
# Vorlesungsskript
# Komplexitätstheorie

Torben Hagerup

MPI–I–96–1–005 March 1996

FORSCHUNGSBERICHT ■ RESEARCH REPORT

MAX-PLANCK-INSTITUT
FÜR
INFORMATIK

Im Stadtwald ■ 66123 Saarbrücken ■ Germany

MAX-PLANCK-INSTITUT FÜR INFORMATIK

# Vorlesungsskript
# Komplexitätstheorie

Torben Hagerup

MPI–I–96–1–005 March 1996

# Vorwort

Der vorliegende Text ist das leicht überarbeitete Begleitmaterial zur Stammvorlesung "Komplexitätstheorie", die im Wintersemester 1995/96 an der Universität des Saarlandes gehalten wurde. Es handelte sich um eine Einführung in das Gebiet, die hauptsächlich von Studenten im dritten Studienjahr besucht wurde. Vorausgesetzt wurde eine gewisse Vertrautheit mit Turing-Maschinen sowie grundlegende Kenntnisse über z.B. Graphen, formale Sprachen und die $O$-Notation. Der Lehrstoff wurde in 31 90-minütigen Vorlesungseinheiten vorgestellt und in 14 Übungsdoppelstunden nachgearbeitet.

Als Vorlage für die Vorlesung und für dieses Skript diente vornehmlich das ausgezeichnete Lehrbuch "Computational Complexity" von Papadimitriou [3]. Insbesondere wurde die dort vertretene Sicht hier übernommen, wonach die Komplexitätstheorie weniger isoliert und um ihrer selbst willen betrieben, dafür mehr als ein mächtiges Hilfsmittel bei der Untersuchung und Klassifizierung der in der Praxis auftretenden Berechnungsaufgaben eingesetzt werden sollte. Das führt an manchen Stellen zu einem für klassische Darstellungen der Komplexitätstheorie ungewohnt algorithmischen Zugang, der dem einen oder anderen Leser befremdlich erscheinen mag. Zum Beispiel werden die Komplexitätsklassen coNP und coRP durch ausführliche Beispiele im Bereich der Primalitätstests mit Leben erfüllt, und der Begriff der Parallelverarbeitung wird durch einen nicht-trivialen parallelen Algorithmus zur Auswertung von arithmetischen Ausdrücken mittels Tree Contraction veranschaulicht.

Das Skript zeichnet sich weiterhin dadurch aus, daß bewußt darauf verzichtet wurde, ein Niveau an mathematischer Strenge einzuschlagen, das von den meisten Studenten nicht eingehalten und zum Teil nicht einmal wahrgenommen werden kann, weil Verständnisschwierigkeiten schon lange vorher einsetzen. Zum Beispiel wird man auf den folgenden Seiten vergebens eine Anleitung suchen, wie Objekte wie Graphen auf dem Band einer Turing-Maschine darzustellen sind; die Vorlesung versuchte vielmehr, den Studenten klar

zu machen, daß und warum die Details einer solchen Darstellung unwichtig und uninteressant sind. Es gibt zwar abwegige Darstellungskonventionen, bei denen unsere Aussagen ihre Gültigkeit verlieren, z.B. weil die Darstellungen "exponentiell aufgebläht" sind; aber auf derartige Konventionen wird man nur kommen, eben um spezielle Effekte zu erzielen, und die Schulung solch formalistischen Denkens war nicht Anliegen der Vorlesung. Vielmehr ging es darum, die Kernideen kraftvoll zu vermitteln und die Studenten ein—nicht nur bei wissenschaftlicher Arbeit nützliches—Gespür dafür entwickeln zu lassen, wann man präzise sein muß und wann nicht. Dadurch, daß wenig Zeit für an und für sich uninteressante Details aufgewendet werden mußte, war es möglich, einen recht großen Themenkreis zu behandeln.

Die Bemerkungen im vorigen Abschnitt sollen natürlich keine Entschuldigung für Schlampigkeit und Fehlerhaftigkeit darstellen. Ich habe mich bemüht, wirkliche Fehler im Skript auszumerzen, und bin zuversichtlich, eine geringere Fehlerhäufigkeit als das Lehrbuch von Papadimitriou erreicht zu haben. Hinweise auf etwaige verbliebene Fehler nehme ich gern entgegen. Leser, die eine rigorosere, dafür aber vielleicht weniger unmittelbar verständliche Darstellung der Komplexitätstheorie suchen, seien auf das Lehrbuch von Reischuk [5] verwiesen.

Im Skript wurde fast vollständig auf Quellennachweise verzichtet. Die einzigen genannten Quellen enthalten weiterführendes Material, das in der Vorlesung nicht behandelt wurde. Die fehlenden Angaben über Originalquellen für die Kerngebiete der Komplexitätstheorie können jedem einschlägigen Lehrbuch entnommen werden. Am Ende dieses Vorwortes sind diejenigen Quellen aufgeführt, auf die ich mich bei der Ausarbeitung des Skriptes direkt gestützt habe.

Ich möchte mich bei Jordan Gergov und Volker Priebe für zahlreiche inhaltliche und sprachliche Verbesserungsvorschläge bedanken, die die Qualität der Vorlesung wesentlich erhöht haben. Für nützliche Anregungen in der Vorbereitungsphase danke ich Ingrid Biehl, Shiva Chaudhuri, Kurt Mehlhorn und Stathis Zachos. Zu guter Letzt bedanke ich mich bei meinen Studenten für ihr kritisches Interesse und bei Hein Röhrig für die engagierte Betreuung der Übungen zur Vorlesung.

Torben Hagerup

Saarbrücken, im März 1996

[1] José Luis Balcázar, Josep Díaz and Joaquim Gabarró, *Structural Complexity I*, Springer-Verlag, Berlin, 1988.

[2] Daniel Pierre Bovet and Pierluigi Crescenzi, *Introduction to the Theory of Complexity*, Prentice-Hall, New York, 1994.

[3] Christos H. Papadimitriou, *Computational Complexity*, Addison-Wesley, Reading, MA, 1994.

[4] S. Hougardy, H. J. Prömel and A. Steger, Probabilistically checkable proofs and their consequences for approximation algorithms, *Disc. Math.* **136** (1994), pp. 175–223.

[5] Karl Rüdiger Reischuk, *Einführung in die Komplexitätstheorie*, B. G. Teubner, Stuttgart, 1990.

[6] R. L. Rivest, A. Shamir and L. Adleman, A method for obtaining digital signatures and public-key cryptosystems, *Comm. Assoc. Comput. Mach.* **21** (1978), pp. 120–126.

# Inhaltsverzeichnis

# 1 Überblick—einige Höhepunkte

Die Komplexitätstheorie ist, grob gesagt, die Lehre davon, wieviel an Ressourcen (z.B. Rechenzeit) benötigt wird, um bestimmte Berechnungsprobleme zu lösen. Während die Algorithmik konkrete Probleme sehr genau untersucht, wird in der Komplexitätstheorie eine allgemeinere Sicht angestrebt; oft werden ganze Klassen von Problemen mit ähnlichen Eigenschaften betrachtet.

Dieses Kapitel bietet eine kurze "Geschmacksprobe" dessen, was in den folgenden Kapiteln zu lesen ist. Die Darstellung ist sehr knapp gehalten, und viele Begriffe werden ohne Definition benutzt. In Kapitel 2 fangen wir dann sorgfältiger und ausführlicher wieder von vorne an.

**Die Turing-Maschine (TM)** (Fig. 1.1).

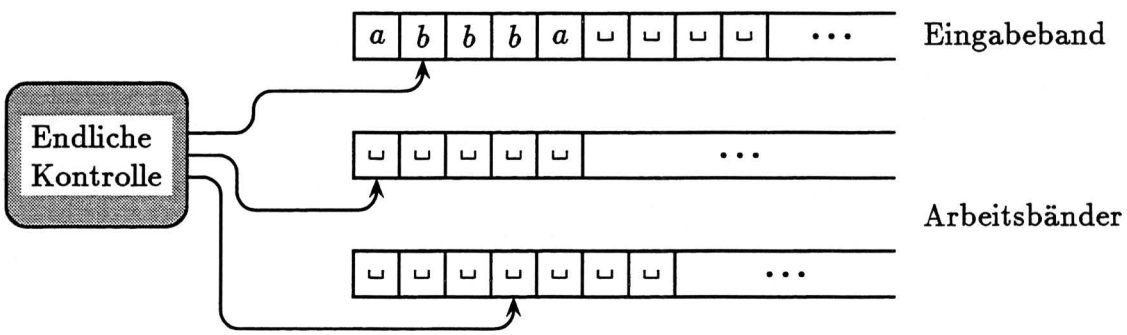



Fig. 1.1. Eine Turing-Maschine.

TIME($f(n)$) = Klasse der Sprachen, die von einer TM in Zeit $O(f(n))$ akzeptiert werden;
SPACE($f(n)$) = Klasse der Sprachen, die von einer TM in Platz $O(f(n))$ akzeptiert werden.

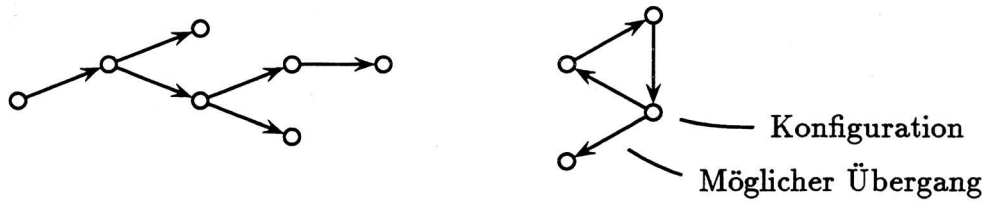



Fig. 1.2. Der Berechnungsgraph einer NTM.

Nicht-deterministische TM (NTM): Die Übergangsfunktion kann mehrdeutig sein. Konfigurationen können mehrere Nachfolgekonfigurationen haben (Fig. 1.2).

Eine NTM akzeptiert eine Eingabe, wenn es mindestens einen akzeptierenden Pfad von der Anfangskonfiguration gibt. Der Verbrauch an Ressourcen wird anhand des teuersten Pfades definiert.

NTIME($f(n)$) = Klasse der Sprachen, die von einer NTM in Zeit $O(f(n))$ akzeptiert werden;
NSPACE($f(n)$) = Klasse der Sprachen, die von einer NTM in Platz $O(f(n))$ akzeptiert werden.

$$\mathrm{P} = \bigcup_{k=1}^{\infty} \mathrm{TIME}(n^k) \qquad \mathrm{NP} = \bigcup_{k=1}^{\infty} \mathrm{NTIME}(n^k)$$
$$\mathrm{PSPACE} = \bigcup_{k=1}^{\infty} \mathrm{SPACE}(n^k) \qquad \mathrm{NPSPACE} = \bigcup_{k=1}^{\infty} \mathrm{NSPACE}(n^k)$$

**Satz** (Savitch): Sei $f : \mathbb{N}_0 \to \mathbb{N}$ konstruierbar und $f(n) \geq \log n$ für alle $n \in \mathbb{N}$. Dann ist $\mathrm{NSPACE}(f(n)) \subseteq \mathrm{SPACE}((f(n))^2)$.

**Korollar**: PSPACE = NPSPACE. (Die entsprechende Relation für Zeit wäre P = NP.)

Sei $C$ eine Komplexitätsklasse. Dann ist $\mathrm{co}C = \{L \mid \overline{L} \in C\}$.

**Beispiel**: $\overline{\mathrm{SAT}} \in \mathrm{coNP}$. $\overline{\mathrm{SAT}}$ = Menge der nicht-erfüllbaren Booleschen Formeln in konjunktiver Normalform (CNF) $\bigcup$ Menge der Eingaben, die überhaupt keine Formeln in CNF darstellen. P = coP.

Sprachen in NP haben kurze "Beweise" (Bsp. für SAT: Eine erfüllende Belegung der Variablen). Sprachen in coNP haben kurze "Gegenbeweise" (Bsp. für $\overline{\mathrm{SAT}}$: Eine erfüllende Belegung).

**Satz** (Immerman/Szelepcsényi): Sei $f : \mathbb{N}_0 \to \mathbb{N}$ konstruierbar und $f(n) \geq \log n$ für alle $n \in \mathbb{N}$. Dann ist NSPACE($f(n)$) = coNSPACE($f(n)$).

(Ein entsprechendes Resultat für Zeit würde ergeben: NP = coNP, ein wichtiges offenes Problem.)

**Vollständigkeit.**

Ein NP-vollständiges Problem ist ein schwierigstes Problem in NP: Alle anderen Probleme in NP lassen sich mit vernachlässigbarem (nämlich polynomiellem) Aufwand darauf reduzieren. Viele andere Komplexitätsklassen haben ebenfalls vollständige Probleme.

Quantified Boolean Formulas:

$$\text{Bsp.:} \qquad \forall x \exists y \forall z : \big((x \vee \neg y) \wedge (y \wedge \neg z)\big).$$

QBF ist die Sprache der wahren quantifizierten Booleschen Formeln.

**Satz:** QBF ist PSPACE-vollständig.

Es gibt auch P-vollständige Probleme. Damit nicht alles Unsinn wird, brauchen wir eine eingeschränktere Form der Reduktion als Polynomialzeitreduktion.

**Definition** (logspace-Reduktion): Eine Sprache $L_1$ ist logspace-reduzierbar auf eine Sprache $L_2$, falls es eine Funktion $R$ gibt, die von einer $TM$ mit $O(\log n)$ Arbeitsplatz berechnet werden kann, so daß $x \in L_1 \Leftrightarrow R(x) \in L_2$ (Fig. 1.3).

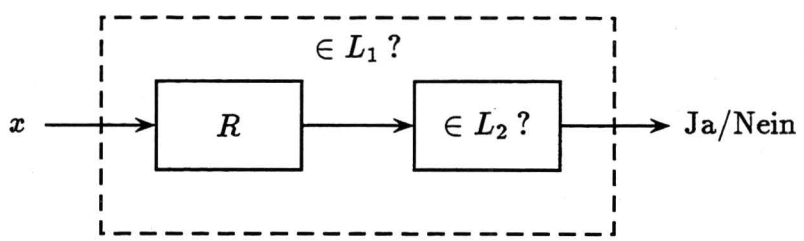



Fig. 1.3. Die Wirkungsweise einer Reduktion $R$ von $L_1$ auf $L_2$.

**Satz** (Circuit-Value-Problem): Das Problem, einen Schaltkreis (mit 0/1 an den Eingängen; s. Fig. 1.4) auszuwerten, ist P-vollständig unter logspace-Reduktion.

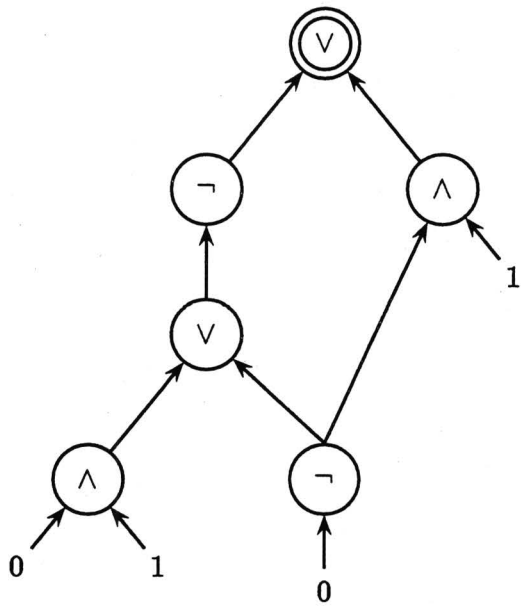



Fig. 1.4. Ein Schaltkreis.

**Parallele Komplexitätstheorie.**

P-Vollständigkeit ist unter anderem für parallele Berechnungen von Interesse. Ist ein P-vollständiges Problem gut parallelisierbar, gilt dasselbe für alle Probleme in P. Man vermutet das Gegenteil.

Ein Schaltkreis, jetzt mit Variablen $x_1, \ldots, x_n$ an den Eingängen, berechnet eine Funktion $f_n : \{0,1\}^n \to \{0,1\}$. Eine Schaltkreisfamilie $\{\gamma_n\}_{n=0}^{\infty}$ berechnet eine Funktion $f : \{0,1\}^* \to \{0,1\}$, akzeptiert also eine Sprache $L \subseteq \{0,1\}^*$. Wir verlangen von Schaltkreisfamilien eine gewisse Regularität (Uniformität); sonst können sie viel zu viele (z.B. nicht berechenbare) Funktionen realisieren.

NC = Menge der Sprachen, die von uniformen Schaltkreisen polynomieller Größe und polylogarithmischer Tiefe akzeptiert werden. Große offene Frage: NC $\stackrel{?}{=}$ P.

Parallele Berechnungsthese: Parallele Zeit $\sim$ Sequentieller Platz.

**Satz**: Sei $L$ eine Sprache, die von einer uniformen Schaltkreisfamilie $\{\gamma_n\}_{n=0}^{\infty}$ mit beschränktem Fanin der Tiefe $f(n) \geq \log n$ und der Größe $2^{O(f(n))}$ akzeptiert wird. Dann ist $L \in \mathrm{SPACE}(f(n))$.

**Satz**: Sei $L \subseteq \{0,1\}^*$ eine Sprache in $\mathrm{NSPACE}(f(n))$, wobei $f : \mathbb{N}_0 \to \mathbb{N}$ eine konstruierbare Funktion mit $f(n) \geq \log n$ für alle $n \in \mathbb{N}$ ist. Dann wird $L$ von einer uniformen Schaltkreisfamilie mit beschränktem Fanin der Tiefe $O((f(n))^2)$ akzeptiert.

**Approximationsverfahren.**

Können wir ein Optimierungsproblem nicht exakt lösen (z.B. weil das entsprechende Entscheidungsproblem NP-vollständig ist), können wir immer noch versuchen, eine annähernd optimale Lösung zu berechnen. Ein Approximationsverfahren für ein Optimierungsproblem arbeitet mit Approximationsgüte $\rho \geq 1$, wenn die von ihm berechnete Lösung höchstens um einen Faktor von $\rho$ schlechter als die optimale Lösung ist (z.B. mit $\rho$-mal so hohen Kosten verbunden ist).

KNAPSACK: Für jedes $\rho > 1$ gibt es ein Approximationsverfahren mit Approximationsgüte $\rho$, das in Polynomialzeit arbeitet.

CLIQUE: Kein Polynomialzeitverfahren kann konstante Approximationsgröße haben (außer wenn P = NP).

KNAPSACK ist ein sehr "gutartiges" Problem, während CLIQUE ein harter Brocken ist—obwohl beide NP-vollständig sind!

**Randomisierte Verfahren.**

Eine probabilistische TM wählt in bestimmten Zuständen mit gleicher Wahrscheinlichkeit zwischen zwei Nachfolgezuständen. PP ist die Klasse der Sprachen, die von probabilistischen TMs akzeptiert werden, die in Polynomialzeit laufen und Fehlerwahrscheinlichkeit $< 1/2$ haben. Sei #SAT die Sprache der Paare $(F, k)$, wobei $F$ eine Boolesche Formel mit mehr als $k$ erfüllenden Belegungen ist.

**Satz:** NP $\subseteq$ PP $\subseteq$ PSPACE.

**Satz:** #SAT ist PP-vollständig.

PP läßt TMs zu, deren Antworten mit Wahrscheinlichkeit fast 1/2 falsch sind. BPP (Bounded-error Probabilistic Polynomial time) ist wie PP definiert, aber die Fehlerwahrscheinlichkeit darf höchstens 1/4 sein. RP läßt außerdem keinen Fehler zu bei Eingaben, die nicht zur betrachteten Sprache gehören (sie müssen immer abgelehnt werden).

**Satz:** P $\subseteq$ RP $\subseteq$ NP $\cap$ BPP.

**Kryptographie.**

RSA: System zur Übertragung verschlüsselter Nachrichten.
Anwendungen:

- Digitale Unterschriften
- Münzwürfe per Telefon.

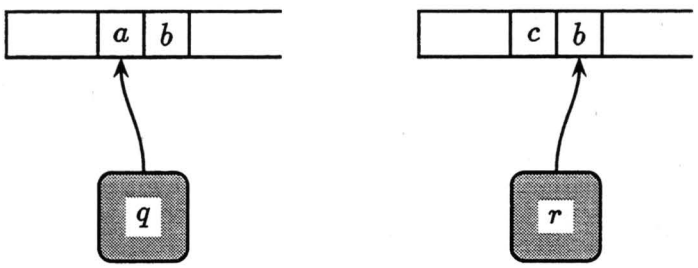
# 2 Turing-Maschinen

Die *Turing-Maschine (TM)* ist unser grundlegendes Berechnungsmodell (einfach, mächtig).

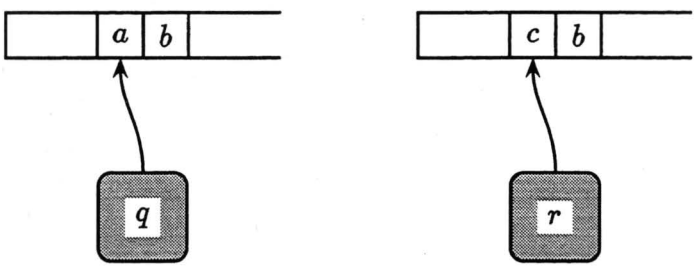
Zur Angabe einer (deterministischen, 1-Band) TM gehören:

- $Q$, eine endliche Menge von *Zuständen*
- $\Sigma$, ein endliches *Bandalphabet*
- $\delta$, eine *Übergangsfunktion* (Fig. 2.1)

$$\delta : Q \times \Sigma \to (Q \cup \{\texttt{accept}, \texttt{reject}\}) \times \Sigma \times \{\to, \leftarrow, \downarrow\}$$




$$\delta(q, a) = (r, c, \to)$$




$$\delta(q, a) = (r, c, \downarrow)$$

Fig. 2.1. Die Bedeutung der Übergangsfunktion $\delta$.

Zur Benutzung der Maschine: Gestartet wird in einem ausgezeichneten *Startzustand* $q_0$ mit dem Kopf auf dem ersten (linkesten) Feld. Eine Eingabe der Länge $n$ steht am Anfang in den ersten $n$ Feldern; alle anderen Felder enthalten anfangs ein ausgezeichnetes *Leersymbol* $\sqcup \in \Sigma$. Formal können wir die TM und die Konventionen für ihre Benutzung in dem Tupel $(Q, \Sigma, \delta, q_0, \sqcup)$ zusammenfassen.

Die Berechnung kann immer weitergehen, bis einer der Zustände `accept` oder `reject` erreicht wird. Soll der Kopf vom linkesten Feld aus laut $\delta$ nach links bewegt werden, bleibt er einfach stehen. Aus Aufgabe 1.1 wissen wir, daß in konstanter Zeit überprüft werden kann, ob sich der Kopf auf dem linkesten Feld befindet.

*Konfiguration* einer TM: Vollständige Beschreibung des Zustandes von Kontrolle + Band, d.h. $(q, y, z)$, wobei $q$ der Zustand, $y$ der Bandinhalt bis einschließlich unter dem Kopf und $z$ der Bandinhalt rechts vom Kopf ist. Eine Konfiguration ist akzeptierend (verwerfend), falls der Zustand der Konfiguration `accept` (`reject`) ist.

*Übergangsfunktion* $\xrightarrow{M}$ einer TM $M$: Bildet jede Konfiguration auf die Nachfolgekonfiguration ab. Zusammen mit einer TM betrachten wir deren *Übergangsgraph* $G_M$: $G_M$ ist ein unendlicher, gerichteter Graph mit einem Knoten für jede Konfiguration von $M$; zwei Knoten $K_1$ und $K_2$ in $G_M$ sind genau dann mit einer Kante $(K_1, K_2)$ verbunden, wenn $K_1 \xrightarrow{M} K_2$. Eine *Berechnung* von $M$ ist ein Pfad in $G_M$, der unendlich ist ($M$ hält nicht) oder in einer Konfiguration ohne Nachfolgekonfiguration endet. Eine Berechnung heißt akzeptierend (verwerfend), wenn sie in einer akzeptierenden (verwerfenden) Konfiguration endet. Eine Berechnung von $M$ *auf* einer Eingabe $x = x_1 \cdots x_n \in (\Sigma \backslash \{\sqcup\})^*$ ist eine Berechnung von $M$, die in der Anfangskonfiguration $Init_M(x) = (q_0, x_1, x_2 \cdots x_n \sqcup\sqcup \cdots)$ beginnt.

Die von $M$ *akzeptierte* oder *erkannte* Sprache ist definiert als

$$L(M) = \{x \in (\Sigma \backslash \{\sqcup\})^* \mid \text{Die Berechnung von } M \text{ auf } x \text{ ist akzeptierend}\}.$$

Bei Eingaben, die nicht zu $L(M)$ gehören, endet die Berechnung von $M$ in einer verwerfenden Konfiguration, oder $M$ hält nicht.

*TMs mit mehreren Bändern:* Die Definitionen lassen sich in der offensichtlichen Weise verallgemeinern. Es gibt ein ausgezeichnetes Eingabeband, auf dem die Eingabe bereitgestellt wird. Meistens fordern wir, daß das Eingabeband nur gelesen, aber nicht verändert wird (die Eingabe ist *read-only*). Dann verlangen wir auch, daß der Kopf auf dem Eingabeband sich nie weiter nach rechts als bis zum ersten $\sqcup$ bewegt.

*Zeitverbrauch* bei Eingabe $x$: Anzahl Schritte in der Berechnung auf $x$ (evtl. $\infty$).

*Platzverbrauch* bei Eingabe $x$: Anzahl Felder, die bei der Berechnung auf $x$ von einem Kopf "berührt" werden (evtl. $\infty$). Ist die Eingabe read-only, wird der Platzverbrauch auf dem Eingabeband nicht mitgezählt.

Sei $f : \mathbb{N}_0 \to \mathbb{N}$ ($\mathbb{N} = \{1, 2, \ldots\}$ und $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$). $M$ arbeitet in Zeit (Platz) $f$: Für alle Eingaben $x \in (\Sigma \backslash \{\sqcup\})^*$ ist der Zeitverbrauch (Platzverbrauch) von $M$ bei Eingabe $x$ durch $f(|x|)$ beschränkt.

$$\text{TIME}(f) = \{L(M) \mid M \text{ ist eine Mehrband-TM, die in Zeit } O(f) \text{ arbeitet}\}$$

$$\text{SPACE}(f) = \{L(M) \mid M \text{ ist eine Mehrband-TM, die in Platz } O(f) \text{ arbeitet}\}$$

In der Definition von SPACE($f$) setzen wir voraus, daß die Eingabe read-only ist. Das erlaubt uns, SPACE($f$) auch für sublineare Funktionen $f$ zu untersuchen.

Wir interessieren uns bei Ressourceschranken nicht für konstante Faktoren, schon deswegen, weil selbst die größenordnungsmäßige Komplexität eines Problems in den seltensten Fällen bekannt ist. Deswegen haben wir oben TIME($f$) und SPACE($f$) gleich unter Benutzung von $O(f)$ statt $f$ definiert. Wir werden die Notation TIME($f$) auch dann benutzen, wenn $f(n)$ nicht ganzzahlig, nicht überall positiv oder nicht für alle $n \in \mathbb{N}_0$ definiert ist (ein Beispiel: TIME($n \log \log n$)). In solchen Fällen meinen wir eigentlich TIME($g$), wobei $g(n) = 1$ für alle $n \in \mathbb{N}_0$, für die $f(n)$ undefiniert ist, und $g(n) = \max\{\lfloor f(n) \rfloor, 1\}$ für alle anderen $n \in \mathbb{N}_0$. Oft schreiben wir TIME($f(n)$) an Stelle von TIME($f$) (das wurde schon im obigen Beispiel getan). Für SPACE($f$) gelten analoge Konventionen.

**Beispiel:** Die Sprache der Palindrome (Wörter, die von vorne und von hinten gelesen gleich sind (wie "LAGERREGAL")) über einem festen Alphabet gehört zu TIME($n$) (kann in Linearzeit erkannt werden) und zu SPACE($\log n$) (kann in logarithmischem Platz erkannt werden). Vgl. auch Aufgabe 3.A.

Man beachte, daß TIME($f$) und SPACE($f$) mit Bezug auf *Mehrband*-TMs definiert sind. Wir erlauben also für die Erkennung einer Sprache eine beliebige (aber konstante, d.h. nicht von der Eingabe abhängige) Anzahl von Arbeitsbändern. Eine TM mit einer kleineren Anzahl von Bändern könnte es eventuell schwerer haben, die betreffende Sprache zu akzeptieren. Dieser Frage gehen wir jetzt nach.

**Satz** 2.1 (Simulation von $k$-Band TMs auf 1-Band TMs): Seien $k \in \mathbb{N}$ und $f, g : \mathbb{N}_0 \to \mathbb{N}$ und sei $M$ eine $k$-Band TM, die in Zeit $f$ und Platz $g$ arbeitet, wobei der Platzverbrauch auf dem Eingabeband mitgerechnet wird. Dann gibt es eine zu $M$ äquivalente 1-Band TM $M'$, die in Zeit $O(f^2)$ und Platz $O(g)$ arbeitet. *Äquivalent* heißt, daß $L(M') = L(M)$.

**Beweis:** Die Aussage folgt durch eine einfache Simulation. Die Maschine $M'$ hält für alle $i \in \mathbb{N}$ im $i$ten Feld ihres Bandes den Inhalt der $i$ten Felder aller $k$ Bänder von $M$ sowie Information darüber, welche von diesen Feldern sich unter einem Kopf befinden. Die Kodierung von $k$-Tupeln von Symbolen von $M$ als Symbole von $M'$ wird so gewählt, daß ein Symbol $a$ von $M$ von $M'$ als das $k$-Tupel $(a, \sqcup, \ldots, \sqcup)$ (ohne Köpfe) interpretiert wird, wobei wir o.B.d.A. davon ausgehen, daß das Eingabeband an erster Stelle steht. Damit ist die Anfangskonfiguration von $M'$ schon korrekt für die Simulation, außer daß das erste Feld modifiziert werden muß, um die Präsenz aller Köpfe anzuzeigen. Als Vorbereitung für die Simulation eines Schrittes von $M$ führt $M'$ seinen Kopf über den gesamten benutzten Bandbereich und merkt sich dabei (in der endlichen Kontrolle) die unter den simulierten Köpfen gesehenen Symbole. Die $k$ simulierten Bänder werden dann entsprechend der Übergangsfunktion von $M$ verändert.

Die Simulation eines Schrittes von $M$ erfolgt in einer Zeit, die proportional zur Gesamtlänge der betretenen Bandbereiche von $M$ ist. Bei einer Eingabe der Länge $n$ ist diese Größe durch $f(n)$ beschränkt, so daß die gesamte Simulation in Zeit $O((f(n))^2)$ läuft. Der benutzte Platz ist $O(g(n))$. $M'$ braucht jetzt nur genau dann zu akzeptieren, wenn $M$ akzeptiert. ∎

Wir brauchen auch einen ähnlichen Satz, bei dem $g$ sublinear sein kann (die Eingabe also read-only ist). Dann müssen wir $M'$ erlauben, zwei Bänder zu haben, denn mit einem einzigen read-only Band ist nicht viel anzufangen.

**Satz** 2.2: Seien $k \in \mathbb{N}$ und $f, g : \mathbb{N}_0 \to \mathbb{N}$ und sei $M$ eine $k$-Band TM mit read-only Eingabe, die in Zeit $f$ und Platz $g$ arbeitet. Dann gibt es eine 2-Band TM $M'$ mit read-only Eingabe und $L(M') = L(M)$, die in Zeit $O(f^2)$ und Platz $O(g)$ arbeitet.

**Beweis:** Das Eingabeband von $M'$ wird genau wie das Eingabeband von $M$ benutzt, während auf dem zweiten Band alle Arbeitsbänder von $M$ wie im vorherigen Beweis simuliert werden. ∎

Die Simulation aus dem Beweis von Satz 2.1 mag einfallslos erscheinen, aber besser geht es nicht: Ein quadratischer Zeitverlust ist im allgemeinen unvermeidbar (Aufgabe 3.A). Meistens interessieren wir uns allerdings nicht für so feine Unterschiede. Arbeitet z.B. $M$ in Polynomialzeit, so gilt das auch für $M'$. Daher ist uns die genaue Anzahl der Bänder einer TM oft egal.

**Bemerkung** 2.3: Eine $k$-Band TM, die in Zeit $f$ arbeitet, kann auf einer 2-Band TM in Zeit $O(f \log f)$ simuliert werden (F. C. Hennie and R. E. Stearns, Two-tape simulation of multitape Turing machines, *J. ACM* **13** (1966), pp. 533–546).

Eine *nichtdeterministische TM (NTM)* ist wie eine (deterministische) TM definiert, außer daß die Übergangsfunktion "mehrdeutig" sein darf (zu jedem Argument gibt es 0, 1 oder mehrere Bilder). Formal ersetzen wir die Funktion

$$\delta : Q \times \Sigma \to (Q \cup \{\texttt{accept}, \texttt{reject}\}) \times \Sigma \times \{\rightarrow, \leftarrow, \downarrow\}$$

durch eine Relation

$$\delta \subseteq [Q \times \Sigma] \times [(Q \cup \{\texttt{accept}, \texttt{reject}\}) \times \Sigma \times \{\rightarrow, \leftarrow, \downarrow\}].$$

Zu einer Konfiguration kann es also 0, 1 oder mehrere (aber konstant viele) Nachfolgekonfigurationen geben. Der Konfigurationsgraph hat (wie gehabt) einen Knoten für jede Konfiguration und enthält eine gerichtete Kante $(K_1, K_2)$ genau dann, wenn $K_2$ eine mögliche Nachfolgekonfiguration von $K_1$ ist. Wir definieren die von einer NTM $M$ akzeptierte Sprache als

$$L(M) = \{x \in (\Sigma \backslash \{\sqcup\})^* \mid \text{Es gibt eine akzeptierende Berechnung von } M \text{ auf } x\}.$$

Der Vergleich mit der entsprechenden Definition für (deterministische) TMs ergibt einen großen Unterschied: Während eine deterministische Maschine eine einzige Berechnung auf $x$ hat, kann es bei einer nichtdeterministischen Maschine viele solcher Berechnungen geben. $x$ wird akzeptiert, wenn auch nur eine dieser Berechnungen akzeptiert. Intuitiv kann man das so verstehen, daß eine NTM $M$ "raten" kann und immer richtig rät, nämlich die Richtung zu einer akzeptierenden Konfiguration. Eine alternative, nützliche Betrachtungsweise besagt, daß $x \in L(M)$ genau dann, wenn es für diese Behauptung einen "kurzen Beweis" gibt, nämlich den akzeptierenden Berechnungspfad auf $x$ (im Gegensatz zum gesamten, vielleicht exponentiell größeren Berechnungsbaum). Im Gegensatz zur TM modelliert die NTM existierende (oder vorstellbare?) Rechner nicht gut, weil sie wegen der "Ratefähigkeit" viel zu mächtig ist. Dennoch ist die NTM für unsere Untersuchungen unentbehrlich.

Der Verbrauch einer NTM an Zeit und Platz bei Eingabe $x$ wird anhand des *teuersten* Berechnungspfades auf $x$ definiert. Die Laufzeit ist also z.B. die Länge des längsten Pfades, der in $Init_M(x)$ anfängt (auch wenn es kürzere akzeptierende Pfade gibt).

$$\text{NTIME}(f) = \{L(M) \mid M \text{ ist eine Mehrband-NTM, die in Zeit } O(f) \text{ arbeitet}\}$$

$$\text{NSPACE}(f) = \{L(M) \mid M \text{ ist eine Mehrband-NTM, die in Platz } O(f) \text{ arbeitet}\}$$

In der Definition von NSPACE($f$) setzen wir wieder voraus, daß die Eingabe read-only ist.

**Bemerkung** 2.4: Wie man leicht sieht, gelten die Aussagen der Sätze 2.1 und 2.2 genauso für nichtdeterministische Berechnungen, wenn also $M$ und $M'$ beide NTMs statt TMs sind.

Bisher haben wir TMs als *Akzeptoren* benutzt, die auf einer Eingabe mit "Ja" oder "Nein" antworten. Manchmal möchten wir eine TM $M$ dazu benutzen, eine Funktion von $(\Sigma\backslash\{\sqcup\})^*$ nach $\Sigma^*$ zu berechnen. Wir setzen dabei voraus, daß $M$ bei jeder Eingabe $x$ hält, und zeichnen ein Band von $M$ als *Ausgabeband* aus. Der zu $x$ gehörige Funktionswert befindet sich auf dem Ausgabeband, nachdem $M$ gehalten hat, und ist definiert als derjenige Teil des Bandinhalts, der sich links vom Ausgabekopf befindet. Das Ausgabeband ist write-only, sein Kopf kann sich nicht nach links bewegen, und das Ausgabeband wird bei der Angabe des Platzverbrauchs nicht berücksichtigt.

*Kodierung:* Alle endlichen Objekte, die bisher vorkamen, wie Zustandsmengen, Bandalphabete, Übergangsfunktionen oder -relationen, TMs und NTMs usw., können über einem festen endlichen Alphabet kodiert bzw. repräsentiert werden. Das sieht man z.B. daran, daß es sicherlich möglich wäre, z.B. eine TM mit Hilfe von ASCII-Zeichen vollständig zu beschreiben. Wir möchten auch Konfigurationen und Konfigurationsgraphen repräsentieren, obwohl diese Objekte unendlich sind. Was Konfigurationen betrifft, ist es einfach, denn Konfigurationen sind nur wegen der unendlichen Bänder unendlich, und wir können eine unendliche Folge von Leersymbolen durch ein einziges Symbol darstellen (es können ja nur endlich viele Symbole von $\sqcup$ verschieden sein). Bei Konfigurationsgraphen interessieren wir uns meist nur für endliche Teilgraphen (z.B. weil eine Eingabe und eine Platzschranke vorgegeben sind), und es reicht, diese zu repräsentieren. Es ist leicht, aber langweilig, eine Standardkodierung all dieser Objekte festzulegen. Nehmen wir an, wir hätten eine solche.

Das erlaubt uns, z.B. TMs als Objekte aufzufassen, mit denen wir herumrechnen können (eine TM ist ja bloß (kodiert als) eine Zeichenfolge). Z.B. können wir pervers genug sein, ausprobieren zu wollen, wie eine TM auf sich selbst als Eingabe reagiert. Wir können jetzt auch einen universellen TM-Simulator bauen, der eine TM $M$ und ein Wort $x$ als Eingabe nimmt und $M$ auf $x$ simuliert. Das ist wie bei einem modernen Rechner, bei dem Hardware und Software getrennt sind.

# 3 Relationen zwischen Komplexitätsklassen

Frage: Welche Ressourcen sind wertvoller als andere, und um wieviel?

Es gibt sehr viele Funktionen von $\mathbb{N}_0$ nach $\mathbb{N}$, und wenn wir sie alle als Ressourceschranken benutzen, treten sonderbare und unerwünschte Effekte auf. Das Problem hängt damit zusammen, daß einige Funktionen so kompliziert sind, daß sie "in ihren eigenen Schranken" nicht berechnet werden können. Daher beschränken wir uns oft auf Funktionen, die *konstruierbar* sind. Eine Funktion $f : \mathbb{N}_0 \to \mathbb{N}$ heißt konstruierbar, falls

(1) $f(n+1) \geq f(n)$ für alle $n \in \mathbb{N}_0$;

(2) Es gibt eine Mehrband-TM $M_f$ mit read-only Eingabe, die bei jeder Eingabe der Länge $n \in \mathbb{N}_0$ die Ausgabe $I^{f(n)}$ berechnet, wobei $I \in \Sigma \backslash \{\sqcup\}$, und die auf Eingaben der Länge $n$ in Zeit $O(n + f(n))$ und Platz $O(f(n))$ arbeitet. Wir sagen auch, daß eine solche TM $f$ berechnet.

Beispiele für konstruierbare Funktionen:

$f(n) =$ konstant

$f(n) = n$

$f(n) = \lceil \log n \rceil$ (zähle die Länge der Eingabe binär).

Die "üblichen" Funktionen oberhalb von $\log n$ sind konstruierbar (Aufgaben 3.B.2 und 3.B.3). Wir werden jetzt zeigen, daß es unterhalb von $\log n$ nur triviale konstruierbare Funktionen gibt.

**Satz** 3.1: Für jede konstruierbare Funktion $f : \mathbb{N}_0 \to \mathbb{N}$ gilt: $f(n) = O(1)$ oder $f(n) = \Omega(\log n)$.

**Beweis:** Sei $f : \mathbb{N}_0 \to \mathbb{N}$ eine konstruierbare Funktion mit $\neg[f(n) = \Omega(\log n)]$ und sei $M_f$ eine TM, die $f$ in Platz $O(f)$ berechnet. Definieren wir eine *Halbkonfiguration* als eine Konfiguration von $M_f$, bei der das Eingabeband außer acht gelassen wird. Da

$\neg[f(n) = \Omega(\log n)]$, gibt es eine Eingabe der Form $I^m$, bei der es weniger als $m$ Halbkonfigurationen gibt, die von $M_f$ erreicht werden können. Daraus folgt, daß bei jedem vollständigen Überqueren der Eingabe eine Halbkonfiguration wiederholt wird, und zwar mit dem Eingabekopf an zwei verschiedenen Stellen, sagen wir $k$ Positionen auseinander. Aber dann würde der Kopf bei Eingabe $I^{m+k}$ nach dem Überqueren der Eingabe in der gleichen Halbkonfiguration wie bei Eingabe $I^m$ ankommen, d.h. diese Überquerung kann nicht zwischen den Eingaben $I^m$ und $I^{m+k}$ unterscheiden. Genereller kann diese Überquerung nicht zwischen $I^m$ und $I^{m+ik}$ unterscheiden, für beliebiges $i \in \mathbb{N}$. Eine sich anschließende Überquerung kann vielleicht schon zwischen zwei solchen Eingaben unterscheiden, weil sie mit einem anderen Wert von $k$ verbunden ist. Alle möglichen Werte von $k$ sind aber durch $m$ beschränkt, so daß sie alle Teiler von $m!$ sind. Damit kann keine Überquerung und auch keine Folge von Überquerungen zwischen $I^m$ und $I^{m+im!}$ unterscheiden, für beliebiges $i \in \mathbb{N}$. Daraus folgt, daß $f(m+im!) = f(m)$ für alle $i \in \mathbb{N}$. Da $f$ nicht-fallend ist, können wir schließen, daß $f(n) = f(m)$ für alle $n \geq m$, d.h. $f(n) = O(1)$. ∎

Ist $M$ eine TM oder NTM, die in Platz $f$ arbeitet und die $c$ Arbeitsbänder zusätzlich zu einem read-only Eingabeband hat, so gibt es zu jeder Eingabe $x$ mit $|x| = n$ eine Menge $V_M(x)$ von Konfigurationen von $M$, die alle Konfigurationen enthält, die $M$ von $Init_M(x)$ aus erreichen kann, und so daß $|V_M(x)| \leq q \cdot k^{cf(n)} \cdot (f(n))^c \cdot (n+1)$, für bestimmte Konstanten $q$ und $k$. Das ist einfach deswegen, weil die Kontrolle von $M$ nur endlich viele Zustände hat (sagen wir $q$), jedes der höchstens $cf(n)$ benutzbaren Felder auf den Arbeitsbändern nur endlich viele Symbole aufweisen kann (sagen wir $k$), während jeder der $c$ Köpfe der Arbeitsbänder sich an $f(n)$ verschiedenen Stellen und der Eingabekopf sich an $n+1$ verschiedenen Stellen befinden kann. Ist $f(n) \geq \log n$ für alle $n \in \mathbb{N}$, so ist $|V_M(x)| = 2^{O(f(n))}$, und ist $f$ zusätzlich konstruierbar, können wir die Elemente von $V_M(x)$ nacheinander in Platz $O(f(n))$ generieren. Wir definieren $G_M(x)$ als den von $V_M(x)$ aufgespannten Teilgraphen von $G_M$. Die oben gemachten Aussagen gelten weiterhin, falls wir nur wissen, daß $M$ in Platz $O(f)$ (und nicht genau $f$) arbeitet.

**Satz** 3.2: Sei $f : \mathbb{N}_0 \to \mathbb{N}$. Dann gilt:

(a) $\mathrm{TIME}(f) \subseteq \mathrm{SPACE}(f) \cap \mathrm{NTIME}(f)$;

(b) $\mathrm{SPACE}(f) \cup \mathrm{NTIME}(f) \subseteq \mathrm{NSPACE}(f)$;

(c) $\mathrm{NTIME}(f) \subseteq \mathrm{SPACE}(f)$;

(d) Ist $f$ konstruierbar und $f(n) \geq \log n$ für alle $n \in \mathbb{N}$, dann ist $\mathrm{SPACE}(f) \cup \mathrm{NTIME}(f) \cup \mathrm{NSPACE}(f) \subseteq \mathrm{TIME}(2^{O(f)})$.

**Bemerkung:** Unter $\text{TIME}(2^{O(f)})$ verstehen wir $\bigcup_{k=1}^{\infty} \text{TIME}(2^{kf})$.

**Beweis:** (a) und (b): Trivial. Z.B. für (a): Eine TM, die eine Sprache $L$ in Zeit $O(f)$ erkennt, ist auch eine NTM, die $L$ in Zeit $O(f)$ erkennt.

(c) Sei $L \in \text{NTIME}(f)$ und sei $M$ eine NTM, die das bezeugt (d.h. $M$ erkennt $L$ in Zeit $O(f)$). Bei Eingabe $x$ mit $|x| = n$ simulieren wir deterministisch und nacheinander alle möglichen Berechnungen von $M$ auf $x$ und akzeptieren $x$ genau dann, wenn mindestens eine akzeptierende Konfiguration angetroffen wird. Der Platzverbrauch von $M$ ist $O(f(n))$, und die Simulation braucht nur $O(f(n))$ zusätzlichen Platz, um die simulierten nichtdeterministischen Entscheidungen der vorherigen Berechnung zu speichern—damit kann die nächste Berechnung durchgeführt werden.

(d) Wegen (b) reicht es, die Relation $\text{NSPACE}(f) \subseteq \text{TIME}(2^{O(f)})$ zu zeigen. Sei also $L \in \text{NSPACE}(f)$ und sei $M$ eine NTM, die das bezeugt. Gegeben eine Eingabe $x$ mit $|x| = n$, erstellen wir $G_M(x)$ komplett (erinnern Sie sich daran, daß $G_M(x)$ der Konfigurationsgraph ist, aber eingeschränkt auf eine Obermenge der Konfigurationen, die tatsächlich von $Init_M(x)$ aus erreicht werden können). Da $f$ konstruierbar ist und $f(n) \geq \log n$, folgt aus den Betrachtungen vor Satz 3.2, daß $G_M(x)$ die Größe $2^{O(f(n))}$ hat und auch in dieser Zeit erstellt werden kann. Wir müssen jetzt nur noch entscheiden, ob es in $G_M(x)$ einen Weg von $Init_M(x)$ zu einer akzeptierenden Konfiguration gibt. Das können wir mittels Tiefensuche (DFS) in $2^{O(f(n))}$ Zeit. Also ist $L \in \text{TIME}(2^{O(f)})$. ∎

Informelle Zusammenfassung von Satz 3.2: Platz ist mächtiger als Zeit, und Nichtdeterminismus ist mächtiger als Determinismus, aber jeweils höchstens exponentiell viel mächtiger.

Beim Übergang von nichtdeterministischem Platz zu deterministischem Platz können wir einen viel kleineren Verlust erreichen:

**Satz** 3.3 (Savitch): Sei $f : \mathbb{N}_0 \to \mathbb{N}$ konstruierbar und $f(n) \geq \log n$ für alle $n \in \mathbb{N}$. Dann ist $\text{NSPACE}(f) \subseteq \text{SPACE}(f^2)$.

**Beweis:** Sei $L \in \text{NSPACE}(f)$ und sei $M$ eine NTM, die das bezeugt. Bei Eingabe $x$ mit $|x| = n$ müssen wir wie im Beweis von Satz 3.2(d) testen, ob es im Konfigurationsgraphen $G_M(x)$ einen Weg von $Init_M(x)$ zu einer akzeptierenden Konfiguration gibt, und zwar deterministisch und unter Benutzung von möglichst wenig Platz.

Es kommt nicht in Frage, den Graphen $G_M(x)$ wie im Beweis von Satz 3.2(d) vollständig zu konstruieren, denn er ist viel zu groß. Wir können auch nicht DFS machen,

ohne den Graphen vorher hinzuschreiben, denn dazu müßten wir zumindest in der Lage sein, uns den im Moment explorierten Pfad von $Init_M(x)$ in einem Keller zu merken, und das könnte Platz $2^{\Omega(f(n))}$ beanspruchen. Statt dessen benutzen wir eine ausgeklügelte "Mittelpunktssuche".

Wir definieren eine rekursive

**function** Erreichbar($K_1, K_2$ : Konfigurationen; $j$ : integer) : boolean;

($*$ Erreichbar$(K_1, K_2, j) \Leftrightarrow K_1 \overset{M}{\rightarrow}^{\leq 2^j} K_2$ $*$)

$K_1 \overset{M}{\rightarrow}^{\leq 2^j} K_2$ heißt, daß $K_2$ von $K_1$ aus in höchstens $2^j$ Schritten zu erreichen ist.

Hauptprogramm:
Acc := *false*;
**for all** $K \in V_M(x)$ **do**
  **if** $K$ ist akzeptierend **and** Erreichbar$(Init_M(x), K, c'f(n))$ ($*$ $c'$ passende Konstante $*$)
  **then** Acc := *true*;
**if** Acc **then** Akzeptiere $x$;

**function** Erreichbar$(K_1, K_2, j)$:
  **if** $j = 0$ **then** return$((K_1 = K_2) \vee (K_1 \overset{M}{\rightarrow} K_2))$ **else**
    Gefunden := *false*;
    **for all** $K \in V_M(x)$ **do**
      ($*$ überprüfe, ob $K$ die mittlere Konfiguration sein kann $*$)
      **if** Erreichbar$(K_1, K, j-1) \wedge$ Erreichbar$(K, K_2, j-1)$ **then**
        Gefunden := *true*;
    return(Gefunden);

Die maximale Rekursionstiefe bei einer Eingabe $x$ mit $|x| = n$ ist $\leq c'f(n) = O(f(n))$. Jeder Rekursionsabschnitt (auf dem Keller) braucht Platz für die Argumente $K_1$, $K_2$ und $j$ sowie für die lokale Variable $K$, zusammen $O(f(n))$ Platz. Der Gesamtplatzverbrauch ist somit $O((f(n))^2)$. ∎

Ist $C$ eine Klasse von Sprachen, definieren wir co$C$ als die Klasse $\{L \mid \overline{L} \in C\}$ der komplementären Sprachen. Die Definition ist insofern ungenau, als wir das Alphabet nicht erwähnen, über dem die Komplementbildung stattfinden soll. Das ist aber immer egal für unsere Zwecke. Alternativ kann man sich vorstellen, daß $C$ binär kodiert ist.

**Satz** 3.4: Für alle $f : \mathbb{N}_0 \rightarrow \mathbb{N}$ ist TIME$(f)$ = coTIME$(f)$.

**Beweis:** Sei $M$ eine TM, die eine Sprache $L$ in Zeit $O(f)$ erkennt. Eine TM, die $\overline{L}$ in Zeit $O(f)$ erkennt, bekommt man, indem man einfach die Zustände `accept` und `reject` bei $M$ vertauscht (vgl. Aufgabe 1.2). ∎

Wir möchten als nächstes ein analoges Ergebnis für Platz statt Zeit beweisen. Allerdings gibt es ein kleines Problem, nämlich daß eine TM eine Eingabe $x$ verwerfen kann, indem sie einfach nicht anhält (wieso war das oben kein Problem?). In diesem Fall würde eine wie im Beweis von Satz 3.4 gebastelte Maschine $x$ ebenfalls verwerfen, was nicht korrekt wäre. Wir müssen erkennen können, daß die vorliegende Maschine auf einer Eingabe $x$ nie halten wird und wir $x$ deshalb akzeptieren können. Dazu müssen wir nicht das Halteproblem lösen, denn wir reden von platzbeschränkten Maschinen.

**Satz** 3.5: Ist $f : \mathbb{N}_0 \to \mathbb{N}$ konstruierbar und $f(n) \geq \log n$ für alle $n \in \mathbb{N}$, ist $\mathrm{SPACE}(f) = \mathrm{coSPACE}(f)$.

**Beweis:** Sei $M$ eine TM, die eine Sprache $L$ in Platz $O(f)$ erkennt. Da $f$ konstruierbar ist und $f(n) \geq \log n$, können wir zu einer vorliegenden Eingabe $x$ mit $|x| = n$ eine obere Schranke $T$ für die Größe von $V_M(x)$ berechnen, so daß $T = 2^{O(f(n))}$. Wir simulieren jetzt $M$ auf der Eingabe $x$ und zählen gleichzeitig die Anzahl der simulierten Schritte. Erreicht diese Anzahl je $T$, können wir sicher sein, daß sich $M$ in einer Schleife befindet, und wir akzeptieren $x$. Sonst akzeptieren wir $x$ genau dann, wenn $x$ von $M$ verworfen wird. Der Zähler muß bis $T$ zählen können und kann deswegen binär in Platz $O(f(n))$ repräsentiert werden. ∎

Es war lange ein offenes Problem, ob auch nichtdeterministische Platzklassen gegenüber Komplementbildung abgeschlossen sind. Die Beantwortung dieser Frage wurde 1995 mit dem Gödel-Preis gewürdigt. Für Zeit statt Platz ist die Frage bis heute unbeantwortet (da gibt es mindestens noch einen Gödel-Preis zu verdienen!).

**Satz** 3.6 (Immerman/Szelepcsényi): Sei $f : \mathbb{N}_0 \to \mathbb{N}$ konstruierbar und $f(n) \geq \log n$ für alle $n \in \mathbb{N}$. Dann ist $\mathrm{NSPACE}(f) = \mathrm{coNSPACE}(f)$.

**Beweis:** Sei $M$ eine NTM, die eine Sprache $L$ in Platz $O(f)$ erkennt. Gesucht ist eine NTM $M'$, die $\overline{L}$ in Platz $O(f)$ erkennt.

Sei $x$ eine Eingabe mit $|x| = n$, sei $V = V_M(x)$ und sei $A \subseteq V$ die Teilmenge der akzeptierenden Konfigurationen. Wir berechnen zunächst eine obere Schranke $T$ für $|V|$

mit $T = 2^{O(f(n))}$. Für $t = 0, \ldots, T$ sei $V_t$ die Menge der Konfigurationen in $V$, die von $Init_M(x)$ aus in höchstens $t$ Schritten erreicht werden können, und sei $m_t = |V_t|$. Wegen

$$M' \text{ akzeptiert } x \Leftrightarrow M \text{ verwirft } x \Leftrightarrow K \notin V_T \text{ für alle } K \in A$$

reicht es, nichtdeterministisch (also mit Hilfe von Raten) entscheiden zu können, daß eine vorliegende Konfiguration *nicht* zu $V_T$ gehört. Beachten Sie genau, daß die umgekehrte Entscheidung, daß eine vorliegende Konfiguration zu $V_T$ gehört, nichtdeterministisch sehr leicht zu treffen ist—man rät einfach den Weg dorthin.

*Idee:* Wir testen $K \notin V_T$, indem wir $m_T$ verschiedene Konfigurationen in $V_T$ raten, die alle von $K$ verschieden sind.

