

Ausgewählte Kapitel der Theoretischen Informatik

Prof. Dr. rer. nat. Ulrich Hoffmann

Überarbeitete Auflage, Dezember 2008

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

Dieses Skript ist die überarbeitete Auflage der Veröffentlichung:
Hoffmann, U: Ausgewählte Kapitel der Theoretischen Informatik, FINAL 17:1, 2007, ISSN 0939-8821.

Gegenüber dieser Auflage wurden die Kapitel 4, 6 und 7 überarbeitet und ergänzt und einige Fehler berichtigt.

Inhaltsverzeichnis

Literaturauswahl	4
1 Einleitung	6
1.1 Einige grundlegende Bezeichnungen und Sachverhalte	7
1.2 Problemklassen.....	27
1.3 Ein intuitiver Algorithmusbegriff.....	32
2 Modelle der Berechenbarkeit und Komplexität von Berechnungen	35
2.1 Deterministische Turingmaschinen.....	35
2.2 Random Access Maschinen	57
2.3 Programmiersprachen.....	67
2.4 Universelle Turingmaschinen	79
2.5 Nichtdeterminismus	87
3 Grenzen der Berechenbarkeit	110
3.1 Jenseits der Berechenbarkeit	110
3.2 Rekursiv aufzählbare und entscheidbare Mengen.....	115
4 Elemente der Theorie Formaler Sprachen und der Automatentheorie	134
4.1 Grammatiken und formale Sprachen.....	134
4.2 Typ-0-Sprachen.....	137
4.3 Typ-1-Sprachen.....	139
4.4 Typ-2-Sprachen.....	151
4.5 Typ-3-Sprachen.....	174
4.6 Eigenschaften im tabellarischen Überblick.....	190
5 Praktische Berechenbarkeit	196
5.1 Der Zusammenhang zwischen Optimierungs- und Entscheidungsproblemen.....	201
5.2 Komplexitätsklassen.....	208
5.3 Die Klassen P und NP	212
5.4 NP-Vollständigkeit.....	227
5.5 Beispiele für den Nachweis der NP-Vollständigkeit.....	239
5.6 Bemerkungen zur Struktur von NP	256
6 Approximation von Optimierungsaufgaben	265
6.1 Absolut approximierbare Probleme	268
6.2 Relativ approximierbare Probleme.....	276
6.3 Polynomiell zeitbeschränkte und asymptotische Approximationsschemata.....	303
7 Weiterführende Konzepte	316
7.1 Beispiele wahrscheinlichkeitstheoretischer Ansätze.....	316
7.2 Modelle randomisierter Algorithmen.....	328

Literatúrauswahl

In die folgende Literatúrauswahl sind ausschließlich Lehrbücher und Monographien aufgenommen worden, die der weiterführenden Lektüre dienen können und die bei der Themenzusammenstellung verwendet wurden. Andere Veröffentlichungen sind teilweise im Text erwähnt.

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**, 2. Aufl., Oldenbourg, 2001.

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

Erk, K.; Priese, L.: **Theoretische Informatik**, 3. Aufl., Springer 2008.

Garey, M.R.; Johnson, D.: **Computers and Intractability, A Guide to the Theory of NP-Completeness**, Freeman, 1979.

Harary, F.: **Graphentheorie**, Oldenbourg, 1974.

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

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

Hoffmann, U.: **Übungen und Lösungen zur Mathematik für Wirtschaftsinformatiker**, FINAL 15:2, 2005.

Hofri, M.: **Probabilistic Analysis of Algorithms**, Springer 1987.

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

Hopcroft, J.E.; Ullman, J.D.: **Introduction to Automata Theory, Languages, and Computation**, Addison Wesley, 1979.

Hromkovič, J.: **Theoretische Informatik**, 3. Aufl., Teubner, 2007.

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**, 5. Aufl., Spektrum Akademischer Verlag, 2008.

Vossen, G.; Witt, K.-U.: **Grundkurs Theoretische Informatik**, 3. Aufl., Vieweg, 2004.

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

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

Yan, S.Y.: **Number Theory for Computing**, Springer, 2002.

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?
- Wo liegen die Ursachen für die hohe Komplexität von Lösungen vieler praktischer Problemstellungen?

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. Informatiker, die sich für eine tiefergehende Behandlung dieser Themen interessieren, werden auf die angegebene Literatur verwiesen. 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.

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),

\emptyset (**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. $\mathbf{P}(A) = \{ B \mid B \subseteq A \}$),

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

(**kartesisches Produkt** der Mengen A_1, A_2, \dots, A_n , d.h.

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

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

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) $A \cup \emptyset = A$, $A \cap \emptyset = \emptyset$.
- (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) $A \cup (B \cap C) = (A \cup B) \cap C$, $A \cap (B \cup C) = (A \cap B) \cup C$ (**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 $M_1 \cap M_2 = \emptyset$ 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 $\overline{(\overline{A^c})^c} = A$.
- (ix) Sind $A \subseteq C$ und $B \subseteq C$, so gelten
 $\overline{(A \cap B)^c} = \overline{A^c \cup B^c}$, $\overline{(A \cup B)^c} = \overline{A^c \cap B^c}$ (**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 Wahrheitstabellen 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) $A \cup \emptyset = A, A \cap \emptyset = \emptyset.$
- (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) $A \cup (B \cap C) = (A \cup B) \cap C,$
 $A \cap (B \cup C) = (A \cap B) \cup C$ **(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 $\overline{\overline{A^c}} = A.$

../..

<p>(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)</p> <p>(x) $(P \Rightarrow Q) \Leftrightarrow (\neg Q \Rightarrow \neg P)$</p>	<p>(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)</p> <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.</p>
---	--

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:

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

häufig auch in der Form

$$f: A \rightarrow B, f(a) = \dots$$

Die Angabe $f: A \rightarrow B$ 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 $f: A \rightarrow B$ 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 $f(a) = \dots$ beschreibt, wie diese Zuordnung für jedes Element $a \in A$ geschieht.

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

Eine Abbildungsvorschrift $f:A \rightarrow B$ 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 $f:A \rightarrow B$ 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 $f:A \rightarrow B$ heißt **bijektiv**, wenn sie injektiv und surjektiv ist. In

diesem Fall gibt es eine eindeutig bestimmte **Umkehrfunktion** $f^{-1}:B \rightarrow A$ 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 $\mathbf{N} = \{ 0, 1, 2, 3, 4, \dots \}$.

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 $a_1 a_2 \dots a_n$ 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 $a_1a_2 \dots a_n$ wird durch $|a_1a_2 \dots a_n| = n$ 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 $x = a_1a_2 \dots a_n$ und $y = b_1b_2 \dots b_m$ zwei Wörter aus Σ^* , so heißt das Wort $xy = a_1a_2 \dots a_nb_1b_2 \dots b_m$ 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:

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

$$(ii) \quad (L^*)^* = L^*$$

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

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

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

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

Beweis:

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

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

Zu (ii): Es sei $w \in (L^*)^*$. Dann ist $w = \varepsilon$, oder es ist $w \in (L^*)^n$ mit $n \geq 1$. Im ersten Fall ist (wegen $\{\varepsilon\} \subseteq L^*$) $w \in L^*$. Im zweiten Fall ist $w = w_1 \dots w_n$ mit $w_i \in L^*$ für $i = 1, \dots, n$. Jedes w_i besteht nur aus Buchstaben aus L , d.h. $w_i = a_{i,1} \dots a_{i,j_i}$ mit $a_{i,k} \in \Sigma$. 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 $w = a_1 \dots a_n$ für ein $n \geq 1$ und $a_i \in L$ für $i = 1, \dots, n$. Im ersten Fall sieht man direkt $w \in (L^*)^*$. Im zweiten Fall ist wegen $a_1 \dots a_n = (a_1 \dots a_n)^1$ und $a_1 \dots a_n \in L^*$: $w \in (L^*)^1$ und damit wegen $(L^*)^1 \subseteq (L^*)^*$ auch $w \in (L^*)^*$.

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 $w = w_1 \dots w_n$ mit $n \geq 1$ und $w_i \in L_1 \cup L_2$ für $i = 1, \dots, n$. 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 $w_1 \dots w_n \in (L_1^* \cdot L_2^*)^*$.

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 $w = a_1 \dots a_n$ mit $n \geq 1$ und $a_i \in L$ für $i = 1, \dots, n$. Es ist $w = a_1 \cdot a_2 \dots a_n = a_1 \dots a_{n-1} \cdot a_n$, 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 $f : A \rightarrow B$ gibt. Wegen der Existenz der bijektiven Umkehrfunktion g zu f ist diese Definition gleichbedeutend mit der Existenz einer bijektiven Abbildung $g : B \rightarrow A$.

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

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

Wie man leicht nachrechnet, handelt es sich hierbei um eine bijektive Abbildung, d.h. die Menge \mathbf{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 $f: \{0, \dots, n-1\} \rightarrow A$ gibt, d.h. man kann die Elemente in A mit den natürlichen Zahlen $0, \dots, n-1$ durchnummerieren: $A = \{f(0), \dots, f(n-1)\} = \{a_0, \dots, a_{n-1}\}$. 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 $f: B \rightarrow A$ 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 $\Sigma = \{a_0, \dots, a_{n-1}\}$ kann man die Wörter aus Σ^* (in **lexikographischer Reihenfolge**, **lexikalischer Reihenfolge**) gemäß folgender Tabelle notieren und durchnummerieren:

Nummer	Wort aus Σ^*	Bemerkung	Nummer	Wort aus Σ^*	Bemerkung
0	ε	Wort der Länge 0	$n + n^2 + 1$	$a_0 a_0 a_0$	Wörter der Länge 3: (Anzahl: n^3)
1	a_0	Wörter der Länge 1: (Anzahl: n)	$n + n^2 + 2$	$a_0 a_0 a_1$	
2	a_1		
...	...		$n^2 + 2n$	$a_0 a_0 a_{n-1}$	
n	a_{n-1}	Wörter der Länge 2: (Anzahl: n^2)	$n^2 + 2n + 1$	$a_0 a_1 a_0$	
$n + 1$	$a_0 a_0$		
$n + 2$	$a_0 a_1$		$n^2 + 3n$	$a_0 a_1 a_{n-1}$	
...	
$2n$	$a_0 a_{n-1}$	Wörter der Länge 3: (Anzahl: n^3)	$n + n^2 + n^3$	$a_{n-1} a_{n-1} a_{n-1}$	
$2n + 1$	$a_1 a_0$		
$2n + 2$	$a_1 a_1$		
...	...	Wörter der Länge 3: (Anzahl: n^3)	
$3n$	$a_1 a_{n-1}$		
...	
$n + n^2$	$a_{n-1} a_{n-1}$	

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 $\sum_{i=1}^k n^i = \frac{n(n^k - 1)}{n - 1}$. 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 ist¹.

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 $M = \{a_0, \dots, a_{n-1}\}$ eine endliche Menge mit n Elementen. Jede Teilmenge $N \subseteq M$ mit k Elementen, etwa $N = \{a_{i_0}, \dots, a_{i_{k-1}}\}$ kann als 0-1-Vektor der Länge n dargestellt werden. Dabei sind alle Komponenten dieses Vektors gleich 0 bis auf die Komponenten an den Positionen i_0, \dots, 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 überabzählbar.

Beweis:

Es wird die **Diagonalisierungstechnik** angewandt:

¹ Siehe: Hoffmann, U.: **Übungen und Lösungen zur Mathematik für Wirtschaftsinformatiker**, FINAL 15:2, 2005.

Es sei $f : \mathbf{N} \rightarrow M$ eine Abzählung von M , d.h. $M = \{ f(0), f(1), f(2), \dots \}$ mit $f(i) \neq f(j)$ für $i \neq j$. Angenommen, $\mathbf{P}(M)$ ist abzählbar, etwa mit Hilfe der bijektiven Abbildung $g : \mathbf{N} \rightarrow \mathbf{P}(M)$, d.h. $\mathbf{P}(M) = \{ g(0), g(1), g(2), \dots \}$. 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 Annahme, $\mathbf{P}(M)$ 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 $f : \mathbf{N} \rightarrow \mathbf{R}$ und $g : \mathbf{N} \rightarrow \mathbf{R}$ 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 $|f(n)| \leq c \cdot |g(n)|$ ist für jedes $n \in \mathbf{N}$ bis auf höchstens endlich viele Ausnahmen („...“, so dass $|f(n)| \leq c \cdot |g(n)|$ 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 $d = \text{const.}$ ist $d \cdot f(n) \in O(f(n))$

Es gelte $|f(n)| \leq c \cdot |g(n)|$ 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

$$O(O(f(n))) = \left\{ h \left| \begin{array}{l} h: \mathbf{N} \rightarrow \mathbf{R} \text{ und es gibt eine Funktion } g \in O(f(n)) \text{ und eine Konstante } c > 0 \\ \text{mit } |h(n)| \leq c \cdot |g(n)| \text{ f\"ur jedes } n \in \mathbf{N} \text{ bis auf h\"ochstens endlich viele Ausnahmen} \end{array} \right. \right\}$$

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:

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

$$O(f(n) \cdot g(n)) \subseteq \{ |f(n)| \} \cdot O(g(n)) = |f(n)| \cdot O(g(n)) \text{ (in vereinfachter Schreibweise),}$$

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

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

$$p(n) \in O(n^k) \text{ f\"ur } k \geq m,$$

$$\text{insbesondere } n^m \in O(n^k) \text{ f\"ur } k \geq m.$$

Beweis:

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

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

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

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

///

Aus der Analysis ist für $a > 1$ und $b > 1$ die Umrechnung $\log_a(n) = \frac{1}{\log_b(a)} \cdot \log_b(n)$ bekannt². 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 $a^{h(n)} = 2^{\log_2(a)h(n)}$ gilt

Satz 1.1-10:

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

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

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 $\log_a(n) \in O(\sqrt[m]{n})$.

Insgesamt ergibt sich damit

Satz 1.1-11:

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

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

Bei der Analyse des Laufzeitverhaltens von Algorithmen treten häufig Funktionen auf, die arithmetische Verknüpfungen von Logarithmusfunktionen, Polynomen und Exponentialfunk-

² Siehe: Hoffmann, U.: **Mathematik für Wirtschaftsinformatiker**, FINAL 15:1, 2005.

tionen sind. Man sagt, eine durch $f(n)$ gegebene Funktion hat **langsames 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 Wachstum

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

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

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) $xa_n \dots a_0$ 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 $xa_n \dots a_0$ mit $a_0 \in \{0, 1\}$, ..., $a_n \in \{0, 1\}$ die für den Booleschen Ausdruck stehende Variable x_i mit $i = \sum_{k=0}^n a_k \cdot 2^k$ eingesetzt.

Beispielsweise wird der Boolesche Ausdruck $((x_1 \wedge (\neg x_{101})) \vee ((\neg x_{11}) \wedge (x_{100} \vee x_{101}))) \vee 0$ 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

$$F = F_1 \wedge \dots \wedge F_m$$

und jedes F_j die Form

$$F_j = (y_{j_1} \vee \dots \vee y_{j_k})$$

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 $V = \{v_1, \dots, v_n\}$ von **Knoten** (vertices) und einer endlichen Menge $E = \{e_1, \dots, e_k\} \subseteq V \times V$ von **Kanten** (edges).

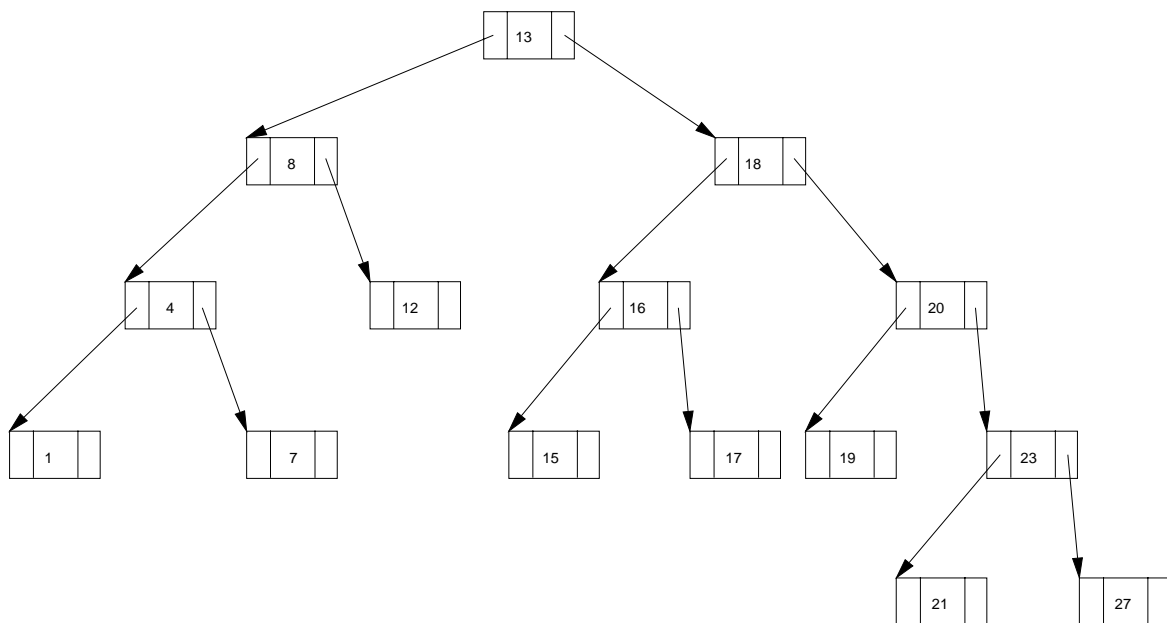
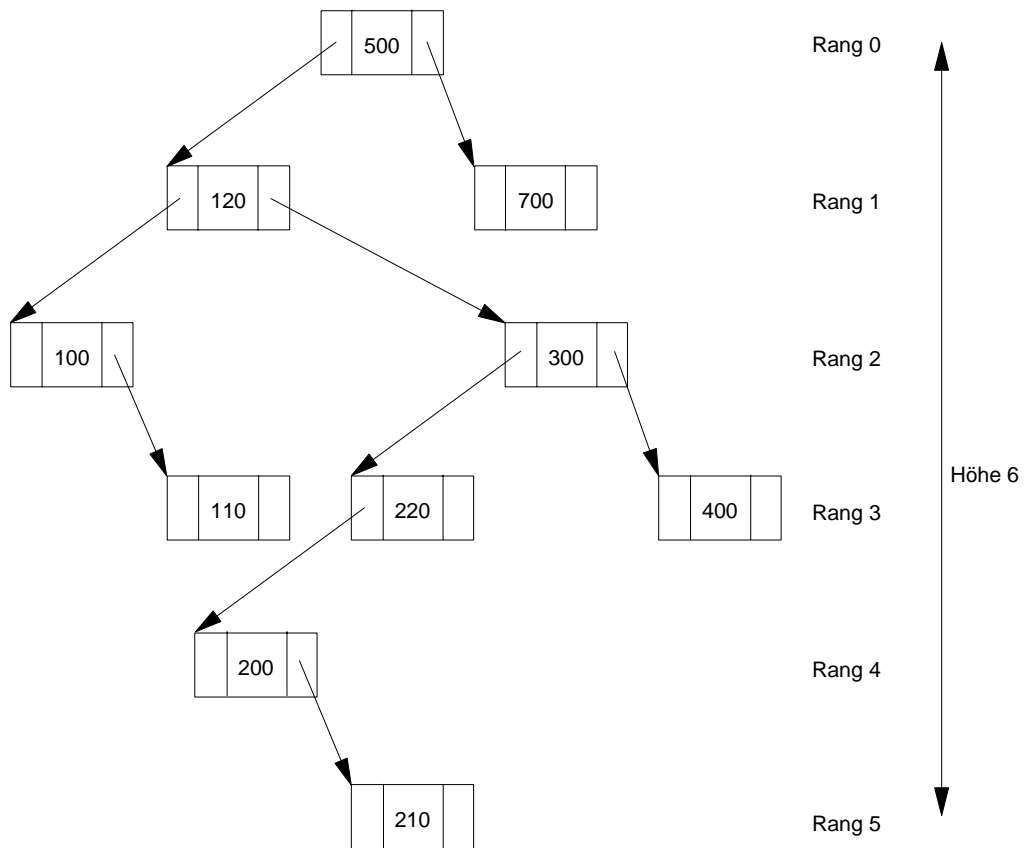
Die Kante $e = (v_i, v_j)$ läuft von v_i nach v_j (verbindet v_i mit v_j). Der Knoten v_i heißt **Anfangsknoten** der Kante $e = (v_i, v_j)$, der Knoten v_j **Endknoten** von $e = (v_i, v_j)$. Zu einem Knoten $v \in V$ heißt $pred(v) = \{v' \mid (v', v) \in E\}$ die **Menge der direkten Vorgänger** von 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 $V = E = \emptyset$ (**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 $((a_0, a_1), (a_1, a_2), \dots, (a_{m-1}, a_m))$ mit $r = a_0$, $v = a_m$ und $(a_{i-1}, a_i) \in E$ für $i = 1, \dots, m$. 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 so viele 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:³

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 $m < \log_2(n + 1) \leq m + 1$, 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 un spezifiziert 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.

³ Teil (v) wird in diesem Text nicht verwendet; der Beweis kann beispielsweise in Hoffmann: **Mathematik für Wirtschaftsinformatiker**, FINAL 15:1, 2005, nachgelesen werden.

Eine spezielle Problemstellung erhält man durch Konkretisierung einer Problem Instanz, 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 $V = \{v_1, \dots, v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \rightarrow \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht.

Lösung: Eine optimale Tour durch G .

Dabei ist eine **Tour** eine geschlossene Kantenfolge

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

in der jeder Knoten des Graphen (als Anfangsknoten einer Kante) genau einmal vorkommt. Die Kosten einer Tour $\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \dots, (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ 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 Durch-

laufen 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 $V = \{v_1, \dots, v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \rightarrow \mathbf{R}_{\geq 0}$ 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 $\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \dots, (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ 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 $V = \{v_1, \dots, v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \rightarrow \mathbf{R}_{\geq 0}$ 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 $\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \dots, (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ 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 $f: \Sigma_{\Pi}^* \rightarrow \Sigma'^*$.

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 $y \in \text{SOL}_{\Pi}(x)$ einen Wert $m_{\Pi}(x, y)$, den **Wert einer zulässigen Lösung**, zuordnet,

4. $\text{goal}_{\Pi} \in \{\min, \max\}$, je nachdem, ob es sich um ein Minimierungs- oder ein Maximierungsproblem handelt.

Lösung: $y^* \in \text{SOL}_{\Pi}(x)$ mit $m_{\Pi}(x, y^*) = \min\{m_{\Pi}(x, y) \mid y \in \text{SOL}_{\Pi}(x)\}$ bei einem Minimierungsproblem (d.h. $\text{goal}_{\Pi} = \min$) bzw. $m_{\Pi}(x, y^*) = \max\{m_{\Pi}(x, y) \mid y \in \text{SOL}_{\Pi}(x)\}$ bei einem Maximierungsproblem (d.h. $\text{goal}_{\Pi} = \max$).

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 $V = \{v_1, \dots, v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \rightarrow \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht

2. $\text{SOL}(G) = \{T \mid T = \langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \dots, (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$ 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 $T \in \text{SOL}(G)$, $T = \langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \dots, (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$, ist die Zielfunktion

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

4. $goal = \min$

Lösung: Eine Tour $T^* \in \text{SOL}(G)$, 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 $f: \Sigma_{\Pi}^* \rightarrow \Sigma'^*$) 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 elementaren 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 $i := \dots$ TO ... DO..., REPEAT ... UNTIL ...**) besitzen.

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

- Hält der Algorithmus immer bei einer gültigen Eingabe nach endlich vielen Schritten an?
- Berechnet der Algorithmus bei einer gültigen Eingabe eine korrekte Antwort?

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

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

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

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

- Gibt es zu einer Problemlösung eventuell ein „noch besseres“ Verfahren (mit weniger Rechenschritten, weniger Speicherplatzbedarf)? 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 $g : \mathbf{N} \rightarrow \{0, 1\}$, 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 $g : \mathbf{N} \rightarrow \{0,1\}$ ist daher eine Teilmenge von Σ^* und damit abzählbar (unendlich). Es gibt daher eine bijektive Funktion $f : \mathbf{N} \rightarrow B$, d.h. $B = \{f(0), f(1), f(2), \dots\}$. Der Wert $f(i)$ bezeichnet das Berechnungsverfahren in B mit der Nummer i . Die von $f(i)$ berechnete Funktion sei $f_i : \mathbf{N} \rightarrow \{0,1\}$.

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

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

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 $g : \mathbf{N} \rightarrow \{0,1\}$.

///

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 Automaten-theorie, 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

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

mit:

1. Q ist eine endliche nichtleere Menge: die **Zustandsmenge**
2. Σ ist eine endliche nichtleere Menge: das **Arbeitsalphabet**; Σ enthält alle Zeichen, die in Feldern der Bänder (siehe unten) stehen können
3. $I \subseteq \Sigma$ ist eine endliche nichtleere Menge: das **Eingabealphabet**; mit den Zeichen aus I werden die Wörter der Eingabe gebildet
4. $b \in \Sigma \setminus I$ ist das **Leerzeichen (Blankzeichen)**
5. $q_0 \in Q$ ist der **Anfangszustand (oder Startzustand)**
6. $q_{accept} \in Q$ ist der **akzeptierende Zustand (Endzustand)**
7. $\delta : (Q \setminus \{q_{accept}\}) \times \Sigma^k \rightarrow Q \times (\Sigma \times \{L, R, S\})^k$ ist eine partielle Funktion, die **Überföhrungsfunktion**; insbesondere ist $\delta(q, a_1, \dots, a_k)$ 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. Die Zellen eines Bandes sind von links nach rechts mit aufsteigenden Nummern beginnend mit 1 durchnummeriert. Die Zelle mit der Nummer j hat die rechte Nachbarzelle mit der Nummer $j + 1$ und bis auf die Zelle mit der Nummer 1 die linke Nachbarzelle mit der Nummer $j - 1$. Jede Zelle eines jeden Bands beinhaltet einen Buchstaben des Arbeitsalphabets. 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 übergelien oder auf der Zelle stehenbleiben. Welche Aktion der jeweilige Schreib/Lesekopf unternimmt, wird in der Überföhrungsfunktion 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 Zustandsüberföhrungen **getaktet** ablaufen und durch δ beschrieben werden:

Liest der Kopf des i -ten Bandes den Buchstaben $a_i \in \Sigma$ ($i = 1, \dots, k$), ist das Steuerwerk im Zustand q und ist

$$\delta(q, a_1, \dots, a_k) = (q', (a'_1, d_1), \dots, (a'_k, d_k)),$$

dann geht das Steuerwerk in einem Takt in den Zustand q' über, der i -te Kopf schreibt a'_i und geht zur linken Nachbarzelle für $d_i = L$, zur rechten Nachbarzelle für $d_i = R$ oder bleibt für $d_i = S$ über der Zelle stehen. Dabei ist zu beachten, dass die Überföhrungsfunktion korrekterweise so entworfen wird, dass bei ihrer Anwendung wöhrend einer Berechnung kein Kopf eines Bandes über das linke Bandende hinausläuft.

Ist $\delta(q, a_1, \dots, a_k)$ 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 $\delta(q, a_1, \dots, a_k)$ für $q = q_{\text{accept}}$ nicht definiert ist. Sie kann aber auch eventuell vorher in einem anderen Zustand anhalten (nämlich dann, wenn $\delta(q, a_1, \dots, a_k)$ 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 $q \neq q_{\text{accept}}$ kommt und $\delta(q, a_1, \dots, a_k) = (q, (a_1, S), \dots, (a_k, S))$ ist. Eine weitere Möglichkeit eines endlosen Weiterlaufens ergibt sich dann, wenn die Turingmaschine in einen Zustand $q \neq q_{\text{accept}}$ kommt, alle Köpfe über Zellen stehen, die das Leerzeichen b enthalten, rechts dieser Zellen auf allen Bändern nur noch Leerzeichen stehen und $\delta(q, b, \dots, b) = (q, (b, R), \dots, (b, R))$ ist.

Die Werte $\delta(q, a_1, \dots, a_k) = (q', (a'_1, d_1), \dots, (a'_k, d_k))$ der Überföhrungsfunktion 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'	a'_1, d_1	...	a'_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:

$$K = (q, (\alpha_1, i_1), \dots, (\alpha_k, i_k)) \text{ mit } q \in Q, \alpha_j \in \Sigma^*, i_j \geq 1 \text{ für } j = 1, \dots, k.$$

Diese Konfiguration K wird folgendermaßen interpretiert:

TM ist im Zustand q , das j -te Band (für $j = 1, \dots, k$) 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 -te Buchstaben 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

$$\delta(q, a_1, \dots, a_k) = (q', (a'_1, d_1), \dots, (a'_k, d_k)),$$

dann geht die *TM* in die **Folgekonfiguration** K' über, die durch

$$K' = (q', (\beta_1, i'_1), \dots, (\beta_k, i'_k))$$

definiert wird. Dabei entsteht β_j aus α_j durch Ersetzen von a_j durch a'_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 $m \in \mathbb{N}$, wenn K' aus K durch m Konfigurationsänderungen hervorgegangen ist, d.h. wenn es m Konfigurationen K_1, \dots, K_m gibt mit

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

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

heißt **Anfangskonfiguration mit Eingabewort** w . Da $w \in I^*$ ist und das Leerzeichen nicht zu I gehört, kann *TM* (bei entsprechender Definition der Überföhrungsfunktion) 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

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

heißt **akzeptierende Konfiguration (Endkonfiguration)**.

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

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

mit einer Endkonfiguration K_{accept} .

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

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

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 Überföhrungsfolge $K_0 \Rightarrow K'$ lässt sich unendlich lang fortsetzen, ohne dass eine Konfiguration erreicht wird, die den Endzustand enthält.

Eine 2-DTM Turingmaschine zur Akzeptanz von

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

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 Überföhrungsfunktion δ , die durch folgende Tabelle definiert ist:

momentaner Zustand	Symbol unter dem Schreib/Lesekopf auf Band		neuer Zustand	neues Symbol, Kopfbewegung auf Band	
	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 $\{0, 1\}$:

$$L(TM) = \{w \mid w \in \{0, 1\}^+ \text{ und } w = a_1 \dots a_{n-1} a_n = a_n a_{n-1} \dots a_1\} \cup \{\varepsilon\}.$$

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

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 Eingabewort 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 Überföhrungsfunktion 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. $\delta(q, \dots)$ 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 $\delta(q, \dots)$ 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** $f_{TM} : I^* \rightarrow \Sigma^*$, 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 $f : \begin{cases} \mathbf{N} & \rightarrow & \mathbf{N} \\ n & \rightarrow & n+1 \end{cases}$

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

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 $Q = \{q_0, \dots, q_{19}\}$, $\Sigma = \{0, 1, b, \#\}$, $I = \{0, 1\}$, $q_{accept} = q_{19}$.

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 $\#0w$, 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 Überföhrungsfunktion 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 $n+1$ markieren
	1	b	q_1	1, S	$\#, R$	
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 Eingabebandes auf 2. Band kopieren
	1	b	q_2	1, R	1, R	
	b	b	q_3	b, S	b, L	
q_3	b	0	q_4	b, S	1, S	1 addieren: anhängende 1'en invertieren, erste 0 von rechts invertieren
	b	1	q_3	b, S	0, L	
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 entfernt werden?
	b	1	q_{15}	b, S	1, S	
q_6	b	0	q_7	b, S	0, L	gelesenes Zeichen im Zustand merken
	b	1	q_8	b, S	1, L	
	b	b	q_{13}	b, S	b, L	
q_7	b	0	q_9	b, S	0, L	2. Kopf um 1 Position nach links setzen
	b	1	q_9	b, S	1, L	
q_8	b	0	q_{10}	b, S	0, L	2. Kopf um 1 Position nach links setzen
	b	1	q_{10}	b, S	1, L	

../..

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 Position nach rechts setzen
	b	1	q_{12}	b, S	1, R	
q_{12}	b	0	q_6	b, S	0, R	2. Kopf 1 weitere Position nach rechts setzen
	b	1	q_6	b, S	1, R	
q_{13}	b	0	q_{14}	b, S	b, L	letztes Zeichen löschen und nach links gehen
	b	1	q_{14}	b, S	b, L	
q_{14}	b	0	$q_{19} = q_{accept}$	b, S	b, S	letztes Zeichen löschen und STOP
	b	1	$q_{19} = q_{accept}$	b, S	b, S	
q_{15}	b	0	q_{16}	b, S	0, L	gelesenes Zeichen im Zustand merken
	b	1	q_{17}	b, S	1, L	
	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 Position nach rechts setzen
	b	1	q_{15}	b, S	1, R	

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

Satz 2.1-1:

- (i) Berechnet die Turingmaschine TM die partielle Funktion $f_{TM} : I^* \rightarrow \Sigma^*$, dann kann man eine Turingmaschine TM' konstruieren mit $L(TM') = \{ w\#y \mid y = f_{TM}(w) \}$. Hierbei ist $\#$ ein neues Symbol, das in Σ nicht vorkommt.
- (ii) Akzeptiert die Turingmaschine TM die Sprache $L(TM)$, dann lässt sich TM so modifizieren, dass sie die partielle Funktion $f_{TM} : I^* \rightarrow \Sigma^*$ berechnet, die definiert ist durch

$f_{TM}(w) = y$ genau dann, wenn gilt:

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

Aussage (i) lässt die Interpretation zu, dass die Verifikation eines Funktionsergebnisses genauso komplex ist wie die Berechnung des Funktionsergebnisses selbst. Aussage (ii) verbindet die Akzeptanz einer Sprache mit dem Berechnungsvorgang einer Funktion.

Beweis:

Beweisideen beider Teile des Satzes werden hier informell durch die Arbeitsweisen der beteiligten Turingmaschinen beschrieben:

Zu (i): Die Turingmaschine TM besitze k Bänder, wobei das erste Band ihr Eingabeband und das k -te Band ihr Ausgabeband ist. Die Turingmaschine TM' besitzt zwei Bänder mehr. Dabei ist das erste Band ihr Eingabeband; die Bänder mit den Nummern 2 bis $k + 1$ simulieren die Bänder von TM (dort mit den Nummern 1 bis k), d.h. das Band von TM' mit der Nummer 2 simuliert das erste Band, nämlich das Eingabeband, von TM , das Band mit der Nummer $k + 1$ simuliert das Ausgabeband von TM . Die Überföhrungsfunktion von TM' wird so entworfen, dass TM' folgendes Verhalten zeigt:

Bei Eingabe einer Zeichenkette $z \in (\Sigma \cup \{\#\})^*$ schreibt TM' das Zeichen $\#$ in die jeweils ersten Zellen der Bänder mit der Nummer 2 und $k + 2$; hierdurch kann später das jeweils linke Bandende erkannt werden. TM' kopiert alle Zeichen der Eingabe bis zum ersten Auftreten des Zeichens $\#$ auf das Band mit der Nummer 2, überspringt das Zeichen $\#$ in der Eingabe und kopiert die restlichen Zeichen der Eingabe auf das Band mit der Nummer $k + 2$. Hierbei wird angenommen, dass die Eingabe z die Form $z = w\#y$ mit $w \in \Sigma^*$ und $y \in \Sigma^*$ besitzt; andernfalls wird die Eingabe nicht akzeptiert. TM' setzt den Kopf des zweiten Bandes auf das erste Zeichen von w zurück; dabei wird die Tatsache genutzt, dass links von w das Begrenzungszeichen $\#$ steht und dadurch verhindert werden kann, dass der Kopf beim Zurücksetzen über das

linke Bandende hinausläuft. Nun geht TM' in einen Zustand, der dem Anfangszustand q_0 von TM entspricht. Die gegenwärtige Konfiguration, in der sich TM' befindet, lautet also:

$$K = \left(q_0, (w\#y, |w| + |y| + 2), \underbrace{(\#w, 2), (\varepsilon, 1), \dots, (\varepsilon, 1)}_{k \text{ Bänder}}, (\#y, |y| + 2) \right).$$

TM' simuliert mit den Bändern mit den Nummern 2 bis $k + 1$ das Verhalten von TM bei Eingabe von w ; das dem Wort w vorangestellte Zeichen $\#$ ist dabei ohne Belang. Falls dabei TM' in den (simulierten) Zustand q_{accept} von TM gelangt, enthält das Band mit der Nummer $k + 1$ den Funktionswert $f_{TM}(w)$. Eventuell steht der Kopf des Bandes innerhalb dieser Zeichenkette. Er wird daher direkt hinter das Ende der Zeichenkette verschoben. Nun vergleicht TM' von rechts nach links zeichenweise, ob die Zeichenkette y des Bandes mit der Nummer $k + 2$ mit $f_{TM}(w)$ übereinstimmt. Auch hierbei wird genutzt, dass das linke Bandende auf dem Band mit der Nummer $k + 2$ wegen des Zeichens $\#$, das der Zeichenkette y vorangestellt wurde, erkannt wird. Die Eingabe $w\#y$ wird von TM' genau dann akzeptiert, wenn $y = f_{TM}(w)$ ist.

Zu (ii): Die Turingmaschine TM wird so modifiziert, dass sie im Falle des Erreichens des akzeptierenden Zustands, den Kopf des Bandes mit der Nummer k noch direkt hinter die Bandinschrift setzt und erst dann akzeptiert, d.h. die Überföhrungsfunktion wird wie folgt erweitert:

$$\delta(q_{accept}, a_1, \dots, a_{k-1}, a_k) = (q_{accept}, (a_1, S), \dots, (a_{k-1}, S), (a_k, R)) \text{ und}$$

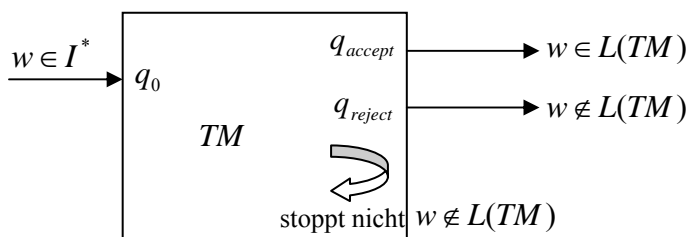
$$\delta(q_{accept}, a_1, \dots, a_{k-1}, b) = (q'_{accept}, (a_1, S), \dots, (a_{k-1}, S), (b, S)) \text{ mit } a_1 \in \Sigma, \dots, a_k \in \Sigma.$$

Hierbei ist q'_{accept} der (neue) akzeptierende Zustand der modifizierten Turingmaschine.

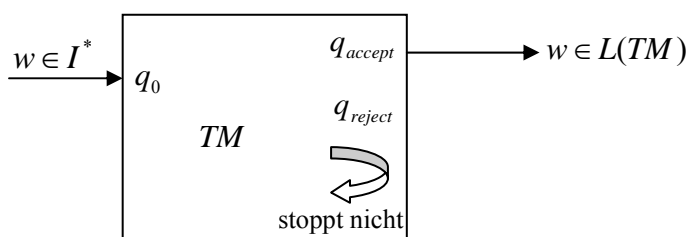
///

Die Überföhrungsfunktion 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 $(a_1, \dots, a_k) \in \Sigma^k$ der Wert $\delta(q, a_1, \dots, a_k)$ nicht definiert, so wird δ um den Eintrag $\delta(q, a_1, \dots, a_k) = (q_{reject}, (a_1, S), \dots, (a_k, S))$ 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 $\delta(q, \dots)$ bisher nicht definiert war (es ist dann $w \notin L(TM)$), so macht die modifizierte Turingmaschine noch eine Überföhrung, 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 Überfunktionsfunktion 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.



