LEUPHANA
UNIVERSITÄT LÜNEBURG

Ausgewählte Kapitel der Theoretischen Informatik

Universität Lüneburg

Fakultät III

Prof. Dr. rer. nat. Ulrich Hoffmann

Überarbeitete Auflage, August 2007

Das vorliegende Skript dient als begleitende Unterlage für die Veranstaltungen *Theoretische Informatik, Algorithmen und Komplexität* und weiterer Veranstaltungen an der Universität Lüneburg. Die Durcharbeitung des Skripts ersetzt nicht den Besuch der Veranstaltungen, da dort zusätzlich Zusammenhänge, ergänzende Sachverhalte und im Skript nicht ausgeführte Inhalte behandelt werden.

Dieses Skript ist die überarbeitete Auflage der Veröffentlichung:
Ulrich Hoffmann: Ausgewählte Kapitel der Theoretischen Informatik, FINAL 13:1, 2003, ISSN 0939-8821.

Inhaltsverzeichnis

Literaturauswahl **4**
1 Einleitung **5**
1.1 Einige grundlegende Bezeichnungen und Sachverhalte 6
1.2 Problemklassen 27
1.3 Ein intuitiver Algorithmusbegriff 32
2 Modelle der Berechenbarkeit und Komplexität von Berechnungen **36**
2.1 Deterministische Turingmaschinen 36
2.2 Random Access Maschinen 57
2.3 Programmiersprachen 68
2.4 Universelle Turingmaschinen 79
2.5 Nichtdeterminismus 87
3 Grenzen der Berechenbarkeit **109**
3.1 Jenseits der Berechenbarkeit 109
3.2 Rekursiv aufzählbare und entscheidbare Mengen 114
4 Elemente der Theorie Formaler Sprachen und der Automatentheorie **133**
4.1 Grammatiken und formale Sprachen 133
4.2 Typ-0-Sprachen 136
4.3 Typ-1-Sprachen 138
4.4 Typ-2-Sprachen 145
4.5 Typ-3-Sprachen 161
4.6 Eigenschaften im tabellarischen Überblick 173
5 Praktische Berechenbarkeit **177**
5.1 Der Zusammenhang zwischen Optimierungs- und Entscheidungsproblemen 182
5.2 Komplexitätsklassen 189
5.3 Die Klassen P und NP 193
5.4 NP-Vollständigkeit 208
5.5 Beispiele für den Nachweis der NP-Vollständigkeit 220
5.6 Bemerkungen zur Struktur von NP 237
6 Approximation von Optimierungsaufgaben **247**
6.1 Absolut approximierbare Probleme 250
6.2 Relativ approximierbare Probleme 257
6.3 Polynomiell zeitbeschränkte und asymptotische Approximationsschemata 279
7 Weiterführende Konzepte **292**
7.1 Randomisierte Algorithmen 292
7.2 Modelle randomisierter Algorithmen 303

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.

Blum, N.: **Theoretische Informatik**, Oldenbourg, 1998.

Bovet, D.P.; Crescenzi, P.: **Introduction to the Theory of Complexity**, Prentice Hall, 1994.

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.

Hoffmann, U.: **Datenstrukturen und Algorithmen**, FINAL 15:2, 2005.

Hoffmann, U.: **Mathematik für Wirtschaftsinformtiker**, FINAL 15:1, 2005.

(*) Hopcroft, J.E.; Motwani, R.; Ullman, J.D.: **Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie**, 2. Aufl., Pearson Studium, 2002.

Hromkovič, J.: **Theoretische Informatik**, Teubner, 2004.

Hromkovič, J.: **Randomisierte Algorithmen**, Teubner, 2004.

Knuth, D.E.: **Arithmetik**, Springer, 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.: **Grundkurs Theoretische Informatik**, Vieweg, 2004.

Wanka, R.: **Approximationsalgorithmen**, Teubner, 2006.

Wegener, I.: **Komplexitätstheorie**, Springer, 2003.

Yan, S.Y.: **Number Theory for Computing**, Springer, 2000.

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.

Die Theoretische Informatik als eine der Säulen der Informatik kann natürlich nicht umfassend im Rahmen einer einsemestrigen Lehrveranstaltung dargestellt werden. Deshalb verzichtet der vorliegende Text auf den Anspruch einer umfassenden oder ansatzweisen Darstellung *aller* wichtigen Teilgebiete der Theoretischen Informatik. Er konzentriert sich vielmehr 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?
- Worin sind die Ursachen für die hohe Komplexität von Lösungen vieler praktischer Problemstellungen zu sehen?

Die Behandlung dieser Fragestellungen führt auf die Betrachtung von Modellen der Berechenbarkeit. Hierbei spielt die Turingmaschine nicht nur aus historischer Sicht eine herausragende Rolle. Ergänzend wird wegen seiner inhaltlichen Nähe zu realen Computern das Modell der Random Access Maschine behandelt. Auf die Darstellung von Theorien wie die μ-rekursiven Funktionen oder das λ-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 (der Begriff „schwer“ in diesem Zusammenhang wird im Text präzisiert). 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 und Sachverhalte

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 ist ein **Element** von A),
$\varnothing$ (**leere Menge**, d.h. die Menge, die kein Element enthält),
$A \subseteq B$ (A ist **Teilmenge** von B),
$A \subset B$ (A ist **echte Teilmenge** von B, d.h. $A \subseteq B$ und $A \neq B$),
$A \cup B$ (**Vereinigungsmenge** von A und B),
$A \cap B$ (**Schnittmenge** von A und B),
$A \setminus B$ (**Differenz** von A und B),
$\overline{A}^{B}$ (**Komplement** von A bezüglich B, d.h. $B \setminus A$),
$\mathbf{P}(A)$ (**Potenzmenge** von A, d.h. $\mathbf{P}(A) = \{ B \mid B \subseteq A \}$,

$$A_1 \times A_2 \times \ldots \times A_n = \{ (a_1, a_2, \ldots, a_n) \mid a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n \}$$

(**kartesisches Produkt** der Mengen A_1, A_2, ..., A_n, d.h.
$A_1 \times A_2 \times \ldots \times A_n = \{ (a_1, a_2, \ldots, a_n) \mid a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n \}$).

Zwei Mengen A und B sind gleich, geschrieben $A = B$, wenn $A \subseteq B$ und $B \subseteq A$ gilt.

Grundlegende wichtige Regeln der elementaren Mengenlehre werden in folgendem Satz (ohne Beweis) zusammengestellt.

Satz 1.1-1:

Es seien im folgenden A, B und C Mengen. Dann gilt:

(i) $A \cup \varnothing = A,\ A \cap \varnothing = \varnothing$.

(ii) $A \cap B \subseteq A,\ A \cap B \subseteq B,\ A \subseteq A \cup B,\ B \subseteq A \cup B$.

(iii) $A \cup B = B \cup A,\ A \cap B = B \cap A$ (**Kommutativgesetze**).

(iv) $A \cup (B \cup C) = (A \cup B) \cup C,\ A \cap (B \cap C) = (A \cap B) \cap C$ (**Assoziativgesetze**); diese Regeln rechtfertigen die Schreibweisen
$A \cup B \cup C = A \cup (B \cup C) = (A \cup B) \cup C$ und
$A \cap B \cap C = A \cap (B \cap C) = (A \cap B) \cap C$.

(v) $A \cup B = (A \setminus B) \cup (A \cap B) \cup (B \setminus A)$, die rechte Seite ist eine **disjunkte Zerlegung** von $A \cup B$. Dabei heißt eine Zerlegung $M = M_1 \cup M_2$ der Menge M disjunkt, wenn $M_1 \cap M_2 = \varnothing$ ist.

(vi) $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$,
$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ (**Distributivgesetze**)

(vii) $A \cap (A \cup B) = A,\ A \cup (A \cap B) = A$.

(viii) Ist $A \subseteq C$, so ist $\left(\overline{\overline{A}^C}\right)^C = A$.

(ix) Sind $A \subseteq C$ und $B \subseteq C$, so gelten
$\overline{(A \cap B)}^C = \overline{A}^C \cup \overline{B}^C,\ \overline{(A \cup B)}^C = \overline{A}^C \cap \overline{B}^C$ (**Regeln von de Morgan**).

(x) Sind $A \subseteq C$ und $B \subseteq C$, so folgt aus $A \subseteq B$ die Beziehung $\overline{B}^C \subseteq \overline{A}^C$ und umgekehrt.

Grundlegende Definitionen aus der Aussagenlogik werden ebenfalls als bekannt vorausgesetzt. Zur Erinnerung werden im folgenden die Wahrheitswerte von Aussagen aufgeführt, die mit Hilfe **logischer Junktoren** aus einfacher aufgebauten Aussagen zusammengesetzt sind:

Wahrheitswerte von			
P	Q	$(P \wedge Q)$	$(P \vee Q)$
FALSCH	FALSCH	FALSCH	FALSCH
FALSCH	WAHR	FALSCH	WAHR
WAHR	FALSCH	FALSCH	WAHR
WAHR	WAHR	WAHR	WAHR
Bezeichnung		**Konjunktion**	**Disjunktion**

Wahrheitswerte von	
P	$(\neg P)$
FALSCH	WAHR
WAHR	FALSCH
Bezeichnung	**Negation**

Neben diesen drei Junktoren werden in logischen Aussagen häufig noch die Junktoren $\Rightarrow$ („impliziert“, „hat zur Folge“, „aus ... folgt ...“), $\Leftrightarrow$ („... ist gleichbedeutend mit ...“, „... gilt genau dann wenn ... gilt“) und $\oplus$ („exklusives oder“, in der englischsprachigen Fachliteratur auch XOR) verwendet, die durch folgende Wahrheitstafeln definiert sind:

Wahrheitswerte von				
P	Q	$(P \Rightarrow Q)$	$(P \Leftrightarrow Q)$	$(P \oplus Q)$
FALSCH	FALSCH	WAHR	WAHR	FALSCH
FALSCH	WAHR	WAHR	FALSCH	WAHR
WAHR	FALSCH	FALSCH	FALSCH	WAHR
WAHR	WAHR	WAHR	WAHR	FALSCH
Bezeichnung		**Implikation**	**Äquivalenz**	**Antivalenz**

Der folgende Satz zeigt strukturelle Äquivalenzen zwischen Sätzen der elementaren Mengenlehre (Satz 1.1-1) und der Aussagenlogik.

Satz 1.1-2:

Es seien P, Q und R Aussagen. Die Aussage W habe den Wahrheitswert `WAHR`, die Aussage F habe den Wahrheitswert `FALSCH`.

Dann sind die folgenden Aussagen Tautologien.

(i) $(P \vee F) \Leftrightarrow P$, $P \wedge F \Leftrightarrow F$.

(ii) $(P \wedge Q) \Rightarrow P$, $(P \wedge Q) \Rightarrow Q$,
$P \Rightarrow (P \vee Q)$, $Q \Rightarrow (P \vee Q)$.

(iii) $(P \vee Q) \Leftrightarrow (Q \vee P)$, $(P \wedge Q) \Leftrightarrow (Q \wedge P)$ (**Kommutativgesetze**).

(iv) $(P \vee (Q \vee R)) \Leftrightarrow ((P \vee Q) \vee R)$,
$(P \wedge (Q \wedge R)) \Leftrightarrow ((P \wedge Q) \wedge R)$ (**Assoziativgesetze**);
diese Regeln rechtfertigen die Schreibweisen $(P \vee Q \vee R)$ anstelle von $(P \vee (Q \vee R))$ und $(P \wedge Q \wedge R)$ anstelle von $(P \wedge (Q \wedge R))$.

(v) $(P \vee Q) \Leftrightarrow ((P \wedge \neg Q) \vee (P \wedge Q) \vee (Q \wedge \neg P))$.

(vi) $(P \wedge (Q \vee R)) \Leftrightarrow ((P \wedge Q) \vee (P \wedge R))$,
$(P \vee (Q \wedge R)) \Leftrightarrow ((P \vee Q) \wedge (P \vee R))$ (**Distributivgesetze**)

(vii) $(P \wedge (P \vee Q)) \Leftrightarrow P$, $(P \vee (P \wedge Q)) \Leftrightarrow P$.

(viii) $(\neg(\neg P)) \Leftrightarrow P$.

Satz 1.1-1:

Es seien im folgenden A, B und C Mengen.

Dann gilt:

(i) $A \cup \varnothing = A$, $A \cap \varnothing = \varnothing$.

(ii) $A \cap B \subseteq A$, $A \cap B \subseteq B$,
$A \subseteq A \cup B$, $B \subseteq A \cup B$.

(iii) $A \cup B = B \cup A$, $A \cap B = B \cap A$ (Kommutativgesetze).

(iv) $A \cup (B \cup C) = (A \cup B) \cup C$,
$A \cap (B \cap C) = (A \cap B) \cap C$ (Assoziativgesetze).

(v) $A \cup B = (A \setminus B) \cup (A \cap B) \cup (B \setminus A)$.

(vi) $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$,
$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
(Distributivgesetze)

(vii) $A \cap (A \cup B) = A$, $A \cup (A \cap B) = A$.

(viii) Ist $A \subseteq C$, so ist $\left(\overline{A^C}\right)^C = A$.

../..

(ix)	$(\neg(P \wedge Q)) \Leftrightarrow (\neg P \vee \neg Q)$, $(\neg(P \vee Q)) \Leftrightarrow (\neg P \wedge \neg Q)$ (**Regeln von de Morgan**)	(ix)	Sind $A \subseteq C$ und $B \subseteq C$, so gelten $\overline{(A \cap B)}^C = \overline{A}^C \cup \overline{B}^C$, $\overline{(A \cup B)}^C = \overline{A}^C \cap \overline{B}^C$ (Regeln von de Morgan)
(x)	$(P \Rightarrow Q) \Leftrightarrow (\neg Q \Rightarrow \neg P)$	(x)	Sind $A \subseteq C$ und $B \subseteq C$, so folgt aus $A \subseteq B$ die Beziehung $\overline{B}^C \subseteq \overline{A}^C$ und umgekehrt.

Allgemein gilt, dass ein Satz der elementaren Mengenlehre, der nur die Relationenzeichen = und $\subseteq$ und die Operatoren $\cup, \cap$ bzw. $^{-C}$ (Komplement) verwendet, eine korrespondierende logische Aussage besitzt und umgekehrt, indem man Ersetzungen gemäß folgender Tabelle vornimmt.

Elementare Mengenlehre	=	$\subseteq$	$\cup$	$\cap$	$^{-C}$
Aussagenlogik	$\Leftrightarrow$	$\Rightarrow$	$\vee$	$\wedge$	$\neg$

Die Erweiterung der Aussagenlogik führt auf die **Prädikatenlogik (erster Stufe)**, in der logische Sätze formuliert werden, die die **Quantoren** $\forall$ („für alle ...“) und $\exists$ („es gibt ...“) enthalten können, wobei über „freie Variablen“ in Formeln quantifiziert wird.

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

$f: A \to B$, $f(a) = \ldots$

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 $f(a) = \ldots$ 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
$f(A) = \{ b \mid b \in B, \text{und es gibt } a \in A \text{ mit } f(a) = b \}$
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: $f(A) = B$. Dieses bedeutet, dass es zu jedem $b \in B$ ein $a \in A$ mit $f(a) = b$ gibt.

Eine Abbildungsvorschrift $f: A \to B$ heißt **bijektiv**, wenn sie injektiv und surjektiv ist. In diesem Fall gibt es eine eindeutig bestimmte **Umkehrfunktion** $f^{-1}: B \to A$ mit $f^{-1}(f(a)) = a$ und $f\left(f^{-1}(b)\right) = b$. 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 $a_1a_2 \ldots a_n$ wird durch $|a_1a_2 \ldots a_n| = n$ definiert. Das **leere Wort (leere Zeichenkette)**, das keinen Buchstaben enthält, wird mit ε bezeichnet; es gilt $|\varepsilon| = 0$.

Formal lassen sich die **Wörter über** Σ wie folgt definieren:

(i) ε ist ein Wort über Σ mit $|\varepsilon| = 0$

(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 $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ gesetzt.

Sind $x = a_1a_2 \ldots a_n$ und $y = b_1b_2 \ldots b_m$ zwei Wörter aus Σ^*, so heißt das Wort $xy = a_1a_2 \ldots a_nb_1b_2 \ldots b_m$ die **Konkatination** von x und y. Die Länge von xy ist $|xy| = |x| + |y|$.

Für $x \in \Sigma^*$ definiert man $x^0 = \varepsilon$ 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 (i) – (iii) definiert:

(i) $L^0 = \{\varepsilon\}$

(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 $L^* = L^+ \cup \{\varepsilon\}$.

Der folgende Satz führt einige Rechenregeln auf:

Satz 1.1-3:

Es sei Σ ein Alphabet, und es seien L, L_1 und L_2 Sprachen über Σ. Dann gilt:

(i) $\varnothing^* = \{\varepsilon\}$.

(ii) $\left(L^*\right)^* = L^*$

(iii) $\left(L_1 \cup L_2\right)^* = \left(L_1^* \cdot L_2^*\right)^*$.

(iv) $L^+ = L \cdot L^* = L^* \cdot L$.

(v) $L \cdot \left(L_1 \cup L_2\right) = \left(L \cdot L_1\right) \cup \left(L \cdot L_2\right)$.

(vi) $L \cdot \left(L_1 \cap L_2\right) \subseteq \left(L \cdot L_1\right) \cap \left(L \cdot L_2\right)$.

Beweis:

Zu (i): Für jede Sprache L gilt: $L^0 = \{\varepsilon\}$ und $L^0 \subseteq L^*$, insbesondere auch für $L = \varnothing$. Daher ist $\{\varepsilon\} \subseteq \varnothing^*$.

Ist umgekehrt $w \in \varnothing^*$, dann ist $w \in \varnothing^0$ oder $w \in \varnothing^n$ mit $n \geq 1$. Im ersten Fall ist $w = \varepsilon$; der zweite Fall kann wegen $\varnothing^n = \varnothing^{n-1} \cdot \varnothing = \varnothing$ nicht auftreten. Daher ist $\varnothing^* \subseteq \{\varepsilon\}$.

Zu (ii): Es sei $w \in \left(L^*\right)^*$. Dann ist $w = \varepsilon$, oder es ist $w \in \left(L^*\right)^n$ mit $n \geq 1$. Im ersten Fall ist (wegen $\{\varepsilon\} \subseteq L^*$) $w \in L^*$. Im zweiten Fall ist $w = w_1 \ldots w_n$ mit $w_i \in L^*$ für $i = 1, \ldots, n$. Jedes w_i besteht nur aus Buchstaben aus L, d.h. $w_i = a_{i,1} \ldots a_{i,j_i}$ mit $a_{l,k} \in \Sigma$. Setzt man $m = \sum_{i=1}^{n} j_i$, so sieht man $w \in L^m$, und wegen $L^m \subseteq L^*$ ist auch in diesem Fall $w \in L^*$.

Ist umgekehrt $w \in L^*$, so ist entweder $w = \varepsilon$ oder $w = a_1 \ldots a_n$ für ein $n \geq 1$ und $a_i \in L$ für $i = 1, \ldots, n$. Im ersten Fall sieht man direkt $w \in \left(L^*\right)^*$. Im zweiten Fall ist wegen $a_1 \ldots a_n = \left(a_1 \ldots a_n\right)^1$ und $a_1 \ldots a_n \in L^*$: $w \in \left(L^*\right)^1$ und damit wegen $\left(L^*\right)^1 \subseteq \left(L^*\right)^*$ auch $w \in \left(L^*\right)^*$.

Zu (iii): Es sei $w \in (L_1 \cup L_2)^*$. Ist $w = \varepsilon$, so ist $w \in (L_1^* \cdot L_2^*)^*$. Ist $w \neq \varepsilon$, so ist $w = w_1 ... w_n$ mit $n \geq 1$ und $w_i \in L_1 \cup L_2$ für $i = 1, ..., n$. Ist $w_i \in L_1$, so ist wegen $w_i = w_i \cdot \varepsilon$, $\varepsilon \in L_2^*$ und $L_1 \subseteq L_1^*$ auch $w_i \in L_1^* \cdot L_2^*$. Ist $w_i \in L_2$, so ist wegen $w_i = \varepsilon \cdot w_i$, $\varepsilon \in L_1^*$ und $L_2 \subseteq L_2^*$ auch $w_i \in L_1^* \cdot L_2^*$. Insgesamt ist daher $w_1 ... w_n \in (L_1^* \cdot L_2^*)^*$.

Ist umgekehrt $w \in (L_1^* \cdot L_2^*)^*$, so besteht w aus Buchstaben aus $L_1 \cup L_2$, d.h. $w \in (L_1 \cup L_2)^*$.

Zu (iv): Es sei $w \in L^+$. Dann ist $w = a_1 ... a_n$ mit $n \geq 1$ und $a_i \in L$ für $i = 1, ..., n$. Es ist $w = a_1 \cdot a_2 ... a_n = a_1 ... a_{n-1} \cdot a_n$, also $w \in L \cdot L^*$ und $w \in L^* \cdot L$, d.h. $L^+ \subseteq L \cdot L^*$ und $L^+ \subseteq L^* \cdot L$. Ist umgekehrt $w \in L \cdot L^*$, so besteht *w* aus Buchstaben aus *L* und ist nicht das leere Wort. Daher ist $w \in L^+$, d.h. $L \cdot L^* \subseteq L^+$. Genauso ergibt sich $L^* \cdot L \subseteq L^+$. Insgesamt folgt $L \cdot L^* = L^+ = L^* \cdot L$.

Zu (v): Es gilt $w \in L \cdot (L_1 \cup L_2)$ genau dann, wenn $w = w_1 w_2$ ist mit $w_1 \in L$ und $w_2 \in L_1 \cup L_2$. Das ist gleichbedeutend mit $w \in L \cdot L_1$ oder $w \in L \cdot L_2$ bzw. $w \in L \cdot L_1 \cup L \cdot L_2$.

Zu (vi): Ist $w \in L \cdot (L_1 \cap L_2)$, dann ist $w = w_1 w_2$ ist mit $w_1 \in L$ und $w_2 \in L_1 \cap L_2$. Daher gilt $w \in L \cdot L_1$ und $w \in L \cdot L_2$ bzw. $w \in L \cdot L_1 \cap L \cdot L_2$.

///

Bemerkung: Mit $L = \{\varepsilon, 1\}$, $L_1 = \{0\}$ und $L_2 = \{10\}$ sieht man, dass die Inklusion in Satz 1.1-3 (vi) nicht umkehrbar ist.

Zwei Mengen *A* und *B* heißen **gleichmächtig**, wenn es eine bijektive Abbildung $f : A \to B$ gibt. Wegen der Existenz der bijektiven Umkehrfunktion *g* zu *f* ist diese Definition gleichbedeutend mit der Existenz einer bijektiven Abbildung $g : B \to A$.

Bei unendlichen Mengen *A* und *B* tritt die folgende Situation auf, die sich am Beispiel der Vorgängerfunktion *pred* auf den natürlichen Zahlen verdeutlichen lässt:

$$pred : \begin{cases} \mathbf{N}_{>0} & \to & \mathbf{N} \\ n & \to & n-1 \end{cases}$$

Wie man leicht nachrechnet, handelt es sich hierbei um eine bijektive Abbildung, d.h. die Menge **N** der natürlichen Zahlen ist gleichmächtig mit der echten Teilmenge $\mathbf{N}_{>0} = \mathbf{N} \setminus \{ 0 \}$. Dieses Phänomen erlaubt es, die Endlichkeit bzw. Unendlichkeit einer Menge exakt zu definieren:

Eine Menge A ist **endlich von der Mächtigkeit** n, wenn es eine bijektive Abbildung $f : \{ 0, ..., n-1 \} \to A$ gibt, d.h. man kann die Elemente in A mit den natürlichen Zahlen 0, ..., $n-1$ durchnumerieren: $A = \{ f(0), ..., f(n-1) \} = \{ a_0, ..., a_{n-1} \}$. Hierbei ist $f(i) \neq f(j)$ bzw. $a_i \neq a_j$ für $i \neq j$.

Eine Menge A ist **von der Mächtigkeit unendlich**, wenn es eine bijektive Abbildung $f : B \to A$ zwischen einer echten Teilmenge $B \subset A$ und A gibt.

Eine Menge heißt **abzählbar**, wenn sie entweder endlich oder gleichmächtig zu den natürlichen Zahlen ist. Eine unendliche Menge, die nicht abzählbar ist, heißt **überabzählbar**.

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	ε	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:	$n+n^2+2$	$a_1\ a_1\ a_2$	(Anzahl: n^3)
2	a_2	(Anzahl: n)	...	...	
...	...		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:	...	...	
$n+2$	$a_1\ a_2$	(Anzahl: n^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 ε 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 1.1-4:

Es sei Σ ein endliches Alphabet. Dann gilt:

(i) Σ^* ist abzählbar unendlich.

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

Teil (ii) des Satzes folgt aus der Tatsache, dass jede Teilmenge einer abzählbar unendlichen Menge abzählbar ist.

Satz 1.1-5:

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

Beweis:

Es sei $M = \{ m_0, ..., m_{n-1} \}$ eine endliche Menge mit *n* Elementen. Jede Teilmenge $N \subseteq M$ mit *k* Elementen, etwa $N = \{ m_{i_0}, ..., m_{i_{k-1}} \}$ 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.

///

Satz 1.1-6:

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** angewandt:
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, $\mathbf{P}(M)$ ist 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 sofort das folgende Ergebnis:

Satz 1.1-7:
Die Menge $\{ L \mid L \subseteq \Sigma^* \}$ aller Sprachen über einem endlichen Alphabet Σ ist überabzählbar.

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, die sich direkt aus der Definition der Größenordnung einer Funktion herleiten lassen, lauten:

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

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 1.1-8:

(i) Ist $|f(n)|$ für fast jedes $n \in \mathbf{N}$ durch eine Konstante beschränkt, so ist $f(n) \in O(1)$.

(ii) 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 $k \ge m$,
insbesondere $n^m \in O(n^k)$ für $k \ge m$.

Beweis:

Zu (i): Die Aussage folgt direkt aus der Definition des Operators *O*.

Zu (ii): Mit $c = \max\{|a_0|, ..., |a_m|\}$ und $n \ge 2$ ist

$$|p(n)| \le \sum_{i=0}^{m} |a_i| \cdot n^i \le c \cdot \sum_{i=0}^{m} n^i = c \cdot \frac{n^{m+1} - 1}{n - 1} = c \cdot \frac{n^m - 1/n}{1 - 1/n} \le 2c \cdot n^m \le 2c \cdot n^k$$

für $k \ge m$, also nach Definition $p(n) \in O(n^k)$.

///

Aus der Analysis ist für $a > 1$ und $b > 1$ die Umrechnung $\log_a(n) = \frac{1}{\log_b(a)} \cdot \log_b(n)$ bekannt.

Daher gilt

Satz 1.1-9:

Für $a > 1$ und $b > 1$ ist $\log_a(n) \in O(\log_b(n))$.

Statt $O(\log_b(n))$ schreibt man daher $O(\log(n))$.

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

Satz 1.1-10:

Für $a > 1$ ist $a^{h(n)} \in O\left(2^{O(h(n))}\right)$.

Da eine Exponentialfunktion der Form $f(n) = a^n$ mit $a > 1$ schneller wächst als jedes Polynom $p(n)$, genauer: $\lim\limits_{n \to \infty} \dfrac{|p(n)|}{a^n} = 0$, ist $a^n \notin O(p(n))$, jedoch $p(n) \in O\left(a^n\right)$.

Entsprechend gelten für jede Logarithmusfunktion $\log_a(n)$ mit $a > 1$ und jedes Polynom $p(n)$ die Beziehungen $p(n) \notin O(\log_a(n))$ und $\log_a(n) \in O(p(n))$.

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

Insgesamt ergibt sich damit

Satz 1.1-11:

Es seien $a > 1$, $b > 1$, $m \in \mathbf{N}_{>\mathbf{0}}$, $p(n)$ ein Polynom; dann ist

$O(\log_b(n)) \subset O\left(\sqrt[m]{n}\right) \subset O(p(n)) \subset O\left(a^n\right)$.

Bei der Analyse des Laufzeitverhaltens von Algorithmen treten häufig Funktionen auf, die arithmetische Verknüpfungen von Logarithmusfunktionen, Polynomen und Exponentialfunktionen sind. Man sagt, eine durch $f(n)$ gegebene Funktion hat **langsameres Wachstum** als eine durch $g(n)$ gegebene Funktion, wenn $O(f(n)) \subseteq O(g(n))$ ist. Entsprechend weist g schnelleres Wachstum als f auf. So sind beispielsweise die folgenden Funktionen gemäß ihrer Wachstumsgeschwindigkeit geordnet:

langsames Wachstum $\longrightarrow$ schnelles Wachsstum

$(\log_2(n))^5 \longrightarrow n^2 \longrightarrow n^3 \longrightarrow \frac{n^4}{\log_2(n)}$ und

$(\log_2(n))^2 \longrightarrow n \longrightarrow n \cdot (\log_2(n))^3 \longrightarrow \frac{n^2}{\log_2(n)}$.

Ein **Boolescher Ausdruck** (**Boolesche Formel**) ist eine Zeichenkette über dem Alphabet $A = \{\wedge, \vee, \neg, (,), 0, 1, x\}$, der gemäß folgenden Regeln aufgebaut ist:

(i) 0 und 1 sind Boolesche Ausdrücke und heißen **Konstanten.**

(ii) $xa_n \ldots a_0$ mit $a_0 \in \{0,1\}$, ..., $a_n \in \{0,1\}$ ist ein Boolescher Ausdruck; mit $i = \sum_{k=0}^{n} a_k \cdot 2^k$ steht dieser Boolesche Ausdruck für die **Variable** x_i.

(iii) Ist F ein Boolescher Ausdruck, so ist auch $(\neg F)$ ein Boolescher Ausdruck und heißt **Negation** des Booleschen Ausdrucks F.

(iv) Sind F_1 und F_2 Boolesche Ausdrücke, so ist auch $(F_1 \wedge F_2)$ ein Boolescher Ausdruck und heißt **Konjunktion** der Booleschen Ausdrücke F_1 und F_2.

(v) Sind F_1 und F_2 Boolesche Ausdrücke, so ist auch $(F_1 \vee F_2)$ ein Boolescher Ausdruck und heißt **Disjunktion** der Booleschen Ausdrücke F_1 und F_2.

(vi) Ein Boolescher Ausdruck entsteht genau aus einer endlichen Anzahl von Anwendungen der Regeln (i) bis (v).

Die Zeichen $\neg$, $\wedge$ und $\vee$ heißen **Junktoren**.

Zur Vereinfachung Boolescher Ausdrücke kann man überflüssige Klammern weglassen, wenn man unterschiedliche **Bindungsstärke der Junktoren** voraussetzt: In der Reihenfolge $\neg$, $\wedge$ und $\vee$ bindet ein weiter vorn stehender Junktor jeweils stärker als ein weiter hinten stehender Junktor. Im Zweifelsfalle werden jedoch zur Verdeutlichung Klammern beibehalten.

Außerdem wird meist anstelle des Booleschen Ausdrucks $xa_n \ldots a_0$ mit $a_0 \in \{0,1\}$, ..., $a_n \in \{0,1\}$ die für den Boolescher Ausdruck stehende Variable x_i mit $i = \sum_{k=0}^{n} a_k \cdot 2^k$ eingesetzt.

Beispielsweise wird der Boolesche Ausdruck $(((x1 \wedge (\neg x101)) \vee ((\neg x11) \wedge (x100 \vee x101))) \vee 0)$ auf diese Weise vereinfacht zu $(x_1 \wedge \neg x_5) \vee (\neg x_3 \wedge (x_4 \vee x_5)) \vee 0$.

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

Die Variablen eines Booleschen Ausdrucks können mit den **Wahrheitswerten** FALSE (FALSCH) bzw. TRUE (WAHR) belegt werden. Die Konstante 0 bekommt den Wahrheitswert FALSE (FALSCH), die Konstante 1 den Wahrheitswert TRUE (WAHR). Enthält ein Boolescher Ausdruck *n* unterschiedliche Variablen, so gibt es 2^n mögliche Belegungen mit Wahrheitswerten. Nach Festlegung einer Belegung der Variablen und Konstanten eines Booleschen Ausdrucks mit Wahrheitswerten kann dieser ausgewertet werden und so der Wahrheitswert des gesamten Booleschen Ausdrucks unter der vorgegebenen Belegung bestimmt werden. Die Auswertung erfolgt „von innen nach außen" gemäß seinem Aufbau, wie er durch die Reihenfolge der Anwendung obiger Regeln (iii) bis (v) entstanden ist. Dabei hängt der Wahrheitswert eines zusammengesetzten Booleschen Ausdrucks, der aufgrund der Anwendung einer der Regeln (iii) bis (v) entstanden ist, etwa $(\neg F)$, $(F_1 \wedge F_2)$ bzw. $(F_1 \vee F_2)$, von den Wahrheitswerten der im Aufbau einfacheren Booleschen Ausdrücke F, F_1 bzw. F_2 wie folgt ab:

Wahrheitswerte von	
F	$(\neg F)$
FALSE	TRUE
TRUE	FALSE
Bezeichnung	**Negation**

Wahrheitswerte von			
F_1	F_2	$(F_1 \wedge F_2)$	$(F_1 \vee F_2)$
FALSE	FALSE	FALSE	FALSE
FALSE	TRUE	FALSE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	TRUE	TRUE	TRUE
Bezeichnung		**Konjunktion**	**Disjunktion**

In vielen Anwendungen tritt auch noch die Exklusiv-Oder-Operation $\oplus$ auf, die definiert ist durch:

Wahrheitswerte von		
F_1	F_2	$(F_1 \oplus F_2)$
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	FALSE
Bezeichnung		**Exklusiv-Oder**

Ein Boolescher Ausdruck *F* heißt **erfüllbar**, wenn es eine Belegung der Variablen und Konstanten in *F* durch Werte FALSE bzw. TRUE gibt, so dass sich bei Auswertung der Wahrheitswert TRUE ergibt.

Ein Boolescher Ausdruck *F* ist in **konjunktiver Normalform**, wenn *F* die Form
$F = F_1 \wedge ... \wedge F_m$
und jedes F_j die Form
$F_j = \left(y_{j_1} \vee ... \vee y_{j_k}\right)$
hat, wobei y_{j_k} ein Literal ist, d.h. für eine Variable (d.h. $y_{j_k} = x_i$) oder für eine negierte Variable (d.h. $y_{j_k} = \neg x_i$) steht.

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

Eine Klausel F_j von *F* ist durch eine Belegung der in *F* vorkommenden Variablen erfüllt, wenn mindestens ein Literal in F_j 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. Die Konstruktion eines Booleschen Ausdrucks in konjunktiver Normalform zu einem gegebenen Booleschen Ausdruck „in allgemeiner Form“ wird in Kapitel 5.5 beschrieben.

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).

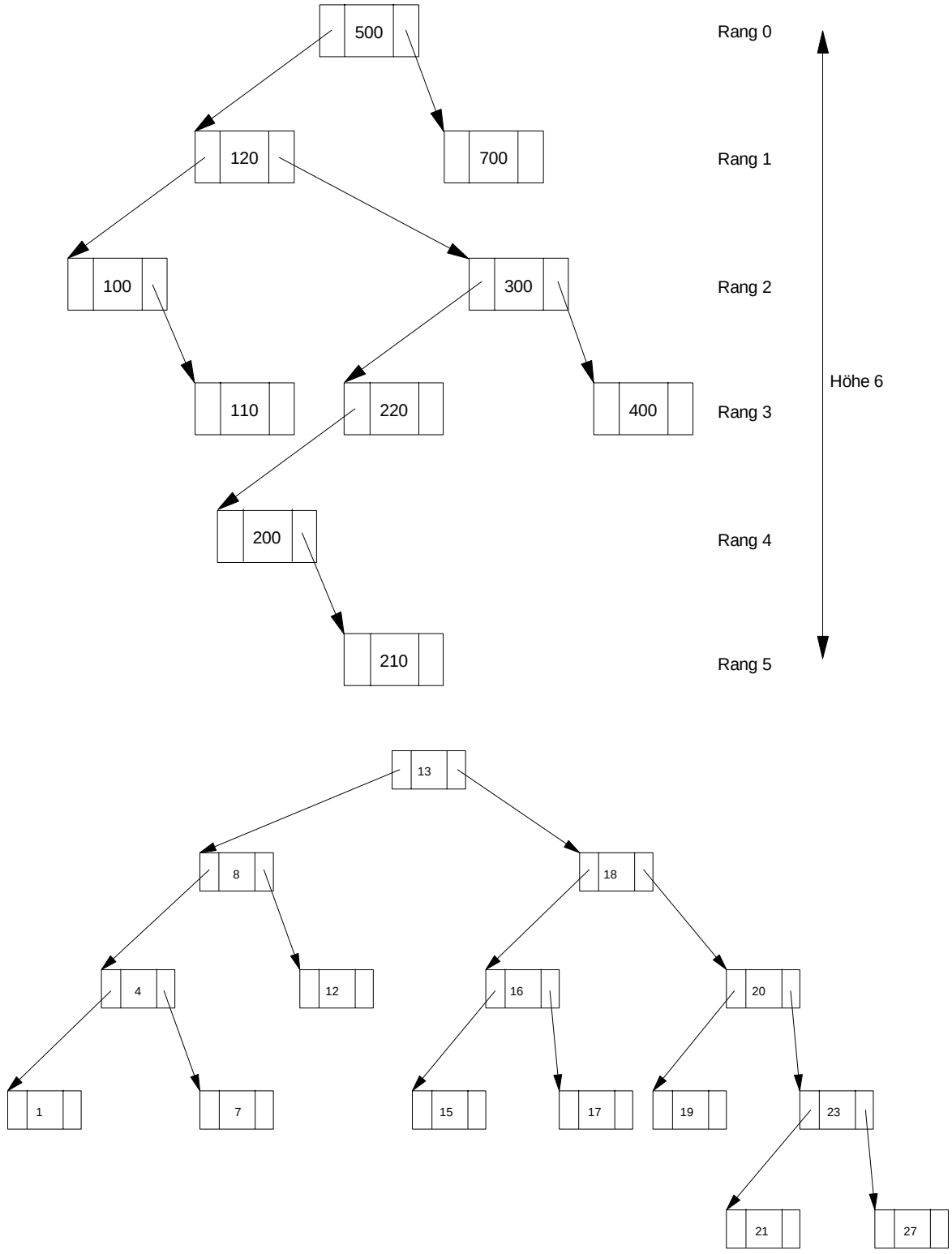
Die Kante $e = (v_i, v_j)$ läuft von v_i nach v_j (verbindet v_i mit v_j). Der Knoten v_i heißt **Anfangsknoten** der Kante $e = (v_i, v_j)$, der Knoten v_j **Endknoten** von $e = (v_i, v_j)$. Zu einem Knoten $v \in V$ heißt $pred(v) = \{v' \mid (v', v) \in E\}$ die **Menge der direkten Vorgänger** von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

Bei einem **ungerichteten Graphen** ist keine Richtung der Kanten festgelegt. Eine Kante hat also die Form $\{v_i, v_j\}$. Im folgenden wird jedoch auch bei einem ungerichteten Graphen eine Kante meist mit (v_i, v_j) bezeichnet.

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 $\big((a_0, a_1), (a_1, a_2), \ldots, (a_{m-1}, a_m)\big)$ 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 1.1-12:

(i) 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 $\sum_{i=0}^{j} 2^i = 2^{j+1} - 1$ Knoten.

(ii) 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:
$\lceil \log_2(n+1) \rceil \leq h(B_n) \leq n$.

(iii) Für die Anzahl $b(h)$ der Blätter eines Binärbaums der Höhe $h \geq 1$ gilt
$1 \leq b(h) \leq 2^{h-1}$.

(iv) Für die Mindesthöhe $h(b)$ eines Binärbaums mit $b \geq 1$ vielen Blättern gilt
$h(b) \geq \lceil \log_2(b) + 1 \rceil$.

(v) 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 $\sqrt{\pi n} + C$ 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*.

Beweis:[1]

Zu (i): Die Aussage ergibt sich durch vollständige Induktion.

Zu (ii): Die obere Abschätzung $h(B_n) \leq n$ ist offensichtlich. Für die untere Abschätzung betrachtet man einen Binärbaum mit *n* Knoten und minimaler Höhe *h*. Jedes Niveau, bis eventuell das höchste Niveau *m*, ist vollständig gefüllt. Es ist $h = m+1$. Bis zum Niveau $m-1$ enthält der Binärbaum gemäß (i) insgesamt $2^{m-1+1} - 1 = 2^m - 1$ viele Knoten, auf Niveau *m* sind es mindestens einer und höchstens 2^m. Daraus folgt: $2^m - 1 + 1 \leq n \leq 2^m - 1 + 2^m$, also $2^m \leq n \leq 2^{m+1} - 1$ und damit $m < \log_2(n+1) \leq m+1$,

1 Teil (v) wird in diesem Text nicht verwendet; der Beweis kann beispielsweise in Graham, Knuth, Patashnik: **Concrete Mathematics**, 7. Auflage, Addison-Wesley, 1991, nachgelesen werden.

d.h. $\lceil \log_2(n+1) \rceil = m+1 = h$. Für einen beliebigen Binärbaum mit n Knoten gilt daher $\lceil \log_2(n+1) \rceil \leq h(B_n)$.

Zu (iii): Jeder Binärbaum mit Höhe $h \geq 1$ hat mindestens ein Blatt. Daraus ergibt sich die untere Abschätzung in (iii). Ein Binärbaum hat besonders viele Blätter, wenn diese sämtlich auf dem maximalen Niveau $m = h-1$ liegen. Mit Teil (i). folgt $b(h) \leq 2^{h-1}$.

Zu (iv): Löst man die zweite Ungleichung in (iii) nach $h = h(b)$ auf, so ergibt sich $h \geq \log_2(b)+1$. Da h eine natürliche Zahl ist, folgt die Abschätzung $h(b) \geq \lceil \log_2(b)+1 \rceil$.

///

1.2 Problemklassen

In der Informatik beschäftigt man sich häufig damit, Problemstellungen mit Hilfe von Rechnerprogrammen zu lösen. Ein **Problem** ist eine zu beantwortende Fragestellung, die von **Problemparametern** als Eingaben abhängt, deren genaue Werte in der Problembeschreibung zunächst unspezifiert sind, deren Typbeschreibung jedoch in die Problembeschreibung eingeht. Ein Problem wird beschrieben durch:

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

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 drei grundlegende **Problemtypen** unterschieden und zunächst an Beispielen erläutert.

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 optimale Tour durch *G*.

Dabei ist eine **Tour** 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 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
$\sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}}) + w(v_{i_n}, v_{i_1})$. Eine optimale Tour ist eine Tour mit minimalen Kosten (unter allen möglichen Touren durch *G*).

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 die *Kosten* 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: $[G, K]$,

$G = (V, E, w)$ 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 nichtnegatives Gewicht; $K \in \mathbf{R}_{\geq 0}$

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 optimalen 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 Optimierungs- und das Berechnungsproblem.

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.

Die Beispiele lassen 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
$L_\Pi = \{ u \in \Sigma_\Pi^* \mid u \text{ hat die beschriebene Eigenschaft} \}$

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 besitzt, die genau allen Elementen in L_Π zukommt, 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 $m(G,T) = \sum_{j=1}^{n-1} w(v_{i_j}, v_{i_{j+1}}) + w(v_{i_n}, v_{i_1})$

4. $goal = \min$

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

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 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 Schleifen (**WHILE** ... **DO** ..., **FOR** `i` := ... **TO** … **DO**..., **REPEAT** … **UNTIL** ...) 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 $f(x)$ in einem einzigen (elementaren) Schritt einen von endlich vielen Werten als korrekten Wert aussuchen“ ist als Teil eines deterministischen Verfahrens nicht zulässig[2].

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.

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.

---

2 Die Forderung nach Determinismus des Verfahrens wird in späteren Kapiteln gelegentlich fallengelassen.

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)? Wie viele Schritte wird jeder Algorithmus mindestens durchführen, der das vorgelegte Problem löst?

Die Beantwortung dieser Frage liefert untere Komplexitätsschranken.

Mit diesem noch sehr „vagen" Algorithmusbegriff lässt sich bereits zeigen, dass es sehr einfache Funktionen gibt, die algorithmisch 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 1.1-6):
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. $B = \{ f(0), f(1), f(2), ... \}$. Der Wert $f(i)$ bezeichnet das Berechnungsverfahren in *B* mit der Nummer *i*. Die von $f(i)$ berechnete Funktion sei $f_i : \mathbf{N} \to \{0,1\}$.

Es wird eine Funktion $g_0 : \mathbf{N} \to \{0,1\}$ wie folgt definiert:

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

Angenommen, die Funktion g_0 ist berechenbar, d.h. es gibt ein Berechnungsverfahren für g_0 in *B*. Es sei $i_g \in \mathbf{N}$ die Nummer dieses Berechnungsverfahrens, 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 eine nichtberechenbare Funktion der Form $g:\mathbf{N}\to\{0,1\}$.

///

2 Modelle der Berechenbarkeit und Komplexität von Berechnungen

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 „vernünftige“ 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 μ-rekursiven Funktionen oder des Churchsche λ-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 einen ab-

zählbar 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
$TM = \left(Q, \Sigma, I, \delta, b, q_0, q_{accept} \right)$
mit:

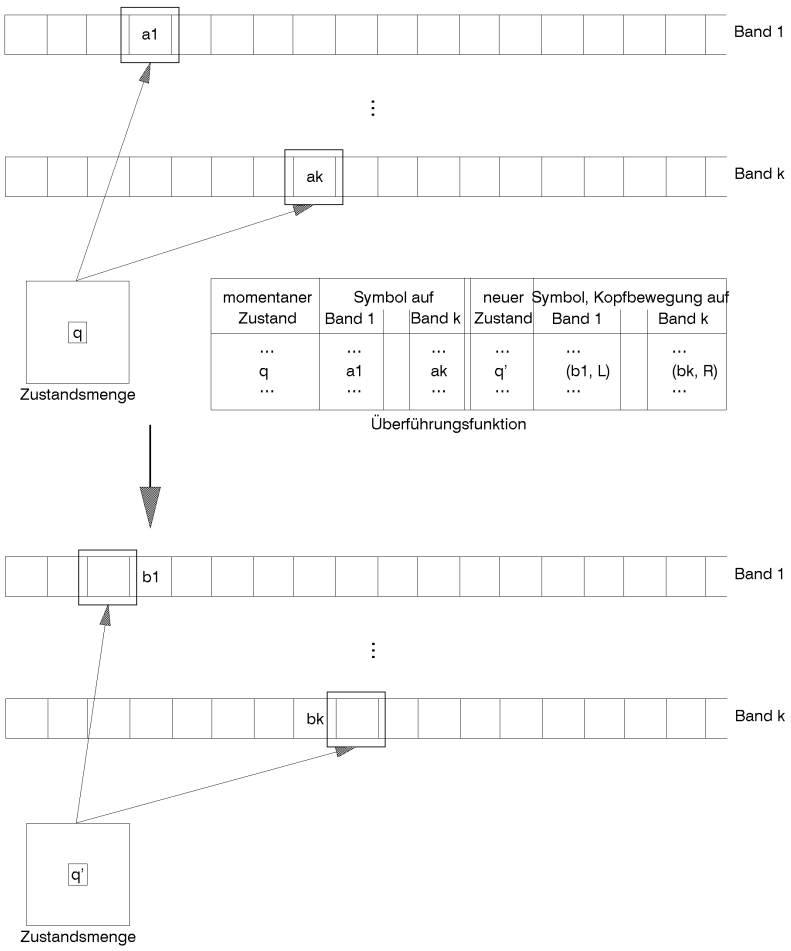
1. Q ist eine endliche nichtleere Menge: die **Zustandsmenge**
2. Σ ist eine endliche nichtleere Menge: das **Arbeitsalphabet**; Σ enthält alle Zeichen, die in Feldern der Bänder (siehe unten) stehen können
3. $I \subseteq \Sigma$ ist eine endliche nichtleere Menge: das **Eingabealphabet**; mit den Zeichen 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. $\delta : (Q \setminus \{q_{accept}\}) \times \Sigma^k \to Q \times (\Sigma \times \{L, R, S\})^k$ ist eine partielle Funktion, die **Überführungsfunktion**; insbesondere ist $\delta(q, a_1, \ldots, a_k)$ für $q = q_{accept}$ nicht definiert; zu beachten ist, dass δ für 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 δ 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 δ 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
$\delta(q, a_1, \ldots, a_k) = (q', (b_1, d_1), \ldots, (b_k, d_k))$,
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 $\delta(q,a_1,\ldots,a_k)$ 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 $\delta(q,a_1,\ldots,a_k)$ für $q = q_{accept}$ nicht definiert ist. Sie kann aber auch eventuell vorher in einem anderen Zustand anhalten (nämlich dann, wenn $\delta(q,a_1,\ldots,a_k)$ 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 $\delta(q,a_1,\ldots,a_k) = (q,(a_1,S),\ldots,(a_k,S))$ ist. Eine weitere 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 $\delta(q,b,\ldots,b) = (q,(b,R),\ldots,(b,R))$ ist.

Die Werte $\delta(q,a_1,\ldots,a_k) = (q',(b_1,d_1),\ldots,(b_k,d_k))$ 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:

$K = (q, (\alpha_1, i_1), ..., (\alpha_k, i_k))$ mit $q \in Q$, $\alpha_j \in \Sigma^*$, $i_j \geq 1$ 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 α_j (jeder Buchstabe von α_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 $i_j \in [1 : |\alpha_j|]$ ist dieses der i_j-teBuchstaben von α_j, für $i_j \geq |\alpha_j| + 1$ steht der Schreib/Lesekopf hinter α_j über einer Zelle, die das Leerzeichen enthält.

Ist a_j der i_j-te Buchstabe von α_j bzw. $a_j = b$ für $i_j \geq |\alpha_j| + 1$, und ist
$\delta(q, a_1, ..., a_k) = (q', (b_1, d_1), ..., (b_k, d_k))$,
dann geht die TM in die **Folgekonfiguration** K' über, die durch
$K' = (q', (\beta_1, i_1'), ..., (\beta_k, i_k'))$
definiert wird. Dabei entsteht β_j aus α_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 linksbündig 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

$K_0 = (q_0, (w,1), (\varepsilon,1), ..., (\varepsilon,1))$,

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 zur Vereinfachung der Notation häufig 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

$K_{accept} = (q_{accept}, (\alpha_1, i_1), ..., (\alpha_k, i_k))$,

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

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

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

mit einer Endkonfiguration K_{accept}.

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

$$L(TM) = \{ w \mid w \in I^* \text{ und } w \text{ wird von } TM \text{ akzeptiert} \}$$
$$= \{ w \mid w \in I^* \text{ und } (q_0, (w,1), (\varepsilon,1), ..., (\varepsilon,1)) \Rightarrow^* (q_{accept}, (\alpha_1, i_1), ..., (\alpha_k, i_k)) \}.$$

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 $\} \cup \{\varepsilon\}$

Die 2-DTM $TM = (Q, \Sigma, I, \delta, b, q_0, q_{accept})$ wird gegeben durch
$Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1, b\}$, $I = \{0,1\}$, $q_{accept} = q_1$ mit der Überführungsfunktion δ, 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_{accept}	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 $w = a_1 ... a_{n-1} a_n = a_n a_{n-1} ... a_1 \} \cup \{\varepsilon\}$.

$TM = (Q, \Sigma, I, \delta, b, q_0, q_{accept})$ 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 $q_{accept} = q_5$ und stoppt. Sonst stoppt die Maschine in einem von q_{accept} 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. $\delta(q,...)$ ist nicht definiert). Dann ist $w \notin L(TM)$.

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

3. Fall: *TM* ist noch nicht stehengeblieben, d.h. *TM* befindet sich in einem Zustand $q \neq q_{accept}$, für den $\delta(q,...)$ 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).

$TM = (Q, \Sigma, I, \delta, b, q_0, q_{accept})$ 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:

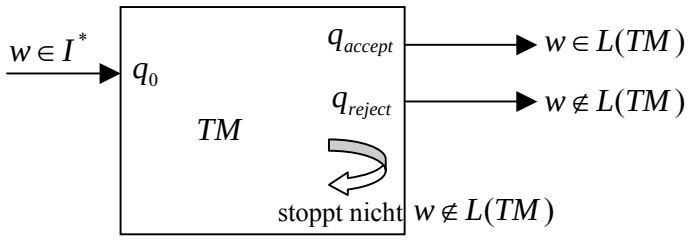
Berechnet die Turingmaschine *TM* die partielle Funktion $f_{TM} : I^* \to \Sigma^*$, dann kann man eine Turingmaschine TM' konstruieren mit $L(TM') = \{ w\#y \mid y = f_{TM}(w) \}$.

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:
$(q_0,(w,1),(\varepsilon,1),...,(\varepsilon,1),(\varepsilon,1)) \Rightarrow^* (q_{accept},(\alpha_1,i_1),(\alpha_2,i_2),...,(\alpha_{k-1},i_{k-1})(y,|y|+1))$

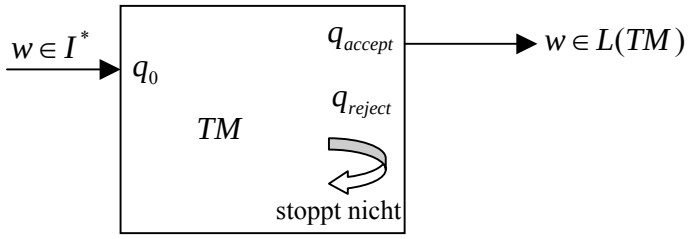
Die erste Aussage lässt die Interpretation zu, dass die Verifikation eines Funktionsergebnisses genauso komplex ist wie die Berechnung des Funktionsergebnisses selbst. Die zweite Aussage verbindet die Akzeptanz einer Sprache mit dem Berechnungsvorgang einer Funktion.

Die Überführungsfunktion δ einer Turingmaschine *TM* werde folgendermaßen modifiziert: Die Zustandsmenge *Q* wird um einen neuen Zustand q_{reject} erweitert. Ist für einen bisherigen Zustand $q \in Q$ mit $q \neq q_{accept}$ und für $(a_1,...,a_k) \in \Sigma^k$ der Wert $\delta(q,a_1,...,a_k)$ *nicht* definiert, so wird δ um den Eintrag $\delta(q,a_1,...,a_k) = (q_{reject},(a_1,S),...,(a_k,S))$ erweitert. Für q_{reject} ist δ 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 $\delta(q,...)$ 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 δ 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.

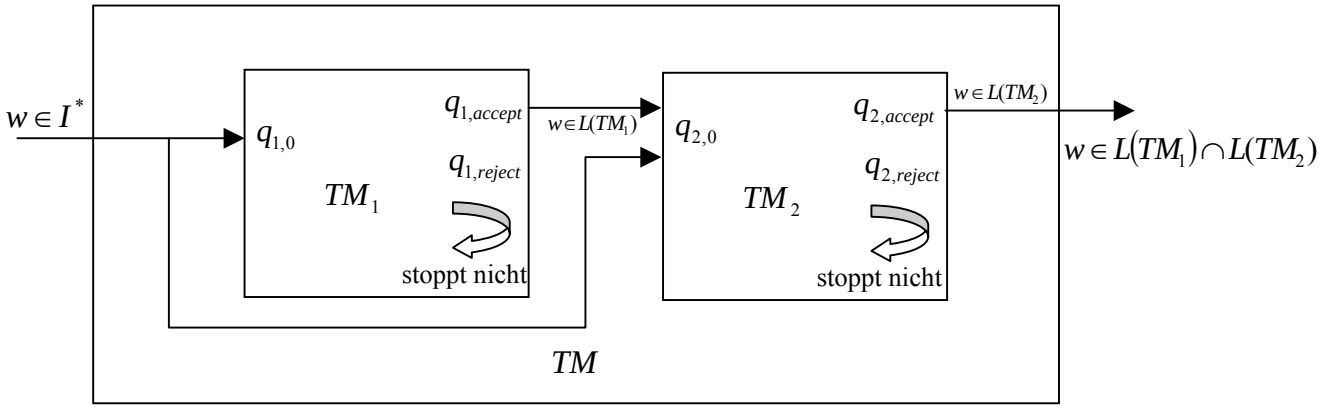


Da die Turingmaschine *TM* dazu verwendet wird, um festzustellen, ob ein Eingabewort $w \in I^*$ von ihr akzeptiert wird, kann man *TM* vereinfacht auch folgendermaßen darstellen:



Turingmaschinen können zu neuen Turingmaschinen zusammengesetzt werden: Es seien zwei Turingmaschinen $TM_1 = \left(Q_1, \Sigma_1, I, \delta_1, b, q_{1,0}, q_{1,accept}\right)$ und $TM_2 = \left(Q_2, \Sigma_2, I, \delta_2, b, q_{2,0}, q_{2,accept}\right)$ 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 unübersichtlich:

Es ist $TM = \left(Q, \Sigma, I, \delta, b, q_{1,0}, q_{2,accept}\right)$. Dabei ist
$Q = Q_1 \cup Q_2 \cup \left\{q_{copy}, q_{skip}\right\}$ 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,
$\delta : (Q \setminus \{q_{2,accept}\}) \times \Sigma^{k_1+k_2} \to Q \times \left(\Sigma \times \{L, R, S\}\right)^{k_1+k_2}$ die Überführungsfunktion, die sich aus den Überführungsfunktionen δ_1 und δ_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 steht der Kopf des ersten Bands über der (*n*+1)-ten Zelle und der Kopf des $(k_1 + 1)$-ten Bands über der (*n*+2)-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 δ von *TM* aufgenommen:

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

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

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

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

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

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

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

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

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 δ lauten:

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

für jeden Eintrag $\delta_1(q_1, a_1, a_2, ..., a_{k_1}) = (q_1', (b_1, d_1), (b_2, d_2), ..., (b_{k_1}, d_{k_1}))$ in der Überführungsfunktion δ_1 von TM_1 und $a \in I$.

