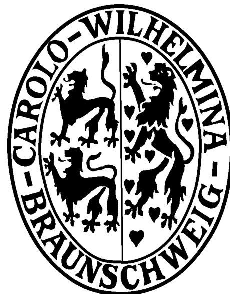
Theoretische Informatik

Vorlesungsskript

Dietmar Wätjen



Institut für Theoretische Informatik

Technische Universität Braunschweig

Oktober 2000

Inhaltsverzeichnis

Vorwort **3**

1 Endliche Automaten **5**

1.1 Einführung 5
1.2 Mealy- und Moore-Automaten 7
1.3 Reduktion von Automaten 16

2 Turingmaschinen **23**

2.1 Definitionen 23
2.2 Beispiele für Turingmaschinen und ihre Zusammensetzbarkeit 25
2.3 Modifizierte Turingmaschinen 31
2.4 Turing-Berechenbarkeit 34
2.5 Gödelisierung 40
2.6 Universelle Turingmaschinen 43
2.7 Unentscheidbare Probleme 44
2.8 Registermaschinen 49

3 Rekursive Funktionen **61**

3.1 Primitiv-rekursive Funktionen 61
3.2 Die Ackermann-Funktion 66
3.3 Der μ-Operator und μ-rekursive Funktionen 72

4 Sprachen, Grammatiken und erkennende Automaten **79**

4.1 Einführung 79
4.2 Die Chomsky-Hierarchie 81
4.3 Endliche erkennende Automaten und reguläre Sprachen 85
4.4 Reguläre Ausdrücke 92
4.5 Weitere Automatentypen und ihre zugehörigen Sprachen 96

5 Fixpunkttheorie und kontextfreie Sprachen **101**

5.1 Partielle Ordnungen und Fixpunkte 101
5.2 Fixpunkttheorie und kontextfreie Sprachen 110

6 Deterministische Polynomialzeitalgorithmen **117**

6.1 Beispiele effizienter Algorithmen 118
6.2 Die Komplexitätsklasse P 129
6.3 Berechnungsprobleme und Reduzierbarkeit 135
6.4 Die Robustheit der Klassen P und FP 141
6.5 Effiziente geometrische Algorithmen 145

7 Nichtdeterministische Polynomialzeitalgorithmen **153**

7.1 Die Komplexitätsklasse NP 153
7.2 NP-Vollständigkeit 157
7.3 Weitere NP-vollständige Probleme 163

8 Komplexität von Optimierungsalgorithmen **171**
8.1 Optimierungsprobleme 171
8.2 Approximation von Optimierungsproblemen 178

9 Raumkomplexität **185**

10 Parallele Algorithmen **191**
10.1 Das PRAM-Modell 191
10.2 PRAM-Algorithmen 192
10.3 Simulationen zwischen verschiedenen PRAM-Modellen 200
10.4 Die Komplexitätsklasse *NC* 203
10.5 Boolesche Schaltkreise und PRAMs 206
10.6 Netzwerkmodelle 214

Literaturverzeichnis **221**

Index **223**

6 Deterministische Polynomialzeit-algorithmen

In Kapitel 2 haben wir uns intensiv mit der Fragestellung beschäftigt, ob eine Funktion berechenbar oder ob ein Problem entscheidbar ist. Der Algorithmusbegriff wurde dabei durch Turingmaschinen formalisiert. In Satz 2.8.1 haben wir die Äquivalenz von Turingmaschinen und Registermaschinen bewiesen. Dabei haben wir uns überhaupt nicht dafür interessiert, wie lange eine solche Berechnung oder Entscheidung dauert, wieviele Schritte eine solche Maschine also bis zur Lösung durchführen muß. Es ist aber klar, daß die Dauer einer Berechnung eine äußerst wichtige Frage ist. Es gibt Probleme, für die es zwar sehr einfach zu notierende Algorithmen gibt, die jedoch trotzdem die Lösung in vernünftiger Zeit nicht liefern können. In diesem und den folgenden Kapiteln wollen wir uns daher mit der Effizienz von Algorithmen befassen.

Zunächst führen wir die Klasse P aller Probleme ein, die „effizient" lösbar sind. Genauer, wenn auch informal, definieren wir:

$P =$ Klasse aller Entscheidungsprobleme, die durch deterministische Algorithmen in polynomialer Zeit lösbar sind.

Dabei besitzt ein Entscheidungsproblem als Lösung für jeden speziellen Fall (jede Eingabe) des Problems die möglichen Antworten „ja" oder „nein". Der Zeitbedarf des Algorithmus für die ungünstigsten Fälle des Problems, also seine Zeitkomplexität, ist durch ein Polynom in der Länge der Eingabe nach oben beschränkt. Ein solcher Algorithmus hält immer an und liefert für jeden Fall des Problems die Antwort „ja" oder „nein".

Es zeigt sich, daß die Klasse P nicht von dem Modell der Berechnung abhängt. Das Modell kann eine Turingmaschine, eine Registermaschine oder auch ein PC sein. Abhängig von der Art des Modells wird das Polynom, das die Schranke der Zeitkomplexität ausdrückt, ganz unterschiedlich sein. Der „Grad der Effizienz" hängt also sehr von der konkreten Implementierung ab, wobei jedoch in jedem Fall die Beschränkung durch ein Polynom erfolgt.

Als ein Beispiel aus der Klasse P nennen wir das Problem, die Knoten eines vorgelegten Graphen mit zwei Farben so zu färben, daß benachbarte Knoten verschiedene Farben besitzen. Dieses Problem werden wir in Abschnitt 6.1 genauer betrachten.

Eine umfassendere Klasse ist

$NP =$ Klasse aller Entscheidungsprobleme, die durch nichtdeterministische Algorithmen in polynomialer Zeit lösbar sind.

Ein nichtdeterministischer Algorithmus, der allerdings nur ein theoretisches Konzept darstellt, hat nach jedem Schritt in nichtdeterministischer Weise mehrere Zwischenergebnisse zur Verfügung. Er kann diejenigen Zwischenergebnisse „erraten", die ihn zu einer Lösung des Problems führen. Der Algorithmus hält für jeden Fall, der die Antwort „ja" liefert, in polynomialer Zeit an. Der Zeitbedarf richtet sich nach der kürzest möglichen Rechnung, die zu dieser Lösung führt. In anderen Fällen muß der

Algorithmus nicht halten. Für jeden „ja“-Fall des Problems kann eine Überprüfung in polynomialer Zeit durchgeführt werden.

Offenbar gilt $P \subset NP$. Bis heute ist nicht bekannt, ob $P = NP$ gilt. Jedoch sind viele sogenannte NP-vollständige Probleme gefunden worden. Wenn für ein einziges dieser Probleme nachgewiesen werden könnte, daß es in polynomialer Zeit gelöst werden kann, dann gilt es auch für alle anderen Probleme aus NP. Das heißt, es würde $P = NP$ folgen. Da es so viele wichtige und viel untersuchte NP-vollständige Probleme gibt, ist es sehr unwahrscheinlich, daß die Gleichheit gilt.

Das Problem, die Knoten eines vorgelegten Graphen mit drei Farben so zu färben, daß benachbarte Knoten verschiedene Farben besitzen, ist NP-vollständig. Dies werden wir in Satz 7.3.3 beweisen. Eine Überprüfung, ob eine vorgeschlagene Färbung eines vorgelegten Graphen wirklich eine gültige 3-Färbung ist, kann in polynomialer Zeit durchgeführt werden. Für das Finden einer solchen Färbung ist jedoch kein effizienter Algorithmus bekannt.

Ein nichtdeterministischer Algorithmus für ein Problem aus NP kann mit Hilfe eines deterministischen Algorithmus mit exponentiellem Zeitbedarf simuliert werden. Dieser deterministische Algorithmus ist im allgemeinen sehr ineffizient und kaum handzuhaben.

Dieses Kapitel hat, ebenso wie die vorhergehenden, einen theoretischen Charakter. So werden bei der „effizienten“ Klasse P beliebige Polynome zur Zeitbeschränkung verwendet, was bei entsprechendem Grad des Polynoms, z.B. 363, nicht gerade als effizient gelten kann. Dennoch hat die Komplexitätstheorie wichtige praktische Auswirkungen. Wenn Sie in einer Firma beauftragt werden, die Software für ein bestimmtes Problem zu entwerfen und Sie nach vielem Nachdenken zu keiner Lösung gekommen sind, dann wäre es natürlich am besten, wenn Sie Ihrem Chef beweisen könnten, daß er Ihnen ein unlösbares Problem gestellt hat, daß Sie also mit einer Lösung dieses Problems auch eine Lösung des Halteproblems hätten. Wenn Sie ihm aber sagen können, daß sie keine effiziente Lösung haben finden können, weil sich das Problem als NP-vollständig herausgestellt hat, müssen Sie auch nicht mit Ihrer Entlassung wegen Unfähigkeit rechnen. Es ist ja bekannt, daß viele kluge Leute auf der ganzen Welt versucht haben, für solche Probleme effiziente Lösungen zu finden. Wäre es nur einmal gelungen, dann hätten auch Sie Ihr Problem effizient lösen können.

6.1 Beispiele effizienter Algorithmen

In diesem Abschnitt wollen wir die Effizienz von Algorithmen anhand einiger Beispiele genauer analysieren. Die Algorithmen werden dabei nicht durch Turingmaschinen oder Registermaschinen formal beschrieben, sondern intuitiv oder in Pseudoprogrammcode dargestellt. Wir beginnen mit einem einfachen Beispiel, nämlich der Multiplikation von zwei Zahlen. Aus der Schule ist der Standardalgorithmus bekannt, der

z.B. für $1984 \cdot 6713$ nach dem folgenden Schema abläuft:

$$\begin{array}{cccccccc}
1 & 9 & 8 & 4 & \cdot & 6 & 7 & 1 & 3 \\
\hline
 & & & & & 5 & 9 & 5 & 2 \\
 & & & & 1 & 9 & 8 & 4 & \\
 & & & 1 & 3 & 8 & 8 & 8 & \\
 & & 1 & 1 & 9 & 0 & 4 & & \\
\hline
 & & 1 & 3 & 3 & 1 & 8 & 5 & 9 & 2
\end{array}$$

Bei zwei n-stelligen Zahlen wird jede der n Zeilen unter dem oberen Strich in n Schritten errechnet, insgesamt sind dies n^2 Schritte. Die Addition der Zeilen benötigt höchstens $2n \cdot n$ Schritte. Insgesamt erhalten wir also eine Schranke von $3n^2$ Schritten. Um von der Konstante 3 absehen zu können, verwenden wir die O-Notation. Wir wiederholen die Definition 1.3.5, wobei wir zusätzlich die Ω-Notation angeben, mit der ein Mindestzeitbedarf für eine Rechnung ausgedrückt werden kann.

Definition 6.1.1 Es seien $f, g : \mathbb{N}_0 \to \mathbb{R}^+$ Funktionen.

(a) Es gilt $f(n) = O(g(n))$ (kurz $f = O(g)$), wenn $c \in \mathbb{R}$ mit $c > 0$ und ein $n_0 \in \mathbb{N}$ existieren, so daß für alle $n \in \mathbb{N}$ mit $n \geq n_0$ die Ungleichung

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

erfüllt ist.

(b) Es gilt $f(n) = \Omega(g(n))$ (kurz $f = \Omega(g)$), wenn $c \in \mathbb{R}$ mit $c > 0$ und ein $n_0 \in \mathbb{N}$ existieren, so daß für alle $n \in \mathbb{N}$ mit $n \geq n_0$ die Ungleichung

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

erfüllt ist. □

Beispiel 6.1.1 Jedes quadratische Polynom gehört zur Klasse $O(n^2)$. Dies sehen wir wie folgt ein. Für ein Polynom $f(n) = an^2 + bn + c$ mit Konstanten $a, b, c \in \mathbb{R}^+$ gilt

$$f(n) = \left(a + \frac{b}{n} + \frac{c}{n^2}\right) n^2.$$

Für $n \geq \max\{b, c\}$ erhalten wir $f(n) \leq (2 + a)n^2$. Bei der Wahl von $n_0 \geq \max\{b, c\}$ und $c = 2 + a$ folgt die Behauptung. Die Zugehörigkeit zur Klasse $\Omega(n^2)$ ist wegen $f(n) \geq an^2$ trivial.

Allgemeiner gehört ein Polynom $f(n) = a_k n^k + \ldots a_1 n + a_0 = \left(a_k + \frac{a_{k-1}}{n} + \frac{a_0}{n^k}\right) n^k$ vom Grade k zur Klasse $O(n^k)$. Hier sind $n_0 \geq \max\{a_{k-1}, \ldots, a_0\}$ und $c = k + a_k$ zu wählen.

Die Exponentialfunktion $f(n) = c^n$ für ein $c \in \mathbb{R}$, $c > 1$, gehört dagegen wegen $\lim_{n\to\infty} \frac{c^n}{n^k} = \infty$ nicht zu $O(n^k)$ für irgendein $k \in \mathbb{N}$. □

Der Zeitbedarf für die Multiplikation ist sicher proportional zu der Anzahl der Schritte. Wir erhalten für die Multiplikation nach der obigen Methode einen Zeitbedarf von $O(n^2)$. Wir werden jetzt zeigen, was vielleicht auf dem ersten Blick überraschend ist, daß die Multiplikation mit einer kleineren Zeitschranke ausgeführt werden kann.

Satz 6.1.1 Es gibt einen Algorithmus für die Multiplikation von zwei n-stelligen Zahlen, der einen Zeitbedarf von $O(n^{\log_2 3})$ besitzt.

Beweis: Zur Vereinfachung nehmen wir an, daß die Zahlen in Binärdarstellung gegeben sind. Wenn wir für $n \in \mathbb{N}$ zwei $2n$-Bit-Zahlen $u = u_{2n-1} \ldots u_0$ und $v = v_{2n-1} \ldots v_0$ mit $u_i, v_i \in \{0,1\}$, $i = 0, \ldots, 2n-1$, vorliegen haben, dann können wir

$$u = 2^n U_1 + U_0, \ v = 2^n V_1 + V_0$$

mit $U_1 = u_{2n-1} \ldots u_n$ (die „oberen" Bits von u) und $U_0 = u_{n-1} \ldots u_0$ (die „unteren" Bits von u) sowie $V_1 = v_{2n-1} \ldots v_n$ und $V_0 = v_{n-1} \ldots v_0$ schreiben. Es folgt

$$\begin{aligned} uv &= 2^{2n} U_1 V_1 + 2^n U_1 V_0 + 2^n U_0 V_1 + U_0 V_0 \\ &= (2^{2n} + 2^n) U_1 V_1 + 2^n (U_1 - U_0)(V_0 - V_1) + (2^n + 1) U_0 V_0. \end{aligned}$$

Bei dieser Rechnung werden drei Multiplikationen von n-Bit-Zahlen durchgeführt, nämlich $U_1 V_1$, $(U_1 - U_0)(V_0 - V_1)$ und $U_0 V_0$. Dazu kommen einige Additionen und für die Multiplikation mit einer Zweierpotenz einige Shifts. Wenn $T(n)$ die Zeit für die Ausführung der Multiplikation zweier n-Bit Zahlen ist, dann erhalten wir offenbar

$$T(2n) \leq 3T(n) + cn$$

mit einer geeigneten Konstanten c. Nun können wir weiter rekursiv vorgehen.

Wählen wir speziell $n = 2^{k-1}$ für ein $k \in \mathbb{N}$, so gilt

$$T(2^k) \leq 3T(2^{k-1}) + c \cdot 2^{k-1}.$$

Wir beweisen durch Induktion

$$T(2^k) \leq c(3^k - 2^k) \text{ für } k \geq 1.$$

Dabei ist c groß genug gewählt, damit die Ungleichung für $k = 1$ erfüllt ist, was auch der Induktionsbeginn ist. Wir nehmen nun an, daß die Ungleichung für k richtig ist, und beweisen sie für $k + 1$. Es ist

$$\begin{aligned} T(2^{k+1}) &\leq 3T(2^k) + c \cdot 2^k \\ &\leq 3c(3^k - 2^k) + c \cdot 2^k \\ &= c(3 \cdot 3^k - 3 \cdot 2^k + 2^k) \\ &= c \cdot (3^{k+1} - 2^{k+1}). \end{aligned}$$

Es folgt

$$T(n) \leq T(2^{\lceil \log_2 n \rceil}) \leq c(3^{\lceil \log_2 n \rceil} - 2^{\lceil \log_2 n \rceil}) < 3c \cdot 3^{\log_2 n} = 3cn^{\log_2 3}. \quad \square$$

Die Laufzeit der Multiplikation kann somit von $O(n^2)$ auf $O(n^{\log_2 3}) = O(n^{1{,}585\ldots})$ herabgesetzt werden. Beide Algorithmen zur Multiplikation haben einen polynomialen Zeitbedarf. Damit der Zeitgewinn tatsächlich sichtbar wird und nicht durch die Konstante c aufgehoben wird, muß n genügend groß sein. Dies ist typisch für viele ausgefeilte Algorithmen für verschiedene bekannte Probleme, die schneller sind als die

klassischen üblichen Verfahren. Erst bei entsprechend großem Umfang des Problems tritt ein tatsächlicher Zeitgewinn ein.

Wir verzichten darauf, den Algorithmus für die aus dem Beweis von Satz 6.1.1 sich ergebende Multiplikation im Einzelnen anzugeben. Der Zeitbedarf für die Multiplikation kann noch weiter verringert werden. Es gilt z.B. (siehe [13], Section 4.3.3, Theorem A), daß für alle $\varepsilon > 0$ ein Multiplikationsalgorithmus existiert, für den die Anzahl der Operationen $T(n) < c(\varepsilon)n^{1+\varepsilon}$ ist mit einer von n unabhängigen Konstanten $c(\varepsilon)$.

In diesem und auch dem nächsten Kapitel werden Graphen eine wichtige Rolle spielen. Wir geben zunächst ihre Definition an. In der folgenden Definition bedeute $\mathcal{P}_2(M)$ für eine Menge M die Menge aller ihrer zweielementigen Teilmengen, wobei für jedes $x \in M$ auch $\{x, x\} \in \mathcal{P}_2(M)$ gelten soll.

Definition 6.1.2 $G = (V, E)$ heißt (endlicher) (*ungerichteter*) *Graph*, wenn folgende Eigenschaften erfüllt sind:

(a) V und E sind endliche Mengen, die Menge der *Knoten* (englisch: *vertices*) und die Menge der *Kanten* (englisch: *edges*).

(b) Es gilt $E \subset \mathcal{P}_2(V)$, wobei ein Element $\{v_1, v_2\} \in E$ als Kante zwischen den Knoten v_1 und v_2 bezeichnet wird.

$G = (V, E)$ heißt (endlicher) *gerichteter Graph*, wenn (a) sowie die folgende Eigenschaft erfüllt ist:

(c) Es gilt $E \subset V \times V$, wobei ein Element $(v_1, v_2) \in E$ als Kante vom Knoten v_1 in den Knoten v_2 bezeichnet wird. □

Wir vereinbaren, daß wir eine Kante eines ungerichteten Graphen ebenfalls in der Form (v_1, v_2) schreiben dürfen. Dann bedeutet (v_2, v_1) dieselbe Kante, aber nur eines der beiden Paare muß notiert werden. Es kann auch eine Kante von einem Knoten v_1 in sich geben (eine Schlinge) Daneben ist es auch üblich, *Multigraphen* zu betrachten. Bei Multigraphen kann es mehrere (parallele) Kanten zwischen zwei Knoten geben. Um auch sie zu berücksichtigen, müßte Definition 6.1.2 entsprechend abgeändert werden, worauf wir hier jedoch verzichten wollen. Wir gehen davon aus, daß die betrachteten Graphen *schlicht* sind, d.h. keine Schlingen und parallele Kanten haben.

Ein Graph kann in offensichtlicher, aber nicht eindeutiger Weise bildlich dargestellt werden. Dies eignet sich nicht zur Eingabe und weiteren Verarbeitung im Rechner. Dafür kann man jedoch die Knoten- und Kantenmenge verwenden. Eine weitere Darstellungsmöglichkeit für einen gerichteten Graphen $G = (V, E)$ mit n Knoten ergibt sich zum Beispiel durch seine *Adjazenzmatrix*. Es handelt sich um die $n \times n$-Matrix $A = (a_{i,j})$ mit

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

für alle $i, j \in \{1, \ldots, n\}$. Eine andere Möglichkeit der Beschreibung erfolgt durch *Adjazenzlisten*. Dies sind n lineare Listen, wobei die i-te Liste alle Knoten j mit $(i, j) \in E$ enthält und die Listenköpfe in einem Array gespeichert sind. Wir betrachten dazu ein Beispiel.

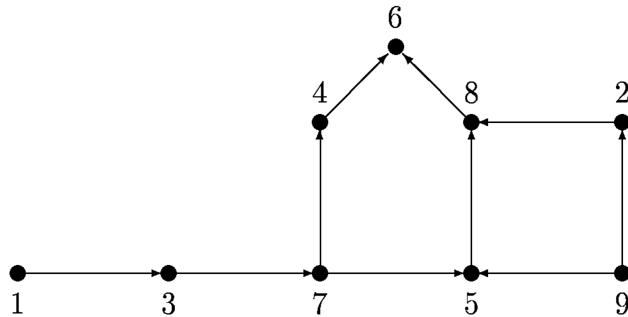
Beispiel 6.1.2 Der gerichtete Graph besitze die Knotenmenge

$$V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

und die Kantenmenge

$$E = \{(1,3), (3,7), (7,5), (7,4), (5,8), (9,2), (9,5), (2,8), (4,6), (8,6)\}.$$

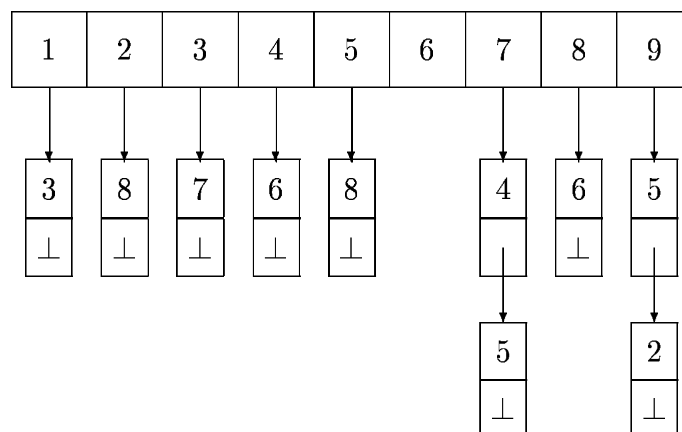
Er kann dann wie folgt bildlich dargestellt werden:



Die zugehörige Adjazenzmatrix ist

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

Die Adjazenzlisten sind durch



gegeben. □

Definition 6.1.3 Es sei $G = (V, E)$ ein Graph. Ein *Weg* (auch *Pfad* genannt) in G von einem Knoten v_{i_0} in einen Knoten v_{i_r}, $r \in \mathbb{N}$, ist eine Folge von paarweise verschiedenen Kanten

$$(v_{i_0}, v_{i_1}), (v_{i_1}, v_{i_2}), \ldots, (v_{i_{r-1}}, v_{i_r})$$

mit $v_{i_j} \in V$, $j = 1, \ldots, r$. Durch die Anzahl der Kanten ist dabei die *Länge* des Weges definiert. Ein *Zyklus* (auch *Kreis* genannt) ist ein Weg mit $v_{i_0} = v_{i_r}$. Ein Weg oder Zyklus heißt *einfach*, wenn jeder seiner Knoten außer $v_{i_0} = v_{i_r}$ nur einmal vorkommt. Ein Graph heißt *zyklenfrei*, wenn er keine Zyklen besitzt. □

Für gerichtete Graphen gibt es auch die Begriffe *gerichteter Weg* und *gerichteter Zyklus*. Dies geschieht in offensichtlicher Weise und kann symbolisch genauso wie in Definition 6.1.3 geschehen. Der Graph aus Beispiel 6.1.2 ist als gerichteter Graph zyklenfrei, nicht jedoch als ungerichteter Graph. Ein zyklenfreier gerichteter Graph $G = (V, E)$ kann auch als die partiell geordnete Menge $(V, \leq)$ mit der durch

$$v \leq v' \iff (v, v') \in E$$

gegebenen Ordnungsrelation $\leq$ aufgefaßt werden (siehe Definition 5.1.1).

Definition 6.1.4 Es sei $G = (V, E)$ ein gerichteter zyklenfreier Graph mit $V = \{v_1, \ldots, v_n\}$ für ein $n \in \mathbb{N}$. Eine Knotenreihenfolge $R = (v_{i_1}, \ldots, v_{i_n})$ aller Knoten aus V heißt *topologische Ordnung* von G, wenn für alle Kanten $(v_{i_j}, v_{i_k}) \in E$ die Beziehung $j < k$ gilt, d.h., wenn v_{i_j} links von v_{i_k} in R steht. □

Eine einfache Lösungsidee für das topologische Sortieren wird durch den folgenden nicht sehr detaillierten Algorithmus beschrieben.

Algorithmus 6.1.1
{Eingabe: ein gerichteter zyklenfreier Graph $G = (V, E)$,
Ausgabe: eine topologische Ordnung von V}
while $V \neq \emptyset$
do wähle $v \in V$, für das kein $v' \in V$ mit $(v', v) \in E$ existiert;
gebe v in die Ausgabe;
$V := V - \{v\}$
od □

Zur genaueren Beschreibung des Algorithmus ist die Art der Darstellung von G von Bedeutung. Auf der Basis von Algorithmus 6.1.1 können wir bei Verwendung der $n \times n$-Adjazenzmatrix A des Graphen direkt wie folgt vorgehen: Wir suchen eine Spalte von A, die aus lauter Nullen besteht, z. B. die j-te Spalte. Das bedeutet, daß der j-te Knoten keinen Vorgänger hat. Er kommt in die Ausgabe. Um diesen Knoten nicht mehr in der Adjazenzmatrix zu berücksichtigen, wird die j-te Spalte und j-te Zeile von A gestrichen und in dieser reduzierten Matrix wird wieder nach einer Spalte aus lauter Nullen gesucht. Das Verfahren wird fortgesetzt, bis A zu einer einelementigen Matrix reduziert ist.

Das Suchen einer Spalte aus lauter Nullen erfordert im schlechtesten Fall das Betrachten aller Elemente von A, also von n^2 Elementen. Im nächsten Schritt sind dies maximal $(n-1)^2$ Elemente. Insgesamt müssen im schlechtesten Fall

$$n^2 + (n-1)^2 + \ldots + 2^2 + 1^2 = \frac{n(n+1)(2n+1)}{6}$$

Elemente angesehen werden. Der Algorithmus hat bei dieser Verwendung von Adjazenzmatrizen also einen Zeitbedarf von $O(n^3)$.

Falls Graphen in Form von Adjazenzlisten implementiert werden, kann dieser Algorithmus viel effizienter durchgeführt werden. Wir nehmen an, daß die Knoten von G fortlaufend durch die Zahlen $1, 2, \ldots, n$ gegeben sind. Für jeden Knoten v bezeichnen wir mit $in(v)$ die Anzahl aller Kanten $(v', v) \in E$. Dadurch wird ein Array bestimmt. Wird der Graph im Laufe des Algorithmus reduziert, dann wird auch $in(v)$ verringert. Wir setzen weiter

$$U = \{v \mid v \in V,\ in(v) = 0\}.$$

U ist also die Menge aller Knoten ohne Vorgänger. Sie kann als Schlange implementiert werden.

Algorithmus 6.1.2

{Eingabe: Gerichteter Graph $G = (V, E)$ mit n Knoten
Ausgabe: Topologische Ordnung ord: $V \to \{1, \ldots, n\}$ von G, falls G zyklenfrei ist}
{Initialisierung:}
$i := 0;$
$U := \emptyset;$
$in(1) := 0; \ldots ; in(n) := 0;$
for $v := 1$ **to** n {für alle Knoten aus V}
    do for alle Knoten v' mit $(v, v') \in E$
        do $in(v') := in(v') + 1$
        od
    od
for $v := 1$ **to** n {für alle Knoten aus V}
    do if $in(v) = 0$
        then $U := U \cup \{v\}$
    od
{Rekursionsschritt:}
while U einen Knoten v enthält
        do $U := U - \{v\};$
            $i := i + 1;$
            $\text{ord}(v) := i;$
            for alle Knoten w mit $(v, w) \in E$
                do $in(w) := in(w) - 1;$
                    if $in(w) = 0$ **then** $U := U \cup \{w\}$
                od
        od
{Ausgabe:}
if $n = i$ **then** „ord ist eine topologische Ordnung von G“;
if $n > i$ **then** „G ist nicht zyklenfrei“ □

Satz 6.1.2 Es sei $G = (V, E)$ ein Graph mit n Knoten und k Kanten. Algorithmus 6.1.2 sortiert die Knoten von G topologisch, falls G zyklenfrei ist. Ist G nicht zyklenfrei, so stellt der Algorithmus dies fest. Die Zeitkomplexität des Algorithmus ist $O(n+k) = O(n) + O(k)$.

Beweis: Falls G zyklenfrei ist, dann bilden die noch nicht sortierten Knoten zu jedem Zeitpunkt einen zyklenfreien Graphen. Jeder zyklenfreie Graph besitzt einen Knoten ohne Vorgänger. Solange noch nicht alle Knoten sortiert sind, gilt folglich $U \neq \emptyset$. Für jede Kante $(v, w) \in E$ gilt folgende Überlegung: falls im Rekursionsschritt der Knoten $v \in U$ sortiert wird ($\text{ord}(v) = i$), so gilt vor diesem Schleifendurchgang $in(w) \geq 1$. Das bedeutet, daß w erst in einem späteren Schleifenlauf sortiert wird und somit $\text{ord}(v) < \text{ord}(w)$ gilt, die topologische Ordnung also erfüllt ist. Besitzt G dagegen einen Zyklus, so können die Knoten des Zyklus niemals in U aufgenommen werden, da sie immer einen Vorgänger haben. Die **while**-Schleife wird also mit einem Wert $i < n$ verlassen.

Wir kommen jetzt zur Zeitkomplexität. Diese hängt von der Implementierung ab. Wir nehmen an, daß das Hinzufügen und Entfernen von Knoten zu bzw. aus (der Schlange) U jeweils eine konstante Zahl a von Zeiteinheiten benötigt, und auch jede Zuweisung soll in höchstens dieser Zeit ausführbar sein. Der Graph ist durch die Adjazenzlisten gegeben. Wir betrachten zunächst den Zeitbedarf für die erste **for**-Schleife. Es werden alle n Knoten durchlaufen, wobei für jeden Knoten die Knoten der zugehörigen Adjazenzliste betrachtet werden. Dies erfordert eine Zeit von $a(n + k)$. Damit dauert die Initialisierung insgesamt höchstens

$$a(1 + 1 + n + (n + k) + 2n) = 2a + 4an + ak$$

Zeiteinheiten. Beim Rekursionsschritt wird die **while**-Schleife höchstens n-mal durchlaufen. Die **for**-Schleife hat bei Berücksichtigung aller Durchläufe der **while**-Schleife einen Gesamtzeitbedarf von $3ak$. Somit ergibt sich für die Rekursion eine Zeitschranke von

$$a(n + n + n + 3k) = 3a(n + k).$$

Wenn wir die Ausgabe mit zwei Schritten in höchstens konstanter Zeit a ansetzen, hat der Algorithmus einen Zeitbedarf von höchstens

$$4a + 7an + 4ak.$$

Die Zeitkomplexität ist also von der Ordnung $O(n) + O(k) = O(n + k)$. □

Diese lineare Zeitkomplexität läßt sich nicht erreichen, wenn die Graphen durch Adjazenzmatrizen implementiert sind. Man benötigt dann mindestens $\frac{n^2}{2}$ Schritte. Jeder Algorithmus muß für jedes Paar (i, j) mindestens eines der Elemente $a_{i,j}$ oder $a_{j,i}$ in der Matrix aufsuchen. Dies sehen wir wie folgt ein. Wir nehmen an, daß ein Algorithmus für das topologische Sortieren existiert, der dies nicht tun muß. Für einen Graphen G mit n Knoten und keiner Kante liefert er natürlich eine topologische Ordnung. Wenn wir nun durch Einführung von zwei Kanten (i, j) und (j, i) aus G einen neuen Graphen G' konstruieren, so würde A diesen Graphen G' auf genau dieselbe Weise wie G bearbeiten und zum selben Ergebnis kommen, obwohl G' kein zyklenfreier Graph ist, ein Widerspruch.

Beispiel 6.1.3 Mit dem Graphen G aus Beispiel 6.1.2 liefert Algorithmus 6.1.2 nach der Initialisierung den Vektor

$$(in(1), \ldots, in(9)) = (0, 1, 1, 1, 2, 2, 1, 2, 0).$$

Die Menge U kann durch die Schlange $9 \to 1$ dargestellt werden. Nach Abschluß des Algorithmus erhalten wir die topologische Ordnung

$$R = (1, 9, 3, 2, 7, 4, 5, 8, 6)$$

von G. □

Als nächstes betrachten wir das Problem der 2-FÄRBBARKEIT von Graphen. Dies ist die Aufgabe, die Knoten eines ungerichteten Graphen mit zwei Farben, etwa rot und blau, so zu färben, daß *benachbarte* Knoten v und v', also Knoten, die durch eine Kante (v, v') verbunden sind, nicht dieselbe Farbe erhalten. Es ist klar, daß bei Graphen, die ein *Dreieck* (paarweise verschiedene Knoten v_1, v_2, v_3 mit einem einfachen Zyklus $(v_1, v_2), (v_2, v_3), (v_3, v_1))$ enthalten, eine solche 2-Färbung nicht möglich ist. Allgemeiner sehen wir sofort ein, daß es auch für Graphen mit beliebigen Zyklen ungerader Länge keine solche Färbung gibt. Somit können auch Schlingen, also Kanten (v, v), in einem 2-färbbaren Graphen offensichtlich nicht vorkommen. Wenn ein Graph 2-färbbar ist, dann gibt es in G keine Zyklen ungerader Länge.

Falls ein Graph G keine Zyklen ungerader Länge hat, können wir einen einfachen Algorithmus zur 2-Färbung angeben. Wir wählen einen beliebigen Knoten v und färben ihn rot. Dann werden alle Nachbarn von v blau gefärbt. Da G kein Dreieck enthält, ist dies problemlos möglich. Danach werden alle Nachbarn der zuletzt gefärbten Knoten rot gefärbt. Da G kein Dreieck und keinen einfachen Zyklus aus 5 Knoten (ein Fünfeck) enthält, ist auch dieses möglich. Das Verfahren wird fortgesetzt, bis die ganze *Komponente von v* (alle Knoten von G, die durch einen ungerichteten Weg mit v verbunden sind) gefärbt ist. Dann wählen wir einen beliebigen, noch nicht gefärbten Knoten von G, färben ihn rot und fahren wie zuvor fort. Dieser Algorithmus soll jetzt formaler beschrieben werden. Die Mengen aller (schon) roten oder blauen Knoten werden mit ROT bzw. BLAU bezeichnet. Die jeweils zuletzt gefärbten Knoten bilden die Menge F. Wir nehmen von vornherein an, daß keine Schlingen im Graphen enthalten sind. Das Vorhandensein von Schlingen kann bei Implementierung des Graphen (mit n Knoten und k Kanten) durch Adjazenzlisten in der Zeit $O(n + k)$ getestet werden.

Algorithmus 6.1.3
{Eingabe: Ungerichteter Graph $G = (V, E)$
Ausgabe: Eine 2-Färbung $V = \text{ROT} \cup \text{BLAU}$, falls es eine gibt}
{Initialisierung:}
ROT:= $\emptyset$;
BLAU:= $\emptyset$;
$F := \emptyset$;
{Rekursionsschritt:}
while $V \neq \emptyset$
        do wähle $v \in V$;
            $V := V - \{v\}$;

```
        ROT:= ROT ∪ {v};
        F := F ∪ {v};
        while F ≠ ∅
             do wähle w ∈ F;
                for alle u ∈ V mit (u, w) ∈ E
                    do if w ∈ ROT then BLAU:= BLAU ∪ {u}
                                  else ROT:= ROT ∪ {u};
                       F := F ∪ {u};
                       V := V − {u}
                    od;
                F := F − {w}
             od
    od;
{Ausgabe:}
ROT und BLAU
```

(Falls die Teilgraphen ROT oder BLAU, die aus den entsprechend gefärbten Knoten mit den zugehörigen Kanten bestehen, eine Kante enthalten, ist G nicht 2-färbbar. Anderenfalls liegt durch ROT und BLAU eine 2-Färbung von G vor.) □

Satz 6.1.3 Es sei G ein Graph mit n Knoten und k Kanten. Algorithmus 6.1.3 liefert eine 2-Färbung von G, falls sie existiert. Existiert sie nicht, so stellt der Algorithmus dies fest. Die Zeitkomplexität des Algorithmus ist bei Implementierung des Graphen durch Adjazenzlisten $O(n + k) = O(n) + O(k)$.

Beweis: Wir werden zeigen, daß die Teilgraphen ROT oder BLAU genau dann eine Kante enthalten, wenn G Zyklen ungerader Länge besitzt, wenn also G nicht 2-färbbar ist. Aufgrund der Bemerkung vor Algorithmus 6.1.3 beschränken wir uns auf Graphen ohne Schlingen.

In jedem Schritt des Algorithmus gilt entweder $v \in V$, $v \in$ ROT oder $v \in$ BLAU für jeden Knoten v. In jedem Fall wird $V = \emptyset$ erreicht, so daß am Ende jeder Knoten rot oder blau gefärbt ist. Falls G Zyklen ungerader Länge besitzt, müssen dann mindestens zwei benachbarte Knoten gleich gefärbt sein. Folglich enthalten dann ROT oder BLAU eine Kante.

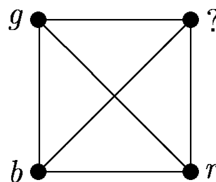
Für die umgekehrte Beweisrichtung gehen wir davon aus, daß ROT oder BLAU eine Kante enthalten. Es sei v der zu Beginn des Rekursionsschritts gewählte Knoten, der rot gefärbt wird. Wir beweisen, daß jeder Knoten u, der irgendwann rot (oder blau) gefärbt wird, einen Weg von gerader (oder ungerader) Länge nach v hat. Vor Beginn der inneren **while**-Schleife ist das immer richtig, denn es gibt nur den rot gefärbten Knoten v mit einem Weg der Länge 0 zu sich selbst. Jeder Durchgang der inneren Schleife erhält diese Eigenschaft. Falls nämlich w rot ist und somit einen Weg gerader Länge nach v hat, dann hat der blau zu färbende Knoten u, für den es eine Kante (u, w) gibt, offensichtlich einen Weg ungerader Länge nach v. Entsprechend hat, falls w blau ist, der rot zu färbende Knoten u einen Weg gerader Länge nach v.

Es sei (u, u') eine Kante in ROT. Dadurch erhalten wir einen Kreis, der mit einem Weg gerader Länge von u nach v beginnt, dann durch einen Weg gerader Länge von v nach u' fortgesetzt wird, um schließlich mit der Kante (u', u) abgeschlossen zu werden. Er hat folglich eine ungerade Länge. Analog können wir für eine Kante in BLAU

argumentieren.

Es bleibt die Aussage über die Zeitkomplexität des Algorithmus zu beweisen. Die Initialisierung benötigt die Zeit $O(1)$. Die Färbung einer Komponente K von G mit $k(K)$ Kanten (das heißt ein Durchlauf des Körpers der äußeren **while**-Schleife, also eine vollständige Ausführung der inneren **while**-Schleife) hat den Zeitbedarf $O(k(K))$, da jede Kante nur einmal aufgesucht wird. Falls G genau r Komponenten K_i, $i = 1, \ldots, r$, besitzt, gilt $k = k(K_1) + \ldots k(K_r)$. Die äußere **while**-Schleife wird n-mal durchlaufen, insgesamt ergibt sich dafür also die Zeitkomplexität $O(n + k)$. Für die Ausgabe muß überprüft werden, ob ROT oder BLAU eine Kante enthalten. Dies ist mit einem Durchsuchen der Adjazenzlisten und jeweiliger Überprüfung der Farben ebenfalls in der Zeit $O(n+k)$ möglich. Insgesamt benötigt der Algorithmus somit eine Zeit von $O(n + k)$. □

Statt mit zwei Farben die Knoten eines Graphen zu färben, kann man dies auch mit drei Farben versuchen. Jeder 2-färbbare Graph ist auch 3-färbbar. Ein Dreieck jedoch ist 3-färbbar, nicht aber 2-färbbar. Andererseits ist jeder *vollständige* Graph mit mehr als 4 Knoten, also ein Graph, bei dem jeder Knoten mit jedem anderen Knoten eine gemeinsame Kante besitzt, nicht 3-färbbar. Wenn wir rot, grün und blau durch die Buchstaben r, g und b kennzeichnen, wird dies durch den 3-Färbbarkeitsversuch in der folgenden Skizze deutlich.



Das Problem der 3-FÄRBBARKEIT von Graphen ist sehr viel schwieriger als das der 2-FÄRBBARKEIT. Es ist dafür kein effizienter Algorithmus bekannt. Wir werden später einsehen (siehe Satz 7.3.3), warum es wahrscheinlich keinen effizienten Algorithmus gibt. Ein ineffizienter Algorithmus kann sofort angegeben werden. Er probiert einfach alle Färbungen aus. Da jeder der n Knoten eines Graphen mit 3 unterschiedlichen Farben versehen werden kann, hat ein solcher Algorithmus einen exponentiellen Zeitbedarf von $O(3^n)$. Dies ist sehr aufwendig.

Zum Abschluß dieses Abschnitts betrachten wir die folgende Aufgabe. Es sind ein gerichteter Graph $G = (V, E)$ und zwei Knoten $v, w \in V$ gegeben. Es ist zu überprüfen, ob ein (gerichteter) Weg von v nach w führt. Dabei sei M die jeweilige Menge der Knoten, zu denen ein Weg von v führt. Dies leistet offenbar

Algorithmus 6.1.4

$M := \{v\}$;
while eine Kante $(x, y) \in E$ mit $x \in M$, $y \notin M$ existiert
    do $M := M \cup \{y\}$;
        $E := E - \{(x, y)\}$
    od;
if $w \in M$ **then** existiert ein Weg von v nach w
        else existiert kein solcher Weg □

Satz 6.1.4 Es sei $G = (V, E)$ ein gerichteter Graph mit n Knoten und k Kanten, und es gelte $v, w \in V$. Falls G durch Adjazenzlisten implementiert ist, entscheidet Algorithmus 6.1.4 mit einem Zeitbedarf von $O(n + k)$, ob ein gerichteter Weg von v nach w in G existiert.

Beweis: Die Aussage über den Zeitbedarf ist richtig, da die Adjazenzlisten nur jeweils einmal durchlaufen werden. □

6.2 Die Komplexitätsklasse P

Zu Beginn dieses Kapitels haben wir schon in informaler Weise die Klasse P der Entscheidungsprobleme eingeführt, die in polynomialer Zeit in bezug auf die Länge der Darstellung der jeweiligen Fälle des Problems lösbar sind. Dabei stellt sich natürlich die Frage, wie ein Lösungsalgorithmus implementiert ist. Wir haben im vorhergehenden Abschnitt zum Beispiel die Multiplikation sowie das topologische Sortieren betrachtet. Diese Probleme sind zunächst keine Entscheidungsprobleme, aber im Fall des topologischen Sortierens löst der Algorithmus auch das Entscheidungsproblem, ob ein vorgelegter gerichteter Graph topologisch sortiert werden kann. Multiplikation und topologisches Sortieren können in polynomialer Zeit gelöst werden, wobei wir in beiden Fällen unterschiedliche Algorithmen mit unterschiedlichem Grad des Polynoms angeben konnten. Für jede Implementierung gibt es also ein individuelles Polynom. Wir wollen zeigen, daß die Existenz eines Polynoms von der Implementierung unabhängig ist, wobei jedoch der Grad unterschiedlich ausfallen kann. Dies beweisen wir dadurch, daß wir für die Klasse P Turingmaschinen als Implementierungsmodell nehmen und zeigen, daß dieselbe Klasse auch durch das Modell der Registermaschinen beschrieben werden kann. Daß auch andere Berechnungsmodelle die Klasse P charakterisieren, kann ebenfalls bewiesen werden.

Im folgenden verwenden wir Turingmaschinen gemäß Definition 4.5.1. Die Begriffe einer haltenden Rechnung sowie der Begriff der Länge einer Rechnung werden wie in Definition 4.5.2 benutzt.

Definition 6.2.1 Es sei $T = (Z, X, \delta, z_0, F)$ eine deterministische Turingmaschine. Dann heißt

$$K_T(n) = \max_{w \in X^*, |w| = n} \{k \mid k \text{ ist die Länge einer haltenden Rechnung an } w\}$$

die Zeitkomplexität von T. Falls ein w mit $|w| = n$ existiert, an dem T nicht hält, wird $K_T(n) = \infty$ gesetzt. □

Wir erinnern daran, daß nach Definition 4.5.2 die Turingmaschine T über dem ersten Symbol des vorgegebenen Wortes w startet oder, falls $w = \varepsilon$ gilt, über dem Blankzeichen. Bei deterministischen Turingmaschinen spielen die Endzustände für die Bestimmung der Zeitkomplexität keine Rolle, da die Maximumbildung über alle haltenden Rechnungen erfolgt.