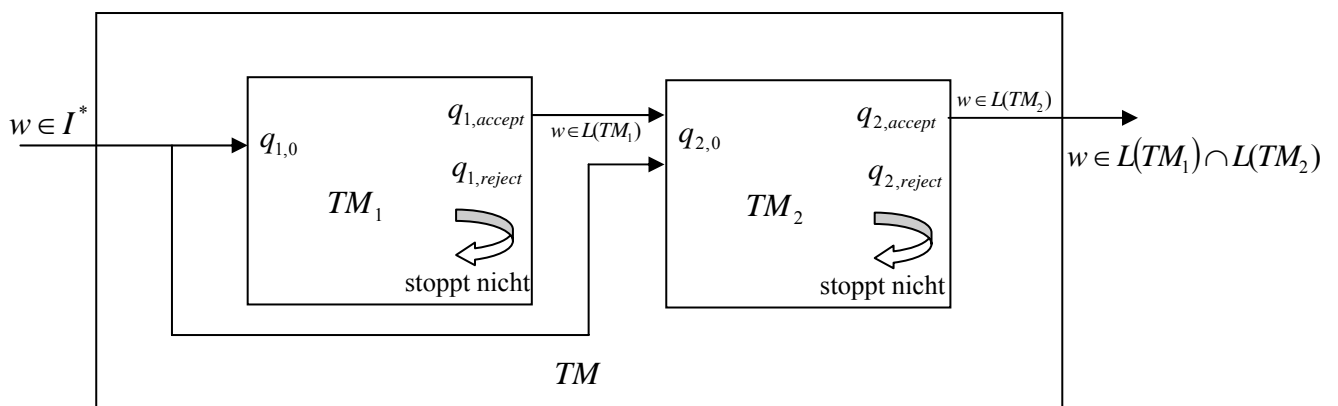
Da die Turingmaschine TM dazu verwendet wird, um festzustellen, ob ein Eingabewort $w \in I^*$ von ihr akzeptiert wird, kann man TM vereinfacht auch folgendermaßen darstellen:



Turingmaschinen können zu neuen Turingmaschinen zusammengesetzt werden: Es seien zwei Turingmaschinen $TM_1 = (Q_1, \Sigma_1, I, \delta_1, b, q_{1,0}, q_{1,accept})$ und $TM_2 = (Q_2, \Sigma_2, I, \delta_2, b, q_{2,0}, q_{2,accept})$ mit disjunkten Zustandsmengen ($Q_1 \cap Q_2 = \emptyset$) 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 Überföhrungsfunktion lautet:

Es ist $TM = (Q, \Sigma, I, \delta, b, q_{1,0}, q_{2,accept})$. Dabei ist

$Q = Q_1 \cup Q_2 \cup \{q_{copy}, q_{skip}\}$ die Zustandsmenge von TM mit zwei neuen Zuständen q_{copy} und q_{skip} , die nicht in $Q_1 \cup Q_2$ enthalten sind,

$\Sigma = \Sigma_1 \cup \Sigma_2 \cup \{\#\}$ das Arbeitsalphabet von TM mit einem neuen Symbol $\#$, das nicht in $\Sigma_1 \cup \Sigma_2$ enthalten ist,

$\delta : (Q \setminus \{q_{2,accept}\}) \times \Sigma^{k_1+k_2} \rightarrow Q \times (\Sigma \times \{L, R, S\})^{k_1+k_2}$ die Überföhrungsfunktion, die sich aus den Überföhrungsfunktionen δ_1 und δ_2 wie folgt ergibt:

Die Bänder von TM werden von 1 bis $k_1 + k_2$ nummeriert; 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 Überföhrungsfunktion δ von TM aufgenommen:

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

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

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

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

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

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

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

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

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 Überföhrungsfunktion δ lauten:

$$\delta \left(q_1, \underbrace{a_1, a_2, \dots, a_{k_1}}_{k_1}, \underbrace{a, b, \dots, b}_{k_2} \right) = \left(q'_1, \underbrace{(b_1, d_1), (b_2, d_2), \dots, (b_{k_1}, d_{k_1})}_{k_1}, \underbrace{(a, S), (b, S), \dots, (b, S)}_{k_2} \right)$$

für jeden Eintrag $\delta_1(q_1, a_1, a_2, \dots, a_{k_1}) = (q'_1, (b_1, d_1), (b_2, d_2), \dots, (b_{k_1}, d_{k_1}))$ in der Überföhrungsfunktion 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 Überföhrungsfunktion δ aufgenommen:

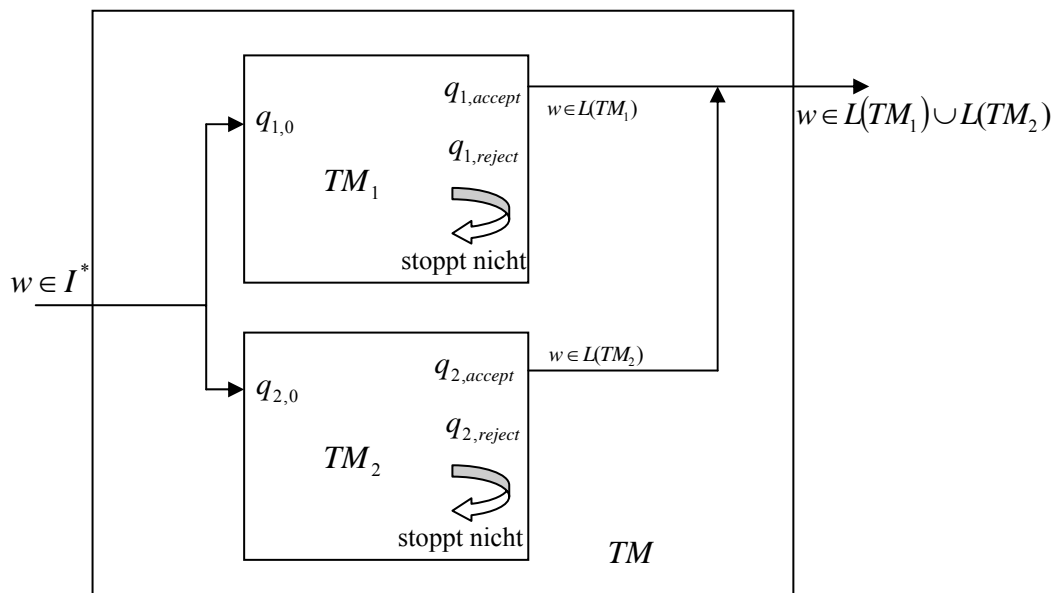
$$\delta \left(q_{1,accept}, \underbrace{a_1, a_2, \dots, a_{k_1}}_{k_1}, \underbrace{a, b, \dots, b}_{k_2} \right) = \left(q_{2,0}, \underbrace{(a_1, S), (a_2, S), \dots, (a_{k_1}, S)}_{k_1}, \underbrace{(a, S), (b, S), \dots, (b, S)}_{k_2} \right)$$

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

$$\delta \left(q_2, \underbrace{a_1, a_2, \dots, a_{k_1}}_{k_1}, \underbrace{g_1, g_2, \dots, g_{k_2}}_{k_2} \right) = \left(q'_2, \underbrace{(a_1, S), (a_2, S), \dots, (a_{k_1}, S)}_{k_1}, \underbrace{(h_1, d_1), (h_2, d_2), \dots, (h_{k_2}, d_{k_2})}_{k_2} \right)$$

für jeden Eintrag $\delta_2(q_2, g_1, g_2, \dots, g_{k_2}) = (q'_2, (h_1, d_1), (h_2, d_2), \dots, (h_{k_2}, d_{k_2}))$ in der Überföhrungsfunktion δ_2 von TM_2 und $a_1 \in \Sigma_1, \dots, a_{k_1} \in \Sigma_1$.

- 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: $L(TM) = L(TM_1) \cup L(TM_2)$.



Hier lautet die formale Notation von TM mit Hilfe der Überföhrungsfunktion:

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,

$\delta : (Q \setminus \{(q_{acc}, q_{acc})\}) \times \Sigma^{k_1+k_2} \rightarrow Q \times (\Sigma \times \{L, R, S\})^{k_1+k_2}$ die Überföhrungsfunktion, die sich aus den Überföhrungsfunktionen 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 $k_1 + k_2$ nummeriert; 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 Überföhrungsfunktion lauten jetzt:

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

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

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

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

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

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

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

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

für alle $a' \in I \cup \{b\}$.

Nun wird parallel das Verhalten von TM_1 und von TM_2 simuliert, wobei für die Simulation von TM_1 nur auf die Inhalte der Bänder mit den Nummern 1 bis k_1 (entsprechend den ursprünglichen Bändern von TM_1) und für die Simulation von TM_2 nur auf die Inhalte der Bänder mit den Nummern $k_1 + 1$ bis $k_1 + k_2$ (entsprechend den ursprünglichen Bändern von TM_2) zugegriffen wird. Die dafür erforderlichen Zeilen in der Überföhrungsfunktion δ lauten:

$$\delta \left((q_1, q_2), \underbrace{a_1, a_2, \dots, a_{k_1}}_{k_1}, \underbrace{g_1, g_2, \dots, g_{k_2}}_{k_2} \right) = \left((q'_1, q'_2), \underbrace{(b_1, d_1), (b_2, d_2), \dots, (b_{k_1}, d_{k_1})}_{k_1}, \underbrace{(h_1, e_1), (h_2, e_2), \dots, (h_{k_2}, e_{k_2})}_{k_2} \right),$$

falls $\delta_1(q_1, a_1, a_2, \dots, a_{k_1}) = (q'_1, (b_1, d_1), (b_2, d_2), \dots, (b_{k_1}, d_{k_1}))$ in der Überföhrungsfunktion δ_1 von TM_1 und $\delta_2(q_2, g_1, g_2, \dots, g_{k_2}) = (q'_2, (h_1, e_1), (h_2, e_2), \dots, (h_{k_2}, e_{k_2}))$ in der Überföhrungsfunktion 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 Überföhrungsfunktion δ aufgenommen:

$$\delta \left((q_{1,reject}, q_2), \underbrace{a_1, a_2, \dots, a_{k_1}}_{k_1}, \underbrace{g_1, g_2, \dots, g_{k_2}}_{k_2} \right) = \left((q_{1,reject}, q'_2), \underbrace{(a_1, S), (a_2, S), \dots, (a_{k_1}, S)}_{k_1}, \underbrace{(h_1, e_1), (h_2, e_2), \dots, (h_{k_2}, e_{k_2})}_{k_2} \right)$$

mit $a_1 \in \Sigma_1, \dots, a_{k_1} \in \Sigma_1$ und falls $\delta_2(q_2, g_1, g_2, \dots, g_{k_2}) = (q'_2, (h_1, e_1), (h_2, e_2), \dots, (h_{k_2}, e_{k_2}))$ in der Überföhrungsfunktion δ_2 von TM_2 ist und

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

$$= \left((q'_1, q_{2, \text{reject}}), \underbrace{(b_1, d_1), (b_2, d_2), \dots, (b_{k_1}, d_{k_1})}_{k_1}, \underbrace{(g_1, S), (g_2, S), \dots, (g_{k_2}, S)}_{k_2} \right)$$

für jeden Eintrag $\delta_1(q_1, a_1, a_2, \dots, a_{k_1}) = (q'_1, (b_1, d_1), (b_2, d_2), \dots, (b_{k_1}, d_{k_1}))$ in der Überföhrungsfunktion δ_1 von TM_1 und $g_1 \in \Sigma_2, \dots, g_{k_2} \in \Sigma_2$.

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

$$\begin{aligned} & \delta \left((q_{1, \text{accept}}, q_2), \underbrace{a_1, a_2, \dots, a_{k_1}}_{k_1}, \underbrace{g_1, g_2, \dots, g_{k_2}}_{k_2} \right) \\ &= \left((q_{\text{acc}}, q_{\text{acc}}), \underbrace{(a_1, S), (a_2, S), \dots, (a_{k_1}, S)}_{k_1}, \underbrace{(g_1, S), (g_2, S), \dots, (g_{k_2}, S)}_{k_2} \right) \text{ und} \\ & \delta \left((q_1, q_{2, \text{accept}}), \underbrace{a_1, a_2, \dots, a_{k_1}}_{k_1}, \underbrace{g_1, g_2, \dots, g_{k_2}}_{k_2} \right) \\ &= \left((q_{\text{acc}}, q_{\text{acc}}), \underbrace{(a_1, S), (a_2, S), \dots, (a_{k_1}, S)}_{k_1}, \underbrace{(g_1, S), (g_2, S), \dots, (g_{k_2}, S)}_{k_2} \right) \end{aligned}$$

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

- 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_i 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 wird dem Leser als Übung überlassen.

Man sagt, eine Klasse \mathbf{K} von Sprachen, die jeweils über demselben Alphabet definiert sind, ist **gegenüber einer Operation** \circ **abgeschlossen**, wenn 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 $TM = (Q, \Sigma, I, \delta, b, q_0, q_{accept})$ und eine Eingabe $w \in L(TM)$ gelte

$$(q_0, (w, 1), (\varepsilon, 1), \dots, (\varepsilon, 1)) \Rightarrow^m K_{accept}$$

mit einer Endkonfiguration $K_{accept} = (q_{accept}, (\alpha_1, i_1), \dots, (\alpha_k, i_k))$. Dann wird durch $t_{TM}(w) = m$ eine partielle Funktion $t_{TM} : \Sigma^* \rightarrow \mathbf{N}$ 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 $T_{TM} : \mathbf{N} \rightarrow \mathbf{N}$ mit

$$T_{TM}(n) = \max\{t_{TM}(w) \mid w \in \Sigma^* \text{ und } |w| \leq n\}.$$

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 $f : \mathbf{N} \rightarrow \mathbf{N}$, 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 $|w| \leq n$ 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 $f: \begin{cases} \mathbf{N} & \rightarrow \mathbf{N} \\ n & \rightarrow n+1 \end{cases}$ hat folgende Zeitkomplexität: Der Wert n wird in Binärdarstellung auf das Eingabeband gegeben. Die Eingabe $w = \text{bin}(n)$ belegt daher für $n \geq 1$ $|w| = |\text{bin}(n)| = \lfloor \log_2(n) \rfloor + 1$ 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 $SUM: \begin{cases} \mathbf{N} \times \mathbf{N} & \rightarrow \mathbf{N} \\ (n, m) & \rightarrow n + m \end{cases}$ 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 $w = \text{bin}(n)\#\text{bin}(m)$. Dann werden die Zahlen n und m stellengerecht

addiert. Auch für die arithmetischen Funktionen $DIF: \begin{cases} \mathbf{N} \times \mathbf{N} & \rightarrow \mathbf{N} \\ (n, m) & \rightarrow \begin{cases} n - m & \text{für } n \geq m, \\ 0 & \text{für } n < m \end{cases} \end{cases}$,

$MULT: \begin{cases} \mathbf{N} \times \mathbf{N} & \rightarrow \mathbf{N} \\ (n, m) & \rightarrow n \cdot m \end{cases}$ und $DIV: \begin{cases} \mathbf{N} \times \mathbf{N}_{>0} & \rightarrow \mathbf{N} \\ (n, m) & \rightarrow \lfloor n/m \rfloor \end{cases}$ lassen sich entsprechende Tu-

ringmaschinen 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 $w = \text{bin}(n)\#\text{bin}(m)$, und die Zeitkomplexitäten geben an, wie viele Bitoperationen zur Berechnung der jeweiligen Funktionen erforderlich sind. Die Ergebnisse sind in folgendem Satz zusammengefasst; auf den Beweis wird hier verzichtet, da in Kapitel 2.3 im Programmiersprachenkonzept die entsprechenden Ergebnisse im Detail behandelt werden.

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 $\lfloor n \rfloor \geq \lfloor m \rfloor$, $k = size(n) \geq size(m)$, $k \in O(\log(n))$. 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 $O((\log(n))^2)$ 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 $O((\log(n))^2)$ 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 $O(T_{TM}^2(n))$ 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 $f: \mathbf{Z}^n \rightarrow \mathbf{Z}^m$ 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 $f: \Sigma^* \rightarrow \Sigma'^*$ 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 Überföhrungsfunktion besteht, entspricht die Programmierung einer RAM eher der Programmierung eines Computers auf Maschinensprachebene.

Eine RAM hat folgende Bestandteile:

- **Eingabeband:** eine Folge von n Zellen, die als Eingabe n ganze (positive oder negative) Zahlen x_1, \dots, x_n enthalten und von einem Lesekopf 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 in jede Zelle eine 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, \dots 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** $c: \mathbf{N} \rightarrow \mathbf{Z}$ verwendet. $c(i)$ gibt zu jedem Zeitpunkt des Programmlaufs den Inhalt des Registers r_i an. Zu Beginn des Programmlaufs wird jedes Register mit dem Wert 0 initialisiert, d.h. es ist $c(i) = 0$ für jedes $i \in \mathbf{N}$.

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

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

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

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

../..

SUB i	$c(0) := c(0) - i$
SUB $c(i)$ für $i \geq 0$	$c(0) := c(0) - c(i)$
SUB $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := c(0) - c(c(i))$
MULT i	$c(0) := c(0) * i$
MULT $c(i)$ für $i \geq 0$	$c(0) := c(0) * c(i)$
MULT $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := c(0) * c(c(i))$
DIV i für $i \neq 0$	$c(0) := \lfloor c(0) / i \rfloor$
DIV $c(i)$ für $i \geq 0$	$c(0) := \lfloor c(0) / c(i) \rfloor$, falls $c(i) \neq 0$ ist, sonst stoppt die RAM
DIV $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := \lfloor c(0) / c(c(i)) \rfloor$, falls $c(c(i)) \neq 0$ ist, sonst stoppt die RAM
READ $c(i)$ für $i \geq 0$	$c(i) :=$ momentaner Eingabewert, und der Lesekopf rückt eine Zelle nach rechts
READ $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(c(i)) :=$ momentaner Eingabewert, und der Lesekopf rückt eine Zelle nach rechts
WRITE i	i wird auf das Ausgabeband gedruckt, und der Schreibkopf rückt um eine Zelle nach rechts
WRITE $c(i)$ für $i \geq 0$	$c(i)$ wird auf das Ausgabeband gedruckt, und der Schreibkopf rückt um eine Zelle nach rechts
WRITE $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(c(i))$ wird auf das Ausgabeband gedruckt, und der Schreibkopf rückt um eine Zelle nach rechts
JUMP m	Der Befehlszähler wird mit der Nummer der Anweisung geladen, die die Marke m trägt (nächste auszuführende Anweisung)
JGTZ m	Der Befehlszähler wird mit der Nummer der Anweisung geladen, die die Marke m trägt (nächste auszuführende Anweisung), falls $c(0) > 0$ ist, sonst wird der Inhalt des Befehlszählers um 1 erhöht
JZERO m	Der Befehlszähler wird mit der Nummer der Anweisung geladen, die die Marke m trägt (nächste auszuführende Anweisung), falls $c(0) = 0$ ist, sonst wird der Inhalt des Befehlszählers um 1 erhöht
HALT	Die RAM stoppt

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

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

$$f_{RAM}(x_1, \dots, x_n) = (y_1, \dots, y_m).$$

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

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 <i>Write</i> (0)
3	JGTZ pos	
4	WRITE 0	
5	JUMP endif	
pos	LOAD $c(1)$	
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	
endif	WRITE $c(2)$	<i>Write</i> (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 $1, 2, \dots, |\Sigma|$ 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, \dots, x_n des Wortes entsprechenden ($w = x_1 \dots x_n$). 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 $c(1)$	<i>Read</i> (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	<i>Read</i> (x)
endwhile	LOAD $c(2)$ JZERO output HALT	IF $d = 0$ THEN <i>Write</i> (1)
output	WRITE 1 HALT	

Auch beim RAM-Modell interessieren **Zeit- und Speicherplatzkomplexität 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, \dots, x_n und durchlaufe bis zum Erreichen der HALT-Anweisung m viele Anweisungen (einschließlich der HALT-Anweisung). Dann wird durch $t_{RAM}(x_1, \dots, x_n) = m$ eine partielle Funktion $t_{RAM} : \mathbf{Z}^* \rightarrow \mathbf{N}$ definiert. Hält RAM bei Eingabe von x_1, \dots, x_n nicht an, dann ist $t_{RAM}(x_1, \dots, x_n)$ 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 $T_{RAM} : \mathbf{N} \rightarrow \mathbf{N}$ mit

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

Entsprechend wird die **Speicherplatzkomplexität** 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	$c(0) := c(0) * i$	$l(c(0)) + l(i)$
MULT $c(i)$ für $i \geq 0$	$c(0) := c(0) * c(i)$	$l(c(0)) + l(i) + l(c(i))$
MULT $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := c(0) * c(c(i))$	$l(c(0)) + l(i) + l(c(i)) + l(c(c(i)))$
DIV i für $i \neq 0$	$c(0) := \lfloor c(0) / i \rfloor$	$l(c(0)) + l(i)$
DIV $c(i)$ für $i \geq 0$	$c(0) := \lfloor c(0) / c(i) \rfloor$	$l(c(0)) + l(i) + l(c(i))$
DIV $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(0) := \lfloor c(0) / c(c(i)) \rfloor$	$l(c(0)) + l(i) + l(c(i)) + l(c(c(i)))$
READ $c(i)$ für $i \geq 0$	$c(i) := \text{Eingabewert}$	$l(\text{Eingabewert}) + l(i)$
READ $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	$c(c(i)) := \text{Eingabewert}$	$l(\text{Eingabewert}) + l(c(i))$

...

WRITE i	Ausgabe von i	$l(i)$
WRITE $c(i)$ für $i \geq 0$	Ausgabe von $c(i)$	$l(i) + l(c(i))$
WRITE $c(c(i))$ für $i \geq 0$ und $c(i) \geq 0$	Ausgabe von $c(c(i))$	$l(i) + l(c(i)) + l(c(c(i)))$
JUMP m	Befehlszähler := m	1
JGTZ m	Befehlszähler := m , falls $c(0) > 0$ ist	$l(c(0))$
JZERO m	Befehlszähler := m , falls $c(0) = 0$ ist	$l(c(0))$
HALT	Die RAM stoppt	1

Die **Zeitkosten eines RAM-Programms** bei gegebenen Eingabewerten **unter dem logarithmischen Kostenkriterium** ist gleich der Summe der Kosten der abzuarbeitenden Anweisungen. Entsprechend sind die **Speicherplatzkosten 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

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

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

Kosten des RAM-Programms P zur Akzeptanz von

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

	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 $1, \dots, |\Sigma|$ 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, \dots, r_c als zusätzliche Arbeitsregister für RAM reserviert werden. Unter diesen Arbeitsregistern werden k Register, etwa r_1, \dots, 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 2.2-1:⁴

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 Aus-

⁴ Auf den Beweis wird hier verzichtet; er kann nachgelesen werden in Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.: **The Design and Analysis of Computer Algorithms**, Addison-Wesley, 1974.

führung 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.

Simulierte Maschine B	simulierende Maschine A		
	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. Man kann sehr einfache Programmiersprachen definieren, 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 \mathbf{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** *bedin-*

gung **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 Speicherplatzkomplexität fragen.

Ein Programm *PROG* habe die Eingabe $x = [x_1, \dots, x_n]$, 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 $y = [y_1, \dots, y_m]$, die sich aus m Teilen zusammensetzt. Beispielsweise kann y das Ergebnis der Berechnung einer Funktion f sein, d.h. $[y_1, \dots, y_m] = f(x_1, \dots, x_n)$. Oder *PROG* soll entscheiden, ob die Eingabe x eine spezifizierte Eigenschaft besitzt; dann ist $y \in \{\text{TRUE}, \text{FALSE}\}$. In der Regel wird die Zeit- und Speicherplatzkomplexität von *PROG* in Abhängigkeit von der Eingabe $x = [x_1, \dots, x_n]$ 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 Speicherplatzkomplexität 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 Speicherplatzkomplexität nur von der *Anzahl* n der Komponenten der Eingabe $x = [x_1, \dots, x_n]$ 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 Speicherplatzkomplexität 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 Speicherplatzkomplexität 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 Speicherplatzkomplexität nicht nur von der Anzahl n der Komponenten der Eingabe $x = [x_1, \dots, x_n]$ abhängt, sondern wie im Beispiel der Prozedur `zweier_potenz` von den Zahlenwerten x_1, \dots, x_n selbst, wird folgender Ansatz gewählt. Der gleiche Ansatz ist angebracht, wenn arithmetische Operationen im Spiel sind, deren Anzahl die Zeit- und Speicherplatzkomplexität 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 Komple-

mentdarstellung 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 $x = [x_1, \dots, x_n]$ ist $size(x) = \sum_{i=1}^n (size(x_i) + 1) + 1$.

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, \dots, x_n umfassenden eckigen Klammern und Trennsymbolen zwischen den Werten x_1, \dots, x_n . Häufig wird auch als Größe der Eingabe der Wert $n \cdot (size(\max\{x_1, \dots, x_n\}) + 1) + 1$ verwendet. Dieser Maßstab ist wohl etwas gröber als $size(x)$. Für Analysezwecke eignet er sich jedoch, da in einer Komplexitätsbetrachtung berücksichtigt werden muss, dass es Eingaben x gibt, deren Zahlen in der Folge x_1, \dots, x_n zusammen genau $n \cdot size(\max\{x_1, \dots, x_n\})$ viele Bits belegen. Es kann dann die Abschätzung $size(x) \in O(n \cdot size(\max\{x_1, \dots, x_n\}))$ verwendet werden.

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**) **Speicherplatzkomplexität** 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 Turingmaschine in Abhängigkeit von der Länge der Eingabe, d.h. der Anzahl benötigter Bits, gezählt wird.

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 $x \geq y$, $k = size(x)$ 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 $x = [x_{k-1}x_{k-2} \dots x_1x_0]_2$ bzw. $y = [y_{l-1}y_{l-2} \dots y_1y_0]_2$.

Addition $z := x + y$

Für $l \leq k$ wird y durch Voranstellung von $k-l$ führende binäre Nullen auf dieselbe Länge wie x gebracht, so dass $y = [y_{k-1}y_{k-2} \dots y_1y_0]_2$ ist. Das Ergebnis z besitzt eventuell eine Binärstelle mehr als x : $z = [z_kz_{k-1} \dots z_1z_0]_2$. 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 $u = [u_ku_{k-1} \dots u_1u_0]$ die Bitfolge der Überträge aus der jeweils vorherigen Bitaddition bezeichnet, so ist

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

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:

