University of Applied Sciences

Fachhochschule Nordostniedersachsen

Lüneburg
Buxtehude
Suderburg

Arbeitsberichte aus der Informatik

Incl. CD

FINAL

üneburg

rbeitsberichte

formatik

achhochschule Nordostniedersachsen

Ausgewählte Kapitel der Theoretischen Informatik

Ulrich Hoffmann

13. Jahrgang, Heft 1, Januar 2003, ISSN 0939-8821

Technical Reports and Working Papers
Hrsg: Hinrich E. G. Bonin
Volgershall 1, D-21339 Lüneburg
Phone: xx49.4131.677175 Fax: xx49.4131.677140

Inhaltsverzeichnis

1 Einleitung

Der vorliegende Text trifft eine (subjektive) Auswahl aus Themen der Theoretischen Informatik, deren Kenntnis neben vielen anderen Themen als unabdingbar im Rahmen der Informatikausbildung angesehen wird. Auch wenn die direkte praktische Relevanz der dargestellten Inhalte nicht immer sofort sichtbar wird, so zählen diese doch zu den grundlegenden Bildungsinhalten, die jedem Informatiker und Wirtschaftsinformatiker vermittelt werden sollten. Zudem steht für die (dem einen oder anderen auch sehr theoretisch erscheinenden) Ergebnisse gerade der Komplexitätstheorie außer Frage, daß sie einen unmittelbaren Einfluß auf die praktische Umsetzung und algorithmische Realisierung von Problemlösungen haben.

Der Text verzichtet stellenweise auf die Darstellung ausformulierter (mathematisch exakter) Beweise. Diese können in der angegebenen Literatur nachgelesen werden bzw. sie werden in der zugehörigen Vorlesung und den Übungen behandelt.

Die Theoretische Informatik als eine der Säulen der Informatik kann natürlich nicht umfassend im Rahmen einer einsemestrigen Lehrveranstaltung dargestellt werden. Ein Weg, sich den wesentlichen Inhalten zu nähern, zu denen die Theorie der Berechenbarkeit und Entscheidbarkeit, die Automatentheorie, die Theorie der Formalen Sprachen und die Komplexitätstheorie zählen, wird in dem Lehrbuch von Schöning: *Theoretische Informatik kurzgefaßt* aufgezeigt, in dem die angesprochenen Themen in kompakter Form präsentiert werden. Dem vorliegenden Text liegt ein ähnlicher Ansatz zugrunde, wobei hier jedoch eine teilweise unterschiedliche Schwerpunktbildung erfolgt. Er verzichtet auf den Anspruch einer umfassenden oder ansatzweisen Darstellung *aller* wichtigen Teilgebiete der Theoretischen Informatik und konzentriert sich auf einige Grundlagen, die zum Verständnis der prinzipiellen Leistungsfähigkeit von Algorithmen und Computern notwendig erscheinen. Der Blick richtet sich also auf Inhalte, die der Beantwortung von Fragen dienlich sein können wie

- Wie kann man die Begriffe der algorithmischen Berechenbarkeit formal fassen?
- Was können Computer und Algorithmen leisten und gibt es prinzipielle Grenzen der algorithmischen Berechenbarkeit?
- Mit welchen Modellierungswerkzeugen und Beschreibungsmitteln lassen sich berechenbare Menge charakterisieren?
- Mit welchem algorithmischen Aufwand ist im konkreten Fall bei einer Problemlösung zu rechnen?
- Wie kann man die inhärente Komplexität einiger Problemstellungen praktisch nutzen?

Die Behandlung dieser Fragestellungen führt auf die Betrachtung von Modellen der Berechenbarkeit. Hierbei spielt die Turingmaschine eine herausragende Rolle. Ergänzend wird wegen seiner inhaltlichen Nähe zu realen Computern das Modell der Random Access Ma-

schine behandelt. Auf die Darstellung von Theorien wie die m-rekursiven Funktionen oder das l-Kalkül soll hier (im wesentlichen aus Platzgründen) verzichtet werden, obwohl sie eine hohe mathematische Eleganz und Relevanz aufweisen.

Innerhalb der Theoretischen Informatik nimmt das Teilgebiet Automatentheorie und Formale Sprachen einen breiten Raum ein. Historisch hat es einen großen Einfluß auf die Entwicklung von Programmiersprachen und Sprachübersetzern, sowie auf die Modellierung komplexer Anwendungssysteme und Betriebssysteme ausgeübt. Im folgenden wird dieses Teilgebiet jedoch nur im Überblick dargestellt, da die Themen Programmiersprachen und Compilerbau in der Wirtschaftsinformatikausbildung höchstens am Rand gestreift werden und auch die Modellierungsmöglichkeiten beispielsweise paralleler Abläufe durch Petrinetze und anderer Automatentypen innerhalb des jeweiligen Anwendungsgebiets behandelt werden können.

Das Thema der praktisch durchführbaren Berechnungen, d.h. der in polynomieller Zeit zu lösenden Probleme, führt auf die Theorie der NP-Vollständigkeit und der Approximation schwerer Optimierungsprobleme. Die damit verbundenen Fragestellungen werden in den beiden letzten Kapiteln aufgezeigt.

Im folgenden Text werden im einzelnen nicht die Literaturquellen angegeben, denen die jeweiligen Inhalte entnommen wurden. Das Literaturverzeichnis führt eine Reihe wichtiger Standardwerke auf, die die Themen in großer Ausführlichkeit behandeln. Dort finden sich auch zusätzliche Übungen und weiterführende Quellenangaben.

1.1 Einige grundlegende Bezeichnungen

Im vorliegenden Kapitel werden grundlegende Definitionen angeführt und einige in den folgenden Kapiteln verwendete (mathematische) Grundlagen zitiert. Dabei wird eine gewisse Vertrautheit mit der grundlegenden Symbolik der Mathematik vorausgesetzt, z.B. mit Schreibweisen wie
$a \in A$, $A \subseteq B$, $A \cup B$, $A \cap B$, $A \setminus B$.

Es seien *A* und *B* mathematisch logische Aussagen, die wahr oder falsch sein können. In den nachfolgenden Kapiteln werden häufig daraus Aussagen der Form
„Aus *A* folgt *B*“ bzw.
„*A* impliziert *B*“ bzw.
„Wenn *A* gilt, dann gilt auch *B*“, gelegentlich auch geschrieben
„ $A \Rightarrow B$ “
gebildet und bewiesen.

Für einen Beweis der Aussage $A \Rightarrow B$ geht man folgendermaßen vor:

Man nimmt an, daß *A* wahr ist. Durch eine „geeignete“ Argumentation (Anwendung logischer Schlüsse) zeigt man, daß dann auch *B* wahr ist.

Alternativ kann man auch folgendermaßen argumentieren:

Man nimmt an, daß *B nicht* wahr ist. Durch eine „geeignete“ Argumentation (Anwendung logischer Schlüsse) zeigt man, daß dann auch *A* nicht wahr ist. Diese Argumentation beruht auf der Tatsache, daß $A \Rightarrow B$ logisch äquivalent (siehe unten) zu $\neg B \Rightarrow \neg A$ ist, d.h. $A \Rightarrow B$ und $\neg B \Rightarrow \neg A$ haben stets denselben Wahrheitswert.

Die Aussage „$A \Leftrightarrow B$“ steht für $A \Rightarrow B$ und $B \Rightarrow A$. Um die Aussage $A \Leftrightarrow B$ zu beweisen, sind also zwei „Richtungen“ zu zeigen, d.h. jeweils ein Beweis für $A \Rightarrow B$ und ein Beweis für $B \Rightarrow A$ zu erbringen. Alternativ kann man auch $A \Rightarrow B$ und $\neg A \Rightarrow \neg B$ beweisen. Die Gültigkeit von $A \Leftrightarrow B$ bedeutet, daß *A* und *B* beide wahr oder beide falsch sind.

Statt $A \Leftrightarrow B$ sagt man auch
„*A* ist (logisch) äquivalent zu *B*“ bzw.
„*A* gilt genau dann, wenn *B* gilt“.

Die Elemente einer Menge *A* seien durch eine Eigenschaft E_A beschrieben, die allen Elementen von *A* zukommt: $A = \{a \mid a \text{ hat die Eigenschaft } E_A\}$. Entsprechend werde die Menge *B* durch eine Eigenschaft E_B bestimmt: $B = \{b \mid b \text{ hat die Eigenschaft } E_B\}$. Um zu beweisen, daß die Teilmengenbeziehung $A \subseteq B$ gilt, geht man folgendermaßen vor:

Man nehme $x \in A$. Dann weiß man, daß *x* die Eigenschaft E_A besitzt. Durch eine „geeignete“ Argumentation unter Ausnutzung der Eigenschaft E_A weist man nach, daß *x* auch die Eigenschaft E_B besitzt, d.h. $x \in B$. Man zeigt also $x \in A \Rightarrow x \in B$.

Logisch äquivalent ist folgende Argumentation:

Man nehme für ein Element *x* an, daß es *nicht* in *B* liegt: $x \notin B$, d.h. *x* hat *nicht* die Eigenschaft E_B. Durch eine „geeignete“ Argumentation unter Ausnutzung der Tatsache, daß für *x* die Eigenschaft E_B nicht zutrifft, weist man nach, daß *x* auch nicht die Eigenschaft E_A besitzt, d.h. daß $x \notin A$ ist.

Um die Gleichheit zweier Mengen *A* und *B* nachzuweisen, d.h. $A = B$, hat man die beiden Inklusionen $A \subseteq B$ und $B \subseteq A$ nachzuweisen, d.h. $x \in A \Leftrightarrow x \in B$.

Mit $\mathbf{P}(M)$ wird die **Potenzmenge** der Menge *M* bezeichnet, d.h. $\mathbf{P}(M) = \{ L \mid L \subseteq M \}$.

Es gilt für alle Mengen *A*, *B* und *C*:
$A \cap B \subseteq A,\ A \cap B \subseteq B,\ A \subseteq A \cup B,\ B \subseteq A \cup B$
$A \cup B = B \cup A,\ A \cap B = B \cap A$ (Kommutativgesetze)
$A \cup B = (A \setminus B) \cup (A \cap B) \cup (B \setminus A)$, die rechte Seite ist eine disjunkte Zerlegung[1] von $A \cup B$,
$A \cap (B \cup C) = (A \cap B) \cup (A \cap C),\ A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ (Distributivgesetze).

Für Mengen $A_1,\ A_2,\ ...,\ A_n$ wird das **kartesische Produkt** definiert als
$A_1 \times A_2 \times ... \times A_n = \{ (a_1, a_2, ..., a_n) \mid a_1 \in A_1, a_2 \in A_2, ..., a_n \in A_n \}$.

Für eine Menge *A* bezeichnet |*A*| die **Anzahl der Elemente (**oder die **Mächtigkeit)** von *A*.

Es seien *A* und *B* Mengen. Eine Vorschrift, die jedem Element $a \in A$ *genau* ein Element $b \in B$ zuordnet, heißt **(totale) Funktion** von *A* nach *B*, geschrieben:

$$f: \begin{cases} A \to B \\ a \to f(a) \end{cases}$$

häufig auch in der Form
$f: A \to B,\ f(a) = \ ...$

Die Angabe $f: A \to B$ legt fest, daß einem Element vom (Daten-) Typ, der „charakteristisch“ für *A* ist, jeweils ein Element vom (Daten-) Typ, der „charakteristisch“ für *B* ist, zugeordnet wird. Beispielsweise könnte die Menge *A* aus Objekten vom Objekttyp *T* und die Menge *B* aus natürlichen Zahlen bestehen. Dann legt die Angabe $f: A \to B$ fest, daß jedem Objekt vom Objekttyp *T* in der Menge *A* durch *f* eine natürliche Zahl, die beispielsweise als Primärschlüsselwert interpretierbar ist, zugeordnet wird. Die Angabe $f(a) = \ ...$ beschreibt, wie diese Zuordnung für jedes Element $a \in A$ geschieht.

Formal ist eine (totale) Funktion $f: A \to B$ eine Abbildungsvorschrift (genauer: eine zweistellige Relation), die folgenden beiden Bedingungen genügt:

(i) Sie ist **linkstotal**: für jedes $a \in A$ gibt es $b \in B$ mit $f(a) = b$

(ii) Sie ist **rechtseindeutig**: $f(a) = b_1$ und $f(a) = b_2$ implizieren $b_1 = b_2$.

Die Menge *A* heißt **Definitionsbereich** von *f*, die Menge

[1] Eine Zerlegung $M = M_1 \cup M_2$ der Menge *M* heißt disjunkt, wenn $M_1 \cap M_2 = \varnothing$ ist.

$f(A) = \{ b \mid b \in B$, und es gibt $a \in A$ mit $f(a) = b \}$
heißt **Wertebereich** von *f*. Es ist $f(A) \subseteq B$.

Eine Abbildungsvorschrift $f: A \to B$ heißt **partielle Funktion**, wenn lediglich die obige Bedingung (ii) erfüllt ist. Eine partielle Funktion $f: A \to B$ ordnet u.U. nicht jedem $a \in A$ ein $b \in B$ zu.

Eine Abbildungsvorschrift $f: A \to B$ heißt **injektiv**, wenn sie **linkseindeutig** ist, d.h. wenn gilt:
für jedes $a_1 \in A$ und jedes $a_2 \in A$ gilt: Aus $f(a_1) = f(a_2)$ folgt $a_1 = a_2$. Gleichbedeutend damit ist:
für jedes $a_1 \in A$ und jedes $a_2 \in A$ mit $a_1 \neq a_2$ ist $f(a_1) \neq f(a_2)$.

Eine Abbildungsvorschrift $f: A \to B$ heißt **surjektiv**, wenn sie rechtstotal ist, d.h. wenn gilt: $f(A) = B$.

Eine Abbildungsvorschrift $f: A \to B$ heißt **bijektiv**, wenn sie injektiv und surjektiv ist. In diesem Fall gibt es eine eindeutig bestimmte **Umkehrfunktion** $\overset{-1}{f}: B \to A$ mit $\overset{-1}{f}(f(a)) = a$ und $f\left(\overset{-1}{f}(b)\right) = b$. Hierbei sind $a \in A$ und $b \in B$.

Die **Menge der natürlichen Zahlen** wird definiert durch **N** = { 0, 1, 2, 3, 4, ... }.

Die Theoretische Informatik untersucht häufig Eigenschaften von Zeichenketten, die sich aus einzelnen Symbolen (Buchstaben, Zeichen) zusammensetzen.

Es sei Σ eine endliche nichtleere Menge. Die Elemente von Σ heißen **Symbole** oder **Zeichen** oder **Buchstaben**. Σ heißt **Alphabet**. Eine Aneinanderreihung $a_1 a_2 \ldots a_n$ von *n* Symbolen (Zeichen, Buchstaben) $a_1 \in \Sigma$, $a_2 \in \Sigma$, ..., $a_n \in \Sigma$ heißt **Wort (Zeichenkette) über** Σ. **Die Länge** von $a_1 a_2 \ldots a_n$ wird durch $|a_1 a_2 \ldots a_n| = n$ definiert. Das **leere Wort (leere Zeichenkette)**, das keinen Buchstaben enthält, wird mit e bezeichnet; es gilt $|e| = 0$.

Formal lassen sich die Wörter über Σ wie folgt definieren:

(i) e ist ein Wort über Σ mit $|e| = 0$
(ii) Ist *x* ein Wort über Σ mit $|x| = n - 1$ und $a \in \Sigma$, dann ist *xa* ein Wort über Σ mit $|xa| = n$

(iii) *y* ist ein Wort über Σ genau dann, wenn es mit Hilfe der Regeln (i) und (ii) konstruiert wurde.

Die **Menge aller Wörter (beliebiger Länge)** über Σ, einschließlich des leeren Worts, wird mit Σ^* bezeichnet. Zusätzlich wird $\Sigma^+ = \Sigma^* \setminus \{e\}$ gesetzt.

Sind $x = a_1 a_2 \ldots a_n$ und $y = b_1 b_2 \ldots b_m$ zwei Wörter aus Σ^*, so heißt das Wort $xy = a_1 a_2 \ldots a_n b_1 b_2 \ldots b_m$ die **Konkatination** von *x* und *y*. Die Länge von *xy* ist $|xy| = |x| + |y|$.

Für $x \in \Sigma^*$ definiert man $x^0 = e$ und $x^{n+1} = x^n x$ für $n \geq 0$.

Eine Teilmenge $L \subseteq \Sigma^*$ heißt **(formale) Sprache** über dem Alphabet Σ.

Für Sprachen $L_1 \subseteq \Sigma^*$ und $L_2 \subseteq \Sigma^*$ definiert man das **Produkt** von L_1 und L_2 durch $L_1 \cdot L_2 = \{ xy \mid x \in L_1 \text{ und } y \in L_2 \}$.

Der **Abschluß** L^* einer Sprache $L \subseteq \Sigma^*$ wird durch folgende Regeln definiert:

(i) $L^0 = \{e\}$
(ii) $L^{n+1} = L^n \cdot L$ für $n \geq 0$
(iii) $L^* = \bigcup_{n \geq 0} L^n$.

Unter dem **positiven Abschluß** L^+ einer Sprache $L \subseteq \Sigma^*$ versteht man $L^+ = \bigcup_{n \geq 1} L^n$.

Offensichtlich ist $L^* = L^+ \cup \{e\}$.

Es sei $w \in L^*$. Dann ist entweder $w = e$ oder es gibt ein $n \in \mathbf{N}$ mit $n \geq 1$ und $w \in L^n$. In diesem Fall gibt es *n* Wörter $w_1 \in L$, ..., $w_n \in L$ mit $w = w_1 \ldots w_n$. Jedes Wort w_1, ..., w_n werde durch seine einzelnen Buchstaben dargestellt: $w_1 = a_{1,1} \ldots a_{1,i_1}$, ..., $w_n = a_{n,1} \ldots a_{n,i_n}$ mit $a_{l,k} \in \Sigma$ für *l* = 1, ..., *n* und *k* = 1, ..., i_l. Dann ist

$w = a_{1,1} \ldots a_{1,i_1} \ldots a_{n,1} \ldots a_{n,i_n}$. Für die Länge von *w* gilt $|w| = \sum_{l=1}^{n} |w_l| = \sum_{l=1}^{n} i_l$.

Ein Wort $w \in L^+$. Kann man also in der Form $w = w_1 x = y w_n$ mit $x \in L^{n-1}$ und $y \in L^{n-1}$ schreiben. Wegen $L^{n-1} \subseteq L^*$ sind beide Teilwörter *x* und *y* in L^*. Insgesamt ergibt sich $w \in L \cdot L^*$ und $w \in L^* \cdot L$ und damit $L^+ \subseteq L \cdot L^*$ und $L^+ \subseteq L^* \cdot L$. Die umgekehrten Inklusionen $L \cdot L^* \subseteq L^+$ und $L^* \cdot L \subseteq L^+$ ergeben sich entsprechend, so daß damit

$L^+ = L \cdot L^* = L^* \cdot L$

gezeigt ist.

Es gilt
$L \cdot (L_1 \cup L_2) = (L \cdot L_1) \cup (L \cdot L_2)$,
$L \cdot (L_1 \cap L_2) \subseteq (L \cdot L_1) \cap (L \cdot L_2)$, und es gibt Beispiele für Mengen L, L_1 und L_2, für die diese Inklusion nicht umkehrbar ist (beispielsweise wählt man $L = \{e, 1\}$, $L_1 = \{0\}$ und $L_2 = \{10\}$).

Eine Menge M ist **endlich von der Mächtigkeit (Kardinalität)** n, wenn es eine bijektive Abbildung $f : \{0, ..., n-1\} \to M$ gibt, d.h. man kann die Elemente in M mit den natürlichen Zahlen $0, ..., n-1$ durchnumerieren: $M = \{m_0, ..., m_{n-1}\}$. Hierbei ist $m_i \neq m_j$ für $i \neq j$.

Eine Menge M ist **(von der Mächtigkeit bzw. Kardinalität) unendlich**, wenn sie eine bijektive Abbildung $f : N \to M$ zwischen einer echten Teilmenge $N \subset M$ und M gibt.

Zwei Mengen M_1 und M_2 heißen **gleichmächtig** (oder **von gleicher Kardinalität**), wenn es eine bijektive Abbildung zwischen M_1 und M_2 gibt.

Eine Menge M, die gleichmächtig mit der Menge **N** (der natürlichen Zahlen) ist, heißt **abzählbar unendlich.** Eine Menge heißt **(höchstens) abzählbar**, wenn sie endlich oder abzählbar unendlich ist.

Ist M eine abzählbar unendliche Menge, dann gibt es eine bijektive Abbildung $f : \mathbf{N} \to M$, d.h. $M = \{f(0), f(1), f(2), ...\}$. Intuitiv: Man kann die Elemente von M mit natürlichen Zahlen durchnumerieren; jedes $m \in M$ hat dabei eine eindeutige Nummer i, d.h. $m = f(i)$.

Für ein endliches Alphabet $\Sigma = \{a_1, ..., a_n\}$ kann man die Wörter aus Σ^* (in **lexikographischer Reihenfolge**) gemäß folgender Tabelle notieren und durchnumerieren:

Nummer	Wort aus Σ^*	Bemerkung	Nummer	Wort aus Σ^*	Bemerkung
0	e	Wort der Länge 0	$n+n^2+1$	$a_1\ a_1\ a_1$	Wörter der Länge 3
1	a_1	Wörter der Länge 1	$n+n^2+2$	$a_1\ a_1\ a_2$	
2	a_2		...	...	
...	...		n^2+2n	$a_1\ a_1\ a_n$	
n	a_n		n^2+2n+1	$a_1\ a_2\ a_1$	
$n+1$	$a_1\ a_1$	Wörter der Länge 2	...	...	
$n+2$	$a_1\ a_2$		n^2+3n	$a_1\ a_2\ a_n$	
...	...		...	...	
$2n$	$a_1\ a_n$		$n+n^2+n^3$	$a_n\ a_n\ a_n$	
$2n+1$	$a_2\ a_1$		...	...	
$2n+2$	$a_2\ a_2$				
...	...				
$3n$	$a_2\ a_n$				
...	...				
$n+n^2$	$a_n\ a_n$				

Das leere Wort e erhält dabei die Nummer 0; die Wörter der Länge $k>0$ bekommen die Nummern $\left(\sum_{i=1}^{k-1} n^i\right)+1=\frac{n^k-1}{n-1}$ bis $\sum_{i=1}^{k} n^i=\frac{n\left(n^k-1\right)}{n-1}$. Es gilt also:

Satz 1.1-1:

Es sei Σ ein endliche Alphabet. Dann gilt:

1. Σ^* ist abzählbar unendlich.

2. Jede Sprache $L\subseteq\Sigma^*$ ist entweder endlich oder abzählbar unendlich.

Teil 2. des Satzes folgt aus der Tatsache, daß jede Teilmenge einer abzählbar unendlichen Menge abzählbar ist.

Satz 1.1-2:

Die Potenzmenge $\mathbf{P}(M)=\{L\,|\,L\subseteq M\}$ einer endlichen Menge *M* mit *n* Elementen enthält 2^n Elemente (Teilmengen von *M*).

Beweis:

Es sei $M = \{a_0, ..., a_{n-1}\}$ eine endliche Menge mit *n* Elementen. Jede Teilmenge $N \subseteq M$ mit *k* Elementen, etwa $N = \{a_{i_0}, ..., a_{i_{k-1}}\}$ kann als 0-1-Vektor der Länge *n* dargestellt werden. Dabei sind alle Komponenten dieses Vektors gleich 0 bis auf die Komponenten an den Positionen i_0, ..., i_{k-1}; dort steht jeweils eine 1. Umgekehrt kann jeder 0-1-Vektor der Länge *n* als Teilmenge von *M* interpretiert werden. ///

Ist *M* dagegen abzählbar unendlich, so gilt:

Satz 1.1-3:

Die Potenzmenge $\mathbf{P}(M) = \{L \mid L \subseteq M\}$ einer abzählbar unendlichen Menge *M* ist nicht abzählbar (man sagt: sie ist **überabzählbar**).

Beweis:

Es wird die **Diagonalisierungstechnik** angewendet:
Es sei $f : \mathbf{N} \to M$ eine Abzählung von *M*, d.h. $M = \{f(0), f(1), f(2), ...\}$ mit $f(i) \neq f(j)$ für $i \neq j$. Angenommen, $\mathbf{P}(M)$ sei abzählbar, etwa mit Hilfe der bijektiven Abbildung $g : \mathbf{N} \to \mathbf{P}(M)$, d.h. $\mathbf{P}(M) = \{g(0), g(1), g(2), ...\}$. Ein Wert $f(i)$ ist ein Element von *M*, ein Wert $g(i)$ ist eine Teilmenge von *M*. Es wird eine Teilmenge *D* von *M* durch $D = \{f(i) \mid f(i) \notin g(i)\}$ definiert.
Da *D* eine Teilmenge von *M* ist (denn alle Werte $f(i)$ sind Elemente von *M*), ist *D* Element von $\mathbf{P}(M)$, hat also eine Nummer $n_D \in \mathbf{N}$, d.h. $D = g(n_D)$. Es wird nun untersucht, ob das Element von *M* mit der Nummer n_D (das ist das Element $f(n_D)$) in *D* liegt oder nicht:
Es gilt $f(n_D) \in D$ nach Definition von *D* genau dann, wenn $f(n_D) \notin g(n_D)$ ist. Wegen $D = g(n_D)$ ist dieses gleichbedeutend mit $f(n_D) \notin D$. Dieser Widerspruch zeigt, daß die Annnahme, $\mathbf{P}(M)$ sei abzählbar, falsch ist. ///

Mit $M = \Sigma^*$ sieht man:

Satz 1.1-4:

Die Menge $\{L \mid L \subseteq \Sigma^*\}$ aller Sprachen über einem endlichen Alphabet Σ ist nicht abzählbar.

Im folgenden seien $f : \mathbf{N} \to \mathbf{R}$ und $g : \mathbf{N} \to \mathbf{R}$ zwei Funktionen. Die Funktion *f* ist von der (**Größen-) Ordnung** $O(g(n))$, geschrieben $f(n) \in O(g(n))$, wenn gilt:
es gibt eine Konstante $c > 0$, die von *n* nicht abhängt, so daß $|f(n)| \le c \cdot |g(n)|$ ist für jedes $n \in \mathbf{N}$ bis auf höchstens endlich viele Ausnahmen („ ..., so daß $|f(n)| \le c \cdot |g(n)|$ ist für fast alle $n \in \mathbf{N}$ “).

Einige Regeln:

$f(n) \in O(f(n))$

Für *d* = const. ist $d \cdot f(n) \in O(f(n))$

Es gelte $|f(n)| \le c \cdot |g(n)|$ für jedes $n \in \mathbf{N}$ bis auf höchstens endlich viele Ausnahmen. Dann ist $O(f(n)) \subseteq O(g(n))$, insbesondere $f(n) \in O(g(n))$.

$O(O(f(n))) = O(f(n))$;
hierbei ist

$$O(O(f(n))) = \left\{ h \;\middle|\; \begin{array}{l} h : \mathbf{N} \to \mathbf{R} \text{ und es gibt eine Funktion } g \in O(f(n)) \text{ und eine Konstante } c > 0 \\ \text{mit } |h(n)| \le c \cdot |g(n)| \text{ für jedes } n \in \mathbf{N} \text{ bis auf höchstens endlich viele Ausnahmen} \end{array} \right\}$$

Im folgenden seien S_1 und S_2 zwei Mengen, deren Elemente miteinander arithmetisch verknüpft werden können, etwa durch den Operator $\circ$. Dann ist
$S_1 \circ S_2 = \{ s_1 \circ s_2 \mid s_1 \in S_1 \text{ und } s_2 \in S_2 \}$.
Mit dieser Notation gilt:

$O(f(n)) \cdot O(g(n)) \subseteq O(f(n) \cdot g(n))$,

$O(f(n) \cdot g(n)) \subseteq |f(n)| \cdot O(g(n))$,

$O(f(n)) + O(g(n)) \subseteq O(|f(n)| + |g(n)|)$.

Satz 1.1-5:

Ist *p* ein Polynom vom Grade *m*, $p(n) = \sum_{i=0}^{m} a_i \cdot n^i$ mit $a_i \in \mathbf{R}$ und $a_m \neq 0$, so ist
$p(n) \in O(n^k)$ für $k \ge m$,
$n^m \in O(n^k)$ für $k \ge m$.

Konvergiert $f(n)=\sum_{i=0}^{\infty} a_i \cdot n^i$ für $n \le r$, so ist für jedes $k \ge 0$: $f(n)=\sum_{i=0}^{k} a_i \cdot n^i + g(n)$ mit $g(n) \in O\left(n^{k+1}\right)$.

Für $a>1$ und $b>1$ ist $\log_a(n)=\frac{1}{\log_b(a)} \cdot \log_b(n)$. Daher gilt

Satz 1.1-6:
Für $a>1$ und $b>1$ ist $\log_a(n) \in O\left(\log_b(n)\right)$.

Wegen $e^{h(n)}=2^{\log_2(e)\cdot h(n)}$ gilt

Satz 1.1-7:
$e^{h(n)} \in O\left(2^{O(h(n))}\right)$.

Für jedes Polynom $p(n)$ und jede Exponentialfunktion a^n mit $a>1$ ist $a^n \notin O\left(p(n)\right)$, jedoch $p(n) \in O\left(a^n\right)$.

Für jede Logarithmusfunktion $\log_a(n)$ mit $a>1$ und jedes Polynom $p(n)$ ist $p(n) \notin O\left(\log_a(n)\right)$, jedoch $\log_a(n) \in O\left(p(n)\right)$.

Für jede Logarithmusfunktion $\log_a(n)$ mit $a>1$ und jede Wurzelfunktion $\sqrt[m]{n}$ ist $\sqrt[m]{n} \notin O\left(\log_a(n)\right)$, jedoch $\log_a(n) \in O\left(\sqrt[m]{n}\right)$.

Insgesamt ergibt sich damit

Satz 1.1-8:
Es seien $a>1$, $b>1$, $m \in \mathbf{N}_{>0}$, $p(n)$ ein Polynom; dann ist
$O\left(\log_b(n)\right) \subset O\left(\sqrt[m]{n}\right) \subset O\left(p(n)\right) \subset O\left(a^n\right)$.

Ein **Boolescher Ausdruck** (**Boolesche Formel, Formel der Aussagenlogik**) ist eine Zeichenkette über dem Alphabet $A = \{\wedge, \vee, \neg, (,), 0, 1, x\}$, der eine Aussage der Aussagenlogik darstellt[2]. Innerhalb eines Booleschen Ausdrucks kommen Variablen vor, die mit dem Zeichen *x* beginnen und von einer Folge von Zeichen aus {0, 1} ergänzt werden. Diese ergänzende 0-1-Folge, die an das Zeichen *x* „gebunden" ist, wird als Indizierung der Variablen interpretiert. Im folgenden werden daher Variablen als indizierte Variablen geschrieben, wobei die indizierende 0-1-Folge als Binärzahl in Dezimaldarstellung angegeben wird. Kommen die Zeichen 0 bzw. 1 innerhalb eines Booleschen Ausdrucks nicht an das Zeichen *x* gebunden vor, so werden sie als Konstanten FALSE (FALSCH) bzw. TRUE (WAHR) interpretiert.

Beispielsweise steht $(x_1 \wedge \neg x_5) \vee (\neg x_3 \wedge (x_4 \vee x_5)) \vee 0$ für den Booleschen Ausdruck $(x1 \wedge \neg x101) \vee (\neg x11 \wedge (x100 \vee x101)) \vee 0$.

Die Zeichen $\wedge$, $\vee$ bzw. $\neg$ stehen für die üblichen logischen Junktoren *UND*, *ODER* bzw. *NICHT*. Ein Boolescher Ausdruck wird nur bei korrekter Verwendung der Klammern „(" und „)" als syntaktisch korrekt angesehen: zu jeder sich schließenden Klammer „)" muß es weiter links in der Zeichenkette eine sich öffnende Klammer „(" geben, und die Anzahlen der sich öffnenden und schließenden Klammern müssen gleich sein; außerdem müssen die Junktoren $\wedge$, $\vee$ bzw. $\neg$ korrekt verwendet werden.

Die Bedeutungen der Junktoren $\wedge$, $\vee$ bzw. $\neg$ ergeben sich aus folgenden Tabellen.

x	$\neg x$
FALSE	TRUE
TRUE	FALSE

$x \vee y$	y = FALSE	y = TRUE
x = FALSE	FALSE	TRUE
x = TRUE	TRUE	TRUE

$x \wedge y$	y = FALSE	y = TRUE
x = FALSE	FALSE	FALSE
x = TRUE	FALSE	TRUE

Unter einem **Literal** innerhalb eines Booleschen Ausdrucks versteht man eine Variable x_i oder eine negierte Variable $\neg x_i$.

Ein Boolescher Ausdruck *F* heißt **erfüllbar**, wenn es eine Ersetzung der Variablen in *F* durch Werte FALSE bzw. TRUE gibt, wobei selbstverständlich gleiche Variablen durch die gleichen Werte ersetzt werden (man sagt: die Variablen werden mit FALSE bzw. TRUE **belegt**), so daß sich bei Auswertung der so veränderten Formel *F* gemäß den Regeln über die Junktoren der Wert TRUE ergibt.

---

[2] Die Syntax eines Booleschen Ausdrucks bzw. einer Formel der Aussagenlogik wird hier als bekannt vorausgesetzt.

Ein Boolescher Ausdruck *F* ist in **konjunktiver Normalform**, wenn *F* die Form
$F = F_1 \wedge ... \wedge F_m$
und jedes F_i die Form
$F_i = \left(y_{i_1} \vee ... \vee y_{i_k}\right)$
hat, wobei y_{i_j} ein Literal oder eine Konstante 0 oder 1 ist, d.h. für eine Variable (d.h. $y_{i_j} = x_l$) oder für eine negierte Variable (d.h. $y_{i_j} = \neg x_l$) oder für eine Konstante 0 bzw. 1 steht.

Die Teilformeln F_i von *F* bezeichnet man als die **Klauseln** (von *F*). Natürlich ist eine Klausel selbst in konjunktiver Normalform.

Eine Klausel F_i von *F* ist durch eine Belegung der in *F* vorkommenden Variablen erfüllt, d.h. besitzt den Wahrheitswert TRUE, wenn mindestens ein Literal in F_i erfüllt ist d.h. den Wahrheitswert TRUE besitzt. *F* ist erfüllt, wenn alle in *F* vorkommenden Klauseln erfüllt sind.

Zu jedem Booleschen Ausdruck *F* gibt es einen Booleschen Ausdruck in konjunktiver Normalform, der genau dann erfüllbar ist, wenn *F* erfüllbar ist.

Nicht jeder Boolesche Ausdruck, selbst wenn er syntaktisch korrekt ist, ist erfüllbar. Beispielsweise ist $x \wedge \neg x$ nicht erfüllbar. Mit Hilfe eines systematischen Verfahrens kann man die Erfüllbarkeit eines Booleschen Ausdrucks *F* mit *n* Variablen dadurch testen, daß man nacheinander alle 2^n möglichen Belegungen der Variablen in *F* mit Wahrheitswerten TRUE bzw. FALSE erzeugt. Immer, wenn eine neue Belegung generiert worden ist, wird diese in die Variablen eingesetzt und der Boolesche Ausdruck ausgewertet. Die Entscheidung, ob *F* erfüllbar ist oder nicht, kann u.U. erst nach Überprüfung aller 2^n Belegungen erfolgen.

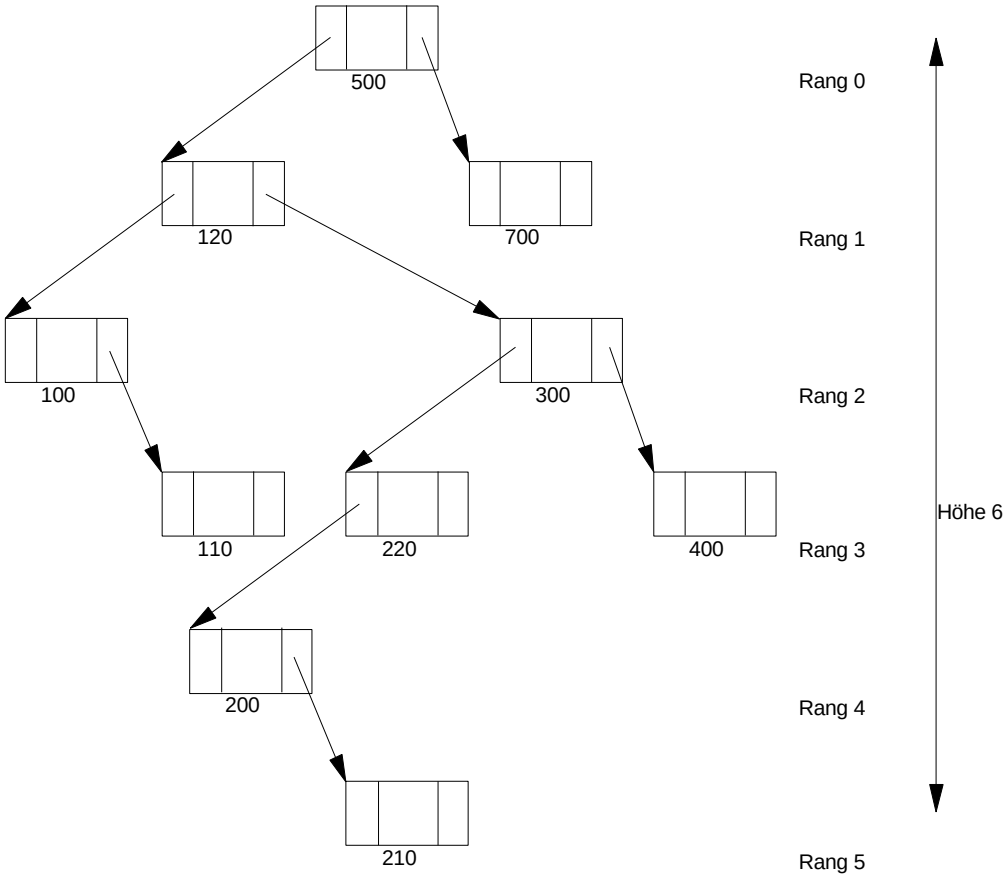
Ein **gerichteter Graph** *G* = (*V*, *E*) besteht aus einer endlichen Menge $V = \{v_1, ..., v_n\}$ von **Knoten** (vertices) und einer endlichen Menge $E = \{e_1, ..., e_k\} \subseteq V \times V$ von **Kanten** (edges).

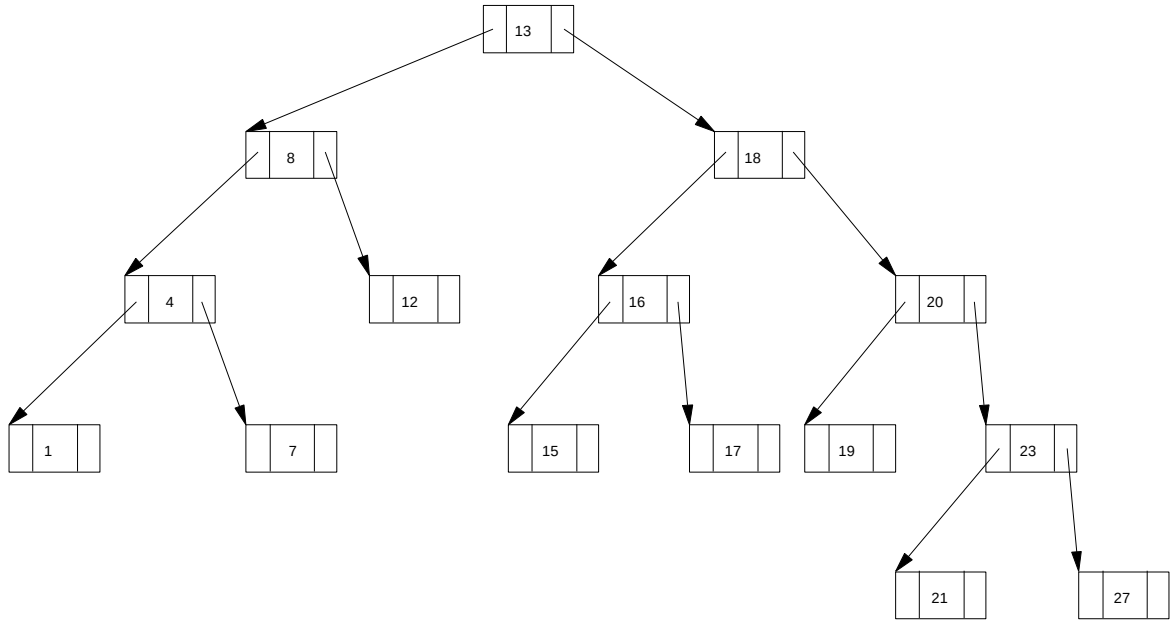
Die Kante $e = (v_i, v_j)$ läuft von v_i nach v_j (verbindet v_i mit v_j). Der Knoten v_i heißt **Anfangsknoten** der Kante $e = (v_i, v_j)$, der Knoten v_j **Endknoten** von $e = (v_i, v_j)$. Zu einem Knoten $v \in V$ heißt $pred(v) = \{v' | (v', v) \in E\}$ die **Menge der direkten Vorgänger** von *v*, $succ(v) = \{v' | (v, v') \in E\}$ die **Menge der direkten Nachfolger** von *v*.

Ein **Binärbaum** $B_n = (V, E)$ mit n Knoten wird durch folgende Eigenschaften charakterisiert:

1. Entweder ist $n \geq 1$ und $|V| = n \geq 1$ und $|E| = n - 1$,
   oder es ist $n = 0$ und $V = E = \varnothing$ (**leerer Baum**).
2. Bei $n \geq 1$ gibt es genau einen Knoten $r \in V$, dessen Menge direkter Vorgänger leer ist; dieser Knoten heißt **Wurzel** von B_n.
3. Bei $n \geq 1$ besteht die Menge der direkten Vorgänger eines jeden Knotens, der nicht die Wurzel ist, aus genau einem Element.
4. Bei $n \geq 1$ besteht die Menge der direkten Nachfolger eines jeden Knotens aus einem Element oder zwei Elementen oder ist leer. Ein Knoten, dessen Menge der direkten Nachfolger leer ist, heißt **Blatt**.

In einem Binärbaum $B = (V, E)$ gibt es für jeden Knoten $v \in V$ genau einen **Pfad** von der Wurzel r zu v, d.h. es gibt eine Folge $\left((a_0, a_1), (a_1, a_2), \ldots, (a_{m-1}, a_m)\right)$ mit $r = a_0$, $v = a_m$ und $(a_{i-1}, a_i) \in E$ für $i = 1, ..., m$. Der Wert m gibt die **Länge des Pfads** an. Um den Knoten v von der Wurzel aus über die Kanten des Pfads zu erreichen, werden m Kanten durchlaufen. Diese Länge wird auch als **Rang des Knotens** v bezeichnet.





Der Rang eines Knotens läßt sich auch folgendermaßen definieren:

(i) Die Wurzel hat den Rang 0.
(ii) Ist *v* ein Knoten im Baum mit Rang $r-1$ und *w* ein direkter Nachfolger von *v*, so hat *w* den Rang *r*.

Unter der **Höhe eines Binärbaums** versteht man den maximal vorkommenden Rang eines Blattes + 1. Die Höhe ist gleich der Anzahl der Knoten, die auf einem Pfad maximaler Länge von der Wurzel zu einem Blatt durchlaufen werden.

In einem Binärbaum bilden alle Knoten mit demselben Rang ein **Niveau des Baums**. Das Niveau 0 eines Binärbaums enthält genau einen Knoten, nämlich die Wurzel. Das Niveau 1 enthält mindestens 1 und höchstens 2 Knoten. Das Niveau *j* enthält höchstens doppelt soviele Knoten wie das Niveau $j-1$. Daher gilt:

Satz 1.1-9:

1. Das Niveau $j \geq 0$ eines Binärbaums enthält mindestens einen und höchstens 2^j Knoten. Die Anzahl der Knoten vom Niveau 0 bis zum Niveau *j* (einschließlich) beträgt mindestens $j+1$ Knoten und höchstens $\sum_{i=0}^{j} 2^j = 2^{j+1} - 1$ Knoten.
2. Ein Binärbaum hat maximale Höhe, wenn jedes Niveau genau einen Knoten enthält. Er hat minimale Höhe, wenn jedes Niveau eine maximale Anzahl von Knoten enthält. Also gilt für die Höhe $h(B_n)$ eines Binärbaums mit *n* Knoten:
   $\lceil \log_2(n+1) \rceil \leq h(B_n) \leq n$.
3. Für die Anzahl $b(h)$ der Blätter eines Binärbaums der Höhe *h* gilt
   $1 \leq b(h) \leq 2^{h-1}$.
4. Für die Mindesthöhe $h(b)$ eines Binärbaums mit $b \geq 1$ vielen Blättern gilt
   $h(b) \geq \lceil \log_2(b) + 1 \rceil$.
5. Die Anzahl (strukturell) verschiedener Binärbäume mit *n* Knoten beträgt
   $\frac{1}{n+1}\binom{2n}{n} = \frac{4^n}{n\sqrt{pn}} + C\frac{4n}{\sqrt{n^5}}$ mit einer reellen Konstanten $C > 0$.
6. Die *mittlere* Anzahl von Knoten, die von der Wurzel aus bis zur Erreichung eines beliebigen Knotens eines Binärbaums mit *n* Knoten (gemittelt über alle *n* Knoten) besucht werden muß, d.h. der *mittlere „Abstand" eines Knotens von der Wurzel*, ist $\sqrt{pn} + C$ mit einer reellen Konstanten $C > 0$. Im günstigsten Fall (wenn also alle Niveaus voll besetzt sind) ist der größte Abstand eines Knotens von der Wurzel in einem Binärbaum mit *n* Knoten gleich $\lceil \log_2(n+1) \rceil \approx \log_2(n)$, im ungünstigsten Fall ist dieser Abstand gleich *n*.

1.2 Problemklassen

In der Informatik beschäftigt man sich häufig damit, Problemstellungen mit Hilfe von Rechnerprogrammen zu lösen. Meist wird man dabei das Programm so entwerfen, daß es nicht nur die Lösung für eine einzige konkrete Problemstellung findet, sondern für eine ganze Klasse von Problemen, die natürlich alle „ähnlich" spezifiziert sind. Beispielsweise wird ein Sortierverfahren in der Lage sein, nicht nur einen Satz von 100 Zahlen zu sortieren, sondern eine beliebige Anzahl von Zahlen, eventuell sogar von feiner strukturierten Objekten.

Ein **Problem** ist eine zu beantwortende Fragestellung, die von **Problemparametern** (Variablenwerten, Eingaben usw.) abhängt, deren genaue Werte in der Problembeschreibung zunächst unspezifiert sind, deren Typen jedoch in die Problembeschreibung eingeht. Ein Problem wird beschrieben durch:

1. eine allgemeine Beschreibung aller Parameter, von der die Problemlösung abhängt; diese Beschreibung spezifiziert die (Problem-) **Instanz (Eingabeinstanz)**

2. die Eigenschaften, die die Antwort, d.h. die Problemlösung, haben soll.

Eine spezielle Problemstellung erhält man durch Konkretisierung einer Probleminstanz, d.h. durch die Angabe spezieller Parameterwerte in der Problembeschreibung.

Im folgenden werden einige grundlegende **Problemtypen** unterschieden und zunächst an Beispielen erläutert.

Problem des Handlungsreisenden als Optimierungsproblem
(minimum traveling salesperson problem)

Instanz: $x = (C, d)$
$C = \{c_1, ..., c_n\}$ ist eine endliche Menge und $d : C \times C \to \mathbf{N}$ eine Funktion.

Die Menge *C* kann beispielsweise als eine Menge von Orten und die Wert $d(c_i, c_j)$ können als Abstand zwischen c_i und c_j interpretiert werden.

Lösung: Eine Permutation (Anordnung) $p : [1:n] \to [1:n]$, d.h. (implizit) eine Anordnung $\langle c_{p(1)}, ..., c_{p(n)} \rangle$ der Werte in *C*, die den Wert

$$\sum_{i=1}^{n} d\left(c_{p(i)}, c_{p(i+1)}\right) + d\left(c_{p(n)}, c_{p(1)}\right)$$

minimiert (unter allen möglichen Permutationen von $[1:n]$).
Dieser Ausdruck gibt die Länge einer „kürzesten Tour“ an, die in $c_{p(1)}$ startet, nacheinander alle Orte einmal besucht und direkt vom letzten Ort $c_{p(n)}$ nach $c_{p(1)}$ zurückkehrt. Der Wert $p(i)$ gibt an, wo man sich im *i*-ten Schritt befindet.

Ein Beispiel einer Instanz ist die Menge $C = \{c_1, ..., c_4\}$ mit den Werten $d(c_1, c_2) = 10$, $d(c_1, c_3) = 5$, $d(c_1, c_4) = 9$, $d(c_2, c_3) = 6$, $d(c_2, c_4) = 9$, $d(c_3, c_4) = 3$ und $d(c_i, c_j) = d(c_j, c_i)$.

Die Permutation $p = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 4 & 3 \end{pmatrix}$, d.h. die Anordnung $\langle c_1, c_2, c_4, c_3 \rangle$ der Werte in *C* ist eine (optimale) Lösung mit Wert 27.

Eine verallgemeinerte Formulierung des Problems des Handlungsreisenden bezogen auf endliche Graphen lautet:

Problem des Handlungsreisenden auf Graphen als Optimierungsproblem

Instanz: $G=(V,E,w)$

$G=(V,E,w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V=\{v_1,...,v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E\subseteq V\times V$; die Funktion $w: E\to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e\in E$ ein nichtnegatives Gewicht.

Lösung: Eine **Tour** durch *G*, d.h. eine geschlossene Kantenfolge

$\langle (v_{i_1},v_{i_2}),(v_{i_2},v_{i_3}),...,(v_{i_{n-1}},v_{i_n}),(v_{i_n},v_{i_1})\rangle$ mit $(v_{i_j},v_{i_{j+1}})\in E$ für $j=1,...,n-1$,

in der jeder Knoten des Graphen (als Anfangsknoten einer Kante) genau einmal vorkommt, die minimale Kosten (unter allen möglichen Touren durch *G*) verursacht. Man kann o.B.d.A. $v_{i_1}=v_1$ setzen.

Die Kosten einer Tour $\langle (v_{i_1},v_{i_2}),(v_{i_2},v_{i_3}),...,(v_{i_{n-1}},v_{i_n}),(v_{i_n},v_{i_1})\rangle$ ist dabei die Summe der Gewichte der Kanten auf der Tour, d.h. der Ausdruck

$$\sum_{j=1}^{n-1} w(v_{i_j},v_{i_{j+1}})+w(v_{i_n},v_{i_1}).$$

Das Problem des Handlungsreisenden findet vielerlei Anwendungen in den Bereichen

- Transportoptimierung:
  Ermittlung einer kostenminimalen Tour, die im Depot beginnt, $n-1$ Kunden erreicht und im Depot endet.
- Fließbandproduktion:
  Ein Roboterarm soll Schrauben an einem am Fließband produzierten Werkstück festdrehen. Der Arm startet in einer Ausgangsposition (über einer Schraube), bewegt sich dann von einer zur nächsten Schraube (insgesamt *n* Schrauben) und kehrt in die Ausgangsposition zurück.
- Produktionsumstellung:
  Eine Produktionsstätte stellt verschiedene Artikel mit denselben Maschinen her. Der Herstellungsprozeß verläuft in Zyklen. Pro Zyklus werden *n* unterschiedliche Artikel produziert. Die Änderungskosten von der Produktion des Artikels v_i auf die des Artikels v_j betragen $w((v_i,v_j))$ (Geldeinheiten). Gesucht wird eine kostenminimale Produktionsfolge. Das Durchlaufen der Kante (v_i,v_j) entspricht dabei der Umstellung von Artikel v_i auf Artikel v_j. Gesucht ist eine Tour (zum Ausgangspunkt zurück), weil die Kosten des nächsten, hier des ersten, Zyklusstarts mit einbezogen werden müssen.

Problem des Handlungsreisenden auf Graphen als Berechnungsproblem

Instanz: $G = (V, E, w)$

$G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

Lösung: Der Wert $\sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}}) + w(v_{i_n}, v_{i_1})$ einer kostenminimalen Tour $\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ durch G.

Zu beachten ist hierbei, daß nicht eine kostenminimale Tour selbst gesucht wird, sondern lediglich der *Wert* einer kostenminimalen Tour. Eventuell ist es möglich, diesen Wert (durch geeignete Argumentationen und Hinweise) zu bestimmen, ohne eine kostenminimale Tour explizit anzugeben. Das Berechnungsproblem scheint daher „einfacher“ zu lösen zu sein als das Optimierungsproblem.

Problem des Handlungsreisenden auf Graphen als Entscheidungsproblem

Instanz: $G = (V, E, w)$ und $K \in \mathbf{R}_{\geq 0}$

$G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

Lösung: Die Antwort auf die Frage:

Gibt es eine kostenminimale Tour $\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ durch G, deren Wert $\leq K$ ist?

Bei dieser Problemstellung ist nicht einmal der Wert einer kostenoptimalen Tour gesucht, sondern lediglich eine Entscheidung, ob dieser Wert „nicht zu groß“, d.h. kleiner als eine vorgegebene Schranke ist. Das Entscheidungsproblem scheint daher „noch einfacher“ zu lösen zu sein als das Optimierungsproblem.

Im vorliegenden Fall des Problems des Handlungsreisenden befindet man sich jedoch im Irrtum: In einem noch zu präzisierenden Sinne sind bei diesem Problem alle Problemvarianten algorithmisch gleich schwierig zu lösen.

Das Beispiel läßt sich verallgemeinern, wobei im folgenden vom (vermeintlich) einfacheren Problemtyp zum komplexeren Problemtyp übergegangen wird.

Die Instanz x eines Problems Π ist eine endliche Zeichenkette über einem endlichen Alphabet Σ_Π, das dazu geeignet ist, derartige Problemstellungen zu formulieren, d.h. $x \in \Sigma_\Pi^*$.

Es werden folgende **Problemtypen** unterschieden:

Entscheidungsproblem Π:

Instanz: $x \in \Sigma_\Pi^*$
und Spezifikation einer Eigenschaft, die einer Auswahl von Elementen aus Σ_Π^* zukommt, d.h. die Spezifikation einer Menge $L_\Pi \subseteq \Sigma_\Pi^*$ mit
$L_\Pi = \{u \in \Sigma^* \mid u \text{ hat die beschriebene Eigenschaft}\}$

Lösung: Entscheidung „ja“, falls $x \in L_\Pi$ ist,
Entscheidung „nein“, falls $x \notin L_\Pi$ ist.

Bemerkung: In konkreten Beispielen für Entscheidungsprobleme wird gelegentlich die Problemspezifikation nicht in dieser strengen formalen Weise durchgeführt. Insbesondere wird die Spezifikation der die Menge $L_\Pi \subseteq \Sigma_\Pi^*$ beschreibenden Eigenschaft direkt bei der Lösung angegeben.

Bei der Lösung eines Entscheidungsproblems geht es also darum, bei Vorgabe einer Instanz $x \in \Sigma_\Pi^*$ zu entscheiden, ob x zur Menge L_Π gehört, d.h. eine genau spezifizierte Eigenschaft, die genau allen Elementen in L_Π zukommt, besitzt, oder nicht.

Es zeigt sich, daß der hier formulierte Begriff der Entscheidbarkeit sehr eng gefaßt ist. Eine erweiterte Definition eines Entscheidungsproblems verlangt bei der Vorgabe einer Instanz $x \in \Sigma_\Pi^*$ nach endlicher Zeit lediglich eine positive Entscheidung „ja“, wenn $x \in L_\Pi$ ist. Ist $x \notin L_\Pi$, so kann die Entscheidung eventuell nicht in endlicher Zeit getroffen werden. Dieser Begriff der Entscheidbarkeit führt auf die rekursiv aufzählbaren Mengen (im Gegensatz zu den entscheidbaren Mengen) und ist Gegenstand von Kapitel 3.

Berechnungsproblem Π:

Instanz: $x \in \Sigma_\Pi^*$
und die Beschreibung einer Funktion $f : \Sigma_\Pi^* \to \Sigma'^*$.

Lösung: Berechnung des Werts $f(x)$.

Optimierungsproblem Π:

Instanz:
1. $x \in \Sigma_\Pi^*$
2. Spezifikation einer Funktion SOL_Π, die jedem $x \in \Sigma_\Pi^*$ **eine Menge zulässiger Lösungen** zuordnet
3. Spezifikation einer **Zielfunktion** m_Π, die jedem $x \in \Sigma_\Pi^*$ und $y \in \mathrm{SOL}_\Pi(x)$ einen Wert $m_\Pi(x, y)$, den **Wert einer zulässigen Lösung**, zuordnet
4. $goal_\Pi \in \{\min, \max\}$, je nachdem, ob es sich um ein Minimierungs- oder ein Maximierungsproblem handelt.

Lösung: $y^* \in \mathrm{SOL}_\Pi(x)$ mit $m_\Pi(x, y^*) = \min\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$ bei einem Minimierungsproblem (d.h. $goal_\Pi = \min$) bzw. $m_\Pi(x, y^*) = \max\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$ bei einem Maximierungsproblem (d.h. $goal_\Pi = \max$).
Der Wert $m_\Pi(x, y^*)$ einer optimalen Lösung wird auch mit $m_\Pi^*(x)$ bezeichnet.

In dieser (formalen) Terminologie wird das Handlungsreisenden-Minimierungsproblem wie folgt formuliert:

Instanz:
1. $G = (V, E, w)$
   $G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus *n* Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht
2. $\mathrm{SOL}(G) = \{T \mid T = \langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle \text{ ist eine Tour durch } G\}$;
   eine Tour durch *G* ist eine geschlossene Kantenfolge, in der jeder Knoten des Graphen (als Anfangsknoten einer Kante) genau einmal vorkommt
3. für $T \in \mathrm{SOL}(G)$, $T = \langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$, ist die Zielfunktion

definiert durch $m(G,T) = \sum_{j=1}^{n-1} w\left(v_{i_j}, v_{i_{j+1}}\right) + w\left(v_{i_n}, v_{i_1}\right)$

4. $goal = \min$

Lösung: Eine Tour $T^* \in \mathrm{SOL}(G)$, die minimale Kosten (unter allen möglichen Touren durch G) verursacht, und $m(G,T^*)$.

Im folgenden werden zunächst hauptsächlich Entscheidungsprobleme (kommt einem Wort $x \in \Sigma_\Pi^*$ eine spezifische Eigenschaft zu, d.h. gilt $x \in L_\Pi$ für eine Sprache $L_\Pi \subseteq \Sigma_\Pi^*$?) und Berechnungsprobleme (man berechne den Wert $f(x)$ für eine Funktion $f : \Sigma_\Pi^* \to \Sigma'^*$) behandelt. Diese Probleme sollen algorithmisch gelöst werden. Dazu müssen die Begriffe der Entscheidbarkeit und der Berechenbarkeit präziser gefaßt und geeignete Modelle Berechnungsmodelle definiert werden.

1.3 Ein intuitiver Algorithmusbegriff

Ein **Algorithmus** ist eine Verfahrensvorschrift (Prozedur, Berechnungsvorschrift), die aus einer endlichen Menge eindeutiger Regeln besteht, die eine endliche Aufeinanderfolge von Operationen spezifiziert, so daß eine Lösung zu einem Problem bzw. einer spezifischen Klasse von Problemen daraus erzielt wird.

Konkret kann man sich einen Algorithmus als ein Computerprogramm vorstellen, das in einer **Pascal-ähnlichen Programmiersprache** formuliert ist. Darunter versteht man Programmiersprachen, die

- Deklarationen von Variablen (mit geeigneten Datentypen) zulassen
- die üblichen arithmetischen Operationen mit Konstanten und Variablen und Wertzuweisungen an Variablen enthalten
- Kontrollstrukturen wie Sequenz (Hintereinanderreihung von Anweisungen, blockstrukturierte Anweisungen, Prozeduren), Alternativen (**IF** ... **THEN** ... **ELSE, CASE ... END**) und WHILE-Schleifen (**WHILE** ... **DO** ...) besitzen.

Von einem Algorithmus erwartet man eine Reihe von **Eigenschaften**, damit er als „effektives Rechenverfahren“ gelten kann:

1. Die Verfahrensvorschrift (das Programm) soll aus einem endlichen Text bestehen.

2. Der Ablauf einer Berechnung soll schrittweise als Folge elementarer Rechenschritte erfolgen.

3. Das Verfahren soll **deterministisch** sein, d.h. in jedem Stadium einer Berechnung soll vollständig und eindeutig bestimmt sein, welcher elementare Rechenschritt als nächster getan wird. Ein Text „Bei Eingabe von x kann man für $f(x)$ einen von endlich vielen Werten aussuchen“ ist als Teil eines Algorithmus nicht zulässig.

4. Das Verfahren soll abgeschlossen sein, d.h. welcher Rechenschritt als nächster getan wird, soll ausschließlich von den Eingabewerten und den vorangegangenen berechneten Zwischenergebnissen abhängen. Ein Text „ $f(x) = x$, wenn es gerade regnet bzw. = $2x$ sonst “ ist als Teil eines Algorithmus nicht zulässig.

5. Das Verfahren soll im Prinzip beliebig große Zahlen handhaben können.

Typische **Fragestellungen** bei einem gegebenen Algorithmus für eine Problemlösung sind:

- Hält der Algorithmus immer bei einer gültigen Eingabe nach endlich vielen Schritten an?

- Berechnet der Algorithmus bei einer gültigen Eingabe eine korrekte Antwort?

Die positive Beantwortung beider Fragen erfordert einen mathematischen **Korrektheitsbeweis** des Algorithmus. Bei positiver Beantwortung nur der zweiten Frage spricht man von **partieller Korrektheit**. Für die partielle Korrektheit ist lediglich nachzuweisen, daß der Algorithmus bei einer gültigen Eingabe, bei der er nach endlich vielen Schritten anhält, ein korrektes Ergebnis liefert.

- Wieviele Schritte benötigt der Algorithmus bei einer gültigen Eingabe **höchstens (worst case analysis)** bzw. **im Mittel (average case analysis)**, d.h. welche **(Zeit-) Komplexität** hat er im schlechtesten Fall bzw. im Mittel? Dabei ist es natürlich besonders interessant nachzuweisen, daß die Komplexität des Algorithmus von der jeweiligen Formulierungsgrundlage (Programmiersprache, Maschinenmodell) weitgehend unabhängig ist.

Entsprechend kann man nach dem benötigten **Speicherplatzbedarf (Platzkomplexität)** eines Algorithmus fragen.

Die Beantwortung dieser Fragen für den schlechtesten Fall gibt **obere Komplexitätsschranken** (Garantie für das Laufzeitverhalten bzw. den Speicherplatzbedarf) an.

- Gibt es zu einer Problemlösung eventuell ein „noch besseres“ Verfahren (mit weniger Rechenschritten, weniger Speicherplatzbedarf)? Wieviele Schritte wird jeder Algorithmus mindestens durchführen, der das vorgelegte Problem löst?

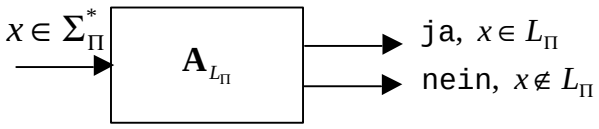
  Die Beantwortung dieser Frage liefert untere Komplexitätsschranken.

Entsprechend der verschiedenen Problemtypen (Entscheidungsproblem, Berechnungsproblem, insbesondere Optimierungsproblem) gibt es auch unterschiedliche **Typen von Algorithmen**:

Ein **Algorithmus $\mathbf{A}_{L_\Pi}$ für ein Entscheidungsproblem** Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ hat die Schnittstellen und die Form

Eingabe: $x \in \Sigma_\Pi^*$

Ausgabe: `ja` (accept), falls $x \in L_\Pi$ gilt
`nein` (reject), falls $x \notin L_\Pi$ gilt.



Bei Eingabe von $x \in \Sigma_\Pi^*$ wird mit $\mathbf{A}_{L_\Pi}(x)$, $\mathbf{A}_{L_\Pi}(x) \in \{$ `ja, nein` $\}$, die Entscheidung von $\mathbf{A}_{L_\Pi}$ bezeichnet.

Es gilt also für $x \in \Sigma_\Pi^*$:
Es ist $x \in L_\Pi$ genau dann, wenn $\mathbf{A}_{L_\Pi}$ bei Eingabe von $x \in \Sigma_\Pi^*$ mit $\mathbf{A}_{L_\Pi}(x) =$ `ja` stoppt.

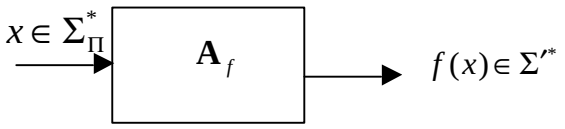
Wie bereits in Kapitel 1.2 bemerkt, erweist sich der definierte Begriff des Entscheidungsalgorithmus als zu eng gefaßt, wenn man fordert, daß der Algorithmus bei jeder Eingabe $x \in \Sigma_\Pi^*$ stoppt (entweder mit Antwort `ja` oder `nein`). Es werden daher auch Entscheidungsprobleme algorithmisch behandelt, die Verfahren zulassen, die nicht bei jeder Eingabe anhal-

ten. Bei Eingabe von $x \in \Sigma_\Pi^*$ stoppt der Algorithmus mit Antwort `ja`, falls $x \in L_\Pi$ ist. Für $x \notin L_\Pi$ wird zugelassen, daß der Algorithmus entweder mit Antwort `nein` stoppt oder nicht terminiert. Dieser Algorithmusbegriff führt auf den Begriff der **rekursiv aufzählbaren Mengen**.

Ein **Algorithmus $\mathbf{A}_f$ für ein Berechnungsproblem** Π (Berechnung einer Funktion $f : \Sigma_\Pi^* \to \Sigma'^*$) hat die Schnittstellen und die Form

Eingabe: $x \in \Sigma_\Pi^*$

Ausgabe: $f(x) \in \Sigma'^*$



Mit $\mathbf{A}_f(x)$ wird das Berechnungsergebnis von $\mathbf{A}_f$ bei Eingabe von $x \in \Sigma_\Pi^*$ bezeichnet, d.h. $\mathbf{A}_f(x) = f(x)$. Falls $\mathbf{A}_f$ bei einer Eingabe $x \in \Sigma_\Pi^*$ nicht anhält, ist $f(x)$ nicht definiert. In diesem Fall berechnet $\mathbf{A}_f$ eine partielle Funktion.

Mit diesen sehr „vagen“ Begriff der Berechenbarkeit läßt sich bereits zeigen, daß es Funktionen gibt, die nicht berechenbar sind. Genauer:

Satz 1.3-1:
Es gibt Funktionen $g : \mathbf{N} \to \{0,1\}$, die nicht berechenbar sind.

Beweis:

Wieder wird die Diagonalisierungstechnik verwendet (vgl. den Beweis von Satz 1.1-3):
Jedes Berechnungsverfahren ist ein Algorithmus, der mit Hilfe eines endlichen Alphabets Σ formuliert werden kann. Die Menge *B* aller Berechnungsverfahren für Funktionen der Form $g : \mathbf{N} \to \{0,1\}$ ist daher eine Teilmenge von Σ^* und damit abzählbar (unendlich). Es gibt daher eine bijektive Funktion $f : \mathbf{N} \to B$, d.h. $B = \{ f(0), f(1), f(2), ... \}$. Die folgende Argumentation wird klarer, wenn man *B* in der Form $B = \{ f_0, f_1, f_2, ... \}$ darstellt, d.h. die Nummer einer Funktion in *B* als Index angibt. Es wird eine Funktion $g_0 : \mathbf{N} \to \{0,1\}$ wie folgt definiert:

$$g_0 : \begin{cases} \mathbf{N} & \rightarrow & \{0,1\} \\ n & \rightarrow & \begin{cases} 0 & \text{falls } f_n(n) = 1 \\ 1 & \text{falls } f_n(n) = 0 \end{cases} \end{cases} \text{ ist.}$$

Angenommen, die Funktion g_0 komme in *B* vor. Dann hat g_0 eine Nummer $i_g \in \mathbf{N}$ in der Aufzählung von *B*, also $g_0 = f_{i_g}$. Nun gibt es zwei Möglichkeiten: entweder $g_0(i_g) = 0$ oder $g_0(i_g) = 1$.

Ist $g_0(i_g) = 0$, dann ist (nach Definition von g_0) $f_{i_g}(i_g) = 1$. Wegen $g_0 = f_{i_g}$ entsteht der Widerspruch $0 = g_0(i_g) = f_{i_g}(i_g) = 1$.

Ist $g_0(i_g) = 1$, dann ist (nach Definition von g_0) $f_{i_g}(i_g) = 0$. Wieder ergibt sich ein Widerspruch, nämlich $1 = g_0(i_g) = f_{i_g}(i_g) = 0$.

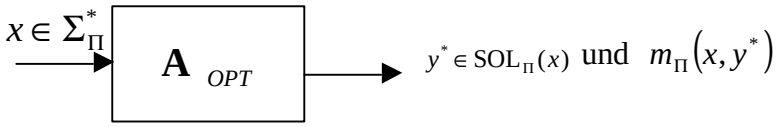
Daher ist $g_0 \notin B$ und damit eine nichtberechenbare Funktion der Form $g : \mathbf{N} \rightarrow \{0,1\}$. ///

Der Vollständigkeit halber soll noch die Form eines Algorithmus zur Lösung eines Optimierungsproblems, wie es in Kapitel 1.2 definiert wurde, angeführt werden:

Ein **Algorithmus $\mathbf{A}_{OPT}$ für ein Optimierungsproblem** Π mit Zielfunktion m_Π und Optimierungsziel $goal_\Pi \in \{\min, \max\}$ hat die Schnittstellen und die Form

Eingabe: $x \in \Sigma_\Pi^*$

Ausgabe: $y^* \in \mathrm{SOL}_\Pi(x)$ und $m_\Pi(x, y^*) = goal_\Pi\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$



Mit $\mathbf{A}_{OPT}(x)$ wird die von $\mathbf{A}_{OPT}$ bei Eingabe von $x \in \Sigma_\Pi^*$ ermittelte Lösung bezeichnet, d.h. $\mathbf{A}_{OPT}(x) = (y^*, m_\Pi(x, y^*))$. Hier wird implizit vorausgesetzt,daß für $x \in \Sigma_\Pi^*$ die Menge $\mathrm{SOL}_\Pi(x)$ nichtleer ist.

2 Modelle der Berechenbarkeit

In diesem Kapitel werden zwei unterschiedliche Ansätze beschrieben, die den Begriff der Berechenbarkeit mathematisch exakt formulieren: das Modell der Turingmaschine und das Modell der Random Access Maschine. Beide Modelle sind „theoretische“ Modelle, da man weder eine Turingmaschine noch eine Random Access Maschine (wegen der Modellierung des eingesetzten Speichers) physisch bauen kann. Sie ähneln jedoch sehr der Architektur heutiger Rechner.

Beide Modelle haben ihre Stärken. Die Turingmaschine ist Basismodell in der Automatentheorie, die ihrerseits eng verknüpft mit der Theorie der Formalen Sprachen ist. Diese wiederum hat erst die Grundlage dazu gelegt, daß wir heute über Programmiersprachen und schnelle Compiler und über mächtige Modellierungswerkzeuge in der Anwendungsentwicklung verfügen. Die Random Access Maschine ist ein Modell eines Rechners einschließlich einer sehr einfachen Assemblersprache, so daß es eher einen mechanischen Zugang zum Begriff der Berechenbarkeit liefert. Es stellt sich heraus, daß die Modelle äquivalent in dem Sinne sind, daß sie in der Lage sind, die gleiche Menge von Funktionen zu berechnen bzw. die gleichen Mengen zu entscheiden (in einem noch zu präzisierenden Sinn).

Ein weiterer Ansatz, der jedoch auch hier nicht vertieft wird, beschäftigt sich damit, eine Programmiersprache auf ihre minimale Anzahl möglicher Anweisungstypen zu reduzieren. Auch hier stellt sich heraus, daß die Berechnungsfähigkeit dieses Modells mit der einer Turingmaschine äquivalent ist.

Auch weitere hier nicht behandelte Ansätze wie die eher mathematisch ausgerichteten Theorien der m-rekursiven Funktionen oder des Churchsche l-Kalküls haben bisher keinen formalen umfassenderen Berechenbarkeitsbegriff erzeugt. Daher wird die als **Churchsche These** bekannte Aussage als gültig angesehen, nach der das im folgenden behandelte Modell der Turingmaschine die Formalisierung des Begriffs „Algorithmus“ darstellt.

2.1 Deterministische Turingmaschinen

Das hier beschriebene Modell zur Berechenbarkeit wurde 1936 von Alan Turing (1912 – 1954) vorgestellt, und zwar noch vor der Entwicklung der Konzepte moderner Computer. Grundlage ist dabei die Vorstellung, daß eine Berechnung mit Hilfe eines mechanischen Verfahrens durchgeführt wird, wobei Zwischenergebnisse auf einem Rechenblatt notiert und später wieder verwendet werden können. Die seinem Initiator zu Ehren genannte Turingmaschine stellt ein *theoretisches Modell* zur Beschreibung der Berechenbarkeit dar und ist trotz seiner Ähnlichkeit mit heutigen Computern hardwaremäßig *nicht* realisierbar, da es über ei-

nen abzählbar unendlich großen Speicher verfügt. Es hat wesentlichen Einfluß sowohl auf die mathematische Logik als auch auf die Entwicklung heutiger Rechner genommen.

Eine **deterministische *k*-Band-Turingmaschine (*k*-DTM)** *TM* ist definiert durch
$TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$
mit:

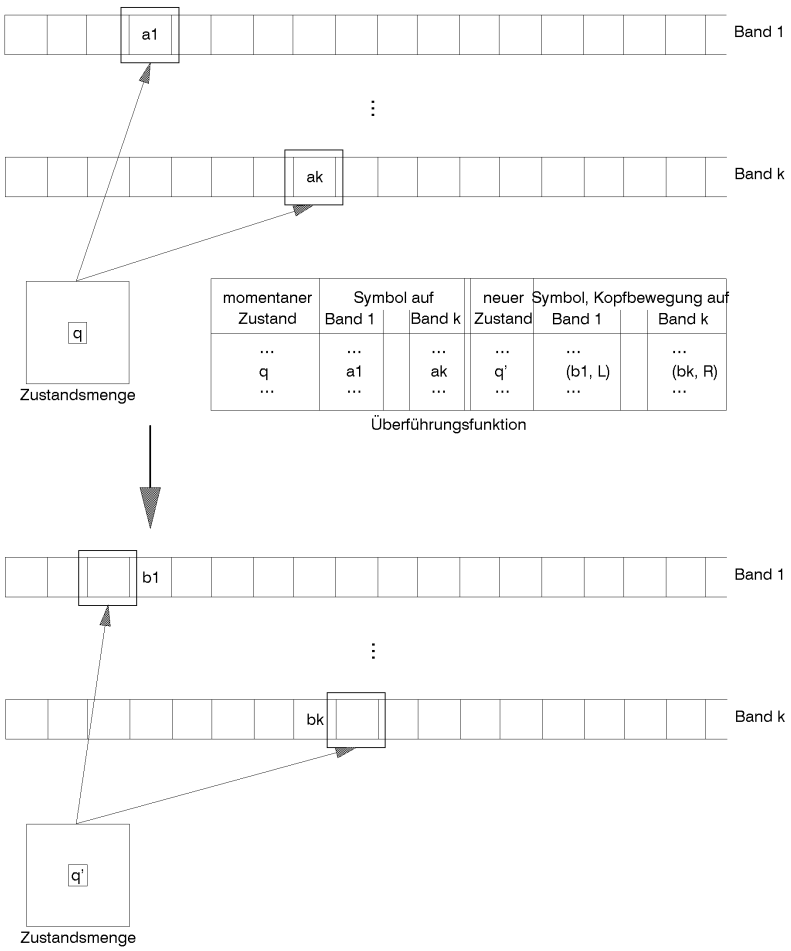
1. Q ist eine endliche nichtleere Menge: die **Zustandsmenge**
2. Σ ist eine endliche nichtleere Menge: das **Arbeitsalphabet**; Σ enthält alle Zeichen, die in Feldern der Bänder (siehe unten) stehen können
3. $I \subseteq \Sigma$ ist eine endliche nichtleere Menge: das **Eingabealphabet**; mit den Zeichen aus I werden die Wörter der Eingabe gebildet
4. $b \in \Sigma \setminus I$ ist das **Leerzeichen (Blankzeichen)**
5. $q_0 \in Q$ ist der **Anfangszustand (**oder **Startzustand)**
6. $q_{accept} \in Q$ ist der **akzeptierende Zustand (Endzustand)**
7. $d : (Q \setminus \{q_{accept}\}) \times \Sigma^k \rightarrow Q \times (\Sigma \times \{L, R, S\})^k$ ist eine partielle Funktion, die **Überführungsfunktion**; insbesondere ist $d(q, a_1, \ldots, a_k)$ für $q = q_{accept}$ nicht definiert; zu beachten ist, daß d für einige weitere Argumente eventuell nicht definiert ist.

Anschaulich kann man sich die **Arbeitsweise einer *k*-DTM** wie folgt vorstellen:

Es gibt *k* (Speicher-) **Bänder**, die jeweils in Zellen eingeteilt sind und nach rechts unendlich lang sind. Jede Zelle eines jeden Bands kann einen Buchstaben des Arbeitsalphabets aufnehmen. Eine Zelle, die das Leerzeichen enthält, wird als **leere Zelle** bezeichnet. Für jedes Band gibt es genau einen **Schreib/Lesekopf**, der jeweils genau über einer Zelle steht. Dieser kann den Zellinhalt lesen, neu beschreiben und zur linken oder rechten Nachbarzelle übergehen oder auf der Zelle stehenbleiben. Welche Aktion der jeweilige Schreib/Lesekopf unternimmt, wird in der Überführungsfunktion d in Abhängigkeit vom Zustand (des Steuerwerks) und der auf den Bändern gelesenen Zeichen angegeben.

Die Arbeitsweise der *k*-DTM wird durch ein **endliches Steuerwerk** festgelegt, dessen Zustandsüberführungen **getaktet** ablaufen und durch d beschrieben werden:

Liest der Kopf des *i*-ten Bandes den Buchstaben $a_i \in \Sigma$ (i = 1, ..., k), ist das Steuerwerk im Zustand q und ist
$d(q, a_1, \ldots, a_k) = (q', (b_1, d_1), \ldots, (b_k, d_k))$,
dann geht das Steuerwerk in den Zustand q' über, der *i*-te Kopf schreibt b_i und geht zur linken Nachbarzelle für $d_i = L$ (falls dieses möglich ist), zur rechten Nachbarzelle für $d_i = R$ oder bleibt für $d_i = S$ über der Zelle stehen.



Ist $d(q,a_1,...,a_k)$ nicht definiert, so **hält** die *k*-DTM **im Zustand** q **an** (**stoppt im Zustand** q). Sobald also der akzeptierende Zustand q_{accept} erreicht wird, hält die Turingmaschine an, da $d(q,a_1,...,a_k)$ für $q = q_{accept}$ nicht definiert ist. Sie kann aber auch eventuell vorher in einem anderen Zustand anhalten (nämlich dann, wenn $d(q,a_1,...,a_k)$ nicht definiert ist), oder sie **kann beliebig lange weiterlaufen (sie hält nicht an)**. Diese Situation tritt beispielsweise dann ein, wenn die Turingmaschine in einen Zustand $q \neq q_{accept}$ kommt und $d(q,a_1,...,a_k) = (q,(a_1,S),...,(a_k,S))$ ist. Eine andere Möglichkeit eines endlosen Weiterlaufens ergibt sich dann, wenn die Turingmaschine in einen Zustand $q \neq q_{accept}$ kommt, alle Köpfe über Zellen stehen, die das Leerzeichen b enthalten, rechts dieser Zellen auf allen Bändern nur noch Leerzeichen stehen und $d(q,b,...,b) = (q,(b,R),...,(b,R))$ ist.

Die Werte $d(q,a_1,...,a_k) = (q',(b_1,d_1),...,(b_k,d_k))$ der Überführungsfunktion werden häufig in Form einer endlichen Tabelle angegeben:

momentaner Zustand	Symbol unter dem Schreib/Lesekopf auf			neuer Zustand	neues Symbol, Kopfbewegung auf		
	Band 1	...	Band *k*		Band 1	...	Band *k*
q	a_1	...	a_k	q'	b_1, d_1	...	b_k, d_k

Eine **Konfiguration** *K* einer *k*-DTM *TM* beschreibt den gegenwärtigen Gesamtzustand von *TM*, d.h. den gegenwärtigen Zustand zusammen mit den Bandinhalten und den Positionen der Schreib/Leseköpfe:
$K = (q,(a_1,i_1),...,(a_k,i_k))$ mit $q \in Q$, $a_j \in \Sigma^*$, $i_j \geq 1$ für j = 1, ..., *k*.

Diese Konfiguration *K* wird folgendermaßen interpretiert:

TM ist im Zustand *q*, das *j*-te Band (für *j* = 1, ..., *k*) enthält linksbündig die endliche Zeichenkette a_j (jeder Buchstabe von a_j belegt eine Zelle), gefolgt von Zellen, die das Leerzeichen enthalten (leere Zellen), der Schreib/Lesekopf des *j*-ten Bands steht über der i_j-ten Zelle; für $i_j \in [1:|a_j|]$ ist dieses der i_j-teBuchstaben von a_j, für $i_j \geq |a_j|+1$ steht der Schreib/Lesekopf hinter a_j über einer Zelle, die das Leerzeichen enthält.

Ist a_j der i_j-te Buchstabe von a_j bzw. $a_j = b$ für $i_j \geq |a_j|+1$, und ist
$d(q,a_1,...,a_k) = (q',(b_1,d_1),...,(b_k,d_k))$,
dann geht die *TM* in die **Folgekonfiguration** K' über, die durch
$K' = (q',(b_1,i_1'),...,(b_k,i_k'))$
definiert wird. Dabei entsteht b_j aus a_j durch Ersetzen von a_j durch b_j. Die Positionen i_j' der Schreib/Leseköpfe der Folgekonfiguration K' lauten

$$i_j' = \begin{cases} i_j + 1 & \text{für } d_j = R \\ i_j & \text{für } d_j = S \\ i_j - 1 & \text{für } d_j = L \text{ und } i_j \geq 2. \end{cases}$$

Man schreibt in diesem Fall
$K \Rightarrow K'$.
Die Bezeichnung $K \Rightarrow^* K'$ besagt, daß entweder keine Konfigurationsänderung stattgefunden hat (es ist dann $K = K'$) oder daß es eine Konfiguration K_1 gibt mit $K \Rightarrow K_1$ und $K_1 \Rightarrow^* K'$ (auch geschrieben als $K \Rightarrow K_1 \Rightarrow^* K'$).

Man schreibt $K \Rightarrow^m K'$ mit $m \in \mathbf{N}$, wenn K' aus K durch m Konfigurationsänderungen hervorgegangen ist, d.h. wenn es m Konfigurationen K_1, ..., K_m gibt mit

$K \Rightarrow K_1 \Rightarrow ... \Rightarrow K_m = K'$.

Für $m = 0$ ist dabei $K = K'$.

Eine Konfiguration K_0, die den Anfangszustand q_0 und auf dem 1. Band ein Wort $w \in I^*$ enthält, wobei sich auf allen anderen Bändern nur Leerzeichen befinden und die Köpfe über den am weitesten links stehenden Zellen stehen, d.h. eine Konfiguration der Form

$K_0 = (q_0, (w,1), (e,1), ..., (e,1))$,

heißt **Anfangskonfiguration mit Eingabewort** *w*. Da $w \in I^*$ ist und das Leerzeichen nicht zu *I* gehört, kann *TM* (bei entsprechender Definition der Überführungsfunktion) das Ende von *w*, nämlich das erste Leerzeichen im Anschluß an *w*, erkennen. Im folgenden wird gelegentlich für eine Eingabe $w \in I^*$ auch $w \in \Sigma^*$ geschrieben und dabei implizit angenommen, daß *w* nur Buchstaben aus *I* enthält.

Eine Konfiguration K_{accept}, die den akzeptierenden Zustand q_{accept} enthält, d.h. eine Konfiguration der Form

$K_{accept} = (q_{accept}, (a_1, i_1), ..., (a_k, i_k))$,

heißt **akzeptierende Konfiguration** (**Endkonfiguration)**.

Ein Wort *w* über dem Eingabealphabet wird von *TM* **akzeptiert**, wenn gilt:

$(q_0, (w,1), (e,1), ..., (e,1)) \Rightarrow^* K_{accept}$

mit einer Endkonfiguration K_{accept}.

Die von einer *k*-DTM *TM* **akzeptierte Sprache** ist die Menge

$$L(TM) = \{ w \mid w \in I^* \text{ und } w \text{ wird von } TM \text{ akzeptiert} \}$$
$$= \{ w \mid w \in I^* \text{ und } (q_0, (w,1), (e,1), ..., (e,1)) \Rightarrow^* (q_{accept}, (a_1, i_1), ..., (a_k, i_k)) \}.$$

Zu beachten ist, daß *TM* eventuell auch dann bereits eine Endkonfiguration erreicht, wenn das Eingabewort *w* noch gar nicht komplett gelesen ist. Auch in diesem Fall gehört *w* zu *L*(*TM*).

Für $w \notin L(TM)$ hält *TM* entweder nicht im Zustand q_{accept}, oder *TM* läuft unendlich lange weiter, d.h. die Überführungsfolge $K_0 \Rightarrow K'$ läßt sich unendlich lang fortsetzen, ohne daß eine Konfiguration erreicht wird, die den Endzustand enthält.

Eine 2-DTM Turingmaschine zur Akzeptanz von

$L(TM) = \{ w \mid w \in \{0,1\}^+$ und w ist die Binärdarstellung einer geraden Zahl $\} \cup \{e\}$

Die 2-DTM $TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$ wird gegeben durch
$Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1, b\}$, $I = \{0,1\}$, $q_{accept} = q_1$ mit der Überführungsfunktion d, die durch folgende Tabelle definiert ist:

momentaner Zustand	Symbol unter dem Schreib/Lesekopf auf		neuer Zustand	neues Symbol, Kopfbewegung auf	
	Band 1	Band 2		Band 1	Band 2
q_0	b	b	q_1	b, S	1, S
	0	b	q_0	0, R	b, S
	1	b	q_2	1, R	b, S
q_2	b	b	q_2	b, R	b, S
	0	b	q_0	0, R	b, S
	1	b	q_2	1, R	b, S

TM stoppt bei Eingabe von *w* nicht, falls *w* die Binärdarstellung einer ungeraden Zahl ist.

Eine 2-DTM Turingmaschine zur Akzeptanz der Palindrome über { 0, 1}

Die folgende Turingmaschine 2-DTM *TM* akzeptiert genau die Palindrome über {0, 1}:
$L(TM) = \{ w \mid w \in \{0,1\}^+$ und $w = a_1 ... a_{n-1} a_n = a_n a_{n-1} ... a_1 \} \cup \{e\}$.

$TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$ mit $Q = \{q_0, ..., q_5\}$, $\Sigma = \{0, 1, b, \#\}$, $I = \{0,1\}$, $q_{accept} = q_5$.

TM arbeitet wie folgt:

1. Die 1. Zelle des 2. Bandes wird mit # markiert. Dann wird das Eingabewort auf das 2. Band kopiert; der Kopf des 1. Bandes steht jetzt unmittelbar rechts des Eingabeworts.
2. Der Kopf des 2. Bandes wird bis zum Zeichen # zurückgesetzt.
3. Der Kopf des 1. Bandes wird jeweils um 1 Zelle nach links und der Kopf des 2. Bandes um 1 Zelle nach rechts verschoben. Wenn die von den Köpfen jeweils gelesenen Symbole sämtlich übereinstimmen, ist das Eingabenwort ein Palindrom, und die Maschine geht in den Zustand $q_f = q_5$ und stoppt. Sonst stoppt die Maschine in einem von q_f verschiedenen Zustand.

Die Überführungsfunktion wird durch folgende Tabelle gegeben:

momentaner Zustand	Symbol unter dem Schreib/Lesekopf auf		neuer Zustand	neues Symbol, Kopfbewegung auf	
	Band 1	Band 2		Band 1	Band 2
q_0	0	b	q_1	0, S	#, R
	1	b	q_1	1, S	#, R
	b	b	q_{accept}	b, S	b, S
q_1	0	b	q_1	0, R	0, R
	1	b	q_1	1, R	1, R
	b	b	q_2	b, S	b, L
q_2	b	0	q_2	b, S	0, L
	b	1	q_2	b, S	1, L
	b	#	q_3	b, L	#, R
q_3	0	0	q_4	0, S	0, R
	1	1	q_4	1, S	1, R
q_4	0	0	q_3	0, L	0, S
	0	1	q_3	0, L	1, S
	1	0	q_3	1, L	0, S
	1	1	q_3	1, L	1, S
	0	b	q_{accept}	0, S	b, S
	1	b	q_{accept}	1, S	b, S

Beobachtet man die Arbeitsweise einer Turingmaschine *TM* bei einem Eingabewort *w*, so liegt nach endlich vielen Überführungen eine der folgenden Situationen vor:

1. Fall: *TM* ist in einem Zustand $q \neq q_{accept}$ stehengeblieben (d.h. $d(q,...)$ ist nicht definiert). Dann ist $w \notin L(TM)$.

2. Fall: *TM* ist im Zustand q_{accept} stehengeblieben. Dann ist $w \in L(TM)$.

2. Fall: *TM* ist im Zustand q_{accept} stehengeblieben. Dann ist $w \in L(TM)$.

3. Fall: *TM* ist noch nicht stehengeblieben, d.h. *TM* befindet sich in einem Zustand $q \neq q_{accept}$, für den $d(q,...)$ definiert ist. Dann ist noch nicht entschieden, ob $w \in L(TM)$ oder

$w \notin L(TM)$ gilt. *TM* ist eventuell noch nicht lange genug beobachtet worden. Es ist nicht „vorhersagbar“, wie lange *TM* beobachtet werden muß, um eine Entscheidung zu treffen (es ist **algorithmisch unentscheidbar**).

