliebig großer Wahrscheinlichkeit < 1 eine korrekte Antwort liefern. Um diese näher zu besprechen rufen wir uns zunächst den kleinen Satz von Fermat in Erinnerung:

Satz 9.5 (Kleiner Satz von Fermat)**.** *Sei $n \in \mathbb{P}$, $a \geq 1$, $\mathrm{ggT}(a, n) = 1$, dann ist*

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

Auf Basis dieses Satz ist ein einfacher Primzahltest möglich: Gegeben $n \in \mathbb{N}$ wählen wir zufällig ein $a \in \{2, \ldots, n-1\}$ und überprüfen (mit dem euklidischen Algorithmus) ob $\mathrm{ggT}(a, n) = 1$. Ist das nicht der Fall ist a ein Teiler von n und damit $n \notin \mathbb{P}$. Falls $\mathrm{ggT}(a, n) = 1$ ist dann

überprüfen wir ob $a^{n-1} \equiv 1 \pmod{n}$. Ist das nicht der Fall ist $n \notin \mathbb{P}$ (auch wenn wir keinen Teiler von n kennen). Ist das der Fall endet der Algorithmus *mit der Vermutung dass* $n \in \mathbb{P}$. Die Laufzeit dieses Verfahrens beträgt $O(\log n)$. Dieser Algorithmus kann nun für verschiedene zufällig gewählte a wiederholt werden. Falls sich dabei herausstellt dass $a^{n-1} \equiv 1 \pmod{n}$ für viele verschiedene a erhöht das unser Vertrauen in die Vermutung dass $n \in \mathbb{P}$.

Es gibt zusammengesetzte Zahlen n so dass für alle $a \in \{2, \ldots, n-1\}$ mit $\mathrm{ggT}(a, n) = 1$ gilt: $a^{n-1} \equiv 1 \pmod{n}$. Solche Zahlen heißen Carmichael-Zahlen. Es gibt unendlich viele Carmichael-Zahlen, die kleinste ist $3 \cdot 11 \cdot 17 = 561$.

Mit dem Fermat-Test können also nicht alle zusammengesetzen Zahlen als solche erkannt werden. Ein Test der diesen Defekt nicht hat ist der Miller-Rabin Test. Dieser basiert auf dem folgenden Satz

Satz 9.6. *Sei $n \geq 1$ ungerade, sei $n - 1 = 2^s d$, d ungerade, sei $a \geq 1$ mit $\mathrm{ggT}(a, n) = 1$. Falls $n \in \mathbb{P}$, dann ist*

$$a^d \equiv 1 \pmod{n}$$

oder es gibt ein $r \in \{0, \ldots, s-1\}$ so dass

$$a^{2^r d} \equiv -1 \pmod{n}.$$

Ohne Beweis.

Definition 9.1. Sei $n \geq 1$ ungerade, sei $n - 1 = 2^s d$, d ungerade. Dann heißt $a \in \mathbb{N}$ *(Miller-Rabin) Zeuge gegen die Primalität von n* falls $\mathrm{ggT}(a, n) = 1$ und $a^d \not\equiv 1 \pmod{n}$ sowie $a^{2^r d} \not\equiv -1 \pmod{n}$ für alle $r \in \{0, \ldots, s-1\}$:

Gegeben n und $a \in \{1, \ldots, n-1\}$ kann in Zeit $O(\log n)$ festgestellt werden ob a ein Zeuge gegen die Primalität von n ist. Die Bedingung $\mathrm{ggT}(a, n) = 1$ kann mit dem euklidischen Algorithmus überprüft werden. Die Berechnung von a^d modulo n kann durch die auf Binärdarstellung basierende schnelle Exponentation in Zeit $O(\log n)$ durchgeführt werden. Die Berechnung der $a^{2^r d}$ modulo n geschieht durch sukzessives quadrieren.

Satz 9.7. *Sei $n \geq 3$ eine ungerade zusammengesetzte Zahl, dann gibt es in $\{1, \ldots, n-1\}$ höchstens $\frac{n-1}{4}$ Zahlen, die zu n teilerfremd sind aber keine Zeugen gegen die Primalität von n sind.*

Ohne Beweis.

Korollar 9.1. *Sei $n \geq 3$ ungerade und zusammengesetzt, dann ist die Wahrscheinlichkeit dass* MillerRabin(n, t) *mit "wahrscheinlich prim" antwortet höchstens 2^{-2t}.*

Ein Primzahltest wie dieser wird zur Erzeugung von großen Primzahlen wie folgt eingesetzt. Man wählt eine zufällige Zahl n der gewünschten Größenordnung (also z.B. mit 2048 Bits). Für ein geeignetes t wird dann MillerRabin(n, t) ausgeführt. Ergibt das die Antwort "wahrscheinlich prim" wird n als Primzahl betrachtet und zurückgegeben. Ergibt das die Antwort "zusammengesetzt" wiederholt man das Verfahren für eine neue Zufallszahl n. Ein Wahl von z.B. $t = 15$ garantiert bereits eine Fehlerwahrscheinlichkeit höchstens 1 zu einer Milliarde. Das ist für die meisten praktische Anwendungen ausreichend.

Algorithmus 45 Miller-Rabin Primzahltest

```
Prozedur MillerRabin(n, t)
    Für i := 1, ..., t
        a := Zufall(1, n − 1)
        Falls ggT(a, n) = 1 dann
            Falls a Zeuge gegen Primalität von n ist dann
                Antworte "zusammengesetzt"
            Ende Falls
        sonst
            Antworte "zusammengesetzt"
        Ende Falls
    Ende Für
    Antworte "wahrscheinlich prim"
Ende Prozedur
```

9.5 Der RSA-Algorithmus

Als Anwendung großer Primzahlen wollen wir das RSA-Verschlüsselungsverfahren betrachten. Das RSA-Verfahren ist ein public-key-Verfahren, d.h. der Schlüssel einer Person X besteht aus zwei Teilen: einem öffentlichen Schlüssel der publiziert wird und zur Verschlüsselung von Nachrichten an X verwendet werden kann und einem privatem Schlüssel der im Besitz von X ist und geheim bleibt. Die Schlüsselerzeugung funktioniert wie folgt: Person X wählt zufällig zwei Primzahlen p und q wie im vorherigen Abschnitt beschrieben. Dann wird $n = pq$ als RSA-Modul bezeichnet. Weiters wählt X eine natürliche Zahl e mit

$$1 < e < (p-1)(q-1) \text{ und } \mathrm{ggT}(e, (p-1)(q-1)) = 1$$

und[2] berechnet eine Zahl d mit

$$1 < d < (p-1)(q-1) \text{ und } ed \equiv 1 \pmod{(p-1)(q-1)}$$

was mit dem erweiterten euklidischen Algorithmus möglich ist. Dann ist (n, e) der öffentliche Schlüssel und d der private Schlüssel.

Will nun eine Person Y eine Nachricht an X schicken so geht sie zur Verschlüsselung wie folgt vor. Zunächst nehmen wir an, dass für die Nachricht m gilt: $0 \leq m < n$. Die Nachricht m wird dann verschlüsselt zu

$$c \equiv m^e \pmod{n}.$$

Um diese Verschlüsselung durchzuführen reicht es, den öffentlichen Schlüssel (n, e) zu kennen. Die Entschlüsselung basiert dann auf dem folgenden Satz

Satz 9.8. *Sei (n, e) ein öffentlicher Schlüssel, sei d der dazugehörige private Schlüssel und sei $0 \leq m < n$. Dann gilt $(m^e)^d \equiv m \pmod{n}$.*

Um diesen Satz zu beweisen machen wir noch kurz die folgende zahlentheoretische Beobachtung.

Lemma 9.3. *Seien $n_1, n_2 \in \mathbb{Z}$ relativ prim, seien $a, b \in \mathbb{Z}$ mit $a \equiv b \pmod{n_1}$ und $a \equiv b \pmod{n_2}$, dann ist $a \equiv b \pmod{n_1 n_2}$.*

[2] Das Auftreten von $(p-1)(q-1)$ erklärt sich durch den Wert der Eulerschen φ-Funktion auf n: $\varphi(n) = \varphi(p)\varphi(q) = (p-1)(q-1)$.

Beweis. Da $a \equiv b \pmod{n_1}$ und $a \equiv b \pmod{n_2}$ gibt es $k_1, k_2 \in \mathbb{Z}$ so dass $a - b = n_1 k_1$ und $a - b = n_2 k_2$. Also ist $n_1 k_1 = n_2 k_2$ und da n_1 und n_2 relativ prim sind gilt $n_2 \mid k_1$, d.h. also es gibt ein $l_2 \in \mathbb{Z}$ mit $k_1 = n_2 l_2$. Damit ist $a - b = n_1 n_2 l_2$, d.h. also $a \equiv b \pmod{n_1 n_2}$. □

Beweis von Satz 9.8. Da $ed \equiv 1 \pmod{(p-1)(q-1)}$ existiert ein $l \in \mathbb{Z}$ so dass $ed = 1 + l(p-1)(q-1)$ und damit

$$(m^e)^d = m^{ed} = m^{1+l(p-1)(q-1)} = m(m^{(p-1)(q-1)})^l.$$

Dann gilt

$$(m^e)^d \equiv m(m^{p-1})^{(q-1)l} \equiv m \pmod{p}$$

wegen dem kleinen Satz von Fermat falls m kein Vielfaches von p ist. Falls m ein Vielfaches von p ist, dann gilt diese Gleichung ebenfalls, da beide Seiten kongruent 0 sind. Analog dazu sieht man dass $(m^e)^d \equiv m \pmod{q}$. Aus Lemma 9.3 folgt damit $(m^e)^d \equiv m \pmod{n}$. □

Die Sicherheit dieses Verfahrens basiert darauf, dass d aus $n = pq$ und e nicht, bzw. nur mit unrealistisch großem Aufwand berechnet werden kann. Man beachte aber dass d sehr wohl effizient aus e und $(p-1)(q-1)$ berechnet werden kann. Könnte man also n schnell genug in p und q faktorisieren, wäre das RSA-Verfahren geknackt. Je nach Sicherheitsanforderungen ist es nach aktuellem Stand empfehlenswert für n eine Zahl mit zirka 500 - 3000 Bits zu wählen. Die beiden Primzahlen p und q sollten so gewählt werden, dass sie ungefähr die selbe Größenordnung haben, aber auch nicht zu nahe beieinander sind.