**function** Erreichbar($t$ : integer; $K_1, \ldots, K_r$ : Konfigurationen) : boolean;
  (* Eingabe: $t \in \mathbb{N}_0$, $K_1, \ldots, K_r \in V$; $r = O(1)$;
    Ausgabe: Ist $\{K_1, \ldots, K_r\} \cap V_t \neq \emptyset$ ? *)
  Anzahl := 0;
  Gefunden := *false*;
  **for all** $K \in V$ **do** (* teste, ob $K \in V_t$ *)
    Rate nichtdeterministisch eine Berechnung $Init_M(x) \xrightarrow{M} J_1 \xrightarrow{M} \cdots \xrightarrow{M} J_s$ der Länge $s \leq t$;
    **if** $J_s = K$ **then** Anzahl := Anzahl + 1;
    **if** $J_s \in \{K_1, \ldots, K_r\}$ **then** Gefunden := *true*;
  **if** Anzahl $< m_t$ (* schlecht geraten *)
  **then** halte ohne Akzept (* das Ergebnis ist unzuverlässig und wird weggeworfen *)
  **else** return(Gefunden);

Liefert Erreichbar den Wert *true*, ist sicherlich $\{K_1, \ldots, K_r\} \cap V_t \neq \emptyset$. Liefert Erreichbar den Wert *false*, wurden $m_t$ verschiedene Konfigurationen in $V_t$ gefunden, die alle nicht in $\{K_1, \ldots, K_r\}$ liegen; also ist dann $\{K_1, \ldots, K_r\} \cap V_t = \emptyset$. Falls die Maschine anhält, ohne ein Ergebnis zu produzieren, wissen wir nichts, aber nach Definition einer NTM beeinflußt eine solche erfolglose Berechnung das Endergebnis nicht. Da es immer möglich ist, richtig zu raten und damit auch ein Ergebnis zu bekommen, können wir so tun, als würde Erreichbar immer ein (korrektes) Ergebnis zurückliefern.

Die Funktion Erreichbar kann auf einer NTM in Platz $O(f(n))$ implementiert werden. Wir brauchen im wesentlichen nur Platz für konstant viele Konfigurationen sowie für Zähler, die über $V$ laufen.

Aber: Die Zahl $m_t$ wurde als bekannt vorausgesetzt.

**function** $m_t$ : integer;
  **if** $t = 0$ **then** return(1) **else**
    $m := 0$;
    **for all** $K \in V$ **do**
      $\{K_1, \ldots, K_r\} :=$ {Direkte Vorgängerkonfigurationen von $K$};
        (* davon gibt es nur konstant viele *)
      **if** Erreichbar$(t-1, K, K_1, \ldots, K_r)$ **then** $m := m + 1$;
    return($m$);

Den Wert von $m_0$ kennen wir. Im allgemeinen können wir, wenn wir den Wert von $m_{t-1}$ kennen, auf den Wert von $m_t$ schließen. Das erfordert einen Aufruf von Erreichbar$(t-1, \ldots)$, der aber glücklicherweise nur $m_{t-1}$ braucht. So können wir uns hochhangeln, ohne jemals mehr als einen unvollendeten Aufruf von Erreichbar zu haben (wir dürfen keinen Rekursionskeller anlegen!) und ohne mehr als einen Wert $m_i$ aufzuheben (sie können groß sein!). Schließlich kennen wir $m_T$ und können die Aufrufe von Erreichbar ausführen, die uns eigentlich interessieren.

Kann die Berechnung zu Ende geführt werden, ist sie zuverlässig. Also gilt

$$M' \text{ akzeptiert } x \Rightarrow x \notin L.$$

Umgekehrt ist es möglich, so zu raten, daß die Berechnung zu Ende geführt werden kann. Also gilt auch

$$x \notin L \Rightarrow M' \text{ akzeptiert } x. \quad \blacksquare$$

# 4 Hierarchiesätze

Frage: Um wieviel müssen Ressourceschranken vergrößert werden, damit wir echt mehr berechnen können? Beispiel: Ist SPACE($n^2$) die gleiche Klasse wie SPACE($n$), oder größer?

**Satz** 4.1 (Hierarchiesatz für deterministischen Platz): Seien $f, g : \mathbb{N}_0 \to \mathbb{N}$, sei $g$ konstruierbar und gelte $f(n)/g(n) \to 0$ für $n \to \infty$. Dann ist SPACE($f$) $\subset$ SPACE($g$).

**Bemerkung**: Der Satz sagt also, daß wir mehr in Platz $g$ als in Platz $f$ erkennen können, wenn $g$ "echt schneller" als $f$ wächst.

**Beweis**: Aus Satz 3.1 wissen wir, daß $g(n) = \Omega(\log n)$. Wir konstruieren eine Sprache $L \subseteq \{0,1\}^*$ in SPACE($g$), die von keiner $f$-platzbeschränkten TM erkannt werden kann. $L$ wird von der folgenden TM $\overline{U}$ erkannt:

Sei $x \in \{0,1\}^*$ die Eingabe und $n = |x|$;

Markiere $g(n)$ Felder auf einem Arbeitsband (hier benutzen wir die Konstruierbarkeit von $g$); akzeptiere $x$, sollte der Kopf jemals versuchen, diesen Bereich zu verlassen;

Überprüfe, ob $x$ von der Form $1^*0\langle M_x\rangle$ ist, wobei $\langle M_x\rangle$ die binär kodierte Form einer TM $M_x$ ist (das geht in Platz $O(\log n) = O(g(n))$); sonst verwirf $x$;

Simuliere $M_x$ auf der Eingabe $x$ auf dem markierten Bandbereich (benutze die Simulation aus dem Beweis von Satz 2.2), aber höchstens $2^{g(n)}$ Schritte lang;

Kann die Simulation durchgeführt werden (sie benutzt weder mehr als $g(n)$ Platz noch mehr als $2^{g(n)}$ Zeit) und akzeptiert sie $x$, dann verwirf $x$, sonst akzeptiere $x$ (dies ist ein *Diagonalisierungsargument*).

Es ist klar, daß $L \in$ SPACE($g$), denn wir haben es sorgfältig vermieden, mehr als $O(g(n))$ Platz zu benutzen. Wir müssen zeigen, daß $L \notin$ SPACE($f$).

Nehmen wir an, es gäbe eine TM $M$, die $L$ in Platz $O(f)$ erkennt. Dann kann die Berechnung von $M$ auf einer Eingabe $x$ mit $|x| = n$ durch die obige TM $\overline{U}$ ebenfalls in Platz $O(f(n))$ simuliert werden (wie im Beweis von Satz 2.2, aber Vorsicht: Die simulierende

Maschine $\overline{U}$ muß mit einem festen Bandalphabet $\{0,1\}$ auskommen, so daß das viel größere Alphabet aus dem Beweis von Satz 2.2 binär in mehreren Feldern repräsentiert werden muß. Auch hat $\overline{U}$ eine feste Zustandsmenge, so daß sie sich den Zustand von $M$ sowie die Symbole unter den Köpfen nicht in der endlichen Kontrolle, sondern auf einem Stück Arbeitsband (der Länge $O(1)$) merken muß). Wegen der Voraussetzung $\lim_{n\to\infty} f(n)/g(n) = 0$ gibt es damit ein $n_0 \in \mathbb{N}$, so daß die Simulation von $M$ auf allen Eingaben der Länge $\geq n_0$ auf den markierten Bandbereich paßt und, falls sie hält, dies nach höchstens $2^{g(n)}$ Schritten tut. Betrachte jetzt die Eingabe $x = 1^{n_0}0\langle M\rangle$. Da $|x| \geq n_0$, kann die Simulation bei Eingabe $x$ zu Ende geführt werden, oder sie hält nicht. Also ist $x \in L$ genau dann, wenn $x$ von $M$ abgelehnt wird. Aber das ist ein Widerspruch zur Annahme, daß $M$ die Sprache $L$ erkennt. ∎

Was passiert eigentlich in dem Beweis?

Wir möchten für zwei Komplexitätsklassen $C_1$ und $C_2$ zeigen, daß $C_1 \subset C_2$. Die Voraussetzungen sind so gewählt, daß $C_2$ einen universellen Simulator $U$ für $C_1$ enthält, der eine Eingabe $x$ als eine Maschine $M_x$ interpretiert und $M_x$ auf der Eingabe $x$ simuliert ($x \in L(U) \Leftrightarrow M_x$ akzeptiert $x$). Außerdem ist $U$ mit einer "Kontrollvorrichtung" versehen, die verhindert, daß $U$ mehr Ressourcen als für $C_2$ zulässig verbraucht. Daher ist $L(U) \in C_2$. *Ist jetzt $C_2$ abgeschlossen unter Komplementbildung*, ist auch $\overline{L(U)} \in C_2$. Aber $\overline{L(U)} \notin C_1$, denn würde $\overline{L(U)}$ von einer $C_1$-Maschine $M$ akzeptiert werden, würde $M$, auf sich selbst angesetzt, genau dann akzeptieren, wenn $M$ von $M$ abgelehnt wird, ein Widerspruch.

Da nichtdeterministische Platzklassen unter Komplementbildung abgeschlossen sind (Satz 3.6), erhalten wir, analog zu Satz 4.1:

**Satz** 4.2 (Hierarchiesatz für nichtdeterministischen Platz): Seien $f, g : \mathbb{N}_0 \to \mathbb{N}$, sei $g$ konstruierbar und gelte $f(n)/g(n) \to 0$ für $n \to \infty$. Dann ist $\mathrm{NSPACE}(f) \subset \mathrm{NSPACE}(g)$.

Die Sätze 4.1 und 4.2 besagen, daß wir mit mehr Platz, und sei es noch so wenig, auch mehr Sprachen erkennen können. Man könnte das für einleuchtend halten, aber es gibt Fälle, wo diese Intuition versagt, wo wir also mit mehr Platz überhaupt nichts anfangen können. Zum Beispiel ist $\mathrm{SPACE}(\log\log\log n) = \mathrm{SPACE}(1)$ (Aufgabe 5.3).

Für Zeit statt Platz sind nur weniger dichte Hierarchien bekannt:

**Satz** 4.3 (Hierarchiesatz für deterministische Zeit): Seien $f, g : \mathbb{N}_0 \to \mathbb{N}$, sei $g$ konstruierbar und $g(n) \geq n$ für alle $n \in \mathbb{N}_0$, und es gelte $(f(n))^2/g(n) \to 0$ für $n \to \infty$. Dann ist $\mathrm{TIME}(f) \subset \mathrm{TIME}(g)$.

**Beweis:** Aufgabe 5.1. ∎

Die Hierarchiesätze konstruieren zwar Sprachen, die die betrachteten Komplexitätsklassen voneinander "trennen", aber es sind künstliche Sprachen ohne praktische Bedeutung. In den meisten Fällen ist man nicht in der Lage, natürliche trennende Sprachen anzugeben (obwohl das sehr interessant wäre).

# 5 Reduktionen und vollständige Probleme

Bisher haben wir uns mit abstrakten Komplexitätsklassen wie TIME($f$) beschäftigt. Jetzt wollen wir konkrete Klassen wie P und NP genauer betrachten und zum Beispiel vollständige Probleme darin suchen.

$$\mathrm{L} = \mathrm{SPACE}(\log n) \qquad \mathrm{NL} = \mathrm{NSPACE}(\log n)$$
$$\mathrm{P} = \bigcup_{k=1}^{\infty} \mathrm{TIME}(n^k) \qquad \mathrm{NP} = \bigcup_{k=1}^{\infty} \mathrm{NTIME}(n^k)$$
$$\mathrm{PSPACE} = \bigcup_{k=1}^{\infty} \mathrm{SPACE}(n^k) \qquad \mathrm{NPSPACE} = \bigcup_{k=1}^{\infty} \mathrm{NSPACE}(n^k)$$
$$\mathrm{EXP} = \bigcup_{k=1}^{\infty} \mathrm{TIME}(2^{n^k}) \qquad \mathrm{NEXP} = \bigcup_{k=1}^{\infty} \mathrm{NTIME}(2^{n^k})$$
$$\mathrm{EXPSPACE} = \bigcup_{k=1}^{\infty} \mathrm{SPACE}(2^{n^k}) \qquad \mathrm{NEXPSPACE} = \bigcup_{k=1}^{\infty} \mathrm{NSPACE}(2^{n^k})$$

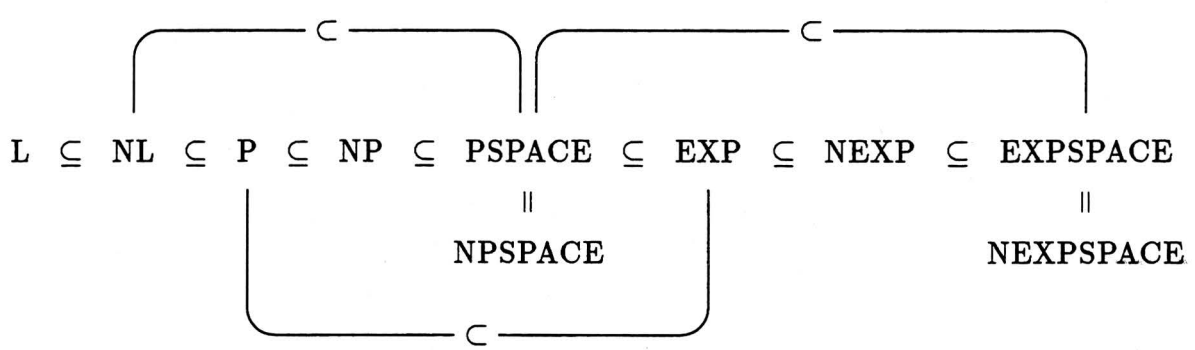



Fig. 5.1. Relationen zwischen Komplexitätsklassen.

Die Relationen aus Fig. 5.1 werden in Aufgabe 6.1 bewiesen. Alle Inklusionsrelationen zwischen den abgebildeten Klassen, die sich nicht aus dem Diagramm ergeben, sind unbekannt, außer daß auch NP $\subset$ NEXP (für den recht komplizierten Beweis siehe J. I. Seiferas, M. J. Fischer and A. R. Meyer, Separating nondeterministic time complexity classes, *J. ACM* **25** (1978), pp. 146–167).

Wenn wir ein Problem derart in ein anderes überführen können, daß eine Lösung für das zweite Problem zugleich eine Lösung für das erste Problem liefert, sagen wir, daß wir das erste Problem auf das zweite *reduziert* haben. Intuitiv folgt daraus, daß das erste Problem nicht schwieriger als das zweite ist; allerdings stimmt das nur, falls die Reduktion zwischen den Problemen nicht zu aufwendig ist—es sollte erheblich leichter sein, die Reduktion anzuwenden, als das erste Problem direkt zu lösen. Es gibt verschiedene Typen von Reduktionen, wie zum Beispiel Polynomialzeitreduktion. Die letztere ist allerdings ein zu grobes Instrument, um sinnvolle Reduktionen zwischen Problemen in P zu gestatten (jedes Problem ist trivial auf jedes andere reduzierbar), und daher benutzen wir hier noch einfachere Reduktionen, *logspace-Reduktionen*, die mit logarithmischem Arbeitsplatz ausgeführt werden können. Da eine Berechnung, die mit logarithmischem Arbeitsplatz auskommen muß, höchstens polynomiell lange laufen kann, ohne in eine Schleife zu geraten, sind logspace-Reduktionen in der Tat zumindest nicht mächtiger als Polynomialzeitreduktionen.

**Definition:** Seien $\Sigma_1$ und $\Sigma_2$ Alphabete. Eine Sprache $L_1 \subseteq \Sigma_1^*$ ist *(logspace-)reduzierbar* auf eine Sprache $L_2 \subseteq \Sigma_2^*$ (das schreiben wir manchmal als $L_1 \leq L_2$), wenn es eine Funktion $R : \Sigma_1^* \to \Sigma_2^*$ gibt, die auf Eingaben der Länge $n$ von einer TM mit $O(\log n)$ Arbeitsplatz berechnet wird, so daß

$$x \in L_1 \iff R(x) \in L_2.$$

Warum gerade logarithmischer Platz? Nun, in logarithmischem Platz können wir konstant viele Zahlen repräsentieren, die polynomiell in der Eingabegröße sind und die zum Beispiel benutzt werden können, um über die Eingabe zu iterieren ("**for** $i := 1$ **to** $n$"). Damit können wir schon recht frei rechnen. Man stelle sich zum Beispiel vor, die Eingabe stehe in einem read-only Array, und wir dürfen sie mit einem beliebigen Pascal-Programm ohne Rekursion und ohne zusätzliche Arrays bearbeiten, schon ein recht mächtiges Berechnungsmodell. Mit weniger als logarithmischem Platz hingegen können wir so gut wie gar nichts machen (vgl. Satz 3.1). Logarithmischer Platz ist also einerseits sozusagen das absolute Minimum, das andererseits aber auch ausreicht.

Wenn wir die Relation $\leq$ als "ist nicht schwieriger als" interpretieren, sollte sie natürlich transitiv sein. Das wollen wir jetzt zeigen.

**Satz** 5.1: $\leq$ ist eine transitive Relation.

**Beweis:** Wir müssen zeigen, daß $L_1 \leq L_2$ und $L_2 \leq L_3$ zusammen $L_1 \leq L_3$ implizieren. Gegeben sind also Sprachen $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$ und $L_3 \subseteq \Sigma_3^*$ sowie logspace-Funktionen $R : \Sigma_1^* \to \Sigma_2^*$ und $S : \Sigma_2^* \to \Sigma_3^*$, so daß $x \in L_1 \Leftrightarrow R(x) \in L_2$ für alle $x \in \Sigma_1^*$ und $y \in L_2 \Leftrightarrow S(y) \in L_3$ für alle $y \in \Sigma_2^*$, und wir müssen eine logspace-Funktion $T : \Sigma_1^* \to \Sigma_3^*$ angeben, so daß $x \in L_1 \Leftrightarrow T(x) \in L_3$ für alle $x \in \Sigma_1^*$.

Wie wäre es mit $T = S \circ R$ (Fig. 5.2)? Für alle $x \in \Sigma_1^*$ ist sicherlich

$$x \in L_1 \iff R(x) \in L_2 \iff S(R(x)) \in L_3 \iff T(x) \in L_3.$$

Es bleibt also nur die Frage, ob $T$ in logarithmischem Platz berechnet werden kann.

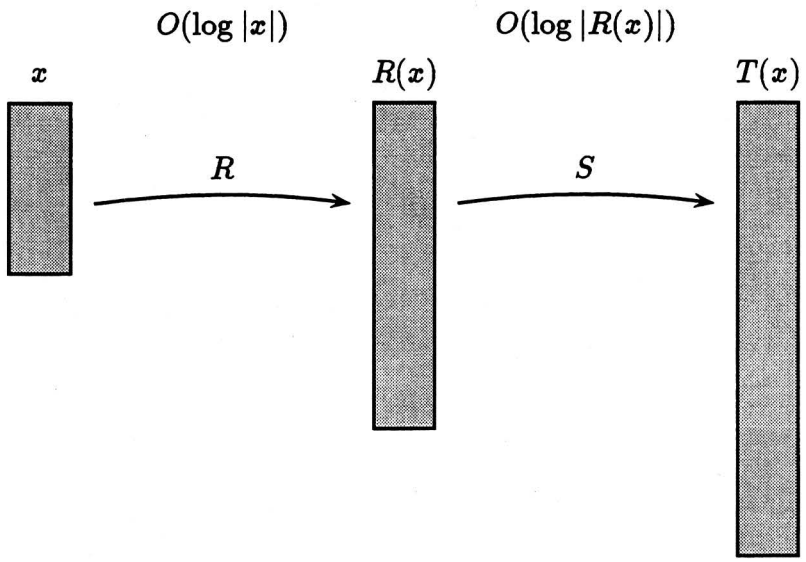



Fig. 5.2. Hintereinanderschaltung von Reduktionen.

Wieso ist das keine Trivialität? Immerhin ist die Laufzeit und damit die Ausgabegröße von $R$ polynomiell beschränkt, so daß $\log |R(x)| = O(\log |x|)$. Damit müßten wir zuerst $R(x)$ und danach $S(R(x)) = T(x)$ berechnen können.

Das (einzige) Problem dabei ist, daß wir keinen Platz haben, um $R(x)$ zwischenzuspeichern. Wir haben nur logarithmischen Platz, und $R(x)$ ist nicht mehr Ausgabe (deren Größe nach Definition nicht mitgezählt wird), sondern ein Zwischenergebnis. Lösung: Wir starten die Berechnung von $S$ auf der Eingabe $R(x)$, ohne $R(x)$ vorher berechnet zu haben! Wann immer die Berechnung ein Symbol von $R(x)$ abfragen möchte, z.B. das $i$te, starten wir die Berechnung von $R(x)$, werfen aber die gesamte Ausgabe weg, bis das $i$te Symbol

produziert wird, mit dem wir dann die Berechnung von $S(R(x))$ fortsetzen. Brauchen wir das $i$te Symbol von $R(x)$ später wieder, wird es von neuem berechnet. So kann $T(x)$ in logarithmischem Platz berechnet werden. ∎

**Definition:** Sei $C$ eine Komplexitätsklasse. Eine Sprache $L$ ist *$C$-hart* (unter logspace-Reduktionen), falls $L' \leq L$ für alle $L' \in C$. $L$ ist *$C$-vollständig* bzw. *vollständig für $C$* (unter logspace-Reduktionen), falls $L$ $C$-hart ist und $L \in C$.

Intuitiv: $C$-vollständige Probleme sind die schwierigsten Probleme in $C$, denn können wir ein $C$-vollständiges Problem lösen, können wir jedes andere Problem in $C$ auch lösen, wobei wir nur den vernachlässigbaren zusätzlichen Aufwand für die Reduktion in Kauf nehmen müssen. Es gibt Komplexitätsklassen, für die keine vollständigen Probleme bekannt sind. Allerdings kann man oft künstliche vollständige Problem wie im folgenden Beispiel konstruieren.

**Beispiel** 5.2: Sei $L = \{(\langle M \rangle, x, I^t) \mid M$ ist eine NTM, die mindestens eine akzeptierende Berechnung auf $x$ der Länge höchstens $t$ hat$\}$, wobei $\langle M \rangle$ für eine Zeichenfolge steht, die die Maschine $M$ kodiert. Wir behaupten, daß $L$ NP-vollständig ist. Dazu sind zwei Dinge zu zeigen, nämlich erstens, daß $L \in$ NP, und zweitens, daß $L' \leq L$ für jedes $L' \in$ NP.

Warum ist $L \in$ NP? Nun, erstens ist es leicht, in Polynomialzeit zu überprüfen, daß die Eingabe tatsächlich von der Form $(\langle M \rangle, x, I^t)$ ist (sonst können wir sie gleich ablehnen), und zweitens können wir nichtdeterministisch eine Berechnung $\pi$ von $M$ auf $x$ der Länge höchstens $t$ raten und die Eingabe akzeptieren, falls $\pi$ akzeptierend ist. Wird $x$ von $M$ innerhalb von $t$ Schritten akzeptiert, so gibt es eine Berechnung $\pi$, die das "bezeugt", und einer unserer nichtdeterministischen Versuche wird $\pi$ ausprobieren und akzeptieren; andernfalls wird kein Versuch akzeptieren, und die Eingabe wird abgelehnt.

Sei jetzt $L' \in$ NP. Dann gibt es also eine NTM $M$, die $L'$ akzeptiert und auf Eingaben der Länge $n$ in Zeit höchstens $p(n)$ läuft, wobei $p$ ein Polynom (mit Koeffizienten aus $\mathbb{N}_0$) ist. Sei $R(x) = (\langle M \rangle, x, I^{p(|x|)})$. $R$ kann in logarithmischem Platz berechnet werden, und

$$x \in L' \iff x \text{ wird von } M \text{ akzeptiert}$$

$$\iff x \text{ wird von } M \text{ in höchstens } p(|x|) \text{ Schritten akzeptiert} \iff R(x) \in L.$$

Damit ist also $R$ eine logspace-Reduktion, die $L'$ auf $L$ reduziert.

Solche unnatürlichen vollständigen Probleme sind selten besonders nützlich. Wir wollen als nächstes die NP-Vollständigkeit eines viel natürlicheren Problems zeigen.

Eine Boolesche Formel in *konjunktiver Normalform (CNF)*:

$$(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3) \wedge (\overline{x_3} \vee x_4)$$

Also: Eine große Konjunktion von *Klauseln*. Jede Klausel ist die Disjunktion von *Literalen*, und jedes Literal ist entweder eine Variable, $x_i$, oder die Negation davon, $\overline{x_i}$.

$k$-CNF ($k \in \mathbb{N}$): Jede Klausel besteht aus höchstens $k$ Literalen.

Eine Boolesche Formel über die Variablen $x_1, \ldots, x_n$ heißt *erfüllbar*, wenn es (mindestens) eine Belegung

$$\phi : \{x_1, \ldots, x_n\} \to \{true, false\}$$

gibt, die die Formel wahr macht. Es ist leicht, Boolesche Formeln als Zeichenfolgen zu kodieren (wir schreiben sie ja im wesentlichen schon in dieser Form). SAT ist das Problem, zu einer vorliegenden Booleschen Formel in CNF zu entscheiden, ob sie erfüllbar ist, also

$$\text{SAT} = \{F \mid F \text{ ist eine erfüllbare Boolesche Formel in CNF}\}.$$

**Satz** 5.3 (Cook): SAT ist NP-vollständig.

**Beweis:** SAT gehört zu NP. Denn erstens ist es leicht, in Polynomialzeit zu überprüfen, ob die Eingabe $x$ tatsächlich eine Boolesche Formel $F$ in CNF kodiert. Ist das so, können wir zweitens nichtdeterministisch eine Belegung der in $F$ vorkommenden Variablen raten und überprüfen, ob sie $F$ erfüllt; wenn ja, akzeptieren wir die Eingabe. Ist $x \in \text{SAT}$, wird $x$ von einer Berechnung akzeptiert, die gerade eine erfüllende Belegung rät; ist $x \notin \text{SAT}$, gibt es keine solche Belegung, und $x$ wird von keiner Berechnung akzeptiert.

Wir kommen jetzt zum schwierigeren Teil, nämlich zu zeigen, daß $L \leq \text{SAT}$ für beliebiges $L \in \text{NP}$. Zu einer vorliegenden Eingabe $x$ müssen wir also eine Boolesche Formel $F$ konstruieren, die genau dann erfüllbar ist, wenn $x \in L$.

Sei $M$ eine NTM, die $L$ akzeptiert und deren Laufzeit und Platzverbrauch auf Eingaben der Größe $n$ durch $p(n)$ beschränkt ist, wobei $p(n) \geq n$ ein Polynom ist. Wir können o.B.d.A. davon ausgehen, daß $M$ nur ein Band hat (Bemerkung 2.4) und daß sich alle Nachfolgekonfigurationen jeder festen Konfiguration nur in ihrem Zustand unterscheiden (möchte die Maschine z.B. nichtdeterministisch wählen, entweder den Kopf stehenzulassen

oder ihn nach rechts zu bewegen, erreichen wir dies, indem die Maschine zuerst in einen von zwei neuen Zuständen wechselt, der dann anschließend die gewünschte Kopfbewegung deterministisch erzwingt).

Mit $k \approx \log(|Q||\Sigma|)$ Booleschen Variablen können wir für ein bestimmtes Bandfeld zu einem bestimmten Zeitpunkt einer Berechnung von $M$ die folgende Information kodieren:

- Das Symbol in dem betrachteten Feld;
- Ob der Kopf sich auf diesem Feld befindet und, falls ja,
- den Zustand von $M$.

Wir wählen die Kodierung so, daß eine ausgezeichnete *Akzeptvariable* unter den $k$ Variablen genau dann den Wert *true* hat, wenn der Kopf sich auf dem betrachteten Feld befindet und der Zustand von $M$ **accept** ist. Die $k$ Variablen fassen wir in einem Vektor $X$ zusammen. Für jede Eingabe $x$ der Länge $n$ können wir mit $p(n)$ solchen Vektoren, einem für jedes Feld, die gesamte Konfiguration von $M$ zu einem beliebigen Zeitpunkt während der Berechnung auf $x$ beschreiben, da nie mehr als $p(n)$ Felder benutzt werden. Mit $p(n)+1$ solchen Reihen von Vektoren, einer für jeden Zeitpunkt, können wir die gesamte Berechnung beschreiben. Insgesamt haben wir $p(n)(p(n)+1)$ Vektoren von jeweils $k$ Variablen eingeführt, die man sich am besten als in einem zweidimensionalen Feld angeordnet vorstellt (Fig. 5.3): Der Vektor $X_{i,j}$ in der $i$ten Reihe und $j$ten Spalte, für $0 \leq i \leq p(n)$ und $1 \leq j \leq p(n)$, beschreibt den Inhalt des $j$ten Feldes, sowie möglicherweise den Zustand, nach $i$ Schritten der Berechnung.

Eine beliebige Belegung der insgesamt $kp(n)(p(n)+1)$ Booleschen Variablen entspricht vielleicht einer gültigen Berechnung von $M$ auf $x$, vielleicht auch nicht. Unser Ziel ist, eine CNF-Formel niederzuschreiben, die ausdrückt, daß die Berechnung gültig und akzeptierend ist.

Zunächst modifizieren wir die Maschine so, daß jede akzeptierende Konfiguration sich selbst als Nachfolgekonfiguration hat—damit stellen wir sicher, daß es eine erfüllende Belegung aller Variablen geben wird, auch wenn die Maschine nach weniger als $p(n)$ Schritten akzeptiert. Jetzt ist es leicht zu überprüfen, ob eine gültige Berechnung akzeptierend ist, nämlich durch eine große Disjunktion über alle Akzeptvariablen der letzten Zeile; das ist unsere erste Klausel.

Die Variablen der ersten Zeile müssen alle bestimmte Werte haben (*true* oder *false*), damit die erste Zeile tatsächlich die Konfiguration $Init_M(x)$ kodiert. Das stellen wir durch triviale Klauseln sicher; $x_i$, falls $x_i$ den Wert *true* haben soll, und $\overline{x_i}$, falls $x_i$ den Wert *false* haben soll.

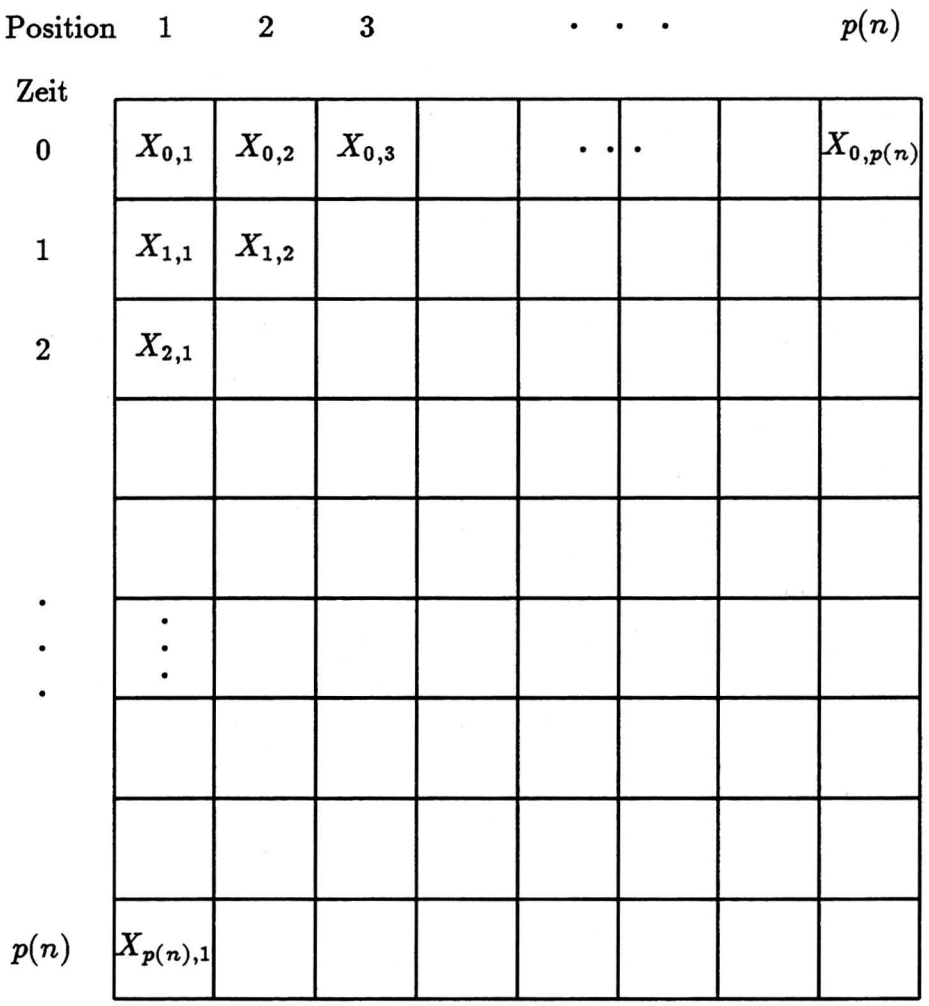



Fig. 5.3. Die Berechnung einer TM.

Jetzt müssen wir nur noch sicherstellen, daß jede Zeile korrekt aus der vorherigen Zeile hervorgeht, also entsprechend der (modifizierten) Übergangsfunktion $\delta$. Aber das ist genau dann der Fall, wenn jeder Vektor $X_{i,j}$, für $i = 1, \ldots, p(n)$ und $j = 1, \ldots, p(n)$, korrekt aus den drei Vektoren $X_{i-1,j-1}$, $X_{i-1,j}$ und $X_{i-1,j+1}$ aus der vorherigen Zeile hervorgeht, wobei $X_{i-1,j-1}$ entfällt, falls $j = 1$, und $X_{i-1,j+1}$ entfällt, falls $j = p(n)$. Das ist keine Trivialität: hier benutzen wir, daß nach Voraussetzung nur der Zustand nichtdeterministisch unterschiedlich festgelegt werden kann—das betrifft nur ein Feld und erfordert somit keine "Koordination" zwischen Feldern. Aber die vier Vektoren $X_{i,j}$, $X_{i-1,j-1}$, $X_{i-1,j}$ und $X_{i-1,j+1}$ umfassen insgesamt nur $4k = O(1)$ Elementarvariablen, so daß wir die korrekte Ableitung sicherlich durch eine Boolesche Formel konstanter Größe ausdrücken können. Ferner können wir diese Formel in konjunktiver Normalform schreiben, ohne daß

die konstante Größe verloren geht (Aufgabe 6.2). Das gibt uns für jeden Vektor $X_{i,j}$ mit $i = 1,\ldots,p(n)$ und $j = 1,\ldots,p(n)$ eine Anzahl neuer Klauseln. Die Verknüpfung aller eingeführten Klauseln ergibt eine Formel $F$ in CNF, die genau dann erfüllbar ist, wenn $x$ von $M$ akzeptiert wird, und $F$ kann problemlos mit logarithmischem Arbeitsplatz erstellt werden. ∎

Mit einem fast identischen Beweis können wir die P-Vollständigkeit eines anderen Problems nachweisen.

Ein *Schaltkreis* (s. Fig. 5.4) ist ein azyklischer gerichteter Graph mit Knoten von Ingrad 2, jeder beschriftet mit $\wedge$ oder $\vee$, und Knoten von Ingrad 1, beschriftet mit $\neg$. Knoten ohne Vorgänger (Ingrad 0) sind entweder mit Konstanten (*true* oder *false* bzw. 0 oder 1) oder mit Variablen $(x_1, x_2, \ldots)$ beschriftet, und es gibt einen ausgezeichneten *Ausgangsknoten.* Die Knoten eines Schaltkreises heißen *Gatter*, die Kanten heißen *Drähte.* Werden konkrete Werte (*true* oder *false*) für die Variablen substituiert, berechnet jedes Gatter einen Wert: Gatter ohne Vorgänger "berechnen" die entsprechende Konstante, und alle anderen Gatter wenden die Funktion, mit der sie beschriftet sind, auf die Werte an, die von ihren Vorgängern berechnet werden. Der Wert des Schaltkreises ist der Wert des Ausgangsgatters.

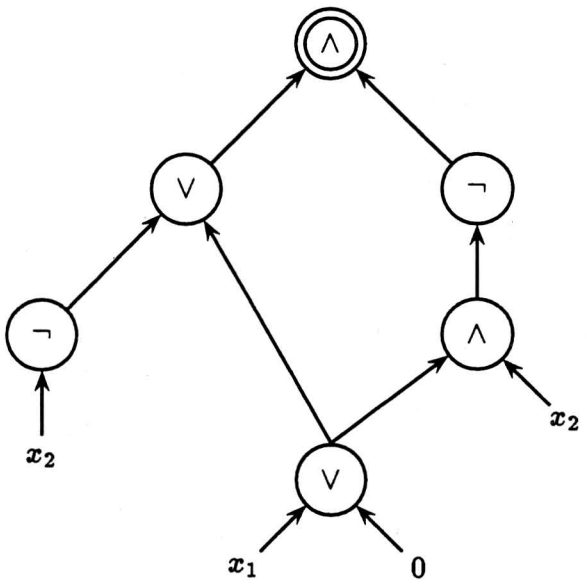
Schaltkreise können wie andere Graphen als Zeichenfolgen kodiert werden. CVP (Circuit-Value-Problem) ist das Problem, den Wert eines Schaltkreises ohne Variablen zu bestimmen, also

$$\text{CVP} = \{\gamma \mid \gamma \text{ ist ein Schaltkreis ohne Variablen mit dem Wert } \textit{true}\}.$$

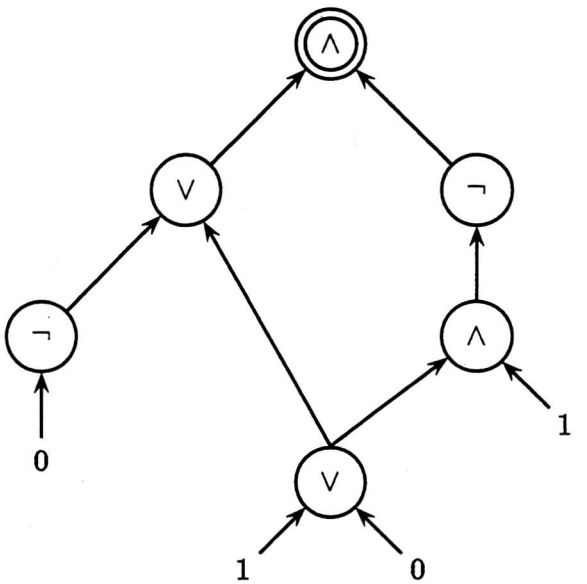
**Satz** 5.4: CVP ist P-vollständig.

**Beweis:** Es ist leicht zu sehen, daß ein Schaltkreis in polynomieller Zeit ausgewertet werden kann—dazu bedarf es im wesentlichen einer topologischen Sortierung. Wir müssen also nur noch zeigen, daß $L \leq \text{CVP}$ für alle $L \in \text{P}$.

Sei $M$ eine 1-Band TM, die $L$ akzeptiert und deren Laufzeit und Platzverbrauch auf Eingaben der Größe $n$ durch $p(n)$ beschränkt ist, wobei $p(n) \geq n$ ein Polynom ist. Wir benutzen im wesentlichen die Konstruktion aus dem Beweis von Satz 5.3, ersetzen aber jede Variable in der ersten Zeile durch den entsprechenden Wert (*true* oder *false*) und jede Variable in einer anderen Zeile durch ein Gatter. Da $M$ deterministisch arbeitet, ergibt sich der korrekte Wert eines jeden Gatters in Zeile $i \geq 1$ als Funktion der Werte von




Schaltkreis mit Variablen




Schaltkreis ohne Variablen

Fig. 5.4. Schaltkreise.

konstant vielen Gattern in Zeile $i-1$. Wir stellen diese Funktion mit Hilfe von zusätzlichen Gattern als Schaltkreis dar und identifizieren die Eingangsgatter des Schaltkreises mit den entsprechenden Gattern in Zeile $i-1$ und das Ausgangsgatter mit dem betrachteten Gatter

in Zeile $i$ (wir konstruieren mit anderen Worten "Hardware", die Zeile $i$ aus Zeile $i-1$ berechnen kann). Schließlich verbinden wir die $p(n)$ ausgezeichneten Ausgangsgatter in der letzten Zeile über einen Baum von $p(n)-1$ $\vee$-Gattern und wählen die Wurzel des Baums als Ausgangsgatter. So erhalten wir einen Schaltkreis ohne Variablen, dessen Wert genau dann *true* ist, wenn die Maschine $M$ ihre Eingabe akzeptiert. Der Schaltkreis kann mit logarithmischem Arbeitsplatz erstellt werden. ∎

Es mag nützlich sein, sich an dieser Stelle zu überlegen, warum wir nicht gerade gezeigt haben, daß CVP NP-vollständig ist.

# 6 Weitere NP-vollständige Probleme

In diesem Kapitel zeigen wir die NP-Vollständigkeit einiger weiterer Probleme, die wir später (Kap. 11 und 12) in anderem Zusammenhang brauchen werden. Der Einfachheit halber werden wir diese nicht mehr als Sprachen, sondern als Ja/Nein-Probleme formulieren. Also nicht mehr

$$\text{SAT} = \{F \mid F \text{ ist eine erfüllbare Boolesche Formel in CNF}\},$$

sondern

SAT:

Eingabe: Eine Boolesche Formel in CNF.

Frage: Ist $F$ erfüllbar?

Eine *unabhängige (Knoten-)Menge* in einem ungerichteten Graphen $G = (V, E)$ ist eine Menge $I \subseteq V$ mit der Eigenschaft, daß für keine Kante in $E$ beide Endpunkte in $I$ liegen. Die Knoten in $I$ sind also nicht benachbart (Fig. 6.1).

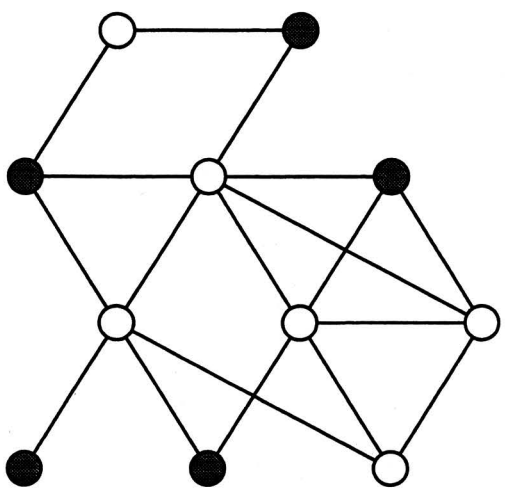


Fig. 6.1. Knoten einer unabhängigen Menge.

Wir betrachten jetzt das Graphenproblem

INDEPENDENT SET:

Eingabe: Ein ungerichteter Graph $G$ und eine Zahl $k \in \mathbb{N}$.

Frage: Gibt es in $G$ eine unabhängige Knotenmenge $I$ der Größe $\geq k$?

**Bemerkung:** Natürlicher wäre es, nach der größten unabhängigen Menge zu fragen, aber vorläufig beschränken wir uns auf *Entscheidungsprobleme*, die nur Ja/Nein-Antworten haben. NP enthält nach Definition auch nur solche Probleme (nämlich eigentlich nur Sprachen). In Kap. 11 werden wir mehr über *Optimierungsprobleme* zu sagen haben.

**Satz** 6.1: INDEPENDENT SET ist NP-vollständig.

**Beweis:** Zunächst ist klar, daß INDEPENDENT SET $\in$ NP: Wir können die Menge $I$ nichtdeterministisch raten und leicht überprüfen, ob sie unabhängig ist.

Wir müssen jetzt zeigen, daß $L \leq$ INDEPENDENT SET für alle $L \in$ NP. Da wir aber schon wissen, daß SAT NP-vollständig ist, reicht es zu zeigen, daß SAT $\leq$ INDEPENDENT SET, denn wir wissen bereits, daß $L \leq$ SAT, und der Rest folgt aus der Transitivität von $\leq$ (Satz 5.1). Mit anderen Worten: Um zu zeigen, daß ein Problem in NP vollständig für NP ist, reicht es, ein schon als NP-vollständig bekanntes Problem darauf zu reduzieren, was meistens erheblich leichter ist.

Zu einer vorliegenden Formel $F$ in CNF müssen wir also einen ungerichteten Graphen $G$ und eine Schranke $k \in \mathbb{N}$ berechnen, so daß $G$ genau dann eine unabhängige Menge der Größe $\geq k$ enthält, wenn $F$ erfüllbar ist. Aber das ist nicht schwierig: $G$ enthält einen Knoten für jedes Vorkommen eines Literals in $F$ (kommt z.B. das Literal $x_1$ in mehreren Klauseln vor, gibt es einen Knoten für jedes Vorkommen). Zwei Knoten in $G$, die Vorkommen von Literalen $\alpha$ und $\beta$ repräsentieren, sind genau dann durch eine Kante verbunden, wenn die beiden Vorkommen zur gleichen Klausel gehören oder wenn $\alpha$ und $\beta$ von der Form $x_i$ und $\overline{x_i}$ sind, also nicht gleichzeitig wahr sein können (Fig. 6.2). $k$ wählen wir als die Anzahl der Klauseln in $F$.

Wenn $G$ eine unabhängige Menge $I$ der Größe $\geq k$ hat, muß $I$ genau einen Knoten aus jeder Klausel enthalten. Geben wir dem entsprechenden Literal den Wert *true*, handeln wir uns keine Inkonsistenzen ein, denn Literale, die nicht gleichzeitig wahr sein können, sind immer durch eine Kante verbunden, gehören also nicht beide zu $I$. Geben wir Variablen, die dadurch nicht festgelegt werden, beliebige Werte, erhalten wir eine Belegung, die $F$ erfüllt. Gibt es umgekehrt eine solche Belegung, können wir aus jeder Klausel ein Literal

$$(x_1 \vee x_2 \vee x_3 \vee x_4) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_4)$$

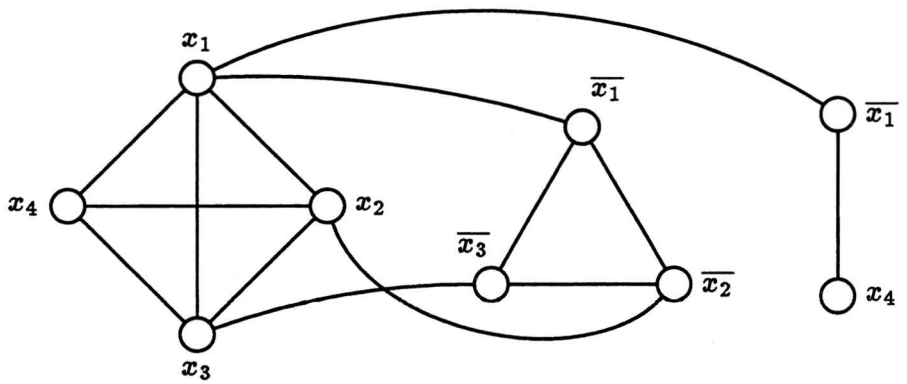



Fig. 6.2. Eine Boolesche Formel und der dazugehörige Graph.

auswählen, das den Wert *true* hat, und den entsprechenden Knoten in $I$ tun. Die so entstehende Menge $I$ ist unabhängig, denn wir können nicht sowohl $x_i$ als auch $\overline{x_i}$ auswählen, und ihre Größe ist genau $k$. Das Paar $(G, k)$ hat also die geforderten Eigenschaften, und es kann mit logarithmischem Arbeitsplatz berechnet werden. ∎

Wir betrachten als nächstes zwei Probleme, die eigentlich nur Varianten von INDEPENDENT SET sind. Eine *Clique* bzw. ein *vollständiger Teilgraph* in einem ungerichteten Graphen $G = (V, E)$ ist ein Teilgraph $H = (U, E')$ von $G$, der sämtliche $\binom{|U|}{2}$ mögliche Kanten enthält. Ein *Vertex Cover* bzw. eine *überdeckende Knotenmenge* in $G$ ist eine Menge $U \subseteq V$ mit der Eigenschaft, daß jede Kante in $E$ mindestens einen Endpunkt in $U$ hat.

CLIQUE:

Eingabe: Ein ungerichteter Graph $G$ und eine Zahl $k \in \mathbb{N}$.

Frage: Enthält $G$ eine Clique mit $\geq k$ Knoten?

VERTEX COVER:

Eingabe: Ein ungerichteter Graph $G$ und eine Zahl $k \in \mathbb{N}$.

Frage: Enthält $G$ ein Vertex Cover der Größe $\leq k$?

**Satz** 6.2: CLIQUE und VERTEX COVER sind NP-vollständig.

**Beweis:** Sei $G = (V, E)$ ein ungerichteter Graph und sei $\overline{G} = (V, \overline{E})$ sein Komplement ($\overline{G}$ enthält genau diejenigen Kanten, die nicht in $G$ vorkommen). Dann induziert eine Knotenmenge $U \subseteq V$ genau dann eine Clique in $G$, wenn $U$ eine unabhängige Menge in $\overline{G}$ ist. Also

enthält $G$ genau dann eine Clique mit mindestens $k$ Knoten, wenn $\overline{G}$ eine unabhängige Knotenmenge der Größe $\geq k$ enthält, woraus folgt, daß CLIQUE $\leq$ INDEPENDENT SET (und umgekehrt).

Ferner ist $U$ genau dann ein Vertex Cover in $G$, wenn $V \backslash U$ eine unabhängige Menge in $G$ ist. Also gibt es genau dann in $G$ ein Vertex Cover der Größe $\leq k$, wenn $G$ eine unabhängige Knotenmenge der Größe $\geq |V| - k$ enthält, woraus folgt, daß VERTEX COVER $\leq$ INDEPENDENT SET (und umgekehrt). Der Beweis wird in Aufgabe 8.3 abgeschlossen. ∎

Ein *Hamilton-Kreis* in einem gerichteten oder ungerichteten Graphen $G = (V, E)$ ist ein Teilgraph von $G$, der ein einfacher Kreis ist (also ohne Knotenwiederholungen) und alle Knoten in $V$ enthält.

DIRECTED HAMILTON CYCLE:

Eingabe: Ein gerichteter Graph $G = (V, E)$.

Frage: Gibt es einen Hamilton-Kreis in $G$?

**Satz** 6.3: DIRECTED HAMILTON CYCLE ist NP-vollständig.

**Beweis:** Wir reduzieren SAT auf DIRECTED HAMILTON CYCLE. Gegeben ist also eine Boolesche Formel $F$ in CNF mit $m$ Variablen $x_1, \ldots, x_m$ und $r$ Klauseln $C_1, \ldots, C_r$, und wir müssen einen gerichteten Graphen $G$ erstellen, der genau dann einen Hamilton-Kreis besitzt, wenn $F$ erfüllbar ist. Für jede Variable $x_i$ konstruieren wir zunächst ein "Gadget" (Gerät, Vorrichtung) $G_i$ wie links in Fig. 6.3 dargestellt, wobei die Anzahl der $b_{i,j}$-Knoten die Anzahl der Vorkommen von $x_i$ und $\overline{x_i}$ um 1 übersteigen soll.

Wird $G_i$ nur über die Knoten $a_i$ und $d_i$ mit dem restlichen Graphen verbunden, sieht man leicht, daß ein Hamilton-Kreis nur die zwei rechts in Fig. 6.3 gezeigten Möglichkeiten hat, $G_i$ zu durchlaufen. Wir interpretieren diese als "$x_i = \mathit{true}$" und "$x_i = \mathit{false}$".

$G_1, \ldots, G_m$ werden, wie in Fig. 6.4 gezeigt, zu einem Kreis verbunden. Wir haben jetzt offensichtlich genau einen Hamilton-Kreis für jede Belegung $\phi : \{x_1, \ldots, x_m\} \to \{\mathit{true}, \mathit{false}\}$ und müssen weiter an dem Graphen basteln, um zu erreichen, daß nur diejenigen Belegungen Hamilton-Kreisen entsprechen, die jede Klausel erfüllen.

Hierzu benutzen wir für jede Klausel $C_l$ eine Kopie $H_l$ des in Fig. 6.5 dargestellten Gadgets, wobei die Anzahl der Spalten (3 im Beispiel) als die Anzahl der Literale in der Klausel gewählt wird. Eine wichtige Eigenschaft dieses Gadgets ist, daß ein beliebiger Hamilton-Kreis jedesmal, wenn er das Gadget betritt, es in der gleichen Spalte wieder verlassen muß (ist $u_i$ der erste Knoten dieses Betretens, muß $v_i$ der letzte sein)—sonst

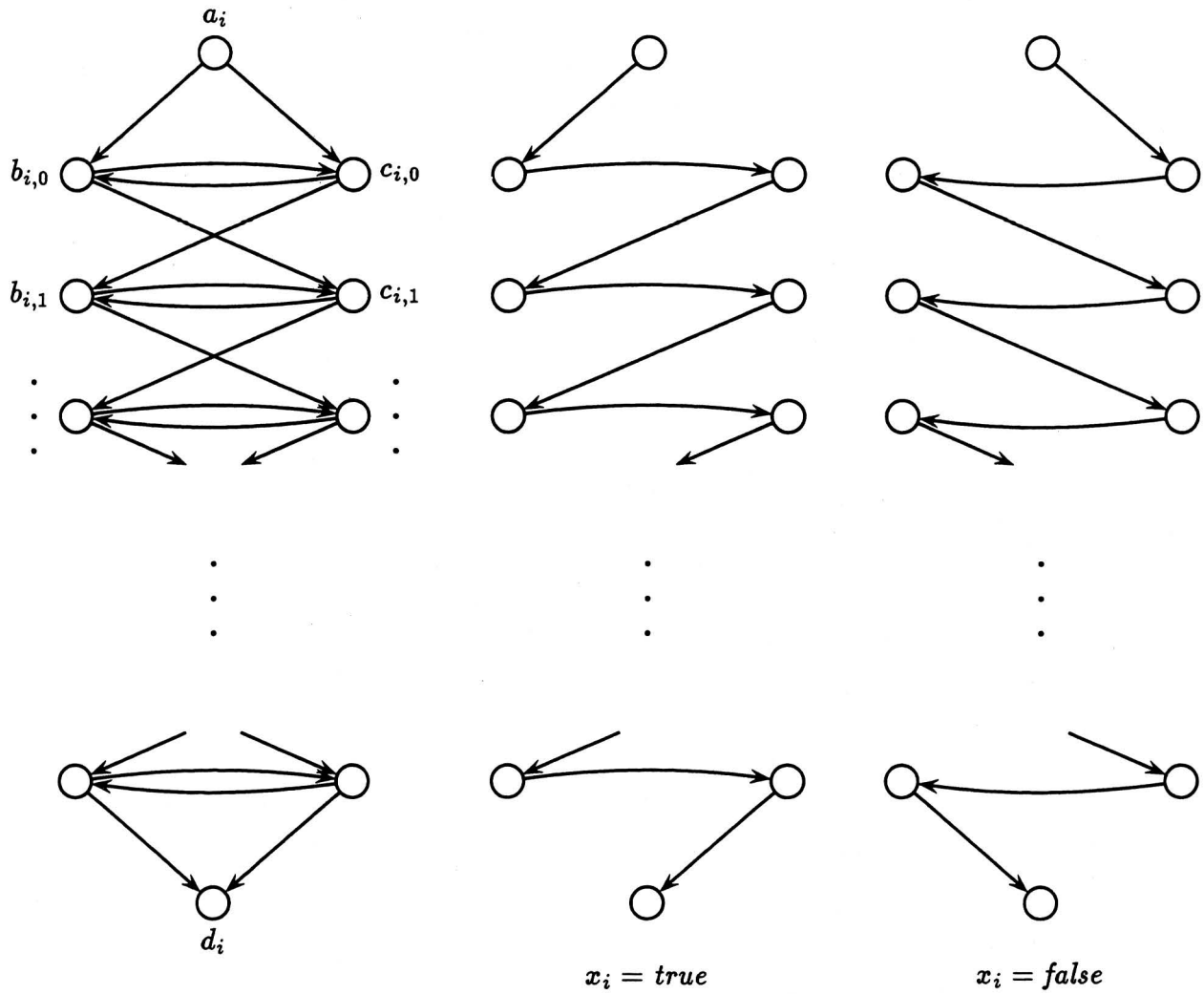



Fig. 6.3. Der Graph $G_i$ und zwei Wege durch ihn.

gibt es nämlich einen unbenutzten Knoten, dessen sämtliche entweder Vorgänger oder Nachfolger benutzt werden, so daß der Knoten auch später nicht vom Kreis betreten werden kann. Enthält eine Klausel $C_l$ das Literal $x_i$, wird eine Spalte von $H_l$ in das Gadget $G_i$ für $x_i$ "eingeschleust", und zwar zwischen $b_{i,j}$ und $c_{i,j+1}$ (es werden also z.B. Kanten $(b_{i,j}, u_1)$ und $(v_1, c_{i,j+1})$ eingeführt, wobei $u_1$ und $v_1$ Knoten aus $H_l$ sind, aber die Kante $(b_{i,j}, c_{i,j+1})$ verbleibt im Graphen). Der Wert von $j$ ist hierbei beliebig, außer daß er von keinem anderen Vorkommen von $x_i$ oder $\overline{x_i}$ mitbenutzt werden soll (hierfür haben wir $G_i$ "lang" genug gemacht). Enthält $C_l$ $\overline{x_i}$, verbinden wir statt dessen eine Spalte von $H_l$ zwischen $c_{i,j}$ und $b_{i,j+1}$.