Eine Turingmaschine *TM* kann als **Berechnungsvorschrift** definiert werden: Das 1. Band wird als **Eingabeband** und ein Band, etwa das *k*-te Band, als **Ausgabeband** ausgezeichnet. Die Turingmaschine *TM* **berechnet eine partielle Funktion** $f_{TM} : I^* \to \Sigma^*$, wenn gilt:

Startet *TM* im Anfangszustand mit *w*, d.h. startet *TM* mit *w* auf dem Eingabeband im Zustand q_0 (alle anderen Bänder sind leer), und stoppt *TM* nach endlich vielen Schritten im akzeptierenden Zustand q_{accept}, dann wird die Bandinschrift *y* des Ausgabebands von *TM* als Funktionswert $y = f_{TM}(w)$ interpretiert. Falls *TM* überhaupt nicht oder nicht im Endzustand stehenbleibt, dann ist $f_{TM}(w)$ nicht definiert. Daher ist f_{TM} eine partielle Funktion.

Eine 2-DTM zur Berechnung der (totalen) Funktion $f : \begin{cases} \mathbf{N} & \to & \mathbf{N} \\ n & \to & n+1 \end{cases}$

Die folgende 2-DTM berechnet die (totale) Funktion $f : \begin{cases} \mathbf{N} & \to & \mathbf{N} \\ n & \to & n+1 \end{cases}$:

Hierbei wird ein $n \in \mathbf{N}$ als Zeichenkette in seiner Binärdarstellung auf das Eingabeband geschrieben; die höchstwertige Stelle steht dabei ganz links. Der Wert 0 wird als Zeichenkette 0 eingegeben, alle anderen Werte $n \in \mathbf{N}$ ohne führende Nullen. Entsprechend steht abschließend *n* + 1 in seiner Binärdarstellung ohne führende Nullen auf dem Ausgabeband (2. Band).

$TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$ mit $Q = \{q_0, ..., q_{19}\}$, $\Sigma = \{0, 1, b, \#\}$, $I = \{0, 1\}$, $q_{accept} = q_{19}$.

TM arbeitet wie folgt:

1. Auf das 2. Band wird zunächst das Zeichen # geschrieben, das das linke Ende des 2. Bandes markieren soll. Dann wird auf das 2. Band eine 0 geschrieben und das Eingabewort *w* auf das 2. Band kopiert (auf dem 2. Band steht jetzt #0*w*, d.h. das Eingabewort *w* ist durch eine führende 0 ergänzt worden).
2. Auf dem 2. Band werden von rechts alle 1‘en in 0‘en invertiert, bis die erste 0 erreicht ist; diese wird durch 1 ersetzt (Addition *n* := *n* + 1).
3. Der Kopf des 2. Bandes wird auf die Anfangsmarkierung # zurückgesetzt und geprüft, ob rechts dieses Zeichens eine 0 steht (das bedeutet, daß die anfangs an *w* angefügte führende 0 wieder entfernt werden muß).

4. In diesem Fall wird der Inhalt des 2. Bandes komplett um zwei Position nach links verschoben. Dadurch werden die führende 0 und das Zeichen # auf dem 2. Band entfernt. Es ist zu beachten, daß am rechten Ende des 2. Bandes 2 Leerzeichen gesetzt werden.
5. Andernfalls wird der Inhalt des 2. Bandes um 1 Position nach links verschoben und dadurch das Zeichen # entfernt.

Die Überführungsfunktion wird durch folgende Tabelle gegeben:

momentaner Zustand	Symbol unter dem Schreib/Lesekopf auf		neuer Zustand	neues Symbol, Kopfbewegung auf		Bemerkung
	Band 1	Band 2		Band 1	Band 2	
q_0	0	*b*	q_1	0, *S*	#, *R*	linkes Ende für *n*+1
	1	*b*	q_1	1, *S*	#, *R*	markieren
q_1	0	*b*	q_2	0, *S*	0, *R*	führende 0 erzeugen
	1	*b*	q_2	1, *S*	0, *R*	
q_2	0	*b*	q_2	0, *R*	0, *R*	Inhalt des Eingabe-
	1	*b*	q_2	1, *R*	1, *R*	bandes auf 2. Band
	b	*b*	q_3	*b*, *S*	*b*, *L*	kopieren
q_3	*b*	0	q_4	*b*, *S*	1, *S*	1 addieren:
	b	1	q_3	*b*, *S*	0, *L*	anhängende 1‘en invertieren, erste 0 von rechts invertieren
q_4	*b*	0	q_4	*b*, *S*	0, *L*	auf # zurückgehen
	b	1	q_4	*b*, *S*	1, *L*	
	b	#	q_5	*b*, *S*	#, *R*	
q_5	*b*	0	q_6	*b*, *S*	0, *R*	muß führende 0
	b	1	q_{15}	*b*, *S*	1, *S*	entfernt werden?
q_6	*b*	0	q_7	*b*, *S*	0, *L*	gelesenes Zeichen
	b	1	q_8	*b*, *S*	1, *L*	im Zustand merken
	b	*b*	q_{13}	*b*, *S*	*b*, *L*	
q_7	*b*	0	q_9	*b*, *S*	0, *L*	2. Kopf um 1 Positi-
	b	1	q_9	*b*, *S*	1, *L*	on nach links setzen
q_8	*b*	0	q_{10}	*b*, *S*	0, *L*	2. Kopf um 1 Positi-
	b	1	q_{10}	*b*, *S*	1, *L*	on nach links setzen

../..

q_9	b	0	q_{11}	b, S	$0, R$	0 schreiben
	b	1	q_{11}	b, S	$0, R$	
	b	#	q_{11}	b, S	$0, R$	
q_{10}	b	0	q_{11}	b, S	$1, R$	1 schreiben
	b	1	q_{11}	b, S	$1, R$	
	b	#	q_{11}	b, S	$1, R$	
q_{11}	b	0	q_{12}	b, S	$0, R$	2. Kopf um 1 Positi-
	b	1	q_{12}	b, S	$1, R$	on nach rechts set- zen
q_{12}	b	0	q_6	b, S	$0, R$	2. Kopf 1 weitere
	b	1	q_6	b, S	$1, R$	Position nach rechts setzen
q_{13}	b	0	q_{14}	b, S	b, L	letztes Zeichen
	b	1	q_{14}	b, S	b, L	löschen und nach links gehen
q_{14}	b	0	$q_{19} = q_{accept}$	b, S	b, S	letztes Zeichen
	b	1	$q_{19} = q_{accept}$	b, S	b, S	löschen und STOP
q_{15}	b	0	q_{16}	b, S	$0, L$	gelesenes Zeichen
	b	1	q_{17}	b, S	$1, L$	im Zustand merken
	b	b	q_{14}	b, S	b, L	
q_{16}	b	0	q_{18}	b, S	$0, R$	0 schreiben
	b	1	q_{18}	b, S	$0, R$	
	b	#	q_{18}	b, S	$0, R$	
q_{17}	b	0	q_{18}	b, S	$1, R$	1 schreiben
	b	1	q_{18}	b, S	$1, R$	
	b	#	q_{18}	b, S	$1, R$	
q_{18}	b	0	q_{15}	b, S	$0, R$	2. Kopf um 1 Posi-
	b	1	q_{15}	b, S	$1, R$	tion nach rechts set- zen

Die Konzepte der Berechnung partieller Funktionen und das Akzeptieren von Sprachen mittels Turingmaschinen sind äquivalent:

Satz 2.1-1:

Berechnet die Turingmaschine *TM* die partielle Funktion $f_{TM} : I^* \to \Sigma^*$, dann kann man eine Turingmaschine TM' konstruieren mit $L(TM') = \{ w\# y \mid y = f_{TM}(w) \}$.

Akzeptiert die Turingmaschine *TM* die Sprache $L(TM)$, dann läßt sich *TM* so modifizieren, daß sie die partielle Funktion $f_{TM} : I^* \to \Sigma^*$ berechnet, die definiert ist durch $f_{TM}(w) = y$ genau dann, wenn gilt:

$$(q_0,(w,1),(e,1),...,(e,1),(e,1)) \Rightarrow^* (q_{accept},(a_1,i_1),(a_2,i_2),...,(a_{k-1},i_{k-1})(y,|y|+1))$$

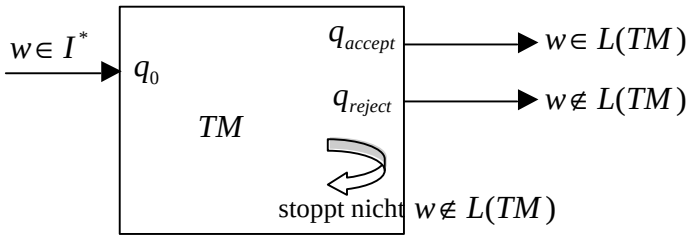
Die Aussage läßt die Interpretation zu, daß die Verifikation eines Funktionsergebnisses genauso komplex ist wie die Berechnung des Funktionsergebnisses selbst.

Die Überführungsfunktion d einer Turingmaschine *TM* werde folgendermaßen modifiziert: Die Zustandsmenge *Q* wird um einen neuen Zustand q_{reject} erweitert. Für alle bisherigen Zustände $q \in Q$ mit $q \neq q_{accept}$, für die $d(q,...)$ nicht definiert ist, werden in d die Zeilen

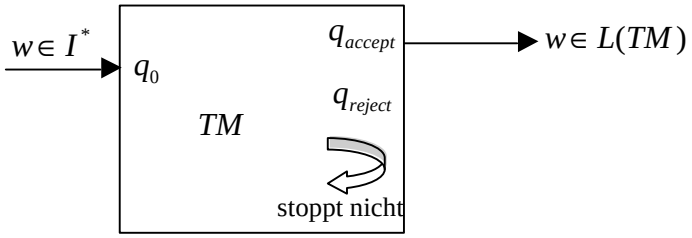
$d(q, a_1, ..., a_k) = (q_{reject}, (a_1, S), ..., (a_k, S))$ mit $a_i \in \Sigma$ für $i = 1, ..., k$

aufgenommen. Für q_{reject} ist d nicht definiert. Kommt die so modifizierte Turingmaschine bei Eingabe eines Wortes $w \in I^*$ in einen Zustand $q \in Q$ mit $q \neq q_{accept}$, für die $d(q,...)$ bisher nicht definiert war (es ist dann $w \notin L(TM)$), so macht die modifizierte Turingmaschine noch eine Überführung, ohne die Bandinhalte und die Positionen der Köpfe zu ändern, und stoppt im Zustand q_{reject}, ohne das Wort *w* zu akzeptieren. Man kann daher annehmen, daß eine Turingmaschine *TM* so definierte Zustände q_{accept} und q_{reject} enthält. Der Zustand q_{accept} heißt **akzeptierender Zustand**, der Zustand q_{reject} heißt **nicht-akzeptierender (verwerfender) Zustand**. Startet *TM* bei Eingabe eines Wortes $w \in I^*$ im Anfangszustand q_0, so zeigt sie folgendes Verhalten: Entweder kommt *TM* in den Zustand q_{accept}, dann ist $w \in L(TM)$ und **die Eingabe** *w* **wird akzeptiert**; oder *TM* kommt in den Zustand q_{reject}, dann ist $w \notin L(TM)$ und **die Eingabe** *w* **wird verworfen**; oder *TM* läuft unendlich lange weiter, dann ist ebenfalls $w \notin L(TM)$. Man kann *TM* daher als Blackbox darstellen, die eine Eingangsschnittstelle besitzt, über die im Anfangszustand q_0 ein Wort $w \in I^*$ eingegeben wird und der Start der Berechnung gemäß der Überführungsfunktion d erfolgt. Sie besitzt zwei „aktivierbare“ Ausgangsschnittstellen: Die erste Ausgangsschnittstelle wird von *TM* dann aktiviert, wenn *TM* bei der Berechnung den Zustand q_{accept} erreicht und stoppt; es ist dann $w \in L(TM)$. Die zweite Ausgangsschnittstelle wird von *TM* aktiviert, wenn *TM* bei der Berechnung den Zustand q_{reject} erreicht und stoppt; es ist dann $w \notin L(TM)$. Wird keine Ausgangsschnittstelle aktiviert,

weil *TM* nicht anhält, dann ist ebenfalls $w \notin L(TM)$. Eine graphische Repräsentation von *TM* sieht wie folgt aus.

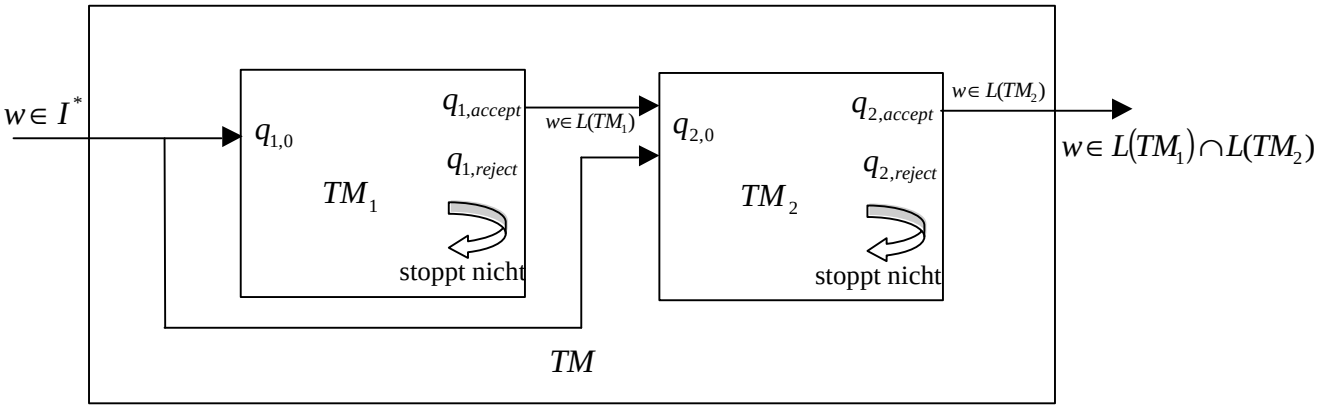


Da die Turingmaschine *TM* dazu verwendet wird, um festzustellen, ob ein Eingabewort $w \in I^*$ von ihr akzeptiert wird, kann man *TM* vereinfacht auch folgendermaßen darstellen:



Turingmaschinen können zu neuen Turingmaschinen zusammengesetzt werden: Es seien zwei Turingmaschinen $TM_1 = (Q_1, \Sigma_1, I, d_1, b, q_{1,0}, q_{1,accept})$ und $TM_2 = (Q_2, \Sigma_2, I, d_2, b, q_{2,0}, q_{2,accept})$ mit disjunkten Zustandsmengen ($Q_1 \cap Q_2 = \varnothing$) und demselben Eingabealphabet $I \subseteq \Sigma_1$ und $I \subseteq \Sigma_2$ und jeweils getrennten Bändern gegeben. Die Anzahl der Bänder von TM_1 sei k_1, die von TM_2 sei k_2. Jeweils das 1. Band ist das Eingabeband. Beispiele für die Möglichkeiten der **Zusammensetzung** sind:

- Man kann eine neue Turingmaschine *TM* durch **Hintereinanderschaltung** von TM_1 und TM_2 konstruieren, die die Bänder von TM_1 und TM_2 umfaßt, d.h. $k_1 + k_2$ viele Bänder besitzt, und folgendermaßen arbeitet: Eine Eingabe $w \in I^*$ für die zusammengesetzte Turingmaschine *TM* wird auf das 1. Band von TM_1 gegeben und auf das 1. Band von TM_2 kopiert. Jetzt wird zunächst das Verhalten von TM_1 auf *w* simuliert. Falls TM_1 das Wort *w* akzeptiert, d.h. in den Zustand wird $q_{1,accept}$ gelangt, wird das Verhalten von TM_2 auf *w* simuliert. *TM* akzeptiert das Wort *w*, falls TM_2 das Wort *w* akzeptiert. Es gilt: $L(TM) = L(TM_1) \cap L(TM_2)$. Graphisch läßt sich *TM* wie folgt darstellen.



Die formale Notation von *TM* mit Hilfe der Überführungsfunktion ist etwas mühsam:

Es ist $TM = (Q, \Sigma, I, d, b, q_{1,0}, q_{2,accept})$. Dabei ist
$Q = Q_1 \cup Q_2 \cup \{q_{copy}, q_{skip}\}$ die Zustandsmenge von *TM* mit zwei neuen Zuständen q_{copy} und q_{skip}, die nicht in $Q_1 \cup Q_2$ enthalten sind,
$\Sigma = \Sigma_1 \cup \Sigma_2 \cup \{\#\}$ das Arbeitsalphabet von *TM* mit einem neuen Symbol #, das nicht in $\Sigma_1 \cup \Sigma_2$ enthalten ist,
$d : (Q \setminus \{q_{2,accept}\}) \times \Sigma^{k_1+k_2} \to Q \times (\Sigma \times \{L, R, S\})^{k_1+k_2}$ die Überführungsfunktion, die sich aus den Überführungsfunktionen d_1 und d_2 wie folgt ergibt:

Die Bänder von *TM* werden von 1 bis $k_1 + k_2$ numeriert; die ersten k_1 Bänder sind die ursprünglichen Bänder von TM_1. Insbesondere ist das Eingabeband von *TM* das ursprüngliche Eingabeband von TM_1. Das Band mit der Nummer $k_1 + 1$ ist das ursprüngliche Eingabeband von TM_2. Ein Eingabewort $w \in I^*$ wird auf das erste Band von *TM* gegeben. Es sei $n = |w|$. In die erste Zelle des Bands mit der Nummer $k_1 + 1$ wird das Zeichen # geschrieben und *w* auf dieses Band kopiert. Das Zeichen # dient dazu, das linke Ende des Bands mit der Nummer $k_1 + 1$ zu erkennen. Nach diesem Kopiervorgang stehen die Köpfe des ersten und des $(k_1 + 1)$-ten Bands jeweils über der (*n*+1)-ten Zelle, alle übrigen Köpfe wurden nicht bewegt. Das Ende des Kopiervorgangs wurde dadurch erkannt, daß auf dem ersten Band das Leerzeichen gelesen wurde. Nun werden die Köpfe des ersten und des $(k_1 + 1)$-ten Bands beide zurück auf das erste Zeichen von *w* gesetzt.

Für den Kopier- und den Rücksetzvorgang der Köpfe werden folgende Zeilen in die Überführungsfunktion d von *TM* aufgenommen:

$$d\left(q_{1,0}, \underbrace{a,b,...,b}_{k_1}, \underbrace{b,...,b}_{k_2}\right) = \left(q_{copy}, \underbrace{(a,S),(b,S),...,(b,S)}_{k_1}, \underbrace{(\#,R),(b,S),...,(b,S)}_{k_2}\right) \quad \text{für alle}$$

$a \in \Sigma_1$ (b ist das Leerzeichen),

$$d\left(q_{copy}, \underbrace{a,b,...,b}_{k_1}, \underbrace{b,...,b}_{k_2}\right) = \left(q_{copy}, \underbrace{(a,R),(b,S),...,(b,S)}_{k_1}, \underbrace{(a,R),(b,S),...,(b,S)}_{k_2}\right) \quad \text{für alle}$$

$a \in I$ (man beachte, daß a = Leerzeichen hierbei nicht vorkommt),

$$d\left(q_{copy}, \underbrace{b,b,...,b}_{k_1}, \underbrace{b,...,b}_{k_2}\right) = \left(q_{skip}, \underbrace{(b,S),(b,S),...,(b,S)}_{k_1}, \underbrace{(b,L),(b,S),...,(b,S)}_{k_2}\right),$$

$$d\left(q_{skip}, \underbrace{a',b,...,b}_{k_1}, \underbrace{a,b,...,b}_{k_2}\right) = \left(q_{skip}, \underbrace{(a',L),(b,S),...,(b,S)}_{k_1}, \underbrace{(a,L),(b,S),...,(b,S)}_{k_2}\right) \text{ für alle}$$

$a' \in I \cup \{b\}$ und $a \in I$,

$$d\left(q_{skip}, \underbrace{a',b,...,b}_{k_1}, \underbrace{\#,b,...,b}_{k_2}\right) = \left(q_{1,0}, \underbrace{(a',S),(b,S),...,(b,S)}_{k_1}, \underbrace{(\#,R),(b,S),...,(b,S)}_{k_2}\right) \text{ für alle}$$

$a' \in I \cup \{b\}$.

Nun wird das Verhalten von TM_1 simuliert, wobei die Inhalte der Bänder mit den Nummern k_1+1 bis k_1+k_2 (entsprechend den ursprünglichen Bändern von TM_2) und die Positionen der Köpfe auf diesen Bändern nicht verändert werden. Die dafür erforderlichen Zeilen in der Überführungsfunktion d lauten:

$$d\left(q_1, \underbrace{a_1,a_2,...,a_{k_1}}_{k_1}, \underbrace{a,b,...,b}_{k_2}\right) = \left(q_1', \underbrace{(b_1,d_1),(b_2,d_2),...,(b_{k_1},d_{k_1})}_{k_1}, \underbrace{(a,S),(b,S),...,(b,S)}_{k_2}\right) \text{für}$$

jeden Eintrag $d_1(q_1,a_1,a_2,...,a_{k_1}) = (q_1',(b_1,d_1),(b_2,d_2),...,(b_{k_1},d_{k_1}))$ in der Überführungsfunktion d_1 von TM_1 und $a \in \Sigma_1$.

Falls bei der Simulation der akzeptierende Zustand $q_{1,accept}$ von TM_1 erreicht wird, d.h. $w \in L(TM_1)$, wird die Simulation des Verhaltens von TM_2 auf w gestartet. Auf den ersten k_1 ändert sich dabei nichts mehr. Folgende Zeilen werden in die Überführungsfunktion d aufgenommen:

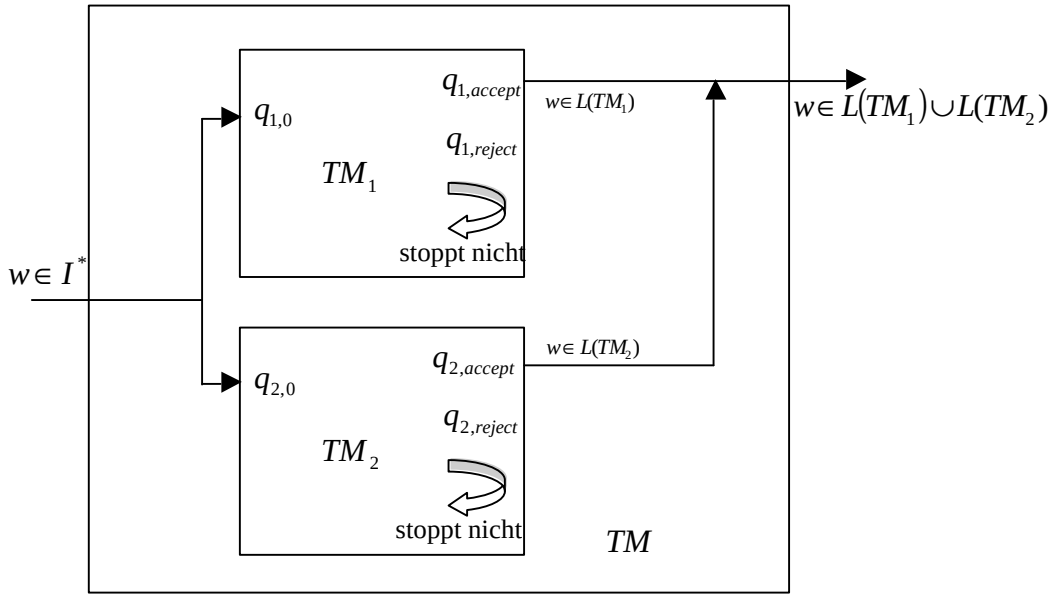
$$d\left(q_{1,accept}, \underbrace{a_1,a_2,...,a_{k_1}}_{k_1}, \underbrace{a,b,...,b}_{k_2}\right) = \left(q_{2,0}, \underbrace{(a_1,S),(a_2,S),...,(a_{k_1},S)}_{k_1}, \underbrace{(a,S),(b,S),...,(b,S)}_{k_2}\right)$$

mit $a_1 \in \Sigma_1$, ..., $a_{k_1} \in \Sigma_1$ und $a \in \Sigma_1$ und

$$d\left(q_2, \underbrace{a_1, a_2, ..., a_{k_1}}_{k_1}, \underbrace{g_1, g_2, ..., g_{k_2}}_{k_2} \right) = \left(q_2', \underbrace{(a_1, S), (a_2, S), ..., (a_{k_1}, S)}_{k_1}, \underbrace{(h_1, d_1), (h_2, d_2), ..., (h_{k_2}, d_{k_2})}_{k_2} \right)$$

für jeden Eintrag $d_2(q_2, g_1, g_2, ..., g_{k_2}) = (q_2', (h_1, d_1), (h_2, d_2), ..., (h_{k_2}, d_{k_2}))$ in der Überführungsfunktion d_2 von TM_2 und $a_1 \in \Sigma_1$, ..., $a_{k_1} \in \Sigma_1$.

- Durch **Parallelschaltung** kann man aus TM_1 und TM_2 eine neue Turingmaschine *TM* bilden: ein Wort *w* wird sowohl auf das 1. Band von TM_1 als auch auf das 1. Band von TM_2 gegeben und das Verhalten beider Turingmaschinen simultan simuliert. Sobald eine der beiden Turingmaschinen *w* akzeptiert, wird *w* von *TM* akzeptiert. Es gilt:
  $L(TM) = L(TM_1) \cup L(TM_2)$.



Auch hier ist die formale Notation von *TM* mit Hilfe der Überführungsfunktion etwas mühsam:

Es ist $TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$. Dabei ist
$Q = (Q_1 \times Q_2) \cup \{(q_{copy}, q_{2,0}), (q_{skip}, q_{2,0}), (q_{acc}, q_{acc})\}$ die Zustandsmenge von *TM* mit zwei neuen Zuständen q_{copy} und q_{skip}, die nicht in Q_1 enthalten sind, und einem neuen Zustand q_{acc}, der nicht in $Q_1 \cup Q_2$ enthalten ist (die Zustände bestehen jetzt aus Paaren von Zuständen der Turingmaschinen TM_1 und TM_2 und drei neuen Zuständen),
$\Sigma = \Sigma_1 \cup \Sigma_2 \cup \{\#\}$ das Arbeitsalphabet von *TM* mit einem neuen Symbol #, das nicht in Σ_2 enthalten ist,

$d : (Q \setminus \{(q_{acc}, q_{acc})\}) \times \Sigma^{k_1+k_2} \to Q \times (\Sigma \times \{L, R, S\})^{k_1+k_2}$ die Überführungsfunktion, die sich aus den Überführungsfunktionen d_1 und d_2 ergibt (siehe unten),
$q_0 = (q_{1,0}, q_{2,0})$ der Anfangszustand und
$q_{accept} = (q_{acc}, q_{acc})$ der akzeptierende Zustand von *TM*.

Die Bänder von *TM* werden wieder von 1 bis $k_1 + k_2$ numeriert; die ersten k_1 Bänder sind die ursprünglichen Bänder von TM_1. Insbesondere ist das Eingabeband von *TM* das ursprüngliche Eingabeband von TM_1. Das Band mit der Nummer $k_1 + 1$ ist das ursprüngliche Eingabeband von TM_2. Ein Eingabewort $w \in I^*$ wird auf das erste Band von *TM* gegeben. Wieder wird in die erste Zelle des Bands mit der Nummer $k_1 + 1$ das Zeichen # geschrieben und *w* auf dieses Band kopiert. Anschließend werden der Kopf des ersten Bands auf den Bandanfang und der Kopf des $(k_1 + 1)$-ten Bands auf die zweite Zelle (rechte Nachbarzelle der Zelle, die das Zeichen # enthält) zurückgesetzt. Der Vorgang verläuft ähnlich dem Vorgang, der bei der Hintereinanderschaltung von TM_1 und TM_2 beschrieben wurde. Die erforderlichen Einträge in der Überführungsfunktion lauten jetzt:

$$d\left((q_{1,0}, q_{2,0}), \underbrace{a, b, ..., b}_{k_1}, \underbrace{b, ..., b}_{k_2}\right) = \left((q_{copy}, q_{2,0}), \underbrace{(a, S), (b, S), ..., (b, S)}_{k_1}, \underbrace{(\#, R), (b, S), ..., (b, S)}_{k_2}\right)$$

für alle $a \in \Sigma_1$ (*b* ist das Leerzeichen),

$$d\left((q_{copy}, q_{2,0}), \underbrace{a, b, ..., b}_{k_1}, \underbrace{b, ..., b}_{k_2}\right) = \left((q_{copy}, q_{2,0}), \underbrace{(a, R), (b, S), ..., (b, S)}_{k_1}, \underbrace{(a, R), (b, S), ..., (b, S)}_{k_2}\right)$$

für alle $a \in I$ (man beachte, daß a = Leerzeichen hierbei nicht vorkommt),

$$d\left((q_{copy}, q_{2,0}), \underbrace{b, b, ..., b}_{k_1}, \underbrace{b, ..., b}_{k_2}\right) = \left((q_{skip}, q_{2,0}), \underbrace{(b, S), (b, S), ..., (b, S)}_{k_1}, \underbrace{(b, L), (b, S), ..., (b, S)}_{k_2}\right)$$

$$d\left((q_{skip}, q_{2,0}), \underbrace{a', b, ..., b}_{k_1}, \underbrace{a, b, ..., b}_{k_2}\right) = \left((q_{skip}, q_{2,0}), \underbrace{(a', L), (b, S), ..., (b, S)}_{k_1}, \underbrace{(a, L), (b, S), ..., (b, S)}_{k_2}\right)$$

für alle $a' \in I \cup \{b\}$ und $a \in I$,

$$d\left((q_{skip}, q_{2,0}), \underbrace{a', b, ..., b}_{k_1}, \underbrace{\#, b, ..., b}_{k_2}\right) = \left((q_{1,0}, q_{2,0}), \underbrace{(a', S), (b, S), ..., (b, S)}_{k_1}, \underbrace{(\#, R), (b, S), ..., (b, S)}_{k_2}\right)$$

für alle $a' \in I \cup \{b\}$.

Nun wird parallel das Verhalten von TM_1 und von TM_2 simuliert, wobei für die Simulation von TM_1 nur auf die Inhalte der Bänder mit den Nummern 1 bis k_1 (entsprechend den ursprünglichen Bändern von TM_1) und für die Simulation von TM_2 nur auf die Inhalte der Bänder mit den Nummern $k_1 + 1$ bis $k_1 + k_2$ (entsprechend den ursprünglichen

Bändern von TM_2) zugegriffen wird. Die dafür erforderlichen Zeilen in der Überführungsfunktion d lauten:

$$d\left((q_1, q_2), \underbrace{a_1, a_2, ..., a_{k_1}}_{k_1}, \underbrace{g_1, g_2, ..., g_{k_2}}_{k_2} \right)$$

$$= \left((q_1', q_2'), \underbrace{(b_1, d_1), (b_2, d_2), ..., (b_{k_1}, d_{k_1})}_{k_1}, \underbrace{(h_1, e_1), (h_2, e_2), ..., (h_{k_2}, h_{k_2})}_{k_2} \right),$$

falls $d_1(q_1, a_1, a_2, ..., a_{k_1}) = (q_1', (b_1, d_1), (b_2, d_2), ..., (b_{k_1}, d_{k_1}))$ in der Überführungsfunktion d_1 von TM_1 und $d_2(q_2, g_1, g_2, ..., g_{k_2}) = (q_2', (h_1, e_1), (h_2, e_2), ..., (h_{k_2}, e_{k_2}))$ in der Überführungsfunktion d_2 von TM_2 ist.

Falls bei der Simulation ein Zustand der Form $(q_{1,reject}, q_2)$ mit $q_2 \in Q_2$ bzw. der Form $(q_1, q_{2,reject})$ mit $q_1 \in Q_1$ erreicht wird, muß sichergestellt werden, daß die Simulation von TM_2 bzw. von TM_1 weiterläuft. Daher werden auch folgende Einträge in die Überführungsfunktion d aufgenommen:

$$d\left((q_{1,reject}, q_2), \underbrace{a_1, a_2, ..., a_{k_1}}_{k_1}, \underbrace{g_1, g_2, ..., g_{k_2}}_{k_2} \right)$$

$$= \left((q_{1,reject}, q_2'), \underbrace{(a_1, S), (a_2, S), ..., (a_{k_1}, S)}_{k_1}, \underbrace{(h_1, e_1), (h_2, e_2), ..., (h_{k_2}, e_{k_2})}_{k_2} \right)$$

mit $a_1 \in \Sigma_1$, ..., $a_{k_1} \in \Sigma_1$ und falls $d_2(q_2, g_1, g_2, ..., g_{k_2}) = (q_2', (h_1, e_1), (h_2, e_2), ..., (h_{k_2}, e_{k_2}))$ in der Überführungsfunktion d_2 von TM_2 ist und

$$d\left((q_1, q_{2,reject}), \underbrace{a_1, a_2, ..., a_{k_1}}_{k_1}, \underbrace{g_1, g_2, ..., g_{k_2}}_{k_2} \right)$$

$$= \left((q_1', q_{2,reject}), \underbrace{(b_1, d_1), (b_2, d_2), ..., (b_{k_1}, d_{k_1})}_{k_1}, \underbrace{(g_1, S), (g_2, S), ..., (g_{k_2}, S)}_{k_2} \right)$$

für jeden Eintrag $d_1(q_1, a_1, a_2, ..., a_{k_1}) = (q_1', (b_1, d_1), (b_2, d_2), ..., (b_{k_1}, d_{k_1}))$ in der Überführungsfunktion d_1 von TM_1 und $g_1 \in \Sigma_2$, ..., $g_{k_2} \in \Sigma_2$.

Schließlich soll *TM* die Eingabe *w* akzeptieren, wenn TM_1 oder TM_2 die Eingabe akzeptiert. Folgende Einträge werden daher in die Überführungsfunktion von *TM* aufgenommen:

$$d\left(\left(q_{1,accept}, q_2 \right), \underbrace{a_1, a_2, ..., a_{k_1}}_{k_1}, \underbrace{g_1, g_2, ..., g_{k_2}}_{k_2} \right)$$

$$= \left(\left(q_{acc}, q_{acc} \right), \underbrace{(a_1, S), (a_2, S), ..., (a_{k_1}, S)}_{k_1}, \underbrace{(g_1, S), (g_2, S), ..., (g_{k_2}, S)}_{k_2} \right) \text{ und}$$

$$d\left(\left(q_1, q_{2,accept} \right), \underbrace{a_1, a_2, ..., a_{k_1}}_{k_1}, \underbrace{g_1, g_2, ..., g_{k_2}}_{k_2} \right)$$

$$= \left(\left(q_{acc}, q_{acc} \right), \underbrace{(a_1, S), (a_2, S), ..., (a_{k_1}, S)}_{k_1}, \underbrace{(g_1, S), (g_2, S), ..., (g_{k_2}, S)}_{k_2} \right)$$

für alle $q_1 \in Q_1$, $q_2 \in Q_2$, $a_1 \in \Sigma_1$, ..., $a_{k_1} \in \Sigma_1$ und $g_1 \in \Sigma_2$, ..., $g_{k_2} \in \Sigma_2$.

- Dadurch, daß man die Turingmaschine TM_2 als Unterprogramm in die Berechnung der Turingmaschine TM_1 einsetzt, erhält man eine neue Turingmaschine *TM*, die prinzipiell folgendermaßen abläuft: Eine Eingabe $w \in I^*$ wird in TM_1 eingegeben. Ein Band von TM_1 wird als „Parameterübergabeband“ für TM_2 ausgezeichnet. Sobald TM_1 in einen als Parameterübergabezustand ausgezeichneten Zustand $q_?$ gelangt, wird der Inhalt des Parameterübergabebands auf das erste Band von TM_2 kopiert. Wird diese Eingabe von TM_2 akzeptiert, wobei bei Akzeptanz der Inhalt des k_2-ten Bands *y* lautet, wird das Parameterübergabeband gelöscht, *y* darauf kopiert, das erste Band von TM_2 gelöscht und die Berechnung von TM_1 fortgesetzt. Die Ausformulierung der Details dieser Konstruktion werden dem Leser als Übung überlassen.

Obige Überlegungen und Ausführungen in Kapitel 3.2 zeigen:

Satz 2.1-2:

Die Klasse der von Turingmaschinen akzeptierten Mengen ist gegenüber Vereinigungsbildung und Schnittbildung abgeschlossen.

Die Klasse der von Turingmaschinen akzeptierten Mengen ist gegenüber Komplementbildung nicht abgeschlossen: Wenn $L \subseteq \Sigma^*$ von einer Turingmaschine *TM* akzeptiert wird, d.h. $L = L(TM)$, muß das Komplement $\Sigma^* \setminus L$ von *L* nicht auch notwendigerweise von einer Turingmaschine akzeptiert werden.

Für eine *k*-DTM $TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$ und eine Eingabe $w \in L(TM)$ gelte
$(q_0, (w,1), (e,1), ..., (e,1)) \Rightarrow^m K_{accept}$
mit einer Endkonfiguration $K_{accept} = (q_{accept}, (a_1, i_1), ..., (a_k, i_k))$. Dann wird durch $t_{TM}(w) = m$ eine partielle Funktion $t_{TM} : \Sigma^* \to \mathbf{N}$ definiert, die angibt, **wieviele Überführungen *TM* macht, um *w* zu akzeptieren**. Es handelt sich hierbei um eine partielle Funktion, da t_{TM} für Wörter $w \notin L(TM)$ nicht definiert ist.

Die **Zeitkomplexität** von *TM* (**im schlechtesten Fall, worst case**) wird definiert durch die partielle Funktion $T_{TM} : \mathbf{N} \to \mathbf{N}$ mit
$T_{TM}(n) = \max\{ t_{TM}(w) \mid w \in \Sigma^* \text{ und } |w| \le n \}$.

Bemerkung: In der Literatur findet man auch die Definition
$$T_{TM}(n) = \max\{ t_{TM}(w) \mid w \in \Sigma^* \text{ und } |w| = n \}$$

Entsprechend kann man die **Platzkomplexität** von *TM* (**im schlechtesten Fall, worst case**) $S_{TM}(n)$ als den maximalen Abstand vom linken Ende eines Bandes definieren, den ein Schreib/Lesekopf bei der Akzeptanz eines Worts $w \in L(TM)$ mit $|w| \le n$ erreicht.

Beispiele:

Die oben angegebene Turingmaschine zur Akzeptanz der Palindrome über $\{0,1\}$ hat eine Zeitkomplexität der Größe $T_{TM}(n) = 4n + 3$ und eine Platzkomplexität $S_{TM}(n) = n + 2$.

Die oben angegebene Turingmaschine zur Berechnung der Funktion $f : \begin{cases} \mathbf{N} & \to & \mathbf{N} \\ n & \to & n+1 \end{cases}$ hat folgende Zeitkomplexität: Der Wert n wird in Binärdarstellung auf das Eingabeband gegeben. Die Eingabe $w = bin(n)$ belegt daher für $n \ge 1$ $|w| = |bin(n)| = \lfloor \log_2(n) \rfloor + 1$ viele Zeichen bzw. ein Zeichen für $n = 0$. Dann führt die Turingmaschine $O(|w|) = O(\log(n))$ viele Schritte aus. Die Zeitkomplexität ist linear in der Länge der Eingabe.

Bei der Betrachtung der Zeitkomplexität werden in diesem Beispiel also die **Bitoperationen** gezählt, die benötigt werden, um $f(n)$ zu berechnen, wenn *n* in Binärdarstellung gegeben ist. Die Turingmaschine dieses Beispiels läßt sich leicht zu einer Turingmaschine modifizieren, die die Funktion $SUM : \begin{cases} \mathbf{N} \times \mathbf{N} & \to & \mathbf{N} \\ (n, m) & \to & n + m \end{cases}$ berechnet. Hierbei werden die Zahlen *n* und *m* in Binärdarstellung, getrennt durch das Zeichen #, auf das Eingabeband geschrieben, d.h. die

Eingabe hat die Form $w = bin(n)\#bin(m)$. Dann werden die Zahlen *n* und *m* stellengerecht addiert. Auch für die arithmetischen Funktionen $DIF: \begin{cases} \mathbf{N}\times\mathbf{N} & \rightarrow & \mathbf{N} \\ (n,m) & \rightarrow & \begin{cases} n-m & \text{für } n \geq m \\ 0 & \text{für } n < m \end{cases} \end{cases}$,

$MULT: \begin{cases} \mathbf{N}\times\mathbf{N} & \rightarrow & \mathbf{N} \\ (n,m) & \rightarrow & n\cdot m \end{cases}$ und $DIV: \begin{cases} \mathbf{N}\times\mathbf{N}_{>0} & \rightarrow & \mathbf{N} \\ (n,m) & \rightarrow & \lfloor n/m \rfloor \end{cases}$ lassen sich entsprechende Turingmaschinen angeben. Dabei wird beispielsweise die Berechnung von *MULT* auf sukzessive Anwendung der Berechnung für *SUM* zurückgeführt. In jedem Fall erfolgt die Eingabe in der Form $w = bin(n)\#bin(m)$, und die Zeitkomplexitäten geben an, wieviele Bitoperationen zur Berechnung der jeweiligen Funktionen erforderlich sind. Die Ergebnisse sind in folgendem Satz zusammengefaßt.

Satz 2.1-3:

Es seien *n* und *m* natürliche Zahlen. Mit $size(n)$ werde die Länge der Binärdarstellung der Zahl *n* (ohne führende binäre Nullen) bezeichnet. Dabei sei $size(0) = 1$. Es gelte $|n| \geq |m|$, $k = size(n) \geq size(m)$, $k \in O(\log(n))$. Dann gibt es Turingmaschinen, die die Funktionen $SUM(n,m)$, $DIF(n,m)$, $MULT(n,m)$ und $DIV(n,m)$ berechnen und folgende Zeitkomplexitäten besitzen:

(i) Die Addition und Differenzenbildung der Zahlen *n* und *m* (Berechnung von $SUM(n,m)$ und $DIF(n,m)$) ist jeweils von der Ordnung $O(k)$. Es werden also $O(\log(n))$ viele Bitoperationen ausgeführt.

(ii) Die Multiplikation der Zahlen *n* und *m* (Berechnung von $MULT(n,m)$) kann mit einer Zeitkomplexität der Ordnung $O(k^2)$, also mit $O\left((\log(n))^2\right)$ vielen Bitoperationen, ausgeführt werden.

(iii) Ist $size(n) \geq 2 \cdot size(m)$, dann kann die Berechnung des ganzzahligen Quotienten $\lfloor n/m \rfloor$ (Berechnung von $DIV(n,m)$) mit einer Zeitkomplexität der Ordnung $O(k^2)$, also mit $O\left((\log(n))^2\right)$ vielen Bitoperationen, ausgeführt werden.

Bemerkung: Der Wert $O\left((\log(n))^2\right)$ für die Anzahl der erforderlichen Bitoperationen zur Multiplikation zweier Zahlen läßt sich verbessern zu $O(\log(n)\cdot\log(\log(n))\cdot\log(\log(\log(n))))$.

Meist ist man nicht am exakten Wert der Anzahl der Konfigurationsänderungen interessiert, sondern nur an der **Größenordnung der Zeitkomplexität (**in Abhängigkeit von der Größe der Eingabe). Bei der Analyse wird man meist eine obere Schranke $g(n)$ für $T_{TM}(n)$ herlei-

ten, d.h. man wird eine Aussage der Form $T_{TM}(n) \le g(n)$ begründen. In diesem Fall ist $T_{TM}(n) \in O(g(n))$. Eine zusätzliche Aussage $T_{TM}(n) \le h(n) \le g(n)$ mit einer Funktion $h(n)$ führt auf die verbesserte Abschätzung $T_{TM}(n) \in O(h(n))$. Man ist also bei der worst case-Analayse an einer möglichst kleinen oberen Schranke für $T_{TM}(n)$ interessiert.

Es gibt eine Reihe von **Varianten von Turingmaschinen**:

- Die Turingmaschine besitzt Bänder, die nach links und rechts unendlich lang sind.
- Die Turingmaschine besitzt ein einziges nach links und rechts oder nur zu einer Seite unendlich langes Band.
- Die Turingmaschine besitzt mehrdimensionale Bänder bzw. ein mehrdimensionales Band.
- Das Eingabealphabet der Turingmaschine besteht aus zwei Zeichen, etwa $I = \{0, 1\}$.
- Das Arbeitsalphabet der Turingmaschine besteht aus zwei Zeichen, etwa $\Sigma = \{0, 1\}$.
- Die Turingmaschine hat endlich viele Endzustände.

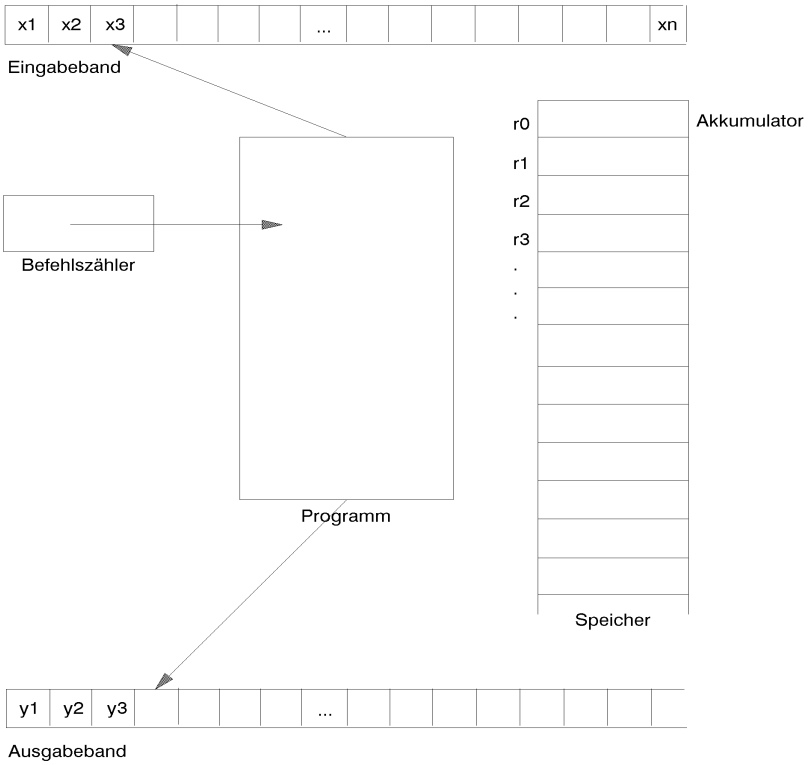
Es zeigt sich, daß alle Definitionen **algorithmisch äquivalent** sind, d.h. eine Turingmaschinenvariante kann eine andere Turingmaschinenvariante simulieren. Allerdings ändert sich dabei u.U. die Zeitkomplexität, in der von der jeweiligen Turingmaschinenvariante Sprachen erkannt werden. Beispielsweise kann eine *k*-DTM *TM* mit Zeitkomplexität $T_{TM}(n)$ durch eine 1-DTM mit Zeitkomplexität $O\left(T_{TM}^2(n)\right)$ simuliert werden.

2.2 Random Access Maschinen

Eine **Random Access Maschine (RAM)** modelliert einen Algorithmus in Form eines sehr einfachen Computers, der ein Programm ausführt, das sich nicht selbst modifizieren kann und fest in einem Programmspeicher geladen ist. Das Konzept der RAM ist zunächst als Modell für die Berechnung partieller Funktionen $f : \mathbf{Z}^n \to \mathbf{Z}^m$ gedacht, die *n*-stellige Zahlenfolgen (ganzer Zahlen) auf *m*-stellige Zahlenfolgen abbilden. Eine Turingmaschine dagegen ist zur Akzeptanz von Sprachen $L \subseteq \Sigma^*$ bzw. zur Berechnung von partieller Funktionen $f : \Sigma^* \to \Sigma'^*$ konzipiert, die Zeichenketten (Wörtern) über einem endlichen Alphabet auf Zeichenketten über eventuell einem anderen Alphabet abbilden. Es wird sich jedoch zeigen, daß beide Modelle äquivalent sind. Beide Modelle sind in der Lage, numerische Funktionen zu berechnen und als Sprachakzeptoren eingesetzt zu werden. Während die „Programmierung“ einer Turingmaschine in der Definition der Überführungsfunktion besteht, eine Vorgehensweise, die verglichen mit der gängigen Programmierung eines Computers zumindest gewöhnungsbedürftig ist, entspricht die Programmierung einer RAM eher der Programmierung eines Computers auf Maschinensprachebene.

Eine RAM hat folgende Bestandteile:

- **Eingabeband**: eine Folge von Zellen, die als Eingabe *n* ganze (positive oder negative) Zahlen x_1, ..., x_n enthalten und von einem Lesekopf nur gelesen werden können. Nachdem eine Zahl x_i gelesen wurde, rückt der Lesekopf auf die benachbarte Zelle, die die ganze Zahl x_{i+1} enthält.
- **Ausgabeband**: eine Folge von Zellen, über die ein Schreibkopf von links nach rechts wandert, der nacheinander ganze Zahlen y_j schreibt. Ist eine Zahl geschrieben, kann sie nicht mehr verändert werden; der Schreibkopf rückt auf das rechts benachbarte Feld.
- **Speicher**: eine unendliche Folge r_0, r_1, r_2, ..., von **Registern**, die jeweils in der Lage sind, eine beliebig große ganze Zahl aufzunehmen. Das Register r_0 wird als **Akkumulator** bezeichnet. In ihm finden alle arithmetische Operationen statt.
- **Programm**: eine Folge aufsteigend numerierter Anweisungen, die fest in der RAM „verdrahtet“ sind. Zur besseren Lesbarkeit eines RAM-Programms können einzelne Anweisungen auch mit **symbolischen Marken** versehen werden. Jede RAM stellt ein eigenes Programm dar, das natürlich mit unterschiedlichen Eingaben konfrontiert werden kann. Das Programm kann sich nicht modifizieren. Ein Programm ist aus einfachen Befehlen aufgebaut (siehe unten). Die einzelnen Anweisungen werden nacheinander ausgeführt.
- **Befehlszähler**: enthält die Nummer der nächsten auszuführenden Anweisung.



Um die Bedeutung des **Befehlsvorrats** einer RAM festzulegen, wird die **Speicherabbildungsfunktion** $c : \mathbf{N} \to \mathbf{Z}$ verwendet. $c(i)$ gibt zu jedem Zeitpunkt des Programmlaufs den Inhalt des Registers r_i an. Zu Beginn des Programmlaufs wird jedes Register mit dem Wert 0 initialisiert, d.h. es ist $c(i) = 0$ für jedes $i \in \mathbf{N}$.

Jeder **Anweisung (Befehl)** ist aus einem **Operationscode** und einem **Operanden** aufgebaut. Ein Operand kann eine der folgenden Formen aufweisen:

1. i
2. $c(i)$, für $i \geq 0$; bei $i < 0$ stoppt die RAM
3. $c(c(i))$, für $i \geq 0$ und $c(i) \geq 0$; bei $i < 0$ oder $c(i) < 0$ stoppt die RAM.

Die RAM-Anweisungen und ihre Bedeutung sind folgender Tabelle zu entnehmen. Dabei steht *i* für eine natürliche Zahl und *m* für eine Anweisungsnummer bzw. für eine Anweisungsmarke im Programm der RAM.

RAM-Anweisung	Bedeutung
LOAD i	$c(0) := i$
LOAD $c(i)$ für $i \geq 0$	$c(0) := c(i)$
LOAD $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := c(c(i))$
STORE $c(i)$ für $i \geq 0$	$c(i) := c(0)$
STORE $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(c(i)) := c(0)$
ADD i	$c(0) := c(0) + i$
ADD $c(i)$ für $i \geq 0$	$c(0) := c(0) + c(i)$
ADD $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := c(0) + c(c(i))$
SUB i	$c(0) := c(0) - i$
SUB $c(i)$ für $i \geq 0$	$c(0) := c(0) - c(i)$
SUB $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := c(0) - c(c(i))$
MULT i	$c(0) := c(0) * i$
MULT $c(i)$ für $i \geq 0$	$c(0) := c(0) * c(i)$
MULT $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := c(0) * c(c(i))$
DIV i für $i \neq 0$	$c(0) := \lfloor c(0) / i \rfloor$
DIV $c(i)$ für $i \geq 0$	$c(0) := \lfloor c(0) / c(i) \rfloor$, falls $c(i) \neq 0$ ist, sonst stoppt die RAM
DIV $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := \lfloor c(0) / c(c(i)) \rfloor$, falls $c(c(i)) \neq 0$ ist, sonst stoppt die RAM
READ $c(i)$ für $i \geq 0$	$c(i) :=$ momentaner Eingabewert, und der Lesekopf rückt eine Zelle nach rechts
READ $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(c(i)) :=$ momentaner Eingabewert, und der Lesekopf rückt eine Zelle nach rechts
WRITE i	i wird auf das Ausgabeband gedruckt, und der Schreibkopf rückt um eine Zelle nach rechts
WRITE $c(i)$ für $i \geq 0$	$c(i)$ wird auf das Ausgabeband gedruckt, und der Schreibkopf rückt um eine Zelle nach rechts
WRITE $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(c(i))$ wird auf das Ausgabeband gedruckt, und der Schreibkopf rückt um eine Zelle nach rechts

../..

JUMP *m*	Der Befehlszähler wird mit der Nummer der Anweisung geladen, die die Marke *m* trägt (nächste auszuführende Anweisung)
JGTZ *m*	Der Befehlszähler wird mit der Nummer der Anweisung geladen, die die Marke *m* trägt (nächste auszuführende Anweisung), falls $c(0) > 0$ ist, sonst wird der Inhalt des Befehlszählers um 1 erhöht
JZERO *m*	Der Befehlszähler wird mit der Nummer der Anweisung geladen, die die Marke *m* trägt (nächste auszuführende Anweisung), falls $c(0) = 0$ ist, sonst wird der Inhalt des Befehlszählers um 1 erhöht
HALT	Die RAM stoppt

Falls eine Anweisung nicht definiert ist, z.B. STORE 3 oder STORE $c(c(3))$ mit $c(3) = -10$, dann stoppt die RAM, desgleichen bei einer Division durch 0.

Die RAM *RAM* definiert bei Beschriftung der ersten n Zellen des Eingabebandes mit x_1, ..., x_n eine partielle Funktion $f_{RAM} : \mathbf{Z}^n \to \mathbf{Z}^m$ (die Funktion ist partiell, da *RAM* nicht bei jeder Eingabe stoppen muß): falls *RAM* bei Eingabe von x_1, ..., x_n stoppt, nachdem y_1, ..., y_m in die *m* ersten Zellen des Ausgabebands geschrieben wurde, dann ist
$f_{RAM}(x_1, ..., x_n) = (y_1, ..., y_m)$.

Eine RAM zur Berechnung der Funktion $f : \begin{cases} \mathbf{N} & \to & \mathbf{N} \\ n & \to & \begin{cases} 1 & \text{für} \quad n = 0 \\ n^n & \text{für} \quad n \geq 1 \end{cases} \end{cases}$

Ein entsprechendes RAM-Programm lautet:

RAM-Anweisungsfolge		Bemerkung zur Bedeutung
Zeilennummer	RAM-Befehl	
1	READ *c*(1)	$r_1 := n$, Eingabe
2 3 4 5	LOAD *c*(1) JGTZ pos WRITE 0 JUMP endif	**IF** $r_1 \leq 0$ **THEN** *Write*(0)
pos 7	LOAD *c*(1) STORE *c*(2)	$r_2 := r_1$
8 9 10	LOAD *c*(1) SUB 1 STORE *c*(3)	$r_3 := r_1 - 1$
while 12 13	LOAD *c*(3) JGTZ continue JUMP endwhile	**WHILE** $r_3 > 0$ **DO**
continue 15 16	LOAD *c*(2) MULT *c*(1) STORE *c*(2)	$r_2 := r_2 * r_1$
17 18 19	LOAD *c*(3) SUB 1 STORE *c*(3)	$r_3 := r_3 - 1$
20	JUMP while	
endwhile	WRITE *c*(2)	*Write* (r_2)
endif	HALT	

Eine RAM kann auch als **Akzeptor einer Sprache** interpretiert werden. Die Elemente des endlichen Alphabets Σ werden dazu mit den Zahlen 1, 2, ..., $|\Sigma|$ identifiziert. Das RAM-Programm *RAM* akzeptiert $L \subseteq \Sigma^*$ auf folgende Weise. In die ersten *n* Zellen des Eingabebandes werden Zahlen geschrieben, die den Buchstaben x_1, ..., x_n eines Wortes $w \in \Sigma^*$, $w = x_1 \ldots x_n$, entsprechenden. In die $(n+1)$-te Zelle kommt als Endmarkierung der Wert 0. *RAM* akzeptiert *w,* falls alle den Buchstaben von *w* entsprechenden Zahlen und die Endmarkierung 0 gelesen wurden, von *RAM* eine 1 auf das Ausgabeband geschrieben wurde und *RAM* stoppt, dann ist $w \in L$. Falls $w \notin L$ ist, dann stoppt *RAM* mit einer Ausgabe ungleich 1, oder *RAM* stoppt nicht. Die auf diese Weise akzeptierten Wörter aus Σ^* bilden die Menge $L(RAM)$.

Ein RAM-Programm *P* zur Akzeptanz von

$$L(P) = \{ w \mid w \in \{1,2\}^* \text{ und die Anzahl der 1'en ist gleich der Anzahl der 2'en } \}$$

P liest jedes Eingabesymbol nach Register r_1 (hier wird angenommen, daß nur die Zahlen 0, 1 und 2 auf dem Eingabeband stehen) und berechnet in Register r_2 die Differenz *d* der Anzahlen der bisher gelesenen 1‘en und 2‘en. Wenn beim Lesen der Endmarkierung 0 diese Differenz gleich 0 ist, wird eine 1 auf das Ausgabeband geschrieben, und *P* stoppt.

RAM-Anweisungsfolge		Bemerkung zur Bedeutung
Zeilennummer	RAM-Befehl	
	LOAD 0 STORE *c*(2)	$d := 0$
	READ *c*(1)	*Read* (*x*)
while	LOAD *c*(1) JZERO endwhile	**WHILE** $x \neq 0$ **DO**
	LOAD *c*(1) SUB 1 JZERO one	**IF** $x \neq 1$
	LOAD *c*(2) SUB 1 STORE *c*(2)	**THEN** $d := d - 1$
	JUMP endif	
one	LOAD *c*(2) ADD 1 STORE *c*(2)	**ELSE** $d := d + 1$
endif	READ *c*(1) JUMP while	*Read* (*x*)
endwhile	LOAD *c*(2) JZERO output HALT	**IF** $d = 0$ **THEN** *Write* (1)
output	WRITE 1 HALT	

Auch beim RAM-Modell interessieren **Zeit- und Raumkomplexität einer Berechnung**.

Mit $\mathbf{Z}^n$ werde das *n*-fache kartesische Produkt von $\mathbf{Z}$ bezeichnet, d.h. $\mathbf{Z}^n = \underbrace{\mathbf{Z} \times \ldots \times \mathbf{Z}}_{n\text{-mal}}$. Die Menge $\mathbf{Z}^*$ aller endlichen Folgen ganzer Zahlen sei definiert durch $\mathbf{Z}^* = \bigcup_{n \geq 0} \mathbf{Z}^n$.

Ein RAM-Programm *RAM* lese die Eingabe $x_1, ..., x_n$ und durchlaufe bis zum Erreichen der HALT-Anweisung *m* viele Anweisungen (einschließlich der HALT-Anweisung). Dann wird durch $t_{RAM}(x_1, ..., x_n) = m$ eine partielle Funktion $t_{RAM} : \mathbf{Z}^* \to \mathbf{N}$ definiert. Falls *RAM* bei Eingabe von $x_1, ..., x_n$ nicht anhält, dann ist $t_{RAM}(x_1, ..., x_n)$ nicht definiert.

In diesem Modell werden zwei Kostenkriterien unterschieden:

Beim **uniformen Kostenkriterium** benötigt die Ausführung jeder Anweisung eine Zeiteinheit, und jedes Register belegt eine Platzeinheit. Die **Zeitkomplexität** von *RAM* (**im schlechtesten Fall, worst case**) wird **unter dem uniformen Kostenkriterium** definiert durch die partielle Funktion $T_{RAM} : \mathbf{N} \to \mathbf{N}$ mit

$T_{RAM}(n) = \max\{ t_{RAM}(w) \mid w \text{ besteht aus bis zu } n \text{ vielen natürlichen Zahlen} \}$.

Entsprechend wird die **Platzkomplexität** von *RAM* (**im schlechtesten Fall, worst case**) $S_{RAM}(n)$ **unter dem uniformen Kostenkriterium** als die während der Rechnung benötigte maximale Anzahl an Registern definiert, wenn bis zu *n* viele Zahlen eingelesen werden.

Ein Register bzw. eine Zelle des Eingabe- und Ausgabebandes kann im RAM-Modell eine beliebig große Zahl aufnehmen. Eine Berechnung mit *n* „großen Zahlen“ ist unter dem uniformen Kostenkriterium genauso komplex wie eine Berechnung mit *n* „kleinen Zahlen“. Diese Sichtweise entspricht häufig nicht den praktischen Gegebenheiten. Dies gilt in der Praxis besonders dann, wenn Speicherzellen eine beschränkte Größe aufweisen, so daß für die Handhabung großer Zahlen pro Zahl mehrere Speicherzellen erforderlich sind. Es erscheint daher sinnvoll, die Größenordnung der bei einer Berechnung beteiligten Zahlen bzw. Operanden mit zu berücksichtigen. Dieses führt auf das **logarithmische Kostenkriterium**, das die Größen der bei einer Berechnung beteiligten Zahlen (gemessen in der Anzahl der Stellen zur Darstellung einer Zahl in einem geeigneten Stellenwertsystem) berücksichtigt.

Mit $l(i)$ werde die **Länge** einer ganzen Zahl *i* bezeichnet:

$$l(i) = \begin{cases} \lfloor \log|i| \rfloor + 1 & \text{für } i \neq 0 \\ 1 & \text{für } i = 0 \end{cases}$$

Die **Kosten eines Operanden** einer RAM-Anweisung faßt folgende Tabelle zusammen.

Operand	Kosten
i	$l(i)$
$c(i)$	$l(i)+l(c(i))$
$c(c(i))$	$l(i)+l(c(i))+l(c(c(i)))$

Beispielsweise erfordert die Ausführung einer Anweisung ADD $c(c(i))$ folgende Einzelschritte:

1. „Decodieren“ des Operanden i: Kosten $l(i)$
2. „Lesen“ von $c(i)$: Kosten $l(c(i))$
3. „Lesen“ von $c(c(i))$: Kosten $l(c(c(i)))$
4. Ausführung von ADD $c(c(i))$: Kosten $l(c(0))+l(i)+l(c(i))+l(c(c(i)))$.

Die folgende Tabelle zeigt die Kosten unter dem logarithmischen Kostenkriterium der Ausführung für jede RAM-Anweisung.

RAM-Anweisung	Bedeutung	Kosten der Ausführung
LOAD i	$c(0):=i$	$l(i)$
LOAD $c(i)$ für $i\geq 0$	$c(0):=c(i)$	$l(i)+l(c(i))$
LOAD $c(c(i))$ für $i\geq 0$ und $c(i)\geq 0$	$c(0):=c(c(i))$	$l(i)+l(c(i))+l(c(c(i)))$
STORE $c(i)$ für $i\geq 0$	$c(i):=c(0)$	$l(c(0))+l(i)$
STORE $c(c(i))$ für $i\geq 0$ und $c(i)\geq 0$	$c(c(i)):=c(0)$	$l(c(0))+l(i)+l(c(i))$
ADD i	$c(0):=c(0)+i$	$l(c(0))+l(i)$
ADD $c(i)$ für $i\geq 0$	$c(0):=c(0)+c(i)$	$l(c(0))+l(i)+l(c(i))$
ADD $c(c(i))$ für $i\geq 0$ und $c(i)\geq 0$	$c(0):=c(0)+c(c(i))$	$l(c(0))+l(i)+l(c(i))+l(c(c(i)))$
SUB i	$c(0):=c(0)-i$	$l(c(0))+l(i)$
SUB $c(i)$ für $i\geq 0$	$c(0):=c(0)-c(i)$	$l(c(0))+l(i)+l(c(i))$
SUB $c(c(i))$ für $i\geq 0$ und $c(i)\geq 0$	$c(0):=c(0)-c(c(i))$	$l(c(0))+l(i)+l(c(i))+l(c(c(i)))$

../..

MULT i	$c(0) := c(0) * i$	$l(c(0)) + l(i)$
MULT $c(i)$ für $i \geq 0$	$c(0) := c(0) * c(i)$	$l(c(0)) + l(i) + l(c(i))$
MULT $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := c(0) * c(c(i))$	$l(c(0)) + l(i) + l(c(i)) + l(c(c(i)))$
DIV i für $i \neq 0$	$c(0) := \lfloor c(0) / i \rfloor$	$l(c(0)) + l(i)$
DIV $c(i)$ für $i \geq 0$	$c(0) := \lfloor c(0) / c(i) \rfloor$	$l(c(0)) + l(i) + l(c(i))$
DIV $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := \lfloor c(0) / c(c(i)) \rfloor$	$l(c(0)) + l(i) + l(c(i)) + l(c(c(i)))$
READ $c(i)$ für $i \geq 0$	$c(i) :=$ Eingabewert	$l(\text{Eingabewert}) + l(i)$
READ $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(c(i)) :=$ Eingabewert	$l(\text{Eingabewert}) + l(c(i))$
WRITE i	Ausgabe von i	$l(i)$
WRITE $c(i)$ für $i \geq 0$	Ausgabe von $c(i)$	$l(i) + l(c(i))$
WRITE $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	Ausgabe von $c(c(i))$	$l(i) + l(c(i)) + l(c(c(i)))$
JUMP m	Befehlszähler := m	1
JGTZ m	Befehlszähler := m, falls $c(0) > 0$ ist	$l(c(0))$
JZERO m	Befehlszähler := m, falls $c(0) = 0$ ist	$l(c(0))$
HALT	Die RAM stoppt	1

Die **Zeitkosten eines RAM-Programms** bei gegebenen Eingabewerten **unter dem logarithmischen Kostenkriterium** ist gleich der Summe der Kosten der abzuarbeitenden Anweisungen. Entsprechend sind die **Platzkosten eines RAM-Programms** bei gegebenen Eingabewerten **unter dem logarithmischen Kostenkriterium** gleich dem Produkt der während der Rechnung benötigte maximalen Anzahl an Registern und der Länge der längsten während der Abarbeitung abgespeicherten Zahl.

Kosten des RAM-Programms zur Berechnung der Funktion

$$f: \begin{cases} \mathbf{N} & \rightarrow & \mathbf{N} \\ n & \rightarrow & \begin{cases} 1 & \text{für} \quad n = 0 \\ n^n & \text{für} \quad n \geq 1 \end{cases} \end{cases}$$

	uniformes Kostenkriterium	logarithmisches Kostenkriterium
Zeitkomplexität	$O(n)$	$O(n^2 \cdot \log(n))$
Platzkomplexität	$O(1)$	$O(\log(n))$

Kosten des RAM-Programm *P* zur Akzeptanz von

$$L(P) = \left\{ w \mid w \in \{1,2\}^* \text{ und die Anzahl der 1'en ist gleich der Anzahl der 2'en} \right\}$$

	uniformes Kostenkriterium	logarithmisches Kostenkriterium
Zeitkomplexität	$O(n)$	$O(n \cdot \log(n))$
Platzkomplexität	$O(1)$	$O(\log(n))$

Eine RAM *RAM* kann eine deterministische Turingmaschine mit *k* Bändern (*k*-DTM) *TM* simulieren. Dazu wird jede Zelle von *TM* durch ein Register von *RAM* nachgebildet. Genauer: der Inhalt der *i*-ten Zelle des *j*-ten Bandes von *TM* kann in Register r_{ik+j+c} gespeichert werden. Hierbei wird angenommen, daß die Symbole des Alphabets Σ von *TM* für *RAM* mit den Zahlen 1, ..., $|\Sigma|$ identifiziert werden, so daß man davon sprechen kann, daß „ein Symbol aus Σ in einem Register oder einer Zelle des Eingabe- oder Ausgabebandes von *RAM* gespeichert werden kann“. Der Wert $c \geq k$ ist eine Konstante, durch die die Register r_1, ..., r_c als zusätzliche Arbeitsregister für *RAM* reserviert werden. Unter diesen Arbeitsregistern werden *k* Register, etwa r_1, ..., r_k, dazu verwendet, die jeweilige Kopfposition von *TM* während der Simulation festzuhalten (r_j enthält die Kopfposition des *j*-ten Bandes). Um den Inhalt einer Zelle von *TM* in der Simulation zu lesen, wird indirekte Adressierung eingesetzt. Beispielsweise kann der Inhalt der simulierten Zelle, über der sich gerade der Schreib/Lesekopf des *j*-ten Bandes befindet, in den Akkumulator durch die RAM-Anweisung LOAD *c*(*c*(*j*)) geladen werden.

Satz 2.2-1:

Hat die *k*-DTM *TM* die Zeitkomplexität $T_{TM}(n) \geq n$, dann gibt es eine RAM *RAM*, die die Eingabe von *TM* in die Register liest, die das erste Band nachbilden, und anschließend die $T_{TM}(n)$ Schritte von *TM* simuliert. Dieser Vorgang verläuft unter uniformem Kostenkriterium in $O(T_{TM}(n))$ vielen Schritten, unter logarithmischem Kostenkriterium in $O(T_{TM}(n) \cdot \log(T_{TM}(n)))$ vielen Schritten.

Umgekehrt gilt:

Es sei *L* eine Sprache, die von einem RAM-Programm der Zeitkomplexität $T_{RAM}(n)$ unter dem logarithmischen Kostenkriterium akzeptiert wird. Falls das RAM-Programm keine Multiplikationen oder Divisionen verwendet, dann wird *L* von einer *k*-DTM (für ein geeignetes *k*) mit Zeitkomplexität $O(T_{RAM}{}^2(n))$ akzeptiert.

Falls das RAM-Programm Multiplikationen oder Divisionen einsetzt, werden diese Operationen jeweils durch Unterprogramme simuliert, in denen als arithmetische Operationen nur Additionen und Subtraktionen verwendet werden. Es läßt sich zeigen (vgl. Satz 2.1-3), daß diese Unterprogramme so entworfen werden können, daß die logarithmischen Kosten zur Ausführung des jeweiligen Unterprogramms höchstens das Quadrat der logarithmischen Kosten der Anweisung betragen, die es nachbildet.

Die folgende Übersicht zeigt (in Ergänzung mit den Aussagen aus Kapitel 2.1) die Anzahl der Schritte (Zeitkomplexität), die Maschinenvariante *A* benötigt, um die Ausführung einer Rechnung zu simulieren, der auf Maschinenvariante *B* bei einem akzeptierten Eingabewort *w* der Länge *n* eine Schrittzahl (Anzahl der Überführungen bis zum Akzeptieren) von $T(n)$ benötigt. Dabei wird für das RAM-Modell das logarithmische Kostenkriterium angenommen.

	simulierende Maschine *A*		
Simulierte Maschine *B*	1-DTM	*k*-DTM	RAM
1-DTM	-	$O(T(n))$	$O(T(n)\log(T(n)))$
k-DTM	$O(T^2(n))$	-	$O(T(n)\log(T(n)))$
RAM	$O(T^3(n))$	$O(T^2(n))$	-

2.3 Programmiersprachen

In den meisten Anwendungen werden Algorithmen mit Hilfe der gängigen Programmiersprachen formuliert (siehe Kapitel 1.3). Es zeigt sich (siehe angegebene Literatur), daß man sehr einfache Programmiersprachen definieren kann, so daß man mit in diesen Sprachen formulierten Algorithmen Turingmaschinen simulieren kann. Umgekehrt kann man mit Turingmaschinen das Verhalten dieser Algorithmen nachbilden. Da eine Turingmaschine einen unendlich großen Speicher besitzt, muß man für Algorithmen in diesen Programmiersprachen unendlich viele Variablen zulassen bzw. voraussetzen, daß die Algorithmen auf Rechnern ablaufen, die einen unendlich großen Speicher besitzen. Dieses Prinzip wurde ja auch im RAM-Modell verwirklicht.

Eine Programmiersprache, deren Programme die Turingberechenbarkeit nachzubilden in der Lage sind, benötigt die Definition von:

- abzählbar vielen Variablen, die Werte aus **N** annehmen können (ganzzahlige und rationale Werte werden durch natürlichzahlige Werte nachgebildet, vgl. die Festpunktdarstellung von Zahlen, reelle Werte durch natürlichzahlige Werte approximiert, vgl. die Gleitpunktdarstellung von Zahlen)
- elementaren Anweisungen wie Wertzuweisungen an Variablen, einfachen arithmetischen Operationen (Addition, Subtraktion, Multiplikation, ganzzahlige Division) zwischen Variablen und Konstanten
- zusammengesetzten Anweisungen (Sequenz *anweisung*$_1$; *anweisung*$_2$;), Blockbildung (**BEGIN** *anweisung*$_1$; ...; *anweisung*$_n$ **END)**, bedingte Anweisungen (**IF** *bedingung* **THEN** *anweisung*$_1$ **ELSE** *anweisung*$_2$; hierbei ist *bedingung* ein Boolescher Ausdruck, der ein logisches Prädikat mit Variablen darstellt), Wiederholungsanweisungen (**WHILE** *bedingung* **DO** *anweisung*;)
- einfachen Ein/Ausgabeanweisungen (**READ** (*x*), **WRITE** (*x*)).

Auf eine formale Beschreibung dieses Ansatzes soll hier verzichtet werden.

Auch bei der Formulierung eines Algorithmus mit Hilfe einer Programmiersprache kann man nach der Zeit- und Raumkomplexität fragen. Dabei muß man wieder zwischen uniformen und logarithmischen Kostenkriterium unterscheiden.

Ein Programm *PROG* habe die Eingabe $x = [x_1, ..., x_n]$, die sich aus *n* Parametern (Formalparameter bei der Spezifikation des Programms, Aktualparameter bei Aufruf des Programms) zusammensetzt. Beispielsweise kann *x* ein Graph sein, der *n* Knoten besitzt. *PROG* berechne eine Ausgabe $y = [y_1, ..., y_m]$, die sich aus *m* Teilen zusammensetzt. Beispielsweise kann *y*

das Ergebnis der Berechnung einer Funktion f sein, d.h. $[y_1, ..., y_m] = f(x_1, ..., x_n)$. Oder *PROG* soll entscheiden, ob die Eingabe x eine spezifizierte Eigenschaft besitzt; dann ist $y \in \{\mathsf{TRUE}, \mathsf{FALSE}\}$. In der Regel wird die Zeit- und Raumkomplexität von *PROG* in Abhängigkeit der Eingabe $x = [x_1, ..., x_n]$ gemessen.

Bei der Berechnung der **uniformen Zeitkomplexität** von *PROG* bei Eingabe von x wird die Anzahl der durchlaufenen Anweisungen gezählt. Bei der Berechnung der **uniformen Raumkomplexität** von *PROG* bei Eingabe von x wird die Anzahl der benötigten Variablen gezählt.

Der Ansatz bietet sich immer dann an, wenn die so ermittelten Werte der Zeit- und Raumkomplexität nur von der Anzahl n der Komponenten der Eingabe $x = [x_1, ..., x_n]$ abhängen. Diese Situation liegt beispielsweise vor, wenn *PROG* die Aufgabe hat, die Komponenten der Eingabe (nach einem „einfachen“ Kriterium) zu sortieren.

Das folgende Beispiel zeigt jedoch, daß der Ansatz nicht immer adäquat ist. Die Pascal-Prozedur `zweier_potenz` berechnet bei Eingabe einer Zahl $n > 0$ den Wert $c = 2^{2^n} - 1$.

```
PROCEDURE zweier_potenz (    n : INTEGER;
                         VAR c : INTEGER);

VAR idx : INTEGER;
    p   : INTEGER;

BEGIN {zweier-potenz }
  idx := n;                 { Anweisung 1 }
  p   := 2;                 { Anweisung 2 }
  WHILE idx > 0 DO          { Anweisung 3 }
    BEGIN
      p   := p*p;           { Anweisung 4 }
      idx := idx - 1;       { Anweisung 5 }
    END;
  c := p - 1;               { Anweisung 6 }
END   { zweier-potenz };
```

Werden nur die Anzahl der ausgeführten Anweisungen gezählt, so ergibt sich bei Eingabe der Zahl $n > 0$ eine Zeitkomplexität der Ordnung $O(n)$ und eine Raumkomplexität der Ordnung $O(1)$. Das Ergebnis $c = 2^{2^n} - 1$ belegt jedoch 2^n viele binäre Einsen und benötigt zu seiner Erzeugung mindestens 2^n viele (elementare) Schritte. Vergrößert man die Eingabe n um eine Konstante k, d.h. betrachtet man die Eingabe $n + k$, so bleibt die Laufzeit in der Ordnung $O(n)$, während das Ergebnis um den *Faktor* 2^k größer wird.

Es bietet sich daher eine etwas sorgfältigere Definition der Zeit- und Raumkomplexität an. Diese wird hier aufgezeigt, wenn die Ein- und Ausgabe eines Programms aus numerischen Daten bzw. aus Daten besteht, die als numerische Daten dargestellt werden können (beispielsweise die Datentypen **BOOLEAN** oder **SET OF** ... in Pascal).

Wenn die Zeit- und Raumkomplexität nicht nur von der Anzahl *n* der Komponenten der Eingabe $x = [x_1, ..., x_n]$ abhängt, sondern wie im Beispiel der Prozedur `zweier_potenz` von den Zahlenwerten $x_1, ..., x_n$ selbst, wird folgender Ansatz gewählt. Der gleiche Ansatz ist angebracht, wenn arithmetische Operationen im Spiel sind, deren Anzahl die Zeit- und Raumkomplexität wesentlich beeinflussen.

Für eine ganze Zahl *z* sei die **Größe** $size(z) = |bin(z)|$ die Anzahl signifikanter Bits, die benötigt werden, um *z* im Binärsystem darzustellen. Für negative Werte von *z* wird die Komplementdarstellung gewählt. Dabei sei $size(0) = 1$. Es gilt also $size(z) = \begin{cases} \lfloor \log_2(|z|) \rfloor + 1 & \text{für } z \neq 0 \\ 1 & \text{für } z = 0 \end{cases}$ und $size(z) \in O(\log(|z|))$. Für $x = [x_1, ..., x_n]$ sei $size(x) = n \cdot size(\max\{x_1, ..., x_n\})$.

Der Wert $size(x)$ gibt damit eine obere Schranke für die Anzahl der Bits an, um die Zahlenwerte darzustellen, die in *x* vorkommen.

Die **logarithmische Zeitkomplexität** eines Programms *PROG* bei Eingabe von *x* ist die Anzahl der Bitoperationen in den durchlaufenen Anweisungen, gemessen in Abhängigkeit von der so definierten Größe $size(x)$. Bei der Berechnung der **logarithmischen Raumkomplexität** von *PROG* bei Eingabe von *x* wird die Anzahl der Bits gezählt, um alle von *PROG* benötigten Variablen abzuspeichern; dieser Wert wird in Abhängigkeit von $size(x)$ genommen.

In obiger Pascal-Prozedur `zweier_potenz` gilt für die Eingabe *n* die Beziehung $k = size(n) = c \cdot \log_2(n)$. Die Anweisungen 1, 2 und 6 werden jeweils einmal durchlaufen. Die Wertzuweisung in Anweisung 1 ist von der Ordnung $O(\log(n))$, die arithmetische Operation in Anweisung 6 von der Ordnung $O(\log(2^{2^n})) = O(2^n)$. Die Anweisungen der Schleife werden insgesamt *n*-mal durchlaufen (Anweisung 3 sogar $(n+1)$-mal). Zu Beginn des *i*-ten Schleifendurchlaufs hat `p` den Wert $2^{2^{i-1}}$, und `idx` hat den Wert $n - i + 1$. Die Anzahl der Bitoperationen zur Durchführung der Anweisung 4 im *i*-ten Schleifendurchlauf ist daher nach Satz 2.1-3 von der Ordnung $O(2^{2i})$, die Anzahl der Bitoperationen für Anweisung 5 von der Ordnung $O(\log(n - i + 1))$. Insgesamt benötigt die Schleife eine Anzahl von Bitoperationen, die sich durch

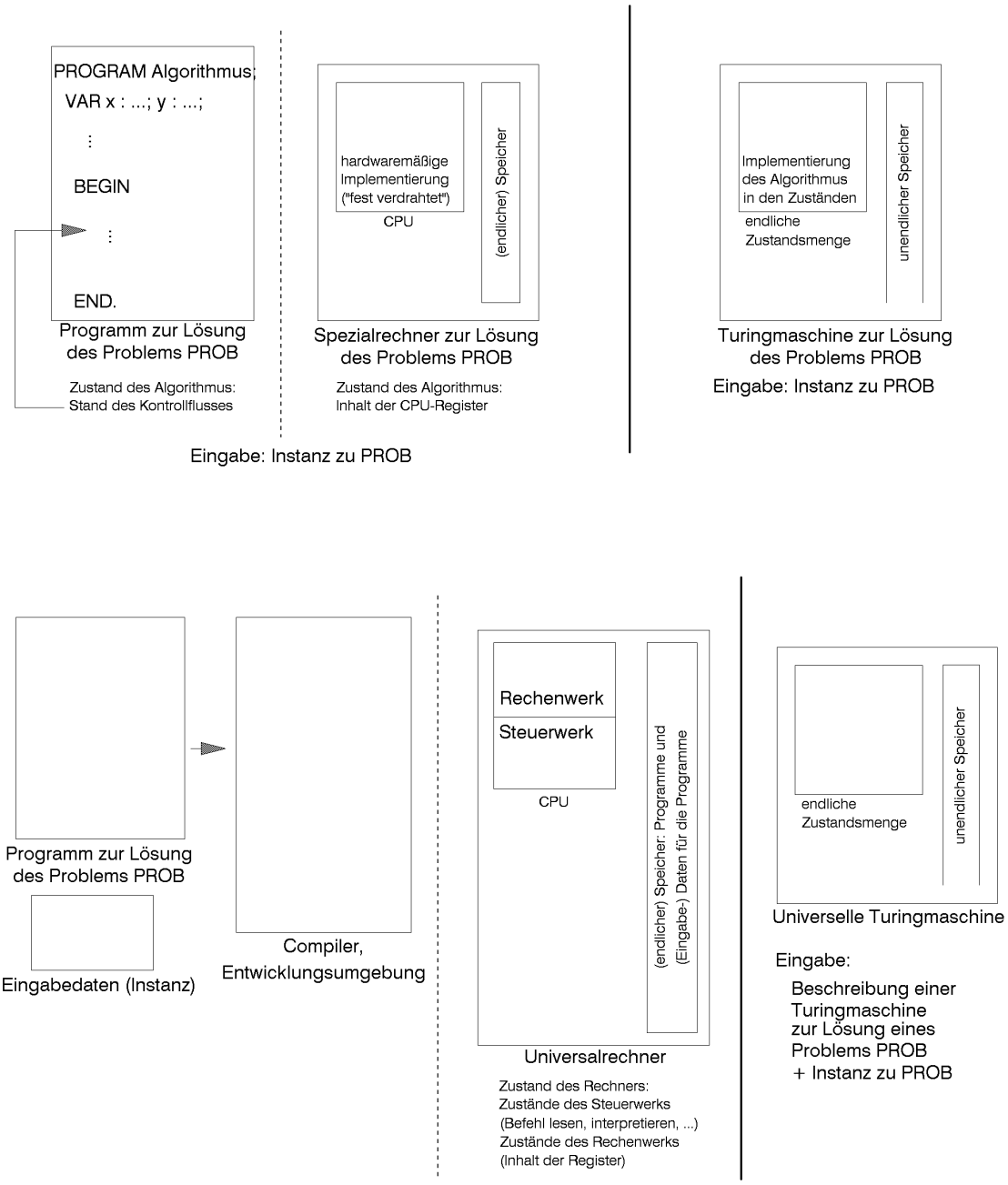
$$c_1 \cdot \sum_{i=1}^{n} \log(n - i + 1) + c_2 \cdot \sum_{i=1}^{n} 2^{2i} \leq c_1 \cdot n \cdot \log(n) + c_2 \cdot 4/3 \cdot (2^{2n} - 1)$$

mit geeigneten Konstanten c_1 und c_2 abschätzen läßt. Dieser Wert ist von der Ordnung $O\left(2^{O(n)}\right)$. Insgesamt wird eine Anzahl von Bitoperationen durchgeführt, die von der Ordnung $O\left(2^{2^k}\right)$ mit $k = size(n)$ ist. Dieser Wert gibt die Realität exakt wieder.

2.4 Universelle Turingmaschinen

Jede Turingmaschine und jeder in einer Programmiersprache formulierte Algorithmus ist zur Lösung eines spezifischen Problems entworfen. Dabei ist natürlich die Eingabe wechselnder Werte der Problemparameter möglich. Entsprechend könnte man in dieser Situation den Algorithmus hardwaremäßig implementieren und hätte dadurch einen „Spezialrechner“, der in der Lage ist, das spezifische Problem, für das er entworfen ist, zu lösen. Ein Computer wird jedoch üblicherweise anders eingesetzt: ein Algorithmus wird in Form eines in einer Programmiersprache formulierten Quelltextes einem Compiler (etwa innerhalb einer Entwicklungsumgebung) vorgelegt. Das Programm wird im Computer in Maschinensprache übersetzt, so daß er in der Lage ist, das Programm „zu interpretieren“. Anschließend wird es von diesem Computer mit entsprechenden eingegebenen Problemparametern ausgeführt. Der Computer ist also in der Lage, nicht nur einen speziellen Algorithmus zur Lösung eines spezifischen Problems, sondern jede Art von Algorithmen auszuführen, solange sie syntaktisch korrekt formuliert sind.

Diese Art der Universalität ist auch im Turingmaschinen-Modell möglich. Im folgenden wird dazu eine **universelle Turingmaschine** *UTM* definiert. Diese erhält als Eingabe die Beschreibung einer Turingmaschine *TM* (ein Problemlösungsalgorithmus) und einen Eingabedatensatz *w* für *TM*. Die universelle Turingmaschine verhält sich dann genauso, wie sich *TM* bei Eingabe von *w* verhalten würde.



Im folgenden wird zunächst gezeigt, wie man die Beschreibung einer Turingmaschine aus ihrer formalen Definition erhält.

Eine deterministische *k*-Band-Turingmaschine $TM = \left(Q, \Sigma, I, d, b, q_0, q_{accept}\right)$ kann als endliches Wort über {0, 1} kodiert werden. Dazu wird angegeben, wie eine mögliche Kodierung von *TM* aussehen kann (die hier vorgestellte Kodierung ist natürlich nur eine von vielen möglichen):

Es sei # ein neues Zeichen, das in $Q \cup \Sigma$ nicht vorkommt. Das Zeichen # dient als syntaktischen Begrenzungssymbol innerhalb der Kodierung.

Die Zustandsmenge von *TM* sei $Q = \{q_0, .., q_{|Q|-1}\}$, ihr Arbeitsalphabet $\Sigma = \{a_0, .., a_{|\Sigma|-1}\}$. Man kann annehmen, daß der Startzustand q_0 und der akzeptierende Zustand $q_{accept} = q_{|Q|-1}$ ist. Das Blankzeichen *b* kann mit a_0 identifiziert werden. Die Anzahl der Zustände, die Anzahl der Elemente in Σ und die Anzahl der Bänder werden in einem Wort $w_0 \in \{0,1,\#\}^*$ festgehalten: $w_0 = \#bin(|Q|)\#bin(|\Sigma|)\#bin(k)\#\#$.

Hierbei bezeichnet wieder $bin(n)$ die Binärdarstellung von *n*.

Die Überführungsfunktion d habe *d* viele Zeilen. Die *t*- te Zeile laute:
$d(q_i, a_{i_1}, ..., a_{i_k}) = (q_j, (a_{j_1}, d_1), ..., (a_{j_k}, d_k))$.
Es sind $a_{i_l} \in \Sigma$, $a_{j_m} \in \Sigma$ und $d_i \in \{L, R, S\}$. Diese Zeile wird zunächst als Zeichenkette
$w_t = (bin(i)\#bin(i_1)\#...\#bin(i_k))(bin(j)\#(bin(j_1)\#d_1)\#...\#(bin(j_k)\#d_k))\#$
kodiert. Diese Zeichenkette ist ein Wort über dem Alphabet $\{0,1,(,),\#,L,R,S\}$. Zu beachten ist, daß w_t sich öffnende und schließende Klammern enthält; der Ausdruck $bin(i)$ enthält keine Klammern, sondern ist eine Zeichenkette über $\{0,1\}$.

Die gesamte Turingmaschine *TM* läßt sich durch Konkatination w_{TM} der Wörter w_0, w_1, ..., w_d kodieren: $w_{TM} = w_0 w_1 ... w_d$. Es ist $w_{TM} \in \{0,1,(,),\#,L,R,S\}^*$. Die 8 Buchstaben dieses Alphabets werden buchstabenweise gemäß der Tabelle

Zeichen	Umsetzung	Zeichen	Umsetzung
0	000	#	100
1	001	*L*	101
(	010	*R*	110
)	011	*S*	111

umgesetzt. Das Resultat der Umsetzung von w_{TM} in eine Zeichenkette über {0, 1} wird mit $code(TM)$ bezeichnet.

Beispiel: Kodierung einer Turingmaschine

Gegeben sei die 1-DTM $TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$ mit $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0,1,b\}$, $I = \{0,1\}$, $q_{accept} = q_2$ und d gemäß der folgenden Tabelle:

Überführungsfunktion $d(q_i, a_{i_1}, ..., a_{i_k})$ $= (q_j, (a_{j_1}, d_1), ..., (a_{j_k}, d_k))$	Zwischenkodierung $w_t = (bin(i)\#bin(i_1)\#...\#bin(i_k))$ $(bin(j)\#(bin(j_1)\#d_1)\#...\#(bin(j_k)\#d_k))$; entsprechend dem Wort über {0, 1}
$d(q_0,1) = (q_1, (0,R))$	$w_1 = (0\#1)(1\#(0\#R))\#$ 01000010000101110100011000100001001100110111 00
$d(q_1,0) = (q_0, (1,R))$	$w_2 = (1\#0)(0\#(1\#R))\#$ 01000110000001101000010001000110011001101110 0
$d(q_1,1) = (q_2, (0,R))$	$w_3 = (1\#1)(10\#(0\#R))\#$ 010001100001011010001000100010000100110011011100
$d(q_1,b) = (q_1, (1,L))$	$w_4 = (0\#11)(1\#(1\#L))\#$ 010001100001001011010001100010001100101011011100

Das Wort w_0 lautet: w_0 =#11#11#1##. Dieses Wort wird übersetzt in eine 0-1-Kodierung und ergibt 10000100110000100110011001 00.