Kapitel 10

Lineare Optimierung

10.1 Einführung

Viele Optimierungsprobleme bestehen darin eine gewisse Zielfunktion unter Einhaltung gewisser Bedingungen zu maximieren oder zu minimieren. Falls sowohl die Zielfunktion affin linear ist als auch die Bedingungen affin lineare Gleichungen und Ungleichungen sind, dann spricht man von einem *linearen Optimierungsproblem*. In diesem Kapitel werden wir einige Schritte in das weitläufige Thema der linearen Optimierung machen und beginnen dazu mit einem Beispiel.

Beispiel 10.1. Eine Fabrik hat eine Maschine zur Verfügung mit der zwei verschiedene Produkte erzeugt werden können. Der Einsatz der Maschine im kommenden Monat soll geplant werden, wie immer natürlich mit dem Ziel den Gewinn zu maximieren. Dabei unterliegt die Verwendung dieser Maschine gewissen Bedingungen und Einschränkungen, die wir im folgenden beschreiben und formalisieren werden. Dabei sei x_1 die Stückzahl des ersten Produkts und x_2 die Stückzahl des zweiten Produkts. Somit ist also $x_1 \geq 0$ und $x_2 \geq 0$.

1. Mit einer Einheit des ersten Produkts lässt sich 15€ Gewinn erzielen, mit einer Einheit des zweiten Produkts 10€. Wir wollen also

   $$15x_1 + 10x_2$$

   maximieren.

2. Im kommenden Monat stehen auf dieser Maschine 20000 Arbeitsminuten (also ca. 16h pro Arbeitstag) zur Verfügung. Die Produktion einer Einheit des ersten Produkts benötigt 20 Minuten, einer Einheit des zweiten Produkts 10 Minuten. Also

   $$20x_1 + 10x_2 \leq 20000, \text{ d.h. } 2x_1 + x_2 \leq 2000.$$

3. Ein bestehender Vertrag verpflichtet zur Produktion von mindestens 200 Stück des zweiten Produkts, also

   $$x_2 \geq 200.$$

4. Die Produktion einer Einheit des ersten Produkts erzeugt 10 Gramm CO_2, einer Einheit des zweiten Produkts sogar 40 Gramm. Der CO_2-Ausstoß pro Monat ist auf 38 kg beschränkt. Also

   $$10x_1 + 40x_2 \leq 38000, \text{ d.h. } x_1 + 4x_2 \leq 3800$$

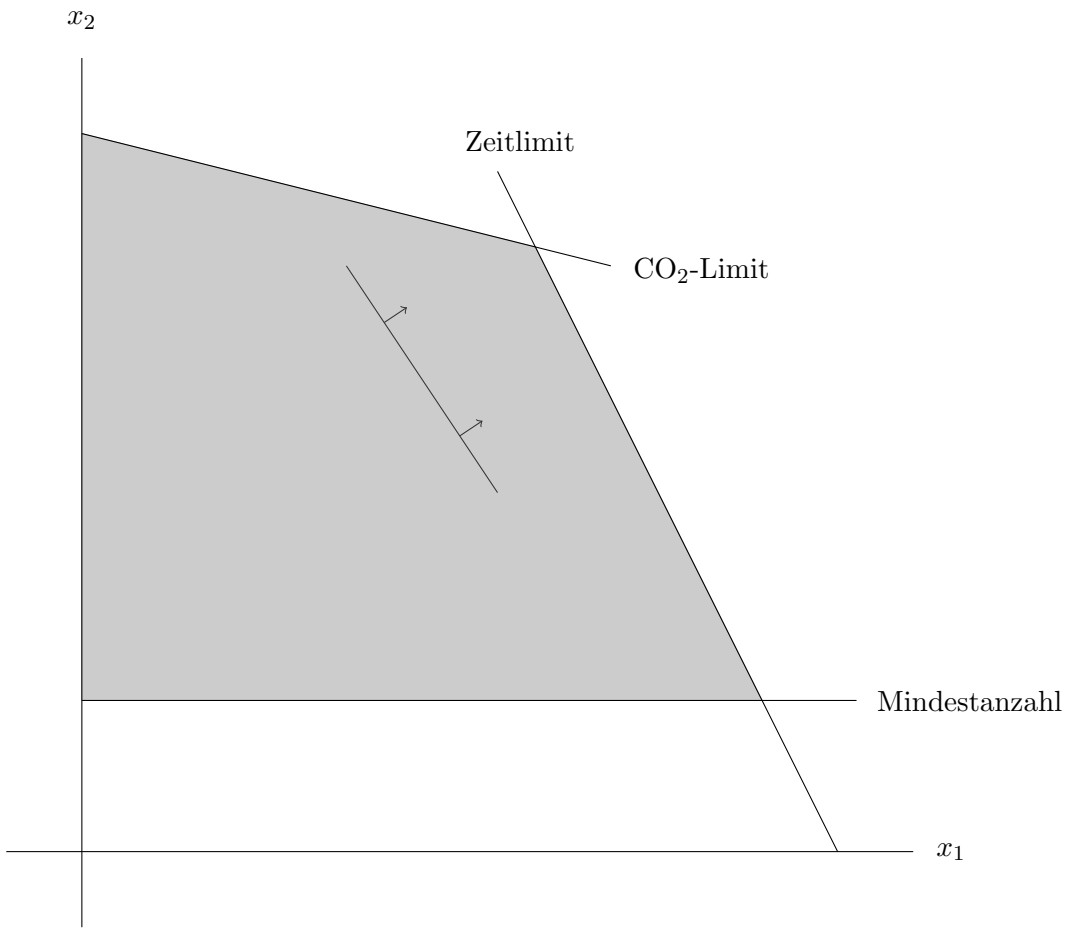


Abbildung 10.1: Beispiel für lineare Optimierung

Wir suchen also (x_1, x_2) so dass

$$\begin{aligned} 2x_1 + x_2 &\leq 2000 \\ x_2 &\geq 200 \\ x_1 + 4x_2 &\leq 3800 \end{aligned}$$

und, unter allen Paaren die diese Bedingungen erfüllen, soll $15x_1 + 10x_2$ maximal sein. Die Situation ist in Abbildung 10.1 graphisch dargestellt. Der graue Bereich besteht aus jenen (x_1, x_2) die alle Bedingungen erfüllen. Ein maximaler Punkt kann darin auf geometrische Weise wie folgt gefunden werden: Wir betrachten die Gerade $z = 15x_1 + 10x_2$ für wachsendes z und schieben sie auf diese Weise durch den zulässigen Bereich bis jede weitere Verschiebung um ein $\varepsilon > 0$ zu einem leeren Schnitt mit dem zulässigen Bereich führen würde. Hier geschieht, wenn die Gerade den Schnittpunkt der Zeit- mit der CO_2-Gerade schneidet. Dieser ist, wie man leicht anhand der Geradengleichungen berechnen kann, $(x_1, x_2) = (600, 800)$. Der maximale Gewinn wird also erreicht bei Erzeugung von 600 Einheiten des ersten Produkts und 800 Einheiten des zweiten Produkts und beträgt somit 17.000€.

Ausgehend von diesem Beispiel können wir nun die allgemeine Problemstellung betrachten. Eine *affine Gleichung* ist ein Ausdruck der Form $\sum_{i=1}^{n} a_i x_i = b$ wobei $a_1, \ldots, a_n, b \in \mathbb{R}$ und $x_1, \ldots, x_n$ reelwertige Variablen sind. Eine *affine Ungleichung* ist ein Ausdruck der Form $\sum_{i=1}^{n} a_i x_i \leq b$

wobei $a_1, \ldots, a_n, b \in \mathbb{R}$ und $x_1, \ldots, x_n$ reelwertige Variablen sind. Eine *affine Funktion* $f : \mathbb{R}^n \to \mathbb{R}$ ist gegeben durch $f(x) = \sum_{i=1}^{n} a_i x_i + b$ wobei $a_1, \ldots, a_n, b \in \mathbb{R}$.

Definition 10.1. Ein *lineares Programm* in den Variablen $x_1, \ldots, x_n$ besteht aus einer endlichen Menge E von affinen Gleichungen und affinen Ungleichungen sowie einer affinen Funktion $f : \mathbb{R}^n \to \mathbb{R}$ und der Information ob f maximiert oder minimiert werden soll. Ein Vektor $x \in \mathbb{R}^n$ heißt *zulässige Lösung* falls x die Bedingungen E erfüllt. Eine zulässige Lösung x heißt *optimale Lösung* falls $f(x)$ optimal unter allen zulässigen Lösungen ist.

Die Funktion f wird auch als *Zielfunktion* bezeichnet. Für ein $x \in \mathbb{R}^n$ wird $f(n)$ auch als *Zielwert von* x bezeichnet. Zunächst kann man sich überlegen dass für ein gegebenes lineares Programm E, f einer der folgenden Fälle zutrifft.

