Skript[1] zur Vorlesung „Theoretische Informatik 1“

Prof. Dr. Georg Schnitger

WS 2012/13

[1]Herzlichen Dank an Dr. Maik Weinard für viele wichtige Beiträge zu diesem Skript!

Hinweise auf Fehler und Anregungen zum Skript bitte an

besser@thi.cs.uni-frankfurt.de

oder

matthias@thi.informatik.uni-frankfurt.de

Mit einem Stern gekennzeichnete Abschnitte werden in der Vorlesung nur kurz angesprochen.

Inhaltsverzeichnis

Kapitel 1

Einführung

Die Vorlesung behandelt fundamentale Algorithmen, allgemeine Methoden für den Entwurf und die Analyse von Algorithmen im ersten Teil, die Erkennung schwieriger algorithmischer Probleme, nämlich die NP-Vollständigkeit im zweiten und die Grenzen der Berechenbarkeit im vierten Teil. Algorithmen für schwierige Probleme sind Inhalt des dritten Teils.

In Teil I entwickeln wir effiziente Algorithmen für Sortier- und Ordungsprobleme sowie für graph-theoretische Probleme wie die Traversierung von Graphen sowie die Berechnung kürzester Wege und minimaler Spannbäume. Desweiteren stellen wir die wichtigsten Entwurfsprinzipien wie

- Divide & Conquer,
- dynamisches Programmieren
- und Greedy-Algorithmen

vor. Wir geben auch eine kurze, allerdings oberflächliche Einführung in die lineare Programmierung, dem mächtigsten, effizienten Optimierungsverfahren.

Im Teil II beschäftigen wir uns mit schwierigen Problemen. Wir finden heraus, dass viele wichtige Entscheidungsprobleme, die nach der Existenz von Lösungen fragen, algorithmisch gleichschwer sind. Dazu führen wir den Begriff eines nichtdeterministischen Programms ein, das erfolgreich Lösungen rät, wenn Lösungen existieren. Wir werden dann auf die Klasse NP und auf die Klasse der NP-vollständigen Probleme geführt, die alle im Wesentlichen den gleichen Schwierigkeitsgrad besitzen. Wir werden herausfinden, dass die Lösung eines einzigen NP-vollständigen Problems bedingt, dass heutige Rechner „effizient raten“ können und müssen folgern, dass wahrscheinlich kein einziges NP-vollständige Problem effizient lösbar ist.

Leider sind gerade die NP-vollständigen (oder NP-harten) Probleme für praktische Anwendungen wichtig, und wir müssen deshalb nach Wegen suchen, approximative Antworten zu erhalten. Genau das tun wir im Teil III, wo die wichtigsten Verfahren zur Lösung schwieriger Entscheidungs- und Optimierungsprobleme, beziehungsweise für die Bestimmung von Gewinnstrategien für nicht-triviale Zwei-Personen Spiele vorgestellt werden.

Wenn die Laufzeit eines Programms keine Rolle spielt, wenn wir uns also in einer Traumwelt aus einem anderen Universum befinden, dann sind alle Probleme in NP völlig harmlos. Können wir denn jedes Problem knacken, wenn Laufzeit (und Speicherplatz) keine Rolle spielen? Diese Frage werden wir im Teil IV „Berechenbarkeit“ verneinen müssen. Wir werden dort den Begriff der Berechenbarkeit einführen und Beispiele für nicht entscheidbare Probleme angeben, also

für Probleme die wir nicht mit Hilfe eines Rechners knacken können. Schließlich weisen wir mit dem Satz von Rice nach, dass fast alle interessanten Fragen über das Verhalten eines Programms unentscheidbar sind. Insbesondere wird ein „Supercompiler“, der nur feststellen soll, ob ein vorgebenes Anwenderprogramm immer hält, selbst manchmal nicht halten!

Das Skript beginnt mit einem Grundlagenkapitel, dass für die Algorithmenanalyse wichtige Gleichungen und relevante Ergebnisse zum Beispiel aus der elementaren Stochastik enthält. Desweiteren wird an die Konzepte der Asymptotik erinnert. Das Skript endet mit einem Ausblick auf weiterführende Veranstaltungen.

Was möchten wir in dieser Veranstaltung erreichen? Fundamentale Algorithmen sollen bekannt sein und der Prozess des Entwurfs und der Analyse von Algorithmen soll eigenständig durchgeführt werden können. Aber nicht nur das, sondern wir müssen auch die Grenzen der (effizienten) Berechenbarkeit kennen und ein Gefühl dafür entwickeln, wann ein vorgegebenes Problem zu schwierig ist und wir unsere Erwartungshaltung anpassen müssen, also zum Beispiel mit approximativen statt optimalen Lösungen zufrieden sein müssen.

1.1 Grundlagen

Die beiden folgenden Gleichungen werden wir häufig benutzen.

- Die Summe der ersten n Zahlen stimmt mit $n(n+1)/2$ überein, es gilt also

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

  Warum stimmt diese Gleichung? Ein 2-dimensionales Array mit n Zeilen und Spalten hat genau n^2 Einträge. Wir können die Summe $1+2+\cdots n$ als die Anzahl der Einträge unterhalb und einschließlich der Diagonale interpretieren. Aber wieviele Einträge sind dies? Wenn wir die Diagonale herausnehmen, dann zerfällt das Array in zwei gleichgroße Hälften: Jede dieser Hälften besteht also aus genau $(n^2-n)/2$ Einträgen. Die Summe $1+2+\cdots n$ stimmt aber mit der Anzahl der Einträge in der unteren Hälfte überein, wenn wir noch zusätzlich die n Einträge der Diagonale hinzuaddieren: Es ist

$$1+2+\cdots n = \frac{n^2-n}{2} + n = \frac{n(n+1)}{2}.$$

- Für jede reelle Zahl $a \neq 1$ gilt

$$\sum_{k=0}^{n} a^k = \frac{a^{n+1}-1}{a-1}.$$

  Auch diese Gleichung lässt sich einfach einsehen, denn

$$(a-1)\cdot\sum_{k=0}^{n} a^k = a\cdot\sum_{k=0}^{n} a^k - \sum_{k=0}^{n} a^k = \sum_{k=0}^{n} a^{k+1} - \sum_{k=0}^{n} a^k = a^{n+1} - 1.$$

Wie ist die Logarithmus-Funktion definiert und wie rechnet man mit Logarithmen? Wenn $a > 1$ und $x > 0$ reelle Zahlen sind, dann ist $\log_a(x)$ der Logarithmus von x zur Basis a und stimmt genau dann mit y überein, wenn $a^y = x$ gilt.
Warum ist der Logarithmus eine zentale Funktion für die Informatik?

Aufgabe 1
Wieviele Bitpositionen besitzt die Binärdarstellung einer natürlichen Zahl n?

Lemma 1.1 (Logarithmen) *$a, b > 1$ und $x, y > 0$ seien reelle Zahlen. Dann gilt*

(a) $\log_a(x \cdot y) = \log_a x + \log_a y$.

(b) $\log_a(x^y) = y \cdot \log_a(x)$.

(c) $a^{\log_a x} = x$.

(d) $\log_a x = (\log_a b) \cdot (\log_b x)$.

(e) $b^{\log_a x} = x^{\log_a b}$.

1.1.1 Einige Grundlagen aus der Stochastik

Wir werden in dieser Vorlesung meistens die pessimistische Sicht des Worst-Case-Szenarios analysieren. Das heißt, wir werden uns bei der Analyse von Lösungsstrategien fragen, was schlimmstenfalls geschehen könnte. Manchmal verzerrt dieser Blick aber die wahren Gegebenheiten zu stark, und eine Betrachtung dessen, was man vernünftigerweise *erwarten* sollte, ist geboten. Insbesondere verliert die Worst-Case-Analyse drastisch an Wert, wenn der Worst-Case extrem *unwahrscheinlich* ist.

Eine kurze Wiederholung elementarer Begriffe der Stochastik ist daher angebracht. Wir konzentrieren uns auf endliche **Wahrscheinlichkeitsräume**. Also ist eine endliche Menge Ω von **Elementarereignissen** gegeben sowie eine **Wahrscheinlichkeitsverteilung** p, die jedem Elementarereignis $e \in \Omega$ die Wahrscheinlichkeit $p(e)$ zuweist. Desweiteren muss $\sum_{e\in\Omega} p(e) = 1$ und $p(e) \geq 0$ für alle $e \in \Omega$ gelten. Ein **Ereignis** E ist eine Teilmenge von Ω und $\text{prob}[E] = \sum_{e\in E} p(e)$ ist die Wahrscheinlichkeit von E.

Beispiel 1.1 Wir wählen die Menge der 37 Fächer eines Roulettspiels als unsere Menge von Elementarereignissen. Die Ereignisse *gerade, ungerade, rot* oder *schwarz* können dann als Vereinigung von Elementarereignissen beschrieben werden. Die Wahrscheinlichkeit eines solchen Ereignisses ergibt sich aus der Summe der jeweiligen Elementarereignisse.

Lemma 1.2 (Rechnen mit Wahrscheinlichkeiten) *Seien A und B Ereignisse über dem endlichen Wahrscheinlichkeitsraum $\Omega = \{e_1, \ldots, e_n\}$ und sei $p = (p_1, \ldots, p_n)$ eine Wahrscheinlichkeitsverteilung.*

(a) $prob[A \cap B] = \sum_{e\in A\cap B} p(e)$. *Insbesondere ist* $0 \leq prob[A \cap B] \leq \min\{prob[A], prob[B]\}$.

(b) $prob[A \cup B] = \sum_{e\in A\cup B} p(e)$. *Insbesondere ist*

$$\begin{aligned} max\{prob[A], prob[B]\} &\leq prob[A \cup B] \\ &= prob[A] + prob[B] - prob[A \cap B] \leq prob[A] + prob[B]. \end{aligned}$$

(c) $prob[\neg A] = \sum_{e\notin A} p(e) = 1 - prob[A]$.

Erwartungswerte spielen bei der Analyse von erwarteten Laufzeiten eine zentrale Rolle. Die allgemeine Definition ist wie folgt.

Definition 1.3

(a) *Die Menge $A \subseteq \mathbb{R}$ sei gegeben. Eine Zufallsvariable $X : \Omega \to A$ wird durch eine Wahrscheinlichkeitsverteilung $(q(a) \mid a \in A)$ spezifiziert. Wir sagen, dass $q(a)$ die Wahrscheinlichkeit des Ereignisses $X = a$ ist.*

(b) *Sei X eine Zufallsvariable und sei $q(a)$ die Wahrscheinlichkeit für das Ereignis $X = a$. Dann wird der Erwartungswert von X durch*

$$E[X] = \sum_{a \in A} a \cdot q(a)$$

definiert.

Die möglichen Ausgänge a der Zufallsvariable werden also mit ihrer Wahrscheinlichkeit gewichtet und addiert. Eine zentrale Eigenschaft des Erwartungswerts ist seine Additivität.

Lemma 1.4 *Für alle Zufallsvariablen X und Y ist $E[X + Y] = E[X] + E[Y]$.*

Der Informationsgehalt eines Erwartungswertes hängt dabei vom konkreten Experiment ab. Dass man bei einem handelsüblichen Würfel eine 3.5 erwartet ist eine wenig nützliche Information. Dass der erwartete Gewinn bei einem Einsatz von 10 Euro auf rot am Roulettetisch $\frac{18}{37} \cdot 20$ Euro $+ \frac{19}{37} \cdot 0$ Euro $= 9.73$ Euro ist, hat dagegen die nützliche Botschaft: *Finger weg.*

Aufgabe 2
Wir betrachten die Menge $\Omega = \{1, 2, \ldots, n\}$. In einem ersten Experiment bestimmen wir eine Menge $A \subseteq \Omega$, indem wir jedes Element aus Ω mit Wahrscheinlichkeit p_A in A aufnehmen. Wir wiederholen das Experiment und bilden eine Menge B, wobei wir jedes Element aus Ω mit Wahrscheinlichkeit p_B in B aufnehmen.

(a) **Bestimme** $E(|A \cap B|)$, den Erwartungswert der Mächtigkeit der Schnittmenge.

(b) **Bestimme** $E(|A \cup B|)$, den Erwartungswert der Mächtigkeit der Vereinigungsmenge.

Aufgabe 3
Wir spielen ein Spiel gegen einen Gegner. Der Gegner denkt sich zwei Zahlen aus und schreibt sie für uns nicht sichtbar auf je einen Zettel. Wir wählen *zufällig* einen Zettel und lesen die darauf stehende Zahl. Sodann haben wir die Möglichkeit, diese Zahl zu behalten oder sie gegen die uns unbekannt gebliebene Zahl zu tauschen. Sei x die Zahl, die wir am Ende haben, und y die andere. Dann ist unser (möglicherweise negativer) Gewinn $x - y$.

- Wir betrachten Strategien S_t der Form *„Gib Zahlen $< t$ zurück und behalte diejenigen $\geq t$“*. Analysiere den Erwartungswert $E_{x,y}(Gewinn(S_t))$ des Gewinns dieser Strategie in Abhängigkeit von t, x und y.
- Gib eine randomisierte Strategie an, die für beliebige $x \neq y$ einen positiven erwarteten Gewinn für uns aufweist.

Häufig untersuchen wir wiederholte Zufallsexperimente. Wenn die Wiederholungen unabhängig voneinander sind, hilft die Binomialverteilung weiter.

Lemma 1.5 (Binomialverteilung) *Sei A ein Ereignis, welches mit Wahrscheinlichkeit p auftritt. Wir führen n Wiederholungen des betreffenden Experimentes durch und zählen, wie häufig ein Erfolg eintritt, das heißt wie häufig A eingetreten ist. Die Zufallsvariable X möge genau dann den Wert k annehmen, wenn k Erfolge vorliegen.*

(a) Die Wahrscheinlichkeit für k Erfolge ist gegeben durch

$$prob[X = k] = \binom{n}{k} \cdot p^k \cdot (1 - p)^{n-k}.$$

(b) Die Wahrscheinlichkeit, dass die Anzahl der Erfolge im Intervall $[a, b]$ liegt ist also

$$prob[X \in [a, b]\,] = \sum_{k=a}^{b} \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k}.$$

(c) $E[X] = n \cdot p$ ist die erwartete Anzahl der Erfolge.

Damit haben wir den Erwartungswert einer binomialverteilten Zufallsvariable berechnet. Zufallsvariablen mit prinzipiell unendlich vielen Ausgängen werden uns zum Beispiel immer dann begegnen, wenn wir uns fragen, wie lange man auf das Eintreten eines bestimmten Ereignisses warten muss. Warten wir zum Beispiel am Roulettetisch auf die erste 0 des Abends, so gibt es keine Anzahl von Runden, so dass die erste 0 innerhalb dieser Rundenzahl gefallen sein *muss*. Natürlich wird aber anhaltendes Ausbleiben immer unwahrscheinlicher und deswegen ergibt sich trotzdem ein endlicher Erwartungswert. Wir haben damit die *geometrische Verteilung* beschrieben.

Lemma 1.6 (geometrische Verteilung) *Sei A ein Ereignis mit Wahrscheinlichkeit p. Die Zufallsvariable X beschreibe die Anzahl der Wiederholungen des Experimentes bis zum ersten Eintreten von A.*

(a) Die Wahrscheinlichkeit, dass X den Wert k annimmt ist

$$prob[X = k] = (1-p)^{k-1} \cdot p.$$

(b) Der Erwartungswert ist $E(X) = \frac{1}{p}$.

Aufgabe 4
Wir nehmen in einem Casino an einem Spiel mit Gewinnwahrscheinlichkeit $p = 1/2$ teil. Wir können einen beliebigen Betrag einsetzen. Geht das Spiel zu unseren Gunsten aus, erhalten wir den Einsatz zurück und zusätzlich denselben Betrag aus der Bank. Endet das Spiel ungünstig, verfällt unser Einsatz. Wir betrachten die folgende Strategie:

```
i:=0
REPEAT
    setze 2^i$
    i:=i+1
UNTIL(ich gewinne zum ersten mal)
```

Bestimme den erwarteten Gewinn dieser Strategie und die erwartete notwendige Liquidität (also den Geldbetrag, den man zur Verfügung haben muss, um diese Strategie ausführen zu können).

1.1.2 Asymptotik

Wir schauen uns die in der Vorlesung „Datenstrukturen“ eingeführte asymptotische Notation nochmal im Detail an. Wir haben Groß-Oh und Klein-Oh betrachtet, ebenso wie Ω und Θ.

Definition 1.7 *Seien $f, g : \mathbb{N} \to \mathbb{R}_{\geq 0}$ Funktionen, die einer Eingabelänge $n \in \mathbb{N}$ eine nicht-negative Laufzeit zuweisen.*

(a) Die Groß-Oh Notation: $f = O(g) \Leftrightarrow$ Es gibt eine positive Konstante $c > 0$ und eine natürliche Zahl $n_0 \in \mathbb{N}$, so dass, für alle $n \geq n_0$

$$f(n) \leq c \cdot g(n)$$

gilt.

(b) $f = \Omega(g) \Leftrightarrow g = O(f)$.

(c) $f = \Theta(g) \Leftrightarrow f = O(g)$ *und* $g = O(f)$.

(d) Die Klein-Oh Notation: $f = o(g) \Leftrightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

(e) $f = \omega(g) \Leftrightarrow \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$.

Was besagen die einzelnen Notationen? $f = O(g)$ drückt aus, dass f asymptotisch nicht stärker als g wächst. Ein Algorithmus mit Laufzeit f ist also, unabhängig von der Eingabelänge, um höchstens einen konstanten Faktor langsamer als ein Algorithmus mit Laufzeit g. (Beachte, dass $f = O(g)$ sogar dann gelten kann, wenn $f(n)$ stets größer als $g(n)$ ist.) Gilt hingegen $f = o(g)$, dann ist ein Algorithmus mit Laufzeit f für hinreichend große Eingabelänge wesentlich schneller als ein Algorithmus mit Laufzeit g, denn der Schnelligkeitsunterschied kann durch keine Konstante beschränkt werden. Während $f = \Omega(g)$ (und damit äquivalent $g = O(f)$) besagt, dass f zumindest so stark wie g wächst, impliziert $f = \Theta(g)$, dass f und g gleichstark wachsen.

Die asymptotischen Notationen erwecken zunächst den Anschein, als würden einfach Informationen *weggeworfen*. Dieser Eindruck ist auch sicher nicht falsch. Man mache sich jedoch klar, dass wir, wenn wir eine Aussage wie etwa

$$4n^3 + \frac{2n}{3} - \frac{1}{n^2} = O(n^3)$$

machen, nur untergeordnete Summanden und konstante Faktoren weglassen. Wir werfen also gezielt nachrangige Terme weg, also die Terme, die für hinreichend hohe Werte von n nicht ins Gewicht fallen, und wir verzichten auf die führende Konstante (hier 4), da die konkreten Konstanten bei realen Implementierungen ohnehin compiler- und rechnerabhängig sind. Man kann also sagen: *Wir reduzieren einen Ausdruck auf das asymptotisch Wesentliche.*

Die asymptotischen Notationen erlauben es uns, Funktionen in verschiedene *Wachstumsklassen* einzuteilen. Wie gewünscht, können wir jetzt unterscheiden, ob die Laufzeit eines Algorithmus logrithmisch, linear, quadratisch, kubisch oder gar exponentiell ist. Was nützt uns diese Information? Wenn wir einen Algorithmus mit kubischer Laufzeit starten, haben wir natürlich keine Vorstellung von der Laufzeit, die sich in Minuten oder Sekunden ausdrücken ließe. Wir können jedoch zum Beispiel verlässlich vorhersagen, dass der Algorithmus, wenn man ihn auf eine doppelt so große Eingabe ansetzt, seine Laufzeit in etwa verachtfachen wird. Diese Prognose wird um so zutreffender sein, je größer die Eingabe ist, da sich dann die Vernachlässigung der niederen Terme zunehmend rechtfertigt.

Aufgabe 5
Angenommen, ein Algorithmus hat Laufzeit n^k. Um welchen Faktor wächst die Laufzeit, wenn wir die Eingabelänge verdoppeln?

Natürlich entscheiden konstante Faktoren in der Praxis, denn wer möchte schon mit mit einem Algorithmus arbeiten, der um den Faktor 100 langsamer ist als ein zweiter Algorithmus. Neben der asymptotischen Analyse muss deshalb auch eine experimentelle Analyse treten, um das tatsächliche Laufzeitverhalten auf „realen Daten“ zu untersuchen. Dieser Ansatz, also die Kombination der asymptotischen und experimentellen Analyse und ihre Wechselwirkung mit dem Entwurfsprozess, sollte das Standard-Verfahren für den Entwurf und die Evaluierung von Algorithmen und Datenstrukturen sein.

1.2 Literatur

Die folgenden Textbücher ergänzen und vertiefen das Skript:

- J. Kleinberg und E. Tardos, Algorithm Design, Addison-Wesley 2005.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest und C. Stein, Introduction to Algorithms, Third Edition, MIT Press 2009.
- M. Nebel, Entwurf und Analyse von Algorithmen, Springer 2012.

Teil I

Effiziente Algorithmen

Kapitel 2

Sortieren

Im Sortierproblem sind n Zahlen $A[1], \ldots A[n] \in \mathbb{Z}$ (oder aus einer beliebigen, aber mit einer vollständigen Ordnung versehenen Menge) vorgegeben. Die Zahlen sind in aufsteigender Reihenfolge auszugeben.

Das Sortierproblem wird einen breiten Raum einnehmen, wir werden uns in diesem Kapitel aber auch mit dem Auswahlproblem beschäftigen. Im Auswahlproblem sind n Zahlen $A[1], \ldots, A[n]$ gegeben sowie eine natürliche Zahl k. Die k-kleinste Zahl ist zu bestimmen. Warum betrachten wir das Auswahlproblem, wenn wir doch zuerst das Array sortieren und dann die Antwort für das Auswahlproblem sofort ablesen könnten? Weil uns eine schnellere Lösung für das Auswahlproblem gelingen wird!

Wir wählen stets n, die Anzahl der zu sortierenden Zahlen, als Eingabelänge.

2.1 Bubble Sort, Selection Sort und Insertion Sort

Bubble Sort beruht auf Vergleich/Vertausch Operationen benachbarter Zahlen. Bubble Sort ist in (höchstens) $n-1$ Phasen aufgeteilt, wobei in jeder Phase benachbarte Zahlen verglichen und, wenn notwendig, durch die Funktion `swap` vertauscht werden.

```
active = 1; for (i=1; active; i++)
  {
    //Phase i beginnt:
    active = 0;
    for (j=1; j <= n-i; j++)
      if (A[j] > A[j+1]) {active = 1; swap(A, j, j+1);}
  }
```

In Phase 1 werden also sukzessiv, von links nach rechts, alle Nachbarn verglichen und, wenn nötig, vertauscht. In diesem Prozeß erreicht die größte Zahl Position n, wo sie auch hingehört. In Phase 2 brauchen wir also Position n nicht mehr zu berücksichtigen. Mit dem gleichen Argument stellen wir fest, dass die zweitgrößte Zahl am Ende der zweiten Phase auf Position $n-1$ landet, und allgemein, dass die $i$tgrößte Zahl am Ende von Phase i ihre Position $n-i+1$ erreicht. Also muss Phase $i+1$ somit nur noch die Zahlen $A[1], \ldots, A[n-i]$ sortieren und genau das geschieht in der obigen Schleife. $n-1$ Phasen sind also ausreichend.

Bestimmen wir die Laufzeit von Bubble Sort. In der i-ten Phase werden $n-i$ Vergleiche (und möglicherweise Vertauschungen) durchgeführt. Insgesamt führt Bubble Sort also höchstens

$$\sum_{i=1}^{n-1}(n-i) = \sum_{j=1}^{n-1} j = \frac{n\cdot(n-1)}{2}$$

Vergleiche und Vertauschungen durch. Diese beiden Operationen dominieren aber die Laufzeit von Bubble Sort und deshalb

Satz 2.1 *Bubble Sort besitzt die worst-case Laufzeit* $\Theta(n^2)$.

Wir haben oben nur die obere Schranke $O(n^2)$ nachgewiesen, warum die Θ-Schranke? Bubble Sort benötigt mindestens $\frac{n\cdot(n-1)}{2}$ Schritte für die Eingabe $(2,3,4,\ldots,n,1)$! In einer Phase geschieht denkbar wenig, denn nur die Eins wird um eine Zelle nach links bewegt! Wir benötigen also alle $n-1$ Phasen.

Die worst-case Rechenzeit von Bubble Sort ist also im vergleich mit zum Beispiel Heapsort miserabel. Wie sieht es aber mit der durchschnittlichen Rechenzeit aus? Wir nehmen an, dass jede Permutation von $(1,2,\ldots,n)$ mit Wahrscheinlichkeit $\frac{1}{n!}$ als Eingabe für Bubble Sort auftritt: Alle Permutationen sind als Eingaben für Bubble Sort gleichwahrscheinlich.

In diesem Fall wird die Zahl 1 mit Wahrscheinlichkeit mindestens $1/2$ auf einer der Positionen $\{\frac{n}{2}+1,\ldots,n\}$ erscheinen und Bubble Sort benötigt dann mindestens $n/2$ Phasen. Also benötigt Bubble Sort mit Wahrscheinlichkeit mindestens $1/2$ mindestens $n/2$ Phasen. In $n/2$ Phasen werden aber mindestens

$$\sum_{i=1}^{n/2}(n/2-i) = \sum_{j=1}^{n/2-1} j = \frac{n}{4}\left(\frac{n}{2}-1\right)$$

Vergleichs- und Vertauschungsoperationen durchgeführt. Damit erhalten wir also für die durchschnittliche (oder erwartete) Laufzeit $D(n)$:

$$D(n) \geq \frac{1}{2}\left(\frac{n^2}{8}-\frac{n}{4}\right).$$

Satz 2.2 *Alle Permutationen von* $(1,2,\ldots,n)$ *seien als Eingabe gleichwahrscheinlich. Dann ist* $\Theta(n^2)$ *die erwartete Laufzeit von Bubble Sort.*

Mit anderen Worten, Bubble Sort zeigt seine quadratische Laufzeit schon für die meisten Eingaben. Zur Ehrenrettung von Bubble Sort:

- Bubble Sort ist schnell, wenn sich die Elemente bereits *in der Nähe ihrer endgültigen Position* befinden,
- ist einfach zu programmieren und
- kann „ohne Sorgen“ für wenige Zahlen angewandt werden.

Aufgabe 6
Wir nehmen an, dass jeder Schlüssel in einem Array $(a_1,\ldots,a_n)$ höchstens die Distanz d von seiner endgültigen Position im sortierten Array $(a_{p(1)},\ldots,a_{p(n)})$ besitzt, d.h. $|i-p(i)| \leq d$ für alle $i=1,\ldots,n$. **Wieviele** Phasen benötigt dann Bubblesort?

Auch **Selection Sort** ist in Phasen aufgeteilt. Während Bubble Sort in einer Phase die größte Zahl erfolgreich bestimmt, berechnet Selection Sort stets die kleinste Zahl:

```
for (i=1; i < n ; i++)
  finde die kleinste der Zahlen A[i],...,A[n] durch lineare Suche
  und vertausche sie mit A[i].
```

In der i-ten Phase benötigt Selection Sort $n - i$ Vergleiche, aber nur eine Vertauschung. Insgesamt werden also

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

Vergleiche und $n-1$ Vertauschungen benötigt. (Beachte, dass die Laufzeit von Selection Sort für alle Eingaben gleicher Länge identisch ist.) Damit ist Selection Sort ein gutes Sortierverfahren für lange Datensätze, da nur $n-1$ Vertauschungen benötigt werden. Aber seine (worst-case, best-case oder erwartete) Laufzeit ist wie für Bubble Sort quadratisch.

Insertion Sort sortiert sukzessive die Teilarrays $(A[1], A[2])$, $(A[1], A[2], A[3])$ usw.:

```
for (i=1; i < n; i++)
  //Phase i beginnt. Zu diesem Zeitpunkt bilden die Zahlen
  //A[1],...,A [i] bereits eine sortierte Folge.
  Fuege A[i+1] in die Folge (A[1],...,A[i]) ein
    und zwar verschiebe die groesseren Zahlen um eine Position
    nach rechts und fuege A[i+1] in die freie Position ein.
```

Dieses Verfahren entspricht der Methode, nach der man Spielkarten in der Hand sortiert. In Phase i wird $A[i+1]$ mit bis zu i Zahlen verglichen. Die Eingabe $(n, n-1, n-2, \ldots, 3, 2, 1)$ ist ein worst-case Beispiel für Insertion Sort, denn es werden

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

Vergleiche durchgeführt. Es kann gezeigt werden, dass auch die erwartete Anzahl der Vergleiche quadratisch ist. Beachte aber, dass $n-1$ die best-case Anzahl der Vergleiche ist: Diese Anzahl wird für sortierte Arrays erreicht. Allgemein ist Insertion Sort ein gutes Sortierverfahren für „fast-sortierte“ Daten.

Von den drei betrachteten Verfahren, wenn eine Empfehlung ausgesprochen werden muss, ist Insertion Sort vorzuziehen: Seine durchschnittliche Laufzeit ist niedriger als die seiner Kontrahenten und das Verfahren ist auch gut geeignet für „fast-sortierte“ Eingabefolgen. Tatsächlich wird Insertion Sort oft für das Sortieren kleiner Datenmengen benutzt. Selection Sort liefert gute Ergebnisse für kurze Arrays mit langen Datensätzen.

Aufgabe 7
Zeige, dass die *erwartete* Laufzeit von Insertion Sort $\Theta(n^2)$ beträgt. Dabei nehmen wir an, dass die Eingabe aus einer Permutation der Zahlen $1, 2, \ldots, n-1, n$ besteht. Jede Permutation tritt mit Wahrscheinlichkeit $1/(n!)$ als Eingabe auf.

Aufgabe 8
Bestimme für die Sortieralgorithmen *Insertion Sort, Bubble Sort* und *Selection Sort* asymptotisch exakt die Anzahl der Vergleiche und die Anzahl der Vertauschungen auf den folgenden Zahlenfolgen (Begründung, wie immer, erforderlich).

(a) $N, 1, 2, 3, \ldots, N-1$

(b) $1, \frac{N}{2}+1, 2, \frac{N}{2}+2, \ldots, \frac{N}{2}, N (N \text{ gerade})$

Aufgabe 9
In dieser Aufgabe betrachten wir *konservative* Sortieralgorithmen: Ein Sortieralgorithmus heißt konservativ, wenn in jedem Schritt nur die Inhalte **benachbarter** Zellen des Eingabe-Arrays vertauscht werden.

Beachte, dass Bubble-Sort ein konservativer Sortieralgorithmus ist; auch Selection Sort und Insertion Sort lassen sich als konservative Sortieralgorithmen auffassen. Kann es schnelle konservative Sortieralgorithmen geben? Die Antwort ist nein; selbst Verallgemeinerungen von Bubble Sort, Selection Sort oder Insertion Sort werden somit eine schlechte Laufzeit besitzen.

Genauer: **Zeige**, dass die worst-case Laufzeit aller konservativen Sortieralgorithmen mindestens $\Omega(n^2)$ beträgt, wenn n Zahlen zu sortieren sind.

2.2 Quicksort

Quicksort ist einer der schnellsten und beliebtesten Sortieralgorithmen. Wir beschreiben zuerst die Grundidee.

- Das Array $(A[1], \ldots, A[n])$ sei zu sortieren.
- Eine Funktion `pivot` berechnet einen Index p mit $1 \leq p \leq n$. Die Zahl $\alpha = A[p]$ wird als Pivotelement bezeichnet.
- Eine Funktion `partition` permutiert das Array A und berechnet auch die neue Position i von α. Die Permutation hat die folgenden Eigenschaften:
  (a) α befindet sich bereits an seiner endgültigen Position,
  (b) alle Elemente links von α sind kleiner oder gleich α und
  (c) alle Elemente rechts von α sind größer oder gleich α.
- Das Array A besteht jetzt aus 3 Teilarrays: Das Teilarray aller Zahlen links von α, das Teilarray, das nur aus α besteht und das Teilarray aller Zahlen rechts von α.
  Quicksort wird jetzt rekursiv auf das erste und dritte Teilarray angewandt.

Eine C++ Implementierung:

```
void quicksort (int links, int rechts)
  {
    int p, i;
    if (links < rechts)
     {
       p = pivot (links, rechts);
       i = partition (p, links, rechts);
       quicksort (links, i-1);
       quicksort (i+1, rechts);
     }
  }
```

Zuerst nehmen wir nur die einfachste Pivotwahl an: `pivot` (links, rechts) wählt stets links, also den Anfangsindex des gegenwärtig bearbeiteten Teilarrays von A. Später werden wir Pivot-Strategien kennenlernen, die zu einer beträchtlichen Laufzeitbeschleunigung führen können.

Wir führen als Nächstes einen Korrektheitsbeweis für Quicksort durch, wobei wir annehmen, dass `pivot` und `partition` korrekt funktionieren. Offensichtlich bietet sich eine Induktion über die Eingabelänge $L = \text{rechts} - \text{links} + 1$ an.

Basis: Wenn $L = 1$, dann terminiert Quicksort ohne Ausführung einer einzigen Operation. Das ist richtig, da ein-elementige Arrays bereits sortiert ist.

Induktionsschritt: Wir nehmen an, dass Quicksort für Eingabelänge $L \leq n$ korrekt sortiert und haben nachzuweisen, dass Quicksort (links, rechts) auch korrekt sortiert, falls rechts $-$ links $+ 1 = n + 1$. Was passiert in Quicksort (links, rechts)?
Die beiden rekursiven Aufrufe beziehen sich auf höchstens n Zahlen und wir können die Induktionsannahme anwenden!

Wir müssen also nur noch die Funktion

```
partition (int p, int links, int rechts)
```

implementieren. partition arbeitet mit zwei „Zeigern“ l und r.

(1) Wir bezeichnen das Pivotelement $A[p]$ wiederum mit α. Setze $l =$ links $- 1$, $r =$ rechts und vertausche A [rechts] und $A[p]$.

(2) Wiederhole solange, bis $l \geq r$.

   (a) Berechne $l{+}{+}$ und verschiebe den l-Zeiger so lange nach rechts bis $A[l] \geq \alpha$.

   (b) Berechne $r - -$ und verschiebe den r-Zeiger so lange nach links bis $A[r] \leq \alpha$.

   (c) Wenn $(l < r)$, dann vertausche A [l] und $A[r]$.

(3) Vertausche A [rechts] und A [l].

Was passiert? Zuerst wird $A[p]$ auf Position rechts gesetzt. l zeigt auf Position links-1, r auf Position rechts. Damit gilt zu Anfang die Invariante:

> Alle Zellen links von l (einschließlich l) besitzen nur Zahlen $\leq \alpha$, alle Zellen rechts von r (einschließlich r) besitzen nur Zahlen $\geq \alpha$.

Wenn $l < r$, wird l um 1 erhöht und die erste Zahl rechts von Position l gesucht, die größer oder gleich α ist. Dann wird r um 1 erniedrigt und die erste Zahl links von r gesucht, die kleiner oder gleich α ist. (Warum wird zuerst erhöht bzw. erniedrigt? Die Invariante sichert, dass am Anfang der Phase $A[l] \leq \alpha$ und $A[r] \geq \alpha$. Wir sparen also zwei Vergleiche durch das sofortige Erhöhen bzw. Erniedrigen.) Wir nehmen jetzt eine Fallunterscheidung vor.

Fall 1: Nachdem beide Zeiger zur Ruhe gekommen sind, gilt $l < r$.

Damit ist nach dem Vertauschen von $A[l]$ und $A[r]$ die Invariante wiederhergestellt!

Fall 2: Nachdem beide Zeiger zur Ruhe gekommen sind, gilt $l \geq r$. In diesem Fall sind alle Zahlen links von l kleiner oder gleich α. Alle Zahlen rechts von l, sind auch rechts von r und damit größer oder gleich α. Also bezeichnet l die endgültige Position von α nach der Partition!

Damit ist also alles in Butter! Ganz und gar nicht, denn wir nehmen an, dass beide Zeiger auch zur Ruhe kommen. Der l-Zeiger wird stets zur Ruhe kommen: notfalls wird er auf Position rechts von α gestoppt. Aber der r-Zeiger wird zum Beispiel dann nicht stoppen, wenn α die kleinste Zahl des Teilarrays (A [links], …, A [rechts]) ist! Dieses Problem läßt sich wie folgt ohne weitere Abfragen lösen:

> Setze $A[0] = y$, wobei y kleiner als alle Zahlen in $(A[1], \ldots, A[n])$ ist.

Damit haben wir zumindest beim ersten Aufruf von partition, also beim Aufruf von partition $(p, 1, n)$ keine Probleme: r wird notfalls durch $A[0]$ gestoppt. Was passiert bei den rekursiven Aufrufen quicksort $(1, i-1)$ und quicksort $(i+1, n)$: $A[0]$ dient als Stopper für den ersten Aufruf und das Pivotelement in Position i als Stopper für den zweiten Aufruf!

Warum haben wir uns soviel Mühe gegeben, die Partitionsfunktion effizient zu kodieren? Der Schnelligkeit und Speichereffizienz gerade dieser Funktion verdankt Quicksort seinen Champion-Status. Eine asymptotisch optimale Implementierung ist trivial; wir haben mehr erreicht, wir haben nämlich das letzte Quentchen Geschwindigkeit aus partition herausgeholt.

2.2.1 Eine Laufzeit-Analyse von Quicksort

Zuerst zur worst-case Analyse. Wir betrachten die Eingabe $(1, 2, \ldots n)$, also eine bereits sortierte Eingabe! Quicksort geht dennoch mit großem Eifer zur Sache und bewirkt die folgenden Anrufe

$$\text{quicksort } (1, n),\ \text{quicksort } (2, n),\ \text{quicksort } (3, n), \ldots,\ \text{quicksort } (n, n)$$

Warum? quicksort (i, n) wird die Zahl i als Pivot wählen und die Aufrufe quicksort $(i, i-1)$ sowie quicksort $(i+1, n)$ durchführen. Da der Anruf quicksort $(i, i-1)$ sofort terminiert, haben wir ihn nicht aufgeführt.

Damit erhalten wir zwei negative Konsequenzen. Zuerst ist die Rekursionstiefe für diese Eingabe $n-1$: Bevor irgendeiner der Anrufe quicksort $(1, n), \ldots,$ quicksort $(n-1, n)$ terminieren kann, muss quicksort (n, n) ausgeführt werden.

Zuletzt wird für nichts und wieder nichts quadratische Zeit verbraucht: quicksort (i, n) wird, über seine Partitionsfunktion, $n - i$ Vergleiche ausführen. Also werden insgesamt mindestens

$$\begin{aligned} \sum_{i=1}^{n} n - i &= 1 + 2 + 3 + \ldots + n - 1 \\ &= \frac{n \cdot (n-1)}{2} \end{aligned}$$

Vergleiche benötigt. Allerdings ist quadratische Zeit auch ausreichend: Offensichtlich läuft partition (p, links, rechts) in linearer Zeit (also in Zeit $O(\text{rechts} - \text{links} + 1)$). Weiterhin werden wir die Funktion `pivot` stets so wählen, dass sie in konstanter Zeit läuft. Insgesamt benötigt quicksort(links,rechts) also Zeit höchstens $c \cdot (\text{rechts} - \text{links} + 1)$ für die *nicht-rekursiven* Anweisungen, wobei c eine Konstante mit $c \geq 1$ ist.

Wenn also $Q(n)$ die worst-case Laufzeit von Quicksort für n Zahlen bezeichnet, erhalten wir

$$Q(n) \leq \ \max \{Q(i-1) + Q(n-i) + c \cdot n \ \mid \ 1 \leq i \leq n\}.$$

Wir machen den Ansatz

$$Q(n) \leq cn^2$$

Der Ansatz ist offensichtlich richtig für $n = 1$, da $c \geq 1$ gilt. Wenn $Q(n) \leq Q(i-1) + Q(n-i) + cn$, dann folgt induktiv

$$Q(n) \leq c(i-1)^2 + c(n-i)^2 + c \cdot n.$$

Die Funktion x^2 ist konvex und deshalb wird die Summe $(i-1)^2+(n-i)^2$ maximal am Rand, also für $i = 1$, beziehungsweise für $i = n$. Also ist

$$\begin{aligned} Q(n) &\leq c(n-1)^2 + c\ n \\ &= c\ n^2 - c(n-1) \\ &\leq c\ n^2, \end{aligned}$$

und unser Ansatz ist induktiv bestätigt. Nebenbei haben wir auch erhalten, dass die sortierte Folge tatsächlich eine worst-case Eingabe ist!

Wie sieht es im best-case aus? Der beste Fall tritt ein, wenn Quicksort gleichgroße Teilprobleme liefert, denn dann wird die Summe $(i-1)^2+(n-i)^2$ minimiert. Dies führt uns auf die Rekursion

$$Q_{\text{best}}(n) = 2 \cdot Q_{\text{best}}\left(\frac{n}{2}\right) + c\ n.$$

Das Mastertheorem liefert also

$$Q_{\text{best}}(n) = \Theta(n \log_2\ n).$$

Satz 2.3 *Die worst-case Laufzeit von Quicksort ist* $\Theta\left(n^2\right)$. *Die best-case Laufzeit ist* $\Theta(n \log_2 n)$.

Trotz der schlechten worst-case Rechenzeit ist Quicksort in der Praxis ein sehr schnelles Sortierverfahren. Was ist passiert? Die worst-case Rechenzeit ist für Quicksort „trügerisch“, die erwartete Laufzeit hingegen erzählt die wahre Geschichte.

Wir analysieren nun die erwartete Laufzeit $Q_E(n)$ für Quicksort bei *zufälliger Wahl der Pivotelemente* und *einer beliebigen, aber fixierten Eingabepermutation der Zahlen von 1 bis n.*

Zu diesem Zweck führen wir $\binom{n}{2}$ Zufallsvariablen $V_{i,j}$ mit $1 \leq i < j \leq n$ ein. Es sei

$$V_{i,j} = \begin{cases} 1 & \text{falls es zum Vergleich der Werte } i \text{ und } j \text{ kommt,} \\ 0 & \text{sonst.} \end{cases}$$

Beachte, dass Quicksort, im Laufe einer Berechnung, niemals dieselben Elemente mehrfach miteinander vergleicht. Die Anzahl der Vergleiche, die im Laufe des gesamten Verfahrens zur Ausführung kommen, können wir deshalb als

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} V_{i,j}$$

darstellen. Da Vergleiche zwischen den zu sortierenden Elementen die Laufzeit von Quicksort dominieren, können wir nun

$$\begin{aligned} Q_E(n) &= \Theta\left(E\left[\sum_{i=1}^{n} \sum_{j=i+1}^{n} V_{i,j}\right]\right) \\ &= \Theta\left(\sum_{i=1}^{n} \sum_{j=i+1}^{n} E[V_{i,j}]\right). \end{aligned}$$

folgern, wobei wir im zweiten Schritt die Additivität des Erwartungswerts ausgenutzt haben. Sei nun p_{ij} die Wahrscheinlichkeit, mit der es zum Vergleich zwischen i und j (für $i < j$) kommt. Dann ist

$$\begin{aligned} E[V_{i,j}] &= 1 \cdot p_{ij} + 0 \cdot (1 - p_{ij}) \\ &= p_{ij}. \end{aligned}$$

Wir haben also die p_{ij} zu bestimmen. Dazu stellen wir die folgenden Beobachtungen über den Ablauf von Quicksort zusammen.

- i und j werden genau dann verglichen, wenn sich beide im selben rekursiven Aufruf befinden und einer das Pivotelement ist.
- i und j werden genau dann in verschiedene rekursive Aufrufe weitergeleitet, wenn das Pivot-Element kleiner als i oder größer als j ist.

Die Frage, welches Element aus dem Intervall $[i, j]$ als Erstes zum Pivotelement wird, ist also entscheidend. Da jedes Element gleichwahrscheinlich ist, genügt es festzuhalten, dass das Intervall $j - i + 1$ Elemente enthält und es nur bei der Wahl von i oder j zum Vergleich zwischen i und j kommt. Also ist

$$p_{ij} = \frac{2}{j - i + 1}.$$

Für benachbarte Elemente $j = i + 1$ ist $p_{ij} = 1$. Richtig! Benachbarte Elemente müssen verglichen werden, da sie sich allen dritten Elementen gegenüber gleich verhalten und nur durch den direkten Vergleich getrennt werden können.

Wir können unseren Ansatz von oben weiter verfolgen:

$$\begin{aligned} \sum_{i=1}^{n} \sum_{j=i+1}^{n} E[V_{i,j}] &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} p_{ij} \\ &= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j - i + 1} \\ &= 2 \cdot \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{j - i + 1} \\ &= 2 \cdot \sum_{i=1}^{n} \sum_{j=2}^{n-i+1} \frac{1}{j}. \end{aligned}$$

Die innere Summe hat die allgemeine Form $\sum_{i=1}^{K} \frac{1}{i}$. Wir betrachten zuerst die Teilsumme $\sum_{i=K/2+1}^{K} \frac{1}{i}$ und stellen fest, dass $K/2$ Zahlen der Größe höchstens $\frac{2}{K}$ addiert werden müssen und deshalb ist $\sum_{i=K/2+1}^{K} \frac{1}{i} \leq 1$. Aber aus genau demselben Grund ist $\sum_{i=K/4+1}^{K/2} \frac{1}{i} \leq 1$ und im allgemeinen Fall ist die Teilsumme in den Grenzen von $K/2^{r+1} + 1$ bis $K/2^r$ durch 1 nach oben beschränkt. Wir erhalten also

$$\sum_{i=1}^{2^r} \frac{1}{i} \leq r + 1$$

und deshalb ist

$$\sum_{i=1}^{K} \frac{1}{i} \leq \sum_{i=1}^{2^{\lceil \log_2 K \rceil}} \frac{1}{i} \leq 1 + \lceil \log_2 K \rceil.$$

Wir setzen unsere Analyse der erwarteten Anzahl der Vergleiche fort und erhalten

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i+1}^{n} E[V_{i,j}] &\leq 2\cdot\sum_{i=1}^{n}\sum_{j=2}^{n-i+1}\frac{1}{j} \\
&\leq \sum_{i=1}^{n}(1+\lceil\log(n-i+1)\rceil) \\
&\leq \sum_{i=1}^{n}(1+\lceil\log(n)\rceil) \\
&= O(n\cdot\log(n)).
\end{aligned}$$

Satz 2.4 *Die erwartete Laufzeit von Quicksort für n Zahlen, bei zufälliger Wahl des Pivotelements, ist $\Theta(n\log_2 n)$. Dieses Ergebnis gilt für* **jede** *Eingabepermutation.*

Das gleiche Ergebnis kann mit fast identischem Argument erhalten werden, wenn wir stets das linkeste Pivotelement wählen und den Erwartungswert aber diesmal über alle Eingabepermutationen bilden.

2.2.2 Weitere Verbesserungen von Quicksort

Eine erste wesentliche Verbesserung besteht, wie gerade gesehen, in der zufälligen Wahl des Pivotelements, da dann die erwartete Laufzeit für jede Eingabepermutation mit der best-case Laufzeit asymptotisch übereinstimmt.

Bemerkung 2.1 Zufallszahlen beschafft man sich über „Pseudo-Zufallsgeneratoren". Ein beliebter Generator basiert auf linearen Kongruenzen:

> Für eine Zahl (seed) x_0 wird die „Zufallszahl" $x_1 = (a\cdot x_0 + b) \mod m$ ausgegeben. Die $(i+1)$-te Zufallszahl x_{i+1} hat die Form
>
> $$x_{i+1} = (a\cdot x_i + b) \mod m.$$

Hierbei sollten Primzahlen (eine beliebte Wahl ist $2^{31}-1$) für den Modulus m benutzt werden.

Versuchen wir als Nächstes, noch mehr Geschwindigkeit aus Quicksort herauszuholen. Wir werden uns dabei vor allem um die folgenden Punkte kümmern

- Beseitigung der Rekursion (bei cleveren Compilern nicht nötig),
- Herunterdrücken der Rekursionstiefe von n auf $\log_2 n$
- und Sonderbehandlung von kleinen Teilarrays.

Die ersten beiden Punkte erledigen wir in einem Aufwasch. Zuerst arbeitet Quicksort die Pivot- und die Partition-Funktion ab, gefolgt von den beiden rekursiven Aufrufen. Eine nichtrekursive Quicksort-Prozedur würde also `pivot` und `partition` ausführen, das „rechte" Teilproblem auf den Stack legen und mit dem „linken" Teilproblem fortfahren. Stattdessen legen wir das längere Teilproblem auf den Stack und fahren mit dem kürzeren Teilproblem fort.

Wir behaupten, dass die maximale Höhe des Stacks durch $\lfloor \log_2 n \rfloor$ beschränkt ist und führen einen induktiven Beweis. (Es gibt also keine Speicherplatzprobleme durch zu große Stacks.)

Zuerst wird ein Problem der Größe $m \geq \frac{n}{2}$ auf den Stack gelegt und Quicksort arbeitet auf einem Problem der Größe $n-m \leq \frac{n}{2}$ weiter. Nach Induktionsannahme wird ein Stack der Höhe höchstens $\lfloor \log_2(n-m) \rfloor \leq \lfloor \log_2 n/2 \rfloor = \lfloor \log_2 n \rfloor - 1$ ausreichen, um das Problem der Größe $n-m$ erfolgreich abzuarbeiten. Die Stackhöhe wird in dieser Zeit also $(\lfloor \log_2 n \rfloor - 1) + 1 = \lfloor \log_2 n \rfloor$ nicht überschreiten. Das zuunterst liegende Problem hat eine Größe kleiner als n und wird nach Induktionsannahme mit Stackhöhe höchstens $\lfloor \log_2 n \rfloor$ abgearbeitet.

Die Implementierung:

```
void quicksort (int links, int rechts) {
    int p, i; stack s(60);
    // Wir benutzen die Klasse stack und nehmen an, dass die Stackhoehe 60
    // nicht ueberschritten wird. (Dies ist nur moeglich, wenn n groesser
    // als eine Milliarde ist!

    for (;;) // Endlosschleife!
    {
        while (rechts > links) {
        {
          p = pivot (links, rechts);
          i = partition (p, links, rechts);
          if (i - links > rechts - i)
            // Das erste Teilproblem ist groesser
            {s.push(links); s.push(i-1); links  = i+1;}
          else
            {s.push(i+1); s.push(rechts); rechts = i-1;}
        }
        if (s.empty ( )) break;
        rechts = s.pop( ); links = s.pop( );
    }
}
```

Wir kommen zur letzten Verbesserung, nämlich der Behandlung kleiner Teilprobleme. Experimentell hat sich herausgestellt, dass für Teilprobleme mit höchstens 25 Zahlen Quicksort kein ernsthafter Wettbewerber ist. Deshalb:

> Sortiere Teilprobleme der Größe höchstens 25 mit Insertion Sort.

2.2.3 Das Auswahlproblem

Wir können aber noch weiteren Profit aus Quicksort für das am Anfang des Kapitels erwähnte Auswahlproblem ziehen. Man erinnere sich, dass die k-kleinste Zahl des Arrays $(A[1], \ldots, A[n])$ zu bestimmen ist. Unsere Vorgehensweise schauen wir Quicksort ab: zuerst wenden wir die Funktionen `pivot` und `partition` an. Sei i die Position des Pivotelements *nach* Durchführung von partition.

Angenommen, wir suchen das s-kleinste Element des Arrays $(A[links], \ldots, A[rechts])$. Wo sitzt es? Wenn $s \leq i - links$, dann wird es im Teilarray A[links], $\ldots, A[i-1]$ zu finden sein. Wenn $s = i-$links$+1$, dann haben wir das s-kleinste Element gefunden: Es stimmt

mit dem Pivotelement überein. Ansonsten müssen wir im Teilarray $(A[i+1], \ldots, A$ [rechts]) weitersuchen. In jedem Fall, wir erhalten nur *ein* Teilproblem. Beachte, dass wir im letzten Fall jetzt nach der

$$[s - (i - \text{links}) - 1]\text{-kleinsten}$$

Zahl zu suchen haben. Wir nennen diese Prozedur *Select*.

Satz 2.5 *Die erwartete Laufzeit von Select für n Zahlen ist für jedes k und für jede Eingabe linear, falls das Pivotelement stets zufällig gewählt wird.*

Wir beschränken uns auf eine Motivation. Sei $S(n)$ die erwartete Laufzeit von Select für n Zahlen. Wenn n genügend groß ist und alle Eingabezahlen verschieden sind, dann wird unser Teilproblem mit Wahrscheinlichkeit mindestens $\frac{1}{2}$ nicht größer als $\frac{3}{4}n$ sein. (Warum? Die Hälfte aller möglichen Pivots garantiert, dass das größte Teilarray nicht größer als $\frac{3}{4}n$ sein wird.) Das „führt“ uns heuristisch auf die Rekursion

$$S(n) \leq S\left(\frac{3}{4}n\right) + c \cdot n$$

Diese Rekursionsgleichung können wir mit unserer allgemeinen Formel lösen und erhalten

$$S(n) = O(n).$$

Aufgabe 10
Zeige, daß zum Finden der zweitkleinsten aus n Zahlen $n + \lceil \log_2 n \rceil$ Vergleiche ausreichen.

Hinweis: Bestimme zuerst das Minimum so, daß nur noch wenige Zahlen Kandidaten für die zweitkleinste Zahl sind.

2.3 Mergesort

Wir werden mit Mergesort ein weiteres Sortierverfahren kennenlernen, dessen worst-case Laufzeit $\Theta(n \log_2 n)$ ist. Mergesort ist besonders geeignet für **externes Sortieren**, also für das Sortieren von Daten auf einem externen Speichermedium.

Im folgenden nehmen wir an, dass n, die Anzahl der zu sortierenden Daten, eine Potenz von 2 ist. Mergesort ist ein rekursives Sortierverfahren. Das Eingabearray $(A[1], \ldots, A[n])$ wird in die beiden Teilarrays $(A[1], \ldots, A[n/2])$ und $(A[n/2+1], \ldots, A[n])$ zerlegt und Mergesort wird rekursiv auf beide Teilarrays angewandt. Damit ist das Sortierproblem auf das **Mischproblem** zurückgeführt: zwei sortierte Arrays sind in ein sortiertes Array zu verschmelzen.

```
void mergesort (int links, int rechts) {
   int mitte;
   if (rechts > links)
     {
       mitte = (links + rechts)/2;
       mergesort (links, mitte);
       mergesort (mitte + 1, rechts);
       merge (links, mitte, rechts);
     }
}
```

Wir müssen noch die Funktion merge erklären:

```
void merge (int links, int mitte, int rechts) {
   int i, j, k;
   // B ist ein global definiertes integer-Array.
   for (i = links; i <= mitte; i++)
     B[i] = A [i];
   // Das erste Teilarray wird nach B kopiert.
   for (i = mitte + 1; i <= rechts; i++)
   B [i] = A [rechts - i + mitte + 1];
   // Das zweite Teilarray wird in umgekehrter Reihenfolge
   // nach B kopiert
   i = links, j = rechts;
   for (k = links; k <= rechts; k++)
     A [k] = (B[i] < B[j])? B[i++] : B[j--];
}
```

Offensichtlich funktioniert Mergesort, wenn die Funktion Merge korrekt ist. Merge sortiert die beiden sortierten Teilarrays, indem es sukzessive nach der kleinsten Zahl in beiden Teilarrays fragt.

Das von uns benutzte Verfahren des Kopierens von A nach B erlaubt es, ohne Endmarken (für das erste bzw. zweite Teilarray) zu arbeiten. Warum funktioniert dieses Verfahren? Die Gefahr ist, dass der i-Index (bzw. der j-Index) seinen Bereich (also den Bereich in B, der dem Teilarray des Index entspricht) verläßt.
Angenommen, das größte Element befindet sich im ersten Teilarray. Offensichtlich besteht keine Gefahr für den i-Index: er wird bis zuletzt in seinem Bereich arbeiten. Aber die akute Gefahr besteht, dass der j-Index seinen Bereich verläßt. Kein Problem, er stößt auf die größte Zahl und muss bis zum Ende warten. Hier zahlt sich die Umkehrung der Reihenfolge aus.

Wir geben als Nächstes eine Laufzeitanlayse an. Gibt es einen Unterschied zwischen best-case und worst-case Laufzeit? Nein, die Laufzeit ist für alle Arrays mit n Zahlen identisch! (Damit sind also best-case, worst-case und erwartete Laufzeit identisch.)

$M(n)$ bezeichne die worst-case Laufzeit von Mergesort für Arrays mit n Zahlen. Als eine erste Beobachtung stellen wir fest, dass Merge Zeit $c \cdot n$ benötigt, um zwei sortierte Arrays der Gesamtlänge n zu mischen. Damit erhalten wir die Rekursion

$$M(n) = 2M(n/2) + c \cdot n$$

und die Lösung $M(n) = \Theta(n \log_2\ n)$.

Satz 2.6 *Die worst-case Laufzeit (wie auch die best-case und die erwartete Laufzeit) von Mergesort ist* $\Theta(n \log_2\ n)$.

Betrachten wir als Nächstes eine nicht-rekursive Version von Mergesort. Mergesort arbeitet seinen Rekursionsbaum in Postorder-Reihenfolge ab: Zuerst werden die beiden Teilprobleme gelöst und dann werden die Lösungen durch Merge zusammengesetzt. Wir könnten also eine nicht-rekursive Version der Postorder-Reihenfolge als Struktur für eine nicht-rekursive Mergesort-Version zu Grunde legen. Aber es geht einfacher.

Wir nehmen wieder an, dass n eine Zweierpotenz ist. Der von Mergesort produzierte Rekursionsbaum ist dann ein vollständiger binärer Baum. Wir denken uns die Knoten des Rekursionsbaumes mit den ihnen entsprechenden Werten von (links, rechts) markiert. Zum Beispiel

wird die Wurzel das Paar $(1, n)$ erhalten, das linke Kind das Paar $(1, \frac{n}{2})$ und das rechte Kind das Paar $(\frac{n}{2} + 1, n)$. Welches Paar erhält das i-te Blatt? Das Paar (i, i)! Der Rekursionsbaum ist also, unabhängig von der Eingabe, stets einfach zu bestimmen.

Wie arbeitet Mergesort? Es bricht zuerst das Sortierproblem in die trivial-lösbaren Einzelprobleme auf und setzt dann die jeweiligen Lösungen gemäß der Postorder-Reihenfolge durch die Merge-Funktion zusammen. Wir können einfacher vorgehen. Zuerst mischen wir nacheinander die Einzellösungen **aller** benachbarten Blätter. Also:

- ```
  for (i = 0; i < n/2; i++)
    mische A[2*i + 1] und A[2*(i + 1)];
  ```
  Was ist die Situation? Die Teilarrays $(A[1], A[2]), (A[3], A[4]), \ldots$ sind sortiert. Wie geht's weiter?

- ```
  for(i = 0; i < n/4; i++)
    mische (A[4*i + 1], A[4*i + 2]) und (A[4*i + 3], A[4*(i + 1)]);
  ```
  Jetzt sind die Teilarrays $(A[1], A[2], A[3], A[4]), (A[5], A[6], A[7], A[8]), \ldots$ sortiert.

- Es sei $z(i) = 2^i$. In der k-ten Phase benutzen wir das Programmsegment

  ```
  for (i=0; i < n/z(k); i++)
    mische (A[i * z(k) + 1],...,A[i * z(k) + z(k-1) ])
    und    (A[i * z(k) + z(k-1) + 1],...,A[(i+1) * z(k)]).
  ```

Diese nicht-rekursive Version ist der rekursiven Version vorzuziehen, da sie den Prozeß des sukzessiven Aufbrechens in Teilprobleme vorwegnimmt: Die nicht-rekursive Version startet ja mit den Basisproblemen.
Der große Nachteil von Mergesort, im Vergleich zu Quicksort, ist die Benutzung des zusätzlichen Arrays B in der Merge-Prozedur, denn Quicksort benötigt im wesentlichen nur einen logarithmisch großen Stack. Andererseits können optimierte Versionen des nicht-rekursiven Mergesort durchaus mit Quicksort vergleichbare Laufzeiten erzielen.

Aufgabe 11
Wir betrachten das allgemeine Problem des Mischens von k aufsteigend sortierten Arrays $A[0], \ldots, A[k-1]$ von jeweils n ganzen Zahlen. **Entwirf** einen effizienten Algorithmus, der das allgemeine Mischproblem löst und **analysiere** seine Laufzeit. Laufzeit $O(n \cdot k \cdot \log_2 k)$ ist erreichbar.

Die Stärke von Mergesort zeigt sich im Problem des externen Sortierens: große Datenmengen, die zum Beispiel auf der Festplatte gespeichert sind, sind zu sortieren. Die zentrale Schwierigkeit ist jetzt die Langsamkeit eines Zugriffs auf die Festplatte, verglichen mit einem Zugriff auf den Hauptspeicher. Ein externes Sortierverfahren muss also versuchen mit möglichst wenigen Zugriffen zu arbeiten.

Wir besprechen als Nächstes das Verfahrens des **ausgeglichenen Mehrwegmischens** am Beispiel von $2b$ Festplatten. Wir beschreiben zuerst die allgemeine Situation:

- Eine Festplatte enthält n zu sortierende Daten $A[1], \ldots, A[n]$.
- $2b$ Festplatten stehen zum Sortieren zur Verfügung und
- $b \cdot m$ Daten können simultan in den Hauptspeicher geladen werden.

Wir beginnen mit dem Schritt des *Vorsortierens*:

```
for (i = 0; i < n/m; i++)
  {
    lade (A[i * m + 1],...,A [(i + 1) * m ]) in den Hauptspeicher
    und sortiere.
    Das sortierte Array wird auf Platte i mod b zurueckgeschrieben.
  }
```

Mit anderen Worten, die ersten m Daten werden, nach ihrer Sortierung, auf Platte 0 geschrieben. Die zweiten m (sortierten) Daten werden auf Platte 1 geschrieben usw. Nach b Schritten erhält auch Platte $b-1$ seine m sortierten Daten. Im Schritt $b+1$ (also $i=b$) erhält Platte 0 seinen zweiten Satz von m sortierten Daten, der anschließend an den ersten Satz abgelegt wird. Das gleiche passiert auch für die restlichen Platten.

Der erste Block von m Daten jeder Platte ist jeweils sortiert und die Blöcke können damit (durch Mehrwegmischen) gemischt werden. Wie? Die ersten Blöcke der Platten $0,\ldots,b-1$ werden in den Hauptspeicher geladen. Im Hauptspeicher werden die ersten m kleinsten Zahlen bestimmt und auf die freie Platte b geschrieben. (Wie? Mit einem Heap, der die jeweils kleinsten Zahlen der b Platten verwaltet!) Dieses Verfahren der sukzessiven Minimumbestimmung und des Schreiben nach Platte b wird solange wiederholt, bis sämtliche der ersten m Zahlen für jede der ersten b Platten verarbeitet wurden.

Was passiert dann? Wir wiederholen unser Vorgehen für den zweiten Block von m Zahlen jeder Platte. Einziger Unterschied: die gefundenen kleinsten Zahlen werden jetzt auf Platte $b+1$ geschrieben. Allgemein: Platte $b+(i \mod b)$ wird als Ausgabeplatte für das Mehrwegmischen der i-ten Blöcke benutzt, wobei wir mit $i=0$ beginnen.

Damit ist die erste Misch-Phase abgeschlossen. Jede der Platten $b, b+1, \ldots, 2b-1$ besteht wiederum aus sortierten Blöcken, aber die Blocklänge ist von m auf $b \cdot m$ angestiegen.

Wie sieht die zweite Misch-Phase aus? Der einzige Unterschied zur ersten Misch-Phase ist, dass nun die Platten $b, b+1, \ldots, 2b-1$ die Rolle der Eingabeplatten spielen, während die Platten $0,\ldots,b-1$ zu Ausgabeplatten werden.

Satz 2.7 *Das ausgeglichene Mehrwegmischen für n Daten, $2b$ Platten und Speicherkapazität $b \cdot m$ des Hauptspeichers benötigt, neben dem Vorsortieren,*

$$\lceil \log_b(n/m) \rceil$$

Phasen. Es wird also insgesamt auf keine Zahl mehr als $1 + \lceil \log_b(n/m) \rceil$ - mal zugegriffen.

Beweis : Wir nehmen induktiv an, dass nach i Phasen die Blocklänge $b^i \cdot m$ erreicht wurde. In Phase $i+1$ werden somit b Blöcke der Länge $b^i m$ zu einem Block der Länge $b^{i+1}m$ verschmolzen. Wir können nach i Phasen abbrechen, falls alle Zahlen sortiert sind. Dies ist gewährleistet, falls $b^i \cdot m \geq n$ oder $b^i \geq n/m$. Das Ergebnis folgt jetzt durch logarithmieren.

Um diese Formel zu interpretieren, ein kleines Rechenexempel: 1 Milliarde Wörter sind zu sortieren, wobei 4 Platten benutzt werden dürfen und 4 Millionen Wörter in den Hauptspeicher passen. Dann sind

$$\lceil \log_2 1000 \rceil = 10$$

Phasen notwendig: Nicht schlecht für eine Milliarde Wörter.

2.4 Eine untere Schranke

Wir haben bisher nur Sortierverfahren kennengelernt, die mindestens $\Omega(n \log_2 n)$ Operationen benötigen, um n Zahlen zu sortieren.

Frage: Benötigt jedes Sortierverfahren $\Omega(n \log_2 n)$ Vergleiche oder haben wir die guten Verfahren noch nicht gefunden?

Bis heute ist diese Frage unbeantwortet! Wir können allerdings eine Antwort geben, wenn wir uns auf *vergleichsorientierte* Sortierverfahren beschränken. Vergleichsorientierte Sortierverfahren (wie Heapsort, Bubble Sort, Insertion Sort, Selection Sort, Quicksort oder Mergesort) greifen auf die Eingabezahlen nur dann zu, wenn ein Vergleich von Zahlen durchzuführen ist. Abhängig von dem Ausgang des Vergleichs wird dann der nächste Vergleich ausgeführt.

Vergleichs-orientierte Sortierverfahren verhalten sich damit genauso wie **Vergleichsbäume**, die wir in der nächsten Definition vorstellen.

Definition 2.8 *Ein Vergleichsbaum für n Eingaben $x_1, \ldots, x_n$ ist ein binärer Baum, so dass jedem inneren Knoten genau ein Vergleich der Form $x_i < x_j$ zugeordnet ist.*

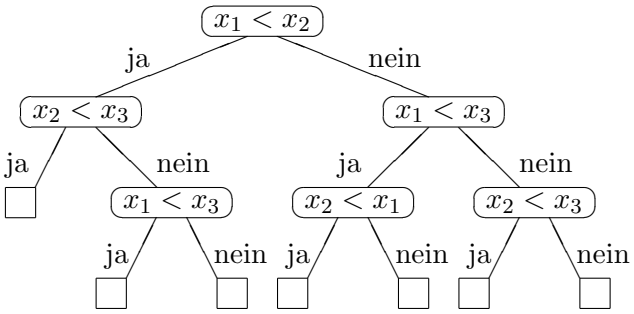
Bei Eingabe $(a_1, \ldots, a_n)$ wird der Baum, mit der Wurzel beginnend, durchlaufen: Falls die Eingabe den Knoten v erreicht hat und falls der Vergleich $x_i < x_j$ auszuführen ist, wird der linke (rechte) Ast gewählt, falls $a_i < a_j$ (bzw. falls $a_i \geq a_j$).
Wir sagen, dass ein Vergleichsbaum sortiert, falls alle Eingaben, die dasselbe Blatt erreichen, denselben Ordnungstyp besitzen. In diesem Fall sprechen wir von einem **Sortierbaum**.

Der **Ordnungstyp** einer Eingabe $a = (a_1, \ldots, a_n)$ ist als die Permutation π definiert, die a sortiert; also

$$a_{\pi(1)} < a_{\pi(2)} < \ldots < a_{\pi(n)}.$$

Ein Vergleichsbaum modelliert also, dass der nächste Vergleich abhängig vom Ausgang des vorherigen Vergleichs gewählt werden kann. Wenn wir ein Sortierverfahren durch einen Vergleichsbaum simulieren wollen, muss der Vergleichsbaum die Sortierbaumeigenschaft besitzen: wenn wir uns an einem Blatt befinden, werden keine weiteren Vergleiche durchgeführt und die Eingabe muss sortiert sein.

Beispiel 2.1 Wir zeigen, dass Bubble Sort sich als Sortierbaum auffassen läßt. Dazu wählen wir $n = 3$.



Nebenbei, dieser Sortierbaum zeigt, dass Bubble Sort nicht der Klügste ist: Der Vergleich $x_2 < x_1$ (im rechten Teilbaum) ist für die sortierte Folge irrelevant.

Wenn ein vergleichsorientiertes Sortierverfahren höchstens $V(n)$ Vergleiche für n Zahlen benötigt, dann wird die Tiefe seines Sortierbaums auch höchstens $V(n)$ sein. Um $V(n) = \Omega(n \log_2 n)$ zu zeigen, genügt es also zu zeigen, dass jeder Sortierbaum Tiefe $\Omega(n \log_2 n)$ besitzt.

Satz 2.9 *Sei T ein Sortierbaum für n Eingaben. Dann gilt*

$$\textit{Tiefe } (T) \geq \log_2(n!) \geq \frac{n}{2}\log_2\frac{n}{2}.$$

Beweis : Ein binärer Baum der Tiefe t hat höchstens 2^t Blätter. (Warum? Beweis durch Induktion nach t). Mit anderen Worten, ein Sortierbaum mit vielen Blättern benötigt eine dementsprechend große Tiefe. Aber warum benötigt ein Sortierbaum viele Blätter? Nach Definition eines Sortierbaumes ist jedem Blatt genau ein Ordnungstyp zugewiesen. Da es für jeden Ordnungstyp aber auch ein zugehöriges Blatt geben muss, gibt es mindestens so viele Blätter wie Ordnungstypen.

Mit anderen Worten, ein Sortierbaum hat mindestens $n!$ Blätter und benötigt deshalb mindestens Tiefe $\log_2(n!)$. Aber

$$\begin{aligned}
\log_2(n!) &\geq \log_2(n\cdot(n-1)(n-2)\ldots(n/2+1)) \\
&\geq \log_2(\frac{n}{2}\cdot\frac{n}{2}\cdot\frac{n}{2}\cdots\frac{n}{2}) \\
&= \log_2\left(\left(\frac{n}{2}\right)^{n/2}\right) \\
&= \frac{n}{2}\log_2\left(\frac{n}{2}\right)
\end{aligned}$$

□

Aufgabe 12
Zeige, daß jeder vergleichsorientierte Sortieralgorithmus mindestens eine erwartete Laufzeit von $\Omega(n\log_2 n)$ benötigt.

Betrachte zur Bestimmung der erwarteten Laufzeit Eingaben der Form $(\pi(1),\ldots,\pi(n))$ für Permutationen π. Die Eingabe $(\pi(1),\ldots,\pi(n))$ tritt mit Wahrscheinlichkeit $\frac{1}{n!}$ auf.

Aufgabe 13
Gegeben sind Folgen von n Zahlen. Diese Folgen bestehen aus n/k Abschnitten von je k Zahlen, wobei für alle i alle Zahlen in Abschnitt i größer als alle Zahlen in Abschnitt $i-1$ sind. Durch getrenntes Sortieren der einzelnen Abschnitte ist es möglich, solche Folgen in Zeit $O(n\log k)$ zu sortieren. **Zeige**, daß jeder vergleichsorientierte Algorithmus für dieses Problem worst-case Laufzeit $\Omega(n\log k)$ benötigt.

Aufgabe 14
In dieser Aufgabe analysieren wir die Komplexität des Sortierproblems bei beschränktem Abstand der Elemente von ihrer endgültigen Position. Wir haben eine Eingabe $a_1,\ldots,a_n$ gegeben. Es gilt $\forall i\ :\ |i-p(i)|\leq d$, wobei $a_{p(1)},\ldots,a_{p(n)}$ die sortierte Folge beschreibt.

(a) **Beweise,** dass jeder vergleichsorientierte Algorithmus zur Sortierung solcher Eingaben eine Worst-Case Laufzeit von $\Omega(n\cdot\log(d))$ hat.

Hinweis: Es ist nicht nötig, die Zahl der Blätter im Entscheidungsbaum exakt zu bestimmen, um diese Schranke herleiten zu können.

(b) **Gib** einen Algorithmus **an**, der Eingaben mit maximalem Abstand d in einer Worst-Case Laufzeit von $O(n\cdot\log(d))$ sortiert.

Aufgabe 15
Beweise, dass jeder vergleichsorientierte Algorithmus zum Finden einer vorgegebenen Zahl in einem sortierten Array eine worst-case Laufzeit von $\Omega(\log(n))$ hat. Die Position der gesuchten Zahl im Array soll ausgegeben werden, falls die Zahl enthalten ist. Andernfalls soll ausgegeben werden, dass die Zahl nicht enthalten ist.

2.5 Distribution Counting, Radix-Exchange und Radixsort

Die in diesem Paragraph betrachteten Sortiermethoden werden alle lineare Laufzeit besitzen, solange die Eingabedaten spezielle Eigenschaften haben. Im wesentlichen werden wir verlangen, dass die n zu sortierenden Zahlen „nicht sehr viel größer“ als n sind.

Damit wird die untere Schranke $\Omega(n \log_2 n)$ geschlagen, die bereits für jedes vergleichsorientierte Sortierverfahren gilt, das Zahlen aus $\{1, \ldots, n\}$ erfolgreich sortiert. Was ist faul? Gar nichts, unsere Sortierverfahren sind nicht vergleichsorientiert. Sie führen sogar keinen einzigen Vergleich durch!

Wir beginnen mit der Sortiermethode **Distribution Counting**. Wir fordern, dass

$$0 \leq A[i] \leq m - 1.$$

Distribution Counting benötigt ein zusätzliches Array Zaehle[m]. Für i mit $0 \leq i \leq m-1$ hält Zaehle[i] fest, wie oft die Zahl i im Array A vorkommt. Dann wird das Array A überschrieben: Die 0 wird Zaehle[0]-mal eingetragen, gefolgt von Zaehle[1]-vielen Einsen, gefolgt von Zaehle[2] vielen Zweien usw.:

```
void counting ( )
  {
    int Zaehle[m]; int wo = 0;
    for (int i = 0; i < m; i++)  //  m Iterationen
       Zaehle[i] = 0;
    for (int j = 1; j <= n; j++) // n Iterationen
       Zaehle[A[j]]++;
    for (i = 0; i < m; i++)  // n + m Iterationen
      {
       for (j = 1; j <= Zaehle[i]; j++)
         A[wo + j] = i;
       wo += Zaehle[i];
      }
   }
```

Satz 2.10 *Distribution-Counting sortiert in Zeit $O(n + m)$.*

Distribution Counting sollte nur für $m = O(n \log_2 n)$ benutzt werden. Für $m \leq n$ hat Distribution Counting lineare Laufzeit und ist der Champion aller Sortierverfahren. Leider ist der Fall $m \leq n$ nicht sehr realistisch.

Aufgabe 16

(a) Wir haben eine Eingabe aus n Zahlen gegeben. Wir wissen, dass die ersten $n - \frac{n}{\log n}$ Zahlen bereits aufsteigend sortiert sind. Lediglich die letzten $\frac{n}{\log n}$ Zahlen sind in keiner Weise vorsortiert. (Man denke etwa an eine große Datenbank, die sortiert gehalten wird, und in die nun eine relativ kleine Menge neuer Datensätze einsortiert werden soll.) **Zeige**, wie man eine so vorsortierte Eingabe in einer Worst-Case Laufzeit von $O(n)$ sortieren kann.

(b) Die Verkehrssünderdatei in Flensburg verwaltet die n Verkehrssünder Deutschlands in irgendeiner Reihenfolge. Für eine statistische Auswertung ist die Behörde aufgefordert, den Verkehrsministerien der einzelnen Bundesländer eine Liste aller dort gemeldeter Verkehrssünder zuzustellen. In welcher Laufzeit kann die Behörde die Daten nach Bundesländern sortieren?

Radix-Exchange berechnet für jedes i die Binärdarstellung von $A[i]$. Radix-Exchange ist in Phasen aufgeteilt, die sehr stark der Partitionsphase von Quicksort ähneln. In seiner ersten Phase verteilt Radix-Exchange die Zahlen gemäß ihres führenden Bits: Zahlen mit führendem Bit 1 folgen auf Zahlen mit führendem Bit 0. Damit ist das Eingabearray in zwei Arrays aufgeteilt worden. Nachfolgende Phasen (Verteilung gemäß dem 2. Bit usw.) verfeinern diese Aufteilung solange, bis eine sortierte Folge erreicht ist.

Satz 2.11 *Für das Eingabearray A gelte, dass $0 \leq A[i] < m$. Dann sortiert Radix-Exchange in Zeit*

$$O(n \log_2 m).$$

Beweis: Radix-Exchange benötigt höchstens $\lceil \log_2 m \rceil$ Phasen und jede Phase läuft in Zeit $O(n)$. Damit ist Radix-Exchange besser als Distribution-Counting, falls $n + m >> n \log_2\ m$. Dies ist der Fall, wenn $m = \omega(n \log_2\ n)$. □

Tatsächlich läßt sich Radix-Exchange beschleunigen, wenn wir statt von der binären Darstellung von der n-ären Darstellung der Zahlen ausgehen. Allerdings steigt der Programmieraufwand und das Sortierverfahren **Radixsort**, das eine ähnliche Idee verfolgt, erlaubt eine einfachere Implementierung.

Wie Radix-Exchange ist auch Radixsort aus Phasen aufgebaut. Sei b eine natürliche Zahl. Radixsort berechnet zuerst die b-äre Darstellung aller Zahlen. (Tatsächlich werden die Ziffern der b-ären Darstellung über die einzelnen Phasen verteilt berechnet, wobei Phase i die i-niedrigstwertige Ziffer berechnet.)

Jede Phase ist aufgebaut aus einer *Verteilungsphase* gefolgt von einer *Sammelphase*. Für Phase i läuft die Verteilungsphase nach der folgenden Vorschrift ab:

```
Ein Array von b leeren Queues wird angelegt.

for (j = 1; j < n; j++)
  fuege A[j] in die Queue mit Nummer s(j,i) ein, wobei s(j,i)
  die i-niedrigstwertige Ziffer von A[j] sei.
```

In der Sammelphase werden sämtliche Queues zurück in das Array A entleert:

```
zaehle =1;
for (j = 0; j < b; j++)
  solange Queue j nicht-leer ist, entferne das Element am Kopf der
  Queue und weise es A[zaehle] zu. Erhoehe zaehle um 1.
```

Phase i ist *stabil*[1]: Wenn $A[s]$ zu Beginn von Phase i vor $A[t]$ erscheint und wenn die i-niedrigstwertige Ziffer von $A[s]$ kleiner oder gleich der i-niedrigstwertigen Ziffer von $A[t]$ ist, dann wird $A[s]$ auch vor $A[t]$ erscheinen, wenn Phase i terminiert. (Beachte, dass Stabilität von den Queues erzwungen wird.)

[1]Wir sagen, dass ein Sortierverfahren *stabil* ist, wenn es die Reihenfolge von Elementen mit gleichem Schlüssel bewahrt. Wird beispielsweise eine nach Vornamen geordnete Namensliste stabil nach den Nachnamen sortiert, so erhält man eine Liste, in welcher Leute mit gleichem Nachnamen nach Vornamen sortiert sind.

Warum ist Stabilität wichtig? Wenn die höchstwertigen Ziffern übereinstimmen, dann entscheiden die restlichen Ziffern den Größenvergleich. Stabilität sichert also, dass die Vergleichsergebnisse bezüglich der niederwertigen Ziffern nicht verloren gehen!

Kommen wir zur Laufzeitanalyse, wobei wir wieder annehmen, dass

$$0 \leq A[i] < m$$

für alle i. Wieviele Phasen müssen durchlaufen werden? So viele Phasen wie Ziffern in der b-ären Darstellen, also höchstens

$$\lceil \log_b m \rceil$$

Phasen. Jede Verteilungsphase läuft in Zeit $O(n)$, jede Sammelphase in Zeit $O(n+b)$:

Satz 2.12 *Für das Eingabearray A gelte, dass $0 \leq A[i] < m$. Dann sortiert Radixsort, mit Basis b, in Zeit*

$$O((n+b) \cdot \log_b m).$$

Aufgabe 17
Sind die Sammelphasen wirklich notwendig?

In Anwendungen sollte man $b \leq n$ wählen und wir erhalten die Laufzeit $O(n \log_b\ m)$. Für $b = n$ erhalten wir $O(n \cdot k)$ als Laufzeit für $m = n^k$, denn es ist $\log_n n^k = k$: Radixsort hat lineare Laufzeit, wenn k eine Konstante ist. Andererseits, wenn $m = n^n$ und wenn wiederum $b = n$ gewählt wird, erhalten wir quadratische Laufzeit: Radixsort versagt für zu große Zahlen, ist aber ein gutes Sortierverfahren für moderat große Zahlen. (Warum sollte b nicht zu groß gewählt werden?)

Aufgabe 18
Die Zahlenfolge

$$19, 45, 61, 8, 54, 23, 17, 34, 57, 49, 16, 12$$

ist durch Radixsort mit $b = 4$ zu sortieren. **Zeige** die Belegung der Schlangen nach jeder Verteilungsphase sowie das Array A nach jeder Sammelphase.

Aufgabe 19
Welche Sortierverfahren aus der Vorlesung sind stabil, welche nicht? **Begründe jede Antwort**.

2.6 Sample Sort: Paralleles Sortieren

Wir nehmen jetzt an, dass wir ein verteiltes System von p eigenständigen Rechnern zur Verfügung haben. Wir möchten die zu sortierende Folge von n Schlüsseln gleichmäßig auf die p Rechner aufteilen und erhoffen uns dann eine möglichst optimale Beschleunigung von Quicksort. Da die erwartete Laufzeit von Quicksort $O(n \log_2 n)$ beträgt, ist unser Ziel somit die Entwicklung einer parallelen Quicksort Variante mit der Laufzeit $O(\frac{n}{p} \cdot \log_2 n)$. Wir erreichen unser Ziel mit Sample Sort, wenn die Zahl p der Rechner nicht zu groß ist.

Die Idee hinter Sample Sort ist einfach: Quicksort ist schon fast ein paralleler Algorithmus, da die Folge der Schlüssel, die kleiner als das Pivotelement sind, und die Folge der Schlüssel, die größer als das Pivotelement sind, unabhängig voneinander sortiert werden können. Allerdings können wir nur zwei Rechner beschäftigen. Wenn wir hingegen p Rechnern zur Verfügung haben, dann liegt es nahe,

- nicht ein, sondern $p-1$ Pivotelemente zu wählen,

- dann parallel die n Schlüssel, entsprechend den $p-1$ Pivotelementen in jeweils p Intervalle aufzuteilen,
- parallel $p-1$ Intervalle zu versenden (und ein Intervall zu behalten) und
- schließlich die erhaltenen und das verbliebene Interval parallel zu sortieren.

Wir verfolgen diese Idee, allerdings mit einer kleinen Modifikation.

- In einem ersten Schritt sortieren die Rechner ihre $\frac{n}{p}$ Schlüssel parallel,
- kooperieren dann um eine gute Stichprobe S von $p-1$ Pivotelementen zu berechnen,
- zerlegen ihre sortierten Teilfolgen gemäß S in p Intervalle, versenden $p-1$ Intervalle und behalten ein Intervall,
- um die erhaltenen und verbliebenen Schlüssel dann wieder durch Mischen zu sortieren.

Die zweite Variante erlaubt die Berechnung einer guten Stichprobe, wie die folgende Aufgabe zeigt.

Aufgabe 20
Nachdem die Rechner ihre $\frac{n}{p}$ Schlüssel sortiert haben, wählt jeder Rechner Schlüssel in den Positionen $i \cdot \frac{n}{p^2}$ für $i = 0, \ldots p-1$ als seine Stichprobe und sendet diese Stichprobe an Rechner 1, der dann alle p^2 Schlüssel sortiert. Rechner 1 bestimmt die endgültige Stichprobe S, indem alle Schlüssel in den Positionen $i \cdot p$ für $i = 1, \ldots, p-1$ ausgewählt werden.

Wieviele Schlüssel erhält jeder Rechner in Schritt (4) von Algorithmus 2.1 höchstens?

Die Details der zweiten Variante zeigen wir in der folgenden Beschreibung.

Algorithmus 2.1 **Sample Sort.**

/* Die Folge $A = (A[1], \ldots, A[n])$ ist mit p Rechnern zu sortieren. Die Folge A wird in p gleich große Intervalle der Länge $\frac{n}{p}$ aufgeteilt. */

(1) Jeder Rechner sortiert seine $\frac{n}{p}$ Schlüsssel mit Quicksort und bestimmt eine Stichprobe der Größe s, die an Rechner 1 gesandt wird.

/* Zeit $O(\frac{n}{p} \log_2 \frac{n}{p})$ genügt für das sequentielle Sortieren. */

(2) Rechner 1 sortiert die erhaltenen $(p-1) \cdot s$ Schlüssel zusammen mit den eigenen s Schlüsseln und bestimmt eine endgültige Stichprobe S der Größe $p-1$. Die Stichprobe S wird durch einen Broadcast an die restlichen Rechner verteilt.

/* In Zeit $O(ps \log_2 ps)$ werden die Schlüssel aus den p Stichproben sortiert. In der Stichprobe $S = \{s_1 < s_2 < \cdots < s_{p-1}\}$ befinden sich die Schlüssel zu den Rängen $s, 2 \cdot s, \ldots, (p-1) \cdot s$. */

(3) Jeder Rechner partitioniert seine Schlüssel in die p durch S definierten Intervalle $I_1 =]-\infty, s_1[, I_2 = [s_1, s_2[, \ldots, I_p = [s_{p-1}, \infty[$.

Dann werden alle Schlüssel, die im Intervall I_j liegen, an Rechner j weitergeleitet.