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

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

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 : $z = [z_{2k-1}z_{2k-2} \dots z_1z_0]_2$. Gemäß der „Schulmethode“ wird folgender (Pseudo-) Code ausgeführt:

```

z := 0;           { Anweisung 1 }
FOR i := l-1 DOWNT0 0 DO   { Anweisung 2 }
  BEGIN
    SHL (z, 1);         { Anweisung 3 }
    IF yi = 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 $O(\text{size}(z))$ entsteht. Wegen $\text{size}(z) \leq 2k$ und $l \leq k$ werden zur Multiplikation $O(k^2) = O((\log(x))^2)$ viele Bitoperationen ausgeführt.

Ein schnelleres Multiplikationsverfahren ist als Karatsuba-Methode bekannt. Zur Vereinfachung der Darstellung sei k eine Zweierpotenz, d.h. $k = 2^m$ mit $m \in \mathbf{N}$.

Es ist $x = [x_{k-1}x_{k-2} \dots x_1x_0]_2 = [x_{k-1}x_{k-2} \dots x_{k/2+1}x_{k/2}]_2 \cdot 2^{k/2} + [x_{k/2-1}x_{k/2-2} \dots x_1x_0]_2 = p \cdot 2^{k/2} + q$ mit den $k/2$ -stelligen Binärzahlen $p = [x_{k-1}x_{k-2} \dots x_{k/2+1}x_{k/2}]_2$ und $q = [x_{k/2-1}x_{k/2-2} \dots x_1x_0]_2$. Entsprechend ist

$y = [y_{k-1}y_{k-2} \dots y_1y_0]_2 = [y_{k-1}y_{k-2} \dots y_{k/2+1}y_{k/2}]_2 \cdot 2^{k/2} + [y_{k/2-1}y_{k/2-2} \dots y_1y_0]_2 = r \cdot 2^{k/2} + s$ mit den $k/2$ -stelligen Binärzahlen $r = [y_{k-1}y_{k-2} \dots y_{k/2+1}y_{k/2}]_2$ und $s = [y_{k/2-1}y_{k/2-2} \dots y_1y_0]_2$. 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 Bitoperatio-

nen) 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

$$M(k) \leq 3 \cdot M(k/2) + c \cdot k \text{ mit einer Konstanten } c > 0.$$

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

$$M(2^m) \leq 3 \cdot M(2^{m-1}) + c \cdot 2^m \text{ für } m \geq 1 \text{ und } M(1) = 1.$$

Die allgemeine Form dieser Rekursionsgleichung lautet

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

Mit $l = m$ ergibt sich

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

mit einer Konstanten $d > 0$. Insgesamt:

$$M(k) = M(2^m) \leq d \cdot (3^m - 2^m) = d \cdot ((2^m)^{\log_2(3)} - 2^m) = d \cdot (k^{1,58\dots} - k), \text{ d.h. } M(k) \in O(k^{1,58\dots}).$$

Gegenüber der Schulmethode zur Multiplikation mit der Komplexität $O(k^2)$ 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 $O(c(\varepsilon) \cdot k^{1+\varepsilon})$ 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 praktischem 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 Multiplikationsalgorithmus⁵ benötigt unter Einsatz einer Variante der diskreten Fouriertransformation eine Anzahl an Bitmultiplikationen der Ordnung $O(k \cdot \log(k) \cdot \log(\log(k)))$.

⁵ Siehe: Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.: **The Design and Analysis of Computer Algorithms**, Addison-Wesley, 1974.

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 $O(k) = O(\text{size}(x))$ begrenzt ist. Insgesamt ist die Anzahl der Bitoperationen nach diesem Verfahren von der Ordnung $O\left(k \cdot \frac{x}{y}\right)$, d.h. im ungünstigsten Fall, nämlich für kleine Werte von y , exponentiell von der Ordnung $O(k \cdot 2^k)$.

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	$16 + 2^2 = 20$	
			10	1		
			20	2		
		17	5	0	$20 + 2^1 = 22$	
			10	1		
		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, \dots, i_t . Dann gilt:

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

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

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

...

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

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

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

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 zurückführen⁶, 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.

⁶ 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 $(a_n)_{n \in \mathbb{N}}$ 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 $y = [y_{l-1}y_{l-2} \dots y_1y_0]_2$ ist

$$x^y = x^{y_{l-1}2^{l-1} + y_{l-2}2^{l-2} + \dots + y_12 + y_0} = \prod_{i=0}^{l-1} x^{y_i 2^i} = \prod_{i=0}^{l-1} (x^{2^i})^{y_i}.$$

Man könnte zur Berechnung von x^y zunächst l viele Zahlen e_i ($i = l-1, \dots, 1, 0$) 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 zusammen-

multipliziert. 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 $i = l-1, \dots, 1, 0$ 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}
4	x	0
3	x^2	1
2	x^4	0
1	x^5	1
	x^{10}	
	x^{20}	
	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 $c := c * c \pmod{n}$; bzw. $c := c * x \pmod{n}$; 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.

In obiger Pascal-Prozedur `zweier_potenz` lässt sich die Zeitkomplexität nun wie folgt abschätzen: Für die Eingabe n und $k = \text{size}(n)$ gilt 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 $O(2^{2^i})$, 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

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

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 $k = \text{size}(n)$ ist. Dieser Wert gibt die Realität exakt wieder.

2.4 Universelle Turingmaschinen

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

Diese Art der Universalität ist auch im Turingmaschinen-Modell möglich. Im folgenden wird dazu eine **universelle Turingmaschine** UTM definiert. Diese erhält als Eingabe die Beschreibung einer Turingmaschine TM (ein Problemlösungsalgorithmus) und einen Eingabedatensatz 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 $TM = (Q, \Sigma, I, \delta, b, q_0, q_{accept})$ kann als endliches Wort über $\{0, 1\}$ kodiert werden. Dazu wird angegeben, wie eine mögliche **Kodierung einer Turingmaschine** aussehen kann (die hier vorgestellte Kodierung ist natürlich nur eine von vielen möglichen):

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

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

$$w_0 = \# \text{bin}(Q) \# \text{bin}(\Sigma) \# \text{bin}(k) \# \# .$$

Hierbei bezeichnet wieder $\text{bin}(n)$ die Binärdarstellung von n .

Die Überföhrungsfunktion δ habe d viele Zeilen. Die t -te Zeile laute:

$$\delta(q_i, a_{i_1}, \dots, a_{i_k}) = (q_j, (a_{j_1}, d_1), \dots, (a_{j_k}, d_k)).$$

Es sind (für $l = 1, \dots, k$ und $m = 1, \dots, k$) $a_{i_l} \in \Sigma$, $a_{j_m} \in \Sigma$ und $d_m \in \{L, R, S\}$. Diese Zeile wird zunäcst als Zeichenkette

$$w_t = (\text{bin}(i) \# \text{bin}(i_1) \# \dots \# \text{bin}(i_k)) (\text{bin}(j) \# (\text{bin}(j_1) \# d_1) \# \dots \# (\text{bin}(j_k) \# d_k)) \#$$

kodiert. Diese Zeichenkette ist ein Wort über dem Alphabet $\{0, 1, (,), \#, L, R, S\}$. Zu beachten ist, dass w_t sich öffnende und schließende Klammern enthält; der Ausdruck $\text{bin}(i)$ enthält keine Klammern, sondern ist eine Zeichenkette über $\{0, 1\}$.

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

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

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

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:

Überföhrungsfunktion $\delta(q_i, a_{i_1}, \dots, a_{i_k})$ $= (q_j, (a_{j_1}, d_1), \dots, (a_{j_k}, d_k))$	Zwischenkodierung $w_i = (bin(i)\#bin(i_1)\#\dots\#bin(i_k))$ $(bin(j)\#(bin(j_1)\#d_1)\#\dots\#(bin(j_k)\#d_k))$; entsprechend dem Wort über $\{0, 1\}$
$\delta(q_0, 1) = (q_1, (0, R))$	$w_1 = (0\#1)(1\#(0\#R))\#$ 0100001000010111010001100010000100110011011100
$\delta(q_1, 0) = (q_0, (1, R))$	$w_2 = (1\#0)(0\#(1\#R))\#$ 010001100000011010000100010001100110011011100
$\delta(q_1, 1) = (q_2, (0, R))$	$w_3 = (1\#1)(10\#(0\#R))\#$ 0100011000010111010001000100010000100110011011100
$\delta(q_1, b) = (q_1, (1, L))$	$w_4 = (0\#11)(1\#(1\#L))\#$ 010001100001001011010001100010001100101011011100

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

Die Turingmaschine TM ist also kodiert durch $code(TM) =$
 100001001100001001100110010001000010000101101000110001000010011001101110
 001000110000001101000010001000110011001101110001000110000101101000100010
 0010000100110011011100010001100001001011010001100010001100101011011100.

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

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

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

Andererseits kann man einem Wort $w \in \{0,1\}^*$ „ansetzen“, 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 Überföhrungsfunktion kodiert (dazu ist eine Reihe von Bedingungen zu prüfen, etwa die der Klammersetzung entsprechende Kodierung, die korrekte Angabe der Stellenzahl der Überföhrungsfunktion, die korrekte Verwendung von Zustandsnummern und Buchstabenummern des Arbeitsalphabets in der Überföhrungsfunktion, die Tatsache, dass die Überföhrungsfunktion 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

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

Falls ein Wort $w \in \{0,1\}^*$ eine Turingmaschine TM kodiert, d.h. $w = code(TM)$, dann bezeichnet man mit $TM = code^{-1}(w)$ **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) $VERIFIZIERE_TM(w) = \text{TRUE}$
- (ii) die Menge $\left\{ x \mid \begin{array}{l} x \in \{0,1\}^* \text{ und } x \text{ liegt in der lexikographischen Ordnung vor } w \\ \text{und } VERIFIZIERE_TM(x) = \text{TRUE} \end{array} \right\}$ 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'; {  $x \in \{0,1\}^*$  }
  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, $VERIFIZIERE_TM(w) = \text{TRUE}$ ist (d.h. $w = \text{code}(TM)$ 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 $VERIFIZIERE_TM(w) = \text{TRUE}$ 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 stoppt⁷.

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 $K_0 = \left(q_0, \underbrace{(u, 1), (\varepsilon, 1), \dots, (\varepsilon, 1)}_k \right)$. Hierbei kann ein ähnlicher Umsetzungsmechanismus wie zur Erzeugung der Kodierung einer Turingmaschine verwendet werden.

Eine Konfiguration von K_w der Form $K = (q_t, (\alpha_1, i_1), \dots, (\alpha_k, i_k))$, wobei q_t der t -te Zustand von K_w und $\alpha_j \in \Sigma^*$, $i_j \geq 1$ für $j = 1, \dots, k$ ist, wird zunächst (gedanklich) umgesetzt in $(bin(t)\#(\beta_1\#bin(i_1))\#\dots\#(\beta_k\#bin(i_k)))$. Hierbei erhält man β_j aus α_j (für $j = 1, \dots, k$), 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 $\beta_j = 1\#1\#1\#1\#1\#$. Die Zeichenfolge $(bin(t)\#(\beta_1\#bin(i_1))\#\dots\#(\beta_k\#bin(i_k)))$ wird mittels anfangs angegebener Tabelle in eine Folge über $\{0, 1\}$ umgesetzt.

⁷ Ist UTM eine k -DTM mit der Überföhrungsfunktion 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 $\delta_{UTM}(q, a_1, \dots, a_k) = (q, (a_1, S), \dots, (a_k, S))$ für $a_1 \in \Sigma_{UTM}, \dots, a_k \in \Sigma_{UTM}$. 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:

01000010001000100010000100010000110000110000100010010000101110001000010010001011100010000100100010010111000100001001000010111011.

Im Endstück v des Wortes $w = 100v_1100v_2100v_3100100v$ ist die Überföhrungsfunktion 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 Überföhrungsfunktion schrittweise ändert. Auf dem Anfangsstück $010m100$ 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, \dots, a_k , durch den eindeutigen Wert $\delta(q, a_1, \dots, a_k)$ der Überföhrungsfunktion 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 eingeföhrt.

Eine **nichtdeterministische k -Band-Turingmaschine (k -NDTM) TM** ist definiert durch

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

mit:

1. Q ist eine endliche nichtleere Menge: die **Zustandsmenge**
2. Σ ist eine endliche nichtleere Menge: das **Arbeitsalphabet**
3. $I \subseteq \Sigma$ ist eine endliche nichtleere Menge: das **Eingabealphabet**
4. $b \in \Sigma \setminus I$ ist das **Leerzeichen**
5. $q_0 \in Q$ ist der **Anfangszustand** (oder **Startzustand**)
6. $q_{accept} \in Q$ ist der **akzeptierende Zustand (Endzustand)**
7. $\delta : (Q \setminus \{q_{accept}\}) \times \Sigma^k \rightarrow \mathbf{P}(Q \times (\Sigma \times \{L, R, S\})^k)$ ist eine partielle Funktion, die **Überföhrungsfunktion**; insbesondere ist $\delta(q, a_1, \dots, a_k)$ 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 Überföhrungsfunktion δ . Die Überföhrungsfunktion ordnet nun jedem Argument (q, a_1, \dots, a_k) nicht einen einzigen (eindeutigen) Wert $(q', (a'_1, d_1), \dots, (a'_k, d_k))$, sondern eine *endliche Menge von Werten* der Form $\{(q_1, (b_{11}, d_{11}), \dots, (b_{1k}, d_{1k})), \dots, (q_t, (b_{t1}, d_{t1}), \dots, (b_{tk}, d_{tk}))\}$ 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:

$$K = (q, (\alpha_1, i_1), \dots, (\alpha_k, i_k)) \text{ mit } q \in Q, \alpha_j \in \Sigma^*, i_j \geq 1 \text{ für } j = 1, \dots, k.$$

TM startet wie im deterministischen Fall im Anfangszustand mit einer **Anfangskonfiguration** $K_0 = (q_0, (w, 1), (\varepsilon, 1), \dots, (\varepsilon, 1))$. Ist TM in eine Konfiguration $K = (q, (\alpha_1, i_1), \dots, (\alpha_k, i_k))$ gekommen, ist a_j der i_j -te Buchstabe von α_j bzw. $a_j = b$ für $i_j \geq |\alpha_j| + 1$, und ist $\delta(q, a_1, \dots, a_k)$ definiert, dann besteht $\delta(q, a_1, \dots, a_k)$ aus einer endlichen Menge von Werten der Form $(q', (b_1, d_1), \dots, (b_k, d_k))$, d.h.

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

Als **Folgekonfiguration** von K wird eine der t möglichen Konfigurationen

$$K_1 = (q_1, (\beta_{11}, i'_{11}), \dots, (\beta_{1k}, i'_{1k})), \dots, K_t = (q_t, (\beta_{t1}, i'_{t1}), \dots, (\beta_{tk}, i'_{tk}))$$

genommen. Dabei entsteht K_i für $i = 1, \dots, t$ aus K dadurch, dass man wie im deterministischen Fall q durch q_i und a_1, \dots, a_k durch b_{i1}, \dots, b_{ik} ersetzt und die Köpfe so bewegt, wie es

$(q_i, (b_{i1}, d_{i1}), \dots, (b_{ik}, d_{ik}))$ in $\delta(q, a_1, \dots, a_k)$ 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

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

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, \dots, K_m gibt mit

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

Für $m = 0$ ist dabei $K = K'$.

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

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

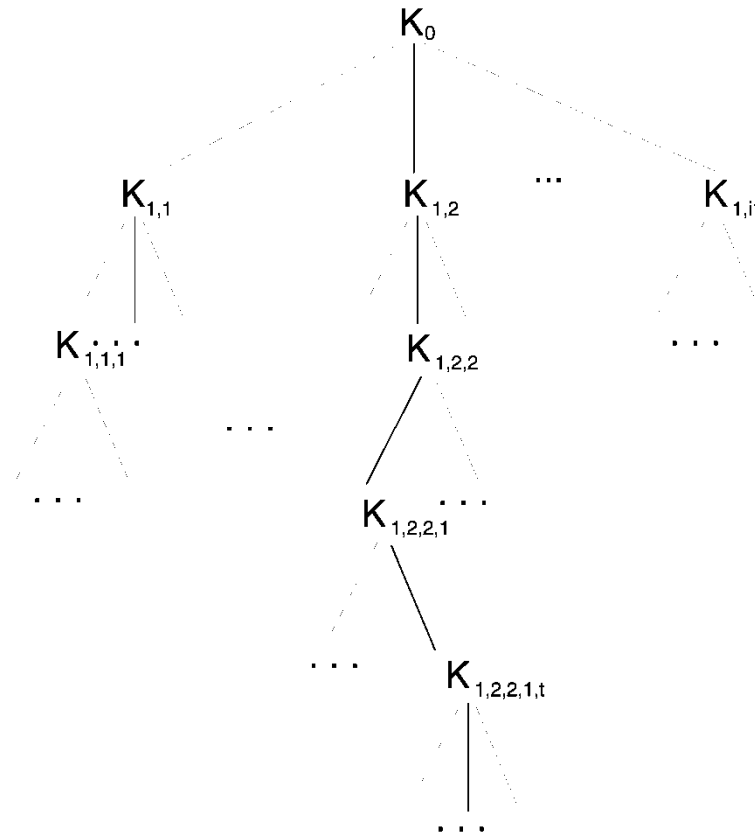
Wie im deterministischen Fall kann man die Überföhrungsfunktion δ modifizieren, indem man die Zustandsmenge Q um einen neuen Zustand q_{reject} erweitert und δ um entsprechende Zeilen ergnzt (siehe Kapitel 2.1), so dass alle Berechnungen, die in einem Zustand $q \neq q_{accept}$ enden, f ur die $\delta(q, a_1, \dots, a_k)$ mit $(a_1, \dots, a_k) \in \Sigma^k$ bisher nicht definiert ist, um eine  berf ohrung in den Zustand q_{reject} fortgesetzt werden k onnen. F ur q_{reject} ist δ nicht definiert.

Im *deterministischen* Fall kann man sich den Ablauf einer Berechnung als eine Folge

$$K_0 \Rightarrow \dots \Rightarrow K \Rightarrow K' \Rightarrow \dots$$

vorstellen, wobei jeder Schritt $K \Rightarrow K'$ eindeutig durch die  berf ohrungsfunktion 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 $K_0 = (q_0, (w, 1), (\varepsilon, 1), \dots, (\varepsilon, 1))$ mit einem Wort $w \in I^*$ markiert. Ist die Konfiguration $K = (q, (\alpha_1, i_1), \dots, (\alpha_k, i_k))$ die Markierung eines Knotens und kann hier der Eintrag $\delta(q, a_1, \dots, a_k)$ der  berf ohrungsfunktion angewendet werden, etwa

$\delta(q, a_1, \dots, a_k) = \{(q_1, (b_{11}, d_{11}), \dots, (b_{1k}, d_{1k})), \dots, (q_t, (b_{t1}, d_{t1}), \dots, (b_{tk}, d_{tk}))\}$, dann enthält dieser Knoten des Berechnungsbaums t direkte Nachfolger, die mit den sich ergebenden Nachfolgekonfigurationen $K_1 = (q_1, (\beta_{11}, i'_{11}), \dots, (\beta_{1k}, i'_{1k})), \dots, K_t = (q_t, (\beta_{t1}, i'_{t1}), \dots, (\beta_{tk}, i'_{tk}))$ markiert sind. Führt einer der Pfade von einer Anfangskonfiguration K_0 zu einer Endkonfiguration $K_{accept} = (q_{accept}, (\alpha_1, i_1), \dots, (\alpha_k, i_k))$, so wird das in der Anfangskonfiguration stehende Wort w akzeptiert.



Das Partitionenproblem mit ganzzahligen Eingabewerten

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

I ist eine Menge von n natürlichen Zahlen.

Lösung: Entscheidung „ja“, falls es eine Teilmenge $J \subseteq \{1, \dots, n\}$ 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 $I = \{a_1, \dots, a_n\}$ 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 $I = \{a_1, a_2, \dots, a_n\}$ wird dabei als Wort $w = 10^{a_1}10^{a_2} \dots 10^{a_n}$ kodiert.

Die 3-NDTM $TM = (\{q_0, \dots, q_5\}, \{0, 1, b, \$\}, \{0, 1\}, \delta, b, q_0, q_5)$ 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 Überföhrungsfunktion 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 kann so entworfen werden, dass sie bei Eingabe von w zunächst an der Position des Zeichens $\#$ feststellt, wo y

beginnt, und dann x und y buchstabenweise auf einen Unterschied hin vergleicht (falls w kein oder mehr als ein Zeichen $\#$ enthält, wird w nicht akzeptiert). Eine nichtdeterministische Turingmaschine NTM könnte 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 $x = x_1 \dots x_n$ und $y = y_1 \dots y_m$, 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 Überföhrungsfunktion zur nichtdeterministischen Erzeugung von z den Wert

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

Hierbei ist 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 beendet.

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 Überföhrungsfunktion einer nichtdeterministischen Turingmaschine TM deterministische und nichtdeterministische Teile, d.h. Einträge der Form $\delta(q, a_1, \dots, a_k) = \{ (q_1, (b_{11}, d_{11}), \dots, (b_{1k}, d_{1k})), \dots, (q_t, (b_{t1}, d_{t1}), \dots, (b_{tk}, d_{tk})) \}$ mit $t = 1$ (deterministisch) und Teile mit $t > 1$ (nichtdeterministisch). Soll in einer Konfiguration der Eintrag $\delta(q, a_1, \dots, a_k)$ 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 $(q_i, (b_{i1}, d_{i1}), \dots, (b_{ik}, d_{ik}))$ mit $1 \leq i \leq t$ ausgewählt, um die Nachfolgekonfiguration zu bilden. Dabei wird in dieser Situation ein „richtiger“ Eintrag $(q_i, (b_{i1}, d_{i1}), \dots, (b_{ik}, d_{ik}))$ 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 $(q_i, (b_{i1}, d_{i1}), \dots, (b_{ik}, d_{ik}))$ 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 $(q_i, (b_{i1}, d_{i1}), \dots, (b_{ik}, d_{ik}))$ 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)⁸. 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 Speicherplatzkomplexität (im schlechtesten Fall, worst case) einer nichtdeterministischen Turingmaschine wird ähnlich wie im deterministischen Fall definiert:

Für eine k -NDTM $TM = (Q, \Sigma, I, \delta, b, q_0, q_{accept})$ und eine Eingabe $w \in L(TM)$ gelte

$$(q_0, (w, 1), (\varepsilon, 1), \dots, (\varepsilon, 1)) \Rightarrow^m K_{accept}$$

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 $t_{TM} : \Sigma^* \rightarrow \mathbf{N}$ 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 $T_{TM} : \mathbf{N} \rightarrow \mathbf{N}$ mit

$$T_{TM}(n) = \max\{t_{TM}(w) \mid w \in \Sigma^* \text{ und } |w| \leq n\}.$$

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

⁸ Diese „externe Instanz“ wird hier nicht als Orakel bezeichnet, um keine Verwechslung mit Orakel-Turingmaschinen hervorzurufen.

Entsprechend kann man die **Speicherplatzkomplexität** 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 $|w| \leq n$ *mindestens* erreicht.

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

Eine Funktion $T : \mathbf{N} \rightarrow \mathbf{N}$ 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 Konfigura-

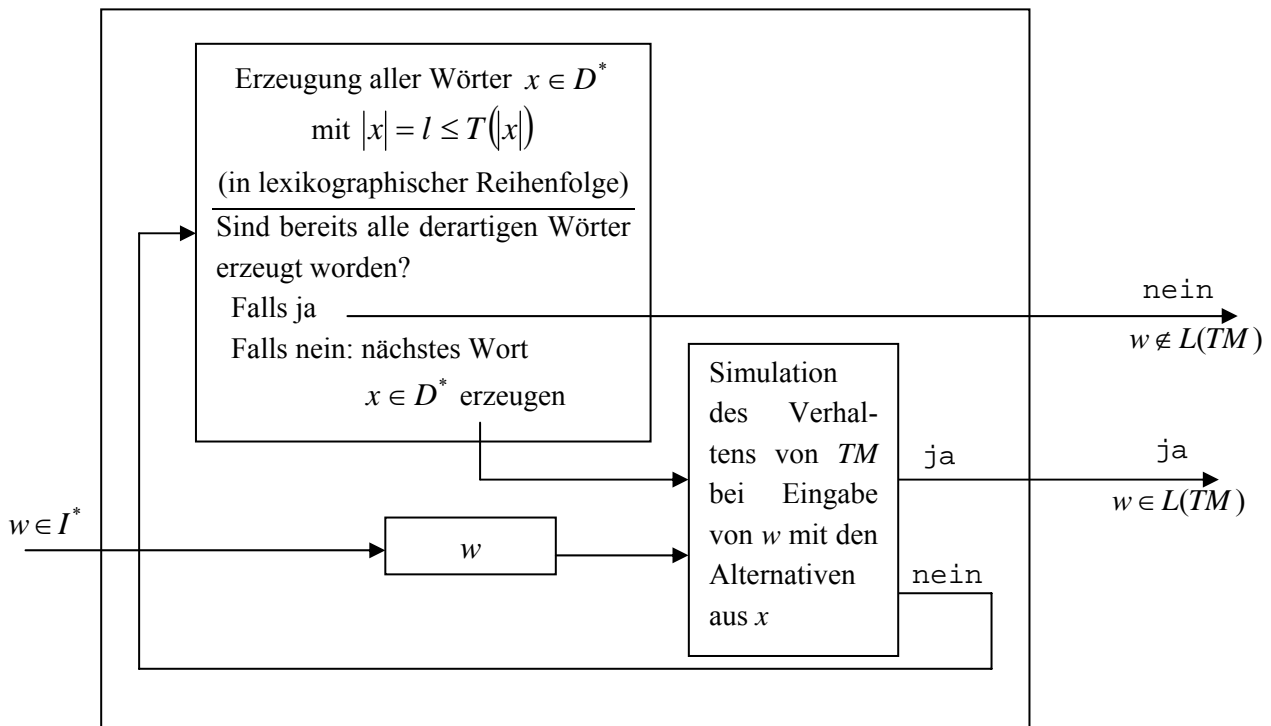
tionen, die von einer Anfangskonfiguration K_0 mit w zu einer akzeptierenden Konfiguration K_{accept} führt und deren Länge $\leq T(n)$ ist.

Jede Zeile der Tabelle, mit der die Überföhrungsfunktion δ von TM gegeben wird (Überföhrungstabelle), hat die Form

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

(hier stehen t „Teilzeilen“). Wenn in einer Konfigurationenfolge $\delta(q, a_1, \dots, a_k)$ angewendet wird, gibt es t mögliche Folgekonfigurationen. Der maximale Wert t an Teilzeilen in einer Zeile der Überföhrungstabelle sei $d \geq 2$. In einer Berechnung von TM gibt es für eine Konfiguration maximal d mögliche Folgekonfigurationen. Es sei $D = \{1, \dots, d\}$. Dann kann man eine von TM ausgeführte Berechnung bzw. die dabei durchlaufene Konfigurationsfolge der Länge l durch ein Wort $x = d_1 \dots d_l$ über D beschreiben: d_i gibt an, dass in der i -ten Überföhrung die d_i -te Alternative in der entsprechenden Zeile der Überföhrungstabelle 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 $|x| = l \leq T(n)$, $x = d_1 \dots d_l$, 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 deterministisch 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 Überföhrung die d_i -te Alternative in der entsprechenden Zeile der Überföhrungstabelle verwendet wird, falls es eine derartige Alternative überhaupt gibt. Falls die Simulation auf den akzeptierenden Zustand führt, wird das Wort w akzeptiert, ansonsten wird das nächste Wort $x \in D^*$ erzeugt und die Simulation erneut ausgeführt.



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

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

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

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

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

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

///

Diese deterministische Simulation der k -NDTM TM erfolgt also zum Preis einer exponentiellen Steigerung des Laufzeitverhaltens, denn $O(T(n) \cdot d^{T(n)}) \subseteq O(C^{T(n)})$ mit einer Konstanten $C > 0$; denn es gilt $c \cdot T(n) \cdot d^{T(n)} \leq c \cdot d^{T(n)} \cdot d^{T(n)} = c \cdot d^{2T(n)} = c \cdot (d^2)^{T(n)}$, 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 $T : \mathbf{N} \rightarrow \mathbf{N}$ eine zeitkonstruierbare Funktion mit $T(n) \geq n$. Ferner seien 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

$$L = \left\{ w \mid w \in \Sigma^* \text{ und es gibt } x \in \{0,1\}^* \text{ mit } |x| \leq c \cdot T(|w|) \text{ und } (w, x) \in L(TM') \right\}$$

(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 $T_1 : \mathbf{N} \rightarrow \mathbf{N}$ benötigt. Dann gibt es eine $(T(n) + T_1(n))$ -zeitbeschränkte k -NDTM TM mit $L(TM) = \left\{ w \mid w \in \Sigma^* \text{ und es gibt } y \in \Sigma_0^* \text{ mit } |y| \leq c \cdot T(|w|) \text{ und } (w, y) \in L' \right\}$.

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 $D = \{1, \dots, d\}$ 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.

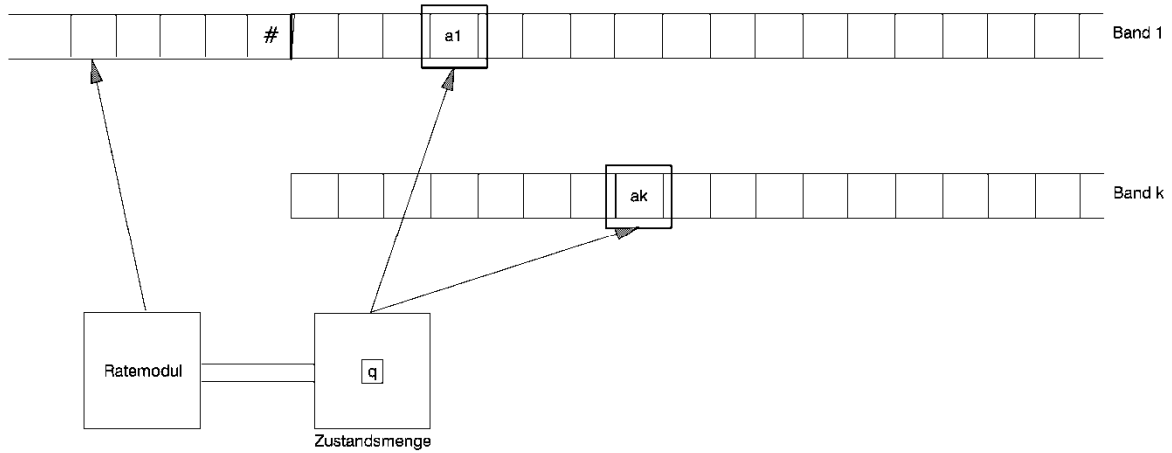
Zu (ii): Die im Satz anzugebende nichtdeterministische Turingmaschine TM wird durch ihr Verhalten informell beschrieben:

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 Schritten eine „Zusatzinformation“ $y \in \Sigma_0^*$ erzeugt wird, um dann die Zugehörigkeit des Paares (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 Überföhrungsfunktion ist eine partielle Abbildung $\delta : (Q \setminus \{q_{accept}\}) \times \Sigma^k \rightarrow \mathbf{P}(Q \times (\Sigma \times \{L, R, S\})^k)$, 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 $0, -1, -2, \dots$ 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 nicht-positiven 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 Zellnummer 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 $S : \mathbf{N} \rightarrow \mathbf{N}$ heißt **speicherplatzkonstruierbar (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 speicherplatzkonstruierbaren Speicherplatzkomplexität $S(n)$, dann gibt es eine k' -DTM TM' mit einer Speicherplatzkomplexität der Ordnung $O(S^2(n))$ und $L(TM') = L(TM)$.

Beweis:

Es sei $TM = (Q, \Sigma, I, \delta, b, q_0, q_{accept})$ eine k -NDTM mit einer Speicherplatzkonstruierbaren 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 $O(S^2(n))$ beschränkt ist. Es sei $K = (q, (\alpha_1, i_1), \dots, (\alpha_k, i_k))$ eine Konfiguration in einer Berechnung von TM bei Eingabe von w . Wegen $|\alpha_j| \leq S(n)$ und $1 \leq i_j \leq S(n)$ für $j = 1, \dots, k$ ist die Anzahl verschiedener Konfigurationen durch $|Q| \cdot |\Sigma|^{k \cdot S(n)} \cdot (S(n))^k \leq c^{S(n)}$ 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 $TM = (Q, \Sigma, I, \delta, b, q_0, q_{accept})$ mit einer Speicherplatzkonstruierbaren 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 κ_1 und κ_2 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  $\Rightarrow$  K2) OR (K1 = K2)
        THEN resultat := TRUE;
        END
    ELSE BEGIN
        Für jede Konfiguration K3 =  $(q, (\alpha_1, i_1), \dots, (\alpha_k, i_k))$ 
        mit  $|\alpha_j| \leq S(n)$  und  $1 \leq i_j \leq S(n)$  DO
        BEGIN
            resultat := test (K1, K3, (i+1) DIV 2)
            AND
            test (K3, K2, i DIV 2);
            IF resultat = TRUE THEN Break;
        END;
    END;
    test := resultat;
END { test };

BEGIN { space_Simulation }
K0 :=  $(q_0, (w, 1), (\varepsilon, 1), \dots, (\varepsilon, 1))$ 
OK := FALSE;
FOR_EACH Endkonfiguration Kf =  $(q_f, (\alpha_1, |\alpha_1|+1), \dots, (\alpha_k, |\alpha_k|+1))$  mit  $|\alpha_j| \leq S(n)$ 
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 Wert des dritten Parameters 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.

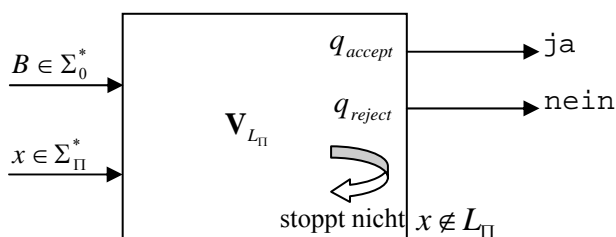
///

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 für ein Entscheidungsproblem unter Einsatz eines Verifizierers** führt. Dabei ist ein **Verifizierer für das Entscheidungsproblem** Π über einem Alphabet Σ_{Π} ein deterministischer Algorithmus $V_{L_{\Pi}}$, der eine Eingabe $x \in \Sigma_{\Pi}^*$ und eine **Zusatzinformation** (einen **Beweis**, ein **Zertifikat**) $B \in \Sigma_0^*$ lesen kann und die Frage „ $x \in L_{\Pi}$?“ 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 das 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 $V_{L_{\Pi}}$ für das Entscheidungsproblem Π** mit der Menge $L_{\Pi} \subseteq \Sigma_{\Pi}^*$ hat die Schnittstellen und die Form

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

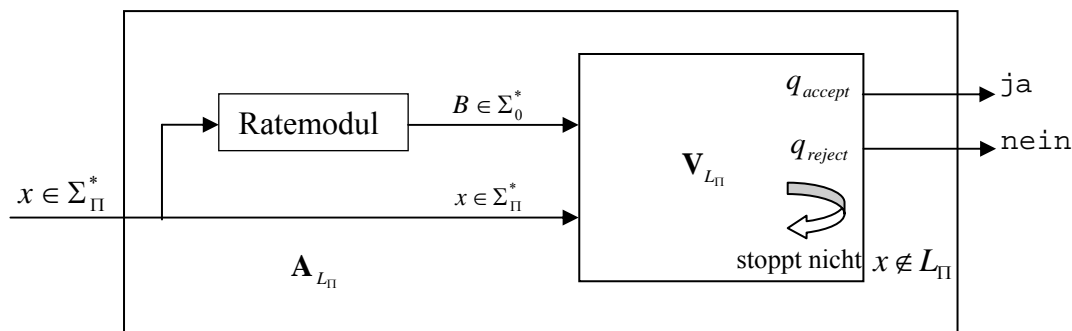
Ausgabe: ja (accept), falls die in B enthaltene Information die Eigenschaft $x \in L_{\Pi}$ belegt
 nein (reject), falls die in B enthaltene Information die Eigenschaft $x \in L_{\Pi}$ nicht belegt
 falls $V_{L_{\Pi}}$ bei einer Eingabe $x \in \Sigma_{\Pi}^*$ nicht stoppt, gilt $x \notin L_{\Pi}$



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

Damit ergibt sich:

Ein **nichtdeterministischer Algorithmus** A_{L_Π} 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 V_{L_Π} bei Eingabe von x und B_x mit $V_{L_\Pi}(x, B_x) = \text{ja}$ stoppt;

$x \notin L_\Pi$ genau dann, wenn für jeden Beweis $B \in \Sigma_0^*$ gilt: entweder stoppt V_{L_Π} bei Eingabe von x und B nicht, oder V_{L_Π} stoppt mit $V_{L_\Pi}(x, B) = \text{nein}$.

Die **Laufzeit eines nichtdeterministischen Algorithmus** bei Eingabe von $x \in \Sigma_\Pi^*$ mit $|x| = n$ ist die Anzahl der Schritte, einschließlich der nichtdeterministischen Ratephase, in der der Beweis erzeugt, d.h. zeichenweise niedergeschrieben wird, bis der Algorithmus zur Entscheidung kommt. Die Laufzeit wird in Abhängigkeit von $\text{size}(x) = |x| = n$ angegeben.

Das Partitionenproblem mit ganzzahligen Eingabewerten

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

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

Lösung: Entscheidung „ja“, falls es eine Teilmenge $J \subseteq \{1, \dots, n\}$ 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  $\mathbf{A}_{PARTITION}(I)$ : ...;

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

BEGIN {  $\mathbf{A}_{PARTITION}(I: \dots)$  }
  { 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  $\mathbf{A}_{PARTITION}$  := 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  $\mathbf{A}_{PARTITION}$  := ja
      ELSE  $\mathbf{A}_{PARTITION}$  := nein;
    END;
  END {  $\mathbf{A}_{PARTITION}(I)$  };

```

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

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

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 $i = 1, \dots, n$ 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 beispielsweise 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 wird 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. Das Ratemodul kann, falls F erfüllbar ist, in der Tat eine 0-1-Folge erzeugen, die einer erfüllenden Belegung der Variablen von F ent-

spricht. 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.

Enthält F n Variablen und m Junktoren, so ist die Anzahl der Zeichen in F von der Ordnung $O(n+m)$, d.h. $size(F) \in O(n+m)$. Der Verifizierer für das Erfüllbarkeitsproblem kann so entworfen werden, dass er ein Laufzeitverhalten der Ordnung $O(n+m)$ hat. Insgesamt ist die Laufzeit dieses nichtdeterministischen Algorithmus also von der Ordnung $O(n+m)$, da das 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 $O(p(n))$ mit einem Polynom p testet.

Ein nichtdeterministischer Algorithmus ist also zunächst ein **Konzept** oder **Gedankenmodell**, da nicht explizit gesagt wird, wie für den Fall $x \in L_{\Pi}$ vom Ratemodul der korrekte Beweis $B_x \in \Sigma_0^*$ erzeugt wird. Mit einer ähnlichen Argumentation wie im Beweis zu Satz 2.5-1 kann man jedoch das Verhalten eines nichtdeterministischen Algorithmus deterministisch simulieren und erhält damit einen funktional gleichwertigen „normalen“ deterministischen Algorithmus. Die folgende Betrachtung soll dabei auf den Fall beschränkt werden, dass der Verifizierer, der innerhalb des nichtdeterministischen Algorithmus einen vom Ratemodul erzeugten 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 $O(T(n) \cdot d^{O(T(n))})$ mit einer Konstanten $d \geq 2$ besitzt.

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

Beweis:

Die Simulation erzeugt bei Eingabe von $x \in \Sigma_{\Pi}^*$ mit $|x|=n$ systematisch alle Beweise $B \in \Sigma_0^*$ mit $|B| \leq c \cdot T(n)$. Dazu muss natürlich der Wert $T(n)$ in „kontrollierbarer Zeit“ berechenbar sein (das trifft zu, da T als zeitkonstruierbar angenommen wird). Der folgende Pseudocode für $\mathbf{A}'_{L_{\Pi}}$ beschreibt die wesentlichen Aspekte der Simulation:

```

FUNCTION  $\mathbf{A}'_{L_{\Pi}}(x)$ ;

VAR n          : INTEGER;
    lng        : INTEGER;
    B          :  $\Sigma_0^*$ ;
    antwort    : BOOLEAN;
    weiter     : BOOLEAN;

BEGIN {  $\mathbf{A}'_{L_{\Pi}}(x)$  }
    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  $\Sigma_0^*$  mit Länge  $\leq$  lng erzeugt worden)
        THEN weiter := FALSE
        ELSE
            BEGIN
                B := nächstes Wort aus  $\Sigma_0^*$  mit Länge  $\leq$  lng;
                IF  $\mathbf{V}_{\Pi}(x, B) = \text{ja}$ 
                THEN BEGIN
                    antwort := TRUE;
                    weiter  := FALSE;
                END;
            END;

    CASE antwort OF
    TRUE  :  $\mathbf{A}'_{L_{\Pi}}(x) := \text{ja}$ ;
    FALSE :  $\mathbf{A}'_{L_{\Pi}}(x) := \text{nein}$ ;
    END;

END {  $\mathbf{A}'_{L_{\Pi}}(x)$  };

```

Die Anzahl der zu überprüfenden Beweise $B \in \Sigma_0^*$ mit $|B| \leq c \cdot T(n)$ ist von der Ordnung $O(|\Sigma_0|^{c \cdot T(n)})$. 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 $L \neq \emptyset$ ist genau dann rekursiv aufzählbar, wenn es eine totale berechenbare Funktion $h: (\Sigma \cup \{0, 1, \#\})^* \rightarrow \Sigma^*$ gibt mit $L = h((\Sigma \cup \{0, 1, \#\})^*)$.

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 $h: (\Sigma \cup \{0, 1, \#\})^* \rightarrow \Sigma^*$ mit $L = h((\Sigma \cup \{0, 1, \#\})^*)$ wird folgendermaßen bewiesen. Wegen $L \neq \emptyset$ 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 Turingmaschi-

ne TM' angeben, die zwei Bänder mehr als TM besitzt (nämlich ein zusätzliches Eingabeband und ein Ausgabeband), bei jeder Eingabe stoppt und die gesuchte Funktion $h : (\Sigma \cup \{0, 1, \#\})^* \rightarrow \Sigma^*$ 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 $L = h((\Sigma \cup \{0, 1, \#\})^*)$:

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 $L \subseteq h((\Sigma \cup \{0, 1, \#\})^*)$.

Ist umgekehrt $u \in h((\Sigma \cup \{0, 1, \#\})^*)$, 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. $h((\Sigma \cup \{0, 1, \#\})^*) \subseteq L$.

Zum Beweis der anderen „Richtung“ der Aussage wird angenommen, dass $L = h((\Sigma \cup \{0, 1, \#\})^*)$ 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 $f : \mathbf{N} \rightarrow M$ 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 \mathbf{N} durch $(\Sigma \cup \{0, 1, \#\})^*$,

so kommt man auf den Begriff der rekursiven Aufzählbarkeit. Für eine rekursiv aufzählbare Menge L gilt $L = h((\Sigma \cup \{0, 1, \#\})^*)$ mit einer totalen und berechenbaren Funktion $h : (\Sigma \cup \{0, 1, \#\})^* \rightarrow \Sigma^*$. 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 $\Sigma = \{a_0, \dots, a_{|\Sigma|-1}\}$ einer Turingmaschine lassen sich als Zeichenkette über dem Alphabet $\{0, 1\}$ kodieren: der Buchstabe a_i hat dabei die Kodierung $\text{bin}(i)$. Ein Wort $w = a_{i_1} \dots a_{i_n}$ kann man dann zunächst in $\text{bin}(i_1)\# \dots \# \text{bin}(i_n)\#$ 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 $w = a_{i_1} \dots a_{i_n}$ und $v = b_{j_1} \dots b_{j_m}$ zunächst in $(\text{bin}(i_1)\# \dots \# \text{bin}(i_n)\#\# \text{bin}(j_1)\# \dots \# \text{bin}(j_m)\#)$ 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 $L \neq \emptyset$ ist genau dann rekursiv aufzählbar, wenn es eine totale berechenbare Funktion $h : \{0, 1\}^* \rightarrow \Sigma^*$ 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 $VERIFIZIERE_TM(w_i) = \text{TRUE}$ 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

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

steht. Da die ersten Spaltenmarkierungen w_0, w_1, w_2, \dots 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:

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

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 $L_d \neq \emptyset$.

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 $\text{VERIFIZIERE_TM}(w_i) = \text{TRUE}$ 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.