1. E, f besitzt eine optimale Lösung mit einem Zielwert $z \in \mathbb{R}$. Das ist der interessanteste Fall der auch in Beispiel 10.1 auftritt. Man beachte, dass in diesem Fall zwar der Zielwert eindeutig ist, die Lösung aber im Allgemeinen nicht.

2. E, f besitzt keine zulässigen Lösungen. In diesem Fall heißt E, f *unlösbar*. Ein Beispiel für diesen Fall ist erhält man aus Beispiel 10.1 wenn (etwa aus einem weiteren Vertrag) eine Mindeststückzahl des ersten Produkts von z.B. 1000 Stück gefordert wäre.

3. E, f besitzt Lösungen mit beliebig großem (für Maximierung) bzw. beliebig kleinem (bei Minimierung) Zielwert. In diesem Fall heißt E, f *unbeschränkt*. Dieser Fall würde auftreten, wenn man in Beispiel 10.1 die limitierenden Zeit- und CO_2-Beschränkungen weglassen würde.

Wir können also das folgende Berechnungsproblem formulieren.

Lineare Optimierung
Eingabe: ein lineares Programm E, f
Ausgabe: "unbeschränkt" falls E, f unbeschränkt ist, "unlösbar" falls E, f unlösbar ist und eine optimale Lösung $x \in \mathbb{R}^n$ von E, f sonst

Es gibt polynomiale Algorithmen die dieses Problem lösen. Wir werden in Abschnitt 10.3 den Simplex-Algorithmus besprechen. Dieser ist zwar nicht polynomial, praktisch allerdings recht effizient, da für den Simplex-Algorithmus schlechte Eingaben in der Praxis selten sind. Außerdem ist der Simplex-Algorithmus in gewissem Sinn eine Verallgemeinerung des gaußschen Eliminationsverfahrens und alleine deswegen schon interessant.

10.2 Reduktionen auf lineare Optimierung

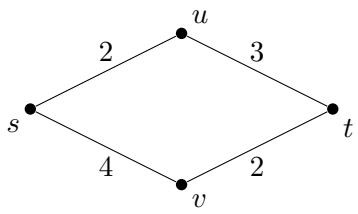
Auf den ersten Blick erweckt das Problem der linearen Optimierung vielleicht den Eindruck recht spezifisch für diese Art von Produktionsplanung und verwandte Situationen zu sein. In der Tat ist das Problem aber sehr allgemein. Um das zu illustrieren werden wir hier einige verschiedene Probleme auf lineare Optimierung reduzieren.

Kürzester Pfad. Gegeben ein zusammenhängender Graph $G = (V, E)$ und eine Kostenfunktion $c : E \to \mathbb{R}_{\geq 0}$ sowie einen Startknoten $s \in V$ und einen Zielknoten $t \in V$ wollen wir die Länge des kürzesten (d.h. billigsten) Pfades von s nach t bestimmen. In Abschnitt 7.4 haben wir den Algorithmus von Dijkstra kennengelernt, der einen solchen Pfad berechnet. Um dieses Problem als lineares Programm darzustellen führen wir für jedes $v \in V$ eine reelwertige Variable d_v ein, die die Länge des kürzesten Pfades von s nach v darstellen soll. Dann gilt

$$\begin{aligned} &d_s = 0 \quad \text{und} \\ &d_v \leq d_u + c(u, v) \quad \text{für alle } (u, v) \in E. \end{aligned}$$

Wenn wir unter diesen Bedingungen d_t maximieren, stellen wir sicher, dass implizit ein Pfad von s nach t ausgewählt wird, da für jedes d_v für $v = t, \ldots, s$ eine der Ungleichungen mit Gleichheit erfüllt wird. Jeder Knoten u der das erreicht ist der v vorausgehende auf einem kürzesten Pfad.

Beispiel 10.2. Der Graph



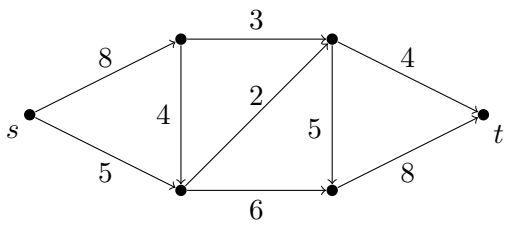
führt zu dem linearen Programm

$$\begin{aligned} &d_t \text{ max!} \\ &d_s = 0, d_u \leq d_s + 2, d_v \leq d_s + 4, d_t \leq d_u + 3, d_t \leq d_v + 2. \end{aligned}$$

von dem $(d_s, d_u, d_v, d_t) = (0, 2, 0, 5)$ eine optimale Lösung ist.

Maximaler Fluss. Wir betrachten jetzt das Problem des maximalen Flusses (in einem Transportnetzwerk). Das modellieren wir wie folgt: Wir gehen von einem gerichteten, zusammenhängenden Graphen $G = (V, E)$ aus sowie von einem ausgezeichneten Quellknoten $s \in V$ und einem ausgezeichneten Zielknoten $t \in V$. Weiters hat jede Kante eine gewisse (Transport-)kapazität die durch eine Funktion $c : E \to \mathbb{R}_{\geq 0}$ angegeben wird. Es gibt an den Knoten außer den im Graph angegebenen keine Möglichkeit der Entnahme oder Einspeisung. Weiters ist nirgendwo eine Lagerung möglich. Wir wollen so viel wie möglich von s nach t transportieren. Als Anwendungsbeispiel können wir uns etwa den Transport von Rohöl durch ein Netzwerk von Pipelines vorstellen.

Beispiel 10.3. Ein Beispiel für einen Graphen mit Kapazitätsbeschriftungen auf den Kanten ist:



Definition 10.2. Sei $G = (V, E)$ ein gerichteter Graph, $s, t, \in V$ und $c : E \to \mathbb{R}_{\geq 0}$ eine Kapazitätsfunktion. Ein *Fluss* für ist dann eine Funktion $f : E \to \mathbb{R}_{\geq 0}$ so dass

1. $f(e) \leq c(e)$ für alle $e \in E$ und
2. $\sum_{(u,v)\in E} f(u,v) = \sum_{(v,w)\in E} f(v,w)$ für alle $v \in V \in \{s,t\}$.

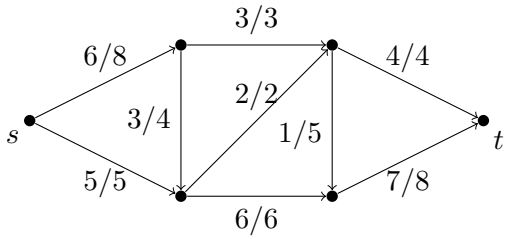
Das Problem der Bestimmung eines maximalen Flusses kann dann wie folgt formuliert werden.

Maximaler Fluss

Eingabe: gerichteter zusammenhängender Graph (V,E), $s,t \in V$, Kapazitätsfunktion $c : E \to \mathbb{R}_{\geq 0}$

Ausgabe: ein Fluss $f : E \to \mathbb{R}_{\geq 0}$ so dass $\sum_{(v,t)\in E} f(v,t)$ maximal ist

Beispiel 10.4. Ein maximaler Fluss des Graphen aus Beispiel 10.3 ist:



Zur Lösung dieses Problems mittels linearer Optimierung stellen wir einen Fluss f dar indem wir für jedes $e \in E$ eine reellwertige Variable f_e einführen, die $f(e)$ darstellen soll. Das lineare Programm ist dann:

$$0 \leq f_e \quad \text{für alle } e \in E$$
$$f_e \leq c(e) \quad \text{für alle } e \in E$$
$$\sum_{(u,v)\in E} f_{u,v} = \sum_{(v,w)\in E} f_{v,w} \quad \text{für alle } v \in V$$
$$\sum_{(v,t)\in E} f_{v,t} \text{ max!}$$

Es ist leicht zu sehen dass eine optimale Lösung dieses linearen Programms ein maximaler Fluss ist.

Schaltkreisevaluierung. Ein letztes Problem das wir nun auf lineare Optimierung reduzieren wollen ist die Schaltkreisevaluierung. Das Problem der Schaltkreisevaluierung ist **P**-vollständig[1]. Diese Reduktion zeigt also, dass auch lineare Optimierung **P**-vollständig ist, d.h., informell gesagt, dass jedes (mit einem beliebigen Algorithmus) in polynomialer Zeit lösbare Problem durch ein Verfahren für lineare Optimierung in polynomialer Zeit gelöst werden kann.

Definition 10.3. Ein Boolescher Schaltkreis ist ein endlicher gerichteter azyklischer Graph $G = (V,E)$ mit einer Knotenbeschriftung $l : V \to \{\textbf{wahr}, \textbf{falsch}, \wedge, \vee, \neg\}$ so dass $\mathrm{d}^-(v) \leq 2$ für alle $v \in V$ und:

[1]Wir geben hier keine formale Definition von **P**-Vollständigkeit und verweisen stattdessen auf Quellen über Komplexitätstheorie.

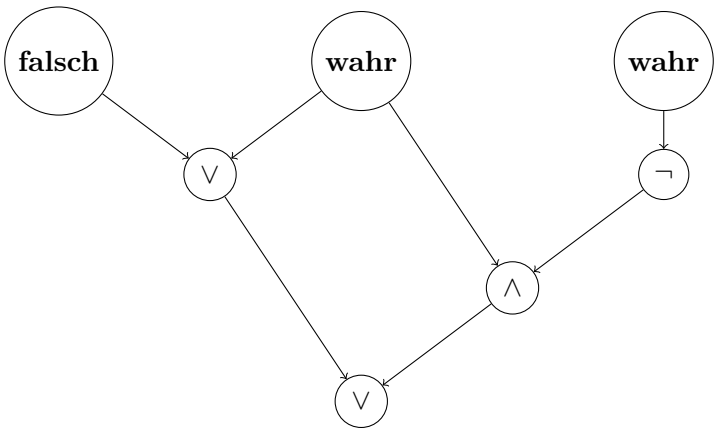
1. Falls $l(v) \in \{\textbf{wahr}, \textbf{falsch}\}$, dann ist $\mathrm{d}^-(v) = 0$.
2. Falls $l(v) = \neg$, dann ist $\mathrm{d}^-(v) = 1$.
3. Falls $l(v) \in \{\wedge, \vee\}$, dann ist $\mathrm{d}^-(v) = 2$.