/* Die Partionierung gelingt in Zeit $O(\frac{n}{p} \cdot \log_2 p)$, denn die Schlüssel wurden ja schon in Schritt (1) sortiert. */

(4) Jeder Rechner mischt die erhaltenen p sortierten Folgen.

/* Zeit $O(\frac{n}{p} \cdot \log_2 p)$ reicht aus, wenn jeder Rechner ungefähr $\frac{n}{p}$ Schlüssel nach Schritt (3) erhält. Warum kann so schnell gemischt werden? */

Wir nehmen $p \cdot s \leq n/p$, beziehungsweise $s \leq n/p^2$, an. Dann ist die benötigte Zeit höchstens $O(\frac{n}{p} \cdot \log_2 \frac{n}{p} + \frac{n}{p} \cdot \log_2 p + \frac{n}{p} \cdot \log_2 p) = O(\frac{n}{p} \log_2 n)$. Wenn wir, wie in der Aufgabe vorgeschlagen, $s = p$ setzen, dann müssen wir $p \leq n^{1/3}$ fordern, und diesem Fall ist die Zeit durch $O(\frac{n}{p} \cdot \log_2 n)$ beschränkt.

Satz 2.13 *Für $p \leq n^{1/3}$ sortiert Sample Sort n Schlüssel mit p Rechnern in Zeit $O(\frac{n}{p} \cdot \log_2 n)$.*

Beachte, dass wir allerdings die Kommunikationskosten unterschlagen haben.

2.7 Zusammenfassung

Wir haben die wichtigsten Sortierverfahren besprochen. Elementare Sortierverfahren wie Bubble Sort, Insertion Sort oder Selection Sort haben quadratische Laufzeit und sind für große Datenmengen indiskutabel. Die beliebteste Sortiermethode, Quicksort, hat auch eine quadratische worst-case Laufzeit, doch ist die erwartete Laufzeit durch $O(n \log_2\ n)$ beschränkt. Durch Verbesserungen wie

- Beseitigung der Rekursion,
- Reduzierung der Rekursionstiefe auf $\log_2 n$
- Sonderbehandlung kleiner Teilprobleme und
- eine clevere Pivotwahl

ist Quicksort nur schwer zu schlagen. Eine Version von Quicksort fand eine weitere Anwendung für das Auswahlproblem. Mergesort, wie auch Heapsort, besitzt $O(n \log_2\ n)$ auch als worst-case Laufzeit. Die nicht-rekursive Version von Mergesort erzielt in der Praxis ebenfalls sehr gute Ergebnisse (nach weiteren Verbesserungen) und ist vor allem für das externe Sortierproblem das dominierende Sortierverfahren.

Weiterhin haben wir gesehen, dass jedes vergleichsorientierte Sortierverfahren mindestens $\log_2(n!) = \Theta(n \log_2\ n)$ Vergleiche durchführen muss. Damit sind Heapsort und Mergesort asymptotisch-optimale vergleichsorientierte Sortierverfahren.

Wenn wir die Größe der Eingabezahlen beschränken, erhalten wir allerdings schnellere Sortierverfahren, nämlich Distribution Counting und Radixsort. Wegen der eingeschränkten Größe der Eingabezahlen müssen diese Sortierverfahren aber als „special purpose“ bezeichnet werden.

Kapitel 3

Graphalgorithmen

Definition 3.1 *(a) Ein* **ungerichteter Graph** $G = (V, E)$ *besteht aus einer endlichen Knotenmenge* V *und einer Kantenmenge* $E \in \{\{i, j\} \mid i, j \in V, i \neq j\}$.

(b) Ein **gerichteter Graph** $G = (V, E)$ *besteht ebenfalls aus einer endlichen Knotenmenge* V *und einer Kantenmenge* $E \in \{(i, j) \mid i, j \in V, i \neq j\}$.

Wir erinnern zuerst an die wichtigsten graph-theoretischen Begriffe.

Definition 3.2 *Sei* $G = (V, E)$ *ein gerichteter oder ungerichteter Graph.*

(a) *Zwei Knoten* $u, v \in V$ *heißen* **adjazent**, *falls sie durch eine Kante verbunden sind.*

(b) *Eine Kante ist mit einem Knoten* **inzident**, *wenn der Knoten ein Endpunkt der Kante ist. Wenn* $\{u, v\} \in E$ *eine Kante in einem ungerichteten Graphen ist, dann heißt* u *ein* **Nachbar** *von* v. *Ist* $(u, v) \in E$ *eine Kante in einem gerichteten Graphen, dann heißt* v *ein* ***direkter Nachfolger*** *von* u.

(c) *Eine Folge* $(v_0, v_1, ..., v_m)$ *heißt ein* **Weg in** G, *falls für jedes* i $(0 \leq i < m)$

- $(v_i, v_{i+1}) \in E$ *(für gerichtete Graphen) oder*
- $\{v_i, v_{i+1}\} \in E$ *(für ungerichtete Graphen).*

Die **Weglänge** *ist* m, *die Anzahl der Kanten. Ein Weg heißt* **einfach**, *wenn kein Knoten zweimal auftritt. Ein Weg heißt ein* **Kreis**, *wenn* $v_0 = v_m$ *und* $(v_0, ...v_{m-1})$ *ein einfacher Weg ist.*

(d) *Eine Teilmenge* $V' \subseteq V$ *ist* **zusammenhängend**, *wenn es für je zwei Knoten* $v, w \in V'$ *einen Weg von* v *nach* w *gibt.* V' *heißt eine* **Zusammenhangskomponente** *von* G, *falls* V' *zusammenhängend ist, aber keine echte Obermenge von* V' *zusammenhängend ist.*

Graphen werden vor Allem in der Modellierung eingesetzt, wobei Rechnernetze, Schaltungen, Verkehrsverbindungen oder das Internet zu den wichtigsten Beispielen gehören. In der Vorlesung „Datenstrukturen“ haben wir Graphen mit **Adjazenzlisten** dargestellt, um einerseits Nachbarn oder direkte Nachfolger eines Knotens schnell zu bestimmen und um andererseits Graphen proportional zu ihrer Größe darstellen zu können. Wir erinnern an die Definition der Adjazenzliste eines ungerichteten Graphen $G = (V, E)$ als ein Array A_G von Listen, wobei die Liste $A_G[v]$ alle Nachbarn von v aufführt. In gerichteten Graphen sind alle direkten Nachfolger

w von v in der Liste $A_G[v]$ aufgeführt. Der Speicherplatzbedarf beträgt $\Theta(|V| + |E|)$ und ist damit *proportional* zur Größe des Graphen. Wichtig ist, dass die Bestimmung der Nachbarn, bzw. der Nachfolger von v, in schnellstmöglicher Zeit gelingt, nämlich in Zeit proportional zur Anzahl der Nachbarn, bzw. der direkten Nachfolger.

Weiterhin haben wir mit **Tiefensuche** und **Breitensuche** zwei Verfahren vorgestellt, um Graphen zu durchsuchen, also alle Knoten des Graphen zu besuchen. Der Vollständigkeit halber stellen wir die wesentlichen Ergebnisse nochmal im folgenden Abschnitt zusammen.

Wir möchten zwei weitere grundlegende Probleme lösen, nämlich die **Bestimmung kürzester Wege** bei beliebigen, nicht-negativ gewichteten Kantenlängen sowie die **Bestimmung minimaler Spannbäume**. Bisher kennen wir nur die Algorithmen, nämlich den Algorithmus von Dijkstra sowie die Algorithmen von Prim und Kruskal. In diesem Kapitel werden wir zeigen, dass diese Algorithmen korrekt sind, also „genau das tun, was sie sollen“.

3.1 Suche in Graphen

Wir beginnen mit der Tiefensuche für ungerichtete und gerichtete Graphen.

3.1.1 Tiefensuche

Wir nehmen an, dass der (gerichtete oder ungerichtete) Graph durch seine Adjazenzliste repräsentiert ist. Die Adjazenzliste bestehe aus einem Array $A_$Liste von Zeigern auf die Struktur Knoten mit

```
struct Knoten {
  int name;
  Knoten * next;}
```

Tiefensuche wird wie folgt global organisiert:

```
void Tiefensuche()
{
  for (int k = 0; k < n; k++) besucht[k] = 0;
  for (k = 0; k < n; k++)
    if (! besucht[k]) tsuche(k);
}
```

Die globale Organisation garantiert, dass jeder Knoten besucht wird: Es ist nämlich möglich, dass innerhalb des Aufrufes tsuche(0) nicht alle Knoten besucht werden. Deshalb sind weitere Aufrufe notwendig.

Beachte, dass wir `tsuche(v)` nur dann aufrufen, wenn v als nicht besucht markiert ist. Unsere erste Aktion innerhalb eines Aufrufs von tsuche ist deshalb eine Markierung von v als besucht. Danach durchlaufen wir die Adjazenzliste von v und besuchen rekursiv alle Nachbarn (oder Nachfolger) von v, die noch nicht besucht wurden. Unsere sofortige Markierung von v verhindert somit einen mehrmaligen Besuch von v.

```
void tsuche(int v)
{
   Knoten *p ; besucht[v] = 1;
   for (p = A_Liste [v]; p !=0; p = p->next)
     if (!besucht [p->name]) tsuche(p->name);
}
```

Wir kommen als Nächstes zu den zentralen Eigenschaften von Tiefensuche für ungerichtete Graphen.

Satz 3.3 *Sei $G = (V, E)$ ein ungerichteter Graph. W_G sei der Wald der Tiefensuche für G.*

(a) Wenn $\{u, v\} \in E$ eine Kante ist und wenn tsuche(u) vor tsuche(v) aufgerufen wird, dann ist v ein Nachfahre von u in W_G.

Die Bäume von W_G entsprechen genau den Zusammenhangskomponenten von G.

(b) Tiefensuche besucht jeden Knoten von V genau einmal. Die Laufzeit von Tiefensuche ist linear, also durch

$$O(|V| + |E|)$$

beschränkt.

(c) G besitzt nur Baum- und Rückwärtskanten.

Aufgabe 21
Warum können wir in Zeit $O(|V| + |E|)$ überprüfen, ob der ungerichtete Graph $G = (V, E)$ ein Baum ist, zusammenhängend oder bipartit ist?

Warum kann mit Tiefensuche schnell aus einem Labyrinth finden?

Die Situation für gerichtete Graphen ist ein wenig komplizierter.

Satz 3.4 *Sei $G = (V, E)$ ein gerichteter Graph.*

(a) Für jeden Knoten v in V gilt: tsuche(v) wird genau die Knoten besuchen, die auf einem unmarkierten Weg mit Anfangsknoten v liegen. (Ein Weg ist unmarkiert, wenn alle Knoten vor Beginn von tsuche(v) unmarkiert sind).

(b) Die Laufzeit von Tiefensuche() ist linear, also durch

$$O(|V| + |E|)$$

beschränkt.

(c) G besitzt nur Baumkanten, Rückwärts- und Vorwärtskanten sowie Rechts-nach-Links Querkanten, also Querkanten die von einem später zu einem früher besuchten Knoten führen.

Aufgabe 22
Zeige dass die folgenden Probleme in Zeit $O(|V| + |E|)$ für gerichtete Graphen gelöst werden können:
Gibt es einen Weg von Knoten u nach Knoten v, ist G kreisfrei oder ist G stark zusammenhängend, gibt es also für je zwei Knoten stets einen Weg vom ersten zum zweiten Knoten?

3.1.2 Breitensuche

Breitensuche folgt dem „breadth first“-Motto: Für jeden Knoten werden zuerst alle Nachfolger besucht, gefolgt von der Generation der Enkel und so weiter. Die Knoten werden also in der Reihenfolge ihres Abstands von v, dem Startknoten der Breitensuche, erreicht.

```
void Breitensuche(int v) {
  Knoten *p; int w;
  queue q; q.enqueue(v);
  for (int k =0; k < n ; k++)
    besucht[k] = 0;
  besucht[v] = 1;
  /* Die Schlange q wird benutzt. v wird in die Schlange eingefuegt
  und markiert. */

  while (!q.empty ( ))
  {
    w = q. dequeue ( );
    for (p = A_List[w]; p != 0; p = p->next)
      if (!besucht[p->name])
      {
        q.enque(p->name);
        besucht[p->name] = 1;
      }
  }
}
```

Breitensuche(v) erzeugt einen Baum T_v: Anfänglich ist T_v leer. Wenn in Breitensuche(v) ein Knoten u aus der Schlange entfernt wird und ein Nachbar (bzw. Nachfolger) u' in die Schlange eingefügt wird, dann fügen wir die Kante $\{u, u'\}$ (bzw. (u, u')) in T_v ein.

Satz 3.5 *Sei $G = (V, E)$ ein gerichteter oder ungerichteter Graph, der als Adjazenzliste vorliegt. Sei v ein Knoten von G.*

(a) Breitensuche(v) besucht jeden von v aus erreichbaren Knoten genau einmal und sonst keinen anderen Knoten.

Der von Breitensuche(v) erzeugte Baum $T(v)$ ist ein Baum kürzester Wege: Wenn w von v aus erreichbar ist, dann ist w ein Knoten in $T(v)$ und der Weg in $T(v)$, von der Wurzel v zum Knoten w, ist ein kürzester Weg.

(b) Breitensuche(v) besucht jeden von v aus erreichbaren Knoten genau einmal. Die Laufzeit ist linear, also proportional in der Anzahl von v erreichbaren Knoten und der Gesamtzahl ihrer Kanten.

Breitensuche ist also ebenso effizient wie Tiefensuche. Für ungerichtete Graphen wird Breitensuche(v) (ebenso wie tsuche(w)) die Zusammenhangskomponente von v besuchen. Auch für gerichtete Graphen zeigen Breitensuche und Tiefensuche ähnliches Verhalten: Breitensuche(v) und tsuche(v) besuchen alle von v aus erreichbaren Knoten. Jeder hat seine Stärken und Schwächen: Während Tiefensuche problemlos feststellen kann, ob ein gerichteter Graph kreisfrei ist, bereitet Kreisfreiheit der Breitensuche großes Kopfzerbrechen, aber Breitensuche rächt sich mit einer schnellen Bestimmung kürzester Wege.

3.2 Kürzeste Wege

Zuerst beschreiben wir die Problemstellung. Ein gerichteter Graph $G = (V, E)$ ist vorgegeben. Jeder Kante wird durch die Funktion

$$\text{länge} : E \to \mathbb{R}_{\geq 0}$$

eine nicht-negative Länge zugewiesen. Für einen Knoten $s \in V$ suchen wir kürzeste Wege von s zu allen anderen Knoten in V. Die Länge des Weges $p = (v_0, v_1, \ldots, v_m)$ ist dabei definiert durch

$$\text{länge}(p) = \sum_{i=1}^{m} \text{ länge } (v_{i-1}, v_i).$$

Damit haben wir das Problem der Berechnung kürzester Wege, das wir mit Breitensuche bereits gelöst haben, auf das Problem der Berechnung kürzester gewichteter Wege verallgemeinert. Beachte, dass das verallgemeinerte Problem realistischer ist. Für die Erstellung schnellster Verbindungen bei der deutschen Bahn ist nicht die Anzahl der Bahnhöfe interessant, an denen der Zug hält, sondern die benötigten Zeitspanne, um von einem Bahnhof zum nächsten zu gelangen.

Es scheint sinnvoller, eine einfachere Problemvariante zu betrachten, nämlich das Problem für gegebene Knoten s und t einen kürzesten Weg von s nach t zu finden. Bisher sind aber keine Lösungsansätze bekannt, die nicht gleichzeitig auch kürzeste Wege von s zu allen anderen Knoten bestimmen. Ein Sprung in der asymptotischen Komplexität besteht erst zwischen der Aufgabe, von einen Startknoten s aus die kürzesten Wege zu allen anderen Knoten zu finden, einerseits und der Aufgabe, die kürzesten Wege zwischen allen Punkten zu finden, andererseits. Wir werden uns mit beiden Varianten beschäftigen. Zunächst der Fall mit ausgezeichnetem Startknoten, das **SSSP-Problem** (*single source shortest path*).

Wir beschreiben Dijkstras Algorithmus. Sei $S \subseteq V$ vorgegeben und es gelte $s \in S$. Wir nehmen an, dass wir bereits kürzeste Wege von s zu allen Knoten in S kennen. Darüberhinaus nehmen wir an, dass wir für jeden Knoten $w \in V \setminus S$ den Wert

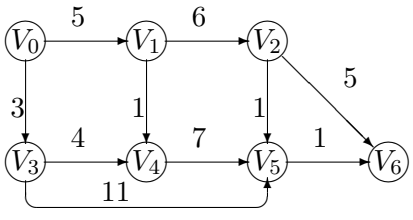
> distanz (w) = Länge eines kürzesten Weges W von s nach w, so dass W, mit Ausnahme des Endknotens w, nur Knoten in S durchläuft.

kennen. Dijkstras Algorithmus wird dann einen Knoten $v \in V - S$ mit kleinstem Distanz-Wert suchen und behaupten, dass

$$\text{distanz}[v] = \text{Länge eines kürzesten Weges von } s \text{ nach } v.$$

Der Knoten v wird dann zu der Menge S hinzugefügt und die Distanz-Werte für Knoten in $V \setminus S$ werden aktualisiert. Ist ein Knoten $w \notin S$ nicht über einen solchen fast nur in S verlaufenden Weg erreichbar, so werden wir einfach $distanz(w) = \infty$ setzen.

Beispiel 3.1 Wir betrachten den Graphen



Es sei $s = V_0$ und $S = \{V_0, V_3, V_1\}$. Dann ist

$$\text{distanz}[V_2] = 11,\ \text{distanz}[V_4] = 6,\ \text{distanz}[V_5] = 14,\ \text{distanz}[V_6] = \infty.$$

Knoten V_4 ist der Knoten mit kleinstem Distanzwert. Dijkstras Algorithmus wird sich deshalb festlegen und behaupten, dass 6 auch die Länge des kürzesten Weges von V_0 nach V_4 ist.

Knoten V_4 wird in die Menge S eingefügt. Dann sind aber die Distanzwerte der Nachfolger von V_4 neu zu bestimmen. Knoten V_5 ist der einzige Nachfolger von V_4 und wir setzen

$$\begin{aligned}&distanz[V_5] = \\ &(distanz[V_5] > distanz[V_4] + \text{länge}(V_4, V_5))\ ?\ distanz[V_4] + \text{länge}(V_4, V_5) : distanz[V_5];\end{aligned}$$

In diesem Fall sinkt der Distanzwert von Knoten V_5 von 14 auf 13. Wir haben einen kürzeren Weg von V_0 nach V_5 gefunden, indem wir zuerst einen kürzesten Weg von s nach V_4 laufen und dann die Kante (V_4, V_5) wählen. Wir haben also jetzt die Distanzwerte

$$distanz[V_2] = 11, distanz[V_5] = 13, distanz[V_6] = \infty$$

erhalten. Dijkstras Algorithmus wiederholt sein Vorgehen: Der Algorithmus bekennt sich zu Knoten V_2 und behauptet, dass 11 auch die Länge eines kürzesten Weges von V_0 nach V_2 ist. Die Distanzwerte der Nachfolger von V_2 werden neu berechnet und wir erhalten

$$distanz[V_5] = 12, distanz[V_6] = 16.$$

Wir haben also wiederum einen kürzeren Weg von V_0 nach V_5 gefunden: Diesmal verfolgen wir einen kürzesten Weg von V_0 nach V_2 und setzen ihn mit Kante (V_2, V_5) fort. Im nächsten Schritt legt sich der Algorithmus auf Knoten V_5 fest und wird einen kürzeren Weg von V_0 nach V_6 finden, indem zuerst der kürzeste Weg von V_0 nach V_5 eingeschlagen wird. Der Algorithmus endet mit der Festlegung auf Knoten V_6.

Algorithmus 3.1 **Dijkstras Algorithmus** für $(G = (V, E))$:

(1) Setze $S = \{s\}$ und

$$\text{distanz}[v] = \begin{cases} \text{länge}(s, v) & \text{wenn } (s, v) \in E, \\ \infty & \text{sonst.} \end{cases}$$

(2) Solange $S \neq V$ wiederhole

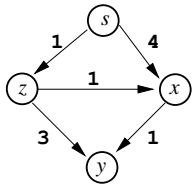
(2a) wähle einen Knoten $w \in V \setminus S$ mit kleinstem Distanz-Wert.

(2b) Füge w in S ein.

(2c) Berechne die neuen Distanz-Werte der Nachfolger von w. Insbesondere setze für jeden Nachfolger $u \in V \setminus S$ von w

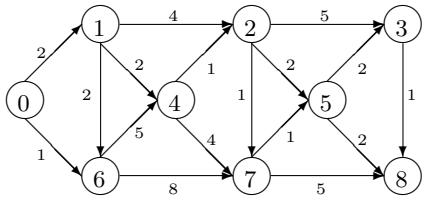
```
        c = distanz[w] + laenge(w,u);
distanz[u] = (distanz[u] > c) ? c : distanz[u];
```

Aufgabe 23
Beschreibe den Verlauf von Dijkstras Algorithmus auf folgendem Graphen:



Wähle dabei s als Startknoten.

Aufgabe 24
Gegeben sei der folgende gewichtete Graph G:



Berechne die Länge der kürzesten Wege von Knoten 0 zu allen anderen Knoten aus G nach dem Algorithmus von Dijkstra. Bestimme nach jedem Schritt des Algorithmus die Menge S sowie das *distanz*-Array (wenn sich das Array geändert hat).

Aufgabe 25
Wir möchten eine Variante des Problems der Bestimmung kürzester Wege betrachten. Für einen gerichteten Graphen $G = (V, E)$ und einen Startknoten $s \in V$, sind wieder kürzeste Wege von s zu allen anderen Knoten zu bestimmen. Allerdings: Wenn es für einen Knoten $v \in V$ mehr als einen kürzesten Weg von s nach v gibt, dann ist ein kürzester Weg mit einer kleinsten Anzahl von Kanten zu bestimmen.

Beschreibe eine Modifikation von Dijkstras Algorithmus, die diese Variante so effizient wie möglich löst.

Wir führen zuerst einen **Korrektheitsbeweis:** Wir sagen, dass ein Weg W ein S-Weg ist, wenn W im Knoten s beginnt und bis auf seinen Endknoten nur Knoten in S durchläuft. Der Korrektheitsbeweis wird die folgende Schleifeninvariante nachweisen:

> für jedes $u \in V \setminus S$ ist
> distanz $[u]$ = die Länge eines kürzesten S-Weges mit Endpunkt u.

Offensichtlich ist diese Invariante zu Anfang erfüllt, denn S-Wege stimmen zu Anfang mit Kanten überein. Angenommen, die Schleifeninvariante gilt und ein Knoten w (mit kleinstem Distanz-Wert unter den Knoten in $V \setminus S$) wird zu S hinzugefügt.

Zwischenbehauptung: distanz $[w]$ = Länge eines kürzesten Weges von s nach w

Beweis : Angenommen, es gibt einen kürzeren Weg

$$p = (s, v_1, \ldots, v_i, v_{i+1}, \ldots, w).$$

Sei v_{i+1} der erste Knoten in p, der nicht zu S gehört. Dann ist

(1) distanz $[w] \leq$ distanz $[v_{i+1}]$, denn w hatte ja den kleinsten Distanz-Wert unter allen Knoten in $V \setminus S$.

(2) Da die Invariante gilt, ist distanz $[v_{i+1}] \leq$ länge (p), denn der Teilweg $(s, v_1, \ldots, v_i, v_{i+1})$ ist ein S-Weg und distanz $[v_{i+1}]$ ist die Länge des kürzesten S-Weges.

Aus (1) und (2) folgt aber

$$\text{distanz}[w] \leq \text{ länge}(p)$$

und p ist nicht kürzer! □

Dijkstras Algorithmus fügt w zur Menge S hinzu. Sei u ein beliebiger Knoten in $V \setminus S$. Wie sieht ein kürzester S-Weg für u aus?
Angenommen $(s, \ldots, w, v, \ldots, u)$ ist ein kürzester S-Weg. Dann ist auch $(s, \ldots, w, v)$ ein kürzester S-Weg nach v. v wurde aber vor w in S eingefügt und sein damalig kürzester S-Weg p' war auch ein kürzester Weg von s nach v, der natürlich w nicht durchläuft. Wir erhalten damit einen garantiert ebenso kurzen S-Weg nach u, wenn wir $(s, \ldots, w, v)$ durch p' ersetzen.
Die Hinzunahme von w zu S bewirkt also nur dann kürzere S-Wege nach u, wenn w der vorletzte Knoten des S-Weges ist. Damit ist aber u ein Nachfolger von w. Der Algorithmus ändert aber die Distanz-Werte der Nachfolger von w, falls erforderlich, und garantiert so korrekte Distanz-Werte für die nächste Iteration. Damit ist der Korrektheitsbeweis abgeschlossen.

Aufgabe 26
Beweise oder **widerlege** die folgende Aussage: Dijkstras Algorithmus arbeitet korrekt für Graphen, bei denen auch negative Kantengewichte zugelassen sind, *solange* keine Kreise negativer Länge existieren.

Jetzt zur Laufzeit. Die meiste Arbeit wird in der Schleife geleistet. Die wesentlichen Operationen ($|V| - 1$ mal wiederholt) sind

- die Wahl eines Knotens mit kleinstem Distanz-Wert in Schritt (2a) und
- die Berechnung der neuen Distanz-Werte seiner Nachfolger in Schritt (2c).

Wenn wir annehmen, dass G als Adjazenzliste repräsentiert ist, finden wir die Nachfolger ohne jede Mühe. Wie finden wir einen Knoten mit kleinstem Distanz-Wert? Wenn wir einen Heap zur Verfügung stellen, wobei

- die Priorität durch den Distanz-Wert definiert ist
- und die Ordnung umgekehrt wird, so dass die Operationen insert und deletemin unterstützt werden.

Aber wie können veränderte Distanz-Werte „dem Heap mitgeteilt werden“, ohne eine lineare Suche nach den zu ändernden Distanz-Werten auszulösen?

Zwei Alternativen bieten sich an. In der ersten Alternative wird ein Array *Adresse* zur Verfügung gestellt, das für jeden Knoten $v \in V \setminus S$ die Adresse von v im Heap speichert. Die Heap-Operation change_priority wird dann die Änderung des Distanz-Wertes ausführen.

Die zweite Alternative ist „quick-and-dirty“: Wann immer sich ein Distanz-Wert für Knoten v ändert, füge v mit dem neuen Distanzwert in den Heap ein. Ein Knoten ist dann möglicherweise mit vielen verschiedenen Distanzwerten im Heap vertreten, die deletemin-Operation wird aber stets den kleinsten Distanzwert ermitteln, und die quick-and-dirty Version arbeitet korrekt, wenn wir deletemin-Ausgaben, die sich auf Knoten in S beziehen, mit strenger Ignoranz strafen.

Man beachte, dass in jedem Fall höchstens $2 \cdot (|V| + |E|)$ Operationen auf dem Heap durchgeführt werden: Mit $|V| - 1$ Einfüge-Operationen wird der Heap initialisiert. Jede Neuberechnung eines Distanzwertes wird durch eine Kante des Graphen verursacht und eine Kante veranlasst höchstens eine Neuberechnung. Also werden werden höchstens $|E|$ Neuberechnungen durchgeführt und zusätzlich sind höchstens $|V| + |E|$ deletemin-Operationen erforderlich.

Da eine Heap-Operation in Zeit $O(\log_2(|V| + |E|)) = O(\log_2 |V|)$ gelingt, folgt

Satz 3.6 *Sei $G = (V, E)$ ein gerichteter Graph. Dijkstras Algorithmus berechnet alle kürzesten Wege von s zu allen anderen Knoten in G in Zeit*

$$O((|V| + |E|) \log_2 |V|).$$

Bemerkung 3.1 **(a)** Was passiert, wenn stets länge $(e) = 1$ für alle Kanten $e \in E$? Dijkstras Algorithmus wird die Menge S gemäß der Breitensuche-Reihenfolge berechnen! Diese Beobachtung zeigt auch, dass Dijkstras Algorithmus im wesentlichen nur die Schlange der Breitensuche durch eine Prioritätswarteschlange ersetzt!

(b) Wir haben die Länge der kürzesten Wege, nicht aber die kürzesten Wege selbst berechnet. Wie sind diese kürzesten Wege zu berechnen? Wir können uns bei der Speicherung eines kürzesten Weges von s nach u auf die Angabe des unmittelbaren Vorgängers w von u auf irgendeinem kürzesten Weg nach u beschränken. Kennen wir w und einen kürzesten Weg von s nach w, so haben wir einen kürzesten Weg nach w rekonstruiert. Die Angabe des Vorgängers erlaubt jetzt eine sukzessive Rekonstruktion Kante nach Kante.

Wie bestimmen wir den unmittelbaren Vorgänger auf einem kürzesten Weg? Wir repräsentieren kürzeste Wege durch einen Baum B in Vaterdarstellung. Insbesondere setzen wir genau dann Vater$[u] = w$ in Schritt (2c) von Dijkstras Algorithmus, wenn distanz$[u]$ sich verringert, wenn wir zuerst einen kürzesten Weg von s nach w laufen und dann die Kante (w, u) benutzen.

Am Beispiel des obigen Graphen bekommen wir den folgenden Ablauf:

	V_0		V_1		V_2		V_3		V_4		V_5		V_6	
S	d	p	d	p	d	p	d	p	d	p	d	p	d	p
$\{V_0\}$	0	-	5	V_0	∞		3	V_0	∞		∞		∞	
$\{V_0, V_3\}$	0	-	5	V_0	∞		3	V_0	7	V_3	14	V_3	∞	
$\{V_0, V_1, V_3\}$	0	-	5	V_0	11	V_1	3	V_0	6	V_1	14	V_3	∞	
$\{V_0, V_1, V_3, V_4\}$	0	-	5	V_0	11	V_1	3	V_0	6	V_1	13	V_4	∞	
$\{V_0, V_1, V_2, V_3, V_4\}$	0	-	5	V_0	11	V_1	3	V_0	6	V_1	12	V_2	16	V_2
$\{V_0, V_1, V_2, V_3, V_4, V_5\}$	0	-	5	V_0	11	V_1	3	V_0	6	V_1	12	V_2	13	V_5
$\{V_0, V_1, V_2, V_3, V_4, V_5, V_6\}$	0	-	5	V_0	11	V_1	3	V_0	6	V_1	12	V_2	13	V_5

Die d-Spalten enthalten die Distanzwerte, in den p-Spalten ist der Vorgänger vermerkt.

Wie rekonstruieren wir jetzt den Verlauf des kürzesten Weges beispielsweise zu V_6? Wir hangeln uns von Vorgänger zu Vorgänger zurück. Vorgänger von V_6 ist V_5, Vorgänger von V_5 ist V_2, der wiederum V_1 als Vorgänger hat. V_1 führt uns schließlich zu unserem Ausgangsknoten V_0. Der Weg ist damit: $V_0 - V_1 - V_2 - V_5 - V_6$.

Aufgabe 27
Es sollen die kürzesten Wege in einem gerichteten Graphen $G = (V, E)$ mit Kantengewichten aus $\{1, \ldots, W\}$ und einem Startknoten s bestimmt werden. Beschreibe einen Algorithmus mit Laufzeit $O(W(|V| + |E|))$ für dieses Problem.

Aufgabe 28
$G = (V, E)$ sei ein gerichteter Graph ohne Kreise. **Beschreibe** einen *möglichst effizienten* Algorithmus, der die Länge des längsten Weges in G bestimmt. **Bestimme** die Laufzeit deines Algorithmus.

Hinweis: Das Problem des topologischen Sortierens wird sich als nützlich herausstellen.

Aufgabe 29
Ein gerichteter Graph $G = (V, E)$ ist in Adjazenzlisten-Darstellung gegeben. Ebenso ist ein Startknoten $s \in V$ gegeben. Einige Knoten sind schwarz gefärbt, die restlichen Knoten sind weiß gefärbt. Die Länge eines Weges wird als die *Anzahl der schwarzen Knoten des Weges* definiert.

Entwirf einen *möglichst effizienten* Algorithmus, der kürzeste Wege vom Startknoten s zu allen anderen Knoten des Graphen bestimmt. **Bestimme** die Laufzeit deines Algorithmus.

Aufgabe 30
Ein Unternehmen für Schwertransporte ist bestrebt, für seine Lastwagen Routen mit einer möglichst geringen maximalen Steigung zu finden, da sich die Ladekapazität nach der maximalen Steigung entlang eines Weges richtet. Wir stellen ein Straßennetz wieder als einen gerichteten Graphen $G = (V, E)$ dar. Die Straßen (Kanten) sind mit der maximalen Steigung dieser Straße beschriftet. Die Zentrale der Firma liege im Knoten $S \in V$.

Bestimme die Wege mit geringster maximaler Steigung von S zu allen anderen Knoten. Entwerfe hierzu einen möglichst effizienten Algorithmus.

3.3 Minimale Spannbäume

Definition 3.7 *Sei $G = (V, E)$ ein ungerichteter Graph. Ein Baum $T = (V', E')$ heißt ein Spannbaum für G, falls $V' = V$ und $E' \subseteq E$.*

Sei $G = (V, E)$ ein ungerichteter Graph und sei

$$\text{länge} : E \to \mathbb{R}$$

vorgegeben. Für einen Spannbaum $T = (V, E')$ für G setzen wir

$$\text{länge}(T) = \sum_{e \in E'} \text{länge}(e).$$

Offensichtlich besitzen nur zusammenhängende Graphen Spannbäume; wir werden deshalb in diesem Paragraphen stets annehmen, dass der zu Grunde liegende Graph G zusammenhängend ist. Unser Ziel ist die Berechnung eines **minimalen Spannbaums** T, also eines Spannbaums minimaler Länge unter allen Spannbäumen von G.

Beispiel 3.2 Als ein Anwendungsbeispiel betrachte das Problem des Aufstellens eines Kommunikationsnetzwerks zwischen einer Menge von Zentralen. Jede Zentrale ist im Stande, Nachrichten, die es von anderen Zentralen erhält, in Richtung Zielzentrale weiterzugeben. Für die Verbindungen zwischen den Zentralen müssen Kommunikationsleitungen gekauft werden, deren Preis von der Distanz der Zentralen abhängt. Es genügt, so viele Leitungen zu kaufen, dass je zwei Zentralen indirekt (über andere Zentralen) miteinander kommunizieren können. Wie sieht ein billigstes Kommunikationsnetzwerk aus?

In der graph-theoretischen Formulierung repräsentieren wir jede Zentrale durch einen eigenen Knoten. Zwischen je zwei Knoten setzen wir eine Kante ein und markieren die Kante mit der Distanz der entsprechenden Zentralen. Wenn G der resultierende Graph ist, dann entspricht ein billigstes Kommunikationsnetzwerk einem minimalen Spannbaum!

Beispiel 3.3 Im metrischen Traveling Salesman Problem ($M - TSP$) ist der vollständige ungerichtete Graph $V_n = \{1, \ldots, n\}, E_n)$ mit n Knoten gegeben: Für je zwei Knoten (oder Städte) $u, v \in \{1, \ldots, n\}$ besitzt V_n die Kante $\{u, v\}$. Zusätzlich ist eine Längenangabe

$$\text{länge} : \{1, \ldots, n\} \to \mathbb{R}_{\geq 0}$$

gegeben, wobei länge(u, v) die Distanz zwischen den Städten u und v angibt. Wir nehmen an, das `länge` einer Metrik[1] ist.

[1]Eine Funktion $d : X^2 \to \mathbb{R}_{\geq 0}$ heißt eine Metrik für die Menge X, wenn $d(x, y) = 0$ genau dann gilt, wenn $x = y$), wenn d symmetrisch ($d(x, y) = d(y, x)$) und transitiv ($d(x, y) + d(y, z) \leq d(x, z)$) ist.

In $M-TSP$ muss ein Handlungsreisender, von seinem Ausgangspunkt aus, alle Städte in einer Rundreise minimaler Länge besuchen. TSP ist ein schwieriges Optimierungsproblem, wie wir in Satz 6.6 zeigen werden. Wir entwerfen deshalb die „minimale Spannbaum Heuristik“, die zwar eine im Allgemeinen nicht optimale, aber stets eine gute Rundreise berechnen wird.

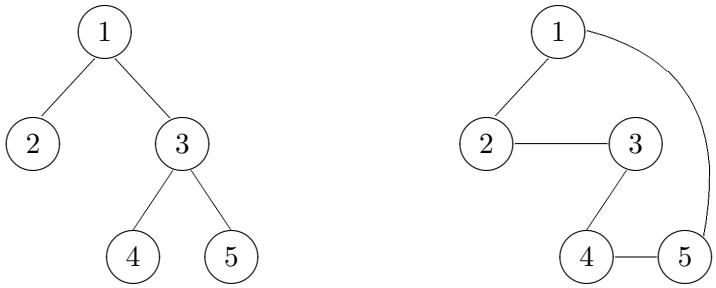
Wir nehmen an, dass minimale Spannbäume effizient berechnet werden können.

- Berechne einen minimalen Spannbaum B für $\{1,\ldots,n\}$ und die Metrik länge : $E_n \to \mathbb{R}_{\geq 0}$.

  Wenn wir eine Kante aus einer Rundreise entfernen, dann erhalten wir einen Spannbaum B'. Da B ein minimaler Spannbaum ist, folgt länge $(B) \leq$ länge(B') und länge(B) ist höchstens so groß wie die Länge `opt` einer kürzesten Rundreise.

- Durchlaufe den minimalen Spannbaum B in Präorder-Reihenfolge und benutze diese Reihenfolge als Tour.

  Wir behaupten, dass die erhaltene Rundreise genau doppelt so lang wie der minimale Spannbaum B ist. Warum? Wir betrachten den folgenden Baum mit seiner Präorder-Rundreise.



  Wie lang ist unsere Rundreise höchstens? Die Länge stimmt mit

$$\text{länge}(1,2) + (\text{länge}(2,1) + \text{länge}(1,3)) + \text{länge}(3,4) +$$
$$(\text{länge}(4,3) + \text{länge}(3,5)) + (\text{länge}(5,3) + \text{länge}(3,1))$$

  überein und wir beobachten, dass alle Kanten des Spannbaums genau zweimal auftreten. Unsere Behauptung stimmt somit für dieses Beispiel, aber unsere Argumentation kann auch für beliebige Bäume angewandt werden.

Aufgabe 31
Benutze die Dreiecksungleichung

$$||p_i - p_j|| \leq ||p_i - p_k|| + ||p_k - p_j||,$$

um zu zeigen, dass die aus einem minimalen Spannbaum erhaltene Rundreise stets höchstens doppelt so lang wie der minimale Spannbaum ist.

Wir haben also

$$\text{länge}(B) \leq \texttt{opt} \leq 2 \cdot \text{länge}(B)$$

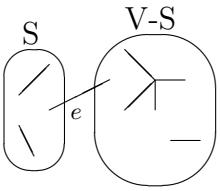
erhalten und unsere „Spannbaum-Rundreise ist höchstens doppelt so lang wie die optimale Rundreise.

Sei $G = (V, E)$ ein beliebiger ungerichteter zusammenhängender Graph. Das Konzept “kreuzender Kanten“ ist das zentrale Hilfsmittel für die Berechnung minimaler Spannbäume: Für eine Knotenmenge $S \subseteq V$ sagen wir, dass eine Kante $e = \{u, v\}$ die Menge *kreuzt*, wenn genau ein Endpunkt von e in S liegt.

Lemma 3.8 *Sei $G = (V, E)$ ein ungerichteter Graph und $W = (V, E')$ sei ein Wald mit $E' \subseteq E$. Der Wald W sei in einem minimalen Spannbaum T enthalten und keine seiner Kanten kreuze die Menge S.*

Wenn e eine Kante in E ist, die unter allen S-kreuzenden Kanten minimale Länge besitzt, dann ist auch $W' = (V, E' \cup \{e\})$ in einem minimalen Spannbaum enthalten.

Beweis



Angenommen, der minimale Spannbaum T enthält W, aber nicht W'. T besitzt also die Kante e nicht.
Wenn wir e zu T hinzufügen würden, schließen wir damit genau einen Kreis. Dieser Kreis besitzt neben e mindestens eine weitere kreuzende Kante e^*.

Wir entfernen e^*, setzen e ein und nennen den entstehenden Graphen T'. T' ist ein Baum, denn den einzigen durch die Hereinnahme von e geschaffenen Kreis haben wir durch die Herausnahme von e^* wieder gebrochen. Da nach Definition von e

$$\text{länge } (e) \leq \text{ länge}(e^*),$$

gilt, folgt länge$(T') \leq$ länge(T). Also ist T' ein minimaler Spannbaum, der W' enthält. □

Dieses Lemma zeigt uns, wie wir minimale Spannbäume konstruieren können: Finde sukzessive kürzeste kreuzende Kanten für geeignete Knotenmengen $S \subseteq V$. Als ein erstes Beispiel:

Algorithmus 3.2 **Prim**

(1) Setze $S = \{0\}$ und $E' = \emptyset$.

(2) Solange $S \neq V$, wiederhole:

(2a) Bestimme eine kürzeste S-kreuzende Kante $e = \{u, v\} \in E$ mit $u \in S$ and $v \in V \backslash S$. Es gilt also

$$\text{länge}(e) = \min\{\text{länge}(e') \mid e' \in E \text{ und } e \text{ kreuzt } S\}.$$

(2b) Setze $S = S \cup \{v\}$ und $E' = E' \cup \{e\}$.

Nach Lemma 3.8 baut Prims Algorithmus einen minimalen Spannbaum, denn der Algorithmus fügt jeweils eine kürzeste kreuzende Kante ein. Betrachten wir einen zweiten Algorithmus.

Algorithmus 3.3 **Kruskal**

(1) Sortiere die Kanten gemäß aufsteigendem Längenwert.
Sei $W = (V, E')$ der leere Wald, also $E' = \emptyset$.

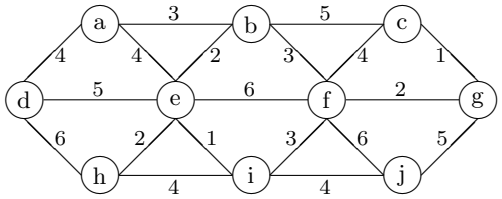
(2) Solange W kein Spannbaum ist, wiederhole

(2a) Nimm die gegenwärtige kürzeste Kante e und entferne sie aus der sortierten Folge.

(2b) Verwerfe e, wenn e einen Kreis in W schließt.

(2c) Ansonsten akzeptiere e und setze $E' = E' \cup \{e\}$.

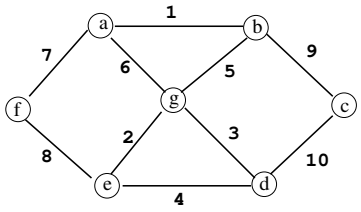
Aufgabe 32
Gegeben sei der folgende gewichtete, ungerichtete Graph G:



(a) Benutze den Algorithmus von Kruskal, um einen minimalen Spannbaum für G zu konstruieren. **Gib** die Reihenfolge der vom Algorithmus gefundenen Kanten **an**.

(b) Ist der minimale Spannbaum für obigen Graphen G eindeutig? **Begründe** deine Antwort!

Aufgabe 33
Gegeben sei der folgende gewichtete, ungerichtete Graph G:



Gib die *Reihenfolgen* der von Kruskals und von Prims Algorithmen gefundenen Kanten **an**. Wähle Knoten a als Startknoten für Prims Algorithmus.

Angenommen, Kruskals Algorithmus wählt zu einem bestimmten Zeitpunkt eine gegenwärtig kürzeste Kante e. Der Wald W bestehe aus den Bäumen $T_1, \ldots T_s$. Wenn e einen Kreis in W schließt, dann schließt e einen Kreis in einem Baum T_i und kann offensichtlich nicht benutzt werden. Ansonsten besitzt e einen Endpunkt in T_i und einen Endpunkt in T_j (für $i \neq j$). Für

$$S = \text{ die Menge aller Knoten von } T_i$$

ist e eine kürzeste kreuzende Kante: Auch Kruskals Algorithmus funktioniert!

Es bleibt das Problem, beide Algorithmen mit effizienten Datenstrukturen auszustatten, damit wir effiziente Implementierungen beider Algorithmen erhalten. Wir kümmern uns zuerst um Prims Algorithmus.

Angenommen, wir kennen für jeden Knoten $v \in V \setminus S$ die Länge $e(v)$ einer kürzesten kreuzenden Kante mit Endpunkt v. Was ist zu tun, wenn Knoten $w \in V \setminus S$ in die Menge S eingefügt wird? Nur die Nachbarn w' von w können eine neue kürzeste kreuzende Kante, nämlich die Kante $\{w, w'\}$, erhalten: Für jeden Nachbarn w' von w setze deshalb

$$e(w') = (e(w') > \text{ länge } (\{w, w'\})) \ ? \ \text{länge}(\{w, w'\}) \ : \ e(w');$$

wobei wir anfänglich

$$e(w) = \begin{cases} \text{länge}(\{0, w\}) & \{0, w\} \text{ ist eine Kante,} \\ \infty & \text{sonst} \end{cases}$$

setzen. Wie berechnen wir eine, unter allen kreuzenden Kanten, kürzeste Kante? Wir verwalten die Kantenlängen $e(u)$, für $u \in V \setminus S$, in einem Heap und bestimmen die kürzeste kreuzende Kante durch eine deletemin-Operation!

Betrachten wir die Laufzeit dieser Implementierung. Höchstens $|E|$-mal sind die $e(u)$-Werte anzupassen und höchstens $(|V| - 1)$ mal ist eine deletemin-Operation anzuwenden. Damit haben wir höchstens $|V| + |E|$ Heap-Operationen auszuführen, die jeweils in Zeit

$$O(\log_2(|V| + |E|)) = O(\log_2 |V|)$$

laufen. Da $|V| \leq |E| + 1$ für zusammenhängende Graphen gilt, folgt:

Satz 3.9 *Sei $G = (V, E)$ ein ungerichteter zusammenhängender Graph. Dann berechnet Prims Algorithmus einen minimalen Spannbaum in Zeit*

$$O(|E| \cdot \log_2 |V|).$$

Betrachten wir als letztes Kruskals Algorithmus. Die wesentlichen Operationen sind hier:

- Die Sortierung aller Kanten in Schritt 1: Zeit $O(|E| \log_2 |E|))$ wird benötigt.
- Die $|E|$-malige Wiederholung der Schleife in Schritt 2.
  - Die Bestimmung der gegenwärtig kürzesten Kante $e = \{u, v\}$ gelingt in konstanter Zeit, da wir zu Anfang die Kanten sortiert haben.
  - Wir müssen in Schritt (2b) feststellen, ob e einen Kreis schließt. Dazu genügt die Überprüfung, ob u and v zum selben Baum gehören.

    Wir erfinden dazu die union-find Datenstruktur, die die Operationen `union`(i, j) und `find`(u) unterstützt: `find`(u) bestimmt die Wurzel des Baums, der u als Knoten besitzt. `union`(i, j) vereinigt die Knotenmenge des Baums mit Wurzel i mit der Knotenmenge des Baums mit Wurzel j.

    Die Kante $e = \{u, v\}$ schließt genau dann keinen Kreis, wenn `find`$(u) \neq$ `find`(v). Wir wenden in diesem Fall die union-Operation an, um die Vereinigung der Knotenmenge des Baums von u mit der Knotenmenge des Baums von v zu berechnen.

Also genügt es, die Operationen `find`(u) und `union`(i, j) durch eine geeignete Datenstruktur schnell zu unterstützen.

Wir müssen die Knotenmenge V_B eines jeden Baums $B = (V_B, E_B)$ im Wald W darstellen. Das könnten wir einfach durch die Darstellung des Baums B erreichen, aber dann werden die union- und find-Operationen zu langsam. Unsere Grundidee ist stattdessen die Implementierung

- der union-Operation durch ein Anhängen der Wurzel des einen Baums unter die Wurzel des anderen Baums und
- die Implementierung der find-Operation durch ein Hochklettern im Baum.

Wir benötigen also zwei Datenstrukturen. Die erste Darstellung gibt alle tatsächlich vorhandenen Kanten wieder und dies kann mit der Adjazenzlisten-Darstellung erreicht werden. Die zweite Datenstruktur ist für die Unterstützung der union- und find Operationen zuständig.

Hier stellen wir nur die Knotenmengen der einzelnen Bäume dar und wählen hier die denkbar einfächste Repräsentation, nämlich die Vater-Darstellung. Da wir mit einem leeren Wald beginnen, setzen wir anfänglich

$$\text{Vater}[u] = u \quad \text{für} \quad u = 1, \ldots, |V|$$

und interpretieren generell Vater$[u] = u$ als die Beschreibung einer Wurzel. Einen find-Schritt führen wir wie bereits angekündigt aus, indem wir den Baum mit Hilfe des Vater-Arrays hochklettern bis die Wurzel gefunden ist. Damit benötigt ein find-Schritt Zeit höchstens proportional zur Tiefe des jeweiligen Baumes. Wie garantieren wir, dass die Bäume nicht zu tief werden? Indem wir in einem Union-Schritt stets die Wurzel des kleineren Baumes unter die Wurzel des größeren Baumes anhängen.

Betrachten wir einen beliebigen Knoten v. Seine Tiefe vergrößert sich nur dann um 1, wenn v dem kleineren Baum angehört. Das aber heißt,

> die Tiefe von v vergrößert sich nur dann um 1, wenn sein Baum sich in der Größe mindestens verdoppelt.

Also: Die Tiefe aller Bäume ist durch $\log_2(|V|)$ beschränkt. Damit läuft ein union-Schritt in konstanter Zeit, während ein find-Schritt höchstens logarithmische Zeit benötigt.

Satz 3.10 *Sei $G = (V, E)$ ein ungerichteter Graph. Dann berechnet Kruskals Algorithmus einen minimalen Spannbaum in Zeit*

$$O(|E| \cdot \log_2 |V|).$$

Aufgabe 34
$G = (V, E)$ sei ein ungerichteter Graph, dessen Kanten gewichtet sind. Sei $e \in E$ eine Kante, die in (irgendeinem) minimalen Spannbaum T enthalten ist. **Zeige**, dass es eine Knotenmenge $S \subseteq V$ gibt, so dass e eine kürzeste kreuzende Kante ist (das heißt, e hat minimales Gewicht unter allen Kanten mit einem Endpunkt in S und einem Endpunkt in $V - S$).

Aufgabe 35
Gegeben ist ein zusammenhängender ungerichteter Graph $G = (V, E)$, sowie eine Gewichtung seiner Kanten. Ebenso ist ein minimaler Spannbaum T von G gegeben. G wie auch T werden durch ihre Adjazenzlisten repräsentiert.

Angenommen die Kosten *einer* Kante e von G werden verändert. (Wir erhalten diese Kante e und das neue Gewicht der Kante zusätzlich als Eingabe.)
Beschreibe einen möglichst effizienten Algorithmus, der entweder einen neuen minimalen Spannbaum findet oder entscheidet, dass T noch immer ein minimaler Spannbaum ist. **Bestimme** die Laufzeit deines Algorithmus.

Hinweis: Unterscheide vier Fälle: Kante e gehört zu T oder nicht, die Kosten von e sind angestiegen oder gefallen.

Aufgabe 36
Sei $G = (V, E)$ ein ungerichteter, gewichteter Graph. **Beweise** oder **widerlege** die folgende Aussage: Wenn alle Gewichte in G verschieden sind, dann gibt es genau einen minimalen Spannbaum für G.

3.4 Zusammenfassung

Graphen sollten im allgemeinen durch Adjazenzlisten repräsentiert werden. Adjazenzlisten sind speicher-effizient und unterstützen die wichtige Operation

bestimme alle Nachbarn, bzw. alle direkten Nachfolger

schnellstmöglich. Die Durchsuchungsmethoden Tiefensuche und Breitensuche liefern schnelle Algorithmen für viele graphentheoretische Probleme wie

- Finden der Zusammenhangskomponenten,
- Existenz von Kreisen (Tiefensuche),
- Bestimmung kürzester (ungewichteter) Wege (Breitensuche).

Die Bestimmung kürzester (nicht-negativ gewichteter) Wege war das erste zentrale Thema dieses Kapitels, und wir haben mit dem Algorithmus von Dijkstra eine schnelle Lösung gefunden. Im wesentlichen ersetzt Dijkstras Algorithmus die Schlange in der Breitensuche durch eine Prioritätswarteschlange.

Mit der Bestimmung minimaler Spannbäume haben wir zum ersten Mal die Methode der **Greedy-Algorithmen** kennengelernt: Kruskals Algorithmus legt sich „gierig“ auf die jeweils kürzeste Kante fest, die keinen Kreis schließt.

Dijkstras, Kruskals wie auch Prims Algorithmus benötigen den Einsatz schnellster Datenstrukturen, um die algorithmische Idee effizient zu implementieren. Im Fall von Dijkstras Algorithmus, wie auch für den Algorithmus von Prim, haben wir einen Heap eingesetzt. Für Kruskals Algorithmus wurde eine Datenstruktur benötigt, die union- und find-Operationen effizient unterstützt.

Im nächsten Kapitel wenden wir die Methode der dynamischen Programmierung an, um alle kürzesten Wege (Floyds Algorithmus) zu bestimmen, selbst wenn einige Kanten eine negative Länge besitzen, aber keine Kreise negativer Länge existieren. Wir werden auch weitere Beispiele von Greedy-Algorithmen kennen lernen.

Kapitel 4

Entwurfsmethoden

Wir beginnen mit einer Beschreibung von Greedy-Algorithmen und wenden uns dann den Entwurfsmethoden Divide & Conquer und Dynamischer Programmierung zu. Das Kapitel schließt mit einer kurzen Betrachtung der linearen Programmierung, der mächtigsten effizienten Methode in der exakten Lösung von Optimierungsproblemen.

4.1 Greedy-Algorithmen

Greedy-Algorithmen (gierige Algorithmen) werden meist für die exakte oder approximative Lösung von Optimierungsproblemen verwandt, wobei man exakte Lösungen nur für einfachste Optimierungsprobleme erhält. Typischerweise konstruiert ein Greedy-Algorithmus einen besten Lösungsvektor Komponente nach Komponente, wobei der Wert einer gesetzten Komponente **nie** zurückgenommen wird. Die zu setzende Komponente und ihr Wert wird nach einfachen Kriterien bestimmt.

Diese schwammige Erklärung kann formalisiert werden. So wurden Klassen von Optimierungsproblemen (nämlich Matroide und Greedoide) definiert, die Greedy-Algorithmen besitzen. Die entsprechende Formalisierung können wir in dieser Vorlesung aber nicht besprechen.

Beispiele von Greedy-Algorithmen sind die Algorithmen von Prim und Kruskal; aber auch Dijkstras Algorithmus kann als ein Greedy-Algorithmus angesehen werden. Allen drei Algorithmen ist gemeinsam, dass der Lösungsvektor Komponente nach Komponente konstruiert wird, ohne dass der Wert einer gesetzten Komponente je modifiziert wird.

Wir betrachten hier weitere Anwendungen von Greedy-Algorithmen, nämlich Anwendungen in Scheduling Problemen sowie die Konstruktion von *Huffmann-Codes* für die Datenkomprimierung.

4.1.1 Scheduling

Scheduling mit minimaler Verspätung auf einem Prozessor

n Aufgaben $1, \ldots, n$ sind gegeben, wobei Aufgabe i die Frist f_i und die Laufzeit l_i besitzt. Alle Aufgaben sind auf einem einzigen Prozessor auszuführen, wobei die maximale Verspätung zu minimieren ist: Wenn Aufgabe i zum Zeitpunkt t_i fertig ist, dann ist $v_i = t_i - f_i$ ihre Verspätung. Unser Ziel ist also eine Ausführungssequenz, so dass die maximale Verspätung

$$\max_{1 \leq i \leq n} v_i$$

kleinstmöglich ist.

Wir haben verschiedene Möglichkeiten wie etwa das Sortieren der Aufgaben nach ihrem „Slack“ $f_i - l_i$ und der Ausführung von Aufgaben mit kleinstem Slack zuerst. Eine solche Idee ist zu naiv, da wir Fristen ignorieren: Bei zwei Aufgaben mit $f_1 = 3, l_1 = 1$ und $f_2 = T + 1, l_2 = T$ wird zuerst die zweite Aufgabe ausgeführt und beansprucht das Intervall $[0, T]$. Die erste Aufgabe beansprucht dann das Intervall $[T + 1, T + 2]$ und erleidet damit die Verspätung $v_1 = T + 2 - 3 = T - 1$. Für $T \geq 3$ ist aber eine Ausführung der Aufgaben in ursprünglicher Reihenfolge sehr viel besser, da dann Aufgabe 1 im Intervall $[0, 1]$ und Aufgabe 2 im Intervall $[2, T + 2]$ mit den jeweiligen Verspätungen $v_1 = 1 - 3 = -2$ und $v_2 = T + 2 - (T + 1) = 1$ ausgeführt werden.

Im zweiten Versuch sortieren wir Aufgaben nach aufsteigender Frist und führen Aufgaben nach dem Motto „Frühe Fristen zuerst“ aus. Auch das sollte nicht funktionieren, weil wir jetzt Laufzeiten ignorieren, aber versuchen wir es trotzdem.

Algorithmus 4.1 **Frühe Fristen zuerst**

(1) Sortiere alle Aufgaben gemäß aufsteigender Frist.

(2) Solange noch Aufgaben vorhanden sind, wiederhole

führe die Aufgabe i mit kleinster Frist aus und entferne diese Aufgabe.

Satz 4.1 *Das Prinzip „Frühe Fristen zuerst“ funktioniert: Eine Ausführungsfolge mit minimaler Verspätung wird für n Aufgaben in Zeit $O(n \cdot \log_2 n)$ berechnet.*

Beweis: Wir fixieren eine Ausführungssequenz mit minimaler größter Verspätung. Wir sagen, dass die Ausführungssequenz eine Inversion (i, j) besitzt, wenn i nach j ausgeführt wird, aber $f_i < f_j$ gilt. Die Anzahl der Inversionen misst also den „Abstand“ zwischen der optimalen und unserer Ausführungssequenz. Wie zeigen jetzt, dass wir eine Inversion beseitigen können, ohne die größte Verspätung zu erhöhen. Da eine jede Ausführungssequenz höchstens $\binom{n}{2}$ Inversionen besitzt, erhalten wir nach genügend häufiger „Inversionsreduktion“ eine Ausführung ohne Inversionen. Sind wir dann fertig? Ja, denn die größte Verspätung wird nie erhöht und:

Beobachtung 1: Alle Ausführungen ohne Inversionen besitzen dieselbe maximale Verspätung.

Beweis: Zwei Ausführungen ohne Inversionen unterscheiden sich nur in der Reihenfolge, mit der Aufgaben mit selber Frist f ausgeführt werden. Die größte Verspätung wird für die jeweils zuletzt ausgeführte Aufgabe mit Frist f erreicht, aber die größten Verspätungen sind identisch, denn die jeweils zuletzt ausgeführten Aufgaben werden zum selben Zeitpunkt fertig! □

Wir möchten die vorgegebene optimale Ausführungssequenz nur wenig ändern und suchen deshalb nach Aufgaben i und j mit $f_i < f_j$, wobei aber Aufgabe i *direkt* nach Aufgabe j ausgeführt werde.

Beobachtung 2: Wenn eine Ausführungssequenz mindestens eine Inversion hat, dann gibt es direkt aufeinanderfolgende Aufgaben, die eine Inversion bilden.

Beweis: Das ist offensichtlich, denn wenn je zwei aufeinanderfolgende Aufgaben inversionsfrei sind, dann hat die Ausführungssequenz keine Inversion. □

Wir können jetzt einen Reduktionsschritt ausführen.

Beobachtung 3: In einer beliebigen Ausführungssequenz werde Aufgabe i direkt nach Aufgabe j ausgeführt und es gelte $f_i < f_j$. Wenn wir die beiden Aufgaben vertauschen, dann steigt die größte Verspätung nicht an.

Beweis: Aufgabe i wird jetzt früher ausgeführt und seine Verspätung wird abnehmen. Das Problem ist die später ausgeführte Aufgabe j, die jetzt zu dem (späteren) Zeitpunkt T fertiggestellt wird.

Aber, und das ist der kritische Punkt, wann wurde denn Aufgabe i in der ursprünglichen Ausführungsfolge fertiggestellt? Zum selben Zeitpunkt T und die neue Verspätung $v_j = T - f_j$ von Aufgabe j ist kleiner als die alte Verspätung $v_i = T - f_i$ von Aufgabe i, denn $f_i < f_j$. □

Wir wenden Beobachtung 3 wiederholt an und erhalten eine inversionsfreie Ausführungsfolge mit höchstens so großer Verspätung. Jede Ausführungssequenz ohne Inversionen hat aber dieselbe größte Verspätung und unser Motto „Früheste Frist zuerst“ funktioniert tatsächlich. □

Intervall Scheduling auf einem Prozessor

Aufgaben $1, \ldots, n$ sind auf einem einzigen Prozessor auszuführen. Aufgabe i besitzt eine Startzeit s_i, eine Terminierungszeit t_i und beschreibt somit das Zeitintervall $[s_i, t_i]$. Wir sagen, dass zwei Aufgaben *kollidieren*, wenn ihre Zeitintervalle einen gemeinsamen Punkt haben. Das Ziel:

> Führe eine größtmögliche Anzahl nicht-kollidierender Aufgabe aus.

Eine beispielhafte Anwendung des Intervall Scheduling ist die Durchführung möglichst vieler Veranstaltungen in einem Veranstaltungssaal.

In einem ersten Versuch könnten wir Aufgaben nach aufsteigender Startzeit ausführen, aber dies ist keine gute Idee: Wenn eine frühe Aufgabe eine späte Terminierungszeit hat, werden viele andere Aufgaben ausgeschlossen. Ein zweiter Versuch wäre die Ausführung von Aufgaben gemäß ihrer Länge, aber eine kurze Aufgabe kann zwei kollisionsfreie lange Aufgaben blockieren.

Wir probieren einen dritten Ansatz. Angenommen, eine optimale Ausführungsfolge führt Aufgabe i als erste Aufgabe aus, obwohl $t_j < t_i$ gilt. In diesem Fall erhalten wir eine mindestens ebenso gute Ausführungsfolge, wenn wir Aufgabe j als erste Aufgabe ausführen, denn jede von Aufgabe j ausgeschlossene Aufgabe wird auch von Aufgabe i ausgeschlossen. Also können wir die Aufgaben i und j ohne Bestrafung vertauschen und es gibt somit eine optimale Ausführungssequenz, in der eine Aufgabe mit kleinster Terminierungszeit als erste Aufgabe ausgeführt wird.

Welche Aufgabe kann als zweite Aufgabe ausgeführt werden? Wir wenden wieder das obige Argument an und führen eine mit der ersten Aufgabe nicht-kollidierende Aufgabe aus, die unter allen nicht-kollidierenden Aufgaben die kleinste Terminierungszeit besitzt!

Algorithmus 4.2 **Frühe Terminierungszeiten zuerst**

(1) Sortiere alle Aufgaben gemäß aufsteigender Terminierungszeit.

(2) Solange noch Aufgaben vorhanden sind, wiederhole

führe die Aufgabe i mit kleinster Terminierungszeit aus, entferne diese Aufgabe und alle Aufgaben j mit $s_j \leq t_i$.
/* Alle kolliderenden Aufgaben werden entfernt. */

Satz 4.2 *Das Prinzip „Frühe Terminierungszeiten zuerst“ funktioniert: Eine größtmögliche Anzahl kollisionsfreier Aufgaben wird für n Aufgaben in Zeit $O(n \log_2 n)$ berechnet.*

Beweis: Die Laufzeit wird durch das Sortieren im Schritt (1) dominiert, da jede Aufgabe in Schritt (2) nur „einmal angefasst“ wird: Entweder wird die Aufgabe ausgeführt oder entfernt. □

Intervall Scheduling mit einer kleinstmöglichen Prozessorzahl

Wir nehmen wie im Intervall Scheduling für einen Prozessor an, dass n Aufgaben mit ihren Zeitintervallen $[s_i, t_i]$ gegeben sind. Unser Ziel:

Führe alle Aufgaben auf einer kleinstmöglichen Zahl von Prozessoren aus.

Wieviele Prozessoren benötigen wir mindestens? Für jeden Zeitpunkt t bestimmen wir die Anzahl d_t der Aufgaben, die Zeitpunkt t mit ihrem Intervall überdecken. Wenn

$$d = \max_t d_t,$$

dann gibt es einen Zeitpunkt T, der von d Intervallen überdeckt wird, und die diesen Intervallen entsprechenden Aufgaben benötigen somit (mindestens) d Prozessoren. Überraschenderweise genügen aber auch d Prozessoren, um *alle* Aufgaben auszuführen!

Algorithmus 4.3 **Frühe Startzeiten zuerst**

(1) Sortiere alle Aufgaben gemäß aufsteigender Startzeit.

(2) Solange noch Aufgaben vorhanden sind, wiederhole

Sei Aufgabe i die Aufgabe mit kleinster Startzeit. Entferne diese Aufgabe.
Führe i auf einem beliebigen „freien“ Prozessor aus. (Ein Prozessor ist frei, wenn er keine mit i kollidierende Aufgabe ausführt.)
/* d ist die größte Anzahl von Aufgaben, die alle einen gemeinsamen Zeitpunkt T überdecken. Also ist die Anzahl der Aufgaben, die die Startzeit s_i von Aufgabe i überdecken, durch d beschränkt: Höchstens $d-1$ Prozessoren führen „frühere“ Aufgaben aus, die mit Aufgabe i kollidieren und zu keinem Zeitpunkt werden mehr als Prozessoren benötigt. */

Satz 4.3 *Der Algorithmus führt alle Aufgaben mit der kleinstmöglichen Anzahl von Prozessoren aus. Für n Aufgaben genügt Laufzeit $O(n \cdot \log_2 n)$.*

Beweis: d Prozessoren werden benötigt, aber sie genügen auch und unser Algorithmus arbeitet deshalb optimal. □

Aufgabe 37
Implementiere den Greedy-Algorithmus, so dass die Laufzeit für n Aufgaben durch höchstens $O(n \cdot \log_2 n)$ beschränkt ist.

4.1.2 Huffman-Codes

Gegeben sei ein Text T über einem Alphabet Σ. Für jeden Buchstaben $a \in \Sigma$ sei

$$H(a)$$

die Häufigkeit mit der Buchstabe a im Text T vorkommt. Wir nehmen an, dass $H(a) > 0$ für alle Buchstaben $a \in \Sigma$ gilt und dass Σ aus mindestens zwei Buchstaben besteht. Unser Ziel ist die Konstruktion eines binären Präfix-Codes für T, so dass der kodierte Text T minimale Länge besitzt.

Was ist ein binärer Präfix-Code? In einem binären Code weisen wir jedem Buchstaben $a \in \Sigma$ einen Bitstring code $(a) \in \{0,1\}^*$ zu. Diese Kodierung ist ein Präfix-Code, wenn eine Dekodierung durch einfaches Lesen des Codes von links nach rechts gelingt. Genauer,

Definition 4.4 *(a) Die Funktion* $\texttt{code} : \Sigma \to \{0,1\}^*$ *ist ein Präfix-Code, wenn kein Codewort* $\texttt{code}(a)$ *ein Präfix eines anderen Codewortes* $\texttt{code}(b)$ *ist.*

(b) Sei $T = a_1 \dots a_n$ *ein Text mit Buchstaben* $a_1, \dots, a_n \in \Sigma$*. Dann definieren wir*

$$\texttt{code}(T) = \texttt{code}(a_1) \cdots \texttt{code}(a_n)$$

und erhalten eine Kodierung der Länge

$$\sum_{a \in \Sigma} H(a) \cdot |\texttt{code}(a)|.$$

(c) Ein Huffman-Code für T *ist ein Präfix-Code minimaler Länge.*

Es ist $\texttt{code}(T) = \texttt{code}(a_1) \cdots \texttt{code}(a_n)$, und wir wissen, dass kein Codewort $\texttt{code}(a)$ ein Präfix eines anderen Codeworts $\texttt{code}(b)$ ist: Die Dekodierung von $\texttt{code}(T)$ „sollte“ problemlos gelingen. Wie sollte ein Huffman-Code für T aussehen? Häufige Buchstaben sollten kurze, seltene Buchstaben lange Codewörter erhalten.

Für den Entwurf eines optimalen Algorithmus sind die beiden folgenden Beobachtungen zentral. Zuerst stellen wir fest, dass Huffman-Codes durch binäre Bäume gegeben sind, deren Blätter den Buchstaben des Alphabets Σ entsprechen. Die Buchstabenkodierungen erhalten wir als die Markierung des Weges von der Wurzel zu einem Blatt.

Lemma 4.5 *Ein Huffman-Code kann stets durch einen binären Baum* B *repräsentiert werden, so dass*

(a) alle inneren Knoten genau 2 Kinder besitzen,

(b) jede Kante mit einem Bit $b \in \{0,1\}$ *markiert ist und*

(c) jedem Buchstaben $a \in \Sigma$ *genau ein Blatt* b_a *zugewiesen ist, so dass die Markierung des Weges von der Wurzel zum Blatt* b_a *mit* $\texttt{code}(a)$ *übereinstimmt.*

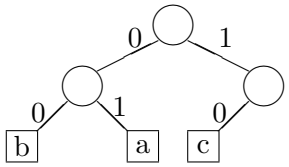
Beweis : Ein Huffman-Code sei durch die Funktion $\texttt{code} : \Sigma \to \{0,1\}^*$ gegeben. L sei die Länge des längsten Codewortes.

Wir konstruieren jetzt einen Binärbaum mit den obigen Eigenschaften. Zuerst betrachten wir den vollständigen binären Baum B^* der Tiefe L. In B^* sei jede Kante zu einem linken Kind mit 0 und jede Kante zu einem rechten Kind mit 1 markiert.

Sei V die Menge aller Knoten von B^*, so dass die Markierung des Weges von der Wurzel zu v mit einem Codewort übereinstimmt. Da ein Huffman-Code insbesondere ein Präfix-Code ist, kann es **keine** zwei Knoten $u, v \in V$ geben, so dass u ein Vorfahre von v ist. Der Binärbaum B entsteht aus B^*, in dem wir alle die Knoten w aus B^* entfernen, die einen Knoten aus V als Vorfahren besitzen.

Nach Konstruktion erfüllt B die Eigenschaften (b) und (c). Angenommen, B besitzt einen Knoten v mit nur einem Kind. Dann ist der Code aber unnötig lang: Wir erhalten einen kürzeren Code, indem wir v entfernen und den Vater von v mit dem Kind von v direkt verbinden.

Beispiel 4.1 Der Code $a = 01, b = 00, c = 10$ besitzt den Binärbaum



Der Binärbaum besitzt einen Knoten mit genau einem Kind, aber der Code ist kein Huffman-Code, denn der Code $a = 01, b = 00, c = 1$ ist kürzer. Der dem neuen Code entsprechende Baum besitzt alle in Lemma 4.5 geforderten Eigenschaften.

Wie bereits angedeutet sollten Huffman-Codes lange Codewörter für seltene Buchstaben benutzen. Genau diese Intuition können wir jetzt formalisieren.

Lemma 4.6 *Sei der Text T vorgegeben. Dann gibt es einen Huffman-Code für T, repräsentiert durch den Binärbaum B, mit der folgenden Eigenschaft:*

> *B besitzt einen inneren Knoten v maximaler Tiefe und v hat zwei Buchstaben geringster Häufigkeit als Blätter.*

Beweis Sei ein beliebiger Huffman-Code, repräsentiert durch den Binärbaum B^*, vorgegeben. L sei die Länge des längsten Codewortes.

Gemäß Lemma 4.5 (a) besitzt B^* mindestens zwei Codewörter der Länge L. Andererseits besitzen nur Buchstaben geringster Häufigkeit die längsten Codewörter. Die beiden Buchstaben $\alpha, \beta \in A$ geringster Häufigkeit werden somit Codewörter der Länge L in B^* besitzen.

Wir arrangieren die Blattmarkierungen für Blätter der Tiefe L so um, dass α und β benachbarten Blättern zugewiesen werden. Der neue Baum B besitzt die gleiche Kodierungslänge, da immer noch mit Worten der Länge L kodiert wird und entspricht deshalb auch einem Huffmann-Code. B erfüllt aber die Aussage des Lemmas. □

Also sollte unser Algorithmus die beiden Buchstaben geringster Häufigkeit aufsuchen. Seien dies $\alpha, \beta \in A$. Was dann? Wir müssen

$$\sum_{a \in \Sigma} H(a) \cdot |\mathtt{code}(a)| = (H(\alpha) + H(\beta)) \cdot |\mathtt{code}(\alpha)| + \sum_{a \in \Sigma, \alpha \neq a \neq \beta} H(a) \cdot |\mathtt{code}(a)|$$

minimieren. Hier ist die zentrale Überlegung:

Ersetze jedes Auftreten von α oder β im Text T durch einen neuen Buchstaben γ und setze

$$H(\gamma) = H(\alpha) + H(\beta).$$

Wenn wir das kleinere Kodierungsproblem für das neue Alphabet $\Sigma' = \Sigma \setminus \{\alpha, \beta\} \cup \{\gamma\}$ mit einem „Huffman-Baum“ B gelöst haben, dann erhalten wir einen Huffman-Baum für Σ, wenn wir an das (ursprüngliche) Blatt für γ in B zwei Kinder anheften: Die beiden Kinder kodieren dann α und β.

Algorithmus 4.4 **Berechnung von Huffman-Codes**

(1) Initialisiere einen Heap. Füge alle Buchstaben $a \in \Sigma$ in den Heap ein. Der Heap sei gemäß kleinstem Häufigkeitswert geordnet.

Der Algorithmus wird einen Huffman-Code durch einen Binärbaum B repräsentieren. Zu Anfang ist B ein Wald von Knoten v_a, wobei v_a dem Buchstaben a zugeordnet ist.

(2) Solange der Heap mindestens zwei Buchstaben enthält, wiederhole:

(2a) Entferne die beiden Buchstaben α und β geringster Häufigkeit.

(2b) Füge einen neuen Buchstaben γ in den Heap ein, wobei $H(\gamma) = H(\alpha) + H(\beta)$ gesetzt wird.

(2c) Schaffe einen neuen Knoten v_γ für γ und mache v_α und v_β Kinder von v_γ. Markiere die Kante $\{v_\alpha, v_\gamma\}$ mit 0 und die Kante $\{v_\beta, v_\gamma\}$ mit 1.

Satz 4.7 *Der obige Algorithmus findet einen Huffman-Code für n Buchstaben. Die Laufzeit ist durch $O(n \log_2 n)$ beschränkt.*

Beweis: Es werden höchstens $3n$ Heap-Operationen durchgeführt, da nach Entfernen von zwei Buchstaben und Einfügen eines neuen Buchstabens der Heap insgesamt einen Buchstaben verloren hat. □

Der obige Algorithmus bestimmt den binären Baum eines Huffman-Codes Kante nach Kante und wir bezeichnen ihn deshalb als Greedy-Algorithmus.

Aufgabe 38

Im Problem des Münzwechsels sind k Münztypen der Werte $a_1, a_2, \ldots, a_k$ gegeben, wobei die Münzwerte natürliche Zahlen sind, mit $1 = a_1 < a_2 < \ldots < a_k$. Für einen Betrag A ist eine *kleinstmögliche* Anzahl von Münzen zu bestimmen, deren Gesamtwert A ergibt.

(a) Entwirf einen *möglichst effizienten Greedy-Algorithmus*, der das Münzwechsel-Problem für den Münztyp $(a_1, \ldots, a_k)$ mit $a_i = 3^{i-1}$ löst. **Bestimme** die Laufzeit deines Algorithmus.

(b) Bestimme einen Münztyp $(a_1, a_2, \ldots, a_k)$, mit $1 = a_1 < a_2 < \ldots < a_k$, und einen Betrag A, so dass dein Greedy-Algorithmus *keine* kleinstmögliche Anzahl von Münzen ausgibt.

(c) Entwirf einen *möglichst effizienten* Algorithmus, der das Münzwechsel-Problem für *alle* Münztypen (mit $a_1 = 1$) löst. Benutze dabei die Entwurfsmethode des dynamischen Programmierens. **Bestimme** die Laufzeit deines Algorithmus.

Aufgabe 39

Es sollen Vorlesungen im einzigen verbliebenem Hörsaal einer Universität abgehalten werden. n Vorlesungen sind vorgeschlagen, dabei gibt es für jede eine Anfangszeit a_i und eine Schlußzeit s_i. Gesucht ist eine maximale Menge von Vorlesungen, die ohne zeitliche Überschneidungen abgehalten werden können. (Wenn die Menge von Vorlesungen also Vorlesungen i, j mit $i \neq j$ enthält, gilt entweder $a_i \geq s_j$ oder $a_j \geq s_i$.)

Beschreibe einen Greedy-Algorithmus, der das Problem *möglichst effizient* löst und **zeige**, dass die Ausgabe eine maximale Menge von Vorlesungen ist.

4.2 Divide & Conquer

In Divide & Conquer Algorithmen wird das gegebene Problem in kleinere Teilprobleme zerlegt, deren Lösung eine Lösung des Ausgangsproblems bedingt. Wir haben schon einige Divide & Conquer Algorithmen kennengelernt wie:

- Binärsuche,
- Durchsuchungsmethoden für Bäume und Graphen sowie
- Quicksort und Mergesort.

Wir betrachten hier weitere Anwendungen, nämlich schnelle Algorithmen zum Multiplizieren von zwei n-Bit Zahlen und zum Multiplizieren zweier Matrizen.

4.2.1 Eine schnelle Multiplikation natürlicher Zahlen

Seien $x = \sum_{i=0}^{n-1} a_i 2^i$ und $y = \sum_{i=0}^{n-1} b_i 2^i$ zwei vorgegebene n-Bit Zahlen. Gemäß der Schulmethode würden wir die Zahlen

$$x \cdot 2^i$$

für jedes i mit $b_i = 1$ addieren. Damit müssen bis zu n Zahlen mit bis zu $2n-1$ Bits addiert werden. Die Schulmethode benötigt also

$$\Theta(n^2) \text{ Bit-Operationen.}$$

Nun zum Entwurf von Divide & Conquer Algorithmen. Wir nehmen der Einfachheithalber an, dass n eine Zweierpotenz ist (Wenn notwendig, fülle mit führenden Nullen auf, um die Bitanzahl auf die nächste Zweierpotenz zu heben). Dann können wir schreiben

$$x = x_1 2^{n/2} + x_2, \quad y = y_1 2^{n/2} + y_2$$

wobei x_1, x_2, y_1 und y_2 Zahlen mit jeweils $n/2$ Bits sind. Ein trivialer Divide & Conquer Algorithmus berechnet zuerst die vier Produkte

$$x_1 \cdot y_1, x_1 \cdot y_2, x_2 \cdot y_1 \text{ und } x_2 \cdot y_2$$

und bestimmt $x \cdot y$ durch Summation der Produkte nach Shifts

$$x \cdot y = x_1 \cdot y_1 + 2^{n/2} \cdot (x_1 \cdot y_2 + x_2 \cdot y_1) + 2^n x_2 \cdot y_2.$$

Die Rekursionsgleichung für die Anzahl $M_0(n)$ der benötigten Bit-Operationen ist dann

$$M_0(n) = 4 \cdot M_0(\frac{n}{2}) + c \cdot n.$$

Der additive Term $c \cdot n$ mißt die Anzahl der nicht-rekursiven Bit-Operationen (Shifts und Additionen). Haben wir einen schnelleren Multiplikationsalgorithmus erhalten? Keineswegs, unsere allgemeine Formel für Rekursionsgleichungen liefert das Ergebnis

$$M_0(n) = \Theta(n^2).$$

Die wesentliche Beobachtung ist, dass tatsächlich drei Multiplikationen ausreichend sind:

- Berechne $z = (x_1 + x_2)(y_1 + y_2)$ sowie
- $z_1 = x_1 \cdot y_1$ und $z_2 = x_2 \cdot y_2$.
- Dann ist $x \cdot y = z_1 \cdot 2^n + (z - z_1 - z_2)2^{n/2} + z_2$.

Damit genügen 3 Multiplikationen und wir erhalten die Rekursionsgleichung

$$M_1(n) = 3M_1(n/2) + d \cdot n.$$

In dem Term $d \cdot n$ werden wiederum die Anzahl der nicht-rekursiven Bit-Operationen, also Shifts und Additionen aufgenommen. Die asymptotische Lösung ist

$$M_1(n) = \Theta(n^{\log_2 3}) = \Theta(n^{1,59\ldots}).$$

Wir haben also die Nase vorn im asymptotischen Sinne. Für kleine Werte von n sind wir allerdings zu langsam.

Satz 4.8 *$O(n^{\log_2 3})$ Bit-Operationen sind ausreichend, um zwei n-Bit Zahlen zu multiplizieren.*

Der gegenwärtige „Weltrekord“ ist $O(n \cdot \log_2 n)$ Bit-Operationen.

4.2.2 Eine Schnelle Matrizenmultiplikation

Wir betrachten jetzt das **Matrizenmultiplikationsproblem** $A \cdot B = C$ für $n \times n$ Matrizen. Wir nehmen wieder an, dass n eine Zweierpotenz ist. Wir zerlegen A und B in $n/2 \times n/2$ Teilmatrizen:

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix},$$

$$B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix}$$

Um C zu berechnen, benötigt die Schulmethode 8 Matrizenmultiplikationen der $A_{i,j}$ und $B_{j,k}$. Strassen beobachtete in 1969, dass 7 Matrizenmultiplikationen ausreichen:

$$\begin{aligned}
M_1 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}) \\
M_2 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_3 &= (A_{1,1} - A_{2,1})(B_{1,1} + B_{1,2}) \\
M_4 &= (A_{1,1} + A_{1,2})B_{2,2} \\
M_5 &= A_{1,1}(B_{1,2} - B_{2,2}) \\
M_6 &= A_{2,2}(B_{2,1} - B_{1,1}) \\
M_7 &= (A_{2,1} + A_{2,2})B_{1,1}
\end{aligned}$$

Die vier Teilmatrizen von C können jetzt wie folgt berechnet werden:

$$\begin{aligned}
C_{1,1} &= M_1 + M_2 - M_4 + M_6 \\
C_{1,2} &= M_4 + M_5 \\
C_{1,3} &= M_6 + M_7 \\
C_{1,4} &= M_2 - M_3 + M_5 - M_7
\end{aligned}$$

Wir werden also damit auf die Rekursion

$$A(n) = 7A(n/2) + c \cdot n^2$$

geführt, wenn wir nur die arithmetischen Operationen der Addition, Subtraktion und Multiplikation der Matrizeneinträge zählen. Damit ist

$$A(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2,81\ldots})$$

und wir haben die Schulmethode, die $\Theta(n^3)$ Operationen, nämlich $n-1$ Additionen und n Multiplikationen für jede der n^2 Einträge der Produktmatrix, asymptotisch geschlagen. Für Matrizen mit wenigen Hunderten von Zeilen und Spalten bleibt die Schulmethode aber überlegen.

Satz 4.9 *Zwei $n \times n$ Matrizen können mit*

$$\Theta(n^{\log_2 7})$$

arithmetischen Operationen multipliziert werden.

4.3 Dynamische Programmierung

Der Ansatz der dynamischen Programmierung ist derselbe wie für Divide & Conquer: Das Ausgangsproblem P wird in eine Hierarchie kleinerer Teilprobleme P_i zerlegt. Die dynamische Programmierung ist aber eine weitreichende Verallgemeinerung von Divide & Conquer. Wir betrachten dazu den *Lösungsgraphen*: Die Knoten des Lösungsgraphen entsprechen den Teilproblemen P_i, und wir setzen eine Kante (P_i, P_j) ein, wenn die Lösung von P_i für die Lösung von P_j benötigt wird. Um P lösen zu können, müssen wir offensichtlich fordern, dass

- der Lösungsgraph azyklisch ist, also keine Kreise besitzt und dass
- die Quellen des Lösungsgraphen, also die Knoten ohne eingehende Kanten, trivialen Teilproblemen entsprechen, die direkt lösbar sind.

Beachte, dass in Divide & Conquer nur Bäume als Lösungsgraphen auftreten, während wir jetzt beliebige azyklische Graphen zulassen. Weiterhin werden, im Unterschied zu Divide & Conquer, Lösungen für Teilprobleme *mehrmals* benötigt. Wir werden deshalb Lösungen abspeichern, um Mehrfach-Berechnungen zu verhindern.

Der schwierige Schritt im Entwurf von dynamischen Programmier-Algorithmen ist die Bestimmung der Teilprobleme: Die Teilprobleme müssen sukzessive einfacher werden, und es muss sichergestellt sein, dass schwierigere Teilprobleme direkt lösbar sind, wenn die Lösungen von etwas leichteren Problemen vorliegen. Gleichzeitig dürfen wir nicht zuviele Teilprobleme erzeugen, da wir sonst keine effizienten Algorithmen erhalten.

Der **Orakel-Ansatz** hilft oft im Entwurf guter Teilprobleme. Dazu stellen wir uns in einem Gedankenexperiment vor, dass ein Orakel Fragen über eine optimale Lösung $\mathcal{O}$ beantwortet.

(1) Wir stellen unsere Frage und erhalten die Orakel-Antwort, die uns die Bestimmung einer optimalen Lösung erleichern soll.

(2) Um dieses etwas leichtere Problem zu lösen, stellen wir weitere Fragen und denken diesen Prozess fortgesetzt, bis eine Lösung offensichtlich ist.

Sämtliche Fragen müssen wir letztlich beantworten und diese Fragen definieren unsere Teilprobleme.

Wir veranschaulichen den Orakel-Ansatz mit TSP, dem Problem des Handlungsreisenden aus Beispiel 3.3, wobei wir sogar auf die Annahme verzichten, dass die Kantenlängen eine Metrik bilden.

Beispiel 4.2 n Städte $1, \ldots, n$ sind gegeben. Wir müssen eine Rundreise minimaler Länge bestimmen, wobei die Distanz zwischen zwei Städten u, v mit `länge(u,v)` übereinstimmt.