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 $L = \{w_1, \dots, w_n\}$ 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 ;
realisiert.

```

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} er-

reicht, wenn $w \in L$ gilt, und den verwerfenden Zustand q_{reject} erreicht, wenn $w \notin L$ ist. Eine auf allen Eingaben $w \in \Sigma^*$ stoppende Turingmaschine TM' mit $L(TM') = \Sigma^* \setminus L$ erhält man aus TM , indem man die Rollen von q_{accept} und q_{reject} vertauscht.

Zu (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 $h: \Sigma^* \rightarrow \Sigma'^*$, die folgende beiden Eigenschaften (i) und (ii) besitzt:

- (i) h ist total und von einer Turingmaschine berechenbar.
- (ii) Es gilt $w \in L$ genau dann, wenn $h(w) \in L'$ ist.

Eigenschaft (ii) kann auch so formuliert werden:

$w \in L$ impliziert $h(w) \in L'$, und $w \notin L$ impliziert $h(w) \notin L'$.

Die Bedeutung der Reduzierbarkeit zeigt folgender Satz:

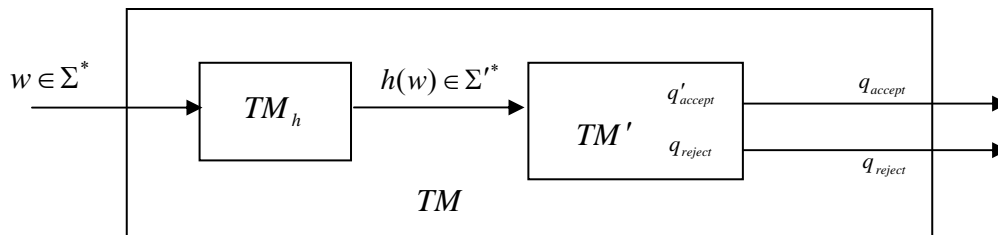
Satz 3.2-3:

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 $h: \Sigma^* \rightarrow \Sigma^*$, 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

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

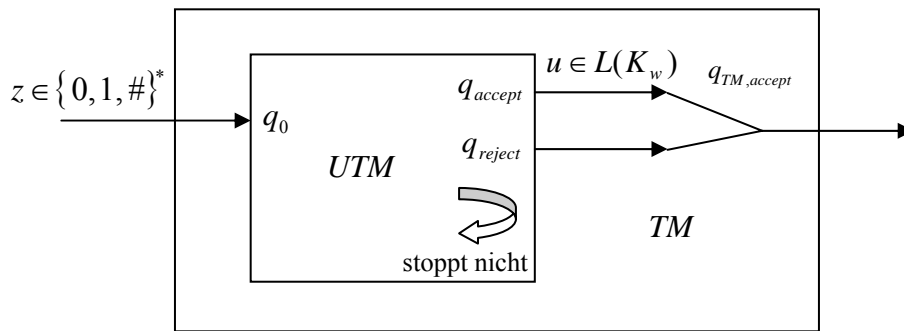
Satz 3.2-5:

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 Überföhrungsfunktion δ_{UTM} von UTM so geändert, dass noch eine Überföhrung in den Zustand $q_{TM,accept}$ ausgeföhrt 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 $u \in \{0, 1\}^*$, $w \in \{0, 1\}^*$, $VERIFIZIERE_TM(w) = \text{TRUE}$ 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.

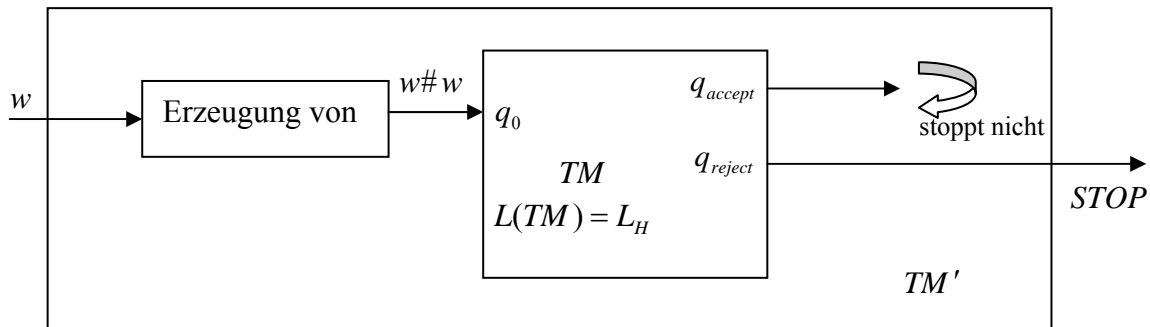
Ist $z \notin L_H$, dann hat z entweder nicht die syntaktische Form $z = u\#w$ mit $u \in \{0, 1\}^*$ und $w \in \{0, 1\}^*$ oder $VERIFIZIERE_TM(w) = \text{FALSE}$ oder K_w stoppt bei Eingabe von u nicht. In den ersten beiden Fällen stoppt UTM bei Eingabe von z nicht (vgl. Beweis von Satz 2.4-1). Im 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 nicht⁹. 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.

⁹ Ist TM' eine k -DTM mit der Überföhrungsfunktion δ' , dann ist $\delta'(q', a_1, \dots, a_k) = (q', (a_1, S), \dots, (a_k, S))$ für $a_1 \in \Sigma, \dots, a_k \in \Sigma$.



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

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

Satz 3.2-6:

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

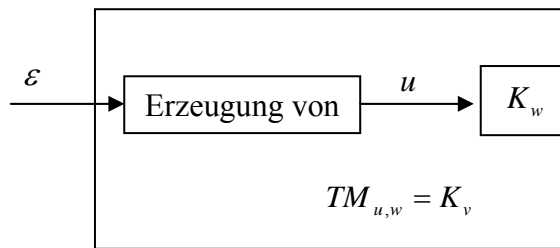
Beweis:

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 $h: \{0,1,\#\}^* \rightarrow \{0,1\}^*$ 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 $\text{VERIFIZIERE_TM}(w) = \text{TRUE}$ 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 $VERIFIZIERE_TM(v) = \text{TRUE}$.



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 gilt¹⁰:

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

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

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

///

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.

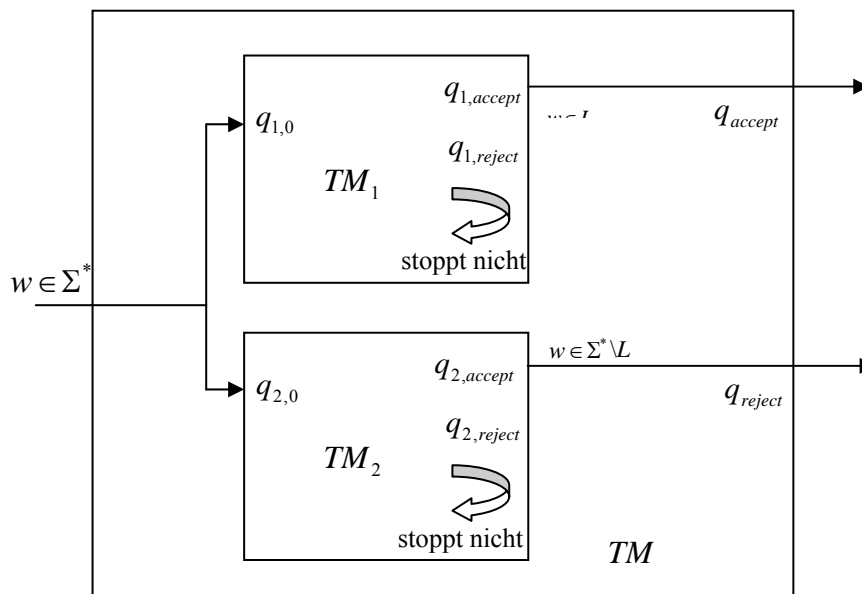
Beweis:

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 $\Sigma^* \setminus L$ entscheidbar und damit rekursiv aufzählbar ist.

¹⁰ Die Abkürzung *gdw* steht für „genau dann, wenn“.

Gilt umgekehrt für eine Menge $L \subseteq \Sigma^*$, dass mit ihr auch das Komplement $\Sigma^* \setminus L$ rekursiv aufzählbar ist, dann gibt es zwei Turingmaschinen TM_1 und TM_2 mit $L = L(TM_1)$ und $\Sigma^* \setminus L = L(TM_2)$. 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 rekursiv 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 $\{0, 1, \#\}^* \setminus L_H$, 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 $\Sigma^* \setminus L$ 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 noch $\Sigma^* \setminus L$ ist rekursiv aufzählbar

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

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

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

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

$$L_{uni} = L(UTM) = \left\{ z \mid \begin{array}{l} z = u\#w \text{ mit } u \in \{0, 1\}^*, w \in \{0, 1\}^*, \\ \text{VERIFIZIERE_TM}(w) = \text{TRUE} \text{ und } u \in L(K_w) \end{array} \right\} \subseteq \{0, 1, \#\}^*$$

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

$$L_d = \{w \mid w \in \{0,1\}^*, VERIFIZIERE_TM(w) = FALSE \text{ oder } w \notin L(K_w)\}$$

nicht rekursiv aufzählbar ist. Daher ist die Menge $\{0,1\}^* \setminus L_d$ nicht entscheidbar. Die

Funktion $g : \{0,1\}^* \rightarrow \{0,1,\#\}^*$ mit $g(w) = w\#w$ ist total und durch eine Turingmaschine berechenbar. Es gilt

$$\begin{aligned} w \in \{0,1\}^* \setminus L_d & \quad gdw \quad w \notin L_d \\ & \quad gdw \quad VERIFIZIERE_TM(w) = TRUE \text{ und } w \in L(K_w) \\ & \quad gdw \quad w\#w \in L_{uni} \\ & \quad gdw \quad g(w) \in L_{uni} . \end{aligned}$$

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

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

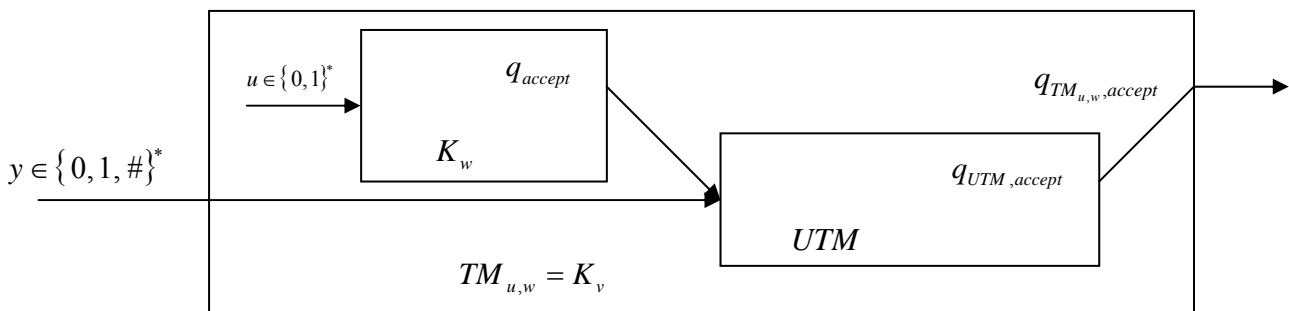
Begründung: Satz 3.2-3 und 2.

4. Es gilt $\bar{L}_{uni} \leq L_e$.

Zur Begründung: Das Aussehen der dabei beteiligten totalen und berechenbaren Funktion $h : \{0,1,\#\}^* \rightarrow \{0,1\}^*$ 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.



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

Bemerkung: Offensichtlich ist $L(K_v)$ genau dann entscheidbar, wenn $u \notin L(K_w)$ ist; denn dann ist $L(K_v) = \emptyset$. 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

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

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

$z \in \bar{L}_{uni}$ gdw 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

$$VERIFIZIERE_TM(w) = \text{FALSE}$$

oder

z hat die Form $z = u\#w$ mit $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$ und

$$VERIFIZIERE_TM(w) = \text{TRUE} \text{ und } u \notin L(K_w).$$

Es sei $z \in \bar{L}_{uni}$. 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 $z \notin \bar{L}_{uni}$, dann ist $z \in L_{uni}$, d.h. $z = u\#w$ mit $u \in \{0,1\}^*$, $w \in \{0,1\}^*$, $VERIFIZIERE_TM(w) = \text{TRUE}$ 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 $z \in \bar{L}_{uni}$ gdw $h(z) \in L_e$, und damit ist $\bar{L}_{uni} \leq L_e$ 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) $L \neq \emptyset$
- (ii) L enthält nicht die Kodierungen aller Turingmaschinen
- (iii) für zwei Turingmaschinen TM_1 und TM_2 , die durch w_1 bzw. w_2 kodiert werden, d.h. $TM_1 = K_{w_1}$ und $TM_2 = K_{w_2}$, impliziert $L(TM_1) = L(TM_2)$: Es gilt $w_1 \in L$ genau dann, wenn $w_2 \in L$ gilt.

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

- Ist L entscheidbar?
- 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.

$$L = \left\{ w \mid \begin{array}{l} \text{die durch } w \text{ kodierte Turingmaschine } K_w \text{ berechnet} \\ \text{eine mit } g \text{ identische Funktion} \end{array} \right\};$$
das Entscheidungsproblem lautet: Ist es entscheidbar, ob die von einer Turingmaschine berechnete Funktion mit g übereinstimmt?
3. $L_{=\emptyset} = \{ w \mid \text{für die durch } w \text{ kodierte Turingmaschine } K_w \text{ gilt } L(K_w) = \emptyset \}$; das Entscheidungsproblem lautet: Ist es entscheidbar, ob die von einer Turingmaschine akzeptierte Menge leer ist?
4. $L_{\neq\emptyset} = \{ w \mid \text{für die durch } w \text{ kodierte Turingmaschine } K_w \text{ gilt } L(K_w) \neq \emptyset \}$; das Entscheidungsproblem lautet: Ist es entscheidbar, ob die von einer Turingmaschine akzeptierte Menge mindestens ein Wort enthält?

5. $L = \{ w \mid \text{für die durch } w \text{ kodierte Turingmaschine } K_w \text{ gilt: } L(K_w) \text{ ist endlich} \}$; 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
 $L = \{ w \mid \text{für die durch } w \text{ kodierte Turingmaschine } K_w \text{ gilt: } L(K_w) = L_0 \}$;
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 $L_{H_0} \leq \{0,1\}^* \setminus L$ 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 $\{0,1\}^* \setminus L$ 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 $L(TM_0) = \emptyset$ und $w_0 = \text{code}(TM_0)$, d.h. $TM_0 = K_{w_0}$. Es werden die beiden Fälle

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

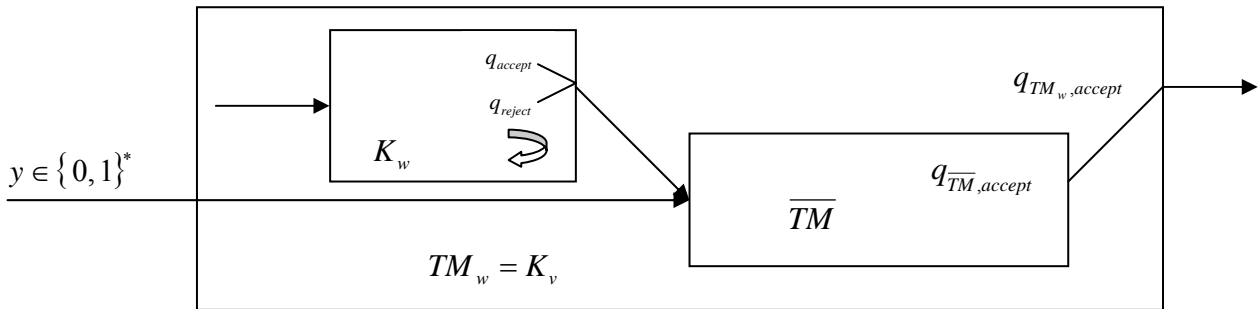
unterschieden.

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

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

Es sei $w \in \{0,1\}^*$ mit $VERIFIZIERE_TM(w) = \text{TRUE}$. 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 $VERIFIZIERE_TM(v) = \text{TRUE}$. Das Wort v ist deterministisch berechenbar, da nur Beschreibungen von Turingmaschinen zusammengestellt wurden.



Man sieht:

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

Die Funktion h wird definiert durch:

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

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

Ist $w \in L_{H_0}$, dann ist (nach Definition von L_{H_0}) $VERIFIZIERE_TM(w) = \text{TRUE}$ 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 $VERIFIZIERE_TM(w) = \text{FALSE}$, 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 $VERIFIZIERE_TM(w) = \text{TRUE}$, dann ist $h(w) = code(TM_w) = v$ und $L(K_v) = \emptyset = L(K_{w_0})$. 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 $h: \{0,1\}^* \rightarrow \{0,1\}^*$ 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 $L = \emptyset$ 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 komplexer¹¹:

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

$$L_{\neq \emptyset} = \left\{ w \mid w \in \{0, 1\}^*, \text{VERIFIZIERE_TM}(w) = \text{TRUE} \text{ und } L(K_w) \neq \emptyset \right\} \text{ und}$$

$$L_{=\emptyset} = \left\{ w \mid w \in \{0, 1\}^*, \text{VERIFIZIERE_TM}(w) = \text{TRUE} \text{ und } L(K_w) = \emptyset \right\}$$

nicht entscheidbar sind. Es wird jedoch nichts darüber gesagt, ob sie rekursiv aufzählbar sind oder nicht. Aus Satz 3.2.-13 folgt, dass $L_{=\emptyset}$ nicht rekursiv aufzählbar ist; denn Bedingung (i) ist verletzt: Die Menge $L_{=\emptyset}$ enthält das Wort w_0 mit $L(K_{w_0}) = \emptyset$ aus dem Beweis zu Satz 3.2-10. Setzt man $L_1 = L(K_{w_0})$ und $L_2 = \{0, 1\}^*$, so ist L_2 trivialerweise rekursiv aufzählbar, etwa $L_2 = L(K_{w_2})$ für eine Kodierung $w_2 \in \{0, 1\}^*$. Es gilt $L_1 \subseteq L_2$, aber $w_2 \notin L_{=\emptyset}$.

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

¹¹ Siehe: Hopcroft, J.E.; Ullman, J.D.: **Introduction to Automata Theory, Languages, and Computation**, Addison Wesley, 1979.

zum anderen zum Nachweis der Relation $\{0,1\}^* \setminus L_{\neq \emptyset} \leq L_{=\emptyset}$ eine „geeignete“ Funktion $h : \{0,1\}^* \rightarrow \{0,1\}^*$.

TM arbeitet wie folgt: Als Eingabe erhält TM ein Wort $w \in \{0,1\}^*$. Falls

$VERIFIZIERE_TM(w) = \text{FALSE}$ ist, wird w nicht akzeptiert. Andernfalls beginnt TM , alle Worte $z \in \{0,1,\#\}^*$ der Form $z = v\#bin(i)$ mit $v \in \{0,1\}^*$ in lexikographischer Reihenfolge zu erzeugen. Jedesmal, wenn ein derartiges Wort generiert worden ist, simuliert TM das Verhalten von K_w bei Eingabe von v für höchstens i Schritte. Wird v dabei akzeptiert, akzeptiert TM die Eingabe w . Andernfalls wird das nächste Wort $v'\#bin(i')$ erzeugt und getestet. Es gilt: Ist $w \in L(TM)$, dann gibt es ein Wort $v \in \{0,1\}^*$, so dass K_w dieses Wort in höchstens i Schritten für ein $i \in \mathbf{N}$ akzeptiert. Daher ist $L(K_w) \neq \emptyset$ und $w \in L_{\neq \emptyset}$.

Ist umgekehrt $w \in L_{\neq \emptyset}$, d.h. $L(K_w) \neq \emptyset$, dann gibt es ein Wort $v \in L(K_w)$. Dieses Wort wird in endlich vielen, etwa j Schritten akzeptiert. Wenn TM also bei Eingabe von w das Wort $v\#bin(j)$ generiert hat, stoppt K_w nach der Simulation von j Schritten im akzeptierenden Zustand, und TM akzeptiert w , d.h. $w \in L(TM)$.

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

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

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

Für die in Satz 3.2-9 untersuchten Sprachen

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

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

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

Zusammenfassung wichtiger Beispiele:

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

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 **Ableitungsregeln (Erzeugungsregeln, Produktionen)** mit $R \subseteq \{v \rightarrow w \mid v \in (N \cup \Sigma)^+, w \in (N \cup \Sigma)^*, |v| \geq 1\}$.

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 Ableitungsregel $v \rightarrow w$ in R gibt. Man schreibt dann: $xvy \xrightarrow{v \rightarrow w} xwy$ oder $xvy \xrightarrow[G]{} xwy$.

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 $x_i \in (N \cup \Sigma)^*$, $i = 0, \dots, t$, gibt mit $x = x_0$, $x_i \xrightarrow[G]{} x_{i+1}$ für $i = 0, \dots, t-1$, $w = x_t$. Man schreibt dann auch $x \xrightarrow[G]^* w$. Die Menge $L(G) = \left\{ w \mid w \in \Sigma^* \text{ und } S \xrightarrow[G]^* w \right\}$ heisst **die von G erzeugte Sprache**.

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

In den folgenden Beispielen a , b und c verschiedene Terminalsymbole.

1. $G_1 = (\{a, b\}, \{S, A, B\}, S, \{S \rightarrow AB, A \rightarrow aA, A \rightarrow \varepsilon, B \rightarrow bB, B \rightarrow \varepsilon\})$ erzeugt die Sprache $L_1 = \{a^i b^j \mid i \in \mathbf{N}, j \in \mathbf{N}\}$. Die Sprache L_1 wird auch von der Grammatik $G'_1 = (\{a, b\}, \{S, B\}, S, \{S \rightarrow \varepsilon, S \rightarrow aS, S \rightarrow bB, B \rightarrow bB, B \rightarrow \varepsilon\})$ erzeugt.
2. $G_2 = (\{a, b\}, \{S\}, S, \{S \rightarrow \varepsilon, S \rightarrow aSb\})$ erzeugt die Sprache $L_2 = \{a^n b^n \mid n \in \mathbf{N}\}$.
3. $G_3 = (\{a, b, c\}, \{S, A, B\}, S, \{S \rightarrow AB, A \rightarrow \varepsilon, A \rightarrow aAb, B \rightarrow \varepsilon, B \rightarrow cB\})$ erzeugt die Sprache $L_3 = \{a^n b^n c^i \mid n \in \mathbf{N}, i \in \mathbf{N}\}$.
4. $G_4 = (\{a, b, c\}, \{S, B\}, S, \{S \rightarrow aSBC, S \rightarrow abc, cB \rightarrow Bc, bB \rightarrow bb\})$ erzeugt die Sprache $L_4 = \{a^n b^n c^n \mid n \in \mathbf{N}, n \geq 1\}$. Die Sprache L_4 wird auch von der Grammatik $G'_4 = \left(\{a, b, c\}, \{S, B, C\}, S, \left\{ \begin{array}{l} S \rightarrow aSBC, S \rightarrow aBC, CB \rightarrow BC, aB \rightarrow ab, bB \rightarrow bb, \\ bC \rightarrow bc, cC \rightarrow cc \end{array} \right\} \right)$ erzeugt.
5. $G_5 = (\{0, 1\}, \{S, A\}, S, \{S \rightarrow 1A0A, S \rightarrow 0A1A, A \rightarrow \varepsilon, A \rightarrow 0A1A, A \rightarrow 1A0A\})$ erzeugt die Sprache $L_5 = \{w \mid w \in \{0, 1\}^+ \text{ und die Anzahl der Zeichen } 0 \text{ in } w \text{ ist gleich der Anzahl der Zeichen } 1\}$

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

Für ein Wort $w \in \{0, 1\}^*$ ist

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

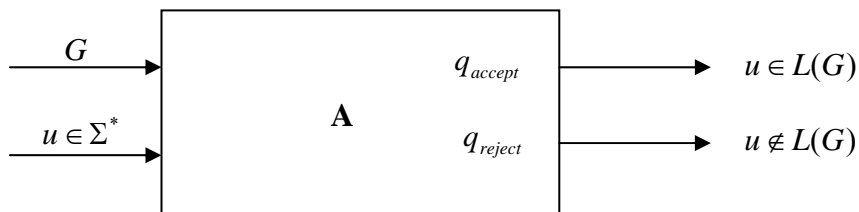
Die durch ein Wort $w \in \{0, 1\}^*$ mit $VERIFIZIERE_G(w) = \text{TRUE}$ kodierte Grammatik sei KG_w .

Die zum Wortproblem gehörende Menge L_{wort} wird definiert durch

$$L_{wort} = \left\{ u\#w \mid u \in \Sigma^*, w \in \{0, 1\}^*, VERIFIZIERE_G(w) = \text{TRUE} \text{ und } u \in L(KG_w) \right\}.$$

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.

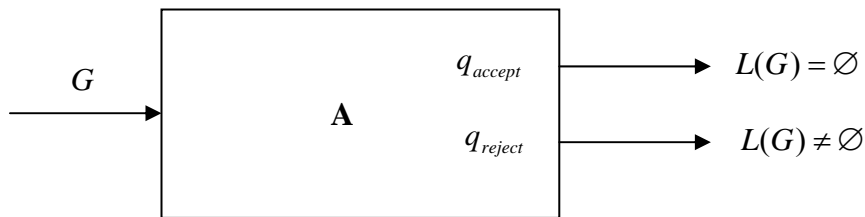


Entsprechend kann man das Leerheitsproblem definieren:

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

$$L_{\text{leer}} = \left\{ w \mid w \in \{0,1\}^*, \text{VERIFIZIERE_}G(w) = \text{TRUE und } L(KG_w) = \emptyset \right\}.$$

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



4.2 Typ-0-Sprachen

Eine Sprache, die von einer Grammatik $G = (\Sigma, N, S, R)$ erzeugt wird, für die

$$R \subseteq \left\{ v \rightarrow w \mid v \in (N \cup \Sigma)^+ \setminus \Sigma^*, w \in (N \cup \Sigma)^* \right\}$$

gilt, heißt **Typ-0-Sprache**, die Grammatik heißt **Typ-0-Grammatik**. Die Ableitungsregeln $v \rightarrow w$ 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 $xvy \xrightarrow{v \rightarrow w} xwy$ 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:

- (i) Zu jeder von einer Typ-0-Grammatik G erzeugten Sprache $L(G)$ gibt es eine Turingmaschine TM_G mit $L(TM_G) = L(G)$.
- (ii) Zu jeder von einer Turingmaschine TM akzeptierten Sprache $L(TM)$ gibt es eine Typ-0-Grammatik G_{TM} mit $L(G_{TM}) = L(TM)$.

Auf den Beweis wird hier verzichtet, da der Schwerpunkt der Themenauswahl aus Gründen der größeren Praxisnähe bei den in den folgenden Kapiteln behandelten Sprachtypen liegt. Er kann in der angegebenen Literatur nachgelesen werden¹².

In Kapitel 3.2 wird gezeigt, dass die Menge

$$L_{uni} = \left\{ u\#w \mid u \in \{0,1\}^*, w \in \{0,1\}^*, VERIFIZIERE_TM(w) = \text{TRUE} \text{ und } u \in L(K_w) \right\}$$

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

$$VERIFIZIERE_G_TYP_0(w)$$

$$= \begin{cases} \text{TRUE} & \text{falls } w \text{ die Kodierung einer Typ-0-Grammatik darstellt} \\ \text{FALSE} & \text{falls } w \text{ nicht die Kodierung einer Typ-0-Grammatik darstellt} \end{cases}$$

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

$$L_{Wort_Typ-0} = \left\{ u\#w \mid u \in \Sigma^*, w \in \{0,1\}^*, VERIFIZIERE_G_TYP_0(w) = \text{TRUE} \text{ und } u \in L(KG_w) \right\}$$

entscheidbar ist. Diese Frage muss verneint werden.

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

$$L_{leer_Typ-0} = \left\{ w \mid w \in \{0,1\}^*, VERIFIZIERE_G_TYP_0(w) = \text{TRUE} \text{ und } L(KG_w) = \emptyset \right\}$$

ist nicht entscheidbar, da die Menge

$$L_{=\emptyset} = \left\{ w \mid w \in \{0,1\}^*, VERIFIZIERE_TM(w) = \text{TRUE} \text{ und } L(K_w) = \emptyset \right\}$$

nicht rekursiv aufzählbar und damit auch nicht entscheidbar ist (vgl. Kapitel 3.2).

¹² Siehe: Hopcroft, J.E.; Ullman, J.D.: **Introduction to Automata Theory, Languages, and Computation**, Addison Wesley, 1979.

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 $L(G) = \emptyset$ gilt.

4.3 Typ-1-Sprachen

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

$R \subseteq \{v \rightarrow w \mid v \in (N \cup \Sigma)^+ \setminus \Sigma^*, w \in (N \cup \Sigma)^*\}$ folgender zusätzlicher Einschränkung unterliegen:

Für alle Regeln $v \rightarrow w$ gilt $|v| \leq |w|$. Als einzige Ausnahme ist die Regel $S \rightarrow \varepsilon$ zugelassen; wenn diese Regel vorkommt, darf S auf keiner rechten Seite einer Regel vorkommen.

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

Wie bei einer Typ-0-Grammatik dürfen auf der linken Seite einer Ableitungsregel 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 $S \rightarrow \varepsilon$ mindestens so groß wie die Länge der linken Seite. Daher wird in einem Ableitungsschritt $xvy \xrightarrow{v \rightarrow w} xwy$ die Länge des Ableitungsergebnisses nicht kürzer. In einer Ableitung $S \xRightarrow{*} w$ eines Wortes $w \in \Sigma^+$, etwa $S \Rightarrow x_1 \Rightarrow \dots \Rightarrow x_i \Rightarrow x_{i+1} \Rightarrow \dots \Rightarrow w$, gilt für alle Zwischenschritte $1 = |S| \leq |x_1| \leq \dots \leq |x_i| \leq |x_{i+1}| \leq \dots \leq |w|$. Diese Beobachtung wird wichtig, wenn man den Turingmaschinentyp charakterisieren möchte, der eine von einer Typ-1-Grammatik erzeugten Sprache erkennt.

Enthält R die Ableitungsregel $S \rightarrow \varepsilon$, dann ist $\varepsilon \in L(G)$, und die Anwendung der Ableitungsregel $S \rightarrow \varepsilon$ stellt die einzige Möglichkeit dar, um ε aus S abzuleiten.

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 die folgenden Beispiele zeigen, die in den anschließenden Kapiteln wieder aufgenommen werden.

Eine Typ-1-Sprache zur Erzeugung der Zweierpotenzen in unärer Darstellung

Es wird eine Typ-1-Grammatik entwickelt, die die Sprache $\{a^{2^n} \mid n \in \mathbf{N}_{>0}\}$ erzeugt. Ein erster Ansatz führt auf die Grammatik $G_1 = (\{a\}, \{S, A, B, C, D, E\}, S, R)$, wobei R aus den Ableitungsregeln

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

besteht. Die Nichtterminalzeichen A und B dienen als „Begrenzer“; C ist eine Markierung, die durch die Zeichenkette $a \dots a$ zwischen A und B hindurch läuft 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 Ableitungsregel ansetzt, sind jeweils unterstrichen):

$$\begin{aligned} \underline{S} &\Rightarrow \underline{ACaB} \Rightarrow \underline{AaaCB} \Rightarrow \underline{AaaDB} \Rightarrow \underline{AaDaB} \Rightarrow \underline{ADaaB} \Rightarrow \underline{ACaaB} \Rightarrow^* \underline{AaaaaCB} \\ &\Rightarrow^* \underline{ADaaaaB} \Rightarrow^* \underline{Aa^{16}CB} \Rightarrow \underline{Aa^{15}aE} \Rightarrow \underline{Aa^{14}aEa} \Rightarrow^* \underline{AEaa^{15}} \Rightarrow a^{16} \end{aligned}$$

Offensichtlich sind in G_1 folgende Ableitungen möglich:

$$S \Rightarrow^* ACa^{2^0}B \text{ und für } i \geq 0:$$

$$S \Rightarrow^* ACa^{2^i}B \Rightarrow^* Aa^{2^{i+1}}CB \Rightarrow Aa^{2^{i+1}}DB \Rightarrow^* ADA^{2^{i+1}}B \Rightarrow ACa^{2^{i+1}}B.$$

Daraus folgt durch vollständige Induktion, dass G_1 die Ableitungen $S \Rightarrow^* ACa^{2^n}B$ für alle $n \in \mathbf{N}$ erlaubt.

$$\text{Weiterhin gilt für } i \geq 0: S \Rightarrow^* ACa^{2^i}B \Rightarrow^* Aa^{2^{i+1}}CB \Rightarrow Aa^{2^{i+1}}E \Rightarrow^* AEaa^{2^{i+1}-1} \Rightarrow a^{2^{i+1}}.$$

Daraus folgt $L(G_1) = \{a^{2^n} \mid n \in \mathbf{N}_{>0}\}$.

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, \dots, E werden mit Zeichen a zu neuen nichtterminalen Zeichen zusammengesetzt:

$G_2 = (\{a\}, N_2, S, R_2)$ 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) $S \rightarrow Z_{ACaB}$ | (5) $aZ_{Da} \rightarrow Z_{Da}a$ |
| (2) $Z_{Ca}a \rightarrow aaZ_{Ca}$ | $Z_{aDB} \rightarrow Z_{DaB}$ |
| $Z_{Ca}Z_{aB} \rightarrow aaZ_{CaB}$ | $Z_{Aa}Z_{Da} \rightarrow Z_{ADa}a$ |
| $Z_{ACa}a \rightarrow Z_{Aa}aZ_{Ca}$ | $aZ_{DaB} \rightarrow Z_{Da}Z_{aB}$ |
| $Z_{ACa}Z_{aB} \rightarrow Z_{Aa}aZ_{CaB}$ | $Z_{Aa}Z_{DaB} \rightarrow Z_{ADa}Z_{aB}$ |
| $Z_{ACaB} \rightarrow Z_{Aa}Z_{aCB}$ | (6) $Z_{ADa} \rightarrow Z_{ACa}$ |
| $Z_{CaB} \rightarrow aZ_{aCB}$ | (7) $aZ_{Ea} \rightarrow Z_{Ea}a$ |
| (3) $Z_{aCB} \rightarrow Z_{aDB}$ | $Z_{aE} \rightarrow Z_{Ea}$ |
| (4) $Z_{aCB} \rightarrow Z_{aE}$ | $Z_{Aa}Z_{Ea} \rightarrow Z_{AEa}a$ |
| | (8) $Z_{AEa} \rightarrow a$ |

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

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

Eine Typ-1-Sprache zur Erzeugung der Quadratzahlen in unärer Darstellung

Die folgende Typ-1-Grammatik $G = (\{a\}, \{S, A, B, C, D, E\}, S, R)$, erzeugt die Sprache $\{a^{n^2} \mid n \in \mathbf{N}\}$. Für die „kleinen“ Werte $n=0$, $n=1$ und $n=2$ werden die Wörter a^{n^2} direkt abgeleitet. Dazu dienen die Ableitungsregeln

(1) $S \rightarrow \varepsilon$, $S \rightarrow a$ und $S \rightarrow aaaa$.

Für längere Wörter wird die Identität $n^2 = \sum_{i=0}^{n-1} (2 \cdot i + 1)$ genutzt: In einer aus dem Startsymbol

ableitbaren Satzform werden sowohl die Werte n^2 als auch der obere Laufindex $n-1$ kodiert, genauer: Die in (2) angegebenen Ableitungsregeln erlauben die Erzeugung der Zeichenketten $a^{n^2} AC^{n-1} B$ für alle $n \geq 2$:

(2) $S \rightarrow a^4 ACB$
 $B \rightarrow EaaB$
 $CE \rightarrow EaaC$
 $AE \rightarrow aAC$
 $Ca \rightarrow aC$
 $Aa \rightarrow aA$

Denn für $n = 2$ wird die erste der Ableitungsregeln in (2) verwendet, und für $n \geq 2$ gilt

$$\begin{aligned} a^{n^2} AC^{n-1} B &\Rightarrow a^{n^2} AC^{n-1} EaaB \Rightarrow^* a^{n^2} AE \underbrace{aaC \dots aaC}_{(n-1)\text{-mal } aaC} aab \Rightarrow a^{n^2} aAC \underbrace{aaC \dots aaC}_{(n-1)\text{-mal } aaC} aab \\ &\Rightarrow^* a^{n^2} aAa^{2 \cdot n} C^n B \Rightarrow^* a^{n^2} a^{2 \cdot n + 1} AC^n B . \end{aligned}$$

Wegen $n^2 + (2 \cdot n + 1) = (n+1)^2$ ist $a^{n^2} a^{2 \cdot n + 1} AC^n B = a^{(n+1)^2} AC^n B$.

Eine weitere Gruppe (3) von Ableitungsregeln erlaubt für $n \geq 2$ Ableitungen $a^{n^2} AC^{n-1} B \Rightarrow^* a^{(n+1)^2}$:

(3) $CB \rightarrow Daa$
 $CD \rightarrow Daa$
 $AD \rightarrow aaa$

Denn $a^{n^2} AC^{n-1} B \Rightarrow a^{n^2} AC^{n-2} Daa \Rightarrow^* a^{n^2} AD \underbrace{aa \dots aa}_{(n-2)\text{-mal } aa} aa \Rightarrow a^{n^2} aaa \underbrace{aa \dots aa}_{(n-2)\text{-mal } aa} aa$. Die Anzahl

der Terminalzeichen a ist $n^2 + 2 \cdot (n-2) + 5 = n^2 + 2 \cdot n + 1 = (n+1)^2$.

Insgesamt folgt $L(G) = \{a^{n^2} \mid n \in \mathbf{N}\}$.

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 $O(2^{O(n)})$ vielen Schritten getroffen werden; das Entscheidungsverfahren hat also exponentielle Laufzeit.

Beweis:

Es sei $G = (\Sigma, N, S, R)$ eine kontextsensitive Grammatik. Zu zeigen ist: $L(G)$ ist entscheidbar. Dazu wird eine auf allen Eingaben $w \in \Sigma^*$ stoppende Turingmaschine TM_G angegeben, die w genau dann akzeptiert, wenn $w \in L(G)$ ist. Die Arbeitsweise von TM_G wird hier informell beschrieben:

Es sei $w \in \Sigma^*$ eine Eingabe von TM_G . Für $w = \varepsilon$ akzeptiert TM_G diese Eingabe genau dann, wenn G die Ableitungsregel $S \rightarrow \varepsilon$ 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 $S \xRightarrow{*} w$ 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. $\alpha \xRightarrow{G} \beta$, 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, die eine Laufzeit der Ordnung $O(m^3)$ besitzen, falls der entsprechende Graph m Knoten besitzt¹³. Der beschriebene Graph besitzt eine Anzahl an Knoten der Ordnung $O(c^n)$ (mit einer Konstanten $c = c(G)$). Daher lässt sich die Laufzeit des Pfadsuchalgorithmus durch eine Funktion der Ordnung $O(c^{3n})$ beschränken. Insgesamt ist die Laufzeit des Entscheidungsverfahrens 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

¹³ Siehe: Hoffmann, U.: **Datenstrukturen und Algorithmen**, FINAL 15:2, 2005.

$n+1$ wird das Ende des Eingabeworts erkannt). Die von einem LBA akzeptierte Menge $L(LBA)$ wird wie bei Turingmaschinen definiert.

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:

- (i) Zu jeder von einer kontextsensitiven Grammatik G erzeugten Sprache $L(G)$ gibt es einen linear beschränkten Automaten LBA_G mit $L(LBA_G) = L(G)$.
- (ii) Zu jeder von einem linear beschränkten Automaten LBA akzeptierten Sprache $L(LBA)$ gibt es eine kontextsensitive Grammatik G_{LBA} mit $L(G_{LBA}) = L(LBA)$.
- (iii) Die von einem linear beschränkten Automaten LBA akzeptierte Menge $L(LBA)$ über einem Alphabet Σ ist entscheidbar.

Beweis:

Zu (i): Es sei $G = (\Sigma, N, S, R)$ eine kontextsensitive Grammatik. Es wird informell die Arbeitsweise eines nichtdeterministischen linear beschränkten Automaten LBA_G mit $L(LBA_G) = L(G)$ beschrieben:

LBA_G besitzt neben seinem Eingabeband (und eventuell weiterer Hilfsbänder) ein zweites Band, auf dem im wesentlichen Zeichenketten erzeugt werden, die sich in G aus dem Startsymbol herleiten lassen. Es werde ein Eingabewort w auf das Eingabeband gegeben. Ist $w = \varepsilon$ und enthält G die Ableitungsregel $S \rightarrow \varepsilon$, so akzeptiert LBA_G die Eingabe ohne weitere Kopfbewegung; enthält G die Ableitungsregel $S \rightarrow \varepsilon$ nicht, so akzeptiert LBA_G die Eingabe $w = \varepsilon$ nicht.

Im folgenden sei $w \neq \varepsilon$, $|w| = n$.

LBA_G erzeugt auf seinen zweiten Band die Zeichenkette $\underbrace{Sb \dots b}_{n \text{ Zeichen}}\#$; hierbei ist b das

Leerzeichen und $\#$ ein Zeichen, das in $\Sigma \cup N$ nicht vorkommt. Hierbei wird genutzt, dass das Ende von w am ersten Blankzeichen rechts von w erkannt wird. Zur Erzeugung des Inhalts auf dem zweiten Band müssen die Köpfe des ersten und zweiten Bandes nicht über die Position $n+1$ hinausrücken. Das zweite Band enthält jetzt eine Zeichenkette der Form $\underbrace{\alpha b \dots b}_{n \text{ Zeichen}}\#$ mit $\alpha \in (\Sigma \cup N)^*$, $|\alpha| \leq n$, $|\alpha b \dots b| = n$ und $S \Rightarrow^* \alpha$.

Es werden die beiden folgenden Schritte 1. und 2. durchlaufen, bis LBA_G entweder akzeptiert oder verwirft:

1. Schritt: Die Zeichenkette α des zweiten Bandes wird mit w auf dem Eingabeband zeichenweise verglichen. Bei Gleichheit wird die Eingabe akzeptiert, ansonsten Schritt 2. ausgeführt.
2. Schritt: Es wird nichtdeterministisch innerhalb α eine Position bestimmt, an der sich eine Ableitungsregel $v \rightarrow v'$ ansetzen lässt, d.h. α hat dann die Form $\alpha = \alpha'v\alpha''$. Die Auswahl der Ableitungsregel $v \rightarrow v'$ erfolgt ebenfalls nichtdeterministisch. Ist eine Auswahl nicht möglich, wird die Eingabe nicht akzeptiert. Der Inhalt des zweiten Bandes wird ersetzt durch $\alpha'v'\alpha''$. Dazu muss die Zeichenkette α'' eventuell in den Bereich der Blankzeichen nach rechts verschoben werden. Wenn dabei versucht wird, das Zeichen # zu überschreiben, wird die Eingabe w nicht akzeptiert, ansonsten mit Schritt 1. fortgesetzt, wobei $\alpha'v'\alpha''$ die Rolle von α übernimmt.

Ist $w \in L(G)$, etwa mit der Ableitung $S \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_i \Rightarrow \alpha_{i+1} \Rightarrow \dots \Rightarrow w$, so gilt für alle Zwischenschritte $1 = |S| \leq |\alpha_1| \leq \dots \leq |\alpha_i| \leq |\alpha_{i+1}| \leq \dots \leq |w|$. Daher rücken keine Köpfe von LBA_G auf eine Position jenseits der Position $n+1$, und w wird in Schritt 1. akzeptiert. Ist $w \notin L(G)$, so scheitert die Simulation von LBA_G in Schritt 2. entweder weil sich keine Ableitungsregel passend auswählen lässt oder weil versucht wird, das Zeichen # nach rechts zu versetzen.

Zu (ii): Die etwas aufwendige Konstruktion einer Typ-1-Grammatik aus dem linear beschränkten Automaten LBA kann in der Literatur nachgelesen werden¹⁴.

Zu (iii): Die Aussage folgt aus Satz 4.3-1.

///

Möchte man daher Aussagen über kontextsensitive Sprachen beweisen, so kann man dazu mit kontextsensitiven Grammatiken oder linear beschränkten Automaten argumentieren. Das folgende Beispiel zeigt, dass die Menge Primzahlen in unärer Darstellung kontextsensitiv ist.

Ein linear beschränkter Automat zur Akzeptanz der Primzahlen in unärer Darstellung

Es wird informell die Arbeitsweise eines linear beschränkten Automaten LBA mit $L(LBA) = \{a^n \mid n \text{ ist Primzahl}\}$ beschrieben; eine Reihe technischer Details werden dabei vernachlässigt. LBA besitzt drei Bänder und eventuell weitere Hilfsbänder. Bei Eingabe eines Wortes $w \in \{a^n \mid n \in \mathbf{N}\}$ prüft LBA , ob $n=0$ oder $n=1$ ist; in diesem Fall wird w nicht ak-

¹⁴ Siehe: Hopcroft, J.E.; Ullman, J.D.: **Introduction to Automata Theory, Languages, and Computation**, Addison Wesley, 1979.

zeptiert. Ist $n = 2$, so wird w akzeptiert. Andernfalls, d.h. $n \geq 3$, durchläuft LBA die folgenden Schritte 1. bis 3., bis LBA entweder akzeptiert oder verwirft:

1. Schritt: LBA erzeugt auf seinem dritten Band die Zeichenkette $a \underbrace{ab \dots b}_{\substack{n-3 \\ \text{Blanks}}} \#$.
2. Schritt: LBA erzeugt auf seinem zweiten Band die Zeichenkette $\underbrace{b \dots b}_{n \text{ Blanks}} \#$.
3. Schritt: Das dritte Band enthält eine Zeichenkette der Form $a^d \underbrace{b \dots b}_{\substack{n-d-1 \\ \text{Blanks}}} \#$ mit $2 \leq d < n$ (der

Wert d ist ein potentieller Teiler von n). LBA kopiert die Zeichenkette a^d solange hintereinander auf das zweite Band, bis versucht wird, das Zeichen $\#$ zu überschreiben. Passen m Kopien der Zeichenkette a^d auf dem zweiten Band genau in den Platz bis zum Zeichen $\#$, so ist $n = d \cdot m$, und die Eingabe wird nicht akzeptiert. Ansonsten wird die Zeichenkette auf dem dritten Band um die Zeichenkette a erweitert, falls dabei nicht das Zeichen $\#$ zu überschreiben versucht wird (das dritte Band enthält nun $a^{d+1} \underbrace{b \dots b}_{\substack{n-d-2 \\ \text{Blanks}}} \#$), und es wird mit dem 2. Schritt fortge-

setzt. Falls versucht wird, das Zeichen $\#$ zu überschreiben, so wird w akzeptiert; denn keine der Werte d mit $2 \leq d < n$ sind Teiler von n .

Ein linear beschränkter Automat zur Akzeptanz der Zweierpotenzen in unärer Darstellung

Es wird informell die Arbeitsweise eines linear beschränkten Automaten LBA mit $L(LBA) = \{a^{2^n} \mid n \in \mathbf{N}_{>0}\}$ beschrieben: LBA besitzt zwei Bänder und eventuell weitere Hilfsbänder. Bei Eingabe eines Wortes $w \in \{a^n \mid n \in \mathbf{N}\}$ prüft LBA , ob $n = 0$ oder $n = 1$ ist; in diesem Fall wird w nicht akzeptiert. Ist $n = 2$, so wird w akzeptiert. Für $n \geq 3$ führt LBA die folgenden Aktionen aus:

LBA erzeugt auf seinem zweiten Band die Zeichenkette 0. Der Inhalt des zweiten Bandes kann als Binärzahl interpretiert werden. Die Eingabe w auf dem ersten Band wird von links nach rechts gelesen, bis das erste Blankzeichen, das das Ende von w markiert, erreicht ist. Bei jedem Lesen eines Zeichens a wird 1 auf die Binärzahl, die auf dem zweiten Band steht, addiert. Enthält das zweite Band eine Zeichenkette der Form $1 \underbrace{0 \dots 0}_{i \text{ mal}} 0$, so wird die Eingabe akzeptiert;

denn dann hat w die Form a^{2^i} .

Ein linear beschränkter Automat zur Akzeptanz der Quadratzahlen in unärer Darstellung

Zur Feststellung der Quadratzahleigenschaft wird wieder die Identität $n^2 = \sum_{i=0}^{n-1} (2 \cdot i + 1)$ genutzt. Es wird informell die Arbeitsweise eines linear beschränkten Automaten LBA mit $L(LBA) = \{a^{n^2} \mid n \in \mathbf{N}\}$ beschrieben. LBA besitzt drei Bänder und eventuell weitere Hilfsbänder. Bei Eingabe eines Wortes $w \in \{a^n \mid n \in \mathbf{N}\}$ prüft LBA , ob $n = 0$ oder $n = 1$ ist; in diesem Fall wird w akzeptiert. Andernfalls durchläuft LBA die folgenden Schritte 1. bis 3., bis LBA entweder akzeptiert oder verwirft:

1. Schritt: LBA erzeugt auf seinem zweiten und dritten Band jeweils die Zeichenkette

$$a \underbrace{b \dots b}_{\substack{n-1 \\ \text{Blanks}}} \#.$$

2. Schritt: Der Inhalt des zweiten Bands hat die Form $a^d \underbrace{b \dots b}_{\substack{n-d \\ \text{Blanks}}} \#$ mit $d = \sum_{i=0}^k (2 \cdot i + 1)$ und

$k \geq 0$. Die Zeichenkette a^d des zweiten Bands wird mit w auf dem ersten Band verglichen. Bei Gleichheit wird w akzeptiert. Andernfalls wird mit dem dritten Schritt fortgesetzt.

3. Schritt: Das dritte Band hat einen Inhalt der Form $a^e \underbrace{b \dots b}_{\substack{n-e \\ \text{Blanks}}} \#$ mit $e = 2 \cdot i + 1$. Dieser wird

um zwei Zeichen a erweitert. Falls dabei versucht wird, das Zeichen $\#$ zu überschreiben, wird w verworfen. Andernfalls wird versucht, die Zeichenkette $a^e aa$ des dritten Bands zwischen die Zeichenkette a^d und das Zeichen $\#$ linksbündig zu kopieren. Falls dabei versucht wird, das Zeichen $\#$ zu überschreiben, wird w nicht akzeptiert. Andernfalls wird mit dem 2. Schritt fortgefahren.

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 eine rekursiv aufzählbare Menge mit

$$L_0 \subseteq \left\{ w \mid w \in \{0, 1\}^*, \text{VERIFIZIERE_TM}(w) = \text{TRUE} \text{ und } K_w \text{ stoppt auf jeder Eingabe} \right\}.$$

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

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

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 $h: \{0,1\}^* \rightarrow \{0,1\}^*$ mit $L_0 = h(\{0,1\}^*)$.

Es wird $L = \left\{ w \mid w \in \{0,1\}^* \text{ und } w \notin L(K_{h(w)}) \right\}$ 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 $\text{VERIFIZIERE_TM}(h(w)) = \text{TRUE}$. 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

$$w \in L \text{ gdw } w \notin L(K_{h(w)}) \text{ (nach Definition von } L)$$

$$\text{gdw } w \notin L(K_{w_L}) \text{ (wegen } h(w) = w_L)$$

$$\text{gdw } w \notin L(TM) \text{ (da } w_L \text{ die Kodierung von } TM \text{ ist)}$$

$$\text{gdw } w \notin L \text{ (wegen } L(TM) = L).$$

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 $h: \{0, 1\}^* \rightarrow \{0, 1\}^*$:

$$(i \leftrightarrow) bin(i) \rightarrow i\text{-te kontextsensitive Grammatik } G_i \rightarrow TM_{G_i} \rightarrow code(TM_{G_i}).$$

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 mit Satz 3.1-2 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 Überföhrungsfunktion die Form $\delta: Q \times \Sigma^k \rightarrow \mathbf{P}(Q \times (\Sigma \times \{L, R, S\})^k)$ hat, ein **deterministischer linear beschränkter Automat** liegt vor, wenn die Überföhrungsfunktion die Form $\delta: Q \times \Sigma^k \rightarrow Q \times (\Sigma \times \{L, R, S\})^k$ 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 *deterministischen* linear beschränkten Automaten akzeptiert wird. Dieses offene Problem heißt **LBA-Problem**. Zu beachten ist dabei folgendes: Ist L eine kontextsensitive Sprache, dann gibt es einen nichtdeterministischen linear beschränkten Automaten LBA mit $L = L(LBA)$. Die Akzeptanz eines Wortes w mit $|w| = n$ benötigt eine Anzahl von Zellen, die durch die lineare Funktion $S(n) = n + 1$ gegeben ist. Das nichtdeterministische Verhalten einer Turingmaschine, die durch eine Funktion der Ordnung $O(f(n))$ speicherplatzbeschränkt ist mit einer speicherplatzkonstruierbaren 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 ist¹⁵. 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

$VERIFIZIERE_G_TYP_1(w)$

$$= \begin{cases} \text{TRUE} & \text{falls } w \text{ die Kodierung einer Typ-1-Grammatik darstellt} \\ \text{FALSE} & \text{falls } w \text{ nicht die Kodierung einer Typ-1-Grammatik darstellt} \end{cases}$$

definiert. Dieses Prädikat ist berechenbar. Dazu ist unter anderem zu prüfen, ob die Ableitungsregeln in KG_w den Bedingungen einer kontextsensitiven Grammatik genügen.

Das **Wortproblem für Typ-1-Grammatiken** fragt danach, ob die Menge

$$L_{\text{Wort_Typ-1}} = \left\{ u\#w \mid u \in \Sigma^*, w \in \{0,1\}^*, VERIFIZIERE_G_TYP_1(w) = \text{TRUE und } u \in L(KG_w) \right\}$$

entscheidbar ist.

Aus dem Beweis von Satz 4.3-2 folgt, dass das Wortproblem für kontextsensitive Grammatiken entscheidbar ist. Das dort angegebene Entscheidungsverfahren hat exponentielle Laufzeit, und es ist nicht bekannt, ob es ein Entscheidungsverfahren gibt, das polynomielle Laufzeit aufweist.

Das **Leerheitsproblem für Typ-1-Grammatiken**, nämlich die Frage, ob die Menge

$$L_{\text{leer_Typ-1}} = \left\{ w \mid w \in \{0,1\}^*, VERIFIZIERE_G_TYP_1(w) = \text{TRUE und } L(KG_w) = \emptyset \right\}$$

entscheidbar ist, muss wie bei Typ-0-Grammatiken verneint werden¹⁶.

¹⁵ Bovet, D.P.; Crescenzi, P.: **Introduction to the Theory of Complexity**, Prentice Hall, 1994.

¹⁶ Hopcroft, J.E.; Ullman, J.D.: **Introduction to Automata Theory, Languages, and Computation**, Addison Wesley, 1979.

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 $L(G) = \emptyset$ gilt.

4.4 Typ-2-Sprachen

Es sei $G = (\Sigma, N, S, R)$ eine Grammatik, in der für die Ableitungsregeln $R \subseteq \{A \rightarrow w \mid A \in N \text{ und } w \in (N \cup \Sigma)^*\}$ 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 **Typ-2-Grammatik** oder **kontextfreie Grammatik**. Beispielsweise sind die Sprachen

$L_2 = \{a^n b^n \mid n \in \mathbf{N}\}$, $L_3 = \{a^n b^n c^i \mid n \in \mathbf{N}, i \in \mathbf{N}\}$ und

$L_5 = \{w \mid w \in \{0, 1\}^+ \text{ und die Anzahl der Zeichen } 0 \text{ in } w \text{ ist gleich der Anzahl der Zeichen } 1\}$ aus Kapitel 4.1 kontextfreie Sprachen (Typ-2-Sprachen).

Die kontextfreien Sprachen zählen zu den am intensivsten erforschten Sprachen. Einen über-
ragenden Erfolg haben sie in der Anwendung und der Theorie des Compilerbaus. Moderne
Programmiersprachen sind erst nach intensiver Erforschung der kontextfreien Sprachen und
unter Anwendung dieser Theorie entwickelt worden. Die Syntax der meisten heute üblichen
Programmiersprachen wird in Form einer Grammatik definiert, die weitestgehend kontextfrei
ist. Man kann jedoch formal zeigen, dass eine Programmiersprache, in der Variablen mit Da-
tentypen 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 deklarerter Objekte zulässig ist usw., nicht komplett durch eine kon-
textfreie 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 Angabe der Form $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ steht hier für die n Ableitungsregeln $A \rightarrow \alpha_1$, $A \rightarrow \alpha_2$, ..., $A \rightarrow \alpha_n$. Ein Angabe in eckigen Klammern innerhalb einer Ableitungsregel bezeichnet einen optionalen Teil, d.h. $A \rightarrow \alpha[\beta]\gamma$ steht für die Ableitungsregeln $A \rightarrow \alpha\beta\gamma$ und $A \rightarrow \alpha\gamma$.