Die Turingmaschine *TM* ist also kodiert durch *code*(*TM*) =
100001001100001001100110010001000010000101101000110001000010011001101110
001000110000001101000010001000110011001101110001000110000101101000100010
001000010011001101110001000110000100101101000110001000110010101101110 0.

Die Turingmaschine TM_{min} mit der kürzesten Kodierung ist die *k*-DTM $TM_{min} = (Q, \Sigma, I, d, b, q_0, q_{accept})$ mit $k = 0$, $Q = \{q_0\}$, $\Sigma = \{b\}$ und $d = \varnothing$. Das dieser Turingmaschine entsprechende Wort w_0 lautet: w_0 =#1#1#0##, übersetzt in eine 0-1-Kodierung 100001100001100000100100. Da $d = \varnothing$ ist, stimmt $code(TM_{min})$ mit dieser 0-1-Folge bereits überein.

Zu jeder Turingmaschine *TM* gibt es also ein Wort $w \in \{0,1\}^*$, $w = code(TM)$, das *TM* kodiert. Offensichtlich ist $\{code(TM) \mid TM \text{ ist eine Turingmaschine}\} \subseteq \{0,1\}^*$. Die Übersetzung der Angabe einer Turingmaschine *TM* in ihre Kodierung *code*(*TM*) gemäß der angegebenen Regeln kann mit Hilfe eines algorithmischen Verfahrens erfolgen.

Für unterschiedliche Turingmaschinen TM_1 und TM_2 gilt dabei $code(TM_1) \neq code(TM_2)$, d.h. die so definierte Abbildung *code* ist injektiv.

Andererseits kann man einem Wort $w \in \{0,1\}^*$ „ansehen“, ob es eine Turingmaschine kodiert. Nicht jedes Wort $w \in \{0,1\}^*$ kodiert eine Turingmaschine, und natürlich kann es vorkommen, daß die durch ein Wort $w \in \{0,1\}^*$ kodierte Turingmaschine wenig Sinnvolles leistet. Die oben informell beschriebenen Regeln der Kodierung einer Turingmaschine *TM* durch ein Wort $code(TM)$ erlauben ein algorithmisches Verfahren der syntaktischen Analyse, das feststellt, ob ein Wort $w \in \{0,1\}^*$ die Kodierung einer Turingmaschine darstellt. Beispielsweise muß dabei untersucht werden, ob *w*

- eine durch drei teilbare Länge besitzt
- mit einem Teilwort der Form $100v_1100v_2100v_3100100$ beginnt, wobei sich v_1, v_2 und v_3 ausschließlich aus der Konkatination von Zeichenfolgen 000 und 001 zusammensetzen (nimmt man von v_1 jedes dritte Zeichen, so erhält man eine Binärzahl, die die Anzahl der Zustände der durch *w* kodierten Turingmaschine angibt; der Zustand mit der höchsten Nummer ist der akzeptierende Zustand, der Zustand mit der Nummer 0 der Anfangszustand; entsprechend erhält man aus v_2 die Anzahl der Zeichen des Arbeitsalphabets und damit implizit das Arbeitsalphabet Σ und aus v_3 die Anzahl *k* der Bänder)
- in seinem restlichen Teilwort syntaktisch korrekt eine Überführungsfunktion kodiert (dazu ist eine Reihe von Bedingungen zu prüfen, etwa die der Klammersetzung entsprechende Kodierung, die korrekte Angabe der Stellenzahl der Überführungsfunktion, die korrekte Verwendung von Zustandsnummern und Buchstabennummern des Arbeitsalphabets in der Überführungsfunktion, die Tatsache, daß die Überführungsfunktion für den akzeptierenden Zustand nicht definiert ist usw.).

Dieses Verfahren, auf dessen detaillierte Darstellung hier verzichtet werden soll, werde mit *VERIFIZIERE_TM* bezeichnet.

Für ein Wort $w \in \{0,1\}^*$ ist

$$VERIFIZIERE\_TM(w) = \begin{cases} \mathsf{TRUE} & \text{falls } w \text{ die Kodierung einer Turingmaschine darstellt} \\ \mathsf{FALSE} & \text{falls } w \text{ nicht die Kodierung einer Turingmaschine darstellt} \end{cases}$$

Falls ein Wort $w \in \{0,1\}^*$ eine Turingmaschine *TM* kodiert, d.h. $w = code(TM)$, dann bezeichnet man mit $TM = \overset{-1}{code}(w)$ **die durch *w* kodierte Turingmaschine** (diese Notation ist zulässig, da *code* injektiv ist). Wir schreiben dafür kürzer
$TM = K_w$.

Mit Hilfe der Kodierungen von Turingmaschinen kann man eine **lineare Ordnung auf der Menge der Turingmaschinen** definieren. Es sei $w \in \{0,1\}^*$. Für $i \in \mathbf{N}_{>0}$ heißt ***w* Kodierung der *i*-ten Turingmaschine**, falls gilt:

(i) *VERIFIZIERE_TM*(*w*) = TRUE

(ii) die Menge $\left\{ x \,\middle|\, \begin{array}{l} x \in \{0,1\}^* \text{ und } x \text{ liegt in der lexikographischen Ordnung vor } w \\ \text{und } \mathit{VERIFIZIERE\_TM}(x) = \mathsf{TRUE} \end{array} \right\}$ enthält genau $i-1$ viele Elemente.

Falls $w = \mathit{code}(TM)$ die Kodierung der *i*-ten Turingmaschine ist, dann heißt $TM = K_w$ die ***i*-te Turingmaschine**.

Der folgende Algorithmus ermittelt bei Eingabe einer Zahl $i \in \mathbf{N}_{>0}$ die Kodierung der *i*-ten Turingmaschine.

Eingabe: Eine natürliche Zahl $i \in \mathbf{N}_{>0}$

Verfahren: Aufruf der Funktion `Generiere_TM (`*i*`)`

Ausgabe: $w \in \{0,1\}^*$, *w* ist die Kodierung der *i*-ten Turingmaschine.

Die Funktion `Generiere_TM` wird in Pseudocode beschrieben.

```
FUNCTION Generiere_TM (i : INTEGER) : STRING;

VAR x : STRING;
    y : STRING;
    k : INTEGER;

BEGIN { Generiere_TM }
  x := '0';  { x∈{0,1}* }
  k := 0;

  WHILE k < i DO
     BEGIN
       IF VERIFIZIERE_TM(x)
       THEN BEGIN
              k := k + 1;
              y := x;
            END;
```

```
        x := Nachfolger von x in der lexikographischen Reihenfolge von {0,1}*;
      END;

 Generiere_TM := y;
END   { Generiere_TM }
```

Der folgende Algorithmus ermittelt bei Eingabe von $w \in \{0,1\}^*$ die Nummer i in der oben definierten Ordnung auf der Menge der Turingmaschinen, falls w überhaupt eine Turingmaschine kodiert; ansonsten gibt er den Wert 0 aus.

Eingabe: $w \in \{0,1\}^*$

Verfahren: Aufruf der Funktion `Ordnung_TM (`w`)`

Ausgabe: $i > 0$, falls w die Kodierung der i-ten Turingmaschine ist,
$i = 0$, sonst.

Die Funktion `Ordnung_TM` wird in Pseudocode beschrieben.

```
FUNCTION Ordnung_TM (w : STRING) : INTEGER;

VAR x : STRING;
    k : INTEGER;

BEGIN { Ordnung_TM }
  IF NOT VERIFIZIERE_TM(w)
  THEN BEGIN
         Ordnung_TM := 0;
         Exit;
       END;

  x := '0';  { x∈{0,1}* }
  k := 1;

  WHILE NOT (x = w) DO
     BEGIN
       IF VERIFIZIERE_TM(x)
       THEN k := k + 1;
       x := Nachfolger von x in der lexikographischen Reihenfolge von {0,1}*;
     END;
 Ordnung_TM := k;
END   { Ordnung_TM };
```

Ein Wort $w \in \{0,1\}^*$ kann damit sowohl als die Kodierung der *i*-ten Turingmaschine als auch als ein Eingabewort für eine gegebene Turingmaschine oder auch als Binärzahl interpretiert werden. Die Nummer *i* kann aus *w* algorithmisch deterministisch ermittelt werden (falls *w* überhaupt eine Turingmaschine kodiert); ebenso kann die Kodierung der *i*-ten Turingmaschine erzeugt werden.

Satz 2.4-1:

Es gibt eine **universelle Turingmaschine** *UTM* mit Eingaben $z \in \{0,1,\#\}^*$, so daß $z \in L(UTM)$ genau dann gilt, wenn *z* die Form $z = u\#w$ mit $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$ besitzt, *VERIFIZIERE_TM*(*w*) = TRUE ist (d.h. $w = code(TM)$ für eine Turingmaschine *TM*, $TM = K_w$) und $u \in L(K_w)$ gilt.

Falls *z* die Form $z = u\#w$ mit $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$ besitzt und *VERIFIZIERE_TM*(*w*) = TRUE ist, simuliert die universelle Turingmaschine *UTM* das Verhalten von K_w auf *u*.

Beweis:

Zunächst prüft *UTM*, ob die Eingabe $z \in \{0,1,\#\}^*$ genau ein Zeichen # enthält. Ist dieses nicht der Fall, so wird *z* nicht akzeptiert. Hat *z* die Form $z = u\#w$ mit $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$, so überprüft *UTM*, ob *w* eine Turingmaschine kodiert (siehe Funktion *VERIFIZIERE_TM*(*w*)). Falls dieses nicht zutrifft, wird *z* nicht akzeptiert. *UTM* kann so konstruiert werden, daß *UTM* in diesen beiden nichtakzeptierenden Fällen nicht stoppt[3].

Andernfalls hat *w* die Form $w = 100v_1100v_2100v_3100100v$, wobei sich v_1, v_2 und v_3 ausschließlich aus der Konkatination von Zeichenfolgen 000 und 001 zusammensetzen und $v \in \{0,1\}^*$ ist. Aus v_1 läßt sich die Anzahl der Zustände, aus v_2 die Anzahl der Zeichen des Arbeitsalphabets und damit implizit das Arbeitsalphabet Σ und aus v_3 die Anzahl *k* der Bänder von K_w ermitteln (siehe Beschreibung von *VERIFIZIERE-TM*). Der Zustand mit der

[3] Ist *UTM* eine *k*-DTM mit der Überführungsfunktion δ_{UTM} und dem Arbeitsalphabet Σ_{UTM} und befindet sich *UTM* nach erfolgloser Korrektheitsprüfung von *z* im Zustand *q*, so enthält δ_{UTM} die Zeile $\delta_{UTM}(q, a_1, ..., a_k) = (q, (a_1, S), ..., (a_k, S))$ für $a_1 \in \Sigma_{UTM}$, ..., $a_k \in \Sigma_{UTM}$.
Dadurch stoppt *UTM* nicht, wobei die Bandinhalte nicht mehr verändert werden.

höchsten Nummer ist der akzeptierende Zustand, der Zustand mit der Nummer 0 der Anfangszustand.

UTM erzeugt jetzt auf einem weiteren Band, das als Konfigurations-Simulationsband bezeichnet werden soll, die Kodierung der Anfangskonfiguration von K_w mit Eingabewort *u*, d.h. die Kodierung von $K_0 = \left(q_0, \underbrace{(u,1),(e,1),...,(e,1)}_{k} \right)$. Hierbei kann ein ähnlicher Umsetzungsmechanismus wie zur Erzeugung der Kodierung einer Turingmaschine verwendet werden. Eine Konfiguration von K_w der Form $K = (q_t, (a_1, i_1), ..., (a_k, i_k))$, wobei q_t der *t*-te Zustand von K_w und $a_j \in \Sigma^*$, $i_j \geq 1$ für *j* = 1, ..., *k* ist, wird zunächst (gedanklich) umgesetzt in $(bin(t)\#(b_1\#bin(i_1))\#...\#(b_k\#bin(i_k)))$. Hierbei erhält man b_j aus a_j (für *j* = 1, ..., *k*), indem man jeden Buchstaben durch den Binärwert seiner Nummer in Σ, gefolgt vom Zeichen # ersetzt. Ist beispielsweise $a_j = aacca$ und sind *a* bzw. *c* die Zeichen in Σ mit den Nummern 1 bzw. 3, so ist $b_j = 1\#1\#11\#11\#1\#$. Die Zeichenfolge $(bin(t)\#(b_1\#bin(i_1))\#...\#(b_k\#bin(i_k)))$ wird mittels anfangs angegebener Tabelle in eine Folge über $\{0,1\}$ umgesetzt.

Ist K_w etwa eine 3-DTM mit $\Sigma = \{b, 0, 1, a\}$, wobei *b* das Blankzeichen (mit Nummer 0) bezeichnet und die Zeichen 0, 1 und *a* die Nummern 1 bis 3 tragen, so lautet die Anfangskonfiguration K_0 von K_w mit dem Eingabewort $u = 11001$:
$K_0 = (q_0, (11001,1), (e,1), (e,1), (e,1))$ bzw.
$(0\#(10\#10\#1\#1\#10\#\#1)\#(0\#\#1)\#(0\#\#1))$ und in eine 0-1-Folge kodiert:
0100001000100010001000010001000011000011000010001001000010111000100001001000010111000100001001000010111011.

Im Endstück *v* des Wortes $w = 100v_1100v_2100v_3100100v$ ist die Überführungsfunktion von K_w kodiert. *UTM* kann mit Hilfe der dort enthaltenen Informationen das Verhalten von K_w simulieren, indem *UTM* die Einträge auf dem Konfigurations-Simulationsband entsprechend der aus *v* gelesenen Überführungsfunktion schrittweise ändert. Auf dem Anfangsstück 010*m*100 des Konfigurations-Simulationsbands (hierbei besteht *m* ausschließlich aus Teilzeichenketten der Form 000 und 001) steht dabei jeweils die Kodierung des aktuellen Zustands von K_w. Endet die Simulation mit einem Wert *m*, der dem akzeptierenden Zustand von K_w entspricht, akzeptiert *UTM* die Eingabe *z*. ///

Die universelle Turingmaschine *UTM* kann so konstruiert werden, daß sie mit einem Band auskommt, da sich jede Turingmaschinen mit mehreren Bändern durch eine einbändige Turingmaschine simulieren läßt.

Das Konzept einer universellen Turingmaschine wird u.a. dazu verwendet, die Grenzen der Berechenbarkeit aufzuzeigen (Kapitel 3.2).

2.5 Nichtdeterminismus

Der Begriff „deterministisch“ in der Definition einer *k*-DTM drückt aus, daß die Nachfolgekonfiguration einer Konfiguration *K* in einer Berechnung der *k*-DTM eindeutig durch den in *K* vorkommenden Zustand *q*, die von den Schreib/Leseköpfen gerade gelesenen Zeichen a_1, ..., a_k, den (eindeutigen) Wert $d(q, a_1, ..., a_k)$ der Überführungsfunktion und die gegenwärtigen Positionen der Schreib/Leseköpfe bestimmt ist. In diesem Kapitel wird das Modell der nichtdeterministischen Turingmaschine eingeführt. Eine nichtdeterministische Turingmaschine unterscheidet sich von einer deterministischen Turingmaschine in der Definition der Überführungsfunktion d .

Eine **nichtdeterministische *k*-Band-Turingmaschine (*k*-NDTM)** *TM* ist definiert durch
$TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$
mit:

1. Q ist eine endliche nichtleere Menge: die **Zustandsmenge**
2. Σ ist eine endliche nichtleere Menge: das **Arbeitsalphabet**
3. $I \subseteq \Sigma$ ist eine endliche nichtleere Menge: das **Eingabealphabet**
4. $b \in \Sigma \setminus I$ ist das **Leerzeichen**
5. $q_0 \in Q$ ist der **Anfangszustand (**oder **Startzustand)**
6. $q_{accept} \in Q$ ist der **akzeptierende Zustand (Endzustand)**
7. $d : (Q \setminus \{q_{accept}\}) \times \Sigma^k \rightarrow \mathbf{P}(Q \times (\Sigma \times \{L, R, S\})^k)$ ist eine partielle Funktion, die **Überführungsfunktion**; insbesondere ist $d(q, a_1, \ldots, a_k)$ für $q = q_{accept}$ nicht definiert; zu beachten ist, daß d für einige weitere Argumente eventuell nicht definiert ist.

Die Überführungsfunktion ordnet also jedem Argument $(q, a_1, \ldots, a_k)$ nicht einen einzigen (eindeutigen) Wert $(q', (b_1, d_1), ..., (b_k, d_k))$, sondern eine *endliche Menge von Werten* der Form $\{(q_1, (b_{11}, d_{11}), ..., (b_{1k}, d_{1k})), ..., (q_t, (b_{t1}, d_{t1}), ..., (b_{tk}, d_{tk}))\}$ zu, von denen in einer Berechnung ein Wert genommen werden kann (natürlich kann diese endliche Menge auch aus einem einzigen Element bestehen).

Damit wird auch das Aussehen einer Berechnung einer nichtdeterministischen Turingmaschine *TM* neu definiert. Auch hierbei wird wieder der Begriff der **Konfiguration** verwendet, um

den gegenwärtigen Gesamtzustand von *TM* zu beschreiben, d.h. den gegenwärtigen Zustand zusammen mit den Bandinhalten und den Positionen der Schreib/Leseköpfe:
$K = (q,(a_1,i_1),...,(a_k,i_k))$ mit $q \in Q$, $a_j \in \Sigma^*$, $i_j \geq 1$ für j = 1, ..., k.

TM startet wie im deterministischen Fall im Anfangszustand mit einer **Anfangskonfiguration** $K_0 = (q_0,(w,1),(e,1),...,(e,1))$. Ist *TM* in eine Konfiguration $K = (q,(a_1,i_1),...,(a_k,i_k))$ gekommen und ist $d(q,a_1,...,a_k)$ definiert, dann besteht $d(q,a_1,...,a_k)$ aus einer endlichen Menge von Werten der Form $(q',(b_1,d_1),...,(b_k,d_k))$, d.h.
$d(q,a_1,...,a_k) = \{(q_1,(b_{11},d_{11}),...,(b_{1k},d_{1k})),...,(q_t,(b_{t1},d_{t1}),...,(b_{tk},d_{tk}))\}$.
Als **Folgekonfiguration** von *K* wird eine der *t* möglichen Konfigurationen
$K_1 = (q_1,(b_{11},i'_{11}),...,(b_{1k},i'_{1k}))$, ..., $K_t = (q_t,(b_{t1},i'_{t1}),...,(b_{tk},i'_{tk}))$
genommen. K_i für i = 1, ..., t entsteht aus *K* dadurch, daß man wie im deterministischen Fall q durch q_i und a_1, ..., a_k durch b_{i1}, ..., b_{ik} ersetzt und die Köpfe so bewegt, wie es $(q_i,(b_{i1},d_{i1}),...,(b_{ik},d_{ik}))$ in $d(q,a_1,...,a_k)$ angibt. Welche der *t* möglichen Folgekonfigurationen auf die Konfiguration *K* folgt, wird nicht gesagt (wird nichtdeterministisch festgelegt); es wird „eine geeignete Folgekonfiguration“ genommen. Man schreibt dann
$K \Rightarrow K_i$.

Wie im deterministischen Fall heißt eine Konfiguration K_{accept}, die den akzeptierenden Zustand q_{accept} enthält, d.h. eine Konfiguration der Form
$K_{accept} = (q_{accept},(a_1,i_1),...,(a_k,i_k))$,
akzeptierende Konfiguration (**Endkonfiguration)**.

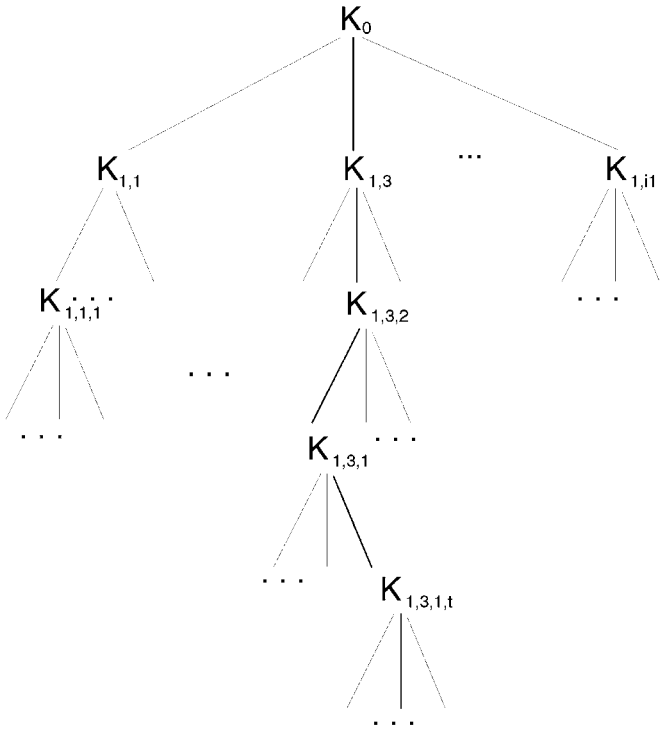
Wie im deterministischen Fall schreibt man $K \Rightarrow^m K'$ mit $m \in \mathbf{N}$, wenn K' aus *K* durch *m* Konfigurationsänderungen hervorgegangen ist, d.h. wenn es *m* Konfigurationen K_1, ..., K_m gibt mit
$K \Rightarrow K_1 \Rightarrow ... \Rightarrow K_m = K'$.
Für $m = 0$ ist dabei $K = K'$.

Die von einer *k*-NDTM *TM* **akzeptierte Sprache** ist die Menge
$L(TM) = \{w \mid w \in I^* \text{ und } w \text{ wird von } TM \text{ akzeptiert}\}$
$= \{w \mid w \in I^* \text{ und } (q_0,(w,1),(e,1),...,(e,1)) \Rightarrow^* (q_{accept},(a_1,i_1),...,(a_k,i_k))\}$.

Wie im deterministischen Fall kann man die Überführungsfunktion d modifizieren, indem man die Zustandsmenge *Q* um einen neuen Zustand q_{reject} erweitert und d um entsprechende Zeilen ergänzt (siehe Kapitel 2.1), so daß alle Berechnungen, die in einem Zustand $q \neq q_{accept}$,

für die $d(q,...)$ bisher nicht definiert ist, um eine Überführung in den Zustand q_{reject} fortgesetzt werden können. Für q_{reject} ist d nicht definiert.

Im *deterministischen* Fall kann man sich den Ablauf einer Berechnung als eine Folge
$K_0 \Rightarrow ... \Rightarrow K \Rightarrow K' \Rightarrow ...$
vorstellen, wobei jeder Schritt $K \Rightarrow K'$ eindeutig durch die Überführungsfunktion d feststeht. Im *nichtdeterministischen* Fall hat man in einem Schritt $K \Rightarrow K'$ eventuell mehrere Alternativen, so daß sich eine Berechnung hier als ein *Pfad* durch einen „Berechnungsbaum" darstellt. Jeder Knoten dieses Berechnungsbaums ist mit einer Konfiguration markiert. Die Wurzel ist mit einer Anfangskonfiguration $K_0 = (q_0,(w,1),(e,1),...,(e,1))$ mit einem Wort $w \in I^*$ markiert. Ist die Konfiguration $K = (q,(a_1,i_1),...,(a_k,i_k))$ die Markierung eines Knotens und kann hier der Eintrag $d(q,a_1,...,a_k)$ der Überführungsfunktion angewendet werden, etwa
$d(q,a_1,...,a_k) = \{(q_1,(b_{11},d_{11}),...,(b_{1k},d_{1k})),...,(q_t,(b_{t1},d_{t1}),...,(b_{tk},d_{tk}))\}$, dann enthält dieser Knoten des Berechnungsbaums *t* direkte Nachfolger, die mit den sich ergebenden Nachfolgekonfigurationen $K_1 = (q_1,(b_{11},i'_{11}),...,(b_{1k},i'_{1k}))$, ..., $K_t = (q_t,(b_{t1},i'_{t1}),...,(b_{tk},i'_{tk}))$ markiert sind. Führt einer der Pfade von einer Anfangskonfiguration K_0 zu einer Endkonfiguration K_{accept}, so wird das in der Anfangskonfiguration stehende Wort *w* akzeptiert.



Das Partitionenproblem mit ganzzahligen Eingabewerten

Instanz: $I = \{a_1, ..., a_n\}$

I ist eine Menge von *n* natürlichen Zahlen.

Lösung: Entscheidung „ja“, falls es eine Teilmenge $J \subseteq \{1, ..., n\}$ mit $\sum_{i \in J} a_i = \sum_{i \notin J} a_i$ gibt (d.h. wenn sich die Eingabeinstanz in zwei Teile zerlegen läßt, deren jeweilige Summen gleich groß sind).
Entscheidung „nein“ sonst.

Lösung: Entscheidung „ja“, falls es eine Teilmenge $J \subseteq \{1, ..., n\}$ mit $\sum_{i \in J} a_i = \sum_{i \notin J} a_i$ gibt (d.h. wenn sich die Eingabeinstanz in zwei Teile zerlegen läßt, deren jeweilige Summen gleich groß sind).
Entscheidung „nein“ sonst.

Offensichtlich lautet bei einer Instanz $I = \{a_1, ..., a_n\}$ mit ungeradem $S = \sum_{i=1}^{n} a_i$ die Entscheidung „nein“. Daher kann *S* als gerade vorausgesetzt werden.

Zunächst wird eine 3-NDTM *TM* angegeben, die dieses Problem löst, wenn die Eingaben in „unärer“ Kodierung eingegeben werden: Eine Eingabeinstanz $I = \{a_1, a_2, ..., a_n\}$ wird dabei als Wort $w = 10^{a_1}10^{a_2} ...10^{a_n}$ kodiert.

Die 3-NDTM $TM = (\{q_0, ..., q_5\}, \{0, 1, b, \$\}, \{0, 1\}, d, b, q_0, q_5)$ arbeitet folgendermaßen:

1. Das Eingabewort wird von links nach rechts gelesen. Jedesmal, wenn eine Folge 10^{a_i} erreicht wird, wird die Folge der Nullen entweder auf das 2. oder das 3. Band kopiert
2. Wenn das Ende des Eingabeworts erreicht ist, wird geprüft, ob auf dem 2. und 3. Band die gleiche Anzahl von Nullen steht; dieses geschieht durch simultanes Vorrücken der Köpfe nach links.

Die Überführungsfunktion d wird durch folgende Tabelle gegeben.

momentaner Zustand	Symbol unter dem Schreib/Lesekopf auf			neuer Zustand	neues Symbol, Kopfbewegung auf		
	Band 1	Band 2	Band 3		Band 1	Band 2	Band 3
q_0	1	b	b	q_1	1, S	\$, R	\$, R
q_1	1	b	b	q_2	1, R	b, S	b, S
				q_3	1, R	b, S	b, S
q_2	0	b	b	q_2	0, R	0, R	b, S
	1	b	b	q_1	1, S	b, S	b, S
	b	b	b	q_4	b, S	b, L	b, L
q_3	0	b	b	q_3	0, R	b, S	0, R
	1	b	b	q_1	1, S	b, S	b, S
	b	b	b	q_4	b, S	b, L	b, L
q_4	b	0	0	q_4	b, S	0, L	0, L
	b	\$	\$	q_5	b, S	\$, S	\$, S

Nichtdeterministische Strategien können den Ablauf von Berechnungen vereinfachen. Als Beispiel werde die Sprache

$$L = \left\{ x\#y \mid x \in \{0,1\}^*, y \in \{0,1\}^*, x \neq e, x \neq y \right\}$$

betrachtet. Es sei $w \in \{0,1,\#\}^*$. Eine deterministische Turingmaschine würde bei Eingabe von *w* zunächst an der Position des Zeichens # feststellen, wo *y* beginnt, und dann x und y buchstabenweise auf einen Unterschied hin vergleichen (falls *w* kein oder mehr als ein Zeichen # enthält, würde *w* nicht akzeptiert werden). Eine nichtdeterministische Turingmaschine *NTM* würde ebenfalls (deterministisch) zunächst prüfen, ob *w* genau ein Zeichen # enthält. Ist dieses nicht der Fall, wird *w* nicht akzeptiert. Ist $w = x\#y$ und besteht *x* aus *n* Buchstaben und *y* aus *m* Buchstaben, etwa $x = x_1 \ldots x_n$ und $y = y_1 \ldots y_m$, so überprüft *NTM* (deterministisch), ob $n \neq m$ gilt (in diesem Fall wird *w* akzeptiert). Andernfalls erzeugt *NTM* auf einem seiner Arbeitsbänder *nichtdeterministisch* eine Zeichenkette $z \in \{0,1\}^*$ der Form $z = 0^{k_1}10^{k_2}$ mit $k_1 + k_2 + 1 = n = m$: Ist dieses Arbeitsband etwa das Band mit der höchsten Nummer und wurde es bisher in der Berechnung noch nicht verwendet (der Schreiblesekopf steht dort immer noch über der ersten Zelle), so enthält die Überführungsfunktion zur nichtdeterministischen Erzeugung von *z* den Wert

$$d(q, a_1, \ldots, a_{k-1}, b) = \left\{ \begin{array}{l} (q, (a_1, S), \ldots, (a_{k-1}, S), (0, R)), \\ (q, (a_1, S), \ldots, (a_{k-1}, S), (1, R)), \\ (q', (a_1, S), \ldots, (a_{k-1}, S), (b, S)) \end{array} \right\} \text{ mit } a_1 \in \Sigma, \ldots, a_{k-1} \in \Sigma.$$

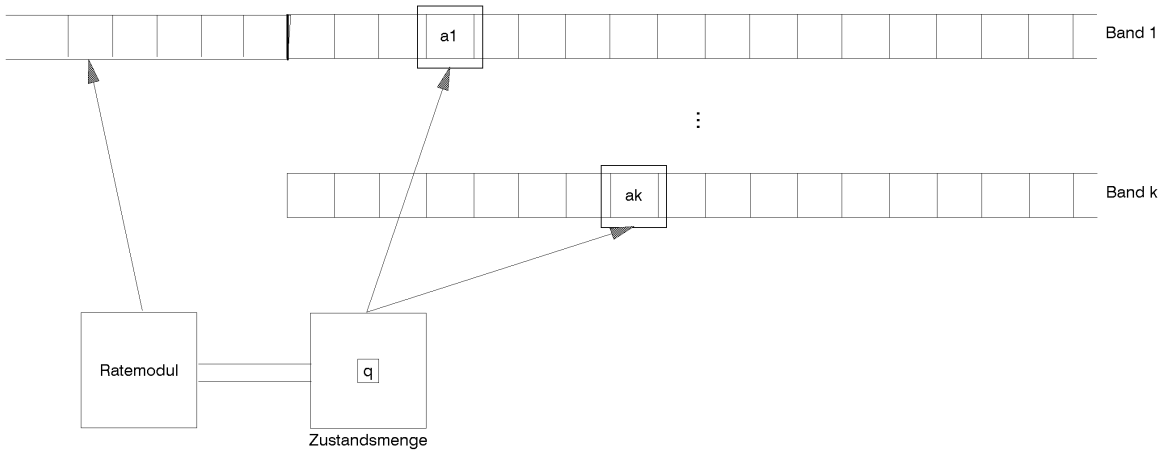
Hierbei sei Σ das Arbeitsalphabet von *NTM*, *q* derjenige Zustand, den *NTM* unmittelbar vor Erreichen des Erzeugungsvorgangs von *z* erreicht hatte, und q' ein Zustand, der den Erzeugungsvorgang von *z* beeendet.
Das Zeichen 1 in *z* gibt die Position an, an der sich *x* und *y* unterscheiden. Diese Position $i = k_1 + 1$ wird von *NTM* „nichtdeterministisch geraten". Anschließend überprüft *NTM* die Buchstaben x_i und y_i. Das Wort *w* wird genau dann akzeptiert, wenn $x_i \neq y_i$ ist. *NTM* rät also nichtdeterministisch die Position, an der sich *x* und *y* unterscheiden und verifiziert anschließend die Richtigkeit des Ratevorgangs. Die nichtdeterministisch erzeugte Zeichenkette *z* kann auch als Beweis (Zertifikat) dafür angesehen werden, daß $x \neq y$ bzw. $w \in L$ gilt.

Die in diesem Beispiel beschriebene Arbeitsweise ist für eine nichtdeterministische Turingmaschine typisch:

Die nichtdeterministische Turingmaschine *NTM* sei konzipiert , um die Menge $L \subseteq \Sigma^*$ zu akzeptieren. Bei einer Eingabe $x \in \Sigma^*$ **rät** *NTM* **auf nichtdeterministische** Weise einen **Beweis** *B* (ein **Zertifikat**) dafür, daß $x \in L$ gilt. Der Beweis ist eine Zeichenkette über einem endlichen Alphabet Σ_0, d.h. $B \in \Sigma_0^*$. Anschließend **verifiziert** *NTM* **den Beweis**.

Eine nichtdeterministische Berechnung kann auch so organisiert werden, daß zunächst alle „nichtdeterministischen Schritte" ausgeführt werden und anschließend nur noch deterministische Schritte erfolgen. Dieser Ansatz führt auf ein **Nichtstandardmodell** der NDTM:

Eine nichtdeterministische *k*-Band-Turingmaschine (*k*-NDTM) *TM* ist definiert durch $TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$. Die Überführungsfunktion ist eine partielle Abbildung $d : (Q \setminus \{q_{accept}\}) \times \Sigma^k \rightarrow \mathbf{P}(Q \times (\Sigma \times \{L, R, S\})^k)$, die sich in zwei Teile d_1 und d_2 ($d = d_1 \cup d_2$) zerlegen läßt. Alle nichtdeterministischen Teile von d sind in d_1 zusammengefaßt, d_2 besteht ausschließlich aus deterministischen Teilen. Das Eingabeband (1. Band) wird nach links abzählbar unendlich erweitert; die Zellen werden mit 0, -1, -2, ... numeriert. Zusätzlich verfügt *TM* über ein „Ratemodul".



Die Arbeitsweise von *TM* verläuft in zwei Phasen. Ein Eingabewort $w = a_1 \ldots a_n,\ w \in I^*$, wird auf das 1. Band in die Zellen mit Nummern 1, ..., *n* eingegeben (allen anderen Zellen enthalten das Leerzeichen, die Schreib/Leseköpfe stehen auf den jeweiligen Bändern über der ersten Zelle, der Schreibkopf des Ratemoduls steht über der Zelle mit Nummer -1). Nun beginnt Phase 1. Das Ratemodul steuert getaktet den Schreibkopf: in jedem Takt wird ein Symbol aus Σ auf das 1. Band geschrieben und der Schreibkopf um eine Zelle nach links bewegt, oder der Ratemodul stoppt und wird „inaktiv“. Dieser Vorgang wird durch den nichtdeterministischen Teil d_1 von d gesteuert. Die so vom Ratemodul auf dem 1. Band erzeugte Zeichenkette wird auch als **Beweis (Zertifikat)** bezeichnet. *TM* befindet sich im Anfangszustand q_0. Eventuell stoppt das Ratemodul nicht, so daß die Turingmaschine bei der entsprechenden Eingabe nicht anhält. Sobald das Ratemodul stoppt und *TM* in den Zustand q_0 geht, beginnt Phase 2. *TM* verhält sich deterministisch, gesteuert durch den deterministischen Teil d_2 von d, wie eine „normale“ *k*-DTM, wobei die Zeichenkette, die das Ratemodul in Phase 1 erzeugt hat, in die Berechnung einbezogen wird.

Phase 2 kann auch als **Verifikationsphase** von TM bezeichnet werden. Es wird nämlich die „Brauchbarkeit“ der in Phase 1 erzeugten („geratenen“) Zeichenkette, der in Phase 1 erzeugte Beweis, für die Berechnung verifiziert.

Die Zeitkomplexität und die Platzkomplexität (im schlechtesten Fall, worst case) einer nichtdeterministischen Turingmaschine wird ähnlich wie im deterministischen Fall definiert:

Für eine *k*-NDTM $TM = \left(Q, \Sigma, I, d, b, q_0, q_{accept} \right)$ und eine Eingabe $w \in L(TM)$ gelte
$(q_0, (w,1), (e,1), ..., (e,1)) \Rightarrow^m K_{accept}$
mit einer Endkonfiguration K_{accept}. Hierbei sei *m* der *kleinste* Wert, so daß eine Endkonfiguration erreicht wird. Dann wird durch $t_{TM}(w) = m$ eine partielle Funktion $t_{TM} : \Sigma^* \to \mathbf{N}$ definiert, die angibt, wieviele Überführungen *TM in der kürzesten Berechnung* macht, um *w* zu akzeptieren (eventuell ist $t_{TM}(w)$ nicht definiert, nämlich dann, wenn die Eingabe von *w* nicht auf eine Endkonfiguration führt).

Die **Zeitkomplexität** von *TM* (**im schlechtesten Fall, worst case**) wird definiert durch die partielle Funktion $T_{TM} : \mathbf{N} \to \mathbf{N}$ mit
$T_{TM}(n) = \max\{ t_{TM}(w) \mid w \in \Sigma^* \text{ und } |w| \le n \}$.

Entsprechend kann man die **Platzkomplexität** von *TM* (**im schlechtesten Fall, worst case**) $S_{TM}(n)$ als den maximalen Abstand vom linken Ende eines Bandes definieren, den ein Schreib/Lesekopf bei der Akzeptanz eines Worts $w \in L(TM)$ mit $|w| \le n$ *mindestens* erreicht.

Beispielsweise ist die oben angegebene Turingmaschine zur Lösung des Partitionenproblems (2*n* + 2)-zeitbeschränkt und (*n* + 1)-raumbeschränkt.

Eine Funktion $T : \mathbf{N} \to \mathbf{N}$ heißt **zeitkonstruierbar**, wenn es eine *k*-DTM *TM* gibt, die bei Eingabe eines Wortes *w* der Länge *n* auf dem *k*-ten Band (Ausgabeband) die Zeichenkette $0^{T(n)}$ generiert und im akzeptierenden Zustand stoppt und dabei eine Anzahl Schritte höchstens der Ordnung $O(T(n))$ benötigt.

Der Funktionswert einer zeitkonstruierbaren Funktion kann also deterministisch berechnet werden, wobei die Anzahl der ausgeführten Schritte in derselben Größenordnung wie der Funktionswert liegt. Die meisten gewöhnlichen monotonen Funktionen mit $T(n) \ge n$ sind zeitkonstruierbar.

Der folgende Satz zeigt, daß sich nichtdeterministisches Verhalten deterministisch simulieren läßt:

Satz 2.5-1:

Falls *L* von einer *k*-NDTM *TM* akzeptiert wird, dann gibt es eine k'-DTM *TM'* mit $L(TM') = L(TM)$. Man kann *TM'* so konstruieren, daß gilt:

Falls *TM* $T(n)$-zeitbeschränkt mit einer zeitkonstruierbaren Funktion $T(n) \geq n$ ist, dann ist *TM'* $O\left(T(n) \cdot d^{T(n)}\right)$-zeitbeschränkt mit einer Konstanten $d \geq 2$.

Beweis:

Der Beweis wird zunächst für den Fall skizziert, daß *TM* $T(n)$-zeitbeschränkt mit einer zeitkonstruierbaren Funktion $T(n) \geq n$ ist.

Es sei *TM* eine *k*-NDTM, die $T(n)$-zeitbeschränkt mit einer zeitkonstruierbaren Funktion $T(n) \geq n$ ist. Für jedes Wort $w \in L(TM)$ mit $|w| = n$ gibt es also eine Folge von Konfigurationen, die von einer Anfangskonfiguration K_0 mit *w* zu einer akzeptierenden Konfiguration K_{accept} führt und deren Länge $\leq T(n)$ ist.
Jede Zeile der Tabelle, mit der die Überführungsfunktion d von *TM* gegeben wird (Überführungstabelle), hat die Form

$$d(q, a_1, ..., a_k) = \{(q_1, (b_{11}, d_{11}), ..., (b_{1k}, d_{1k})), ..., (q_t, (b_{t1}, d_{t1}), ..., (b_{tk}, d_{tk}))\}$$

(hier stehen *t* „Teilzeilen"). Wenn in einer Konfigurationenfolge $d(q, a_1, ..., a_k)$ angewendet wird, gibt es *t* mögliche Folgekonfigurationen. Der maximale Wert *t* an Teilzeilen in einer Zeile der Überführungstabelle sei $d \geq 2$. In einer Berechnung von *TM* gibt es für eine Konfiguration maximal *d* mögliche Folgekonfigurationen. Es sei $D = \{0, 1, ..., d-1\}$. Dann kann man eine von *TM* ausgeführte Berechnung bzw. die dabei durchlaufene Konfigurationsfolge der Länge *l* durch ein Wort $x = d_1 ... d_l$ über *D* beschreiben: d_i gibt an, daß in der *i*-ten Überführung die $(d_i + 1)$-te Alternative in der entsprechenden Zeile der Überführungstabelle verwendet wird. Eventuell beschreibt nicht jedes Wort über *D* gültige Konfigurationenfolgen von *TM*.

Eine Simulation des Verhaltens von *TM* durch eine deterministische Turingmaschine *TM'* kann folgendermaßen durchgeführt werden. Eine Eingabe $w \in I^*$ wird auf einem Band von *TM'* zwischengespeichert (es wird eine Kopie angelegt). *TM'* erzeugt ein Wort $x \in D^*$ mit $|x| = l \leq T(n)$, $x = d_1 ... d_l$, falls es noch ein nicht bereits erzeugtes Wort dieser Art gibt. Insgesamt können die Wörter in lexikographischer Reihenfolge erzeugt werden. Hierzu muß einmal der Wert $T(n)$ berechnet werden. Aufgrund der Voraussetzung der Zeitkonstruierbarkeit von $T(n)$ kann dieses determininistisch in einer Anzahl von Schritten der Ordnung $O(T(n))$ erfolgen. Dann kopiert *TM'* das Eingabewort *w* aus dem Zwischenspeicher auf das Eingabeband von *TM* und simuliert auf deterministische Weise das Verhalten von *TM* für *l*

Schritte, wobei in der *i*-ten Überführung die $(d_i + 1)$-te Alternative in der entsprechenden Zeile der Überführungstabelle verwendet wird. Falls die Simulation auf den akzeptierenden Zustand führt, wird das Wort *w* akzeptiert, ansonsten wird das nächste Wort $x \in D^*$ erzeugt und die Simulation erneut ausgeführt.

Um ein Wort $x \in D^*$ mit $|x| = l \leq T(n)$ zu erzeugen, benötigt man $O(l)$ viele Schritte (entsprechend der Addition einer 1 auf eine Zahl mit *l* Stellen). Anschließend wird *w* in $O(n)$ vielen Schritten aus dem Zwischenspeicher auf das Eingabeband von *TM* kopiert. Das Verhalten von *TM* wird dann für *l* viele Schritte simuliert. Es gibt d^l viele Möglichkeiten für $x \in D^*$ mit $|x| = l$. Insgesamt ergibt sich ein zeitlicher Aufwand der Größe

$$T'(n) = c_0 \cdot T(n) + \sum_{l=0}^{T(n)} (c_1 \cdot l + c_2 \cdot n + c_3 \cdot l) \cdot d^l$$

(der erste Term gibt den zeitlichen Aufwand zur anfänglichen Berechnung von $T(n)$ an, der erste Term unter dem Summenzeichen beschreibt den zeitlichen Aufwand, um *x* zu erzeugen, der zweite Term, um *w* aus dem Zwischenspeicher auf das Eingabeband von *TM* zu kopieren, der dritte Term, um *l* Schritte von *TM* zu simulieren). Es gilt

$$T'(n) \leq c\left(T(n) + \sum_{l=0}^{T(n)} \left(l \cdot d^l + n \cdot \sum_{l=0}^{T(n)} d^l\right)\right) = c\left(T(n) + \frac{d - (T(n)+1)d^{T(n)+1} + T(n)d^{T(n)+2}}{(d-1)^2} + n\frac{d^{T(n)+1} - 1}{d-1}\right)$$

mit einer geeigneten Konstanten *c*. Der Ausdruck rechts ist von der Ordnung $O\left(T(n) \cdot d^{T(n)}\right)$; hierbei wurde $n \leq T(n)$ verwendet.

Ist man lediglich an der deterministischen Simulation des nichtdeterministischen Verhaltens von *TM* interessiert, entfällt die anfängliche Berechnung von $T(n)$. Es werden dann nacheinander alle Wörter $x \in D^*$ (beliebiger Länge) in lexikographischer Reihenfolge erzeugt und jeweils der durch *x* beschriebene Berechnungsablauf deterministisch simuliert. ///

Die deterministische Simulation der *k*-NDTM *TM* erfolgt also zum Preis einer exponentiellen Steigerung des Laufzeitverhaltens, denn $O\left(T(n) \cdot d^{T(n)}\right) \subseteq O\left(C^{T(n)}\right)$ mit einer Konstanten $C > 0$: es gilt $c \cdot T(n) \cdot d^{T(n)} \leq c \cdot d^{T(n)} \cdot d^{T(n)} = c \cdot d^{2T(n)} = c \cdot \left(d^2\right)^{T(n)}$, d.h. man kann $C = d^2$ nehmen.

Bisher ist keine nicht-triviale untere Schranke für die Simulation einer nichtdeterministischen Turingmaschine durch eine deterministische Turingmaschine bekannt. Insbesondere ist nicht bekannt, ob eine nichtdeterministische Turingmaschine, deren Laufzeit durch ein Polynom begrenzt ist, durch eine deterministische Turingmaschine simuliert werden kann, deren Laufzeit ebenfalls ein Polynom ist (eventuell von einem sehr viel höheren Grad).

Anders verhält es sich bei Betrachtung der Platzkomplexität:

Eine Funktion $S : \mathbf{N} \to \mathbf{N}$ heißt **platzkonstruierbar**, wenn es eine *k*-DTM *TM* gibt, die bei Eingabe eines Wortes der Länge *n* ein spezielles Symbol in die $S(n)$-te Zelle eines ihrer Bänder schreibt, ohne jeweils mehr als $S(n)$ viele Zellen auf allen Bändern zu verwenden.

Satz 2.5-2:

Ist *TM* eine *k*-NDTM mit einer platzkonstruierbaren Speicherplatzkomplexität $S(n)$, dann gibt es eine k'-DTM TM' mit einer Speicherplatzkomplexität der Ordnung $O\left(S^2(n)\right)$ und $L(TM') = L(TM)$.

Beweis:

Es sei $TM = \left(Q, \Sigma, I, d, b, q_0, q_{accept}\right)$ eine *k*-NDTM mit einer platzkonstruierbaren Speicherplatzkomplexität $S(n)$. Es wird ein deterministischer Algorithmus, d.h. eine deterministische k'-DTM TM', angegeben, der das Verhalten von *TM* bei Eingabe eines Wortes *w* mit $|w| = n$ simuliert und dessen Speicherplatzbedarf durch einen Wert der Größenordnung $O\left(S^2(n)\right)$ beschränkt ist. Es sei $K = \left(q, (a_1, i_1), ..., (a_k, i_k)\right)$ eine Konfiguration in einer Berechnung von *TM* bei Eingabe von *w*. Wegen $|a_j| \le S(n)$ und $1 \le i_j \le S(n)$ für *j* = 1, ..., *k* ist die Anzahl verschiedener Konfigurationen durch $|Q| \cdot |\Sigma|^{k \cdot S(n)} \cdot \left(S(n)\right)^k \le c^{S(n)}$ mit einer Konstanten $c > 0$ begrenzt. Falls in einer Berechnung von *TM* also $K_1 \Rightarrow^* K_2$ gilt, dann kann man annehmen, daß dabei höchstens $c^{S(n)}$ viele Überführungen vorkommen. Es gilt: $K_1 \Rightarrow^* K_2$ in höchstens *i* Schritten genau dann, wenn es eine Konfiguration K_3 gibt, so daß $K_1 \Rightarrow^* K_3$ in höchstens $\lceil i/2 \rceil$ vielen Schritten und $K_3 \Rightarrow^* K_2$ in höchstens $\lfloor i/2 \rfloor$ vielen Schritten abläuft. Für K_3 kommen nur $c^{S(n)}$ viele Möglichkeiten in Frage. Diese Überlegung führt auf folgenden Algorithmus:

Eingabe: Eine *k*-NDTM $TM = \left(Q, \Sigma, I, d, b, q_0, q_{accept}\right)$ mit einer platz-konstruierbaren Speicherplatzkomplexität $S(n)$ und ein Wort $w \in I^*$ mit $|w| = n$

Verfahren: Aufruf der Funktion `space_Simulation (`*TM*`, w)`

Ausgabe: `TRUE`, falls $w \in L(TM)$, `FALSE` sonst.

Die Funktion `space_Simulation` wird in Pseudocode beschrieben; sie besitzt zwei Formalparameter `TM` und `w`, über die die Beschreibung einer *k*-NDTM und ein Eingabewort für

diese Turingmaschine eingegeben werden. Innerhalb von `space_Simulation` wird eine Funktion `test` verwendet, die als Parameter zwei Konfigurationen `K1` und `K2` und eine natürliche Zahl `i` hat. Auf die genaue syntaktische Spezifikation soll hier verzichtet werden.

```
FUNCTION space_Simulation (TM : ...;
                           w  : ...) : BOOLEAN;

VAR Kf : ...;
    K0 : ...;
    OK : BOOLEAN;

     FUNCTION test (K1 :...;
                    K2 : ...;
                    i  : INTEGER) : BOOLEAN;
       VAR resultat : BOOLEAN;
           K3       : ...;
       BEGIN { test }
         resultat := FALSE;
         IF i = 1 THEN BEGIN
                         IF (K1⇒K2) OR (K1 = K2)
                         THEN resultat := TRUE;
                       END
                  ELSE BEGIN
                         Für jede Konfiguration K3 = (q,(a_1,i_1),...,(a_k,i_k))
                         mit |a_j| ≤ S(n) und 1 ≤ i_j ≤ S(n) DO
                         BEGIN
                           resultat := test (K1, K3, (i+1) DIV 2)
                                       AND
                                       test (K3, K2, i DIV 2);
                           IF resultat = TRUE THEN Break;
                         END;
                       END;
         test := resultat;
       END    { test };

BEGIN { space_Simulation }
  K0 := (q_0,(w,1),(e,1),...,(e,1))
  OK := FALSE;
  FOR_EACH Endkonfiguration Kf = (q_f,(a_1,|a_1|+1),...,(a_k,|a_k|+1)) mit |a_j| ≤ S(n)
    DO BEGIN
         OK := test (K0, Kf, c^S(n));
         IF OK THEN Break;
       END;
 space_Simulation := OK;
END   { space_Simulation };
```

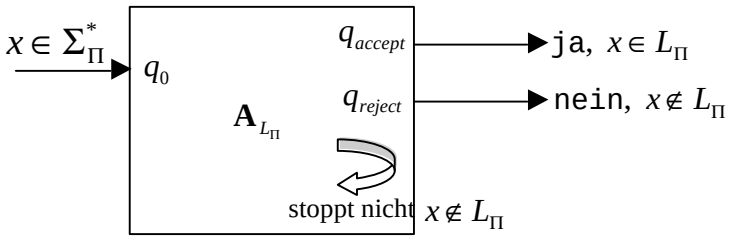
Eine genauere Untersuchung der Aufrufe der rekursiven Funktion `test` zeigt, daß bei jedem Aufruf innerhalb von `test` der dritte Parameter im wesentlichen halbiert wird. Der maximale Wert des dritten Parameters ist `i` = $c^{S(n)}$. Daher werden zu keinem Zeitpunkt der (rekursiven) Abarbeitung von `test` mehr als $1+\log(\lfloor c^{S(n)} \rfloor) \in O(S(n))$ viele Aktivierungsrecords auf dem Stack benötigt. Jeder Aktivierungsrecord hat eine Größe der Ordnung $O(S(n))$, so daß der Speicherplatzbedarf von der Ordnung $O(S^2(n))$ ist. ///

In Kapitel 1.3 und Kapitel 2.1 wird ein Algorithmus für ein Entscheidungsproblem wie folgt definiert. Als Programm in einer Programmiersprache beschrieben, handelt es sich dabei um einen **deterministischen Algorithmus**.

Ein **deterministischer Algorithmus $\mathbf{A}_{L_\Pi}$ für ein Entscheidungsproblem** Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ hat die Schnittstellen und die Form

Eingabe: $x \in \Sigma_\Pi^*$

Ausgabe: `ja` (accept), falls $x \in L_\Pi$ gilt
`nein` (reject), falls $x \notin L_\Pi$ gilt
falls $\mathbf{A}_{L_\Pi}$ bei einer Eingabe $x \in \Sigma_\Pi^*$ nicht hält, gilt ebenfalls $x \notin L_\Pi$.



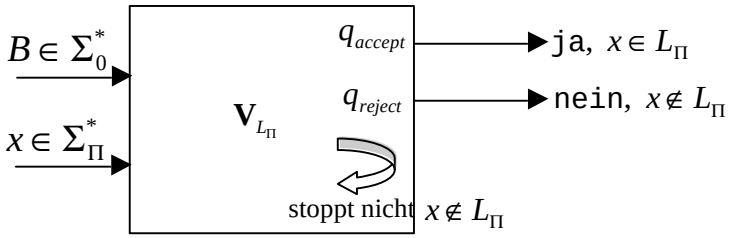
Der Konzept des Nichtdeterminismus läßt sich auf Algorithmen (formuliert als Programm in einer Programmiersprache, vgl. Kapitel 2.3) übertragen und führt auf die Definition **eines nichtdeterministischen Algorithmus unter Einsatz eines Verifizierers**. Dabei ist ein **Verifizierer für das Entscheidungsproblem** Π über einem Alphabet Σ_Π ein deterministischer Algorithmus $\mathbf{V}_{L_\Pi}$, der eine Eingabe $x \in \Sigma_\Pi^*$ und eine **Zusatzinformation** (einen **Beweis**, ein **Zertifikat**) $B \in \Sigma_0^*$ lesen kann und die Frage „$x \in L_\Pi$?“ mit Hilfe des Beweises B entscheidet. Die Bereitstellung des Beweises B ist nicht Aufgabe des Verifizierers; B wird „von außen“ vorgegeben. Eventuell stoppt der Verifizierer bei Eingabe von $x \in \Sigma_\Pi^*$ nicht. **Der Veri-**

fizierer repräsentiert also die Verifikationsphase im Nichtstandardmodell einer nichtdeterministischen Turingmaschine; der erforderliche Beweis wird zuvor vom **Ratemodul auf nichtdeterministische Weise** erzeugt, **wobei der Ratemodul nur einmal durchlaufen wird**. Der Verifizierer überprüft mit Hilfe des Beweises $B \in \Sigma_0^*$, ob die Eingabe $x \in \Sigma_\Pi^*$ die Eigenschaft „$x \in L_\Pi$“ besitzt, d.h. eine Eigenschaft, durch die die Elemente aus L_Π definiert werden. Er akzeptiert in diesem Fall die Eingabe *x*; andernfalls verwirft er sie oder stoppt nicht.

Ein **Verifizierer $\mathbf{V}_{L_\Pi}$ für das Entscheidungsproblem** Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ hat die Schnittstellen und die Form

Eingabe: $x \in \Sigma_\Pi^*$, $B \in \Sigma_0^*$

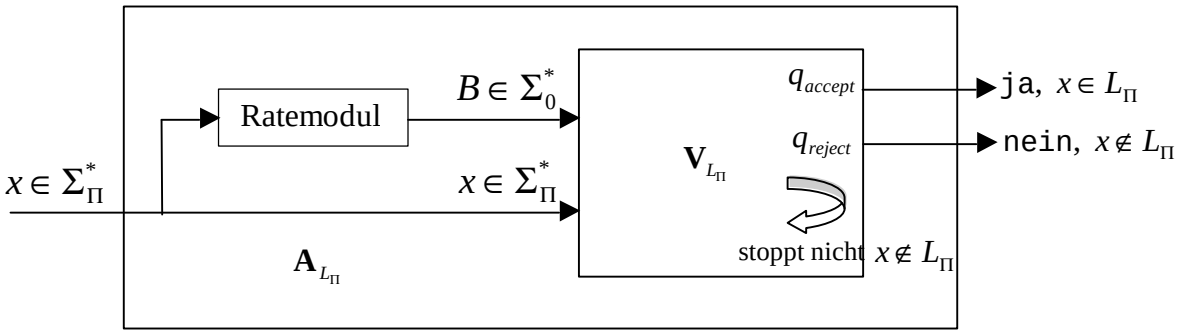
Ausgabe: `ja` (accept), falls die in *B* enthaltene Information die Eigenschaft $x \in L_\Pi$ belegt
`nein` (reject), falls die in *B* enthaltene Information die Eigenschaft $x \notin L_\Pi$ belegt
falls $\mathbf{V}_{L_\Pi}$ bei einer Eingabe $x \in \Sigma_\Pi^*$ nicht stoppt, gilt $x \notin L_\Pi$



Falls $\mathbf{V}_{L_\Pi}$ bei Eingabe von $x \in \Sigma_\Pi^*$ stoppt, wird mit $\mathbf{V}_{L_\Pi}(x,B)$, $\mathbf{V}_{L_\Pi}(x,B) \in \{\texttt{ja, nein}\}$, die Entscheidung von $\mathbf{V}_{L_\Pi}$ bezeichnet.

Damit ergibt sich:

Ein **nichtdeterministischer Algorithmus $\mathbf{A}_{L_\Pi}$ für ein Entscheidungsproblem** Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ hat die Form



Es gilt für $x \in \Sigma_\Pi^*$:

$x \in L_\Pi$, falls es einen Beweis $B_x \in \Sigma_0^*$ gibt, so daß $\mathbf{V}_{L_\Pi}$ bei Eingabe von x und B_x mit $\mathbf{V}_{L_\Pi}(x, B_x) = \texttt{ja}$ stoppt;

$x \notin L_\Pi$, falls für jeden Beweis $B \in \Sigma_0^*$ gilt: entweder stoppt $\mathbf{V}_{L_\Pi}$ bei Eingabe von x und B nicht, oder $\mathbf{V}_{L_\Pi}$ stoppt mit $\mathbf{V}_{L_\Pi}(x, B) = \texttt{nein}$.

Die **Laufzeit eines nichtdeterministischen Algorithmus** bei Eingabe von $x \in \Sigma_\Pi^*$ mit $|x| = n$ ist die Anzahl der Schritte, einschließlich der nichtdeterministischen Ratephase, bis der Algorithmus zur Entscheidung kommt. Die Laufzeit wird in Abhängigkeit von $|x| = n$ angegeben.

Beispiel:

Das Partitionenproblem mit ganzzahligen Eingabewerten

Instanz: $I = \{a_1, ..., a_n\}$

I ist eine Menge von n natürlichen Zahlen mit $a_i > 0$ für $i = 1, ..., n$. Als Größe der Eingabe wird $size(I) = k = n \cdot A$ mit $A = \max\{\lfloor \log_2(a_i) \rfloor + 1 \mid i = 1, ..., n\}$ genommen, d.h. k gibt eine obere Schranke für die Anzahl der Bits an, um I darzustellen

Lösung: Entscheidung „ja“, falls es eine Teilmenge $J \subseteq \{1, ..., n\}$ mit $\sum_{i \in J} a_i = \sum_{i \notin J} a_i$ gibt (d.h. wenn sich die Eingabeinstanz in zwei Teile zerlegen läßt, deren jeweilige Summen

gleich groß sind).
Entscheidung „nein“ sonst.

Ein nichtdeterministischer Algorithmus $\mathbf{A}_{PARTITION}(I)$ zur Lösung des Partitionenproblems wird in folgendem Pesudocode vorgeschlagen:

```
CONST n = ...;    { Problemgröße }

FUNCTION A_PARTITION(I): ...;

VAR S: INTEGER;
    F: ARRAY [1..2] OF INTEGER;
    i: 1..n;
    J: ARRAY [1..n] OF INTEGER;

BEGIN { A_PARTITION(I: ...) }
  { Ratephase: }
  FOR i := 1 TO n DO
   setze (nichtdeterministisch) J[i] := 1 bzw. J[i] := 2;

  { Verifizierer, Verifikationsphase }
  S := 0;
  FOR i := 1 TO n DO                    { Zeile 1 }
    S := S + a_i;                       { Zeile 2 }
  IF (S MOD 2) = 1
  THEN A_PARTITION := nein
  ELSE BEGIN
         F[1] := 0;
         F[2] := 0;
         FOR i := 1 TO n DO             { Zeile 3 }
           F[J[i]] := F[J[i]] + a_i;    { Zeile 4 }
         IF F[1] = F[2]                 { Zeile 5 }
         THEN A_PARTITION := ja
         ELSE A_PARTITION := nein;
       END;
END    { A_PARTITION(I) };
```

Die Zeitkomplexität wird in Abhängigkeit von der Größe der Eingabe gemessen, wobei hier wieder die Anzahl der erforderlichen Bitoperationen abgeschätzt wird. Man kann sich leicht davon überzeugen, daß die Ratephase dabei einen Aufwand der Ordnung $O(n)$ erzeugt. In

den Zeilen 1 und 2 wird mit *n* Additionen der Wert $S = \sum_{i=1}^{n} a_i$ berechnet. Das Ergebnis $\sum_{l=1}^{i} a_l$ jeder einzelnen Addition in Zeile 2 hat eine Bitlänge $\leq \lfloor \log_2(S) \rfloor + 1$, so daß eine Ausführung der Operation in Zeile 2 höchstens $c_1 \cdot \lfloor \log_2(S) \rfloor$ viele Bitoperationen (mit einer Konstanten $c_1 > 0$) erfordert. Die Anzahl der Bitoperationen für die Zeilen 1 und 2 zusammen läßt sich durch

$$c_1 \cdot n \cdot \lfloor \log_2(S) \rfloor \leq c_1 \cdot n \cdot \log_2(S) \leq c_1 \cdot n \cdot \sum_{i=1}^{n} \log_2(a_i)$$

$$\leq c_1 \cdot n \cdot \sum_{i=1}^{n} (\lfloor \log_2(a_i) \rfloor + 1) \leq c_1 \cdot n \cdot \sum_{i=1}^{n} A = c_1 \cdot n^2 \cdot A$$

abschätzen. In Zeile 4 wird entweder zu `S[1]` oder zu `S[2]` addiert. In jedem Fall ist das Ergebnis $\leq S$, so daß für die Zeilen 3 und 4 ein Aufwand der Ordnung $O(n^2 \cdot A)$ resultiert. Der Aufwand in Zeile 5 ist von der Ordnung $O(\log_2(S)) \subseteq O(n \cdot A)$. Der Gesamtaufwand läßt sich daher durch einen Wert der Ordnung $O(n^2 \cdot A)$ abschätzen. Wegen $a_i > 0$ für $i = 1, ..., n$ ist $1 \leq A$ und damit $n^2 \cdot A \leq (n \cdot A)^2 = k^2$. Mit $k = size(I)$ und $k \in O(n \cdot A)$ ergibt sich eine obere Schranke für den Aufwand dieses nichtdeterministischen Algorithmus der Ordnung $O(k^2)$, also nichtdeterministisch polynomielles Laufzeitverhalten (gemessen in Bitoperationen).

Beispiel:

Erfüllbarkeit Boolescher Ausdrücke

Nicht jeder Boolesche Ausdruck, selbst wenn er syntaktisch korrekt ist, ist erfüllbar. Mit Hilfe eines deterministischen Algorithmus kann man bespielsweise die Erfüllbarkeit eines Booleschen Ausdrucks *F* mit *n* Variablen dadurch testen, daß man systematisch nacheinander alle 2^n möglichen Belegungen der Variablen in *F* mit Wahrheitswerten `TRUE` bzw. `FALSE` erzeugt. Immer, wenn eine neue Belegung generiert worden ist, wird diese in die Variablen eingesetzt und der Boolesche Ausdruck ausgewertet. Die Entscheidung, ob *F* erfüllbar ist oder nicht, kann u.U. erst nach Überprüfung aller 2^n Belegungen erfolgen. Das Verfahren ist daher von der Ordnung $O(2^n)$.

Ein nichtdeterministischer Algorithmus würde bei Eingabe eines Booleschen Ausdrucks *F* mit *n* Variablen (in geeigneter Kodierung) im Ratemodul zunächst nichtdeterministisch eine 0-1-Folge der Länge *n* erzeugen. Anschließend wird diese 0-1-Folge durch einen Verifizierer als Belegung der *n* Variablen interpretiert (0 entspricht `FALSE`, 1 entspricht `TRUE`). Der Beweis B_F ist hier die 0-1-Folge bzw. deren Interpretation als Belegung der in *F* vorkommenden Variablen. Die durch B_F angegebene Belegung durch den Verifizierer in *F* eingesetzt, *F* ausgewertet und die Entscheidung „*F* ist erfüllbar“ genau dann getroffen, wenn bei dieser Aus-

wertung der Wahrheitswert TRUE entsteht. Der Ratemodul kann, falls *F* erfüllbar ist, in der Tat eine 0-1-Folge erzeugen, die einer erfüllenden Belegung der Variablen von *F* entspricht. Das geht nicht, wenn *F* nicht erfüllbar ist; in diesem Fall wird die Auswertung jeder in *F* eingesetzten Belegung, die einer vom Ratemodul erzeugten 0-1-Folge entspricht, den Wahrheitswert FALSE ergeben.

Der Verifizierer für das Erfüllbarkeitsproblem können so entworfen werden, daß er ein Laufzeitverhalten der Ordnung $O(n)$ hat. Insgesamt ist die Laufzeit dieses nichtdeterministischen Algorithmus also von der Ordnung $O(n)$, da der Ratemodul in diesem Beispiel in linearer Zeit arbeitet. Es ist nicht bekannt, ob es einen *deterministischen* Algorithmus gibt, der die Erfüllbarkeit Boolescher Ausdrücke mit einem Laufzeitverhalten der Ordnung $O(p(n))$ mit einem Polynom *p* testet.

Ein nichtdeterministischer Algorithmus ist also zunächst ein **Konzept** oder **Gedankenmodell**, da nicht explizit gesagt wird, wie für den Fall $x \in L_\Pi$ vom Ratemodul der korrekte Beweis $B_x \in \Sigma_0^*$ erzeugt wird. Mit einer ähnlichen Argumentation wie im Beweis zu Satz 2.5-1 kann man jedoch das Verhalten eines nichtdeterministischen Algorithmus deterministisch simulieren und erhält damit einen funktional gleichwertigen „normalen“ deterministischen Algorithmus. Die folgende Betrachtung soll dabei auf den Fall beschränkt werden, daß der Verifizierer, der innerhalb des nichtdeterministischen Algorithmus einen vom Ratemodul erzeigten Beweis *B* verifiziert, stets zu einer ja/nein-Entscheidung kommt, d.h. für jede Eingabe *x* und *B* mit einer Antwort stoppt. Außerdem habe das Laufzeitverhalten des gesamten nichtdeterministischen Algorithmus die Ordnung $O(T(n))$ mit einer zeitkonstruierbaren Funktion $T(n) \geq n$. Bei Eingabe von $x \in \Sigma_\Pi^*$ kommt der Algorithmus also innerhalb einer Laufzeit zur ja/nein-Entscheidung, die durch $c \cdot T(|x|)$ mit einer Konstanten $c > 0$ beschränkt ist. Daher hat der vom Ratemodul erzeugte Beweis eine durch $c \cdot T(|x|)$ beschränkte Länge.

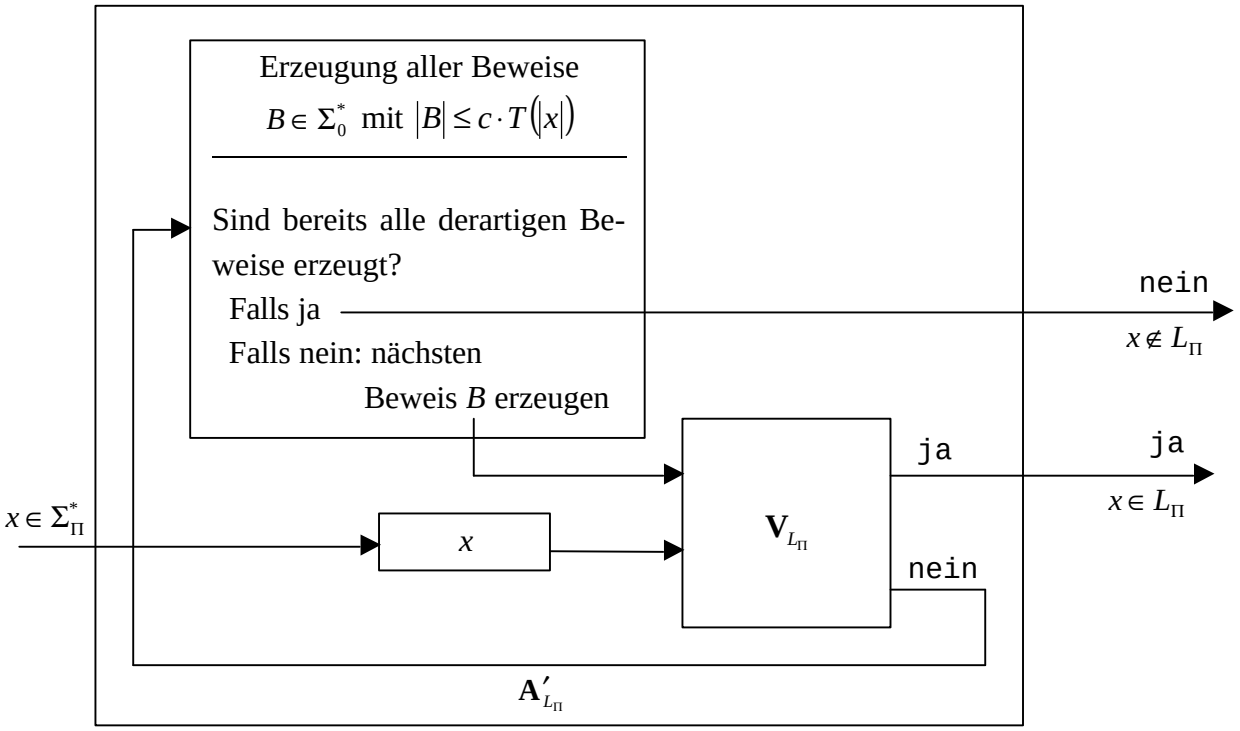
Satz 2.5-3:

Es sei $\mathbf{A}_{L_\Pi}$ ein nichtdeterministischer Algorithmus für ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$, der seine Entscheidung mit Hilfe des Verifizierers $\mathbf{V}_{L_\Pi}$ in einer Laufzeit trifft, die die Ordnung $O(T(n))$ mit einer zeitkonstruierbaren Funktion $T(n) \geq n$ besitzt. Dann gibt es einen deterministischen Algorithmus $\mathbf{A}'_{L_\Pi}$, der für jedes $x \in \Sigma_\Pi^*$ mit $|x| = n$ entscheidet, ob $x \in L_\Pi$ gilt oder nicht und die Zeitkomplexität $O(T(n) \cdot d^{O(T(n))})$ mit einer Konstanten $d \geq 2$ besitzt.

Bemerkung: Es gilt $O(T(n) \cdot d^{O(T(n))}) \subseteq O(2^{O(T(n))})$.

Beweis:

Die Simulation erzeugt bei Eingabe von $x \in \Sigma_{\Pi}^{*}$ mit $|x| = n$ systematisch alle Beweise $B \in \Sigma_0^{*}$ mit $|B| \leq c \cdot T(n)$. Dazu muß natürlich der Wert $T(n)$ in „kontrollierbarer Zeit“ berechenbar sein (das trifft zu, da *T* als zeitkonstruierbar angenommen wird). Die folgende Abbildung gibt die wesentlichen Bestandteile der Simulation:



Der folgende Pseudocode für $\mathbf{A}'_{L_{\Pi}}$ beschreibt die wesentlichen Aspekte der Simulation:

Bemerkung: Ein Beweis *B*, den der Verifizierer liest, ist ein Wort über einem Alphabet Σ_0.

```
FUNCTION A'_L(x);

VAR n       : INTEGER;
    lng     : INTEGER;
    B       : Σ0*;
    antwort : BOOLEAN;
    weiter  : BOOLEAN;

BEGIN { A'_L(x) }
  n       := size(x);      { Größe der Eingabe             }
  lng     := C * T(n);     { maximale Länge eines Beweises }
```

```
  antwort := FALSE;
  weiter  := TRUE;

  WHILE weiter DO
    IF (es sind bereits alle Wörter aus Σ0* mit Länge ≤ lng erzeugt worden)
    THEN weiter := FALSE
    ELSE
      BEGIN
        B:= nächstes Wort aus Σ0* mit Länge ≤ lng;
        IF  V_Π(x,B) = ja
        THEN BEGIN
               antwort := TRUE;
               weiter  := FALSE;
             END;
      END;

  CASE antwort OF
  TRUE  : A'_Π(x) := ja;
  FALSE : A'_Π(x) := nein;
  END;

END   { A'_{L_Π}(x) };
```

Die Anzahl der zu überprüfenden Beweise $B \in \Sigma_0^*$ mit $|B| \le c \cdot T(n)$ ist von der Ordnung $O\left(|\Sigma_0|^{c \cdot T(n)}\right)$. Damit ergibt sich wie im Beweis von Satz 2.5-1 die Komplexitätsabschätzung der Simulation. ///

3 Grenzen der Berechenbarkeit

In Kapitel 2 werden zwei Ansätze vorgestellt, um Berechenbarkeit zu modellieren. Beide Modelle haben sich als äquivalent erwiesen in dem Sinn, daß die Fähigkeit, etwas zu berechnen, beiden Modellen in gleicher Weise zukommt. Selbst das Konzept des Nichtdeterminismus erweitert nicht die Berechenbarkeit, da nichtdeterministisches Verhalten deterministisch simuliert werden kann, auch wenn dabei die Dauer einer Berechnung exponentiell wächst. Von den Modellen, die in Kapitel 2 behandelt werden, zeichnet sich das Modell der Turingmaschine aufgrund seiner Einfachheit und Universalität und nicht zuletzt aus historischen Gründen gegenüber den anderen Modellen aus.

Im folgenden wird unter dem Begriff Turingmaschine eine *k*-DTM verstanden (siehe Kapitel 2.1) mit einer dem jeweiligen Problem angemessenen geeigneten Anzahl *k* an Bändern.

3.1 Jenseits der Berechenbarkeit

Es sei Σ ein endliches Alphabet und $L \subseteq \Sigma^*$. Die Menge *L* heißt **rekursiv aufzählbar**, wenn es eine Turingmaschine *TM* gibt mit $L = L(TM)$.

Die Bezeichnung *rekursiv aufzählbar* leitet sich aus der Tatsache her, daß eine von einer Turingmaschine akzeptierte Menge $L \subseteq \Sigma^*$ durch eine totale (d.h. überall definierte) berechenbare Funktion „aufgezählt“ werden kann. Genauer:

Satz 3.1-1:

$L \subseteq \Sigma^*$ mit $L \neq \varnothing$ ist genau dann rekursiv aufzählbar, wenn es eine totale berechenbare Funktion $h : (\Sigma \cup \{0,1,\#\})^* \to \Sigma^*$ gibt mit $L = h\left((\Sigma \cup \{0,1,\#\})^*\right)$.

Die Funktion *h* „zählt *L* rekursiv auf“ (erzeugt *L*).

Beweis:

Diese Aussage enthält zwei „Richtungen“, deren Beweise beide eine genauere Betrachtung verdienen:

Die eine „Richtung“ dieser Aussage, nämlich der Nachweis der Existenz einer totalen und berechenbaren Funktion $h : (\Sigma \cup \{0,1,\#\})^* \to \Sigma^*$ mit $L = h\left((\Sigma \cup \{0,1,\#\})^*\right)$, d.h. einer Funktion *h*, die *L* aufzählt, wird folgendermaßen bewiesen. Wegen $L \neq \varnothing$ gibt es ein Wort $w_0 \in L$. Da

L als rekursiv aufzählbar angenommen wird, gibt es eine Turingmaschine TM mit $L = L(TM)$. Es wird eine Turingmaschine TM' angegeben, die zwei Bänder mehr als TM besitzt (nämlich ein zusätzliches Eingabeband und ein Ausgabeband), bei jeder Eingabe stoppt und die gesuchte Funktion $h : (\Sigma \cup \{0,1,\#\})^* \to \Sigma^*$ berechnet:

Bei Eingabe von $w \in (\Sigma \cup \{0,1,\#\})^*$ prüft TM' zunächst, ob w die Form $w = u\#bin(i)$ mit $u \in \Sigma^*$ und der 0-1-Folge $bin(i)$ hat. Falls nicht, schreibt TM' das Wort w_0 auf sein Ausgabeband und stoppt. Falls w diese Form hat, schreibt TM' den Teil u auf das 1. Band von TM und simuliert das Verhalten von TM für höchstens i viele Schritte. Falls TM das Wort u innerhalb dieser Schrittzahl akzeptiert, wird u auf das Ausgabeband geschrieben, und TM' stoppt. Falls TM das Wort u innerhalb dieser Schrittzahl nicht akzeptiert, schreibt TM' das Wort w_0 auf sein Ausgabeband und stoppt. Es sei h die von TM' berechnete Funktion. Dann gilt $L = h\left((\Sigma \cup \{0,1,\#\})^*\right)$:

Ist nämlich $u \in L$, dann akzeptiert TM das Wort in einer endlichen Anzahl i von Schritten. Mit $w = u\#bin(i)$ gilt $h(w) = u$. Das bedeutet $L \subseteq h\left((\Sigma \cup \{0,1,\#\})^*\right)$.

Ist umgekehrt $u \in h\left((\Sigma \cup \{0,1,\#\})^*\right)$, etwa $u = h(w)$ für ein Wort $w \in (\Sigma \cup \{0,1,\#\})^*$, dann ist entweder $u = w_0$ oder $w = u\#bin(i)$ und TM' hat höchstens i viele Schritte von TM simuliert und dabei $u \in L$ festgestellt. In beiden Fällen ist also $u \in L$, d.h. $h\left((\Sigma \cup \{0,1,\#\})^*\right) \subseteq L$.

Zum Beweis der anderen „Richtung“ der Aussage wird angenommen, daß $L = h\left((\Sigma \cup \{0,1,\#\})^*\right)$ mit einer totalen und berechenbaren Funktion h gilt, und es ist zu zeigen, daß es eine Turingmaschine TM gibt, die genau L akzeptiert:

Bei Eingabe von $w \in \Sigma^*$ verhält sich TM wie folgt. TM erzeugt alle Wörter $u \in (\Sigma \cup \{0,1,\#\})^*$ in lexikographischer Reihenfolge. Sobald ein Wort u erzeugt ist, berechnet TM den Wert $h(u)$. Gilt $h(u) = w$, so stoppt TM und akzeptiert w. Andernfalls wird das nächste Wort u erzeugt. Es läßt sich $L = L(TM)$ zeigen:

Ist $w \in L$, dann ist nach Voraussetzung $w = h(u)$ für ein Wort $u \in (\Sigma \cup \{0,1,\#\})^*$. Da $|u| < \infty$ ist, findet TM dieses Wort u (bei der Erzeugung aller Wörter in lexikographischer Reihenfolge. Da h total und berechenbar ist, kann TM den Wert $h(u)$ ermitteln und feststellen, daß $h(u) = w$ gilt. Daher wird w von TM akzeptiert, d.h. $L \subseteq L(TM)$.

Ist $w \notin L$, dann gilt für jedes $u \in (\Sigma \cup \{0,1,\#\})^*$: $h(u) \neq w$. Dann stoppt TM bei Eingabe von w nicht, d.h. $w \notin L(TM)$. Daher gilt $L(TM) \subseteq L$. ///

In Kapitel 1.1 wurde der Begriff der Abzählbarkeit definiert: Eine Menge M ist abzählbar unendlich, wenn es eine bijektive Abbildung $f : \mathbf{N} \to M$ gibt, d.h. $M = \{ f(i) \mid i \in \mathbf{N} \}$, und jedes Element $m \in M$ trägt eine eindeutige Nummer i: $m = f(i)$. Die Elemente von M können mit natürlichen Zahlen durchnumeriert bzw. indiziert werden. Ersetzt man in dieser Definition

den Begriff „bijektiv“ durch „total und berechenbar“ und **N** durch $(\Sigma \cup \{0,1,\#\})^*$, so kommt man auf den Begriff der rekursiven Aufzählbarkeit. Für eine rekursiv aufzählbare Menge L gilt $L = h\left((\Sigma \cup \{0,1,\#\})^*\right)$ mit einer totalen und berechenbaren Funktion $h : (\Sigma \cup \{0,1,\#\})^* \to \Sigma^*$. Die Elemente $u \in L$ können aus Elementen gewonnen werden, die als Zusatzinformation eine obere Schranke für die Schrittzahl enthalten, die eine Turingmaschine benötigt, um das jeweilige Wort zu akzeptieren.

Die Buchstaben des endlichen Alphabets $\Sigma = \{a_0, .., a_{|\Sigma|-1}\}$ einer Turingmaschine lassen sich als Zeichenkette über dem Alphabet {0, 1} kodieren: der Buchstabe a_i hat dabei die Kodierung $bin(a_i)$. Ein Wort $w = a_1 \ldots a_n$ kann man dann zunächst in $bin(a_1)\# \ldots \# bin(a_n)\#$ umsetzen und dann die darin vorkommenden einzelnen Zeichen aus {0, 1, #} in eine 0-1-Folge, wie es etwa in Kapitel 2.4 beschrieben wurde, umkodieren. Entsprechend kann man Paare (w, v) von Worten $w = a_1 \ldots a_n$ und $v = b_1 \ldots b_m$ zunächst in $(bin(a_1)\# \ldots \# bin(a_n)\#\# bin(b_1)\# \ldots \# bin(b_m)\#)$ umsetzen und die darin vorkommenden Zeichen aus { 0, 1, #, (,) } ähnlich wie in Kapitel 2.4 in eine 0-1-Folge umkodieren. Je nach Anwendung soll es daher erlaubt sein, daß eine Turingmaschine Eingaben der Form $w \in \Sigma^*$, aber auch Eingaben der Form $(w, v) \in \Sigma^* \times \Sigma^*$ verarbeitet. Letztlich lassen sich diese aus Paaren bestehenden Eingaben mit „normalen“ 0-1-Folgen identifizieren.

Der obige Satz kann dann auch so formuliert werden:

Satz 3.1-2:

$L \subseteq \Sigma^*$ mit $L \neq \varnothing$ ist genau dann rekursiv aufzählbar, wenn es eine totale berechenbare Funktion $h : \{0,1\}^* \to \Sigma^*$ gibt mit $L = h\left(\{0,1\}^*\right)$.

Die Funktion *h* „zählt *L* rekursiv auf“ (erzeugt *L*).

Folgender Satz ist unmittelbar einsichtig:

Satz 3.1-3:

Jede rekursiv aufzählbare Menge über einem Alphabet Σ ist endlich oder abzählbar unendlich ist.

Es fragt sich, ob umgekehrt jede abzählbare Teilmenge von Σ^* bereits rekursiv aufzählbar ist. Folgende Überlegungen zeigen, daß diese Frage verneint werden muß.

Zu jeder rekursiv aufzählbaren Teilmenge von Σ^* gibt es nach Definition eine Turingmaschine, die die Menge akzeptiert. Natürlich kann eine rekursiv aufzählbare Menge von verschiedenen Turingmaschinen akzeptiert werden. Andererseits ist jede von einer Turingmaschine akzeptierte Menge rekursiv aufzählbar. Daher gibt es höchstens so viele rekursiv aufzählbare Teilmengen von Σ^* wie Turingmaschinen mit dem Alphabet Σ. Die Menge der Turingmaschinen mit dem Alphabet Σ ist abzählbar; denn jede Turingmaschine *TM* läßt sich mit Hilfe der injektiven Funktion *code* auf ein endlich langes Wort *code*(*TM*) über $\{0,1\}$ abbilden, siehe Kapitel 2.4). Satz 1.1-4 sagt aus, daß es überabzählbar viele Teilmengen von Σ^* gibt. Daher existieren auch nicht-rekursiv aufzählbare Teilmengen von Σ^*.

Eine derartige nicht rekursiv aufzählbare Teilmenge von Σ^* kann auch konstruktiv durch eine Technik angegeben werden, die man als **Diagonalisierung** bezeichnet. Der Einfachheit halber soll hier $\Sigma = \{0,1\}$ angenommen werden. Die Elemente von $\Sigma^* = \{0,1\}^*$ werden in lexikographischer Reihenfolge aufgezählt:

Nummer *i*	Wort $w_i \in \{0,1\}^*$	Nummer *i*	Wort $w_i \in \{0,1\}^*$
0	*e*	11	100
1	0	12	101
2	1	13	110
3	00	14	111
4	01	15	0000
5	10	16	0001
6	11	17	0010
7	000	18	0011
8	001	19	0100
9	010	20	0101
10	011	...	...

Das *i*-te Wort w_i kann als Eingabe für eine Turingmaschine mit Eingabealphabet {0, 1} und auch, falls $VERIFIZIERE\_TM(w_i) = \mathsf{TRUE}$ gilt, als die von w_i kodierte Turingmaschine K_{w_i} interpretiert werden (siehe Kapitel 2.4). Es läßt sich daher eine Matrix *M* definieren, deren Zeilen und Spalten mit den Wörtern w_i markiert sind. Die Zeilenmarkierung w_i stellt ein derartiges Eingabewort für eine Turingmaschine dar, die Spaltenmarkierung w_j bezeichnet eventuell die Turingmaschine K_{w_j}. Im Schnittpunkt der *i*-ten Zeile von *M* mit der *j*-ten Spalte wird entweder der Wert 0 oder der Wert 1 eingetragen, und zwar so, daß dort

$$\begin{cases} 0 & \text{für } VERIFIZIERE\_TM(w_j) = \mathsf{FALSE} \text{ oder } w_i \notin L(K_{w_j}) \\ 1 & \text{für } VERIFIZIERE\_TM(w_j) = \mathsf{TRUE} \text{ und } w_i \in L(K_{w_j}) \end{cases}$$

steht. Da die ersten Spaltenmarkierungen w_0, w_1, w_2, ... nicht sinnvolle Turingmaschinen kodieren, enthält die Matrix *M* im linken Teil nur die Einträge 0.

Satz 3.1-4:

Die Menge Menge $L_d \subseteq \{0,1\}^*$ sei definiert durch:

Es ist $w \in L_d$ genau dann, wenn $w = w_i$ ist (das *i*-te Wort in der lexikographischen Reihenfolge) und *M* in der *i*-ten Zeile und *i*-ten Spalte den Eintrag 0 hat.

Dann gilt:
Die so definierte Menge L_d ist nicht rekursiv aufzählbar, d.h. es gibt keine Turingmaschine *TM* mit $L(TM) = L_d$.

Es ist also $L_d = \left\{ w \middle| \begin{array}{l} w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{FALSE} \\ \text{oder} \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE} \text{ und } w \notin L(K_w) \end{array} \right\}$.

Beweis:

Offensichtlich ist $L_d \neq \varnothing$.

Angenommen, L_d ist rekursiv aufzählbar. Dann gibt es eine Turingmaschine *TM* mit $L(TM) = L_d$. Die Kodierung dieser Turingmaschine sei $w = w_i$, d.h. $TM = K_{w_i}$ und $L_d = L(K_{w_i})$. Gilt nun $w_i \in L_d$? Falls $w_i \in L_d$ gilt, dann enthält nach Definition von L_d die Matrix *M* in der *i*-ten Zeile und *i*-ten Spalte den Eintrag 0. Nach Definition von *M* und wegen $VERIFIZIERE\_TM(w_i) = \mathsf{TRUE}$ bedeutet das aber: $w_i \notin L(K_{w_i})$, also $w_i \notin L_d$. Dieser Widerspruch erlaubt nur die Alternative $w_i \notin L_d$. Nach Definition von L_d enthält die Matrix *M* in der *i*-ten Zeile und *i*-ten Spalte den Eintrag 1. Das bedeutet nach Definition von *M*: $w_i \in L(K_{w_i})$ und damit $w_i \in L_d$. Dieser Widerspruch zeigt, daß die ursprüngliche Annahme, daß L_d rekursiv aufzählbar sei, falsch ist. ///

Mit L_d ist also ein erstes Beispiel einer abzählbaren, aber nicht rekursiv aufzählbaren Teilmenge von $\{0,1\}^*$ gefunden.

3.2 Rekursiv aufzählbare und entscheidbare Mengen

In Kapitel 2.1 wurden bereits einige Eigenschaften rekursiv aufzählbarer Mengen angegeben, die im folgenden Satz in Teil 1 noch einmal aufgeführt werden:

Satz 3.2-1:

1. Sind $L_1 \subseteq \Sigma^*$ und $L_2 \subseteq \Sigma^*$ rekursiv aufzählbar, dann sind auch $L_1 \cup L_2$ und $L_1 \cap L_2$ rekursiv aufzählbar. Die Klasse der rekursiv aufzählbaren Mengen über einem endlichen Alphabet Σ ist abgeschlossen gegenüber Vereinigung- und Schnittbildung.
2. Ist $L \subseteq \Sigma^*$ rekursiv aufzählbar, dann ist nicht notwendigerweise auch $\Sigma^* \setminus L$ rekursiv aufzählbar. Die Klasse der rekursiv aufzählbaren Mengen über einem endlichen Alphabet Σ ist nicht abgeschlossen gegenüber Komplementbildung.

Der Beweis für die Gültigkeit der Aussage im 2. Teil ergibt sich aus den folgenden Sätzen 3.2-5 und 3.2-7.

Die Menge $L \subseteq \Sigma^*$ werde von der *k*-DTM *TM* akzeptiert. Wird $w \in L$ auf das Eingabeband von *TM* gegeben, dann hält *TM* nach endlich vielen Schritten im Zustand q_{accept} an. Wird ein Wort *w* mit $w \in \Sigma^* \setminus L$ auf das Eingabeband gegeben, dann hält *TM* eventuell nicht an. Es wäre natürlich wünschenswert, daß *TM* auch in diesem Fall anhält, z.B. im Zustand q_{reject}, der ausdrückt, daß $w \notin L$ gilt. Gibt es also zu jeder von einer Turingmaschine *TM* akzeptierten Sprache $L \subseteq \Sigma^*$ eine Turingmaschine TM' mit $L(TM') = L$ und der Eigenschaft, daß TM' bei jedem Wort $w \in \Sigma^*$ anhält: im Zustand q_{accept}, falls $w \in L$ ist, und im Zustand q_{reject}, falls $w \notin L$ ist (nichtstoppende Berechnungen kommen nicht vor)? Diese Frage führt auf die folgende Definition:

Die Menge $L \subseteq \Sigma^*$ heißt **entscheidbar (**oder **rekursiv)**, wenn es eine Turingmaschine *TM* gibt mit der Eigenschaft:
TM stoppt bei jeder Eingabe $w \in \Sigma^*$, und *TM* akzeptiert *w* genau dann, wenn $w \in L$ ist.