Wir haben bereits in Beispiel 3.3 daraufhingewiesen, dass TSP ein schwieriges Optimierungsproblem ist und dürfen deshalb keine Wunder erwarten. Eine triviale Lösung besteht aus der Aufzählung aller möglichen Permutationen von $\{1, \ldots, n\}$ und der darauffolgenden Bestimmung der besten Permutation. Die Laufzeit ist fürchterlich, nämlich proportional zu $\Theta(n!)$. Für $n = 10$ sind 3,6 Millionen Touren zu inspizieren und das ist kein größeres Problem. Für $n = 13$, müssen wir 6,2 Milliarden Touren untersuchen und es wird kitzlig. Für $n = 15$ ist die Anzahl der Touren auf 1,3 Billion gestiegen und für $n = 18$ haben wir 6,3 Billiarden Touren erhalten: Das Hissen der weißen Fahne ist bereits für 18 Städte angebracht.

Wir wenden die dynamische Programmierung an und benutzen den Orakel-Ansatz. Unsere erste Frage ist auf den ersten Blick arg naiv, denn wir fragen, welche Stadt nach Stadt 1 in einer optimalen Rundreise $\mathcal{O}$ besucht wird. Aber zumindest hilft uns die Antwort in der Bestimmung einer optimalen Rundreise. Wenn nämlich die Antwort die Stadt j ist, dann benutzt $\mathcal{O}$ die Kante $(1, j)$. Unsere zweite Frage ist klar, wir fragen welche Stadt nach Stadt j besucht wird. Ist dies die Stadt k, dann wissen wir, dass $\mathcal{O}$ das Wegstück $1 \to j \to k$ durchläuft.

Die Bestimmung einer optimalen Lösung ist jetzt trivial, aber leider müssen wir unsere Fragen selbst beantworten. Auf welche Teilprobleme stoßen wir? Zu jedem Zeitpunkt arbeiten wir mit einem Wegstück $1 \to j \to \cdots \to k$ und fragen tatsächlich

> nach einem kürzesten Weg $W(1, k)$, der in k startet, alle noch nicht besuchten Städte besucht und dann in 1 endet.

Wie können wir unsere Fragen selbst beantworten? Sei $L(k, S)$ die Länge eines kürzesten Weges, der in der Stadt k beginnt, dann alle Städte in der Menge $S \subseteq V$ besucht und in Stadt 1 endet. Für jedes $k \neq 1$ und für jede Teilmenge $S \subseteq \{2, \ldots n\} \setminus \{k\}$ erhalten wir somit ein Teilproblem (k, S).

Unser dynamischer Programmieralgorithmus löst zuerst alle trivialen Teilprobleme. In unserem Fall sind dies die Teilprobleme „Bestimme $L(k, \emptyset)$“, denn es ist offensichtlich

$$L(k, \emptyset) = \texttt{länge}(k, 1).$$

Wie sieht die Hierarchie unserer Teilprobleme aus? Wenn wir $L(k, S)$ bestimmen möchten, dann sollten wir alle Möglichkeiten für die erste Kante $(k, ?)$ eines kürzesten Weges durchspielen, also alle Knoten $v \in S$ als Endpunkt der ersten Kante probieren. Wenn wir Knoten v probieren, dann muss der kürzeste Weg in v starten, alle Knoten in $S \setminus \{v\}$ besuchen und schließlich im Knoten 1 enden. Wir haben also die **Rekursionsgleichungen**

$$L(k, S) = \min_{v \in S} \{\texttt{länge}(k, v) + L(v, S \setminus \{v\})\}$$

erhalten und können sämtliche Teilprobleme nacheinander lösen.

Aber wie groß ist der Aufwand? Die Anzahl der Teilprobleme ist höchstens $(n-1)\cdot 2^{n-2}$, denn es ist stets $k \neq 1$ und $S \subseteq \{2,\dots n\} \setminus \{k\}$. Da wir für jedes Teilproblem (v,S) alle Elemente $v \in S$ ausprobieren müssen, kann jedes Teilproblem in $O(n)$ Schritten gelöst werden.

Satz 4.10 *Das allgemeine Traveling Salesman Problem für n Städte kann in Zeit $O(n^2 \cdot 2^n)$ gelöst werden.*

Wir haben also eine riesige Zahl von Teilproblemen produziert, aber unser Ansatz ist weitaus schneller als die triviale Lösung: Für $n = 15$ verlangt die triviale Lösung über 1 Billion Schritte, während wir mit 7,2 Millionen Operationen noch nicht gefordert werden. Für $n = 18$ stehen den 6,3 Billiarden Schritten der trivialen Lösung 83 Millionen Schritte der dynamischen Programmierlösung gegenüber. Während das definitive Aus der trivialen Lösung erfolgt ist, befinden wir uns weiter im grünen Bereich.

Aufgabe 40
Beschreibe einen *möglichst effizienten* Algorithmus, der für ein Array $(A[1],\dots,A[n])$ von n ganzen Zahlen die Länge einer längsten echt monoton wachsenden Teilfolge bestimmt. **Bestimme** die Laufzeit deines Algorithmus.

(Eine echt monoton wachsende Teilfolge hat die Form $(A[i_1], A[i_2], \dots, A[i_k])$ mit $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ und $A[i_1] < A[i_2] < \cdots < A[i_k]$.)

Beispiel: Das Array $(12, 18, 6, 19, 10, 14)$ besitzt $(6, 10, 14)$ (wie auch $(12, 18, 19)$) als eine längste echt monoton wachsende Teilfolge. Dein Algorithmus sollte für diese Eingabe also die Antwort 3 geben.

Aufgabe 41
Zwei Worte

$$X \equiv x_1 x_2 \cdots x_n \text{ und } Y \equiv y_1 y_2 \cdots y_m$$

über dem Alphabet $\{a, b, c, \dots, x, y, z\}$ seien gegeben. **Beschreibe** einen **möglichst effizienten** Algorithmus, der *mit einer minimalen Anzahl von Streichungen, Ersetzungen und Einfügungen von Buchstaben* das Wort X in das Wort Y überführt. **Bestimme** die Laufzeit deines Algorithmus.

Beispiel: Es sei $X \equiv abbac$ und $Y \equiv abcbc$. Eine mögliche Transformation ist zum Beispiel

abbac	$\rightarrow$	*abac*	Streichung von b
	$\rightarrow$	*ababc*	Einfügung von b
	$\rightarrow$	*abcbc*	Ersetzung von a durch c.

Es gibt aber eine kürzere Transformation, nämlich

abbac	$\rightarrow$	*abcbac*	Einfügung von c
	$\rightarrow$	*abcbc*	Streichung von a.

Hinweis: Dynamisches Programmieren ist angebracht. Betrachte die Teilprobleme $P(i,j)$, in denen das Wort $X(i) \equiv x_1 x_2 \cdots x_i$ in das Wort $Y(j) \equiv y_1 y_2 \cdots y_j$ mit möglichst wenigen Operationen zu überführen ist. In jeder Transformation von $X(i)$ nach $Y(j)$ wird irgendwann

- x_i durch y_j ersetzt oder
- x_i gestrichen oder
- y_j wird hinter x_i in $X(i)$ eingefügt.

Es kann angenommen werden, dass die jeweilige Operation am Anfang der Transformation ausgeführt wird.

Aufgabe 42
Wir möchten einen Text bestehend aus n Worten im Blocksatz setzen. Das heißt, der Text mit Ausnahme der

letzten Zeile wird links- und rechtsbündig dargestellt. Der Text bestehe aus den Worten $w_1, \ldots, w_n$. Eine Zeile biete Platz für L Zeichen. Wir nehmen $|w_i| \leq L$ für alle i an.

Um den Blocksatz zu erzielen, darf die Reihenfolge der Worte naheliegenderweise nicht verändert werden. Auch sollen Worte nicht getrennt werden. Also bleibt uns nur die Möglichkeit, die einzelnen Zeilen etwas zu strecken oder zu stauchen. Davon wollen wir nur sparsam Gebrauch machen, damit der Text möglichst lesbar ist.

Um die Lesbarkeit einer Zeile zu beschreiben, ist eine Funktion `ugly()` entwickelt worden. Sie erhält als Eingabe eine Folge von Worten und eine Zeilenlänge und liefert als Ergebnis eine Zahl, die umso höher ist, je hässlicher die Worte in einer Zeile der Länge L aussehen. (Der Wert 0 bedeutet, *keine Beanstandungen*. Je höher das Ergebnis, umso verrenkter die Zeile.)

Wir nehmen weiter an, dass speziell für die letzte Zeile eines Textes eine weitere Funktion `ugly'()` zur Verfügung steht.

Die Aufgabe ist nun, Zeilenumbrüche zu finden, so dass die Summe der `ugly()` Werte (inklusive des `ugly'()` Wertes) aller Zeilen minimiert wird.

Entwerfe einen Algorithmus, der dieses Problem in Zeit $O(n^2)$ löst. Hinweis: Dynamisches Programmieren. Die Funktion `ugly()` kann einfach aufgerufen werden. Ihre Laufzeit darf als konstant unterstellt werden.

Übrigens: Ein **mögliches** Beispiel für $ugly()$ wäre:

$$ugly(v_1, \ldots, v_r, L) = \max \left\{ \frac{L - \sum_{i=1}^{r} |v_i|}{r-1}, \frac{r-1}{L - \sum_{i=1}^{r} |v_i|} \right\} - 1$$

4.3.1 Das gewichtete Intervall Scheduling

Wir greifen das Intervall Scheduling Problem aus Abschnitt 4.1.1 wieder auf. n Aufgaben sind gegeben, wobei Aufgabe i durch das Zeitintervall $[s_i, t_i]$ beschrieben wird. Diesmal erhält jede Aufgabe i aber auch einen Wert w_i, der die Wichtigkeit der Aufgabe wiedergibt. Wir müssen eine kollisionsfreie Menge von Aufgaben auswählen, die dann auf einem einzigen Prozessor ausgeführt werden. Der Gesamtwert der ausgeführten Aufgaben ist zu maximieren.

Der Greedy-Algorithmus 4.2, der ein Scheduling nach dem Motto „Frühe Terminierungszeiten zuerst“ berechnet hat, funktioniert nicht mehr, da die Werte der Aufgaben nicht beachtet werden. Tatsächlich ist das Scheduling Problem durch die Aufnahme der Werte so stark verkompliziert worden, dass optimale Auswahlen durch Greedy-Algorithmen wahrscheinlich nicht berechnet werden können. Wir brauchen neue Ideen, übernehmen aber die Grundidee des Greedy-Algorithmus und fordern, dass die Aufgaben nach aufsteigender Terminierungszeit angeordnet sind; es gelte also $t_i \leq t_j$ falls $i < j$. Insbesondere kann Aufgabe n also nur als letzte Aufgabe ausgeführt werden.

Wir spielen den **Orakel-Ansatz** durch und fragen, ob die letzte Aufgabe n zur optimalen Auswahl A gehört.

- Bei einer positiven Antwort besteht A, neben der Aufgabe n, nur aus Aufgaben, die enden bevor Aufgabe n beginnt. Um die restlichen Auswahlen optimal zu treffen, müssen wir also nur ein kleineres Intervall Scheduling Problem lösen.

- Bei einer negativen Antwort wissen wir, dass Aufgabe n nicht gewählt wurde, und wir werden auch diesmal auf ein kleineres Intervall Scheduling Problem geführt.

Wir denken uns den Frageprozess wieder so weit ausgeführt bis optimale Lösungen trivial zu bestimmen sind. Auf welche Teilprobleme werden wir geführt, wenn wir unsere Fragen selbst beantworten müssen? Auf die Teilprobleme

Bestimme $\texttt{opt}(i)$ = Gesamtwert einer optimalen Auswahl für die ersten i Aufgaben.

Wir definieren $a(i)$ als die Aufgabe j mit spätester Terminierungszeit t_j, wobei Aufgabe j enden muss bevor Aufgabe i beginnt. Es gilt also $t_{a(i)} < s_i$ –Aufgabe $a(i)$ endet bevor Aufgabe i beginnt– und $t_j \geq s_i$ für $j > a(i)$ –Aufgaben mit späterer Terminierungszeit enden, nachdem Aufgabe i beginnt–. Wir können jetzt den zweiten zentralen Schritt durchführen, nämlich die Aufstellung der **Rekursionsgleichung**

$$\texttt{opt}(i) = \max\{\texttt{opt}(a(i)) + w_i, \texttt{opt}(i-1)\},$$

denn entweder enthält die optimale Auswahl Aufgabe i und damit sind nur Aufgaben aus der Menge $\{1, \ldots, a(i)\}$ „vorher dran“, oder die optimale Auswahl enthält Aufgabe i nicht.

Algorithmus 4.5 **Eine Lösung für das gewichtete Intervall Scheduling**

(1) Sortiere alle n Aufgaben nach aufsteigender Terminierungszeit.

Wir nehmen an, dass $t_i \leq t_j$ für $i < j$ gilt und bestimmen $a(i)$, die Aufgabe mit spätester Terminierungszeit, die vor Aufgabe i endet. Wenn es keine vor Aufgabe i endende Aufgabe gibt, dann setze $a(i) = 0$.

(2) Setze opt$(0) = 0$.

// Wenn keine Aufgabe auszuführen ist, dann entsteht kein Wert.

(3) for $(i = 1,\ i \leq n;\ i{+}{+})$

$\texttt{opt}(i) = \max\{\texttt{opt}(a(i)) + w_i, \texttt{opt}(i-1)\},$

So weit so gut. Allerdings kennen wir nur den Wert einer optimalen Auswahl, nicht aber die optimale Auswahl selbst. Kein Problem: Wir stellen die optimale Auswahl durch eine Liste dar und zwar definieren wir ein Vorgänger-Array V, das wir für die imaginaäre Aufgabe 0 mit Null initialisieren.

- Wenn $\texttt{opt}(i) = \texttt{opt}(a(i)) + w_i$, dann gehört i zur optimalen Auswahl unter den ersten i Aufgaben und wir setzen $V[i] = i$.
- Ansonsten ist $\texttt{opt}(i) = \texttt{opt}(i-1)$ und wir setzen $V[i] = V[i-1]$.

Wie kann die optimale Auswahl aus V berechnet werden?

Aufgabe 43
Zeige, dass die optimale Auswahl A mit Hilfe von V und der Funktion a durch den folgenden Algorithmus berechnet wird:

(1) $A = \emptyset$; $i = n$;
(2) while $(i! = 0)$
$A = A \cup \{V[i]\}$;
$j = a(V[i])$;
(2) Entferne die imaginäre Aufgabe aus A, falls vorhanden.

Aufgabe 44
Berechne $a(1), \ldots, a(n)$ in Zeit höchstens $O(n \cdot \log_2 n)$.

Wie groß ist der Aufwand von Algorithmus 4.5? Wir haben insgesamt n Aufgaben und die Lösung einer jeden Aufgabe gelingt in konstanter Zeit. Damit dominiert das Sortieren der Aufgaben in Schritt (1) und wir erhalten:

Satz 4.11 *Algorithmus 4.5 bestimmt eine optimale Lösung für das gewichtete Intervall Scheduling. Für n Aufgaben ist die Laufzeit durch $O(n)$ beschränkt.*

4.3.2 Kürzeste Wege und Routing im Internet

Wir beginnen mit dem All-Pairs-Shortest-Path Problem und beschreiben dann das Routing Problem im Internet als eine Anwendung.

Das All-Pairs-Shortest-Path Problem

Sei $G = (\{1, \ldots, n\}, E)$ ein gerichteter Graph mit der Längenfunktion

$$\text{länge} \ : E \to \mathbb{R}_{\geq 0}.$$

Für je zwei Knoten u und v ist

$$\text{distanz}[u][v] = \text{ die Länge eines kürzesten Weges von } u \text{ nach } v$$

zu berechnen.

Wir wenden den **Orakel-Ansatz** an und fragen nach der ersten Kante eines kürzesten Weges $W(u, v)$ von u nach v. Wenn dies die Kante (u, w) ist, dann wird $W(u, v)$, nach Herausnahme der ersten Kante, ein kürzester Weg von w nach v sein, und dieser Weg hat eine Kante weniger! Wir wiederholen unsere Fragen, bis die Bestimmung kürzester Probleme trivial ist und werden auf die Teilprobleme

$$\begin{aligned} \text{Bestimme distanz}_i[u][v] \quad &= \quad \text{die Länge eines kürzesten Weges von } u \text{ nach } v \\ &\qquad\quad \text{mit höchstens } i \text{ Kanten} \end{aligned}$$

geführt. Die **Rekursionsgleichungen**

$$\text{distanz}_i[u][v] = \min_{w,(u,w)\in E} \{ \text{ länge}(u, w) + \text{distanz}_{i-1}[w][v] \ \}$$

formalisieren das Offensichtliche: Die Länge eines kürzesten Weges von u nach v mit höchstens i Kanten ist das Minimum über die Längen aller kürzesten Wege, die von u zu einem Nachbarn w und dann von w mit höchstens $i - 1$ Kanten nach v laufen.

Algorithmus 4.6 Der Bellman-Ford Algorithmus

(1) Der gerichtete Graph $G = (V, E)$ und eine Kantengewichtung länge : $E \to \mathbb{R}$ ist gegeben. Es gelte $\text{länge}(u, u) = 0$ für alle Knoten u.

(2) Setze $\text{Nächster}[u][v] = \begin{cases} v & (u, v) \in E, \\ \text{nil} & \text{sonst} \end{cases}$ und $\text{distanz}[u][v] = \begin{cases} \text{länge}(u, v) & (u, v) \in E, \\ \infty & \text{sonst.} \end{cases}$

(3) for $(i = 1;\ i \leq n - 1,\ i{+}{+})$

for $(u = 1;\ u \leq n,\ u{+}{+})$

distanz$[u][v] = \min_{w,(u,w)\in E}\{$ länge$(u, w) +$ distanz$[w][v]\ \}$.

Wenn der Wert von distanz$[u][v]$ sinkt, dann setze $\text{Nächster}_u(v) = w$.

/* Warum können wir $\text{distanz}_i[u][v]$ durch distanz$[u][v]$ ersetzen? */

Ein kürzester Weg von u nach v kann jetzt leicht mit Hilfe des Arrays „Nächster“ bestimmt werden, da Nächster$[u][v]$ den ersten „Hop“ auf dem Weg von u nach v beschreibt: Von u müssen wir nach $u_1 =$ Nächster$[u][v]$ wandern und dann von u_1 nach Nächster$[u_1][v]$ und so weiter.

Wir kommen zur Aufwandsberechnung und betrachten die $i$te Runde. Für jeden Knoten v und jede Kante (u, w) müssen wir länge(u, w) + distanz$[w][v]$ berechnen. Also müssen wir in Runde i insgesamt $O(|V| \cdot |E|)$ Operationen ausführen und der Gesamtaufwand ist durch $O(|V|^2 \cdot |E|)$ beschränkt.

Satz 4.12 *Der gerichtete Graph $G = (V, E)$ habe n Knoten und m Kanten. Dann löst der Algorithmus von Bellman-Ford das All-Pairs-Shortest-Path Problem in Zeit $O(n^2 \cdot m)$.*

Wir könnten natürlich auch den Algorithmus von Dijkstra für jeden Knoten als Startknoten ausführen und erhalten dann die Laufzeit $O(n \cdot m \cdot \log_2 n)$: Die n-fache Anwendung von Dijkstras Algorithmus ist also wesentlich schneller. Allerdings können wir Algorithmus 4.6 beschleunigen, wenn wir länge(u, w) + distanz$[w][v]$ nur dann neu berechnen, wenn sich distanz$[w][v]$ verringert hat. Darüber hinaus werden wir später sehen, dass der Algorithmus von Bellman-Ford relativ schnelle parallele Varianten besitzt, die auch in asynchronen verteilten Systemen korrekt arbeiten.

Jedoch ist unser nächstes Ziel, eine Beschleunigung zu erreichen. Wir wenden wiederum den **Orakel-Ansatz** an, fragen aber diesmal, ob ein kürzester Weg von u nach v den Knoten n besucht.

> Wenn ja, dann wird ein kürzester Weg $W(u, v)$ von u nach v zuerst einen kürzesten Weg $W(u, n)$ von u nach n und dann einen kürzesten Weg $W(n, v)$ von n nach v durchlaufen!
>
> Und was ist die große Erkenntnis? Weder $W(u, n)$ noch $W(n, v)$ besitzen n als inneren Knoten. Diese Wege sind einfacher zu bestimmen.

- Wenn nein, dann wissen wir, dass nicht als inneren Knoten benutzt wird.

Wenn wir diesen Ansatz wiederholen, müssen wir im nächsten Schritt fragen, ob der Knoten $n - 1$ durchlaufen wird. Wir werden also zwangsläufig auf die Teilprobleme

$$\begin{aligned} \text{distanz}_k[u][v] \quad = \quad & \text{die Länge eines kürzesten Weges von } u \text{ nach } v, \\ & \text{der nur } \textit{innere} \text{ Knoten in } \{0, 1, \ldots k\} \text{ besucht.} \end{aligned}$$

geführt. Das einfachste Problem, nämlich die Berechnung von

$$\text{distanz}_0[u][v] = \begin{cases} \text{länge}(u, v) & (u, v) \in E, \\ \infty & \text{sonst} \end{cases}$$

ist trivial, denn wir dürfen keine inneren Knoten durchlaufen. Der rekursive Schritt gelingt mit den **Rekursionsgleichungen**

$$\text{distanz}_k[u][v] = \min\{\ \text{distanz}_{k-1}[u][v],\ \text{distanz}_{k-1}[u][k] +\ \text{distanz}_{k-1}[k][v]\}.$$

Mit anderen Worten, die beiden Alternativen „durchlaufe Knoten k nicht“ beziehungsweise „laufe von u nach k und dann nach v“ werden verglichen.

Eine erste Implementierung benutzt ein 3-dimensionales Array distanz und nimmt an, dass G als Adjazenzmatrix A repräsentiert ist mit

$$A[u][v] = \begin{cases} \text{länge } (u, v) & \text{falls } (u, v) \in E, \\ \infty & \text{sonst.} \end{cases}$$

Das entsprechende C++ Programm lautet:

```
for (u=1; u<=n; u++)
  for (v=1; v<=n; v++)
    distanz[-1][u][v] = A[u][v];
for (k=1; k<=n; k++)
  for (u=1; u<=n; u++)
    for (v=1; v<=n; v++)
      {
        temp = distanz[k-1][u][k] + distanz[k-1][k][v];
        distanz[k][u][v] = (distanz[k-1][u][v]> temp) ?
                           temp : distanz[k-1][u][v];
      }
```

Beachte, dass der erste Index k ohne Schaden unterdrückt werden kann: Zur Berechnung von $\text{distanz}_k[u][v]$ wird nur auf $\text{distanz}_{k-1}[u][k]$ und $\text{distanz}_{k-1}[k][v]$ zugegriffen und es ist stets

$$\text{distanz}_{k-1}[u][k] = \text{distanz}_k[u][k] \quad \text{sowie} \quad \text{distanz}_{k-1}[k][v] = \text{distanz}_k[k][v],$$

da es sinnlos ist, den Endpunkt k als inneren Knoten zu durchlaufen. Also ist das Feld distanz insgesamt überflüssig und das Segment

Algorithmus 4.7 **Der Algorithmus von Floyd**

```
for (k=1; k<=n; k++)
 for (u=1; u<=n; u++)
  for (v=1; v<=n; v++)
   {
    temp = A [u][k] + A [k][v];
    if (A[u][v] > temp) A [u][v] = temp;
   }
```

ist ausreichend. Die Laufzeit beträgt offensichtlich $\Theta(n^3)$, da wir $|V| = n$ angenommen haben. Im Vergleich zu einer Mehrfachanwendung von Dijkstras Algorithmus sind wir sogar schneller, falls der Graph fast quadratisch viele Kanten hat. Für „schlanke“ Graphen hat Dijkstra aber auch diesmal die Nase vorn. Der Programmieraufwand im Algorithmus von Floyd ist natürlich vernachlässigbar.

Satz 4.13 *Floyds Algorithmus berechnet alle kürzesten Wege für einen gerichteten Graphen mit n Knoten in Zeit*

$$\Theta(n^3).$$

Packet Routing im Internet

Die maßgebliche Aufgabe des Internet ist die Durchführung des weltweiten Datenverkehrs: Dateien verschiedenster Größe werden in kleine Pakete aufgeteilt und mit Kontrollinformationen sowie einem Absender und einer Adresse versehen. Dann werden die Pakete über eine Folge von Routern an ihr Ziel geleitet.

Die Router besitzen „Routing-Tabellen“, um den nächsten „Hop“, also den nächsten Router für ein Paket zu bestimmen. Wie sollte man die Routing-Tabellen berechnen? Pakete sollten schnell an ihr Ziel gelangen und Router sollten Pakete möglichst auf kürzesten Wegen

befördern! Allerdings haben die Verbindungen keine fixen Verzögerungen, sondern dynamische Aspekte wie Verkehrsaufkommen oder das „Ableben“ von Routern ändern Beförderungszeiten auf den Verbindungen dynamisch und entsprechend müssen sich auch Routing-Tabellen ändern. Wir besprechen die drei wichtigsten Verfahren für die Aktualierung der Routing-Tabellen.

1. Link State Algorithmen

Hier wird Dijkstras Algorithmus benutzt. Dazu muss jeder Router die Qualität aller Verbindungen des Netzes kennen. In periodischen Abständen geben deshalb die Router die Abstände zu ihren Nachbarn durch einen *Flooding* Algorithmus bekannt: Zuerst gibt jeder Router seine Nachbar-Abstände an alle Nachbarn weiter. Danach werden alle Nachbar-Informationen in $|V| - 1$ Runden durch das Netz gepumpt. Nach Beendigung des Flooding Algorithmus löst Router u das Single-Source-Shortest-Path Problem für Quelle u mit Hilfe von Dijkstras Algorithmus.

Der Berechnungs- und Nachrichtenaufwand ist natürlich beträchtlich und Link State Algorithmen skalieren deshalb schlecht bei wachsender Netzgröße. Weiterhin werden kürzeste Wege unabhängig voneinander ausgewählt, was dann zu Verbindungsüberlastungen und dementsprechend zu Qualitätsänderungen und schließlich zu neuen Wegen führen kann.

In dem OSPF-Protokoll (open shortest path first) werden Router deshalb in Areas[1] aufgeteilt: Router kennen die Qualität aller Verbindungen in ihrer Area, und damit können Link State Algorithmen innerhalb der Area eingesetzt werden. Die Verbindungswahl zwischen Areas kann ebenfalls von Link State Algorithmen wahrgenommen werden, in dem die relativ wenigen Backbone Router (Areas-verbindende Router) sich austauschen. OSPF benutzt mehrfache Distanzmetriken (wie etwa geographische Distanz, Durchsatz und Verzögerung) für die Definition der Gewichte in Dijkstras Algorithmus.

2. Distanzvektor Protokolle

Link State Algorithmen sind globale Routing Verfahren, da jeder Router die Verkehrsverhältnisse im gesamten (Teil-)Netz kennen muss. Im dezentralen Routing besitzt ein Router nur Information über die direkt mit ihm verbundenen Router. Ein jeder Router u versucht, seinen Distanzvektor D_u zu berechnen, wobei

$$D_u(v) = \text{Länge eines kürzesten Weges von Knoten } u \text{ nach Knoten } v.$$

Der Distanzvektor D_u kann zum Beispiel mit dem parallelen Bellman-Ford Algorithmus berechnet werden:

Algorithmus 4.8 **Der parallele Bellman-Ford Algorithmus**

(1) Der gerichtete Graph $G = (V, E)$ und eine Kantengewichtung länge : $E \to \mathbb{R}$ ist gegeben. Es gelte länge$(u, u) = 0$ für alle Knoten u.

(2) Setze $\text{Nächster}_u(v) = \begin{cases} v & (u,v) \in E, \\ \text{nil} & \text{sonst} \end{cases}$ und $D_u(v) = \begin{cases} \text{länge}(u,v) & (u,v) \in E, \\ \infty & \text{sonst.} \end{cases}$

[1] Typischerweise besteht eine Area aus zwischen 300-500 Routern und zwischen 700-1700 Area-lokalen Links zwischen den Routern.

(3) Wiederhole $|V| - 1$ mal:

Parallel, für jeden Knoten u berechne

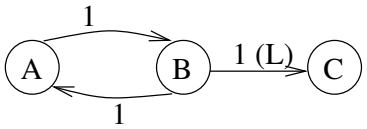
$$D_u(v) = \min_{w,(u,w)\in E} \{\text{länge}(u,w) + D_w(v)\}$$

für alle Knoten v. Wenn der Wert von $D_u(v)$ sinkt, dann setze $\text{Nächster}_u(v) = w$.

Die parallele Laufzeit von Bellman-Ford ist durch $O(|V|^2)$ beschränkt, wenn wir annehmen, dass jeder Knoten nur beschränkt viele Nachbarn hat: In $|V| - 1$ Runden sind die Distanzvektoren zu aktualisieren, wobei pro Runde eine Aktualisierung in Zeit $O(|V|)$ gelingt.

Der Algorithmus von Bellman-Ford funktioniert sogar dann noch, wenn Router asynchron rechnen und das ist im Vergleich zu anderen kürzeste Wege Algorithmen seine wesentliche Stärke. Natürlich ist die Laufzeit dann von dem langsamsten Router abhängig, aber der Algorithmus rechnet auch bei verschiedensten Taktungen der einzelnen Router weiterhin korrekt.

Allerdings reagiert Bellman-Ford träge auf veränderte Verkehrsverbindungen wie das folgende Beispiel zeigt. Wir betrachten drei Knoten A, B und C, die wie folgt angeordnet sind.



Anfänglich besitzen alle Links das Gewicht 1, aber nach einer gewissen Zeit steige das Gewicht des Links von B nach C auf L. Wenn der Bellman-Ford Algorithmus synchron ausgeführt wird, dann wird Knoten B seine Distanz zu Knoten C auf 3 von vorher 1 setzen, denn A behauptet, einen Weg der Länge 2 nach C zu besitzen. Dieser Weg führt zwar über die jetzt wesentlich langsamer gewordene Verbindung von B nach C, aber das weiss B aufgrund seiner nur lokalen Information nicht.

Daraufhin wird A seine Prognose auf 4 hochschrauben, denn seine Verbindung nach B hat die Länge 1. Nach i Iterationen behauptet B für ungerades i die Distanz $i+2$ und A die Distanz $i+1$. Erst nach proportional zu L vielen Schritten hat B die korrekte Distanz gefunden und der Prozess kommt zum Stillstand. Ist der Link von B nach C zusammengebrochen, also $L = \infty$, dann wird der Prozess sogar nicht zum Stillstand kommen und man spricht vom „Count-to-infinity“ Problem.

3. Pfadvektor Protokolle

Distanzvektor Protokolle werden zum Beispiel im „Route Information Protokoll“ (RIP) eingesetzt. RIP wird aber zum Beispiel aufgrund des Count-to-Infinity Problems nicht mehr in größeren Netzen eingesetzt, ist aber in kleineren Netzen immer noch populär. Stattdessen werden Pfadvektor Protokolle eingesetzt: Die Router berechnen nicht mehr nur den nächsten Hop, sondern halten eine möglichst vollständige Wegbeschreibung parat.

4.3.3 Paarweises Alignment in der Bioinformatik

DNA-Moleküle besitzen eine Doppelhelix (gedrehte Leiter) als Gerüst, auf dem die vier Basen *Adenin* (Abk: A), *Guanin* (Abk: G), *Thymin* (Abk: T) und *Cytosin* (Abk: C) angeordnet sind.

Der Code, also die Reihenfolge der Basen, steuert sowohl die Proteinsynthese wie auch die RNA-Synthese.

Für DNA-Sequenzen, RNA-Sequenzen wie auch für Proteine wurde empirisch festgestellt, dass häufig die Ähnlichkeit von Sequenzen auch ihre funktionelle Ähnlichkeit impliziert. „Mutter Natur“ hat sich anscheinend entschlossen, erfolgreiche Sequenzen über viele Arten hinweg mehrfach zu benutzen, wobei allerdings der direkte Vergleich durch die Evolution, also durch Mutationen erschwert wird.

Diese Gemeinsamkeiten sind die Grundlage für die Übertragbarkeit von Erkenntnissen aus der Genomanalyse von Modellorganismen auf das menschliche Genom und damit eine wichtige Basis, um genetisch beeinflusste Erkrankungen des Menschen zu diagnostizieren und zu therapieren. Da DNA-Sequenzen wie auch Proteine in ihrer Primärstruktur als Strings über dem Alphabet $\Sigma = \{A, C, G, T\}$ aufgefasst werden können, ist somit eine Ähnlichkeitsanalyse von Strings von besonderem Interesse.

Definition 4.14 *Sei Σ eine endliche Menge, das Alphabet.*

Wir bezeichnen den leeren String mit λ. Für $n \in \mathbb{N}$ ist Σ^n die Menge der Strings über Σ der Länge n und $\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$ die Menge aller Strings über Σ.

Für einen String $w \in \Sigma^*$ bezeichnet w_i den $i$ten Buchstaben von w.

Sei Σ ein endliches Alphabet und das Symbol $-$ bezeichne das Blanksymbol. Wir nehmen für das Folgende stets an, dass Σ das Blanksymbol *nicht* enthalte.

Wir führen den fundamentalen Begriff des Alignments zweier Sequenzen ein, um eine Ähnlichkeitsanlyse zweier Strings durchführen zu können.

Definition 4.15 (a) *Zwei Strings $u^*, v^* \in (\Sigma \cup \{-\})^*$ bilden ein Alignment der Strings $u, v \in \Sigma^*$ genau dann, wenn*

(1) u durch Entfernen der Blanksymbole aus u^* entsteht und v durch Entfernen der Blanksymbole aus v^* entsteht,

(2) u^* und v^* dieselbe Länge besitzen

(3) und wenn $u_i^* \neq -$ oder $v_i^* \neq -$ für jedes i.

(b) *Eine Funktion $d : (\Sigma \cup \{-\}) \times (\Sigma \cup \{-\}) \to \mathbb{R}$ heißt ein Ähnlichkeitsmaß. Die Qualität eines Alignments u^*, v^* von u und v ist definiert durch*

$$q(u^*, v^*) = \sum_i d(u_i^*, v_i^*).$$

In typischen Anwendungen wird ein „Match“ nicht-negativ gewichtet (also $d(a, a) \geq 0$ für alle $a \in \Sigma \cup \{-\}$), während ein „Mismatch“ nicht-positiv gewichtet wird (also $d(a, b) \leq 0$ für alle $a \neq b \in \Sigma \cup \{-\}$).

Im Problem des **paarweisen globalen Alignments** sind zwei Strings $u, v \in \Sigma^*$ sowie ein Ähnlichkeitsmaß d gegeben. Gesucht ist ein Alignment u^* und v^* von u und v mit maximaler Qualität.

Bemerkung 4.1 Ein optimales paarweises Alignment kann selbst bei ähnlichen Strings eine relativ geringe Qualität besitzen, weil sich die Ähnlichkeit nur lokal zeigt. Deshalb werden weitere Alignment-Varianten wie semi-globales und lokales Alignment untersucht.

Wir betrachten die Sequenzen $u \equiv GCTGATATAGCT$ und $v \equiv GGGTGATTAGCT$ und erhalten zum Beispiel das Alignment

$$\begin{aligned} u^* &\equiv \; - \; G \; C \; T \; G \; A \; T \; A \; T \; A \; G \; C \; T \\ v^* &\equiv \; G \; G \; G \; T \; G \; A \; T \; - \; T \; A \; G \; C \; T \end{aligned}$$

Dieses Alignment können wir uns auch wie folgt entstanden vorstellen, wenn wir u durch „Point-Mutationen“ nach v transformieren. Zuerst wurde in der Sequenz u der Buchstabe G in Position 1 eingefügt, sodann der Buchstabe C in Position 3 durch den Buchstaben G ersetzt und schließlich der Buchstabe A in Position 8 gelöscht. (Man bezeichnet eine Einfüge- oder Löschoperation auch als Indel-Operation –Insert/delete Operation–.) Wir erhalten jetzt eine alternative Problemstellung, wenn wir nach einer möglichst billigen Transformation von u nach v suchen: Im Problem der **minimalen Editier-Distanz** sind zwei Strings $u, v \in \Sigma^*$ sowie Kosten für das Einfügen, Löschen und Substituieren von Buchstaben gegeben, wobei diese Kosten von den betroffenen Buchstaben abhängen können; gesucht ist eine billigste Folge von Einfüge-, Lösch- und Ersetzungsoperationen, die u in v überführt. Die Gesamtkosten einer solchen billigsten Überführung werden auch als die Editierdistanz zwischen u und v bezeichnet.

Das Problem des paarweisen globalen Alignments, bzw. das äquivalente Editierproblem, erlaubt somit eine Ähnlichkeitsmessung bei etwa gleichlangen Sequenzen. Dabei hofft man, dass die funktionell-entscheidenden Regionen der jeweiligen Sequenzen nur unwesentlich durch die Evolution verändert wurden: Ein Alignment hilft somit möglicherweise in der Bestimmung dieser wichtigen Regionen.

Wir geben jetzt einen dynamischen Programmieralgorithmus für das Alignment Problem wie auch für das äquivalente Editierproblem an. (Wenn d das benutzte Ähnlichkeitsmaß ist, dann definiere $-d(a,-)$ als die Kosten einer Löschung des Buchstabens a, $-d(-,a)$ als die Kosten eines Hinzufügens des Buchstabens a und $-d(a,b)$ als die Kosten einer Ersetzung des Buchstabens a durch den Buchstaben b. Schließlich fordern wir $d(a,a) = 0$ für jeden Buchstaben a. Als Übungsaufgabe überlege man sich, dass $-D$ die Editierdistanz ist, falls D der optimale Wert eines Alignments ist.)

Wir wenden den **Orakel-Ansatz** an und fragen nach den beiden letzten Buchstaben eines optimalen Alignments von u und v. Wenn die letzten Buchstaben von u und v miteinander aligniert werden, dann müssen wir das kleinere Alignment-Problem auf den Präfixen von u und v, nach Entfernen der jeweilig letzten Buchstaben, lösen. Bestehen die letzten Buchstaben eines optimalen Alignments aus dem letzten Buchstaben von u und dem Blanksymbol, dann haben wir ebenfalls eine Reduktion erreicht, denn wir müssen ein optimales Alignment für u, mit dem letzten Buchstaben entfernt, und v bestimmen. Die Situation ist ähnlich, wenn die letzten Buchstaben eines optimalen Alignments aus dem letzten Buchstaben von v und dem Blanksymbol bestehen.

Wenn wir uns die Fragen fortgesetzt denken, erhalten wir die Teilprobleme

> „Bestimme den Wert $D(i,j)$ eines optimalen Alignments zwischen den Präfixen $u^{(i)} \equiv u_1 \cdots u_i$ und $v^{(j)} \equiv v_1 \cdots v_j$.

Zuerst beachten wir, dass

$$D(i,0) = \sum_{k=1}^{i} d(u_k,-) \;\text{ und }\; D(0,j) = \sum_{k=1}^{j} d(-,v_k),$$

denn wir müssen ausschließlich Buchstaben löschen bzw. hinzufügen. Für die Berechnung von $D(i,j)$ unterscheiden wir drei Fälle, abhängig von der Behandlung von u_i bzw. v_j.

Fall 1: Ein optimales Alignment fügt Blanksymbole nach u_i ein. Wir erhalten

$$D(i,j) = D(i,j-1) + d(-,v_j),$$

denn in diesem Fall ist ein optimales Alignment zwischen $u^{(i)}$ und $v^{(j)}$ durch ein optimales Alignment zwischen $u^{(i)}$ und $v^{(j-1)}$ gegeben. Der letzte Buchstabe v_j von $v^{(j)}$ wird gegen ein Blank aligniert und provoziert den Term $d(-,v_j)$.

Fall 2: Ein optimales Alignment fügt Blanksymbole nach v_j ein. Wir erhalten

$$D(i,j) = D(i-1,j) + d(u_i,-),$$

denn diesmal ist ein optimales Alignment zwischen $u^{(i)}$ und $v^{(j)}$ durch ein optimales Alignment zwischen $u^{(i-1)}$ und $v^{(j)}$ gegeben. Der letzte Buchstabe u_i von $u^{(i)}$ wird gegen ein Blank aligniert und provoziert den Term $d(u_i,-)$.

Fall 3: Ein optimales Alignment fügt Blanksymbole weder nach u_i noch nach v_j ein. Damit wird also die letzte Position von u^i gegen die letzte Position von v^j „gematcht“ und wir erhalten ein optimales Alignment zwischen $u^{(i)}$ und $v^{(j)}$ über ein optimales Alignment zwischen $u^{(i-1)}$ und $v^{(j-1)}$ mit anschließendem Match zwischen den letzten Buchstaben u_i und v_j. Wir erhalten somit die **Rekursionsgleichungen**

$$D(i,j) = D(i-1,j-1) + d(u_i,v_j).$$

Wir haben zwar in jedem der drei Fälle eine Rekursion erhalten, wissen aber natürlich, welcher dieser drei Fälle denn wirklich eintritt! Kein Problem, wir müssen alle drei Fälle simultan verfolgen und wählen dann den besten Fall.

Algorithmus 4.9 **Paarweises Alignment**

(1) /* Initialisierung */
$D(0,0) = 0$;
for ($i = 1$; $i <= n$; $i++$)
  $D(i,0) = \sum_{k=1}^{i} d(u_k,-)$;
for ($j = 1$; $j <= m$; $j++$)
  $D(0,j) = \sum_{k=1}^{j} d(-,v_k)$;

(2) /* Iteration */
for ($i = 1$; $i <= n$; $i++$)
  for ($j = 1$; $j <= m$; $j++$)
    $D(i,j) = \max\{D(i,j-1)+d(-,v_j), D(i-1,j)+d(u_i,-), D(i-1,j-1)+d(u_i,v_j)\}$.

Satz 4.16 *Der Algorithmus löst das globale Alignment Problem für zwei Sequenzen u und v der Längen n und m in Zeit und Platz $O(n \cdot m)$.*

Beweis: Wir haben genau $(n+1) \cdot (m+1)$ Teilprobleme $D(i,j)$. Da jedes Teilproblem in konstanter Zeit gelöst werden kann, ist die Laufzeit durch $O(n \cdot m)$ beschränkt. □

4.3.4 Voraussage der RNA Sekundärstruktur

Während die Doppelhelix der DNA aus zwei komplementären Strängen mit den komplementären Basenpaaren (A, T) und (G, C) zusammengesetzt ist, besitzt die RNA nur einen einzigen Strang. Zusätzlich wird Thymin durch Uracil (Abk: U) ersetzt. Von besonderem Interesse für die Funktionalität eines RNA Moleküls ist seine *Sekundärstruktur*, also das zweidimensionale Faltungsverhalten des Moleküls: Der Strang des RNA Molekls faltet sich, um Bindungen zwischen den (neuen) komplementären Basen $\{A, U\}$ und $\{C, G\}$ einzugehen.

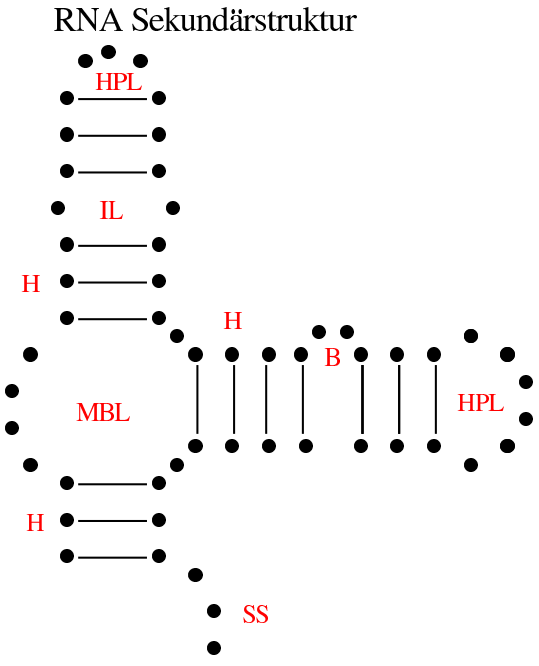
Definition 4.17 *Sei $R \in \{A, C, G, U\}^n$ die Primärstruktur eines RNA Moleküls. Die Sekundärstruktur von R wird durch eine Menge $P \subseteq \{\, \{i, j\} \mid 1 \leq i \neq j \leq n\} \,\}$ von Paarmengen beschrieben, wobei folgende Bedingungen gelten müssen:*

(a) Für jedes Paar $\{i, j\} \in P$ gilt $|i - j| \geq 5$. Bindungen können also nur ab Distanz 5 eingegangen werden, weil sonst die Faltung zu „scharf abknickt“.

(b) Die Paare in P bilden ein Matching. Jede Position $i \in \{1, \ldots, n\}$ gehört somit zu höchstens einem Paar in P.

(c) Paare in P entsprechen komplementären Basen. Für jedes Paar $(i, j) \in P$ ist somit entweder $\{R_i, R_j\} = \{A, U\}$ oder $\{R_i, R_j\} = \{C, G\}$.

(d) Die Faltung besitzt keine „Überkreuzungen“: Wenn $\{i, j\} \in P$ und $\{k, l\} \in P$ für $i < j$ und $k < l$ gilt, dann ist $i < k < j < l$ ausgeschlossen.



B bulge, H helix, HPL hairpin loop, IL internal loop, MBL multibranched loop or junction, SS single strand [2]

Die obigen Bedingungen beschreiben die Sekundärstruktur nur partiell. Zum Einen gibt es durchaus (allerdings nur eher seltene) Ausnahmen. Zum Anderen lassen wir bisher sogar die leere Menge als Sekundärstruktur zu. Die gängige Hypothese ist aber, daß ein RNA-Molekül

[2] Aus: www.zib.de/MDGroup/temp/lecture/l5/p_secondary/rna_second.pdf

versucht, seine freie Energie zu minimieren und wir nehmen hier vereinfacht an, dass die freie Energie minimiert wird, wenn möglichst viele Bindungen zwischen komplementären Basen eingegangen werden. Hier ist also unser algorithmisches Problem:

> Ein String $R \in \{A, C, G, U\}^n$ sei vorgegeben. Bestimme eine RNA-Sekundärstruktur P für R gemäß Definition 4.17, so daß P möglichst groß ist.

Wie üblich wenden wir den **Orakel-Ansatz** an und fragen diesmal, ob und wenn ja mit welcher Position k die letzte Position n von R eine Bindung eingeht. Der Ausschluß von Überkreuzungen, also die Eigenschaft (d) in Definition 4.17, ist jetzt eine große Hilfe: Wenn eine optimale Sekundärstruktur P_{opt} das Paar $\{k, n\}$ besitzt, dann zerfällt P_{opt} in zwei, notwendigerweise optimale Sekundärstrukturen und zwar in die Sekundärstrukturen für den Präfix von R zu den Positionen $1, \ldots, k-1$ und den Suffix zu den Positionen $k+1, \ldots, n-1$.

Setzen wir den Frageprozess fort, dann sind wir gezwungen, optimale Sekundärstrukturen für alle Teilstrings von R zu berechnen. Wir definieren deshalb die Teilprobleme

Bestimme $D(i, j) =$ Die Größe einer optimalen Sekundärstruktur für den Teilstring $R_i \cdots R_j$

Um $D(i, j)$ zu berechnen, müssen wir zwei Fälle unterscheiden. Wenn j nicht in einem Match involviert ist, dann ist $D(i, j) = D(i, j-1)$ und dieser Fall ist einfach.

Wenn P_{opt} hingegen das *wählbare* Paar (k, j) besitzt, dann ist $D(i, j) = D(i, k-1) + D(k+1, j-1) + 1$. (Das Paar (k, j) heißt wählbar, falls $i \leq k \leq j-5$ und $\{R_k, R_j\} = \{A, U\}$ oder $\{R_k, R_j\} = \{C, G\}$ gilt. Damit setzen wir also die Eigenschaften (a) und (c) aus Definition 4.17 um. Durch die Wahl der Intervallgrenzen $(i, k-1)$ und $(k+1, j-1)$ sichern wir auch Eigenschaft (b), denn weder k noch j kann in einem weiteren Paar auftreten.) Damit erhalten wir also die **Rekursionsgleichungen**

$$D(i, j) = \max_{k, (k,j) \text{ ist waehlbar}} \{D(i, k-1) + D(k+1, j-1) + 1, D(i, j-1)\} \tag{4.1}$$

und werden auf das folgende Verfahren geführt.

Algorithmus 4.10 **RNA Sekundärstruktur**

(1) for $(l = 1;\ l <= 4;\ l{+}{+})$
  for $(i = 1;\ i <= n - l;\ i{+}{+})$
    Setze $D(i, i + l) = 0$;

(2) for $(l = 5;\ l <= n - 1;\ l{+}{+})$
  for $(i = 1;\ i <= n - l;\ i{+}{+})$
    Bestimme $D(i, i + l)$ mit Rekursion 4.1.

Wir müssen insgesamt $\binom{n}{2}$ Teilprobleme $D(i, j)$ lösen, wobei die Lösung eines Teilproblems in Zeit $O(n)$ gelingt. damit ist kubische Laufzeit $O(n^3)$ ausreichend.

Satz 4.18 *Die RNA Sekundärstruktur in der obigen Definition kann für einen String der Länge n in Zeit $O(n^3)$ berechnet werden.*

Aufgabe 45
Wir wollen den Absatz eines Textes auf einem Drucker ausgeben, so dass er nicht allzu häßlich aussieht. Der Absatz besteht aus den Wörtern $w_1, \ldots, w_n$, die die Längen $l_1, \ldots, l_n$ haben; jede Zeile habe die Länge M.

Wenn eine Zeile die Wörter i bis j enthält und jeweils ein Leerzeichen - dieses hat die Länge 1 - zwischen den Wörtern steht, so bleibt am Ende ein leerer Platz der Länge $p = M - j + i - \sum_{k=i}^{j} l_k$. Die „Kosten“ eines Absatzes seien dann die summierten dritten Potenzen dieser Werte (um so Zeilen mit viel verbleibendem Platz stärker zu ahnden) für alle Zeilen außer der letzten, da diese kurz sein darf, ohne dass das Layout verunstaltet wird. Diese Kosten sollen mit Hilfe des dynamischen Programmierens minimiert werden. So betragen die Kosten für

```
Wir wollen einen
Absatz eines
Textes
```

für $M = 17$ gerade $1^3 + 5^3 = 126$. Es kann allerdings sein, dass diese Formatierung nicht optimal ist.

Gib einen Algorithmus **an**, der die minimalen Kosten für gegebene Wörter $w_1, \ldots, w_n$ mit gegebenen Längen $l_1, \ldots, l_n$ und gegebener Zeilenlänge M löst.

Aufgabe 46
Gegeben sind n Objekte mit den Gewichten $g_1, \ldots, g_n$ und den Werten $w_1, \ldots, w_n$. (D.h. das Objekt i hat das Gewicht g_i und den Wert w_i.) Außerdem erhalten wir einen Rucksack mit einer Gewichtsschranke G.

Wir wollen den Rucksack optimal bepacken, d.h. der Wert der eingepackten Objekte soll maximal sein, ohne dass die Gewichtsschranke G des Rucksacks überschritten wird.

Jedes Objekt steht uns einmal zur Verfügung und muss immer *ganz* oder *gar nicht* eingepackt werden.

(Wir können die Aufgabe wie folgt formalisieren:
Der Wert W einer Bepackung ist $W = \lambda_1 w_a + \lambda_2 w_2 + \ldots + \lambda_n w_n$, wobei für $1 \leq i \leq n$ gilt: $\lambda_i = 1$ (bzw. 0), falls Objekt i in den Rucksack gepackt wird (bzw. nicht in den Rucksack gepackt wird). Der Wert W ist zu *maximieren* unter der Bedingung, dass $\lambda_1 g_1 + \lambda_2 g_2 + \ldots + \lambda_n g_n \leq G$ gilt.)

Entwirf einen *möglichst effizienten* Algorithmus, der den Wert einer optimalen Bepackung des Rucksacks für beliebige Eingaben $g_1, \ldots, g_n, w_1, \ldots, w_n, G \in \mathbb{N}$ berechnet. (Laufzeit $O(n \cdot G)$ ist möglich.) **Benutze** die Entwurfsmethode des dynamischen Programmierens.

4.4 Die Lineare Programmierung

Die Lineare Programmierung ist eine sehr mächtige Technik für die Lösung von Optimierungsproblemen. Bei dieser Methode sind Probleme als **lineares Ungleichungssystem** über n Unbekannten darzustellen. Außerdem ist eine ebenfalls lineare **Zielfunktion** anzugeben, deren Wert zu optimieren ist.

Optimale Produktion: Ein Unternehmer verfügt über eine Maschine, auf der er zwei verschiedene Produkte P_1 und P_2 produzieren kann. Eine Einheit von P_1 kann er mit 7 Euro Gewinn verkaufen, eine Einheit P_2 bringt 4 Euro im Verkauf. Die Produktion einer Einheit von P_1 nimmt die Maschine 4 Minuten in Anspruch, eine Einheit P_2 ist in 2 Minuten fertig. Das Ziel ist natürlich, eine Gewinnmaximierung.

Weiterhin nehmen wir folgende Beschränkungen und Auflagen an:

- Insgesamt hat der Unternehmer im Monat 240 Maschinenstunden zur Verfügung. Das sind 14400 Minuten.

- Für die Produktion von P_1 ist man auf einen Rohstoff angewiesen. Für jede Einheit werden 3 Liter dieses Rohstoffes gebraucht. Insgesamt kann der Zulieferer maximal 9000 Liter des Rohstoffes bereitstellen.

- Bei der Produktion einer Einheit von P_2 werden 5 Gramm eines umweltlich nicht ganz unbedenklichen Gases freigesetzt. Die Unternehmer haben es sich daher zur Auflage gemacht, den monatlichen Ausstoß dieses Gases auf 20kg zu beschränken.

- Für 200 Einheiten P_1 und 500 Einheiten P_2 existieren bereits Verträge. Diese Mengen müssen also mindestens produziert werden.

- Wird die Maschine weniger als 70 Stunden eingesetzt, besteht die Gefahr, dass sie Schaden nimmt. Das muss verhindert werden.

- Nach der Produktion müssen die Erzeugnisse eine gewisse Zeit gelagert werden, bevor man sie absetzen kann. Das Lager der Firma bietet Platz für 5000 Einheiten.

Erstellen wir zunächst das entsprechende Ungleichungssystem. Dabei steht x_1 für die Anzahl der produzierten Einheiten von P_1 und x_2 analog für die von P_2 produzierte Menge. Die Zielfunktion

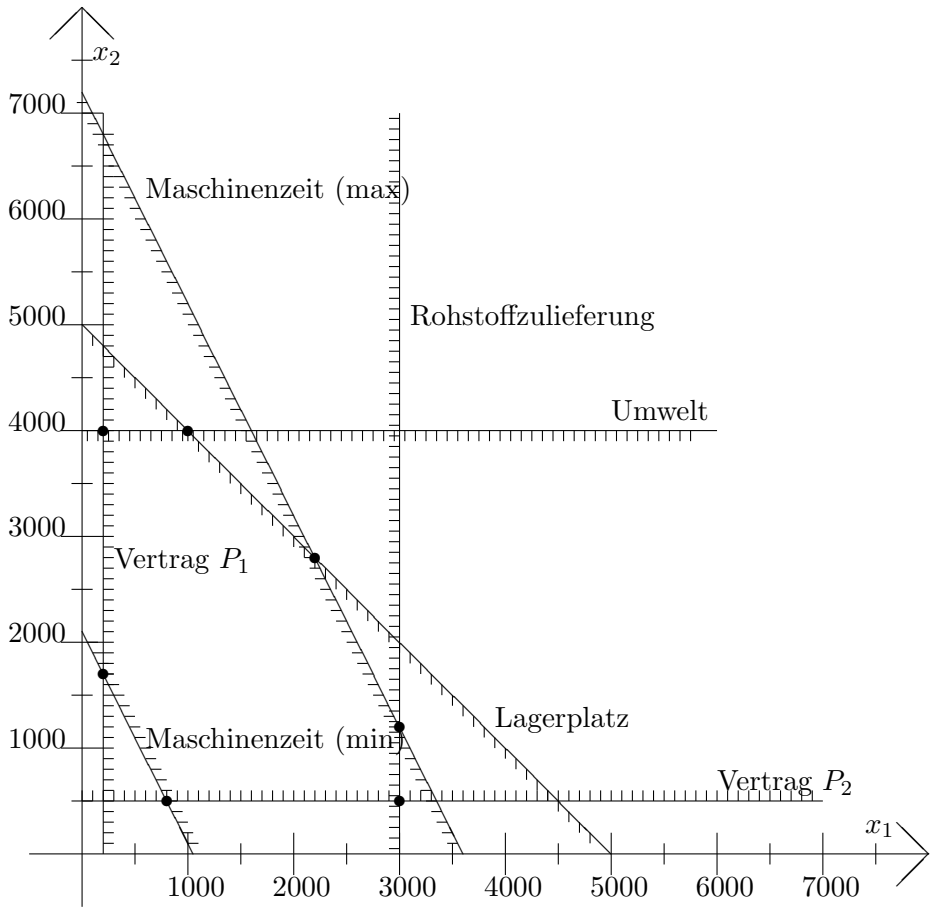
$$Gewinn(x_1, x_2) = 7x_1 + 4x_2$$

ist dann unter den Nebenbedingungen

$$\begin{aligned} 4x_1 + 2x_2 &\leq 14400 \\ 3x_1 &\leq 9000 \\ 5x_2 &\leq 20000 \\ x_1 &\geq 200 \\ x_2 &\geq 500 \\ 4x_1 + 2x_2 &\geq 4200 \\ x_1 + x_2 &\leq 5000 \end{aligned}$$

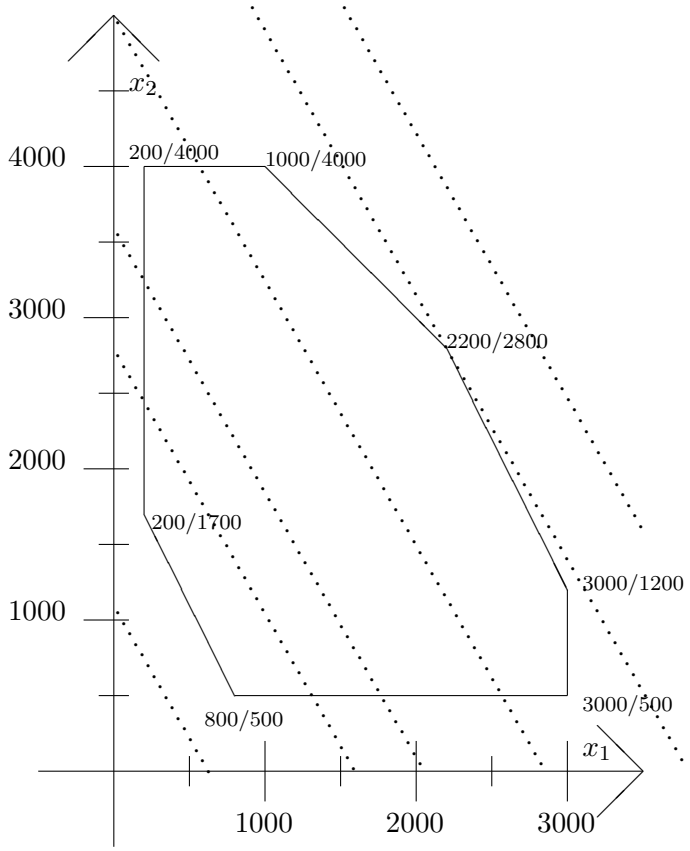
zu maximieren.

Das Ungleichungssystem faßt die Einschränkungen an den *Lösungsraum* zusammen. Genau die (x_1, x_2) Tupel, die konform zu allen Restriktionen sind, sind Lösungen. Unter ihnen ist die Lösung mit dem optimalen Zielwert ausfindig zu machen. Veranschaulichen wir uns die Restriktionen einmal grafisch.



Wir erkennen ein Siebeneck innerhalb dessen sich die Menge aller *legalen* x_1, x_2-Kombinationen befindet. Sieben ist auch die Anzahl der Restriktionen des Systems. In unserem Fall hat jede Restriktion zu einer Einschränkung der Lösungsmenge geführt. Das ist nicht zwangsläufig immer so. Läge die Begrenzung durch den Rohstoffzulieferer beispielsweise bei $x_1 = 4000$, so wäre sie implizit schon durch die schranke für die Maschinenzeit erfüllt. Solche Restriktionen, die zu keiner Einschränkung der Lösungsmenge führen, nennt man *redundant*.

Auch hätte es passieren können, dass die Menge der mit allen Restriktionen verträglichen Punkte kein geschlossenes Vieleck bildet, sondern in eine Richtung unbegrenzt ist. In dem Fall sind die folgenden Betrachtungen sinngemäß anzupassen.



In dieser Skizze ist das Vieleck der Lösungen noch einmal unter Angabe der Eckpunkte eingezeichnet. Gepunktet sehen wir eine Auswahl von linearen Gleichungen, die $7x_1 + 4x_2 = c$ (für diverse c) erfüllen. Alle Punkte gleichen Zielwertes liegen auf einer solchen Gerade mit eben dieser Steigung. Weiter oben verlaufende Geraden stellen dabei höhere Zielwerte dar. Um unser Optimierungsproblem zu lösen, müssen wir die höchste dieser Geraden, die das *Planungsvieleck* berührt, finden.

In unserem konkreten Fall können wir ablesen, dass der Punkt $(2200, 2800)$ optimal ist. Er liegt auf der höchsten Gerade, die das Vieleck berührt. Der optimale Zielwert ist also $7 \cdot 2200 + 4 \cdot 2800 = 26600$.

Zum Vergleich: Die benachbarten Eckpunkte (1000/4000) bzw. (3000/1200) haben die Zielwerte 23000 bzw. 25800.

Wir können hier erkennen, dass bei der Bestimmung der optimalen Lösung zwei Fälle auftreten können.

- **1.Fall :** Die Geraden der Punkte gleicher Zielwerte verlaufen **nicht** parallel zu einer der relevanten (hier oberen) Seiten des Vielecks. In dem Fall stellt ein konkreter Eckpunkt des Vielecks eindeutig die optimale Lösung dar. So ist es in unserem Fall.
- **2.Fall :** Die Geraden der Punkte gleicher Zielwerte verlaufen parallel zu einer der relevanten Seiten des Vielecks. In diesem Fall hat jeder Punkt entlang dieser Seitenlinie des Vielecks denselben optimalen Zielwert. Auch die beiden Eckpunkte, die die Seite begrenzen, sind optimale Lösungen.

In jedem der beiden Fälle ist eine Ecke des Vielecks optimal. Es genügt also einfach die Ecken des Restriktionsvielecks auszuwerten. Ist das aber im Falle von zwei Veränderlichen eine noch recht überschaubare Aufgabe, so wird es bei mehr Unbekannten (das entspricht dem Arbeiten in höheren Dimensionen) schnell sehr viel schwerer.

Für die Beschreibung effizienter Algorithmen der linearen Programmierung verweisen wir auf die Bachelor Veranstaltung „Effiziente Algorithmen“ oder auf die Hauptstudiumsvorlesung „Approximationsalgorithmen“.

An dieser Stelle wollen wir an einem weiteren Beispiel zeigen, dass sich auch solche Probleme als lineares Programm formulieren lassen, die auf den ersten Blick nichts mit linearen Ungleichungen oder linearen Zielfunktionen zu tun haben.

Das **Gewichtete Matching Problem**

Aufgabe ist es zu einem gegebenen gewichteten bipartiten Graphen $G = (V, E)$ mit Kantengewichtung $w : E \to \mathbb{R}_{\geq 0}$ ein maximales Matching $E' \subseteq E$ zu bestimmen.

E' heißt ein Matching, wenn verschiedene Kanten in E' keinen gemeinsamen Endpunkt besitzen. E' ist ein maximales Matching, wenn das Gesamtgewicht aller Kanten in E' maximal unter allen Matchings ist.

Was hat dies nun mit linearen Restriktionen und linearen Zielfunktionen zu tun? Was sind hier die Veränderlichen? Wir gehen das Problem wie folgt an: Für jede Kante $e \in E$ richten wir eine Variable x_e ein. Wir verlangen für jede dieser Variablen die Ungleichung $0 \leq x_e \leq 1$ und beabsichtigen die folgende Interpretation:

$$\begin{aligned} x_e = 1 &\quad \leftrightarrow \quad e \in E' \\ x_e = 0 &\quad \leftrightarrow \quad e \notin E' \end{aligned}$$

Unsere Veränderlichen sollen also als *Indikatorvariablen* Verwendung finden. Unser Ziel läßt sich nun als zu maximierende Funktion schreiben.

$$MAX(\sum_{e \in E} w(e) \cdot x_e)$$

Die Anforderung, dass die Teilmenge ein Matching darstellen muss, läßt sich durch die Ungleichung

$$\sum_{v \in V \text{s.d.} \{u,v\} = e \in E} x_e \quad \leq 1$$

für jeden einzelnen Knoten $u \in V$ formulieren.

Ist die lineare Programmierung also das Allheilmittel? Leider nicht. Wir müssen nämlich noch auf eine wesentliche Problematik aufmerksam machen. Dass der Algorithmus nicht weiß, wie seine Lösung interpretiert wird, führt auch dazu, dass er sinnlose „Lösungen“ produzieren kann. Insbesondere ist es für viele Anwendungen notwendig, dass die ermittelten Unbekannten ganzzahlig sind.

Denken wir noch mal an den Unternehmer, der die optimale Produktion für seine Maschine sucht. Wenn es sich bei den beiden Produkten beispielsweise um zwei verschiedene Biersorten handelt (gemessen in hl), so haben wir keine Probleme, eine Ausgabe von z.B. 722.5 / 1019.32 zu akzeptieren. Handelt es sich aber um Fernseher (gemessen in Stück), so tun wir uns deshalb schwer, weil wir für den halben Fernseher voraussichtlich nicht den halben Preis am Markt erzielen werden.

Ebenso verhält es sich bei unserem Beispiel mit dem maximalen Matching. Die diversen x_e sollten dringend ganzzahlig sein. Für eine Ausgabe *„Wähle Kante e zur Hälfte aus!“*, haben wir keine Verwendung.

Für die Klasse der bipartiten Graphen läßt sich zeigen, dass alle Ecken ganzzahlig sind. Für allgemeine Graphen ist diese Eigenschaft leider falsch, und die Gefahr, eine unbrauchbare *Lösung* zu erhalten, bleibt.

Die entscheidende Frage ist nun: Wie schwer ist das Problem der linearen Programmierung? Die Antwort: Glücklicherweise gibt es schnelle Algorithmen. Wie schwer ist dann lineare Programmierung, wenn wir optimale ganzzahlige Lösungen benötigen? In Kapitel 6.2.3 zeigen wir, dass die ganzzahlige Programmierung ein NP-vollständiges Problem und damit sehr schwer ist. Es existieren also höchstwahrscheinlich keine effizienten Algorithmen.

4.5 Zusammenfassung

Wir haben die Entwurfsmethoden Greedy-Algorithmen, Divide & Conquer, dynamische Programmierung, und die lineare Programmierung betrachtet. Wir werden im Teil III weitere Entwurfsmethoden kennenlernen, um schwierige Berechnungsprobleme angreifen zu können.

Greedy-Algorithmen berechnen einen optimalen Lösungsvektor Komponente nach Komponente, wobei der Wert einer gesetzten Komponente nie verändert wird. Bereits in Kapitel 3 haben wir die Greedy-Algorithmen von Kruskal, Prim und Dijkstra besprochen. Wir haben zusätzlich noch Greedy-Algorithmenn für die Lösung einfacher Scheduling Probleme beschrieben und Huffman-Codes konstruiert. Approximationsalgorithmen für schwierige Optimierungsprobleme bilden ein sehr wichtiges Anwendungsgebiet von Greedy-Algorithmen; wir lernen Beispiele in Kapitel 7 kennen.

In Divide & Conquer ist ein Ausgangsproblem in kleinere Teilbäume zu zerlegen, deren Lösung eine Lösung des Ausgangsproblems ermöglicht. Beispiele für diese Entwurfsmethode sind viele Sortierverfahren, Suchverfahren in Bäumen und Graphen, Matrizenmultiplikation und die schnelle Multiplikation ganzer Zahlen.

In der dynamischen Programmierung sind zusätzlich die Lösungen der Teilprobleme abzuspeichern, da sie vielfach benötigt werden. Als Beispiele haben wir die kürzesten-Wege Algorithmen von Floyd und Bellman-Ford sowie die Probleme des paarweisen Alignments und der RNA Sekundärstruktur in der Bioinformatik betrachtet. Während in Divide & Conquer die Teilprobleme durch einen Baum, dem Rekursionsbaum, angeordnet sind, entspricht die Problemhierarchie für das dynamische Programmieren einem azyklisch gerichteten Graphen, da auf Problemlösungen mehrfach zugegriffen wird.

Mit der linearen Programmierung haben wir eine mächtige Technik in der exakten Lösung von Optimierungsproblemen kennengelernt. Eine in größere Tiefe gehende Behandlung findet man in der Veranstaltung „Approximationsalgorithmen“.

Teil II

NP-Vollständigkeit

Kapitel 5

P, NP und die NP-Vollständigkeit

Bereits in den 70er Jahren hat man festgestellt, dass eine große Zahl algorithmischer Probleme „anscheinend“ keine effizienten Lösungen zulassen. Wir möchten diesem Phänomen auf den Grund gehen und betrachten anscheinend einfache Ja-Nein Varianten dieser Probleme, nämlich ihre Entscheidungsprobleme:

- Das Erfüllbarkeitsproblem SAT: für eine aussagenlogische Formel α ist zu entscheiden, ob α erfüllbar ist. Das Entscheidungsproblem des Erfüllbarkeitsproblems ist

$$SAT = \{\alpha \mid \text{die aussagenlogische Formel } \alpha \text{ ist erfüllbar } \}.$$

- Das Clique Problem $CLIQUE$: für einen ungerichteten Graphen G und eine Zahl $k \in \mathbb{N}$ ist zu zu entscheiden, ob G eine Clique[1] der Größe k besitzt. Das Entscheidungsproblem des Clique Problems ist

$$CLIQUE = \{(G, k) \mid G \text{ hat eine Clique der Größe } k \ \}.$$

- Das Problem LW der längsten Wege: für einen gerichteten Graphen G und eine Zahl k ist zu entscheiden, ob G einen Weg der Länge mindestens k besitzt. Das Entscheidungsproblem von LW ist

$$LW = \Big\{(G, k) \,|\, G \text{ besitzt einen Weg der Länge mindestens } k\Big\}.$$

  Das Problem der Bestimmung kürzester Wege ist effizient lösbar, zum Beispiel durch Dijkstras Algorithmus. Das Problem längster Wege wird sich hingegen als NP-vollständig herausstellen und besitzt somit in aller Wahrscheinlichkeit keine effizienten Algorithmen.

- Das Binpacking Problem $BINPACKING$: Es ist zu entscheiden, ob n Objekte mit den Gewichten $0 \leq w_1, \ldots, w_n \leq 1$ in höchstens k Behälter gepackt werden können. Dabei darf ein Behälter nur Objekte vom Gesamtgewicht höchstens 1 aufnehmen. Das Entscheidungsproblem des Binpacking Problems ist

$$BINPACKING = \{(w_1, \ldots, w_n, k) \mid k \text{ Behälter können alle Objekte aufnehmen } \}.$$

[1] G besitzt eine Clique der Größe k, falls es k Knoten gibt, so das je zwei Knoten durch eine Kante verbunden sind.

- Die ganzzahlige Programmierung GP: für eine $m \times n$ Matrix A von rationalen Zahlen und Vektoren $b \in \mathbb{Q}^m, c \in \mathbb{Q}^n$ sowie einen Schwellenwert $t \in \mathbb{Q}$ ist zu entscheiden, ob es einen Lösungsvektor $x \in \mathbb{Z}^n$ mit

  $$A \cdot x \geq b \text{ und } c^T \cdot x \leq t$$

  gibt. Das Entscheidungsproblem der ganzzahligen Programmierung ist

  $$GP = \{(A, b, c) \mid \text{es gibt } x \in \mathbb{N}^n \text{ mit } A \cdot x \geq b \text{ und } c^t x \leq t\}.$$

- Das Shortest-Common-Superstring Problem SCS: Für Strings $s_1, \ldots, s_n$ über einem gemeinsamen Alphabet Σ und für eine Zahl $m \in \mathbb{N}$ ist zu entscheiden, ob es einen „Superstring“ s der Länge höchstens m gibt, der alle Strings s_i als Teilstrings enthält. Das Shortest-Common-Superstring Problem modelliert die Shotgun-Sequenzierung, in der mehrere Kopien eines DNA-Strings zuerst in kleine, sich überlappende Fragmente zerlegt wird, die Fragmente darauffolgend sequenziert werden und der ursprüngliche DNA-String dann aus den squenzierten Fragmenten zu rekonstruieren ist. In dieser Formulierung nimmt man also an, dass der DNA-String ein kürzester Superstring seiner Fragmente ist. Das Entscheidungsproblem von SCS ist

  $$SCS = \{(s_1, \ldots, s_k, m) \mid \text{es gibt einen Superstring der Länge höchstens } m \}.$$

Was ist die gemeinsame Struktur dieser Probleme? Im Erfüllbarkeitsproblem müssen wir überprüfen, ob eine aussagenlogische Formel α erfüllbar ist. α besitzt aussagenlogische Variablen $x_1, \ldots, x_n$ und es genügt, wenn wir eine Belegung $x_1 = b_1, \ldots, x_n = b_n$ für $b_1, \ldots, b_n \in \{0, 1\}$ *raten* und dann überprüfen, ob $\alpha(b_1, \ldots, b_n)$ wahr ist. Die Überprüfung kann dann natürlich effizient durchgeführt werden. Diese Eigenschaft, nämlich

kurze, schnell verifizierbare Lösungen zu besitzen,

finden wir auch in allen anderen Probleme wieder. Im Clique Problem zum Beispiel würden wir k Knoten raten und in Zeit $O(k^2)$ überprüfen, dass die k Knoten tasächlich eine Clique bilden. Im Shortest-Common-Superstring Problem genügt es einen String s der Länge höchstens m zu raten; wir können dann in Zeit höchstens $O(n \cdot m^2)$ überprüfen, ob tatsächlich jeder String s_i ein Teilstring von s ist. (In Zeit $O(m^2)$ können wir entscheiden, ob ein String s_i Teilstring von s ist. Tatsächlich kann dieses Teilstring-Problem sogar sehr viel schneller, nämlich in Zeit $O(m)$ gelöst werden.)

Nun haben unsere Rechner allerdings nicht die Fähigkeit eine Lösung erfolgreich zu raten, wenn denn eine Lösung existiert, aber wir stellen uns einen hypothetischen Rechner mit dieser Fähigkeit, nämlich einen **nichtdeterministischen Rechner** vor. Dieser nichtdeterministische Rechner kann alle obigen Probleme mit Leichtigkeit lösen, denn nicht nur gelingt stets eine schnelle Verifikation, sondern die Anzahl geratener Wahrheitswerte (für SAT), geratener Knoten (für $CLIQUE$) und die Länge eines geratenen Superstrings (für SCS) ist im Vergleich zur Länge der Eingabe moderat.

Diese Beobachtung war Anlass, nichtdeterministische Rechner zu untersuchen und die Komplexitätsklasse NP zu definieren: NP besteht, grob gesprochen, aus allen algorithmischen Probleme, die von nichtdeterministischen Rechnern effizient gelöst werden können, indem kurz geraten und schnell verifiziert wird. Die überraschende Erkenntnis war, dass viele interessante Probleme in der Klasse NP tatsächlich zu den schwierigsten Probleme in NP gehören. Diese schwierigsten Probleme wurden dann NP-vollständig genannt.

Was helfen uns diese Überlegungen? Wir werden die folgende zentrale Beobachtung zeigen:

> Wenn irgendein NP-vollständiges Problem eine (konventionelle, also deterministische) effiziente Berechnung besitzt, dann können alle Probleme in NP effizient gelöst werden.

Ist denn eine effiziente Berechnung aller Probleme in NP zu erwarten? Keinesfalls, denn dies bedeutet intuitiv, dass unsere heutigen Rechner bereits die Fähigkeit des Ratens haben. Die effiziente Berechnung irgendeines NP-vollständigen Problems würde eine Revolution der Informatik nach sich ziehen.

Und was bedeutet dies für den Algorithmenentwurf? Als Teil des Entwurfsprozesses muss man sich zuallererst fragen, ob das zu lösende algorithmische Problem überhaupt mit sinnvollen Rechnerressourcen gelöst werden kann. Ist das Problem aber NP-vollständig, dann sollte man keine Entwicklungszeit in der vergeblichen Jagd nach effizienten Algorithmen vergeuden und stattdessen nach Problemaufweichungen fragen, die effiziente Algorithmen erlauben. In Teil III des Skripts werden wir zum Beispiel approximative Lösungen von Optimierungsproblemen als Problemaufweichung untersuchen.

Bemerkung 5.1 An dieser Stelle mag der Eindruck auftreten, dass alle Probleme mit nichtdeterministischen Rechnern effizient gelöst werden können. Das ist ganz und gar nicht der Fall. Schon eine einfache Modifikation des Erfüllbarkeitsproblem stellt nichtdeterministische Rechner vor schier unüberwindbare Schwierigkeiten: Betrachte diesmal aussagenlogische Formeln der Form

$$\beta \equiv \ \forall y_1, \ldots, \forall y_m \ \alpha(y_1, \ldots, y_m),$$

in denen die Variablen $y_1, \ldots, y_m$ durch den Allquantor gebunden sind. Wir möchten feststellen, ob β wahr ist.

Offensichtlich ist β genau dann wahr, wenn $\alpha(y_1, \ldots, y_m)$ eine Tautologie ist, wenn also $\neg\alpha(y_1, \ldots, y_m)$ *nicht* erfüllbar ist. Das Problem festzustellen, ob die Formel β wahr ist, ist also für konventionelle Rechner genauso so schwierig wie das Erfüllbarkeitsproblem, für nichtdeterministische Rechner hingegen ist das Erfüllbarkeitsproblem trivial, die Überprüfung der Wahrheit von β hingegen extrem schwer!

Um das skizzierte Vorgehen durchführen zu können, müssen wir die folgenden Begriffe und Fragen klären:

- Wie sieht ein Rechnermodell aus, das nicht nur die Kraft heute hergestellter Rechner beschreibt, sondern auch die Kraft aller zukünftigen Rechnergenerationen?
- Wann sollten wir eine Berechnung effizient nennen?
- Was ist ein nichtdeterministischer Rechner?
- Wie sollten wir NP-Vollständigkeit definieren? Gibt es überhaupt schwierigste, also NP-vollständige Probleme?

Wir beginnen mit einer Antwort auf die beiden ersten Fragen.

5.1 Turingmaschinen und die Klasse P

Was genau ist ein Rechner? Betrachten wir die wesentlichen Anforderungen an ein vernünftiges Rechnermodell:

> Ein Rechner muss auf die Eingabe zugreifen können und Rechnungen auf einem unbeschränkt großen Speicher ausführen können.

Wir formalisieren ein gewissermaßen einfachstes Rechnermodell mit Hilfe des Begriffs der **Turingmaschine**.

Eine Turingmaschine besitzt ein nach links und nach rechts unendliches, lineares Band, das in Zellen unterteilt ist; die Zellen besitzen Zahlen aus $\mathbb{Z}$ als Adressen. Anfänglich ist die Eingabe $w = w_1 \cdots w_n$ in den Zellen mit den Adressen $1 \ldots, n$ abgelegt, wobei Zelle i (mit $1 \leq i \leq n$) den Buchstaben w_i speichert; alle verbleibenden Zellen speichern das **Blanksymbol** $\mathtt{B}$. Die Turingmaschine manipuliert ihr Band mit Hilfe eines **Lese-/Schreibkopfes**, der sich zu Beginn der Rechnung auf der Zelle 1 befindet. Während zu Anfang also alle Buchstaben entweder aus dem **Eingabealphabet** Σ stammen oder mit dem Blanksymbol übereinstimmen, darf die Turingmaschine im Verlauf der Rechnung Zellen mit Buchstaben eines **Arbeitsalphabets** Γ mit $\Sigma \cup \{\mathtt{B}\} \subseteq \Gamma$ beschriften.

Wie rechnet eine Turingmaschine? Das Programm einer Turingmaschine wird durch eine partiell definierte **Zustandsüberführungsfunktion** δ mit

$$\delta \colon Q \times \Gamma \to Q \times \Gamma \times \{\mathtt{links}, \mathtt{bleib}, \mathtt{rechts}\}$$

beschrieben. Die endliche Menge Q heißt Zustandsmenge, wobei wir annehmen, dass die Maschine im Anfangszustand $q_0 \in Q$ startet. Wenn die Maschine sich im Zustand $q \in Q$ befindet und den Buchstaben $\gamma \in \Gamma$ liest, dann ist der „Befehl“ $\delta(q, \gamma)$ anzuwenden. Wenn

$$\delta(q, \gamma) = (q', \gamma', \mathtt{Richtung}),$$

dann wird γ' gedruckt (und damit wird γ überdruckt), die Maschine wechselt in den Zustand q' und der Kopf wechselt zur linken Nachbarzelle ($\mathtt{Richtung} = \mathtt{links}$), zur rechten Nachbarzelle ($\mathtt{Richtung} = \mathtt{rechts}$) oder bleibt stehen ($\mathtt{Richtung} = \mathtt{bleib}$).

Wir sagen, dass die Maschine im Zustand q hält, wenn sich die Maschine im Zustand $q \in Q$ befindet, den Buchstaben $\gamma \in \Gamma$ liest und wenn δ auf (q, γ) nicht definiert ist. Schließlich zeichnen wir eine Teilmenge $F \subseteq Q$ als **Menge der akzeptierenden Zustände** aus und sagen, dass die Maschine die Eingabe w akzeptiert, wenn sie in einem Zustand aus F hält.

Zusammengefaßt: Eine Turingmaschine wird durch das 6-Tupel $(Q, \Sigma, \delta, q_0, \Gamma, F)$ beschrieben. Wir sagen, dass die Turingmaschine $M = (Q, \Sigma, \delta, q_0, \Gamma, F)$ das Entscheidungsproblem

$$L(M) = \{w \mid M \text{ akzeptiert } w\}$$

löst. Schließlich messen wir die Laufzeit und definieren $\mathtt{schritte}_M(x)$ als die Anzahl der Schritte von M auf Eingabe $x \in \Sigma^*$. Die worst-case Laufzeit von M auf Eingaben der Länge n ist dann

$$\mathtt{Zeit}_M(n) = \max\{\mathtt{schritte}_M(x) \mid x \in \Sigma^n\}.$$

Ist uns eine wirklich überzeugende Definition eines Rechnermodells gelungen? Der erste Eindruck ist negativ, denn eine Turingmaschine erinnert eher an eine Nähmaschine als an die Modellierung eines modernen Rechners. Tatsächlich werden wir im Folgenden aber sehen, dass Nähmaschinen modernste Rechner und sogar Parallelrechner ohne Probleme simulieren können, solange die Rechenzeit der Nähmaschine **polynomiell** größer sein darf als die Rechenzeit des zu simulierende Rechners: So zeigen wir zum Beispiel im nächsten Abschnitt, dass eine „moderne“ Registermaschine, die in Zeit $t(n)$ rechnet, in Zeit $O(t^6(n))$ simuliert werden kann.

Dieses Resultat heißt sicherlich nicht, dass Intel oder AMD in Zukunft Nähmaschinen verkaufen sollten, denn der Geschwindigkeitsabfall von Laufzeit $t(n)$ auf Laufzeit $O(t(n)^6)$ ist gewaltig. Wenn wir uns aber nur für Probleme interessieren, die in polynomieller Rechenzeit lösbar sind, dann können wir die entsprechende Problemklasse durch Supercomputer oder durch Nähmaschinen definieren ohne die Klasse zu ändern und genau das tun wir in der Definition der Klasse P aller effizient lösbaren Probleme.

Definition 5.1 *Die Klasse* P *besteht aus allen Entscheidungsproblemen L für die es eine Turingmaschine M mit $L = L(M)$ gibt, so dass $\texttt{Zeit}_M(n) = O((n+1)^k)$ für eine Konstante k gilt.*

Wir sagen, dass L effizient berechenbar ist, wenn $L \in$ P gilt.

Beispiel 5.1 Die folgenden Entscheidungsprobleme gehören zur Klasse P. (Wir nehmen jeweils eine „geeignete“ Darstellung der Eingabe an; zum Beispiel repräsentieren wir Graphen durch ihre Adjazenzmatrix, die wir durch Hintereinanderschreiben ihrer Zeilen als ein binäres Wort auffassen können.)

- Reguläre und kontextfreie Sprachen. Diese fundamentalen Sprachklassen werden in den Vorlesungen „Diskrete Modellierung“ sowie „Theoretische Informatik 2“ behandelt.

- Sämtliche bisher behandelten Graphprobleme, wie etwa das Zusammenhangsproblem $Zh = \Big\{G \,|\, G \text{ ist ein ungerichteter, zusammenhängender Graph}\Big\}$ gehören zu P. Warum? In der Vorlesung „Datenstrukturen“ wird zum Beispiel die Tiefensuche benutzt, um den Zusammenhang eines Graphen zu überprüfen: Der Graph ist genau dann zusammenhängend, wenn ein einfacher Aufruf der Tiefensuche alle Knoten besucht.