Ein Hamilton-Kreis muß natürlich jedes Klauselgadget $H_l$ besuchen. Das kann nur

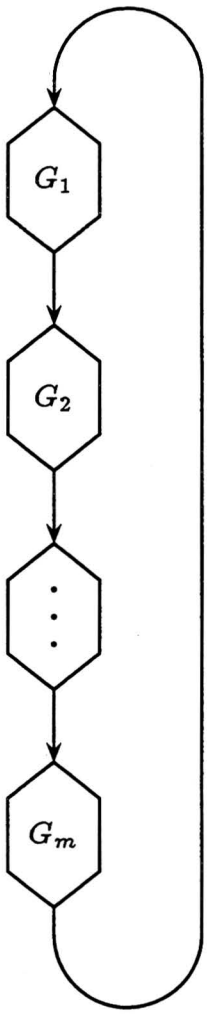



Fig. 6.4. Der Kreis der Graphen $G_1, \ldots, G_m$.

so geschehen, daß der Kreis auf dem Weg durch das Gadget von mindestens einer der in $C_l$ vorkommenden Variablen den Weg durch $H_l$ an Stelle der direkten Kante wählt. Das ist aber nur dann möglich, wenn diese Kante tatsächlich benutzt werden würde, wenn die Klauselgadgets nicht da wären, was wiederum genau dann der Fall ist, wenn das betreffende Literal den Wert *true* hat. Also kann es einen Hamilton-Kreis nur dann geben, wenn $F$ erfüllbar ist, und umgekehrt gibt es auch zu jeder erfüllenden Belegung mindestens einen Hamilton-Kreis. ∎

UNDIRECTED HAMILTON CYCLE:

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

Frage: Gibt es einen Hamilton-Kreis in $G$?

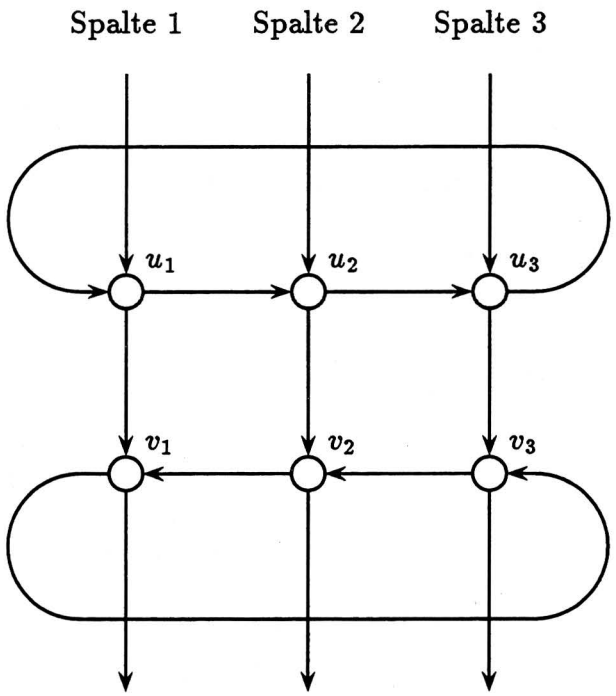



Fig. 6.5. Ein Klauselgadget für drei Literale.

**Satz** 6.4: UNDIRECTED HAMILTON CYCLE ist NP-vollständig.

**Beweis**: Aufgabe 8.1. ∎

Unter Benutzung von UNDIRECTED HAMILTON CYCLE können wir leicht die NP-Vollständigkeit des Traveling-Salesman-Problems (TSP) nachweisen.

TSP:

Eingabe: Ein vollständiger ungerichteter Graph $G$, in dem jede Kante mit einer *Länge* aus $\mathbb{N}$ beschriftet ist, sowie eine Zahl $k \in \mathbb{N}$.

Frage: Gibt es eine *Tour* (einen einfachen Kreis, der alle Knoten besucht) in $G$, deren Gesamtlänge durch $k$ beschränkt ist?

**Satz** 6.5: TSP ist NP-vollständig.

**Beweis**: Zu einer gegebenen Instanz von UNDIRECTED HAMILTON CYCLE, also einem ungerichteten Graphen $G = (V, E)$, erstellen wir eine Instanz von TSP folgendermaßen: Der Graph ist der vollständige Graph auf der Knotenmenge $V$, jede Kante wird mit 1 beschriftet, wenn sie in $E$ liegt, sonst mit 2, und wir nehmen $k = |V|$. Trivialerweise hat das so

definierte TSP-Problem genau dann eine Lösung, wenn $G$ einen Hamilton-Kreis besitzt. Also ist UNDIRECTED HAMILTON CYCLE $\leq$ TSP. ∎

Wir betrachten jetzt eingeschränkte Varianten von SAT:

$k$SAT ($k \in \mathbb{N}$):

Eingabe: Eine Boolesche Formel $F$ in $k$-CNF.

Frage: Ist $F$ erfüllbar?

**Satz** 6.6: 3SAT ist NP-vollständig, aber 2SAT $\in$ P.

**Beweis:** Aufgaben 7.1 und 7.3. ∎

Jetzt kommen wir zu einer etwas komischen Variante von 3SAT, die aber manchmal nützlich ist.

NAESAT (not-all-equal-SAT):

Eingabe: Eine Boolesche Formel $F$ in 3-CNF.

Frage: Gibt es eine Belegung der in $F$ vorkommenden Variablen, die in jeder Klausel von $F$ mindestens ein Literal *true* und mindestens ein Literal *false* setzt?

**Satz** 6.7: NAESAT ist NP-vollständig.

**Beweis:** Wir reduzieren 3SAT auf NAESAT. Gegeben ist also eine Formel $F$ in 3-CNF, und wir müssen eine andere Formel $F'$ konstruieren, so daß $F \in$ SAT $\Leftrightarrow$ $F' \in$ NAESAT. Durch Wiederholen von Literalen können wir annehmen, daß jede Klausel von $F$ aus genau drei Literalen besteht. Wir erhalten $F'$, indem wir jede Klausel $(\alpha_1 \vee \alpha_2 \vee \alpha_3)$ durch die sieben Klauseln

$$(\overline{\alpha_1} \vee x \vee z) \wedge (\overline{\alpha_2} \vee x \vee z) \wedge (\alpha_1 \vee \alpha_2 \vee \overline{x}) \wedge (\overline{x} \vee y \vee z) \wedge (\overline{\alpha_3} \vee y \vee z) \wedge (x \vee \alpha_3 \vee \overline{y}) \wedge (y \vee z)$$

ersetzen, wobei $x$ und $y$ neue Variablen sind, die nur für diese Klausel benutzt werden, während $z$ eine neue Variable ist, die allen Klauseln gemeinsam ist.

Sei $F \in$ SAT und betrachte eine erfüllende Belegung, die insbesondere die Klausel $(\alpha_1 \vee \alpha_2 \vee \alpha_3)$ erfüllt. Wir suchen eine Belegung mit den gleichen Wahrheitswerten für $\alpha_1$, $\alpha_2$ und $\alpha_3$, die den entsprechenden Teil von $F'$ im Sinne von NAESAT erfüllt. Dazu nehmen wir $x = \alpha_1 \vee \alpha_2$, $y =$ *true* und $z =$ *false*. Das erfüllt trivialerweise die drei Klauseln, die $y \vee z$ enthalten, und man sieht leicht, daß auch die übrigen vier Klauseln erfüllt sind. Also ist $F' \in$ NAESAT.

Sei nun umgekehrt $F' \in$ NAESAT. Dann gibt es eine im Sinne von NAESAT erfüllende Belegung, und die "umgekehrte" Belegung, bei der *true* und *false* vertauscht werden, ist auch im Sinne von NAESAT erfüllend. Wir wählen die Belegung, bei der $z$ den Wert *false* hat, und behaupten, daß diese Belegung, eingeschränkt auf die Variablen, die in $F$ vorkommen, $F$ erfüllt. Betrachten wir dazu eine bestimmte Klausel $(\alpha_1 \vee \alpha_2 \vee \alpha_3)$ und die daraus abgeleiteten sieben Klauseln. Da $z = \mathit{false}$, muß $y = \mathit{true}$, woraus wiederum folgt, daß $x = \mathit{true}$ oder $\alpha_3 = \mathit{true}$. Ist $\alpha_3 = \mathit{true}$, sind wir fertig, und sonst ist $\alpha_1 = \mathit{true}$ oder $\alpha_2 = \mathit{true}$. ∎

Eine *(Knoten-)Färbung* eines ungerichteten Graphen mit $k$ Farben ist eine Beschriftung der Knoten des Graphen mit Elementen aus $\{1, \ldots, k\}$ ("Farben"), so daß keine zwei benachbarten Knoten die gleiche Beschriftung tragen.

Mit Hilfe von NAESAT können wir leicht zeigen, daß es NP-vollständig ist zu entscheiden, ob ein vorliegender Graph mit $k$ Farben gefärbt werden kann, wobei $k$ Teil der Eingabe ist. Genauer können wir es sogar für $k = 3$ zeigen ($k$ ist nicht mehr Teil der Eingabe, sondern fest, wodurch das Problem nur leichter werden kann). Für $k = 1, 2$ liegt das Problem in P, was Sie wahrscheinlich ohne größere Schwierigkeiten einsehen können.

$k$-COLORING ($k \in I\!N$):

Eingabe: Ein ungerichteter Graph $G$.

Frage: Gibt es eine Färbung von $G$ mit $k$ Farben?

**Satz** 6.8: 3-COLORING ist NP-vollständig.

**Beweis**: Wir reduzieren NAESAT auf 3-COLORING. Gegeben ist also eine Formel $F$ in 3-CNF, und wir müssen einen Graphen $G$ konstruieren, der genau dann 3-färbbar ist, wenn $F$ im Sinne von NAESAT erfüllbar ist. Durch Wiederholung von Literalen können wir annehmen, daß jede Klausel von $F$ genau drei Literale enthält.

Für jede in $F$ vorkommende Variable $x_i$ nehmen wir ein Dreieck (einen vollständigen Graphen mit 3 Knoten), dessen Knoten mit $x_i$, $\overline{x_i}$ und $a$ beschriftet sind, wobei der mit $a$ beschriftete Knoten allen diesen Dreiecken gemeinsam ist. Für jede Klausel $C = (\alpha_1 \vee \alpha_2 \vee \alpha_3)$ nehmen wir auch ein Dreieck, und wir verbinden die Knoten dieses Dreiecks mit den Knoten der Variablendreiecke, die den Literalen von $C$ entsprechen (Fig. 6.6).

Betrachten wir eine Färbung von $G$ mit den Farben 1, 2 und 3. O.B.d.A. (durch Vertauschen der Farben) können wir annehmen, daß der mit $a$ beschriftete Knoten die Farbe 3 trägt. Für jede Variable $x_i$ müssen die mit $x_i$ und $\overline{x_i}$ beschrifteten Knoten verschiedene

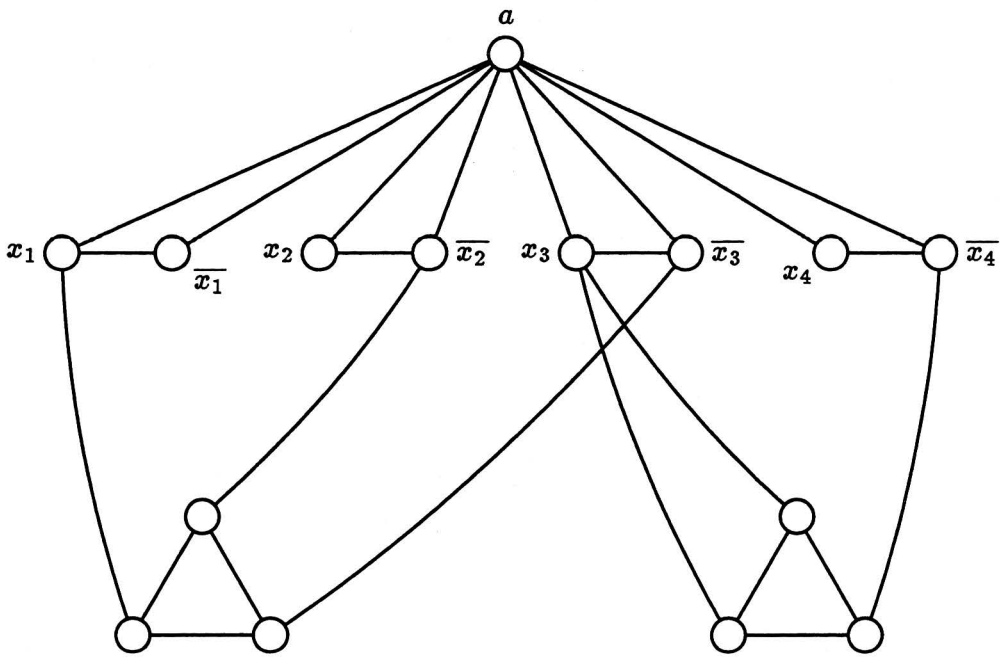



Fig. 6.6. Die Reduktion von NAESAT auf 3-COLORING.

Farben tragen, und zwar 1 und 2. Wir interpretieren 1 als *true* und 2 als *false*. Jede Klausel muß ein wahres und ein falsches Literal haben, denn sonst steht eine der drei Farben für die Färbung des entsprechenden Dreiecks nicht zur Verfügung. Also ist $F$ im Sinne von NAESAT erfüllbar. Umgekehrt ist es leicht, von einer im Sinne von NAESAT erfüllenden Belegung zu einer 3-Färbung zu kommen. ∎

Es gibt auch eine sehr direkte Reduktion von NAESAT zum Problem, einen *maximalen Schnitt* in einem Graphen zu berechnen.

MAX-CUT:

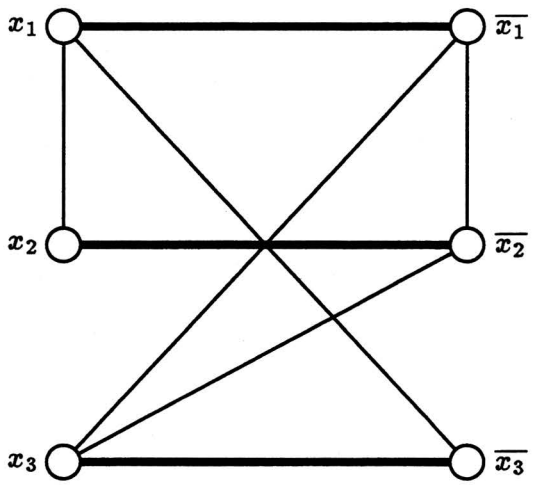
Eingabe: Ein ungerichteter Graph $G = (V, E)$, in dem jede Kante $e$ mit einer *Kapazität* $c(e) \in \mathbb{N}$ beschriftet ist, sowie eine Zahl $k \in \mathbb{N}$.

Frage: Gibt es eine Knotenmenge $U \subseteq V$ mit $\sum_{\substack{u \in U \\ v \in V \setminus U}} c(\{u, v\}) \geq k$?

**Satz** 6.9: MAX-CUT ist NP-vollständig.

**Beweis:** Wir reduzieren NAESAT auf MAX-CUT. Gegeben ist eine Formel in 3-CNF mit $m$ Variablen $x_1, \ldots, x_m$, $r_2$ 2-Klauseln (Klauseln mit zwei verschiedenen Literalen) und $r_3$ 3-

Klauseln. Wir setzen $r = r_2 + r_3$ und konstruieren einen Graphen wie folgt (Fig. 6.7): Die Knotenmenge ist $V = \{x_1, \overline{x_1}, x_2, \overline{x_2}, \ldots, x_m, \overline{x_m}\}$, und für $i = 1, \ldots, m$ gibt es eine Kante $\{x_i, \overline{x_i}\}$ mit Kapazität $3r$. Außerdem ist für jede Klausel jedes Paar von verschiedenen Literalen in der Klausel durch eine Kante mit Kapazität 1 verbunden. Wir definieren $k$ als $3rm + r_2 + 2r_3$.




$(x_1 \vee x_2) \wedge (x_1 \vee \overline{x_3}) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$

Fig. 6.7. Die Reduktion von NAESAT auf MAX-CUT.

Gegeben eine Belegung, die $F$ im Sinne von NAESAT erfüllt, können wir $U = \{\alpha \in V \mid \alpha = \mathit{true}\}$ nehmen. Dann erhält $\sum_{\substack{u \in U \\ v \in V \setminus U}} c(\{u, v\})$ einen Beitrag von $3r$ von jeder Kante $\{x_i, \overline{x_i}\}$, einen Beitrag von 1 von jeder 2-Klausel und einen Beitrag von 2 von jeder 3-Klausel, insgesamt $3rm + r_2 + 2r_3 = k$. Umgekehrt sieht man leicht, daß eine Kapazität von $k$ nur auf diese Weise zustandekommen kann, und nur entsprechend einer im Sinne von NAESAT erfüllenden Belegung. Also ist NAESAT $\leq$ MAX-CUT. ∎

Fig. 6.8 zeigt die NP-vollständigen Probleme, mit denen wir uns in diesem Kapitel beschäftigen (drei stehen noch aus), sowie die durch unsere Reduktionen auf diesen Problemen induzierte Baumstruktur.

EXACT COVER:

Eingabe: Eine Familie $\mathcal{F} = \{A_1, \ldots, A_m\}$ von Teilmengen eines Universums $U$.

Frage: Gibt es eine Teilfamilie $\mathcal{F}' \subseteq \mathcal{F}$ von disjunkten Mengen, deren Vereinigung $U$ ist?

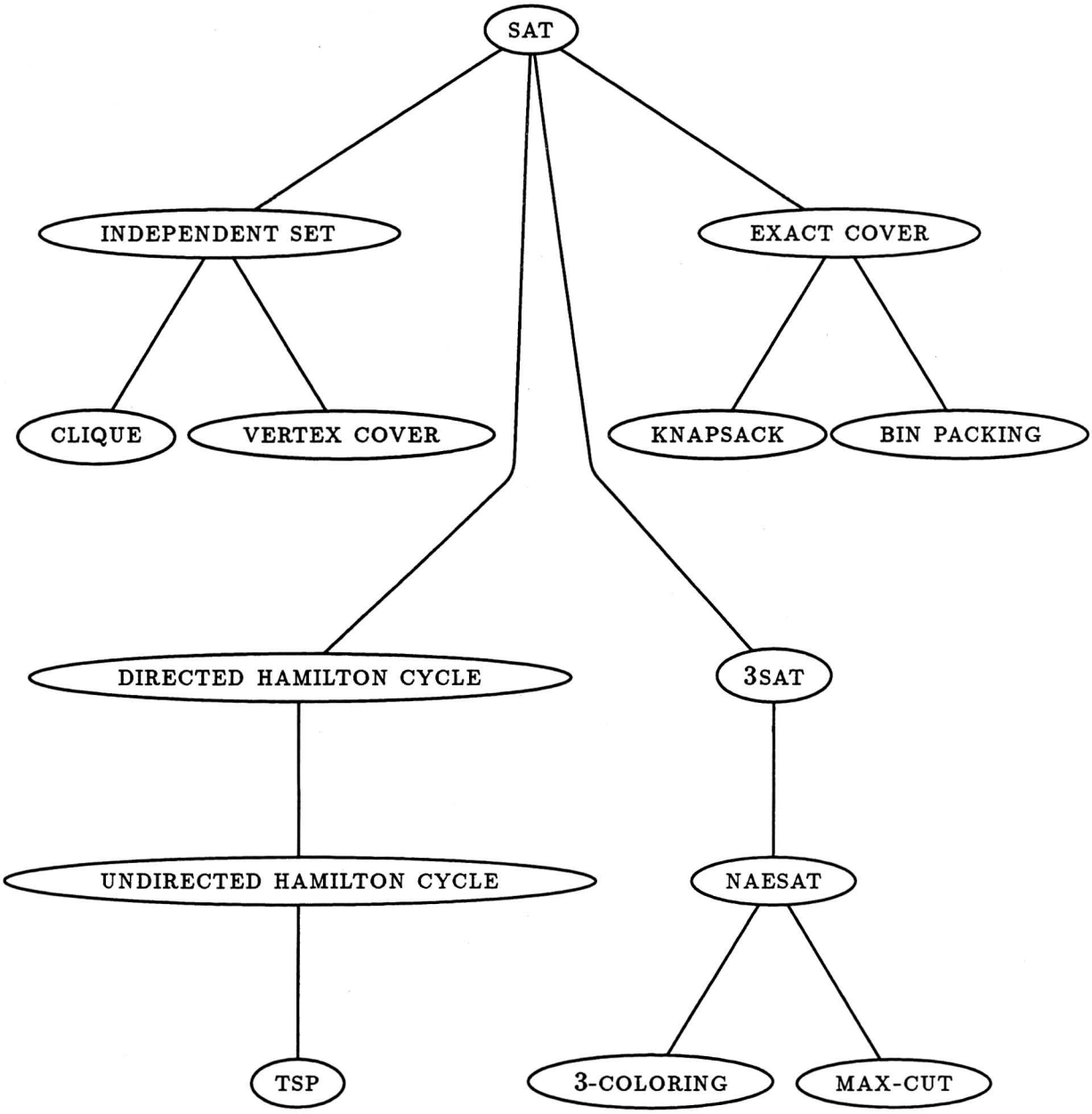



Fig. 6.8. NP-vollständige Probleme und Reduktionen zwischen ihnen.

**Beispiel:** $U = \{1,2,3,4,5\}$.

$$A_1 = \{1,2\}$$
$$A_2 = \{1,3,4\}$$
$$A_3 = \{1,4\}$$
$$A_4 = \{2,3\}$$
$$A_5 = \{2,3,5\}$$
$$A_6 = \{3\}$$

Die Antwort ist hier Ja, denn $U$ ist die disjunkte Vereinigung von $A_3$ und $A_5$.

**Satz** 6.10: EXACT COVER ist NP-vollständig.

**Beweis:** Wir reduzieren SAT auf EXACT COVER. Gegeben ist also eine Formel $F$ in CNF mit Klauseln $C_1, \ldots, C_r$. O.B.d.A. gehen wir davon aus, daß keine Klausel, die ein Literal $\alpha$ enthält, auch das Literal $\overline{\alpha}$ enthält. Unser Universum $U$ wird Elemente von zwei verschiedenen Typen enthalten: Die Elemente vom *Typ* 1 sind genau die Klauseln $C_1, \ldots, C_r$. Die Elemente vom *Typ* 2 sind alle Mengen der Form $\{C, C', \alpha, \overline{\alpha}\}$, wobei $C$ und $C'$ Klauseln in $F$ sind und $\alpha$ ein Literal ist. (Vorsicht: Wir reden von Familien von Teilmengen des Universums, aber einige Elemente des Universums sind auch schon Mengen.) Für jede Klausel $C$ und für jedes in $C$ vorkommende Literal $\alpha$ generieren wir eine Teilmenge $A_{C,\alpha}$ von $U$. $A_{C,\alpha}$ besteht aus dem Typ-1-Element $C$ sowie aus allen Typ-2-Elementen der Form $\{C, C', \alpha, \overline{\alpha}\}$, für die $C'$ das Literal $\overline{\alpha}$ enthält. Die Mengenfamilie $\mathcal{F}$ besteht aus allen solchen Mengen $A_{C,\alpha}$ sowie aus allen einelementigen Teilmengen von $U$, deren eines Element vom Typ 2 ist.

Gibt es eine erfüllende Belegung von $F$, wählen wir in jeder Klausel $C$ ein Literal $\alpha$ mit dem Wert *true* und fügen $A_{C,\alpha}$ in $\mathcal{F}'$ ein. Die Mengen in der so entstehenden Familie $\mathcal{F}'$ sind disjunkt, denn sonst hätten wir in einer Klausel $\alpha$ und in einer anderen Klausel $\overline{\alpha}$ gewählt, und ihre Vereinigung enthält alle Typ-1-Elemente aus $U$. Da alle Elemente von Typ 2 einzeln zur Verfügung stehen, können wir $\mathcal{F}'$ leicht zu einer disjunkten Überdeckung von ganz $U$ erweitern.

Gibt es umgekehrt eine solche Überdeckung $\mathcal{F}'$, muß $\mathcal{F}'$ für jede Klausel $C$ eine Menge der Form $A_{C,\alpha}$ enthalten, wobei $\alpha$ (und nicht $\overline{\alpha}$) in $C$ vorkommt, und wir können $\alpha = \mathit{true}$ setzen, da $\mathcal{F}'$ dann keine Menge der Form $A_{C',\overline{\alpha}}$ enthält, wobei $\overline{\alpha}$ (und nicht $\alpha$) in $C'$ vorkommt. Wird diese Belegung beliebig auf Variablen ausgedehnt, die noch keinen Wert haben, erhalten wir eine Belegung, die $F$ erfüllt. ∎

Das KNAPSACK-Problem formalisiert Situationen wie die folgende: Wir packen einen Rucksack für eine Wandertour. Von vornherein steht fest, daß wir nicht mehr als $W$ Kilos schleppen wollen, weswegen nicht alle in Frage kommenden Gegenstände mitgenommen werden können. Jeder Gegenstand hat einen "Wert", der angibt, wie sehr uns daran gelegen ist, ihn mitzunehmen, und wir fragen jetzt, ob es eine mögliche Auswahl an Gegenständen gibt, die einen Gesamtwert von mindestens $k$ erreicht, ohne dabei die Gewichtsgrenze zu verletzen. (Natürlich ist das eine Vereinfachung des realen Problems, bei dem vielfach zusätzliche Bindungen zwischen den Gegenständen bestehen; z.B. hat es wenig Sinn, Film mitzunehmen, wenn die Kamera zu Hause bleibt.)

KNAPSACK:

Eingabe: $2m + 2$ positive ganze Zahlen $w_1, \ldots, w_m, W, v_1, \ldots, v_m, k$.

Frage: Gibt es eine Teilmenge $I \subseteq \{1, \ldots, m\}$ mit $\sum_{i \in I} w_i \leq W$, aber $\sum_{i \in I} v_i \geq k$?

**Satz** 6.11: KNAPSACK ist NP-vollständig.

**Beweis**: Wir reduzieren EXACT COVER auf eine spezielle Variante von KNAPSACK, bei der $w_i = v_i$ für $i = 1, \ldots, m$ und $W = k$. Gegeben sind bei dieser speziellen Variante also positive Zahlen $v_1, \ldots, v_m$, und die Frage ist, ob eine geeignete Teilmenge dieser Zahlen eine vorgegebene Summe $k$ hat.

Gegeben seien jetzt Mengen $A_1, \ldots, A_m$ über einem Universum, das wir o.B.d.A. mit $U = \{0, \ldots, r-1\}$ identifizieren können. Wir können jede Menge $A_i$ als einen Bitvektor der Länge $r$ auffassen. Addieren wir einige solche Vektoren komponentenweise (Addition einzelner Komponenten erfolgt über $\mathbb{N}_0$), können wir leicht am Ergebnis ablesen, ob die entsprechenden Mengen $A_i$ disjunkt sind und die Vereinigungsmenge $U$ haben, denn dann und nur dann ist das Ergebnis der Vektor $(1, \ldots, 1)$. Das sieht fast wie KNAPSACK aus: Wir fragen, ob eine vorgegebene Summe durch geeignete Auswahl der Summanden zustande kommen kann. Nur werden beim KNAPSACK-Problem ganze Zahlen addiert, während wir es hier mit Vektoraddition zu tun haben. Wir können aber leicht die Addition von ganzen Zahlen so umfunktionieren, daß sie unsere Vektoren addiert. Dazu wird jeder Vektor $(a_{r-1}, \ldots, a_0)$ mit der ganzen Zahl $\sum_{i=0}^{r-1} a_i (m+1)^i$ identifiziert. Jetzt ist die gewünschte Summe natürlich $\sum_{i=0}^{r-1} (m+1)^i$, und da es höchstens $m$ Summanden gibt, können keine "Überträge" zwischen den $r$ "Positionen" auftreten. Wir haben gezeigt, daß EXACT COVER $\leq$ KNAPSACK. ∎

Mit fast der gleichen Reduktion können wir auch die NP-Vollständigkeit des BIN PACKING-Problems nachweisen. Wir haben hier $m$ Objekte der Größen $v_1, \ldots, v_m$ und möchten

diese in $k$ Behälter verteilen, von denen jeder die Kapazität $C$ hat.

BIN PACKING:

Eingabe: $m+2$ positive ganze Zahlen $v_1, \ldots, v_m, C, k$.

Frage: Können die Zahlen $v_1, \ldots, v_m$ in $k$ Gruppen partitioniert werden, so daß die Summe der Zahlen in jeder Gruppe $C$ nicht übersteigt?

**Satz** 6.12: BIN PACKING ist NP-vollständig.

**Beweis**: Wir reduzieren EXACT COVER auf BIN PACKING. Gegeben Mengen $A_1, \ldots, A_m$ mit $\bigcup_{i=1}^{m} A_i = \{0, \ldots, r-1\}$, konstruieren wir zunächst die gleichen $m$ Zahlen $v_1, \ldots, v_m$ der Form $\sum_{i=0}^{r-1} a_i(m+1)^i$ wie in der vorherigen Reduktion und setzen $S = \sum_{j=1}^{m} v_j$ und $N = \sum_{i=0}^{r-1}(m+1)^i \leq S$. Wir fügen zwei neue Objekte ein durch $v_{m+1} = 3S - 2N$ und $v_{m+2} = 2S$. Nun ist $\sum_{j=1}^{m+2} v_j = 6S - 2N$, und eine Partition von $v_1, \ldots, v_{m+2}$ in $k = 2$ Gruppen, jede mit der Summe $C = 3S - N$, muß notwendigerweise $v_{m+1}$ und $v_{m+2}$ trennen. Aber dann summieren die Elemente aus $v_1, \ldots, v_m$, die zusammen mit $v_{m+1}$ die Summe $C$ ergeben, exakt auf $N$, so daß die vorliegende Instanz des EXACT COVER-Problems eine Lösung hat. Umgekehrt gibt jede solche Lösung natürlich Anlaß zu einer Partition in zwei Gruppen, jede mit der Summe $C$. Damit ist EXACT COVER $\leq$ BIN PACKING. ∎

# 7 coNP und Primalitätsbeweise

NP ist die Klasse der Sprachen, die kurze "Beweise" (für die Zugehörigheit) haben (Beispiel für SAT: Eine erfüllende Belegung). Demnach ist coNP die Klasse der Sprachen, die kurze "Gegenbeweise" haben (Beispiel für $\overline{\text{SAT}}$: Eine erfüllende Belegung).

**Satz** 7.1: $\text{P} \subseteq \text{NP} \cap \text{coNP}$.

**Satz** 7.2: Falls $\text{P} = \text{NP}$ oder $\text{P} = \text{coNP}$, dann ist $\text{P} = \text{NP} = \text{coNP}$.

**Satz** 7.3: Sei $L$ eine NP-vollständige Sprache. Dann gilt $L \in \text{coNP} \Leftrightarrow \text{NP} = \text{coNP}$.

**Satz** 7.4: $L$ ist NP-vollständig $\Leftrightarrow$ $\overline{L}$ ist coNP-vollständig.

**Beweis:** Aufgabe 9.3. ∎

Nach den Sätzen 7.1–7.4 können wir die Situation wie in Fig. 7.1 zusammenfassen. Man vermutet, daß P eine echte Teilmenge von NP ∩ coNP ist, aber es ist nur für sehr wenig Probleme bekannt, daß sie zu NP ∩ coNP gehören, ohne daß man auch gleichzeitig weiß, daß sie in P liegen. Im restlichen Teil dieses Kapitels werden wir ein Problem in dieser erlesenen Gruppe kennenlernen, nämlich das Primalitätsproblem.

PRIMES:

Eingabe: Eine ganze Zahl $m \geq 3$ von $n$ Bits.

Frage: Ist $m$ eine Primzahl?

Wir können leicht in $m^{O(1)}$ Zeit überprüfen, ob $m$ eine Primzahl ist, indem wir $m$ durch alle kleineren natürlichen Zahlen teilen; aber die Problemgröße ist $n \approx \log m$, so daß die Laufzeit dieses Verfahrens exponentiell in der Problemgröße ist. Wie oben erwähnt, ist es unbekannt, ob PRIMES $\in$ P.

**Satz** 7.5 (Pratt): PRIMES $\in \text{NP} \cap \text{coNP}$.

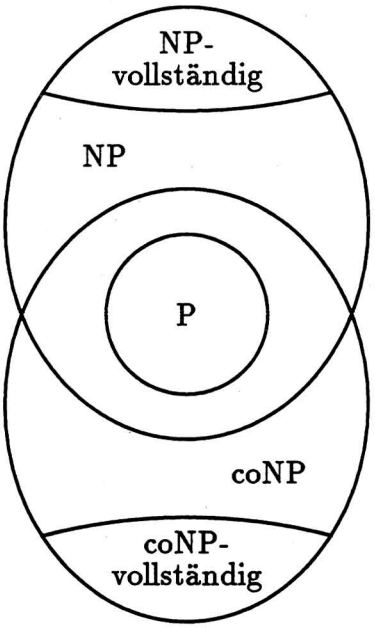



Fig. 7.1. Die Komplexitätslandschaft um NP und coNP.

Der Beweis von Satz 7.5 wird uns eine Weile aufhalten. Ein Teil ist leicht: PRIMES $\in$ coNP, denn ist $m$ keine Primzahl, können wir das demonstrieren, indem wir einen Teiler von $m$ angeben. Der schwierigere Teil ist, einen kurzen Beweis für die Primalität einer Zahl zu finden. Zuerst einige zahlentheoretische Grundlagen, die Sie vielleicht teilweise schon kennen.

Sind $a$ und $m$ ganze Zahlen, nicht beide Null, schreiben wir $(a,m)$ für den größten gemeinsamen Teiler von $a$ und $m$ (Bsp.: $(12,20) = 4$, $(3,0) = 3$). Ist $(a,m) = 1$, heißen $a$ und $m$ *teilerfremd*. $a$ und $m$ sind genau dann teilerfremd, wenn es ganze Zahlen $s$ und $t$ mit $sa + tm = 1$ gibt. Denn ist $(a,m) = 1$, können $s$ und $t$ mit dem Euklidischen Algorithmus bestimmt werden, während umgekehrt $(a,m)$ die Größe $sa + tm$ teilt, so daß $(a,m) = 1$ sein muß, falls $sa + tm = 1$. Für jede ganze Zahl $k$ gilt $(a,m) = 1 \Rightarrow (a + km, m) = 1$, denn $sa + tm = 1 \Rightarrow s(a + km) + (t - sk)m = 1$. Wir definieren jetzt für alle $m \in \mathbb{N}$ $\mathbb{Z}_m^*$ als die Teilmenge von $\mathbb{Z}_m$ der Zahlen $a$ (eigentlich der Restklassen $a + m\mathbb{Z}$) mit $(a,m) = 1$. Nach obiger Beobachtung ist $\mathbb{Z}_m^*$ wohldefiniert: Es spielt keine Rolle, welcher Repräsentant $a$ gewählt wird.

**Lemma** 7.6: Für alle $m \in \mathbb{N}$ ist $\mathbb{Z}_m^*$ eine multiplikative Gruppe.

**Beweis:** Zunächst einmal ist $\mathbb{Z}_m^*$ unter Multiplikation abgeschlossen, denn

$$\left.\begin{aligned} s_1 a + t_1 m &= 1 \\ s_2 b + t_2 m &= 1 \end{aligned}\right\} \quad \Rightarrow \quad (s_1 s_2)ab + (\cdots)m = 1.$$

Da offensichtlich $1 \in \mathbb{Z}_m^*$, müssen wir nur noch zeigen, daß jedes Element $a \in \mathbb{Z}_m^*$ ein Inverses in $\mathbb{Z}_m^*$ hat. Aber nach Definition von $\mathbb{Z}_m^*$ gibt es ganze Zahlen $s$ und $t$ mit $sa + tm = 1$. Daraus sieht man sofort, daß $s \bmod m$ das gesuchte inverse Element ist. ∎

**Korollar** 7.7: $\mathbb{Z}_p$ ist ein Körper für alle Primzahlen $p$.

Wir setzen $\phi(m) = |\mathbb{Z}_m^*|$ (die Eulersche $\phi$-Funktion).

| $m$ | = | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $\cdots$ |
|---|---|---|---|---|---|---|---|---|---|---|
| $\phi(m)$ | = | 1 | 1 | 2 | 2 | 4 | 2 | 6 | 4 | $\cdots$ |

**Lemma** 7.8 (Chinesischer Restsatz): Sind $a, b \in \mathbb{N}$ teilerfremd, so ist die Abbildung $f : x \mapsto (x \bmod a, x \bmod b)$ eine Bijektion (a) von $\mathbb{Z}_{ab}$ nach $\mathbb{Z}_a \times \mathbb{Z}_b$ und (b) von $\mathbb{Z}_{ab}^*$ nach $\mathbb{Z}_a^* \times \mathbb{Z}_b^*$.

**Beweis:** (a) $f$ ist injektiv, denn aus $x \bmod a = y \bmod a$ und $x \bmod b = y \bmod b$ folgt, daß sowohl $a$ als auch $b$ Teiler von $x - y$ sind, und daher, daß $ab$ ein Teiler von $x - y$ ist, so daß $x$ und $y$ modulo $ab$ identisch sind. Da die Mengen $\mathbb{Z}_{ab}$ und $\mathbb{Z}_a \times \mathbb{Z}_b$ die gleiche Kardinalität haben, folgt die Bijektivität von $f$ in Teil (a).

(b) Sind $x$ und $ab$ teilerfremd, gilt das auch für $x$ und $a$ und dann ebenfalls für $x \bmod a$ und $a$. Also wird $\mathbb{Z}_{ab}^*$ tatsächlich nach $\mathbb{Z}_a^* \times \mathbb{Z}_b^*$ abgebildet. Umgekehrt sieht man leicht, daß ein Argument, das nach $\mathbb{Z}_a^* \times \mathbb{Z}_b^*$ abgebildet wird, notwendigerweise aus $\mathbb{Z}_{ab}^*$ stammen muß. Also gilt die Bijektivität von $f$ auch in Teil (b). ∎

**Korollar** 7.9: Sind $a, b \in \mathbb{N}$ teilerfremd, ist $\phi(ab) = \phi(a)\phi(b)$.

**Beweis:** Das folgt sofort aus Lemma 7.8, Teil (b). ∎

**Lemma** 7.10: Sei $m \in \mathbb{N}$ und sei $P$ die Menge der Primzahlen, die $m$ teilen. Dann ist

$$\phi(m) = m \prod_{p \in P} (1 - \tfrac{1}{p}).$$

**Beweis:** Nach Korollar 7.9 reicht es, die Behauptung für solche $m$ zu zeigen, die von der Form $m = p^i$ sind, wobei $p$ eine Primzahl und $i$ eine natürliche Zahl ist. Aber das ist leicht, denn eine Zahl ist genau dann nicht teilerfremd zu $p^i$, wenn sie durch $p$ teilbar ist, und das betrifft genau $1/p$ der Zahlen im Bereich $\{0, \ldots, p^i - 1\}$. ∎

**Lemma** 7.11: Sei $m \in \mathbb{N}$ und sei $D$ die Menge der positiven Teiler von $m$. Dann ist

$$\sum_{d \in D} \phi(d) = m.$$

**Beweis:** Sei $\prod_{i=1}^{l} p_i^{k_i}$ die Primfaktorzerlegung von $m$. Wir betrachten das Produkt

$$\prod_{i=1}^{l} (\phi(1) + \phi(p_i) + \phi(p_i^2) + \cdots + \phi(p_i^{k_i})).$$

Nach Lemma 7.10 ist der $i$te Faktor in diesem Produkt $1 + (p_i - 1) + (p_i^2 - p_i) + \cdots + (p_i^{k_i} - p_i^{k_i - 1}) = p_i^{k_i}$, so daß das ganze Produkt nichts anderes ist als $m$.

Multiplizieren wir das Produkt aus, bekommen wir genau einen Term für jedes Element aus $D$. Der Term, der $d = \prod_{i=1}^{l} p_i^{k_i'}$ entspricht, wobei $0 \leq k_i' \leq k_i$ für $i = 1, \ldots, l$, ist genau $\prod_{i=1}^{l} \phi(p_i^{k_i'})$, welches, nach Korollar 7.9, $\phi(d)$ ist. Die Behauptung folgt. ∎

Ist $G$ eine endliche Gruppe mit neutralem Element 1, definieren wir für alle $a \in G$ die *Ordnung* von $a$ als $\mathrm{ord}(a) = \min\{i \in \mathbb{N} \mid a^i = 1\}$. Das ist wohldefiniert (d.h. $a^i$ wird irgendwann 1), denn in der Folge $a, a^2, a^3, \ldots$ muß es Wiederholungen geben, und ist $a^i = a^j$ mit $i < j$, ist $a^{j-i} = 1$. Es ist leicht zu sehen, daß $a^i = 1$, für $i \in \mathbb{N}$, genau dann, wenn $i$ ein Vielfaches von $\mathrm{ord}(a)$ ist.

Eine Teilmenge $H \subseteq G$, die unter der Gruppenoperation abgeschlossen ist ($h_1, h_2 \in H \Rightarrow h_1 \cdot h_2 \in H$), ist notwendigerweise eine Untergruppe von $G$. Denn sei $a \in H$. Dann ist $a \cdot a^{\mathrm{ord}(a)-1} = 1$, so daß $a^{-1} \in H$ und $1 \in H$.

**Lemma** 7.12 (Lagrange): Ist $H$ eine Untergruppe von $G$, so ist $|H|$ ein Teiler von $|G|$.

**Beweis** (Skizze): Für $g \in G$ sei $gH = \{gh \mid h \in H\}$. Das Mengensystem $\{gH \mid g \in G\}$ bildet eine Partition von $G$, in der jede Klasse $|H|$ Elemente hat. ∎

Nach Lemma 7.12 ist $\mathrm{ord}(a)$ ein Teiler von $|G|$ für alle $a \in G$, denn $\{a, a^2, a^3, \ldots\}$ ist eine Untergruppe von $G$ mit $\mathrm{ord}(a)$ Elementen. Im folgenden interessieren wir uns

für die Ordnungen der Elemente in $\mathbb{Z}_p^*$, wobei $p$ eine Primzahl ist. Für $d \in \mathbb{N}$ sei $R(d)$ die Anzahl der Elemente in $\mathbb{Z}_p^*$ der Ordnung $d$. Wir wissen, daß $R(d) = 0$, wenn $d$ kein Teiler von $|\mathbb{Z}_p^*| = p - 1$ ist. Insbesondere ist $p - 1$ die größtmögliche Ordnung. Wir wollen nachweisen, daß es tatsächlich Elemente der Ordnung $p - 1$ gibt; ein solches heißt ein *erzeugendes Element*, weil alle anderen Elemente daraus (durch Potenzieren) "erzeugt" werden können. Zuerst überzeugen wir uns, daß Polynome vom Grad $d$ über einem beliebigen Körper höchstens $d$ Nullstellen haben.

**Lemma** 7.13: Sei $\mathbb{F}$ ein Körper und sei $g$ ein Polynom über $\mathbb{F}$ vom Grad $d$, das nicht das Nullpolynom ist. Dann hat die Gleichung $g(x) = 0$ höchstens $d$ verschiedene Lösungen.

**Beweis:** Durch Induktion über $d$. Ein Polynom vom Grad 0 ist eine Konstante; ist die Konstante nicht 0, gibt es keine Nullstellen. Sei jetzt $g(x) = a_d x^d + \cdots + a_1 x + a_0$ ein Polynom vom Grad $d \geq 1$, von dem wir annehmen, daß es $d + 1$ Nullstellen $x_1, \ldots, x_{d+1}$ hat. Dann ist $h(x) = g(x) - a_d \prod_{i=1}^{d}(x - x_i)$ ein Polynom vom Grad höchstens $d - 1$ (die Terme vom Grad $d$ heben sich gegenseitig auf), $h$ ist nicht das Nullpolynom, denn $h(x_{d+1}) \neq 0$, aber $h$ hat die $d$ Nullstellen $x_1, \ldots, x_d$, ein Widerspruch. ∎

Nehmen wir an, daß $\mathbb{Z}_p^*$ mindestens ein Element der Ordnung $d$ enthält, wobei $d \in \mathbb{N}$, und sei $a$ ein solches Element. Dann sind $1, a, a^2, \ldots, a^{d-1}$ alle verschieden, und alle sind sie Lösungen der Gleichung $x^d \equiv 1 \pmod{p}$, denn $(a^i)^d = (a^d)^i \equiv 1 \pmod{p}$ für $i = 0, \ldots, d - 1$. Nach Lemma 7.13 hat die Gleichung $x^d \equiv 1 \pmod{p}$ keine weiteren Lösungen. Also sind die Elemente $1, a, a^2, \ldots, a^{d-1}$ die einzigen, die Ordnung $d$ haben können. Sie haben aber nicht alle Ordnung $d$, denn ist $0 \leq i \leq d - 1$ und $(d, i) = k \geq 2$, dann ist $(a^i)^{d/k} = (a^{i/k})^d \equiv 1 \pmod{p}$, so daß die Ordnung von $a^i$ höchstens $d/k$ sein kann. Hat also $a^i$ die Ordnung $d$, müssen $i$ und $d$ teilerfremd sein. Es folgt, daß $R(d) \leq \phi(d)$ für alle $d \in \mathbb{N}$. Sei $D$ die Menge der positiven Teiler von $p - 1$. Da $R(d) = 0$ für $d \notin D$, während jedes Element in $\{1, \ldots, p - 1\}$ in genau einem der $R(d)$ "gezählt" wird, ist

$$p - 1 = \sum_{d \in D} R(d) \leq \sum_{d \in D} \phi(d) = p - 1,$$

wobei in der letzten Umformung Lemma 7.11 benutzt wurde. Aber daraus folgt, daß $R(d) = \phi(d)$ für alle $d \in D$. Insbesondere ist $R(p - 1) = \phi(p - 1) \geq 1$ (z.B. ist immer $p - 2 \in \mathbb{Z}_{p-1}^*$). Mit anderen Worten hat jede Gruppe der Form $\mathbb{Z}_p^*$, wobei $p$ eine Primzahl ist, in der Tat mindestens ein erzeugendes Element.

Ist $m \geq 3$ eine Primzahl und $a$ ein erzeugendes Element von $\mathbb{Z}_m^*$, gelten natürlich die folgenden Bedingungen:

(1) $a^{m-1} \equiv 1 \pmod{m}$;

(2) $a^{(m-1)/q} \not\equiv 1 \pmod{m}$ für alle Primfaktoren $q$ von $m-1$.

Sei jetzt $m \geq 3$ keine Primzahl und sei $a \in \{1, \ldots, m-1\}$ beliebig. Wir zeigen, daß die Bedingungen (1) und (2) dann nicht beide gelten können. Gilt Bedingung (1), muß $a \in \mathbb{Z}_m^*$ sein, und ord($a$) ist ein Teiler von $m-1$. Da $|\mathbb{Z}_m^*| < m-1$, kann die Ordnung von $a$ nicht $m-1$ sein, sondern sie muß kleiner sein. Aber damit teilt sie auch $(m-1)/q$ für mindestens einen Primfaktor $q$ von $m-1$ (in Primfaktoren auflösen!), woraus folgt, daß $a^{(m-1)/q} \equiv 1 \pmod{m}$, ein Widerspruch zu Bedingung (2).

Damit sind wir nahe dran, unseren kurzen Beweis für die Primalität von $m$ zu haben: Möchte ich Sie davon überzeugen, daß $m \geq 3$ eine Primzahl ist, gebe ich Ihnen ein erzeugendes Element von $\mathbb{Z}_m^*$ (wir wissen, daß ein solches existiert), und Sie verifizieren die Bedingungen (1) und (2), die ja erfüllt sind, wenn $m$ in der Tat eine Primzahl ist, aber verletzt sein müssen, falls ich Sie belüge.

Die Sache hat einen Haken, nämlich daß es schwer für Sie sein könnte, die Primfaktoren von $m-1$ zu finden, die Sie ja brauchen, um Bedingung (2) zu überprüfen. Da muß ich Ihnen helfen, indem ich Ihnen als Teil meines Beweises auch die Primfaktorzerlegung von $m-1$ verrate. Hier könnte ich Sie aber wieder belügen, so daß Sie darauf bestehen müssen, daß ich Sie rekursiv von der Primalität der auftretenden angeblichen Primfaktoren überzeuge—daß diese das richtige Produkt haben, nämlich $m-1$, können Sie leicht selbst überprüfen. Ein Beweis für die Primalität von 5987 sieht also zum Beispiel aus, wie in Fig. 7.2 dargestellt.

Wir müssen uns jetzt nur noch davon überzeugen, daß der Beweis polynomielle Größe hat und in Polynomialzeit verifiziert werden kann, so daß eine NP-Maschine den Beweis raten und überprüfen kann. "Polynomiell" heißt hier polynomiell in der Anzahl $n$ der Bits der zu prüfenden Eingabezahl.

Unser Beweis ist baumstrukturiert. Definieren wir die *Größe* eines im Beweis vorkommenden Knotens als den binären Logarithmus aus der Zahl, um die es in diesem Knoten geht (das ist die Zahl in der ersten Zeile der Kästchen in Fig. 7.2). Die Größe der Wurzel ist (praktisch) unsere Eingabegröße $n$. Die Tiefe des Baums ist $O(n)$, denn die Größe jedes Knotens außer der Wurzel ist um mindestens 1 kleiner als die Größe seines Vaters. Außerdem übersteigt die Gesamtgröße aller Kinder eines Knotens $v$ nicht die Größe von $v$. Aber daraus folgt durch Induktion, daß die Gesamtgröße aller Knoten derselben Tiefe im Baum $O(n)$ ist, und daher, daß die Gesamtgröße aller Knoten $O(n^2)$ ist. Um zu sehen, daß die Größe des Beweises polynomiell ist, fehlt jetzt nur noch die Beobachtung, daß die in

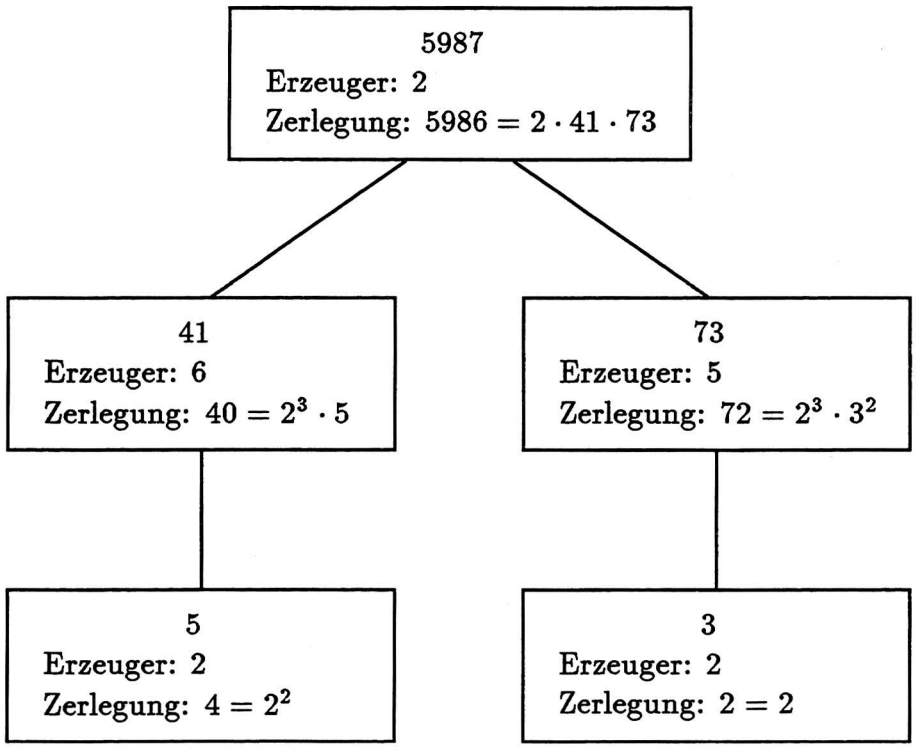



Fig. 7.2. Ein Beweis für die Primalität von 5987.

einem Knoten der Größe $s$ enthaltene Information in Platz $O(s)$ niedergeschrieben werden kann.

Damit hat der Beweis also tatsächlich polynomielle Größe. Bei seiner Überprüfung treten zwei nicht-triviale Operationen auf, nämlich Multiplikation und modulare Potenzierung. Multiplikation kann sicher in Polynomialzeit ausgeführt werden, und das gleiche gilt für modulare Potenzierung, vorausgesetzt, man potenziert durch wiederholte Quadrierung (um z.B. $a^{23}$ (modulo $m$) zu bestimmen, generiert man zuerst $a$, $a^2$, $a^4$, $a^8$ und $a^{16}$ und erhält dann $a^{23}$ als $a^{16} \cdot a^4 \cdot a^2 \cdot a$). Der Beweis kann also in Polynomialzeit überprüft werden, und wir haben gezeigt, daß PRIMES $\in$ NP. Damit ist auch die zweite Hälfte von Satz 7.5 bewiesen. ∎

# 8 RP und ein randomisierter Primalitätstest

Ist das Polynom

$$g(x,y) = (x-1)^4(y+2) + (2x^3+6)(y^2+1) + (x^2-x+1)^2(y-2)^3 - (x^4+14)y^3$$

identisch Null, also nur eine komplizierte Schreibweise für das Nullpolynom? Wir könnten das Polynom ausmultiplizieren und versuchen, es zu vereinfachen. Wäre $g$ statt dessen die Determinante einer $m \times m$ Matrix, deren Einträge Polynome in $m$ Unbekannten $x_1, \ldots, x_m$ sind—$g$ wäre dann auch selbst ein Polynom in $x_1, \ldots, x_m$—wäre das aber praktisch nicht mehr machbar, da die Größe der expandierten Determinante im allgemeinen exponentiell in $m$ ist. Wir werden jetzt ein Verfahren kennenlernen, das in beiden Fällen gleichermaßen gut funktioniert. Das Verfahren heißt ... Ausprobieren!

Wir könnten versuchen, einfach ein Paar $(x,y)$ mit $g(x,y) \neq 0$ zu raten, womit die Sache klar wäre. Es gibt tatsächlich solche Paare—eins zu finden, überlasse ich Ihnen—woraus wir schließen können, daß $g$ nicht das Nullpolynom ist. Was ist aber, wenn wir zehn Paare $(x,y)$ ausprobiert hätten, ohne jemals einen Wert von $g$ ungleich Null anzutreffen? Könnten wir dann umgekehrt schließen, daß $g$ wohl tatsächlich das Nullpolynom ist? Unsere Intuition erlaubt vielleicht diesen Schluß, aber zumindest mathematisch steht er auf schwachen Füßen, zumal es Polynome $g(x,y)$ gibt, die nicht das Nullpolynom sind, aber für unendlich viele Paare $(x,y)$ den Wert Null annehmen—denken Sie nur an $g(x,y) = x - y$. Wir wollen jetzt ein Verfahren dieser Art mathematisch rechtfertigen.