In praktischen Anwendungen sind die entscheidbaren Mengen gerade diejenigen Sprachen, mit denen man „umgehen“ kann. Denn man möchte ja bei Eingabe von $w \in \Sigma^*$ in einen Entscheidungsalgorithmus nach endlicher Zeit die Frage geklärt haben, ob *w* eine spezifizierte Eigenschaft hat, d.h. ob $w \in L$ ist , oder nicht. Ist *L* entscheidbar, läßt sich diese Entscheidung definitiv herbeiführen. Bei einer rekursiv aufzählbaren Menge *L* und einer passenden Turing-

maschine (Entscheidungsverfahren), der man eine Eingabe $w \in \Sigma^*$ vorgelegt hat und die bereits einige Rechenschritte vollzogen hat, ohne bisher eine Entscheidung zu treffen, kann man nicht sicher sein, ob überhaupt jemals eine Entscheidung getroffen wird (vgl. dazu Satz 3.1-4).

Satz 3.2-2:

1. Jede entscheidbare Menge ist rekursiv aufzählbar.
   Die Klasse der entscheidbaren Mengen über einem endlichen Alphabet Σ ist in der Klasse der rekursiv aufzählbaren Mengen über diesem Alphabet enthalten.

2. Jede endliche Menge ist entscheidbar.

3. Ist $L \subseteq \Sigma^*$ entscheidbar, dann ist auch $\Sigma^* \setminus L$ entscheidbar.
   Die Klasse der entscheidbaren Mengen über einem endlichen Alphabet Σ ist abgeschlossen gegenüber Komplementbildung.

4. Sind $L_1 \subseteq \Sigma^*$ und $L_2 \subseteq \Sigma^*$ entscheidbar, dann sind auch $L_1 \cup L_2$ und $L_1 \cap L_2$ entscheidbar.
   Die Klasse der entscheidbaren Mengen über einem endlichen Alphabet Σ ist abgeschlossen gegenüber Vereinigung- und Schnittbildung.

Beweis:

Zu 1. Die Aussage ergibt sich direkt aus der Definition.

Zu 2. Ist $L = \{w_1, ..., w_n\}$ eine endliche Menge, $L \subseteq \Sigma^*$, so ist eine auf allen Eingaben $w \in \Sigma^*$ stoppen Turingmaschine *TM* anzugeben, die genau dann im akzeptierenden Zustand hält, wenn *w* mit einem der endlich vielen Werte in *L* übereinstimmt. Es ist also eine Turingmaschine zu definieren, die das (Pseudocode-) Statement

```
CASE w OF
w_1 : accept;
...
w_n : accept;
ELSE reject;
END;
```

realisert.

Zu 3. Es sei *TM* eine auf allen Eingaben $w \in \Sigma^*$ stoppende Turingmaschine mit $L = L(TM)$. Man kann annehmen, daß *TM* den akzeptierenden Zustand q_{accept} hat, der erreicht wird, wenn $w \in L$ gilt, und den verwerfenden Zustand q_{reject} hat, wenn $w \notin L$ ist. Eine

auf allen Eingaben $w \in \Sigma^*$ stoppende Turingmaschine TM' mit $L(TM') = \Sigma^* \setminus L$ erhält man aus *TM*, indem man die Rollen von q_{accept} und q_{reject} vertauscht.

Zu 4. Diese Aussage wird genauso gezeigt wie bei rekursiv aufzählbaren Mengen. ///

Die folgende Definition stellt ein Hilsmittel bereit, das es erlaubt, auf einfache Weise Entscheidbarkeitsergebnisse zwischen Mengen zu übertragen, die eventuell mit unterschiedlichen Alphabeten formuliert sind und damit unterschiedlichen Anwendungsgebieten entnommen sind.

Es seien Σ und Σ' endliche Alphabete, $L \subseteq \Sigma^*$ und $L' \subseteq \Sigma'^*$. Die Sprache L heißt auf die Sprache L' **reduzierbar**, geschrieben $L \leq L'$, wenn gilt:

Es gibt eine Funktion $h : \Sigma^* \to \Sigma'^*$, die folgende beiden Eigenschaften besitzt:

(i) h ist total und von einer Turingmaschine berechenbar
(ii) es gilt $w \in L \Leftrightarrow h(w) \in L'$.

Eigenschaft (ii) kann auch so formuliert werden:
$w \in L \Rightarrow h(w) \in L'$ und $w \notin L \Rightarrow h(w) \notin L'$.

Die Bedeutung der Reduzierbarkeit zeigt folgender Satz:

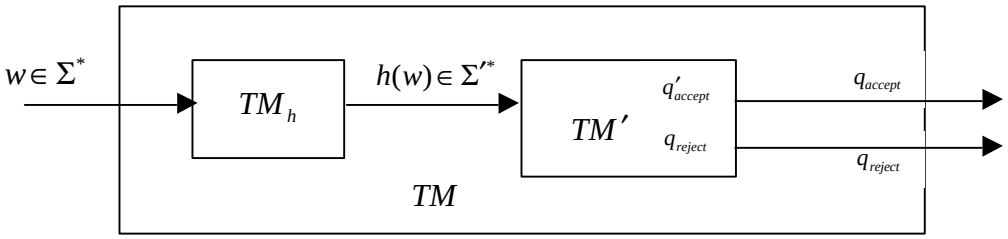
Satz 3.2-3:

Es seien Σ und Σ' endliche Alphabete, $L \subseteq \Sigma^*$ und $L' \subseteq \Sigma'^*$ und $L \leq L'$. Dann gilt:

1. Ist L' entscheidbar, dann ist auch L entscheidbar.
2. Ist L nicht entscheidbar, dann ist auch L' nicht entscheidbar.
3. Ist L' rekursiv aufzählbar, dann ist auch L rekursiv aufzählbar.
4. Ist L nicht rekursiv aufzählbar, dann ist auch L' nicht rekursiv aufzählbar.

Beweis:

Zu 1. Es sei TM' eine auf allen Eingaben $u \in \Sigma'^*$ stoppende Turingmaschine mit $L(TM') = L'$. Die Funktion $h: \Sigma^* \to \Sigma'^*$, die in der Definition der Relation $L \leq L'$ vorkommt, werde durch die Turingmaschine TM_h berechnet. Durch Hintereinanderschalten von TM_h und TM' erhält man eine Turingmaschine *TM*:



TM stoppt auf allen Eingaben $w \in \Sigma^*$, weil *h* total ist und TM' auf allen Eingaben stoppt).
Ist $w \in \Sigma^*$ eine Eingabe für *TM*, die zum Zustand q_{accept} führt, d.h. $w \in L(TM)$, dann hat die Eingabe $h(w)$ in TM' zum Zustand q'_{accept} geführt. Das bedeutet $h(w) \in L'$ und wegen Eigenschaft (ii) in der Definition der Relation $L \leq L'$: $w \in L$. Damit ist $L(TM) \subseteq L$ gezeigt.
Ist umgekehrt $w \in L$, dann gilt wegen $L \leq L'$: $h(w) \in L'$. Die Eingabe $h(w)$ in TM' führt zum Zustand q'_{accept}, d.h. *TM* kommt bei Eingabe von *w* in den Zustand q_{accept}. Das bedeutet $w \in L(TM)$ und damit $L \subseteq L(TM)$.

Zu 2. Hier ist nichts zu beweisen, da es sich um eine logisch äquivalente Formulierung zu Teil 1 handelt.

Zu 3. Die Argumentation verläuft wie in Teil 1, wobei im vorherigen Bild nur der mit q_{accept} markierte Ausgang betrachtet wird.

Zu 4. Es handelt sich um eine logisch äquivalente Formulierung zu Teil 3. ///

Satz 3.2-4:

Die Klasse der entscheidbaren Mengen ist eine echte Untermenge der Klasse der rekursiv aufzählbaren Mengen.

Dazu ist mindestens ein Beispiel einer rekursiv aufzählbaren Menge, die nicht entscheidbar sind, anzugeben. Dieses kann konstruktiv geschehen; das Ergebnis ist in folgendem Satz 3.2-5 formuliert:

Zur Erinnerung (siehe Kapitel 2.4): Für ein Wort $w \in \{0,1\}^*$ ist *VERIFIZIERE_TM*(*w*) = TRUE genau dann, wenn *w* die Kodierung einer Turingmaschine ist. Die durch *w* kodierte Turingmaschine wird mit K_w bezeichnet.

Die zum **Halteproblem gehörende Sprache** $L_H \subseteq \{0,1,\#\}^*$ wird definiert als die Menge

$$L_H = \left\{ z \;\middle|\; \begin{array}{l} z = u\#w \text{ mit } u \in \{0,1\}^*, w \in \{0,1\}^*, \mathit{VERIFIZIERE\_TM}(w) = \mathsf{TRUE} \text{ und} \\ K_w \text{ stoppt bei Eingabe von } u \end{array} \right\}.$$

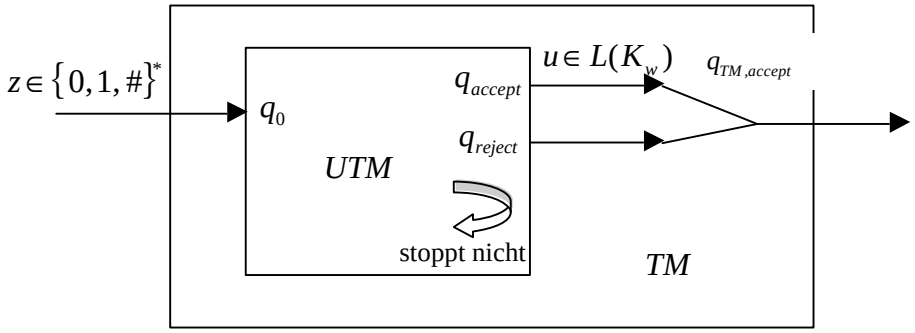
Es gilt:

Satz 3.2-5:
L_H ist rekursiv aufzählbar, aber nicht entscheidbar.

Insbesondere heißt das: es gibt keine immer anhaltende Turingmaschine *TM* mit akzeptierendem Zustand q_{accept} und nicht-akzeptierendem Zustand q_{reject} mit folgender Eigenschaft: Bei Eingabe von *u#w* stoppt *TM* im akzeptierenden Zustand q_{accept}, falls K_w auf *u* stoppt, und *TM* stoppt im nicht-akzeptierendem Zustand q_{reject}, falls K_w auf *u* nicht stoppt.

Beweis:

Um die rekursive Aufzählbarkeit von L_H zu zeigen, ist eine Turingmaschine *TM* mit $L(TM) = L_H$ anzugeben. *TM* ist eine einfache Modifikation der in Kapitel 2.4 beschriebenen universellen Turingmaschine *UTM*: Man kann annehmen, daß *UTM* nur ein Band hat und einen akzeptierenden Zustand q_{accept} und einen nicht-akzeptierenden Zustand q_{reject} besitzt. Kommt *UTM* in einen dieser beiden Zustände, stoppt *UTM*. Eventuell stoppt *UTM* jedoch bei einer Eingabe nicht. Es wird ein neuer akzeptierender Zustand $q_{TM,accept}$ in *TM* eingeführt und die Überführungsfunktion d_{UTM} von *UTM* so geändert, daß noch eine Überführung in den Zustand $q_{TM,accept}$ ausgeführt wird, wenn *UTM* in q_{accept} oder q_{reject} kommt. Dazu wird d_{UTM} um die Zeilen $d_{UTM}(q_{accept}, a) = (q_{TM,accept}, (a,S))$ und $d_{UTM}(q_{reject}, a) = (q_{TM,accept}, (a,S))$ für jedes Symbol *a* aus dem Arbeitsalphabet von *UTM* erweitert.



Es sei $z \in L_H$, d.h. $z = u\#w$ mit $u \in \{0,1\}^*$, $w \in \{0,1\}^*$, $VERIFIZIERE\_TM(w) = \mathsf{TRUE}$ und K_w stoppt bei Eingabe von *u*. Daher kommt *UTM* bei Eingabe von *z* in den Zustand q_{accept} oder in den Zustand q_{reject}. *TM* kommt in beiden Fällen in den Zustand $q_{TM,accept}$, so daß $z \in L(TM)$ gilt.

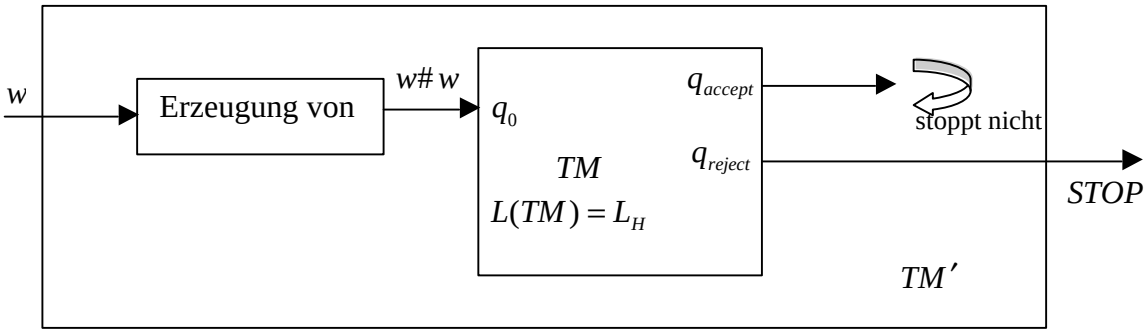
Ist $z \notin L_H$, dann hat *z* entweder nicht die syntaktische Form $z = u\#w$ mit $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$ oder $VERIFIZIERE\_TM(w) = \mathsf{FALSE}$ oder K_w stoppt bei Eingabe von *u* nicht. In den ersten beiden Fällen stoppt *UTM* bei Eingabe von *z* nicht (vgl. Beweis von Satz 2.4-1). Im dritten Fall startet *UTM* wohl die Simulation von K_w, die jedoch nicht zu einem stoppenden Zustand führt. Daher ist in diesem Fall $z \notin L(TM)$.

Insgesamt ist damit $L(TM) = L_H$ gezeigt.

Zum Beweis der Nichtentscheidbarkeit von L_H wird wieder die Methode der Diagonalisierung eingesetzt:

Angenommen, es gibt eine auf allen Eingaben $z \in \{0,1,\#\}^*$ stoppende Turingmaschine *TM* mit $L(TM) = L_H$. Dann wird *TM* leicht abgewandelt zu einer Turingmaschine TM', die wie folgt arbeitet: TM' liest eine Eingabe $w \in \{0,1\}^*$ und erzeugt mit einer Kopie von *w* das Wort *w#w*. Dann verhält sich TM' wie *TM* bei Eingabe von *w#w*. Falls *TM* im akzeptierenden Zustand q_{accept} stoppt, geht TM' in einen neuen Zustand q' über und stoppt nicht[4]. Falls *TM* im nicht-akzeptierenden Zustand q_{reject} stoppt, dann stoppt TM'; dabei ist es irrelevant, ob TM' in einem akzeptierenden oder einem nicht-akzeptierenden Zustand stoppt.

[4] Ist TM' eine *k*-DTM mit der Überführungsfunktion d', dann ist $d'(q', a_1, ..., a_k) = (q', (a_1, S), ..., (a_k, S))$ für $a_1 \in \Sigma, ..., a_k \in \Sigma$.



Falls *TM* existiert, dann auch TM', und zwar mit einer Kodierung $u \in \{0,1\}^*$, d.h. $TM' = K_u$. Nun gibt es zwei Alternativen: TM' stoppt auf der eigenen Kodierung *u*, oder TM' stoppt auf der eigenen Kodierung *u* nicht. Im ersten Fall ist $u\#u \notin L_H$; nach Definition von L_H stoppt $K_u = TM'$ bei Eingabe von *u* nicht. Dieser Widerspruch läßt nur die zweite Alternative zu; das bedeutet aber $u\#u \in L_H$, also stoppt $K_u = TM'$ bei Eingabe von *u*. Beide Alternativen führen auf einen Widerspruch. Daher existiert *TM* nicht. ///

Die zum **Halteproblem mit leerem Eingabeband gehörende Sprache** $L_{H_0} \subseteq \{0,1\}^*$ wird definiert als die Menge

$L_{H_0} = \{w \mid w \in \{0,1\}^*, VERIFIZIERE\_TM(w) = \text{TRUE und } K_w \text{ stoppt bei leerem Eingabeband}\}$.

Satz 3.2-6:

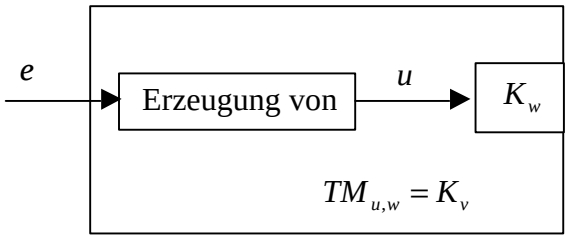
L_{H_0} ist rekursiv aufzählbar, aber nicht entscheidbar.

Beweis:

Die rekursive Aufzählbarkeit läßt sich ähnlich wie im Beweis zu Satz 3.2-5 nachweisen.

Um die Nichtentscheidbarkeit von L_{H_0} zu beweisen, zeigt man, daß L_H nach L_{H_0} reduzierbar ist, d.h. daß $L_H \leq L_{H_0}$ gilt. Dazu ist eine totale und berechenbare Funktion $h: \{0,1,\#\}^* \to \{0,1\}^*$ anzugeben, für die $z \in L_H$ genau dann gilt, wenn $h(z) \in L_{H_0}$ ist. Aus Satz 3.2-3, Teil 2 folgt dann die Aussage des Satzes.

Zu $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$ mit *VERIFIZIERE_TM*(*w*) = TRUE sei $TM_{u,w}$ eine Turingmaschine, die folgendes Verhalten aufweist: $TM_{u,w}$ startet mit leerem Eingabeband und erzeugt das Wort $u \in \{0,1\}^*$. Dann simuliert $TM_{u,w}$ das Verhalten von K_w. Die Kodierung von $TM_{u,w}$ sei *v*. Es gilt $v \in \{0,1\}^*$ und *VERIFIZIERE_TM*(*v*) = TRUE.



Mit Hilfe eines entsprechenden Algorithmus, eines „Turingmaschinengenerators“, kann man bei Vorgabe von *u* und *w* die Kodierung *v* von $TM_{u,w}$ auf deterministische Weise erzeugen. Die Funktion *h* wird definiert durch

$$h: \begin{cases} \{0,1,\#\}^* & \to & \{0,1\}^* \\ & & \\ z & \to & \begin{cases} code(TM_{u,w}) & \text{falls } z = u\#w \text{ mit } u \in \{0,1\}^* \text{ und } w \in \{0,1\}^* \\ & \text{und } VERIFIZIERE\_TM(w) = \text{TRUE} \\ 1 & \text{sonst} \end{cases} \end{cases}$$

Diese ist total und berechenbar. Nach Definition von L_H gilt:

$u\#w \in L_H \Leftrightarrow K_w$ stoppt bei Eingabe von *u*

$\Leftrightarrow TM_{u,w} = K_v$ stoppt bei leerem Eingabeband

$\Leftrightarrow h(u\#w) = v$ und $v \in L_{H_0}$. ///

Eine Charakterisierung der Entscheidbarkeit einer Menge liefert der folgende Satz:

Satz 3.2-7:

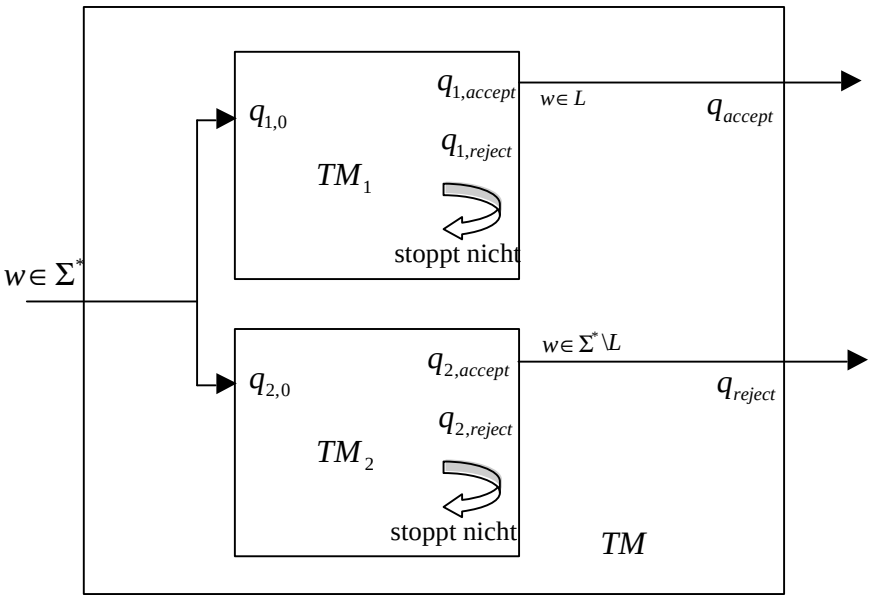
Eine Menge $L \subseteq \Sigma^*$ ist genau dann entscheidbar, wenn sowohl *L* als auch das Komplement $\Sigma^* \setminus L$ von *L* rekursiv aufzählbar sind.

Beweis:

Auch dieser Satz beinhaltet wieder zwei „Richtungen“:

Ist die Menge $L \subseteq \Sigma^*$ entscheidbar, dann ist sie rekursiv aufzählbar. Satz 3.2-2 Teil 3 besagt, daß mit *L* auch $\Sigma^* \setminus L$ entscheidbar und damit rekursiv aufzählbar ist.

Gilt umgekehrt für eine Menge $L \subseteq \Sigma^*$, daß mit ihr auch das Komplement $\Sigma^* \backslash L$ rekursiv aufzählbar ist, dann gibt es zwei Turingmaschinen TM_1 und TM_2 mit $L = L(TM_1)$ und $\Sigma^* \backslash L = L(TM_2)$. Aus diesen beiden Turingmaschinen wird durch Parallelschaltung eine auf allen Eingaben $w \in \Sigma^*$ stoppende Turingmaschine *TM* konstruiert, die *w* genau dann akzeptiert, wenn $w \in L$ gilt. Dabei wird eine ähnliche Konstruktion gewählt, wie sie zum Nachweis der Abgeschlossenheit der rekusiv aufzählbaren Mengen bezüglich Vereinigungsbildung angegeben wurde (Kapitel 2.1):



Es gilt für $w \in \Sigma^*$: Ist $w \in L$, dann stoppt TM_1 im Zustand $q_{1,accept}$ und damit stoppt *TM* im Zustand q_{accept}. Ist $w \notin L$, dann stoppt TM_2 im Zustand $q_{2,accept}$ und damit stoppt *TM* im Zustand q_{reject}. Da entweder $w \in L$ oder $w \notin L$ gilt, stoppt *TM* bei allen Eingaben $w \in \Sigma^*$ und akzeptiert genau die Menge *L*. ///

Die oben aufgeführte zum Halteproblem gehörige Menge L_H ist rekursiv aufzählbar, aber nicht entscheidbar. Das Komplement von L_H, die Menge $\{0,1,\#\}^* \backslash L_H$, kann daher nicht rekursiv aufzählbar sein. Daher ist die Klasse der rekursiv aufzählbaren Mengen gegenüber Komplementbildung nicht abgeschlossen (das ist Satz 3.2-1 Teil 2).

Da die Rollen von *L* und $\Sigma^* \backslash L$ im letzten Satz „symmetrisch“ sind, ergibt sich als Konsequenz:

Satz 3.2-8:

Es sei $L \subseteq \Sigma^*$. Dann gilt

Entweder

1. sowohl L als auch $\Sigma^* \setminus L$ ist entscheidbar

oder

2. weder L als noch $\Sigma^* \setminus L$ ist entscheidbar

oder

3. entweder L oder $\Sigma^* \setminus L$ ist rekursiv aufzählbar, aber nicht entscheidbar, und die jeweils andere Menge ist nicht rekursiv aufzählbar.

Die Klasse der rekursiv aufzählbaren Mengen über einem Alphabet Σ ist eine echte Teilklasse von $\mathbf{P}(\Sigma^*)$. Die Klasse der entscheidbaren Mengen über Σ ist echt enthalten in der Klasse der rekursiv aufzählbaren Mengen. Es stellt sich die Frage, ob man mit Hilfe eines durch eine Turingmaschine definierten algorithmischen Verfahrens die Klasse der entscheidbaren bzw. die Klasse der nicht-entscheidbaren Mengen erkennen kann oder wenigstens rekursiv aufzählen kann. Genauer: Es wird danach gefragt, ob es eine Turingmaschine gibt, die die Kodierungen aller derjenigen Turingmaschinen erzeugt (rekursiv aufzählt), deren akzeptierte Sprachen gerade die entscheidbaren bzw. nichtentscheidbaren Mengen sind. Leider ist das nicht möglich, wie folgender Satz zeigt:

Satz 3.2-9:

Es seien

$L_e = \left\{ w \middle| w \in \{0,1\}^*, \mathit{VERIFIZIERE\_TM}(w) = \mathsf{TRUE} \text{ und } L(K_w) \text{ ist entscheidbar} \right\}$ und

$L_{ne} = \left\{ w \middle| w \in \{0,1\}^*, \mathit{VERIFIZIERE\_TM}(w) = \mathsf{TRUE} \text{ und } L(K_w) \text{ ist nicht entscheidbar} \right\}$.

Dann ist weder L_e noch L_{ne} rekursiv aufzählbar (und damit auch nicht entscheidbar).

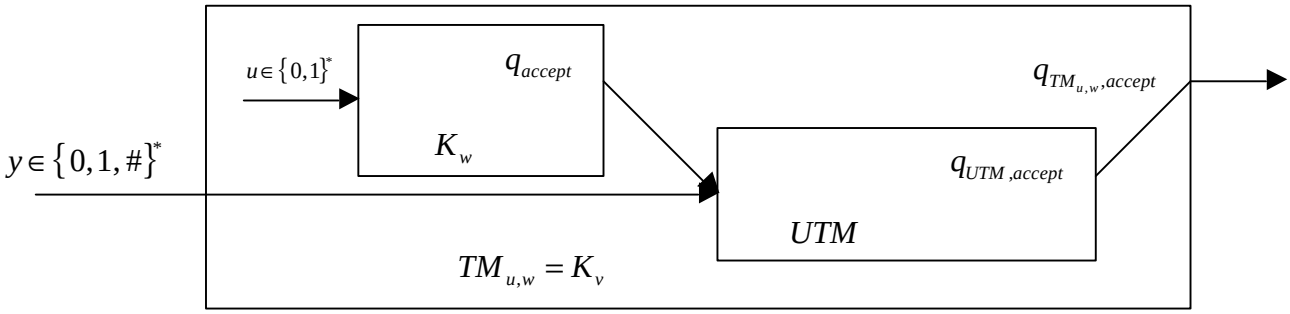
Beweis:

Es wird die von der universellen Turingmaschine *UTM* akzeptierte Sprache $L_{uni} = L(UTM) = \left\{ z \middle| \begin{array}{l} z = u\#w \text{ mit } u \in \{0,1\}^*, w \in \{0,1\}^*, \\ \mathit{VERIFIZIERE\_TM}(w) = \mathsf{TRUE} \text{ und } u \in L(K_w) \end{array} \right\} \subseteq \{0,1,\#\}^*$ herangezogen. L_{uni} ist rekursiv aufzählbar, aber nicht entscheidbar (siehe Übungsaufgaben). Daher ist das Komplement $\overline{L}_{uni} = \{0,1,\#\}^* \setminus L_{uni}$ nicht rekursiv aufzählbar. Es läßt sich zeigen, daß $\overline{L}_{uni} \leq L_e$ gilt; das Aussehen der dabei beteiligten totalen und berechenbaren Funktion

$h:\{0,1,\#\}^* \to \{0,1\}^*$ wird im folgenden skizziert. Aus Satz 3.2-3 Teil 4 folgt, daß L_e nicht rekursiv aufzählbar ist. Der Beweis für L_{ne} verläuft analog (siehe Übungsaufgaben).

Ähnlich wie oben wird zu $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$ mit *VERIFIZIERE_TM*(*w*) = TRUE mit Hilfe eines deterministisch arbeitenden Turingmaschinengenerators eine Turingmaschine $TM_{u,w}$ konstruiert, die wie folgt arbeitet:

Bei Eingabe von $y \in \{0,1,\#\}^*$ wird die Eingabe zunächst nicht berücksichtigt, sondern $TM_{u,w}$ verhält sich wie K_w auf der Eingabe *u*. Falls $u \in L(K_w)$ festgestellt wird, simuliert $TM_{u,w}$ das Verhalten von *UTM* auf der Eingabe *y*. Die Kodierung von $TM_{u,w}$ sei *v*. Es gilt $v \in \{0,1\}^*$ und *VERIFIZIERE_TM*(*v*) = TRUE. Das Wort *v* ist deterministisch berechenbar, da nur Beschreibungen von Turingmaschinen zusammengestellt wurden.



Es gilt $L(K_v) = L(TM_{u,w}) = \begin{cases} L(UTM) = L_{uni} & \text{falls } u \in L(K_w) \text{ ist} \\ \varnothing & \text{falls } u \notin L(K_w) \text{ ist} \end{cases}$

Bemerkung: Offensichtlich ist $L(K_v)$ genau dann entscheidbar, wenn $u \notin L(K_w)$ ist; denn dann ist $L(K_v) = \varnothing$. Für $u \in L(K_w)$ ist $L(K_v) = L_{uni}$, und diese Menge ist nicht entscheidbar.

Es sei $w_0 \in \{0,1\}^*$ die Kodierung einer Turingmaschine mit $L(K_{w_0}) = \{0,1\}^*$. Es ist $w_0 \in L_e$. Die gesuchte Funktion *h* wird definiert durch

$$h: \begin{cases} \{0,1,\#\}^* & \to & \{0,1\}^* \\ z & \to & \begin{cases} code(TM_{u,w}) & \text{falls } z = u\#w \text{ mit } u \in \{0,1\}^* \text{ und } w \in \{0,1\}^* \\ & \text{und } VERIFIZIERE\_TM(w) = \text{TRUE} \\ w_0 & \text{sonst} \end{cases} \end{cases}$$

Wie obige Ausführungen zeigen, ist *h* ist total und berechenbar. Außerdem gilt

$z \in \overline{L}_{uni} \Leftrightarrow$ *z* hat nicht die Form *z* = *u*#*w* mit $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$

oder z hat die Form $z = u\#w$ mit $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$ und
VERIFIZIERE_TM(w) = FALSE
oder z hat die Form $z = u\#w$ mit $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$ und
VERIFIZIERE_TM(w) = TRUE und $u \notin L(K_w)$.

Es sei $z \in \overline{L}_{uni}$. In den ersten beiden Fällen ist $h(z) = w_0$, also $h(z) \in L_e$. Im dritten Fall ist $h(z) = code(TM_{u,w}) = v$ und nach obiger Bemerkung und nach Definition von L_e: $h(z) \in L_e$.

Ist $z \notin \overline{L}_{uni}$, dann ist $z \in L_{uni}$, d.h. $z = u\#w$ mit $u \in \{0,1\}^*$, $w \in \{0,1\}^*$, *VERIFIZIERE_TM*(w) = TRUE und $u \in L(K_w)$. Daher ist $h(z) = code(TM_{u,w}) = v$ und nach obiger Bemerkung $v \notin L_e$.

Insgesamt gilt also $z \in \overline{L}_{uni} \Leftrightarrow h(z) \in L_e$, und damit ist $\overline{L}_{uni} \leq L_e$ gezeigt. ///

Im folgenden wird nach einem allgemeinen **algorithmischen Verfahren** gefragt, das einer Turingmaschine irgendeine nichttriviale Eigenschaft ansieht. Unter einem algorithmischen Verfahren ist hierbei eine auf allen Eingaben (entweder akzeptierend oder nicht akzeptierend) stoppende Turingmaschine zu verstehen. Der Begriff „nichttriviale Eigenschaft“ wird wie folgt definiert:

Eine Menge $L \subseteq \{0,1\}^*$ von Kodierungen von Turingmaschinen heißt **nichttriviales Entscheidungsproblem über Turingmaschinen**, wenn gilt:

(i) $L \neq \emptyset$

(ii) L enthält nicht die Kodierungen aller Turingmaschinen

(iii) für zwei Turingmaschinen TM_1 und TM_2, die durch w_1 bzw. w_2 kodiert werden, d.h. $TM_1 = K_{w_1}$ und $TM_2 = K_{w_2}$, impliziert $L(TM_1) = L(TM_2)$: Es gilt $w_1 \in L$ genau dann, wenn $w_2 \in L$ gilt.

Die **Frage nach der Entscheidbarkeit eines nichttrivialen Entscheidungsproblems** L **über Turingmaschinen** lautet dann in unterschiedlichen äquivalenten Formulierungen:

- Ist L entscheidbar?
- Gibt es ein algorithmisches Verfahren, das bei Eingabe eines Wortes $w \in \{0,1\}^*$ nach endlich vielen Schritten entscheidet, ob $w \in L$ gilt oder nicht?
- Gibt es ein algorithmisches Verfahren, das bei Eingabe der Beschreibung einer Turingmaschine nach endlich vielen Schritten entscheidet, ob die Kodierung der Turingmaschine zu L gehört oder nicht?
- Die Turingmaschinen, deren Kodierungen zu L gehören, haben alle eine durch L beschriebene Eigenschaft E_L. **Gibt es ein algorithmisches Verfahren, das bei Eingabe einer Turingmaschine nach endlich vielen Schritten entscheidet, ob die Turingmaschine die Eigenschaft E_L aufweist oder nicht?**

- Kann man mit Hilfe eines algorithmischen Verfahrens entscheiden, ob eine Turingmaschine eine definierte Eigenschaft E_L aufweist oder nicht?

Beispiele nichttrivialer Entscheidungsprobleme:

1. $L = \{ w \mid$ die durch w kodierte Turingmaschine K_w berechnet eine konstante Funktion $\}$; das Entscheidungsproblem lautet: Ist es entscheidbar, ob eine Turingmaschine eine konstante Funktion berechnet?
2. Es sei g eine Turingberechenbare Funktion.
   $L = \{ w \mid$ die durch w kodierte Turingmaschine K_w berechnet eine mit g identische Funktion $\}$; das Entscheidungsproblem lautet: Ist es entscheidbar, ob die von einer Turingmaschine berechnete Funktion mit g übereinstimmt?
3. $L_{=\varnothing} = \{ w \mid$ für die durch w kodierte Turingmaschine K_w gilt $L(K_w) = \varnothing \}$; das Entscheidungsproblem lautet: Ist es entscheidbar, ob die von einer Turingmaschine akzeptierte Menge leer ist?
4. $L_{\neq\varnothing} = \{ w \mid$ für die durch w kodierte Turingmaschine K_w gilt $L(K_w) \neq \varnothing \}$; das Entscheidungsproblem lautet: Ist es entscheidbar, ob die von einer Turingmaschine akzeptierte Menge mindestens ein Wort enthält?
5. $L = \{ w \mid$ für die durch w kodierte Turingmaschine K_w gilt : $L(K_w)$ ist endlich $\}$; das Entscheidungsproblem lautet: Ist es entscheidbar, ob die von einer Turingmaschine akzeptierte Menge endlich ist?
6. Es sei $L_0 \subseteq \{0,1\}^*$ eine entscheidbare Menge und
   $L = \{ w \mid$ für die durch w kodierte Turingmaschine K_w gilt : $L(K_w) = L_0 \}$; das Entscheidungsproblem lautet: Ist es entscheidbar, ob eine Turingmaschine die Menge L_0 akzeptiert?

Satz 3.2-10:

Jedes nichttriviale Entscheidungsproblem $L \subseteq \{0,1\}^*$ über Turingmaschinen ist nicht entscheidbar.

Beweis:

Es wird gezeigt, daß entweder $L_{H_0} \leq L$ oder $L_{H_0} \leq \{0,1\}^* \setminus L$ gilt. Da L_{H_0} nicht entscheidbar ist, ist im ersten Fall L nicht entscheidbar (Satz 3.2-3 Teil 2), im zweiten Fall ist $\{0,1\}^* \setminus L$ nicht entscheidbar (Satz 3.2-3 Teil 2) und damit auch L nicht (Satz 3.2-2 Teil 3).

Der Beweis folgt dem Schema der Beweise für die Sätze 3.2-6 und 3.2-9:

Es sei TM_0 eine Turingmaschine mit $L(TM_0) = \varnothing$ und $w_0 = code(TM_0)$, d.h. $TM_0 = K_{w_0}$. Es werden die beiden Fälle

1. Fall: $w_0 \in L$ und
2. Fall: $w_0 \notin L$

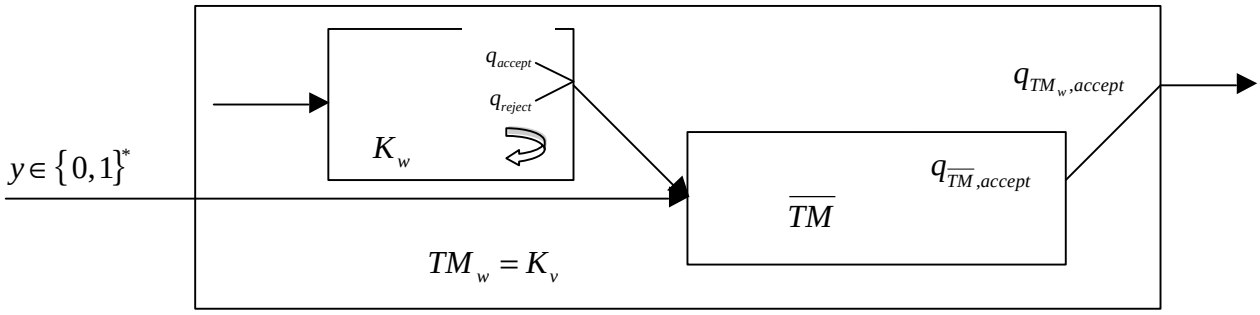
unterschieden.

Zum 1. Fall ($w_0 \in L$):

Nach obiger Bedingung (ii) für L gibt es eine Turingmaschine $\overline{TM}$ mit Kodierung $\overline{w} \in \{0,1\}^*$ und $\overline{w} \notin L$. Es wird die Gültigkeit von $L_{H_0} \leq \{0,1\}^* \setminus L$ gezeigt, indem eine totale und berechenbare Funktion $h : \{0,1\}^* \to \{0,1\}^*$ angegeben wird, für die $w \in L_{H_0} \Leftrightarrow h(w) \in \{0,1\}^* \setminus L$ gilt.

Es sei $w \in \{0,1\}^*$ mit *VERIFIZIERE_TM*(*w*) = TRUE. Mit Hilfe eines deterministisch arbeitenden Turingmaschinengenerators wird aus *w* eine Turingmaschine TM_w konstruiert, die wie folgt arbeitet:

Bei Eingabe von $y \in \{0,1\}^*$ wird die Eingabe zunächst nicht berücksichtigt, sondern TM_w simuliert das Verhalten von K_w bei leerem Eingabeband. Falls K_w stoppt, wird *y* in $\overline{TM}$ eingegeben, und TM_w simuliert das Verhalten von $\overline{TM}$ auf der Eingabe *y*. Die Kodierung von TM_w sei *v*. Es gilt $v \in \{0,1\}^*$ und *VERIFIZIERE_TM*(*v*) = TRUE. Das Wort *v* ist deterministisch berechenbar, da nur Beschreibungen von Turingmaschinen zusammengestellt wurden.



Man sieht:

$$L(K_v) = L(TM_w) = \begin{cases} L(\overline{TM}) & \text{falls } w \in L_{H_0} \text{ ist} \\ \varnothing & \text{sonst} \end{cases}.$$

Die Funktion *h* wird definiert durch:

$$h : \begin{cases} \{0,1\}^* & \to & \{0,1\}^* \\ w & \to & \begin{cases} code(TM_w) & \text{falls } VERIFIZIERE\_TM(w) = \text{TRUE} \\ w_0 & \text{sonst} \end{cases} \end{cases}$$

Die Funktion *h* ist total und berechenbar. Außerdem ist diese Funktion für den Nachweis der Relation $L_{H_0} \leq \{0,1\}^* \backslash L$ geeignet:

Ist $w \in L_{H_0}$, dann ist (nach Definition von L_{H_0}) *VERIFIZIERE_TM*(*w*) = TRUE und $h(w) = code(TM_w) = v$. Außerdem gilt (siehe oben) $L(K_v) = L(\overline{TM}) = L(K_{\overline{w}})$. Nach Bedingung (iii) ist wegen $\overline{w} \notin L$ auch $v \notin L$, d.h. $h(w) \notin L$ bzw. $h(w) \in \{0,1\}^* \backslash L$.

Ist $w \notin L_{H_0}$ und ist *VERIFIZIERE_TM*(*w*) = FALSE, dann ist $h(w) = w_0$ und $h(w) \in L$ bzw. $h(w) \notin \{0,1\}^* \backslash L$.

Ist $w \notin L_{H_0}$ und ist *VERIFIZIERE_TM*(*w*) = TRUE, dann ist $h(w) = code(TM_w) = v$ und $L(K_v) = \varnothing = L(K_{w_0})$. Bedingung (iii), jetzt jedoch zusammen mit der Annahme $w_0 \in L$, impliziert $h(w) \in L$ bzw. $h(w) \notin \{0,1\}^* \backslash L$.

Zum 2. Fall ($w_0 \notin L$):

Nach obiger Bedingung (ii) gibt es eine Turingmaschine $\overline{\overline{TM}}$ mit Kodierung $\overline{\overline{w}} \in \{0,1\}^*$ und $\overline{\overline{w}} \in L$. Es wird die Gültigkeit von $L_{H_0} \leq L$ gezeigt, indem eine totale und berechenbare Funktion $h: \{0,1\}^* \to \{0,1\}^*$ angegeben wird, für die $w \in L_{H_0} \Leftrightarrow h(w) \in L$ gilt. Der Beweis erfolgt wie im 1. Fall; dabei wird die dortige Rolle von $\overline{TM}$ von $\overline{\overline{TM}}$ übernommen. ///

Satz 3.2-10 läßt sich auch so formulieren:

Satz 3.2-11:

Ist $L \subseteq \{0,1\}^*$ eine Menge von Kodierungen von Turingmaschinen. Dann ist *L* nur in den Fällen entscheidbar, daß $L = \varnothing$ ist oder daß *L* aus der Menge der Kodierungen aller Turingmaschinen besteht.

Sämtliche oben angeführten Entscheidungsprobleme sind nicht entscheidbar, d.h. es ist beispielsweise nicht entscheidbar, ob

- eine Turingmaschine eine konstante Funktion berechnet
- eine Turingmaschine eine vorgegebene Funktion berechnet
- die von einer Turingmaschine akzeptierte Sprache leer ist
- die von einer Turingmaschine akzeptierte Sprache endlich ist
- die von einer Turingmaschine akzeptierte Sprache mit einer entscheidbaren Menge übereinstimmt.

Der letzte Punkt kann in der Praxis folgendermaßen verwendet werden. Es zeigt sich nämlich, daß Programmverifikation bzw. die Verifikation einer Spezifikation algorithmisch unmöglich ist.

Satz 3.2-12:

Für kein algorithmisch lösbares Problem (für keine entscheidbare Menge) läßt sich durch einen einzigen Algorithmus testen, ob ein entworfener Algorithmus eine korrekte Lösung des Problems liefert.

Sucht man also nach Beispielen für entscheidbare Mengen, so wird man bei Mengen, die aus Kodierungen von Turingmaschinen mit speziellen Eigenschaften bestehen, nur in den trivialen Fällen fündig, die in Satz 3.2-11 beschrieben sind. Selbstverständlich gibt es nichttriviale entscheidbare Mengen (siehe Übungen); diese bestehen dann aber nicht ausschließlich aus Kodierungen von Turingmaschinen.

Die Bedingungen, unter denen Mengen aus Kodierungen von Turingmaschinen rekursiv aufzählbar ist, sind wesentlich komplexer:

Satz 3.2-13:

Es sei $L \subseteq \{0,1\}^*$ eine Menge von Kodierungen von Turingmaschinen. Dann ist L genau dann rekursiv aufzählbar, wenn folgende Eigenschaften (i) – (iii) gelten:

(i) Ist $w_1 \in L$, $L_1 = L(K_{w_1})$ und ist L_2 eine rekursiv aufzählbare Menge, etwa $L_2 = L(K_{w_2})$ für ein Wort $w_2 \in \{0,1\}^*$, mit $L_1 \subseteq L_2$, dann ist auch $w_2 \in L$.

(ii) Ist $w_1 \in L$ und $L_1 = L(K_{w_1})$ eine unendliche Menge, dann gibt es eine endliche Teilmenge $L_2 \subseteq L_1$ mit $L_2 = L(K_{w_2})$ und $w_2 \in L$.

(iii) Es sei $E \subseteq L$ die Menge der Kodierungen, die Turingmaschinen kodieren, deren akzeptierte Sprachen endlich sind; dann ist E entscheidbar.

Beispielsweise folgt aus Satz 3.2-11, daß die Mengen

$$L_{\neq\varnothing} = \left\{ w \,\middle|\, w \in \{0,1\}^*, \mathit{VERIFIZIER\,E\_TM}(w) = \mathsf{TRUE} \text{ und } L(K_w) \neq \varnothing \right\} \text{ und}$$

$$L_{=\varnothing} = \left\{ w \,\middle|\, w \in \{0,1\}^*, \mathit{VERIFIZIER\,E\_TM}(w) = \mathsf{TRUE} \text{ und } L(K_w) = \varnothing \right\}$$

nicht entscheidbar sind. Es wird jedoch nichts darüber gesagt, ob sie rekursiv aufzählbar sind oder nicht. Aus Satz 3.2.-13 folgt, daß $L_{=\varnothing}$ nicht rekursiv aufzählbar ist; denn Bedingung (i) ist verletzt: Die Menge $L_{=\varnothing}$ enthält das Wort w_0 mit $L(K_{w_0}) = \varnothing$ aus dem Beweis zu Satz

3.2-10. Setzt man $L_1 = L(K_{w_0})$ und $L_2 = \{0,1\}^*$, so ist L_2 trivialerweise rekursiv aufzählbar, etwa $L_2 = L(K_{w_2})$ für eine Kodierung $w_2 \in \{0,1\}^*$. Es gilt $L_1 \subseteq L_2$, aber $w_2 \notin L_{=\emptyset}$.

Im Fall von $L_{=\emptyset}$ und $L_{\neq\emptyset}$ kann man auch anders argumentieren: Man zeigt direkt, daß die Sprache $L_{\neq\emptyset}$ rekursiv aufzählbar und folglich $\{0,1\}^* \setminus L_{\neq\emptyset}$ nicht rekursiv aufzählbar ist (denn sonst wäre $L_{\neq\emptyset}$ entscheidbar). Weiterhin gilt $\{0,1\}^* \setminus L_{\neq\emptyset} \leq L_{=\emptyset}$, und daher ist $L_{=\emptyset}$ nicht rekursiv aufzählbar. Dazu ist zum einen eine Turingmaschine *TM* anzugeben mit $L(TM) = L_{\neq\emptyset}$, zum anderen zum Nachweis der Relation $\{0,1\}^* \setminus L_{\neq\emptyset} \leq L_{=\emptyset}$ eine „geeignete“ Funktion $h : \{0,1\}^* \to \{0,1\}^*$.

TM arbeitet wie folgt: Als Eingabe erhält *TM* ein Wort $w \in \{0,1\}^*$. Falls
VERIFIZIERE_TM(*w*) = FALSE ist, wird *w* nicht akzeptiert. Andernfalls beginnt *TM*, alle Worte $z \in \{0,1,\#\}^*$ der Form $z = v\#bin(i)$ mit $v \in \{0,1\}^*$ in lexikographischer Reihenfolge zu erzeugen. Jedesmal, wenn ein derartiges Wort generiert worden ist, simuliert *TM* das Verhalten von K_w bei Eingabe von *v* für höchstens *i* Schritte. Wird *v* dabei akzeptiert, akzeptiert *TM* die Eingabe *w*. Andernfalls wird das nächste Wort $v'\#bin(i')$ erzeugt und getestet. Es gilt: Ist $w \in L(TM)$, dann gibt es ein Wort $v \in \{0,1\}^*$, so daß K_w dieses Wort in höchstens *i* Schritten für ein $i \in \mathbf{N}$ akzeptiert. Daher ist $L(K_w) \neq \emptyset$ und $w \in L_{\neq\emptyset}$.
Ist umgekehrt $w \in L_{\neq\emptyset}$, d.h. $L(K_w) \neq \emptyset$, dann gibt es ein Wort $v \in L(K_w)$. Dieses Wort wird in endlich vielen, etwa *j* Schritten akzeptiert. Wenn *TM* also bei Eingabe von *w* das Wort $v\#bin(j)$ generiert hat, stoppt K_w nach der Simulation von *j* Schritten im akzeptierenden Zustand, und *TM* akzeptiert *w*, d.h. $w \in L(TM)$.

Man kann sich leicht davon überzeugen, daß zum Nachweis der Relation $\{0,1\}^* \setminus L_{\neq\emptyset} \leq L_{=\emptyset}$ folgende Funktion geeignet ist:

$$h : \begin{cases} \{0,1\}^* & \to & \{0,1\}^* \\ w & \to & \begin{cases} w & \text{falls } VERIFIZIERE\_TM(w) = \text{TRUE} \\ w_0 & \text{sonst} \end{cases} \end{cases}$$

Hierbei ist w_0 die Kodierung einer Turingmaschine mit $L(K_{w_0}) = \emptyset$.

Für die in Satz 3.2-9 untersuchten Sprachen

$$L_e = \left\{ w \,\middle|\, w \in \{0,1\}^*, VERIFIZIERE\_TM(w) = \text{TRUE} \text{ und } L(K_w) \text{ ist entscheidbar} \right\} \text{ und}$$

$$L_{ne} = \left\{ w \,\middle|\, w \in \{0,1\}^*, VERIFIZIERE\_TM(w) = \text{TRUE} \text{ und } L(K_w) \text{ ist nicht entscheidbar} \right\}$$

folgt ebenfalls aus Satz 3.2-13, daß sie nicht rekursiv aufzählbar sind; in beiden Fällen ist wieder Bedingung (i) verletzt.

Zusammenfassung wichtiger Beispiele:

nicht entscheidbar, aber rekursiv aufzählbar	nicht rekursiv aufzählbar
$L_H = \left\{ u\#w \,\middle|\, \begin{array}{l} u \in \{0,1\}^*, w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ K_w \text{ stoppt bei Eingabe von } u \end{array} \right\}$	
$L_{H_0} = \left\{ w \,\middle|\, \begin{array}{l} w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ K_w \text{ stoppt bei leerem Eingabeband} \end{array} \right\}$	
$\overline{L}_d = \left\{ w \,\middle|\, \begin{array}{l} w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ w_i \in L(K_{w_i}) \end{array} \right\}$	$L_d = \left\{ w \,\middle|\, \begin{array}{l} w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{FALSE} \\ \text{oder } w_i \notin L(K_{w_i}) \end{array} \right\}$
$L_{uni} = \left\{ u\#w \,\middle|\, \begin{array}{l} u \in \{0,1\}^*, w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ u \in L(K_w) \end{array} \right\}$	$\overline{L}_{uni} = \left\{ z \,\middle|\, \begin{array}{l} z \text{ hat nicht die Form } z = u\#w \\ \text{mit } u \in \{0,1\}^*, w \in \{0,1\}^* \text{ oder} \\ VERIFIZIERE\_TM(w) = \mathsf{FALSE} \text{ oder} \\ u \notin L(K_w) \end{array} \right\}$
$L_{\neq\emptyset} = \left\{ w \,\middle|\, \begin{array}{l} w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ L(K_w) \neq \emptyset \end{array} \right\}$	$L_{=\emptyset} = \left\{ w \,\middle|\, \begin{array}{l} w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ L(K_w) = \emptyset \end{array} \right\}$
	$L_e = \left\{ w \,\middle|\, \begin{array}{l} w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ L(K_w) \text{ ist entscheidbar} \end{array} \right\}$
	$L_{ne} = \left\{ w \,\middle|\, \begin{array}{l} w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE}, \\ L(K_w) \text{ ist nicht entscheidbar} \end{array} \right\}$

4 Elemente der Theorie Formaler Sprachen und der Automatentheorie

Bisher wurden Sprachen über die Akzeptanz durch Turingmaschinen definiert. Das Ergebnis ist die Klasse der rekursiv aufzählbaren Sprachen. Diese soll nun weiter strukturiert werden, indem das Modell der Turingmaschine eingeschränkt wird. Eine Einschränkung wurde bereits in Kapitel 3.1 behandelt: es werden dort Turingmaschinen betrachtet, die bei jeder Eingabe stoppen. Diese Spezialisierung führte auf die Klasse der entscheidbaren Sprachen, die eine echte Teilklasse der Klasse der rekursiv aufzählbaren Sprachen ist.

Insgesamt werden in den folgenden Unterkapiteln vier Sprachklassen beschrieben. Diese Einteilung ist nach Noam Chomsky benannt, der diese Klassen als mögliche Modelle für natürliche Sprachen charakterisiert hat. Allerdings definiert die **Chomskyhierarchie** Sprachen nicht über die Akzeptanz durch geeignete Maschinenmodelle, sondern definiert jede Klasse durch die Form von syntaktischen Regeln, nach denen Wörter der jeweiligen Sprache „erzeugt“ werden können. Für jede Sprache wird eine entsprechende (formale) Grammatik festgelegt. Je nach Art der erzeugenden Regeln ist die erzeugte Sprache eine Typ-0-Sprache, Typ-1-Sprache, Typ-2-Sprache bzw. Typ-3-Sprache. Die zugrundeliegende Theorie heißt **Theorie der Formalen Sprachen**. Zu jedem Sprachtypen der Chomskyhierarchie gibt es ein entsprechendes Berechnungsmodell (Automatentyp), das sich aus dem Modell der Turingmaschine (durch die erwähnten Einschränkungen) ableitet. Somit besteht ein enger Zusammenhang zwischen der Theorie der Formalen Sprachen und der **Automatentheorie**, die sich wesentlich mit der Definition von Modellen der Berechenbarkeit und Übersetzbarkeit beschäftigt. Beide Ansätze werden in den folgenden Unterkapiteln gegenübergestellt.

4.1 Grammatiken und formale Sprachen

Die Akzeptanz (das Erkennen) einer Sprache $L \subseteq \Sigma^*$ über einem endlichen Alphabet Σ mit Hilfe eines Berechnungsmodells wie der Turingmaschine kann als „analytischer“ Ansatz bezeichnet werden. Dem gegenüber steht ein „synthetischer“ Ansatz, der beschreibt, wie die Wörter einer Sprache mit Hilfe von Regeln erzeugt werden können. Dieser Ansatz wird in der **Theorie der formalen Sprachen** verfolgt.

Eine **Grammatik** $G = (\Sigma, N, S, R)$ wird definiert durch

1. das endliche Alphabet Σ der **Terminalsymbole**
2. das endliche Alphabet N der **Nichtterminalsymbole (Variablen)**
3. das **Startsymbol** $S \in N$
4. eine endliche Menge R von **Erzeugungsregeln** (**Ableitungsregeln**, **Produktionen**) mit

$$R \subseteq \left\{ v \to w \mid v \in (N \cup \Sigma)^+, w \in (N \cup \Sigma)^*, |v| \geq 1 \right\}.$$

Für $x \in (N \cup \Sigma)^*$, $y \in (N \cup \Sigma)^*$, $v \in (N \cup \Sigma)^*$, $w \in (N \cup \Sigma)^*$ heißt das Wort *xwy* **von** *G* **in einem Schritt aus** *xvy* **erzeugt**, wenn es eine Erzeugungsregel $v \to w$ in *R* gibt. Man schreibt dann: $xvy \overset{v \to w}{\Rightarrow} xwy$ oder $xvy \underset{G}{\Rightarrow} xwy$.

Wenn der Zusammenhang klar ist, werden die Super- bzw. Subskripte auch weggelassen.

Ein Wort $w \in (N \cup \Sigma)^*$ heißt **von** *G* **aus** $x \in (N \cup \Sigma)^*$ **erzeugt**, wenn es eine Folge von Wörtern $x_i \in (N \cup \Sigma)^*$, $i = 0, ..., t$, gibt mit $x = x_0$, $x_i \underset{G}{\Rightarrow} x_{i+1}$ für $i = 0, ..., t-1$, $w = x_t$. Man schreibt dann auch $x \overset{*}{\underset{G}{\Rightarrow}} w$. Die Menge $L(G) = \left\{ w \,\middle|\, w \in \Sigma^* \text{ und } S \overset{*}{\underset{G}{\Rightarrow}} w \right\}$ heißt **die von** *G* **erzeugte Sprache**.

Beispiele:

Grammatik für einige Sprachen über $\{a,b,c\}$

1. $G_1 = (\{a,b\}, \{S, A, B\}, S, \{S \to AB, A \to aA, A \to e, B \to bB, B \to e\})$ erzeugt die Sprache $L_1 = \{a^i b^j \mid i \in \mathbf{N},\ j \in \mathbf{N}\}$. Die Sprache L_1 wird auch von der Grammatik $G_1' = (\{a,b\}, \{S, B\}, S, \{S \to e, S \to aS, S \to bB, B \to bB, B \to e\})$ erzeugt.

2. $G_2 = (\{a,b\}, \{S\}, S, \{S \to e, S \to aSb\})$ erzeugt die Sprache $L_2 = \{a^n b^n \mid n \in \mathbf{N}\}$

3. $G_3 = (\{a,b\}, \{S, A, B\}, S, \{S \to AB, A \to e, A \to aAb, B \to e, B \to cB\})$ erzeugt die Sprache $L_3 = \{a^n b^n c^i \mid n \in \mathbf{N}, i \in \mathbf{N}\}$

4. $G_4 = (\{a,b,c\}, \{S, B\}, S, \{S \to aSBc, S \to abc, cB \to Bc, bB \to bb\})$ erzeugt die Sprache $L_4 = \{a^n b^n c^n \mid n \in \mathbf{N}, n \geq 1\}$. Die Sprache L_4 wird auch von der Grammatik $G_4' = (\{a,b,c\}, \{S, B, C\}, S, \{S \to aSBC, S \to aBC, CB \to BC, aB \to ab, bB \to bb, bC \to bc, cC \to cc\})$ erzeugt.

5. $G_5 = (\{0,1\}, \{S, A, B\}, S, \{S \to 1A, S \to 0B, A \to 0, A \to 0S, A \to 1AA, B \to 1, B \to 1S, B \to 0BB\})$ erzeugt die Sprache

$L_5 = \{ w \mid w \in \{0,1\}^+$ und die Anzahl der Zeichen 0 in w ist gleich der Anzahl der Zeichen $1\}$

Eine Grammatik $G = (\Sigma, N, S, R)$ kann ähnlich wie eine Turingmaschine (siehe Kapitel 2.4) durch eine Zeichenkette aus $\{0,1\}^*$ (algorithmisch auf deterministische Weise) kodiert werden. Die Kodierung ist dabei eine injektive Abbildung, die jeder Grammatik G ein Wort $code(G) \in \{0,1\}^*$ zuordnet. Zusätzlich kann die Codierung so entworfen werden, daß man entscheiden kann, ob ein Wort $w \in \{0,1\}^*$ die Kodierung einer Grammatik darstellt, und daß man aus der Kodierung einer Grammatik diese rekonstruieren kann. Auf Details der Darstellung und der Verfahren soll hier verzichtet werden.

Für ein Wort $w \in \{0,1\}^*$ ist

$$VERIFIZIERE\_G(w) = \begin{cases} \mathsf{TRUE} & \text{falls } w \text{ die Kodierung einer Grammatik darstellt} \\ \mathsf{FALSE} & \text{falls } w \text{ nicht die Kodierung einer Grammatik darstellt} \end{cases}$$

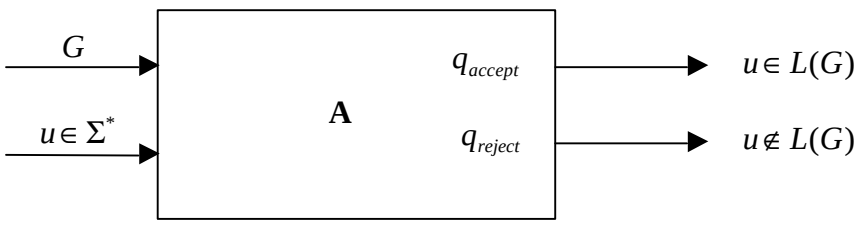
Die durch ein Wort $w \in \{0,1\}^*$ mit *VERIFIZIERE_G*(w) = TRUE kodierte Grammatik sei KG_w.

Die **zum Wortproblem gehörende Menge** L_{Wort} wird definiert durch

$$L_{Wort} = \left\{ u\#w \;\middle|\; u \in \Sigma^*, w \in \{0,1\}^*, VERIFIZIERE\_G(w) = \mathsf{TRUE} \text{ und } u \in L(KG_w) \right\}.$$

Das **Wortproblem ist entscheidbar**, wenn L_{Wort} entscheidbar ist. In diesem Fall gibt es einen auf allen Eingaben der Form $u\#w$ mit $u \in \Sigma^*$ und $w \in \{0,1\}^*$ stoppenden Algorithmus, der die Eingabe genau dann akzeptiert, wenn $u\#w \in L_{Wort}$ ist.

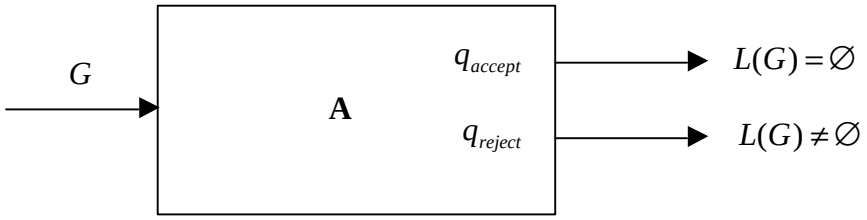
Vereinfacht ausgedrückt ist **das Wortproblem entscheidbar**, wenn es einen auf jeder Eingabe stoppenden Algorithmus **A** gibt, der eine aus zwei Teilen bestehende Eingabe, nämlich bestehend aus (der Kodierung) einer Grammatik $G = (\Sigma, N, S, R)$ und einer Zeichenkette $u \in \Sigma^*$, erhält und die Eingabe genau dann akzeptiert, wenn $u \in L(G)$ gilt, d.h. wenn sich u aus dem Startsymbol von G durch Anwendung der in G definierten Ableitungsregeln herleiten läßt.



Entsprechend kann man das Leerheitsproblem definieren:

Die **zum Leerheitsproblem gehörende Menge** L_{leer} wird definiert durch
$L_{leer} = \left\{ w \mid \; w \in \{0,1\}^*, VERIFIZIERE\_G(w) = \mathsf{TRUE} \text{ und } L(KG_w) = \varnothing \right\}.$

Das **Leerheitsproblem ist entscheidbar** (hier in der vereinfachten Darstellung), wenn es einen auf jeder Eingabe stoppenden Algorithmus **A** gibt, der als Eingabe eine Grammatik $G = (\Sigma, N, S, R)$ erhält und die Eingabe genau dann akzeptiert, wenn $L(G) = \varnothing$ gilt.



4.2 Typ-0-Sprachen

Eine Sprache, die von einer Grammatik $G = (\Sigma, N, S, R)$ erzeugt wird, für die $R \subseteq \left\{ v \to w \mid v \in (N \cup \Sigma)^+ \setminus \Sigma^*, w \in (N \cup \Sigma)^* \right\}$ gilt, heißt **Typ-0-Sprache**. Die Regeln $v \to w$ einer Typ-0-Sprache erfüllen also die Bedingung $|v| \geq 1$, und die linke Seite *v* einer Regel enthält mindestens ein nichtterminales Symbol; außerdem kann in einer Ableitung durch Anwendung einer Regel keine einmal entstandene rein terminale Zeichenkette mehr ersetzt werden. In einem Ableitungsschritt $xvy \overset{v \to w}{\Rightarrow} xwy$ kann es aber durchaus vorkommen, daß das Ergebnis *xwy* des Ableitungsschritts kürzer als die Ausgangszeichenkette ist.

Man kann zeigen, daß jede Typ-0-Sprache mit einem Alphabet Σ von einer Turingmaschine mit Eingabealphabet Σ akzeptiert werden kann. Dazu wird der Erzeugungsvorgang einer Grammatik mit Hilfe einer nichtdeterministischen Turingmaschine simuliert. Umgekehrt läßt sich jede rekursiv aufzählbare Menge durch eine Typ-0-Grammatik erzeugen. Daher gilt:

Satz 4.2-1:
Die Klasse der rekursiv aufzählbaren Mengen (von Turingmaschinen akzeptierte Mengen) über einem endlichen Alphabet Σ ist mit der Klasse der Typ-0-Sprachen mit Alphabet Σ identisch.

In Kapitel 3.2 wird gezeigt, daß die Menge

$$L_{uni} = \left\{ u\#w \,\middle|\, u \in \{0,1\}^*, w \in \{0,1\}^*, VERIFIZIERE\_TM(w) = \mathsf{TRUE} \text{ und } u \in L(K_w) \right\}$$

nicht entscheidbar ist. Satz 4.2-1 legt nahe, in der Definition von L_{uni} das Prädikat *VERIFIZIERE_TM* durch *VERIFIZIERE_G* zu ersetzen und dieses dann durch das leicht modifizierte Prädikat

$$VERIFIZIERE\_G\_TYP\_0(w) = \begin{cases} \mathsf{TRUE} & \text{falls } w \text{ die Kodierung einer Typ-0-Grammatik darstellt} \\ \mathsf{FALSE} & \text{falls } w \text{ nicht die Kodierung einer Typ-0-Grammatik darstellt} \end{cases}$$

Man erhält dann das **Wortproblem für Typ-0-Grammatiken**, das danach fragt, ob die Menge

$$L_{Wort\_Typ-0} = \left\{ u\#w \,\middle|\, u \in \Sigma^*, w \in \{0,1\}^*, VERIFIZIERE\_G\_TYP\_0(w) = \mathsf{TRUE} \text{ und } u \in L(KG_w) \right\}$$

entscheidbar ist. Diese Frage muß verneint werden.

Ähnlich verhält es sich mit dem **Leerheitsproblem für Typ-0-Grammatiken**. Die Menge

$$L_{leer\_Typ-0} = \left\{ w \,\middle|\, w \in \{0,1\}^*, VERIFIZIERE\_G\_TYP\_0(w) = \mathsf{TRUE} \text{ und } L(KG_w) = \varnothing \right\}$$

ist nicht entscheidbar, da (vgl. Kapitel 3.2) die Menge

$$L_{=\varnothing} = \left\{ w \,\middle|\, w \in \{0,1\}^*, VERIFIZIERE\_TM(w) = \mathsf{TRUE} \text{ und } L(K_w) = \varnothing \right\}$$

nicht rekursiv aufzählbar und damit auch nicht entscheidbar ist.

Insgesamt ergibt sich

Satz 4.2-2:

Das Wortproblem und das Leerheitsproblem für Typ-0-Grammatiken sind nicht entscheidbar:

Es gibt keinen auf allen Eingaben stoppenden Algorithmus**,** der bei Eingabe einer Typ-0-Grammatik $G = (\Sigma, N, S, R)$ und eines Wortes $u \in \Sigma^*$ entscheidet, ob $u \in L(G)$ ist.

Es gibt keinen auf allen Eingaben stoppenden Algorithmus**,** der bei Eingabe einer Typ-0-Grammatik $G = (\Sigma, N, S, R)$ entscheidet, ob $L(G) = \varnothing$ gilt.

4.3 Typ-1-Sprachen

Es sei $G = (\Sigma, N, S, R)$ eine Grammatik, in der die Erzeugungsregeln

$R \subseteq \{v \to w \mid v \in (N \cup \Sigma)^+ \setminus \Sigma^*, w \in (N \cup \Sigma)^*\}$ folgender zusätzlicher Einschränkung unterliegen:
Für alle Regeln $v \to w$ gilt $|v| \le |w|$. Als einzige Ausnahme ist die Regel $S \to e$ zugelassen; wenn diese Regel vorkommt, darf *S* auf keiner rechten Seite einer Regel vorkommen.

Wie bei einer Typ-0-Grammatik dürfen auf der linken Seite einer Regel in einer Typ-1-Grammatik also sowohl terminale als auch nichtterminale Zeichen vorkommen, wobei links mindestens ein nichtterminales Zeichen steht. Die Länge der rechten Seite einer Regel ist bis auf die Ausnahme $S \to e$ mindestens so groß wie die Länge der linken Seite. Daher wird in einem Ableitungsschritt $xvy \overset{v \to w}{\Rightarrow} xwy$ die Länge des Ableitungsergebnisses nicht kürzer. In einer Ableitung $S \overset{*}{\Rightarrow} w$ eines Wortes $w \in \Sigma^+$, etwa $S \Rightarrow x_1 \Rightarrow ... \Rightarrow x_i \Rightarrow x_{i+1} \Rightarrow ... \Rightarrow w$, gilt für alle Zwischenschritte $1 = |S| \le |x_1| \le ... \le |x_i| \le |x_{i+1}| \le ... \le |w|$. Diese Beobachtung wird wichtig, wenn man den Turingmaschinentyp charakterisieren möchte, der eine von einer Typ-1-Grammatik erzeugten Sprache erkennt.

Enthält *R* die Regel $S \to e$, dann ist $e \in L(G)$, und die Anwendung der Regel $S \to e$ stellt die einzige Möglichkeit dar, um e aus *S* abzuleiten.

Die von einer derartigen Grammatik erzeugte Sprache heißt **Typ-1-Sprache** oder **kontextsensitive Sprache**. Die Grammatik heißt **kontextsensitive Grammatik**. Beispielsweise ist die Sprache $L_4 = \{a^n b^n c^n \mid n \in \mathbf{N}, n \ge 1\}$ aus Kapitel 4.1 eine kontextsensitive Sprache (Typ-1-Sprache).

Jede Typ-1-Sprache ist natürlich auch eine Typ-0-Sprache.

Es stellt sich die Frage, ob es Typ-0-Sprachen gibt, die nicht kontextsensitiv sind. Zur Beantwortung dieser Frage wird folgender Ansatz gewählt: Es wird ein Berechnungsmodell vorgestellt, das sich aus dem Modell der Turingmaschine ableitet und genau kontextsensitive Sprachen akzeptiert, und dann untersucht, wie sich dieses Berechnungsmodell zum Modell der Turingmaschine verhält:

Ein **linear beschränkter Automat** *LBA* ist eine nichtdeterministische Turingmaschine, deren Schreib/Leseköpfe auf allen Bändern bei Eingabe eines Wortes *w* mit $|w| = n$ jeweils nicht mehr als $n+1$ Zellen auf den Bändern verwenden. Insbesondere bewegt sich kein Kopf weiter nach rechts als bis zur Position $n+1$ (an dem Blank auf dem Eingabeband bei Position $n+1$ wird das Ende des Eingabeworts erkannt). Die von einem *LBA* akzeptierte Menge $L(LBA)$ wird wie bei Turingmaschinen definiert.

Man kann zeigen, daß es zu jeder von einem linear beschränkten Automaten *LBA* akzeptierten Sprache $L = L(LBA)$ eine kontextsensitive Grammatik G_{LBA} gibt mit $L(G_{LBA}) = L(LBA)$. Umgekehrt gibt es zu jeder von einer kontextsensitiven Grammatik *G* erzeugten Sprache $L' = L(G)$ einen linear beschränkten Automaten LBA_G mit $L(LBA_G) = L(G)$. In diese Überlegungen geht wesentlich ein, daß die Produktionen $v \to w$ der kontextsensitiven Grammatik der Bedingung Regeln $|v| \le |w|$ genügen, so daß zur Akzeptanz auf allen Bändern jeweils ein durch die Länge des Eingabeworts linear beschränkter Speicherplatz ausreicht.

Satz 4.3-1:

Die Klasse der von linear beschränkten Automaten akzeptierten Sprachen über einem endlichen Alphabet Σ ist mit der Klasse der kontextsensitiven Sprachen (Typ-1-Sprachen) über Σ identisch.

Möchte man daher Aussagen über kontextsensitive Sprachen beweisen, so kann man dazu mit kontextsensitiven Grammatiken oder linear beschränkten Automaten argumentieren. Satz 4.3-1 läßt vermuten, daß die Klasse der kontextsensitiven Sprachen über Σ eine echte Teilklasse der rekursiv aufzählbaren Sprachen über Σ ist. Der folgende Satz bestätigt diese Vermutung.

Satz 4.3-2:

Die von einem linear beschränkten Automaten *LBA* akzeptierte Menge $L(LBA)$ über einem Alphabet Σ ist entscheidbar.

Jede kontextsensitive Sprache über einem Alphabet Σ ist entscheidbar.

Die Entscheidung kann bei einem Wort $w \in \Sigma^*$ mit Länge *n* in $O(2^{O(n)})$ vielen Schritten getroffen werden; das Entscheidungsverfahren hat also exponentielle Laufzeit.

Beweis:

Es sei $G = (\Sigma, N, S, R)$ eine kontextsensitive Grammatik. Zu zeigen ist: $L(G)$ ist entscheidbar. Dazu wird eine auf allen Eingaben $w \in \Sigma^*$ stoppende Turingmaschine TM_G angegeben, die *w* genau dann akzeptiert, wenn $w \in L(G)$ ist. Die Arbeitsweise von TM_G wird hier informell in Form eines Algorithmus beschrieben:

Bei Eingabe von $w \in \Sigma^*$ mit $|w| = n$ erzeugt TM_G einen Graphen, dessen Knoten mit den Zeichenketten in $(N \cup \Sigma)^*$ markiert sind, die eine Länge besitzen, die kleiner oder gleich *n* ist

(andere Zeichenketten kommen in einer Ableitung $S \overset{*}{\Rightarrow} w$ nicht vor). Ist dabei der Knoten K_i mit der Zeichenkette $a \in (N \cup \Sigma)^*$ und der Knoten K_j mit $b \in (N \cup \Sigma)^*$ markiert und kann man in *G* durch Anwendung einer Regel b aus a in einem Ableitungsschritt herleiten, d.h. $a \underset{G}{\Rightarrow} b$, dann wird eine Kante von K_i nach K_j eingefügt. Ein Knoten ist mit *S* markiert und einer mit *w*. Es ist $w \in L(G)$ genau dann, wenn es einen Pfad von dem mit *S* markierten Knoten zu dem mit *w* markierten Knoten gibt. Um diese Tatsache festzustellen, kann man einen der bekannten Algorithmen zum Auffinden von Pfaden in Graphen zwischen definierten Knoten anwenden. Da der beschriebene Graph $O(c^n)$ viele Knoten (mit einer Konstanten $c = c(G)$) besitzt und sich die Laufzeit des Pfadsuchalgorithmus durch eine Funktion der Ordnung $O(m^3)$ beschränken läßt, wobei $m \in O(c^n)$ die Anzahl der Knoten im Graphen angibt, ist das gesamte Entscheidungsverfahren von der Ordnung $O(2^{O(n)})$. ///

Die Klasse der kontextsensitiven Sprachen ist eine echte Teilklasse der Klasse der entscheidbaren Sprachen, wie folgende Sätze zeigen.

Mit der Diagonalisierungstechnik läßt sich zunächst folgender (technischer) Hilfssatz zeigen:

Satz 4.3-3:

Es sei

$$L_0 \subseteq \left\{ w \,\middle|\, w \in \{0,1\}^*, \mathit{VERIFIZIERE\_TM}(w) = \mathsf{TRUE} \text{ und } K_w \text{ stoppt auf jeder Eingabe} \right\}$$

eine rekursiv aufzählbare Menge. Dann gibt es eine entscheidbare Sprache *L*, die von einer auf allen Eingaben stoppenden Turingmaschine *TM* erkannt wird, deren Kodierung in L_0 nicht vorkommt.

Bemerkung: Da die Menge

$$L_e = \left\{ w \,\middle|\, w \in \{0,1\}^*, \mathit{VERIFIZIERE\_TM}(w) = \mathsf{TRUE} \text{ und } L(K_w) \text{ ist entscheidbar} \right\}$$

aus Kapitel 3.2 nicht rekursiv aufzählbar ist, gilt $L_0 \neq L_e$.

Beweis:

Da L_0 als rekursiv aufzählbar angenommen wird, gibt es nach Satz 3.1-2 eine totale berechenbare Funktion $h : \{0,1\}^* \to \Sigma^*$ mit $L_0 = h(\{0,1\}^*)$.

Es wird $L = \left\{ w \,\middle|\, w \in \{0,1\}^* \text{ und } w \notin L(K_{h(w)}) \right\}$ gesetzt.

Dann ist *L* entscheidbar: Eine auf allen Eingaben $w \in \{0,1\}^*$ stoppende Turingmaschine *TM* mit $L(TM) = L$, d.h. die entscheidet, ob $w \in L$ gilt oder nicht, arbeitet wie folgt: *TM* berech-

net zunächst $h(w)$. Dabei ist $h(w) \in L_0$, insbesondere $VERIFIZIERE\_TM(h(w)) = \mathsf{TRUE}$. Jetzt simuliert *TM* das Verhalten von $K_{h(w)}$ bei Eingabe von *w*. Da $K_{h(w)}$ nach Definition von L_0 auf allen Eingaben stoppt, kann *TM* feststellen, ob $w \in L(K_{h(w)})$ gilt oder nicht. Das Wort *w* wird von *TM* genau dann akzeptiert, wenn bei dieser Simulation $w \notin L(K_{h(w)})$ festgestellt wird.

TM habe die Kodierung w_L. Falls $w_L \in L_0$ gilt, dann sei $w \in \{0,1\}^*$ so gewählt, daß $h(w) = w_L$ ist. Dann folgt (nach Definition von *L*) der Widerspruch

$w \in L \Leftrightarrow w \notin L(K_{h(w)})$ (nach Definition von *L*)

$\Leftrightarrow w \notin L(K_{w_L})$ (wegen $h(w) = w_L$)

$\Leftrightarrow w \notin L(TM)$ (da w_L die Kodierung von *TM* ist)

$\Leftrightarrow w \notin L$ (wegen $L(TM) = L$).

Daher kommt die Kodierung w_L von *TM* in L_0 nicht vor. ///

Satz 4.3-3 kann genutzt werden, um zu zeigen, daß es eine entscheidbare Menge gibt, die nicht kontextsensitiv ist:

Jede kontextsensitive Grammatik kann ähnlich wie eine Turingmaschine (siehe Kapitel 2.4 und Kapitel 4) durch eine Zeichenkette aus $\{0,1\}^*$ (algorithmisch auf deterministische Weise) kodiert werden. Wie in Kapitel 2.4 auf der Menge der Turingmaschinen kann man dann auf der Menge der kontextsensitiven Grammatiken auf Basis ihrer Kodierungen eine lineare Ordnung definieren. Man kann also von der *i*-ten kontextsensitiven Grammatik G_i sprechen. Nach Satz 4.3-2 ist $L(G_i)$ entscheidbar mit einer (dort angegebenen) Turingmaschine TM_{G_i}; die Turingmaschine TM_{G_i} stoppt (nach Konstruktion) auf jeder Eingabe. Deren Kodierung sei (gemäß dem Vorgehen aus Kapitel 2.4) $code(TM_{G_i})$. Die Berechnung des Werts $code(TM_{G_i})$ bei Vorgabe von *i* bzw. von $bin(i)$ erfolgt insgesamt auf deterministische Weise und definiert eine Funktion $h: \{0,1\}^* \to \{0,1\}^*$:

$(i \leftrightarrow) bin(i) \to i$-te kontextsensitve Grammatik $G_i \to TM_{G_i} \to code(TM_{G_i})$.

Diese Abbildung ist injektiv, wie man leicht nachprüfen kann. Durch *h* wird jeder Zeichenkette $u = bin(i)$ die Kodierung einer Turingmaschine zugeordnet, die auf jeder Eingabe stoppt. Setzt man $L_0 = h(\{0,1\}^*)$, so sind die Voraussetzungen in Satz 4.3-3 erfüllt. Daher gilt der folgende Satz.

Satz 4.3-4:

Die Klasse der kontextsensitiven Sprachen (Typ-1-Sprachen) über einem endlichen Alphabet Σ ist eine echte Untermenge der entscheidbaren Sprachen über Σ und damit auch der Typ-0-Sprachen über Σ.

Wie im allgemeinen Fall der Turingmaschine unterscheidet man auch bei linear beschränkten Automaten nichtdeterministisches und deterministisches Verhalten. Ein **nichtdeterministischer linear beschränkter Automat** liegt vor, wenn die Überführungsfunktion die Form $d : Q \times \Sigma^k \to \mathbf{P}\left(Q \times \left(\Sigma \times \{L,R,S\}\right)^k\right)$ hat, ein **deterministischer linear beschränkter Automat** liegt vor, wenn die Überführungsfunktion die Form $d : Q \times \Sigma^k \to Q \times \left(\Sigma \times \{L,R,S\}\right)^k$ hat. Im deterministischen Fall ist die Folgekonfiguration (falls sie überhaupt existiert) einer Konfiguration eindeutig bestimmt; zum Nichtdeterminismus vgl. Kapitel 2.5. Es ist nicht bekannt, ob jede von einem *nichtdeterministischen* linear beschränkten Automaten auch von einem *deterministischen* linear beschränkten Automaten akzeptiert wird. Dieses offene Problem heißt **LBA-Problem**. Zu beachten ist dabei folgendes: Ist L eine kontextsensitive Sprache, dann gibt es einen nichtdeterministischen linear beschränkten Automaten LBA mit $L = L(LBA)$. Die Akzeptanz eines Wortes w mit $|w| = n$ benötigt eine Anzahl von Zellen, die durch die lineare Funktion $S(n) = n + 1$ gegeben ist. Das nichtdeterministische Verhalten einer Turingmaschine, die durch eine Funktion der Ordnung $O(f(n))$ platzbeschränkt ist, kann deterministisch simuliert werden, wobei dabei der Speicherplatzbedarf die Ordnung $O\left(f^2(n)\right)$ annimmt. Da ein linear beschränkter Automat auch eine nichtdeterministische Turingmaschine ist, könnte man versuchen, diese deterministische Simulation hier zu verwenden. Diese führt jedoch aus der Klasse der linear beschränkten Automaten heraus, da sie quadratischen Speicherplatz der Ordnung $O\left(S^2(n)\right) = O\left(n^2\right)$ benötigt. Daher trägt die deterministische Simulation einer nichtdeterministischen Turingmaschine in dieser Allgemeinheit zur Lösung des LBA-Problems nichts bei.

Die Klasse der von deterministischen linear beschränkten Automaten akzeptierten Sprachen ist abgeschlossen gegen Komplementbildung (das zeigt man wie in Satz 3.2-2 Teil 3.). Man hat lange Zeit vermutet, daß diese Abschlußeigenschaft für die Klasse der von nichtdeterministischen linear beschränkten Automaten akzeptierten Sprachen nicht zutrifft. Die Richtigkeit dieser Vermutung hätte die (negative) Lösung des LBA-Problems nach sich gezogen. Inzwischen weiß man jedoch, daß auch die Klasse der von nichtdeterministischen linear beschränkten Automaten akzeptierten Sprachen abgeschlossen gegen Komplementbildung ist. Das LBA-Problem ist weiterhin ungelöst.

Die Definition des Wortproblems für Typ-0-Grammatiken kann man auf Typ-1-Grammatiken übertragen: Dazu wird das Prädikat

$VERIFIZIERE\_G\_TYP\_1(w)$

$$= \begin{cases} \mathsf{TRUE} & \text{falls } w \text{ die Kodierung einer Typ-1-Grammatik darstellt} \\ \mathsf{FALSE} & \text{falls } w \text{ nicht die Kodierung einer Typ-1-Grammatik darstellt} \end{cases}$$

definiert. Dieses Prädikat ist berechenbar. Dazu ist unter anderem zu prüfen, ob die Erzeugungsregeln in KG_w den Bedingungen einer kontextsensitiven Grammatik genügen.

Das **Wortproblem für Typ-1-Grammatiken** fragt danach, ob die Menge

$$L_{Wort\_Typ-1} = \left\{ u\#w \,\middle|\, u \in \Sigma^*, w \in \{0,1\}^*, VERIFIZIERE\_G\_TYP\_1(w) = \mathsf{TRUE} \text{ und } u \in L(KG_w) \right\}$$

entscheidbar ist.

Aus dem Beweis von Satz 4.3-2 folgt, daß das Wortproblem für kontextsensitive Grammatiken entscheidbar ist. Das **Leerheitsproblem für Typ-1-Grammatiken**, nämlich die Frage, ob die Menge

$$L_{leer\_Typ-1} = \left\{ w \,\middle|\, w \in \{0,1\}^*, VERIFIZIERE\_G\_TYP\_1(w) = \mathsf{TRUE} \text{ und } L(KG_w) = \varnothing \right\}$$

entscheidbar ist, muß wie bei Typ-0-Grammatiken verneint werden (der Beweis findet sich in der angegebenen Literatur).

Zusammenfassend ergibt sich

Satz 4.3-5:

Das Wortproblem für Typ-1-Grammatiken ist entscheidbar; das Leerheitsproblem für Typ-1-Grammatiken ist nicht entscheidbar:

Es gibt einen auf allen Eingaben stoppenden Algorithmus**,** der bei Eingabe einer Typ-1-Grammatik $G = (\Sigma, N, S, R)$ und eines Wortes $u \in \Sigma^*$ entscheidet, ob $u \in L(G)$ ist.

Es gibt keinen auf allen Eingaben stoppenden Algorithmus**,** der bei Eingabe einer Typ-1-Grammatik $G = (\Sigma, N, S, R)$ entscheidet, ob $L(G) = \varnothing$ gilt.

4.4 Typ-2-Sprachen

Es sei $G = (\Sigma, N, S, R)$ eine Grammatik, in der für die Erzeugungsregeln $R \subseteq \left\{ A \to w \,|\, A \in N \text{ und } w \in (N \cup \Sigma)^* \right\}$ gilt. Auf der linken Seite einer Regel steht immer genau ein nichtterminales Symbol, rechts kann auch die leere Zeichenkette vorkommen.

Die von einer derartigen Grammatik erzeugte Sprache heißt **Typ-2-Sprache** oder **kontextfreie Sprache**. Die zugehörige Grammatik heißt **kontextfreie Grammatik**. Beispielsweise sind die Sprache $L_2 = \left\{ a^n b^n \,|\, n \in \mathbf{N} \right\}$, $L_3 = \left\{ a^n b^n c^i \,|\, n \in \mathbf{N}, i \in \mathbf{N} \right\}$ und

$L_5 = \{ w \mid w \in \{0,1\}^+ \text{ und die Anzahl der Zeichen 0 in } w \text{ ist gleich der Anzahl der Zeichen 1} \}$ aus Kapitel 4.1 kontextfreie Sprachen (Typ-2-Sprachen).

Die kontextfreien Sprachen zählen zu den am intensivsten erforschten Sprachen. Einen überragenden Erfolg haben sie in der Anwendung und der Theorie des Compilerbaus. Programmiersprachen wie Pascal und ihre Nachfolger sind erst nach intensiver Erforschung der kontextfreien Sprachen und unter Anwendung dieser Theorie entwickelt worden. Die Syntax der meisten heute üblichen Programmiersprachen wird in Form einer Grammatik definiert, die weitestgehend kontextfrei ist. Man kann jedoch formal zeigen, daß eine Programmiersprache, in der Variablen mit Datentypen deklariert werden, so daß Typverträglichkeit verlangt wird, in der die Anzahl von Formal- und Aktualparametern in Prozeduren übereinstimmen müssen, in der ausschließlich die Verwendung vorher deklarierter Objekte zulässig ist usw., nicht komplett durch eine kontextfreie Grammatik beschrieben werden kann.

Beispiel:

Ausschnitt aus der Sprachdefinition der Programmiersprache Object Pascal

Die Syntax der Sprache Object Pascal wird wie bei vielen anderen Programmiersprachen (weitgehend) durch eine kontextfreie Grammatik definiert. Nichtterminale Symbole werden dabei durch Bezeichner angegeben, die mit einem Großbuchstaben beginnen und weitere Kleinbuchstaben enthalten. Bezeichner, die nur Großbuchstaben enthalten, stehen für jeweils ein einziges terminales Symbol. So bezeichnet im folgenden Ausschnitt der Bezeichner `Ziel` das Startsymbol der Sprache, während der Bezeichner `UNIT` für ein einziges terminales Symbol steht.