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 Bändern ändert sich dabei nichts mehr. Folgende Zeilen werden in die Überführungsfunktion δ aufgenommen:

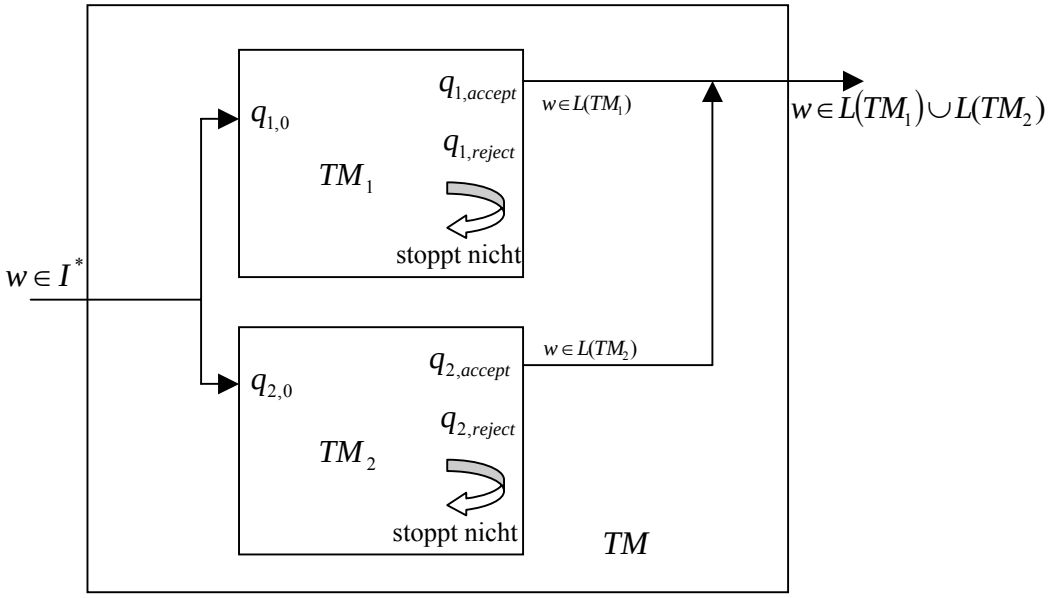
$$\delta\left(q_{1,accept}, \underbrace{a_1, a_2, ..., a_{k_1}}_{k_1}, \underbrace{a, b, ..., b}_{k_2}\right) = \left(q_{2,0}, \underbrace{(a_1, S), (a_2, S), ..., (a_{k_1}, S)}_{k_1}, \underbrace{(a, S), (b, S), ..., (b, S)}_{k_2}\right)$$

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

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

für jeden Eintrag $\delta_2(q_2, g_1, g_2, ..., g_{k_2}) = (q_2', (h_1, d_1), (h_2, d_2), ..., (h_{k_2}, d_{k_2}))$ in der Überführungsfunktion δ_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 $TM = (Q, \Sigma, I, \delta, b, q_0, q_{accept})$. 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,

$\delta: (Q \setminus \{(q_{acc}, q_{acc})\}) \times \Sigma^{k_1+k_2} \to Q \times (\Sigma \times \{L,R,S\})^{k_1+k_2}$ die Überführungsfunktion, die sich aus den Überführungsfunktionen δ_1 und δ_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:

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

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

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

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

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

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

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

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

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 ursprüng-

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

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

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

falls $\delta_1(q_1, a_1, a_2, ..., a_{k_1}) = (q_1', (b_1, d_1), (b_2, d_2), ..., (b_{k_1}, d_{k_1}))$ in der Überführungsfunktion δ_1 von TM_1 und $\delta_2(q_2, g_1, g_2, ..., g_{k_2}) = (q_2', (h_1, e_1), (h_2, e_2), ..., (h_{k_2}, e_{k_2}))$ in der Überführungsfunktion δ_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 δ aufgenommen:

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

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

mit $a_1 \in \Sigma_1$, ..., $a_{k_1} \in \Sigma_1$ und falls $\delta_2(q_2, g_1, g_2, ..., g_{k_2}) = (q_2', (h_1, e_1), (h_2, e_2), ..., (h_{k_2}, e_{k_2}))$ in der Überführungsfunktion δ_2 von TM_2 ist und

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

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

für jeden Eintrag $\delta_1(q_1, a_1, a_2, ..., a_{k_1}) = (q_1', (b_1, d_1), (b_2, d_2), ..., (b_{k_1}, d_{k_1}))$ in der Überführungsfunktion δ_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:

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

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

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

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

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.

Man sagt, eine Klasse **K** von Sprachen, die jeweils über demselben Alphabet definiert sind, ist **gegenüber einer Operation** $\circ$ **abgeschlossen**, mit $L_1 \in \mathbf{K}$ und $L_2 \in \mathbf{K}$ auch $L_1 \circ L_2 \in \mathbf{K}$ gilt.

Obige Überlegungen und spätere Ausführungen in Kapitel 3.2 zeigen:

Satz 2.1-2:

Die Klasse der von Turingmaschinen akzeptierten Mengen über einem Alphabet Σ ist gegenüber Vereinigungsbildung und Schnittbildung abgeschlossen.

Die Klasse der von Turingmaschinen akzeptierten Mengen über einem Alphabet Σ 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 $TM = \left(Q, \Sigma, I, \delta, b, q_0, q_{accept} \right)$ und eine Eingabe $w \in L(TM)$ gelte
$(q_0, (w,1), (\varepsilon,1), ..., (\varepsilon,1)) \Rightarrow^m K_{accept}$
mit einer Endkonfiguration $K_{accept} = (q_{accept}, (\alpha_1, i_1), ..., (\alpha_k, i_k))$. Dann wird durch $t_{TM}(w) = m$ eine partielle Funktion $t_{TM} : \Sigma^* \to \mathbf{N}$ definiert, die angibt, **wie viele Überführungen *TM* macht, um *w* zu akzeptieren**. Es handelt sich hierbei um eine partielle Funktion, da t_{TM} nur für Wörter $w \in L(TM)$ 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
$$T_{TM}(n) = \max\{ t_{TM}(w) \mid w \in \Sigma^* \text{ und } |w| = n \}$$

Eine Turingmaschine *TM* heißt *f*(*n*)-**zeitbeschränkt** mit einer Funktion $f : \mathbf{N} \to \mathbf{N}$, wenn für ihre Zeitkomplexität T_{TM} die Abschätzung $T_{TM}(n) \in O(f(n))$ gilt.

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.

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 $n \geq 1$ $|w| = |bin(n)| = \lfloor \log_2(n) \rfloor + 1$ 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 $DIF: \begin{cases} \mathbf{N} \times \mathbf{N} & \to & \mathbf{N} \\ (n,m) & \to & \begin{cases} n-m & \text{für } n \geq m \\ 0 & \text{für } n < m \end{cases} \end{cases}$,

$MULT: \begin{cases} \mathbf{N} \times \mathbf{N} & \to & \mathbf{N} \\ (n,m) & \to & n \cdot m \end{cases}$ und $DIV: \begin{cases} \mathbf{N} \times \mathbf{N}_{>0} & \to & \mathbf{N} \\ (n,m) & \to & \lfloor n/m \rfloor \end{cases}$ 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, wie viele 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.

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 $T_{TM}(n)$ herleiten, d.h. man wird eine Aussage der Form $T_{TM}(n) \leq g(n)$ begründen. In diesem Fall ist $T_{TM}(n) \in O(g(n))$. Eine zusätzliche Aussage $T_{TM}(n) \leq h(n) \leq g(n)$ mit einer Funktion $h(n)$ führt auf die verbesserte Abschätzung $T_{TM}(n) \in O(h(n))$. Man ist also bei der worst case-Analyse 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.

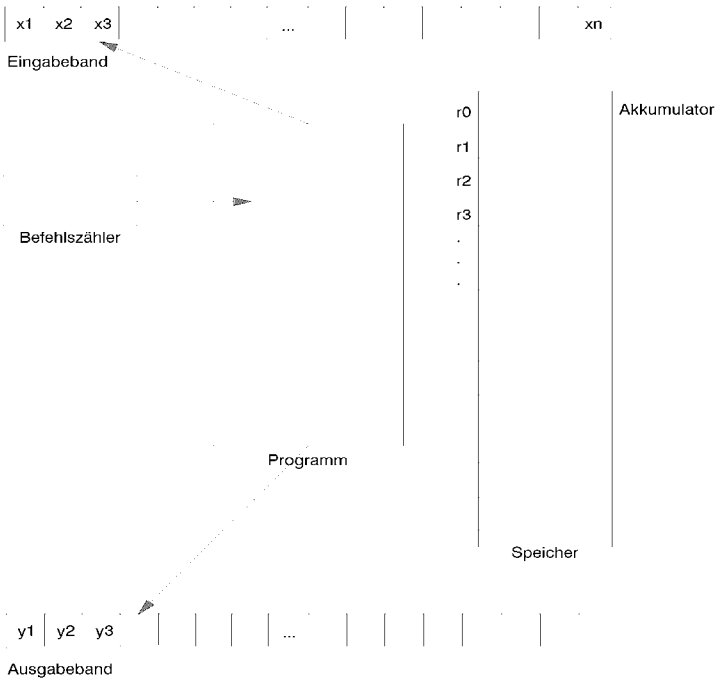
Es zeigt sich, dass alle Definitionen **algorithmisch äquivalent** sind, d.h. eine Turingmaschinenvariante kann eine andere Turingmaschinenvariante simulieren. Allerdings ändert sich dabei u.U. die Zeitkomplexität, in der von der jeweiligen Turingmaschinenvariante Sprachen erkannt werden. Beispielsweise 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, 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 $i \neq 0$ DIV $c(i)$ für $i \geq 0$ DIV $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := \lfloor c(0)/i \rfloor$ $c(0) := \lfloor c(0)/c(i) \rfloor$, falls $c(i) \neq 0$ ist, sonst stoppt die RAM $c(0) := \lfloor c(0)/c(c(i)) \rfloor$, falls $c(c(i)) \neq 0$ ist, sonst stoppt die RAM
READ $c(i)$ für $i \geq 0$ READ $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(i) :=$ momentaner Eingabewert , und der Lesekopf rückt eine Zelle nach rechts $c(c(i)) :=$ momentaner Eingabewert , und der Lesekopf rückt eine Zelle nach rechts
WRITE i WRITE $c(i)$ für $i \geq 0$ WRITE $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	i wird auf das Ausgabeband gedruckt, und der Schreibkopf rückt um eine Zelle nach rechts $c(i)$ wird auf das Ausgabeband gedruckt, und der Schreibkopf rückt um eine Zelle nach rechts $c(c(i))$ 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 $f: \begin{cases} \mathbf{N} & \rightarrow & \mathbf{N} \\ n & \rightarrow & \begin{cases} 1 & \text{für} \quad n = 0 \\ n^n & \text{für} \quad n \geq 1 \end{cases} \end{cases}$

Ein entsprechendes RAM-Programm lautet:

RAM-Anweisungsfolge		Bemerkung zur Bedeutung
Zeilennummer	RAM-Befehl	
1	READ $c(1)$	$r_1 := n$, Eingabe
2	LOAD $c(1)$	**IF** $r_1 \leq 0$ **THEN** *Write*(0)
3	JGTZ pos	
4	WRITE 0	
5	JUMP endif	
pos	LOAD $c(1)$	$r_2 := r_1$
7	STORE $c(2)$	
8	LOAD $c(1)$	$r_3 := r_1 - 1$
9	SUB 1	
10	STORE $c(3)$	
while	LOAD $c(3)$	**WHILE** $r_3 > 0$ **DO**
12	JGTZ continue	
13	JUMP endwhile	
continue	LOAD $c(2)$	$r_2 := r_2 * r_1$
15	MULT $c(1)$	
16	STORE $c(2)$	
17	LOAD $c(3)$	$r_3 := r_3 - 1$
18	SUB 1	
19	STORE $c(3)$	
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** ein Wort $w \in \Sigma^*$ auf folgende Weise. In die ersten *n* Zellen des Eingabebandes werden die Zahlen geschrieben, die den Buchstaben x_1, ..., x_n des Wortes entsprechenden ($w = x_1 \ldots x_n$). 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. Falls *RAM* mit einer Ausgabe ungleich 1 stoppt, oder *RAM* nicht stoppt, wird *w* nicht akzeptiert. Die auf diese Weise akzeptierten Wörter aus Σ^* bilden die Menge $L(RAM)$.

Ein RAM-Programm *P* zur Akzeptanz von

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

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**.

Es sei $\mathbf{Z}^*$ die Menge aller endlichen Folgen ganzer Zahlen, d.h. $\mathbf{Z}^* = \bigcup_{n \geq 0} \mathbf{Z}^n$.

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

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 ganze 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. 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 MULT $c(i)$ für $i \geq 0$ MULT $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := c(0) * i$ $c(0) := c(0) * c(i)$ $c(0) := c(0) * c(c(i))$	$l(c(0)) + l(i)$ $l(c(0)) + l(i) + l(c(i))$ $l(c(0)) + l(i) + l(c(i)) + l(c(c(i)))$
DIV i für $i \neq 0$ DIV $c(i)$ für $i \geq 0$ DIV $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := \lfloor c(0) / i \rfloor$ $c(0) := \lfloor c(0) / c(i) \rfloor$ $c(0) := \lfloor c(0) / c(c(i)) \rfloor$	$l(c(0)) + l(i)$ $l(c(0)) + l(i) + l(c(i))$ $l(c(0)) + l(i) + l(c(i)) + l(c(c(i)))$
READ $c(i)$ für $i \geq 0$ READ $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(i) :=$ Eingabewert $c(c(i)) :=$ Eingabewert	$l(\text{Eingabewert}) + l(i)$ $l(\text{Eingabewert}) + l(c(i))$
WRITE i WRITE $c(i)$ für $i \geq 0$ WRITE $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	Ausgabe von i Ausgabe von $c(i)$ Ausgabe von $c(c(i))$	$l(i)$ $l(i) + l(c(i))$ $l(i) + l(c(i)) + l(c(c(i)))$
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 (negative ganzzahlige Werte werden in Komplementdarstellung abgelegt; rationale Werte können als Paare ganzzahliger Werte nachgebildet werden; reelle Werte werden durch natürlichzahlige Werte approximiert, siehe 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.

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 $y \in \{\texttt{TRUE}, \texttt{FALSE}\}$. In der Regel wird die Zeit- und Raumkomplexität von *PROG* in Abhängigkeit von der Eingabe $x = [x_1, ..., x_n]$ gemessen.

In Anlehnung an die uniforme Zeitkomplexität im Random-Access-Maschinenmodell könnte man als Maß für die Zeitkomplexität die Anzahl an Anweisungen festlegen, die *PROG* bei Eingabe von *x* durchläuft, bis das Ergebnis *y* berechnet ist. Entsprechend könnte man zur Berechnung der Raumkomplexität von *PROG* bei Eingabe von *x* die Anzahl der benötigten Variablen zählen.

Dieser 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 und nicht von den Werten x_i selbst. 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 $size(x) = \sum_{i=1}^{n} (size(x_i) + 1) + 1$.

Der Wert $size(x)$ gibt damit die Anzahl der Bits an, um die Zahlenwerte darzustellen, die in *x* vorkommen, einschließlich der die Zahlenfolge $x_1, ..., x_n$ umfassenden eckigen Klammern und Trennsymbolen zwischen den Werten $x_1, ..., x_n$. Häufig wird auch als Größe der Eingabe der Wert $n \cdot (size(\max\{x_1, ..., x_n\}) + 1) + 1$ verwendet. Dieser Maßstab ist wohl etwas gröber als $size(x)$. Für Analysezwecke eignet er sich jedoch, da in einer Komplexitätsbetrachtung meist die Abschätzung $size(x) \in O(n \cdot size(\max\{x_1, ..., x_n\}))$ verwendet wird. Zudem gibt es Eingaben *x*, deren Zahlen in der Folge $x_1, ..., x_n$ zusammen genau $n \cdot size(\max\{x_1, ..., x_n\})$ viele Bits belegen.

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.

Die so definierte Zeitkomplexität eines Programms entspricht der für eine Turingmaschine definierten Zeitkomplexität, wenn bei der Turingmaschine die Eingabe in Form von Binärwerten auf das Eingabeband gegeben werden und die Anzahl der Arbeitsschritte der Turing-

maschine in Abhängigkeit von der Länge der Eingabe, d.h. der Anzahl benötigter Bits, gezählt wird.

In obiger Pascal-Prozedur `zweier_potenz` gilt für die Eingabe *n* und $k = size(n)$ die Beziehung $k \in O(\log(n))$. Die Anweisungen 1, 2 und 6 werden jeweils einmal durchlaufen. Die Wertzuweisung in Anweisung 1 ist von der Ordnung $O(\log(n))$, die Wertzuweisung in Anweisung 2 von der Ordnung $O(1)$ und 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

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

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

Da für die Bestimmung der Zeitkomplexität die Anzahl der Bitoperationen eine wesentliche Rolle spielt, werden diese (in Anlehnung an Satz 2.1-3) im folgenden **für die gängigen arithmetischen Operationen** zusammengefasst. Dabei erfolgt eine Beschränkung auf natürliche Zahlen; negative ganze Zahlen werden in Komplementdarstellung behandelt.

Es seien *x* und *y* natürliche Zahlen mit $x \ge y$, $k = size(x)$ und $l = size(y)$. Hier geben $size(x)$ bzw. $size(y)$ die Anzahl an Bits an, um *x* bzw. *y* im Binärsystem darzustellen. Die Binärdarstellungen seien $x = [x_{k-1}x_{k-2} \ldots x_1x_0]_2$ bzw. $y = [y_{l-1}y_{l-2} \ldots y_1y_0]_2$.

Addition $z := x + y$

Für $l \le k$ wird *y* durch Voranstellung von $k-l$ führende binäre Nullen auf dieselbe Länge wie *x* gebracht, so dass $y = [y_{k-1}y_{k-2} \ldots y_1y_0]_2$ ist. Das Ergebnis *z* besitzt eventuell eine Binärstelle mehr als *x*: $z = [z_k z_{k-1} \ldots z_1 z_0]_2$. Bei der bitweisen stellengerechten Addition werden für Stelle *i* die Bits x_i und y_i und ein eventueller Übertrag aus der Stelle $i-1$ addiert. Wird mit $u = [u_k u_{k-1} \ldots u_1 u_0]$ die Bitfolge der Überträge aus der jeweils vorherigen Bitaddition bezeichnet, so ist

$u_0 = 0$, $z_k = u_k$ und $z_i = x_i + y_i + u_i \pmod 2$ für $i = 0, ..., k-1$.

Die Addition „+“ lässt sich mit Hilfe der logischen Operationen $\wedge$, $\vee$ und $\oplus$ (Exklusiv-Oder-Operation (Addition modulo 2) ausdrücken: Hierzu werden die in Kapitel 1.1 definierten logischen Operationen $\wedge$, $\vee$, $\neg$ und $\oplus$ auf Variablen übertragen, die Werte aus $\mathbf{Z}/2\mathbf{Z}$ annehmen, indem man den Wahrheitswert `TRUE` (in der Belegung eines Booleschen Ausdrucks) mit dem numerischen Wert 1 und den Wahrheitswert `FALSE` mit dem numerischen Wert 0 gleichsetzt und die Tabellen für die arithmetischen Operationen entsprechend anpasst:

Werte von				
x	y	$(x \wedge y)$	$(x \vee y)$	$(x \oplus y)$
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Werte von	
x	$\neg x$
0	1
1	0

Wie man leicht nachprüft, ist die Operation $\oplus$ assoziativ, d.h. es gilt $(x \oplus y) \oplus z = x \oplus (y \oplus z)$, so dass man im Ausdruck $(x \oplus y) \oplus z$ die Klammern weglassen kann.

Es ist dann

$z_i = x_i \oplus y_i \oplus u_i$ und $u_{i+1} = (x_i \wedge y_i) \vee (u_i \wedge (x_i \vee y_i))$ für $i = 0, ..., k-1$.

Bei der Addition werden also $O(k) = O(\log(x))$ viele Bitoperationen ausgeführt.

Multiplikation $z := x \cdot y$

Das Ergebnis z besitzt eventuell doppelt so viele Binärstellen wie x: $z = [z_{2k-1} z_{2k-2} \ldots z_1 z_0]_2$.
Gemäß der „Schulmethode“ wird folgender (Pseudo-) Code ausgeführt:

```
z := 0;                          { Anweisung 1 }
FOR i := l – 1 DOWNTO 0 DO       { Anweisung 2 }
  BEGIN
    SHL (z, 1);                  { Anweisung 3 }
    IF y_i = 1 THEN z := z + x;  { Anweisung 4 }
  END;
```

Der Aufwand für Anweisung 1 ist von der Ordnung $O(1)$. Die `FOR`-Schleife (Anweisungen 2 bis 4) wird *l*-mal durchlaufen, wobei jeweils ein Aufwand der Ordnung $O(size(z))$ entsteht. Wegen $size(z) \le 2k$ und $l \le k$ werden zur Multiplikation $O(k^2) = O((\log(x)^2)$ viele Bitoperationen ausgeführt.

Ein schnelleres Multiplikationsverfahren beruht auf einer Idee von A. Karatsuba. Zur Vereinfachung der Darstellung sei *k* eine Zweierpotenz, d.h. $k = 2^m$ mit $m \in \mathbf{N}$.

Es ist $x = [x_{k-1}x_{k-2} \ldots x_1x_0]_2 = [x_{k-1}x_{k-2} \ldots x_{k/2+1}x_{k/2}]_2 \cdot 2^{k/2} + [x_{k/2-1}x_{k/2-2} \ldots x_1x_0]_2 = p \cdot 2^{k/2} + q$ mit den $k/2$-stelligen Binärzahlen $p = \lfloor x_{k-1}x_{k-2} \ldots x_{k/2+1}x_{k/2} \rfloor_2$ und $q = \lfloor x_{k/2-1}x_{k/2-2} \ldots x_1x_0 \rfloor_2$. Entsprechend ist
$y = [y_{k-1}y_{k-2} \ldots y_1y_0]_2 = [y_{k-1}y_{k-2} \ldots y_{k/2+1}y_{k/2}]_2 \cdot 2^{k/2} + [y_{k/2-1}y_{k/2-2} \ldots y_1y_0]_2 = r \cdot 2^{k/2} + s$ mit den $k/2$-stelligen Binärzahlen $r = \lfloor y_{k-1}y_{k-2} \ldots y_{k/2+1}y_{k/2} \rfloor_2$ und $s = \lfloor y_{k/2-1}y_{k/2-2} \ldots y_1y_0 \rfloor_2$. Es gilt

$$\begin{aligned} x \cdot y &= (p \cdot r) \cdot 2^k + (p \cdot s + q \cdot r) \cdot 2^{k/2} + q \cdot s \\ &= (p \cdot r) \cdot 2^k + (p \cdot r + q \cdot s - (q-p) \cdot (s-r)) \cdot 2^{k/2} + q \cdot s \ . \end{aligned}$$

Die Multiplikation zweier *k*-stelliger Zahlen *x* und *y* kann also auf drei Multiplikation von $k/2$-stelligen Zahlen, nämlich auf die Bildung der drei Produkte $p \cdot r$, $q \cdot s$ und $(q-p) \cdot (s-r)$, zurückgeführt werden (die Faktoren $(q-p)$ und $(s-r)$ sind ebenfalls $k/2$-stellig). Die Ergebnisse dieser Multiplikationen müssen dann noch zur Bildung der Summe $p \cdot r + q \cdot s - (q-p) \cdot (s-r)$ additiv verknüpft werden; die einzelnen Summanden sind dabei höchstens *k*-stellig, so dass hierbei ein Aufwand der Ordnung $O(k)$ entsteht. Die Teilergebnisse $p \cdot r$, $p \cdot r + q \cdot s - (q-p) \cdot (s-r)$ und $q \cdot s$ werden dann noch in einen Binärvektor der Länge $2k$ eingefügt. Die Multiplikation zweier *k*-stelliger Zahlen *x* und *y* wird also rekursiv auf drei Multiplikation von $k/2$-stelligen Zahlen (und einigen Additionen und Bitoperationen) zurückgeführt, wobei für $k = 1$ die Multiplikation direkt erfolgt.

Bezeichnet $M(k)$ die Anzahl erforderlicher Bitoperationen zur Multiplikation zweier *k*-stelliger Zahlen, so gilt bei diesem Verfahren
$M(k) \le 3 \cdot M(k/2) + c \cdot k$ mit einer Konstanten $c > 0$.
Der Summand $c \cdot k$ resultiert aus der Bildung der Summe $p \cdot r + q \cdot s - (q-p) \cdot (s-r)$, die aus drei Summanden mit maximal *k* Bits entsteht. Außerdem ist $M(1) = 1$. Mit $k = 2^m$ ergibt sich damit
$M(2^m) \le 3 \cdot M(2^{m-1}) + c \cdot 2^m$ für $m \ge 1$ und $M(1) = 1$.

Die allgemeine Form dieser Rekursionsgleichung lautet

$$M\left(2^m\right) \le 3^l \cdot M\left(2^{m-l}\right) + c \cdot \sum_{i=0}^{l-1} 3^i \cdot 2^{m-i} .$$

Mit $l = m$ ergibt sich

$$M\left(2^m\right) \le 3^m + c \cdot \sum_{i=0}^{m-1} 3^i \cdot 2^{m-i} = 3^m + c \cdot 2^m \cdot \sum_{i=0}^{m-1} \left(3/2\right)^i = 3^m + 2 \cdot c \cdot \left(3^m - 2^m\right) \le d \cdot \left(3^m - 2^m\right)$$

mit einer Konstanten $d > 0$. Insgesamt:

$M(k) = M\left(2^m\right) \le d \cdot \left(3^m - 2^m\right) = d \cdot \left(\left(2^m\right)^{\log_2(3)} - 2^m\right) = d \cdot \left(k^{1{,}58\ldots} - k\right)$, d.h. $M(k) \in O\left(k^{1{,}58\ldots}\right)$.

Gegenüber der Schulmethode zur Multiplikation mit der Komplexität $O\left(k^2\right)$ stellt diese Methode also eine Verbesserung dar, die sich besonders bei der Multiplikation von Zahlen mit sehr vielen Stellen zeigt.

Durch Aufteilung der Zahlen *x* und *y* in mehr als zwei kleinere Teile und Anwendung sorgfältig entworfener Verfahren zur Multiplikation der entstandenen Teile erhält man ein Multiplikationsverfahren (Toom-Cook-Verfahren), das zur Multiplikation zweier *k*-stelliger Zahlen eine Komplexität der Ordnung $O\left(c(\varepsilon) \cdot k^{1+\varepsilon}\right)$ besitzt; hierbei hängt ε von der Anzahl Aufteilungen ab, und $c(\varepsilon)$ ist eine von *k* unabhängige Konstante. Dieses Verfahren ist jedoch eher von theoretischem als von praktischen Interesse, da es schwierig zu implementieren ist und erst für Zahlen mit einer sehr großen Stellenzahl überhaupt eine signifikante Verbesserung gegenüber der Karatsuba-Methode aufweist.

Bemerkung: Der schnellste bekannte Multiplikationsalgorithmus benötigt unter Einsatz einer Variante der diskreten Fouriertransformation eine Anzahl an Bitmultiplikationen der Ordnung $O\left(k \cdot \log(k) \cdot \log(\log(k))\right)$.

Ganzzahlige Division $z := \lfloor x/y \rfloor$ für $y \neq 0$

Die Berechnung von $\lfloor x/y \rfloor$ erfolgt durch sukzessive Additionen:

```
IF y <= x THEN BEGIN
                  z := -1;
                  w := 0;
                  WHILE w <= x DO
                    BEGIN
                      z := z + 1;
                      w := w + y;
                    END

                END
            ELSE z := 0;
```

Offensichtlich ist $z = \lfloor x/y \rfloor$. Die `WHILE`-Schleife wird $O(x/y)$-mal durchlaufen, wobei wegen $w \leq x$, $y \leq x$ und $z \leq x$ jeweils die Anzahl der Bitoperationen durch eine Größe der Ordnung $O(k) = O(size(x))$ begrenzt ist. Insgesamt ist die Anzahl der Bitoperationen nach diesem Verfahren von der Ordnung $O\left(k \cdot x/y\right)$, d.h. im ungünstigsten Fall, nämlich für kleine Werte von *y*, exponentiell von der Ordnung $O\left(k \cdot 2^k\right)$.

Der folgende Ansatz bringt eine Laufzeitverbesserung:

```
IF y <= x THEN BEGIN
                  v := x;
                  z := 0;
                  WHILE v >= y DO          { Schleife 1 }
                    BEGIN
                      w := y;
                      u := 0;
                      WHILE 2*w <= v DO    { Schleife 2 }
                        BEGIN
                           w := SHL (w, 1);  { w := 2 * w }
                           u := u + 1;
                        END;
                      z := z + SHL (1, u);
                      v := v – w;
                    END;
                END
             ELSE z := 0;
```

Da der Ablauf des Verfahrens nicht ganz offensichtlich ist und um das Verständnis des nachfolgenden Korrektheitsbeweises zu fördern, wird er an einem Beispiel erläutert: Bei der Berechnung von $\lfloor 117/5 \rfloor$ nehmen die Variablen nacheinander folgende Werte an:

x	y	v	w	u	z
117	5	117	5	0	0
			10	1	
			20	2	
			40	3	
			80	4	$2^4 = 16$
		37	5	0	
			10	1	
			20	2	$16 + 2^2 = 20$
		17	5	0	
			10	1	$20 + 2^1 = 22$
		7	5	0	$22 + 2^0 = 23$
		2			

Die Korrektheit des Verfahrens sieht man wie folgt:

Schleife 1 werde *t*-mal durchlaufen. Die Werte der Variablen u am Ende der Schleife 2 seien $i_1, ..., i_t$. Dann gilt:

$2^{i_1} \cdot y \le x$ und $2^{i_1+1} \cdot y > x$,

$2^{i_2} \cdot y \le x - 2^{i_1} \cdot y$ und $2^{i_2+1} \cdot y > x - 2^{i_1} \cdot y$,

$2^{i_3} \cdot y \le x - 2^{i_1} \cdot y - 2^{i_2} \cdot y$ und $2^{i_3+1} \cdot y > x - 2^{i_1} \cdot y - 2^{i_2} \cdot y$,

...,

$2^{i_t} \cdot y \le x - 2^{i_1} \cdot y - 2^{i_2} \cdot y - ... - 2^{i_{t-1}} \cdot y$ und $2^{i_t+1} \cdot y > x - 2^{i_1} \cdot y - 2^{i_2} \cdot y - ... - 2^{i_{t-1}} \cdot y$.

Daraus folgt $i_1 > i_2 > ... i_{t-1} > i_t$, $\left(2^{i_1} + 2^{i_2} + 2^{i_3} + ... + 2^{i_t}\right) \cdot y \le x$ und

$\lfloor x/y \rfloor = 2^{i_1} + 2^{i_2} + 2^{i_3} + ... + 2^{i_t}$.

Es ist $t \in O(\log(\lfloor x/y \rfloor))$. Der Aufwand an Bitoperationen für einen Durchlauf der gesamten Schleife 2 kann durch einen Wert der Größenordnung $O(i_1 \cdot (\log(x) + \log(\lfloor x/y \rfloor)))$ abgeschätzt werden. Insgesamt ist damit die Anzahl an Bitoperationen beschränkt durch einen Ausdruck der Ordnung

$$O(t \cdot (\log(y) + i_1 \cdot (\log(x) + \log(\lfloor x/y \rfloor)) + \log(\lfloor x/y \rfloor) + \log(x))) \subseteq O((\log(\lfloor x/y \rfloor))^3 \subseteq O((\log(x))^3)).$$

Die Anzahl der erforderlichen Bitoperationen zur Bildung von $z = \lfloor x/y \rfloor$ lässt sich jedoch durch Anwendung „subtilerer“ Verfahren auf die Anzahl der Bitoperationen bei der Multiplikation zurückführen[3], so dass die ganzzahlige Division ebenfalls mit einem Aufwand an Bitoperationen von der Ordnung $O(k^2) = O((\log(x)^2)$ bzw. sogar von der Ordnung $O(k \cdot \log(k) \cdot \log(\log(k)))$ durchgeführt werden kann.

Modulo-Operation $z := x \bmod y$

Wegen $x \bmod y = x - y \cdot \lfloor x/y \rfloor$ ist der Aufwand an Bitoperationen bei der Modulo-Operation von derselben Größenordnung wie die Multiplikation.

3 Das Verfahren führt die Division auf die Multiplikation mit reziproken Werten zurück: $x/y = x \cdot (1/y)$ und $1/y$ kann mittels des Newtonverfahrens sehr schnell durch die Folge $(a_n)_{n \in \mathbf{N}}$ mit $a_{n+1} = a_n \cdot (2 - y \cdot a_n)$ approximiert werden. Siehe Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.: **The Design and Analysis of Computer Algorithms**, Addison-Wesley, 1974.

Exponentiation $z := x^y$ **und modulare Exponentiation** $z := x^y \bmod n$

Wegen $y = [y_{l-1} y_{l-2} \ldots y_1 y_0]_2$ ist

$$x^y = x^{y_{l-1}\cdot 2^{l-1} + y_{l-21}\cdot 2^{l-2} + \ldots + y_1 \cdot 2 + y_0} = \prod_{i=0}^{l-1} x^{y_i \cdot 2^i} = \prod_{i=0}^{l-1} \left(x^{2^i}\right)^{y_i}.$$

Man könnte zur Berechnung von x^y zunächst l viele Zahlen e_i ($i = l - 1, \ldots, 1, 0$) bestimmen, die durch $e_i = \begin{cases} x^{2^i} & \text{für } y_i = 1 \\ 1 & \text{für } y_i = 0 \end{cases}$ definiert sind. Diese Zahlen werden dann zusammenmultipliziert. Die Berechnung einer Zahl $e_i = x^{2^i}$ könnte durch sukzessives Quadrieren erfolgen:

```
c := x;
FOR idx := 1 TO  i DO c := c*c;
e_i := c;
```

Hierzu sind für $i = l - 1, \ldots, 1, 0$ jeweils maximal i viele Multiplikationen, also insgesamt maximal $l(l-1)/2$ viele Multiplikationen, erforderlich. Anschließend müssten die l vielen Zahlen e_i in $l - 1$ vielen Multiplikationen zusammenmultipliziert werden. Die Anzahl der Multiplikationen beträgt bei diesem Ansatz maximal $l(l-1)/2 + l - 1 = l(l+1)/2 - 1$, ist also von der Ordnung $O(l^2)$. Die Berechnung und das Zusammenmultiplizieren aller Werte e_i kann gleichzeitig erfolgen:

```
c := x;
FOR i := l - 1 DOWNTO 1 DO
   BEGIN
     c := c*c;
     IF  y_{i-1} = 1 DO c := c * x;
   END;
```

Der abschließende Wert der Variablen c ist x^y.

Auch hier soll der Ablauf an einem Beispiel gezeigt werden. Bei der Berechnung von x^{21} nehmen die Variablen nacheinander folgende Werte an ($y = 21_{10} = [10101]_2$):

i	c	y_{i-1}
	x	
4	x^2	0
3	x^4	1
	x^5	
2	x^{10}	0
1	x^{20}	1
	x^{21}	

Offensichtlich kommt man mit $2(l-1)$, d.h. mit $O(l) = O(\log(y))$ vielen Multiplikationen aus. Belegt die größte bei dieser Berechnung auftretende Zahl höchstens *h* Binärstellen, so kann x^y mit höchstens $O(h^2 \cdot \log(y))$ vielen Bitoperationen berechnet werden.

Das Verfahren kann auf die Modulo-Arithmetik übertragen werden. Zur Berechnung von $x^y \bmod n$ wird dabei in obigem Verfahren jede Multiplikation `c := c*c;` bzw. `c := c * x;` durch $c := c \cdot c \,(\bmod n)$; bzw. $c := c \cdot x \,(\bmod n)$; ersetzt. Alle bei der Berechnung auftretenden Zahlen belegen dann höchstens $h = \lfloor \log_2(n+1) \rfloor$ viele Binärstellen. Daher werden zur Berechnung von $x^y \bmod n$ höchstens $O((\log(n))^2 \cdot \log(y))$ viele Bitoperationen benötigt.

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 Be-

schreibung einer Turingmaschine *TM* (ein Problemlösungsalgorithmus) und einen Eingabedatensatz *u* für *TM*. Die universelle Turingmaschine verhält sich dann genauso, wie sich *TM* bei Eingabe von *u* verhalten würde.

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

Eine deterministische *k*-Band-Turingmaschine $TM = \left(Q, \Sigma, I, \delta, b, q_0, q_{accept} \right)$ 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 $Q = \left\{ q_0, .., q_{|Q|-1} \right\}$, ihr Arbeitsalphabet $\Sigma = \left\{ a_0, .., a_{|\Sigma|-1} \right\}$. 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 $w_0 \in \left\{ 0, 1, \# \right\}^*$ festgehalten:
$w_0 = \#bin(|Q|)\#bin(|\Sigma|)\#bin(k)\#\#$.

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

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

Die gesamte Turingmaschine *TM* lässt sich durch Konkatination w_{TM} der Wörter w_0, w_1, ..., w_d kodieren: $w_{TM} = w_0 w_1 \ldots w_d$. Es ist $w_{TM} \in \left\{ 0, 1, (,), \#, L, R, S \right\}^*$. 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 $TM = (Q, \Sigma, I, \delta, b, q_0, q_{accept})$ mit $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1, b\}$, $I = \{0, 1\}$, $q_{accept} = q_2$ und δ gemäß der folgenden Tabelle:

Überführungsfunktion $\delta(q_i, a_{i_1}, ..., a_{i_k}) = (q_j, (a_{j_1}, d_1), ..., (a_{j_k}, d_k))$	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}
$\delta(q_0, 1) = (q_1, (0, R))$	$w_1 = (0\#1)(1\#(0\#R))\#$ 0100001000010110100011000100001001100110111100
$\delta(q_1, 0) = (q_0, (1, R))$	$w_2 = (1\#0)(0\#(1\#R))\#$ 0100011000000110100001000100011001100110111100
$\delta(q_1, 1) = (q_2, (0, R))$	$w_3 = (1\#1)(10\#(0\#R))\#$ 0100011000010110100010001000100001001100110111100
$\delta(q_1, b) = (q_1, (1, L))$	$w_4 = (0\#11)(1\#(1\#L))\#$ 0100011000010010110100011000100011001010110111100

Das Wort w_0 lautet: $w_0 = \#11\#11\#1\#\#$. Dieses Wort wird übersetzt in eine 0-1-Kodierung und ergibt 1000010011000010011001100100.

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 $TM_{min} = \left(Q, \Sigma, I, \delta, b, q_0, q_{accept} \right)$ mit $k = 0$, $Q = \{ q_0 \}$, $\Sigma = \{ b \}$ und $\delta = \varnothing$. Das dieser Turingmaschine entsprechende Wort w_0 lautet: $w_0 =$#1#1#0##, übersetzt in eine 0-1-Kodierung 100001100001100000100100. Da $\delta = \varnothing$ ist, stimmt $code\left(TM_{min}\right)$ 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. Weiterhin ist es möglich, dass verschiedene Worte $w_1 \in \{ 0,1 \}^*$ und $w_2 \in \{ 0,1 \}^*$ unterschiedliche Turingmaschinen kodieren, die dieselbe Sprache akzeptieren. 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 Arbeitsal-

phabets 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) $VERIFIZIERE\_TM(w) = \mathsf{TRUE}$