Ein *Polynom* über $x_1, \ldots, x_m$ ist eine Summe der Form

$$g(x_1, \ldots, x_m) = \sum_{\alpha} c_\alpha x_1^{\alpha_1} \cdots x_m^{\alpha_m},$$

wobei $\alpha = (\alpha_1, \ldots, \alpha_m)$ über eine endliche Menge von Vektoren der Länge $m$ läuft, deren Komponenten nichtnegative ganze Zahlen sind. Der maximale Wert von $\alpha_1 + \cdots + \alpha_m$, für den der Koeffizient $c_\alpha$ ungleich Null ist, heißt der *totale Grad* von $g$.

**Satz** 8.1 (Schwartz/Zippel): Sei $\mathbb{F}$ ein Körper, sei $S$ eine endliche Teilmenge von $\mathbb{F}$ und sei $g(x_1,\ldots,x_m)$ ein vom Nullpolynom verschiedenes Polynom über $\mathbb{F}$ vom totalen Grad $d$. Dann hat die Gleichung $g(x_1,\ldots,x_m)=0$ höchstens $d\cdot|S|^{m-1}$ Lösungen in $S^m$.

**Beweis:** Durch Induktion über $m$. Für $m=1$ ist die Aussage, daß ein vom Nullpolynom verschiedenes Polynom in einer Variablen vom Grad $d$ höchstens $d$ Nullstellen hat; dies haben wir schon in Lemma 7.13 gezeigt. Sei jetzt $m\geq 2$. Wir schreiben $g$ als ein Polynom $g'$ in der einen Variablen $x_m$ mit Koeffizienten, die Polynome in den restlichen Variablen $x_1,\ldots,x_{m-1}$ sind. Die Koeffizienten von $g'$ sind nicht alle das Nullpolynom (denn dann wäre es $g$ auch). Sei $d'$ die höchste vorkommende Potenz von $x_m$, deren Koeffizient $h(x_1,\ldots,x_{m-1})$ in $g'$ vom Nullpolynom verschieden ist; der totale Grad von $h$ ist höchstens $d-d'$. Wir teilen die Nullstellen von $g$ in zwei Gruppen und zählen die Nullstellen in jeder Gruppe separat.

*Gruppe* 1: Nullstellen $(a_1,\ldots,a_m)$ mit $h(a_1,\ldots,a_{m-1})=0$. Nach Induktion hat $h$ höchstens $(d-d')|S|^{m-2}$ Nullstellen. Schlimmstenfalls kann jede davon mit einem beliebigen Wert $a_m$ von $x_m$ kombiniert werden. Also enthält Gruppe 1 höchstens $(d-d')|S|^{m-2}\cdot|S|=(d-d')|S|^{m-1}$ Nullstellen von $g$.

*Gruppe* 2: Nullstellen $(a_1,\ldots,a_m)$ mit $h(a_1,\ldots,a_{m-1})\neq 0$. Für jede feste Wahl von $a_1,\ldots,a_{m-1}$ ist $g'$ ein Polynom in $x_m$ mit Koeffizienten in $\mathbb{F}$. Gibt es Nullstellen der Form $(a_1,\ldots,a_{m-1},x)$ in Gruppe 2, so ist $g'$ nicht das Nullpolynom, und jedes solche $x$ ist eine Nullstelle von $g'$. Also gibt es höchstens $d'$ Nullstellen in Gruppe 2 für jedes Tupel $(a_1,\ldots,a_{m-1})$, insgesamt höchstens $d'|S|^{m-1}$ Nullstellen.

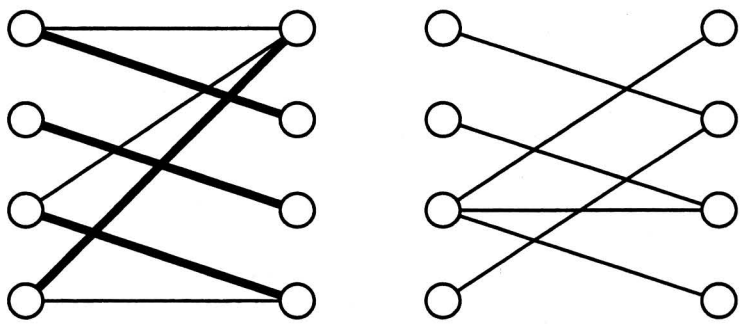
Addieren wir die oberen Schranken für die Anzahl der Nullstellen in den Gruppen 1 und 2, erhalten wir gerade $d|S|^{m-1}$. ∎

Falls wir testen wollen, ob $g(x_1,\ldots,x_m)$ das Nullpolynom ist, können wir eine passende Menge $S$ festlegen und $g$ an einem zufällig gewählten Tupel aus $S^m$ auswerten. Wenn $g$ das Nullpolynom ist, werden wir natürlich immer den Wert 0 erhalten. Nehmen wir also an, daß $g$ nicht das Nullpolynom ist. Es gibt $|S|^m$ zur Auswahl stehende Tupel, aber Satz 8.1 besagt, daß höchstens $d|S|^{m-1}$ von diesen Tupeln Nullstellen von $g$ sind, wobei $d$ der totale Grad von $g$ ist. Die Wahrscheinlichkeit, eine Nullstelle zu erwischen, ist also höchstens $d/|S|$. Der totale Grad unseres Beispielpolynoms $g(x,y)$ ist 7. Werten wir $g$ an einem zufälligen Paar $(x,y)\in\{0,\ldots,13\}^2$ aus, erkennen wir also mit Wahrscheinlichkeit mindestens $1/2$ sofort, daß $g$ nicht das Nullpolynom ist. Wollen wir uns mit einer Fehlerwahrscheinlichkeit von $1/2$ nicht zufriedengeben, brauchen wir den Test nur zu wiederholen. Die Wahrscheinlichkeit, daß wir in 10 aufeinanderfolgenden unabhängi-

gen Versuchen immer nur Nullstellen von $g$ treffen, ist höchstens $2^{-10} \approx 1/1000$, was im praktischen Leben oft mit Null gleichgesetzt werden kann.

Das obige Verfahren ist unser erstes Beispiel für einen randomisierten Algorithmus: Wir wählen die Argumente, auf denen das Polynom ausgewertet werden soll, zufällig aus, denn wir kennen kein deterministisches Verfahren, um Argumente zu finden, die keine Nullstellen sind, ohne fast alle Möglichkeiten auszuprobieren. Wir werden bald den Begriff eines randomisierten Algorithmus formalisieren, aber zuerst ein Beispiel, das zeigt, daß es in der Tat manchmal nützlich sein kann, testen zu können, ob ein vorliegendes Polynom das Nullpolynom ist.

**Beispiel** 8.2 (Perfektes Matching): Sei $G = (V \cup W, E)$ ein bipartiter Graph auf den Knotenmengen $V = \{v_1, \ldots, v_m\}$ und $W = \{w_1, \ldots, w_m\}$ (d.h. jede Kante in $E$ verbindet einen Knoten in $V$ und einen Knoten in $W$). Ein *perfektes Matching* in $G$ paart jeden Knoten $v_i$ in $V$ mit einem Knoten $w_{\sigma(i)}$ in $W$, wobei $(v_i, w_{\sigma(i)}) \in E$ und $\sigma$ eine Permutation von $\{1, \ldots, m\}$ ist (Fig. 8.1).



Bipartiter Graph mit eingezeichnetem perfektem Matching

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

Bipartiter Graph ohne perfektes Matching

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

Fig. 8.1. Graphen mit und ohne perfektes Matching.

Wir wollen überprüfen, ob $G$ ein perfektes Matching besitzt. Sei dazu $A = (a_{i,j})$ die $m \times m$ Adjazenzmatrix von $G$ mit

$$a_{i,j} = \begin{cases} 1, & \text{falls } (v_i, w_j) \in E; \\ 0, & \text{sonst.} \end{cases}$$

Betrachten wir die Determinante von $A$,

$$\det A = \sum_{\sigma} \operatorname{sgn}(\sigma) a_{1,\sigma(1)} \cdots a_{m,\sigma(m)},$$

wobei $\sigma$ über alle Permutationen von $\{1,\ldots,m\}$ läuft und $\operatorname{sgn}(\sigma)$ das Vorzeichen von $\sigma$ angibt ($\operatorname{sgn}(\sigma) = 1$, falls $\sigma$ durch eine gerade Anzahl von Transpositionen aus der Identitätspermutation erhalten werden kann, $\operatorname{sgn}(\sigma) = -1$ sonst). Man sieht, daß jeder Term in $\det A$, der von Null verschieden ist, einem perfekten Matching in $G$ entspricht, und auch umgekehrt. Hat $G$ kein perfektes Matching, ist daher sicherlich $\det A = 0$. Die Umkehrung gilt aber nicht: Ist z.B. $a_{i,j} = 1$ für alle $i$ und $j$, ist $\det A = 0$, aber der entsprechende Graph $G$ hat ein perfektes Matching (er hat sogar $m!$ perfekte Matchings). Der Grund ist, daß Terme in $\det A$ sich gegenseitig aufheben können. Ersetze jetzt aber jeden Eintrag $a_{i,j}$ in $A$ durch $a_{i,j} \cdot x_{i,j}$, wobei $x_{i,j}$ eine Variable (Unbekannte) ist, und nenne die so entstehende Matrix $A'$. $\det A'$ ist keine reelle Zahl mehr, sondern ein Polynom in den $m^2$ Variablen $x_{1,1},\ldots,x_{m,m}$, und es ist leicht zu sehen, daß $\det A'$ genau dann das Nullpolynom ist, wenn $G$ kein perfektes Matching besitzt—jetzt können sich Terme nicht mehr gegenseitig aufheben. Um zu entscheiden, ob $G$ ein perfektes Matching hat, müssen wir also nur noch entscheiden, ob ein vorliegendes Polynom das Nullpolynom ist, und das haben wir gerade gelernt. Da der totale Grad des Polynoms durch $m$ beschränkt ist, bekommen wir eine Fehlerwahrscheinlichkeit von höchstens $1/2$, wenn wir $|S| = 2m$ wählen. Determinanten können in Polynomialzeit berechnet werden (das wollen wir hier nicht zeigen). Wiederholen wir den Test z.B. $m$-mal, haben wir also einen Algorithmus, der in Polynomialzeit läuft. Hat $G$ kein perfektes Matching, ist die Antwort des Algorithmus immer korrekt; sonst ist die Fehlerwahrscheinlichkeit höchstens $2^{-m}$.

Man könnte meinen, daß wir, um randomisierte Algorithmen formal zu beschreiben, die TM mit der Fähigkeit ausstatten müßten, zufällige Zahlen zu generieren. Statt dessen werden wir probabilistische TMs als gewöhnliche NTMs auffassen, die allerdings ein neues Akzeptkriterium benutzen. Es ist praktisch, die NTMs vorher zu standardisieren.

**Definition:** Wir nennen eine NTM $M$ *normal*, wenn es eine Funktion $f : \mathbb{N}_0 \to \mathbb{N}$ gibt, so daß jede Berechnung von $M$ auf einer Eingabe der Länge $n$ genau $f(n)$ Schritte lang ist, und wenn jede Konfiguration von $M$ genau zwei Nachfolgekonfigurationen hat.

Das folgende Lemma zeigt, daß wir nichts Wesentliches verlieren, wenn wir uns auf normale NTMs zurückziehen.

**Lemma** 8.3: Jede Sprache in NP wird von einer normalen NTM in Polynomialzeit akzeptiert.

**Beweis:** Sei $L$ eine Sprache in NP und $M$ eine NTM, die das bezeugt. Zuerst modifizieren wir die Übergangsfunktion von $M$ so, daß keine Konfiguration mehr als zwei Nachfolgekonfigurationen hat, indem wir jede nichtdeterministische Wahl zwischen $k \geq 3$ Möglichkeiten ($k$ ist eine Konstante) durch einen "Fanout"-Baum der Tiefe $\lceil \log k \rceil$ ersetzen, in dem an jedem Knoten zwischen höchstens zwei Möglichkeiten gewählt wird. Eine Laufzeit, die nur von der Eingabelänge abhängt, erzwingen wir mit Hilfe eines Zählers, der polynomiell lange die Schritte mitzählt und die Berechnung dann beendet, wobei das Polynom natürlich so gewählt wird, daß die eigentliche Berechnung vorher zu Ende kommt. Schließlich "verdoppeln" wir jeden Zustand der Maschine durch Einführung eines neuen Zustands, der exakt wie sein Partner behandelt wird, wodurch deterministische Übergänge trivialerweise zu nichtdeterministischen Übergängen mit zwei Wahlmöglichkeiten gemacht werden können. Die Maschine ist jetzt normal und akzeptiert immer noch $L$ in Polynomialzeit. ∎

Eine normale NTM $M$ mit Laufzeit $f : \mathbb{N}_0 \to \mathbb{N}$ hat auf einer Eingabe $x$ mit $|x| = n$ genau $2^{f(n)}$ verschiedene Berechnungen, und wir können uns den von $Init_M(x)$ aus erreichbaren Teilgraphen des Konfigurationsgraphen als einen vollständigen binären Baum $T_M(x)$ der Tiefe $f(n)$ vorstellen (selbst wenn einige Knoten im Baum eigentlich identische Konfigurationen repräsentieren). Wir stellen uns $M$ folgendermaßen als probabilistische Maschine vor: In jedem Schritt ihrer Berechnung befindet sich $M$ an einem inneren Knoten $v$ von $T_M(x)$ und entscheidet zufällig, zu welchem Kind von $v$ sie geht. Jedes Kind wird mit Wahrscheinlichkeit $1/2$ gewählt, und die zufälligen Entscheidungen an verschiedenen Knoten in $T_M(x)$ sind (stochastisch) unabhängig. Noch bildlicher können wir uns vorstellen, daß die Maschine in jedem Schritt eine Münze wirft und dem Ergebnis entsprechend ihre Berechnung fortsetzt. In einem eigentlich deterministischen Schritt wird die Münze auch geworfen, weil es das Modell verlangt, aber das Ergebnis wird ignoriert. Es sollte klar sein, daß alle Blätter von $T_M(x)$ mit der gleichen Wahrscheinlichkeit erreicht werden, nämlich mit der Wahrscheinlichkeit $2^{-f(n)}$.

**Definition:** Eine normale NTM $M$ *R-akzeptiert* ("R" steht für "Random") eine Sprache $L$, wenn folgende Bedingungen erfüllt sind:

(1) Für $x \notin L$ gibt es keine akzeptierenden Blätter in $T_M(x)$;

(2) Für $x \in L$ akzeptiert mindestens die Hälfte der Blätter in $T_M(x)$.

In der probabilistischen Denkweise werden Eingaben, die nicht zu $L$ gehören, also mit Wahrscheinlichkeit 1 abgelehnt, während Eingaben in $L$ mit Wahrscheinlichkeit mindestens $1/2$ akzeptiert werden. Das war genau die Situation in unserem Beispiel, wenn $L$ als die Sprache PERFECT BIPARTITE MATCHING genommen wird.

PERFECT BIPARTITE MATCHING:

Eingabe: Ein bipartiter Graph $G = (V \cup W, E)$ mit $|V| = |W|$.

Frage: Besitzt $G$ ein perfektes Matching?

**Bemerkung:** Jede normale NTM akzeptiert eine Sprache, aber nicht jede normale NTM R-akzeptiert eine Sprache. Es könnte z.B. Eingaben geben, die mit Wahrscheinlichkeit $1/4$ akzeptiert werden, und das ist nicht zulässig.

Jetzt können wir RTIME($f$) und RSPACE($f$) in der offensichtlichen Weise definieren. Wir brauchen vor allem

$$\text{RP} = \{L \mid L \text{ wird von einer normalen NTM R-akzeptiert, die in Polynomialzeit läuft}\}.$$

Im Beispiel haben wir gesehen, daß PERFECT BIPARTITE MATCHING $\in$ RP. Was können wir sonst noch über RP aussagen?

**Satz** 8.4: $\text{P} \subseteq \text{RP} \subseteq \text{NP}$.

**Beweis:** Sei $L \in \text{P}$ und sei $M$ eine TM, die das bezeugt. Wir können $M$ in eine normale NTM umfunktionieren, die alle Münzwürfe ignoriert. Bei Eingaben in $L$ akzeptieren alle Blätter, bei Eingaben in $\overline{L}$ lehnen alle Blätter ab; insbesondere sind die Bedingungen für R-Akzeptanz erfüllt.

Sei nun $L \in \text{RP}$ und sei $M$ eine normale NTM, die das bezeugt. Für Eingaben in $L$ akzeptiert mindestens die Hälfte der Blätter, insbesondere gibt es mindestens eine akzeptierende Berechnung. Für Eingaben in $\overline{L}$ gibt es keine akzeptierende Berechnung. Also akzeptiert $M$ die Sprache $L$ auch im üblichen Sinne. ∎

Die Anwendung von Randomisierung erlaubt uns möglicherweise, Probleme zu lösen, die nicht in P liegen. Das Problem PERFECT BIPARTITE MATCHING ist hierfür kein gutes Beispiel, denn es gibt auch deterministische Algorithmen für PERFECT BIPARTITE MATCHING, die in Polynomialzeit arbeiten (allerdings ist unser randomisierter Algorithmus fast trivial zu programmieren, wenn eine Subroutine für die Auswertung von Determinanten schon zur Verfügung steht). Wir können natürlich nicht hoffen, für ein Problem in RP zu zeigen, daß es nicht in P liegt, denn damit hätten wir auch P und NP getrennt, aber wir werden jetzt einen schnellen randomisierten Algorithmus für ein Problem kennenlernen, für das zumindest nicht bekannt ist, ob es in P liegt. Es handelt sich um unseren alten Freund, das Primalitätsproblem.

Man kann zeigen, daß PRIMES $\in$ RP (L. M. Adleman and M. A. Huang, Recognizing primes in random polynomial time, 19th Annual ACM Symp. on Theory of Computing (1987), pp. 462–469). Das ist allerdings eine Ecke zu schwierig für uns, und wir begnügen uns damit, die Aussage PRIMES $\in$ coRP zu beweisen (erinnern Sie sich daran, daß PRIMES $\in$ coNP auch viel leichter als PRIMES $\in$ NP zu zeigen war). Gesucht ist also ein randomisiertes Verfahren, das eine zusammengesetzte Zahl mit Wahrscheinlichkeit mindestens $1/2$ als solche entlarvt, während bei Primzahlen kein Fehler erlaubt ist.

**Lemma** 8.5 (Kleiner Satz von Fermat): Ist $m$ eine Primzahl und $a$ eine ganze Zahl mit $1 \leq a < m$, dann ist $a^{m-1} \equiv 1 \pmod{m}$.

**Beweis:** Die Ordnung von $a$ in der Gruppe $\mathbb{Z}_m^*$ ist ein Teiler von $|\mathbb{Z}_m^*| = m - 1$, woraus die Behauptung sofort folgt. ∎

Lemma 8.5 gibt uns ein Kriterium für Primalität: Ist $a^{m-1} \not\equiv 1 \pmod{m}$ für ein $a$ mit $1 \leq a < m$, wissen wir sicher, daß $m$ keine Primzahl ist; $a$ ist ein *Zeuge* dafür, daß $m$ zusammengesetzt ist. Ist $a^{m-1} \equiv 1 \pmod{m}$, können wir aber nicht umgekehrt schließen, daß $m$ eine Primzahl ist. Wir könnten dann ein neues $a$ zufällig wählen und das Kriterium wieder anwenden, aber leider gibt es zusammengesetzte Zahlen (die sogenannten *Carmichael-Zahlen*), die fast keine Zeugen besitzen, so daß dieses Verfahren zum Scheitern verurteilt ist. Ein ähnliches Kriterium basiert auf der folgenden Beobachtung:

**Lemma** 8.6: Sei $m$ eine Primzahl und sei $x$ eine ganze Zahl. Dann gilt

$$x \equiv 1 \text{ oder } x \equiv -1 \text{ oder } x^2 \not\equiv 1 \pmod{m}.$$

**Beweis:** Das Polynom $x^2 - 1$ hat im Körper $\mathbb{Z}_m$ für $m \geq 3$ genau zwei Nullstellen, nämlich 1 und $-1$ (Lemma 7.13). ∎

Wir können also ein $x$ wählen und schauen, ob es die Bedingung aus Lemma 8.6 verletzt; wenn ja, ist $x$ wieder ein Zeuge dafür, daß $m$ zusammengesetzt ist. Wieder gibt es aber Zahlen $m$ (z.B. 4 und 27), die gar keine Zeugen in diesem Sinne haben. Wenn wir beide Kriterien anwenden, stellt sich aber heraus, daß jede zusammengesetzte Zahl viele Zeugen hat.

Im folgenden schreiben wir die Aussage $x \equiv 1$ oder $x \equiv -1 \pmod{m}$ als $x \equiv \pm 1 \pmod{m}$ und nennen die beiden Kriterien aus den Lemmata 8.5 und 8.6 den "Fermattest" und den "Wurzeltest".

Nehmen wir an, $a$ ist kein Zeuge für den Fermattest, d.h.

$$a^{m-1} \equiv 1 \pmod{m}.$$

Wo könnten wir hoffen, einen Zeugen für den Wurzeltest zu finden? Natürlich bei $x = a^{(m-1)/2}$. Ist $x \not\equiv \pm 1 \pmod{m}$, haben wir unseren Zeugen gefunden. Sonst, falls $x \equiv 1 \pmod{m}$ und $(m-1)/2$ gerade ist, können wir es mit $x = a^{(m-1)/4}$ weiter versuchen. Und so weiter. Wir halten wenn

| | | | |
|---|---|---|---|
| | $x = a^{(m-1)/2^i} \not\equiv \pm 1$ | $\pmod{m}$ | (Erfolg: Zeuge gefunden) |
| oder | $x \equiv -1$ | $\pmod{m}$ | (Kein Erfolg) |
| oder | $(m-1)/2^i$ ist ungerade | | (Kein Erfolg). |

Etwas präziser können wir den Algorithmus so beschreiben:

**Algorithmus** 8.7 (Rabin):

Eingabe: Eine $n$-Bitzahl $m \geq 3$.

Ausgabe: "Probably prime" oder "Composite".

$m$ ist eine Primzahl: Die Ausgabe ist "Probably prime";

$m$ ist zusammengesetzt: Die Ausgabe ist mit W'keit $\geq 1/2$ "Composite".

$a :=$ eine zufällige Zahl zwischen 1 und $m-1$;

**if**

(*) $m$ ist gerade **or**

(**) $(a, m) \neq 1$ **or**

(***) $m = x^y$ für ganze Zahlen $x, y \geq 2$ ($*$ $m$ ist eine nichttriviale Potenz $*$) **or**

$a^{m-1} \not\equiv 1 \pmod{m}$ ($*$ der Fermattest $*$)

**then** return("Composite");

Sei $m - 1 = 2^k u$ mit $u$ ungerade und $k, u \geq 1$;

**if** $\exists h \in \{0, \ldots, k-1\}$ :

$a^{2^h u} \not\equiv \pm 1 \pmod{m}$ **and** $a^{2^{h+1} u} \equiv 1 \pmod{m}$

**then** return("Composite");

return("Probably prime"); ($*$ alle Tests bestanden $*$)

Die Zeilen (*), (**) und (***) können weggelassen werden (Aufgaben 10.3, 11.1 und 11.2).

Wir wollen jetzt zeigen, daß der Algorithmus von Rabin korrekt ist, also daß die Kommentare im Algorithmenkopf zutreffen. Dazu betrachten wir verschiedene Fälle.

*Fall 1:* $m$ ist eine Primzahl. In diesem Fall ist $m$ ungerade, $(a, m) = 1$, $m \neq x^y$ für alle $x, y \geq 2$, $a^{m-1} \equiv 1 \pmod{m}$ (Satz von Fermat), und $\mathbb{Z}_m$ enthält keine nichttrivialen Quadratwurzeln von 1. Also wird der Algorithmus "Probably prime" ausgeben.

*Fall 2:* $m$ ist gerade oder $m = x^y$ mit $x, y \geq 2$. In diesem Fall gibt der Algorithmus "Composite" aus, und das ist natürlich korrekt.

*Fall 3* (der interessante Fall): $m = rs$, wobei $(r, s) = 1$ und $r, s \geq 3$.

Sei $h$ die größte ganze Zahl, so daß die Ordnung von mindestens einem Element $b \in \mathbb{Z}_m^*$ mit $b^{m-1} \equiv 1 \pmod{m}$ ein Vielfaches von $2^{h+1}$ ist. $h \geq 0$, denn $-1$ hat Ordnung 2.

Wir wählen jetzt $k$ und $u$ wie im Algorithmus (dann ist $h < k$) und setzen

$$G = \{a \in \mathbb{Z}_m^* \mid a^{2^h u} \equiv \pm 1 \pmod{m}\}.$$

**Lemma** 8.8: $G$ ist eine Untergruppe von $\mathbb{Z}_m^*$.

**Beweis:** $a, b \in G \Rightarrow (ab)^{2^h u} = a^{2^h u} b^{2^h u} \equiv (\pm 1)(\pm 1) = \pm 1 \pmod{m} \Rightarrow ab \in G.$ ∎

Sind $a, b \in \mathbb{N}$ teilerfremd, schreiben wir im folgende $\mathrm{ord}_a(b)$ für die Ordnung von $b$ in $\mathbb{Z}_a^*$.

**Lemma** 8.9: $G \neq \mathbb{Z}_m^*$.

**Beweis:** Wir wählen $b \in \mathbb{Z}_m^*$, so daß $2^{h+1} \mid \mathrm{ord}_m(b)$; das ist nach Definition von $h$ möglich. Nach Aufgabe 10.1 ist $\mathrm{ord}_m(b)$ das kleinste gemeinsame Vielfache von $\mathrm{ord}_r(b)$ und $\mathrm{ord}_s(b)$. Daraus können wir schließen, daß $2^{h+1} \mid \mathrm{ord}_r(b)$, gegebenenfalls nach Vertauschen von $r$ und $s$.

Wir wählen nun $a$ entsprechend dem chinesischen Restsatz (Lemma 7.8) so, daß

$$a \equiv b \pmod{r}$$
$$a \equiv 1 \pmod{s}.$$

Dann ist $a \in \mathbb{Z}_m^*$ (Aufgabe 10.2). Wir zeigen, daß $a \notin G$, womit Lemma 8.9 bewiesen sein wird.

$$a^{2^h u} \equiv b^{2^h u} \not\equiv 1 \pmod{r},$$

denn aus $2^{h+1} \mid \mathrm{ord}_r(b)$ und $\mathrm{ord}_r(b) \mid 2^h u$ würde $2^{h+1} \mid 2^h u$ folgen, was offensichtlich falsch ist, da $u$ ungerade ist. Ferner ist

$$a^{2^h u} \equiv 1 \pmod{s}.$$

Aus dem chinesischen Restsatz können wir jetzt schließen, daß $a^{2^h u} \not\equiv \pm 1 \pmod{m}$, denn die Reste von $a^{2^h u}$ modulo $r$ und $s$ stimmen weder mit denen von 1 (nämlich 1 und 1), noch mit denen von $-1$ (nämlich $-1$ und $-1$) überein. Aber $a^{2^h u} \not\equiv \pm 1 \pmod{m}$ bedeutet gerade $a \notin G$. ∎

Sei jetzt $a$ die vom Algorithmus gewählte Zahl. Sind $a$ und $m$ nicht teilerfremd, gibt der Algorithmus natürlich "Composite" aus.

**Lemma** 8.10: Ist $a \in \mathbb{Z}_m^* \backslash G$, gibt der Algorithmus "Composite" aus.

**Beweis:** Da $0 \leq h < k$, müssen wir nur zeigen, daß $a^{2^{h+1}u} \equiv 1 \pmod{m}$. Sei $\mathrm{ord}_m(a) = 2^i v$, wobei $i \in \mathbb{N}_0$ und $v$ ungerade ist. Da o.B.d.A. $a^{2^k u} \equiv 1 \pmod{m}$ (sonst besteht $a$ den Fermattest nicht), gilt $2^i v \mid 2^k u$ und daher $v \mid u$. Ferner ist $i \leq h+1$ nach Definition von $h$. Also ist

$$a^{2^{h+1}u} = (a^{2^i v})^{2^{h+1-i}u/v} \equiv 1 \pmod{m}. \quad \blacksquare$$

Der Algorithmus irrt sich also nur, wenn $a \in G$. Aber nach dem Satz von Lagrange (Lemma 7.12) ist

$$|G| \leq \tfrac{1}{2}|\mathbb{Z}_m^*| \leq \tfrac{m-1}{2}.$$

Ist $m$ zusammengesetzt, sind also mindestens die Hälfte der Zahlen in $\{1, \ldots, m-1\}$ Zeugen im Sinne des Algorithmus von Rabin.

Wir haben ein kleines technisches Problem, nämlich daß unser Maschinenmodell (die probabilistische TM) es nicht erlaubt, $a$ aus der uniformen Verteilung auf $\{1, \ldots, m-1\}$ zu wählen, es sei denn, $m-1$ ist eine Zweierpotenz. Das liegt daran, daß wir nur einfache "Münzwürfe" zur Verfügung haben, mit denen wir kein Ereignis mit Wahrscheinlichkeit z.B. $1/6$ konstruieren können. Statt dessen müssen wir $a$ in $\lceil \log_2(m-1) \rceil$ aufeinanderfolgenden Münzwürfen uniform aus dem größeren Bereich $\{1, \ldots, M\}$ wählen, wobei $M$ die kleinste Zweierpotenz $\geq m-1$ ist. Generieren wir eine nicht zulässige Zahl $a$ (d.h. $a \in \{m, \ldots, M\}$), geben wir "Probably prime" aus, da wir bei Primzahleingabe keinen Fehler machen dürfen. Da $M \leq 2(m-1)$, ist die Wahrscheinlichkeit, mit der eine zusammengesetzte Zahl zur Ausgabe "Composite" führt, immer noch mindestens $1/4$. Führen wir 3 unabhängige Versuche nacheinander aus, schrumpft die Wahrscheinlichkeit, mit der eine zusammengesetzte Zahl nicht als solche erkannt wird, auf höchstens $(3/4)^3 < 1/2$, womit wir im Einklang mit der Definition von coRP sind.

Wenn die Eingabe $m$ eine $n$-Bitzahl ist, kann der Algorithmus von Rabin mit $O(n)$ arithmetischen Operationen auf $n$-Bitzahlen ausgeführt werden (Aufgabe 11.3), also in Zeit polynomiell in $n$. Damit haben wir gezeigt:

**Satz** 8.11: PRIMES $\in$ coRP.

Wie schon erwähnt, ist es unbekannt, ob PRIMES $\in$ P. Wir haben aber die kuriose Situation, daß ein deterministisches Verfahren bekannt ist, das in Polynomialzeit läuft

und das vielleicht die Menge der Primzahlen erkennt. "Vielleicht" heißt hier, falls die sogenannte *erweiterte Riemannsche Vermutung* korrekt ist (G. L. Miller, Riemann's hypothesis and tests for primality, *J. Comput. System Sci.* **13** (1976), pp. 300–317). Für einen recht unterhaltsamen Artikel, der diese und andere Fragen im Umfeld von Primzahlen erörtert, s. L. M. Adleman, Algebraic number theory - the complexity contribution, 35th Annual Symp. on Foundations of Computer Science (1994), pp. 88–113.

# 9 Weitere probabilistische Komplexitätsklassen

Die Definition von RP verlangt, daß Eingaben in der Sprache mit Wahrscheinlichkeit mindestens $1/2$ akzeptiert werden, während Eingaben, die nicht zur Sprache gehören, immer abgelehnt werden müssen. Es dürfte klar sein, daß die Konstante $1/2$ dabei bedeutungslos ist—jede andere Konstante $\epsilon$ mit $0 < \epsilon < 1$ würde dieselbe Klasse definieren. Durch polynomiell viele unabhängige Wiederholungen können wir die Fehlerwahrscheinlichkeit bei Eingabelänge $n$ sogar für jedes Polynom $p$ auf $2^{-p(n)}$ drücken. Auch wenn wir die Fehlerwahrscheinlichkeit sehr weit reduzieren können, bekommen wir nie Gewißheit darüber, daß die Ausgabe eines RP-Algorithmus korrekt ist. Liegt eine Sprache $L$ aber nicht nur in RP, sondern auch in coRP, ändert sich das, denn dann können wir den folgenden Algorithmus ausführen, wobei $M_L$ und $M_{\overline{L}}$ normale NTMs sind, die $L$ bzw. $\overline{L}$ in Polynomialzeit R-akzeptieren, und $x$ die Eingabe ist:

**repeat**
  $a := M_L(x)$; (* `accept` oder `reject` *)
  $b := M_{\overline{L}}(x)$;
**until** $a \neq b$;
return($a$);

Der Algorithmus entscheidet immer korrekt, ob $x \in L$. Denn für jede Eingabe, ob nun in $L$ oder in $\overline{L}$, darf eine der Maschinen $M_L$ und $M_{\overline{L}}$ keinen Fehler machen; ihre Ausgabe ist somit immer korrekt. Wenn $a \neq b$, müssen daher beide Ausgaben $a$ und $b$ korrekt sein. Insbesondere ist $a =$ `accept` genau dann, wenn $x \in L$.

Die Ausgabe des obigen Algorithmus ist immer korrekt, also keine Zufallsvariable. Das einzige, was noch vom Zufall abhängt, ist die Laufzeit des Algorithmus, die beliebig groß werden kann. Allerdings ist jeder Durchlauf der **repeat**-Schleife mit Wahrscheinlichkeit mindestens $1/2$ der letzte, so daß es extrem unwahrscheinlich ist, daß der Algorithmus sehr viele Durchläufe braucht.

Probleme in RP $\cap$ coRP haben also für die Praxis sehr attraktive Eigenschaften—man kann sie "so gut wie" in Polynomialzeit lösen, ohne Fehler hinnehmen zu müssen. Die Klasse RP $\cap$ coRP wird auch mit ZPP (Zero error Probability, Polynomial time) bezeichnet, also

$$\text{ZPP} = \text{RP} \cap \text{coRP}.$$

Eine andere Charakterisierung von ZPP wird in Aufgabe 12.2 gezeigt. Satz 8.11 und das von uns nicht bewiesene Ergebnis von Adleman und Huang zeigen, daß PRIMES $\in$ ZPP.

Ein randomisiertes Verfahren, das nie eine falsche Antwort produziert, heißt auch *Las Vegas*-Algorithmus. Dem gegenüber stehen *Monte Carlo*-Algorithmen, die sich gelegentlich irren dürfen (dafür aber meistens eine feste Laufzeit haben).

RP erlaubt nur einseitige Fehler: Bei Eingaben, die nicht zur Sprache gehören, muß die Antwort immer korrekt sein. Manchmal ist diese Forderung zu streng, und wir müssen Fehler auch bei Eingaben zulassen, die nicht zur Sprache gehören. Zunächst generalisieren wir den Begriff der R-Akzeptanz.

**Definition:** Eine normale NTM $M$ *akzeptiert eine Sprache* $L$ *(probabilistisch) mit Fehlerwahrscheinlichkeiten* $(\epsilon_+, \epsilon_-)$ wenn das folgende gilt:
(1) Für $x \in L$ lehnt höchstens ein Bruchteil von $\epsilon_+$ der Blätter in $T_M(x)$ ab;
(2) Für $x \notin L$ akzeptiert höchstens ein Bruchteil von $\epsilon_-$ der Blätter in $T_M(x)$.
Wir erlauben, daß $\epsilon_+$ und $\epsilon_-$ Funktionen der Eingabelänge $|x|$ sind. Ist $\epsilon_+ = \epsilon_- = \epsilon$, sagen wir auch einfach "...mit Fehlerwahrscheinlichkeit $\epsilon$".

RP ist also die Klasse der Sprachen, die von normalen NTMs mit polynomieller Laufzeit und Fehlerwahrscheinlichkeiten $(1/2, 0)$ akzeptiert werden.

Wir könnten jetzt versuchen, einen doppelseitigen Fehler dadurch zuzulassen, daß wir $(1/2, 0)$ durch $(1/2, 1/2)$ ersetzen; aber das wäre Unsinn, denn nach einer solchen Definition würde jede Sprache in konstanter Zeit von einer Maschine probabilistisch akzeptiert, die die Eingabe ignoriert, eine Münze wirft und bei Kopf akzeptiert.

**Definition:** BPP (Bounded error Probability, Polynomial time) ist die Klasse der Sprachen, die von normalen NTMs mit polynomieller Laufzeit und Fehlerwahrscheinlichkeit $1/4$ akzeptiert werden.

Die Konstante $1/4$ in der Definition könnte durch jede andere Konstante $\epsilon$ mit $0 < \epsilon < 1/2$ ersetzt werden, und wie bei RP können wir die Fehlerwahrscheinlichkeit sogar

exponentiell klein machen. Hier ist es nicht ganz so offensichtlich, und wir beweisen es im folgenden Satz.

**Satz** 9.1: Sei $\epsilon$ eine Konstante mit $0 < \epsilon < 1/2$ und sei $L$ eine Sprache, die von einer normalen NTM $M$ in Polynomialzeit mit Fehlerwahrscheinlichkeit $\epsilon$ akzeptiert wird. Dann wird $L$ auch für jedes Polynom $p : \mathbb{N}_0 \to \mathbb{N}$ bei Eingabelänge $n$ von einer normalen NTM in Polynomialzeit mit Fehlerwahrscheinlichkeit $2^{-p(n)}$ akzeptiert.

**Beweis**: Die Idee ist einfach, viele unabhängige Berechnungen von $M$ durchzuführen und dem Mehrheitsentscheid (Akzeptieren oder Ablehnen) zu folgen. Wir müssen nur zeigen, daß das hinreichend zuverlässig ist.

Wir betrachten eine konkrete Eingabe $x$ mit $|x| = n$ und definieren $\epsilon_x \leq \epsilon$ als die Wahrscheinlichkeit, mit der $M$ bei Eingabe $x$ die falsche Antwort gibt. Seien $\delta = 1 - \epsilon$ und $\delta_x = 1 - \epsilon_x \geq \epsilon_x$. Da die Funktion $y \mapsto y(1-y)$ auf dem Intervall $[0, 1/2]$ streng monoton wächst und in $1/2$ ein Maximum von $1/4$ annimmt, ist $\delta_x \epsilon_x \leq \delta\epsilon < 1/4$.

$p(n)$ kann in Polynomialzeit berechnet werden. Wir führen die Berechnung von $M$ $m$-mal hintereinander aus, wobei $m = 2q + 1$ und $q \in \mathbb{N}$ später in Abhängigkeit von $p(n)$ festgelegt wird und durch ein Polynom beschränkt ist. Dadurch ist sichergestellt, daß unsere Laufzeit polynomiell ist. Wir akzeptieren, falls mehr als $q$ der Berechnungen von $M$ akzeptieren.

Unsere Antwort ist genau dann falsch, wenn $M$ höchstens $q$-mal die richtige Antwort liefert. Die Wahrscheinlichkeit hierfür ist genau

$$\sum_{j=0}^{q} \binom{m}{j} \delta_x^j \epsilon_x^{m-j} \leq \sum_{j=0}^{q} \binom{m}{j} \delta_x^{m/2} \epsilon_x^{m/2}$$

$$\leq (\delta\epsilon)^{m/2} \sum_{j=0}^{q} \binom{m}{j} \leq (\delta\epsilon)^{m/2} \cdot 2^m = (4\delta\epsilon)^{m/2} \leq (4\delta\epsilon)^q.$$

Wählen wir $c \in \mathbb{N}$, so daß $(4\delta\epsilon)^c \leq 1/2$, d.h. $c \geq -1/\log(4\delta\epsilon)$, und setzen $q = cp(n)$, ist die Fehlerwahrscheinlichkeit höchstens $2^{-p(n)}$. ∎

Wegen Satz 9.1 wird BPP als die probabilistische Klasse angesehen, deren Probleme noch mit vertretbarem Aufwand gelöst werden können: Die Fehlerwahrscheinlichkeit kann unter Wahrung der Polynomialzeit für praktische Zwecke verschwindend klein gemacht werden.

**Satz** 9.2: RP $\subseteq$ BPP.

**Beweis:** RP erlaubt die Fehlerwahrscheinlichkeiten $(1/2, 0)$. Durch zweimaliges Ausführen erreichen wir die Fehlerwahrscheinlichkeiten $(1/4, 0)$, die für die Definition von BPP hinreichend klein sind. ∎

**Satz** 9.3: RP ist gegenüber Bildung von Vereinigung und Durchschnitt abgeschlossen, und BPP ist gegenüber Bildung von Komplement, Vereinigung und Durchschnitt abgeschlossen.

**Beweis:** Aufgabe 12.1. ∎

Eine letzte probabilistische Komplexitätsklasse PP ist wie folgt definiert: $L \in$ PP, wenn es eine normale NTM $M$ mit polynomieller Laufzeit gibt, so daß $x \in L$ genau dann, wenn mehr als die Hälfte der Blätter in $T_M(x)$ akzeptiert. Wir sagen, daß $L$ die von $M$ *mit Mehrheit* akzeptierte Sprache ist.

Schauen wir uns die Definition von PP genauer an. "Mehr als die Hälfte" muß nicht viel mehr als die Hälfte sein, ganz anders als bei BPP, während bei Eingaben in $\overline{L}$ bis zur Hälfte der Blätter akzeptieren darf. Es ist nicht klar, daß wir mit vertretbarem Aufwand durch Experimente mit $M$ einigermaßen sicher entscheiden können, ob $x \in L$. Während Probleme in BPP, RP oder gar ZPP in der Praxis gut lösbar sind, können wir das keineswegs von einem Problem wissen, bloß weil es in PP liegt. Das wird auch vom folgenden Satz bestätigt.

**Satz** 9.4: NP $\subseteq$ PP $\subseteq$ PSPACE.

**Beweis:** Wir zeigen zuerst, daß NP $\subseteq$ PP. Sei $L \in$ NP und sei $M$ eine normale NTM, die $L$ in Polynomialzeit akzeptiert (Lemma 8.3). Wir modifizieren $M$ geringfügig durch Hinzufügen eines zusätzlichen Schrittes am Ende der Berechnung, in dem eine Münze geworfen wird. Die modifizierte Maschine $M'$ akzeptiert genau dann, wenn die ursprüngliche Berechnung akzeptiert *oder* die am Ende geworfene Münze Kopf zeigt.

Bei Eingaben in $\overline{L}$ akzeptiert genau die Hälfte der Blätter im Berechnungsbaum von $M'$, denn keine Berechnung von $M$ akzeptiert—es kommt also nur auf den Münzwurf an. Bei Eingaben in $L$ akzeptiert mehr als die Hälfte der Blätter im Berechnungsbaum von $M'$, denn mindestens eine Berechnung von $M$ akzeptiert. $L$ ist also genau die von $M'$ mit Mehrheit akzeptierte Sprache.

Wir zeigen jetzt, daß PP $\subseteq$ PSPACE. Sei $L \in$ PP und sei $M$ eine normale NTM, die $L$ in Polynomialzeit mit Mehrheit akzeptiert. Wir simulieren einfach nacheinander alle Berechnungen von $M$ auf einer vorliegenden Eingabe, zählen dabei deren Gesamtanzahl und die Anzahl der akzeptierenden Berechnungen und akzeptieren, falls der Bruchteil der akzeptierenden Berechnungen am Ende 1/2 übersteigt. ∎

Wir fragen uns nun, ob PP gegenüber Komplementbildung abgeschlossen ist. Man könnte versucht sein, einfach die Zustände `accept` und `reject` zu vertauschen. Wegen der leichten Asymmetrie in der Definition von PP ist das aber nicht korrekt—Eingaben, bei denen genau die Hälfte der Blätter akzeptiert, würden dabei nicht richtig behandelt werden (vgl. den Beweis von Satz 9.4). Solche Eingaben können aber vermieden werden.

**Satz** 9.5: PP ist gegenüber Komplementbildung abgeschlossen (d.h. PP = coPP).

**Beweis:** Sei $L \in$ PP und sei $M$ eine normale NTM, die $L$ in Polynomialzeit mit Mehrheit akzeptiert. Wir müssen nur zeigen, daß wir $M$ so modifieren können (ohne die mit Mehrheit akzeptierte Sprache zu verändern), daß für keine Eingabe $x$ genau die Hälfte der Blätter in $T_M(x)$ akzeptiert, denn danach können wir in der Tat `accept` und `reject` vertauschen.

Fig. 9.1 zeigt die Situation, die wir im Moment haben, sowie die, die wir erreichen wollen.

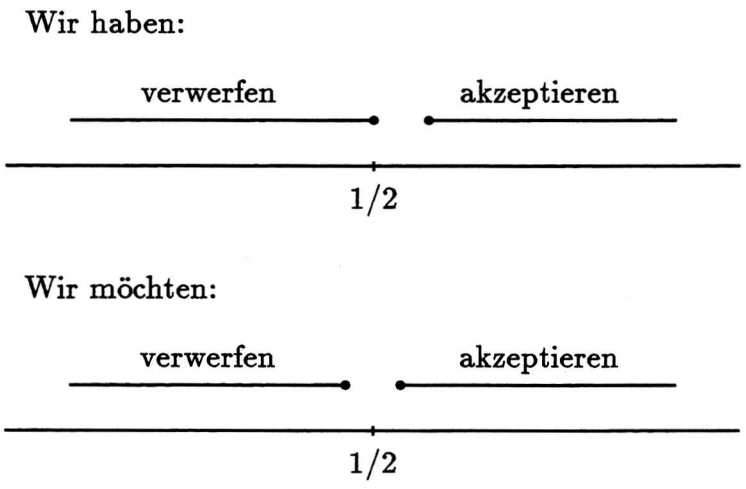



Fig. 9.1. Die Verschiebung der Akzeptwahrscheinlichkeit.

Diese "Umskalierung" erreichen wir, indem wir jede akzeptierende Endkonfiguration

"mit geringer Wahrscheinlichkeit verwerfend machen". Wir benutzen dazu eine normale NTM $M'$, die sich wie folgt verhält (Fig 9.2):

Simuliere $M$; das dauert $t$ Schritte;
**if** $M$ verwirft **then** verwirf
**else**
  Wirf eine Münze $t$-mal;
  **if** $t$-mal Kopf **then** verwirf **else** akzeptiere;

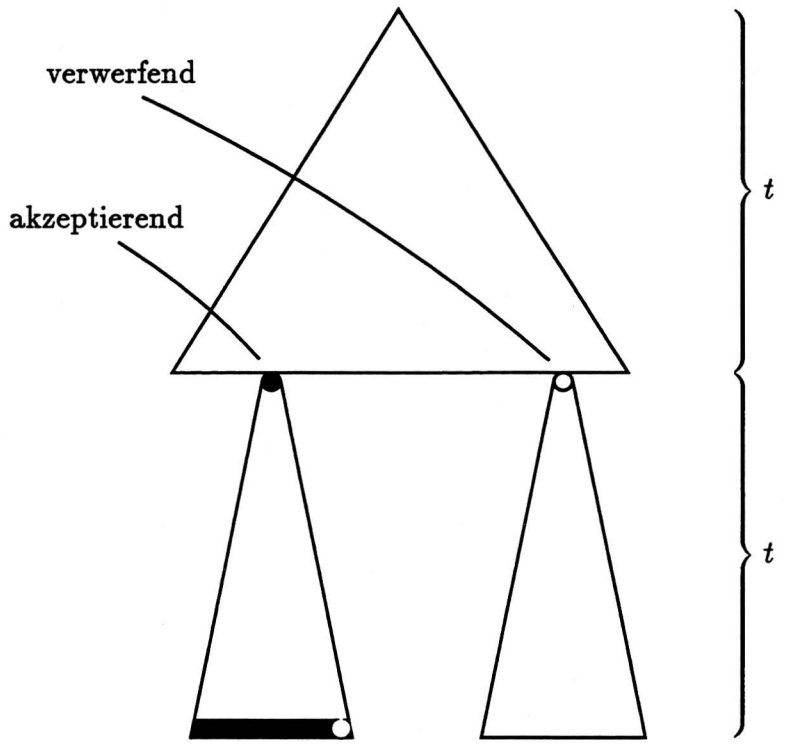



Fig. 9.2. Der Algorithmus aus dem Beweis von Satz 9.5.

Seien jetzt $a$ und $a'$ die Anzahlen der akzeptierenden Blätter in $T_M(x)$ und $T_{M'}(x)$ für eine vorliegende Eingabe $x$. Nach Konstruktion ist immer $a' = (2^t - 1)a$. O.B.d.A. ist $t \geq 2$. Dann gilt:

$a \leq 2^{t-1}$ ($M$ verwirft) $\Rightarrow a' < 2^{2t-1}$ ($M'$ verwirft).

$a > 2^{t-1}$ ($M$ akzeptiert) $\Rightarrow a \geq 2^{t-1} + 1 \Rightarrow a' \geq (2^t - 1)(2^{t-1} + 1) = 2^{2t-1} + 2^t - 2^{t-1} - 1 = 2^{2t-1} + (2^{t-1} - 1) > 2^{2t-1}$ ($M'$ akzeptiert).

$M'$ akzeptiert also auch die Sprache $L$ mit Mehrheit, und für keine Eingabe ist der Bruchteil der akzeptierenden Blätter genau $1/2$. ∎

Es kann bewiesen werden, daß PP auch gegenüber Bildung von Vereinigung und Durchschnitt abgeschlossen ist (R. Beigel, N. Reingold and D. Spielman, PP is closed under intersection, 23rd Annual ACM Symp. on Theory of Computing (1991), pp. 1–9).

Jede normale NTM akzeptiert eine Sprache mit Mehrheit. Damit hängt auch zusammen, daß wir relativ leicht eine Sprache angeben können, die für PP vollständig ist.

#SAT:

Eingabe: Eine Boolesche Formel $F$ und eine Zahl $k \in \mathbb{N}_0$.

Frage: Hat $F$ mehr als $k$ erfüllende Belegungen?

**Satz** 9.6: #SAT ist PP-vollständig.

**Beweis:** Wir zeigen zuerst, daß #SAT $\in$ PP. Sei $(F, k)$ eine Eingabe und seien $x_1, \ldots, x_m$ die in $F$ vorkommenden Variablen. Ist gerade $k = 2^{m-1}$, brauchen wir nur eine zufällige Belegung von $x_1, \ldots, x_m$ zu wählen, $F$ an dieser Belegung auszuwerten und zu akzeptieren, falls das Ergebnis *true* ist. Die Akzeptwahrscheinlichkeit ist genau dann größer als $1/2$, wenn $F$ mehr als $k$ erfüllende Belegungen hat, wie gewünscht. Ist $k$ aber zum Beispiel kleiner als $2^{m-1}$, ist die Akzeptwahrscheinlichkeit zu klein; wir müssen sie irgendwie größer machen. Das tun wir wie folgt, ähnlich wie im vorigen Beweis: Wir werfen zuerst eine Münze. Mit Wahrscheinlichkeit $1/2$ wählen wir eine zufällige Belegung von $x_1, \ldots, x_m$, werten $F$ an dieser Belegung aus und akzeptieren, falls das Ergebnis *true* ist. Mit Wahrscheinlichkeit $1/2$ akzeptieren wir statt dessen mit Wahrscheinlichkeit $1 - k/2^m$; das können wir tun, indem wir mit Hilfe von $m$ Münzwürfen eine zufällige Zahl $r$ aus dem Bereich $\{0, \ldots, 2^m - 1\}$ generieren und akzeptieren, falls $r < 2^m - k$. Hat $F$ genau $i$ erfüllende Belegungen, akzeptiert dieses Verfahren mit Wahrscheinlichkeit insgesamt $(1/2)(i/2^m + (1 - k/2^m)) = 1/2 + (i - k)/2^{m+1}$. Das ist genau dann mehr als $1/2$, wenn $i > k$, wie gewünscht.

Wir zeigen nun, daß $L \leq$ #SAT für jede Sprache $L \in$ PP. Die Idee ist wie folgt: $L$ wird mit Mehrheit von einer normalen NTM $M$ akzeptiert, von der wir o.B.d.A. annehmen können, daß sie nur ein Band besitzt. Damit können wir die im Beweis des Satzes von Cook benutzte Formel konstruieren, die eine erfüllende Belegung für jede akzeptierende Berechnung von $M$ hat. Damit müssen wir nur testen, ob die Anzahl der erfüllenden Belegungen von $F$ $2^{t-1}$ übersteigt, wobei $t$ die Laufzeit von $M$ ist. Es folgt ein detaillierteres Argument, da man an verschiedenen Stellen ein wenig aufpassen muß.

Sei also $L \in$ PP und sei $M$ eine normale NTM, die $L$ in Polynomialzeit mit Mehrheit

akzeptiert. Da wir uns vorstellen können, daß $M$ mit einer "Uhr" ausgestattet ist, die die Berechnung nach polynomieller Zeit beendet, können wir o.B.d.A. davon ausgehen, daß die Laufzeit $t$ von $M$ bei Eingaben der Länge $n$ in $O(\log n)$ Platz (als Teil der zu beschreibenden Reduktion) berechnet werden kann. Nach Bemerkung 2.4 wissen wir, daß es eine NTM $M'$ mit nur einem Band gibt, die die Berechnung von $M$ auf jeder Eingabe in Polynomialzeit simuliert.

Sei jetzt $x$ eine Eingabe mit $|x| = n$. Wir wissen, daß $x$ genau dann akzeptiert werden soll, wenn $M$ mehr als $k = 2^{t-1}$ akzeptierende Berechnungen auf $x$ hat. Da die Simulation von $M$ durch $M'$ keinen zusätzlichen Nichtdeterminismus einführt, soll $x$ auch genau dann akzeptiert werden, wenn $M'$ mehr als $k$ akzeptierende Berechnungen auf $x$ hat. Wir erstellen jetzt wie im Beweis von Satz 5.3 (Cook) eine Boolesche Formel $F$, die die Berechnung von $M'$ auf der Eingabe $x$ beschreibt—das können wir, weil $M'$ nur ein Band hat. Wir haben damals gezeigt, daß $F$ genau dann erfüllbar ist, wenn $M'$ mindestens eine akzeptierende Berechnung auf $x$ hat. Es gilt aber mehr: Die Anzahl der erfüllenden Belegungen von $F$ ist genau gleich der Anzahl der akzeptierenden Berechnungen von $M'$ auf $x$ (überlegen Sie). Also müssen wir als Ausgabe der Reduktion nur das Paar $(F, k)$ erstellen. Das geht unter Benutzung von nur logarithmischem Platz (bemerken Sie, daß $t$ in $O(\log t) = O(\log n)$ Bits dargestellt ist; daraus generieren wir die Zahl $k = 2^{t-1}$, binär in $t$ Bits dargestellt, unter Benutzung von $O(\log t)$ Arbeitsplatz). ∎

Das Diagramm in Fig. 9.3 zeigt einige Inklusionsrelationen zwischen probabilistischen Komplexitätsklassen, die wir entweder schon bewiesen haben oder die unmittelbar aus den schon bewiesenen Relationen folgen. Sind zwei Klassen durch eine Kante verbunden, bedeutet das, daß diejenige der beiden Klassen, die höher auf der Seite plaziert ist, die andere Klasse enthält, und die Kante ist mit den Nummern der relevanten Sätze beschriftet. Es fehlt noch die folgende einfache Beobachtung:

**Lemma** 9.7: Sind $C$ und $D$ Komplexitätsklassen, gilt $C \subseteq D \Rightarrow \text{co}C \subseteq \text{co}D$.

**Beweis:** $L \in \text{co}C \Rightarrow \overline{L} \in C \Rightarrow \overline{L} \in D \Rightarrow L \in \text{co}D$. ∎

Wie üblich ist es unbekannt, ob die Inklusionsrelationen aus Fig. 9.3 echt sind. Was das Diagramm nicht zeigt, ist mindestens so interessant wie das, was es zeigt. Insbesondere ist das Verhältnis zwischen BPP und NP gänzlich unbekannt; eine Klasse könnte in der anderen enthalten sein, oder BPP und NP könnten unter Inklusion unvergleichbar sein.