Eine Regelangabe der Form $A \to a_1 \mid a_2 \mid ... \mid a_n$ steht für die *n* Regeln $A \to a_1$, $A \to a_2$, ..., $A \to a_n$. Ein Angabe in eckigen Klammern innerhalb einer Regel bezeichnet einen optionalen Teil, d.h. $A \to a[b]g$ steht für die Regeln $A \to abg$ und $A \to ag$.

```
Ziel -> (Programm | Package  | Bibliothek  | Unit)
Programm -> [PROGRAM Bezeichner ['('Bezeichnerliste')'] ';']

            Programmblock '.'

Unit -> UNIT Bezeichner ';'

        interface-Abschnitt
        implementation-Abschnitt
        initialization-Abschnitt '.'

Package -> PACKAGE Bezeichner ';'

           [requires-Klausel]
           [contains-Klausel]
```

```
           END '.'

Bibliothek -> LIBRARY Bezeichner ';'

              Programmblock '.'

Programmblock -> [uses-Klausel]

                 Block

uses-Klausel -> USES Bezeichnerliste ';'
interface-Abschnitt -> INTERFACE

                       [uses-Klausel]
                       [interface-Deklaration]...

interface-Deklaration -> const-Abschnitt

                      -> type-Abschnitt
                      -> var-Abschnitt
                      -> exported-Kopf

exported-Kopf -> Prozedurkopf ';' [Direktive]

              -> Funktionskopf ';' [Direktive]

implementation-Abschnitt -> IMPLEMENTATION

                            [uses-Klausel]
                            [Deklarationsabschnitt]...

Block -> [Deklarationsabschnitt]

         Verbundanweisung

Deklarationsabschnitt -> Label-Deklarationsabschnitt

                      -> const-Abschnitt
                      -> type-Abschnitt
                      -> var-Abschnitt
                      -> Prozedurdeklarationsabschnitt

...

Einfache Anweisung -> Designator ['(' Ausdrucksliste ')']

                   -> Designator ':=' Ausdruck
                   -> INHERITED
                   -> GOTO Label-Bezeichner

Strukturierte Anweisung -> Verbundanweisung

                         -> Bedingte Anweisung
                         -> Schleifenanweisung
                         -> with-Anweisung

Verbundanweisung -> BEGIN Anweisungsliste END
Bedingte Anweisung -> if-Anweisung
```

```
                    -> case-Anweisung

if-Anweisung -> IF Ausdruck THEN Ausdruck [ELSE Ausdruck]
case-Anweisung -> CASE Ausdruck OF case-Selektor/';'... [ELSE Ausdruck] [';'] END
case-Selektor -> case-Label/','... ':' Anweisung
case-Label -> Konstanter Ausdruck ['..' Konstanter Ausdruck]
Schleifenanweisung -> repeat-Anweisung

                    -> while-Anweisung
                    -> for-Anweisung

repeat-Anweisung -> REPEAT Anweisung UNTIL Ausdruck
while-Anweisung -> WHILE Ausdruck DO Anweisung
for-Anweisung -> FOR Qualifizierter Bezeichner ':=' Ausdruck (TO | DOWNTO)
                   Ausdruck DO Anweisung
with-Anweisung -> WITH Bezeichnerliste DO Anweisung
Prozedurdeklarationsabschnitt -> Prozedurdeklaration

                              -> Funktionsdeklaration

...

Klassentyp -> CLASS [Klassenvererbung]

              [Klassenfelderliste]
              [Klassenmethodenliste]
              [Klasseneigenschaftenliste]
              END

Klassenvererbung -> '(' Bezeichnerliste ')'
Klassensichtbarkeit -> [PUBLIC | PROTECTED | PRIVATE | PUBLISHED]
Klassenfelderliste -> (Klassensichtbarkeit Objektfelderliste)/';'...
Klassenmethodenliste -> (Klassensichtbarkeit Methodenliste)/';'...
Klasseneigenschaftenliste -> (Klassensichtbarkeit Eigenschaftenliste ';')...
Eigenschaftenliste -> PROPERTY Bezeichner [Eigenschaftsschnittstelle]
                        Eigenschaftsbezeichner

Eigenschaftsschnittstelle -> [Eigenschaftsparameterliste] ':' Bezeichner
Eigenschaftsparameterliste -> '[' (Bezeichnerliste ':' Typbezeichner)/';'... ']'
Eigenschaftsbezeichner -> [INDEX Konstanter Ausdruck]

                          [READ Ident]
                          [WRITE Bezeichner]
                          [STORED Bezeichner | Konstante)]
                          [(DEFAULT Konstanter Ausdruck) | NODEFAULT]
                          [IMPLEMENTS Typbezeichner]

Schnittstellentyp -> INTERFACE [Schnittstellenvererbung]

                     [Klassenmethodenliste]
                     [Klasseneigenschaftenliste]
                     END

Schnittstellenvererbung -> '(' Bezeichnerliste ')'
requires-Klausel -> REQUIRES Bezeichnerliste... ';'
contains-Klausel -> CONTAINS Bezeichnerliste... ';'
Bezeichnerliste -> Bezeichner/','...
```

```
Qualifizierter Bezeichner -> [Unit-Bezeichner '.'] Bezeichner
Typbezeichner -> [Unit-Bezeichner '.'] <Typbezeichner>
Ident -> <Bezeichner>
ConstExpr -> <Konstanter Ausdruck>
UnitId -> <Unit-Bezeichner>
LabelId -> <Label-Bezeichner>

Number -> <Nummer>
String -> <String>
```

Die Erzeugungsregeln einer kontextfreien Grammatik erfüllen (auf den ersten Blick) auch die Bedingung, die man an eine kontextsensitive Grammtik stellt. In einer kontextfreien Grammatik können jedoch auch Produktionen der Form $A \to e$ vorkommen, wobei $A \neq S$ oder zusätzlich A auf der rechten Seite einer Produktion vorkommt. In diesem Fall kann man in einem endlichen Verfahren die Regeln und nichtterminalen Symbole der Grammatik so abändern, daß keine Produktionen der Form $A \to e$ mehr vorkommen außer wenn eventuell A das Startsymbol ist (A steht dann auf keiner rechten Seite einer Produktion). Die Änderung kann so erfolgen, daß weiterhin dieselbe Sprache erzeugt wird. Es gilt daher:

Satz 4.4-1:

Zu jeder kontextfreien Grammatik G gibt es eine kontextsensitive Grammatik G', die dieselbe Sprache erzeugt:
Die Klasse der kontextfreien Sprachen (Typ-1-Sprachen) über einem endlichen Alphabet Σ ist eine echte Teilmenge der kontextsensitiven Sprachen über Σ.

Die Tatsache, daß die Klasse der kontextfreien Sprachen über einem endlichen Alphabet Σ echt in der Klasse der kontextsensitiven Sprachen über Σ enthalten ist, wird weiter unten bewiesen.

Der folgende Satz, der auch hier ohne Beweis aufgeführt wird, besagt, daß man jede kontextfreie Grammatik in eine Grammatik in **Normalform** überführen kann, deren Regeln eine einfache Struktur aufweisen.

Satz 4.4-2:

Jede kontextfreie Grammatik kann so umgeformt werden, daß alle Regeln die Form
$S \to e$ oder
$A \to BC$ mit $A \in N$, $B \in N$, $C \in N$ oder
$A \to a$ mit $A \in N$, $a \in \Sigma$
haben (Chomsky-Normalform) und dabei dieselbe Sprache erzeugt wird.

Der folgende Satz (*uvwxy*-**Theorem, pumping lemma**) liefert ein Beweismittel, mit dessen Hilfe man zeigen kann, daß eine Sprache nicht kontextfrei ist.

Satz 4.4-3:

Zu jeder kontextfreien Sprache L gibt es eine Zahl $n_0 > 0$ mit der Eigenschaft:

jedes $z \in L$ mit $|z| \geq n_0$ läßt sich zerlegen in $z = uvwxy$ mit

(i) $|vwx| \leq n_0$

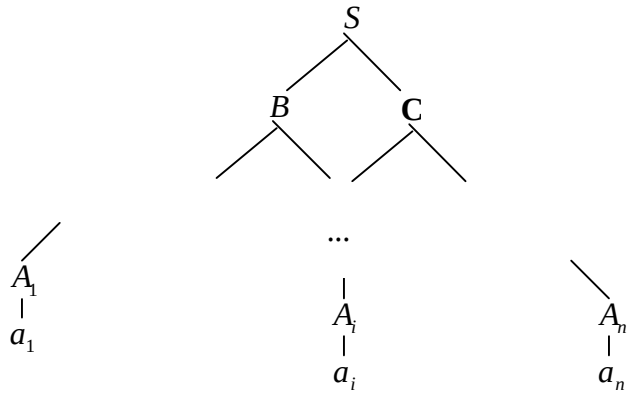
(ii) $|vx| > 0$

(iii) $uv^k wx^k y \in L$ für jedes $k \in \mathbf{N}$.

Beweis:

Es sei $G = (\Sigma, N, S, R)$ eine kontextfreie Grammatik in Chomsky-Normalform mit $L = L(G)$ und $n_0 = 2^{|N|+1}$.

Für ein Wort $z \in L$, etwa $z = a_1 ... a_n$ mit $a_1 \in \Sigma$, ..., $a_n \in \Sigma$ und $|z| = n \geq n_0$, hat eine Ableitung aus S die Form $S \Rightarrow BC \Rightarrow ... \Rightarrow a_1 ... a_n$. Diese Ableitung kann als Ableitungsbaum geschrieben werden, in dem jeder Knoten durch ein in der Ableitung vorkommendes Symbol markiert ist: Die Wurzel des Ableitungsbaums ist mit S markiert. Wird in der Ableitung eine Regel der Form $A \to BC$ mit $B \in N$ und $C \in N$ angewendet, so gibt es im Ableitungsbaum einen dieser Regelanwendung entsprechenden mit A markierten Knoten, dessen direkte Nachfolger mit B bzw. C markiert sind. Wird eine Regel der Form $A \to a$ mit $a \in \Sigma$ angewendet, so gibt es im Ableitungsbaum einen dieser Regelanwendung entsprechenden mit A markierten Knoten, dessen direkter Nachfolger mit a markiert ist. Da G Chomsky-Normalform hat, ist dieser Ableitungsbaum ein Binärbaum, in dem jeder innere Knoten genau zwei Nachfolger hat und der Anwendung einer Regel der Form $A \to BC$ entspricht. Die Blätter des Baums sind mit a_1, ..., a_n markiert. Ein mit a_i markiertes Blatt mit seinem mit A_i markierten Vorgänger entspricht der Anwendung der Regel $A_i \to a_i$. Nur an den Blättern werden Regeln dieser Form angewendet.



Für die Anzahl n der Blätter dieses Ableitungsbaums gilt $n = |z| \geq n_0 = 2^{|N|+1}$. Dann hat der Baum nach Satz 1.1-9 Teil 4. eine Mindesthöhe von $\lceil \log_2(n) + 1 \rceil \geq \left| \log_2\left(2^{|N|+1}\right) + 1 \right| = |N| + 2$.

Es gibt also einen Pfad von der Wurzel zu einem Blatt, das mit einem terminalen Zeichen a_i markiert ist, auf dem mindestens $|N|+1$ viele innere Knoten liegen, die mit nichtterminalen Symbol markiert sind. Da die Grammatik nur $|N|$ viele nichtterminale Symbole enthält, kommt auf diesem Pfad ein $A \in N$ mindestens zweimal vor. Es werde hier die Situation betrachtet, bei der dieses zum ersten Mal geschieht:

$S \Rightarrow^* w_1 A w_2 \Rightarrow^+ w_1 w_3 A w_4 w_2 \Rightarrow^* uvwxy$ mit $w_1 \Rightarrow^* u$, $w_3 \Rightarrow^* v$, $A \Rightarrow^* w$, $w_4 \Rightarrow^* x$ und $w_2 \Rightarrow^* y$.

Da *G* kontextfrei ist, hängen diese Ableitungen nicht zusammen, man kann sie unabhängig voneinander in einer beliebigen Reihenfolge ausführen. Daher sind in *G* auch folgende Ableitungen möglich:

$S \Rightarrow^* uAy \Rightarrow^+ uw_3 A w_4 y \Rightarrow^* uvAxy \Rightarrow^* uvwxy$.

Der erste Schritt in der Teilableitung $uAy \Rightarrow^+ uw_3 A w_4 y$ erfolgt durch Anwendung einer Regel der Form $A \to BC$, daher ergibt sich

$uAy \Rightarrow uBCy \Rightarrow^* uw_3 A w_4 y$ und $BC \Rightarrow^* w_3 A w_4 \Rightarrow^* vAx$. Wären beide Teilworte *v* und *x* leer, so könnte man in *G* eine Ableitung $BC \Rightarrow^* A$ durchführen. Dieses ist nicht möglich, da *G* in Chomsky-Normalform vorliegt. Daher gilt Eigenschaft (ii).

Die Teilableitung $A \Rightarrow^* vwx$ hat höchstens $|N|$ viele Ableitungsschritte, da in *G* wegen der Chomsky-Normalform keine Ableitungen der Form $A \Rightarrow^+ B$ möglich sind. Daher gilt $|vwx| \le 2^{|N|} < n_0$ (Eigenschaft (i)).

Die in *G* mögliche Ableitung $A \Rightarrow^* vAx$ kann man beliebig oft in die Ableitung $S \Rightarrow^* uvwxy$ einbauen und mit der Ableitung $A \Rightarrow^* w$ kombinieren:

$S \Rightarrow^* uAy \Rightarrow^* uvAxy \Rightarrow^* uvvAxxy \Rightarrow^* uv^k A x^k y \Rightarrow^* uv^k w x^k y$ für $k \ge 1$; außerdem ist die Ableitung $S \Rightarrow^* uAy \Rightarrow^* uwy$ möglich. Insgesamt ist dieses die Eigenschaft (iii). ///

Mit Hilfe dieses Satzes läßt sich zeigen, daß die Sprache $L_4 = \{a^n b^n c^n \mid n \in \mathbf{N}, n \ge 1\}$ (aus Kapitel 4.1) nicht kontextfrei (aber kontextsensitiv ist). Es sei $n = n_0$. Für das Wort $z = a^n b^n c^n$ ist $|z| = 3n_0 \ge n_0$. Dann läßt sich *z* zerlegen in $z = uvwxy$ mit den obigen Eigenschaften (i), (ii) und (iii). Der Teil *vwx* enthält höchstens n_0 viele Zeichen und kann daher nicht gleichzeitig aus *a*'s, *b*'s und *c*'s bestehen. Eigenschaft (iii) besagt, daß auch $uv^0 w x^0 y \in L_4$ ist, d.h. $uwy \in L_4$. Es werden drei Fälle unterschieden:

1. Fall: *vwx* enthält kein Zeichen *c*. Dann ist $uvwx = a^n b^m$, $y = b^{n-m} c^n$. Da $|vx| > 0$ ist, enthält vx $k \ge 1$ Zeichen *a* oder *b*. Das Wort *uwy* enthält $n + m - k + n - m = 2n - k < 2n$ Zeichen *a* oder *b* und *n* Zeichen *c*. Daher ist $uwy \notin L_4$.

2. Fall: *vwx* enthält mindestens ein Zeichen *c*. Wegen $|vwx| \le n_0$ enthält es kein Zeichen *a*. Es ist $u = a^n b^m$, $vwx = b^{n-m} c^l$, $y = c^{n-l}$ mit $l \ge 1$. Da $|vx| > 0$ ist, enthält *vx* $k \ge 1$ Zeichen *b* oder *c*. Das Wort *uwy* enthält n Zeichen *a*, $m + n - m + l - k = n + l - k$ Zeichen *b* oder *c* und $n - l$ Zeichen *c*. Daher ist $uwy \notin L_4$.

In beiden Fällen ergibt sich ein Widerspruch. Daher ist Sprache $L_4 = \{a^n b^n c^n \mid n \in \mathbf{N}, n \ge 1\}$ nicht kontextfrei.

Ein weiteres Beispiel ist die kontextsensitive Sprache $L = \{a^{2^i} \mid i \ge 1\}$.

Es gilt daher die in Satz 4.4-1 formulierte Aussage, daß die Klasse der kontextfreien Sprachen über einem endlichen Alphabet Σ eine echte Teilmenge der Klasse der kontextsensitiven Sprachen über Σ ist.

Exemplarisch für viele interessante Entscheidungsprobleme im Zusammenhang mit kontextfreien Grammatiken und kontextfreien Sprachen sollen wieder das Wortproblem und das Leerheitsproblem, jetzt bezogen auf kontextfreie Sprachen, betrachtet werden.

Wieder wird (analog zu den Typ-0- und Typ-1-Grammatiken) ein Prädikat
$VERIFIZIERE\_G\_TYP\_2(w)$

$$= \begin{cases} \mathsf{TRUE} & \text{falls } w \text{ die Kodierung einer Typ-2-Grammatik darstellt} \\ \mathsf{FALSE} & \text{falls } w \text{ nicht die Kodierung einer Typ-2-Grammatik darstellt} \end{cases}$$

definiert. Dieses Prädikat ist berechenbar. Dazu ist unter anderem zu prüfen, ob die Erzeugungsregeln in KG_w den Bedingungen einer kontextfreien Grammatik genügen.

Das **Wortproblem für Typ-2-Grammatiken** fragt danach, ob die Menge

$$L_{Wort\_Typ-2} = \left\{ u\#w \;\middle|\; u \in \Sigma^*, w \in \{0,1\}^*, VERIFIZIERE\_G\_TYP\_2(w) = \mathsf{TRUE} \text{ und } u \in L(KG_w) \right\}$$

entscheidbar ist. In vereinfachter Darstellung wird also danach gefragt, ob es einen auf allen Eingaben stoppenden Algorithmus gibt, der bei Eingabe (der Kodierung) einer kontextfreien Grammatik *G* über dem Alphabet Σ und eines Worts $u \in \Sigma^*$ entscheidet, ob $u \in L(G)$ gilt oder nicht. Da dieses Entscheidungsproblem für kontextsensitive Sprachen entscheidbar ist, ist es auch im kontextfreien Fall entscheidbar. Das Entscheidungsverfahren aus Kapitel 4.3 ist im Falle einer kontextfreien Grammatik jedoch zu aufwendig (exponentielles Laufzeitverhalten). In der Theorie des Compilerbaus, die sich intensiv mit der Frage beschäftigt, ob ein Wort zur Sprache einer gegebenen Grammatik gehört, wird gezeigt, daß die Frage sogar mit einem Zeitaufwand der Ordnung $O(|u|^3)$ bei gegebener Grammatik *G* entschieden werden kann. Auf Details soll hier verzichtet werden.

Das **Leerheitsproblem**, das für Typ-0- und Typ-1-Grammatiken nicht entscheidbar ist, fragt **für Typ-2-Grammatiken**, ob die Menge

$$L_{leer\_Typ-2} = \left\{ w \,\middle|\, w \in \{0,1\}^*, VERIFIZIERE\_G\_TYP\_2(w) = \mathsf{TRUE} \text{ und } L(KG_w) = \varnothing \right\}$$

entscheidbar ist, bzw. in vereinfachter Darstellung, ob es einen auf allen Eingaben stoppenden Algorithmus gibt, der bei Eingabe (der Kodierung) einer Grammatik G entscheidet, ob $L(G) = \varnothing$ gilt oder nicht.

Der folgende (Pseudocode-) Algorithmus entscheidet bei Eingabe einer kontextfreien Grammatik die $G = (\Sigma, N, S, R)$ die Frage „$L(G) \neq \varnothing$?“ . Aus diesem Algorithmus läßt sich leicht ein Entscheidungsalgorithmus für das Leerheitsproblem für Typ-2-Grammtiken gewinnen.

Eingabe: Eine kontextfreie Grammatik $G = (\Sigma, N, S, R)$

Verfahren: Aufruf der Funktion `ist_nichtleer (`G`)`

Ausgabe: `TRUE`, falls $L(G) \neq \varnothing$, `FALSE` sonst.

```
FUNCTION ist_nichtleer (G): BOOLEAN;
        { G = (Σ, N, S, R) }

VAR V : ...;
    W : ...;

BEGIN { ist_nichtleer }
  V := ∅;
  W := { A | A ∈ N und A → w ist eine Erzeugungsregel mit w ∈ Σ* };

  WHILE NOT (V = W) DO
    BEGIN
      V := W;
      W := { A | A ∈ N und A → w ist eine Erzeugungsregel mit w ∈ (V ∪ Σ)* } ∪ V ;
    END;

  IF S ∈ W THEN ist_nichtleer := TRUE
           ELSE ist_nichtleer := FALSE;
END   { ist_nichtleer };
```

Es läßt sich zeigen, daß die `WHILE`-Schleife höchstens *n*-mal durchlaufen wird, wenn *G n* Erzeugungsregeln enthält. Daher bricht das Verfahren ab. Die Korrektheit des Verfahrens ergibt sich aus der Behauptung

> Ein nichtterminales Symbol $A \in N$ wird im *i*-ten Durchlauf der `WHILE`-Schleife genau dann in `W` aufgenommen, wenn es ein $u \in \Sigma^*$ gibt mit $A \Rightarrow^* u$.

Zusammenfassend ergibt sich

Satz 4.4-4:

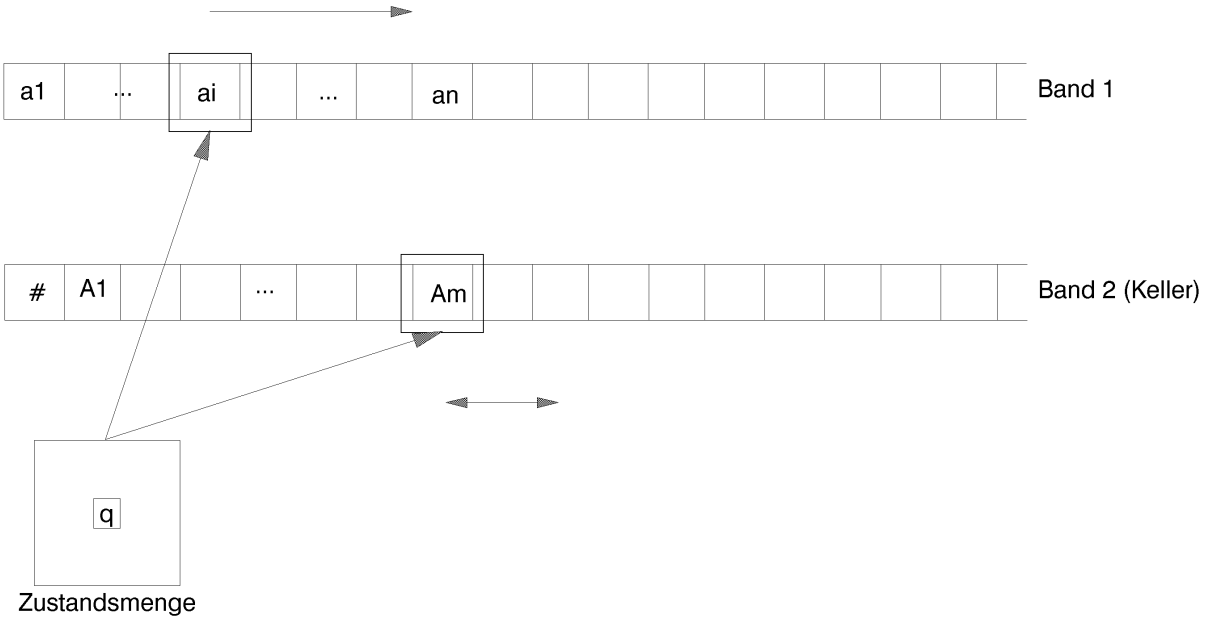
Das Wortproblem und das Leerheitsproblem für Typ-2-Grammatiken sind entscheidbar:

Es gibt einen auf allen Eingaben stoppenden Algorithmus**,** der bei Eingabe einer Typ-2-Grammatik $G = (\Sigma, N, S, R)$ und eines Wortes $u \in \Sigma^*$ entscheidet, ob $u \in L(G)$ ist.

Es gibt einen auf allen Eingaben stoppenden Algorithmus**,** der bei Eingabe einer Typ-2-Grammatik $G = (\Sigma, N, S, R)$ entscheidet, ob $L(G) = \varnothing$ gilt.

Auch zu den kontextfreien Sprachen gibt es ein Berechnungsmodell, das genau diese Sprachen erkennt und als eine Einschränkung einer Turingmaschine gesehen werden.

Ein **nichtdeterministischer Kellerautomat (NKA)** ist eine 2-NDTM, die ihre beiden Bänder in einer speziellen Weise nutzt. Vor der Angabe einer formalen Definition eines nichtdeterministischen Kellerautomaten wird das Modell informell beschrieben. Das 1. Band (Eingabeband) eines nichtdeterministischen Kellerautomaten *NKA* kann nur gelesen werden, wobei der Lesekopf nach dem Lesen eines Zeichens um eine Zelle nach rechts rückt. Das 1. Band enthält bei Start von *NKA* ein Wort über einem Eingabealphabet, der Lesekopf steht über dem ersten Zeichen des Eingabeworts. *NKA* kann auch tätig werden, wenn das Eingabeband eine leere Zeichenkette enthält; dann findet ein „spontaner (Konfigurations-) Übergang“ statt. Er kann auch Zustandsänderungen durchführen, ohne ein Symbol des Eingabebands zu lesen; man nennt diese Konfigurationsübergänge e -Überführungen. Das 2. Band heißt **Keller**; auf dem Keller bewegt sich ein Schreib/Lesekopf. Der Keller enthält zu Beginn der Berechnungsfolge von *NKA* ein spezielles Symbol #, und der Schreib/Lesekopf steht über diesem Symbol. Bei jeder Konfigurationsänderung wird das Symbol unter dem Schreib/Lesekopf durch ein endlich langes Wort ersetzt, das mit Hilfe eines Kelleralphabets gebildet wird, und der Schreib/Lesekopf rückt über das letzte Zeichen der nun im Keller befindlichen Zeichenkette. Es kann auch das leere Wort in den Keller geschrieben werden; dann wird der Inhalt des Kellers verkürzt. Auf diese Weise kann immer nur auf das zuletzt in den Keller gebrachte Zeichen zugegriffen werden. Der Schreib/Lesekopf im Keller rückt nicht über das Ende des Kellers auf Zeichen, die weiter links stehen (last-in-first-out-Prinzip).



Eine formale Definition eines nichtdeterministischen Kellerautomaten wandelt die Definition einer nichtdeterministischen Turingmaschine ab. Die Definition einer Konfiguration und des Konfigurationsübergangs wird der besonderen Arbeitsweise eines Kellerautomaten angepaßt. In der Literatur finden sich häufig Varianten der hier gegebenen Definitionen, die aber auf dieselbe Sprachklasse führen.

Ein **nichtdeterministischer Kellerautomat** *NKA* ist definiert durch
$NKA = (Q, \Sigma, \Gamma, d, q_0, \#, F)$ mit:

1. Q ist eine endliche nichtleere Menge: die **Zustandsmenge**
2. Σ ist eine endliche nichtleere Menge: das **Eingabealphabet**
3. Γ ist eine endliche nichtleere Menge: das **Kelleralphabet**
4. $d : Q \times (\Sigma \cup \{e\}) \times \Gamma \to \mathbf{P}_e(Q \times \Gamma^*)$ ist eine partielle Funktion, die **Überführungsfunktion**; hierbei bezeichnet $\mathbf{P}_e(Q \times \Gamma^*)$ die Menge aller endlichen Teilmengen von $Q \times \Gamma^*$
5. $q_0 \in Q$ ist der **Anfangszustand (**oder **Startzustand)**
6. $\# \in \Gamma$ ist das **Startsymbol im Keller**
7. $F \subseteq Q$ ist die **Menge der Endzustände**.

In jedem Schritt eines nichtdeterministischen Kellerautomaten ist klar, wo sich der Kopf des jeweiligen Bandes befindet: Wenn das Eingabewort *w* die Form $w = w_1 w_2$ hat und der Kellerautomat bereits die Zeichen in w_1 gelesen hat, dann steht der Lesekopf des 1. Bandes über

dem ersten Zeichen von w_2, und auf die Zeichen von w_1 kann der Kopf nicht mehr zugreifen. Anstelle daher in einer Konfiguration für das 1. Band den gesamten Bandinhalt und die Position des Kopfes in der Form $(w_1 w_2, |w_1|+1)$ festzuhalten, wird der Inhalt des 1. Bandes nur durch die Zeichenkette w_2 beschrieben; implizit wird angenommen, daß der Kopf über dem ersten Zeichen von w_2 steht. Entsprechend braucht man in einer Konfiguration für das 2. Band nicht den gesamten Bandinhalt und die Position des Kopfes in der Form $(a, |a|)$ festzuhalten, sondern es genügt die Zeichenkette a zusammen mit der impliziten Annahme, daß der Kopf über dem zuletzt in den Keller geschriebenen Zeichen steht. Daher wird **eine Konfiguration für einen nichtdeterministischen Kellerautomaten** in der Form (q, w, a) mit $q \in Q$, $w \in \Sigma^*$ und $a \in \Gamma^*$ notiert. Die Konfiguration drückt aus, daß sich der Kellerautomat gegenwärtig im Zustand *q* befindet, daß das Eingabeband eine Zeichenkette $w_1 w$ enthält, von der bisher die Zeichen in $w = a_1 \ldots a_n$ noch nicht gelesen wurden, daß der Lesekopf des 1. Bandes über dem ersten Zeichen a_1 steht, daß sich im Keller die Zeichenkette $a = A_1 \ldots A_m$ befindet, und daß der Schreib/Lesekopf des 2. Bandes über A_m steht. Falls $d(q, a_1, A_m)$ definiert ist (das ist eine endliche Teilmenge von $Q \times \Gamma^*$**)** und ein Element $(q', B_1 \ldots B_k)$ enthält, dann kann der Kellerautomat in eine **Nachfolgekonfiguration** übergehen, die den Zustand q' enthält, den Lesekopf auf dem 1. Band um eine Position nach rechts verrückt, im Keller das Zeichen A_m durch $B_1 \ldots B_k$ ersetzt und den Schreiblesekopf im Keller auf B_k positioniert hat. Formal wird für Konfigurationen die Übergangsrelation $\Rightarrow$ definiert durch

$$(q, a_1 \ldots a_n, A_1 \ldots A_m) \Rightarrow \begin{cases} (q', a_2 \ldots a_n, A_1 \ldots A_{m-1} B_1 \ldots B_k) & \text{falls } (q', B_1 \ldots B_k) \in d(q, a_1, A_m) \\ (q', a_1 a_2 \ldots a_n, A_1 \ldots A_{m-1} B_1 \ldots B_k) & \text{falls } (q', B_1 \ldots B_k) \in d(q, e, A_m) \end{cases}$$

Entsprechend bedeutet für Konfigurationen *K* und K' die Beziehung $K \Rightarrow^* K'$:
Es gibt Konfigurationen $K_0, K_1, \ldots, K_l$ mit $l \geq 0$ und $K = K_0$, $K' = K_l$ und $K_i \Rightarrow K_{i+1}$ für $i = 0, \ldots, l-1$.

Eine **Anfangskonfiguration für einen nichtdeterministischen Kellerautomaten** *NKA* hat die Form $(q_0, w, \#)$, eine **Endkonfiguration** hat die Form (q, e, e) mit $q \in F$.

Ein Wort $w \in \Sigma^*$ wird von *NKA* **akzeptiert**, wenn $(q_0, w, \#) \Rightarrow^* (q, e, e)$ mit $q \in F$ gilt. Die von *NKA* **akzeptierte Sprache** ist $L(NKA) = \{ w \mid w \in \Sigma^*, \text{und } w \text{ wird von } NKA \text{ akzeptiert} \}$.

Beispiel:

Ein nichtdeterministischer Kellerautomat zum Erkennen der Sprache

$$L = \{ w \mid w = a_1 ... a_m a_m ... a_1 \, , a_i \in \{a, b\}\}$$

Die Sprache $L = \{ w \mid w = a_1 ... a_m a_m ... a_1 \, , a_i \in \{a, b\}\}$ läßt sich beispielsweise durch die kontextfreie Grammatik $G = (\{a,b\}, \{S\}, S, \{S \to aSa, S \to bSb, S \to aa, S \to bb\})$ erzeugen.

Der nichtdeterministische Kellerautomaten *NKA* wird gegeben durch
$NKA = (\{q_0, q_1, q_2\}, \{a,b\}, \{A, B, \#\}, d, q_0, \#, \{q_2\})$ mit der durch folgende Tabelle festgelegten Überführungsfunktion d :

momentaner Zustand	Symbol unter dem Kopf auf		neuer Zustand	neues Wort im Keller
	Band 1	Keller		
q_0	a	X mit $X \in \{A, B, \#\}$	q_0	XA
	a	X mit $X \in \{A, B, \#\}$	q_1	XA
	b	X mit $X \in \{A, B, \#\}$	q_0	XB
	b	X mit $X \in \{A, B, \#\}$	q_1	XB
q_1	a	A	q_1	e
	b	B	q_1	e
	e	$\#$	q_2	e

Der Nichtdeterminismus wird in diesem Beispiel eingesetzt, um die Mitte eines Eingabewortes „zu raten“.

Auch im Modell des Kellerautomaten kann man Nichtdeterminismus und Determinismus unterscheiden:

Ein **deterministischer Kellerautomat** *DKA* ist wie ein nichtdeterministischer Kellerautomat definiert mit dem Unterschied, daß die Überführungsfunktion die Form
$d : Q \times (\Sigma \cup \{e\}) \times \Gamma \to Q \times \Gamma^*$
aufweist.

Beispiel:

Ein deterministischer Kellerautomat zum Erkennen der Sprache
$$L = \{a^n b^n \mid n \in \mathbf{N}\}$$

Die Sprache $L = \{a^n b^n \mid n \in \mathbf{N}\}$ läßt sich beispielsweise durch die kontextfreie Grammatik $G = (\{a,b\}, \{S\}, S, \{S \to aSb, S \to e\})$ erzeugen.

Der deterministische Kellerautomaten *DKA* wird gegeben durch
$DKA = (\{q_0, q_1, q_2\}, \{a,b\}, \{a,b,\#\}, d, q_0, \#, \{q_0\})$ mit der durch folgende Tabelle festgelegten Überführungsfunktion d :

momentaner Zustand	Symbol unter dem Kopf auf		neuer Zustand	neues Wort im Keller
	Band 1	Keller		
q_0	a	#	q_1	#a
	e	#	q_0	e
q_1	a	a	q_1	aa
	b	a	q_2	e
q_2	b	A	q_2	e
	e	#	q_0	e

Satz 4.4-5:
Die Klasse der von nichtdeterministischen Kellerautomaten akzeptierten Sprachen über einem endlichen Alphabet Σ ist mit der Klasse der kontextfreien Sprachen (Typ-2-Sprachen) über Σ identisch.

Beweis:

Es ist zu zeigen, daß es zu jeder von einem nichtdeterministischen Kellerautomaten *NKA* akzeptierten Sprache $L = L(NKA)$ eine kontextfreie Grammatik G_{NKA} gibt mit $L = L(G_{NKA})$. Die umgekehrte Richtung, nämlich der Nachweis, daß es zu jeder von einer kontextfreien Grammatik G erzeugten Sprache $L' = L(G)$ einen nichtdeterministischen Kellerautomaten NKA_G mit $L(NKA_G) = L'$ gibt, ist von größerer praktischer Relevanz, da hierdurch implizit gezeigt wird, wie man zu einer vorgegebenen Grammatik einen Algorithmus (hier in Form eines

Kellerautomaten) entwerfen kann, der die Wörter der von der Grammatik erzeugten Sprache erkennt. Diese Beweisrichtung soll daher skizziert werden.

Es sei $G=(\Sigma, N, S, R)$ eine kontextfreie Grammatik. Ein nichtdeterministischer Kellerautomat $NKA_G=(Q,\Sigma,\Gamma,d,q_0,\#,F)$ mit $L(NKA_G)=L(G)$ wird gegeben durch
die Zustandsmenge $Q=\{z\}$, das Kelleralphabet $\Gamma=N\cup\Sigma$, den Anfangszustand $q_0=z$, das Startsymbol im Keller $\#=S$, die Menge der Endzustände $F=\{z\}$ und die Überführungsfunktion d, die folgendermaßen definiert ist:

(i) Für jede Erzeugungsregel $A\to a$ aus R wird (z,a^R) in $d(z,e,A)$ aufgenommen; hierbei ist a^R die Konkatination der Buchstaben von a in umgekehrter Reihenfolge (Spiegelung von a)

(ii) Für jedes $a\in\Sigma$ wird (z,e) in $d(z,a,a)$ aufgenommen.

Wegen (i) kann immer dann, wenn das im Keller gelesene Symbol ein nichtterminales Zeichen ist, dieses durch die rechte Seite einer Regel (in gespiegelter Reihenfolge) ersetzt werden. Ein Terminalzeichen, das gerade im Keller gelesen wird, wird entfernt, wenn es mit dem nächsten Eingabesymbol übereinstimmt. Man kann $L(NKA_G)=L(G)$ zeigen. ///

Beispiel:

Ein nichtdeterministischer Kellerautomat für eine kontextfreie Grammatik

Gegeben sei die Grammatik $G=(\Sigma, N, S, R)$ mit $\Sigma=\{a,b,c\}$, $N=\{S,A\}$ und der Produktionsmenge $R=\{S\to A, A\to aAb, A\to c\}$. Die in der Konstruktion beschriebene Überführungsfunktion wird durch folgende Tabelle gegeben:

momentaner Zustand	Symbol unter dem Kopf auf		neuer Zustand	neues Wort im Keller
	Band 1	Keller		
z	e	S	z	A
	e	A	z	bAa
	e	A	z	c
	A	a	z	e
	B	b	z	e
	C	c	z	e

Das Wort $aacbb\in L(G)$ wird durch folgenden Konfigurationsübergänge akzeptiert:

$$(z, aacbb, S)\Rightarrow(z, aacbb, A)\Rightarrow(z, aacbb, bAa)\Rightarrow(z, acbb, bA)\Rightarrow(z, acbb, bbAa)$$
$$\Rightarrow(z, cbb, bbA)\Rightarrow(z, cbb, bbc)\Rightarrow(z, bb, bb)\Rightarrow(z, b, b)$$
$$\Rightarrow(z, e, e).$$

Jede von einem deterministischen Kellerautomaten akzeptierte Sprache wird auch von einem nichtdeterministischen Kellerautomaten akzeptiert. Umgekehrt gibt es Sprachen, etwa $L = \{ w \mid w = a_1 ... a_m a_m ... a_1 , a_i \in \{a, b\}\}$, die nicht von einem deterministischen Kellerautomaten akzeptiert werden können. Insbesondere weisen die Klassen der deterministisch kontextfreien und die Klasse der nichtdeterministisch kontextfreien Sprachen unterschiedliche Abschlußeigenschaften auf (siehe Zusammenstellung in Kapitel 4.6).

Satz 4.4-5:

Die Klasse der deterministisch kontextfreien Sprachen über einem endlichen Alphabet Σ ist eine echte Teilmenge der Klasse der nichtdeterministisch kontextfreien Sprachen Σ.

Die Klasse der deterministisch kontextfreien Sprachen spielt eine besondere Rolle im Compilerbau. Eine derartige Sprache erlaubt die Syntaxanalyse eines Wortes (in der Anwendung ist dieses ein Programm in einer Programmiersprache), indem nur wenige Zeichen des Wortes während des Analysevorgangs im Vorgriff gelesen werden, um festzustellen, welche Produktion in einer Ableitung als nächstes anzuwenden ist bzw. daß das Wort nicht zu der von der Grammatik erzeugten Sprache gehört. Für deterministisch kontextfreie Sprachen ist das Wortproblem für ein Wort *u* mit einem Algorithmus entscheidbar, der eine Zeitkomplexität der Ordnung $O(|u|)$ aufweist. Die meisten Programmiersprachen sind weitgehend deterministisch kontextfrei, so daß in der Praxis die Syntaxanalyse bei Programmiersprachen sehr effizient abläuft.

4.5 Typ-3-Sprachen

Es sei $G = (\Sigma, N, S, R)$ eine Grammatik, in der die Erzeugungsregeln
$R \subseteq \{ A \to w \mid A \in N \text{ und } w \in (N \cup \Sigma)^* \}$ folgende zusätzliche Bedingung erfüllen:
Alle Regeln haben die Form $A \to aB$ mit $A \in N$, $B \in N$, $a \in \Sigma$ oder $A \to e$.

Die von einer derartigen Grammatik erzeugte Sprache heißt **Typ-3-Sprache** oder **rechtslineare Sprache** oder **reguläre Sprache**. Die zugehörige Grammatik heißt **Typ-3-Grammatik** oder **rechtslineare Grammatik**.

Beispielsweise ist die Sprache $L_1 = \{ a^i b^j \mid i \in \mathbf{N}, j \in \mathbf{N} \}$ aus Kapitel 4.1 regulär (rechtslinear, Typ-3-Sprache), da sie durch die Grammatik

$G_1' = (\{a, b\}, \{S, B\}, S, \{S \to e, S \to aS, S \to bB, B \to bB, B \to e\})$ erzeugt wird.

Der Name „rechtslineare Sprache“ erklärt sich durch die Form der Erzeugungsregeln $A \to aB$ der zugehörigen Grammatik: Betrachtet man die Nichtterminalsymbole als „Variablen“ (im Sinne eines Gleichungssystems) und die Terminalsymbole als „Konstanten“, so liegt in der Form $A \to aB$ eine lineare Beziehung zwischen den Variablen vor, in der die Variablen rechts der Konstanten stehen.

Der Name „reguläre Sprache“ weist auf eine alternative Möglichkeit hin, um eine Typ-3-Sprache zu definieren: eine derartige Sprache läßt sich durch einen „regulären Ausdruck“ repräsentieren. Der Vollständigkeit soll hier die Definition eines regulären Ausdrucks und der durch ihn repräsentierten Sprache angeführt werden, wenn auch im folgenden auf dieses Konzept nicht weiter eingegangen wird.

Ein **regulärer Ausdruck über** Σ und **die durch ihn repräsentierte Sprache** wird rekursiv definiert durch:

(i) $\varnothing$ ist ein regulärer Ausdruck, der die reguläre Sprache $\varnothing$ repräsentiert

(ii) e ist ein regulärer Ausdruck, der die reguläre Sprache $\{e\}$ repräsentiert

(iii) a aus Σ ist ein regulärer Ausdruck, der die reguläre Sprache $\{a\}$ repräsentiert

(iv) Sind p bzw. q reguläre Ausdrücke, die die regulären Sprachen P bzw. Q repräsentieren, dann ist

$(p+q)$ ein regulärer Ausdruck, der die reguläre Sprache $P \cup Q$ repräsentiert,

(pq) ein regulärer Ausdruck, der die reguläre Sprache $P \cdot Q$ repräsentiert,

$(p)^*$ ein regulärer Ausdruck, der die reguläre Sprache P^* repräsentiert

(v) Ein regulärer Ausdruck und die von ihm repräsentierte Sprache wird genau durch die Punkte (i) bis (iv) definiert.

Folgender Satz läßt sich beweisen:

Satz 4.5-1:

Eine Sprache L über einem endlichen Alphabet Σ wird genau dann durch einen regulären Ausdruck repräsentiert, wenn es eine Typ-3-Grammatik (rechtslineare Grammatik) $G = (\Sigma, N, S, R)$ mit $L = L(G)$ gibt.

Im folgenden werden daher die Begriffe „Typ-3-Sprache“, „rechtslineare Sprache“ und „reguläre Sprache“ synonym verwendet.

Auch für die regulären Sprachen läßt sich ein Berechnungsmodell zur Akzeptanz gerade dieses Typs angeben. Man erhält es durch Einschränkung eines Kellerautomaten. Der Keller eines Kellerautomaten erlaubt die Zwischenablage von (evtl. transformierten) Eingabezeichen. Entfernt man den Keller, so kommt man auf das folgende Modell:

Ein **nichtdeterministischer endlicher Automat** *NEA* ist definiert durch
$NEA = (Q, \Sigma, d, q_0, F)$ mit:

1. Q ist eine endliche nichtleere Menge: die **Zustandsmenge**
2. Σ ist eine endliche nichtleere Menge: das **Eingabealphabet**
3. $d : Q \times \Sigma \to \mathbf{P}(Q)$ ist eine partielle Funktion, die **Überführungsfunktion**; hierbei bezeichnet $\mathbf{P}(Q)$ die Menge aller Teilmengen von Q.
4. $q_0 \in Q$ ist der **Anfangszustand (**oder **Startzustand)**
5. $F \subseteq Q$ ist die **Menge der Endzustände**.

Entsprechend liegt ein **deterministischer endlicher Automat** *DEA* vor, wenn die Überführungsfunktion d die Form $d : Q \times \Sigma \to Q$ aufweist.

Ein deterministischer bzw. nichtdeterministischer endlicher Automat läßt sich auch in Form eines endlichen gerichteten Graphen G mit markierten Kanten als **Transitionsdiagramm** darstellen. Die Knotenmenge von G ist Q. Ist $z' = d(z, a)$ bzw. im nichtdeterministischen Fall $z' \in d(z, a)$, so wird eine mit a markierte Kante in G aufgenommen. Die Menge der den Endzuständen entsprechenden Knoten werden explizit markiert.

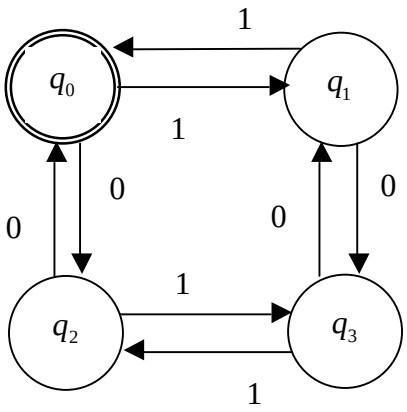
Beispiel:

Ein deterministischer endlicher Automat

Es sei $DEA = (Q, \Sigma, d, q_0, F)$ der deterministische endliche Automat mit $Q = \{ q_0, q_1, q_2, q_3 \}$, $\Sigma = \{ 0, 1 \}$, $F = \{ q_0 \}$, d gemäß folgender Tabelle:

momentaner Zustand	gelesenes Symbol	neuer Zustand
q_0	0	q_2
	1	q_1
q_1	0	q_3
	1	q_0
q_2	0	q_0
	1	q_3
q_3	0	q_1
	1	q_2

Das entsprechende Transitionsdiagramm ist



Es ist

$L(DEA) = \{ w \mid w \in \{0,1\}^* \text{ und } w \text{ enthält eine gerade Anzahl von Zeichen 0 und Zeichen 1} \}$.

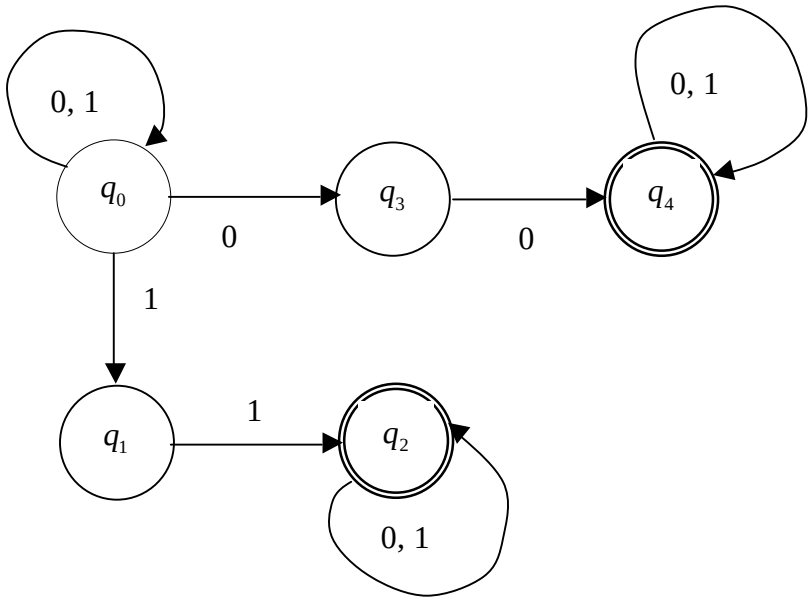
Beispiel:

Ein nichtdeterministischer endlicher Automat

$NEA = (Q, \Sigma, d, q_0, F)$ mit $Q = \{ q_0, q_1, q_2, q_3, q_4 \}$, $\Sigma = \{0,1\}$, $F = \{ q_2, q_4 \}$, d gemäß folgender Tabelle:

momentaner Zustand	gelesenes Symbol	neuer Zustand
q_0	0	q_0
		q_3
	1	q_0
		q_1
q_1	1	q_2
q_2	0	q_2
	1	q_2
q_3	0	q_4
q_4	0	q_4
	1	q_4

Das zugehörige Transitionsdiagramm lautet



Es ist

$$L(NEA) = \left\{ w \,\middle|\, \begin{array}{l} w \in \{0,1\}^+ \\ \text{und } w \text{ enthält mindestens zwei aufeinanderfolgende Zeichen 0 oder Zeichen 1} \end{array} \right\}.$$

Wie bei einem Kellerautomaten durch Weglassen des Kellers kann man auch hier Konfigurationen und Konfigurationsübergänge definieren. Ein Wort $w \in \Sigma^*$ wird von *NEA* (*DEA*) **akzeptiert**, wenn $(q_0, w) \Rightarrow^* (q, e)$ mit $q \in F$ gilt. Die von *NEA* (*DEA*) **akzeptierte Sprache** ist $L(NEA) = \{ w \mid w \in \Sigma^*, \text{ und } w \text{ wird von } NEA \text{ akzeptiert} \}$ (entsprechend für den deterministischen Fall).

Bei Turingmaschinen fallen Determinismus und Nichtdeterminismus zusammen, jedoch zum Preis einer exponentiell wachsenden Berechnungs- bzw. Laufzeitkomplexität. Der folgende Satz zeigt, daß auch bei den regulären Sprachen Nichtdeterminismus keine zusätzliche Berechnungsfähigkeit gegenüber dem Determinismus bringt. In den dazwischenliegenden Sprachklassen der kontextfreien und kontextsensitiven Sprachen ist der Determinismus vom Nichtdeterminismus zu unterscheiden bzw. der Unterschied zwischen Determinismus und Nichtdeterminismus ist nicht bekannt.

Satz 4.5-2:

Für jede von einem nichtdeterministischen endlichen Automaten akzeptierte Sprache über einem endlichen Alphabet Σ gibt es einen deterministischen endlichen Automaten, der die Sprache akzeptiert.

Beweis:

Es ist bei gegebenem nichtdetermistischen endlichen Automaten $NEA = (Q, \Sigma, d, q_0, F)$ ein determistischer endlicher Automat $DEA = (Q', \Sigma, d', q_0', F')$ anzugeben, der dieselbe Sprache akzeptiert. *DEA* wird definiert durch:

$Q' = \mathbf{P}(Q)$, $q_0' = \{q_0\}$, $d'(Z', a) = \bigcup_{z \in Z'} d(z, a)$ für $Z' \subseteq Q$ und $a \in \Sigma$,

$F' = \{ Z' \mid Z' \subseteq Q \text{ und } Z' \cap F \neq \varnothing \}$. ///

Der folgende Satz besagt, daß das Konzept der endlichen erkennenden Automaten das adäquate Modell für die regulären Mengen ist.

Satz 4.5-3:

Die Klasse der von (nichtdeterministischen bzw. deterministischen) endlichen Automaten akzeptierten Sprachen über einem endlichen Alphabet Σ ist mit der Klasse der regulären Sprachen (Typ-3-Sprachen) über Σ identisch.

Beweis:

Es sei $NEA = (Q', \Sigma, d', q_0', F')$ ein nichtdeterministischer endlicher Automat. Dann gibt es nach Satz 4.5-2 einen deterministischen endlichen Automaten $DEA = (Q, \Sigma, d, q_0, F)$ mit $L(DEA) = L(NEA)$. Es wird eine rechtslineare Grammatik $G = (\Sigma, N, S, R)$ angegeben, für die $L(G) = L(DEA)$ gilt:

$N = Q$, $S = q_0$, die Menge R der Regeln wird definiert durch:

R enthält genau dann eine Erzeugungsregel $q \to aq'$ mit $q \in Q$, $q' \in Q$ und $a \in \Sigma$, wenn $d(q,a) = q'$ ist; für $q \in F$ enthält R außerdem die Regel $q \to e$.

Es sei umgekehrt $G = (\Sigma, N, S, R)$ eine rechtslineare Grammatik. Dann erkennt der folgende Automat $NEA = (Q, \Sigma, d, q_0, F)$ die Sprache $L(G)$:

Es sei A ein nichtterminales Symbol mit $A \notin N \cup \Sigma$. Es ist

$Q = N \cup \{A\}$, $q_0 = S$, $F = \begin{cases} \{A\} & \text{für } e \notin L(G) \\ \{S, A\} & \text{für } e \in L(G) \end{cases}$, die Überführungsfunktion $d : Q \times (\Sigma \cup \{e\}) \to \mathbf{P}(Q)$ wird definiert durch:

$d(B,a)$ enthält genau dann C, wenn R die Regel $B \to aC$ enthält; für jede Regel der Form $B \to e$ wird $d(B,e) = \{A\}$ gesetzt. Der Automat $NEA = (Q, \Sigma, d, q_0, F)$ erfüllt noch nicht die Definition eines endlichen Automaten; denn diese läßt keine e-Übergänge zu, d.h. ein endlicher Automat macht keine Überführungen, in denen kein Eingabesymbol gelesen wird. Der hier definierte Automat läßt jedoch eventuell derartige Überführungen zu. Es läßt sich jedoch zeigen (siehe angegebene Literatur), daß man *NEA* so zu einem nichtdeterministischen endlichen Automaten $NEA' = (Q', \Sigma, d', q_0', F')$ modifizieren kann, so daß dessen Überführungsfunktion $d' : Q' \times \Sigma \to \mathbf{P}(Q')$ die Definition eines nichtdeterministischen endlichen Automaten erfüllt und $L(NEA') = L(G)$ gilt. ///

Auch für reguläre Sprachen gibt es einen dem *uvwxy*-Theorem entsprechenden Satz, der wieder ein Beweismittel liefert, mit dessen Hilfe man zeigen kann, daß eine Sprache nicht regulär ist:

Satz 4.5-4:

Zu jeder regulären (Typ-3-, rechtslinearen) Sprache L gibt es eine Zahl $n_0 > 0$ mit der Eigenschaft:

jedes $z \in L$ mit $|z| \geq n_0$ läßt sich zerlegen in $z = uvw$ mit

(i) $|uv| \leq n_0$

(ii) $|v| > 0$

(iii) $uv^k w \in L$ für jedes $k \in \mathbf{N}$.

Beweis:

Die Argumentation kann ähnlich über die Ableitung eines Wortes mit Hilfe einer regulären Grammatik verlaufen wie im Beweis des *uvwxy*-Theorems. Einfacher geht es hier über die Erkennung eines Wortes mit Hilfe eines deterministischen endlichen Automaten:

Es sei regulären Sprache L und *DEA* ein deterministischer endlicher Automat mit $L(DEA) = L$. *DEA* enthalte n_0 viele Zustände. Bei der Akzeptanz eines Wortes $z \in L$ mit $|z| \geq n_0$ durchläuft *DEA* einschließlich des Anfangszustands $|z|+1$ viele Zustände. Hierbei geht wesentlich ein, daß *DEA* bei jeder Überführung ein Eingabesymbol liest. In der Folge der Zustandsüberführungen vom Anfangszustand q_0 bis zu einem akzeptierenden Zustand $q_{accept} \in F$ muß sich also ein Zustand q_i wiederholen. Es sei *u* das Anfangsteilwort von *z*, das in den Überführungen gelesen wurde, die *DEA* von q_0 aus bis zum ersten Erreichen von q_i gemacht hat. Das Teilwort *v* von *z* besteht aus den Symbolen, die *DEA* dann von q_i aus bis zum nächsten Erreichen von q_i gelesen hat. Schließlich ist *w* das Teilwort von *z*, das *DEA* liest, bis der akzeptierende Zustand q_{accept} erreicht ist. Mit dieser Zerlegung von *z* gelten die angegebenen drei Eigenschaften. ///

Mit Hilfe dieses Satzes läßt sich zeigen, daß die Menge der Palindrome $L = \{ w \mid w \in \{0,1\}^* \text{ und } w = a_1 ... a_{n-1} a_n = a_n a_{n-1} ... a_1 \} \cup \{e\}$ nicht regulär ist; L ist jedoch kontextfrei. Es sei $n_0 > 0$ der Wert aus obigem Satz und $n \geq n_0$ die kleinste gerade Zahl $\geq n_0$. Das Wort $z = \underbrace{1010...10}_{n \text{ Zeichen}} \underbrace{0101...01}_{n \text{ Zeichen}}$ liegt in L und läßt sich in der Form $z = uvw$ mit den Eigenschaften (i), (ii) und (iii) zerlegen. Zu beachten ist, daß die einzige Stelle, an der zwei gleiche Zeichen (00) aufeinanderfolgen, in der Mitte von *z* liegt. Das Teilwort *uv* ist Teil von $\underbrace{1010...10}_{n \text{ Zeichen}}$.

1. Fall: $v = 1$: Das Wort uv^2w ist in L und enthält zwei aufeinanderfolgende Zeichen 1. Damit uv^2w ein Palindrom ist müßten diese beiden Zeichen 1 jedoch in der Mitte des Worts liegen, da weiter rechts und links keine aufeinanderfolgenden Zeichen 1 stehen. Rechts der beiden Zeichen 1 gibt es noch die beiden Zeichen 0 (die ursprüngliche Mitte des Worts). Zu ihnen korrespondieren links der beiden Zeichen 1 keine entsprechende Zeichenfolge 00. Daher ist $uv^2w \notin L$.
2. Fall: $v = 0$: Das Wort uv^3w ist in L und enthält drei aufeinanderfolgende Zeichen 0. Damit uv^3w ein Palindrom ist müßten diese drei Zeichen 0 jedoch in der Mitte des Worts liegen, da weiter rechts und links keine aufeinanderfolgenden drei Zeichen 0 stehen. Zur ursprünglichen Mitte des Worts korrespondieren links der drei Zeichen 0 keine entsprechende Zeichenfolge 00. Daher ist $uv^3w \notin L$.
3. Fall: $|v| > 1$ und $v = 10...1$: Dann enthält Wort uv^2w zwei aufeinanderfolgende Zeichen 1 und man kann wie im 1. Fall argumentieren.
4. Fall: $|v| > 1$ und $v = 10...10$: Dann enthält Wort uv^2w zwei aufeinanderfolgende Zeichen 0 (die ursprüngliche Mitte) und links davon nur Teilzeichenketten 01 als rechts, und zwar mehr als rechts. Daher ist $uv^2w \notin L$.

In allen Fällen ergibt sich ein Widerspruch.

Die Regeln einer rechtslinearen Grammatik erfüllen die Bedingungen, die an die Regeln einer kontextfreien Grammatik gestellt werden. Jede von einer rechtslinearen Grammatik erzeugte Sprache ist daher auch eine kontextfreie Sprache. Die Sätze 4.5-2 und 4.5-3 implizieren, daß es zu jeder regulären Sprache über einem endlichen Alphabet Σ einen deterministischen Kellerautomaten gibt, der die Sprache akzeptiert. Andererseits läßt sich mit Satz 4.5-4 zeigen, daß die deterministisch kontextfreie Sprache $L = \{a^n b^n \mid n \in \mathbf{N}\}$ nicht regulär ist. Es gilt daher:

Satz 4.5-5:

Die Klasse der regulären (Typ-3-, rechtslinearen) Sprachen über einem endlichen Alphabet Σ ist eine echte Teilmenge der deterministisch kontextfreien Sprachen über Σ.

Auch für Typ-3-Sprachen gibt es wieder viele interessante Entscheidungsprobleme. Hier sollen jedoch wieder nur das Wortproblem und das Leerheitsproblem, jetzt bezogen auf Typ-3-Sprachen, betrachtet werden. Dazu wird (analog zu den Typ-0-, Typ-1- und Typ-2-Grammatiken) ein Prädikat

$VERIFIZIERE\_G\_TYP\_3(w)$

$$= \begin{cases} \mathsf{TRUE} & \text{falls } w \text{ die Kodierung einer Typ-3-Grammatik darstellt} \\ \mathsf{FALSE} & \text{falls } w \text{ nicht die Kodierung einer Typ-3-Grammatik darstellt} \end{cases}$$

definiert. Dieses Prädikat ist berechenbar. Dazu ist unter anderem zu prüfen, ob die Erzeugungsregeln in KG_w den Bedingungen einer Typ-3-Grammatik genügen.

Das **Wortproblem für Typ-3-Grammatiken** fragt danach, ob die Menge

$$L_{Wort\_Typ-3} = \left\{ u\#w \;\middle|\; u \in \Sigma^*, w \in \{0,1\}^*, VERIFIZIERE\_G\_TYP\_3(w) = \text{TRUE und } u \in L(KG_w) \right\}$$

entscheidbar ist. In vereinfachter Darstellung wird also danach gefragt, ob es einen auf allen Eingaben stoppenden Algorithmus gibt, der bei Eingabe (der Kodierung) einer rechtslinearen Grammatik G über dem Alphabet Σ und eines Worts $u \in \Sigma^*$ entscheidet, ob $u \in L(G)$ gilt oder nicht. Da dieses Problem für Typ-2-Grammatiken entscheidbar ist, gilt dieses auch für Typ-3-Grammatiken. In diesem Fall bietet sich jedoch ein einfacher Algorithmus an: Anstelle der Überprüfung $u \in L(G)$ wird untersucht, ob $u \in L(DEA)$ gilt, wobei $DEA = (Q, \Sigma, d, q_0, F)$ ein deterministischer endlicher Automat mit $L(DEA) = L(G)$ ist:

Eingabe: Ein deterministischer endlicher Automat $DEA = (Q, \Sigma, d, q_0, F)$ und ein Wort $u \in \Sigma^*$

Verfahren: Aufruf der Funktion `accept (`*DEA*, *u*`)`

Ausgabe: `TRUE`, falls $u \in L(G)$, `FALSE` sonst.

```
FUNCTION accept (DEA : ...;
                 u : STRING): BOOLEAN;
        { DEA = (Q, Σ, d, q0, F),
          u   = a1 ... an       }

VAR q : ...;
    v : STRING;
    a : CHAR;
    i : INTEGER;

BEGIN { accept }
  q := q0;
  v := u;

  FOR i := 1 TO Length(u) DO
    BEGIN
      a := Copy (v, 1, 1);
      Delete (v, 1, 1);
      q := d(q, a);
    END;
```

```
  IF q∈ F  THEN accept := TRUE
           ELSE accept := FALSE;

END   { accept };
```

Diese Algorithmus hat eine Laufzeit der Ordnung $O(|u|)$, d.h. lineare Laufzeit.

Das **Leerheitsproblem** fragt **für Typ-3-Grammatiken**, ob die Menge

$L_{leer\_Typ-3} = \{ w \mid w \in \{0,1\}^*, VERIFIZIERE\_G\_TYP\_3(w) = \text{TRUE und } L(KG_w) = \varnothing \}$

entscheidbar ist. Diese Frage ist natürlich positiv zu beantworten, da das Leerheitsproblem für Typ-2-Grammatiken entscheidbar ist. Im Fall einer regulären L Menge kann man mit Satz 4.5-4 argumentieren. Es sei n_0 die in Satz 4.5-4 angegebene natürliche Zahl. Dann gilt:

$L \neq \varnothing$ genau dann, wenn es ein Wort $u \in L$ gibt mit $|u| < n_0$

Eine Überprüfung der Frage „$L = \varnothing$?“ kann also so ablaufen, daß man alle Wörter $u \in \Sigma^*$ mit $|u| < n_0$ darauf hin überprüft, ob $u \in L$ gilt (Wortproblem).

4.6 Eigenschaften im tabellarischen Überblick

Die folgenden Tabellen stellen wichtige Eigenschaften der behandelten Sprachklassen zusammen. Nicht alle aufgeführten Eigenschaften wurden in den vorherigen Kapiteln bewiesen; es wird vielmehr auf die angegebene Literatur verwiesen.

Beschreibungsmittel	
Typ 0	Typ-0-Grammatik Turingmaschine
Typ 1	kontextsensitive Grammatik linear beschränkter Automat
Typ 2	kontextfreie Grammatik Kellerautomat
deterministisch kontextfrei	$LR(k)$-Grammatik deterministischer Kellerautomat
Typ 3	regulärer Ausdruck, rechtslineare Grammatik endlicher Automat

Sind das nichtdeterministische und das deterministische Modell äquivalent?	
Turingmaschine	ja
Linear beschränkter Automat	?
Kellerautomat	nein
Endlicher Automat	ja

Abschlußeigenschaften					
Operation	Schnitt $L_1 \cap L_2$	Vereinigung $L_1 \cup L_2$	Komplement $\Sigma^* \setminus L$	Produkt $L_1 \cdot L_2$	Stern L^*
Typ 0	ja	ja	nein	ja	ja
Typ 1	ja	ja	ja	ja	ja
Typ 2	nein	ja	nein	ja	ja
det. kontextfrei	nein	nein	ja	nein	nein
Typ 3	ja	ja	ja	ja	ja

Wortproblem	
Typ 0	unlösbar
Typ 1	lösbar mit Komplexität der Ordnung $O\left(2^{O(n)}\right)$
Typ 2 (bei gegebener kfr. Grammatik)	lösbar mit Komplexität der Ordnung $O\left(n^3\right)$
deterministisch kontextfrei	lösbar mit Komplexität der Ordnung $O(n)$
Typ 3	lösbar mit Komplexität der Ordnung $O(n)$

Leerheitsproblem	
Typ 0	unlösbar
Typ 1	unlösbar
Typ 2 (bei gegebener kfr. Grammatik)	lösbar
deterministisch kontextfrei	lösbar
Typ 3	lösbar

5 Praktische Berechenbarkeit

Im vorliegenden Kapitel geht es um die Lösung von Problemen „aus der Praxis“, insbesondere um die Untersuchung des Aufwandes, den diese Lösungen erfordern. **Alle hier behandelten (Entscheidungs-) Probleme sind entscheidbar, d.h. es gibt jeweils einen Algorithmus, der bei jeder Eingabe mit einer akzeptierenden oder verwerfenden Entscheidung stoppt**. Wie ein derartiger Algorithmus formuliert wird, ob als deterministische oder nichtdeterministische Turingmaschine, RAM oder Programm in einer Programmiersprache, ergibt sich jeweils aus dem Zusammenhang; das für die Darstellung angemessene Modell wird verwendet.

Der Begriff der Entscheidbarkeit bezieht sich auf Entscheidungsprobleme, in der Praxis hat man jedoch häufig mit Optimierungsproblemen zu tun. Beispielsweise wurde in Kapitel 1.2 das Problem des Handlungsreisenden als Optimierungsproblem, Entscheidungsproblem und Berechnungsproblem beschrieben. Der Zusammenhang zwischen diesen Problemtypen wird in Kapitel 5.2 genauer betrachtet.

Im folgenden Kapitel 5.1 werden zunächst zwei Beispiele für Optimierungsprobleme angeführt, die sich auf Graphen beziehen. Für beide Probleme werden Lösungsalgorithmen angegeben, die sich in ihrer Laufzeit wesentlich unterscheiden. Die Beispiele zeigen, daß es „ähnlich“ lautende Graphenprobleme gibt, die sich in ihrer Komplexität trotz gleicher Problemdarstellungsmethode deutlich unterscheiden.

5.1 Zwei Graphenprobleme

Das Problem der Wege mit minimalem Gewicht in gerichteten Graphen

Instanz: $G = (V, E, w)$

$G = (V, E, w)$ ist ein gewichteter gerichteter Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

Lösung: Die minimalen Gewichte d_i der Wege vom Knoten v_1 zu allen anderen Knoten v_i des Graphs für $i = 1, ..., n$.

Lösung: Die minimalen Gewichte d_i der Wege vom Knoten v_1 zu allen anderen Knoten v_i des Graphs für $i = 1, ..., n$.

Zur algorithmischen Behandlung wird der Graph $G = (V, E, w)$ in Form seiner **Adjazenzmatrix** $A(G)$ gespeichert. Diese ist definiert durch

$$A(G) = A_{(n,n)} = \lfloor a_{i,j} \rfloor_{(n,n)} \text{ mit } a_{i,j} = \begin{cases} w((v_i, v_j)) & \text{falls } (v_i, v_j) \in E \\ \infty & \text{sonst} \end{cases}.$$

Es werden folgende Datentypen verwendet:

```
CONST n         = ...;       { Problemgröße }
      unendlich = ...;       { hier kann der maximal mögliche
                               REAL-Wert verwendet werden       }

TYPE  knotenbereich = 1..n;
      matrix        = ARRAY [knotenbereich, knotenbereich] OF REAL;
      bit_feld      = ARRAY [knotenbereich] OF BOOLEAN;
      feld          = ARRAY [knotenbereich] OF REAL;
```

Die folgende Funktion wird zur Ermittlung des Minimums zweier REAL-Zahlen eingesetzt:

```
FUNCTION min (a, b: REAL):REAL;

BEGIN { min }
  IF a <= b THEN min := a
            ELSE min := b;
END   { min };
```

Der folgende Algorithmus zur Lösung des Problems der Wege mit minimalem Gewicht in gerichteten Graphen ist ein Beispiel für ein allgemeines Lösungsprinzip: die **Greedy-Methode**. Eine (global) optimale Lösung wird schrittweise gefunden, indem Entscheidungen gefällt werden, die auf „lokalen“ Informationen beruhen. Für eine Entscheidung wird hierbei nur ein kleiner (lokaler) Teil der Informationen genutzt, die über das gesamte Problem zur Verfügung stehen. Es wird die in der jeweiligen Situation optimal erscheinende Entscheidung getroffen, unabhängig von vorhergehenden und nachfolgenden Entscheidungen. Die einmal getroffenen Entscheidungen werden im Laufe des Rechenprozesses nicht mehr revidiert. Diese Methode ist jedoch nicht auf alle Optimierungsprobleme anwendbar.

Zunächst gelten alle Knoten v_i bis auf den Knoten v_1 als „nicht behandelt“. Aus den nicht behandelten Knoten wird ein Knoten ausgewählt, der zu einer bisherigen Teillösung hinzugenommen wird, und zwar so, daß dadurch eine bzgl. der Teillösung, d.h. bzgl. der behandelten Knoten, optimale Lösung entsteht.

Eingabe: $G = (V, E, w)$ ein gewichteter gerichteter Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht

```
VAR A       : matrix;        { Adjazenzmatrix }
    i       : knotenbereich;
    j       : knotenbereich;
    distanz : feld;

FOR i := 1 TO n DO
  FOR j := 1 TO n DO
    IF (v_i, v_j) ∈ E THEN A[i, j] := w((v_i, v_j))
                      ELSE A[i, j] := unendlich;
```

Verfahren: Aufruf der Prozedur `minimale_wege (A, distanz);`

Ausgabe: $d_i =$ `distanz[i]` für $i = 1, ..., n$.

```
PROCEDURE minimale_wege ( A           : matrix;
                          VAR distanz : feld);

VAR rest_knoten : bit_feld;      { Kennzeichnung der noch nicht
                                   behandelten Knoten: ist v_i ein
                                   nicht behandelter Knoten, so
                                   ist rest_knoten[i] = TRUE      }
    idx         : knoten_bereich;
    u           : knoten_bereich;
    exist       : BOOLEAN;

   PROCEDURE auswahl (VAR knoten   : knoten_bereich;
                      VAR gefunden : BOOLEAN);

     { die Prozedur wählt unter den noch nicht behandelten
       Knoten einen Knoten mit minimalem distanz-Wert aus
      (in knoten); gibt es einen derartigen Knoten, dann ist
       gefunden = TRUE, sonst ist gefunden = FALSE               }

       VAR  i   : knoten_bereich;
            min : REAL;

      BEGIN { auswahl }
        gefunden := FALSE;
        min      := unendlich;

        FOR i := 1 TO n DO
```

```
        IF rest_knoten [i]
        THEN BEGIN
               IF distanz [i] < min
               THEN BEGIN
                      gefunden := TRUE;
                      knoten   := i;
                      min      := distanz [i]
                    END
             END
     END { auswahl };


BEGIN { minimale_wege }
  { Gewichte zum Startknoten initialisieren und alle Knoten,
    bis auf den Startknoten, als nicht behandelt kennzeichnen: }
   FOR idx := 1 TO n DO
     BEGIN
       distanz[idx]     := A[1, idx];
       rest_knoten[idx] := TRUE
     END;
   distanz[1]     := 0;
   rest_knoten[1] := FALSE;

   { einen Knoten mit minimalem Distanzwert aus den Restknoten
     auswählen:                                                 }
   auswahl (u, exist);

   WHILE exist DO
     BEGIN
       { den gefundenen Knoten aus den noch nicht behandelten
          Knoten herausnehmen:                                  }
       rest_knoten[u] := FALSE;

       { für jeden noch nicht behandelten Knoten idx, der mit
         dem Knoten u verbunden ist, den distanz-Wert auf den
         neuesten Stand bringen:                                }
       FOR idx := 1 TO n DO
         IF rest_knoten[idx]
            AND
            (A[u, idx] <> unendlich)
         THEN distanz [idx] := min (distanz[idx],
                                    distanz[u] + A[u, idx]);

       { einen neuen Knoten mit minimalem Distanzwert aus den
         Restknoten auswählen:                                  }
       auswahl (u, exist);
```

```
        END  {WHILE exist }

   END { minimale_wege };
```

Es gilt bei Eintritt in die `WHILE exist DO`-Schleife folgende Schleifeninvariante:

(1) Ist der Knoten v_i ein bereits behandelter Knoten, d.h. `rest_knoten[i]=FALSE`, dann ist `distanz[i]` das minimale Gewicht eines Weges von v_1 nach v_i, der nur behandelte Knoten v_j enthält, für die also `rest_knoten[j]=FALSE` ist.

(2) Ist der Knoten v_i ein nicht behandelter Knoten, d.h. `rest_knoten[i]=TRUE`, dann ist `distanz[i]` das minimale Gewicht eines Weges von v_1 nach v_i, der bis auf v_i nur behandelte Knoten v_j enthält, für die also `rest_knoten[j]=FALSE` ist.

Aus der Schleifeninvariante folgt die Korrektheit des Algorithmus.

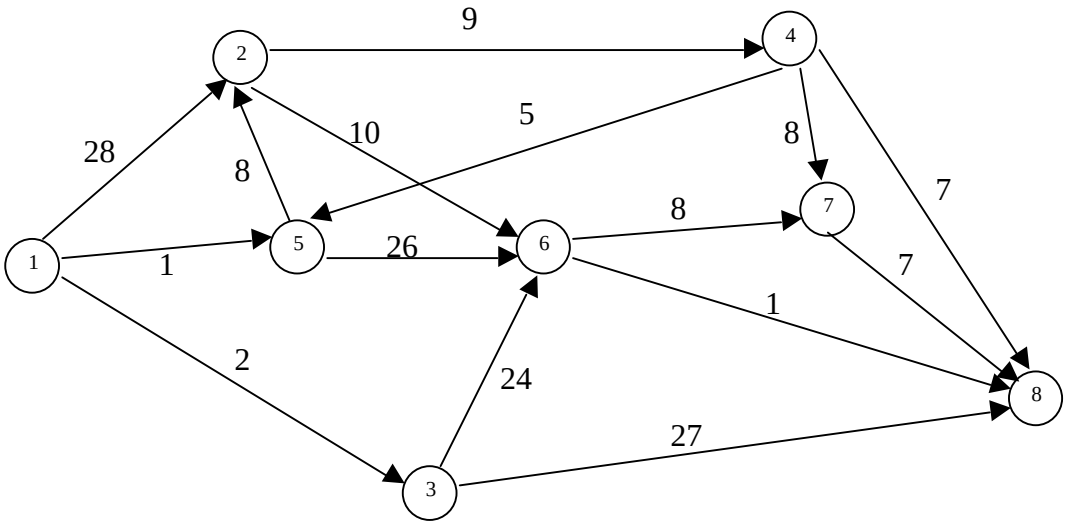
Satz 5.1-1:

Ist $G=(V,E,w)$ eine Instanz des Problems der Wege mit minimalem Gewicht in gerichteten Graphen mit *n* Knoten, dann liefert das beschriebene Verfahren mit der Prozedur `minimale_wege` im Feld `distanz` die minimalen Gewichte d_i der Wege vom Knoten v_1 zu allen anderen Knoten v_i des Graphs für *i* = 1, ..., *n*. Die (worst-case-) Zeitkomplexität des Verfahrens ist von der Ordnung $O(n^2)$.

Bemerkung: Die Laufzeit des angegebenen Verfahrens hängt im wesentlichen von der Anzahl der Konten des Graphen ab. Es kommen jedoch auch arithmetische Operationen vor, so daß es sich anbietet, die Bitoperationen zu zählen: Für eine Eingabeinstanz $G=(V,E,w)$ ist $size(G)=k=n\cdot A$ mit $A=\max\{\lfloor\log_2(|w(e)|)\rfloor+1 \mid e\in E\}$ die Anzahl der Bits, um die Eingabeinstanz darzustellen (für $w(e)=0$ wird anstelle von $\lfloor\log_2(|w(e)|)\rfloor+1$ der Wert 1 genommen). Eine genauere Untersuchung zeigt, daß die Laufzeit weiterhin in der Ordnung $O(k^2)$ bleibt.

Beispiel:

Die Instanz $G=(V,E,w)$ sei gegeben durch folgende graphische Darstellung:



Die Adjazenzmatrix zu *G* lautet

$$A(G) = \begin{bmatrix} \infty & 28 & 2 & \infty & 1 & \infty & \infty & \infty \\ \infty & \infty & \infty & 9 & \infty & 10 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 24 & \infty & 27 \\ \infty & \infty & \infty & \infty & 5 & \infty & 8 & 7 \\ \infty & 8 & \infty & \infty & \infty & 26 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 8 & 1 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 7 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

u wird jeweils beim Aufruf der Prozedur `auswahl (u, exist)` ausgewählt:

u	nichtbehandelte Knoten	`distanz [1]`	`distanz [2]`	`distanz [3]`	`distanz [4]`	`distanz [5]`	`distanz [6]`	`distanz [7]`	`distanz [8]`
	2 3 4 5 6 7 8	---	28	2	∞	1	∞	∞	∞
5	2 3 4 6 7 8		9	2	∞	---	27	∞	∞
3	2 4 6 7 8		9	---	∞		26	∞	29
2	4 6 7 8		---		18		19	∞	∞
4	6 7 8				---		19	26	25
6	7 8						---	26	20
8	7							26	---
7	leer							---	

Das zweite Besipiel löst das Handlungsreisenden-Optimierungsproblem nach der sogenannten **Branch-and-Bound-Methode**. Potentielle, aber nicht unbedingt optimale Lösungen werden systematisch erzeugt. Diese werden in Teilmengen aufgeteilt, die sich auf Knoten eines Ent-

scheidungsbaums abbilden lassen. Es wird eine Abschätzung für das Optimum mitgeführt und während des Verfahrensverlaufs laufend aktualisiert. Potentielle Lösungen, deren Zielfunktion einen „weit von der Abschätzung entfernten Wert“ aufweisen, werden nicht weiter betrachtet, d.h. der Teilbaum, der bei einer derartigen Lösung beginnt, muß nicht weiter durchlaufen werden.

Problem des Handlungsreisenden auf Graphen als Optimierungsproblem

Instanz: $G = (V, E, w)$

$G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

Lösung: Eine **Tour** durch *G*, d.h. eine geschlossene Kantenfolge

$\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ mit $(v_{i_j}, v_{i_{j+1}}) \in E$ für $j = 1, ..., n-1$,

in der jeder Knoten des Graphen (als Anfangsknoten einer Kante) genau einmal vorkommt, die minimale Kosten (unter allen möglichen Touren durch *G*) verursacht. Man kann o.B.d.A. $v_{i_1} = v_1$ setzen.

Die Kosten einer Tour $\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ ist dabei die Summe der Gewichte der Kanten auf der Tour, d.h. der Ausdruck

$$\sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}}) + w(v_{i_n}, v_{i_1}).$$

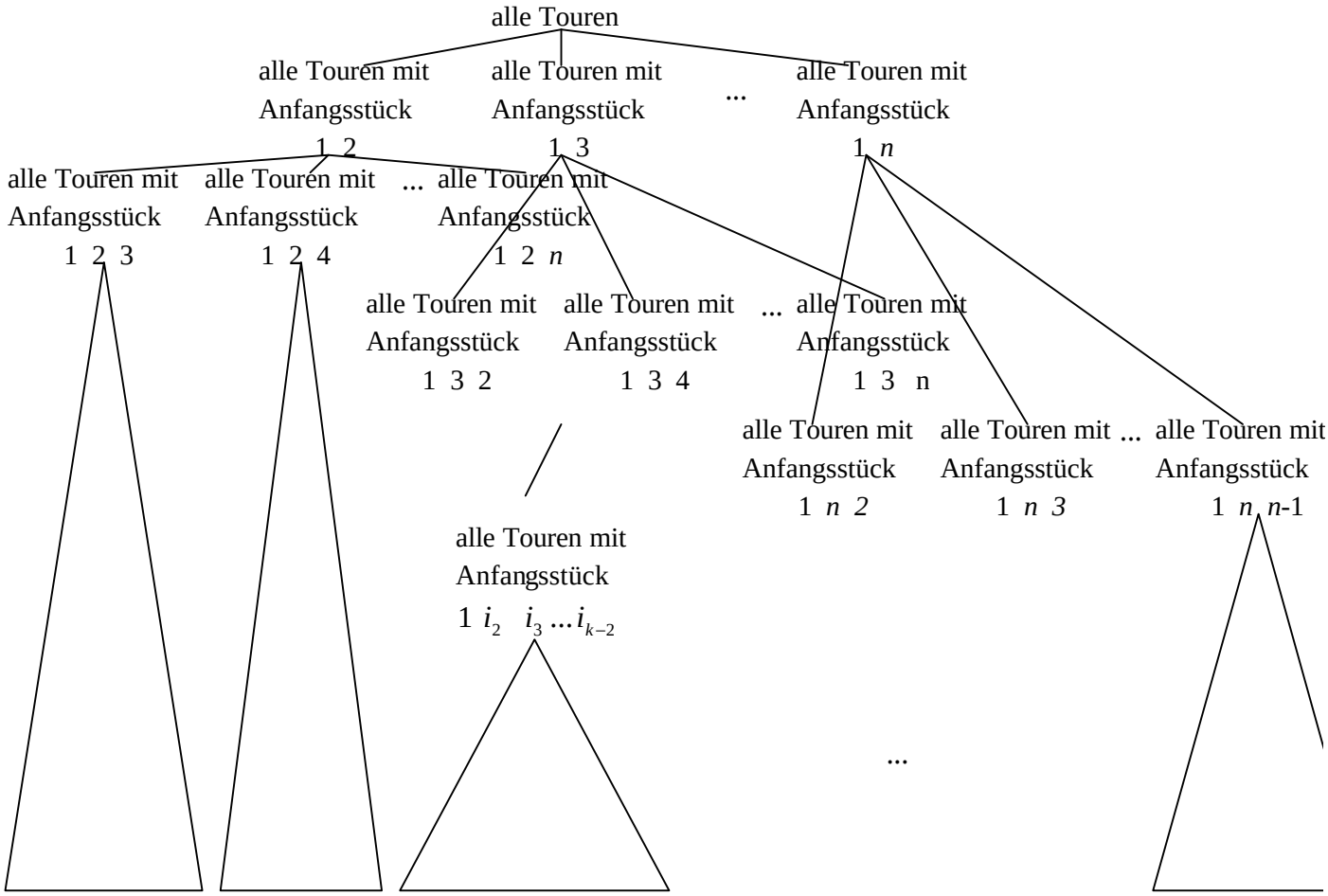
Es wird angenommen, daß eine Tour bei v_1 beginnt und endet. Außerdem wird angenommen, daß der Graph **vollständig** ist, d.h. daß $w((v_i, v_j))$ für jedes $v_i \in V$ und $v_j \in V$ definiert ist (eventuell ist $w((v_i, v_j)) = \infty$).

Im folgenden wird (zur Vereinfachung der Darstellung) die Knotenmenge $V = \{v_1, ..., v_n\}$ mit der Menge der Zahlen $\{1, ..., n\}$ gleichgesetzt (dem Knoten v_i entspricht die Zahl *i*). Eine Tour durch *G* läßt sich dann als Zahlenfolge

$\langle 1\ i_1\ i_2 \ldots i_{n-1}\ 1 \rangle$

darstellen, wobei alle i_j paarweise verschieden (und verschieden von 1) sind. Alle Touren erhält man, wenn man für i_j in $\langle 1\ i_1\ i_2 \ldots i_{n-1}\ 1 \rangle$ alle $(n-1)!$ Permutationen der Zahlen 2, ..., *n* einsetzt. Manche dieser Touren haben eventuell das Gewicht ∞.

Alle Touren (Zahlenfolgen der beschriebenen Art) lassen sich als Blätter eines Baums darstellen:



Alle Touren (Zahlenfolgen der beschriebenen Art) werden systematisch erzeugt (siehe unten). Dabei wird eine obere Abschätzung für das Gewicht einer optimalen Tour (Tour mit minimalem Gewicht) in der globalen Variablen `bound` mitgeführt (Anfangswert `bound` = ∞). Wird ein Anfangsstück einer Tour erzeugt, deren Gewicht größer als der Wert in der Variablen `bound` ist, dann brauchen alle Touren, die mit diesem Anfangsstück beginnen, nicht weiter betrachtet zu werden. Es wird also ein ganzer Teilbaum aus dem Baum aller Touren abgeschnitten. Ist schließlich eine Tour gefunden, deren Gewicht kleiner oder gleich dem Wert in der Variablen `bound` ist, so erhält die Variable `bound` als neuen Wert dieses Gewicht, und die Tour wird als temporär optimale Tour vermerkt. Eine Variable `opttour` nimmt dabei die Knotennummern einer temporär optimalen Tour auf; die Variable `teilfeld` dient der Aufnahme des Anfangsstücks einer Tour.

Ist bereits ein Anfangsstück $\langle 1\ i_1\ i_2 \ldots i_{k-1} \rangle$ einer Tour erzeugt, so sind die Zahlen $1, i_1, i_2, \ldots, i_{k-1}$ alle paarweise verschieden. Eine weitere Zahl i kann in die Folge aufgenommen werden, wenn

(1) i unter den Zahlen $1, i_1, i_2, \ldots, i_{k-1}$ nicht vorkommt und
(2) $w((v_{i_{k-1}}, v_i)) < \infty$ ist und
(3) das Gewicht der neu entstehenden Teiltour $\langle 1, i_1, i_2, \ldots, i_{k-1}\ i \rangle$ kleiner oder gleich dem Wert in der Variablen `bound` ist.

Treffen (1) oder (2) nicht zu, kann i nicht als Fortsetzung der Teiltour $1, i_1, i_2, \ldots, i_{k-1}$ genommen werden, denn es entsteht keine Tour. Trifft (3) nicht zu (aber (1) und (2)), dann brauchen alle Touren, die mit dem Anfangsstück $\langle 1, i_1, i_2, \ldots, i_{k-1}\ i \rangle$ beginnen, nicht weiter berücksichtigt zu werden.

Algorithmus zur Lösung des Problems des Handlungsreisenden nach der Branch-and-bound-Methode:

Es werden wieder die bereits bekannten Deklarationen (ergänzt um neue Deklarationen) verwendet:

```
CONST n         = ...;       { Problemgröße }
      unendlich = ...;       { hier kann der maximal mögliche
                               REAL-Wert verwendet werden      }

TYPE  knotenbereich = 1..n;
      matrix        = ARRAY [knotenbereich, knotenbereich] OF REAL;
      tourfeld      = ARRAY [1..(n+1)] OF INTEGER;
```

Die Knotennummern einer optimalen Tour werden im Feld `opttour` abgelegt.

Eingabe: $G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, \ldots, v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

```
VAR A      : matrix;        { Adjazenzmatrix }
    i      : knotenbereich;
    j      : knotenbereich;
    opttour : tourfeld;
    bound  : REAL;
```

```
FOR i := 1 TO n DO
  FOR j := 1 TO n DO
    IF (v_i,v_j) ∈ E THEN A[i, j] := w((v_i,v_j))
                     ELSE A[i, j] := unendlich;
```

Verfahren: Aufruf der Prozedur `salesman_bab (A, opttour, bound).`

Ausgabe: `bound` gibt das minimale Gewicht einer Tour durch G an, die beim Knoten v_1 beginnt und endet; `opttour` enthält die Knotennummern einer optimalen Tour.

```
PROCEDURE salesman_bab
                  (A           : matrix;
                   VAR opttour : tourfeld; { optimale Tour        }
                   VAR bound   : REAL      { Länge der Tour       }
                   );

  VAR i : INTEGER;

  PROCEDURE  bab_g
                  (teilfeld : tourfeld;
                                          { zu ergänzendes Anfangsstück
                                            einer Tour               }
                   gewicht  : REAL;       { Gewicht des Anfangsstücks }
                   position : INTEGER     { Position, an der zu
                                            ergänzen ist             }
                  );

     VAR i  : INTEGER;
         j  : INTEGER;
         ok : BOOLEAN;
         w  : REAL;

     BEGIN {bab_g }
       IF position = n + 1
       THEN BEGIN
              w := A[teilfeld[n], 1];
              IF (w < unendlich)
                 AND
                 (gewicht + w < bound)
              THEN BEGIN
                     teilfeld[position] := 1;
                     bound   := gewicht + w;
                     opttour := teilfeld;
                   END;
            END
```

```
      ELSE BEGIN
             FOR i := 2 TO n DO
               BEGIN
                 w  := A[teilfeld[position - 1], i];
                 ok := TRUE;
                 { Bedingung (2):}
                 FOR j := 2 TO position - 1 DO
                   IF teilfeld[j] = i
                   THEN BEGIN
                          ok := FALSE;
                          Break;
                        END;
                 IF ok
                    AND
                    (w < unendlich)
                    AND
                    (gewicht + w < bound)
                 THEN BEGIN
                        teilfeld[position] := i;
                        bab_g (teilfeld, gewicht + w,
                                                 position + 1);
                      END;
               END;
           END;

    END   {bab_g };

BEGIN { salesman_bab }

  FOR i := 2 TO n + 1 DO
   opttour[i]  := 0;
  opttour[1] := 1;
  bound   := unendlich;

  bab_g(opttour, 0, 2);

END   {salesman_bab };
```

Das Verfahren erzeugt u.U. alle $(n-1)!$ Folgen $\langle 1\ i_1\ i_2 \ldots i_{n-1}\ 1\rangle$, bis es eine optimale Tour gefunden hat. In der Praxis werden jedoch schnell Folgen mit Anfangsstücken, die ein zu großes Gewicht aufweisen, ausgeschlossen. Die (worst-case-) Zeitkomplexität des Verfahrens bleibt jedoch mindestens exponentiell in der Anzahl der Knoten des Graphen in der Eingabeinstanz. Das Laufzeitverhalten verbessert sich natürlich nicht, d.h. es bleibt exponentiell, wenn man die Bitoperationen in Abhängigkeit von der Anzahl der Bits untersucht, die zur Darstellung einer Eingabeinstanz erforderlich sind.

In der Praxis gilt ein Algorithmus mit exponentiellem Laufzeitverhalten als **schwer durchführbar (intractable)**, ein Algorithmus mit polynomiellem Laufzeitverhalten als **leicht durchführbar (tractable)**. Natürlich kann dabei ein Algorithmus mit exponentiellem Laufzeitverhalten für einige Eingabeinstanzen schnell ablaufen. Trotzdem bleibt die Ursache zu untersuchen, die dazu führt, daß manche Probleme „leicht lösbar“ sind (d.h. einen Algorithmus mit polynomiellem Laufzeitverhalten besitzen), während für andere Probleme bisher nur Lösungsalgorithmen mit exponentiellem Laufzeitverhalten bekannt sind.