- $Match = \Big\{(G,k) \,|\, G \text{ besitzt } k \text{ Kanten, die keinen gemeinsamen Endpunkt haben}\Big\}$

  ist ein weiteres Beispiel eines Entscheidungsproblems in P. Das Matching-Problem ist wesentlicher komplexer als das Zusammenhangsproblem und kann zum Beispiel mit Methoden der linearen Programmierung gelöst werden (siehe Kapitel 4.4). Aber auch das Problem der linearen Programmierung, also die Lösung der linearen Optimierungsaufgabe

  $$\min c^T \cdot x \text{ so dass } A \cdot x \geq b, x \geq 0,$$

  gelingt in polynomieller Zeit und moderne Algorithmen beherrschen Systeme mit mehreren Tausenden von Variablen. (Die benötigte Zeit ist polynomiell in der Länge der Binärdarstellung für die Matrix A und die Vektoren b, c.)

- Viele arithmetische Probleme wie

  $$\Big\{n \,|\, n \text{ ist eine Quadratzahl}\Big\} \text{ oder } \Big\{(k,n) \,|\, k \text{ teilt } n\Big\}$$

  gehören zur Klasse P. Nach langer Forschungsarbeit ist sogar gezeigt worden, dass Primzahlen effizient erkannt werden können, das Entscheidungsproblem

  $$\{n \mid n \text{ ist eine Primzahl}\}$$

  gehört also auch zu P. (Man beachte, dass wir in all diesen Fällen annehmen, dass die Zahl n in Binärdarstellung gegeben ist. Polynomielle Laufzeit bedeutet also eine Laufzeit der Form $\text{poly}(\log_2 n)$.) Das Faktorisierungsproblem

Bestimme die Teiler einer Zahl n

ist hingegen in aller Wahrscheinlichkeit nicht effizient lösbar. Das Faktorisierungsproblem nimmt eine Sonderstellung ein, weil es wahrscheinlich nicht mit NP-vollständigen Problemen in Zusammenhang gebracht werden kann. Die vermutete Schwierigkeit des Faktorisierungsproblems ist die Grundlage vieler Public-Key Kryptosysteme.

Wir betrachten nur Entscheidungsprobleme und können damit anscheinend keine Aussagen über Optimierungsprobleme machen. Dem ist nicht so, wie die folgende Aufgabe zeigt.

Aufgabe 47
Wir betrachten in dieser Aufgabe drei verschiedene Varianten des Problems *SET PACKING SP*: Sei $C = \{C_1, \ldots, C_m\}$ eine Menge endlicher Mengen. Untersucht werden disjunkte Teilmengen aus C.

Variante 1: Enthält C mindestens k disjunkte Teilmengen? Hierbei ist die natürliche Zahl k Teil der Eingabe.

Variante 2: Berechne die maximale Anzahl disjunkter Teilmengen aus C.

Variante 3: Gib eine größtmögliche Menge von disjunkten Teilmengen aus C aus.

Die erste Variante heißt auch Entscheidungsvariante. Die zweite Variante fragt nach dem Wert einer optimalen Lösung. Die dritte Variante erfordert die Berechnung einer optimalen Lösung.

(a) Angenommen Variante 1 ist in P. Ist dann auch Variante 2 deterministisch in Polynomialzeit lösbar? **Begründe** deine Antwort.

(b) Angenommen Variante 1 ist in P. Ist dann auch Variante 3 deterministisch in Polynomialzeit lösbar? **Begründe** deine Antwort.

Wir untersuchen die Berechnungskraft von Turingmaschinen im nächsten Abschnitt. Nichtdeterministische Turingmaschinen und die Klasse NP werden im Kapitel 5.2 eingeführt, während die NP-Vollständigkeit im Kapitel 5.3 definiert wird.

5.1.1 Die Berechnungskraft von Turingmaschinen*

Aufgabe 48
Gegeben sei die folgende Turingmaschine:

$Q = \{q_0, q_1, q_2\}$, Eingabealphabet $\Sigma = \{1\}$, Startzustand q_0, Bandalphabet $\Gamma = \{1, \mathtt{B}\}$ und $F = \{q_2\}$. Das Symbol $\perp$ stehe für „undefiniert“. Das Programm δ sei

$$\begin{array}{lcl} \delta(q_0, \mathtt{B}) & = & \perp \\ \delta(q_0, 1) & = & (q_1, 1, \mathtt{rechts}) \\ \delta(q_1, \mathtt{B}) & = & \perp \\ \delta(q_1, 1) & = & (q_2, 1, \mathtt{rechts}) \\ \delta(q_2, \mathtt{B}) & = & \perp \\ \delta(q_2, 1) & = & (q_1, 1, \mathtt{rechts}) \end{array}$$

Welches Entscheidungsproblem wird durch die Turingmaschine gelöst?
Stoppt die Turingmaschine immer?

Beispiel 5.2 Das Entscheidungsproblem $L = \{a^n\ b^n \mid n \in \mathbb{N}\}$ ist zu lösen. Das Eingabealphabet ist $\Sigma = \{a, b\}$. Wir wählen $\Gamma = \Sigma \cup \{\mathtt{B}\}$ als Bandalphabet. Beachte, dass unsere Turingmaschine die a's nicht mitzählen kann: Da unsere Zustandsmenge endlich sein muss, können wir nicht für jede mögliche Anzahl der a's einen Zustand vergeben.

Unsere Idee ist es, immer abwechselnd vom Anfang des Wortes ein a und dann vom Ende des Wortes ein b zu löschen. Landen wir mit diesem Verfahren beim leeren Wort, so war das ursprüngliche Wort aus L. Bleibt ein Überschuß von einem der beiden Buchstaben oder finden

wir einen Buchstaben an einer Stelle, wo er nicht auftauchen darf, so gehört das Ausgangswort nicht zu L.

Der Anfangszustand ist q_0. Wir dürfen annehmen, dass der Lese-Schreibkopf auf dem ersten (linkesten) Element der Eingabe steht. Finden wir hier ein B, so haben wir es mit $a^0b^0 \in L$ zu tun. In diesem Fall springen wir in den akzeptierenden Zustand q_{ja}. Finden wir ein b, so ist das ein Grund, das Wort zu verwerfen. Dafür richten wir den (verwerfenden) Zustand q_{nein} ein. Finden wir ein a, so löschen wir es. Sodann wechseln wir in den Zustand q_{rechts_a} und bewegen uns nach rechts.

$$\begin{aligned}\delta(q_0, \mathtt{B}) &= (q_{ja}, \mathtt{B}, \mathtt{bleib})\\ \delta(q_0, a) &= (q_{rechts\_a}, \mathtt{B}, \mathtt{rechts})\\ \delta(q_0, b) &= (q_{nein}, b, \mathtt{bleib})\end{aligned}$$

Die Zustände q_{ja} und q_{nein} sollen *Endzustände* werden. Das heißt, sie werden nicht mehr verlassen. Demzufolge sehen die betreffenden Teile der Übergangsfunktion so aus:

$$\begin{aligned}\delta(q_{ja}, x) &= \bot \qquad \forall x \in \{a, b, \mathtt{B}\}\\ \delta(q_{nein}, x) &= \bot \qquad \forall x \in \{a, b, \mathtt{B}\},\end{aligned}$$

wobei $\bot$ wieder andeutet, dass die Übergangsfunktion undefiert ist, M hält somit in dem akzeptierenden Zustand q_{ja} wie auch in dem verwerfenden Zustand q_{nein}.

Die Maschine bleibt im Zustand q_{rechts_a}, solange a's gelesen werden und wechselt beim ersten b in den Zustand q_{rechts_b}. Stoßen wir auf das Blanksymbol, solange wir noch im Zustand q_{rechts_a} sind, so liegt ein Restbestand von as vor, dem keine bs mehr entsprechen. Wir wechseln in den Zustand q_{nein}.

$$\begin{aligned}\delta(q_{rechts\_a}, \mathtt{B}) &= (q_{nein}, \mathtt{B}, \mathtt{bleib})\\ \delta(q_{rechts\_a}, a) &= (q_{rechts\_a}, a, \mathtt{rechts})\\ \delta(q_{rechts\_a}, b) &= (q_{rechts\_b}, b, \mathtt{rechts})\end{aligned}$$

Im Zustand q_{rechts_b} wird bis zum ersten Blanksymbol gelesen. Das Auffinden eines a's ist ein Fehler:

$$\begin{aligned}\delta(q_{rechts\_b}, \mathtt{B}) &= (q_{zu\_löschen}, \mathtt{B}, \mathtt{links})\\ \delta(q_{rechts\_b}, a) &= (q_{nein}, a, \mathtt{bleib})\\ \delta(q_{rechts\_b}, b) &= (q_{rechts\_b}, b, \mathtt{rechts})\end{aligned}$$

Im Zustand $q_{zu\_löschen}$ wird das erste b (vom rechten Bandende) gelöscht:

$$\delta(q_{zu\_löschen}, b) = (q_{rücklauf}, \mathtt{B}, \mathtt{links})$$

(Beachte, dass im Zustand $q_{zu\_löschen}$ stets der Buchstabe B gelesen wird. Wir geben deshalb keine Befehle für die Buchstaben a und b an). Wir haben nun je einen Buchstaben vom Anfang und vom Ende des Wortes entfernt. Unser Ausgangswort gehört genau dann zu L, wenn der aktuelle Bandinhalt zu L gehört. Wir laufen also zurück zum Anfang des Wortes und gehen wieder in den Zustand q_0.

$$\begin{aligned}\delta(q_{rücklauf}, \mathtt{B}) &= (q_0, \mathtt{B}, \mathtt{rechts})\\ \delta(q_{rücklauf}, a) &= (q_{rücklauf}, a, \mathtt{links})\\ \delta(q_{rücklauf}, b) &= (q_{rücklauf}, b, \mathtt{links})\end{aligned}$$

Zusammengefaßt: mit der Zustandsmenge

$$Q \;=\; \Big\{q_0,\, q_{rechts\_a},\, q_{rechts\_b},\, q_{rücklauf},\, q_{zu\_löschen},\, q_{ja},\, q_{nein}\Big\},$$

Eingabealphabet $\Sigma = \{a, b\}$, Programm δ, Startzustand q_0, Bandalphabet $\Gamma = \{a, b, \mathtt{B}\}$ und $F = q_{ja}$, haben wir die Turingmaschine

$$M = (Q, \Sigma, \delta, q_0, \Gamma, F)$$

konstruiert mit

$$L(M) = \{a^n\, b^n \,|\, n \in \mathbb{N}\}.$$

Wir haben einen ersten *Programmiertrick* kennengelernt, nämlich die Benutzung von Zuständen zur Speicherung: Zum Beispiel „erinnert“ sich der Zustand q_{rechts_a} daran, dass bisher nur a's gelesen wurden. Wesentlich ist, dass die Menge der Informationen, die sich die Maschine in den Zuständen merkt, beschränkt ist. Zu keinem Zeitpunkt „weiß“ die Maschine, wie lang das Wort ist, wie oft der Buchstabe a darin enthalten ist oder ähnliches.

Im nächsten Beispiel lernen wir einen weiteren Trick kennen, die Bildung von Spuren.

Beispiel 5.3 Die Kopierfunktion

$$f\colon \Sigma^* \to \Sigma^* \circ \{\diamond\} \circ \Sigma^*$$

mit

$$f(w) = w \diamond w$$

ist zu berechnen. Wir verzichten hier auf die genaueren Details der Programmierung und skizzieren hier nur das Vorgehen einer Turingmaschine M. Durch Unterstreichung drücken wir die aktuelle Position des Lese/Schreibkopfs aus. M beginnt im Startzustand `Drucke-Kasten`, läuft bis zum Ende der Eingabe und druckt einen Kasten

$$\mathtt{B}\underline{w_1} \cdots w_n\mathtt{B} \quad\longrightarrow\quad \mathtt{B}w_1 \cdots \underline{w_n} \diamond .$$

Nachdem der Kasten gedruckt wurde, läuft die Maschine M im Zustand `Zurück- vom-Kasten` zurück bis zum ersten Blanksymbol

$$\mathtt{B}w_1 \cdots \underline{w_n}\diamond \quad\longrightarrow\quad \underline{\mathtt{B}}w_1 \cdots w_n \diamond .$$

M wechselt in den Zustand `*-Drucken` und legt eine zweite Spur an, indem w_1 durch $\binom{w_1}{*}$ ersetzt wird. Dabei steht $\binom{w_1}{*}$ für ein Zeichen, welches nicht im Eingabealphabet enthalten ist und dem Zeichen w_1 eindeutig zuzuordnen ist.

$$\underline{\mathtt{B}}w_1 \cdots w_n\diamond \quad\longrightarrow\quad \underline{\mathtt{B}\binom{w_1}{*}}w_2 \cdots w_n \diamond .$$

Für jeden Buchstaben $a \in \Sigma$ verfügt M über einen Rechtslauf-Zustand `rechts_a`. In dem w_1 entsprechenden Zustand läuft M nach rechts bis zum ersten Blanksymbol und druckt, dort angekommen, w_1.

$$\underline{\mathtt{B}\binom{w_1}{*}}w_2 \cdots w_n\diamond \quad\longrightarrow\quad \mathtt{B}\binom{w_1}{*}w_2 \cdots w_n \diamond \underline{w_1}.$$

Dann wechselt M in den Zustand `Zurück-vom-Kopieren` und läuft bis zum markierten Buchstaben. (Zu diesem Zeitpunkt ist es M schon wieder gleichgültig, was sie soeben gedruckt hat.)

$$\mathtt{B}\begin{pmatrix} w_1 \\ * \end{pmatrix} w_2 \cdots w_n \diamond \underline{w_1}\mathtt{B} \quad \longrightarrow \quad \underline{\mathtt{B}\begin{pmatrix} w_1 \\ * \end{pmatrix}} w_2 \cdots w_n \diamond w_1\mathtt{B}.$$

Die Markierung wird entfernt. M bewegt den Lese-Schreibkopf auf die nächste Position w_2 und wechselt in den Zustand `*-Drucken`. Von jetzt ab wiederholt M ihre Vorgehensweise, bis im Zustand `*-Drucken` der Kasten gelesen wird. In diesem Fall wechselt M in den Zustand `Fast-Fertig`, läuft nach links bis zum ersten Blank, wo es zur rechten Nachbarzelle im Zustand `Fertig` wechselt und dann hält. Also arbeitet M mit dem Eingabealphabet $\Sigma = \{\sigma_1, \sigma_2, ...\}$, der Zustandsmenge

$$\begin{aligned} Q \quad = \quad & \{q_{Drucke-Kasten},\, q_{Zurück-vom-Kasten},\, q_{*-Drucken},\, q_{Zurück-vom-Kopieren}, \\ & q_{Fast-Fertig},\, q_{Fertig},\, q_{rechts\_\sigma_1},\, q_{rechts\_\sigma_2}, \ldots\}, \end{aligned}$$

Startzustand `Drucke-Kasten` und dem Bandalphabet

$$\Gamma = \left\{ \mathtt{B}, \sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\diamond, \begin{pmatrix} \sigma_1 \\ * \end{pmatrix}, \begin{pmatrix} \sigma_2 \\ * \end{pmatrix} \ldots, \begin{pmatrix} \sigma_{|\Sigma|} \\ * \end{pmatrix} \right\}.$$

Unsere Turingmaschine M erfüllt $\text{Zeit}_M(n) = \Theta\left(n^2\right)$. Warum? M kopiert die n Buchstaben der Eingabe, indem es jeden der n Buchstaben von w über eine Strecke von n Zellen transportiert.

Aufgabe 49
Gegeben ist das Entscheidungsproblem P aller Palindrome (über dem Alphabet $\{0,1\}$); also

$$P = \{w | w \in \{0,1\}^* \text{ und } w = w^R\}$$

wobei $w^R = w_n w_{n-1} \cdots w_2 w_1$, falls $w = w_1 w_2 \cdots w_{n-1} w_n$.

Beschreibe explizit eine Turingmaschine, die genau das Entscheidungsproblem P löst.

Aufgabe 50
Die Speicherzellen einer (konventionellen) Turingmaschine sind auf einem *beidseitig* unendlichen Band angeordnet. In dieser Aufgabe betrachten wir ein anscheinend stark eingeschränktes Modell, nämlich Turingmaschinen mit einem nur nach rechts unendlichen Band.

Einige Kommentare zu diesem eingeschränkten Modell. Wir denken uns die Speicherzellen mit den Adressen $1, 2, 3 \ldots$ nummeriert und nehmen wie üblich an, dass die Eingabe in den Speicherzellen $1, 2, \ldots, n$ erscheint. Wir definieren, dass eine Berechnung einer eingeschränkten Turingmaschine *scheitert*, falls der Kopf versucht Zelle 0 zu lesen.

Gegeben sei eine (konventionelle) Turingmaschine M, die für Eingabealphabet Σ durch das Bandalphabet Γ, die Zustandsmenge Q, den Anfangszustand q_0, der Menge F von akzeptierenden Zuständen sowie durch das Programm δ beschrieben ist. **Beschreibe** eine eingeschränkte Turingmaschine M^*, die M simuliert. Insbesondere, **beschreibe** Bandalphabet, Zustandsmenge und Übergangsfunktion.

Aufgabe 51
Beschreibe explizit eine Einband Turingmaschine, die für Eingabe $w = (w_1, w_2, \ldots, w_{n-1}, w_n) \in \{0,1\}^n$ die Ausgabe $(w_1, w_1, w_2, w_2, \ldots, w_{n-1}, w_{n-1}, w_n, w_n)$ liefert. D.h. nach Terminierung der Berechnung soll für $i \leq n$ in Zellen $2i-1$ und $2i$ des Bandes der Wert w_i gespeichert sein. **Gib** in deiner Beschreibung das Bandalphabet Γ, die Zustandsmenge Q, Anfangszustand q_0 und die Menge F der Endzustände, sowie das Programm δ an. Deine Maschine sollte asymptotisch möglichst schnell arbeiten.

Bestimme die Laufzeit deiner Maschine in θ-Notation. (Die Laufzeit ist natürlich eine Funktion von n, der Anzahl der eingegebenen Buchstaben.)

Wir müssen wie angekündigt nachweisen, dass unsere Definition von P weitgehend technologieunabhängig ist. Deshalb werden wir das anscheinend mächtigere Modell der Registermaschinen, eine Modellierung der modernen von Neumann Rechner, betrachten und feststellen, dass dieses Maschinenmodell durch Turingmaschinen sogar effizient simuliert werden kann. Mit anderen Worten: Der Sprung von der Nähmaschine zum Supercomputer führt nur zu einer polynomiellen Leistungssteigerung!

Wir betrachten zuerst k-Band-Turingmaschinen. Als Neuerung haben wir jetzt eine Architektur von k Bändern, wobei die Eingabe auf Band 1 gespeichert wird.

Jedes Band besitzt einen eigenen Lese-/Schreibkopf. Ein Befehl wird jetzt in Abhängigkeit vom gegenwärtigen Zustand *und* den Inhalten der k gelesenen Zellen ausgewählt. Der ausgewählte Befehl legt für jeden Kopf den zu druckenden Buchstaben fest wie auch die „Marschrichtung“ `links`, `bleib` oder `rechts`. Ansonsten verhalten sich k-Band Maschinen analog zu Turingmaschinen.

Satz 5.2 *Sei M eine k-Band-Maschine, die auf Eingaben der Länge n nach höchstens $T(n)$ Schritten hält. Dann gibt es eine Turingmaschine M', so dass*

- *M' auf Eingaben der Länge n nach höchstens $O\left(T^2(n)\right)$ Schritten hält und*
- *M und M' dieselbe Funktion berechnen bzw. dasselbe Entscheidungsproblem lösen.*

Beweis: Wir „bauen“ eine Turingmaschine M', die die k *Bänder* von M durch k *Spuren* simuliert. Die k Spuren entsprechen den k Bändern, wobei wir in den Spuren auch die Position des jeweiligen Lese/Schreibkopfs vermerken müssen. Wir wählen deshalb für das Bandalphabet Γ von M das neue Bandalphabet

$$\Gamma' = (\Gamma \times \{*, \mathtt{B}\})^k \cup \Gamma.$$

Man mache sich klar, dass bei konstantem k und endlichem Bandalphabet Γ auch dieses Bandalphabet Γ' endlich ist.

Liest nun beispielsweise unsere Turingmaschine bei der Simulation einer 4-Band Maschine in einer Zelle das Zeichen

$$[(a, *), (b, \mathtt{B}), (a, \mathtt{B}), (\mathtt{B}, *)],$$

so bedeutet das, dass in dieser Zelle auf den 4 Bändern die Zeichen $a, b, a, \mathtt{B}$ stehen und, dass die Schreibköpfe der Bänder 1 und 4 zu diesem Zeitpunkt auf unserer Zelle stehen.

Anfangsphase von M': M' ersetzt, für Eingabe $w_1, \ldots, w_n$, den Inhalt der ersten n Zellen durch

$$[(w_1, *),\ (\mathtt{B}, *), \ldots,\ (\mathtt{B}, *)], \ldots, [(w_n, \mathtt{B}),\ (\mathtt{B}, \mathtt{B}), \ldots,\ (\mathtt{B}, \mathtt{B})].$$

Das entspricht unserer beabsichtigten Interpretation. Die Bänder 2 bis k sind leer. Auf jedem Band steht der Lese/Schreibkopf an der ersten Position. Band 1 enthält das Eingabewort.

Wir haben nun zu zeigen, dass wir einen Rechenschritt der k-Band-Maschine M mit M' so simulieren können, dass diese Interpretation auch für die Folgeschritte richtig bleibt. Dies geschieht mittels einer Induktion über die Zahl der bereits erfolgten Berechnungsschritte von M. Nach 0 Schritten stimmt unsere Darstellung, wie wir uns eben klar gemacht haben.

Simulation von Schritt $t+1$ von M: Wir nehmen an, dass M nach Schritt t den Bandinhalt

$$\begin{aligned} &a_1 \cdots \underline{a_i} \cdots a_j \cdots a_l \cdots \text{ Band 1} \\ &b_1 \cdots b_i \cdots \underline{b_j} \cdots b_l \cdots \text{ Band 2} \\ &c_1 \cdots c_i \cdots c_j \cdots \underline{c_l} \cdots \text{ Band } k \end{aligned}$$

besitzt. (Die unterstrichenen Buchstaben kennzeichnen die Position des jeweiligen Kopfes.) Wir nehmen weiterhin an, dass M' den Bandinhalt

$$[(a_1, \mathtt{B}),\ (b_1, \mathtt{B}),\ \ldots,\ (c_1, \mathtt{B})],\ \ldots,\ [(a_i, *),\ (b_i, \mathtt{B}),\ \ldots,\ (c_i, \mathtt{B})],\ \ldots,$$
$$[(a_j, \mathtt{B}),\ (b_j, *),\ \ldots,\ (c_j, \mathtt{B})],\ \ldots,\ [(a_l, \mathtt{B}),\ (b_l, \mathtt{B}),\ \ldots,\ (c_l, *)],\ \ldots$$

besitzt und den gegenwärtigen Zustand von M kennt. Der Kopf von M' steht am linken Ende seines Bandes.

Zuerst durchläuft M' das Band von links nach rechts. M' merkt sich die k von M gelesenen Buchstaben in seinen Zuständen. Damit meinen wir, dass M' zu diesem Zweck über $(|\Gamma|+1)^k$ viele zusätzliche Zustände verfügt. Ist das Bandalphabet etwa $\{a, b\}$ und $k = 3$ so könnten diese

$$q_{???}, q_{??a}, q_{??b}, q_{?a?}, q_{?b?} \ldots, q_{aaa}, q_{bbb}$$

heißen. Beginnend in $q_{???}$ läuft M' von links nach rechts übers Band. Erreicht sie eine Stelle, an der der Lese/Schreibkopf eines Bandes von M steht, so liest sie das entsprechende Zeichen und wechselt in den entsprechenden Zustand. Findet sie beispielsweise als erstes den Kopf von Band 2, und liest dieser ein a, so würde M' in den Zustand $q_{?a?}$ wechseln und die Suche fortsetzen.

Wenn alle Lese/Schreibköpfe gefunden sind, kennt M' durch den Zustand das von M gelesene k-Tupel und kennt damit den auszuführenden Befehl. M' läßt seinen Kopf nach links zurücklaufen, überdruckt alle durch einen $*$ markierten Zellen und versetzt die Markierungen der Leseköpfe wie gefordert. (Auch für diesen Schritt müssen zusätzlich Zustände bereitgestellt werden.)

Wenn M das linke Ende seines Bandes erreicht hat, kann der nächste Simulationsschritt beginnen. Unsere Invariante ist erhalten geblieben.

Angenommen, M benötigt t Schritte für Eingabe w. Dann wird M auf jedem Band nur Zellen mit den Adressen $-t$, …, 0, 1, …, t erreichen können. Die Simulation eines Schrittes von M gelingt also in Zeit höchstens $O(t)$. Also benötigt M' insgesamt nur höchstens $O((2t+1)\cdot t) = O\left(t^2\right)$ Schritte, und die Behauptung folgt. ■

Aufgabe 52
Wir betrachten das Modell der zweidimensionalen Turingmaschinen. Eine Maschine M dieses Modells besitzt ein zweidimensionales Speicherband, also ein unendliches Schachbrett. Am Anfang der Berechnung steht der Lese-/Schreibkopf auf der Zelle $(0, 0)$. Die möglichen Kopfbewegungen von M in einem Rechenschritt sind rechts (von Position (i, j) nach $(i + 1, j)$), oben (von Position (i, j) nach $(i, j + 1)$), links (von Position (i, j) nach $(i - 1, j)$) und unten (von Position (i, j) nach $(i, j - 1)$).

Zeige: Jede zweidimensionale Turingmaschine M kann durch eine „gewöhnliche“ Turingmaschine *effizient* simuliert werden.
(*Effizient* bedeutet hierbei: Falls M auf Eingaben der Länge n nach höchstens $T(n)$ Schritten hält, so benötigt die simulierende Turingmaschine höchstens $O(T^k(n))$ Schritte, für eine Konstante k.)

Anmerkung: Auf den Ergebnissen dieser Aufgabe aufbauend läßt sich auch das parallele Rechnermodell der zweidimensionalen zellulären Automaten durch „gewöhnliche“ Turingmaschinen simulieren.

Ein zellulärer Automat besteht aus identisch programmierten Prozessoren (auf einem unendlichen Schachbrett angeordnet), die gemäß einer Zustandsüberführungsfunktion

$$\delta : Q \times Q^4 \to Q$$

arbeiten. Alle Prozessoren wechseln ihre Zustände parallel. Insbesondere wechselt ein Prozessor mit gegenwärtigem Zustand $q \in Q$ in einen Zustand $q' \in Q$ falls

$$\delta(q, q_{\mathsf{links}}, q_{\mathsf{rechts}}, q_{\mathsf{oben}}, q_{\mathsf{unten}}) = q'$$

gilt, für die Zustände $q_{\mathsf{links}}, q_{\mathsf{rechts}}, q_{\mathsf{oben}}$ bzw. q_{unten} seines linken, rechten, oberen bzw. unteren Nachbarn. Vielleicht kennt ihr ja das *Spiel des Lebens*, welches mit zellulären Automaten gespielt wird.

Aber auch k-Band Maschinen sind alles andere als Modelle moderner Rechner. Deshalb wenden wir uns jetzt den Registermaschinen zu, einem modernen Modell sequentieller Rechner.

Definition 5.3 *(a) Die Architektur einer Registermaschine besteht aus einem* Eingabeband*, dem* Speicher *und einem* Ausgabeband*. Das Eingabe- wie auch das Ausgabeband ist in Zellen unterteilt, wobei eine Zelle eine natürliche Zahl speichern kann. Das Eingabeband (bzw. das Ausgabeband) besitzt einen Lesekopf (bzw. einen Schreibkopf), der die Zellen des Bandes von links nach rechts lesen (bzw. beschreiben) kann.*

Der Speicher besteht aus dem Akkumulator*, der arithmetische und logische Befehle ausführen kann, sowie aus einer unendlichen Anzahl von* Registern*. Der Akkumulator kann, wie auch die Register, natürliche Zahlen speichern. Der Akkumulator erhält die Adresse 0, das i-te Register die Adresse i. Wir bezeichnen den Inhalt des Akkumulators mit $c(0)$ und den Inhalt des i-ten Registers mit $c(i)$.*

(b) Jeder Befehl wird durch ein Label identifiziert. Eine Registermaschine besitzt Lese-/Schreibbefehle, Speicherbefehle, Vergleichsoperationen, arithmetische Befehle *und* Sprung-Befehle*:*

`READ` *und* `WRITE` *sind die Lese-/Schreibbefehle. Der Befehl* `READ` *liest die gegenwärtige gelesene Zahl in den Akkumulator und bewegt den Lesekopf um eine Zelle nach rechts. Der Befehl* `WRITE` *schreibt den Inhalt des Akkumulators auf das Ausgabeband und bewegt den Schreibkopf um eine Zelle nach rechts.*

Wir unterscheiden verschiedene Typen von Speicherbefehlen.

$$\begin{aligned} \texttt{LOAD}\, i &\qquad (c(0) = c(i)) \\ \texttt{INDLOAD}\, i &\qquad (c(0) = c(c(i))) \\ \texttt{KLOAD}\, i &\qquad (c(0) = i) \end{aligned}$$

weisen dem Akkumulator direkt oder indirekt Registerinhalte zu, beziehungsweise erlauben eine Initialisierung durch Konstanten.

Der Akkumulator-Wert kann abgespeichert werden mit Hilfe von

$$\texttt{STORE}\, i \qquad (c(i) = c(0))$$

und

$$\texttt{INDSTORE}\, i \qquad (c(c(i)) = c(0)).$$

Eine Registermaschine besitzt die arithmetischen Befehle

$$\begin{aligned} \texttt{ADD}\, i &\qquad (c(0) = c(0) + c(i)) \\ \texttt{SUBT}\, i &\qquad (c(0) = c(0) - c(i)) \\ \texttt{MULT}\, i &\qquad (c(0) = c(0) * c(i)) \\ \texttt{DIV}\, i &\qquad (c(0) = \lfloor c(0)/c(i) \rfloor) \end{aligned}$$

wie auch die indirekten Varianten

`INDADD`, `INDSUBT`, `INDMULT`, `INDDIV`

und die konstanten Operationen

KADD, KSUBT, KMULT *und* KDIV.

Schließlich haben wir die Vergleichsoperationen

IF $(c(0)\ v\ e)$ THEN GOTO j,

wobei $v \in \{\leq, <, =, >, \geq\}$, *e eine Konstante und j das Label eines Befehls ist. Der Inhalt des Akkumulators wird also mit der Konstanten e verglichen. Wenn der Vergleich positiv ist, wird der Befehl mit Label j als nächster ausgeführt; ansonsten wird der der if-Anweisung folgende Befehl ausgeführt.*

Schließlich ist auch eine „end-of-file"-Abfrage möglich:

IF *(eof)* THEN GOTO j

Die Sprungbefehle

GOTO j

und

END

sind die beiden letzten Befehlstypen.

(c) Initialisierung*: Die Eingabe erscheint linksbündig auf dem Eingabeband. Der Lesekopf des Eingabebandes wie auch der Schreibkopf des Ausgabebandes stehen auf der ersten Zelle ihres Bandes. Der Akkumulator, wie auch alle anderen Register, sind mit 0 initialisiert.*

Der erste auszuführende Befehl ist der Befehl mit Label 1.

Beispiel 5.4 Wir entwerfen eine Registermaschine, die die Kopierfunktion $w \mapsto ww$ für $w \in \{0,1\}^*$ effizient berechnet.

Die Registermaschine liest zuerst die Eingabe $w = w_1 \cdots w_n$ in die Register mit den Adressen 3, 4, ..., $n+2$. Im Register mit Adresse 1 ist 3, die Adresse von w_1, gespeichert. Das Register mit der Adresse 2 speichert n, die Anzahl der Bits. Gleichzeitig werden die eingelesenen Bits sofort auf das Ausgabeband geschrieben. Das folgende Programmsegment führt diesen Ladeprozeß aus:

```
(1)  KLOAD 3
(2)  STORE 1                 / * c(1) = 3 * /
(3)  IF (eof) THEN GOTO 11
(4)  READ
(5)  WRITE
(6)  INDSTORE 1              /* das gelesene Symbol wird gespeichert */
(7)  LOAD 1
(8)  KADD 1
(9)  STORE 1                 / * c(1) = c(1) + 1 * /
(10) GOTO 3
(11) LOAD 1                  / * c(0) = n + 3 * /
(12) KSUBT 3
(13) STORE 2                 / * c(2) = n * /
(14) KLOAD 3
(15) STORE 1                 / * c(1) = 3 * /
```

Der Kopierprozeß wird ausgeführt durch Wiederholung der folgenden Befehlssequenz:

```
(16)  LOAD 2
(17)  IF (c(0) = 0) THEN GOTO 27
(18)  INDLOAD 1
(19)  WRITE
(20)  LOAD 1
(21)  KADD 1
(22)  STORE 1                    /* c(1) = c(1) + 1 */
(23)  LOAD 2
(24)  KSUBT 1
(25)  STORE 2                    /* c(2) = c(2) - 1 */
(26)  GOTO 16
(27)  END
```

Man beachte, dass die Kopierfunktion in Zeit $O(n)$ berechnet wird, also wesentlich schneller als die von Turingmaschinen benötigte quadratische Laufzeit.

Im obigen Beispiel haben wir die Anzahl der ausgeführten Schritte gezählt. Dieses Laufzeitmaß wird auch als *uniforme Laufzeit* bezeichnet. Ein realistischeres Laufzeitmaß ist die *logarithmische Laufzeit*: Hier summieren wir die Bitlängen der beteiligten Operanden für alle ausgeführten Befehle. (Operanden sind die manipulierten Daten wie auch die beteiligten Adressen. Zum Beispiel sind die logarithmischen Kosten des Befehls

$$\texttt{INDLOAD}\ i$$

die Summe der Bitlängen von $c(0)$, i und $c(i)$.) Beachte, dass die Kopierfunktion in Zeit

$$O(n \log n)$$

ausgeführt wird, wenn wir das logarithmische Kostenmaß benutzen. Warum? Die Store- und Load-Operationen sind teuer: Sie können bis zu $O(\log_2 n)$ Zeit kosten.

Wenn wir Turingmaschinen und Registermaschinen vergleichen, müssen wir die Eingabeformate aneinander angleichen: Turingmaschinen erwarten Worte über einem endlichen (aber beliebigen) Alphabet, während Registermaschinen Folgen natürlicher Zahlen erwarten.

Es ist aber einfach, die verschiedenen Formate ineinander zu übersetzen. Die Buchstaben des Alphabets können in einfacher Weise durch natürliche Zahlen kodiert werden, während wir eine natürliche Zahl durch seine Binärdarstellung (und damit durch ein Wort über dem Alphabet $\{0, 1\}$) kodieren können. Wie läßt sich aber eine Folge $(a_1, a_2, \ldots, a_r)$ natürlicher Zahlen kodieren? Indem wir zusätzlich den Buchstaben „,“ (zur Trennung der einzelnen Zahlen) verwenden.

Satz 5.4 *Sei R eine Registermaschine, die auf Eingaben der (logarithmischen) Länge n nach höchstens $t(n)$ Schritten (gemäß dem logarithmischen Kostenmaß) hält. Dann gibt es eine Turingmaschine M, so dass*

(a) M auf Eingaben der Länge n nach höchstens $O(t^6(n))$ Schritten hält und

(b) R und M dieselbe Funktion berechnen.

Beweis: Die Behauptung folgt, wenn wir eine Simulation von R durch eine k-Band Maschine M' in Zeit $O(t^3(n))$ durchführen.

Angenommen R hat auf Eingabe w für i Schritte gerechnet. Dann wird M' die folgende Struktur besitzen:

- Band 1 enthält die kodierte Eingabe. Die gegenwärtig von R gelesene Zelle ist markiert.
- Band 2 enthält die kodierte bisherige Ausgabe von R.
- Band 3 enthält Adressen und Inhalte der bisher von R benutzten Register. Wenn R zum Beispiel die Register mit den Adressen $r_1, \ldots, r_p$ und Inhalten $c(r_1), \ldots, c(r_p)$ zum gegenwärtigen Zeitpunkt benutzt hat, dann wird Band 3 die Form

  $$0\#\,\mathrm{Bin}(c(0))\#\#\,\mathrm{Bin}(r_1)\#\,\mathrm{Bin}(C(r_1))\#\#\cdots\#\#\,\mathrm{Bin}(r_p)\#\,\mathrm{Bin}(C(r_p))\#\#$$

  haben.
- Band 4 speichert das Label des nächsten Befehls von R.
- Weitere Hilfsbänder stehen zur Verfügung.

Die Eingabe w von R möge die logarithmische Länge n haben. Beachte, dass dann Band 3 zu jedem Zeitpunkt die Länge höchstens $t(n)$ besitzt. Wie simuliert M' den $(i+1)$-ten Schritt von R?

Betrachten wir als ein erstes Beispiel den Befehl

`INDLOAD` i.

Dann wird M' zuerst den Inhalt von Band 3 auf Band 5 kopieren. Dies gelingt in Zeit $O(t(n))$, denn Band 3 wird zu jedem Zeitpunkt höchstens die Länge $t(n)$ haben. Dann begibt sich der Kopf von Band 3 auf die Suche nach Register i. Auch dies gelingt in Zeit $O(t(n))$, denn die (konstante!) Adresse i kann in den Zuständen abgespeichert werden.

Wenn das Register i gefunden ist, bewegt sich der Kopf des dritten Bandes auf den Anfang von $\mathrm{Bin}(c(i))$. Der Kopf des fünften Bandes begibt sich jetzt mit Hilfe des Kopfes auf Band 3 auf die Suche nach einem Register mit Adresse $c(i)$. Auch dieses Register kann in Zeit $O(t(n))$ gefunden werden. Abschließend überschreibt der Kopf von Band 3 $\mathrm{Bin}(c(0))$ mit $\mathrm{Bin}(c(c(i)))$, und zwar unter Mithilfe des Kopfes auf Band 3 (Adresse 0 und Trennsymbol müssen möglicherweise neu geschrieben werden). Die Gesamtlaufzeit der Simulation des Befehls ist somit $O(t(n))$.

Betrachten wir als zweites Beispiel den Befehl

`MULT` i.

M' erreicht jetzt eine Simulation, wenn $c(0)$ und $c(i)$ auf die Bänder 5 bzw. 6 geschrieben werden und wenn auf den beiden Bändern mit der Schulmethode multipliziert wird. Die Schulmethode verlangt sukzessives Addieren der geshifteten Zahl $c(0)$ (wenn die entsprechenden Bits von $c(i)$ gleich 1 sind). Damit gelingt die Multiplikation also in Zeit

$$O\Big(\,|\mathrm{Bin}(c(0))|\cdot|\mathrm{Bin}(c(i))|\,\Big) = O\left(t^2(n)\right).$$

Letztlich ist noch der Akkumulator zu überschreiben.

Auch die verbleibenden Befehle von R lassen sich mit analogem Vorgehen jeweils in Zeit $O(t^2(n)$ simulieren. **Zusammengefaßt:** Höchstens $t(n)$ Schritte sind zu simulieren, wobei die Simulation eines jeden Schrittes in Zeit $O(t^2(n))$ gelingt. Wie behauptet, ist Zeit $O(t^3(n))$ ausreichend. ■

5.2 Die Klasse NP

Wir müssen jetzt den Begriff eines nichtdeterministischen Rechnermodells klären und die Klasse NP einführen. Nachdem wir bereits gesehen haben, dass Turingmaschinen Supercomputer mit polynomiellem Mehraufwand simulieren können, liegt es nahe, nichtdeterministische Rechner als nichtdeterministische Turingmaschinen zu definieren und genau das tun wir jetzt.

Wie konventionelle (deterministische) Turingmaschinen beschreiben wir auch eine nichtdeterministische Turingmaschine durch den Vektor

$$M = (Q, \Sigma, \delta, q_0, \Gamma, F),$$

allerdings müssen wir die Zustandsüberführungsfunktion der deterministischen Turingmaschine durch eine Überführungsrelation

$$\delta \subseteq Q \times \Gamma \times Q \times \Gamma \times \Big\{\mathsf{links}, \mathsf{bleib}, \mathsf{rechts}\Big\}.$$

ersetzen: Der nächste auszuführende Befehl kann jetzt aus einer Menge von möglicherweise mehreren Befehlen ausgewählt werden. Wenn sich die nichtdeterministische Turingmaschine im Zustand $q \in Q$ befindet und den Buchstaben $\gamma \in \Gamma$ liest, dann kann M jeden Befehl

$$(q, \gamma) \to (q', \gamma', \texttt{Richtung})$$

ausführen, solange

$$(q, \gamma, q', \gamma', \texttt{Richtung}) \in \delta$$

gilt.

Nichtdeterministische Turingmaschinen haben jetzt die Fähigkeit des Ratens erhalten, weil sie auf jeder Eingabe viele verschiedene Berechnungen ausführen können. Welche Berechnung soll aber „gelten“ und wie ist die Laufzeit einer nichtdeterministischen Turingmaschine M auf einer gegebenen Eingabe zu definieren?

M wird auf Eingabe w viele verschiedene Berechnungen durchlaufen; einige Berechnungen akzeptieren w, andere verwerfen w. Unsere intuitive Vorstellung ist, dass M die Fähigkeit haben soll, eine Lösung zu raten, wenn denn eine Lösung existiert. Wir sollten also sagen, dass M die Eingabe w genau dann akzeptiert, wenn mindestens eine Berechnung von M auf Eingabe w akzeptierend ist! Wie für deterministische Turingmaschinen definieren wir dann das von M gelöste Entscheidungsproblem (bzw. die von M erkannte Sprache) $L(M)$ durch

$$L(M) = \{w \mid M \text{ akzeptiert } w\}.$$

Das nächste Problem ist die Festlegung der Laufzeit von M auf Eingabe w, denn auch hier haben wir es mit mit vielen Berechnungen von unterschiedlichem Zeitaufwand zu tun. Unser Ziel ist die Definition einer möglichst großen Klasse NP, da dann die für NP schwierigsten Probleme „wirklich“ schwierig sein müssen. Deshalb messen wir die Zeit nur für akzeptierte Eingaben w und wählen die Schrittzahl der kürzesten akzeptierenden Berechnung als die Laufzeit von M auf w.

Jetzt können wir die worst-case Laufzeit von M auf Eingaben der Länge n durch

$$\texttt{Zeit}_M(n) = \max\{\text{ Laufzeit von } M \text{ auf } w \mid M \text{ akzeptiert } w \text{ und } w \in \Sigma^n\}$$

definieren. Die Definition von NP ist jetzt zwangsläufig:

Definition 5.5 *Ein Entscheidungsproblem (oder eine Sprache) L gehört genau dann zur Komplexitätsklasse* NP*, wenn es eine nichtdeterministische Turingmaschine M gibt mit $L = L(M)$ und $\texttt{Zeit}_M(n) = O\left((n+1)^k\right)$ für eine Konstante k.*

Beispiel 5.5 Wir weisen nach, dass $CLIQUE \in$ NP. Wir haben in Satz 5.4 gezeigt, dass Registermaschinen effizient durch Turingmaschinen simuliert werden können. Ebenso können wir nichtdeterministische Registermaschinen durch nichtdeterministische Turingmaschinen bei polynomiellem Mehraufwand simulieren. NP ist also auch die Klasse aller Entscheidungsprobleme, die von nichtdeterministischen Registermaschinen in polynomieller Zeit gelöst werden.

Jetzt ist aber $CLIQUE \in$ NP offensichtlich: wir raten k Knoten, die wir auf k Register verteilen. Dann überprüfen wir jedes Paar der k Register und verifizieren, dass das entsprechende Knotenpaar durch eine Kante verbunden ist. Wenn der Graph n Knoten hat, dann ist $k \leq n$ und wir haben offensichtlich polynomielle Laufzeit in n erreicht.

Mit genau demselben Argument können wir auch zeigen, dass SAT, $CLIQUE$, SCS und $BINPACKING$ zur Klasse NP gehören. Warum ist es aber überhaupt nicht offensichtlich, dass GP zur Klasse NP gehört?

Aufgabe 53
Nichtdeterministische Turingmaschinen können das nichtdeterministische Raten und das deterministische Verifizieren beliebig verzahnen. In dieser Aufgabe sehen wir, dass die Berechnungskraft nichtdeterministischer Turingmaschinen nicht eingeschränkt wird, wenn wir Raten und Verifizieren trennen.

Beweise den folgenden Satz:

Es sei M eine nichtdeterministische Turingmaschine, die das Entscheidungsproblem L löst.

Sei $Zeit_M(n)$ die Rechenzeit von M für Eingabelänge n. Dann gibt es eine nichtdeterministische Turingmaschine M' mit zwei Bändern, die das Entscheidungsproblem L löst und die folgenden Eigenschaften besitzt:

M' rät zu Beginn auf dem zweiten Band einen String aus $\{0,1\}^*$ und verhält sich danach deterministisch. Die Rechenzeit von M' ist dabei $Zeit_{M'}(n) = O(Zeit_M(n))$.

Aufgabe 54
Welche der folgenden Entscheidungsprobleme liegen in NP? Begründe deine Anwort.

$TAUTOLOGIE = \{\alpha \mid \alpha$ ist eine aussagenlogische Formel und alle möglichen Belegungen ihrer Variablen erfüllen die Formel $\}$

$PARTITION = \{(a_1, a_2, \ldots, a_m) \mid$ Es gibt eine Indexmenge $I \subseteq \{1, \ldots m\}$ mit $\sum_{i \in I} a_i = \sum_{j \in \{1,\ldots,m\} \setminus I} a_j\}$.

Können in der Klasse NP beliebig komplexe Entscheidungsprobleme liegen? Nein, denn:

Satz 5.6

(a) $\texttt{P} \subseteq \texttt{NP}$.

(b) Sei M eine nichtdeterministische Turingmaschine, die in Zeit $q(n)$ rechnet. Dann gibt es eine deterministische Turingmaschine M^* mit

$$L(M) = L(M^*) \text{ und } \texttt{Zeit}_{M^*}(n) \leq 2^{O(q(n))}.$$

Beweis (a): $\texttt{P} \subseteq \texttt{NP}$, denn eine deterministische Turingmaschine ist auch eine nichtdeterministische Turingmaschine.

(b) Sei w eine beliebige Eingabe. Wir simulieren M, für Eingabe w, auf *jeder* Berechnung der Länge höchstens $q(|w|)$. Die Eingabe w wird akzeptiert, sobald eine erste akzeptierende Berechnung gefunden wird und ansonsten verworfen.

Diese Strategie führen wir durch eine deterministische Turingmaschine M^* aus:

(a) M^* schreibt alle möglichen Folgen von $q(|w|)$ Befehlen auf Band 2.

(b) Dann führt M^* nacheinander alle Befehlsfolgen aus und

(c) akzeptiert, wenn eine legale, akzeptierende Befehlsfolge gefunden wurde.

Was ist die Laufzeit von M^*? Sei B die Anzahl der Befehle von M. Dann gibt es offensichtlich $B^{q(|w|)}$ verschiedene Befehlsfolgen. Für jede Befehlsfolge gelingt die Simulation in Zeit $O(q(|w|))$. Insgesamt benötigt M^* also Zeit

$$O\left(B^{q(|w|)} \cdot q(|w|)\right) = O\left(B^{q(|w|)} \cdot 2^{q(|w|)}\right) = 2^{O(q(|w|))}.$$

□

5.3 Der Begriff der Reduktion und die NP-Vollständigkeit

Wir kommen jetzt zum anscheinend härtesten Problem, denn wir müssen den Begriff eines schwierigsten Entscheidungsproblems in NP einführen. Zuerst formalisieren wir, dass ein Entscheidungsproblem L_1 auf ein zweites Entscheidungsproblem L_2 „reduziert“ werden kann.

Definition 5.7 *Σ_1 und Σ_2 seien zwei Alphabete. Für Entscheidungsprobleme L_1 (über Σ_1) und L_2 (über Σ_2) sagen wir, dass L_1 genau dann auf L_2* polynomiell reduzierbar ist (formal: *$L_1 \leq_p L_2$), wenn es eine deterministische Turingmaschine M gibt, so dass*

$$w \in L_1 \iff M(w) \in L_2$$

für alle $w \in \Sigma_1^*$ gilt. M rechnet in polynomieller Zeit und berechnet die „transformierte“ Eingabe $M(w)$.

Wir nennen M eine transformierende Turingmaschine.

Angenommen $L_1 \leq_p L_2$ gilt, und A ist ein Algorithmus, der in Zeit $t(n)$ entscheidet, ob eine Eingabe x der Länge n zum Entscheidungsproblem L_2 gehört. Wir entwerfen einen Algorithmus B mit Hilfe von A, der entscheidet, ob die Eingabe w zum Entscheidungsproblem L_1 gehört.

Algorithmus B:

(1) Wende die transformierende Turingmaschine M auf w an. Wir erhalten die transformierte Eingabe $M(w)$.

/* Beachte, dass $M(w)$ in polynomieller Zeit in der Länge von w berechnet werden kann. */

(2) Wende Algorithmus A auf die transformierte Eingabe $M(w)$ an und akzeptiere w genau dann, wenn Algorithmus A die transformierte Eingabe $M(w)$ akzeptiert.

/* Da wir $L_1 \leq_p L_2$ annehmen, folgt $w \in L_1 \Leftrightarrow M(w) \in L_2$. Algorithmus B ist also korrekt. */

Beachte, dass Algorithmus B die Laufzeit höchstens $t(\text{poly}(n))$ besitzt. Wenn L_2 in P liegt und wenn Algorithmus B einen Algorithmus für L_2 mit polynomieller Laufzeit benutzt, dann ist also auch $L_1 \in$ P.

Satz 5.8 *Es gelte $L_1 \leq_{\mathrm{p}} L_2$. Wenn $L_2 \in \mathtt{P}$, dann auch $L_1 \in \mathtt{P}$.*

In diesem Sinne interpretieren wir eine polynomielle Reduktion $L_1 \leq_{\mathrm{p}} L_2$ als die Aussage, dass L_2 mindestens so schwierig wie L_1 ist. Jetzt ist aber auch klar, wie wir die schwierigsten Entscheidungsprobleme in NP definieren sollten.

Definition 5.9 *Sei L ein Entscheidungsproblem.*

(a) L heißt NP-hart genau dann, wenn

$$K \leq_p L \text{ für alle Entscheidungsprobleme } K \in \mathtt{NP}.$$

(b) L heißt NP-vollständig genau dann, wenn $L \in \mathtt{NP}$ und L ein NP-hartes Entscheidungsproblem ist.

Der nächste Satz zeigt, dass **keine** NP-vollständiges Entscheidungsproblem effizient berechenbar ist, wenn $\mathtt{P} \neq \mathtt{NP}$. Die Hypothese $\mathtt{P} \neq \mathtt{NP}$ scheint sehr plausibel, da ansonsten *Raten deterministisch simulierbar ist.* Allerdings erinnern wir uns daran, dass es bis heute nicht gelungen ist nachzuweisen, dass die Klasse P tatsächlich eine echte Teilmenge von NP ist!

Satz 5.10 *Das Entscheidungsproblem L sei NP-vollständig. Wenn $\mathtt{P} \neq \mathtt{NP}$, dann ist $L \notin \mathtt{P}$.*

Beweis Angenommen, $L \in \mathtt{P}$. Dann genügt der Nachweis, dass $\mathtt{P} = \mathtt{NP}$.

Sei K ein beliebiges Entscheidungsproblem in NP. Da L NP-vollständig ist, gilt $K \leq_p L$. Mit Satz 5.8 erhalten wir also $K \in \mathtt{P}$, denn wir haben ja $L \in \mathtt{P}$ angenommen. Da aber K ein beliebiges Entscheidungsproblem in NP ist, haben wir $\mathtt{NP} \subseteq \mathtt{P}$ nachgewiesen und mit Satz 5.6 folgt $\mathtt{P} = \mathtt{NP}$. □

Aufgabe 55
Angenommen, es gibt ein NP-vollständiges Entscheidungsproblem L, so dass $\overline{L} \in \mathtt{NP}$. **Zeige** dann, dass für jedes Entscheidungsproblem $K \in \mathtt{NP}$ gilt $\overline{K} \in \mathtt{NP}$.

Wie können wir zeigen, dass ein Entscheidungsproblem NP-vollständig ist? Wir gehen wie folgt vor: Zuerst zeigen wir in Kapitel 6.1 mit einer relativ aufwändigen Konstruktion, dass eine Version von *SAT* zu den schwierigsten Entscheidungsproblemen in NP gehört, also NP-vollständig ist.

Wie erhalten wir weitere NP-vollständige Entscheidungsprobleme? Die Intuition sagt, dass ein Problem $L_2 \in \mathtt{NP}$ ebenfalls NP-vollständig ist, wenn wir ein bereits als NP-vollständig bekanntes Problem L_1 auf L_2 reduzieren können, wenn also $L_1 \leq_P L_2$ gilt. Man sollte nämlich annehmen, dass L_2 ein schwierigstes Problemproblem ist, wenn wir mit Hilfe von L_2 ein schwierigstes Problem L_1 lösen können. Diese Intuition ist richtig.

Satz 5.11 *Seien L_1, L_2 und L_3 Entscheidungsprobleme.*

(a) Wenn $L_1 \leq_{\mathrm{p}} L_2$ und $L_2 \leq_{\mathrm{p}} L_3$, dann $L_1 \leq_{\mathrm{p}} L_3$.

(b) Sei L_1 NP-vollständig, und es gelte $L_1 \leq_{\mathrm{p}} L_2$. Dann ist L_2 NP-hart.
Wenn zusätzlich $L_2 \in \mathtt{NP}$ gilt, dann ist L_2 NP-vollständig.

Beweis (a): M_1 (bzw. M_2) sei eine transformierende Turingmaschine für die Reduktion $L_1 \leq_{\mathrm{p}} L_2$ (bzw. $L_2 \leq_{\mathrm{p}} L_3$). Dann ist die Turingmaschine M, die zuerst M_1 simuliert und dann M_2 auf der Ausgabe von M_1 simuliert, eine transformierende Turingmaschine für die Reduktion $L_1 \leq_{\mathrm{p}} L_3$, denn aus

$$\forall w\ (w \in L_1 \Leftrightarrow M_1(w) \in L_2) \text{ und } \forall u\ (u \in L_2 \Leftrightarrow M_2(u) \in L_3)$$

folgt

$$\forall w\ (w \in L_1 \Leftrightarrow M_1(w) \in L_2 \Leftrightarrow M_2(M_1(w)) \in L_3).$$

(b) Sei K ein beliebige Entscheidungsproblem in NP. Dann müssen wir die Reduktion $K \leq_{\mathrm{p}} L_2$ nachweisen. Da L_1 NP-vollständig ist, wissen wir, dass

$$K \leq_p L_1$$

gilt. Aber nach Teil (a), folgt $K \leq_{\mathrm{p}} L_2$ aus $K \leq_p L_1$ und $L_1 \leq_{\mathrm{p}} L_2$. □

Kapitel 6

NP-vollständige Probleme

Zuerst erledigen wir den wichtigsten Schritt, wir zeigen nämlich die NP-Vollständigkeit des Entscheidungsproblems *KNF–SAT*, wobei

$$KNF\text{–}SAT = \Big\{\alpha \quad | \quad \alpha \text{ ist eine Formel in konjunktiver Normalform, und } \alpha \text{ ist erfüllbar}\Big\}.$$

KNF–SAT ist also eine Variante von *SAT*, in der alle erfüllbaren aussagenlogischen Formeln in konjunktiver Normalform gesammelt sind.

Danach wenden wir Satz 5.11 wiederholt an, um weitere NP-Vollständigkeitsergebnisse zu erhalten.

6.1 Der Satz von Cook: Die NP-Vollständigkeit von *KNF–SAT*

Beachte, dass eine Formel α in konjunktiver Normalform die Form

$$\alpha \equiv k_1 \wedge k_2 \wedge \ldots \wedge k_r$$

hat. Die *Klauseln* $k_1, \ldots, k_r$ sind Disjunktionen von *Literalen* (also Disjunktionen von Variablen oder negierten Variablen). α heißt *erfüllbar*, wenn es eine Belegung der aussagenlogischen Variablen gibt, die α wahr macht.

Beispiel 6.1 $\alpha \equiv (x \vee y) \wedge (\neg x \vee y) \wedge (x \vee \neg y)$ ist erfüllbar, denn die Belegung $x =$ wahr und $y =$ wahr macht auch α wahr. Die Formel $\beta \equiv (x \vee y) \wedge (\neg x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee \neg y)$ ist hingegen nicht erfüllbar. Warum? $\neg\beta \equiv (\neg x \wedge \neg y) \vee (x \wedge \neg y) \vee (\neg x \wedge y) \vee (x \wedge y)$ ist eine Tautologie, da alle vier Belegungsmöglichkeiten für x und y eine Konjunktion erfüllen.

Das Ziel dieses Kapitels ist der Nachweis von

Satz 6.1 *(Der Satz von Cook)*

KNF–SAT ist NP*-vollständig.*

Beweis von Satz 6.1: Für ein beliebiges Entscheidungsproblem $L \in$ NP müssen wir zeigen, dass $L \leq_{\mathrm{p}} KNF\text{–}SAT$. Was wissen wir über L? Es gibt eine nichtdeterministische Turingmaschine $M_L = (Q, \Sigma, \delta, q_0, \Gamma, F)$ und ein Polynom p mit

- $L = L(M_L)$ und
- M_L hat Laufzeit p, es gibt also für jedes Wort $w \in L$ eine akzeptierende Berechnung der Länge höchstens $p(|w|)$.

Nach Umbenennung von Zuständen und Buchstaben des Bandalphabets Γ können wir annehmen, dass

- M_L die Zustandsmenge $\{0, 1, \ldots, q\}$ besitzt
- sowie das Bandalphabet $\Gamma = \{0, 1, \mathtt{B}\}$.
- 0 ist Anfangszustand, und 1 ist der einzige akzeptierende Zustand. Weiterhin wird M_L stets halten, wenn Zustand 1 erreicht wird.

Aufgabe 56
Warum können wir fordern, dass M_L die obigen Eigenschaften hat?

Was ist zu zeigen? Für jede Eingabe w ist eine Formel α_w in konjunktiver Normalform zu konstruieren, so dass

$$\begin{aligned} w \in L &\Leftrightarrow \text{Es gibt eine Berechnung von } M_L, \text{ die } w \text{ akzeptiert} \\ &\Leftrightarrow \alpha_w \in \mathit{KNF\text{–}SAT}. \end{aligned}$$

Die Funktion $w \mapsto \alpha_w$ muss von einer transformierenden Turingmaschine M in (deterministisch) polynomieller Zeit konstruierbar sein.

Wir werden dabei ausnutzen, dass wir das Raten einer akzeptierenden Berechnung als das Raten einer erfüllenden Belegung deuten können. Um dies erreichen zu können, muss sich die Aussagenlogik als genügend ausdrucksstark herausstellen, damit wir in der Aussagenlogik über Turingmaschinen „sprechen“ können. Anders ausgedrückt, wir müssen die Aussagenlogik als eine Programmiersprache verstehen und in dieser Programmiersprache ein zu M_L äquivalentes Programm schreiben.

Die Variablen von α_w: Berechnungen von M_L können nach $T = p(|w|)$ vielen Schritten abgebrochen werden: Wenn es eine akzeptierende Berechnung gibt, dann gibt es eine akzeptierende Berechnung mit höchstens T vielen Schritten. Beachte, dass es gemäß unserem Abbruchskriterium möglich ist anzunehmen, dass akzeptierende Berechnungen für *genau* T viele Schritte laufen: Wir lassen die Maschine einfach genügend viele Schritte auf der Stelle treten.

Da wir nur Berechnungen der Länge T betrachten, kann der Kopf nur Zellen mit den Adressen $-T, \ldots, T$ erreichen. Wir arbeiten mit drei Klassen von Variablen. Mit den Variablen

$$\text{Zustand}_t(i), \quad 0 \le t \le T,\ i \in \{0, \ldots, q\}$$

beabsichtigen wir auszudrücken, dass M_L zum Zeitpunkt t den Zustand i besitzt. Eine Variable

$$\text{Kopf}_t(\mathtt{wo}), \quad 0 \le t \le T,\ -T \le \mathtt{wo} \le T$$

soll genau dann wahr sein, wenn sich der Kopf von M_L zum Zeitpunkt t auf der Zelle mit Adresse *wo* befindet. Schießlich besitzen wir noch die Variablen

$$\text{Zelle}_t(\mathtt{wo}, \mathtt{was}), \quad 0 \le t \le T,\ -T \le \mathtt{wo} \le T,\ \mathtt{was} \in \Gamma$$

Hier wird versucht auszudrücken, dass die Zelle mit Adresse wo zum Zeitpunkt t das Symbol was speichert.

Die Klauseln von α_w müssen so gewählt werden, dass die beabsichtigte Interpretation garantiert wird: Eine erfüllende Belegung von α_w muss einer Berechnung von M_L entsprechen. Wir sind aber natürlich nur an akzeptierenden Berechnungen interessiert. Deshalb wird α_w die (triviale) Klausel

$$\text{Zustand}_T(1)$$

besitzen, wobei wir uns daran erinnern, dass 1 der einzige akzeptierende Zustand ist.

Klauselbeschreibung der Startkonfiguration: Wir drücken aus, dass M_L

- sich im Zustand 0 befindet,
- ihr Kopf die Zelle 1 liest,
- die Zellen $1, \ldots, |w|$ die Eingabe w speichern
- und die restlichen Zellen das Blanksymbol speichern.

Dies gelingt „beinahe“ durch die Konjunktion

$$\alpha_0 \equiv \text{Zustand}_0(0) \wedge \text{Kopf}_0(1) \wedge \bigwedge_{\mathtt{wo}=1}^{|w|} \text{Zelle}_0(\mathtt{wo}, w_{\mathtt{wo}}) \wedge \bigwedge_{\mathtt{wo} \in \{-T,\ldots,T\}\setminus\{1,\ldots,|w|\}} \text{Zelle}_0(\mathtt{wo}, \mathsf{B}).$$

Wir müssen zusätzlich noch ausdrücken, dass M_L

- sich in genau einem Zustand befindet,
- ihr Kopf genau eine Zelle liest und
- jede Zelle genau ein Symbol speichert.

Allgemein müssen wir durch eine Formel in konjunktiver Normalform ausdrücken, dass von vorgegebenen Variablen $y_1, \ldots, y_s$ genau eine Variable wahr ist. Dies gelingt, wenn wir $y_1 \vee \ldots \vee y_s$ fordern (mindestens eine Variable ist wahr), sowie

$$\bigwedge_{1 \le i < j \le s} \neg y_i \vee \neg y_j$$

fordern (keine zwei Variablen sind wahr). Die Konjunktion dieser $\binom{s}{2} + 1$ Klauseln bezeichnen wir als die Formel $\gamma_s(y_1, \ldots, y_s)$. Wir können jetzt erreichen, dass M_L *zu jedem Zeitpunkt*

- genau einen Zustand besitzt,
- genau eine Lesekopfposition hat
- und jede Zelle genau ein Symbol speichert.

wenn wir die Formel

$$\begin{aligned}\gamma \equiv \quad & \bigwedge_{t=0}^{T} \gamma_{q+1}\Big(\text{Zustand}_t(0), \ldots, \text{Zustand}_t(q)\Big) \\ \wedge \quad & \bigwedge_{t=0}^{T} \gamma_{2T+1}\Big(\text{Kopf}_t(-T), \ldots, \text{Kopf}_t(T)\Big) \\ \wedge \quad & \bigwedge_{t=0}^{T} \bigwedge_{\texttt{wo}=-T}^{T} \gamma_3\Big(\text{Zelle}_t(\texttt{wo}, 0), \text{Zelle}_t(\texttt{wo}, 1), \text{Zelle}_t(\texttt{wo}, B)\Big).\end{aligned}$$

verwenden.

Klauselbeschreibung eines Update-Schrittes von M_L: Angenommen, wir haben erreicht, dass eine erfüllende Belegung von

$$\gamma \wedge \alpha_0 \wedge \alpha_1 \wedge \ldots \wedge \alpha_t$$

einer Berechnung von M_L der Länge t entspricht. Wie muss α_{t+1} konstruiert werden, damit sich erfüllende Belegungen von

$$\gamma \wedge \alpha_0 \wedge \alpha_1 \wedge \ldots \wedge \alpha_t \wedge \alpha_{t+1}$$

und Berechnungen von M_L der Länge $t+1$ gegenseitig entsprechen? α_{t+1} muss ausdrücken, dass sich Zustand, Kopfposition und Zelleninhalt gemäß des Programms δ und der Konfiguration zum Zeitpunkt t ändern.

Zuerst drücken wir aus, dass nur die vom Kopf gelesene Zelle verändert werden kann. Für jede Position $\texttt{wo} \in \{-T, \ldots, T\}$ und jedes Symbol $\texttt{was} \in \{0, 1, \texttt{B}\}$ werden wir deshalb

$$\text{Zelle}_t(\texttt{wo}, \texttt{was}) \wedge \neg\, \text{Kopf}_t(\texttt{wo}) \rightarrow \text{Zelle}_{t+1}(\texttt{wo}, \texttt{was})$$

als Klausel in die zu konstruierende Formel α_{t+1} aufnehmen.

Wir denken uns als nächstes die Befehle von δ durchnummeriert. Eine Berechnung von M_L wird nun mit einem von möglicherweise mehreren ausführbaren Befehlen fortgesetzt. Um dies auszudrücken, führen wir die Befehlsvariablen

$$\text{Befehl}_{t+1}(i)$$

ein mit der beabsichtigten Interpretation, dass $\text{Befehl}_{t+1}(i)$ wahr ist, wenn der Befehl i zum Zeitpunkt $t+1$ ausführbar ist. Für jedes Paar (z, a) von Zustand z und Bandsymbol a sei

$$B(z,a) = \Big\{ i \quad | \quad \text{Befehl } i \text{ ist ausführbar, wenn Zustand } z \text{ erreicht ist und Buchstabe } a \text{ gelesen wird} \Big\}.$$

Wir drücken jetzt aus, dass M_L einen ausführbaren Befehl nichtdeterministisch wählt:

$$\text{Zelle}_t(\texttt{wo}, \texttt{was}) \wedge \text{Kopf}_t(\texttt{wo}) \wedge \text{Zustand}_t(z) \rightarrow \bigvee_{i \in B(z, \texttt{was})} \text{Befehl}_{t+1}(i).$$

Zusätzlich müssen wir aber noch fordern, dass genau ein Befehl ausgeführt wird und erreichen dies durch die Formel $\gamma_b\Big(\text{Befehl}_{t+1}(0), \text{Befehl}_{t+1}(1), \ldots, \text{Befehl}_{t+1}(b)\Big)$. (Wir haben hier angenommen, dass δ aus genau b Befehlen besteht.) Für jeden der b Befehle fügen wir als

nächstes Klauseln ein, die die Aktion des Befehls beschreiben. Exemplarisch betrachten wir aber nur den i-ten Befehl, der die Form

$$(z, \texttt{was}) \to (z', \texttt{was}', \texttt{Richtung})$$

habe. Wir nehmen ohne Beschränkung der Allgemeinheit an, dass $\texttt{Richtung} = \texttt{links}$ gilt. Die folgenden Klauseln werden in α_{t+1} erscheinen:

$$\begin{aligned}
&\text{Zelle}_t(\texttt{wo}, \texttt{was}) \wedge \text{Kopf}_t(\texttt{wo}) \wedge \text{Zustand}_t(z) \wedge \text{Befehl}_{t+1}(i) \to \text{Zustand}_{t+1}(z') \\
&\text{Zelle}_t(\texttt{wo}, \texttt{was}) \wedge \text{Kopf}_t(\texttt{wo}) \wedge \text{Zustand}_t(z) \wedge \text{Befehl}_{t+1}(i) \to \text{Zelle}_{t+1}(\texttt{wo}, \texttt{was}') \\
&\text{Zelle}_t(\texttt{wo}, \texttt{was}) \wedge \text{Kopf}_t(\texttt{wo}) \wedge \text{Zustand}_t(z) \wedge \text{Befehl}_{t+1}(i) \to \text{Kopf}_{t+1}(\texttt{wo} - 1).
\end{aligned}$$

α_{t+1} ist die Konjunktion aller obigen Klauseln. Beachte, dass α_{t+1} aus höchstens polynomiell (in $|w|$) vielen Klauseln besteht.

Definition von α_w: Wir setzen

$$\alpha_w = \gamma \wedge \alpha_0 \wedge \alpha_1 \wedge \ldots \wedge \alpha_T \wedge \text{Zustand}_T(1).$$

Dann hat α_w, binär kodiert, höchstens polynomielle Länge (in $|w|$) und kann in polynomieller Zeit (in $|w|$) von einer transformierenden Turingmaschine M konstruiert werden.

Zusätzlich haben wir α_w so konstruiert, dass sich erfüllende Belegungen von α_w und akzeptierende Berechnungen von M_L auf Eingabe w eindeutig entsprechen. Als Konsequenz haben wir somit unser anfänglich angekündigtes Ziel,

$$\begin{aligned}
w \in L &\Leftrightarrow \text{Es gibt eine Berechnung von } M_L, \text{ die } w \text{ akzeptiert} \\
&\Leftrightarrow \alpha_w \in \mathit{KNF\text{–}SAT},
\end{aligned}$$

erreicht. Die Reduktion $L \leq_{\text{p}} \mathit{KNF\text{–}SAT}$ ist nachgewiesen und $\mathit{KNF\text{–}SAT}$ ist NP-vollständig. □

Eine Abschwächung von $\mathit{KNF\text{–}SAT}$ ist das Entscheidungsproblem

$$k\text{–}\mathit{SAT} = \Big\{\alpha \in \mathit{KNF\text{–}SAT} \,|\, \alpha \text{ besitzt höchstens } k \text{ Literale pro Klausel}\Big\}.$$

Wir zeigen jetzt, dass sogar das anscheinend einfachere $3\text{–}\mathit{SAT}$ bereits NP-vollständig ist.

Satz 6.2 *$\mathit{KNF\text{–}SAT} \leq_{\text{p}} 3\text{–}\mathit{SAT}$ und damit ist $3\text{–}\mathit{SAT}$ NP-vollständig.*

Beweis: Wir müssen eine polynomiell zeitbeschränkte, transformierende Turingmaschine M konstruieren. Sei α mit $\alpha \equiv k_1 \wedge \ldots \wedge k_r$ eine Eingabe für $\mathit{KNF\text{–}SAT}$. M transformiert eine Klausel k_i in eine Konjunktion

$$k_{i,1} \wedge \ldots \wedge k_{i,r_i} \equiv \alpha_i$$

von Klauseln mit höchstens drei Literalen.

Fall 1: k_i hat höchstens drei Literale.

Dann setzen wir $\alpha_i \equiv k_i$: Die Klausel k_i wird also nicht verändert.

Fall 2: k_i hat $l \geq 4$ Literale.

Insbesondere sei

$$k_i \equiv y_1 \vee \ldots \vee y_l$$

mit Literalen $y_1, \ldots, y_l$. Wir erfinden *neue* Variablen $z_{i,1}, z_{i,2}, z_{i,3}, \ldots, z_{i,l-3}$ und setzen

$$\begin{aligned} k_{i,1} &\equiv y_1 \vee y_2 \vee z_{i,1} \\ k_{i,2} &\equiv \neg z_{i,1} \vee y_3 \vee z_{i,2} \\ k_{i,3} &\equiv \neg z_{i,2} \vee y_4 \vee z_{i,3} \\ &\vdots \\ k_{i,l-3} &\equiv \neg z_{i,l-4} \vee y_{l-2} \vee z_{i,l-3} \\ k_{i,l-2} &\equiv \neg z_{i,l-3} \vee y_{l-1} \vee y_l. \end{aligned}$$

Eine Belegung der Literale $y_1, \ldots, y_l$ sei vorgegeben. Wir behaupten:

$$\alpha_i \equiv k_{i,1} \wedge k_{i,2} \wedge k_{i,3} \wedge \ldots \wedge k_{i,l-2} \text{ ist erfüllbar} \Leftrightarrow \; k_i \equiv y_1 \vee \cdots \vee y_l \text{ ist wahr.}$$

Beweis: Angenommen, k_i ist wahr. Dann ist ein Literal y_j wahr und wir können setzen:

$$\begin{aligned} z_{i,1} &= \text{wahr} \quad (\text{und } k_{i,1} \text{ ist wahr}), \\ z_{i,2} &= \text{wahr} \quad (\text{und } k_{i,2} \text{ ist wahr}), \\ &\vdots \\ z_{i,j-2} &= \text{wahr} \quad (\text{und } k_{i,j-2} \text{ ist wahr}). \end{aligned}$$

Beachte, dass $k_{i,j-1} = \neg z_{i,j-2} \vee y_j \vee z_{i,j-1}$. Die Klausel ist also wahr, da y_j wahr ist, und wir können $z_{i,j-1} =$ falsch setzen. Dann wird aber

$$k_{i,j} \equiv \neg z_{i,j-1} \vee y_{j+1} \vee z_{i,j}$$

wahr, und wir können wieder $z_{i,j} =$ falsch setzen. Wenn wir dieses Argument wiederholen, sehen wir, dass alle Klauseln $k_{i,1}, \ldots, k_{i,l-2}$ wahr sind, falls $z_{i,1} = \ldots = z_{i,j-2} =$ wahr und $z_{i,j-1} = \ldots = z_{i,l-3} =$ falsch.

Angenommen, k_i ist falsch. Dann ist keines der Literale $y_1, \ldots, y_l$ wahr und wir sind gezwungen, sukzessive $z_{i,1} = z_{i,2} = \ldots = z_{i,l-2} =$ wahr zu setzen: Die letzte Klausel $k_{i,l-2}$ wird falsch!

M weist der Formel $\alpha \equiv k_1 \wedge \ldots \wedge k_r$ die Formel

$$M(\alpha) = \alpha_1 \wedge \ldots \wedge \alpha_r$$

zu, wobei α_i eine Konjunktion von Klauseln mit drei Literalen ist. Es gilt

$$\begin{aligned} \alpha \in \mathit{KNF\text{–}SAT} &\Leftrightarrow \text{Es gibt eine Belegung, die jede der Klauseln } k_1, \ldots, k_r \text{ wahr macht.} \\ &\Leftrightarrow \text{Es gibt eine Belegung, die alle Formeln } \alpha_1, \ldots, \alpha_r \text{ wahr macht.} \\ &\Leftrightarrow M(\alpha) \in 3\text{–}\mathit{SAT} \end{aligned}$$

und die Behauptung ist gezeigt. □

Also wird aussagenlogisches Schließen für Formeln in konjunktiver Normalform schon für Formeln mit nur 3 Literalen pro Klausel schwierig.

Aufgabe 57
Zeige, dass 2–$\mathit{SAT} \in$ P.

6.2 Weitere NP-vollständige Probleme

Achtung! Bei allen Entscheidungsproblemen in diesem Kapitel überzeuge man sich, dass sie zur Klasse NP gehören. Wir geben keine Beweise hierfür an.

6.2.1 Clique

Im nächsten Schritt zeigen wir, dass das *CLIQUE-Problem* schwierig ist, indem wir 3–*SAT* auf *CLIQUE* reduzieren. Wir müssen also ein Problem der Aussagenlogik mit graph-theoretischen Mitteln effizient lösen!

Satz 6.3 *3–SAT* $\leq_{\mathrm{p}}$ *CLIQUE und CLIQUE ist* NP*-vollständig.*

Beweis: Wir müssen eine transformierende Turingmaschine M konstruieren, so dass

$$\alpha \in 3\text{–}SAT \quad \Leftrightarrow \quad M(\alpha) \in CLIQUE$$

gilt. Die Formel α habe die Form $\alpha \equiv k_1 \wedge \cdots \wedge k_r$ mit den Klauseln $k_1, \ldots, k_r$, wobei

$$k_i \equiv l_{i,1} \vee l_{i,2} \vee l_{i,3}.$$

M weist der Formel α einen ungerichteten Graphen $G(\alpha) = (V, E)$ zu. $G(\alpha)$ besitzt für jede Klausel k_i eine Gruppe $g_i = \{(i,1), (i,2), (i,3)\}$ von 3 Knoten. Wir definieren dann

$$\bigcup_{i=1}^{r} g_i = V$$

als die Knotenmenge von $G(\alpha)$. Wir verbinden Knoten einer Gruppe *nicht* mit Kanten. Ansonsten verbinden wir Knoten (i,s) und Knoten (j,t) mit $i \neq j$ nur dann *nicht*, wenn

$$l_{i,s} \equiv \neg l_{j,t}$$

gilt, das heißt, wenn $l_{i,r}$ und $l_{j,s}$ nicht beide simultan erfüllbar sind. Wir behaupten

$$\alpha \in 3\text{–}SAT \quad \Leftrightarrow \quad G(\alpha) \text{ hat eine Clique der Größe } r.$$

Wir haben genau dann Kanten zwischen zwei Knoten eingesetzt, wenn sich „ihre“ Literale nicht widersprechen.

Beweis: Wir müssen also zeigen, dass α genau dann erfüllbar ist, wenn $G(\alpha)$ eine Clique der Größe r besitzt. Da $G(\alpha)$ aus r Gruppen aufgebaut ist, wobei Knoten derselben Gruppe nicht miteinander verbunden sind, muss eine Clique der Größe r genau einen Knoten pro Gruppe auswählen!

Wenn $\alpha \in 3\text{–}SAT$, dann gibt es eine Belegung, die mindestens ein Literal $l_{i,j(i)}$ für jede Klausel k_i wahr macht. Dann ist aber $\{(1, j(1)), (2, j(2)), \ldots, (r, j(r))\}$ eine Clique, denn je zwei Literale sind simultan erfüllbar, und deshalb ist $(G(\alpha), r) \in CLIQUE$.

Andererseits sei $(G(\alpha), r) \in CLIQUE$. Dann gibt es eine Clique C der Größe r. C hat also die Form

$$C = \Big\{(1, j(1)), (2, j(2)), \ldots, (r, j(r))\Big\}.$$

Es genügt zu zeigen, dass wir alle Literale $l_{1,j(1)}, l_{2,j(2)}, \ldots, l_{r,j(r)}$ simultan wahr machen können, denn dann haben wir mindestens ein Literal pro Klausel wahr gemacht, und die Formel α ist erfüllt. Aber diese Literale können tatsächlich wahr gemacht werden, da sich keine zwei Literale widersprechen, also nicht die Negation voneinander sind. □

Aufgabe 58
Wir haben bisher Konjunktionen von Klauseln untersucht und hierbei die NP-vollständigen Probleme $KNF\text{-}SAT$ und $k\text{-}SAT$ kennengelernt. Insbesondere haben wir gezeigt, dass 3-SAT bereits NP-vollständig ist. Das Problem wird einfach, wenn jede Klausel nur 2 Literale enthält, denn 2-$SAT \in P$.

Wie sieht es mit Konjunktionen von Klauseln aus, falls alle Klauseln **genau** drei Literale enthalten (hierbei darf innerhalb einer Klausel jedes Literal höchstens einmal vorkommen)?
$X3\text{-}SAT = \{\ \alpha \in KNF\text{–}SAT \mid \alpha$besitzt genau 3 verschiedene Literale pro Klausel$\}$

Zeige, dass $X3\text{-}SAT$ NP-vollständig ist.

Aufgabe 59
In dieser Aufgabe untersuchen wir Graph-Färbbarkeitsprobleme. Für ungerichtete Graphen $G = (V, E)$ und eine natürliche Zahl k soll entschieden werden, ob sich die Knoten von G mit k Farben färben lassen, d.h. ob es eine Knoten-Färbungs-Funktion $f : V \to \{0, \ldots, k-1\}$ gibt, so dass $f(u) \neq f(v)$ für alle $\{u, v\} \in E$ ist.

In Worten: Es soll entschieden werden, ob die Knoten des Graphen mit höchstens k Farben gefärbt werden können, wobei Knoten, die durch eine Kante verbunden sind, verschiedene Farben erhalten müssen.

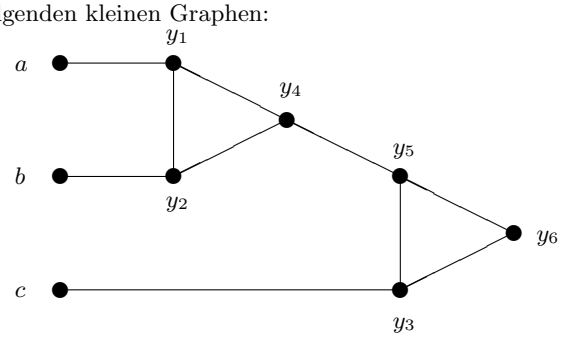
Dementsprechend definieren wir das Problem GC_k ($GRAPH\text{-}k\text{-}COLORIBILITY$) wie folgt: $GC_k = \{(G, k) \mid$ G hat n Knoten und ist mit $\leq k$ Farben färbbar $\}$.

(a) Zeige, dass GC_2 in P liegt.

(b) Zeige, dass $X3\text{-}SAT \leq_p GC_3$ gilt.

Hinweis: Betrachte den folgenden kleinen Graphen:

H



Der Graph H hat die folgenden zwei Eigenschaften:

(1) Ist f eine 3-Färbung von H mit $f(a) = f(b) = f(c)$, gilt $f(a) = f(y_6)$.

(2) Jede 3-Färbung der Knoten a, b und c, die die Farbe 1 an mindestens einen der drei Knoten vergibt, kann zu einer 3-Färbung von H mit $f(y_6) = 1$ erweitert werden.

Weise einer Formel $\alpha \in X3\text{-}SAT$ einen Graphen G_α zu, so dass in jeder 3-Färbung von G_α folgende Punkte gelten:

- Knoten, die Literalen entsprechen, sind nur mit 0 oder 1 gefärbt.
- Knoten, die zueinander komplementären Literalen entsprechen, sind verschieden gefärbt.
- Falls a, b und c mit 0 gefärbt sind, so muss aufgrund der Eigenschaft (1) von H auch y_6 die Farbe 0 erhalten. Deshalb liegt es nahe den Graphen G_α so zu konstruieren, dass die Farbe 1 für Knoten y_6 erzwungen wird.

6.2.2 Independent Set, Vertex Cover und Set Cover

Wir betrachten drei weitere kombinatorische Probleme. Zuerst betrachten wir das Knotenüberdeckungsproblem VC („Vertex Cover“), mit

$$VC = \Big\{(G, k) \quad | \quad \text{Es gibt eine Knotenmenge } \ddot{U} \text{ von höchstens } k \text{ Knoten, so dass jede Kante mindestens einen Endpunkt in } \ddot{U} \text{ besitzt}\Big\}.$$

Das Problem *Set Cover* (oder *Mengenüberdeckung*) *SC* ist

$$SC = \left\{ (A_1, \ldots, A_r, k) | \text{Es gibt höchstens } k \text{ Mengen } A_{i_1}, \ldots, A_{i_k} \text{mit } \bigcup_{j=1}^{k} A_{i_j} = \bigcup_{l=1}^{r} A_l \right\}.$$

Im *Set Cover* Problem wird also gefragt, ob das „Universums“ $\bigcup_{l=1}^{r} A_l$ eine Überdeckung mit nur wenigen der Mengen $A_1, \ldots, A_r$ besitzt.

Beispiel 6.2 Die Teilmengen mögen von der folgende Form sein:

$$\begin{aligned} A_1 &= \{1,2,7\} \\ A_2 &= \{3,4\} \\ A_3 &= \{1,5\} \\ A_4 &= \{4,7,10\} \\ A_5 &= \{5,6,7,8\} \\ A_6 &= \{8,10,11,12\} \\ A_7 &= \{5,6,9,10\} \\ A_8 &= \{2,6,9\} \end{aligned}$$

Die Vereinigung aller Mengen bildet das Universum $U = \{1, \ldots, 12\}$. Es gibt bereits eine Überdeckung mit $k = 4$ Mengen, nämlich den Mengen A_1, A_2, A_6 und A_7.

Das *Independent Set-Problem IS* (oder das *Problem der unabhängigen Mengen*) ist definiert durch

$$IS = \left\{ (G,k) \;\middle|\; \begin{array}{l} \text{Es gibt eine Knotenmenge } I \text{ von mindestens } k \text{ Knoten,} \\ \text{so dass keine zwei Knoten in } I \text{ durch eine Kante} \\ \text{verbunden sind} \end{array} \right\}.$$

Satz 6.4 *(a)* $CLIQUE \leq_{\mathrm{p}} IS$

(b) $IS \leq_{\mathrm{p}} VC$

(c) $VC \leq_{\mathrm{p}} SC$

und die NP*-Vollständigkeit von* VC*,* SC *und* IS *folgt.*

Beweis: Es stellt sich heraus, dass *CLIQUE*, *IS* und *VC* „fast“ dieselben Probleme sind, weshalb die Reduktionen (1) und (2) sehr einfach sind. Die Reduktion (3) ist ein klein wenig kitzliger.