```
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 Ableitungsregeln einer kontextfreien Grammatik erfüllen (auf den ersten Blick) auch die Bedingung, die man an eine kontextsensitive Grammatik stellt. In einer kontextfreien Grammatik können jedoch auch Ableitungsregeln der Form $A \rightarrow \varepsilon$ vorkommen, wobei $A \neq S$ oder zusätzlich A auf der rechten Seite einer Ableitungsregel vorkommt. Der Beweis des folgenden Satzes liefert ein endliches Verfahren, um die Ableitungsregeln und nichtterminalen Symbole

der Grammatik so abzuändern, dass weiterhin dieselbe Sprache erzeugt wird und die Bedingungen an eine kontextsensitive Grammatik erfüllt werden.

Satz 4.4-1:

Zu jeder kontextfreien Grammatik $G = (\Sigma, N, S, R)$ gibt es eine kontextfreie Grammatik $G' = (\Sigma, N', S', R')$, die dieselbe Sprache erzeugt und für die gilt:

Ist $\varepsilon \in L(G)$, so enthält R' die Ableitungsregel $S' \rightarrow \varepsilon$, und S' kommt auf keiner rechten Seite einer Ableitungsregel vor. Ist $\varepsilon \notin L(G)$, dann gilt für alle Ableitungsregeln $A \rightarrow w$ in R' : $w \in (\Sigma \cup N')^+$.

Die Klasse der kontextfreien Sprachen (Typ-1-Sprachen) über einem endlichen Alphabet Σ ist eine echte Teilmenge der kontextsensitiven Sprachen über Σ .

Beweis:

Es sei $N_\varepsilon = \{B \mid B \in N \text{ und } B \Rightarrow^* \varepsilon\}$. Die Menge N_ε lässt sich leicht bestimmen, indem man zunächst alle Nichtterminalzeichen A in N_ε aufnimmt, für die es eine Ableitungsregel $A \rightarrow \varepsilon$ in R gibt; anschließend werden solange zusätzliche Nichtterminalzeichen A zu N_ε hinzugefügt, für die es eine Ableitungsregel $A \rightarrow B_1 \dots B_k$ mit $k \geq 1$ und $B_i \in N_\varepsilon$ für $i = 1, \dots, k$ gibt, bis keine Änderung an N_ε mehr auftritt.

Jede Ableitungsregel in R lässt sich in der Form $A \rightarrow \alpha_0 B_1 \alpha_1 \dots B_k \alpha_k$ mit $k \geq 0$ und $B_i \in N_\varepsilon$ schreiben; dabei enthalten die Zeichenketten α_i keine Zeichen aus N_ε . Für jede dieser Ableitungsregeln wird $A \rightarrow \alpha_0 X_1 \alpha_1 \dots X_k \alpha_k$ mit $X_i = \varepsilon$ oder $X_i = B_i$ in R' aufgenommen, wobei $A \rightarrow \varepsilon$ nicht zugelassen ist; $X_i = B_i$ ist erforderlich, da eventuell $B_i \Rightarrow^* w_i$ mit $w_i \neq \varepsilon$ möglich ist.

Es sei $\varepsilon \in L(G)$. Dieses trifft genau dann zu, wenn $S \in N_\varepsilon$ ist. Es sei S' ein neues Nichtterminalzeichen. In R' werden zusätzlich die Ableitungsregeln $S' \rightarrow S$ und $S' \rightarrow \varepsilon$ aufgenommen und $N' = N \cup \{S'\}$ gesetzt.

Ist $\varepsilon \notin L(G)$, so ist $N' = N$ und $S' = S$.

Nach Konstruktion gilt $L(G) = L(G')$.

Die kontextfreie Grammatik G' erfüllt die Bedingungen, die an eine kontextsensitive Grammatik gestellt werden. Daher ist die Klasse der kontextfreien Sprachen über einem endlichen Alphabet Σ in der Klasse der kontextsensitiven Sprachen über Σ enthalten ist. Dass diese Inklusion echt ist, wird in den Beispielen im Anschluss an Satz 4.4-3 gezeigt.

///

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 $S \rightarrow \varepsilon$ oder $A \rightarrow BC$ mit $A \in N$, $B \in N$, $C \in N$ oder $A \rightarrow a$ mit $A \in N$, $a \in \Sigma$ haben (**Chomsky-Normalform**) und dabei dieselbe Sprache erzeugt wird.

Beweis:

Es sei $G = (\Sigma, N, S, R)$ eine kontextfreie Grammatik, von der gemäß Satz 4.4-1 bereits folgendes angenommen werden kann: Ist $\varepsilon \in L(G)$, so enthält R die Ableitungsregel $S \rightarrow \varepsilon$, und S kommt auf keiner rechten Seite einer Ableitungsregel vor. Ist $\varepsilon \notin L(G)$, dann gilt für alle Ableitungsregeln $A \rightarrow w$ in R : $w \in (\Sigma \cup N)^+$. Enthält die Menge der Ableitungsregeln einen Zyklus $B_1 \rightarrow B_2, B_2 \rightarrow B_3, \dots, B_k \rightarrow B_1$, so wird ein neues Nichtterminalsymbol B eingeführt, jedes Vorkommen der Nichtterminalsymbole B_i für $i = 1, \dots, k$ in Ableitungsregeln aus R durch B ersetzt und alle Ableitungsregeln der Form $B \rightarrow B$ und die Nichtterminalsymbole B_i für $i = 1, \dots, k$ aus N entfernt. Die so entstandene kontextfreie Grammatik sei $G' = (\Sigma, N', S', R')$. Offensichtlich gilt $L(G) = L(G')$.

Die Nichtterminalsymbole in $N' = \{A_1, \dots, A_n\}$ lassen sich jetzt so numerieren, dass für jede Ableitungsregel der Form $A_i \rightarrow A_j$ in R' $i < j$ ist. Es seien $A_{i_1} \rightarrow A_{i_2}, \dots, A_{i_{k-1}} \rightarrow A_{i_k}$ alle Ableitungsregeln in R' , die auf der rechten Seite nur ein einziges Nichtterminalsymbol besitzen. Für $j = k-1, \dots, 1$ (absteigend) werden nacheinander alle Ableitungsregeln $A_{i_j} \rightarrow A_{i_{j+1}}$ betrachtet: Enthält R' Ableitungsregeln der Form $A_{i_{j+1}} \rightarrow x_1, \dots, A_{i_{j+1}} \rightarrow x_l$ (die Zeichenketten x_i bestehen wegen der Numerierung nicht nur aus einem einzigen Nichtterminalsymbol), so wird $A_{i_j} \rightarrow A_{i_{j+1}}$ aus R' entfernt und $A_{i_j} \rightarrow x_1, \dots, A_{i_j} \rightarrow x_l$ in die Menge der Ableitungsregeln aufgenommen. Nichtterminalsymbole, die in keiner Ableitung vorkommen können,

werden zusätzlich aus N' entfernt. Die so entstandene kontextfreie Grammatik sei $G'' = (\Sigma, N'', S'', R'')$. Offensichtlich gilt $L(G) = L(G') = L(G'')$.

Alle Ableitungsregeln in R'' , die noch nicht den Bedingungen der Chomsky-Normalform genügen, haben jetzt die Form $A \rightarrow X_1 X_2 \dots X_m$ mit $m \geq 2$ und $X_i \in N'' \cup \Sigma$ für $i = 1, \dots, m$. Ist $X_i \in \Sigma$, so werden ein neues Nichtterminalsymbol C_{X_i} und eine neue Ableitungsregel $C_{X_i} \rightarrow X_i$ eingeführt und X_i in obiger Ableitungsregel durch C_{X_i} ersetzt. Alle Ableitungsregeln, die jetzt noch nicht den Bedingungen der Chomsky-Normalform genügen, haben die Form $A \rightarrow B_1 B_2 \dots B_m$ mit $m \geq 3$ und Nichtterminalsymbolen B_i für $i = 1, \dots, m$. Für eine derartige Ableitungsregel werden neue Nichtterminalsymbole D_1, \dots, D_{m-2} eingeführt, und die Ableitungsregel wird durch die neuen Ableitungsregeln $A \rightarrow B_1 D_1$, $D_1 \rightarrow B_2 D_2$, \dots , $D_{m-3} \rightarrow B_{m-2} D_{m-2}$, $D_{m-2} \rightarrow B_{m-1} B_m$ ersetzt. Die so entstandene kontextfreie Grammatik sei $G''' = (\Sigma, N''', S''', R''')$. Offensichtlich liegt diese Grammatik in Chomsky-Normalform vor, und es gilt $L(G) = L(G') = L(G'') = L(G''')$.

///

Der folgende Satz (**uvwxy-Theorem, pumping lemma**) liefert ein Beweismittel, mit dessen Hilfe man zeigen kann, dass eine Sprache nicht kontextfrei ist.

Satz 4.4-3:

Zu jeder kontextfreien Sprache L gibt es eine Zahl $n_0 > 0$ mit der Eigenschaft:

jedes $z \in L$ mit $|z| \geq n_0$ lässt sich so in $z = uvwxy$ zerlegen, dass die folgenden Eigenschaften (i), (ii) und (iii) gelten:

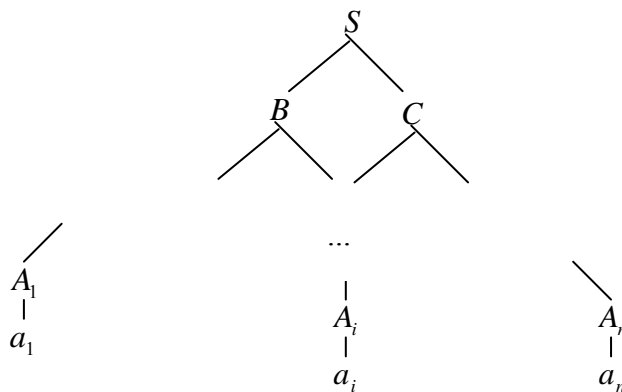
- (i) $|vwx| \leq n_0$
- (ii) $|vx| > 0$
- (iii) $uv^k wx^k y \in L$ für jedes $k \in \mathbf{N}$.

Beweis:

Es sei $G = (\Sigma, N, S, R)$ eine kontextfreie Grammatik in Chomsky-Normalform mit $L = L(G)$ und $n_0 = 2^{|\mathbf{N}|+1}$.

Für ein Wort $z \in L$, etwa $z = a_1 \dots a_n$ mit $a_1 \in \Sigma, \dots, a_n \in \Sigma$ und $|z| = n \geq n_0$, hat eine Ableitung aus S die Form $S \Rightarrow BC \Rightarrow \dots \Rightarrow a_1 \dots a_n$. 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 $A \rightarrow BC$ 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 $A \rightarrow a$ 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 besitzt und der Anwendung einer Regel der Form $A \rightarrow BC$ entspricht. Die Blätter des Baums sind mit a_1, \dots, a_n markiert. Ein mit a_i markiertes Blatt mit seinem mit A_i markierten Vorgänger entspricht der Anwendung der Regel $A_i \rightarrow a_i$. Nur an den Blättern werden Regeln dieser Form angewendet.



Für die Anzahl n der Blätter dieses Ableitungsbaums gilt $n = |z| \geq n_0 = 2^{|N|+1}$. Daher hat der Baum nach Satz 1.1-12 (iv) eine Mindesthöhe von $\lceil \log_2(n) + 1 \rceil \geq \log_2(2^{|N|+1}) + 1 = |N| + 2$. Es gibt also einen Pfad von der Wurzel zu einem Blatt, das mit einem Terminalzeichen a_i markiert ist, auf dem mindestens $|N| + 1$ viele innere Knoten liegen, die mit Nichtterminalsymbolen markiert sind. Da die Grammatik nur $|N|$ viele Nichtterminalsymbole enthält, kommt auf diesem Pfad ein $A \in N$ mindestens zweimal vor. Es kann natürlich mehrere Pfade geben, die die Eigenschaft besitzen, mehrere gleiche Nichtterminalsymbole zu enthalten. Es werde ein Pfad P betrachtet, in dem das zweite Auftreten eines Nichtterminalsymbols möglichst weit von der Wurzel des Ableitungsbaums entfernt liegt. Dieses zweite Auftreten des Nichtterminalsymbols A auf dem Pfad P finde auf Niveau k_2 statt, das Auftreten davor auf dem Niveau k_1 . Zwischen dem Niveau k_1 dem Niveau k_2 enthält der Pfad P das Nichtterminalsymbol A nicht. Ist das Niveau k_1 komplett aufgefüllt, so hat es die Form $w_1 A w_2$, und in G ist die Ableitung $S \Rightarrow^* w_1 A w_2 \Rightarrow^* z$ möglich. Ist das Niveau k_1 nicht komplett aufgefüllt, so entstammen die Symbole in w_1 bzw. in w_2 eventuell niedrigeren Niveaus links bzw. rechts der Position von A . Das Niveau k_2 hat entsprechend die Form $v_1 A v_2$. Man betrachte den Teilbaum des Ableitungsbaums, der auf Niveau k_1 mit dem Symbol A auf Pfad P beginnt und dem Pfad P folgend auf Niveau k_2 endet. Dieser Teilbaum entspricht einer Ableitung $A \Rightarrow^+ w_3 A w_4$. Da G kontextfrei ist, kann die Ableitung $S \Rightarrow^* z$ so organisiert werden, dass gilt:

$S \Rightarrow^* w_1 A w_2 \Rightarrow^+ w_1 w_3 A w_4 w_2 \Rightarrow^* uvwxy = z$ mit $w_1 \Rightarrow^* u$, $w_3 \Rightarrow^* v$, $A \Rightarrow^* w$, $w_4 \Rightarrow^* x$ und $w_2 \Rightarrow^* y$, und in der Teibleitung $w_1 A w_2 \Rightarrow^+ w_1 w_3 A w_4 w_2$ wird in keinem Ableitungsschritt, bis auf den ersten, das Nichtterminalsymbol A ersetzt. Ebenso werden nach Wahl von k_2 in jedem Ableitungsschritt der Teibleitungen $w_3 \Rightarrow^* v$, $A \Rightarrow^* w$ und $w_4 \Rightarrow^* x$ jeweils von A verschiedene Nichtterminalsymbole 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:

$$S \Rightarrow^* w_1 A w_2 \Rightarrow^* u A y \Rightarrow^+ u w_3 A w_4 y \Rightarrow^* u v A x y \Rightarrow^* u v w x y .$$

Der erste Schritt in der Teibleitung $u A y \Rightarrow^+ u w_3 A w_4 y$ erfolgt durch Anwendung einer Regel der Form $A \rightarrow BC$, daher ergibt sich

$$u A y \Rightarrow u B C y \Rightarrow^* u w_3 A w_4 y \text{ und } B C \Rightarrow^* w_3 A w_4 \Rightarrow^* v A x .$$

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 Teibleitung $A \Rightarrow^* v w x$ 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 $|v w x| \leq 2^{|N|} < n_0$ (Eigenschaft (i)).

Die in G mögliche Ableitung $A \Rightarrow^* v A x$ kann man beliebig oft in die Ableitung $S \Rightarrow^* u v w x y$ einbauen und mit der Ableitung $A \Rightarrow^* w$ kombinieren:

$S \Rightarrow^* u A y \Rightarrow^* u v A x y \Rightarrow^* u v v A x x y \Rightarrow^* u v^k A x^k y \Rightarrow^* u v^k w x^k y$ für $k \geq 1$; außerdem ist die Ableitung $S \Rightarrow^* u A y \Rightarrow^* u w y$ möglich. Insgesamt ist dieses die Eigenschaft (iii).

///

Mit Hilfe dieses Satzes lässt sich zeigen, dass die Sprache $L_4 = \{a^n b^n c^n \mid n \in \mathbf{N}, n \geq 1\}$ (aus Kapitel 4.1) nicht kontextfrei (aber kontextsensitiv ist). Es sei $n = n_0$. Für das Wort $z = a^n b^n c^n$ ist $|z| = 3n_0 \geq n_0$. 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 $uv^0 w x^0 y \in L_4$ ist, d.h. $uwy \in L_4$. Es werden zwei Fälle unterschieden:

1. Fall: vwx enthält kein Zeichen c . Dann ist $uvw x = a^n b^m$, $y = b^{n-m} c^n$. Da $|v x| > 0$ ist, enthält $v x$ $k \geq 1$ 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: $vw x$ enthält mindestens ein Zeichen c . Wegen $|vw x| \leq n_0$ enthält es kein Zeichen a . Es ist $u = a^n b^m$, $vw x = b^{n-m} c^l$, $y = c^{n-l}$ mit $l \geq 1$. Da $|vx| > 0$ ist, enthält vx $k \geq 1$ Zeichen b oder c . Das Wort $uw y$ enthält n Zeichen a , $m+n-m+l-k = n+l-k$ Zeichen b oder c und $n-l$ Zeichen c . Daher ist $uw y \notin L_4$.

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 $f : \mathbf{N} \rightarrow \mathbf{N}$ eine Funktion mit einem Wachstumsverhalten, das für jedes $n \in \mathbf{N}$ durch $f(n+1) - f(n) > n$ beschrieben wird. Es sei a ein (terminales) Symbol. 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; $f(1) > 0 + f(0) \geq 0$, also $f(1) \geq 1$; mit der Induktionsvoraussetzung $f(n) \geq n$ für $n \geq 1$ 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 $f(n+1) > f(n)$.

Es ist $f(n+1) > f(n) + n \geq f(n)$.

(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 $z = uvwx y$ mit den angegebenen Eigenschaften zerlegen lässt. Insbesondere enthält L auch das Wort uv^2wx^2y , für dessen Länge $f(n_0) < |uv^2wx^2y| = f(n_0) + |vx| \leq f(n_0) + n_0 < f(n_0+1)$ gilt. Wegen Eigenschaft (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 und die Annahme, L sei kontextfrei, falsch.

///

Satz 4.4-4 zeigt, dass die kontextsensitiven Sprachen $L_1 = \{a^{2^n} \mid n \in \mathbf{N}_{>0}\}$ und $L_2 = \{a^{n^2} \mid n \in \mathbf{N}\}$ 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

$VERIFIZIERE_G_TYP_2(w)$

$$= \begin{cases} \text{TRUE} & \text{falls } w \text{ die Kodierung einer Typ-2-Grammatik darstellt} \\ \text{FALSE} & \text{falls } w \text{ nicht die Kodierung einer Typ-2-Grammatik darstellt} \end{cases}$$

definiert. Dieses Prädikat ist berechenbar. Dazu ist unter anderem zu prüfen, ob die Ableitungsregeln in KG_w den Bedingungen einer kontextfreien Grammatik genügen.

Das **Wortproblem für Typ-2-Grammatiken** fragt danach, ob die Menge

$$L_{\text{Wort_Typ-2}} = \left\{ u \# w \mid u \in \Sigma^*, w \in \{0, 1\}^*, VERIFIZIERE_G_TYP_2(w) = \text{TRUE} \text{ und } u \in L(KG_w) \right\}$$

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 wegen des exponentiellen Laufzeitverhaltens zu aufwendig. In der Theorie des Compilerbaus, die sich u.a. intensiv mit der Frage beschäf-

tigt, ob ein Wort zur Sprache einer gegebenen Grammatik gehört, wird gezeigt, dass die Frage sogar mit einem Zeitaufwand der Ordnung $O(|u|^3)$ bei gegebener Grammatik G entschieden werden kann. Auf Details soll hier mit dem Verweis die Grundlagenliteratur zum Compilerbau¹⁷ verzichtet werden.

Das **Leerheitsproblem**, das für Typ-0- und Typ-1-Grammatiken nicht entscheidbar ist, fragt für **Typ-2-Grammatiken**, ob die Menge

$$L_{\text{leer_Typ-2}} = \left\{ w \mid w \in \{0, 1\}^*, \text{VERIFIZIERE_G_TYP_2}(w) = \text{TRUE} \text{ und } L(KG_w) = \emptyset \right\}$$

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 $L(G) = \emptyset$ gilt oder nicht.

Der folgende (Pseudocode-) Algorithmus entscheidet bei Eingabe einer kontextfreien Grammatik die $G = (\Sigma, N, S, R)$ die Frage „ $L(G) \neq \emptyset$?“. Aus diesem Algorithmus lässt sich leicht ein Entscheidungsalgorithmus für das Leerheitsproblem für Typ-2-Grammatiken gewinnen.

Eingabe: Eine kontextfreie Grammatik $G = (\Sigma, N, S, R)$

Verfahren: Aufruf der Funktion `ist_nichtleer (G)`

Ausgabe: TRUE, falls $L(G) \neq \emptyset$, 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 Ableitungsregel mit w ∈ (V ∪ Σ)* } ∪ V;
```

```
    END;
```

¹⁷ Siehe: Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.: **The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing**, Addison-Wesley, 1972.