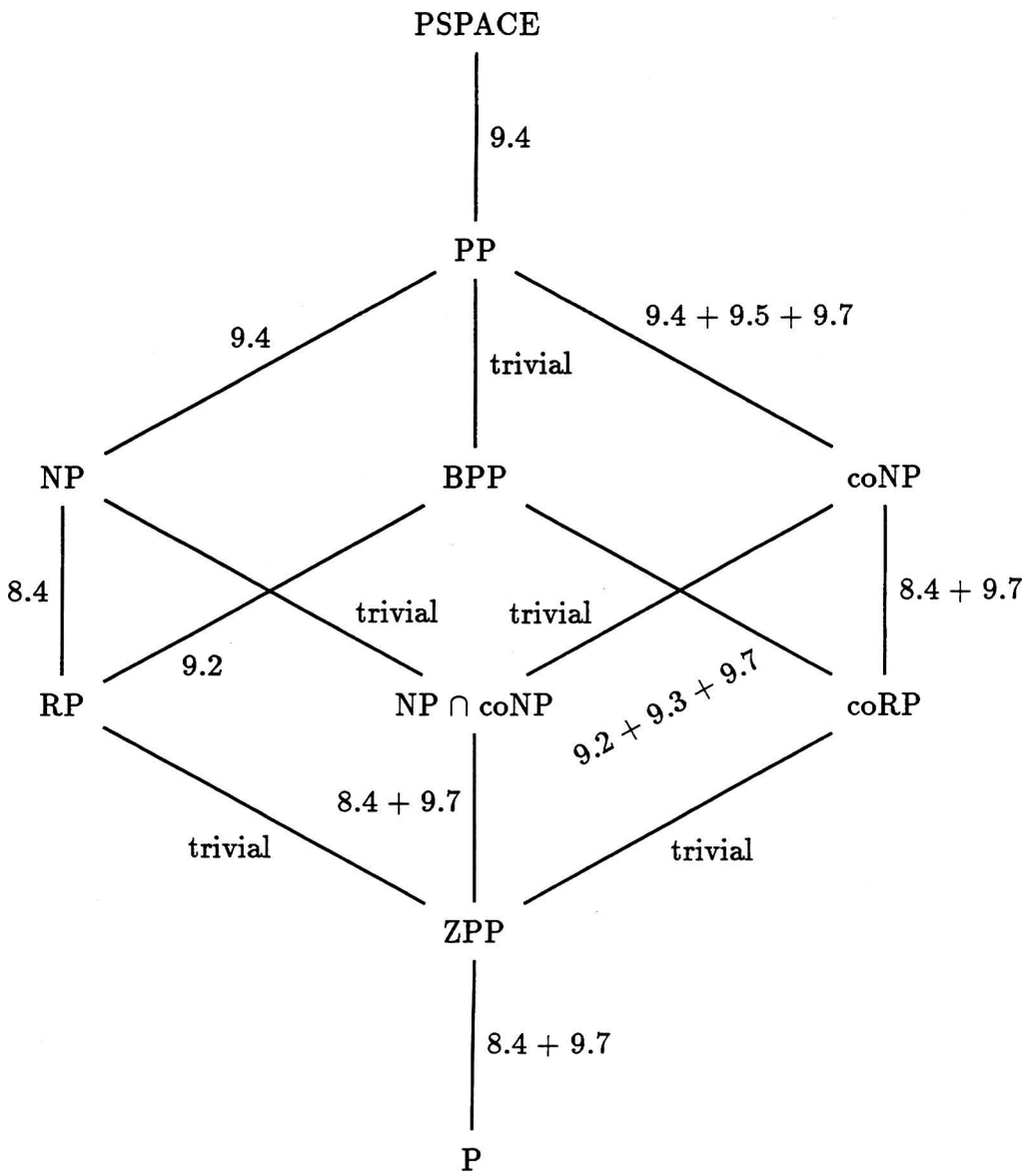



Fig. 9.3. Probabilistische Komplexitätsklassen.

# 10 Kryptographie

Alle bisherigen Kapitel haben sich in der gleichen, sehr einfachen "sozialen Situation" abgespielt: Es gab einen "guten" Algorithmus, der versucht hat, ein "böses" Problem zu besiegen. In diesem Kapitel beschäftigen wir uns mit komplizierteren Situationen, in denen es mehrere aktive Parteien gibt, die unterschiedliche Ziele verfolgen und nicht das gleiche Wissen besitzen.

Zuerst betrachten wir eine Situation, in der zwei Personen, die traditionell Alice und Bob heißen, geheime Nachrichten austauschen möchten, während eine dritte Person, Eve, durch Abhören versucht, hinter die Geheimnisse von Alice und Bob zu kommen. Wir interessieren uns hier nicht dafür, wie man Eve physikalisch daran hindern könnte, die ausgetauschten Nachrichten abzufangen, sondern gehen davon aus, daß Eve alle Nachrichten mithört, die zwischen Alice und Bob gesendet werden. Alice und Bob können ihre Nachrichten aber *verschlüsseln.* Wir werden hier ein besonders einfaches und attraktives Verschlüsselungsprotokoll kennenlernen, das RSA-Protokoll (RSA steht nicht für Republic of South Africa, sondern für Rivest/Shamir/Adleman).

Im folgenden beschreiben wir nur, wie Alice eine Nachricht an Bob übermitteln kann; die Kommunikation von Bob zu Alice verläuft genauso. Es gibt eine *Verschlüsselungsfunktion* $E$ und eine *Entschlüsselungsfunktion* $D$, die beide allgemein bekannt sind. $E$ nimmt als Argument eine Nachricht $x$—einen binär repräsentierten String—und produziert daraus eine verschlüsselte Nachricht $y$. $D$ nimmt $y$ als Argument und stellt daraus $x$ wieder her; $E$ und $D$ sollen also zueinander inverse Funktionen sein. Vorläufig gibt es keine Geheimhaltung: Jeder, also auch Eve, kann die Funktion $D$ auf $y$ anwenden, um die ursprüngliche Nachricht $x$ zu erfahren. Wir erweitern daher $E$ und $D$ um jeweils ein weiteres Argument, den *öffentlichen Schlüssel* $e$ und den *geheimen Schlüssel* $d$. $e$ wird von Bob bekanntgegeben und ist allgemein bekannt, während nur Bob $d$ kennt. Alice berechnet jetzt $y = E(e, x)$ und übermittelt diese verschlüsselte Nachricht an Bob. Bob berechnet $D(d, y)$, das wieder $x$ sein soll (Fig. 10.1). Ohne Kenntnis von $d$ kann Eve aber nicht das gleiche tun.

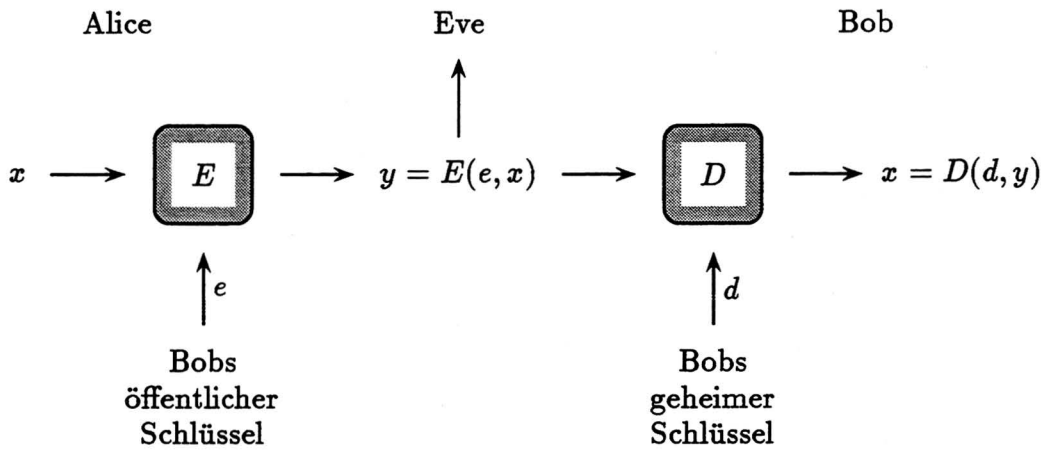



Fig. 10.1. Die Kommunikation von Alice zu Bob.

Wir müssen jetzt geeignete Funktionen $E$ und $D$ angeben und beschreiben, wie Bob zu den Schlüsseln $e$ und $d$ kommen kann. Das System soll für Nachrichten $x$ von $n$ Bits funktionieren; in der Praxis benutzt man Werte von $n$ von ein paar hundert. Bob fängt damit an, zwei große Primzahlen $p$ und $q$ zu bestimmen, deren Produkt $m$ etwas größer als $2^n$ ist; jede Nachricht kann damit als ein Element von $\{0,\ldots,m-1\}$ aufgefaßt werden. Außerdem bestimmt Bob zwei zu $\phi(m)$ teilerfremde Zahlen $d$ und $e$ aus $\{1,\ldots,\phi(m)\}$, die in der Gruppe $\mathbb{Z}^*_{\phi(m)}$ zueinander invers sind. Bob gibt $m$ und $e$ bekannt und behält $d$ sowie die Faktoren $p$ und $q$ von $m$ für sich. $E$ und $D$ sind jetzt die folgenden Funktionen:

$$E(e,x) = x^e \bmod m$$
$$D(d,y) = y^d \bmod m.$$

$E$ und $D$ sind also eigentlich die gleiche Funktion, modulare Potenzierung. Wir müssen jetzt überprüfen, daß das System korrekt, effizient und sicher ist.

*Korrektheit.* Hiermit meinen wir einfach, daß $D(d,E(e,x)) = x$ für jede Nachricht $x$, daß also die Entschlüsselung tatsächlich die ursprüngliche Nachricht liefert. Wir bemerken zuerst, daß

$$D(d,E(e,x)) = E(e,x)^d \bmod m = (x^e)^d \bmod m = x^{ed} \bmod m.$$

Da $e$ und $d$ Inverse modulo $\phi(m)$ sind, gibt es eine Zahl $k \in \mathbb{N}_0$, so daß $ed = 1 + k\phi(m)$.

Wenn $x \bmod p \neq 0$, ist

$$x^{p-1} \equiv 1 \pmod p$$

(Lemma 8.5). Da $p-1 \mid \phi(m) = (p-1)(q-1)$ (Lemma 7.10), folgt daraus

$$x^{1+k\phi(m)} \equiv x \pmod{p}.$$

Das gilt aber auch, wenn $x \bmod p = 0$. Analog ist

$$x^{1+k\phi(m)} \equiv x \pmod{q}.$$

Aus dem chinesischen Restsatz (Lemma 7.8) können wir dann schließen, daß

$$x^{ed} = x^{1+k\phi(m)} \equiv x \pmod{m},$$

also $D(d, E(e,x)) = x$.

*Effizienz.* Die Funktionen $E$ und $D$ auszuwerten ist einfach modulare Potenzierung. Da wir mit Zahlen von $O(n)$ Bits operieren, ist die benötigte Zeit polynomiell in $n$. Wir müssen jetzt nur noch den Aufwand abschätzen, der damit verbunden ist, die Parameter des Systems festzulegen.

Die Primzahlen $p$ und $q$ zu finden ist nicht schwierig: Zum einen haben wir einen effizienten randomisierten Primalitätstest (Satz 8.11). Zum anderen wissen wir, daß es viele Primzahlen gibt (Aufgabe 4.1). Man wähle also einfach zufällige Zahlen von etwas mehr als $n/2$ Bits, bis man zwei Primzahlen gefunden hat (um exakt zu sein, muß man hier einige zusätzliche Betrachtungen anstellen, um zu zeigen, daß jedes hinreichend große Intervall auch viele Primzahlen enthält). Danach kommt das Auffinden einer Zahl $e$ in $\mathbb{Z}^*_{\phi(m)}$. Da $\phi(m) = (p-1)(q-1)$, gibt es viele solche Zahlen, z.B. jede Primzahl, die größer als $\max\{p,q\}$, aber kleiner als $(p-1)(q-1)$ ist, und wir haben auch hier einen effizienten Test. Schließlich muß das Inverse $d$ zu $e$ in $\mathbb{Z}^*_{\phi(m)}$ berechnet werden, aber das kann sehr leicht mit dem Euklidischen Algorithmus geschehen. Die Zeit, die für die gesamte Vorberechnung benötigt wird, ist mit hoher Wahrscheinlichkeit polynomiell in $n$.

*Sicherheit.* Hiermit meinen wir, daß Eve ohne Kenntnis des geheimen Schlüssels $d$ mit der verschlüsselten Nachricht $y$ sowie den öffentlichen Parametern $m$ und $e$ nichts anfangen kann. Wenn Eve über genügend große Berechnungsressourcen verfügt, ist das allerdings nicht wahr: Es reicht, $m$ durch Ausprobieren aller möglichen Faktoren $p$ zu faktorisieren. Danach kann $d$ leicht bestimmt werden—Bob führt dieselbe Berechnung aus. Ist insbesondere P = NP, kann $d$ in Polynomialzeit bestimmt werden, da das Entscheidungsproblem, ob das erste (zweite, usw.) Bit von $d$ 1 ist, nach dem obigen Argument sicherlich in NP liegt. Also kann das RSA-System nur dann sicher sein, wenn P $\neq$ NP. Na gut, damit

können wir uns vielleicht auch zufrieden geben, denn sollte jemand P = NP zeigen, wird er wahrscheinlich etwas Besseres mit diesem Wissen anzufangen wissen, als Alice und Bob zu belauschen. Nur leider können wir nicht einmal zeigen, daß das System sicher ist, wenn P $\neq$ NP. Die Sicherheit des RSA-Systems ist im wesentlichen eine Erfahrungssache. Viele Leute haben versucht, das System zu "brechen", aber bisher ist es keinem gelungen (zumindest hat es niemand öffentlich angekündigt). Gelingt es Eve, die Zahl $m$ zu faktorisieren, kann sie die Sicherheit des Systems brechen. Die Erfahrung lehrt aber, daß es schwierig ist, große Zahlen zu faktorisieren, und viel mehr können wir in puncto Sicherheit nicht sagen.

Varianten des RSA-Protokolls haben neben der geheimen Nachrichtenübertragung viele andere Anwendungen. Wir betrachten hier kurz zwei davon.

**Digitale Unterschriften.**

Wie kann Bob eine Nachricht $x$, z.B. einen Kaufvertrag, unterschreiben, bevor er sie an Alice übermittelt, so daß Alice später, z.B. vor Gericht, beweisen kann, daß die Nachricht tatsächlich von Bob kam? Zuerst beschreiben wir eine Lösung, die nicht ganz korrekt ist.

Bob benutzt hierzu wieder seinen öffentlichen Schlüssel $e$ und seinen privaten Schlüssel $d$. Die von Bob unterschriebene Nachricht $x$ wird durch die Zahl $D(d,x)$ dargestellt. Eigentlich ist es Unsinn, die Entschlüsselungsfunktion $D$ auf $x$ anzuwenden, denn $x$ ist gar nicht verschlüsselt. Es ist aber nicht nur

$$D(d, E(e,x)) = x$$

für alle $x \in \{0,\ldots,m-1\}$ (Korrektheit), sondern auch

$$E(e, D(d,x)) = x$$

für alle $x \in \{0,\ldots,m-1\}$ (Kommutativität); beide Größen sind ja nichts anderes als $x^{ed} \bmod m$. Beim Empfang einer unterschriebenen Nachricht $y$ kann Alice daher unter Benutzung des öffentlichen Schlüssels von Bob die ursprüngliche Nachricht $x$ als $E(e,y)$ erhalten. Auf den ersten Blick sieht es auch so aus, als ob tatsächlich niemand außer Bob die Nachricht abgeschickt haben könnte, denn kann man $y = D(d,x)$ ohne Kenntnis von $d$ berechnen, hat man das RSA-System gebrochen. Hier müssen wir aber aufpassen: Es ist zwar (vermutlich) schwer, $y$ aus $x$ zu berechnen, aber es ist leicht, $x = E(e,y)$ aus $y$ zu berechnen. Ein Widersacher könnte also solange zufällige Strings $y$ wählen und die dazugehörigen Nachrichten $x$ generieren, bis er auf eine Nachricht stößt, die er Bob in die

Schuhe schieben möchte. Man würde zwar erwarten, daß die allermeisten so (praktisch zufällig) generierten Nachrichten $x$ keinerlei Sinn ergeben, aber darauf möchten wir uns lieber nicht verlassen.

Lösung: Wir benutzen zusätzlich eine feste Funktion $F$, die niemand umzukehren weiß (z.B. die Bijektion $x \mapsto a^x \pmod p$, wobei $p$ eine große Primzahl und $a$ ein erzeugendes Element in $\mathbb{Z}_p^*$ ist, deren Umkehrfunktion die "diskrete Logarithmenfunktion" ist), und Bob schickt nicht $D(d,x)$, sondern $(x, F(x), D(d, F(x)))$. Ein mittleres Element $F(x)$, das mit dem ersten Element $x$ *oder* mit dem letzten Element $D(d,F(x))$ konsistent ist, kann man leicht finden, aber ein total konsistentes Tripel ohne Kenntnis von $d$ zu generieren ist jetzt (vermutlich) schwer.

**Münzwürfe per Telefon.**

Alice und Bob möchten am Telefon auslosen, wer von ihnen die gewonnene Reise für eine Person nach Martinique antreten darf. Alice könnte natürlich eine Münze werfen und Bob das Ergebnis mitteilen, aber diese Methode hat offensichtliche Nachteile. Besser ist es, so vorzugehen: Alice bestimmt die Parameter $p$, $q$, $e$ und $d$ eines RSA-Systems, behält sie aber alle bis auf $m = pq$ für sich. Außerdem vereinbaren Alice und Bob zwei große verschiedene Zahlen $x$ und $y$ aus $\{1, \ldots, m-1\}$, von denen eine "Martinique" und die andere "zu Hause bleiben" repräsentiert. Alice berechnet jetzt $E(e,x)$ und $E(e,y)$ und schickt diese beiden Zahlen in einer zufällig gewählten Reihenfolge an Bob. Da Bob $d$ nicht kennt, hat er keine Möglichkeit, herauszufinden, welche der beiden Nachrichten "Martinique" bedeutet. Er behält eine der beiden Nachrichten und schickt die andere an Alice zurück, womit die Entscheidung gefallen ist. Alice verrät danach $e$ und $d$, so daß das Ergebnis der Auslosung auch von Bob überprüft werden kann.

Bob kann nicht mogeln, ohne das RSA-System zu brechen (allerdings sollte man bei der Wahl von $x$ und $y$ etwas aufpassen; z.B. ist es keine gute Idee, $x$ oder $y$ als 1 zu wählen). Alice könnte mogeln, wenn sie es schafft, zwei verschiedene Paare $(e_1, d_1)$ und $(e_2, d_2)$ zu finden, so daß $E(e_1, x) = E(e_2, y)$ und $E(e_1, y) = E(e_2, x)$—sie würde am Ende dann gerade dasjenige RSA-System verraten, das das für sie günstige Ereignis impliziert. Dieses Unterfangen sieht allerdings nicht viel leichter aus, als das RSA-System zu brechen.

Man kann zu Recht mit der Sicherheit der in diesem Kapitel dargestellten kryptographischen Verfahren unzufrieden sein. Obwohl scheinbar niemand in der Lage ist, sie zu brechen, kann man nicht beweisen, daß es unmöglich ist, nicht einmal unter plausiblen Annahmen wie P $\neq$ NP. Das ist allerdings der momentane Stand der Dinge: Vorbehaltlos sichere Kryptosysteme sind nicht bekannt.

# 11 Approximationsalgorithmen

Die Welt ist voller Berechnungsprobleme, die wir für realistische Problemgrößen nicht exakt lösen können. Viele dieser Probleme sind *Optimierungsprobleme*, die in folgender allgemeinen Form beschrieben werden können: Zu jeder Eingabe $x$ gibt es eine nichtleere endliche Menge $Z(x)$ von *zulässigen Lösungen*. Es gibt auch eine Funktion $c$, die jede zulässige Lösung $z$ auf eine positive reelle Zahl $c(z)$ abbildet. Wir nennen $c$ die *Kostenfunktion* und $c(z)$ die Kosten von $z$. Gesucht ist eine zulässige Lösung $z_0$, die $c$ minimiert, also ein $z_0 \in Z(x)$ mit

$$c(z_0) = \min\{c(z) \mid z \in Z(x)\}.$$

Jedes solche $z_0$ heißt *optimale Lösung*, und seine Kosten sind die *optimalen Kosten*.

**Beispiel:** Eine Optimierungsvariante des Traveling-Salesman-Problems ist wie folgt definiert: Die Eingabe $x$ ist ein vollständiger ungerichteter Graph $G = (V, E)$, in dem jede Kante mit einer Länge aus $\mathbb{N}$ beschriftet ist. Eine zulässige Lösung $z$ ist eine Tour in $G$, und wir definieren ihre Kosten als ihre Gesamtlänge. Eine optimale Lösung ist somit eine kürzeste Tour.

Wir wissen aus Satz 6.5, daß die Entscheidung NP-vollständig ist, ob ein ungerichteter Graph mit Kantenlängen eine Tour besitzt, deren Gesamtlänge durch eine vorgegebene Schranke $k$ beschränkt ist. Daher sollten wir uns keine Hoffnung machen, das entsprechende Optimierungsproblem exakt in Polynomialzeit zu lösen. Wir könnten aber immer noch versuchen, eine Tour zu finden, deren Gesamtlänge die minimale Länge um höchstens 10% übersteigt. Satz 6.5 sagt nichts darüber aus, ob das in Polynomialzeit möglich ist, und eine derartige fast optimale Lösung könnte für eine bestimmte Anwendung durchaus noch akzeptabel sein. Mit anderen Worten könnten wir versuchen, einen *Approximationsalgorithmus* für das Traveling-Salesman-Problem zu entwerfen.

Betrachten wir jetzt ein Optimierungsproblem $P$ mit Kostenfunktion $c$ und bezeichnen die optimalen Kosten einer zulässigen Lösung bei Eingabe $x$ mit $Opt(x)$. Ein *Approximationsalgorithmus* für $P$ ist einfach ein Algorithmus $A$, der zu jeder Eingabe $x$ eine zulässige

Lösung produziert, die wir mit $A(x)$ bezeichnen wollen. Es ist immer $c(A(x)) \geq Opt(x)$; wenn der Algorithmus $A$ etwas taugt, ist $c(A(x))$ aber nicht allzuviel größer als $Opt(x)$. Wir definieren die *Approximationsgüte* von $A$ bei Eingabe $x$ als das Verhältnis

$$\rho(x) = \frac{c(A(x))}{Opt(x)},$$

das immer $\geq 1$ ist. Weiter definieren wir die Approximationsgüte von $A$ bei Eingabelänge $n$ als $\rho(n) = \max_{|x|=n} \rho(x)$. Ist $\rho$ nach oben beschränkt, können wir auch die Approximationsgüte von $A$ schlechthin als $\sup_{n \in \mathbb{N}_0} \rho(n)$ definieren.

Unser erster konkreter Approximationsalgorithmus ist für eine Optimierungsvariante des NP-vollständigen Problems VERTEX COVER (Satz 6.2).

MINIMUM VERTEX COVER:

Eingabe: Ein ungerichteter Graph $G$.

Ausgabe: Ein minimales Vertex Cover in $G$.

Bei einem Eingabegraphen $G = (V, E)$ besteht die Menge $Z(G)$ der zulässigen Lösungen also aus allen Knotenmengen $U \subseteq V$, die von jeder Kante in $E$ mindestens einen Endpunkt enthalten. Die Kosten einer zulässigen Lösung $U$ sind einfach ihre Größe $|U|$.

**Satz** 11.1: Es gibt einen Approximationsalgorithmus für MINIMUM VERTEX COVER mit polynomieller Laufzeit und Approximationsgüte $\leq 2$.

**Beweis**: Wir benutzen den folgenden Algorithmus:

$U := \emptyset$;
**while** $G$ enthält mindestens eine Kante **do**
  Wähle eine Kante $\{u, v\}$ aus $G$;
  $U := U \cup \{u, v\}$;
  Entferne $u$ und $v$ und alle ihre inzidenten Kanten aus $G$;

Es ist offensichtlich, daß der Algorithmus terminiert, in Polynomialzeit ausgeführt werden kann und eine Knotenmenge $U$ berechnet. $U$ ist überdeckend, denn eine Kante wird nur dann aus $G$ entfernt, wenn einer ihrer Endpunkte schon in $U$ aufgenommen worden ist. Wir müssen jetzt zeigen, daß $U$ höchstens doppelt so groß ist wie eine beliebige überdeckende Knotenmenge $W$. Aber das folgt daraus, daß die Kanten $\{u, v\}$, die vom Algorithmus gewählt wurden, keine gemeinsamen Endpunkte haben: Von jeder dieser Kanten muß mindestens ein Endpunkt in $W$ sein; $U$ enthält alle Endpunkte dieser Kanten und keine weiteren Knoten; also ist $|U| \leq 2|W|$. ∎

Damit haben wir seltsamerweise auch einen der besten bekannten Approximationsalgorithmen für MINIMUM VERTEX COVER kennengelernt: Es ist kein Verfahren bekannt, das in Polynomialzeit eine Approximationsgüte von z.B. 1,99 garantiert. Siehe hierzu auch Aufgabe 13.1.

Bisher haben wir der Einfachheit halber nur *Minimierungsprobleme* betrachtet, bei denen der Wert einer Zielfunktion $c$ so klein wie möglich gemacht werden soll. Es gibt auch Optimierungsprobleme, die sich am natürlichsten als *Maximierungsprobleme* formulieren lassen. In solchen Fällen definieren wir die Approximationsgüte eines Algorithmus $A$ bei Eingabe $x$ als

$$\rho(x) = \frac{Opt(x)}{c(A(x))},$$

wieder eine Zahl $\geq 1$, und erweitern die verwandten Definitionen entsprechend. Wir betrachten als nächstes ein Maximierungsproblem.

MAXIMUM CUT:

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

Ausgabe: Eine Knotenmenge $U \subseteq V$, die die Anzahl der Kanten aus $E$ mit genau einem Endpunkt in $U$ maximiert.

**Satz** 11.2: Es gibt einen Approximationsalgorithmus für MAXIMUM CUT mit polynomieller Laufzeit und Approximationsgüte $\leq 2$.

**Beweis:** Für $U \subseteq V$ sei $c(U) = |\{\{u,v\} \in E : |\{u,v\} \cap U| = 1\}|$; das ist gerade die Größe, die wir maximieren wollen. Wir benutzen den folgenden Algorithmus:

$U :=$ eine beliebige Teilmenge von $V$ (z.B. $U = \emptyset$);
**while** $c(U)$ kann erhöht werden, indem ein Knoten
    zu $U$ hinzugenommen oder aus $U$ entfernt wird **do**
erhöhe $c(U)$ auf diese Weise;

Jede Iteration der **while**-Schleife kann in Polynomialzeit ausgeführt werden. Es gibt höchstens $|E|$ Iterationen, da jede Iteration $c(U)$ um mindestens 1 erhöht, während $c(U)$ nie größer als $|E|$ werden kann. Also läuft das Verfahren insgesamt in Polynomialzeit. Wir müssen jetzt nur zeigen, daß nach der Ausführung $c(U) \geq \frac{1}{2}c(W)$ für eine beliebige Teilmenge $W \subseteq V$. Aber dazu müssen wir nur beobachten, daß nach der Ausführung für jeden Knoten $v \in V$ mindestens die Hälfte der zu $v$ inzidenten Kanten "in $c(U)$ gezählt wird"; sonst könnte $c(U)$ nämlich dadurch noch erhöht werden, daß $v$ auf die andere Seite gebracht wird. Summiert über alle Knoten $v$ ergibt das $c(U) \geq \frac{1}{2}|E| \geq \frac{1}{2}c(W)$. ∎

Das Problem, bei einer vorgegebenen Schranke $k$ zu entscheiden, ob es eine Menge $U \subseteq V$ mit $c(U) \geq k$ gibt, ist das Entscheidungsproblem, das MAXIMUM CUT entspricht. Ist dieses Entscheidungsproblem NP-vollständig? Ja, aber das haben wir nicht gezeigt, denn Satz 6.9 befaßte sich mit *gewichteten* Graphen (die Gewichte hießen damals Kapazitäten), während der obige Algorithmus auf einem ungerichteten Graphen arbeitet. Der Algorithmus wird leicht mit Kantengewichten aus $\mathbb{N}$ fertig—dazu muß nur $c(U)$ in der offensichtlichen Weise umdefiniert werden—aber die Laufzeit leidet darunter. Unser Argument war ja einfach, daß jede Iteration $c(U)$ erhöht, während stets $c(U) \leq |E|$. Diese letzte Relation gilt jetzt nicht mehr, und wir können nur schließen, daß die Anzahl der Iterationen durch das totale Kantengewicht beschränkt ist. Betrachten wir jetzt eine Variante des Problems, bei dem Kantengewichte zugelassen sind, aber bei Eingabegröße $n$ höchstens $p(n)$ groß sein dürfen, wobei $p$ ein Polynom ist. Wie wir gerade gesehen haben, arbeitet unser Algorithmus bei dieser Variante des Problems immer noch in Polynomialzeit. Umgekehrt können wir aber zeigen, daß das entsprechende Entscheidungsproblem bei geeigneter Wahl von $p$ schon NP-vollständig ist. Dazu müssen wir nur beobachten, daß alle Kantengrößen, die im Beweis von Satz 6.9 auftreten, durch $3n$ beschränkt sind.

Der Algorithmus aus dem Beweis von Satz 11.2 versucht, eine zulässige Lösung "in kleinen Schritten" zu verbessern, indem untersucht wird, ob durch eine kleine Veränderung in der Lösung eine bessere Lösung erreicht werden kann. Diese Idee ist ein ganz allgemeines Prinzip, das unter Namen wie "Local Search" oder "Hill-Climbing" bekannt ist. Es wird in der Praxis bei der Lösung der verschiedensten Probleme eingesetzt. Allerdings kann man meist nicht viel über das Verhalten von Algorithmen beweisen, die auf Local Search basieren—das obige Argument war eine der seltenen Ausnahmen.

Bisher haben wir zwei Approximationsverfahren kennengelernt, die eine Güte von $\leq 2$ garantieren. Jetzt wollen wir ein Problem betrachten, für das wir etwas erheblich Besseres erreichen können. In der Tat werden wir einen *exakten* Algorithmus für eine Optimierungsvariante des NP-vollständigen KNAPSACK-Problems beschreiben!

MAXIMUM-VALUE KNAPSACK:

Eingabe: $2m + 1$ positive ganze Zahlen $w_1, \ldots, w_m, W, v_1, \ldots, v_m$.

Ausgabe: Eine Teilmenge $I \subseteq \{1, \ldots, m\}$ mit $\sum_{i \in I} w_i \leq W$, die $\sum_{i \in I} v_i$ maximiert.

Verglichen mit dem Entscheidungsproblem KNAPSACK verlangen wir also immer noch, daß das Gesamtgewicht durch $W$ beschränkt ist, aber wir maximieren den damit verbundenen Gesamtwert, anstatt zu verlangen, daß er mindestens $k$ ist.

Sei $V = \max\{v_i \mid 1 \leq i \leq m\}$. Unser Algorithmus berechnet eine Tabelle $A[0..m,$

$-V..mV]$ mit der folgenden Interpretation: Für $j = 0,\ldots,m$ und $v = -V,\ldots,mV$ soll $A[j,v]$ das kleinste Gewicht einer Teilmenge der $j$ ersten Gegenstände sein, deren Gesamtwert exakt $v$ ist, also

$$A[j,v] = \min\Big\{\sum_{i\in I} w_i \mid I \subseteq \{1,\ldots,j\} \text{ und } \sum_{i\in I} v_i = v\Big\}.$$

Existiert keine solche Teilmenge, setzen wir $A[j,v] = \infty$.

Für $v = -V,\ldots,-1$ können wir sofort $A[j,v] = \infty$ für $j = 0,\ldots,m$ setzen, denn einen negativen Gesamtwert können wir überhaupt nicht erreichen. Weiter ist $A[0,0] = 0$ und $A[0,v] = \infty$ für $v = 1,\ldots,mV$, denn ohne Gegenstände können wir den Gesamtwert 0 bei Gesamtgewicht 0 und sonst keinen Gesamtwert erreichen. Die restlichen Einträge können mit dem folgenden Algorithmus berechnet werden:

**for** $j := 1$ **to** $m$ **do**
  **for** $v := 0$ **to** $mV$ **do**
    $A[j,v] := \min\{A[j-1,v], A[j-1,v-v_j] + w_j\}$;

Begründung: Einen Gesamtwert von $v$ unter Benutzung von einigen der ersten $j$ Gegenstände können wir auf zwei Arten erreichen. Entweder benutzen wir den $j$ten Gegenstand nicht; dann ist das kleinstmögliche Gewicht weiterhin $A[j-1,v]$. Oder wir benutzen den $j$ten Gegenstand. Dann müssen wir aus den $j-1$ ersten Gegenständen einen Gesamtwert von genau $v - v_j$ erreichen, und das können wir mit Minimalgewicht $A[j-1,v-v_j]$; dazu kommt dann das Gewicht $w_j$ des $j$ten Gegenstandes. Das kleinstmögliche Gewicht, das mit Gesamtwert $v$ verbunden werden kann, errechnet sich aus dem Minimum dieser beiden Möglichkeiten, und genau das wird im Programm gemacht. Eine Möglichkeit (oder beide), den Gesamtwert von $v$ zu erreichen, läßt sich vielleicht nicht realisieren (z.B. könnte $v$ kleiner als $v_j$ sein, so daß der $j$te Gegenstand nicht benutzt werden kann). In diesem Fall trifft der Algorithmus aber auf den Wert $\infty$, der die Bildung des Minimums nicht beeinflußt.

Haben wir erst einmal die Tabelle $A$ berechnet, können wir durch Betrachten ihrer letzten Reihe den maximalen Wert feststellen, den wir bei Gewicht höchstens $W$ erreichen können; das ist nämlich der Index der letzten (rechtesten) Spalte, die einen Eintrag $\leq W$ hat. Wir können auch leicht "zurückverfolgen", wie dieser Wert zustande gekommen ist, und dadurch eine optimale Lösung $I$ bestimmen. Damit haben wir

**Satz** 11.3: Es gibt eine Konstante $t$, so daß Instanzen des MAXIMUM-VALUE KNAPSACK-Problems der Größe $n$ und mit Maximalwert $V$ in $O((nV)^t)$ Zeit (exakt) gelöst werden können.

Ist diese polynomielle Laufzeit ein Widerspruch zur NP-Vollständigkeit des KNAPSACK-Problems (Satz 6.11)? Nein, denn als Argument des Polynoms tritt nicht nur die Eingabegröße $n$ auf, sondern auch der Parameter $V$, der schlimmstenfalls exponentiell in $n$ sein kann ($V$ ist binär in der Eingabe repräsentiert). Also haben wir nicht gezeigt, daß P = NP. Dafür haben wir aber einen guten Ausgangspunkt gewonnen, um einen Approximationsalgorithmus für MAXIMUM-VALUE KNAPSACK zu entwickeln, der in echter Polynomialzeit läuft. Wir benutzen dabei die sogenannte *Skalierungstechnik*.

Wie können wir mit großen Werten von $V$ fertig werden, die die Laufzeit in die Höhe treiben? Nun, wir teilen $v_1, \ldots, v_m$ durch eine geeignete Zahl $K$—das kriegt selbst die größte Zahl klein, verändert aber die optimale Lösung nicht. Das Problem dabei ist, daß die so modifizierten Zahlen $v'_1, \ldots, v'_m$ nicht notwendigerweise ganz sind, so daß unser Algorithmus von vorhin mit Sicherheit nicht mehr funktioniert (die Werte werden als Arrayindizes benutzt!). Da wir ja aber sowieso darauf verzichten, die exakte Lösung zu finden, runden wir einfach $v'_1, \ldots, v'_m$ zur nächstkleineren ganzen Zahl. Wir setzen also

$$v'_i = \left\lfloor \frac{v_i}{K} \right\rfloor$$

für $i = 1, \ldots, m$ und lösen das so modifizierte Problem exakt mit dem obigen Verfahren (Gegenstände, die den modifizierten Wert 0 bekommen, also formal unzulässig sind, können einfach ignoriert werden). Nehmen wir an, wir erhalten dadurch die (natürlich zulässige) Lösung $I$, und sei $I^*$ eine optimale Lösung des ursprünglichen Problems. Dann gilt

$$c(I) = \sum_{i \in I} v_i \geq K \sum_{i \in I} v'_i \geq K \sum_{i \in I^*} v'_i \geq K \sum_{i \in I^*} (v_i/K - 1) = \sum_{i \in I^*} v_i - K|I^*| \geq c(I^*) - Km.$$

Wir können o.B.d.A. davon ausgehen, daß der wertvollste Gegenstand ganz allein eine zulässige Lösung bildet (sonst entferne vorher alle Gegenstände, die so schwer sind, daß sie, allein genommen, schon die Gewichtsgrenze verletzen). Damit können wir auch trivialerweise dafür sorgen, daß $c(I) \geq V$. Dann ist also

$$c(I^*) \leq c(I) + Km \leq c(I)\left(1 + \frac{Km}{V}\right).$$

Sei jetzt $\epsilon$ eine vorgegebene positive rationale Zahl. Wir wählen

$$K = \max\left\{\left\lfloor \frac{\epsilon V}{m} \right\rfloor, 1\right\}$$

und untersuchen, wie es mit der Approximationsgüte und der Laufzeit steht. Wenn $K = 1$, haben wir das Problem gar nicht verändert ($v_i' = v_i$ für $i = 1, \ldots, m$), so daß die gefundene Lösung exakt ist. Andernfalls ist $K \leq \epsilon V/m$ und daher

$$\frac{c(I^*)}{c(I)} \leq \left(1 + \frac{Km}{V}\right) \leq 1 + \epsilon.$$

In beiden Fällen ist die Approximationsgüte also höchstens $1 + \epsilon$.

Die Laufzeit wird von dem größten Wert in dem modifizierten Problem bestimmt, also von

$$V' = \left\lfloor \frac{V}{K} \right\rfloor.$$

Da

$$K = \max\left\{ \left\lfloor \frac{\epsilon V}{m} \right\rfloor, 1 \right\} \geq \frac{\epsilon V}{2m},$$

ist aber $V' \leq \frac{2m}{\epsilon}$. Daraus sehen wir:

**Satz** 11.4: Es gibt eine Konstante $t$ und einen Approximationsalgorithmus für das MAXIMUM-VALUE KNAPSACK-Problem, der als Eingabe eine Instanz des Problems der Größe $n$ und eine positive rationale Zahl $\epsilon$ nimmt, Approximationsgüte höchstens $1 + \epsilon$ erreicht und Laufzeit $O((n/\epsilon)^t)$ hat.

Hier erreichen wir also in Polynomialzeit nicht nur Approximationsgüte 2, sondern $1 + \epsilon$ für jedes feste $\epsilon > 0$. Es ist nützlich, sich noch einmal vor Augen zu führen, wie Satz 11.4 durch Satz 11.3 impliziert wurde.

Wir betrachten jetzt eine Optimierungsvariante des NP-vollständigen BIN PACKING-Problems (Satz 6.12). Aus Gründen der Bequemlichkeit vereinbaren wir eine feste Bin-Größe von 1.

MINIMUM BIN PACKING:

Eingabe: $m$ rationale Zahlen $v_1, \ldots, v_m$ mit $0 \leq v_i \leq 1$ für $i = 1, \ldots, m$.

Ausgabe: Eine Partition von $v_1, \ldots, v_m$ in möglichst wenig Gruppen, so daß die Summe der Zahlen in jeder Gruppe höchstens 1 beträgt.

Die zu minimierende Größe ist natürlich die Anzahl der benutzten Bins (d.h. Gruppen).

Können wir auch für MINIMUM BIN PACKING in Polynomialzeit eine Approximationsgüte von $1 + \epsilon$ für jedes $\epsilon > 0$ erreichen? Nein, denn für $\epsilon < 1/2$ müßten wir dann, wenn die optimale Binzahl 2 ist, auch eine Lösung berechnen, die mit 2 Bins auskommt

(3 Bins wären schon nicht mehr zulässig). Wir wissen aber aus dem Beweis von Satz 6.12, daß die Entscheidung, ob 2 Bins ausreichen, NP-vollständig ist.

Diese Antwort ist allerdings etwas unbefriedigend, denn alles, was wir sagen, ist, daß wir manchmal 3 Bins benutzen müssen, wenn 2 gereicht hätten. Was wäre, wenn wir immer mit einem zusätzlichen Bin auskommen? Das wäre ein wunderbarer Approximationsalgorithmus, auch wenn die formale Definition einer kleinen Approximationsgüte nicht erfüllt ist. Ein solches Ergebnis ist nicht bekannt, aber wir werden jetzt ein Ergebnis sehen, das ein wenig in diese Richtung geht: Für jedes feste $\epsilon > 0$ können wir in Polynomialzeit unter Benutzung von höchstens $(1+\epsilon)c^* + 1$ Bins packen, wobei $c^*$ die optimale Binanzahl ist. Der Beweis ist relativ kompliziert. Wir geben zunächst einen exakten Algorithmus für den Fall an, daß es nur wenige verschiedene Objektgrößen gibt.

**Lemma** 11.5: Instanzen des MINIMUM BIN PACKING-Problems mit $m$ Objekten von höchstens $r$ verschiedenen Größen können exakt in einer Zeit gelöst werden, die polynomiell in $m^r$ ist.

**Beweis:** Seien $s_1, \ldots, s_r$ die vorkommenden Objektgrößen. Wir definieren ein *Teilproblem* als einen $r$-Vektor mit ganzzahligen Komponenten zwischen 0 und $m$ (inklusive). Das Teilproblem $(a_1, \ldots, a_r)$ bedeutet intuitiv, daß $a_1$ Objekte der Größe $s_1$, $a_2$ Objekte der Größe $s_2$ usw. in möglichst wenig Bins zu plazieren sind. Ein Teilproblem $(a_1, \ldots, a_m)$ heißt *Packungstyp*, wenn ein Bin ausreicht, also wenn $\sum_{i=1}^{r} a_i s_i \leq 1$. Es gibt genau $(m+1)^r$ Teilprobleme. Wir schreiben sie alle nieder und überprüfen für jedes Teilproblem, ob es ein Packungstyp ist—das ist leicht. Danach lösen wir das Problem wie im Beweis von Satz 11.3 mit Hilfe von dynamischem Programmieren. Wir benutzen ein Array $A$, dessen Spalten mit den Teilproblemen und dessen Reihen mit den Packungstypen markiert sind. Der Eintrag $A[i,j]$ soll dabei die minimale Anzahl Bins angeben, die man braucht, um das $j$te Teilproblem nur mit Hilfe der Packungstypen $1, \ldots, i$ zu lösen (d.h. Bins dürfen nur entsprechend diesen Packungstypen gefüllt werden). Wieder errechnet sich jeder Eintrag aus Einträgen in der vorherigen Reihe: Entweder man benutzt den $i$ten Packungstyp gar nicht, oder 1mal, 2mal, ..., $m$-mal, und das restliche Teilproblem muß optimal mit den Packungstypen $1, \ldots, i-1$ gelöst werden. Der Eintrag in der letzten Reihe in der Spalte, die mit dem ursprünglichen Problem markiert ist, gibt die nötige Anzahl Bins an, und wieder kann man die gesamte Lösung durch "Zurückverfolgen" rekonstruieren. Die benötigte Zeit ist offensichtlich polynomiell in der Anzahl der Teilprobleme, also in $m^r$. ∎

Als nächstes beschreiben wir einen Approximationsalgorithmus für den Fall, daß alle

Objekte relativ groß sind (sie dürfen jetzt aber alle verschieden sein).

**Lemma** 11.6: Gegeben sei eine Instanz des MINIMUM BIN PACKING-Problems mit $m$ Objekten, alle der Größe mindestens $\delta > 0$, und optimaler Binanzahl $c^*$. Dann können wir in Zeit polynomiell in $m^{1/\delta^2}$ eine zulässige Lösung finden, die höchstens $(1+\delta)c^* + 1$ Bins benutzt.

**Beweis:** O.B.d.A. ist $\delta < 1$. Sei $k = \lceil \delta^2 m \rceil$. Wir sortieren die $m$ Objekte nach fallender Größe in Gruppen zu je $k$ Objekten: die erste Gruppe enthält die $k$ größten Objekte, die zweite Gruppe die $k$ nächstgrößeren usw. Die letzte Gruppe kann weniger als $k$ Objekte umfassen. Aus der ursprünglichen Probleminstanz $x$ erhalten wir jetzt eine modifizierte Instanz $x'$, indem wir erstens die Objekte in der ersten Gruppe entfernen und zweitens die Größe jedes Objektes in einer anderen Gruppe auf die kleinste Größe in der vorherigen Gruppe erhöhen. $x'$ hat höchstens $\lfloor m/k \rfloor \leq 1/\delta^2$ verschiedene Objektgrößen (vielleicht gar keine, nämlich wenn $k = m$, aber das ist in Ordnung), und wir lösen $x'$ exakt mit dem Algorithmus aus Lemma 11.5, wofür wir gerade Zeit polynomiell in $m^{1/\delta^2}$ brauchen.

Wir können die Objekte von $x'$ injektiv auf die Objekte von $x$ abbilden, so daß kein Bild kleiner als sein Urbild ist. Also ist die optimale Binanzahl $Opt(x')$ von $x'$ nicht größer als die optimale Binanzahl $Opt(x)$ von $x$. Umgekehrt können wir auch die Objekte von $x$ außerhalb der ersten Gruppe injektiv auf die Objekte von $x'$ abbilden, so daß kein Bild kleiner als sein Urbild ist. Indem wir (ganz grob) $k$ Bins für die erste Gruppe benutzen, erlaubt uns das, eine Lösung für $x$ zu konstruieren, die $Opt(x') + k$ Bins benutzt. Aber $Opt(x) \geq \delta m$, und daher ist

$$Opt(x') + k \leq Opt(x) + \delta^2 m + 1 \leq Opt(x)(1+\delta) + 1. \quad \blacksquare$$

Wir wollen jetzt endlich das allgemeine Problem in Angriff nehmen und müssen dazu nur angeben, wie mit den kleinen Objekten (denjenigen der Größe $< \delta$) zu verfahren ist. Aber das ist einfach: Wir lösen zuerst das Problem für die großen Objekte mit dem Algorithmus aus Lemma 11.6 und durchlaufen dann die kleinen Objekte. Jedes kleine Objekt wird in einen Bin gesteckt, der noch genügend Platz hat; nur wenn kein solcher Bin existiert, wird ein neuer Bin angefangen.

Gegeben sei ein MINIMUM BIN PACKING-Problem mit $m$ Objekten und optimaler Binanzahl $c^*$ sowie eine vorgegebene rationale Zahl $\epsilon$ mit $0 < \epsilon \leq 1$. Wir setzen $\delta = \epsilon/2$ und berechnen eine zulässige Lösung mit Binanzahl $c$ wie oben beschrieben.

Fangen wir keine neuen Bins an, ist natürlich $c \leq c^*(1+\delta)+1 \leq c^*(1+\epsilon)+1$. Andernfalls ist die Summe der Objekte in jedem Bin, außer dem letzten, mindestens $1-\delta$, denn sonst hätten wir den letzten Bin nicht angefangen. Aber dann ist offensichtlich $c^* \geq (c-1)(1-\delta)$ und daher $c \leq \frac{c^*}{1-\delta}+1 \leq (1+\epsilon)c^*+1$. Damit haben wir bewiesen:

**Satz** 11.7: Gegeben sei eine Instanz des MINIMUM BIN PACKING-Problems mit $m$ Objekten und optimaler Binanzahl $c^*$ sowie eine rationale Zahl $\epsilon > 0$. Dann können wir in Zeit polynomiell in $m^{1/\epsilon^2}$ eine zulässige Lösung finden, die höchstens $(1+\epsilon)c^*+1$ Bins benutzt.

Ein *Approximationsschema* für ein Problem $P$ ist eine Menge, die für jedes rationale $\epsilon > 0$ einen Algorithmus $A_\epsilon$ enthält, der $P$ mit Approximationsgüte höchstens $1+\epsilon$ löst. Hat jeder Algorithmus $A_\epsilon$ polynomielle Laufzeit in der Eingabelänge, reden wir von einem *Polynomialzeit-Approximationsschema* oder *PTAS* (Polynomial-Time Approximation Scheme). Wir wissen aus Satz 11.4, daß das MAXIMUM-VALUE KNAPSACK-Problem ein PTAS besitzt. Wir wissen auch, daß dasselbe nicht für MINIMUM BIN PACKING gilt, aber Satz 11.7 verspricht etwas sehr Ähnliches (nur das +1 stört). Wir reden hier von einem APTAS (Asymptotic PTAS); auf die genaue Definition wollen wir nicht eingehen. Ein anderer wichtiger Unterschied zwischen den Sätzen 11.4 und 11.7 ist die Art, wie $\epsilon$ in die Laufzeit eingeht. Die Abhängigkeit von $1/\epsilon$ ist in Satz 11.4 polynomiell, in Satz 11.7 hingegen exponentiell. Das macht den Algorithmus aus Satz 11.4 wesentlich attraktiver, und in der Tat gilt KNAPSACK, obwohl NP-vollständig, als ein "leichtes" Problem. Formal reden wir von einem FPTAS (Fully Polynomial-Time Approximation Scheme), wenn die Laufzeit von $A_\epsilon$ polynomiell nicht nur in der Eingabelänge, sondern auch in $1/\epsilon$ ist. Können wir ein Problem nicht in Polynomialzeit lösen, ist ein FPTAS mehr oder weniger das Beste, worauf wir hoffen können.

Obwohl wir es nicht gezeigt haben, hat das MINIMUM BIN PACKING-Problem nicht nur ein asymptotisches PTAS, sondern auch ein asymptotisches FPTAS (N. Karmarkar and R. M. Karp, An efficient approximation scheme for the one-dimensional bin-packing problem, 23rd Annual Symp. on Foundations of Computer Science (1982), pp. 312–320).

# 12 Grenzen der Approximierbarkeit

In diesem Kapitel werden wir voraussetzen, daß P $\neq$ NP, um ständige Wiederholungen von ", es sei denn, P = NP" zu vermeiden. Außerdem betrachten wir nur Algorithmen, die in Polynomialzeit laufen.

Wir wissen bereits, daß kein Approximationsalgorithmus für MINIMUM BIN PACKING mit Approximationsgüte $< 3/2$ existieren kann. Dennoch kann man, wie wir gesehen haben, das Problem recht gut lösen. Es gibt aber auch wichtige Optimierungsprobleme, bei denen man mit Approximation fast nichts ausrichten kann. Das ist das Thema dieses Kapitels.

Ein Problem, für das wir leicht zeigen können, daß kein vernünftiger Approximationsalgorithmus existiert, ist das Traveling-Salesman-Problem.

MINIMUM TSP:

Eingabe: Ein vollständiger ungerichteter Graph $G$, in dem jede Kante mit einer *Länge* aus $\mathbb{N}$ beschriftet ist.

Ausgabe: Eine Tour in $G$ mit minimaler Gesamtlänge.

**Satz** 12.1: Falls P $\neq$ NP, gibt es keinen Approximationsalgorithmus für MINIMUM TSP, der in Polynomialzeit läuft und konstante Approximationsgüte hat.

**Beweis**: Nehmen wir an, wir hätten einen solchen Approximationsalgorithmus $A$ mit konstanter Approximationsgüte $k$. Wir zeigen, daß wir mit Hilfe von $A$ das NP-vollständige UNDIRECTED HAMILTON CYCLE-Problem (Satz 6.4) in Polynomialzeit lösen können. Sei also $G = (V, E)$ eine Instanz des UNDIRECTED HAMILTON CYCLE-Problems und sei $m = |V|$. Wir konstruieren eine Instanz $H$ des MINIMUM TSP-Problems wie folgt: Der Graph ist der vollständige Graph auf der Knotenmenge $V$, und eine Kante hat Länge 1, wenn sie in $E$ vorkommt, und $km + 1$ sonst. Dann benutzen wir $H$ als Eingabe für $A$.

Besitzt $G$ einen Hamilton-Kreis, hat $H$ eine Tour mit Gesamtlänge $m$, und $A$ muß eine Tour mit Gesamtlänge höchstens $km$ produzieren. Besitzt $G$ keinen Hamilton-Kreis,

hat jede Tour Gesamtlänge mindestens $km + 1$. Die Ausgabe von $A$ sagt uns also, ob es in $G$ einen Hamilton-Kreis gibt. ∎

Der Beweis schließt noch viel schlechtere Approximationen aus: Bei $m$ Knoten kann es nicht einmal einen Approximationsalgorithmus mit Güte $2^m$ geben. Das allgemeine Traveling-Salesman-Problem läßt sich also so gut wie gar nicht approximieren. Anders steht es mit speziellen Varianten. Zum Beispiel kennen Sie vielleicht einen Approximationsalgorithmus mit Approximationsgüte höchstens 3/2 für den Fall, daß die Kantengewichte die Dreiecksungleichung erfüllen.

MAXIMUM CLIQUE:

Eingabe: Ein ungerichteter Graph $G$.

Ausgabe: Eine Clique maximaler Größe in $G$.

Als nächstes werden wir zeigen, daß das MAXIMUM CLIQUE-Problem entweder sehr gut oder sehr schlecht approximierbar ist. Dazu brauchen wir eine Funktion $\psi$, die einen ungerichteten Graphen $G = (V, E)$ auf den ungerichteten Graphen $\psi(G) = (V \times V, E')$ abbildet ($\psi(G)$ wird auch manchmal mit $G \times G$ bezeichnet), wobei

$$\{(u,v),(u',v')\} \in E' \iff (u = u' \text{ und } \{v,v'\} \in E) \text{ oder } \{u,u'\} \in E.$$

Fig. 12.1 zeigt ein Beispiel dieser Konstruktion. Die für uns wichtige Eigenschaft von $\psi$ ist:

**Lemma** 12.2: Für jeden ungerichteten Graphen $G$ und jede positive reelle Zahl $k$ gilt: $G$ hat eine Clique der Größe mindestens $k$ genau dann, wenn $\psi(G)$ eine Clique der Größe mindestens $k^2$ hat.

**Beweis**: Sei $C$ eine Clique in $G = (V, E)$ der Größe $k$. Dann ist $\{(u,v) \in V \times V \mid u,v \in C\}$ eine Clique in $\psi(G)$ der Größe $k^2$.