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

Falls $w = code(TM)$ 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` in Pseudocode lautet:

```
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:

Die Turingmaschine *UTM* wird informell durch ihre Arbeitsweise beschrieben:

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[4].

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 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 $K_0 = \left(q_0, \underbrace{(u,1),(\varepsilon,1),...,(\varepsilon,1)}_{k} \right)$. Hierbei kann ein ähnlicher Umsetzungsmechanismus wie zur Erzeugung der Kodierung einer Turingmaschine verwendet werden. Eine Konfiguration von K_w der Form $K = (q_t, (\alpha_1, i_1), ..., (\alpha_k, i_k))$, wobei q_t der *t*-te Zustand von K_w und $\alpha_j \in \Sigma^*$, $i_j \geq 1$ für *j* = 1, ..., *k* ist, wird zunächst (gedanklich) umgesetzt in $(bin(t)\#(\beta_1\#bin(i_1))\#...\#(\beta_k\#bin(i_k)))$. Hierbei erhält man β_j aus α_j (für *j* = 1, ..., *k*), indem man jeden Buchstaben durch den Binärwert seiner Nummer in Σ, gefolgt vom Zeichen # ersetzt. Ist beispielsweise $\alpha_j = aacca$ und sind *a* bzw. *c* die Zeichen in Σ mit den Nummern 1 bzw. 3, so ist $\beta_j = 1\#1\#11\#11\#1\#$. Die Zeichenfolge $(bin(t)\#(\beta_1\#bin(i_1))\#...\#(\beta_k\#bin(i_k)))$ wird mittels anfangs angegebener Tabelle in eine Folge über $\{0,1\}$ umgesetzt.

---

[4] 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.

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 $u = 11001$:
$K_0 = (q_0, (11001, 1), (\varepsilon, 1), (\varepsilon, 1), (\varepsilon, 1))$ bzw.
$(0\#(10\#10\#1\#1\#10\#\#1)\#(0\#\#1)\#(0\#\#1))$ und in eine 0-1-Folge kodiert:
010000100010001000100001000100001100001100001000100100001011100010000100100
001011100010000100100001011011.

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, durch den eindeutigen Wert $\delta(q, a_1, ..., a_k)$ der Überführungsfunktion und bezüglich der Positionierung der Schreib/Leseköpfe durch deren gegenwärtige Positionen bestimmt ist. In diesem Kapitel wird das Modell des Nichtdeterminismus in Form einer nichtdeterministischen Turingmaschine und eines nichtdeterministischen Algorithmus eingeführt.

Eine **nichtdeterministische *k*-Band-Turingmaschine (*k*-NDTM)** *TM* ist definiert durch
$TM = \left(Q, \Sigma, I, \delta, b, q_0, q_{accept} \right)$
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. $\delta : (Q \setminus \{q_{accept}\}) \times \Sigma^k \rightarrow \mathbf{P}\left(Q \times \left(\Sigma \times \{L, R, S\}\right)^k\right)$ ist eine partielle Funktion, die **Überführungsfunktion**; insbesondere ist $\delta(q, a_1, \ldots, a_k)$ für $q = q_{accept}$ nicht definiert; zu beachten ist, dass δ für einige weitere Argumente eventuell nicht definiert ist.

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

Damit muss auch die Form 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:
$K = (q, (\alpha_1, i_1), \ldots, (\alpha_k, i_k))$ mit $q \in Q$, $\alpha_j \in \Sigma^*$, $i_j \geq 1$ für $j = 1, \ldots, k$.

TM startet wie im deterministischen Fall im Anfangszustand mit einer **Anfangskonfiguration** $K_0 = (q_0, (w,1), (\varepsilon,1), \ldots, (\varepsilon,1))$. Ist *TM* in eine Konfiguration $K = (q, (\alpha_1, i_1), \ldots, (\alpha_k, i_k))$ gekommen und ist $\delta(q, a_1, \ldots, a_k)$ definiert, dann besteht $\delta(q, a_1, \ldots, a_k)$ aus einer endlichen Menge von Werten der Form $(q', (b_1, d_1), \ldots, (b_k, d_k))$, d.h.
$\delta(q, a_1, \ldots, a_k) = \{(q_1, (b_{11}, d_{11}), \ldots, (b_{1k}, d_{1k})), \ldots, (q_t, (b_{t1}, d_{t1}), \ldots, (b_{tk}, d_{tk}))\}$.
Als **Folgekonfiguration** von *K* wird eine der *t* möglichen Konfigurationen
$K_1 = (q_1, (\beta_{11}, i'_{11}), \ldots, (\beta_{1k}, i'_{1k}))$, ..., $K_t = (q_t, (\beta_{t1}, i'_{t1}), \ldots, (\beta_{tk}, i'_{tk}))$
genommen. K_i für $i = 1, \ldots, t$ entsteht aus *K* dadurch, dass man wie im deterministischen Fall q durch q_i und $a_1, \ldots, a_k$ durch $b_{i1}, \ldots, b_{ik}$ ersetzt und die Köpfe so bewegt, wie es

$(q_i,(b_{i1},d_{i1}),...,(b_{ik},d_{ik}))$ in $\delta(q,a_1,...,a_k)$ 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
$K_{accept} = (q_{accept},(\alpha_1,i_1),...,(\alpha_k,i_k))$,
akzeptierende Konfiguration (**Endkonfiguration)**.

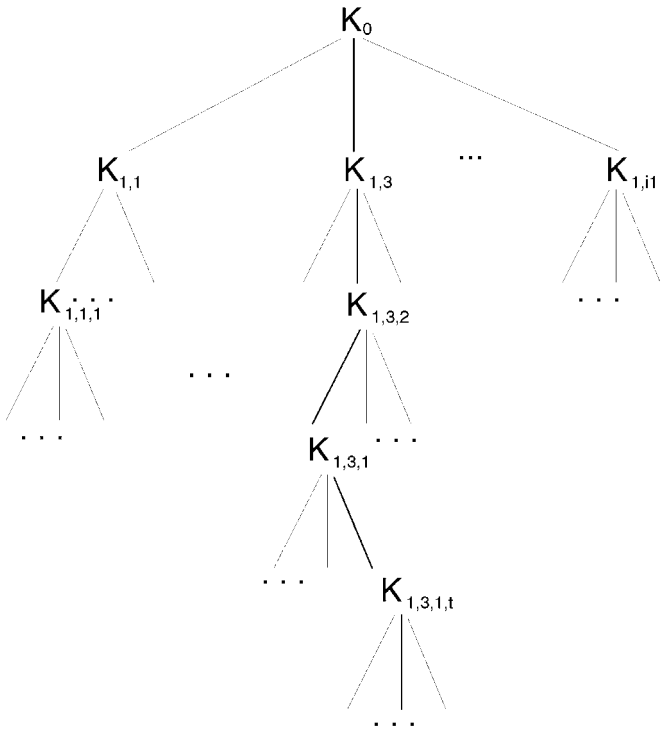
Wie im deterministischen Fall schreibt man $K \Rightarrow^m K'$ mit $m \in \mathbf{N}$, wenn K' aus *K* durch *m* Konfigurationsänderungen hervorgegangen ist, d.h. wenn es *m* Konfigurationen K_1, ..., K_m gibt mit
$K \Rightarrow K_1 \Rightarrow ... \Rightarrow K_m = K'$.
Für $m = 0$ ist dabei $K = K'$.

Die von einer *k*-NDTM *TM* **akzeptierte Sprache** ist die Menge
$L(TM) = \{ w \mid w \in I^* \text{ und } w \text{ wird von } TM \text{ akzeptiert} \}$
$= \{ w \mid w \in I^* \text{ und } (q_0,(w,1),(\varepsilon,1),...,(\varepsilon,1)) \Rightarrow^* (q_{accept},(\alpha_1,i_1),...,(\alpha_k,i_k)) \}$.

Wie im deterministischen Fall kann man die Überführungsfunktion δ modifizieren, indem man die Zustandsmenge *Q* um einen neuen Zustand q_{reject} erweitert und δ um entsprechende Zeilen ergänzt (siehe Kapitel 2.1), so dass alle Berechnungen, die in einem Zustand $q \neq q_{accept}$ enden, für die $\delta(q,a_1,...,a_k)$ mit $(a_1,...,a_k) \in \Sigma^k$ bisher nicht definiert ist, um eine Überführung in den Zustand q_{reject} fortgesetzt werden können. Für q_{reject} ist δ 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 δ 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 $K_0 = (q_0,(w,1),(\varepsilon,1),...,(\varepsilon,1))$ mit einem Wort $w \in I^*$ markiert. Ist die Konfiguration $K = (q,(\alpha_1,i_1),...,(\alpha_k,i_k))$ die Markierung eines Knotens und kann hier der Eintrag $\delta(q,a_1,...,a_k)$ der Überführungsfunktion angewendet werden, etwa

$\delta(q,a_1,\ldots,a_k)=\{(q_1,(b_{11},d_{11}),\ldots,(b_{1k},d_{1k})),\ldots,(q_t,(b_{t1},d_{t1}),\ldots,(b_{tk},d_{tk}))\}$, dann enthält dieser Knoten des Berechnungsbaums *t* direkte Nachfolger, die mit den sich ergebenden Nachfolgekonfigurationen $K_1=(q_1,(\beta_{11},i'_{11}),\ldots,(\beta_{1k},i'_{1k}))$, ..., $K_t=(q_t,(\beta_{t1},i'_{t1}),\ldots,(\beta_{tk},i'_{tk}))$ 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: $I=\{a_1,\ldots,a_n\}$

I ist eine Menge von *n* natürlichen Zahlen.

Lösung: Entscheidung „ja“, falls es eine Teilmenge $J\subseteq\{1,\ldots,n\}$ 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 $I=\{a_1,\ldots,a_n\}$ 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 $TM = (\{q_0, ..., q_5\}, \{0,1,b,\$\}, \{0,1\}, \delta, b, q_0, q_5)$ 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 δ 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

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

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). Bei $n = m$ 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

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

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 Arbeitsweise der Turingmaschine in diesem Beispiel ist typisch für nichtdeterministisches Verhalten. Im allgemeinen enthält die Überführungsfunktion einer nichtdeterministischen Turingmaschine *TM* deterministische und nichtdeterministische Teile, d.h. Einträge der Form

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

mit $t = 1$ (deterministisch) und Teile mit $t > 1$ (nichtdeterministisch). Soll in einer Konfiguration der Eintrag $\delta(q, a_1, \ldots, a_k)$ angewendet werden und ist $t = 1$, so ist die Nachfolgekonfiguration wie bei einer deterministischen Turingmaschine eindeutig bestimmt. Ist $t > 1$, so wird einer der Einträge $(q_i, (b_{i1}, d_{i1}), \ldots, (b_{ik}, d_{ik}))$ mit $1 \leq i \leq t$ ausgewählt, um die Nachfolgekonfiguration zu bilden. Dabei wird in dieser Situation ein „richtiger" Eintrag $(q_i, (b_{i1}, d_{i1}), \ldots, (b_{ik}, d_{ik}))$ ausgewählt, nämlich ein Eintrag, der die Berechnung schließlich in eine akzeptierende Endkonfiguration führt, falls das Eingabewort aus $L(TM)$ ist. Ist das Eingabewort nicht aus $L(TM)$, so wird auch ein Eintrag $(q_i, (b_{i1}, d_{i1}), \ldots, (b_{ik}, d_{ik}))$ ausgewählt, es ist jedoch gleichgültig, welche Alternative gewählt wird: in keinem Fall wird die Berechnung in einer akzeptierenden Endkonfiguration enden.

Es kann hilfreich sein, sich diese nichtdeterministische Auswahl auch folgendermaßen vorzustellen: bei der Auswahl einer der Alternativen $(q_i, (b_{i1}, d_{i1}), \ldots, (b_{ik}, d_{ik}))$ mit $1 \leq i \leq t$ bemüht

man eine „externe Instanz“, die eine Zusatzinformation liefert, aus der man schließen kann welche Alternative zu wählen ist (die Befragung die externen Instanz und die Auswertung der Antwort ist nicht Teil der Berechnung)[5]. Die externe Instanz liefert diese Zusatzinformation in jedem Fall, ob das Eingabewort nun aus $L(TM)$ ist oder nicht. Ist das Eingabewort aus $L(TM)$, so wird hier eine Zusatzinformation geliefert, die in diesem Schritt zu einer Folgekonfiguration führt, die sich zu einer akzeptierenden Berechnung fortsetzen lässt. Die externe Instanz ist also „verlässlich“. Ist das Eingabewort nicht aus $L(TM)$, so würde auch eine andere Antwort der externen Instanz in dieser Situation nicht zu einer Folgekonfiguration führen, die sich zu einer akzeptierenden Berechnung fortsetzen ließe. In jedem Fall muss die Berechnung fortgesetzt werden, d.h. die Zusatzinformationen müssen verifiziert werden.

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

Für eine *k*-NDTM $TM = \left(Q, \Sigma, I, \delta, b, q_0, q_{accept} \right)$ und eine Eingabe $w \in L(TM)$ gelte
$(q_0, (w,1), (\varepsilon,1), ..., (\varepsilon,1)) \Rightarrow^m K_{accept}$
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, wie viele 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
$T_{TM}(n) = \max\{ t_{TM}(w) \mid w \in \Sigma^* \text{ und } |w| \le n \}$.

Die Turingmaschine *TM* heißt *f*(*n*)-**zeitbeschränkt** mit einer Funktion $f : \mathbf{N} \to \mathbf{N}$, wenn für ihre Zeitkomplexität T_{TM} die Abschätzung $T_{TM}(n) \in O(f(n))$ gilt.

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.

[5] Diese „externe Instanz” wird hier nicht als Orakel bezeichnet, um keine Verwechslung mit Orakel-Turingmaschinen bervorzurufen.

Beispielsweise ist die oben angegebene Turingmaschine zur Lösung des Partitionenproblems $(2n + 2)$-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 $T(n) \geq n$ 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, d.h. $L = L(TM)$, dann gibt es eine k'-DTM TM' mit $L(TM') = L$. Man kann TM' so konstruieren, dass gilt:
Falls TM $T(n)$-zeitbeschränkt mit einer zeitkonstruierbaren Funktion $T(n) \geq n$ ist, dann ist TM' $O(T(n) \cdot d^{T(n)})$-zeitbeschränkt mit einer Konstanten $d \geq 2$.

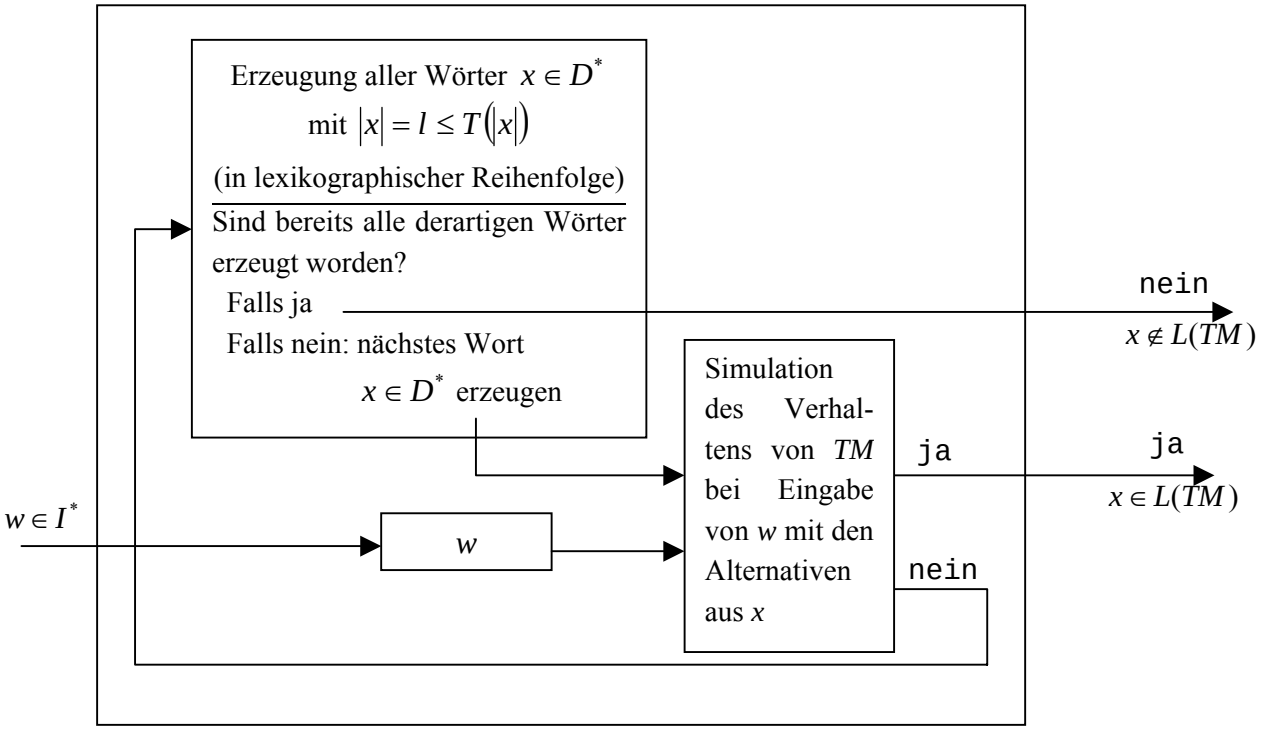
Beweis:

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 δ von TM gegeben wird (Überführungstabelle), hat die Form
$\delta(q, a_1, \ldots, a_k) = \{ (q_1, (b_{11}, d_{11}), \ldots, (b_{1k}, d_{1k})), \ldots, (q_t, (b_{t1}, d_{t1}), \ldots, (b_{tk}, d_{tk})) \}$
(hier stehen t „Teilzeilen“). Wenn in einer Konfigurationenfolge $\delta(q, a_1, \ldots, a_k)$ 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 $D = \{1, ..., d\}$. Dann kann man eine von *TM* ausgeführte Berechnung bzw. die dabei durchlaufene Konfigurationsfolge der Länge *l* durch ein Wort $x = d_1 \ldots d_l$ über *D* beschreiben: d_i gibt an, dass in der *i*-ten Überführung die d_i-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 $|x| = l \leq T(n)$, $x = d_1 \ldots d_l$, 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-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 $|x| = l \le T(n)$ 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

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

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

///

Diese 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 $c \cdot T(n) \cdot d^{T(n)} \le c \cdot d^{T(n)} \cdot d^{T(n)} = c \cdot d^{2T(n)} = c \cdot \left(d^2\right)^{T(n)}$, d.h. man kann $C = d^2$ nehmen.

Bisher ist keine nicht-exponentiell wachsende untere Zeitschranke 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).

Der folgende Satz beschreibt den Zusammenhang von Nichtdeterminismus und Determinismus.

Satz 2.5-2:

Es sei $T : \mathbf{N} \to \mathbf{N}$ eine zeitkonstruierbare Funktion mit $T(n) \geq n$. Ferner seinen Σ und Σ_0 endliche Alphabete. Dann gilt:

(i) Wird $L \subseteq \Sigma^*$ von einer $T(n)$-zeitbeschränkten *k*-NDTM *TM* akzeptiert, d.h. $L = L(TM)$, dann gibt es eine (von *TM* abhängige) Konstante $c > 0$ und eine k'-DTM TM' mit
$L = \{ w \mid w \in \Sigma^* \text{ und es gibt } x \in \{0,1\}^* \text{ mit } |x| \leq c \cdot T(|w|) \text{ und } (w, x) \in L(TM') \}$

(ii) Es sei $c > 0$ eine Konstante. Die Menge $L' \subseteq \{ (w, y) \mid w \in \Sigma^*, y \in \Sigma_0^*, |y| \leq c \cdot T(|w|) \}$ werde von einer k'-DTM TM' akzeptiert, die für jede Eingabe (w, y) mit $w \in \Sigma^*$ und $|y| \leq c \cdot T(|w|)$ höchstens $T_1(|w|)$ viele Schritte mit einer Funktion $T_1 : \mathbf{N} \to \mathbf{N}$ benötigt. Dann gibt es eine $(T(n) + T_1(n))$-zeitbeschränkte *k*-NDTM *TM* mit
$L(TM) = \{ w \mid w \in \Sigma^* \text{ und es gibt } y \in \Sigma_0^* \text{ mit } |y| \leq c \cdot T(|w|) \text{ und } (w, y) \in L' \}$.

Bemerkung: Aussage (ii) ist eine Art Umkehrung von Aussage (i), wobei in (ii) die allgemeinere Formulierung mit einem beliebigen Alphabet Σ_0 anstelle von $\{0,1\}$ und die Zeitkomplexität „vorsichtiger" gewählt wurden.

Beweis:

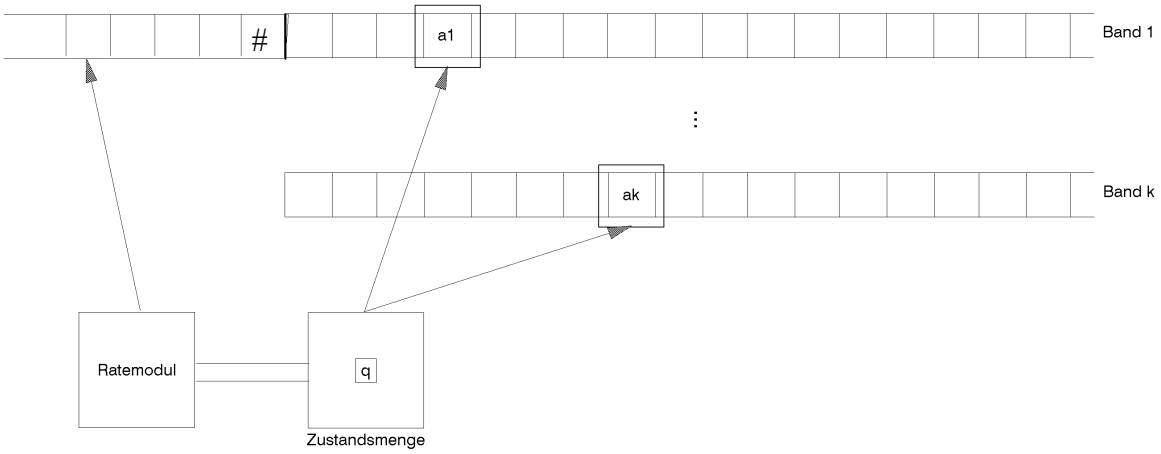
Zu (i): Der Beweis ist im wesentlichen die Konstruktion des Beweises von Satz 2.5-1. Kodiert man dort die Elemente der Menge $D = \{1, ..., d\}$ als Binärwerte, so ist $c = \lfloor \log_2(d) \rfloor + 1$. TM' ist die deterministische Simulation des Verhaltens von *TM* bei Eingabe von *w* mit den in *x* beschriebenen Alternativen.

Zu (ii): Die im Satz anzugebende nichtdeterministische Turingmaschine *TM* wird durch ihr Verhalten informell beschrieben:
Bei Eingabe eines Worts $w \in \Sigma^*$ berechnet *TM* den Wert $c \cdot T(|w|)$ und erzeugt auf nichtdeterministische Weise ein Wort $y \in \Sigma_0^*$ mit $|y| \leq c \cdot T(|w|)$. Die Anzahl der ausgeführten Berechnungsschritte ist von der Ordnung $O(T(|w|))$. Anschließend simuliert *TM* das Verhalten von TM' bei Eingabe (w, y) und akzeptiert genau dann, wenn TM' akzeptiert. Hierbei werden höchstens $T_1(|w|)$ viele Schritte ausgeführt.

///

Die Aussage in Satz 2.5-2 (i) legt eine Modellvorstellung für nichtdeterministische Turingmaschinen nahe, in der bei Eingabe eines Worts $w \in \Sigma^*$ zunächst in nichtdeterministischen Schritte eine „Zusatzinformation“ $y \in \Sigma_0^*$ erzeugt wird, um dann die Zugehörigkeit des Paars (w, x) zu einer deterministischen Sprache zu überprüfen. Die nichtdeterministischen und deterministischen Berechnungsteile werden also separiert. Diese Modellvorstellung führt auf einen Spezialfall einer nichtdeterministischen Turingmaschine, das **RV-Modell (rate und verifiziere) einer nichtdeterministischen Turingmaschine**:

Eine nichtdeterministische *k*-Band-Turingmaschine (*k*-NDTM) *TM* gemäß dem RV-Modell ist definiert durch $TM = (Q, \Sigma, I, \delta, b, q_0, q_{accept})$. Die Überführungsfunktion ist eine partielle Abbildung $\delta : (Q \setminus \{q_{accept}\}) \times \Sigma^k \to \mathbf{P}(Q \times (\Sigma \times \{L, R, S\})^k)$, die sich in zwei Teile δ_1 und δ_2 ($\delta = \delta_1 \cup \delta_2$) zerlegen lässt. Alle nichtdeterministischen Teile von δ sind in δ_1 zusammengefasst, δ_2 besteht ausschließlich aus deterministischen Teilen. Das Eingabeband (1. Band) ist nach links abzählbar unendlich erweitert worden; die Zellen werden mit 0, -1, -2, ... nummeriert. Die Zelle mit der Nummer 0 enthält ein eindeutiges Markierungszeichen #, so dass die beiden Bandteile mit positiven und negativen Nummern während des folgenden Ablaufs separiert werden können. Zusätzlich verfügt *TM* über ein „Ratemodul“, das einen zusätzlichen Schreib/Lesekopf besitzt, der über dem Bandabschnitt des 1. Bands mit nichtpositiven Nummern operiert. Anfangs steht dieser Kopf über der Zelle mit der Nummer 0.



Ein Eingabewort $w \in I^*$ steht auf dem 1. Band in den Zellen mit Nummern 1, ..., n mit $n = |w|$. Allen übrigen Zellen, außer der Zelle mit der Nummer 0, enthalten jeweils das Leerzeichen. Die übrigen Schreib/Leseköpfe stehen auf den jeweiligen Bändern über der ersten Zelle.

Die Arbeitsweise von *TM* verläuft in zwei Phasen. In Phase 1 schreibt das Ratemodul getaktet mit seinem Schreib/Lesekopf auf das 1. Band ein Symbol aus Σ_0 mit $\Sigma_0 \subseteq \Sigma$ und bewegt jeweils den Kopf um eine Zelle nach links. Dann wird der Vorgang der Zeichenerzeugung gestoppt; wie viele Zeichen erzeugt werden, wird auf nichtdeterministische Weise entschieden. Anschließend bewegt das Ratemodul den Kopf des 1. Bands auf die Zelle mit der Nummer –1 (zur Erkennung der Zellennummer wird die Markierung in Zelle 0 verwendet). Dieser Vorgang der Phase 1 wird durch den nichtdeterministischen Teil δ_1 von δ gesteuert. Die so vom Ratemodul auf dem 1. Band erzeugte Zeichenkette wird auch als **Beweis (Zertifikat)** bezeichnet. Das Ratemodul **rät** auf diese Weise **nichtdeterministisch** einen in Phase 2 „brauchbaren“ Beweis, falls es ihn überhaupt gibt. Das Ratemodul „weiß“, wie viele Zeichen im Beweis und welche Zeichen für den nun folgenden Berechnungsabschnitt der Phase 2 benötigt werden.

In Phase 2 verhält sich *TM* deterministisch, gesteuert durch den deterministischen Teil δ_2 von δ, wie eine „normale“ *k*-DTM, wobei die Zeichenkette, die das Ratemodul in Phase 1 erzeugt hat, in die Berechnung einbezogen wird. Dazu wird der in Phase 1 erzeugte Beweis gelesen.

Ein typischer Beweis besteht beispielsweise aus einer Bitfolge, die abschnittsweise als Zahlenfolge natürlicher Zahlen interpretiert werden kann. Auch δ_2 enthält zunächst nichtdeterministische Anteile. Jede der Zahlen auf Band 1 bestimmt in einem nachfolgenden durch δ_2 gesteuerten eigentlich nichtdeterministischen Schritt die Nummer einer Alternative von δ_2, so dass dann Nichtdeterminismus vermieden werden kann. Die syntaktische Form muss aber im allgemeinen keine Bitfolge sein, die als Folge von Zahlenwerten interpretierbar ist. In den Beispielen der folgenden Kapitel, insbesondere Kapitel 5, werden die syntaktische Form und die Bedeutung eines Beweises explizit angegeben. Eventuell stoppt das Ratemodul nicht, so dass die Turingmaschine bei der entsprechenden Eingabe nicht anhält.

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.

Alternativ könnte man den Bandabschnitt des 1. Bands mit nichtpositiven Zellennummern auch als zusätzliches Band modellieren. In der hier gewählten Darstellung wird jedoch eher deutlich, dass sowohl das Eingabewort $w \in I^*$ als auch der selbst erzeugte Beweis die Funktionen von Eingaben für die Verifikationsphase übernehmen.

Man kann wie in Satz 2.5-2 zeigen, dass sich jede NDTM in eine äquivalente Turingmaschine im RV-Modell überführen lässt. Je nach Anwendung kann man daher Nichtdeterminismus im ursprünglichen oder im speziellen RV-Modell beschreiben.

Die deterministische Simulation des nichtdeterministischen Verhaltens einer Turingmaschine aus Satz 2.5-1 führt auf eine exponentielle Zeitsteigerung. Wie bereits erwähnt, ist bis heute keine Simulation bekannt, die diesen exponentiellen Zeitsprung vermeidet. 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 2.5-3:

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 $TM = \left(Q, \Sigma, I, \delta, b, q_0, q_{accept}\right)$ 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 $K = \left(q, (\alpha_1, i_1), ..., (\alpha_k, i_k)\right)$ eine Konfiguration in einer Berechnung von *TM* bei Eingabe von *w*. Wegen $|\alpha_j| \le S(n)$ 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 $TM = \left(Q, \Sigma, I, \delta, b, q_0, q_{accept}\right)$ 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,(α1,i1),...,(αk,ik))
                         mit |αj| ≤ S(n) und 1 ≤ ij ≤ 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 := (q0,(w,1),(ε,1),...,(ε,1))
  OK := FALSE;
  FOR_EACH Endkonfiguration Kf = (qf,(α1,|α1|+1),...,(αk,|αk|+1)) mit |αj| ≤ 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 $1+\log(\lceil c^{S(n)} \rceil) \in O(S(n))$ 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.

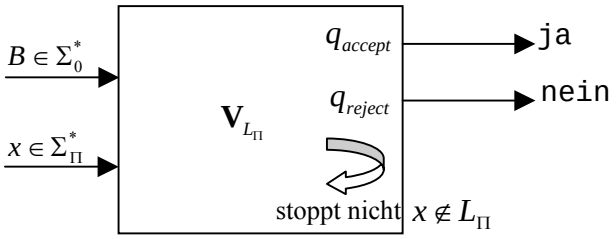
///

Das Konzept des Nichtdeterminismus lässt sich auf Algorithmen (formuliert als Programm in einer Programmiersprache, vgl. Kapitel 2.3) übertragen. Es soll hier der dem RV-Modell einer nichtdeterministischen Turingmaschine entsprechende Ansatz beschrieben werden, der auf die Definition **eines nichtdeterministischen Algorithmus unter Einsatz eines Verifizierers** führt. 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 Verifizierer repräsentiert also die Verifikationsphase** im RV-Modell einer nichtdeterministischen Turingmaschine; der erforderliche Beweis wird zuvor von einem **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: $x \in \Sigma_\Pi^*$, $B \in \Sigma_0^*$

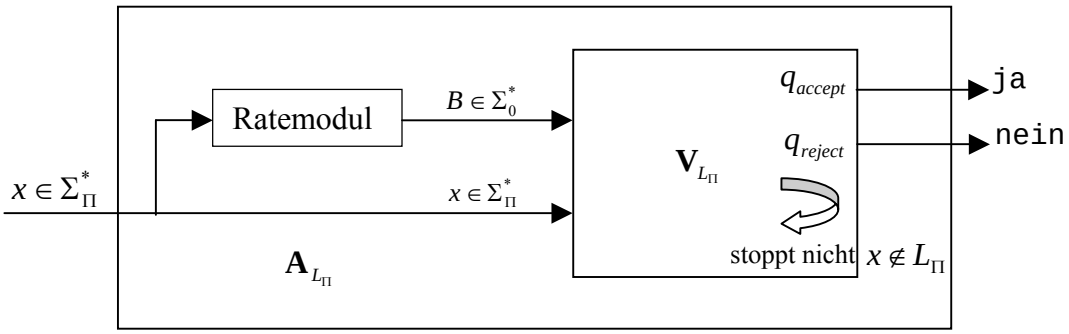
Ausgabe: `ja` (accept), falls die in *B* enthaltene Information die Eigenschaft $x \in L_\Pi$ belegt
`nein` (reject), falls die in *B* enthaltene Information die Eigenschaft $x \in L_\Pi$ nicht belegt
falls $\mathbf{V}_{L_\Pi}$ bei einer Eingabe $x \in \Sigma_\Pi^*$ nicht stoppt, gilt $x \notin L_\Pi$



Falls $\mathbf{V}_{L_\Pi}$ bei Eingabe von $x \in \Sigma_\Pi^*$ stoppt, wird mit $\mathbf{V}_{L_\Pi}(x,B)$, $\mathbf{V}_{L_\Pi}(x,B) \in \{$ `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



Für $x \in \Sigma_\Pi^*$ wird folgende Entscheidung getroffen:
$x \in L_\Pi$ genau dann, wenn es einen Beweis $B_x \in \Sigma_0^*$ gibt, so dass $\mathbf{V}_{L_\Pi}$ bei Eingabe von *x* und B_x mit $\mathbf{V}_{L_\Pi}(x,B_x) =$ `ja` stoppt;
$x \notin L_\Pi$ genau dann, wenn 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 $\mathbf{V}_{L_\Pi}(x,B) =$ `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.

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 Problemgröße der Eingabe wird $n \cdot A$ mit $A = \lfloor \log_2(\max(\{a_i \mid i = 1, ..., n\})) \rfloor + 1$ 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 Pseudocode 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

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

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

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 $n^2 \cdot A \leq (n \cdot A)^2$. Ist k die Problemgröße der Eingabeinstanz, dann ist $k \in O(n \cdot A)$, und es ergibt sich eine obere Schranke für den Aufwand dieses nichtdeterministischen Algorithmus der Ordnung $O(k^2)$, also nichtdeterministisch polynomielles Laufzeitverhalten (gemessen in Bitoperationen).

Für dieses Problem ist bisher kein deterministischer Entscheidungsalgorithmus bekannt, dessen Laufzeit sich durch ein Polynom beschränken lässt.

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 Auswertung 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 kann 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. Auch für dieses Problem ist nicht bekannt, ob es einen *deterministischen* Algorithmus gibt, der die Erfüllbarkeit Boolescher Ausdrücke mit einem Laufzeitverhalten der Ordnung $O\big(p(n)\big)$ 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 simulie-

ren 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 2.5-4:

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 $O\left(T(n) \cdot d^{O(T(n))}\right)$ mit einer Konstanten $d \geq 2$ besitzt.

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

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). 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_Π}(x);

VAR n       : INTEGER;
    lng     : INTEGER;
    B       : Σ*_0;
    antwort : BOOLEAN;
    weiter  : BOOLEAN;

BEGIN { A'_{L_Π}(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 Σ*_0 mit Länge ≤ lng erzeugt worden)
    THEN weiter := FALSE
    ELSE
      BEGIN
        B:= nächstes Wort aus Σ*_0 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 deterministische Turingmaschine mit einer dem jeweiligen Problem angemessenen geeigneten Anzahl an Bändern verstanden.

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 „auf algorithmische Weise 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 $L = h\left((\Sigma \cup \{0,1,\#\})^*\right)$ wird folgenderma-

ßen 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 wird der Begriff der Abzählbarkeit definiert: Eine Menge *M* ist abzählbar unendlich, wenn es eine bijektive Abbildung $f : \mathbf{N} \to M$ gibt, d.h. $M = \{ f(i) \mid i \in \mathbf{N} \} = f(\mathbf{N})$,

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 $bin(i)$. Ein Wort $w = a_{i_1} \ldots a_{i_n}$ kann man dann zunächst in $bin(i_1)\# \ldots \# bin(i_n)\#$ 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 $w = a_{i_1} \ldots a_{i_n}$ und $v = b_{j_1} \ldots b_{j_m}$ zunächst in $(bin(i_1)\# \ldots \# bin(i_n)\#\# bin(j_1)\# \ldots \# bin(j_m)\#)$ 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 $L = h(\{0,1\}^*)$.

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 1.1-7 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 **Diagonalisierung** konstruieren. 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	ε	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:

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

Dann gilt:

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

Beweis:

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.

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

Satz 3.2-1:

(i) 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.

(ii) 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 in Teil (ii) 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 Turingmaschine (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:

(i) 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.

(ii) Jede endliche Menge ist entscheidbar.

(iii) 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.

(iv) 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 (i): Die Aussage ergibt sich direkt aus der Definition.

Zu (ii): 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
w1 : accept;
...
wn : accept;
ELSE reject;
END;
```

realisert.

Zu (iii): 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 $L(TM') = \Sigma^* \backslash L$ erhält man aus *TM*, indem man die Rollen von q_{accept} und q_{reject} vertauscht.

Zu (iv): Diese Aussage wird genauso gezeigt wie bei rekursiv aufzählbaren Mengen.

///

Die folgende Definition stellt ein Hilfsmittel 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:

(i) Ist L' entscheidbar, dann ist auch *L* entscheidbar.

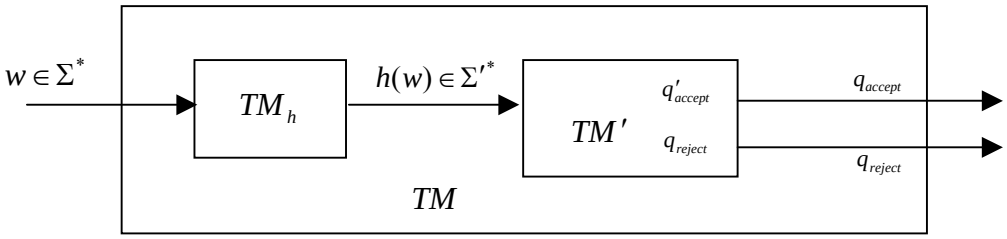
(ii) Ist *L* nicht entscheidbar, dann ist auch L' nicht entscheidbar.

(iii) Ist L' rekursiv aufzählbar, dann ist auch *L* rekursiv aufzählbar.

(iv) Ist *L* nicht rekursiv aufzählbar, dann ist auch L' nicht rekursiv aufzählbar.

Beweis:

Zu (i): 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).
Die folgende Argumentation zeigt, dass $L(TM) = L$ gilt:
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 (ii): Hier ist nichts zu beweisen, da es sich um eine logisch äquivalente Formulierung zu (i) handelt.

Zu (iii): Die Argumentation verläuft wie in (i), wobei im vorherigen Bild nur der mit q_{accept} markierte Ausgang betrachtet wird.

Zu (iv): Es handelt sich um eine logisch äquivalente Formulierung zu (iii).

///

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:

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

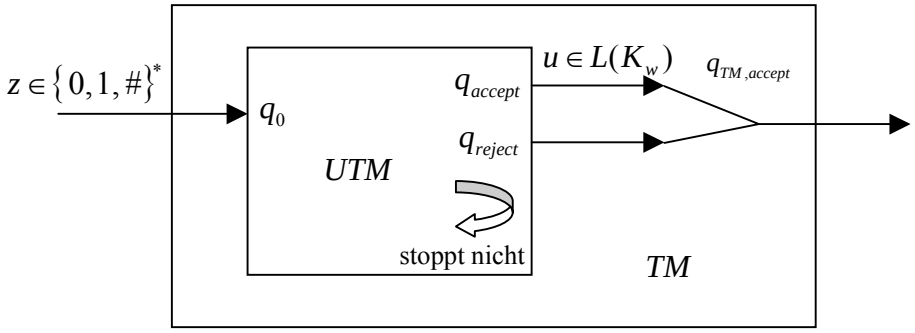
$$L_H = \left\{ z \;\middle|\; \begin{array}{l} z = u\#w \text{ mit } u \in \{0,1\}^*, w \in \{0,1\}^*, VERIFIZIERE\_TM(w) = \mathsf{TRUE} \text{ und} \\ K_w \text{ stoppt bei Eingabe von } u \end{array} \right\}.$$

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 δ_{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 δ_{UTM} um die Zeilen $\delta_{UTM}(q_{accept}, a) = (q_{TM,accept}, (a,S))$ und $\delta_{UTM}(q_{reject}, a) = (q_{TM,accept}, (a,S))$ 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.

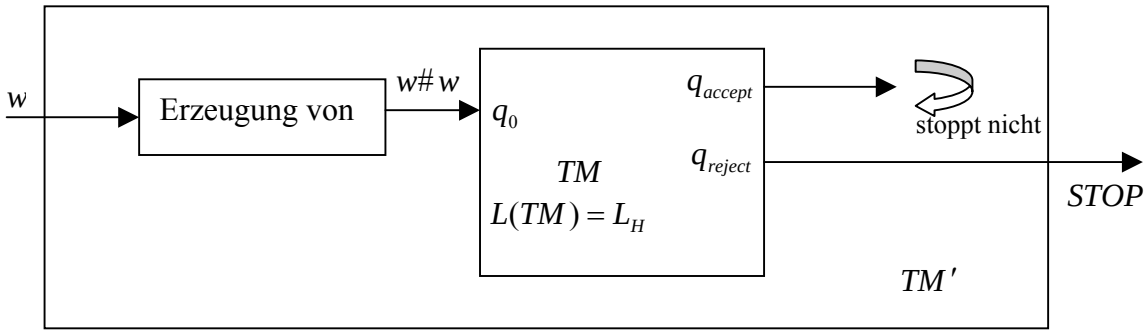
Ist $z \notin L_H$, dann hat *z* entweder nicht die syntaktische Form $z = u\#w$ mit $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$ oder $VERIFIZIERE\_TM(w) = \mathsf{FALSE}$ oder K_w stoppt bei Eingabe von *u* nicht. In den ersten beiden Fällen stoppt *UTM* bei Eingabe von *z* nicht (vgl. Beweis von Satz 2.4-1). 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[6]. 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.

[6] Ist TM' eine *k*-DTM mit der Überführungsfunktion δ', dann ist $\delta'(q', a_1, ..., a_k) = (q', (a_1, S), ..., (a_k, S))$ 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

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

Satz 3.2-6:

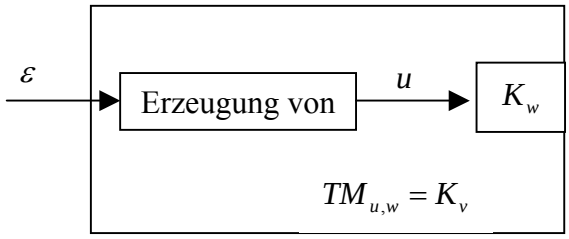
L_{H_0} ist rekursiv aufzählbar, aber nicht entscheidbar.

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 (ii) 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

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

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

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

$\Leftrightarrow TM_{u,w} = K_{code(TM_{u,v})} = K_{h(u\#w)}$ stoppt bei leerem Eingabeband

$\Leftrightarrow h(u\#w) \in L_{H_0}$.

///

Eine Charakterisierung der Entscheidbarkeit einer Menge liefert der folgende Satz:

Satz 3.2-7:

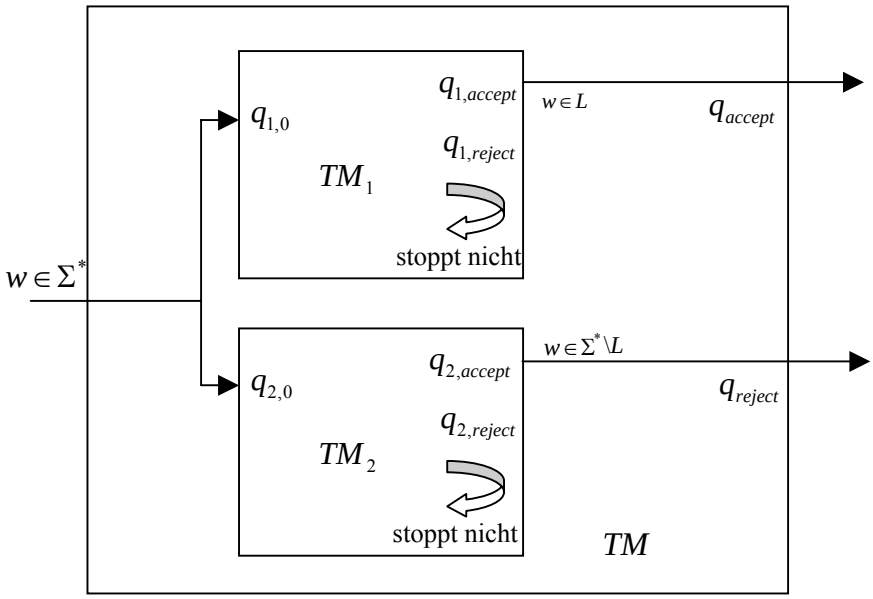
Eine Menge $L \subseteq \Sigma^*$ ist genau dann entscheidbar, wenn sowohl *L* als auch das Komplement $\Sigma^* \setminus L$ von *L* rekursiv aufzählbar sind.

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 $\Sigma^* \backslash L$ 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

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

oder

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

oder

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:

Der Beweis erfolgt in mehreren Schritten, die jeweils einzeln begründet werden. Es wird dabei 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.

1. Die Menge L_{uni} ist rekursiv aufzählbar.

   Begründung: L_{uni} wird gerade von der Turingmaschine *UTM* akzeptiert.

2. Die Menge L_{uni} ist nicht entscheidbar.

   Zur Begründung: Satz 3.1-4 zeigt, dass die Menge

   $L_d = \left\{ w \,\middle|\, w \in \{0,1\}^*, \mathit{VERIFIZIERE\_TM}(w) = \mathsf{FALSE} \text{ oder } w \notin L(K_w) \right\}$

   nicht rekursiv aufzählbar ist. Daher ist die Menge $\{0,1\}^* \setminus L_d$ ist nicht entscheidbar. Die Funktion $g : \{0,1\}^* \to \{0,1,\#\}^*$ mit $g(w) = w\#w$ ist total und durch eine Turingmaschine berechenbar. Es gilt

   $w \in \{0,1\}^* \setminus L_d \quad \Leftrightarrow \; w \notin L_d$

   $\Leftrightarrow$ *VERIFIZIERE_TM*(*w*) = TRUE und $w \in L(K_w)$

   $\Leftrightarrow \; w\#w \in L_{uni}$

   $\Leftrightarrow \; g(w) \in L_{uni}$.

   Das bedeutet $\{0,1\}^* \setminus L_d \le L_{uni}$. Nach Satz 3.2-3 ist daher L_{uni} nicht entscheidbar.

3. Das Komplement $\overline{L}_{uni} = \{0,1,\#\}^* \setminus L_{uni}$ nicht rekursiv aufzählbar.

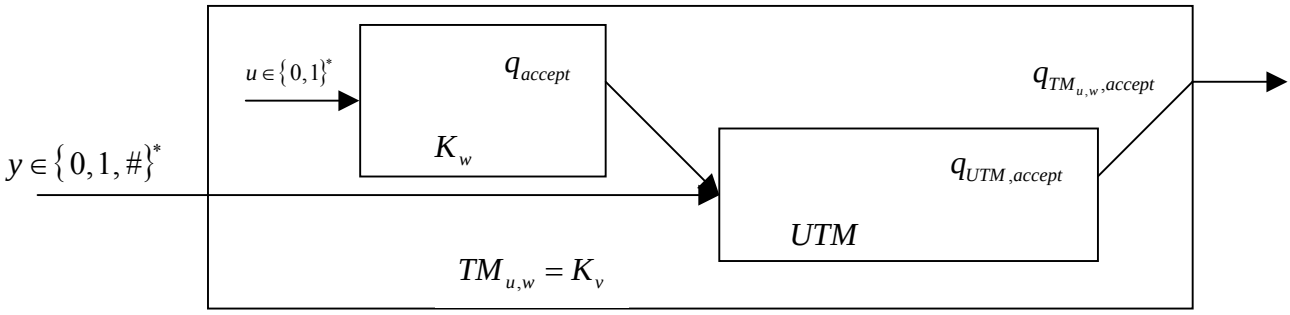
   Begründung: Satz 3.2-3 und 2.

4. Es gilt $\overline{L}_{uni} \le L_e$.

   Zur Begründung: Das Aussehen der dabei beteiligten totalen und berechenbaren Funktion $h : \{0,1,\#\}^* \to \{0,1\}^*$ wird im folgenden skizziert. Aus Satz 3.2-3 folgt, dass L_e nicht rekursiv aufzählbar ist.

   Zu $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$ mit *VERIFIZIERE_TM*(*w*) = TRUE wird 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

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

Wie obige Ausführungen zeigen, ist *h* ist total und berechenbar. Außerdem gilt nach Definition von $\overline{L}_{uni}$ bzw. L_{uni}

$z \in \overline{L}_{uni} \Leftrightarrow$ *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
$VERIFIZIERE\_TM(w) = \mathsf{FALSE}$
oder
z hat die Form $z = u\#w$ mit $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$ und
$VERIFIZIERE\_TM(w) = \mathsf{TRUE}$ 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 $u \in \{0,1\}^*$, $w \in \{0,1\}^*$, $VERIFIZIERE\_TM(w) = \mathsf{TRUE}$ 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 $\overline{L}_{uni} \le L_e$ gezeigt.

Der Beweis für L_{ne} verläuft entsprechend.

///

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) $L \neq \varnothing$

(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?
- 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. $L = \{ w \mid \text{die durch } w \text{ kodierte Turingmaschine } K_w \text{ berechnet eine konstante Funktion} \}$; das Entscheidungsproblem lautet: Ist es entscheidbar, ob eine Turingmaschine eine konstante Funktion berechnet?
2. Es sei g eine Turingberechenbare Funktion.
$$L = \left\{ w \;\middle|\; \begin{array}{l} \text{die durch } w \text{ kodierte Turingmaschine } K_w \text{ berechnet} \\ \text{eine mit } g \text{ identische Funktion} \end{array} \right\};$$
das Entscheidungsproblem lautet: Ist es entscheidbar, ob die von einer Turingmaschine berechnete Funktion mit g übereinstimmt?
3. $L_{=\varnothing} = \{ w \mid \text{für die durch } w \text{ kodierte Turingmaschine } K_w \text{ gilt } L(K_w) = \varnothing \}$; 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 entscheidbare 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 $L_{H_0} \leq \{0,1\}^* \backslash L$ gilt. Mit Satz 3.2-3 lässt sich dann argumentieren: Da L_{H_0} nicht entscheidbar ist, ist im ersten Fall *L* nicht entscheidbar und im zweiten Fall $\{0,1\}^* \backslash L$ und damit auch *L* nicht entscheidbar (Satz 3.2-2).

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: $w_0 \notin L$

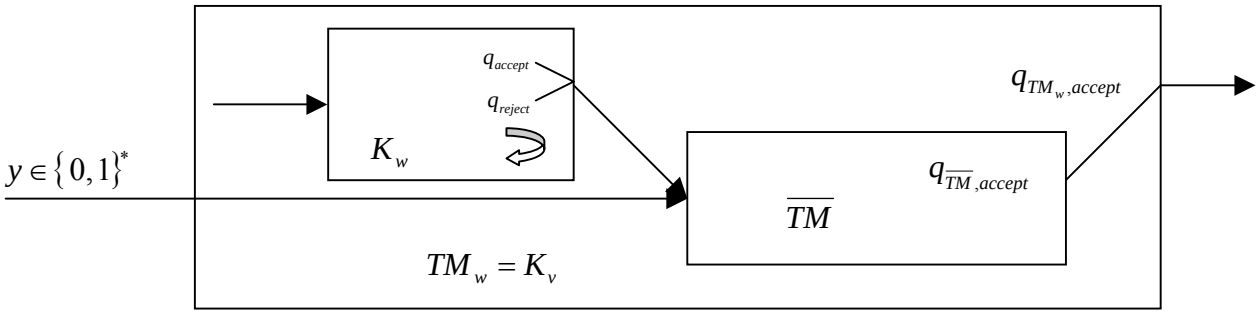
unterschieden.

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 $L_{H_0} \leq \{0,1\}^* \backslash L$ gezeigt, indem eine totale und berechenbare Funktion $h : \{0,1\}^* \to \{0,1\}^*$ angegeben wird, für die $w \in L_{H_0} \Leftrightarrow h(w) \in \{0,1\}^* \backslash L$ 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 $L_{H_0} \le \{0,1\}^* \setminus L$ geeignet:

Ist $w \in L_{H_0}$, dann ist (nach Definition von L_{H_0}) *VERIFIZERE_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. $h(w) \in \{0,1\}^* \setminus L$.

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

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. $h(w) \notin \{0,1\}^* \setminus L$.

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 trivia-

len Fällen fündig, die in Satz 3.2-11 beschrieben sind. Selbstverständlich gibt es nichttriviale entscheidbare Mengen; 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

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

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

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 $w_2 \notin L_{=\varnothing}$.

Im Fall von $L_{=\varnothing}$ und $L_{\neq\varnothing}$ kann man auch anders argumentieren: Man zeigt direkt, dass die Sprache $L_{\neq\varnothing}$ rekursiv aufzählbar und folglich $\{0,1\}^* \setminus L_{\neq\varnothing}$ nicht rekursiv aufzählbar ist (denn sonst wäre $L_{\neq\varnothing}$ entscheidbar). Weiterhin gilt $\{0,1\}^* \setminus L_{\neq\varnothing} \leq L_{=\varnothing}$, und daher ist $L_{=\varnothing}$ nicht rekursiv aufzählbar. Dazu ist zum einen eine Turingmaschine *TM* anzugeben mit $L(TM) = L_{\neq\varnothing}$,

zum anderen zum Nachweis der Relation $\{0,1\}^* \setminus L_{\neq\varnothing} \leq L_{=\varnothing}$ 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 $L(K_w) \neq \varnothing$ und $w \in L_{\neq\varnothing}$.
Ist umgekehrt $w \in L_{\neq\varnothing}$, d.h. $L(K_w) \neq \varnothing$, 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 $\{0,1\}^* \setminus L_{\neq\varnothing} \leq L_{=\varnothing}$ 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 $L(K_{w_0}) = \varnothing$.

Für die in Satz 3.2-9 untersuchten Sprachen

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

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

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

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 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**.

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

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

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

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

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

5. $G_5 = (\{0,1\}, \{S, A\}, S, \{S \to 1A0A, S \to 0A1A, A \to \varepsilon, A \to 0A1A, A \to 1A0A\})$ erzeugt die Sprache
$L_5 = \left\{ w \mid w \in \{0,1\}^+ \text{ und die Anzahl der Zeichen 0 in } w \text{ ist gleich der Anzahl der Zeichen 1} \right\}$

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

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

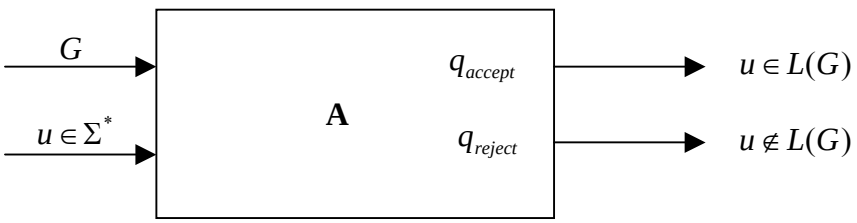
Die durch ein Wort $w \in \{0,1\}^*$ mit *VERIFIZIERE_G*(w) = TRUE kodierte Grammatik sei KG_w.

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.

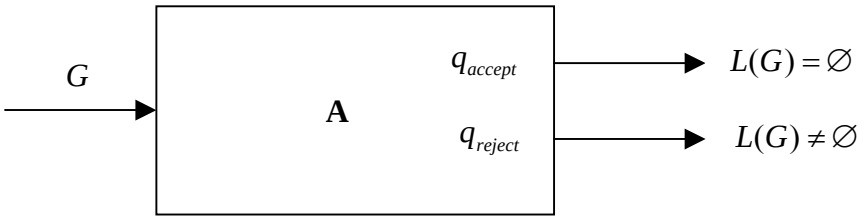


Entsprechend kann man das Leerheitsproblem definieren:

Die **zum Leerheitsproblem gehörende Menge** L_{leer} wird definiert durch

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

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 $R \subseteq \left\{ v \to w \,|\, v \in (N \cup \Sigma)^+ \setminus \Sigma^*, w \in (N \cup \Sigma)^* \right\}$ 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 $R \subseteq \left\{ v \to w \mid v \in (N \cup \Sigma)^+ \setminus \Sigma^*, w \in (N \cup \Sigma)^* \right\}$ folgender zusätzlicher Einschränkung unterliegen:
Für alle Regeln $v \to w$ gilt $|v| \le |w|$. Als einzige Ausnahme ist die Regel $S \to \varepsilon$ 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 $S \to \varepsilon$ 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 $S \to \varepsilon$, dann ist $\varepsilon \in L(G)$, und die Anwendung der Regel $S \to \varepsilon$ stellt die einzige Möglichkeit dar, um ε 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 $L_4 = \left\{ a^n b^n c^n \mid n \in \mathbf{N}, n \ge 1 \right\}$ aus Kapitel 4.1 eine kontextsensitive Sprache (Typ-1-Sprache).

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

Es ist nicht immer einfach, für eine gegebene Sprache die sie erzeugende Grammatik zu entwickeln, wie folgendes Beispiel zeigt:

Eine Typ-1-Sprache für $L = \left\{ a^{2^i} \mid i \in \mathbf{N}_{>0} \right\}$

Ein erster Ansatz führt auf die Grammatik $G_1 = \left(\{ a \}, \{ S, A, B, C, D, E \}, S, R \right)$, wobei R aus den Regeln

(1) $S \to ACaB$
(2) $Ca \to aaC$
(3) $CB \to DB$
(4) $CB \to E$
(5) $aD \to Da$
(6) $AD \to AC$
(7) $aE \to Ea$
(8) $AE \to a$

besteht. Die Nichtterminalzeichen *A* und *B* dienen als „Begrenzer"; *C* ist eine Markierung, die durch die Zeichenkette *a* ... *a* zwischen *A* und *B* hindurchläuft und dabei die Anzahl der Zeichen *a* verdoppelt; *D* läuft in der anderen Richtung zwischen *B* und *A*. Die Ableitung der Zeichenkette a^{16} lautet (die Zeichen, an denen die nächste Anwendung einer Regel ansetzt, sind jeweils unterstrichen):

$$\underline{S} \Rightarrow A\underline{Ca}B \Rightarrow Aaa\underline{CB} \Rightarrow Aa\underline{aD}B \Rightarrow A\underline{aD}aB \Rightarrow \underline{AD}aaB \Rightarrow A\underline{Ca}aB \Rightarrow^* AaaaaCB$$
$$\Rightarrow^* \underline{AD}aaaaB \Rightarrow^* Aa^{16}\underline{CB} \Rightarrow Aa^{15}\underline{aE} \Rightarrow Aa^{14}\underline{aE}a \Rightarrow^* \underline{AE}a^{16} \Rightarrow a^{16}$$

Die Grammatik G_1 ist aufgrund der syntaktischen Form der Regeln (4) und (8) nicht kontextsensitiv. Es gibt jedoch eine kontextsensitive Grammatik G_2 auf der Basis von G_1, die *L* erzeugt, so dass *L* kontextsensitiv ist: Die nichtterminalen Zeichen *A*, ..., *E* werden mit Zeichen *a* zu neuen nichtterminalen Zeichen zusammengesetzt:

$G_2 = \left(\{ a \}, N_2, S, R_2 \right)$ mit
$N_2 = \{ S, Z_{ACaB}, Z_{Ca}, Z_{aB}, Z_{CaB}, Z_{ACa}, Z_{Aa}, Z_{aCB}, Z_{aDB}, Z_{aE}, Z_{Da}, Z_{DaB}, Z_{ADa}, Z_{Ea}, Z_{AEa} \}$; aus den ursprünglichen Regeln (1) bis (8) wird:

(1) $S \to Z_{ACaB}$
(2) $Z_{Ca}a \to aaZ_{Ca}$
$Z_{Ca}Z_{aB} \to aaZ_{CaB}$
$Z_{ACa}a \to Z_{Aa}aZ_{Ca}$
$Z_{ACa}Z_{aB} \to Z_{Aa}aZ_{CaB}$
$Z_{ACaB} \to Z_{Aa}Z_{aCB}$
$Z_{CaB} \to aZ_{aCB}$
(3) $Z_{aCB} \to Z_{aDB}$
(4) $Z_{aCB} \to Z_{aE}$
(5) $aZ_{Da} \to Z_{Da}a$
$Z_{aDB} \to Z_{DaB}$
$Z_{Aa}Z_{Da} \to Z_{ADa}a$
$aZ_{DaB} \to Z_{Da}Z_{aB}$
$Z_{Aa}Z_{DaB} \to Z_{ADa}Z_{aB}$
(6) $Z_{ADa} \to Z_{ACa}$
(7) $aZ_{Ea} \to Z_{Ea}a$
$Z_{aE} \to Z_{Ea}$
$Z_{Aa}Z_{Ea} \to Z_{AEa}a$
(8) $Z_{AEa} \to a$

Die Ableitung von a^{16} in G_2 lautet (wieder sind die Zeichen, an denen die nächste Anwendung einer Regel ansetzt, jeweils unterstrichen):

$$\begin{aligned}
\underline{S} &\Rightarrow Z_{ACaB} \Rightarrow Z_{Aa}\underline{Z_{aCB}} \Rightarrow Z_{Aa}\underline{Z_{aDB}} \Rightarrow \underline{Z_{Aa}Z_{DaB}} \Rightarrow \underline{Z_{ADa}Z_{aB}} \Rightarrow \underline{Z_{ACa}Z_{aB}} \Rightarrow Z_{Aa}a\underline{Z_{CaB}} \\
&\Rightarrow Z_{Aa}aa\underline{Z_{aCB}} \Rightarrow Z_{Aa}aa\underline{Z_{aDB}} \Rightarrow Z_{Aa}a\underline{aZ_{DaB}} \Rightarrow Z_{Aa}\underline{aZ_{Da}}Z_{aB} \Rightarrow \underline{Z_{Aa}Z_{Da}}aZ_{aB} \\
&\Rightarrow \underline{Z_{ADa}}aaZ_{aB} \Rightarrow \underline{Z_{ACa}}aaZ_{aB} \Rightarrow Z_{Aa}a\underline{Z_{Ca}}aZ_{aB} \Rightarrow Z_{Aa}aaa\underline{Z_{Ca}Z_{aB}} \Rightarrow Z_{Aa}aaaaa\underline{Z_{CaB}} \\
&\Rightarrow Z_{Aa}aaaaaa\underline{Z_{aCB}} \Rightarrow Z_{Aa}aaaaaa\underline{Z_{aDB}} \Rightarrow^{*} Z_{Aa}a^{14}\underline{Z_{aCB}} \Rightarrow Z_{Aa}a^{14}\underline{Z_{aE}} \Rightarrow Z_{Aa}a^{13}\underline{aZ_{Ea}} \\
&\Rightarrow Z_{Aa}a^{12}\underline{aZ_{Ea}}a \Rightarrow^{*} \underline{Z_{Aa}Z_{Ea}}a^{14} \Rightarrow \underline{Z_{AEa}}aa^{14} \Rightarrow a^{16}
\end{aligned}$$

Es stellt sich die Frage, ob es Typ-0-Sprachen gibt, die nicht kontextsensitiv sind. Der folgende Satz beantwortet die Frage.

Satz 4.3-1:

Jede kontextsensitive Sprache über einem Alphabet Σ ist entscheidbar.

Die Entscheidung kann bei einem Wort $w \in \Sigma^*$ mit Länge *n* in $O\left(2^{O(n)}\right)$ 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:

Es sei $w \in \Sigma^*$ eine Eingabe von TM_G. Für $w = \varepsilon$ akzeptiert TM_G diese Eingabe genua dann, wenn *G* die Ableitungsregel $S \to \varepsilon$ enthält. Ist $|w| = n \geq 1$, so 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 $\alpha \in (N \cup \Sigma)^*$ und der Knoten K_j mit $\beta \in (N \cup \Sigma)^*$ markiert und kann man in *G* durch Anwendung einer Regel β aus α in einem Ableitungsschritt herleiten, d.h. $\alpha \underset{G}{\Rightarrow} \beta$, 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 $O(2^{O(n)})$.

///

Eine nichtentscheidbare Menge kann also nicht kontextsensitiv sein.

Wie bei Typ-0-Sprachen lassen sich Typ-1-Sprachen durch ein Berechnungsmodell beschreiben, das sich aus dem Modell der Turingmaschine ableitet und genau kontextsensitive Sprachen akzeptiert:

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[7].

Satz 4.3-2:

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.

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

---

7 Den Beweis dieser Aussage findet man beispielsweise in Hopcroft, J.E.; Ullman, J.D.: **Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie**, Addison Wesley, 1988.

Möchte man daher Aussagen über kontextsensitive Sprachen beweisen, so kann man dazu mit kontextsensitiven Grammatiken oder linear beschränkten Automaten argumentieren.

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* berechnet zunächst $h(w)$. Dabei ist $h(w) \in L_0$, insbesondere $\mathit{VERIFIZIERE\_TM}(h(w)) = \mathsf{TRUE}$. 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 Klasse 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 $\delta: Q \times \Sigma^k \to \mathbf{P}(Q \times (\Sigma \times \{L,R,S\})^k)$ hat, ein **deterministischer linear beschränkter Automat** liegt vor, wenn die Überführungsfunktion die Form $\delta: Q \times \Sigma^k \to Q \times (\Sigma \times \{L,R,S\})^k$ 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 *deter-*

ministischen 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 mit einer platzkonstruierbaren Funktion f, kann deterministisch simuliert werden, wobei dabei der Speicherplatzbedarf die Ordnung $O(f^2(n))$ annimmt (Satz 2.5-2). 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 $O(S^2(n)) = O(n^2)$ 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[8].

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 $R \subseteq \{ A \to w \mid A \in N \text{ und } w \in (N \cup \Sigma)^* \}$ 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 $L_2 = \{ a^n b^n \mid n \in \mathbf{N} \}$, $L_3 = \{ a^n b^n c^i \mid n \in \mathbf{N}, i \in \mathbf{N} \}$ 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

8 Den Beweis dieser Aussage findet man ebenfalls in Hopcroft, J.E.; Ullman, J.D.: **Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie**, Addison Wesley, 1988.

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 (diese Tatsache wird am Ende von Kapitel 4.5 erläutert).