Beispiel 6.2.1 Die Sprache

$$L = \{0, 1\}^* - \{0, 1\}^*\{11\}\{0, 1\}^*,$$

also die Sprache aller Wörter über $\{0, 1\}$, die keine benachbarten Symbole 1 haben, kann von der Turingmaschine $T = (\{z_0, z_1\}, \{0, 1\}, \delta, z_0, \{z_0\})$ akzeptiert werden, wobei die Überführungsabbildung δ durch die folgende Turingtafel bestimmt ist.

z_0	b	z_0	s
z_0	0	z_0	r
z_0	1	z_1	r
z_1	b	z_0	s
z_1	0	z_0	r
z_1	1	z_1	s

Die initiale Bandinschrift ist $\ldots bwb \ldots$ mit $w \in \{0, 1\}^*$, wobei die Turingmaschine auf dem ersten Zeichen von w startet. Sie läuft nach rechts, bis sie entweder auf zwei aufeinander folgende Vorkommen von 1 stößt und mit dem Nichtendzustand z_1 hält oder aber beim ersten Blankzeichen mit dem Endzustand z_0 hält und somit das vorgelegte Wort akzeptiert. Offenbar gilt $K_T(n) = n + 1$. □

Beispiel 6.2.2 Wir betrachten die Sprache L = PALINDROM $= \{w \mid w \in X^*, sp(w) = w\} \in P$ aller Palindrome über dem Alphabet X. Wir geben eine deterministische Turingmaschine

$$T = (\{z_0, z_1, z_r, z_2, z_l, z_f\} \times \bar{X}, X, \delta, (z_0, b), \{(z_f, b)\})$$

mit $L(T) = L$ an, wobei ihre kommentierte Turingtafel (ohne unnötige Zeilen) durch

(z_0, b)	b	(z_f, b)	s	Test auf Palindromeigenschaft eines Wortes gerader Länge erfolgreich
(z_0, b)	x	(z_1, x)	b	$x \in X$, das Symbol x am linken Rand wird gelöscht und sich im Zustand gemerkt
(z_1, x)	b	(z_r, x)	r	$x \in X$, Einleitung der Rechtsbewegung
(z_r, x)	y	(z_r, x)	r	$x, y \in X$, Rechtsbewegung
(z_r, x)	b	(z_2, x)	l	$x \in X$, Finden des rechten Randes
(z_2, x)	x	(z_2, b)	b	$x \in X$, das Symbol am rechten Rand stimmt mit dem schon oben gelöschten am linken überein, es wird gelöscht
(z_2, x)	b	(z_f, b)	s	Test auf Palindromeigenschaft eines Wortes ungerader Länge erfolgreich
(z_2, b)	b	(z_l, b)	l	Einleitung der Linksbewegung
(z_2, x)	y	(z_2, x)	s	$x, y \in X$, $x \neq y$, es liegt kein Palindrom vor
(z_l, b)	x	(z_l, b)	l	$x \in X$, Linksbewegung
(z_l, b)	b	(z_0, b)	r	Übergang in die Anfangssituation, wobei zuvor links und rechts dasselbe Symbol x gelöscht wurde

definiert ist. Wir berechnen die Anzahl der Schritte, die T bei einem $w \in L$ der Länge $|w| = n$ durchführen muß. Für ein ebenso langes Wort $w \notin L$ sind offenbar weniger Schritte erforderlich. Für $n \geq 2$ sind bis einschließlich zum Löschen des rechten Symbols von w $n + 3$ Schritte erforderlich, bis zum Übergang in die Anfangssituation weitere n Schritte, also insgesamt $2n + 3$ Schritte zum Löschen des Symbols x auf der

rechten und linken Seite des Worts. Dieser Prozeß wird für $n-2 \geq 2$ mit $n-2$ statt n wiederholt usw. Ist w gerade, so erhält man bis zum Abschluß

$$(2n+3)+(2(n-2)+3)+\ldots+(2\cdot 2+3)+1$$

Schritte. Ist n ungerade, so beginnt man für das letzte Symbol x mit der Bandinschrift $\ldots b\underline{x}b\ldots$ im Zustand (z_0, b). Dann kommt man in 3 Schritten und einem abschließenden Schritt zur Akzeptierung des Wortes. Es sind also in diesem Fall insgesamt

$$(2n+3)+(2(n-2)+3)+\ldots+(2\cdot 3+3)+4$$

Schritte erforderlich. Dann kann man beide Summen und damit $K_T(n)$ sehr großzügig nach oben durch

$$(2n+3)+(2n+1)+\ldots+5+3+1=n^2+4n+4$$

(Beweis der Gleichheit durch vollständige Induktion) abschätzen, dabei speziell 1 und 4 durch $1+3+5$. Eine untere Schranke ist für $n \geq 2$ durch $\frac{1}{2}(n^2+4n+4)$ gegeben. Der Zeitbedarf ist also quadratisch in der Länge der Eingabe. □

Wir können für die Sprache der Palindrome aus Beispiel 6.2.2 eine fast doppelt so schnelle Turingmaschine konstruieren, indem wir die ersten beiden Symbole von jedem w speichern und dann sofort auf dem Band löschen, um sie anschließend mit den letzten beiden Symbolen, die dann auch gelöscht werden, zu vergleichen. Eine noch schnellere Turingmaschine würde drei, vier oder noch mehr Symbole auf einmal vergleichen. Für ein beliebiges $k \in \mathbb{N}$ können wir eine für große n fast k-mal schnellere Turingmaschine konstruieren. Jede dieser Turingmaschinen hat jedoch, da k ein konstanter Faktor ist, eine quadratische Zeitkomplexität.

Der entsprechende Beschleunigungsprozeß kann auch für Turingmaschinen durchgeführt werden, indem k Felder zu einem neuen Feld der neuen Turingmaschine zusammengefaßt werden. Aus diesem Grund ist nicht die konkrete Zeitkomplexität $K_T(n)$ einer Turingmaschine von Interesse, sondern nur die „Wachstumsgeschwindigkeit“, also die Klasse $O(K_T(n))$.

Definition 6.2.2 Es ist

$$P = \{L \mid \text{es existiert eine deterministische Turingmaschine } T \text{ mit } L = L(T) \text{ und } K_T(n) \leq p_T(n) \text{ für ein Polynom } p_T \text{ und alle } n\}$$

die *Komplexitätsklasse der in polynomialer Zeit entscheidbaren Sprachen.* □

Da für ein beliebiges Polynom p und ein beliebiges $n \in \mathbb{N}_0$ der Wert $p(n)$ endlich ist, folgt nach der Definition der Zeitkomplexität einer deterministischen Turingmaschine, daß für jedes $L \in P$ die zugehörige Turingmaschine T immer halten muß.

In Definition 6.2.2 haben wir P als eine Menge von Sprachen definiert. Entscheidungsprobleme wie z.B. das Problem, ob ein Graph 2-gefärbt werden kann, stellt man sich jedoch im allgemeinen nicht als Spracherkennungsprobleme vor. Bei Problemen erwartet man für jede Fragestellung die Antwort „ja“ oder „nein“. Daher identifiziert man jedes Problem mit der Menge seiner Fragestellungen, auf die die Antwort „ja“

lautet. Zur Bestimmung der mit den Entscheidungsproblemen assoziierten Sprache ist es notwendig, die Fragestellungen als Worte über irgendeinem Alphabet zu kodieren.

Man kann also mit jedem ja/nein-Problem in natürlicher Weise eine Sprache verbinden, nämlich die Menge der Fragestellungen mit positiver Antwort. Die zugehörige Sprache hängt von der Kodierung der Fragestellung ab. Weiterhin hängt die Komplexität des Problems von der Kodierung ab. Bei „vernünftigen" Kodierungen wird dadurch, wie wir schon zuvor erwähnt haben, die Klasse P nicht beeinflußt. Als vernünftige Regeln der Kodierung können wir folgendes ansehen:

(1) Zahlen werden binär oder dezimal dargestellt, jedoch nicht unär.
(2) Graphen werden durch Adjazenzmatrizen oder Adjazenzlisten beschrieben.
(3) Mengentheoretische Operationen wie Hinzufügen und Wegnehmen eines Elements und auch Vereinigungs- und Durchschnittsbildung werden so implementiert, daß ihre Ausführung eine konstante Zahl von Zeiteinheiten benötigt.

Satz 6.2.1 (a) PALINDROM $\in P$.
(b) 2-FÄRBBARKEIT $\in P$ (das Problem, ob ein Graph mit zwei Farben gefärbt werden kann).

Beweis: Die Aussage (a) folgt aus Beispiel 6.2.2, denn die Turingmaschine des Beispiels hat die Zeitkomplexität $O(n^2)$. Das Problem aus (b) wird nach Satz 6.1.3 durch Algorithmus 6.1.3 in linearer Zeit gelöst. Um 2-FÄRBBARKEIT $\in P$ formal zu beweisen, müßte eine Turingmaschine mit polynomialer Zeitkomplexität zur Lösung dieses Problems angegeben werden. Dies erfordert einen gewissen Aufwand. Der vorgegebene Graph muß als Eingabe auf das Band der Turingmaschine geschrieben werden, indem zum Beispiel dort seine Adjazenzliste geeignet kodiert wird. Daneben müssen dann auch noch die Farben zu den Knoten auf dem Band erscheinen, ggf. in einer zusätzlichen Spur. Intuitiv ist einsichtig, daß eine solche Turingmaschine mit polynomialer Zeitkomplexität angegeben werden kann. □

Wie in (b) werden wir häufig die nach Definition 6.2.2 notwendige Turingmaschine nicht explizit konstruieren, sondern einen mehr oder weniger intuitiven Algorithmus angeben. Nach der These von Church sind wir dann überzeugt, daß es uns mit einigem Aufwand gelingen würde, die nötige Turingmaschine zu konstruieren. Dabei gehen wir davon aus, daß auch die polynomiale Zeitkomplexität erhalten bleibt. Diese Robustheit der Klasse P gegenüber den verschiedenen Rechnermodellen werden wir in Abschnitt 6.4 genauer untersuchen.

Wir haben schon erwähnt, daß das Problem 3-FÄRBBARKEIT wahrscheinlich nicht zu P gehört. Es gibt weitere Probleme, bei denen die Variante mit einer geringeren Anzahl einer gewissen Größe des Problems (wie die Anzahl der Farben beim Färbungsproblem) effizient lösbar ist, jedoch bei größerer Anzahl wahrscheinlich nicht. Wir werden in diesem Zusammenhang das Erfüllbarkeitsproblem der Aussagenlogik betrachten.

Erfüllbarkeitsproblem der Aussagenlogik: Gegeben sei ein logischer Ausdruck mit den Verknüpfungen $\wedge$, $\vee$ und $\neg$. Existiert eine Belegung der Variablen mit 0 oder 1, so daß der gesamte Ausdruck 1, d.h. wahr, wird? □

Dieses Problem ist in seiner allgemeinen Form wahrscheinlich nicht effizient. Diese Annahme folgt aus Satz 7.2.2, wobei wir uns zum Nachweis auf Ausdrücke in kon-

junktiver Normalform beschränken können. In diesem Abschnitt werden wir beweisen, daß ein spezielles solches Problem zu P gehört. In der folgenden Definition werden zunächst einige Bezeichnungen festgelegt.

Definition 6.2.3 Es sei $V = \{x_1, x_2, \ldots\}$ eine unendliche Menge von logischen Variablen.

(a) x_i und $\neg x_i$ heißen *Literale.*

(b) Eine Disjunktion $y_1 \vee \ldots \vee y_k$ heißt *Klausel vom Grad k*, wenn $y_1, \ldots, y_k$ Literale sind.

(c) Eine Konjunktion $c_1 \wedge \ldots \wedge c_m$ heißt *Ausdruck in konjunktiver Normalform mit höchstens k Literalen pro Klausel*, wenn $c_1, \ldots, c_m$ Klauseln vom Grad $\leq k$ sind.

(d) Eine Abbildung $\Psi : V \rightarrow \{0,1\}$ heißt *Belegung der Variablen.* Man kann Ψ auf Literale, Klauseln und Ausdrücke erweitern und erhält damit eine logische Funktion, die bei n Variablen im Ausdruck eine Abbildung $\{0,1\}^n \rightarrow \{0,1\}$ liefert.

(e) Ein Ausdruck α heißt *erfüllbar*, wenn eine Belegung Ψ existiert mit $\Psi(\alpha) = 1$. □

Jeder logische Ausdruck besitzt eine konjunktive Normalform. Dies wollen wir an einem Beispiel verdeutlichen.

Beispiel 6.2.3 Wir betrachten den logischen Ausdruck

$$\alpha = (x_1 \rightarrow x_2 \wedge \neg x_3) \wedge (\neg x_1 \rightarrow \neg x_3).$$

Wir benutzen die Äquivalenz von

$$a \rightarrow b \;\text{ mit }\; \neg a \vee b$$

und erhalten damit den zu α äquivalenten Ausdruck

$$(\neg x_1 \vee (x_2 \wedge \neg x_3)) \wedge (x_1 \vee \neg x_3).$$

Da nach den de Morganschen Regeln $\neg x_1 \vee (x_2 \wedge \neg x_3)$ äquivalent zu $(\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_3)$ ist, erhalten wir damit den Ausdruck

$$\alpha' = (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_1 \vee \neg x_3)$$

in konjunktiver Normalform mit 2 Literalen pro Klausel. Dieser Ausdruck ist erfüllbar, denn mit der Belegung $\Psi : \{x_1, x_2, x_3\} \rightarrow \{0,1\}$, die durch $\Psi(x_1) = 1$ (wahr), $\Psi(x_2) = 1$ (wahr) und $\Psi(x_3) = 0$ (falsch) gegeben ist, erhalten wir $\Psi(\alpha') = 1$. □

Ausdrücke werden in natürlicher Weise kodiert, indem die Indizes der Variablen in Binärdarstellung hinter das Symbol x geschrieben werden, wobei die Indexmenge immer ein Anfangsabschnitt von $\mathbb{N}$ ist. Damit erhält man Ausdrücke in *Standardkodierung.* Der Ausdruck

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$$

wird z.B. durch

$$(x1 \vee \neg x10) \wedge (\neg x1 \vee x10)$$

kodiert.

Definition 6.2.4 SAT sei die Menge der erfüllbaren Ausdrücke in konjunktiver Normalform. Speziell sei SAT(k) die Menge der erfüllbaren Ausdrücke in konjunktiver Normalform mit höchstens k Literalen pro Klausel. □

SAT(2) gehört zur Klasse P. Man kann zum Nachweis dieser Eigenschaft einen effizienten Algorithmus für SAT(2) angeben, also einen Algorithmus, der polynomial in der Länge der Darstellung eines gegebenen Ausdrucks ist. Das wollen wir hier nicht tun. Stattdessen werden wir später in Satz 6.3.6 zeigen, daß es eine effiziente Reduktion dieses Problems auf ein anderes Problem der Klasse P gibt. Daraus können wir dann schließen, daß auch SAT(2) $\in P$ erfüllt sein muß. Dagegen können wir nicht nachweisen, daß SAT oder auch speziell SAT(3) $\in P$ gilt. In Satz 7.2.2 und Satz 7.3.1 werden wir zeigen, daß SAT bzw. SAT(3) zur Klasse der NP-vollständigen Probleme gehören und daher zum Beispiel ebenso schwierig wie 3-FÄRBBARKEIT sind.

Alle bisher erwähnten Probleme haben einen polynomialen Zeitaufwand, oder aber es ist, wie bei 3-FÄRBBARKEIT oder SAT(3), nicht bekannt, ob die Probleme mit polynomialen Aufwand gelöst werden können, obwohl dies nicht sehr wahrscheinlich ist. Zum Abschluß dieses Abschnitts wollen wir ein Problem angeben, für das bewiesen wurde, daß es nicht effizient lösbar ist. Wir betrachten reguläre Ausdrücke (siehe Definition 4.4.1). Über den Symbolen eines Alphabets V werden reguläre Ausdrücke mit Hilfe der Operationen $\cup$, $\cdot$ und $*$ aufgebaut. Wir verwenden hier die komprimierte Notation. Statt zum Beispiel $xx\dots x$ (n-mal) für ein $x \in V$ schreiben wir $x{\uparrow}n$ (n in binärer Darstellung), also etwa $xxxxxx = x{\uparrow}110$. Über dem Alphabet $V = \{a, b\}$ wird zum Beispiel der reguläre Ausdruck

$$(aaa \cup b^*)(aaa \cup b^*)$$

in komprimierter Notation durch

$$(a{\uparrow}11 \cup b^*){\uparrow}10$$

dargestellt.

Definition 6.2.5 Ein regulärer Ausdruck über einem Alphabet V heißt *universell*, wenn die durch ihn bezeichnete Sprache gleich V^* ist. □

Der Beweis des folgenden Satzes findet sich in [11], S. 383–386.

Satz 6.2.2 Das Entscheidungsproblem, ob ein gegebener regulärer Ausdruck in komprimierter Notation über dem Alphabet $\{a, b\}$ universell ist, gehört nicht zu P. □

Man kann beweisen, daß jede Turingmaschine T, die das Problem löst, eine Zeitkomplexität $K_T(n) \geq 2^{\frac{cn}{\log n}}$ für eine geeignete Konstante c hat. Für beliebig große Zahlen n gibt es also reguläre Ausdrücke in komprimierter Notation dieser Länge, für die die Berechnung mindestens die angegebene Anzahl von Schritten erfordert. Damit liegt jedoch keine polynomiale Zeitkomplexität vor, denn es gilt

$$\lim_{n\to\infty} \frac{2^{\frac{cn}{\log n}}}{n^k} = \infty.$$

6.3 Berechnungsprobleme und Reduzierbarkeit

Die Klasse P aus dem vorhergehenden Abschnitt wurde für Entscheidungsprobleme eingeführt. Wir haben jedoch schon mehrere Algorithmen kennengelernt, die in polynomialer Zeit in bezug auf die Länge der Eingabe einen Wert berechnen können, zum Beispiel die Multiplikation. Wir wollen im folgenden die Klasse *FP* der Berechnungsprobleme einführen, die alle in polynomialer Zeit berechenbaren Probleme enthält. Die jeweiligen Fälle der Probleme und die berechneten Werte müssen in geeigneter Weise kodiert werden.

Beispiel 6.3.1 2-FÄRBBARKEIT ist als ein Entscheidungsproblem aus P betrachtet worden. Andererseits können wir mit Algorithmus 6.1.3 eine 2-Färbung berechnen, und zwar in linearer Zeit (siehe Satz 6.1.3). Die Berechnung der 2-Färbung kann zum Beispiel durch eine Funktion $f : \{0,1\}^* \to \{0,1\}^*$ erfolgen, die jedem binären Kode $c(G)$ des Graphen den Kode seiner 2-Färbung $f(c(G))$ zuordnet. □

Als Berechnungsmodell solcher Funktionen verwenden wir wieder deterministische Turingmaschinen. Der Begriff einer Rechnung wird wie in Definition 4.5.2 verwendet.

Definition 6.3.1 Es sei $T = (Z, X, \delta, z_0)$ eine deterministische Turingmaschine, und es gelte $\Sigma \subset X$, $\Gamma \subset X$. Es sei $f : \Sigma^* \to \Gamma^*$ eine totale Abbildung. T *berechnet* f, wenn für alle $w \in \Sigma^*$ die Bandinschrift der Endkonfiguration der Rechnung von T an w durch $\ldots b f(w) \underline{b} \ldots$ gegeben ist. □

Wir erinnern daran, daß nach Definition 4.5.2 der Kopf von T zu Beginn der Rechnung über dem ersten Symbol von w steht.

Definition 6.3.2 Es ist

$FP = \{f : \Sigma^* \to \Gamma^* \mid$ es existiert eine deterministische Turingmaschine T, die f berechnet und für alle $w = x_1 \ldots x_n$, $x_1, \ldots, x_n \in \Sigma$, $n \in \mathbb{N}_0$, in höchstens $p_T(n)$ Schritten, p_T ein Polynom, hält$\}$

die *Klasse aller totalen Funktionen, die in polynomialer Zeit berechnet werden.* □

Nach den Überlegungen aus Beispiel 6.3.1 und den Bemerkungen in Satz 6.2.1 gilt also

Satz 6.3.1 2-FÄRBBARKEIT $\in$ *FP* (das Problem, für einen Graphen eine 2-Färbung anzugeben). □

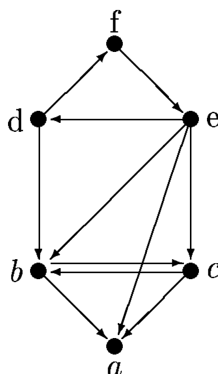
Man beachte, daß wir mit 2-FÄRBBARKEIT sowohl das Entscheidungsproblem als auch das Berechnungsproblem bezeichnet haben.

Zwei Knoten v, v' eines gerichteten Graphen können genau dann in beiden Richtungen durch einen gerichteten Weg verbunden werden, wenn sie auf einem gerichteten Kreis liegen. Man sagt dann, daß sie in derselben starken Komponente von G liegen.

Definition 6.3.3 Es sei $G = (V, E)$ ein gerichteter Graph. Eine *starke Komponente* von G ist eine Menge K von Knoten von G mit der Eigenschaft, daß je zwei Knoten

$v, v' \in K$ auf einem gemeinsamen gerichteten Kreis von G liegen und K maximal bezüglich dieser Eigenschaft ist, das heißt, ein Knoten $v'' \in V - K$ liegt mit keinem Knoten von K auf demselben Kreis. □

Beispiel 6.3.2 Der Graph G



besitzt drei starke Komponenten, und zwar $\{e, d, f\}$, $\{b, c\}$ und $\{a\}$. Bei $\{a\}$ hat der zugehörige Kreis die Länge 0. □

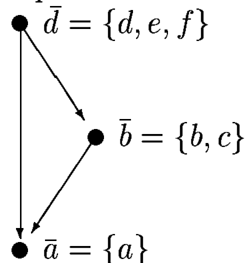
Mit STARKE KOMPONENTEN bezeichnen wir das Problem, die starken Komponenten eines Graphen zu berechnen.

Satz 6.3.2 STARKE KOMPONENTEN $\in$ *FP*.

Beweis: Wir skizzieren einen Algorithmus, der für einen Graphen G mit n Knoten und k Kanten die starken Komponenten in der Zeit $O(n^2(n+k))$ berechnet. Es sei v ein beliebiger Knoten. Für jeden weiteren Knoten w überprüfen wir mit Algorithmus 6.1.4 nach Satz 6.1.4 in der Zeit $O(n+k)$, ob ein gerichteter Weg von v nach w führt. Ist das der Fall, so überprüfen wir entsprechend, ob es umgekehrt auch einen Weg von w nach v gibt. Auf diese Weise wird in der Zeit $O(n(n+k))$ die starke Komponente von G bestimmt, in der auch v liegt. Insgesamt kann es höchstens n starke Komponenten geben, so daß der Gesamtzeitbedarf von der Ordnung $O(n^2(n+k))$ ist. □

Der im Beweis skizzierte Algorithmus ist nicht sehr geschickt gewählt. Man kann einen Algorithmus angeben (siehe Algorithm 3.8 in [2], p. 162), der die starken Komponenten in linearer Zeit $O(n+k)$ berechnet.

Zwischen topologischem Sortieren (Algorithmus 6.1.2) und starken Komponenten gibt es einen Zusammenhang, den wir später im Beweis von Satz 6.3.6 benutzen werden. Für jeden gerichteten Graphen G bezeichnen wir mit $\bar{G}$ den Graphen, den wir aus G erhalten, wenn seine Knoten $\bar{v}$ die starken Komponenten der Knoten v von G sind. Außerdem gibt es für $\bar{v} \neq \bar{w}$ eine Kante $(\bar{v}, \bar{w})$ in $\bar{G}$ genau dann, wenn Knoten $v' \in \bar{v}$, $w' \in \bar{w}$ mit einer Kante (v', w') in G existieren. Zum Beispiel erhalten wir zum Graphen G aus Beispiel 6.3.2 den Graphen $\bar{G}$ mit der Darstellung



Allgemein ist der neue Graph $\bar{G}$ zyklenfrei, denn anderenfalls könnte ein Kreis in $\bar{G}$ zu einem Kreis in G erweitert werden, der Knoten aus mehreren starken Komponenten enthalten würde, ein Widerspruch. Mit Hilfe von Algorithmus 6.1.2 können die starken Komponenten eines jeden gerichteten Graphen $G = (V, E)$ so sortiert werden, daß

$$(v, w) \in E \Longrightarrow \text{ord}(\bar{v}) \leq \text{ord}(\bar{w})$$

gilt.

Die Zugehörigkeit zur Klasse P haben wir im vorhergehenden Abschnitt dadurch bewiesen, daß wir einen Polynomialzeitalgorithmus zur Lösung des Problems angegeben haben. Eine andere Möglichkeit besteht in der Reduktion des Problems auf ein anderes Problem, das bereits zu P gehört. Dazu benötigen wir die folgende Definition.

Definition 6.3.4 Es seien Σ, Γ Alphabete und $L_1 \subset \Sigma^*$, $L_2 \subset \Gamma^*$. Die Sprache L_1 ist *in polynomialer Zeit* auf die Sprache L_2 *reduzierbar* (in Zeichen: $L_1 \leq L_2$), wenn eine (totale) Funktion $f : \Sigma^* \to \Gamma^*$ aus *FP* existiert, so daß

$$w \in L_1 \iff f(w) \in L_2 \quad \text{für alle } w \in \Sigma^*$$

erfüllt ist. □

Wichtig in dieser Definition ist die Forderung $f \in FP$. Ohne diese Forderung ist es nämlich für $L_2 \neq \Gamma^*$ trivial, eine Funktion f zu definieren, die die Äquivalenz erfüllt. Wir wählen einfach zwei Wörter $w \in L_2$ und $w' \in \Gamma^* - L_2$ und setzen

$$f(u) = \begin{cases} w & \text{falls } u \in L_1 \\ w' & \text{falls } u \notin L_1. \end{cases}$$

Um ein erstes Beispiel einer Reduktion nach Definition 6.3.4 anzugeben, benötigen wir zunächst den Begriff eines dualen Graphen.

Definition 6.3.5 Es sei $G = (V, E)$ ein ungerichteter Graph. Wir setzen

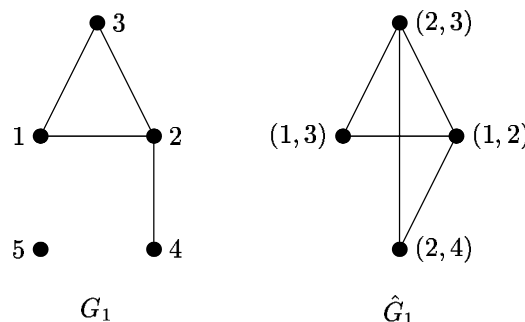
$H = \{((v_1, v_2), (v_1', v_2')) \mid (v_1, v_2), (v_1', v_2') \in E, \{v_1, v_2\} \neq \{v_1', v_2'\}, \{v_1, v_2\} \cap \{v_1', v_2'\} \neq \emptyset\}$.

Dann ist $\hat{G} = (E, H)$ der *duale Graph* von G. □

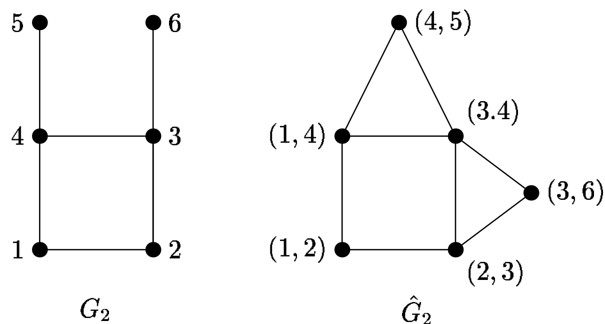
Die Definition bedeutet, daß eine Kante von G zu einem Knoten in $\hat{G}$ wird, während die Kanten von $\hat{G}$ durch benachbarte Kanten von G gegeben sind.

Beispiel 6.3.3 Wir geben zwei Beispiele an:

(a)



(b)



Wir definieren ein Problem, das in gewisser Weise dual zum Entscheidungsproblem 2-FÄRBBARKEIT ist.

Definition 6.3.6 Das Problem KANTEN-2-FÄRBBARKEIT ist wie folgt gegeben: Es sei G ein ungerichteter Graph. Existiert eine Färbung der Kanten mit zwei verschiedenen Farben, so daß benachbarte Kanten nicht gleich gefärbt sind? □

Wir erkennen sofort, daß im Graphen G_1 aus Beispiel 6.3.3 die Kanten und in $\hat{G}_1$ die Knoten nicht 2-färbbar sind. In G_2 sind die Kanten 2-färbbar und in $\hat{G}_2$ die Knoten. Definition 6.3.5 zeigt sofort, daß allgemein die Kanten eines ungerichteten Graphen genau dann 2-färbbar sind, wenn es die Knoten seines dualen Graphen sind. Diese Eigenschaft nutzen wir im folgenden Satz aus.

Satz 6.3.3 Es gilt KANTEN-2-FÄRBBARKEIT $\leq$ 2-FÄRBBARKEIT.

Beweis: Nach den vorstehenden Bemerkungen müssen wir für den Nachweis von Definition 6.3.4 nur beweisen, daß es eine Funktion f der Klasse *FP* gibt, die einen Graphen G (bzw. seine Kodierung) in seinen dualen Graphen $\hat{G}$ überführt. Ist ein Graph G mit n Knoten durch seine Adjazenzmatrix $A = (a_{i,j})$, $1 \leq i,j \leq n$, dargestellt, so hat der duale Graph $\hat{G}$ die Adjazenzmatrix $f(A) = \hat{A}$, deren Zeilen und Spalten mit genau den Paaren (i,j) mit $1 \leq i \leq j \leq n$ und $a_{i,j} = 1$ indiziert sind. Dabei gilt $\hat{a}_{(i,j),(i',j')} = 1$, wenn $\{i,j\} \neq \{i',j'\}$ und $\{i,j\} \cap \{i',j'\} \neq \emptyset$ ist. Diese Matrix kann in der Zeit $O(m^2)$ aufgestellt werden, wenn $m = O(n^2)$ die Größe der eingegebenen Adjazenzmatrix ist. Mit einer Turingmaschine kann diese Konstruktion ebenfalls in polynomialer Zeit durchgeführt werden. □

Da nach Satz 6.2.1(b) 2-FÄRBBARKEIT $\in P$ gilt, zeigt der folgende Satz, daß auch KANTEN-2-FÄRBBARKEIT zu P gehört.

Satz 6.3.4 Es gelte $L_1 \leq L_2$ und $L_2 \in P$. Dann folgt $L_1 \in P$.

Beweis: Wegen $L_2 \in P$ existiert eine deterministische Turingmaschine D_2, die L_2 in Polynomialzeit akzeptiert. Durch das Polynom p sei eine obere Schranke für die Laufzeit von D_2 gegeben. Wegen $L_1 \leq L_2$ existiert eine Funktion $f \in FP$, für die $w \in L_1$ genau dann gilt, wenn $f(w) \in L_2$ ist. Weiter sei D_0 eine deterministische

Turingmaschine, die f gemäß Definition 6.3.1 berechnet. Eine obere Schranke für die Laufzeit von D_0 werde durch das Polynom q bestimmt. Wir konstruieren nun eine neue deterministische Turingmaschine

$$D: \quad D_0 \to \mathbf{Lr} \to D_2,$$

wobei **L** die große Linksmaschine und **r** die Rechtsmaschine ist. Wir verlangen, daß D die gleiche Endzustandsmenge wie D_2 hat. Die Turingmaschine **Lr** dient dazu, nach der Berechnung von $f(w)$ gemäß D_0 den Kopf von D vom Feld unmittelbar rechts von $f(w)$ zum Feld mit dem ersten Symbol von $f(w)$ zu bewegen. Offenbar akzeptiert D die Eingabe w genau dann, wenn D_2 den Wert $f(w)$ akzeptiert. Daraus folgt, daß L_1 von D erkannt wird. Die Laufzeit von D ist

$$\leq q(|w|) + |f(w)| + 2 + p(|f(w)|).$$

Um eine Schranke für $|f(w)|$ zu bestimmen, überlegen wir uns, daß die Turingmaschine D_2, die mit dem ersten Nichtblankzeichen von $\ldots bwb \ldots$ startet und mit der Bandinschrift $\ldots bf(w)\underline{b}\ldots$ stoppt, jedes Feld von $f(w)$ mindestens einmal betreten muß. Folglich ist

$$|f(w)| \leq q(|w|).$$

Die Laufzeit von D ist also

$$\leq q(|w|) + q(|w|) + 2 + p(q(|w|)) \leq r(|w|)$$

mit einem geeigneten Polynom r. Somit ist $L_1 \in P$. $\square$

Der folgende Satz zeigt, daß die Relation $\leq$ aus Definition 6.3.4 transitiv ist.

Satz 6.3.5 Es seien L_1, L_2 und L_3 Sprachen mit $L_1 \leq L_2$, $L_2 \leq L_3$. Dann folgt $L_1 \leq L_3$.

Beweis: Nach Definition 6.3.4 existiert wegen $L_1 \leq L_2$ eine Transformation f, für die $w \in L_1$ genau dann gilt, wenn $f(w) \in L_2$ ist. Wegen $L_2 \leq L_3$ existiert weiter eine Transformation g, für die $w' \in L_2$ genau dann gilt, wenn $g(w') \in L_3$ ist. Insgesamt erhalten wir

$$w \in L_1 \Longleftrightarrow g(f(w)) \in L_3,$$

d.h., $g \circ f$ transformiert L_1 auf L_3. Dabei bleibt zu zeigen, daß $g \circ f$ in Polynomialzeit berechenbar ist. Es seien T_f und T_g die Turingmaschinen, die f bzw. g in Polynomialzeit berechnen. Dann wird eine deterministische Turingmaschine, die $g \circ f$ berechnet, durch

$$D': \quad T_f \to \mathbf{Lr} \to T_g$$

gegeben. Ihre Laufzeitabschätzung wird ähnlich wie die für D im Beweis von Satz 6.3.4 durchgeführt. $\square$

Auch STARKE KOMPONENTEN kann man als ein Entscheidungsproblem auffassen, indem neben dem Graphen $G = (V, E)$ eine endliche Menge $F \subset V \times V$ als ein Fall des Problems vorgegeben wird. Es wird gefragt, ob für alle $(v, v') \in F$ die Knoten v und v' in verschiedenen starken Komponenten von G liegen. Da das Problem als Berechnungsproblem zu FP gehört, ist es klar, daß es als Entscheidungsproblem zu P gehört. Zum Abschluß dieses Abschnitts werden wir zeigen, daß SAT(2) auf STARKE KOMPONENTEN reduziert werden kann.

Satz 6.3.6 Es gilt SAT(2) $\leq$ STARKE KOMPONENTEN und somit auch SAT(2) $\in P$.

Beweis: Für jeden Ausdruck φ in konjunktiver Normalform mit zwei Literalen pro Klausel konstruieren wir einen gerichteten Graphen $G_\varphi = (V_\varphi, E_\varphi)$ mit

$$V_\varphi = \{x, \neg x \mid x \text{ Variable von } \varphi\}$$

und

$$E_\varphi = \{(\neg\alpha, \beta), (\neg\beta, \alpha) \mid \alpha \vee \beta \text{ Klausel von } \varphi\},$$

außerdem wird die Menge $F_\varphi \subset V_\varphi \times V_\varphi$ durch

$$F_\varphi = \{(x, \neg x) \mid x \text{ Variable von } \varphi\}$$

gegeben. Wir merken zunächst an, daß wegen der logischen Äquivalenz der Klausel $\alpha \vee \beta$ mit den Implikationen $\neg\alpha \Longrightarrow \beta$ sowie $\neg\beta \Longrightarrow \alpha$ jede Kante $(\alpha', \beta') \in E_\varphi$ eine Implikation $\alpha' \Longrightarrow \beta'$ repräsentiert. Wir zeigen, daß diese Konstruktion eine Reduktion in polynomialer Zeit ist, daß also

(a) die Funktion f, die jedem Ausdruck φ den Graphen G_φ und damit auch die Menge F_φ zuordnet, eine polynomiale (sogar lineare) Zeitkomplexität besitzt und

(b) φ genau dann erfüllbar ist, wenn x und $\neg x$ für alle Variablen x von φ in verschiedenen starken Komponenten von G_φ liegen.

(a) ist erfüllt, denn falls n die Länge des Ausdrucks φ ist, so benötigt die Konstruktion von G_φ in der Darstellung durch Adjazenzlisten und die Konstruktion von F_φ die Zeit $O(n)$.

Wir beweisen (b). Wir setzen zunächst voraus, daß eine Belegung Ψ der Variablen existiert, so daß jede Klausel $\alpha\vee\beta$ von φ den Wert 1 besitzt. Das bedeutet, daß für jede Kante $(\neg\alpha, \beta) \in E_\varphi$ wegen $\Psi(\alpha \vee \beta) = 1$ aus $\Psi(\neg\alpha) = 1$ auch die Beziehung $\Psi(\beta) = 1$ folgen muß. Das entsprechende gilt für Kanten $(\neg\beta, \alpha)$. Wenn somit $\Psi(x) = 1$ gilt, kann wegen $\Psi(\neg x) = 0$ kein gerichteter Weg von x nach $\neg x$ führen. Gilt dagegen $\Psi(x) = 0$, dann kann es wegen $\Psi(\neg x) = 1$ keinen gerichteten Weg von $\neg x$ nach x geben. In beiden Fällen liegen x und $\neg x$ in verschiedenen starken Komponenten.

Umgekehrt nehmen wir an, daß x und $\neg x$ für alle Variablen von φ in verschiedenen starken Komponenten von G_φ liegen. Nach den Überlegungen im Anschluß an Satz 6.3.2 können die starken Komponenten $\bar{v}$ der Knoten v von G_φ so sortiert werden, daß

$$(v, w) \in E_\varphi \Longrightarrow \operatorname{ord}(\bar{v}) \leq \operatorname{ord}(\bar{w})$$

gilt. Wir belegen die Variablen von φ durch

$$\Psi(x) = 1 \iff \operatorname{ord}(\overline{\neg x}) < \operatorname{ord}(\bar{x}).$$

Wir müssen beweisen, daß für jede Klausel $\alpha \vee \beta$ von φ die Beziehung $\Psi(\alpha) = 1$ oder $\Psi(\beta) = 1$ gilt, insgesamt also $\Psi(\varphi) = 1$ erfüllt ist. Wir nehmen $\Psi(\alpha) = 0$ an. Nach der Definition der Belegung Ψ folgt

$$\operatorname{ord}(\bar{\alpha}) \leq \operatorname{ord}(\overline{\neg\alpha}).$$

Wegen der Existenz der Klausel $\alpha \vee \beta$ enthält der Graph G_φ die Kanten $(\neg\alpha, \beta)$ und $(\neg\beta, \alpha)$. Somit erhalten wir die Relationen

$$\mathrm{ord}(\overline{\neg\alpha}) \leq \mathrm{ord}(\bar{\beta}) \text{ und } \mathrm{ord}(\overline{\neg\beta}) \leq \mathrm{ord}(\bar{\alpha}).$$

Wir schließen

$$\mathrm{ord}(\overline{\neg\beta}) \leq \mathrm{ord}(\bar{\alpha}) \leq \mathrm{ord}(\overline{\neg\alpha}) \leq \mathrm{ord}(\bar{\beta}).$$

Da β und $\neg\beta$ in verschiedenen Komponenten liegen, ist $\overline{\neg\beta} \neq \bar{\beta}$. Es folgt

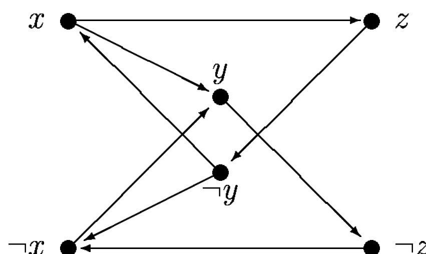
$$\mathrm{ord}(\overline{\neg\beta}) < \mathrm{ord}(\bar{\beta}).$$

Die Definition der Belegung liefert daraus $\Psi(\beta) = 1$. □

Beispiel 6.3.4 Wir betrachten den Ausdruck

$$\varphi = (x \vee y) \wedge (\neg x \vee z) \wedge (\neg z \vee \neg y) \wedge (\neg x \vee y),$$

der den Graphen



liefert. Wir sehen, daß wir zwei starke Komponenten erhalten, zwischen denen es die Kanten (x, y) und $(\neg y, \neg x)$ gibt. Folglich werden die starken Komponenten durch $\mathrm{ord}(\bar{x}) < \mathrm{ord}(\overline{\neg x})$ topologisch sortiert. Die Belegungsfunktion liefert

$$\Psi(x) = 0,\ \Psi(y) = 1,\ \Psi(z) = 0$$

und damit eine erfüllende Belegungen von φ. Entfernt man die letzte Klausel aus φ, so müssen auch die Kanten (x, y) und $(\neg y, \neg x)$ entfernt werden. Dann gibt es keine Kanten zwischen den starken Komponenten. Neben der angegebenen topologischen Ordnung und Belegung ist dann auch noch $\mathrm{ord}(\overline{\neg x}) < \mathrm{ord}(\bar{x})$ mit

$$\Psi(x) = 1,\ \Psi(y) = 0,\ \Psi(z) = 1$$

möglich. □

6.4 Die Robustheit der Klassen P und FP

Die Definitionen der Klassen P und FP waren auf dem Modell der Turingmaschine aus Definition 2.1.1 aufgebaut. Wir werden hier zeigen, daß wir dieselbe Klasse erhalten, wenn wir von dem Modell der k-Band-Turingmaschinen aus Abschnitt 2.3(d) ausgehen. Dasselbe gilt auch, wenn wir Registermaschinen zugrunde legen, wobei die Akzeptierung einer Sprache durch das Berechnen der zugehörigen charakteristischen

Funktion erfolgt und die Symbole eines allgemeinen Alphabets durch Zahlen kodiert werden.

Eine k-Band-Turingmaschine hat k Bänder mit k voneinander unabhängigen Lese- und Schreibköpfen, jedoch mit einem gemeinsamen Zustandsteil. Aufgrund des Zustands und der Gesamtheit der gelesenen Symbole wird der Zustand geändert und jeweils eine Aktion auf jedem der k Bänder durchgeführt. Das erste Band kann als Eingabeband aufgefaßt werden. Dabei wollen wir annehmen, daß der erste Kopf in Übereinstimmung mit den Definitionen 4.5.2 und 6.3.1 über dem ersten Symbol des eingegebenen Wortes steht, während die anderen Bänder mit lauter Blankzeichen beschrieben sind. Bei einer eine Funktion berechnenden Turingmaschine befindet sich am Ende die Ausgabe auf dem letzten Band, wobei der k-te Kopf auf dem letzten Blankzeichen hinter dem Funktionswert steht.

Satz 6.4.1 Es sei $k \in \mathbb{N}$. Jede deterministische k-Band-Turingmaschine T mit einer Zeitkomplexität $K_T(n) \geq n$ kann durch eine deterministische 1-Band Turingmaschine T' mit $K_{T'}(n) = O((K_T(n))^2)$ simuliert werden.

Beweis: Da die k-Band-Turingmaschine T für eine Eingabe der Länge n höchstens $K_T(n)$ Schritte durchführt, kann sie auch die Köpfe im Laufe der Arbeit höchstens $K_T(n)$ Felder nach links oder rechts von den initialen Positionen ausgehend bewegen. Wir simulieren T mit einer 1-Band-Turingmaschine T' mit $2k$ Spuren, wie es in Abschnitt 2.3(d) beschrieben worden ist. Zu Beginn stehen alle k Kopfmarkierer in den entsprechenden Komponenten des Feldes 0. Die Simulation eines Schrittes von T erfordert einen Lauf vom linken Kopf zum rechten Kopf unter Aufnahme der von den Köpfen jeweils gelesenen Symbole in den Zustand und anschließend einen Lauf vom rechten Kopf zum linken Kopf, um aufgrund der Gesamtinformation die Aktion der jeweiligen Köpfe auszuführen. Für diese Ausführung werden jeweils höchstens eine konstante Anzahl von Schritten, also $O(1)$ Schritte, benötigt. Das Hin- und Herlaufen erfordert $O(K_T(n))$ Schritte, so daß ein einzelner Schritt von T durch T' in $O(K_T(n))$ Schritten simuliert wird. Da T insgesamt höchstens $K_T(n)$ Schritte durchführt, hat T' eine Zeitkomplexität von $O((K_T(n))^2)$. □

Aufgrund des zu Anfang dieses Abschnitts angegebenen Akzeptier- und Berechnungsverhaltens von k-Band-Turingmaschinen erhalten wir

Satz 6.4.2 Es sei L eine Sprache und $f : \Gamma^* \to \Sigma^*$ eine Funktion.

(a) Die Sprache L wird genau dann von einer deterministischen k-Band-Turingmaschine in polynomialer Zeit akzeptiert, wenn sie von einer deterministischen 1-Band-Turingmaschine in polynomialer Zeit akzeptiert wird.

(b) Die Funktion f wird genau dann von einer deterministischen k-Band-Turingmaschine in polynomialer Zeit berechnet, wenn sie von einer deterministischen 1-Band-Turingmaschine in polynomialer Zeit berechnet wird.

Beweis: Falls $K_T(n)$ ein Polynom ist, ist auch $(K_T(n))^2$ ein Polynom. □

Die Definitionen von P (siehe Definition 6.2.2) und FP (siehe Definition 6.3.2) würden sich also nicht ändern, wenn wir statt deterministischer 1-Band-Turingmaschinen deterministische k-Band-Turingmaschinen benutzt hätten.

In Abschnitt 2.8 haben wir Registermaschinen (RAM) als ein realistischeres Berechnungsmodell als das der Turingmaschinen eingeführt. In Satz 2.8.1 haben wir bewiesen, daß Turingmaschinen und Registermaschinen dieselben Funktionen berechnen. Um zu zeigen, daß mit Registermaschinen als Modell sich ebenfalls die Klassen P und FP ergeben, müssen wir zunächst klären, was Zeitkomplexität für Registermaschinen bedeutet. In der Literatur werden zwei Varianten betrachtet. Bei der *uniformen Zeitkomplexität* entspricht wie bei Turingmaschinen jede Durchführung eines Befehls einem Schritt. Diese Variante ist allerdings nicht sehr realistisch, denn in einem Feld einer RAM können beliebig große Zahlen dargestellt werden. So kann zum Beispiel die ganze Eingabe als eine sehr große Zahl i kodiert und in einem Feld gespeichert werden. In diesem Fall hat die Eingabe einfach die Größe 1. Deshalb hat die uniforme Zeitkomplexität nur geringe theoretische Bedeutung.

Bei der *logarithmischen Zeitkomplexität* wird dagegen die Größe der Darstellung der Zahlen in Betracht gezogen. In Dualzahldarstellung hat eine Zahl n die Größe $\lceil \log_2(n+1) \rceil$. Jeder Befehl einer RAM erfordert, abhängig von der Größe seiner Argumente, unterschiedlich viele Schritte. Zum Beispiel benötigt der Befehl

$$\text{LOAD } i$$

$\lceil \log_2(i+1) \rceil + \lceil \log_2(n+1) \rceil$ Schritte, wobei $r(i) = n$ für den Inhalt $r(i)$ des Registers R_i gilt. Zur Ausführung des Befehls

$$\text{LOAD } *i$$

sind $\lceil \log_2(i+1) \rceil + \lceil \log_2(n+1) \rceil + \lceil \log_2(m+1) \rceil$ Schritte erforderlich mit $r(i) = n$ und $r(n) = m$.

Wenn die Registermaschinen eine Funktion $f : \Sigma^* \to \Gamma^*$ wie in Definition 6.3.1 berechnen sollen, dann muß die RAM-Berechenbarkeit aus Definition 2.8.3 entsprechend abgeändert werden. Das bedeutet, daß bei einer Eingabe $w = a_1 \dots a_n$ mit $a_i \in \Sigma$, $i = 1, \dots, n$, $n \in \mathbb{N}$, diese auf den ersten n Feldern des Eingabebandes steht, wobei jedes Symbol a_i als eine Zahl kodiert wird. Entsprechend findet sich schließlich die Ausgabe $f(w)$ auf den ersten $|f(w)|$ Feldern des Ausgabebandes. Auf die durch die endlich vielen Symbole des Alphabets X veranlaßten Schritte bei der Abarbeitung eines einzelnen RAM-Befehls entfällt bei logarithmischer Zeitkomplexität jeweils nur ein konstanter Zeitanteil $O(1)$.