Ein Knoten v eines Schaltkreises mit $\mathrm{d}^+(v) = 0$ heißt *Ausgabeknoten*. Für jeden Knoten v wird jetzt induktiv sein Wert $[\![v]\!]$ der Semantik der Booleschen Logik folgend definiert: Falls $l(v) = \textbf{wahr}$, dann $[\![v]\!] = 1$. Falls $l(v) = \textbf{falsch}$, dann $[\![v]\!] = 0$. Falls $l(v) \in \{\neg, \wedge, \vee\}$, dann ergibt sich $[\![v]\!]$ aus den folgenden Tabellen:

a	$\neg a$
0	1
1	0

a	b	$a \wedge b$	$a \vee b$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Beispiel 10.5. Der Schaltkreis

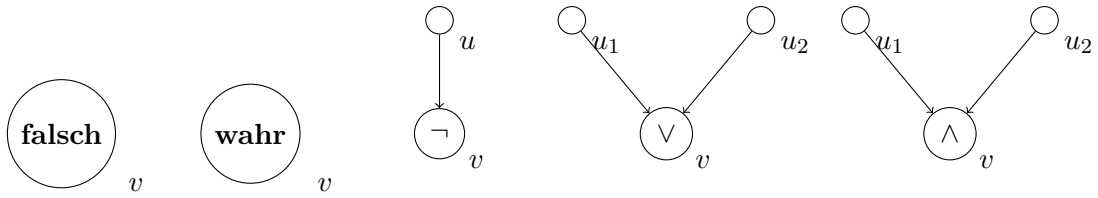


hat einen einzigen Ausgabeknoten v dessen Wert $[\![v]\!] = 1$ ist.

Das Berechnungsproblem der Schaltkreisevaluierung kann dann wie folgt formuliert werden:

Schaltkreisevaluierung
Eingabe: Ein Schaltkreis mit genau einem Ausgabeknoten
Ausgabe: Der Wert des Ausgabeknotens

Dieses Problem wird durch das folgende lineare Programm gelöst. Wir führen für jedes $v \in V$ eine Variable x_v ein, die den Wert von v repräsentieren soll. Für jedes x_v werden die Ungleichungen $0 \leq x_v$ und $x_v \leq 1$ aufgenommen. Für jeden Knoten werden weitere Ungleichungen wie folgt aufgenommen



$x_v = 0$	$x_v = 1$	$x_v = 1 - x_u$	$x_v \geq x_{u_1}$, $x_v \geq x_{u_2}$ $x_v \leq x_{u_1} + x_{u_2}$	$x_v \leq x_{u_1}$, $x_v \leq x_{u_2}$ $x_v \geq x_{u_1} + x_{u_2} - 1$

Dann lässt sich mit Induktion über den Schaltkreis nachweisen dass die einzige zulässige Lösung dieses linearen Programms jene ist mit $x_v = [\![v]\!]$ für alle $v \in V$. Insbesondere gilt dadurch $x_v \in \{0, 1\}$. Deshalb ist in diesem Fall auch irrelevant ob und was minimiert oder maximiert wird.

10.3 Der Simplex-Algorithmus

Wir kommen jetzt zurück zum Problem der lineare Optimierung. Zur seiner algorithmischen Behandlung ist es sinnvoll den Formalismus ein wenig einzuschränken.

Definition 10.4. Ein *lineares Programm in Standardform* besteht aus einer Matrix $A \in \mathbb{R}^{m \times n}$, einem Vektor $b \in \mathbb{R}^m$ und einem Vektor $c \in \mathbb{R}^n$.

Die Matrix A gemeinsam mit dem Vektor b spezifiziert E durch die Ungleichungen $Ax \leq b$. Der Vektor c spezifiziert f als $x \mapsto c^{\mathrm{T}}x$. Zusätzlich schränken wir uns bei einem linearen Programm in Standardform darauf ein f zu maximieren und nur solche Lösungen zu betrachten wo $x_i \geq 0$ für alle $i \in \{0, \ldots, n\}$. Wir definieren also

Definition 10.5. Sei A, b, c ein lineares Programm in Standardform. Ein $x \in \mathbb{R}^n$ heißt *zulässige Lösung* falls $Ax \leq b$ und $x_i \geq 0$ für alle $i \in \{0, \ldots, n\}$. Eine zulässige Lösung $x \in \mathbb{R}^n$ heißt *optimale Lösung* falls $c^{\mathrm{T}}x$ maximal unter allen zulässigen Lösungen ist.

Lineare Optimierung (Standardform)
Eingabe: ein lineares Programm A, b, c in Standardform
Ausgabe: "unbeschränkt" falls A, b, c unbeschränkt ist, "unlösbar" falls A, b, c unlösbar ist, eine optimale Lösung $x \in \mathbb{R}^n$ von A, b, c sonst

Definition 10.6. Zwei max-lineare Programme E, f und E', f' heißen *äquivalent* falls:

1. Für jede zulässige Lösung x von E existiert eine zulässige Lösung x' von E' mit $f(x) = f'(x')$.
2. Für jede zulässige Lösung x' von E' existiert eine zulässige Lösung x von E mit $f(x) = f'(x')$.

Ein min-lineares Programm E, f und ein max-lineares Programm E', f' heißen *äquivalent* falls:

1. Für jede zulässige Lösung x von E existiert eine zulässige Lösung x' von E' mit $f(x) = -f'(x')$.
2. Für jede zulässige Lösung x' von E' existiert eine zulässige Lösung x von E mit $f(x) = -f'(x')$.

Lemma 10.1. *Jedes lineare Programm kann in ein äquivalentes lineares Programm in Standardform transformiert werden.*

Beweis. Ein lineares Programm kann sich von einem linearen Programm in Standardform in den folgenden Aspekten unterscheiden: 1. Falls die Zielfunktion f einen konstanten Teil $d \in \mathbb{R}$

hat, dann fügen wir eine neue Variable v sowie die Gleichung $v = d$ hinzu und ersetzen d in f durch v. Das dadurch entstehende lineare Programm ist äquivalent zum ursprünglichen. 2. Falls die Zielfunktion f minimiert werden soll, wird stattdessen $-f$ maximiert. Diese beiden linearen Programme sind dann äquivalent. 3. Eine Variable v ohne die Bedingung $v \geq 0$ wird entfernt und durch zwei neue Variablen v^+ und v^- simuliert, indem jedes Vorkommen von v durch $v^+ - v^-$ ersetzt wird. Weiters werden die Bedingungen $v^+ \geq 0$ und $v^- \geq 0$ aufgenommen. Diese beiden linearen Programme sind äquivalent da jedes $a \in \mathbb{R}$ geschrieben werden kann als $a = a^+ - a^-$ mit $a^+, a^- \geq 0$. 4. Eine Gleichung der Form $f(\bar{x}) = g(\bar{x})$ für zwei affine Funktionen f und g wird ersetzt durch die beiden Ungleichungen $f(\bar{x}) \leq g(\bar{x})$ und $g(\bar{x}) \leq f(\bar{x})$. Diese beiden linearen Programme sind äquivalent. Schließlich werden die affinen Ungleichungen in der Form $Ax \leq b$ geschrieben. □

Wir machen nun einige grundlegende Beobachtungen über die Situation in deren Rahmen der Simplex-Algorithmus motiviert und geometrisch erklärt werden kann.

Zunächst beobachten wir dass jede der Ungleichungen eines linearen Programms in Standardform einen Halbraum von $\mathbb{R}^n$ definiert. Die Menge der zulässigen Lösungen ist also ein Schnitt von Halbräumen und damit ein *konvexes Polytop*.

Die zu maximierende Funktion $x \mapsto c^{\mathrm{T}}x$ induziert für festes $z \in \mathbb{R}$ die Hyperebene $z = c^{\mathrm{T}}x$. Der Maximierung von z entspricht eine Verschiebung der Hyperebene durch das Polytop der zulässigen Lösungen (vgl. dazu auch Beispiel 10.1 im $\mathbb{R}^2$). Falls das lineare Programm lösbar ist, endet diese Verschiebung mit einem z das eine Hyperebene definiert, die ein k-dimensionales Unterpolytop enthält, also einen Knoten, eine Kante, eine Seitenfläche, etc. des Polytops der zulässigen Lösungen[2]. Ein Unterpolytop enthält aber mindestens einen Knoten. Wir erkennen also: *Falls das lineare Programm lösbar ist, dann ist ein Knoten des Polytops eine optimale Lösung.*

Es reicht also, die Hyperebene von Knoten zu Knoten zu schieben. Die *Grundidee* für den Simplex-Algorithmus ist nun: wir starten an einem beliebigen Knoten des Polytops. In jedem Schritt bewegen wir uns zu einem benachbarten Knoten an dem der Zielwert mindestens so groß wie beim vorherigen ist.