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 $A \to \alpha_1 | \alpha_2 | ... | \alpha_n$ steht für die *n* Regeln $A \to \alpha_1$, $A \to \alpha_2$, ..., $A \to \alpha_n$. Ein Angabe in eckigen Klammern innerhalb einer Regel bezeichnet einen optionalen Teil, d.h. $A \to \alpha[\beta]\gamma$ steht für die Regeln $A \to \alpha\beta\gamma$ und $A \to \alpha\gamma$.

```
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 $A \to \varepsilon$ 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 $A \to \varepsilon$ mehr vorkommen außer für den Fall, dass 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.[9]

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 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
$S \to \varepsilon$ 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.

[9] Den Beweis dieses und des folgenden Satzes findet man in Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.: **The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing**, Addison-Wesley, 1972.

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) $|vwx| \leq n_0$

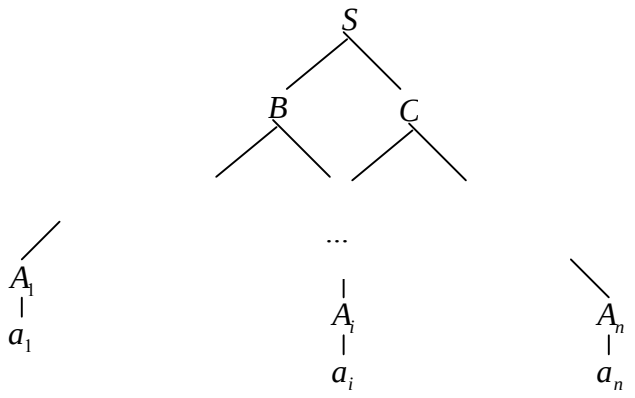
(ii) $|vx| > 0$

(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 $z = a_1 \ldots a_n$ mit $a_1 \in \Sigma$, ..., $a_n \in \Sigma$ und $|z| = n \geq n_0$, hat eine Ableitung aus S die Form $S \Rightarrow BC \Rightarrow \ldots \Rightarrow a_1 \ldots a_n$. 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 $\lceil \log_2(n)+1 \rceil \geq \log_2\left(2^{|N|+1}\right)+1 = |N|+2$. 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. Da *G* kontextfrei ist, kann die Ableitung $S \Rightarrow^* z$ so organisiert werden, dass gilt:
$S \Rightarrow^* w_1 A w_2 \Rightarrow^+ w_1 w_3 A w_4 w_2 \Rightarrow^* uvwxy$ mit $w_1 \Rightarrow^* u$, $w_3 \Rightarrow^* v$, $A \Rightarrow^* w$, $w_4 \Rightarrow^* x$ und $w_2 \Rightarrow^* y$, und in der Teilableitung $w_1 A w_2 \Rightarrow^+ w_1 w_3 A w_4 w_2$ wird in keinem Ableitungsschritt, bis auf den ersten, das nichtterminale Symbol *A* ersetzt. Ebenso werden in jedem Ableitungsschritt der Teilableitungen $w_3 \Rightarrow^* v$, $A \Rightarrow^* w$ und $w_4 \Rightarrow^* x$ jeweils von *A* verschiedene nichtterminale Symbole ersetzt.
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 und jeweils von *A* verschiedene nichtterminale Symbole ersetzt werden. Daher gilt $|vwx| \leq 2^{|N|} < n_0$ (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 $k \geq 1$; 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 $L_4 = \left\{ a^n b^n c^n \mid n \in \mathbf{N}, n \geq 1 \right\}$ (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 $|z| = 3n_0 \geq n_0$. 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 $uv^0wx^0y \in L_4$ ist, d.h. $uwy \in L_4$. Es werden zwei 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* $k \geq 1$ 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 $|vwx| \leq n_0$ enthält es kein Zeichen *a*. Es ist $u = a^n b^m$, $vwx = b^{n-m}c^l$, $y = c^{n-l}$ mit $l \geq 1$. Da $|vx| > 0$ ist, enthält *vx* $k \geq 1$ 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 $L_4 = \{a^n b^n c^n \mid n \in \mathbf{N}, n \geq 1\}$ nicht kontextfrei.

Satz 4.4-4:

Es sei $f : \mathbf{N} \to \mathbf{N}$ eine Funktion mit einem Wachstumsverhalten, das für jedes $n \in \mathbf{N}$ durch $f(n+1) - f(n) > n$ beschrieben wird. Dann ist die Sprache $L = \{a^{f(n)} \mid n \in \mathbf{N}\}$ nicht kontextfrei.

Beweis:

Die Funktion *f* hat folgende Eigenschaften:

(i) Für jedes $n \in \mathbf{N}$ ist $f(n) \geq n$.
Dieses lässt sich durch Induktion zeigen: $f(0) \geq 0$, da $f(0) \in \mathbf{N}$ ist; mit der Induktionsvoraussetzung $f(n) \geq n$ folgt die Aussage auch für $n+1$: $f(n+1) > f(n) + n \geq 2 \cdot n \geq n + 1$.

(ii) Für jedes $n \in \mathbf{N}$ ist $f(n+1) > f(n)$.
Denn $f(n+1) > f(n) + n \geq f(n)$.

(iii) Für $n_1 < n_0$ ist $f(n_1) < f(n_1 + 1) \leq f(n_0)$.
Denn wegen $n_1 + 1 \leq n_0$ folgt aus (ii): $f(n_1) < f(n_1 + 1) \leq f(n_0)$.

Angenommen, die Sprache $L = \{a^{f(n)} \mid n \in \mathbf{N}\}$ sei kontextfrei. Dann betrachte man das Wort $z = a^{f(n_0)}$ mit n_0 wie in Satz 4.4-3. Es ist $|z| = f(n_0) \geq n_0$, so dass sich es sich in $z = uvwxy$ mit den angegebenen Eigenschaften zerlegen lässt. Insbesondere enthält *L* auch das Wort uv^2wx^2y, für dessen Länge $f(n_0) < |uv^2wx^2y| \leq f(n_0) + n_0 < f(n_0 + 1)$ gilt. Wegen Eigen-

schaft (iii) von f gibt es weder $n_1 \in \mathbf{N}$ mit $n_1 < n_0$ und $f(n_0) < f(n_1) < f(n_0+1)$ noch $n_2 \in \mathbf{N}$ mit $n_0 + 1 < n_2$ und $f(n_0) < f(n_2) < f(n_0+1)$. Daher ist uv^2wx^2y nicht in *L*. Daher ist die Annahme, *L* sei kontextfrei, falsch.

///

Satz 4.4-4 zeigt, dass die kontextsensitiven Sprachen $L_1 = \{ a^{2^i} \mid i \geq 1 \}$ und $L_2 = \{ a^{i^2} \mid i \geq 1 \}$ nicht kontextfrei sind.

Etwas anders muss man argumentieren, um nachzuweisen, dass die Sprache $L_3 = \{ a^n \mid n \text{ ist Primzahl} \}$ nicht kontextfrei ist: Angenommen L_3 sei kontextfrei. Dann wähle man eine Primzahl $n \geq \max\{ n_0, 2 \}$ mit n_0 wie in Satz 4.4-3 und $z = a^n$. Das Wort lässt sich in $z = uvwxy$ zerlegen, und $uv^k wx^k y \in L_3$ für alle $k \in \mathbf{N}$, d.h. alle Zahlen $|uv^k wx^k y|$ sind Primzahlen. Es ist $|uv^k wx^k y| = |uvwxy| + (k-1) \cdot |vx|$. Speziell für $k = |uvwxy| + 1 \geq 3$ ist $|uv^k wx^k y| = (k-1) + (k-1) \cdot |vx| = (k-1) \cdot (1 + |vx|)$. Beide Faktoren sind ≥ 2, so da die Primzahl $|uv^k wx^k y|$ in zwei nichttriviale Faktoren zerlegt wurde. Daher ist die Annahme, L_3 sei kontextfrei, falsch.

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 $O\left(|u|^3\right)$ 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

$$L_{leer\_Typ-2} = \left\{ w \,\middle|\, w \in \{0,1\}^*, VERIFIZIERE\_G\_TYP\_2(w) = \mathsf{TRUE} \text{ und } L\left(KG_w\right) = \varnothing \right\}$$

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 4.4-5:

Das Wortproblem und das Leerheitsproblem für Typ-2-Grammatiken sind entscheidbar:

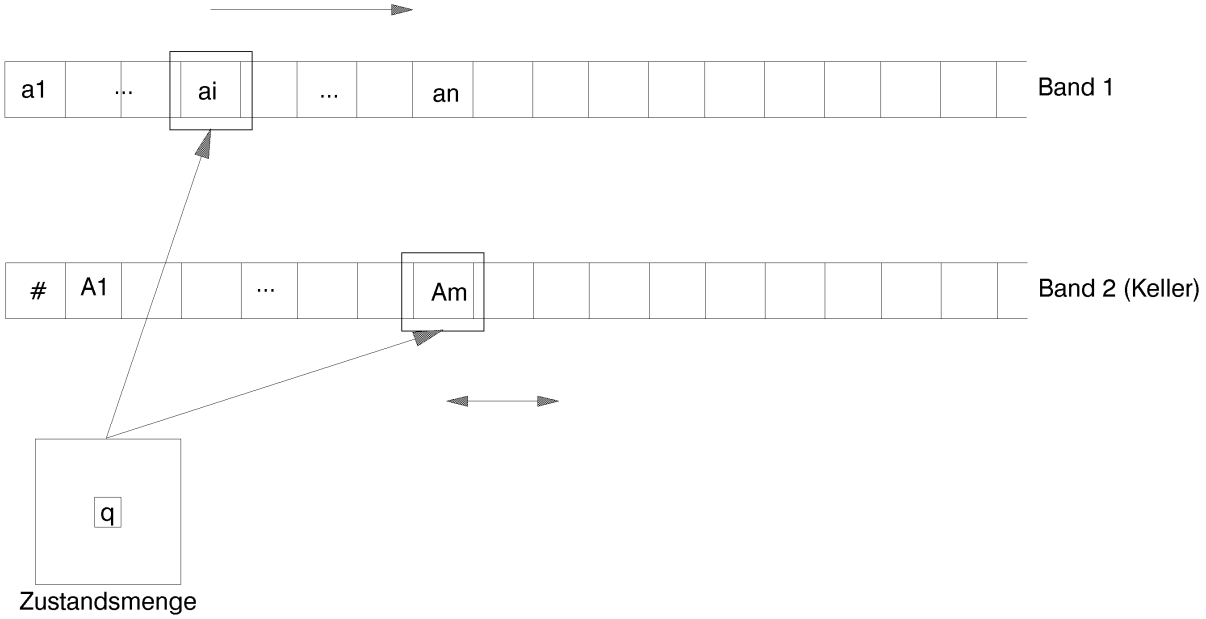
Es gibt einen auf allen Eingaben stoppenden Algorithmus**,** der bei Eingabe einer Typ-2-Grammatik $G = (\Sigma, N, S, R)$ und eines Wortes $u \in \Sigma^*$ entscheidet, ob $u \in L(G)$ ist.

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 ε-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
$NKA = (Q, \Sigma, \Gamma, \delta, q_0, \#, F)$ 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. $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathbf{P}_e(Q \times \Gamma^*)$ 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 $(\alpha, |\alpha|)$ festzuhalten, sondern es genügt die Zeichenkette α 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, α) mit $q \in Q$, $w \in \Sigma^*$ und $\alpha \in \Gamma^*$ 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 $\alpha = A_1 \ldots A_m$ befindet, und dass der Schreib/Lesekopf des 2. Bandes über A_m steht. Falls $\delta(q, a_1, A_m)$ 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

$$(q, a_1 \ldots a_n, A_1 \ldots A_m) \Rightarrow \begin{cases} (q', a_2 \ldots a_n, A_1 \ldots A_{m-1} B_1 \ldots B_k) & \text{falls } (q', B_1 \ldots B_k) \in \delta(q, a_1, A_m) \\ (q', a_1 a_2 \ldots a_n, A_1 \ldots A_{m-1} B_1 \ldots B_k) & \text{falls } (q', B_1 \ldots B_k) \in \delta(q, \varepsilon, A_m) \end{cases}$$

Entsprechend bedeutet für Konfigurationen *K* und K' die Beziehung $K \Rightarrow^* K'$:
Es gibt Konfigurationen K_0, K_1, ..., 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 $(q, \varepsilon, \varepsilon)$ mit $q \in F$.

Ein Wort $w \in \Sigma^*$ wird von *NKA* **akzeptiert**, wenn $(q_0, w, \#) \Rightarrow^* (q, \varepsilon, \varepsilon)$ mit $q \in F$ gilt. Die von *NKA* **akzeptierte Sprache** ist $L(NKA) = \{ w \mid w \in \Sigma^*, \text{und } w \text{ wird von } NKA \text{ akzeptiert} \}$.

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
$NKA = (\{ q_0, q_1, q_2 \}, \{ a, b \}, \{ A, B, \# \}, \delta, q_0, \#, \{ q_2 \})$ mit der durch folgende Tabelle festgelegten Überführungsfunktion δ:

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	ε
	b	B	q_1	ε
	ε	$\#$	q_2	ε

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

$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to Q \times \Gamma^*$
aufweist.

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 $G = (\{a,b\}, \{S\}, S, \{S \to aSb, S \to \varepsilon\})$ erzeugen.

Der deterministische Kellerautomaten *DKA* wird gegeben durch
$DKA = (\{q_0, q_1, q_2\}, \{a,b\}, \{a,b,\#\}, \delta, q_0, \#, \{q_0\})$ mit der durch folgende Tabelle festgelegten Überführungsfunktion δ:

momentaner Zustand	Symbol unter dem Kopf auf		neuer Zustand	neues Wort im Keller
	Band 1	Keller		
q_0	a	#	q_1	$\#a$
	ε	#	q_0	ε
q_1	a	a	q_1	aa
	b	a	q_2	ε
q_2	b	a	q_2	ε
	ε	#	q_0	ε

Satz 4.4-6:
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 Kel-

lerautomaten) 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 $NKA_G = (Q, \Sigma, \Gamma, \delta, q_0, \#, F)$ 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 δ, die folgendermaßen definiert ist:

(i) Für jede Erzeugungsregel $A \to \alpha$ aus R wird (z, α^R) in $\delta(z, \varepsilon, A)$ aufgenommen; hierbei ist α^R die Konkatination der Buchstaben von α in umgekehrter Reihenfolge (Spiegelung von α)

(ii) Für jedes $a \in \Sigma$ wird (z, ε) in $\delta(z, a, a)$ 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.

///

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	ε	S	z	A
	ε	A	z	bAa
	ε	A	z	c
	a	a	z	ε
	b	b	z	ε
	c	c	z	ε

Das Wort $aacbb \in L(G)$ wird durch folgenden Konfigurationsübergänge akzeptiert:

$$(z, aacbb, S) \Rightarrow (z, aacbb, A) \Rightarrow (z, aacbb, bAa) \Rightarrow (z, acbb, bA) \Rightarrow (z, acbb, bbAa)$$
$$\Rightarrow (z, cbb, bbA) \Rightarrow (z, cbb, bbc) \Rightarrow (z, bb, bb) \Rightarrow (z, b, b)$$
$$\Rightarrow (z, \varepsilon, \varepsilon).$$

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 4.4-7:

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 $A \to \varepsilon$.

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 $L_1 = \left\{ a^i b^j \mid i \in \mathbf{N},\ j \in \mathbf{N} \right\}$ aus Kapitel 4.1 regulär (rechtslinear, Typ-3-Sprache), da sie durch die Grammatik
$G_1' = \left(\{a, b\}, \{S, B\}, S, \{S \to \varepsilon, S \to aS, S \to bB, B \to bB, B \to \varepsilon\} \right)$ 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) ε ist ein regulärer Ausdruck, der die reguläre Sprache $\{\varepsilon\}$ 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.

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.[10]

[10] Der Beweis dieses Satzes findet man in Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.: **The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing**, Addison-Wesley, 1972.

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
$NEA = (Q, \Sigma, \delta, q_0, F)$ mit:

1. Q ist eine endliche nichtleere Menge: die **Zustandsmenge**
2. Σ ist eine endliche nichtleere Menge: das **Eingabealphabet**
3. $\delta : Q \times \Sigma \to \mathbf{P}(Q)$ 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 δ die Form $\delta : Q \times \Sigma \to Q$ aufweist.

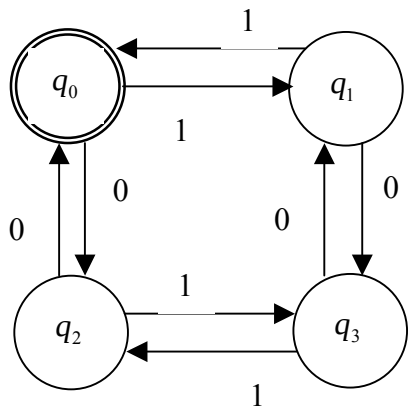
Ein deterministischer bzw. nichtdeterministischer endlicher Automat lässt sich auch in Form eines endlichen gerichteten Graphen G mit markierten Kanten als **Transitionsdiagramm** darstellen. Die Knotenmenge von G ist Q. Ist $z' = \delta(z, a)$ bzw. im nichtdeterministischen Fall $z' \in \delta(z, a)$, so wird eine mit a markierte Kante in G aufgenommen. Die Menge der den Endzuständen entsprechenden Knoten werden explizit markiert.

Ein deterministischer endlicher Automat

Es sei $DEA = (Q, \Sigma, \delta, q_0, F)$ der deterministische endliche Automat mit $Q = \{ q_0, q_1, q_2, q_3 \}$, $\Sigma = \{ 0,1 \}$, $F = \{ q_0 \}$, δ 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

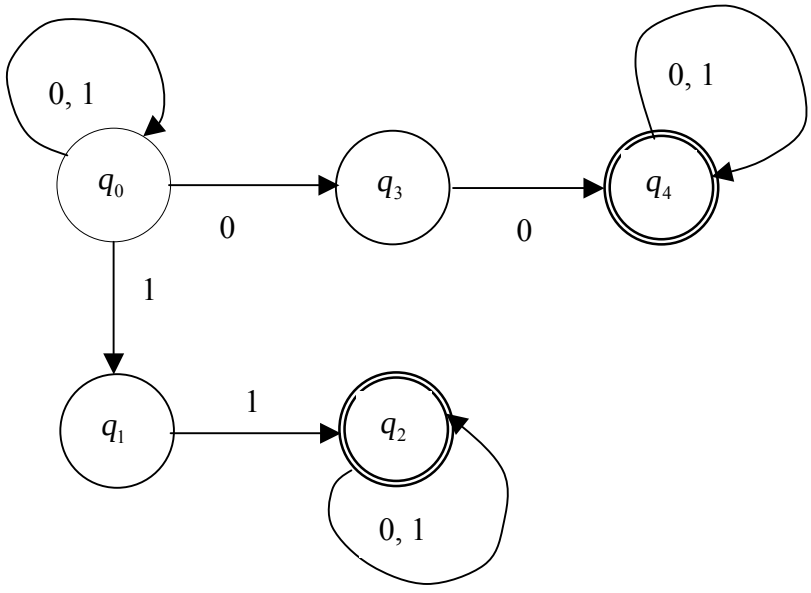
$L(DEA) = \{ w \mid w \in \{ 0,1 \}^* \text{ und } w \text{ enthält eine gerade Anzahl von Zeichen 0 und Zeichen 1} \}$.

Ein nichtdeterministischer endlicher Automat

$NEA = (Q, \Sigma, \delta, q_0, F)$ mit $Q = \{ q_0, q_1, q_2, q_3, q_4 \}$, $\Sigma = \{0,1\}$, $F = \{ q_2, q_4 \}$, δ 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 $(q_0, w) \Rightarrow^* (q, \varepsilon)$ 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 nichtdeterministischen endlichen Automaten $NEA = (Q, \Sigma, \delta, q_0, F)$ ein deterministischer endlicher Automat $DEA = (Q', \Sigma, \delta', q_0', F')$ anzugeben, der dieselbe Sprache akzeptiert. *DEA* wird definiert durch:

$Q' = \mathbf{P}(Q)$, $q_0' = \{q_0\}$, $\delta'(Z', a) = \bigcup_{z \in Z'} \delta(z, a)$ 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 $NEA = (Q', \Sigma, \delta', q_0', F')$ ein nichtdeterministischer endlicher Automat. Dann gibt es nach Satz 4.5-2 einen deterministischen endlichen Automaten $DEA = (Q, \Sigma, \delta, q_0, F)$ 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 $\delta(q,a) = q'$ ist; für $q \in F$ enthält *R* außerdem die Regel $q \to \varepsilon$.

Es sei umgekehrt $G = (\Sigma, N, S, R)$ eine rechtslineare Grammatik. Dann erkennt der folgende Automat $NEA = (Q, \Sigma, \delta, q_0, F)$ die Sprache $L(G)$:

Es sei *A* ein nichtterminales Symbol mit $A \notin N \cup \Sigma$. Es ist

$Q = N \cup \{A\}$, $q_0 = S$, $F = \begin{cases} \{A\} & \text{für } \varepsilon \notin L(G) \\ \{S, A\} & \text{für } \varepsilon \in L(G) \end{cases}$, die Überführungsfunktion $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathbf{P}(Q)$ wird definiert durch:

$\delta(B,a)$ enthält genau dann *C*, wenn *R* die Regel $B \to aC$ enthält; für jede Regel der Form $B \to \varepsilon$ wird $\delta(B,\varepsilon) = \{A\}$ gesetzt. Der Automat $NEA = (Q, \Sigma, \delta, q_0, F)$ erfüllt noch nicht die Definition eines endlichen Automaten; denn diese lässt keine ε-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 $NEA' = (Q', \Sigma, \delta', q_0', F')$ modifizieren kann, so dass dessen Überführungsfunktion $\delta' : Q' \times \Sigma \to \mathbf{P}(Q')$ 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 $L = \left\{ w \mid w \in \{0,1\}^* \text{ und } w = a_1 ... a_{n-1} a_n = a_n a_{n-1} ... a_1 \right\} \cup \{\varepsilon\}$ nicht regulär ist; L ist jedoch kontextfrei[11]. Es sei $n_0 > 0$ der Wert Satz 4.5-4 und n eine gerade Zahl mit $n > n_0$. 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

[11] Eine kontextfreie Grammatik, die *L* erzeugt, ist
$G = (\{0,1\}, \{S\}, S, \{S \to \varepsilon, S \to 0S0, S \to 1S1\})$.

(00) aufeinanderfolgen, in der Mitte von *z* liegt. Das Teilwort *uv* ist wegen $|uv| \le n_0 < n$ ein echtes Anfangsstück von $\underbrace{1010...10}_{n \text{ Zeichen}}$.

1. Fall: $v = 1$: Das Wort uv^2w ist wegen (iii) 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: $v = 0$: Da *uv* ein echtes Teilwort von $\underbrace{1010...10}_{n \text{ Zeichen}}$ ist, stellt *v* nicht die letzte 0 in $\underbrace{1010...10}_{n \text{ Zeichen}}$ dar. Das Wort uv^3w ist wegen (iii) in *L* und enthält genau einmal drei und genau einmal zwei aufeinanderfolgende Zeichen 0. Damit uv^3w ein Palindrom ist, müssten die drei Zeichen 0 oder die zwei Zeichen 0 in der Mitte des Worts liegen. Liegt die Teilzeichenfolge 000 in der Mitte, dann korrespondiert in uv^3w zu 00 keine entsprechende Zeichenfolge links der Mitte; liegt die Teilzeichenfolge 00 in der Mitte, dann korrespondiert in uv^3w zu 000 keine entsprechende Zeichenfolge rechts der Mitte. Daher ist $uv^3w \notin L$.

3. Fall: $|v| > 1$ und $v = 0...1$: Dann enthält Wort uv^2w die Zeichenfolge 00 (die ursprüngliche Mitte) genau einmal, die jedoch wegen (ii) nicht mehr in der Mitte liegt. Links und rechts davon stehen nur Teilzeichenfolgen der Form 10. Daher ist $uv^2w \notin L$.

4. Fall: $|v| > 1$ und $v = 1...1$: Dann enthält Wort uv^2w die Teilzeichenfolgen 00 (die ursprüngliche Mitte) und 11 genau einmal, die beide in der Mitte liegen müssten, damit uv^2w ein Palindrom ist. Daher ist $uv^2w \notin L$.

5. Fall: $|v| > 1$ und $v = v'0$. Hier kann man ähnlich wie im 3. bzw. 4.Fall argumentieren.

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 leicht 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

$$L_{Wort\_Typ-3} = \left\{ u\#w \,\middle|\, u \in \Sigma^*, w \in \{0,1\}^*, VERIFIZIERE\_G\_TYP\_3(w) = \mathsf{TRUE} \text{ und } u \in L(KG_w) \right\}$$

entscheidbar ist. In vereinfachter Darstellung wird also danach gefragt, ob es einen auf allen Eingaben stoppenden Algorithmus gibt, der bei Eingabe (der Kodierung) einer 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 $DEA = (Q, \Sigma, \delta, q_0, F)$ ein deterministischer endlicher Automat mit $L(DEA) = L(G)$ ist:

Eingabe: Ein deterministischer endlicher Automat $DEA = (Q, \Sigma, \delta, q_0, F)$ 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,δ,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 := δ(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

$$L_{leer\_Typ-3} = \left\{ w \;\middle|\; w \in \{0,1\}^*, VERIFIZIERE\_G\_TYP\_3(w) = \text{TRUE und } L(KG_w) = \varnothing \right\}$$

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).

Der folgende Satz wird in den anschließenden Überlegungen zur Kontextfreiheit von Programmiersprachen benötigt.

Satz 4.5-6:

Ist L eine kontextfreie Sprache über dem Alphabet Σ und R eine reguläre Menge über Σ, dann ist $L \cap R$ kontextfrei.
Ist L dabei deterministisch kontextfrei, dann ist auch $L \cap R$ deterministisch kontextfrei.

Die Klasse der kontextfreien Sprachen über einem endlichen Alphabet Σ ist also abgeschlossen bezüglich Schnitten mit regulären Mengen.

Beweis:

Ist L eine kontextfreie Sprache über dem Alphabet Σ, dann gibt es einen nichtdeterministischen Kellerautomaten $K = (Q_K, \Sigma, \Gamma, \delta_K, q_{K,0}, \#, F_K)$ mit $L = L(K)$. Zu R gibt es einen deterministischen endlichen Automaten $E = (Q_E, \Sigma, \delta_E, q_{E,0}, F_E)$ mit $R = L(E)$. Der folgende nichtdeterministische Kellerautomat $NKA = (Q_K \times Q_E, \Sigma, \Gamma, \delta, (q_{K,0}, q_{E,0}), \#, F_K \times F_E)$ ist eine Parallelschaltung von K und E, und es gilt $L(NKA) = L \cap R$; hierbei ist die Überführungsfunktion δ definiert durch:
$\delta((p,q),a,\gamma)$ enthält genau dann $((p',q'),\gamma')$, wenn $(p',\gamma') \in \delta_K(p,a,\gamma)$ und $q' = \delta_E(q,a)$ gilt.

Ist L deterministisch kontextfrei, dann nimmt man anstelle von *NKA* einen deterministischen Kellerautomaten. Die beschriebene Parallelschaltung ist dann ein deterministischer Kellerautomat.

///

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. Die Menge $R = \{ a^i b^j a^k b^l \mid i \in \mathbf{N},\ j \in \mathbf{N}, k \in \mathbf{N}, l \in \mathbf{N} \}$ ist regulär. Daher ist $L_2 = \{ ww \mid w \in \{a,b\}^+ \}$ nicht kontextfrei; denn sonst wäre mit Satz 4.5-6 wegen $L_1 = L_2 \cap R$ auch L_1 kontextfrei. Es seien c_1, ..., c_n neue paarweise verschiedene Buchstaben. Dann 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? Denn wäre L_3 kontextfrei, dann gäbe es eine kontextfreie Grammatik G mit $L_3 = L(G)$. Ersetzt man in jeder Ableitungsregel $A \to \alpha$ in der Zeichenkette α vorkommende Symbole c_1, ..., c_n jeweils durch das leere Wort, so erhält man eine kontextfreie Grammatik G' mit $L(G') = L_2$, d.h. L_2 wäre kontextfrei.

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

$R = \{ \textbf{PROGRAM A VAR } w_1 : \textbf{INTEGER}; \textbf{BEGIN } w_2 := 1; \textbf{END.} \mid w_i \in \{a, b\}^+, i = 1, 2 \}.$

Wäre L_P, die Menge aller korrekten Pascal-Programme, kontextfrei, so wäre auch $L_4 = L_P \cap R$ kontextfrei. L_4 ist ein Spezialfall von L_3 und damit nicht kontextfrei.

Offensichtlich lassen sich nicht alle Teile einer Programmiersprache durch kontextfreie Grammatiken beschreiben. Im Compilerbau, in dem Übersetzer für kontextfreie Sprachen erzeugt werden, wird dieser Tatsache durch zusätzliche semantische Regeln Rechnung getragen.

4.6 Eigenschaften im tabellarischen Überblick

Die folgenden Tabellen stellen wichtige Eigenschaften der behandelten Sprachklassen zusammen. Einige der aufgeführten Eigenschaften wurden in den vorherigen Kapiteln bewiesen; die Abschlusseigenschaften bezüglich Schnitt-, Vereinigung- und Komplementbildung werden erläutert; für die Nachweise der übrigen Abschlusseigenschaften wird 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 (Satz 2.5-1)
Linear beschränkter Automat	?
Kellerautomat	nein (siehe Abschlusseigenschaften)
Endlicher Automat	ja (Satz 4.5-2)

Abschlusseigenschaften (die Zahlenangaben verweisen auf die nachfolgenden Bemerkungen)

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 (Satz 2.1-2)	ja (Satz 2.1-2)	nein (Satz 3.2-1)	ja	ja
Typ 1	ja (Satz 2.1-2 mit LBA)	ja (Satz 2.1-2 mit LBA)	ja (siehe Text in Kapitel 4.3)	ja	ja
Typ 2	nein (1)	ja (2)	nein (3)	ja	ja
det. kontextfrei	nein (4)	nein (5)	ja (6)	nein	nein
Typ 3	ja (7)	ja (8)	ja (9)	ja	ja

(In den vorherigen Kapiteln wurden die Produkt- und Sternbildung von Mengen nicht behandelt. Die Ergebnisse diesbezüglichen können in der angegebenen Literatur nachgelesen werden.)

Zu den Zahlenangaben:

(1) Die Sprachen $L_1 = \left\{ a^n b^n c^i \mid n \in \mathbf{N}, i \in \mathbf{N} \right\}$ und $L_2 = \left\{ a^i b^n c^n \mid n \in \mathbf{N}, i \in \mathbf{N} \right\}$ sind kontextfrei, nicht aber $L_1 \cap L_2 = \left\{ a^n b^n c^n \mid n \in \mathbf{N} \right\}$ (siehe Kapitel 4.4).

(2) Es seien L_1 und L_2 kontextfreie Sprachen, die jeweils von den kontextfreien Grammatiken $G_1 = (\Sigma, N_1, S_1, R_1)$ bzw. $G_2 = (\Sigma, N_2, S_2, R_2)$ erzeugt werden. Man kann annehmen, dass $N_1 \cap N_2 = \varnothing$ ist, ansonsten benennt man die Nichtterminalzeichen entsprechend um. Es sei S ein neues nichtterminales Symbol. Dann wird $L_1 \cup L_2$ durch die kontextfreie Grammatik $G = (\Sigma, N_1 \cup N_2 \cup \{S\}, S, R_1 \cup R_2 \cup \{S \to S_1, S \to S_2\})$ erzeugt.

(3) Angenommen, die Klasse der kontextfreien Sprachen wären gegen Komplementbildung abgeschlossen. Dann wäre sie auch gegenüber Schnittbildung abgeschlossen, da der Schnitt zweier Mengen durch Vereinigung- und Komplementbildung ausgedrückt werden kann: $L_1 \cap L_2 = \Sigma^* \setminus \left(\left(\Sigma^* \setminus L_1 \right) \cup \left(\Sigma^* \setminus L_2 \right) \right)$.

(4) Die Sprachen $L_1 = \left\{ a^n b^n c^i \mid n \in \mathbf{N}, i \in \mathbf{N} \right\}$ und $L_2 = \left\{ a^i b^n c^n \mid n \in \mathbf{N}, i \in \mathbf{N} \right\}$ aus (1) sind jeweils deterministisch kontextfrei (siehe Kapitel 4.4).

(5) Die Konstruktion aus (2) baut in den ersten Ableitungsschritt einer Ableitung $S \Rightarrow^* w$ Nichtdeterminismus ein; denn eine der beiden Produktionen $S \to S_1$ bzw. $S \to S_2$ wird nichtdeterministisch ausgewählt. Setzt man die Gültigkeit der Aussage voraus, dass die Klasse der deterministisch kontextfreien Sprachen abgeschlossen gegenüber Komplementbildung ist und nimmt man an, dass diese Klasse gegenüber Vereinigungsbildung abgeschlossen ist, dann wäre sie auch gegenüber Schnittbildung abgeschlossen, denn $L_1 \cap L_2 = \Sigma^* \setminus \left(\left(\Sigma^* \setminus L_1 \right) \cup \left(\Sigma^* \setminus L_2 \right) \right)$.

(6) Hier erfolgt die Argumentation mit Hilfe eines deterministischen Kellerautomaten. Details finden sich in der angegebenen Literatur.

(7) Wird L_1 vom deterministischen endlichen Automaten $E_1 = \left(Q_1, \Sigma, \delta_1, q_{1,0}, F_1 \right)$ und L_2 vom deterministischen endlichen Automaten $E_2 = \left(Q_2, \Sigma, \delta_2, q_{2,0}, F_2 \right)$ erkannt, dann ist $L_1 \cap L_2 = L(E)$ mit dem deterministischen endlichen Automaten $E = \left(Q_1 \times Q_2, \Sigma, \delta, \left(q_{1,0}, q_{2,0} \right), F_1 \times F_2 \right)$, dessen Überführungsfunktion durch $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ definiert ist.

(8) Setzt man die Gültigkeit der Aussage voraus, dass die Klasse der regulären Sprachen abgeschlossen gegenüber Komplementbildung ist, dann ist mit zwei regulären Sprachen L_1 und L_2 wegen $L_1 \cup L_2 = \Sigma^* \setminus \left(\left(\Sigma^* \setminus L_1 \right) \cap \left(\Sigma^* \setminus L_2 \right) \right)$ auch deren Vereinigung regulär.

(9) Es sei L eine reguläre Menge und $E = \left(Q, \Sigma, \delta, q_0, F \right)$ ein deterministischer endlicher Automat mit $L(E) = L$. Dann gilt für den deterministischen endlichen Automaten $E' = \left(Q, \Sigma, \delta, q_0, Q \setminus F \right)$: $L(E') = \Sigma^* \setminus L$.

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.

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 Loga-

rithmusfunktion wegen ihres langsamen Wachstums den Wert stark vergrößern, während bei der schnell anwachsenden Exponentialfunktion nur eine kleine additive konstante Steigerung 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 $h_i(n_i) = y_0$	$\overline{n}_i$ mit $h_i(\overline{n}_i) = 10 \cdot y_0$
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	$n_5 + 3{,}322$

Im vorliegenden Kapitel sollen die Gründe dafür aufgezeigt werden, dass für viele in der Praxis auftretenden Optimierungs- und Entscheidungsprobleme bisher keine effizienten Lösungsalgorithmen gefunden wurden. **Effizienz** bedeutet in diesem Zusammenhang die Existenz eines deterministischen polynomiell zeitbeschränkten Lösungsverfahrens. Ob es jemals derartige Lösungsverfahren geben wird, ist ungewiss.

In Berechnungsmodell der Turingmaschine (Kapitel 2.1) wird das Laufzeitverhalten eines Verfahrens in Abhängigkeit von der Anzahl Zeichen, d.h. der Länge der Eingabeinstanz gemessen. Wird das Programmiersprachenmodell (Kapitel 2.3) zugrundegelegt und besteht eine Eingabeinstanz x aus einer einzelnen ganzen Zahl, wird als Maßstab $size(x)$ zur Bestimmung des Laufzeitverhaltens die Anzahl signifikanter Bits genommen, die benötigt werden, um die Zahl im Binärsystem darzustellen, d.h.

$$size(x) = |bin(x)| = \begin{cases} \lfloor \log_2(|x|) \rfloor + 1 & \text{für } x \neq 0 \\ 1 & \text{für } x = 0 \end{cases}.$$

Ist man nur an der Größenordnung des Laufzeitverhaltens interessiert, kann statt des Binärsystems auch ein anderes Stellenwertsystem verwendet werden, da sich die Anzahlen signifikanter Bits in den unterschiedlichen Stellenwertsystemen lediglich jeweils um konstante Faktoren unterscheiden.

Ist die Eingabeinstanz ein Graph $G = (V, E)$ mit der Knotenmenge $V = \{v_1, ..., v_n\}$ und der Kantenmenge $E \subseteq V \times V$, so kann man diese auf verschiedene Weisen darstellen, z.B. als Knoten- und Kantenliste, als Liste der jeweiligen Nachfolger oder als Adjazenzmatrix. Dabei ergeben sich unterschiedliche Anzahlen an Zeichen, d.h. unterschiedliche Längen der Eingabeinstanz. Beispielsweise lässt sich der ungerichtete Graph $G = (V, E)$ mit $V = \{v_1, v_2, v_3, v_4\}$ und $E = \{(v_1, v_2), (v_2, v_3)\}$ folgendermaßen darstellen (hierbei ist zu beachten, dass mit einer

Kante $(v_i, v_j) \in E$ implizit auch $(v_j, v_i) \in E$ ist; diese Tatsache hat Einfluss auf die Darstellung als Nachfolgerliste und Adjazenzmatrix):

	Darstellung	Länge
Knoten- und Kantenliste	$v1v2v3v4(v1v2)(v2v3)$	20 Zeichen
Nachfolgerliste	$(v2)(v1v3)(v2)(\ )$	16 Zeichen
Adjazenzmatrix	0100/1010/0100/0000	19 Zeichen

Für einen ungerichteten Graphen $G = (V, E)$ mit der Knotenmenge $V = \{v_1, ..., v_n\}$ und der Kantenmenge $E \subseteq V \times V$ mit $|E| = m$ ergeben sich die in folgender Tabelle zusammengestellten Abschätzungen für die Anzahl der Zeichen bei obigen unterschiedlichen Darstellungsformen einer Eingabeinstanz. Zu beachten ist dabei die Tatsache, dass zur Darstellung der Knotenindizes (Knotennumern) im Binärsystem jeweils bis zu $\lfloor \log_2(n) \rfloor + 1$ zusätzliche Binärzeichen pro Knoten benötigt werden. Die untere Grenze berücksichtigt nicht die Indizierung der Knoten (für jede Knotennummer steht ein einziges Zeichen).

	untere Grenze an Zeichen	obere Grenze an Zeichen
Knoten- und Kantenliste	$2 \cdot n + 6 \cdot m$	$2 \cdot n + 6 \cdot m + (n + 2 \cdot m) \cdot (\lfloor \log_2(n) \rfloor + 1)$
Nachfolgerliste	$2 \cdot n + 4 \cdot m$	$2 \cdot n + 4 \cdot m + 2 \cdot m \cdot (\lfloor \log_2(n) \rfloor + 1)$
Adjazenzmatrix	$n^2 + n - 1$	$n^2 + n - 1$

Da $m \leq n^2$ gilt, sind die einzelnen Längen polynomiell miteinander verknüpft, so dass ein polynomiell zeitbeschränktes Entscheidungsverfahren bei Eingabe einer Instanz, bei der eine spezielle Darstellungsform gewählt wurde, polynomiell zeitbeschränkt bleibt, wenn eine andere Darstellungsform gewählt wird, wobei jeweils als Maßstab die Anzahl der Zeichen der Eingabeinstanz zugrundegelegt wird.

In den Beispielen der folgenden Kapitel treten noch andere Typen von Eingabeinstanzen auf (Boolesche Ausdrücke, gewichtete und ungewichtete Graphen, Matrizen, Graphen und Zahlen usw.). Eine „vernünftige“ Definition der **Problemgröße** $size(x)$ einer Eingabeinstanz x legt einen Wert fest, der polynomiell verknüpft ist mit Anzahl der Zeichen (Binärwerte, Zeichen über einem endlichen Alphabet), die benötigt werden, um eine Eingabeinstanz darzustellen. Meist ist $size(x)$ proportional zu dieser Zeichenanzahl. Die folgende Zusammenstellung gibt Definitionen der jeweiligen Problemgröße *size* aus unterschiedlich strukturierten Anwendungsbereichen. Dabei werden in einigen Fällen Größenordnungen für eine untere und obere Abschätzung von $size(x)$ gegeben und damit ein „feinerer“ bzw. „gröberer“ Maßstab zur Messung der Komplexität eines Entscheidungsverfahrens bereitgestellt. Werte in diesen Grenzen sind ebenfalls als mögliche Definitionen für $size(x)$ geeignet.

Instanz: x

$x \in \mathbf{N}$ oder $x \in \mathbf{Z}$;

$$size(x) = \begin{cases} \lfloor \log_2(|x|) \rfloor + 1 & \text{für } x \neq 0 \\ 1 & \text{für } x = 0 \end{cases}.$$

Es ist $size(x) \in O(\log(x))$.

Instanz: $x = [x_1, ..., x_n]$

$x_i \in \mathbf{N}$ oder $x_i \in \mathbf{Z}$ für $i = 1, ..., n$;

$$size(x) = \sum_{i=1}^{n} (size(x_i) + 1) + 1.$$

Der Wert $size(x)$ gibt damit die Anzahl der Bits an, um die Zahlenwerte darzustellen, die in x vorkommen, einschließlich der die Zahlenfolge $x_1, ..., x_n$ umfassenden eckigen Klammern und Trennsymbolen zwischen den Werten $x_1, ..., x_n$. Häufig wird auch als Größe der Eingabe der Wert $n \cdot (size(\max\{x_1, ..., x_n\}) + 1) + 1$ verwendet. Dieser Maßstab ist wohl etwas gröber als $size(x)$. Für Analysezwecke eignet er sich jedoch, da in einer Komplexitätsbetrachtung meist die Abschätzung $size(x) \in O(n \cdot size(\max\{x_1, ..., x_n\}))$ verwendet wird. Zudem gibt es Eingaben x, deren Zahlen in der Folge $x_1, ..., x_n$ zusammen genau $n \cdot size(\max\{x_1, ..., x_n\})$ viele Bits belegen.

Der hier beschriebene Typ einer Eingabeinstanz eignet sich auch zur Darstellung rationaler Zahlen $r = p/q$ mit $p \in \mathbf{Z}$, $q \in \mathbf{N}$ und $q \neq 0$ in der Form $[p, q]$.

Instanz: F

F ist ein Boolescher Ausdruck Variablenmenge $V = \{x_1, ..., x_n\}$ und m Junktoren;

$size(F) = |F|$ = Anzahl der Zeichen in F.

Wie man durch Induktion über den Aufbau von F zeigen kann, gilt $n \leq m + 1$. Zu jedem Junktor in F gehört ein Klammerpaar, also enthält F (bei Beibehaltung der kompletten Klammerung) $2 \cdot m$ Klammern. Werden die n Variablen mit 1, ..., n durchnumeriert, so benötgt man zur Notation von x_i (einschließlich des Variablenzeichens x) $\lfloor \log_2(i) \rfloor + 2$ Zeichen. Insgesamt ist die Anzahl der Zeichen in F, d.h. die Länge von F, durch einen Wert der Ordnung $O(n \cdot \log(n) + m)$ beschränkt. Notiert man Variablen nicht in Form des Variablenzeichens x, versehen mit einem Index als Binärzahl, sondern wählt für jede Variable ein eigenes Zeichen, so ist die Anzahl der Zeichen in F durch einen Wert der Ordnung $O(n + m)$ beschränkt.

Instanz: G

$G = (V, E)$ ist ein ungerichteter oder gerichteter Graph mit der Knotenmenge V und der Kantenmenge $E \subseteq V \times V$;

$size(G)$ = Anzahl der Zeichen, um *G* darzustellen.

Je nach Darstellungsform (siehe oben) liegt $size(G)$ zwischen einem Wert der Ordnung $O(|V|+|E|)$ und $O(|V|^2)$.

Instanz: $[G,k]$

$G=(V,E)$ ist ein gerichteter oder ungerichteter Graph mit der Knotenmenge *V* und der Kantenmenge $E \subseteq V \times V$, und *k* ist eine Zahl (natürliche oder ganze Zahl, rationale Zahl in Paardarstellung oder reelle Zahl in Approximation durch eine rationale Zahl oder als Gleitpunktzahl);

$size([G,k])$ = Anzahl der Zeichen, um $[G,k]$ darzustellen.

Je nach Darstellungsform liegt $size([G,k])$ zwischen einem Wert der Ordnung $O(|V|+|E|+\log(k))$ und $O(|V|^2+\log(k))$.

Instanz: $[(V,E,w),k]$

$G=(V,E,w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V=\{v_1,...,v_n\}$ 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; $k \in \mathbf{R}_{>0}$;

$size([(V,E,w),k])$ = Anzahl der Zeichen, um $[(V,E,w),k]$ darzustellen.

Die Darstellung des Gewichts $w(v_i,v_j)$ einer gewichteten Kante $(v_i,v_j,w(v_i,v_j))$ erfordert $size(w(v_i,v_j))$ Zeichen. Im Falle natürlichzahliger oder ganzzahliger Gewichte ist $size(w(v_i,v_j)) \in O(\log(w(v_i,v_j)))$. Je nach Darstellungsform liegt $size([(V,E,w),k])$ dann zwischen einem Wert der Ordnung

$O\left(|V|+|E|+\sum_{e\in E}\log(w(e))+\log(k)\right)$ und einem Wert der Ordnung

$O(|V|^2 \cdot A+\log(k))$ mit $A=\lfloor \log_2(\max\{w(e) \mid e \in E\})\rfloor+1$.

Instanz: $[A,\vec{b},\vec{z}]$

$A \in \mathbf{Z}^{m \cdot n}$ ist eine ganzzahlige Matrix mit *m* Zeilen und *n* Spalten, $\vec{b} \in \mathbf{Z}^m$ ist ein ganzzahliger Vektor, $\vec{z}$ ist ein Spaltenvektor von *n* Variablen, die nur die Werte 0 oder 1 annehmen können;

$size([A,\vec{b},\vec{z}])$ = Anzahl der Zeichen, um $[A,\vec{b},\vec{z}]$ darzustellen.

Eine untere Schranke für $size([A,\vec{b},\vec{z}])$ ist ein Wert der Ordnung $O\left(\sum_{a_{i,j}\in A} size(a_{i,j})+\sum_{b_j\in \vec{b}} size(b_j)+n\right)$, eine obere Schranke ist ein Wert der Ordnung

$O(M \cdot m \cdot (n+1) + n)$, wobei M die maximale Anzahl an Bits ist, die jeweils in den Binärdarstellungen der in A und $\vec{b}$ vorkommenden Zahlen benötigt wird.

5.1 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.

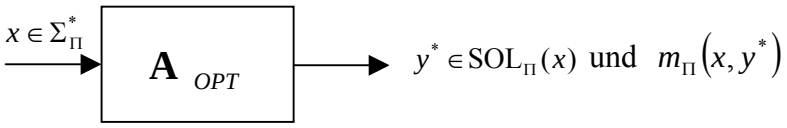
Ein Π Optimierungsproblem mit einer reellwertigen Zielfunktion wird in Kapitel 1.2 wie folgt beschrieben:

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.

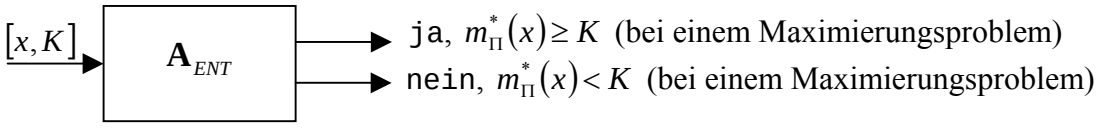
Das zu Π **zugehörige Entscheidungsproblem** Π_{ENT} wird definiert durch

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 5.1-1:

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.

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**; die Gewichtsfunktion wird durch die Festlegung $w((v_i,v_j))=\infty$ für $(v_i,v_j)\notin E$ auf $V\times V$ erweitert.
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 $m(G,T^*)$.

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 $w:E\to\mathbf{N}$ gibt jeder Kante $e\in E$ ein natürlichzahliges Gewicht; die Gewichtsfunktion wird durch die Festlegung $w((v_i,v_j))=\infty$ für $(v_i,v_j)\notin E$ auf $V\times V$ erweitert;

$K\in\mathbf{N}$.

Lösung: Entscheidung „ja“, falls es eine Tour durch G gibt mit minimalen Kosten $m^*(G)\le K$ 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 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; ; die Gewichtsfunktion wird durch die Festlegung $w((v_i,v_j))=\infty$ für $(v_i,v_j)\notin E$ auf $V\times V$ erweitert.

```
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 };
```

Es sei $m = \sum_{e \in E} w(e)$. Dann sind in obigem Verfahren höchstens $\lceil \log_2(m+1) \rceil$ viele Aufrufe des Entscheidungsverfahrens $\mathbf{A}_{TSPENT}([G, t])$ erforderlich (Binärsuche). Es gilt:

$\lceil \log_2(m+1) \rceil = \lfloor \log_2(m) \rfloor + 1 \le \log_2(m) + 1 \le \sum_{e \in E} \log_2(w(e)) + 1$.

Definiert man die Problemgröße $size(G)$ des Graphen $G = (V, E, w)$ als einen Wert der Ordnung $O\left(|V| + |E| + \sum_{e \in E} \log(w(e)) \right)$, dann gilt: $\lceil \log_2(m+1) \rceil \in O(size(G))$.

Definiert man die Problemgröße $size(G)$ des Graphen $G = (V, E, w)$ als einen Wert der Ordnung $O(|V|^2 \cdot A)$ mit $A = \lfloor \log_2(\max\{ w(e) \mid e \in E \}) \rfloor + 1$, dann gilt wegen

$\lceil \log_2(m+1) \rceil \le \sum_{e \in E} \log_2(w(e)) + 1 \le \sum_{e \in E} (\log_2(\max\{ w(e) \mid e \in E \})) + 1 \le |V|^2 \cdot A + 1$

ebenfalls $\lceil \log_2(m+1) \rceil \in O(size(G))$.

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 Entscheidungsproblem. In diesem Fall ist das Entscheidungsproblem genauso schwer zu lösen wie das Optimierungsproblem.

5.2 Komplexitätsklassen

Zu einem Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ gebe es einen deterministischen Algorithmus $\mathbf{A}_{L_\Pi}$, der für jede Eingabe $x \in \Sigma_\Pi^*$ mit $size(x) = n$ die `ja`/`nein`-Entscheidung $\mathbf{A}_{L_\Pi}(x)$ nach einer Anzahl von Schritten getroffen, die in $O(f(n))$ liegt. Dann sagt man, L_Π gehört zur Klasse $TIME(f(n))$ bzw. $L_\Pi \in TIME(f(n))$.

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 deterministischen Algorithmus $\mathbf{A}_{L_\Pi}$ gibt, der für jede Eingabe $x \in \Sigma_\Pi^*$ mit $size(x) = n$ zum Finden der `ja`/`nein`-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 $L_4 = \left\{ a^n b^n c^n \mid n \in \mathbf{N}, n \geq 1 \right\}$ aus Kapitel 4.1 zeigt, die in **P** liegt, aber nicht kontextfrei ist.

- die Klasse der Entscheidungsprobleme, die mit einem Speicherplatzbedarf deterministisch gelöst werden können, der proportional zu einem Polynom in der Größe der Eingabe ist:

  $$\mathbf{PSPACE} = \bigcup_{k=0}^{\infty} SPACE\left(n^k\right)$$

- die Klasse der Entscheidungsprobleme, die in einer Zeit gelöst werden können, die proportional zu einer Exponentialfunktion in der Größe der Eingabe ist:

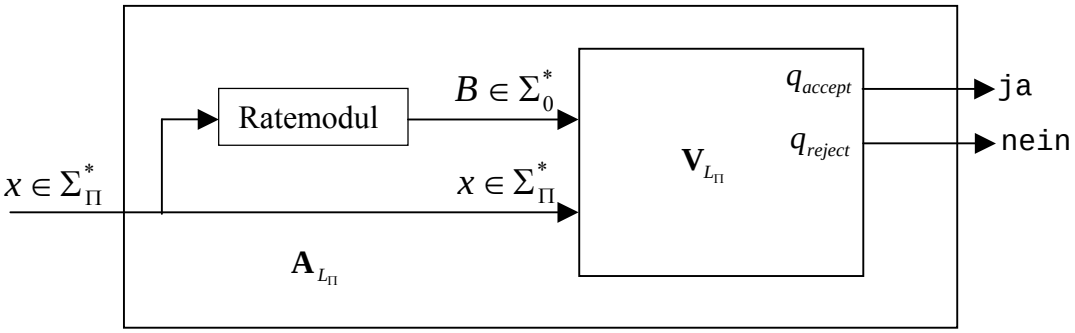
  $$\mathbf{EXP} = \bigcup_{k=0}^{\infty} TIME\left(2^{n^k}\right).$$

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 $TM = \left(Q, \Sigma, I, \delta, b, q_0, q_{accept}\right)$ in zunächst unterschiedlichen Modell-Sichtweisen definiert. Entweder wird die Überführungsfunktion als partielle Funktion der Form $\delta : Q \times \Sigma^k \to \mathbf{P}\left(Q \times \left(\Sigma \times \{L, R, S\}\right)^k\right)$ 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 $\delta : Q \times \Sigma^k \to Q \times \left(\Sigma \times \{L, R, S\}\right)^k$ 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 $\mathbf{A}_{L_\Pi}$ für ein Entscheidungsproblem** Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ 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:



Für $x \in \Sigma_\Pi^*$ wird folgende Entscheidung getroffen:

$x \in L_\Pi$ genau dann, wenn 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;

$x \notin L_\Pi$ genau dann, wenn für jeden Beweis $B \in \Sigma_0^*$ gilt: $\mathbf{V}_{L_\Pi}$ stoppt mit $\mathbf{V}_{L_\Pi}(x, B) = \texttt{nein}$.

Zu beachten ist, dass die Laufzeit des nichtdeterministischen Algorithmus in Abhängigkeit von der Größe von $x \in \Sigma_\Pi^*$ gemessen wird. Ist der nichtdeterministische Algorithmus $f(n)$-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 $\leq C \cdot f(n)$, denn das Ratemodul wird nur einmal durchlaufen, 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:

$$\mathbf{NP} = \bigcup_{k=0}^{\infty} NTIME\left(n^k\right)$$

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

$\mathbf{P} = \mathbf{NP}$ oder $\mathbf{P} \neq \mathbf{NP}$

ist bisher ungelöst (**P-NP-Problem**). Vieles spricht für $\mathbf{P} \neq \mathbf{NP}$.

Am Anfang des Kapitels wird die Klasse $\mathbf{PSPACE} = \bigcup_{k=0}^{\infty} SPACE\left(n^k\right)$ eingeführt, die Klasse der Entscheidungsprobleme, die deterministisch mit einem Speicherplatzbedarf gelöst werden können, der proportional zu einem Polynom in der Größe der Eingabe ist. Entsprechend kann man die Klasse der Entscheidungsprobleme, die deterministisch mit einem Speicherplatzbedarf gelöst werden können, der proportional zu einem Polynom in der Größe der Eingabe ist, und **NPSPACE** bezeichnen. Aus Satz 2.5-3 folgt direkt, dass **PSPACE** = **NPSPACE** gilt.

5.3 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 $size(x) = n$

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 $size(x) = n$

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.

Der deterministische Verifizierer $\mathbf{V}_{L_\Pi}$ für eine Menge $L_\Pi \subseteq \Sigma_\Pi^*$ in **NP** kann so entworfen werden, dass er jede Eingabe der Form $(x, B) \in \Sigma^* \times \Sigma_0^*$ ablehnt, für die $|B| > C \cdot p(|x|)$ gilt. Da

Polynome zeitkonstruierbar sind, kann diese Überprüfung in polynomieller Zeit (in Abhängigkeit von $|x|$) erfolgen. Offensichtlich ist daher $L(\mathbf{V}_{L_\Pi}) \in \mathbf{P}$.

Gehört umgekehrt eine Menge $L' \subseteq \Sigma^* \times \Sigma_0^*$ zu **P** und ist *p* ein Polynom, dann ist die Menge $L = \{ x \mid x \in \Sigma^* \text{ und es gibt } B \in \Sigma_0^* \text{ mit } |B| \le p(|x|) \text{ und } (x, B) \in L' \}$ in **NP**. Zum Nachweis muss man eine polynomiell zeitbeschränkte nichtdeterministische Turingmaschine *TM* angeben, die *L* akzeptiert. Diese Turingmaschine wird wieder informell über ihre Arbeitsweise bei Eingabe eines Worts $x \in \Sigma^*$ beschrieben: *TM* erzeugt zunächst nichtdeterministisch ein Wort $B \in \Sigma_0^*$ mit $|B| \le p(|x|)$ und akzeptiert *x* genau dann, wenn $(x, B) \in L'$ gilt. Ist p_1 das Polynom, das in der Definition von $L' \in \mathbf{P}$ verwendet wird, so ist die (nichtdeterministische) Laufzeit von *TM* beschränkt durch $c \cdot p(|x|) + p_1(|x| + p(|x|) + 3)$ mit einer Konstanten $c > 0$, also polynomiell in $|x|$. Insgesamt ergibt sich mit diesen Überlegungen:

Satz 5.3-1:

Eine Sprache $L \subseteq \Sigma^*$ ist genau dann in **NP**, wenn es eine Sprache $L' \in \mathbf{P}$ mit $L' \subseteq \Sigma^* \times \Sigma_0^*$ und ein Polynom *p* gibt, so dass

$L = \{ x \mid x \in \Sigma^* \text{ und es gibt } B \in \Sigma_0^* \text{ mit } |B| \le p(|x|) \text{ und } (x, B) \in L' \}$ ist.

Auf der Grundlage von Satz 5.3-1 werden häufig Beweise dafür geführt, dass eine Sprache $L \subseteq \Sigma^*$ in **NP** liegt. Dazu wird zu $x \in \Sigma^*$ konkret die syntaktische Form eines Beweises $B \in \Sigma_0^*$ beschrieben und für das Paar (x, B) in polynomieller Zeit deterministisch eine definierte Eigenschaft nachgewiesen.

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 kann**.

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 inzwischen gut etablierte Theorie, die **Theorie der NP-Vollständigkeit**, Ursachen dieses Phänomens. Eine Einführung in diese Theorie gibt Kapitel 5.4.

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 $V = \{x_1, ..., x_n\}$ und m Junktoren, $f : V \to \{\mathsf{TRUE}, \mathsf{FALSCH}\}$ 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.

Das Entscheidungsproblem erfSAT sucht also nach einem Entscheidungsalgorithmus für die Menge

$$L_{\text{erfSAT}} = \left\{ [F, f] \;\middle|\; \begin{array}{l} F \text{ ist ein Boolescher Ausdruck in konjunktiver Normalform, und} \\ f \text{ ist eine Belegung der Variablen mit Wahrheitswerten, so dass} \\ F \text{ bei Auswertung gemäß dieser Belegung den Wahrheitswert} \\ \mathsf{TRUE} \text{ erhält} \end{array} \right\},$$

der in polynomieller Zeit arbeitet. Ein möglicher Entscheidungsalgorithmus 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 $V = \{ x_1, ..., x_n \}$ und m Junktoren.

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.

Die zum Entscheidungsproblem CSAT zugehörige zu entscheidende Menge ist
$L_{\text{CSAT}} = \{ F \mid F \text{ ist ein Boolescher Ausdruck in konjunktiver Normalform, der erfüllbar ist} \}$.

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 $V = \{ x_1, ..., x_n \}$ und *m* Junktoren.

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**; die mittels eines polynomiell zeitsbeschränkten nichtdeterministischen Algorithmus zu entscheidende Menge ist
$L_{\text{SAT}} = \{ F \mid F \text{ ist ein Boolescher Ausdruck, der erfüllbar ist} \}$.

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_j 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 5.3-2:
Ist *F* eine Instanz für 2-CSAT mit *n* Variablen und *m* Klauseln, dann liefert das im folgenden 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_j genau zwei Literale enthält, d.h. jedes F_j hat die Form $F_j = (y_{j_1} \vee y_{j_2})$.

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 };
```

Eine Eingabeinstanz *F* hat die Form $(y_{l_1} \vee y_{l_2}) \wedge \ldots \wedge (y_{m_1} \vee y_m)$ mit Literalen y_{j_i} (unter Weglassen der äußeren Klammerung). Unter der Annahme (für die untere Grenze), dass jede Variable durch ein einziges Symbol dargestellt werden kann und keine Negationszeichen in *F* auftreten, und unter Berücksichtigung der Indizierung der Variablen (für die obere Grenze) ist $6 \cdot m - 1 \le |F| \le 8 \cdot m - 1 + 2 \cdot m \cdot (\lfloor \log_2(n) \rfloor + 1) = 2 \cdot m \cdot (\lfloor \log_2(n) \rfloor + 5) - 1$. Sind die Variablen in allen Klauseln von *F* paarweise verschieden, so ist $n = 2 \cdot m$; im allgemeinen ist $n \le 2 \cdot m$. Die Laufzeit des Verfahrens ist von der Ordnung $O(n \cdot m)$ bzw. von der Größe $C \cdot n \cdot m$ mit einer Konstanten $C > 0$. Nimmt man als Problemgröße die untere Grenze $|F| = d \cdot m$ mit einer Konstanten $d > 0$, so ist $C \cdot n \cdot m \le (2 \cdot C) / d^2 \cdot |F|^2$, d.h. die Laufzeit ist begrenzt durch einen Wert der Ordnung $O(|F|^2)$.

Nimmt man als Problemgröße die obere Grenze $|F| = d' \cdot m \cdot \log(n) = d'' \cdot m \cdot \log(m)$ mit Konstanten $d' > 0$ und $d'' > 0$, so gilt für fast alle *m* die Abschätzung

$C \cdot n \cdot m \le 2 \cdot C \cdot m^2 \le 2 \cdot C \cdot (m \cdot \log(m))^2 = (2 \cdot C) / d''^2 \cdot |F|^2$, d.h. die Laufzeit ist ebenfalls begrenzt durch einen Wert der Ordnung $O(|F|^2)$.

///

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 **Beispiele für Probleme** Π **aus NP** aufgeführt, von denen nicht bekannt ist, ob sie in **P** liegen. Für jedes Problem Π wird die zu entscheidende Menge L_Π angegeben. Für eine Instanz $x \in \Sigma_\Pi^*$ ist mit Hilfe eines polynomiell zeitbeschränkten nichtdeterministischen Algorithmus unter Einsatz eines Verifizierers die Entscheidung $x \in L_\Pi$ bzw. $x \notin L_\Pi$ zu treffen. Der bei Eingabe einer Instanz $x \in \Sigma_\Pi^*$ jeweils verwendete Beweis $B_x \in \Sigma_0^*$, den der Verifizierer zur `ja/nein`-Entscheidung heranzieht, wird ebenfalls angegeben.

Mit *size*(*x*) wird die **Größe einer Eingabeinstanz** bezeichnet. Hierbei wird nicht in allen Fällen die exakte Anzahl der Zeichen genommen, um eine Instanz *x* zu notieren, sondern häufig ein Wert $k \in O(|x|)$, der einfach zu bestimmen ist und die Abschätzung der Zeitkomplexität erleichtert.