(a) Sei $G = (V, E)$ ein Graph. Den Komplementgraphen bezeichnen wir mit $\overline{G} = (V, \overline{E})$, wobei

$$\overline{E} = \left\{ \{i,j\} \,|\, i \neq j \text{ und } \{i,j\} \notin E \right\}.$$

gelte. Offensichtlich folgt:

$$C \text{ ist eine Clique für } G \;\Leftrightarrow\; C \text{ ist eine unabhängige Menge für } \overline{G},$$

denn eine Clique in G ist eine unabhängige Menge in $\overline{G}$ und umgekehrt. Demgemäß setzen wir

$$M\Big((G,k)\Big) = \big(\overline{G}, k\big)\,.$$

(b) Wir behaupten,

$$I \text{ ist eine unabhängige Menge } \Leftrightarrow V \setminus I \text{ ist eine Knotenüberdeckung.}$$

Die Intuition ist klar, wenn I eine unabhängige Menge ist, dass sind keine zwei Knoten aus I durch eine Kante verbunden. Sämtliche Kanten besitzen somit einen Endpunkt in $V \setminus I$. Formal,

$$\begin{aligned} I \text{ ist unabhängig} &\Leftrightarrow \text{jede Kante von } G \text{ hat mindestens einen Endpunkt in } V \setminus I \\ &\Leftrightarrow V \setminus I \text{ ist eine Knotenüberdeckung.} \end{aligned}$$

Deshalb setzen wir

$$M\Big((G,k)\Big) = (G, |V| - k).$$

(c) Sei (G,k) eine vorgegebene Eingabe für VC. Wie können wir im Mengenüberdeckungsproblem VC über das Knotenüberdeckungsproblem SC „sprechen“? Wir nehmen an, dass G die Knotenmenge $V = \{1, 2, \ldots, n\}$ besitzt. Für jeden Knoten $i \in V$ definieren wir die Menge

$$A_i = \Big\{ e \in E \,|\, e \text{ besitzt } i \text{ als Endpunkt} \Big\}.$$

Damit ist die folgende Äquivalenz offensichtlich:

$$\begin{aligned} \ddot{U} \subseteq V \text{ ist eine Knotenüberdeckung} &\Leftrightarrow \text{jede Kante } e \in E \text{ besitzt einen Endpunkt in } \ddot{U} \\ &\Leftrightarrow \bigcup_{i \in \ddot{U}} A_i = \bigcup_{j=1}^{n} A_j = E. \end{aligned}$$

Wir können deshalb die Transformation

$$M\Big((G,k)\Big) = (A_1, \ldots, A_n, k).$$

definieren. □

Aufgabe 60

Wir haben schon verschiedene Überdeckungsprobleme kennengelernt. Das Matchingproblem

$$Match = \{(G,k) \mid G \text{ besitzt } k \text{ Kanten, die keinen Endpunkt gemeinsam haben. }\}$$

liegt in P, während das *Vertex Cover* Problem, wie auch *Set Cover*, NP-vollständig sind.

Wir betrachten hier mit dem Graphproblem *Edge Cover*, für ungerichtete Graphen $G = (V,E)$, ein weiteres Überdeckungsproblem:

$EC = \{(G,k) \mid G = (V,E)$ und es gibt eine Kantenmenge $E' \subseteq E$, mit $|E'| \leq k$, die alle Knoten überdeckt $\}$.

Ist EC NP-vollständig ? Ist $EC \in$ P? **Begründe** Deine Antwort.

Aufgabe 61

Für diese Aufgabe darf die **NP**-Vollständigkeit der folgenden Probleme als bekannt vorausgesetzt werden: *KNF-SAT, 3-SAT, Independent-Set, Vertex-Cover* oder *Clique*.

Die Eingaben für das Entscheidungsproblem *Hitting Set* bestehen aus natürlichen Zahlen n und k, sowie Teilmengen $S_1, \ldots, S_m$ der Menge $\{1, \ldots, n\}$.
Eine Eingabe $(n, k, S_1, \ldots, S_m)$ gehört zum Entscheidungsproblem *Hitting Set* genau dann, wenn es eine Teilmenge $T \subseteq \{1, \ldots n\}$ mit höchstens k Elementen gibt, die mit **jeder** Menge S_i mindestens ein Element gemeinsam hat.

Zeige, dass das Entscheidungsproblem *Hitting Set* **NP**-vollständig ist.

Hinweis: Zum Einen muss gezeigt werden, dass *Hitting Set* in der Klasse **NP** liegt, zum Anderen muss ein **NP**-vollständiges Problem auf *Hitting Set* polynomiell reduziert werden.

6.2.3 0-1 Programmierung und Ganzzahlige Programmierung

Wir erinnern an das Problem GP der ganzzahligen Programmierung: Eine Matrix A mit m Zeilen und n Spalten ist gegeben. Weiterhin gegeben sind zwei Vektoren $b = (b_1, \ldots, b_m)$ und $c = (c_1, \ldots, c_n)$, sowie eine natürliche Zahl k. Es wird gefragt, ob es einen Vektor $x \in \mathbb{Z}^m$ gibt mit

(a) $A \cdot x \leq b$ komponentenweise und

(b) $\sum_{i=1}^{m} c_i \cdot x_i \geq k$.

In der 0-1 Programmierung $01P$ ist die Fragestellung ähnlich; diesmal wird aber nach der Existenz eines Vektors x (mit Eigenschaften (a) und (b)) gefragt, wobei x nur 0- und 1-Komponenten besitzen darf.

Beide Problemvarianten sind Erweiterungen des Problems der linearen Programmierung, in dem nach der Existenz eines reellwertigen Vektors x gefragt wird. Das Problem der linearen Programmierung haben wir bereits in Kapitel 4.4 besprochen und dort erwähnt, dass schnelle Algorithmen wie Interior Point Verfahren existieren. Im Gegensatz zur linearen Programmierung besitzen $01P$ und GP in aller Wahrscheinlichkeit keine effizienten Lösungen, denn:

Satz 6.5

(a) $3\text{–}SAT \leq_{\mathrm{p}} 01P$,

(b) $01P \leq_{\mathrm{p}} GP$

und die NP*-Vollständigkeit von* $01P$ *und* GP *folgt.*

Aufgabe 62
Beweise Satz 6.5.

In der Vorlesung Approximationsalgorithmen wird auch gezeigt, dass effiziente Approximationsalgorithmen (in aller Wahrscheinlichkeit) nicht existieren. Approximationsalgorithmen versuchen einen 0-1 Vektor (bzw. ganzzahligen Vektor) x mit $A \cdot x \leq b$ zu bestimmen, wobei $\sum_{i=1}^{m} c_i \cdot x_i$ größtmöglich ist.

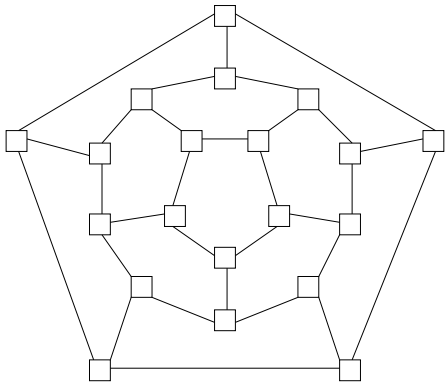
6.2.4 Wege in Graphen

Wir schließen die Betrachtung schwieriger graph-theoretischer Probleme mit der folgenden Gruppe von vier Problemen: Wir betrachten wieder das Traveling Salesman Problem *TSP* aus Beispiel 3.3,fordern diesmal aber nicht, dass die Kantenlängen eine Metrik definieren.

Im *Hamiltonschen Kreis-Problem HC* ist die Eingabe ein ungerichteter Graph G. Es wird gefragt, ob G einen Hamiltonschen Kreis besitzt, also einen Kreis, der jeden Knoten genau einmal durchläuft. Also

$$HC = \Big\{ G \,|\, G \text{ besitzt einen Hamiltonschen Kreis} \Big\}.$$

Beispiel 6.3 Betrachte folgenden ungerichteten Graphen:



Dieser Graph besitzt einen Hamiltonschen Kreis (welchen?) und gehört deshalb zu HC.

Wir betrachten ebenso das analoge Problem DHC für gerichtete Graphen wie auch LW, das Problem der längsten Wege.

Satz 6.6

(a) $DHC \leq_{\mathrm{p}} HC$

(b) $HC \leq_{\mathrm{p}} TSP$

(c) $HC \leq_{\mathrm{p}} LW$

und die NP*-Vollständigkeit von* HC*,* DHC*,* TSP *und* LW *folgt, falls* DHC NP*-vollständig ist.*

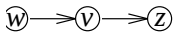
Beweis (a): Sei $G = (V, E)$ ein *gerichteter* Graph. Wir müssen einen *ungerichteten* Graphen G' konstruieren, so dass

G besitzt einen Hamiltonschen Kreis $\Leftrightarrow$ G' besitzt einen Hamiltonschen Kreis.

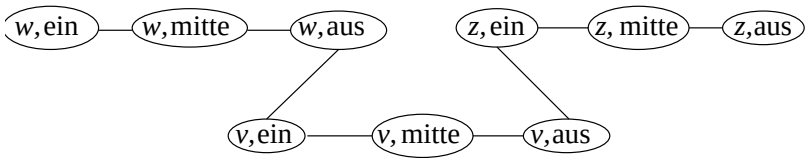
Wir spalten jeden Knoten v von G in drei Knoten (v, ein), (v, mitte) und (v, aus) auf. G' wird also die Knotenmenge

$$\Big\{(v, \text{ein}) \,|\, v \in V\Big\} \quad \cup \quad \Big\{(v, \text{mitte}) \,|\, v \in V\Big\} \quad \cup \quad \Big\{(v, \text{aus}) \,|\, v \in V\Big\}$$

besitzen. Wir verbinden zuerst für jeden Knoten v von G den Knoten (v, ein) (von G') mit (v, mitte) sowie (v, mitte) mit (v, aus). Für jede Kante (w, v) in G setzen wir schließlich die ungerichteten Kanten $\{(w, \text{aus}), (v, \text{ein})\}$ in den Graphen G' ein. (Wenn



ein Weg in G ist, dann erhalten wir



als Weg in G'.) Wir behaupten:

$$G \in DHC \quad \Leftrightarrow \quad G' \in HC.$$

Beweis: „⇒“ Ein gerichteter Hamiltonscher Kreis in G übersetzt sich sofort in einen ungerichteten Kreis in G'.

„⇐“ Wie sehen Hamiltonsche Kreise in G' aus? Wenn ein Kreis den Knoten (v, ein) (beziehungsweise den Knoten (v, aus)) durchläuft, muss unmittelbar danach der Knoten (v, mitte) durchlaufen werden. Warum? Sonst kann der Kreis den Knoten (v, mitte) nicht mehr durchlaufen!

Dann durchläuft ein Hamiltonscher Kreis in G' jede Knotengruppe also entweder stets in der Richtung ein→mitte→aus oder stets in der Richtung aus→mitte→ein. Und Hamiltonsche Kreise in G' übersetzen sich sofort in Hamiltonsche Kreise in G.

Die Transformation

$$M(G) = G'$$

weist somit die Reduktion $DHC \leq_{\text{p}} HC$ nach.

(b) Das *TSP* ist ein äußerst schwieriges und ausdrucksstarkes Problem, weswegen eine Reduktion sofort gelingt: Für Eingabe $G = (\{1, \ldots, n\}, E)$ wählen wir den vollständigen Graphen V_n, definieren die länge-Funktion

$$\text{länge}\Big(\{i, j\}\Big) = \begin{cases} 1 & \text{wenn } \{i, j\} \in E, \\ 2 & \text{sonst} \end{cases}$$

und wählen $B = n$. Die Transformation hat somit die Form

$$M(G) = (V_n, \text{länge}, B)$$

Wir müssen zeigen:

$$G \in HC \quad \Leftrightarrow \quad (V_n, \text{länge}, B) \in TSP.$$

Beweis: „⇒“ Wenn $G \in HC$, dann besitzt G einen Hamiltonschen Kreis. In V_n hat jede Kante des Kreises die Länge 1. Deshalb finden wir also eine Tour der Länge n, und $(V_n, \text{länge}, B) \in TSP$.

„⇐“ Angenommen, wir finden eine Tour der Länge n. Da eine Tour n Kanten hat und da jede Kante die Mindestlänge 1 besitzt, besteht die Tour (oder der Kreis) aus n Kanten der Länge 1 und somit nur aus Kanten des Graphen. Also ist $G \in HC$.

(c) Sei G ein ungerichteter Graph mit Knotenmenge $V = \{1, \ldots, n\}$. Dann besitzt G genau dann einen Hamiltonschen Kreis, wenn G einen Weg der Länge $n - 1$ besitzt, der

- im Knoten 1 beginnt und
- mit einem Nachbarn vom Knoten 1 endet.

Wir erfinden einen neuen Knoten $1'$, den wir zu G hinzufügen. Wir verbinden $1'$ mit allen Nachbarn von Knoten 1. Wir erfinden zwei weitere Knoten 0 und $0'$ und verbinden 0 mit 1 und $0'$ mit $1'$.

Den neuen Graphen mit Knotenmenge $V = \{0, 1, \ldots, n, 0', 1'\}$ nennen wir G'. Dann folgt

$$\begin{aligned} G \in HC \quad &\Leftrightarrow \quad G \text{ hat einen Weg der Länge } n-1, \text{ der in 1 beginnt} \\ &\qquad\quad \text{und mit einem Nachbarn von 1 endet} \\ &\Leftrightarrow \quad G' \text{ hat einen Weg der Länge } n+2. \end{aligned}$$

Die letzte Äquivalenz folgt, da ein Weg der Länge $n+2$ jeden Knoten in G' durchläuft. Da 0 und $0'$ jeweils nur einen Nachbarn besitzen, muss dieser Weg in 0 (bzw. $0'$) beginnen und in $0'$ (bzw. 0) enden. Wenn wir 0 und $0'$ entfernen, sind 1 und $1'$ die neuen Endpunkte eines Weges der Länge n. Wenn wir jetzt auch noch Knoten $1'$ entfernen, haben wir die Äquivalenz nachgewiesen. □

Das *Hamiltonsche Kreis-Problem* für gerichtete Graphen (*DHC*) ist das einfachste Problem der obigen Gruppe. Wir zeigen jetzt, dass *DHC* tatsächlich schon ein sehr komplexes Problem ist und als Konsequenz ist jedes Problem der Gruppe NP-vollständig.

Satz 6.7 $3\text{–}SAT \leq_{\mathrm{p}} DHC$

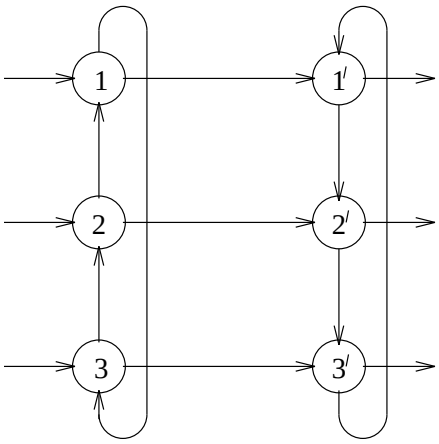
Beweis: Die Formel $\alpha \equiv \alpha_1 \wedge \ldots \wedge \alpha_r$ sei gegeben, wobei die Klausel α_i die Form

$$\alpha_i \equiv l_{i,1} \vee l_{i,2} \vee l_{i,3}$$

habe. Wir nehmen an, dass genau die Variablen $x_1, \ldots, x_n$ (bzw. ihre Negationen) in α vorkommen. Wir werden einen gerichteten Graphen $G(\alpha)$ in polynomieller Zeit konstruieren, so dass

$$\alpha \text{ ist erfüllbar} \quad \Leftrightarrow \quad G(\alpha) \text{ besitzt einen Hamiltonschen Kreis.}$$

$G(\alpha)$ besteht aus den n Variablenknoten $x_1, \ldots, x_n$ und r „Klauselgraphen“ $G_1, \ldots, G_r$, wobei G_i der Klausel α_i entspricht. Die Klauselgraphen sind Kopien des Graphen



Die restlichen Knoten und Kanten von G sind wie folgt einzufügen:

- Jeder Variablenknoten x_i hat zwei ausgehende Kanten. Wir bezeichnen die eine Kante als positiv und die andere als negativ. Die positive Kante wird auf die erste Klausel (d. h. auf den ersten Klauselgraphen) gesetzt, in der (in dem) x_i vorkommt. Die negative Kante wird entsprechend auf die erste Klausel gesetzt, in der $\neg x_i$ vorkommt.

  Wir müssen klären, über welche der drei Kanten (zum Beispiel) x_i mit der Klausel verbunden wird: Wir wählen die erste (zweite, dritte) Kante der Klausel, wenn x_i das erste (zweite, dritte) Literal der Klausel ist.

  Wir sagen, dass die gewählte Kante dem Literal x_i gehört. Die entsprechende ausgehende Kante des Klauselgraphen ordnen wir ebenfalls x_i zu.

- Angenommen, die erste (bzw. zweite oder dritte) Kante, die in einen Klauselgraphen eintritt, stammt von dem Literal x_i. Dann führt die ausgehende und ebenfalls zu x_i gehörende Kante auf die nächste Klausel, in der x_i vorkommt; die in die nächste Klausel eintretende Kante gehört natürlich auch zu x_i.

  Gibt es keine solche Klausel, wird die Kante stattdessen auf x_{i+1} gerichtet (auf x_1, wenn $i = n$). Somit besitzt jeder Variablenknoten x_i auch zwei eingehende (positive oder negative) Kanten.

Jeder Hamiltonsche Kreis in $G(\alpha)$ wird entweder die positive oder die negative Kante eines Variablenknotens x_i durchlaufen. Damit „entscheidet“ ein Kreis eine Belegung der Variablen.

Annahme: Ein Hamiltonscher Kreis in G verlässt jeden Klauselgraphen genau mit dem Kantentyp mit dem der Klauselgraph erreicht wird.

Dann können wir wie folgt fortfahren: Sei K ein Hamiltonscher Kreis. Wir verfolgen K beginnend mit x_1. Angenommen, K wählt die negative Kante von x_1. Nach unserer Annahme muss K nun alle Klauseln nacheinander durchlaufen, in denen $\neg x_1$ vorkommt. Versuchsweise setzen wir den Wahrheitswert von x_1 auf falsch. Wenn K die letzte Klausel von $\neg x_1$ erreicht hat, muss mit dem Knoten x_2 fortgefahren werden. Auch diesmal werden nacheinander alle Klauseln durchlaufen, in denen x_2 vorkommt, wenn K die positive Kante von x_2 wählt. Versuchsweise setzen wir in diesem Fall den Wahrheitswert von x_2 auf wahr.

Wenn K schließlich zu x_1 zurückkehrt, sind alle Klauseln durchlaufen. Und damit müssen die von uns definierten Wahrheitswerte die Formel α erfüllen! Also

$$G(\alpha) \in DHC \quad \Longrightarrow \quad \alpha \in 3\text{–}SAT.$$

Umgekehrt möge die Belegung B die Formel α erfüllen. Wir wählen zur Konstruktion eines Hamiltonschen Kreises die positive (negative) Kante von x_1, wenn $B(x_1) =$ wahr (falsch). Wenn x_1 das einzige wahre Literal einer Klausel ist, wählen wir den Weg

$$1 \to 3 \to 2 \to 2' \to 3' \to 1'$$

Wenn auch das dritte Literal (zum Beispiel) die Klausel erfüllt, wählen wir die Wege

$$1 \to 1'$$

für x_1 und

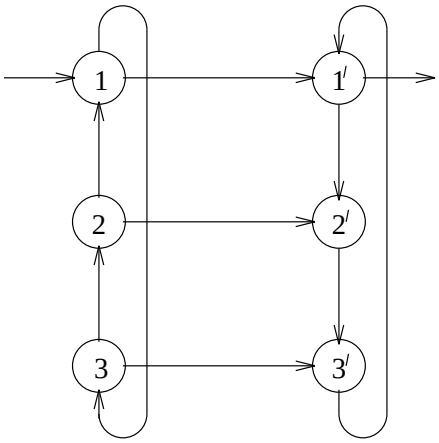
$$3 \to 2 \to 2' \to 3'$$

für das dritte Literal. (Die weiteren Fälle sind analog zu betrachten.)

Wir erhalten auf diese Weise einen Hamiltonschen Kreis, denn jede Klausel wird von mindestens einer Variable erfüllt. Also folgt

$$\alpha \in 3\text{–}SAT \quad \Rightarrow \quad G(\alpha) \in DHC.$$

Bis auf den Nachweis der Annahme ist unser Beweis vollständig. Den Nachweis der Annahme überlassen wir dem Leser, untersuchen aber beispielhaft den Fall, dass eine Klausel nur von seiner ersten Variablen durchlaufen wird:



Der Weg muss zwangsläufig

$$1 \to 3 \to 2 \to 2' \to 3' \to 1'$$

sein, da die Wahl anderer Kanten zur Auslassung von Knoten führt. □

Aufgabe 63
Wir betrachten das *Hamiltonsche Pfad-Problem HP*. Die Eingabe ist ein ungerichteter Graph $G = (V, E)$. Es wird gefragt, ob G einen *Hamiltonschen Pfad* besitzt, dies ist ein Pfad der Länge $|V| - 1$, der jeden Knoten genau einmal berührt:

$$HP = \{G \mid G \text{ besitzt einen Hamiltonschen Pfad }\}.$$

Zeige, dass $HC \leq_p HP$ gilt.

6.3 NP-Vollständigkeit in Anwendungen*

Wir haben bereits die NP-Vollständigkeit diverser Probleme nachgewiesen. In diesem Abschnitt wollen wir an einigen Beispielen aufzeigen, wo in der Praxis NP-vollständige Probleme anzutreffen sind.

6.3.1 Bioinformatik

Wir stellen eine kleine Auswahl NP-vollständigen Probleme in der Bioinformatik vor.

Mehrfaches Alignment

Wir haben das *paarweise Alignment*, also das Alignment von zwei Sequenzen in Kapitel 4.3.3 untersucht und Algorithmen der Laufzeit $O(n \cdot m)$ für Sequenzen der Längen n und m entworfen. Im Problem des *mehrfachen Alignment* betrachtet man viele Sequenzen und möchte ein optimales Alignment bestimmen. Die Frage, ob ein optimales Alignment einer bestimmten Mindestqualität erreicht werden kann, ist leider im Gegensatz zum paarweisen Alignment NP-vollständig.

Faltung von Proteinen

Man bezeichnet die Kette der Aminosäuren (oder den String aus den 20 „Aminosäurenbuchstaben“) als die *Primärstruktur* des Proteins. Das zweidimensionale Faltungs- oder Windungsverhalten des Moleküls wird als *Sekundärstruktur* bezeichnet; die Sekundärstruktur, wie auch höhere Strukturen, ist im wesentlichen eine Funktion der Primärstruktur, da bestimmte Aminosäuren Wasserstoffbindungen, Schwefelbrücken oder elektrostatische Verbindungen eingehen und damit Faltungen des eindimensionalen Strings produzieren. Die *Tertiärstruktur* des Proteins beschreibt die dreidimensionale Struktur des Proteins als Konsequenz der Interaktion von Aminosäuren an verschiedenen Positionen der Kette. Die Tertiärstruktur besitzt einen entscheidenden Einfluss auf die chemische Aktivität und damit auf die Funktion des Moleküls. Die drei-dimensionale Struktur besitzt Vertiefungen und Ausbuchtungen und erlaubt deshalb chemische Reaktionen nur mit drei-dimensional „komplementären“ Molekülen.

Die 3-dimensionale Struktur ist somit wesentlich für das Reaktionsverhalten und damit für die Funktionalität des Proteins. Umso spannender ist die Frage, ob sich die Tertiärstruktur eines Proteins aus der Primärstruktur berechnen lässt.

Das HP-Modell ist ein erster solcher Versuch. Man unterscheidet hydrophile („wasserfreundliche“) und hydrophobe („wasserfeindliche“) Aminosäuren. Die biologische Grundlage dieses Modells ist die Annahme, dass ein „hydrophober Kollaps“ ein Hauptantrieb für die Faltung des Proteins ist. In diesem Kollaps bewegen sich die hydrophoben Aminosäuren nach innen, während die hydrophilen Aminosäuren am Rand verweilen. Nach dieser Annahme versucht sich das Protein so zu falten, dass der Kontakt zwischen hydrophilen und hydrophoben Komponenten minimiert wird.

In einer Modellierung des Faltungsproblems wird angenommen, dass die Aminosäuren auf den Knoten eines Gitters auszulegen sind. Das **kubische Gitter** besitzt die Knotenmenge $\mathbb{Z}^3$ und hat Kanten der Form $\{v, v + e_i\}$ für $v \in \mathbb{Z}^3$ und $e_i \in \{(1,0,0),(0,1,0),(0,0,1)\}$. Eine zweite Variante ist das **triangulierte** Gitter. (Das zwei-dimensionale triangulierte Gitter erhält man durch Verschieben des gleichseitigen Dreiecks mit Seitenlänge Eins, während im drei-dimensionalen triangulierten Gitter das gleichseitige Tetraeder (mit Seitenlänge Eins) zu verschieben ist.) Für beide Gitter modellieren wir ein Protein durch den binären String der hydrophoben (Bit Null) und der hydrophilen (Bit Eins) Komponenten.

Definition 6.8 *Sei S ein binärer String der Länge m.*

(a) *Eine Auslegung von S auf einem Gitter Γ ist ein Weg $W : \{1,\ldots,m\} \to \Gamma$, so dass $W(i)$ und $W(i+1)$ für alle i $(1 \le i < m)$ im Gitter benachbart sind. Kein Knoten des Gitters darf mehr als einmal durchlaufen werden.*

(b) *W liefert einen Kontakt zwischen i und j, falls*

- *$|i-j| > 1$ und $W(i)$ und $W(j)$ benachbart sind und*
- *$S_i = S_j = 0$.*

(c) *Die freie Energie einer Auslegung W ist die negative Anzahl der Kontakte von W.*

(d) *Gesucht ist eine Auslegungen mit minimaler freier Energie, d.h. eine Auslegung mit maximaler Anzahl von Kontakten.*

Die Frage nach der Existenz von Auslegungen mit beschränkter freier Energie ist ebenfalls NP-vollständig. Diese Aussage gilt für das kubische Gitter wie auch für das 3-dimensionale

triangulierte Gitter. Diese Charakterisierung verheißt nichts Gutes, denn Proteine wissen wie sie sich zu falten haben: Ein entfaltenes Protein wird sich innerhalb weniger Sekunden in seine ursprüngliche 3-dimensionale Faltung zurückbegeben! Mithin deutet die NP-Vollständigkeit daraufhin, dass das Auslegungsproblem keine erfolgreiche Modellierung darstellt.

Phylogenetische Bäume

In der Phylogenetik versucht man die genetischen Beziehungen zwischen verschiedenen, vorgegebenen Arten zu bestimmen. Insbesondere versucht man, einen *phylogenetischen Baum* zu konstruieren, der der die wechselseitigen genetischen Beziehungen möglichst gut erklärt: Die Blätter des Baums sind mit jeweils einer Art zu beschriften, während die inneren Knoten Vorgängerarten entsprechen.

Natürlich ist die Bewertung eines phylogenetischen Baums hier das zentrale Problem. In dem Maximum-Parsimony Ansatz nimmt man an, dass verschiedenste Eigenschaften (wie etwa die Größe oder die Aminosäure, die an einer bestimmten Position des Genoms des Organismus vorkommt) für alle Arten bestimmt werden können. Für jede Kante e des Baums wird dann die Distanz $d(e)$ zwischen den Eigenschaftsvektoren der beiden Endpunkte bestimmt und ein Baum $B = (V, E)$ mit minimalem Distanzwert $\sum_{e \in E} d(e)$ ist zu bestimmen. Auch diesmal ist die Frage, ob ein phylogenetischer Baum existiert, dessen Distanzwert einen vorgegebenen Schwellenwert nicht überschreitet, NP-vollständig. (Im konventionellen Maximum Parsimony Problem wird die Distanz zwischen zwei Eigenschaftsvektoren durch den Hammingabstand, also die Anzahl unterschiedlicher Positionen, definiert.)

6.3.2 VLSI Entwurf

Beim Entwurf von VLSI Schaltkreisen ist man stets bemüht, nicht mehr Platz (Bausteine) als nötig zu verwenden. Jedes unnötige zusätzliche Gatter nimmt wertvollen Platz ein und erhöht die Produktionskosten. Schaltungen sind heute jedoch so komplex, dass man weite Teile der Bausteinoptimierung dafür gebauten Tools überlassen möchte.

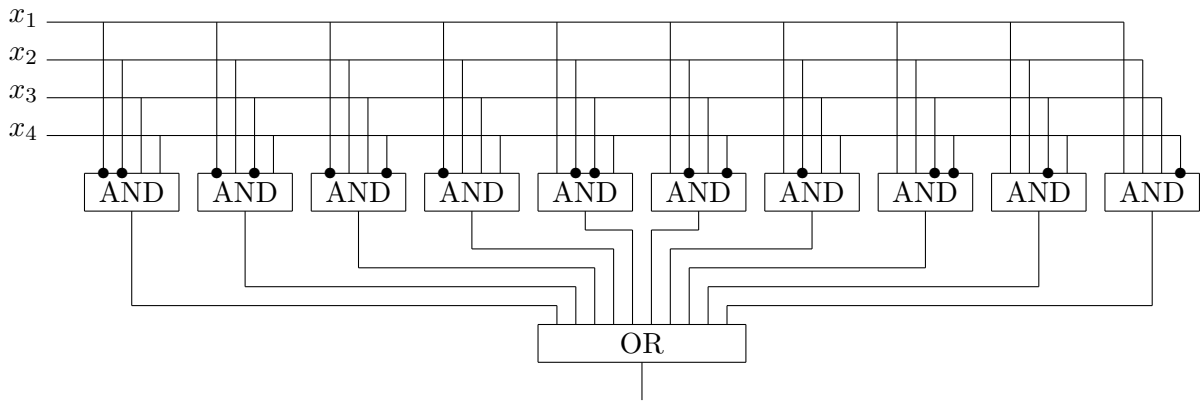
Beispiel 6.4 Die boolsche Funktion

$$f(x_1, x_2, x_3, x_4) = \begin{cases} 1 & \text{falls 2 oder 3 Eingänge logisch 1 sind} \\ 0 & \text{sonst} \end{cases}$$

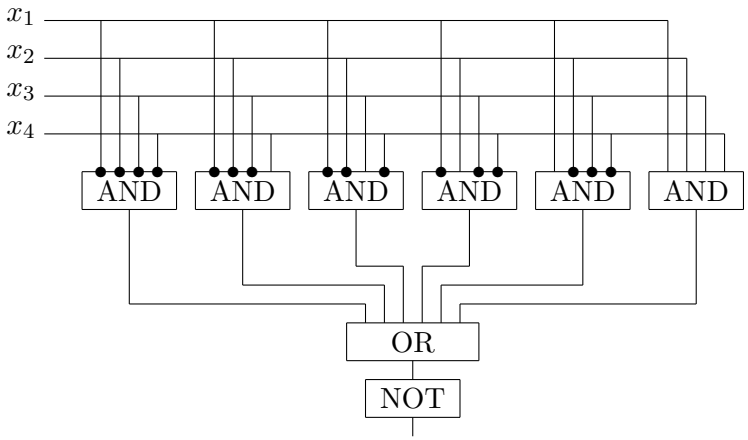
sei zu realisieren. Wir haben $2^4 = 16$ mögliche Eingangskombinationen. Von denen sollen 10 eine 1 und 6 eine 0 am Ausgang liefern.

x_1	x_2	x_3	x_4	$f(x_1, x_2, x_3, x_4)$
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

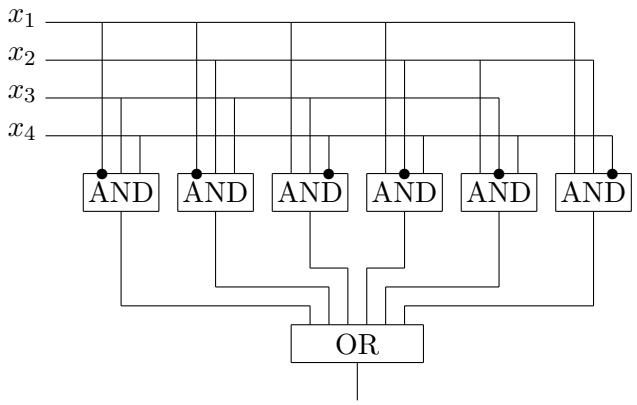
Ein erster Ansatz könnte nun sein, diese Wahrheitstafel als disjunktive Normalform direkt in eine Schaltung umzusetzen. Ein Punkt an einem Gattereingang bedeutet, dass eine Negation vorgeschaltet wird.



Diese Schaltung benötigt also 10 vierstellige ANDs, ein zehnstelliges OR und 16 NOTs. Eine bessere Lösung erhalten wir, wenn wir die 6 negativen Fälle abfragen und das Resultat negieren.



Hier brauchen wir 6 vierstellige ANDs, ein sechsstelliges OR und 17 NOTs. Die folgende Schaltung kommt mit 6 dreistelligen ANDs, einem sechsstelligen OR und 6 NOTs aus.



Ist die letzte Schaltung optimal?

Leider gibt es keine effizienten Tools zur Schaltkreisminimierung, da schon die Entscheidungsversion *„Gibt es einen äquivalenten Schaltkreis mit höchstens k Gattern?“* hart ist, wie wir gleich sehen werden.

Wir betrachten Schaltkreise mit $\wedge, \vee, \neg$ Gattern von unbeschränktem Fan-in. (Der Fan-in eines Gatters ist die Anzahl eingehender Drähte). Gatter mit konstanter Ausgabe 0 oder 1 sind ebenfalls erlaubt, werden aber wegen ihrer Einfachheit in der Bestimmung der Größe eines Schaltkreises nicht gezählt.

Wir betrachten das Entscheidungsproblem

$$MIN = \{\ (S,k) \quad | \quad \text{Es gibt einen zu } S \text{ äquivalenten } \{\wedge, \vee, \neg\}\text{ -Schaltkreis mit höchstens } k \text{ Gattern }\}.$$

Satz 6.9 $\overline{KNFSAT} \leq_p MIN$.

$\overline{KNFSAT}$ ist das Entscheidungsproblem aller nicht-erfüllbaren Formeln in konjunktiver Normalform.

Beweis: Einer vorgegebenen KNF-Formel α weisen wir einen Schaltkreis S_α wie folgt zu:

(a) Unsere Schaltung besitzt für jede Variable einen Eingang.

(b) Für jede Klausel existiert ein OR Gatter mit so vielen Eingängen wie Literale in der Klausel enthalten sind.

(c) Jeder Eingang wird nun mit den Klausel-Gattern verbunden, in denen die betreffende Variable auftaucht. Liegt sie negiert vor, so wird ein Negationsgatter vorgeschaltet.

(d) Ein AND-Gatter, dessen Fanin der Zahl der Klauseln entspricht, erhält die Ausgänge der OR-Gatter als Eingänge.

(e) Der Ausgang dieses Gatters ist auch der Ausgang des Schaltkreises.

Die Schaltung S_α ist in polynomieller Zeit erstellbar and wir transformieren α auf $(S_\alpha, 0)$. Falls α *nicht* erfüllbar ist, so ist unsere Schaltung nur eine komplizierte Darstellung der logischen 0 und $(S_\alpha, 0)$ gehört zu MIN.

Ist α erfüllbar, so wird mindestens ein Gatter benötigt, da eine Formel in konjunktiver Normalform keine Tautologie sein kann, und $(S_\alpha, 0)$ gehört nicht zu MIN. □

In der Praxis muss man also damit leben, dass man Schaltungen mittels *Approximationsalgorithmen* nur annähernd minimiert oder, da wo Zeit und Problemgröße es zulassen, *von Hand* minimiert.

Aufgabe 64
Wir beschäftigen uns hier mit einem anscheinend schwächeren Problem: der Schaltkreisoptimierung von $\vee$-Schaltkreisen mit Fan-in 2.

$\vee$-*SK*: Für eine Variablenmenge U ist eine Menge $C = \{c_1, \ldots, c_m\}$ von Klauseln mit ausschließlich positiven Literalen über U gegeben. Für eine natürliche Zahl B ist zu entscheiden, ob es einen $\vee$-Schaltkreis mit $\leq B$ Bausteinen vom Fan-in 2 gibt, der sämtliche Klauseln aus C berechnet.

Wir beschreiben die polynomielle Reduktion von VC auf $\vee$-SK: Der ungerichtete Graph $G = (V, E)$ sowie die Konstante k seien die Eingabe für VC. Die Knotenmenge von G sei $V = \{v_1, \ldots, v_n\}$. Wir definieren als Eingabe für $\vee$-SK die Variablenmenge $U = \{u_0, u_1, \ldots, u_n\}$. Unsere Reduktion ordnet einem Knoten v_i die Variable u_i zu und einer Kante $\{v_i, v_j\}$ die Klausel $u_0 \vee u_i \vee u_j$. Schließlich wird der Schranke k die Schranke $B = k + |E|$ zugeordnet. Natürlich kann (U, C, B) in polynomieller Zeit berechnet werden.

Zeige: G besitzt einen Vertex Cover der Größe $\leq k \Leftrightarrow$ Es gibt einen $\vee$-Schaltkreis mit höchstens B Bausteinen vom Fan-in 2, der alle Klauseln aus C berechnet.

Eine große Hilfe bei manuellen Versuchen, eine Schaltung zu minimieren, wäre ein Algorithmus, der entscheidet, ob zwei gegebene Schaltkreise dasselbe tun, also dieselbe Funktion berechnen. Mit einem solchen Algorithmus lassen sich die einzelnen Schritte einer manuellen oder rechnergestützten Optimierung verifizieren. Leider gilt:

Satz 6.10 *Das Problem zu entscheiden, ob zwei vorgelegte Schaltkreise verschiedene Funktionen berechnen, ist* NP*-vollständig.*

Beweis: Diesmal können wir *KNF–SAT* auf das Problem der Inäquivalenz reduzieren. Dazu transformiere eine Formel α auf das Paar $(S_\alpha, Null)$, wobei S_α der in Satz 6.9 konstruierte Schaltkreis für α ist und der Schaltkreis $Null$ die konstante Ausgabe 0 hat. Offensichtlich ist α genau dann erfüllbar, wenn die Schaltkreise S_α und $Null$ verschiedene Funktionen berechnen.

Die Zugehörigkeit zu NP läßt sich wie folgt einsehen: Wir erraten eine Belegung, bei der die Schaltkreise unterschiedliche Werte am Ausgang annehmen und verifizieren, dass dem so ist. □

Bemerkung: Natürlich ist es möglich, die logische Äquivalenz zweier Schaltungen nachzuweisen, indem man alle möglichen Eingangskombinationen simuliert. Deren Anzahl ist jedoch exponentiell in der Zahl der Eingänge.

6.3.3 Betriebssysteme

Eine wesentliche Aufgabe von Betriebssystemen ist das Ressourcenmanagement. Man kann hier an verschiedene Prozesse denken, die um die Rechenzeit eines (oder sogar mehrerer) Prozessoren konkurrieren.

Das Problem einer Druckerwarteschlange für zwei Drucker, das wir im folgenden betrachten, steht nur stellvertretend für eine große Familie von Problemen ähnlicher Natur. Gegeben sind zwei Drucker mit gleicher Geschwindigkeit. Für sie werden n Druckjobs mit jeweiliger Druckzeit t_i, $1 \leq i \leq n$, abgesetzt. Wir nehmen an, dass es sich bei den Zeiten um natürliche Zahlen handelt. Ziel ist es, alle Aufträge schnellstmöglich zu erledigen. Dazu ist es notwendig, die Zahlen t_i *möglichst gut* aufzuteilen.

Formal: Gesucht ist eine Teilmenge $I \subseteq \{1, \ldots, n\}$ so dass

$$|\sum_{i \in I} t_i - \sum_{i \in \{1,\ldots,n\} \setminus I} t_i|$$

minimal wird. Wir wollen dieses Problem $OPT_2 - PARTITION$ nennen.

Wir zeigen in Kürze, dass bereits das $PARTITION$ Problem NP-vollständig ist. Hier wird danach gefragt, ob sich die Indizes i in zwei Mengen I und $\{1, \ldots, n\} \setminus I$ aufteilen lassen, so dass

$$\sum_{i \in I} t_i = \sum_{i \in \{1,\ldots,n\} \setminus I} t_i$$

gilt.

Aufgabe 65
Wir definieren das Rucksackproblem **RUCKSACK:**
Gegeben: Ein Rucksack und n Objekte mit Gewichten $g_1, \ldots, g_n$ sowie eine Gewichtsschranke G. Zusätzlich seien $a_1, \ldots, a_n$ die Nutzwerte für die Objekte.
Frage: Gibt es zu gegebenem Nutzwert A eine Bepackung des Rucksackes, die das Gewichtsschranke nicht überschreitet und mindestens den Nutzen A erreicht ?

Konstruiere eine polynomielle Reduktion von PARTITION auf RUCKSACK.

Wir sehen sofort, dass ein effizienter Algorithmus zur Lösung von $OPT_2 - PARTITION$ das $PARTITION$ Problem effizient entscheidet. Aus der NP-Vollständigkeit von $PARTITION$ können wir also folgern, dass wohl kein solcher Algorithmus existiert.

Als drittes betrachten wir das $SUBSET-SUM$ Problem: Gegeben ist eine Menge natürlicher Zahlen $t_1, \ldots, t_n$ und ein Zielwert $Z \in \mathbb{N}$. Die Frage lautet, ob eine Teilmenge I existiert, so dass

$$\sum_{i \in I} t_i = Z$$

gilt.

Lemma 6.11

$$SUBSET - SUM \leq_p PARTITION.$$

Beweis: Sei $t_1, \ldots, t_n, Z$ eine Eingabe für $SUBSET - SUM$. Wir setzen $T := \sum_i^n t_i$.

Wir transformieren die Eingabe $t_1, \ldots, t_n, Z$ auf die Eingabe $t_1 \ldots, t_n, (T - Z + 1), (Z + 1)$ für $PARTITION$. Die Summe all dieser Werte ist $T + (T - Z + 1) + (Z + 1) = 2T + 2$. Wir behaupten nun, dass diese Werte genau dann zu $PARTITION$ gehören, wenn die Werte $t_1, \ldots, t_n$ eine Teilmenge mit Summe Z besitzen.

Existiert eine Zweiteilung im Sinne von $PARTITION$, so ist eine Zerlegung in zwei Teilmengen mit der jeweiligen Summe $T+1$ möglich. Wir wissen, dass die beiden Elemente $(T-Z+1)$ und $(Z+1)$ dann in verschiedenen Teilmengen liegen müssen. Die t_i-Werte ergänzen die beiden Mengen dann zu $T+1$. Der Menge, die (T-Z+1) enthält, müssen also t_i-Werte im Umfang von genau Z hinzugefügt worden sein. Diese Teilmenge ist eine Lösung des $SUBSET-SUM$ Problems.

Existiert andererseits unter den t_i eine Teilmenge mit Summe Z, so impliziert diese Teilmenge zusammen mit $(T - Z + 1)$ eine Lösung für $PARTITION$. □

Wir reduzieren nun $3SAT$ auf $SUBSET - SUM$.

Satz 6.12

$$3SAT \leq_p SUBSET - SUM.$$

Die Probleme PARTITION und SUBSET_SUM sind also NP*-vollständig.*

Beweis: Sei α eine 3-KNF Formel mit $\alpha = \alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_m$, wobei $\alpha_i = (l_{i,1} \vee l_{i,2} \vee l_{i,3})$ mit $l_{i,j} \in \{x_1, \bar{x_1}, x_2, \bar{x_2}, \ldots, x_n, \bar{x_n}\}$. Wir geben nun die Wertemenge und den Parameter Z des $SUBSET - SUM$ Problems in Dezimaldarstellung an. Die resultierenden Zahlen sind zwar von exponentieller Größe, können jedoch in polynomieller Zeit in Dezimaldarstellung aufgeschrieben werden. Zuerst setzen wir

$$Z = \underbrace{4\,4\ldots4}_{\text{m mal}}\underbrace{1\,1\ldots1}_{\text{n mal}}$$

Die Menge, für die eine Teilmenge mit der Summe Z gefunden werden soll, hat $2n + 2m$ Elemente und ist von der folgenden Form:

$$\{pos_1, \ldots, pos_n, neg_1, \ldots, neg_n, small_1, \ldots, small_m, big_1, \ldots, big_m\}$$

Wir geben nun an, wie diese Zahlen aussehen. Wir stellen alle Zahlen mit $n + m$ Ziffern dar und stören uns nicht an führenden Nullen. Die Ziffern jeder Zahl teilen wir in zwei Blöcke: einen linken, m Ziffern breiten und einen rechten, n Ziffern breiten Block.

- Die Zahl pos_i enthält Informationen über das Literal x_i. Im linken (m Stellen breiten) Ziffernblock enthält pos_i an Stelle j eine 1, falls x_i in Klausel j auftaucht. Im rechten (n Stellen breiten) Ziffernblock steht an Position i eine 1. Sonst sind alle Ziffern 0.
- Die Zahl neg_i enthält Informationen über das Literal $\bar{x_i}$. Im linken (m Stellen breiten) Ziffernblock enthält neg_i an Stelle j eine 1, falls $\bar{x_i}$ in Klausel j auftaucht. Im rechten (n Stellen breiten) Ziffernblock steht an Position i eine 1. Sonst sind alle Ziffern 0.
- Die Zahl $small_i$ enthält nur an der Stelle i im ersten Ziffernblock eine 1. Sonst sind alle Ziffern 0. Auch im rechten Block sind alle Ziffern 0.
- $big_i := 2 \cdot small_i$

Diese Konstruktion verdeutlichen wir an einem Beispiel. Sei α die Konjunktion

$$\begin{array}{cccccccccc} & (x_1 \vee x_3 \vee x_4) & \wedge & (x_1 \vee x_2 \vee \bar{x_4}) & \wedge & (\bar{x_1} \vee \bar{x_2} \vee x_3) & \wedge & (\bar{x_2} \vee \bar{x_3} \vee \bar{x_4}) \\ \wedge & (x_1 \vee \bar{x_3} \vee x_4) & \wedge & (\bar{x_1} \vee x_2 \vee x_3) & \wedge & (\bar{x_1} \vee x_2 \vee \bar{x_3}) & & \end{array}$$

Dann erhalten wir die folgenden Werte:

pos_1	1100100	1000	neg_1	0010011	1000
pos_2	0100011	0100	neg_2	0011000	0100
pos_3	1010010	0010	neg_3	0001101	0010
pos_4	1000100	0001	neg_4	0101000	0001
$small_1$	1000000	0000	big_1	2000000	0000
$small_2$	0100000	0000	big_2	0200000	0000
$small_3$	0010000	0000	big_3	0020000	0000
$small_4$	0001000	0000	big_4	0002000	0000
$small_5$	0000100	0000	big_5	0000200	0000
$small_6$	0000010	0000	big_6	0000020	0000
$small_7$	0000001	0000	big_7	0000002	0000

Der Zielwert Z ist 4444444 1111. Wir beachten, dass die Summe aller Werte 6666666 2222 ist. Egal wie wir hier eine Teilmenge auswählen, wir erhalten bei der Addition keine Überträge. Das ist wesentlich, da wir so die einzelnen Stellen unseres Zielwertes Z gesondert betrachten können.

Untersuchen wir zunächst die n (hier $n = 4$) hinteren Stellen. Bei unserem Zielwert sollen hier Einsen stehen. Die können wir nur dann erreichen, wenn wir für jedes $i \in \{1, \ldots, n\}$ entweder pos_i oder neg_i auswählen. (Dies entspricht der Zuweisung von Wahrheitswerten.)

Die m (hier $m = 7$) vorderen Stellen entsprechen den Klauseln. Durch die Auswahl von *pos* und *neg* Zahlen zu unserer Teilmenge werden für jede Klausel Beiträge aufaddiert. Wir erhalten Werte zwischen 0 und 3, je nachdem wieviele Literale aus der Klausel durch unsere Auswahl auf `wahr` gesetzt werden. Die Werte 1,2 und 3 können durch geeignete Auswahl der *small* und der *big* Zahlen auf den Sollwert 4 gebracht werden. Ist jedoch eine Position nach Wahl der *pos* und *neg* Zahlen nach wie vor auf 0 (das entspricht einer Belegung, bei der kein Literal der Klausel erfüllt ist), so kann die 4 in dieser Position nicht mehr erreicht werden.

Besitzt die Formel eine erfüllende Belegung, so impliziert diese Belegung eine Teilmenge mit Summe Z. Andererseits beschreibt jede Teilmenge mit Summe Z eine erfüllende Belegung. □

Unsere Formel α wird zum Beispiel durch die Belegung x_1, x_3 `falsch` und x_2, x_4 `wahr` erfüllt. Wir wählen also zunächst $neg_1, pos_2, neg_3, pos_4$ aus und erhalten die Zwischensumme:

$$\begin{array}{rr} & 00100111000 \\ + & 01000110100 \\ + & 00011010010 \\ + & 10001000001 \\ \hline = & 11112231111 \end{array}$$

Aus den ersten 4 Klauseln ist jeweils ein Literal wahr, aus der 5. und 6. zwei, in der 7. Klausel sind sogar alle 3 Literale erfüllt. Durch die Auswahl von $small_1, \ldots, small_4$, $big_1, \ldots, big_4$, big_5,big_6, $small_7$ komplettieren wir unsere Teilmenge.

$$\begin{array}{rr} & 00100111000 \\ + & 01000110100 \\ + & 00011010010 \\ + & 10001000001 \\ + & 10000000000 \\ + & 01000000000 \\ + & 00100000000 \\ + & 00010000000 \\ + & 20000000000 \\ + & 02000000000 \\ + & 00200000000 \\ + & 00020000000 \\ + & 00002000000 \\ + & 00000200000 \\ + & 00000010000 \\ \hline = & 44444441111 \end{array}$$

Bemerkung: Die Probleme gehören auch zur Klasse `NP`, da wir die zu bestimmenden Teilmengen raten und anschließend effizient verifizieren können. Unser ursprüngliches Problem der optimalen Zweiteilung läßt sich also nicht effizient lösen, es sei denn dass `P=NP`.

6.3.4 Datenbanken

Im Umgang mit Datenbanken stößt man häufig auf Probleme, die mit Mengen und Mengensystemen zu tun haben. Wir erinnern deshalb zuerst an das Problem SC der Mengenüberdeckung. Gegeben ist ein endliches Universum U und eine natürliche Zahl k. Außerdem sind Teilmengen $A_1, A_2, \ldots, A_m$ mit $A_1 \ldots A_m \subseteq U$ und $\bigcup_{i=1}^{m} A_i = U$ gegeben. Es wird gefragt, ob U bereits von k der m Teilmengen überdeckt werden kann. Beachte, dass SC nach Satz 6.4 NP-vollständig ist.

Wenden wir uns nun einem Problem aus dem Bereich der Datenbanken zu. Wir betrachten das folgende, vereinfachte Modell einer Datenbank.

- Die Datenbank enthält *Attribute*. Das sind Kategorien wie *Name, Straße, Hausnummer, Vorname, Kontonummer,....* Man kann hier an die Spaltenüberschriften einer Tabelle denken.
- Die Datenbank enthält Datensätze. Diese entsprechen den Eintragungen in eine solche Tabelle, etwa *Meier, Hauptstr., 42, Otto, 0815-4711.*

Für das Arbeiten mit einer Datenbank ist es nun unentbehrlich, einen konkreten Datensatz exakt spezifizieren zu können. Hier kommt der Begriff Key in Spiel. Ein Key ist eine Teilmenge der Attribute, so dass alle Datensätze anhand dieser Teilmenge unterschieden werden können.

Es genügt beispielsweise häufig nicht, allein den Nachnamen als Schlüssel zu wählen. Ein Blick ins Telefonbuch zeigt auch, dass die Kombination Vor- und Nachname noch nicht zwingend eine Person eindeutig festlegt.

Ein wichtige Aufgabe ist nun, eine möglichst kleine Attributmenge zu finden, die als Key dienen kann. Oder als Entscheidungsproblem formuliert: „Gibt es für die Datenbank einen Key bestehend aus höchstens k Attributen?“ Nennen wir dieses Problem: $KEY - FIND$.

Satz 6.13 *Das $KEY - FIND$ Problem ist NP-vollständig.*

Beweis: Die Zugehörigkeit zu NP ist klar. Wir raten eine Teilmenge der Attribute und verifizieren anschließend die Key-Eigenschaft sowie die Einhaltung der Größenschranke k.

Wir zeigen:

$$SC \leq_p KEY - FIND$$

Sei $U = \{1, 2, \ldots, n\}$ das Universum und $A_1, \ldots, A_m$ das Mengensystem für SC. Sei k die erlaubte Größe der Überdeckung. Wir definieren nun eine Datenbank:

- Wir richten für jede Menge A_i ein Attribut ein. Der Wertebereich der Attribute sind jeweils die ganzen Zahlen.
- Für jedes $i \in U$ schaffen wir zwei Datensätze d_i und d_i'. Diese beiden Datensätze erhalten den Eintrag i für Attribut j, wenn i *nicht* in A_j enthalten ist. Gilt aber $i \in A_J$, so erhalten d_i und d_i' die Werte i bzw. $-i$.

Beispiel 6.5 Die Teilmengen mögen von der folgende Form sein:

$$\begin{aligned} A_1 &= \{1, 2, 7\} \\ A_2 &= \{3, 4\} \\ A_3 &= \{1, 5\} \end{aligned}$$

$$\begin{aligned} A_4 &= \{4, 7, 10\} \\ A_5 &= \{5, 6, 7, 8\} \\ A_6 &= \{8, 10, 11, 12\} \\ A_7 &= \{5, 6, 9, 10\} \\ A_8 &= \{2, 6, 9\} \end{aligned}$$

Wir erhalten dann die folgende Datenbank.

	Attr.1	Attr.2	Attr.3	Attr.4	Attr.5	Attr.6	Attr.7	Attr.8
d_1	1	1	1	1	1	1	1	1
d'_1	−1	1	−1	1	1	1	1	1
d_2	2	2	2	2	2	2	2	2
d'_2	−2	2	2	2	2	2	2	−2
d_3	3	3	3	3	3	3	3	3
d'_3	3	−3	3	3	3	3	3	3
d_4	4	4	4	4	4	4	4	4
d'_4	4	−4	4	−4	4	4	4	4
d_5	5	5	5	5	5	5	5	5
d'_5	5	5	−5	5	−5	5	−5	5
d_6	6	6	6	6	6	6	6	6
d'_6	6	6	6	6	−6	6	−6	−6
d_7	7	7	7	7	7	7	7	7
d'_7	−7	7	7	−7	−7	7	7	7
d_8	8	8	8	8	8	8	8	8
d'_8	8	8	8	8	−8	−8	8	8
d_9	9	9	9	9	9	9	9	9
d'_9	9	9	9	9	9	9	−9	−9
d_{10}	10	10	10	10	10	10	10	10
d'_{10}	10	10	10	−10	10	−10	−10	10
d_{11}	11	11	11	11	11	11	11	11
d'_{11}	11	11	11	11	11	−11	11	11
d_{12}	12	12	12	12	12	12	12	12
d'_{12}	12	12	12	12	12	−12	12	12

Sobald wir irgendein Attribut wählen, sind bereits alle Datensatz-Paare (d_i, d_j) (bzw. (d_i, d'_j), (d'_i, d_j) oder (d'_i, d'_j)) für $i \neq j$ unterscheidbar. Die Paare (d_i, d'_i) können aber nur durch Attribute zu solchen Mengen unterschieden werden, die i enthalten.

Offenbar gilt: Es gibt genau dann einen Key mit höchstens k Attributen, wenn die höchstens k zugehörigen Teilmengen das Universum überdecken. Also haben wir SC erfolgreich auf $KEY - FIND$ reduziert. □

6.3.5 Existenz von Gewinnstrategien

Definition 6.14 *Unter einem Spiel verstehen wir hier ein Szenario, das folgende Bedingungen erfüllt.*

(a) Das Spiel ist durch eine endliche Zahl von Konfigurationen beschrieben.

(b) Eine der Konfigurationen ist die Anfangskonfiguration, in der das Spiel beginnt.

(c) Die Spieler führen abwechselnd Züge aus.

(d) Ein Zug ist ein mit den Regeln des Spiels konformer Übergang von einer Konfiguration in eine andere.

(e) Eine Menge von Endkonfigurationen ist ausgezeichnet. Erreicht das Spiel eine Endkonfiguration, so ist es beendet und es läßt sich nachprüfen, welcher der Spieler gewonnen hat bzw., ob ein Unentschieden vorliegt.

(f) Das Spiel nimmt nach endlich langer Zeit eine Endkonfiguration ein.

Beispiel 6.6 Wir betrachten das folgende Spiel für zwei Spieler: Zu Beginn liegen 21 Streichhölzer auf dem Tisch. Ein Zug besteht nun darin, dass ein Spieler 1,2 oder 3 der Hölzer entfernt. Es verliert der Spieler, der das letzte Holz nimmt.

Verifizieren wir kurz, dass es sich hierbei um ein Spiel im Sinne unserer Definition handelt.

(a) Das Spiel kennt 22 Konfigurationen, nämlich die 22 möglichen Anzahlen von Hölzern. Wir nennen diese Konfigurationen $H_0, \ldots, H_{21}$.

(b) Anfangskonfiguration ist H_{21}.

(c) Abwechselnde Züge sind verlangt.

(d) Die in Konfiguration i erlaubten Züge sind

$$\begin{array}{lcll} H_i & \to & H_{i-1} & \text{falls } i \geq 1 \\ H_i & \to & H_{i-2} & \text{falls } i \geq 2 \\ H_i & \to & H_{i-3} & \text{falls } i \geq 3 \end{array}$$

(e) H_0 ist Endkonfiguration. Verlierer ist, wer das Spiel in diese Position gebracht hat.

(f) Offenbar endet das Spiel nach spätestens 21 Zügen.

Lemma 6.15 *Der nachziehende Spieler hat bei diesem Spiel eine Gewinnstrategie.*

Beweis: Wenn Spieler 1 in seinem Zug i Hölzer entfernt, so entfernt Spieler 2 anschließen $4 - i$ Hölzer. Egal wie Spieler 1 gezogen hat, wenn er wieder am Zug ist, befinden sich stets 4 Hölzer weniger auf dem Tisch. Spieler 1 ist also bei $H_{21}, H_{17}, H_{13}, H_9, H_5$ und H_1 am Zug. In Konfiguration H_1 kann Spieler 1 dann nur das letzte Holz entfernen, was für ihn die Niederlage bedeutet. □

Man mache sich klar, dass bei Spielen im Sinne unserer Definition stets genau eine der drei Alternativen gelten muss.

- Spieler 1 hat eine Gewinnstrategie.
- Spieler 2 hat eine Gewinnstrategie.
- Beide Spieler haben eine Strategie, mit der sie ein Unentschieden erzwingen können.

Damit sind eigentlich alle Spiele langweilig. Der eigentliche Charakter eines Spiels wird erst dann deutlich, wenn optimale Strategien schwierig zu finden sind. Im Schach gibt es beispielsweise 20 mögliche Eröffnungszüge. Nachdem jeder Spieler einen Zug gemacht hat, sind bereits 400 verschiedene Stellungen möglich. Die Anzahl der Stellungen, die im Laufe eines Spieles auf einem Schachbrett auftreten können, liegt in der Größenordnung der Zahl der Atome im

Universum. Es ist bis heute nicht bekannt, welche der drei oben genannten Alternativen für Schach zutrifft. [1] Nähern wir uns der Frage nach Gewinnstrategien etwas formaler.

- Spieler x gewinnt, wenn das Spiel in einer für ihn günstigen Endposition ist.
- Spieler x gewinnt, wenn Spieler y am Zug ist und das Spiel *für jeden* möglichen Zug von y in eine Gewinnposition für Spieler x läuft.
- Spieler x gewinnt, wenn Spieler x am Zug ist und *es einen Zug gibt*, so dass *für jeden* möglichen Zug von Spieler y das Spiel in eine Gewinnposition für Spieler x läuft.
- Spieler x gewinnt, wenn Spieler y am Zug ist und *für jeden* Zug von Spieler y *es einen Zug von Spieler x gibt*, so dass *für jeden* möglichen Zug von Spieler y das Spiel in eine Gewinnposition für Spieler x läuft.
- ...

Wir können die Frage nach der Existenz einer Gewinnstrategie also als eine Kette von Quantoren notieren und in die Sprache der Prädikatenlogik übersetzen:

$$\begin{aligned}&\text{Spieler } x \text{ hat eine Gewinnstrategie} \Leftrightarrow\\ &\exists_{Zug\,x_1}\forall_{Zug\,y_1}\exists_{Zug\,x_2}\forall_{Zug\,y_2}\ldots\exists_{Zug\,x_n}\forall_{Zug\,y_n} x \text{ gewinnt}\end{aligned}$$

Dies ist der prinzipielle Aufbau. Wir führen deshalb das Entscheidungsproblem QBF, Quantifizierte Boolesche Formeln, ein. QBF modelliert die Frage nach Gewinnstrategien für einfache Spiele, d.h. Spiele, deren Regeln durch die Aussagenlogik ausgedrückt werden kann:

$$\begin{aligned}QBF = \{\beta \quad | \quad &\beta = \exists_{x_1}\forall_{x_2}\exists_{x_3}\ldots\forall_{x_{n-1}}\exists_{x_n} E(x_1,\ldots x_n),\\ &E \text{ ist eine Formel der Aussagenlogik und } \beta \text{ ist wahr. }\}.\end{aligned} \tag{6.1}$$

Wir stellen fest:

Satz 6.16 *QBF ist* NP*-hart.*

Beweis: Sei α eine Formel in konjunktiver Normalform mit den Variablen $x_1,\ldots,x_n$. Dann ist

$$\beta = \forall_{x'_1}\exists_{x_1}\forall_{x'_2}\exists_{x_2}\ldots\forall_{x'_n}\exists_{x_n}\alpha$$

eine Formulierung von *KNF–SAT*. □

Bemerkung: Der Beweis hat im Wesentlichen nur ausgenutzt, dass das Finden einer optimalen letzten Zugfolge bereits sehr schwierig ist. Und damit ist die Komplexität des 2-Personen Spiels noch gar nicht zum Tragen gekommen.

Man nimmt deshalb auch an, dass QBF aller Wahrscheinlichkeit nach nicht in NP liegt. Man kann nämlich zeigen, dass dieses Problem unter allen auf polynomiellen Platz berechenbaren Problemen ein „schwierigstes“ Problem ist.

Genauer: Die Klasse $PSPACE$ ist gerade die Klasse der auf polynomiellen Platz berechenbaren Probleme, und QBF ist $PSPACE$-vollständig unter polynomiellen Reduktionen. Aussagen wie diese werden in der Vorlesung „Komplexitätstheorie“ behandelt.

[1] Beachte, dass über die Regel von Zugwiederholungen, unendlich lange Spiele verhindert werden. Schach ist also ein Spiel im Sinne unserer Definition.

6.4 Zusammenfassung

Wir haben zuerst den Begriff der *effizienten* deterministischen Berechenbarkeit durch die Definition der Klasse P eingeführt; wir haben also „effizient“ durch „polynomielle Laufzeit“ übersetzt.

Unsere Definition effizienter Berechenbarkeit ist nicht nur auf Turingmaschinen, sondern auch auf Registermaschinen anwendbar, denn Registermaschinen lassen sich in polynomieller Zeit durch Turingmaschinen simulieren. Der Begriff der effizienten Berechenbarkeit führt somit für alle gegenwärtig benutzten Rechnermodelle auf dieselbe Sprachklasse.

Quantenrechner können aber mit der Faktorisierung ein Problem berechnen, das wahrscheinlich nicht in P liegt. Effiziente Berechenbarkeit für Quantenrechner beinhaltet somit wahrscheinlich sehr viel mehr Probleme als die Klasse P.

Das Ziel dieses Kapitels war, mehr über die Struktur nicht-effizient berechenbarer Entscheidungsproblem zu erfahren. Wir haben dazu Probleme betrachtet, die

- nach der Existenz kurzer Lösungen fragen,
- wobei effizient geprüft werden kann, ob potentielle Lösungen auch tatsächliche Lösungen sind.

Wir haben viele Probleme mit dieser Struktur kennengelernt: Das Erfüllbarkeitsproblem, das Cliquen-Problem, das Knotenüberdeckungsproblem, das Problem der Null-Eins-Programmierung, das Hamiltonsche Kreisproblem, das Partitionsproblem, das Traveling-Salesman-Problem usw. Mit Hilfe nichtdeterministischer Turingmaschinen wurde die entsprechende Komplexitätsklasse, nämlich die Klasse NP definiert. Durch den Begriff der polynomiellen Reduktion haben wir die schwierigsten Probleme in NP, nämlich die NP-vollständigen Probleme definiert.

Eine effiziente Lösung irgendeines NP-vollständigen Problems bedingt, dass alle Probleme in NP effizient lösbar sind. Genau aus diesem Grund ist zu vermuten, dass realistische Rechnertechnologien auch in der Zukunft an NP-vollständigen Problemen scheitern werden, da ansonsten *Raten effizient simulierbar ist.* Diese Vermutung gilt auch für Quantenrechner.

Wir haben auch festgestellt, dass jedes Problem aus NP in Zeit $2^{\text{polynom}(n)}$ gelöst werden kann. Es ist nicht unwahrscheinlich, dass die Laufzeit $2^{(n^{\varepsilon})}$ –für irgendeine Konstente $\varepsilon > 0$– auch notwendig ist, um NP-vollständige Probleme zu lösen. Allerdings ist sogar die einfachere Frage, nämlich ob $\mathtt{P} \neq \mathtt{NP}$, seit über 30 Jahren offen.

Wir haben als zentrales Resultat dieses Kapitels die NP-Vollständigkeit des Erfüllbarkeitsproblems *KNF–SAT* nachgewiesen und weitere NP-vollständige Probleme durch polynomielle Reduktionen, ausgehend von *KNF–SAT*, erhalten.

Warum ist eine Betrachtung nicht-effizient berechenbarer Probleme ein wesentlicher Aspekt des Algorithmenentwurfs? Wenn wir wissen oder stark vermuten, dass ein Problem nicht effizient berechenbar ist, dann brauchen wir keine unnötigen Energien verschwenden, sondern können zum Beispiel nach Abschwächungen des Problems Ausschau halten, die effiziente Lösungen erlauben.

Natürlich ist die NP-Vollständigkeit eines Problems kein Grund aufzugeben. Ganz im Gegenteil, die NP-Vollständigkeit macht das Problem eher interessant: Wir können nach approximativen Lösungen suchen oder Algorithmen entwerfen, die zwar nicht für alle, aber doch für viele interessante Eingaben funktionieren. Der nächste Teil des Skripts ist genau dieser Herausforderung gewidmet.

Teil III

Algorithmen für schwierige Probleme

Wir betrachten Optimierungsprobleme der Form

$$\text{opt}_y f(x,y) \text{ so dass } L(x,y),$$

wobei wir $\text{opt} \in \{\min, \max\}$ annehmen. Wir nennen x die Eingabe oder die Beschreibung des Optimierungsproblems. y heißt eine **Lösung** für Eingabe x, wenn y das Prädikat L erfüllt, wenn also $L(x,y)$ zutrifft. Für jede Eingabe x müssen wir die **Zielfunktion** $f(x,y)$ über alle Lösungen y für Eingabe x optimieren. Das Optimierungsproblem kürzen wir durch das Tripel (opt, f, L) ab.

Beispiel 6.7 Das Clique Problem besitzt Graphen $G = (V,E)$ als Eingaben. Lösungen sind die Knotenmengen $U \subseteq V$, für die je zwei Knoten in U durch eine Kante verbunden sind. Die zu *maximierende* Zielfunktion ist die Grösse einer Knotenmenge. Das Clique Problem ist also ein Optimierungsproblem der Form $(\max, f, L)$ mit

$$L(G,U) \Leftrightarrow U \subseteq V \text{ und je zwei Knoten in } U \text{ sind durch eine Kante in } G \text{ verbunden}$$

und

$$f(G,U) = |U|.$$

Beispiel 6.8 Im Rucksackproblem sind n Objekte mit Gewichten $g_1, \ldots, g_n \in \mathbb{R}$ und Werten $w_1, \ldots, w_n \in \mathbb{N}$ vorgegeben ebenso wie eine Gewichtsschranke $G \in \mathbb{R}$. Der Rucksack ist mit einer Auswahl von Objekten zu bepacken, so dass einerseits die Gewichtsschranke G nicht überschritten wird und andererseits der Gesamtwert maximal ist.

Die Eingabe x spezifiziert die Gewichtsschranke G sowie das Gewicht und den Wert eines jeden Objekts. Die Lösungen entsprechen genau den Vektoren $y \in \{0,1\}^n$ mit $\sum_{i=1}^n g_i \cdot y_i \leq G$, es ist also

$$L(x,y) \Leftrightarrow y \in \{0,1\}^n \text{ und } \sum_{i=1}^n g_i \cdot y_i \leq G.$$

Schließlich ist

$$f(x,y) = \sum_{i=1}^n w_i \cdot y_i$$

die zu maximierende Zielfunktion.

Beispiel 6.9 Ein lineares Programmierproblem wird durch die Matrix A und die Vektoren b und c beschrieben. y ist eine Lösung für die Eingabe $x = (A,b,c)$ genau dann, wenn

$$A \cdot y \geq b \text{ und } y \geq 0.$$

Die Zielfunktion

$$f(x,y) = c^T \cdot y$$

ist zu minimieren. Im Problem der ganzzahligen Optimierung wird zusätzlich noch gefordert, dass jede Komponente einer Lösung ganzzahlig ist. In der 0-1 Programmierung werden schließlich nur binäre Vektoren als Lösungen zugelassen

Wie nicht anders zu erwarten, gibt es keine Patentrezepte zur Lösung schwieriger Entscheidungs- oder Optimierungsprobleme. Trotzdem haben sich allgemeine, einfach anwendbare Verfahren bewährt wie etwa die lokale Suche mit ihren Varianten (siehe Kapitel 8). Problemspezifisches Wissen (siehe Kapitel 7) wird über Heuristiken vermittelt. Exakte Algorithmen

wie Backtracking und Branch & Bound (siehe Kapitel 9) werden erst durch den Einsatz von Heuristiken realistisch, da dann große Teile des Suchraums ausgespart werden können.

Schließlich betrachten wir auch die Bestimmung von Gewinnstrategien in nicht-trivialen Zwei-Personen Spielen und stellen den Alpha-Beta Algorithmus vor, der zum Beispiel „hinter den modernen Schachprogrammen steckt“.

Kapitel 7

Approximationsalgorithmen*

In diesem Kapitel ist unser Ziel nicht die exakte Lösung eines Optimierungsproblems, sondern vielmehr die Berechnung einer möglichst guten Lösung, die also das Optimum möglichst scharf approximiert. Natürlich verlangen wir als „Payoff“, dass unsere Algorithmen effizient sind.

Wir messen die Qualität einer Approximation mit Hilfe des Approximationsfaktors.

Definition 7.1 *Sei $P = (\text{opt}, f, L)$ ein Optimierungsproblem.*

(a) *y^* heißt eine optimale Lösung für Eingabe x, falls*

$$f(x, y^*) = \text{opt}\ \{f(x, y) \mid y \text{ ist eine Lösung, d.h. es gilt } L(x, y)\ \}.$$

Wir setzen $\text{opt}_P(x) := f(x, y^*)$.

(b) *Eine Lösung y von P für Eingabe x heißt δ-**approximativ**, wenn*

$$f(x, y) \geq \frac{\max_P(x)}{\delta}$$

für ein Maximierungsproblem (also opt $= \max$), beziehungsweise wenn

$$f(x, y) \leq \delta \cdot \min_P(x)$$

für ein Minimierungsproblem (also opt $= \min$) gilt.

(c) *Ein Algorithmus A heißt Approximationsalgorithmus für P, falls A für jede Eingabe x eine Lösung $A(x)$ berechnet. Wir sagen, dass A δ-approximativ ist (oder dass A den **Approximationsfaktor** δ besitzt), wenn $A(x)$ für jede Eingabe x eine δ-approximative Lösung ist.*

Wir sagen also zum Beispiel, dass ein Algorithmus A den Approximationsfaktor 2 besitzt, wenn $A(x)$ bis auf den Faktor 2 das Optimum erreicht, wenn also $f(x, A(x)) \geq \frac{\text{opt}}{2}$ gilt. Für ein Minimierungsproblem hat A den Approximationsfaktor 2, wenn der erreichte Wert höchstens doppelt so groß wie das Minimum ist, wenn also stets $f(x, A(x)) \leq 2 \cdot \text{opt}$ gilt.

Beachte, dass der Approximationsfaktor stets mindestens eins ist. Der Approximationsfaktor eins besagt, dass wir das Optimierungsproblem exakt gelöst haben und unsere Approximation ist umso besser, je kleiner der Approximationsfaktor ist.

Wir stellen Approximationsalgorithmen für ein Problem der Lastverteilung, für das Rucksackproblem und für das Vertex Cover Problem vor. Wir beginnen mit Greedy-Algorithmen

für die Last-Verteilung, arbeiten dann mit dynamischer Programmierung für das Rucksackproblem und wenden schließlich die lineare Programmierung an, um das gewichtete Vertex Cover Problem zu lösen.

Auf der Webseite `http://www.nada.kth.se/~viggo/problemlist/compendium.html` wird die Approximierbarkeit einer Vielzahl wichtiger Optimierungsprobleme beschrieben.

7.1 Last-Verteilung

Wir haben m identische Maschinen, auf denen wir n Aufgaben mit Rechenzeiten $t_1, \ldots, t_n$ ausführen möchten. Wir möchten die Aufgaben so über die Maschinen verteilen, dass alle Aufgaben frühestmöglich abgearbeitet werden. Der Zeitpunkt, zu dem alle Aufgaben fertiggestellt sind, wird auch als Makespan bezeichnet. Wir versuchen eine denkbar einfache Strategie:

> Führe die Aufgaben in irgendeiner Reihenfolge aus, wobei die aktuelle Aufgabe auf der Maschine mit der bisher geringsten Last ausgeführt wird.

Diese Strategie ist damit ein On-line Algorithmus, da die aktuelle Aufgabe ohne Kenntnis der zukünftig auszuführenden Aufgaben ausgeführt wird.

Satz 7.2 *Die On-line Strategie ist 2-approximativ.*

Beweis: Für eine gegebene Instanz sei Maschine i die am schwersten „beladene“ Maschine, die also als letzte Maschine noch rechnet. A_j sei die letzte von i ausgeführte Aufgabe. Wenn T die Gesamtlaufzeit von i ist, dann waren zum Zeitpunkt $T - t_j$ alle anderen Maschinen beschäftigt, denn ansonsten hätte eine freie Maschine die Aufgabe A_j früher übernommen. Folglich ist

$$\sum_{k=1}^{n} t_k \geq (m-1) \cdot (T - t_j) + T = mT - (m-1) \cdot t_j \geq mT - m \cdot t_j.$$

Wir dividieren durch m und erhalten

$$T - t_j \leq \frac{1}{m} \cdot \sum_{k=1}^{n} t_k.$$

Sei opt der optimale Makespan. Dann gilt $T - t_j \leq \frac{1}{m} \cdot \sum_{k=1}^{n} t_k \leq$ opt ebenso wie $t_j \leq$ opt und folglich ist $T = T - t_j + t_j \leq 2 \cdot$ opt. Unsere Strategie ist also tatsächlich 2-approximativ. □

Unsere Analyse kann nicht verbessert werden wie das folgende Beispiel zeigt. $m \cdot (m-1)$ Aufgaben der Länge 1 sowie eine Aufgabe der Länge m sind gegeben. Offensichtlich lässt sich der Makespan m erreichen, wenn eine Maschine für die lange Aufgabe reserviert wird und die restlichen Maschinen die kurzen Aufgaben unter sich aufteilen. Werden andererseits zuerst die kurzen Aufgaben gleichmäßig über die m Maschinen verteilt und folgt dann die lange Aufgabe, so erhalten wir den Makespan $2 \cdot m - 1$.

Wir haben unsere Strategie durch die zuletzt präsentierte lange Aufgabe genarrt. Wir ändern deshalb unsere Strategie und fordern, dass die Aufgaben gemäß absteigender Bearbeitungszeit präsentiert werden, also zuerst die langen und dann die kürzer werdenden Aufgaben. Das worst-case Beispiel funktioniert nicht mehr und tatsächlich erhalten wir einen besseren Approximationsfaktor.

Satz 7.3 *Der Approximationsfaktor sinkt auf höchstens $\frac{3}{2}$, wenn Aufgaben gemäß fallender Bearbeitungszeit präsentiert werden.*

Beweis: Wir nehmen ohne Beschränkung der Allgemeinheit an, dass $t_1 \geq t_2 \geq \cdots t_m \geq t_{m+1} \geq \cdots \geq t_n$ gilt. Sei opt der optimale Makespan. Die folgende Beobachtung ist zentral.

Lemma 7.4 *opt* $\geq 2t_{m+1}$.

Beweis: Wir betrachten nur die ersten $m+1$ Aufgaben, die jeweils mindestens die Bearbeitungszeit t_{m+1} besitzen. Mindestens eine Maschine wird zwei Aufgaben erhalten und der optimale Makespan ist somit mindestens $2t_{m+1}$. □

Der Rest des Arguments verläuft parallel zum Beweis von Satz 7.2. Wir betrachten die Maschine i mit größter Last, die zuletzt die Aufgabe j ausführen möge. Es ist $t_j \leq t_{m+1}$, denn unsere Maschine arbeitet die Aufgaben nach fallender Bearbeitungszeit ab. Aber $2t_{m+1} \leq \text{opt}$ mit Lemma 7.4 und deshalb ist $t_j \leq t_{m+1} \leq \frac{\text{opt}}{2}$.

Sei T der Makespan unserer neuen Strategie. Mit dem Argument von Satz 7.2 erhalten wir wiederum

$$T - t_j \leq \frac{1}{m} \cdot \sum_{k=1}^{n} t_k \leq \text{opt und } t_j \leq \frac{\text{opt}}{2}.$$

Folglich ist $T = T - t_j + t_j \leq \text{opt} + \frac{\text{opt}}{2} = \frac{3\text{opt}}{2}$ und das war zu zeigen. □

Tatsächlich kann sogar gezeigt werden, dass der Approximationsfaktor höchstens $\frac{4}{3}$ ist. Beachte aber, dass unsere neue Strategie kein On-line Algorithmus mehr ist.

7.2 Das Rucksack Problem

Im Rucksackproblem (siehe Beispiel 6.8) sind n Objekte mit Gewichten $g_1, \ldots, g_n \in \mathbb{R}$ und Werten $w_1, \ldots, w_n \in \mathbb{N}$ vorgegeben ebenso wie eine Gewichtsschranke $G \in \mathbb{R}$. Der Rucksack ist mit einer Auswahl von Objekten zu bepacken, so dass einerseits die Gewichtsschranke G nicht überschritten wird und andererseits der Gesamtwert maximal ist.

Obwohl das Rucksackproblem NP-vollständig ist, können wir eine optimale Bepackung schnell berechnen, *wenn* die Summe $W = \sum_{i=1}^{n} w_i$ aller Werte nicht zu groß ist.

Satz 7.5 *Das Rucksackproblem für n Objekte und Wertesumme $W = \sum_{i=1}^{n} w_i$ kann in Zeit $O(n \cdot W)$ gelöst werden.*

Beweis: Wir beschreiben einen dynamischen Programmieralgorithmus. Dazu definieren wir für jedes i mit $1 \leq i \leq n$ und für jeden möglichen Wert w mit $0 \leq w \leq W$ das Teilproblem

$$\begin{aligned} \text{Gewicht}_i(w) \quad = \quad & \text{das minimale Gewicht einer Bepackung aus den ersten} \\ & i \text{ Objekten mit Gesamtwert genau } w. \end{aligned}$$

Wie können wir $\text{Gewicht}_i(w)$ bestimmen? Wenn wir das $i$te Objekt wählen, dann müssen wir eine Bepackung von minimalem Gewicht aus den ersten $i-1$ Objekten zusammenstellen. Der zu erzielende Wert ist $w - w_i$. In diesem Fall ist also $\text{Gewicht}_i(w) = \text{Gewicht}_{i-1}(w - w_i) + g_i$.

Ansonsten lassen wir das $i$te Objekt aus und erhalten $\text{Gewicht}_i(w) = \text{Gewicht}_{i-1}(w)$. Wir erhalten also die Rekursion

$$\text{Gewicht}_i(w) := \min\{\text{Gewicht}_{i-1}(w - w_i) + g_i,\ \text{Gewicht}_{i-1}(w)\}.$$

Wir haben höchstens $nW = n \cdot \sum_{i=1}^{n} w_i$ Teilprobleme $\text{Gewicht}_i(w)$, die jeweils in Zeit $O(1)$ gelöst werden können und die Gesamtlaufzeit ist deshalb $O(nW)$. □

Aufgabe 66

Der dynamische Programmieralgorithmus in Satz 7.5 ist nicht vollständig beschrieben.

a) Ergänze den Algorithmus um die Initialisierung der Variablen $\text{Gewicht}_i(w)$ und beschreibe, wie der Wert der optimalen Bepackung ausgegeben wird.

b) Bisher bestimmen wir nur den Wert einer optimalen Bepackung. Wie kann eine optimale Bepackung bestimmt werden?

c) Wende den Algorithmus auf das Problem $n = 4, G = 9, g_1 = 3, w_1 = 1, g_2 = 2, w_2 = 2, g_3 = 3, w_3 = 4, g_4 = 4, w_4 = 3$ an und stelle alle Zwischenergebnisse in einer Tabelle für $\text{Gewicht}_i(W^*)$ dar.

Wir zeigen jetzt, wie man aus der exakten Lösung für kleine Werte eine scharfe Approximation für beliebige Werte erhält.

Algorithmus 7.1 **Ein Approximationsalgorithmus für das Rucksackproblem**

(1) Sei $(w_1, \ldots, w_n, g_1, \ldots, g_n, G)$ eine beliebige Eingabe des Rucksackproblems. Der Approximationsfaktor $1 + \varepsilon$ sei vorgegeben. Entferne alle Objekte, deren Gewicht die Gewichtsschranke G übersteigt.

(2) Packe nur das Objekt mit größtem Wert $w_{\max}$ in den Rucksack.

(3) Die Werte werden nach unten skaliert, nämlich setze

$$w_i^* = \lfloor \frac{w_i}{s} \rfloor \text{ für den Skalierungsfaktor } s = \frac{\varepsilon \cdot w_{\max}}{n}.$$

(4) Berechne eine exakte Lösung x für die Eingabe $(w_1^*, \ldots, w_n^*, g_1, \ldots, g_n, G)$. Wenn in x die Objektmenge $I \subseteq \{1, \ldots, n\}$ in den Rucksack gepackt wird, dann setze $W_2 = \sum_{i \in I} w_i$ und übernimm die Bepackung für die alten Werte.

(5) Gib die beste der beiden Bepackungen aus.

Satz 7.6 *Algorithmus 7.1 ist* $(1 + \varepsilon)$*-approximativ mit Laufzeit* $O(\frac{1}{\varepsilon} \cdot n^3)$.

Beweis: Wir können annehmen, dass alle Objekte eingepackt werden können, da der Algorithmus zu schwere Objekte zuerst entfernt. Die neuen Werte sind durch $\frac{n}{\varepsilon}$ beschränkte natürliche Zahlen, denn $w_i^* = \lfloor w_i \cdot \frac{n}{\varepsilon \cdot \max_j w_j} \rfloor \leq \frac{n}{\varepsilon}$. Also ist die neue Wertesumme durch $\frac{n^2}{\varepsilon}$ beschränkt, und wir können somit das Optimierungsproblem in den neuen Werten (und alten Gewichten) in Zeit $O(n \cdot \frac{n^2}{\varepsilon}) = O(\frac{1}{\varepsilon} \cdot n^3)$ exakt lösen.

Wir verwenden die gefundene Bepackung B auch für die alten Werte und vergleichen B mit B_{opt}, der optimalen Bepackung für die alten Werte.

$$\begin{aligned}
\sum_{i \in B_{\text{opt}}} w_i &\leq sn + \sum_{i \in B_{\text{opt}}} s \cdot \lfloor \frac{w_i}{s} \rfloor \\
&\leq sn + \sum_{i \in B} s \cdot \lfloor \frac{w_i}{s} \rfloor \quad \text{denn } B \text{ ist die beste Bepackung für die Werte } w_i^* = \lfloor \tfrac{w_i}{s} \rfloor \\
&\leq sn + \sum_{i \in B} w_i \ .
\end{aligned}$$

Fall 1: Wir betrachten zuerst den (wahrscheinlichen) Fall, dass der Wert der Bepackung B mindestens so groß wie der maximale Wert eines Objekts ist, das heißt wir nehmen an, dass $\sum_{i\in B} w_i \geq \max_j w_j$ gilt. In diesem Fall gibt Algorithmus 7.1 den Wert W_2 der Bepackung B aus. Es ist

$$\frac{\sum_{i\in B_{\text{opt}}} w_i}{\sum_{i\in B} w_i} \leq \frac{\sum_{i\in B} w_i + sn}{\sum_{i\in B} w_i} \leq 1 + \frac{sn}{\sum_{i\in B} w_i} \leq 1 + \frac{sn}{\max_j w_j} \tag{7.1}$$

$$\leq 1 + \frac{n \cdot \frac{\varepsilon \cdot \max_j w_j}{n}}{\max_j w_j} = 1 + \varepsilon. \tag{7.2}$$

Fall 2: $\sum_{i\in B} w_i < \max_j w_j$. Jetzt findet Algorithmus 7.1 eine Bepackung mit Wert $\max_j w_j$. Es ist

$$\frac{\sum_{i\in B_{\text{opt}}} w_i}{\max_j w_j} \leq \frac{\sum_{i\in B} w_i + sn}{\max_j w_j} \leq \frac{\max_j w_j + sn}{\max_j w_j},$$

denn wir haben die Fallannahme benutzt. Die Behauptung folgt jetzt analog zu (7.1). □

Algorithmus 7.1 ist besonders komfortabel, da er nicht nur die Eingabe sondern auch eine Vorgabe für die Qualität der Approximation akzeptiert. Man spricht in diesem Fall auch von einem vollen Approximationsschema, da die Laufzeit polynomiell in der Länge n der Eingabe und $\frac{1}{\varepsilon}$ ist.

Aufgabe 67
Wir stellen eine Heuristik für das Rucksackproblem vor: Zuerst sortiert die Heuristik die Objekte gemäß absteigendem „Wert pro Kilo“, d.h. gemäß den Brüchen w_i/g_i. Schließlich werden die Objekte der Reihe nach, gemäß ihrer sortierten Reihenfolge, in den Rucksack verpackt, bis die Gewichtsschranke erreicht ist.

(a) Zeige, dass der Approximationsfaktor unbeschränkt ist.