Zur algebraischen Realisierung dieses Verfahrens führen wir zunächst sogenannte Schlupfvariablen ein. Damit wird eine Ungleichung der Form

$$\sum_{i=1}^{k} a_i x_i \leq b$$

transformiert zu

$$s = b - \sum_{i=1}^{k} a_i x_i \quad \text{und} \quad s \geq 0.$$

Dabei wird s als *Schlupfvariable* bezeichnet. Variablen die keine Schlupfvariablen sind, werden als *freie Variablen* bezeichnet.

Definition 10.7. Ein lineares Programm in *Schlupfform* in den Variablen $\{x_1, \ldots, x_n\}$ ist gegeben durch die Menge S der Indizes der Schlupfvariablen, die Menge F der Indizes der

[2] Falls es unlösbar ist, hat diese Verschiebung keinen Anfang, falls es unbeschränkt ist keine Ende.

freien Variablen mit $S \cap F = \emptyset$ und $S \cup F = \{1, \ldots, n\}$ sowie den Gleichungen

$$z = \nu + \sum_{j \in F} c_j x_j \text{ max!}$$
$$x_i = b_i - \sum_{j \in F} a_{i,j} x_j \text{ für } i \in S$$

und inkludiert implizit die Nichtnegativitätsbedingungen $x_1 \geq 0, \ldots, x_n \geq 0$.

Ein lineares Programm in Schlupfform ist also durch das Tupel (F, S, A, b, c, ν) vollständig spezifiziert.

Lemma 10.2. *Jedes lineare Programm in Standardform kann in ein äquivalentes lineares Programm in Schlupfform transformiert werden.*

Beweis. Durch wiederholte Einführung von Schlupfvariablen werden zum ursprünglichen Programm äquivalente lineare Programm erzeugt. Dieser Vorgang endet wenn nur noch Ungleichungen von der Form $x_i \geq 0$ übrig sind. □

Definition 10.8. Sei $L = F, S, A, b, c, \nu$ ein lineares Programm in Schlupfform in den Variablen $\{x_1, \ldots, x_n\}$. Dann gibt es genau ein $x \in \mathbb{R}^n$ so dass $x_i = 0$ für alle $i \in F$ und x alle Gleichungen erfüllt. Dieses $x \in \mathbb{R}^n$ wird als *Basislösung* von L bezeichnet. Falls x auch alle Nichtnegativitätsbedingungen erfüllt, dann heißt x *zulässige Basislösung*.

Über seine Basislösung spezifiziert eine Schlupfform also einen bestimmten Punkt im $\mathbb{R}^n$. Im Simplex-Verfahren werden wir (nach einer Phase der Initialisierung) nur solche Schlupfformen betrachten deren Basislösung zulässig ist. Auf diese Weise spezifiziert also eine Schlupfform einen Knoten des Polytops der zulässigen Lösungen. Der Simplex-Algorithmus realisiert nun die Bewegung von einem Knoten zu einem Nachbarknoten durch Transformation einer Schlupfform mit zulässiger Basislösung in eine andere Schlupfform mit zulässiger Basislösung. Diese Transformation geschieht durch Austausch einer freien Variablen x_f mit einer Schlupfvariablen x_s. In der neuen Basislösung ist dann nicht mehr $x_f = 0$, sondern $x_s = 0$. Wir illustrieren die Funktionsweise des Simplex-Algorithmus auf der algebraischen Ebene zunächst anhand eines Beispiels.

Beispiel 10.6. Wir wollen auf das folgende lineare Programm in Standardform den Simplex-Algorithmus anwenden:

$$z = x_1 + 2x_2 \text{ max!}$$
$$3x_1 + x_2 \leq 15$$
$$x_1 + 4x_2 \leq 16$$

Dieses lineare Programm wird in Abbildung 10.2 graphisch dargestellt. Zunächst bringen wir dieses lineare Programm in Schlupfform:

$$z = x_1 + 2x_2$$
$$x_3 = 15 - 3x_1 - x_2$$
$$x_4 = 16 - x_1 - 4x_2$$

wobei $F = \{1, 2\}$, $S = \{3, 4\}$ und $x_1, x_2, x_3, x_4 \geq 0$ implizit sind. Die Basislösung dieses Programms ist $(0, 0, 15, 16)$. Der Vektor $(0, 0, 15, 16)$ ist eine zulässige Basislösung. Seine Einschränkung auf die ersten beiden Komponenten ist der Vektor $(x_1, x_2) = (0, 0)$ der den aktuellen Knoten des Polytops im ursprünglichen Problem angibt, siehe Abbildung 10.2.

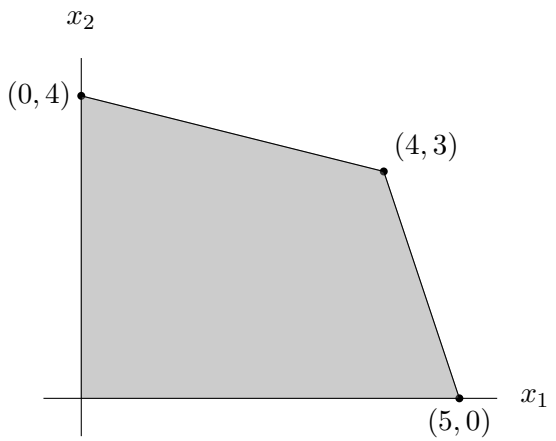


Abbildung 10.2: Beispiel für den Simplex-Algorithmus

Da x_2 in der Zielfunktion den höchsten positiven Koeffizienten hat, wählen wir x_2 zum Vertauschen mit einer Schlupfvariablen. Aus der Gleichung von x_3 folgt dass $x_2 \leq 15$ ist. Aus der Gleichung von x_4 folgt dass $x_2 \leq 4$ sein muss. In Hinblick darauf, in der nächsten Basislösung $x_2 = 4$ und damit $x_4 = 0$ zu setzen vertauschen wir also x_2 und x_4. Dazu schreiben wir zunächst die Gleichung von x_4 zu $x_2 = 4 - \frac{1}{4}x_1 - \frac{1}{4}x_4$ um und ersetzen dadurch in allen anderen Gleichungen x_2. So erhalten wir das neue lineare Programm

$$\begin{aligned} z &= 8 + \frac{1}{2}x_1 - \frac{1}{4}x_4 \\ x_2 &= 4 - \frac{1}{4}x_1 - \frac{1}{4}x_4 \\ x_3 &= 11 - \frac{11}{4}x_1 + \frac{1}{4}x_4 \end{aligned}$$

wobei $F = \{1, 4\}$ und $S = \{2, 3\}$ ist. Die Basislösung dieses Programms ist $(0, 4, 15, 0)$ was dem Punkt $(x_1, x_2) = (0, 4)$ entspricht, einem benachbarten Knoten des Polytops der zulässigen Lösungen.

Nun hat x_1 in der Zielfunktion den höchsten positiven Koeffizienten. Aus der Gleichung von x_2 folgt dass $x_1 \leq 16$ ist. Aus der Gleichung von x_3 folgt dass $x_1 \leq 4$ ist. Wir vertauschen also x_1 und x_3. Wir haben $x_1 = 4 - \frac{4}{11}x_3 + \frac{1}{11}x_4$ und damit erhalten wir das neue Programm

$$\begin{aligned} z &= 10 - \frac{4}{22}x_3 - \frac{9}{44}x_4 \\ x_1 &= 4 - \frac{4}{11}x_3 + \frac{1}{11}x_4 \\ x_2 &= 3 + \frac{1}{11}x_3 - \frac{12}{44}x_4 \end{aligned}$$

wobei $F = \{3, 4\}$, $S = \{1, 2\}$. Die Basislösung dieses Programms ist $(4, 3, 0, 0)$ was dem Punkt $(x_1, x_2) = (4, 3)$ entspricht.

Da nun alle Koeffizienten in der Zielfunktion negativ sind, kann durch Erhöhung der freien Variablen der Zielwert nicht mehr erhöht werden. Wir haben also eine optimale Lösung gefunden: $x_1 = 4$, $x_2 = 3$ und $z = 10$.

Der Simplex-Algorithmus kann als Pseudocode wie in Algorithmus 46 formuliert werden. Dabei

Algorithmus 46 Der Simplex-Algorithmus

Prozedur SIMPLEX(A, b, c)
  $(F, S, A, b, c, \nu) :=$ INITSIMPLEX(A, b, c)
  Solange $\exists j \in F$ mit $c_j > 0$
    Wähle $f \in F$ mit $c_f > 0$
    $\Delta :=$ BESCHRÄNKUNGEN(S, A, b, f)
    Sei $s \in S$ so dass $\Delta[s]$ minimal ist
    Falls $\Delta[s] = \infty$ **dann**
      Antworte "unbeschränkt"
    sonst
      $(F, S, A, b, c, \nu) :=$ AUSTAUSCH$(F, S, A, b, c, \nu, s, f)$
    Ende Falls
  Ende Solange
  Antworte BASISLÖSUNG(F, S, A, b, c, ν)
Ende Prozedur

werden die folgenden Unterprozeduren verwendet:

Die Prozedur BESCHRÄNKUNGEN(S, A, b, f) liefert ein Datenfeld Δ der Länge $n = |F| + |S|$ zurück so dass, für alle $s \in S$ der Wert $\Delta[s]$ angibt welchen Wert x_f maximal annehmen kann, ohne die Schlupfvariable x_s kleiner als 0 zu machen.

Die Prozedur AUSTAUSCH$(F, S, A, b, c, \nu, s, f)$ löst die Gleichung von x_s nach x_f auf und setzt diese Darstellung von x_f in alle anderen Gleichungen ein. Dadurch erzeugt sie eine neue Schlupfform die sich von F, S, A, b, c, ν dadurch unterscheidet dass s von einer Schlupfvariable zu einer freien Variable geworden ist und f von einer freien Variable zu einer Schlupfvariable. Diese neue Schlupfform hat die selbe Menge zulässiger Lösungen und die selbe Zielfunktion, lediglich deren Darstellung wurde verändert. Diese neue Schlupfform wird dann von AUSTAUSCH zurückgegeben.