- **Erfüllbarkeitsproblem für Boolesche Ausdrücke in konjunktiver Normalform (CSAT)** bzw. **Boolescher Ausdrücke in allgemeiner Form (SAT)**:

  Instanz: F

  F ist ein Boolescher Ausdruck in konjunktiver Normalform bzw. in allgemeiner Form mit der Variablenmenge $V = \{ x_1, ..., x_n \}$ und m Junktoren.

  Zu entscheidende Menge:

  $$L_{\text{CSAT}} = \left\{ F \;\middle|\; \begin{array}{l} F \text{ ist ein Boolescher Ausdruck in konjunktiver Normalform,} \\ \text{und } F \text{ ist erfüllbar} \end{array} \right\} \text{ bzw.}$$

  $$L_{\text{SAT}} = \{ F \mid F \text{ ist ein Boolescher Ausdruck, und } F \text{ ist erfüllbar} \}$$

  Beweis: B_F ist eine 0-1-Folge der Länge n

  Arbeitsweise des Verifizierers:

  Der i-te Wert in B_F (für $i = 1, ..., n$) 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`.

- **Erfüllbarkeitsproblem der Aussagenlogik mit Formeln in konjunktiver Normalform mit genau 3 Literalen pro Klausel (3-CSAT):**

  Instanz: F

  F ist ein Boolescher Ausdruck in konjunktiver Normalform mit höchstens 3 Literalen pro Klausel und der Variablenmenge $V = \{ x_1, ..., x_n \}$ und m Junktoren; F hat die Form $F = F_1 \wedge F_2 \wedge ... F_m$; hierbei hat jedes F_j die Form $F_j = (y_{j_1} \vee y_{j_2} \vee y_{j_3})$, und y_{i_k} steht für eine Boolesche Variable (d.h. $y_{i_k} = x_i$) oder für eine negierte Boolesche Variable (d.h. $y_{i_k} = \neg x_i$)

  Bemerkung: Häufig wird 3-CSAT so definiert, dass höchstens 3 Literale pro Klausel vorkommen; in diesem Fall kann man eine Klausel durch Wiederholung von Literalen, die in ihr vorkommen, auf genau drei Literale pro Klausel ergänzen.

  Zu entscheidende Menge:

  $$L_{\text{3-CSAT}} = \left\{ F \;\middle|\; \begin{array}{l} F \text{ ist ein Boolescher Ausdruck in konjunktiver Normalform mit} \\ \text{genau 3 Literalen pro Klausel, und } F \text{ ist erfüllbar} \end{array} \right\}$$

  Beweis: B_F ist eine 0-1-Folge der Länge n

  Arbeitsweise des Verifizierers:

  Der i-te Wert in B_F (für $i = 1, ..., n$) 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 ausge-

wertet. Falls sich bei dieser Auswertung von *F* der Wert `TRUE` ergibt, wird `ja` ausgegeben, ansonsten `nein`.

Bemerkung: Es gilt $L_{\text{3-CSAT}} \subseteq L_{\text{CSAT}} \subseteq L_{\text{SAT}}$.

- **Kliquenproblem (KLIQUE):**

Instanz: $[G,k]$

$G=(V,E)$ ist ein ungerichteter Graph mit der Knotenmenge $V=\{v_1,...,v_n\}$ und der Kantenmenge $E \subseteq V \times V$, und *k* ist eine natürliche Zahl mit $k \le n$.

Zu entscheidende Menge:

$L_{\text{KLIQUE}} = \{ [G,k] \mid G \text{ ist ein ungerichteter Graph und besitzt eine Klique der Größe } k \}$

Eine Klique der Größe *k* ist eine Teilmenge $V' \subseteq V$ der Knotenmenge mit $|V'|=k$, so dass für alle $u \in V'$ und alle $v \in V'$ mit $u \neq v$ gilt: $(u,v) \in E$.

Beweis: $B_{[G,k]}$ ist eine Folge $bin(i_1)\#...\#bin(i_k)$ von *k* natürlichen Zahlen in Binärdarstellung, die durch das Zeichen # getrennt sind, mit $i_j \le n$ für $j=1,...,k$;

$|B_{[G,k]}| \le k \cdot (\lfloor \log_2(n) \rfloor + 1) + k - 1$, d.h. $|B_{[G,k]}| \in O(n \cdot \log(n))$

Arbeitsweise des Verifizierers:

Jede Zahl i_j wird als mögliche Knotennummer interpretiert. Es wird geprüft, ob $\{v_{i_1},...,v_{i_k}\} \subseteq V$ gilt und ob $\{v_{i_1},...,v_{i_k}\}$ eine Klique in *G* darstellt, d.h. ob $(v_{i_s},v_{i_t}) \in E$ für alle $v_{i_s} \in \{v_{i_1},...,v_{i_k}\}$ und $v_{i_t} \in \{v_{i_1},...,v_{i_k}\}$ mit $i_s \neq i_t$ gilt. In diesem Fall wird `ja` ausgegeben, ansonsten `nein`.

- **0/1-Rucksack-Entscheidungsproblem (RUCKSACK):**

Instanz: $[a_1,...,a_n,b]$

$a_1,...,a_n$ und *b* sind natürliche Zahlen mit $a_i > 0$ für $i=1,...,n$ und $b>0$.

Zu entscheidende Menge:

$$L_{\text{RUCKSACK}} = \left\{ [a_1,...,a_n,b] \;\middle|\; \text{Es gibt eine Teilmenge } J \subseteq \{1,...,n\} \text{ mit } \sum_{j \in J} a_j = b \right\}$$

Beweis: $B_{[a_1,...,a_n,b]}$ ist eine 0-1-Folge $x_1,...,x_n$ der Länge *n*

Arbeitsweise des Verifizierers:

Es wird geprüft, ob $\sum_{i=1}^{n} x_i \cdot a_i = b$ gilt. In diesem Fall wird `ja` ausgegeben, ansonsten `nein`.

- **Partitionenproblem (PARTITION):**

Instanz: $[a_1, \ldots, a_n]$

$a_1, \ldots, a_n$ sind natürliche Zahlen mit $a_i > 0$ für $i = 1, \ldots, n$.

Zu entscheidende Menge:

$$L_{\text{PARTITION}} = \left\{ [a_1, \ldots, a_n] \,\middle|\, \text{Es gibt eine Teilmenge } J \subseteq \{1, \ldots, n\} \text{ mit } \sum_{j \in J} a_j = \sum_{j \notin J} a_j \right\}$$

Beweis: $B_{[a_1, \ldots, a_n]}$ ist eine 0-1-Folge $x_1, \ldots, x_n$ der Länge n

Arbeitsweise des Verifizierers:

Es wird $J = \{ j \mid x_j = 0 \}$ gesetzt und geprüft, ob $\sum_{j \in J} a_j = \sum_{j \notin J} a_j$ gilt. In diesem Fall wird `ja` ausgegeben, ansonsten `nein`.

- **Packungsproblem (BINPACKING):**

Instanz: $[a_1, \ldots, a_n, b, k]$

$a_1, \ldots, a_n$ („Objekte“) sind natürliche Zahlen mit $0 < a_i \leq b$ für $i = 1, \ldots, n$, $b \in \mathbf{N}$ („Behältergröße“), $k \in \mathbf{N}$ mit $k \leq n$;

Zu entscheidende Menge:

Die Aussage „Die Objekte können so auf k Behälter der Behältergröße b verteilt werden, dass kein Behälter überläuft“ drückt umgangssprachlich aus, dass es eine Abbildung $f : \{1, \ldots, n\} \to \{1, \ldots, k\}$ gibt, so dass für alle $j \in \{1, \ldots, k\}$ gilt: $\sum_{f(i)=j} a_i \leq b$ („diejenigen Objekte, die in den j-ten Behälter gelegt werden, überschreiten die Behältergröße nicht“, $f(i)$ ist die Nummer des Behälters, in den a_i gelegt wird)

$$L_{\text{BINPACKING}} = \left\{ [a_1, \ldots, a_n, b, k] \,\middle|\, \begin{array}{l} \text{Die Objekte } a_1, \ldots, a_n \text{ lassen sich so auf } k \text{ Behälter} \\ \text{der Behältergröße } b \text{ aufteilen, dass kein Behälter} \\ \text{überläuft} \end{array} \right\}$$

Beweis: $B_{[a_1, \ldots, a_n, b, k]}$ ist eine Folge $bin(i_1)\#\ldots\#bin(i_n)$ von n Zahlen in Binärdarstellung, die durch das Zeichen # getrennt sind, mit $1 \leq i_j \leq k$ für $j = 1, \ldots, n$;

$\left| B_{[a_1, \ldots, a_n, b, k]} \right| \leq n \cdot (\lfloor \log_2(k) \rfloor + 1) + n - 1$, d.h. $\left| B_{[a_1, \ldots, a_n, b, k]} \right| \in O(n \cdot \log(n))$

Arbeitsweise des Verifizierers:

Die Zahl i_j wird als „Nummer des Behälters“ interpretiert, in den das Objekt a_j gelegt wird; es wird geprüft, ob kein Behälter überläuft, d.h. ob gilt:

$\sum_{i_j=1} a_j \leq b$, ..., $\sum_{i_j=k} a_j \leq b$. In diesem Fall wird `ja` ausgegeben, ansonsten `nein`.

- **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 Knotenmenge $V=\{v_1,\ldots,v_n\}$ und Kantenmenge $E\subseteq V\times V$.

Zu entscheidende Menge:

$$L_{\text{HAMILTONKREIS}}=\left\{ G \,\middle|\, \begin{array}{l} G \text{ ist ein gerichteter bzw. ungerichteter Graph, und} \\ G \text{ besitzt einen Hamiltonkreis} \end{array} \right\}$$

Ein Hamiltonkreis ist eine Anordnung $(v_{\pi(1)},v_{\pi(2)},\ldots,v_{\pi(n)})$ der Knoten mittels einer Permutation π der Knotenindizes, so dass für $i=1,\ldots,n-1$ gilt: $(v_{\pi(i)},v_{\pi(i+1)})\in E$ und $(v_{\pi(n)},v_{\pi(1)})\in E$.

Beweis: B_G ist eine Folge $bin(i_1)\#\ldots\#bin(i_n)$ von n Zahlen in Binärdarstellung, die durch das Zeichen # getrennt sind, mit $1\le i_j\le n$ für $j=1,\ldots,n$;

$|B_G|\le n\cdot(\lfloor\log_2(n)\rfloor+1)+n-1$, d.h. $|B_G|\in O(n\cdot\log(n))$

Arbeitsweise des Verifizierers:

Es wird geprüft, ob $i_1,\ldots,i_n$ eine Permutation der Knotenindizes ist, d.h. ob alle Werte paarweise verschieden sind und die Menge $\{1,\ldots,n\}$ darstellen. Anschließend wird geprügt, ob die Bedingungen $(v_{i_t},v_{i_{t+1}})\in E$ für $t=1,\ldots,n-1$ und $(v_{i_n},v_{i_1})\in E$ gelten. In diesem Fall wird `ja` ausgegeben, ansonsten `nein`.

- **Problem des Handlungsreisenden (HANDLUNGSREISENDER):**

Instanz: $[G,k]$

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

Zu entscheidende Menge:

$$L_{\text{HANDLUNGSREISENDER}}=\left\{ [G,k] \,\middle|\, \begin{array}{l} G \text{ ist ein gewichteter gerichteter bzw. ungerichteter Graph,} \\ \text{und } G \text{ besitzt ein Tour mit minimalen Kosten} \le k \end{array} \right\}$$

Eine Tour durch G ist eine geschlossene Kantenfolge

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

in der jeder Knoten des Graphen (als Anfangsknoten einer Kante) genau einmal vorkommt; die Kosten einer Tour $\langle(v_{i_1},v_{i_2}),(v_{i_2},v_{i_3}),\ldots,(v_{i_{n-1}},v_{i_n}),(v_{i_n},v_{i_1})\rangle$ ist die Summe der Gewichte der Kanten auf der Tour, d.h. der Ausdruck

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

Beweis: $B_{[G,k]}$ ist eine Folge $bin(i_1)\#...\#bin(i_n)$ von n Zahlen in Binärdarstellung, die durch das Zeichen # getrennt sind, mit $1 \le i_j \le n$ für $j = 1,...,n$;

$\left|B_{[G,k]}\right| \le n \cdot \left(\lfloor \log_2(n) \rfloor + 1\right) + n - 1$, d.h. $\left|B_{[G,k]}\right| \in O(n \cdot \log(n))$

Arbeitsweise des Verifizierers:

Es wird geprüft, ob $B_{[G,k]}$ eine Tour durch G beschreibt, und es werden die Kosten K dieser Tour ermittelt. Falls $K \le k$ gilt, wird `ja` ausgegeben, ansonsten `nein`.

- **Problem der Färbbarkeit der Knoten in einem ungerichteten Graphen (FÄRBBARKEIT):**

Instanz: $[G,k]$

$G = (V,E)$ ist ein ungerichteter Graph mit Knotenmenge $V = \{v_1,...,v_n\}$ und Kantenmenge $E \subseteq V \times V$; $k \in \mathbf{N}$ mit $1 \le k \le n$

Zu entscheidende Menge:

Die Knoten des Graphen können so mit k Farben gefärbt werden, dass jeweils zwei durch eine Kante verbundene Knoten unterschiedliche Farben erhalten.

$$L_{\text{FÄRBBARKEIT}} = \left\{ [G,k] \,\middle|\, \begin{array}{l} G \text{ ist ein ungerichteter Graph, der eine zulässige} \\ \text{Knotenfärbung mit } k \text{ Farben besitzt} \end{array} \right\}$$

Eine zulässige Knotenfärbung mit k Farben ist eine Abbildung $f : V \to \{1,...,k\}$ mit $f(v) \ne f(v')$ für jede Kante $(v,v') \in E$.

Beweis: $B_{[G,k]}$ ist eine Folge $bin(i_1)\#...\#bin(i_n)$ von n Zahlen in Binärdarstellung, die durch das Zeichen # getrennt sind, mit $1 \le i_j \le k$ für $j = 1,...,n$;

$\left|B_{[G,k]}\right| \le n \cdot \left(\lfloor \log_2(k) \rfloor + 1\right) + n - 1$, d.h. $\left|B_{[G,k]}\right| \in O(n \cdot \log(n))$

Arbeitsweise des Verifizierers:

Es wird geprüft, ob durch $f(v_j) = bin(i_j)$ eine zulässige Kontenfärbung gegeben ist, d.h. ob $f(v_j) \ne f(v_l)$ für alle Kanten $(v_j, v_l) \in E$ gilt. In diesem Fall wird `ja` ausgegeben, ansonsten `nein`.

- **Problem der $\{0,1\}$-Linearen Programmierung ($\{0,1\}$-LP)**

  Instanz: $[A,\vec{b},\vec{z}]$

  $A \in \mathbf{Z}^{m \cdot n}$ ist eine ganzzahlige Matrix mit *m* Zeilen und *n* Spalten, $\vec{b} \in \mathbf{Z}^m$ ist ein ganzzahliger Vektor, $\vec{z}$ ist ein Spaltenvektor von *n* Variablen, die nur die Werte 0 oder 1 annehmen können.

  Zu entscheidende Menge:

  $$L_{\{0,1\}\text{-LP}} = \left\{ [A,\vec{b},\vec{z}] \,\middle|\, \begin{array}{l} \text{es gibt eine Zuweisung der Werte 0 und 1 an die Variablen in } \vec{z}, \\ \text{so dass das lineare Ungleichungssystem } A \cdot \vec{z} \le / \ge \vec{b} \text{ erfüllt ist} \end{array} \right\}$$

  Beweis: $B_{[A,\vec{b},\vec{z}]}$ ist eine 0-1-Folge der Länge *n*

  Arbeitsweise des Verifizierers:

  Es wird geprüft, ob die durch $B_{[A,\vec{b},\vec{z}]}$ definierte Zuweisung von Werten an die Variablen in $\vec{z}$ die Bedingung $A \cdot \vec{z} \le / \ge \vec{b}$ erfüllt. In diesem Fall wird `ja` ausgegeben, ansonsten `nein`.

- **Problem des längsten Wegs in einem gewichteten Graphen (LÄNGSTER WEG):**

  Instanz: $[G, v_i, v_j, k]$

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

  Zu entscheidende Menge:

  $$L_{\text{LÄNGSTER WEG}} = \left\{ [G, v_i, v_j, k] \,\middle|\, \begin{array}{l} G \text{ ist ein gewichteter gerichteter bzw. ungerichteter Graph,} \\ v_i \text{ und } v_j \text{ sind zwei Knoten aus } G, \\ \text{und } G \text{ besitzt einen einfachen Weg von } v_i \text{ nach } v_j \text{ mit Gewicht} \ge k \end{array} \right\}$$

  Ein einfacher Weg ist ein Pfad ohne Knotenwiederholungen

  Beweis: $B_{[G,v_i,v_j,k]}$ ist eine Folge $bin(i_1)\#...\#bin(i_t)$ von $t \le n$ Zahlen in Binärdarstellung, die durch das Zeichen # getrennt sind, mit $1 \le i_j \le n$ für $t = 1, ..., n$;

  $\left|B_{[G,v_i,v_j,k]}\right| \le n \cdot (\lfloor \log_2(n) \rfloor + 1) + n - 1$, d.h. $\left|B_{[G,v_i,v_j,k]}\right| \in O(n \cdot \log(n))$

  Arbeitsweise des Verifizierers:

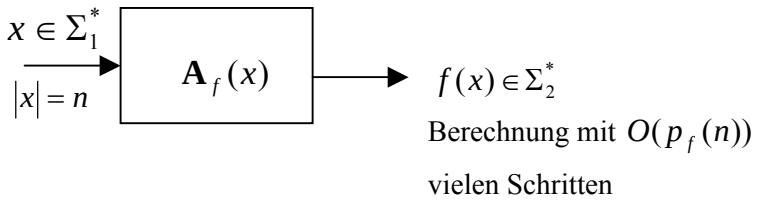
  Es wird überprüft, ob $\{v_{i_1}, v_{i_2}, ..., v_{i_t}\} \subseteq V$ gilt und ob die Folge $v_{i_1}, v_{i_2}, ..., v_{i_k}$ einen einfachen Weg (d.h. ohne Knotenwiederholungen) von v_i nach v_j mit Gewicht $\ge k$ beschreibt. In diesem Fall wird `ja` ausgegeben, ansonsten `nein`.

Man kennt heute mehrere tausend Probleme aus **NP**. In der angegebenen Literatur werden geordnet nach Anwendungsgebieten umfangreiche Listen von **NP**-Problemen aufgeführt.

5.4 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 $size(x) = |x| = n$ 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 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:
$x \in L_1 \Leftrightarrow f(x) \in L_2$.

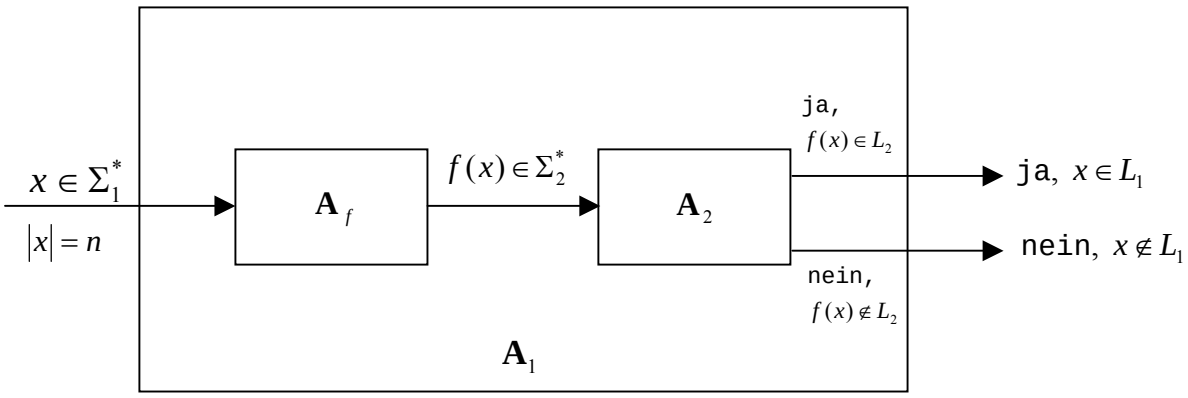
Diese Eigenschaft kann auch so formuliert werden:
$x \in L_1 \Rightarrow f(x) \in L_2$ und $x \notin L_1 \Rightarrow f(x) \notin L_2$.

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:

$\{0,1\}$**-LP:**

Instanz: $[A, \vec{b}, \vec{z}]$

$A \in \mathbf{Z}^{m \cdot n}$ 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 $\leq_m^p$ $\{0,1\}$-LP.

Zum Nachweis ist eine in polynomieller Zeit berechnenbare Funktion *f* anzugeben, die jedem Booleschen Ausdruck *F* in konjunktiver Normalform eine Instanz von $\{0,1\}$-LP 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 Unglei-

chung $t_{i_1} + ... + t_{i_k} \geq 1$ 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 $f(F) = \left[A, \vec{b}, \vec{z}\right]$ 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 $f(F) = \left[A, \vec{b}, \vec{z}\right]$ 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*.

Satz 5.4-1:

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:

(i) $L \leq_m^p L$, d.h. die Relation $\leq_m^p$ ist reflexiv.

(ii) 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.

(iii) Gilt $L_1 \leq_m^p L_2$ und $L_2 \in \mathbf{P}$, dann ist $L_1 \in \mathbf{P}$.

(iv) Gilt $L_1 \leq_m^p L_2$ und $L_2 \in \mathbf{NP}$, dann ist $L_1 \in \mathbf{NP}$.

Beweis:

Zu (i): Die Funktion $f : \begin{cases} \Sigma^* & \rightarrow & \Sigma^* \\ x & \rightarrow & x \end{cases}$ liefert die Reduktion $L \leq_m^p L$.

Zu (ii): Die Reduktion $L_1 \leq_m^p L_2$ erfolge mit der Funktion $f_1 : \Sigma_1^* \rightarrow \Sigma_2^*$, die Reduktion $L_2 \leq_m^p L_3$ mit der Funktion $f_2 : \Sigma_2^* \rightarrow \Sigma_3^*$. Dann gilt $L_1 \leq_m^p L_3$ mit der Reduktion $f = f_2 \circ f_1$: Da sich $f_1(x)$ in polynomieller Zeit deterministisch (in der Länge von *x*) berechnen lässt, hat $f_1(x)$ polynomielle Länge (in der Länge von *x*). Der Wert $f(x) = f_2\left(f_1(x)\right)$ lässt sich in polynomieller Zeit deterministisch (in der Länge von $f_1(x)$) berechnen, so dass die gesamte Berechnung deterministisch in polynomieller Zeit (in der Länge von *x*) erfolgen kann.

Zu (iii): Die obigen Überlegungen zur Bedeutung der Reduzierbarkeit zeigen, dass der Entscheidungsalgorithmus $\mathbf{A}_1$ für L_1 ein deterministischer polynomiell zeitbeschränkter Algorithmus ist, wenn dieses für $\mathbf{A}_2$ ein deterministischer polynomiell zeitbeschränkter Entscheidungsalgorithmus für L_2 ist.

Zu (iv): Die entsprechenden Aussagen wie in (iii) kann man treffen, wenn $\mathbf{A}_2$ ein nichtdeterministischer polynomiell zeitbeschränkter Entscheidungsalgorithmus für L_2 ist.

///

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 5.4-2:
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 $\{F \mid F$ 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 RV-Modell 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
$zust_{t,q}$	$t = 0, ..., p_{L_\Pi}(n)$ $q \in Q$	$zust_{t,q} = \text{TRUE}$ genau dann, wenn sich TM_{L_Π} im Schritt *t* im Zustand *q* befindet Anzahl an Variablen $zust_{t,q}$: $(k+1)\cdot(p_{L_\Pi}(n)+1)$
$pos_{t,i}$	$t = 0, ..., p_{L_\Pi}(n)$ $i = -p_{L_\Pi}(n), ..., p_{L_\Pi}(n)+1$	$pos_{t,i} = \text{TRUE}$ genau dann, wenn sich der Schreib/Lesekopf von TM_{L_Π} im Schritt *t* über der Zelle mit Nummer *i* befindet Anzahl an Variablen $pos_{t,i}$: $2\cdot(p_{L_\Pi}(n)+1)^2$
$band_{t,i,a}$	$t = 0, ..., p_{L_\Pi}(n)$	$band_{t,i,a} = \text{TRUE}$ genau dann, wenn sich im

	$i = -p_{L_\Pi}(n), ..., p_{L_\Pi}(n)+1,$ $a \in \Sigma_\Pi$	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:

$$R = \bigwedge_{t=0}^{p_{L_\Pi}(n)} \left[G(zust_{t,q_0}, ..., zust_{t,q_k}) \wedge G(pos_{t,-p_{L_\Pi}(n)}, ..., pos_{t,p_{L_\Pi}(n)+1}) \wedge \left(\bigwedge_{i=-p_{L_\Pi}(n)}^{p_{L_\Pi}(n)+1} G(band_{t,i,a_0}, ..., band_{t,i,a_l}) \right) \right]$$

Die Anzahl an Literalen in *R* beträgt

$$(p_{L_\Pi}(n)+1) \cdot \left((k+1)^2 + 4(p_{L_\Pi}(n)+1)^2 + 2(p_{L_\Pi}(n)+1)(l+1)^2\right) \in O\left((p_{L_\Pi}(n))^3\right).$$

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 $2(p_{L_\Pi}(n)+2) \in O(p_{L_\Pi}(n))$ 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:

$$\ddot{U}_1 = \bigwedge_{t,q,i,a} \left[zust_{t,q} \wedge pos_{t,i} \wedge band_{t,i,a} \Rightarrow \vee \left\{ zust_{t+1,q'} \wedge pos_{t+1,i+y} \wedge band_{t+1,i,a'} \mid (q',a',y) \in \delta(q,a) \right\} \right].$$

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

$2\left(p_{L_\Pi}(n)+1\right)^2 \cdot (k+1) \cdot (l+1)(3(k+1)(l+1)+3) \in O\left(\left(p_{L_\Pi}(n)\right)^2\right)$ viele Literale.

Um zu zeigen, wie sich auch $\ddot{U}_1$ in eine äquivalente Formel in konjunktiver Normalform umformen lässt, soll exemplarisch ein Ausdruck

$zust_{t,q} \wedge pos_{t,i} \wedge band_{t,i,a} \Rightarrow \vee \left\{ zust_{t+1,q'} \wedge pos_{t+1,i+y} \wedge band_{t+1,i,a'} \mid (q',a',y) \in \delta(q,a) \right\}$ 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:

$$\ddot{U}_2 = \bigwedge_{t,i,a} \left[(\neg pos_{t,i} \wedge band_{t,i,a}) \Rightarrow band_{t+1,i,a} \right].$$

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:

$$\ddot{U}_2 = \bigwedge_{t,i,a}\left[\left(\neg pos_{t,i} \wedge band_{t,i,a}\right) \Rightarrow band_{t+1,i,a}\right]$$
$$= \bigwedge_{t,i,a}\left[\left(pos_{t,i} \vee \neg band_{t,i,a} \vee band_{t+1,i,a}\right)\right].$$

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 5.4-3:
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 5.4-4:
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 5.4-1 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 5.4-1) weitere **NP**-vollständige Probleme ableiten:

Satz 5.4-5:

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.

Der folgende Satz zeigt, dass es „relativ unwahrscheinlich" ist, dass **P** = **NP** gilt, da man bisher nur **NP**-vollständige Sprachen kennt, die einen Entscheidungsalgorithmus mit exponentieller Komplexität besitzen.

Satz 5.4-6:

P = **NP** impliziert, dass jede endliche nichtleere Menge **NP**-vollständig ist.

Beweis:

Es sei *M* eine endliche nichtleere Menge, die als eine Menge von Wörtern über einem Alphabet Σ aufgefasst werden kann, d.h. $M \subseteq \Sigma^*$, und es gibt $w_0 \in \Sigma^*$ mit $w_0 \in M$ und es gibt $w_1 \in \Sigma^*$ mit $w_1 \notin M$. Trivialerweise liegt *M* in **P** und damit in **NP**. Zu zeigen bleibt, dass für jede Sprache L' in **NP** eine Reduktion $L' \leq_m^p M$ möglich ist.

Es sei $L' \subseteq \Sigma'^*$ eine Sprache in **NP**. Gilt **P** = **NP**, so gehört L' zu **P**. Das bedeutet, dass es einen polynomiell zeitbeschränkten deterministischen Algorithmus $\mathbf{A}_{L'}$ gibt mit $\mathbf{A}_{L'}(w) = \texttt{ja}$ genau dann, wenn $w \in L'$ ist. Die folgende Abbildung *f* ist deterministisch polynomiell zeitbeschränkt berechenbar:

$$f: \begin{cases} \Sigma'^* & \rightarrow & \Sigma^* \\ w & \rightarrow & \begin{cases} w_0 & \text{für } w \in L' \\ w_1 & \text{für } w \notin L', \end{cases} \end{cases}$$

da man mit Hilfe von $\mathbf{A}_{L'}$ in polynomieller Zeit für ein Wort $w \in \Sigma'^*$ zwischen den beiden Alternativen unterscheiden kann und dann ein konstanter Wert, nämlich w_0 oder w_1, als Funktionswert von *f* gesetzt wird. Außerdem gilt $w \in L'$ genau dann, wenn $\mathbf{A}_{L'}(w) = \texttt{ja}$, d.h. wenn $f(w) = w_0$ bzw. $f(w) \in M$ ist.

///

Ergänzt man den Input für die universelle Turingmaschine (Kapitel 2.4) um die Angabe der Anzahl auszuführender Schritte, so erhält man eine **NP**-vollständige Menge:

Satz 5.4-7:
Die Menge

$$L = \left\{ z \;\middle|\; \begin{array}{l} z = u\#w\#0^t \text{ mit } u \in \{0,1\}^*, w \in \{0,1\}^*, \text{ und} \\ \text{die nichtdeterministische Turingmaschine } K_w \text{ akzeptiert } u \text{ in höchstens } t \text{ Schritten} \end{array} \right\}$$

ist **NP**-vollständig.

Beweis:

Zu zeigen ist

1. $L \in \mathbf{NP}$
2. für jede Sprache $L_1 \in \mathbf{NP}$ gilt $L_1 \leq_m^p L$.

Zu 1.: Die universelle Turingmaschine *UTM* (siehe Kapitel 2.4) mit $L(UTM) = \left\{ u\#w \;\middle|\; u \in \{0,1\}^*, w \in \{0,1\}^* \text{ und } u \in L(K_w) \right\}$ wird um ein Band zur Turingmaschine *TM* erweitert. Bei Eingabe von $z = u\#w\#0^t$ mit $u \in \{0,1\}^*, w \in \{0,1\}^*$ (die syntaktisch korrekte Form kann hier vorausgesetzt werden) schreibt *TM* die Zeichen # und 0 auf das zusätzliche Band und kopiert die Teilzeichenkette nach dem zweiten Zeichen #, d.h. die Teilzeichenkette 0^t, auf das zusätzliche Band. Das Zeichen # dient zur Erkennung des linken Bandendes. Nun wird mit Hilfe von *UTM* das Verhalten von K_w bei Eingabe von *u* simuliert. Bei jedem simulierten Schritt wird ein Zeichen 0 auf dem zusätzlichen Band gelöscht und der Kopf nach links gerückt. Die Simulation stoppt und akzeptiert die Eingabe *z*, wenn entweder $u \in L(K_w)$ ist und noch mindestens ein Zeichen 0 auf dem zusätzlichen Band steht. Oder die Simulation stoppt und akzeptiert die Eingabe *z* nicht, wenn kein Zeichen 0 mehr auf dem

zusätzlichen Band steht, d.h. der Kopf dieses Bandes über dem Zeichen # angekommen ist.

Offensichtlich wird eine Eingabe $z = u\#w\#0^t$ mit $u \in \{0,1\}^*$, $w \in \{0,1\}^*$ in einer Anzahl von Schritten akzeptiert oder verworfen, die von der Ordnung $O(t)$, d.h. von der Ordnung $O(|z|)$ ist.

Zu 2.: Es sei $L_1 \subseteq \Sigma_1^*$ eine Sprache in **NP**. Zur Vereinfachung der Darstellung sei $\Sigma_1^* = \{0,1\}^*$. Es gibt eine polynomiell zeitbeschränkte nichtdeterministische Turingmaschine TM_1 mit $L_1 = L(TM_1)$. Das hierbei beteiligte Polynom sei p_1. Für ein Wort $u \in \{0,1\}^*$ wird in $p_1(|u|)$ viele Schritten von TM_1 entschieden, ob $u \in L_1$ gilt oder nicht. Die Kodierung von TM_1 sei $w_1 \in \{0,1\}^*$, d.h. $L_1 = L(TM_1) = L(K_{w_1})$. Da Polynome zeitkonstruierbar sind, ist die folgende Abbildung *f* deterministisch in polynomieller Zeit berechenbar:

$$f : \begin{cases} \{0,1\}^* & \to & \{0,1,\#\}^* \\ u & \to & u\#w_1\#0^{p_1(|u|)} \end{cases}.$$

Es gilt $u \in L_1$ genau dann, wenn $u \in L(K_{w_1})$ ist (diese Entscheidung erfordert nicht mehr als $p_1(|u|)$ viele Schritte), wenn also für $f(u) = u\#w_1\#0^{p_1(|u|)}$ die Beziehung $f(u) \in L$ ist.

///

Satz 5.4-8:

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). Alle in Kapitel 5.3 aufgeführten Beispiele von Entscheidungsproblemen aus **NP** sind **NP**-vollständig.

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. Exemplarisch werden im folgenden Kapitel die Beweise für

SAT $\leq_m^p$ 3-CSAT $\leq_m^p$ RUCKSACK $\leq_m^p$ PARTITION $\leq_m^p$ BINPACKING,

3-CSAT $\leq_m^p$ FÄRBBARKEIT,

3-CSAT $\leq_m^p$ KLIQUE und

3-CSAT $\leq_m^p$ GERICHTETER HAMILTONKREIS

$\leq_m^p$ UNGERICHTETER HAMILTONKREIS $\leq_m^p$ HANDUNGSREISENDER.

angegeben.

5.5 Beispiele für den Nachweis der NP-Vollständigkeit

In Kapitel 5.3 werden u.a. die folgenden Probleme definiert und ihre Zugehörigkeit zur Klasse **NP** skizziert:

SAT, 3-CSAT, RUCKSACK, PARTITION, BINPACKING, FÄRBBARKEIT, KLIQUE, GERICHTETER HAMILTONKREIS, UNGERICHTETER HAMILTONKREIS und HANDUNGSREISENDER.

In Satz 5.4-2 wird die **NP**-Vollständigkeit von SAT gezeigt, und aus Satz 5.4-5 folgt dann die **NP**-Vollständigkeit der übrigen Probleme, wenn der Nachweis der polynomiellen Reduzierbarkeit zwischen den Problemen erbracht ist. Dieses soll für die angegebenen Probleme im vorliegenden Kapitel geschehen. Dabei wurden Probleme aus verschiedenen Anwendungsgebieten mit unterschiedlichen zugrundeliegenden Alphabeten ausgewählt.

Satz 5.5-1:

SAT $\leq_m^p$ 3-CSAT

Mit $L_{\text{SAT}} = \{ F \mid F \text{ ist ein Boolescher Ausdruck und } F \text{ ist erfüllbar} \}$ und

$$L_{\text{3-CSAT}} = \left\{ F \;\middle|\; \begin{array}{l} F \text{ ist ein Boolescher Ausdruck in konjunktiver Normalform,} \\ \text{jede Klausel enthält genau 3 Literale,} \\ \text{und } F \text{ ist erfüllbar} \end{array} \right\}$$

steht die Aussage des Satzes für $L_{\text{SAT}} \leq_m^p L_{\text{3-CSAT}}$.

Beweis:

Zu zeigen ist: Es gibt eine Funktion *f*, die jedem (syntaktisch korrekten) Booleschen Ausdruck *F* einen Booleschen Ausdruck $f(F)$ in konjunktiver Normalform zuordnet, dessen Klauseln genau drei Literale enthalten, so dass *F* genau dann erfüllbar ist, wenn $f(F)$ erfüllbar ist. Diese Zuordnung kann mit einem deterministischen polynomiell zeitbeschränkten Algorithmus (gemessen in $size(F)$) erfolgen.

Ein Boolescher Ausdruck *F* enthält Variablen, Klammern und die Junktoren $\neg$, $\wedge$ und $\vee$. Zunächst wird *F* durch einen logisch äquivalenten F_1 ersetzt, indem alle Negationszeichen unmittelbar vor die Variablen gebracht werden. Dabei werden die logischen Äquivalenzen
$\neg(x \wedge y) \Leftrightarrow (\neg x \vee \neg y)$, $\neg(x \vee y) \Leftrightarrow (\neg x \wedge \neg y)$ und $\neg(\neg x) \Leftrightarrow x$
angewandt; hierbei stehen *x* und *y* für Variablen oder Teilformeln.

Im Ausdruck F_1 wird jedem Auftreten eines Junktors $\circ \in \{\wedge, \vee\}$ eine neue Variable *v* zugeordnet. Anschließend wird jedes Literal und jede Teilformel durch einen „Repräsentanten" dargestellt und eine neue Menge *M* von logischen Ausdrücken gebildet:

Der Repräsentant eines Literals ist das Literal selbst. Enthält F_1 die Teilformel $(F' \circ F'')$ mit $\circ \in \{\wedge, \vee\}$, wobei F' bzw. F'' durch v' bzw. v'' repräsentiert werden und dem Junktor $\circ$ die Variable *v* zugeordnet wurde, so wird *v* der Repräsentant von $(F' \circ F'')$, und es wird der Ausdruck $(v \Leftrightarrow (v' \circ v''))$ in *M* aufgenommen. Man kann außerdem annehmen, dass F_1 die Form $(F_1' \circ F_1'')$ besitzt; dem Junktor $\circ$ sei hierbei die Variable v_0 zugeordnet worden. Dann wird zusätzlich ein Ausdruck $(v_0 \vee v_0 \vee v_0)$ in *M* aufgenommen (v_0 ist der Repräsentant des gesamten Ausdrucks F_1).

Alle Formeln in *M* werden mit $\wedge$ verknüpft; der so entstandene Ausdruck sei F_2.

In F_2 wird jede Teilformel der Form
$(v \Leftrightarrow (v' \wedge v''))$ ersetzt durch $(\neg v \vee v' \vee v') \wedge (\neg v \vee v'' \vee v'') \wedge (v \vee \neg v' \vee \neg v'')$
und jede Teilformel der Form
$(v \Leftrightarrow (v' \vee v''))$ ersetzt durch $(v \vee v \vee \neg v') \wedge (\neg v \vee v' \vee v'') \wedge (v \vee v \vee \neg v'')$.

Schließlich werden noch doppelte Negationszeichen vor Variablen entfernt. Der so aus *F* entstandene Boolesche Ausdruck sei $f(F)$ und ist in konjunktiver Normalform, wobei jede Klausel genau drei Literale enthält.

Jeder Schritt in der Konstruktion von $f(F)$ aus *F* erfolgt deterministisch. Es lässt sich leicht einsehen, dass die Anzahl der Zeichen in $f(F)$ polynomiell mit der Anzahl der Zeichen in *F* verknüpft ist, so dass die Konstruktion in polynomieller Zeitbegrenzung erfolgt.

Es sei eine erfüllende Belegung der Variablen in *F* gegeben. Dann ist mit dieser Belegung der aus *F* entstandene Ausdruck F_1 ebenfalls erfüllt, da F_1 nur durch logisch äquivalente Umformungen aus *F* hervorgeht. Der Repräsentant *v* einer Teilformel $(F' \circ F'')$ von F_1 erhält den Wahrheitswert, der sich ergibt, wenn man das Ergebnis der Operation $(v' \circ v'')$ auswertet, wobei v' bzw. v'' die Repräsentanten von F' bzw. F'' sind. Hierbei werden zuerst die Ergebnisse der Verknüpfung der Literale ermittelt und dann den übrigen Repräsentanten Wahrheitswerte zugeordnet. Insbesondere erhält die oben beschriebene Variable v_0 den Wahrheitswert TRUE, so dass auch $f(F)$ erfüllt wird.

Ist umgekehrt $f(F)$ erfüllbar, so hat der Repräsentant v_0 des gesamten Ausdrucks F_1 den Wahrheitswert TRUE, da $f(F)$ die Klausel $(v_0 \vee v_0 \vee v_0)$ enthält, d.h. F_1 ist erfüllbar und damit *F*.

///

Satz 5.5-2:

3-CSAT $\leq_m^p$ RUCKSACK

Mit $L_{\text{3-CSAT}} = \left\{ F \;\middle|\; \begin{array}{l} F \text{ ist ein Boolescher Ausdruck in konjunktiver Normalform,} \\ \text{jede Klausel enthält genau 3 Literale,} \\ \text{und } F \text{ ist erfüllbar} \end{array} \right\}$ und

$L_{\text{RUCKSACK}} = \left\{ [a_1, \ldots, a_k, b] \;\middle|\; \text{Es gibt eine Teilmenge } J \subseteq \{1, \ldots, k\} \text{ mit } \sum_{j \in J} a_j = b \right\}$

steht die Aussage des Satzes für $L_{\text{3-CSAT}} \leq_m^p L_{\text{RUCKSACK}}$.

Beweis:

Es wird eine Funktion *f* beschrieben, die jedem (syntaktisch korrekten) Booleschen Ausdruck *F* in konjunktiver Normalform, dessen Klauseln genau drei Literale enthalten, eine Eingabeinstanz $f(F) = [a_1, \ldots, a_k, b]$ für das 0/1-Rucksack-Entscheidungsproblem zuordnet, so dass *F* genau dann erfüllbar ist ($F \in L_{\text{CSAT}}$), wenn es eine Teilmenge $J \subseteq \{1, \ldots, k\}$ mit $\sum_{j \in J} a_j = b$ gibt ($f(F) \in L_{\text{RUCKSACK}}$). Hierbei sind $a_1, \ldots, a_k$ und b natürliche Zahlen. Diese Zuordnung

kann mit einem polynomiell (in $size(F)$) zeitbeschränkten deterministischen Algorithmus berechnet werden.

Es sei F ein Booleschen Ausdruck in konjunktiver Normalform mit der Variablenmenge $\{x_1, ..., x_n\}$ und m Klauseln, die jeweils genau drei Literale enthalten, d.h.

$F = F_1 \wedge ... \wedge F_m$,

und jedes F_j hat die Form $F_j = (y_{j_1} \vee y_{j_2} \vee y_{j_3})$, wobei y_{j_k} ein Literal ist, d.h. für eine Variable (d.h. $y_{j_k} = x_i$) oder für eine negierte Variable (d.h. $y_{j_k} = \neg x_i$) 1 steht. Die $a_1, ..., a_k$ und b werden in Dezimaldarstellung angegeben und wie folgt bestimmt.

$$b = \underbrace{4...4}_{m\text{-mal}}\underbrace{1...1}_{n\text{-mal}} = \sum_{i=n}^{n+m-1} 4 \cdot 10^i + \sum_{i=0}^{n-1} 10^i \ .$$

Die Zahl b besteht also aus zwei Ziffernblöcken: der erste (führende) Ziffernblock aus m Dezimalziffern enthält nur die Ziffer 4, der zweite Ziffernblock aus n Dezimalziffern nur die Ziffer 1.

Es werden n Dezimalzahlen $v_1, ..., v_n$ bestimmt, die wie b jeweils aus $m+n$ Dezimalziffern bestehen. In v_i für $i = 1, ..., n$ werden die Ziffernpositionen im ersten (führenden) Ziffernblock aus m Dezimalziffern von links nach rechts mit 1, ,,, m durchnumeriert. In v_i steht im ersten Ziffernblock an Position j der Wert

$$\begin{cases} 0, \text{falls das Literal } x_i \text{ in der Klausel } F_j \text{ nicht vorkommt} \\ 1, \text{falls das Literal } x_i \text{ in der Klausel } F_j \text{ genau einmal vorkommt} \\ 2, \text{falls das Literal } x_i \text{ in der Klausel } F_j \text{ genau zweimal vorkommt} \\ 3, \text{falls das Literal } x_i \text{ in der Klausel } F_j \text{ genau dreimal vorkommt.} \end{cases}$$

Die Positionen im zweiten Ziffernblock in v_i aus n Dezimalziffern werden ebenfalls von links nach rechts mit 1, ..., n durchnumeriert. In v_i steht im zweiten Ziffernblock an Position j der Wert

$$\begin{cases} 0 \text{ für } i \neq j \\ 1 \text{ für } i = j. \end{cases}$$

Weiterhin werden n Dezimalzahlen $v'_1, ..., v'_n$ bestimmt, die wie b jeweils aus $m+n$ Dezimalziffern bestehen. In v'_i für $i = 1, ..., n$ werden auf gleiche Weise die Ziffernpositionen im ersten (führenden) und zweiten Ziffernblock jeweils von links nach rechts durchnumeriert. In v'_i steht im ersten Ziffernblock an Position j der Wert

$$\begin{cases} 0, \text{falls das Literal } \neg x_i \text{ in der Klausel } F_j \text{ nicht vorkommt} \\ 1, \text{falls das Literal } \neg x_i \text{ in der Klausel } F_j \text{ genau einmal vorkommt} \\ 2, \text{falls das Literal } \neg x_i \text{ in der Klausel } F_j \text{ genau zweimal vorkommt} \\ 3, \text{falls das Literal } \neg x_i \text{ in der Klausel } F_j \text{ genau dreimal vorkommt.} \end{cases}$$

Im zweiten Ziffernblock steht an Position j der Wert

$\begin{cases} 0 \text{ für } i \neq j \\ 1 \text{ für } i = j. \end{cases}$

Zusätzlich werden *m* Dezimalzahlen $c_1, \ldots, c_m$ bestimmt, die jeweils aus $m+n$ Dezimalziffern bestehen. Auch in c_i für $i = 1, \ldots, m$ werden die Ziffernpositionen im ersten (führenden) Ziffernblock aus *m* Dezimalziffern von links nach rechts mit 1, ,,, *m* durchnumeriert. In c_i steht im ersten Ziffernblock an Position *j* der Wert

$\begin{cases} 0 \text{ für } i \neq j \\ 1 \text{ für } i = j. \end{cases}$

Die *n* Positionen im zweiten Ziffernblock in c_i enthalten an allen Positionen den Wert 0.

Schließlich werden *m* Dezimalzahlen $d_1, \ldots, d_m$ bestimmt, die jeweils aus $m+n$ Dezimalziffern bestehen. Auch in d_i für $i = 1, \ldots, m$ werden die Ziffernpositionen im ersten (führenden) Ziffernblock aus *m* Dezimalziffern von links nach rechts durchnumeriert. In d_i steht im ersten Ziffernblock an Position *j* der Wert

$\begin{cases} 0 \text{ für } i \neq j \\ 2 \text{ für } i = j. \end{cases}$

Die *n* Positionen im zweiten Ziffernblock in d_i enthalten an allen Positionen den Wert 0.

Es wird

$$f(F) = [a_1, ,,,, a_k, b]$$
$$= [v_1, \ldots, v_n, v_1', \ldots, v_n', c_1, \ldots, c_m, d_1, \ldots, d_m, b]$$

gesetzt; d.h. die Zahlen $a_1, \ldots, a_k$ mit $k = 2 \cdot (n+m)$ sind $a_1 = v_1, \ldots, a_n = v_n$, $a_{n+1} = v_1', \ldots, a_{2n} = v_n'$, $a_{2n+1} = c_1, \ldots, a_{2n+m} = c_m$, $a_{2n+m+1} = d_1, \ldots, a_{2\cdot(n+m)} = d_m$.

Die Bestimmung von $f(F)$ erfolgt deterministisch, und es ist $size(f(F)) \in O((size(F))^2)$.

Es sei $F = F_1 \wedge \ldots \wedge F_m$ erfüllbar. Es wird eine Auswahl $J \subseteq \{1, \ldots, k\}$ mit $\sum_{j \in J} a_j = b$ angegeben:

Jede Variable x_i für $i = 1, \ldots, n$ ist mit einem Wahrheitswert TRUE oder FALSE belegt. In die Indexmenge *J* wird *i* aufgenommen, wenn x_i mit TRUE belegt wurde (entsprechend der Zahl v_i); in die Indexmenge *J* wird $n+i$ aufgenommen, wenn x_i mit FALSE belegt wurde (entsprechend der Zahl v_i'). Mit der gegenwärtigen Indexmenge *J* sei $S = \sum_{a_i \in J} a_i$. Dann besteht *S* aus $n+m$ Dezimalziffern mit folgender Eigenschaft: der erste (führende) Ziffernblock aus *m* Dezimalziffern enthält nur die Ziffern 1 oder 2 oder 3; der anschließende Ziffernblock aus *n* Dezimalziffern enthält nur die Ziffer 1. Diese Feststellung ergibt sich aus der Tatsache, dass in jeder Klausel F_j für $j = 1, \ldots, m$ mindestens ein Literal und höchstens drei Literale mit TRUE belegt sind und in *J* nicht gleichzeitig die Indizes *i* (entsprechend der Zahl v_i) und $n+i$

(entsprechend der Zahl v_i') aufgenommen wurden. Numeriert man die Positionen im ersten Ziffernblock von *S* wieder von links nach rechts mit 1, ..., *m*, so werden folgende Indizes zusätzlich in *J* aufgenommen: $2n+j$ (entsprechend der Zahl c_j), falls die *j*-te Ziffer im ersten Ziffernblock von *S* gleich 3 ist, $2n+m+j$ (entsprechend der Zahl d_j), falls die *j*-te Ziffer im ersten Ziffernblock von *S* gleich 2 ist, $2n+j$ und $2n+m+j$ (entsprechend den Zahlen c_j und d_j), falls die *j*-te Ziffer im ersten Ziffernblock von *S* gleich 1 ist. Es gilt nun $\sum_{j\in J} a_j = b$.

Es gebe umgekehrt eine Auswahl $J \subseteq \{1,...,2\cdot(n+m)\}$ mit $\sum_{j\in J} a_j = b$. Da nur die Zahlen v_i bzw. v_i' die Ziffer 1 im zweiten Ziffernblock von *b* an Position *i* für $i = 1,...,n$ entstehen lassen können, ist entweder *i* (entsprechend der Zahl v_i) oder $n+i$ (entsprechend der Zahl v_i') in der Auswahl *J*. Im ersten Fall wird x_i mit TRUE belegt, im zweiten Fall mit FALSE. Es bleibt zu zeigen, dass jede Klausel F_j in *F* den Wahrheitswert TRUE erhält. Die Ziffer 4 an Position *j* im ersten Ziffernblock von *b* kann nicht allein durch Summation der Zahlen c_j bzw. d_j entstanden sein. Das bedeutet, dass für diese Position *j* eine Zahl v_i oder v_i', aber nicht beide, einen Anteil liefert. Im ersten Fall kommt die mit TRUE belegte Variable x_i in F_j vor, im zweiten Fall kommt die mit FALSE belegte Variable x_i in der Form $\neg x_i$ in F_j vor. In beiden Fällen erhält die Klausel F_j den Wahrheitswert TRUE.

///

Satz 5.5-3:
RUCKSACK $\leq_m^p$ PARTITION

Mit $L_{\text{RUCKSACK}} = \left\{ [a_1,\dots,a_n,b] \,\middle|\, \text{Es gibt eine Teilmenge } J \subseteq \{1,...,n\} \text{mit} \sum_{j\in J} a_j = b \right\}$ und

$L_{\text{PARTITION}} = \left\{ [a_1,\dots,a_k] \,\middle|\, \text{Es gibt eine Teilmenge } J \subseteq \{1,...,k\} \text{mit} \sum_{j\in J} a_j = \sum_{j\notin J} a_j \right\}$

steht die Aussage des Satzes für $L_{\text{RUCKSACK}} \leq_m^p L_{\text{PARTITION}}$.

Beweis:

Es sei $I = [a_1,\dots,a_n,b]$ eine Instanz von RUCKSACK, die aus *n* natürlichen Zahlen mit $a_i > 0$ für $i = 1,...,n$ und einer natürlichen Zahl $b > 0$ besteht. Es wird $M = \sum_{i=1}^{n} a_i$ gesetzt. Der Instanz *I* wird die Instanz $I' = [a_1,\dots,a_n, M-b+1, b+1]$ für PARTITION zugeordnet, die

aus $k = n+2$ Zahlen besteht ($a_{n+1} = M - b + 1$, $a_{n+2} = b + 1$). Diese Zuordnung kann deterministisch polynomiell (in $size(I)$) zeitbeschränkt erfolgen, da die Berechnung von M mit einem Aufwand der Ordnung $O\left((size(I))^2\right)$ erfolgen kann.

Es gelte $I \in L_{\text{RUCKSACK}}$, d.h. es gibt eine Teilmenge $J \subseteq \{1, ..., n\}$ mit $\sum_{j \in J} a_j = b$. Es wird $J' = J \cup \{n+1\}$ gesetzt. Dann gilt:

$M = \sum_{i=1}^{n} a_i = \sum_{j \in J} a_j + \sum_{j \notin J} a_j = b + \sum_{j \notin J} a_j$ und damit

$\sum_{j \in J'} a_j = \sum_{j \in J} a_j + (M - b + 1) = b + b + \sum_{j \notin J} a_j - b + 1 = \sum_{j \notin J} a_j + (b+1) = \sum_{j \notin J} a_j + a_{n+2} = \sum_{j \notin J'} a_j$, d.h. $I' \in L_{\text{PARTITION}}$.

Sei umgekehrt $I' \in L_{\text{PARTITION}}$ mit der Auswahl $J' \subseteq \{1, ..., n+2\}$.
Sind weder $n+1$ noch $n+2$ in J', so ist

$\sum_{j \in J'} a_j \leq \sum_{i=1}^{n} a_i = M < M - b + 1 + b + 1 = a_{n+1} + a_{n+2} \leq \sum_{j \notin J'} a_j$, so dass $I' \notin L_{\text{PARTITION}}$ wäre. Also ist mindestens einer der Indizes $n+1$ und $n+2$ in J'.

Wegen $a_{n+1} + a_{n+2} = (M - b + 1) + (b + 1) > M = \sum_{i=1}^{n} a_i$ können aber nicht beide Indizes $n+1$ und $n+2$ in J' sein.

1. Fall: $n+1 \in J'$, $n+2 \notin J'$

Dann ist $I = [a_1, ..., a_n, b]$ in L_{RUCKSACK} mit der Indexmenge $J = J' \setminus \{n+1\}$ (zu beachten ist, dass wegen $n+2 \notin J'$ die so definierte Indexmenge J die Beziehung $J \subseteq \{1, ..., n\}$ erfüllt), denn:

$\sum_{j \in J'} a_j = \sum_{\substack{j \in J' \\ j \neq n+1}} a_j + a_{n+1} = \sum_{j \notin J'} a_j = \sum_{\substack{j \notin J' \\ j \in \{1,...,n\}}} a_j + a_{n+2}$ und $M = \sum_{i=1}^{n} a_i = \sum_{\substack{j \in J' \\ j \in \{1,...,n\}}} a_j + \sum_{\substack{j \notin J' \\ j \in \{1,...,n\}}} a_j$ und damit

$\sum_{\substack{j \in J' \\ j \in \{1,...,n\}}} a_j + \sum_{\substack{j \in J' \\ j \in \{1,...,n\}}} a_j + \sum_{\substack{j \notin J' \\ j \in \{1,...,n\}}} a_j - b + 1 = \sum_{\substack{j \in J' \\ j \in \{1,...,n\}}} a_j + M - b + 1 = \sum_{\substack{j \in J' \\ j \in \{1,...,n\}}} a_j + a_{n+1} = \sum_{\substack{j \notin J' \\ j \in \{1,...,n\}}} a_j + a_{n+2}$

(die letzte Gleichung gilt wegen $I' \in L_{\text{PARTITION}}$). Damit ergibt sich nacheinander, indem man in der linken und rechten Seite der Gleichung den Ausdruck $\sum_{\substack{j \notin J' \\ j \in \{1,...,n\}}} a_j + 1$ abzieht:

$2 \cdot \sum_{\substack{j \in J' \\ j \in \{1,...,n\}}} a_j - b = b + 1 - 1$, $2 \cdot \sum_{\substack{j \in J' \\ j \in \{1,...,n\}}} a_j = 2 \cdot b$ und $\sum_{\substack{j \in J' \\ j \in \{1,...,n\}}} a_j = \sum_{j \in J} a_j = b$.