Sei umgekehrt $D$ eine Clique in $\psi(G) = (V \times V, E')$ der Größe $k^2$. Dann ist $C = \{u \in V \mid \exists v \in V : (u,v) \in D\}$ eine Clique in $G$, denn sind $u$ und $u'$ verschiedene Knoten in $C$, gibt es $v, v' \in V$ mit $(u,v),(u',v') \in D$. Da $\{(u,v),(u',v')\} \in E'$, ist dann $\{u,u'\} \in E$. Ist $|C| \geq k$, sind wir fertig. Ist $|C| < k$, wählen wir $u_0 \in V$ mit $|C_{u_0}| \geq k$, wobei $C_{u_0} = \{v \in V \mid (u_0,v) \in D\}$ (möglich, da $|D| \geq k^2$) und beobachten, daß $C_{u_0}$ eine Clique in $G$ ist. ∎

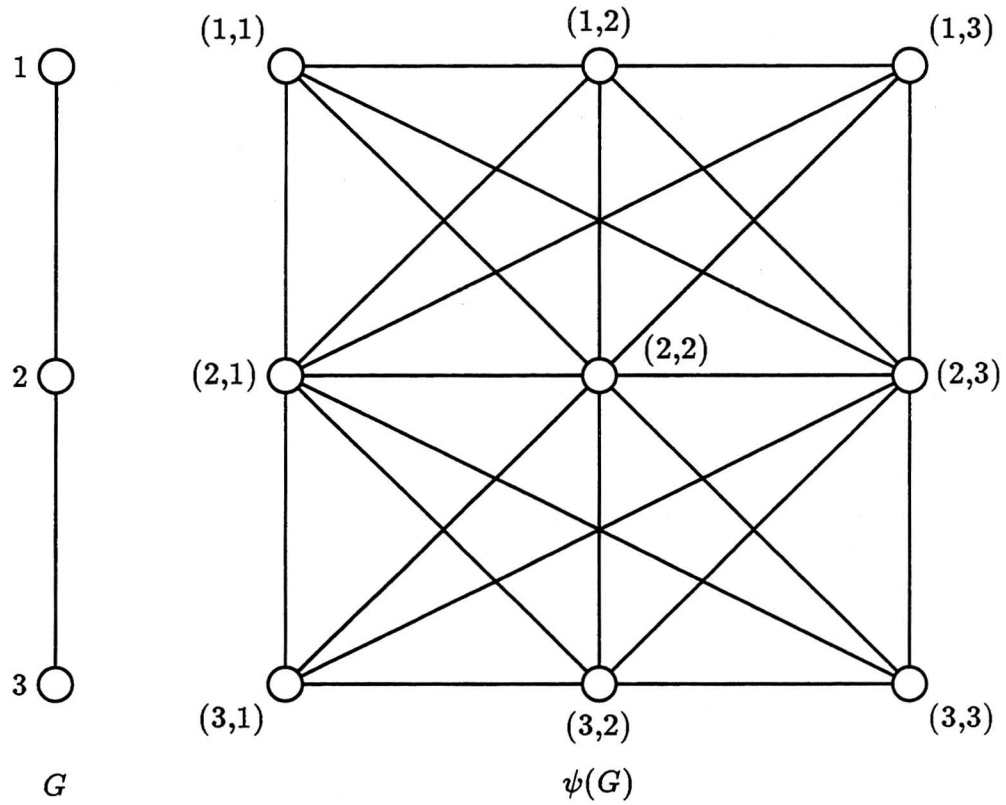



Fig. 12.1. Die Konstruktion von $\psi(G)$.

**Lemma** 12.3: Gibt es einen Approximationsalgorithmus für MAXIMUM CLIQUE mit polynomieller Laufzeit und konstanter Approximationsgüte, dann gibt es auch ein PTAS für MAXIMUM CLIQUE.

**Beweis:** Nehmen wir an, wir hätten einen solchen Algorithmus $A$ mit Approximationsgüte $1+r$ für irgendein $r > 0$. Wir betrachten den Algorithmus $A'$, der wie folgt vorgeht: Bei Eingabe $G$ erstellt $A'$ zuerst $\psi(G)$, dann wird $A$ dazu benutzt, eine große Clique $D$ in $\psi(G)$ zu finden, und schließlich gibt $A'$ die Clique $C$ in $G$ aus, die sich mit dem Argument aus dem Beweis von Lemma 12.2 aus $D$ herleiten läßt.

Sei $k$ die Größe einer größten Clique in $G$. Dann hat $\psi(G)$ eine Clique der Größe $k^2$, und der Aufruf von $A$ muß eine Clique $D$ der Größe mindestens $k^2/(1+r)$ liefern. Dann ist $|C| \geq \sqrt{|D|} \geq k/\sqrt{1+r}$, d.h. die Approximationsgüte von $A'$ ist höchstens $\sqrt{1+r} \leq 1 + r/2$.

Aus jedem Polynomialzeit-Algorithmus mit Güte $1+r$ können wir also einen Algorithmus herleiten, der immer noch in Polynomialzeit läuft, dessen Güte sich aber auf $\leq 1+r/2$ verbessert hat. Wiederholen wir diese Konstruktion hinreichend oft, können wir für jedes feste $\epsilon > 0$ die Güte unterhalb von $1+\epsilon$ treiben, ohne die polynomielle Laufzeit zu

verlieren. Das gibt uns ein PTAS für MAXIMUM CLIQUE. ∎

Zuletzt soll kurz eine Entwicklung erwähnt werden, die in den letzten Jahren das Gebiet der Approximationsalgorithmen zu einem der "heißesten" der theoretischen Informatik gemacht hat. Es wurden ganz überraschende Zusammenhänge zwischen sogenannten *interaktiven Beweissystemen* und Approximationsalgorithmen entdeckt, die beide Gebiete revolutioniert haben.

Wir wissen, daß eine Sprache $L$ genau dann zu NP gehört, wenn es für jedes $x \in L$ einen "Beweis" der Tatsache $x \in L$ gibt, der in Polynomialzeit überprüft werden kann. Wir betrachten jetzt eine allgemeinere "Beweissituation", die durch zwei Funktionen $r, q : \mathbb{N}_0 \to \mathbb{N}$ charakterisiert ist. Der Verifizierer, der entscheiden muß, ob ein Eingabewort $x$ mit $|x| = n$ zu $L$ gehört, bekommt $x$ sowie $O(r(n))$ Zufallsbits. Außerdem gibt es einen (angeblichen) Beweis $B$ (einen binär kodierten String) für die Tatsache $x \in L$, auf den der Verifizierer aber nicht direkt zugreifen kann. Statt dessen berechnet der Verifizierer zuerst $O(q(n))$ positive ganze Zahlen (vermutlich unter Benutzung der Zufallsbits) und erfährt danach die $O(q(n))$ Bits, die an den entsprechenden Positionen in $B$ vorkommen (die Zahlen fungieren also als Adressen). Die Komplexitätsklasse $\mathrm{PCP}(r(n), q(n))$ (PCP steht für "Probabilistically Checkable Proofs") ist wie folgt definiert: $L \in \mathrm{PCP}(r(n), q(n))$, wenn es einen Verifizierer $V$ (einen Algorithmus) gibt, der in Polynomialzeit arbeitet und für jedes Eingabewort $x$ mit $|x| = n$ $O(r(n))$ Zufallsbits benutzt, $O(q(n))$ Beweisbits inspiziert und die folgenden Eigenschaften hat:

(1) Ist $x \in L$, gibt es einen String $B$, so daß $V$ mit Wahrscheinlichkeit 1 akzeptiert (es gibt einen Beweis $B$, der $V$ überzeugt);

(2) Ist $x \notin L$, akzeptiert $V$ für jeden String $B$ mit Wahrscheinlichkeit höchstens $1/2$ (kein falscher Beweis kann $V$ "fest" überzeugen).

Wir wissen bereits, daß $\mathrm{NP} = \bigcup_{k=1}^{\infty} \mathrm{PCP}(0, n^k)$, denn das ist nur eine Umformulierung dessen, was oben steht. Die erstaunliche Tatsache ist aber:

**Satz** 12.4: $\mathrm{NP} = \mathrm{PCP}(\log n, 1)$.

Der Verifizierer ist also fast gänzlich auf sich gestellt. Er bekommt nur $O(\log n)$ Zufallsbits und darf konstant viele Bits in einem Beweis inspizieren, wobei er sich aber von keinem falschen Beweis täuschen lassen darf (es genügt also nicht, daß der Beweis "Ja" oder "Nein" behauptet; das kennen wir auch von der alten Charakterisierung von NP).

Und dieses wenige genügt, sagt der Satz, um die Klasse der akzeptierten Sprachen von P auf NP anzuheben.

Wir wollen hier nicht auf den umfangreichen Beweis von Satz 12.4 eingehen. Interessierte Leser seien verwiesen auf S. Arora, C. Lund, R. Motwani, M. Sudan and M. Szegedy, Proof verification and hardness of approximation problems, 33rd Annual Symp. on Foundations of Computer Science (1992), pp. 14–23.

Mit Hilfe von Satz 12.4 können wir entscheiden, ob das MAXIMUM CLIQUE-Problem sich sehr gut oder sehr schlecht approximieren läßt.

**Satz** 12.5: Falls P $\neq$ NP, gibt es keinen Approximationsalgorithmus für MAXIMUM CLIQUE, der in Polynomialzeit läuft und konstante Approximationsgüte hat.

**Beweis:** Nach Lemma 12.3 genügt es zu zeigen, daß die Existenz eines Approximationsalgorithmus $A$ mit einer Approximationsgüte unterhalb von 2 zu einem Widerspruch führt.

Sei $L$ eine NP-vollständige Sprache und sei $x$ eine Eingabe mit $|x| = n$. Wir zeigen, wie unter Benutzung von $A$ in Polynomialzeit entschieden werden kann, ob $x \in L$.

Wir betrachten das in der Aussage $L \in \mathrm{PCP}(\log n, 1)$ implizite Beweissystem. Wir können annehmen, daß der Verifizierer $V$ genau $r(n) = O(\log n)$ Zufallsbits bekommt und genau $q$ Beweisbits inspiziert. Wir definieren jetzt einen Graphen $G_x$, dessen Knoten alle die Eingabe $x$ akzeptierenden Berechnungen von $V$ der Form $(\alpha, b_1, \ldots, b_q)$ sind, wobei $\alpha$ eine Bitfolge der Länge $r(n)$ und $b_1, \ldots, b_q$ Bits sind. Wir identifizieren hierbei ein Tupel $(\alpha, b_1, \ldots, b_q)$ mit einer Berechnung von $V$, in der $V$ die Zufallsbits $\alpha$ und die Beweisbits $b_1, \ldots, b_q$ erfährt. Zwei Knoten $(\alpha, b_1, \ldots, b_q)$ und $(\alpha', b'_1, \ldots, b'_q)$ sind genau dann in $G_x$ durch eine Kante verbunden, wenn sie miteinander *konsistent* sind, d.h.: Wird dieselbe Bitposition im Beweis von den beiden Berechnungen abgefragt (das hängt von $\alpha$ und $\alpha'$ ab), müssen die beiden Ergebnisse (das könnten $b_i$ und $b'_j$ sein) identisch sein. Der Graph $G_x$ hat polynomielle Größe und kann in polynomieller Zeit aufgebaut werden.

Sei $\omega(G_x)$ die *Cliquenzahl* von $G_x$, also die Größe einer größten Clique in $G_x$. Wir wollen zeigen, daß

$$x \in L \Rightarrow \omega(G_x) = 2^{r(n)},$$
$$x \notin L \Rightarrow \omega(G_x) \leq 2^{r(n)}/2,$$

womit der Satz bewiesen sein wird.

Für jeden Beweis $B$ sei $P(B)$ die Wahrscheinlichkeit, daß $V$ mit dem Beweis $B$ die Eingabe $x$ akzeptiert (das Verhalten von $V$ hängt von den Zufallsbits ab). Für einen

festen Beweis $B$ sind zumindest alle diejenigen Knoten in $G_x$ paarweise adjazent, die Berechnungen beschreiben, die mit $B$ konsistent sind (die abgefragten Bits haben in $B$ tatsächlich die Werte $b_1, \ldots, b_q$). Also ist $\omega(G_x)$ mindestens so groß wie die Anzahl der mit $B$ konsistenten Berechnungen von $V$. Diese Anzahl ist aber genau $2^{r(n)}P(B)$.

Ist umgekehrt $C$ eine Clique in $G_x$, müssen alle Berechnungen in $C$, die das gleiche Beweisbit abfragen, die gleiche Antwort erhalten. Daraus folgt, daß es einen Beweis $B$ geben muß, der mit *allen* Berechnungen in $C$ konsistent ist. Insbesondere gibt es einen Beweis $B_0$, der mit allen Berechnungen in einer maximalen Clique konsistent ist. Daraus folgt, daß $\omega(G_x)$ nicht größer ist als die Anzahl der mit $B_0$ konsistenten Berechnungen, die wiederum $2^{r(n)}P(B_0)$ ist.

Zusammenfassend haben wir gezeigt, daß

$$\omega(G_x) = 2^{r(n)} \max_B P(B).$$

Nach der Definition des Beweissystems ist aber $\max_B P(B) = 1$, falls $x \in L$, während $\max_B P(B) \leq 1/2$, falls $x \notin L$. ∎

Man kann sogar zeigen, daß es eine Konstante $\epsilon > 0$ gibt, so daß die Cliquenzahl in einem Graphen mit $m$ Knoten nicht einmal mit Approximationsgüte $m^\epsilon$ berechnet werden kann. Die beste obere Schranke ist eine Approximationsgüte von $O(m/(\log m)^2)$ (R. Boppana and M. M. Halldórsson, Approximating maximum independent sets by excluding subgraphs, BIT **32** (1992), pp. 180–196). Das ist beklagenswert schlecht, wie man leicht sieht. Das Clique-Problem gehört, theoretisch wie praktisch, zu den sehr schwierigen NP-vollständigen Problemen.

# 13 Parallele Algorithmen

Bisher haben wir ausschließlich Berechnungen betrachtet, die von *einem* Prozessor ausgeführt werden. Solche Berechnungen heißen *sequentiell.* Dagegen werden *parallele* Berechnungen von mehreren Prozessoren ausgeführt, die an der Lösung eines gemeinsamen Problems kooperieren. Parallele Berechnungen und parallele Algorithmen gewinnen mit der zunehmenden Verbreitung von Parallelrechnern immer mehr an Bedeutung, und es ist somit auch Aufgabe der Komplexitätstheorie, diese zu studieren und möglichst allgemeine Aussagen über sie zu formulieren. Wir fangen damit an, uns ein wenig mit den Möglichkeiten der Parallelverarbeitung vertraut zu machen.

Wir betrachten zuerst ein einfaches Beispiel, ohne uns um Einzelheiten zu kümmern. Das Problem ist das folgende: Jede von $n = 10^6$ Personen kennt eine ganze Zahl. Wir möchten die Summe dieser Zahlen berechnen. Dabei nehmen wir an, daß die Personen weit auseinander wohnen, daß aber jede ein Telefon hat. Die Telefonnummern der $n$ Personen sind gerade die Zahlen $1, \ldots, n$. Im folgenden sei "Person $i$" die Person mit der Telefonnummer $i$, für $i = 1, \ldots, n$.
Eine sequentielle Lösung des Problems wäre z.B., wenn Person 1 das folgende ausführt:

$S :=$ eigene Zahl;
**for** $i := 2$ **to** $n$ **do**
  rufe Person $i$ an, erfrage deren Zahl und addiere sie zu $S$;

Eine parallele Lösung hingegen könnte so vorgehen: Plaziere die $n$ Personen gedanklich an den Blättern eines vollständigen binären Baums und lasse Person $i$ sich von Blatt $i$ in Richtung Wurzel bewegen (s. Fig. 13.1 und 13.2).

Nähert sich eine Person einem inneren Knoten von links, ruft sie die Person an, die sich dem Knoten von rechts nähert, teilt ihr die eigene Zahl mit und hält. Nähert sich eine Person einem inneren Knoten von rechts, wartet sie auf den Anruf der von links kommenden Person, addiert die mitgeteilte Zahl zur eigenen Zahl und geht weiter Richtung Wurzel (es sei denn, sie ist bereits an der Wurzel; dann ist ihre Zahl das Endergebnis).

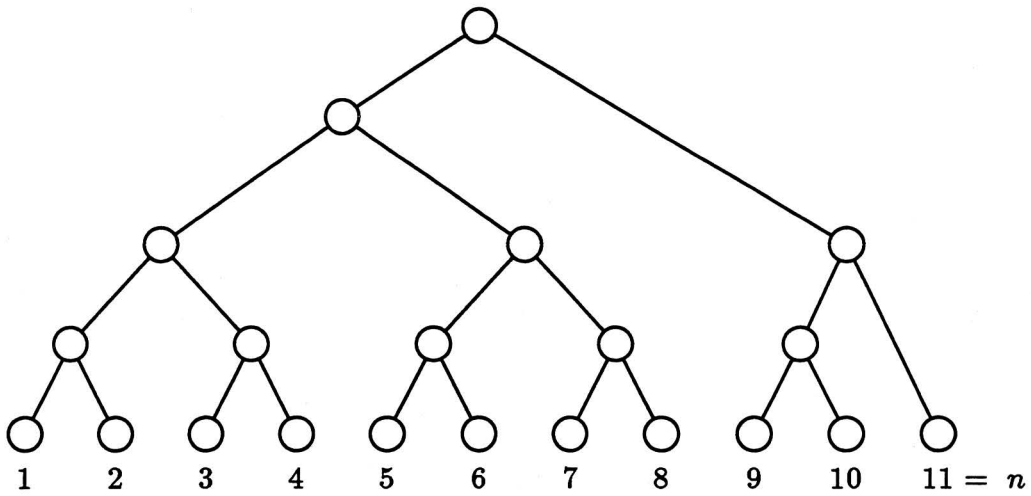



Fig. 13.1. Ein binärer Baum über $n$ Personen.

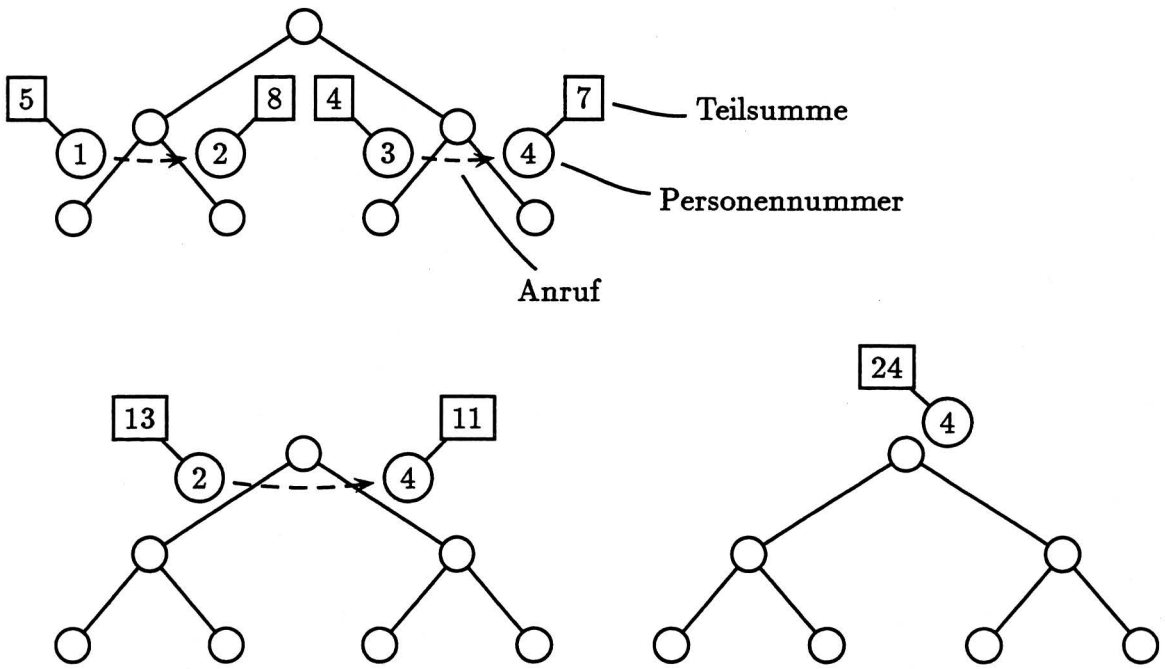



Fig. 13.2. Der zeitliche Ablauf des Telefonsummierens.

Die Höhe des Baums ist $\lceil \log_2 n \rceil$. Alle Anrufe bei Knoten der gleichen Tiefe können gleichzeitig erfolgen. Nehmen wir an, ein Anruf dauert 1 Minute, wobei die Zeit für Addition und Verwaltung (wen muß ich anrufen?) bereits eingerechnet wurde.

Laufzeit des sequentiellen Algorithmus: $(10^6 - 1)$ Min. $\approx$ 2 Jahre
Laufzeit des parallelen Algorithmus: $\lceil \log_2 10^6 \rceil$ Min. = 20 Min.

Beobachtungen:

- Der parallele Algorithmus ist wesentlich schneller;
- aber auch um einiges komplizierter;
- Der binäre Baum ist eine nützliche Berechnungsstruktur;
- Die fortlaufende Numerierung der Personen ist wichtig, damit jede herausfinden kann, was sie zu tun hat.

Wir suchen jetzt ein Berechnungsmodell, das uns erlaubt, den obigen Algorithmus sinngemäß auszudrücken. Die TM ist dafür nicht geeignet (man könnte eine von der Eingabe abhängige Anzahl von Lese/Schreibköpfen vereinbaren, aber diese Köpfe würden immer noch zu lange brauchen, um sich über die Bänder zu bewegen). Für den Entwurf von parallelen Algorithmen ist die *PRAM (Parallel Random Access Machine)* das beliebteste Modell.

Eine PRAM besteht aus unendlich vielen Prozessoren $P_1, P_2, \ldots$ und einem globalen Speicher (Fig. 13.3).

Der globale Speicher enthält unendlich viele Speicherzellen $M_1, M_2, \ldots$, von denen jede eine beliebige ganze Zahl enthalten kann. Jeder Prozessor $P_i$ ist eine RAM mit endlich vielen Registern, die ebenfalls beliebige ganze Zahlen enthalten können, und den üblichen Instruktionen wie ADD, SUB, MULT, DIV (ganzzahlige Division), JUMP, JZERO (jump if zero), JPOS (jump if positive), HALT usw., die alle auf den Registern arbeiten. Eine zusätzliche Instruktion LOADINDEX lädt den eigenen Index $i$ in ein Register. Außerdem gibt es zwei Instruktionen, die auf dem globalen Speicher arbeiten: LOAD lädt in ein Register den Inhalt von $M_j$, wobei $j$ der Inhalt eines anderen Registers ist, und STORE überschreibt den Inhalt von $M_j$ mit dem Inhalt eines Registers, wobei $j$ der Inhalt eines anderen Registers ist. Mehrere Prozessoren können gleichzeitig versuchen, ein und dieselbe globale Speicherzelle zu beschreiben (*Schreibkonflikt)*; dann gewinnt immer derjenige beteiligte Prozessor, der den kleinsten Index $i$ hat (PRIORITY-*Regel*). Wir nehmen an, daß jede Instruktion in konstanter Zeit ausgeführt werden kann. Alle Prozessoren haben das gleiche Programm und arbeiten synchron (sie haben einen gemeinsamen "Taktgeber"). Eine Eingabe der Länge $n$ besteht aus $n$ ganzen Zahlen und steht anfangs in $M_1, \ldots, M_n$.

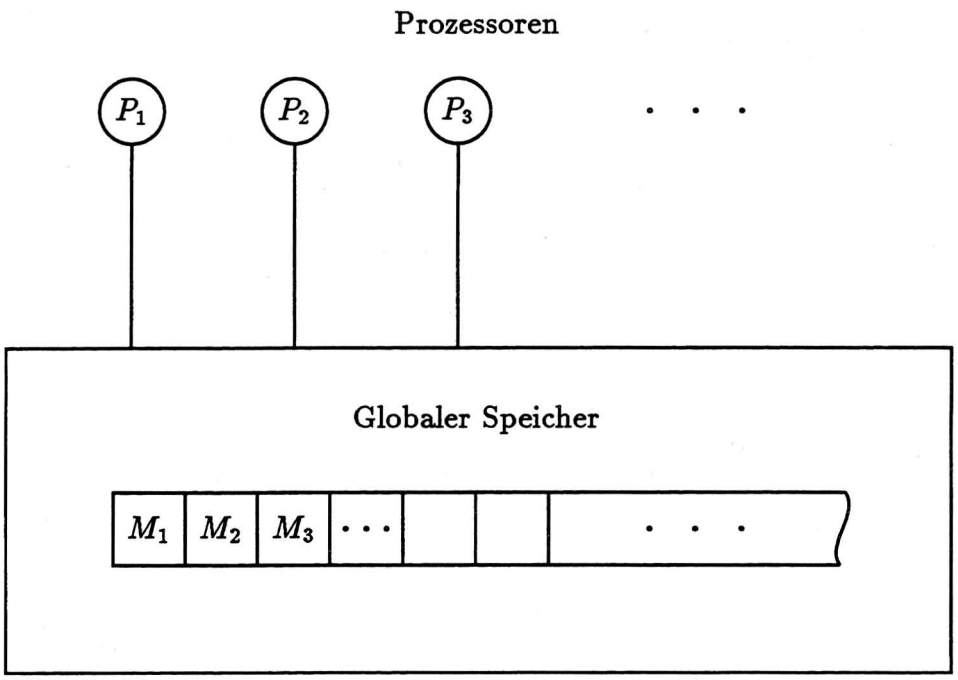



Fig. 13.3. Die PRAM.

Alle anderen globalen Speicherzellen und alle Register aller Prozessoren sind mit 0 vorbesetzt. Die Ausgabe soll am Ende der Berechnung in ähnlicher Weise in den ersten globalen Speicherzellen stehen.

Die Prozessoren einer PRAM führen das gleiche Programm aus, aber das heißt nicht, daß sie auch in jedem Schritt genau das gleiche tun (dann wäre es sinnlos, mehr als einen Prozessor einzusetzen): Die Instruktion LOADINDEX erlaubt es den Prozessoren, verschiedene Berechnungen auszuführen. Die PRAM enthält keine expliziten Mechanismen ("Telefone") zur Kommunikation zwischen den Prozessoren. Das ist aber auch nicht nötig, denn ein Prozessor kann sich einem anderen Prozessor mitteilen, indem er einfach eine Nachricht für ihn in einer globalen Speicherzelle hinterlegt. Wir haben die PRAM mit unendlich vielen Prozessoren ausgestattet. Das ist natürlich insofern unrealistisch, als jeder reale Parallelrechner eine feste Anzahl von Prozessoren hat. Es hat aber auch kein sequentieller Rechner unendlich viel Speicherplatz, und trotzdem ist es bequem, die TM mit unendlich langen Arbeitsbändern auszustatten. So, wie wir den tatsächlichen Verbrauch an Speicherplatz bei TMs angeben, werden wir die Anzahl der von einer parallelen Berechnung tatsächlich benutzten Prozessoren angeben und versuchen, diese Zahl möglichst gering zu

halten. Wir definieren diese Anzahl, bei einer Eingabe $x$, als

$$\max(\{1\} \cup \{i \in \mathbb{N} \mid P_i \text{ führt bei Eingabe } x \text{ einen STORE-Befehl aus}\}).$$

Diese Definition ist dadurch motiviert, daß ein Prozessor, der nie in den globalen Speicher schreibt, die Ausgabe weder direkt noch indirekt beeinflußt und somit entbehrlich ist. Wenn ein PRAM-Algorithmus $p$ Prozessoren benutzt, sagen wir auch, daß der Algorithmus auf einer PRAM "mit $p$ Prozessoren" ausgeführt werden kann. Eine PRAM mit nur einem Prozessor können wir mit einer gewöhnlichen RAM identifizieren.

Hier ist nun ein PRAM-Algorithmus zur Addition von $n$ Zahlen, die anfangs in einem Array $A[1..n]$ im globalen Speicher stehen. Wir nehmen der Einfachheit halber an, daß $n$ eine Zweierpotenz ist, $n = 2^k$.

**for** $t := 1$ **to** $k$ **do**
  **for** $i \in \{1, \ldots, n\}$ **pardo** (* tue parallel *)
    **if** $i/2^t \in \mathbb{N}$ (* nur dann hat $P_i$ noch etwas zu tun *)
    **then** $A[i] := A[i] + A[i - 2^{t-1}]$;

Nach der Berechnung steht das Ergebnis in $A[n]$.

Die obige Beschreibung benutzt eine Menge Konventionen, die zur Klarheit beitragen, aber die Übersetzung in PRAM-Code weniger direkt machen. Eine erste Konvention ist, daß wir überhaupt eine höhere Programmiersprache benutzen; diese muß natürlich nach Maschinensprache kompiliert werden können, aber so haben wir es bereits bei TMs gemacht. Das ungewohnte Schlüsselwort **pardo** wird dabei mit Hilfe von LOADINDEX übersetzt: Die $n$ Werte für $i$ werden mit Hilfe der Prozessorindizes auf $n$ verschiedene Prozessoren verteilt, von denen jeder das folgende Statement mit dem ihm zugeteilten Wert von $i$ ausführt. Das Array $A$ kann durch einen zusammenhängenden Speicherblock repräsentiert werden. Die Zahl $n$ wird nicht explizit in der Eingabe erwähnt, aber wir nehmen an, daß wir zusammen mit einem Array auch dessen Größe erfahren (die wirkliche Eingabekonvention ist also vielleicht $M_1 = n$ und $M_{i+1} = A[i]$, für $i = 1, \ldots, n$). Die Zahl $k = \log_2 n$ ist gar nicht Teil der Eingabe. Es wird stillschweigend vorausgesetzt, daß jeder Prozessor zuerst $k$ aus $n$ berechnet, was sicherlich in $O(\log n)$ Schritten möglich ist. Die Größe $2^{t-1}$ kann mit unserem Instruktionssatz nicht in konstanter Zeit aus dem Nichts berechnet werden, aber wohl aus dem entsprechenden Wert im vorherigen Schleifendurchlauf; der Test, ob $i/2^t \in \mathbb{N}$, ist damit auch leicht. Schließlich kann das Ergebnis auf triviale Art von $A[n]$ nach $M_1$ bewegt werden, wo es laut Vereinbarung stehen soll. Mit diesen Erklärungen sollte es jetzt offensichtlich sein, daß das Programm, genau wie das

"Telefonsummieren", dem es nachempfunden ist, in $O(\log n)$ Zeit ausgeführt werden kann, wobei $n$ Prozessoren benutzt werden.

**Satz** 13.1: Die Summe von $n$ ganzen Zahlen kann in $O(\log n)$ Zeit auf einer PRAM mit $n$ Prozessoren berechnet werden.

Ein paralleler Algorithmus kann sequentiell simuliert werden, wobei die Zeit natürlich entsprechend größer wird.

**Satz** 13.2: Seien $p, t : \mathbb{N}_0 \to \mathbb{N}$ Funktionen, so daß eine RAM in $O(p(n)t(n))$ Zeit $p(n)$ aus $n$ berechnen kann. Dann kann ein Problem, das bei Eingabegröße $n$ auf einer PRAM mit $p(n)$ Prozessoren in $t(n)$ Zeit gelöst werden kann, auch auf einer RAM in $O(p(n)t(n))$ Zeit gelöst werden.

**Beweis:** Die RAM berechnet zuerst $p(n)$ und simuliert danach den ersten Schritt der $p(n)$ Prozessoren der PRAM, den zweiten Schritt der PRAM-Prozessoren, usw. Jeder Schritt der PRAM kann in $O(p(n))$ Zeit simuliert werden, so daß die gesamte Simulation in $O(p(n)t(n))$ Zeit erfolgen kann. Aufpassen muß man lediglich, damit bei Schreibkonflikten tatsächlich die PRIORITY-Regel Anwendung findet. Das erreicht man am einfachsten, indem die Prozessoren in der Reihenfolge fallender Prozessorindizes simuliert werden (dann überschreibt ein Prozessor mit kleinerer Nummer gegebenenfalls den von einem höhernumerierten Prozessor geschriebenen Wert). ∎

Wenn $p(n)$ hinreichend leicht zu berechnen ist (das ist in der Praxis immer der Fall), kann das Prozessor-Zeit-Produkt $p(n)t(n)$ eines PRAM-Algorithmus die sequentielle RAM-Komplexität $T(n)$ des betrachteten Problems also höchstens um einen konstanten Faktor (der den Simulationsaufwand ausdrückt) unterschreiten. Wenn $p(n)t(n) = O(T(n))$, hat der PRAM-Algorithmus *optimalen Speedup*: Mit $p(n)$ Prozessoren geht es tatsächlich $\Theta(p(n))$-mal schneller—das Beste, worauf wir hoffen können.

Der Algorithmus aus Satz 13.1 hat nicht optimalen Speedup, denn das Prozessor-Zeit-Produkt ist nur durch $O(n \log n)$ beschränkt, während die RAM-Komplexität offensichtlich $O(n)$ ist. Das ist ein Grund, mit dem Algorithmus unzufrieden zu sein. Wir können aber leicht einen neuen Algorithmus mit optimalem Speedup entwerfen. Zwar läßt sich die Zeit nicht auf $O(1)$ reduzieren, aber wir können die gleiche Zeit (bis auf einen konstanten Faktor) mit nur $O(n/\log n)$ Prozessoren erreichen: Die $n$ Eingabezahlen werden dazu etwa gleichmäßig auf $O(n/\log n)$ Prozessoren verteilt, jeder Prozessor bildet zuerst in $O(\log n)$

Zeit die Summe der Zahlen in seiner Gruppe, und danach führen die Prozessoren den alten Algorithmus aus—wir haben jetzt nicht mehr Zahlen als Prozessoren.

**Satz** 13.3: Die Summe von $n$ ganzen Zahlen kann in $O(\log n)$ Zeit auf einer PRAM mit $O(n/\log n)$ Prozessoren berechnet werden.

Wir betrachten jetzt ein interessanteres Problem, das *List Ranking*-Problem. Die Eingabe für dieses Problem ist eine "verknäulte" lineare Liste, und es geht darum, jedes Element in der Liste mit seiner Position innerhalb der Liste (von hinten gerechnet) zu markieren. Fig. 13.4 zeigt eine Beispieleingabe, die dadurch dargestellte Liste sowie die geforderte Ausgabe.

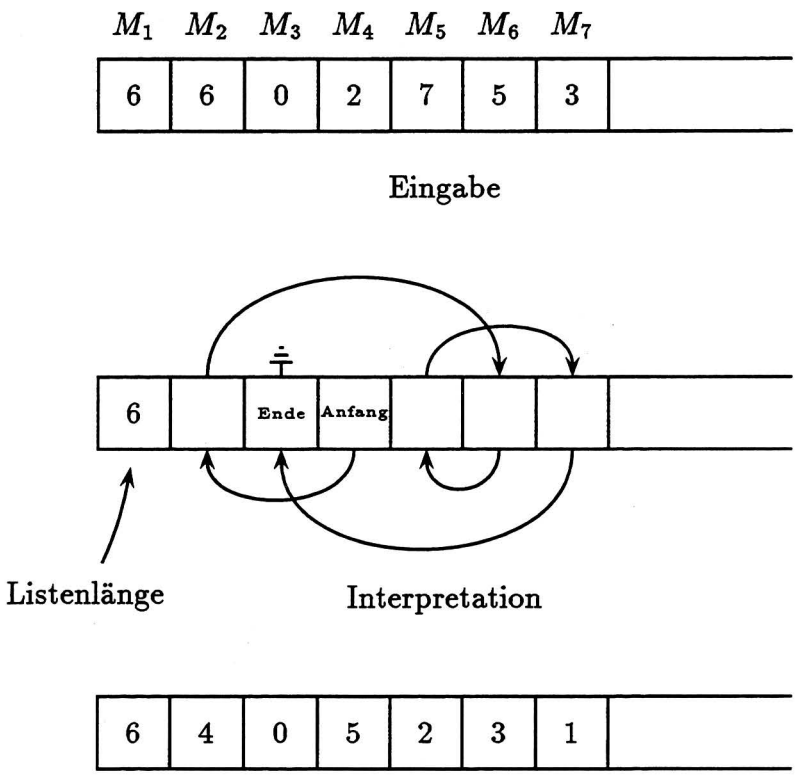



Fig. 13.4. Das List Ranking-Problem.

List Ranking ist ein fundamentales Problem, das beim Entwurf von parallelen Algorithmen immer wieder auftaucht. Wir werden später in diesem Kapitel eine Anwendung

von List Ranking sehen. Instanzen des List Ranking-Problems der Größe $n$ (d.h. die Liste hat $n$ Elemente) können natürlich sequentiell in $O(n)$ Zeit gelöst werden. Aber wie können wir mit mehr Prozessoren schneller sein?

Zuerst generalisieren wir das Problem. Gegeben sei jetzt eine Liste mit $n$ Elementen, bei der jede Kante (zwischen zwei Nachbarelementen) mit einer ganzen Zahl beschriftet ist. Gesucht ist für jedes Listenelement $u$ die Summe der Kantenbeschriftungen von $u$ bis zum Ende der Liste. Das ursprüngliche Problem erhalten wir als Spezialfall, wenn wir jede Kante mit 1 beschriften. Es genügt also, das allgemeinere Problem zu lösen. Das tun wir unter Benutzung der sogenannten *Pointer Jumping*-Methode, die in Fig. 13.5 dargestellt ist.

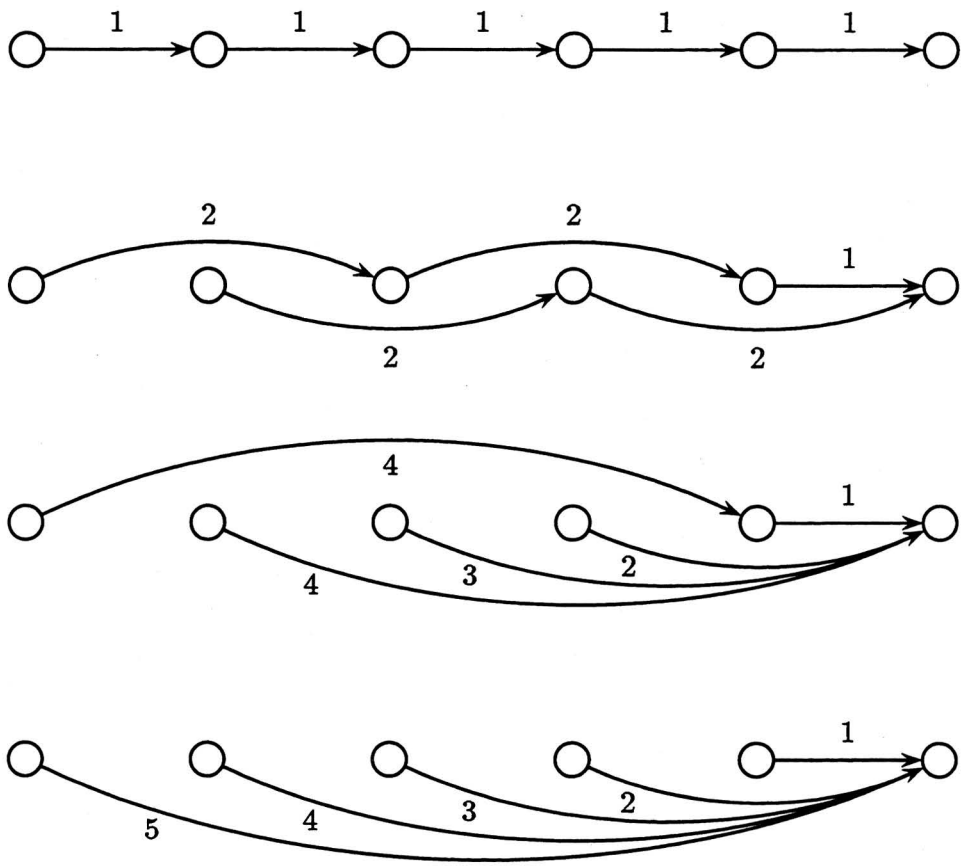



Fig. 13.5. Pointer Jumping.

Dabei ist es nützlich, sich vorzustellen, daß jedes Listenelement ein Prozessor ist. Wiederholt passiert das folgende: Jedes Listenelement $u$ mit Abstand $\geq 2$ vom Listenende

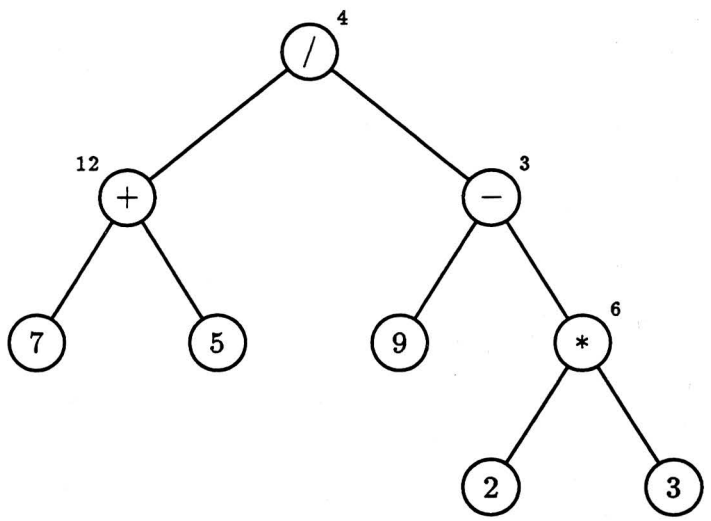
bestimmt seinen Nachfolger $v$ sowie die Beschriftung $a$ der Kante $(u, v)$. Weiter bestimmt $u$ den Nachfolger $w$ von $v$ sowie die Beschriftung $b$ der Kante $(v, w)$. Schließlich versetzt $u$ seinen "Zeiger", so daß sein Nachfolger nicht mehr $v$, sondern $w$ ist; die neue Kante $(u, w)$ bekommt die Beschriftung $a + b$. Man kann beobachten, daß die so entstehende Probleminstanz die gleiche Lösung hat wie die ursprüngliche Instanz, wobei allerdings aus einer Liste zwei Listen (etwa der halben Länge) geworden sind. Einmaliges Pointer Jumping kann in konstanter Zeit ausgeführt werden. Wird die Operation $\lceil \log_2 n \rceil$-mal nacheinander ausgeführt, zeigt danach jedes Listenelement außer dem letzten auf das letzte Element, und die Lösung des Problems ist direkt abzulesen.

**Satz** 13.4: Instanzen des List Ranking-Problems der Größe $n$ können in $O(\log n)$ Zeit auf einer PRAM mit $n$ Prozessoren gelöst werden.

Auch für das List Ranking-Problem gibt es PRAM-Algorithmen mit optimalem Speed-up (R. Cole and U. Vishkin, Approximate parallel scheduling. Part I: The basic technique with applications to optimal parallel list ranking in logarithmic time, *SIAM J. Comput.* **17** (1988), pp. 128–142; R. J. Anderson and G. L. Miller, Deterministic parallel list ranking, 3rd Aegean Workshop on Computing (1988), Springer Lecture Notes in Computer Science, Vol. 319, pp. 81–90). Diese sind allerdings wesentlich komplizierter als der obige Algorithmus.

Ein letztes nicht-triviales Beispiel soll die Mächtigkeit der PRAM illustrieren. Es geht dabei um die Auswertung von arithmetischen Ausdrücken. Um das Beispiel interessanter zu machen, wollen wir annehmen, daß die PRAM nicht nur mit ganzen, sondern mit beliebigen reellen Zahlen rechnen kann, insbesondere daß sie reelle Zahlen dividieren kann. Der auszuwertende Ausdruck hat also Operatoren $+$, $-$, $*$ und $/$ und reelle Zahlen als Operanden. Der Einfachheit halber wollen wir annehmen, daß der Ausdruck in Form eines Baums gegeben ist (Fig. 13.6).

Jeder innere Knoten ist mit einem der vier Operatoren $+$, $-$, $*$ und $/$ beschriftet und hat ein linkes und ein rechtes Kind, und jedes Blatt ist mit einer reellen Zahl beschriftet. Der Baum ist in üblicher Weise repräsentiert: Jeder innere Knoten hat Zeiger auf seine Kinder, und jeder Knoten außer der Wurzel hat einen Zeiger auf seinen Vater. Jeder Knoten ist in natürlicher Weise mit einem reellen *Wert* assoziiert (s. Fig. 13.6), und die Aufgabe besteht darin, den Wert der Wurzel zu bestimmen. Wir wollen dabei annehmen, daß keine Division durch Null auftritt. Sei $n$ die Anzahl der Blätter (d.h. der Operanden). Da es genau $n - 1$ innere Knoten gibt, ist die Größe des Baums $O(n)$, und wir nehmen an (wie




$$(7+5)/(9-(2*3))$$

Fig. 13.6. Ein arithmetischer Ausdruck und der entsprechende Baum.

es natürlich ist), daß seine Darstellung anfangs in den ersten $O(n)$ globalen Speicherzellen steht.

Es ist nicht schwer zu sehen, daß eine PRAM den Baum in einer Zeit auswerten kann, die proportional zu seiner Höhe ist, einfach indem alle Knoten auf gleicher Höhe gleichzeitig ausgewertet werden, und zwar von den Blättern zur Wurzel. Ist der Baum schön balanciert, kann er also in $O(\log n)$ Zeit ausgewertet werden, und auf etwas Besseres können wir kaum hoffen. Der Baum könnte aber auch stark unbalanciert sein, die Höhe könnte $\Theta(n)$ sein, und der obige PRAM-Algorithmus wäre kaum schneller als eine sequentielle Auswertung. Wir wollen zeigen, daß der Baum, egal, wie er aussieht, in $O(\log n)$ Zeit ausgewertet werden kann. Dabei benutzen wir die sogenannte *Tree Contraction*-Technik.

Der erste Schritt ist, wie beim List Ranking, das Problem durch Einführung von Kantenbeschriftungen zu verallgemeinern. Die Kantenbeschriftungen sollen Funktionen in der Menge $\mathcal{F}$ sein, die wie folgt definiert ist: $\mathcal{F}$ ist die Menge aller Funktionen $f$ der Form

$$f(x) = \frac{ax+b}{cx+d}, \quad a,b,c,d \in \mathbb{R}$$

mit Definitionsmenge $Dom(f) = \{x \in \mathbb{R} \mid cx+d \neq 0\}$. Wir repräsentieren die Funktion $f$ durch das 4-Tupel $(a,b,c,d)$.

Wir brauchen die folgenden Abgeschlossenheitseigenschaften von $\mathcal{F}$.

**Lemma** 13.5: Sei $f \in \mathcal{F}$ und sei $g$ eine Funktion der Form $g = c \odot f$ oder $g = f \odot c$, wobei $c \in \mathbb{R}$ und $\odot \in \{+, -, *, /\}$. Dann enthält $\mathcal{F}$ eine Erweiterung $h$ von $g$, die in konstanter Zeit aus $f$ und $c$ berechnet werden kann.

**Bemerkung:** Eine *Erweiterung* von $g$ ist eine Funktion $h$ mit $Dom(g) \subseteq Dom(h)$ und $h(x) = g(x)$ für alle $x \in Dom(g)$.

**Beispiel:** Sei $f \in \mathcal{F}$ die Funktion $\frac{x}{x-1}$ (d.h. $(a,b,c,d) = (1,0,1,-1)$ und $Dom(f) = \mathbb{R}\backslash\{1\}$).

(1) Sei $g = f + 3$. Dann ist $Dom(g) = Dom(f) = \mathbb{R}\backslash\{1\}$ und $g(x) = \frac{4x-3}{x-1}$ für alle $x \in Dom(g)$, d.h. $g$ hat das 4-Tupel $(4,-3,1,-1)$.

(2) Sei $g = 1/f$. Dann ist $Dom(g) = \mathbb{R}\backslash\{0,1\}$ und $g(x) = \frac{x-1}{x}$ für alle $x \in Dom(g)$. $g \notin \mathcal{F}$, aber $h \in \mathcal{F}$, wobei $h$ die Erweiterung von $g$ mit $Dom(h) = \mathbb{R}\backslash\{0\}$ und $h(x) = \frac{x-1}{x}$ für alle $x \in Dom(h)$ ist.

**Lemma** 13.6: Seien $f, g \in \mathcal{F}$. Dann enthält $\mathcal{F}$ eine Erweiterung $h$ von $g \circ f$, die in konstanter Zeit aus $f$ und $g$ berechnet werden kann.

**Beweis:** Aufgabe 14.2. ∎

Ähnlich wie beim "Telefonsummieren" kann man sich die Auswertung des vorliegenden Baums so vorstellen, daß reelle Werte von den Blättern zur Wurzel aufsteigen, wobei sie an inneren Knoten den jeweiligen Operatoren entsprechend kombiniert werden. Die Funktion, mit der eine Kante beschriftet ist, soll jetzt zusätzlich den Wert transformieren, der die Kante überquert. Wir wählen sämtliche Kantenbeschriftungen anfangs als die Identitätsfunktion (repräsentiert durch $(1,0,0,1)$), womit klar ist, daß die Einführung der Kantenbeschriftungen die Lösung des Problems, also den Wert der Wurzel, nicht verändert. Ähnlich wie beim List Ranking beschreiben wir jetzt, wie wir den Baum transformieren können, ohne den Wert der Wurzel zu verändern. Die Baumtransformation, die wir benutzen wollen, besteht in der Entfernung eines Blattes $v$ zusammen mit seinem Vater $u$, wobei $u$ nicht die Wurzel ist (Fig 13.7).

Sei $w$ der Bruder von $v$, sei $a_v$ die reelle Zahl, mit der $v$ beschriftet ist, sei $\odot$ die Operation, mit der $u$ beschriftet ist, und seien $f_u$, $f_v$ und $f_w$ die Beschriftungen der Kanten, die $u$, $v$ und $w$ mit ihren Vätern verbinden. Ist $v$ der rechte Sohn von $u$, entfernen wir $u$ und $v$, machen $w$ zu einem Sohn des Vaters von $u$ und beschriften die Kante zwischen

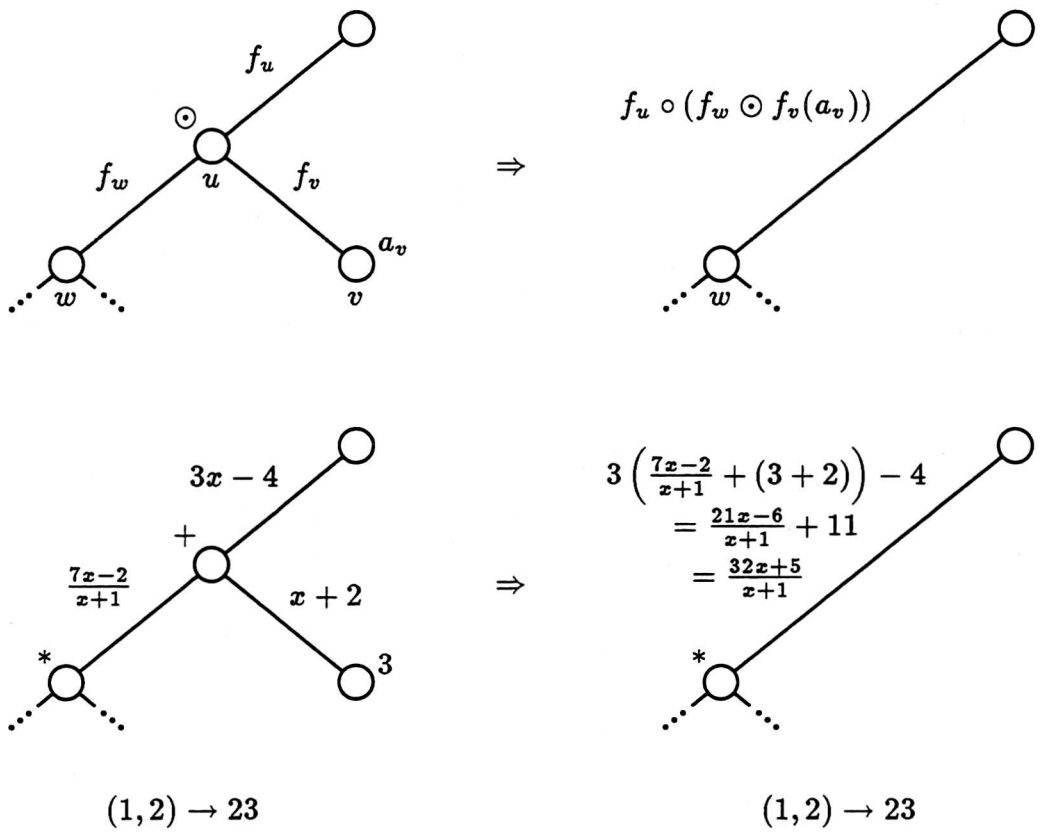



Fig. 13.7. Die *Shunt*-Operation.

$w$ und seinem neuen Vater mit einer Erweiterung der Funktion

$$x \mapsto f_u(f_w(x) \odot f_v(a_v)),$$

die in $\mathcal{F}$ liegt. Ist $v$ der linke Sohn von $u$, verfahren wir identisch, außer daß $f_w(x) \odot f_v(a_v)$ durch $f_v(a_v) \odot f_w(x)$ ersetzt wird. Nach den Lemmata 13.5 und 13.6 kann die beschriebene Operation in konstanter Zeit von einem Prozessor ausgeführt werden, und man kann sich davon überzeugen, daß der Wert der Wurzel unverändert bleibt.

Wir haben somit eine Operation *Shunt* definiert, mit der wir ein beliebiges Blatt zusammen mit seinem Vater entfernen können, ohne neue Blätter entstehen zu lassen. Wir können *Shunt*-Operationen gleichzeitig an verschiedenen Blättern ausführen, solange die Mengen der jeweils betroffenen Knoten ($u$, $v$ und $w$ oben) paarweise disjunkt sind. Das ist dann gewährleistet, wenn zwei Blätter $v$ und $v'$, die gleichzeitig entfernt werden, zwei Bedingungen erfüllen: (1) $v$ und $v'$ sind entweder beide rechte oder beide linke Söhne ihrer Väter, und (2) in der Reihenfolge aller Blätter im Baum von links nach rechts folgen

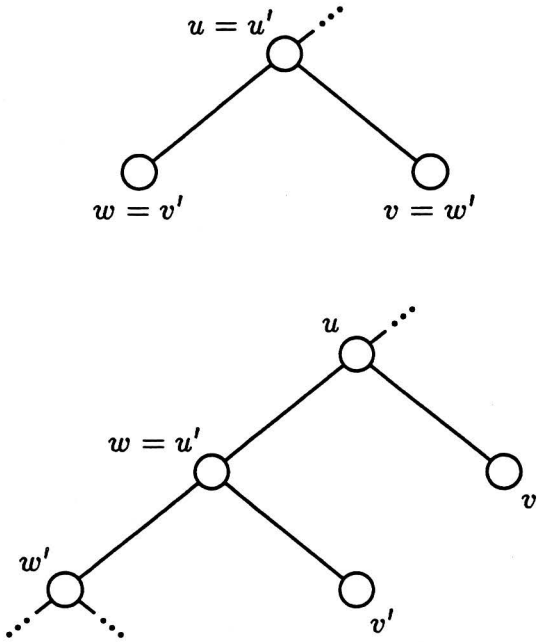



Fig. 13.8. Die Blätter $v$ und $v'$ können nicht gleichzeitig entfernt werden.

$v$ und $v'$ nicht unmittelbar aufeinander (Fig. 13.8).

Wir gehen jetzt so vor: Zuerst numerieren wir die Blätter fortlaufend von links nach rechts. Dazu benutzen wir die sogenannte *Euler-Tour*-Technik (Fig. 13.9). Eine Liste wird um den Baum gelegt, die alle Blätter in der Reihenfolge von rechts nach links besucht. Jede Kante der Liste, die ein Blatt verläßt, wird mit 1 markiert, und alle anderen Kanten werden mit 0 markiert. Danach lösen wir das dadurch definierte generalisierte List Ranking-Problem mit dem Algorithmus aus Satz 13.4, wodurch wir die Nummern der Blätter erhalten.