Der Grund dafür, daß ein Verfahren mit exponentielle Laufzeit als nicht durchführbar gilt, wird aus den folgenden Tabelle sichtbar. Es werden dort jeweils fünf Funktionen $h_i : \mathbf{N}_{>0} \to \mathbf{R}, i = 1, ..., 5$ und einige ausgewählte (gerundete) Funktionswerte gezeigt. Jede Funktion soll als Laufzeit für Algorithmen interpretiert werden: Der Funktionswert $h_i(n)$ gibt die Anzahl der Rechenschritte an, die bei einer Eingabeinstanz der Größe *n* durchlaufen werden. Selbst kleine Problemgrößen führen bereits auf eine immense Laufzeit.

Spalte 1	Spalte 2	Spalte 3	Spalte 4	Spalte 5
i	$h_i(n)$	$h_i(10)$	$h_i(100)$	$h_i(1000)$
1	$\log_2(n)$	3,3219	6,6439	9,9658
2	$\sqrt{n}$	3,1623	10	31,6228
3	n	10	100	1000
4	n^2	100	10.000	1.000.000
5	2^n	1024	$1{,}2676506 \cdot 10^{30}$	$> 10^{693}$

Die folgende Tabelle zeigt noch einmal die fünf Funktionen $h_i : \mathbf{N}_{>0} \to \mathbf{R}, i = 1, ..., 5$. Es sei $y_0 > 0$ ein fester Wert. Die dritte Spalte zeigt für jede der fünf Funktionen Werte n_i mit $h_i(n_i) = y_0$. In der vierten Spalte sind diejenigen Werte $\overline{n_i}$ aufgeführt, für die $h_i(\overline{n_i}) = 10 \cdot y_0$ gilt, d.h. dort ist angegeben, auf welchen Wert man die Problemgröße n_i vergrößern kann, wenn man die 10-fachen Laufzeit in Kauf nimmt. Wie man sieht, kann man bei der Logartihmusfunktion wegen ihres langsamen Wachstums den Wert stark vergrößern, während bei der schnell anwachsenden Exponentialfunktion nur eine additive konstante Steigerung um ca. 3,3 möglich ist. Das bedeutet, daß selbst bei einer Verzehnfachung der Rechenleistung nur eine geringfügig vergrößerte Problemgröße zu bewältigen ist.

Spalte 1	Spalte 2	Spalte 3	Spalte 4
i	$h_i(n)$	n_i mit $h_i(n_i) = y_0$	$\bar{n}_i$ mit $h_i(\bar{n}_i) = 10 \cdot y_0$
1	$\log_2(n)$	n_1	$(n_1)^{10}$
2	$\sqrt{n}$	n_2	$100 \cdot n_2$
3	n	n_3	$10 \cdot n_3$
4	n^2	n_4	$\approx 3{,}162 \cdot n_4$
5	2^n	n_5	$n_5 + 3{,}322$

5.2 Der Zusammenhang zwischen Optimierungs- und Entscheidungsproblemen

Optimierungsprobleme spielen in der Anwendung eine bedeutende Rolle; in der Theorie der Berechenbarkeit sind es eher die Entscheidungsprobleme. Zwischen beiden Typen besteht ein enger Zusammenhang.

Es sei Π ein Optimierungsproblem mit einer reellwertigen Zielfunktion:

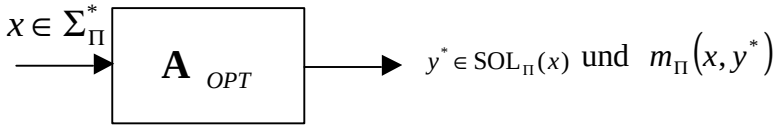
Instanz:
1. $x \in \Sigma_\Pi^*$
2. Spezifikation einer Funktion SOL_Π, die jedem $x \in \Sigma_\Pi^*$ eine Menge zulässiger Lösungen zuordnet
3. Spezifikation einer Zielfunktion m_Π, die jedem $x \in \Sigma_\Pi^*$ und $y \in \mathrm{SOL}_\Pi(x)$ einen Wert $m_\Pi(x, y)$, den Wert einer zulässigen Lösung, zuordnet
4. $goal_\Pi \in \{\min, \max\}$, je nachdem, ob es sich um ein Minimierungs- oder ein Maximierungsproblem handelt.

Lösung: $y^* \in \mathrm{SOL}_\Pi(x)$ mit $m_\Pi(x, y^*) = \min\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$ bei einem Minimierungsproblem (d.h. $goal_\Pi = \min$)
bzw.
$m_\Pi(x, y^*) = \max\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$ bei einem Maximierungsproblem (d.h. $goal_\Pi = \max$).
Der Wert $m_\Pi(x, y^*)$ einer optimalen Lösung wird auch mit $m_\Pi^*(x)$ bezeichnet.

Eine Instanz *x* ist eine Zeichenkette $x \in \Sigma_\Pi^*$. Hier wird das Alphabet Σ_Π für das Problem „geeignet“ gewählt. Für ein Graphenproblem ist sicherlich dabei ein anderes Alphabet geeignet

als zur Darstellung Boolescher Ausdrücke. Letztlich kann man sich natürlich auf ein allgemein gültiges „Grundalphabet“ verständigen, etwa $\Sigma_0 = \{0,1\}$.

Ein Lösungsalgorithmus $\mathbf{A}_{OPT}$ für Π hat die Form



Die von $\mathbf{A}_{OPT}$ bei Eingabe von $x \in \Sigma_\Pi^*$ ermittelte Lösung ist $\mathbf{A}_{OPT}(x) = (y^*, m_\Pi(x, y^*))$, d.h. sie besteht aus einer optimalen Lösung y^* und dem Wert der Zielfunktion $m_\Pi(x, y^*) = m_\Pi^*(x)$ bei einer optimalen Lösung. Natürlich kann es für ein Optimierungsproblem mehrere optimale Lösungen geben; der Wert $m_\Pi^*(x)$ ist jedoch eindeutig.

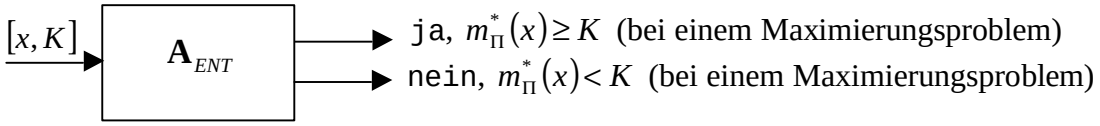
Das zu Π **zugehörige Entscheidungsproblem** Π_{ENT} wird definiert durch

Instanz: $[x, K]$
mit $x \in \Sigma_\Pi^*$ und $K \in \mathbf{R}$

Lösung: Entscheidung „ja“, falls für den Wert $m_\Pi^*(x)$ einer optimalen Lösung
$m_\Pi^*(x) \geq K$ bei einem Maximierungsproblem
(d.h. $goal_\Pi = \max$)
bzw.
$m_\Pi^*(x) \leq K$ bei einem Minimierungsproblem
(d.h. $goal_\Pi = \min$)
gilt,
Entscheidung „nein“, sonst.

Hierbei ist zu beachten, daß beim Entscheidungsproblem weder nach einer optimalen Lösung y^* noch nach dem Wert $m_\Pi^*(x) = m_\Pi(x, y^*)$ der Zielfunktion bei einer optimalen Lösung gefragt wird.

Ein Lösungsalgorithmus für Π_{ENT} hat zwei mögliche Ausgänge, nämlich einen akzeptierenden und einen verwerfenden Ausgang. Der erste Ausgang wird erreicht (aktiviert), wenn bei Eingabe einer Instanz für Π_{ENT} die Entscheidung „ja“ getroffen wird, der zweite Ausgang entspricht der Entscheidung „nein“. Ein Lösungsalgorithmus $\mathbf{A}_{ENT}$ für Π_{ENT}, hier für ein Maximierungsproblem dargestellt, hat daher die Form



Aus der Kenntnis eines Algorithmus $\mathbf{A}_{OPT}$ für ein Optimierungsproblem läßt sich leicht ein Algorithmus $\mathbf{A}_{ENT}$ für das zugehörige Entscheidungsproblem konstruieren, der im wesentlichen dieselbe Komplexität besitzt:

Eingabe: $[x, K]$
mit $x \in \Sigma_{\Pi}^{*}$ und $K \in \mathbf{R}$

Verfahren: Man berechne $\mathbf{A}_{OPT}(x) = (y^{*}, m_{\Pi}(x, y^{*}))$ und vergleiche das Resultat $m_{\Pi}(x, y^{*}) = m_{\Pi}^{*}(x)$ mit K:

Ausgabe: $\mathbf{A}_{ENT}([x,K]) = \texttt{ja}$, falls $m_{\Pi}^{*}(x) \geq K$ bei einem Maximierungsproblem ist
bzw.
$\mathbf{A}_{ENT}([x,K]) = \texttt{ja}$, falls $m_{\Pi}^{*}(x) \leq K$ bei einem Minimierungsproblem ist,
$\mathbf{A}_{ENT}([x,K]) = \texttt{nein}$ sonst.

Daher gilt:

Satz 5.2-1:

Das zu einem Optimierungsproblem Π zugehörige Entscheidungsproblem Π_{ENT} ist im wesentlichen *zeitlich nicht aufwendiger zu lösen* als das Optimierungsproblem.

Falls man andererseits bereits weiß, daß das Entscheidungsproblem Π_{ENT} immer nur „schwer lösbar“ ist, z.B. beweisbar nur Lösungsalgorithmen mit exponentiellem Laufzeitverhalten besitzt, dann ist das Optimierungsproblem ebenfalls nur „schwer lösbar“.

In den folgenden Kapiteln wird für einige *Entscheidungsprobleme* Π_{ENT} gezeigt, daß sie (vermutlich) keine schnell laufenden Lösungsverfahren besitzen. Daher können die zugehörigen Optimierungsprobleme, an deren Lösung man in der Praxis interessiert ist, ebenfalls nur mit sehr lang laufenden Verfahren angegangen werden. Die algorithmische Untersuchung von Entscheidungsproblemen ist in diesen Fällen daher für die Praxis von großem Interesse. Man hat zudem abzuwägen, ob man nicht mit einer Lösung zufrieden sein kann, die „schnell“ zu

finden ist und sich der optimalen Lösung annähert. Diese Fragestellung führt auf das Gebiet der **Approximationsalgorithmen** in Kapitel 6.

In einigen Fällen läßt sich auch die „umgekehrte“ Argumentationsrichtung zeigen: Aus einem Algorithmus $\mathbf{A}_{ENT}$ für das zu einem Optimierungsproblem zugehörige Entscheidungsproblem läßt sich ein Algorithmus $\mathbf{A}_{OPT}$ für das Optimierungsproblem konstruieren, dessen Laufzeitverhalten im wesentlichen dieselbe Komplexität aufweist. Insbesondere ist man dabei an Optimierungsproblemen interessiert, für die gilt: Hat $\mathbf{A}_{ENT}$ zur Lösung des zu einem Optimierungsproblem zugehörigen Entscheidungsproblem polynomielles Laufzeitverhalten (in der Größe der Eingabe), so hat auch der aus $\mathbf{A}_{ENT}$ konstruierte Algorithmus $\mathbf{A}_{OPT}$ zur Lösung des Optimierungsproblems polynomielles Laufzeitverhalten.

Ein Beispiel ist das Problem des Handlungsreisenden auf Graphen mit natürlichzahligen Gewichten:

Problem des Handlungsreisenden auf Graphen mit ganzzahligen Gewichten als Optimierungsproblem

Instanz: 1. $G = (V, E, w)$

$G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{N}$ gibt jeder Kante $e \in E$ ein **natürlichzahliges Gewicht**; es ist $w((i, j)) = \infty$ für $(i, j) \notin E$

Als Problemgröße wird hier der Wert $k = n \cdot A$ mit

$A = \max\{\lfloor \log_2(|w(e)|) \rfloor + 1 \mid e \in E\}$ definiert, da in das Verfahren wesentlich die numerischen Größen der beteiligten Kantengewichte eingehen (für $w(e) = 0$ wird anstelle von $\lfloor \log_2(|w(e)|) \rfloor + 1$ der Wert 1 genommen).

2. $\mathrm{SOL}(G) = \{T \mid T = \langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ ist eine Tour durch $G\}$
3. für $T \in \mathrm{SOL}(G)$, $T = \langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$, ist die Zielfunktion definiert durch $m(G, T) = \sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}}) + w(v_{i_n}, v_{i_1})$
4. $goal = \min$

Lösung: Eine Tour $T^* \in \mathrm{SOL}(G)$, die minimale Kosten (unter allen möglichen Touren durch G) verursacht, und $m(G, T^*)$.

Ein Algorithmus, der bei Eingabe einer Instanz $G=(V,E,w)$ eine optimale Lösung erzeugt, werde mit $\mathbf{A}_{TSPOPT}$ bezeichnet. Die von $\mathbf{A}_{TSPOPT}$ bei Eingabe von *G* ermittelte Tour mit minimalen Kosten sei T^*, die ermittelten minimalen Kosten $m^*(G)$:
$\mathbf{A}_{TSPOPT}(G)=(T^*,m^*(G))$.

Das zugehörige Entscheidungsproblem lautet:

Problem des Handlungsreisenden auf Graphen mit ganzzahligen Gewichten als Entscheidungsproblem

Instanz: $[G,K]$

$G=(V,E,w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V=\{v_1,...,v_n\}$, bestehend aus *n* Knoten, und der Kantenmenge $E\subseteq V\times V$; die Funktion $w:E\rightarrow \mathbf{N}$ gibt jeder Kante $e\in E$ ein natürlichzahliges Gewicht; es ist $w((i,j))=\infty$ für $(i,j)\notin E$;

$K\in \mathbf{N}$.

Die Problemgröße ist $k=n\cdot A$ mit $A=\max\{\lfloor \log_2(w(e))\rfloor+1 \mid e\in E\}$.

Lösung: Entscheidung „ja“, falls es eine Tour durch *G* gibt mit minimalen Kosten $m^*(G)\leq K$ gibt,

Entscheidung „nein“, sonst.

Ein Algorithmus, der bei Eingabe einer Instanz $[G,K]$ eine entsprechende Entscheidung fällt, werde mit $\mathbf{A}_{TSPENT}$ bezeichnet. Die von $\mathbf{A}_{TSPENT}$ bei Eingabe von $[G,K]$ getroffene `ja/nein`-Entscheidung sei $\mathbf{A}_{TSPENT}([G,K])$. Mit Hilfe von $\mathbf{A}_{TSPENT}$ wird ein Algorithmus $\mathbf{A}_{TSPOPT}$ zur Lösung des Optimierungsproblem konstruiert. Dabei wird wiederholt Binärsuche eingesetzt, um zunächst die Kosten einer optimalen Tour zu ermitteln.

Der Algorithmus $\mathbf{A}_{TSPOPT}$ zur Lösung des Optimierungsproblem verwendet eine als Pseudocode formulierte Prozedur `A_TSPOPT`. Der Eingabegraph kann wieder in Form seiner Adjazenzmatrix verarbeitet werden. Da Implementierungsdetails hier nicht weiter betrachtet werden sollen, können wir annehmen, daß es zur Darstellung eines gewichteten Graphen einen geeigneten Datentyp

```
TYPE gewichteter_Graph = ...;
```

und zur Speicherung einer Tour (Knoten und Kanten) in dem Graphen einen Datentyp

```
TYPE tour = ...;
```

gibt. Der Algorithmus $\mathbf{A}_{TSPOPT}$ hat folgendes Aussehen:

Eingabe: $G=(V,E,w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V=\{v_1,...,v_n\}$, bestehend aus *n* Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \to \mathbf{N}$ gibt jeder Kante $e \in E$ ein natürlichzahliges Gewicht; ; es ist $w((i,j))=\infty$ für $(i,j) \notin E$

```
VAR G      : gewichter_Graph;
    opttour : tour;
    opt     : INTEGER;
```

G := $G=(V,E,w)$

Verfahren: Aufruf der Prozedur `A_TSPOPT (G, opt, opttour).`

Ausgabe: `opttour` = T^*, d.h. eine Tour mit minimalen Kosten, `opt` = die ermittelten minimalen Kosten $m^*(G)$.

```
PROCEDURE A_TSPOPT (G           : gewichter_Graph;
                    {  Graph mit natürlichzahligen Kantengewichten }
                    VAR opt     : INTEGER;
                     { Gewicht einer optimalen Tour               }
                    VAR opttour : graph;
                     { ermittelte Tour mit minimalem Gewicht      }

VAR s : INTEGER;


   PROCEDURE TSPOPT (G       : gewichter_graph;
                     VAR opt : INTEGER;)
        { ermittelt im Parameter opt das Gewicht einer
          optimalen Tour in G :                                    }

      VAR min : INTEGER;
          max : INTEGER;
          t   : INTEGER;

   BEGIN { TSPOPT }
     min := 0;
     max := ∑ w(e) ;
            e∈E

     { Binärsuche auf dem Intervall [0..max]: }
     WHILE max - min >= 1 DO
       BEGIN
```

```
        t := ⌈(min + max) / 2⌉;
        IF A_TSPENT([G, t]) = „ja“
        THEN max := t
        ELSE min := t + 1;
      END;

    { t enthält nun das Gewicht einer optimalen Tour in G: }
    opt := t;
  END   { TSPOPT };

BEGIN { A_TSPOPT }
  TSPOPT (G, opt);

  FOR alle e ∈ E DO
    BEGIN
      ersetze in G das Gewicht w(e) der Kante e durch w(e) + 1;
      TSPOPT (G, s);
      IF s > opt
      THEN { es gibt keine optimale Tour in G, die e nicht enthält }
           ersetze in G das Gewicht w(e) der Kante e durch w(e) - 1;
    END;

  { alle Kanten mit nicht erhöhten Gewichten bestimmen eine Tour
    mit minimalen Kosten                                          }
  tour := Menge der Kanten mit nicht erhöhten Gewichten und zugehörige Knoten;
END   { A_TSPOPT };
```

Der Graph G habe die Problemgröße $k = n \cdot A$ mit $A = \max\{\lfloor \log_2(w(e)) \rfloor + 1 \mid e \in E\}$. Es sei $m = \sum_{e \in E} w(e)$. Dann sind in obigem Verfahren höchstens $\lceil \log_2(m) \rceil$ viele Aufrufe des Entscheidungsverfahrens $\mathbf{A}_{TSPENT}([G, t])$ erforderlich (Binärsuche). Es gilt:

$$\lceil \log_2(m) \rceil \le \log_2(m) + 1 \le \sum_{e \in E} \log_2(w(e)) + 1$$
$$\le \sum_{e \in E} (\lfloor \log_2(w(e)) \rfloor + 1) + 1 \le n^2 \cdot A + 1 \le k^2 + 1 \in O(k^2),$$

d.h. $\lceil \log_2(m) \rceil \in O(k^2)$.

Falls man also weiß, daß $\mathbf{A}_{TSPENT}$ ein in der Größe der Eingabe polynomiell zeitbeschränkter Algorithmus ist, ist das gesamte Verfahren zur Bestimmung einer optimalen Lösung polynomiell zeitbeschränkt.

Falls man umgekehrt weiß, daß es beweisbar keinen schnellen Algorithmus zur Lösung des Problems des Handlungsreisenden auf Graphen mit ganzzahligen Gewichten als Optimierungsproblem gibt, gibt es einen solchen auch nicht für das zugehörige Entscheidungspro-

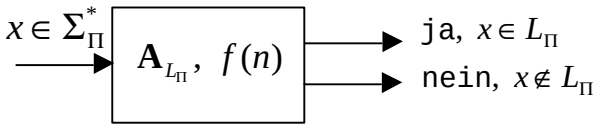
blem. In diesem Fall ist das Entscheidungsproblem genauso schwer zu lösen wie das Optimierungsproblem.

5.3 Komplexitätsklassen

Für ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ gilt $L_\Pi \in TIME(f(n))$, wenn es einen (deterministischen) Algorithmus $\mathbf{A}_{L_\Pi}$ folgender Form gibt:

Eingabe: $x \in \Sigma_\Pi^*$ mit $|x| = n$

Ausgabe: `ja`, falls $x \in L_\Pi$ gilt
`nein`, falls $x \notin L_\Pi$ gilt.



Hierbei wird die Entscheidung $\mathbf{A}_{L_\Pi}(x)$ nach einer Anzahl von Schritten getroffen, die in $O(f(n))$ liegt.

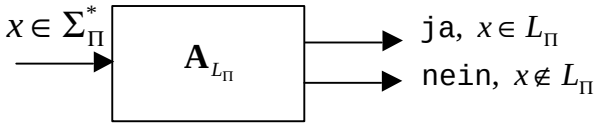
Die Aussage „das Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in $TIME(f(n))$“ steht synonym für $L_\Pi \in TIME(f(n))$; man sagt auch abkürzend „das Entscheidungsproblem Π liegt in $TIME(f(n))$“. In diesem Sinn bezeichnet $TIME(f(n))$ die **Klasse der Entscheidungsprobleme, die in der Zeitkomplexität $O(f(n))$ gelöst werden können.**

Eine entsprechende Definition gilt für die Speicherplatzkomplexität:

Für ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ gilt $L_\Pi \in SPACE(f(n))$, wenn es einen Algorithmus $\mathbf{A}_{L_\Pi}$ folgender Form gibt:

Eingabe: $x \in \Sigma_\Pi^*$ mit $|x| = n$

Ausgabe: `ja`, falls $x \in L_\Pi$ gilt
`nein`, falls $x \notin L_\Pi$ gilt.



Hierbei werden zur Findung der Entscheidung $\mathbf{A}_{L_\Pi}(x)$ eine Anzahl von Speicherzellen verwendet, die in $O(f(n))$ liegt.

Die Aussage „das Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in $SPACE(f(n))$“ steht synonym für $L_\Pi \in SPACE(f(n))$; man sagt auch „das Entscheidungsproblem Π liegt in $SPACE(f(n))$“. In diesem Sinn bezeichnet $SPACE(f(n))$ die **Klasse der Entscheidungsprobleme, die mit Speicherplatzkomplexität $O(f(n))$ gelöst werden können.**

Wichtige Komplexitätsklassen sind

- die Klasse der Entscheidungsprobleme, die in einer Zeit gelöst werden können, die proportional zu einem Polynom in der Größe der Eingabe ist:

  $$\mathbf{P} = \bigcup_{k=0}^{\infty} TIME(n^k)$$

  Beispielsweise gibt es für jede von einer kontextfreien Grammatik G erzeugten Sprache $L(G) \subseteq \Sigma^*$ ein Entscheidungsverfahren, das für ein Wort $u \in \Sigma^*$ mit $|u| = n$ in $O(n^3)$ vielen Schritten entscheidet, ob $u \in L(G)$ gilt oder nicht (vgl. Kapitel 4.4). Daher ist die Klasse der kontextfreien Sprachen in **P** enthalten. Diese Inklusion ist echt, wie das Beispiel der Sprache $L_4 = \{a^n b^n c^n \mid n \in \mathbf{N}, n \geq 1\}$ aus Kapitel 4.1 zeigt, die in **P** liegt, aber nicht kontextfrei ist.
- die Klasse der Entscheidungsprobleme, die in mit einem Speicherplatzbedarf gelöst werden können, der proportional zu einem Polynom in der Größe der Eingabe ist:

  $$\mathbf{PSPACE} = \bigcup_{k=0}^{\infty} SPACE(n^k)$$

- die Klasse der Entscheidungsprobleme, die in einer Zeit gelöst werden können, die proportional zu einer Exponentialfunktion in der Größe der Eingabe ist:

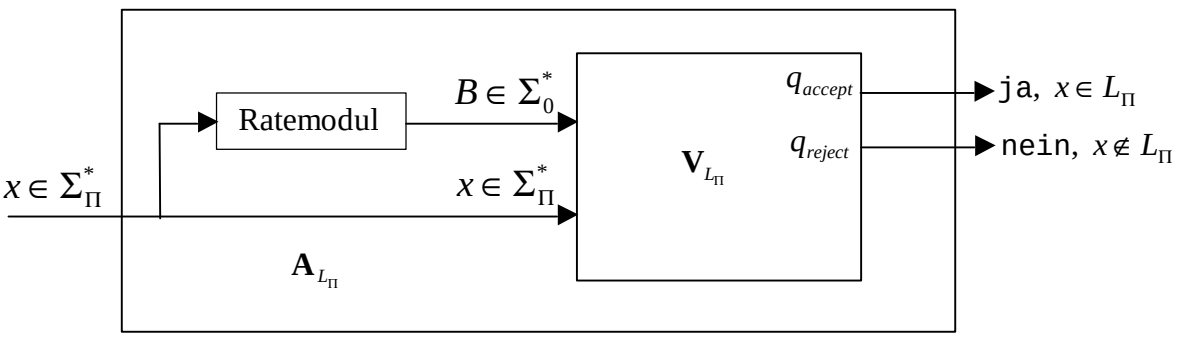
  $$\mathbf{EXP} = \bigcup_{k=0}^{\infty} TIME(2^{n^k}).$$

Es gilt $\mathbf{P} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP}$. Die Frage, ob eine dieser Inklusionen echt ist, d.h. ob $\mathbf{P} \subset \mathbf{PSPACE}$ oder $\mathbf{PSPACE} \subset \mathbf{EXP}$ gilt, ist ein bisher ungelöstes Problem der Komplexitätstheorie. Man weiß jedoch, daß $\mathbf{P} \subset \mathbf{EXP}$ gilt.

In Kapitel 2.5 wird eine nichtdeterministische Turingmaschinen $TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$ in zunächst unterschiedlichen Modell-Sichtweisen definiert. Entweder wird die Überführungsfunktion als partielle Funktion der Form $d : Q \times \Sigma^k \to \mathbf{P}(Q \times (\Sigma \times \{L, R, S\})^k)$ angegeben; während der Berechnung kann die Turingmaschine nichtdeterministische Schritte ausführen, indem sie bei Erreichen einer Konfiguration aus mehreren möglichen Folgekonfigurationen „die richtige“ auswählt. Oder die nichtdeterministische Turingmaschine wird als deterministische Turingmaschine gesehen, die mit einem „Ratemodul“ ausgestattet ist, das in nichtdeterministischer Weise zunächst eine Zusatzinformation generiert, deren Brauchbarkeit anschließend mit Hilfe einer Überführungsfunktion der Form $d : Q \times \Sigma^k \to Q \times (\Sigma \times \{L, R, S\})^k$ deterministisch verifiziert wird. Diese Überlegung führte schließlich auf die Definition eines nichtdeterministischen Algorithmus unter Einsatz eines Verifizierers. Die Definition wird hier noch einmal wiederholt. Da hier nur entscheidbare Probleme behandelt werden, stoppt hier der Verifizierer bei allen Eingaben.

Ein nichtdeterministischer Algorithmus für ein Entscheidungsproblem Π über einem Alphabet Σ_Π besteht aus einem Ratemodul und einem Verifizierer $\mathbf{V}_{L_\Pi}$. Bei Eingabe von $x \in \Sigma_\Pi^*$ erzeugt der Ratemodul auf nichtdeterministische Weise eine Zusatzinformation (einen Beweis) $B \in \Sigma_0^*$; dabei wird er nur einmal durchlaufen. Die Eingabe $x \in \Sigma_\Pi^*$ und der erzeugte Beweis $B \in \Sigma_0^*$ werden in den Verifizierer eingeben, dessen Aufgabe darin besteht, auf deterministische Weise mit Hilfe des Beweises *B* die Frage „$x \in L_\Pi$?“ zu entscheiden:

Ein **nichtdeterministischer Algorithmus $\mathbf{A}_{L_\Pi}$ für ein Entscheidungsproblem** Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ hat die Form



Es gilt für $x \in \Sigma_{\Pi}^{*}$:

$x \in L_{\Pi}$, falls es einen Beweis $B_x \in \Sigma_0^{*}$ gibt, so daß $\mathbf{V}_{L_{\Pi}}$ bei Eingabe von x und B_x mit $\mathbf{V}_{L_{\Pi}}(x, B_x) = \texttt{ja}$ stoppt;

$x \notin L_{\Pi}$, falls für jeden Beweis $B \in \Sigma_0^{*}$ gilt: $\mathbf{V}_{L_{\Pi}}$ stoppt mit $\mathbf{V}_{L_{\Pi}}(x, B) = \texttt{nein}$.

Es sei $f : \mathbf{N} \to \mathbf{N}$ eine Funktion. Falls der nichtdeterministische Algorithmus mit Hilfe des Verifizierers $\mathbf{V}_{L_{\Pi}}$ die ja/nein-Entscheidung bei Eingabe von $x \in \Sigma_{\Pi}^{*}$ mit $|x| = n$ nach einer Anzahl von Schritten trifft, die in $O(f(n))$ liegt, so ist der Algorithmus $f(n)$-**zeitbeschränkt**.

Zu beachten ist, daß die Laufzeit des nichtdeterministischen Algorithmus in Abhängigkeit von der Größe von $x \in \Sigma_{\Pi}^{*}$ gemessen wird. Ist dieser $f(n)$-zeitbeschränkt, so werden auch nur $C \cdot f(n)$ viele Zeichen eines Beweises generiert (hierbei ist C eine Konstante). Wenn es also überhaupt einen Beweis B_x mit $\mathbf{V}_{L_{\Pi}}(x, B_x) = \texttt{ja}$ gibt, dann gibt es auch einen Beweis mit Länge $\leq C \cdot f(n)$, so daß man für den Beweis gleich eine durch $C \cdot f(n)$ beschränkte Länge annehmen kann.

Für ein Entscheidungsproblem Π mit der Menge $L_{\Pi} \subseteq \Sigma_{\Pi}^{*}$ gilt $L_{\Pi} \in NTIME(f(n))$, wenn es einen nichtdeterministischen $f(n)$-zeitbeschränkten Algorithmus zur Akzeptanz von L_{Π} gibt.

Die Aussage „das Entscheidungsproblem Π mit der Menge $L_{\Pi} \subseteq \Sigma_{\Pi}^{*}$ liegt in $NTIME(f(n))$“ steht synonym für $L_{\Pi} \in NTIME(f(n))$. In diesem Sinn bezeichnet $NTIME(f(n))$ die **Klasse der Entscheidungsprobleme, die auf nichtdeterministische Weise in der Zeitkomplexität $O(f(n))$ gelöst werden können.**

Eine der wichtigsten Klassen, die Klasse **NP** der Entscheidungsprobleme, die nichtdeterministisch mit polynomieller Zeitkomplexität gelöst werden können, ergibt sich, wenn f ein Polynom ist:

$$\mathbf{NP} = \bigcup_{k=0}^{\infty} NTIME(n^k)$$

In Kapitel 2.5 wird gezeigt, daß ein $T(n)$-zeitbeschränkter nichtdeterministischer Algorithmus durch einen deterministischen $O\left(c^{T(n)}\right)$-zeitbeschränkten Algorithmus simuliert werden kann. Setzt man $T(n)=n^k$, so sieht man unmittelbar, daß $NTIME\left(n^k\right) \subseteq TIME\left(2^{O\left(n^k\right)}\right) \subseteq TIME\left(2^{n^{k_1}}\right)$ für einen Wert $k_1 \geq k$ gilt. Daraus folgt $\mathbf{NP} = \bigcup_{k=0}^{\infty} NTIME\left(n^k\right) \subseteq \bigcup_{k=0}^{\infty} TIME\left(2^{n^k}\right) = \mathbf{EXP}$.

Da ein polynomiell zeitbeschränkter Algorithmus $\mathbf{A}_{L_\Pi}$, wie er in der Definition von **P** vorkommt, ein spezieller polynomiell zeitbeschränkter nichtdeterministischer Algorithmus ist, nämlich ein Algorithmus, der für seine Entscheidung ohne die Zuhilfenahme eines von einem Ratemodul erzeugten Beweises auskommt, gilt

$\mathbf{P} \subseteq \mathbf{NP}$.

Insgesamt ergibt sich $\mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{EXP}$. Zusammen mit $\mathbf{P} \subset \mathbf{EXP}$ fragt sich, welche der Inklusionen echt ist. Die dabei wichtigste Frage $\mathbf{P} = \mathbf{NP}$ bzw. $\mathbf{P} \neq \mathbf{NP}$ ist bisher ungelöst (**P-NP-Problem**). Vieles spricht für $\mathbf{P} \neq \mathbf{NP}$.

5.4 Die Klassen P und NP

Das vorliegende Kapitel behandelt Beispiele aus den Klassen **P** und **NP**.

Zur Verdeutlichung des Unterschieds zwischen **P** und **NP** werden die entsprechenden Definitionen der beiden Klassen noch einmal gegenübergestellt:

Ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in **P**, wenn es einen deterministischen Algorithmus $\mathbf{A}_{L_\Pi}$ und ein Polynom $p(n)$ gibt, so daß ein Entscheidungsverfahren für Π folgende Form hat:

Eingabe: $x \in \Sigma_\Pi^*$ mit $|x| = n$

Ausgabe: Entscheidung „$x \in L_\Pi$“, falls $\mathbf{A}_{L_\Pi}(x) = \texttt{ja}$ gilt,
Entscheidung „$x \notin L_\Pi$“, falls $\mathbf{A}_{L_\Pi}(x) = \texttt{nein}$ gilt.

Hierbei wird die Entscheidung $\mathbf{A}_{L_\Pi}(x)$ nach einer Anzahl von Schritten getroffen, die in $O(p(n))$ liegt.

Ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in **NP**, wenn es einen nichtdeterministischen Algorithmus unter Einsatz eines Verifizierer $\mathbf{V}_{L_\Pi}$, der ein vom Ratemodul erzeugten Beweis verifiziert, und ein Polynom $p(n)$ gibt, so daß ein Entscheidungsverfahren für Π folgende Form hat:

Eingabe: $x \in \Sigma_\Pi^*$ mit $|x| = n$

Ausgabe: Entscheidung „$x \in L_\Pi$“, falls es einen Beweis $B_x \in \Sigma_0^*$ gibt mit $|B_x| \le C \cdot p(n)$ und $\mathbf{V}_{L_\Pi}(x, B_x) = \texttt{ja}$;
Entscheidung „$x \notin L_\Pi$“ falls für alle Beweise $B \in \Sigma_0^*$ mit $|B| \le C \cdot p(n)$ gilt: $\mathbf{V}_{L_\Pi}(x, B) = \texttt{nein}$.

Hierbei wird die Entscheidung des Algorithmus und insbesondere das Ergebnis der Verifikationsphase $\mathbf{V}_{L_\Pi}(x, B)$ nach einer Anzahl von Schritten getroffen, die in $O(p(n))$ liegt.

In den meisten Fällen zur Entscheidung einer Menge in **NP** ist es offensichtlich, wie ein „geeigneter“ Beweis auszusehen hat und wie er nichtdeterministisch erzeugt werden kann. Die wesentlichen Schritte liegen in der Verifikation. Diese ist zeitlich polynomiell beschränkt. Umgangssprachlich kann man daher die Klassen **P** und **NP** etwa folgendermaßen beschreiben:

Die Klasse **P** ist die Klasse der Entscheidungsprobleme, für die bei Vorgabe einer Instanz **in polynomieller Zeit** (in Abhängigkeit von der Größe der Instanz) **entschieden wird, ob die Instanz eine das Problem definierende Eigenschaft besitzt oder nicht**.

Die Klasse **NP** ist die Klasse der Entscheidungsprobleme, für die bei Vorgabe einer Instanz und eines Beweises, der zeigen soll, daß die Instanz eine das Problem definierende Eigenschaft besitzt, **der Beweis in polynomieller Zeit** (in Abhängigkeit von der Größe der Instanz) **verifiziert werden**.

Man kann die Klassen **P** bzw. **NP** auch durch deterministische bzw. nichtdeterministische Turingmaschinen charakterisieren.

Ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in **P**, wenn es eine deterministische Turingmaschine TM_{L_Π} gibt, die bei Eingabe eines Wortes $x \in \Sigma_\Pi^*$ entscheidet, ob $x \in L_\Pi$ gilt oder nicht. Diese Entscheidung wird in $p_{L_\Pi}(|x|)$ vielen Schritten getroffen, wobei p_{L_Π} ein Polynom mit $p_{L_\Pi}(n) \ge n$ ist. Ist $x \in L_\Pi$, dann stoppt TM_{L_Π} im Zustand q_{accept} („ja-Zustand“,

„ja-Entscheidung“). Ist $x \notin L_\Pi$, dann befindet sich TM_{L_Π} nach höchstens $p_{L_\Pi}(|x|)$ in einem Zustand $q \neq q_{accept}$, für den es keinen Nachfolgezustand gibt. Dann stoppt TM_{L_Π} in einem nichtakzeptierenden Zustand („nein-Zustand“, „nein-Entscheidung“). Man kann darüber hinaus annehmen, daß TM_{L_Π} nur ein Band besitzt, da jede *k*-DTM durch eine 1-DTM simuliert werden kann; diese Simulation ist von der Ordnung $O\left(p^2_{L_\Pi}(n)\right)$, also wieder polynomiell zeitbeschränkt, so daß man für die weiteren Betrachtungen dann das Polynom $C \cdot p^2_{L_\Pi}(n)$ anstelle von $p_{L_\Pi}(n)$ nimmt. Da TM_{L_Π} also nicht mehr als $p_{L_\Pi}(|x|)$ viele Schritte ausführt, werden von TM_{L_Π} auch höchstens nur die Zellen mit den Nummern 1, ..., $p_{L_\Pi}(|x|)+1$ besucht.

Ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma^*_\Pi$ liegt in **NP**, wenn die soeben beschriebene Turingmaschine mit einer nichtdeterministischen Überführungsfunktion arbeitet. Legt man dabei das in Kapitel 2.5 beschriebene Nichtstandardmodell einer Turingmaschine TM_{L_Π} mit Ratemodul zugrunde, dann kann man annehmen, daß das Band von TM_{L_Π} beidseitig unbegrenzt ist; das Eingabewort *x* belegt anfangs die Zellen mit den Nummern 1, ..., $|x|$, in Zelle 0 wird eine linke Markierung # geschrieben. Die Zellen mit negativen Nummern werden vom Ratemodul zur anfänglichen Erzeugung eines Beweises verwendet, der dann anschließend deterministisch verifiziert wird. In diesem Fall besucht TM_{L_Π} höchstens die Zellen $-p_{L_\Pi}(|x|)$, ..., 0, 1, ..., $p_{L_\Pi}(|x|)+1$.

Einige Beispiele für Probleme aus der Klasse $\mathbf{P} = \bigcup_{k=0}^{\infty} TIME\left(n^k\right)$ wurden bereits in den vorherigen Kapiteln behandelt. Dazu gehört das Problem, festzustellen, ob zwischen zwei Knoten eines gewichteten Graphen ein Pfad mit minimalem Gewicht existiert, das eine vorgegebene Schranke nicht überschreitet.

Entscheidungsprobleme leiten sich häufig von Optimierungsproblemen ab. Leider stellt sich heraus, daß für eine Vielzahl dieser Entscheidungsprobleme keine Lösungsalgorithmen bekannt sind, die polynomielles Laufzeitverhalten aufweisen. Das gilt insbesondere für viele Entscheidungsprobleme, die zu Optimierungsproblemen gehören, die für die Praxis relevant sind (Problem des Handlungsreisenden, Partitionenproblem usw.). Für diese (Optimierungs-) Probleme verfügt man meist über deterministische Lösungsalgorithmen, deren Laufzeitverhalten exponentiell in der Größe der Eingabe ist, bzw. man kann den Nachweis erbringen, daß die zugehörigen Entscheidungsprobleme in **NP** liegen. Derartige Verfahren werden als praktisch nicht durchführbar (intractable) angesehen, obwohl durch den Einsatz immer schnellerer Rechner immer größere Probleme behandelt werden können. Es stellt sich daher die Frage, wieso trotz intensiver Suche nach polynomiellen Lösungsverfahren derartige schnelle Algorithmen (bisher) nicht gefunden wurden. Seit Anfang der 1970‘er Jahre erklärt eine inzwi-

schen gut etablierte Theorie, die **Theorie der NP-Vollständigkeit**, Ursachen dieses Phänomens. Eine Einführung in diese Theorie gibt Kapitel 5.5.

Ein wichtiges Beispiel für Probleme auf der Grenze zwischen **P** und **NP** ist das Problem der Erfüllbarkeit Boolescher Ausdrücke (siehe Kapitel 1.1).

Das folgende Entscheidungsproblem liegt in **P**:

Erfüllende Wahrheitsbelegung (erfSAT)

Instanz: $[F, f]$

F ist ein Boolescher Ausdruck in konjunktiver Normalform mit der Variablenmenge $V = \{x_1, ..., x_n\}$, $f : V \to \{\mathsf{TRUE}, \mathsf{FALSCH}\}$ ist eine Belegung der Variablen mit Wahrheitswerten.

Lösung: Entscheidung „ja“, falls die durch f gegebene Belegung dem Ausdruck F den Wahrheitswert TRUE gibt,
Entscheidung „nein“, sonst.

Ein Lösungsalgorithmus setzt einfach die in der Eingabe-Instanz $[F, f]$ gelieferte Belegung f in die Formel F ein und ermittelt den Wahrheitswert der Formel gemäß den Auswertungsregeln für die beteiligten Junktoren.

Das folgende Entscheidungsproblem liegt in **PSPACE**:

Erfüllbarkeitsproblem für Boolesche Ausdrücke in konjunktiver Normalform (CSAT)

Instanz: F

F ist ein Boolescher Ausdruck in konjunktiver Normalform mit der Variablenmenge $V = \{x_1, ..., x_n\}$.

Lösung: Entscheidung „ja“, falls es eine Belegung der Variablen von F gibt, so daß sich bei Auswertung der Formel F der Wahrheitswert TRUE ergibt,
Entscheidung „nein“, sonst.

CSAT liegt in **PSPACE** (und damit in **EXP)**. Für den Beweis genügt es zu zeigen, daß nacheinander alle möglichen 2^n Belegungen der n Variablen in einer Eingabe-Instanz F mit einem Speicherplatzverbrauch von polynomiell vielen Speicherzellen erzeugt, in die Formel F eingesetzt und ausgewertet werden können.

Es ist nicht bekannt, ob CSAT in **P** liegt (vieles spricht dagegen).

Die Probleme erfSAT mit einer Eingabeinstanz $[F, f]$ und CSAT mit einer Eingabeinstanz F unterscheiden sich grundsätzlich dadurch, daß bei ersterem in der Eingabeinstanz $[F, f]$ eine wesentliche Zusatzinformation, nämlich eine potentiell erfüllende Belegung f der Variablen vorgegeben ist, die nur noch daraufhin überprüft werden muß, ob sie wirklich die in der Eingabeinstanz enthaltene Formel F erfüllt. Zur Entscheidung, ob eine Eingabeinstanz F von CSAT erfüllbar ist, muß also entweder eine erfüllende Belegung f konstruiert bzw. aufgrund geeigneter Argumente die Erfüllbarkeit gezeigt werden, oder es muß der Nachweis erbracht werden, daß keine Belegung der Variablen von F die Formel erfüllt. Wenn dieser Nachweis nur dadurch gelingt, daß *alle* 2^n möglichen Belegungen überprüft werden, ist mit einem polynomiellen Entscheidungalgorithmus nicht zu rechnen. Intuitiv ist diese Entscheidungsaufgabe also schwieriger zu bewältigen, weil weniger Anfangsinformationen vorliegen, als lediglich die Verifikation einer potentiellen Lösung.

Eine einfache Überlegung zeigt, daß CSAT in **NP** liegt. Ein Verifizierer für CSAT arbeitet wie folgt: Er erhält mit einer Eingabeinstanz F (mit n Variablen) für CSAT als Zusatzinformation (Beweis) eine Belegung B_F der Variablen in F. Wenn F erfüllbar ist, nimmt man für B_F gerade eine erfüllende Belegung. Wenn F nicht erfüllbar ist, liefert keine Belegung B_F der Variablen den Wahrheitswert `TRUE`. Der Verifizierer überprüft lediglich (auf deterministische Weise) in $O(n)$ vielen Schritten, ob sich der Wert `TRUE` ergibt, wenn man die in B_F vorkommenden Wahrheitsheitswerte in F einsetzt; in diesem Fall wird F akzeptiert, ansonsten nicht. Insgesamt liegt polynomielles Laufzeitverhalten vor.

Da bei diesem Verfahren nicht wesentlich eingeht, ob F in konjunktiver Normalform vorliegt, sondern lediglich, ob F ein korrekter Boolescher Ausdruck und erfüllbar ist, ergibt sich, daß auch das folgende allgemeinere Problem SAT in **NP** liegt.

Erfüllbarkeitsproblem für Boolesche Ausdrücke in allgemeiner Form (SAT)

Instanz: F
F ist ein Boolescher Ausdruck mit der Variablenmenge $V = \{x_1, ..., x_n\}$.

Lösung: Entscheidung „ja“, falls es eine Belegung der Variablen von F gibt, so daß sich bei Auswertung der Formel F der Wahrheitswert `TRUE` ergibt,
Entscheidung „nein“, sonst.

SAT liegt in **NP**.

Schränkt man das Problem CSAT auf diejenigen Booleschen Ausdrücke in konjunktiver Normalform ein, in denen jede Klausel genau 2 Literale enthält, so erhält man das Problem 2-SAT. Entsprechend ist 3-SAT dadurch definiert, daß hier jede Klausel genau 3 Literale enthält.

Erfüllbarkeitsproblem für Boolesche Ausdrücke in konjunktiver Normalform mit genau 2 Literalen pro Klausel (2-CSAT)

Instanz: F

$F = F_1 \wedge ... \wedge F_m$ ist ein Boolescher Ausdruck in konjunktiver Normalform mit der Variablenmenge $V = \{x_1, ..., x_n\}$ mit der zusätzlichen Eigenschaft, daß jedes F_i genau zwei Literale enthält.

Lösung: Entscheidung „ja“, falls es eine Belegung der Variablen von *F* gibt, so daß sich bei Auswertung der Formel F der Wahrheitswert `TRUE` ergibt,
Entscheidung „nein“, sonst.

Man kann zeigen, daß 2-SAT in **P** liegt:

Satz 5.4-1:
Ist *F* eine Instanz für 2-CSAT mit *n* Variablen und *m* Klauseln, dann liefert das beschriebene Verfahren mit der Prozedur `Erfuellbarkeit_2CSAT` eine korrekte ja/nein-Entscheidung. Die (worst-case-) Zeitkomplexität des Verfahrens ist von der Ordnung $O(n \cdot m)$. Dieser Aufwand ist polynomiell in der Größe der Eingabe.

Beweis:

Ein Algorithmus zur Ermittlung einer erfüllenden Belegung kann etwa nach folgender Strategie vorgehen:

Man nehme eine bisher noch nicht betrachtete Klausel $F_i = (y_1 \vee y_2)$ von *F*. Falls eines der Literale während des bisherigen Ablaufs bereits den Wahrheitswert `TRUE` erhalten hat, dann wird F_i als erfüllt erklärt. Falls eines der Literale, etwa y_1, den Wert `FALSE` hat, dann erhält das Literal y_2 den Wahrheitswert `TRUE` und $\neg y_2$ den Wahrheitswert `FALSE`. *F* gilt dann als erfüllt. Dabei kann jedoch ein Konflikt auftreten, nämlich daß ein Literal durch die Zuweisung einen Wahrheitswert bekommen soll, jedoch den komplementären Wahrheitswert bereits vorher erhalten hat. In diesem Fall wird die vorherige Zuweisung rückgängig gemacht. Falls es dann wieder zu einem Konflikt mit diesem Literal kommt, ist die Formel nicht erfüllbar.

Der folgende Pseudocode-Algorithmus implementiert diese Strategie; aus Gründen der Lesbarkeit wird auf die exakte Deklaration der lokalen Variablen und der verwendeten Datentypen und auf deren Implementierung verzichtet.

Algorithmus zur Lösung des Erfüllbarkeitsproblems für Boolesche Ausdrücke in konjunktiver Normalform mit genau 2 Literalen pro Klausel (2-CSAT)

Eingabe: $F = F_1 \wedge ... \wedge F_m$

F ist ein Boolescher Ausdruck in konjunktiver Normalform mit der Variablenmenge $V = \{x_1, ..., x_n\}$ mit der zusätzlichen Eigenschaft, daß jedes F_i genau zwei Literale enthält, d.h. jedes F_i hat die Form $F_i = (y_{i_1} \vee y_{i_2})$

$size(F) = n$.

Der Datentyp
`TYPE KNF_typ = ...;`
beschreibe den Typ einer Formel in konjunktiver Normalform, der Datentyp
`TYPE Literal_typ = ...;`
den Typ eines Literals bzw. einer Variablen in einem Booleschen Ausdruck.

```
VAR F            : KNF_typ;
    Entscheidung : Entscheidungs_typ;
```

`F :=` $F_1 \wedge ... \wedge F_m$

Verfahren: Aufruf der Prozedur
`Erfuellbarkeit_2CSAT (F, Entscheidung);`

Im Ablauf der Prozedur werden Variablen Wahrheitswerte `TRUE` bzw. `FALSE` zugewiesen. Zu Beginn des Ablaufs hat jede Variable noch einen undefinierten Wert; in diesem Fall wird die Variable als „noch nicht zugewiesen“ bezeichnet. Jede Klausel F_i von F gilt zu Beginn des Prozedurablaufs als „unerfüllt“ (da ihren Literalen ja noch kein Wahrheitswert zugewiesen wurde). Sobald einem Literal in F_i der Wahrheitswert `TRUE` zugewiesen wurde, gilt die Klausel als „erfüllt“.

Ausgabe: `Entscheidung = ja`, falls F erfüllbar ist,
`Entscheidung = nein` sonst.

```
PROCEDURE Erfuellbarkeit_2CSAT (F                : KNF_typ;
                                VAR Entscheidung:Entscheidungs_typ);
VAR C          : SET OF KNF_typ;
```

```
    V          : SET OF Literal_typ;
    x          : Literal_typ;
    firstguess : BOOLEAN;

BEGIN { Erfuellbarkeit_2CSAT }
  { F = F_1 ∧ ... ∧ F_m }
  C := {F_1, ..., F_m};
  erkläre alle Klauseln in C als unerfüllt;
  V := Menge der in F vorkommenden Variablen;
  erkläre alle Variablen in V als nicht zugewiesen;

  WHILE (V enthält eine Variable x) DO
    BEGIN
      x          := TRUE;
      firstguess := TRUE;
      WHILE (C enthält eine unerfüllte Klausel F_i = (y_i1 ∨ y_i2), wobei mindestens einem
             Literal ein Wahrheitswert zugewiesen ist) DO
        BEGIN
          IF (y_i1 = TRUE) OR (y_i2 = TRUE)
          THEN erkläre F_i als erfüllt
          ELSE IF (y_i1 = FALSE) AND (y_i2 = FALSE)
               THEN BEGIN
                      IF NOT firstguess
                      THEN BEGIN
                             Entscheidung := nein;
                             Exit
                           END
                      ELSE BEGIN
                             erkläre alle Klauseln in C als unerfüllt;
                             erkläre alle Variablen in V als nicht zugewiesen;
                             x          := FALSE;
                             firstguess := FALSE;
                           END;
                    END
               ELSE BEGIN
                      IF y_i1 = FALSE
                      THEN y_i2 := TRUE
                      ELSE y_i1 := TRUE;
                      erkläre F_i als erfüllt;
                    END;
        END { WHILE C enthält eine unerfüllte Klausel };
      entferne aus C die erfüllten Klauseln;
```

```
        entferne aus V die Variablen, denen ein Wahrheitswert zugewiesen wurde;
    END { WHILE V enthält eine Variable x};
  Entscheidung := ja;

END   { Erfuellbarkeit_2CSAT };
///
```

Für das zu 2-CSAT analoge Problem 3-CSAT der Erfüllbarkeit, bei dem jede Klausel genau 3 Literale enthält, ist kein polynomielles Entscheidungsverfahren bekannt. Es wird vermutet, daß es auch kein derartiges Verfahren gibt. Natürlich liegt 3-CSAT in **NP**.

Im folgenden werden einige wenige **Beispiele für Probleme** Π **aus NP** aufgeführt, von denen nicht bekannt ist , ob sie in **P** liegen. Dabei wird für Instanzen $x \in \Sigma_{\Pi}^{*}$ jeweils der Beweis $B_x \in \Sigma_0^{*}$ angegeben, den der Verifizierer zur `ja/nein`-Entscheidung heranzieht. Die Angabe von B_x erfolgt informell, sie muß entsprechend (beispielsweise in eine binäre Zeichenkette) übersetzt werden. Mit $size(x)$ wird die Größe einer Eingabeinstanz angegeben.

- Erfüllbarkeitsproblem für Boolesche Ausdrücke in konjunktiver Normalform (CSAT) bzw. Boolescher Ausdrücke in allgemeiner Form (SAT):
  Instanz: F ist ein Boolescher Ausdruck in konjunktiver Normalform bzw. in allgemeiner Form mit der Variablenmenge $V = \{x_1, ..., x_n\}$; $size(F) = n$
  Beweis: B_F ist eine 0-1-Folge der Länge n
  Arbeitsweise des Verifizierers: Der i-te Wert in B_F wird als Belegung von x_i interpretiert, und zwar wird der Wert 0 in `FALSE` und der Wert 1 in `TRUE` übersetzt. Die so entstehende Belegung der Variablen wird in F eingesetzt und die Formel ausgewertet. Falls sich bei dieser Auswertung von F der Wert `TRUE` ergibt, wird `ja` ausgegeben, ansonsten `nein`.

- Problem des Handlungsreisenden:
  Instanz: $[G,K]$, $G = (V,E,w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w : E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein Gewicht; $K \in \mathbf{R}_{>0}$; $size([G,K]) = k = n \cdot A$ mit
  $A = \max(\{\lfloor \log_2(w(e)) \rfloor + 1 \mid e \in E\} \cup \{\lfloor \log_2(K) \rfloor + 1\})$ (für $w(e) = 0$ wird anstelle von $\lfloor \log_2(|w(e)|) \rfloor + 1$ der Wert 1 genommen)
  Beweis: $B_{[G,K]}$ ist eine Permutation der Zahlen 1, ..., n in Binärkodierung (mit Länge in $O(n \cdot \log(n))$)

Arbeitsweise des Verifizierers: Überprüfung, ob die Permutation $B_{[G,K]}$ eine Rundreise in G beschreibt, deren Gewicht $\leq K$ ist. In diesem Fall wird `ja` ausgegeben, ansonsten `nein`. Mit Binärsuche läßt sich sogar in polynomieller Zeit überprüfen, ob es eine Rundreise mit minimalen Gewicht $\leq K$ gibt.

- Partitionenproblem:

  Instanz: $I = \{a_1, ..., a_n\}$

  I ist eine Menge von n natürlichen Zahlen mit $a_i > 0$ für $i = 1, ..., n$; $size(I) = n \cdot A$ mit $A = \max\{\lfloor \log_2(a_i) \rfloor + 1 \mid i = 1, ..., n\}$

  Beweis: B_I ist eine Teilmenge von $\{1, ..., n\}$ in Binärkodierung (mit Länge in $O(n \cdot \log(n))$)

  Arbeitsweise des Verifizierers: Überprüfung, ob $\sum_{j \in B_I} a_j = \sum_{j \notin B_I} a_j$ gilt. In diesem Fall wird `ja` ausgegeben, ansonsten `nein`.

- 0/1-Rucksackproblem:

  Instanz: $I = (A, M)$

  $A = \{a_1, ..., a_n\}$ ist eine Menge von n natürlichen Zahlen mit $a_i > 0$ für $i = 1, ..., n$; $M \in \mathbf{N}$, $M > 0$; $size(I) = n \cdot A$ mit

  $A = \max(\{\lfloor \log_2(a_i) \rfloor + 1 \mid i = 1, ..., n\} \cup \{\lfloor \log_2(M) \rfloor + 1\})$

  Beweis: B_I ist eine 0-1-Folge $x_1, ..., x_n$ (mit Länge n)

  Arbeitsweise des Verifizierers: Überprüfung, ob $\sum_{i=1}^{n} x_i \cdot a_i = M$ gilt. In diesem Fall wird `ja` ausgegeben, ansonsten `nein`.

- Problem der $\{0,1\}$-Linearen Programmierung

  Instanz: $[A, \vec{b}, \vec{c}, K]$, $A \in \mathbf{Z}^{m \cdot n}$ ist eine ganzzahlige Matrix mit m Zeilen und n Spalten, $\vec{b} \in \mathbf{Z}^m$ ist ein ganzzahliger Vektor, $\vec{c} \in \mathbf{N}^n$ ist ein nichtnegativer ganzzahliger Vektor, $K \in \mathbf{N}$, $size([A, \vec{b}, \vec{c}]) = k = m \cdot n \cdot B$ mit

  $B = \max\{\lceil \log(e) \rceil \mid e \in A \vee e \in \vec{b} \vee e \in \vec{c}\}$

  Beweis: $B_{[A,\vec{b},\vec{c}]}$ ist eine 0-1-Folge der Länge n (für den Vektor $\vec{x}$)

  Arbeitsweise des Verifizierers: Überprüfung, ob für $\vec{x} = B_{[A,\vec{b},\vec{c}]}$ die Bedingungen $A \cdot \vec{x} \geq \vec{b}$ und $\sum_{i=1}^{n} c_i \cdot x_i \leq K$ gelten. In diesem Fall wird `ja` ausgegeben, ansonsten `nein`.

- Problem des längsten Wegs in einem gewichteten Graphen:
  Instanz: $\lfloor G, v_i, v_j, K \rfloor$, $G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; $v_i \in V; v_j \in V$; die Funktion $w: E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein Gewicht; $K \in \mathbf{R}_{>0}$; $size(\lfloor G, K \rfloor) = k = n \cdot A$ mit $A = \max(\{\lfloor \log_2(w(e)) \rfloor + 1 \mid e \in E\} \cup \{\lfloor \log_2(K) \rfloor + 1\})$ (für $w(e) = 0$ wird anstelle von $\lfloor \log_2(|w(e)|) \rfloor + 1$ der Wert 1 genommen)
  Beweis: $B_{[G, v_i, v_j, K]}$ ist eine Folge $i_1, i_2, ..., i_k$ von natürlichen Zahlen aus $\{1, ..., n\}$ (mit Länge in $O(n \cdot \log(n))$
  Arbeitsweise des Verifizierers: Überprüfung, ob $v_{i_1}, v_{i_2}, ..., v_{i_k}$ einen einfachen Weg (d.h. ohne Knotenwiederholungen) von v_{i_1} nach v_{i_k} mit Gewicht $\geq K$ beschreibt. In diesem Fall wird `ja` ausgegeben, ansonsten `nein`.

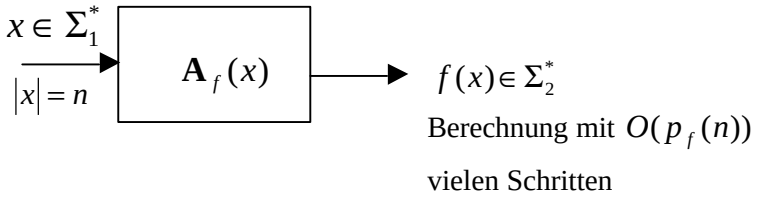
Man kennt heute mehrere tausend Probleme aus **NP**. In der angegebenen Literatur werden geordnet nach Anwendungsgebieten umfangreiche Listen von **NP**-Problemen aufgeführt.

5.5 NP-Vollständigkeit

Die bisher beschriebenen Probleme entstammen verschiedenen Anwendungsgebieten. Dementsprechend unterscheiden sich die Alphabete, mit denen man die jeweiligen Instanzen bildet. Boolesche Ausdrücke werden mit einem anderen Alphabet kodiert als gewichtete Graphen oder Instanzen für das Partitionenproblem. Selbstverständlich lassen sich letztlich alle Probleme mit Hilfe des Alphabets $\{0,1\}$ kodieren, so daß man mit einem einzigen Alphabet auskommen könnte. Aber selbst, wenn die Eingabeinstanzen unterschiedlicher Probleme mit dem Alphabet $\{0,1\}$ kodiert werden, weisen die dann so kodierten Bestandteile der betrachteten Objekte wie Graphen, Boolesche Variablen oder Zahlen immer noch grundlegend verschiedene Eigenschaften auf, so daß man weiterhin davon ausgehen kann, daß man unterschiedliche Anwendungen mit Hilfe unterschiedlicher Alphabete kodiert. Im folgenden wird eine Verbindung zwischen den unterschiedlichen Problemen und ihren zugrundeliegenden Alphabeten hergestellt.

Eine Funktion $f: \Sigma_1^* \to \Sigma_2^*$, die Wörter über dem endlichen Alphabet Σ_1 auf Wörter über dem endlichen Alphabet Σ_2 abbildet, heißt **durch einen deterministischen Algorithmus in polynomieller Zeit berechenbar**, wenn gilt: Es gibt einen deterministischen Algorithmus

$\mathbf{A}_f$ mit Eingabemenge Σ_1^* und Ausgabemenge Σ_2^* und ein Polynom p_f mit folgenden Eigenschaften:
bei Eingabe von $x \in \Sigma_1^*$ mit der Größe $|x| = n$ erzeugt der Algorithmus die Ausgabe $f(x) \in \Sigma_2^*$ und benötigt dazu höchstens $O(p_f(n))$viele Schritte.



Es seien $L_1 \subseteq \Sigma_1^*$ und $L_2 \subseteq \Sigma_2^*$ zwei Mengen aus Zeichenketten (Wörtern) über jeweils zwei endlichen Alphabeten. L_1 heißt **polynomiell (many-one) reduzierbar** auf L_2, geschrieben
$L_1 \leq_m^p L_2$,
wenn gilt: Es gibt eine Funktion $f : \Sigma_1^* \to \Sigma_2^*$, die durch einen deterministischen Algorithmus in polynomieller Zeit berechenbar ist und für die gilt:
$x \in L_1 \Leftrightarrow f(x) \in L_2$.

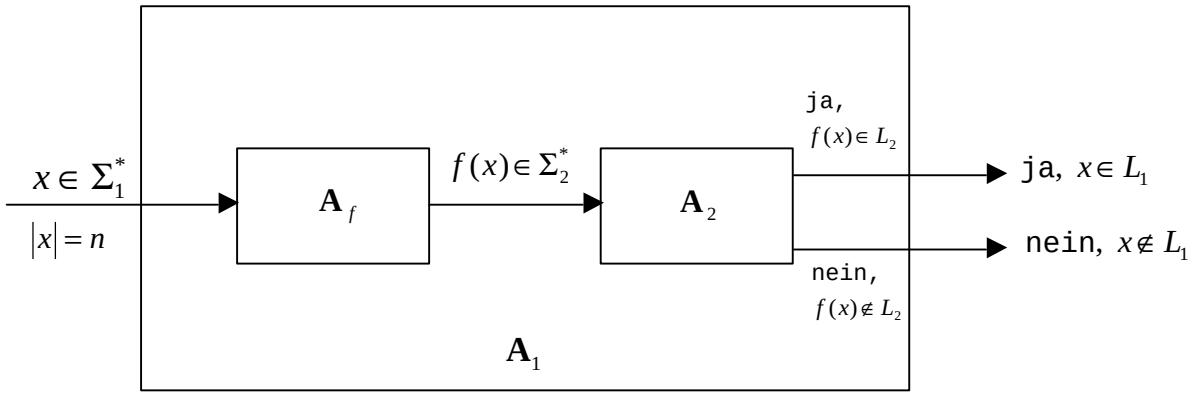
Diese Eigenschaft kann auch so formuliert werden:
$x \in L_1 \Rightarrow f(x) \in L_2$ und $x \notin L_1 \Rightarrow f(x) \notin L_2$.

Bemerkung: Es gibt noch andere Formen der Reduzierbarkeit zwischen Mengen, z.B. die allgemeinere Form der Turing-Reduzierbarkeit mit Hilfe von Orakel-Turingmaschinen.

Die **Bedeutung der Reduzierbarkeit** $\leq_m^p$ zeigt folgende Überlegung.

Für die Mengen $L_1 \subseteq \Sigma_1^*$ und $L_2 \subseteq \Sigma_2^*$ gelte $L_1 \leq_m^p L_2$ mittels der Funktion $f : \Sigma_1^* \to \Sigma_2^*$ bzw. des Algorithmus $\mathbf{A}_f$. Seine Zeitkomplexität sei das Polynom p_f. Für die Menge L_2 gebe es einen polynomiell zeitbeschränkten deterministischen Algorithmus $\mathbf{A}_2$, der L_2 erkennt; seine Zeitkomplexität sei das Polynom p_2 :

Mit Hilfe von $\mathbf{A}_f$ und $\mathbf{A}_2$ läßt sich durch Hintereinanderschaltung beider Algorithmen ein polynomiell zeitbeschränkter deterministischer Algorithmus $\mathbf{A}_1$ konstruieren, der L_1 erkennt:



Erkennung von L_1 mit Zeitaufwand der Größenordnung $O\left(p_f(n) + p_2\left(p_f(n)\right)\right)$, d.h. polynomiellem Zeitaufwand

Ein $x_2 \in \Sigma_2^*$ kann dazu dienen, für mehrere (*many*) $x \in \Sigma_1^*$ die Frage „$x \in L_1$?“ zu entscheiden (nämlich für alle diejenigen $x \in \Sigma_1^*$, für die $f(x) = x_2$ gilt). Allerdings darf man für jede Eingabe $f(x)$ den Algorithmus $\mathbf{A}_2$ nur einmal (*one*) verwenden, nämlich bei der Entscheidung von $f(x)$.

Gilt für ein Problem Π_0 zur Entscheidung der Menge $L_{\Pi_0} \subseteq \Sigma_{\Pi_0}^*$ und für ein Problem Π zur Entscheidung der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ die Relation $L_\Pi \leq_m^p L_{\Pi_0}$, so heißt das Problem Π auf das Problem Π_0 **polynomiell (many-one) reduzierbar**, geschrieben $\Pi \leq_m^p \Pi_0$.

Beispiel:

$\{0,1\}$-Lineare Programmierung:

Instanz: $\left[A, \vec{b}, \vec{z}\right]$

$A \in \mathbf{Z}^{m \cdot n}$ ist eine ganzzahlige Matrix mit *m* Zeilen und *n* Spalten, $\vec{b} \in \mathbf{Z}^m$, $\vec{z}$ ist ein Vektor von *n* Variablen, die die Werte 0 oder 1 annehmen können.

Lösung: Entscheidung „ja“, falls gilt:
Es gibt eine Zuweisung der Werte 0 oder 1 an die Variablen in $\vec{z}$, so daß das lineare Ungleichungssystem $A \cdot \vec{z} \leq / \geq \vec{b}$ erfüllt ist. Die Bezeichnung $\leq / \geq$ steht hier für entweder $\leq$ oder $\geq$ in einer Ungleichung.

Es gilt CSAT $\leq_m^p$ $\{0,1\}$-Lineare Programmierung.

Zum Nachweis ist eine in polynomieller Zeit berechnenbare Funktion *f* anzugeben, die jedem Booleschen Ausdruck *F* in konjunktiver Normalform eine Instanz von $\{0,1\}$-Lineare Programmierung zuordnet, so daß *F* genau dann erfüllbar ist, wenn es eine 0-1-Zuweisung an die in $f(F)$ vorkommenden Variablen gibt, die das durch $f(F)$ beschriebene lineare Ungleichungssystem erfüllt.

Es sei $F = F_1 \wedge ... \wedge F_m$ ein Boolescher Ausdruck in konjunktiver Normalform. Die in *F* vorkommende Variablenmenge sei $V = \{x_1, ..., x_n\}$. Jedes F_i hat die Form $F_i = (y_{i_1} \vee ... \vee y_{i_k})$, wobei y_{i_j} für eine Variable (d.h. $y_{i_j} = x_l$) oder für eine negierte Variable (d.h. $y_{i_j} = \neg x_l$) steht. Die Instanz $f(F) = [A, \vec{b}, \vec{z}]$ wird definiert durch $\vec{z} = \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix}$, $A \in \mathbf{Z}^{m \cdot n}$ und $\vec{b} \in \mathbf{Z}^m$. Hierbei ergeben sich *A* und $\vec{b}$ wie folgt: Für jede Klausel $F_i = (y_{i_1} \vee ... \vee y_{i_k})$ wird eine Ungleichung $t_{i_1} + ... + t_{i_k} \geq 1$ formuliert. Ist das Literal y_{i_j} in F_i eine Variable, etwa $y_{i_j} = x_l$, dann ist $t_{i_j} = z_l$; ist das Literal y_{i_j} in F_i eine negierte Variable, etwa $y_{i_j} = \neg x_l$, dann ist $t_{i_j} = 1 - z_l$. Alle Konstanten 1 in $t_{i_1} + ... + t_{i_k} \geq 1$ werden auf die rechte Seite der Ungleichung gebracht. Der nach dieser Umformung auf der rechten Seite entstehende Wert ist die Komponente b_i im Vektor $\vec{b}$, die *i*-te Zeile von *A* ergibt sich aus den Faktoren auf der linken Seite der Ungleichung. Offensichtlich ist $f(F)$ in polynomieller Zeit konstruierbar.

Es sei eine Belegung der Variablen in *V* gegeben, die den Booleschen Ausdruck *F* erfüllt. Dann wird das Ungleichungssystem $f(F) = [A, \vec{b}, \vec{z}]$ durch die Wertzuweisung $z_l = \begin{cases} 1 & \text{für } x_l = \mathsf{TRUE} \\ 0 & \text{für } x_l = \mathsf{FALSE} \end{cases}$ erfüllt. Ist umgekehrt eine Wertzuweisung an die Variablen in $f(F) = [A, \vec{b}, \vec{z}]$ gegeben, so daß das Ungleichungssystem erfüllt ist, so erfüllt die Belegung $x_l = \begin{cases} \mathsf{TRUE} & \text{für } z_l = 1 \\ \mathsf{FALSE} & \text{für } z_l = 0 \end{cases}$ den Booleschen Ausdruck *F*.

Einige leicht nachzuweisende Eigenschaften (Übungsaufgabe) der polynomiellen many-one-Reduzierbarkeit faßt der nächste Satz zusammen.

Satz 5.5-1:

Es seien $L \subseteq \Sigma^*$, $L_1 \subseteq \Sigma_1^*$, $L_2 \subseteq \Sigma_2^*$ und $L_3 \subseteq \Sigma_3^*$ Sprachen über endlichen Alphabeten. Dann gilt:

1. $L \leq_m^p L$, d.h. die Relation $\leq_m^p$ ist reflexiv.
2. Aus $L_1 \leq_m^p L_2$ und $L_2 \leq_m^p L_3$ folgt $L_1 \leq_m^p L_3$, d.h. die Relation $\leq_m^p$ ist transitiv.
3. Gilt $L_1 \leq_m^p L_2$ und $L_2 \in \mathbf{P}$, dann ist $L_1 \in \mathbf{P}$.
4. Gilt $L_1 \leq_m^p L_2$ und $L_2 \in \mathbf{NP}$, dann ist $L_1 \in \mathbf{NP}$.

Eine der zentralen Definitionen in der Angewandten Komplexitätstheorie ist die **NP**-Vollständigkeit:

Ein Entscheidungsproblem Π_0 zur Entscheidung (Akzeptanz, Erkennung) der Menge $L_{\Pi_0} \subseteq \Sigma_{\Pi_0}^*$ heißt **NP-vollständig**, wenn gilt:

(i) $L_{\Pi_0} \in \mathbf{NP}$

(ii) für jedes Problem Π zur Entscheidung (Akzeptanz, Erkennung) der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ mit $L_\Pi \in \mathbf{NP}$ gilt $L_\Pi \leq_m^p L_{\Pi_0}$.

Mit obiger Definition der Reduzierbarkeit zwischen Problemen kann man auch sagen:

Das Entscheidungsproblem Π_0 ist **NP**-vollständig, wenn Π_0 in **NP** liegt und für jedes Entscheidungsprobleme Π in **NP** die Relation $\Pi \leq_m^p \Pi_0$ gilt.

Das erste Beispiel eines **NP**-vollständigen Problems wurde 1971 von Stephen Cook gefunden. Es gilt:

Satz 5.5-2:

Das Erfüllbarkeitsproblem für Boolesche Ausdrücke in allgemeiner Form (SAT) ist **NP**-vollständig.

Beweis:

Da es sich bei dieser Aussage um den zentralen Satz der angewandten Komplexitätstheorie handelt, soll die Beweisidee skizziert werden:

1. Die zu SAT gehörige Sprache ist $L_{\text{SAT}} = \{F \mid F \text{ ist ein erfüllbarer Boolescher Ausdruck }\}$.

Bezeichnet Σ^*_{BOOLE} die Menge {F | F ist ein Boolescher Ausdruck}, dann ist $L_{\text{SAT}} \subseteq \Sigma^*_{\text{BOOLE}}$. In Kapitel 5.3 wird gezeigt, daß SAT in **NP** liegt.

2. Für jedes Problem Π mit der Menge $L_\Pi \subseteq \Sigma^*_\Pi$ in **NP** ist die Relation $L_\Pi \leq^p_m L_{\text{SAT}}$ zu zeigen. Die einzige Eigenschaft, die bezüglich L_Π in einem Beweis genutzt werden kann, ist die Tatsache, daß es eine 1-NDTM TM_{L_Π} gibt, die bei Eingabe eines Wortes $x \in \Sigma^*_\Pi$ entscheidet, ob $x \in L_\Pi$ gilt oder nicht. Diese Entscheidung wird in $p_{L_\Pi}(|x|)$ vielen Schritten getroffen, wobei p_{L_Π} ein Polynom mit $p_{L_\Pi}(n) \geq n$ ist. Ist $x \in L_\Pi$, dann stoppt TM_{L_Π} im akzeptierenden Zustand q_{accept}. Ist $x \notin L_\Pi$, dann stoppt TM_{L_Π} in einem Zustand $q \neq q_{accept}$. TM_{L_Π} besucht dabei höchstens die Zellen mit den Nummern $-p_{L_\Pi}(|x|)$, ..., 0, 1, ..., $p_{L_\Pi}(|x|)+1$; hierbei wird das Nichtstandardmodell einer nichtdeterministischen Turingmaschine genommen (vgl. Kapitel 5.3).
   Zum Nachweis von $L_\Pi \leq^p_m L_{\text{SAT}}$ ist eine Funktion $f_\Pi : \Sigma^*_\Pi \to \Sigma^*_{\text{BOOLE}}$ anzugeben, die die Eigenschaft „$x \in L_\Pi \Leftrightarrow f_\Pi(x)$ ist erfüllbar“ besitzt ($f_\Pi(x)$ ist ein Boolescher Ausdruck); außerdem muß $f_\Pi(x)$ in $p_{f_\Pi}(|x|)$ vielen Schritten deterministisch berechenbar (konstruierbar) sein mit einem Polynom p_{f_Π}. Im folgenden wird beschrieben, wie $f_\Pi(x)$ aus x erzeugt werden kann. Der Boolesche Ausdruck $f_\Pi(x)$ beschreibt im wesentlichen, wie eine Berechnung von TM_{L_Π} bei Eingabe von $x \in \Sigma^*_\Pi$ abläuft.

   Die Zustandsmenge von TM_{L_Π} sei $Q = \{q_0, ..., q_k\}$, das Arbeitsalphabet von TM_{L_Π} sei $\Sigma_\Pi = \{a_0, ..., a_l\}$. Auf dem Eingabeband stehe das Wort $x \in \Sigma^*_\Pi$, $x = x_1 ... x_n$ mit $x_i \in \Sigma_\Pi$ für i = 1, ..., n. Es wird eine Reihe Boolescher Variablen erzeugt, deren Interpretation folgender Tabelle zu entnehmen ist:

Variable	Indizes	Interpretation
$zust_{t,q}$	$t = 0, ..., p_{L_\Pi}(n)$ $q \in Q$	$zust_{t,q} = \mathsf{TRUE}$ genau dann, wenn sich TM_{L_Π} im Schritt t im Zustand q befindet Anzahl an Variablen $zust_{t,q}$: $(k+1)\cdot(p_{L_\Pi}(n)+1)$
$pos_{t,i}$	$t = 0, ..., p_{L_\Pi}(n)$ $i = -p_{L_\Pi}(n), ..., p_{L_\Pi}(n)+1$	$pos_{t,i} = \mathsf{TRUE}$ genau dann, wenn sich der Schreib/Lesekopf von TM_{L_Π} im Schritt t über der Zelle mit Nummer i befindet Anzahl an Variablen $pos_{t,i}$: $2\cdot(p_{L_\Pi}(n)+1)^2$
$band_{t,i,a}$	$t = 0, ..., p_{L_\Pi}(n)$ $i = -p_{L_\Pi}(n), ..., p_{L_\Pi}(n)+1,$ $a \in \Sigma_\Pi$	$band_{t,i,a} = \mathsf{TRUE}$ genau dann, wenn sich im Schritt t in der Zelle mit Nummer i das Zeichen a befindet Anzahl an Variablen $band_{t,i,a}$: $2(l+1)(p_{L_\Pi}(n)+1)^2$

Der zu konstruierende Boolesche Ausdruck $f_\Pi(x)$ besteht aus mehreren Teilen und enthält insbesondere mehrmals eine Teilformeln G, die genau dann den Wahrheitswert TRUE erhält, wenn genau eine der in G vorkommenden Variablen den Wahrheitswert TRUE trägt. Sind $y_1, ..., y_m$ die in G vorkommenden Variablen, so lautet G:

$$G = G(y_1, ..., y_m) = \left(\bigvee_{i=1}^{m} y_i\right) \wedge \left(\bigwedge_{j=1}^{m-1} \bigwedge_{i=j+1}^{m} \neg(y_j \wedge y_i)\right)$$
$$= \left(\bigvee_{i=1}^{m} y_i\right) \wedge \left(\bigwedge_{j=1}^{m-1} \bigwedge_{i=j+1}^{m} (\neg y_j \vee \neg y_i)\right).$$

Die zweite Zeile zeigt, daß $G = G(y_1, ..., y_m)$ in konjunktiver Normalform formulierbar ist, ohne die Anzahl an Literalen zu ändern, und m^2 viele Literale enthält.

In $f_\Pi(x)$ kommen mehrere Teilformeln, die gemäß G aufgebaut sind, mit unterschiedlichen Variablen aus obiger Tabelle vor.

$f_\Pi(x)$ hat die Bauart $f_\Pi(x) = R \wedge A \wedge \ddot{U}_1 \wedge \ddot{U}_2 \wedge E$. Die Teilformel R beschreibt Randbedingungen, A Anfangsbedingungen, $\ddot{U}_1$ und $\ddot{U}_2$ beschreiben Übergangsbedingungen, und E beschreibt Endebedingungen.

In R wird ausgedrückt, daß TM_{L_Π} zu jedem Zeitpunkt t in genau einem Zustand q ist, daß der Schreib/Lesekopf über genau einer Zelle mit einer Nummer i aus dem Intervall von $-p_{L_\Pi}(n)$ bis $p_{L_\Pi}(n)+1$ steht, und daß jede Zelle genau ein Zeichen $a \in \Sigma_\Pi$ enthält:

$$R = \bigwedge_{t=0}^{p_{L_\Pi}(n)} \left[G\left(zust_{t,q_0}, ..., zust_{t,q_k}\right) \wedge G\left(pos_{t,-p_{L_\Pi}(n)}, ..., pos_{t,p_{L_\Pi}(n)+1}\right) \wedge \left(\bigwedge_{i=-p_{L_\Pi}(n)}^{p_{L_\Pi}(n)+1} G\left(band_{t,i,a_0}, ..., band_{t,i,a_l}\right) \right) \right]$$

Die Anzahl an Literalen in *R* beträgt

$$\left(p_{L_\Pi}(n)+1\right)\cdot\left((k+1)^2 + 4\left(p_{L_\Pi}(n)+1\right)^2 + 2\left(p_{L_\Pi}(n)+1\right)(l+1)^2\right) \in O\left(\left(p_{L_\Pi}(n)\right)^3\right).$$

A beschreibt die Situation zum Zeitpunkt $t = 0$ (hierbei ist $b \in \Sigma_\Pi$ das Blankzeichen):

$$A = zust_{0,q_0} \wedge pos_{0,1} \wedge \left(\bigwedge_{i=1}^{n} band_{0,i,x_i} \right) \wedge \left(\bigwedge_{i=-p_{L_\Pi}(n)}^{0} band_{0,i,b} \right) \wedge \left(\bigwedge_{i=n+1}^{p_{L_\Pi}(n)+1} band_{0,i,b} \right).$$

A enthält $2\left(p_{L_\Pi}(n)+2\right) \in O\left(p_{L_\Pi}(n)\right)$ viele Literale.

$\ddot{U}_1$ beschreibt den Übergang von der zum Zeitpunkt *t* bestehenden Konfiguration zur Konfiguration zum Zeitpunkt $t + 1$ für $t = 0, ..., p_{L_\Pi}(n)$. Anstelle der Kopfbewegung *L*, *S* bzw. *R* werden hier die Werte $y = -1$, $y = 0$ bzw. $y = 1$ verwendet:

$$\ddot{U}_1 = \bigwedge_{t,q,i,a} \left[zust_{t,q} \wedge pos_{t,i} \wedge band_{t,i,a} \Rightarrow \vee \left\{ zust_{t+1,q'} \wedge pos_{t+1,i+y} \wedge band_{t+1,i,a'} \mid (q',a',y) \in d(q,a) \right\} \right].$$

Die Indizes nehmen die Werte $t = 0, ..., p_{L_\Pi}(n)$, $q \in Q$, $i = -p_{L_\Pi}(n), ..., p_{L_\Pi}(n)+1$ und $a \in \Sigma_\Pi$ an.

Die geschweifte Klammer in $\ddot{U}_1$ enthält für feste Werte *t*, *q*, *a* und *i* höchstens $3(k+1)(l+1)$ viele Literale, in der eckigen Klammer sind es daher höchstens $3(k+1)(l+1)+3$. Insgesamt enthält $\ddot{U}_1$ höchstens

$2\left(p_{L_\Pi}(n)+1\right)^2 \cdot (k+1) \cdot (l+1)\left(3(k+1)(l+1)+3\right) \in O\left(\left(p_{L_\Pi}(n)\right)^2\right)$ viele Literale.

Um zu zeigen, wie sich auch $\ddot{U}_1$ in eine äquivalente Formel in konjunktiver Normalform umformen läßt, soll exemplarisch ein Ausdruck $zust_{t,q} \wedge pos_{t,i} \wedge band_{t,i,a} \Rightarrow \vee \left\{ zust_{t+1,q'} \wedge pos_{t+1,i+y} \wedge band_{t+1,i,a'} \mid (q',a',y) \in d(q,a) \right\}$ innerhalb der eckigen Klammer, der im rechten Teil zwei Alternativen $zust_{t+1,q'} \wedge pos_{t+1,i+y'} \wedge band_{t+1,i,a'}$ und $zust_{t+1,q''} \wedge pos_{t+1,i+y''} \wedge band_{t+1,i,a''}$ enthält, umgeformt werden. Um den Vorgang übersichtlich zu halten, wird dieser Ausdruck in der Form $X_1 \wedge X_2 \wedge X_3 \Rightarrow (Y_1 \wedge Y_2 \wedge Y_3) \vee (Z_1 \wedge Z_2 \wedge Z_3)$ geschrieben. Hier stehen X_i, Y_i und Z_i für jeweils drei Variablen. Es gilt

$$\begin{aligned} X_1 \wedge X_2 \wedge X_3 &\Rightarrow (Y_1 \wedge Y_2 \wedge Y_3) \vee (Z_1 \wedge Z_2 \wedge Z_3) \\ &= (\neg X_1 \vee \neg X_2 \vee \neg X_3) \vee (Y_1 \wedge Y_2 \wedge Y_3) \vee (Z_1 \wedge Z_2 \wedge Z_3) \\ &= (\neg X_1 \vee \neg X_2 \vee \neg X_3 \vee Y_1) \wedge (\neg X_1 \vee \neg X_2 \vee \neg X_3 \vee Y_2) \wedge (\neg X_1 \vee \neg X_2 \vee \neg X_3 \vee Y_3) \\ &\quad \wedge (\neg X_1 \vee \neg X_2 \vee \neg X_3 \vee Z_1) \wedge (\neg X_1 \vee \neg X_2 \vee \neg X_3 \vee Z_2) \wedge (\neg X_1 \vee \neg X_2 \vee \neg X_3 \vee Z_3). \end{aligned}$$

Wie man sieht, werden für jedes Y_i und Z_i vier Literale notiert, so daß sich für die Anzahl der Literale in der äquivalenten Formel innerhalb eines Ausdrucks in der eckigen Klammer im allgemeinen Fall eine Obergrenze von $12(k+1)(l+1)$ angeben läßt. Daher

bleibt die Anzahl der Literale in $\ddot{U}_1$, selbst in der konjunktiven Normalform, von der Ordnung $O\left(\left(p_{L_\Pi}(n)\right)^2\right)$.

$\ddot{U}_2$ besagt, daß sich der Inhalt von Zellen, über denen der Schreib/Lesekopf nicht steht, nicht ändert:
$\ddot{U}_2 = \underset{t,i,a}{\wedge}\left[\left(\neg pos_{t,i} \wedge band_{t,i,a}\right) \Rightarrow band_{t+1,i,a}\right]$.
Hierbei nehmen die Indizes die Werte $t = 0, ..., p_{L_\Pi}(n)$, $i = -p_{L_\Pi}(n), ..., p_{L_\Pi}(n)+1$ und $a \in \Sigma_\Pi$ an.
Die Anzahl an Literalen in $\ddot{U}_2$ beträgt $6\left(p_{L_\Pi}(n)+1\right)^2 \cdot (l+1) \in O\left(\left(p_{L_\Pi}(n)\right)^2\right)$.
$\ddot{U}_2$ läßt sich in eine äquivalente Formel in konfunktiver Normalform umformen, ohne die Anzahl an Literalen zu ändern:

$$\ddot{U}_2 = \underset{t,i,a}{\wedge}\left[\left(\neg pos_{t,i} \wedge band_{t,i,a}\right) \Rightarrow band_{t+1,i,a}\right]$$
$$= \underset{t,i,a}{\wedge}\left[\left(pos_{t,i} \vee \neg band_{t,i,a} \vee band_{t+1,i,a}\right)\right].$$

E prüft nach, ob der Endzustand q_{accept} zum Zeitpunkt $t = p_{L_\Pi}(n)$ erreicht ist:
$E = zust_{p_{L_\Pi}(n),q_{accept}}$.

Insgesamt enthält $f_\Pi(x)$ eine Anzahl von Literalen der Ordnung $O\left(\left(p_{L_\Pi}(n)\right)^3\right)$, d.h. einer in der Länge des Eingabewortes polynomiellen Ordnung. Werden diese Literale durchnumeriert, so daß man $O\left(\left(p_{L_\Pi}(n)\right)^3\right)$ viele Literalpositionen bekommt, und dann binär kodiert (jede Zahl hat dann eine Länge der Ordnung $O\left(\log\left(\left(p_{L_\Pi}(n)\right)^3\right)\right) = O\left(\log\left(p_{L_\Pi}(n)\right)\right)$), so hat $f_\Pi(x)$ eine Länge der Ordnung $O\left(\left(p_{L_\Pi}(n)\right)^4\right)$. Daher ist $f_\Pi(x)$ bei Vorgabe von $x \in \Sigma_\Pi^*$ (für festes Π) deterministisch in polynomieller Zeit berechenbar.

Es läßt sich leicht nachweisen, daß die Eigenschaft „$x \in L_\Pi \Leftrightarrow f_\Pi(x)$ ist erfüllbar“ gilt. ///

Die Beweisskizze zeigt sogar:

Satz 5.5-3:
Das Erfüllbarkeitsproblem für Boolesche Ausdrücke in konjunktiver Normalform (CSAT) ist **NP**-vollständig.

NP-vollständige Probleme kann man innerhalb der Klasse **NP** als die am schwersten zu lösenden Probleme betrachten, denn sie entscheiden die **P**-**NP**-Frage:

Satz 5.5-4:

Gibt es mindestens ein **NP**-vollständiges Problem, das in **P** liegt , so ist **P** = **NP**.

Beweis:

Wegen $\mathbf{P} \subseteq \mathbf{NP}$ ist die umgekehrte Inklusion $\mathbf{NP} \subseteq \mathbf{P}$ zu zeigen. Es sei L_0 ein **NP**-vollständiges Problem, das in **P** liegt. Es sei $L \in \mathbf{NP}$. Zu zeigen ist $L \in \mathbf{P}$. Da L_0 **NP**-vollständig ist, gilt $L \leq_m^p L_0$. Mit Satz 5.5-1 folgt (wegen der Annahme $L_0 \in \mathbf{P}$) $L \in \mathbf{P}$. ///

Aus **NP**-vollständigen Problemen lassen sich aufgrund der Transitivität der $\leq_m$-Relation (Satz 5.5-1) weitere **NP**-vollständige Probleme ableiten:

Satz 5.5-5:

Ist das Problem zur Entscheidung einer Menge $L_0 \subseteq \Sigma_0^*$ **NP**-vollständig und ist $L_0 \leq_m^p L_1$ für eine Menge $L_1 \subseteq \Sigma_1^*$ so gilt:
Ist L_1 in **NP**, so ist L_1 ebenfalls **NP**-vollständig.

Satz 5.5-6:

Bei $\mathbf{P} \neq \mathbf{NP}$ gilt:

Ist ein Entscheidungsproblem Π zur Entscheidung der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ **NP**-vollständig, so ist das Problem Π schwer lösbar (intractable), d.h. es gibt keinen polynomiell zeitbeschränkten deterministischen Lösungsalgorithmus zur Entscheidung von L_Π. Insbesondere ist das zugehörige Optimierungsproblem, falls es ein solches gibt, erst recht schwer (d.h. nur mit mindestens exponentiellem Aufwand) lösbar.

NP-vollständige Probleme mit praktischer Relevanz sind heute aus vielen Gebieten bekannt, z.B. aus der Graphentheorie, dem Netzwerk-Design, der Theorie von Mengen und Partitionen, der Datenspeicherung, dem Scheduling, der Maschinenbelegung, der Personaleinsatzplanung, der mathematischen Programmierung und Optimierung, der Analysis und Zahlentheorie, der Kryptologie, der Logik, der Automatentheorie und der Theorie Formaler Sprachen, der Programm- und Codeoptimierung usw. (siehe Literatur). Die folgende Zusammenstellung listet einige wenige Beispiele auf, die z.T. bereits behandelt wurden.