2. Fall: $n+2 \in J'$, $n+1 \notin J'$

Dann ist $I = [a_1, \ldots, a_n, b]$ in L_{RUCKSACK} mit der Indexmenge $J = \{1, \ldots, n\} \setminus J'$, denn:

$\sum_{j \in J'} a_j = \sum_{\substack{j \in J' \\ j \neq n+2}} a_j + a_{n+2} = \sum_{\substack{j \in J' \\ j \neq n+2}} a_j + b + 1 = \sum_{j \notin J'} a_j = \sum_{\substack{j \notin J' \\ j \in \{1,\ldots,n\}}} a_j + a_{n+1} = \sum_{\substack{j \notin J' \\ j \in \{1,\ldots,n\}}} a_j + M - b + 1$. Die Terme der Gleichung werden sortiert, und es ergibt sich:

$$2b = \sum_{\substack{j \notin J' \\ j \in \{1,\ldots,n\}}} a_j - \sum_{\substack{j \in J' \\ j \neq n+2}} a_j + M = \sum_{\substack{j \notin J' \\ j \in \{1,\ldots,n\}}} a_j - \sum_{\substack{j \in J' \\ j \neq n+2}} a_j + \sum_{\substack{j \notin J' \\ j \in \{1,\ldots,n\}}} a_j + \sum_{\substack{j \notin J' \\ j \in \{1,\ldots,n\}}} a_j = 2 \cdot \sum_{\substack{j \notin J' \\ j \in \{1,\ldots,n\}}} a_j = 2 \cdot \sum_{j \in J} a_j ,$$

also auch in diesem Fall $\sum_{j \in J} a_j = b$.

///

Satz 5.5-4:

PARTITION $\leq_m^p$ BINPACKING

Mit $L_{\text{PARTITION}} = \left\{ [a_1, \ldots, a_n] \,\middle|\, \text{Es gibt eine Teilmenge } J \subseteq \{1, \ldots, n\} \text{ mit } \sum_{j \in J} a_j = \sum_{j \notin J} a_j \right\}$ und

$L_{\text{BINPACKING}} = \left\{ [a_1, \ldots, a_h, b, k] \,\middle|\, \begin{array}{l} \text{Die Objekte } a_1, \ldots, a_h \text{ lassen sich so auf } k \text{ Behälter} \\ \text{der Behältergröße } b \text{ aufteilen, dass kein Behälter überläuft} \end{array} \right\}$

steht die Aussage des Satzes für $L_{\text{PARTITION}} \leq_m^p L_{\text{BINPACKING}}$.

Beweis:

Es sei $I = [a_1, \ldots, a_n]$ eine Instanz von PARTITION, die aus n natürlichen Zahlen mit $a_i > 0$ für $i = 1, \ldots, n$ besteht. Der Instanz I wird die Instanz $I' = [a_1, \ldots, a_n, b, 2]$ für BINPACKING mit $b = \frac{1}{2} \cdot \left\lfloor \sum_{i=1}^{n} a_i \right\rfloor$ zugeordnet. Diese Zuordnung kann deterministisch polynomiell (in $size(I)$) zeitbeschränkt erfolgen, da die Berechnung von M mit einem Aufwand der Ordnung $O\left((size(I))^2\right)$ erfolgen kann.

Es gelte $I \in L_{\text{PARTITION}}$, d.h. es gibt eine Teilmenge $J \subseteq \{1, \ldots, n\}$ mit $\sum_{j \in J} a_j = \sum_{j \notin J} a_j$. Daher ist $\sum_{i=1}^{n} a_i$ gerade und $b = \frac{1}{2} \cdot \left\lfloor \sum_{i=1}^{n} a_i \right\rfloor = \frac{1}{2} \cdot \sum_{i=1}^{n} a_i$. Außerdem gilt $\sum_{i=1}^{n} a_i = \sum_{j \in J} a_j + \sum_{j \notin J} a_j = 2 \cdot \sum_{j \in J} a_j$.

Für die BINPACKING-Instanz I' nimmt der erste Behälter alle Eingabewerte a_j mit Index $j \in J$ und der zweite Behälter alle Eingabewerte a_j mit Index $j \notin J$ auf. Die Füllhöhe im ersten Behälter beträgt $\sum_{j \in J} a_j = \frac{1}{2} \cdot \sum_{i=1}^{n} a_i = b$, die Füllhöhe im zweiten Behälter

$$\sum_{j \notin J} a_j = \sum_{j \in J} a_j = \frac{1}{2} \cdot \sum_{i=1}^{n} a_i = b .$$

Ist umgekehrt $I' = [a_1, \ldots, a_n, b, 2]$ in $L_{\text{BINPACKING}}$, so wird mit $J = \{ j \mid a_j \text{ liegt im ersten Behälter} \}$ eine Indexmenge für I bestimmt, so dass $I \in L_{\text{PARTITION}}$ ist.

///

Satz 5.5-5:
3-CSAT $\leq_m^p$ FÄRBBARKEIT

Mit $L_{\text{3-CSAT}} = \left\{ F \;\middle|\; \begin{array}{l} F \text{ ist ein Boolescher Ausdruck in konjunktiver Normalform,} \\ \text{jede Klausel enthält genau 3 Literale,} \\ \text{und } F \text{ ist erfüllbar} \end{array} \right\}$ und

$L_{\text{FÄRBBARKEIT}} = \left\{ [G,k] \;\middle|\; \begin{array}{l} G \text{ ist ein ungerichteter Graph, der eine zulässige} \\ \text{Knotenfärbung mit } k \text{ Farben besitzt} \end{array} \right\}$ steht die Aussage des Satzes für $L_{\text{3-CSAT}} \leq_m^p L_{\text{FÄRBBARKEIT}}$.

Beweis:

Es sei *F* eine Instanz von 3-CSAT, d.h. *F* ist ein Booleschen Ausdruck in konjunktiver Normalform mit der Variablenmenge $\{ x_1, \ldots, x_n \}$ und *m* Klauseln, die jeweils genau drei Literale enthalten. Es kann angenommen werden, dass eine Klausel mit einem Literal *y* nicht auch das Literal $\neg y$ enthält; denn dann ist sie trivialerweise erfüllbar.

Dem Ausdruck $F = F_1 \wedge \ldots \wedge F_m$ wird auf folgende Weise ein ungerichteter Graph $G = (V, E)$ zugeordnet.

Für jede Variable x_i wird ein Knoten v_i und ein Knoten v_i' in *V* aufgenommen. Zusätzlich enthält *V* *n* Knoten, die mit w_1, ..., w_n bezeichnet werden. Für jede Klausel F_j enthält *V* einen Knoten, der mit f_j benannt wird. Man könnte die Knoten auch einheitlich mit den Nummern 1, ..., $2 \cdot n + m$ nummerieren, wobei die obigen Knoten v_i die Nummern 1, ..., *n*,

die Knoten v_i' die Nummern $n+1, ..., 2 \cdot n$ und die Knoten f_j die Nummern $2 \cdot n+1, ..., 2 \cdot n+m$ erhalten; die hier gewählte Darstellung erleichtert jedoch die Beschreibung der Reduktion.

Die ungerichteten Kanten von *G* sind:
(w_i, w_j) mit $i = 1, ..., n$, $j = 1, ..., n$ und $i \neq j$,
(w_i, v_j) und (w_i, v_j') mit $i = 1, ..., n$, $j = 1, ..., n$ und $i \neq j$,
(v_i, v_i') für $i = 1, ..., n$,
(v_i, f_j), falls x_i in F_j nicht vorkommt, und (v_i', f_j), falls $\neg x_i$ in F_j nicht vorkommt, mit $i = 1, ..., n$ und $j = 1, ..., m$.

Die Anzahl *k* zu verwendender Farben ist $k = n+1$.

Die Konstruktion von $[G, k]$ kann mit einem in $size(F)$ polynomiellen deterministischen Verfahren erfolgen.

Es sei *F* erfüllbar. Der konstruierte Graphen *G* lässt sich zulässig mit $n+1$ Farben färben:

Die Knoten w_1, ..., w_n bilden einen vollständigen Teilgraph, so dass zu dessen zulässiger Färbung *n* Farben, etwa mit den Nummern 1, ..., *n*, benötigt werden. Jeder Knoten v_j und jeder Knoten v_j' ist mit jedem Knoten w_i mit $i \neq j$ verbunden, so dass v_j und v_j' nicht dieselbe Farbe erhalten können wie die Knoten w_i, aber wie w_j, denn es ist weder v_j noch v_j' mit w_j durch eine Kante verbunden. Da v_j und v_j' durch eine Kante verbunden sind, können sie nicht dieselbe Farbe erhalten. Der Knoten v_j wird mit einer neuen Farbe mit der Nummer $n+1$ und v_j' mit der Farbe von w_j gefärbt, falls $x_j = \mathsf{FALSE}$ ist. Ist $x_j = \mathsf{TRUE}$, so erhält v_j' die Farbe mit der Nummer $n+1$ und v_j die Farbe von w_j. Die Klausel F_j werde durch das Literal $y \in \{ x_i, \neg x_i \}$ erfüllt; für *y* kommen bis zu drei verschiedene derartige Literale in Frage. Ist $y = x_i = \mathsf{TRUE}$, so ist f_j nicht mit v_i verbunden, und v_i hat die Farbe von w_i, also eine Farbe mit einer Nummer zwischen 1 und *n*. Ist $y = \neg x_i = \mathsf{TRUE}$, so ist f_j nicht mit v_i' verbunden, und v_i' hat die Farbe von w_i, also eine Farbe mit einer Nummer zwischen 1 und *n*. In beiden Fällen kann f_j mit der Farbe von w_i gefärbt werden. Der konstruierte Graph besitzt also eine zulässige Knotenfärbung mit $n+1$ Farben.

Besitzt umgekehrt der konstruierte Graph eine zulässige Knotenfärbung mit $n+1$ Farben, so werden für den vollständigen Teilgraphen, der aus den Knoten w_1, ..., w_n besteht, *n* Farben, etwa mit den Nummern 1, ..., *n*, benötigt. Wie oben folgt, dass genau einer der Knoten v_j o-

der v'_j dieselbe Farbe wie w_j besitzt und der andere der beiden Knoten mit der Farbe mit der Nummer $n+1$ gefärbt ist (es stehen ja nur $n+1$ Farben zur Verfügung). Es kann daher folgende Belegung der Variablen definiert werden:

Es ist $x_j = \mathsf{TRUE}$ genau dann, wenn v'_j die Farbe mit der Nummer $n+1$ trägt.

Wegen $n \geq 4$ gibt es zu jeder Klausel F_j eine Variable x_i, so dass weder x_i noch $\neg x_i$ in F_j vorkommt. Daher ist f_j mit v_i und v'_i jeweils durch eine Kante verbunden und nicht mit der Farbe mit der Nummer $n+1$ gefärbt, sondern mit einer Farbe mit kleinerer Nummer. Es sei $F_j = (y_{j_1} \vee y_{j_2} \vee y_{j_3})$ mit $y_{j_1} \in \{x_{i_1}, \neg x_{i_1}\}$, $y_{j_2} \in \{x_{i_2}, \neg x_{i_2}\}$ und $y_{j_3} \in \{x_{i_3}, \neg x_{i_3}\}$. Die drei Literale seien paarweise verschieden. Ist $y_{j_l} = x_{i_l}$, dann ist f_j nach Konstruktion von *G* mit v'_{i_l} über eine Kante verbunden; ist $y_{j_l} = \neg x_{i_l}$, dann ist f_j mit v_{i_l} verbunden. In beiden Fällen werde der Knoten, der mit f_j aufgrund des Auftretens von y_{j_l} in F_j verbunden ist, der zu y_{j_l} gehörende komplementäre Knoten genannt. Angenommen, die zu den drei Literalen in F_j gehörenden komplementären Knoten wären jeweils mit einer Farbe Nummer $\leq n$ gefärbt. Nach Konstruktion ist f_j mit $2 \cdot n - 3$ Knoten verbunden (darunter die drei komplementären Knoten). Von diesen Knoten tragen $n-3$ die Farbe mit der Nummer $n+1$ und (einschließlich der drei komplementären Knoten) $n-3+3=n$ Knoten eine Farbe mit Nummer $\leq n$ (diese Farben sind aufgrund der Konstruktion von *G* paarweise verschieden). Da f_j auch eine Farbe mit Nummer $\leq n$ besitzt und mit *n* Knoten mit *n* unterschiedlichen Farben mit Nummern $\leq n$ verbunden ist, ist die Färbung der Knoten von *G* nicht zulässig. Daher gibt es in F_j mindestens ein Literal, dessen zugehöriger komplementärer Knoten mit der Farbe mit Nummer $n+1$ gefärbt ist. Nach Definition der Belegung der Variablen in *F* hat dieses Literal den Wahrheitswert TRUE, und F_j ist erfüllbar. Sind die drei Literale in F_j nicht paarweise verschieden, so kann man genauso argumentieren.

///

In Satz 5.5-2 wird 3-CSAT $\leq_m^p$ RUCKSACK gezeigt, d.h. ein Boolescher Ausdruck wird auf eine Zahlenmenge abgebildet, so dass die Strukturinformation des Ausdrucks in der Zahlenmenge kodiert ist. In den beiden folgenden Sätzen wird die Strukturinformation eines Booleschen Ausdrucks in Form eines ungerichteten bzw. gerichteten Graphen kodiert.

Satz 5.5-6:

3-CSAT $\leq_m^p$ KLIQUE

Mit $L_{\text{3-CSAT}} = \left\{ F \;\middle|\; \begin{array}{l} F \text{ ist ein Boolescher Ausdruck in konjunktiver Normalform,} \\ \text{jede Klausel enthält genau 3 Literale,} \\ \text{und } F \text{ ist erfüllbar} \end{array} \right\}$ und

$L_{\text{KLIQUE}} = \left\{ [G,k] \,\middle|\, G \text{ ist ein ungerichteter Graph und besitzt eine Klique der Größe } k \right\}$

steht die Aussage des Satzes für $L_{\text{3-CSAT}} \leq_m^p L_{\text{KLIQUE}}$.

Beweis:

Es sei *F* eine Instanz von 3-CSAT, d.h. *F* ist ein Booleschen Ausdruck in konjunktiver Normalform mit der Variablenmenge $\{x_1, ..., x_n\}$ und *m* Klauseln, die jeweils genau drei Literale enthalten:
$F = F_1 \wedge ... \wedge F_m$, und jedes F_j hat die Form $F_j = (y_{j_1} \vee y_{j_2} \vee y_{j_3})$, wobei y_{j_k} ein Literal ist, d.h. für eine Variable (d.h. $y_{j_k} = x_i$) oder für eine negierte Variable (d.h. $y_{j_k} = \neg x_i$) steht.

Dem Ausdruck *F* wird auf folgende Weise ein ungerichteter Graph $G = (V, E)$ zugeordnet.

Es sei $V = \{\langle j,l \rangle \,|\, 1 \leq j \leq m, 1 \leq l \leq 3\}$. Der Knoten $\langle j,l \rangle$ steht für die Klausel F_j und das *l*-te Literal in F_j.
In *E* wird eine Kante $(\langle j,l \rangle, \langle k,h \rangle)$ für $j \neq k$ und $y_{j_l} \neq \neg y_{k_h}$ aufgenommen. Anschaulich bedeutet dieses, dass zwei Knoten $\langle j,l \rangle$ und $\langle k,h \rangle$ dann durch eine Kante verbunden werden, wenn die den Knoten entsprechenden Literale in unterschiedlichen Klauseln vorkommen und die beiden Literale gleichzeitig mit demselben Wahrheitswert belegt werden können (es ist entweder $y_{j_l} = y_{k_h}$ oder y_{j_l} und y_{k_h} sind komplementierte oder nichtkomplementierte Versionen unterschiedlicher Variablen).
Offensichtlich ist $[G, m]$ ist eine Instanz von KLIQUE mit $size(G) \in O((size(F))^2)$.

Ist *F* mit einer Belegung der Variablen $x_1, ..., x_n$ erfüllbar, so ist jede Klausel F_j erfüllbar, d.h. in jeder Klausel F_j gibt es mindestens ein Literal y_{j_l} mit Wert TRUE. Es seien F_j und F_k verschiedene Klauseln und y_{j_l} bzw. y_{k_h} Literale in F_j bzw. F_k, die mit TRUE belegt sind. Dann ist $y_{j_l} \neq \neg y_{k_h}$, und *E* enthält eine Kante $(\langle j,l \rangle, \langle k,h \rangle)$. Insgesamt sind je zwei unterschiedliche Knoten der Form $\langle j,l \rangle$ und $\langle k,h \rangle$ mit $j \in \{1, ..., m\}$, $k \in \{1, ..., m\}$ und $j \neq k$ durch eine Kante in *G* verbunden, d.h. *G* enthält eine Klique der Größe *m*: $[G, m] \in L_{\text{KLIQUE}}$.

Enthält umgekehrt der konstruierte Graph *G* eine Klique der Größe *m*, so haben alle Knoten $\langle j,l \rangle$ dieser Klique eine unterschiedliche erste Komponente *j*. Jeder Knoten korrespondiert nach Konstruktion zu einem Literal y_{j_l} in F_j. Außerdem gilt für die Literale y_{j_l} und y_{k_h}, die zu unterschiedlichen Knoten $\langle j,l \rangle$ und $\langle k,h \rangle$ dieser Klique korrespondieren: $y_{j_l} \neq \neg y_{k_h}$. Daher erhält man durch folgende Vorschrift eine widerspruchsfreie Belegung der Literale, die zu der gefundenen Klique korrespondieren bzw. zu den Variablen, die in diesen Literalen vorkommen:

Ist $y_{j_l} = x_i$, so wird x_i mit TRUE belegt. Ist $y_{j_l} = \neg x_i$, so wird x_i mit FALSE belegt.

Alle bisher nicht belegten Variablen in *F* können mit beliebigen Wahrheitswerten belegt werden.

Diese Belegung erfüllt jede Klausel $F_1, ..., F_m$ und damit *F*.

///

Satz 5.5-7:

3-CSAT $\leq_m^p$ GERICHTETER HAMILTONKREIS

Mit $L_{\text{3-CSAT}} = \left\{ F \;\middle|\; \begin{array}{l} F \text{ ist ein Boolescher Ausdruck in konjunktiver Normalform,} \\ \text{jede Klausel enthält genau 3 Literale,} \\ \text{und } F \text{ ist erfüllbar} \end{array} \right\}$ und

$L_{\text{GERICHTETER HAMILTONKREIS}} = \left\{ G \;\middle|\; \begin{array}{l} G \text{ ist ein gerichteter Graph,} \\ \text{und } G \text{ besitzt einen Hamiltonkreis} \end{array} \right\}$

steht die Aussage des Satzes für $L_{\text{3-CSAT}} \leq_m^p L_{\text{GERICHTETER HAMILTONKREIS}}$.

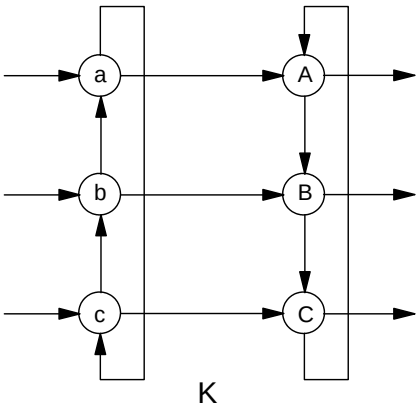
Beweis:

Es sei *F* eine Instanz von 3-CSAT, d.h. *F* ist ein Booleschen Ausdruck in konjunktiver Normalform mit der Variablenmenge $\{x_1, ..., x_n\}$ und *m* Klauseln, die jeweils genau drei Literale enthalten:

$F = F_1 \wedge ... \wedge F_m$, und jedes F_j die Form $F_j = (y_{j_1} \vee y_{j_2} \vee y_{j_3})$, wobei y_{j_k} ein Literal ist, d.h. für eine Variable (d.h. $y_{j_k} = x_i$) oder für eine negierte Variable (d.h. $y_{j_k} = \neg x_i$) steht.

Dem Ausdruck *F* wird ein gerichteter Graph $G = (V,E)$ mit Knotenmenge $V = \{v_1, ..., v_k\}$ und Kantenmenge $E \subseteq V \times V$ zugeordnet, so dass *F* genau dann erfüllbar ist, wenn *G* einen Hamiltonkreis besitzt. Diese Zuordnung erfolgt durch ein deterministisch polynomiell (in $size(F)$) zeitbeschränktes algorithmisches Verfahren.

Zunächst werden *n* Knoten $v_1, \ldots, v_n$ in *V* aufgenommen (sie entsprechen den Variablen $x_1, ..., x_n$), aus denen jeweils zwei Kanten herausführen, die als „erster Ausgang“ bzw. „zweiter Ausgang“ bezeichnet werden. Außerdem führen in einen Knoten jeweils zwei Kanten hinein, die als „erster Eingang“ bzw. „zweiter Eingang“ bezeichnet werden. Die End- bzw. Zielknoten einer Kante werden unten spezifiziert. Zusätzlich enthält *G* genau *m* Kopien $K_1, ..., K_j$ des folgenden Teilgraphen *K*:



Jede Kopie K_j steht für eine Klausel F_j. Der Teilgraph hat 3 Eingänge, die mit den Knoten *a*, *b* und *c* identifiziert werden, und drei Ausgänge, die den Knoten *A*, *B* und *C* entsprechen.

Es sei $F_{i,1}, ..., F_{\hat{\imath},j_i}$ eine Aufzählung der Klauseln, die das Literal x_i enthalten; die den Klauseln entsprechenden Teilgraphen sind $K_{i,1}, ..., K_{\hat{\imath},j_i}$. Die erste aus v_i herausführende Kante wird mit dem Eingang *a* von $K_{i,1}$ verbunden, wenn x_i in $F_{i,1}$ an Position 1 vorkommt, bzw. mit dem Eingang *b* von $K_{i,1}$, wenn x_i in $F_{i,1}$ an Position 2 vorkommt, bzw. mit dem Eingang *c* von $K_{i,1}$, wenn x_i in $F_{i,1}$ an Position 3 vorkommt. Im Fall, dass in $K_{i,1}$ mit Eingang *a* verbunden wurde, wird der Ausgang *A* von $K_{i,1}$ mit einem Eingang von $K_{i,2}$ verbunden (und zwar dort mit dem Eingang *a*, *b* bzw. *c*, je nachdem, ob x_i in $F_{i,2}$ an Position 1, 2 bzw. 3 vorkommt). Im Fall, dass in $K_{i,1}$ mit Eingang *b* verbunden wurde, wird der Ausgang *B* von $K_{i,1}$ mit einem Eingang von $K_{i,2}$ auf entsprechende Weise verbunden. Im Fall, dass in $K_{i,1}$ mit Eingang *c* verbunden wurde, wird der Ausgang *C* von $K_{i,1}$ mit einem Eingang von $K_{i,2}$ verbunden. Kommt x_i in $F_{i,1}$ mehrmals vor, so übernimmt $F_{i,1}$ zunächst die Rolle von $F_{i,2}$. Der auf diese Weise bestimmt Ausgang des Teilgraphen $K_{\hat{\imath},j_i}$ wird mit dem ersten Eingang des Knotens v_{i+1} verbunden. Kommt ein Literal x_i in *F* überhaupt nicht vor, so wird der erste Ausgang von v_i direkt mit dem ersten Eingang von v_{i+1} verbunden. Ein zu verbindender Ausgänge von K_{n,j_n} wird mit dem ersten Eingang von v_1 rückgekoppelt bzw. im Fall, dass das

Literal x_n in *F* nicht vorkommt, wird der erste Ausgang von v_n mit dem ersten Eingang von v_1 verbunden.

Auf gleiche Weise wird mit dem Literal $\neg x_i$ verfahren, nur werden jetzt statt des ersten Ausgangs bzw. ersten Eingangs der zweite Ausgang bzw. zweite Eingang an den Knoten v_i und v_{i+1} genommen.

Die Teilgraphen K_j haben folgende Eigenschaft:

> Wenn der den Teilgraphen K_j umgebende Graph einen Hamiltonkreis besitzt, so verlässt dieser den Teilgraphen K_j am Ausgang *A*, falls er in K_j beim Eingang *a* hineinläuft; er verlässt K_j am Ausgang *B*, falls er in K_j beim Eingang *b* hineinläuft; er verlässt K_j am Ausgang *C*, falls er in K_j beim Eingang *c* hineinläuft.

Diese Eigenschaft kann man leicht durch Nachvollzug aller möglichen Fälle überprüfen.

Die Konstruktion von *G* lässt sich durch einen deterministischen Algorithmus durchführen. Die Größe von *G* ist polynomiell in $size(F)$ beschränkt.

Ist *F* erfüllbar, so wird durch folgende Vorschrift ein Hamiltonkreis in *G* beschrieben: Man startet in einem Knoten v_i, etwa in v_1, und verlässt den Knoten über den ersten Ausgang, falls x_i mit TRUE belegt ist, ansonsten über den zweiten Ausgang. Erreicht man einen Teilgraphen K_j, so wird dieser nach einer drei Möglichkeiten durchlaufen, je nachdem, wie viele weitere Literale in F_j mit TRUE belegt sind. Falls kein weiteres Literal in F_j mit TRUE belegt ist und K_j am Eingang *a* erreicht wird, durchläuft man K_j in der Reihenfolge *a – c – b – B – C – A*. Falls genau ein weiteres Literal in F_j mit TRUE belegt ist, K_j am Eingang *a* erreicht wird und dieses weitere Literal etwa an den Eingang *b* gekoppelt ist, durchläuft man K_j in der Reihenfolge *a – c – C – A*. Falls zwei weitere Literale in F_j mit TRUE belegt ist und K_j am Eingang *a* erreicht wird, durchläuft man K_j in der Reihenfolge *a –A*. Entsprechend sind die übrigen Durchlaufsmöglichkeiten festgelegt.

Besitzt umgekehrt der konstruierte Graph einen Hamiltonkreis, so wird jeder Knoten v_i genau einmal durchlaufen. Falls der Hamiltonkreis den Knoten am ersten Ausgang verlässt, wird x_i mit TRUE belegt, ansonsten mit FALSE. Da jeder Teilgraph K_j auf dem Hamiltonkreis mindestens einmal passiert wird, ist jede Klausel F_j erfüllt.

///

Satz 5.5-8:
GERICHTETER HAMILTONKREIS $\leq_m^p$ UNGERICHTETER HAMILTONKREIS

Die Sprachklassen, denen die jeweiligen Instanzen entstammen, unterscheiden sich lediglich dadurch, dass die Graphen gerichtet bzw. ungerichtet sind.

Beweis:

Es sei $G = (V, E)$ ein gerichteter Graph. Es wird ein ungerichteter Graph $G' = (V', E')$ aus G auf deterministische Weise in polynomieller Zeit konstruiert:

Für jeden Knoten $v \in V$ werden drei Knoten v', v'' und v''' in V' aufgenommen. Für jede in v hereinführende Kante $(v_1, v) \in E$ wird eine ungerichtete Kante (v_1''', v') in E' aufgenommen. Für jede aus v herausführende Kante $(v, v_1) \in E$ wird eine ungerichtete Kante (v''', v_1') in E' aufgenommen. Außerdem werden ungerichtete Kanten (v', v'') und (v'', v''') in E' aufgenommen.

Besitzt G einen Hamiltonkreis, so offensichtlich auch G'.

Es sei ein Hamiltonkreis in G' gegeben. Man betrachte die aus einem Knoten $v \in V$ gebildeten Knoten v', v'' und v''' in V'. Da alle Knoten in V' durchlaufen werden, insbesondere auch v'', und v'' nur mit v' und v''' über eine ungerichtete Kante verbunden ist, enthält der Hamiltonkreis die ungerichtete Kantenfolge $((v', v''), (v'', v'''))$. Der Vorgängerknoten von v' auf dem Hamiltonkreis ist ein Knoten der Form $v_1''' \in V'$ mit $v_1 \neq v$, da es in G' nach Konstruktion keine ungerichteten Kanten der Form (v', v_1'') und der Form (v', v_1') mit $v_1 \neq v$ gibt. Entsprechend ist der Nachfolgerknoten von v''' auf dem Hamiltonkreis ein Knoten der Form $v_2' \in V'$ mit $v_2 \neq v$. Daher kann in G eine Kantenfolge konstruiert werden, in der nacheinander die Knoten v_1, v und v_2 durchlaufen werden; denn nach Konstruktion von G' enthält der Graph G die gerichteten Kanten (v_1, v) und (v, v_2). Da alle Knoten der Form v'' auf dem Hamiltonkreis in G' genau einmal durchlaufen werden, ist die so konstruierte Kantenfolge in G ein Hamiltonkreis in G.

///

Satz 5.5-9:
UNGERICHTETER HAMILTONKREIS $\leq_m^p$ HANDLUNGSREISENDER

Mit $L_{\text{UNGERICHTETER HAMILTONKREIS}} = \left\{ G \middle| \begin{matrix} G \text{ ist ein ungerichteter Graph,} \\ \text{und } G \text{ besitzt einen Hamiltonkreis} \end{matrix} \right\}$ und

$L_{\text{HANDLUNGSREISENDER}} = \left\{ [G,k] \middle| \begin{matrix} G \text{ ist ein gewichteter gerichteter bzw. ungerichteter Graph,} \\ \text{und } G \text{ besitzt ein Tour mit minimalen Kosten} \leq k \end{matrix} \right\}$

steht die Aussage des Satzes für $L_{\text{UNGERICHTETER HAMILTONKREIS}} \leq_m^p L_{\text{HANDLUNGSREISENDER}}$.

Beweis:

Es $G = (V,E)$ eine Instanz von UNGERICHTETER HAMILTONKREIS mit $V = \{v_1, \ldots, v_n\}$. Dann wird ein gewichteter Graph $G' = (V, E', w)$ mit $E' = V \times V$ und Gewichtsfunktion

$$w(e') = \begin{cases} 1, \text{falls } e' \in E \\ 2, \text{falls } e' \notin E \end{cases}$$

definiert. Der Ausdruck $[G', n]$ definiert eine Instanz von HANDLUNGSREISENDER, und es ist $G \in L_{\text{UNGERICHTETER HAMILTONKREIS}}$ genau dann, wenn $G' \in L_{\text{HANDLUNGSREISENDER}}$ ist:

Jeder Hamiltonkreis in G führt zu einer Tour durch G', wobei nur Kanten aus E verwendet werden. Das Gewicht dieser Tour ist gleich n, d.h. G' besitzt eine Tour mit minimalen Kosten, die höchstens n betragen und damit $G' \in L_{\text{HANDLUNGSREISENDER}}$.

Besitzt umgekehrt G' eine Tour mit minimalen Kosten $\leq n$, so kann auf dieser Tour keine Kante liegen, die nicht bereits in E vorkam, da jeder Knoten aus V genau einmal durchlaufen wird. Diese Tour beschreibt daher einen Hamiltonkreis in G.

///

5.6 Bemerkungen zur Struktur von NP

Im vorliegenden Kapitel werden grundlegende Struktureigenschaften der Klasse **NP** beschrieben.

Satz 5.6-1:

(i) Die Klasse **NP** ist abgeschlossen bezüglich Schnittbildung und Vereinigungsbildung.

(ii) Mit $L \in \mathbf{NP}$ gilt auch $L^* \in \mathbf{NP}$.

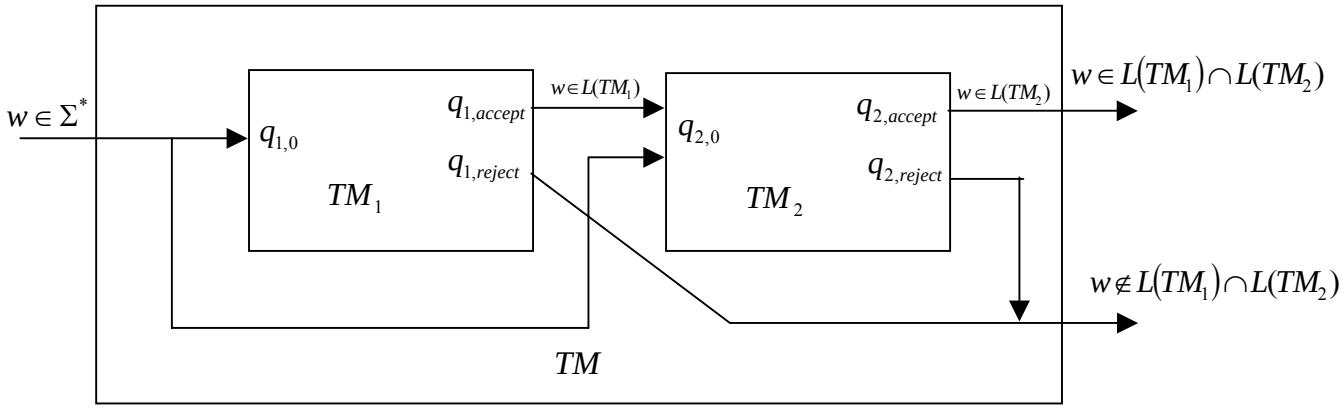
(iii) Die Klasse **P** ist abgeschlossen bezüglich Schnittbildung, Vereinigungsbildung und Komplementbildung.

Bemerkung: Es ist bisher nicht bekannt, ob die Klasse **NP** auch bezüglich Komplementbildung abgeschlossen ist (siehe unten).

Beweis:

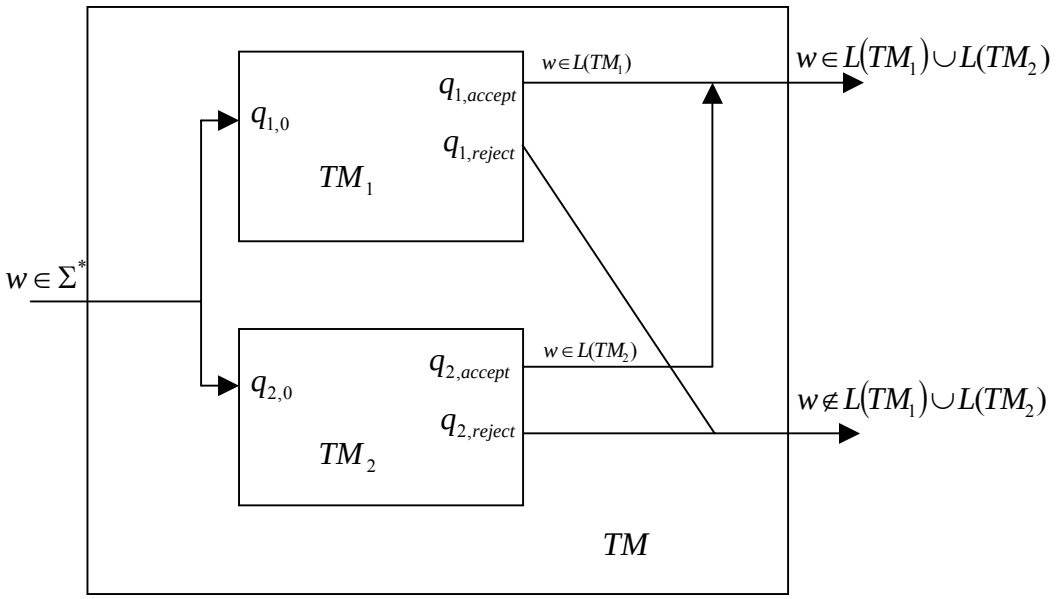
Zu (i): Es seien $L_1 \in \mathbf{NP}$ und $L_2 \in \mathbf{NP}$. Beide Sprachen seien über demselben Alphabet Σ definiert. Zu zeigen ist, dass dann auch $L_1 \cap L_2 \in \mathbf{NP}$ und $L_1 \cup L_2 \in \mathbf{NP}$ gelten.
Zu L_1 bzw. L_2 gibt es nichtdeterministische Turingmaschinen TM_1 bzw. TM_2 mit $L_1 = L(TM_1)$ bzw. $L_2 = L(TM_2)$, die L_1 bzw. L_2 in polynomieller Zeit entscheiden; die dabei auftretenden Polynome seinen p_1 bzw. p_2.

Durch Hintereinanderschaltung kann man wie in Kapitel 2.1 eine nichtdeterministische Turingmaschine *TM* mit $L(TM) = L_1 \cap L_2$ konstruieren. Die Konstruktion ist in folgender Abbildung skizziert:



Ein Eingabewort $w \in \Sigma^*$ wird von TM_1 in einer Schrittanzahl der Ordnung $O(p_1(|w|))$ entschieden. Im Fall $w \in L_1$ startet TM_2 mit Eingabe *w* und entscheidet innerhalb einer Schrittzahl der der Ordnung $O(p_2(|w|))$. Die gesamte Entscheidung hat die Zeitkomplexität $c_1 \cdot p_1(|w|) + c_2 \cdot p_2(|w|)$ mit Konstanten c_1 und c_2, ist also polynomiell.

Zur Akzeptanz von $L_1 \cup L_2$ durch eine nichtdeterministische Turingmaschine *TM* ist ebenfalls eine Konstruktion aus Kapitel 2.1 geeignet, nämlich die Parallelschaltung von TM_1 und TM_2:



Sobald eine der beiden Turingmaschinen TM_1 oder TM_2 das Eingabewort $w \in \Sigma^*$ akzeptiert, akzeptiert *TM* das Wort *w*. Ist $w \notin L_1 \cup L_2$, so stellt TM_1 diese Tatsache nach einer Schrittzahl fest, die durch $c_1 \cdot p_1(|w|)$ beschränkt ist mit einer Konstanten c_1, und TM_2 stellt diese Tatsache nach einer Schrittzahl fest, die durch $c_2 \cdot p_2(|w|)$ mit einer Konstanten c_2 beschränkt ist. Die Zugehörigkeit eines Worts $w \in \Sigma^*$ zu $L_1 \cup L_2$ wird also in der polynomiellen Zeit $\max\{c_1 \cdot p_1(|w|), c_2 \cdot p_2(|w|)\}$ entschieden.

Zu (ii): Wegen $L^* = \bigcup_{i \in \mathbf{N}} L^i$ bedeutet $w \in L^*$: es gibt ein $i \in \mathbf{N}$ mit $w \in L^i$, d.h. $w = w_1 \ldots w_i$ mit $w_k \in L$ für $k = 1, .., i$.

Die nichtdeterministische polynomiell zeitbeschränkte Turingmaschine zur Akzeptanz von *L* sei TM_0, d.h. $L = L(TM_0)$; das beteiligte Polynom sei p_0. Eine nichtdeterministische Turingmaschine zur Akzeptanz von $L^* \subseteq \Sigma^*$ arbeitet wie folgt:

Bei Eingabe von $w \in \Sigma^*$ mit $|w| = n$ werden $i-1 \le n$ Zahlen pos_1, ..., pos_{i-1} mit $1 \le pos_k \le n$ für $k = 1,..,i-1$ in Binärformat nichtdeterministisch geraten. Die Zahlenfolge pos_1, ..., pos_{i-1} zerlegt das Wort $w = a_1 \ldots a_n$ in *i* Teilworte:

$w = \underbrace{a_1 \ldots a_{pos_1}}_{=w_0} \underbrace{a_{pos_1+1} \ldots a_{pos_2}}_{=w_1} \ldots \underbrace{a_{pos_{i-1}+1} \ldots a_n}_{=w_{i-1}} = w_0 w_1 \ldots w_{i-1}$. Das Wort *w* wird genau dann akzeptiert, wenn $w_k \in L(TM_0)$ für $k = 0,..,i-1$ gilt. Dazu wird *i*-mal eine akzeptierende Entscheidung von TM_0 gesucht.

Die Anzahl der Bits in der Zahlenfolge pos_1, ..., pos_{i-1} ist wegen $1 \le pos_k \le n$ für $k = 1,..,i-1$ und $i-1 \le n$ von der Ordnung $O(n \cdot \log(n))$. Alle Teilwörter w_k für $k = 0,..,i-1$ haben eine durch *n* beschränkte Länge. Daher ist der Gesamtaufwand der Berechnung beschränkt durch eine Funktion der Ordnung $O(n \cdot \log(n) + n \cdot p_0(n))$, also polynomiell.

Zu (iii): Die Abgeschlossenheit der Klasse **P** bezüglich Schnitt- und Vereinigungsbildung ergibt sich wie im Beweis zu (i) mit dem Unterschied, dass die dort verwendeten Turingmaschinen deterministisch arbeiten. Die Abgeschlossenheit der Klasse **P** bezüglich Komplementbildung, folgt aus der Tatsache, dass man aus einer polynomiell zeitbeschränkten deterministischen Turingmaschine zur Akzeptanz einer Sprache $L \in \mathbf{P}$, $L \subseteq \Sigma^*$, eine polynomiell zeitbeschränkte deterministische Turingmaschine zur Akzeptanz der Sprache $\Sigma^* \setminus L$ erhält, indem man lediglich den akzeptierenden mit dem verwerfenden Zustand vertauscht.

///

Aus einem polynomiell zeitbeschränkten deterministischen Algorithmus zur Akzeptanz einer Sprache $L \in \mathbf{P}$, $L \subseteq \Sigma^*$, erhält man also durch Vertauschen der beiden `ja`/`nein`-Ausgänge einen polynomiell zeitbeschränkten deterministischen Algorithmus zur Akzeptanz der Sprache $\Sigma^* \setminus L$. Diese Technik funktioniert bei allen deterministischen zeitbeschränkten Sprachklassen mit zeitkonstruierbarer Komplexitätsschranke:

Satz 5.6-2:

Die Funktion $f : \mathbf{N} \to \mathbf{N}$ sei zeitkonstruierbar. Mit co-$TIME(f(n))$ werde die Klasse $\{L \mid L \subseteq \Sigma^* \text{ und } \Sigma^* \setminus L \in TIME(f(n))\}$ bezeichnet. Dann gilt

$TIME(f(n)) = \text{co-}TIME(f(n))$.

Zu beachten ist, dass $TIME(f(n))$ nur deterministische Sprachen enthält.

Bei nichtdeterministischen Sprachklassen kann man im allgemeinen die Technik des Vertauschens der Ausgänge eines Entscheidungsalgorithmus nicht anwenden. Als Beispiel kann die Sprache $L_{\text{SAT}} = \{F \mid F \text{ ist ein Boolescher Ausdruck, und } F \text{ ist erfüllbar}\}$ genommen werden. Das Komplement von L_{SAT} ist die Sprache

$\overline{L_{\text{SAT}}} = \{F \mid F \text{ ist ein Boolescher Ausdruck, und } F \text{ ist nicht erfüllbar}\}$; als Grundmenge wurde hier die Menge der syntaktisch korrekten Booleschen Ausdrücke genommen. Der in Kapitel 5.3 beschriebene polynomiell zeitbeschränkten nichtdeterministischen Algorithmus zur Akzeptanz von L_{SAT} lautet (in Pseudocode-Notation):

```
TYPE Boolescher_Ausdruck_typ = ...;
           { beschreibt den Typ eines Booleschen Ausdrucks }
     Variablen_typ = ...;
           { beschreibt den Typ einer Variablen           }
     Entscheidungs_typ = ...;
           { beschreibt den Typ der ja/nein-Entscheidung  }

PROCEDURE Erfuellbarkeit (F               : Boolescher_Ausdruck_typ;
                          VAR Entscheidung: Entscheidungs_typ);

  VAR V : SET OF Variablen_typ;

BEGIN { Erfuellbarkeit }
  V := Menge der in F vorkommenden Variablen;
  wähle für jede Variable in V einen Wert aus {TRUE, FALSE};
  setze die Belegung in F ein und werte F aus;
  IF F = TRUE THEN Entscheidung := ja
              ELSE Entscheidung := nein;
END   { Erfuellbarkeit };
```

Vertauschen der ja/nein-Ausgänge des Algorithmus liefert den nichtdeterministischen Algorithmus

```
PROCEDURE A (F               : Boolescher_Ausdruck_typ;
             VAR Entscheidung: Entscheidungs_typ);

  VAR V : SET OF Variablen_typ;

BEGIN { A }
  V := Menge der in F vorkommenden Variablen;
  wähle für jede Variable in V einen Wert aus {TRUE, FALSE};
  setze die Belegung in F ein und werte F aus;
  IF F = TRUE THEN Entscheidung := nein
              ELSE Entscheidung := ja;
END   { A };
```

Für einen Booleschen Ausdruck F ist A(F) = ja genau dann, wenn es eine Belegung der Variablen in F gibt, die bei Auswertung von F auf den Wert FALSE führt, d.h. wenn es eine Belegung der Variablen von F gibt, die F nicht erfüllt. Beispielsweise führt mit $F = (x_1 \wedge \neg x_2)$ der Aufruf von A auf den ja-Ausgang (etwa mit der Belegung $x_1 = \mathsf{FALSE}$ und $x_2 = \mathsf{FALSE}$), aber $F \notin \overline{L_{\mathrm{SAT}}}$, denn F ist mit der Belegung $x_1 = \mathsf{TRUE}$ und $x_2 = \mathsf{FALSE}$ erfüllbar.

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 $\mathbf{NPC} \cap \mathbf{P} = \varnothing$.

Es gilt folgender Satz[12]:

Satz 5.6-3:

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, $A \leq_m^p B$, aber nicht $B \leq_m^p A$ gelten.

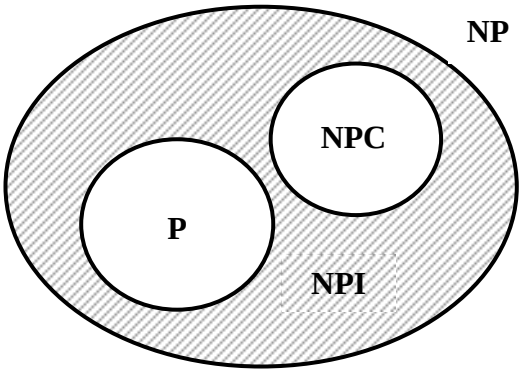
Satz 5.6-3 lässt sich folgendermaßen anwenden:

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:

12 Ladner, R.E.: On the structure of polynomial time reducibility, J.ACM, 22, 155-171, 1975.

Satz 5.6-4:
Unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ ist $\mathbf{NP} \setminus (\mathbf{P} \cup \mathbf{NPC}) \neq \varnothing$.

Bezeichnet man $\mathbf{NP} \setminus (\mathbf{P} \cup \mathbf{NPC})$ als die Menge **NPI** der **NP-unvollständigen Sprachen**, so zeigt **NP** (unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$) 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 $(v,w) \in E_1 \Leftrightarrow (f(v), f(w)) \in E_2$.

Beispielsweise sind die folgenden beiden Graphen isomorph:

Die folgende Sprachklasse besteht aus Sprachen, deren Komplemente in **NP** liegen:

$$\textbf{co - NP} = \left\{ \Sigma^* \setminus L \mid L \text{ ist eine Sprache über } \Sigma \text{ und } L \in \textbf{NP} \right\}.$$

Aufgrund der Abgeschlossenheit der Klasse **P** bezüglich Komplementbildung ist mit einer Sprache $L \in \textbf{P}$, $L \subseteq \Sigma^*$, auch ihr Komplement $\Sigma^* \setminus L \in \textbf{P}$. Daher gilt:

Satz 5.6-5:
$\textbf{P} \subseteq \textbf{NP} \cap \textbf{co - NP}$.

Für $n \in \mathbf{N}$ sei $I_n = \left\{ 0, ..., 2^n - 1 \right\}$ die Menge natürlichen Zahlen, die mit *n* Bits darstellbar sind. Die Abbildung $f : I_n \to I_n$ sei bijektiv und besitze die Eigenschaften

(i) Jeder Wert $f(x)$ ist mit polynomiellem Zeitaufwand (gemessen in $size(x)$) deterministisch berechenbar; hierbei wird *x* in Binärdarstellung in den entsprechenden Algorithmus eingegeben, der dann die Binärdarstellung von $f(x)$ erzeugt.

(ii) $f^{-1}(y)$ kann nicht mit polynomiellem Zeitaufwand berechnet werden.

Dann liegt die Menge $L = \left\{ bin(x)\#bin(y) \;\middle|\; x \in I_n, y \in I_n \text{ und } f^{-1}(y) < x \right\}$ in $\textbf{NP} \cap \textbf{co - NP} \setminus \textbf{P}$.

Zum Beweis dieser Aussage sind drei Eigenschaften zu zeigen:

1. $L \in \textbf{NP}$: Dazu ist ein polynomiell zeitbeschränkter nichtdeterministischer Algorithmus anzugeben, der bei Eingabe eines Worts $w = bin(x)\#bin(y)$ mit $x \in I_n$ und $y \in I_n$ genau dann die `ja`-Entscheidung trifft, wenn $f^{-1}(y) < x$ ist. Dieser Algorithmus arbeitet wie folgt:
   Bei Eingabe von $w = bin(x)\#bin(y)$ wird eine 0-1-Zeichenkette der Länge $\le n$ geraten. Diese Zeichenkette kann als Binärdarstellung einer Zahl $z \in I_n$ angesehen werden. Nun wird deterministisch in polynomieller Zeit der Wert $f(z)$ berechnet. Die Eingabe *w* wird genau dann akzeptiert, wenn $bin(f(z)) = bin(y)$ und $z < x$ gilt (die zweite Eigenschaft kann an den Binärdarstellungen von *z* und *x* überprüft werden).
2. $L \in \textbf{co-NP}$: Hierzu ist ein polynomiell zeitbeschränkter nichtdeterministischer Algorithmus anzugeben, der bei Eingabe eines Worts genau dann die `ja`-Entscheidung trifft, wenn das Wort entweder nicht die syntaktische Form $w = bin(x)\#bin(y)$ mit

$x \in I_n$ und $y \in I_n$ besitzt oder $f^{-1}(y) \geq x$ ist. Die syntaktische Überprüfung erfolgt in der üblichen Weise. Die Bedingung $f^{-1}(y) \geq x$ kann mit einer Modifikation des Algorithmus aus 1. erfolgen: Die Eingabe $w = bin(x)\#bin(y)$ wird jetzt jedoch dann akzeptiert, wenn $bin(f(z)) = bin(y)$ und $z \geq x$ gilt.

3. $L \notin \mathbf{P}$: Angenommen, L liegt in **P**. Dann gibt es einen polynomiell zeitbeschränkten deterministischen Entscheidungsalgorithmus $\mathbf{A}_L$ für L. Mit Hilfe von $\mathbf{A}_L$ kann man zu $y \in I_n$ in polynomieller Zeit deterministisch mit Binärsuche den Wert $f^{-1}(y)$ berechnen; dieses widerspricht der Voraussetzung in (ii). Dazu wird für $y \in I_n$ der Wert $1\underbrace{0\ldots0}_{(n-1)\text{-mal}}\#bin(y)$ in $\mathbf{A}_L$ eingegeben; die Zeichenkette $1\underbrace{0\ldots0}_{(n-1)\text{-mal}}$ entspricht der Zahl 2^{n-1}. Antwortet $\mathbf{A}_L$ mit `ja` (d.h. es ist $f^{-1}(y) < 2^{n-1}$), dann wird $1\underbrace{0\ldots0}_{(n-2)\text{-mal}}\#bin(y)$ in $\mathbf{A}_L$ eingegeben, d.h. es wird geprüft, ob $f^{-1}(y) < 2^{n-2}$ gilt; antwortet $\mathbf{A}_L$ mit `nein` (d.h. es ist $f^{-1}(y) \geq 2^{n-1}$), dann wird $11\underbrace{0\ldots0}_{(n-2)\text{-mal}}\#bin(y)$ in $\mathbf{A}_L$ eingegeben, d.h. es wird geprüft, ob $2^{n-1} \leq f^{-1}(y) < 3 \cdot 2^{n-2}$ gilt. Die Binärsuche wird so lange fortgesetzt, bis der exakte Wert von $f^{-1}(y)$ gefunden ist. Dazu sind insgesamt höchstens n Aufrufe von $\mathbf{A}_L$ erforderlich.

Die Existenz einer Funktion mit den oben beschriebenen Eigenschaften hat also zur Folge, dass die Inklusion in Satz 5.6-5 echt ist. Dieses ist jedoch eine bisher nichtgeklärte Frage.

Man hat für viele Probleme in **co - NP** jedoch nicht nachweisen können, dass sie in **NP** liegen. 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 5.6-6:
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 $L_1 \in \mathbf{co} - \mathbf{NP}$ nachweisbar ist:

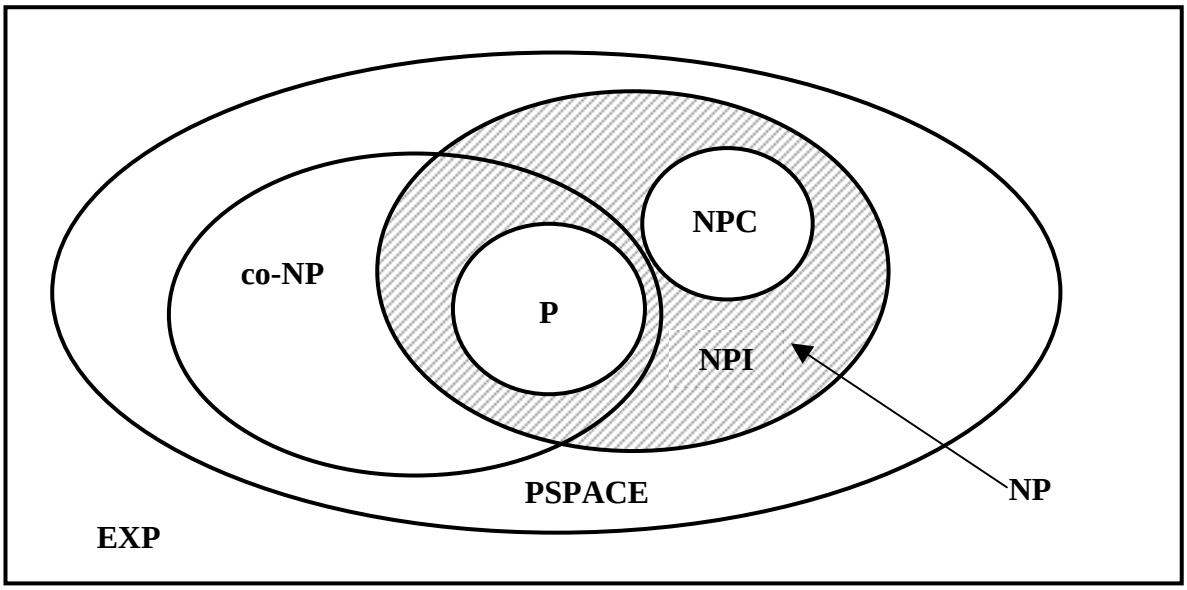
Wegen $L \in \mathbf{NPC}$ ist $L_1 \leq_m^p L$. Die dabei verwendete in polynomieller Zeit berechenbare totale Funktion sei f, d.h. es gilt $f : \Sigma_1^* \to \Sigma^*$ mit $w \in L_1 \Leftrightarrow f(w) \in L$. Der Übergang auf die Komplemente ergibt $w \in \Sigma_1^* \setminus L_1 \Leftrightarrow f(w) \in \Sigma^* \setminus L$.
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 $\mathbf{NP} \neq \mathbf{co}\text{-}\mathbf{NP}$ 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. Er wurde zuvor nichtdeterministisch in polyomieller Zeit erzeugt.

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 $\mathbf{V}(x,B_x) = \texttt{ja}$.

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 deterministischen 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 in polynomieller Zeit (gemessen in $|x|$) ermittelt.

Offensichtlich ist $\mathbf{PO} \subseteq \mathbf{NPO}$.

Satz 6-2:
Ist $\mathbf{P} \neq \mathbf{NP}$, dann ist $\mathbf{PO} \neq \mathbf{NPO}$.

Beweis:

Es sei $\mathbf{PO} = \mathbf{NPO}$. Dann folgt $\mathbf{P} = \mathbf{NP}$:

Ist Π aus **NPO** ein Optimierungsproblem, dessen zugehöriges Entscheidungsproblem Π_{Ent} **NP**-vollständig ist. In den folgenden Unterkapiteln werden eine Reihe derartiger Probleme behandelt. Mit Satz 5.1-1 folgt, dass Π_{Ent} in **P** liegt, und mit Satz 5.4-4 folgt $\mathbf{P} = \mathbf{NP}$.

///

Im Laufe dieses Kapitels wird die Struktur der Klasse **NPO** genauer untersucht. Alle im folgenden behandelten Optimierungsprobleme liegen 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.

6.1 Absolut approximierbare Probleme

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 $k \geq 0$ gibt mit $D(x, \mathbf{A}(x)) = \left| m_\Pi^*(x) - m_\Pi(x, \mathbf{A}(x)) \right| \leq k$. Das Optimierungsproblem Π heißt in diesem Fall **absolut approximierbar**. Die Klasse der absolut approximierbaren Probleme in **NPO** wird mit **ABS** bezeichnet.

Das folgende Beispiel zeigt, dass $\mathbf{ABS} \neq \varnothing$ ist.

Das Färbbarkeits-Minimierungsproblem der Knoten in einem ungerichteten Graphen

Instanz: 1. $I = G = (V, E)$
$G = (V, E)$ ist ein ungerichteter Graph mit Knotenmenge $V = \{v_1, \ldots, v_n\}$ und Kantenmenge $E \subseteq V \times V$

2. Eine Abbildung $c_V : V \to \mathbf{N}$ heißt **zulässige Knotenfärbung** von *G*, wenn für jede ungerichtete Kante $(v, w) \in E$ die Bedingung $c_V(v) \neq c_V(w)$ gilt.
$\mathrm{SOL}(I) = \{ c_V \mid c_V \text{ ist eine zulässige Knotenfärbung von } G \}$

3. $m(I, c_V) = |c_V(V)|$ für $c_V \in \mathrm{SOL}(I)$

4. $goal = \min$

Es wird eine zulässige Färbung der Knoten mit einer minimalen Anzahl an Farben gesucht. Man kann die Farben in einer zulässigen Färbung mit den natürlichen Zahlen 1, ..., $|c_V(V)|$ i-

dentifizieren. Es lässt sich leicht zeigen, dass dieses Problem in **NPO** liegt; in Satz 5.5-5 wird direkt gezeigt, dass das zugehörige Entscheidungsproblem **NP**-vollständig ist.