Zunächst zeigen wir die partielle Korrektheit (d.h. ohne Betrachtung der Termination) der Simplex-Schleife. Mit Simplex-Schleife ist hier die Schleife in Algorithmus 46 gemeinsam mit der letzten Zeile, die mit der aktuellen Basislösung antwortet, gemeint. Dazu stellen wir zuerst noch eine Überlegung zur optimalen Lösung an.

Definition 10.9. Sei $A \subseteq \mathbb{R}^n$ und $f : A \to \mathbb{R}$. Ein $x \in A$ heißt *(globales) Optimum* von f falls $f(y) \leq f(x)$ für alle $y \in A$. Ein $x \in A$ heißt *lokales Optimum* von f falls es eine Umgebung U von x gibt so dass $f(y) \leq f(x)$ für alle $y \in U \cap A$.

Lemma 10.3. *Sei $A \subseteq \mathbb{R}^n$ konvex, $f : A \to \mathbb{R}$ linear und x ein lokales Optimum von f, dann ist x auch ein globales Optimum.*

Beweis. Sei U eine Umgebung von x so dass $f(y) \leq f(x)$ für alle $y \in U \cap A$. Sei $z \in A$. Dann existiert aufgrund der Konvexität von A ein $t \in (0, 1)$ so dass $tx + (1 - t)z \in U \cap A$. Damit ist $f(x) \geq f(tx + (1 - t)z)$ und, aufgrund der Linearität von f, $f(x) \geq tf(x) + (1 - t)f(z)$. Also $f(x) \geq f(z)$. □

Lemma 10.4. *Falls $L = F, S, A, b, c, \nu$ eine Schlupfform mit zulässiger Basislösung ist und die Simplex-Schleife, angewandt auf L, terminiert dann gilt:*

1. *Falls L unbeschränkt ist, dann antwortet die Simplex-Schleife mit "unbeschränkt".*

2. *Falls L eine optimale Lösung mit Zielwert $z \in \mathbb{R}$ hat, dann liefert die Simplex-Schleife eine Lösung mit Zielwert z.*

Beweis. Die Simplex-Schleife kann nur entweder mit "unbeschränkt" oder mit einer Lösung antworten. Wir beginnen mit den folgenden beiden Beobachtungen: 1. Falls die Simplex-Schleife mit einer Lösung x antwortet, dann ist x ein lokales Optimum weil alle $c_i \leq 0$ sind und damit wegen Lemma 10.3 auch ein globales Optimum. 2. Falls die Simplex-Schleife mit "unbeschränkt" antwortet, dann ist $(x_1, \ldots, x_{f-1}, a, x_{f+1}, \ldots, x_{|S|+|F|})$ für alle $a \geq x_f$ zulässig. Da $c_f > 0$ ist damit L unbeschränkt.

Falls also L unbeschränkt ist, dann kann wegen 1. die Simplex-Schleife nicht mit einer Lösung antworten (sonst hätte L ja eine optimale Lösung), also antwortet L mit "unbeschränkt". Falls L eine optimale Lösung hat, dann kann wegen 2. die Simplex-Schleife nicht mit "unbeschränkt" antworten (sonst wäre L ja unbeschränkt), also antwortet L mit einer Lösung, die nach 1., eine optimale Lösung ist. □

Wir wenden uns nun der Initialisierung zu. Die Prozedur InitSimplex transformiert ein gegebenes lineares Programm in Standardform in eines in Schlupfform dessen Basislösung zulässig ist oder liefert "unlösbar" zurück falls das Eingabeproblem unlösbar ist. Zunächst kann man leicht beobachten, dass die Basislösung einer Schlupfform zulässig ist genau dann wenn $b \geq 0$ ist. Für die Initialisierung des Simplex-Verfahrens (Phase 1) ist also vor allem der Fall wo $b \neq 0$ ist interessant. In diesem Fall gehen wir wie folgt vor:

Definition 10.10. Sei $L = A, b, c$ ein lineares Programm in Standardform mit $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$ und $c \in \mathbb{R}^n$ so dass $b \not\geq 0$. Sei $k \in \{1, \ldots, m\}$ so dass b_k minimal ist. Dann definieren wir das lineare Programm S_L in Schlupfform in den Variablen $\{x_0, \ldots, x_{m+n}\}$ mit $F = \{1, \ldots, n, n+k\}$ und $S = \{0, n+1, \ldots, n+k-1, n+k+1, \ldots, n+m\}$ als:

$$z = b_k - \sum_{j=1}^{n} a_{k,j} x_j - x_{n+k} \text{ max!}$$

$$x_0 = -b_k + \sum_{j=1}^{n} a_{k,j} x_j + x_{n+k}$$

$$x_{n+i} = (b_i - b_k) + \sum_{j=1}^{n} (a_{k,j} - a_{i,j}) x_j + x_{n+k} \quad \text{für } i \in \{1, \ldots, k-1, k+1, \ldots, m\}$$

Lemma 10.5. *Sei $L = A, b, c$ ein lineares Programm mit $b \not\geq 0$, dann ist die Basislösung von S_L zulässig. Weiters ist L lösbar genau dann wenn S_L eine optimale Lösung hat mit $x_0 = 0$.*

Beweis. Zunächst beobachten wir dass die Basislösung von S_L zulässig ist, da $-b_k \geq 0$ und, für alle $i \in \{1, \ldots, m\}$ gilt dass $b_i \geq b_k$ und damit $b_i - b_k \geq 0$.

Um die zweite Behauptung zu zeigen, transformieren wir zunächst $L =$

$$z = \sum_{j=1}^{n} c_j x_j \text{ max!}$$

$$\sum_{j=1}^{n} a_{i,j} x_j \leq b_i \quad \text{für } i = 1, \ldots, m$$

$$x_j \geq 0 \quad \text{für } j = 1, \ldots, n$$

in das folgende Hilfsprogramm L' in Standardform:

$$z = -x_0 \,\text{max!}$$
$$\sum_{j=1}^{n} a_{i,j}x_j - x_0 \leq b_i \quad \text{für } i = 1, \ldots, m$$
$$x_j \geq 0 \quad \text{für } j = 0, \ldots, n$$

Zunächst sieht man dass L' lösbar ist: für beliebige $x_1, \ldots, x_n \geq 0$ wählen wir einfach x_0 hinreichend groß um alle Ungleichungen zu erfüllen. Weiters gilt: L ist lösbar genau dann wenn L' eine optimale Lösung mit $x_0 = 0$ hat. Falls nämlich L lösbar ist, dann erhält man durch Setzung von $x_0 = 0$ eine Lösung von L' und diese Lösung ist optimal da ja $x_0 \geq 0$ sein muss. Für die Gegenrichtung sei $(0, x_1, \ldots, x_n)$ optimale Lösung von L', dann ist $(x_1, \ldots, x_n)$ auch Lösung von L.

Die Schlupfform von L' ist:

$$z = -x_0 \,\text{max!}$$
$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{i,j}x_j + x_0 \quad \text{für } i = 1, \ldots, m$$
$$x_j \geq 0 \quad \text{für } j = 0, \ldots, n+m$$

mit $F = \{0, 1, \ldots, n\}$ und $S = \{n+1, \ldots, n+m\}$. Ein Austausch von x_0 mit x_{n+k} führt dann über die Gleichung $x_0 = -b_k + \sum_{j=1}^{n} a_{k,j}x_j + x_{n+k}$ zu S_L. □

Auf Basis dieses Resultats kann jetzt die Prozedur InitSimplex(A, b, c) wie in Algorithmus 47 angegeben werden, wobei wir davon ausgehen dass die Prozedur Simplex-Schleife mit einer Schlupfform als Ergebnis antwortet. Damit ist das Simplex-Verfahren vollständig spezifiziert

Algorithmus 47 Initialisierung des Simplex-Algorithmus

```
Prozedur InitSimplex(A, b, c)
    Falls b ≥ 0 dann
        Antworte Schlupfform(A, b, c)
    sonst
        U := Simplex-Schleife(S_{A,b,c})
        (x_0, ..., x_{n+m}) := Basislösung(U)
        Falls x_0 = 0 dann
            U' := Transformiere U in Schlupfform von A, b, c
            Antworte U'
        sonst
            Antworte "unlösbar"
        Ende Falls
    Ende Falls
Ende Prozedur
```

und wir können nun seine partielle Korrektheit beweisen.