Satz 6.4.3 Jede deterministische Turingmaschine T mit einer Zeitkomplexität $K_T(n) \geq n$ kann durch eine Registermaschine R mit der logarithmischen Zeitkomplexität $O(K_T(n) \cdot \log K_T(n))$ simuliert werden. Umgekehrt habe eine Registermaschine R die logarithmische Zeitkomplexität $K_R(n) \geq n$. Dann kann sie durch eine deterministische 6-Band-Turingmaschine T mit der Zeitkomplexität $O((K_R(n))^2)$ simuliert werden.

Beweis: Wir müssen zeigen, daß die gegenseitigen Simulierungen von Turingmaschine und RAM aus Satz 2.8.1 in der angegebenen Zeit möglich ist. Wir gehen zunächst von einer Funktion $f : \Sigma^* \to \Gamma^*$ aus, die von einer deterministischen Turingmaschine $T = (Z, X, \delta, z_0)$ gemäß Definition 6.3.1 berechnet wird. Wir konstruieren dazu die RAM R aus der ersten Hälfte von Satz 2.8.1 mit den Änderungen bei Ein- und

Ausgabe, wie es oben beschrieben wurde. Das bedeutet, daß bei einer Eingabe w die Initialisierung des Speichers mit $O(|w|)$ RAM-Befehlen und die Ausgabe mit $O(|f(w)|$ RAM-Befehlen erfolgt. Die Simulation der eigentlichen Arbeit der Turingmaschine T benötigt offenbar $O(K_T(|w|))$ RAM-Befehle. Insgesamt sind also $O(K_T(|w|))$ RAM-Schritte erforderlich. Die Zugriffe der Befehle auf die Kodierungen der endlich vielen Elemente von $\bar{X}$ und Z erfolgt jeweils in konstanter Zeit $O(1)$. Daneben können in den Registern von R noch Zahlen vorkommen, die die Nummern der Felder der zu simulierenden Turingmaschine enthalten. Da nur $O(K_T(|w|))$ Felder von T besucht werden, hat die Darstellung dieser Zahlen die Ordnung $O(\log K_T(|w|))$. Ein einzelner RAM-Befehl erfordert folglich höchstens $O(\log K_T(|w|))$ Einzelschritte. Somit erfolgt die Simulierung der Turingmaschine T durch die RAM bei einer Eingabe der Länge n mit der Zeitkomplexität $O(K_T(n) \cdot \log K_T(n))$.

Für die umgekehrte Richtung müssen wir beweisen, daß die Simulierung einer RAM R durch eine 6-Band-Turingmaschine T aus dem Beweis von Satz 2.8.1 mit der Zeitkomplexität $O((K_R(n))^2)$ erfolgt. Die Konstruktion von Satz 2.8.1 wird dabei so abgeändert, daß die vorkommenden Zahlen auf den Bändern von T als Dualzahlen dargestellt werden. Für jede Eingabe der Länge n dauert die Berechnung von R höchstens $K_R(n)$ Schritte, wobei höchstens $K_R(n)$ RAM-Befehle durchgeführt werden. Wir müssen zeigen, daß jeder dieser Befehle in höchstens $O(K_R(n))$ Schritten durch die Turingmaschine T simuliert werden kann. Dann folgt, daß die ganze Berechnung von T nur $O((K_R(n))^2)$ Schritte benötigt.

Exemplarisch betrachten wir die indirekte Adressierung

$$\text{STORE} * i,$$

deren Simulierung wir auch im Beweis von Satz 2.8.1 ausführlich beschrieben haben. Zuerst sucht Kopf 2 von rechts kommend das erste Vorkommen von $bbibj$ auf Band 2. Da i eine feste Zahl des Programms ist, kann bei diesem Lauf durch entsprechende Zustandsänderung festgestellt werden, wann eine solche Zahl auf dem Band vorliegt. Da auf jedem Band von T höchstens $O(K_R(n))$ Felder beschriftet werden, erfordert diese Suche $O(K_R(n))$ Schritte. Ist ein solches Vorkommen gefunden, dann wird zunächst das Hilfsband 5 gelöscht und anschließend j durch gleichzeitige Arbeit der Köpfe 2 und 5 auf Band 5 geschrieben sowie der Befehlszähler um 1 erhöht. Dazu werden ebenfalls $O(K_R(n))$ Schritte benötigt, denn j hat $O(K_R(n))$ Ziffern, und es gibt höchstens eine feste Zahl von verschiedenen Befehlen. Wir sehen also, daß STORE $* i$ in $O(K_R(n))$ Schritten simuliert werden kann. Die Zeitkomplexität der anderen Schritte läßt sich analog abschätzen. □

Aus den Sätzen 6.4.2 und 6.4.3 folgt

Satz 6.4.4 Es sei L eine Sprache und $f : \Gamma^* \to \Sigma^*$ eine Funktion.

(a) Die Sprache L wird genau dann von einer deterministischen Turingmaschine in polynomialer Zeit akzeptiert, wenn sie von einer Registermaschine in polynomialer logarithmischer Zeit akzeptiert wird.

(b) Die Funktion f wird genau dann von einer deterministischen Turingmaschine in polynomialer Zeit berechnet, wenn sie von einer Registermaschine in polynomialer logarithmischer Zeit berechnet wird. □

Wir sehen, daß die Klassen P und FP unabhängig davon sind, ob wir bei der Definition das Modell einer normalen deterministischen Turingmaschine, einer deterministischen k-Band-Turingmaschine oder einer Registermaschine wählen. Da die Registermaschinen der Realität schon recht nahe sind, macht es diese Robustheit in bezug auf das gewählte Berechnungsmodell plausibel, daß auch Probleme oder Funktionen, die durch intuitiv beschriebene Algorithmen in (intuitiver) polynomialer Zeit entschieden bzw. berechnet werden können, ebenfalls zur Klasse P bzw. FP gehören. Man ist davon überzeugt, daß man auch in diesen Fällen eine deterministische Turingmaschine mit polynomialem Zeitbedarf für eine Lösung konstruieren kann.

6.5 Effiziente geometrische Algorithmen

Viele Probleme, die in der ebenen oder dreidimensionalen Geometrie, insbesondere im Zusammenhang mit der Computergraphik, entstehen, gehören zu den Klassen P oder FP, falls man für die üblichen Rechnungen mit reelen Zahlen konstanten Zeitbedarf ansetzt (siehe Seite 146). In diesem Abschnitt wollen wir aus dem weiten Feld der möglichen Probleme einige Beispiele aus dem Bereich der konvexen Polygone betrachten.

Wir arbeiten hier in der Ebene, wobei *Punkte* als Paare $p = (x, y)$ mit $x, y \in I\!R$ dargestellt werden. Eine *Strecke* $L(p, q)$ wird durch ein Paar von Punkten repräsentiert.

Definition 6.5.1 (a) $P = (v_0, v_1, \ldots, v_{n-1})$ mit $n \in I\!N$, $n \geq 3$, heißt Polygon, wenn $v_0, \ldots, v_{n-1}$ Punkte sind, die als *Ecken* von P bezeichnet werden.

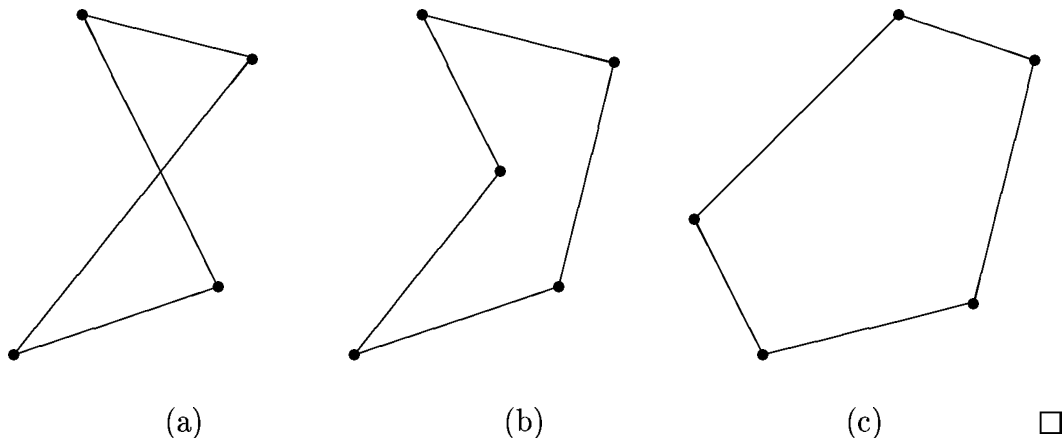
(b) Ein Polygon $P = (v_0, \ldots, v_{n-1})$ heißt *einfach*, wenn sich die Strecken $L(v_i, v_{i+1})$ und $L(v_j, v_{j+1})$, $0 \leq i < j \leq n-1$ (Indizes werden mod n berechnet) nur für $j = i + 1$ oder $i = 0, j = n - 1$ schneiden, und zwar in der gemeinsamen Ecke.

(c) Es sei P ein einfaches Polygon. Werden die Strecken $L(v_0, v_1), \ldots, L(v_{n-2}, v_{n-1}), L(v_{n-1}, v_0)$ aus der Ebene entfernt, so wird sie in zwei Bereiche geteilt. Der beschränkte Teil wird das *Innere* des Polygons genannt, der unbeschränkte Teil das *Äußere*. Wir nehmen an, daß die Ecken eines einfachen Polygons so geordnet sind, daß das Innere auf der rechten Seite liegt, wenn wir die Folge $v_0, v_1, \ldots, v_{n-1}, v_0$ durchlaufen (die Ecken werden also im Uhrzeigersinn durchlaufen).

(d) Ein einfaches Polygon heißt *konvex*, wenn es für alle inneren Punkte p, q von P die gesamte Strecke $L(p, q)$ enthält. □

Wir haben ein Polygon mit demselben Buchstaben bezeichnet wie die Klasse P. Das sollte jedoch nicht zu Verwechslungen führen.

Beispiel 6.5.1 Wir geben je ein Beispiel für ein Polygon an, das (a) nicht einfach, (b) einfach, aber nicht konvex und (c) konvex ist.



(a) (b) (c) □

Als Berechnungsmodell für geometrische Algorithmen reichen nicht die bisher betrachteten Turingmaschinen oder ganzzahligen Registermaschinen aus, da ganz allgemein mit reellen Zahlen gearbeitet werden muß. Wir nehmen daher an, daß in den Registern und den Feldern des Ein- und Ausgabebandes einer RAM beliebige reelle Zahlen gespeichert werden können. Ein Zugriff auf diese Zahlen soll in konstanter Zeit möglich sein. Das bedeutet, daß wir hier für die Registermaschinen uniforme Zeitkomplexität verwenden. Die arithmetischen Operationen werden mit reellen Zahlen durchgeführt. Einfache Additionen, Multiplikationen und Divisionen benötigen dann auch nur konstante Zeit $O(1)$. Die so entstehenden Zeitkomplexitätsabschätzungen entsprechen sicher dann der Wirklichkeit, wenn die geometrischen Algorithmen auf üblichen Rechnern durchgeführt werden und dort die üblichen internen Darstellungen der reellen Zahlen mit den entsprechenden Ausführungen der arithmetrischen Operationen verwendet werden.

Solche Registermaschinen können die grundlegenden Entscheidungen der ebenen Geometrie in einer konstanten Zahl von Schritten durchführen. So kann zum Beispiel die Frage gestellt werden, ob bei gegebenen drei Punkten p_1, p_2 und p_3 der Punkt p_3 auf der Strecke $L(p_1, p_2)$ liegt. Dies ist der Fall, wenn die Koordinaten (x_i, y_i) der Punkte p_i, $i = 1, 2, 3$, die Gleichung

$$\frac{x_3 - x_1}{y_3 - y_1} = \frac{x_2 - x_1}{y_2 - y_1} \qquad (\text{falls } y_3 \neq y_1 \neq y_2)$$

oder allgemeiner

$$(x_3 - x_1)(y_2 - y_1) = (x_2 - x_1)(y_3 - y_1)$$

erfüllen. Da hier nur eine konstante Zahl von arithmetischen Operationen durchgeführt wir, ist die Entscheidung in konstanter Zeit $O(1)$ möglich.

Analog kann eine solche Registermaschine in konstanter Zeit $O(1)$ entscheiden, ob ein Punkt p_3 über oder unter der durch die Punkte p_1 und p_2 laufenden Geraden liegt. Zunächst kann in konstanter Zeit aus

$$\frac{x - x_1}{y - y_1} = \frac{x_2 - x_1}{y_2 - y_1}$$

die Geradengleichung

$$y = m \cdot x + q$$

berechnet werden. Dann muß nur überprüft werden, ob

$$y_3 > m \cdot x_3 + q \text{ oder } y_3 < m \cdot x_3 + q$$

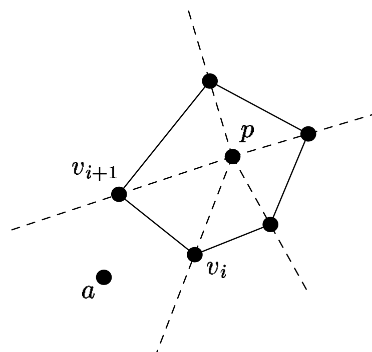
gilt.

Zur Vereinfachung unserer Überlegungen nehmen wir im folgenden an, daß die Ecken eines Polygons immer verschiedene x-Koordinaten besitzen. Da diese Situation durch eine Drehung erreicht werden kann, ist dies keine Beschränkung der Allgemeinheit. Weiter wollen wir annehmen, daß zwei benachbarte Strecken eines Polygons nicht auf derselben Geraden liegen.

Definition 6.5.2 Es sei P ein konvexes Polygon und $a \in \mathbb{R}^2$ ein Punkt. Mit INNERE EINES POLYGONS bezeichnen wir das Problem, ob a im Inneren von P liegt. □

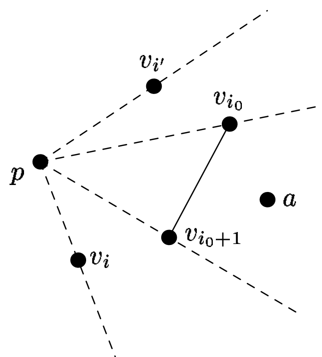
Satz 6.5.1 INNERE EINES POLYGONS kann für ein konvexes Polygon mit n Ecken in der Zeit $O(\log n)$ entschieden werden.

Beweis: Wir wählen einen Punkt p, z.B. den Schwerpunkt, im Inneren des Polygons $P = (v_0, \ldots, v_{n-1})$. Wir verlängern die Strecken $L(p, v_i)$, $i = 0, \ldots, n-1$, über die jeweiligen v_i hinaus und teilen dadurch die Ebene in n Sektoren (v_i, p, v_{i+1}) auf.



Wir suchen $i_0 \in \{0, \ldots, n-1\}$, so daß a im Sektor (v_{i_0}, p, v_{i_0+1}) liegt. Dazu gehen wir mit binärer Suche vor, indem wir zunächst für den Knoten v_i, $i = \lfloor \frac{n}{2} \rfloor$, entscheiden, ob $i < i_0$, $i > i_0$ oder $i = i_0$ gilt. Ist nicht schon $i = i_0$ gefunden, dann setzen wir zum Beispiel für $i < i_0$ die Suche mit den Ecken $v_0, \ldots, v_{\lfloor \frac{n}{2} \rfloor}$ fort. Wenn jeder einzelne Schritt in konstanter Zeit $O(1)$ entschieden werden kann, erhalten wir offenbar den Gesamtzeitbedarf von $O(\log n)$.

Falls $a = (x_a, y_a)$ rechts von $p = (x_p, y_p)$ liegt, also $x_p < x_a$ gilt (analog kann a links von p behandelt werden), dann können wir die binäre Suche weiter einschränken, indem wir nur die Ecken rechts von p betrachten. Es sei also v_i oder $v_{i'}$ eine solche Ecke, für die die Entscheidung durchgeführt werden soll.



Wenn a unter der durch $L(p, v_{i'})$ bestimmten Geraden liegt, gilt offenbar $i' \leq i_0$, liegt a über der durch $L(p, v_i)$ bestimmten Geraden, dann ist $i \geq i_0 + 1$. Liegt speziell a unter $L(p, v_i)$ und über $L(p, v_{i+1})$, dann muß folglich $i_0 = i$ gelten. Diese Entscheidungen sind in einer konstanten Zahl von Schritten möglich.

Ist der Sektor (v_{i_0}, p, v_{i_0+1}) gefunden, in dem a liegt, so liegt a genau dann im Inneren des Polygons, wenn es im Inneren des durch v_{i_0}, p und v_{i_0+1} bestimmten Dreiecks Δ liegt. Es sei

$$\alpha x + \beta y + \gamma = 0$$

die Gleichung der durch v_{i_0} und v_{i_0+1} bestimmten Geraden. Nach bekannten Sätzen der analytischen Geometrie liegt a genau dann im Inneren von Δ, wenn sich beim Einsetzen der Koordinaten (x_a, y_a) und (x_p, y_p) in die linke Seite der Geradengleichung jeweils dasselbe Vorzeichen ergibt. Auch diese Überprüfung ist in der Zeit $O(1)$ möglich. □

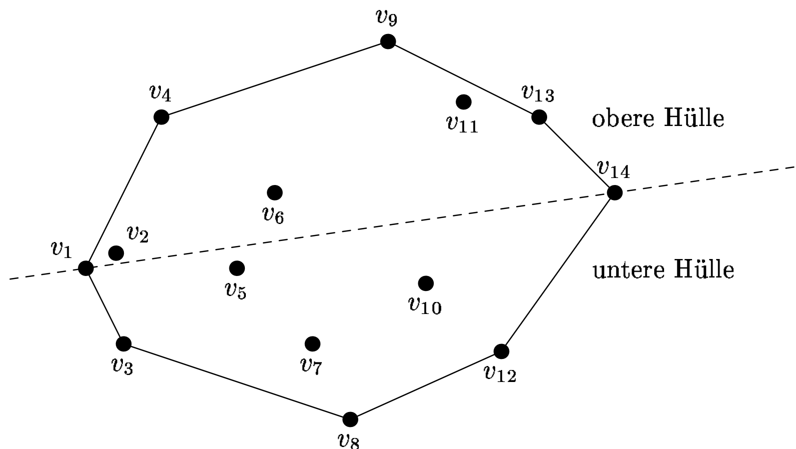
Definition 6.5.3 Es seien $v_1, \ldots, v_n \in \mathbb{R}^2$ Punkte der Ebene. Mit KONVEXE HÜLLE bezeichnen wir das Berechnungsproblem, das kleinste konvexe Polygon zu bestimmen, das alle Punkte $v_1, \ldots, v_n$ enthält. Dabei sollen die Punkte des Polygons im Uhrzeigersinn geordnet ausgegeben werden. □

Jede Ecke der konvexen Hülle gehört auch zu der gegebenen Menge von Punkten. Dies machen wir uns am „Gummibandmodell“ klar. In jedem Punkt steckt ein Nagel. Das Gummiband wird so gestreckt, daß es alle Nägel umfaßt. Anschließend wird es losgelassen, seine Form stellt dann das konvexe Polygon dar.

Satz 6.5.2 KONVEXE HÜLLE kann für n Punkte der Ebene in der Zeit $O(n \log n)$ berechnet werden.

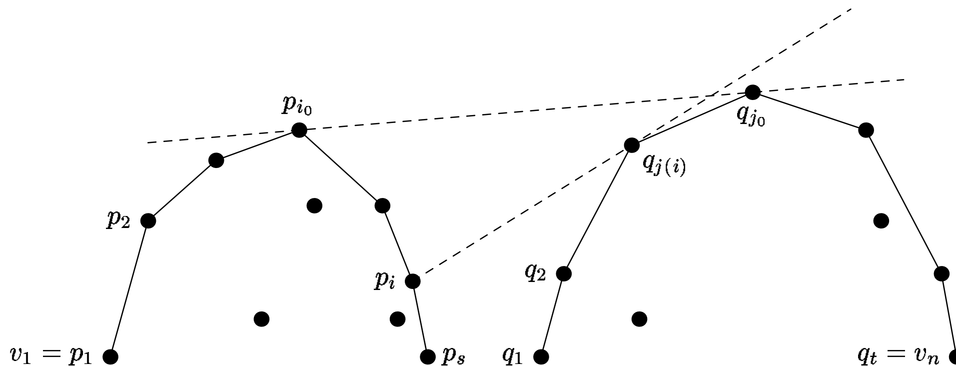
Beweis: Zunächst sortieren wir die gegebenen Punkte nach ihren x-Koordinaten. Das hat (zum Beispiel mit Heapsort) einen Zeitbedarf von $O(n \log n)$. Nach erfolgter Sortierung können wir annehmen, daß für alle $i = 1, \ldots, n-1$ die Ecke v_i links von v_{i+1} steht.

Der Punkt v_1 liegt links und v_n rechts von allen anderen gegebenen Punkten. Es folgt, daß v_1 und v_n in der konvexen Hülle liegen. Die konvexe Hülle besteht aus einer Liste von Punkten von v_1 nach v_n, die wir *obere Hülle* nennen und einer Liste von Punkten von v_n nach v_1, die mit *untere Hülle* bezeichnet wird.



In der Abbildung ist die obere Hülle durch $\{v_1, v_4, v_9, v_{13}, v_{14}\}$ gegeben, die untere durch $\{v_{14}, v_{12}, v_8, v_3, v_1\}$.

Die Berechnung der oberen Hülle ergibt sich aus der folgenden „divide-and-conquer“-Strategie. Die Gesamtheit aller Punkte $v_1, \ldots, v_n$ wird durch Berechnung von $\lfloor \frac{n}{2} \rfloor$ in die etwa gleich großen Bereiche $v_1, \ldots, v_{\lfloor \frac{n}{2} \rfloor}$ (linke Hälfte) und $v_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, v_n$ (rechte Hälfte) aufgeteilt. Für beide Hälften bestimmen wir rekursiv die oberen Hüllen, wobei die obere Hülle von $v_1, \ldots, v_{\lfloor \frac{n}{2} \rfloor}$ mit $p_1, \ldots, p_s$ (im Uhrzeigersinn) und diejenige von $v_{\lfloor \frac{n}{2} \rfloor + 1}, \ldots, v_n$ mit $q_1, \ldots, q_t$ bezeichnet sein soll.



Unter der oberen Tangente der beiden gefundenen oberen Hüllen (diese Tangente existiert immer!) liegen alle Punkte $p_1, \ldots, p_s, q_1, \ldots, q_t$. Wie diese Tangente in der Zeit $O(n)$ berechnet wird, geben wir weiter unten an. Die obere Tangente bestimmt die Strecke $L(p_{i_0}, q_{j_0})$. Damit ergibt sich die obere Hülle von $v_1, \ldots, v_n$ durch $p_1, p_2, \ldots, p_{i_0}, q_{j_0}, q_{j_0+1}, \ldots, q_t$. Diese Punkte können in $O(n)$ Schritten aufgeschrieben werden.

Der Gesamtzeitbedarf $t(n)$ zur Bestimmung der oberen Hülle der n Ecken ist aufgrund der Rekursion und den weiteren oben genannten Zeiterfordernissen durch

$$2 \cdot t\left(\frac{n}{2}\right) + kn$$

mit einer Konstanten $k \in \mathbb{N}$ für alle genügend große n beschränkt. Es existieren also

Konstanten $n_0, K \in \mathbb{N}$, so daß

$$t(n) \leq 2t\left(\frac{n}{2}\right) + K \cdot n \text{ für alle } n \geq n_0$$

gilt. Da die Gleichung

$$t(n) = 2t\left(\frac{n}{2}\right) + K \cdot n$$

die Lösung $t(n) = Kn\log_2(n)$ besitzt

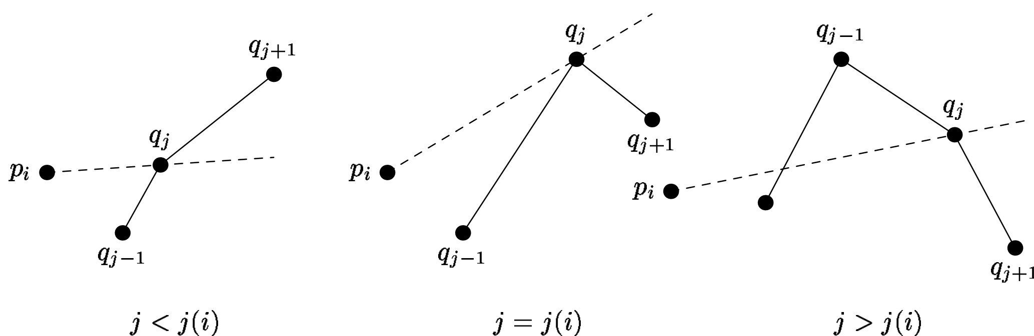
$$\left(\text{es gilt } 2K \cdot \frac{n}{2} \cdot \log_2\left(\frac{n}{2}\right) + Kn = Kn(\log_2 n - \log_2 2) + Kn = Kn\log_2 n\right),$$

gehört die Zeitkomplexität der Bestimmung der oberen Hülle erst recht zu $O(n \log n)$.

Analoge Überlegungen gelten für die Berechnung der unteren Hülle. Die konvexe Hülle ergibt sich dann in weiteren $O(n)$ Schritten aus der oberen Hülle, gefolgt von der unteren Hülle. Der Gesamtzeitbedarf ist daher von der Ordnung $O(n \log n)$.

Es bleibt zu beweisen, daß die obere Tangente in der Zeit $O(n)$ gefunden werden kann. Zunächst zeigen wir, daß wir für jeden Punkt p_i, $i = 1, \ldots, s$, die obere Tangente von p_i zu den Punkten $q_1, \ldots, q_t$ in der Zeit $O(\log t)$ berechnen können. Das bedeutet, daß wir einen Index $j(i) \in \{1, \ldots, t\}$ finden können, so daß alle Punkte $q_1, \ldots, q_t$ unterhalb der durch $L(p_i, q_{j(i)})$ bestimmten Geraden liegen. Im Bild von Seite 149 ist eine solche obere Tangente eingezeichnet. Für ein festes i gilt für jeden einzelnen Index j, $j \in \{1, \ldots, t\}$ (siehe untenstehende Abbildung):

$$\begin{array}{lcl} j < j(i) & \Longleftrightarrow & \text{(a) } q_{j+1} \text{ über } L(p_i, q_j) \\ j = j(i) & \Longleftrightarrow & \text{(b) } q_{j-1} \text{ und } q_{j+1} \text{ unter } L(p_i, q_j) \\ j > j(i) & \Longleftrightarrow & \text{(c) } q_{j-1} \text{ über } L(p_i, q_j) \end{array}$$

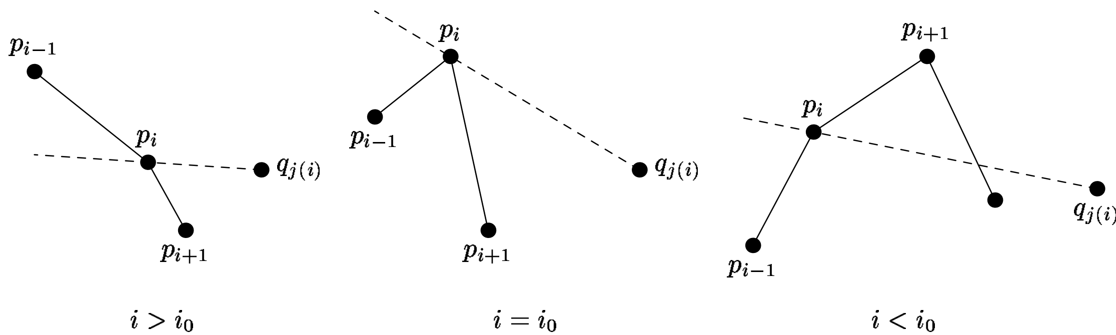


Ein jeder solcher Test erfordert die Zeit $O(1)$. Eine binäre Suche über den Punkten $q_1, \ldots, q_t$ wird jetzt so durchgeführt, daß jeweils der „mittlere“ Punkt der betrachteten Punkte getestet wird. Im Fall (a) wird dann mit der rechten, im Fall (c) mit der linken Hälfte fortgefahren, wohingegen man im Fall (b) den Index $j(i)$ gefunden hat. Folglich kann in der Zeit $O(\log t)$ die Strecke $L(p_i, q_{j(i)})$ bestimmt werden.

Wir zeigen jetzt, wie wir unter Benutzung geeigneter Strecken $L(p_i, q_{j(i)})$ die obere Tangente bzw. die Strecke $L(p_{i_0}, q_{j_0})$ finden können. Offensichtlich muß für das gesuchte i_0 die Gleichung $j_0 = j(i_0)$ gelten. Durch eine binäre Suche über den Punkten $p_1, \ldots, p_s$ kann i_0 gefunden werden. Die binäre Suche betrachtet $O(\log s)$ Punkte p_i, für die

jeweils wie zuvor beschrieben in der Zeit $O(\log t)$ der Punkt $p_{j(i)}$ und die Strecke $L(p_i, q_{j(i)})$, also die Tangente von p_i zu den Punkten $q_1, \ldots, q_t$, bestimmt werden. In konstanter Zeit $O(1)$ kann dann überprüft werden, welche der folgenden Relationen erfüllt ist:

$$\begin{array}{lcl} i > i_0 & \iff & (\alpha)\ p_{i-1} \text{ über } L(p_i, q_{j(i)}) \\ i = i_0 & \iff & (\beta)\ p_{i-1} \text{ und } p_{i+1} \text{ unter } L(p_i, q_{j(i)}) \\ i < i_0 & \iff & (\gamma)\ p_{i+1} \text{ über } L(p_i, q_{j(i)}) \end{array}$$



Im Fall (α) geht man zur linken, im Fall (γ) zur rechten Hälfte über. Im Fall (β) erhalten wir $i = i_0$. Es gilt dann $j(i_0) = j_0$, und die obere Tangente ist gefunden. Der Zeitbedarf dafür beträgt

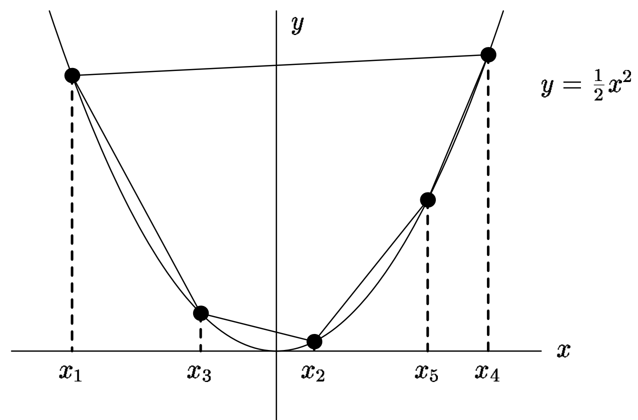
$$O(\log s) \cdot O(\log t) = O(\log s \cdot \log t).$$

Wegen $\log s \log t \leq \log^2(s+t) = \log^2 n \leq n$ für genügend große n ist der Zeitbedarf zum Finden der oberen Tangente von der Ordung $O(n)$. □

Zum Abschluß zeigen wir, daß der Mindestzeitbedarf für KONVEXE HÜLLE von der Ordnung $n \log n$ ist.

Satz 6.5.3 KONVEXE HÜLLE hat eine Zeitkomplexität der Ordnung $\Omega(n \log n)$.

Beweis: Wir nehmen an, daß KONVEXE HÜLLE bei n gegebenen Punkten der Ebene eine Zeitkomplexität von $t(n)$ hat. Wir zeigen, daß dann eine Menge $\{x_1, \ldots, x_n\} \subset \mathbb{R}$ in der Zeit $O(t(n)) + O(n)$ sortiert werden kann. Da bekannt ist, daß Sortieren einen Zeitbedarf der Ordnung $\Omega(n \log n)$ besitzt, kann somit auch KONVEXE HÜLLE nicht schneller berechnet werden.



Für die Zahlen $x_1, \ldots, x_n$ bestimmen wir in der Zeit $O(n)$ mit Hilfe der Parabelgleichung $y = \frac{1}{2}x^2$ die Punkte $v_i = (x_i, y_i)$, $i = 1, \ldots, n$. Da die Parabel konvex ist, liegt jeder dieser Punkte auf der konvexen Hülle. Jeder Algorithmus, der KONVEXE HÜLLE berechnet, liefert als Ausgabe die Punkte $v_{i_1}, \ldots, v_{i_n}$, im Uhrzeigersinn sortiert. In linearer Zeit $O(n)$ können wir dann den Index k des Punktes v_{i_k} bestimmen, der am weitesten rechts liegt. In

der Zeit $O(n)$ erhalten wir anschließend die sortierte Menge

$$x_{i_k} > x_{i_{k+1}} > \ldots > x_{i_n} > x_{i_1} > x_{i_2} > \ldots > x_{i_{k-1}}.$$

Folglich ist insgesamt die Sortierung in der Zeit $O(t(n)) + O(n)$ möglich. $\square$

7 Nichtdeterministische Polynomialzeitalgorithmen

7.1 Die Komplexitätsklasse *NP*

Im vorangegangenen Kapitel haben wir Probleme betrachtet, die sich durch deterministische Turingmaschinen in polynomialer Zeit lösen lassen. Jetzt wollen wir zur Lösung nichtdeterministische Turingmaschinen mit polynomialem Zeitbedarf verwenden. Wir erhalten so die P umfassende Komplexitätsklasse *NP*. Intuitiv kann man die Klasse *NP* auch dadurch beschreiben, daß ein Entscheidungsproblem zu ihr gehört, wenn für eine „angebotene" Lösung eines speziellen Falls des Problems in polynomialer Zeit überprüft werden kann, ob es sich dabei tatsächlich um eine Lösung handelt. Dies wird in Satz 7.2.3 formaler dargestellt. Wir wollen jedoch diese intuitive Beschreibung zunächst an einem Beispiel verdeutlichen.

Beispiel 7.1.1 ZERLEGBARKEIT ist das Problem zu entscheiden, ob eine natürliche Zahl $n \in \mathbb{N}$ als Produkt $n = pq$ zweier kleinerer Zahlen $p, q \in \mathbb{N}$ dargestellt werden kann. Obwohl dieses Problem schon seit Jahrtausenden in der Mathematik betrachtet wird, ist bis jetzt noch kein effizienter Algorithmus dafür gefunden worden. Wir wissen also nicht, ob ZERLEGBARKEIT zu P gehört. Es ist im übrigen zu hoffen, daß eine solche Zugehörigkeit nicht gilt, weil sonst ein Public-Key-Kryptosystem, das heute im Geschäfts- und Bankwesen vielfach benutzt wird, geknackt wäre. Aber ZERLEGBARKEIT gehört zu *NP*. Wenn jemand Zahlen p und q nennt, können wir überprüfen, ob $n = pq$ gilt.

Dieses Vorgehen kann durch eine nichtdeterministische Turingmaschine beschrieben werden. Sie hat n als Eingabe. Sie schreibt zusätzlich nichtdeterministisch Zahlen p und q auf ihr Band. Sie multipliziert p und q und vergleicht das Ergebnis mit n. Falls die Werte gleich sind, hält die Turingmaschine in einem akzeptierenden Zustand, anderenfalls in einem Nichtendzustand. Wir verzichten auf ihre genaue Definition. Offensichtlich hält die Turingmaschine in polynomialer Zeit in bezug auf die Länge von n. □

In Definition 6.2.1 haben wir die Zeitkomplexität von deterministischen Turingmaschinen definiert. Für endliche Zeitkomplexität muß die Turingmaschine in jedem Fall halten. Bei nichtdeterministischen Turingmaschinen ist die Situation etwas anders. Nur für erkannte Wörter muß es eine akzeptierende Rechnung in der angegebenen Zeit geben.

Definition 7.1.1 Es sei $T = (Z, X, \delta, z_0, F)$ eine nichtdeterministische Turingmaschine. Dann heißt

$$K_T(n) = \max_{w \in L(T), |w|=n} \min\{k \mid k \text{ ist die Länge einer akzeptierenden Rechnung an } w\}$$

die *Zeitkomplexität von T*. □

Man bildet das Maximum nur über Wörter der erkannten Sprache. Für jedes Wort wird dabei die Länge der kürzesten akzeptierenden Rechnung berücksichtigt. Wir kommen jetzt zur Defintion der Klasse *NP*.

Definition 7.1.2 Es ist

$$NP = \{L \mid \text{es existiert eine nichtdeterministische Turingmaschine } T \text{ mit } L = L(T) \text{ und } K_T(n) \leq p_T(n) \text{ für ein Polynom } p_T \text{ und alle } n\}$$

die *Komplexitätsklasse der in nichtdeterministischer polynomialer Zeit entscheidbaren Sprachen.* □

Jede deterministische Turingmaschine kann auch als nichtdeterministische Turingmaschine aufgefaßt werden. Die Definition 6.2.1 der Zeitkomplexität bei deterministischen Turingmaschinen ist strenger als bei nichtdeterministischen. Gilt nämlich $K_T(n) \leq p_T(n)$ für alle Wörter $w \in X^*$, $|w| = n$, so gilt dies erst recht für alle $w \in L(T)$ mit $|w| = n$. Wir erhalten als Folgerung:

Satz 7.1.1 $P \subset NP$. □

Nach wie vor ist die Frage offen, ob diese Inklusion echt ist.

In Abschnitt 6.4 haben wir gesehen, daß deterministische Einband- und Mehrband-Turingmaschinen sowie deterministische Registermaschinen bei höchstens polynomialem Zeitverlust sich gegenseitig simulieren können. Die Klassen *P* und *FP* sind unabhängig von dem jeweiligen Modell. Die entsprechenden Sätze 6.4.1, 6.4.2(a), 6.4.3 und 6.4.4(a) lassen sich analog auch für die jeweiligen nichtdeterministischen Maschinenmodelle formulieren und beweisen. Das bedeutet, daß auch die Definition der Klasse *NP* nicht von dem jeweiligen nichtdeterministischen Maschinenmodell abhängig ist.

Wir geben jetzt einige Probleme an, für die bisher nicht gezeigt werden konnte, daß sie zur Klasse *P* gehören, die jedoch in der Klasse *NP* liegen.

Satz 7.1.2 ZERLEGBARKEIT $\in NP$.

Beweis: Dies folgt aus den Überlegungen von Beispiel 7.1.1. □

Satz 7.1.3 SAT $\in NP$.

Beweis: Es sei α ein Ausdruck in konjunktiver Normalform und $V_\alpha = \{x_1, \ldots, x_m\}$, $m \in \mathbb{N}$, die Menge der in α auftretenden Variablen. Dann gilt $m \leq |\alpha|$, wobei $|\alpha|$ die Länge der Darstellung von α ist. Intuitiv ist klar, daß man einen nichtdeterministischen Algorithmus finden kann, der eine Belegung errät und überprüft, ob mit dieser Belegung der Ausdruck α erfüllbar ist. Das Raten und die Überprüfung sind dabei in polynomialer Zeit möglich.

Formal muß nach Definition 7.1.2 eine nichtdeterministische Turingmaschine T konstruiert werden, die SAT in Polynomialzeit akzeptiert. Wir geben eine Konstruktionsidee für T an. Zuerst steht T links auf dem ersten Feld von α und errät nichtdeterministisch eine Belegung für $x1$. Das bedeutet, daß sich T die Zahl 0 oder 1 im Zustand merkt. T durchläuft dann das Wort α von links nach rechts und erniedrigt bei jeder Variablen den Index um 1. Dabei kann der anfängliche Index 1 dadurch identifiziert werden, daß er auf den Wert 0 herabgesetzt wird. An diesen Stellen, an

denen also anfangs die Variable $x1$ stand, wird jeweils der Wert für $x1$ eingesetzt, den sich die Turingmaschine im Zustand gemerkt hat. Dabei werden nicht benötigte Felder z.B. mit dem Symbol $*$ gefüllt. Im Zustand wird die Information gehalten, ob noch eine Variable vorhanden ist, deren Index nicht auf 0 erniedrigt wurde. Wenn dies nach Durchlaufen von α noch der Fall ist, wandert T nach links und wiederholt das obige Verfahren. Das bedeutet, daß die Variable $x10$, die im ersten Durchgang zu $x01$ umbenannt wurde, überall in α durch denjenigen Wert ersetzt wird, der zu Beginn dieses Durchlaufs erraten und im Zustand gemerkt wurde. Dieses Verfahren wird so lange wiederholt, bis alle Werte eingesetzt sind. Das ist durch den Zustand feststellbar. Der Zeitbedarf für diese Arbeiten von T ist $O(|\alpha|^2)$.

Dann läuft T nach links, und die Klauseln werden der Reihe nach getestet. Falls in der ersten Klausel *keine* 1 enthalten ist, geht T in eine nichtakzeptierende Endkonfiguration über. Anderenfalls geht T weiter zur nächsten Klausel. Ist schließlich ganz α durchlaufen, dann geht T in eine akzeptierende Endkonfiguration über. Der Zeitbedarf für diese Arbeiten ist $O(|\alpha|)$. Da T genau dann das Wort α akzeptiert, wenn der Wert von α unter einer der geratenen Belegungen 1 ist, folgt nun, daß SAT von T akzeptiert wird. Der Gesamtzeitbedarf dafür ist $O(|\alpha|^2)$. Also gilt SAT $\in$ *NP*. □

Es folgt jetzt ein Problem von großer praktischer Bedeutung.

Definition 7.1.3 Das TRAVELING-SALESMAN-PROBLEM (kurz TSP), das Problem des Handlungsreisenden, besitzt als Eingabe eine Liste von Städten $S_1, \ldots, S_n$ mit $n \in \mathbb{N}$ sowie eine Matrix

$$(d(i,j)), i,j = 1, \ldots, n, \text{ mit } d(i,j) \in \mathbb{N}_0,$$

die die Entfernungen von der jeweiligen Stadt S_i in die Stadt S_j darstellen (oder die Kosten der Reise von S_i nach S_j), außerdem eine Zahl $k \in \mathbb{N}$. Das Problem ist zu entscheiden, ob eine Rundfahrt, also eine Permutation $(i_1, i_2, \ldots, i_n)$ der Zahlen $1, \ldots, n$ existiert, so daß

$$d(i_1, i_2) + d(i_2, i_3) + \ldots + d(i_{n-1}, i_n) + d(i_n, i_1) \leq k$$

gilt. □

Man beachte, daß Entfernungen in Kilometer und Meter sowie Kosten in Euro und Cent als natürliche Zahlen ausgedrückt werden können.

Satz 7.1.4 TSP $\in$ *NP*

Beweis: Falls eine Permutation gegeben wird, benötigt die Überprüfung der Ungleichung aus Definition 7.1.3 nur lineare Zeit in bezug auf die Länge des Problems. Wir verzichten hier auf die explizite Angabe einer entsprechenden Turingmaschine. Nach dem Satz 7.2.3 aus dem nächsten Abschnitt folgt dann das Ergebnis. □

Definition 7.1.4 Das KNAPSACK-PROBLEM (Knapsack = Rucksack) ist wie folgt gegeben:

Es sei $A = \{a_1, \ldots, a_n\}$, $a_i \in \mathbb{N}$, $i = 1, \ldots, n$ und $S \in \mathbb{N}$. Existiert ein $A' \subset A$ mit $\sum\limits_{a' \in A'} a' = S$? □

Die a_i repräsentieren dabei Gegenstände des Volumens a_i. Das Problem ist, ob ein Rucksack des Volumens S gefüllt werden kann, ohne Platz zu verschenken.

Satz 7.1.5 KNAPSACK-PROBLEM $\in NP$.

Beweis: Für jedes $A' \subset A$ kann die Überprüfung $\sum_{a' \in A'} a' = S$ in polynomialer Zeit durchgeführt werden kann. □

Ein deterministischer Algorithmus kann alle 2^n Möglichkeiten $A' \subset A$ durchprobieren. Seine Zeitkomplexität beträgt dann $\Omega(2^n)$. Im folgenden untersuchen wir den Zusammenhang zwischen deterministischer und nichtdeterministischer Zeitkomplexität allgemeiner. Die beste bekannte Simulation nichtdeterministischer Maschinen durch deterministische bringt einen exponentiellen Zeitverlust.

Satz 7.1.6 Es sei p ein Polynom, $L \subset X^*$ eine Sprache und T eine nichtdeterministische Turingmaschine mit $L(T) = L$ und $K_T(n) = O(p(n))$. Dann existiert eine deterministische Turingmaschine D mit $L(D) = L$ und $K_D(n) = O(c^{p(n)})$ für ein geeignetes $c \in I\!N$.

Beweisskizze: Es sei $T = (Z, X, \delta, z_0, F)$. Für jedes $w \in L$ existiert eine akzeptierende Rechnung der Länge $\leq K_T(|w|)$. Wir setzen

$$k = \max\{|\delta(z,a)| \mid z \text{ Zustand von } T,\ a \in \bar{X}\}.$$

Die Turingmaschine T kann also bei jedem Schritt zwischen höchstens k Aktionen wählen. Zum Test, ob $w \in L$ gilt, müssen $\leq k^{K_T(|w|)}$ Rechnungen durchprobiert werden, wobei $K_T(|w|) \leq m\ p(|w|)$ gilt mit einer Konstanten $m \in I\!N$. Beim Durchprobieren zählt man von 0 bis $k^{m \cdot p(|w|)}-1$. Jede solche Zahl kann unter Verwendung von führenden Nullen mit genau $m \cdot p(|w|)$ Ziffern zur Basis k geschrieben werden. Sie ist dann die Darstellung einer möglichen Rechnung, und zwar gibt die i-te Ziffer der k-adischen Darstellung die Wahl der Aktion im i-ten Schritt von T an.