Der **Grad eines Knotens** *v* in einem Graphen *G* ist die Anzahl der Knoten, die mit *v* über eine Kante verbunden sind. Der maximal auftretende Grad eines Knotens in einem Graphen *G* ist der **Grad des Graphen** und wird mit $\Delta(G)$ bezeichnet.

Der folgende Greedy-Algorithmus liefert bei Eingabe eines ungerichteten Graphen eine zulässige Knotenfärbung:

Algorithmus COL zur Knotenfärbung eines ungerichteten Graphen:

Eingabe: $I = G = (V, E)$
$G = (V, E)$ ist ein ungerichteter Graph mit Knotenmenge $V = \{v_1, \ldots, v_n\}$ und Kantenmenge $E \subseteq V \times V$

Verfahren: Der Knoten v_1 erhält die Farbe 1, d.h. $c_V(v_1) = 1$.
Sind $v_1, \ldots, v_i$ bereits mit den Farben 1, ..., *k* gefärbt, so wird v_{i+1} mit einer der Farben 1, ..., *k* gefärbt, falls dabei eine zulässige Knotenfärbung entsteht (gibt es mehrere Möglichkeiten einer Farbwahl aus $\{1, \ldots, k\}$, dann wird die kleinste Farbnummer genommen). Ist eine zulässige Knotenfärbung mit Farben aus $\{1, \ldots, k\}$ nicht möglich, d.h. es gibt mindesten *k* bereits gefärbte Knoten, die mit v_{i+1} über eine Kante verbunden sind, dann wird $c_V(v_{i+1}) = k + 1$ gesetzt.

Ausgabe: $\mathbf{COL}(I) = \{c_V(v_1), \ldots, c_V(v_n)\}$ = Menge der verwendeten Farben.

Satz 6.1-1:

(i) Es sei $G = (V, E)$ ein ungerichteter Graph. Der Algorithmus **COL** zur Knotenfärbung berechnet in polynomieller Zeit eine zulässige Knotenfärbung mit höchstens $\Delta(G) + 1$ Farben.

(ii) Es sei $G = (V, E)$ ist ein ungerichteter Graph mit mindestens einer Kante. Dann gilt für die Anzahl $m(G, \mathbf{COL}(G))$ der vom Algorithmus **COL** ermittelten Farben zur Knotenfärbung von *G*:
$|m^*(G) - m(G, \mathbf{COL}(G))| \leq \Delta(G) - 1$.

Beweis:

Zu (i): $G = (V, E)$ ein ungerichteter mit der Knotenmenge $V = \{v_1, \ldots, v_n\}$.

Für $i = 1, ..., n$ sei k_i die Anzahl an Farben, die **COL** in seinem Ablauf zur Färbung der Knoten $v_1, ..., v_i$ verwendet. Für einen Knoten v_j sei $d_i(v_j)$ die Anzahl der bereits gefärbten Knoten, die nach Färbung von v_i mit v_j über eine Kante verbunden sind. Durch Induktion über *i* lässt sich die Beziehung

$k_i \le 1 + \max\{ d_i(v_j) \mid 1 \le j \le i \}$

zeigen:

Es ist $k_1 = 1$ und $d_1(v_1) = 0$.

Es gelte $k_{i-1} \le 1 + \max\{ d_{i-1}(v_j) \mid 1 \le j \le i-1 \}$. Mit G_{i-1} werde der durch $\{ v_1, ..., v_{i-1} \}$ induzierte Teilgraph von *G* bezeichnet. Alle Knoten in G_{i-1} sind (zulässig) gefärbt. Entsprechend bezeichne G_i den durch $\{ v_1, ..., v_i \}$ induzierte Teilgraph von *G*. Es wird im *i*-ten Schritt v_i gefärbt. Für alle Knoten v_j in G_{i-1}, die mit v_i in G_i verbunden sind, gilt $d_i(v_j) = d_{i-1}(v_j) + 1$. Bei allen anderen Knoten v_j ändert sich beim Übergang von G_{i-1} zu G_i nichts. Daher gilt $d_{i-1}(v_j) \le d_i(v_j)$. Wird zur Färbung von v_i keine neue Farbe benötigt, so ist mit der Induktionsvoraussetzung

$k_i = k_{i-1} \le 1 + \max\{ d_{i-1}(v_j) \mid 1 \le j \le i-1 \} \le 1 + \max\{ d_i(v_j) \mid 1 \le j \le i-1 \}$. Da keine neue Farbe benötigt wird, ist $d_i(v_i) < k_{i-1}$ bzw. $d_i(v_i) + 1 \le k_{i-1}$ und damit

$k_i = k_{i-1} \le 1 + \max\{ d_i(v_j) \mid 1 \le j \le i \}$.

Wird zur Färbung von v_i eine neue Farbe benötigt, so ist $k_i = k_{i-1} + 1$ und $d_i(v_i) \ge k_{i-1}$ bzw. $d_i(v_i) + 1 \ge k_{i-1} + 1 = k_i$. Damit ergibt sich ebenfalls

$k_i \le 1 + \max\{ d_i(v_j) \mid 1 \le j \le i \}$.

Für $i = n$ folgt $m(G, \mathbf{COL}(G)) = k_n \le 1 + \max\{ d_n(v_j) \mid 1 \le j \le n \} = 1 + \Delta(G)$.

Zu (ii): Da *G* mindestens eine Kante besitzt, ist $m^*(G) \ge 2$. Damit ergibt sich

$$\left| m^*(G) - m(G, \mathbf{COL}(G)) \right| = m(G, \mathbf{COL}(G)) - m^*(G) \le \Delta(G) + 1 - 2 = \Delta(G) - 1.$$

///

Bei Eingabe des folgenden Graphen *G* ermittelt der Algorithmus **COL** eine zulässige Kantenfärbung mit $\Delta(G) + 1$ Farben:

$G = (V, E)$ mit der Knotenmenge $V = \{ v_1, ..., v_{2 \cdot m} \}$ mit $m \ge 1$. Zwischen Knoten mit geradem Index gibt es keine Kanten; ebenso gibt es keine Kanten zwischen Knoten mit ungeradem Index. Es gibt jeweils eine Kante zwischen dem Knoten $v_{2 \cdot i}$ und allen Knoten mit ungeradem Index, außer mit dem Knoten $v_{2 \cdot i - 1}$. Es ist $\Delta(G) = m - 1$. Offensichtlich ist $m^*(G) = 2$: Die Knoten mit ungeradem Index erhalten die Farbe 1, diejenigen mit geradem Index die Farbe 2.

Man kann sich leicht davon überzeugen, dass der obige Algorithmus die Knoten mit ungeradem Index nacheinander mit den Farben 1, ..., *m* färbt, ebenso die Knoten mit geradem Index, d.h. $m(G,\mathbf{COL}(G)) = \Delta(G)+1$ und $\left|m^*(G) - m(G,\mathbf{COL}(G))\right| = \Delta(G)-1$. Die in Satz 6.1-1 angegebene absolute Abweichung ist also (für unendlich viele Graphen) nicht zu verbessern.

Weiterhin liefert Satz 6.1.-1 keine absolute Approximation, da $\Delta[G]-1$ keine Konstante ist. Für eine spezielle Klasse von Graphen, nämlich für planare Graphen, gibt es jedoch eine absolute polynomiell zeitbeschränkte Approximation:

Ein Graph heißt **planar**, wenn man ihn kantenüberschneidungsfrei in die Ebene einbetten kann. Es gilt:

Satz 6.1-2:

(i) Jeder ungerichtete planare Graph $G = (V,E)$ ist in polynomieller Zeit mit 6 Farben färbbar.

(ii) Es gibt einen polynomiell zeitbeschränkten Algorithmus **A**, der bei Anwendung auf einen ungerichteten planaren Graphen $G = (V,E)$ eine Knotenfärbung mit $\left|m^*(G) - m(G,\mathbf{A}(G))\right| \le 3$ und $m(G,\mathbf{A}(G))/m^*(G) \le 2$ ermittelt.

Beweis:

Zu (i): Der planare Graph $G = (V,E)$ habe *n* Knoten und *m* Kanten. Die Anzahl seiner durch Kantenzüge eingeschlossenen Flächen, einschließlich der den Graphen umgebenden Fläche, sei *f*. Man kann annehmen, dass *G* zusammenhängend ist; ansonsten bezieht man die folgenden Überlegungen auf jede Zusammenhangskomponente. Der Beweis erfolgt durch den Nachweis mehrerer Hilfssätze:

1. Es gilt $n - m + f = 2$.

Beweis: Für $m = 0$ ist $f = 1$ und wegen des Zusammenhangs von *G* auch $n = 1$.
Es sei $m \ge 1$, und die Behauptung gelte für alle planaren zusammenhängenden Graphen mit höchstens $m-1$ Kanten.
Es sei $(u,v) \in E$ eine Kante, die keine Brücke ist (eine Brücke ist eine Kante, deren Entfernung die Anzahl der Zusammenhangskomponenten erhöht). Dann ist $G' = (V,E')$ mit $E' = E \setminus \{(u,v)\}$ zusammenhängend. Außerdem ist $|E'| = m-1$, und die Anzahl der Flächen von E' ist $f-1$. Nach In-

duktionsvoraussetzung gilt $n-(m-1)+(f-1)=2$, also $n-m+f=2$. Besteht G nur aus Brücken, dann ist G ein Baum, und es ist $m=n-1$ und $f=1$, also $n-(n-1)+1=2$.

2. Die Länge eines kürzesten Kreises in G sei g. Für einen kreisfreien Graphen sei $g=\infty$. Dann ist $m \le g/(g-2)\cdot(n-2)$.

Beweis: Jede innere Fläche in G hat mindestens g begrenzende Kanten. Jede innere Kante begrenzt genau 2 Flächen. Daher ist $(f-1)\cdot g \le 2\cdot m$. Damit ergibt sich $f\cdot g=(f-1)\cdot g+g \le (f-1)\cdot g \le 2\cdot m$. Mit 1. folgt
$2\cdot m \ge f\cdot g = 2\cdot g - g\cdot n + g\cdot m$, also
$(g-2)\cdot m \le (n-2)\cdot g$ bzw. $m \le g/(g-2)\cdot(n-2)$.

3. Es gilt $m \le 3\cdot n-6$.

Beweis: Wegen $g \ge 3$ ist $g/(g-2) \le 3$. Mit 2. folgt die Behauptung.

4. Es gibt mindestens einen Knoten in G mit Grad ≤ 5.

Beweis: Angenommen, alle Knoten in G hätten einen Grad ≥ 6. Bezeichnet $d(v)$ den Grad des Knotens v, dann ist $\sum_{v\in V} d(v) = 2\cdot m \ge 6\cdot n$ bzw. $m \ge 3\cdot n$ im Widerspruch zu 3.

Im Algorithmus **COL** ist die Reihenfolge, in der die Knoten gefärbt werden, nicht spezifiziert; sie richtet sich nach der willkürlichen Numerierung der Knoten. Es wird nun „rückwärts“ eine Reihenfolge $(v_1,...,v_n)$ der Knoten festgelegt: v_n sei ein Knoten mit minimalem Grad in G. Sind die Knoten $v_{i+1},...,v_n$ bereits bestimmt, ist v_i ein Knoten mit minimalem Grad im Teilgraph, der durch $V \setminus \{v_{i+1},...,v_n\}$ induziert wird. Es wird der Algorithmus **COL** mit der so definierten Reihenfolge der Färbung von $\{v_1,...,v_n\}$ angewendet. Der Knoten v_n hat wegen 4. einen Grad ≤ 5. Der durch $V \setminus \{v_{i+1},...,v_n\}$ induzierte Teilgraph ist ebenfalls planar, und v_i hat wieder wegen 4. einen Grad ≤ 5. Daher sind höchstens 6 Farben ausreichend.

Zu (ii): Mit Hilfe des folgenden Verfahrens kann man in einer Laufzeit der Ordnung $O(n\cdot m)$, d.h. in polynomieller Zeit, testen, ob die Knoten des Graphen $G=(V,E)$ mit der Knotenmenge $V=\{v_1,...,v_n\}$ und m Kanten mit zwei Farben, etwa mit den Nummern 0 und 1, zulässig färbbar sind:

Der Knoten v_1 erhält die Farbe 0. Ist *v* ein Knoten in *V*, der mit mindestens einem bereits gefärbten Knoten, etwa mit der Farbe mit Nummer *f*, über eine Kante verbunden ist, so wird geprüft, ob alle bereits gefärbten Knoten, mit denen *v* über eine Kante verbunden ist, dieselbe Farbe *f* haben. In diesem Fall erhält *v* die Farbe $1-f$. Andernfalls bricht das Verfahren ab, und *G* wird als nicht 2-färbbar entschieden. Sind schließlich alle Knoten in *V* gefärbt, so ist die Knotenmenge von *G* zulässig mit zwei Farben färbbar.

Der Algorithmus **A** hat bei Eingabe eines planaren Graphen *G* folgenden Ablauf:

Ist die Knotenmenge von *G* zulässig mit 2 Farben färbbar, so wird diese Färbung ausgegeben. Andernfalls ermittelt **A** eine zulässige Färbung der Knotenmenge mit höchstens 6 Farben.

Dann gilt:

Im ersten Fall ist $m^*(G) = 2 = m(G, \mathbf{A}(G))$ und daher $\left|m^*(G) - m(G, \mathbf{A}(G))\right| = 0$ und $m(G, \mathbf{A}(G)) \big/ m^*(G) = 1$. Im zweiten Fall ist $m^*(G) \geq 3$ und $m(G, \mathbf{A}(G)) = 6$, also $\left|m^*(G) - m(G, \mathbf{A}(G))\right| = m(G, \mathbf{A}(G)) - m^*(G) \leq 3$ und $m(G, \mathbf{A}(G)) \big/ m^*(G) \leq 2$.

///

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

2. $\mathrm{SOL}(I) = \left\{ (x_1, ..., x_n) \;\middle|\; x_i = 0 \text{ oder } x_i = 1 \text{ für } i = 1, ..., n \text{ und } \sum_{i=1}^{n} x_i \cdot w_i \leq M \right\}$

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) $m\left(I, \left(x_1^*, ..., x_n^*\right)\right) = \sum_{i=1}^{n} x_i^* \cdot p_i$ ist maximal unter allen möglichen Auswahlen $x_1, ..., x_n$, die (1) und (2) erfüllen.

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 6.1-3:

Es sei $k > 0$ 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 $D(I, \mathbf{A}(I)) = \left| m^*(I) - m(I, \mathbf{A}(I)) \right| \le k$ gilt.

Bei $\mathbf{P} \neq \mathbf{NP}$ ist also $\mathbf{PO} \subset \mathbf{ABS} \subset \mathbf{NPO}$.

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 5.4-4).

Bei Annahme der Existenz von **A** kann ein Algorithmus $\tilde{\mathbf{A}}$ definiert werden, der folgendermaßen 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 $\sum_{i=1}^{n} x_i \cdot w_i \le M$ und

$$\left| m^*(\tilde{I}) - m(\tilde{I}, \mathbf{A}(\tilde{I})) \right| = \left| m^*(\tilde{I}) - \sum_{i=1}^{n} x_i \cdot (k+1) \cdot p_i \right| \le k \qquad \text{(Ungleichung (*)).}$$

$\tilde{\mathbf{A}}$ 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) $m(I, \tilde{\mathbf{A}}(I)) = \frac{m(\tilde{I}, \mathbf{A}(\tilde{I}))}{k+1}$ 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 $\le$-Zeichens in der Ungleichung (*) ausklammern und erhält

$$\left| m^*(\tilde{I}) - m(\tilde{I}, \mathbf{A}(\tilde{I})) \right| = \left| m^*(\tilde{I}) - \sum_{i=1}^{n} x_i \cdot (k+1) \cdot p_i \right| = (k+1) \cdot R \le k$$

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 $m(I, \tilde{\mathbf{A}}(I)) \ge m(I, y)$ für jedes $y \in \mathrm{SOL}(I)$ zu zeigen):

$$m(I, \tilde{\mathbf{A}}(I)) = \frac{m(\tilde{I}, \mathbf{A}(\tilde{I}))}{k+1} = \frac{m^*(\tilde{I})}{k+1} \ge \frac{m(\tilde{I}, y)}{k+1} = m(I, y) \text{ für jedes } y \in \mathrm{SOL}(I).$$

///

6.2 Relativ approximierbare Probleme

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, der eine zulässige Lösung $\mathbf{A}(x)$ ermittelt. Ist $x \in \Sigma_\Pi^*$ eine Instanz von Π, so gilt trivialerweise $m(x, \mathbf{A}(x)) \le m_\Pi^*(x)$ bei einem Maximierungsproblem bzw. $m_\Pi^*(x) \le m(x, \mathbf{A}(x))$ bei einem Minimierungsproblem.

Die **relative Approximationsgüte** $R_{\mathbf{A}}(x)$ von **A** bei Eingabe einer Instanz $x \in \Sigma_{\Pi}^{*}$ wird definiert durch

$R_{\mathbf{A}}(x) = \dfrac{m_{\Pi}^{*}(x)}{m(x, \mathbf{A}(x))}$ bei einem Maximierungsproblem

bzw.

$R_{\mathbf{A}}(x) = \dfrac{m(x, \mathbf{A}(x))}{m_{\Pi}^{*}(x)}$ 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.

Es sei $r \geq 1$. Der Approximationsalgorithmus **A** für das Optimierungsproblem Π aus **NPO** heißt ***r*-approximativer Algorithmus**, wenn $R_{\mathbf{A}}(x) \leq r$ für jede Instanz $x \in \Sigma_{\Pi}^{*}$ gilt.

Bemerkung: Es sei **A** ein Approximationsalgorithmus für das Minimierungsproblem Π, und es gelte $m(x, \mathbf{A}(x)) \leq r \cdot m_{\Pi}^{*}(x) + k$ 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 6.3).

Die **Klasse APX** besteht aus denjenigen Optimierungsproblemen aus **NPO**, für die es einen r-approximativen Algorithmus für ein $r \geq 1$ gibt.

Offensichtlich ist $\mathbf{APX} \subseteq \mathbf{NPO}$.

Die relative Approximationsgüte liefert eine Abschätzung des Werts einer approximativen Lösung im Vergleich zum Optimum, die unmittelbar aus der Definition folgt:

Satz 6.2-1:

Für die relative Approximationsgüte $R_{\mathbf{A}}(x)$ eines Approximationsalgorithmus **A** bei Eingabe einer Instanz $x \in \Sigma_{\Pi}^*$ gilt:

(i) $1 \le R_{\mathbf{A}}(x)$. Je dichter $R_{\mathbf{A}}(x)$ bei 1 liegt, um so besser ist die Approximation.

(ii) Ist Π ein Maximierungsproblem, so bedeutet die Aussage „$R_{\mathbf{A}}(x) \le r$“ mit einer Konstanten $r \ge 1$ für alle Instanzen $x \in \Sigma_{\Pi}^*$:
$\frac{1}{r} \cdot m_{\Pi}^*(x) \le m(x, \mathbf{A}(x)) \le m_{\Pi}^*(x)$ und $m(x, \mathbf{A}(x)) \le m_{\Pi}^*(x) \le r \cdot m(x, \mathbf{A}(x))$, insbesondere
$\frac{1}{R_{\mathbf{A}}(x)} \cdot m_{\Pi}^*(x) \le m(x, \mathbf{A}(x)) \le m_{\Pi}^*(x)$ und $m(x, \mathbf{A}(x)) \le m_{\Pi}^*(x) \le R_{\mathbf{A}}(x) \cdot m(x, \mathbf{A}(x))$.

(iii) Ist Π ein Minimierungsproblem, so bedeutet die Aussage „$R_{\mathbf{A}}(x) \le r$“ mit einer Konstanten $c \ge 1$ für alle Instanzen $x \in \Sigma_{\Pi}^*$:
$m_{\Pi}^*(x) \le m(x, \mathbf{A}(x)) \le r \cdot m_{\Pi}^*(x)$ und $\frac{1}{r} \cdot m(x, \mathbf{A}(x)) \le m_{\Pi}^*(x) \le m(x, \mathbf{A}(x))$, insbesondere
$m_{\Pi}^*(x) \le m(x, \mathbf{A}(x)) \le R_{\mathbf{A}}(x) \cdot m_{\Pi}^*(x)$ und $\frac{1}{R_{\mathbf{A}}(x)} \cdot m(x, \mathbf{A}(x)) \le m_{\Pi}^*(x) \le m(x, \mathbf{A}(x))$.

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 $0 < a_i \le 1$ für $i = 1, \ldots, n$

2. $\mathrm{SOL}(I) = \left\{ [B_1, \ldots, B_k] \;\middle|\; \begin{array}{l} [B_1, \ldots, B_k] \text{ ist eine Partition (disjunkte Zerlegung)} \\ \text{von } I \text{ mit } \sum_{a_i \in B_j} a_i \le 1 \text{ für } j = 1, \ldots, k \end{array} \right\}$, 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.

In Kapitel 5.3 wird das Binpacking-Entscheidungsproblem beschrieben. Dort sind die Objekte natürliche Zahlen; die Behältergröße *b* ist eine natürliche Zahl, die eventuell größer als 1 ist. Normiert man dort die Objekte und die Behältergröße durch Division durch *b*, so erhält man die hier vorliegende Formulierung des Binpacking-Problems. Sind umgekehrt die Objekte rationale Zahlen der Form $a_i = p_i/q_i$ für $i = 1, ..., n$, so kann man das Binpacking-Problem durch Wahl der Behältergröße $b = kgV(q_1, ..., q_n)$ und die Normierung $a_i \cdot b$ auf die Formulierung in Kapitel 5.3 bringen.

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 $i = 1, ..., n$

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. $\sum_{a_l \in B_j} a_l \leq 1$ 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 6.2-2:
Ist $I = [a_1, \ldots, a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $R_{\mathbf{NF}}(I) \leq 2$, d.h. der Nextfit-Algorithmus ist 2-approximativ ($m(I, \mathbf{NF}(I)) \leq 2 \cdot m^*(I)$).
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

$$k < 2 \cdot \sum_{j=1}^{k} u_j + 1 - (u_1 + u_k) = 2 \cdot \sum_{i=1}^{n} a_i + 1 - (u_1 + u_k) \le 2 \cdot \sum_{i=1}^{n} a_i + 1 \text{ und } k \le 2 \cdot \sum_{i=1}^{n} a_i .$$

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: $I = [a_1, ..., a_{2m}]$. 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 $a_{\max} = \max\{a_i \mid i = 1, ..., n\}$ 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 $i = 1, \ldots, n$

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 $B_1, \ldots, B_j$ nicht gibt, lege a_{i+1} in einen neuen (bisher leeren) Behälter B_{j+1}.

Ausgabe: $\mathbf{FF}(I) =$ die benötigten nichtleeren Behälter $[B_1, \ldots, B_k]$.

Satz 6.2-3:
Ist $I = [a_1, \ldots, a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $m(I,\mathbf{FF}(I)) \le 1{,}7\cdot m^*(I) + 2$.

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 6.3) 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 $i = 1, \ldots, n$

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 $B_1, \ldots, B_j$ mit dem kleinsten Index, in den das Objekt a_{i+1} noch passt, ohne dass er überläuft. Falls es einen derartigen Behälter unter $B_1, \ldots, B_j$ nicht gibt, lege a_{i+1} in einen neuen (bisher leeren) Behälter B_{j+1}.

Ausgabe: $\mathbf{FFD}(I) =$ die benötigten nichtleeren Behälter $[B_1, \ldots, B_k]$.

Satz 6.2-4:
Ist $I = [a_1, \ldots, a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $m(I,\mathbf{FFD}(I)) \leq 1{,}5 \cdot m^*(I) + 1$.

Beweis:

Die Objekte der Eingabeinstanz $I = [a_1, \ldots, a_n]$ seien nach absteigender Größe sortiert, d.h. $a_1 \geq \ldots \geq a_n$. 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; $t_C \leq 1$.

$m^*(\tilde{I}) = t_A + t_{BC} + t_{CC} + t_B + t_C$ 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 $b + c \leq 1$. 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 $m^*(I) \leq m(I, \mathbf{FFD}(I)) = m(\tilde{I}, \mathbf{FFD}(\tilde{I})) = m^*(\tilde{I}) \leq m^*(I)$; 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 6.3).

Eine genauere Analyse des FirstfitDecreasing-Algorithmus zeigt, dass die in Satz 6.2-4 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
$m(I, \mathbf{FFD}(I)) \leq 11/9 \cdot m^*(I) + 4$ 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 $1/2 + \delta$ mit $0 < \delta < 1/40$, *m* Objekte der Größe $1/4 + 2\delta$, *m* Objekte der Größe $1/4 + \delta$ und $2m$ Objekte der Größe $1/4 - 2\delta$. 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, ..., *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 $B_1, ..., B_j$, der den kleinsten freien Platz aufweist. Falls a_{i+1} in keinen der Behälter $B_1, ..., B_j$ passt, lege a_{i+1} in einen neuen (bisher leeren) Behälter B_{j+1}.

Ausgabe: $\mathbf{BFD}(I) =$ die benötigten nichtleeren Behälter $[B_1, ..., B_k]$.

Satz 6.2-5:
Ist $I = [a_1, \ldots, a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $m(I, \mathbf{BFD}(I)) \le 11/9 \cdot m^*(I) + 4$.

Auch der BestfitDecreasing-Algorithmus nur asymptotisch $11/9$-approximativ (siehe Kapitel 6.3). 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 Satz 6.2-10) 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 6.2-4) $m(I, \mathbf{FFD}(I)) \le 11/9 \cdot m^*(I) + 4$; $\quad 11/9 = 1{,}222$
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 Satz 6.1-3 wird 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

2. $\mathrm{SOL}(I) = \left\{ (x_1, ..., x_n) \mid x_i = 0 \text{ oder } x_i = 1 \text{ für } i = 1, ..., n \text{ und } \sum_{i=1}^{n} x_i \cdot w_i \leq M \right\}$; 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 $(x_1, ..., x_n) \in \mathrm{SOL}(I)$

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) $\sum_{i=1}^{n} x_i^* \cdot w_i \leq M$

(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.

Der Approximationsalgorithmus **RUCK_APP** wird hier informell beschrieben:

1. Bei Eingabe einer Instanz $I = (A, M)$ sortiert 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 vergleicht das Ergebnis im Schritt 1 mit der Rucksackfüllung, die man erhält, wenn man nur das Objekt mit dem Gewinn $p_{\max}$ al-

lein in den Rucksack legt. Man nimmt diejenige Rucksackfüllung, die den größeren Wert der Zielfunktion liefert.

Das Ergebnis des Verfahrens ist eine 0-1-Folge $\mathbf{RUCK\_APP}(I) = (x_1, ..., x_n)$ mit dem Wert $m(I, \mathbf{RUCK\_APP}(I))$ der Zielfunktion.

Satz 6.2-6:

Für jede Instanz $I = (A, M)$ des 0/1-Rucksack-Maximierungsproblems gilt: $1 \le m^*(I) / m(I, \mathbf{RUCK\_APP}(I)) \le 2$, 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 $x_j = (M - \overline{w}) / w_j$. Der Rucksack ist dann gefüllt, d.h. im 1. Teil von **RUCK_APP** wird für die modifizierte Aufgabenstellung $x_{j+1} = ... = x_n = 0$ gesetzt. Es lässt sich zeigen, dass jetzt bereits eine optimale Lösung des modifizierten 0/1-Rucksack-Maximierungsproblems gefunden ist, und zwar mit einem Profit $m_{MOD} = \overline{p} + ((M - \overline{w}) / w_j) \cdot p_j$. Da jede zulässige Lösung des (ursprünglichen) 0/1-Rucksack-Maximierungsproblems eine zulässige Lösung des modifizierten 0/1-Rucksack-Maximierungsproblems ist, folgt $m^*(I) \le \overline{p} + ((M - \overline{w}) / w_j) \cdot p_j < \overline{p} + p_j$ (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 $p_{\max} \geq p_j > \overline{p}$ und $m^*(I) < \overline{p} + p_j < 2 \cdot p_{\max} \leq 2 \cdot m\big(I, \textbf{RUCK\_APP}(I)\big)$.

In beiden Fällen folgt die Abschätzung $1 \leq m^*(I) \big/ m\big(I, \textbf{RUCK\_APP}(I)\big) \leq 2$.

///

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 $m\big(I, \textbf{RUCK\_APP}(I)\big) = M/2 + 2$. 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

$$m^*(I) \Big/ m\big(I, \textbf{RUCK\_APP}(I)\big) = \frac{M}{M/2+2} = \frac{2M}{M+4} = \frac{2 \cdot (M+4) - 8}{M+4} = 2 - \frac{8}{M+4}.$$

Bei genügend großem *M* kommt dieser Wert beliebig dicht an 2 heran, so dass die Abschätzung in Satz 6.2-5 nicht verbessert werden kann.

Der folgende Satz zeigt, dass es unter der Voraussetzung $\textbf{P} \neq \textbf{NP}$ nicht für jedes Optimierungsproblem aus **NPO** einen relativ 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. $\text{SOL}(G) = \left\{ T \mid T = \left\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \right\rangle \text{ ist eine Tour durch } G \right\}$

3. für $T \in \text{SOL}(G)$, $T = \left\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), ..., (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \right\rangle$, ist die Zielfunktion definiert durch $m(G, T) = \sum_{j=1}^{n-1} w\left(v_{i_j}, v_{i_{j+1}}\right) + w\left(v_{i_n}, v_{i_1}\right)$

4. $goal = \min$

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

Satz 6.2-7:

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 6.1-3.

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 $\mathbf{P} = \mathbf{NP}$.

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 $(v_{\pi(1)}, v_{\pi(2)}, \ldots, v_{\pi(n)})$ der Knoten mittels einer Permutation π der Knotenindizes, so dass für $i = 1, \ldots, n-1$ gilt: $(v_{\pi(i)}, v_{\pi(i+1)}) \in E$ und $(v_{\pi(n)}, v_{\pi(1)}) \in E$.

Es sei **A** ein *r*-approximativen Algorithmus **A** für das Handlungsreisenden-Minimierungsproblem, d.h. für alle Instanzen *I* dieses Problems gilt $m(I, \mathbf{A}(I)) \leq r \cdot m^*(I)$. Man kann $r > 1$ annehmen, denn für $r = 1$ ermittelt **A** bereits eine optimale Lösung. Es wird ein Algorithmus $\widetilde{\mathbf{A}}$ für UNGERICHTETER HAMILTONKREIS konstruiert, der folgendendermaßen arbeitet:

Bei Eingabe eines ungerichteten Graphen $G = (V, E)$ mit $V = \{v_1, \ldots, v_n\}$ konstruiert $\widetilde{\mathbf{A}}$ einen vollständigen bewerteten ungerichteten Graphen $\widetilde{G} = (V, \widetilde{E})$ (die Knotenmenge wird beibehalten) mit $\widetilde{E} = \{(v_i, v_j) \mid v_i \in V \text{ und } v_j \in V\}$ und der Kantenbewertung $w : \widetilde{E} \to \mathbf{R}_{\geq 0}$, 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 $\widetilde{G} = (V, \widetilde{E})$ wird in den Algorithmus **A** eingegeben und das Ergebnis $m(\widetilde{G}, \mathbf{A}(\widetilde{G}))$ mit dem Wert $r \cdot n$ verglichen. $\widetilde{\mathbf{A}}$ trifft die Entscheidung

$$\widetilde{\mathbf{A}}(G) = \begin{cases} \texttt{ja} & \text{für } m(\widetilde{G}, \mathbf{A}(\widetilde{G})) \le r \cdot n \\ \texttt{nein} & \text{für } m(\widetilde{G}, \mathbf{A}(\widetilde{G})) > r \cdot n \end{cases} .$$

Falls **A** ein polynomiell zeitbeschränkter Algorithmus ist, dann auch $\widetilde{\mathbf{A}}$.

Besitzt *G* einen Hamiltonkreis (mit Länge *n*), dann ist $m^*(\widetilde{G}) = n$, und $\widetilde{\mathbf{A}}(G) = \texttt{ja}$, da **A** ein *r*-approximativen Algorithmus **A** für das Handlungsreisenden-Minimierungsproblem ist und damit $m(\widetilde{G}, \mathbf{A}(\widetilde{G})) \le r \cdot m^*(\widetilde{G}) = r \cdot n$ ist.

Besitzt *G* keinen Hamiltonkreis, dann enthält jeder Hamiltonkreis in $\widetilde{G}$ mindestens eine Kante, die mit $r \cdot n$ bewertet ist (da $\widetilde{G}$ ein vollständiger Graph ist, enthält $\widetilde{G}$ einen Hamiltonkreis). Damit ergibt sich $m(\widetilde{G}, \mathbf{A}(\widetilde{G})) \ge m^*(\widetilde{G}) \ge n - 1 + r \cdot n > r \cdot n$, d.h. $\widetilde{\mathbf{A}}(G) = \texttt{nein}$.

In beiden Fällen trifft $\widetilde{\mathbf{A}}$ 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 $w(v_i, v_j) \le w(v_i, v_k) + w(v_k, v_j)$.

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 $1 \le t < r$ gibt. Der folgende Satz 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 6.2-8:

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 $\varepsilon > 0$ mit folgender Eigenschaft:

Für jedes $x \in \Sigma'^*$ ist $m^*(f(x)) = \begin{cases} c(x) & \text{für } x \in L_{\Pi'} \\ c(x) \cdot (1+\varepsilon) & \text{für } x \notin L_{\Pi'} \end{cases}$.

Dann gibt es keinen polynomiell zeitbeschränkten *r*-approximativen Algorithmus für Π mit $r < 1 + \varepsilon$, außer $\mathbf{P} = \mathbf{NP}$.

Beweis:

Angenommen, es gibt einen polynomiell zeitbeschränkten *r*-approximativen Algorithmus **A** für Π mit $r < 1 + \varepsilon$. 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 $c(x) \cdot (1+\varepsilon)$ verglichen. $\mathbf{A}'$ trifft die Entscheidung

$$\mathbf{A}'(x) = \begin{cases} \texttt{ja} & \text{für } m(f(x), \mathbf{A}(f(x))) < c(x) \cdot (1+\varepsilon) \\ \texttt{nein} & \text{für } m(f(x), \mathbf{A}(f(x))) \ge c(x) \cdot (1+\varepsilon) \end{cases}.$$

$\mathbf{A}'$ 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 $\frac{m(f(x), \mathbf{A}(f(x)))}{m^*(f(x))} \le r < 1 + \varepsilon$. Wegen $x \in L_{\Pi'}$ ist $m^*(f(x)) = c(x)$. Also ist $m(f(x), \mathbf{A}(f(x))) < m^*(f(x)) \cdot (1+\varepsilon) = c(x) \cdot (1+\varepsilon)$, und $\mathbf{A}'(x) = \texttt{ja}$.

Es sei $x \notin L_{\Pi'}$. Dann folgt $m^*(f(x)) = c(x) \cdot (1+\varepsilon)$,
$m(f(x), \mathbf{A}(f(x))) \geq m^*(f(x)) = c(x) \cdot (1+\varepsilon)$ und $\mathbf{A'}(x) = \texttt{nein}$.

In beiden Fällen trifft $\mathbf{A}'$ die korrekte Entscheidung.

///

Bemerkung: Der Beweis von Satz 6.2-8 zeigt, dass Satz 6.2-8 gültig bleibt, wenn die dort formulierte Voraussetzung über $m^*(f(x))$ ersetzt durch die Voraussetzung

$$\text{Für jedes } x \in \Sigma'^* \text{ ist } m^*(f(x)) \begin{cases} = c(x) & \text{für } x \in L_{\Pi'} \\ \geq c(x) \cdot (1+\varepsilon) & \text{für } x \notin L_{\Pi'} \end{cases}.$$

Ein entsprechender Satz gilt für Maximierungsprobleme:

Satz 6.2-9:

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 $\varepsilon > 0$ mit folgender Eigenschaft:

$$\text{Für jedes } x \in \Sigma'^* \text{ ist } m^*(f(x)) = \begin{cases} c(x) & \text{für } x \in L_{\Pi'} \\ c(x) \cdot (1-\varepsilon) & \text{für } x \notin L_{\Pi'} \end{cases}.$$

Dann gibt es keinen polynomiell zeitbeschränkten *r*-approximativen Algorithmus für Π mit $r < 1/(1-\varepsilon)$, 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 6.2-10:

Ist $\mathbf{P} \neq \mathbf{NP}$, dann gibt es keinen *r*-approximativen Algorithmus für das Binpacking-Minimierungsproblem mit $r \leq 3/2 - \varepsilon$ für $\varepsilon > 0$.

Beweis:

In Satz 6.2-8 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 6.2-8 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 $I = [a_1, \ldots, a_n]$ des Partitionenproblems sei $B = \sum_{a_i \in I} a_i$. Es wird $f(I) = [(2a_1)/B, \ldots, (2a_n)/B]$ definiert, falls sich dadurch eine Instanz des Binpacking-Minimierungsproblems ergibt, d.h. falls $(2a_i)/B \le 1$ für $i = 1, \ldots, n$ 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 $\{a_1, \ldots, a_n\}$ in zwei disjunkte Teile I_1 und I_2 mit $\sum_{a_i \in I_1} a_i = \sum_{a_i \in I_2} a_i$, dann gilt $a_i \le B/2$ für $i = 1, \ldots, n$, 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 $f(I) = [(2a_1)/B, \ldots, (2a_n)/B]$ des Binpacking-Minimierungsproblems sind genau 2 Behälter erforderlich, d.h. $m^*(f(I)) = 2 = c(I)$. .

Ist $I \notin L_{\Pi'}$ und $\left[(2a_1)/B, ..., (2a_n)/B\right]$ 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 $\left[(2a_1)/B, ..., (2a_n)/B\right]$ eine Instanz des Binpacking-Minimierungsproblems, dann ist $f(I) = \left[(2a_1)/B, ..., (2a_n)/B\right]$ und $m^*(f(I)) \geq 3$.
In beiden Fällen gilt $m^*(f(I)) \geq 3 = 2 \cdot \frac{3}{2} = c(I) \cdot \left(1 + \frac{1}{2}\right)$.

Aus Satz 6.2-8 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 **P** = **NP**.

///

Ein weiteres Beispiel für ein „interessantes“ Problem in **APX** ist das

3-SAT-Maximierungsproblem

Instanz:
1. $I = (K, V)$
   $K = \{F_1, F_2, ..., F_m\}$ 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. $y_{i_j} = \neg x_l$)
2. SOL(I) = Belegung der Variablen in *V* mit Wahrheitswerten TRUE oder FALSE, d.h. SOL(*I*) ist eine Abbildung $f : V \to \{\mathsf{TRUE}, \mathsf{FALSE}\}$
3. Für $f \in \mathrm{SOL}(I)$ ist $m(I, f)$ = Anzahl der Klauseln in *K*, die durch *f* erfüllt werden

goal = max

Belegt man bei Vorgabe einer Instanz $I = (K, V)$ für das 3-SAT-Maximierungsproblem alle in *V* vorkommenden Variablen mit TRUE und erhält dabei m_1 viele erfüllte Klauseln und belegt anschließend alle Variablen in *V* mit FALSE, wobei m_2 viele Klauseln erfüllt werden, und nimmt dann diejenige der beiden Belegungen, die mehr Klauseln erfüllt, so gilt $\max\{m_1, m_2\} \geq \frac{1}{2} \cdot m$. Denn wenn unter der Belegung aller Variablen in *V* mit TRUE eine Klausel nicht erfüllt wird, hat sie die Form $(\neg x_{i_1} \vee \neg x_{i_2} \vee \neg x_{i_3})$. Daher wird diese Klausel bei der zweiten Belegung der Variablen mit FALSE erfüllt. Es ist also $m_2 \geq m - m_1$ bzw.

$m_1 + m_2 \geq m$. Daher können nicht beide Werte m_1 und m_2 $< \frac{1}{2} \cdot m$ sein. Wegen $m^*(I) \leq m$ erhält man also einen Approximationsalgorithmus **A** mit $R_{\mathbf{A}}(I) = \frac{m^*(I)}{m(I, \mathbf{A}(I))} \leq \frac{m}{1/2 \cdot m} = 2$, indem man diejenige der beiden Belegungen nimmt, die mehr Klauseln erfüllt. Daher liegt das 3-SAT-Maximierungsproblem in **APX**.

Man kann sogar zeigen, dass es für dieses Problem, wenn es auf Instanzen eingeschränkt wird, in denen jede Klausel in den Literalen genau 3 verschiedene Variablen enthält, einen $8/7$-approximativen polynomiell zeitbeschränkten Algorithmus gibt, und unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ kein polynomiell zeitbeschränkter Algorithmus eine bessere relative Approximation liefert.

Der $8/7$-approximativen polynomiell zeitbeschränkten Algorithmus soll hier informell beschrieben werden, weil er eine spezielle Technik, die Derandomisierung eines randomisierten Algorithmus einsetzt.

Eine Belegung der Variablen $x_1, ..., x_n$ mit Wahrheitswerten TRUE bzw. FALSE werde zufällig erzeugt, wobei $P(x_i = \mathsf{TRUE}) = P(x_i = \mathsf{FALSE}) = 1/2$ für $i = 1, ..., n$ gelte. Diese Belegung induziert eine zufällige Belegung der drei verschiedenen Literale in der Klausel F_j für $j = 1, ..., m$. Von den 8 möglichen Belegungen der Literale in F_j mit Wahrheitswerten, ergibt genau eine Belegung den Wahrheitswert FALSE, die 7 übrigen den Wert TRUE. Das bedeutet $P(F_j \text{ hat den Wahrheitswert } \mathsf{TRUE}) = 7/8$.

Die Zufallsvariable X_j für $j = 1, ..., m$ werde definiert durch

$$X_j = \begin{cases} 1 & F_j \text{ hat den Wert TRUE unter einer zufälligen Belegung der Literale} \\ 0 & F_j \text{ hat den Wert FALSE unter einer zufälligen Belegung der Literale} \end{cases}.$$

Weiterhin sei $X = X_1 + ... + X_m$.

Dann ist $\mathbf{E}[X_j] = P(X_j = \mathsf{TRUE}) = 7/8$ und $\mathbf{E}[X] = 7/8 \cdot m$. Mit $m^*(I) \leq m$ ergibt sich für die Approximationsgüte einer zufälligen Belegung der Variablen mit Wahrheitswerten ein Wert $\leq 8/7$.

Aus dieser Überlegung heraus lässt sich ein Algorithmus entwerfen, der nacheinander alle Variablen $x_1, ..., x_n$ in dieser aufsteigenden Reihenfolge mit Wahrheitswerten belegt.

Es gilt

$$\begin{aligned}\mathbf{E}[X] &= P(x_1 = \mathsf{TRUE}) \cdot \mathbf{E}[X \mid x_1 = \mathsf{TRUE}] + P(x_1 = \mathsf{FALSE}) \cdot \mathbf{E}[X \mid x_1 = \mathsf{FALSE}] \\ &= 1/2 \cdot (\mathbf{E}[X \mid x_1 = \mathsf{TRUE}] + \mathbf{E}[X \mid x_1 = \mathsf{FALSE}]) \\ &= 1/2 \cdot \left(\sum_{j=1}^{m} \mathbf{E}[X_j \mid x_1 = \mathsf{TRUE}] + \sum_{j=1}^{m} \mathbf{E}[X_j \mid x_1 = \mathsf{FALSE}] \right).\end{aligned}$$

Die Werte $\mathbf{E}[X_j \mid x_1 = \mathsf{TRUE}]$ und $\mathbf{E}[X_j \mid x_1 = \mathsf{FALSE}]$ lassen sich leicht ermitteln:

$\mathbf{E}[X_j \mid x_1 = \mathsf{TRUE}] = 1$, wenn F_j durch die Belegung $x_1 = \mathsf{TRUE}$ erfüllt wird;

$\mathbf{E}[X_j \mid x_1 = \mathsf{TRUE}] = 7/8$, wenn F_j durch die Belegung $x_1 = \mathsf{TRUE}$ nicht erfüllt wird. Entsprechendes gilt für $\mathbf{E}[X_j \mid x_1 = \mathsf{FALSE}]$.

Es liegt daher nahe, x_1 genau dann mit TRUE zu belegen, wenn

$$\mathbf{E}[X \mid x_1 = \mathsf{TRUE}] = \sum_{j=1}^{m} \mathbf{E}[X_j \mid x_1 = \mathsf{TRUE}] \geq \sum_{j=1}^{m} \mathbf{E}[X_j \mid x_1 = \mathsf{FALSE}] = \mathbf{E}[X \mid x_1 = \mathsf{FALSE}] \text{ ist.}$$

Den Variablen $x_1, ..., x_{i-1}$ seien bereits Wahrheitswerte zugewiesen. Dann ist

$$\mathbf{E}[X_j \mid x_1, ..., x_{i-1}, x_i = \mathsf{TRUE}] = \begin{cases} 1 & \text{falls die Belegung von } x_1, ..., x_i \text{ die Klausel } F_j \text{ erfüllt} \\ & \text{andernfalls :} \\ 7/8 & \begin{cases} \text{falls kein Literal in } F_j \text{ bisher mit einem Wahrheitswert} \\ \text{belegt wurde} \end{cases} \\ 3/4 & \begin{cases} \text{falls genau ein Literal in } F_j \text{ den Wahrheitswert FALSE hat} \\ \text{und den anderen beiden Literalen noch kein Wahrheitswert} \\ \text{zugewiesen wurde} \end{cases} \\ 1/2 & \begin{cases} \text{falls genau zwei Literale in } F_j \text{ den Wahrheitswert FALSE} \\ \text{haben und dem dritten Literal noch kein Wahrheitswert} \\ \text{zugewiesen wurde} \end{cases} \\ 0 & \begin{cases} \text{falls alle Literale in } F_j \text{ den Wahrheitswert} \\ \text{FALSE haben} \end{cases} \end{cases}$$

Entsprechende Werte gelten für $\mathbf{E}[X_j \mid x_1, ..., x_{i-1}, x_i = \mathsf{FALSE}]$. Damit lassen sich

$$\mathbf{E}[X \mid x_1, ..., x_{i-1}, x_i = \mathsf{TRUE}] = \sum_{j=1}^{m} \mathbf{E}[X_j \mid x_1, ..., x_{i-1}, x_i = \mathsf{TRUE}],$$

$$\mathbf{E}[X \mid x_1, ..., x_{i-1}, x_i = \mathsf{FALSE}] = \sum_{j=1}^{m} \mathbf{E}[X_j \mid x_1, ..., x_{i-1}, x_i = \mathsf{FALSE}] \text{ und}$$

$$\mathbf{E}[X \mid x_1, ..., x_{i-1}, x_i] = 1/2 \cdot (\mathbf{E}[X \mid x_1, ..., x_{i-1}, x_i = \mathsf{TRUE}] + \mathbf{E}[X \mid x_1, ..., x_{i-1}, x_i = \mathsf{FALSE}])$$

ermitteln.

Der folgende Algorithmus **A** (in Pseudocode) ermittelt eine Belegung der Variablen mit $\frac{m^*(I)}{m(I, \mathbf{A}(I))} \leq \frac{m}{(7/8 \cdot m)} = \frac{8}{7}$ und hat eine polynomielle Laufzeit der Ordnung $O(n \cdot m)$.

Approximationsalgorithmus für das 3-SAT-Maximierungsproblem

Eingabe: $I = (K,V)$

$K = \{F_1, F_2, ..., F_m\}$ 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 sind in jeder Klausel in den Literalen genau 3 verschiedene Variablen.

Verfahren:

```
VAR W0 : REAL;
    W1 : REAL;
    i : INTEGER;

BEGIN
  FOR i := 1 TO n DO
   BEGIN
     W0 := E[X | x1, ..., xi-1, xi = FALSE];
     W1 := E[X | x1, ..., xi-1, xi = TRUE];
     IF W0 <= W1 THEN xi := TRUE
                 ELSE xi := FALSE;
   END;
```

Ausgabe: $x_1, ..., x_n$.

Abschließend soll ein Zusammenhang zwischen den Klassen **ABS** und **APX** hergestellt werden:

Satz 6.2-11:

Es sei Π ein Optimierungsproblem, für dessen Zielfunktion $m_\Pi(x,y) \geq 1$ mit $x \in \Sigma_\Pi^*$ und $y \in \mathrm{SOL}_\Pi(x)$ ist. Dann gilt:

Liegt Π in **ABS**, dann auch in **APX**.

Beweis:

Es sei Π in **ABS** und **A** ein Approximationsalgorithmus mit $\left|m^*_\Pi(x) - m_\Pi(x, \mathbf{A}(x))\right| \le k$ für ein $k \ge 0$ und jedes $x \in \Sigma^*_\Pi$ und jedes $y \in \mathrm{SOL}_\Pi(x)$.
Ist Π ein Maximierungsproblem, dann folgt aus
$m^*_\Pi(x) - m_\Pi(x, \mathbf{A}(x)) = \left|m^*_\Pi(x) - m_\Pi(x, \mathbf{A}(x))\right| \le k$:

$$R_{\mathbf{A}}(x) = \frac{m^*_\Pi(x)}{m(x, \mathbf{A}(x))} \le \frac{k}{m(x, \mathbf{A}(x))} + 1 \le k + 1.$$

Ist Π ein Minimierungsproblem, dann folgt aus
$m_\Pi(x, \mathbf{A}(x)) - m^*_\Pi(x) = \left|m^*_\Pi(x) - m_\Pi(x, \mathbf{A}(x))\right| \le k$:

$$R_{\mathbf{A}}(x) = \frac{m(x, \mathbf{A}(x))}{m^*_\Pi(x)} \le \frac{k}{m^*_\Pi(x)} + 1 \le k + 1.$$

In beiden Fällen ist Π in **APX**.

///

6.3 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 Beispiel.

Partitionen-Minimierungsproblem

Instanz:
1. $I = [a_1, \ldots, a_n]$
   $a_1, \ldots, a_n$ („Objekte“) sind natürliche Zahlen mit $a_i > 0$ für $i = 1, ..., n$
2. $\mathrm{SOL}(I) = \{[Y_1, Y_2] \mid [Y_1, Y_2]$ 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 $i = 1, ..., n$

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;
END;
```

Ausgabe: $[Y_1, Y_2]$.

Satz 6.3-1:
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 $size(I)$ 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. $k+1 \geq (2-r+r-1)/(r-1) = 1/(r-1)$.

Die Anzahl an Zeichen, um die Eingabeinstanz darzustellen, die Problemgröße $size(I)$, ist durch $2 \cdot n \cdot \lceil \log(A+1) \rceil$ mit $A = \max\{ a_i \mid i = 1,...,n \}$ beschränkt.

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 $O(size(I))$. Da $k(r) = \lceil (2-r)/(r-1) \rceil \leq (2-r)/(r-1) + 1 = 1/r - 1$ ist, d.h. $k(r) \in O(1/(r-1))$, ist das Laufzeitverhalten bei festem *r* polynomiell in der Größe der Eingabeinstanz, jedoch exponentiell in $O(size(I))$ und $1/(r-1)$.

///

Offensichtlich ist $\mathbf{PTAS} \subseteq \mathbf{APX}$. In Satz 6.2-10 wird gezeigt, dass es unter der Annahme $\mathbf{P} \neq \mathbf{NP}$ keinen *r*-approximativen Algorithmus für das Binpacking-Minimierungsproblem mit $r \leq 3/2 - \varepsilon$ für $\varepsilon > 0$ gibt. Daraus folgt:

Satz 6.3-2:
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* und einer rationalen Zahl $r > 1$ eine *r*-approximative Lösung für *x* in einer Laufzeit liefert, die polynomiell von $size(x)$ und $1/(r-1)$ abhängt. Einen derartigen Algorithmus nennt man **voll polynomiell zeitbeschränktes Approximationsschema** (fully polynomial-time approximation scheme). Die Optimierungsprobleme aus **NPO**, die einen derartigen Algorithmus zulassen, bilden die Klasse **FPTAS**. Zu beachten ist hier, dass die Laufzeit polynomiell abhängig von $1/(r-1)$ und nicht von der erforderlichen Anzahl an Bits zur Darstellung von *r*, d.h. von $\log(1/(r-1))$, ist, da dieses eine zu strenge Einschränkung der Klasse **FPTAS** zur Folge hätte.

Der folgende Satz zeigt, dass das ganzzahlige 0/1-Rucksackproblem als Maximierungsproblem in **FPTAS** liegt**.** Daher ist die Klasse **FPTAS** nicht leer. Dazu muss ein voll polynomiell zeitbeschränktes Approximationsschema für dieses Problem angegeben werden.

Voll polynomiell zeitbeschränktes Approximationsschema für das ganzzahlige 0/1-Rucksack-Maximierungsproblem

Eingabe: $I = (A, M)$, rationale Zahl $r > 1$

$A = \{a_1, ..., a_n\}$ ist eine Menge von *n* Objekten und $M \in \mathbf{N}_{>0}$ die „Rucksackkapazität". Jedes Objekt a_i, $i = 1, ..., n$, hat die Form $a_i = (w_i, p_i)$; hierbei bezeichnet $w_i \in \mathbf{N}_{>0}$ das Gewicht und $p_i \in \mathbf{N}_{>0}$ den Wert (Profit) des Objekts a_i. Es sei $p_{max} = \max\{p_1, ..., p_n\} = p_{i_{max}}$.

Verfahren: `VAR k : REAL;`

`BEGIN`

`k :=` $\lfloor (r-1)\cdot p_{max}/n \rfloor$`;`

`FOR` i `:= 1 TO` n `DO`

p'_i `:=` $\left\lfloor p_i / \mathrm{k} \right\rfloor$`;`

berechne eine optimale Rucksackfüllung für die Eingabeinstanz $I' = (A', M)$ mit $A' = \{a'_1, ..., a'_n\}$ und $a'_i = (w_i, p'_i)$ für $i = 1, ..., n$; die zugehörige Entscheidungsfolge sei $(x'^*_1, ..., x'^*_n)$;

`END;`

`IF` $\sum_{i=1}^{n} x'^*_i \cdot p_i \geq p_{max}$ `THEN` Ausgabe $(x'^*_1, ..., x'^*_n)$ und $\sum_{i=1}^{n} x'^*_i \cdot p_i$

`ELSE` Ausgabe $\underbrace{(0, ..., 0, 1, 0, ..., 0)}_{1 \text{ an Position } i_{max}}$ und p_{max} .

Satz 6.3-3:
Das ganzzahlige 0/1-Rucksack-Maximierungsproblem liegt in **FPTAS**.

Beweis:

Zu zeigen ist, dass der obige Algorithmus **A** ein voll polynomiell zeitbeschränktes Approximationsschema für das ganzzahlige 0/1-Rucksack-Maximierungsproblem ist, d.h. bei Eingabe von I und r polynomiell laufzeitbeschränkt in $size(I)$ und $1/(r-1)$ ist und dass $R_{\mathbf{A}}(I) = \dfrac{m^*_{\Pi}(I)}{m(I, \mathbf{A}(I))} \leq r$ gilt.

Die Folge $(x'^*_1, ..., x'^*_n)$ ist eine optimale Entscheidungsfolge für $I' = (A', M)$ mit $A' = \{a'_1, ..., a'_n\}$ und $a'_i = (w_i, p'_i)$ für $i = 1, ..., n$, d.h. $m^*_{\Pi}(I') = \sum_{i=1}^{n} x'^*_i \cdot \left\lfloor p_i / k \right\rfloor$ und $\sum_{i=1}^{n} x'^*_i \cdot w_i \leq M$.

Es sei $(x^*_1, ..., x^*_n)$ eine optimale Entscheidungsfolge für die Ausgangsinstanz $I = (A, M)$, d.h. $m^*_{\Pi}(I) = \sum_{i=1}^{n} x^*_i \cdot p_i$ und $\sum_{i=1}^{n} x^*_i \cdot w_i \leq M$. Die letzte Ungleichung zeigt, dass $(x^*_1, ..., x^*_n)$ eine zulässige Lösung auch für die Instanz $I' = (A', M)$ ist. Daher ist

$$m^*_{\Pi}(I') = \sum_{i=1}^{n} x'^*_i \cdot \left\lfloor p_i / k \right\rfloor \geq \sum_{i=1}^{n} x^*_i \cdot \left\lfloor p_i / k \right\rfloor .$$

$$m(I,\mathbf{A}(I)) = \sum_{i=1}^{n} x_i'^{*} \cdot p_i \geq \sum_{i=1}^{n} x_i'^{*} \cdot k \cdot \left\lfloor p_i / k \right\rfloor = k \cdot \sum_{i=1}^{n} x_i'^{*} \cdot \left\lfloor p_i / k \right\rfloor = k \cdot m_{\Pi}^{*}(I')$$

$$\geq k \cdot \sum_{i=1}^{n} x_i^{*} \cdot \left\lfloor p_i / k \right\rfloor \geq k \cdot \sum_{i=1}^{n} x_i^{*} \cdot \left(p_i / k - 1 \right)$$

$$= \sum_{i=1}^{n} x_i^{*} \cdot (p_i - k) = m_{\Pi}^{*}(I) - k \cdot \sum_{i=1}^{n} x_i^{*} \geq m_{\Pi}^{*}(I) - k \cdot n \ ,$$

also $m_{\Pi}^{*}(I) \leq m(I,\mathbf{A}(I)) + k \cdot n$. Daraus folgt

$$R_{\mathbf{A}}(I) = \frac{m_{\Pi}^{*}(I)}{m(I,\mathbf{A}(I))} \leq 1 + \frac{k \cdot n}{m(I,\mathbf{A}(I))} \leq 1 + \frac{k \cdot n}{p_{max}} = 1 + \frac{\lfloor (r-1) \cdot p_{max}/n \rfloor \cdot n}{p_{max}}$$

$$\leq 1 + \frac{((r-1) \cdot p_{max}/n) \cdot n}{p_{max}} = r \ .$$

Die Ermittlung einer optimalen Rucksackfüllung für die Eingabeinstanz $I' = (A', M)$ mit $A' = \{a_1', ..., a_n'\}$ und $a_i' = (w_i, p_i')$ für $i = 1, ..., n$ kann beispielsweise mit Hilfe eines Algorithmus erfolgen, der nach dem Prinzip der Dynamischen Programmierung arbeitet[13]:

Bestimmung einer optimalen Lösung für das ganzzahlige 0/1-Rucksack-Maximierungsproblem

Eingabe: $I' = (A', M)$
$A' = \{a_1', ..., a_n'\}$ ist eine Menge von n Objekten und $M \in \mathbf{N}_{>0}$ die „Rucksackkapazität". Jedes Objekt a_i, $i = 1, ..., n$, hat die Form $a_i' = (w_i, p_i')$; hierbei bezeichnet $w_i \in \mathbf{N}_{>0}$ das Gewicht und $p_i' \in \mathbf{N}_{>0}$ den Wert (Profit) des Objekts a_i.

Verfahren: Für $j = 1, ..., n$ und $y \leq M$ bezeichne $f_j(y)$ den Wert der Zielfunktion einer optimalen Lösung für die Eingabeinstanz (A_j', y) mit $A_j' = \{a_1', ..., a_j'\}$; die zugehörige Entscheidungsfolge sei $(x_1'^{*}, ..., x_j'^{*})$. Gesucht werden $(x_1'^{*}, ..., x_n'^{*}) \in \mathrm{SOL}(I')$ der Wert $m^{*}(I') = f_n(M)$. Es gelten folgende Rekursionsgleichungen:

$$f_0(y) = 0 \text{ für } 0 \leq y \leq M \text{ und } f_0(y) = -\infty \text{ für } y < 0 \ ,$$
$$f_j(y) = \max\{f_{j-1}(y),\ f_{j-1}(y - w_j) + p_j'\} \text{ für } j = 1, ..., n, 0 \leq y \leq M \ .$$

[13] Hoffmann, U.: **Datenstrukturen und Algorithmen**, FINAL 15:2, 2005 (Kap. 7.3.3).

Die Sprungstellen dieser Treppenfunktionen bestimmen eindeutig den Funktionsverlauf und werden mitttels Dynamischer Programmierung berechnet. Es sei S_j für $j = 0, ..., n$ die Menge der Sprungstellen von f_j.

Zur Erzeugung der Funktionen $f_0, ..., f_n$ bzw. der Mengen ihrer Sprungstellen $S_0, ..., S_n$ und der Ermittlung einer optimalen Lösung werden im folgenden folgende Begriffe verwendet:

Ein Zahlenpaar (W,P) *dominiert* ein Zahlenpaar (W',P'), wenn $W \le W'$ und $P \ge P'$ gilt.

Für zwei Mengen *A* und *B* von Zahlenpaaren sei $A \otimes B$ die Vereinigungsmenge von A und B ohne die Paare aus *A*, die durch ein Paar aus B dominiert werden, und ohne die Paare aus *B*, die durch ein Paar aus *A* dominiert werden.