Satz 10.1. *Sei $L = A, b, c$ ein lineares Programm in Standardform so dass* Simplex(A, b, c) *terminiert, dann gilt:*

1. *Falls L unbeschränkt ist, dann antwortet* Simplex(A, b, c) *mit "unbeschränkt".*

2. *Falls L unlösbar ist, dann antwortet* $\textsc{Simplex}(A, b, c)$ *mit "unlösbar".*

3. *Falls L eine optimale Lösung mit Zielwert $z \in \mathbb{R}$ hat, dann liefert* $\textsc{Simplex}(A, b, c)$ *eine Lösung mit Zielwert z.*

Beweis. Da $\textsc{Simplex}(A, b, c)$ terminiert, terminiert auch die Simplex-Schleife im Aufruf $\textsc{InitSimplex}(A, b, c)$ mit, nach Lemma 10.4, der optimalen Lösung $x = (x_0, \ldots, x_{n+m})$ von S_L. Falls L unlösbar ist, dann ist nach Lemma 10.5 $x_0 \neq 0$ und damit antwortet die Prozedur $\textsc{InitSimplex}(A, b, c)$ mit "unlösbar" und das Verfahren wird abgebrochen. Falls L lösbar ist, dann ist nach Lemma 10.5 $x_0 = 0$ und damit erzeugt die Prozedur $\textsc{InitSimplex}(A, b, c)$ eine Schlupfform von A, b, c mit zulässiger Basislösung. Da $\textsc{Simplex}(A, b, c)$ terminiert, terminiert nun auch die Simplex-Schleife und damit folgt dieser Satz unmittelbar aus Lemma 10.4. □

Offen ist jetzt noch die Termination des Simplex-Verfahrens. Dazu wollen wir uns zunächst überlegen wie es passieren kann, dass der Zielwert nicht strikt steigt. Da nur solche $f \in F$ ausgewählt werden die $c_f > 0$ haben kann dieser Fall nur eintreten wenn eine Gleichung für ein $s \in S$ gemeinsam mit ihrer Nichtnegativitätsbedingung $x_s \geq 0$ die Variable x_f so einschränkt dass x_f gar nicht erhöht werden kann. In diesem Fall muss $b_s = 0$ sein und wir sprechen von einer degenerierten Schlupfform.

Beispiel 10.7. Betrachten wir zum Beispiel

$$\begin{aligned} z &= 3 - \frac{1}{2}x_1 + 2x_2 \\ x_3 &= 1 - \frac{1}{2}x_1 \\ x_4 &= x_1 - x_2 \end{aligned}$$

Diese Schlupfform hat die zulässige Basislösung $(0, 0, 1, 0)$ mit Zielwert 3. Dann wird die freie Variable x_2 zum Vertauschen ausgewählt. Nun schränkt aber die Gleichung und Nichtnegativitätsbedingung von x_4 die Variable x_2 auf $x_2 \leq 0$ ein. Wir erhalten durch den Austausch die Schlupfform

$$\begin{aligned} z &= 3 + \frac{3}{2}x_1 - 2x_4 \\ x_2 &= x_1 - x_4 \\ x_3 &= 1 - \frac{1}{2}x_1 \end{aligned}$$

die ebenfalls die zulässige Basislösung $(0, 0, 1, 0)$ mit Zielwert 3 hat. Nun muss die freie Variable x_1 ausgewählt werden die durch die Gleichung von x_3 am stärksten eingeschränkt wird. Nach Durchführung dieser Vertauschung erhalten wir die Schlupfform

$$\begin{aligned} z &= 6 - 3x_3 - 2x_4 \\ x_1 &= 2 - 2x_3 \\ x_2 &= 2 - 2x_3 - x_4 \end{aligned}$$

mit Basislösung $(2, 2, 0, 0)$ und Zielwert 6 wobei es sich, da alle $c_j \leq 0$ sind, um eine optimale Lösung handelt.

Falls Zwischenschritte auftreten, in denen sich der Wert der Zielfunktion nicht verändert, spricht man von *stalling*. Das tritt in der Praxis häufig auf und ist (bis auf den Zeitaspekt) kein Problem

solange der Algorithmus zu einem späteren Zeitpunkt wieder ein Schritt durchführt der den Zielwert strikt erhöht. Allerdings kann es auch dazu kommen, das der Algorithmus kreist, d.h. dass er von einer Schlupfform U nach einigen Schritten die den Zielwert nicht erhöhen wieder bei U ankommt. Dieses Verhalten, wenn auch theoretisch möglich, tritt in der Praxis so selten auf, dass die meisten Implementierungen es gar nicht behandeln.

Beispiel 10.8. Die Schlupfform

$$\begin{aligned} z &= 10x_1 - 57x_2 - 9x_3 - 24x_4 \\ x_5 &= -\frac{1}{2}x_1 + \frac{11}{2}x_2 + \frac{5}{2}x_3 - 9x_4 \\ x_6 &= -\frac{1}{2}x_1 + \frac{3}{2}x_2 + \frac{1}{2}x_3 - x_4 \\ x_7 &= 1 - x_1 \end{aligned}$$

kann mit den Austauschschritten $x_1 \leftrightarrow x_5$, $x_2 \leftrightarrow x_6$, $x_3 \leftrightarrow x_1$, $x_4 \leftrightarrow x_2$, $x_5 \leftrightarrow x_3$, $x_4 \leftrightarrow x_6$ wieder auf sich selbst zurückgeführt werden.

Es gibt verschiedene Strategien um Kreisen zu verhindern, z.B. die Regel von Bland die darin besteht bei mehreren Möglichkeiten für die Auswahl eines x_i immer jene zu bevorzugen wo i minimal ist. Mit einer derartigen Erweiterung kann die Terminierung des Simplex-Verfahren sichergestellt werden. Für dieses Verfahren gilt dann eine entsprechend stärkere Form von Satz 10.1.

Kapitel 11

Geometrische Algorithmen

11.1 Elementare Begriffe und Operationen

Seien $p, q \in \mathbb{R}^n$, dann ist die *Strecke* $\overline{pq}$ definiert als $\overline{pq} = \{(1-t)p+tq \mid t \in [0,1]\}$. Die *gerichtete Strecke* $\overrightarrow{pq}$ ist definiert als $\overline{pq}$ gemeinsam mit der Information dass p der Startpunkt und q der Endpunkt ist. Somit ist $\overline{pq} = \overline{qp}$ aber (außer im Fall $p = q$) $\overrightarrow{pq} \neq \overrightarrow{qp}$. Eine *gerichtete Gerade* ist eine Gerade gemeinsam mit einem auf der Gerade liegenden Richtungsvektor.

Das Kreuzprodukt zweier Punkte $p_1 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$ und $p_2 = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}$ ist definiert als

$$p_1 \times p_2 = \det \begin{pmatrix} x_1 & x_2 \\ y_1 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1.$$

Es gibt den vorzeichenbehafteten Flächeninhalt des von den Vektoren p_1 und p_2 vom Ursprung aus aufgespannten Parallelogramms an, siehe Abbildung 11.1. Der Flächeninhalten ist in dem Sinn vorzeichenbehaftet als gilt: $p_1 \times p_2 > 0$ genau dann wenn p_2 auf der linken Seite der durch $\overrightarrow{0p_1}$ führenden gerichteten Gerade ist, $p_1 \times p_2 < 0$ genau dann wenn p_2 auf der rechten Seite der durch $\overrightarrow{0p_1}$ führenden gerichteten Gerade ist und $p_1 \times p_2 = 0$ genau dann wenn 0, p_1 und p_2 auf einer Gerade liegen, siehe Algorithmus 48. In Abbildung 11.2 ist die durch $\overrightarrow{0p_1}$ führende gerichtete Gerade sowie deren linke Seite (in grau) eingezeichnet. Mit Hilfe des Kreuzprodukts kann zum Beispiel festgestellt werden ob drei gegebene Punkte p_1, p_2, p_3 eine Linkskurve oder eine Rechtskurve bilden Dazu berechnen wir einfach $d = (p_2 - p_1) \times (p_3 - p_1)$ und überprüfen ob $d > 0$, $d < 0$ oder $d = 0$ gilt, sh Abbildung 11.3.

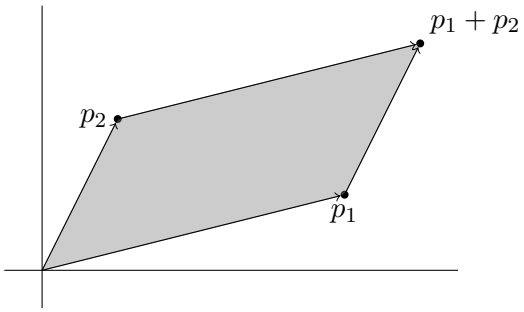


Abbildung 11.1: Das Kreuzprodukt

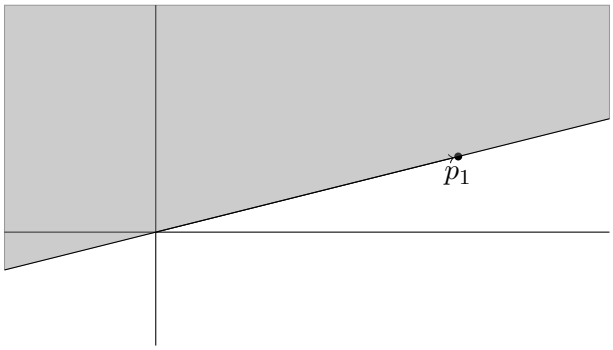


Abbildung 11.2: Die linke Seite einer gerichteten Geraden

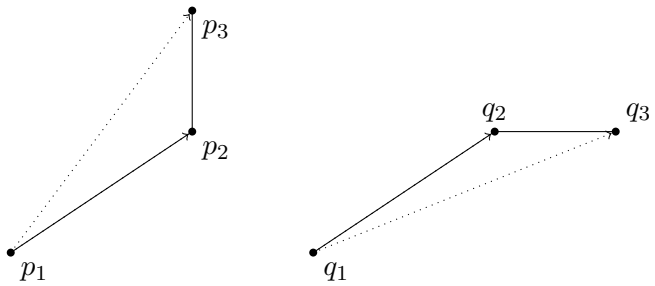


Abbildung 11.3: p_1, p_2, p_3 in einer Linkskurve und q_1, q_2, q_3 in einer Rechtskurve

Algorithmus 48 Relative Lage dreier Punkte

Prozedur LINKSKURVE(p_1, p_2, p_3)
  Antworte $(p_2 - p_1) \times (p_3 - p_1) > 0$
Ende Prozedur
Prozedur RECHTSKURVE(p_1, p_2, p_3)
  Antworte $(p_2 - p_1) \times (p_3 - p_1) < 0$
Ende Prozedur
Prozedur KOLLINEAR(p_1, p_2, p_3)
  Antworte $(p_2 - p_1) \times (p_3 - p_1) = 0$
Ende Prozedur

Diese Vorgehensweise ist effizienter (und numerisch stabiler) als die Berechnung des Winkels an p_2 mit trigonometrischen Funktionen.