(b) Sei opt der Wert einer besten Bepackung, wpk der Wert der von unserer Heuristik erreicht wird und sei $w_{\max}$ der maximale Wert eines Objekts. Zeige

$$\text{opt} \leq 2 \cdot \max\{\text{wpk}, w_{\max}\}.$$

Aufgabe 68
Beim fraktionalen Rucksackproblem sind n Objekte $D_1, \ldots, D_n$ mit Gewichten g_i und Werten w_i gegeben. Es soll eine Beladung des Rucksacks gefunden werden, so dass eine gegebene Gewichtsschranke G nicht überschritten wird, aber der Gesamtwert der eingepackten Objekte maximal ist. Eine Beladung darf dabei von jedem Objekt D_i einen Bruchteil $x_i \in [0, 1]$ wählen, und dieser Bruchteil hat Gewicht $x_i \cdot g_i$ und Wert $x_i \cdot w_i$. Löse das fraktionale Rucksackproblem durch einen effizienten Algorithmus und weise die Optimalität der gefundenen Lösung nach.

7.3 Approximation und lineare Programmierung

Wir beginnen mit einem kombinatorischen Algorithmus für das Vertex-Cover Problem. Die Idee ist simpel: Wir konstruieren eine nicht-vergrößerbare Menge M knoten-disjunkter Kanten, also ein nicht-vergrößerbares Matching, und wählen die Endpunkte der Kanten aus M als unseren Vertex Cover U. Es ist klar, dass ein optimales Vertex Cover mindestens $|M|$ Knoten besitzen muss, da jede Kante aus M überdeckt werden muss. Damit ist U höchstens doppelt so groß wie ein optimaler Vertex Cover, aber ist U überhaupt ein Vertex Cover?

Algorithmus 7.2

(1) Die Eingabe besteht aus einem ungerichteten Graphen $G = (V, E)$. Setze $M := \emptyset$.

(2) while $(E \neq \emptyset)$

Wähle eine Kante $e \in E$ und füge e zu M hinzu.

Entferne alle Kanten aus E, die einen Endpunkt mit e gemeinsam haben.

(3) Gib die Knotenmenge $U = \{v \in V \mid v \text{ ist Endpunkt einer Kante in } M\}$ aus.

Satz 7.7 *Algorithmus 7.2 ist ein 2-Approximationsalgorithmus für VC.*

Beweis: Sei $\text{vc}(G)$ die minimale Grösse eines Vertex Covers. Offensichtlich gilt

$$\text{vc}(G) \geq |M|,$$

denn jeder Vertex Cover muss mindestens einen Endpunkt für jede der $|M|$ knoten-disjunkten Kanten besitzen. Andererseits ist das Matching M nicht erweiterbar und damit hat jede Kante $e \in E$ einen Endpunkt mit einer Kante aus M gemeinsam. Also findet unser Algorithmus einen Vertex Cover der Größe

$$|U| = 2 \cdot |M| \leq 2 \cdot \text{vc}(G)$$

und wir haben damit eine 2-approximative Lösung erhalten. □

Bemerkung 7.1 Natürlich stellt sich die Frage, ob Algorithmus 9.1 eine bessere Approximationskonstante besitzt. Leider ist die Antwort negativ: Der vollständige bipartite Graph mit n Knoten „auf jeder Seite“ besitzt natürlich eine überdeckende Knotenmenge der Grösse n, aber Algorithmus 9.1 wird alle $2 \cdot n$ Knoten als Überdeckung ermitteln.

Im *gewichteten Vertex-Cover Problem* wird jedem Knoten $v \in V$ zusätzlich ein Gewicht w_v zugeordnet. Gesucht ist ein Vertex-Cover U, so dass das Gesamtgewicht der Knoten in U minimal ist. Wir schreiben das gewichtete Vertex Cover Problem als das lineare Programm

$$\text{minimiere } \sum_{v \in V} w_v \cdot x_v, \qquad \text{so dass } x_u + x_v \geq 1 \text{ für alle } \{u, v\} \in E \qquad (7.3)$$
$$\text{und } x_u \geq 0 \text{ für alle } u \in V.$$

Wir nehmen im Folgenden an, dass $V =$ ist. Wenn $U \subseteq V$ ein Vertex Cover ist, dann ist der Inzidenzvektor[1] x_u von U eine Lösung von (7.3), denn alle Ungleichungen werden erfüllt. Aber leider hat das lineare Programm zusätzliche fraktionale Lösungen, deren Wert sehr viel kleiner sein kann als der Wert des besten Vertex Covers.

Beispiel 7.1 Wir setzen $w_v = 1$ für jeden Knoten v und betrachten den vollständigen Graphen V_n mit n Knoten: Je zwei Knoten sind also durch eine Kante verbunden. Offensichtlich benötigt jeder Vertex Cover mindestens $n - 1$ Knoten, aber das lineare Programm lässt den Vektor $(\frac{1}{2}, \ldots, \frac{1}{2})$ als Lösung zu und behauptet somit den nicht erreichbaren Wert $\frac{n}{2}$.

[1] Der Inzidenzvektor x einer Teilmenge $U \subseteq \{1, \ldots, n\}$ hat genau dann an Position i eine Eins, wenn $i \in U$, und anderenfalls ist $x_i = 0$.

Glücklicherweise ist der vollständige Graph auch bereits ein worst-case Beispiel wie wir gleich sehen werden. Wir benutzen die **Methode des Rundens** fraktionaler Lösungen: Sei x^* eine optimale Lösung. Da x^* im allgemeinen ein reellwertiger Vektor ist, runden wir x^*, um einen Vertex-Cover zu erhalten. Wir setzen also

$$U = \{v \in V \mid x_v^* \geq \frac{1}{2}\}.$$

U ist ein Vertex-Cover, denn für jede Ungleichung $x_u^* + x_v^* \geq 1$ ist $x_u^* \geq \frac{1}{2}$ oder $x_v^* \geq \frac{1}{2}$ und damit folgt $u \in U$ oder $v \in U$. Andererseits ist

$$\sum_{v \in U} w_v \leq \sum_{v \in V} w_v \cdot (2x_v^*) = 2 \cdot \sum_{v \in V} w_v \cdot x_v^* = 2 \cdot \text{opt}$$

und wir haben eine 2-approximative Lösung erhalten.

Satz 7.8 *Das gewichtete Vertex-Cover Problem kann mit Approximationsfaktor* 2 *effizient gelöst werden.*

Die Methode des Rundens erlaubt in unserem Beispiel somit die Anwendung der mächtigen Methode der linearen Programmierung, obwohl optimale Lösung möglicherweise fraktional sind. Wir erhalten zwar im Allgemeinen keine optimalen Lösungen mehr, aber die erhaltenen Lösungen sind „recht gut“.

Kapitel 8

Lokale Suche*

Wir untersuchen jetzt das vielleicht einfachste, aber in vielen Anwendungen auch erfolgreiche Prinzip der lokalen Suche. Dazu nehmen wir an, dass das Minimierungsproblem $P = (\min, f, L)$ zu lösen ist, wobei zusätzlich zu jeder Lösung y auch eine Umgebung $\mathcal{N}(y)$ benachbarter Lösungen gegeben ist.

Algorithmus 8.1 **Strikte lokale Suche**

(1) Sei $y^{(0)}$ eine Lösung für die Eingabe x. Setze $i = 0$.

(2) Wiederhole solange, bis eine lokal optimale Lösung gefunden ist:

(2a) Bestimme einen Nachbarn $y \in \mathcal{N}(y^{(i)})$, so dass $f(x, y) < f(x, y^{(i)})$ und y eine Lösung ist.

(2b) Setze $y^{(i+1)} = y$ und $i = i + 1$.

Man beachte, dass die strikte lokale Suche kein Greedy-Algorithmus (bei gegebenen Nachbarschaften) ist, da die Suche nicht notwendigerweise mit dem besten Nachbarn fortgesetzt wird: Die Fortsetzung der Suche mit irgendeinem besseren Nachbarn ist erlaubt.

Im Entwurf eines lokalen Suchverfahrens müssen die folgenden Fragen beantwortet werden.

(1) Wie sind die Nachbarschaften $\mathcal{N}(y)$ zu definieren?

Häufig werden k-Flip Nachbarschaften gewählt: Wenn nur Elemente aus $\{0,1\}^n$ als Lösungen in Frage kommen und wenn $y \in \{0,1\}^n$ die gegenwärtige Lösung ist, dann ist

$$\mathcal{N}_k(y) = \{y' \in \{0,1\}^n \mid y \text{ und } y' \text{ unterscheiden sich in höchstens } k \text{ Positionen}\}$$

die k-Flip Nachbarschaft von y. Man beachte aber, dass die k-Flip Nachbarschaft $\binom{n}{k}$ Elemente besitzt, und die Bestimmung eines Nachbarn mit kleinstem Funktionswert ist schon für kleine Werte von k eine Herausforderung.

Wir beschreiben im Anschluss die lokale Suche in variabler Tiefe, ein Suchverfahren mit beachtlich guten Ergebnissen zum Beispiel für das MINIMUM TRAVELLING SALESMAN Problem und das MINIMUM BALANCED CUT Problem.

(2) Mit welcher Anfangslösung $y^{(0)}$ soll begonnen werden?

Häufig werden lokale Suchverfahren benutzt, um eine durch eine Heuristik gefundene Lösung zu verbessern. In einem solchen Fall muss der Nachbarschaftsbegriff sorgfältig gewählt werden, damit die ursprüngliche Lösung nicht schon ein lokales Optimum ist. Eine zufällig ausgewürfelte Lösung ist eine zweite Option.

(3) Mit welcher Nachbarlösung soll die Suche fortgesetzt werden?

Wenn die Nachbarschaft genügend klein ist, dann liegt die Wahl eines Nachbarn mit kleinstem Zielfunktionswert nahe. Bei zu großen Nachbarschaften wählt man häufig benachbarte Lösungen mit Hilfe einer Heuristik, bzw. man wählt einen zufälligen Nachbarn.

Aufgabe 69
Im Max-Cut Problem ist ein ungerichteter Graph $G = (V, E)$ gegeben. Es ist eine Knotenmenge $W \subseteq V$ zu bestimmen, so dass die Anzahl kreuzender Kanten (also die Anzahl der Kanten mit genau einem Endpunkt in W) größtmöglich ist.

Zeige, dass lokale Suche für *jede* Anfangslösung den Approximationsfaktor 2 besitzt. Wir nehmen dazu an, dass die Umgebung einer Knotenmenge W aus allen Knotenmengen U mit $|W \oplus U| = 1$ besteht. $W \oplus U = (W \cup V) \setminus (W \cap V)$ bezeichnet dabei die symmetrische Differenz der Mengen W und U.

Aufgabe 70
Im Max-2-SAT Problem sind ein Menge von Klauseln mit jeweils genau 2 Literalen gegeben. Literale sind positive oder negierte Ausprägungen der Variablen. Wir setzen voraus, dass jede Variable in mindestens einer Klausel auftritt, und dass die trivialen Klauseln $(x_i \vee \overline{x_i})$, $(x_i \vee x_i)$ sowie $(\overline{x_i} \vee \overline{x_i})$ nicht vorkommen. Gesucht ist eine Belegung der Variablen, so dass die Anzahl der erfüllten Klauseln maximiert wird.

Für die Variablen $x_1, \ldots, x_k$ fassen wir ihre Belegung als einen Vektor $\vec{x} \in \{0,1\}^k$ auf. Wir definieren $U(\vec{x}) = \{\vec{y} | \sum_{i=1}^{k} |x_i - y_i| = 1\}$ als Umgebung von $\vec{x}$.

Die lokale Suche erlaubt also genau dann die Belegung *einer* Variablen zu negieren, wenn sich dadurch die Anzahl der erfüllten Klauseln erhöht.

(a) **Zeige**, dass lokale Suche für Max-2-SAT für *jede* Anfangslösung den Approximationsfaktor 2 erreicht.

(b) **Konstruiere** für unendlich viele natürliche Zahlen k Max-2-Sat Instanzen über den Variablen $x_1, \ldots, x_k$, so dass die Instanzen ein möglichst kleines lokales Maximum im Vergleich zum globalen Maximum besitzen. Wie groß kann der Bruch $\frac{\text{Optimum}}{\text{Wert des lokalen Maximums}}$ gemacht werden?

Aufgabe 71
In dieser Aufgabe untersuchen wir den Einfluss des Abstandsbegriffs auf die Güte der Approximation bei der lokalen Suche. Wir betrachten das Problem des gewichteten bipartiten Matchings. Gegeben ist ein bipartiter Graph $G = (V_1 \cup V_2, E)$ mit $V_1 \cap V_2 = \emptyset$ und $E \subseteq V_1 \times V_2$ und eine Gewichtsfunktion $w : E \to \mathbb{R}$. Gesucht ist ein Matching maximalen Gewichts. Das Gewicht eines Matchings M ist die Summe der Gewichte der enthaltenen Kanten. Zu gegebenem Matching M definieren wir die k-Flip Umgebung als

$$\mathcal{N}_k(M) := \left\{ M' \,|\, \left|(M' \setminus M) \cup (M \setminus M')\right| \leq k \right\}.$$

Die lokale Suche gemäß der k-Flip Umgebung erlaubt den Übergang von einem Matching M zu einem Matching aus $\mathcal{N}_k(M)$.

(a) **Zeige,** dass die lokale Suche gemäß der 2-Flip Umgebung für das gewichtete bipartite Matching keinen konstant beschränkten Approximationsfaktor hat.

(b) **Zeige,** dass die lokale Suche gemäß der 3-Flip Umgebung einen Approximationsfaktor von höchstens 2 hat.

(c) **Zeige,** dass die lokale Suche gemäß der 3-Flip Umgebung einen Approximationsfaktor von mindestens 2 hat.

Beispiel 8.1 **Der Simplex-Algorithmus für das lineare Programmieren**

Der Lösungsraum des linearen Programmierproblems

$$\min c^T \cdot y, \text{ so dass } A \cdot y \geq b \text{ und } y \geq 0$$

ist ein Durchschnitt von Halbräumen der Form $\{y \mid \alpha^T \cdot y \geq \beta\}$. Man kann sich deshalb leicht davon überzeugen, dass das Optimum an einer „Ecke“ des Lösungsraums angenommen wird.

Der Simplex-Algorithmus für die lineare Programmierung wandert solange von einer Ecke zu einer besseren, aber im Lösungsraum benachbarten Ecke, bis keine Verbesserung erreichbar ist. Da der Lösungsraum konvex[1] ist, kann man zeigen, dass Simplex stets eine optimale Ecke findet. (Siehe dazu auch Kapitel 4.4.) Wenn wir also nur Ecken als Lösungen zulassen, dann ist Algorithmus 8.1 die Grobstruktur des sehr erfolgreichen Simplex-Algorithmus.

Beispiel 8.2 **Minimierung durch Gradientenabstieg**

Wir nehmen an, dass das Minimierungsproblem $P = (\min, f, L)$ eine differenzierbare Zielfunktion $f : \mathbb{R}^n \to \mathbb{R}$ besitzt und dass der Lösungsraum eine kompakte Teilmenge des $\mathbb{R}^n$ ist. Wenn wir uns gegenwärtig in der Lösung a befinden, in welcher Richtung sollten wir nach kleineren Funktionswerten suchen? In der Richtung des negativen Gradienten!

Lemma 8.1 (Gradientenabstieg) *Sei $f : \mathbb{R}^n \to \mathbb{R}$ eine zweimal im Punkt $a \in \mathbb{R}^n$ stetig differenzierbare Funktion mit $\nabla f(a) \neq 0$. Dann gibt es ein $\eta > 0$ mit*

$$f\left(a - \eta \cdot \nabla f(a)\right) < f(a),$$

wobei $\nabla f(a)$ der Gradient von f an der Stelle a ist.

Beweis: Wir können die Funktion f durch ihre linearen Taylor-Polynome approximieren und erhalten für hinreichend kleines z, dass

$$f(a + z) = f(a) + \nabla f(a)^T \cdot z + O(z^T \cdot z).$$

Für $z = -\eta \cdot \nabla f(a)$ bei hinreichend kleinem $\eta > 0$ erhalten wir

$$f(a - \eta \cdot \nabla f(a)) = f(a) - \eta \cdot \ ||\nabla f(a)||^2 + \eta^2 \cdot O(||\nabla f(a)||^2).$$

Für hinreichend kleines $\eta < 1$ wird somit $f(a - \eta \cdot \nabla f(a))$ kleiner als $f(a)$ sein. □

Wir verringern also den Funktionswert durch eine Bewegung in Richtung des negativen Gradienten und die Strategie

$$x^{(i+1)} = x^{(i+1)} - \eta \cdot \nabla f(x^{(i)})$$

für ein genügend kleines η liegt nahe. Wenn wir die Nachbarschaft einer Lösung x als einen kleinen Ball um x definieren, dann beschreibt Algorithmus 8.1 die Methode des iterierten Gradientenabstiegs. Diese Methode hat zum Beispiel in dem neuronalen Lernverfahren „Backpropagation“ Anwendungen in der Informatik.

Die große Gefahr in Anwendungen der lokalen Suche sind lokale Optima, also Lösungen x für die keine benachbarte Lösung besser als x ist.

Beispiel 8.3 Wir führen eine lokale Suche für das Vertex Cover Problem durch. Für einen gegebenen Graphen $G = (V, E)$ und einen Vertex Cover y definieren wir die Nachbarschaft $\mathcal{N}(y)$ als die Klasse aller Teilmengen $z \subseteq V$, die durch das Hinzufügen oder das Entfernen eines Elementes aus y entstehen. Wir beginnen die lokale Suche mit der Lösung $U = V$. Da der Suchalgorithmus 8.1 fordert, dass nur Nachbarn mit kleinerem Zielfunktionswert gewählt werden dürfen, werden nacheinander Elemente entfernt bis ein lokales Optimum erreicht ist.

[1] Eine Teilmenge $X \subseteq \mathbb{R}^n$ heißt konvex, wenn X mit je zwei Punkten auch die verbindende Gerade enthält.

Für den leeren Graphen $G = (V, \emptyset)$ funktioniert die lokale Suche komplikationslos und $U = \emptyset$ wird nach $|V|$ Suchschritten als lokales Optimum ausgegeben.

Betrachten wir als nächstes den Sterngraphen mit dem Sternzentrum 0 und den Satelliten $1, \ldots, n$. Die einzigen Kanten des Sterngraphen verbinden das Zentrum 0 mit allen Satelliten. Wird im ersten Schritt das Zentrum entfernt, dann haben wir ein sehr schlechtes lokales Minimum erreicht. Wird hingegen im ersten Schritt ein Satellit entfernt, dann wird zwangsläufig das nur aus dem Zentrum bestehende globale Minimum gefunden. (Beachte, dass nur Nachbarn betrachtet werden, die auch Vertex Covers sind.)

Komplizierter ist die Situation, wenn wir den aus den n Knoten $0 \ldots, n-1$ bestehenden Weg $0 \to 1 \to 2 \to \cdots \to n-2 \to n-1$ als Graphen wählen. Für gerades n ist $U_{\text{opt}} = \{0, 2, 4, \ldots, n-2\}$ ein optimaler Vertex Cover.

Aufgabe 72
Bestimme alle lokalen Optima für den Weg mit n Knoten. Bestimme den Wert des schlechtesten lokalen Optimums.

8.1 Lokale Suche in variabler Tiefe

Die große Schwäche der lokalen Suche ist, dass nur Abwärtsbewegungen erlaubt sind. Als Konsequenz wird eine Lösung kurz über lang in ein lokales Minimum mit großer Sogwirkung fallen. Wir betrachten deshalb die lokale Suche mit variabler Tiefe, die auch Aufwärtsbewegungen erlaubt. Wir beschreiben diese Variante der lokalen Suche für den Fall, dass der Lösungsraum eine Teilmenge von $\{0,1\}^n$ ist und dass die k-Flip Nachbarschaft für eine (kleine) Konstante k gewählt wird.

Algorithmus 8.2 **Lokale Suche in variabler Tiefe**

y sei die gegenwärtige Lösung eines Optimierungsproblems mit der k-Flip Nachbarschaft. Der Lösungsraum sei eine Teilmenge von $\{0,1\}^n$. Eine bessere, aber nicht notwendigerweise benachbarte Lösung y' ist zu bestimmen.

(1) Initialisierungen: Setze EINGEFROREN $= \emptyset$, LÖSUNGEN $= \{y\}$ und $z = y$.

/* EINGEFROREN wird stets eine Menge von Positionen sein. LÖSUNGEN ist eine Menge von Lösungen, die während der Berechnung inspiziert werden. */

(2) Wiederhole, solange EINGEFROREN $\neq \{1, \ldots, n\}$

(2a) Bestimme eine beste Lösung $z' \neq z$ in der k-Flip Nachbarschaft von z.

(2b) Friere alle im Wechsel von z nach z' geänderten Positionen ein, füge z' zu LÖSUNGEN hinzu und setze $z = z'$.

/* Beachte, dass z' durchaus eine schlechtere Lösung als z sein darf. Wir erlauben also Aufwärtsbewegungen. Durch das Einfrieren geänderter Positionen wird die Schleife höchstens n Mal durchlaufen. */

(3) Gib die beste Lösung in LÖSUNGEN als y' aus.

Beispiel 8.4 Im NP-vollständigen MINIMUM BALANCED CUT Problem ist ein ungerichteter Graph $G = (V, E)$ gegeben. Eine Zerlegung $V = V_1 \cup V_2$ ist zu bestimmen, so dass $|V_1| = \lfloor \frac{|V|}{2} \rfloor, |V_2| = \lceil \frac{|V|}{2} \rceil$ und so dass die Anzahl kreuzender Kanten, also die Anzahl aller Kanten mit einem Endpunkt in V_1 und einem Endpunkt in V_2, minimal ist.

Wir wählen die 2-Flip Nachbarschaft und repräsentieren eine Zerlegung $V = V_1 \cup V_2$ durch den Inzidenzvektor y von V_1. Beachte, dass eine eine Lösung W_1 nur dann mit V_1 benachbart ist, wenn W_1 durch das Entfernen und das nachfolgende Einfügen eines Elementes aus V_1 entsteht.

8.2 Der Metropolis Algorithmus und Simulated Annealing

Wir betrachten wieder das allgemeine Minimierungsproblem $P = (\min, f, L)$ und nehmen an, dass für jede Lösung eine Nachbarschaft $\mathcal{N}(y)$ von Lösungen definiert ist. Wir möchten diesmal Aufwärtsbewegeungen zulassen, aber werden dies nur widerstrebend tun. Die Bereitschaft schlechtere Nachbarn zuzulassen wird durch den Temperatur-Parameter T gesteuert: Je höher die Temperatur, umso höher die Wahrscheinlichkeit, dass ein schlechterer Nachbar akzeptiert wird.

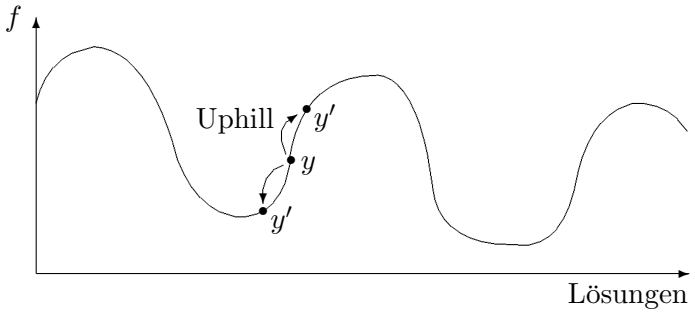
Algorithmus 8.3 **Der Metropolis Algorithmus**

(1) Sei y eine Anfangslösung und sei T die Temperatur.

(2) Wiederhole hinreichend oft:

(2a) Wähle zufällig einen Nachbarn $y' \in \mathcal{N}(y)$.

(2b) Wenn $f(y') \leq f(y)$, dann akzeptiere y' und setze $y = y'$. Ansonsten setze $y = y'$ mit Wahrscheinlichkeit $e^{-\frac{f(y')-f(y)}{T}}$.
/* Je schlechter der Wert des neuen Nachbarn y', umso geringer die Wahrscheinlichkeit, dass y' akzeptiert wird. Schlechte Nachbarn haben nur eine Chance bei entsprechend hoher Temperatur. */



Mit welcher Wahrscheinlichkeit wird die Lösung y besucht, wenn der Metroplis-Algorithmus genügend häufig wiederholt wird? Das folgende Ergebnis kann gezeigt werden:

Satz 8.2 *$f_y(t)$ bezeichne die relative Häufigkeit, mit der der Metropolis-Algorithmus die Lösung y in den ersten t Schritten besucht. Dann gilt*

$$\lim_{t\to\infty} f_y(t) = \frac{e^{-f(y)/T}}{Z}, \text{ wobei } \quad Z = \sum_{y, L(y)} e^{-f(y)/T}.$$

Auf den ersten Blick ist dies eine hervorragende Eigenschaft des Metroplis-Algorithmus, da die Wahrscheinlichkeit eines Besuchs für gute Lösungen y am größten ist. Allerdings spielt der Grenzwert in vielen Anwendungen nicht mit: Die Konvergenzgeschwindigkeit ist leider sehr langsam.

Beispiel 8.5 Wir greifen Beispiel 8.3 wieder auf, wenden diesmal aber nicht die lokale Suche, sondern den Metropolis Algorithmus auf das Vertex Cover Problem an. Wir betrachten wieder den leeren Graphen $G = (V, \emptyset)$ und beginnen den Metropolis Algorithmus mit V als Knotenmenge. Anfänglich wird die Knotenmenge nur reduziert und Metropolis verhält sich wie die lokale Suche. Das Problem beginnt, wenn die gegenwärtige Lösung y nur noch wenige Knoten besitzt: y hat sehr viel mehr schlechtere als bessere Nachbarn und dementsprechend werden schlechtere Nachbarn mit sehr viel höherer Wahrscheinlichkeit gewählt. Die Akzeptanzwahrscheinlichkeit einer schlechteren Lösung ist nur unwesentlich kleiner und schlechtere Lösungen, wenn nicht im ersten oder zweiten Anlauf, werden kurz über lang gewählt: Der Metropolis-Algorithmus bekommt Angst vor der eigenen Courage, wenn gute, aber längst nicht optimale Lösungen erreicht werden.

Betrachten wir den Sterngraphen als zweites Beispiel. Hier zeigt der Metropolis-Algorithmus seine Stärke. Selbst wenn das Zentrum irgendwann entfernt wird, so ist die Wahrscheinlichkeit hoch, dass das Zentrum nach nicht zu langer Zeit wieder betrachtet wird. Mit beträchtlicher Wahrscheinlichkeit wird die neue Lösung akzeptiert und die schlechte Entscheidung, das Zentrum zu entfernen, wird revidiert. Danach, wenn mindestens ein Satellit nicht in der jeweiligen Lösung liegt, bleibt das Zentrum erhalten, da sonst kein Vertex Cover vorliegt. Allerdings hat Metropolis auch für den Sterngraphen Angst vor der eigenen Courage, wenn die Überdeckung genügend klein ist.

Das obige Beispiel legt nahe, dass wir versuchen sollten, die Temperatur vorsichtig zu senken, um Aufwärtsbewegungen nach entsprechend langer Suchzeit signifikant zu erschweren. Genau dieses Vorgehen wird in dem Simulated Annealing Verfahren durchgeführt. Die folgende Analogie aus der Physik erklärt das Verfahren: Erhitzt man einen festen Stoff so stark, dass er flüssig wird und läßt man ihn dann wieder langsam abkühlen, so ist der Stoff bestrebt, möglichst wenig der zugeführten Energie zu behalten. Der Stoff bildet eine Kristallgitterstruktur (Eiskristalle sind ein Beispiel.) Je behutsamer nun das Ausglühen (*engl.: Annealing*) durchgeführt wird, umso reiner ist die Gitterstruktur; Unregelmäßigkeiten in der Gitterstruktur, die durch zu rasches Abkühlen entstehen, stellen lokale Energieminima dar.

Algorithmus 8.4 **Simulated-Annealing**

(1) Sei y die Anfangslösung und sei T die Anfangstemperatur.

(2) Wiederhole hinreichend oft:

(2a) Wähle zufällig einen Nachbarn $y' \in \mathcal{N}(y)$.

(2b) Wenn $f(y') \leq f(y)$, dann akzeptiere y' und setze $y = y'$. Ansonsten setze $y = y'$ mit Wahrscheinlichkeit $e^{-\frac{f(y)-f(x)}{T}}$.

(3) Wenn die Temperatur noch nicht genügend tief ist, dann wähle eine neue, niedrigere Temperatur T und führe Schritt (2) aus. Ansonsten gib die erhaltene Lösung y aus.

Die Wahl des Abkühlprozesses ist problemabhängig und erfordert experimentelle Arbeit. Aber selbst dann kann keine Garantie gegeben werden, dass eine gute Lösung auch gefunden wird: Wird zum falschen Zeitpunkt abgekühlt, bleibt man in einem lokalen Optimum gefangen und verliert die Möglichkeit zu entkommen.

Beispiel 8.6 **MINIMUM BALANCED CUT**

Um Simulated-Annealing auf MINIMUM BALANCED CUT anzuwenden, lässt man beliebige Zerlegungen $(W, V \setminus W)$ zu und wählt

$$f(W) := |\{e \in E \mid |\{e\} \cap W| = 1\}| + \underbrace{\alpha \cdot (|W| - |V \setminus W|)^2}_{\text{Strafterm}}$$

als zu minimierende Zielfunktion. Wir versuchen durch den Strafterm, eine perfekte Aufteilung, also $|W| = \frac{1}{2} \cdot |V|$, zu erzwingen.

Die Lösungsmenge ist die Potenzmenge von V. Als Startlösung für Simulated-Annealing wählen wir ein perfekte, zufällig gewählte Aufteilung. Für eine Knotenteilmenge $W \subseteq V$ definieren wir die Umgebung von W als

$$\mathcal{N}(W) := \{W' \subseteq V \mid |W \oplus W'| \leq 1\},$$

also als die 1-Flip Nachbarschaft von W. In [D.S. Johnson. C.R. Aragon. L.A.McGeoch und C.Schevon (1989): **Simulated Annealing: An experimental Evaluation, Part I: Graph Partitioning**, Operation Research, Band 37, Nr.6, Seiten 865-892.] wird die Anfangstemperatur so gewählt, dass 40% aller Nachbarn akzeptiert werden. Die Temperatur wird über den Zeitraum von $16 \cdot |V|$ konstant gehalten und dann bei jeder Wiederholung von Schritt (3) um den Faktor $0,95$ gesenkt („geometrische Abkühlung"). Bei *zufällig gewählten* Graphen schneidet Simulated-Annealing (erstaunlicherweise?) erfolgreicher ab als maßgeschneiderte Algorithmen wie Algorithmus 8.2. Dieses Phänomen tritt auch auf, wenn wir den maßgeschneiderten Algorithmen die gleiche (große) Laufzeit wie der Simulated-Annealing-Methode erlauben.

Weitere Experimente wurden für *strukturierte* Graphen durchgeführt. 500 (bzw. 1000) Punkte werden zufällig aus dem Quadrat $[0, 1]^2$ gewählt und zwei Punkte werden verbunden, wenn sie nahe beieinander liegen. Für diese Graphenklasse „bricht" die Simulated-Annealing-Methode ein und der Algorithmus 8.2 ist deutlich überlegen. Eine mögliche Erklärung für das unterschiedliche Abschneiden wird in der unterschiedlichen Struktur der lokalen Minima liegen. Die geometrisch generierten Graphen werden Minima mit weitaus größeren Anziehungsbereichen als die zufällig generierten Graphen haben. Diese „Sogwirkung" lokaler Minima ist bei den geometrisch generierten Graphen schwieriger als für Zufallsgraphen zu überwinden.

Bemerkung 8.1 Simulated Annealing wie auch der Metropolis-Algorithmus sind lokale Suchverfahren, die Uphillbewegungen erlauben. Ihr großer Vorteil, nämlich die Anwendbarkeit auf eine große Klasse von Problemen, ist allerdings auch ihre große Schwäche: Eigenschaften des vorliegenden Optimierungsproblems können nur durch die Wahl des Umgebungsbegriffs und durch die Temperaturregelung ausgenutzt werden.

Wenn genügend Laufzeit investiert werden kann, dann ist eine Anwendung dieser randomisierten Suchverfahren als ein *erster* Schritt in der Untersuchung eines Optimierungsproblems sicherlich zu empfehlen.

8.3 Evolutionäre Algorithmen

Für Simulated Annealing haben wir das sorgfältige Abkühlen eines verflüssigten Stoffes in Verbindung mit der Lösung eines Minimierungsproblems gebracht: Das Erreichen einer reinen

Kristallgitterstruktur entspricht einem globalen Minimum, während Unreinheiten lokalen Minima entsprechen. Diesmal untersuchen wir Maximierungsprobleme[2] $(\max, f, L)$ und wählen die Evolution als Vorbild. Wie können wir das Erfolgsprinzip der Evolution, „Survival of the Fittest“, in einen Approximationsalgorithmus umsetzen?

Es liegt nahe, die Fitness eines Individuums (oder einer Lösung) y mit dem Funktionswert $f(y)$ gleichzusetzen. Weiterhin sollten wir versuchen, eine gegebene Population von Lösungen zu verbessern, indem wir Lösungen „mutieren“ oder zwei oder mehrere Lösungen miteinander „kreuzen“. Um zu erklären, was wir unter Mutieren und Kreuzen verstehen, werden wir einen Black-Box Ansatz verfolgen und mit *problemunabhängigen* Mutations- und Kreuzungsoperatoren arbeiten. Der Vorteil dieses Ansatzes ist, dass evolutionäre Algorithmen einfach anzuwenden sind, der definitive Nachteil ist, dass evolutionäre Algorithmen nicht die Effektivität von auf das Problem maßgeschneiderten Strategien besitzen werden.

Algorithmus 8.5 **Die Grobstruktur eines evolutionären Algorithmus**

(1) **Initialisierung:** Eine Anfangspopulation P von μ Lösungen ist zu bestimmen.

Verschiedene Methoden kommen zum Ansatz wie etwa die zufällige Wahl von Lösungen oder die sorgfältige Wahl von Anfangslösungen über problem-spezifische Heuristiken. Im letzten Fall muss aber *Diversität* gewährleistet sein: Die Anfangspopulation sollte den gesamten Lösungsraum „repräsentieren“.

(2) Wiederhole, bis eine genügend gute Lösung gefunden wurde:

(2a) **Selektion zur Reproduktion:** Jede Lösung $y \in P$ wird mit ihrer Fitness $f(y)$ bewertet. Um Nachkommen der Population P zu erzeugen, wird zuerst eine Teilmenge $P' \subseteq P$ von Elternlösungen ausgewählt. Die Auswahlverfahren sind häufig randomisiert, um Diversität sicher zu stellen. Zu den häufig benutzten Auswahlverfahren gehören

- die zufällige Auswahl nach der Gleichverteilung: Jede Lösung in P wird mit gleicher Wahrscheinlichkeit gewählt,
- die zufällige Auswahl nach der Fitness: Falls f positiv ist, wird $y \in P$ mit Wahrscheinlichkeit $\frac{f(y)}{N}$ für $N = \sum_{z \in P} f(z)$ oder mit Wahrscheinlichkeit $\frac{e^{f(x)/T}}{M}$ für $M = \sum_{z \in P} e^{f(z)/T}$ und einen Temperaturparameter T gewählt oder die
- Turnier-Selektion: Wähle zuerst k Lösungen aus P nach der Gleichverteilung und lasse die k' fittesten der k ausgewählten Lösungen zu.

(2b) **Variation:** μ Nachkommen werden aus P' mit Hilfe von Mutations- und Crossover-Operatoren erzeugt.

(2c) **Selektion zur Ersetzung:** Die neue Generation ist festzulegen.
Wende Verfahren wie in Schritt (2a) an, um die neue Generation aus der alten Generation und ihren Nachkommen zu bestimmen. Zum Beispiel werden in der Plus-Auswahl die μ Fittesten aus den μ Lösungen der alten Generation und ihren λ Nachkommen bestimmt; man spricht von der $(\mu + \lambda)$-Selektion. In der Komma-Auswahl wird $\lambda \geq \mu$ angenommen, und die μ fittesten Nachkommen bilden die neue Generation; man spricht von der (μ, λ)-Selektion.

Bevor wir die Mutations- und Crossover-Operatoren beschreiben, schränken wir die Suchräume ein. Im Wesentlichen werden drei verschiedene Typen von Suchräumen betrachtet. Neben dem

[2]Natürlich können wir auch Maximierungsprobleme mit Simulated Annealing und Minimierungsproblem mit evolutionären Algorithmen lösen: Die Maximierung von f ist äquivalent zur Minimierung von $-f$.

n-dimensionalen Würfel $\mathbb{B}_n = \{0,1\}^n$ werden vor Allem der $\mathbb{R}^n$ sowie $\mathbb{S}^n$, der Raum aller Permutationen von n Objekten, angewandt.

Mutationsoperatoren

Wir beginnen mit **Bitmutationen** für den n-dimensionalen Würfel $\mathbb{B}^n$ Entweder flippt man jedes Bit einer Elternlösung $y \in \mathbb{B}^n$ mit einer kleinen Wahrscheinlichkeit p (mit $p \cdot n = \Theta(1)$) oder man ersetzt y durch einen Nachbarn y' in der k-Flip Nachbarschaft von y für kleine Werte von k.

Im $\mathbb{R}^n$ ersetzt man eine Elternlösung y durch $\mathbf{y'} = \mathbf{y} + \mathbf{m}$, wobei die Komponenten von m zufällig und unabhängig voneinander mit Erwartungswert 0 gewählt werden. Entweder man wählt die Komponenten m_i von m zufällig aus einem fixierten Intervall $[-a, a]$ oder man erlaubt unbeschränkte Komponenten, die zum Beispiel gemäß der Normalverteilung gezogen werden. Im letzten Fall erlaubt man eine hohe Standardabweichung bei schlechten Lösungen und reduziert die Standardabweichung je besser die Lösung ist.

Für den Permutationsraum $\mathbb{S}^n$ betrachtet man zum Beispiel **Austausch-** und **Sprungoperatoren**. In einem Austauschoperator wird ein Paar (i, j) mit $1 \leq i \neq j \leq n$ zufällig und gleichverteilt aus allen möglichen Paaren gezogen und die $i$te und $j$te Komponente der Elternlösung werden vertauscht. Auch für einen Sprungoperator wird ein Paar (i, j) mit $1 \leq i \neq j \leq n$ zufällig und gleichverteilt aus allen möglichen Paaren gezogen. Die $i$te Komponente y_i einer Elternlösung y wird an die Position j gesetzt und die restlichen Komponenten werden passend verschoben.

Crossover-Operatoren

$y_1, \ldots, y_k$ seien k Eltern, wobei $k \geq 2$ gelte.

Für den Würfel betrachtet man nur den Fall $k = 2$. Im **r-Punkt Crossover** wählt man r Positionen $i_1, \ldots, i_r$ mit $1 \leq i_1 < \cdots < i_r \leq n$. Der Sprößling z von y_1 und y_2 erbt die ersten i_1 Bits von y_1, die $i_2 - i_1$ Bits in den Positionen $[i_1 + 1, i_2]$ von y_2, die $i_3 - i_2$ Bits in den Positionen $[i_2 + 1, i_3]$ von y_1 und so weiter. Typischerweise wird $r = 1$ oder $r = 2$ gewählt.

Auch im $\mathbb{R}^n$ verwendet man den r-Punkt Crossover. Sehr verbreitet ist das **arithmetische Crossover**, in dem $\sum_{i=1}^{k} \alpha_i y_i$ zum Nachkommen von $y_1, \ldots, y_k$ wird. Der wichtige Spezialfall $\alpha_1 = \cdots = \alpha_k = \frac{1}{k}$ wird intermediäre Rekombination genannt.

Im Permutationsraum $\mathbb{S}^n$ wählt man meistens 2 Eltern y_1 und y_2. Zum Beispiel werden Positionen $1 \leq i_1 < i_2 \leq n$ zufällig ausgewürfelt. Die in den Positionen i_1 bis i_2 liegenden Komponenten von y_1 werden dann gemäß y_2 umgeordnet.

Beispiel 8.7 **Einfache Genetische Algorithmen**

Der evolutionäre Algorithmus mit

- Suchraum $\mathbb{B}^n$,
- zufälliger Auswahl nach Fitness für die Selektion zur Reproduktion,
- Bitmutation mit Wahrscheinlichkeit $p \leq \frac{1}{n}$ und 1-Punkt-Crossover sowie
- (μ, μ)-Selektion zur Ersetzung

wird ein einfacher genetischer Algorithmus genannt. Hier erzeugt man häufig zwei Nachkommen, wobei der zweite Nachkomme genau die Elternanteile erbt, die der erste Nachkomme nicht erhalten hat.

Die Wahrscheinlichkeit, dass ein Nachfahre durch Crossover erzeugt wird, liegt zwischen $0,5$ und $0,9$; die Mutation gilt hier als weniger wichtiger Hintergrundoperator. Andere evolutionäre Algorithmen betonen hingegen die Mutation.

Bemerkung 8.2 **(a)** Streng genommen sollte man evolutionäre Algorithmen nicht unter lokaler Suche subsumieren. Während die Mutation typisch für die lokale Suche ist, führt der Crossover-Operator globale Sprünge ein.

(b) Wir können auch Entscheidungsprobleme mit evolutionären Algorithmen lösen. Dazu betrachten wir exemplarisch das Entscheidungsproblem $KNF - SAT$. Hier bietet sich der Suchraum $\mathbb{B}^n$ aller Belegungen an. Wir transformieren $KNF - SAT$ in ein Optimierungsproblem, indem wir die Zahl erfüllter Klauseln maximieren.

Kapitel 9

Exakte Algorithmen*

Bisher haben wir mit Hilfe von Approximationsalgorithmen und Varianten der lokalen Suche versucht, gute Lösungen zu erhalten. Jetzt bestehen wir auf optimalen Lösungen. Für Entscheidungsprobleme werden wir Backtracking einsetzen, um festzustellen, ob eine Lösung existiert und wenn ja, um eine Lösung zu konstruieren. Für Optimierungsprobleme benutzen wir das Branch & Bound Verfahren, um eine optimale Lösung zu bestimmen. Schließlich stellen wir die Alpha-Beta Suche vor, um Gewinnstrategien für nicht-triviale Zei-Personen Spiele zu bestimmen. Alle Verfahren versuchen, den Lösungsraum, beziehungsweise den Raum der Spielstellungen intelligent zu durchsuchen und nur erfolgversprechende (potentielle) Lösungen oder Spielstellungen zu betrachten. Trotzdem können wir keine Wunder erwarten, sondern müssen im Allgemeinen auf den massiven Einsatz von Rechnerressourcen gefasst sein.

Wir beginnen mit dem Sonderfall, dass nicht zu große Minima existieren. Das nächste Beispiel zeigt am Beispiel des Vertex Cover Problems, dass wir dieses Vorwissen ausnutzen können.

9.1 VC für kleine Überdeckungen

Wir führen n Tests aus, wobei die Ergebnisse einiger weniger Tests stark verfälscht sind. Leider wissen wir nicht welche Testergebnisse verfälscht sind, sondern nur, dass von wenigen falschen Ergebnissen auszugehen ist. Um diese Ausreißer zu entdecken, wenden wir das Vertex Cover Problem an: Wir bauen einen ungerichteten Graphen mit n Knoten und setzen eine Kante $\{i, j\}$ ein, wenn die Ergebnisse des $i$ten und $j$ten Test zu stark voneinander abweichen. Die wenigen stark verfälschten Tests „sollten“ dann einerseits einem kleinen Vertex Cover entsprechen und ein kleinstes Vertex Cover „sollte“ andererseits aus allen Ausreißern bestehen.

Haben wir einen Vorteil in der exakten Lösung des VC-Problems, wenn wir wissen, dass ein Graph $G = (V, E)$ kleine Überdeckungen hat? Die Antwort ist erfreulicherweise positiv. Wir nehmen an, dass G ein Vertex Cover der (relativ kleinen) Größe höchstens K besitzt und suchen nach exakten Algorithmen mit einer Laufzeit der Form $f(K) \cdot \text{poly}(n)$.

Um das Optimierungsproblem zu lösen, genügt eine Lösung des Entscheidungsproblems „Hat G einen Vertex Cover der Größe k?“: Durch Binärsuche können wir dann das Optimum mit $\log_2 K$-maliger Lösung des Entscheidungsproblems berechnen, wenn wir wissen, dass G ein Vertex Cover der Größe K besitzt. Wann hat G einen Vertex Cover der Größe k? Nur dann, wenn der Graph nicht zu viele Kanten besitzt.

Lemma 9.1 *Der Graph G habe n Knoten und einen Vertex Cover der Größe k. Dann hat G höchstens $k \cdot n$ Kanten.*

Beweis: Ein Vertex Cover $C \subseteq V$ muss einen Endpunkt für jede Kante des Graphen besitzen. Ein Knoten kann aber nur Endpunkt von höchstens $n-1$ Kanten sein und deshalb hat G höchstens $k \cdot (n-1) \leq k \cdot n$ Kanten. □

Wir haben natürlich die Möglichkeit, alle k-elementigen Teilmengen $U \subseteq V$ zu durchlaufen und zu überprüfen, ob U ein Vertex Cover ist, aber dies bedeutet die Inspektion von $\binom{n}{k}$ Teilmengen. Hier ist ein weitaus schnelleres Verfahren. ($G-w$ ist der Graph G nach Herausnahme von Knoten w.)

Algorithmus 9.1 **VC(G, k)**

(1) Setze $U = \emptyset$.

/* U soll einen Vertex Cover der Größe höchstens k speichern. */

(2) Wenn G mehr als $k \cdot n$ Kanten hat, dann hat G nach Lemma 9.1 keinen Vertex Cover der Größe k. Der Algorithmus bricht mit einer „erfolglos“ Meldung ab. Ansonsten, für $k = 0$, brich mit einer „erfolgreich“ Meldung ab.

/* Wir können ab jetzt annehmen, dass G höchstens $k \cdot n$ Kanten besitzt. Beachte, dass der Aufruf für $k = 0$ stets abbricht. */

(3) Wähle eine beliebige Kante $\{u, v\} \in E$.

- Rufe **VC(G − u, k − 1)** auf. Wenn die Antwort „erfolgreich“ ist, dann setze $U = U \cup \{u\}$ und brich mit der Nachricht „erfolgreich“ ab.
- Ansonsten rufe **VC(G − v, k − 1)** auf. Wenn die Antwort „erfolgreich“ ist, dann setze $U = U \cup \{v\}$ und brich mit der Nachricht „erfolgreich“ ab.
- Ansonsten sind beide Aufrufe erfolglos. Brich mit der Nachricht „erfolglos“ ab.

/* Der Vertex Cover U muss einen Endpunkt der Kante $\{u, v\}$ besitzen. Algorithmus 9.1 untersucht zuerst rekursiv die Option $u \in U$ und bei Mißerfolg auch die Option $v \in U$. */

Aufgabe 73
Ist die Abfrage auf die Kantenzahl in Schritt (2) notwendig oder genügt die Abfrage, ob Kanten für $k = 0$ existieren?

Unser Algorithmus scheint nur mäßig intelligent zu sein, aber seine Laufzeit ist dennoch wesentlich schneller als die Überprüfung aller möglichen k-elementigen Teilmengen. Warum? Algorithmus 9.1 erzeugt mit seinem ersten Aufruf VC(G, k) einen Rekursionsbaum. Der Rekursionsbaum ist binär und hat Tiefe k. Also haben wir höchstens 2^k rekursive Aufrufe, wobei ein Aufruf höchstens Zeit $O(n)$ benötigt.

Satz 9.2 *Algorithmus 9.1 überprüft in Zeit $O(2^k \cdot n)$, ob ein Graph mit n Knoten einen Vertex Cover der Größe höchstens k besitzt.*

Wir können also das NP-vollständige Problem VC effizient lösen, wenn k höchstens ein (nicht zu großes) Vielfaches von $\log_2 n$ ist.

Man bezeichnet das Forschungsgebiet von Problemen mit fixierten Parametern auch als parametrisierte Komplexität.

9.2 Backtracking

Sämtliche Probleme in NP sind Entscheidungsprobleme, d.h. es ist zu entscheiden, ob es eine *tatsächliche* Lösung, wie etwa eine erfüllende Belegung in $KNF - SAT$, in einer Menge *potentieller* Lösungen, wie etwa der Menge aller Belegungen, gibt. Zusätzlich, wenn Lösungen existieren, möchten wir auch eine Lösung bestimmen.

Sei U das Universum aller potentiellen Lösungen. Backtracking versucht, eine Lösung Schritt für Schritt aus *partiellen* Lösungen zu konstruieren. Um die Arbeitsweise von Backtracking zu protokollieren, bauen wir einen Backtracking Baum $\mathcal{B}$, der anfänglich nur aus der Wurzel $v = U$. Backtracking arbeitet mit einem Branching-Operator B. Anfänglich wird B auf das Universum U angewandt und bestimmt eine Zerlegung $B(U) = U_1 \cup \cdots \cup U_k$; dementsprechend machen wir die Knoten $v_1 = U_1, \ldots, v_k = U_k$ zu Kindern der Wurzel. Im allgemeinen Schritt wählt Backtracking ein Blatt $v \subseteq U$ und macht die durch $B(v)$ beschriebenen Teilmengen von v zu Kindern von v im Baum $\mathcal{B}$: Der Branching Operator lässt den Knoten v in die Teilmengen der Zerlegung verzweigen.

Das Wachstum des Baums endet mit ein-elementigen Mengen, den potentiellen Lösungen. Ein innerer Knoten entspricht einer partiellen, also noch nicht vollständig entwickelten Lösung, während ein Blatt einer potentiellen Lösung entspricht, die möglicherweise eine tatsächliche Lösung ist. Beachte, dass damit eine partielle Lösung nicht anderes als eine Menge potentieller Lösungen ist. Der Branching-Operator B muss die Eigenschaft haben, dass jede tatsächliche Lösung als ein Blatt auftritt, also durch mehrmalige Anwendung von B auf die Wurzel U als ein-elementige Menge erhalten wird.

Wie sollten wir den Baum $\mathcal{B}$ durchsuchen?

- Teilmengen, für die ausgeschlossen ist, dass sie eine tatsächliche Lösung enthalten, werden nicht aktiviert. Alle nicht aktivierten Blätter werden nicht mehr expandiert: Unser Suchverfahren nimmt den Branching Schritt zurück und führt „Backtracking“ durch.
- Die **wesentliche Aufgabe** einer Backtracking Implementierung ist die frühzeitig Erkennung von Knoten ohne tatsächliche Lösungen: Wenn tatsächliche Lösungen nicht existieren, ist dies die einzige Möglichkeit einer signifikanten Verringerung der Suchzeit.
- Wenn eine tatsächliche Lösung gefunden wird, dann endet Backtracking.

Um einen möglichst kleinen Teil des Backtracking Baums zu durchsuchen, muss das jeweils zu expandierende Blatt intelligent gewählt werden.

Algorithmus 9.2 **Backtracking**

Anfänglich besteht der Backtracking Baum nur aus der Wurzel v. v ist aktiviert.

(1) Wiederhole, solange es noch aktivierte Blätter gibt

(1a) **Wähle das erfolgversprechendste** aktivierte Blatt v.

(1b) Mache die von $B(v)$ beschriebenen Teilmengen zu Kindern von v.

- Ein Kind wird allerdings nur dann aktiviert, wenn es möglicherweise eine tatsächliche Lösung enthält.
- Wird eine tatsächliche Lösung gefunden, dann wird der Algorithmus mit einer Erfolgsmeldung abgebrochen.

(2) Brich mit einer Erfolglos-Meldung ab.

Wir stellen Backtracking Implementierungen vor, die gegenüber der „Brute Force“ Aufzählung aller partiellen Lösungen zu einer signifikanten Beschleunigung führen.

Beispiel 9.1 $KNF - SAT$

Eine aussagenlogische Formel α in konjunktiver Normalform ist gegeben, und eine erfüllende Belegung, falls vorhanden, ist zu konstruieren. α besitze die Variablen $x_1, \ldots, x_n$.

Wir definieren das Universum U als die Menge $\{0,1\}^n$ aller Belegungen. Der Operator B wird sukzessive die Wahrheitswerte der Variablen festlegen. Die Knoten des Backtracking Baums $\mathcal{B}$ werden somit durch Paare (J, b) mit $J \subseteq \{1, \ldots, n\}$ und einer Belegung $b : J \to \{0,1\}$ beschrieben: Der Knoten entspricht der Menge aller Belegungen $x \in \{0,1\}^n$ mit $x_j = b(j)$ für alle $j \in J$.

Wie ist der Branching Operator B zu definieren? Wenn wir ein Blatt v mit Beschreibung (J, b) gewählt haben, dann ersetzen wir alle Variablen x_j für $j \in J$ durch ihre Wahrheitswerte $b(j)$ und wählen eine bisher nicht erfüllte Klausel k mit der geringsten Anzahl verbliebener Variablen. Schließlich fixieren wir eine beliebige Variable x_i in k und machen $(J \cup \{i\}, b_0)$ und $(J \cup \{i\}, b_1)$ zu Kindern von v, wobei $b_k(j) = b(j)$ für $j \in J$ und $b_k(i) = k$ ist.

Warum expandieren wir bezüglich einer kürzesten Klausel? Wir hoffen Sackgassen möglichst frühzeitig zu entdecken!

Ein Kind $(J \cup \{i\}, b_k)$ wird allerdings nur dann aktiviert, wenn der folgende Test bestanden wird. Wir setzen die Wahrheitswerte für die Variablen x_j mit $j \in J$ in die Formel α ein und suchen nach ein-elementigen Klauseln, die dann den Wert „ihrer“ Variablen festlegen. Wir setzen die Wahrheitswerte aller festgelegten Variablen in α ein und wiederholen dieses Vorgehen, bis wir einen Widerspruch gefunden haben oder bis alle verbliebenen Klauseln mindestens zwei-elementig sind. Nur im zweiten Fall wird $(J \cup \{i\}, b_k)$, erweitert um die Wahrheitswerte zwischenzeitlich festgelegter Variablen, aktiviert.

Aufgabe 74

Wir betrachten einen schwierigeren Test: Anstatt nach ein-elementigen Klauseln zu suchen, suchen wir diesmal nach höchstens zwei-elementigen Klauseln, wählen eine Variable aus und setzen diese Variable auf Null. Wir explorieren sämtliche Konsequenzen dieser Festlegung: werden wir auf keinen Widerspruch geführt, dann behalten wir die Festlegung bei und wiederholen unser Verfahren, falls möglich, mit einer weiteren höchstens zwei-elementigen Klausel. Werden wir hingegen auf einen Widerspruch geführt, setzen wir diesmal die Variable auf Eins. Bei einem erneuten Widerspruch erhält die partielle Belegung keine erfüllende Belegung, ansonsten machen wir uns wieder auf die Suche nach einer höchstens zwei-elementigen Klausel.

Kann dieser erweiterte Test effizient durchgeführt werden?

Wir definieren ein erfolgversprechendstes Blatt als das aktivierte Blatt v mit der kürzesten Klausel nach Einsetzen der festgelegten Wahrheitswerte. Diese Definition folgt somit der Definition des Branching Operators: Auch hier versuchen wir, Sackgassen so früh wie möglich zu entdecken.

Beispiel 9.2 $SUBSET - SUM$

Zahlen $t_1, \ldots, t_n \in \mathbb{N}$ und ein Zielwert $Z \in \mathbb{N}$ sind gegeben. Es ist festzustellen, ob es eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} t_i = Z$ gibt. Wir nehmen ohne Beschränkung der Allgemeinheit an, dass $t_1 \geq t_2 \geq \cdots \geq t_1$ gilt.

Wir werden die Zahlen nach absteigender Größe abarbeiten: Wenn wir uns bereits entschieden haben, welche der ersten i Zahlen in die Menge I aufzunehmen sind, dann erhalten wir zwei

Teilprobleme, nämlich die Frage, ob $i+1$ in die Menge I aufgenommen wird oder nicht. Dieses Vorgehen ist dem dynamischen Programmieransatz für das verwandte Rucksackproblem abgeschaut.

Dementsprechend zerlegt der Branching Operator B eine Auswahl $I \subseteq \{1, \ldots, i\}$ in die beiden Optionen $I \cup \{i+1\}$ und I. Das jeweilige Kind $J \subseteq \{1, \ldots, i\}$ wird allerdings nicht aktiviert, falls der Zielwert „überschossen“ wird ($\sum_{j \in J} t_j > Z$) oder falls der Zielwert nicht mehr erreichbar ist ($\sum_{j \in J} t_j + \sum_{j=i+1}^{n} t_j < Z$). Die Anordnung der Zahlen nach absteigender Größe hilft, Sackgassen durch das Überschießen und die Nicht-Erreichbarkeit frühzeitig zu entdecken.

Wir wählen das aktivierte Blatt, das am weitesten festgelegt ist, als erfolgversprechendstes Blatt. Damit durchsucht Backtracking in diesem Fall den Baum $\mathcal{B}$ nach dem Tiefensuche-Verfahren.

9.3 Branch & Bound

Wir versuchen jetzt eine optimale Lösung für ein Optimierungsproblem $(\min, f, L)$ zu bestimmen. Wie für Backtracking arbeiten wir mit einem Branching Operator B, dessen wiederholte Anwendung einen Branch & Bound Baum $\mathcal{B}$ erzeugt. Wir unterscheiden diesmal nicht mehr potentielle und tatsächliche Lösungen, vielmehr suchen wir eine Lösung y mit kleinstem Wert $f(y)$. Die wesentliche Neuerung gegenüber Backtracking ist aber die Annahme einer unteren Schranke. Für jeden Knoten v von $\mathcal{B}$ nehmen wir nämlich an, dass eine untere Schranke `unten(v)` gegeben ist, so dass

$$\text{unten}(v) \leq f(y)$$

für jede Lösung $y \in v$ gilt.

> v kann verworfen werden, wenn $\text{unten}(v) \geq f(y_0)$ für eine aktuelle beste Lösung y_0 gilt, denn keine Lösung in v ist besser als die Lösung y_0.

Algorithmus 9.3 **Branch & Bound**

Das Minimierungsproblem $(\min, f, L)$ sei zu lösen. Der Branching Operator B sei gegeben. Anfänglich besteht der Branch & Bound Baum $\mathcal{B}$ nur aus der (aktivierten) Wurzel.

(1) Eine Lösung y_0 wird mit Hilfe einer **Heuristik** berechnet.

(2) Wiederhole, solange es aktivierte Blätter gibt:

(2a) Wähle das **erfolgversprechendste** aktivierte Blatt v.

(2b) **Branching:** Wende den Branching Operator B auf v an, um die Kinder $v_1, \ldots, v_k$ zu erhalten. Inspiziere die k Teilmengen nacheinander:

- Wenn v_i eine Lösung y_i enthält, die besser als y_0 ist, dann setze $y_0 = y_i$ und deaktiviere gegebenenfalls Blätter.
- Ansonsten führe den **Bounding-Schritt** durch: Nur wenn $\text{unten}(v_i) < f(y_0)$, wird v_i aktiviert.

(3) Gib die Lösung y_0 als optimale Lösung aus.

Eine erfolgreiche Implementierung von Branch & Bound muss die folgenden Probleme lösen:

- Die Anfangslösung y_0 muss möglichst nahe am Optimum liegen, damit der Bounding-Schritt schlechte Knoten frühzeitig disqualifiziert. Aus genau demselben Grund muss die untere Schranke für v die Qualität der besten Lösung in v möglichst gut voraussagen.
- Die Wahl eines erfolgversprechendsten Blatts bestimmt die Art und Weise, in der $\mathcal{B}$ durchsucht wird und bestimmt damit den Speicherplatzverbrauch.
  - Tiefensuche schont den Speicherplatzverbrauch. Allerdings wird die schnelle Entdeckung guter Lösungen leiden, da stets mit dem letzten, und nicht mit dem besten Knoten gearbeitet wird.
  - Der große Speicherverbrauch schließt Breitensuche als ein praktikables Suchverfahren aus.
  - In der „best first search“ wird der Knoten v mit der niedrigsten unteren Schranke gewählt. Man versucht also, schnell gute Lösungen zu erhalten.
  - Häufig werden Varianten der Tiefensuche und der best-first search kombiniert, um einerseits mit dem vorhandenen Speicherplatz auszukommen und um andererseits gute Lösungen möglichst schnell zu entdecken.

Beispiel 9.3 **Das Rucksack-Problem**

n Objekte mit Gewichten $g_1, \ldots, g_n \in \mathbb{R}$ und Werten $w_1, \ldots, w_n \in \mathbb{N}$ sind vorgegeben ebenso wie eine Gewichtsschranke $G \in \mathbb{R}$. Der Rucksack ist mit einer Auswahl von Objekten zu bepacken, so dass einerseits die Gewichtsschranke G nicht überschritten wird und andererseits der Gesamtwert maximal ist.

Bevor wir den Branch & Bound Algorithmus entwerfen, betrachten wir das fraktionale Rucksackproblem. Hier dürfen wir Anteile x_i $(0 \leq x_i \leq 1)$ des $i$ten Objekts in den Rucksack packen. Das fraktionale Rucksackproblem kann sehr einfach mit einem Greedy-Algorithmus gelöst werden. Wir nehmen dazu ohne Beschränkung der Allgemeinheit an, dass die Objekte bereits absteigend nach ihrem „Wert pro Kilo“ sortiert sind, dass also

$$\frac{w_1}{g_1} \geq \frac{w_2}{g_2} \geq \cdots \geq \frac{w_n}{g_n}$$

gilt. Wenn $\sum_{i=1}^{k} g_i \leq G < \sum_{i=1}^{k+1} g_i$, dann erhalten wir eine optimale Lösung, wenn wir die Objekte $1, \ldots k$ einpacken und die Restkapazität des Rucksacks mit dem entsprechenden Anteil an Objekt $k+1$ füllen.

Angeregt durch den Greedy-Algorithmus sortieren wir die Objekte absteigend nach ihrem Wert pro Kilo. Jeder Knoten v des Branch & Bound Baums wird dann durch ein Paar (J, i) beschrieben: Die Objekte aus $J \subseteq \{1, \ldots, i\}$ wurden bereits, ohne die Kapazität G zu überschreiten, in den Rucksack gepackt. Der Branching Operator erzeugt die beiden Kinder mit den Beschreibungen $(J, i+1)$ und $(J \cup \{i+1\}, i+1)$ entsprechend dem Auslassen oder der Hinzunahme des $i+1$.sten Objekts.

Beachte, dass wir diesmal ein Maximierungsproblem lösen möchten und Branch & Bound benötigt eine obere statt einer unteren Schranke. Wir besorgen uns sehr gute obere Schranken mit Hilfe des Greedy-Algorithmus für das fraktionalen Rucksackproblem: Für den Knoten (J, i) berechnen wir die Restkapazität $G' = G - \sum_{j \in J} g_j$ und wenden den Greedy-Algorithmus auf die Objekte $i+1, \ldots, n$ mit der neuen Gewichtsschranke G' an. Da der Greedy-Algorithmus eine optimale Lösung des fraktionalen Rucksackproblems berechnet und da der optimale Wert des fraktionalen Problems mindestens so groß wie der optimale Wert des ganzzahligen Problems ist, haben wir die gewünschte obere Schranke gefunden.

Eine Anfangslösung y_0 können wir mit dem dynamischen Programmier-Algorithmus aus Satz 7.5 bestimmen. Schließlich ist eine erfolgversprechendste Lösung zu bestimmen. Die Wahl einer wertvollsten Bepackung ist sicherlich eine sinnvolle Option.

Bemerkung 9.1 Wir haben die obere Schranke im Rucksackproblem durch die Methode der Relaxation gefunden: Wir haben die Forderung der Ganzzahligkeit abgeschwächt und reelle Lösungen zugelassen. In vielen Fällen werden dadurch Problemstellungen stark vereinfacht.

Beispiel 9.4 Wir betrachten das metrische Traveling Salesman Problem $M - TSP$, das wir in Beispiel 3.3 eingeführt haben. n Städte $1, \ldots, n$ sowie Distanzen `länge(i,j)` zwischen je zwei Städten sind gegeben. Unser Ziel ist die Bestimmung einer kürzesten Rundreise, die jede Stadt genau einmal besucht.[1]

Eine Aufzählung aller Permutationen

Der Lösungsraum besteht somit aus allen Permutationen der n Punkte. Ein erster Lösungsansatz besteht in der Aufzählung aller möglichen Lösungen. Wie zählt man alle Permutationen von n Objekten in Zeit $O(n!)$ auf? Wir lösen ein noch allgemeineres Problem, nämlich die Aufzählung aller einfachen Wege der Länge s in einem (gerichteten oder ungerichteten) Graphen G. Wenn wir für G den vollständigen Graphen mit n Knoten wählen, erhalten wir die Menge aller Permutationen als die Menge aller einfachen Wege der Länge $s = n - 1$. Wir verwenden einen zur Tiefensuche ähnliche Ansatz. Angenommen, wir haben bereits einen einfachen Weg

$$(a_1, \ldots, a_i) = w$$

erzeugt. Unser Ziel ist die Erzeugung aller einfachen Wege der Länge s, die w fortsetzen. Dazu markieren wir zuerst die Knoten $a_1, \ldots, a_i$: Sie dürfen nicht mehr durchlaufen werden. Sukzessive werden wir w mit jedem Nachbarn oder Nachfolger von a_i fortsetzen müssen und dann den Nachbarn (Nachfolger) markieren. Was tun, wenn aber zum Beispiel alle Fortsetzungen von $(a_1, \ldots, a_i, b)$ berechnet wurden? Dann muss die Markierung von b entfernt werden, damit auch Wege der Form

$$(a_1, \ldots, a_i, c, \ldots, b, \ldots)$$

aufgezählt werden. Das folgende Programm benutzt das globale Array *nummer*, das auf -1 initialisiert ist. Die globale Variable *nr* hat den anfänglichen Wert 0. Das 2-dimensionale Array A speichert die Adjazenzmatrix von G.

```
void wege (int u, int s)
 {
   int v;
   nummer[u] = nr++;                          // u wird markiert.
   if (nr == s) gib das nr-Array aus;
   else
     for (v = 0; v < n; v ++)
       // alle unmarkierten Nachfolger werden besucht:
       if ( (nummer[v] == -1) && (A[u][v]))
         wege (v,s);
   nr --; nummer [u] = -1;                    // u wird freigegeben.
  }
```

[1] Die Webseite `www.tsp.gatech.edu` erhält weitere Informationen

Aufgabe 75
Bestimme die Laufzeit für $G = V_n$ und $s = n - 1$.

Wenn alle Permutationen aufzuzählen sind, dann verwalte alle nicht markieren Knoten in einer Schlange. (Wenn Knoten u besucht wird, dann füge zuerst ein Trennsymbol ein: Alle während `wege(u,s)` wieder freigegebenen Nachbarn von u werden nach dem Trennzeichen eingefügt und ein mehrfacher Besuch während der Ausführung von `wege(u,s)` kann verhindert werden.) Wenn die Ausgabe der Permutationen nicht gezählt wird, dann erhalten wir Laufzeit $O(n!)$, wenn in der for-Schleife die Schlange geleert wird.

Damit können wir also alle für das Problem des Handlungsreisenden benötigten Permutationen berechnen. Aber schon die Aufzählung aller Permutationen für $n = 18$ ist ein hoffnungsloses Unterfangen. Wir wenden deshalb Branch & Bound an.

Branch & Bound mit 1-Bäumen

Es gibt viele Heuristiken für das Traveling Salesman Problem, die wir zur Beschaffung einer Anfangslösung heranziehen können. Zum Beispiel haben wir die Spannbaum-Heuristik in Beispiel 3.3 kennengelernt. Eine erfolgreichere Heuristik ist die **Nearest-Insertion** Regel, die mit der kürzesten Kante beginnt und dann iterativ eine Rundreise konstruiert. In einem Iterationsschritt wird eine Stadt gewählt, die unter allen noch nicht erfassten Städten einen kleinsten Abstand zur gegenwärtigen partiellen Rundreise besitzt. Diese Stadt wird dann zwischen ein Paar benachbarter Städte der partiellen Rundreise eingefügt, wobei das Paar gewählt wird, das zu einem möglichst kleinen Längenanstieg führt. Man kann zeigen dass die Nearest-Insertion Regel 2-approximativ ist.

Unsere Branch & Bound Implementierung erzeugt einen Branch & Bound Baum $\mathcal{B}$, dessen Knoten durch **Paare** $(\mathbf{S}, \mathbf{T})$ beschrieben werden: $S \subseteq \{\ \{r, s\} \mid r \neq s\}$ ist eine Menge erzwungener Kanten, die eine Rundreise durchlaufen muss, und $T \subseteq \{\ \{r, s\} \mid r \neq s\}$ eine Menge verbotener Kanten. Das Ausgangsproblem wird durch $(\emptyset, \emptyset)$ beschrieben, nachfolgende Anwendungen des Branching Operators schreiben aber Kanten vor bzw. verbieten Kanten.

Das schwierige Problem im Entwurf von Branch & Bound Algorithmen ist im Allgemeinen die Ableitung guter unterer Schranken. Um die Diskussion zu erleichtern führen wir untere Schranken nur für das Ausgangsproblem $(\emptyset, \emptyset)$ ein. Wir haben in Beispiel 3.3 gesehen, dass die Länge eines minimalen Spannbaums eine untere Schranke für die Länge von Rundreisen ist. Wir zeigen jetzt, wie man diese untere Schranke deutlich verbessert. Zuerst beobachten wir, dass ein Spannbaum nur $n - 1$ Kanten besitzt, während eine Rundreise n Kanten durchläuft. Wir ersetzen deshalb die Spannbaum Methode durch die **Methode der 1-Bäume**:

> Wir wählen eine beliebige Stadt s aus und berechnen einen minimalen Spannbaum $B_{\min}$ für alle Städte bis auf Stadt s. Dann fügen wir s zu $B_{\min}$ hinzu, indem wir s mit den beiden nächstliegenden Städten verbinden.
>
> Durch die Hinzunahme von s und seiner beiden billigsten Kanten haben wir einen 1-Baum $B^*_{\min}$ erhalten, nämlich einen Graphen mit einem einzigen Kreis. Die Länge des 1-Baums ist unsere neue untere Schranke.

Haben wir tatsächlich eine untere Schranke erhalten? Wenn wir die Stadt s aus einer Rundreise entfernen, dann erhalten wir einen Spannbaum für die verbleibenden $n - 1$ Städte und die Länge dieses Spannbaums ist natürlich mindestens so groß wie die Länge eines minimalen Spannbaums. Schließlich haben wir s über seine beiden kürzesten Kanten mit dem minimalen

Spannbaum verschweißt und diese beiden Kanten sind nicht länger als die beiden Kanten der Rundreise.

Aber selbst die neue untere Schranke ist nicht gut genug. In dem Verfahren von Volgenant und Jonker werden deshalb die Distanzen $d_{r,s}$ zwischen Stadt r und Stadt s wie folgt geändert: Wir wählen Parameter u_r für jede Stadt r und legen $d^*_{r,s} = d_{r,s} + u_r + u_s$ als neue Distanz fest.

- Wenn L die Länge einer Rundreise für die alten Distanzen ist, dann ist $L + 2 \cdot \sum_{r=1}^{n} u_r$ die Länge für die neuen Distanzen. Insbesondere wird eine für die neuen Distanzen optimale Rundreise auch für die alten Distanzen optimal sein.

- Sei $\text{grad}_{B*}(r)$ die Anzahl der Nachbarn von Stadt r in einem beliebigen 1-Baum B^*. Wenn $l(B^*)$ die Länge von B^* für die alten Distanzen ist, dann ist $l(B^*) + \sum_{i=r}^{n} u_r \cdot \text{grad}_{B^*}(r)$ die Länge von B^* für die neuen Distanzen.

Wie sollte der Vektor $u = (u_1, \ldots, u_n)$ gewählt werden? Für eine optimale Rundreise mit Länge L_{opt} für die alten Distanzen und für jeden Vektor u gilt

$$L_{\text{opt}} + 2 \cdot \sum_{r=1}^{n} u_r \geq \min_{B^*} l(B^*) + \sum_{r=1}^{n} u_r \cdot \text{grad}_{B^*}(r), \tag{9.1}$$

da wir eine Rundreise mit einem minimalen 1-Baum vergleichen. Wir betrachten deshalb die Funktion

$$f(u) = \min_{B^*} l(B^*) + \sum_{r=1}^{n} u_r \cdot (\text{grad}_{B^*}(r) - 2).$$

Es ist stets $L_{\text{opt}} \geq f(u)$ für jeden Vektor u, und wir sollten deshalb f zumindest approximativ maximieren, um eine möglichst gute untere Schranke $f(u)$ zu erhalten.

Das **erfolgversprechendste Blatt** $v = (S, T)$ wählen wir stets gemäß Tiefensuche. Um den **Branching Operator** für das Blatt v zu beschreiben, nehmen wir zuerst wieder $S = T = \emptyset$ an. Angenommen, wir haben den Vektor u^* für die Bestimmung der unteren Schranke berechnet und $B^*_{\min}$ ist ein minimaler 1-Baum für die neuen Distanzen. Wir versuchen mit dem Branching Operator entweder $B^*_{\min}$ als optimalen 1-Baum auszuschalten oder aber ein signifikant einfacheres TSP-Problem zu erhalten.

Zuerst beachten wir, dass $B^*_{\min}$ eine Stadt s vom Grad mindestens drei besitzt, denn sonst ist $B^*_{\min}$ bereits eine Rundreise und wegen (9.1) sogar eine optimale Rundreise. Die Kanten $\{r, s\}$ und $\{t, s\}$ seien die längsten, mit s inzidenten Kanten von $B^*_{\min}$. Der Branching Operator erzeugt drei Kinder, wobei alle Kinder $B^*_{\min}$ als optimalen 1-Baum ausschalten.

(1) Für das erste Kind verbieten wir die Benutzung der Kante $\{r, s\}$.

(2) Für das zweite Kind wird die Benutzung von $\{r, s\}$ erzwungen, aber die Benutzung von $\{t, s\}$ verboten.

(3) Für das dritte Kind wird sowohl die Benutzung von $\{r, s\}$ wie auch die Benutzung von $\{t, s\}$ erzwungen. Alle anderen mit s inzidenten Kanten werden verboten.

Wird im folgenden ein Branching-Schritt für einen Knoten s durchgeführt, der bereits eine erzwungene Kante besitzt, dann erzeugt man das dritte Kind nicht und verbietet für das zweite Kind alle sonstigen mit s inzidenten Kanten, bis auf die erzwungene Kante.

9.3.1 Branch & Cut Verfahren

Wir haben in der NP-Vollständigkeit ausgenutzt, dass viele wichtige Entscheidungsprobleme äquivalent sind und haben dementsprechend die große Klasse der NP-vollständigen Probleme erhalten. Wenn wir aber Optimierungsprobleme betrachten, dann zeigt sich eine große Variation: Einige Optimierungsprobleme wie etwa das Rucksackproblem sind relativ harmlos und lassen, wenn wir moderat mit Laufzeit bezahlen, beliebig gute Approximationen zu. Andere Probleme wie etwas die ganzzahlige Programmierung sind extrem harte Brocken, die im worst-case keine signifikanten Approximationen durch effiziente Algorithmen erlauben.

Wir konzentrieren uns dennoch jetzt auf die ganzzahlige Programmierung und betrachten beispielhaft eine Formulierung des Traveling Salesman Problems durch das folgende 0-1 Programm. Wir nehmen an, dass wir eine kürzeste Rundreise für n Städte bestimmen müssen, wobei $d_{r,s}$ die Distanz zwischen Stadt r und Stadt s sei. Die 0-1 wertigen Variablen $x_{r,s}$ geben an, ob wir die Kante $\{r,s\}$ wählen ($x_{r,s}=1$) oder nicht ($x_{r,s}=0$).

$$\min \sum_{r=1}^{n} \sum_{s \neq r} d_{r,s} \cdot x_{r,s} \quad \text{so dass} \quad \sum_{r,r \neq s} x_{r,s} = 2 \text{ für jede Stadt } s, \tag{9.2}$$

$$\sum_{r \in S, s \notin S} x_{r,s} \geq 2 \text{ für jede nicht-leere, echte Teilmenge } S \subseteq V$$

$$\text{und } x_{r,s} \in \{0,1\} \text{ für alle } r \neq s.$$

Wir fordern also zuerst, dass genau zwei mit s inzidente Kanten für jede Stadt s gewählt werden. Selbst verknüpft mit der Forderung, dass die Variablen $x_{r,s}$ nur die Werte 0 oder 1 annehmen, sind diese Gleichungen nicht ausreichend: Wir lassen mehrere disjunkte Kreise, oder Subtouren, zu. Deshalb haben wir noch die *Subtour-Eliminationsbedingungen* hinzugefügt, die für jede nichtleere, echte Teilmenge S fordern, dass mindestens zwei Kanten Städte in S mit den verbleibenden Städten verbinden.

Aufgabe 76
Zeige, dass das Programm (9.2) nur Rundreisen als Lösungen besitzt.

Aber leider haben wir, neben den Integralitätsforderungen, eine Ungleichung für jede nichtleere, echte Teilmenge von $\{1,\ldots,n\}$ und damit müssen wir mit den viel zu vielen 2^n-2 Ungleichungen arbeiten.

> Wir entfernen sämtliche Integralitätsbedingungen $x_{r,s} \in \{0,1\}$ wie auch sämtliche Subtour-Eliminationsbedingungen. Stattdessen setzen wir darauf, dass nur wenige Subtour-Eliminationsbedingungen entscheidend sind und versuchen, diese entscheidenden Bedingungen eigenständig zu bestimmen.

Das neue lineare Programm lässt sich effizient lösen, aber die optimale Lösung `opt` wird eine Vereinigung mehrerer Kreise sein. (Man kann zeigen, dass alle Ecken 0-1 wertig sind und deshalb einer Sammlung von Kreisen entsprechen.) Dann aber wird eine Subtour-Eliminationsbedingung verletzt sein. Wir fügen diese Bedingung, auch **Cut** genannt, hinzu und wiederholen unser Vorgehen. Allerdings müssen wir mit zwei Schwierigkeiten kämpfen:

- Das Hinzufügen der verletzten Subtour-Eliminationsbedingungen führt dazu, dass die Ecken der neuen Polytope im Allgemeinen nicht mehr integral sein werden. Dementsprechend ist das Auffinden der von `opt` verletzten Subtour-Eliminationsbedingungen schwieriger geworden.

- Die Subtour-Eliminationsbedingungen reichen im Allgemeinen nicht aus: Wenn wir die Integralitätsbedingungen $x_{r,s}$ entfernen und sämtliche Subtour-Eliminationsbedingungen hinzufügen, dann kann es optimale Lösungen `opt` geben, die keiner Rundreisen entsprechen.

Deshalb, wenn das Auffinden verletzter Bedingungen zu schwierig oder gar unmöglich geworden ist, übergeben wir das Problem einem Branch & Bound Verfahren; zumindest sollten wir eine gute untere Schranke erhalten haben.

Wir wenden einen Branching Operator an, der eine oder mehrere Variablen so festlegt, dass die bisher optimale Lösung `opt` ausgeschaltet wird. (Dieses Vorgehen ähnelt somit dem Ausschalten optimaler 1-Bäume in Beispiel 9.4.) Für die Kinder-Probleme werden wir wieder nach verletzten Bedingungen suchen und den Branching Operator anwenden, wenn dies nicht mehr möglich ist. Diesen Prozess wiederholen wir solange, bis wir eine optimale Lösung gefunden haben.

Dieses Verfahren, um zusätzliche Bedingungen sowie zusätzliche Heuristiken erweitert, ist der Grundbaustein moderner Optimierungsmethoden für TSP. Probleme mit mehreren Tausend Städten sind mittlerweile exakt lösbar. Der gegenwärtige Weltrekord ist eine schwierige TSP-Instanz mit 33810 Städten.

Bemerkung 9.2 Das gerade vorgestellte Verfahren ist ein **Branch & Cut Algorithmus**, das für die Lösung allgemeiner Instanzen der ganzzahligen Programmierung benutzt wird. Zuerst werden die Integralitätsbedingungen entfernt. Wenn die optimale Lösung `opt` integral ist, dann haben wir das Optimierungsproblem gelöst. Ansonsten müssen wir eine neue Bedingung erfinden, die

> von allen integralen Lösungen erfüllt wird, aber von der fraktionalen Lösung `opt` verletzt wird.

Wir fügen diese Bedingung, bzw. diesen Cut, dem Programm hinzu und wiederholen unser Vorgehen. Wenn wir hingegen nach einiger Zeit keine verletzten Bedingungen mehr finden können, dann übergeben wir an ein Branch & Bound Verfahren. Der Branching Operator produziert Kinder-Probleme, die das bisherige Optimum ausschalten, und wir versuchen, weitere Bedingungen für die jeweiligen Kinder-Probleme zu erfinden, um einer optimalen integralen Lösung näher zu kommen.

9.4 Der Alpha-Beta Algorithmus

Wir betrachten ein Zwei-Personen Spiel mit den Spielern Alice und Bob. Alice beginnt und die beiden alternieren mit ihren Zügen. Wir fordern, dass alle möglichen Spiele endlich sind und mit einer „Auszahlung“ an Alice enden: Wenn die Auszahlungen nur $-1, 0$ oder 1 sind, dann gewinnt Alice bei der Auszahlung 1, während Bob bei der Auszahlung -1 gewinnt. Wir stellen uns die Aufgabe, eine Gewinnstrategie für Alice zu bestimmen, also eine Strategie, die ihr eine größtmögliche Auszahlung garantiert. Wir haben bereits in Kapitel 6.3.5 gesehen, dass eine effiziente Bestimmung von Gewinnstrategien für nicht-triviale Spiele ausgeschlossen ist. Wir müssen deshalb auch hier versuchen, das unter den Umständen Beste zu erreichen.

Jedes solche Spiel besitzt einen Spielbaum $\mathcal{B}$. Die Wurzel r entspricht der Ausgangsstellung und ist mit dem zuerst ziehenden Spieler, also mit Alice markiert. Wenn der Knoten v der Stellung S_v entspricht und wenn v mit dem aktuell ziehenden Spieler markiert ist, dann

erzeugen wir für jeden möglichen Zug des ziehenden Spielers ein Kind w von v: Das Kind w ist mit dem Gegenspieler markiert und entspricht der durch den gerade ausgeführten Zug veränderten Stellung S_v. Wenn hingegen das Spiel in v entschieden ist, dann wird v zu einem Blatt, und wir markieren v mit der Auszahlung $A(v)$ an Alice.

Die bestmögliche Auszahlung an Spieler können wir mit der **Minimax**-Auswertung von $\mathcal{B}$, einer Anwendung der Tiefensuche, bestimmen. Dazu sagen wir, dass ein mit Alice beschrifteter Knoten ein **Max-Knoten** und ein mit Bob beschrifteter Knoten ein **Min-Knoten** ist.

Algorithmus 9.4 Die Funktion Minimax(v)

(1) Wenn v ein Blatt ist, dann gib den Wert $A(v)$ zurück.

(2) Wenn v ein Max-Knoten ist, dann // Alice ist am Zug.

- Max $= -\infty$,
- durchlaufe alle Kinder w von v und setze Max $= \max\{\text{Max}, \text{Minimax}(w)\}$.
  // Alice wählt den für sie besten Zug.
- Gib den Wert Max zurück.

(3) Wenn v ein Min-Knoten ist, dann // Bob ist am Zug.

- Min $= \infty$,
- durchlaufe alle Kinder w von v und setze Min $= \min\{\text{Min}, \text{Minimax}(w)\}$.
  //Der Zug von Bob minimiert die Auszahlung an Alice.
- Gib den Wert Min zurück.

Tatsächlich können wir die Minimax-Auswertung wesentlich beschleunigen. Betrachten wir dazu die Auswertung eines Max-Knotens v und nehmen wir an, dass v einen Min-Knoten u als Vorfahren besitzt, dessen Min-Variable zum Zeitpunkt der Auswertung von v den Wert β besitzt.

Um den Max-Knoten v auszuwerten, betrachten wir sämtliche Kinder w von v, also sämtliche Züge von Alice. Wenn Alice für irgendein Kind w eine Auszahlung größer oder gleich β erzwingt, dann kann Bob das Erreichen von v durch seine vorigen Züge unbeschadet verhindern: Zum Beispiel kann er im Knoten u so ziehen, dass v nicht erreichbar ist und die Auswertung von Alice auf höchstens β beschränkt ist. Die Auswertung von v kann nach der Auswertung des Kinds w abgebrochen werden!

Natürlich gilt ein analoges Argument für die Auswertung eines Min-Knotens v: Sei der Vorfahre u ein Max-Knoten und seine Max-Variable habe gegenwärtig den Wert α. Wenn Bob einen besonders cleveren Zug (v, w) findet, der die Auszahlung an Alice auf höchstens α beschränkt, dann wird Alice so ziehen, dass v nicht erreichbar ist und die Auswertung in v kann deshalb ebenfalls abgebrochen werden.

Wie sind α bzw β zu definieren, wenn wir einen Min-Knoten bzw. Max-Knoten v auswerten?

> α sollte der maximale Wert der Max-Variable eines „Max-Vorfahrens“ von v sein und β sollte der minimale Wert der Min-Variable eines „Min-Vorfahrens“ von v sein.

Wenn wir diese Überlegungen umsetzen, dann werden wir auf den Alpha-Beta Algorithmus geführt, der den Spielbaum wieder mit Hilfe der Tiefensuche auswertet, aber die Suche intelligent mit Hilfe der Parameter α und β beschleunigt.

Algorithmus 9.5 Die Funktion Alpha-Beta (v, α, β).

(1) Wenn v ein Blatt ist, dann gib den Wert $A(v)$ zurück.