Nach der Numerierung der Blätter führen wir den folgenden Algorithmus aus (Fig. 13.10):

**for** $t := 1$ **to** $\lceil \log_2 n \rceil$ **do**

(*) Entferne alle Blätter mit ungerader Nummer, die linke Söhne sind;

(**) Entferne alle Blätter mit ungerader Nummer, die rechte Söhne sind;

Teile alle verbleibenden Blattnummern durch 2;

Ein Blatt, dessen Vater die Wurzel ist (ein solches Blatt muß das erste oder das letzte sein), wird dabei nicht wirklich entfernt, sondern wir tun nur so, als wäre es nicht mehr da. Wichtig ist ebenfalls, daß die Ausführung des Befehls (*) die Menge der Blätter nicht

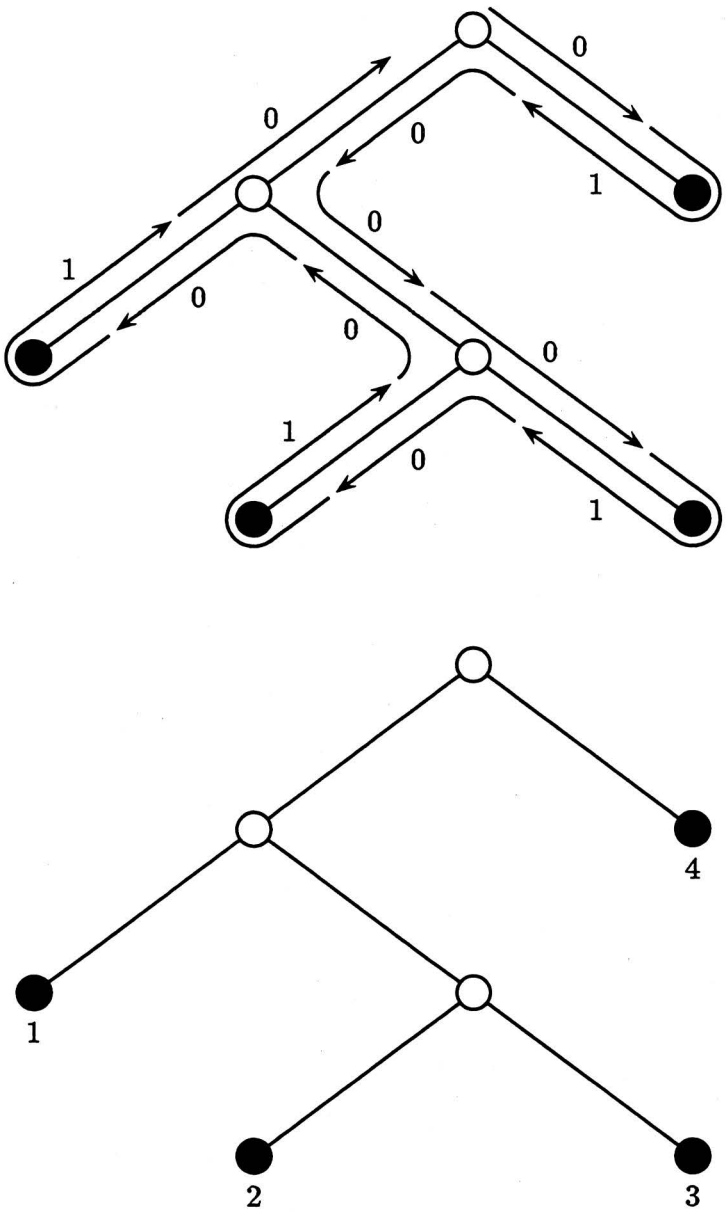



Fig. 13.9. Numerieren der Blätter mit der Euler-Tour-Technik.

verändert, die vom Befehl (**) betroffen sind.

Da ein Schleifendurchlauf alle Blätter mit ungerader Nummer entfernt, numeriert die letzte Zeile die verbleibenden Blätter gerade wieder richtig. Jeder Durchlauf entfernt mindestens die Hälfte der Blätter. Daraus folgt, daß nach $\lceil \log_2 n \rceil$ Iterationen gar keine Blätter mehr übrig sind—außer den beiden, die wir nicht wirklich entfernt haben. Wir haben jetzt also einen Baum mit drei Knoten, dessen Wurzel den gleichen Wert hat wie

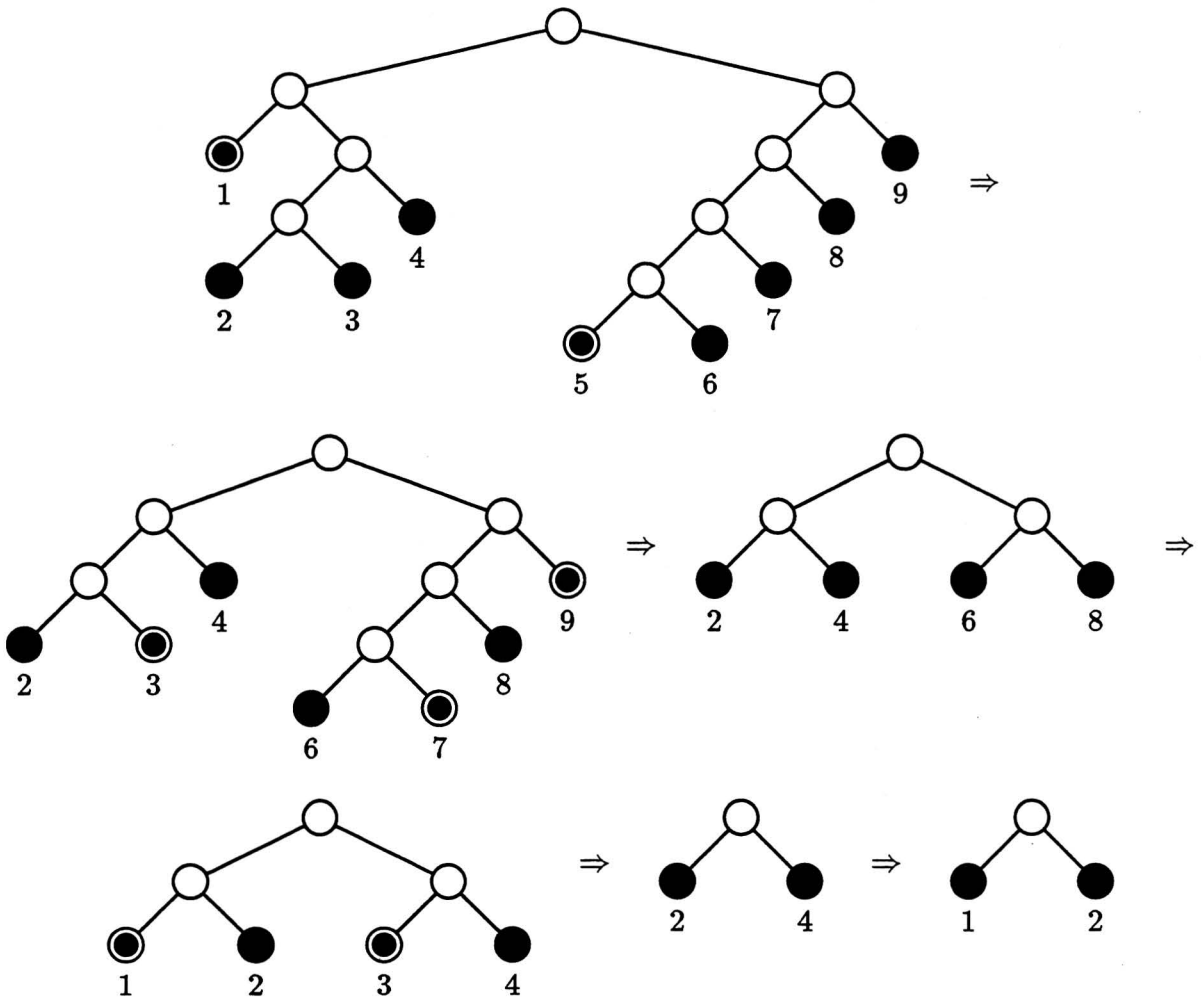



Fig. 13.10. Baumauswertung mit Tree Contraction.

die Wurzel des ursprünglichen Baums, und brauchen diesen Baum nur noch sequentiell auszuwerten.

**Satz** 13.7: Ein als Baum gegebener arithmetischer Ausdruck mit $n$ reellwertigen Operanden, bei dessen Auswertung keine Division durch Null auftritt, kann auf einer PRAM mit $O(n)$ Prozessoren, die arithmetische Operationen auf reellen Zahlen in konstanter Zeit ausführen können, in $O(\log n)$ Zeit ausgewertet werden.

# 14 Parallele Komplexitätstheorie

Während sich die PRAM für den praktischen Entwurf von parallelen Algorithmen großer Beliebtheit erfreut, ist es für komplexitätstheoretische Untersuchungen oft vorteilhaft, sich auf ein einfacheres Modell zurückzuziehen, nämlich das der uniformen Schaltkreisfamilien.

Wir haben bereits in Kapitel 5 kurz mit Schaltkreisen Bekanntschaft gemacht. Ein Schaltkreis mit $n$ Eingängen berechnet eine Funktion von $\{0,1\}^n$ nach $\{0,1\}$. Eine *Schaltkreisfamilie* ist eine Menge $\{\gamma_n\}_{n=0}^{\infty}$, wobei $\gamma_n$ ein Schaltkreis mit $n$ Eingängen ist, für alle $n \in \mathbb{N}_0$. Eine Schaltkreisfamilie berechnet eine Funktion von $\{0,1\}^*$ nach $\{0,1\}$, akzeptiert also eine Sprache $L \subseteq \{0,1\}^*$.

Die Schaltkreise, die wir bisher gesehen haben, waren solche mit *beschränktem Fanin*: Jedes $\wedge$- oder $\vee$-Gatter hat genau zwei Eingänge. Dem gegenüber stehen Schaltkreise mit *unbeschränktem Fanin*, bei denen $\wedge$- und $\vee$-Gatter beliebig viele Eingänge haben dürfen ($\neg$-Gatter haben immer genau einen Eingang). In beiden Fällen definieren wir die *Größe* $|\gamma|$ eines Schaltkreises $\gamma$ als die Anzahl seiner Gatter plus die Anzahl seiner Drähte. Die *Tiefe* eines Schaltkreises $\gamma$ ist die Länge eines längsten Pfades in $\gamma$ von einem Eingangsgatter zum Ausgangsgatter. Tiefe kann man sich als Berechnungszeit vorstellen.

Eine Schaltkreisfamilie $\{\gamma_n\}_{n=0}^{\infty}$ heißt *uniform*, wenn es eine TM gibt, die bei Eingabe $I^n$ den Schaltkreis $\gamma_n$ (bzw. seine Darstellung) unter Benutzung von $O(\log |\gamma_n|)$ Arbeitsplatz berechnet, für alle $n \in \mathbb{N}_0$. Wir wissen bereits aus Aufgabe 1.3, daß Schaltkreisfamilien ohne die Einschränkung der Uniformität viel zu mächtig sind.

**Satz** 14.1: Zu jedem Schaltkreis der Größe $s$ und der Tiefe $d$ mit unbeschränktem Fanin gibt es einen äquivalenten Schaltkreis (d.h. einen Schaltkreis, der die gleiche Funktion berechnet) der Größe $O(s)$ und der Tiefe $O(d \log s)$, dessen Fanin beschränkt ist.

**Beweis:** Ersetze jedes $\wedge$- oder $\vee$-Gatter mit mehr als zwei Eingängen durch einen Baum von $\wedge$- oder $\vee$-Gattern mit zwei Eingängen. Das verändert die Größe des Schaltkreises

höchstens um einen konstanten Faktor, und da der Fanin jedes Gatters durch $s$ beschränkt ist, erhöht es die Tiefe um einen Faktor von $O(\log s)$. ∎

Da die Konstruktion aus dem Beweis von Satz 14.1 offenbar auch eine uniforme Familie in eine uniforme Familie überführt, brauchen wir im folgenden Satz 14.2 nicht anzugeben, ob wir Schaltkreise mit beschränktem oder unbeschränktem Fanin meinen.

**Satz** 14.2: Eine Sprache $L \in \{0,1\}^*$ liegt genau dann in P, wenn sie von einer uniformen Schaltkreisfamilie polynomieller Größe akzeptiert wird.

**Beweis:** Zu einer beliebigen Sprache $L$ in P und einer beliebigen Eingabe $x$ haben wir im Beweis von Satz 5.4 einen Schaltkreis konstruiert, der genau dann den Wert 1 berechnet, wenn $x \in L$. Die Eingabe $x$ war damals fest in den Schaltkreis "verdrahtet", kann aber leicht variabel gemacht werden. Die so entstehende Schaltkreisfamilie hat polynomielle Größe und ist uniform.

Wird $L$ umgekehrt von einer uniformen Schaltkreisfamilie polynomieller Größe akzeptiert, können wir zu einer vorliegenden Eingabe $x$ in polynomieller Zeit den entsprechenden Schaltkreis konstruieren und auswerten. ∎

Die "parallele Berechnungsthese" besagt, daß parallele Zeit (oder Tiefe) in engem Zusammenhang mit sequentiellem Platz steht, nämlich daß die beiden Ressourcen ungefähr "gleich mächtig" sind. Die These ist kein rigoroser Satz, und sie schlägt auch in bestimmten Situationen fehl; sie ist aber eine nützliche Richtschnur. Die beiden folgenden Sätze belegen die parallele Berechnungsthese in einem konkreten Fall.

**Satz** 14.3: Sei $L$ eine Sprache, die von einer uniformen Schaltkreisfamilie $\{\gamma_n\}_{n=0}^{\infty}$ mit beschränktem Fanin der Tiefe $f(n) \geq \log n$ und der Größe $2^{O(f(n))}$ akzeptiert wird. Dann ist $L \in \mathrm{SPACE}(f(n))$.

**Beweis:** Sei $x$ eine Eingabe mit $|x| = n$. Auf Grund der Uniformität kann eine Beschreibung von $\gamma_n$ mit $O(f(n))$ Arbeitsplatz erstellt werden. Wir werden $\gamma_n$ jetzt im wesentlichen an der Eingabe $x$ mit einer rekursiven Prozedur auswerten, die ein Gatter dadurch auswertet, daß die Werte der höchstens zwei Vorgängergatter rekursiv bestimmt werden, wonach die relevante Funktion auf die so erhaltenen Werte angewandt wird. Aufgerufen wird die rekursive Prozedur im Hauptprogramm mit dem Ausgangsgatter als Argument. Ein Problem bei dieser Vorgehensweise ist, daß wir nicht genug Arbeitsplatz haben, um die Beschreibung von $\gamma_n$ tatsächlich zu speichern. Um dieses Problem zu lösen, müssen wir

aber nur, wie im Beweis von Satz 5.1, die Teile der Beschreibung, die wir gerade brauchen, immer wieder neu generieren. Ein etwas schwierigeres Problem ist, daß die natürliche rekursive Prozedur zu viel Platz für den Rekursionsstack braucht, nämlich $f(n)$ Rekursionsabschnitte, von denen jeder $O(f(n))$ Platz für einen Gatternamen braucht. Wir lösen dieses Problem, indem wir neue Gatternamen einführen: Der Name des Ausgangsgatters ist das leere Wort, und generell hat das erste Vorgängergatter eines Gatters mit Namen $s$ den Namen $s0$, während das zweite Vorgängergatter den Namen $s1$ bekommt (Fig. 14.1).

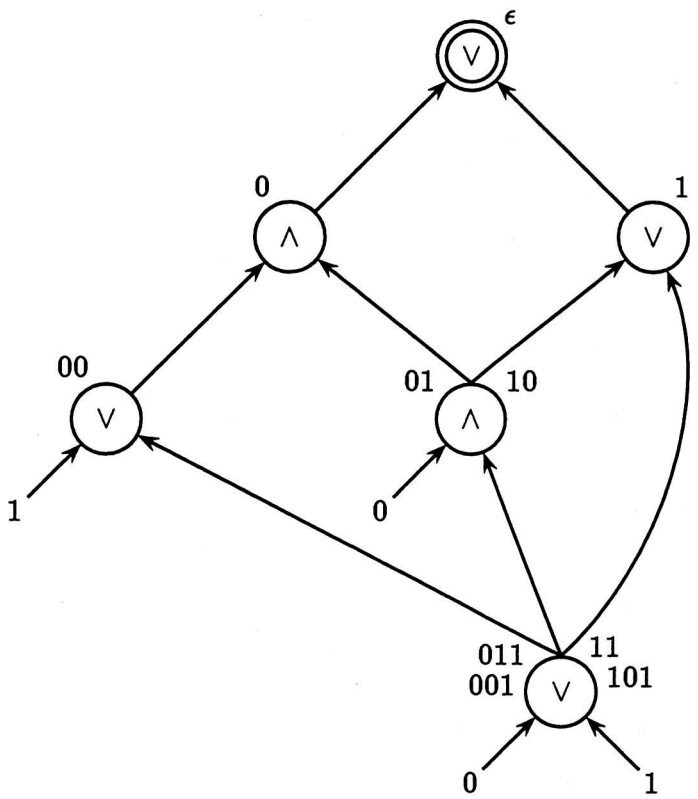



Fig. 14.1. Neue Gatternamen.

Ein Gatter kann dadurch viele Namen erhalten, nämlich so viele wie die Anzahl der Pfade von ihm zum Ausgangsgatter, aber das macht nichts. Wir müssen nur zu jedem Namen das entsprechende Gatter finden können, und das ist gewährleistet. Jetzt sind die Namen aller Gatter auf dem Rekursionsstack die Präfixe des Namens des letzten Gatters, so daß wir nur den letzten Namen aufheben müssen. Dafür brauchen wir nur $O(f(n))$ Platz. ∎

**Satz** 14.4: Sei $L \subseteq \{0,1\}^*$ eine Sprache in NSPACE$(f(n))$, wobei $f : \mathbb{N}_0 \to \mathbb{N}$ eine

konstruierbare Funktion mit $f(n) \geq \log n$ für alle $n \in \mathbb{N}$ ist. Dann wird $L$ von einer uniformen Schaltkreisfamilie mit beschränktem Fanin der Tiefe $O((f(n))^2)$ und der Größe $2^{O(f(n))}$ akzeptiert.

**Beweis:** Matrizen werden oft über dem Ring der reellen Zahlen multipliziert. Es gibt auch die *Boolesche* Matrixmultiplikation $\otimes$, bei der alle Einträge 0 oder 1 sind, während $\wedge$ die Rolle der reellen Multiplikation und $\vee$ die Rolle der reellen Addition übernimmt; sind $A = (a_{ij})$ und $B = (b_{ij})$ Boolesche $n \times n$ Matrizen, ist der $(i,k)$te Eintrag im Booleschen Produkt $A \otimes B$ also gerade

$$\bigvee_{j=1}^{n} (a_{ij} \wedge b_{jk}),$$

für $i = 1, \ldots, n$ und $k = 1, \ldots, n$.

Sei jetzt $G = (V, E)$ ein gerichteter Graph auf der Knotenmenge $V = \{1, \ldots, n\}$ und sei $A = (a_{ij})$ die Adjazenzmatrix von $G$, also

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

für $i, j \in V$. Wir definieren $A^{(0)} = (a_{ij}^{(0)})$ als $A + I$, wobei $I$ die $n \times n$ Einheitsmatrix ist, und $A^{(k)} = (a_{ij}^{(k)})$ als $A^{(k-1)} \otimes A^{(k-1)}$, für alle $k \in \mathbb{N}$. Man sieht leicht durch Induktion, daß $a_{ij}^{(k)} = 1$ genau dann, wenn $G$ einen Weg von $i$ nach $j$ der Länge $\leq 2^k$ enthält, für alle $i, j \in V$ und $k \in \mathbb{N}_0$. Also kann man überprüfen, ob es einen Weg in $G$ von einem Startknoten $s$ zu einem Endknoten $t$ gibt, indem man $A^{(0)}$ $\lceil \log_2 n \rceil$-mal nacheinander quadriert und den $(s,t)$ten Eintrag in der Ergebnismatrix inspiziert. Das kann mit einem Schaltkreis mit beschränktem Fanin, Tiefe $O((\log n)^2)$ und Größe $O(n^3 \log n)$ getan werden ($O(\log n)$ Matrixmultiplikationen; jede braucht $n^2$ binäre Bäume von $\vee$-Gattern über $n$ Blättern).

Sei jetzt $L \in \mathrm{NSPACE}(f(n))$, und sei $M$ eine NTM, die das bezeugt. Wie in Kapitel 3 betrachten wir zu jeder Eingabe $x$ mit $|x| = n$ eine Menge $V_M(x)$ von Konfigurationen, die alle von $Init_M(x)$ aus erreichbaren Konfigurationen umfaßt. Da solche Konfigurationen höchstens $cf(n)$ Platz benutzen, für eine passende Konstante $c \in \mathbb{N}$, und $f(n) \geq \log n$, wissen wir aus Kapitel 3, daß wir davon ausgehen können, daß $|V_M(x)| = 2^{O(f(n))}$. Wir möchten jetzt den obigen Matrixmultiplikationsalgorithmus auf dem von $V_M(x)$ aufgespannten Teilgraphen des Konfigurationsgraphen anwenden, um zu entscheiden, ob es einen Weg von $Init_M(x)$ zu einer akzeptierenden Endkonfiguration gibt. Dazu brauchen wir einen Schaltkreis der Tiefe $O((f(n))^2)$ und der Größe $2^{O(f(n))}$, wie gewünscht (wir bilden zuletzt ein großes Oder über alle akzeptierenden Endzustände). Allerdings besteht

das folgende Problem, das zu verstehen wichtig ist: Wir wenden hier den Matrixmultiplikationsalgorithmus nicht auf eine feste Adjazenzmatrix $A$ an, sondern auf eine Matrix $A_x$, die von der Eingabe $x$ abhängt. Der Schaltkreis, den wir konstruieren, darf jedoch nicht von $x$, sondern nur von $n = |x|$ abhängen. Aber wenn wir für die Dauer dieses Beweises zwei Konfigurationen als identisch ansehen, wenn sie sich höchstens in der Eingabe (nicht aber in der Position des Eingabekopfes) unterscheiden, kann jeder Eintrag in $A_x$ in konstanter Tiefe aus $x$ berechnet werden; denn ob ein Übergang $K \to K'$ zwischen zwei Konfigurationen möglich ist, hängt von höchstens einem Bit in $x$ ab, nämlich von dem in $K$ gelesenen Bit. Damit können wir in $O((f(n))^2)$ Tiefe und $2^{O(f(n))}$ Größe einen Schaltkreis konstruieren, der für alle Eingaben $x$ mit $|x| = n$ korrekt entscheidet, ob sie von $M$ akzeptiert werden. Die entsprechende Schaltkreisfamilie ist uniform und akzeptiert die Sprache $L$. ∎

Die wichtigste Komplexitätsklasse, die durch Schaltkreise definiert wird, ist NC (NC steht für "Nick's Class"; "Nick" ist Nicholas Pippenger). Für jedes $k \in \mathbb{N}_0$ sei $\mathrm{NC}^k$ die folgende Klasse: $L \in \mathrm{NC}^k$ genau dann, wenn es eine uniforme Familie $\{\gamma_n\}_{n=0}^{\infty}$ von Schaltkreisen mit beschränktem Fanin gibt, die $L$ akzeptiert, so daß $\gamma_n$ für alle $n \in \mathbb{N}$ Tiefe $O((\log n)^k)$ und Größe $n^{O(1)}$ hat. Wir setzen $\mathrm{NC} = \bigcup_{k=0}^{\infty} \mathrm{NC}^k$. $\mathrm{AC}^k$, für $k \in \mathbb{N}_0$, und AC sind analog zu $\mathrm{NC}^k$ und NC definiert; nur wird nicht mehr gefordert, daß die Schaltkreise beschränkten Fanin haben müssen. Mit den Sätzen 14.1 und 14.2 weist man leicht nach, daß

$$\mathrm{NC}^0 \subseteq \mathrm{AC}^0 \subseteq \mathrm{NC}^1 \subseteq \mathrm{AC}^1 \subseteq \mathrm{NC}^2 \subseteq \mathrm{AC}^2 \subseteq \cdots \subseteq \mathrm{NC} = \mathrm{AC} \subseteq \mathrm{P}.$$

Es ist leicht zu sehen, daß $\mathrm{NC}^0 \subset \mathrm{AC}^0$; $\mathrm{NC}^0$ ist keine sehr interessante Klasse. Man kann sogar zeigen, daß $\mathrm{AC}^0 \subset \mathrm{NC}^1$: Das Problem, die Parität von $n$ Bits zu bestimmen (d.h. Ist die Summe der Bits ungerade?), liegt in $\mathrm{NC}^1$ (überlegen Sie!), aber nicht in $\mathrm{AC}^0$ (M. Furst, J. B. Saxe and M. Sipser, Parity, circuits, and the polynomial-time hierarchy, 22nd Annual Symp. on Foundations of Computer Science (1981), pp. 260–270; J. Hastad, Almost optimal lower bounds for small depth circuits, 18th Annual ACM Symp. on Theory of Computing (1986), pp. 6–20).

NC wird traditionell als die Klasse der Sprachen angesehen, die gut parallelisierbar sind. Wenn wir Tiefe mit Zeit und Schaltkreisgröße mit "Hardwareaufwand" gleichsetzen, so ist NC die Klasse der Sprachen, die in polylogarithmischer Zeit mit polynomiell viel Hardware akzeptiert werden können, also sehr schnell und mit noch vertretbaren Hardwarekosten. Die große offene Frage hier lautet: Ist NC = P? Lassen sich, mit anderen

Worten, alle Probleme in P gut parallelisieren? Man vermutet das Gegenteil. Es gibt Probleme in P, die sich hartnäckig jedem Parallelisierungsversuch widersetzen. Zu diesen gehören alle P-vollständigen Probleme.

**Satz** 14.5: Sei $L$ ein P-vollständiges Problem. Dann gilt: $L \in \mathrm{NC} \Leftrightarrow \mathrm{NC} = \mathrm{P}$.

**Beweis:** Eine Richtung ist klar. Sei also $L \in \mathrm{NC}$ und sei $L'$ eine beliebige Sprache in P. Da $L$ P-vollständig ist, gibt es eine Reduktion $R$, die mit logarithmischem Arbeitsplatz berechnet werden kann, so daß $x \in L' \Leftrightarrow R(x) \in L$. Durch "Auffüllen" mit einem passenden Sonderzeichen können wir erreichen, daß $|R(x)|$ nur von $|x|$ abhängt. Das Entscheidungsproblem, ein festes Bit von $R(x)$ zu berechnen, kann mit logarithmischem Arbeitsplatz gelöst werden, liegt also nach Satz 14.4 in NC. Unter Benutzung einer uniformen Schaltkreisfamilie mit polylogarithmischer Tiefe und polynomieller Größe können wir also gleichzeitig alle Bits von $R(x)$ berechnen. Speisen wir diese Bits in einen Schaltkreis mit $|R(x)|$ Eingängen aus einer Familie, die $L$ akzeptiert, erhalten wir eine Schaltkreisfamilie, die $L'$ akzeptiert. Nach Annahme kann die Familie, die $L$ akzeptiert, so gewählt werden, daß sie uniform ist und polylogarithmische Tiefe und polynomielle Größe hat. Das gilt dann auch für die Familie, die $L'$ akzeptiert. ∎

Es ist leicht zu sehen, daß ein Schaltkreis mit unbeschränktem Fanin, Tiefe $d$ und Größe $s$ im wesentlichen in Zeit $O(d)$ von einer PRAM mit $O(s)$ Prozessoren simuliert werden kann; höchstens die Konstruktion des Schaltkreises, die mit Uniformität zusammenhängt, könnte dabei ein Problem sein. Umgekehrt kann eine PRAM mit $p$ Prozessoren, die in Zeit $t$ arbeitet, im wesentlichen von einem Schaltkreis mit unbeschränktem Fanin, Tiefe $O(t)$ und Größe $(pt)^{O(1)}$ simuliert werden. Für den Beweis muß man die PRAM "in Hardware realisieren", wobei neue Gatter für jeden neuen Schritt benutzt werden (ein Schaltkreis darf ja keine Kreise enthalten). Das ist leicht, aber umständlich; unter anderem wird das Ergebnis aus Aufgabe 15.1 benötigt. Allerdings ist es notwendig, die PRAM so einzuschränken, daß sie nicht auf extrem großen Zahlen (in konstanter Zeit) operieren kann, denn diese Fähigkeit macht sie zu mächtig. Z. B. kann man die Multiplikations-Instruktion entfernen, die als einzige schnell große Zahlen entstehen lassen kann. Die Details dieser Simulation sind in (L. Stockmeyer and U. Vishkin, Simulation of parallel random access machines by circuits, *SIAM J. Comput.* **13** (1984), pp. 409–422) beschrieben. Auf den angedeuteten engen Zusammenhang zwischen PRAMs und Schaltkreisen mit unbeschränktem Fanin wollen wir hier nicht näher eingehen.

# 15 Orakel und die Polynomialzeit-Hierarchie

Eine *Orakelmaschine* ist eine Mehrband-TM oder NTM mit einem ausgezeichneten Arbeitsband, dem Orakelband, und drei ausgezeichneten Zuständen, **query**, **yes** und **no**. Bevor die Maschine rechnen kann, muß eine Orakelmenge $A$ festgelegt werden. Wann immer die Maschine in den Zustand **query** geht, passiert das folgende: Sei $x$ das auf dem Orakelband links vom Kopf stehende Wort. Ist $x \in A$, geht die Maschine in den Zustand **yes** über; sonst geht sie in den Zustand **no** über. In beiden Fällen wird gleichzeitig das Orakelband gelöscht (mit Leerzeichen aufgefüllt + Kopf am linken Bandende).

Informell gesprochen kann eine Orakelmaschine ihr Orakel über die Zugehörigkeit von beliebigen Wörtern zur Orakelmenge $A$ befragen. Alternativ kann man sich vorstellen, daß die Maschine mit einer mächtigen Instruktion $x \stackrel{?}{\in} A$ ausgestattet ist. In beiden Fällen kostet eine Befragung nur soviel Zeit, wie für das Niederschreiben der Frage benötigt wird. Eine TM mit einem mächtigen Orakel ist kein realistisches Berechnungsmodell mehr. Dennoch sind Orakelmaschinen und die von ihnen akzeptierten Sprachen, ähnlich wie NTMs und NP, wichtige Werkzeuge der Komplexitätstheorie.

Eine Orakelmaschine $M$, ausgestattet mit dem Orakel $A$, bezeichnen wir mit $M^A$. Ist $C$ eine Komplexitätsklasse, die durch zeitbeschränkte TMs oder NTMs definiert ist (z.B. P oder NP), schreiben wir $C^A$ für die Klasse der Sprachen, die von Orakelmaschinen mit dem Orakel $A$ und der gleichen Zeitbeschränkung akzeptiert werden (z.B. $\mathrm{NP}^A$ = Klasse der Sprachen, die in Polynomialzeit von NTMs mit Orakel $A$ akzeptiert werden). Ist $C$ eine Klasse von Sprachen, setzen wir ferner

$$\mathrm{P}^C = \bigcup_{A \in C} \mathrm{P}^A \quad \text{und} \quad \mathrm{NP}^C = \bigcup_{A \in C} \mathrm{NP}^A.$$

Beispiel: $\mathrm{P}^{\mathrm{P}} = \mathrm{P}$, denn eine Polynomialzeit-Maschine, die ein Polynomialzeit-Orakel befragen darf, kann auch in Polynomialzeit ohne das Orakel auskommen. Die *Polynomialzeit-Hierarchie* ist das System der wie folgt definierten Klassen:

$$\Delta_0 = \Sigma_0 = \Pi_0 = \mathrm{P}$$

$$\left.\begin{aligned} \Delta_k &= \mathrm{P}^{\Sigma_{k-1}} \\ \Sigma_k &= \mathrm{NP}^{\Sigma_{k-1}} \\ \Pi_k &= \mathrm{co}\Sigma_k \end{aligned}\right\} \quad \text{für alle } k \in \mathbb{N}.$$

Das folgende Lemma gibt unter anderem darüber Aufschluß, warum wir nur die $\Sigma$-Klassen als Exponenten benutzen—die $\Delta$- und $\Pi$-Klassen führen zu keinen neuen Klassen.

**Lemma** 15.1:

(a) $\Delta_k = \mathrm{co}\Delta_k$, für $k \in \mathbb{N}_0$;

(b) $\Delta_1 = \mathrm{P}$, $\Sigma_1 = \mathrm{NP}$ und $\Pi_1 = \mathrm{coNP}$;

(c) $\mathrm{P}^{\Pi_{k-1}} = \Delta_k$, $\mathrm{NP}^{\Pi_{k-1}} = \Sigma_k$ und $\mathrm{coNP}^{\Pi_{k-1}} = \Pi_k$, für $k \in \mathbb{N}$;

(d) $\mathrm{P}^{\Delta_k} = \Delta_k$, für $k \in \mathbb{N}_0$;

(e) $\mathrm{NP}^{\Delta_k} = \Sigma_k$ und $\mathrm{coNP}^{\Delta_k} = \Pi_k$, für $k \in \mathbb{N}$.

NB: Zum Beispiel $\mathrm{coNP}^{\Pi_{k-1}}$ bedeutet $\mathrm{co}(\mathrm{NP}^{\Pi_{k-1}})$, nicht $(\mathrm{coNP})^{\Pi_{k-1}}$ (das wir gar nicht definiert haben).

**Beweis:** (a) Falls $L \in \Delta_k$, wird $L$ von einer Polynomialzeit-TM $M$ mit einem Orakel $A$ entschieden. Durch Vertauschen der Zustände `accept` und `reject` in $M$, wie schon im Beweis von Satz 3.4, erhalten wir eine TM mit dem Orakel $A$, die $\overline{L}$ in Polynomialzeit akzeptiert.

(b) Wir haben bereits gesehen, daß $\Delta_1 = \mathrm{P}$. Die Relation $\Sigma_1 = \mathrm{NP}$ folgt genauso, und sie impliziert auch die dritte Relation $\Pi_1 = \mathrm{coNP}$.

(c) Hier wird behauptet, daß man eine Klasse $C$ als Orakelklasse durch co$C$ ersetzen kann, ohne daß sich etwas ändert. Das sieht man dadurch, daß man die Antworten des Orakels "andersrum" interpretieren kann (`yes` als `no` und umgekehrt).

(d) $k = 0$: Folgt aus Teil (b). $k \geq 1$: $\mathrm{P}^{\Delta_k} = \mathrm{P}^{\mathrm{P}^{\Sigma_{k-1}}} = \mathrm{P}^{\Sigma_{k-1}} = \Delta_k$, wobei die zweite Umformung dadurch begründet wird, daß eine P-Maschine keine zweite P-Maschine braucht, die ihre Anfragen für ein Orakel aus $\Sigma_{k-1}$ bearbeitet—das kann sie gleich selbst tun.

(e) $\mathrm{NP}^{\Delta_k} = \mathrm{NP}^{\mathrm{P}^{\Sigma_{k-1}}} = \mathrm{NP}^{\Sigma_{k-1}} = \Sigma_k$ mit einer ähnlichen Begründung wie im Beweis von Teil (d). ∎

**Lemma** 15.2: $\Sigma_k \cup \Pi_k \subseteq \Delta_{k+1} \subseteq \Sigma_{k+1} \cap \Pi_{k+1}$, für alle $k \in \mathbb{N}_0$.

**Beweis:**

$$\Sigma_k \subseteq \mathrm{P}^{\Sigma_k} = \Delta_{k+1} \subseteq \mathrm{NP}^{\Sigma_k} = \Sigma_{k+1}$$
$$\Pi_k \subseteq \mathrm{P}^{\Pi_k} = \Delta_{k+1} = \mathrm{co}\Delta_{k+1} \subseteq \mathrm{co}\Sigma_{k+1} = \Pi_{k+1}. \quad \blacksquare$$

Nach Lemma 15.2 hat die Polynomialzeit-Hierarchie die in Fig. 15.1 dargestellte Struktur. Wir setzen $\mathrm{PH} = \bigcup_{k=0}^{\infty} \Sigma_k = \bigcup_{k=0}^{\infty} \Delta_k = \bigcup_{k=0}^{\infty} \Pi_k$.

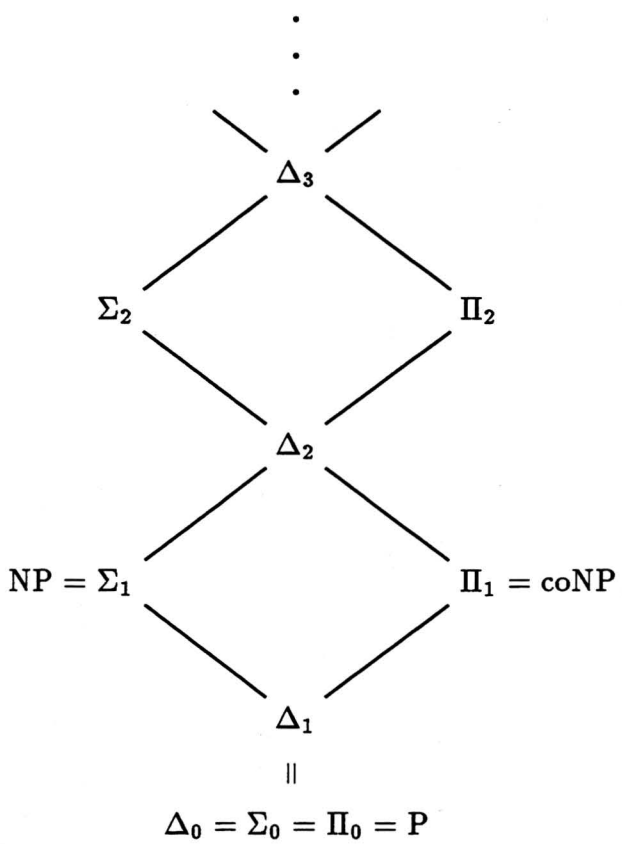



Fig. 15.1. Die Polynomialzeit-Hierarchie.

Wo liegt eigentlich diese Polynomialzeit-Hierarchie? Sie enthält offensichtlich NP; aber enthält sie zum Beispiel auch PSPACE?

**Satz** 15.3: $\mathrm{PH} \subseteq \mathrm{PSPACE}$.

**Beweis:** Wir zeigen durch Induktion über $k$, daß $\Sigma_k \subseteq \mathrm{PSPACE}$ für alle $k \in \mathbb{N}_0$, woraus die Behauptung folgt. Für $k = 0$ und $k = 1$ wissen wir es schon aus Aufgabe 6.1. Sei jetzt $k \geq 1$ und sei $L \in \Sigma_k$. $L$ wird also von einer NTM $M$ akzeptiert, die in Polynomialzeit läuft

und ein Orakel $A$ aus $\Sigma_{k-1}$ benutzt. Wie im Beweis von Satz 3.2(c) simulieren wir jetzt nacheinander alle Berechnungen von $M$ und akzeptieren die Eingabe genau dann, wenn mindestens eine akzeptierende Berechnung gefunden wird. Wenn eine Frage an das Orakel $A$ gestellt wird, unterbrechen wir die Hauptsimulation, um das Orakel zu simulieren. Nach der Induktionsvoraussetzung ist $A \in \mathrm{PSPACE}$, so daß wir auch hierfür nur polynomiell viel Platz brauchen. ∎

**Lemma** 15.4: Sei $C$ eine Klasse aus der Polynomialzeit-Hierarchie. Sind $L_0$ und $L_1$ Sprachen, die beide zu $C$ gehören, gilt das auch für die Sprache

$$L_0 \uplus L_1 = \{(b,x) \mid (b = 0 \text{ und } x \in L_0) \text{ oder } (b = 1 \text{ und } x \in L_1)\}.$$

**Bemerkung:** $L_0 \uplus L_1$ heißt die *disjunkte Vereinigung* von $L_0$ und $L_1$.

**Beweis:** Wir zeigen die Behauptung zuerst für $C = \Sigma_k$ durch Induktion über $k$. Für $k = 0$ ist $C = \mathrm{P}$, und die Behauptung ist offensichtlich. Seien jetzt $k \geq 1$ und $L_0, L_1 \in \Sigma_k$. Wir haben also NTMs $M_0^{A_0}$ und $M_1^{A_1}$, die in Polynomialzeit laufen und $L_0$ und $L_1$ erkennen, wobei die Orakel $A_0$ und $A_1$ aus $\Sigma_{k-1}$ sind. Nach Induktionsvoraussetzung ist $A_0 \uplus A_1 \in \Sigma_{k-1}$. Es ist aber leicht, eine NTM mit dem Orakel $A_0 \uplus A_1$ anzugeben, die $L_0 \uplus L_1$ in Polynomialzeit akzeptiert: Bei Eingabe $(b,x)$ wird die Maschine $M_b$ auf der Eingabe $x$ simuliert, wobei die Anfragen an das Orakel $A_b$ in der offensichtlichen Weise durch Anfragen an das Orakel $A_0 \uplus A_1$ simuliert werden.

Für $C = \Delta_k$ läuft der Beweis genauso; nur sind die Maschinen $M_0$ und $M_1$ jetzt deterministisch.

Für $C = \Pi_k$ gehen wir so vor: $L_0, L_1 \in \Pi_k \Rightarrow \overline{L_0}, \overline{L_1} \in \mathrm{co}\Pi_k = \Sigma_k \Rightarrow \overline{L_0} \uplus \overline{L_1} \in \Sigma_k \Rightarrow \overline{L_0 \uplus L_1} \in \Sigma_k \Rightarrow L_0 \uplus L_1 \in \mathrm{co}\Sigma_k = \Pi_k$. ∎

Man kann die Aussage von Lemma 15.4 so verstehen, daß zwei Orakel (allgemeiner: konstant viele Orakel) z. B. aus $\Sigma_k$ durch ein einziges Orakel aus $\Sigma_k$ ersetzt werden können. Damit zeigt man leicht, daß die Klassen aus der Polynomialzeit-Hierarchie unter Vereinigung und Durchschnitt abgeschlossen sind.

**Lemma** 15.5: Sei $C$ eine Klasse aus der Polynomialzeit-Hierarchie und seien $L_0, L_1 \in C$. Dann ist auch $L_0 \cup L_1 \in C$ und $L_0 \cap L_1 \in C$.

**Beweis:** Sei $C = \Sigma_k$ mit $k \geq 1$. Für $b = 0, 1$ wird $L_b$ von einer NTM $M_b$ entschieden, die in Polynomialzeit arbeitet und ein Orakel $A_b$ aus $\Sigma_{k-1}$ benutzt. Es ist trivial, die Maschinen $M_0$ und $M_1$ so zu modifizieren, daß sie statt dessen mit dem Orakel $A = A_0 \uplus A_1$ arbeiten, das nach Lemma 15.4 ebenfalls in $\Sigma_{k-1}$ liegt. NTMs mit dem Orakel $A$, die $L_0 \cup L_1$ bzw. $L_0 \cap L_1$ in Polynomialzeit akzeptieren, sind jetzt leicht anzugeben: Die Maschinen $M_0$ und $M_1$ werden beide auf der Eingabe simuliert, und die Eingabe wird dann akzeptiert, wenn eine der Simulationen akzeptiert (bzw. wenn beide akzeptieren). Der Rest des Beweises wird dem Leser überlassen. ∎

Sind die Klassen in der Polynomialzeit-Hierarchie auch unter Komplement abgeschlossen? Für die Klassen $\Delta_k$ haben wir das bereits gesehen (Lemma 15.1(a)). Für die anderen Klassen hätte eine positive Antwort auf diese Frage drastische Konsequenzen (Satz 15.9).

Wir wollen jetzt eine sehr nützliche Charakterisierung der Klassen $\Sigma_k$ und $\Pi_k$ mit Hilfe von Quantoren herleiten. Die Zugehörigkeit konkreter Probleme zu Klassen der Polynomialzeit-Hierarchie wird in der Praxis mit dieser Charakterisierung nachgewiesen.

**Lemma** 15.6: Sei $k \in \mathbb{N}$. Eine Sprache $L$ gehört genau dann zu $\Sigma_k$, wenn sie in der Form

$$L = \{x \mid \exists y : |y| \leq p(|x|) \text{ und } (x, y) \in R\}$$

geschrieben werden kann, wobei $p : \mathbb{N}_0 \to \mathbb{N}$ ein Polynom und $R$ eine Sprache in $\Pi_{k-1}$ ist.

**Beweis:** Wir zeigen zuerst, daß jede Sprache $L$ der angegebenen Form zu $\Sigma_k$ gehört. Dazu müssen wir $L$ in Polynomialzeit mit einer NTM akzeptieren, die ein Orakel aus $\Sigma_{k-1}$ benutzen darf. Aber das ist leicht: Die Maschine rät einfach den String $y$ und überprüft mit Hilfe des Orakels $\overline{R} \in \Sigma_{k-1}$, daß in der Tat $(x, y) \in R$.

Für die umgekehrte Richtung benutzen wir Induktion über $k$. Für $k = 1$ haben wir unsere altbekannte Charakterisierung von NP mit Hilfe von Zeugen polynomieller Länge, die in polynomieller Zeit verifiziert werden können. Sei jetzt also $k \geq 2$ und sei $L \in \Sigma_k$. $L$ wird in Polynomialzeit von einer NTM $M$ entschieden, die ein Orakel $A$ aus $\Sigma_{k-1}$ benutzt.

Zu einer gegebenen Eingabe $x$ müssen wir einen geeigneten Zeugen $y$ angeben und nachweisen, daß er "in $\Pi_{k-1}$" überprüft werden kann. Was das eigentlich heißt, ist, daß wir eine NTM mit einem Orakel aus $\Sigma_{k-2}$ angeben müssen, die in Polynomialzeit läuft und die Eingabe $(x, y)$ genau dann akzeptiert, wenn der Zeuge $y$ *nicht* zu $x$ paßt. Der Zeuge $y$ besteht zunächst einmal aus (der Kodierung) einer akzeptierenden Berechnung von $M$.

In Polynomialzeit können wir leicht überprüfen, ob jede Konfiguration korrekt aus der vorherigen hervorgeht usw. Ist das nicht der Fall, können wir sofort akzeptieren, aber sonst müssen wir auch überprüfen, ob alle Fragen an das Orakel $A$ korrekt beantwortet sind.

Da $A \in \Sigma_{k-1}$, gibt es nach Induktionsvoraussetzung ein Polynom $q$ und eine Sprache $S \in \Pi_{k-2}$, so daß $z \in A$ genau dann, wenn es einen Zeugen $w$ mit $|w| \leq q(|z|)$ und $(z,w) \in S$ gibt. Für jede Frage, die $M$ an sein Orakel stellt und die positiv beantwortet wird, ergänzen wir den Zeugen $y$ um den entsprechenden Zeugen $w$. Mit dem Orakel $\overline{S} \in \Sigma_{k-2}$ können wir jetzt leicht akzeptieren, wenn es ein $z$ mit zugehörigem "Unterzeugen" $w$ gibt, so daß $(z,w) \notin S$. Also können wir die positiven Antworten "in $\Pi_{k-1}$" überprüfen.

Bei einer Frage $z \stackrel{?}{\in} A$, die negativ beantwortet wird, gibt es keinen Zeugen $w$. Aber $A$ wird in Polynomialzeit von einer NTM mit einem Orakel aus $\Sigma_{k-2}$ akzeptiert. Mit einem solchen Orakel können wir akzeptieren, wenn nicht alle Strings $z$ tatsächlich in $\overline{A}$ liegen. Damit haben wir auch die negativen Antworten "in $\Pi_{k-1}$" überprüft.

Bisher haben wir implizit zwei Sprachen $R_0$ und $R_1$ in $\Pi_{k-1}$ definiert, so daß $(x,y) \in R_0$, wenn $y$ eine Berechnung auf $x$ kodiert und alle positiven Orakelantworten korrekt sind, während $(x,y) \in R_1$, wenn $y$ eine Berechnung auf $x$ kodiert und alle negativen Orakelantworten korrekt sind. Wir setzen jetzt $R = R_0 \cap R_1$. Dann ist $L = \{x \mid \exists y : |y| \leq p(|x|)$ und $(x,y) \in R\}$, und nach Lemma 15.5 liegt $R$ in $\Pi_{k-1}$, wie gewünscht. ∎

Ganz analog zu Lemma 15.6 haben wir:

**Lemma** 15.7: Sei $k \in \mathbb{N}$. Eine Sprache $L$ gehört genau dann zu $\Pi_k$, wenn sie in der Form

$$L = \{x \mid \forall y, |y| \leq p(|x|) : (x,y) \in R\}$$

geschrieben werden kann, wobei $p : \mathbb{N}_0 \to \mathbb{N}$ ein Polynom und $R$ eine Sprache in $\Sigma_{k-1}$ ist.

**Beweis:** $L$ gehört genau dann zu $\Pi_k$, wenn $\overline{L}$ zu $\Sigma_k$ gehört, also wenn es ein Polynom $p$ und eine Sprache $S \in \Pi_{k-1}$ gibt, so daß

$$\overline{L} = \{x \mid \exists y : |y| \leq p(|x|) \text{ und } (x,y) \in S\}.$$

Aber dann ist

$$L = \{x \mid \forall y, |y| \leq p(|x|) : (x,y) \in \overline{S}\},$$

und mit $R = \overline{S} \in \Sigma_{k-1}$ haben wir das Gewünschte gezeigt. ∎

Die Lemmata 15.6 und 15.7 erlauben uns, "eine Stufe der Hierarchie durch einen Quantor einzutauschen". Wiederholen wir diese Operation, bis wir die Basisstufe P erreichen, erhalten wir den folgenden Satz.

**Satz** 15.8: Sei $k \in \mathbb{N}_0$.

(a) Eine Sprache $L$ gehört genau dann zu $\Sigma_k$, wenn es ein Polynom $p$ und eine Sprache $R \in \mathrm{P}$ gibt, so daß $L$ in der Form

$$L = \{x \mid \exists y_1 \forall y_2 \exists y_3 \cdots Q y_k : (x, y_1, y_2, \ldots, y_k) \in R\}$$

geschrieben werden kann, wobei die Strings $y_1, \ldots, y_k$ alle über der Menge $\{y : |y| \leq p(|x|)\}$ quantifiziert sind und $\exists$- und $\forall$-Quantoren sich abwechseln (der letzte Quantor $Q$ ist also genau dann $\exists$, wenn $k$ ungerade ist).

(b) Eine Sprache $L$ gehört genau dann zu $\Pi_k$, wenn es ein Polynom $p$ und eine Sprache $R \in \mathrm{P}$ gibt, so daß $L$ in der Form

$$L = \{x \mid \forall y_1 \exists y_2 \forall y_3 \cdots Q y_k : (x, y_1, y_2, \ldots, y_k) \in R\}$$

geschrieben werden kann, wobei die Strings $y_1, \ldots, y_k$ alle über der Menge $\{y : |y| \leq p(|x|)\}$ quantifiziert sind.

**Beweis:** Durch Induktion über $k$ und Benutzung der Lemmata 15.6 und 15.7. $p$ wählt man einfach als ein Polynom, das alle $k$ auftretenden Polynome dominiert. ∎

Wichtig ist in Satz 15.8, wie oft zwischen $\exists$- und $\forall$-Quantoren gewechselt wird, nicht die absolute Anzahl der Quantoren. Stehen zwei Quantoren der gleichen Art nebeneinander, z.B. $\exists y_i \exists y_{i+1}$, können wir sie durch einen Quantor der gleichen Art ersetzen, $\exists y_i'$, wobei $y_i'$ über einen größeren Bereich quantifiziert und wie ein Paar $(y_i, y_{i+1})$ aufgefaßt wird. Der erste (linkeste) Quantor $Q$ entscheidet, ob wir es mit einer Klasse der Form $\Sigma_k$ ($Q = \exists$) oder mit einer Klasse der Form $\Pi_k$ ($Q = \forall$) zu tun haben (eine gewisse Analogie zwischen Addition, $\vee$ und $\exists$ sowie zwischen Multiplikation, $\wedge$ und $\forall$ hat gerade die Namen $\Sigma_k$ und $\Pi_k$ motiviert).

Wir sind jetzt bestens gerüstet, um uns auf die Suche nach Problemen in der Polynomialzeit-Hierarchie zu begeben. Hier soll nur ein Beispiel angeführt werden.

MINIMUM CIRCUIT:

Eingabe: Ein Schaltkreis $\gamma$.

Frage: Ist es wahr, daß kein kleinerer Schaltkreis die gleiche Funktion wie $\gamma$ berechnet?

Wir haben es hier mit einem schwierigen Problem zu tun. Zum Beispiel ist nicht klar, daß MINIMUM CIRCUIT in NP liegt (wo ist der kurze Beweis?). Mit Hilfe von Satz 15.8 können wir das Problem auf der zweiten Stufe der Polynomialzeit-Hierarchie ansiedeln. Eine Eingabe $\gamma$ liegt genau dann in MINIMUM CIRCUIT, wenn es für jeden Schaltkreis $\gamma'$ kleiner als $\gamma$ eine Eingabe $\alpha$ gibt, auf der $\gamma$ und $\gamma'$ verschiedene Werte annehmen. Wir haben hier einen $\forall$-Quantor ("für jeden Schaltkreis $\gamma'$") gefolgt von einem $\exists$-Quantor ("es gibt eine Eingabe $\alpha$"), und der abschließende Test ("$\gamma$ und $\gamma'$ nehmen an $\alpha$ verschiedene Werte an") kann offensichtlich in Polynomialzeit ausgeführt werden. Also ist MINIMUM CIRCUIT $\in \Pi_2$. Wir haben damit natürlich nicht gezeigt, daß das Problem nicht schon zu einer niedrigeren Stufe in der Hierarchie gehört; aber so etwas ist zumindest unbekannt. Man bemerke, daß das MINIMUM CIRCUIT-Problem, obwohl scheinbar sehr schwierig (oberhalb von NP), durchaus in der Praxis wichtig sein kann—es ist kein abwegiges Problem.

Wie schon so oft ist es nicht bekannt, ob die Klassen in der Polynomialzeit-Hierarchie alle verschieden sind; man vermutet, daß es sich so verhält. Falls $\Sigma_k = \Pi_k$ für ein $k \in \mathbb{N}$ sein sollte (für $k = 1$ hieße das: NP = coNP), folgt aber, daß alle "darüberliegenden" Klassen auch mit $\Sigma_k$ übereinstimmen und also daß PH $= \Sigma_k$. Man sagt dazu auch: Die Polynomialzeit-Hierarchie "bricht auf der $k$ten Stufe zusammen".

**Satz** 15.9: Sei $k \in \mathbb{N}$. Falls $\Sigma_k = \Pi_k$, ist $\Sigma_l = \Pi_l = \Delta_l = \Sigma_k$ für alle $l > k$.

**Beweis**: Es reicht zu zeigen, daß $\Sigma_{k+1} = \Sigma_k$. Denn daraus folgt $\Pi_{k+1} = \text{co}\Sigma_{k+1} = \text{co}\Sigma_k = \text{co}\Pi_k = \Sigma_k$ und, da $\Sigma_k \subseteq \Delta_{k+1} \subseteq \Sigma_{k+1}$, auch $\Delta_{k+1} = \Sigma_k$, und man sieht leicht, daß sich die Hierarchie ab der $(k+1)$sten Stufe unverändert fortsetzt.

Sei also $L \in \Sigma_{k+1}$; wir wollen zeigen, daß $L \in \Sigma_k$. Nach Lemma 15.6 gibt es ein Polynom $p$ und eine Sprache $R$ in $\Pi_k$, so daß

$$L = \{x \mid \exists y : |y| \leq p(|x|) \text{ und } (x,y) \in R\}.$$

Aber nach Voraussetzung ist $R \in \Sigma_k$, woraus folgt (wieder Lemma 15.6), daß es ein Polynom $q$ und eine Sprache $S$ in $\Pi_{k-1}$ gibt, so daß $(x,y) \in R$ genau dann, wenn es einen String $z$ mit $|z| \leq q(|(x,y)|)$ und $((x,y),z) \in S$ gibt. Insgesamt haben wir also einen polynomiell großen Zeugen, nämlich $(y,z)$, der "in $\Pi_{k-1}$" überprüft werden kann (indem wir das Paar $(y,z)$ betrachten, kombinieren wir implizit zwei $\exists$-Quantoren. Aber das heißt (Lemma 15.6 ein letztes Mal), daß $L \in \Sigma_k$. ∎

Mit Satz 15.8 können wir leicht vollständige Sprachen in $\Sigma_k$ und $\Pi_k$ angeben. Für $k \in \mathbb{N}$ sei $\mathrm{QBF}_k$ (QBF = Quantified Boolean Formulas) die Sprache der Formeln der Form

$$\exists X_1 \forall X_2 \exists X_3 \cdots Q X_k : \phi,$$

wobei z.B. $\exists X_i$ die existentielle Quantifizierung möglicherweise mehrerer Boolescher Variablen repräsentiert: $\exists x_{i1} \exists x_{i2} \cdots \exists x_{i\ell_i}$, für ein $\ell_i \in \mathbb{N}_0$, und $\phi$ eine Boolesche Formel in allen Booleschen Variablen ist, so daß die Gesamtformel den Wert *true* hat.

**Beispiel:**

$$\exists x \forall y, z : (x \vee y) \wedge (x \vee \overline{z})$$

gehört zu $\mathrm{QBF}_2$, aber

$$\exists x \forall y, z : (x \wedge y) \vee (x \wedge \overline{z})$$

gehört nicht zu $\mathrm{QBF}_2$.