Unter Berücksichtigung des Polynoms p und der Konstanten m konstruieren wir eine deterministische Turingmaschine D, die zunächst $p(|w|)$ berechnet und sich dann nacheinander die Zahlen von 0 bis $k^{mp(|w|)} - 1$ auf das Band schreibt. Ist eine Zahl geschrieben, so simuliert D auf einem benachbarten Bereich des Bandes gemäß den entsprechenden Ziffern eine Rechnung der Länge $\leq m \cdot p(|w|)$ der nichtdeterministischen Turingmaschine T. Für jede Zahl muß sie bei jedem der höchstens $m \cdot p(|w|)$ zu simulierenden Schritte von T zum Feststellen der Ziffern über das Band der Länge $O(m \cdot p(|w|))$ hin- und herlaufen, so daß sich für eine Rechnung von T höchstens die Zeit $m_1 \cdot (m \cdot p(|w|))^2$ mit einer geeigneten Konstanten m_1 ergibt. D simuliert somit alle Rechnungen mit einem geeigneten $m_2 \in I\!N$ in höchstens

$$\begin{aligned} m_1 \cdot (m \cdot p(|w|))^2 \cdot k^{m \cdot p(|w|)} &\leq m_2 \cdot 2^{m \cdot p(|w|)} \cdot k^{m \cdot p(|w|)} = m_2 \cdot ((2k)^m)^{p(|w|)} \\ &= O(c^{p(|w|)}) \end{aligned}$$

Schritten, wobei $c = (2k)^m$ gesetzt wird. □

Als Folgerung erhalten wir

Satz 7.1.7 Jedes Problem aus der Klasse NP kann von einem (deterministischen) Algorithmus mit exponentieller Zeitkomplexität gelöst werden. □

7.2 *NP*-Vollständigkeit

Für keines der in Abschnitt 7.1 angegebenen Probleme aus *NP* ist ein effizienter Algorithmus bekannt. Allgemeiner ist die Frage offen, ob $P = NP$ gilt oder nicht. Trotz jahrzentelanger Bemühungen und trotz vieler Nebenergebnissen zu dieser Thematik ist die Antwort immer noch nicht gefunden. Es gibt aber, wie schon in der Einleitung zu Kapitel 6 angemerkt wurde, in der Klasse *NP* tausend und mehr Probleme, die eine besondere Stellung haben, die sogenannten *NP*-vollständigen Probleme. Wenn nur für ein einziges solches Problem die Zugehörigkeit zu P gezeigt werden könnte, dann fielen die Klassen P und *NP* zusammen. Für die Definition der *NP*-Vollständigkeit erinnern wir uns an den Begriff der Reduktion in polynomialer Zeit aus Definition 6.3.4.

Definition 7.2.1 Es sei L eine Sprache. L heißt *NP-vollständig*, wenn
(a) $L \in NP$ ist und
(b) $L' \leq L$ für alle $L' \in NP$ gilt. □

Der nächste Satz erhellt die Bedeutung von Definition 7.2.1.

Satz 7.2.1 Es sei L_0 *NP*-vollständig. Dann gelten die folgenden Aussagen:
(a) Es ist $L_0 \in P$ genau dann, wenn $P = NP$ gilt.
(b) Es sei $L_0 \leq L_1$ und $L_1 \in NP$. Dann ist L_1 *NP*-vollständig.

Beweis: (a) Es gelte $P = NP$. Da L_0 nach Voraussetzung *NP*-vollständig ist, folgt daraus nach Definition 7.2.1(a) $L_0 \in NP = P$. Umgekehrt sei $L_0 \in P$ und L eine beliebige Sprache aus *NP*. Nach Definition 7.2.1(b) gilt $L \leq L_0$. Satz 6.3.4 liefert $L \in P$.
(b) Es sei L_0 *NP*-vollständig, $L_1 \in NP$, $L_0 \leq L_1$ und $L \in NP$ beliebig. Zu zeigen ist $L \leq L_1$. Da L_0 *NP*-vollständig ist, gilt $L \leq L_0$. Wegen $L_0 \leq L_1$ folgt nach Satz 6.3.5 die Relation $L \leq L_1$. □

Die Aussage (a) des Satzes bedeutet, daß *NP*-vollständige Probleme die schwierigsten Probleme in *NP* sind. Wenn ein *NP*-vollständiges Problem in Polynomialzeit lösbar ist, dann gilt dies für alle Probleme in *NP*. Gilt umgekehrt $P \neq NP$, was allgemein angenommen wird, dann folgt $L_0 \notin P$, d.h., *NP*-vollständige Probleme sind nicht in Polynomialzeit lösbar.

Weiter liefert die Aussage (b) eine einfache Technik, um die *NP*-Vollständigkeit einer Sprache L_1 nachzuweisen. Wenn nämlich eine *NP*-vollständige Sprache L_0 bekannt ist, reicht der Nachweis von $L_0 \leq L_1$ und $L_1 \in NP$ aus.

Wir zeigen als erstes, daß SAT *NP*-vollständig ist. Anschließend beweisen wir die *NP*-Vollständigkeit einiger anderer Probleme mit Hilfe von Satz 7.2.1(b).

Satz 7.2.2 SAT ist *NP*-vollständig.

Beweis: Es sei $L \in NP$. Wegen Satz 7.1.3 und Definition 7.2.1 bleibt zu zeigen, daß $L \leq$ SAT gilt. Wegen $L \in NP$ existiert eine nichtdeterministische Turingmaschine T, die L in der Zeit p akzeptiert, wobei p ein Polynom ist. Wir werden nun eine

deterministische Turingmaschine angeben, die in Polynomialzeit zu jedem $w \in X^*$ einen Ausdruck $\alpha(w)$ konstruiert mit:

$$w \in L \Longleftrightarrow \alpha(w) \text{ erfüllbar (d.h. } \alpha(w) \in \text{SAT}).$$

Dann ist der Satz bewiesen.

Wir betrachten T. Die Turingmaschine T besitze die Zustände $z_0, \dots, z_s$, das Bandalphabet $X = \{A_2, \dots, A_v\}$ sowie das Blankzeichen $A_1 = b$, den Startzustand z_0 und die Endzustandsmenge $F = \{z_r, z_{r+1}, \dots, z_s\}$. O.B.d.A. kann $z_0 \notin F$ angenommen werden. Für diesen Beweis wollen wir die Arbeitsweise der Turingmaschine so ändern, daß bei einer Endkonfiguration nicht gestoppt wird, sondern diese unendlich oft wiederholt wird. Um dies zu erreichen, müssen wir zu einer Turingmaschine T' gemäß Definition 4.5.1 eine äquivalente Turingmaschine T konstruieren, die diese Arbeitsweise besitzt. Dabei wird zunächst jede Zeile der Turingtafel von T', die die Gestalt $z\ x\ z'\ s$ hat, durch zwei Zeilen $z\ x\ z''\ x$ und $z''\ x\ z''\ x$ ersetzt mit je einem neuen Zustand z'' für jedes solche z'. Die Endzustandsmenge F von T bestehe aus genau solchen Zuständen z'', für die z' ein Endzustand von T' ist. Eine akzeptierende Endkonfiguration von T ist daher äquivalent zu einer Konfiguration mit einem Zustand $z \in F$. Bei diesem Modell gilt nun: T akzeptiert w mit $|w| = n$ genau dann, wenn eine Folge $C_0, C_1, \dots, C_{p(n)}$ von Konfigurationen existiert, in der C_0 die Anfangskonfiguration zu w, C_{i+1} die Folgekonfiguration von C_i für $0 \leq i \leq p(n) - 1$ und $C_{p(n)} = (n', \Gamma(\beta), z)$ eine akzeptierende Endkonfiguration mit $z \in F$ und $n' \in \mathbb{Z}$ ist.

Es sei m die Anzahl der Zeilen der zu T gehörenden Turingtafel. Es sei $w \in X^*$ mit $|w| = n$. Wir werden den Ausdruck $\alpha(w)$ aus folgenden Variablen aufbauen:

Variable	Bereich	beabsichtigte Bedeutung
$z_{t,k}$	$0 \leq t \leq p(n)$ $0 \leq k \leq s$	$z_{t,k} = 1 \iff T$ ist nach t Schritten im Zustand z_k
$a_{t,i,j}$	$0 \leq t \leq p(n)$ $-p(n) \leq i \leq p(n)$ $1 \leq j \leq v$	$a_{t,i,j} = 1 \iff \beta(i) = A_j$ nach t Schritten
$s_{t,i}$	$0 \leq t \leq p(n)$ $-p(n) \leq i \leq p(n)$	$s_{t,i} = 1 \iff T$ liest nach t Schritten das Feld i
$b_{t,l}$	$0 \leq t \leq p(n)$ $1 \leq l \leq m$	$b_{t,l} = 1 \iff T$ wendet beim Übergang von der Zeit t zur Zeit $t+1$ die l-te Zeile der Turingtafel an

Die beabsichtigte Bedeutung ist, daß die Variablen $z_{t,k}$ den Zustand, die Variablen $a_{t,i,j}$ die Bandinschrift, $s_{t,i}$ die Position des Kopfes und $b_{t,l}$ das Übergangsverhalten der Turingmaschine kodieren. In einer akzeptierenden Rechnung werden höchstens $p(n)$ Schritte betrachtet, wobei die Turingmaschine nach links oder rechts laufen kann. Dadurch ist der Bereich für den Index i bestimmt. Im folgenden werden Ausdrücke gebildet, die den Variablen im Hinblick auf eine Rechnung der Turingmaschine T genau die Bedeutung geben, wie sie in der Tabelle notiert ist. Dabei müssen wir dafür sorgen, daß folgendes eingehalten wird:

(a) Die Anfangsbedingung, d.h., T startet im Zustand z_0 auf dem Feld 0 und $\ldots b^{p(n)} w b^{p(n)-n+1} \ldots$ ist die Bandinschrift, wobei das erste Symbol von w im Feld 0 steht.
(b) Die Randbedingungen, d.h., T befindet sich zu jedem Zeitpunkt in genau einem Zustand und liest genau ein Zeichen. Jedes Feld enthält genau ein Zeichen, und es wird genau eine Zeile der Turingtafel angewendet.
(c) Das Übergangsverhalten, d.h., der Zustand, die Kopfposition und die Bandinschriften zu aufeinanderfolgenden Zeiten sind mit der Turingtafel verträglich.

Wir sehen, daß das Wort w nur für die Anfangsbedingung berücksichtigt werden muß. Die beiden anderen Bedingungen beruhen allein auf der Gestalt von T. Mit A werde der Ausdruck für die Anfangsbedingung, mit R der Ausdruck für die Randbedingungen und mit U der Ausdruck für das Übergangsverhalten bezeichnet. Damit setzen wir

$$\alpha(w) = A \wedge R \wedge U \wedge (z_{p(n),r} \vee z_{p(n),r+1} \vee \ldots \vee z_{p(n),s}).$$

Der Inhalt der Klammer bedeutet in Übereinstimmung mit der Tabelle, daß sich die Turingmaschine am Ende der Rechnung in einem Endzustand befindet. Hat ein solches $\alpha(w)$ den Wert 1, so entspricht dies aufgrund der beabsichtigten Bedeutung einer akzeptierenden Rechnung der Turingmaschine. Wir konstruieren nun A, R und U in konjunktiver Normalform.

(a) Es sei $w = A_{j_1} \ldots A_{j_n}$. Dann setzen wir

$$\begin{aligned} A \;=\; & (z_{0,0} \wedge s_{0,0} \wedge a_{0,0,j_1} \wedge a_{0,1,j_2} \wedge \ldots \wedge a_{0,n-1,j_n} \\ & \wedge a_{0,-p(n),1} \wedge \ldots \wedge a_{0,-1,1} \wedge a_{0,n,1} \wedge \ldots \wedge a_{0,p(n),1}). \end{aligned}$$

Dies soll bedeuten, daß sich T zur Zeit 0 im Zustand z_0 befindet, das Feld 0 liest und $\ldots b^{p(n)} w b^{p(n)-n+1} \ldots$ die Bandinschrift ist. A ist ein Ausdruck in konjunktiver Normalform. Es treten $2p(n) + 3 = O(p(n))$ Variablen auf.

(b) Im Ausdruck R müssen wir Bedingungen ausdrücken wie „T befindet sich in genau einem Zustand“. Allgemein wird der Sachverhalt, daß von einer Menge von Variablen $x_1, \ldots, x_h$ genau eine Variable wahr ist, durch die Formel

$$\begin{aligned} &\text{Genau-Eine}(x_1, \ldots, x_h) \\ &\quad = \text{Mindestens-Eine}(x_1, \ldots, x_h) \wedge \text{Höchstens-Eine}(x_1, \ldots, x_h), \end{aligned}$$

beschrieben, wobei

$$\text{Mindestens-Eine}(x_1, \ldots, x_h) = (x_1 \vee \ldots \vee x_h) \quad \text{und}$$

$$\begin{aligned} &\text{Höchstens-Eine}(x_1, \ldots, x_h) = \neg(\text{Mindestens-Zwei}(x_1, \ldots, x_h)) \\ &\quad = \neg \bigvee_{1 \le i < j \le h} (x_i \wedge x_j) = \bigwedge_{1 \le i < j \le h} (\bar{x}_i \vee \bar{x}_j) \end{aligned}$$

gesetzt wird. Genau-Eine$(x_1, \ldots, x_h)$ ist ein Ausdruck in konjunktiver Normalform. Es treten $h + 2\binom{h}{2} = h + 2h(h-1)/2 = h^2$ Variablen auf. Wir sehen, daß eine Belegung Ψ der Variablen $x_1, \ldots, x_h$ genau dann Genau-Eine$(x_1, \ldots, x_h)$ erfüllt, wenn $\Psi(x_i) = 1$ für genau ein i gilt mit $1 \le i \le h$.
Es sei nun

$$R \;=\; \bigwedge_{0 \le t \le p(n)} (R_{\mathit{Zustände}}(t) \wedge R_{\mathit{Position}}(t) \wedge R_{\mathit{Bandinschrift}}(t) \wedge R_{\mathit{Zugauswahl}}(t)),$$

wobei die Ausdrücke in der Klammer noch definiert werden müssen. Sie sollen entsprechend der vorliegenden Reihenfolge angeben, daß zum Zeitpunkt t

(1) T sich in genau einem Zustand befindet,
(2) T genau ein Feld des Bandes liest,
(3) jedes Feld genau ein Zeichen enthält und
(4) T beim Übergang von der Zeit t zur Zeit $t+1$ genau eine Zeile der Turingtafel anwendet.

Um diese Bedeutung zu erreichen, setzen wir

$$\begin{aligned}
R_{Zustände}(t) &= \text{Genau-Eine}(z_{t,0}, \ldots, z_{t,s}), \\
R_{Position}(t) &= \text{Genau-Eine}(s_{t,-p(n)}, \ldots, s_{t,p(n)}), \\
R_{Bandinschrift}(t) &= \bigwedge_{-p(n)\le i\le p(n)} \text{Genau-Eine}(a_{t,i,1}, \ldots, a_{t,i,v}), \\
R_{Zugauswahl}(t) &= \text{Genau-Eine}(b_{t,1}, \ldots, b_{t,m}).
\end{aligned}$$

Dies sind Ausdrücke in konjunktiver Normalform, in denen $(s+1)^2$, $(2p(n)+1)^2$, $(2p(n)+1)v^2$ bzw. m^2 Variablen vorkommen. Folglich ist R ein Ausdruck in konjunktiver Normalform, in dem

$$(p(n)+1)((s+1)^2 + (2p(n)+1)^2 + (2p(n)+1)v^2 + m^2) = O((p(n))^3)$$

Variablen auftreten.

(c) Wir setzen $U = \bigwedge_{0\le t<p(n)} U(t)$. Der Ausdruck $U(t)$ soll beschreiben, daß der Übergang von der Zeit t zur Zeit $t+1$ verträglich ist mit der Zeile der Turingtafel, die durch die jeweilige Variable aus $b_{t,1}, \ldots, b_{t,m}$ bestimmt wird. Die l-te Zeile der Turingtafel $(1 \le l \le m)$ sei

$$z_{k_l}\ A_{j_l}\ z_{\tilde{k}_l}\ B_{j_l},$$

wobei $B_{j_l} = A_{\tilde{j}_l}$ mit einem $\tilde{j}_l \in \{1, \ldots, v\}$ oder $B_{j_l} \in \{r, l\}$ gilt. Für $B_{j_l} = l$ setzen wir $R_l = -1$, für $B_{j_l} = r$ wird diese Variable durch $R_l = 1$ bestimmt. Das Übergangsverhalten der Turingmaschine T kann dadurch beschrieben werden, daß man für jedes Feld i des Bandes, $-p(n) \le i \le p(n)$, folgendes ausdrückt:

(1) Falls T zur Zeit t das i-te Feld *nicht* liest, dann ändert sich sein Inhalt nicht.
(2) Falls T zur Zeit t das i-te Feld liest und die l-te Zeile der Turingtafel angewendet werden soll, dann
   (α) ist zur Zeit t der Zustand durch z_{k_l} und der Inhalt des i-ten Feldes durch A_{j_l} gegeben, und
   (β) zur Zeit $t+1$ ist $z_{\tilde{k}_l}$ der Zustand,
   und
   entweder steht A_{j_l} im i-ten Feld, und T liest das Feld $i + R_l$,
   oder $A_{\tilde{j}_l}$ steht im i-ten Feld, und T liest das Feld i.

Für ein festes Feld i wird (1) beschrieben durch

$$\bigwedge_{1\le j\le v} (\bar{s}_{t,i} \wedge a_{t,i,j} \Longrightarrow a_{t+1,i,j}).$$

Wegen der logischen Äquivalenz von $A \Longrightarrow B$ mit $\bar{A} \vee B$ ist dies äquivalent zu

$$\bigwedge_{1 \leq j \leq v} (s_{t,i} \vee \bar{a}_{t,i,j} \vee a_{t+1,i,j}).$$

Unter Verwendung dieser Äquivalenz ergibt sich bei Berücksichtigung der durch (1) und (2) gegebenen beabsichtigten Bedeutung

$$U(t) = \bigwedge_{-p(n) \leq i \leq p(n)} \{ \bigwedge_{1 \leq j \leq v} (s_{t,i} \vee \bar{a}_{t,i,j} \vee a_{t+1,i,j})$$

$$\wedge \bigwedge_{1 \leq l \leq m} [(\bar{s}_{t,i} \vee \bar{b}_{t,l} \vee z_{t,k_l}) \wedge (\bar{s}_{t,i} \vee \bar{b}_{t,l} \vee a_{t,i,j_l})$$

$$\wedge (\bar{s}_{t,i} \vee \bar{b}_{t,l} \vee z_{t+1,\tilde{k}_l}) \wedge (\bar{s}_{t,i} \vee \bar{b}_{t,l} \vee a_{t+1,i,\tilde{\tilde{j}}_l}) \wedge (\bar{s}_{t,i} \vee \bar{b}_{t,l} \vee s_{t+1,i+M_l})]\}.$$

Für $B_{j_l} = A_{\tilde{j}_l}$ gilt dabei $\tilde{\tilde{j}}_l = \tilde{j}_l$ und $M_l = 0$, da in diesem Fall das Feld i beschrieben wird und keine Bewegung des Kopfes stattfindet. Für $B_{j_l} \in \{r, l\}$ dagegen ist $\tilde{\tilde{j}}_l = j_l$ und $M_l = R_l$. $U(t)$ ist ein Ausdruck in konjunktiver Normalform. In ihm kommen

$$(2p(n) + 1)(3v + 15m) = O(p(n))$$

Variablen vor. Damit ist auch U ein Ausdruck in konjunktiver Normalform, und es treten $O(p(n)^2)$ Variablen in U auf.

Insgesamt haben wir damit einen Ausdruck

$$\alpha(w) = A \wedge R \wedge U \wedge (z_{p(n),r} \vee \ldots \vee z_{p(n),s})$$

in konjunktiver Normalform bestimmt.

Aus $w \in L$ folgt, daß eine akzeptierende Berechnung $C_0, C_1, \ldots, C_{p(n)}$ der Länge $p(n)$ auf w existiert. Diese Berechnung induziert eine Belegung Ψ der Variablen:

$$\Psi(z_{t,k}) = \begin{cases} 1, & \text{falls } z_k \text{ der Zustand in } C_t \text{ ist} \\ 0 & \text{sonst,} \end{cases}$$

$$\Psi(a_{t,i,j}) = \begin{cases} 1, & \text{falls } A_j \text{ das } i\text{-te Zeichen der Bandinschrift von } C_t \text{ ist} \\ 0 & \text{sonst,} \end{cases}$$

$$\Psi(s_{t,i}) = \begin{cases} 1, & \text{falls in } C_t \text{ das } i\text{-te Feld gelesen wird} \\ 0 & \text{sonst,} \end{cases}$$

$$\Psi(b_{t,l}) = \begin{cases} 1, & \text{falls beim Übergang von } C_t \text{ zu } C_{t+1} \text{ die } l\text{-te} \\ & \text{Zeile der Turingtafel von } T \text{ benutzt wird} \\ 0 & \text{sonst.} \end{cases}$$

Nach der Konstruktion von $\alpha(w)$ ist mit dieser Belegung von Ψ der Ausdruck $\alpha(w)$ erfüllt. Umgekehrt ist die Konstruktion von $\alpha(w)$ mit Hilfe von T und $p(n)$ so durchgeführt worden, daß zu jeder erfüllbaren Belegung eine akzeptierende Berechnung von T auf w gehört. Also gilt $w \in L$ genau dann, wenn $\alpha(w)$ erfüllbar ist.

Zu zeigen bleibt, daß die Konstruktion von $\alpha(w)$ aus w mit einer deterministischen Turingmaschine in Polynomialzeit möglich ist. Wir führen zunächst Überlegungen zur

Länge der Darstellung von $\alpha(w)$ durch. In $\alpha(w)$ treten $O(p(n)^3)$ Variablen auf, und $\alpha(w)$ wird in Standardkodierung aufgeschrieben. Dafür werden $z_{t,k}$, $a_{t,i,j}$, $s_{t,i}$ und $b_{t,l}$ mit den Variablen $x_1, \ldots, x_q$ identifiziert. Die Bereiche der Variablen zeigen, daß $q = O(p(n)^2)$ gilt. Folglich haben die Indizes in Binärdarstellung die Länge

$$O(\log p(n)^2) = O(\log p(n)).$$

Außerdem ist die Anzahl der Zeichen $\wedge$, $\vee$, $\neg$, „)" und „(" zu der Anzahl der Variablen proportional. Also ist die Länge der Standardkodierung von $\alpha(w)$

$$O(p(n)^3 \cdot \log p(n)).$$

Intuitiv erkennt man, daß $\alpha(w)$ bei Kenntnis von T und p aus w leicht zu konstruieren ist. Nach der Berechnung von $p(|w|)$ in polynomialer Zeit können die Ausdrücke A, R und U und damit auch $\alpha(w)$ in polynomialer Zeit aufgeschrieben werden. Formal muß diese Konstruktion durch eine deterministische Turingmaschine erfolgen. Dabei müssen die gegebene Turingmaschine T und das Polynom p im Zustand und damit auch in der Überführungsfunktion der deterministischen Turingmaschine berücksichtigt werden. Sie muß zum einen $p(|w|)$ berechnen. Zum anderen muß sie dann unter weiterer Berücksichtigung von T den Ausdruck $\alpha(w)$ der Länge $O(p(|w|)^3 \cdot \log p(|w|))$ aufschreiben. Man erkennt, daß man mit einigem Aufwand eine solche deterministische Turingmaschine für jede nichtdeterministische Turingmaschine T und jedes Polynom p konstruieren kann. Für jedes w berechnet sie $\alpha(w)$ in polynomialer Zeit. Daraus folgt, daß SAT NP-vollständig ist. □

Die schon zu Beginn von Abschnitt 7.1 angesprochene Idee, daß eine Sprache zu NP gehört, wenn die angebotenen Lösungen in polynomialer Zeit überprüft werden können, wollen wir durch den Begriff des Zertifikats formalisieren. Zum Beispiel ist bei 3-FÄRBBARKEIT ein Zertifikat eines ungerichteten Graphen eine beliebige Färbung seiner Knoten mit drei Farben, unabhängig davon, ob sie eine korrekte 3-Färbung liefert oder nicht.

Definition 7.2.2 Es seien Σ, Γ Alphabete. Eine Sprache $L_{\text{check}} \subset (\Sigma \cup \Gamma)^*$ heißt *Zertifikatensprache* für die Sprache $L \subset \Sigma^*$, wenn die folgenden Eigenschaften gelten:

(a) $L_{\text{check}} \in P$, und
(b) es existiert ein Polynom p, so daß für alle $w \in \Sigma^*$ die Äquivalenz

$$w \in L \iff \text{es existiert } z \in \Gamma^*, |z| \leq p(|w|), \text{ mit } wz \in L_{\text{check}}$$

erfüllt ist. □

Wenn wir intuitiv unter w einen Fall des durch die Sprache L gegebenen Problems verstehen, so können wir uns unter z eine angebotene Lösung dieses Falls vorstellen .

Satz 7.2.3 Eine Sprache L gehört genau dann zur Klasse NP, wenn sie eine Zertifikatensprache besitzt.

Beweis: Wir gehen zunächst davon aus, daß $L_{\text{check}} \subset (\Sigma \cup \Gamma)^*$ eine Zertifikatensprache von $L \subset \Sigma^*$ ist. Es sei T eine deterministische Turingmaschine, die mit polynomialer

Zeitkomplexität q diese Zertifikatensprache akzeptiert. Wir konstruieren eine nichtdeterministische Zweiband-Turingmaschine $\hat{T}$. Sie beginnt mit dem Wort $w \in \Sigma^*$ auf Band 1. Dann berechnet sich $\hat{T}$ auf dem zweiten Band in polynomialer Zeit in bezug auf $|w|$ den Wert $p(|w|)$. Anschließend schreibt sie nichtdeterministisch hinter das Wort w ein Wort $z \in \Gamma^*$ mit $|z| \le p(|w|)$. Auf diesem Wort wz des ersten Bandes wird die Turingmaschine T simuliert. Falls T hält und akzeptiert, hält auch $\hat{T}$ und akzeptiert. Das bedeutet, daß $\hat{T}$ genau dann ein Wort w akzeptiert, wenn es ein $z \in \Gamma^*$ mit $|z| \le p(|w|)$ und $wz \in L_{\text{check}}$ gibt. Wir erhalten also $L(\hat{T}) = L$. Die Erzeugung von z erfordert $O(p(|w|))$ Schritte und die Simulierung von T $O(q(|w| + |z|)) = O(q(|w| + p(|w|)))$ Schritte. Insgesamt hat also $\hat{T}$ einen polynomialen Zeitbedarf. Somit erhalten wir $L \in NP$.

Umgekehrt konstruieren wir für jede Sprache $L \subset \Sigma^*$ eine Zertifikatensprache. Wir gehen von einer nichtdeterministischen Turingmaschine T mit $L(T) = L$ aus, deren Zeitkomplexität durch ein Polynom q bestimmt ist. Für jedes Wort $w \in \Sigma^*$ gibt es nach dem Beweis von Satz 7.2.2 einen Booleschen Ausdruck $\alpha(w)$ der Länge $O((q(|w|))^3 \log q(|w|))$ in Standardkodierung, der genau dann erfüllbar ist, wenn es eine akzeptierende Berechnung von w gibt. Es existiert eine Konstante k, so daß $|\alpha(w)|$ durch $k(q(|w|))^4$ beschränkt ist. Dies ist auch eine Schranke für die Anzahl der Variablen in $\alpha(w)$. Eine Belegung $\bar{\psi}$ von $\alpha(w)$ kann durch ein Wort $\psi \in \{0,1\}^*$ dargestellt werden, wobei z.B. $\bar{\psi}(x1)$ das erste Symbol von ψ ist, $\bar{\psi}(x10)$ das zweite Symbol, usw. Wir wählen $\Gamma = \{0,1\}$ und nehmen ohne Beschränkung der Allgemeinheit an, daß $\Sigma \cap \Gamma = \emptyset$ gilt. Wir definieren die Sprache $L_{\text{check}} \subset (\Sigma \cup \Gamma)^*$ durch

$$L_{\text{check}} = \{w\psi \mid w \in \Sigma^*, \psi \in \{0,1\}^* \text{ mit } \bar{\psi}(\alpha(w)) = 1 \text{ und } |\psi| \le k \cdot (q(|w|))^4\}.$$

Wir müssen beweisen, daß L_{check} die Bedingungen aus Definition 7.2.2 erfüllt. $L_{\text{check}} \in P$ gilt, falls für ein beliebiges $u \in (\Sigma \cup \Gamma)^*$ die Zugehörigkeit zu L_{check} in polynomialer Zeit überprüft werden kann. Ob u die Gestalt $w\psi$ mit $w \in \Sigma^*$ und $\psi \in \Gamma^*$, $|\psi| \le k(q(|w|))^4$, hat, kann in polynomialer Zeit bezüglich der Länge von u festgestellt werden. In polynomialer Zeit kann dann $\alpha(w)$ bestimmt und anschließend $\bar{\psi}(\alpha(w)) = 1$ überprüft werden. Die zweite Bedingung der Definition 7.2.2 folgt aus der Konstruktion von L_{check}, wenn das Polynom p der Definition hier speziell zu $k \cdot q^4$ gewählt wird. □

7.3 Weitere *NP*-vollständige Probleme

Satz 7.2.2 hat gezeigt, daß das Problem SAT und damit das allgemeine Erfüllbarkeitsproblem der Aussagenlogik *NP*-vollständig ist. Wir wissen nach Satz 6.3.6, daß jedoch der Spezialfall SAT(2) zur Klasse P gehört. Aber schon das Erfüllbarkeitsproblem für Boolesche Ausdrücke in konjunktiver Normalform, die drei Literale pro Klausel enthalten, also SAT(3), ist *NP*-vollständig.

Satz 7.3.1 SAT(3) ist *NP*-vollständig.

Beweis: Wegen SAT $\in NP$ gilt erst recht SAT(3) $\in NP$. Nach Satz 7.2.1(b) bleibt SAT $\le$ SAT(3) zu zeigen. Wir ersetzen im folgenden eine beliebige Klausel $x_1 \vee \ldots \vee x_n$ durch eine Konjunktion von Klauseln vom Grad ≤ 3. Es sei $Y = \{y_1, \ldots, y_n\}$ eine

Menge neuer Variablen, und wir betrachten den Ausdruck

$$\alpha = (x_1 \vee \neg y_1) \wedge (y_1 \vee x_2 \vee \neg y_2) \wedge \ldots \wedge (y_{n-1} \vee x_n \vee \neg y_n) \wedge y_n.$$

Wir behaupten: Es sei $\psi : X = \{x_1, \ldots, x_n\} \to \{0,1\}$ eine Belegung. Dann gilt:

$$\psi(x_1 \vee \ldots \vee x_n) = 1 \iff \text{ex. Erweiterung } \varphi : X \cup Y \to \{0,1\} \text{ von } \psi \text{ mit } \varphi(\alpha) = 1.$$

Zum Beweis dieser Behauptung sei zunächst $\psi(x_1 \vee \ldots \vee x_n) = 1$. Wir setzen

$$i_0 = \min\{i \mid \psi(x_i) = 1, i = 1, \ldots, n\} \text{ und definieren}$$

$$\varphi(z) = \begin{cases} \psi(x_j) & \text{für } z = x_j,\, j = 1, \ldots, n \\ 0 & \text{für } z = y_j,\, j = 1, \ldots, i_0 - 1 \\ 1 & \text{für } z = y_j,\, j = i_0, \ldots, n. \end{cases}$$

Wegen $\psi(x_1 \vee \ldots \vee x_n) = 1$ ist $i_0 \leq n$ wohldefiniert. Dann betrachten wir die verschiedenen Klauseln von α. Für die letzte Klausel gilt wegen $i_0 \leq n$

$$\varphi(y_n) = 1.$$

Für die erste Klausel unterscheiden wir zwei Fälle: für $i_0 = 1$ gilt $\varphi(x_1) = 1$ und $\varphi(\neg y_1) = 0$, und für $i_0 \neq 1$ gilt $\varphi(x_1) = 0$ und $\varphi(\neg y_1) = 1$. In jedem Fall erhalten wir also

$$\varphi(x_1 \vee \neg y_1) = 1.$$

Für die übrigen Klauseln gelten die folgenden Überlegungen: für $1 \leq j < i_0 - 1$ ist $\varphi(\neg y_{j+1}) = 1$, für $j = i_0 - 1$ ist $\varphi(x_{j+1}) = \varphi(x_{i_0}) = 1$, und für $i_0 \leq j \leq n-1$ ist $\varphi(y_j) = 1$. Somit folgt für alle j, $1 \leq j \leq n-1$,

$$\varphi(y_j \vee x_{j+1} \vee \neg y_{j+1}) = 1.$$

Insgesamt ergibt sich $\varphi(\alpha) = 1$.

Die umgekehrte Richtung beweisen wir durch Kontraposition. Es sei $\psi(x_1 \vee \ldots \vee x_n) = 0$. Wir nehmen an, daß eine Erweiterung $\varphi : X \cup Y \to \{0,1\}$ von ψ existiert mit $\varphi(\alpha) = 1$. Wegen $\psi(x_1 \vee \ldots \vee x_n) = \varphi(x_1 \vee \ldots \vee x_n) = 0$ folgt $\varphi(x_i) = \psi(x_i) = 0$ für alle i, $1 \leq i \leq n$. Wir betrachten die verschiedenen Klauseln von α. Wegen $\varphi(\alpha) = 1$ und $\varphi(x_1) = 0$ muß $\varphi(\neg y_1) = 1$, also $\varphi(y_1) = 0$ gelten. Wegen $\varphi(x_2) = 0$ ist dann $\varphi(\neg y_2) = 1$ und damit $\varphi(y_2) = 0$. Wir erhalten schließlich $\varphi(\neg y_n) = 1$ und so $\varphi(y_n) = 0$, also den Widerspruch $\varphi(\alpha) = 0$.

Offenbar ist α in Polynomialzeit aus $x_1 \vee \ldots \vee x_n$ konstruierbar. Die Konstruktion kann auf beliebige Ausdrücke $w = c_1 \wedge \cdots \wedge c_k$, $k \in \mathbb{N}$, in konjunktiver Normalform erweitert werden. Für jede Klausel c_i, $i = 1, \ldots, k$, wird dabei entsprechend dem obigen Vorgehen ein Ausdruck α_i konstruiert, wobei wir annehmen können, daß in verschiedenen α_i auch verschiedene Variablen y_i eingeführt werden. Der so bestimmte Ausdruck $\alpha_1 \wedge \cdots \wedge \alpha_k$ kann in Polynomialzeit konstruiert werden.

Ist w erfüllbar, so haben bei der entsprechenden Belegung ψ alle Klauseln c_i den Wert 1. Dann kann wie oben für jedes i, $i = 1, \ldots, k$, eine Erweiterung φ_i von ψ mit $\varphi_i(\alpha_i) = 1$ konstruiert werden. Wegen $\varphi_i(x_j) = \psi(x_j)$ für alle $i = 1, \ldots, k$ und $j = 1, \ldots, n$ ergibt sich daraus eine Erweiterung φ von ψ und allen φ_i, für die offenbar $\varphi(\alpha_1 \wedge \cdots \wedge \alpha_k) = 1$ gilt. Ist w nicht erfüllbar, so existiert für jede Belegung ψ eine

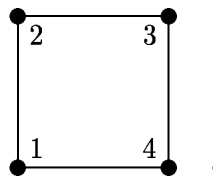
Klausel c_i mit $\psi(c_i) = 0$. Für jede Erweiterung φ folgt aus der Annahme $\varphi(\alpha_1 \wedge \ldots \wedge \alpha_k) = 1$ wie oben $\varphi(\alpha_i) = 0$ und damit der Widerspruch $\varphi(\alpha_1 \wedge \ldots \wedge \alpha_k) = 0$. □

Die polynomialzeit-beschränkte Transformation von SAT auf SAT(3) im Beweis von Satz 7.3.1 ist nicht explizit durch eine Turingmaschine angegeben worden. Aufgrund der Churchschen These und unseren Überlegungen zur Robustheit der Klassen *P* und *FP* ist es jedoch klar, daß eine solche Turingmaschine konstruiert werden kann.

Nach Satz 7.2.1(b) kann der Nachweis der *NP*-Vollständigkeit weiterer Probleme mit Hilfe eines schon bekannten *NP*-vollständigen Problems geführt werden. Die notwendigen Algorithmen werden üblicherweise nur intuitiv angegeben. Aufgrund seiner beschränkten Struktur ist für diesen Nachweis SAT(3) geeigneter als SAT. Dies wollen wir speziell am Clique-Problem sowie am Problem der 3-Färbbarkeit zeigen.

Definition 7.3.1 Das Problem CLIQUE ist wie folgt gegeben: Es sei $G = (V, E)$ ein Graph mit der Knotenmenge V und der Kantenmenge E, und es sei $k \in \mathbb{N}$ mit $k \leq |V|$. Enthält G eine Clique der Größe k, d.h., existiert eine Teilmenge $V' \subset V$ mit $|V'| = k$, so daß je zwei Knoten in V' durch eine Kante in E verbunden sind? □

Beispiel 7.3.1 Wir betrachten den Graphen



Es existiert keine Clique der Größe ≥ 3. Es ist z.B. $\{1, 2\}$ eine Clique der Größe 2, $\{1, 3\}$ jedoch nicht. □

Satz 7.3.2 CLIQUE ist *NP*-vollständig.

Beweis: Für einen Graphen $G = (V, E)$ wird nichtdeterministisch eine Teilmenge $V' \subset V$ erraten. Dann kann deterministisch in quadratischer Zeit bezüglich der Anzahl der Knoten von V überprüft werden, ob V' eine Clique ist. Das Problem gehört also zur Klasse *NP*.

Weiter betrachten wir für $k \in \mathbb{N}$ einen Ausdruck in konjunktiver Normalform mit höchstens drei Literalen pro Klausel. Ohne Beschränkung der Allgemeinheit gehen wir dabei von genau drei Literalen pro Klausel aus. Anderenfalls können fehlende Literale geeignet ergänzt werden, zum Beispiel kann man $x_1 \vee \neg x_2$ durch $x_1 \vee x_1 \vee \neg x_2$ ersetzen. Es sei also

$$\alpha = c_1 \wedge \ldots \wedge c_k \quad \text{mit} \quad c_i = x_{i1}^{\delta_{i1}} \vee x_{i2}^{\delta_{i2}} \vee x_{i3}^{\delta_{i3}}, \delta_{ij} \in \{0, 1\}, i = 1, \ldots, k, j = 1, 2, 3.$$

Dabei sind die x_{ij} Variablen, und es gilt $x_{ij}^0 = \neg x_{ij}$ und $x_{ij}^1 = x_{ij}$. Zu α konstruieren wir einen Graphen $G(\alpha) = (V, E)$ mit $V = \{u_{ij} \mid 1 \leq i \leq k, 1 \leq j \leq 3\}$ und

$$(u_{ij}, u_{i'j'}) \in E \iff i \neq i' \text{ und } (x_{ij} \neq x_{i'j'} \text{ oder } \delta_{ij} = \delta_{i'j'}).$$

Wir beweisen die Äquivalenz

$$\alpha \text{ erfüllbar} \iff G(\alpha) \text{ besitzt eine Clique der Größe } k.$$

Es sei α erfüllbar. Dann existiert eine Belegung Ψ der Variablen mit $\Psi(\alpha) = 1$. Folglich existiert zu jedem i, $1 \leq i \leq k$, ein $j(i)$, $1 \leq j(i) \leq 3$, mit $\Psi(x_{ij(i)}^{\delta_{ij(i)}}) = 1$. Wir setzen $V' = \{u_{ij(i)} \mid i = 1, \ldots, k\}$. Es seien $u_{ij(i)}, u_{i'j(i')} \in V'$ Knoten mit $u_{ij(i)} \neq u_{i'j(i')}$. Dann ist $i \neq i'$. Falls $x_{ij(i)} = x_{i'j(i')}$ gilt, muß wegen $\Psi(x_{ij(i)}^{\delta_{ij(i)}}) = \Psi(x_{i'j(i')}^{\delta_{i'j(i')}}) = 1$ die Gleichung $\delta_{ij(i)} = \delta_{i'j(i')}$ folgen. Somit ist $(u_{ij(i)}, u_{i'j(i')}) \in E$ und V' eine Clique der Größe k.

Umgekehrt sei $V' \subset V$ eine Clique der Größe k. Aus der Cliqueneigenschaft und der Eigenschaft $i \neq i'$ in der Definition von E folgt dann $V' = \{u_{ij(i)} \mid 1 \leq i \leq k\}$ mit einem geeigneten $j(i)$, $1 \leq j(i) \leq 3$, für alle i, $1 \leq i \leq k$. Wir definieren eine Belegung Ψ der Variablen durch

$$\Psi(x) = \begin{cases} 1, & \text{falls } x = x_{ij(i)} \text{ für ein } i \text{ mit } \delta_{ij(i)} = 1 \\ 0, & \text{falls } x = x_{ij(i)} \text{ für ein } i \text{ mit } \delta_{ij(i)} = 0 \\ 0 & \text{sonst.} \end{cases}$$

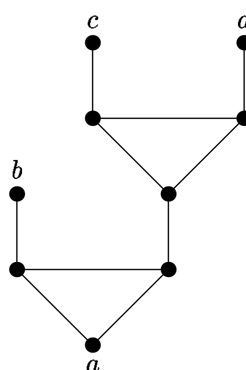
Ψ ist wohldefiniert, denn wäre $x = x_{ij(i)} = x_{i'j(i')}$ mit $\delta_{ij(i)} \neq \delta_{i'j(i')}$, so würde nach Definition von E folgen, daß $(u_{ij(i)}, u_{i'j(i')}) \notin E$ gilt und damit V' keine Clique ist. Die Alternative zwischen 1 und 0 ist gerade so gewählt, daß $\Psi(x_{ij(i)}^{\delta_{ij(i)}}) = 1$ für alle i, $1 \leq i \leq k$, folgt. Wir erhalten $\Psi(\alpha) = 1$ und damit die Erfüllbarkeit von α.

Die Konstruktion von $G(\alpha)$ aus α kann offenbar in Polynomialzeit bezüglich der Länge der Darstellung von α durchgeführt werden. □

Satz 7.3.3 3-FÄRBBARKEIT ist *NP*-vollständig.

Beweis: 3-FÄRBBARKEIT liegt in *NP*, denn für eine vorgeschlagene 3-Färbung der Knoten können wir durch Besuch aller Knoten bei Überprüfung ihrer Farben und der ihrer jeweiligen Nachbarknoten in polynomialer Zeit entscheiden, ob sie korrekt ist. Bei geeigneter Implementierung durch Adjazenzlisten ist dies sogar in linearer Zeit in bezug auf die Länge der Darstellung möglich. Für den Beweis der *NP*-Vollständigkeit reicht es dann nach Satz 7.2.1(b) aus, die Gültigkeit der Relation SAT(3) $\leq$ 3-FÄRBBARKEIT zu zeigen.

Bei unserem Beweis benutzen wir den folgenden Hilfsgraphen:



Offenbar ist er 3-färbbar, sagen wir mit den Farben

rot, weiß, blau.

Diese Färbung besitzt die folgenden Eigenschaften.

(1) Falls b,c,d rot gefärbt sind, muß es auch a sein.

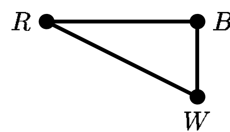
(2) Falls einer der Knoten b, c oder d weiß ist, darf auch a weiß gefärbt werden.

Dies ist leicht zu überprüfen. Im Fall (1) ist keiner der c oder d benachbarten Knoten rot. Folglich muß die untere Spitze des rechten Dreiecks rot sein. Mit demselben Argument ist dann die untere Spitze des linken Dreiecks, also der Knoten a, rot gefärbt. In ähnlicher Weise läßt sich auch für Fall (2) argumentieren.

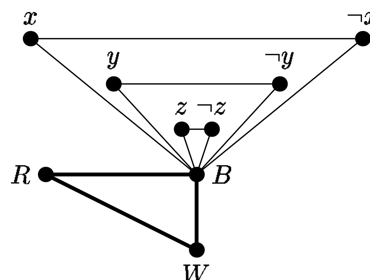
Zum Nachweis der obigen Relation konstruieren wir für jeden Booleschen Ausdruck α mit genau 3 Literalen pro Klausel einen Graphen G_α, so daß

$$\alpha \text{ erfüllbar} \iff G_\alpha \text{ ist 3-färbbar}$$

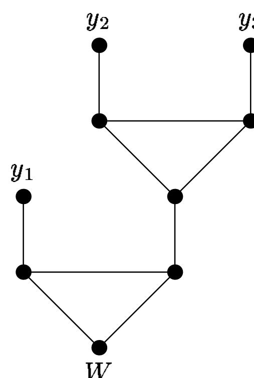
gilt. Der Graph G_α hat drei verschiedene Typen von Knoten. Es gibt genau drei Knoten des Typs 1, die mit den drei Farben markiert sind und das Dreieck



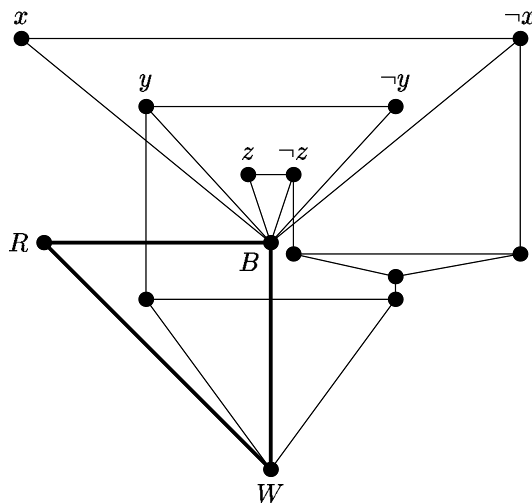
bilden. Jeweils zwei Knoten des Typs 2 erhalten wir für jede Variable x des Ausdrucks α. Diese Knoten werden mit x bzw. $\neg x$ markiert und bilden jeweils mit dem mit B markierten Knoten des Typs 1 ein Dreieck. Zum Beispiel erhalten wir bei den Variablen x, y, z den Graphen



Schließlich bestimmt jede Klausel $y_1 \vee y_2 \vee y_3$ von α einen Teilgraphen



entsprechend dem obigen Hilfsgraphen. Der mit W markierte Knoten wird mit dem entsprechenden Knoten des Typs 1 identifiziert. Die Knoten y_1, y_2 und y_3 werden mit den so markierten Knoten des Typs 2 identifiziert. Die übrigen Knoten sind vom Typ 3. Als Beispiel betrachten wir die Klausel $\alpha = y \vee \neg x \vee \neg z$, die zum Graphen



führt. Für jede weitere Klausel wird ein weiterer Hilfsgraph zwischen dem mit W markierten Knoten und den Knoten des Typs 2 eingehängt. Damit ist insgesamt G_α definiert.

Wir zeigen zunächst, daß aus der Erfüllbarkeit des Ausdrucks α die 3-Färbbarkeit von G_α folgt. Es sei Ψ eine Belegung der Variablen mit $\Psi(\alpha) = 1$. Dann werden die Knoten des Typs 1 rot (R), weiß (W) bzw. blau (B) gefärbt, so wie sie zuvor bereits markiert worden sind. Ein Knoten des Typs 2 mit der Markierung y wird weiß gefärbt, falls $\Psi(y) = 1$ gilt, für $\Psi(y) = 0$ wird er rot gefärbt. Da jede Klausel $y_1 \vee y_2 \vee y_3$ erfüllbar ist, muß mindestens einer der durch y_1, y_2 oder y_3 markierten Knoten des entsprechenden Teilgraphen (die b, c, d des Hilfsgraphen entsprechen) weiß gefärbt sein. Nach der Eigenschaft (2) des Hilfsgraphen kann dann eine 3-Färbung des Teilgraphen gefunden werden, bei der d weiß gefärbt wird, also in Übereinstimmung mit der zuvor angegebenen Färbung des entsprechenden Knotens vom Typ 1. Die Farben der Knoten vom Typ 3 in einem Teilgraphen werden nur in Abhängigkeit von den Knoten des Typs 1 und 2 in diesem Teilgraphen gefärbt, da sie nicht zu Knoten des Typs 3 in anderen Teilgraphen benachbart sind. Insgesamt haben wir damit eine 3-Färbung von G_α gefunden.