Beispiele für NP-vollständige Entscheidungsprobleme

- **Erfüllbarkeitsproblem der Aussagenlogik (SAT):**

  Instanz: *F*,
  F ist ein Boolescher Ausdruck (Formel der Aussagenlogik)

  Lösung: Entscheidung „ja“, falls gilt:
  F ist erfüllbar, d.h. es gibt eine Belegung der Variablen in *F* mit Werten `TRUE` bzw. `FALSE`, wobei gleiche Variablen mit gleichen Werten belegt werden, so daß sich bei Auswertung der Formel *F* der Wert `TRUE` ergibt.

  Kodiert man Formeln der Aussagenlogik über dem Alphabet $A = \{\wedge, \vee, \neg, (,), x, 0, 1\}$, so ist eine Formel der Aussagenlogik ein Wort über dem Alphabet *A*. Nicht jedes Wort über dem Alphabet *A* ist eine Formel der Aussagenlogik (weil es eventuell syntaktisch nicht korrekt ist) und nicht jede Formel der Aussagenlogik als Wort über dem Alphabet *A* ist erfüllbar.
  $L_{\text{SAT}} = \{F \mid F \text{ ist eine } \textbf{erfüllbare} \text{ Formel der Aussagenlogik}\}$.

- **Erfüllbarkeitsproblem der Aussagenlogik mit Formeln in konjunktiver Normalform mit höchstens 3 Literalen pro Klausel (3-CSAT):**

  Instanz: *F*,
  F ist eine Formel der Aussagenlogik in konjunktiver Normalform mit höchstens 3 Literalen pro Klausel, d.h. sind $x_1, ..., x_n$ die verschiedenen in *F* vorkommenden Booleschen Variablen, so hat *F* die Form
  $F = F_1 \wedge F_2 \wedge ... F_m$
  Hierbei hat jedes F_i die Form $F_i = (y_{i_1} \vee y_{i_2} \vee y_{i_3})$ oder $F_i = (y_{i_1} \vee y_{i_2})$ oder $F_i = (y_{i_1})$, und y_{i_j} steht für eine Boolesche Variable (d.h. $y_{i_j} = x_l$) oder für eine negierte Boolesche Variable (d.h. $y_{i_j} = \neg x_l$) oder für eine Konstante 0 (d.h. $y_{i_j} = 0$) bzw. 1 (d.h. $y_{i_j} = 1$).

  Lösung: Entscheidung „ja“, falls gilt:
  F ist erfüllbar.

  Kodiert man die Formeln wie bei SAT, so gilt $L_{\text{3-CSAT}} \subseteq L_{\text{SAT}}$.

- **Kliquenproblem (KLIQUE):**

  Instanz: $[G,k]$,
  $G=(V,E)$ ist ein ungerichteter Graph und *k* eine natürliche Zahl.

  Lösung: Entscheidung „ja“, falls gilt:
  G besitzt eine „Klique“ der Größe *k*. Dieses ist eine Teilmenge $V' \subseteq V$ der Knotenmenge mit $|V'| = k$, und für alle $u \in V'$ und alle $v \in V'$ mit $u \neq v$ gilt $(u,v) \in E$.

- **0/1-Rucksack-Entscheidungsproblem (RUCKSACK):**

  Instanz: $[a_1, \ldots, a_n, b]$,
  $a_1, \ldots, a_n$ und *b* sind natürliche Zahlen.

  Lösung: Entscheidung „ja“, falls gilt:
  Es gibt eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} a_i = b$.

- **Partitionenproblem (PARTITION):**

  Instanz: $[a_1, \ldots, a_n]$,
  $a_1, \ldots, a_n$ sind natürliche Zahlen.

  Lösung: Entscheidung „ja“, falls gilt:
  Es gibt eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$.

- **Packungsproblem (BINPACKING):**

  Instanz: $[a_1, \ldots, a_n, b, k]$,
  $a_1, \ldots, a_n$ („Objekte“) sind natürliche Zahlen mit $a_i \leq b$ für *i* = 1, ..., *n*, $b \in \mathbf{N}$ („Behältergröße“), $k \in \mathbf{N}$.

  Lösung: Entscheidung „ja“, falls gilt:
  Die Objekte können so auf *k* Behälter der Füllhöhe *b* verteilt werden, so daß kein Behälter überläuft, d.h. es gibt eine Abbildung $f: \{1, \ldots, n\} \rightarrow \{1, \ldots, k\}$,

so daß für alle $j \in \{1, ..., k\}$ gilt: $\sum_{f(i)=j} a_i \leq b$

- **Problem des Hamiltonschen Kreises in einem gerichteten (bzw. ungerichteten) Graphen (GERICHTETER bzw. UNGERICHTETER HAMILTONKREIS):**

  Instanz: *G*,
  $G = (V, E)$ ist ein gerichteter bzw. ungerichteter Graph mit $V = \{v_1, ..., v_n\}$.

  Lösung: Entscheidung „ja“, falls gilt:
  G besitzt einen Hamiltonschen Kreis. Dieses ist eine Anordnung $(v_{p(1)}, v_{p(2)}, ..., v_{p(n)})$ der Knoten mittels einer Permutation π der Knotenindizes, so daß für $i = 1, ..., n-1$ gilt: $(v_{p(i)}, v_{p(i+1)}) \in E$ und $(v_{p(n)}, v_{p(1)}) \in E$.

- **Problem des Handlungsreisenden (HANDLUNGSREISENDER):**

  Instanz: $[M, k]$,
  $M = (M_{i,j})$ ist eine $(n \times n)$-Matrix von „Entfernungen“ zwischen *n* „Städten“ und eine Zahl *k*.

  Lösung: Entscheidung „ja“, falls gilt:
  Es gibt eine Permutation π (eine Tour, „Rundreise“), so daß $\sum_{i=1}^{n-1} M_{p(i),p(i+1)} + M_{p(n),p(1)} \leq k$ gilt?

Zum Nachweis der **NP**-Vollständigkeit für eines dieser Probleme Π ist neben der Zugehörigkeit von Π zu **NP** jeweils die Relation $\Pi_0 \leq_m^p \Pi$ zu zeigen, wobei hier Π_0 ein Problem ist, für das bereits bekannt ist, daß es **NP**-vollständig ist, und das eine „ähnliche“ Struktur aufweist. Häufig beweist man

SAT $\leq_m^p$ 3-CSAT $\leq_m^p$ RUCKSACK $\leq_m^p$ PARTITION $\leq_m^p$ BINPACKING,

3-CSAT $\leq_m^p$ KLIQUE und

3-CSAT $\leq_m^p$ GERICHTETER HAMILTONKREIS

$\leq_m^p$ UNGERICHTETER HAMILTONKREIS $\leq_m^p$ HANDUNGSREISENDER.

5.6 Bemerkungen zur Struktur von NP

Es sei **NPC** die Menge der **NP**-vollständigen Sprachen über einem endlichen Alphabet Σ. $\mathbf{NPC} \subseteq \mathbf{NP}$. Unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ gilt $\mathbf{NPC} \cap \mathbf{P} = \emptyset$.

Es gilt folgender Satz:

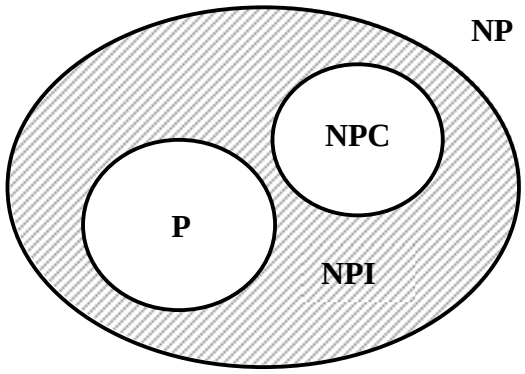
Satz 5.6-1:
Es sei B eine entscheidbare Menge mit $B \notin \mathbf{P}$. Dann gibt es eine Sprache $D \in \mathbf{P}$, so daß $A = D \cap B$ nicht zu **P** gehört, $A \leq_m B$, aber nicht $B \leq_m^p A$ gelten.

Satz 5.6-1 läßt sich folgendermaßen anwenden:

Es sei B eine **NP**-vollständige Sprache. Unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ gilt $B \notin \mathbf{P}$. Die Sprache $A = D \cap B$ gehört zu **NP**, da $D \in \mathbf{P}$ und $B \in \mathbf{NP}$ sind und die Klasse **NP** bezüglich Schnittbildung abgeschlossen ist. Nach dem obigen Satz gilt nicht $B \leq_m^p A$, also ist A nicht **NP**-vollständig. Folglich gilt:

Satz 5.6-2:
Unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ ist $\mathbf{NP} \setminus (\mathbf{P} \cup \mathbf{NPC}) \neq \emptyset$.

Bezeichnet man $\mathbf{NP} \setminus (\mathbf{P} \cup \mathbf{NPC})$ als die Menge **NPI** der **NP-unvollständigen Sprachen**, so zeigt **NP** folgende Struktur:



Die Sprachen in **NPI** liegen bezüglich ihrer Komplexität also zwischen den „leichten“ Sprachen in **P** und den „schweren“ Sprachen in **NPC**. Obwohl es (bei $\mathbf{P} \neq \mathbf{NP}$) unendlich viele Sprachen in **NPI** geben muß, ist die Angabe konkreter Beispiele schwierig. Bisher kennt man kein Problem aus **NPI**. Von dem folgenden Graphenisomorphieproblem ISO wird vermutet, daß es in **NPI** liegt, da bisher (trotz großer Anstrengung) weder der Nachweis für $\mathrm{ISO} \in \mathbf{P}$ noch der Nachweis $\mathrm{ISO} \in \mathbf{NPC}$ gelungen ist.

Graphenisomorphieproblem (ISO):

Instanz: Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$

Lösung: Entscheidung „ja“, falls gilt:
G_1 und G_2 sind isomorph, d.h. es gibt eine Abbildung $f : V_1 \to V_2$ mit der Eigenschaft $(v,w) \in E_1 \Leftrightarrow (f(v), f(w)) \in E_2$.

Lösung: Entscheidung „ja“, falls gilt:
G_1 und G_2 sind isomorph, d.h. es gibt eine Abbildung $f : V_1 \to V_2$ mit der Eigenschaft $(v,w) \in E_1 \Leftrightarrow (f(v), f(w)) \in E_2$.

Beispielsweise sind die folgenden beiden Graphen isomorph:

Die folgende Sprachklasse besteht aus Sprachen, deren Komplemente in **NP** liegen:

$$\textbf{co - NP} = \{ \Sigma^* \setminus L \mid L \text{ ist eine Sprache über } \Sigma \text{ und } L \in \textbf{NP} \}.$$

Für das Primzahlproblem PRIMES kann man zeigen, daß es in **NP** und in **co - NP** liegt (es wird vermutet, daß PRIMES in **P** liegt):

Primzahlproblem (PRIMES):

Instanz: $n \in \mathbf{N}$ in binärer Darstellung, $size(n) = \log(n)$.

Lösung: Entscheidung „ja“, falls n eine Primzahl ist.

Lösung: Entscheidung „ja“, falls n eine Primzahl ist.

Zum Beweis wird ein zahlentheoretischer Satz benötigt, der an dieser Stelle angeführt wird und dessen Beweis in der angegebenen Literatur nachgelesen werden kann:

Satz 5.6-3:

Die Zahl $n \in \mathbf{N}$ mit $n > 2$ ist genau dann eine Primzahl, wenn es eine Zahl $a \in \mathbf{N}$ mit $1 < a \leq n-1$ gibt, die folgende Eigenschaften besitzt:

(i) $a^{n-1} \equiv 1 (\mathrm{mod}\, n)$

(ii) $a^{(n-1)/p} \neq 1 (\mathrm{mod}\, n)$ für jeden Primteiler p von $n-1$.

Satz 5.6-4:

PRIMES $\in \mathbf{NP} \cap \mathbf{co\text{-}NP}$.

Beweis:

Zu zeigen ist, daß es sowohl für $L_1 = \{ n \mid n \in \mathbf{N} \text{ und } n \text{ ist eine Primzahl} \}$ als auch für $L_2 = \{ n \mid n \in \mathbf{N} \text{ und } n \text{ ist keine Primzahl} \}$ jeweils einen polynomiell zeitbeschränkten nichtdeterministischen Algorithmus gibt, der L_1 bzw. L_2 akzeptiert. Der Entwurf des nichtdeterministischen Algorithmus für L_1 basiert auf Satz 5.6-3, ein nichtdeterministischer Algorithmus für L_2 ergibt sich direkt aus den Eigenschaften zusammengesetzter Zahlen. Beide Algorithmen werden hier in Form von Pseudocode beschrieben. Es wird jeweils vorausgesetzt, daß für die Eingabe $n > 1$ gilt.

Ein nichtdeterministischer Algorithmus $\mathbf{A}_{L_2}$ zur Akzeptanz von L_2 lautet:

```
FUNCTION A_L2 (n : INTEGER): ...;

VAR B : INTEGER;

BEGIN { A_L2 }
   erzeuge (nichtdeterministisch) eine 0-1-Zeichenkette der Länge log2(⌊n⌋)+1
                                                    { = Länge der Binärdarstellung
                                                        von n                    }
   und weise diese der Variablen B (als Binärzahl) zu;
   A_L2 := V_L2 (n, B);
END   { A_L2 };
```

Die Arbeitsweise des Verifizierers $\mathbf{V}_{L_2}$ lautet:

```
FUNCTION V_L2 (n : INTEGER;
               B : INTEGER) : ...;

BEGIN { V_L2 }
  IF 1<B≤n-1                { Zeile 1 }
  THEN IF (n mod B)=0       { Zeile 2 }
       THEN V_L2 := ja
       ELSE V_L2 := nein
  ELSE V_L2 := nein;
END   { V_L2 };
```

Die Eingabe *n* für $\mathbf{V}_{L_2}$ belegt *k* Bits mit $k \in O(\log(n))$, $k = size(n)$. Die Überprüfung in Zeile 1 benötigt $O((\log(n))) = O(k)$ Operationen. Die Überprüfung in Zeile 2 (die Berechnung des Wertes $(n \bmod B)$) kann mit $O((\log(n))^2) = O(k^2)$ vielen Bitoperationen durchgeführt werden; denn $(n \bmod B) = n - (n \text{ DIV } B) \cdot B$, und alle arithmetische Operationen benötigen nach Satz 2.1-3 höchstens quadratisch viele Bitoperationen. Daher arbeitet $\mathbf{V}_{L_2}$ in polynomieller Zeit, gemessen in der Größe der Eingabe *n*.

Ist *n* keine Primzahl, dann besitzt *n* einen Teiler $a \in \mathbf{N}$ mit $1 < a \leq n-1$. Gibt man diesen Wert (Beweis) in $\mathbf{V}_{L_2}$ für den Parameter *B* ein, so antwortet $\mathbf{V}_{L_2}$ mit $\mathbf{V}_{L_2}(n,a) = \texttt{ja}$.
Ist *n* eine Primzahl, dann teilt kein $a \in \mathbf{N}$ mit $1 < a \leq n-1$ die Zahl *n*, d.h. für alle $a \in \mathbf{N}$ mit $1 < a \leq n-1$ ist $(n \bmod a) \neq 0$. Daher wird $\mathbf{V}_{L_2}$ bei Eingabe von *n* und eines beliebigen Beweises *B* immer die Antwort $\mathbf{V}_{L_2}(n,B) = \texttt{nein}$ geben.

Insgesamt ist damit $L_2 \in \mathbf{NP}$ gezeigt, also PRIMES $\in$ **co - NP**.

Ein nichtdeterministischer Algorithmus $\mathbf{A}_{L_1}$, der mit Hilfe eines Verifizierers $\mathbf{V}_{L_1}$ die Menge $L_1 = \{n \mid n \in \mathbf{N} \text{ und } n \text{ ist eine Primzahl}\}$ entscheidet, ist komplizierter zu konstruieren und setzt die in Satz 5.6-3 beschriebene Charakterisierung von Primzahlen um. Zur Vereinfachung der Darstellung, sind die nichtdeterministischen Schritte des Ratemoduls in den Code des Verifizierers aufgenommen worden.

```
FUNCTION V_L1 (n : INTEGER;
               B : INTEGER) : ...;

BEGIN { V_L1 }
  IF n = 2
  THEN V_L1 := ja
  ELSE IF (NOT (1 < B ≤ n-1)) OR ( B^(n-1) ≠ 1 (mod n) )          { Zeile 1 }
       THEN V_L1 := nein
       ELSE BEGIN
              erzeuge nichtdeterministisch k ≤ log(n-1) Zahlen p_1, ..., p_k
                     mit 3 ≤ p_i ≤ (n-1)/2
                     und k Zahlen B_1, ..., B_k mit 1 < B_i ≤ p_i - 1
                     für i = 1,...,k ;                            { Zeile 2 }
              IF NOT( (n-1 mod p_i) ≠ 0 für i = 1,...,k )         { Zeile 3 }
              THEN V_L1 := nein
              ELSE BEGIN
                     IF (( V_L1 ( p_i, B_i )= ja) für i = 1,...,k )   { Zeile 4 }
                     THEN IF ( B^((n-1)/p_i) ≠ 1 (mod n) ) für i = 1,...,k ){ Zeile 5 }
                          THEN V_L1 := ja
                          ELSE V_L1 := nein
                     ELSE V_L1 := nein;
                   END;
            END;
END   { V_L1 };
```

Für die Korrektkeit ist zu beachten, daß in Zeile 2 auf nichtdeterministische Weise die Menge der Primfaktoren von $n-1$ erzeugt wird. Die Werte B_i sind die zugehörigen Beweise. Die Anzahl der Primfaktoren von $n-1$ ist durch $\log(n-1)$ beschränkt. Außerdem gilt in Zeile 2 wegen $n \geq 3$, daß jeder Primfaktor von $n-1$ die Abschätzung $3 \leq p_i \leq (n-2)/2$ erfüllt. In Zeile 4 wird also $\mathbf{V}_{L_1}$ mit kleineren Werten rekursiv aufgerufen, deren Stellenzahl echt kleiner als die Stellenzahl von *n* ist. In Zeile 3 wird geprüft, ob die erzeugten Zahlen p_i Teiler von $n-1$ sind, und in Zeile 4 wird geprüft, ob die Zahlen p_i Primzahlen sind. Zeile 5 prüft Bedingung (ii) aus Satz 5.6-3. Es läßt sich zeigen (siehe angegebene Literatur), daß die Berechnung in den Zeilen 1 und 5 durch ein Laufzeitverhalten der Ordnung $O\left((\log(n))^3\right)$ abgeschätzt werden kann. Insgesamt läßt sich damit zeigen, daß die Laufzeit von $\mathbf{V}_{L_1}$ polynomiell in $size(n)$ ist. ///

Man hat für viele Probleme in **co - NP** jedoch nicht nachweisen können, daß sie in **NP** liegen (PRIMES gehört nicht dazu). Daher wird angenommen (obwohl es noch keinen Beweis dafür gibt), daß $\mathbf{NP} \neq \mathbf{co\text{-}NP}$ gilt. Aus der Gültigkeit von $\mathbf{NP} \neq \mathbf{co\text{-}NP}$ würde übrigens folgen, daß $\mathbf{P} \neq \mathbf{NP}$ ist, da wegen der Abgeschlossenheit der Klasse **P** gegenüber Komplementbildung $\mathbf{P} = \mathbf{co\text{-}P}$ gilt. Falls also der Nachweis $\mathbf{NP} \neq \mathbf{co\text{-}NP}$ gelingt, wäre damit das **P**-**NP**-Problem gelöst. Es ist jedoch nicht auszuschließen, daß $\mathbf{NP} = \mathbf{co\text{-}NP}$ und $\mathbf{P} \neq \mathbf{NP}$ gelten.

Satz 5.6-5:
Gibt es eine Sprache $L \in \mathbf{NPC}$ mit $L \in \mathbf{co\text{-}NP}$, dann ist $\mathbf{NP} = \mathbf{co\text{-}NP}$.

Beweis:

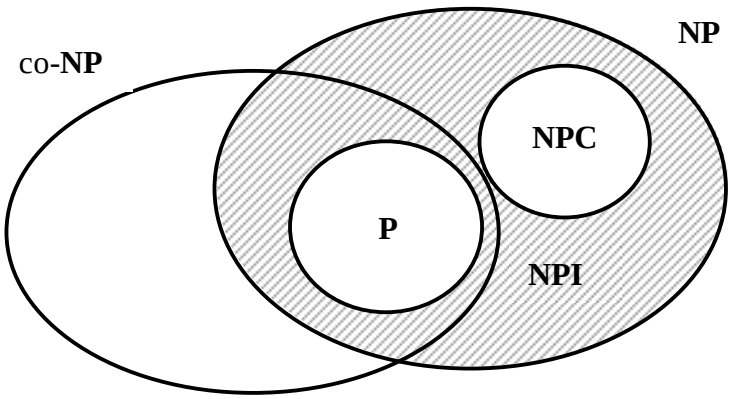
Es sei $L \in \mathbf{NPC} \cap \mathbf{co\text{-}NP}$. Das der Sprache *L* zugrundeliegende Alphabet sei Σ, d.h. $L \subseteq \Sigma^*$.
Es gilt $\mathbf{NP} \subseteq \mathbf{co\text{-}NP}$: Dazu wird gezeigt, daß für jede Sprache $L_1 \in \mathbf{NP}$, $L_1 \subseteq \Sigma_1^*$, die Beziehung $L_1 \in \mathbf{co-NP}$ nachweisbar ist:

Wegen $L \in \mathbf{NPC}$ ist $L_1 \leq_m^p L$. Die dabei verwendete in polynomieller Zeit berechenbare totale Funktion sei *f*, d.h. es gilt $f : \Sigma_1^* \to \Sigma^*$ mit $w \in L_1 \Leftrightarrow f(w) \in L$. Der Übergang auf die Komplemente ergibt $w \in \Sigma_1^* \setminus L_1 \Leftrightarrow f(w) \in \Sigma^* \setminus L$.
Wegen $L \in \mathbf{co\text{-}NP}$ ist $\Sigma^* \setminus L \in \mathbf{NP}$. Es sei $\overline{NTM}$ eine nichtdeterministische polynomiell zeitbeschränkte Turingmaschine mit $L(\overline{NTM}) = \Sigma^* \setminus L$. Aus $\overline{NTM}$ und der Turingmaschine TM_f zur Berechnung von *f* läßt sich durch Hintereinanderschalten eine nichtdeterministische polynomiell zeitbeschränkte Turingmaschine $\overline{NTM}_1$ mit $L(\overline{NTM}_1) = \Sigma_1^* \setminus L_1$ konstruieren: $\overline{NTM}_1$ arbeitet bei Eingabe $w \in \Sigma_1^*$ wie folgt: Mittels TM_f wird zunächst $f(w)$ berechnet. Das Ergebnis wird dann in $\overline{NTM}$ eingegeben. $\overline{NTM}_1$ antwortet mit derselben Antwort, die $\overline{NTM}$ (bei Eingabe von $f(w)$) gibt. Offensichtlich ist $\overline{NTM}_1$ eine polynomiell zeitbeschränkte nichtdeterministische Turingmaschine, und es gilt:

$w \in \Sigma_1^* \setminus L_1 \Leftrightarrow f(w) \in \Sigma^* \setminus L$
$\Leftrightarrow \overline{NTM}$ stoppt im akzeptierenden Zustand
$\Leftrightarrow \overline{NTM}_1$ stoppt im akzeptierenden Zustand
$\Leftrightarrow w \in L(\overline{NTM}_1)$.

Daher ist $\Sigma_1^* \setminus L_1 \in \mathbf{NP}$ bzw. $L_1 \in \mathbf{co\text{-}NP}$.
Die umgekehrte Inklusion $\mathbf{co\text{-}NP} \subseteq \mathbf{NP}$ läßt sich mit ähnlichen Argumenten zeigen. ///

Unter den Annahmen $\mathbf{P} \neq \mathbf{NP}$ und $\mathbf{NP} \neq \mathbf{co\text{-}NP}$ ergibt sich folgendes Gesamtbild der beschriebenen Klassen:



6 Approximation von Optimierungsaufgaben

Gegeben sei das Optimierungsproblem Π:

Instanz:
1. $x \in \Sigma_\Pi^*$
2. Spezifikation einer Funktion SOL_Π, die jedem $x \in \Sigma_\Pi^*$ eine Menge zulässiger Lösungen zuordnet
3. Spezifikation einer Zielfunktion m_Π, die jedem $x \in \Sigma_\Pi^*$ und $y \in \mathrm{SOL}_\Pi(x)$ den Wert $m_\Pi(x, y)$ einer zulässigen Lösung zuordnet
4. $goal_\Pi \in \{\min, \max\}$, je nachdem, ob es sich um ein Minimierungs- oder ein Maximierungsproblem handelt.

Lösung: $y^* \in \mathrm{SOL}_\Pi(x)$ mit $m_\Pi(x, y^*) = \min\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$ bei einem Minimierungsproblem (d.h. $goal_\Pi = \min$) bzw. $m_\Pi(x, y^*) = \max\{m_\Pi(x, y) \mid y \in \mathrm{SOL}_\Pi(x)\}$ bei einem Maximierungsproblem (d.h. $goal_\Pi = \max$).
Der Wert $m_\Pi(x, y^*)$ einer optimalen Lösung wird auch mit $m_\Pi^*(x)$ bezeichnet.

Im folgenden (und in den vorhergehenden Beispielen) werden Optimierungsproblemen untersucht, die auf der Grenze zwischen praktischer Lösbarkeit (tractability) und praktischer Unlösbarkeit (intractability) stehen. In Analogie zu Entscheidungsproblemen in **NP** bilden diese die Klasse **NPO**:

Das Optimierungsproblem Π gehört zur **Klasse NPO**, wenn gilt:

1. Die Menge der Instanzen $x \in \Sigma_\Pi^*$ ist in polynomieller Zeit entscheidbar
2. Es gibt ein Polynom q mit der Eigenschaft: für jedes $x \in \Sigma_\Pi^*$ und jede zulässige Lösung $y \in \mathrm{SOL}_\Pi(x)$ gilt $|y| \le q(|x|)$, und für jedes y mit $|y| \le q(|x|)$ ist in polynomieller Zeit entscheidbar, ob $y \in \mathrm{SOL}_\Pi(x)$ ist
3. Die Zielfunktion m_Π ist in polynomieller Zeit berechenbar.

Alle bisher behandelten Beispiele für Optimierungsprobleme liegen in der Klasse **NPO**. Dazu gehören insbesondere auch solche, deren zugehöriges Entscheidungsproblem **NP**-vollständig ist.

Den Zusammenhang zwischen den Klassen **NPO** und **NP** beschreibt der Satz

Satz 6-1:
Für jedes Optimierungsproblem in **NPO** ist das zugehörige Entscheidungsproblem in **NP**.

Beweis:

Der Beweis wird hier nur für ein Maximierungsproblem erbracht, für ein Minimierungsproblem kann man in ähnlicher Weise argumentieren.

Es sei Π ein Maximierungsproblem in **NPO** (die im Beweis verwendeten Funktionen werden in den obigen Definitionen benannt). Das zugehörige Entscheidungsproblem sei Π_{Ent}. Hierbei soll bei Vorlage einer Instanz $[x,K]$ mit $x \in \Sigma_\Pi^*$ und $K \in \mathbf{N}$ genau dann die Entscheidung $x \in L_{\Pi_{Ent}}$ getroffen werden, wenn $m_\Pi^*(x) \geq K$ ist. Um zu zeigen, daß Π_{Ent} in **NP** liegt, ist ein Verifizierer **V** für Π_{Ent} anzugeben, der bei Eingabe einer Instanz $[x,K]$ und eines Beweises B diesen in polynomieller Zeit verifiziert. Ein Beweis ist hier eine Zeichenkette, die über dem Alphabet Σ_0 gebildet wird, mit dem man auch die zulässigen Lösungen von *x* formuliert.

Der Verifizierer **V** für Π_{Ent} wird durch den folgenden Pseudocode gegeben:

```
FUNCTION V ([x,K] : ...;
            B     : ...) : ...;
          { x ∈ Σ*_Π , K ∈ N , B ∈ Σ*_0 }

BEGIN { V }
  IF (|B| ≤ q(|x|)) AND (B ∈ SOL_Π(x))          { Zeile 1 }
  THEN IF m_Π(x,B) ≥ K THEN V := „ja“           { Zeile 2 }
                       ELSE V := „nein“
  ELSE V := „nein“;
END   { V };
```

Zu zeigen ist

1. **V** arbeitet in polynomieller Zeit, gemessen in $|x|$
2. $[x,K] \in L_{\Pi_{Ent}} \Leftrightarrow$ es gibt $B_x \in \Sigma_0^*$ mit $\mathbf{V}(x,B_x) = \mathtt{ja}$.

Zu 1.: Da Π in **NPO** ist, lassen sich die Berechnungen in den Zeilen 1 und 2 in polynomieller Zeit ausführen; dieses wird durch die Punkte 2. und 3. in der Definition der Klasse **NPO** gesichert.

Zu 2.: Es sei $[x,K] \in L_{\Pi_{Ent}}$. Dann gibt es eine optimale Lösung $y^* \in \mathrm{SOL}_\Pi(x)$ mit $m^*_\Pi(x) = m_\Pi(x,y^*) \geq K$. Da Π in **NPO** ist, gilt $|y^*| \leq q(|x|)$. Bei Eingabe von $[x,K]$ und y^* in **V** ergibt sich $\mathbf{V}([x,K],y^*) = \mathtt{ja}$.
Es gelte $[x,K] \notin L_{\Pi_{Ent}}$. Dann gilt für jedes $y \in \mathrm{SOL}_\Pi(x)$: $m_\Pi(x,y) \leq m^*_\Pi(x) < K$. Es sei $B \in \Sigma_0^*$ mit $|B| \leq q(|x|)$. Ist $B \in \mathrm{SOL}_\Pi(x)$, dann antwortet **V** wegen $m_\Pi(x,B) < K$ in Zeile 2 mit $\mathbf{V}([x,K],B) = \mathtt{nein}$. Ist $B \notin \mathrm{SOL}_\Pi(x)$, dann antwortet **V** wegen Zeile 1 mit $\mathbf{V}([x,K],B) = \mathtt{nein}$. ///

Analog zur Definition der Klasse **P** innerhalb **NP** läßt sich innerhalb **NPO** eine Klasse **PO** definieren:

Ein Optimierungsproblem Π aus **NPO** gehört zur **Klasse PO**, wenn es einen polynomiell zeitbeschränkten Algorithmus gibt, der für jede Instanz $x \in \Sigma_\Pi^*$ eine optimale Lösung $y^* \in \mathrm{SOL}_\Pi(x)$ zusammen mit dem optimalen Wert $m^*_\Pi(x)$ der Zielfunktion ermittelt.

Offensichtlich ist $\mathbf{PO} \subseteq \mathbf{NPO}$.

Es gilt:

Satz 6-2:
Ist $\mathbf{P} \neq \mathbf{NP}$, dann ist $\mathbf{PO} \neq \mathbf{NPO}$.

Im Laufe dieses Kapitels wird die Struktur der Klasse **NPO** genauer untersucht. Im folgenden liegen daher alle behandelten Optimierungsprobleme in **NPO**.

Bei Optimierungsaufgaben gibt man sich häufig mit Näherungen an die optimale Lösung zufrieden, insbesondere dann, wenn diese „leicht“ zu berechnen sind und vom Wert der optimalen Lösung nicht zu sehr abweichen. Unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ gibt es für ein Optimierungsproblem, dessen zugehöriges Entscheidungsproblem **NP**-vollständig ist, kein Verfahren, das eine optimale Lösung in polynomieller Laufzeit ermittelt. Gerade diese Probleme sind in der Praxis jedoch häufig von großem Interesse.

Für eine Instanz $x \in \Sigma_\Pi^*$ und für eine zulässige Lösung $y \in \mathrm{SOL}_\Pi(x)$ bezeichnet
$D(x,y) = \left| m_\Pi^*(x) - m_\Pi(x,y) \right|$
den **absoluten Fehler von** *y* **bezüglich** *x*.

Ein Algorithmus **A** ist ein **Approximationsalgorithmus (Näherungsalgorithmus)** für Π, wenn er bei Eingabe von $x \in \Sigma_\Pi^*$ eine zulässige Lösung liefert, d.h. wenn $\mathbf{A}(x) \in \mathrm{SOL}_\Pi(x)$ gilt. **A** heißt **absoluter Approximationsalgorithmus (absoluter Näherungsalgorithmus)**, wenn es eine Konstante *k* gibt mit $D(x, \mathbf{A}(x)) = \left| m_\Pi^*(x) - m_\Pi(x, \mathbf{A}(x)) \right| \le k$. Das Optimierungsproblem Π heißt in diesem Fall **absolut approximierbar**.

Ist Π ein Optimierungsproblem, dessen zugehöriges Entscheidungsproblem **NP**-vollständig ist, so sucht man natürlich nach absoluten Approximationsalgorithmen für Π, die polynomielle Laufzeit aufweisen und für die der Wert $D(x, \mathbf{A}(x))$ möglichst klein ist. Das folgende Beispiel zeigt jedoch, daß unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ nicht jedes derartige Problem absolut approximierbar ist. Dazu werde der folgende Spezialfall des 0/1-Rucksack-Maximierungsproblems betrachtet:

Das ganzzahlige 0/1-Rucksack-Maximierungsproblem

Instanz: 1. $I = (A, M)$
$A = \{a_1, ..., a_n\}$ ist eine Menge von *n* Objekten und $M \in \mathbf{N}$ die „Rucksackkapazität". Jedes Objekt a_i, $i = 1, ..., n$, hat die Form $a_i = (w_i, p_i)$; hierbei bezeichnet $w_i \in \mathbf{N}$ das Gewicht und $p_i \in \mathbf{N}$ den Wert (Profit) des Objekts a_i.
$size(I) = n$

2. $\mathrm{SOL}(I) = \left\{ (x_1, ..., x_n) \;\middle|\; x_i = 0 \text{ oder } x_i = 1 \text{ für } i = 1, ..., n \text{ und } \sum_{i=1}^{n} x_i \cdot w_i \le M \right\}$

3. $m(I, (x_1, ..., x_n)) = \sum_{i=1}^{n} x_i \cdot p_i$ für $(x_1, ..., x_n) \in \mathrm{SOL}(I)$

4. *goal* = max

Lösung: Eine Folge $x_1^*, ..., x_n^*$ von Zahlen mit

(1) $x_i^* = 0$ oder $x_i^* = 1$ für $i = 1, ..., n$

(2) $\sum_{i=1}^{n} x_i^* \cdot w_i \le M$

(3) $m(I, (x_1^*, ..., x_n^*)) = \sum_{i=1}^{n} x_i^* \cdot p_i$ ist maximal unter allen möglichen Auswahlen $x_1, ..., x_n$, die (1) und (2) erfüllen.

Das zugehörige Entscheidungsproblem ist **NP**-vollständig, so daß dieses Optimierungsproblem unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ keinen polynomiell zeitbeschränkten Lösungsalgorithmus (der eine *optimale* Lösung ermittelt) besitzt. Es gilt sogar:

Satz 6-3:

Es sei *k* eine vorgegebene Konstante. Unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ gibt es keinen polynomiell zeitbeschränkten Approximationsalgorithmus **A** für das ganzzahlige 0/1-Rucksack-Maximierungsproblem, der bei Eingabe einer Instanz $I = (A, M)$ eine zulässige Lösung $\mathbf{A}(I) \in \mathrm{SOL}(I)$ berechnet, für deren absoluter Fehler $D(I, \mathbf{A}(I)) = \left| m^*(I) - m(I, \mathbf{A}(I)) \right| \leq k$ gilt.

Beweis:

Die verwendete Beweistechnik ist auch auf andere Probleme übertragbar.

Es wird angenommen, daß es (bei Vorgabe der Konstanten *k*) einen derartigen polynomiell zeitbeschränkten Approximationsalgorithmus **A** gibt und zeigt dann, daß dieser (mit einer einfachen Modifikation) dazu verwendet werden kann, in polynomieller Zeit eine optimale Lösung für das ganzzahlige 0/1-Rucksack-Maximierungsproblem zu ermitteln. Damit kann man dann das zu diesem Problem gehörende Entscheidungsproblem in polynomieller Zeit lösen. Da dieses **NP**-vollständig ist, folgt $\mathbf{P} = \mathbf{NP}$ (vgl. Satz 5.1-1).

Bei Annahme der Existenz von **A** kann ein Algorithmus $\tilde{\mathbf{A}}$ definiert werden, der folgendendermaßen arbeitet:
Aus einer Eingabe einer Instanz $I = (A, M)$ mit $A = \{(w_1, p_1), ..., (w_n, p_n)\}$ für das ganzzahlige 0/1-Rucksack-Maximierungsproblem erzeugt $\tilde{\mathbf{A}}$ eine Instanz $\tilde{I} = (\tilde{A}, M)$ mit $\tilde{A} = \{(w_1, (k+1) \cdot p_1), ..., (w_n, (k+1) \cdot p_n)\}$ (es werden dabei also lediglich alle Profite p_i durch $(k+1) \cdot p_i$ ersetzt). Diese neue Instanz $\tilde{I} = (\tilde{A}, M)$ wird in **A** eingegeben. **A** ermittelt eine approximative Lösung $\mathbf{A}(\tilde{I}) = (x_1, ..., x_n)$ mit $\sum_{i=1}^{n} x_i \cdot w_i \leq M$ und

$$\left| m^*(\tilde{I}) - m(\tilde{I}, \mathbf{A}(\tilde{I})) \right| = \left| m^*(\tilde{I}) - \sum_{i=1}^{n} x_i \cdot (k+1) \cdot p_i \right| \leq k \qquad \text{(Ungleichung (*)).}$$

$\tilde{\mathbf{A}}$ gibt die von **A** bei Eingabe von $\tilde{I}$ ermittelte Lösung $\tilde{\mathbf{A}}(I) = \mathbf{A}(\tilde{I}) = (x_1, ..., x_n)$ mit (dem Wert der Zielfunktion) $m(I, \tilde{\mathbf{A}}(I)) = \dfrac{m(\tilde{I}, \mathbf{A}(\tilde{I}))}{k+1}$ aus. Es ist $\tilde{\mathbf{A}}(I) \in \mathrm{SOL}(I)$. Es gilt sogar $m(I, \tilde{\mathbf{A}}(I)) = m^*(I)$, d.h. $\tilde{\mathbf{A}}$ liefert in polynomieller Zeit (da **A** in polynomieller Zeit arbeitet) eine optimale Lösung für die Eingabeinstanz *I*:

Da $m^*(\tilde{I})$ ein Vielfaches von $k+1$ ist, denn jeder Profit in $\tilde{I}$ ist ein Vielfaches von $k+1$, kann man den Faktor $k+1$ auf der linken Seite des $\leq$-Zeichens in der Ungleichung (*) ausklammern und erhält

$$\left|m^*(\tilde{I}) - m(\tilde{I}, \mathbf{A}(\tilde{I}))\right| = \left|m^*(\tilde{I}) - \sum_{i=1}^{n} x_i \cdot (k+1) \cdot p_i\right| = (k+1) \cdot R \leq k$$

mit einer natürlichen Zahl *R*. Diese Ungleichung ist nur mit $R = 0$ möglich, so daß $\left|m^*(\tilde{I}) - m(\tilde{I}, \mathbf{A}(\tilde{I}))\right| = 0$ folgt, d.h. $\mathbf{A}(\tilde{I})$ ist eine optimale Lösung für $\tilde{I}$.

Jede zulässige Lösung für $\tilde{I}$ ist auch eine zulässige Lösung für *I*, jedoch mit dem $(k+1)$-fachen Profit. Umgekehrt ist jede zulässige Lösung für *I* eine zulässige Lösung für $\tilde{I}$. Damit folgt die Optimalität von $\tilde{\mathbf{A}}(I)$ (dazu ist $m(I, \tilde{\mathbf{A}}(I)) \geq m(I, y)$ für jedes $y \in \mathrm{SOL}(I)$ zu zeigen):

$$m(I, \tilde{\mathbf{A}}(I)) = \frac{m(\tilde{I}, \mathbf{A}(\tilde{I}))}{k+1} = \frac{m^*(\tilde{I})}{k+1} \geq \frac{m(\tilde{I}, y)}{k+1} = m(I, y) \text{ für jedes } y \in \mathrm{SOL}(I). \text{ ///}$$

Die Forderung nach der Garantie der Einhaltung eines absoluten Fehlers ist also häufig zu stark. Es bietet sich daher an, einen Approximationsalgorithmus nach seiner relativen Approximationsgüte zu beurteilen.

Es sei ein Π wieder ein Optimierungsproblem und **A** ein Approximationsalgorithmus für Π (siehe oben), der eine zulässige Lösung $\mathbf{A}(x)$ ermittelt. Ist $x \in \Sigma_\Pi^*$ eine Instanz von Π, so gilt trivialerweise $m(x, \mathbf{A}(x)) \leq m_\Pi^*(x)$ bei einem Maximierungsproblem bzw. $m_\Pi^*(x) \leq m(x, \mathbf{A}(x))$ bei einem Minimierungsproblem.

Die **relative Approximationsgüte** $R_\mathbf{A}(x)$ von **A** bei Eingabe einer Instanz $x \in \Sigma_\Pi^*$ wird definiert durch

$$R_\mathbf{A}(x) = \frac{m_\Pi^*(x)}{m(x, \mathbf{A}(x))} \text{ bei einem Maximierungsproblem}$$

bzw.

$$R_\mathbf{A}(x) = \frac{m(x, \mathbf{A}(x))}{m_\Pi^*(x)} \text{ bei einem Minimierungsproblem.}$$

Bemerkung: Um nicht zwischen Maximierungs- und Minimierungsproblem in der Definition unterscheiden zu müssen, kann man die relative Approximationsgüte von **A** bei Eingabe einer Instanz $x \in \Sigma_\Pi^*$ auch durch

$$R_\mathbf{A}(x) = \max\left\{\frac{m_\Pi^*(x)}{m(x, \mathbf{A}(x))}, \frac{m(x, \mathbf{A}(x))}{m_\Pi^*(x)}\right\}$$

definieren.

Offensichtlich gilt immer $1 \le R_{\mathbf{A}}(x)$. Je dichter $R_{\mathbf{A}}(x)$ bei 1 liegt, um so besser ist die Approximation.

Die Aussage „$R_{\mathbf{A}}(x) \le c$“ mit einer Konstanten $c \ge 1$ für alle Instanzen $x \in \Sigma_{\Pi}^*$ bedeutet bei einem Maximierungsproblem $1 \le \frac{m_{\Pi}^*(x)}{m(x, \mathbf{A}(x))} \le c$, also $m(x, \mathbf{A}(x)) \ge 1/c \cdot m_{\Pi}^*(x)$, d.h. der Approximationsalgorithmus liefert eine Lösung, die sich mindestens um $1/c$ dem Optimum nähert. Gewünscht ist also ein möglichst großer Wert von $1/c$ bzw. ein Wert von *c*, der möglichst klein (d.h. dicht bei 1) ist.

Bei einem Minimierungsproblem impliziert die Aussage „$R_{\mathbf{A}}(x) \le c$“ die Beziehung $m(x, \mathbf{A}(x)) \le c \cdot m_{\Pi}^*(x)$, d.h. der Wert der Approximation überschreitet das Optimum um höchstens das *c*-fache. Auch hier ist also ein Wert von *c* gewünscht, der möglichst klein (d.h. dicht bei 1) ist.

6.1 Relativ approximierbare Probleme

Es sei $r \ge 1$. Der Approximationsalgorithmus **A** für das Optimierungsproblem Π aus **NPO** heißt ***r*-approximativer Algorithmus**, wenn $R_{\mathbf{A}}(x) \le r$ für jede Instanz $x \in \Sigma_{\Pi}^*$ gilt.

Bemerkung: Es sei **A** ein Approximationsalgorithmus für das Minimierungsproblem Π, und es gelte $m(x, \mathbf{A}(x)) \le r \cdot m_{\Pi}^*(x) + k$ für alle Instanzen $x \in \Sigma_{\Pi}^*$ (mit Konstanten *r* und *k*). Dann ist **A** lediglich $(r + k)$-approximativ und nicht etwa *r*-approximativ, jedoch **asymptotisch** *r*-approximativ (siehe Kapitel 6.2).

Die **Klasse APX** besteht aus denjenigen Optimierungsproblemen aus **NPO**, für die es einen *r*-approximativen Algorithmus für ein $r \ge 1$ gibt.

Offensichtlich ist $\mathbf{APX} \subseteq \mathbf{NPO}$.

Das folgende Optimierungsproblem liegt in **APX**:

Binpacking-Minimierungsproblem:

Instanz:
1. $I = [a_1, \ldots, a_n]$
   $a_1, \ldots, a_n$ („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, \ldots, n$
   $size(I) = n$
2. $\mathrm{SOL}(I) = \left\{ [B_1, \ldots, B_k] \;\middle|\; \begin{array}{l} [B_1, \ldots, B_k] \text{ ist eine Partition (disjunkte Zerlegung)} \\ \text{von } I \text{ mit } \sum_{a_i \in B_j} a_i \leq 1 \text{ für } j = 1, \ldots, k \end{array} \right\}$, d.h. in einer zulässigen Lösung werden die Objekte so auf k „Behälter“ der Höhe 1 verteilt, daß kein Behälter „überläuft“.
3. Für $[B_1, \ldots, B_k] \in \mathrm{SOL}(I)$ ist $m(I, [B_1, \ldots, B_k]) = k$, d.h. als Zielfunktion wird die Anzahl der benötigten Behälter definiert
4. *goal* = min

Lösung: Eine Partition der Objekte in möglichst wenige Teile $B_1, \ldots, B_{k^*}$ und (implizit) die Anzahl k^* der benötigten Teile.

Bemerkung: Da das Laufzeitverhalten der folgenden Approximationsalgorithmen nicht von den Größen der in den Instanzen vorkommenden Objekte abhängt, sondern nur von deren Anzahl, kann man für eine Eingabeinstanz $I = [a_1, \ldots, a_n]$ als Größe den Wert $size(I) = n$ wählen.

Das Binpacking-Minimierungsproblem ist eines der am besten untersuchten Minimierungsprobleme einschließlich der Verallgemeinerungen auf mehrdimensionale Objekte. Das zugehörige Entscheidungsproblem ist **NP**-vollständig, so daß unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ kein polynomiell zeitbeschränkter Optimierungsalgorithmus erwartet werden kann.

Der folgende in Pseudocode formulierte Algorithmus approximiert eine optimale Lösung:

Nextfit-Algorithmus zur Approximation von Binpacking:

Eingabe: $I = [a_1, \ldots, a_n]$,
$a_1, \ldots, a_n$ („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, \ldots, n$

Verfahren: $B_1 := a_1$;

Sind $a_1, ..., a_i$ bereits in die Behälter $B_1, ..., B_j$ gelegt worden, ohne daß einer dieser Behälter überläuft, so lege a_{i+1} nach B_j (in den Behälter mit dem höchsten Index), falls dadurch B_j nicht überläuft, d.h. $\sum_{a_l \in B_j} a_l \le 1$ gilt; andernfalls lege a_{i+1} in einen neuen (bisher leeren) Behälter B_{j+1}.

Ausgabe: $\mathbf{NF}(I) =$ die benötigten nichtleeren Behälter $[B_1, ..., B_k]$.

Satz 6.1-1:

Ist $I = [a_1, ..., a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $R_{\mathbf{NF}}(I) \le 2$, d.h. der Nextfit-Algorithmus ist 2-approximativ ($m(I, \mathbf{NF}(I)) \le 2 \cdot m^*(I)$). Das Binpacking -Minimierungsproblem liegt in **APX**.

Beweis:

Interessanterweise kann man die Aussage $m(I, \mathbf{NF}(I)) \le 2 \cdot m^*(I)$ machen, ohne $m^*(I)$ oder $m(I, \mathbf{NF}(I))$ zu kennen. Dazu wird $m^*(I)$ abgeschätzt:

Für eine Instanz *I* habe der Nextfit-Algorithmus *k* Behälter ermittelt, d.h. $m(I, \mathbf{NF}(I)) = k$. Die Füllhöhe im *j*-ten Behälter sei u_j für *j* = 1, ..., *k*. Das erste Element, das der Nextfit-Algorithmus in den Behälter B_j gelegt hat, sei b_j.

Man betrachte die beiden Behälter B_j und B_{j+1} (für $1 \le j < k$): Da b_{j+1} nicht mehr in den Behälter B_j paßte, gilt $1 - u_j < b_j \le u_{j+1}$ und damit $u_j + u_{j+1} > 1$. Summiert man diese $k-1$ Ungleichungen auf, so erhält man

$(u_1 + u_2) + (u_2 + u_3) + ... + (u_{k-1} + u_k) = u_1 + 2u_2 + ... + 2u_{k-1} + u_k > k - 1$.

Auf beide Seiten werden die Füllhöhen des ersten und letzten Behälters addiert, und man erhält $2 \cdot \sum_{j=1}^{k} u_j > k - 1 + u_1 + u_k$. Die Summe der Füllhöhen in den Behältern ist gleich der Summe der Objekte, die gepackt wurden. Damit folgt

$k < 2 \cdot \sum_{j=1}^{k} u_j + 1 - (u_1 + u_k) = 2 \cdot \sum_{i=1}^{n} a_i + 1 - (u_1 + u_k) \le 2 \cdot \sum_{i=1}^{n} a_i + 1$ und $k \le 2 \cdot \sum_{i=1}^{n} a_i$.

Trivialerweise gilt $m^*(I) \ge \sum_{i=1}^{n} a_i$ (hier gilt „=“, wenn in einer optimalen Packung alle Behälter bis zur maximalen Füllhöhe 1 aufgefüllt werden). Damit ergibt sich schließlich $R_{\mathbf{NF}}(I) = k / m^*(I) \le 2$. ///

Die Grenze 2 im Nextfit-Algorithmus ist asymptotisch optimal: es gibt Instanzen *I*, für die $R_{\mathbf{NF}}(I)$ beliebig dicht an 2 herankommt. Dazu betrachte man etwa eine Eingabeinstanz *I* mit $n = 2m$ vielen Objekten, wobei *m* eine gerade Zahl ist: $I = [a_1, \ldots, a_{2m}]$. Die Werte a_i seien definiert durch $a_i = \begin{cases} 1/2 - 1/3m & \text{für ungerades } i \\ 1/m & \text{für gerades } i \end{cases}$. Mit dieser Instanz gilt $m(I, \mathbf{NF}(I)) = m$ und $m^*(I) = m/2 + 1$, so daß $R_{\mathbf{NF}}(I) = 2 \cdot \frac{m}{m+2}$ folgt, und dieser Wert kann für große *m* beliebig dicht an 2 herangehen.

Es gilt sogar eine genauere Abschätzung der relativen Approximationsgüte, falls die Größe aller Objekte beschränkt ist: Mit $a_{\max} = \max\{a_i \mid i = 1, \ldots, n\}$ ist

$$m(I, \mathbf{NF}(I)) \le \begin{cases} (1 + a_{\max} / (1 - a_{\max})) \cdot m^*(I) & \text{falls } 0 < a_{\max} \le 1/2 \\ 2 \cdot m^*(I) & \text{falls } 1/2 < a_{\max} \le 1 \end{cases}$$

Zu beachten ist, daß der Nextfit-Algorithmus ein online-Algorithmus ist, d.h. die Objekte der Reihenfolge nach inspiziert, und sofort eine Entscheidung trifft, in welchen Behälter ein Objekt zu legen ist, ohne alle Objekte gesehen zu haben. Die Laufzeit des Nextfit-Algorithmus bei einer Eingabeinstanz der Größe *n* liegt in $O(n)$.

Eine asymptotische Verbesserung der Approximation liefert der Firstfit-Algorithmus, der ebenfalls ein online-Algorithmus ist und bei geeigneter Implementierung ein Laufzeitverhalten der Ordnung $O(n \cdot \log(n))$ hat:

Firstfit-Algorithmus zur Approximation von Binpacking:

Eingabe: $I = [a_1, \ldots, a_n]$,
$a_1, \ldots, a_n$ („Objekte“) sind rationale Zahlen mit $0 < a_i \le 1$ für *i* = 1, ..., *n*

Verfahren: $B_1 := a_1$;
Sind $a_1, \ldots, a_i$ bereits in die Behälter $B_1, \ldots, B_j$ gelegt worden, ohne daß einer dieser Behälter überläuft, so lege a_{i+1} in den Behälter unter $B_1, \ldots, B_j$ mit dem kleinsten Index, in den das Objekt a_{i+1} noch paßt, ohne daß er überläuft. Falls

es einen derartigen Behälter unter $B_1, ..., B_j$ nicht gibt, lege a_{i+1} in einen neuen (bisher leeren) Behälter B_{j+1}.

Ausgabe: $\mathbf{FF}(I) =$ die benötigten nichtleeren Behälter $[B_1, ..., B_k]$.

Satz 6.1-2:

Ist $I = [a_1, \ldots, a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $m(I, \mathbf{FF}(I)) \leq 1{,}7 \cdot m^*(I) + 2$.

Der Beweis verwendet eine „Gewichtung“ der Objekte der Eingabeinstanz. Wegen der Länge des Beweises muß auf die Literatur verwiesen werden.

Die Grenze 1,7 im Firstfit-Algorithmus ist ebenfalls asymptotisch optimal: es gibt Instanzen *I* mit beliebig großem Wert $m^*(I)$, für die $m(I, \mathbf{FF}(I)) \geq 1{,}7 \cdot (m^*(I) - 1)$ gilt. Daher kommt $R_{\mathbf{FF}}(I)$ asymptotisch beliebig dicht an 1,7 heran.

Zu beachten ist weiterhin, daß der Firstfit-Algorithmus kein 1,7-approximativer Algorithmus ist, sondern nur asymptotisch 1,7-approximativ (siehe Kapitel 6.2) ist.

Eine weitere asymptotische Verbesserung erhält man, indem man die Objekte vor der Aufteilung auf Behälter nach absteigender Größe sortiert. Die entstehenden Approximationsalgorithmen sind dann jedoch offline-Algorithmen, da zunächst alle Objekte vor der Aufteilung bekannt sein müssen.

FirstfitDecreasing-Algorithmus zur Approximation von Binpacking:

Eingabe: $I = [a_1, \ldots, a_n]$,
$a_1, \ldots, a_n$ („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für *i* = 1, ..., *n*

Verfahren: Sortiere die Objekte $a_1, \ldots, a_n$ nach absteigender Größe. Die Objekte erhalten im folgenden wieder die Benennung $a_1, \ldots, a_n$, d.h. es gilt $a_1 \geq \ldots \geq a_n$.
$B_1 := a_1$;

Sind $a_1, ..., a_i$ bereits in die Behälter $B_1, ..., B_j$ gelegt worden, ohne daß einer dieser Behälter überläuft, so lege a_{i+1} in den Behälter unter $B_1, ..., B_j$ mit dem kleinsten Index, in den das Objekt a_{i+1} noch paßt, ohne daß er überläuft. Falls es einen derartigen Behälter unter $B_1, ..., B_j$ nicht gibt, lege a_{i+1} in einen neuen (bisher leeren) Behälter B_{j+1}.

Ausgabe: **FFD**$(I) =$ die benötigten nichtleeren Behälter $[B_1, ..., B_k]$.

Satz 6.1-3:
Ist $I = [a_1, ..., a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $m(I, \mathbf{FFD}(I)) \leq 1{,}5 \cdot m^*(I) + 1$.

Beweis:

Die Objekte der Eingabeinstanz $I = [a_1, ..., a_n]$ seien nach absteigender Größe sortiert, d.h. $a_1 \geq ... \geq a_n$. Sie werden in vier Größenordnungen eingeteilt. Dazu wird
$A = \{a_i \mid a_i \in I \text{ und } a_i > 2/3\}$,
$B = \{a_i \mid a_i \in I \text{ und } 2/3 \geq a_i > 1/2\}$,
$C = \{a_i \mid a_i \in I \text{ und } 1/2 \geq a_i > 1/3\}$ und
$D = \{a_i \mid a_i \in I \text{ und } 1/3 \geq a_i > 0\}$ gesetzt.
Der FirstfitDecreasing-Algorithmus habe eine Packung mit *k* Behältern ermittelt, d.h. $m(I, \mathbf{FFD}(I)) = k$. Die Füllhöhe des *j*-ten Behälters B_j sei u_j.

1. Fall: Es gibt mindestens einen Behälter, der nur Objekte aus *D* enthält.
Dann sind alle anderen Behälter, bis eventuell auf den Behälter B_k zu mindestens $2/3$ gefüllt. Es gilt dann
$\sum_{i=1}^{n} a_i = \sum_{j=1}^{k-1} u_j + u_k \geq \sum_{j=1}^{k-1} 2/3 = 2/3 \cdot (k-1)$ und folglich
$m(I, \mathbf{FFD}(I)) = k \leq 3/2 \cdot \sum_{i=1}^{n} a_i + 1 \leq 3/2 \cdot m^*(I) + 1$.

2. Fall: Kein Behälter enthält nur Objekte aus *D*.
Es sei $\tilde{I} = I \setminus D$. Alle Objekte in $\tilde{I}$ haben eine Größe von mehr als $1/3$. Dann gilt $\mathbf{FFD}(\tilde{I}) = \mathbf{FFD}(I)$, da die Objekte in *D* noch auf die übrigen Behälter verteilt werden können und erst dann verteilt werden, wenn alle Objekte in *A*, *B* und *C* verteilt sind.

Es gilt sogar $m(\tilde{I}, \mathbf{FFD}(\tilde{I})) = m^*(\tilde{I})$, d.h. der FirstfitDecreasing-Algorithmus liefert bei Eingabe von $\tilde{I}$ eine optimale Packung:

Dazu wird das Aussehen einer optimalen Packung von $\tilde{I}$ betrachtet:

- Es gibt t_A Behältern, die nur ein einziges Objekt aus *A* enthalten. Objekte aus *B* oder *C* passen dort nicht mehr hinein.
- Kein Behälter enthält 3 oder mehr Objekte.
- Es gibt *t* Behälter mit 2 Objekten, davon jeweils höchstens eines aus *B*, eventuell beide aus *C*. Die Anzahl der Behälter mit 2 Objekten, von denen eines aus *B* und eines aus *C* kommt, sei t_{BC}; die Anzahl der Behälter mit 2 Objekten, von denen beide aus *C* kommt, sei t_{CC}.
- Es gibt t_B Behälter, die nur ein Objekt aus *B* enthalten.
- Es gibt t_C Behälter, die nur ein Objekt aus *C* enthalten; $t_C \le 1$.

$m^*(\tilde{I}) = t_A + t_{BC} + t_{CC} + t_B + t_C$ und $|C| = t_{BC} + 2 \cdot t_{CC} + t_C$.

Bei der Abarbeitung von $\tilde{I}$ durch den FirstfitDecreasing-Algorithmus werden zuerst die Objekte aus $A \cup B$ gepackt; dazu werden $t_A + t_{BC} + t_B$ Behälter benötigt. Das erste Objekt $c \in C$ kommt in den Behälter, der ein Objekt aus *B* enthält, und zwar das größte Objekt $b \in B$ mit $b + c \le 1$. Ein Objekt aus *C* kommt erst dann in einen neuen Behälter oder in einen Behälter, der bereits ein Element aus *C* enthält, wenn es keinen Behälter gibt, der genau ein Element aus *B* enthält und zu dem man es legen könnte. Daher kommen auch t_{BC} Objekte aus *C* in Behälter mit einem Element aus *B*. Für die übrigen Objekte aus *C* benötigt der FirstfitDecreasing-Algorithmus noch $2 \cdot t_{CC} + t_C$ Behälter. Insgesamt ergibt sich

$m(\tilde{I}, \mathbf{FFD}(\tilde{I})) = t_A + t_{BC} + t_{CC} + t_B + t_C = m^*(\tilde{I})$.

Damit ergibt sich $m^*(I) \le m(I, \mathbf{FFD}(I)) = m(\tilde{I}, \mathbf{FFD}(\tilde{I})) = m^*(\tilde{I}) \le m^*(I)$; die letzte Ungleichung folgt aus der Inklusion $\tilde{I} \subseteq I$. Das bedeutet $m(I, \mathbf{FFD}(I)) = m^*(I)$. ///

Auch der FirstfitDecreasing-Algorithmus kein 1,5-approximativer Algorithmus, sondern nur asymptotisch 1,5-approximativ (siehe Kapitel 6.2).

Eine genauere Analyse des FirstfitDecreasing-Algorithmus zeigt, daß die in Satz 6.1-3 angegebene Schranke 1,5 verbessert werden kann. Es läßt sich zeigen, daß für jede Instanz $I = [a_1, \ldots, a_n]$ des Binpacking-Minimierungsproblems die Abschätzung $m(I, \mathbf{FFD}(I)) \le 11/9 \cdot m^*(I) + 4$ gilt. Das folgende Beispiel zeigt, daß die $11/9$-Grenze nicht verbessert werden kann: Man betrachte die Instanz *I*, die aus $n = 5m$ vielen Objekten besteht, und zwar *m* Objekte der Größe $1/2 + d$ mit $0 < d < 1/40$, *m* Objekte der Größe $1/4 + 2d$, *m* Objekte der Größe $1/4 + d$ und $2m$ Objekte der Größe $1/4 - 2d$. Die Objekte dieser Instanz

sind nach absteigender Größe sortiert. Es ist $m^*(I) = 3m/2$ und $m(I, \mathbf{FFD}(I)) = 11m/6$, also $m(I, \mathbf{FFD}(I)) = 11/9 \cdot m^*(I)$.

BestfitDecreasing-Algorithmus zur Approximation von Binpacking:

Eingabe: $I = [a_1, \ldots, a_n]$,
$a_1, \ldots, a_n$ („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, \ldots, n$

Verfahren: Sortiere die Objekte $a_1, \ldots, a_n$ nach absteigender Größe. Die Objekte erhalten im folgenden wieder die Benennung $a_1, \ldots, a_n$, d.h. es gilt $a_1 \geq \ldots \geq a_n$.
$B_1 := a_1$;
Sind $a_1, \ldots, a_i$ bereits in die Behälter $B_1, \ldots, B_j$ gelegt worden, ohne daß einer dieser Behälter überläuft, so lege a_{i+1} in den Behälter unter $B_1, \ldots, B_j$, der den kleinsten freien Platz aufweist. Falls a_{i+1} in keinen der Behälter $B_1, \ldots, B_j$ paßt, lege a_{i+1} in einen neuen (bisher leeren) Behälter B_{j+1}.

Ausgabe: $\mathbf{BFD}(I) =$ die benötigten nichtleeren Behälter $[B_1, \ldots, B_k]$.

Satz 6.1-4:
Ist $I = [a_1, \ldots, a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $m(I, \mathbf{BFD}(I)) \leq 11/9 \cdot m^*(I) + 4$.

Auch der BestfitDecreasing-Algorithmus nur asymptotisch $11/9$-approximativ (siehe Kapitel 6.2). Das obige Beispiel zeigt, daß auch für diesen Algorithmus die $11/9$-Grenze nicht verbessert werden kann.

Die folgende Zusammenstellung zeigt noch einmal die mit den verschiedenen Approximationsalgorithmen für das Binpacking-Problem zu erzielenden Approximationsgüten. In der letzten Zeile ist dabei ein Algorithmus erwähnt, der in einem gewissen Sinne (siehe Kapitel 6.2) unter allen approximativen Algorithmen mit polynomieller Laufzeit eine optimale relative Approximationsgüte erzielt. Zu beachten ist ferner, daß im Sinne der Definition die Algorithmen Firstfit, FirstfitDecreasing und BestfitDecreasing nicht *r*-approximativ (mit *r* = 1,7 bzw. *r* = 1,5 bzw. *r* = 1,22), sondern nur asymptotisch *r*-approximativ sind.

Approximationsalgorithmus	Approximationsgüte
Nextfit	$m(I,\mathbf{NF}(I)) \le \begin{cases} (1+a_{\max}/(1-a_{\max}))\cdot m^*(I) & \text{falls } 0 < a_{\max} \le 1/2 \\ 2\cdot m^*(I) & \text{falls } 1/2 < a_{\max} \le 1 \end{cases}$
Firstfit	$m(I,\mathbf{FF}(I)) \le 1{,}7\cdot m^*(I)+2$
FirstfitDecreasing	$m(I,\mathbf{FFD}(I)) \le 1{,}5\cdot m^*(I)+1$ (Satz 6.1-3) $m(I,\mathbf{FFD}(I)) \le 11/9\cdot m^*(I)+4$; $\quad 11/9 = 1{,}222$
BestFitDecreasing	$m(I,\mathbf{BFD}(I)) \le 11/9\cdot m^*(I)+4$
Simchi-Levi, 1994	$m(I,\mathbf{SL}(I)) \le 1{,}5\cdot m^*(I)$

In Kapitel 6 wurde gezeigt, daß das 0/1-Rucksack-Maximierungsproblem unter der Annahme $\mathbf{P} \neq \mathbf{NP}$ nicht absolut approximierbar ist. Es gibt jedoch für dieses Problem einen 2-approximativen Algorithmus:

Das 0/1-Rucksackproblem als Maximierungsproblem
(maximum 0/1 knapsack problem)

Instanz: 1. $I = (A, M)$

$A = \{a_1, ..., a_n\}$ ist eine Menge von n Objekten und $M \in \mathbf{R}_{>0}$ die „Rucksackkapazität". Jedes Objekt a_i, $i = 1, ..., n$, hat die Form $a_i = (w_i, p_i)$; hierbei bezeichnet $w_i \in \mathbf{R}_{>0}$ das Gewicht und $p_i \in \mathbf{R}_{>0}$ den Wert (Profit) des Objekts a_i.

$size(I) = n$

2. $\text{SOL}(I) = \left\{ (x_1, ..., x_n) \mid x_i = 0 \text{ oder } x_i = 1 \text{ für } i = 1, ..., n \text{ und } \sum_{i=1}^{n} x_i \cdot w_i \le M \right\}$; man beachte, daß hier nur Werte $x_i = 0$ oder $x_i = 1$ zulässig sind

3. $m(I,(x_1, ..., x_n)) = \sum_{i=1}^{n} x_i \cdot p_i$ für $(x_1, ..., x_n) \in \text{SOL}(I)$

4. *goal* = max

Lösung: Eine Folge $x_1^*, ..., x_n^*$ von Zahlen mit

(1) $x_i^* = 0$ oder $x_i^* = 1$ für $i = 1, ..., n$

(2) $\sum_{i=1}^{n} x_i^* \cdot w_i \le M$

(3) $m\left(I,\left(x_1^*,...,x_n^*\right)\right)=\sum_{i=1}^{n} x_i^* \cdot p_i$ ist maximal unter allen möglichen Auswahlen $x_1,...,x_n$, die (1) und (2) erfüllen.

Der Approximationsalgorithmus **RUCK_APP** wird hier informell beschrieben:

1. Bei Eingabe einer Instanz $I=(A,M)$ sortiere man die Objekte nach absteigenden Werten p_i/w_i. Die Objekte werden in der durch diese Sortierung bestimmten Reihenfolge bearbeitet. Ein Objekt a_i wird dabei in den Rucksack gelegt, wenn es noch paßt; in diesem Fall wird $x_i=1$ gesetzt. Paßt das Objekt a_i nicht, dann wird $x_i=0$ gesetzt und zum nächsten Objekt übergegangen.
2. Es sei $p_{\max}=\max\{p_i \mid i=1,...,n\}$. Man vergleiche das Ergebnis im Schritt 1 mit der Rucksackfüllung, die man erhält, wenn man nur das Objekt mit dem Gewinn $p_{\max}$ allein in den Rucksack legt. Man nehme diejenige Rucksackfüllung, die den größeren Wert der Zielfunktion liefert.

Das Ergebnis des Verfahrens ist eine 0-1-Folge **RUCK_APP**$(I)=\left(x_1,...,x_n\right)$ mit dem Wert $m\left(I,\textbf{RUCK\_APP}(I)\right)$ der Zielfunktion.

Satz 6.1-5:
Für jede Instanz $I=(A,M)$ des 0/1-Rucksack-Maximierungsproblems gilt: $1\le m^*(I)/m\left(I,\textbf{RUCK\_APP}(I)\right)\le 2$, d.h. das 0/1-Rucksack-Maximierungsproblem liegt in **APX**.

Beweis:

Es sei a_j das erste Objekt, das im 1. Teil des Algorithmus **RUCK_APP** nicht mehr in den Rucksack paßt. Zu diesem Zeitpunkt hat der Rucksack eine Füllung $\overline{w}=\sum_{i=1}^{j-1} w_i \le M$. Es gilt also $w_j > M-\overline{w}$. Der bisher entstandene Profit ist $\overline{p}=\sum_{i=1}^{j-1} p_i$.

Es werde eine modifizierte Aufgabenstellung des 0/1-Rucksack-Maximierungsproblems betrachtet, in dem die Forderung „$x_i=0$ oder $x_i=1$“ durch „$0\le x_i\le 1$“ ersetzt ist. Für die so modifizierte Aufgabenstellung kann im 1. Teil des Algorithmus **RUCK_APP** das Objekt a_j noch in den Rucksack gelegt werden, jedoch nur mit einem Anteil $x_j=\left(M-\overline{w}\right)/w_j$. Der Rucksack ist dann gefüllt, d.h. im 1. Teil von **RUCK_APP** wird für die modifizierte Aufga-

benstellung $x_{j+1} = ... = x_n = 0$ gesetzt. Es läßt sich zeigen, daß jetzt bereits eine optimale Lösung des modifizierten 0/1-Rucksack-Maximierungsproblems gefunden ist, und zwar mit einem Profit $m_{MOD} = \overline{p} + \left((M - \overline{w})/w_j\right) \cdot p_j$. Da jede zulässige Lösung des (ursprünglichen) 0/1-Rucksack-Maximierungsproblems eine zulässige Lösung des modifizierten 0/1-Rucksack-Maximierungsproblems ist, folgt $m^*(I) \le \overline{p} + \left((M - \overline{w})/w_j\right) \cdot p_j < \overline{p} + p_j$ (die letzte Ungleichung ergibt sich aus $w_j > M - \overline{w}$).

Außerdem gilt $\overline{p} \le \max\{\overline{p}, p_{\max}\} \le m(I, \mathbf{RUCK\_APP}(I)) \le m^*(I)$.

Es werden zwei Fälle unterschieden:

1.Fall: $p_j \le \overline{p}$

Dann gilt $m^*(I) < 2 \cdot \overline{p} \le 2 \cdot m(I, \mathbf{RUCK\_APP}(I))$.

2.Fall: $p_j > \overline{p}$

Dann gilt $p_{\max} \ge p_j > \overline{p}$ und $m^*(I) < \overline{p} + p_j < 2 \cdot p_{\max} \le 2 \cdot m(I, \mathbf{RUCK\_APP}(I))$.

In beiden Fällen folgt die Abschätzung $1 \le m^*(I)/m(I, \mathbf{RUCK\_APP}(I)) \le 2$. ///

Gibt man die Instanz $I = \left(\left(\frac{M}{2}+1, \frac{M}{2}+2\right), \left(\frac{M}{2}, \frac{M}{2}\right), \left(\frac{M}{2}, \frac{M}{2}\right), M\right)$ mit $M > 4$ in den Algorithmus **RUCK_APP** ein, so liefert er das Ergebnis $x_1 = 1$, $x_2 = 0$ und $x_3 = 0$ und den Profit $m(I, \mathbf{RUCK\_APP}(I)) = M/2 + 2$. Die optimale Lösung ist jedoch Ergebnis $x_1^* = 0$, $x_2^* = 1$ und $x_3^* = 1$ mit dem Profit $m^*(I) = M > M/2 + 2$. Damit ist

$$m^*(I) \Big/ m(I, \mathbf{RUCK\_APP}(I)) = \frac{M}{M/2+2} = \frac{2M}{M+4} = \frac{2 \cdot (M+4) - 8}{M+4} = 2 - \frac{8}{M+4}.$$

Bei genügend großem *M* kommt dieser Wert beliebig dicht an 2 heran, so daß die Abschätzung in Satz 6.1-5 nicht verbessert werden kann.

Der folgende Satz zeigt, daß es unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ nicht für jedes Optimierungsproblem aus **NPO** einen approximativen Algorithmus gibt. Dazu sei noch einmal das Handlungsreisenden-Minimierungsproblem mit ungerichteten Graphen angegeben:

Das Handlungsreisenden-Minimierungsproblem

Instanz: 1. $G=(V,E,w)$

$G=(V,E,w)$ ist ein gewichteter ungerichteter Graph mit der Knotenmenge $V=\{v_1,...,v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E\subseteq V\times V$; die Funktion $w: E\to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e\in E$ ein nichtnegatives Gewicht

2. $\mathrm{SOL}(G)=\left\{T \mid T=\left\langle (v_{i_1},v_{i_2}),(v_{i_2},v_{i_3}),...,(v_{i_{n-1}},v_{i_n}),(v_{i_n},v_{i_1})\right\rangle \text{ ist eine Tour durch } G\right\}$

3. für $T\in \mathrm{SOL}(G)$, $T=\left\langle (v_{i_1},v_{i_2}),(v_{i_2},v_{i_3}),...,(v_{i_{n-1}},v_{i_n}),(v_{i_n},v_{i_1})\right\rangle$, ist die Zielfunktion definiert durch $m(G,T)=\sum_{j=1}^{n-1} w(v_{i_j},v_{i_{j+1}})+w(v_{i_n},v_{i_1})$

4. $goal=\min$

Lösung: Eine Tour $T^*\in \mathrm{SOL}(G)$, die minimale Kosten (unter allen möglichen Touren durch G) verursacht, und $m(G,T^*)$.

Satz 6.1-6:

Ist $\mathbf{P}\neq\mathbf{NP}$, so gibt es keinen r-approximativen Algorithmus für das Handlungsreisenden-Minimierungsproblem (mit $r\in\mathbf{R}_{\geq 1}$).

Ist $\mathbf{P}\neq\mathbf{NP}$, so ist $\mathbf{APX}\subset\mathbf{NPO}$.

Beweis:

Die Argumentation folgt der Idee aus dem Beweis von Satz 6-3.

Es wird angenommen, daß es bei Vorgabe des Wertes $r\in\mathbf{R}_{\geq 1}$ einen r-approximativen Algorithmus **A** für das Handlungsreisenden-Minimierungsproblem gibt und zeigt dann, daß dieser (mit einer einfachen Modifikation) dazu verwendet werden kann, das Problem des Hamiltonschen Kreises in einem ungerichteten Graphen (UNGERICHTETER HAMILTONKREIS, vgl. Kapitel 5.4) in polynomieller Zeit zu entscheiden. Da dieses **NP**-vollständig ist, folgt $\mathbf{P}=\mathbf{NP}$ (vgl. Satz 5.1-1).

Das Problem des Hamiltonschen Kreises in einem ungerichteten Graphen (UNGERICHTETER HAMILTONKREIS) lautet wie folgt:

Instanz: G,
$G=(V,E)$ ist ein ungerichteter Graph mit $V=\{v_1,\ldots,v_n\}$.

Lösung: Entscheidung „ja“, falls gilt:
G besitzt einen Hamiltonschen Kreis. Dieses ist eine Anordnung $(v_{p(1)},v_{p(2)},\ldots,v_{p(n)})$ der Knoten mittels einer Permutation π der Knotenindizes, so daß für $i=1,\ldots,n-1$ gilt: $(v_{p(i)},v_{p(i+1)})\in E$ und $(v_{p(n)},v_{p(1)})\in E$.

Es sei **A** ein r-approximativen Algorithmus **A** für das Handlungsreisenden-Minimierungsproblem, d.h. für alle Instanzen I dieses Problems gilt $m(I,\mathbf{A}(I))\le r\cdot m^*(I)$. Man kann $r>1$ annehmen, denn für $r=1$ ermittelt **A** bereits eine optimale Lösung. Es wird ein Algorithmus $\tilde{\mathbf{A}}$ für UNGERICHTETER HAMILTONKREIS konstruiert, der folgendendermaßen arbeitet:

Bei Eingabe eines ungerichteten Graphen $G=(V,E)$ mit $V=\{v_1,\ldots,v_n\}$ konstruiert $\tilde{\mathbf{A}}$ einen vollständigen bewerteten ungerichteten Graphen $\tilde{G}=(V,\tilde{E})$ (die Knotenmenge wird beibehalten) mit $\tilde{E}=\{(v_i,v_j)\mid v_i\in V \text{ und } v_j\in V\}$ und der Kantenbewertung $w:\tilde{E}\to\mathbf{R}_{\ge 0}$, die durch

$$w(v_i,v_j)=\begin{cases}1 & \text{falls } (v_i,v_j)\in E\\ nr & \text{sonst}\end{cases}$$ definiert ist. Diese Konstruktion erfolgt in polynomieller Zeit, da höchsten $O(n^2)$ viele Kanten hinzuzufügen und $O(n^2)$ viele Kanten zu bewerten sind. Alle Bewertungen haben eine Länge der Ordnung $O(\log(n))$. Der Graph $\tilde{G}=(V,\tilde{E})$ wird in den Algorithmus **A** eingegeben und das Ergebnis $m(\tilde{G},\mathbf{A}(\tilde{G}))$ mit dem Wert $r\cdot n$ verglichen. $\tilde{\mathbf{A}}$ trifft die Entscheidung

$$\tilde{\mathbf{A}}(G)=\begin{cases}\texttt{ja} & \text{für } m(\tilde{G},\mathbf{A}(\tilde{G}))\le r\cdot n\\ \texttt{nein} & \text{für } m(\tilde{G},\mathbf{A}(\tilde{G}))> r\cdot n\end{cases}.$$

Falls **A** ein polynomiell zeitbeschränkter Algorithmus ist, dann auch $\tilde{\mathbf{A}}$.

Besitzt G einen Hamiltonkreis (mit Länge n), dann ist $m^*(\tilde{G})=n$, und $\tilde{\mathbf{A}}(G)=\texttt{ja}$, da **A** ein r-approximativen Algorithmus **A** für das Handlungsreisenden-Minimierungsproblem ist und damit $m(\tilde{G},\mathbf{A}(\tilde{G}))\le r\cdot m^*(\tilde{G})=r\cdot n$ ist.

Besitzt G keinen Hamiltonkreis, dann enthält jeder Hamiltonkreis in $\tilde{G}$ mindestens eine Kante, die mit $r\cdot n$ bewertet ist (da $\tilde{G}$ ein vollständiger Graph ist, enthält $\tilde{G}$ einen Hamiltonkreis). Damit ergibt sich $m(\tilde{G},\mathbf{A}(\tilde{G}))\ge m^*(\tilde{G})\ge n-1+r\cdot n>r\cdot n$, d.h. $\tilde{\mathbf{A}}(G)=\texttt{nein}$.

In beiden Fällen trifft $\tilde{\mathbf{A}}$ die korrekte Entscheidung. ///

Das **metrische Handlungsreisenden-Minimierungsproblem** liegt jedoch in **APX**. Dieses stellt eine zusätzliche Bedingung an die Gewichtsfunktion einer Eingabeinstanz, nämlich die Gültigkeit der **Dreiecksungleichung**:

Für $v_i \in V$, $v_j \in V$ und $v_k \in V$ gilt $w(v_i, v_j) \leq w(v_i, v_k) + w(v_k, v_j)$.

Auch hierbei ist das zugehörige Entscheidungsproblem **NP**-vollständig. Es gibt (unabhängig von der Annahme $\mathbf{P} \neq \mathbf{NP}$) im Gegensatz zum (allgemeinen) Handlungsreisenden-Minimierungsproblem für dieses Problem einen 1,5-approximativen Algorithmus (siehe Literatur). Es ist nicht bekannt, ob es einen Approximationsalgorithmus mit einer kleineren relativen Approximatiosgüte gibt oder ob aus der Existenz eines derartigen Algorithmus bereits $\mathbf{P} = \mathbf{NP}$ folgt.

Hat man für ein Optimierungsproblem aus **APX** einen *r*-approximativen Algorithmus gefunden, so stellt sich die Frage, ob dieser noch verbessert werden kann, d.h. ob es einen *t*-approximativen Algorithmus mit $1 \leq t < r$ gibt. Der folgende Satz (gap theorem) besagt, daß man unter Umständen an Grenzen stößt, daß es nämlich Optimierungsprobleme in **APX** gibt, die in polynomieller Zeit nicht beliebig dicht approximiert werden können, außer es gilt $\mathbf{P} = \mathbf{NP}$.

Satz 6.1-7:

Es sei Π' ein **NP**-vollständiges Entscheidungsproblem über Σ'^* und Π aus **NPO** ein Minimierungsproblem über Σ^*. Es gebe zwei in polynomieller Zeit berechenbare Funktionen $f : \Sigma'^* \to \Sigma^*$ und $c : \Sigma'^* \to \mathbf{N}$ und eine Konstante $e > 0$ mit folgender Eigenschaft:

Für jedes $x \in \Sigma'^*$ ist $m^*(f(x)) = \begin{cases} c(x) & \text{für } x \in L_{\Pi'} \\ c(x) \cdot (1+e) & \text{für } x \notin L_{\Pi'} \end{cases}$.

Dann gibt es keinen polynomiell zeitbeschränkten *r*-approximativen Algorithmus für Π mit $r < 1+e$, außer $\mathbf{P} = \mathbf{NP}$.

Beweis:

Angenommen, es gibt einen polynomiell zeitbeschränkten *r*-approximativen Algorithmus **A** für Π mit $r < 1+e$. Dann kann man daraus einen polynomiell zeitbeschränkten Algorithmus $\mathbf{A}'$ für Π' konstruieren, der genau $L_{\Pi'}$ erkennt. Das bedeutet $\mathbf{P} = \mathbf{NP}$.

Die Arbeitsweise von $\mathbf{A}'$ wird informell beschrieben:

Bei Eingabe von $x \in \Sigma'^{*}$ in $\mathbf{A}'$ werden in polynomieller Zeit die Werte $f(x)$ und $c(x)$ berechnet. Der Wert $f(x)$ wird in den polynomiell zeitbeschränkten *r*-approximativen Algorithmus **A** für Π eingegeben und der Näherungswert $m(f(x), \mathbf{A}(f(x)))$ mit $c(x) \cdot (1+e)$ verglichen. $\mathbf{A}'$ trifft die Entscheidung

$$\mathbf{A}'(x) = \begin{cases} \texttt{ja} & \text{für } m(f(x), \mathbf{A}(f(x))) < c(x) \cdot (1+e) \\ \texttt{nein} & \text{für } m(f(x), \mathbf{A}(f(x))) \geq c(x) \cdot (1+e) \end{cases}.$$

$\mathbf{A}'$ ist ein polynomiell zeitbeschränkter Entscheidungsalgorithmus. Zu zeigen ist, daß die von $\mathbf{A}'$ getroffene Entscheidung korrekt ist, d.h. daß $\mathbf{A}'(x) = \texttt{ja}$ genau dann gilt, wenn $x \in L_{\Pi'}$ ist.

Es sei $x \in L_{\Pi'}$. Da **A** *r*-approximativ ist, gilt $\frac{m(f(x), \mathbf{A}(f(x)))}{m^{*}(f(x))} \leq r < 1+e$. Wegen $x \in L_{\Pi'}$ ist $m^{*}(f(x)) = c(x)$. Also ist $m(f(x), \mathbf{A}(f(x))) < m^{*}(f(x)) \cdot (1+e) = c(x) \cdot (1+e)$, und $\mathbf{A}'(x) = \texttt{ja}$.

Es sei $x \notin L_{\Pi'}$. Dann folgt $m^{*}(f(x)) = c(x) \cdot (1+e)$,
$m(f(x), \mathbf{A}(f(x))) \geq m^{*}(f(x)) = c(x) \cdot (1+e)$ und $\mathbf{A}'(x) = \texttt{nein}$.

In beiden Fällen trifft $\mathbf{A}'$ die korrekte Entscheidung. ///

Bemerkung: Der Beweis von Satz 6.1-7 zeigt, daß Satz 6.1-7 gültig bleibt, wenn die dort formulierte Voraussetzung über $m^{*}(f(x))$ ersetzt durch die Voraussetzung

$$\text{Für jedes } x \in \Sigma'^{*} \text{ ist } m^{*}(f(x)) \begin{cases} = c(x) & \text{für } x \in L_{\Pi'} \\ \geq c(x) \cdot (1+e) & \text{für } x \notin L_{\Pi'} \end{cases}.$$

Ein entsprechender Satz gilt für Maximierungsprobleme:

Satz 6.1-8:

Es sei Π' ein **NP**-vollständiges Entscheidungsproblem über Σ'^* und Π aus **NPO** ein Maximierungsproblem über Σ^*. Es gebe zwei in polynomieller Zeit berechenbare Funktionen $f:\Sigma'^* \to \Sigma^*$ und $c:\Sigma'^* \to \mathbf{N}$ und eine Konstante $e>0$ mit folgender Eigenschaft:

Für jedes $x \in \Sigma'^*$ ist $m^*(f(x)) = \begin{cases} c(x) & \text{für } x \in L_{\Pi'} \\ c(x)\cdot(1-e) & \text{für } x \notin L_{\Pi'} \end{cases}$.

Dann gibt es keinen polynomiell zeitbeschränkten *r*-approximativen Algorithmus für Π mit $r < 1/(1-e)$, außer $\mathbf{P} = \mathbf{NP}$.

Mit Hilfe dieser Sätze läßt sich zeigen beispielsweise, daß die Grenze *r* = 1,5 für einen *r*-approximativen (polynomiell zeitbeschränkten) Algorithmus unter der Annahme $\mathbf{P} \neq \mathbf{NP}$ für das Binpacking-Minimierungsproblem optimal ist:

Satz 6.1-9:

Ist $\mathbf{P} \neq \mathbf{NP}$, dann gibt es keinen *r*-approximativen Algorithmus für das Binpacking-Minimierungsproblem mit $r \leq 3/2 - e$ für $e > 0$.

Beweis:

In Satz 6.1-7 übernimmt das Binpacking-Minimierungsproblem die Rolle von Π; für Π' wird das **NP**-vollständige Partitionenproblem (siehe Kapitel 5.4) genommen. Es wird definiert durch

Instanz: $I = [a_1, \ldots, a_n]$,
$a_1, \ldots, a_n$ sind natürliche Zahlen.

Lösung: Entscheidung „ja“ genau dann, wenn gilt:
Es gibt eine Aufteilung der Menge $\{a_1, \ldots, a_n\}$ in zwei disjunkte Teile I_1 und I_2 mit $\sum_{a_i \in I_1} a_i = \sum_{a_i \in I_2} a_i$.

Es werden zwei Abbildungen *f* und *c* definiert, die den Bedingungen in Satz 6.1-7 genügen. Die Abbildung f ordnet jeder Instanz $I = [a_1, \ldots, a_n]$ des Partitionenproblems eine Instanz $f(I)$ des Binpacking-Minimierungsproblems zu.

Die Funktion *c* mit $c(I) = 2$ für alle Instanzen *I* des Partitionenproblems ist trivialerweise in polynomieller Zeit berechenbar.

Für eine Instanz $I=[a_1,...,a_n]$ des Partitionenproblems sei $B=\sum_{a_i\in I} a_i$. Es wird $f(I)=[(2a_1)/B,...,(2a_n)/B]$ definiert, falls sich dadurch eine Instanz des Binpacking-Minimierungsproblems ergibt, d.h. falls $(2a_i)/B\le 1$ für $i=1,...,n$ gilt. Ansonsten wird $f(I)=[1,1,1]$ gesetzt. In jedem Fall ist $f(I)$ eine Instanz des Binpacking-Minimierungsproblems. Da dieses in **NPO** liegt und wegen Satz 2.1-3 ist $f(I)$ in polynomieller Zeit berechenbar.

Ist $I\in L_{\Pi'}$, d.h. I ist eine „ja"-Instanz des Partitionenproblems bzw. es gibt eine Aufteilung der Menge $\{a_1,...,a_n\}$ in zwei disjunkte Teile I_1 und I_2 mit $\sum_{a_i\in I_1} a_i=\sum_{a_i\in I_2} a_i$, dann gilt $a_i\le B/2$ für $i=1,...,n$, denn sonst könnte man I nicht in zwei gleichgroße Teile aufteilen. Daher ist $(2a_i)/B\le 1$, $\sum_{a_i\in I_1} a_i=\sum_{a_i\in I_2} a_i=B/2$ und $\sum_{a_i\in I_1}(2a_i)/B=\sum_{a_i\in I_2}(2a_i)/B=1$. Zur Packung der Instanz $f(I)=[(2a_1)/B,...,(2a_n)/B]$ des Binpacking-Minimierungsproblems sind genau 2 Behälter erforderlich, d.h. $m^*(f(I))=2=c(I)$..

Ist $I\notin L_{\Pi'}$ und $[(2a_1)/B,...,(2a_n)/B]$ keine Instanz des Binpacking-Minimierungsproblems, d.h. es gibt mindestens ein a_i mit $(2a_i)/B>1$, dann ist $f(I)=[1,1,1]$ und $m^*(f(I))=3$.
Ist $I\notin L_{\Pi'}$ und $[(2a_1)/B,...,(2a_n)/B]$ eine Instanz des Binpacking-Minimierungsproblems, dann ist $f(I)=[(2a_1)/B,...,(2a_n)/B]$ und $m^*(f(I))\ge 3$.
In beiden Fällen gilt $m^*(f(I))\ge 3=2\cdot\frac{3}{2}=c(I)\cdot\left(1+\frac{1}{2}\right)$.

Aus Satz 6.1-7 zusammen mit der sich dort anschließenden Bemerkung folgt, daß es keinen polynomiell zeitbeschränkten *r*-approximativen Algorithmus für das Binpacking-Minimierungsproblem mit $r<1+1/2=1.5$ gibt, außer $\mathbf{P}=\mathbf{NP}$. ///

6.2 Polynomiell zeitbeschränkte und asymptotische Approximationsschemata

In vielen praktischen Anwendungen möchte man die relative Approximationsgüte verbessern. Dabei ist man sogar bereit, längere Laufzeiten der Approximationsalgorithmen in Kauf zu nehmen, solange sie noch polynomielles Laufzeitverhalten bezüglich der Größe der Eingabeinstanzen haben. Bezüglich der relativen Approximationsgüte *r* akzeptiert man eventuell sogar ein Laufzeitverhalten, das exponentiell von $1/(r-1)$ abhängt: Je besser die Approximation ist, um so größer ist die Laufzeit. In vielen Fällen kann man so eine optimale Lösung beliebig dicht approximieren, allerdings zum Preis eines dramatischen Anstiegs der Rechenzeit.