```

IF  $S \in W$  THEN ist_nichtleer := TRUE
      ELSE ist_nichtleer := FALSE;
END   { ist_nichtleer };

```

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 $L(G) = \emptyset$ gilt.

Beweis:

Die Entscheidbarkeit des Wortproblems für Typ-2-Grammatiken wird in den obigen Bemerkungen begründet.

Zur Entscheidung des Leerheitsproblems für eine gegebene kontextfreie Grammatik G wird die oben angegebene Funktion `ist_nichtleer` verwendet, und zwar ist $L(G) = \emptyset$, falls `ist_nichtleer(G) = FALSE` ist. Das Verfahren bricht ab, da die `WHILE`-Schleife höchstens n -mal durchlaufen wird, falls G n Ableitungsregeln enthält. Die Korrektheit des Verfahrens ergibt sich aus folgender Behauptung:

Ein Nichtterminalsymbol $A \in N$ wird im i -ten Durchlauf der `WHILE`-Schleife genau dann in w aufgenommen, wenn es ein $w \in \Sigma^*$ gibt mit $A \Rightarrow^* w$.

Der Beweis der Behauptung enthält zwei Richtungen, die – hier exemplarisch für viele Argumentationsketten in der Theorie der Formalen Sprachen – über vollständige Induktion bewiesen werden:

Das Nichtterminalsymbol $A \in N$ werde im i -ten Durchlauf der `WHILE`-Schleife in w aufgenommen.

Für $i = 0$ ist $A \rightarrow w$ mit $w \in \Sigma^*$ eine Ableitungsregel, d.h. $A \Rightarrow^* w$.

A werde im $(i+1)$ -ten Durchlauf der `WHILE`-Schleife in w aufgenommen. Dann ist $A \rightarrow X_1 \dots X_k$ eine Ableitungsregel mit $X_j \in \Sigma$, oder X_j wurde in w in einem vorhergehenden `WHILE`-Schleifendurchlauf aufgenommen ($j = 1, \dots, k$). Nach Induktionsvoraussetzung gibt es für $j = 1, \dots, k$ Zeichenketten $w_j \in \Sigma^*$ mit $X_j \Rightarrow^* w_j$ (für $X_j \in \Sigma$ wird $w_j = X_j$ genommen). Insgesamt gilt $A \Rightarrow X_1 \dots X_k \Rightarrow^* w_1 \dots w_k$.

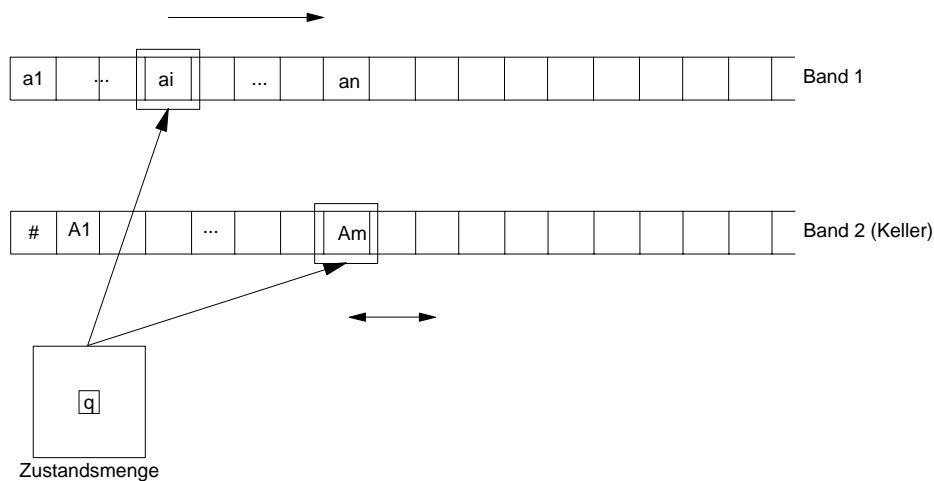
Umgekehrt gelte $A \Rightarrow^* w$ mit $w \in \Sigma^*$. Der Beweis erfolgt durch Induktion über die Länge der Ableitung:

Gilt $A \Rightarrow w$, so ist $A \rightarrow w$ eine Ableitungsregel mit $w \in \Sigma^*$, und A wird vor dem ersten Durchlauf der WHILE-Schleife in w aufgenommen ($i = 0$). Hat die Ableitung $A \Rightarrow^* w$ die Länge $i+1$, so hat sie die Form $A \Rightarrow w_1 \Rightarrow^* w$ mit einer Ableitungsregel $A \rightarrow w_1$. Es ist $w_1 = X_1 \dots X_k$ mit $X_j \in \Sigma \cup N$ und $X_j \Rightarrow^* w_j$ mit höchstens i Ableitungsschritten für $j=1, \dots, k$ und $w = w_1 \dots w_k$. Ist $X_j \in N$, so wurde $X_j \in N$ in einem Durchlauf der WHILE-Schleife in w mit einer Nummer i_j aufgenommen. Mit $m = \max\{i_j \mid j=1, \dots, k\}$ wird A spätestens im $(m+1)$ -ten Durchlauf der WHILE-Schleife in w aufgenommen.

///

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 dieses eine ε -Überführung. 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. $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow \mathbf{P}_e(Q \times \Gamma^*)$ ist eine partielle Funktion, die **Überföhrungsfunktion**; 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 $w = w_1w_2$ 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 $(w_1w_2, |w_1|+1)$ 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 $w = a_1 \dots a_n$ noch nicht gelesen wurden, dass der Lesekopf des 1. Bandes über dem ersten Zeichen a_1 steht, dass sich im Keller die Zeichenkette $\alpha = A_1 \dots A_m$ 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 $(q', B_1 \dots B_k)$ 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 $B_1 \dots B_k$ ersetzt und den Schreiblesekopf im Keller auf B_k positioniert hat. Formal wird für Konfigurationen die Übergangsrelation \Rightarrow definiert durch

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

Entsprechend bedeutet für Konfigurationen K und K' die Beziehung $K \Rightarrow^* K'$:

Es gibt Konfigurationen K_0, K_1, \dots, K_l mit $l \geq 0$ und $K = K_0$, $K' = K_l$ und $K_i \Rightarrow K_{i+1}$ für $i = 0, \dots, l-1$.

Eine **Anfangskonfiguration für einen nichtdeterministischen Kellerautomaten NKA** hat die Form $(q_0, w, \#)$, eine **Endkonfiguration** hat die Form (q, ε, α) mit $q \in F$ und $\alpha \in \Gamma^*$.

Ein Wort $w \in \Sigma^*$ wird von **NKA akzeptiert**, wenn $(q_0, w, \#) \Rightarrow^* (q, \varepsilon, \alpha)$ mit $q \in F$ und $\alpha \in \Gamma^*$ gilt. Die von **NKA akzeptierte Sprache** ist

$$L(NKA) = \{ w \mid w \in \Sigma^*, \text{ und } w \text{ wird von NKA akzeptiert} \}.$$

Ein nichtdeterministischer Kellerautomat zum Erkennen der Sprache

$$L = \{ w \mid w = a_1 \dots a_m a_m \dots a_1, a_i \in \{ a, b \} \}$$

Die Sprache $L = \{ w \mid w = a_1 \dots a_m a_m \dots a_1, a_i \in \{ a, b \} \}$ lässt sich beispielsweise durch die kontextfreie Grammatik $G = (\{ a, b \}, \{ S \}, S, \{ S \rightarrow aSa, S \rightarrow bSb, S \rightarrow aa, S \rightarrow bb \})$ 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 Überfunktionsfunktion δ (mit dem Zeichen „ b “ ist in diesem Beispiel das Terminalsymbol b gemeint und nicht das Blankzeichen):

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“.

Zu beachten ist, dass nicht gefordert wird, dass in einer Endkonfiguration der Keller geleert ist; wichtig ist allein, dass ein Endzustand $q \in F$ erreicht wird. Alternativ kann man definieren: Ein Wort $w \in \Sigma^*$ wird von NKA **mit leerem Keller akzeptiert**, wenn $(q_0, w, \#) \Rightarrow^* (q, \varepsilon, \varepsilon)$ mit $q \in Q$. In diesem Fall setzt man $F = \emptyset$. Die von NKA **mit leerem Keller akzeptierte Sprache** ist

$L_\varepsilon(NKA) = \{w \mid w \in \Sigma^*, \text{ und } w \text{ wird von } NKA \text{ mit leerem Keller akzeptiert}\}$. Der folgende Satz zeigt, dass sich beide Definitionen in ihrer Wirkung auf das Akzeptanzverhalten von Sprachen nicht unterscheiden.

Satz 4.4-6:

- (i) Es sei NKA ein nichtdeterministischer Kellerautomat. Dann gibt es einen nichtdeterministischen Kellerautomaten NKA' mit $L_\varepsilon(NKA') = L(NKA)$.
- (ii) Es sei $NKA = (Q, \Sigma, \Gamma, \delta, q_0, \#, \emptyset)$ ein nichtdeterministischer Kellerautomat. Dann gibt es einen nichtdeterministischen Kellerautomaten NKA' mit $L(NKA') = L_\varepsilon(NKA)$.

Beweis:

Zu (i): NKA sei definiert durch $NKA = (Q, \Sigma, \Gamma, \delta, q_0, \#, F)$. Der nichtdeterministische Kellerautomat NKA' wird gegeben durch

$NKA' = (Q \cup \{q_\varepsilon, q'_0\}, \Sigma, \Gamma \cup \{\#\}, \delta', q'_0, \#, \emptyset)$. Hierbei sind q_ε und q'_0 neue Zustände, die in Q nicht vorkommen, und $\#$ ist ein neues Symbol, das in Γ nicht vorkommt. Die Überföhrungsfunktion δ' wird durch folgende Festlegungen gegeben:

- (1) für jedes (z, α) in $\delta(q, a, A)$ ist (z, α) in $\delta'(q, a, A)$
- (2) $\delta'(q'_0, \varepsilon, \#) = \{(q_0, \#\#\}$
- (3) für jedes $q \in Q$ und für jedes $A \in \Gamma \cup \{\#\}$ ist $(q_\varepsilon, \varepsilon)$ in $\delta'(q, \varepsilon, A)$
- (4) für jedes $A \in \Gamma \cup \{\#\}$ ist $(q_\varepsilon, \varepsilon)$ in $\delta'(q_\varepsilon, \varepsilon, A)$.

NKA' schreibt zunächst das Kellersymbol $\#$ von NKA in seinen Keller, ohne ein Eingabesymbol zu lesen (Festlegung (2)), und simuliert dann das Verhalten von NKA (Festlegung (1)). Mit der Festlegung (3) geht NKA' in den Zustand q_ε über, in dem der Keller geleert wird (Festlegung (4)).

In NKA gilt genau dann $(q_0, w, \#) \Rightarrow^* (q, \varepsilon, \alpha A)$ mit $q \in F$, $A \in \Gamma$ und $\alpha \in \Gamma^*$ (d.h. $w \in L(NKA)$), und w wird akzeptiert, ohne dass der Keller geleert wird), wenn in NKA' gilt:

$$(q'_0, w, \#) \Rightarrow (q_0, w, \#\#\} \Rightarrow^* (q, \varepsilon, \#\alpha A) \Rightarrow (q_\varepsilon, \varepsilon, \#\alpha) \Rightarrow^* (q_\varepsilon, \varepsilon, \varepsilon).$$

Außerdem gilt in NKA genau dann $(q_0, w, \#) \Rightarrow^* (q, \varepsilon, \varepsilon)$ mit $q \in F$ (d.h. $w \in L(NKA)$), wobei der Keller leer ist), wenn in NKA' gilt:

$$(q'_0, w, \#) \Rightarrow (q_0, w, \#\#\} \Rightarrow^* (q, \varepsilon, \#\} \Rightarrow (q_\varepsilon, \varepsilon, \varepsilon).$$

Beide Aussagen zusammen zeigen $L_\varepsilon(NKA') = L(NKA)$. Hierbei wird wesentlich genutzt, dass NKA' seinen Keller nur im Zustand q_ε leeren kann; selbst wenn NKA seinen Keller leert, bleibt bei NKA' noch das Kellersymbol $\#'$ erhalten, solange sich NKA' noch in der Simulationsphase von NKA befindet.

Zu (ii): Der nichtdeterministische Kellerautomat NKA' wird gegeben durch

$NKA' = (Q \cup \{q'_0, q_f\}, \Sigma, \Gamma \cup \{\#'\}, \delta', q'_0, \#', \{q_f\})$. Hierbei sind q_f und q'_0 neue Zustände, die in Q nicht vorkommen, und $\#'$ ist ein neues Symbol, das in Γ nicht vorkommt. Die Überföhrungsfunktion δ' wird durch folgende Festlegungen gegeben:

- (1) für jedes (z, α) in $\delta(q, a, A)$ ist (z, α) in $\delta'(q, a, A)$
- (2) $\delta'(q'_0, \varepsilon, \#') = \{(q_0, \#\#\}$
- (3) für jedes $q \in Q$ ist (q_f, ε) in $\delta'(q, \varepsilon, \#')$.

NKA' schreibt zunächst das Kellersymbol $\#$ von NKA in seinen Keller, ohne ein Eingabesymbol zu lesen (Festlegung (2)), und simuliert dann das Verhalten von NKA (Festlegung (1)). Sobald NKA seinen Keller geleert hat, wird die Eingabe von NKA' akzeptiert (Festlegung (3)).

Der Beweis der Aussage $L(NKA') = L_\varepsilon(NKA)$ verläuft wie in Teil (i).

///

Der folgende Satz zeigt den Zusammenhang zwischen kontextfreien Sprachen und den von nichtdeterministischen Kellerautomaten akzeptierten Sprachen.

Satz 4.4-7:

- (i) Es sei G eine kontextfreie Grammatik. Dann gibt es einen nichtdeterministischen Kellerautomaten NKA_G mit $L_\varepsilon(NKA_G) = L(G)$.
- (ii) Es sei NKA ein nichtdeterministischer Kellerautomat. Dann gibt es eine kontextfreie Grammatik G_{NKA} gibt mit $L(G_{NKA}) = L_\varepsilon(NKA)$.

Die Klasse der von nichtdeterministischen Kellerautomaten akzeptierten Sprachen über einem endlichen Alphabet Σ ist also mit der Klasse der kontextfreien Sprachen (Typ-2-Sprachen) über Σ identisch.

Beweis:

Wegen Satz 4.4-6 ist bei einem Kellerautomaten die Akzeptanz nach der ursprünglichen Definition und die Akzeptanz mit leerem Keller äquivalent, so dass die letzte Aussage des Satzes gerechtfertigt ist.

Zu (i): Es sei $G = (\Sigma, N, S, R)$ eine kontextfreie Grammatik. Die Überföhrungsfunktion δ des nichtdeterministischen Kellerautomaten $NKA_G = (\{z\}, \Sigma, N \cup \Sigma, \delta, z, S, \emptyset)$ mit $L_\varepsilon(NKA_G) = L(G)$ wird durch folgende Festlegungen gegeben:

- (1) für jede Ableitungsregel $A \rightarrow \alpha$ aus R wird (z, α^R) in $\delta(z, \varepsilon, A)$ aufgenommen; hierbei ist α^R die Konkation der Buchstaben von α in umgekehrter Reihenfolge (Spiegelung von α)
- (2) für jedes $a \in \Sigma$ wird (z, ε) in $\delta(z, a, a)$ aufgenommen.

Wegen (1) 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.

Durch Induktion über die Länge der Ableitung in G bzw. über die Anzahl der Überföhrungen in NKA_G lässt sich zeigen:

Es gilt genau dann $S \Rightarrow^* w$ mit $w \in \Sigma^*$, wenn $(z, w, S) \Rightarrow^* (z, \varepsilon, \varepsilon)$ gilt.

Daraus folgt $L_\varepsilon(NKA_G) = L(G)$.

Zu (ii): NKA sei gegeben durch $NKA = (Q, \Sigma, \Gamma, \delta, q_0, \#, F)$. Es wird eine Menge N von Nichtterminalzeichen durch $N = \{[q A q'] \mid q \in Q, A \in \Gamma, q' \in Q\} \cup \{S\}$ definiert. Die folgenden Festlegungen definieren die Menge R der Ableitungsregeln der kontextfreien Grammatik $G = (\Sigma, N, S, R)$:

- (1) für $(q', A_1 \dots A_k) \in \delta(q, a, A)$ mit $k \geq 1$ sind für alle $q_1 \in Q, \dots, q_k \in Q$ die Konstrukte $[q A q_1] \rightarrow a[q' A_k q_k][q_k A_{k-1} q_{k-1}] \dots [q_2 A_1 q_1]$ Ableitungsregeln in R
- (2) für $(q', \varepsilon) \in \delta(q, a, A)$ ist $[q A q'] \rightarrow a$ eine Ableitungsregel in R
- (3) für jedes $q \in Q$ ist $S \rightarrow [q_0 \# q]$ in R .

Dann gilt: In G ist die Ableitung $[q A p] \Rightarrow^* w$ mit $w \in \Sigma^*$ genau dann möglich, wenn es in NKA eine Folge von Konfigurationsübergängen $(q, w, A) \Rightarrow^* (p, \varepsilon, \varepsilon)$ gibt.

Der Beweis erfolgt über Induktion über die Länge der Ableitung bzw. über die Anzahl der Konfigurationsübergänge. Hier soll exemplarisch nur die Beweisrichtung „von links nach rechts“ gezeigt werden:

Die Ableitung $[q A p] \Rightarrow^* w$ mit $w \in \Sigma^*$ in G habe die Länge m .

Für $m=1$ ist $[q A p] \Rightarrow w$, d.h. $[q A p] \rightarrow w$ ist eine Ableitungsregel, die nur gemäß (2) entstanden sein kann; daher ist $w \in \Sigma \cup \{ \varepsilon \}$ und $(p, \varepsilon) \in \delta(q, w, A)$. Also ist in NKA der Konfigurationsübergang $(q, w, A) \Rightarrow (p, \varepsilon, \varepsilon)$ möglich.

Die Behauptung gelte für alle Ableitungen in G bis zur Länge $m-1$. Der erste Schritt der Ableitung $[q A p] \Rightarrow^* w$ erfolgt mit einer Ableitungsregel, die gemäß (1) entstanden ist, d.h. mit einer Regel der Form

$[q A p] \rightarrow a[q' A_k q_k][q_k A_{k-1} q_{k-1}] \dots [q_2 A_1 p]$. Da diese Regel in R aufgenommen wurde, ist $(q', A_1 \dots A_k) \in \delta(q, a, A)$. Die Ableitung $[q A p] \Rightarrow^* w$ in G hat also die Form $[q A p] \Rightarrow a[q' A_k q_k][q_k A_{k-1} q_{k-1}] \dots [q_2 A_1 p] \Rightarrow^* w$, w hat die Form

$w = aw_k w_{k-1} \dots w_1$, und es gilt in G : $[q' A_k q_k] \Rightarrow^* w_k$, $[q_k A_{k-1} q_{k-1}] \Rightarrow^* w_{k-1}$, ..., $[q_2 A_1 p] \Rightarrow^* w_1$ mit jeweils höchstens $m-1$ vielen Ableitungsschritten, so dass auf diese Ableitungen die Induktionsvoraussetzung anwendbar ist. Daher gibt es in NKA

Folgen von Konfigurationsübergänge $(q', w_k, A_k) \Rightarrow^* (q_k, \varepsilon, \varepsilon)$,

$(q_k, w_{k-1}, A_{k-1}) \Rightarrow^* (q_{k-1}, \varepsilon, \varepsilon)$, ..., $(q_2, w_1, A_1) \Rightarrow^* (p, \varepsilon, \varepsilon)$. Insgesamt ist in NKA möglich:

$$\begin{aligned} (q, aw_k \dots w_1, A) &\Rightarrow (q', w_k \dots w_1, A_1 \dots A_k) \Rightarrow^* (q_k, w_{k-1} \dots w_1, A_1 \dots A_{k-1}) \\ &\Rightarrow^* (q_{k-1}, w_{k-2} \dots w_1, A_1 \dots A_{k-2}) \\ &\Rightarrow^* (q_2, w_1, A_1) \\ &\Rightarrow^* (p, \varepsilon, \varepsilon). \end{aligned}$$

Es sei $w \in L(G)$, d.h. $S \Rightarrow [q_0 \# q] \Rightarrow^* w$. In NKA gilt dann $(q_0, w, \#) \Rightarrow^* (q, \varepsilon, \varepsilon)$, d.h. $w \in L_\varepsilon(NKA)$. Die umgekehrte Inklusion $L_\varepsilon(NKA) \subseteq L(G)$ zeigt man entsprechend.

///

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 Ableitungsmenge $R = \{S \rightarrow A, A \rightarrow aAb, A \rightarrow c\}$. Die in der Konstruktion des ersten Teils des Beweises von Satz 4.4-7 beschriebene Überföhrungsfunktion 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:

$$\begin{aligned}
 (z, aacbb, S) &\Rightarrow (z, aacbb, A) \Rightarrow (z, aacbb, bAa) \Rightarrow (z, acbb, bA) \Rightarrow (z, acbb, bbAa) \\
 &\Rightarrow (z, cbb, bbA) \Rightarrow (z, cbb, bbc) \Rightarrow (z, bb, bb) \Rightarrow (z, b, b) \\
 &\Rightarrow (z, \varepsilon, \varepsilon).
 \end{aligned}$$

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 Überföhrungsfunktion die Form

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

aufweist und die beiden Bedingung (i) und (ii) gelten:

- (i) Ist $\delta(q, a, A)$ für $q \in Q$ und $a \in \Sigma$ definiert, dann ist $\delta(q, \varepsilon, A)$ nicht definiert,
- (ii) ist $\delta(q, \varepsilon, A)$ für $q \in Q$ definiert, dann ist $\delta(q, a, A)$ für $a \in \Sigma$ nicht definiert.

Eine Sprache L heißt **deterministisch kontextfrei**, wenn es einen deterministischen Kellerautomaten DKA mit $L = L(DKA)$ gibt.

Ein deterministischer Kellerautomat zum Erkennen der Sprache

$$L = \{a^n b^n c^i \mid n \in \mathbf{N}, n \geq 1, i \in \mathbf{N}, i \geq 1\}$$

Die folgende Typ-2-Grammatik G erzeugt L :

$$G = (\{a, b, c\}, \{S, A, B\}, S, \{S \rightarrow aAbcB, A \rightarrow \varepsilon, A \rightarrow aAb, B \rightarrow \varepsilon, B \rightarrow cB\}).$$

Der deterministische Kellerautomaten DKA wird gegeben durch

$DKA = (\{q_0, q_1, q_2\}, \{a, b, c\}, \{a, b, c, \#\}, \delta, q_0, \#, \{q_2\})$ mit der durch folgende Tabelle festgelegten Überföhrungsfunktion δ (mit dem Zeichen „ b “ ist in diesem Beispiel das Terminalsymbol b gemeint und nicht das Blankzeichen):

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

Ein deterministischer Kellerautomat zum Erkennen der Sprache

$$L = \{ a^n b^i c^i \mid n \in \mathbf{N}, n \geq 1, i \in \mathbf{N}, i \geq 1 \}$$

Die folgende Typ-2-Grammatik G erzeugt L :

$$G = (\{ a, b, c \}, \{ S, A, B \}, S, \{ S \rightarrow aAbBc, A \rightarrow \varepsilon, A \rightarrow aA, B \rightarrow \varepsilon, B \rightarrow bBc \}).$$

Der deterministische Kellerautomaten DKA wird gegeben durch

$DKA = (\{ q_0, q_1, q_2, q_3 \}, \{ a, b, c \}, \{ a, b, c, \# \}, \delta, q_0, \#, \{ q_3 \})$ mit der durch folgende Tabelle festgelegten Überföhrungsfunktion δ (mit dem Zeichen „ b “ ist in diesem Beispiel das Terminalsymbol b gemeint und nicht das Blankzeichen):

momentaner Zustand	Symbol unter dem Kopf auf		neuer Zustand	neues Wort im Keller
	Band 1	Keller		
q_0	a	$\#$	q_1	$\#$
q_1	a	$\#$	q_1	$\#$
	b	$\#$	q_2	$\#b$
q_2	b	b	q_2	bb
	c	b	q_3	ε
q_3	c	b	q_3	ε

Ein deterministischer Kellerautomat ist ein Spezialfall eines nichtdeterministischen Kellerautomaten. Daher ist jede deterministisch kontextfreie Sprache (nichtdeterministisch) kontextfrei. Umgekehrt gibt es Sprachen, etwa $L = \{ w \mid w = a_1 \dots a_m a_m \dots a_1, a_i \in \{a, b\} \}$, 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 4.4-8:

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. Dabei wird das zu analysierende Wort w von links nach rechts gelesen, wobei $k \geq 1$ Zeichen im Vorgriff gelesen werden und eine **Rechtsableitung** erzeugt wird: in jedem Ableitungsschritt hat diese die Form $S \Rightarrow^* \alpha Ax \Rightarrow^* w$ mit $\alpha \in (N \cup \Sigma)^*$, $A \in N$ und $x \in \Sigma^*$, und im nächsten Ableitungsschritt wird das Nichtterminalsymbol A durch eine eindeutig bestimmte Ableitungsregel ersetzt. Eine kontextfreie Sprache, die dieses Vorgehen erlaubt, heißt **LR(k)-Sprache**. Es lässt sich zeigen, dass jede LR(k)-Sprache deterministisch kontextfrei und jede deterministisch kontextfreie Sprache eine LR(1)-Sprache ist. Das Verfahren weist eine Zeitkomplexität der Ordnung $O(|w|)$ auf. Die meisten Programmiersprachen sind weitgehend deterministisch kontextfrei, so dass in der Praxis die Syntaxanalyse bei Programmiersprachen sehr effizient abläuft. Details können in der Grundlagenliteratur zum Compilerbau nachgelesen werden¹⁸.

4.5 Typ-3-Sprachen

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

$R \subseteq \{A \rightarrow w \mid A \in N \text{ und } w \in (N \cup \Sigma)^*\}$ folgende zusätzliche Bedingung erfüllen:

Alle Regeln haben die Form $A \rightarrow aB$ oder $A \rightarrow \varepsilon$ mit $A \in N$, $B \in N$, $a \in \Sigma$.

Die von einer derartigen Grammatik erzeugte Sprache heißt **Typ-3-Sprache** oder **rechtslineare Sprache** oder **reguläre Sprache**. Die zugehörige Grammatik heißt **Typ-3-Grammatik** oder **rechtslineare Grammatik**.

Beispielsweise ist die Sprache $L_1 = \{a^i b^j \mid i \in \mathbf{N}, j \in \mathbf{N}\}$ aus Kapitel 4.1 Typ-3-Sprache (regulär, rechtslinear), da sie durch die Grammatik

¹⁸ Siehe: Aho, A.V.; Hopcroft, J.E.; Ullman, J.D.: **The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing**, Addison-Wesley, 1972.

$G' = (\{a, b\}, \{S, B\}, S, \{S \rightarrow \varepsilon, S \rightarrow aS, S \rightarrow bB, B \rightarrow bB, B \rightarrow \varepsilon\})$ erzeugt wird.

Der Name „rechtslineare Sprache“ erklärt sich durch die Form der Ableitungsregeln $A \rightarrow aB$ der zugehörigen Grammatik: Betrachtet man die Nichtterminalsymbole als „Variablen“ (im Sinne eines Gleichungssystems) und die Terminalsymbole als „Konstanten“, so liegt in der Form $A \rightarrow aB$ 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:

Ein **regulärer Ausdruck über Σ** und **die durch ihn repräsentierte Sprache** wird rekursiv definiert durch:

- (i) \emptyset ist ein regulärer Ausdruck, der die reguläre Sprache \emptyset 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.

Beweis:

Zunächst wird gezeigt, dass sich jede durch einen regulären Ausdruck repräsentierte Menge durch eine Typ-3-Grammatik erzeugen lässt. Der Beweis folgt einem für rekursive Definitionen typischen Muster, nämlich Induktion über den Aufbau eines regulären Ausdrucks:

L werde durch einen rekursiven Ausdruck repräsentiert, $L \subseteq \Sigma^*$.

Ist $L = \emptyset$, so gilt für die Typ-3-Grammatik $G_\emptyset = (\Sigma, \{S\}, S, \emptyset)$: $L(G_\emptyset) = L$.

Ist $L = \{ \varepsilon \}$, so gilt für die Typ-3-Grammatik $G_\varepsilon = (\Sigma, \{ S \}, S, \{ S \rightarrow \varepsilon \})$: $L(G_\varepsilon) = L$.

Ist $L = \{ a \}$ mit $a \in \Sigma$, so gilt für die Typ-3-Grammatik $G_a = (\Sigma, \{ S \}, S, \{ S \rightarrow a \})$: $L(G_a) = L$.

L werde durch den regulären Ausdruck $(p+q)$ repräsentiert; die durch p repräsentierte Menge sei P , die durch q repräsentierte Menge sei Q ; es ist $L = P \cup Q$. Zu P gibt es eine Typ-3-Grammatik $G_p = (\Sigma, N_p, S_p, R_p)$ mit $L(G_p) = P$, und zu Q gibt es eine Typ-3-Grammatik $G_q = (\Sigma, N_q, S_q, R_q)$ mit $L(G_q) = Q$. Man kann $N_p \cap N_q = \emptyset$ annehmen, ansonsten werden die Nichtterminalsymbole in N_q umbenannt. Es sei S ein neues Nichtterminalsymbol, $S \notin N_p \cup N_q$. In $R_p \cup R_q$ wird jedes Auftreten von S_p und jedes Auftreten von S_q durch S ersetzt. Die so aus $R_p \cup R_q$ entstandene Ableitungsregelmenge sei R . dann gilt für die Typ-3-Grammatik $G_{(p+q)} = (\Sigma, N_p \cup N_q \cup \{ S \}, S, R)$ offensichtlich $L(G_{(p+q)}) = L$.

L werde durch den regulären Ausdruck (pq) repräsentiert; die durch p repräsentierte Menge sei P , die durch q repräsentierte Menge sei Q ; es ist $L = P \cdot Q$. Zu P gibt es eine Typ-3-Grammatik $G_p = (\Sigma, N_p, S_p, R_p)$ mit $L(G_p) = P$; entsprechend gibt es zu Q eine Typ-3-Grammatik $G_q = (\Sigma, N_q, S_q, R_q)$ mit $L(G_q) = Q$. Man kann $N_p \cap N_q = \emptyset$ annehmen. Für die Grammatik $G_{(pq)} = (\Sigma, N_p \cup N_q, S_p, R)$ sei die Ableitungsregelmenge R wie folgt definiert: Jede Regel der Form $A \rightarrow aB$ in R_p ist in R ; enthält R_p eine Regel der Form $B \rightarrow \varepsilon$, so ist $B \rightarrow \alpha$ in R für jede Ableitungsregel $S_q \rightarrow \alpha$ in R_q . Zusätzlich sind alle Ableitungsregeln aus R_q in R .

Es sei $w \in L$, d.h. $w \in P \cdot Q$ bzw. $w = w_1 w_2$ mit $w_1 \in P$ und $w_2 \in Q$. Folglich gilt $S_p \xRightarrow{*}_{G_p} w_1$ und $S_q \xRightarrow{*}_{G_q} w_2$. Für $|w_1| \geq 1$, etwa $w_1 = a_1 \dots a_n$, hat die erste Ableitung die Form $S_p \xRightarrow{*}_{G_p} a_1 \dots a_{n-1} A \xRightarrow{*}_{G_p} a_1 \dots a_{n-1} a_n B \xRightarrow{*}_{G_p} a_1 \dots a_{n-1} a_n$; im letzten Ableitungsschritt wurde die Ableitungsregel $B \rightarrow \varepsilon$ verwendet. In der Ableitung $S_q \xRightarrow{*}_{G_q} w_2$ sei der erste Ableitungsschritt aus der Anwendung einer Regel $S_q \rightarrow \alpha$ entstanden: $S_q \xRightarrow{*}_{G_q} \alpha \xRightarrow{*}_{G_q} w_2$. Nach Konstruktion enthält R die Ableitungsregel $B \rightarrow \alpha$, und da S_p das Startsymbol von $G_{(pq)}$ ist, kann man in $G_{(pq)}$ die Ableitung $S_p \xRightarrow{*} a_1 \dots a_{n-1} A \Rightarrow a_1 \dots a_{n-1} a_n B \Rightarrow a_1 \dots a_{n-1} a_n \alpha \Rightarrow^* a_1 \dots a_{n-1} a_n w_2$ durchführen. Auf ähnliche Weise kann man für $|w_1| = 0$ argumentieren. Diese Überlegungen zeigen $P \cdot Q \subseteq L(G_{(pq)})$.

Es lasse sich umgekehrt ein Wort $w = a_1 \dots a_m$ in $G_{(pq)}$ ableiten. Da $N_p \cap N_q = \emptyset$ gilt und S_p das Startsymbol der Typ-3-Grammatik $G_{(pq)}$ ist, hat diese Ableitung die Form

$S_p \Rightarrow^* a_1 \dots a_i A \Rightarrow a_1 \dots a_i \alpha \Rightarrow^* a_1 \dots a_i a_{i+1} \dots a_m$, wobei im ersten Teil dieser Ableitungen bis einschließlich zum Ableitungsschritt mit der Ableitungsregel $A \rightarrow \alpha$ nur Nichtterminalsymbole aus N_p ersetzt werden. In der Zeichenkette α trete zum ersten Mal ein Nichtterminalsymbol aus N_q auf. Wegen $N_p \cap N_q = \emptyset$ können in den folgenden Ableitungsschritten nur Nichtterminalsymbole aus N_q ersetzt werden. Da $A \rightarrow \alpha$ eine Ableitungsregel in R mit $A \in N_p$ ist, und α kein Nichtterminalsymbol aus N_p enthält, sind nach Konstruktion von R die Ableitungsregeln $A \rightarrow \varepsilon$ und $S_q \rightarrow \alpha$ in R_q . Insgesamt folgt $S_p \xRightarrow{G_p}^* a_1 \dots a_i A \xRightarrow{G_p} a_1 \dots a_i$ und $S_q \xRightarrow{G_q}^* \alpha \xRightarrow{G_q}^* a_{i+1} \dots a_m$, d.h. $a_1 \dots a_i a_{i+1} \dots a_m \in P \cdot Q$ bzw. $L(G_{(pq)}) \subseteq P \cdot Q$.

L werde durch den regulären Ausdruck $(p)^*$ repräsentiert; die durch p repräsentierte Menge sei P ; es ist $L = P^*$. Zu P gibt es eine Typ-3-Grammatik $G_p = (\Sigma, N_p, S_p, R_p)$ mit $L(G_p) = P$. Es sei S ein Nichtterminalsymbol mit $S \notin N_p$. Die Menge R der Ableitungsregeln in der Grammatik $G_{(p)^*} = (\Sigma, N_p \cup \{S\}, S, R)$ werde folgendermaßen definiert:

Alle Ableitungsregeln aus R_p sind in R ; für jede Ableitungsregel der Form $B \rightarrow \varepsilon$ in R_p enthält R die Ableitungsregel $B \rightarrow \alpha$, falls $S_p \rightarrow \alpha$ in R_p ist; zusätzlich enthält R die Ableitungsregeln $S \rightarrow \varepsilon$ und $S \rightarrow \alpha$ für jede Ableitungsregel $S_p \rightarrow \alpha$ in R_p .

Es sei $w \in P^*$. Für $w = \varepsilon$ ist $w \in L(G_{(p)^*})$. Es gelte $|w| \geq 1$, d.h. $w \in w_1 \dots w_n$ mit $w_i \in P$ für $i = 1, \dots, n$. Man kann $|w_i| \geq 1$ für $i = 1, \dots, n$ annehmen, da man andernfalls w_i aus w streichen kann. Dann gilt $S_p \xRightarrow{G_p}^+ w_i$ für $i = 1, \dots, n$. Der erste Ableitungsschritt erfolgt jeweils durch die Anwendung einer Regel der Form $S_p \rightarrow \alpha_i$, der letzte Ableitungsschritt jeweils durch Anwendung einer Ableitungsregel der Form $A_i \rightarrow \varepsilon$, d.h. es ist $S_p \xRightarrow{G_p}^* \alpha_i \xRightarrow{G_p}^* w_i A_i \xRightarrow{G_p} w_i$ für $i = 1, \dots, n$. Nach Konstruktion enthält R die Ableitungsregeln $A_i \rightarrow \alpha_{i+1}$ für $i = 1, \dots, n-1$, und in $G_{(p)^*}$ ist folgende Ableitung möglich:

$S \Rightarrow \alpha_1 \Rightarrow^* w_1 A_1 \Rightarrow w_1 \alpha_2 \Rightarrow^* w_1 w_2 A_2 \Rightarrow w_1 w_2 \alpha_3 \Rightarrow^* w_1 w_2 \dots w_n A_n \Rightarrow w_1 w_2 \dots w_n$. Das zeigt $P^* \subseteq L(G_{(p)^*})$.

Es sei $w \in L(G_{(p)^*})$. Für $w = \varepsilon$ ist $w \in P^*$. Es sei $w \in a_1 \dots a_m$ mit $a_i \in \Sigma$ für $i = 1, \dots, m$ und $m \geq 1$. In den Satzformen innerhalb der Ableitung $S \Rightarrow \alpha \Rightarrow^* a_1 \dots a_m$ kommt mit Ausnahme der terminalen Zeichenkette $a_1 \dots a_m$ genau ein Nichtterminalsymbol vor. Nach Konstruktion von R enthält R_p die Ableitungsregel $S_p \rightarrow \alpha$. Werden in der Ableitung $\alpha \Rightarrow^* a_1 \dots a_m$ nur

Ableitungsregeln aus R_p angewandt, so gilt $S_p \xRightarrow{G_p} \alpha \xRightarrow{G_p}^* a_1 \dots a_m$, also $w \in L(G_p)$ bzw. $w \in P$ und $w \in P^*$. Andernfalls seien die Ableitungsschritte i_1, \dots, i_k diejenigen, in denen eine Ableitungsregel der Form $A_{i_j} \rightarrow \alpha_{i_j}$ angewandt wird, die in R aufgenommen wurde, weil $A_{i_j} \rightarrow \varepsilon$ und $S_p \rightarrow \alpha_{i_j}$ in R_p sind. Zwischen diesen Ableitungsschritten werden nur Ableitungsregeln aus R_p angewandt. In $G_{(p)^*}$ gilt:

$$\begin{aligned} S &\Rightarrow \alpha \xRightarrow{G_p}^* a_1 \dots a_{i_1} A_{i_1} \Rightarrow a_1 \dots a_{i_1} \alpha_{i_1} \xRightarrow{G_p}^* a_1 \dots a_{i_1} a_{i_1+1} \dots a_{i_2} A_{i_2} \Rightarrow a_1 \dots a_{i_1} a_{i_1+1} \dots a_{i_2} \alpha_{i_2} \\ &\xRightarrow{G_p}^* a_1 \dots a_{i_1} a_{i_1+1} \dots a_{i_2} \dots a_{i_k} A_{i_k} \Rightarrow a_1 \dots a_{i_1} a_{i_1+1} \dots a_{i_2} \dots a_{i_k} \alpha_{i_k} \xRightarrow{G_p}^* a_1 \dots a_{i_1} a_{i_1+1} \dots a_{i_2} \dots a_{i_k} a_{i_k+1} \dots a_m. \end{aligned}$$

Dann gilt: $S_p \xRightarrow{G_p} \alpha \xRightarrow{G_p}^* a_1 \dots a_{i_1} A_{i_1} \Rightarrow a_1 \dots a_{i_1}$, $S_p \xRightarrow{G_p} \alpha_{i_1} \xRightarrow{G_p}^* a_{i_1+1} \dots a_{i_2} A_{i_2} \Rightarrow a_{i_1+1} \dots a_{i_2}$, \dots , $S_p \xRightarrow{G_p} \alpha_{i_k} \xRightarrow{G_p}^* a_{i_k+1} \dots a_m$. Also ist $a_1 \dots a_{i_1} \in P$, $a_{i_1+1} \dots a_{i_2} \in P$, \dots , $a_{i_k+1} \dots a_m \in P$ und $w \in P^*$.

Der Beweis der „umgekehrten“ Aussage, dass jede Typ-3-Sprache durch einen regulären Ausdruck repräsentiert wird, folgt direkt aus den Sätzen 4.5-3 und 4.5-4.

///

Auch für die Typ-3-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. $\delta: Q \times \Sigma \rightarrow \mathbf{P}(Q)$ ist eine partielle Funktion, die **Überföhrungsfunktion**; 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 Überföhrungsfunktion δ die Form $\delta: Q \times \Sigma \rightarrow Q$ aufweist.

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).

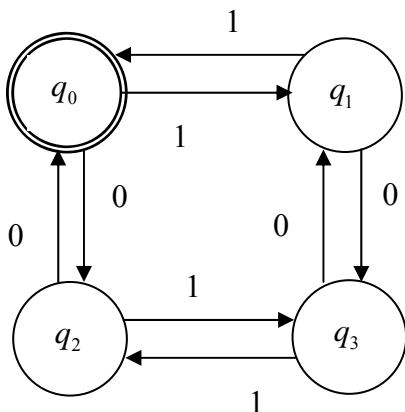
Ein deterministischer bzw. nichtdeterministischer endlicher Automat lässt sich auch in Form eines endlichen gerichteten Graphen G mit markierten Kanten als **Transitionsdiagramm** darstellen. Die Knotenmenge von G ist Q . Ist $z' = \delta(z, a)$ bzw. im nichtdeterministischen Fall $z' \in \delta(z, a)$, 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

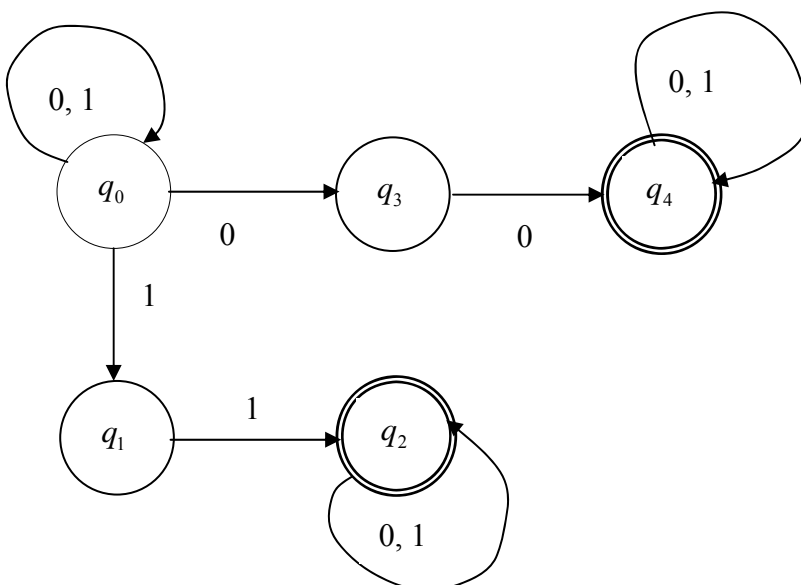
$$L(DEA) = \{ w \mid w \in \{0,1\}^* \text{ und } w \text{ enthält eine gerade Anzahl von Zeichen 0 und Zeichen 1} \}.$$

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
	1	q_3
q_1	0	q_0
	1	q_1
q_2	1	q_2
q_3	0	q_2
	1	q_2
q_4	0	q_4
q_4	0	q_4
	1	q_4

Das zugehörige Transitionsdiagramm lautet



Es ist

$$L(NEA) = \left\{ w \mid \begin{array}{l} w \in \{0,1\}^+ \\ \text{und } w \text{ enthält mindestens zwei aufeinanderfolgende Zeichen 0 oder Zeichen 1} \end{array} \right\}.$$

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:

$$Q' = \mathbf{P}(Q), \quad q'_0 = \{q_0\}, \quad \delta'(Z, a) = \bigcup_{q \in Z} \delta(q, a) \text{ für } Z \subseteq Q \text{ und } a \in \Sigma,$$

$$F' = \{Z \mid Z \subseteq Q \text{ und } Z \cap F \neq \emptyset\}.$$

Es sei $w \in \Sigma^*$. Durch vollständige Induktion über die Länge m der Ableitung lässt sich zeigen,

dass $(Z, w) \xRightarrow{DEA}^m (Z', \varepsilon)$ genau dann gilt, wenn $Z' = \bigcup_{q \in Z} \left\{ p \mid (q, w) \xRightarrow{NEA}^m (p, \varepsilon) \right\}$ ist:

Hat die Ableitung die Länge 0, so stimmt die Aussage trivialerweise. Die Ableitung habe die Länge $m+1$ und w die Form $w = aw'$: $(Z, aw') \xRightarrow{DEA}^m (Z'', w') \xRightarrow{DEA}^m (Z', \varepsilon)$; nach Definition von δ'

ist $Z'' = \bigcup_{z \in Z} \delta(z, a)$, und nach Induktionsvoraussetzung ist $Z' = \bigcup_{r \in Z''} \left\{ p \mid (r, w') \xRightarrow{NEA}^m (p, \varepsilon) \right\}$. Es sei

$z \in Z'$. Genau dann gibt es $r \in Z''$ mit $(r, w') \xRightarrow{NEA}^m (z, \varepsilon)$. Es ist $r \in \delta(q, a)$ für ein $q \in Z$. Daher

ist in NEA folgende Konfigurationsabfolge möglich: $(q, aw') \xRightarrow{NEA}^m (r, w') \xRightarrow{NEA}^m (z, \varepsilon)$, d.h.

$Z' \subseteq \bigcup_{q \in Z} \left\{ p \mid (q, w) \xrightarrow[NEA]{m+1} (p, \varepsilon) \right\}$. Da alle Argumentationsschritte umkehrbar sind, folgt die Behauptung.

Setzt man $Z = \{q_0\}$, so folgt:

$(\{q_0\}, w) \xrightarrow[DEA]^* (Z', \varepsilon)$ gilt genau dann, wenn $Z' = \left\{ p \mid (q_0, w) \xrightarrow[NEA]^* (p, \varepsilon) \right\}$ ist.

Es sei $w \in L(NEA)$. Dann gibt es $q \in F$ mit $(q_0, w) \xrightarrow[NEA]^* (q, \varepsilon)$. Für $Z' = \left\{ p \mid (q_0, w) \xrightarrow[NEA]^* (p, \varepsilon) \right\}$

ist $q \in Z'$, also $Z' \cap F \neq \emptyset$, d.h. $Z' \in F'$, und $(\{q_0\}, w) \xrightarrow[DEA]^* (Z', \varepsilon)$. Daher $w \in L(DEA)$.

Ist umgekehrt $w \in L(DEA)$, so gibt es $Z' \in F'$ mit $(\{q_0\}, w) \xrightarrow[DEA]^* (Z', \varepsilon)$, also

$Z' = \left\{ p \mid (q_0, w) \xrightarrow[NEA]^* (p, \varepsilon) \right\}$. Da $Z' \cap F \neq \emptyset$ ist, gibt es $q \in F$ mit $(q_0, w) \xrightarrow[NEA]^* (q, \varepsilon)$, d.h.

$w \in L(NEA)$.

Insgesamt ergibt sich $L(NEA) = L(DEA)$.

///

Der folgende Satz besagt, dass das Konzept der endlichen erkennenden Automaten das adäquate Modell für die Typ-3-Sprachen 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 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 Typ-3-Grammatik $G = (\Sigma, N, S, R)$ angegeben, für die $L(G) = L(DEA)$ gilt:

$N = Q$, $S = q_0$, die Menge R der Ableitungsregeln wird definiert durch:

R enthält genau dann eine Ableitungsregel $q \rightarrow aq'$ 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 Ableitungsregel $q \rightarrow \varepsilon$.

Es sei $w \in L(DEA)$. Für $w = \varepsilon$ ist $q_0 \in F$, und in G ist die Ableitung $q_0 \Rightarrow \varepsilon$ möglich, d.h. $w \in L(G)$. Ist $w = a_1 \dots a_n$ mit $n \geq 1$, so gibt es in DEA Zustände $q_1 \dots q_n$, so dass folgende Überführungen stattfinden: $(q_0, a_1 \dots a_n) \Rightarrow (q_1, a_2 \dots a_n) \Rightarrow \dots \Rightarrow (q_{n-1}, a_n) \Rightarrow (q_n, \varepsilon)$; hierbei ist $q_n \in F$. Nach Definition enthält G die Ableitungsregeln $q_{i-1} \rightarrow a_i q_i$ für $i = 1, \dots, n$ und $q_n \rightarrow \varepsilon$. In G ist daher die Ableitung $q_0 \Rightarrow a_1 q_1 \Rightarrow a_1 a_2 q_2 \Rightarrow \dots \Rightarrow a_1 \dots a_n q_n \Rightarrow a_1 \dots a_n$ möglich, d.h. $w \in L(G)$.

Der Beweis der Inklusion $L(G) \subseteq L(DEA)$ verläuft entsprechend.

Es sei umgekehrt $G = (\Sigma, N, S, R)$ eine Typ-3- Grammatik. Dann erkennt der folgende Automat $NEA = (Q, \Sigma, \delta, q_0, F)$ die Sprache $L(G)$:

$Q = N$, $q_0 = S$, $F = \{B \mid B \rightarrow \varepsilon \text{ ist eine Ableitungsregel}\}$, die Überföhrungsfunktion $\delta: Q \times \Sigma \rightarrow \mathbf{P}(Q)$ wird definiert durch:

$\delta(B, a)$ enthält genau dann C , wenn R die Ableitungsregel $B \rightarrow aC$ enthält.

Es sei $w \in L(G)$. Für $w = \varepsilon$ enthält G die Ableitungsregel $S \rightarrow \varepsilon$, es ist $S \in F$, und NEA macht bei Eingabe von w keine Überföhrung und stoppt im Endzustand S , d.h. $w \in L(NEA)$.

Für $w = a_1 \dots a_n$ mit $n \geq 1$ gibt es in G Nichtterminalsymbole $A_1 \dots A_n$, so dass folgende Ableitungsschritte in G möglich sind: $S \Rightarrow a_1 A_1 \Rightarrow a_1 a_2 A_2 \Rightarrow \dots \Rightarrow a_1 \dots a_n A_n \Rightarrow a_1 \dots a_n$. In NEA sind dann folgende Überföhrungen möglich:

$(S, a_1 \dots a_n) \Rightarrow (A_1, a_2 \dots a_n) \Rightarrow \dots \Rightarrow (A_{n-1}, a_n) \Rightarrow (A_n, \varepsilon)$. Da G die Ableitungsregel $A_n \rightarrow \varepsilon$ enthält, ist $A_n \in F$ und $w \in L(NEA)$.

Der Beweis der Inklusion $L(NEA) \subseteq L(G)$ verläuft entsprechend.

///

Der folgende Satz stellt eine Verbindung zwischen den von endlichen Automaten erkannten und den von regulären Ausdröcken repräsentierten Sprachen her.

Satz 4.5-4:

Die von einem (nichtdeterministischen bzw. deterministischen) endlichen Automaten akzeptierte Sprache über einem endlichen Alphabet Σ lässt sich durch einen regulären Ausdruck über Σ repräsentieren.

Beweis:

Es sei $L = L(DEA)$ mit einem deterministischen endlichen Automaten $DEA = (Q, \Sigma, \delta, q_0, F)$. Es sei $\Sigma = \{a_1, \dots, a_n\}$. Zur Vereinfachung der Darstellung sei $Q = \{1, \dots, m\}$ und $q_0 = 1$. O.B.d.A. kann angenommen werden, dass F einelementig ist und dass $F = \{m\}$ ist. Es sei

$$A_{i,j}^{(k)} = \left\{ u \mid \begin{array}{l} u \in \Sigma^*, (i, uw) \Rightarrow^* (j, w), \\ \text{und es werden nur Zwischenzustände aus } \{1, \dots, k\} \text{ angenommen} \end{array} \right\} \quad \text{für } i = 1, \dots, m, \\ j = 1, \dots, m, k = 0, \dots, m.$$

Speziell für $k = 0$: $A_{i,j}^{(0)} \subseteq \Sigma$.

Es wird durch vollständige Induktion über k gezeigt:

$A_{i,j}^{(k)}$ für $i = 1, \dots, m, j = 1, \dots, m, k = 0, \dots, m$ lässt sich durch einen regulären Ausdruck repräsentieren:

Für $k = 0$ ist $A_{i,j}^{(0)} = \{a_{i_1}, \dots, a_{i_j}\}$ und durch den regulären Ausdruck $(\dots(a_{i_1} + a_{i_2}) + \dots + a_{i_j})$ repräsentiert.

Für $k > 0$ werden die Mengen $A_{i,j}^{(0)}, \dots, A_{i,j}^{(k-1)}$ durch die regulären Ausdrücke $p_{i,j}^{(0)}, \dots, p_{i,j}^{(k-1)}$ repräsentiert. Es wird gezeigt, dass $A_{i,j}^{(k)}$ durch den regulären Ausdruck $(p_{i,j}^{(k-1)} + (p_{i,k}^{(k-1)} \cdot ((p_{k,k}^{(k-1)})^* \cdot p_{k,j}^{(k-1)})))$ repräsentiert wird. Dazu ist

$$A_{i,j}^{(k)} = A_{i,j}^{(k-1)} \cup A_{i,k}^{(k-1)} \cdot (A_{k,k}^{(k-1)})^* \cdot A_{k,j}^{(k-1)} \text{ zu zeigen:}$$

Es sei $u \in A_{i,j}^{(k)}$. Werden beim Übergang vom Zustand i zum Zustand j nur Zwischenzustände aus $\{1, \dots, k-1\}$ angenommen, so ist $u \in A_{i,j}^{(k-1)}$. Falls der Zwischenzustand k angenommen wird, dann ist $u = u_1 u_2 \dots u_t u_{t+1}$ mit $u_1 \in A_{i,k}^{(k-1)}, u_2 \in A_{k,k}^{(k-1)}, \dots, u_t \in A_{k,k}^{(k-1)}, u_{t+1} \in A_{k,j}^{(k-1)}$. Daraus folgt $u \in A_{i,j}^{(k-1)} \cup A_{i,k}^{(k-1)} \cdot (A_{k,k}^{(k-1)})^* \cdot A_{k,j}^{(k-1)}$.

Offensichtlich gilt die umgekehrte Inklusion $A_{i,j}^{(k-1)} \cup A_{i,k}^{(k-1)} \cdot (A_{k,k}^{(k-1)})^* \cdot A_{k,j}^{(k-1)} \subseteq A_{i,j}^{(k)}$.

Damit lässt sich $A_{1,m}^{(m)} = L(DEA)$ durch einen regulären Ausdruck repräsentieren.

///

Zusammenfassend gilt

Satz 4.5-5:

Die Klasse der von (nichtdeterministischen bzw. deterministischen) endlichen Automaten akzeptierten Sprachen über einem endlichen Alphabet Σ ist mit der Klasse der Typ-3-Sprachen über Σ und der durch reguläre Ausdrücke über Σ repräsentierten Sprachen identisch.

Auch für Typ-3-Sprachen gibt es einen dem $uvwxy$ -Theorem (Satz 4.4-3) entsprechenden Satz, der wieder ein Beweismittel liefert, mit dessen Hilfe man zeigen kann, dass eine Sprache nicht Typ-3-Sprache ist:

Satz 4.5-6:

Zu jeder Typ-3-Sprache (regulären, 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 Typ-3-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 L eine Typ-3-Sprache und DEA ein deterministischer endlicher Automat mit $L(DEA) = L$. DEA enthalte n_0 viele Zustände. Bei der Akzeptanz eines Wortes $z \in L$ mit $|z| \geq n_0$ durchläuft DEA einschließlich des Anfangszustands $|z|+1$ viele Zustände. Hierbei geht wesentlich ein, dass DEA bei jeder Überführung ein Eingabesymbol liest. In der Folge der Zustandsüberführungen vom Anfangszustand q_0 bis zu einem akzeptierenden Zustand $q_{accept} \in F$ muss sich also ein Zustand q_i wiederholen. Es sei u das Anfangsteilwort von z , das in den Überführungen gelesen wurde, die DEA von q_0 aus bis zum ersten Erreichen von q_i gemacht hat. Das Teilwort v von z besteht aus den Symbolen, die DEA dann von q_i aus bis zum nächsten Erreichen von q_i gelesen hat. Schließlich ist w das Teilwort von z , das DEA liest, bis der akzeptierende Zustand q_{accept} erreicht ist. Mit dieser Zerlegung von z gelten die angegebenen drei Eigenschaften.

///

Mit Hilfe dieses Satzes lässt sich zeigen, dass die Menge der Palindrome

$L = \{w \mid w \in \{0,1\}^* \text{ und } w = a_1 \dots a_{n-1} a_n = a_n a_{n-1} \dots a_1\} \cup \{\varepsilon\}$ nicht regulär ist; L ist jedoch kontextfrei¹⁹. Es sei $n_0 > 0$ der Wert Satz 4.5-4 und n eine gerade Zahl mit $n > n_0$. Das Wort $z = \underbrace{1010 \dots 100101 \dots 01}_{n \text{ Zeichen}}$ 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

¹⁹ Eine kontextfreie Grammatik, die L erzeugt, ist

$G = (\{0,1\}, \{S\}, S, \{S \rightarrow \varepsilon, S \rightarrow 0S0, S \rightarrow 1S1\})$.

(00) aufeinanderfolgen, in der Mitte von z liegt. Das Teilwort uv ist wegen $|uv| \leq n_0 < n$ ein echtes Anfangsstück von $\underbrace{1010\dots10}_{n \text{ Zeichen}}$.

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 $\underbrace{1010\dots10}_{n \text{ Zeichen}}$ ist, stellt v nicht die letzte 0 in $\underbrace{1010\dots10}_{n \text{ Zeichen}}$ 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 $v = 0\dots1$: 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 $v = 1\dots1$: 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 Typ-3-Grammatik erfüllen die Bedingungen, die an die Regeln einer kontextfreien Grammatik gestellt werden. Jede von einer Typ-3-Grammatik erzeugte Sprache ist daher auch eine kontextfreie Sprache. Die Sätze 4.5-2 und 4.5-3 implizieren, dass es zu jeder Typ-3-Sprache über einem endlichen Alphabet Σ einen deterministischen Kellerautomaten gibt, der die Sprache akzeptiert. Andererseits lässt sich mit Satz 4.5-6 leicht zeigen, dass die deterministisch kontextfreie Sprache $L = \{a^n b^n \mid n \in \mathbf{N}\}$ nicht regulär ist. Es gilt daher:

Satz 4.5-7:

Die Klasse der Typ-3-Sprachen (reguläre, rechtslineare 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

$VERIFIZIERE_G_TYP_3(w)$

$$= \begin{cases} \text{TRUE} & \text{falls } w \text{ die Kodierung einer Typ-3-Grammatik darstellt} \\ \text{FALSE} & \text{falls } w \text{ nicht die Kodierung einer Typ-3-Grammatik darstellt} \end{cases}$$

definiert. Dieses Prädikat ist berechenbar. Dazu ist unter anderem zu prüfen, ob die Ableitungsregeln in KG_w den Bedingungen einer Typ-3-Grammatik genügen.

Das **Wortproblem für Typ-3-Grammatiken** fragt danach, ob die Menge

$$L_{\text{Wort_Typ-3}} = \left\{ u\#w \mid u \in \Sigma^*, w \in \{0,1\}^*, VERIFIZIERE_G_TYP_3(w) = \text{TRUE} \text{ und } u \in L(KG_w) \right\}$$

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 Wortes $u \in \Sigma^*$ 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 $\text{accept}(DEA, u)$

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

$$L_{\text{leer_Typ-3}} = \left\{ w \mid w \in \{0, 1\}^*, \text{VERIFIZIERE_G_TYP_3}(w) = \text{TRUE} \text{ und } L(KG_w) = \emptyset \right\}$$

entscheidbar ist. Diese Frage ist natürlich positiv zu beantworten, da das Leerheitsproblem für Typ-2-Grammatiken entscheidbar ist. Im Fall einer Typ-3-Sprache L kann man mit Satz 4.5-6 argumentieren: Es sei n_0 die in Satz 4.5-6 angegebene natürliche Zahl. Ist $L \neq \emptyset$ und $z \in L$ ein Wort mit minimaler Länge, dann gilt $|z| < n_0$. Denn wäre $|z| \geq n_0$, dann könnte man z zerlegen in $z = uvw$ mit $|v| > 0$, und L enthält auch das Wort uv^0w , dessen Länge kürzer als die Länge von z ist. Das bedeutet:

Es sei n_0 die in Satz 4.5-6 angegebene natürliche Zahl. Dann gilt:

$$L \neq \emptyset \text{ genau dann, wenn es ein Wort } u \in L \text{ gibt mit } |u| < n_0$$

Eine Überprüfung der Frage „ $L = \emptyset$?“ 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 4.5-8:

Ist L eine kontextfreie Sprache über dem Alphabet Σ und R eine Typ-3-Sprache (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 Überföhrungsfunktion 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 $uvwxyx$ -Theorems für kontextfreie Sprachen (Satz 4.4-3) 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 4.5-8 wegen $L_1 = L_2 \cap R$ auch L_1 kontextfrei. Es seien c_1, \dots, c_n neue paarweise verschiedene Buchstaben. Dann ist $L_3 = \{x_1 w x_2 w x_3 \mid w \in \{a, b\}^+, x_i \in \{c_1, \dots, c_n\}^* \text{ für } i=1, 2, 3\}$ nicht kontextfrei.

Denn wäre L_3 kontextfrei, dann gäbe es eine kontextfreie Grammatik G mit $L_3 = L(G)$. Ersetzt man in jeder Ableitungsregel $A \rightarrow \alpha$ in der Zeichenkette α vorkommende Symbole c_1, \dots, 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

$$L_4 = \left\{ \text{PROGRAM A VAR } w : \text{INTEGER}; \text{BEGIN } w := 1; \text{END.} \mid w \in \{a, b\}^+ \right\}.$$

Zusätzlich wird die reguläre Menge R definiert durch

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

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.

Beschreibungsmittel	
Typ 0	Typ-0-Grammatik Turingmaschine
Typ 1	kontextsensitive Grammatik linear beschränkter Automat
Typ 2	kontextfreie Grammatik nichtdeterministischer 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 (1)	ja (2)
Typ 1	ja (Satz 2.1-2 mit LBA)	ja (Satz 2.1-2 mit LBA)	ja (siehe Text in Kapitel 4.3)	ja (3)	ja (4)
Typ 2	nein (5)	ja (6)	nein (7)	ja (8)	ja (9)
det. kontextfrei	nein (10)	nein (11)	ja (12)	nein (13)	nein (14)
Typ 3	ja (15)	ja (16)	ja (17)	ja (18)	ja (19)

Zu den Zahlenangaben:

- (1) Die Sprache $L_1 \subseteq \Sigma^*$ werde von einer Turingmaschine TM_1 akzeptiert, die Sprache $L_2 \subseteq \Sigma^*$ werde von einer Turingmaschine TM_2 akzeptiert. Eine Turingmaschine, die $L_1 \cdot L_2$ akzeptiert, arbeitet wie folgt: Bei Eingabe eines Worts $w \in \Sigma^*$ wird eine Position innerhalb des Worts nichtdeterministisch geraten, die w in zwei Teilworte w_1 und w_2 zerlegt: $w = w_1 w_2$. Dann wird das Verhalten von TM_1 auf w_1 simuliert. Falls TM_1 akzeptiert, wird das Verhalten von TM_2 auf w_2 simuliert. Das Wort w wird genau dann akzeptiert, wenn TM_1 und TM_2 das jeweilige Teilwort akzeptieren.
- (2) Die Sprache $L \subseteq \Sigma^*$ werde von einer Turingmaschine TM akzeptiert. Eine Turingmaschine, die L^* akzeptiert, arbeitet wie folgt: Bei Eingabe eines Worts $w \in \Sigma^*$ werden $k-1$ Positionen innerhalb des Worts nichtdeterministisch geraten, die w in k Teilworte w_1, \dots, w_k zerlegen: $w = w_1 \dots w_k$. Für jedes Teilwort w_i mit $i=1, \dots, k$ wird geprüft, ob $w_i \in L(TM)$ gilt. Genau in diesem Fall wird w akzeptiert.

- (3) Anstelle der in (1) beschriebenen Turingmaschinen werden jetzt linear beschränkte Automaten eingesetzt. Alternativ kann man mit kontextsensitiven Grammatiken argumentieren: Es seien L_1 und L_2 kontextsensitive Sprachen, die jeweils von den Typ-1-Grammatiken $G_1 = (\Sigma, N_1, S_1, R_1)$ bzw. $G_2 = (\Sigma, N_2, S_2, R_2)$ erzeugt werden. Man kann $N_1 \cap N_2 = \emptyset$ annehmen (ansonsten werden die Nichtterminalsymbole in N_2 umbenannt). Für das (neue) Nichtterminalsymbol S gelte $S \notin N_1 \cup N_2$. Ist $\varepsilon \notin L(G_1)$ und $\varepsilon \notin L(G_2)$, dann wird $L_1 \cdot L_2$ durch die Typ-1-Grammatik $G = (\Sigma, N_1 \cup N_2 \cup \{S\}, S, R_1 \cup R_2 \cup \{S \rightarrow S_1 S_2\})$ erzeugt. Ist $\varepsilon \in L(G_1) \cap L(G_2)$, dann wird $L_1 \cdot L_2$ durch die Typ-1-Grammatik $G = (\Sigma, N_1 \cup N_2 \cup \{S\}, S, (R_1 \cup R_2 \cup \{S \rightarrow S_1 S_2, S \rightarrow \varepsilon\}) \setminus \{S_1 \rightarrow \varepsilon, S_2 \rightarrow \varepsilon\})$ erzeugt. Ist $\varepsilon \notin L(G_1)$ und $\varepsilon \in L(G_2)$, dann wird $L_1 \cdot L_2$ durch die Typ-1-Grammatik $G = (\Sigma, N_1 \cup N_2 \cup \{S\}, S, (R_1 \cup R_2 \cup \{S \rightarrow S_1 S_2, S \rightarrow S_1\}) \setminus \{S_2 \rightarrow \varepsilon\})$ erzeugt. Ist $\varepsilon \in L(G_1)$ und $\varepsilon \notin L(G_2)$, dann wird $L_1 \cdot L_2$ durch die Typ-1-Grammatik $G = (\Sigma, N_1 \cup N_2 \cup \{S\}, S, (R_1 \cup R_2 \cup \{S \rightarrow S_1 S_2, S \rightarrow S_2\}) \setminus \{S_1 \rightarrow \varepsilon\})$ erzeugt.
- (4) Die kontextsensitive Sprachen L werde durch die Typ-1-Grammatik $G = (\Sigma, N, S, R)$ erzeugt. Es sei $S' \notin N$ ein neues Nichtterminalsymbol. Die Grammatik $G^* = (\Sigma, N \cup \{S'\}, S', (R \cup \{S' \rightarrow \varepsilon, S' \rightarrow S, S \rightarrow SS\}) \setminus \{S \rightarrow \varepsilon\})$ erzeugt L^* .
- (5) Die Sprachen $L_1 = \{a^n b^n c^i \mid n \in \mathbf{N}, n \geq 1, i \in \mathbf{N}, i \geq 1\}$ und $L_2 = \{a^i b^n c^n \mid n \in \mathbf{N}, n \geq 1, i \in \mathbf{N}, i \geq 1\}$ sind kontextfrei, nicht aber $L_1 \cap L_2 = \{a^n b^n c^n \mid n \in \mathbf{N}, n \geq 1\}$ (siehe Kapitel 4.4).
- (6) Es seien L_1 und L_2 kontextfreie Sprachen, die jeweils von den Typ-2-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 $N_1 \cap N_2 = \emptyset$ ist, ansonsten benennt man die Nichtterminalzeichen entsprechend um. Es sei S ein neues Nichtterminalsymbol. Dann wird $L_1 \cup L_2$ durch die Typ-2-Grammatik $G = (\Sigma, N_1 \cup N_2 \cup \{S\}, S, R_1 \cup R_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\})$ erzeugt.
- (7) Angenommen, die Klasse der kontextfreien Sprachen wäre gegen Komplementbildung abgeschlossen. Dann wäre sie auch gegenüber Schnittbildung abgeschlossen, da der Schnitt zweier Mengen durch Vereinigung- und Komplementbildung ausgedrückt werden kann: $L_1 \cap L_2 = \Sigma^* \setminus ((\Sigma^* \setminus L_1) \cup (\Sigma^* \setminus L_2))$.
- (8) Die Argumentation verläuft wie bei (3) mit Typ-2-Grammatiken anstelle von Typ-1-Grammatiken.

- (9) Die Argumentation verläuft wie bei (4) mit einer Typ-2-Grammatik anstelle einer Typ-1-Grammatik.
- (10) Die Sprachen $L_1 = \{a^n b^n c^i \mid n \in \mathbf{N}, n \geq 1, i \in \mathbf{N}, i \geq 1\}$ und $L_2 = \{a^i b^n c^n \mid n \in \mathbf{N}, n \geq 1, i \in \mathbf{N}, i \geq 1\}$ aus (5) sind jeweils deterministisch kontextfrei (siehe Kapitel 4.4).
- (11) Die Konstruktion aus (6) baut in den ersten Ableitungsschritt einer Ableitung $S \Rightarrow^* w$ Nichtdeterminismus ein; denn eine der beiden Produktionen $S \rightarrow S_1$ bzw. $S \rightarrow S_2$ 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 $L_1 \cap L_2 = \Sigma^* \setminus ((\Sigma^* \setminus L_1) \cup (\Sigma^* \setminus L_2))$.
- (12) Hier erfolgt die Argumentation mit Hilfe eines deterministischen Kellerautomaten. Details finden sich in der angegebenen Literatur²⁰.
- (13) Es seien $L_1 \subseteq \Sigma^*$ und $L_2 \subseteq \Sigma^*$ zwei deterministisch kontextfreie Sprachen, deren Vereinigung nicht deterministisch kontextfrei ist, siehe (11). Für das Terminalsymbol $\#$ gelte $\# \notin \Sigma^*$. Dann ist die Menge $L_3 = \{\#\} \cdot L_1 \cup L_2$ deterministisch kontextfrei: Ein Wort $w \in (\Sigma \cup \{\#\})^*$ der Form $w = \#w'$ ist genau dann in L_3 , wenn $w' \in L_1$ gilt; beginnt w mit einem Terminalsymbol, das verschieden von $\#$ ist, dann ist w genau dann in L_3 , wenn $w \in L_2$ gilt; die Gültigkeit der jeweiligen Alternative kann von dem deterministischen Kellerautomaten geprüft werden, der L_1 bzw. L_2 akzeptiert. Offensichtlich ist die Menge $\{\#\}^*$ als reguläre Menge deterministisch kontextfrei. Die Menge Σ^* ist regulär. Wäre nun $\{\#\}^* \cdot L_3$ deterministisch kontextfrei, dann ist nach Satz 4.5-8 auch die Menge $L_4 = (\{\#\}^* \cdot L_3) \cap (\{\#\}^* \cdot \Sigma^*)$ deterministisch kontextfrei; die runden Klammern dienen hier als metasyntaktische Trennzeichen. Nach Satz 1.1-3(v) ist
- $$\begin{aligned} L_4 &= (\{\#\}^* \cdot L_3) \cap (\{\#\}^* \cdot \Sigma^*) \\ &= (\{\#\}^* \cdot (\{\#\} \cdot L_1 \cup L_2)) \cap (\{\#\}^* \cdot \Sigma^*) \\ &= (\{\#\}^+ \cdot L_1 \cup \{\#\}^* \cdot L_2) \cap (\{\#\}^* \cdot \Sigma^*) \\ &= \{\#\} \cdot L_1 \cup \{\#\} \cdot L_2 \\ &= \{\#\} \cdot (L_1 \cup L_2) . \end{aligned}$$

²⁰ Siehe: Hopcroft, J.E.; Ullman, J.D.: **Introduction to Automata Theory, Languages, and Computation**, Addison Wesley, 1979.

Es sei DKA ein deterministischer Kellerautomat mit $L(DKA) = L_4$. Dann gibt es einen deterministischen Kellerautomaten, der $L_1 \cup L_2$ akzeptiert: Bei einer Eingabe $w \in \Sigma^*$ simuliert DKA zunächst das Verhalten bei einer gedachten Eingabe $\#$ und anschließend das Verhalten auf w . Es gilt $w \in L_1 \cup L_2$ genau dann, wenn DKA bei dieser Simulation akzeptiert. Nach Voraussetzung ist aber $L_1 \cup L_2$ nicht deterministisch kontextfrei.

- (14) Es sei $L_5 = \{\#\} \cup L_3$ mit L_3 wie in (13). Da L_3 deterministisch kontextfrei ist, gilt dieses auch für L_5 : Ein Wort $w \in (\Sigma \cup \{\#\})^*$ ist genau dann in L_5 , wenn $w = \#$ oder $w \in L_3$ ist. Ähnlich wie in (13) kann man zeigen, dass L_5^* aber nicht deterministisch kontextfrei ist.
- (15) Wird L_1 vom deterministischen endlichen Automaten $E_1 = (Q_1, \Sigma, \delta_1, q_{1,0}, F_1)$ und L_2 vom deterministischen endlichen Automaten $E_2 = (Q_2, \Sigma, \delta_2, q_{2,0}, F_2)$ erkannt, dann ist $L_1 \cap L_2 = L(E)$ mit dem deterministischen endlichen Automaten $E = (Q_1 \times Q_2, \Sigma, \delta, (q_{1,0}, q_{2,0}), F_1 \times F_2)$, dessen Überföhrungsfunktion durch $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ definiert ist.
- (16) Setzt man die Gültigkeit der Aussage voraus, dass die Klasse der Typ-3-Sprachen abgeschlossen gegenüber Komplementbildung ist, dann ist mit zwei Typ-3-Sprachen L_1 und L_2 wegen $L_1 \cup L_2 = \Sigma^* \setminus ((\Sigma^* \setminus L_1) \cap (\Sigma^* \setminus L_2))$ auch deren Vereinigung eine Typ-3-Sprache.
- (17) Es sei L eine Typ-3-Sprache und $E = (Q, \Sigma, \delta, q_0, F)$ ein deterministischer endlicher Automat mit $L(E) = L$. Dann gilt für den deterministischen endlichen Automaten $E' = (Q, \Sigma, \delta, q_0, Q \setminus F)$: $L(E') = \Sigma^* \setminus L$.
- (18) Es seien L_1 und L_2 zwei Typ-3-Sprachen. Nach Satz 4.5-5 gibt es reguläre Ausdröcke p_1 bzw. p_2 , die L_1 bzw. L_2 repräsentieren. Der reguläre Ausdruck $(p_1 \cdot p_2)$ repräsentiert dann $L_1 \cdot L_2$ und ist damit nach Satz 4.5-5 eine Typ-3-Sprache.
- (19) Es sei L eine Typ-3-Sprache, die durch den regulären Ausdruck p repräsentiert wird. Dann wird nach Definition L^* durch $(p)^*$ repräsentiert und ist damit nach Satz 4.5-5 eine Typ-3-Sprache.

Wortproblem	
Typ 0	nicht entscheidbar
Typ 1	entscheidbar; für eine Typ-1-Grammatik $G = (\Sigma, N, S, R)$ und $w \in \Sigma^*$ mit $ w = n$ entscheidbar mit Komplexität der Ordnung $O(2^{O(n)})$
Typ 2	entscheidbar; für eine Typ-2-Grammatik $G = (\Sigma, N, S, R)$ und $w \in \Sigma^*$ mit $ w = n$ entscheidbar mit Komplexität der Ordnung $O(n^3)$
deterministisch kontextfrei	entscheidbar; für eine $LR(k)$ -Grammatik $G = (\Sigma, N, S, R)$ und $w \in \Sigma^*$ mit $ w = n$ entscheidbar mit Komplexität der Ordnung $O(n)$
Typ 3	entscheidbar; für eine Typ-3-Grammatik $G = (\Sigma, N, S, R)$ und $w \in \Sigma^*$ mit $ w = n$ entscheidbar mit Komplexität der Ordnung $O(n)$

Leerheitsproblem	
Typ 0	nicht entscheidbar
Typ 1	nicht entscheidbar
Typ 2 (bei gegebener kfr. Grammatik)	entscheidbar
deterministisch kontextfrei	entscheidbar
Typ 3	entscheidbar

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; es wird das für die Darstellung angemessene Modell verwendet.

In der Praxis gilt ein Algorithmus mit exponentiellem Laufzeitverhalten als **schwer durchführbar (intractable)**, ein Algorithmus mit polynomiellm 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 polynomiellm Laufzeitverhalten besitzen), während für andere Probleme bisher nur Lösungsalgorithmen mit exponentiellem Laufzeitverhalten bekannt sind.

Der Grund dafür, dass ein Verfahren mit exponentieller Laufzeit als nicht durchführbar gilt, wird aus den folgenden Tabelle sichtbar. Es werden dort jeweils fünf Funktionen $h_i : \mathbf{N}_{>0} \rightarrow \mathbf{R}, i = 1, \dots, 5$ 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 $h_i : \mathbf{N}_{>0} \rightarrow \mathbf{R}, i = 1, \dots, 5$. 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 $h_i(n_i) = y_0$	\bar{n}_i mit $h_i(\bar{n}_i) = 10 \cdot y_0$
1	$\log_2(n)$	n_1	$(n_1)^{10}$
2	\sqrt{n}	n_2	$100 \cdot n_2$
3	n	n_3	$10 \cdot n_3$
4	n^2	n_4	$\approx 3,162 \cdot n_4$
5	2^n	n_5	$n_5 + 3,322$

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 $V = \{v_1, \dots, v_n\}$ 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	$v_1v_2v_3v_4(v_1v_2)(v_2v_3)$	20 Zeichen
Nachfolgerliste	$(v_2)(v_1v_3)(v_2)()$	16 Zeichen
Adjazenzmatrix	0100/1010/0100/0000	19 Zeichen

Für einen ungerichteten Graphen $G = (V, E)$ mit der Knotenmenge $V = \{v_1, \dots, v_n\}$ 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 (Knotennummern) 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	$2 \cdot n + 2 \cdot m$	$2 \cdot n + 2 \cdot m + 2 \cdot m \cdot (\lfloor \log_2(n) \rfloor + 1)$
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. „größerer“ 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: $x = [x_1, \dots, x_n]$

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

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

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, \dots, x_n umfassenden eckigen Klammern und Trennsymbolen zwischen den Werten x_1, \dots, x_n . Häufig wird auch als Größe der Eingabe der Wert $n \cdot (size(\max\{x_1, \dots, x_n\}) + 1) + 1$ verwendet. Dieser Maßstab ist wohl etwas gröber als $size(x)$. Für Analysezwecke eignet er sich jedoch, da in einer Komplexitätsbetrachtung meist die Abschätzung $size(x) \in O(n \cdot size(\max\{x_1, \dots, x_n\}))$ verwendet wird. Zudem gibt es Eingaben x , deren Zahlen in der Folge x_1, \dots, x_n zusammen genau $n \cdot size(\max\{x_1, \dots, x_n\})$ 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 $V = \{x_1, \dots, x_n\}$ und m Junktoren;

$$size(F) = |F| = \text{Anzahl der Zeichen in } F.$$

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 $1, \dots, n$ durchnummeriert, so benötigt 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 $V = \{v_1, \dots, v_n\}$ und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \rightarrow \mathbf{R}_{\geq 0}$ 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: $[A, \vec{b}, \vec{z}]$

$A \in \mathbf{Z}^{m \times n}$ 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 $y \in SOL_{\Pi}(x)$ 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: $y^* \in SOL_{\Pi}(x)$ mit $m_{\Pi}(x, y^*) = \min\{m_{\Pi}(x, y) \mid y \in SOL_{\Pi}(x)\}$ bei einem Minimierungsproblem (d.h. $goal_{\Pi} = \min$)

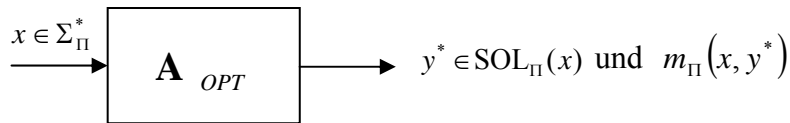
bzw.

$m_{\Pi}(x, y^*) = \max\{m_{\Pi}(x, y) \mid y \in SOL_{\Pi}(x)\}$ 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

$$m_{\Pi}^*(x) \geq K \text{ bei einem Maximierungsproblem}$$

(d.h. $goal_{\Pi} = \max$)

bzw.

$$m_{\Pi}^*(x) \leq K \text{ bei einem Minimierungsproblem}$$

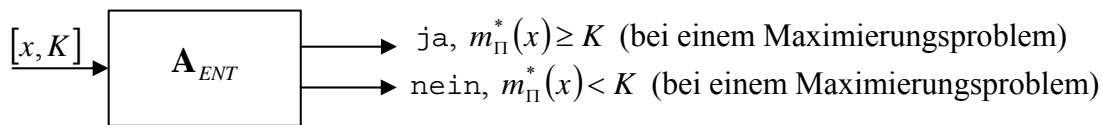
(d.h. $goal_{\Pi} = \min$)

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: $\mathbf{A}_{ENT}([x, K]) = \text{ja}$, falls $m_{\Pi}^*(x) \geq K$ bei einem Maximierungsproblem ist
bzw.
 $\mathbf{A}_{ENT}([x, K]) = \text{ja}$, falls $m_{\Pi}^*(x) \leq K$ bei einem Minimierungsproblem ist,
 $\mathbf{A}_{ENT}([x, K]) = \text{nein}$ 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 Lösungsverfahren 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 $V = \{v_1, \dots, v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \rightarrow \mathbf{N}$ gibt jeder Kante $e \in E$ ein **natürliches 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. $\text{SOL}(G) = \{T \mid T = \langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \dots, (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle \text{ ist eine Tour durch } G\}$

3. für $T \in \text{SOL}(G)$, $T = \langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \dots, (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$, 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 $T^* \in \text{SOL}(G)$, 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 $V = \{v_1, \dots, v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \rightarrow \mathbf{N}$ gibt jeder Kante $e \in E$ ein natürlichzahliges Gewicht; 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 $m^*(G) \leq K$ gibt,
 Entscheidung „nein“, sonst.

Ein Algorithmus, der bei Eingabe einer Instanz $[G, K]$ eine entsprechende Entscheidung fällt, werde mit \mathbf{A}_{TSPENT} bezeichnet. Die von \mathbf{A}_{TSPENT} bei Eingabe von $[G, K]$ getroffene ja/nein-Entscheidung sei $\mathbf{A}_{TSPENT}([G, K])$. Mit Hilfe von \mathbf{A}_{TSPENT} wird ein Algorithmus \mathbf{A}_{TSPOPT} zur Lösung des Optimierungsproblem konstruiert. Dabei wird wiederholt Binärsuche eingesetzt, um zunächst die Kosten einer optimalen Tour zu ermitteln.

Der Algorithmus \mathbf{A}_{TSPOPT} zur Lösung des Optimierungsproblem verwendet eine als Pseudocode formulierte Prozedur \mathbf{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 $V = \{v_1, \dots, v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \rightarrow \mathbf{N}$ gibt jeder Kante $e \in E$ ein natürlichzahliges Gewicht; ; 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 :=  $\sum_{e \in E} w(e)$ ;

  { Binärsuche auf dem Intervall [0..max]: }
  WHILE max - min >= 1 DO
    BEGIN
      t :=  $\left\lceil \frac{\min + \max}{2} \right\rceil$ ;
      IF  $\mathbf{A}_{TSPENT}([G, t]) = \text{„ja“}$ 
      THEN max := t
      ELSE min := t + 1;
    END;
  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 $\mathbf{A}_{TSPOPT}([G, t])$ erforderlich (Binärsuche). Es gilt:

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

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

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

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

Falls man also weiß, dass \mathbf{A}_{TSPOPT} 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 Entscheidungsprob-

lem. 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 A_{L_Π} , der für jede Eingabe $x \in \Sigma_\Pi^*$ mit $size(x) = n$ die ja/nein-Entscheidung $A_{L_\Pi}(x)$ 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 A_{L_Π} gibt, der für jede Eingabe $x \in \Sigma_\Pi^*$ mit $size(x) = n$ zum Finden der ja/nein-Entscheidung $A_{L_\Pi}(x)$ 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 \mathbf{P} enthalten. Diese Inklusion ist echt, wie das Beispiel der Sprache $L_4 = \{a^n b^n c^n \mid n \in \mathbf{N}, n \geq 1\}$ aus Kapitel 4.1 zeigt, die in \mathbf{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:

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

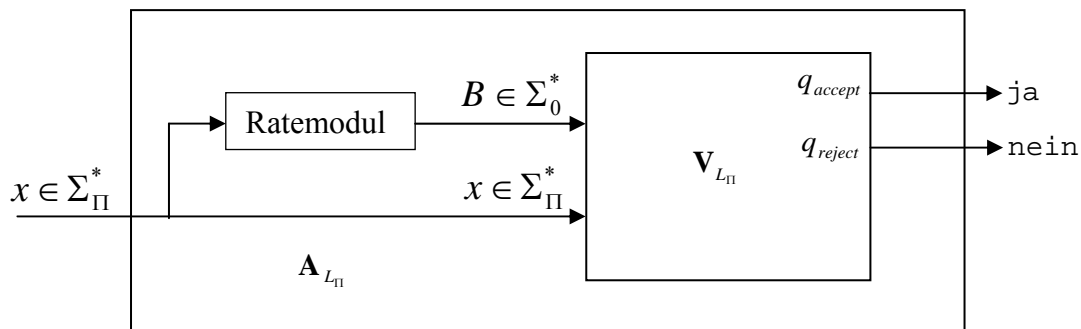
- die Klasse der Entscheidungsprobleme, die in einer Zeit gelöst werden können, die proportional zu einer Exponentialfunktion in der Größe der Eingabe ist:

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

Es gilt $\mathbf{P} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP}$. Die Frage, ob eine dieser Inklusionen echt ist, d.h. ob $\mathbf{P} \subset \mathbf{PSPACE}$ oder $\mathbf{PSPACE} \subset \mathbf{EXP}$ gilt, ist ein bisher ungelöstes Problem der Komplexitätstheorie. Man weiß jedoch, dass $\mathbf{P} \subset \mathbf{EXP}$ gilt.

In Kapitel 2.5 wird eine nichtdeterministische Turingmaschinen $TM = (Q, \Sigma, I, \delta, b, q_0, q_{accept})$ in zunächst unterschiedlichen Modell-Sichtweisen definiert. Entweder wird die Überföhrungsfunktion als partielle Funktion der Form $\delta : Q \times \Sigma^k \rightarrow \mathbf{P}(Q \times (\Sigma \times \{L, R, S\})^k)$ angegeben; während der Berechnung kann die Turingmaschine nichtdeterministische Schritte ausführen, indem sie bei Erreichen einer Konfiguration aus mehreren möglichen Folgekonfigurationen „die richtige“ auswählt. Oder die nichtdeterministische Turingmaschine wird als deterministische Turingmaschine gesehen, die mit einem „Ratemodul“ ausgestattet ist, das in nichtdeterministischer Weise zunächst eine Zusatzinformation generiert, deren Brauchbarkeit anschließend mit Hilfe einer Überföhrungsfunktion der Form $\delta : Q \times \Sigma^k \rightarrow Q \times (\Sigma \times \{L, R, S\})^k$ deterministisch verifiziert wird. Diese Überlegung führte schließlich auf die Definition eines nichtdeterministischen Algorithmus unter Einsatz eines Verifizierers. Die Definition wird hier noch einmal wiederholt. Da hier nur entscheidbare Probleme behandelt werden, stoppt hier der Verifizierer bei allen Eingaben.

Ein **nichtdeterministischer Algorithmus** A_{L_Π} für ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ besteht aus einem Ratemodul und einem Verifizierer V_{L_Π} . Bei Eingabe von $x \in \Sigma_\Pi^*$ erzeugt das Ratemodul auf nichtdeterministische Weise eine Zusatzinformation (einen Beweis) $B \in \Sigma_0^*$; dabei wird das Ratemodul nur einmal durchlaufen. Die Eingabe $x \in \Sigma_\Pi^*$ und der erzeugte Beweis $B \in \Sigma_0^*$ werden in den Verifizierer eingegeben, 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 V_{L_Π} bei Eingabe von x und B_x mit $V_{L_\Pi}(x, B_x) = \text{ja}$ stoppt;

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

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 $V_{L_\Pi}(x, B_x) = \text{ja}$ 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 $L_\Pi \in \text{NTIME}(f(n))$, wenn es einen nichtdeterministischen $f(n)$ -zeitbeschränkten Algorithmus zur Akzeptanz von L_Π gibt.

Die Aussage „das Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ liegt in $NTIME(f(n))$ “

steht synonym für $L_\Pi \in NTIME(f(n))$. In diesem Sinn bezeichnet $NTIME(f(n))$ die **Klasse der Entscheidungsprobleme, die auf nichtdeterministische Weise in der Zeitkomplexität $O(f(n))$ gelöst werden können.**

Eine der wichtigsten Klassen, die Klasse **NP** der Entscheidungsprobleme, die nichtdeterministisch mit polynomieller Zeitkomplexität gelöst werden können, ergibt sich, wenn f ein Polynom ist:

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

In Kapitel 2.5 wird gezeigt, dass ein $T(n)$ -zeitbeschränkter nichtdeterministischer Algorithmus durch einen deterministischen $O(c^{T(n)})$ -zeitbeschränkten Algorithmus simuliert werden kann. Setzt man $T(n) = n^k$, so sieht man unmittelbar, dass

$$NTIME(n^k) \subseteq TIME(2^{O(n^k)}) \subseteq TIME(2^{n^{k_1}}) \text{ für einen Wert } k_1 \geq k$$

gilt. Daraus folgt

$$\mathbf{NP} = \bigcup_{k=0}^{\infty} NTIME(n^k) \subseteq \bigcup_{k=0}^{\infty} TIME(2^{n^k}) = \mathbf{EXP}.$$

Da ein polynomiell zeitbeschränkter Algorithmus \mathbf{A}_{L_Π} , 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 $\mathbf{P} \subseteq \mathbf{EXP}$ fragt sich, welche der Inklusionen echt ist. Die dabei wichtigste Frage

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

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

Am Anfang des Kapitels wird die Klasse $\mathbf{PSPACE} = \bigcup_{k=0}^{\infty} \mathbf{SPACE}(n^k)$ 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 $\mathbf{NPSPACE}$ bezeichnen. Aus Satz 2.5-3 folgt direkt, dass $\mathbf{PSPACE} = \mathbf{NPSPACE}$ gilt.

5.3 Die Klassen \mathbf{P} und \mathbf{NP}

Das vorliegende Kapitel behandelt Beispiele aus den Klassen \mathbf{P} und \mathbf{NP} .

Zur Verdeutlichung des Unterschieds zwischen \mathbf{P} und \mathbf{NP} werden die entsprechenden Definitionen der beiden Klassen noch einmal gegenübergestellt:

Ein Entscheidungsproblem Π mit der Menge $L_{\Pi} \subseteq \Sigma_{\Pi}^*$ liegt in \mathbf{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 $\text{size}(x) = n$

Ausgabe: Entscheidung „ $x \in L_{\Pi}$ “, falls $\mathbf{A}_{L_{\Pi}}(x) = \text{ja}$ gilt,
 Entscheidung „ $x \notin L_{\Pi}$ “, falls $\mathbf{A}_{L_{\Pi}}(x) = \text{nein}$ 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 Verifizierers V_{L_Π} , 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 $|B_x| \leq C \cdot p(n)$

und $V_{L_\Pi}(x, B_x) = \text{ja}$;

Entscheidung „ $x \notin L_\Pi$ “ falls für alle Beweise $B \in \Sigma_0^*$ mit

$|B| \leq C \cdot p(n)$ gilt: $V_{L_\Pi}(x, B) = \text{nein}$.

Hierbei wird die Entscheidung des Algorithmus und insbesondere das Ergebnis der Verifikationsphase $V_{L_\Pi}(x, B)$ nach einer Anzahl von Schritten getroffen, die in $O(p(n))$ liegt.

Der deterministische Verifizierer V_{L_Π} 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 $L(V_{L_\Pi}) \in \mathbf{P}$.

Gehört umgekehrt eine Menge $L' \subseteq \Sigma^* \times \Sigma_0^*$ zu **P** und ist p ein Polynom, dann ist die Menge $L = \{x \mid x \in \Sigma^* \text{ und es gibt } B \in \Sigma_0^* \text{ mit } |B| \leq p(|x|) \text{ und } (x, B) \in L'\}$ 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 $|B| \leq p(|x|)$ 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

$L = \{x \mid x \in \Sigma^* \text{ und es gibt } B \in \Sigma_0^* \text{ mit } |B| \leq p(|x|) \text{ und } (x, B) \in L'\}$ 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 $V = \{x_1, \dots, x_n\}$ und m Junktoren, $f : V \rightarrow \{\text{TRUE}, \text{FALSCH}\}$ 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

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

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 $V = \{x_1, \dots, x_n\}$ 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 sich klar zu machen, 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 (viele sprechen 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 Entscheidungsalgorithmus 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 Wahrheitswerte 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 $V = \{x_1, \dots, x_n\}$ 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

$F = F_1 \wedge \dots \wedge F_m$ ist ein Boolescher Ausdruck in konjunktiver Normalform mit der Variablenmenge $V = \{x_1, \dots, x_n\}$ 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: $F = F_1 \wedge \dots \wedge F_m$

F ist ein Boolescher Ausdruck in konjunktiver Normalform mit der Variablenmenge $V = \{x_1, \dots, x_n\}$ 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 := F1 ∧ ... ∧ Fm`

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 =  $F_1 \wedge \dots \wedge F_m$  }
  C := { $F_1, \dots, F_m$ };
  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  $F_i = (y_{i_1} \vee y_{i_2})$ , wobei mindestens einem
              Literal ein Wahrheitswert zugewiesen ist) DO
        BEGIN
          IF (  $y_{i_1} = \text{TRUE}$  ) OR (  $y_{i_2} = \text{TRUE}$  )
          THEN erkläre  $F_i$  als erfüllt
          ELSE IF (  $y_{i_1} = \text{FALSE}$  ) AND (  $y_{i_2} = \text{FALSE}$  )
          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
        END
      ELSE BEGIN
          IF  $y_{i_1} = \text{FALSE}$ 