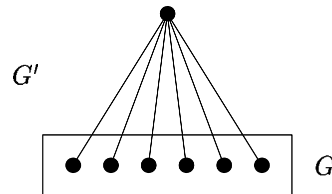
Für die umgekehrte Beweisrichtung gehen wir von einer 3-Färbung der Knoten von G_α aus. Dabei sollen die Knoten des Typs 1 so gefärbt werden, wie es ihren Markierungen in den bildlichen Darstellungen entspricht. Wir setzen dann

$$\Psi(y) = 1 \iff \text{der mit } y \text{ markierte Knoten des Typs 2 ist weiß.}$$

Wir beweisen, daß $\Psi(\alpha) = 1$ gilt, das heißt, daß in jeder Klausel $y_1 \vee y_2 \vee y_3$ von α mindestens eines der Literale y_i, $i = 1, 2, 3$, den Wert $\Psi(y_i) = 1$ hat. Da von jedem Knoten des Typs 2 eine Kante zu dem blau gefärbten Knoten des Typs 1 führt, kann kein Knoten des Typs 2 blau gefärbt werden. Folglich sind sie weiß oder rot gefärbt. Aufgrund der Eigenschaft (1) des Hilfsgraphen können die mit y_1, y_2 und y_3 markierten Knoten nicht gleichzeitig rot sein. Mindestens einer von ihnen, sagen wir y_{i_0}, ist also weiß gefärbt und liefert dann $\Psi(y_{i_0}) = 1$.

Die Konstruktion von G_α aus dem Booleschen Ausdruck α kann offenbar in polynomialer Zeit, ja sogar in linearer Zeit $O(n)$, bezüglich der Länge n der Darstellung von α durchgeführt werden. □

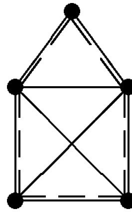
Analog 2- und 3-FÄRBBARKEIT kann auch k-FÄRBBARKEIT für beliebige $k \in \mathbb{N}$, $k > 3$, definiert werden. Alle diese Probleme sind *NP*-vollständig. Das läßt sich dadurch zeigen, daß $(k-1)$-FÄRBBARKEIT auf k-FÄRBBARKEIT in polynomialer Zeit reduziert wird, indem ein Graph G durch einen neuen Knoten und durch Kanten von diesem neuen Knoten zu jedem Knoten des Graphen G zu einem neuen Graphen G' ergänzt wird.



Offenbar ist G genau dann $(k-1)$-färbbar, wenn G' k-färbbar ist.

Definition 7.3.2 Das Problem HAMILTONSCHER KREIS ist wie folgt gegeben: Es sei $G = (V, E)$ ein ungerichteter Graph. Existiert in G ein Hamiltonscher Kreis, d.h., existiert eine Ordnung $(v_1, \ldots, v_n)$ aller Knoten, so daß $(v_n, v_1) \in E$ und $(v_i, v_{i+1}) \in E$ für $i = 1, \ldots, n-1$ gilt? □

Beispiel 7.3.2 Ein Hamiltonscher Kreis des folgenden Graphen ist durch die zusätzliche Strichelung der Kanten dargestellt.



Den Beweis des folgenden Satzes findet man zum Beispiel in [7], Theorem 3.4, S. 56–60.

Satz 7.3.4 HAMILTONSCHER KREIS ist *NP*-vollständig. □

Daraus folgt

Satz 7.3.5 TSP ist *NP*-vollständig.

Beweis: Wir haben in Satz 7.1.4 festgestellt, daß TSP $\in NP$ gilt. Nach Satz 7.2.1(b) reicht es, eine Polynomialzeit-Reduktion von HAMILTONSCHER KREIS auf TSP zu finden. Dies ist jedoch trivial. Für jeden ungerichteten Graphen $G = (V, E)$ mit n Knoten $v_1, \ldots, v_n$ setzen wir durch

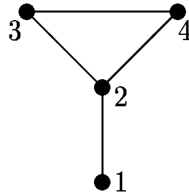
$$d_{ij} = \begin{cases} 1 & \text{falls } (v_i, v_j) \in E \\ 2 & \text{sonst} \end{cases}$$

die Kostenmatrix fest. Offenbar hat dieses Traveling-Salesman-Problem genau dann eine Lösung mit Gesamtkosten $\leq n$ (d.h. hier mit Kosten $= n$), wenn G einen Hamiltonschen Kreis besitzt. □

Wir definieren noch zwei weitere Probleme.

Definition 7.3.3 Das Problem KNOTENÜBERDECKUNG eines Graphen ist wie folgt gegeben: Es sei $G = (V, E)$ ein ungerichteter Graph, und es sei $k \in \mathbb{N}$ mit $k \leq |V|$. Existiert eine Knotenüberdeckung der Größe $\leq k$ für G, d.h., existiert ein $V' \subset V$ mit $|V'| \leq k$, so daß für jede Kante $(u, v) \in E$ $u \in V'$ oder $v \in V'$ gilt? □

Beispiel 7.3.3 Wir betrachten den Graphen



Dabei ist $\{2, 3\}$ eine Knotenüberdeckung der Größe 2. Für $k = 1$ existiert hier keine Knotenüberdeckung. □

Definition 7.3.4 Das Problem PARTITION ist wie folgt gegeben: Es sei A eine endliche Menge und $s : A \to \mathbb{N}$ eine Abbildung. Existiert ein $A' \subset A$ mit

$$\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)? \quad \square$$

Man kann zeigen, daß das KNAPSACK-PROBLEM aus Definition 7.1.4 sowie die eben definierten Probleme PARTITION und KNOTENÜBERDECKUNG ebenfalls NP-vollständig sind. Das bedeutet, daß für sie wie für mehr als tausend andere NP-vollständige Probleme keine effizienten Algorithmen bekannt sind. Darunter sind auch viele praktisch sehr wichtige wie z.B. TSP. Das Problem ZERLEGBARKEIT aus Beispiel 7.1.1 ist eines der wenigen bekannten Probleme, für die kein effizienter Algorithmus gefunden wurde und für die man nicht weiß, ob sie NP-vollständig sind. Falls jedoch $P \neq NP$ gilt, dann kann man zeigen, daß in NP Probleme existieren, die nicht NP-vollständig sind.

8 Komplexität von Optimierungsalgorithmen

8.1 Optimierungsprobleme

Viele Probleme sind weder als Entscheidungs- noch direkt als Berechnungsprobleme zu verstehen, sondern als Probleme, die eine maximale oder minimale Lösung aus einer größeren Gesamtheit von möglichen Lösungen mit verschiedenen Parametern verlangen. Solche Maximierungs- oder Minimierungsprobleme heißen auch *Optimierungsprobleme*. Wir betrachten zunächst

Definition 8.1.1 Das Problem MINIMALE FÄRBBARKEIT ist wie folgt gegeben: Es sei G ein ungerichteter Graph. Gesucht ist eine Färbung von G mit einer minimalen Anzahl von Farben. □

Gibt es für dieses Problem eine effiziente Lösung? Sicherlich nicht, falls $P \neq NP$ gilt, denn jeder Algorithmus für MINIMALE FÄRBBARKEIT kann auch das *NP*-vollständige Problem 3-FÄRBBARKEIT lösen. Wenn nämlich der Algorithmus für MINIMALE FÄRBBARKEIT eine Färbung mit einer, zwei oder drei Farben liefert, ist die Antwort auf die Frage in 3-FÄRBBARKEIT „ja", sonst „nein". Für jedes Minimierungsproblem M gibt es das zugrunde liegende Entscheidungsproblem $\hat{M}$, dessen Eingabe ein Paar (x, y) ist, wobei x die Eingabe von M und y eine Zahl ist. Die Entscheidung lautet, ob $\hat{M}$ eine Lösung der Eingabe x mit einem Parameter $\leq y$ besitzt. Es ist intuitiv klar, daß ein Minimierungsproblem mit einer effizienten Lösung auch die Eigenschaft hat, daß das zugrunde liegende Entscheidungsproblem zur Klasse P gehört. Analoge Überlegungen gelten für Maximierungsprobleme.

Definition 8.1.2 Ein *Optimierungssproblem* ist ein 5-Tupel (Σ, I, S, π, c), das folgende Eigenschaften erfüllt:

(a) Σ ist ein Alphabet (zur Kodierung von Fällen des Problems (Eingaben) und von Lösungen).
(b) $I \subset \Sigma^*$ (I ist die Menge der Kodierungen aller möglichen Fälle des Problems).
(c) $S : I \to \mathcal{P}'(\Sigma^*)$ ist eine Abbildung, die jedem $x \in I$ eine nichtleere endliche Menge von Wörtern zuordnet, die Kodierungen für mögliche Lösungen von x sind. Dabei nehmen wir an, daß für jeden Fall x eine korrekte Lösung aus $S(x)$ existiert.
(d) $\pi : I \times S(I) \to \{0, 1\}$ ist ein Prädikat, das für alle $x \in I$ und $y \in S(x)$ den Wert $\pi(x, y) = 1$ genau dann zuordnet, wenn y eine korrekte Lösung von x ist.
(e) $c : I \times S(I) \to \mathbb{N}_0$ ist eine Abbildung, die jedem $x \in I$ und jeder korrekten Lösung y von x das *Maß* $c(x, y)$ *der Lösung* y zuordnet.

Ein Optimierungsproblem kann ein *Minimierungs-* oder *Maximierungsproblem* sein. Das Lösen des Minimierungsproblems für einen gegebenen Fall $x \in I$ besteht im Finden einer minimalen Lösung für x, d.h. im Finden einer korrekten Lösung $y \in S(x)$ mit

$$c(x, y) = \min\{c(x, z) \mid z \in S(x), \pi(x, z) = 1\}.$$

Ein *Maximierungsproblem* ist analog definiert, wobei in der letzten Gleichung min durch max ersetzt wird.

Eine deterministische Turingmaschine T löst das Optimierungsproblem (Σ, I, S, π, c), wenn sie für jedes $x \in I$ die Kodierung einer optimalen Lösung für x berechnet. Diese wird auch mit $\mathrm{OPT}(x)$ bezeichnet. □

Definition 8.1.3 Das Problem MAXIMALE CLIQUE ist wie folgt gegeben: Es sei G ein Graph. Finde eine Clique maximaler Größe in G. □

Definition 8.1.4 Das Problem MINIMALES TSP ist wie folgt gegeben: Gegeben seien Städte $S_1, S_2, \ldots, S_n$, $n \in \mathbb{N}$, sowie eine Kostenmatrix

$$x = (d(i,j)), i, j = 1, \ldots, n, \text{ mit } d(i,j) \in \mathbb{N}_0,$$

die die Kosten der Reise von S_i nach S_j darstellt. Gesucht ist eine Rundfahrt, also eine Permutation $y = (i_1, i_2, \ldots, i_n)$ der Zahlen $1, \ldots, n$, so daß

$$c(x,y) = d(i_1, i_2) + d(i_2, i_3) + \ldots + d(i_{n-1}, i_n) + d(i_n, i_1)$$

minimal wird. □

Beispiel 8.1.1 (a) MINIMALE FÄRBBARKEIT ist ein Minimierungsproblem. Dabei ist I die Menge der (kodierten) Graphen. S ordnet jedem Fall G des Problems, also jedem Graphen, die Menge aller Färbungen seiner Knoten zu. Ist G durch seine Adjazenzmatrix gegeben, so können wir diese binär, also über $\Sigma = \{0, 1\}$, kodieren. Die Färbungen können zum Beispiel durch $0^{i_1}10^{i_2}1 \ldots 10^{i_n}1$ kodiert werden, wobei der Knoten 1 mit der Farbe i_1 gefärbt wird, der Knoten 2 mit der Farbe i_2 und schließlich der Knoten n mit der Farbe i_n. Ohne Beschränkung der Allgemeinheit kann angenommen werden, daß $1 \leq i_\nu \leq n$ für $\nu = 1, \ldots, n$ gilt. Für jeden Graphen G und jede Färbung $F \in S(G)$ gilt $\pi(G, F) = 1$ genau dann, wenn F eine korrekte Färbung von G ist, je zwei benachbarte Knoten also verschieden gefärbt sind. In diesem Fall bezeichnet $c(G, F)$ die Anzahl der in F verwendeten Farben. □

(b) MAXIMALE CLIQUE ist ein Maximierungsproblem. Dabei ist I die Menge aller Graphen. S ordnet jedem Fall $G \in I$ die Menge aller Teilmengen der Knoten von G zu. Für jeden Graphen G und jede Teilmenge $V' \in S(G)$ gilt genau dann $\pi(G, V') = 1$, wenn V' eine Clique für G ist. Für jedes $G \in I$ und jede Clique $V' \in S(G)$ bezeichnet $c(G, V') = |V'|$ die Größe der Clique V'.

(c) Ähnlich einfach kann gezeigt werden, daß MINIMALES TSP ein Minimierungsproblem ist. □

Wir führen die Klassen *NPO* und *PO* von Optimierungsproblemen ein, die in gewisser Weise den Klassen von Entscheidungsproblemen *NP* und *P* entsprechen.

Definition 8.1.5 Ein Optimierungsproblem (Σ, I, S, π, c) gehört zur Klasse *NPO*, wenn die folgenden Bedingungen erfüllt sind:

(a) $I \in P$.

(b) Es existiert ein Polynom p, so daß für jedes $x \in I$ bei beliebigem $y \in S(x)$ die Relation $|y| \leq p(|x|)$ gilt.

(c) Für jedes $x \in I$ und jedes y mit $|y| \leq p(|x|)$ ist in polynomialer Zeit entscheidbar, ob $y \in S(x)$ gilt.
(d) Das Prädikat π ist in polynomialer Zeit entscheidbar (d.h., die Funktion π gehört zur Klasse *FP*).
(e) Die Funktion c gehört zur Klasse *FP*. □

Beispiel 8.1.2 (a) MINIMALE FÄRBBARKEIT gehört zu *NPO*. Die Bedingungen aus Definition 8.1.5 sind erfüllt:

- (a) Es gilt $I \in P$, denn für ein gegebenes x kann in polynomialer Zeit überprüft werden, ob x die Kodierung der Adjazenzmatrix eines Graphen ist.
- (b) Für einen Graphen $G \in I$ in der Darstellung als Adjazenzmatrix ist die Länge einer beliebigen Färbung (dargestellt wie in Beispiel 8.1.1) offenbar beschränkt durch $k \cdot |G|$ mit einer von G unabhängigen Konstanten k. Man beachte, daß bei n Knoten $|G|$ die Ordnung $O(n^2)$ hat.
- (c) Für jeden Graphen G und jedes vorgelegte Wort y mit $|y| \leq k \cdot |G|$ kann in linearer Zeit überprüft werden, ob y eine Färbung von G ist.
- (d) Für eine Färbung F eines Graphen G kann in polynomialer Zeit überprüft werden, ob F eine korrekte Färbung für G ist.
- (e) Für jeden Graphen G und jede korrekte Färbung F von G kann in polynomialer Zeit die Anzahl der verwendeten Farben berechnet werden.

(b) MAXIMALE CLIQUE gehört zu *NPO*:

- (a) Es gilt $I \in P$ wie in Beispiel 8.1.2(a).
- (b) Für jede Teilmenge V' von Knoten eines Graphen G gilt offenbar $|V'| \leq |G|$.
- (c) Für jeden Graphen G und jedes $y \in \Sigma^*$ mit $|y| \leq |G|$ ist in polynomialer Zeit entscheidbar, ob y die Kodierung einer Teilmenge der Knoten von G ist.
- (d) Für eine Teilmenge V' der Knoten eines Graphen G ist in polynomialer Zeit entscheidbar, ob V' eine Clique ist.
- (e) Die Anzahl der Knoten der Clique kann in linearer Zeit berechnet werden.

(c) Daß MINIMALES TSP zu *NPO* gehört, kann ebenso unmittelbar überprüft werden. □

Definition 8.1.6 *PO* ist die Klasse der Optimierungsprobleme aus *NPO*, die in polynomialer Zeit durch deterministische Turingmaschinen gelöst werden. □

Wir wollen ein Maximierungsproblem der Klasse *PO* angeben. Zunächst sind einige Definitionen erforderlich.

Definition 8.1.7 Es sei $G = (V, E)$ ein ungerichteter Graph. Eine Kantenmenge $E' \subset E$ heißt *Matching* in G, wenn verschiedene Kanten in E' keinen gemeinsamen Knoten besitzen. Ein Matching E' heißt *maximales Matching* (maximale Zuordnung) in G, wenn es unter allen Matchings von G die größte Anzahl von Knoten besitzt. MAXIMALES MATCHING ist das Problem, ein maximales Matching in G zu bestimmen. □

Dies ist ein Problem mit vielen praktischen Anwendungen. Für paare Graphen (d.h. für Graphen (V, E), deren Knotenmenge so in zwei disjunkte Knotenmengen V_1 und V_2 zerlegt werden kann, daß für keine Kante (u, v) die Knoten u und v beide

in V_1 oder beide in V_2 liegen) nennt man es auch das *Heiratsproblem*. Gewisse Paare von Frauen und Männern können sich vorstellen zu heiraten, wobei z.B. eine Frau mit mehreren Männern ein solches Paar bilden kann. Ein maximales Matching liefert dann die maximal möglichen Eheschließungen.

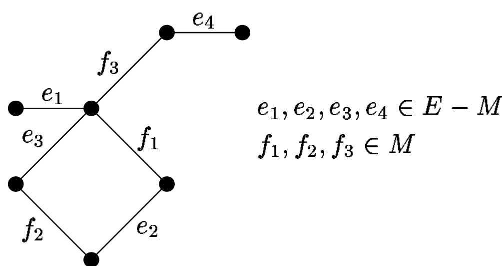
Um einen effizienten Algorithmus für MAXIMALES MATCHING in *schlichten* Graphen (Graphen ohne Schlingen und parallele Kanten) anzugeben, beginnen wir mit einer Definition und zwei Sätzen.

Definition 8.1.8 Es sei $G = (V, E)$ ein schlichter ungerichteter Graph umd $M \subset E$ ein Matching in G. Ein *M-alternierender* Weg $W = (e_1, \ldots, e_k)$ in G ist ein Weg mit der Eigenschaft

$$e_i \in M \iff e_{i+1} \in E - M \text{ für } i \in \{1, \ldots, k-1\}.$$

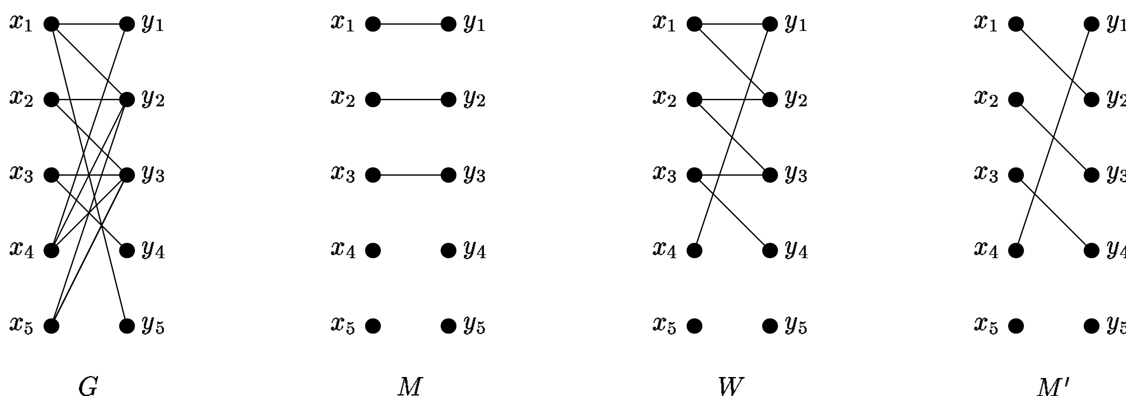
Ein M-alternierender Weg heißt *M-Ergänzungsweg*, wenn er mit v_1 beginnt und mit v_2 endet, wobei $v_1 \neq v_2$ gilt und diese Knoten nicht zu Kanten aus M gehören. □

Wir erkennen, daß bei einem M-Ergänzungsweg W die erste und letzte Kante zu $E - M$ gehören und alle „inneren" Knoten Ecken von Kanten von M sind. Ein solcher Knoten ist Ecke von genau zwei Kanten in W, denn anderenfalls gäbe es Kreuzungen wie in dem Bild



und M wäre kein Matching. Ein M-Ergänzungsweg enthält genau eine Kante von $E - M$ mehr als von M.

Beispiel 8.1.3 Wir betrachten einen Graphen G mit einem Matching M. M hat den M-Ergänzungsweg W. Nach dem folgenden Satz 8.1.1 erhalten wir daraus ein größeres Matching M', für das es keinen M'-Ergänzungsweg gibt.



Satz 8.1.1 Es sei G ein Graph, M ein Matching von G und W ein M-Ergänzungsweg. Dann existiert ein Matching M' mit $|M'| > |M|$, das durch $M' = (K(W) - M) \cup (M - K(W))$ definiert wird. Dabei ist $K(W)$ die Menge der Kanten von W.

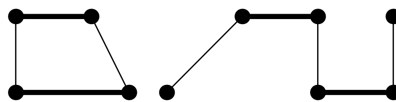
Beweis: Offenbar gilt $|M'| = |M|+1$. Wir stellen weiter fest, daß $M - K(W)$ und $K(W)$ (und damit erst recht auch $K(W) - M$) keine gemeinsamen Ecken haben. Nehmen wir nämlich an, daß ein Knoten v Ecke in beiden Mengen ist, dann ist v ein innerer Knoten von W und somit Ecke einer Kante von M in W. Gleichzeitig ist aber v auch Ecke einer Kante aus $M - K(W)$, also Ecke von zwei verschiedenen Kanten von M. Es folgt, daß M kein Matching ist, ein Widerspruch.

$M - K(W)$ ist offensichtlich ein Matching. Ebenso ist es $K(W) - M$, und zwar nach Definition 8.1.8 und den darauf folgenden Überlegungen. Da beide Mengen keine gemeinsamen Eckknoten haben, ist ihre Vereinigung M' ein Matching. □

Satz 8.1.2 Es sei $G = (V, E)$ ein Graph und M ein Matching von G, für das es keinen M-Ergänzungsweg gibt. Dann ist M maximal.

Beweis: Wir zeigen, daß für jedes Matching M, für das es ein größeres Matching $\hat{M}$ gibt, ein M-Ergänzungsweg existiert.

Es sei $\hat{G} = (V, \hat{E})$ der Graph mit $\hat{E} = (M - \hat{M}) \cup (\hat{M} - M)$. Die Kanten von $\hat{G}$ liegen also in genau einem der Matchings M oder $\hat{M}$. Jeder Knoten von $\hat{G}$ liegt somit auf höchstens zwei Kanten. Das bedeutet, daß $\hat{E}$ eine disjunkte Vereinigung von Zyklen und Wegen ist wie in dem folgenden Bild, in dem die Kanten aus M dicker als die von $\hat{M}$ gezeichnet sind.



Da sich in jedem Zyklus die Kanten von M und $\hat{M}$ abwechseln, ist in jedem Zyklus die Anzahl der Kanten aus M gleich derjenigen aus $\hat{M}$. Andererseits besitzt aber $\hat{G}$ mehr Kanten aus $\hat{M}$ als aus M. Daher gibt es einen maximalen Weg $W = (e_1, \ldots, e_k)$ in $\hat{G}$, auf dem mehr Kanten aus $\hat{M}$ als aus M liegen. Dies ist nur möglich, wenn e_1 und e_k zu $\hat{M}$ gehören. Da W maximal ist, gehören seine Endknoten nicht zu M. Folglich ist W ein M-Ergänzungsweg. □

Algorithmus 8.1.1
{Eingabe: ein ungerichteter schlichter Graph $G = (V, E)$,
Ausgabe: ein maximales Matching $M \subset E$}
$M := \emptyset$;
while ein M-Ergänzungsweg W existiert
    do $M := M'$ mit M' aus Satz 8.1.1
    od □

Satz 8.1.3 MAXIMALES MATCHING $\in PO$.

Beweis: Da der Graph G aus Algorithmus 8.1.1 nur endlich viele Kanten hat, folgt aus Satz 8.1.1, daß die **while**-Schleife nach endlich vielen Schritten abbricht. Nach Satz 8.1.2 liegt dann ein maximales Matching vor.

Die Zeitkomplexität hängt von dem Zeitbedarf zum Suchen des M-Ergänzungsweges ab. Ohne Beweis geben wir an, daß Algorithmus 8.1.3 eine Zeitkomplexität von $O(nk)$ hat, wobei n die Anzahl der Knoten und k die Anzahl der Kanten von G ist. Somit gehört MAXIMALES MATCHING zur Klasse PO. □

Ähnlich dem Finden eines maximalen Matching ist das MAXIMALFLUSSPROBLEM, daß wir im folgenden kurz und vereinfacht darstellen wollen. Eine ausführliche Beschreibung findet sich zum Beispiel in [5], Kapitel 6. Es sei ein gerichteter schlichter Graph $G = (V, E)$ mit zwei Knoten q (Quelle) und s (Senke) gegeben. Gesucht wird ein maximaler Fluß kanten-disjunkter Wege von q nach s, also eine Maximalzahl von Wegen von q nach s, die zwar gemeinsame Knoten, aber keine gemeinsamen Kanten haben dürfen. Die Lösung wird hier mit *flußvergrößernden Wegen* (ähnlich den M-Ergänzungswegen) berechnet. Ein solcher Weg für einen gegebenen Fluß H mit Kanten $\bar{E} \subset E$ ist eine Liste

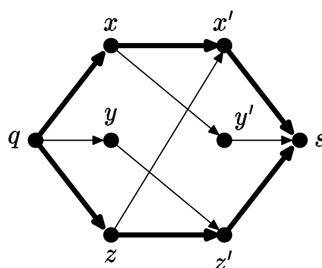
$$q = v_0, v_1, \ldots, v_{n-1}, v_n = s$$

von Knoten von G, so daß für jedes $i = 1, \ldots, n$

$$\text{entweder } (v_{i-1}, v_i) \in E - \bar{E} \text{ oder } (v_i, v_{i-1}) \in \bar{E}$$

gilt. Der Fluß H wird dadurch zu einem Fluß H' erweitert, daß alle Kanten $(v_{i-1}, v_i) \in E - \bar{E}$ des Weges zu H' hinzugenommen werden, während die Kanten $(v_i, v_{i-1}) \in \bar{E}$ des Weges aus dem Fluß entfernt werden.

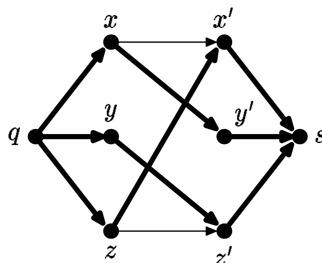
Beispiel 8.1.4 Es sei G der im folgenden gezeichnete Graph. Mit dickeren Linien ist ein Fluß H gekennzeichnet.



Ein flußvergrößernder Weg ist durch

$$q, y, z', z, x', x, y', s$$

gegeben. Er führt zu dem (maximalen) Fluß



Auch hier gibt es einen Algorithmus, der das Problem in der Zeit $O(nk)$ löst (n Anzahl der Knoten, k Anzahl der Kanten). Es ist also MAXIMALFLUSSPROBLEM $\in$ PO. □

Definition 8.1.9 Es sei (Σ, I, S, π, c) ein Minimierungspoblem. Das *zugrunde liegende Entscheidungsproblem* erhalten wir wie folgt: Gegeben seien $x \in I$ und $k \in \mathbb{N}$. Existiert eine korrekte Lösung $y \in S(x)$ mit $c(x,y) \leq k$? Die *zugrunde liegende Sprache* des Minimierungsproblems ist also durch

$$L = \{(x,k) \mid x \in I, k \in \mathbb{N}, \text{ es existiert korrekte Lösung } y \in S(x) \text{ mit } c(x,y) \leq k\}$$

gegeben.

Für ein Maximierungsproblem muß entsprechend $c(x,y) \geq k$ gelten. □

Beispiel 8.1.5 Wir betrachten das Maximierungsproblem MAXIMALE CLIQUE aus Beispiel 8.1.1(b). Für einen Graphen G und ein $k \in \mathbb{N}$ erhalten wir das Entscheidungsproblem CLIQUE aus Definition 7.3.1. □

Der nächste Satz rechtfertigt unser Interesse für die Klasse NPO.

Satz 8.1.4 Gehört ein Optimierungsproblem zu NPO, dann gehört die zugrunde liegende Sprache zu NP.

Beweis: Ohne Beschränkung der Allgemeinheit sei (Σ, I, S, π, c) ein Minimierungsproblem. Das Polynom p sei eine Schranke für die Länge der möglichen Lösungen. Der folgende nichtdeterministische Algorithmus entscheidet die zugrunde liegende Sprache:

begin {Eingabe: $x \in I$, $k \in \mathbb{N}$}
  if $x \notin I$ **then** stop ohne Akzeptierung;
  rate $y \in \Sigma^*$ mit $|y| \leq p(|x|)$;
  if $y \notin S(x)$ **then** stop ohne Akzeptierung;
  if $\pi(x,y) = 0$ **then** stop ohne Akzeptierung;
  if $c(x,y) \leq k$ **then** akzeptiere **else** keine Akzeptierung
end.

Die Bedingungen aus Definition 8.1.5 zeigen, daß dieser Algorithmus polynomiale Zeit benötigt. Er kann auch durch eine nichtdeterministische Turingmaschine mit polynomialem Zeitbedarf beschrieben werden. □

Wenn eine optimale Lösung eines Problems immer in polynomialer Zeit berechnet werden kann, dann ist es einsichtig, daß die zugrunde liegende Sprache zur Klasse P gehört. Dies zeigt

Satz 8.1.5 Gehört ein Optimierungsproblem zu PO, dann gehört die zugrunde liegende Sprache zu P.

Beweis: Ohne Beschränkung der Allgemeinheit sei (Σ, I, S, π, c) ein Minimierungsproblem. Nach Voraussetzung kann für jeden Fall $x \in I$ des Problems eine optimale Lösung y mit min $= c(x,y)$ in polynomialer Zeit berechnet werden. Für jeden Fall (x,k) der zugrunde liegenden Sprache können wir in polynomialer Zeit min mit k vergleichen. Folglich gehört die zugrunde liegende Sprache zu P. □

Satz 8.1.6 Falls $P \neq NP$ gilt, dann gehört jedes Optimierungsproblem aus *NPO* mit zugrunde liegender *NP*-vollständiger Sprache nicht zu *PO*

Beweis: Es sei (Σ, I, S, π, c) ein Optimierungsproblem in *NPO*, und L sei die zugrunde liegende Sprache. Wegen $P \neq NP$ und der *NP*-Vollständigkeit von L folgt nach Satz 7.2.1(a) $L \notin P$. Satz 8.1.5 zeigt, daß (Σ, I, S, π, c) nicht zu *PO* gehört. □

Wir kennen viele Probleme aus *NPO*, deren zugrunde liegende Sprachen *NP*-vollständig sind. Wir erhalten daher

Satz 8.1.7 Falls $P \neq NP$ gilt, ist auch $PO \neq NPO$. □

Man bezeichnet ein Problem auch als *NP-hart*, wenn seine Lösung in polynomialer Zeit zu einer Lösung von SAT in polynomialer Zeit führen würde. Das bedeutet, daß alle Probleme aus *NPO* mit zugrunde liegender *NP*-vollständiger Sprache in diesem Sinn als *NP*-hart bezeichnet werden können.

8.2 Approximation von Optimierungsproblemen

Falls es für ein Optimierungsproblem wahrscheinlich keinen effizienten Algorithmus gibt, da die zugrunde liegende Sprache *NP*-vollständig ist, dann können wir die Forderung nach Optimalität aufgeben und nach approximierenden Algorithmen suchen, die in polynomialer Zeit eine möglichst gute Näherungslösung liefern.

Wir haben in Definition 7.3.3 das Problem KNOTENÜBERDECKUNG betrachtet und angemerkt, daß es *NP*-vollständig ist. Wir geben das zugehörige Minimierungsproblem an.

Definition 8.2.1 Das Problem MINIMALE KNOTENÜBERDECKUNG ist wie folgt gegeben: Es sei $G = (V, E)$ ein ungerichteter Graph. Finde eine minimale Menge von Knoten $V' \subset V$ mit der Eigenschaft, daß $u \in V'$ oder $v \in V'$ für jede Kante $(u, v) \in E$ gilt. □

Für dieses Problem erhalten wir eine (ziemlich schlechte) Näherungslösung durch

Algorithmus 8.2.1
{Eingabe: ein ungerichteter Graph $G = (V, E)$
Ausgabe: eine Knotenüberdeckung C von G}
begin
    $C := \emptyset$;
    $E' := E$;
    while $E' \neq \emptyset$
        do wähle ein $(v_1, v_2) \in E'$;
            $E' := E' - \{(v_1, v_2)\}$;
            if $v_1 \notin C \wedge v_2 \notin C$ $\{(v_1, v_2)$ ist noch nicht überdeckt$\}$
            then $C := C \cup \{v_1, v_2\}$
        od
end. □

Satz 8.2.1 Algorithmus 8.2.1 konstruiert für einen Graphen G in linearer Zeit eine Knotenüberdeckung C. Ist die Größe einer minimalen Knotenüberdeckung von G durch $\min(G)$ gegeben, so gilt

$$|C| \leq 2 \cdot \min(G).$$

Beweis: In der **while** -Schleife wird jede Kante von E genau einmal aufgesucht. Dadurch ergibt sich der lineare Zeitbedarf des Algorithmus. Dabei werden bei jeder Kante (v_1, v_2), bei der noch nicht mindestens einer der Endknoten zu C gehört, sowohl v_1 als auch v_2 zu C hinzugefügt. Folglich ist C am Schluß eine Knotenüberdeckung von G, die einer disjunkten Vereinigung von $\frac{|C|}{2}$ Kanten entspricht. Jede Überdeckung muß nach Definition mindestens einen der Endknoten einer solchen Vereinigung enthalten. Wir schließen $\min(G) \geq \frac{|C|}{2}$. □

Zur Charakterisierung von Algorithmen, die eine Lösung konstruieren, die „genügend nahe“ einer optimalen Lösung kommen, dient die folgende Definition.

Definition 8.2.2 Es sei (Σ, I, S, π, c) ein Optimierungsproblem und $\varepsilon \in \mathbb{R}$, $\varepsilon > 0$. Ein *ε-approximierender Algorithmus* ist ein Algorithmus aus *FP*, der für jedes $x \in I$ ein $y \in S(x)$ mit $\pi(x, y) = 1$ berechnet, so daß der relative Fehler von $c(x, y)$ im Vergleich zum optimalen Wert $\mathrm{OPT}(x)$ kleiner gleich ε ist, d.h.

$$\frac{|c(x,y) - \mathrm{OPT}(x)|}{\mathrm{OPT}(x)} \leq \varepsilon. \quad \square$$

Beispiel 8.2.1 Für MINIMALE KNOTENÜBERDECKUNG ist Algorithmus 8.2.1, der für einen beliebigen Graphen G die Überdeckung C mit $|C| = c(G, C) \leq 2{\cdot}\mathrm{OPT}(G)$ liefert, ein 1-approximierender Algorithmus. Es gilt ja

$$\frac{c(G,C) - \mathrm{OPT}(G)}{\mathrm{OPT}(G)} \leq 1.$$

Dieser Algorithmus ist nicht ε-approximierend für $\varepsilon < 1$, denn für einen Graphen G mit zwei Knoten v_1, v_2 und einer Kante (v_1, v_2) liefert er $|C| = c(G, C) = 2$ und $\mathrm{OPT}(G) = 1$. □

Wir wollen ein Problem betrachten, bei dem ein Plan für die Verteilung unabhängiger Aufgaben aufgestellt werden soll. Bei diesem „Scheduling Independent Tasks“ sollen Aufgaben $W_1, \ldots, W_n$, $n \in \mathbb{N}$, mit jeweils gegebenem Zeitbedarf t_i, $i = 1, \ldots, n$, durch k Maschinen $M_1, \ldots, M_k$ bearbeitet werden. Wir nehmen an, daß jede Maschine unabhängig von den anderen jede Aufgabe erledigen kann. Wir wollen einen optimalen Plan (schedule) erstellen, das heißt, wir wollen jede Aufgabe W_i, $i \in \{1, \ldots, n\}$, einer Maschine $M_{S(i)}$, $S(i) \in \{1, \ldots, k\}$, so zuordnen, daß die Gesamtzeit T zur Erledigung aller Aufgaben minimal ist. Für die Maschine M_m, $m \in \{1, \ldots, k\}$, dauert die gesamte Arbeit $\sum_{i,S(i)=m} t_i$. Folglich erhalten wir

$$T = \max\{\sum_{i,S(i)=1} t_i, \ldots, \sum_{i,S(i)=k} t_i\}.$$

Jede Maschine arbeitet ununterbrochen, so lange Aufgaben vorliegen. Die Zahl der Maschinen k soll konstant sein. Wir erhalten

Definition 8.2.3 Es sei $k \in \mathbb{N}$ eine konstante Zahl. Das Problem SIT(k) ist wie folgt gegeben: Gegeben seien $n \in \mathbb{N}$ und $t_1, \ldots, t_n \in \mathbb{N}$. Gesucht ist eine Funktion $S : \{1, \ldots, n\} \to \{1, \ldots, k\}$, für die

$$T = \max\{ \sum_{i,S(i)=1} t_i, \ldots, \sum_{i,S(i)=k} t_i\}$$

minimal ist. □

SIT(k) ist ein Minimierungsproblem im Sinne von Definition 8.1.2. Dabei ordnet die Funtion c dieser Definition jeder Eingabe x, die hier durch Zahlen $t_1, \ldots, t_n$ gegeben ist, und jeder Funktion $S : \{1, \ldots, n\} \to \{1, \ldots, k\}$ den Wert $c(x, S) = T$ aus Definition 8.2.3 zu. $T_{\text{opt}} = \text{OPT}(x)$ wird dann mit einem Plan S_{opt} mit minimalem Wert T bestimmt.

Das zugrunde liegende Entscheidungsproblem fragt danach, ob ein Plan existiert, so daß für eine vorgegebene Schranke T' die Beziehung $T \leq T'$ gilt, ob also in vorgegebener Zeit alle Aufgaben erledigt werden können. Schon für $k \geq 2$ ist dieses Problem *NP*-vollständig. Man kann KNAPSACK $\leq$ SIT(k) beweisen.

Wir geben zunächst einen $\frac{1}{3}$-approximierenden Algorithmus für SIT(k) an, der zuerst die längeren und dann die kürzeren Aufgaben erledigt. Außerdem bevorzugt er die Maschinen, die bislang Arbeiten mit insgesamt geringstem Zeitbedarf zugeordnet bekommen haben. Wir verwenden die Hilfsvariable T_m, die die Zeit angibt, die die Maschine M_m bereits gearbeitet hat.

Algorithmus 8.2.2
{Eingabe: $t_1, \ldots, t_n \in \mathbb{N}$
Ausgabe: Funktion $S : \{1, \ldots, n\} \to \{1, \ldots, k\}$}
begin
  sortiere die Zahlen t_i, so daß danach ohne Beschränkung der Allgemeinheit
  $t_1 \geq t_2 \geq \ldots \geq t_n$;
  for $m = 1$ **to** k **do** $T_m := 0$ **od**;
  for $i = 1$ **to** n
    do wähle einen Index m mit $T_m = \min\{T_1, \ldots, T_k\}$;
      $S(i) := m$;
      $T_m := T_m + t_i$
    od
end. □

Satz 8.2.2 Algorithmus 8.2.2 liefert die Funktion S in polynomialer Zeit $O(n \log n)$ und ist ein $\frac{1}{3}$-approximierender Algorithmus.

Beweis: Mit der zweiten **for**-Schleife wird jeder Aufgabe W_i eine Maschine $M_{S(i)}$ zugeordnet.

Die Sortierung dauert $O(n \log n)$ Zeiteinheiten. Die erste **for**-Schleife hat, da k konstant ist, einen Zeitbedarf von $O(1)$. Jeder der n Durchgänge der zweiten **for**-Schleife benötigt wieder die Zeit $O(1)$. Insgesamt haben wir also den Zeitbedarf $O(n \log n)$. Das bedeutet, daß der Algorithmus eine Funktion der Klasse *FP* berechnet.

Wir verzichten auf den Beweis, daß dieser Algorithmus $\frac{1}{3}$-approximierend ist. Stattdessen werden wir anschließend zeigen, daß es für jedes $\varepsilon > 0$ einen ε-approximierenden Algorithmus für SIT(k) gibt. □

Wir wollen einen ε-approximierenden Algorithmus für SIT(k) angeben. Dabei wird eine einfache Idee verfolgt. Für eine geeignete Konstante $r \in \mathbb{N}$ wird die Verteilung der ersten r längsten Aufgaben mit dem jeweiligen Zeitbedarf $t_1, \ldots, t_r$ optimal geplant. Es werden einfach alle k^r möglichen Pläne probiert und der beste herausgesucht. Die übrigen Aufgaben werden wie in Algorithmus 8.2.2 verteilt. Nach dem nächsten Satz 8.2.3 erhalten wir bei jeder Wahl von

$$r \geq \frac{k}{\varepsilon}$$

einen ε-approximierenden Algorithmus. Der folgende Algorithmus wird also für ein festes r beschrieben.

Algorithmus 8.2.3
{Eingabe: $t_1 \ldots, t_n \in \mathbb{N}$
Ausgabe: Funktion $S : \{1, \ldots, n\} \to \{1, \ldots, k\}$}
begin
  sortiere die Zahlen t_i, so daß danach ohne Beschränkung der Allgemeinheit
  $t_1 \geq t_2 \geq \ldots \geq t_n$;
  finde für die Eingabe $t_1, \ldots, t_r$ den optimalen Plan $S_{\text{opt}} : \{1, \ldots, r\} \to \{1, \ldots, k\}$;
  for $m = 1$ **to** k **do** $T_m := \sum_{i, S_{\text{opt}}(i)=m} t_i$ **od**;
  for $i = r + 1$ **to** n
    do wähle einen Index m mit $T_m = \min\{T_1, \ldots, T_k\}$;
      $S(i) := m$;
      $T_m := T_m + t_i$
    od
end. □

Satz 8.2.3 Es sei $\varepsilon > 0$ und $r \geq \frac{k}{\varepsilon}$. Algorithmus 8.2.3 liefert die Funktion S in polynomialer Zeit $O(n \log n)$ und ist ein ε-approximierender Algorithmus für SIT(k).

Beweis: Der Algorithmus liefert offensichtlich die angegebene Ausgabe. Wir betrachten seinen Zeitbedarf. Das Sortieren erfolgt wieder in der Zeit $O(n \log n)$. Der optimale Plan S_{opt} kann gefunden werden, indem alle k^r möglichen Pläne durchsucht werden. Für jeden solchen Plan S berechnen wir in der Zeit $O(k \cdot r)$ den Wert

$$T(S) = \max\{\sum_{i \in \{1,\ldots,r\}, S(i)=1} t_i, \ldots, \sum_{i \in \{1,\ldots,r\}, S(i)=k} t_i\},$$

also die Bearbeitungszeit, die die r Aufgaben unter dem Plan S benötigen. Anschließend finden wir in der Zeit $O(k^r)$ das Minimum aller Werte $T(S)$ und damit den optimalen Plan S_{opt} der ersten r Aufgaben. Das Finden dieses optimalen Plans erfordert also $O(krk^r) = O(rk^{r+1})$ Schritte. Die restlichen Laufzeitabschätzungen werden wie im Beweis von Satz 8.2.2 durchgeführt. Insgesamt hat der Algorithmus somit die

Zeitkomplexität $O(n \log n + rk^{r+1})$. Es sind aber r und k Konstanten, so daß der Algorithmus die Zeit $O(n \log n)$ benötigt. Man beachte, daß, falls ε gegen 0 strebt, der Wert von r entsprechend größer gewählt werden muß und in diesem Sinn wegen des Anteils $O(rk^{r+1})$ die Zeitkomplexität exponentiell mit r wächst.

Wir beweisen, daß der Algorithmus ε-approximierend ist. Es ist also

$$\frac{T - T_{\text{opt}}}{T_{\text{opt}}} \leq \varepsilon$$

zu zeigen, wobei T die Zeit ist, in der die Aufgaben bei dem vom Algorithmus gefundenen Plan S erledigt werden. Da jede Maschine ununterbrochen arbeitet, so lange Aufgaben vorliegen, muß es mindestens eine Aufgabe W_i, $i \in \{1, \ldots, n\}$, geben, die genau zur Zeit T beendet wird.

Falls $i \leq r$ gilt, ist W_i schon zu Beginn durch den optimalen Plan S_{opt} der Teilmenge $\{1, \ldots, r\}$ einer Maschine zugeteilt worden. Diese Maschine hat dann wegen des Ablaufs der Gesamtzeit T überhaupt keine Aufgaben W_j mit $j \geq r+1$ bearbeitet. Da in diesem Fall die Zeit T auf der Teilmenge optimal ist und auf der größeren Menge von Aufgaben die Bearbeitungszeit ebenfalls T ist, muß $T = T_{\text{opt}}$ gelten, so daß die obige Ungleichung erfüllt ist.