Es sei Π ein Optimierungsproblem aus **NPO**. Ein Algorithmus **A** heißt **polynomiell zeitbeschränktes Approximationsschema** (polynomial-time approximation scheme) für Π, wenn er für jede Eingabeinstanz *x* von Π und für jede rationale Zahl $r > 1$ bei Eingabe von (x,r) eine *r*-approximative Lösung für *x* in einer Laufzeit liefert, die polynomiell von *size*(*x*) abhängt.

Mit **PTAS** werde die Klasse der Optimierungsprobleme in **NPO** bezeichnet, für die es ein polynomiell zeitbeschränktes Approximationsschema gibt. Daß diese Klasse nicht leer ist, zeigt das folgende Berispiel.

Partitionen-Minimierungsproblem

Instanz:
1. $I = [a_1, \ldots, a_n]$
   $a_1, \ldots, a_n$ („Objekte“) sind natürliche Zahlen mit $a_i > 0$ für $i = 1, \ldots, n$
   $size(I) = n$
2. $\mathrm{SOL}(I) = \{[Y_1, Y_2] \mid [Y_1, Y_2] \text{ ist eine Partition (disjunkte Zerlegung) von } I\}$
3. Für $[Y_1, Y_2] \in \mathrm{SOL}(I)$ ist $m(I, [Y_1, Y_2]) = \max\left\{\sum_{a_i \in Y_1} a_i, \sum_{a_j \in Y_2} a_j\right\}$
4. *goal* = min

Lösung: Eine Partition der Objekte in zwei Teile $[Y_1, Y_2]$, so daß sich die Summen der Objekte in beiden Teilen möglichst wenig unterscheiden.

Der folgende in Pseudocode formulierte Algorithmus ist ein polynomiell zeitbeschränktes Approximationsschema für das Partitionen-Minimierungsproblem.

Polynomiell zeitbeschränktes Approximationsschema für das Partitionen-Minimierungsproblem

Eingabe: $I = [a_1, \ldots, a_n]$, rationale Zahl $r > 1$,
$a_1, \ldots, a_n$ („Objekte“) sind natürliche Zahlen mit $a_i > 0$ für $i = 1, \ldots, n$

Verfahren:

```
VAR Y1 : SET OF INTEGER;
    Y2 : SET OF INTEGER;
    k  : REAL;
    j  : INTEGER;
```

```
BEGIN
  IF r >= 2
   THEN BEGIN
          Y1 := {a1,...,an};
          Y2 := { };
        END
   ELSE BEGIN
          Sortiere die Objekte nach absteigender Größe; die dabei
          entstehende Folge sei (x1,...,xn);
          k := ⌈(2−r)/(r−1)⌉;
          { Phase 1: }
          finde eine optimale Partition [Y1,Y2] für [x1,...,xk];
          { Phase 2: }
          FOR j := k+1 TO n DO
            IF ∑xi∈Y1 xi <= ∑xi∈Y2 xi
            THEN Y1 := Y1 ∪ {xj}
            ELSE Y2 := Y2 ∪ {xj}
        END;
```

Ausgabe: $[Y_1, Y_2]$.

Satz 6.2-1:
Das Partitionen-Minimierungsproblem liegt in **PTAS**.

Beweis:

Es ist zu zeigen, daß der angegebene Algorithmus, der mit **A** bezeichnet werde, bei Eingabe einer Instanz $I = [a_1, \ldots, a_n]$ für das Partitionen-Minimierungsproblem und einer rationalen Zahl $r > 1$ eine r-approximative Lösung für I in einer Laufzeit liefert, die polynomiell von n abhängt.

Es werden $I = [a_1, \ldots, a_n]$ und r in den angegebenen Algorithmus eingegeben. Die Ausgabe sei $[Y_1, Y_2]$. O.B.d.A. sei $\sum_{a_i \in Y_1} a_i \geq \sum_{a_j \in Y_2} a_j$. Dann ist $m(I, \mathbf{A}(I)) = \sum_{a_i \in Y_1} a_i$. Es sei $w(I)$ die Summe aller Objekte der Eingabeinstanz I: $w(I) = \sum_{a_i \in I} a_i$, $w(Y_2)$ die Summe aller Objekte in Y_2: $w(Y_2) = \sum_{a_i \in Y_2} a_i$.

1. Fall: $r \geq 2$

Dann ist $m^*(I) \geq w(I)/2 \geq m(I, \mathbf{A}(I))/2$, also $m(I, \mathbf{A}(I)) \leq 2 \cdot m^*(I) \leq r \cdot m^*(I)$.

2. Fall: $1 \leq r < 2$

Es sei a_h das letzte zu Y_1 hinzugefügte Objekt.

Wurde a_h in Phase 1 des Algorithmus in Y_1 aufgenommen, so läßt sich $m(I, \mathbf{A}(I)) = m^*(I)$ zeigen. Es wird daher angenommen, daß a_h in Phase 2 des Algorithmus in Y_1 aufgenommen wurde. Es folgt nacheinander:

$m(I, \mathbf{A}(I)) - a_h \leq w(Y_2)$,

$2 \cdot m(I, \mathbf{A}(I)) - a_h \leq w(Y_2) + m(I, \mathbf{A}(I)) = w(I)$ und

$m(I, \mathbf{A}(I)) - w(I)/2 \leq a_h/2$.

Außerdem gilt wegen der Sortierung der Objekte $a_h \leq a_j$ für $j = 1, ..., k$. Diese Ungleichungen werden auf a_h aufsummiert zu dem Ergebnis:

$(k+1) \cdot a_h = a_h + k \cdot a_h \leq \sum_{j=1}^{k} a_j + a_h \leq w(I)$, also $a_h \leq w(I)/(k+1)$.

Es gilt $m(I, \mathbf{A}(I)) \geq w(I)/2 \geq w(Y_2)/2$ und $m^*(I) \geq w(I)/2$.

Insgesamt ergibt sich:

$$\frac{m(I, \mathbf{A}(I))}{m^*(I)} \leq \frac{m(I, \mathbf{A}(I))}{w(I)/2} \leq \frac{a_h/2 + w(I)/2}{w(I)/2} = 1 + \frac{a_h}{w(I)} \leq 1 + \frac{1}{k+1} \leq r;$$

die letzte Ungleichung folgt aus der Wahl von *k*, nämlich $k = \lceil (2-r)/(r-1) \rceil$:

$k \geq (2-r)/(r-1)$, d.h. $k+1 \geq= (2-r+r-1)/(r-1) = 1/(r-1) > 1/r$.

Mit $k(r) = \lceil (2-r)/(r-1) \rceil$ ist das Laufzeitverhalten von der Ordnung $O(n \cdot \log(n) + n^{k(r)})$. Der erste Term gibt die Laufzeit zum Sortieren der Objekte der Eingabeinstanz an, der zweite Term die Laufzeit zur Ermittlung einer optimalen Lösung für die ersten *k* Elemente in Phase 1. Die Laufzeit für Phase 2 ist von der Ordnung $O(n)$. Da $k(r) \in O(1/(r-1))$ ist das Laufzeitverhalten bei festem *r* polynomiell in der Größe *n* der Eingabeinstanz, jedoch exponentiell in *n* und $1/(r-1)$. ///

Offensichtlich ist $\mathbf{PTAS} \subseteq \mathbf{APX}$. In Kapitel 6.1 wurde erwähnt, daß es unter der Annahme $\mathbf{P} \neq \mathbf{NP}$ keinen *r*-approximativen Algorithmus für das Binpacking-Minimierungsproblem mit $r \leq 3/2 - e$ für $e > 0$ gibt. Daraus folgt:

Satz 6.2-2:

Ist $\mathbf{P} \neq \mathbf{NP}$, so gilt $\mathbf{PTAS} \subset \mathbf{APX} \subset \mathbf{NPO}$.

Es gibt in **PTAS** Optimierungsprobleme, die ein polynomiell zeitbeschränktes Approximationsschema **A** zulassen, das bei Eingabe einer Instanz *x* von Π und einer rationalen Zahl $r > 1$ eine *r*-approximative Lösung für *x* in einer Laufzeit liefert, die polynomiell von $|x|$ und $1/(r-1)$ abhängt.
Einen derartigen Algorithmus nennt man **voll polynomiell zeitbeschränktes Approximationsschema** (fully polynomial-time approximation scheme). Die Optimierungsprobleme, die einen derartigen Algorithmus zulassen, bilden die Klasse **FPTAS**.

In der Literatur werden Beispiele für Optimierungsprobleme genannt, die in **FPTAS** liegen.

Es läßt sich zeigen:

Satz 6.2-3:
Ist $\mathbf{P} \neq \mathbf{NP}$, so gilt $\mathbf{FPTAS} \subset \mathbf{PTAS} \subset \mathbf{APX} \subset \mathbf{NPO}$.

In Kapitel 6.1 wurden mehrere Approximationsalgorithmen für das Binpacking-Minimierungsproblem angegeben. Die Approximationsgüte $m(I, \mathbf{BFD}(I)) \leq 11/9 \cdot m^*(I) + 4$ von BestfitDecreasing zeigt, daß man (unabhängig von der Annahme $\mathbf{P} \neq \mathbf{NP}$) durchaus einen Approximationsalgorithmus entwerfen kann, dessen relative Approximationsgüte unterhalb der überhaupt für einen *r*-approximativen Algorithmus möglichen Untergrenze liegt. Eventuell gibt es sogar in einem erweiterten Sinne ein polynomiell zeitbeschränktes Approximationsschema für das Binpacking-Minimierungsproblem und andere Optimierungsprobleme. Diese Überlegung führt auf folgende Definition:

Es sei Π ein Optimierungsproblem aus **NPO**. Ein Algorithmus **A** heißt **asymptotisches Approximationsschema** für Π, wenn es eine Konstante *k* gibt, so daß gilt:
für jede Eingabeinstanz *x* von Π und für jede rationale Zahl $r > 1$ liefert **A** bei Eingabe von (x, r) eine (zulässige) Lösung, deren relative Approximationsgüte $R_{\mathbf{A}}(x)$ die Bedingung $R_{\mathbf{A}}(x) \leq r + k/m^*_{\Pi}(x)$ erfüllt. Außerdem ist die Laufzeit von **A** für jedes feste *r* polynomiell in der Größe *size*(*x*) der Eingabeinstanz.

Zur Erinnerung: die relative Approximationsgüte wurde definiert durch

$$R_{\mathbf{A}}(x) = \frac{m^*_{\Pi}(x)}{m(x, \mathbf{A}(x))}$$ bei einem Maximierungsproblem

bzw.

$$R_{\mathbf{A}}(x) = \frac{m(x, \mathbf{A}(x))}{m^*_{\Pi}(x)}$$ bei einem Minimierungsproblem.

Die Bedingung $R_{\mathbf{A}}(x) \leq r + k/m^*_{\Pi}(x)$ besagt also
bei einem Maximierungsproblem:

$m(x, \mathbf{A}(x)) \geq 1/r \cdot m^*_\Pi(x) - k'$ mit $k' = k/(r + k/m^*_\Pi(x)) \leq k/r$,

daher $m(x, \mathbf{A}(x)) \geq 1/r \cdot m^*_\Pi(x) - k/r$

bzw.

bei einem Minimierungsproblem: $m(x, \mathbf{A}(x)) \leq r \cdot m^*_\Pi(x) + k$.

Die Bezeichnung „asymptotisches Approximationsschema“ ist aus der Tatsache zu erklären, daß für „große“ Eingabeinstanzen x der Wert $m^*_\Pi(x)$ der Zielfunktion einer optimalen Lösung ebenfalls groß ist. Daher gilt in diesem Fall $\lim_{size(x)\to\infty} R_{\mathbf{A}}(x) \leq r$.

Die Klasse aller Optimierungsprobleme, die ein asymptotisches Approximationsschema zulassen, wird mit $\mathbf{PTAS}^\infty$ bezeichnet.

Satz 6.2-4:

Das Binpacking-Minimierungsproblem liegt in $\mathbf{PTAS}^\infty$, d.h. es kann asymptotisch beliebig dicht in polynomieller Zeit approximiert werden (auch wenn $\mathbf{P} \neq \mathbf{NP}$ gilt).

Es gilt sogar: Es gibt ein asymptotisches Approximationsschema, das polynomiell in der Problemgröße und in $1/(r-1)$ ist.

Die Klasse $\mathbf{PTAS}^\infty$ ordnet sich in die übrigen Approximationsklassen ein:

Satz 6.2-5:

Ist $\mathbf{P} \neq \mathbf{NP}$, so gilt $\mathbf{FPTAS} \subset \mathbf{PTAS} \subset \mathbf{PTAS}^\infty \subset \mathbf{APX} \subset \mathbf{NPO}$.

7 Weiterführende Konzepte

Die Analyse des Laufzeitverhaltens eines Algorithmus **A** im schlechtesten Fall $T_{\mathbf{A}}(n) = \max\{ t_{\mathbf{A}}(x) \mid x \in \Sigma^* \text{ und } |x| \le n \}$ liefert eine *Garantie* (obere Schranke) für die Zeit, die er bei einer Eingabe zur Lösung benötigt[5]. Für jede Eingabe $x \in \Sigma^*$ mit $|x| = n$ gilt $t_{\mathbf{A}}(x) \le T_{\mathbf{A}}(n)$. Dieses Verhalten ist oft jedoch nicht charakteristisch für das Verhalten bei „den meisten" Eingaben. Ist eine Wahrscheinlichkeitsverteilung *P* der Eingabewerte bekannt oder werden alle Eingaben als gleichwahrscheinlich aufgefaßt, so kann man das Laufzeitverhalten von **A** untersuchen, wie es sich im Mittel darstellt. Die **Zeitkomplexität** von **A** im **Mittel** wird definiert als

$$T_{\mathbf{A}}^{avg}(n) = \mathbf{E}[t_{\mathbf{A}}(x) \mid |x| = n] = \int_{|x|=n} t_{\mathbf{A}}(x) \cdot dP(x).$$

Die Untersuchung des mittleren Laufzeitverhaltens von Algorithmen ist in den meisten Fällen sehr viel schwieriger als die worst-case-Analyse. Trotzdem gibt es sehr ermutigende Ergebnisse, insbesondere bei der Lösung einiger „klassischer" Probleme. Der Einbau von Zufallsexperimenten in Algorithmen hat auf dem Gebiet der Approximationsalgorithmen für Optimierungsprobleme aus **NPO** gute Ergebnissen geliefert. In neuerer Zeit hat der Einsatz von probabilistischen Modellen zu neuen Erkenntnissen auf dem Weg zur Lösung der **P**-**NP**-Frage geführt.

7.1 Randomisierte Algorithmen

Im ansonsten deterministischen Algorithmus werden als zulässige Elementaroperationen Zufallsexperimente zugelassen. Ein derartiges Zufallsexperiment kann beispielsweise mit Hilfe eines Zufallszahlengenerators ausgeführt werden. So wird beispielsweise auf diese Weise entschieden, in welchem Teil einer Programmverzweigung der Algorithmus während seines Ablaufs fortgesetzt wird. Andere Möglichkeiten zum Einsatz eines Zufallsexperiments bestehen bei Entscheidungen zur Auswahl möglicher Elemente, die im weiteren Ablauf des Algorithmus als nächstes untersucht werden sollen.

Man nennt derartige Algorithmen **randomisierte Algorithmen**.

---

[5] $t_{\mathbf{A}}(x) =$ Anzahl der Anweisungen, die von **A** zur Berechnung von $\mathbf{A}(x)$ durchlaufen werden.

Grundsätzlich gibt es zwei Klassen randomisierter Algorithmen: **Las-Vegas-Verfahren**, die stets – wie von deterministischen Algorithmen gewohnt – ein korrektes Ergebnis berechnen. Daneben gibt es **Monte-Carlo-Verfahren**, die ein korrektes Ergebnis nur mit einer gewissen Fehlerwahrscheinlichkeit, aber in jedem Fall effizient, bestimmen. Häufig findet man bei randomisierten Algorithmen einen Trade-off zwischen Korrektheit und Effizienz.

Beispiele für randomisierte Algorithmen vom Las-Vegas-Typ sind:

- Einfügen von Primärschlüsselwerten in einen binären Suchbaum: Statt die Schlüssel $S_1, ..., S_n$ in dieser Reihenfolge sequentiell in den binären Suchbaum einzufügen, wird jeweils der nächste einzufügende Schlüssel aus den restlichen, d.h. noch nicht eingefügten Schlüsseln zufällig ausgewählt und in den binären Suchbaum eingefügt. Das Ergebnis ist eine mittlere Baumhöhe der Ordnung $O(\log(n))$
- `Quicksort` zum Sortieren Elementen, auf denen eine lineare Ordnung definiert ist.

Sortieren von Elementen, auf denen eine Ordnungsrelation besteht

Instanz: $x = \langle a_1, ..., a_n \rangle$

x ist eine Folge von Elementen der Form $a_i = (key_i, info_i)$; die Elemente sind bezüglich ihrer *key*-Komponente vergleichbar, d.h. es gilt für $a_i = (key_i, info_i)$ und $a_j = (key_j, info_j)$ die Beziehung $a_i \le a_j$ genau dann, wenn $key_i \le key_j$ ist.

Lösung: Eine Umordnung $p : [1:n] \to [1:n]$ der Elemente in *x*, so daß $\langle a_{p(1)}, ..., a_{p(n)} \rangle$ nach aufsteigenden Werten der *key*-Komponente sortiert ist, d.h. es gilt: $a_{p(i)} \le a_{p(i+1)}$ für $i = 1, ..., n-1$.

Als Randbedingung gilt, daß die Sortierung innerhalb der Folge *x* erfolgen soll, also insbesondere nicht zusätzlicher Speicherplatz der Größenordnung $O(n)$ erforderlich wird (**internes Sortierverfahren**).

Zur algorithmischen Behandlung wird eine Instanz *x* der Größe *n* in einem Feld `a` gespeichert, das definiert wird durch

```
CONST n         = ...;       { Problemgröße }

TYPE  idx_bereich = 1..n;
      key_typ     = ...;     { Typ der key-Komponente          }
      info_typ    = ...;     { Typ der info-Komponente         }
      entry_typ   = RECORD
                      key  : key_typ;
```

```
                    info : info_typ
                  END;
      feld_typ    = ARRAY [idx_bereich] OF entry_typ;
```

Die folgende Prozedur `interchange` vertauscht zwei Einträge miteinander:

```
PROCEDURE interchange (VAR x, y : entry_typ);
{ die Werte von x und y werden miteinander vertauscht              }

VAR z : entry_typ;

BEGIN { interchange }
  z := x;
  x := y;
  y := z;
END   { interchange };
```

Der Algorithmus `quicksort` beruht auf der Idee der Divide-and-Conquer-Methode und erzeugt die umsortierte Eingabe $\langle a_{p(1)}, ..., a_{p(n)} \rangle$:

Besteht *x* aus keinem oder nur aus einem einzigen Element, dann ist nichts zu tun. Besteht *x* aus mehr Elementen, dann wird aus *x* ein Element *e* (**Pivot-Element**) ausgewählt und das Problem der Sortierung von *x* in zwei kleinere Probleme aufgeteilt, nämlich der Sortierung aller Elemente, die kleiner als *e* sind (das sei das Problem der Sortierung der Folge *x(*lower*)*), und die Sortierung der Elemente die größer oder gleich *e* sind (das sei das Problem der Sortierung der Folge *x(*upper*)*); dabei wird als „Raum zur Sortierung" die Instanz *x* verwendet. Zuvor wird *e* an diejenige Position innerhalb von *x* geschoben, an der es in der endgültigen Sortierung stehen wird. Das Problem der Sortierung der erzeugten kleineren Probleme *x(*lower*)* und *x(*upper*)* wird (rekursiv) nach dem gleichen Prinzip gelöst. Die Position, an der *e* innerhalb von *x* in der endgültigen Sortierung stehen wird und damit die kleineren Probleme *x(*lower*)* und *x(*upper*)* bestimmt, sind dadurch gekennzeichnet, daß gilt:
$k < e$ für alle $k \in x(\text{lower})$ und $k \geq e$ für alle $k \in x(\text{upper})$.

Sortieralgorithmus mit `quicksort`:

Eingabe: $x = \langle a_1, ..., a_n \rangle$ ist eine Folge von Elementen der Form $a_i = (key_i, info_i)$; die Elemente sind bezüglich ihrer *key*-Komponente vergleichbar, d.h. es gilt für $a_i = (key_i, info_i)$ und $a_j = (key_j, info_j)$ die Beziehung $a_i \leq a_j$ genau dann, wenn $key_i \leq key_j$ ist

```
VAR a : feld_typ;          { Eingabefeld                    }
```

```
            i : idx_bereich;

          FOR i := 1 TO n DO
           BEGIN
             a[i].key   := key_i ;
             a.[i].info := info_i
           END;
```

Verfahren: Aufruf der Prozedur `quicksort (a, 1, n);`

Ausgabe: `a` mit `a[i].key` $\leq$ `a[i + 1].key` für $i = 1, ..., n-1$

```
PROCEDURE quicksort (VAR a     : feld_typ;
                         lower : idx_bereich;
                         upper : idx_bereich);

    VAR  t           : entry_typ;
         m           : idx_bereich;
         idx         : idx_bereich;
         zufalls_idx : idx_bereich;

BEGIN { quicksort }
  IF lower < upper
  THEN BEGIN  { ein Feldelement zufällig aussuchen und
                mit a[lower] austauschen                          }
              zufalls_idx := lower
                             + Trunc(Random * (upper - lower + 1));
              interchange (a[lower], a[zufalls_idx]);

              t := a[lower];
              m := lower;

              FOR idx := lower + 1 TO upper DO
                  { es gilt: a[lower+1] , ..., a[m] < t und
                             a[m+1], ... a[idx-1] >= t       }
                  IF a[idx].key < t.key
                  THEN BEGIN
                         m := m+1;
                         { vertauschen von a[m] und a[idx] }
                         interchange (a[m], a[idx])
                       END;

              { a[lower] und a[m] vertauschen }
              interchange (a[lower], a[m]);

              { es gilt: a[lower], ..., a[m-1] < a[m]  und
```

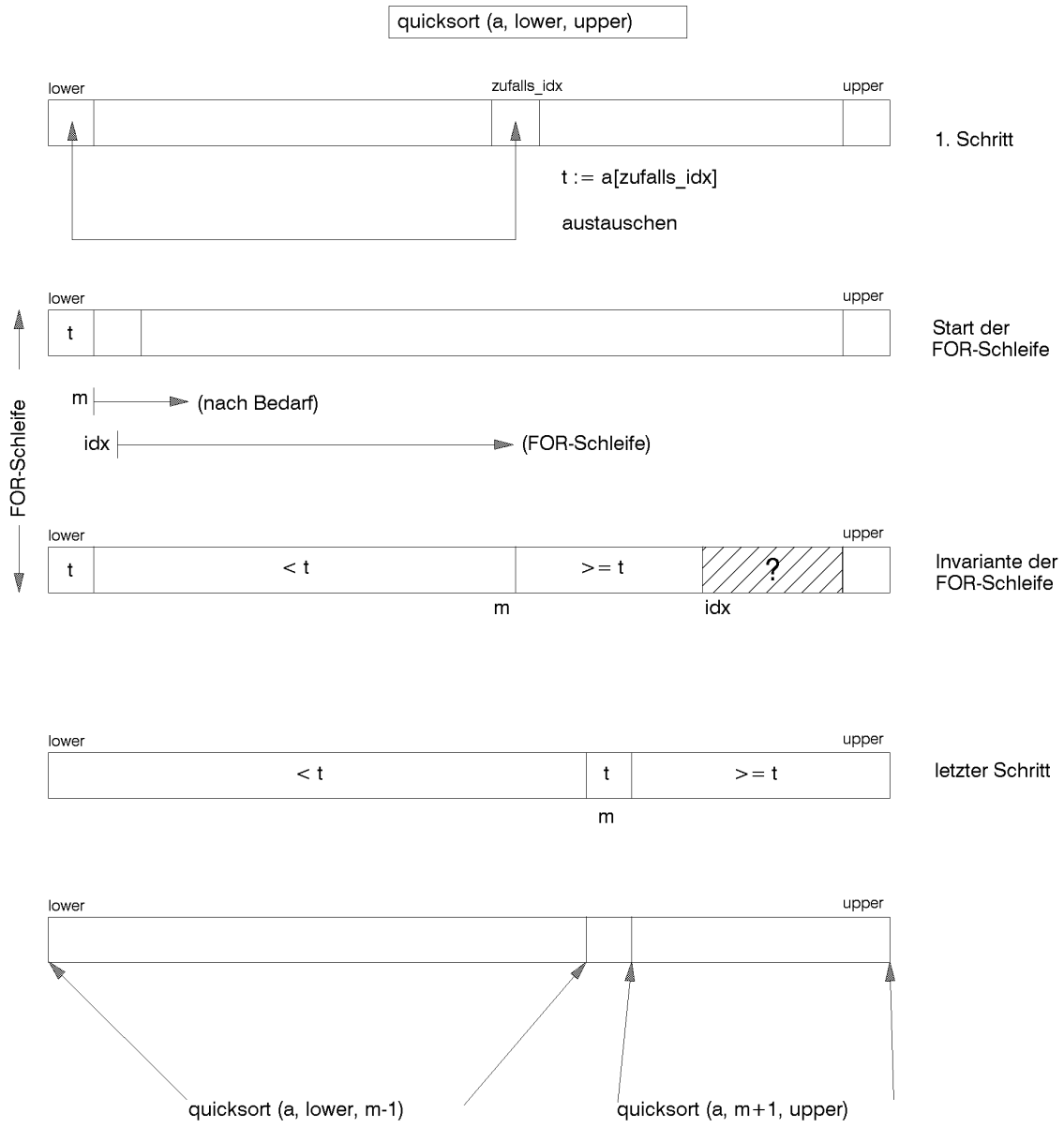
```
                        a[m] <= a[m+1], ..., a[upper]  }

            quicksort (a, lower, m-1);
            quicksort (a, m+1, upper);

          END { IF}

    END { quicksort };
```

Der Algorithmus stoppt, da in jedem `quicksort`-Aufruf ein Element an die endgültige Position geschoben wird und die Anzahl der Elemente in den `quicksort`-Aufrufen in den beiden restlichen Teilfogen zusammen um 1 verringert ist. Die Korrektheit des Algorithmus folgt aus der Invariante der `FOR`-Schleife.

Satz 7.1-1:

Das beschriebene Verfahren mit der Prozedur `quicksort` sortiert eine Folge $x = \langle a_1, ..., a_n \rangle$ aus n Elementen mit einer (worst-case-) Zeitkomplexität der Ordnung $O(n^2)$. Diese Schranke wird erreicht, wenn zufällig in jedem `quicksort`-Aufruf das Pivot-Element so gewählt wird, daß die Teilfolgen *x(*lower*)* kein Element und *x(*upper*)* alle restlichen Elemente enthalten (bzw. umgekehrt). Im Mittel ist das Laufzeitverhalten jedoch wesentlich besser, nämlich von der Ordnung $O(n \cdot \log(n))$.

Es läßt sich zeigen, daß jedes Sortierverfahren, das auf dem Vergleich von Elementgrößen beruht, eine untere Zeitkomplexität mindestens der Ordnung $O(n \cdot \log(n))$ und eine mittlere Zeitkomplexität ebenfalls mindestens der Ordnung $O(n \cdot \log(n))$ hat, so daß das Verfahren mit der Prozedur `quicksort` optimales Laufzeitverhalten im Mittel aufweist. Daß das Verfahren mit der Prozedur `quicksort` in der Praxis ein Laufzeitverhalten zeigt, das fast alle anderen Sortierverfahren schlägt, liegt daran, daß die „ungünstigen“ Eingaben für `quicksort` mit gegen 0 gehender Wahrscheinlichkeit vorkommen und die Auswahl des Pivot-Elements zufällig erfolgt.



Beispiele für randomisierte Algorithmen vom Monte-Carlo-Typ sind die heute üblichen Primzahltests, die hier informell beschrieben werden sollen (Details findet man in der angegebenen Literatur).

Um eine natürliche Zahl $n > 2$ auf Primzahleigenschaft zu testen, könnte man alle Primzahlen von 2 bis $\lfloor \sqrt{n} \rfloor$ daraufhin untersuchen, ob es eine von ihnen gibt, die n teilt. Wenn die Zahl n

nämlich zusammengesetzt ist, d.h. keine Primzahl ist, hat sie einen Primteiler *p* mit $p \le \sqrt{n}$. Umgekehrt, falls alle Primzahlen *p* mit $p \le \sqrt{n}$ keine Teiler von *n* sind, dann ist *n* selbst eine Primzahl. Die Primzahlen könnte man etwa systematisch erzeugen (siehe Literatur unter dem Stichwort „Sieb des Eratosthenes“) oder man könnte sie einer Primzahltabelle (falls vorhanden) entnehmen. Allerdings ist dieser Ansatz für sehr große Werte von *n* nicht praktikabel und benötigt exponentiellen Rechenaufwand (in der Anzahl der Stellen von *n*): Die Anzahl der Stellen $b(n)$ einer Zahl $n \ge 1$ im Zahlensystem zur Basis *B* ist $b(n) = \lfloor \log_B(n) \rfloor + 1 = \lceil \log_B(n+1) \rceil$, d.h. $b(n) \in O(\log(n))$. Die Anzahl der Primzahlen unterhalb $\lfloor \sqrt{n} \rfloor$ beträgt für große *n* nach dem Primzahlsatz der Zahlentheorie $p\left(\sqrt{n}\right) \sim \frac{n^{1/2}}{\ln\left(n^{1/2}\right)}$, also ein Wert der Ordnung $O\left(\frac{2^{b(n)/2}}{b(n)}\right)$. Jede in Frage kommende Primzahl muß daraufhin untersucht werden, ob sie *n* teilt. Dazu sind mindestens $O\left((b(n))^2\right)$ viele Bitoperationen erforderlich. Daher ist der Gesamtaufwand mindestens von der Ordnung $O\left(b(n) \cdot 2^{b(n)/2}\right)$.

Durch Anwendung zahlentheoretischer Erkenntnisse hat man versucht, effiziente Primzahltests zu entwickeln. Der bisher bekannte schnellste Algorithmus zur Überprüfung einer Zahl *n* auf Primzahleigenschaft, der APRCL-Test, hat eine Laufzeit der Ordnung $O\left((\log(n))^{c(\log(\log(\log(n))))}\right)$ mit einer Konstanten $c > 0$. Auch das ist keine polynomielle Laufzeit.

Es sind daher andere Ansätze für effiziente Primzahltests erforderlich. Als erfolgreich haben sich hierbei **probabilistische Algorithmen** erwiesen.

Um zu testen, ob eine Zahl *n* eine Primzahl ist oder nicht, versucht man, einen **Zeugen (witness) für die Primzahleigenschaft von** *n* zu finden. Ein Zeuge ist dabei eine Zahl *a* mit $1 \le a \le n-1$, der eine bestimmte Eigenschaft zukommt, aus der man *vermuten* kann, daß *n* Primzahl ist. Dabei muß diese Eigenschaft bei Vorgabe von *a* einfach zu überprüfen sein, und nach Auffinden einiger weniger Zeugen für die Primzahleigenschaft von *n* muß der Schluß gültig sein, daß *n* mit hoher Wahrscheinlichkeit eine Primzahl ist. Die Eigenschaft „die Primzahl *p* mit $2 \le p \le \lfloor \sqrt{n} \rfloor$ ist kein Teiler von *n*“ ist dabei nicht geeignet, da man im allgemeinen zu viele derartige Zeugen zwischen 2 und $\lfloor \sqrt{n} \rfloor$ bemühen müßte, um sicher auf die Primzahleigenschaft schließen zu können.

Es sei $E(a)$ eine (noch genauer zu definierende) Eigenschaft, die einer Zahl *a* mit $1 \le a \le n-1$ zukommen kann und für die gilt:

(i) $E(a)$ ist algorithmisch mit geringem Aufwand zu überprüfen

(ii) falls *n* Primzahl ist, dann trifft $E(a)$ für alle Zahlen *a* mit $1 \le a \le n-1$ zu

(iii) falls *n* keine Primzahl ist, dann trifft $E(a)$ für weniger als die Hälfte aller Zahlen *a* mit $1 \le a \le n-1$ zu.

Falls $E(a)$ gilt, dann heißt a **Zeuge (witness) für die Primzahleigenschaft von** n. Dann läßt sich folgender randomisierte Primzahltest definieren:

Eingabe: $n \in \mathbf{N}$, n ist ungerade, $m \in \mathbf{N}$

Verfahren: Aufruf der Funktion `random_is_prime (n : INTEGER; m : INTEGER): BOOLEAN;`

Ausgabe: n wird als Primzahl angesehen, wenn `random_is_prime (n, m) = TRUE` ist, ansonsten wird n nicht als Primzahl angesehen.

```
FUNCTION random_is_prime (n : INTEGER;
                          m : INTEGER): BOOLEAN;

   VAR idx      : INTEGER;
       a        : INTEGER;
       is_prime : BOOLEAN;


   BEGIN { random_is_prime }
     is_prime := TRUE;

     FOR idx := 1 TO m DO
       BEGIN
         wähle eine Zufallszahl a zwischen 1 und n-1;
         IF ( E(a) trifft nicht zu)
         THEN BEGIN
                is_prime := FALSE;
                Break;
              END;
       END;

     random_is_prim := is_prime;

   END   { random_is_prime };
```

Der Algorithmus versucht also, m Zeugen für die Primzahleigenschaft von n zu finden. Wird dabei zufällig eine Zahl a mit $1 \le a \le n-1$ erzeugt, für die $E(a)$ nicht zutrifft, dann wird wegen (ii) die korrekte Antwort gegeben. Ist n Primzahl, dann gibt der Algorithmus ebenfalls wegen (ii) die korrekte Antwort. Wurden m Zeugen für die Primzahleigenschaft von n festge-

stellt, kann es trotzdem sein, daß *n* keine Primzahl ist, obwohl der Algorithmus angibt, *n* sei Primzahl. Die Wahrscheinlichkeit, bei einer Zahl *n*, die nicht Primzahl ist, *m* Zeugen zu finden, ist wegen (iii) kleiner als $(1/2)^m$, d.h. die Wahrscheinlichkeit einer fehlerhaften Entscheidung ist in diesem Fall kleiner als $(1/2)^m$. Insgesamt ist die Wahrscheinlichkeit einer korrekten Antwort des Algorithmus größer als $1-(1/2)^m$. Da wegen (i) die Eigenschaft $E(a)$ leicht zu überprüfen ist, ist hiermit ein effizientes Verfahren beschrieben, das mit beliebig hoher Wahrscheinlichkeit eine korrekte Antwort liefert. Diese ist beispielsweise für $m = 20$ größer als 0,999999.

Die Frage stellt sich, ob es geeignete Zeugeneigenschaften $E(a)$ gibt. Einen ersten Versuch legt der Satz von Fermat nahe:

Satz 7.1-2:

Ist $n \in \mathbf{N}$ eine Primzahl und $a \in \mathbf{N}$ eine Zahl mit $ggT(a,n) = 1$, dann ist $a^{n-1} \equiv 1 \pmod{n}$.

Definiert man $E(a) =$ „$ggT(a,n) = 1$ und $a^{n-1} \equiv 1 \pmod{n}$“, dann sieht man, daß (i) und (ii) gelten. Leider gilt (iii) für diese Eigenschaft $E(a)$ nicht. Es gibt nämlich unendlich viele Zahlen *n*, die **Carmichael-Zahlen**, die folgende Eigenschaft besitzen: *n* ist *keine* Primzahl, und es ist $a^{n-1} \equiv 1 \pmod{n}$ für alle *a* mit $ggT(a,n) = 1$. Ist *n* also eine Carmichael-Zahl und *a* eine Zahl mit $1 \le a \le n-1$ und $ggT(a,n) = 1$, dann ist $a^{n-1} \equiv 1 \pmod{n}$. In diesem Fall gilt also für alle Zahlen *a* mit $1 \le a \le n-1$ und $ggT(a,n) = 1$ die Eigenschaft $E(a)$, und das sind mehr als die Hälfte aller Zahlen *a* mit $1 \le a \le n-1$.

Ein zweiter Versuch zur geeigneten Definition einer Zeugeneigenschaften $E(a)$ erweitert den ersten Versuch und wird durch die folgenden beiden Sätze begründet:

Satz 7.1-3:

Es sei *n* eine Primzahl. Dann gilt $x^2 \equiv 1 \pmod{n}$ genau dann, wenn $x \equiv 1 \pmod{n}$ oder $x \equiv -1 \equiv n-1 \pmod{n}$ ist.

Es sei *n* eine Primzahl mit $n > 2$. Dann ist *n* ungerade, d.h. $n = 1 + 2^j \cdot r$ mit ungeradem *r* und $j > 0$ bzw. $n-1 = 2^j \cdot r$. Ist *a* eine Zahl mit $1 \le a \le n-1$, dann ist $ggT(a,n) = 1$ und folglich $a^{n-1} \equiv 1 \pmod{n}$. Wegen $a^{n-1} = a^{(2^{j-1} \cdot r)2} = \left(a^{2^{j-1} \cdot r}\right)^2 \equiv 1 \pmod{n}$ folgt mit Satz 7.1-3 (dort wird $x = a^{2^{j-1} \cdot r}$ gesetzt): $a^{2^{j-1} \cdot r} \equiv 1 \pmod{n}$ oder $a^{2^{j-1} \cdot r} \equiv -1 \pmod{n}$. Ist hierbei $j - 1 > 0$ und

$a^{2^{j-1}\cdot r} \equiv 1 (\mathrm{mod}\, n)$, dann kann man den Vorgang des Wurzelziehens wiederholen: In Satz 7.1-3 wird $x = a^{2^{j-2}\cdot r}$ gesetzt usw. Der Vorgang ist spätestens dann beendet, wenn $a^{2^{j-j}\cdot r} = a^r$ erreicht ist. Es gilt daher der folgende Satz:

Satz 7.1-4:

Es sei *n* eine ungerade Primzahl, $n = 1 + 2^j \cdot r$ mit ungeradem *r* und $j > 0$. Dann gilt für jedes *a* mit $1 \le a \le n-1$:

Die Folge $\left(a^r, a^{2r}, a^{4r}, ..., a^{2^{j-1}\cdot r}, a^{2^j \cdot r}\right)$ der Länge $j+1$, wobei alle Werte modulo *n* reduziert werden, hat eine der Formen

$(1, 1, 1, ..., 1, 1)$ oder

$(*, *, ..., *, -1, 1, ..., 1, 1)$.

Hierbei steht das Zeichen „*“ für eine Zahl, die verschieden von 1 oder -1 ist.

Wenn die in Satz 7.1-4 beschriebene Folge eine der drei Formen

$(*, *, ..., *, 1, 1, ..., 1, 1)$,

$(*, *, ..., *, -1)$ oder

$(*, *, ..., *, *, ..., *)$

aufweist, dann ist *n* mit Sicherheit keine Primzahl. Andererseits ist es nicht ausgeschlossen, daß für eine ungerade zusammengesetzte Zahl *n* und eine Zahl *a* mit $1 \le a \le n-1$ die Folge $\left(a^r, a^{2r}, a^{4r}, ..., a^{2^{j-1}\cdot r}, a^{2^j \cdot r}\right) \mathrm{mod}\, n$ eine der beiden Formen $(1, 1, 1, ..., 1, 1)$ oder $(*, *, ..., *, -1, 1, ..., 1, 1)$ hat. In diesem Fall heißt *n* **streng pseudoprim zu Basis** *a*. Es gilt folgender Satz (Beweis siehe Literatur):

Satz 7.1-5:

Es sei *n* eine ungerade zusammengesetzte Zahl. Dann ist *n* streng pseudoprim für höchtens ein Viertel aller Basen *a* mit $1 \le a \le n-1$.

Eine geeignete Zeugeneigenschaften $E(a)$ für die Funktion

```
random_is_prime (n : INTEGER;
                 m : INTEGER): BOOLEAN;
```

ist daher die folgende Bedingung:

$E(a) =$ „$ggT(a, n) = 1$ und

$a^{n-1} \equiv 1 (\mathrm{mod}\, n)$ und

die Folge $\left(a^r, a^{2r}, a^{4r}, ..., a^{2^{j-1}\cdot r}, a^{2^j \cdot r}\right) \mathrm{mod}\, n$ hat eine der Formen

$(1, 1, 1, ..., 1, 1)$ oder $(*, *, ..., *, -1, 1, ..., 1, 1)$“.

Die Wahrscheinlichkeit einer korrekten Antwort des Algorithmus mit dieser Zeugeneigenschaft ist wegen Satz 7.1-5 größer als $1-(1/4)^m$.

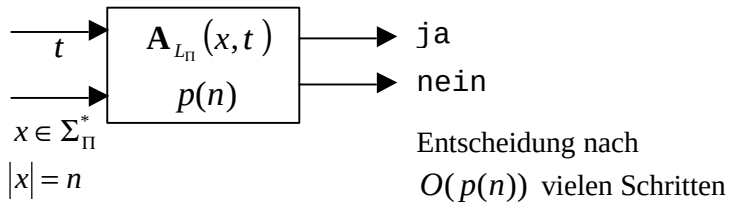
Es ist übrigens nicht notwendig, für eine große Anzahl von Zahlen a mit $1 \le a \le n-1$ die Zeugeneigenschaft zu überprüfen um sicher zu gehen, daß n eine Primzahl ist: Nur eine zusammengesetzte Zahl $n < 2{,}5 \cdot 10^{10}$, nämlich n = 3.215.031.751, ist streng pseudoprim zu den vier Basen a = 2, 3, 5 und 7. Für praktische Belange ist daher das Verfahren ein effizienter Primzahltest. Untersucht man große Zahlen, die spezielle Formen aufweisen, etwa Mersenne-Zahlen, auf Primzahleigenschaft bieten sich speziell angepaßte Testverfahren an. Schließlich gibt es eine Reihe von Testverfahren, die andere zahlentheoretische Eigenschaften nutzen.

7.2 Modelle randomisierter Algorithmen

In Zusammenführung der obigen Ansätze mit den Modellen aus der Theorie der Berechenbarkeit (Turingmaschinen, deterministische und nichtdeterministische Algorithmen) wurde eine Reihe weiterer Berechnungsmodelle entwickelt. Im folgenden werden wieder Entscheidungsprobleme betrachtet.

Ausgehend von der Klasse **P** der deterministisch polynomiell entscheidbaren Probleme erweitert man deren Algorithmen nicht um die Möglichkeit der Verwendung nichtdeterministisch erzeugter Zusatzinformationen (Beweise) wie beim Übergang zu den polynomiellen Verifizierern, sondern läßt zu, daß bei Verzweigungen während des Ablaufs der Algorithmen (Verzweigungen) ein Zufallsexperiment darüber entscheidet, welche Alternative für den weiteren Ablauf gewählt wird. Man gelangt so zu der Klasse **RP** (randomized polynomial time).

Es sei Π ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$. L_Π liegt in **RP** (bzw. „Π liegt in **RP**“), wenn es einen Algorithmus (**RP-Akzeptor**) $\mathbf{A}_{L_\Pi}$ gibt, der neben der Eingabe $x \in \Sigma_\Pi^*$ eine Folge $t \in \{0,1\}^*$ zufälliger Bits liest, wobei jedes Bit unabhängig von den vorher gelesenen Bits ist und $P(0) = P(1) = 1/2$ gilt. Nach polynomiell vielen Schritten (in Abgängigkeit von der Größe $|x|$ der Eingabe) kommt der Akzeptor auf die `ja`-Entscheidung bzw. auf die `nein`-Entscheidung. Eingaben für $\mathbf{A}_{L_\Pi}$ sind also die Wörter $x \in \Sigma_\Pi^*$ und eine Folge $t \in \{0,1\}^*$ zufälliger Bits:



Für jedes $x \in L_\Pi$ ist $P(\mathbf{A}_{L_\Pi}(x,t) = \texttt{ja}) \geq 1/2$,
für jedes $x \notin L_\Pi$ ist $P(\mathbf{A}_{L_\Pi}(x,t) = \texttt{nein}) = 1$

Man läßt also zur Akzeptanz von $x \in L_\Pi$ einen einseitigen Fehler zu. Jedoch muß bei $x \in L_\Pi$ der Algorithmus $\mathbf{A}_\Pi$ bei mindestens der Hälfte aller möglichen Zufallsfolgen τ auf die `ja`-Entscheidung kommen. Für die übrigen darf er auch auf die `nein`-Entscheidung führen. Für $x \notin L_\Pi$ muß er aber immer die `nein`-Entscheidung treffen. Der einseitige Fehler kann durch Einsatz geeigneter Replikations-Techniken beliebig klein gehalten werden.

Da dem Akzeptor nur polynomielle Zeit zur Verfügung steht, kann er auch nur polynomiell viele Bits der Zufallsfolge t lesen, so daß man annehmen kann, daß t aus polynomiell vielen Bits besteht.

Ein Beispiel für eine Sprache aus **RP** ist die Menge $L_2 = \{n \mid n \in \mathbf{N} \text{ und } n \text{ ist keine Primzahl}\}$ aus Kapitel 5.5. Dort wird $L_2 \in \mathbf{NP}$ gezeigt. Einen **RP**-Akzeptor $\mathbf{A}_{L_2}$ für L_2 erhält man aus der Funktion `random_is_prim` aus Kapitel 7.1. $\mathbf{A}_{L_2}$ liest zwei Eingaben, nämlich eine Zahl $n \in \mathbf{N}$ und eine Folge $t \in \{0,1\}^*$ zufälliger Bits der Länge $k = size(n)$. Diese Zufallsfolge wird als Binärkodierung einer Zufallszahl a mit $1 \leq a \leq n-1$ interpretiert (die Fälle $a = 0$ und $a = n$ sollen zur vereinfachten Darstellung hier ausgeschlossen sein). Dazu wird angenommen, daß es eine Funktion `make_INTEGER(`t `,` $size(n)$ `)` gibt, die die Zufallsfolge t in eine natürliche Zahl a mit $1 \leq a \leq n-1$ umwandelt. Dann wird die Bedingung $E(a)$ (siehe Kapitel 7.1) überprüft und eine Entscheidung getroffen:

```
FUNCTION A_L2 (n : INTEGER;
               t : ...      ): ...;

   VAR a : INTEGER;

   BEGIN { A_L2 }
     a := make_INTEGER(t , size(n));
     IF ( E(a) trifft nicht zu)
     THEN A_L2 := ja
     ELSE A_L2 := nein;
END   { A_L2 };
```

Ist $n \in L_2$, d.h. n ist keine Primzahl, dann trifft nach Definition $E(a)$ für weniger als die Hälfte aller Zahlen a mit $1 \le a \le n-1$ zu. Daher trifft $E(a)$ für mehr als die Hälfte aller Zahlen a mit $1 \le a \le n-1$ nicht zu, und es gilt $P(\mathbf{A}_{L_2}(n,t) = \texttt{ja}) > 1/2$.

Ist $n \notin L_2$, d.h. n ist eine Primzahl, dann trifft nach Definition $E(a)$ für alle Zahlen a mit $1 \le a \le n-1$ zu. Daher antwortet $\mathbf{A}_{L_2}$ mit $\mathbf{A}_{L_2}$:= nein.

Satz 7.2-1:

Es gilt $\mathbf{P} \subseteq \mathbf{RP}$ und $\mathbf{RP} \subseteq \mathbf{NP}$.

Ob diese Inklusionen echt sind, ist nicht bekannt, vieles spricht jedoch dafür.

Beweis:

Es sei $L_\Pi \in \mathbf{P}$, $L_\Pi \subseteq \Sigma_\Pi^*$. Dann gibt es einen deterministischen polynomiell zeitbeschränkten Entscheidungsalgorithmus für L_Π. Dieser ist kann als **RP**-Akzeptor betrachtet werden, der ohne Zufallsfolge auskommt. Daher gilt $L_\Pi \in \mathbf{RP}$.

Es sei $L_\Pi \in \mathbf{RP}$, $L_\Pi \subseteq \Sigma_\Pi^*$. Dann gibt es einen **RP**-Akzeptor $\mathbf{A}_{L_\Pi}$ für L_Π. Zu zeigen ist, daß es auch einen Verifizierer, wie er für die Klasse **NP** erforderlich ist, für L_Π gibt („**NP**-Verifizierer"). Der **RP**-Akzeptor $\mathbf{A}_{L_\Pi}$ kann als Verifizierer angesehen werden. Bei Eingabe von $x \in \Sigma_\Pi^*$ bekommt dieser als Beweis B_x eine Zufallsfolge t. Ist $x \in L_\Pi$, dann gibt es eine Zufallsfolge t mit $\mathbf{A}_{L_\Pi}(x,t) = \texttt{ja}$ (da $x \in L_\Pi$ und in diesem Fall $P(\mathbf{A}_{L_\Pi}(x,t) = \texttt{ja}) \ge 1/2$ gelten, führen mindestens die Hälfte aller Zufallsfolgen auf die ja-Entscheidung). Ist $x \notin L_\Pi$, dann führen wegen $P(\mathbf{A}_{L_\Pi}(x,t) = \texttt{nein}) = 1$ alle Zufallsfolgen auf die nein-Entscheidung. ///

Die Zusammenführung der Konzepte des Zufalls und des Nichtdeterminismus bei polynomiell zeitbeschränktem Laufzeitverhalten führt auf die Klasse **PCP** (probabilistically checkable proofs). Hierbei werden Verifizierer, wie sie bei der Definition der Klasse **NP** eingeführt wurden, um die Möglichkeit erweitert, Zufallsexperimente auszuführen:

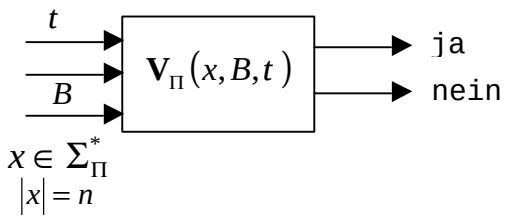
Es seien $r : \mathbf{N} \to \mathbf{N}$ und $q : \mathbf{N} \to \mathbf{N}$ Funktionen. Ein Verifizierer (nichtdeterministischer Algorithmus) $\mathbf{V}_\Pi$ heißt $(r(n), q(n))$-**beschränkter Verifizierer für das Entscheidungsproblem** Π über einem Alphabet Σ_Π, wenn er Zugriff auf eine Eingabe $x \in \Sigma_\Pi^*$ mit $|x| = n$, einen Beweis $B \in \Sigma_0^*$ und eine Zufallsfolge $t \in \{0,1\}^*$ hat und sich dabei folgendermaßen verhält:

1. $\mathbf{V}_\Pi$ liest zunächst die Eingabe $x \in \Sigma_\Pi^*$ (mit $|x| = n$) und $O(r(n))$ viele Bits aus der Zufallsfolge $t \in \{0,1\}^*$.
2. Aus diesen Informationen berechnet $\mathbf{V}_\Pi$ $O(q(n))$ viele Positionen der Zeichen von $B \in \Sigma_0^*$, die überhaupt gelesen (erfragt) werden sollen.
3. In Abhängigkeit von den gelesenen Zeichen in B (und der Eingabe x) kommt $\mathbf{V}_\Pi$ auf die `ja`- bzw- `nein`-Entscheidung.

Die Entscheidung, die $\mathbf{V}_\Pi$ bei Eingabe von $x \in \Sigma_\Pi^*$, $B \in \Sigma_0^*$ und $t \in \{0,1\}^*$ liefert, wird mit $\mathbf{V}_\Pi(x, B, t)$ bezeichnet.

Es sei Π ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$. $L_\Pi \in \mathbf{PCP}(r(n), q(n))$ („Π liegt in der Klasse $\mathbf{PCP}(r(n), q(n))$“), wenn es einen $(r(n), q(n))$-beschränkten Verifizierer $\mathbf{V}_\Pi$ gibt, der L_Π in folgender Weise akzeptiert:

Für jedes $x \in L_\Pi$ gibt es einen Beweis B_x mit $P(\mathbf{V}_\Pi(x, B_x, t) = \texttt{ja}) = 1$,
für jedes $x \notin L_\Pi$ und alle Beweise B gilt $P(\mathbf{V}_\Pi(x, B, t) = \texttt{nein}) \geq 1/2$.



Für Eingaben $x \in L_\Pi$ gibt es also einen Beweis, der immer akzeptiert wird. Der Versuch, eine Eingabe $x \notin L_\Pi$ zu akzeptieren, scheitert mindestens mit einer Wahrscheinlichkeit $\geq 1/2$.

Im folgenden bezeichnet $poly(n)$ die Klasse der Polynome. Dann gilt beispielsweise

$$\mathbf{P} = \bigcup_{k=0}^{\infty} TIME\left(n^k\right) = TIME\left(poly(n)\right) \text{ und } \mathbf{NP} = \bigcup_{k=0}^{\infty} NTIME\left(n^k\right) = NTIME\left(poly(n)\right).$$

Satz 7.2-2:

$$\mathbf{PCP}\left(r(n),q(n)\right) \subseteq NTIME\left(q(n) \cdot 2^{O(r(n))}\right)$$

Beweis:

Es sei $L_\Pi \in \mathbf{PCP}\left(r(n),q(n)\right)$. Dann gibt es einen $\left(r(n),q(n)\right)$-beschränkten Verifizierer $\mathbf{V}_\Pi$ für L_Π. Aus diesem kann man auf folgende Weise einen Verifizierer $\tilde{\mathbf{V}}_\Pi$ (im Sinne der Definition einer nichtdeterministischen Turingmaschine) für L_Π konstruieren:

Es sei $x \in \Sigma_\Pi^*$ mit $|x| = n$ und B_x ein Beweis. Für jede der $2^{O(r(n))}$ vielen Zufallsfolgen der Länge $O\left(r(n)\right)$ simuliert $\tilde{\mathbf{V}}_\Pi$ in jeweils polynomiell vielen Schritten die Berechnung des Verifizierers $\mathbf{V}_\Pi$ und akzeptiert *x* genau dann, wenn $\mathbf{V}_\Pi$ die Eingabe *x* für jede Zufallsfolge akzeptiert. $\tilde{\mathbf{V}}_\Pi$ ist daher eine nichtdeterministische Turingmaschine, die *x* genau dann akzeptiert, wenn $x \in L_\Pi$ gilt. ///

Setzt man im speziellen für $r(n)$ und $q(n)$ Polynome ein, so gilt sogar:

Satz 7.2-3:

$$\mathbf{PCP}\left(poly(n),poly(n)\right) = NTIME\left(2^{poly(n)}\right)$$

Weiter gelten folgende Aussagen:

Satz 7.2-4:

$$\mathbf{PCP}(0,0) = \mathbf{P},$$

$$\mathbf{PCP}\left(0,poly(n)\right) = \mathbf{NP},$$

$$\mathbf{PCP}\left(\log(n),poly(n)\right) = \mathbf{NP}$$

Aus $\mathbf{PCP}(r(n),q(n)) \subseteq NTIME\left(q(n)\cdot 2^{O(r(n))}\right)$ folgt unmittelbar:

Satz 7.2-5:
$\mathbf{PCP}(\log(n),1) \subseteq \mathbf{NP}$

Die Umkehrung dieser Aussage wird als das wichtigste Resultat der Theoretischen Informatik der letzten 10 Jahre angesehen (Arora, Lund, Motwani, Sudan, Szegedy, 1992). Es hat zu zahlreichen Konsequenzen für die Nicht-Approximierbarkeit von Optimierungsaufgaben aus **NPO** geführt.

Satz 7.2-6:
$\mathbf{PCP}(\log(n),1) = \mathbf{NP}$

„Wie man Beweise verifiziert, ohne sie zu lesen“

Das Ergebnis besagt, daß es zu jeder Menge L_Π für ein Entscheidungsproblem Π aus **NP** einen Verifizierer gibt, der bei jeder Eingabe nur konstant viele Stellen des Beweises liest, die er unter Zuhilfenahme von $O(\log(n))$ vielen zufälligen Bits auswählt, um mit hoher Wahrscheinlichkeit richtig zu entscheiden.

Ein Beispiel für die Anwendung von Satz 7.2-6 ist der Beweis des folgenden Satzes. In ihm wird gezeigt, daß das 3-SAT-Maximierungsproblem nicht in **PTAS** liegt (siehe Kapitel 6.2), d.h. kein polynomiell zeitbeschränktes Approximationsschema besitzt, außer $\mathbf{P} = \mathbf{NP}$.

3-SAT-Maximierungsproblem

Instanz: 5. $I = (K,V)$
$K = \{F_1, F_2, ..., F_m\}$ ist eine Menge von Klauseln, die aus Booleschen Variablen aus der Menge $V = \{x_1, ..., x_n\}$ gebildet werden und jeweils die Form $F_i = (y_{i_1} \vee y_{i_2} \vee y_{i_3})$ für $i = 1, ..., m$ besitzen. Dabei steht y_{i_j} für eine Boolesche Variable (d.h. $y_{i_j} = x_l$) oder für eine negierte Boolesche Variable (d.h. $y_{i_j} = \neg x_l$)
size(I) = Anzahl der Zeichen in I
6. SOL(I) = Belegung der Variablen in V mit Wahrheitswerten `TRUE` oder `FAL-`

SE, d.h. SOL(I) ist eine Abbildung $f : V \to \{\mathsf{TRUE}, \mathsf{FALSE}\}$

7. Für $f \in \mathrm{SOL}(I)$ ist $m(I, f) =$ Anzahl der Klauseln in K, die durch f erfüllt werden
8. *goal* = max

Es läßt sich ein 2-approximativer Algorithmus für das 3-SAT-Maximierungsproblem angeben (siehe Literatur), d.h. diese Problem liegt in **APX**.

Satz 7.2-7:
Das 3-SAT-Maximierungsproblem liegt nicht in **PTAS**, d.h. es besitzt kein polynomiell zeitbeschränktes Approximationsschema, außer $\mathbf{P} = \mathbf{NP}$.

Beweis:

Es sei L_{SAT} = {F | F ist ein erfüllbarer Boolescher Ausdruck }.
$L_{\mathrm{SAT}} \subseteq \Sigma^*_{\mathrm{BOOLE}}$ = {F | F ist ein Boolescher Ausdruck}, L_{SAT} ist **NP**-vollständig.

Es wird eine Abbildung f definiert, die jedem $F \in \Sigma^*_{\mathrm{BOOLE}}$ eine Instanz $f(F) = (K,V)$ für das 3-SAT-Maximierungsproblem zuordnet, wobei $f(F)$ in polynomieller Zeit aus F berechnet werden kann, und die folgende Eigenschaft besitzt:
Ist $F \in L_{\mathrm{SAT}}$, dann sind alle Klauseln in K erfüllbar.
Ist $F \notin L_{\mathrm{SAT}}$, dann gibt es eine Konstante $e > 0$, so daß mindestens ein Anteil der Größe e aller Klauseln in K nicht erfüllbar ist. Das bedeutet mit $c(F) = |K|$ für $f(F) = (K,V)$:

$m^*(f(F)) = \begin{cases} = c(F) & \text{für } F \in L_{SAT} \\ \leq c(F) \cdot (1-e) & \text{für } F \notin L_{SAT} \end{cases}$. Mit Satz 6.1-8 folgt, daß es keinen polynomiell zeitbeschränkten r-approximativen Algorithmus für das 3-SAT-Maximierungsproblem mit $r < 1/(1-e)$ gibt, außer $\mathbf{P} = \mathbf{NP}$. Daher besitzt das 3-SAT-Maximierungsproblem kein polynomiell zeitbeschränktes Approximationsschema.

Da L_{SAT} **NP**-vollständig ist, gibt es für L_{SAT} nach Satz 7.2-6 einen $(\log(n),1)$-beschränkten Verifizierer $\mathbf{V}_{SAT}$. In $\mathbf{V}_{SAT}$ werden eine Formel F mit $size(F) = n$, ein Beweis B und eine Zufallsfolge t eingegeben. Man kann annehmen, daß das Alphabet, mit dem Beweise formuliert werden, das Alphabet $\Sigma_0 = \{0,1\}$ ist, d.h. jeder Beweis B ist eine 0-1-Folge. Man kann weiterhin annehmen, daß $\mathbf{V}_{SAT}$ genau $q > 2$ Zeichen von B erfragt (auch wenn nicht alle Zeichen des Beweises zur Verifikation benötigt werden). Da höchstens $c \cdot \log(n)$ Bits (mit einer Konstanten c) aus t und dann q Positionen in B gelesen werden, brauchen nur Beweise betrachtet zu werden, die nicht länger als $q \cdot 2^{c \cdot \log(n)} = q \cdot n^c$ sind.

Es wird eine Menge V von Booleschen Variablen definiert: Für jede Position eines Beweises wird eine Boolesche Variable in V aufgenommen, d.h. $V = \{x_1, x_2, x_3, \ ..., \ x_k\}$ mit $k = q \cdot n^c$. V enthält polynomiell viele Variablen und ist in polynomieller Zeit aus F konstruierbar. Zwischen den Belegungen der Variablenmenge V und den Werten eines Beweises B besteht eine bijektive Abbildung g: Der Wert der j-ten Variablen x_j ist genau dann x_j = TRUE, wenn B an der j-ten Position den Wert $b_j = 1$ hat. Mit $g(x_j)$ werde der Wert an der j-ten Position in B bezeichnet; entsprechend sei $g^{-1}(b_j)$ der Wert der j-ten Variablen in V. Die Abbildung g werde auf Literate $\neg x_j$ durch $g(\neg x_j) = g(x_j)$ erweitert.

Für die Zufallsfolge t seien die q Positionen, die in einem Beweis erfragt werden, die Positionen t_1, ..., t_q. Mit einigen der möglichen Wert-Kombinationen an diesen Positionen kommt $\mathbf{V}_{SAT}$ bei Auswertung zur `ja`-Entscheidung, mit anderen kommt $\mathbf{V}_{SAT}$ zur `nein`-Entscheidung. Mit A_t wird die Menge derjenigen 0-1-Kombinationen bezeichnet, für die $\mathbf{V}_{SAT}$ zur `nein`-Entscheidung kommt. Offensichtlich ist $|A_t| \le |\Sigma_0|^q = 2^q$. Für jedes $(b_1, ..., b_q) \in A_t$ wird eine Formel $(y_{t_1} \vee ... \vee y_{t_q})$ gebildet mit $y_{t_i} = \begin{cases} x_{t_i} & \text{für } b_i = 0 \\ \neg x_{t_i} & \text{für } b_i = 1 \end{cases}$.

Es sei eine Belegung der Variablen in V gegeben. Dann gilt:

(*) Unter dieser Belegung hat die Formel $(y_{t_1} \vee ... \vee y_{t_q})$, die aus $(b_1, ..., b_q) \in A_t$ gebildet wurde, genau dann den Wert TRUE, wenn $(g(y_{t_1}), ..., g(y_{t_q})) \neq (b_1, ..., b_q)$ ist.

Denn $(y_{t_1} \vee ... \vee y_{t_q})$ hat genau dann den Wert TRUE, wenn mindestens eines der Literale den Wert TRUE besitzt, etwa y_{t_j} = TRUE. Im Fall $y_{t_j} = x_{t_j}$ bedeutet dieses (nach Definition von y_{t_j}) $b_j = 0$ und $g(y_{t_j}) = 1$, im Fall $y_{t_j} = \neg x_{t_j}$ ist x_{t_j} = FALSE, d.h. $g(y_{t_j}) = 0$, und $b_j = 1$.

Alle so entstehenden Formeln (für alle Zufallsfolgen t) bilden die Formelmenge K'. Diese enthält höchstens $2^{c \cdot \log(n)} \cdot 2^q = 2^q n^c$ viele Formel. Alle Formeln der Menge K' lassen sich so in Klauseln umformen, daß jede neue Klausel genau 3 Literale enthält. Gilt $q = 3$, haben alle Formeln $(y_{t_1} \vee ... \vee y_{t_q})$ bereits die gewünschte Form. Für $q > 3$ werden für jede Formel $G = (y_{t_1} \vee ... \vee y_{t_q})$ $q - 3$ neue Variablen $z_{G,1}, ..., z_{G,q-3}$ eingeführt und G durch $(y_{t_1} \vee y_{t_2} \vee z_{G,1})$, $(\neg z_{G,1} \vee y_{t_3} \vee z_{G,2})$, ..., $(\neg z_{G,q-4} \vee y_{t_{q-2}} \vee z_{G,q-3})$, $(\neg z_{G,q-3} \vee y_{t_{q-1}} \vee y_{t_q})$ ersetzt. G ist genau dann erfüllbar, wenn alle so entstandenen Formeln erfüllbar sind. Die Va-

riablenmenge *V* wird um die neu hinzugenommen Variablen erweitert; entsprechend wird unter einer Belegung $g(z_{G,j})=1$ genau dann gesetzt, wenn $z_{G,j}=\mathsf{TRUE}$ ist.

Diese eventuell erweiterte Klauselmenge bildet die Menge *K*. Es ist $|K|\le(q-2)\cdot|K'|$, und auch *V* behält eine polynomielle Größe.

Die Berechnung von $f(F)=(K,V)$ erfolgt in polynomieller Zeit.

Ist $F\in L_{\mathrm{SAT}}$, dann gibt es einen Beweis B_F, so daß der Verifizierer $\mathbf{V}_{SAT}$ für jede Zufallsfolgen t die Entscheidung $\mathbf{V}_{SAT}(F,B_F,t)=\mathtt{ja}$ trifft, denn $P(\mathbf{V}_{SAT}(F,B_F,t)=\mathtt{ja})=1$. Ist für eine Zufallsfolgen t die definierte Menge $A_t=\varnothing$, so wurde keine Formel in K' bzw. *K* aufgenommen. Ist $A_t\neq\varnothing$ und sind t_1, ..., t_q die Positionen in B_F, die von $\mathbf{V}_{SAT}$ aufgrund der Zufallsfolge t erfragt werden, etwa mit den Werten $a_1,...,a_q$, dann wird $x_{t_i}=\begin{cases}\mathsf{TRUE} & \text{für } a_i=1\\ \mathsf{FALSE} & \text{für } a_i=0\end{cases}$ gesetzt. Es sei $(b_1,...,b_q)\in A_t$ mit der dazu gebildeten Formel $(y_{t_1}\vee...\vee y_{t_q})\in K'$. Dann ist $(a_1,...,a_q)\neq(b_1,...,b_q)$, denn $(a_1,...,a_q)$ führt auf eine `ja`-Entscheidung, und alle $(b_1,...,b_q)\in A_t$ führen auf eine `nein`-Entscheidung. An mindestens einer Position, etwa an der Position *j* ist $a_j\neq b_j$. Ist $b_j=0$, dann ist $a_j=1$. Das Literal $y_{t_j}=x_{t_j}$ wurde auf `TRUE` gesetzt. Ist $b_j=1$, dann ist $a_j=0$. Das Literal $y_{t_j}=\neg x_{t_j}$ wurde auf `TRUE` gesetzt. In beiden Fällen ist $(y_{t_1}\vee...\vee y_{t_q})$ erfüllt. Das zeigt, daß alle Klauseln in K' und damit alle Klauseln in *K* erfüllbar sind und damit $m^*(f(F))=|K|=c(F)$ gilt.

Ist $F\notin L_{\mathrm{SAT}}$, dann gilt für jeden Beweis *B*, daß der Verifizierer $\mathbf{V}_{SAT}$ für mindestens die Hälfte aller $2^{c\cdot\log(n)}=n^c$ Zufallsfolgen die Entscheidung $\mathbf{V}_{SAT}(F,B,t)=\mathtt{nein}$ trifft, denn für jeden Beweis *B* ist in diesem Fall $P(\mathbf{V}_{SAT}(F,B,t)=\mathtt{nein})\ge 1/2$. Es sei eine Belegung von *V* gegeben, der mittels *g* zugehörige Beweis sei *B*. t sei eine der $2^{c\cdot\log(n)}/2$ Zufallsfolgen, die auf die `nein`-Entscheidung führen. Die Positionen in *B*, die von $\mathbf{V}_{SAT}$ aufgrund der Zufallsfolge t erfragt werden, seien t_1, ..., t_q, die Werte an diesen Positionen in *B* seien $a_1,...,a_q$. Dann ist nach Definition $(a_1,...,a_q)\in A_t$. Die aus $(a_1,...,a_q)$ gebildete Formel $(y_{t_1}\vee...\vee y_{t_q})$ hat wegen (*) den Wert `FALSE`. Für $q>3$ wurde durch obige Konstruktion die Formel $(y_{t_1}\vee...\vee y_{t_q})$ durch $q-3$ Formeln ersetzt. Es läßt sich zeigen, daß im vorliegenden Fall mindestens eine dieser Formeln den Wert `FALSE` trägt, unabhängig davon, wie die dort enthaltenen neuen Variablen $z_{G,j}$ belegt sind. Zu jeder der $2^{c\cdot\log(n)}/2$ Zufallsfolgen, die auf die `nein`-Entscheidung führen, gibt es also mindestens eine Formel in *K*, die nicht erfüllt ist. Daher beträgt der Anteil nichterfüllbarer Klauseln mindestens

$2^{c\cdot\log(n)} / \left(2\cdot|K|\right) \geq 2^{c\cdot\log(n)} / \left(2\cdot(q-2)\cdot 2^{q+c\cdot\log(n)}\right) = 1/\left((q-2)\cdot 2^{q+1}\right).$

Mit $e = 1/\left((q-2)\cdot 2^{q+1}\right)$ folgt die Behauptung. ///

8 Übungsaufgaben

Im folgenden werden Übungsaufgaben zu den einzelnen Kapiteln bereitsgestellt. Einige der Übungsaufgaben werden in der Veranstaltung behandelt. Aufgaben, die eventuell einiges Nachdenken und Nachschlagen in der Fachliteratur erfordern, sind mit dem Zeichen ¤ gekennzeichnet.

8.1 Übungsaufgaben zu Kapitel 1

Aufgabe 1.1.1: Es sei $\Sigma = \{a, b, c\}$. Bestimmen Sie Σ^3 und Σ^*.

Aufgabe 1.1.2: Es sei a ein Buchstabe eines endlichen Alphabets. Bestimmen Sie $\{a^2\}^+$ und $\{a^7\}^* \cdot \{a^3\}$.

Aufgabe 1.1.3: Es seien A und B Mengen. Zeigen Sie: $(A^*)^* = A^*$, $(A \cup B)^* = (A^* \cdot B^*)^*$ und $\varnothing^* = \{e\}$.

Aufgabe 1.1.4: Können L^* bzw. L^+ jemals die leere Menge sein? Unter welchen Umständen sind L^* bzw. L^+ endliche Mengen?

Aufgabe 1.1.5: Es seien A und B Mengen. Beweisen oder widerlegen Sie durch ein entsprechendes Gegenbeispiel die Behauptungen
$(A \cup B)^* = A^* \cup B^*$, $(A \cdot B \cup A)^* \cdot A = A \cdot (B \cdot A \cup A)^*$

Aufgabe 1.1.6: ¤ Für ein endliches Alphabet $\Sigma = \{a_1, ..., a_n\}$ kann man die Wörter aus Σ^* in lexikographischer Reihenfolge durchnumerieren (siehe Kapitel 1.1). Welche Nummer enthält in dieser Numerierung das Wort $a_{i_1} ... a_{i_k}$ der Länge k? Überlegen Sie sich die Lösung der Aufgabe zunächst für eine zweielementige Menge $\Sigma = \{0, 1\}$.

Aufgabe 1.1.7: ¤ Zeigen Sie, daß die Menge der rationalen Zahlen gleichmächtig mit der Menge der natürlichen Zahlen ist. Zur Erinnerung: Man kann die rationalen Zahlen durch $\mathbf{Q} = \left\{ r/t \mid r \in \mathbf{Z} \text{ und } t \in \mathbf{N}_{>0} \right\} = \left\{ (r,t) \mid r \in \mathbf{Z} \text{ und } t \in \mathbf{N}_{>0} \right\}$ definieren.

Aufgabe 1.1.8: Zeigen Sie: Die Mächtigkeit einer unendlich abzählbaren Menge ändert sich nicht, wenn man endlich viele Elemente entfernt.

Aufgabe 1.1.9: ¤ Zeigen Sie: Die Vereinigung abzählbar vieler abzählbarer Mengen ist wieder abzählbar.

Aufgabe 1.1.10: Es seien $f : \mathbf{N} \to \mathbf{R}$ und $g : \mathbf{N} \to \mathbf{R}$ zwei Funktionen. Zeigen Sie:
$O(f(n)) \cdot O(g(n)) \subseteq O(f(n) \cdot g(n))$,
$O(f(n) \cdot g(n)) \subseteq |f(n)| \cdot O(g(n))$ und
$O(f(n)) + O(g(n)) \subseteq O(|f(n)| + |g(n)|)$.

8.2 Übungsaufgaben zu Kapitel 2

Aufgabe 2.1.1: Geben Sie eine Turingmaschine mit 2 Bändern und dem Eingabealphabet $I = \{0,1\}$ durch ihre Überführungsfunktion an, die zunächst das (neue) Zeichen # auf das 2. Band schreibt und dann das Eingabewort vom 1. Band auf das 2. Band kopiert.

Aufgabe 2.1.2: Geben Sie eine Turingmaschine mit dem Eingabealphabet $I = \{0,1\}$ durch ihre Überführungsfunktion an, die das Eingabewort auf dem 1. Band dupliziert, d.h. steht das Wort $w = a_1 \ldots a_n$ im Startzustand auf dem Eingabeband, so steht dort abschließend das Wort $ww = a_1 \ldots a_n a_1 \ldots a_n$.

Aufgabe 2.1.3: ¤ Geben Sie eine Turingmaschine *TM* mit $L(TM) = \{a^n b^n c^n \mid n \in \mathbf{N}\}$ an. Welche Speicherplatzkomplexität hat Ihre Turingmaschine?

Aufgabe 2.1.4: Beschreiben Sie die Arbeitsweise einer Turingmaschine, die die Funktion $f : \begin{cases} \mathbf{N} \times \mathbf{N} & \to & \mathbf{N} \\ (n,m) & \to & n+m \end{cases}$ berechnet. Die Eingabe dieser Turingmaschine auf dem 1. Band

habe die Form $bin(n)\#bin(m)$, wobei # ein von 0 und 1 verschiedenes Zeichen ist. Am Ende der Berechnung soll auf dem Ausgabeband $bin(n+m)\#$ stehen.

Aufgabe 2.2.1: Zeigen Sie, daß das in der Vorlesung behandelte RAM-Programm **P** zur Akzeptanz von
$L(P) = \{ w \mid w \in \{1,2\}^* \text{ und die Anzahl der 1'en ist gleich der Anzahl der 2'en} \}$ folgende Zeit- und Platzkomplexität hat:

	uniformes Kostenkriterium	logarithmisches Kostenkriterium
Zeitkomplexität	$O(n)$	$O(n \cdot \log(n))$
Platzkomplexität	$O(1)$	$O(\log(n))$

Aufgabe 2.2.2: ¤ Entwerfen Sie ein RAM-Programm mit einer Zeitkomplexität der Ordnung $O(\log(n))$ unter dem uniformen Kostenkriterium zur Berechnung von n^n.

Aufgabe 2.6.1: In der Vorlesung wird eine 3-NDTM zur Lösung des Partitionenproblems mit ganzzahligen Eingabewerten angegeben. Das Partitionenproblem ist folgendes Entscheidungsproblem:

Instanz: $I = \{a_1, ..., a_n\}$
I ist eine Menge von *n* natürlichen Zahlen.

Lösung: Entscheidung „ja“, falls es eine Teilmenge $J \subseteq \{1, ..., n\}$ mit $\sum_{i \in J} a_i = \sum_{i \notin J} a_i$ gibt (d.h. wenn sich die Eingabeinstanz in zwei Teile zerlegen läßt, deren jeweilige Summen gleich groß sind).
Entscheidung „nein“ sonst.

Lösung: Entscheidung „ja“, falls es eine Teilmenge $J \subseteq \{1, ..., n\}$ mit $\sum_{i \in J} a_i = \sum_{i \notin J} a_i$ gibt (d.h. wenn sich die Eingabeinstanz in zwei Teile zerlegen läßt, deren jeweilige Summen gleich groß sind).
Entscheidung „nein“ sonst.

Für eine Instanz $I = \{a_1, ..., a_n\}$ sei $B = \sum_{i=1}^{n} a_i$. Falls B ungerade ist, lautet die Entscheidung für I „nein“, also kann B im folgenden als gerade angenommen werden. Es wird ein Boolescher Ausdruck $T[i, j]$ definiert durch
$T[i, j] = \mathsf{TRUE}$ genau dann, wenn es eine Teilmenge von $\{a_1, ..., a_i\}$ gibt, deren Elemente sich auf genau j aufsummieren.

(a) Welche Wahrheitswerte haben $T[1,0]$, $T[1, a_1]$ und $T[1, j]$ für $j \neq 0$ und $j \neq a_1$?
(b) Welcher Zusammenhang besteht zwischen der ja/nein-Entscheidung für eine Instanz $I = \{a_1, ..., a_n\}$ und dem Wert $T[n, B/2]$?
(c) Zeigen Sie: $T[i, j] = \mathsf{TRUE}$ genau dann, wenn entweder $T[i-1, j] = \mathsf{TRUE}$ oder wenn $a_i \leq j$ und $T[i-1, j-a_i] = \mathsf{TRUE}$ ist.
(d) Es sei I = {1, 9, 5, 3, 8 }. Berechnen Sie $T[i, j]$ für $i = 1, ..., n$ und $j = 0, ..., B/2$.
(e) Entwickeln Sie einen deterministischen Algorithmus, der bei Eingabe einer Instanz $I = \{a_1, ..., a_n\}$ für das Partitionenproblem eine ja/nein-Entscheidung trifft und dabei eine Anzahl von Anweisungen der Größenordnung $O(n \cdot B)$ ausführt. Warum handelt es sich hierbei *nicht* um einen Algorithmus, dessen Laufzeit polynomiell in der Größe der Eingabe ist?

Aufgabe 2.6.2: Zeigen Sie, daß die folgenden Funktionen $S_i : \mathbf{N} \to \mathbf{N}$ platzkonstruierbar sind, indem Sie deterministische Turingmaschinen angeben, die bei Eingabe eines Wortes der Länge n ein spezielles Symbol in die $S_i(n)$-te Zelle eines ihrer Bänder schreibt, ohne jeweils mehr als $S_i(n)$ viele Zellen auf allen Bändern zu verwenden.

(a) $S_1(n) = n^2$
(b) $S_2(n) = n^3 - n^2 + 1$
(c) $S_3(n) = 2^n$
(d) $S_4(n) = n!$

Aufgabe 2.6.3: ¤ Zeigen Sie:
Wird L von einer k-NDTM $TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$ mit Zeitkomplexität $T(n)$ akzeptiert, dann wird L von einer 1-NDTM TM' mit Zeitkomplexität der Ordnung $O(T^2(n))$ akzeptiert. Wie läßt sich das Ergebnis auf deterministische Turingmaschinen übertragen?

8.3 Übungsaufgaben zu Kapitel 3

Aufgabe 3.1.2: Zeigen Sie, daß die folgenden Mengen entscheidbar sind. Dabei werden natürliche Zahlen in die entsprechenden Turimgmaschinen in Binärkodierung eingegeben.

(a) $\{bin(n^2) \mid n \in \mathbf{N}\}$

(b) $\{bin(2^n) \mid n \in \mathbf{N}\}$

(c) $\{bin(2n) \mid n \in \mathbf{N}, n \geq 1 \text{ und } 2n = p + q \text{ mit Primzahlen } p \text{ und } q\}$; ist die Menge $\{bin(2n) \mid n \in \mathbf{N}, n \geq 1 \text{ und } 2n = p - q \text{ mit Primzahlen } p \text{ und } q\}$ entscheidbar?

Aufgabe 3.1.3: Zeigen Sie:

(a) Die Menge

$$L_{uni} = \left\{ z \;\middle|\; \begin{array}{l} z = u\#w \text{ mit } u \in \{0,1\}^*, w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE} \text{ und } u \in L(K_w) \end{array} \right\} \subseteq \{0,1,\#\}^*$$

ist rekursiv aufzählbar.

(b) Die Menge $\{0,1\}^* \setminus L_d$ ist nicht entscheidbar; hierbei ist L_d die in der Vorlesung definierte Menge

$$L_d = \{w \mid w \in \{0,1\}^*, VERIFIZIERE\_TM(w) = \mathsf{FALSE} \text{ oder } w_i \notin L(K_{w_i})\}.$$

(c) Die Menge L_{uni} aus Aufgabe (a) ist nicht entscheidbar.

Aufgabe 3.1.4: ¤ Zeigen Sie:

Die Menge

$$L_{ne} = \{w \mid w \in \{0,1\}^*, VERIFIZIERE\_TM(w) = \mathsf{TRUE} \text{ und } L(K_w) \text{ ist nicht entscheidbar}\}$$

ist nicht rekursiv aufzählbar. Gehen Sie dabei vor wie im Beweis zu $L_e = \{w \mid L(K_w) \text{ ist entscheidbar}\}$. Zu gegebenen $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$ mit $VERIFIZIERE\_TM(w) = \mathsf{TRUE}$ entwerfen Sie eine Turingmaschine $TM_{u,w}$ mit

$$L(TM_{u,w}) = \begin{cases} \{0,1\}^*\#\{0,1\}^* & \text{falls } u \in L(K_w) \\ L_{uni} & \text{falls } u \notin L(K_w) \end{cases}.$$

Aufgabe 3.1.5: Es sei Σ ein endliches Alphabet und $L \subseteq \Sigma^*$ rekursiv aufzählbar. Zeigen Sie, daß dann $L \leq L_H$ gilt. Wie kann man diese Aussage interpretieren?

Aufgabe 3.1.6: Es sei Σ ein endliches Alphabet und $L \subseteq \Sigma^*$ entscheidbar. Für die Menge $M \subseteq \Sigma^*$ gelte $M \neq \varnothing$ und $M \neq \Sigma^*$. Zeigen Sie, daß dann $L \leq M$ gilt. Wie kann man diese Aussage interpretieren?

Aufgabe 3.1.7: Formalisieren Sie die folgenden umgangssprachlichen Feststellungen und beweisen Sie die entsprechenden Aussagen:

(a) Es ist nicht entscheidbar, ob ein beliebiges Programm (in einer realen Programmiersprache) für eine Eingaben in eine unendliche Schleife läuft.

(b) Es ist nicht entscheidbar, ob ein beliebiges Programm (in einer realen Programmiersprache) überhaupt eine Ausgabe erzeugt.

8.4 Übungsaufgaben zu Kapitel 4

Aufgabe 4.1.1: Geben Sie eine Grammatik zur Erzeugung der Sprache $\left(0^*1^*\right)^*$ an.

Aufgabe 4.3.1: Zeigen Sie, daß die Sprache $L = \left\{0^{2^i} \mid i \geq 0\right\}$ kontextsensitiv ist.

Aufgabe 4.4.1: Zeigen Sie, daß die Sprache $L = \left\{0^{2^i} \mid i \geq 0\right\}$ nicht kontextfrei ist.

Aufgabe 4.4.2: Es seien *a*, *b* und *c* paarweise verschiedene Symbole. Welche der folgenden Sprachen sind kontextfrei? Begründen Sie Ihre Antwort.

(a) $\left\{a^k b^k c^i \mid k \in \mathbf{N}, i \in \mathbf{N}\right\}$

(b) $\left\{a^i b^j c^k \mid i \in \mathbf{N},\ j \in \mathbf{N}, k \in \mathbf{N},\ i > j \text{ oder } j > k\right\}$

(c) $\left\{a^i b^j a^i b^j \mid i \in \mathbf{N},\ j \in \mathbf{N}\right\}$

(d) $\left\{ww \mid w \in \left\{a\right\}^*\right\}$

(e) $\left\{ww \mid w \in \left\{a, b\right\}^*\right\}$

Aufgabe 4.4.3: Beschreiben Sie, wie ein nichtdeterministischer Kellerautomat mit zwei Kellern das Verhalten einer Turingmaschine simulieren kann.

Aufgabe 4.5.1: ¤ Beweisen Sie Satz 4.5-1.

Aufgabe 4.5.2¤ Zeigen Sie: Wird die Sprache *L* von einem nichtdeterministischen endlichen Automaten erkannt, dann gibt es einen deterministischen Automaten, der *L* erkennt.

Aufgabe 4.5.3: Es seien *a* und *b* verschiedene Buchstaben. Zeigen Sie, daß die folgenden Sprachen regulär sind:

(a) $\{ w \mid w \in \{a, b\}^*, \text{die Anzahl von } a\text{'s ist gerade, die Anzahl von } b\text{'s ist ungerade} \}$

(b) $\{ w \mid w \in \{a, b\}^*, w \text{ enthält höchstens ein Paar aufeinanderfolgende } a\text{'s oder } b\text{'s} \}$

(c) $\{ w \mid w \in \{a, b\}^*, w \text{ enthält nicht die Zeichenkette } aba \}$.

Aufgabe 4.5.4: Es seien *a* und *b* verschiedene Buchstaben. Zeigen Sie, daß die folgenden Sprachen nicht regulär sind:

(a) $\{ a^n b^n \mid n \in \mathbf{N} \}$

(b) $\{ a^n \mid n \in \mathbf{N}, n \text{ ist eine Quadratzahl} \}$

(c) $\{ a^n \mid n \in \mathbf{N}, n \text{ ist eine Primzahl} \}$.

Aufgabe 4.5.5: ¤ Es sei *L* eine kontextfreie Sprache und *R* eine reguläre Sprache. Zeigen Sie:

(a) $L \cap R$ ist kontextfrei.

(b) Es seien *a* und *b* verschiedene Buchstaben. Mit Hilfe des *uvwyx*-Theorems für kontextfreie Sprachen läßt sich zeigen, daß die Sprache $L_1 = \{ a^n b^m a^n b^m \mid n \in \mathbf{N}, m \in \mathbf{N} \}$ nicht kontextfrei ist. Warum ist auch $L_2 = \{ ww \mid w \in \{a, b\}^+ \}$ nicht kontextfrei?

(c) Es seien c_1, ..., c_n neue paarweise verschiedene Buchstaben. Warum ist $L_3 = \{ x_1 w x_2 w x_3 \mid w \in \{a, b\}^+, x_i \in \{c_1, ..., c_n\}^* \text{ für } i = 1, 2, 3 \}$ nicht kontextfrei?

(d) Es sei L_P die Menge aller korrekten Pascal-Programme (dasselbe kann man mit Java-, C++-, Cobolprogrammen usw. machen). Es wird eine Teilmenge L_4 korrekter Pascal-Programme definiert durch

$L_4 = \{ \textbf{PROGRAM A VAR } w : \textbf{INTEGER}; \textbf{BEGIN } w := 1; \textbf{END.} \mid w \in \{a, b\}^+ \}$.

Zusätzlich wird die reguläre Menge *R* definiert durch

$R = \{ \textbf{PROGRAM A VAR } w_1 : \textbf{INTEGER}; \textbf{BEGIN } w_2 := 1; \textbf{END.} \mid w_i \in \{a, b\}^+, i = 1, 2 \}$

Zeigen Sie mit Hilfe von L_4 und *R*, daß L_P nicht kontextfrei ist.

Aufgabe 4.6.1: Begründen Sie die Gültigkeit der in Kapitel 4.6 aufgeführten Eigenschaften der verschiedenen Sprachklassen.

8.5 Übungsaufgaben zu Kapitel 5

Aufgabe 5.3.1: Zeigen Sie:

(a) Die Klasse **NP** ist abgeschlossen bezüglich Schnitt und Vereinigung.

(b) Mit $L \in \mathbf{NP}$ gilt auch $L^* \in \mathbf{NP}$.

Aufgabe 5.3.2: ¤ Zeigen Sie, daß die Sprache

$L = \{(u, w, 0^t) \mid$ die nichtdeterministische Turingmaschine K_w akzeptiert u in höchstens t Schritten$\}$

NP-vollständig ist.

Aufgabe 5.3.3: Zeigen Sie: **P** = **NP** impliziert, daß $L = \{0,1\}$ **NP**-vollständig ist.

Aufgabe 5.4.1: ¤ Zeigen Sie (indem Sie beispielsweise die einschlägige Fachliteratur konsultieren) die Gültigkeit der nicht in der Vorlesung behandelten Reduktionen:

SAT $\leq_m^p$ 3-CSAT $\leq_m^p$ RUCKSACK $\leq_m^p$ PARTITION $\leq_m^p$ BIN PACKING und

3-CSAT $\leq_m^p$ KLIQUE und

3-CSAT $\leq_m^p$ GERICHTETER HAMILTONKREIS

$\leq_m^p$ UNGERICHTETER HAMILTONKREIS $\leq_m^p$ HANDUNGSREISENDER.

Literaturauswahl

Die mit einem Stern **(*)** gekennzeichneten Bücher werden zur vertiefenden und weiterführenden Lektüre besonders empfohlen.

Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.: **The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing**, Addison-Wesley, 1972.

Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.: **The Design and Analysis of Computer Algorithms**, Addison-Wesley, 1974.

(*) Asteroth, A; Baier, C.: **Theoretische Informatik**, Pearson Studium, 2002.

(*) Ausiello, G.; Crescenzi, P.; Gambosi, G.; Kann, V.; Marchetti-Spaccamela, A.; Protasi, M.: **Complexity and Approximation**, Springer, 1999.

Bartholomé, A.; Rung, J.; Kern, H.: **Zahlentheorie für Einsteiger**, Vieweg, 1995.

Blum, N.: **Theoretische Informatik**, Oldenbourg, 1998.

Bovet, D.P.; Crescenzi, P.: **Introduction to the Theory of Complexity**, Prentice Hall, 1994.

Dewdney, A.K.: **Der Turing Omnibus**, Springer, 1995.

Erk, K.; Priese, L.: **Theoretische Informatik**, 2. Aufl., Springer 2002.

(*) Garey, M.R.; Johnson, D.: **Computers and Intractability, A Guide to the Theory of NP-Completeness**, Freeman, 1979.

Harel, D.: **Algorithmics**, 2nd Ed., Addison Wesley, 1992.

Hopcroft, J.E.; Ullman, J.D.: **Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie**, Addison-Wesley, 1989.

Hromkovic, J.: **Algorithmische Konzepte der Informatik**, Teubner, 2001.

Paul, W.J.: **Einführung in die Komplexitätstheorie**, Teubner, 1990.

Salomaa, A.: **Public-Key Cryptography**, 2. Aufl., Springer, 1996.

Schöning, U.: **Theoretische Informatik kurzgefaßt**, 4. Aufl., Spektrum Akademischer Verlag, 2001.

(*) Vossen, G.; Witt, K.-U.: **Grundlagen der Theoretischen Informatik mit Anwendungen**, Vieweg, 2000.

Yan, S.Y.: **Number Theory for Computing**, Springer, 2000.