**Satz** 15.10: Für alle $k \in \mathbb{N}$ gilt: $\mathrm{QBF}_k$ ist $\Sigma_k$-vollständig.

**Beweis:** Zuerst müssen wir zeigen, daß $\mathrm{QBF}_k \in \Sigma_k$. Mit der Charakterisierung aus Satz 15.8 ist das leicht, denn wir müssen nur einsehen, daß die Formel $\phi$ bei gegebenen Variabelwerten in Polynomialzeit ausgewertet werden kann.

Sei jetzt $L$ eine beliebige Sprache in $\Sigma_k$. Nach Satz 15.8 kann $L$ in der Form

$$L = \{x \mid \exists y_1 \forall y_2 \exists y_3 \cdots Q y_k : (x, y_1, y_2, \ldots, y_k) \in R\}$$

geschrieben werden, wobei $y_1, \ldots, y_k$ alle über einer Menge der Form $\{y : |y| \leq p(|x|)\}$ quantifiziert sind, und $R \in \mathrm{P}$. Durch Auffüllen mit einem Sonderzeichen können wir erreichen, daß $(x, y_1, y_2, \ldots, y_k)$ nur dann in $R$ liegen kann, wenn $|y_i| = |x|^c$ für ein festes $c \in \mathbb{N}$ und für $i = 1, \ldots, k$.

Wir unterscheiden jetzt zwei Fälle danach, ob $k$ gerade oder ungerade ist. Sei zuerst $k$ ungerade, so daß die obige Quantorenreihe mit $\exists y_k$ endet. Aus dem Beweis von Satz 5.3 (Cook) wissen wir, daß es eine Boolesche Formel $\phi$ in den Variablen $x, y_1, \ldots, y_k$ (binär kodiert) sowie in weiteren Variablen $z$ gibt, die in Platz $O(\log |x|)$ konstruiert werden kann und die für jede feste Wahl von $x, y_1, \ldots, y_k$ genau dann erfüllbar ist (im Sinne der noch freien Variablen $z$), wenn $(x, y_1, \ldots, y_k) \in R$. Damit ist

$$L = \{x \mid \exists y_1 \forall y_2 \exists y_3 \cdots \forall y_{k-1} \exists y_k \exists z : \phi\}.$$

Da $y_k$ und $z$ zusammengelegt werden können und nur einen $\exists$-Quantor beanspruchen, haben wir $L$ auf $\text{QBF}_k$ reduziert: Die Reduktion berechnet $\phi$ mit den entsprechenden vorangestellten Quantoren.

Sei jetzt $k$ gerade. Dann endet die obige Quantorenreihe mit $\forall y_k$, so daß wir nicht genau analog vorgehen können; $y_k$ und $z$ könnten nicht mehr zusammengelegt werden, so daß wir einen zusätzlichen Quantor bräuchten. Statt dessen betrachten wir eine Formel $\psi$, die ähnlich wie $\phi$ aufgebaut, aber genau dann erfüllbar ist, wenn $(x, y_1, \ldots, y_k) \in \overline{R}$; wir benutzen hier, daß auch $\overline{R}$ in P liegt. $\psi$ ist also genau dann nicht erfüllbar, wenn $(x, y_1, \ldots, y_k) \in R$. Jetzt ist

$$\begin{aligned} L &= \{x \mid \exists y_1 \forall y_2 \exists y_3 \cdots \exists y_{k-1} \forall y_k : (x, y_1, y_2, \ldots, y_k) \in R\} \\ &= \{x \mid \exists y_1 \forall y_2 \exists y_3 \cdots \exists y_{k-1} \forall y_k \forall z : \neg\psi\}, \end{aligned}$$

und wir können die beiden letzten Quantoren zusammenfassen und wie oben argumentieren. ∎

Eine Sprache, die wie $\text{QBF}_k$ definiert ist, aber mit einem $\forall$-Quantor statt eines $\exists$-Quantors an erster Stelle, ist $\Pi_k$-vollständig. Was passiert wohl, wenn wir die Beschränkung in der Anzahl der Quantorenwechsel aufheben, also die Sprache QBF der quantifizierten Booleschen Formeln schlechthin betrachten? Wir können jetzt durch die Einführung von Dummy-Variablen davon ausgehen, daß sich jeder Quantor auf nur eine Boolesche Variable bezieht. Z.B. wird aus der Formel

$$\exists x \forall y, z : (x \vee y) \wedge (x \vee \overline{z})$$

dann die Formel

$$\exists x \forall y \exists w \forall z : (x \vee y) \wedge (x \vee \overline{z}).$$

Man könnte erwarten, daß QBF vollständig für die Sprache $\text{PH} = \bigcup_{k=0}^{\infty} \Sigma_k$ ist. Dem ist aber (vermutlich) ganz und gar nicht so:

**Satz** 15.11: Existiert eine PH-vollständige Sprache, bricht die Polynomialzeit-Hierarchie auf einer endlichen Stufe zusammen.

**Beweis**: Sei $L$ eine PH-vollständige Sprache. Da $L \in \text{PH}$, gibt es ein $k \in \mathbb{N}_0$, so daß $L \in \Sigma_k$. Außerdem ist $L' \leq L$ für jede Sprache $L' \in \text{PH}$. $L'$ wird also von einer Berechnung akzeptiert, die in Polynomialzeit läuft und zuerst eine Reduktion anwendet und danach

(falls $k \geq 1$) ein Orakel aus $\Sigma_{k-1}$ benutzen darf. Daraus sieht man, daß $L' \in \Sigma_k$ und daher $\mathrm{PH} = \Sigma_k$. ∎

QBF ist aber dennoch vollständig in einer uns schon vertrauten Komplexitätsklasse:

**Satz** 15.12: QBF ist PSPACE-vollständig.

**Beweis:** Zuerst zeigen wir, daß QBF $\in$ PSPACE, also daß quantifizierte Boolesche Formeln in polynomiellem Platz ausgewertet werden können. Aber das ist leicht, denn die offensichtliche rekursive Prozedur benutzt sowieso nur polynomiell viel Platz. Sie geht so vor: Bei Eingabe z.B.

$$\exists x_1 \forall x_2 \exists x_3 \cdots Q x_m : \phi$$

bestimmt sie rekursiv die Werte der Formeln $\forall x_2 \exists x_3 \cdots Q x_m : \phi_{|x_1=0}$ und $\forall x_2 \exists x_3 \cdots Q x_m : \phi_{|x_1=1}$ und gibt das Oder der beiden Werte zurück. $\phi_{|x_1=b}$, für $b \in \{0,1\}$, bedeutet dabei die Boolesche Formel, die dadurch aus $\phi$ hervorgeht, daß alle Vorkommen von $x_1$ durch die Konstante $b$ ersetzt werden.

Wir zeigen jetzt, daß $L \leq$ QBF für jede Sprache $L \in$ PSPACE. Sei also $M$ eine TM, die $L$ in polynomiellem Platz akzeptiert, und sei genauer $p : \mathbb{N}_0 \to \mathbb{N}$ ein Polynom mit der folgenden Eigenschaft: Für alle Eingaben $x$ mit $|x| = n$ können alle von $Init_M(x)$ aus erreichbaren Konfigurationen von $M$ als Bitvektoren der Länge $p(n)$ kodiert werden. Wenn wir im folgenden über Konfigurationen quantifizieren, meinen wir damit in Wirklichkeit eine Quantifizierung über $p(n)$ entsprechende Boolesche Variablen.

Wir müssen die Aussage "$x \in L$" als quantifizierte Boolesche Formel ausdrücken. Ansatzweise gehen wir so vor:

$$x \in L \iff \exists K_1 \exists K_2 : (K_1 = Init_M(x) \wedge K_2 \text{ akzeptiert} \wedge K_1 \overset{*}{\vdash} K_2).$$

Die Aussagen "$K_1 = Init_M(x)$" und "$K_2$ akzeptiert" lassen sich leicht als Boolesche Formeln ausdrücken; wir brauchen nicht einmal Quantoren. Die Aussage "$K_1 \overset{*}{\vdash} K_2$" ist schwieriger. Zunächst können wir sie durch "$K_1 \overset{\leq 2^{p(n)}}{\vdash} K_2$" ersetzen, denn keine terminierende Berechnung kann mehr als $2^{p(n)}$ Schritte lang sein. Wir wollen jetzt Aussagen "$K_1 \overset{\leq 2^j}{\vdash} K_2$" konstruieren, für $j = 0, 1, \ldots, p(n)$. Für $j = 0$ ist dies kein Problem. Wie sieht die Rekursion aus?

Die erste Antwort, die einem einfällt, besteht darin, "$K_1 \overset{\leq 2^j}{\vdash} K_2$" so zu schreiben:

$$\exists K : (K_1 \overset{\leq 2^{j-1}}{\vdash} K) \wedge (K \overset{\leq 2^{j-1}}{\vdash} K_2).$$

Das ist aber nicht brauchbar, denn die Rekursionstiefe ist $p(n)$, und auf jeder Rekursionstiefe verdoppelt sich die Formelgröße, was zu einer exponentiell großen Formel führen würde (die wir natürlich nicht mit logarithmischem Arbeitsplatz generieren können). Wir müssen schlauer vorgehen, um die Formel für $\overset{\leq 2^{j-1}}{\vdash}$ nur einmal auftauchen zu lassen. Das kann man so erreichen:

$$\exists K \forall K' \forall K'' : [((K' = K_1) \wedge (K'' = K)) \vee ((K' = K) \wedge (K'' = K_2))] \Rightarrow K' \overset{\leq 2^{j-1}}{\vdash} K''.$$

In Worten: Es gibt eine mittlere Konfiguration $K$ (entspricht dem $K$ vom ersten, gescheiterten Versuch), die von $K_1$ aus in $\leq 2^{j-1}$ Schritten zu erreichen ist und von der aus $K_2$ in $\leq 2^{j-1}$ Schritten erreicht werden kann; $K'$ und $K''$ sind nur "Platzhalter".

Damit können wir die Aussage "$K_1 \overset{\leq 2^{p(n)}}{\vdash} K_2$" mit einer quantifizierten Booleschen Formel ausdrücken, deren Länge polynomiell in $p(n)$ ist und die auch ohne Schwierigkeit mit $O(\log n)$ Arbeitsplatz niedergeschrieben werden kann. ∎

Priv.-Doz. Torben Hagerup
Jordan Gergov
Universität des Saarlandes
Fachbereich 14, Informatik


# Komplexitätstheorie (WS 95/96)

17.10.95

## Übung 1

1

**Aufgabe 1:** Versucht eine Turing-Maschine (TM), einen Kopf nach links zu bewegen, obwohl sich der Kopf schon auf dem linkesten Feld seines Bandes befindet, so nehmen wir an, daß der Kopf einfach stehenbleibt.

(a) Zeigen Sie, daß eine TM, obwohl sie das linke Bandende nicht explizit abfragen kann, dennoch in konstanter Zeit überprüfen kann, ob ein gegebener Kopf am linken Bandende steht (20 Punkte). *Hinweis:* Retten Sie zuerst die Umgebung des Kopfes in den endlichen Zustand hinüber, und erstellen Sie dann eine geeignete "Test-Umgebung".

(b) Beschreiben Sie genauer, was unter Teil (a) eigentlich zu zeigen ist (20 Punkte).

**Aufgabe 2:** Herr F. Alsch beweist folgendermaßen, daß NP = coNP:

*Sei $L$ eine Sprache in NP und sei $M$ eine nicht-deterministische Turing-Maschine (NTM), die $L$ entscheidet. $M$ läuft also in Polynomialzeit und geht bei Eingabe $x$ in den akzeptierenden Endzustand* `accept`*, falls $x \in L$, und in den verwerfenden Endzustand* `reject`*, falls $x \notin L$. Erstelle jetzt eine NTM $M'$, die sich genau wie $M$ verhält, aber bei der die Rollen von* `accept` *und* `reject` *vertauscht sind (z.B. ist also* `reject` *der akzeptierende Zustand). $M'$ akzeptiert $\overline{L}$ in Polynomialzeit; also ist $\overline{L} \in NP$ und daher $L \in coNP$. Symmetrisch dazu folgt aus $L \in coNP$, daß $L \in NP$.*

Begründen Sie, warum, Herrn Alsch zum Trotz, NP $\stackrel{?}{=}$ coNP immer noch ein ungelöstes Problem ist (30 Punkte).

**Aufgabe 3:** Zeigen Sie, daß es (nicht-uniforme) Schaltkreisfamilien gibt, die nicht berechenbare Funktionen von $\{0,1\}^*$ nach $\{0,1\}$ realisieren. Woher kommt das? (30 Punkte).

**Abgabetermin:** 23.10.95.

Priv.-Doz. Torben Hagerup
Jordan Gergov
Universität des Saarlandes
Fachbereich 14, Informatik

# Komplexitätstheorie (WS 95/96)

30.10.95

## Übung 3

3

Auf diesem Übungsblatt können Sie wählen, ob Sie *entweder* die Aufgabe A *oder* die Aufgaben B.1–B.4 beantworten.

**Aufgabe A:** Sei $\Sigma$ ein Alphabet und sei $x \in \Sigma^*$. Das *Spiegelbild* $\overline{x}$ von $x$ ist rekursiv wie folgt definiert:

$$\overline{\epsilon} = \epsilon \quad (\epsilon \text{ ist das leere Wort})$$
$$\overline{xa} = a\overline{x}, \quad \text{für } a \in \Sigma \text{ und } x \in \Sigma^*.$$

Definiere jetzt PALINDROME als die Menge

$$\text{PALINDROME} = \{x \in \{a,b\}^* : x = \overline{x}\}$$

der Palindrome über $\{a,b\}$.

(a) Zeigen Sie, daß eine 2-Band-TM PALINDROME in Linearzeit $O(n)$ erkennen kann (10 Punkte).

(b) Zeigen Sie, daß eine 1-Band-TM PALINDROME in quadratischer Zeit $O(n^2)$ erkennen kann (10 Punkte).

Im restlichen Teil dieser Aufgabe soll gezeigt werden, daß eine 1-Band-TM quadratische Zeit braucht, um PALINDROME zu erkennen. Sei also $M$ eine 1-Band-TM, die PALINDROME erkennt. Für $x \in \Sigma^*$ sei $T(x)$ die Laufzeit von $M$ bei Eingabe $x$, und für $n \in \mathbb{N}_0$ sei $T'(n)$ die worst-case-Laufzeit von $M$ bei Eingaben der Größe $n$. Unser Ziel ist zu zeigen, daß $T'(n) = o(n^2)$ nicht gelten kann.

(c) Zeigen Sie, daß wir o.B.d.A. davon ausgehen können, daß die Maschine $M$ ihren Kopf in jedem Schritt bewegt und daß sie mit dem Kopf auf dem linkesten Feld hält (10 Punkte).

**Definition:** Sei $x \in \{a,b\}^*$ und $i \in \mathbb{N}$. Die $i$'te *Crossing-Sequenz* $CS(x,i)$ von $M$ bei Eingabe $x$ ist die (zeitlich geordnete) Folge der Zustände, die die Maschine $M$ in denjenigen Schritten erreicht, in denen ihr Kopf die Grenze zwischen den Zellen $i$ und $i+1$ überschreitet.

(d) Zeigen Sie, daß $T(x) = \sum_{i=1}^{\infty} |CS(x,i)|$ (10 Punkte).

(e) Zeigen Sie: Erfüllen $x_1, x_2, y_1, y_2 \in \Sigma^* \backslash \{\epsilon\}$ die Bedingung

$$CS(x_1x_2, |x_1|) = CS(y_1y_2, |y_1|),$$

dann gilt

$$M \text{ akzeptiert } x_1x_2 \iff M \text{ akzeptiert } x_1y_2 \qquad (15 \text{ Punkte}).$$

(f) Zeigen Sie, daß es zu jedem $r \in \mathbb{N}$ eine Zahl $c > 0$ mit der folgenden Eigenschaft gibt: Sei $A$ eine endliche nicht-leere Menge, und sei $l : A \to \mathbb{N}_0$ eine Funktion, so daß

$$|\{a \in A : l(a) = m\}| \leq r^m$$

für alle $m \in \mathbb{N}_0$. Dann ist $\sum_{a \in A} l(a) \geq c|A| \log_2 |A|$ (10 Punkte).

Sei jetzt $m \in \mathbb{N}$ und $X = \{a, b\}^m$. Für $x \in X$ sei $f(x) = xb^{2m}\overline{x} \in$ PALINDROME.

(g) Zeigen Sie unter Benutzung von (e): Für $x_1, x_2 \in X$ und $m \leq i \leq 3m$ gilt

$$x_1 \neq x_2 \Rightarrow CS(f(x_1), i) \neq CS(f(x_2), i) \qquad \text{(10 Punkte)}.$$

(h) Benutzen Sie (f) mit $A = \{CS(f(x), i) : x \in X\}$ und $l(a) = |a|$ sowie (d) und (g) um zu zeigen, daß es eine Konstante $c > 0$ gibt (die zwar von $M$, nicht aber von $m$ abhängt), so daß

$$\sum_{x \in X} T(f(x)) \geq cm^2 2^m \qquad \text{(15 Punkte)}.$$

(i) Zeigen Sie, daß es ein $x \in X$ mit $T(f(x)) \geq \frac{c}{16}|f(x)|^2$ gibt, und folgern Sie, daß $T'(n) = o(n^2)$ unmöglich ist (10 Punkte).

**Aufgabe B.1:** Diese Aufgabe soll einen kleinen Fehler aus der Vorlesung am 24.10. korrigieren. Es ging dabei um den

**Satz** 3.1: Für jede konstruierbare Funktion $f : \mathbb{N}_0 \to \mathbb{N}$ gilt: $f(n) = O(1)$ oder $f(n) = \Omega(\log n)$.

Unsere Strategie war zu zeigen, daß, wenn $f(n) = \Omega(\log n)$ nicht gilt, dann $f(n) = O(1)$ sein muß. Dazu haben wir angenommen, daß $f(n) = o(\log n)$, und gefolgert, daß in der Tat $f(n) = O(1)$. Nun ist $f(n) = o(\log n)$ aber nicht das Gegenteil von $f(n) = \Omega(\log n)$! Zur Erinnerung:

$$f(n) = \Omega(g(n)) \iff \exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : f(n) \geq cg(n)$$
$$f(n) = o(g(n)) \iff \forall c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : f(n) < cg(n).$$

Sei $f : \mathbb{N}_0 \to \mathbb{N}$ wie folgt definiert:

$$f(n) = \begin{cases} 1, & \text{falls } n < 4; \\ 2^{2^i}, & \text{falls } 2^{2^{2^i}} \leq n < 2^{2^{2^{i+1}}}, \quad i \in \mathbb{N}_0. \end{cases}$$

(a) Zeigen Sie, daß $\neg[f(n) = o(\log n)]$ (10 Punkte).
(b) Zeigen Sie, daß $\neg[f(n) = \Omega(\log n)]$ (10 Punkte).
(c) Zeigen Sie, daß in der Vorlesung, obwohl $f(n) = o(\log n)$ vorausgesetzt wurde, in Wirklichkeit nur $\neg[f(n) = \Omega(\log n)]$ benutzt wurde. Bis auf dieses kleine Problem war der Beweis von Satz 3.1 also in Ordnung (10 Punkte).

**Aufgabe B.2:** Zeigen Sie: Sind $f$ und $g$ konstruierbare Funktionen, dann auch $f + g$, $f \cdot g$ und $f^g$. Wie steht es mit $f - g$ und mit $f \circ g$? (25 Punkte).

**Aufgabe B.3:** Zeigen Sie, daß die Funktionen $n!$ und $\lfloor \sqrt{n} \rfloor$ konstruierbar sind (25 Punkte).

**Aufgabe B.4:** Sei $M$ eine deterministische Turing-Maschine, die auf jeder Eingabe $x$ in $L(M)$ nach höchstens $c|x|$ Schritten hält, für eine Konstante $c \in \mathbb{N}$, während sie auf Eingaben, die nicht zu $L(M)$ gehören, möglicherweise nicht hält. Beschreiben Sie eine Turing-Maschine $M'$ mit $L(M') = L(M)$, die auf jeder Eingabe nach endlich vielen Schritten anhält (20 Punkte).

**Abgabetermin:** 6.11.95.

Priv.-Doz. Torben Hagerup
Jordan Gergov
Universität des Saarlandes
Fachbereich 14, Informatik


# Komplexitätstheorie (WS 95/96)

6.11.95

## Übung 4

4

**Aufgabe 1:** Sei $f : \mathbb{N}_0 \to \mathbb{N}$ die folgende Funktion: $f(0) = 1$, und $f(n) = \lceil \log_2(p+1) \rceil$ für alle $n \in \mathbb{N}$, wobei $p$ die kleinste Primzahl ist, die $n$ nicht teilt.

(a) Zeigen Sie, daß $f$ unendlich viele verschiedene Werte annimmt (10 Punkte).

(b) Zeigen Sie, daß $f(n) = O(\log\log n)$. Sie dürfen dabei ohne Beweis den folgenden berühmten Satz über die Verteilung der Primzahlen benutzen: $\lim_{n\to\infty} \pi(n) \ln n/n = 1$, wobei $\pi(n)$ die Anzahl der Primzahlen $\leq n$ ist (10 Punkte). (Für einen Beweis der Aussage über die Verteilung der Primzahlen s. z.B. G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers* (5th Ed.), Oxford University Press, Oxford, 1979, Theorem 6.)

(c) Zeigen Sie, daß die Funktion, die jeden String der Länge $n \in \mathbb{N}_0$ auf $I^{f(n)}$ abbildet, wobei $I$ ein Alphabetsymbol ist, in $O(\log\log n)$ Platz berechnet werden kann (wobei der Platzverbrauch auf dem read-only Eingabeband wie üblich ignoriert wird) (10 Punkte).

(d) Erklären Sie, warum die Ergebnisse aus (a) und (c) keinen Widerspruch zu Satz 3.1 aus der Vorlesung darstellen, in dem behauptet wird, daß für jede konstruierbare Funktion $f$ entweder $f(n) = O(1)$ oder $f(n) = \Omega(\log n)$ (10 Punkte).

**Aufgabe 2:** Sei $f : \mathbb{N}_0 \to \mathbb{N}$ eine konstruierbare Funktion mit $f(n) \geq \log n$ für alle $n \in \mathbb{N}$. Welche der Komplexitätsklassen TIME($f$), SPACE($f$), NTIME($f$) und NSPACE($f$) sind für alle solche $f$ gegenüber Bildung von Durchschnitt und Vereinigung abgeschlossen? (Eine Klasse $C$ von Sprachen heißt gegenüber Bildung von Durchschnitt (bzw. Vereinigung) abgeschlossen, falls aus $L_1 \in C$ und $L_2 \in C$ die Behauptung $L_1 \cap L_2 \in C$ (bzw. $L_1 \cup L_2 \in C$) folgt.) Begründen Sie Ihre Antwort. Was wissen Sie über den Abschluß obiger Klassen gegenüber Komplementbildung? (30 Punkte).

**Aufgabe 3:** Wir haben in der Vorlesung zwei deterministische Simulationen von platzbeschränkten NTMs gesehen, eine zeiteffiziente (Satz 3.2(d)) und eine platzeffiziente (Satz 3.3). Wieviel Platz braucht die Simulation aus Satz 3.2(d), und wieviel Zeit braucht die Simulation aus Satz 3.3? Eine genaue Antwort wird nicht erwartet; Ihre Aufgabe besteht darin, eine vernünftige obere Schranke anzugeben und zu begründen (30 Punkte).

**Abgabetermin:** 13.11.95.

Priv.-Doz. Torben Hagerup
Jordan Gergov
Universität des Saarlandes
Fachbereich 14, Informatik


# Komplexitätstheorie (WS 95/96)

13.11.95 **Übung 5** 5

**Aufgabe 1:** Zeigen Sie den Hierarchiesatz für deterministische Zeit (Satz 4.3): Seien $f, g : \mathbb{N}_0 \to \mathbb{N}$, sei $g$ konstruierbar und $g(n) \geq n$ für alle $n \in \mathbb{N}_0$, und es gelte $(f(n))^2/g(n) \to 0$ für $n \to \infty$. Dann ist $\mathrm{TIME}(f) \subset \mathrm{TIME}(g)$ (40 Punkte).
*Hinweis:* Schauen Sie sich an, wie wir das analoge Ergebnis für Platz bewiesen haben, und erinnern Sie sich auch an Satz 2.1.

**Aufgabe 2:** Zeigen Sie, ohne die Sätze 3.6 und 4.2 zu benutzen, daß

$$\mathrm{NSPACE}(\log n) \subset \mathrm{NSPACE}((\log n)^3)$$

(30 Punkte).
*Hinweis:* Was haben wir sonst an nützlichen Sätzen bewiesen?

**Aufgabe 3:** In dieser Aufgabe wollen wir das folgende zeigen:

**Satz:** Sei $f : \mathbb{N}_0 \to \mathbb{N}$ eine Funktion mit $f(n) = o(\log\log n)$. Dann ist $\mathrm{SPACE}(f) = \mathrm{SPACE}(1)$.

Es gibt also gar nichts zwischen den Sprachen, die in $O(\log\log n)$ Platz erkannt werden können, und denen, die in konstantem Platz erkannt werden können (letztere sind übrigens genau die regulären Sprachen). Vergleichen Sie dazu auch Satz 3.1 und Aufgabe 3.B.1.

Sei jetzt also $f$ wie im Satz, sei $L$ eine Sprache in $\mathrm{SPACE}(f)$ und sei $M$ eine TM, die $L$ in Platz $O(f)$ erkennt. Wir nehmen an, daß $L \notin \mathrm{SPACE}(1)$; insbesondere ist $L$ unendlich.

Eine Halbkonfiguration von $M$ auf der Eingabe $x$ wurde in der Vorlesung als eine Konfiguration von $M$ definiert, bei der das Eingabeband außer acht gelassen wird. Weiter können wir o.B.d.A. davon ausgehen, daß die Maschine $M$ ihren Kopf auf dem Eingabeband in jedem Schritt bewegt (Aufgabe 3.A(c)). Eine *erweiterte Crossing-Sequenz* $ECS(x, i)$ von $M$ bei Eingabe $x$, wobei $1 \leq i \leq |x|$, ist die (zeitlich geordnete) Folge der Halbkonfigurationen, die die Maschine $M$ in denjenigen Schritten erreicht, in denen ihr Eingabekopf die Grenze zwischen den Feldern $i$ und $i+1$ überschreitet (vgl. Crossing-Sequenz einer 1-Band-TM, Aufgabe 3.A).

(a) Seien $x \in L$ und $1 \leq i \leq |x|$. Beweisen Sie, daß eine Halbkonfiguration von $M$ höchstens zweimal in $ECS(x, i)$ vorkommen kann (10 Punkte).

(b) Zeigen Sie unter Benutzung von (a), daß ein $n_0 \in \mathbb{N}$ mit der folgenden Eigenschaft existiert: Die Anzahl der erweiterten Crossing-Sequenzen von $M$ auf jeder Eingabe $x \in L$ mit $|x| = n \geq n_0$ ist kleiner als $n/2$. Schließen Sie daraus für jedes $x \in L$ mit $|x| \geq n_0$ auf die Existenz von $i_1$, $i_2$ und $i_3$ mit $1 \leq i_1 < i_2 < i_3 \leq n$ und $ECS(x, i_1) = ECS(x, i_2) = ECS(x, i_3)$ (10 Punkte).

(c) Zeigen Sie die Existenz von $x \in L$ mit $|x| = n \geq n_0$ und $s \in \mathbb{N}$ mit der folgenden Eigenschaft: Die Maschine $M$ besucht bei Eingabe $x$ genau $s$ Felder auf ihren Arbeitsbändern, während sie bei allen Eingaben $y \in L$ mit $|y| < n$ weniger als $s$ Felder besucht. Benutzen Sie dann Teil (b), um eine Eingabe $y \in L$ mit $|y| < n$ zu konstruieren, die zu einem Widerspruch führt, weil $M$ auch bei Eingabe $y$ Platz genau $s$ verbraucht (10 Punkte).

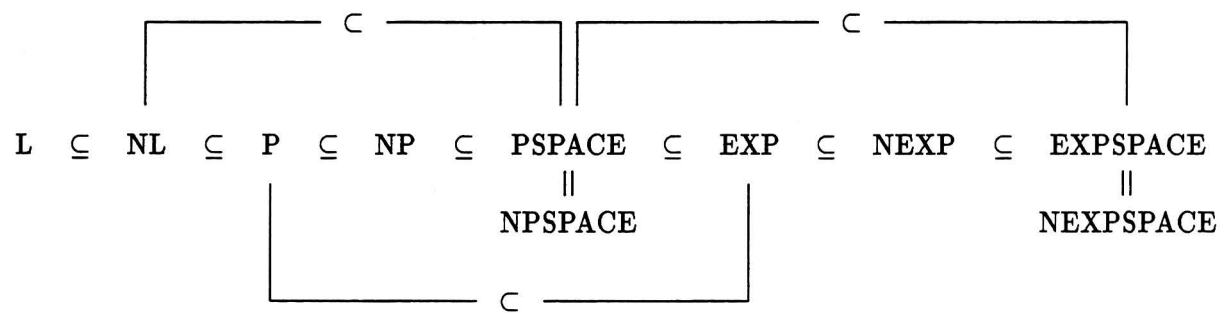
**Abgabetermin:** 20.11.95.

Priv.-Doz. Torben Hagerup
Jordan Gergov
Universität des Saarlandes
Fachbereich 14, Informatik

# Komplexitätstheorie (WS 95/96)

20.11.95 **Übung 6** 6

**Aufgabe 1:** Zeigen Sie alle Relationen zwischen Komplexitätsklassen, die im folgenden Diagramm angegeben sind (40 Punkte):




**Aufgabe 2:** Beweisen Sie, daß jede Funktion $f : \{0,1\}^n \to \{0,1\}$ in konjunktiver Normalform mit $O(n \cdot 2^n)$ Literalen geschrieben werden kann (30 Punkte).
*Hinweis:* Vielleicht hilft es Ihnen, die Aussage zuerst für die (analog definierte) disjunktive Normalform zu zeigen.

**Aufgabe 3:** Der Beweis von Satz 3.3 (Savitch) kann als eine Aussage über ein platzeffizientes Verfahren zum Suchen in implizit gegebenen gerichteten Graphen verstanden werden. "Implizit gegeben" bedeutet hier, daß der Graph nicht explizit, z.B. in Form einer Adjazenzmatrix, vorliegt, sondern daß man lediglich effizient feststellen kann, ob eine gegebene Kante $(i, j)$ vorhanden ist. Programmieren Sie das Verfahren von Savitch in Ihrer Lieblingsprogrammiersprache, wobei die Knotenmenge als $V = \{1, \ldots, n\}$ genommen wird ($n \in \mathbb{N}$) und das Vorhandensein einer Kante $(i, j)$ durch einen Funktionsaufruf *Edge*$(i, j)$ abgefragt wird (30 Punkte).

Für eine vollständige Beantwortung genügt es, die Subroutine *Savitch* anzugeben, wobei Sie insbesondere die Funktion *Edge* undefiniert sein lassen dürfen. Sie können aber auch, was sicherlich mehr Spaß macht, ein komplettes lauffähiges Programm erstellen, bei dem *Edge* z.B. durch Nachschlagen in einer Adjazenzmatrix $A[1..n, 1..n]$ realisiert wird, die anfangs von einer Datei eingelesen wird (dadurch geht natürlich die Platzeffizienz flöten, aber man kann das Prinzip noch erkennen). In wie großen Graphen können Sie noch suchen?

**Abgabetermin:** 27.11.95.

Priv.-Doz. Torben Hagerup
Jordan Gergov
Universität des Saarlandes
Fachbereich 14, Informatik


# Komplexitätstheorie (WS 95/96)

27.11.95 **Übung 7** 7

**Aufgabe 1:** Zeigen Sie, daß 3SAT NP-vollständig ist (40 Punkte).
*Hinweis:* Reduzieren Sie SAT auf 3SAT. Ersetzen Sie dabei jede "zu lange" Klausel $(x_1 \vee x_2 \vee \cdots \vee x_k)$, mit $k \geq 4$, durch

$$(x_1 \vee y_1) \wedge (x_2 \vee \overline{y_1} \vee y_2) \wedge (x_3 \vee \overline{y_2} \vee y_3) \wedge \cdots \wedge (x_{k-1} \vee \overline{y_{k-2}} \vee y_{k-1}) \wedge (x_k \vee \overline{y_{k-1}}),$$

wobei $y_1, \ldots, y_{k-1}$ neue Variable sind.

**Aufgabe 2:** Zeigen Sie: Unter der Annahme, daß SAT $\in$ P, kann zu jeder erfüllbaren Booleschen Formel in CNF in Polynomialzeit eine erfüllende Belegung der darin vorkommenden Variablen gefunden werden (30 Punkte).

**Aufgabe 3:** Zeigen Sie, daß 2SAT $\in$ P. Gegeben ist also eine Boolesche Formel $F$ in 2-CNF, wie z.B.

$$(x_1 \vee \overline{x_2}) \wedge (\overline{x_2} \vee \overline{x_3}) \wedge (x_4) \wedge (x_3 \vee x_4) \wedge (x_4 \vee \overline{x_1}),$$

und Sie müssen in Polynomialzeit herausfinden, ob sie erfüllbar ist.

Ersetzen Sie zuerst jede Klausel, die nur ein Literal $\alpha$ enthält, durch die äquivalente Klausel $(\alpha \vee \alpha)$. Erstellen Sie dann einen gerichteten Graphen $G$ mit einem Knoten für jede vorkommende Variable $x_i$ sowie einem Knoten für ihre Negation $\overline{x_i}$. Für jede Klausel der Form $(\alpha_1 \vee \alpha_2)$, wobei $\alpha_1$ und $\alpha_2$ Literale sind, enthält $G$ die gerichteten Kanten $(\overline{\alpha_1}, \alpha_2)$ und $(\overline{\alpha_2}, \alpha_1)$.

(a) Zeigen Sie: Enthält $G$ einen Pfad von einem Literal $\alpha$ zu seiner Negation $\overline{\alpha}$ und auch einen Pfad von $\overline{\alpha}$ nach $\alpha$, dann ist $F$ nicht erfüllbar (10 Punkte).

(b) Zeigen Sie die Umkehrung: Enthält $G$ kein solches Paar von Pfaden, dann ist $F$ erfüllbar (10 Punkte).
*Hinweis:* Wählen Sie ein Literal $\alpha$, so daß es keinen Pfad von $\alpha$ nach $\overline{\alpha}$ gibt, geben Sie $\alpha$ den Wert *true*, und führen Sie alle "Konsequenzen" dieser Zuweisung durch. Dann wiederholen Sie alles von vorne. Natürlich müssen Sie nachweisen, daß Ihr Verfahren korrekt ist.

(c) Zeigen Sie, daß die Bedingung aus den Teilen (a) und (b) in Polynomialzeit überprüft werden kann (10 Punkte).
*Hinweis:* Ich sehe zwei verschiedene Lösungswege: (1) Berechnen Sie die starken Zusammenhangskomponenten von $G$; (2) Zeigen Sie, daß 2SAT $\in$ NL, und folgern Sie daraus, daß 2SAT $\in$ P.

**Abgabetermin:** 4.12.95.

Priv.-Doz. Torben Hagerup
Jordan Gergov
Universität des Saarlandes
Fachbereich 14, Informatik


# Komplexitätstheorie (WS 95/96)

4.12.95 **Übung 8** 8

**Aufgabe 1:** Zeigen Sie, daß das Problem UNDIRECTED HAMILTON CYCLE NP-vollständig ist (30 Punkte).
*Hinweis:* Reduzieren Sie DIRECTED HAMILTON CYCLE auf UNDIRECTED HAMILTON CYCLE. Ersetzen Sie dabei jeden Knoten durch einen Pfad von drei Knoten.

**Aufgabe 2:** Ein *Hamilton-Pfad* in einem gerichteten oder ungerichteten Graphen $G$ ist ein einfacher Pfad in $G$ (also ohne Knotenwiederholungen), der sämtliche Knoten enthält. Zeigen Sie, daß die Probleme DIRECTED HAMILTON PATH und UNDIRECTED HAMILTON PATH, in der offensichtlichen Weise definiert, NP-vollständig sind (40 Punkte).
*Hinweis:* Für den gerichteten Fall: Schauen Sie sich unsere Reduktion von SAT auf DIRECTED HAMILTON CYCLE noch einmal an.

Nach diesen Aufgaben brauchen Sie sich also nicht daran zu erinnern, ob unser NP-vollständiges Hamilton-Problem für gerichtete oder ungerichtete Graphen und für Kreise oder Pfade formuliert war—alle 4 Varianten sind NP-vollständig.

**Aufgabe 3:** Zeigen Sie im Detail, daß unsere Reduktion von INDEPENDENT SET auf CLIQUE in logarithmischem Platz berechnet werden kann (beschreiben Sie Ihr Verfahren nicht als TM-Übergangsfunktion, sondern in einer informellen Algorithmensprache, ähnlich wie in der Vorlesung). Legen Sie dazu eine Repräsentation von ungerichteten Graphen fest (30 Punkte).

**Abgabetermin:** 11.12.95.

Priv.-Doz. Torben Hagerup
Jordan Gergov
Universität des Saarlandes
Fachbereich 14, Informatik


# Komplexitätstheorie (WS 95/96)

11.12.95 **Übung 9** 9

**Aufgabe 1:** Betrachten Sie das Problem

COLORING:

Eingabe: Ein ungerichteter Graph $G$ und eine Zahl $k \in \mathbb{N}$.

Frage: Gibt es eine Färbung von $G$ mit $k$ Farben?

Zeigen Sie, daß COLORING NP-vollständig ist (10 Punkte).

**Aufgabe 2:** Zeigen Sie, daß 4-COLORING NP-vollständig ist. Wie steht es mit 37-COLORING? (20 Punkte)
*Hinweis:* Reduzieren Sie 3-COLORING auf diese Probleme.

**Aufgabe 3:** Beweisen Sie die Sätze 7.1–7.4 aus der Vorlesung (15 + 15 + 20 + 20 Punkte).

**Satz** 7.1: $P \subseteq NP \cap coNP$.

**Satz** 7.2: Falls $P = NP$ oder $P = coNP$, dann ist $P = NP = coNP$.

**Satz** 7.3: Sei $L$ eine NP-vollständige Sprache. Dann gilt $L \in coNP \Leftrightarrow NP = coNP$.

**Satz** 7.4: $L$ ist NP-vollständig $\Leftrightarrow \overline{L}$ ist coNP-vollständig.

**Abgabetermin:** 18.12.95.

Priv.-Doz. Torben Hagerup
Jordan Gergov
Universität des Saarlandes
Fachbereich 14, Informatik


# Komplexitätstheorie (WS 95/96)

18.12.95 **Übung 10** 10

Die Aufgaben auf diesem Übungsblatt beschäftigen sich alle mit Rabins Primalitätstest (Kapitel 8).

**Aufgabe 1:** Sei $m = rs$, wobei $r$ und $s$ teilerfremde natürliche Zahlen sind, und sei $b \in \mathbb{Z}_m^*$. Zeigen Sie, daß $b \in \mathbb{Z}_r^*$ und $b \in \mathbb{Z}_s^*$ und daß $\mathrm{ord}_m(b)$ gleich dem kleinsten gemeinsamen Vielfachen von $\mathrm{ord}_r(b)$ und $\mathrm{ord}_s(b)$ ist (30 Punkte).
*Hinweis:* Repräsentieren Sie $b$ zusätzlich entsprechend dem chinesischen Restsatz und potenzieren Sie beide Repräsentationen gleichzeitig.

**Aufgabe 2:** Zeigen Sie, daß das im Beweis von Lemma 8.9 mit Hilfe des chinesischen Restsatzes festgelegte Element $a$ tatsächlich in $\mathbb{Z}_m^*$ liegt (20 Punkte).

**Aufgabe 3:** Zeigen Sie, daß die Berechnung und Überprüfung von $(a, m)$ im Algorithmus weggelassen werden kann: Ist $(a, m) \neq 1$, wird der Algorithmus auch ohne die entsprechende Zeile $m$ als zusammengesetzt erkennen (20 Punkte).

**Aufgabe 4:** Zeigen Sie, daß der Test im Algorithmus, ob $m$ eine nichttriviale Potenz ist, in Polynomialzeit durchgeführt werden kann. Zu zeigen ist also, daß mit $(\log m)^{O(1)}$ arithmetischen Operationen überprüft werden kann, ob $m$ von der Form $x^y$ ist, wobei $x, y \geq 2$ ganze Zahlen sind (30 Punkte).
*Hinweis:* Binäre Suche könnte sich als nützlich erweisen.

**Abgabetermin:** 8.1.96.

Priv.-Doz. Torben Hagerup
Jordan Gergov
Universität des Saarlandes
Fachbereich 14, Informatik

# Komplexitätstheorie (WS 95/96)

8.1.96 **Übung 11** 11

Auf diesem Übungsblatt geht es weiterhin um Rabins Primalitätstest (Kapitel 8).

**Aufgabe 1:** In dieser Aufgabe wollen wir zeigen, daß Aufgabe 10.4 überflüssig war, da der Test, ob $m$ eine nichttriviale Potenz ist, ohne Schaden im Algorithmus weggelassen werden kann. Mit anderen Worten wollen wir zeigen, daß nichttriviale Potenzen vom so modifizierten Algorithmus immer noch mit Wahrscheinlichkeit mindestens $1/2$ als zusammengesetzt erkannt werden. Sei also $m = p^q$, wobei $p, q \geq 2$ ganze Zahlen sind.

(a) Wir wollen im folgenden annehmen, daß $p$ eine Primzahl ist. Zeigen Sie, daß diese Annahme zulässig ist (10 Punkte).
*Hinweis:* Sonst tut es der alte Beweis.

(b) Zeigen Sie, daß $\mathrm{ord}_m(a) \mid p^{q-1}(p-1)$ für alle $a \in \mathbb{Z}_m^*$ (10 Punkte).
*Hinweis:* Was ist $\phi(m)$?

(c) Begründen Sie, daß $\mathrm{ord}_m(a) \mid (p^q - 1)$ für alle Nicht-Zeugen $a \in \mathbb{Z}_m^*$ (d.h. für alle $a \in \mathbb{Z}_m^*$, mit denen als Zufallszahl der Algorithmus $m$ nicht als zusammengesetzt erkennt) (10 Punkte).

(d) Folgern Sie aus (b) und (c), daß $\mathrm{ord}_m(a) \mid (p-1)$ für alle Nicht-Zeugen $a \in \mathbb{Z}_m^*$ (10 Punkte).

(e) Zeigen Sie, daß $G = \{a \in \mathbb{Z}_m^* \mid a^{p-1} \equiv 1 \pmod{m}\}$ eine Untergruppe von $\mathbb{Z}_m^*$ ist (10 Punkte).

(f) Begründen Sie, daß es jetzt reicht, $G \neq \mathbb{Z}_m^*$ zu zeigen (10 Punkte).

(g) Überzeugen Sie sich davon, daß $a^{p-1} \equiv 1 \pmod{p^2}$ für alle $a \in G$ (10 Punkte).

(h) Sei $a \in \mathbb{Z}_m^*$. Zeigen Sie, daß auch $a + p \in \mathbb{Z}_m^*$, und daß $a^{p-1} \not\equiv (a+p)^{p-1} \pmod{p^2}$. Beenden Sie das Argument (10 Punkte).

**Aufgabe 2:** Zeigen Sie, daß der Test im Algorithmus, ob $m$ gerade ist, weggelassen werden kann (10 Punkte).
*Hinweis:* Benutzen Sie die Aufgaben 10.3 und 11.1.

**Aufgabe 3:** Zeigen Sie, daß sich Rabins Algorithmus für $n$-Bit-Eingabezahlen mit $O(n)$ arithmetischen Operationen auf $n$-Bit-Zahlen ausführen läßt. Beachten Sie dabei die Aufgaben 10.3 und 11.1 (10 Punkte).

**Aufgabe 4 (freiwillig—ohne Punkte):** Zeigen Sie, daß die folgende Variante von Rabins Algorithmus für alle $m \geq 3$ korrekt arbeitet. Beachten Sie dabei die Aufgaben 10.3 und 11.1. Wenn Sie wollen, können Sie den Algorithmus leicht programmieren; etwas umständlich ist höchstens die modulare Potenzierung. *random*$(A)$ returniert ein zufälliges und uniform verteiltes Element aus $A$.

**function** ProbablyPrime($m$ : *integer*) : *boolean*;
  (* $m$ is prime: ProbablyPrime returns *true*;
    $m$ is composite: With probability $\geq \frac{1}{2}$, ProbablyPrime returns *false* *)
  $i := 0$;
  $\ell := m - 1$;
  **while** *even*$(\ell)$ **do**
    **begin**
      $i := i + 1$;
      $\ell := \ell/2$;
    **end**;
  $x := (random(\{1, \ldots, m-1\}))^{\ell} \bmod m$;
  **repeat**
    $i := i - 1$;
    $y := x$;
    $x := x^2 \bmod m$;
  **until** $x = 1$ or $i < 0$;
  return($y \in \{1, m-1\}$ and $i \geq 0$);

**Abgabetermin:** 15.1.96.

Priv.-Doz. Torben Hagerup
Jordan Gergov
Universität des Saarlandes
Fachbereich 14, Informatik


# Komplexitätstheorie (WS 95/96)

15.1.96 **Übung 12** 12

**Aufgabe 1:** Zeigen Sie, daß RP gegenüber Bildung von Vereinigung und Durchschnitt abgeschlossen ist und daß BPP gegenüber Bildung von Komplement, Vereinigung und Durchschnitt abgeschlossen ist (40 Punkte).

**Aufgabe 2:** Nehmen wir an, wir modifizieren die Definition einer normalen NTM folgendermaßen: Es gibt nicht nur, wie üblich, die beiden Endzustände `accept` und `reject`, sondern zusätzlich einen dritten Endzustand, `undecided`. Entsprechend gibt es nicht nur akzeptierende und verwerfende, sondern auch *unentschlossene* Berechnungen. Wir definieren eine Klasse $C$ von Sprachen so: $L \in C$, wenn es eine modifizierte normale NTM $M$ gibt, die in Polynomialzeit läuft und für jede Eingabe $x$ die folgenden Eigenschaften hat:
(1) $M$ hat eine akzeptierende Berechnung auf $x \Rightarrow x \in L$;
(2) $M$ hat eine verwerfende Berechnung auf $x \Rightarrow x \notin L$;
(3) Höchstens die Hälfte der Berechnungen von $M$ auf $x$ ist unentschlossen.
Zeigen Sie, daß $C = \mathrm{ZPP}$ (30 Punkte).

**Aufgabe 3:** Diskutieren Sie, inwieweit ein effizienter randomisierter Primalitätstest auf dem Beweis von Satz 7.5 (Pratt) basiert werden kann. Gegeben eine zu testende Zahl $m$, wählt man also (einmal oder wiederholt) eine zufällige Zahl $a$ aus $\{1, \ldots, m-1\}$ und überprüft, ob $a$ die Ordnung $m-1$ hat (30 Punkte).

**Aufgabe 4 (freiwillig—ohne Punkte):** Implementieren Sie Rabins Primalitätstest (vgl. Aufgabe 11.4) in einer Programmiersprache Ihrer Wahl. Einige Routinen in C++, die es Ihnen erleichtern, den Algorithmus mit dem GNU C++ Compiler und der GNU C++ Bibliothek zu implementieren, finden Sie in ftp://ftp.mpi-sb.mpg.de/pub/vorlesungen/Komplexitaetstheorie/Uebung12.4.

Erweitern Sie Ihr Programm so, daß es bei der Antwort "Die Eingabe ist keine Primzahl" auch einen Zeugen für diese Aussage liefert. Bei Eingabe $p$ kann der Zeuge zum Beispiel eine Zahl $a$ mit der Eigenschaft $a^{p-1} \not\equiv 1 \pmod{p}$ sein (vgl. Lemmata 8.5 und 8.6).

Beweisen Sie mit Hilfe Ihres Programms, daß $2^{4421} - 1$ keine Primzahl ist. Schicken Sie den kommentierten Programmquelltext und Ihren mit dem Programm erstellten "Beweis" dafür, daß $2^{4421} - 1$ nicht prim ist, per Email an hroehrig@cscip.uni-sb.de. Für diese Aufgabe haben Sie Zeit bis zum 29.1.96.

**Abgabetermin:** 22.1.96.

Priv.-Doz. Torben Hagerup
Jordan Gergov
Universität des Saarlandes
Fachbereich 14, Informatik


# Komplexitätstheorie (WS 95/96)

22.1.96 **Übung 13** 13

**Aufgabe 1:** Zeigen Sie, daß unsere Approximationsalgorithmen für MINIMUM VERTEX COVER und MAXIMUM CUT (Sätze 11.1 und 11.2) nicht besser sind, als durch die Analyse garantiert. Geben Sie dazu einfache Beispiele an, bei denen die Kosten der berechneten zulässigen Lösung um genau einen Faktor von 2 von den optimalen Kosten abweichen (40 Punkte).

**Aufgabe 2:** Beschreiben Sie einen einfachen Approximationsalgorithmus für das MINIMUM BIN PACKING-Problem, der in Polynomialzeit läuft und eine Approximationsgüte von höchstens 2 hat (30 Punkte).
*Hinweis:* Können Sie erreichen, daß alle Bins, bis auf höchstens einen, mehr als halbvoll sind?

**Aufgabe 3:** Die *absolute Approximationsgüte* eines Approximationsalgorithmus $A$ bei Eingabe $x$ ist

$$|c(A(x)) - Opt(x)|,$$

wobei $c(A(x))$ die Kosten der von $A$ produzierten Lösung und $Opt(x)$ die optimalen Kosten sind. Die erlaubte Abweichung ist hier also kein konstanter Faktor, sondern lediglich ein konstanter additiver Term. In dieser Aufgabe wollen wir das folgende zeigen: Falls P $\neq$ NP, gibt es keinen Algorithmus $A$ für das MAXIMUM-VALUE KNAPSACK-Problem, der in Polynomialzeit läuft und für alle Eingaben eine absolute Approximationsgüte hat, die durch eine feste Konstante $r \in \mathbb{N}$ beschränkt ist.

(a) Betrachten Sie zu einer vorliegenden Eingabe $x$ eine Eingabe $x'$, die sich nur dadurch von $x$ unterscheidet, daß alle Werte $v_i$ $(r+1)$-mal größer sind. Beschreiben Sie das Verhältnis zwischen zulässigen Lösungen zu $x$ und zulässigen Lösungen zu $x'$ sowie zwischen den entsprechenden Kosten (15 Punkte).

(b) Wenden Sie einen hypothetischen Algorithmus mit absoluter Approximationsgüte $\leq r$ auf $x'$ an, transformieren Sie die Lösung zurück zu $x$ und leiten Sie einen Widerspruch her (15 Punkte).

**Abgabetermin:** 29.1.96.

Priv.-Doz. Torben Hagerup
Jordan Gergov
Universität des Saarlandes
Fachbereich 14, Informatik


# Komplexitätstheorie (WS 95/96)

29.1.96 **Übung 14** 14

**Aufgabe 1:** Betrachten Sie das folgende Problem: Die Eingabe besteht aus (der Zahl $n \in \mathbb{N}$ sowie) $n$ ganzen Zahlen $a_1, \ldots, a_n$. Zu berechnen sind die *Präfix-Summen* $a_1, a_1 + a_2, \ldots, a_1 + a_2 + \cdots + a_n$; das Problem verallgemeinert also das Problem, die Summe von $n$ Zahlen zu bestimmen, und ist das vielleicht fundamentalste Problem im Bereich der parallelen Algorithmen. Zeigen Sie:

(a) Präfix-Summen-Probleme der Größe $n$ können in $O(\log n)$ Zeit auf einer PRAM mit $n$ Prozessoren gelöst werden (20 Punkte).
*Hinweis:* Ergänzen Sie unsere Berechnung "den Baum hoch" um eine sich anschließende Berechnung "den Baum runter".

(b) Präfix-Summen-Probleme der Größe $n$ können in $O(\log n)$ Zeit auf einer PRAM mit $O(n/\log n)$ Prozessoren gelöst werden, also in logarithmischer Zeit mit optimalem Speedup (20 Punkte).

**Aufgabe 2:** Beweisen Sie Lemma 13.6 aus der Vorlesung (30 Punkte).

**Lemma** 13.6: Seien $f, g \in \mathcal{F}$. Dann enthält $\mathcal{F}$ eine Erweiterung $h$ von $g \circ f$, die in konstanter Zeit aus $f$ und $g$ berechnet werden kann.

Zur Erinnerung: $\mathcal{F}$ ist die Menge aller Funktionen $f$ der Form

$$f(x) = \frac{ax + b}{cx + d}, \quad a, b, c, d \in \mathbb{R}$$

mit Definitionsmenge $Dom(f) = \{x \in \mathbb{R} \mid cx + d \neq 0\}$.

**Aufgabe 3:** Zeigen Sie, daß $n$ ganze Zahlen in $O(\log n)$ Zeit auf einer PRAM mit polynomiell (in $n$) vielen Prozessoren sortiert werden können. Hat Ihr Algorithmus optimalen Speedup? (30 Punkte)
*Hinweis:* Vergleichen Sie zunächst jeden mit jedem.

**Abgabetermin:** 5.2.96.

Priv.-Doz. Torben Hagerup
Jordan Gergov
Universität des Saarlandes
Fachbereich 14, Informatik

# Komplexitätstheorie (WS 95/96)

5.2.96 **Übung 15** 15

**Aufgabe 1:** Definieren wir eine *$AC^0$-Familie* als eine uniforme Familie $\{\gamma_n\}_{n=0}^{\infty}$ von Schaltkreisen mit unbeschränktem Fanin, die konstante Tiefe und polynomielle Größe haben. Wir generalisieren Schaltkreise dabei so, daß mehrere Ausgangsgatter erlaubt sind; ein Schaltkreis kann also gleichzeitig mehrere Bits berechnen. In dieser Aufgabe wollen wir zeigen, daß es eine $AC^0$-Familie gibt, die die Summe $z_{n+1}z_n \ldots z_1$ von zwei $n$-Bitzahlen $x_n \ldots x_1$ und $y_n \ldots y_1$ berechnet.

(a) Überzeugen Sie sich davon, daß es reicht, den Vektor der Überträge, die bei der Addition auftreten, mit einer $AC^0$-Familie zu berechnen (20 Punkte).

(b) Beschreiben Sie einen $AC^0$-Schaltkreis, der bei Eingabe $a_n, \ldots, a_1, b_n, \ldots, b_1, b_0$ das Bit $b_k$ berechnet, wobei

$$k = \max(\{0\} \cup \{j \mid 1 \leq j \leq n \text{ und } a_j = 1\}) \quad \text{(20 Punkte)}.$$

(c) Beschreiben Sie einen $AC^0$-Schaltkreis, der bei Eingabe $a_n, \ldots, a_1, b_n, \ldots, b_1, b_0$ die Bits $b_{k_{n+1}}$, $b_{k_n}, \ldots, b_{k_1}$ berechnet, wobei

$$k_i = \max(\{0\} \cup \{j \mid 1 \leq j < i \text{ und } a_j = 1\}),$$

für $i = 1, \ldots, n+1$ (20 Punkte).

(d) Benutzen Sie Teil (c) mit

$$a_j = \begin{cases} 1, & \text{falls } x_j = y_j \\ 0, & \text{sonst} \end{cases}$$

und $b_j = x_j$, für $j = 1, \ldots, n$ (legen Sie $b_0$ geeignet fest), und beenden Sie das Argument (20 Punkte).

**Aufgabe 2:** Zeigen Sie, daß Satz 3.3 (Savitch) aus den beiden Sätzen 14.3 und 14.4 folgt (20 Punkte).

**Abgabetermin:** 12.2.96.

# Stichwortverzeichnis

Eine Seitenzahl in kursiver Schrift verweist auf die Definition eines Begriffs. Weitere Vorkommen sind vielfach nicht aufgeführt.