Wir nehmen daher im folgenden an, daß $i > r$ gilt. Die Arbeit an der Aufgabe W_i beginnt zur Zeit $T - t_i$. Daraus folgt, daß alle Maschinen im ganzen Zeitraum $[0, T - t_i)$ ständig beschäftigt sind. Würde nämlich eine Maschine zu einer Zeit $T_0 < T - t_i$ aufhören, so hätte der Algorithmus wegen der Minimumsbildung in der zweiten **for**-Schleife die Aufgabe W_i auf dieser Maschine bereits zum Zeitpunkt T_0 beginnend ausführen lassen. Alle k immer beschäftigten Maschinen leisten im Zeitraum $[0, T - t_i)$ die gesamte Arbeit von $k(T - t_i)$ Zeiteinheiten. Aber in diesem Zeitraum werden nur Aufgaben W_j mit $j < i$ bearbeitet, da die Arbeit an W_i erst zur Zeit $T - t_i$ beginnt. Im Zeitraum $[0, T - t_i)$ werden also höchstens die Aufgaben $W_1, \ldots, W_{i-1}$ behandelt. Die geleistete Arbeit von $k(T - t_i)$ Zeiteinheiten ist also höchstens so groß wie die gesamte Zeit $t_1 + \ldots + t_{i-1}$, die die Aufgaben W_1 bis W_{i-1} benötigen. Es gilt also

$$k(T - t_i) \leq t_1 + \ldots + t_{i-1}. \tag{8.1}$$

Daraus ergibt sich die Ungleichung

$$T \leq \frac{t_1 + \ldots + t_{i-1}}{k} + t_i. \tag{8.2}$$

Auf der anderen Seite verbrauchen alle n Aufgaben zusammen insgesamt $t_1 + \ldots + t_n$ Zeiteinheiten. Der optimale Plan mit Zeitbedarf T_{opt} benötigt auf allen k Maschinen zusammen höchstens $k \cdot T_{\text{opt}}$ Zeiteinheiten. Es folgt

$$k \cdot T_{\text{opt}} \geq t_1 + \ldots + t_n \tag{8.3}$$

und damit

$$T_{\text{opt}} \geq \frac{t_1 + \ldots + t_n}{k} \geq \frac{t_1 + \ldots + t_{i-1}}{k}. \tag{8.4}$$

Die Gleichungen 8.2 und 8.4 liefern

$$T - T_{\text{opt}} \leq \frac{t_1 + \ldots t_{i-1}}{k} + t_i - \frac{t_1 + \ldots + t_{i-1}}{k} = t_i. \tag{8.5}$$

Aufgrund der Sortierung zu Beginn des Algorithmus und $i > r$ ist $t_i \leq t_r$, so daß sich aus 8.5 die Ungleichung

$$T - T_{\text{opt}} \leq t_r \tag{8.6}$$

ergibt. Unter Benutzung von 8.4 und der Ungleichungskette $t_1 \geq t_2 \geq \ldots \geq t_n$ erhalten wir

$$T_{\text{opt}} \geq \frac{t_1 + \ldots + t_n}{k} \geq \frac{t_1 + \ldots + t_r}{k} \geq \frac{rt_r}{k}. \tag{8.7}$$

Zusammen mit 8.6 folgt

$$\frac{T - T_{\text{opt}}}{T_{\text{opt}}} \leq \frac{t_r}{\frac{rt_r}{k}} = \frac{k}{r}. \tag{8.8}$$

Wegen $r \geq \frac{k}{\varepsilon}$ ist $\frac{k}{r} \leq \varepsilon$. Damit ist 8.8 die gewünschte Ungleichung. □

Es existieren also Probleme mit zugrunde liegendem *NP*-vollständigen Entscheidungsproblem, die für jedes $\varepsilon > 0$ einen ε-approximierenden Algorithmus besitzen. Falls $P \neq NP$ gilt, gibt es im Gegensatz dazu Optimierungsprobleme mit zugrunde liegendem *NP*-vollständigen Entscheidungsproblem, die für kein ε einen ε-approximierenden Algorithmus haben. Dies zeigt

Satz 8.2.4 Falls $P \neq NP$ gilt, existiert für kein $\varepsilon > 0$ ein ε-approximierender Algorithmus für MINIMALES TSP.

Beweis: Wir wissen, daß HAMILTONSCHER KREIS (Definition 7.3.2) ein *NP*-vollständiges Problem ist. Falls für ein $\varepsilon > 0$ ein ε-approximierender Algorithmus A_ε für MINIMALES TSP existiert, dann leiten wir den Widerspruch HAMILTONSCHER KREIS $\in P$ ab. Wir geben einen Algorithmus A an, der in polynomialer Zeit prüft, ob ein vorgelegter ungerichteter Graph einen Hamiltonschen Kreis besitzt.

Es sei $G = (V, E)$ mit $|V| = n$. Der Algorithmus A wendet den Algorithmus A_ε auf den Fall von MINIMALES TSP an, der durch n Städte und die Kostenmatrix $(d(i,j))$, $i, j = 1, \ldots, n$, mit

$$d(i,j) = \begin{cases} 1, & \text{falls } (i,j) \in E \\ 2 + \lceil \varepsilon n \rceil & \text{sonst} \end{cases}$$

gegeben ist. Wir bezeichnen mit c die Kosten der Rundfahrt, die der approximierende Algorithmus A_ε findet. Der Algorithmus A liefert genau dann die Antwort „ja“, wenn $c \leq (1+\varepsilon)n$ gilt. Zum Beweis müssen wir nur die Äquivalenz

$$G \text{ besitzt einen Hamiltonschen Kreis} \iff c \leq (1+\varepsilon)n$$

zeigen.

Besitzt der Graph einen Hamiltonschen Kreis, so liefert dieser eine Rundfahrt über n Kanten von G und mit optimalen Kosten $c_{\text{opt}} = n$. Da A_ε ε-approximierend ist, gilt mit den Kosten c der durch A_ε gefundenen Rundfahrt $\frac{c-n}{n} \leq \varepsilon$, also $c \leq (1+\varepsilon)n$.

Umgekehrt gelte für die gefundene Rundfahrt $c \leq (1+\varepsilon)n$. Wir beweisen, daß die Rundfahrt nur Kanten aus G benutzt und daher einen Hamiltonschen Kreis in G bildet. Dazu reicht der Nachweis der Ungleichung $c < 2 + \lceil \varepsilon n \rceil + (n-1)$, denn jede Rundfahrt,

die mindestens einmal die Kanten von G verläßt, hat Kosten $\geq 2 + \lceil \varepsilon n \rceil + (n-1)$. Wegen $\varepsilon n < \lceil \varepsilon n \rceil + 1$ folgt

$$c \leq (1+\varepsilon)n < n + \lceil \varepsilon n \rceil + 1 = 2 + \lceil \varepsilon n \rceil + (n-1).$$

Abschließend bestimmen wir die Zeitkomplexität des Algorithmus A. Wir beachten zunächst, daß die Zeitschranke des Algorithmus A_ε durch ein Polynom p bestimmt ist. Aus G konstruiert A das neue Problem von TSP in $O(n^2)$ Schritten. Darauf wird in der Zeit $O(p(n^2))$ der Algorithmus A_ε angewendet. Insgesamt benötigt A somit $O(n^2+p(n^2))$ Schritte. Wir schließen, daß HAMILTONSCHER KREIS im Widerspruch zu seiner *NP*-Vollständigkeit in polynomialer Zeit entschieden werden kann. □

Für Optimierungsaufgaben, für die wir keinen effizienten Algorithmus kennen, können wir versuchen, approximierende Algorithmen zu finden. Es gibt dabei sogar Aufgaben, die für jedes $\varepsilon > 0$ einen ε-approximierenden Algorithmus besitzen (Satz 8.2.3). Leider gibt es wichtige Optimierungsaufgaben, wie es gerade Satz 8.2.4 gezeigt hat, die keinen ε-approximierenden Algorithmus gestatten.

9 Raumkomplexität

Algorithmen werden als effizient betrachtet, wenn sie wenig Zeit zu ihrer Ausführung benötigen. Ein anderer wichtiger Gesichtspunkt ist der Speicherbedarf eines Algorithmus. Wenn wir uns ähnlich wie bei der Zeitkomplexität nur dafür interessieren, ob ein Algorithmus auf einer vorgelegten Eingabe der Länge n mit polynomial beschränktem Speicherplatz bezüglich n auskommt, dann können wir auch hier das Modell der Turingmaschinen verwenden.

Definition 9.1 Es sei $T = (Z, X, \delta, z_0, F)$ eine Turingmaschine. Dann heißt

$$S_T(n) = \max_{w \in X^*, |w|=n}\{k \mid k \text{ ist die Anzahl der bei einer Rechnung an } w \text{ besuchten Felder}\}$$

die *Raumkomplexität* von T. Falls ein w mit $|w| = n$ existiert, bei der eine Rechnung von T an w unendlich viele Felder besucht, wird $S_T(n) = \infty$ gesetzt. □

Definition 9.2 Es ist

$$\text{PSPACE} = \{L \mid \text{es existiert eine deterministische Turingmaschine } T \text{ mit } L = L(T) \text{ und } S_T(n) \leq p_T(n) \text{ für ein Polynom } p_T \text{ und alle } n\}$$

die *Komplexitätsklasse der mit polynomialer Raumkomplexität entscheidbaren Sprachen.* □

Die zuvor eingeführten Komplexitätsklassen *P* und *NP* werden oft auch als PTIME und NPTIME bezeichnet, um sie von PSPACE zu unterscheiden. Offensichtlich gilt

$$\text{PTIME} \subset \text{PSPACE},$$

denn eine Turingmaschine kann in einem Schritt auch nur ein Feld des Bandes besuchen. Daraus ergibt sich unmittelbar, daß Probleme wie zum Beispiel 2-FÄRBUNG und SAT(2) in PSPACE liegen. Das Problem, ob ein gegebener regulärer Ausdruck universell ist (siehe Definition 6.2.5), gehört nicht zur Klasse PSPACE.

In Definition 9.2 werden nur deterministische Turingmaschinen betrachtet. Analog dem Vorgehen bei der Definition der Zeitkomplexitätsklassen könnte man hier auch nichtdeterministische Turingmaschinen zulassen, was zu einer Komplexitätsklasse NPSPACE führen würde. Dies ist jedoch nicht nötig, wie durch den folgenden Satz von *Savitch* gezeigt wird.

Satz 9.1 Jede nichtdeterministische Turingmaschine mit polynomialer Raumkomplexität kann durch eine deterministische Turingmaschine mit polynomialer Raumkomplexität simuliert werden.

Beweis: Es sei $T = (Z, X, \delta, z_0, F)$ eine akzeptierende nichtdeterministische Turingmaschine, deren Raumkomplexität durch ein Polynom p beschränkt ist. Wir konstruieren eine deterministische Turingmaschine $\hat{T}$ mit einer Raumkomplexität $O((p(n))^2)$ für jedes Eingabewort $w \in \bar{X}^*$ der Länge n.

Eine Konfiguration $K = (i, \Gamma(\beta), z)$ der Turingmaschine T gemäß Definition 2.1.4 kann, falls die Rechnung von T an einem Wort der Länge n durchgeführt wird, offenbar durch ein Tripel

$$(i, x_{-p(n)+1} \dots x_{-1} x_0 x_1 \dots x_{p(n)-1}, z)$$

dargestellt werden. Dabei gibt i die Kopfposition an, z ist der aktuelle Zustand, und es gilt $x_i \in \bar{X}$ für $i = -p(n)+1, \dots, p(n)-1$. Der Raumbedarf für eine solche Konfiguration ist, da die Symbole von X und Z direkt auf das Band geschrieben werden können und i auf maximal $\log p(n)$ Felder geschrieben werden kann, von der Ordnung $O(p(n))$. Zu Beginn einer Rechnung an einem Wort $w = a_1 \dots a_n$, $a_i \in X$, $i = 1, \dots, n$, erhalten wir die Anfangskonfiguration $K(w)$ mit $i = 0$, $z = z_0$ und $x_0 = a_1, \dots, x_{n-1} = a_n$, während die übrigen Felder mit $x_j = b$ beschrieben sind. Insgesamt gibt es für $|X| = m$ und $|Z| = r$ insgesamt

$$c(n) = (2p(n) - 1) \cdot m^{2p(n)-1} \cdot r < (m+1)^{2p(n)-1} \cdot m^{2p(n)-1} \cdot r < 2^{Cp(n)}$$

verschiedene Konfigurationen, wobei C eine geeignete Konstante ist.

Für ein Wort $w \in L$ mit $|w| = n$ wählen wir eine kürzeste akzeptierende Rechnung. In einer solchen kürzesten Rechnung können sich keine Konfigurationen wiederholen. Folglich existiert eine akzeptierende Rechnung von höchstens $2^{Cp(n)}$ Schritten. Es gilt also $w \in L$ genau dann, wenn die Anfangskonfiguration $K(w)$ in höchstens $2^{Cp(n)}$ Schritten in eine akzeptierende Endkonfiguration $K_a = (j, y_{-p(n)+1} \dots y_{p(n)-1}, z)$ überführt wird. Wir erinnern daran, daß dann $y_j\ z\ s\ z'$ mit einem $z' \in F$ eine Zeile der Turingtafel von T sein muß.

Allgemein schreiben wir für zwei beliebige Konfigurationen K und K', die in Rechnungen an Wörtern der Länge n vorkommen,

$$\text{TEST}(K, K', i) = 1,$$

falls in höchstens 2^i Schritten aus der Konfiguration K die Konfiguration K' gewonnen werden kann. In anderen Fällen wird der Wert auf 0 gesetzt. $\text{TEST}(K, K', 0) = 1$ bedeutet speziell, daß entweder $K = K'$ gilt oder K' eine Folgekonfiguration von K ist. Man beachte, daß bei einem Eingabewort w der Länge n die Anfangskonfiguration $K(w)$ und alle Folgekonfigurationen dieselbe Länge $2p(n) - 1$ der Darstellung des Bandes haben.

Aufgrund der vorhergehenden Überlegungen stellen wir fest, daß T das Wort w genau dann akzeptiert, wenn $\text{TEST}(K(w), K_a, Cp(n)) = 1$ gilt für eine beliebige akzeptierende Endkonfiguration K_a. Wir müssen also eine deterministische Turingmaschine $\hat{T}$ konstruieren, die für alle möglichen Endkonfigurationen K_a von T den Wert $\text{TEST}(K(w), K_a, Cp(n))$ berechnet, bis sich ggf. der Wert 1 ergibt. Insgesamt muß $\hat{T}$ dabei nach dem folgenden Algorithmus vorgehen:

```
begin
    lese w;
    bestimme n = |w|;
    schreibe Anfangskonfiguration K(w) auf das Band;
    for alle (≤ c(n)) Konfigurationen K'
        do if  K' Endkonfiguration
            then if  TEST(K(w), K', Cp(n)) = 1 then akzeptiere w und stop
        od
end.
```

Da alle Konfigurationen der Reihe nach systematisch auf dem Band von $\hat{T}$ erzeugt werden können, erfordert diese Erzeugung $O(p(n))$ Platz. Für jede Konfiguration kann die oben angegebene Bedingung für eine Endkonfiguration leicht überprüft werden. Es fehlt noch die Implementierung und die Überlegungen zum Raumbedarf von TEST.

```
procedure TEST(K1, K2, i);
      if  i = 0 und (K1 = K2 oder K2 ist Folgekonfiguration von K1)
      then return 1;
      if  i ≥ 1 then
              for alle (≤ c(n)) Konfigurationen K̄
                  do if  TEST(K1, K̄, i − 1) = 1 und TEST(K̄, K2, i − 1) = 1
                      then return 1
                  od;
      return 0
end TEST.
```

Es ist klar, daß TEST das gewünschte leistet. Für $i = 0$ ist dies unmittelbar klar. Für $i > 0$ gilt $\mathrm{TEST}(K_1, K_2, i) = 1$ genau dann, wenn eine Konfiguration $\bar{K}$ existiert mit $\mathrm{TEST}(K_1, \bar{K}, i-1) = 1$ und $\mathrm{TEST}(\bar{K}, K_2, i-1) = 1$. Das bedeutet ja, daß der Übergang von K_1 zu K_2 in höchstens $2^i = 2{\cdot}2^{i-1}$ Schritten äquivalent ist zum Übergang von K_1 zu $\bar{K}$ in höchstens 2^{i-1} Schritten gefolgt von höchstens 2^{i-1} Schritten von $\bar{K}$ zu K_2.

Jeder Aufruf der Prozedur TEST erfordert vor Beginn eines rekursiven Aufrufs Raum für die Konfigurationen K_1, K_2, den Wert $i \leq Cp(n)$ und die Variable $\bar{K}$. Für $\bar{K}$ werden der Reihe nach auf dem Band von $\hat{T}$ in einem festen Bereich die entsprechenen Konfigurationen erzeugt. Der Raumbedarf ist also vor Einstieg in die Rekursion $O(p(n))$. Da die Rekursionstiefe durch $Cp(n)$ beschränkt ist, ist insgesamt Raum der Größe $O((p(n))^2)$ ausreichend. Die vorstehenden Überlegungen zeigen uns, daß man (mit einiger Mühe im Detail) eine deterministische Turingmaschine $\hat{T}$ mit den gewünschten Eigenschaften konstruieren kann. □

Eine unmittelbare Folgerung ist

Satz 9.2 Es gilt PSPACE=NPSPACE. □

Satz 9.3 Es gilt $NP = \mathrm{NPTIME} \subset \mathrm{PSPACE}$.

Beweis: Eine nichtdeterministische Turingmaschine mit polynomialem Zeitbedarf braucht zum Erkennen eines Wortes polynomialen Raum. Aus Satz 9.1 folgt dann das Ergebnis. □

Die Klasse PSPACE enthält also alle Probleme, für die es effizient entscheidbar ist, ob eine angebotene Lösung wirklich das Problem löst. Unter anderen gehören TSP oder 3-FÄRBBARKEIT zu PSPACE. Es ist nicht bekannt, ob NPTIME=PSPACE gilt. Wir geben im folgenden ein Beispiel eines Problems aus PSPACE an, für das die Zugehörigkeit zu NPTIME unbekannt ist.

In Definition 6.2.3 haben wir logische Ausdrücke in konjunktiver Normalform betrachtet, wobei außer Variablen die Verknüpfungen $\wedge$, $\vee$ und $\neg$ benutzt wurden. Zusätzlich sollen jetzt noch die Quantoren $\forall$ („für alle") und $\exists$ („es existiert") verwendet werden.

Definition 9.3 *Quantifizierte Boolesche Formeln* (QBF) über Variablen x_i, $i = 1, 2, \ldots$, werden zusammen wie die Begriffe *freie Variable* und *gebundene Variable* wie folgt definiert:

(a) Jede Variable x_i ist eine quantifizierte Boolesche Formel. Das Vorkommen von x_i in diesem Ausdruck ist frei.
(b) Die Konstanten 0 (falsch) und 1 (wahr) sind quantifizierte Boolesche Formeln.
(c) Falls E_1 und E_2 quantifizierte Boolesche Formeln sind, dann sind es auch $E_1 \vee E_2$, $E_1 \wedge E_2$ und $\neg E_1$. Ein Vorkommen einer Variable in diesen Ausdrücken ist frei bzw. gebunden, wenn es auch in dem Ausdruck E_1 oder E_2, aus dem es stammt, ebenfalls frei bzw. gebunden ist.
(d) Falls E eine quantifizierte Boolesche Formel ist, sind es auch $(\exists x)E$ und $(\forall x)E$ für eine beliebige Variable x. Der Wirkungsbereich von $(\exists x)E$ und $(\forall x)E$ sind alle freien Vorkommen von x in E. Freie Vorkommen von x in E sind gebunden in $(\exists x)E$ und $(\forall x)E$. Alle anderen Vorkommen von Variablen in $(\exists x)E$ und $(\forall x)E$ sind frei oder gebunden, je nachdem, ob sie frei oder gebunden in E sind.

Die Menge aller quantifizierten Booleschen Formeln ist die kleinste Menge, für die (a) bis (d) erfüllt ist.

Eine quantifizierte Boolesche Formel heißt *geschlossen*, wenn sie keine freien Variablen besitzt. □

Definition 9.4 Der *Wert* VAL(E) einer geschlossenen quantifizierten Booleschen Formel E wird wie folgt definiert:

(a) VAL(1)= 1 und VAL(0)= 0.
(b) $\text{VAL}(E_1 \vee E_2) = \text{VAL}(E_1) \vee \text{VAL}(E_2)$ und $\text{VAL}(E_1 \wedge E_2) = \text{VAL}(E_1) \wedge \text{VAL}(E_2)$.
(c) $\text{VAL}(\neg E) = \neg\text{VAL}(E)$.
(d) $\text{VAL}((\forall x)E) = \text{VAL}(E(1/x)) \wedge \text{VAL}(E(0/x))$.
(e) $\text{VAL}((\exists x)E) = \text{VAL}(E(1/x)) \vee \text{VAL}(E(0/x))$.

Dabei ist $E(1/x)$ (bzw. $E(0/x)$) die quantifizierte Boolesche Formel, die wir aus E durch Ersetzung aller freien Vorkommen der Variablen x durch 1 (bzw. 0) erhalten. □

Beispiel 9.1 Wir betrachten die quantifizierte Boolesche Formel

$$E = \forall x(\forall x(\exists y(x \vee y)) \wedge \neg x).$$

Der Wirkungsbereich des inneren $\forall x$ ist das erste Vorkommen von x, der des äußeren $\forall x$ das zweite Vorkommen. In E sind alle Variablen gebunden, E ist also geschlossen. Nach Definition 9.4 gilt

$$\text{VAL}(E) = \text{VAL}(\forall x(\exists y(x \vee y) \wedge \neg 0) \wedge \text{VAL}(\forall x(\exists y(x \vee y) \wedge \neg 1) = 0.$$

Es ist hier nicht nötig, für $E' = \forall x(\exists y(x \vee y)$ den Wert $\text{VAL}(E')$ zu berechnen. Der Vollständigkeit halber geben wir ihn jedoch an. Es ist

$$\text{VAL}(E') = \text{VAL}(\exists y(1 \vee y)) \wedge \text{VAL}(\exists y(0 \vee y)) = 1,$$

da in beiden Teilausdrücken $y = 1$ gewählt werden kann. □

Definition 9.5 Das QBF-Problem ist wie folgt gegeben: Eine geschlossene quantifizierte Boolesche Formel ist die Eingabe. Es wird gefragt, ob sie den Wert 1 hat. □

Wir bemerken, daß ein Ausdruck E in konjunktiver Normalform über den Variablen $x_1, \ldots, x_k$ genau dann erfüllbar ist, wenn die quantifizierte Boolesche Formel $\exists x_1 \exists x_2 \ldots \exists x_k(E)$ wahr ist. Folglich ist SAT ein Spezialfall von QBF. Das bedeutet, daß QBF *NP*-hart ist.

Satz 9.4 Es gilt QBF $\in$ PSPACE.

Beweis: Wir müssen eine deterministische Turingmaschine konstruieren, die für eine vorgegebene geschlossene quantifizierte Boolesche Formel E mit polynomialem Platz entscheidet, ob $\text{VAL}(E) = 1$ gilt. Die Funktion VAL ist rekursiv in Definition 9.4 definiert worden. Für jeden Quantor und jeden Operator kommt es zu einem Rekursionsaufruf, bis schließlich mit Hilfe von (d) und (e) nach und nach alle Variablen durch 0 oder 1 ersetzt werden. Die Tiefe der Rekursion ist also durch die Anzahl aller Quantoren und aller Operatoren beschränkt und somit auch durch die Länge n des gegebenen Ausdrucks. Die Funktion VAL kann durch eine Turingmaschine berechnet werden. Dabei ist ihr Raumbedarf vor Einstieg in die Rekursion offenbar $O(n)$. Da die Rekursion höchstens die Tiefe n hat, ist der Gesamtraumbedarf $O(n^2)$. Es folgt QBF $\in$ PSPACE. □

QBF liegt wahrscheinlich nicht in *NP*. QBF ist nämlich PSPACE-vollständig. Das bedeutet, daß es für jedes Problem aus PSPACE eine Reduktion in polynomialer Zeit auf QBF gibt. Aus QBF $\in$ *NP* würde *NP* = PSPACE folgen.

10 Parallele Algorithmen

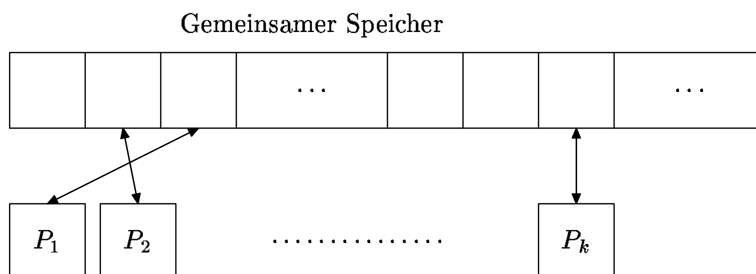
Als Folge der Fortschritte auf dem Gebiet des Mikroprozessorentwurfs ist es in den letzten Jahren möglich geworden, massiv parallele Rechner zu bauen, das heißt Rechner mit einigen zehntausend parallel arbeitenden Prozessoren. Gleichzeitig damit hat eine neue Theorie der parallelen Berechnungen erste Gestalt angenommen. Dabei wurden Modelle für parallele Rechner eingeführt und Probleme charakterisiert, die in hohem Maße dafür geeignet sind, mit ihrer Hilfe gelöst zu werden, und zwar viel effizienter, als es mit klassischen sequentiellen Rechnern der Fall ist.

Im Gegensatz zu sequentiellen Rechnern, wo die verschiedenen von uns betrachteten Modelle (Turingmaschinen, Registermaschinen) prinzipiell dieselben Fähigkeiten haben, gibt es mehrere grundsätzlich unterschiedliche Modelle für parallele Berechnungen. Einige gehen davon aus, daß jeder Prozessor einen lokalen Speicher hat (local memory), in anderen gibt es außerdem einen gemeinsamen Speicher (shared memory), auf den alle Prozessoren Zugriff haben. Diese Modelle lassen sich weiter darin unterscheiden, ob jeder Prozessor unabhängig von den anderen arbeitet (asynchrones Modell) oder ob es eine zentrale Einheit gibt, die die Arbeit der Prozessoren so steuert, daß in jedem Takt alle aktiven Prozessoren genau eine Instruktion durchführen (synchrones Modell).

Wir werden vor allem parallele Registermaschinen (PRAM) betrachten. Daneben werden in Abschnitt 10.6 noch verschiedene Netzwerkmodelle angesprochen.

10.1 Das PRAM-Modell

Das PRAM-Modell (sprich P-RAM) ist die parallele Erweiterung des Registermaschinenmodells aus Definition 2.8.1. Es ist ein synchrones Modell. Die gemeinsame Taktung wird durch eine zentrale Uhr gesteuert. Eine PRAM besteht aus mehreren unabhängigen sequentiellen Prozessoren P_i, jeweils mit einem Index i versehen, von denen jeder seinen eigenen lokalen Speicher hat. Die Kommunikation der Prozessoren erfolgt über den gemeinsamen Speicher. In jeder Zeiteinheit kann jeder Prozessor ein Register des lokalen oder gemeinsamen Speichers lesen, eine einzelne RAM-Instruktion ausführen und in ein Register des lokalen oder gemeinsamen Speichers schreiben. Auf eine formalere Definition wollen wir hier verzichten.



Es stellt sich die Frage, wie sich eine PRAM verhält, wenn mehrere ihrer Prozessoren dasselbe Register des gemeinsamen Speichers lesen oder beschreiben wollen. Es gibt mehrere Variationen des Modells, die diese Zugriffskonflikte auf unterschiedliche

Weisen lösen. Eine

EREW PRAM (*Exclusive-Read Exclusive-Write*)

ist eine PRAM, für die ein gleichzeitiger Zugriff verschiedener Prozessoren zu irgendeinem Register verboten ist, und zwar sowohl für das Lesen als auch für das Schreiben. Ein Programm für eine solche PRAM muß also so geschrieben sein, daß diese Bedingung erfüllt ist. Falls mehrere Prozessoren gleichzeitig aus demselben Register lesen, aber nicht schreiben dürfen, sprechen wir von einer

CREW PRAM (*Concurrent-Read Exclusive-Write*).

Wenn schließlich auch das gleichzeitige Schreiben erlaubt ist, erhalten wir eine

CRCW PRAM (*Concurrent-Read Concurrent-Write*).

In diesem Fall müssen wir jedoch festlegen, wie Schreibkonflikte gelöst werden. Wir betrachten zwei Varianten. Wir sprechen von einer

COMMON CRCW PRAM,

wenn alle Prozessoren, die in dasselbe Register schreiben, denselben Wert schreiben. Dies muß durch das entsprechende Programm der PRAM gewährleistet werden. Bei einer

PRIORITY CRCW PRAM

gibt es eine lineare Ordnung der Indizes der Prozessoren, und der Prozessor mit dem kleinsten Index schreibt in einem Konfliktfall in das Register. Wir werden in Abschnitt 10.3 zeigen, daß die Berechnungsfähigkeiten dieser verschiedenen Modelle sich nicht sehr unterscheiden.

10.2 PRAM-Algorithmen

Wir beginnen mit einem einfachen Problem, für das wir verschiedene PRAM-Algorithmen angeben werden.

Definition 10.2.1 Das Problem MAX ist wie folgt gegeben: Es sei $A = (A[0], \ldots, A[n-1])$, $A[i] \in \mathbb{N}$, $n \in \mathbb{N}$. Gesucht ist $\max\{A[i] \mid i = 0, \ldots, n\}$.

Wir geben zunächst zwei sequentielle Algorithmen für MAX an.

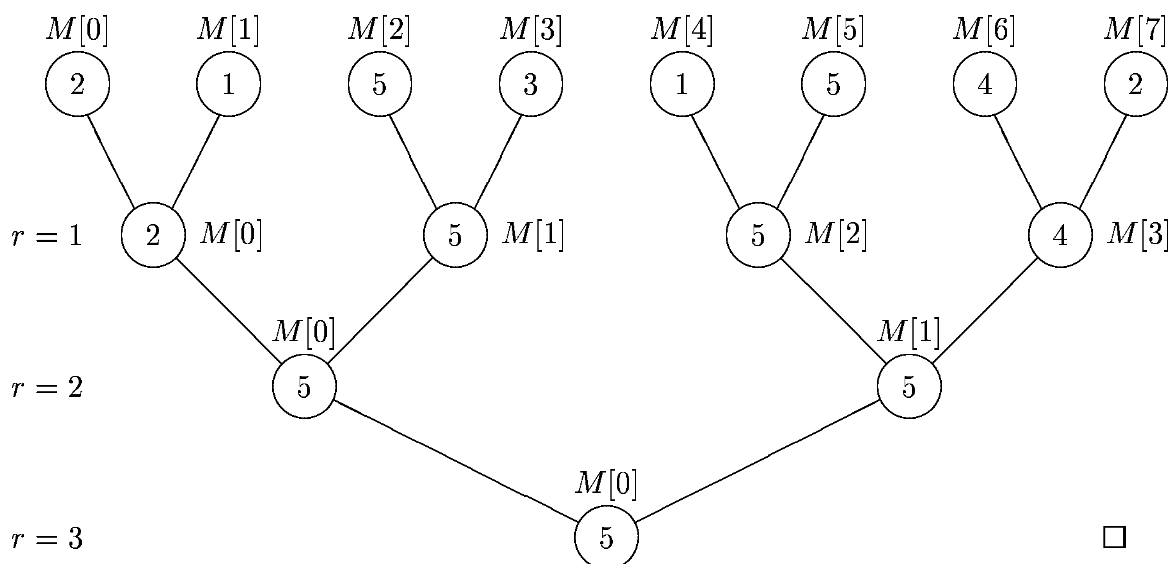
Algorithmus 10.2.1
{Eingabe: n Zahlen $A[0], \ldots, A[n-1]$
Ausgabe: MAX, das Maximum dieser Zahlen}
MAX:= $A[0]$;
for $i = 1$ **to** $n-1$ **do** MAX:= $\max\{A[i], \text{MAX}\}$ **od**. $\square$

Dieser Algorithmus benötigt $O(n)$ Schritte. Es ist klar, daß kein sequentieller Algorithmus MAX in weniger als $O(n)$ Schritten lösen kann. Für eine einfache Parallelisierung ist Algorithmus 10.2.1 allerdings nicht geeignet. Das ist jedoch, wie wir später sehen werden, beim folgenden sogenannten *Turnieralgorithmus* möglich.

Algorithmus 10.2.2
{Eingabe: $n = 2^k$ Zahlen $A[0], \dots, A[n-1]$
Ausgabe: MAX, das Maximum dieser Zahlen}
for $i = 0$ **to** $n-1$ **do** $M[i] := A[i]$;
for $r = 1$ **to** k **do**
  for $i = 0$ **to** $2^{k-r} - 1$ **do**
    $M[i] := \max\{M[2i], M[2i+1]\}$
    od
  od;
MAX:= $M[0]$. □

Zur Veranschaulichung geben wir das folgende Beispiel an.

Beispiel 10.2.1 Es sei $n = 8 = 2^3$. Der Vektor $A = (2, 1, 5, 3, 1, 5, 4, 2)$ sei die Eingabe. Die Blätter des Baums tragen die Werte von M nach der Initialisierung. Darunter stehen die Werte nach den jeweiligen Runden r des Turniers. Das Maximum MAX= 5 ergibt sich nach drei Runden.



□

Algorithmus 10.2.2 benötigt offenbar

$$t(n) = t(2^k) = n + \sum_{r=1}^{k} 2^{k-r} = n + 2^k - 1 = 2n - 1$$

Schritte und ist somit auch von der Ordnung $O(n)$. Die dritte (innere) **for**-Schleife kann parallelisiert werden. Wir benutzen die Notation

for $i = 0$ **to** n **pardo** *stmt* **od**,

um zu beschreiben, daß n Prozessoren das Statement *stmt* parallel ausführen. Das Statement hängt von dem Index i ab und kann selbst eine Folge von Statements sein.

Bevor wir den Algorithmus 10.2.2 entsprechenden parallelen Algorithmus notieren, gehen wir zunächst allgemein auf die Darstellung von parallelen Algorithmen ein. Wir werden die *WT-Darstellung* (work-time presentation) verwenden. Der Algorithmus wird als eine Folge von Operationen zu aufeinanderfolgenden Zeiteinheiten beschrieben, wobei jede Zeiteinheit eine beliebige Zahl von konkurrenten Operationen beinhalten kann. Danach muß man sich überlegen, wie diese parallelen Operationen den einzelnen Prozessoren zugeteilt werden. Diese *Scheduling-Darstellung* kann sehr aufwendig sein, da viele Einzelheiten wie die Verteilung der Aufgaben auf die Prozessoren und die Kommunikation zwischen den Prozessoren bestimmt werden müssen. Wir gehen davon aus, daß diese Verteilung kein Problem darstellt (obwohl es teilweise eine nichttriviale Aufgabe ist) und der Zeitbedarf eines Algorithmus sich somit aus der WT-Darstellung ergibt. Die *Arbeit* $w(n)$ eines Algorithmus ist die Gesamtzahl seiner Operationen.

Der Zusammenhang zwischen den Zeiteinheiten eines parallelen Algorithmus, seiner Arbeit, der Anzahl der verwendeten Prozessoren und der zur Durchführung benötigten Zeit wird im folgenden Satz betrachtet.

Satz 10.2.1 Ein paralleler Algorithmus benötige $t(n)$ Zeiteinheiten. Seine Arbeit sei $w(n)$. Bei p Prozessoren kann der Algorithmus in einer Zeit der Ordnung $O(\frac{w(n)}{p}+t(n))$ durchgeführt werden.

Beweis: Es sei $w_i(n)$ die Anzahl der Operationen in der i-ten Zeiteinheit, $1 \leq i \leq t(n)$. Insgesamt gilt also $\sum_{i=1}^{t(n)} w_i(n) = w(n)$. Die Arbeit der i-ten Zeiteinheit kann in der Zeit $\left\lceil \frac{w_i(n)}{p} \right\rceil \leq \frac{w_i(n)}{p}+1$ mit Hilfe von p Prozessoren durchgeführt werden. Der gesamte Algorithmus benötigt folglich

$$\sum_{i=1}^{t(n)} \left\lceil \frac{w_i(n)}{p} \right\rceil \leq \sum_{i=1}^{t(n)} (\frac{w_i(n)}{p} + 1) = \frac{1}{p} \sum_{i=1}^{t(n)} w_i(n) + t(n) = \frac{w(n)}{p} + t(n)$$

parallele Schritte. □

Wir kommen auf das Problem MAX zurück. Wir geben dafür zwei Algorithmen mit unterschiedlichen PRAM-Modellen an.

Algorithmus 10.2.3
{Eingabe: $n = 2^k$ Zahlen $A[0], \ldots, A[n-1]$
Ausgabe: MAX, das Maximum dieser Zahlen
Modell: EREW PRAM mit n Prozessoren, wobei $A[i]$ und MAX Register des gemeinsamen Speichers bezeichnen. $M[0], \ldots, M[n-1]$ sind Hilfsregister des gemeinsamen Speichers}
for $i = 0$ **to** $n-1$ **pardo** $M[i] := A[i]$ **od**;
for $r = 1$ **to** k **do**
  for $i = 0$ **to** $2^{k-r} - 1$ **pardo**
    $M[i] := \max\{M[2i], M[2i+1]\}$
    od
  od;
MAX:= $M[0]$. □

Satz 10.2.2 Bei n Prozessoren berechnet Algorithmus 10.2.3 das Maximum der Zahlen $A[0], \ldots, A[n-1]$ in $O(\log n)$ Zeiteinheiten. Die Arbeit $w(n)$ ist von der Ordnung $O(n)$.

Beweis: Für die Korrektheit des Algorithmus müssen wir beweisen, daß nach dem k-ten Durchlauf ($k = \log n$) der zweiten **for**-Schleife in dem Register $M[0]$ das Maximum steht. Dies zeigen wir, indem wir für jedes $r = 1, \ldots, k$ durch Induktion über r nachweisen, daß nach dem r-ten Durchlauf

$$M[i] = \max\{A[i \cdot 2^r], A[i \cdot 2^r + 1], \ldots, A[(i+1) \cdot 2^r - 1]\}$$

für $i = 0, \ldots, 2^{k-r} - 1$ gilt. Für $r = 1$ liefert die Zuweisung $M[i] := \max\{M[2i], M[2i+1]\}$ wegen $2i = i \cdot 2^r$ und $2i + 1 = (i+1)2^r - 1$ schon die richtige Gleichung. Die Behauptung sei für $r \geq 1$ erfüllt. Dann liefert die Zuweisung des $(r+1)$-ten Durchlaufs nach Induktionsvoraussetzung

$$\begin{aligned} M[i] &= \max\{\max\{A[(2i) \cdot 2^r], \ldots, A[(2i+1) \cdot 2^r - 1]\}, \\ &\qquad \max\{A[(2i+1) \cdot 2^r], \ldots, A[(2i+2) \cdot 2^r - 1]\}\} \\ &= \max\{A[(2i) \cdot 2^r], \ldots, A[(2i+2) \cdot 2^r - 1]\} \\ &= \max\{A[i \cdot 2^{r+1}], \ldots, A[(i+1) \cdot 2^{r+1} - 1]\}. \end{aligned}$$

Die erste **for**-Schleife erfordert bei n Prozessoren einen parallelen Schritt, die zweite $\log n$ Schritte. Insgesamt ist der Zeitbedarf somit $O(\log n)$. In der ersten **for**-Schleife tragen n Operationen zur Arbeit $w(n)$ bei, in der zweiten $\sum_{r=1}^{k} 2^{k-r} = n - 1$ Operationen. Im letzten Schritt kommt noch eine Operation dazu. Es folgt, daß $w(n)$ von der Ordnung $O(n)$ ist. □

Die Operation max in Algorithmus 10.2.3 kann man durch jede beliebige binäre Operation austauschen, zum Beispiel durch min, + oder OR. Das bedeutet, daß etwa die Summe $\sum_{i=1}^{n} x_i$ auf einer EREW PRAM mit n Prozessoren in $O(\log n)$ Schritten berechnet werden kann.

Unser Algorithmus 10.2.3 auf einer EREW PRAM ist optimal. Es gilt nämlich (siehe [12], Corollary 10.2)

Satz 10.2.3 Jeder Algorithmus für OR auf einer EREW PRAM oder CREW PRAM mit einer beliebigen Anzahl von Prozessoren benötigt $\Omega(\log n)$ Schritte. □

Diese untere Zeitschranke gilt dann natürlich zum Beispiel auch für die Summenbildung.

Mit Hilfe des COMMON CRCW PRAM-Modells geben wir für MAX einen schnelleren Algorithmus an, und zwar einen, der in konstanter Zeit arbeitet.

Algorithmus 10.2.4
{Eingabe: $n = 2^k$ Zahlen $A[0], \ldots, A[n-1]$
Ausgabe: MAX, das Maximum dieser Zahlen
Modell: COMMON CRCW PRAM mit n^2 Prozessoren $P_{i,j}$, $0 \leq i, j \leq n-1$, wobei
$A[i]$, $M[i]$, $i = 0, \ldots, n-1$, und MAX Register des gemeinsamen Speichers.}
for $i = 0$ **to** $n - 1$ **pardo** $M[i] := 1$ **od**;

for $i = 0$ **to** $n-1$ **pardo**
  for $j = 0$ **to** $n-1$ **pardo**
    if $A[i] < A[j]$ **then** $M[i] := 0$
    od
  od;
for $i = 0$ **to** $n-1$ **pardo**
  if $M[i] = 1$ **then** MAX:= $A[i]$
  od. □

Satz 10.2.4 Bei n^2 Prozessoren berechnet Algorithmus 10.2.4 das Maximum der Zahlen $A[0], \ldots, A[n-1]$ in $O(1)$ Zeiteinheiten. Die Arbeit $w(n)$ ist von der Ordnung $O(n^2)$.

Beweis: Es soll $M[i] = 1$ genau dann gelten, wenn $A[i]$ das Maximum ist. Nach der ersten **for**-Schleife wird das für jedes $A[i]$ angenommen. In der nächsten (doppelten) **for**-Schleife werden genau die Elemente $A[i]$, die kein Maximum sind, durch $M[i] = 0$ gekennzeichnet. Für ein festes i können mehrere Prozessoren gleichzeitig $M[i]$ beschreiben, dann aber, in Konsistenz mit dem COMMON CRCW PRAM-Modell, alle denselben Wert 0. In der letzten **for**-Schleife wird das Maximum in das Register MAX geschrieben. Dies kann durch mehrere Prozessoren erfolgen. In diesem Fall schreiben sie wieder alle denselben Wert.

Der Algorithmus erfordert 3 parallele Schritte, die Zeitkomplexität ist daher von der Ordnung $O(1)$. Die Arbeit zu Beginn und am Ende erfordert jeweils n Operationen. Die doppelte **for**-Schleife benötigt n^2 Operationen. Die Arbeitskomplexität ist somit durch $O(n^2)$ gegeben. □

MAX kann in $O(1)$ Schritten auf einer COMMON CRCW PRAM berechnet werden. Speziell kann die logische Funktion OR, das heißt die Auswertung von $x_1 \vee x_2 \vee \ldots \vee x_n$, in $O(1)$ Schritten erfolgen. Dies geschieht einfach durch Anwendung von MAX auf die binäre Liste $(x_1, \ldots, x_n)$. Anstelle von MAX kann Algorithmus 10.2.4 auch MIN berechnen, indem $<$ durch $>$ und MAX durch MIN ersetzt wird. Folglich kann auch AND auf einer COMMON CRCW PRAM in konstanter Zeit berechnet werden.

Es ist bekannt, daß das sequentielle Sortieren von n Zahlen die Zeit $\Omega(n \log n)$ benötigt. Wir geben einen sehr einfachen Algorithmus an, der auf einer COMMON CRCW PRAM mit n^2 Prozessoren in der Zeit $O(\log n)$ sortieren kann.

Algorithmus 10.2.5
{Eingabe: n Zahlen $A[1], \ldots, A[n]$
Ausgabe: Sortierte Liste $S[1] \leq \ldots \leq S[n]$ derselben Zahlen
Modell: COMMON CRCW PRAM mit n^2 Prozessoren $P_{i,j}$, $1 \leq i, j \leq n$
  Im gemeinsamen Speicher:
  $A[i]$, $S[i]$, $i = 1, \ldots, n$, $a_{i,j}$, $a'_{i,j}$, b_j und b'_j, $1 \leq i, j \leq n$
  b_j = Anzahl der Komponenten $\leq A[j]$
  b'_j = Anzahl der Komponenten $< A[j]$}
for $i = 1$ **to** n **pardo**
  for $j = 1$ **to** n **pardo**
    if $A[i] \leq A[j]$ **then** $a_{i,j} := 1$ **else** $a_{i,j} := 0$;

```
        if A[i] < A[j] then a'_{i,j} := 1 else a'_{i,j} := 0
        od
    od;
for j = 1 to n pardo
    b_j := sum_{i=1}^{n} a_{i,j};
    b'_j := sum_{i=1}^{n} a'_{i,j}
    od;
for j = 1 to n pardo
    for i = b'_j + 1 to b_j pardo
        S[i] := A[j]
        od
    od.  □
```

Satz 10.2.5 Bei n^2 Prozessoren sortiert Algorithmus 10.2.5 die Zahlen $A[1], \ldots, A[n]$ in $O(\log n)$ Zeiteinheiten. Die Arbeit $w(n)$ ist von der Ordnung $O(n^2)$.

Beweis: Es ist klar, daß b_j die Anzahl der gegebenen Zahlen $\leq A[j]$ und b'_j die Anzahl derjenigen $< A[j]$ angibt. Das bedeutet, daß in der sortierten Liste das erste Vorkommen von $A[j]$ an der Position $b'_j + 1$ und das letzte an der Position b_j stehen muß. Folglich werden in der letzten (doppelten) **for**-Schleife die Elemente richtig eingetragen, ggf. auch durch mehrere Prozessoren gleichzeitig (siehe Beispiel 10.2.2).

Die Berechnung der $a_{i,j}$ und $a'_{i,j}$ benötigt $O(1)$ parallele Schritte bei einem Arbeitsaufwand von $O(n^2)$. Die Berechnung der Summen b_j und b'_j erfolgt für jedes j nach den Bemerkungen im Anschluß an Satz 10.2.2 ähnlich Algorithmus 10.2.3 in der parallelen Zeit $O(\log n)$ mit n zusätzlichen Prozessoren. Insgesamt reichen dafür also n^2 Prozessoren, der Arbeitsaufwand ist von der Ordnung $O(n^2)$. Die letzte doppelte **for**-Schleife erfordert einen parallelen Schritt bei einer Arbeit von $O(n^2)$. Wir erhalten damit die angegebenen Komplexitätsabschätzungen. □

Es ist ein optimales Sortierverfahren bekannt (siehe [12], Corollary 4.5), das in $O(\log n)$ parallelen Schritten mit dem Arbeitsaufwand $O(n \cdot \log n)$ auf einer EREW PRAM mit n Prozessoren läuft. Die Einzelheiten sind sehr viel komplizierter als in unserem Algorithmus 10.2.5.

Beispiel 10.2.2 Wir betrachten die Liste der Zahlen

$$A = (7, 6, 7, 8, 6).$$

Daraus ergeben sich zunächst die Matrizen

$$(a_{i,j}) = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \quad \text{und} \quad (a'_{i,j}) = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}.$$

Dann ergeben sich als Summe der Spalten die Werte

$$(b_j) = (4,2,4,5,2) \quad \text{und} \quad (b'_j) = (2,0,2,4,0).$$

In der letzten doppelten **for**-Schleife werden die folgenden Eintragungen parallel vorgenommen:

$A[1] = 7$ in die Positionen 3 und 4,
$A[2] = 6$ in die Positionen 1 und 2,
$A[3] = 7$ in die Positionen 3 und 4,
$A[4] = 8$ in die Position 5,
$A[5] = 6$ in die Positionen 1 und 2.