```
VAR j : INTEGER;

{ Berechnung der Funktionen f_j bzw. der Mengen S_j ihrer
  Sprungstellen für j = 1,...,n :                          }

BEGIN
S_0 := {(0,0)};
FOR j := 1 TO n DO
   BEGIN
      Ŝ_j := {(W + w_j, P + p'_j) | (W,P) ∈ S_{j-1}};
      S_j := S_{j-1} ⊗ Ŝ_j ;
   END;
END;
```

{ Die Paare in S_j seien aufsteigend nach der ersten (und damit auch nach der zweiten) Komponente geordnet, d.h.
$S_j = \{(W_{0,j}, P_{0,j}), ..., (W_{i_j,j}, P_{i_j,j})\}$ mit $W_{k,j} \le W_{k+1,j}$ für $0 \le k < i_j$.
Für $y \ge 0$ gelte $W_{k,j} \le y < W_{k+1,j}$.
Dann ist $f_j(y) = P_{k,j}$. }

{ Berechnung einer optimierende Auswahl
$x_1'^*, ..., x_n'^*$: }

Es sei (W,P) dasjenige Paar in S_n mit $f_n(M) = P$.

```
FOR j := n DOWNTO 1 DO
   BEGIN
     IF  (W, P) ∈ S_{j-1}
     THEN  x'_j* := 0
     ELSE BEGIN
             x'_j* := 1 ;
             W := W − w_j ;
             P := P − p'_j ;
          END;
   END;
```

Ausgabe: $\left(x_1'^*, ..., x_n'^*\right)$ und Wert $\sum_{i=1}^{n} x_i'^* \cdot p_i'$ der Zielfunktion.

Die Laufzeit des Verfahrens bei Eingabe von $I' = (A', M)$ mit $A' = \{a_1', ..., a_n'\}$ und $a_i' = (w_i, p_i')$ für $i = 1, ..., n$ ist von der Ordnung $O\left(n \cdot \sum_{i=1}^{n} p_i'\right)$. Es ist

$$n \cdot \sum_{i=1}^{n} p_i' = n \cdot \sum_{i=1}^{n} \left\lfloor p_i / k \right\rfloor \le n \cdot \sum_{i=1}^{n} p_i / k \le n^2 \,.\, p_{max} / \left\lfloor (r-1) \cdot p_{max} / n \right\rfloor$$

$$\le n^2 \,.\, p_{max} / \left((r-1) \cdot p_{max} / n - 1\right)$$

Dieser Ausdruck ist von der Ordnung $O\left(n^3 \cdot \frac{1}{r-1}\right)$. Die Größe $size(I)$ ist proportional zur Anzahl der Bits zu ihrer Darstellung, d.h. $size(I)$ ist von der Ordnung $O(n \cdot \log(A))$ mit $A = \max\{\{p_{max}\} \cup \{w_1, ..., w_n\}\}$. Daher ist die Laufzeit ein polynomieller Wert in $size(I)$ und $1/(r-1)$.

///

Es lässt sich zeigen, dass es unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ Optimierungsprobleme in **PTAS** gibt, die nicht in **FPTAS** liegen. Dazu bedarf es einer weiteren Definition.

Ein Optimierungsproblem Π in **NPO** heißt **polynomiell beschränkt**, wenn es ein Polynom p gibt, so dass für jede Instanz $x \in \Sigma_\Pi^*$ und jede zulässige Lösung $y \in \mathrm{SOL}_\Pi(x)$ die Abschätzung $m_\Pi(x, y) \le p(|x|)$ gilt.

Satz 6.3-4:

Ist $\mathbf{P} \neq \mathbf{NP}$, so liegt kein polynomiell beschränktes Optimierungsproblem aus **NPO**, dessen Instanzen und zulässige Lösungen aus natürlichen Zahlen bestehen und dessen zugehöriges Entscheidungsproblem **NP**-vollständig ist, in **FPTAS**.

Beweis:

Es sei Π ein polynomiell beschränktes Maximierungsproblem aus **NPO**, dessen Instanzen aus natürlichen Zahlen bestehen und dessen zugehöriges Entscheidungsproblem **NP**-vollständig ist. Für Minimierungsprobleme verläuft der Beweis ähnlich. Es gebe ein Polynom *p*, so dass für jede Instanz $x \in \Sigma_\Pi^*$ und jede zulässige Lösung $y \in \mathrm{SOL}_\Pi(x)$ die Abschätzung $m_\Pi(x, y) \leq p(|x|)$ gilt. Es gebe ein voll polynomiell zeitbeschränktes Approximationsschema $\mathbf{A}(x, r)$ für Π, das bei Eingabe einer Instanz $x \in \Sigma_\Pi^*$ und einer Zahl $r > 1$ eine Näherungslösung mit $R_\mathbf{A}(x) = \dfrac{m_\Pi^*(x)}{m(x, \mathbf{A}(x, r))} \leq r$ in einer Zeit ermittelt, die polynomiell in $|x|$ und $1/(r-1)$ beschränkt ist. Das zugehörige Entscheidungsproblem zu Π sei Π_{ENT} zur Entscheidung der Sprache L_{ENT}. Nach Voraussetzung ist L_{ENT} **NP**-vollständig. Es sei $x \in \Sigma_\Pi^*$ eine Instanz aus Π. Dann gilt $m_\Pi^*(x) \leq p(|x|)$. Es wird $r = 1 + 1/p(|x|)$ gewählt und *x* und *r* in **A** eingegeben. Dann ist

$R_\mathbf{A}(x) = \dfrac{m_\Pi^*(x)}{m(x, \mathbf{A}(x, r))} \leq r = 1 + \dfrac{1}{p(|x|)} = \dfrac{p(|x|)+1}{p(|x|)}$. Daraus folgt

$m(x, \mathbf{A}(x, r)) \geq m_\Pi^*(x) \cdot \dfrac{p(|x|)}{p(|x|)+1} = m_\Pi^*(x) - \dfrac{m_\Pi^*(x)}{p(|x|)+1} > m_\Pi^*(x) - 1$. Die letzte Ungleichung ergibt sich aus der Tatsache, dass $m_\Pi^*(x) \leq p(|x|) < p(|x|)+1$ ist. Da **A** eine zulässige Lösung ermittelt und diese eine natürliche Zahl ist, folgt $m(x, \mathbf{A}(x, r)) = m_\Pi^*(x)$. Wegen $\dfrac{1}{r-1} = p(|x|)$ ist die Laufzeit des Verfahrens polynomiell in $|x|$ und $p(|x|)$, also polynomiell in $|x|$. Daher ist L_{ENT} in polynomieller Zeit entscheidbar und folglich $\mathbf{P} = \mathbf{NP}$ im Widerspruch zur Voraussetzung.

///

In der angegebenen Literatur werden Beispiele für Optimierungsprobleme aus **PTAS** angeführt, die die Voraussetzungen aus Satz 6.3-4 erfüllen. Damit ergibt sich der folgende Satz.

Satz 6.3-5:
Ist $\mathbf{P} \neq \mathbf{NP}$, so gilt $\mathbf{FPTAS} \subset \mathbf{PTAS} \subset \mathbf{APX} \subset \mathbf{NPO}$.

In Kapitel 6.2 werden 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 $|x|$ der Eingabeinstanz.

Zur Erinnerung: die relative Approximationsgüte wurde definiert durch

$R_{\mathbf{A}}(x) = \dfrac{m^*_{\Pi}(x)}{m(x, \mathbf{A}(x))}$ bei einem Maximierungsproblem

bzw.

$R_{\mathbf{A}}(x) = \dfrac{m(x, \mathbf{A}(x))}{m^*_{\Pi}(x)}$ 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 $k' = \dfrac{k}{r \cdot (r + k/m^*_{\Pi}(x))} \leq k/r$,

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 in **NPO**, die ein asymptotisches Approximationsschema besitzen, wird mit $\mathbf{PTAS}^{\infty}$ bezeichnet.

Satz 6.3-6:

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).

Unter Verwendung fortgeschrittener Methoden der Angewandten Mathematik lässt sich sogar ein asymptotisches Approximationsschema entwerfen, das polynomiell in der Problemgröße und in $1/(r-1)$ ist.

Beweis:

Es soll hier nur die Beweisidee skizziert werden; Details können in der angegebenen Literatur nachgelesen werden.

Eine Eingabeinstanz des Binpacking-Minimierungsproblems sei in der in Kapitel 5.3 beschriebenen Form $I = [a_1, \ldots, a_n, b]$ gegeben. Hierbei sind $a_1, \ldots, a_n$ („Objekte“) natürliche Zahlen mit $0 < a_i \leq b$ für i = 1, ..., n und $b \in \mathbf{N}$ („Behältergröße“). Gesucht ist eine Plazierung der Objekte in möglichst wenige Behälter der Größe b. Es kann folgendes asymptotisches Approximationsschema entworfen werden, das bei Eingabe einer Instanz I und einer Zahl $r > 1$ in fünf Schritten arbeitet:

1. Es sei $\delta = (r-1)/2$. Aus I werden die Objekte mit Größe $< \delta \cdot b$ entfernt; die entstehende Instanz sei I'; die Anzahl der Objekte in I' sei n'.

2. Es seien $k = \left| (r-1)^2 \cdot n'/2 \right|$ und $m = \lfloor n'/k \rfloor$. Die Objekte in I' werden nach absteigender Größe sortiert; die entstehende Instanz sei $[a'_1, \ldots, a'_{n'}]$, d.h. $a'_1 \geq, \ldots, \geq a'_{n'}$. Diese Zahlenfolge wird in $m+1$ Gruppen G_i mit $G_i = \{ a'_{(i-1)\cdot k+1}, \ldots, a_{i \cdot k} \}$ für $i = 1, \ldots, m$ und $G_m = \{ a'_{m \cdot k+1}, \ldots, a_{n'} \}$ eingeteilt. Es wird eine Instanz I'' definiert, indem jedes Objekt in Gruppe G_i für $i = 2, \ldots, m+1$ durch das größte Objekt in der Gruppe, d.h. durch $a'_{(i-1)\cdot k+1}$ ersetzt wird. Die Objekte der Gruppe G_1 kommen nicht nach I''. Die Behältergröße von I'' sei b.

3. Für I'' wird eine optimale Lösung, d.h. eine Plazierung der Objekte in I'' in eine möglichst kleine Anzahl Behälter der Größe b, bestimmt.

4. In der optimalen Lösung für I'' werden für $i = 2, ..., m+1$ alle Objekte $a'_{(i-1)k+1}$ wieder durch die Objekte der Gruppe G_i ersetzt, und es werden *k* Behälter zur Aufnahme jeweils eines Objekts in G_1 hinzugefügt.

5. Die in Schritt 1. entfernten kleinen Objekte werden nach dem Firstfit-Prinzip in die in Schritt 4. ermittelten Behälter gelegt bzw. - falls erforderlich - in neue Behälter.

Es lässt sich zeigen, dass der zeitliche Aufwand für Schritt 4 von der Ordnung $O\left(n^q \cdot p(n)\right)$ ist, wobei *q* nur von *r* abhängt und *p* ein Polynom ist. Für festes *r* ist der Aufwand also polynomiell.

Das im Satz erwähnte asymptotische Approximationsschema, das polynomiell in der Problemgröße und in $1/(r-1)$ ist, verwendet fortgeschrittene Methoden der Angewandten Mathematik; auf eine Darstellung so
Weiterhin lässt sich zeigen, dass die Anzahl der ermittelten Behälter durch $r \cdot m^*(I)+1$ beschränkt ist.

///

Offensichtlich gilt $\mathbf{PTAS} \subseteq \mathbf{PTAS}^\infty \subseteq \mathbf{APX}$.

Unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ liegt das Binpacking-Minimierungsproblem nach Satz 6.2-10 nicht in **PTAS**; es ist nach Satz 6.3-6 jedoch in $\mathbf{PTAS}^\infty$. Daher gilt (unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$) $\mathbf{PTAS} \subset \mathbf{PTAS}^\infty$. Ebenfalls unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ lässt sich zeigen, dass die Inklusion $\mathbf{PTAS}^\infty \subset \mathbf{APX}$ echt ist.

Die Klasse $\mathbf{PTAS}^\infty$ ordnet sich in die übrigen Approximationsklassen ein:

Satz 6.3-7:
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[14]. 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

$$T_{\mathbf{A}}^{avg}(n) = \mathbf{E}\left[t_{\mathbf{A}}(x) \mid |x| = n\right] = \int_{|x|=n} t_{\mathbf{A}}(x) \cdot dP(x).$$

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**. Eine Einführung in die Methoden zum Entwurf derartiger Verfahren findet man in der angegebenen Literatur.

---

[14] $t_{\mathbf{A}}(x) =$ 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**[15], 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. Hierbei wird aus den zunächst unsortierten Elementen, die in einem Feld abgespeichert sind, ein Element ausgewählt, an die Position geschoben, die es in der sortierten Reihenfolge einnimmt, und die verbleibenden beiden Feldabschnitte anschließend nach dem gleichen Prinzip sortiert. Wählt man das Element zufällig aus, so erreicht der `Quicksort` eine Laufzeit, die seinem mittleren Laufzeitverhalten entspricht, d.h. von der Ordnung $O(n \cdot \log(n))$ ist.

Beispiele für randomisierte Algorithmen vom Monte-Carlo-Typ sind die heute üblichen Primzahltests; exemplarisch wird im folgenden ein Verfahren informell beschrieben (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 $p \leq \sqrt{n}$. Umgekehrt, falls alle Primzahlen *p* mit $p \leq \sqrt{n}$ 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 $\beta(n)$ einer Zahl $n \geq 1$ im Zahlensystem zur Basis *B* ist

[15] Die Typbezeichnung „Monte-Carlo“ steht für *mostly correct*.

$\beta(n) = \lfloor \log_B(n) \rfloor + 1 = \lceil \log_B(n+1) \rceil$, d.h. $\beta(n) \in O(\log(n))$. Die Anzahl der Primzahlen unterhalb $\lfloor \sqrt{n} \rfloor$ beträgt für große *n* nach dem Primzahlsatz der Zahlentheorie $\pi(\sqrt{n}) \sim \frac{n^{1/2}}{\ln(n^{1/2})}$, also ein Wert der Ordnung $O\left(\frac{2^{\beta(n)/2}}{\beta(n)}\right)$. Jede in Frage kommende Primzahl muss daraufhin untersucht werden, ob sie *n* teilt. Dazu sind mindestens $O\left((\beta(n))^2\right)$ viele Bitoperationen erforderlich. Daher ist der Gesamtaufwand mindestens von der Ordnung $O\left(\beta(n) \cdot 2^{\beta(n)/2}\right)$.

Durch Anwendung zahlentheoretischer Erkenntnisse hat man versucht, effiziente Primzahltests zu entwickeln. Im Jahr 2002 wurde ein deterministisches Verfahren veröffentlicht, das eine Zahl *n* daraufhin testet, ob sie eine Primzahl ist[16]. Die Laufzeit dieses Verfahrens ist von der Ordnung $O\left((\log(n))^{12}\right)$ und damit trotz der polynomiellen Laufzeit für große *n* nicht praktikalbel. Der bis dahin 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 $c > 0$. Das ist jedoch 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.

---

[16] Agrawal, M.; Kayal, N.; Saxena, N.: PRIMES is in P, preprint, http://www.cse.iitk.ac.in/news/primality.ps, Aug. 8, 2002.

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* festgestellt, 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 fin-

den, 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 7.1-1:

Ist $n \in \mathbf{N}$ eine Primzahl und $a \in \mathbf{N}$ eine Zahl mit $ggT(a,n) = 1$, dann ist $a^{n-1} \equiv 1 \,(\mathrm{mod}\, n)$.

Für die Primzahl $n = 13$ zeigt folgende Tabelle für alle a mit $1 \le a \le n-1$ die Werte $a^i \,(\mathrm{mod}\, 13)$ mit $i = 1, ..., 12$:

$a \bmod 13$	$a^i \bmod 13$ für $i = 1, ..., 12$
1	1 für $i = 1, ..., 12$
2	$2,\ 4,\ 8,\ 16 \equiv 3,\ 6,\ 12,\ 24 \equiv 11,\ 22 \equiv 9,\ 18 \equiv 5,\ 10,\ 20 \equiv 7,\ 14 \equiv 1$
3	$3,\ 9,\ 27 \equiv 1,\ 3,\ 9,\ 27 \equiv 1,\ 3,\ 9,\ 27 \equiv 1,\ 3,\ 9,\ 27 \equiv 1$
4	$4,\ 16 \equiv 3,\ 12,\ 48 \equiv 9,\ 36 \equiv 10,\ 40 \equiv 1,\ 4,\ 16 \equiv 3,\ 12,\ 48 \equiv 9,\ 36 \equiv 10,\ 40 \equiv 1$
5	$5,\ 25 \equiv 12,\ 60 \equiv 8,\ 40 \equiv 1,\ 5,\ 25 \equiv 12,\ 60 \equiv 8,\ 40 \equiv 1,\ 5,\ 25 \equiv 12,\ 60 \equiv 8,\ 40 \equiv 1$
6	$6,\ 36 \equiv 10,\ 60 \equiv 8,\ 48 \equiv 9,\ 54 \equiv 2,\ 12,\ 72 \equiv 7,\ 42 \equiv 3,\ 18 \equiv 5,\ 30 \equiv 4,\ 24 \equiv 11,\ 66 \equiv 1$
7	$7,\ 49 \equiv 10,\ 70 \equiv 5,\ 35 \equiv 9,\ 63 \equiv 11,\ 77 \equiv 12,\ 84 \equiv 6,\ 42 \equiv 3,\ 21 \equiv 8,\ 56 \equiv 4,\ 28 \equiv 2,\ 14 \equiv 1$
8	$8,\ 64 \equiv 12,\ 96 \equiv 5,\ 40 \equiv 1,\ 8,\ 64 \equiv 12,\ 96 \equiv 5,\ 40 \equiv 1,\ 8,\ 64 \equiv 12,\ 96 \equiv 5,\ 40 \equiv 1$
9	$9,\ 81 \equiv 3,\ 27 \equiv 1,\ 9,\ 81 \equiv 3,\ 27 \equiv 1,\ 9,\ 81 \equiv 3,\ 27 \equiv 1,\ 9,\ 81 \equiv 3,\ 27 \equiv 1$
10	$10,\ 100 \equiv 9,\ 90 \equiv 12,\ 120 \equiv 3,\ 30 \equiv 4,\ 40 \equiv 1,\ 10,\ 100 \equiv 9,\ 90 \equiv 12,\ 120 \equiv 3,\ 30 \equiv 4,\ 40 \equiv 1$
11	$11,\ 121 \equiv 4,\ 44 \equiv 5,\ 55 \equiv 3,\ 33 \equiv 7,\ 77 \equiv 12,\ 132 \equiv 2,\ 22 \equiv 9,\ 99 \equiv 8,\ 88 \equiv 10,\ 110 \equiv 6,\ 66 \equiv 1$
12	$12,\ 144 \equiv 1,\ 12,\ 144 \equiv 1,\ 12,\ 144 \equiv 1,\ 12,\ 144 \equiv 1,\ 12,\ 144 \equiv 1,\ 12,\ 144 \equiv 1$

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 $ggT(a,n) = 1$. Die ersten zehn Carmichael-Zahlen sind 561, 1105, 1729, 2465, 2821, 6601, 8911, 10585, 15841, 29341.

Ist *n* also eine Carmichael-Zahl (wobei man einige Carmichael-Zahlen wie 561, 1105, 2465 oder 10585 leicht als Nichtprimzahlen erkennt) 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 $1 \le a \le n-1$.

Trotzdem führt die Eigenschaft $E(a) =$ „$ggT(a,n) = 1$ und $a^{n-1} \equiv 1 \pmod n$“ schon auf einen brauchbaren randomisierten Primzahltest, der jedoch bei Eingabe einer Carmichael-Zahl versagt. Das Beispiel der Nichtprimzahl *n* = 15 belegt, dass (iii) doch gelten kann:

$a \bmod 15$	$a^i \bmod 15$ für $i = 1,...,14$
1	1 für $i = 1,...,14$
2	$2, 4, 8, 16 \equiv 1, 2, 4, 8, 16 \equiv 1, 2, 4, 8, 16 \equiv 1, 2, 4$
3	$3, 9, 27 \equiv 12, 36 \equiv 6, 18 \equiv 3, 9, 27 \equiv 12, 36 \equiv 6, 18 \equiv 3, 9, 27 \equiv 12, 36 \equiv 6, 18 \equiv 3, 9$
4	$4, 16 \equiv 1, 4, 16 \equiv 1, 4, 16 \equiv 1, 4, 16 \equiv 1, 4, 16 \equiv 1, 4, 16 \equiv 1, 4, 16 \equiv 1$ ***(falscher) Zeuge dafür, dass 15 Primzahl ist***
5	$5, 25 \equiv 10, 50 \equiv 5, 25 \equiv 10, 50 \equiv 5, 25 \equiv 10, 50 \equiv 5, 25 \equiv 10, 50 \equiv 5, 25 \equiv 10, 50 \equiv 5, 25 \equiv 10, 50 \equiv 5, 25 \equiv 10$
6	$6, 36 \equiv 6, 36 \equiv 6, 36 \equiv 6, 36 \equiv 6, 36 \equiv 6, 36 \equiv 6, 36 \equiv 6, 36 \equiv 6, 36 \equiv 6, 36 \equiv 6, 36 \equiv 6, 36 \equiv 6, 36 \equiv 6$
7	$7, 49 \equiv 4, 28 \equiv 13, 91 \equiv 1, 7, 49 \equiv 4, 28 \equiv 13, 91 \equiv 1, 7, 49 \equiv 4, 28 \equiv 13, 91 \equiv 1, 7, 49 \equiv 4$
8	$8, 64 \equiv 4, 32 \equiv 2, 16 \equiv 1, 8, 64 \equiv 4, 32 \equiv 2, 16 \equiv 1, 8, 64 \equiv 4, 32 \equiv 2, 16 \equiv 1, 8, 64 \equiv 4$
9	$9, 81 \equiv 6, 54 \equiv 9, 81 \equiv 6, 54 \equiv 9, 81 \equiv 6, 54 \equiv 9, 81 \equiv 6, 54 \equiv 9, 81 \equiv 6, 54 \equiv 9, 81 \equiv 6, 54 \equiv 9, 81 \equiv 6$
10	$10, 100 \equiv 10, 100 \equiv 10, 100 \equiv 10, 100 \equiv 10, 100 \equiv 10, 100 \equiv 10, 100 \equiv 10, 100 \equiv 10, 100 \equiv 10, 100 \equiv 10, 100 \equiv 10, 100 \equiv 10, 100 \equiv 10$
11	$11, 121 \equiv 1, 11, 121 \equiv 1, 11, 121 \equiv 1, 11, 121 \equiv 1, 11, 121 \equiv 1, 11, 121 \equiv 1, 11, 121 \equiv 1, 11, 121 \equiv 1$ ***(falscher) Zeuge dafür, dass 15 Primzahl ist***
12	$12, 144 \equiv 9, 108 \equiv 3, 36 \equiv 6, 72 \equiv 12, 144 \equiv 9, 108 \equiv 3, 36 \equiv 6, 72 \equiv 12, 144 \equiv 9, 108 \equiv 3, 36 \equiv 6, 72 \equiv 12, 144 \equiv 9$
13	$13, 169 \equiv 4, 52 \equiv 7, 91 \equiv 1, 13, 169 \equiv 4, 52 \equiv 7, 91 \equiv 1, 13, 169 \equiv 4, 52 \equiv 7, 91 \equiv 1, 13, 169 \equiv 4$
14	$14, 196 \equiv 1, 14, 196 \equiv 1, 14, 196 \equiv 1, 14, 196 \equiv 1, 14, 196 \equiv 1, 14, 196 \equiv 1, 14, 196 \equiv 1$ ***(falscher) Zeuge dafür, dass 15 Primzahl ist***

Satz 7.1-2:

Entweder gilt für alle Zahlen a mit $1 \le a \le n-1$ und $ggT(a,n) = 1$ die Eigenschaft $a^{n-1} \equiv 1 \pmod n$ oder für höchstens die Hälfte.

Anders formuliert:

Entweder sind mit der Zeugeneigenschaft $E(a) =$ „$ggT(a,n) = 1$ und $a^{n-1} \equiv 1 \pmod n$“ alle Zahlen a mit $1 \le a \le n-1$ Zeugen für die Primzahleigenschaft für n oder höchstens die Hälfte.

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 7.1-4 (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 $a^{2^{j-1} \cdot r} \equiv 1 \pmod n$, dann kann man den Vorgang des Wurzelziehens wiederholen: In Satz 7.1-4 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 $\left(a^r, a^{2r}, a^{4r}, ..., a^{2^{j-1} \cdot r}, a^{2^j \cdot r}\right) \bmod n$ 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 höchstens 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

$a^{n-1} \equiv 1 \pmod{n}$ und

die Folge $\left(a^r, a^{2r}, a^{4r}, ..., a^{2^{j-1} \cdot r}, a^{2^j \cdot r}\right) \bmod n$ 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 7.1-6 größer als $1 - (1/4)^m$.

Um für praktische Belange die Güte des Verfahrens abzuschätzen, werden in folgender Tabelle einige Werte für $(1/4)^m$ abgeschätzt:

m	10	25	30	50	100	168	1000
$(1/4)^m$	$< 10^{-6}$	$< 10^{-15}$	$< 10^{-18}$	$< 10^{-30}$	$< 10^{-60}$	$< 10^{-101}$	$< 10^{-602}$

Ist *n* also eine zusammengesetzte Zahl und werden $m = 100$ zufällig erzeugte Zahlen *a* ausgewählt und auf Zeugeneigenschaft untersucht, so ist die Wahrscheinlichkeit, dass alle diese ausgewählten Zahlen „falsche Zeugen“ sind, d.h. die Primzahleigenschaft von *n* fälschlicherweise bezeugen, kleiner als 10^{-60}.

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.

In vielen Fällen liefern stochastische Lösungsansätze gute Approximationsergebnisse. Dazu soll noch einmal das Binpacking-Minimierungsproblem betrachtet werden, das in **PTAS**$^{\infty}$ liegt, also asymptotisch beliebig dicht in polynomieller Zeit approximiert werden kann. Das Verfahren, das die Grundlage zu Satz 6.3-6 liefert, ist ein offline-Verfahren. Das bedeutet, dass zu Beginn der Verarbeitungsphase alle Objekte der Eingabeinstanz bekannt sein müssen. Sie werden nämlich zunächst nach aufsteigender Größe sortiert. In vielen praktischen Anwendungen, die gemäß dem Binpacking-Problem modelliert werden, müssen die Objekte jedoch sequentiell verarbeitet werden, und die Entscheidung, in welchen Behälter ein Objekt gelegt werden soll, muss nacheinander für jedes Objekt unwiderruflich getroffen werden, ohne die nachfolgenden Objekte zu kennen. Es wird also ein online-Verfahren gesucht.

Kennt man die Wahrscheinlichkeitsverteilung der Größen der Objekte, so kann man versuchen, diese Kenntnis beim Entwurf eines Approximationsverfahrens zu nutzen. Im folgenden wird ein online-Approximationsverfahren für das Binpacking-Minimierungsproblem beschrieben, das voraussetzt, dass die Größen der Objekte einer Eingabeinstanz über dem Intervall]0, 1] gleichverteilt sind:

Asymptotisches online-Approximationsschema für das Binpacking-Minimierungsproblem
(online-Faltungsalgorithmus)

Eingabe: $I = [a_1, \ldots, a_n]$, $s \in \mathbf{N}$ mit $s \geq 4$ und $s \equiv 0 \bmod(2)$
$a_1, \ldots, a_n$ („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, ..., n$

Verfahren:

```
VAR idx : INTEGER;
    k   : INTEGER;
    I   : ARRAY [1..s] OF Intervall ⊆ ]0,1];

BEGIN
  FOR idx := 1 TO s DO
    I[idx] := ⌈(s-idx)/s, (s-idx+1)/s⌉;
```

```
FOR k := 1 TO n DO
  BEGIN
    idx := s + 1 - ⌈s·a_k⌉; { Bestimmung eines Intervalls
                               I[idx] mit a_k ∈ I[idx]     }
    IF (idx >= 2)
       AND
       (es gibt einen Behälter B, der ein Objekt b ∈ I[s – (idx - 2)]
        enthält)
    THEN BEGIN
            B := B ∪ { a_k };
             kennzeichne B als gefüllt;
         END
    ELSE BEGIN
            plaziere a_k in einen neuen Behälter;
            { für idx = 1 ist dieser Behälter gefüllt }
         END;
  END;
```

Ausgabe: **OFD**$(I) =$ benötigte nichtleere Behälter $B_1, ..., B_k$.

Offensichtlich hat das Verfahren polynomielle Laufzeit in der Größe der Eingabeinstanz und s. Es lässt sich unter Nutzung einer Reihe von weiterführenden Ergebnissen der Wahrscheinlichkeitstheorie, insbesondere aus der Warteschlangentheorie, zeigen[17]:

Satz 7.1-6:

Bezeichnet wieder $m^*(I_n)$ das Maß einer minimalen Lösung bei einer Eingabeinstanz $I_n = [a_1, \ldots, a_n]$ für das Binpacking-Minimierungsproblem und sind die Objekte in einer Eingabeinstanz im Intervall $]0,1]$ gleichverteilt, so gilt

$$1 - \frac{1}{s+1} \le \lim_{n\to\infty} \mathbf{E}\left[m^*(I_n)/m(I_n, \mathbf{OFD}(I_n))\right] \le 1 .$$

Der Erwartungswert des Kehrwerts der Approximationsgüte nähert sich also für Eingabeinstanzen mit vielen Objekten dem Wert 1, und zwar gesteuert durch den Parameter s.

Das Ergebnis lässt sich auch auf andere Verteilungen verallgemeinern, nämlich auf Verteilungen mit monoton fallender Dichtefunktion der Größe der Eingabeobjekte. Diese Fälle sind

[17] Hoffmann, U.: A class of simple stochastic bin packing algorithms, Computing 29, 227 – 239, 1982.

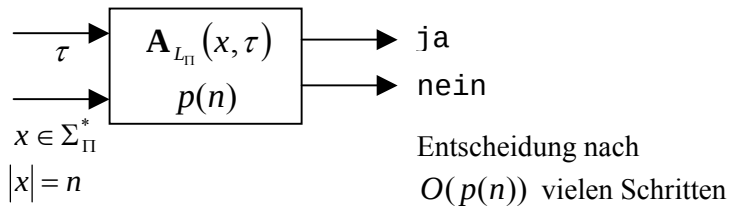
in der Praxis besonders interessant, da unter dieser Voraussetzung kleine Objekte häufiger als große Objekte auftreten.

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 $\tau \in \{0,1\}^*$ 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 $\tau \in \{0,1\}^*$ zufälliger Bits:



Für jedes $x \in L_\Pi$ ist $P(\mathbf{A}_{L_\Pi}(x,\tau) = \texttt{ja}) \geq 1/2$,
für jedes $x \notin L_\Pi$ ist $P(\mathbf{A}_{L_\Pi}(x,\tau) = \texttt{nein}) = 1$

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 τ lesen, so dass man annehmen kann, dass τ aus polynomiell vielen Bits besteht.

Ein Beispiel für eine Sprache aus **RP** ist die Menge $NONPRIMES = \{ n \mid n \in \mathbf{N} \text{ und } n \text{ ist keine Primzahl} \}$. Einen **RP**-Akzeptor $\mathbf{A}_{NONPRIMES}$ für *NOPRIMES* erhält man aus der Funktion `random_is_prim` aus Kapitel 7.1. $\mathbf{A}_{NONPRIMES}$ liest zwei Eingaben, nämlich eine Zahl $n \in \mathbf{N}$ und eine Folge $\tau \in \{0,1\}^*$ zufälliger Bits der Länge $k = size(n)$. Diese Zufallsfolge wird als Binärkodierung einer Zufallszahl a mit $1 \le a \le n-1$ interpretiert (die Fälle $a = 0$ und $a = n$ sollen zur vereinfachten Darstellung hier ausgeschlossen sein). Dazu wird angenommen, dass es eine Funktion `make_INTEGER(`τ`,` $size(n)$`)` gibt, die die Zufallsfolge τ in eine natürliche Zahl a mit $1 \le a \le n-1$ umwandelt. Dann wird die Bedingung $E(a)$ (siehe Kapitel 7.1) überprüft und eine Entscheidung getroffen:

```
FUNCTION A_NONPRIMES (n : INTEGER;
                      τ : ...      ): ...;

   VAR a : INTEGER;

   BEGIN { A_NONPRIMES }
     a := make_INTEGER(τ, size(n));
     IF ( E(a) trifft nicht zu)
     THEN A_NONPRIMES := ja
     ELSE A_NONPRIMES := nein;
END   { A_NONPRIMES };
```

Ist $n \in NONPRIMES$, 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 $P(\mathbf{A}_{NONPRIMES}(n,\tau) = \text{ja}) \ge 1/2$.
Ist $n \notin NONPRIMES$, 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 $\mathbf{A}_{NONPRIMES}$ mit $\mathbf{A}_{NONPRIMES}$ `:= 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 τ. Ist $x \in L_\Pi$, dann gibt es eine Zufallsfolge τ mit $\mathbf{A}_{L_\Pi}(x,\tau) = \texttt{ja}$ (da $x \in L_\Pi$ und in diesem Fall $P(\mathbf{A}_{L_\Pi}(x,\tau) = \texttt{ja}) \geq 1/2$ gelten, führen mindestens die Hälfte aller Zufallsfolgen auf die `ja`-Entscheidung). Ist $x \notin L_\Pi$, dann führen wegen $P(\mathbf{A}_{L_\Pi}(x,\tau) = \texttt{nein}) = 1$ 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:

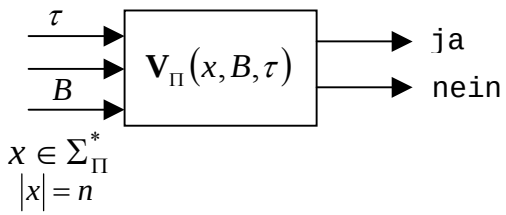
Es seien $r : \mathbf{N} \to \mathbf{N}$ und $q : \mathbf{N} \to \mathbf{N}$ Funktionen. Ein Verifizierer (nichtdeterministischer Algorithmus) $\mathbf{V}_\Pi$ heißt **$(r(n), q(n))$-beschränkter Verifizierer für das Entscheidungsproblem** Π über einem Alphabet Σ_Π, wenn er Zugriff auf eine Eingabe $x \in \Sigma_\Pi^*$ mit $|x| = n$, einen Beweis $B \in \Sigma_0^*$ und eine Zufallsfolge $\tau \in \{0,1\}^*$ 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 $\tau \in \{0,1\}^*$.
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 $\mathbf{V}_\Pi$ in polynomieller Zeit auf die `ja`- bzw- `nein`-Entscheidung.

Die Entscheidung, die $\mathbf{V}_\Pi$ bei Eingabe von $x \in \Sigma_\Pi^*$, $B \in \Sigma_0^*$ und $\tau \in \{0,1\}^*$ liefert, wird mit $\mathbf{V}_\Pi(x, B, \tau)$ 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 $P(\mathbf{V}_\Pi(x, B_x, \tau) = \texttt{ja}) = 1$,
für jedes $x \notin L_\Pi$ und alle Beweise B gilt $P(\mathbf{V}_\Pi(x, B, \tau) = \texttt{nein}) \geq 1/2$.



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

$$\mathbf{P} = \bigcup_{k=0}^{\infty} TIME(n^k) = TIME(poly(n)) \text{ und } \mathbf{NP} = \bigcup_{k=0}^{\infty} NTIME(n^k) = NTIME(poly(n)).$$

Satz 7.2-2:

$$\mathbf{PCP}(r(n), q(n)) \subseteq NTIME(q(n) \cdot 2^{O(r(n))})$$

Beweis:

Es sei $L_\Pi \in \mathbf{PCP}(r(n), q(n))$. Dann gibt es einen $(r(n), q(n))$-beschränkten Verifizierer $\mathbf{V}_\Pi$ für L_Π. Aus diesem kann man auf folgende Weise einen Verifizierer $\widetilde{\mathbf{V}}_\Pi$ (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 $O(r(n))$ simuliert $\widetilde{\mathbf{V}}_\Pi$ in jeweils polynomiell vielen Schritten die Berechnung des Ve-

rifizierers $\mathbf{V}_\Pi$ und akzeptiert x genau dann, wenn $\mathbf{V}_\Pi$ die Eingabe x für jede Zufallsfolge akzeptiert. $\widetilde{\mathbf{V}}_\Pi$ ist daher eine nichtdeterministische Turingmaschine, deren Laufzeit bei Eingabe von $x \in \Sigma_\Pi^*$ mit $|x| = n$ von der Ordnung $O\left(q(n) \cdot 2^{O(r(n))}\right)$ ist. Außerdem gilt:
Ist $x \in L_\Pi$, dann gibt es einen Beweis B_x mit $P\left(\mathbf{V}_\Pi(x, B_x, \tau) = \texttt{ja}\right) = 1$, d.h. $\mathbf{V}_\Pi$ kommt bei Eingabe von x, B_x und jeder Zufallsfolge τ auf den `ja`-Ausgang. Also akzeptiert $\widetilde{\mathbf{V}}_\Pi$ die Eingabe x. Ist $x \notin L_\Pi$, so verwirft $\mathbf{V}_\Pi$ für jeden Beweis B wegen $P\left(\mathbf{V}_\Pi(x, B, \tau) = \texttt{nein}\right) \geq 1/2$ die Eingabe für mindestens die Hälfte aller in Frage kommenden Zufallsfolgen, daher akzeptiert $\widetilde{\mathbf{V}}_\Pi$ die Eingabe x nicht.
Daher ist $L \in NTIME\left(q(n) \cdot 2^{O(r(n))}\right)$.

///

Setzt man im speziellen für $r(n)$ und $q(n)$ Polynome ein, so gilt sogar:

Satz 7.2-3:

$$\mathbf{PCP}\left(poly(n), poly(n)\right) \subseteq NTIME\left(2^{poly(n)}\right)$$

Weiter gelten folgende Aussagen, die die Klassen **P** und **NP** im **PCP**-Modell beschreiben:

Satz 7.2-4:

(i) $\mathbf{PCP}(0,0) = \mathbf{P}$

(ii) $\mathbf{PCP}\left(0, poly(n)\right) = \mathbf{NP}$

(iii) $\mathbf{PCP}\left(\log(n), poly(n)\right) = \mathbf{NP}$

Beweis:

Zu (i): $\mathbf{PCP}(0,0)$ besteht aus den Sprachen L_Π, die wegen $q(n) = 0$ keine Zeichen eines Beweises lesen und ihn daher gar nicht benötigen. Da auch keine Zufallsbits gelesen werden, ist für $x \notin L_\Pi$ und alle Beweise B wegen $P\left(\mathbf{V}_\Pi(x, B, \tau) = \texttt{nein}\right) \geq 1/2$ nur $P\left(\mathbf{V}_\Pi(x, B, \tau) = \texttt{nein}\right) = 1$ möglich. Daher wird L_Π von $\mathbf{V}_\Pi$ deterministisch in polynomieller Zeit entschieden. Das bedeutet $\mathbf{PCP}(0,0) \subseteq \mathbf{P}$.

Umgekehrt ist $\mathbf{P} \subseteq \mathbf{PCP}(0,0)$, da jeder deterministische polynomiell zeitbeschränkter Algorithmus ohne Einsatz eines Beweises und einer Zufallsfolge ein $(0,0)$-beschränkter Verifizierer ist.

Zu (ii): $\mathbf{PCP}(0,poly(n))$ erlaubt beliebig lange Beweise, von denen aber nur poynomiell viele Bits gelesen werden. Da es keine Zufallsfolge gibt, sind dieses immer dieselben Bits, so dass jeder Beweis auf einen in der Länge polynomiell beschränkten Beweis reduziert werden kann. Dieser wird dann ganz gelesen. Wieder ist wegen der fehlenden Zufallsbits bei $x \notin L_\Pi$ und wegen $P(\mathbf{V}_\Pi(x,B,\tau) = \texttt{nein}) \geq 1/2$ nur $P(\mathbf{V}_\Pi(x,B,\tau) = \texttt{nein}) = 1$ möglich. Daher ist $\mathbf{PCP}(0,poly(n)) \subseteq \mathbf{NP}$.

Die Umkehrung $\mathbf{NP} \subseteq \mathbf{PCP}(0,poly(n))$ ist wieder offensichtlich.

Zu (iii): Die Inklusion $\mathbf{PCP}(\log(n),poly(n)) \subseteq \mathbf{NP}$ folgt direkt aus Satz 7.2-2.

Die Beziehung $\mathbf{NP} \subseteq \mathbf{PCP}(\log(n),poly(n))$ folgt aus (ii).

///

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 1990-er 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 $\mathbf{P} = \mathbf{NP}$. In Kapitel 6.1 wird gezeigt, dass es in **APX** liegt.

3-SAT-Maximierungsproblem

Instanz:
1. $I = (K,V)$
   $K = \{F_1, F_2, ..., F_m\}$ 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. $y_{i_j} = \neg x_l$)
2. SOL(I) = Belegung der Variablen in V mit Wahrheitswerten TRUE oder FALSE, d.h. SOL(I) ist eine Abbildung $f : V \to \{\mathsf{TRUE}, \mathsf{FALSE}\}$
3. Für $f \in \text{SOL}(I)$ ist $m(I, f)$ = Anzahl der Klauseln in K, die durch f erfüllt werden
4. *goal* = max

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 $L_{\text{SAT}} = \{F \mid F \text{ ist ein erfüllbarer Boolescher Ausdruck }\}$.
$L_{\text{SAT}} \subseteq \Sigma^*_{\text{BOOLE}} = \{F \mid F \text{ ist ein Boolescher Ausdruck}\}$, L_{SAT} ist **NP**-vollständig.

Es wird eine Abbildung h definiert, die jedem $F \in \Sigma^*_{\text{BOOLE}}$ eine Instanz $h(F) = (K,V)$ für das 3-SAT-Maximierungsproblem zuordnet, wobei $h(F)$ in polynomieller Zeit aus F berechnet werden kann, und die folgende Eigenschaft besitzt:
Ist $F \in L_{\text{SAT}}$, dann sind alle Klauseln in K erfüllbar.

Ist $F \notin L_{\text{SAT}}$, dann gibt es eine Konstante $\varepsilon > 0$, so dass mindestens ein Anteil der Größe ε aller Klauseln in *K* nicht erfüllbar ist. Das bedeutet mit $c(F) = |K|$ für $h(F) = (K,V)$:

$m^*(h(F)) = \begin{cases} = c(F) & \text{für } F \in L_{SAT} \\ \leq c(F) \cdot (1-\varepsilon) & \text{für } F \notin L_{SAT} \end{cases}$. Mit Satz 6.2-9 folgt, dass es keinen polynomiell zeitbeschränkten *r*-approximativen Algorithmus für das 3-SAT-Maximierungsproblem mit $r < 1/(1-\varepsilon)$ 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 τ 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 τ 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. $V = \{x_1, x_2, x_3, \ldots, x_k\}$ mit $k = q \cdot n^c$. *V* enthält polynomiell viele Variablen und ist in polynomieller Zeit aus *F* konstruierbar. Zu jedem Beweis $B = b_1 \ldots b_k$ der Länge $k = q \cdot n^c$ lässt sich eine Belegung $f_B : V \to \{\mathsf{TRUE}, \mathsf{FALSE}\}$ der Variablen in *V* durch

$f_B(x_i) = \begin{cases} \mathsf{TRUE} & \text{für } b_i = 1 \\ \mathsf{FALSE} & \text{für } b_i = 0 \end{cases}$

definieren.
Ist umgekehrt eine Belegung $f : V \to \{\mathsf{TRUE}, \mathsf{FALSE}\}$ der Variablen in *V* gegeben, so lässt sich ein Beweis $B_f = b_{f,1} \ldots b_{f,k}$ angeben mit

$b_{f,i} = \begin{cases} 1 & \text{für } f(x_i) = \mathsf{TRUE} \\ 0 & \text{für } f(x_i) = \mathsf{FALSE} \end{cases}$.

Für die Zufallsfolge τ seien die *q* Positionen, die in einem Beweis erfragt werden, die Positionen $\tau_1, \ldots, \tau_q$. 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_τ wird die Menge derjenigen 0-1-Kombinationen bezeichnet, für die

$\mathbf{V}_{SAT}$ zur `nein`-Entscheidung kommt. Offensichtlich ist $|A_\tau| \le |\Sigma_0|^q = 2^q$. Für jedes $(b_1, ..., b_q) \in A_\tau$ wird eine Formel $(y_{\tau_1} \vee ... \vee y_{\tau_q})$ gebildet mit $y_{\tau_i} = \begin{cases} x_{\tau_i} & \text{für } b_i = 0 \\ \neg x_{\tau_i} & \text{für } b_i = 1 \end{cases}$.

Alle so entstehenden Formeln (für alle Zufallsfolgen τ) 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 $(y_{\tau_1} \vee ... \vee y_{\tau_q})$ bereits die gewünschte Form. Für $q > 3$ werden für jede Formel $G = (y_{\tau_1} \vee ... \vee y_{\tau_q})$ $q-3$ neue Variablen $z_{G,1}, ..., z_{G,q-3}$ eingeführt und G durch $(y_{\tau_1} \vee y_{\tau_2} \vee z_{G,1})$, $(\neg z_{G,1} \vee y_{\tau_3} \vee z_{G,2})$, ..., $(\neg z_{G,q-4} \vee y_{\tau_{q-2}} \vee z_{G,q-3})$, $(\neg z_{G,q-3} \vee y_{\tau_{q-1}} \vee y_{\tau_q})$ ersetzt. Ist *G* erfüllbar, etwa weil y_{τ_1} oder y_{τ_2} den Wert `TRUE` besitzt, dann werden alle neuen Variablen $z_{G,i}$ für $i = 1, ..., q-3$ mit `FALSE` belegt, und damit alle neuen Klauseln erfüllt. Ist *G* erfüllbar, wobei $y_{\tau_1}, ..., y_{\tau_{j-1}}$ mit `FALSE` und y_{τ_j} für $j > 2$ mit `TRUE` belegt ist, dann werden $z_{G,1}$, ..., $z_{G,j-2}$ mit `TRUE` und $z_{G,j-1}$, ..., $z_{G,q-3}$ mit `FALSE` belegt; wieder sind alle neuen Klauseln erfüllt.
Sind umgekehrt alle neuen Klauseln erfüllt, dann können nicht alle y_{τ_i} für $i = 1, ..., q$ mit `FALSE` belegt sein: Denn wären alle Klauseln $(y_{\tau_1} \vee y_{\tau_2} \vee z_{G,1})$, $(\neg z_{G,1} \vee y_{\tau_3} \vee z_{G,2})$, ..., $(\neg z_{G,q-4} \vee y_{\tau_{q-2}} \vee z_{G,q-3})$, $(\neg z_{G,q-3} \vee y_{\tau_{q-1}} \vee y_{\tau_q})$ erfüllt und gleichzeitig alle y_{τ_i} gleich `FALSE`, so müssten $z_{G,1}, ..., z_{G,q-3}$ mit `TRUE` belegt sein; dann ist aber die letzte neue Klausel $(\neg z_{G,q-3} \vee y_{\tau_{q-1}} \vee y_{\tau_q})$ nicht erfüllt. Man sieht also:

G ist genau dann erfüllbar, wenn alle aus *G* neu entstandenen Formeln erfüllbar sind.

Die Variablenmenge *V* wird um die neu hinzugenommen Variablen erweitert. 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 $h(F) = (K, V)$ erfolgt in polynomieller Zeit.

Ist $F \in L_{\text{SAT}}$, dann gibt es einen Beweis B_F, so dass der Verifizierer $\mathbf{V}_{SAT}$ für jede Zufallsfolgen τ die Entscheidung $\mathbf{V}_{SAT}(F, B_F, \tau) = \texttt{ja}$ trifft, denn $P(\mathbf{V}_{SAT}(F, B_F, \tau) = \texttt{ja}) = 1$. Ist für eine Zufallsfolgen τ die definierte Menge $A_\tau = \varnothing$, so wurde keine Formel in K' bzw. *K* aufgenommen. Ist $A_\tau \neq \varnothing$ und sind τ_1, ..., τ_q die Positionen in B_F, die von $\mathbf{V}_{SAT}$ aufgrund der

Zufallsfolge τ erfragt werden, etwa mit den Werten $a_1, ..., a_q$, dann wird durch die Festlegung

$$x_{\tau_i} = \begin{cases} \mathsf{TRUE} & \text{für } a_i = 1 \\ \mathsf{FALSE} & \text{für } a_i = 0 \end{cases}$$

eine partielle Belegung der Variablen in *V* definiert.
Es sei $(b_1, ..., b_q) \in A_\tau$ mit der dazu gebildeten Formel $(y_{\tau_1} \vee ... \vee y_{\tau_q}) \in K'$. Dann ist $(a_1, ..., a_q) \neq (b_1, ..., b_q)$, denn $(a_1, ..., a_q)$ führt auf eine `ja`-Entscheidung, und alle $(b_1, ..., b_q) \in A_\tau$ 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 $y_{\tau_j} = x_{\tau_j}$ wurde auf TRUE gesetzt. Ist $b_j = 1$, dann ist $a_j = 0$. Das Literal $y_{\tau_j} = \neg x_{\tau_j}$ wurde auf TRUE gesetzt. In beiden Fällen ist $(y_{\tau_1} \vee ... \vee y_{\tau_q})$ 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 $F \notin L_{\text{SAT}}$, 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 $\mathbf{V}_{SAT}(F, B, \tau) = \mathtt{nein}$ trifft, denn für jeden Beweis *B* ist in diesem Fall $P(\mathbf{V}_{SAT}(F, B, \tau) = \mathtt{nein}) \geq 1/2$. Es sei eine Belegung $f : V \to \{\mathsf{TRUE}, \mathsf{FALSE}\}$ der Variablen in *V* gegeben. Der zu *f* konstruierbare Beweis $B_f = b_{f,1} ... b_{f,k}$ (siehe oben) ist bestimmt durch

$$b_{f,i} = \begin{cases} 1 & \text{für } f(x_i) = \mathsf{TRUE} \\ 0 & \text{für } f(x_i) = \mathsf{FALSE} \end{cases}.$$

Es sei τ eine der $2^{c \cdot \log(n)}/2$ Zufallsfolgen, die auf die `nein`-Entscheidung führen. Die Positionen in B_f, die von $\mathbf{V}_{SAT}$ aufgrund der Zufallsfolge τ erfragt werden, seien τ_1, ..., τ_q, die Werte an diesen Positionen in B_f seien $a_1, ..., a_q$. Dann ist nach Definition $(a_1, ..., a_q) \in A_\tau$. Die aus $(a_1, ..., a_q)$ gebildete Formel $(y_{\tau_1} \vee ... \vee y_{\tau_q})$ hat den Wert FALSE: Ist nämlich $a_i = 0$, dann ist x_{τ_i} mit FALSE belegt, und $y_{\tau_i} = x_{\tau_i}$ hat ebenfalls den Wert FALSE; ist $a_i = 1$, dann ist x_{τ_i} mit TRUE belegt, und $y_{\tau_i} = \neg x_{\tau_i}$ hat den Wert FALSE. Für $q > 3$ wurde durch obige Konstruktion die Formel $(y_{\tau_1} \vee ... \vee y_{\tau_q})$ durch $q-2$ neue Formeln $(y_{\tau_1} \vee y_{\tau_2} \vee z_{G,1})$, $(\neg z_{G,1} \vee y_{\tau_3} \vee z_{G,2})$, ..., $(\neg z_{G,q-4} \vee y_{\tau_{q-2}} \vee z_{G,q-3})$, $(\neg z_{G,q-3} \vee y_{\tau_{q-1}} \vee y_{\tau_q})$ ersetzt. Es lässt sich zeigen, dass im vorliegenden Fall mindestens eine dieser neuen Formeln den Wert FALSE trägt: Haben nämlich alle neuen Formeln den Wert TRUE, wobei alle y_{τ_i} den Wert FALSE besitzen, dann müssten $z_{G,1}$, ..., $z_{G,q-3}$ mit TRUE belegt sein; die letzte Klausel $(\neg z_{G,q-3} \vee y_{\tau_{q-1}} \vee y_{\tau_q})$ hat dann jedoch nicht den Wert TRUE.

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 $\varepsilon = 1/\left((q-2)\cdot 2^{q+1}\right)$ folgt die Behauptung.

///