11.2 Bestimmung der konvexen Hülle

Definition 11.1. Sei $P \subseteq \mathbb{R}^2$. Die konvexe Hülle von P, geschrieben als $\text{conv}(P)$ ist die kleinste konvexe Menge die P enthält.

Wir wollen in diesem Abschnitt Algorithmen betrachten, die die konvexe Hülle einer endlichen Menge von Punkten $P \subseteq \mathbb{R}^2$ berechnen. Der Rand einer solchen konvexen Hülle ist ein Polygon. Wir können dieses entweder als Liste von Kanten oder Liste von Knoten darstellen. Um dieses Polygon eindeutig zu definieren vereinbaren wir für den Spezialfall dass $k \geq 3$ kollineare Punkte aus P auf dem Rand liegen, dass nur die äußersten beiden als Knoten des Polygons aufzufassen sind. Ist $P = \{p_1, \ldots, p_n\}$ und $Q = \{p_{i_1}, \ldots, p_{i_k}\}$ die Knoten des Polygons, dann ist

$$\text{conv}(P) = \{\sum_{j=1}^{k} t_j p_{i_j} \mid t_1, \ldots, t_j \in \mathbb{R} \text{ mit } \sum_{j=1}^{k} t_j = 1\}.$$

Wir können also jetzt das Berechnungsproblem an dem wir interessiert sind präzise machen:

Konvexe Hülle
Eingabe: Eine endliche Menge $P \subseteq \mathbb{R}^2$
Ausgabe: Die Teilmenge $Q \subseteq P$ der Knoten des Polygons das der Rand von $\text{conv}(P)$ ist

Von nun an sprechen wir nur noch von Knoten und Kanten *der konvexen Hülle von P*. Über die konvexe Hülle können wir jetzt eine erste wichtige Beobachtung machen. Dazu treffen wir die Vereinbarung dass das Polygon (sowohl in der Kanten- als auch in der Knotendarstellung) im Uhrzeigersinn angegeben wird.

Lemma 11.1. *Sei $P = \{p_1, \ldots, p_n\} \subseteq \mathbb{R}^2$, dann ist $\overrightarrow{p_i p_j}$ eine Kante der konvexen Hülle von P genau dann wenn für alle $k \in \{1, \ldots, n\} \setminus \{i, j\}$ gilt: $p_k \in P$ liegt auf $\overline{p_i p_j}$ oder strikt auf der rechten Seite der gerichteten Gerade die $\overrightarrow{p_i p_j}$ enthält.*

Auf Basis dieses Lemmas könnte man sofort einen naiven Algorithmus angeben, der alle Kanten der konvexen Hülle in Zeit $O(n^3)$ bestimmt, indem er die Bedingung für alle $(i, j, k) \in \{1, \ldots, n\}^3$ überprüft.

Ein besserer Algorithmus zur Bestimmung der konvexen Hülle ist der *Jarvis-Marsch* (auch "gift-wrapping Algorithmus" genannt). Dieser Algorithmus basiert auf ebenfalls auf obigem Lemma, geht aber iterativ vor. Ist nämlich ein Punkt q der konvexen Hülle von P gegeben, ist der nächste Punkt q' jener mit der Eigenschaft dass alle $p \in P \setminus \{q, q'\}$ auf $\overline{qq'}$ oder strikt rechts von der gerichteten Gerade die $\overrightarrow{qq'}$ enthält liegen. Aus q kann q' bestimmt werden indem alle Punkte durchlaufen werden und dabei möglichst nach links gegangen wird. Dieser Schritt der Bestimmung eines nächsten Punkts wird wiederholt, bis wieder der Ausgangspunkt erreicht wurde. Es bleibt einen Ausgangspunkt zu finden, von dem garantiert werden kann dass er sich auf der konvexen Hülle befindet. Dafür reicht es z.B. den eindeutig bestimmten $\leq_{\text{lex}}$-minimalen Punkt zu verwenden. Diese Vorgehensweise ist in Algorithmus 49 ausformuliert. Die Zeitkomplexität

Algorithmus 49 Jarvis-Marsch

```
Prozedur FindeNächsten(P, q)
    q_neu := P[1]
    Für i := 2, ..., P.Länge
        Falls Linkskurve(q, q_neu, P[i]) dann
            q_neu := P[i]
        sonst falls Kollinear(q, q_neu, P[i]) und |P[i] − q| > |q_neu − q| dann
            q_neu := P[i]
        Ende Falls
    Ende Für
    Antworte q_neu
Ende Prozedur

Prozedur Jarvis(P)
    Sei Q neue Liste, leer initialisiert
    q_Start := ≤_lex -minimaler Punkt von P
    Q.Hinzufügen(q_Start)
    Wiederhole
        Q.Hinzufügen(FindeNächsten(P, Q.Ende))
    bis Q.Ende = q_Start
    Antworte Q
Ende Prozedur
```

der Prozedur Jarvis(P) ist $\Theta(nh)$ wobei h die Anzahl der Punkte auf der konvexen Hülle von P sind. Bei der Prozedur FindeNächsten handelt es sich im Wesentlichen um die Suche nach einem maximalen Winkel (ohne dass dabei jemals ein Winkel explizit berechnet werden müsste).

Wir wollen nun noch einen weiteren Algorithmus zur Berechnung der konvexen Hülle betrachten, der sich auf ein wichtiges Designprinzip für geometrische Algorithmen stützt: das *Sweep line*-Prinzip[1]. Dabei handelt es sich um Algorithmen, die gegebene Information im $\mathbb{R}^2$ in einer Richtung, z.B. entlang steigender x-Koordinate, verarbeiten. Hier werden wir die konvexe Hülle in zwei Teilen berechnen, zuerst die obere Hülle, danach die untere Hülle, diese beiden Listen von Punkten werden dann durch Zusammenhängen und Entfernung der Duplikate an Anfang und Ende kombiniert. Für dieses Verfahren verwenden wir eine Stapel, der Operationen zur Verfügung stellt die in konstanter Zeit das letzte, das vorletzte und das vorvorletzte Element zurückgeben, siehe Algorithmus 50. Die Korrektheit der Prozedur ObereHülle kann mit der Schleifeninvariante gezeigt werden, dass S die obere Hülle von $P[1], \ldots, P[i-1]$ ist.

Die Laufzeit von ObereHülle sowie von UntereHülle ist jeweils $\Theta(n)$ da kein Punkt zwei Mal auf dem Stapel landet, jeder Punkt wird also einmal hinzugefügt und höchstens einmal entfernt. Die Sortierung benötigt Laufzeit $O(n \log n)$ so dass dieser Algorithmus insgesamt Laufzeit $O(n \log n)$ benötigt. Ob der Sweep line-Algorithmus gegenüber dem Jarvis-Marsch zu bevorzugen ist hängt vom Verhältnis zwischen h und $\log n$ ab, das sich aus der Verteilung der Punkte ergibt. Ein Vorteil des Sweep line-Algorithmus besteht darin, dass er nur Zeit $O(n)$ benötigt falls die Punkte in P bereits in lexikographischer Sortierung vorliegen.

[1] Diese Terminologie kommt vom Kehren (engl. *to sweep*) mit einem Besen.

Algorithmus 50 Sweep line-Algorithmus zur Bestimmung der konvexen Hülle

```
Prozedur KonvexeHülle(P)
    P := Sortiere P nach ≤_lex
    O := ObereHülle(P)
    U := UntereHülle(P)
    Antworte Kombiniere O und U
Ende Prozedur

Prozedur ObereHülle(P)
    Sei S leerer Stapel
    S.Hinzufügen(P[1])
    S.Hinzufügen(P[2])
    Für i := 3, ..., P.Länge
        S.Hinzufügen(P[i])
        Solange |S| ≥ 3 und ¬Rechtskurve(S.Vorvorletzter, S.Vorletzter, S.Letzter)
            S.EntferneVorletzten
        Ende Solange
    Ende Für
    Antworte S
Ende Prozedur

Prozedur UntereHülle(P)
    Sei S leerer Stapel
    S.Hinzufügen(P[P.Länge])
    S.Hinzufügen(P[P.Länge − 1])
    Für i := P.Länge − 2, ..., 1
        S.Hinzufügen(P[i])
        Solange |S| ≥ 3 und ¬Rechtskurve(S.Vorvorletzter, S.Vorletzter, S.Letzter)
            S.EntferneVorletzten
        Ende Solange
    Ende Für
    Antworte S
Ende Prozedur
```

Problemverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[1] Johannes A. Buchmann. *Einführung in die Kryptographie, 6. Auflage.* Springer, 2016.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 3rd edition, 2009.

[3] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh V. Vazirani. *Algorithms.* McGraw-Hill, 2008.

[4] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. *Computational geometry: algorithms and applications, 3rd Edition.* Springer, 2008.

[5] Jon M. Kleinberg and Éva Tardos. *Algorithm design.* Addison-Wesley, 2006.

[6] Markus Nebel. *Entwurf und Analyse von Algorithmen.* Springer Vieweg, 2012.

[7] Thomas Ottmann and Peter Widmayer. *Algorithmen und Datenstrukturen, 5. Auflage.* Spektrum Akademischer Verlag, 2012.

[8] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms.* Addison-Wesley, 2012.

[9] Robert J. Vanderbei. *Linear Programming: Foundations and Extensions.* Springer, 4th edition, 2014.