Diese Arbeit kann von einer COMMON CRCW PRAM in einem Schritt geleistet werden. Wir erhalten die sortierte Liste

$$S = (6,6,7,7,8). \quad \square$$

Der sequentielle Algorithmus für KONVEXE HÜLLE, der im Beweis von Satz 6.5.2 beschrieben wurde und einen Zeitbedarf von $O(n \log n)$ hatte, läßt sich sehr einfach parallelisieren.

Satz 10.2.6 Das Problem KONVEXE HÜLLE für n vorgegebene Punkte der Ebene kann auf einer COMMON CRCW PRAM mit n Prozessoren in der Zeit $O(\log^2 n)$ berechnet werden. Die Arbeit ist von der Ordnung $O(n \log n)$.

Beweis: Wir gehen auf den Beweis von Satz 6.5.2 zurück. Es werden dieselben Operationen wie dort durchgeführt, jedoch teilweise in paralleler Weise. Die Arbeit ist daher von der Ordnung $O(n \log n)$. Die Bezeichnungen werden aus dem Beweis von Satz 6.5.2 übernommen.

Das anfängliche Sortieren kann in $O(\log n)$ parallelen Schritten mit n Prozessoren entsprechend Algorithmus 10.2.5 durchgeführt werden. Die Berechnung der oberen Hülle hängt von der Berechnung der oberen Tangente ab. Es wird zunächst für jeden Punkt p_i, $i = 1, \ldots, s$, die obere Tangente von p_i zu den Punkten $q_1, \ldots, q_t$ in der Zeit $O(\log t)$ berechnet. Dies kann parallel mit $s \leq n$ Prozessoren in der Zeit $O(\log t)$ erfolgen. Dabei wird auch konkurrentes Lesen benötigt. Die obere Tangente kann dann sequentiell wie im Beweis von Satz 6.5.2 in der Zeit $O(\log n)$ gefunden werden. Die obere Hülle ergibt sich rekursiv wie folgt: Parallel werden die beiden Hüllen der linken und rechten Hälfte der Punkte berechnet und anschließend ihre obere Tangente. Folglich gilt für den parallelen Zeitbedarf zur Berechnung der oberen Hülle

$$t(n) \leq t(\frac{n}{2}) + k \log n$$

für eine Konstante k. Dabei werden nicht mehr als n Prozessoren benötigt. Die Ungleichung wird vergröbert zu

$$t(n) \leq t(\frac{n}{2}) + 2k \log n - k.$$

Für die Gleichung $t(n) = t(\frac{n}{2}) + 2k \log n - k$ erhalten wir die Lösung $t(n) = k \log^2 n$

$$(\text{es gilt } k \log^2 \frac{n}{2} + 2k \log n - k = k(\log n - 1)^2 + 2k \log n - k = k \log^2 n).$$

Damit gehört die Zeitkomplexität zur Bestimmung der oberen Hülle zu $O(\log^2 n)$. Die Berechnung der unteren Hülle benötigt analog $O(\log^2 n)$ Zeiteinheiten. Anschließend kann die gesamte konvexe Hülle in einem Schritt mit n Prozessoren geschrieben werden. □

Definition 10.2.2 Es sei $(x_1, \ldots, x_n)$ eine Folge von Elementen einer Menge S und $*$ eine binäre Operation auf S. Die *Präfixsummen* dieser Folge sind die n partiellen „Summen"

$$s_i = x_1 * x_2 * \ldots * x_i, \; 1 \leq i \leq n. \quad \square$$

Es ist klar, daß ein trivialer sequentieller Algorithmus durch Berechnung von $s_i = s_{i-1} * x_i$ für $2 \leq i \leq n$ in der Zeit $O(n)$ die Präfixsummen berechnet. Wir geben einen rekursiven parallelen Algorithmus für diese Berechnung an, der nur die Zeit $O(\log n)$ benötigt.

Algorithmus 10.2.6

```
{Eingabe: Folge (x_1, ..., x_n) von n = 2^k Elementen, k ∈ ℕ_0
Ausgabe:  die Präfixsummen s_i, 1 ≤ i ≤ n
Modell:   EREW PRAM mit n Prozessoren}
procedure prefix(x_1, ..., x_n);
begin
   if n = 1
   then s_1 := x_1
   else for i = 1 to n/2 pardo
            y_i := x_{2i-1} * x_{2i}
        od;
        (z_1, ..., z_{n/2}) := prefix(y_1, ..., y_{n/2});
        for i = 1 to n pardo
            {i gerade          : s_i := z_{i/2}}
            od;
        for i = 1 to n pardo
            {i = 1             : s_1 := x_1
             i > 1 ungerade    : s_i := z_{(i-1)/2} * x_i}
            od
end. □
```

Satz 10.2.7 Algorithmus 10.2.6 berechnet die Präfixsummen von n Elementen auf einer EREW PRAM mit n Prozessoren in der Zeit $O(\log n)$. Die Arbeit ist von der Ordnung $O(n)$.

Beweis: Wir nehmen ohne Beschränkung der Allgemeinheit, wie schon im Algorithmus, $n = 2^k$ für ein $k \in \mathbb{N}_0$ an. Durch Auffüllen mit bzgl. $*$ neutralen Elementen kann dies

immer erreicht werden. Der Beweis erfolgt durch Induktion über k. Im Fall $k = 0$, also $n = 1$, wird durch die erste Alternative des **if**-Statements die richtige Aktion ausgeführt. Wir nehmen an, daß der Algorithmus für alle Folgen der Länge $n = 2^k$, $k \geq 0$, richtig arbeitet. Wir beweisen, daß er die Präfixsummen einer jeden Folge der Länge $n = 2^{k+1}$ berechnet.

Nach Induktionsannahme beinhalten die Variablen $z_1, \ldots z_{\frac{n}{2}}$, die im rekursiven Aufruf von prefix berechnet werden, die Präfixsummen der Folge $(y_1, \ldots, y_{\frac{n}{2}})$, wobei

$$y_i = x_{2i-1} * x_{2i} \text{ für } 1 \leq i \leq \frac{n}{2}$$

gilt. Insbesondere ist $z_j = y_1 * y_2 * \ldots * y_j$ und folglich

$$z_j = x_1 * x_2 * \ldots * x_{2j-1} * x_{2j}.$$

Das bedeutet, daß z_j die Präfixsumme s_{2j} für $1 \leq j \leq \frac{n}{2}$ ist. Ist i gerade, dann folgt $s_i = z_{\frac{i}{2}}$. Anderenfalls gilt $i = 1$ oder $i = 2j + 1$ für ein j mit $1 \leq j \leq \frac{n}{2} - 1$. Der Fall $i = 1$ ist trivial. Für $i = 2j + 1$ erhalten wir

$$s_i = s_{2j+1} = s_{2j} * x_{2j+1} = z_{\frac{i-1}{2}} * x_i.$$

Daher werden alle Fälle in der zweiten und dritten **for**-Schleife richtig behandelt. Es sind am Ende zwei **for**-Schleifen erforderlich, damit es nicht zu einem konkurrenten READ kommt. Der Algorithmus arbeitet also korrekt auf allen Eingaben.

Nun zu den Komplexitätsabschätzungen. Für $n = 1$ benötigen wir $O(1)$ Schritte. Die **for**-Schleifen sind jeweils in $O(1)$ parallelen Schritten ausführbar und benötigen $O(n)$ Operationen. Wegen des rekursiven Aufrufs ergeben sich folglich die Laufzeit $t(n)$ und die Arbeit $w(n)$ des Algorithmus durch die Gleichungen

$$\begin{aligned} t(n) &= t(\tfrac{n}{2}) + a \text{ und} \\ w(n) &= w(\tfrac{n}{2}) + bn, \end{aligned}$$

wobei a und b Konstanten sind. Die Lösungen dieser Rekurrenzgleichungen sind, wie eine Probe sofort beweist, $t(n) = O(\log n)$ und $w(n) = O(n)$. □

10.3 Simulationen zwischen verschiedenen PRAM-Modellen

In diesem Abschnitt werden wir sehen, daß sich die verschiedenen PRAM-Modelle nicht allzu sehr voneinander unterscheiden.

Satz 10.3.1 Jeder Algorithmus, der auf einer
(a) EREW PRAM
(b) CREW PRAM
(c) COMMON CRCW PRAM
mit p Prozessoren in der Zeit T läuft, kann auf einer
(a) CREW PRAM
(b) COMMON CRCW PRAM
(c) PRIORITY CRCW PRAM
mit derselben Anzahl von Prozessoren in derselben Zeit ausgeführt werden.

Beweis: Nach den Definitionen auf Seite 192 ist dies trivial. □

Satz 10.3.2 Jeder Algorithmus, der auf einer PRIORITY CRCW PRAM mit p Prozessoren in der Zeit T läuft, kann auf einer EREW PRAM mit derselben Anzahl von Prozessoren in der Zeit $O(T \log p)$ ausgeführt werden.

Beweis: Es genügt, jede READ- und WRITE-Instruktion mit $\log p$ Schritten der EREW PRAM zu simulieren. Wir beginnen mit dem konkurrenten READ. Wir nehmen an, daß die CRCW PRAM die Prozessoren $Q_1, \dots, Q_p$ besitzt, und der Prozessor Q_i lese das Register $\hat{M}_{j_i}$ des gemeinsamen Speichers. Mit $P_1, \dots, P_p$ bezeichnen wir die Prozessoren der simulierenden EREW PRAM. In ihrem gemeinsamen Speicher werden Register $M_1, \dots, M_p$ für spezielle Benutzung bereitgestellt. Wir erhalten den folgenden Algorithmus:

for $i = 1$ **to** p **pardo**
    $M_i := (j_i, i)$ {Prozessor P_i schreibt in M_i}
    od;
Sortierung der Paare (j_i, i) in lexikographischer Ordnung;
$\{$ergibt Paare (j_{x_i}, x_i), $i = 1, \dots, n\}$
P_1 liest aus $\hat{M}_{j_{x_1}}$;
for $i = 2$ **to** p **pardo**
    if $j_{x_i} \neq j_{x_{i-1}}$ **then** P_i liest aus $\hat{M}_{j_{x_i}}$
    od;
für alle j_i: Weiterreichen des Wertes aus M_{j_i} an alle Prozessoren $P_{i'}$ mit $j_{i'} = j_i$.

Die erste **for**-Schleife erfordert $O(1)$ Schritte und p Prozessoren. Das Sortieren kann nach der Bemerkung im Anschluß an den Beweis von Satz 10.2.5 mit p Prozessoren in der Zeit $O(\log p)$ durchgeführt werden. In der zweiten **for**-Schleife wird für jedes j_i der zugehörige Prozessor mit dem kleinsten Index ausgesucht. Diese Repräsentanten simulieren in der Zeit $O(1)$ ein konkurrentes READ. Die Verteilung des Wertes an die Nachbarn mit demselben j_i erfolgt in $O(\log p)$ Schritten mit p Prozessoren, da es höchstens p Nachbarn gibt. Dies kann durch einen segmentierten Präfixsummenalgorithmus geschehen, wobei die Segmente durch die i mit gleichem j_i bestimmt sind. Ist y_{j_i} der Wert von M_{j_i}, so kann der Wert y_{j_i} im Segment $(y_{j_i}, 0, \dots, 0)$ mit Hilfe von Algorithmus 10.2.6 an das ganze Segment in $O(\log p)$ Schritten verteilt werden. Dabei können die hier speziell mit 0 gekennzeichneten x_i im Algorithmus 10.2.6 einfach weggelassen werden. Folglich entfallen die Operationen $*$, eine entsprechend vereinfachte Version des Algorithmus kann verwendet werden.

Die Simulation einer WRITE-Instruktion ist etwas einfacher. Der Prozessor Q_i schreibe die Daten d_i in das Register $\hat{M}j_i$. Wir erhalten den folgenden Algorithmus:

for $i = 1$ **to** p **pardo**
    $M_i := (j_i, i, d_i)$ {Prozessor P_i schreibt in M_i}
    od;
Sortierung der Tripel (j_i, i, d_i) in lexikographischer Ordnung;
$\{$ergibt Paare (j_{x_i}, x_i, d_{x_i}), $i = 1, \dots, n\}$
P_1 schreibt d_{x_1} in $\hat{M}_{j_{x_1}}$;
for $i = 2$ **to** p **pardo**
    if $j_{x_i} \neq j_{x_{i-1}}$ **then** P_i schreibt d_{x_i} in $\hat{M}_{j_{x_i}}$
    od

Die Zeitabschätzungen sind erfüllt (siehe READ). Aufgrund der Sortierung der Tripel (j_i, i, d_i) ist es klar, daß nur der Prozessor mit dem kleinsten Index schreibt und die anderen inaktiv bleiben. Folglich versuchen niemals mehrere Prozessoren, gleichzeitig in dasselbe Register zu schreiben. □

Aus den Sätzen 10.3.1 und 10.3.2 folgt

Satz 10.3.3 Jeder Algorithmus, der auf einer
(a) PRIORITY CRCW PRAM
(b) CREW PRAM
mit p Prozessoren in der Zeit T läuft, kann auf einer
(a) CREW PRAM
(b) EREW PRAM
mit derselben Anzahl von Prozessoren in der Zeit $O(T \log p)$ ausgeführt werden. □

Diese Simulationsergebnisse können nicht verbessert werden. So zeigt Satz 10.2.4 und die anschließenden Überlegungen, daß das Boolesche OR auf einer COMMON CRCW PRAM und dann nach Satz 10.3.1 erst recht auf einer PRIORITY CRCW PRAM in der Zeit $O(1)$ berechnet werden kann. Andererseits benötigt nach Satz 10.2.3 jeder Algorithmus für OR auf einer EREW PRAM oder CREW PRAM $\Omega(\log n)$ Schritte, wobei n die Anzahl der Variablen ist.

Aus den Sätzen 10.3.1 und 10.3.2 schließen wir, daß die verschiedenen PRAM-Modelle sich gegenseitig simulieren können, wobei der Zeitverlust höchstens durch einen Faktor der Ordnung $O(\log p)$ gegeben wird. Die Simulation einer PRIORITY CRCW PRAM durch eine COMMON CRCW PRAM ist jedoch mit einem konstanten Faktor möglich:

Satz 10.3.4 Jeder Algorithmus, der auf einer PRIORITY CRCW PRAM mit p Prozessoren in der Zeit T läuft, kann auf einer COMMON CRCW PRAM mit p^2 Prozessoren in der Zeit $O(T)$ ausgeführt werden.

Beweis: Es wird ein „Erlaubnisvektor" perm aufgestellt, indem die Schreibwünsche der p Prozessoren $Q_1, \dots, Q_p$ der gegebenen PRAM paarweise verglichen werden. Der Wert perm(i) ist 1, wenn der i-te Prozessor schreiben darf. Wir nehmen an, daß Q_j, $j = 1, \dots, p$, den Wert d_j in das Register $\hat{M}_{r_j}$ schreiben möchte. Wir benutzen p^2 Prozessoren $P_{i,j}$, $1 \leq i, j \leq p$. Die Simulation einer einzelnen WRITE-Instruktion kann durch den folgenden Algorithmus durchgeführt werden:

for $i = 1$ **to** p **pardo**
  perm(i) := 1
  od;
for $i = 1$ **to** p **pardo**
  for $j = 1$ **to** p **pardo**
    if $i < j$ und $r_i = r_j$ **then** perm(j) := 0
    od
  od;
for $i = 1$ **to** p **pardo**
  if perm(i) = 1 **then** $M_{r_i} := d_i$
  od

Ersichtlich ist dies in konstanter Zeit möglich. □

10.4 Die Komplexitätsklasse *NC*

Unsere Algorithmen aus Abschnitt 10.2 haben gezeigt, daß Parallelisierung eine bedeutende Verbesserung der Effizienz nach sich zieht. Für Eingaben der Größe n benötigten die angegebenen Algorithmen statt polynomial vielen Schritten (d.h. $p(n)$ Schritte für ein Polynom p) nur eine Anzahl der Schritte von der Ordnung $O(\log n)$, $O(\log^2 n)$ oder sogar $O(1)$. Die Anzahl der Prozessoren war dabei durch n oder n^2 gegeben. Im allgemeinen können wir sagen, daß ein Problem effizient parallelisierbar ist, falls es durch eine PRAM gelöst werden kann, die nur polylogarithmische Zeitkomplexität hat und nur eine polynomiale Anzahl von Prozessoren benutzt. Unter polylogarithmischer Zeitkomplexität verstehen wir, daß es ein Polynom p gibt, so daß für Eingaben der Länge n höchstens $p(\log n)$ Schritte notwendig sind. Die Forderung, daß höchstens eine polynomiale Anzahl von Prozessoren benutzt werden darf, ist sehr wesentlich, und zwar nicht nur aus praktischen sondern auch aus theoretischen Gründen. Bei exponentiell vielen Prozessoren geht der Unterschied zwischen den Klassen P und NP verloren, was der folgende Algorithmus für SAT(k) zeigt.

Algorithmus 10.4.1
{Eingabe: Es sei α ein Boolescher Ausdruck in konjunktiver Normalform mit n Variablen $x_0, \ldots, x_{n-1}$ und k Klauseln
Ausgabe: $x = 1$, falls α erfüllbar ist, sonst $x = 0$
Modell: PRAM mit 2^n Prozessoren P_i, $i = 0, \ldots, 2^n - 1$
Die Indizes haben die binäre Darstellung $i = i_{n-1} \ldots i_0$
x sei ein Register des gemeinsamen Speichers}
$x := 0$;
for $i = 0$ **to** $2^n - 1$ **pardo**
  if $\Psi_i(\alpha) = 1$ für die Belegung Ψ_i mit $(\Psi_i(x_s) = 1 \iff i_s = 1)$, $s = 0, \ldots, n-1$
  then $x := 1$
  od □

Satz 10.4.1 Es seien $k, n \in \mathbb{N}$. Algorithmus 10.4.1 kann SAT (mit n Varaiablen und k Klauseln) in der Zeit $O(n \cdot k)$ entscheiden, wobei n die Anzahl der Variablen bedeutet.

Beweis: Jeder Prozessor P_i, $i = 0, \ldots, 2^n - 1$, findet für jede Klausel den Wert der Belegung Ψ_i (Bildung von MAX) sequentiell in der Zeit $O(n)$. Insgesamt ergibt sich $\Psi_i(\alpha)$ mit Hilfe von MIN über alle k Klauseln in der Zeit $O(n \cdot k)$. □

Da nach Definition 7.2.1 jedes Problem aus NP sequentiell in polynomialer Zeit auf das NP-vollständige Problem SAT reduziert werden kann, erhalten wir als Folgerung:

Satz 10.4.2 Jedes Problem aus NP der Größe n, $n \in \mathbb{N}$, kann mit 2^n Prozessoren in polynomialer Zeit gelöst werden. □

Die schon oben erwähnte effiziente Parallelisierbarkeit wird durch die Klasse NC (Nick's Class) formalisiert, die 1979 von *Nick Pippenger* in einem anderen Zusammen-

hang eingeführt wurde.

Definition 10.4.1 Es ist
$NC = \{L \mid L \subset \Sigma^*$, es existiert eine PRAM und zwei Polynome p und q, so daß für jedes Wort $w \in \Sigma^*$ mit $|w| = n$ die PRAM in $O(p(\log n))$ Schritten hält, dabei $O(q(n))$ Prozessoren benutzt und w genau dann akzeptiert, wenn $w \in L$ gilt$\}$
die *Komplexitätsklasse der durch PRAMs in polylogarithmischer Zeit mit polynomialer Anzahl von Prozessoren entscheidbaren Sprachen.* □

Für das PRAM-Modell dieser Definition stellen wir uns vor, daß eine PRAM prinzipiell mit beliebig vielen Prozessoren arbeiten kann. Je nach Größe n der Eingabe werden davon $O(q(n))$ benutzt. In Definition 10.4.1 muß nicht auf das spezielle PRAM-Modell Bezug genommen werden. Das ergibt sich einerseits daraus, daß die Simulierungen nach Satz 10.3.1 jeweils in der gleichen Zeit mit der jeweils gleichen Anzahl von Prozessoren erfolgen. Andererseits gibt es für jede PRIORITY CRCW PRAM mit polylogarithmischer Zeitkomplexität $O(p(\log n))$ und $q(n)$ Prozessoren bei Eingaben der Länge n nach Satz 10.3.2 eine sie simulierende EREW PRAM mit $q(n)$ Prozessoren und der Zeitkomplexität

$$O(p(\log n) \cdot \log q(n))$$

für geeignete Polynome p und q. Ist der Grad von p gleich k und der von q gleich r, so erhalten wir

$$\begin{aligned} O(p(\log n) \cdot \log q(n)) &= O((\log n)^k \cdot \log(n^r)) = O((\log n)^k \cdot r \log n) \\ &= O((\log n)^k \log n) = O((\log n)^{k+1}). \end{aligned}$$

Die EREW PRAM hat also eine polylogarithmische Zeitkomplexität.

Beispiel 10.4.1 Nach Satz 10.2.2 gehört das Entscheidungsproblem, ob bei beliebig vorgegebenen Zahlen $A[0], \ldots, A[n-1]$ und einem Wert m die Gleichung $m = \max\{A[i] \mid i = 0, \ldots, n-1\}$ gilt, zu NC. □

Analog zur Klasse *FP* können wir für Funktionen $f : \Gamma^* \to \Sigma^*$ die Klasse *FNC* einführen.

Definition 10.4.2 Es ist
$FNC = \{f : \Sigma^* \to \Gamma^* \mid$ es existiert eine PRAM, die f berechnet und für alle $w = x_1 \ldots x_n$, $x_1, \ldots, x_n \in \Sigma$, $n \in \mathbb{N}_0$, mit höchstens $q(n)$ Prozessoren in $O(p(\log n))$ Schritten hält, wobei p und q Polynome sind$\}$
die *Klasse aller totalen Funktionen, die in polylogarithmischer Zeit mit polynomialer Anzahl von Prozessoren berechnet werden können.* □

Es ist klar, daß die verschiedenen durch die Algorithmen in Abschnitt 10.2 berechneten Funktionen in der Klasse *FNC* liegen.

Satz 10.4.3 $NC \subset P$.

Beweis: Es sei L eine Sprache, die durch eine PRAM mit der Zeitkomplexität $O((\log n)^k)$ und mit $p(n)$ Prozessoren für jede Eingabe der Länge n entschieden wird. Eine solche PRAM kann durch eine RAM, also mit einem einzigen Prozessor, simuliert werden, indem der Prozessor jede parallele Operation, die die $p(n)$ Prozessoren auszuführen haben, sequentiell abarbeitet. Somit kann jeder Schritt der PRAM durch höchstens $O(p(n))$ Schritte der RAM simuliert werden. Wegen $\lim_{n\to\infty} \frac{(\log n)^k}{n} = 0 < 1$ hat die RAM die Zeitkomplexität $(\log n)^k \cdot p(n) = O(n \cdot p(n))$. □

Es ist offen, ob $NC = P$ gilt oder nicht. Allgemein vermutet man, daß $NC \neq P$ gilt. Das würde bedeuten, daß es Probleme gibt, die zwar effizient sequentiell gelöst werden können, die aber keine effiziente Parallelisierung erlauben. Analog zu den *NP*-vollständigen Problemen in der Klasse *NP* gibt es Probleme, die eine besondere Stellung in der Klasse P haben, die sogenannten P-vollständigen Probleme. Sie haben die Eigenschaft, daß alle Probleme der Klasse P effizient parallelisierbar wären, falls für nur ein P-vollständiges Problem diese Eigenschaft nachgewiesen werden könnte.

Definition 10.4.3 Es seien Σ, Γ Alphabete und $L_1 \subset \Sigma^*$, $L_2 \subset \Gamma^*$. Die Sprache L_1 ist *NC-reduzierbar* auf die Sprache L_2, wenn eine (totale) Funktion $f : \Sigma^* \to \Gamma^*$ aus *FNC* existiert, so daß

$$w \in L_1 \iff f(w) \in L_2 \quad \text{für alle } w \in \Sigma^*$$

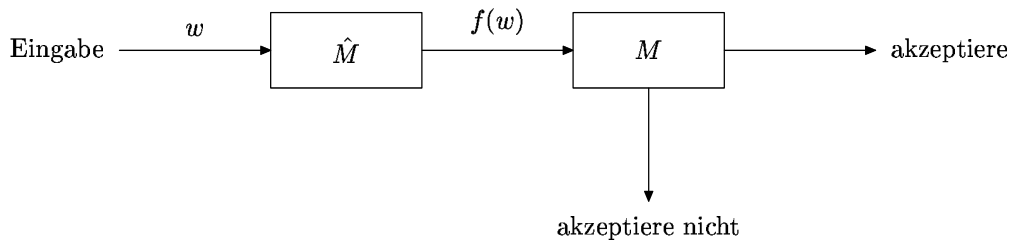
erfüllt ist. □

Definition 10.4.4 Es sei L eine Sprache. L heißt *P-vollständig*, wenn
(a) $L \in P$ ist und
(b) jede Sprache $L' \in P$ eine *NC*-Reduktion auf L besitzt. □

Satz 10.4.4 Es sei L eine P-vollständige Sprache. Es gilt $L \in NC$ genau dann, wenn $P = NC$ ist.

Beweis: Nach Definition 10.4.4(a) gilt $L \in P$. Ist $P = NC$, dann folgt offenbar $L \in NC$. Umgekehrt sei $L \in NC$. Dann existiert eine PRAM M, die ein Wort $w \in L$ mit $|w| = n$ in der Zeit $O(\log p(n))$ bei Aktivierung von $q(n)$ Prozessoren für Polynome p und q akzeptiert. Nach Satz 10.4.3 müssen wir zeigen, daß jede Sprache $L' \in P$ auch zu *NC* gehört. Da L P-vollständig ist, existiert eine *NC*-Reduktion f von L' auf L. Das bedeutet, daß eine PRAM $\hat{M}$ existiert, die für alle $w \in \Sigma^*$ mit $|w| = n$ den Wert $f(w)$ in der Zeit $O(\hat{p}(\log n))$ mit Hilfe von $\hat{q}(n)$ Prozessoren für Polynome $\hat{p}$ und $\hat{q}$ berechnet.

Wir konstruieren eine PRAM M', die L' akzeptiert. M' simuliert zunächst die PRAM $\hat{M}$, um aus der Eingabe w das Wort $f(w)$ zu berechnen. Danach simuliert M' die PRAM M mit der Eingabe $f(w)$. Die PRAM M' akzeptiert genau dann w, wenn M die Eingabe $f(w)$ akzeptiert. Da f eine *NC*-Reduktion von L' auf L ist, gilt $f(w) \in L$ genau dann, wenn $w \in L'$ erfüllt ist. Folglich akzeptiert M' die Sprache L'. Diese Arbeitsweise wird durch das folgende Bild dargestellt.



Die Simulation von $\hat{M}$ dauert $O(\hat{p}(\log n))$ Schritte. In jedem Schritt schreibt ein Prozessor höchstens ein Symbol. Folglich hat $f(w)$ eine Länge $m = O(\hat{q}(n) \cdot \hat{p}(\log n))$. Die Simulation von M benötigt $O(p(\log m))$ Schritte. Da auch $m = O(n^k)$ gilt, wobei k der Grad des Polynoms $\hat{q} \cdot \hat{p}$ ist, folgt

$$p(\log m) = O(p(\log n^k)) = O(p(k \cdot \log n)) = O(p(\log n)).$$

Damit hat M' die polylogarithmische Zeitkomplexität $O(\hat{p}(\log n) + p(\log n))$. Es werden bei der Arbeit von M' zunächst $O(\hat{q}(n))$ Prozessoren von $\hat{M}$ und anschließend $O(q(m)) = O(q(n^k))$ Prozessoren von M aktiviert. Insgesamt werden höchstens $O(\hat{q}(n) + q(n^k))$ Prozessoren benötigt. Somit gilt $L \in NC$ und damit $NC = P$. $\square$

Der folgende Satz nennt ein P-vollständiges Problem.

Satz 10.4.5 Für einen gerichteten schlichten Graphen ist das Problem, ob ein maximaler Fluß eine gerade Anzahl von Wegen hat, P-vollständig. $\square$

Der Beweis dieses Satzes findet sich in [12], S. 543–551. In dem Buch werden noch weitere P-vollständige Probleme besprochen.

10.5 Boolesche Schaltkreise und PRAMs

Parallele Berechnungen werden oft mit (Booleschen) Schaltkreisen modelliert. Es zeigt sich, daß dies unter bestimmten Bedingungen äquivalent zur Berechnung mit CRCW PRAMs ist. Einen Teil dieser Äquivalenz werden wir in Satz 10.5.1 beweisen. Boolesche Schaltkreise berechnen, wie wir unten sehen werden, Boolesche Funktionen. Der Zusammenhang zwischen Booleschen Funktionen $f : \{0,1\}^n \to \{0,1\}$, $n \in \mathbb{N}$, und Entscheidungsproblemen ist klar ersichtlich, wenn man sich auf binär kodierte Probleme beschränkt, also auf formale Sprachen über dem Alphabet $\{0,1\}$. Eine Boolesche Funktion f mit n Variablen wird dazu benutzt, die Entscheidung für Wörter der Länge n zu treffen. Dies geschieht dadurch, daß ein Wort $w = x_1 \ldots x_n$, $x_i \in \{0,1\}$, $i = 1, \ldots, n$, genau dann in L liegen soll, wenn $f(x_1, \ldots, x_n) = 1$ gilt. Es ist klar, daß wir zur Entscheidung einer ganzen Sprache eine Familie von Booleschen Funktionen benötigen, nämlich genau eine für jedes $n \in \mathbb{N}$.

Im folgenden setzen wir $\mathbb{B} = \{0,1\}$, wobei 0 auch mit dem Wahrheitswert „falsch“ und 1 mit „wahr“ identifiziert wird.

Definition 10.5.1 Ein *Schaltkreis* S_n *mit* n *Eingängen*, $n \in \mathbb{N}$, (und einem Ausgang) wird durch einen endlichen gerichteten zyklenfreien Graphen $G = (V, E)$ definiert, bei dem mehrere Kanten gleiche Anfangs- und Endpunkte haben dürfen, und der die folgenden Eigenschaften erfüllt:

(a) Jeder Knoten hat ≤ 2 Vorgänger und ist mit einer Funktion beschriftet, die im folgenden definiert wird.

(b) Es gibt genau n Knoten ohne Vorgänger. Sie sind mit den paarweise verschiedenen Projektionsfunktionen $\pi_i : \mathbb{B}^n \to \mathbb{B}$, $i = 1, \ldots, n$, beschriftet. Diese Knoten heißen auch *Eingänge* von S. Statt π_i können wir uns die Knoten auch mit der entsprechenden Variablen x_i markiert denken.

(c) Alle Knoten mit einem Vorgänger und mindestens einem Nachfolger sind mit der Negationsfunktion $\neg : \mathbb{B} \to \mathbb{B}$ beschriftet.

(d) Jeder Knoten mit zwei Vorgängern ist entweder mit $\wedge : \mathbb{B}^2 \to \mathbb{B}$ oder mit $\vee : \mathbb{B}^2 \to \mathbb{B}$ beschriftet. Es gibt genau einen Knoten mit genau einem Vorgänger und ohne Nachfolger. Dieser wird *Ausgang* von S_n genannt. Er ist mit der Identitätsfunktion $1_{\mathbb{B}} : \mathbb{B} \to \mathbb{B}$ beschriftet.

(e) Die mit $\wedge$, $\vee$ oder $\neg$ beschrifteten Knoten heißen auch *innere Knoten*. □

Die Anzahl der Vorgänger eines Knotens eines Schaltkreises ist ≤ 2. Die Anzahl der Nachfolger ist im allgemeinen nicht beschränkt. Das bedeutet, daß das von einem Knoten berechnete Ergebnis von beliebig vielen anderen Knoten benutzt werden darf. Allgemeiner können auch Schaltkreise mit n Eingängen und m Ausgängen definiert werden. Dann gibt es in (d) m Knoten ohne Nachfolger, die entsprechend mit Injektionsfunktionen $i_j : \mathbb{B} \to \mathbb{B}^m$, $j = 1, \ldots, m$, beschriftet sind.

Definition 10.5.2 Es sei S_n ein Schaltkreis mit n Eingängen. Wir definieren die *durch den Schaltkreis* S_n *berechnete Funktion* $f_{S_n} : \mathbb{B}^n \to \mathbb{B}$ dadurch, daß wir zunächst für eine Eingabe $x \in \mathbb{B}^n$ die Ausgaben der einzelnen Knoten von S_n wie folgt bestimmen:

(a) Ein mit π_i, $i = 1, \ldots, n$, beschrifteter Knoten gibt die i-te Komponente von x aus.

(b) Ein mit $\wedge$, $\vee$ oder $\neg$ beschrifteter Knoten v gibt den Wert aus, den man durch Anwendung der entsprechenden Funktion auf die Ausgaben der Knoten anwendet, die Vorgänger von v sind.

(c) Der mit $1_{\mathbb{B}}$ beschriftete Knoten gibt seine Eingabe aus.

Der Wert $f_{S_n}(x)$ ergibt sich dann als die Ausgabe des mit $1_{\mathbb{B}}$ beschrifteten Knotens. □

Bei m Ausgängen liefert ein mit i_j beschrifteter Knoten ein m-Tupel, dessen j-te Komponente seine Eingabe ist, während die anderen Komponenten 0 sind. Das Gesamtergebnis der Funktion $\mathbb{B}^n \to \mathbb{B}^m$ ergibt sich dann durch die $\vee$-Verknüpfung der verschiedenen Ausgaben dieser Knoten.

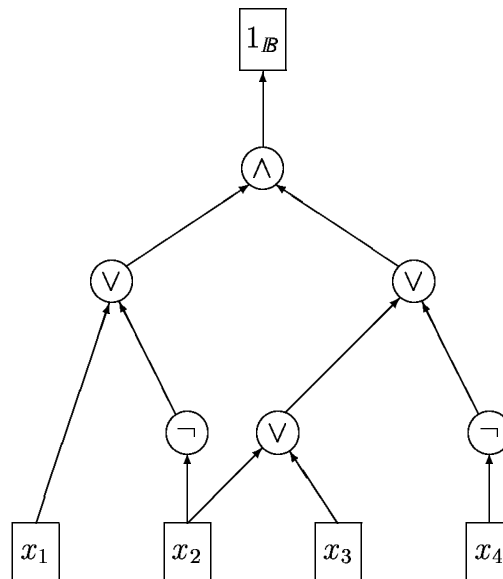
Definition 10.5.3 (a) Liegt eine Abbildung vor, die jedem $n \in \mathbb{N}$ einen Schaltkreis S_n zuordnet, so sprechen wir von der *Schaltkreisfamilie* $\mathcal{S} = \{S_n \mid n \in \mathbb{N}\}$.

(b) Die *durch eine Schaltkreisfamilie* $\mathcal{S}$ *berechnete Funktion* $f_{\mathcal{S}} : \mathbb{B}^+ \to \mathbb{B}$ ist für $w \in \mathbb{B}^+$, $|w| = n$, durch $f_{\mathcal{S}}(w) = f_{S_n}(w)$ gegeben.

(c) Die *von einer Schaltkreisfamilie* $\mathcal{S}$ *erkannte Sprache* ist

$$L(\mathcal{S}) = \{w \in \mathbb{B}^+ \mid f_{\mathcal{S}}(w) = 1\}. \quad \square$$

Beispiel 10.5.1 Wir geben für die Funktion $f_4 : \mathbb{B}^4 \to \mathbb{B}$, die durch den logischen Ausdruck $\alpha = (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3 \vee \neg x_4)$ gekennzeichnet ist, den sie berechnenden Schaltkreis an:



Definition 10.5.4 Es sei S ein Schaltkreis.

(a) Die *Komplexität* $C(S)$ *von* S ist die Anzahl seiner Knoten.

(b) Die *Tiefe* $d_S(v)$ *eines Knotens* v *in* S ist die maximale Länge eines Weges von einem Eingang zu v. Die Gesamtheit der Knoten einer Tiefe d bezeichnen wir als *Schicht der Tiefe* d oder kurz als *Schicht* d.

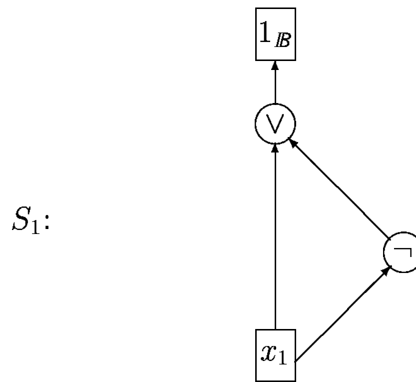
(c) Die *Tiefe* $D(S)$ *von* S ist das Maximum der Tiefen seiner Knoten. □

In Beispiel 10.5.1 haben die mit $\neg$ beschrifteten Knoten die Tiefe 1, der mit $\wedge$ beschriftete Knoten die Tiefe 3. Die Tiefe des Schaltkreises ist $D(S) = 4$. Seine Komplexität ist $C(S) = 11$.

Beispiel 10.5.2 Wir wollen eine Schaltkreisfamilie $\{S_n \mid n \in \mathbb{N}\}$ angeben, die die Sprache PALINDROM aller Palindrome über $\mathbb{B}$ erkennt. Die zugehörigen Funktionen f_{S_n}, $n = 1, \ldots, n$, sind durch

$$f_{S_n}(x_1 \ldots x_n) = 1 \iff x_1 \ldots x_n \text{ ist ein Palindrom}$$

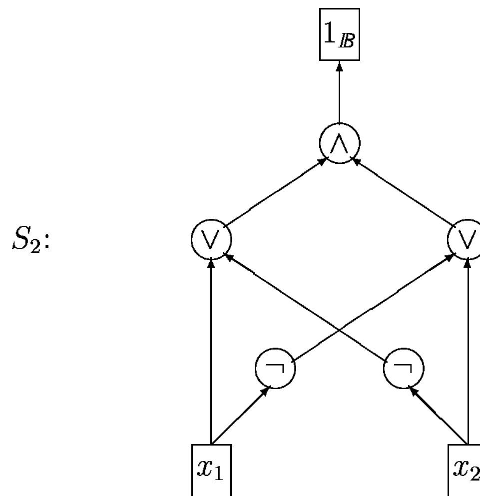
gegeben. Für $n = 1$ erhalten wir den Schaltkreis



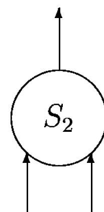
Für $n = 2$ gilt $f_{S_2}(x_1, x_2) = 1$ genau dann, wenn $x_1 = x_2$ gilt. Dies ist gleichwertig damit, daß der Ausdruck

$$(x_1 \leftrightarrow x_2) = (x_1 \to x_2) \wedge (x_2 \to x_1) = (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee x_1)$$

den Wert 1 liefert. Dafür können wir den folgenden Schaltkreis konstruieren:

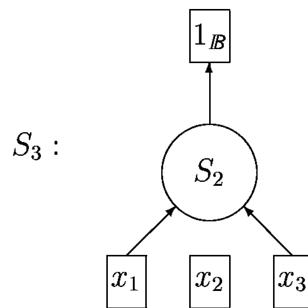


Wir bezeichnen mit

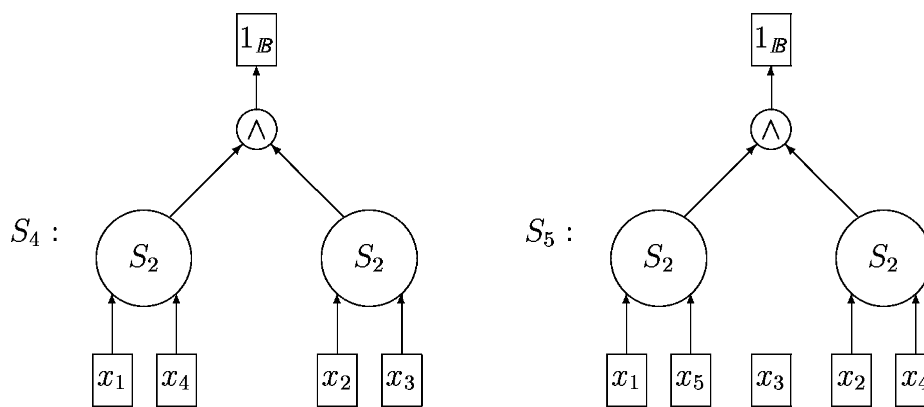


den Schaltkreis S_2, bei dem wir jedoch die Variablen x_1 und x_2 durch zwei eingehende Kanten und den Ausgang durch eine ausgehende Kante ersetzt haben. Dann können

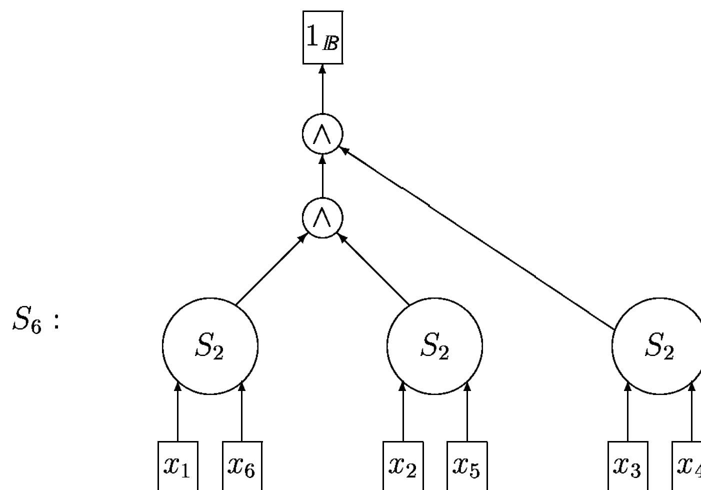
wir für $n = 3$ den Schaltkreis



angeben. Für $n = 4$ und $n = 5$ erhalten wir die Schaltkreise



und für $n = 6$ dann



Diese Konstruktion kann entsprechend fortgesetzt werden. Bei jedem zweiten Schritt wird ein Teilgraph S_2 hinzugefügt. Für die Tiefen der jeweiligen Schaltkreise erhalten wir

$$D(S_2) = 4,\ D(S_3) = 4,\ D(S_4) = 5,\ D(S_5) = 5,\ D(S_6) = 6.$$

Allgemein gelten für $k \in \mathbb{N}$ die Beziehungen

$$D(S_{2k+1}) = D(S_{2k}) \text{ und } D(S_{2k+2}) = D(S_{2k}) + 1.$$

Daraus folgt

$$D(S_n) = \left\lfloor \frac{n}{2} \right\rfloor + 3$$

für alle $n \in \mathbb{N}$, $n \geq 2$. Für die jeweiligen Komplexitäten ergibt sich

$$C(S_2) = 8,\ C(S_3) = 9,\ C(S_4) = 16,\ C(S_5) = 17,\ C(S_6) = 24.$$

Wir sehen, daß für $k \in \mathbb{N}$ die Gleichungen

$$C(S_{2k+1}) = C(S_{2k}) + 1 \text{ und } C(S_{2k+2}) = C(S_{2k}) + 8$$

erfüllt sind. Somit gilt

$$C(S_{2n}) = C(S_{2n+1}) - 1 = 8 + (n-1) \cdot 8$$

für alle $n \in \mathbb{N}$. Daß die Schaltkreisfamilie $\{S_n \mid n \in \mathbb{N}\}$ die Sprache PALINDROM erkennt, ist offensichtlich. □

Aus unserer Konstruktion in Beispiel 10.5.2 folgt ein Algorithmus, der für alle n einen Schaltkreis S_n konstruiert. Wenn man einen solchen Algorithmus nicht verlangt, ist der Begriff einer durch einen Schaltkreis erkannten Sprache aus Definition 10.5.3 zu weitgehend. Es können in diesem Fall sogar nicht rekursiv-aufzählbare Sprachen erkannt werden. Es sei nämlich $A \subset \mathbb{N}$ eine nicht rekursiv-aufzählbare Menge. Dann ist erst recht die Sprache $L_A = \{w \in \mathbb{B}^* \mid |w| \in A\}$ nicht rekursiv-aufzählbar. L_A wird von der trivialen Schaltkreisfamilie erkannt, bei der S_n für alle $n \in A$ die Konstante 1 und für alle $n \notin A$ die Konstante 0 liefert. Wir können jedoch für ein beliebiges n nicht algorithmisch feststellen, welcher Fall zutreffend ist.

Um eine Einschränkung der Schaltkreisfamilien zu erhalten, verlangen wir, daß sie durch Turingmaschinen berechnet werden. Von solchen Schaltkreisfamilien erkannte Sprachen sind sicher alle rekursiv. Wir wollen jedoch einen Zusammenhang zu den von PRAMs erkannten Sprachfamilien herstellen. Es sollten dabei keine zu großen Unterschiede zwischen den Zeitkomplexitäten der beiden Modelle bestehen. Damit das der Fall ist, darf die Konstruktion einer Familie von Schaltkreisen durch eine Turingmaschine weder zu einfach noch zu kompliziert sein. Dies geschieht dadurch, daß bei dieser Konstruktion in gewisser Weise nur beschränkter Platz verwendet wird. Die Definition der Raumkomplexität aus Definition 9.1 ist dafür nicht direkt geeignet. Wir betrachten vielmehr zunächst sogenannte EA-Turingmaschinen, für die die Raumkomplexität speziell definiert wird.

Definition 10.5.5 Eine EA-*Turingmaschine* ist eine Mehrband-Turingmaschine, die ein *Eingabeband*, ein *Ausgabeband* und mehrere *Arbeitsbänder* hat. Das Eingabeband kann nur gelesen werden, das Ausgabeband nur beschrieben. Das Ausgabeband hat einen Kopf, der nur schreiben und sich nach rechts bewegen kann. Das Wort, daß sich in einer Endkonfiguration auf dem Ausgabeband befindet, ist die *Ausgabe* der EA-Turingmaschine.

Bei der Bestimmumg der *Raumkomplexität* einer EA-Turingmaschine werden nur die Arbeitsbänder berücksichtigt. □

Definition 10.5.6 Eine Familie $\{S_n \mid n \in \mathbb{N}\}$ von Schaltkreisen heißt *uniform*, wenn es eine EA-Turingmaschine mit der Raumkomplexität $O(\log C(S_n))$ gibt, die jedem $n \in \mathbb{N}$ (Eingabe 1^n) eine Kodierung des Schaltkreises S_n zuordnet.

Beispiel 10.5.3 Die Schaltkreisfamilie aus Beispiel 10.5.2 ist uniform, denn es wird in jedem Schritt der Konstruktion höchstens ein S_2-Schaltkreis unter Verwendung eines $\wedge$-Operators mit dem bereits konstruierten und auf dem Ausgabeband stehenden Schaltkreis verknüpft. Es werden also der Reihe nach die S_2-Schaltkreise für (x_1, x_n), (x_2, x_{n-1}), (x_3, x_{n-2}), $\ldots$, zusammen mit den $\wedge$-Operatoren, auf das Ausgabeband geschrieben. Es ist klar, daß in jedem Schritt $O(\log n)$ Platz auf den Arbeitsbändern reicht, da von Schritt zu Schritt nur die aktuellen Indizes der beteiligten Eingänge (in binärer Notation) auf den Arbeitsbändern niedergelegt sein müssen. □

Wir zeigen zunächst die Simulation eines einzelnen Schaltkreises durch eine PRAM.