(2) Ansonsten durchlaufe alle Kinder w von v:

- Wenn v ein Max-Knoten ist, dann setzte $\alpha = \max\{\ \alpha,\ \text{Alpha-Beta}(w, \alpha, \beta)\ \}$;
  /* Wenn zu Anfang der Auswertung von v der Wert von α mit dem maximalen Wert der Max-Variable eines Max-Vorfahrens übereinstimmt, dann wird sichergestellt, dass diese Eigenschaft auch für die Kinder w von v gilt: Wenn $\alpha <$ Alpha-Beta(w, α, β), dann ist (v, w) ein besserer Zug für Alice. */
  Wenn $\alpha \geq \beta$, dann terminiere mit Antwort α.
  /* Bob wird das Erreichen von v unbeschadet verhindern können. */
- Wenn v ein Min-Knoten ist, dann setzte $\beta = \min\{\ \beta,\ \text{Alpha-Beta}(v, \alpha, \beta)\ \}$;
  /* Wenn zu Anfang der Auswertung von v der Wert von β mit dem minimalen Wert der Min-Variable eines Min-Vorfahrens übereinstimmt, dann wird sichergestellt, dass diese Eigenschaft auch für die Kinder w von v gilt: Wenn $\beta >$ Alpha-Beta(w, α, β), dann ist (v, w) ein besserer Zug für Bob. */
  Wenn $\alpha \geq \beta$, dann terminiere mit Antwort β.
  /* Alice wird das Erreichen von v unbeschadet verhindern können. */

(3) Gib den Wert α zurück, wenn v ein Max-Knoten ist. Ansonsten gib den Wert β zurück.

Was berechnet Alpha-Beta(v, α, β)?

Lemma 9.3 *A sei die größte von Alice erreichbare Auszahlung im Teilbaum mit Wurzel v.*

(a) Wenn v ein Max-Knoten ist, dann wird $\max\{\alpha, A\}$ ausgegeben, falls $A \leq \beta$.

(b) Wenn v ein Min-Knoten ist, dann wird $\min\{\beta, A\}$ ausgegeben, falls $\alpha \leq A$.

Aufgabe 77
Zeige Lemma 9.3.

Wie sollten die Parameter des ersten Aufrufs gewählt werden? Sicherlich sollten wir die Auswertung von $\mathcal{B}$ an der Wurzel r beginnen. Wenn wir $\alpha = -\infty$ und $\beta = \infty$ setzen, dann garantiert Lemma 9.3, dass die maximal erreichbare Auszahlung an Alice berechnet wird. Der erste Aufruf hat also die Form

$$\text{Alpha-Beta}(r, -\infty, \infty).$$

Aufgabe 78
(a) Was ist der endgültige Wert von β nach Beendigung des Aufrufs Alpha-Beta $(r, -\infty, \infty)$?

(b) Wie kann man einen optimalen ersten Zug von Alice bestimmen?

Die nächste Aufgabe zeigt, dass der Alpha-Beta Algorithmus im Best-Case nur $\Theta(\sqrt{N})$ Knoten besucht, wenn der Spielbaum aus N Knoten besteht.

Aufgabe 79
Sei $\mathcal{B}$ ein vollständiger b-ärer Baum der Tiefe d. Die Blätter seien mit reellen Zahlen beschriftet.

(a) Zeige, dass $\mathcal{B}$ einen kritischen Pfad besitzt, also eine Zugfolge, die von beiden Spieler optimal gespielt wird.

(b) Zeige, dass es eine Alpha-Beta Auswertung von $\mathcal{B}$ gibt, in der höchstens opt $= b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$ Knoten besucht werden.

Im besten Fall wird die Anzahl besuchter Knoten also von $\Theta(b^d)$ auf $\Theta(\sqrt{b^d})$ reduziert. Im Vergleich zur Minimax-Auswertung kann damit die Anzahl simulierter Züge, bei gleichen Ressourcen, verdoppelt werden.

(c) Zeige, dass jede Alpha-Beta Auswertung von $\mathcal{B}$ mindestens opt Knoten besucht.

Die experimentelle Erfahrung zum Beispiel für Schachprogramme zeigt, dass dieser Best-Case auch tatsächlich approximativ erreicht wird. Um den Grund hierfür einzusehen betrachten wir den typischen Aufbau eines Schachprogramms, das aus einer *Bewertungsprozedur*, die Stellungen mit reellen Zahlen bewertet, und einem *Suchalgorithmus* besteht, der die Konsequenz eines jeden möglichen Zugs für die „nächsten“ Züge vorausberechnet. Der Suchalgorithmus basiert meistens auf Varianten des Alpha-Beta Algorithmus, wobei die Tiefensuche hinter dem Alpha-Beta Algorithmus über eine Heuristik gesteuert wird: Tiefensuche setzt seine Suche mit der am höchsten bewerteten Nachfolgestellung fort. Das Erreichen des besten Falls ist also nicht weiter überraschend.

Leider kann aber auch Alpha-Beta optimale Zugfolgen nur für triviale Spiele effizient bestimmen. Zum Beispiel geht man im Schachspiel von ungefähr 38^{84} verschiedenen Stellungen aus und selbst eine Reduktion auf 38^{42} hilft nicht wirklich. Deshalb versuchen Schachprogramme eine möglichst weit gehende Vorausschau, um die Stärke eines Zuges möglichst gut abschätzen zu können; die Bewertungsprozedur wird herangezogen, um die erreichten Endkonfigurationen zu bewerten. Hier zeigt sich die wahre Stärke von Alpha-Beta, da es die Anzahl simulierter Züge im Vergleich zur Minimax-Auswertung verdoppelt!

Teil IV

Berechenbarkeit

Können wir alle algorithmischen Probleme lösen, wenn Laufzeit und Speicherplatz keine Rolle spielen? Um diese Frage für heutige und zukünftige Rechnergenerationen zu beantworten, formalisieren wir den Begriff eines Rechners wieder durch das Modell der Turingmaschinen und erinnern uns, dass Turingmaschinen sogar mit parallelen Supercomputern mithalten können, solange wir einen polynomiellen Anstieg der Rechenzeit tolerieren. Im nächsten Kapitel skizzieren wir eine Simulation von Quantenrechnern durch Turingmaschinen, müssen dabei aber eine möglicherweise expoentiell angestiegene Laufzeit in Kauf nehmen: Wenn uns aber die Laufzeit egal ist, können wir mit Fug und Recht behaupten, dass deterministische Turngmaschinen sogar Quantenberechnungen simulieren können.

Wir werden feststellen, dass es zu komplexe Probleme, nämlich die sogennanten unentscheidbaren Probleme gibt und nicht nur das, sondern die meisten Probleme sind sogar zu hart! Aber wir sollten auch erwarten, dass die meisten Probleme herzlich uninteressant sind: Wie sehen also interessante Probleme aus, die wir nicht knacken können? Wir zeigen, dass Super-Compiler nicht existieren, wobei ein Super-Compiler nicht nur ein Anwenderprogramm übersetzt, sondern auch überprüft, ob das Anwenderprogramm auf allen Eingaben hält. Und wenn nur überprüft werden soll, ob das Anwenderprogramm auf einer einzigen, vorher spezifizierten Eingabe hält? Auch das ist zuviel verlangt, weil wir zeigen werden, dass die Überprüfung irgendeiner nicht-trivialen Eingeschaft von Anwenderprogrammen auf ein nicht entscheidbares Problem führt.

Kapitel 10

Berechenbarkeit und Entscheidbarkeit

Wir arbeiten wieder mit den in Abschnitt 5.1 eingeführten Turingmaschinen, um eine technologie-unabhängige Definition eines Rechnermodells angeben zu können. Welche Funktionen lassen sich durch Rechner irgendeiner zukünftigen Rechnergeneration berechnen und welche Entscheidungsprobleme lassen sich lösen? Wir nehmen den Mund gewaltig voll und beantworten die Frage mit der Klasse aller Entscheidungsprobleme, die von einer stets haltenden Turingmaschine gelöst werden.

Definition 10.1 *Σ, Σ_1 und Σ_2 seien Alphabete.*

*(a) Eine partielle Funktion $f: \Sigma_1^* \to \Sigma_2^*$ heißt genau dann **berechenbar**, wenn es eine Turingmaschine M gibt, die für jede Eingabe $x \in \Sigma_1^*$ hält und die Ausgabe $f(x)$ als Ausgabe produziert, solange f auf x definiert ist.*
Wenn f nicht auf x definiert ist, dann fordern wir, dass M auf Eingabe x nicht hält.

*(b) Ein Entscheidungsproblem $L \subseteq \Sigma^*$ heißt genau dann **entscheidbar**, wenn es eine Turingmaschine M gibt, die für jede Eingabe $x \in \Sigma^*$ hält, so dass*

$$x \in L \;\Leftrightarrow\; M \text{ akzeptiert } x.$$

gilt.

Beispiel 10.1 Wir werden im nächsten Abschnitt viele Beispiele entscheidbarer Probleme kennenlernen, hier betrachten wir nur berechenbare, bzw. nicht berechenbare Funktionen.

- Die Funktion $f : \{1\}^* \to \{1\}^*$ mit $f(1^n) := 1^{2^n}$ für alle $n \in \mathbb{N}$ ist berechenbar.
- Die Funktion $f_\emptyset$ mit leerem Definitionsbereich ist berechenbar durch einen Algorithmus, der nur aus einer Endlosschleife besteht.
- Die Funktion g, die jedem ungerichteten Graphen mit endlicher Knotenmenge die Anzahl seiner Zusammenhangskomponenten zuordnet, ist berechenbar. (Hier müssen wir geeignete Alphabete Σ_1 und Σ_2 bestimmen. Wie sollten wir dies tun?)
- Die Funktion h, die jedem Programm $\langle M \rangle$ und jeder Eingabe w den Wert 1 (bzw. 0) zuordnet, wenn M auf Eingabe w hält (bzw. nicht hält), ist –wie wir später sehen werden– nicht berechenbar.

10.1 Die Church-Turing These

Die **Church-Turing These** behauptet, dass eine Funktion genau dann von Rechnern irgendeiner zukünftigen Rechnergeneration bestimmt werden kann, wenn die Funktion berechenbar ist, genauer:

> *Es gibt genau dann ein stets haltendes „Rechenverfahren" zur Berechnung einer Funktion, wenn diese Funktion durch eine stets haltende Turingmaschine berechnet werden kann.*

Die Church-Turing These behauptet also, dass der intuitive Begriff der Berechenbarkeit mit der Berechenbarkeit durch Turingmaschinen übereinstimmt. Man beachte, dass die Church-Turing These, wenn richtig, weitgehende Konsequenzen hat: Zum Beispiel sollte dann die Funktionsweise des menschlichen Genoms „berechenbar" sein, da wir das Genom als ein Rechenverfahren auffassen können, dass die Reaktionen des Körpers auf innere oder äußere Einflüsse, die Eingaben, steuert.

Im nächsten Abschnitt zeigen wir, dass nichtdeterministische, probabilistische und Quantenberechnungen sogar von deterministischen Turingmaschinen simuliert werden können. Damit erhalten wir überzeugendes Beweismaterial für die Richtigkeit der Church-Turing These. Man beachte aber, dass die Church-Turing These zwar durch die Angabe eines neuen revolutionären Rechenverfahrens widerlegbar ist, aber nicht im positiven Sinne bewiesen werden kann, da das Konzept eines Rechenverfahrens nicht formal definiert ist.

10.2 Entscheidbare Probleme

Wie groß ist die Klasse der entscheidbaren Entscheidungsprobleme? Natürlich sind alle Problem in `P` entscheidbar und damit sind insbesondere auch alle regulären und kontextfreien Sprachen entscheidbar. Sind auch alle Entscheidungsprobleme in `NP` entscheidbar? Wir müssen uns diese Frage stellen, da wir Entscheidbarkeit über deterministische Turingmaschinen definiert haben, während Entscheidungsproblem in `NP` über deterministische Turingmaschinen definert werden.

Die Antwort haben wir bereits in Satz 5.6 gegeben: Eine nichtdeterministische Turingmaschine M, die in Zeit $q(n)$ rechnet, kann durch eine äquivalente deterministische Turingmaschine M' simuliert werden, die in Zeit höchstens $2^{O(q(n))}$ rechnet. Wie haben wir dieses Ergebnis nachgewiesen? Wir haben M mit einer deterministischen Turingmaschine M' simuliert, die alle möglichen Berechnungen von M aufzählt und genau dann akzeptiert, wenn mindestens eine Berechnung von M akzeptiert. Da M höchstens $q(n)$ Schritte ausführt, gibt es höchstens

$$2^{O(q(n))}$$

verschieden Berechnungen. Insbesondere ändert sich die Klasse entscheidbarer Probleme also nicht, wenn wir Entscheidbarkeit über nichtdeterministische Turingmaschinen definieren!

Stimmt `NP` möglicherweise mit der Klasse entscheidbarer Probleme überein? Um Himmels willen, nein: In Bemerkung 5.1 haben wir uns das Tautologie-Problem angeschaut und festgestellt, dass „hochwahrscheinlich" selbst nichtdeterministische Turingmaschinen keine Chance haben in polynomieller Zeit zu überprüfen, ob eine KNF-Formel eine Tautologie ist.

Die Klasse entscheidbarer Probleme ist riesig und enthält zum Beipiel für jedes in endlicher Zeit endende Zwei-Personen Spiel das Problem festzustellen, ob der erste Spieler in einer gegebenen Spielsituation einen Gewinnzug hat. (Siehe Abschnitt 6.3.5.)

Aufgabe 80
Wir definieren das Spiel *GEOGRAPHY*. Dieses Spiel verallgemeinert das Geographie-Spiels, bei dem zwei Spieler abwechselnd noch nicht genannte Städtenamen wählen, wobei jede Stadt mit dem Endbuchstaben der zuvor genannten Stadt beginnen muss.

Eingabe: Ein gerichteter Graph $G = (V, E)$ und ein ausgezeichneter Knoten $s \in V$.

Das Spiel: Zwei Spieler A und B wählen abwechselnd jeweils eine noch nicht benutzte Kante aus E. Spieler A fängt an.

Die zu Beginn des Spiels gewählte Kante muss eine Kante mit Startknoten s sein. Jede anschließend gewählte Kante muss im Endknoten der zuvor gewählten Kante beginnen.

Der Spieler, der als erster keine solche unbenutzte Kante mehr findet, verliert das Spiel.

Es ist zu entscheiden, ob ein optimal spielender Spieler A gegen jeden Spieler B auf G gewinnen kann. **Zeige**, dass *GEOGRAPHY* durch eine deterministische Turingmaschine mit höchstens polynomiellem Speicherplatz gelöst werden kann. Warum gehört *GEOGRAPHY* wahrscheinlich nicht zur Klasse NP?

Wenn wir nur Spiele mit "polynomiell langen" Zugfolgen betrachten, dann erhalten wir das Entscheidungsproblem QBF als Prototyp eines solchen Spiels, das zwischen einem Existenzspieler $\exists$ und einem Allspieler $\forall$ gespielt wird.

$$QBF = \{\beta \quad | \quad \beta = \exists_{x_1}\forall_{x_2}\exists_{x_3}\cdots E(x_1,\ldots x_n) \text{ oder } \beta = \forall_{x_1}\exists_{x_2}\forall_{x_3}\cdots E(x_1,\ldots x_n),\\ E \text{ ist eine Formel der Aussagenlogik und } \beta \text{ ist wahr. }\}.$$

Eine quantifizierte Formel $\exists_{x_1}\forall_{x_2}\exists_{x_3}\cdots E(x_1,\ldots x_n)$ gehört genau dann zu QBF, wenn der Existenzspieler einen Gewinnzu hat, also den richtigen Wert $x_1 = 0$ oder $x_1 = 1$ bestimmen kann. Wenn die Formel hingegen mit einem Allquator beginnt, wenn sie also von der Form $\forall_{x_1}\exists_{x_2}\forall_{x_3}\cdots E(x_1,\ldots x_n)$ ist, dann gehört die Formel genau dann zu QBF, wenn jeder Zug des Allspielers ein Gewinnzug ist. Für die Formel

$$\forall x\exists y\ [\ (\neg x \vee y) \wedge (x \vee \neg y)\ ]$$

ist offenbar sowohl $x = 0$ wie auch $x = 1$ ein Gewinnzug, während die Formel

$$\exists y\forall x\ [\ (\neg x \vee y) \wedge (x \vee \neg y)\ ]$$

nicht zu QBF gehört: Der Exstenzspieler verliert sowohl für $y = 0$ wie auch für $y = 1$. Natürlich(?) ist QBF entscheidbar und dies gelingt sogar mit relativ geringem Speicherplatz.

Aufgabe 81
Sei α eine quantifizierte boolesche Formel, die aus n Symbolen besteht. (Wir zählen alle Quantoren, Variablen, aussagenlogische Operatoren und Klammern.) Entwirf eine deterministische Turingmaschine, die mit Speicherplatz $O(n)$ auskommt und entscheidet, ob α wahr ist oder nicht.

In der Vorlesung "Theoretische Informatik 2" beschäftigt man sich im Detail mit Problemen, die mit polynomiell großem Speicherplatz lösbar sind und nennt die Klasse all dieser Probleme PSPACE.

Definition 10.2 *Die Klasse* PSPACE *besteht aus allen Entscheidungsproblemen, die von deterministischen Turingmaschinem auf polynomiellem Speicherplatz gelöst werden können.*

Beachte, dass das Geographie-Spiel zu QBF gehört. In der „Theoretischen Informatik 2" wird gezeigt, dass QBF unter allen auf polynomiellen Platz berechenbaren Problemen ein „schwierigstes" Problem ist. Genauer: QBF ist PSPACE-vollständig unter polynomiellen Reduktionen.

Die Klasse PSPACE ist sehr mächtig und besitzt zum Beispiel NP als Teilmenge.

Satz 10.3 *Es ist* NP $\subseteq$ PSPACE*. Weiterhin sind alle Entscheidungsprobleme in* PSPACE *entscheidbar.*

Beweis: Sei M eine nichtdeterministische Turingmaschine, die in Zeit $q(n)$ rechnet. Dann wird M höchstens $2^{O(q(n))}$ verschiedene Berechnungen besitzen, wobei jede Berechnung höchstens Speicherplatz $q(n)$ benötigt: In Zeit $q(n)$ können ja höchstens $t(n)$ Speicherzellen aufgesucht werden. Eine deterministische Turingmaschine M' kann dann aber alle Berechnungen von M auf Speicherplatz $O(q(n))$ aufzählen und auswerten!

Da Probleme in PSPACE von deterministischen Turingmaschinen gelöst werden, ist jedes zu PSPACE gehörende Problem natürlich auch entscheidbar. □

Ist denn möglicherweise PSPACE die Klasse aller entscheidbaren Entscheidungsprobleme? Nein, nein und nochmals nein: Die Klasse entscheidbarer Entscheidungsprobleme ist riesig im Vergleich zu PSPACE. Die Klasse der entscheidbaren Probleme enthält natürlich nicht nur alle auf polynomiellem Speicherplatz lösbaren Sprachen, sondern auch alle auf exponentiellem, doppelt exponentiellem ... Speicherplatz lösbaren Probleme und ist in der Tat riesig im Vergleich zu PSPACE.

Betrachten wir zum Beispiel die Theorie $\mathrm{Th}_{\mathbb{R}}(\mathbf{0}, \mathbf{1}, +, -, *, /, \leq)$ der reellen Zahlen, also alle Formeln der Prädikatenlogik erster Stufe (mit den Konstanten **0** und **1** sowie den Operationen $+, -, *, /$ sowie der Relation $\leq$, die für die reellen Zahlen wahr sind. Es kann gezeigt werden, dass $\mathrm{Th}_{\mathbb{R}}(\mathbf{0}, \mathbf{1}, +, -, *, /, \leq)$ entscheidbar ist, aber nicht zur Klasse PSPACE gehört. Die Theorie der reellen Zahlen ist also komplex. Andererseits ist jede Formel ohne freie Variablen entweder beweisbar oder ihre Negation ist beweisbar. Man sagt auch, dass die Theorie der reellen Zahlen vollständig ist: So furchtbar schwer ist die Analysis also nicht.

Eine weiteres entscheidbares Problem, das nicht zu PSPACE gehört, ist zum Beispiel die Theorie $\mathrm{Th}_{\mathbb{N}}(\mathbf{0}, \mathbf{1}, +, -, \leq)$ aller Formeln (mit den Konstanten **0** und **1**, den Operationen $+, -$ und der Relation $\leq$), die für die natürlichen Zahlen wahr sind. Richtig gehen übel wird es wenn wir auch die Multiplikation zulassen. Wir erhalten dann die unentscheidbare „Zahlentheorie“, also alle in den natürlichen Zahlen wahre Formeln.

Was aber passiert, wenn wir probabilistische Berechnungen zulassen? Randomisierte Algorithmen spielen im Algorithmenentwurf eine wichtige Rolle —es sei zum Beispiel an den randomisierten Quicksort Algorithmus erinnert— und wir sollten deshalb fragen, ob die Church-Turing These auch gilt, wenn wir probabilistische Turingmaschines als „konkurrierende Rechenverfahren“ zulassen.

Eine probabilistische Turingmaschine M wird durch den Vektor

$$M = (Q, \Sigma, \delta, q_0, \Gamma, F) \quad \text{bzw.} \quad M = (Q, \Sigma, \delta, q_0, \Gamma)$$

beschrieben. Die Überführungsfunktion δ hat im Gegensatz zu deterministischen Turingmaschinen die Form

$$\delta : \Gamma \times Q \times \Gamma \times Q \times \{\texttt{links}, \texttt{bleib}, \texttt{rechts}\} \Rightarrow [0, 1] \cap \mathbb{Q}$$

und weist damit jedem möglichen Übergang

$$(\gamma, q) \Rightarrow (\gamma', q', \mathit{Richtung})$$

die Wahrscheinlichkeit

$$\delta(\gamma, q, \gamma', q', \mathit{Richtung})$$

zu. Wir verlangen, dass für jedes Paar $(\gamma, q) \in \Gamma \times Q$ eine Wahrscheinlichkeitsverteilung auf den Übergängen vorliegt. Das heißt, wir fordern für jedes Paar (γ, q) die Bedingung

$$\sum_{(\gamma', q', Richtung) \in \Gamma \times Q \times \{\texttt{links}, \texttt{bleib}, \texttt{rechts}\}} \delta(\gamma, q, \gamma', q', Richtung) = 1.$$

Wie arbeitet eine probabilistische Turingmaschine M? Für Eingabe x wird M, abhängig vom Ausgang der Münzwürfe, möglicherweise viele Berechnungen ausführen. Als Wahrscheinlichkeit einer Berechnung B bezeichnen wir das Produkt aller Übergangswahrscheinlichkeiten für die einzelnen Schritte von B, also

$$\text{prob}[\ B\ ] = \prod_{i=0}^{m-1} p_i$$

wenn p_i die Übergangswahrscheinlichkeit des $i$ten Schritts ist. Wie sollen wir das von einer probabilistischen Turingmaschine M gelöste Entscheidungsproblem definieren? Es liegt nahe, für jede Eingabe x die Wahrscheinlichkeit akzeptierender Berechnungen, also

$$p_x = \sum_{B \text{ ist akzeptierende Berechnung von } x} \text{prob}[\ B\ ]$$

zu messen.

Definition 10.4 *Sei M eine probabilistische Turingmaschine. Dann ist*

$$L_M = \left\{ x \in \Sigma^* | p_x > \frac{1}{2} \right\}$$

das von M gelöste Entscheidungsproblem. Wir sagen, dass M mit beschränkten Fehler ε rechnet, falls $p_x \leq \varepsilon$ für alle Eingaben $x \notin L_M$ und $p_x \geq 1 - \varepsilon$ für alle $x \in L_M$ gilt.

Probabilistische Turingmaschinen mit beschränktem Fehler $\varepsilon < 1/2$ sind ein sinnvolles und in der Praxis äußerst nützliches Maschinenmodell: Wenn wir die Maschine k-mal auf einer vorgegebenen Eingabe x laufen lassen und das Mehrheitsergebnis übernehmen, dann wird die Wahrscheinlichkeit eines Fehlers höchstens

$$2^{-\Omega(k)}$$

sein. (Warum ?)

Die Berechnungskraft probabilistischer Turingmaschinen mit beschränktem Fehler ist natürlich mindestens so groß wie die Berechnungskraft deterministischer Turingmaschinen. Es gibt allerdings starke Indizien, die vermuten lassen, dass die in polynomieller Zeit mit beschränktem Fehler berechenbare Problemklasse sogar in deterministischer, polynomieller Zeit berechenbar ist. Wir zeigen jetzt, dass selbst die weitaus mächtigeren Berechnungen mit unbeschränktem Fehler, wenn also der Fehler ε mit wachsender Eingabelänge gegen $1/2$ konvergieren darf, durch deterministische Turingmaschinen simulierbar sind.

Satz 10.5 *Sei M eine probabilistische Turingmaschine (mit nicht notwendigerweise beschränktem Fehler). Wenn die worst-case Laufzeit einer jeden Berechnung für Eingaben der Länge n durch $t(n)$ beschränkt ist, dann gibt es eine deterministische Turingmaschine M', die L_M löst und dazu höchstens Speicherplatz*

$$O(t(n))$$

benötigt. Es muss allerdings gefordert werden, dass die Unärdarstellung von $t(n)$ mit Speicherplatz $O(t(n))$ berechnet werden kann.

Beweisskizze : Die zu konstruierende deterministische Turingmaschine M' wird zuerst $t(n)$ Zellen „abstecken“. Dann werden nacheinander alle höchstens $2^{O(t(n))}$ Berechnungen für eine vorgegebene Eingabe x simuliert. Ein Zähler summiert die Wahrscheinlichkeiten akzeptierender Berechnungen (auf $O(t(n))$ Zellen). Nachdem alle Berechnungen simuliert sind, wird geprüft, ob der Zähler einen Wert größer $\frac{1}{2}$ hat und in diesem Fall wird akzeptiert. ■

Bemerkung (a): Randomisierte Algorithmen führen für viele wichtige Fragestellungen zu den schnellsten bekannten algorithmischen Lösungen, scheinen aber keine exponentielle Beschleunigung gegenüber deterministischen Algorithmen zu bringen. Die Berechnungskraft gegenüber deterministischen Algorithmen nimmt allerdings stark zu, wenn Online-Algorithmen betrachten werden: Ein Online-Algorithmus[1] muss Entscheidungen treffen ohne die Eingabe vollständig zu kennen. Hier hilft es, „gegen die Zukunft zu randomisieren“. Diese und ähnliche Themen werden in der Bachelor-Vorlesung „Effiziente Algorithmen“ behandelt.

(b) Auch das Ergebnis von Satz 10.5 weist auf die Wichtigkeit der Komplexitätsklasse `PSPACE` hin, denn alle Entscheidungsprobleme, die sich in polynomieller Zeit mit unbeschränktem Fehler $\varepsilon < 1/2$ berechnen lassen, gehören zu `PSPACE`.

(c) Wie groß kann die Laufzeit einer Berechnung mit polynomiellen Speicherplatz werden? Wenn eine deterministische Turingmaschine auf $s(n)$ Zellen rechnet, dann wird sie nach höchstens

$$|\Gamma|^{s(n)} \cdot |Q| \cdot s(n)$$

vielen Schritten in eine Schleife eintreten. Diese Schranke ergibt sich aus der Tatsache, dass der aktuelle Bandinhalt ($|\Gamma|^{s(n)}$ Möglichkeiten), der aktuelle Zustand ($|Q|$ Möglichkeiten) und die aktuelle Position des Lese/Schreibkopfs ($s(n)$ Möglichkeiten) eine Konfiguration eindeutig festlegen. Mit anderen Worten, die Laufzeit einer solchen Turingmaschine ist durch

$$2^{O(s(n))}$$

beschränkt. Also ist die Laufzeit durch $2^{O(s(n))}$ beschränkt.

Wir kommen als nächstes zu einer allerdings nur recht oberflächlichen Beschreibung von Quantenrechnern. Zu Anfang erinnern wir an das Rechnen mit komplexen Zahlen. $\mathbb{C} = \{x + iy \mid x, y \in \mathbb{R}\}$ bezeichnet die Menge der komplexen Zahlen und es ist $i = \sqrt{-1}$. Für die komplexe Zahl $z = x + iy$ ist $\bar{z} = x - iy$ die Konjugierte von z. Die Länge von z ist durch

$$|z| = \sqrt{x^2 + y^2}$$

definiert und für komplexe Zahlen $z_1, z_2 \in \mathbb{C}$ mit $z_k = x_k + iy_k$ ist

$$\begin{aligned} z_1 + z_2 &= x_1 + x_2 + i(y_1 + y_2) \\ z_1 \cdot z_2 &= x_1 \cdot x_2 - y_1 \cdot y_2 + i(x_1 \cdot y_2 + x_2 \cdot y_1). \end{aligned}$$

Die Grobstruktur eines Quantenrechners ähnelt der einer probabilistischer Turingmaschine. Diesmal hat aber die Überführungsfunktion δ die Form

$$\delta \ : \ \Gamma \times Q \times \Gamma \times Q \times \{\texttt{links}, \texttt{bleib}, \texttt{rechts}\} \to \mathbb{C}$$

wobei nur komplexe Zahlen der Länge höchstens 1 zugewiesen werden. Wie im Fall probabilistischer Turingmaschinen gibt es zu jedem Paar $(\gamma, q) \in \Gamma \times Q$ potentiell viele Übergänge, wobei diesmal

$$\sum_{\gamma', q', Richtung} |\delta(\gamma, q, \gamma', q', Richtung|^2 = 1$$

[1] Finanztransaktionen sind ein typisches Beispiel für Online Algorithmen: Anlageentscheidungen sind ohne Kenntnis der zukünftigen Entwicklung zu tätigen.

gelten muss. Wir sagen, dass

$$\delta(\gamma, q, \gamma', q', Richtung)$$

die (Wahrscheinlichkeits-)Amplitude ist und, dass

$$|\delta(\gamma, q, \gamma', q', Richtung)|^2$$

die zugewiesene Wahrscheinlichkeit ist. Bisher haben wir nur eine merkwürdige Darstellung der Wahrscheinlichkeit eines Übergangs kennengelernt, der wesentliche Unterschied zu den probabilistischen Turingmaschinen folgt aber sofort: Wir betrachten zu einem beliebigen aber festen Zeitpunkt alle Berechnungen und weisen jeder Berechnung B das Produkt p_B der ihren Übergängen entsprechenden Wahrscheinlichkeitsamplituden zu. Charakteristischerweise werden wir im Allgemeinen aber viele Berechnungen haben, die in derselben Konfiguration C enden. Wir weisen der Konfiguration C die Wahrscheinlichkeitsamplitude

$$\tau_C = \sum_{B \text{ führt auf } C} p_B$$

zu und definieren

$$|\tau_C|^2$$

als die Wahrscheinlichkeit der Konfiguration C. Das von einem Quantenrechner Q gelöste Entscheidungsproblem definieren wir dann als

$$L_Q = \left\{ x \middle| \sum_{C \text{ ist akzeptierende Konfiguration von } Q \text{ auf Eingabe } x} |\tau_C|^2 > \frac{1}{2}, \right\}$$

analog zu probabilistischen Turingmaschinen. Auch hier ist es notwendig Quantenrechner mit beschränktem Fehler einzuführen, um dem gesehenen Ergebnis, notfalls nach genügend häufigem Wiederholen, trauen zu können.

Unsere Beschreibung ist zu diesem Zeitpunkt unvollständig: das beschriebene Rechnermodell ist weitaus mächtiger als das Modell der Quantenrechner. Deshalb noch einmal ein Ausflug in die komplexe Zahlen. Für eine Matrix $A = (z_{i,j})_{1 \leq i,j \leq n}$ mit komplexwertigen Einträgen ist

$$\overline{A} = (\overline{z_{j,i}})_{1 \leq i,j \leq n}$$

die konjugiert Transponierte von A. Wir nennen A *unitär*, falls

$$\overline{A} \cdot A = \text{Einheitsmatrix}$$

gilt. Wir halten jetzt in der Konfigurationsmatrix A_Q die Wahrscheinlichkeitsamplituden eines 1-Schritt Übergangs zwischen je zwei Konfigurationen C und C' fest. Also

$$A_Q[C, C'] = \text{Wahrscheinlichkeitsamplitude des Übergangs von } C' \text{ nach } C.$$

Ein Quantenrechner liegt vor, falls die Matrix A_Q unitär ist.

Satz 10.6 *Wenn ein Quantenrechner Q in Zeit $t(n)$ rechnet, dann gibt es eine deterministische Turingmaschine M, die auf Platz $O(t^2(n))$ das Entscheidungsproblem L_Q löst. Hierzu ist die Forderung eines beschränkten Fehlers ebenso nicht notwendig wie die Forderung, dass die Konfigurationsmatrix A_Q unitär ist.*

Beweisskizze : Die simulierende deterministische Turingmaschine wird die Einträge des Matrix/Vektor-Produkts

$$A_Q^{t(n)} \cdot v$$

nacheinander berechnen und die Wahrscheinlichkeiten akzeptierender Konfigurationen aufsummieren. Es wird akzeptiert, falls die Summe größer als 1/2 ist. Warum funktioniert dieser Ansatz, wenn wir den Vektor v durch

$$v_i = \begin{cases} 0 & i \neq \text{Startkonfiguration} \\ 1 & \textit{sonst} \end{cases}$$

definieren? Der Vektor $A_Q \cdot v$ gibt die Wahrscheinlichkeitsamplituden der 1-Schritt Nachfolger der Startkonfiguration wieder und allgemeiner listet der Vektor $A_Q^k \cdot v$ die Wahrscheinlichkeitsamplituden der k-Schritt Nachfolger auf. Im letzten Schritt (also $k = t(n)$) müssen wir dann nur noch von den Wahrscheinlichkeitsamplituden zu den Wahrscheinlichkeiten übergehen. Wie berechnet man aber $A_Q^{t(n)} \cdot v$ in Platz $O(t^2(n))$? Die Matrix A_Q besitzt ja $2^{O(t(n))}$ Zeilen und Spalten! Hier ist ein Tipp: Der Vektor $A_Q^k \cdot v$ kann in Platz $O(k \cdot t(n))$ berechnet werden. ■

Unsere Beschreibung von Quantenrechnern ist mathematisch und greift nicht auf die motivierenden Hintergründe der Quantenmechanik zurück. Wir verweisen deshalb zum Beispiel auf das Textbuch von *Mika Hirvensalo* über Quantum Computing.

Die Möglichkeiten von Quantenrechnern sind enorm. So hat Peter Shor (*Algorithms for Quantum Computing : Discrete Logarithms and Factoring, Procedings of the 35th Annual Symposium on Foundations of Computer Science, pp.124-134, 1994*) zeigen können, dass natürliche Zahlen in polynomieller Zeit faktorisiert werden können. Dies impliziert, dass die meisten „public-key“ Verschlüsselungstechniken gebrochen werden können! Allerdings ist zu dieser Zeit unklar, ob und wenn ja, wann Quantenrechner tatsächlich realisiert werden können.

Fassen wir zusammen: Nichtdeterministische und probabilistische Turingmaschinen wie auch Quantenrechner sind zwei weitere Rechnermodelle, die die Church-Turing These unterstützen. Während probabilistische Turingmaschinen durch pseudo-Zufallsgeneratoren möglicherweise mit nur polynomieller Verlangsamung simulierbar sind, so scheinen Quantenrechner mächtiger. Wir haben aber gezeigt, dass jede von einem Quantenrechner in polynomieller Zeit berechnete Funktion auch von einer deterministischen Turingmaschine auf polynomiellen Platz und damit in exponentieller Zeit berechenbar ist[2].

Aufgabe 82
Finde heraus, ob die wie folgt definierte Funktion $h : \{1\}^* \to \{0,1\}$ berechenbar ist: $h(x) = 1 \Leftrightarrow$ [In der Dezimaldarstellung der Zahl π gibt es eine Stelle, wo die Ziffer 7 mindestens $|x|$ mal hintereinander vorkommt.] **Begründe** deine Antwort.

(Zur Erinnerung $\pi = 3,141592653589793.....$)

Hinweis: Überlege, welches Aussehen die Funktion h haben kann.

---

[2]Eine effiziente Berechnung NP-vollständiger Probleme durch Quantenrechner erscheint aber höchst unwahrscheinlich.

Kapitel 11

Unentscheidbare Probleme

Das Ziel dieses Kapitels ist die Untersuchung *unentscheidbarer* Probleme, also die Untersuchung von Problemen, die von keiner stets haltenden Turingmaschine gelöst werden können. Warum ist die Existenz unentscheidbarer Probleme alles andere als überraschend? Jede Turingmaschine hat ein endliches Programm, das wir eindeutig durch eine natürliche Zahl kodieren können (für Details siehe weiter unten). Es gibt also nur abzählbar unendlich viele Turingmaschinen.

Aber wieviele Entscheidungsprobleme gibt es (zum Beispiel über dem Alphabet $\{0,1\}$)? Um ein Entscheidungsproblem zu spezifizieren, müssen wir für jedes Wort $w \in \{0,1\}^*$ angeben, ob w zum Entscheidungsproblem gehört. Zu dieser Aufgabe benötigen wir offensichtlich nur ein Bit $\text{bit}(w) \in \{0,1\}$. Den Vektor

$$(\text{bit}(w) \mid w \in \{0,1\}^*)$$

können wir aber als Binärdarstellung einer reellen Zahl im Intervall $[0,1]$ auffassen. Andererseits können wir auch jede reelle Zahl $r \in [0,1]$ als die Beschreibung eines Entscheidungsproblems L_r auffassen. (Wie?) Also stimmt die Kardinalität aller Entscheidungsprobleme über $\{0,1\}$ mit der Kardinalität des reellen Intervalls $[0,1]$ überein. Es gibt also wesentlich mehr Entscheidungsprobleme als Turingmaschinen, und sogar die „meisten“ Entscheidungsprobleme sind unentscheidbar.

Dieser Existenzbeweis läßt aber die Frage offen, wie unentscheidbare Probleme „aussehen“. Bevor wir diese Frage angehen, zeigen wir, dass Turingmaschinen *programmierbar* sind; das heißt, wir werden eine Turingmaschine U entwerfen, die ein (geeignet kodiertes) Turingmaschinenprogramm P und eine Eingabe w für P als Eingabe annimmt und sodann das Programm P auf Eingabe w simuliert. Eine solche Turingmaschine wird auch *universelle Turingmaschine* genannt.

11.1 Universelle Turingmaschinen

Wir schränken zuerst die Turingmaschinenprogramme syntaktisch ein.

Aufgabe 83
Der Akzeptor $M = (Q, \Sigma, \delta, q_0, \Gamma, F)$ sei eine beliebige Turingmaschine. Wir sagen, dass die Turingmaschinen M und $M' = (Q', \Sigma, \delta', q_0', \Gamma', F')$ genau dann äquivalent sind, wenn beide Maschinen dasselbe Entscheidungsproblem lösen (also $L(M) = L(M')$) und wenn beide Maschinen für dieselbe Eingabemenge halten (für alle $w \in \Sigma^*$ muss also gelten: M' hält auf Eingabe $w \Leftrightarrow M$ hält auf Eingabe w.)

(a) Beschreibe eine zu M äquivalente Turingmaschine M' mit $Q' = \{0, 1, \ldots, |Q| - 1\}$ und $q'_0 = 0$. (Zuerst normieren wir also die Zustandsmenge und den Anfangszustand.) Aus der Beschreibung sollte insbesondere hervorgehen, wie δ' aus δ erhalten wird.

(b) Beschreibe eine zu M äquivalente Turingmaschine M' mit $Q' = \{0, 1, \ldots, |Q|\}$, $q'_0 = 0$ **und** $F' = \{1\}$. (Wir fordern also jetzt zusätzlich, dass M' genau einen akzeptierenden Zustand, den Zustand 1, besitzt. Diesmal benötigen wir aber möglicherweise einen zusätzlichen Zustand. Warum?)

(c) Beschreibe, für eine geeignete Zahl m, eine zu M äquivalente Turingmaschine M' mit $Q' = \{0, 1, \ldots, m\}, q'_0 = 0, F' = \{1\}$ **und** $\Gamma' = \{0, 1, B\}$.

Hinweis: Die Zahl m wird von der Turingmaschine M abhängen. Weiterhin, um die Maschine M simulieren zu können, müssen wir versuchen, die Buchstaben des Bandalphabets Γ durch binäre Worte zu kodieren.

Eine umgangssprachliche Beschreibung der Simulation ist ausreichend.

Wir erlauben nur das Eingabealphabet $\Sigma = \{0, 1\}$. Nach der obigen Aufgabe können wir also beim Entwurf der universellen Turingmaschine U annehmen, dass eine Turingmaschine M mit Bandalphabet $\Gamma = \{0, 1, \mathsf{B}\}$ auf einer Eingabe $w \in \{0, 1\}^*$ zu simulieren ist. Wir können weiterhin annehmen, dass M einen einzigen akzeptierenden Zustand besitzt, nämlich den Zustand 1 und dass $Q = \{0, 1, \ldots, |Q| - 1\}$ die Zustandsmenge von Q mit Anfangszustand 0 ist. Es bleibt zu klären, wie wir die Turingmaschine

$$M = (Q, \Sigma, \delta, q_0, \Gamma, F)$$

als binäres Wort kodieren. Σ, q_0, Γ und F sind a priori bekannt, und es genügt, $|Q|$ und δ zu kodieren. Wir definieren

$$\langle M \rangle := 1^{|Q|}\ 0\ \mathrm{code}(\delta)\ 00$$

als die Kodierung von M oder die *Gödelnummer* von M. Die Kodierung $\mathrm{code}(\delta)$ der Zustandsüberführungsfunktion δ erfolgt durch Konkatenation der Kodierungen der einzelnen Befehle, wobei der Befehl

$$\delta(q, a) = (q', b, wohin)$$

durch das Wort

$$1^{q+1}\ 0\ 1^{\mathrm{Zahl}(a)}\ 0\ 1^{q'+1}\ 0\ 1^{\mathrm{Zahl}(b)}\ 0\ 1^{\mathrm{Zahl}(wohin)}\ 0$$

kodiert wird. (Es ist

$$\mathrm{Zahl}(0) = 1, \quad \mathrm{Zahl}(1) = 2, \quad \mathrm{Zahl}(\mathsf{B}) = 3,$$
$$\mathrm{Zahl}(\mathsf{links}) = 1, \quad \mathrm{Zahl}(\mathsf{rechts}) = 2, \quad \mathrm{Zahl}(\mathsf{bleib}) = 3.)$$

Wir müssen jetzt U so programmieren, dass für Eingabe

$$\langle M \rangle\, w$$

die Maschine M auf Eingabe w simuliert wird. Wie? Es genügt gemäß Satz 5.2 eine 3-Band-Maschine zu entwerfen. Zuerst wird die 3-Band-Maschine die Gödelnummer $\langle M \rangle$ vom Eingabeband löschen und auf das zweite Band schreiben. Dann wird überprüft, ob $\langle M \rangle$ syntaktisch korrekt ist, d.h. ob $\langle M \rangle$ einer Gödelnummer entspricht. Ist dies der Fall, wird der Startzustand 0 auf Band 3 geschrieben, und die schrittweise Simulation wird durchgeführt, wobei jedesmal Band 3 den gegenwärtigen Zustand speichert.

Lemma 11.1 *Die universelle Turingmaschine U hat die folgenden Eigenschaften:*

(a) M akzeptiert die Eingabe w genau dann, wenn U die Eingabe $\langle M \rangle\, w$ akzeptiert .

(b) M hält genau dann auf Eingabe w, wenn U auf Eingabe $\langle M \rangle\, w$ hält.

Wir kommen jetzt zum ersten Höhepunkt dieses Abschnitts, nämlich der Unentscheidbarkeit der Diagonalsprache D.

11.2 Diagonalisierung

Definition 11.2 $D = \{\langle M\rangle \mid M$ *akzeptiert die Eingabe* $\langle M\rangle$ nicht$\}$ *heißt Diagonalsprache.*

Satz 11.3 *D ist nicht entscheidbar.*

Beweis: Angenommen, D ist entscheidbar. Dann gibt es eine stets haltende Turingmaschine M^*, so dass für *alle* Turingmaschinen M gilt:

$$\begin{aligned} M^* \text{ akzeptiert } \langle M\rangle &\Leftrightarrow \langle M\rangle \in D \\ &\Leftrightarrow \quad M \text{ akzeptiert } \langle M\rangle \text{ nicht.} \end{aligned}$$

Diese Äquivalenz gilt für alle Turingmaschinen und somit auch für die Turingmaschine $M = M^*$. Damit folgt aber die sinnlose Aussage

$$M^* \text{ akzeptiert } \langle M^*\rangle \Leftrightarrow M^* \text{ akzeptiert } \langle M^*\rangle \text{ nicht,}$$

und wir haben somit einen Widerspruch zur Annahme der Entscheidbarkeit erreicht. ■

Bemerkung 11.1 Warum heisst D die Diagonalsprache? Wir bilden eine unendliche Matrix T und ordnen einer Zeile (bzw. einer Spalte) eine Turingmaschine M (bzw. eine Gödelnummer $< N >$) zu. Für Zeile M und Spalte $< N >$ definieren wir

$$T(M, < N >) = \begin{cases} 1 & M \text{ akzeptiert } < N > \\ 0 & \text{sonst.} \end{cases}$$

Die Matrix T beschreibt also das Verhalten sämtlicher Turingmaschinen auf Gödelnummern. Die Diagonalsprache D „flippt“ die Diagonale von T –D diagonalisiert– und erzwingt damit, dass D von keiner Turingmaschine M^* entscheidbar ist: M^* wird auf Eingabe $< M^* >$ die Anwort $T(M^*, < M^* >)$ geben, während D die geflippte Antwort verlangt.

Das Diagonalisierungsargument wurde von Georg Cantor zum ersten Mal angewandt, um zu zeigen, dass die reellen Zahlen überabzählbar große Mächtigkeit besitzen.

Aufgabe 84
Eine Menge X heißt *abzählbar*, falls jedem $x \in X$ eine Zahl $i(x) \in \mathbb{N}$ zugewiesen werden kann, so dass $i(x) \neq i(y)$ für alle $x \neq y \in X$.

Sei X die Menge aller 0-1 Folgen, also $X = \{x \mid x : \mathbb{N} \to \{0, 1\}\}$.

Zeige, dass X nicht abzählbar ist.

Hinweis: Wende das Diagonalisierungsargument an.

Leider ist die Diagonalsprache ein „künstliches“ Entscheidungsproblem: ihre Unentscheidbarkeit scheint keine Konsequenzen für „real world“-Probleme zu haben. Tatsächlich ist das Gegenteil der Fall, wie wir im nächsten Abschnitt sehen werden.

11.3 Reduktionen

Wir führen den Reduktionsbegriff ohne polynomielle Zeitbeschränkung ein, um weitere unentscheidbare Entscheidungsprobleme zu erhalten.

Definition 11.4 *L_1 und L_2 seien zwei Entscheidungsprobleme über Σ_1 beziehungsweise über Σ_2. Wir sagen, dass L_1 genau dann auf L_2 reduzierbar ist ($L_1 \leq L_2$), wenn es eine stets haltende Turingmaschine T gibt, so dass für alle $w \in \Sigma_1^*$ gilt:*

$$w \in L_1 \;\Leftrightarrow\; T(w) \in L_2.$$

($T(w) \in \Sigma_2^*$ bezeichnet die Ausgabe von T für Eingabe w. T heißt die transformierende *Turingmaschine.)*

Satz 11.5 *L sei ein unentscheidbares Problem.*

(a) Dann ist auch $\overline{L}$, das Komplement von L, unentscheidbar.

(b) Es gelte $L \leq K$. Dann ist auch K unentscheidbar.

Beweis:

(a) Wenn die stets haltende Turingmaschine M das Problem $\overline{L}$ löst, dann können wir eine stets haltende Turingmaschine M' bauen, die L löst: Wir wählen $Q \setminus F$ als neue Menge akzeptierender Zustände.

(b) Angenommen, es gibt eine stets haltende Turingmaschine M, die K löst. Da $L \leq K$, gibt es eine stets haltende Turingmaschine T mit

$$w \in L \;\Leftrightarrow\; T(w) \in K.$$

Also gilt $w \in L \;\Leftrightarrow\; M$ akzeptiert die Eingabe $T(w)$.

Damit wird L aber durch eine stets haltende Turingmaschine (nämlich die Maschine, die zuerst T und dann M simuliert) gelöst. Das verstößt jedoch gegen die Unentscheidbarkeit des Problems L. ■

Satz 11.5 liefert uns die gewünschten Methoden, um neue Unentscheidbarkeitsergebnisse zu erhalten. Wir sind insbesondere an den folgenden Problemen interessiert:

Definition 11.6

(a) $H = \{\langle M\rangle\, w \mid M$ hält auf Eingabe $w\}$ ..... das Halteproblem.

(b) $H_\varepsilon = \{\langle M\rangle \mid M$ hält auf dem leeren Wort $\varepsilon\}$ ..... das spezielle Halteproblem.

(c) $U = \{\langle M\rangle\, w \mid M$ akzeptiert $w\}$ ..... die universelle Sprache.

Satz 11.7 *Die universelle Sprache U ist nicht entscheidbar.*

Es kann somit keinen Super-Compiler geben, der stets korrekt voraussagt, ob ein Programm eine vorgelegte Eingabe akzeptiert, es sei denn, wir erlauben, dass der Super-Compiler selbst „manchmal“ nicht hält.

Beweis: Wir betrachten das Komplement $\overline{D}$ der Diagonalsprache, also

$$\overline{D} = \Big\{ w \in \{0,1\}^* \quad | \quad (w = \langle M \rangle \text{ und } M \text{ akzeptiert Eingabe } \langle M \rangle) \\ \text{oder } (w \text{ ist keine Gödelnummer}) \Big\}.$$

Nach Satz 11.5 (a) ist $\overline{D}$ nicht entscheidbar. Mit Satz 11.5 (b) genügt es, die Reduktion

$$\overline{D} \leq U$$

nachzuweisen. Die transformierende Turingmaschine T prüft zuerst, ob w die Gödelnummer einer Turingmaschine M ist. Ist dies der Fall, dann berechnet T die Ausgabe $\langle M \rangle w$, also die Ausgabe $\langle M \rangle \langle M \rangle$. Ist dies nicht der Fall, dann berechnet T die Ausgabe $\langle M_0 \rangle \varepsilon$, wobei die Turingmaschine M_0 auf jeder Eingabe sofort hält und diese akzeptiert. Offensichtlich hält T für jede Eingabe.

Wir müssen nachweisen, dass

$$w \in \overline{D} \Leftrightarrow T(w) \in U$$

gilt.

Fall 1: w ist keine Gödelnummer.

Dann ist $w \in \overline{D}$. Andererseits ist $T(w) = \langle M_0 \rangle \varepsilon$, wobei M_0 jedes Wort, und damit auch das leere Wort akzeptiert. Es ist also $\langle M_0 \rangle \varepsilon \in U$.

Fall 2: $w = \langle M \rangle$.

Dann ist $T(w) = \langle M \rangle w$ und

$$\begin{aligned} \langle M \rangle \in \overline{D} &\Leftrightarrow M \text{ akzeptiert } \langle M \rangle \\ &\Leftrightarrow \langle M \rangle w \in U \\ &\Leftrightarrow T(w) \in U. \end{aligned} \tag{11.1}$$

■

Was haben wir ausgenutzt? Um das Komplement der Diagonalsprache auf die universelle Sprache zu reduzieren, müssen wir mit Hilfe der universellen Sprache entscheiden, wann die Gödelnummer $\langle M \rangle$ im Komplement von D liegt, d.h. wann die Turingmaschine M auf ihrer eigenen Gödelnummer hält und akzeptiert. Nichts leichter als das, denn M akzeptiert ihre eigene Gödelnummer $\langle M \rangle$ genau dann, wenn $\langle M \rangle \langle M \rangle \in U$. Genau das haben wir in (11.1) ausgenutzt.

Wir zeigen mit denselben Methoden, dass auch das Halteproblem unentscheidbar ist. Wir haben die Option, zu versuchen entweder D, das Komplement $\overline{D}$ oder die universelle Sprache auf H zu reduzieren. Natürlich werden wir die Reduktion $U \leq H$ versuchen, denn die beiden Probleme sind sich sehr ähnlich: U „redet“ vom Akzeptieren und H vom Halten.

Satz 11.8 *Das Halteproblem H ist nicht entscheidbar.*

Beweis: Es genügt zu zeigen, dass die universelle Sprache auf das Halteproblem reduzierbar ist, dass also $U \leq H$ gilt.

Sei die Eingabe $v \in \{0,1\}^*$ gegeben. Wie muss die transformierte Eingabe $T(v)$ aussehen?

Fall 1: Wenn v nicht die Form $v = \langle M \rangle w$ hat, dann setzen wir $T(v) = v$.

Fall 2: $v = \langle M \rangle w$

Die transformierende Turingmaschine T muss die Maschine M durch eine Maschine M' ersetzen, so dass

$$M \text{ akzeptiert } w \;\Leftrightarrow\; M' \text{ hält auf } w \tag{11.2}$$

gilt. Wie bekommen wir das hin? Sei q^* ein neuer Zustand. Wir ersetzen jeden Befehl $\delta(q,a) = (q,a,\mathsf{bleib})$ von M durch den Befehl $\delta(q,a) = (q^*,a,\mathsf{bleib})$, *falls* q *kein* akzeptierender Zustand ist. Sodann fügen wir die Befehle

$$\begin{aligned} \delta(q^*,0) &= (q^*,0,\mathsf{rechts}) \\ \delta(q^*,1) &= (q^*,1,\mathsf{rechts}) \\ \delta(q^*,\mathsf{B}) &= (q^*,\mathsf{B},\mathsf{rechts}) \end{aligned}$$

hinzu. Die neue Maschine M' wird *entweder* ihre Eingabe w akzeptieren (nämlich genau dann, wenn M die Eingabe w akzeptiert) oder aber nicht halten. Es gilt also

$$M \text{ akzeptiert } w \quad\Leftrightarrow\quad M' \text{ hält auf Eingabe } w. \tag{11.3}$$

Wir setzen deshalb, für $v = \langle M \rangle w$,

$$T(v) = \langle M' \rangle w$$

und müssen

$$v \in U \;\Leftrightarrow\; T(v) \in H$$

nachweisen.

Fall 1: v hat *nicht* die Form $v = \langle M \rangle w$.

Dann ist $v \notin U$ und $v \notin H$. Da aber $T(v) = v$, gilt $v \notin U$ und $T(v) \notin H$.

Fall 2: $v = \langle M \rangle w$. Es gilt

$$\begin{aligned} v \in U &\Leftrightarrow M \text{ akzeptiert } w \\ &\Leftrightarrow M' \text{ hält auf Eingabe } w \quad \text{(mit 11.3)} \\ &\Leftrightarrow T(v) \in H. \end{aligned}$$

■

Die Reduktion „wirft“ eine Eingabe $\langle M \rangle w$ auf die Eingabe $\langle M \rangle' w$, wobei die neue Maschine M' die Eigenschaft 11.2 erfüllen muss. Wie haben wir das hingekriegt? M' simuliert M fast immer Schritt für Schritt. Wenn M akzeptiert, dann akzeptiert auch M' und hält natürlich. Wenn M nicht hält, dann hält auch M' nicht und wir sind fein raus[1]. Gefährlich ist nur der Fall, dass M hält und verwirft. In diesem Fall ist $\langle M \rangle w \notin U$ und wir müssen $\langle M \rangle' w \notin H$ erreichen. Wie? Ganz einfach: Wenn M' merkt, dass M hält und verwirft, dann macht sich M' auf die “unendliche Reise nach rechts.“

[1] Achtung, ganz wichtig: Die Maschine M' muss nicht überprüfen, ob M hält. Das bekommt M' nicht hin, denn das Halteproblem ist ja unentscheidbar. Stattdessen simuliert M' stupide jeden Schritt von M, denn es weiss, wenn M nicht hält, dann ist erstens $\langle M \rangle w \notin U$, zweitens wird M' auch nicht halten und wie gewünscht gilt $\langle M \rangle' w \notin H$.

Satz 11.9 *Das spezielle Halteproblem H_ε ist nicht entscheidbar.*

Beweis: Wir zeigen, dass das Halteproblem H auf H_ε reduzierbar ist, dass also $H \leq H_\varepsilon$ gilt. Wir müssen zuerst die transformierende Turingmaschine T beschreiben.

Sei $v \in \{0,1\}^*$ eine Eingabe. Wenn v nicht die Form $v = \langle M \rangle\, w$ hat, dann setze

$$T(v) = 1$$

(Damit ist $T(v)$ keine Gödelnummer, und es ist $v \notin H$ und $T(v) \notin H_\varepsilon$).

Wenn $v = \langle M \rangle\, w$, dann baut T eine Turingmaschine M', die für jede Eingabe

- zuerst das Wort w auf das Band schreibt,
- den Kopf auf den ersten Buchstaben von w positioniert und sodann
- M simuliert.

Offensichtlich gilt

$$\begin{aligned} v \in H &\Leftrightarrow M \text{ hält auf } w \\ &\Leftrightarrow M' \text{ hält auf dem leeren Wort} \\ &\Leftrightarrow \langle M' \rangle \in H_\varepsilon \end{aligned}$$

Wir setzen $T(v) = \langle M' \rangle$ und haben somit

$$v \in H \;\Leftrightarrow\; T(v) \in H_\varepsilon$$

nachgewiesen. Da T für jede Eingabe hält, folgt die Behauptung des Satzes. ■

Warum hat diese Reduktion funktioniert? Das spezielle Halteproblem ist doch anscheinend viel einfacher als das uneingeschränkte Halteproblem?

Aufgabe 85
Welche der folgenden Entscheidungsprobleme sind entscheidbar, welche sind unentscheidbar?

Begründe Deine Antwort durch die Angabe eines stets haltenden Algorithmus oder durch die Angabe einer Reduktion.

(a) $G = (\Sigma, V, S, P)$ sei eine kontextsensitive Grammatik, d.h. die Produktionen in P sind von der Form $\alpha \to \beta$ mit $\alpha, \beta \in (\Sigma \cup V)^*$ und $|\alpha| \leq |\beta|$. Zu betrachten ist die Sprache $L_G = \{w \in \Sigma^* \mid w \in L(G)\}$.

(b) Eine quantifizierte KNF-Formel α hat die Form

$$\alpha = \forall x_1 \exists x_2 \forall x_3 \exists x_4 \cdots Q x_n \beta(x_1, x_2, x_3, x_4, \ldots, x_n)$$

wobei $Q \in \{\forall, \exists\}$, β eine aussagenlogische Formel in konjunktiver Normalform ist und jede Variable von β durch einen Quantor gebunden wird.
Zu betrachten ist das Entscheidungsproblem

$$Q\text{-}KNF = \left\{ \alpha \,\middle|\, \alpha \text{ ist eine quantifizierte KNF-Formel und } \alpha \text{ ist wahr} \right\}.$$

(c) $\left\{ (\langle M \rangle, \langle A \rangle) \,\middle|\, \begin{array}{l} M \text{ ist eine deterministische Turingmaschine, } A \text{ ist ein DFA} \\ \text{und } L(M) = L(A) \end{array} \right\}.$

(d) $\left\{ (\langle M \rangle, w) \,\middle|\, \begin{array}{l} M \text{ ist eine deterministische Turingmaschine} \\ \text{und } M \text{ wird für Eingabe } w \text{ den Kopf stets nach rechts bewegen} \end{array} \right\}.$

(e) $\left\{ (\langle M \rangle, q) \,\middle|\, \begin{array}{l} M \text{ ist eine deterministische Turingmaschine, } q \text{ ist ein Zustand von } M \\ \text{und } M \text{ erreicht } q, \text{ wenn } M \text{ auf dem leeren Wort startet} \end{array} \right\}.$

Aufgabe 86
Es sei

$$L_{\#} = \{ \langle M \rangle \mid M \text{ schreibt für Eingabe } \varepsilon \text{ irgendwann das Zeichen } \# \text{ auf das Band.}\}$$

Zeige, dass $L_{\#}$ nicht entscheidbar ist.

Aufgabe 87
Es sei $L_{non-blank}$ das folgende Problem

$$L_{non-blank} = \{< M > | M \text{ schreibt für Eingabe } \varepsilon \text{ irgendwann ein Nicht-Blank aufs Band}\}$$

Zeige, dass $L_{non-blank}$ entscheidbar ist.

Nun führen wir einen Beweis vor, der zeigt, dass $L_{non-blank}$ doch nicht entscheidbar ist. Es sei $< M > \in L_{non-blank}$ gegeben, konstruiere $< M_1 > \in L_{\#}$ wie folgt:

1. M_1 hat das gleiche Band-Alphabet wie M und zusätzlich das Zeichen #.
2. M_1 hat die gleiche Zustandsmenge wie M.
3. Die Überführungsfunktion von M_1 ist die von M mit der Ausnahme, dass jede Regel, bei der ein Nicht-Blank geschrieben wird, durch die Regel: (schreibe # und halte) ersetzt wird.

Damit ist bewiesen, dass M gestartet auf dem leeren Band genau dann ein Nicht-Blank schreibt, wenn M_1 gestartet auf dem leeren Band das Zeichen # schreibt. Nach dem Reduktionsprinzip folgt, dass $L_{non-blank}$ nicht entscheidbar ist.

Der Beweis ist falsch. **Finde** das fehlerhafte Argument und **begründe** deine Antwort.

Aufgabe 88
Das *Tiling Problem* ist wie folgt definiert. Für eine vorgegebene endliche Menge von Kacheltypen müssen wir die Ebene $\mathbb{Z} \times \mathbb{N}$ mit Kacheln ausfüllen. Wir haben unendlich viele Kacheln jedes Typs zur Verfügung, wobei alle Kacheltypen quadratisch mit Seitenlänge 1 sind. Die Verkachelung muss allerdings die folgenden Bedingungen einhalten: (i) eine spezielle "Anfangskachel" muss auf $(0, 0)$ mit seiner linken unteren Ecke plaziert sein, und (ii) nur bestimmte Kacheltypen dürfen nebeneinander liegen. Das Problem läßt sich wie folgt formalisieren.

Ein *Tiling-Muster* ist ein Tupel $\mathcal{K} = (K, k_0, H, V)$, wobei K eine endliche Menge (von Kacheltypen) ist, $k_0 \in K$ die "Anfangskachel" ist, und $H, V \subseteq K \times K$ die horizontalen bzw. vertikalen Bedingungen für benachbarte Kacheln sind. Das Auslegen von Kacheln ist durch eine Funktion $f : \mathbb{Z} \times \mathbb{N} \to K$ gegeben. Diese Funktion ist eine *Tiling-Funktion für das Muster* $\mathcal{K}$, falls

- $f(0, 0) = k_0$ (der Platz für die Anfangskachel);
- $(f(n, m), f(n + 1, m)) \in H$ für alle $n \in \mathbb{Z}$ und $m \in \mathbb{N}$;
- $(f(n, m), f(n, m + 1)) \in V$ für alle $n \in \mathbb{Z}$ und $m \in \mathbb{N}$.

Das Tiling-Problem ist dann das Entscheidungsproblem

$$TP = \left\{ \mathcal{K} \,\middle|\, \text{es gibt eine Tiling-Funktion } f : \mathbb{Z} \times \mathbb{N} \to K \text{ für } \mathcal{K} \right\}.$$

Wir geben jetzt ein Tiling-Muster $\mathcal{K} = (K, k_0, H, V)$ an, so dass eine "erfolgreiche Verkachelung" die Berechnung einer Turingmaschine $M = (Q, \Sigma, \delta, q_0, \Gamma)$ auf dem leeren Wort wiedergibt: die i-te Zeile der Verkachelung soll dabei der Konfiguration von M zum Zeitpunkt i entsprechen. (Eine Konfiguration $\ldots, a_{t-1}, (q, a_t), a_{t+1}, \ldots$ zum Zeitpunkt i besteht aus dem Bandinhalt, dem Zustand und der Kopfposition zum Zeitpunkt i.)

Wir wählen die folgenden Kacheltypen (Kacheltyp 3 ist unsere Anfangskachel k_0): siehe Abbildung 11.1.

(a) **Wie** sind die Kacheltypen $2, 5$ und 7 in Abhängigkeit von δ zu definieren, **wie** sind die horizontalen Bedingungen H und die vertikalen Bedingungen V zu definieren, so dass eine Verkachelung einer Berechnung von M auf dem leeren Wort entspricht?

(b) **Zeige**, dass das Tiling-Problem unentscheidbar ist.

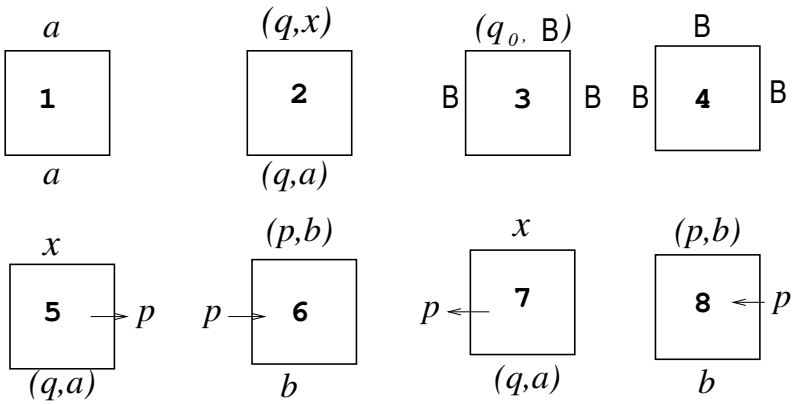


Abbildung 11.1: Hier ist $p, q \in Q$, $a, b, x \in \Gamma$ und B das Blank.

11.4 Der Satz von Rice

Der Satz von Rice besagt im wesentlichen, dass jede Frage nicht entscheidbar ist, die nach einer nicht-trivialen Eigenschaft eines beliebigen Programms fragt. Wir müssen dazu nur fordern, dass Programme in einer beliebigen, aber hinreichend ausdruckstarken Programmiersprache wie etwa C, C++, Java oder Python geschrieben sind. (Was heißt hinreichend ausdrucksstark? Die Programmiersprache muss imstande sein, ein beliebiges Turingmaschinenprogramm zu simulieren: Alles andere als eine Herkulesaufgabe!)

Für Alphabete Σ_1 und Σ_2 definieren wir

$$B_{\Sigma_1,\Sigma_2} = \{f \mid \text{die (partielle) Funktion } f : \Sigma_1^* \to \Sigma_2^* \text{ ist berechenbar } \}$$

als die Menge der berechenbaren Funktionen mit einem in Σ_1^* enthaltenen Definitionsbereich und dem Wertebereich Σ_2^*. Wir interpretieren eine Teilmenge $S \subseteq B_{\Sigma_1,\Sigma_2}$ als eine Eigenschaft. Wir nennen S *nicht-trivial*, wenn S nicht die leere Menge ist und und nicht mit B_{Σ_1,Σ_2} übereinstimmt. Wir können jetzt den Satz von Rice formulieren:

Satz 11.10 *Sei $S \subseteq B_{\Sigma_1,\Sigma_2}$. Dann ist das Problem*

$$B(S) = \{\langle M\rangle \mid \textit{die von } M \textit{ berechnete Funktion gehört zu } S\}$$

genau dann nicht entscheidbar, wenn S nicht-trivial ist, wenn also

$$S \neq \emptyset \quad \textit{und} \quad S \neq B_{\Sigma_1,\Sigma_2}$$

gilt.

Der Satz von Rice zeigt somit, dass alle nicht-trivialen Eigenschaften von Programmen nicht entscheidbar sind.

Beweis: Beachte, dass $B(\emptyset) = \emptyset$ und $B(B_{\Sigma_1,\Sigma_2}) = \{\langle M\rangle \mid M \text{ ist eine Turingmaschine } \}$: $B(S)$ ist also entscheidbar, wenn S trivial ist. Sei also eine nicht-triviale Eigenschaft S gegeben, es gelte somit $S \neq \emptyset$ und $S \neq B_{\Sigma_1,\Sigma_2}$. Wir behaupten, dass

$$\overline{H_\varepsilon} \leq B(S)$$

gilt. Die Behauptung des Satzes folgt damit, denn H_ε und damit auch $\overline{H_\varepsilon}$ ist nicht entscheidbar.

Wir nehmen an, dass die überall undefinierte Funktion nie zu S gehört. Ist dies nicht der Fall, dann ist $\mathsf{nie} \in B_{\Sigma_1,\Sigma_2} \setminus S$. Wir würden dann nachweisen, dass $B(B_{\Sigma_1,\Sigma_2} \setminus S)$ nicht entscheidbar ist und erhalten die Unentscheidbarkeit von $B(S)$, da $B(S) = \overline{B(B_{\Sigma_1,\Sigma_2} \setminus S)}$. Da $S \neq B(B_{\Sigma_1,\Sigma_2})$, gibt es eine Funktion $f \in B_{\Sigma_1,\Sigma_2} \setminus S$. Sei M_f eine Turingmaschine, die f berechnet.

Wir beschreiben jetzt die transformierende Turingmaschine T für den Nachweis der Reduktion $\overline{H_\varepsilon} \leq B(S)$. Sei $v \in \{0,1\}^*$ eine Eingabe. Wenn v keine Gödelnummer ist, dann setzen wir $T(v) = \langle M_0 \rangle$, wobei M_0 eine Turingmaschine ist, die nie hält. Beachte, dass in diesem Fall $v \in \overline{H_\varepsilon}$ und $T(v) \in B(S)$ ist. Ansonsten ist $v = \langle M \rangle$. Die transformierende Turingmaschine T modifiziert M wie folgt:

- Die tatsächliche Eingabe w von M wird nicht beachtet. Stattdessen wird M auf das leere Wort angesetzt. Die Eingabe w wird dazu auf eine zweite Spur kopiert und von der ersten Spur gelöscht.

- Wenn M auf dem leeren Wort hält, wird die Turingmaschine M_f auf der tatsächlichen Eingabe w simuliert, nachdem sämtliche Ergebnisse der Berechnung von M auf ε gelöscht wurden.

Sei M' die resultierende Turingmaschine. Wir setzen $T(\langle M \rangle) = \langle M' \rangle$. Leistet die Reduktion was wir verlangen?

- Wir haben f so gewählt, dass $f \in B_{\Sigma_1,\Sigma_2} \setminus S$ gilt. Die Maschine M_f berechnet also eine Funktion, die nicht zu S gehört und aus $\langle M \rangle \in H_\varepsilon$ folgt $T(\langle M \rangle) = \langle M_f \rangle \notin B(S)$.

- Gilt hingegen $\langle M \rangle \notin H_\varepsilon$, dann berechnet M' die Funktion nie.

Wir können jetzt zusammenfassen: Es gilt

$$\begin{aligned} \langle M \rangle \in \overline{H_\varepsilon} &\Leftrightarrow M \text{ hält nicht auf dem leeren Wort} \\ &\Leftrightarrow M' \text{ berechnet } \mathsf{nie} \\ &\Leftrightarrow \langle M' \rangle \in B(S). \end{aligned}$$

Die Behauptung des Satzes folgt, da die transformierende Turingmaschine T stets hält. ■

Der Satz von Rice liefert eine Fülle unentscheidbarer Probleme. Wir listen einige Beispiele auf:

(a) $L_1 = \{ \langle M \rangle \mid M \text{ hält für irgendeine Eingabe } \}$ ist unentscheidbar,
denn $L_1 = B(B_{\Sigma_1,\Sigma_2} \setminus \{\mathtt{nie}\})$.

(b) $L_2 = \{\langle M \rangle \mid M \text{ hält für keine Eingabe } \}$ ist unentscheidbar,
denn $L_2 = B(\{\mathsf{nie}\})$ mit $\mathsf{nie}(w) =$ undefiniert für alle w).

(c) $L_3 = \{\langle M \rangle \mid M \text{ berechnet auf allen Eingaben die Ausgabe } 0\}$ ist unentscheidbar,
denn $L_3 = \mathrm{TM}(\{0\})$, wobei $0(w) = 0$ für alle Worte w).

Unsere alten Unentscheidbarkeitsergebnisse für die universelle Sprache und das Halteproblem sind allerdings keine Konsequenzen des Satzes von Rice, weil Eingaben von U und H von der Form $\langle M \rangle w$ sind, während im Satz von Rice Eingaben der Form $\langle M \rangle$ erwartet werden. Die Unentscheidbarkeit des speziellen Halteproblems hingegen ist eine direkte Konsequenz: Wähle $S = \{f \mid f(\varepsilon) \text{ ist definiert } \}$.

Aufgabe 89
Zeige die folgenden Aussagen.

(a) Es gibt keinen Algorithmus, der für jedes Programm P entscheidet, ob P unabhängig von seiner Eingabe immer dieselbe Ausgabe produziert.

(b) Sei $L \subseteq \Sigma^*$ eine beliebige Menge. Es gibt keinen Algorithmus, der für jedes Programm P entscheidet, ob P genau die Worte in L akzeptiert.

(c) Es gibt keinen Algorithmus, der für jedes Programms P entscheidet, ob die Menge der von P akzeptierten Worte eine reguläre Sprache ist.

11.5 Das Postsche Korrespondenzproblem

Das Postsche Korrespondenzproblem (PKP) ist ein wichtiges unentscheidbares Problem, das häufig zum Nachweis der Unentscheidbarkeit von Eigenschaften formaler Sprachen und insbesondere von Eigenschaften kontextfreier Sprachen eingesetzt wird.

Die Eingabe von PKP besteht aus einem endlichen Alphabet Σ, einer Zahl $k \in \mathbb{N}$ $(k > 0)$ und einer Folge $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$ von Wortpaaren mit $x_1, y_1, \ldots, x_k, y_k \in \Sigma^+$. Es ist zu entscheiden, ob es ein $n \in \mathbb{N}$ $(n > 0)$ und Indizes $i_1, \ldots, i_n \in \{1, \ldots, k\}$ gibt, so dass $x_{i_1} x_{i_2} \cdots x_{i_n} = y_{i_1} y_{i_2} \cdots y_{i_n}$ gilt. Mit anderen Worten, es ist zu entscheiden, ob es eine x-Folge gibt, deren Konkatenation mit „ihrer“ Partnerfolge übereinstimmt.

Bei PKP handelt es sich somit um eine Art von Dominospiel, bei dem die Dominosteine nicht wie üblich ein linkes und rechtes Feld, sondern diesmal ein oberes Feld (beschriftet mit x_i) und ein unteres Feld (beschriftet mit y_i) besitzen. Wir müssen die Dominosteine so nebeneinander legen, dass die obere mit der unteren Zeile übereinstimmt.

Beispiel 11.1 Das PKP mit Eingabe $\Sigma = \{0, 1\}$, $k = 3$ und

$$(x_1, y_1) = (1, 111), \qquad (x_2, y_2) = (10111, 10), \qquad (x_3, y_3) = (10, 0).$$

hat eine Lösung mit $n = 4$ und $i_1 = 2, i_2 = 1, i_3 = 1, i_4 = 3$, denn wenn wir die „Dominosteine“ nebeneinander legen erhalten wir

$$\begin{aligned} x_2 x_1 x_1 x_3 &= 10111\ 1\ \ 1\ 10 \\ y_2 y_1 y_1 y_3 &= \ \ 10\ 111\ 111\ 0 \end{aligned}$$

und die untere stimmt mit der oberen Zeile überein.

Bevor wir zeigen, dass PKP unentscheidbar ist, betrachten wir einige Problemvarianten. Die Eingabe des modifizierten PKP (MPKP) ist identisch zur Eingabe von PKP. Diesmal ist aber zu entscheiden, ob es eine mit x_1 beginnende x-Folge gibt, die mit ihrer Partnerfolge übereinstimmt.

Beispiel 11.2 Betrachtet als Eingabe für das MPKP hat das obige Beispiel keine Lösung. Warum?

PKP_Σ ist eine letzte PKP-Variante: Hier ist das Alphabet nicht mehr Teil der Eingabe, sondern von vorne herein fixiert.

Satz 11.11 *Es gilt* $U \leq \text{MPKP} \leq \text{PKP} \leq \text{PKP}_{\{0,1\}}$.

Insbesondere sind das Postsche Korrespondenzproblem PKP sowie seine Varianten MPKP und $PKP_{\{0,1\}}$ *nicht entscheidbar.*

Beweisskizze: Wir zeigen zuerst die Reduktion $\text{PKP} \leq \text{PKP}_{\mathbf{\{0,1\}}}$. Für ein endliches Alphabet $\Sigma = \{a_1, \ldots, a_m\}$ geben wir zuerst eine Binärkodierung seiner Buchstaben an und zwar kodieren wir a_j durch $h(a_j) = 0\,1^j$ für alle $j \in \{1, \ldots, m\}$. Die Reduktion von PKP auf $\text{PKP}_{\{0,1\}}$ weist einer Eingabe $\Sigma,\ k,\ (x_1, y_1), \ldots, (x_k, y_k)$ für's PKP die folgende Eingabe für's $\text{PKP}_{\{0,1\}}$ zu: k ist unverändert, aber wir benutzen diesmal die binären Wortpaare $(h(x_1), h(y_1)), \ldots, (h(x_k), h(y_k))$.

Warum funktioniert die Reduktion? Für alle $n \in \mathbb{N}$ $(n > 0)$ und alle $i_1, \ldots, i_n \in \{1, \ldots, k\}$ gilt:

$$x_{i_1} x_{i_2} \cdots x_{i_n} \;=\; y_{i_1} y_{i_2} \cdots y_{i_n} \quad \Longleftrightarrow \quad h(x_{i_1})h(x_{i_2}) \cdots h(x_{i_n}) = h(y_{i_1})h(y_{i_2}) \cdots h(y_{i_n}).$$

Da die Reduktion f berechenbar ist, ist f eine Reduktion von PKP auf $\text{PKP}_{\{0,1\}}$.

Die Reduktion $\text{MPKP} \leq \text{PKP}$ stellen wir als Übungsaufgabe. Wir kommen also jetzt zum zentralen Ziel, nämlich dem Nachweis der Reduktion $U \leq \text{MPKP}$. **Gesucht** ist eine Transformation f, die jedem Wort $u \in \{0,1\}^*$ eine Eingabe $f(u)$ für's MPKP zuordnet, so dass

$$\begin{array}{l} u \in H, \text{ d.h. } u = \langle M \rangle\, w \text{ für eine} \\ \text{TM } M, \text{ die Eingabe } w \text{ akzeptiert} \end{array} \Longleftrightarrow \text{ das MPKP } f(u) \text{ besitzt eine Lösung.}$$

Betrachten wir zuerst den **leichter Fall**, dass u nicht von der Form $\langle M \rangle\, w$. Dann wählen wir eine Eingabe $f(u)$ für's MPKP, die keine Lösung besitzt, z.B., $\Sigma = \{0,1\}$, $k = 1$, $x_1 = 0$, $y_1 = 1$.

Kümmern wir uns um den **schwierigen Fall**, nämlich $u = \langle M \rangle\, w$ für eine Turingmaschine $M = (\Sigma, Q, \Gamma, \delta, q_0, F)$. Wir nehmen o.B.d.A. an, dass M nur dann in einen akzeptierenden Zustand $q \in F$ wechselt, wenn sie unmittelbar danach anhält. Schließlich fordern wir ebenfalls o.B.d.A, dass M nie den Kopf auf einer Zelle stehen lässt und nie seinen Eingabebereich nach links hin verlässt.

Hier ist die Idee der Reduktion. Wir repräsentieren eine Konfigurationen von M durch Worte über dem Alphabet $\Gamma \dot\cup Q$ wie folgt:

> uqv repräsentiert die Situation, bei der M im Zustand q ist, die Bandinschrift uv ist, und der Kopf auf dem ersten Symbol von v steht. Die Startkonfiguration bei Eingabe w wird also durch $q_0 w$ repräsentiert.

Wir geben jetzt eine Eingabe $f(\langle M \rangle\, w)$ für's MPKP an, die aufeinander folgende Konfigurationen von M erzeugt, also die Berechnung von M nachmacht.

- Das Alphabet ist $\Gamma \dot\cup Q \dot\cup \{\#\}$. Das Symbol # dient als Trennsymbol zwischen den einzelnen Konfigurationen.

- Für ein geeignetes $k \in \mathbb{N}$ wählen wir Wortpaare $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$ wie folgt, wobei

  $$(x_1, y_1) \text{ mit } x_1 := \# \text{ und } y_1 := \#q_0w\#$$

  das Startpaar ist. Das Wort y_1 beschreibt also die Startkonfiguration, wenn M auf dem leeren Wort rechnet. (Wenn w das leere Wort ist, dann wähle $y_1 = \#q_0\mathtt{B}\#$.) Als weitere „Regeln“ verwenden wir

  – für Rechtsbewegungen $\delta(q, a) = (q', a', \mathtt{rechts})$ das Paar

  $$(qa, a'q'),$$

  solange $q \neq q_v$,

  – für Linksbewegungen $\delta(q, a) = (q', a', \mathtt{links})$ das Paar

  $$(bqa, q'ba')$$

  mit $b \in \Gamma$, solange $q \neq q_v$,

  – die Kopierregeln mit den Paaren

  $$(a, a)$$

  für $a \in \Gamma$

  – sowie die Abschlußregeln

  $$(\#, \#), \text{ bzw. } (\#, \mathtt{B}\#)$$

  für $q \in F$.

Tatsächlich müssen wir noch weitere Regeln aufnehmen, um die folgende Behauptung zu zeigen.

Behauptung: M akzeptiert Eingabe $w \iff f(\langle M \rangle)$ besitzt eine Lösung mit Startpaar (x_1, y_1).

Wie müssen die fehlenden Regeln aussehen? Betrachten wir die Richtung $\Rightarrow$. Unser Ziel ist die Konstruktion übereinstimmender x- und y-Zeilen. Das Startpaar $(\#, \#q_0w\#)$ erzwingt dann ein Wort x_2 das mit q_0 beginnt und damit muss die Regel $(q_0w_1, a'q')$ für eine Rechtsbewegung $\delta(q_0, w_1) = (q', a', \mathtt{rechts})$ gewählt werden. Eine solche Regel steht aber nur dann zur Verfügung, wenn M auf w_1 eine Rechtsbewegung im Zustand q_0 durchführt: Das muss M aber tun, denn sonst verlässt es seinen Eingabebereich nach links hin. Also ist zwangsläufig $x_2 = q_0w_1$ und $y_2 = a'q'$.

Die x-Zeile ist jetzt $\#q_0w_1$ und die y-Zeile ist $\#q_0w_1 \cdots w_n\#a'q'w_2$. Wir sind gezwungen ein mit w_2 beginnendes Wort x_3 zu wählen und der Begrenzer $\#$ zwingt uns die Kopierregeln anzuwenden, bis wir die x-Zeile $\#q_0w_1 \cdots w_n$ und die y-Zeile $\#q_0w_1 \cdots w_n\#a'q'w_2 \cdots w_n$ erreicht haben. Auch der nächste Schritt ist erzwungen, denn der Begrenzer muss hinzugefügt werden. (Die Abschlussregel $(\#, \mathtt{B}\#)$ ist zu wählen, wenn $w = w_1$ und M somit im nächsten Schritt das Blanksymbol liest.)

Unsere Regeln gewährleisten also, dass x- und y-Zeile eine Berechnung von M auf Eingabe w simulieren! Wir wissen nach Fallannahme, dass M die Eingabe w akzeptiert und deshalb, wenn wir unsere Konstruktion genügend lange fortsetzen, „endet“ die y-Zeile mit einem Zustand in F. Wir fügen jetzt die Löschregeln

$$(qa, a) \text{ bzw. } (aq, q)$$

für jeden akzeptierenden Zustand q hinzu und können damit (und mit den Kopierregeln) die x-Zeile gegenüber der um die letzte Konfiguration längeren y-Zeile langsam auffüllen. Am Ende haben wir fast Gleichstand erhalten, allerdings ist $y \equiv xq\#$. Wir fügen deshalb noch als letzte Regel

$$(q\#\#, \#)$$

hinzu und haben Gleichstand erreicht.

Da die Umkehrung $\Leftarrow$ mit gleicher Argumentation folgt, ist die Beweisskizze erbracht. ■

Die Unentscheidbarkeit des Postschen Korrespondenzproblems hat viele Konsequenzen für die Unentscheidbarkeit von Problemen im Bereich der Formalen Sprachen. Wir geben eine erste solche Konsequenz für *Finite State Transducer* (FST) an: Ein FST $A = (Q, \Sigma, \Gamma, I, F, \delta)$ ist wie ein nichtdeterministischer endlicher Automat mit Ausgabe aufgebaut. Insbesondere ist

- Q die Zustandsmenge, $I \subseteq Q$ die Menge der Anfangszustände und F die Menge der akzeptierenden Zustände,
- Σ ist das Eingabealphabet und Γ das Ausgabealphabet.
- Die Überführungsrelation δ hat die Form

  $$\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\} \times \Gamma \cup \{\varepsilon\}) \times Q.$$

  Wenn sich die Maschine A im Zustand q befindet und den Buchstaben $a \in \Sigma$ liest, dann darf A genau dann in den Zustand q' wechseln und den Buchstaben a' drucken, wenn $(q, a, a', q') \in \delta$.

Wir sagen, dass A unter Eingabe $x \in \Sigma^*$ die Ausgabe $y \in \Gamma^*$ produzieren kann, wenn es einen Anfangszustand $i \in I$ gibt, so dass eine in Zustand i beginnende Berechnung für Eingabe x die Ausgabe y produziert.

Die Äquivalenz von NFA's ist eine durchaus schwierige Aufgabe, die aber entscheidbar ist: Überführe die nfa's in dfa's und überprüfe die Äquivalenz der dfa's. Die Äquivalenz von FST's hingegen ist überraschender Weise (?!) unentscheidbar.

Korollar 11.12 *Die Frage, ob zwei gegebene Finite State Transducer dieselbe Sprache erkennen, ist nicht entscheidbar.*

Beweis: Wir reduzieren $\text{PKP}_{\{0,1\}}$ auf das Äquivalenzproblem für Finite State Transducer.

Seien k und die Paare $(x_1, y_1), \ldots, (x_k, y_k)$ eine Eingabe für $\text{PKP}_{\{0,1\}}$. Für das Eingabealphabet $\Sigma = \{1, \ldots, k\}$ und das Ausgabealphabet $\Gamma = \{0, 1\}$ bauen wir zuerst einen recht trivialen FST A_1, der für jede Eingabe $\xi \in \Sigma^*$ jede Ausgabe $\eta \in \Gamma^*$ produzieren kann. Der FST A_2 ist ebenfalls sehr einfach: Für eine Eingabe $i_1 \cdots i_n \in \Sigma^*$ produziert A_2 jede mögliche Ausgabe $w \in \Gamma^*$, falls $w \neq x_{i_1} \cdots x_{i_n}$ oder $w \neq y_{i_1} \cdots y_{i_n}$.