Satz 10.5.1 Es sei $f : \mathbb{B}^n \to \mathbb{B}$ eine Boolesche Funktion, die durch einen Schaltkreis S_n der Tiefe d und der Komplexität c berechnet wird. Dann kann f durch eine COMMON CRCW PRAM mit $O(c^2)$ Prozessoren in $O(d)$ Schritten simuliert werden.

Beweis: Für die Simulierung von S_n werden im gemeinsamen Speicher der PRAM die Register $M_1, \ldots, M_c$ für die Knoten des Schaltkreises reserviert. Wir nehmen dabei an, daß $M_1, \ldots, M_n$ für die Eingänge und M_c für den Ausgang steht. Für jede Kante (i, j) des Schaltkreises wird ein eigener Prozessor $P_{i,j}$ eingesetzt, außerdem auch für jeden Eingang (dies ist nicht nötig, wenn es mehr Kanten als Eingänge gibt). Dies sind $O(c^2)$ Prozessoren. Der Schaltkreis besteht aus $d + 1$ Schichten. In der Schicht 0 stehen die Eingänge, die Schichten 1 bis $d - 1$ entsprechen den inneren Knoten, und die Schicht d stellt den Ausgang dar. Jeder Prozessor $P_{i,j}$ kennt natürlich die Beschriftung von j (oder genauer die Beschriftung des j entsprechenden Knotens) im Schaltkreis S_n und auch die Schicht dieses Knotens. Die COMMON CRCW PRAM arbeitet nach dem folgenden Algorithmus:

{Eingabe: $x_1, \ldots, x_n \in \mathbb{B}$
Ausgabe: $f(x_1, \ldots, x_n)$ im Register M_c
for alle Kanten (i, j) **pardo**
    {j mit $\wedge$ beschriftet : $M_j := 1$;
     j mit $\vee$ beschriftet : $M_j := 0$}
    od;
for $i = 1$ **to** n **pardo**
    $M_i := x_i$ {Schicht 0}
    od;
for $k := 1$ **to** d **do**
    for alle Kanten (i, j) **pardo**
        if Schicht von j gleich k **then**
            {j mit $\neg$ beschriftet : **if** $M_i = 0$ **then** $M_j := 1$ **else** $M_j := 0$
             j mit $\vee$ beschriftet : **if** $M_i = 1$ **then** $M_j := 1$
             j mit $\wedge$ beschriftet : **if** $M_i = 0$ **then** $M_j := 0$
             j mit $1_{\mathbb{B}}$ beschriftet : $M_j := M_i$}
        od
    od

Die angegebene Anzahl von Prozessoren reicht offenbar für die Durchführung der Schritte aus. Die erste **for**-Schleife ist für die richtige Simulation der letzten Schleife notwendig. Sie braucht nur einen parallelen Schritt, da die Prozessoren $P_{i,j}$ die Beschriftung des Knotens j a priori kennen. In der zweiten Schleife wird in einem parallelen Schritt die Eingabe an die Eingänge übergeben, also an die Knoten der Schicht 0. In der letzten Schleife erfolgt Schicht für Schicht in d parallelen Schritten die Simulation. Die vier Fälle $\neg$, $\vee$, $\wedge$ und $1_{\mathbb{B}}$ werden natürlich wieder in einem parallelen Schritt ausgeführt. Wir sehen sofort, daß $\neg$ richtig behandelt wird. Für $\vee$ ist in der ersten Schleife der Wert 0 in das Register M_j gesetzt worden. Dieser Wert muß geändert werden, wenn für mindestens ein i mit einer Kante (i, j) in einer vorhergehenden Schicht $M_i = 1$ gesetzt wurde. Genau die entsprechende Prozessoren $P_{i,j}$ schreiben in das Register M_j, und zwar alle denselben Wert 1. Dies ist bei einer COMMON CRCW PRAM eine erlaubte Aktion. In analoger Weise liefert auch der Fall $\wedge$ das richtige Ergebnis. Es ist klar, daß der Fall $1_{\mathbb{B}}$ genau einmal auftritt, und zwar in der Schicht d, wenn genau ein Prozessor $P_{i,c}$ aktiv ist und dabei das Ergebnis in das Register M_c schreibt. $\square$

Allgemein kann eine Familie von Schaltkreisen nicht durch eine PRAM simuliert werden. Dies wird sofort klar, wenn wir daran denken, daß es Schaltkreisfamilien gibt, die nicht rekursiv-aufzählbare Sprachen erkennen. Satz 10.5.1 würde verschiedene Programme für verschiedene Eingabegrößen n liefern. Unsere PRAM-Algorithmen sind jedoch für beliebige Werte von n konzipiert worden. Es können sich also beliebige Schaltkreisfamilien und PRAMs nicht entsprechen. Daher beschränkt man sich zum einen auf uniforme Schaltkreisfamilien. Zum anderen betrachtet man, etwas allgemeiner als bisher, uniforme PRAM-Algorithmen. Man sagt, daß ein PRAM-Algorithmus uniform ist, wenn es eine EA-Turingmaschine gibt, die für jedes n in logarithmischem Raum eine zugehörige PRAM für Eingaben der Länge n konstruiert. Alle unsere zuvor betrachteten PRAM-Programme sind uniform, da sie alle über n parametrisiert waren. Man kann dann zeigen, daß jede uniforme Schaltkreisfamilie der Tiefe $D(n)$ und der Komplexität $C(n)$ von einem uniformen COMMON CRCW PRAM-Algorithmus simuliert werden kann, und zwar in einer Zeit polynomial in $D(n)$ und mit einer Anzahl von Prozessoren polynomial in $C(n)$. Umgekehrt kann jeder uniforme COMMON CRCW PRAM-Algorithmus (also auch jeder wie zuvor definierte COMMON CRCW PRAM-Algorithmus), der eine Zeitkomplexität $t(n)$ hat und $p(n)$ Prozessoren aktiviert, durch eine uniforme Schaltkreisfamilie mit einer Tiefe polynomial in $t(n)$ und der Komplexität $p(n)$ simuliert werden.

Zum Abschluß dieses Abschnitts wollen wir kurz auf die These der parallelen Berechenbarkeit zu sprechen kommen. Wir bezeichnen mit P-TIME$(f(n))$ die Familie der Sprachen, die durch ein paralleles Berechnungsmodell $\mathcal{P}$ in der Zeit $O(f(n))$, und weiter mit DSPACE$(g(n))$ die Familie der Sprachen, die durch deterministische Turingmaschinen mit Raumkomplexität $O(g(n))$ entschieden werden. Die *parallele Berechenbarkeitsthese* gilt für das Modell $\mathcal{P}$, wenn Polynome p und q existieren, so daß

$$\text{P-TIME}(f(n)) \subset \text{DSPACE}(p(f(n))) \text{ und } \text{DSPACE}(g(n)) \subset \text{P-TIME}(q(g(n)))$$

ist. Die These behauptet also, daß parallele Zeit in polynomialem Zusammenhang mit sequentiellem Raum steht.

Es existieren parallele Berechnungsmodelle, für die die These nicht gilt, jedoch ist sie sowohl für PRAMs als auch für uniforme Schaltkreisfamilien erfüllt.

10.6 Netzwerkmodelle

Außer den PRAM-Modellen gibt es noch viele weitere Modelle von Parallelrechnern. In diesem Abschnitt wollen wir drei solcher Netzwerkmodelle vorstellen. Wir definieren zunächst den allgemeinen Begriff eines Netzwerkes.

Definition 10.6.1 Ein *Netzwerk* ist ein Graph $G = (V, E)$, wobei jeder Knoten $i \in V$ einen Prozessor und jede Kante $(i, j) \in E$ eine Kommunikationsverbindung zwischen den Prozessoren i und j darstellt. Jeder Prozessor besitzt seinen eigenen lokalen Speicher, einen gemeinsamen Speicher gibt es nicht. □

Diese Definition ist sicher etwas informal, da unter anderem der Begriff eines Prozessors nicht exakt festgelegt ist. Man könnte dabei jedoch an eine RAM denken. Die Operationen eines Netzwerks können synchron oder auch asynchron sein. Man spricht auch davon, daß durch den Graphen des Netzwerkes die *Topologie* der Prozessoren festgelegt wird. Um Algorithmen für Netzwerkmodelle angeben zu können, brauchen wir zusätzliche Programmkonstrukte, um die Kommunikation zwischen den Prozessoren zu beschreiben. Wir verwenden

$$\textbf{send}(x, i) \text{ und } \textbf{receive}(y, j).$$

Ein Prozessor, der die erste Instruktion ausführt, sendet eine Kopie von x (oder eine Kopie des Inhalts des Registers x) an den Prozessor P_i und bearbeitet dann sofort die nächste Anweisung. Führt P die Instruktion **receive**(y, j) durch, so wartet er, bis er Daten vom Prozessor P_j erhalten hat. Die Daten werden im Register y gespeichert, anschließend wird die nächste Anweisung bearbeitet.

Definition 10.6.2 Ein Netzwerk heißt *Ring*, wenn es Prozessoren P_i, $i = 1, \ldots, p$, besitzt, wobei P_i für $i \neq 1$ und $i \neq p$ genau mit P_{i-1} und P_{i+1}, für $i = 1$ mit P_p und P_2 sowie für $i = p$ mit P_{p-1} und P_1 kommuniziert. □

Beispiel 10.6.1 Für $n \in \mathbb{N}$ sei A eine $n \times n$-Matrix und x ein n-dimensionaler Vektor. Wir wollen das Produkt $y = A \cdot x$ mit Hilfe eines Rings von p Prozessoren berechnen. Zur Vereinfachung nehmen wir an, daß p die Zahl n teilt. Wir setzen $r = \frac{n}{p}$. Dann werden A und x in p Blöcke aufgeteilt, und zwar

$$A = (A_1, \ldots, A_p) \text{ und } x = (x_1, \ldots, x_p).$$

Dabei ist jedes A_i eine $n \times r$-Matrix und jedes x_i ein r-dimensionaler Vektor. Offenbar gilt

$$y = Ax = A_1x_1 + \ldots + A_px_p.$$

Die Idee des Algorithmus ist, daß der Prozessor P_i den Wert $y_i = A_ix_i$ berechnet und anschließend die Werte y_i zum Nachbarn mit dem höheren Index weitergegeben und dabei aufaddiert werden. Jeder Prozessor arbeitet gemäß dem folgenden Programm:

{Eingabe: der Prozessorindex i und die Anzahl p der Prozessoren
die $n \times r$-Matrix $A_i = A(1 : n, (i-1)r + 1 : ir)$, wobei $r = \frac{n}{p}$ gilt
der r-dimensionale Vektor $x_i = x((i-1)r + 1 : ir)$

Ausgabe: P_i berechnet $y = A_1x_1 + \ldots + A_ix_i$ und übergibt das Ergebnis an P_{i+1} bzw. (für $i = p$) an P_1. Am Ende hält P_1 das Ergebnis}

begin
  $z_i := A_ix_i$;
  if $i = 1$ **then** $y := 0$
    else receive$(y, i-1)$;
  $y := y + z_i$;
  send$(y, i+1)$; {bzw. **send**$(y, 1)$ für $i = p$}
  if $i = 1$ **then receive**(y, p)
end

Jeder Prozessor berechnet $z_i = A_ix_i$. Im nächsten Schritt lädt der Prozessor P_1 den Wert 0 in das Register y, während die anderen Prozessoren darauf warten, von ihrem Nachbarn P_{i-1} Daten zu erhalten. P_1 schreibt den Wert z_1 in das Register y und sendet z_1 an P_2, danach wartet P_1 auf eine Eingabe. P_2 kann nun $y := y + z_2 = z_1 + z_2$ berechnen, schickt diesen Wert an P_3 und hat seine Arbeit beendet. Dies Vorgehen wird fortgesetzt, bis schließlich P_p den Wert $y = z_1 + \ldots + z_p = Ax$ an P_1 übergibt und stoppt. P_1 empfängt das Ergebnis der Rechnung.

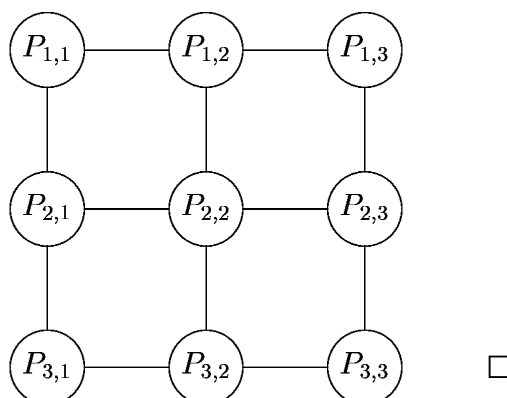
Es ist klar, daß die Prozessoren asynchron arbeiten. Die Rechenzeit des ersten Schrittes ist für alle P_i von der Ordnung $O(rn) = O(\frac{n^2}{p})$. Andererseits muß jedoch zum Beispiel der Prozessor P_1 darauf warten, das Ergebnis $A_1x_1 + \ldots + A_px_p$ zu empfangen. Die gesamte Kommunikationszeit ist von der Ordnung $O(p \cdot c(n))$, wobei $c(n)$ die Zeit ist, n Zahlen von einem Prozessor zum nächsten zu senden. Somit benötigen wir zur Berechnung des Produktes Ax die Zeit $O(\frac{n^2}{p} + pc(n))$. Die Geschwindigkeit der Kommunikation ist also sehr entscheidend. Für $n = p$ ist die Rechenzeit zwar von der Ordnung $O(1)$, aber die Kommunikation kann eventuell sehr aufwendig sein. □

Definition 10.6.3 Ein Netzwerk heißt *zweidimensionales Gitter*, wenn $p = n^2$ Prozessoren $P_{i,j}$, $1 \le i, j \le n$, $n \in \mathbb{N}$, auf einem $n \times n$-Gitter so angeordnet sind, daß der Prozessor $P_{i,j}$ mit den Prozessoren

$$P_{i+1,j},\ P_{i-1,j},\ P_{i,j+1} \text{ und } P_{i,j-1}$$

kommuniziert, falls sie existieren. □

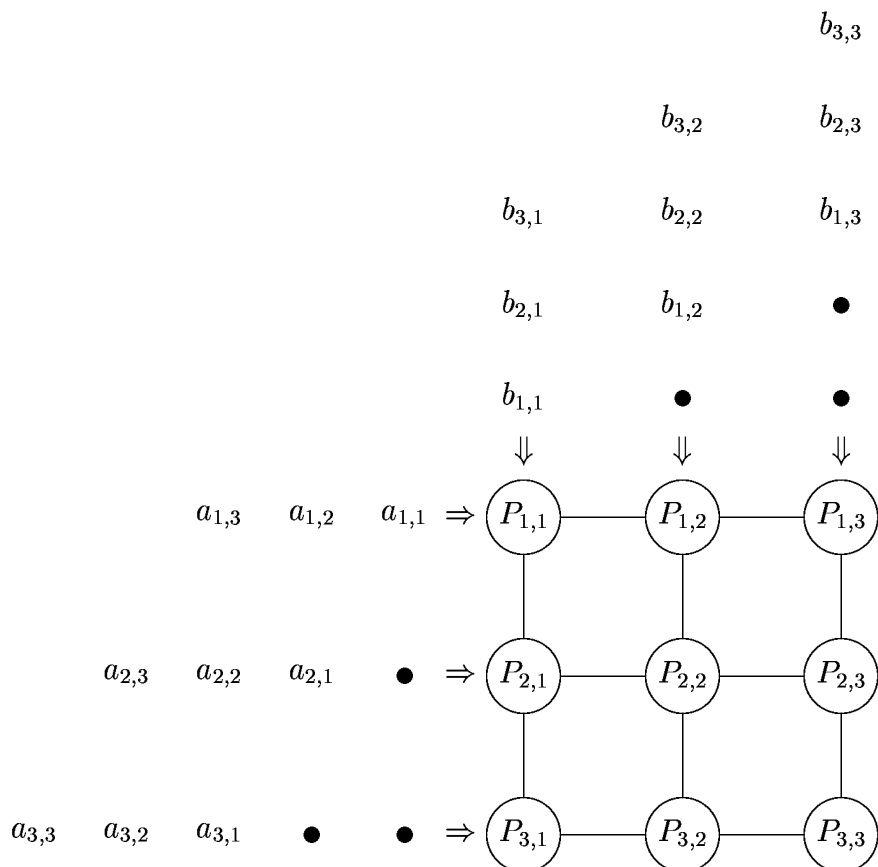
Beispiel 10.6.2 Wir betrachten ein 3×3-Gitter, das $3^2 = 9$ Prozessoren besitzt.



□

Der Vorteil dieser Struktur ist die leichte Erweiterbarkeit. Außerdem ist sie auch vielen Anwendungen gut angepaßt, was durch das folgende Beispiel deutlich wird.

Beispiel 10.6.3 Gegeben seien zwei $n \times n$-Matrizen $A = (a_{i,j})$ und $B = (b_{i,j})$, $1 \leq i,j \leq n$, $n \in \mathbb{N}$. Wir wollen das Matrizenprodukt $C = AB$ auf einem $n \times n$-Gitter berechnen. Das soll mit Hilfe eines sogenannten *systolischen Algorithmus* geschehen. Dabei werden die Zeilen von A synchron in schräg verzögerter Weise in die linke Seite und die Spalten von B synchron in schräg verzögerter Weise in die obere Seite des Gitters eingegeben, wie es in der nächsten Abbildung für den Fall $n = 3$ dargestellt ist. Dies bedeutet genauer, daß die erste Zeile von A an den ersten Prozessor der ersten Zeile des Gitters, also $P_{1,1}$, geschickt wird, also zunächst das Element $a_{1,1}$, einen Takt später $a_{1,2}$, usw., und schließlich nach n Takten $a_{1,n}$. Die zweite Zeile von A wird einen Takt verzögert an die zweite Zeile des Gitters gesendet. Im Takt 2 erhält also der Prozessor $P_{2,1}$ das Element $a_{2,1}$, im Takt 3 das Element $a_{2,2}$, usw., im Takt $n+1$ dann $a_{2,n}$. Die Eingabe der n-ten Zeile der Matrix A beginnt im Takt n. In analoger Weise werden die Spalten der Matrix B an die obere Zeile des Gitters geschickt.



Die Eingabezeilen bzw. -spalten werden taktweise zu jedem Prozessor der entsprechenden Zeile oder Spalte durchgereicht. Jeder Prozessor $P_{i,j}$ besitzt lokale Variablen $A_{i,j}$, $B_{i,j}$ und $C_{i,j}$. Zu Beginn sind alle Variablen auf 0 gesetzt. Die Variablen $A_{i,j}$ und $B_{i,j}$ dienen zur Aufnahme der durchgereichten Werte. Jeder Prozessor $P_{i,j}$ erhält für $j \neq 1$ von seinem linken Nachbarn $P_{i,j-1}$ den Wert $A_{i,l}$ bzw. für $j = 1$ von der linken

Eingabe (i-te Zeile). Er erhält für $i \neq 1$ von seinem oberen Nachbarn $P_{i-1,j}$ den Wert $B_{l,j}$ bzw. für $i = 1$ von der oberen Eingabe (j-te Spalte). Es wird dann jeweils

$$C_{i,j} := C_{i,j} + A_{i,l} \cdot B_{l,j}$$

berechnet. An den rechten bzw. unteren Nachbarn werden die Werte $A_{i,l}$ bzw. $B_{l,j}$ unverändert weitergereicht. Nach $2n$ Schritten enthält jeder Prozessor $P_{i,j}$ in der Variablen $C_{i,j}$ den richtigen Wert

$$c_{i,j} = a_{i,1}b_{1,j} + \ldots + a_{i,n}b_{n,j}.$$

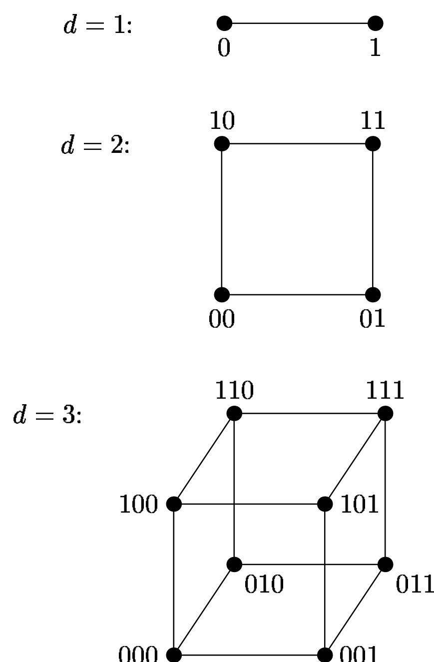
Wir sehen also, daß dieser Algorithmus der Matrizenmultiplikation mit n^2 Prozessoren einen Zeitbedarf von $O(n)$ hat, wohingegen die Standardalgorithmen zur Matrizenmultiplikation $O(n^3)$ Schritte benötigen. □

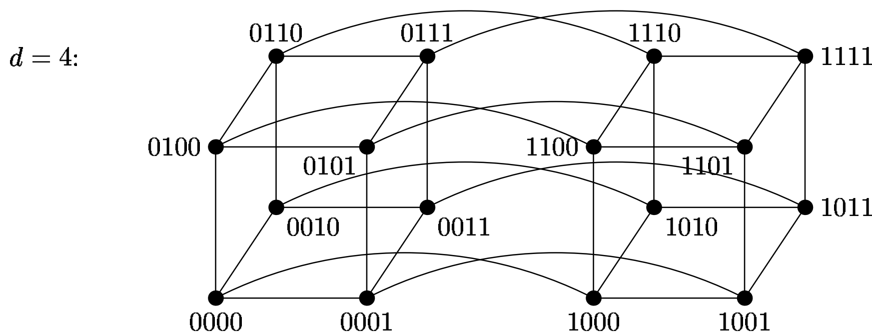
Systolische Algorithmen arbeiten vollständig synchron, wobei zu jedem Zeittakt ein Prozessor Daten von einigen Nachbarn empfängt, einige Berechnungen durchführt und schließlich Daten zu einigen seiner Nachbarn sendet.

Definition 10.6.4 Es sei $d \in \mathbb{N}$. Ein *Hyperwürfel* besteht aus $p = 2^d$ Prozessoren, die gemäß einem d-dimensionalen Booleschen Würfel wie folgt verbunden sind: Es sei $i_{d-1}i_{d-2}\ldots i_0$ die Binärdarstellung von i, $0 \leq i \leq p-1$. Dann ist der Prozessor P_i für alle j, $0 \leq j \leq d-1$, mit dem Prozessor $P_{i^{(j)}}$ verbunden, wobei $i^{(j)} = i_{d-1}\ldots i_{j+1}\bar{i}_j i_{j-1}\ldots i_0$ gilt mit $\bar{i}_j = 1 - i_j$. □

Zwei Prozessoren sind genau dann miteinander verbunden, wenn sich ihre Indizes in genau einer Bitposition unterscheiden.

Ein Hyperwürfel hat eine rekursive Struktur. Wir können einen d-dimensionalen Würfel zu einem $(d+1)$-dimensionalen Würfel erweitern, indem wir entsprechende Prozessoren von zwei d-dimensionalen Würfeln miteinander verbinden. Der eine Würfel erhält das zusätzliche erste Bit 0, der andere das zusätzliche Bit 1. Für die ersten vier Dimensionen erhalten wir die folgenden Darstellungen.





Der *Durchmesser* eines Hyperwürfels, d.h. die maximale Anzahl von Verbindungen, die notwendig ist, um von einem Prozessor zu einem beliebigen anderen zu gelangen, ist offensichtlich $\log p = d$. Entsprechend benötigt man höchstens d Schritte, Daten von einem Prozessor zu irgendeinem anderen Prozessor des Hyperwürfels durchzureichen. Hyperwürfel sind als Berechnungsmodelle beliebt wegen ihrer Regularität, ihres kleinen Durchmessers, wegen ihrer interessanten graphentheoretischen Eigenschaften und der Möglichkeit, mit ihnen viele Berechnungen schnell und einfach durchzuführen.

Beispiel 10.6.4 Für $n \in I\!N$, $n = 2^d$, werde der n-dimensionale Vektor $A = (a_i)$, $0 \leq i \leq n-1$, so im d-dimensionalen Hyperwürfel gespeichert, daß der Prozessor P_i den Wert a_i enthält. Es soll die Summe $S = \sum_{i=0}^{n-1} a_i$ berechnet werden.

Der Algorithmus zur Berechnung von S besteht aus d Iterationen. In der ersten Iteration wird die Summe der Paare von Elementen derjenigen Prozessoren bestimmt, deren Index sich im ersten Bit unterscheidet. Diese Summe wird in dem $(d-1)$-dimensionalen Teilwürfel gespeichert, bei dem das erste Bit der Indizes der Prozessoren 0 ist. Die weiteren Iterationen werden nach demselben Schema fortgesetzt.

Es bedeute $l^{(k)}$ der Index, der sich aus dem Index l ergibt, indem das k-te Bit von l durch sein Komplement ersetzt wird. Ist $A(i)$ eine lokale Variable des Prozessors P_i zur Berechnung des Wertes a_i, so muß P_i nach den obigen Überlegungen eine Instruktion $A(i) := A(i) + A(i^{(l)})$ durchführen. Dies erfordert zwei Teilschritte. Im ersten erhält P_i von $P_{i^{(l)}}$ den Wert der Variablen $A_{i^{(l)}}$. Im zweiten erfolgt dann die Addition, und das Ergebnis wird in $A(i)$ gespeichert.

{Eingabe: $n = 2^d$ Elemente a_i, $0 \leq i \leq n-1$, werden in den lokalen Variablen $A(i)$
der Prozessoren P_i gespeichert

Ausgabe: $S = \sum_{i=0}^{n-1} a_i$, gespeichert in der Variablen $A(0)$ von P_0}

{Algorithmus für P_i:}
begin
  for $k = d-1$ **to** 0 **do**
    if $0 \leq i \leq 2^k - 1$ **then** $A(i) := A(i) + A(i^{(k)})$
end.

Daß der Algorithmus das Gewünschte leistet, machen wir uns an dem Beispiel $n = 8 = 2^3$ klar. In der ersten Iteration der **for**-Schleife (betrachte den Würfel für $d = 3$ in der obigen bildlichen Darstellung) werden die Summen

$$A(0) = A(0)+A(4),\ A(1) = A(1)+A(5),\ A(2) = A(2)+A(6) \text{ und } A(3) = A(3)+A(7)$$

berechnet und in den Prozessoren P_0, P_1, P_2 bzw. P_3 gespeichert. In der nächsten Iteration wird

$$A(0) = A(0)+A(2) = (a_0+a_4)+(a_2+a_6) \text{ und } A(1) = A(1)+A(3) = (a_1+a_5)+(a_3+a_7)$$

berechnet. Es ist klar, daß durch

$$A(0) = A(0) + A(1)$$

nach der letzten Iteration sich die Summe S in $A(0)$ befindet.

Ersichtlich benötigt der Algorithmus $O(d) = O(\log n)$ Schritte. □

Beispiel 10.6.5 Für $n \in \mathbb{N}$, $n = 2^q$, seinen zwei $n \times n$-Matrizen $A = (a_{i,l})$ und $B = (b_{l,j})$, $0 \leq i,j,l \leq n-1$, gegeben. Ihr Produkt $C = AB$ soll mit Hilfe eines synchronen Hyperwürfels mit $p = n^3 = 2^{3q}$ Prozessoren berechnet werden. Wir indizieren die Prozessoren durch Tripel (l,i,j), wobei $P_{l,i,j}$ den Prozessor P_r mit $r = ln^2 + in + j$ darstellt. Wenn wir r als Binärzahl auffassen, entsprechen die ersten q Bits dem Index l, die nächsten q Bits dem Index i und die letzten q Bits dem Index j. Wir erhalten also einen $3q$-dimensionalen Hyperwürfel. Wenn wir ein Paar der Indizes l, i, j festhalten, ergibt sich ein q-dimensionaler Teilwürfel.

Die Matrix A wird in dem Teilwürfel gespeichert, der durch die Prozessoren $P_{l,i,0}$, $0 \leq l,i \leq n-1$, gegeben ist, und zwar steht das Element $a_{l,i}$ im Prozessor $P_{l,i,0}$. Entsprechend wird B in dem durch $P_{l,0,j}$ bestimmten Teilwürfel gespeichert, wobei $P_{l,0,j}$ den Wert $b_{l,j}$ enthält.

Zur Berechnung von $c_{i.j} = \sum_{l=0}^{n-1} a_{i,l} b_{l,j}$, $0 \leq i,j \leq n-1$, sind drei Stadien zu durchlaufen.

(1) Die Eingabe wird weiter so verteilt, daß jeder Prozessor $P_{l,i,j}$ die Werte $a_{i,l}$ und $b_{l,j}$, $0 \leq l,i,j \leq n-1$, enthält.
(2) Für alle l,i,j, $0 \leq l,i,j \leq n-1$, berechnet der Prozessor $P_{l,i,j}$ das Produkt $c'_{l,i,j} = a_{i,l} b_{l,j}$.
(3) Für alle i,j, $0 \leq i,j \leq n-1$, berechnen die Prozessoren $P_{l,i,j}$, $0 \leq l \leq n-1$, gemeinsam die Summe $c_{i,j} = \sum_{l=0}^{n-1} c'_{l,i,j}$.

Die Implementierung des ersten Stadiums erfolgt in zwei Teilstadien. Im ersten Teilstadium wird für jedes i und l der Wert $a_{i,l}$ vom Prozessor $P_{l,i,0}$ an die Prozessoren $P_{l,i,j}$ für $0 \leq j \leq n-1$ verteilt. Da für festes i und l die Prozessoren $P_{l,i,j}$ einen q-dimensionalen Teilwürfel bilden, kann diese Verteilung in $\log 2^q = q$ Schritten durchgeführt werden. In analoger Weise wird $b_{l,j}$ an die Prozessoren $P_{l,i,j}$ für $0 \leq i \leq n-1$ verteilt. Der Zeitbedarf für das Stadium 1 ist daher $2q = O(\log n)$.

Das zweite Stadium benötigt $O(1)$ Schritte.

Zu Beginn des dritten Stadiums befinden sich bei festem i und j in dem q-dimensionalen Teilwürfel $P_{l,i,j}$, $0 \leq l \leq n-1$, die Werte $c'_{l,i,j}$. Die Summe wird wie in Beispiel 10.6.4 in $q = O(\log n)$ parallelen Schritten bestimmt. Sie befindet sich am Ende im Prozessor $P_{0,i,j}$.

Der Gesamtzeitbedarf zur Bestimmung des Matrizenproduktes zweier quadratischer n-zeiliger Matrizen ist also von der Ordnung $O(\log n)$. $\square$

Literaturverzeichnis

[1] *K. Alber, W. Struckmann:* Einführung in die Semantik von Programmiersprachen. Bibliographisches Institut, Wissenschaftsverlag, Mannheim 1988.

[2] *S. Baase:* Computer Algorithms. Adison-Wesley, Reading 1978.

[3] *L. S. Bobrow, M. A. Arbib:* Discrete Mathematics. Saunders, Philadelphia 1974.

[4] *D. P. Bovet, P. Crescenzi:* Introduction to the Theory of Complexity. Prentice Hall, New York 1994.

[5] *A. Brandstädt:* Graphen und Algorithmen. Teubner, Stuttgart 1994.

[6] *W. Brauer:* Automatentheorie. Teubner, Stuttgart 1984.

[7] *M. R. Garey, D. S. Johnson:* Computers and Intractability. Freeman, San Francisco 1979.

[8] *M. Gössel:* Automatentheorie für Ingenieure. Akademie-Verlag, Berlin 1991.

[9] *H. Hermes:* Aufzählbarkeit, Entscheidbarkeit, Berechenbarkeit, 3. Auflage. Springer, Berlin 1978.

[10] *B. Hohlfeld, W. Struckmann:* Einführung in die Programmverifikation. Bibliographisches Institut, Wissenschaftsverlag, Mannheim 1992.

[11] *J. E. Hopcroft, J. D. Ullman:* Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie. Addison-Wesley, Bonn 1990.

[12] *J. JáJá:* An Introduction to Parallel Algorithms. Addison-Wesley, Reading 1992.

[13] *D. E. Knuth:* The Art of Computer Programming. Volume 2 (Seminumerical Algorithms), second edition. Addison Wesley, Reading 1981.

[14] *M. Minsky:* Computation: Finite and Infinite Machines. Prentice-Hall, London 1972.

[15] *R. N. Moll, M. A. Arbib, A. J. Kfoury:* An Introduction to Formal Language Theory. Springer, New York 1988.

[16] *H. Noltemeier:* Informatik 1. Einführung in Algorithmen und Berechenbarkeit. Hanser, München 1981.

[17] *H. Noltemeier, R. Laue:* Informatik 2. Einführung in Rechnerstrukturen und Programmierung. Hanser, München 1984.

[18] *C. Posthoff, K. Schultz:* Grundkurs Theoretische Informatik. Teubner, Stuttgart 1992.

[19] *A. Salomaa:* Formale Sprachen. Springer, Berlin 1978.

[20] *P. Sander, W. Stucky, R. Herschel:* Automaten, Sprachen, Berechenbarkeit. Teubner, Stuttgart 1992.

[21] *P. H. Starke:* Abstrakte Automaten. VEB Deutscher Verlag der Wissenschaften, Berlin 1969.

[22] *R. Vollmar, Th. Worsch:* Modelle der Parallelverarbeitung. Teubner, Stuttgart 1994.

[23] *D. Wätjen:* Theoretische Informatik. Eine Einführung. Oldenbourg, München 1994.

Index

Abbildung einer kontextfreien Grammatik, 111
Ableitung, 81
Ableitungsbaum einer kontextfreien Grammatik, 112
Abschluß unter Sprachoperationen
  Homomorphismus, 95
  Spiegeloperation, 95
  Substitution, 95
Ackermann-Funktion, 67–72
Adjazenzmatrix, 121
Äquivalenz
  von endlichen erkennenden Automaten, 86
  von Grammatiken, 83
  von Turingmaschinen, 26
  von Typ-3-Grammatiken und regulären Ausdrücken, 94
  von Zuständen, 18, 87
  zwischen Moore- und endlichen erkennenden Automaten, 86
  zwischen Turingmaschinen und Registermaschinen, 54
Äquivalenzrelation, 16
  Äquivalenzklasse, 16
  Faktormenge, 16
Alphabet, 79
Anfangssymbol, 82
Antwortfunktion eines Moore-Automaten, 13
Arbeitsweise einer Turingmaschine, 24–25
Aufzählung
  von berechenbaren Funktionen, 46
  von Turingmaschinen, 42
Ausdruck, logischer, 132
  Belegung der Variablen, 133
  erfüllbar, 133
  quantifizierte Boolesche Formel (QBF), 188
  SAT, 134
  Standardkodierung, 133
Automat, *siehe*
abstrakter Automat bei Programmiersprachen
endlicher erkennender Automat
Kellerautomat
linear beschränkter Automat
Mealy-Automat
Moore-Automat
Turingmaschine

Bandfunktion, 24, 40
Bandinhalt, 28
Bandinschrift, 24, 28, 40
Belegung von logischen Variablen, 133
beschränkte Minimalisierung, 76
beschränkter μ-Operator, 73
Blankzeichen, 23

charakteristische Funktion, 46, 72, 79
Chomsky-Hierarchie, 85
Churchsche These, 45, 64, 78
CLIQUE, 165
COMMON CRCW PRAM, 192
cpo, *siehe* vollständige partiell-geordnete Menge
CRCW PRAM, 192
CREW PRAM, 192

3-FÄRBBARKEIT, 128
dualer Graph, 137

EA-Turingmaschine, 211
Einsetzung, 61
endlicher erkennender Automat, 85
  Äquivalenz, 86
  Äquivalenz von Zuständen, 87
  erreichbarer Zustand, 87
  nichtdeterministischer, 88
  reduzierter, 88
  vereinfachter, 87
  Zustandsüberführungsgraph, 86
Entscheidbarkeit, 44
Entscheidungsproblem, 117
varepsilon-approximierender Algorithmus, 179
erfüllbarer Ausdruck, 133

Erfüllbarbeitsproblem, 203
Erfüllbarkeitsproblem, 134
Ersetzungsregel, 81, 82

Familie von Sprachen, 83, 85
Fixpunkt, 108
Fixpunktsatz von *Kleene*, 109
FNC, Komplexitätsklasse, 204
FP, Komplexitätsklasse, 135
freie Halbgruppe, 11
freies Monoid, 11

Gödelisierung, 40, 41
Grammatik, 82
  Abbildung einer kontextfreien Grammatik, 111
  Ableitungsbaum einer kontextfreien Grammatik, 112
  Äquivalenz von Grammatiken, 83
  erzeugte Sprache, 82
  kontextfrei, 83
  kontextsensitiv, 83
  monoton, 83
  regulär, 83
  Typ i, 83
Graph, 121
  Adjazenzlisten, 121
  Adjazenzmatrix, 121
  benachbarte Knoten, 126
  Dreieck, 126
  3-FÄRBBARKEIT, 128
  duale, 137
  k-FÄRBBARKEIT, 169
  Multigraph, 121
  schlichter, 121
  Schlinge, 126
  starke Komponente, 135
  topologische Ordnung, 123
  Weg, 123
  2-FÄRBBARKEIT, 126–135
  2-FÄRBBARKEIT, 128
  zyklenfrei, 123
  Zyklus, 123

Halbgruppe, 11
Halteproblem, 44, 45, 48
HAMILTONSCHER KREIS, 169
Handlungsreisender, 155

Heiratsproblem, 174
Homomorphismus, 95
Hyperwürfel, 217

Induktions-Rekursionsschema, 62
INNERE EINES POLYGONS, 147
Iteration, 79

k-Äquivalenz von Zuständen, 19
k-Band-Turingmaschine, 33, 142
k-FÄRBBARKEIT, 169
KANTEN-2-FÄRBBARKEIT, 138
Kellerautomat, 97–98
  deterministischer, 98
Kette, 104
Klausel vom Grad k, 133
Kleenescher Fixpunktsatz, 109
kleinste obere Schranke, 103, 104
kleinster Fixpunkt, 108
  bei Grammatiken, 113
KNAPSACK-PROBLEM, 155
KNOTENÜBERDECKUNG eines Graphen, 170
kombinatorisches System, 81
Komplexität, 16, 22
Komplexitätsklasse
  FNC, 204
  FP, 135
  NC, 204
  NP, 154
  NPO, 172
  P, 117, 131
  PO, 173
  PSPACE, 185
Komposition von Turingmaschinen, 27
Konfiguration, 24
Konkatenation
  von Sprachen, 79
  von Wörtern, 11
KONVEXE HÜLLE, 148, 198

lba-Problem, 100
leeres Wort ε, 11
linear beschränkter Automat, 99
Linksmaschine, 26
Literal, 133

Matching, 173

Matrizenmultiplikation, 216, 219
MAX, 192
MAXIMALE CLIQUE, 172
MAXIMALFLUSSPROBLEM, 176, 206
Maximierungsproblem, 172
Mealy-Automat, 7, 30
- Äquivalenz mit Moore-Automaten, 12
- Taktung, 8
- Wertetabelle, 8
- Zustandsdiagramm, 9
- Zustandsgraph, 9

Menge von Wörtern, 11
MINIMALE FÄRBBARKEIT, 171
MINIMALE KNOTENÜBERDECKUNG, 178
MINIMALES TSP, 172
Minimalisierung, 76
Minimierungsproblem, 171
Modulo-m-Zähler, 5
Monoid, 11
monotone Abbildung, 106
Moore-Automat, 10
- Äquivalenz mit Mealy-Automaten, 12
- Äquivalenz mit reduziertem Automaten, 18
- Äquivalenz von Zuständen , 18
- Anfangszustand, 13
- Antwortfunktion, 13
- Arbeitsweise, 11
- k-Äquivalenz von Zuständen, 19
- Realisierungsproblem, 14
- Reduktion, 17
- Reduktionsalgorithmus, 20
- reduzierter, 18
- Taktung, 10
- Verhaltensfunktion, 13
- Zustand-Ausgabe-Graph, 10

μ-Operator, 73
μ-rekursiv, 77
Multigraph, 121

NC, Komplexitätsklasse, 204
NC-reduzierbar, 205
Netzwerk, 214
- Hyperwürfel, 217
- Ring, 214
- zweidimensionales Gitter, 215

nichtdeterministische Turingmaschine, 32, 96
nichtdeterministischer endlicher erkennender Automat, 88
normiert Turing-berechenbar, 36, 64
NP, Komplexitätsklasse, 117, 154
NP-hart, 178
NP-vollständig, 157
NPO, Komplexitätsklasse, 172

O-Notation, 22
O-Notation, 119
obere Schranke, 103
Ω-Notation, 119
Optimierungsproblem, 171

P, Komplexitätsklasse, 117, 131
P-vollständig, 205
Palindrome, 130, 208
parallele Berechenbarkeitsthese, 213
partiell-geordnete Menge, 102
- Fixpunkt, 108
- Kette, 104
- kleinste obere Schranke, 103, 104
- kleinstes Element, 103
- monotone Abbildung, 106
- obere Schranke, 103
- Supremum, 104
- vollständige partiell-geordnete Menge, *siehe* vollständige partiell-geordnete Menge

partielle Abbildung, 31
PARTITION, 170
PO, Komplexitätsklasse, 173
Polygon, 145
polynomial reduzierbar, 137
poset, *siehe* partiell-geordnete Menge
Postsches Korrespondenzproblem, 49
Prädikat, 72
- charakteristische Funktion, 72

Präfixsumme, 199
PRAM, 191
- Scheduling-Darstellung, 194
- *WT*-Darstellung, 194

primitiv-rekursive Funktion, 61, 62

Beispiele, 62–63
Grundfunktionen, 61
Nachfolgerfunktion, 61
Nullfunktion, 61
Projektionsfunktion, 61
primitives Rekursionsschema, 62
PRIORITY CRCW PRAM, 192
Produktion, 81, 82
Projektionsfunktion, 10, 61, 105, 107
PSPACE, Komplexitätsklasse, 185

QBF, 188, 189
quantifizierte Boolesche Formel (QBF), 188

RAM, 50, 143
Äquivalenz mit Turingmaschine, 54
Arbeitsweise, 51
Bestandteile, 50
deterministische, 51
nichtdeterministische, 51
Programm, 51
RAM-Berechenbarkeit, 52
RAM-berechenbar, 52
Raumkomplexität, 185
Realisierungsproblem, 14
Rechtsmaschine, 26
Reduktion
NC-, 205
polynomiale, 137
Reduktion von Moore-Automaten, 17
Reduktionsalgorithmus für Moore-Automaten, 20
reduzierter endlicher erkennender Automat, 88
reduzierter Moore-Automat, 18
Registermaschine, *siehe* RAM, 143
logarithmische Zeitkomplexität, 143
uniforme Zeitkomplexität, 143
regulärer Ausdruck, 93
universell, 134
rekursiv, 84
rekursiv-aufzählbar, 84
Ring, 214

SAT, 134, 154, 203
SAT(2), 134, 140
SAT(3), 163
Schaltfunktion, 8
Schaltkreis, 207
Schaltkreisfamilie, 207
uniforme, 212
Schaltkreisfunktion, 207
Schaltnetz, 8
Scheduling Independent Tasks, 180
Schreibmaschine, 26
serielle Addition, 5
SIT(k), 180
Sortieren, 196–198
Spiegeloperation, 95
Sprache, 79
kontextfrei, 83
kontextsensitiv, 83
Palindrome, 130
regulär, 83, 94
rekursiv, 84
rekursiv-aufzählbar, 84
Typ i, 83
von endlichem Automaten erkannt, 85
von nichtdeterministischem endlichen Automaten erkannt, 88
Sprachfamilie, 83, 85
starke Komponente, 135, 140
stetige Abbildung, 106
Strukturbaum, 80
systolischer Algorithmus, 216

Taktung von Mealy-Automaten, 8
Taktung von Moore-Automaten, 10
topologische Ordnung, 123
topologisches Sortieren, 123–126
totale Funktion, 61
TRAVELING-SALESMAN-PROBLEM, 155
TSP, 155, 169
Turing-berechenbar, 34, 36
Turingmaschine, 23
Äquivalenz mit Registermaschine, 54
Äquivalenz von Turingmaschinen, 26
akzeptierende, 96
akzeptierende Rechnung, 96

akzeptierte Sprache, 97
Arbeitsweise, 24–25
Aufzählung, 42
Bandfunktion, 24, 40
Bandinhalt, 28
Bandinschrift, 24, 28, 40
Blankzeichen, 23
deterministische
  Zeitkomplexität, 129
Endzustände, 96
Gödelisierung, 40, 41
große Linksmaschine, 28
große Rechtsmaschine, 28
haltende Rechnung, 96
Halteproblem, 44, 45
Interpretation, 24
Komposition, 27
Konfiguration, 24
Kopiermaschine, 28
Länge der Rechnung, 96
Leerzeichen, 23
linke Translationsmaschine, 29
Links-Suchmaschine, 29
Linksmaschine, 26
m-Kopiermaschine, 30
mit Ein- und Ausgabeband, 211
mit k Bändern, 33, 142
modifizierte Definitionen, 31–34
nichtdeterministische, 32, 96
  Zeitkomplexität, 153
normiert Turing-berechenbar, 36
Raumkomplexität, 185
Rechts-Suchmaschine, 29
Rechtsmaschine, 26
Schreibmaschine, 26
Turing-berechenbar, 34, 36
Turingtafel, 23–24
  universelle, 43
  unvollständige, 31
  Verschiebemaschine, 30
Turingtafel, 23–24
Turnieralgorithmus, 193
Typ-i-Grammatik, 83
Typ-i-Sprache, 83

unbeschränkter μ-Operator, 73
Unentscheidbarkeit, 44
uniforme Schaltkreisfamilie, 212
universelle Turingmaschine, 43
unvollständige Turingmaschine, 31

Verhalten eines Moore-Automaten, 13
Verhaltensfunktion, 14
vollständige partiell-geordnete Menge,
    104
  Fixpunktsatz, 109
  stetige Abbildung, 106

Wertetabelle von Mealy-Automaten, 8
Wort, 11

X^*-Erweiterung, 11, 88

ZERLEGBARKEIT, 153
Zertifikatensprache, 162
Zuordnung, 173
Zustand-Ausgabe-Automat, *siehe*
    Moore-Automat
Zustand-Ausgabe-Graph, 10
Zustandsdiagramm, 6, 9
Zustandsgraph, 6, 9
Zustandsüberführungsgraph, 86
2-FÄRBBARKEIT, 126–135
2-FÄRBBARKEIT, 128
zweidimensionales Gitter, 215
Zyklus, 123