Angenommen, es gibt eine Lösung für $\text{PKP}_{\{0,1\}}$ und die Eingabe $(x_1, y_1), \ldots, (x_k, y_k)$. Dann gibt es $i_1, \ldots, i_n$ mit $x_1 \cdots x_k = y_1 \cdots y_k$. Dann wird aber A_2 nicht die Ausgabe $x_{i_1} \cdots x_{i_n}$ produzieren und damit unterscheidet sich A_1 von A_2. Ist $\text{PKP}_{\{0,1\}}$ für Eingabe $(x_1, y_1), \ldots, (x_k, y_k)$ hingegen nicht lösbar, dann sind A_1 und A_2 äquivalent. ■

Aufgabe 90
Zeige: Die Frage, ob zwei gegebene Kellerautomaten dieselbe Sprache akzeptieren, ist nicht entscheidbar.

Kapitel 12

Rekursiv aufzählbare Probleme

Wir zeigen zuerst, dass entscheidbare Probleme unter den Operationen Vereinigung, Durchschnitt und Komplement abgeschlossen sind.

Satz 12.1 *L_1 und L_2 seien entscheidbare Probleme. Dann sind auch*

$$L_1 \cup L_2, \quad L_1 \cap L_2 \quad \textit{und} \quad \overline{L_1}$$

entscheidbar.

Beweis: L_1 (bzw. L_2) werde durch die stets haltende Turingmaschine M_1 (bzw. M_2) gelöst. Um $L_1 \cup L_2$ zu lösen, entwerfen wir eine Turingmaschine M, die zuerst M_1 simuliert und sodann M_2 simuliert. M wird genau dann akzeptieren, wenn eine der beiden Turingmaschinen akzeptiert. Da M stets hält, ist $L_1 \cup L_2$ entscheidbar.

Wenn L_1 entscheidbar ist, ist offensichtlich auch das Komplement $\overline{L_1}$ entscheidbar; ersetze F durch $Q \setminus F$.

Da $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$, folgt die Entscheidbarkeit des Durchschnitts aus der Entscheidbarkeit von Vereinigung und Komplementbildung. ■

Wir führen als nächstes den Begriff der rekursiven Aufzählbarkeit ein:

Definition 12.2 *Ein Entscheidungsproblem K heißt genau dann* rekursiv aufzählbar *(oder semi-entscheidbar), wenn es eine Turingmaschine M mit $K = L(M)$ gibt, d.h. es muss gelten*

$$w \in K \;\Leftrightarrow\; M \textit{ akzeptiert } w.$$

Die Turingmaschine M muss also genau die Worte in K akzeptieren. Für Eingaben $w \notin K$ muss M entweder verwerfen oder M darf nicht halten.

Warum haben wir den Namen „rekursiv aufzählbar“ benutzt?

Aufgabe 91
L sei ein Problem. **Beweise** die folgende Aussage.

> Es gibt genau dann eine deterministische Turingmaschine, die alle Wörter $w \in L$ (und nur die Wörter $w \in L$) in irgendeiner Reihenfolge (durch ein # getrennt) auf ein Ausgabeband schreibt, wenn L rekursiv aufzählbar ist.

Es ist erlaubt, dass Wörter mehrfach auf dem Ausgabeband der deterministischen Turingmaschine erscheinen.

Offensichtlich ist jedes entscheidbare Problem auch rekursiv aufzählbar. Die Option, für Eingaben außerhalb des Entscheidungsproblems nicht halten zu müssen, führt dazu, dass es sehr viel mehr rekursiv aufzählbare als entscheidbare Probleme gibt. Unser nächstes Ergebnis erklärt warum.

Satz 12.3 *Die Probleme H, H_ε und U sind rekursiv aufzählbar, aber nicht entscheidbar.*

Beweis: Warum ist zum Beispiel das Halteproblem H rekursiv aufzählbar? Für Eingabe v überprüfe zuerst, ob $v = \langle M \rangle w$. Wenn nicht, verwerfe. Ansonsten benutze die universelle Turingmaschine, um M auf Eingabe w zu simulieren. Sollte M auf w halten, so akzeptiere.

Sei N die gerade beschriebene Turingmaschine. Dann gilt offensichtlich

$$\begin{aligned} v \in H &\Leftrightarrow v = \langle M \rangle w \quad \text{und } M \text{ hält auf } w \\ &\Leftrightarrow N \text{ akzeptiert } w. \end{aligned}$$

Beachte aber, dass N nicht stets halten wird! ■

Es gibt weitere wichtige rekursiv aufzählbare Probleme (oder Mengen). Im nächsten Abschnitt lernen wir die Menge aller aus einem gegebenen Axiomensystem ableitbaren Formeln kennen. Diese Menge ist rekursiv aufzählbar, weil wir für jede Formel ϕ systematisch alle Beweise aufzählen können und jeweils nachprüfen können, ob wir gerade ϕ bewiesen haben. Ist ϕ tatsächlich beweisbar, werden wir ihren Beweis irgendwann finden und werden ϕ somit akzeptieren. Ist ϕ hingegen nicht beweisbar, läuft unsere Berechnung ohne Ende weiter: Das ist aber nicht schlimm, da wir ϕ ja sowieso nicht akzeptieren dürfen.

Ein weiteres wichtiges Beispiel eines rekursiv aufzählbaren Problems ist die Menge aller aus einer Grammatik G ableitbaren Worte. Mehr dazu kann man in der Theoretischen Informatik 2 erfahren. Wir zeigen als nächstes, dass rekursiv aufzählbare Probleme unter den Operationen Vereinigung und Durchschnitt abgeschlossen sind.

Satz 12.4 *L_1 und L_2 seien rekursiv aufzählbar. Dann sind auch $L_1 \cup L_2$ und $L_1 \cap L_2$ rekursiv aufzählbar.*

Beweis: Es sei $L_1 = L(M_1)$ und $L_2 = L(M_2)$. Wir bauen eine Turingmaschine M, die die beiden Turingmaschinen M_1 und M_2 „verzahnt“ simuliert:

M hat eine Spur für M_1 und eine Spur für M_2. In der ersten Phase wird M einen Schritt von M_1 und einen Schritt von M_2 simulieren.

Wenn $L_1 \cup L_2$ zu lösen ist, wird M halten und akzeptieren, sobald eine der beiden Maschinen M_1 oder M_2 akzeptiert. (Warum reicht es nicht, zuerst M_1 vollständig zu simulieren und sodann M_2 zu simulieren?)

Wenn $L_1 \cap L_2$ zu lösen ist, wird M akzeptieren, sobald beide Maschinen akzeptiert haben. ■

Aufgabe 92
Zeige, wenn $L_1 \leq L_2$ und L_2 rekursiv aufzählbar ist, dann ist auch L_1 rekursiv aufzählbar.

Ganz im Gegensatz zu entscheidbaren Problemen sind aber rekursiv aufzählbare Probleme *nicht* unter Komplementbildung abgeschlossen.

Satz 12.5

(a) Wenn L und $\overline{L}$ rekursiv aufzählbar sind, dann ist L entscheidbar.

(b) $\overline{H_\varepsilon}$, $\overline{H}$ und $\overline{U}$ sind nicht rekursiv aufzählbar.

Beweis: :

(a) Es sei $L = L(M_1)$ und $\overline{L} = L(M_2)$. Wir bauen, wie in Satz 12.4, eine Turingmaschine M, die M_1 und M_2 verzahnt simuliert.

M akzeptiert genau dann, wenn M_1 akzeptiert und M verwirft genau dann, wenn M_2 akzeptiert. Beachte, dass aber für jede Eingabe w genau eine der Maschinen M_1 oder M_2 akzeptieren muss, da $L(M_1) \cup L(M_2) = \Sigma^*$ und $L(M_1) \cap L(M_2) = \emptyset$.

Also ist M eine stets haltende Turingmaschine, die L löst, und L ist somit entscheidbar.

(b) ist eine Konsequenz der folgenden allgemeineren Aussage: Wenn L rekursiv aufzählbar, aber nicht entscheidbar ist, dann ist $\overline{L}$ nicht rekursiv aufählbar.

Warum? Sonst wäre L ja entscheidbar. ■

Aufgabe 93
Beweise den folgenden Satz:

Ein Problem L ist genau dann rekursiv aufzählbar, wenn es eine nichtdeterministische Turingmaschine gibt, die L löst.

Aufgabe 94
Die Probleme L_1 und L_2, über dem gemeinsamen Alphabet Σ, seien vorgegeben.

Zeige, dass $L = (L_1 \setminus L_2)^*$ entscheidbar ist, wenn L_1 und L_2 entscheidbar sind.

Zeige, dass die Konkatenation $L = L_1 \circ L_2$ rekursiv aufzählbar ist, wenn L_1 und L_2 rekursiv aufzählbar sind.

12.1 Gödels Unvollständigkeitssatz

Die Peano-Arithmetik ist ein Beweissystem der Prädikatenlogik der ersten Stufe, das erlaubt, zahlentheoretische Aussagen herzuleiten. Die Axiome der Peano-Arithmetik formulieren die Eigenschaften der Addition ($x+0 = x$, $x+(y+1) = (x+y)+1$), der Multiplikation ($x \cdot 0 = 0$, $x \cdot (y+1) = x \cdot y + x$) sowie die Induktionseigenschaft: Für jede Formel $\phi(x)$ wird die Formel

$$(\phi(0) \wedge \forall x(\phi(x) \rightarrow \phi(x+1))) \rightarrow \forall x \phi(x)$$

als Axiom aufgenommen. Gibt es wahre zahlentheoretische Aussagen, die aber in der Peano-Arithmetik nicht beweisbar sind? Angenommen, jede wahre Aussage ist auch beweisbar. Mit Satz 12.45 können wir für jedes rekursiv aufzählbare Problem L eine Formel φ_L der Peano-Arithmetik bestimmen, so dass

$$x \in L \Leftrightarrow \varphi_L(x) \text{ ist wahr}$$

gilt. Wenn Wahrheit und Beweisbarkeit übereinstimmen, folgt also

$$x \in L \Leftrightarrow \varphi_L(x) \text{ ist aus den Axiomen der Peano Arithmetik beweisbar}$$

und ebenso natürlich

$$\begin{aligned} x \notin L &\Leftrightarrow \neg\varphi_L(x) \text{ ist wahr.} \\ &\Leftrightarrow (\neg\varphi_L(x)) \text{ ist beweisbar.} \end{aligned}$$

Also ist

$$\bar{L} = \{x | (\neg\varphi_L(x)) \text{ ist beweisbar } \}$$

und damit ist $\bar{L}$ rekursiv aufzählbar. Warum?

Aufgabe 95
Sei P eine rekursiv aufzählbare Menge von Formeln der Zahlentheorie. Dann ist die Menge der aus P beweisbaren Formeln rekursiv aufzählbar.

Also ist für jedes rekursiv aufzählbare Problem L auch das Komplement rekursiv aufzählbar, und jedes rekursiv aufzählbare Problem ist auch entscheidbar! Wir haben einen Widerspruch erhalten und damit gezeigt, dass nur eine echte Teilmenge aller wahren Formeln auch beweisbar ist.

Nun könnte man zu den Axiomen der Peano Arithmetik weitere wahre Formeln hinzunehmen und hoffen, dass das neue Axiomensystem P „vollständig“ ist, also für alle wahren Formeln auch Beweise zulässt. Mitnichten, denn entweder ist P nicht rekursiv aufzählbar (und wir können noch nicht einmal verifizieren, dass ein Axiom zu P gehört) oder es gibt weiterhin wahre, aber nicht beweisbare Aussagen.

Satz 12.6 *Gödelscher Unvollständigkeitssatz.*

Sei P eine rekursiv aufzählbare Menge von wahren Formeln der Zahlentheorie, wobei P die Axiome der Peano-Arithmetik enthalte. Dann gibt es eine wahre Formel, die nicht aus P ableitbar ist.

Beweis: Wenn Wahrheit mit Beweisbarkeit zusammenfällt, dann können wir die obigen Argumente wieder anwenden. (Wir benötigen die rekursive Aufzählbarkeit von P, um wieder schließen zu können, dass die Menge $\bar{L}$ rekursiv aufzählbar ist). ■

Die Menge aller für die natürlichen Zahlen wahren Formeln ist also nicht nur nicht entscheidbar, sondern es gibt sogar kein (rekursiv aufzählbares) Axiomensystem, das erlaubt alle wahren Formeln abzuleiten: `Die Wirklichkeit kann nicht formal beschrieben werden!`

12.2 μ-rekursive Funktionen*

Wir führen einen gänzlich neuen Formalismus einen, der uns aber wiederum das Konzept der berechenbaren Funktionen liefern wird. Dass wir von zwei verschiedenen Konzepten ausgehend dieselbe Menge von Funktionen berechnen, ist ein weiteres Indiz für die Church-Turing These.

Der im folgenden dargestellte Formalismus hat den zusätzlichen Vorteil, dass wir für eine große Klasse von Funktionen, nämlich die primitiv rekursiven Funktionen, einen *konstruktiven* Beweis ihrer Berechenbarkeit erhalten.

Zusätzlich können wir eine Idee vom Gödelschen Unvollständigkeitssatz vermitteln:

> Jede rekursiv aufzählbare Axiomatisierung der Zahlentheorie besitzt wahre, aber nicht beweisbare, Aussagen.

Also ist nicht nur das Ableiten oder Beweisen ableitbarer Aussagen schwierig, sondern wir stoßen auf ein viel fundamentaleres Problem: selbst der mächtige Formalismus der Prädikatenlogik kann eine komplexe Realität nicht exakt widerspiegeln!

12.2.1 Primitiv rekursive Funktionen

Zunächst müssen wir die Verbindung zwischen den Berechnungen einer Maschine und mathematischen Funktionen herstellen. Was tut eine Maschine? Sie nimmt eine Eingabe entgegen und berechnet daraus eine Ausgabe. Da wir bisher ausschließlich über deterministische Maschinenmodelle reden, liefert die Maschine für dieselbe Eingabe auch stets dieselbe Ausgabe. Die Berechnung beschreibt somit eine Funktion.

Entscheidungsprobleme, als Teilmenge von Σ^* verstanden, lassen sich als Prädikate (also Eigenschaften) auffassen. Zu einem gegebenen $x \in \Sigma^*$ soll entschieden werden, ob x zum Entscheidungsproblem L gehört oder nicht. Das entspricht genau der Frage, ob das x das Prädikat *„zu L gehörend“* erfüllt oder nicht.

Als erstes betten wir das Problem der Entscheidbarkeit von Entscheidungsproblemen in das Problem der Berechnung von Funktionen ein.

Definition 12.7 *Sei $L \subseteq \Sigma^*$ ein Entscheidungsproblem. Die Funktion $f_L : \Sigma^* \to \{0,1\}$ heißt charakteristische Funktion oder Indikatorfunktion von L, falls*

$$\forall x \in \Sigma^* : x \in L \leftrightarrow f(x) = 1$$

gilt.

Beispiel 12.1 Sei $L \subseteq \{0,1\}^*$ die Menge der Worte mit gerader Anzahl von Einsen. Dann ist $f_L : \{0,1\}^* \to \{0,1\}$ mit

$$f(w) = 1 - \left(\sum_{i=1}^{|w|} w_i \right) \bmod 2$$

die Indikatorfunktion.

Wir sehen, dass wir ein Entscheidungsproblem immer dann handhaben können, wenn wir seine Indikatorfunktion in den Griff bekommen. Wir konzentrieren uns daher zunächst auf die Berechenbarkeit von Funktionen.

Wie schon bei unserem Ansatz mit dem Modell der Turingmaschine werden wir hier recht spartanisch vorgehen. Wir versuchen nur mit ganz elementaren Konzepten, deren Berechenbarkeit niemand ernsthaft in Frage stellt, auszukommen.

Bei den Turingmaschinen haben wir unterstellt, dass ein Rechner ein Zeichen aus einem vorgegebenen Zeichensatz wiedererkennen (lesen) und reproduzieren (schreiben) kann. Wir haben unterstellt, dass wir uns in einem linearen Speicherband in beide Richtungen bewegen können und angenommen, dass unser Maschinenmodell endlich viele Zustände auseinanderhalten kann. Eigentlich ist das ein recht mageres Anforderungsprofil. Dennoch konnten wir zeigen, dass jede Funktion, die von einem modernen Parallelrechner berechnet werden kann, auch von einer Turingmaschine berechnet werden kann.

Wir beginnen an dieser Stelle ähnlich, indem wir von den folgenden simplen *Grundfunktionen* ausgehen.

Definition 12.8 *Die* Nachfolgerfunktion $S : \mathbb{N} \to \mathbb{N}$ *ordnet jeder natürlichen Zahl x ihren Nachfolger $x + 1$ zu, also die Funktion, die x auf die nächstfolgende Zahl $S(x)$ abbildet.*

$$S(x) = x + 1$$

(Der Buchstabe S soll an das englische *successor* erinnern).

Definition 12.9 *Die* Konstantenfunktion $C_q^k : \mathbb{N}^k \to \mathbb{N}$ *bildet jeden Vektor $\vec{x} \in \mathbb{N}^k$ auf die Konstante q ab.*

$$C_q^k(x_1, x_2, \ldots, x_k) = q$$

Diese Funktionen leisten nicht mehr, als dass sie alle ihre Argumente ignorieren und die betreffende Konstante q als Resultat liefern. Auch diese Funktionen werden uns nur schwerlich Schwierigkeiten bei der Berechnung machen.

Definition 12.10 *Die* Projektionsfunktion $P_i : \mathbb{N}^k \to \mathbb{N}$ *mit $1 \leq i \leq k$ ist die Funktion, die $\vec{x} \in \mathbb{N}^k$ auf die i-te Koordinate x_i abbildet.*

$$P_i^k(x_1, x_2, \ldots, x_i, \ldots, x_k) = x_i$$

Das heißt, wir erlauben uns, aus einer endlichen geordneten Menge von Argumenten ein konkretes auszuwählen und damit weiterzuarbeiten. Auch das ist intuitiv problemlos möglich.

Diese drei Funktionentypen nennt man *Grundfunktionen*. Um jedoch auch interessante Funktionen zu erfassen, benötigen wir noch Konstruktionsprinzipien, die uns erlauben, aus bereits existierenden Funktionen neue zu bauen.

Ein wesentliches Konstruktionsprinzip ist das der *Einsetzung*. Wir wollen zulassen, dass die Argumente einer Funktion ihrerseits Werte von weiteren, bereits berechneten Funktionen sind. Somit erhalten wir die Möglichkeit, durch das Einsetzen von schon bekannten Funktionen zu neuen Funktionen zu kommen.

Definition 12.11 *Sind $\psi, \chi_1, \chi_2, \ldots, \chi_m$ Funktionen mit $\chi_i : \mathbb{N}^n \to \mathbb{N}$ für alle i und mit $\psi : \mathbb{N}^m \to \mathbb{N}$, so heißt die durch*

$$\varphi(x_1, x_2, \ldots, x_n) = \psi(\chi_1(x_1, x_2, \ldots, x_n), \ldots, \chi_m(x_1, x_2, \ldots, x_n))$$

definierte Funktion $\varphi : \mathbb{N}^n \to \mathbb{N}$ die durch Einsetzung von $\chi_1, \chi_2, ..., \chi_n$ in ψ erhaltene Funktion.

Schließlich erlauben wir noch die *induktive* Definition einer Funktion, wie sie aus der Mathematik vertraut ist.

Definition 12.12 *Ist $\chi : \mathbb{N}^2 \to \mathbb{N}$ eine Funktion und $q \in \mathbb{N}$ eine Konstante, so ist $\varphi : \mathbb{N} \to \mathbb{N}$, definiert als*

$$\begin{aligned} \varphi(0) &= q \\ \varphi(S(x)) &= \chi(x, \varphi(x)), \end{aligned}$$

die durch Induktion aus χ mit Anfangswert q hervorgegangene Funktion.

Vollständigkeitshalber erlauben wir noch die Induktion für Funktionen mit mehr als einem Argument. In der Sache ändert sich nicht viel. Es wird jedoch zugelassen, dass der Anfangswert und die Berechnung des Nachfolgerwertes auch von den übrigen Argumenten abhängt.

Das Induktionsprinzip schreibt sich dann wie folgt:

Definition 12.13 *Sind $\chi : \mathbb{N}^{n+1} \to \mathbb{N}$ und $\psi : \mathbb{N}^{n-1} \to \mathbb{N}$ Funktionen, so ist $\varphi : \mathbb{N}^n \to \mathbb{N}$, definiert als*

$$\begin{aligned} \varphi(0, x_2, \ldots, x_n) &= \psi(x_2, \ldots, x_n) \\ \varphi(S(x_1), x_2, \ldots, x_n) &= \chi(x_1, \varphi(x_1, x_2, \ldots, x_n), x_2, \ldots, x_n), \end{aligned}$$

die durch Induktion aus χ mit Anfangsfunktion ψ hervorgegangene Funktion.

Wir werden sehen, dass wir mit den beiden Konstruktionsprinzipien *Einsetzung* und *Induktion* aufbauend auf den Grundfunktionen *Nachfolger*, *Konstanten* und *Projektionen* bereits eine recht beachtliche Menge von Funktionen darstellen können. Diese Menge bezeichnen wir als die Menge der *primitiv rekursiven Funktionen.*

Definition 12.14 *Eine Funktion heißt* primitiv rekursiv*, wenn sie sich durch endlich viele Anwendungen von Einsetzung und Induktion basierend auf den Grundfunktionen darstellen läßt.*

Ein Prädikat heißt primitiv rekursiv, wenn seine Indikatorfunktion primitiv rekursiv ist.

Bevor wir im nächsten Abschnitt die primitive Rekursivität einiger interessanter Funktionen zeigen, versuchen wir an dieser Stelle, eine Brücke zu unseren Betrachtungen des letzten Kapitels zu schlagen.

Was läßt sich über die Berechenbarkeit von primitiv rekursiven Funktionen sagen?

(a) Sicherlich werden wir ein stets terminierendes Programm schreiben können, welches eine natürliche Zahl um 1 erhöht.

(b) Wir schaffen es auch, ein stets haltendes Programm zu schreiben, das alle ihm übergebenen Argumente ignoriert und einfach eine Konstante ausgibt.

(c) Ebenso problemlos können wir ein Programm schreiben, das von n übergebenen Argumenten das i-te ausgibt.

(d) Haben wir für die Funktionen $\psi, \chi_1, \ldots, \chi_m$ jeweils stets ein haltendes Programm, so können wir diese als Prozeduren verwenden und mit ihrer Hilfe nacheinander $\chi_1(x_1, \ldots, x_n)$, $\chi_2(x_1, \ldots, x_n), \ldots, \chi_m(x_1, \ldots, x_n)$ ausrechnen. Diese Werte übergeben wir der Prozedur, die ψ berechnet. Damit liefert unser komplettes Programm also das Resultat der Einsetzung von $\chi_1, \ldots, \chi_m$ in ψ. Da die einzelnen Prozeduren stets halten, wird auch unser Gesamtprogramm stets nach endlicher Zeit terminieren.

(e) Wir betrachten das Induktionsprinzip. Haben wir stets haltende Prozeduren für χ und gegebenenfalls ψ, so könnten wir mittels einer Schleife beginnend bei $\varphi(0)$ bzw. $\varphi(0, x_2, \ldots, x_n)$ die Werte für $\varphi(1), \varphi(2), \ldots, \varphi(x)$ bzw. $\varphi(1, x_2, \ldots, x_n), \varphi(2, x_2, \ldots, x_n), \ldots, \varphi(x_1, x_2, \ldots, x_n)$ ermitteln, indem wir die Prozedur für χ jeweils aufrufen. Da diese Prozedur stets hält, dauert eine Schleifeniteration nur endliche Zeit. Da weiter die Zahl der Iterationen mit x bzw. x_1 begrenzt ist, wird auch das gesamte Programm stets nach endlicher Zeit anhalten.

Da eine primitiv rekursive Funktion durch endlich viele Anwendungen dieser Konstruktionsschritte beschrieben werden kann, existiert für jede primitiv rekursive Funktion ein stets haltendes Programm. Dann existiert folglich auch eine stets terminierende Turingmaschine, die die Funktion berechnet. Wir haben mit diesen Überlegungen den folgenden Satz bewiesen.

Satz 12.15 *Jede primitiv rekursive Funktion ist berechenbar. Jedes Prädikat mit primitiv rekursiver Indikatorfunktion ist entscheidbar.*

12.2.2 Beispiele primitiv rekursiver Funktionen und Prädikate

Wir geben jetzt eine Reihe von Beispielen primitiv rekursiver Funktionen an.

Lemma 12.16 *Die Addition* $+ : \mathbb{N}^2 \to \mathbb{N}$ *mit* $+(x_1, x_2) = x_1 + x_2$ *ist primitiv rekursiv.*

Beweis: $+(x_1, x_2)$ läßt sich wie folgt beschreiben.

$$\begin{aligned} +(0, x_2) &= P_1^1(x_2) \\ +(S(x_1), x_2) &= S(P_2^3(x_1, +(x_1, x_2), x_2)). \end{aligned}$$

Wir benutzen also das Induktionsschema für mehrere Argumente. Die erste Zeile ist sofort einsichtig. Es gilt $0 + x_2 = x_2$ und die Ausgangsfunktion ist in diesem Fall also P_1^1.

Weiter gilt $S(x_1)+x_2 = S(x_1+x_2)$. Zur Berechnung von $+(S(x_1), x_2)$ stehen uns die Werte x_1 sowie $+(x_1, x_2)$ und x_2 zur Verfügung. Als Resultat brauchen wir den Nachfolger des zweiten Argumentes. Aber $P_2^3(x_1, +(x_1, x_2), x_2)$ liefert das zweite Aegument. Das Ergebnis setzen wir in die Nachfolgerfunktion S ein und erhalten das gewünschte Ergebnis.

$+(x_1, x_2)$ ist also durch eine Induktion und eine Einsetzung aus den Grundfunktionen konstruierbar und damit primitiv rekursiv. ■

Da wir die Addition jetzt als primitiv rekursiv nachgewiesen haben, verwenden wir nun zur besseren Lesbarkeit wieder die Schreibweise $a+b$ statt $+(a, b)$. Man behalte aber im Gedächtnis, dass $+$ eine gewöhnliche Funktion von $\mathbb{N}^2$ nach $\mathbb{N}$ ist.

Nun können wir $+$ natürlich auch zur Beschreibung weiterer Funktionen heranziehen. Als erstes nehmen wir uns die Multiplikation vor.

Lemma 12.17 *Die Multiplikation* $\cdot : \mathbb{N}^2 \to \mathbb{N}$ *mit* $\cdot(x_1, x_2) = x_1 \cdot x_2$ *ist primitiv rekursiv.*

Beweis: $\cdot(x_1, x_2)$ läßt sich wie folgt beschreiben.

$$\begin{aligned} \cdot(0, x_2) &= C_0^1(x_2) \\ \cdot(S(x_1), x_2) &= +(P_2^3, P_3^3)(x_1, \cdot(x_1, x_2), x_2). \end{aligned}$$

Im Basisfall wird festgehalten, dass die Multiplikation von 0 identisch mit der konstanten Nullfunktion ist.

Beachte, dass $S(x_1) \cdot x_2 = (x_1 \cdot x_2) + x_2$ und diese Identität wird in der induktiven Definition ausgenutzt. ■

Nun, da wir die Multiplikation haben, können wir uns auch das Potenzieren sichern.

Lemma 12.18 *Die Potenzierung* $exp : \mathbb{N}^2 \to \mathbb{N}$ *mit* $exp(x_1, x_2) = x_1^{x_2}$ *ist primitiv rekursiv.*

Aufgabe 96
Beweise Lemma 12.18.

Das Induktionsschema für ein Argument können wir verwenden, um die Fakultät als primitiv rekursiv nachzuweisen.

Lemma 12.19 *Die Fakultät* $! : \mathbb{N} \to \mathbb{N}$ *mit* $!(x) = x!$ *ist primitiv rekursiv.*

Beweis: Wir behaupten, dass sich $!(x_1)$ wie folgt beschreiben läßt.

$$\begin{array}{rcl} !(0) & = & 1 \\ !(S(x_1)) & = & \cdot(S(P_1^2), P_2^2))(x_1, !(x_1)). \end{array}$$

Die erste Zeile definiert $0! = 1$. In der zweiten wird $S(x_1)!$ als Produkt des Nachfolgers von x_1 und $x_1!$ dargestellt. Also $S(x_1)! = S(x_1) \cdot x_1!$. Das deckt sich mit der Definition der Fakultät. ■

Mit dem gleichen Schema erhalten wir auch die Vorgängerfunktion *pd* (vom englischen *predecessor*).

Lemma 12.20 *Die Vorgängerfunktion* $pd : \mathbb{N} \to \mathbb{N}$ *mit* $pd(x) = x - 1$ *für* $x \geq 1$ *und* $pd(0) = 0$ *ist primitiv rekursiv.*

Beweis: Wir behaupten, dass $pd(x_1)$ sich wie folgt beschreiben läßt.

$$\begin{array}{rcl} pd(0) & = & 0 \\ pd(S(x_1)) & = & P_1^2(x_1, pd(x_1)). \end{array}$$

Die Richtigkeit dieses Ansatzes ist offensichtlich. ■

Da uns die Nachfolgerfunktion zum Addieren geführt hat, sollten wir doch nun auch zum Subtrahieren gelangen können. Wir betrachten eine modifizierte Differenz $\dot{-}$ mit

$$x_1 \dot{-} x_2 = \begin{cases} x_1 - x_2 & \text{für } x_1 \geq x_2 \\ 0 & \text{sonst.} \end{cases}$$

Lemma 12.21 *Die modifizierte Differenz* $\dot{-} : \mathbb{N}^2 \to \mathbb{N}$ *ist primitiv rekursiv.*

Beweis: Natürlich wollen wir ausnutzen, dass $(a \dot{-} S(b)) = pd(a \dot{-} b)$ ist. Das heißt wir, möchten induktiv mit der zweiten Variablen argumentieren. Das Schema der primitiven Rekursion erlaubt uns aber nur eine Induktion auf der ersten Unbekannten. Wir helfen uns, indem wir eine Hilfsfunktion $\ddot{-}$ einführen, die $a \ddot{-} b = b \dot{-} a$ erfüllt.

$\ddot{-}(x_1, x_2)$ läßt sich wie folgt beschreiben.

$$\begin{array}{rcl} \ddot{-}(0, x_2) & = & P_1^1(x_2) \\ \ddot{-}(S(x_1), x_2) & = & pd(P_2^3(x_1, \ddot{-}(x_1, x_2), x_2)). \end{array}$$

Nun müssen wir lediglich noch unser eigentliches Ziel erreichen, nämlich $\dot{-}$ durch $\ddot{-}$ auszudrücken. Dies geschieht jedoch leicht mit:

$$\dot{-}(x_1, x_2) = \ddot{-}(P_2^2, P_1^2)(x_1, x_2).$$

■

Die modifizierte Differenz erlaubt es uns, das Minimum von zwei Elementen zu bestimmen.

Lemma 12.22 *Die Funktion* $min(x_1, x_2) : \mathbb{N}^2 \to \mathbb{N}$ *mit*

$$min(x_1, x_2) = \begin{cases} x_2 & \text{für } x_1 \geq x_2 \\ x_1 & \text{sonst} \end{cases}$$

ist primitiv rekursiv.

Beweis: Wir können das Minimum mit der modifizierten Differenz ausdrücken, denn

$$min(x_1, x_2) = x_2 \dot{-} (x_2 \dot{-} x_1).$$

Falls $x_1 \geq x_2$ ist, so ergibt $(x_2 \dot{-} x_1)$ Null und wir erhalten x_2, das richtige Resultat. Ist $x_2 > x_1$, so ziehen wir von x_2 genau den Wert ab, um den es größer als x_1 ist. Wir erhalten also x_1, das richtige Resultat. ■

Jetzt, da wir den Baustein $min(x_1, x_2)$ haben, können wir leicht für jede beliebige Argumentzahl n die Minimumsfunktion $min(x_1, \ldots, x_n)$ bilden.

Lemma 12.23 *Jede Minimumsfunktion* $min_n(x_1, \ldots, x_n) : \mathbb{N}^n \to \mathbb{N}$ *ist primitiv rekursiv.*

Beweis: Wir setzen einfach immer ein noch nicht betrachtetes Argument und das bisherige Minimum in die zweistellige Minimumsfunktion ein.

$$min(x_1, \ldots, x_n) = min(x_1, min(x_2, min(x_3, \ldots, min(x_{n-2}, (min(x_{n-1}, x_n) \ldots)).$$

Da n fest ist, ist diese Darstellung endlich lang. ■

Auch die folgenden drei Funktionen sind primitiv rekursiv.

Lemma 12.24 *Die Funktion* $max(x_1, x_2) : \mathbb{N}^2 \to \mathbb{N}$ *mit*

$$max(x_1, x_2) = \begin{cases} x_1 & \text{für } x_1 \geq x_2 \\ x_2 & \text{sonst} \end{cases}$$

ist primitiv rekursiv.

Auch hier kann wieder leicht auf Maximumsfunktionen höherer (fester) Ordnung verallgemeinert werden.

Lemma 12.25 *Die Signum-Funktion* $sg(x_1) : \mathbb{N} \to \{0, 1\}$ *mit*

$$sg(x_1) = \begin{cases} 0 & \text{für } x_1 = 0 \\ 1 & \text{sonst} \end{cases}$$

ist primitiv rekursiv.

Lemma 12.26 *Die Differenz* $|x_1 - x_2| : \mathbb{N}^2 \to \mathbb{N}$ *ist primitiv rekursiv.*

Aufgabe 97
Beweise Lemma 12.24, 12.25 und 12.26.

Die ganzzahlige Division erschließen wir uns mit den Funktionen *div* und *mod*. Sie liefern den ganzzahligen Anteil des Quotienten bzw. den Rest bei der Division. Auch diese Funktionen sind primitiv rekursiv.

Lemma 12.27 *Die Funktion* $mod(x_1, x_2) : \mathbb{N}^2 \to \mathbb{N}$ *mit*

$$x_1 \, mod \, x_2 = x_1 - x_2 \cdot (x_1 \, div \, x_2)$$

ist primitiv rekursiv.

Beweis: Wir können induktiv vorgehen.

$$\begin{array}{rcl} mod(0, x_2) & = & 0 \\ mod(S(x_1), x_2) & = & S(mod(x_1, x_2)) \cdot sg(x_2 \dot{-} (S(mod(x_1, x_2))). \end{array}$$

Warum ist das richtig? Der Wert von $(x_1 \, mod \, x_2)$ durchläuft die Zahlen von 0 bis $x_2 - 1$. Wir haben zwei Fälle zu betrachten. Die Signumsfunktion

$$sg(x_2 \dot{-} (S(mod(x_1, x_2)))$$

ist entweder 0 oder 1. Sie ist genau dann 0, wenn $x_2 \dot{-} S(mod(x_1, x_2)) = 0$, also wenn $(x_1 \, mod \, x_2) = x_2 - 1$ ist. In diesem Fall ist $S(x_1)$ ein Vielfaches von x_2 und richtigerweise wird $(S(x_1) \, mod \, x_2)$ auf 0 gesetzt.

Tritt dieser Fall jedoch nicht ein, so liefert die Signumsfunktion stets eine 1 und wir erhalten das korrekte Ergebnis $S(x_1 \, mod \, x_2)$ übrig. ■

Lemma 12.28 *Die Funktion* $div(x_1, x_2) : \mathbb{N}^2 \to \mathbb{N}$ *mit*

$$x_1 \, div \, x_2 = \frac{x_1 - (x_1 \, mod \, x_2)}{x_2}$$

ist primitiv rekursiv.

Aufgabe 98
Beweise Lemma 12.28.

Wir wenden uns nun noch einigen Prädikaten, sprich Eigenschaften, zu. Wir haben gesagt, dass ein Prädikat genau dann primitiv rekursiv ist, wenn es eine primitiv rekursive Indikatorfunktion besitzt. Hier also die wichtigsten Prädikate von natürlichen Zahlen und Tupeln natürlicher Zahlen. Wir beschränken uns hier darauf, die betreffenden Indikatorfunktionen anzugeben. Die Verifikation ist im allgemeinen leicht.

Lemma 12.29 *Die Prädikate*

gerade, ungerade, $<, =, >, \neq, \leq, \geq$, ist Vielfaches von, ist Teiler von

sind primitiv rekursiv.

Beweis:

Prädikat	**Indikatorfunktion**
a gerade	$1\dot{-}(a \, mod \, 2)$
a ungerade	$a \, mod \, 2$
$a < b$	$sg(b\dot{-}a)$
$a = b$	$1\dot{-}\lvert a - b\rvert$
$a > b$	$sg(a\dot{-}b)$
$a \neq b$	$sg(\lvert a - b\rvert)$
$a \leq b$	$sg(S(b)\dot{-}a)$
$a \geq b$	$sg(S(a)\dot{-}b)$
a ist Vielfaches von b	$1\dot{-}(a \, mod \, b)$
a ist Teiler von b	$1\dot{-}(b \, mod \, a)$

■

Wir erhöhen nun die Menge der uns bekannten primitiv rekursiven Prädikate drastisch, indem wir folgende Abgeschlossenheitseigenschaft feststellen.

Satz 12.30 *Sind A und B primitiv rekursive Prädikate, so sind auch die Prädikate*

$$\neg A, \; A \vee B, \; A \wedge B, \; A \leftrightarrow B, \; A \rightarrow B, \; A \text{ XOR } B, \; A \text{ NOR } B, \; A \text{ NAND } B$$

primitiv rekursiv.

Beweis: Da A und B primitiv rekursiv sind, gibt es Indikatorfunktionen f_A und f_B mit Wertebereich {0,1} und $f_A(\vec{x}) = 1 \leftrightarrow A(\vec{x})$ sowie $f_B(\vec{x}) = 1 \leftrightarrow B(\vec{x})$. Daraus lassen sich Indikatorfunktionen für die zusammengesetzten Prädikate bilden.

Prädikat	**Indikatorfunktion**
$\neg A$	$1\dot{-}f_A$
$A \vee B$	$sg(f_A + f_B)$
$A \wedge B$	$f_A \cdot f_b$
$A \leftrightarrow B$	$1\dot{-}\lvert f_A - f_B\rvert$
$A \rightarrow B$	$1\dot{-}(f_A \cdot (1\dot{-}f_B))$
$A \, XOR \, B$	$(f_A + f_B) \, mod \, 2$
$A \, NOR \, B$	$1 \cdot -(f_A + f_B)$
$A \, NAND \, B$	$sg(2\dot{-}(f_A + f_B)$

■

Wie erschließen uns nun weitere Hilfsmittel, indem wir endliche Summen und Produkte betrachten und das hier Beobachtete auf beschränkte Quantoren übertragen.

Definition 12.31 *Sei $\psi : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ eine primitiv rekursive Funktion. Dann nennen wir*

$$\psi_\Sigma(x_1, \ldots, x_n, z) := \sum_{y<z} \psi(x_1, \ldots, x_n, y)$$

die endliche Summe über ψ.

Lemma 12.32 *Ist $\psi : \mathbb{N}^n \rightarrow \mathbb{N}$ eine primitiv rekursive Funktion, dann ist auch die endliche Summe über ψ primitiv rekursiv.*

Beweis: Wir verwenden hier eine Induktion über die obere Schranke z. Man erinnere sich, dass man durch das Vertauschen der Reihenfolge der Argumente (siehe $\dot{-}$) jedes beliebige Argument zum Induktionsargument machen kann. Die Summe läßt sich in „laxer“ Schreibweise so aufschreiben :

$$\begin{aligned}\psi_\Sigma(x_1,\ldots,x_n,0) &= 0\\ \psi_\Sigma(x_1,\ldots,x_n,S(z)) &= \psi_\Sigma(x_1,\ldots,x_n,z)+\psi(x_1,\ldots,x_n,S(z)).\end{aligned}$$

■

Summen mit anders angegebenen Grenzen lassen sich aus dieser mit elementaren Umformungen herleiten.

$$\begin{aligned}\textstyle\sum_{y\le z}\psi(x_1,\ldots,x_n,y) &= \textstyle\sum_{y<S(z)}\psi(x_1,\ldots,x_n,y),\\ \textstyle\sum_{w<y<z}\psi(x_1,\ldots,x_n,y) &= \textstyle\sum_{y<z\dot{-}S(w)}\psi(x_1,\ldots,x_n,(y+S(w))),\\ \textstyle\sum_{w\le y\le z}\psi(x_1,\ldots,x_n,y) &= \textstyle\sum_{y<S(z)\dot{-}w}\psi(x_1,\ldots,x_n,(y+w)).\end{aligned}$$

Die Überlegungen für endliche Produkte folgen analog. Allerdings wird das *leere Produkt* $\prod_{y<0}$ auf 1 gesetzt.

Wir haben nun das nötige Rüstzeug zusammen, um unsere primitiv rekursiven Prädikate um endliche Quantoren zu bereichern.

Lemma 12.33 *Sei A ein n-stelliges primitiv rekursives Prädikat. Dann sind auch die beschränkte Existenzialisierung*

$$\exists_{y<z}A(x_1,\ldots,x_{n-1},y)$$

und die beschränkte Generalisierung

$$\forall_{y<z}A(x_1,\ldots,x_{n-1},y)$$

primitiv rekursive Prädikate.

Beweis: Wir geben primitiv rekursive Indikatorfunktionen an. Sei $f_A:\mathbb{N}^n\to\{0,1\}$ die Indikatorfunktion des Prädikats A. f_A ist primitiv rekursiv, da A primitiv rekursiv ist.

Prädikat	**Indikatorfunktion**
$\exists_{y<z}A(x_1,\ldots,x_{n-1},y)$	$sg(\sum_{y<z}f_A(x_1,\ldots,x_{n-1},y)$
$\forall_{y<z}A(x_1,\ldots,x_{n-1},y)$	$\prod_{y<z}f_A(x_1,\ldots,x_{n-1},y)$

■

Lemma 12.34 *Sei A ein n-stelliges primitiv rekursives Prädikat. Dann ist auch die beschränkte eindeutige Existenzialisierung*

$$\exists!_{y<z}A(x_1,\ldots,x_{n-1},y)$$

(In Worten: „Es gibt genau ein $y<z$ mit $A(x_1,\ldots,x_{n-1},y)$“*)*

primitiv rekursiv.

Aufgabe 99
Beweise Lemma 12.34.

Auch hier können in anderer Form gegebene Grenzen (etwa $\exists_{w<y\leq z}$) analog zum Vorgehen bei den Summen beschrieben werden.

Diese beschränkten Existenz- und Allaussagen erlauben uns nun bereits, sehr komplexe Eigenschaften von natürlichen Zahlen und Zahlentupeln als primitiv rekursive Prädikate darzustellen. Wir nennen gleich ein paar Beispiele. Selbstverständlich ist dies nur eine kleine Auswahl.

Lemma 12.35 *Die folgenden Prädikate sind primitiv rekursiv:*

Prädikat	**Darstellung**
a ist prim	$\forall_{2\leq y<a}\neg(y \textit{ ist Teiler von } a)$
a und b sind relativ prim	$\neg\exists_{2\leq y\leq min(a,b)}(y \textit{ ist Teiler von } a) \wedge (y \textit{ ist Teiler von } b)$
a ist eine Quadratzahl	$\exists_{y\leq a}(y \cdot y = a)$
a ist Carmichael-Zahl	$\neg(a \textit{ prim}) \wedge \forall_{2\leq p<a}(p \textit{ prim}) \rightarrow (a^{p-1} \bmod p = 1)$

Beweis: Die Darstellungen der Prädikate sind endlich und bestehen ihrerseits nur aus primitiv rekursiven Funktionen und Prädikaten. ■

Wir sollten auch erklären, warum wir hier lediglich Funktionen und Prädikate auf natürlichen Zahlen betrachten. Bei Turingmaschinen haben wir stets ein Eingabewort, welches aus den Elementen eines endlichen Eingabealphabets zusammengesetzt ist. Wir weisen nun nach, dass wir durch die Einschränkung auf natürliche Zahlen nichts verlieren.

Sei $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ das Eingabealphabet. Ein endlich langes Wort $a := \sigma_{i_0}\sigma_{i_1} \ldots \sigma_{i_{m-1}}$ kann dann als (n+1)-äre Darstellung einer natürlichen Zahl interpretiert werden.

$$Zahl(a) = \sum_{k=0}^{m-1} i_k \cdot (n+1)^k.$$

12.2.3 Der beschränkte μ-Operator

Im letzten Kapitel haben wir von einigen Funktionen nachgewiesen, dass sie primitiv rekursiv sind. Da alle primitiv rekursiven Funktionen von stets haltenden Turingmaschinen in endlicher Zeit berechnet werden können, sind primitiv rekursive Funktionen zwangsläufig berechenbar. In diesem Kapitel untersuchen wir, ob die Begriffe *primitiv rekursiv* und *berechenbar* deckungsgleich sind.

Existenz- bzw. Allquantoren können wir nur dann als primitiv rekursiv nachweisen, wenn eine endliche obere Schranke angegeben wird, wenn also das Prädikat nur über einem Intervall mit endlich vielen Kandidaten quantifiziert ist. Was passiert mit unbeschränkten Quantoren?

Beispielsweise ist es bis heute eine offene Frage, ob die sogenannte *Goldbachsche Vermutung* gilt.

Goldbachsche Vermutung : Jede natürliche gerade Zahl größer 2 ist die Summe zweier Primzahlen.

$$\forall_{a>2}(a \text{ gerade}) \rightarrow (\exists_{x<a}\, x \text{ prim} \wedge (a \dot{-} x) \text{ prim}).$$

Der Existenzquantor ist durch a beschränkt. Auch die Prädikate *prim, gerade*,die Funktion $\dot{-}$, und die Implikation sind primitiv rekursiv. Allein der Allquantor entzieht sich einer primitiv rekursiven Darstellung.

Wir führen nun zunächst den beschränkten μ-Operator ein.

Definition 12.36 *Sei $A(x_1, \ldots, x_{n-1}, y)$ ein primitiv rekursives Prädikat. Dann wird der beschränkte μ-Operator*

$$\mu_{y<z} A(x_1, \ldots, x_{n-1}, y)$$

durch

$$\begin{cases} min\{y < z | A(x_1, \ldots, x_{n-1}, y)\} & \textit{falls ein solches } y \textit{ existiert} \\ z & \textit{sonst} \end{cases}$$

definiert.

Falls es also im Intervall von 0 bis z (ausschließlich) Werte für y gibt, die das Prädikat erfüllen, dann liefert der μ-Operator den kleinsten dieser Werte zurück. Existiert kein Wert, der A erfüllt, so wird z als Resultat zurückgegeben.

Die folgende Tabelle soll das verdeutlichen. Wir betrachten ein fiktives Prädikat A.

y	0	1	2	3	4	5	6	7
$A(\vec{x}, y)$	nein	nein	nein	nein	ja	nein	ja	ja

Der μ-Operator $\mu_{y<8} A(x_1, \ldots, x_{n-1}, y)$ liefert hier also das Resultat 4, da 4 der kleinste y-Wert ist, der das Prädikat A erfüllt.

Berechnen wir $\mu_{y<3} A(x_1, \ldots, x_{n-1}, y)$, ist das Resultat die Obergrenze 3, da kein $y < 3$ das Prädikat erfüllt.

Lemma 12.37 *Der beschränkte μ-Operator ist primitiv rekursiv.*

Beweis: Wir bauen die Funktion schrittweise auf und überzeugen uns am obigen Beispiel, dass die Konstruktion ihren Zweck erfüllt. Zunächst besitzt $A(x_1, \ldots, x_n, y)$ als primitiv rekursives Prädikat eine primitiv rekursive Indikatorfunktion $f_A(x_1, \ldots, x_n, y)$. Sie ist unser Ausgangspunkt. Am Beispiel:

y	0	1	2	3	4	5	6	7
$A(\vec{x}, y)$	nein	nein	nein	nein	ja	nein	ja	ja
$f_A(\vec{x}, y)$	0	0	0	0	1	0	1	1

Diese Indikatorfunktion negieren wir zunächst, indem wir eine Hilfsfunktion $\alpha(\vec{x}, y) := 1 \dot{-} f_A(\vec{x}, y)$ einführen.

y	0	1	2	3	4	5	6	7
$A(\vec{x}, y)$	nein	nein	nein	nein	ja	nein	ja	ja
$f_A(\vec{x}, y)$	0	0	0	0	1	0	1	1
$\alpha(\vec{x}, y)$	1	1	1	1	0	1	0	0

Nun berechnen für alle y-Werte die endlichen Produkte der α-Funktion bis einschließlich y. Diese Hilfsfunktion nennen wir $\pi(\vec{x}, y) = \prod_{w \leq y} \alpha(\vec{x}, w)$. Die Werte von π sind folglich solange 1, bis erstmalig das Prädikat A wahr ist. Ab dann folgen nur noch Nullen.

y	0	1	2	3	4	5	6	7
$A(\vec{x}, y)$	nein	nein	nein	nein	ja	nein	ja	ja
$f_A(\vec{x}, y)$	0	0	0	0	1	0	1	1
$\alpha(\vec{x}, y)$	1	1	1	1	0	1	0	0
$\pi(\vec{x}, y)$	1	1	1	1	0	0	0	0

Den gesuchten Wert erhalten wir nun, wenn wir die Funktion π aufsummieren. Mit $\mu_{y<z}A(\vec{x},y) = \sum_{y<z}\pi(\vec{x},y)$ sind wir am Ziel.

y	0	1	2	3	4	5	6	7
$A(\vec{x},y)$	nein	nein	nein	nein	ja	nein	ja	ja
$f_A(\vec{x},y)$	0	0	0	0	1	0	1	1
$\alpha(\vec{x},y)$	1	1	1	1	0	1	0	0
$\pi(\vec{x},y)$	1	1	1	1	0	0	0	0
$\sum_{w<y}\pi(\vec{x},w)$	0	1	2	3	4	4	4	4

Man überzeuge sich, dass diese Werte genau der Definition des beschränkten μ-Operators entsprechen. ■

Der μ-Operator kann nun beispielsweise dazu verwendet werden, die $(n+1)$-te Primzahl primitiv rekursiv zu berechnen. Wir zeigen dies in dem folgenden wichtigen Lemma.

Lemma 12.38 *Die Funktion $Pr(x) : \mathbb{N} \to \mathbb{N}$ mit*

$$Pr(x) = y \leftrightarrow y \text{ ist die } (x+1)\text{-te Primzahl}$$

ist primitiv rekursiv.

Beweis: Wir gehen induktiv vor.

$$\begin{aligned} Pr(0) &= 2 \\ Pr(S(x)) &= \mu_{y \le S((Pr(x))!)}\,((\text{y prim}) \wedge (y > Pr(x))). \end{aligned}$$

Die Verankerung ist sofort einsichtig. $Pr(0)$ ist 2, da 2 die erste Primzahl ist. Im Induktionsschritt suchen wir die kleinste Zahl, die prim ist und größer ist als die zuletzt gefundene. Das ist dann gerade die nächste Primzahl. Die obere Schranke $S((Pr(x))!)$ folgt aus Euklids klassischem Beweis für die Existenz unendlich vieler Primzahlen. ■

Warum ist diese Funktion nun von besonderer Bedeutung? Das liegt daran, dass wir mit ihr und den schon bekannten primitiv rekursiven Funktionen (insbesondere *div* und *mod*) Primfaktorzerlegungen durchführen können. Mit Hilfe der Primfaktorzerlegung können wir dann endlich lange Folgen natürlicher Zahlen durch natürliche Zahlen kodieren: Sei $a_1, \ldots, a_m$ eine Folge natürlicher Zahlen. Dieser Folge ordnen wir die Zahl

$$\prod_{i=0}^{m-1} p_i^{S(a_1)} = 2^{S(a_1)} \cdot 3^{S(a_2)} \cdot 5^{S(a_3)} \cdot \ldots \cdot p_m^{S(a_m)}$$

zu. (Warum ist es nötig die Exponenten um 1 zu erhöhen?) Da jede Zahl eine eindeutige Primfaktorzerlegung besitzt, verlieren wir keine Informationen, wenn wir anstelle der Folge nun von der die Folge repräsentierenden Zahl sprechen. Bei Bedarf können wir jederzeit die Folge aus der Zahl mittels Primfaktorzerlegung reproduzieren.

Beispiel 12.2 Wir betrachten die Folge: $10, 4, 5, 2, 7, 3, 2, 1, 0, 0, 1$. Sie entspricht der Zahl

$$2^{11} \cdot 3^5 \cdot 5^6 \cdot 7^3 \cdot 11^8 \cdot 13^4 \cdot 17^3 \cdot 19^2 \cdot 23^1 \cdot 29^1 \cdot 31^2$$

und damit der Zahl

$$2048 \cdot 243 \cdot 15625 \cdot 343 \cdot 214358881 \cdot 28561 \cdot 4913 \cdot 361 \cdot 23 \cdot 29 \cdot 961.$$

Dies ergibt, bitte nachrechnen,

$$185638670771581029106339024870080000000.$$

Diese Zahl ist vielleicht etwas groß, aber sie repräsentiert eindeutig unsere endliche Zahlenfolge. Es gibt nicht mehr endliche Folgen aus natürlichen Zahlen als natürliche Zahlen!

Dieses Resultat stößt das Tor auf zu einer Flut von weiteren mächtigen primitiv rekursiven Funktionen und Konstruktionsprinzipien.

- Werteverlaufsinduktionen sind primitiv rekursiv. Was ist das? Wir haben uns bei Induktionen bisher auf solche Fälle beschränkt, in denen ein Wert $f(x)$ jeweils vom Funktionswert des unmittelbaren Vorgängers $f(x-1)$ abhängt, nicht aber von weiteren schon berechneten Werten wie etwa $f(x-2)$ oder $f(x\,div\,2)$. Nun können wir aber eine Hilfsfunktion basteln, die die Folge aller bisher ermittelten Funktionswerte in eine solche repräsentierende Zahl quetscht. Aus diesem *einen* Funktionswert, können die gewünschten Werte extrahiert werden.
- Simultane Induktionen sind primitiv rekursiv. Unsere Induktionen durchlaufen jeweils nur ein Argument und zählen es herauf. Es ist auch möglich, Induktionen anzugeben, in denen zur Berechnung von $f(a,b)$ sowohl $f(a-1,b)$ und $f(a,b-1)$ als auch $f(a-1,b-1)$ vorliegen muss. Auch deren primitiv rekursive Behandlung ist nun möglich.
- Parallele Induktionen sind primitiv rekursiv. Dies sind solche Induktionen, bei denen zwei (oder mehr) Funktionen parallel zueinander berechnet werden müssen, weil etwa $f(x)$ von $f(x-1)$ und $g(x-1)$ abhängt, $g(x)$ seinerseits aber ebenso $f(x-1)$ und $g(x-1)$ benötigt.

12.2.4 Die Ackermann-Funktion

Dies alles sind bereits ausgesprochen mächtige Werkzeuge. Wir zeigen jedoch nun am Beispiel der Ackermann-Funktion, dass es Funktionen gibt, die zwar berechenbar, aber nicht primitiv rekursiv sind. Der Aufwand, den wir treiben müssen, um diesen Nachweis zu führen, mag als Indiz dafür herhalten, wie weit primitive Rekursion doch reicht.

Definition 12.39 *Die Ackermann-Funktion:* $a : \mathbb{N}^2 \to \mathbb{N}$ *wird durch*

$$\begin{aligned} a(0,y) &= y+1 \\ a(x+1,0) &= a(x,1) \\ a(x+1,y+1) &= a(x,a(x+1,y)) \end{aligned}$$

definiert.

Zunächst müssen wir einige technische Lemmata beweisen, die wir später verwenden werden.

Lemma 12.40 *Die Ackermann-Funktion besitzt folgende Eigenschaften:*

a) $a(x,y) > x$.

b) $a(x,y) > y$.

c) Monotonie in x: $x_2 > x_1 \rightarrow a(x_2, y) \geq a(x_1, y)$.

d) $a(x, y+1) > a(x, y)$.

e) Monotonie in y: $y_2 > y_1 \rightarrow a(x, y_2) > a(x, y_1)$.

f) $a(x+1, y) \geq a(x, y+1)$.

g) $a(x+2, y) \geq a(x, 2y)$.

Beweis: b) Wir zeigen $a(x, y) > y$ durch Induktion nach x. I_x (bzw. I_y) bezeichnet eine Anwendung der Induktionshypothese für x (bzw. y).

$x = 0$: $\quad a(0, y) = y + 1 > y$

$x \rightarrow x+1$: Induktion nach y

$y = 0$: $\quad a(x+1, 0) = a(x, 1) \overset{I_x}{>} 1 > 0 = y$

$y \rightarrow y+1$:

$$\begin{aligned} a(x+1, y+1) &= a(x, a(x+1, y)) \\ &\overset{I_x}{\geq} a(x+1, y) + 1 \\ &\overset{I_y}{>} y + 1. \end{aligned}$$

■

Beweis: d) Wir zeigen $a(x, y+1) > a(x, y)$ durch Induktion nach x.

$x = 0$: $\quad a(0, y+1) = y + 2 > y + 1 = a(0, y)$

$x \rightarrow x+1$:

$$\begin{aligned} a(x+1, y+1) &= a(x, a(x+1, y)) \\ &\overset{b)}{>} a(x+1, y). \end{aligned}$$

■

Beweis: e) $y_2 > y_1 \rightarrow a(x, y_2) > a(x, y_1)$ ergibt sich durch Iteration von d). ■

Beweis: f) Wir zeigen $a(x+1, y) \geq a(x, y+1)$ durch Induktion nach y.

$y = 0$: 2 Fälle treten auf:

$x = 0$: $\quad a(1, 0) = a(0, 1)$.

$x > 0$: $\quad a(x, 0) = a(x-1, 1)$.

$y \rightarrow y+1$: Wiederum treten 2 Fälle auf:

$x = 0$: $\quad a(1, y+1) = a(0, a(1, y))$
wegen b) gilt $a(1, y) > y \geq y + 1$ und e) liefert
$a(1, y+1) \geq a(0, y+1) = y + 2 = a(x, y+1)$.

$x \rightarrow x+1$:

$$\begin{aligned} a(x+1, y+1) &= a(x, a(x+1, y)) \\ &\overset{I_y}{\geq} a(x, a(x, y+1)) \end{aligned}$$

wegen b) gilt $a(x, y+1) \geq y + 2$
und e) liefert $a(x+1, y+1) \geq a(x, y+2)$.

■

Beweis: c) $x_2 > x_1 \rightarrow a(x_2, y) \geq a(x_1, y)$ ist zu zeigen.

$$\begin{array}{rcl} a(x_2, y) & \overset{f)}{\geq} & a(x_2 - 1, y + 1) \overset{f)}{\geq} a(x_2 - 2, y + 2) \overset{f)}{\geq} \ldots \\ & \overset{f)}{\geq} & a(x_2 - (x_2 - x_1), y + (x_2 - x_1)) \\ & = & a(x_1, y + (x_2 - x_1)) \\ & \overset{e)}{\geq} & a(x_1, y). \end{array}$$

■

Beweis: a) $a(x, y) > x$ ist zu zeigen.

$$\begin{array}{rcl} a(x, y) & \overset{f)}{\geq} & a(x - 1, y + 1) \overset{f)}{\geq} a(x - 2, y + 2) \overset{f)}{\geq} \ldots \\ & \overset{f)}{\geq} & a(0, y + x) \\ & = & y + x + 1 \\ & > & x. \end{array}$$

■

Beweis: g) Wir zeigen $a(x + 2, y) \geq a(x, 2y)$ durch Induktion nach y.

$y = 0 :$ $\quad a(x + 2, 0) \overset{c)}{\geq} a(x, 0)$
$y \rightarrow y + 1 :$

$$\begin{array}{rcl} a(x + 2, y + 1) & = & a(x + 1, a(x + 2, y)) \overset{I_y}{\geq} a(x + 1, a(x, 2y)) \\ & \overset{f)}{\geq} & a(x, a(x, 2y) + 1) \\ & \geq & a(x, a(0, 2y) + 1) \\ & \geq & a(x, 2y + 1 + 1) = a(x, 2y + 2). \end{array}$$

■

Ausgerüstet mit diesen Eigenschaften können wir nun nachweisen, dass die Ackermann-Funktion nicht primitiv rekursiv ist. Wir zeigen, dass die Ackermann-Funktion stärker wächst, als jede primitiv rekursive Funktion.

Satz 12.41 *Die Ackermann-Funktion wächst schneller als jede primitiv rekursive Funktion.*

Formal: Sei $f : \mathbb{N}^n \rightarrow \mathbb{N}$ *primitiv rekursiv, dann gibt es einen* Tempoparameter k*, so dass*

$$\forall_{\vec{x} \in \mathbb{N}^n} f(\vec{x}) < a(k, MAX\{x_1, \ldots, x_n\})$$

Satz 12.42 *Die Ackermann-Funktion ist nicht primitiv rekursiv.*

Beweis: Diese Tatsache folgt sofort aus Satz 12.41. ■

Beweis: von Satz 12.41 Unser Beweis muss für alle primitiv rekursiven Funktionen gelten. Wir werden uns zunächst die Grundfunktionen näher ansehen und nachweisen, dass wir für jede von ihnen einen Tempoparameter k mit der im Satz geforderten Eigenschaft angeben können. Wir werden dann sehen, dass sich bei den beiden Konstruktionsprinzipien Induktion und Einsetzung neue Tempoparameter aus den Tempoparametern der zur Konstruktion herangezogenen Funktionen bestimmen lassen. Wir zeigen also, dass die Eigenschaft, irgendwo von der Ackermann-Funktion überholt zu werden, vererbt wird. Formal spricht man hier von einer Induktion über den Formelrang.

Für die Dauer dieses Beweises bezeichne ξ stets das maximale Argument der betrachteten Funktion: $\xi = MAX\{x_1, \ldots, x_n\}$.

Wir beginnen mit den *Grundfunktionen*.

$f = S$ Wir wählen den Tempoparameter $k = 1$ und erhalten

$$a(1, x_1) \geq a(0, x_1 + 1) > x_1 + 1 = S(x_1) = f(x_1).$$

$f = C_q^n$ Wir wählen $k = q$ und erhalten.

$$a(k, \xi) = a(q, \xi) > q = f(x_1, \ldots, x_n).$$

$f = P_i^n$ Hier leistet $k = 0$ das Verlangte:

$$a(k, \xi) = a(0, \xi) = \xi + 1 > \xi \geq f(x_1, \ldots, x_n).$$

Nun nehmen wir uns die *Einsetzung* vor. Von den dabei verwendeten Funktionen ψ und den χ_i wissen wir bereits, dass es für sie k-Werte gibt, die die im Satz geforderte Eigenschaft erfüllen.

Es ist also $f = \psi(\chi_1, \ldots, \chi_m)$. Wir wissen, dass für jedes i aus $\{1, \ldots, m\}$ einen Tempoparameter k_i mit

$$\forall_{\vec{x} \in \mathbb{N}^n} \chi_i(\vec{x}) < a(k_i, \xi)$$

gilt. Der Tempoparameter für ψ sei k_0, d.h. es ist

$$\forall_{\vec{x} \in \mathbb{N}^n} \psi(\vec{x}) < a(k_0, \xi).$$

Die Behauptung ist nun, dass $k = MAX\{k_0, k_1, \ldots, k_m\} + 2$ den Zweck erfüllt. Wir erhalten

$$a(k, \xi) \overset{f)}{\geq} a(k-1, \xi+1) \overset{Def.}{=} a(k-2, a(k-1, \xi)).$$

Definitionsgemäß ist $k - 1 > k_i$ für alle $i \in \{1, \ldots, m\}$. Deswegen gilt auch für alle $i \in \{1, \ldots, m\}$

$$a(k-1, \xi) \geq a(k_i, \xi) > \chi_i(x_1, \ldots, x_n)$$

und wir erhalten

$$\begin{aligned} a(k, \xi) &> a(k-2, a(k-1, \xi)) \\ &> a(k-2, MAX\{\chi_1(\vec{x}), \ldots, \chi_m(\vec{x})\}). \end{aligned}$$

Weiterhin gilt $k - 2 \geq k_0$ nach Definition von k, und wir können die obige Ungleichungskette weiterführen:

$$\begin{aligned} a(k, \xi) &> a(k-2, MAX\{\chi_1(\vec{x}), \ldots, \chi_m(\vec{x})\}) \\ &\geq a(k_0, MAX\{\chi_1(\vec{x}), \ldots, \chi_m(\vec{x})\}) \\ &> \psi(\chi_1(\vec{x}), \ldots, \chi_m(\vec{x})). \end{aligned}$$

Die letzte Abschätzung ergibt sich aus der Annahme über k_0.

Zuletzt müssen wir uns die *Induktion* näher ansehen.

$$f(x_1,\ldots,x_n) = \begin{cases} \psi(x_2,\ldots,x_n) & \text{falls } x_1 = 0 \\ \chi(x-1, f(x_1-1,\ldots,x_n), x_2,\ldots,x_n) & \text{sonst.} \end{cases}$$

Auch hier nehmen wir an, dass uns für ψ und χ bereits Werte k_ψ und k_χ mit der gesuchten Eigenschaft vorliegen, d.h.

$$\forall_{\vec{x}\in\mathbb{N}^{n-1}}\ \psi(\vec{x}) < a(k_\psi, MAX\{x_1,\ldots,x_{n-1}\}),$$

$$\forall_{\vec{x}\in\mathbb{N}^{n+1}}\ \chi(\vec{x}) < a(k_\chi, MAX\{x_1,\ldots,x_{n+1}\}).$$

Die Behauptung ist nun, dass der Ansatz $k = MAX\{k_\chi, k_\psi\}+3$ zum Ziel führt. Wir verwenden den Buchstaben ξ ab jetzt als Maximum der Elemente x_2 bis x_n. Wir zeigen zunächst die Eigenschaft:

$$(*) \qquad \forall_{x_1\in\mathbb{N}}\ (a(k-2, \xi + x_1) > f(\vec{x}).$$

Wir beweisen dies durch eine Induktion nach x_1.

$x_1 = 0$: Es ist $\begin{aligned} f(\vec{x}) = f(0, x_2,\ldots,x_n) &= \psi(x_2,\ldots,x_n) \\ &< a(k_\psi, \xi) \\ &< a(k-2,\xi) < \cdots < a(k-2, \xi + x_1). \end{aligned}$

$x_1 \to x_1 + 1$: Für den Ausdruck $a(k-2, \xi + x_1)$ erhalten wir die Abschätzungen

$$\begin{aligned} a(k-2,\xi+x_1) &> \xi + x_1 > \xi, x_1 \\ a(k-2,\xi+x_1) &> f(x_1, x_2,\ldots,x_n). \end{aligned}$$

Letzteres ist die Induktionsannahme. Wir fassen diese drei Ungleichungen zusammen und erhalten

$$a(k-2,\xi+x_1) > MAX\{x_1, f(x_1,x_2,\ldots,x_n), \xi\}.$$

Wir können nun die folgende Ungleichungskette aufstellen, mit der wir den Induktionsschluß abschliessen.

$$\begin{aligned} a(k-2,\xi+x_1+1) &\overset{Def.}{=} a(k-3, a(k-2,\xi+x_1)) \\ &\geq a(k_\chi, a(k-2,\xi+x_1)) \\ &> a(k_\chi, MAX\{x_1, f(x_1,x_2,\ldots,x_n), x_2,\ldots,x_n\}. \end{aligned}$$

Dieses Maximum bilden wir über die Argumente, auf die χ im Induktionsschritt bei der Berechnung von $f((x_1+1), x_2,\ldots,x_n)$ zugreift. Also können wir unsere Annahme über k_χ verwenden und wir erhalten

$$a(k-2,\xi+x_1+1) > f((x_1+1), x_2,\ldots,x_n).$$

Damit ist $(*)$ gezeigt. Wir verwenden dieses Ergebnis nun, um wie folgt zu schließen:

$$\begin{aligned} a(k, MAX\{x_1,\ldots,x_n\}) &= a(k, MAX\{x_1,\xi\}) \\ &\overset{g)}{\geq} a(k-2, 2\cdot MAX\{x_1,\xi\}) \\ &\geq a(k-2, x_1+\xi) \\ &\overset{*}{>} f(x_1,x_2,\ldots,x_n) \\ &= f(\vec{x}). \end{aligned}$$

Damit ist die Vererbungseigenschaft gezeigt. Da nach Definition nur solche Funktionen primitiv rekursiv sind, die durch iterierte Anwendung der beiden Schemata *Einsetzung* und *Induktion* aus den Grundfunktionen entstehen, haben wir für jede primitiv rekursive Funktionen einen Tempoparameter k gefunden, für den die Ackermann-Funktion dominiert. ■

Bestandsaufnahme: Wir haben gesehen, dass die Ackermann-Funktion nicht primitiv rekursiv ist. Andererseits können wir die rekursive Definition der Acker- mann-Funktion für ein Programm in einer höheren Programmiersprache fast direkt abschreiben:

```
int ackermann(int x, int y)
/* berechnet den Wert der Ackermann-Funktion */
{
        if(x == 0) return(y+1);
        else if(y == 0) return(ackermann(x-1,1));
             else return(ackermann(x-1,ackermann(x,y-1));
}
```

Aufgabe 100
Zeige, dass das Programm ackerman stets hält. Zeige ebenfalls, dass die Laufzeit dieses Programms alle primitiv rekursiven Schranken sprengt.

Selbst beliebig häufig geschachtelte Exponentialfunktionen stellen keine obere Laufzeitschranke dar. Berechnen wir doch mal ackermann(2,2) von Hand.

$$
\begin{aligned}
a(2,2) &= a(1,a(2,1)) \\
&= a(1,a(1,a(2,0))) \\
&= a(1,a(1,a(1,1))) \\
&= a(1,a(1,a(0,a(1,0)))) \\
&= a(1,a(1,a(0,a(0,1)))) \\
&= a(1,a(1,a(0,2))) \\
&= a(1,a(1,3)) \\
&= a(1,a(0,a(1,2))) \\
&= a(1,a(0,a(0,a(1,1)))) \\
&= a(1,a(0,a(0,a(0,a(1,0))))) \\
&= a(1,a(0,a(0,a(0,a(0,1))))) \\
&= a(1,a(0,a(0,a(0,2)))) \\
&= a(1,a(0,a(0,3))) \\
&= a(1,a(0,4)) \\
&= a(1,5) \\
&= a(0,a(1,4)) \\
&= a(0,a(0,a(1,3))) \\
&= a(0,a(0,a(0,a(1,2)))) \\
&= a(0,a(0,a(0,a(0,a(1,1))))) \\
&= a(0,a(0,a(0,a(0,a(0,a(1,0)))))) \\
&= a(0,a(0,a(0,a(0,a(0,a(0,1))))))
\end{aligned}
$$

$$
\begin{aligned}
&= a(0, a(0, a(0, a(0, a(0, 2)))))\\
&= a(0, a(0, a(0, a(0, 3))))\\
&= a(0, a(0, a(0, 4)))\\
&= a(0, a(0, 5))\\
&= a(0, 6)\\
&= 7
\end{aligned}
$$

12.2.5 Der unbeschränkte μ-Operator

Da unser Programm die Ackermann-Funktion berechnet, existiert also auch eine Turingmaschine, die sie berechnet und die Ackermann-Funktion ist berechenbar.

Mit unserem Konzept der primitiven Rekursion können wir also nicht alle berechenbaren Funktionen erfassen. Aber mit der Hinzunahme des unbeschränkten μ-Operators erhalten wir tatsächlich die Klasse der berechenbaren Funktionen.

Definition 12.43 *Sei $A(x_1, \ldots, x_{n-1}, y)$ ein primitiv rekursives Prädikat. Dann wird der unbeschränkte μ-Operator*

$$\mu A(x_1, \ldots, x_{n-1}, y)$$

durch

$$\begin{cases} min\{y \in \mathbb{N} | A(x_1, \ldots, x_{n-1}, y)\} & \text{falls ein solches } y \text{ existiert} \\ 0 & \text{sonst} \end{cases}$$

definiert.

Definition 12.44 *Die Funktionenklasse, die man aus den Grundfunktionen durch endliches Anwenden der*

Einsetzung, Induktion und des unbeschränkten μ-Operators

erhält, heisst die Klase der μ-rekursiven Funktionen.

Die Klasse der μ-rekursiven Funktionen und die Klasse der berechenbaren Funktionen stimmen überein!

Satz 12.45

(a) *Jede μ-rekursive Funktion ist berechenbar.*

(b) *Jede berechenbare Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist μ-rekursiv.*

Aufgabe 101
Beweise Satz 12.45.

Hinweis für Teil (b): Übersetze, ähnlich wie im Nachweis der NP-Vollständigkeit von *KNF–SAT*, die Berechnung einer Turingmaschine in die Sprache der Aussagenlogik.

12.3 Zusammenfassung

Wir haben den Begriff der *Berechenbarkeit von Funktionen* sowie den dazu äquivalenten Begriff für Probleme, nämlich die *Entscheidbarkeit* eingeführt. Wir haben gesagt, dass eine Funktion berechenbar (bzw. ein Problem entscheidbar ist), wenn sie durch eine stets haltende Turingmaschine berechnet (bzw. gelöst) werden kann.

Die *Church-Turing These*, nämlich, dass unsere Definition der Berechenbarkeit und Entscheidbarkeit den intuitiven Begriff der Berechenbarkeit und Entscheidbarkeit exakt widerspiegelt, wird untermauert durch verschiedene Simulationen: Wir haben (sogar parallele) Registermaschinen durch Turingmaschinen simuliert und haben gezeigt, dass auch probabilistische Turingmaschinen und Quanten-Turingmaschinen simuliert werden können. Weiterhin haben wir gezeigt, dass die Klasse der μ-rekursiven Funktionen mit der Klasse der berechenbaren Funktionen zusammenfällt. Man beachte aber, dass wir *keinen* Beweis für die Church-Turing These besitzen.

Wir haben weiterhin gesehen, dass die meisten Entscheidungsprobleme nicht entscheidbar sind. Mit der Diagonalisierungsmethode haben wir nachgewiesen, dass die Diagonalsprache unentscheidbar ist. Wir haben dann den Begriff einer Reduktion von Problem L_1 auf Problem L_2 eingeführt. Da sich die Diagonalsprache auf das Halteproblem, das spezielle Halteproblem und die universelle Sprache reduzieren läßt, haben wir die Unentscheidbarkeit dieser Probleme zeigen können. Weiterhin haben wir gesehen, dass jede nicht-triviale „Eigenschaft“ von Turingmaschinen (wie „nie zu halten“, „eine bestimmte, vorgegebene Funktion zu berechnen“) auf ein unentscheidbares Problem führt. Dies ist Inhalt des Satzes von Rice.

Was ist die Konsequenz dieser Unentscheidbarkeitsresultate? Eine Konsequenz ist, dass es unmöglich ist, korrekte Compiler zu schreiben, die auch nur eines der folgenden Probleme lösen:

- festzustellen, dass eine bestimmte Anweisung eines Programms je ausgeführt wird,
- nachzuprüfen, ob ein Programm für *jede* Eingabe hält,
- nachzuprüfen, ob ein Programm für eine *bestimmte* Eingabe hält.

Dabei ist es unwesentlich, ob wir „Programm“ durch Turingmaschinenprogramm, Pascal-Programm, C^{++}-Programm, ... übersetzen.

Wir haben das Kapitel abgeschlossen, indem wir den Begriff der rekursiven Aufzählbarkeit eingeführt haben. Beachte, dass jedes entscheidbare Problem rekursiv aufzählbar ist. Die Umkehrung gilt nicht, wie Halteproblem, spezielles Halteproblem und universelle Sprache zeigen.

Gibt es Entscheidungsprobleme, die nicht rekursiv aufzählbar sind? Ja, die Komplemente von rekursiv aufzählbaren, aber nicht entscheidbaren Problemen sind *nicht* rekursiv aufzählbar. Als Konsequenz dieser Eigenschaft haben wir den Gödelschen Unvollständigkeitssatz abgeleitet. Hier haben wir auch ausgenutzt, dass die Menge aller beweisbaren Aussagen, die sich von einem rekursiv aufzählbarem Axiomensystem ableiten lassen, selbst rekursiv aufzählbar ist.

Gibt es Entscheidungsprobleme, so dass weder das ursprüngliche Problem noch sein Komplement rekursiv aufzählbar ist? Auch dies ist der Fall. Als Beispiel sei das Problem

$$A = \{\langle M \rangle \mid M \text{ hält für alle Eingaben}\}$$

genannt.

Aufgabe 102
Zeige, dass $\overline{H_\varepsilon} \leq \overline{A}$ und, dass $\overline{H_\varepsilon} \leq A$.

Fazit: Weder A noch $\overline{A}$ sind rekursiv aufzählbar, denn sonst wäre ja $\overline{H_\varepsilon}$ rekursiv aufzählbar.

Es ist möglich, so etwas wie den „Schwierigkeitsgrad“ eines unentscheidbaren Problems anzugeben, nämlich seinen Turing-Grad. Diese und ähnliche Fragestellungen werden in der Rekursionstheorie verfolgt.

Teil V

Ausblick

Wir geben einen kurzen Überblick über weiterführende Veranstaltungen im Bachelor-Studiengang (Effiziente Algorithmen und Computational Learning Theory) und im Master-Studiengang (Approximationsalgorithmen, Internet Algorithmen, Komplexitätstheorie, Parallel and Distributed Algorithms).

Effiziente Algorithmen

Wir haben uns nur am Rande mit randomisierten, also Münzen-werfenden Algorithmen beschäftigt. Es stellt sich aber heraus, dass die Randomisierung nicht nur in vielen zahlentheoretischen, für die Kryptographie wichtigen Problemen (wie Primzahltests oder die Bestimmung von Quadratwurzeln in Körpern) mit Vorteil angewandt werden kann. In verschiedensten algorithmischen Fragestellungen wie minimalen Spannbäumen, Lastverteilungsproblemen oder in der Bestimmung nächstliegender Punkte im Zwei-Dimensionalen wie auch in der Entwicklung von Datenstrukturen führt die Randomisierung auf einfache und schnelle Algorithmen.

Ein weiteres, in Anwendungen wichtiges Gebiet ist die Entwicklung und Analyse von On-Line Algorithmen. Hier muss ein Algorithmus Entscheidungen treffen, ohne die Zukunft zu kennen. Ein Beispiel ist das Paging Problem: Ein Cache fasst nur eine begrenzte Anzahl von Seiten. Welche Seite soll ausgelagert werden, wenn eine neue Seite gespeichert werden muss? Diese Frage kann sicherlich nicht optimal beantwortet werden, weil die gerade ausgelagerte Seite vielleicht schon im nächsten Schritt wieder angefordert wird; aber tatsächlich gibt es sehr gute Heuristiken, die sogar fast mit Off-line Algorithmen, also mit Algorithmen, die die Zukunft kennen, mithalten können.

Approximationsalgorithmen

Wir haben allgemeine, leicht anwendbare Verfahren kennengelernt, um Optimierungsprobleme zu lösen. Diese Verfahren werden im Allgemeinen aber nur approximative Lösungen finden und auch die erreichbaren Approximationskonstanten werden nicht berauschend sein. Diese Situation ändert sich grundlegend, wenn gute problem-spezifische Heuristiken (oder Approximationsalgorithmen) vorliegen. Wir haben das wichtige Thema der Heuristiken aber nur oberflächlich untersuchen können. Zu den wichtigen, offen gebliebenen Fragestellungen gehören

- die systematische Untersuchung von Problemen, die mit Greedy-Algorithmen exakt lösbar sind,
- der Entwurf von Algorithmen für das lineare Programmieren sowie Anwendungen der linearen Programmierung,
- der Entwurf von Heuristiken für fundamentale Optimierungsprobleme wie etwa Scheduling Probleme, das Traveling Salesman Problem oder Clustering Probleme und
- die Bestimmung der durch effiziente Algorithmen erreichbaren Approximationsfaktoren. Diese Frage führt auf eine vollständig neue Sichtweise der Klasse `NP`.

Parallel and Distributed Algorithms

Das Gebiet der parallelen Algorithmen hat lange unter den langsamen Kommunikationszeiten in verteilten Systemen gelitten. Dieser Zustand beginnt sich zu ändern, da rein-sequentielle

Rechner technologisch nicht wesentlich weiter beschleunigt werden können. Die neuen Mehrprozessorkerne von AMD oder Intel belegen dies. Wie kann man aber, verteilt über mehrere Rechner oder aber mit mehreren CPUs, mit Vorteil gegenüber sequentiellen Rechnern rechnen?

Wenn wir zum Beispiel n Schlüssel mit einem Rechner sortieren möchten, dann gelingt uns dies in Zeit $O(n \cdot \log_2 n)$. Bestenfalls wird man bei k Rechnern die Laufzeit $O(\frac{n \cdot \log_2 n}{k})$ erwarten können, aber sind solche optimalen Beschleunigungen auch erreichbar? Die Antwort ist tatsächlich positiv und derartige Beschleunigungen sind auch in vielen Fragestellungen des Scientific Computing wie zum Beispiel dem Lösen von Gleichungssystemen erreichbar.

Welche Probleme sind parallelisierbar, lassen also substantielle Beschleunigungen gegenüber sequentiellen Algorithmen zu? Gibt es algorithmische Probleme, die inhärent sequentiell sind? Diese Fragen werden mit der P-Vollständigkeit, einer Variante der NP-Vollständigkeit beantwortet.

Internet Algorithmen

Das Internet präsentiert völlig neue Herausforderungen aufgrund der immensen Datenmengen, die schnell bewältigt werden müssen:

- Wie sollten Datenpakete befördert werden, so dass diese Pakete schnell an ihr Ziel geleitet werden? Nach welchen Prinzipien sollten die Warteschlangen der Router organisiert werden, um Datenstaus zu vermeiden?
- Wie kann man Denial-of-Service Attacken begegnen?
- Wie kann man Dateien verschlüsseln, so dass sogar der Verlust eines beträchtlichen Prozentsatzes aller Pakete ohne jeglichen Schaden kompensiert werden kann?
- Wie sollte man Peer-to-Peer Systeme aufbauen, so dass eine Vielzahl von Benutzern ohne zentrale Kontrollinstanz schnell und verlässlich bedient werden?

Eine gänzliche neue Situation entsteht durch die Nutzer des Internets, die egoistisch versuchen Profite zu erzielen. Welche Mechanismen oder Protokolle sollte man einführen, so dass egoistisches Verhalten zu sozialem Verhalten wird? Diese Fragen versucht man mit Hilfe der Spieltheorie zu beantworten.

Komplexitätstheorie

Wie schwierig ist das logische Schließen? Nicht so einfach, wie wir aus der NP-Vollständigkeit des $KNF-SAT$ Problems erfahren haben. Was passiert, wenn wir All- und Existenzquantoren erlauben, um die aussagenlogischen Variablen zu binden? Gehen wir in der Schwierigkeit noch höher und fragen nach dem Beweisen von zahlentheoretischen Aussagen, die nur die Addition betreffen. Was ist die Situation, wenn wir sogar Aussagen der uneingeschränkten Zahlentheorie automatisch beweisen möchten? Im letzten Fall stellt sich heraus, dass sogar jede „verständliche“ Axiomatisierung ausgeschlossen ist, wenn wir alle wahren Aussagen herleiten wollen: Wahrheit kann nicht formalisiert werden! Die additive Zahlentheorie ist bereits äußerst komplex und Algorithmen mit doppelt exponentieller Laufzeit werden benötigt. Das quantifizierte $KNF-SAT$ Problem haben wir bereits in Abschnitt 6.3.5 angesprochen: Es stellt sich als gleichschwer heraus, wie das Auffinden optimaler Züge in nicht-trivialen Zweipersonen Spielen.

Weitere Fragestellungen betreffen die Existenz von Pseudo-Random Generatoren und die Frage, um wieviel randomisierte Algorithmen effizienter als deterministische Algorithmen sein können. Kann man etwa erwarten, dass NP-vollständige Probleme geknackt werden können? Um die Antwort auf die letzte Frage vorwegzunehmen: No way!

Computational Learning Theory

Angenommen wir beobachten eine Runde von Experten, die Voraussagen abgeben. Einige Experten werden eine zeitlang treffende Voraussagen abgeben und dann später vielleicht erfolglos sein, während andere konsistent schlecht sind und andere sich wiederum mit der Zeit verbessern. Kann man einen Algorithmus entwickeln, der sich beinahe auf die besten Voraussagen einschießt?

Wann kann man von Beispielen eines Konzepts auf das Konzept schließen? Angenommen, man präsentiert positiv- und negativ-markierte Punkte im $\mathbb{R}^n$, die durch eine Hypereben voneinander getrennt werden können, verrät aber die Hyperebene nicht. Kann man die unbekannte Hyperebene rekonstruieren? Jein! Ja, wenn wir genügend viele charakteristische Beispiele erhalten, nein, wenn nicht. Aber was heißt „genügend viele“ und was heißt „charakteristisch“? Was passiert, wenn wir uns dieselbe Frage stellen, aber diesmal für ein kleines unbekanntes neuronales Netzwerk anstatt einer unbekannten Hyperebene?

Erstaunlicherweise kann man die für die Rekonstruktion eines unbekannten Konzepts notwendige Zahl positiver und negativer Beispiele fast exakt voraussagen, wenn man die Effizienz von Lernverfahren außen vor lässt. Dazu wird der Begriff der „Anzahl der Freiheitsgrade“ einer Klasse von Konzepten formalisiert.

Schließlich werden die wichtigsten Lernverfahren wie Support-Vektor Maschinen, neuronale Netzwerke, statistische Lernverfahren und Entscheidungsbaumverfahren untersucht, und die Boosting-Methode zur Verbesserung der Voraussage-Fähigkeit von Lernverfahren wird beschrieben.