```

```

        THEN  $y_{i_2} := \text{TRUE}$ 
        ELSE  $y_{i_1} := \text{TRUE};$ 
        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 $(y_{i_1} \vee y_{i_2}) \wedge \dots \wedge (y_{i_m} \vee y_{i_n})$ mit Literalen y_{i_j} (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 $6 \cdot m - 1 \leq |F| \leq 8 \cdot m - 1 + 2 \cdot m \cdot (\lfloor \log_2(n) \rfloor + 1) = 2 \cdot m \cdot (\lfloor \log_2(n) \rfloor + 5) - 1$. Sind die Variablen in allen Klauseln von F paarweise verschieden, so ist $n = 2 \cdot m$; im allgemeinen ist $n \leq 2 \cdot m$. 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 $C \cdot n \cdot m \leq (2 \cdot C)/d^2 \cdot |F|^2$, 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 $C \cdot n \cdot m \leq 2 \cdot C \cdot m^2 \leq 2 \cdot C \cdot (m \cdot \log(m))^2 = (2 \cdot C)/d''^2 \cdot |F|^2$, 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 polynomiell Entscheidungungsverfahren 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. Boolesche Ausdrücke in allgemeiner Form (SAT) :**

Instanz: F

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

Zu entscheidende Menge:

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

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

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

Arbeitsweise des Verifizierers:

Der i -te Wert in B_F (für $i = 1, \dots, n$) 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 $V = \{x_1, \dots, x_n\}$ und m Junktoren; F hat die Form $F = F_1 \wedge F_2 \wedge \dots \wedge F_m$; 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:

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

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

Arbeitsweise des Verifizierers:

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

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

- **Kliquenproblem (KLIQUE):**

Instanz: $[G, k]$

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

Zu entscheidende Menge:

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

Eine Clique 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 $\text{bin}(i_1) \# \dots \# \text{bin}(i_k)$ von k natürlichen Zahlen in Binärdarstellung, die durch das Zeichen # getrennt sind, mit $i_j \leq n$ für $j = 1, \dots, k$;

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

Arbeitsweise des Verifizierers:

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

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

Instanz: $[a_1, \dots, a_n, b]$

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

Zu entscheidende Menge:

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

Beweis: $B_{[a_1, \dots, a_n, b]}$ ist eine 0-1-Folge x_1, \dots, 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: $[a_1, \dots, a_n]$

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

Zu entscheidende Menge:

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

Beweis: $B_{[a_1, \dots, a_n]}$ ist eine 0-1-Folge x_1, \dots, 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: $[a_1, \dots, a_n, b, k]$

a_1, \dots, a_n („Objekte“) sind natürliche Zahlen mit $0 < a_i \leq b$ für $i = 1, \dots, n$,
 $b \in \mathbf{N}$ („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 $f: \{1, \dots, n\} \rightarrow \{1, \dots, k\}$ gibt, so dass für alle $j \in \{1, \dots, k\}$ 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)

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

Beweis: $B_{[a_1, \dots, a_n, b, k]}$ ist eine Folge $\text{bin}(i_1) \# \dots \# \text{bin}(i_n)$ von n Zahlen in Binärdarstellung,

die durch das Zeichen # getrennt sind, mit $1 \leq i_j \leq k$ für $j = 1, \dots, n$;

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

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:

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

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

Zu entscheidende Menge:

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

Ein Hamiltonkreis ist eine Anordnung $(v_{\pi(1)}, v_{\pi(2)}, \dots, v_{\pi(n)})$ der Knoten mittels einer Permutation π der Knotenindizes, so dass für $i = 1, \dots, n-1$ 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 $\text{bin}(i_1) \# \dots \# \text{bin}(i_n)$ von n Zahlen in Binärdarstellung, die durch das Zeichen # getrennt sind, mit $1 \leq i_j \leq n$ für $j = 1, \dots, n$;

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

Arbeitsweise des Verifizierers:

Es wird geprüft, ob i_1, \dots, i_n eine Permutation der Knotenindizes ist, d.h. ob alle Werte paarweise verschieden sind und die Menge $\{1, \dots, n\}$ darstellen. Anschließend wird geprüft, ob die Bedingungen $(v_{i_t}, v_{i_{t+1}}) \in E$ für $t = 1, \dots, n-1$ 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 $V = \{v_1, \dots, v_n\}$ und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \rightarrow \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein Gewicht; $k \in \mathbf{R}_{>0}$.

Zu entscheidende Menge:

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

Eine Tour durch G ist eine geschlossene Kantenfolge

$$\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \dots, (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle \text{ mit } (v_{i_j}, v_{i_{j+1}}) \in E \text{ f\"ur } j = 1, \dots, n-1,$$

in der jeder Knoten des Graphen (als Anfangsknoten einer Kante) genau einmal vorkommt; die Kosten einer Tour $\langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \dots, (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$

ist die Summe der Gewichte der Kanten auf der Tour, d.h. der Ausdruck

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

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

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

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 $K \leq k$ 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 $V = \{v_1, \dots, v_n\}$ und Kantenmenge $E \subseteq V \times V$; $k \in \mathbf{N}$ mit $1 \leq k \leq n$

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.

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

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

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

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

Arbeitsweise des Verifizierers:

Es wird geprüft, ob durch $f(v_j) = \text{bin}(i_j)$ eine zulässige Kontenfärbung gegeben ist, d.h. ob $f(v_j) \neq f(v_l)$ 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: $[A, \vec{b}, \vec{z}]$

$A \in \mathbf{Z}^{m \times n}$ 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:

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

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 $A \cdot \vec{z} \leq / \geq \vec{b}$ 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 $V = \{v_1, \dots, v_n\}$ und der Kantenmenge $E \subseteq V \times V$; $v_i \in V$; $v_j \in V$; die Funktion $w: E \rightarrow \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein Gewicht;

$k \in \mathbf{R}_{>0}$.

Zu entscheidende Menge:

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

Ein einfacher Weg ist ein Pfad ohne Knotenwiederholungen

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

$$|B_{[G, v_i, v_j, k]}| \leq n \cdot (\lceil \log_2(n) \rceil + 1) + n - 1, \text{ d.h. } |B_{[G, v_i, v_j, k]}| \in O(n \cdot \log(n))$$

Arbeitsweise des Verifizierers:

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

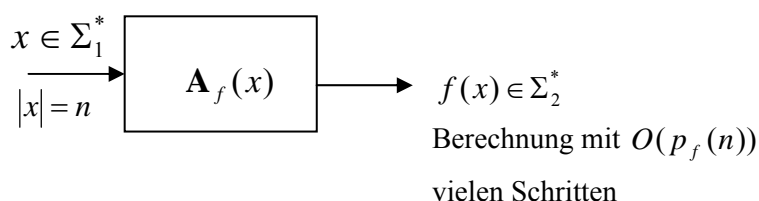
Man kennt heute mehrere tausend Probleme aus **NP**. In der angegebenen Literatur werden geordnet nach Anwendungsgebieten umfangreiche Listen von **NP**-Problemen aufgeführt.

5.4 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 $f: \Sigma_1^* \rightarrow \Sigma_2^*$, 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 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 $f : \Sigma_1^* \rightarrow \Sigma_2^*$, 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:

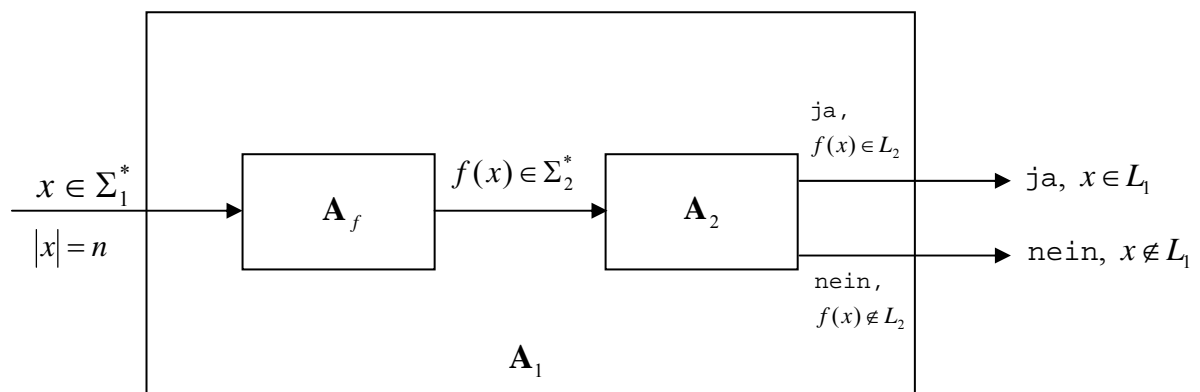
$$x \in L_1 \Rightarrow f(x) \in L_2 \text{ und } x \notin L_1 \Rightarrow f(x) \notin L_2.$$

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 $f : \Sigma_1^* \rightarrow \Sigma_2^*$ 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 $O(p_f(n) + p_2(p_f(n)))$, d.h. polynomiell Zeit

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

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

Zum Nachweis ist eine in polynomieller Zeit berechenbare 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 $F = F_1 \wedge \dots \wedge F_m$ ein Boolescher Ausdruck in konjunktiver Normalform. Die in F vorkommende Variablenmenge sei $V = \{x_1, \dots, x_n\}$. Jedes F_i hat die Form $F_i = (y_{i_1} \vee \dots \vee y_{i_k})$, 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 $\vec{z} = \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix}$, $A \in \mathbf{Z}^{m \times n}$ und $\vec{b} \in \mathbf{Z}^m$. Hier-

bei ergeben sich A und \vec{b} wie folgt: Für jede Klausel $F_i = (y_{i_1} \vee \dots \vee y_{i_k})$ wird eine Ungleichung $t_{i_1} + \dots + t_{i_k} \geq 1$ 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 $t_{i_1} + \dots + t_{i_k} \geq 1$ werden auf die rechte Seite der Ungleichung gebracht. Der nach dieser Umformung auf der rechten Seite entstehende Wert ist die Komponente b_i im Vektor \vec{b} , die i -te Zeile von A ergibt sich aus den Faktoren auf der linken Seite der Ungleichung. Offensichtlich ist $f(F)$ in polynomieller Zeit konstruierbar.

Es sei eine Belegung der Variablen in V gegeben, die den Booleschen Ausdruck F erfüllt. Dann wird das Ungleichungssystem $f(F) = [A, \vec{b}, \vec{z}]$ durch die Wertzuweisung

$$z_l = \begin{cases} 1 & \text{für } x_l = \text{TRUE} \\ 0 & \text{für } x_l = \text{FALSE} \end{cases} \text{ erfüllt. Ist umgekehrt eine Wertzuweisung an die Variablen in}$$

$f(F) = [A, \vec{b}, \vec{z}]$ gegeben, so dass das Ungleichungssystem erfüllt ist, so erfüllt die Belegung

$$x_l = \begin{cases} \text{TRUE} & \text{für } z_l = 1 \\ \text{FALSE} & \text{für } z_l = 0 \end{cases} \text{ den Booleschen Ausdruck } F.$$

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 $f(x) = f_2(f_1(x))$ 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 A_1 für L_1 ein deterministischer polynomiell zeitbeschränkter Algorithmus ist, wenn dieses für A_2 ein deterministischer polynomiell zeitbeschränkter Entscheidungsalgorithmus für L_2 ist.

Zu (iv): Die entsprechenden Aussagen wie in (iii) kann man treffen, wenn 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 $\{F \mid F \text{ ist ein Boolescher Ausdruck}\}$, 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 $L_{\Pi} \leq_m^p L_{\text{SAT}}$ zu zeigen. Die einzige Eigenschaft, die bezüglich L_{Π} in einem Beweis genutzt werden kann, ist die Tatsache, dass es eine 1-NDTM $TM_{L_{\Pi}}$ 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 $p_{L_{\Pi}}$ ein Polynom mit $p_{L_{\Pi}}(n) \geq n$ ist. Ist $x \in L_{\Pi}$, dann stoppt $TM_{L_{\Pi}}$ im akzeptierenden Zustand q_{accept} . Ist $x \notin L_{\Pi}$, dann stoppt $TM_{L_{\Pi}}$ in einem Zustand $q \neq q_{\text{accept}}$. $TM_{L_{\Pi}}$ besucht dabei höchstens die Zellen mit den Nummern $-p_{L_{\Pi}}(|x|), \dots, 0, 1, \dots, p_{L_{\Pi}}(|x|)+1$; hierbei wird das RV-Modell einer nichtdeterministischen Turingmaschine genommen (vgl. Kapitel 5.3).
 Zum Nachweis von $L_{\Pi} \leq_m^p L_{\text{SAT}}$ ist eine Funktion $f_{\Pi} : \Sigma_{\Pi}^* \rightarrow \Sigma_{\text{BOOLE}}^*$ 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 $p_{f_{\Pi}}$. 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 $TM_{L_{\Pi}}$ bei Eingabe von $x \in \Sigma_{\Pi}^*$ abläuft.

Die Zustandsmenge von $TM_{L_{\Pi}}$ sei $Q = \{q_0, \dots, q_k\}$, das Arbeitsalphabet von $TM_{L_{\Pi}}$ sei $\Sigma_{\Pi} = \{a_0, \dots, a_l\}$. Auf dem Eingabeband stehe das Wort $x \in \Sigma_{\Pi}^*$, $x = x_1 \dots x_n$ mit $x_i \in \Sigma_{\Pi}$ für $i = 1, \dots, n$. Es wird eine Reihe Boolescher Variablen erzeugt, deren Interpretation folgender Tabelle zu entnehmen ist:

Variable	Indizes	Interpretation
$zust_{t,q}$	$t = 0, \dots, p_{L_{\Pi}}(n)$ $q \in Q$	$zust_{t,q} = \text{TRUE}$ genau dann, wenn sich $TM_{L_{\Pi}}$ im Schritt t im Zustand q befindet Anzahl an Variablen $zust_{t,q} : (k+1) \cdot (p_{L_{\Pi}}(n)+1)$
$pos_{t,i}$	$t = 0, \dots, p_{L_{\Pi}}(n)$ $i = -p_{L_{\Pi}}(n), \dots, p_{L_{\Pi}}(n)+1$	$pos_{t,i} = \text{TRUE}$ genau dann, wenn sich der Schreib/Lesekopf von $TM_{L_{\Pi}}$ im Schritt t über der Zelle mit Nummer i befindet Anzahl an Variablen $pos_{t,i} : 2 \cdot (p_{L_{\Pi}}(n)+1)^2$
$band_{t,i,a}$	$t = 0, \dots, p_{L_{\Pi}}(n)$ $i = -p_{L_{\Pi}}(n), \dots, p_{L_{\Pi}}(n)+1$, $a \in \Sigma_{\Pi}$	$band_{t,i,a} = \text{TRUE}$ genau dann, wenn sich im Schritt t in der Zelle mit Nummer i das Zeichen a befindet Anzahl an Variablen $band_{t,i,a} : 2(l+1)(p_{L_{\Pi}}(n)+1)^2$

Der zu konstruierende Boolesche Ausdruck $f_{\Pi}(x)$ besteht aus mehreren Teilen und enthält insbesondere mehrmals eine Teilformeln G , die genau dann den Wahrheitswert TRUE erhält, wenn genau eine der in G vorkommenden Variablen den Wahrheitswert TRUE trägt. Sind y_1, \dots, y_m die in G vorkommenden Variablen, so lautet G :

$$G = G(y_1, \dots, y_m) = \left(\bigvee_{i=1}^m y_i \right) \wedge \left(\bigwedge_{j=1}^{m-1} \bigwedge_{i=j+1}^m \neg(y_j \wedge y_i) \right)$$

$$= \left(\bigvee_{i=1}^m y_i \right) \wedge \left(\bigwedge_{j=1}^{m-1} \bigwedge_{i=j+1}^m (\neg y_j \vee \neg y_i) \right).$$

Die zweite Zeile zeigt, dass $G = G(y_1, \dots, y_m)$ 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 $TM_{L_{\Pi}}$ 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:

$$R = \bigwedge_{t=0}^{p_{L_{\Pi}}(n)} \left[G(\text{zust}_{t,q_0}, \dots, \text{zust}_{t,q_k}) \wedge G(\text{pos}_{t,-p_{L_{\Pi}}(n)}, \dots, \text{pos}_{t,p_{L_{\Pi}}(n)+1}) \wedge \left(\bigwedge_{i=-p_{L_{\Pi}}(n)}^{p_{L_{\Pi}}(n)+1} G(\text{band}_{t,i,a_0}, \dots, \text{band}_{t,i,a_l}) \right) \right]$$

Die Anzahl an Literalen in R betragt

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

A beschreibt die Situation zum Zeitpunkt $t = 0$ (hierbei ist $b \in \Sigma_{\Pi}$ das Blankzeichen):

$$A = \text{zust}_{0,q_0} \wedge \text{pos}_{0,1} \wedge \left(\bigwedge_{i=1}^n \text{band}_{0,i,x_i} \right) \wedge \left(\bigwedge_{i=-p_{L_{\Pi}}(n)}^0 \text{band}_{0,i,b} \right) \wedge \left(\bigwedge_{i=n+1}^{p_{L_{\Pi}}(n)+1} \text{band}_{0,i,b} \right).$$

A enthalt $2(p_{L_{\Pi}}(n)+2) \in O(p_{L_{\Pi}}(n))$ viele Literale.

\ddot{U}_1 beschreibt den ubergang von der zum Zeitpunkt t bestehenden Konfiguration zur Konfiguration zum Zeitpunkt $t+1$ fur $t = 0, \dots, p_{L_{\Pi}}(n)$. Anstelle der Kopfbewegung L , S bzw. R werden hier die Werte $y = -1$, $y = 0$ bzw. $y = 1$ verwendet:

$$\ddot{U}_1 = \bigwedge_{t,q,i,a} \left[\text{zust}_{t,q} \wedge \text{pos}_{t,i} \wedge \text{band}_{t,i,a} \Rightarrow \vee \left\{ \text{zust}_{t+1,q'} \wedge \text{pos}_{t+1,i+y} \wedge \text{band}_{t+1,i,a'} \mid (q', a', y) \in \delta(q, a) \right\} \right]$$

Die Indizes nehmen die Werte $t = 0, \dots, p_{L_{\Pi}}(n)$, $q \in Q$, $i = -p_{L_{\Pi}}(n), \dots, p_{L_{\Pi}}(n)+1$ und $a \in \Sigma_{\Pi}$ an.

Die geschweifte Klammer in \ddot{U}_1 enthalt fur feste Werte t, q, a und i hochstens $3(k+1)(l+1)$ viele Literale, in der eckigen Klammer sind es daher hochstens $3(k+1)(l+1)+3$. Insgesamt enthalt \ddot{U}_1 hochstens

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

Um zu zeigen, wie sich auch \ddot{U}_1 in eine aquivalente Formel in konjunktiver Normalform umformen lasst, soll exemplarisch ein Ausdruck

$\text{zust}_{t,q} \wedge \text{pos}_{t,i} \wedge \text{band}_{t,i,a} \Rightarrow \vee \left\{ \text{zust}_{t+1,q'} \wedge \text{pos}_{t+1,i+y} \wedge \text{band}_{t+1,i,a'} \mid (q', a', y) \in \delta(q, a) \right\}$ innerhalb der eckigen Klammer, der im rechten Teil zwei Alternativen

$\text{zust}_{t+1,q'} \wedge \text{pos}_{t+1,i+y'} \wedge \text{band}_{t+1,i,a'}$ und $\text{zust}_{t+1,q''} \wedge \text{pos}_{t+1,i+y''} \wedge \text{band}_{t+1,i,a''}$ enthalt, umgeformt werden. Um den Vorgang ubersichtlich 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 fur 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 fur jedes Y_i und Z_i vier Literale notiert, so dass sich fur die Anzahl der Literale in der aquivalenten Formel innerhalb eines Ausdrucks in der eckigen Klammer im allgemeinen Fall eine Obergrenze von $12(k+1)(l+1)$ angeben lasst. Daher

bleibt die Anzahl der Literale in \ddot{U}_1 , selbst in der konjunktiven Normalform, von der Ordnung $O((p_{L_\Pi}(n))^2)$.

\ddot{U}_2 besagt, dass sich der Inhalt von Zellen, über denen der Schreib/Lesekopf nicht steht, nicht ändert:

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

Hierbei nehmen die Indizes die Werte $t = 0, \dots, p_{L_\Pi}(n)$, $i = -p_{L_\Pi}(n), \dots, p_{L_\Pi}(n)+1$ und $a \in \Sigma_\Pi$ an.

Die Anzahl an Literalen in \ddot{U}_2 beträgt $6(p_{L_\Pi}(n)+1)^2 \cdot (l+1) \in O((p_{L_\Pi}(n))^2)$.

\ddot{U}_2 lässt sich in eine äquivalente Formel in konfunktiver Normalform umformen, ohne die Anzahl an Literalen zu ändern:

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

E prüft nach, ob der Endzustand q_{accept} zum Zeitpunkt $t = p_{L_\Pi}(n)$ erreicht ist:

$$E = \text{zust}_{p_{L_\Pi}(n), q_{accept}}.$$

Insgesamt enthält $f_\Pi(x)$ eine Anzahl von Literalen der Ordnung $O((p_{L_\Pi}(n))^3)$, d.h. einer in der Länge des Eingabewortes polynomiellen Ordnung. Werden diese Literale durchnumeriert, so dass man $O((p_{L_\Pi}(n))^3)$ viele Literalpositionen bekommt, und dann binär kodiert (jede Zahl hat dann eine Länge der Ordnung $O(\log((p_{L_\Pi}(n))^3)) = O(\log(p_{L_\Pi}(n)))$), so hat $f_\Pi(x)$ eine Länge der Ordnung $O((p_{L_\Pi}(n))^4)$. 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 $\mathbf{P} = \mathbf{NP}$, so gehört L' zu **P**. Das bedeutet, dass es einen polynomiell zeitbeschränkten deterministischen Algorithmus $\mathbf{A}_{L'}$ gibt mit $\mathbf{A}_{L'}(w) = \text{ja}$ 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 $\mathbf{A}_{L'}(w) = \text{ja}$, 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

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

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

$$L(UTM) = \left\{ u\#w \mid u \in \{0,1\}^*, w \in \{0,1\}^* \text{ und } u \in L(K_w) \right\}$$

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:

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

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 $P \neq NP$ 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 Prog-

ramm- 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

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

$\text{3-CSAT} \leq_m^p \text{FÄRBBARKEIT}$,

$\text{3-CSAT} \leq_m^p \text{KLIQUE}$ und

$\text{3-CSAT} \leq_m^p \text{GERICHTETER HAMILTONKREIS}$

$\leq_m^p \text{UNGERICHTETER HAMILTONKREIS} \leq_m^p \text{HANDUNGSREISENDER}$.

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:

$$\text{SAT} \leq_m^p \text{3-CSAT}$$

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

$$L_{3\text{-CSAT}} = \left\{ F \left| \begin{array}{l} F \text{ ist ein Boolescher Ausdruck in konjunktiver Normalform,} \\ \text{jede Klausel enthalt genau 3 Literale,} \\ \text{und } F \text{ ist erfullbar} \end{array} \right. \right\}$$

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

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 erfullbar ist, wenn $f(F)$ erfullbar ist. Diese Zuordnung kann mit einem deterministischen polynomiell zeitbeschrankten Algorithmus (gemessen in $\text{size}(F)$) erfolgen.

Ein Boolescher Ausdruck F enthalt Variablen, Klammern und die Junktoren \neg , \wedge und \vee . Zunachst wird F durch einen logisch aquivalenten F_1 ersetzt, indem alle Negationszeichen unmittelbar vor die Variablen gebracht werden. Dabei werden die logischen Aquivalenzen $\neg(x \wedge y) \Leftrightarrow (\neg x \vee \neg y)$, $\neg(x \vee y) \Leftrightarrow (\neg x \wedge \neg y)$ und $\neg(\neg x) \Leftrightarrow x$ angewandt; hierbei stehen x und y fur 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 „Reprasantanten“ dargestellt und eine neue Menge M von logischen Ausdrucken gebildet:

Der Reprasantant eines Literals ist das Literal selbst. Enthalt F_1 die Teilformel $(F' \circ F'')$ mit $\circ \in \{\wedge, \vee\}$, wobei F' bzw. F'' durch v' bzw. v'' reprasentiert werden und dem Junktor \circ die Variable v zugeordnet wurde, so wird v der Reprasantant von $(F' \circ F'')$, und es wird der Ausdruck $(v \Leftrightarrow (v' \circ v''))$ in M aufgenommen. Man kann auerdem 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 zusatzlich ein Ausdruck $(v_0 \vee v_0 \vee v_0)$ in M aufgenommen (v_0 ist der Reprasantant des gesamten Ausdrucks F_1).

Alle Formeln in M werden mit \wedge verknupft; der so entstandene Ausdruck sei F_2 .

In F_2 wird jede Teilformel der Form

$$(v \Leftrightarrow (v' \wedge v'')) \text{ ersetzt durch } (\neg v \vee v' \vee v') \wedge (\neg v \vee v'' \vee v'') \wedge (v \vee \neg v' \vee \neg v'')$$

und jede Teilformel der Form

$$(v \Leftrightarrow (v' \vee v'')) \text{ ersetzt durch } (v \vee v \vee \neg v') \wedge (\neg v \vee v' \vee v'') \wedge (v \vee v \vee \neg v'').$$

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:

$$3\text{-CSAT} \leq_m^P \text{RUCKSACK}$$

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

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

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

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 $f(F) = [a_1, \dots, a_k, b]$ für das 0/1-Rucksack-Entscheidungsproblem zuordnet, so dass F

genau dann erfüllbar ist ($F \in L_{\text{CSAT}}$), wenn es eine Teilmenge $J \subseteq \{1, \dots, k\}$ mit $\sum_{j \in J} a_j = b$ gibt ($f(F) \in L_{\text{RUCKSACK}}$). Hierbei sind a_1, \dots, a_k und b natürliche Zahlen. Diese Zuordnung kann mit einem polynomiell (in $\text{size}(F)$) zeitbeschränkten deterministischen Algorithmus berechnet werden.

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

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

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, \dots, a_k und b werden in Dezimaldarstellung angegeben und wie folgt bestimmt.

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

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, \dots, v_n bestimmt, die wie b jeweils aus $m+n$ Dezimalziffern bestehen. In v_i für $i=1, \dots, n$ werden die Ziffernpositionen im ersten (führenden) Ziffernblock aus m Dezimalziffern von links nach rechts mit 1, ..., m durchnummeriert. 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 durchnummeriert. 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, \dots, v'_n bestimmt, die wie b jeweils aus $m+n$ Dezimalziffern bestehen. In v'_i für $i=1, \dots, n$ werden auf gleiche Weise die Ziffernpositionen im ersten (führenden) und zweiten Ziffernblock jeweils von links nach rechts durchnummeriert. 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, \dots, c_m bestimmt, die jeweils aus $m+n$ Dezimalziffern bestehen. Auch in c_i für $i = 1, \dots, m$ werden die Ziffernpositionen im ersten (führenden) Ziffernblock aus m Dezimalziffern von links nach rechts mit $1, \dots, m$ durchnummeriert. 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, \dots, d_m bestimmt, die jeweils aus $m+n$ Dezimalziffern bestehen. Auch in d_i für $i = 1, \dots, m$ werden die Ziffernpositionen im ersten (führenden) Ziffernblock aus m Dezimalziffern von links nach rechts durchnummeriert. 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

$$\begin{aligned} f(F) &= [a_1, \dots, a_k, b] \\ &= [v_1, \dots, v_n, v'_1, \dots, v'_n, c_1, \dots, c_m, d_1, \dots, d_m, b] \end{aligned}$$

gesetzt; d.h. die Zahlen a_1, \dots, a_k mit $k = 2 \cdot (n+m)$ sind $a_1 = v_1, \dots, a_n = v_n$, $a_{n+1} = v'_1, \dots, a_{2n} = v'_n$, $a_{2n+1} = c_1, \dots, a_{2n+m} = c_m$, $a_{2n+m+1} = d_1, \dots, a_{2(n+m)} = d_m$.

Die Bestimmung von $f(F)$ erfolgt deterministisch, und es ist $\text{size}(f(F)) \in O((\text{size}(F))^2)$.

Es sei $F = F_1 \wedge \dots \wedge F_m$ erfüllbar. Es wird eine Auswahl $J \subseteq \{1, \dots, k\}$ mit $\sum_{j \in J} a_j = b$ angegeben:

Jede Variable x_i für $i = 1, \dots, n$ 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 $j = 1, \dots, m$ 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 $1, \dots, m$, 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 $J \subseteq \{1, \dots, 2 \cdot (n + m)\}$ 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 $i = 1, \dots, n$ 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 $L_{\text{RUCKSACK}} = \left\{ [a_1, \dots, a_n, b] \mid \text{Es gibt eine Teilmenge } J \subseteq \{1, \dots, n\} \text{ mit } \sum_{j \in J} a_j = b \right\}$ und

$L_{\text{PARTITION}} = \left\{ [a_1, \dots, a_k] \mid \text{Es gibt eine Teilmenge } J \subseteq \{1, \dots, k\} \text{ mit } \sum_{j \in J} a_j = \sum_{j \notin J} a_j \right\}$

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 $i = 1, \dots, n$ 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 $\text{size}(I)$) zeitbeschränkt erfolgen, da die Berechnung von M mit einem Aufwand der Ordnung $O((\text{size}(I))^2)$ erfolgen kann.

Es gelte $I \in L_{\text{RUCKSACK}}$, d.h. es gibt eine Teilmenge $J \subseteq \{1, \dots, n\}$ mit $\sum_{j \in J} a_j = b$. Es wird

$J' = J \cup \{n+1\}$ gesetzt. Dann gilt:

$$M = \sum_{i=1}^n a_i = \sum_{j \in J} a_j + \sum_{j \notin J} a_j = b + \sum_{j \notin J} a_j \text{ und damit}$$

$$\sum_{j \in J'} a_j = \sum_{j \in J} a_j + (M - b + 1) = b + b + \sum_{j \notin J} a_j - b + 1 = \sum_{j \notin J} a_j + (b + 1) = \sum_{j \notin J} a_j + a_{n+2} = \sum_{j \in J'} a_j, \text{ d.h.}$$

$$I' \in L_{\text{PARTITION}}.$$

Sei umgekehrt $I' \in L_{\text{PARTITION}}$ mit der Auswahl $J' \subseteq \{1, \dots, n+2\}$.

Sind weder $n+1$ noch $n+2$ in J' , so ist

$$\sum_{j \in J'} a_j \leq \sum_{i=1}^n a_i = M < M - b + 1 + b + 1 = a_{n+1} + a_{n+2} \leq \sum_{j \notin J'} a_j, \text{ so dass } I' \notin L_{\text{PARTITION}} \text{ wäre. Also}$$

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 $I = [a_1, \dots, a_n, b]$ 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 $J \subseteq \{1, \dots, n\}$ erfüllt), denn:

$$\sum_{j \in J'} a_j = \sum_{\substack{j \in J' \\ j \neq n+1}} a_j + a_{n+1} = \sum_{j \notin J'} a_j = \sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j + a_{n+2} \quad \text{und} \quad M = \sum_{i=1}^n a_i = \sum_{j \in \{1, \dots, n\}} a_j + \sum_{j \in \{1, \dots, n\}} a_j \quad \text{und}$$

damit

$$\sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j + \sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j + \sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j - b + 1 = \sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j + M - b + 1 = \sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j + a_{n+1} = \sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j + a_{n+2}$$

(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 $\sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j + 1$ ab-

zieht:

$$2 \cdot \sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j - b = b + 1 - 1, \quad 2 \cdot \sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j = 2 \cdot b \quad \text{und} \quad \sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j = \sum_{j \in J} a_j = b.$$

2. Fall: $n+2 \in J'$, $n+1 \notin J'$

Dann ist $I = [a_1, \dots, a_n, b]$ in L_{RUCKSACK} mit der Indexmenge $J = \{1, \dots, n\} \setminus J'$, denn:

$$\sum_{j \in J'} a_j = \sum_{\substack{j \in J' \\ j \neq n+2}} a_j + a_{n+2} = \sum_{\substack{j \in J' \\ j \neq n+2}} a_j + b + 1 = \sum_{j \in J'} a_j = \sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j + a_{n+1} = \sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j + M - b + 1. \quad \text{Die}$$

Terme der Gleichung werden sortiert, und es ergibt sich:

$$2b = \sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j - \sum_{\substack{j \in J' \\ j \neq n+2}} a_j + M = \sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j - \sum_{\substack{j \in J' \\ j \neq n+2}} a_j + \sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j + \sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j = 2 \cdot \sum_{\substack{j \in J' \\ j \in \{1, \dots, n\}}} a_j = 2 \cdot \sum_{j \in J} a_j,$$

also auch in diesem Fall $\sum_{j \in J} a_j = b$.

///

Satz 5.5-4:

PARTITION \leq_m^p BINPACKING

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

$L_{\text{BINPACKING}} = \left\{ [a_1, \dots, a_h, b, k] \mid \text{Die Objekte } a_1, \dots, a_h \text{ lassen sich so auf } k \text{ Behälter} \right.$
 $\left. \text{der Behältergröße } b \text{ aufteilen, dass kein Behälter überläuft} \right\}$

steht die Aussage des Satzes für $L_{\text{PARTITION}} \leq_m^p L_{\text{BINPACKING}}$.

Beweis:

Es sei $I = [a_1, \dots, a_n]$ eine Instanz von PARTITION, die aus n natürlichen Zahlen mit $a_i > 0$ für $i = 1, \dots, n$ besteht. Der Instanz I wird die Instanz $I' = [a_1, \dots, a_n, b, 2]$ für BINPACKING

mit $b = \frac{1}{2} \cdot \left[\sum_{i=1}^n a_i \right]$ zugeordnet. Diese Zuordnung kann deterministisch polynomiell (in $\text{size}(I)$) zeitbeschränkt erfolgen, da die Berechnung von M mit einem Aufwand der Ordnung $O((\text{size}(I))^2)$ erfolgen kann.

Es gelte $I \in L_{\text{PARTITION}}$, d.h. es gibt eine Teilmenge $J \subseteq \{1, \dots, n\}$ mit $\sum_{j \in J} a_j = \sum_{j \notin J} a_j$. Daher ist

$$\sum_{i=1}^n a_i \text{ gerade und } b = \frac{1}{2} \cdot \left[\sum_{i=1}^n a_i \right] = \frac{1}{2} \cdot \sum_{i=1}^n a_i. \text{ Außerdem gilt } \sum_{i=1}^n a_i = \sum_{j \in J} a_j + \sum_{j \notin J} a_j = 2 \cdot \sum_{j \in J} a_j.$$

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 $I' = [a_1, \dots, a_n, b, 2]$ 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:

$$3\text{-CSAT} \leq_m^p \text{FÄRBBARKEIT}$$

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

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

des Satzes für $L_{3\text{-CSAT}} \leq_m^p L_{\text{FÄRBBARKEIT}}$.

Beweis:

Es sei F eine Instanz von 3-CSAT, d.h. F ist ein Boolescher Ausdruck in konjunktiver Normalform mit der Variablenmenge $\{x_1, \dots, x_n\}$ 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 $F = F_1 \wedge \dots \wedge F_m$ 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, \dots, 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 $1, \dots, 2 \cdot n + m$ nummerieren, wobei die obigen Knoten v_i die Nummern $1, \dots, n$, die Knoten v'_i die Nummern $n+1, \dots, 2 \cdot n$ und die Knoten f_j die Nummern $2 \cdot n + 1, \dots, 2 \cdot n + m$ erhalten; die hier gewählte Darstellung erleichtert jedoch die Beschreibung der Reduktion.

Die ungerichteten Kanten von G sind:

(w_i, w_j) mit $i = 1, \dots, n$, $j = 1, \dots, n$ und $i \neq j$,

(w_i, v_j) und (w_i, v'_j) mit $i = 1, \dots, n$, $j = 1, \dots, n$ und $i \neq j$,

(v_i, v'_i) für $i = 1, \dots, n$,

(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 $i = 1, \dots, n$ und $j = 1, \dots, m$.

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, \dots, w_n bilden einen vollständigen Teilgraph, so dass zu dessen zulässiger Färbung n Farben, etwa mit den Nummern $1, \dots, n$, 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 $x_j = \text{FALSE}$ ist. Ist $x_j = \text{TRUE}$, 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 $y = x_i = \text{TRUE}$, 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 $y = \neg x_i = \text{TRUE}$, 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, \dots, w_n besteht, n Farben, etwa mit den Nummern $1, \dots, n$, benötigt. Wie oben folgt, dass genau einer der Knoten v_j oder 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 $x_j = \text{TRUE}$ 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 $y_{j_1} = x_{i_1}$, dann ist f_j nach Konstruktion von G mit v'_i über eine Kante verbunden; ist $y_{j_1} = \neg x_{i_1}$, dann ist f_j mit v_i verbunden. In beiden Fällen werde der Knoten, der mit f_j aufgrund des Auftretens von y_{j_1} in F_j verbunden ist, der zu y_{j_1} 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 $3\text{-CSAT} \leq_m^p \text{RUCKSACK}$ 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:

$$3\text{-CSAT} \leq_m^p \text{KLIQUE}$$

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

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

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

Beweis:

Es sei F eine Instanz von 3-CSAT, d.h. F ist ein Boolescher Ausdruck in konjunktiver Normalform mit der Variablenmenge $\{x_1, \dots, x_n\}$ und m Klauseln, die jeweils genau drei Literale enthalten:

$F = F_1 \wedge \dots \wedge F_m$, 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 $V = \{ \langle j, l \rangle \mid 1 \leq j \leq m, 1 \leq l \leq 3 \}$. 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 $[G, m]$ eine Instanz von KLIQUE mit $\text{size}(G) \in O((\text{size}(F))^2)$.

Ist F mit einer Belegung der Variablen x_1, \dots, 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 $j \in \{1, \dots, m\}$, $k \in \{1, \dots, m\}$ und $j \neq k$ durch eine Kante in G verbunden, d.h. G enthält eine Clique der Größe m : $[G, m] \in L_{\text{KLIQUE}}$.

Enthält umgekehrt der konstruierte Graph G eine Clique der Größe m , so haben alle Knoten $\langle j, l \rangle$ dieser Clique 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 Clique korrespondieren: $y_{j_l} \neq \neg y_{k_h}$. Daher erhält man durch folgende Vorschrift eine widerspruchsfreie Belegung der Literale, die zu der gefundenen Clique 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, \dots, F_m und damit F .

///

Satz 5.5-7:

3-CSAT \leq_m^p GERICHTETER HAMILTONKREIS

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

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

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

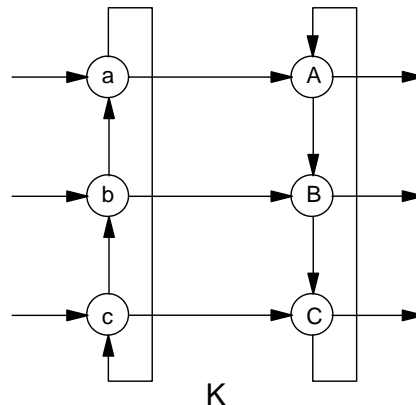
Beweis:

Es sei F eine Instanz von 3-CSAT, d.h. F ist ein Boolescher Ausdruck in konjunktiver Normalform mit der Variablenmenge $\{x_1, \dots, x_n\}$ und m Klauseln, die jeweils genau drei Literale enthalten:

$F = F_1 \wedge \dots \wedge F_m$, 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 $V = \{v_1, \dots, v_k\}$ 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 $\text{size}(F)$) zeitbeschränktes algorithmisches Verfahren.

Zunächst werden n Knoten v_1, \dots, v_n in V aufgenommen (sie entsprechen den Variablen x_1, \dots, 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, \dots, 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 $F_{i,1}, \dots, F_{i,j_i}$ eine Aufzählung der Klauseln, die das Literal x_i enthalten; die den Klauseln entsprechenden Teilgraphen sind $K_{i,1}, \dots, K_{i,j_i}$. 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 bestimmte Ausgang des Teilgraphen K_{i,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 Ausganges 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 $L_{\text{UNGERICHTETER HAMILTONKREIS}} = \left\{ G \mid \begin{array}{l} G \text{ ist ein ungerichteter Graph,} \\ \text{und } G \text{ besitzt einen Hamiltonkreis} \end{array} \right\}$ und

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

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 $V = \{v_1, \dots, v_n\}$.

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.

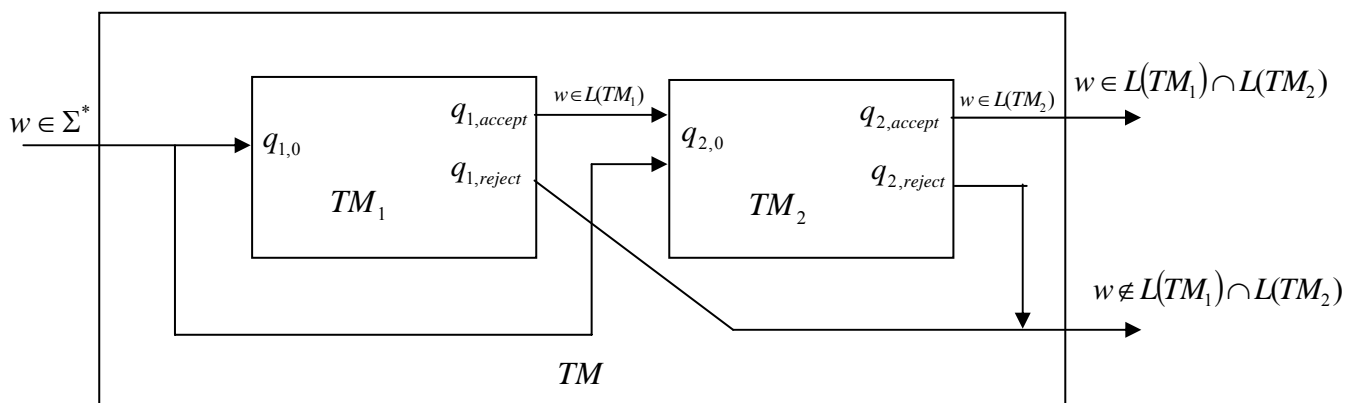
Bemerkung: Es ist bisher nicht bekannt, ob die Klasse **NP** auch bezüglich Komplementbildung abgeschlossen ist (siehe unten).

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.

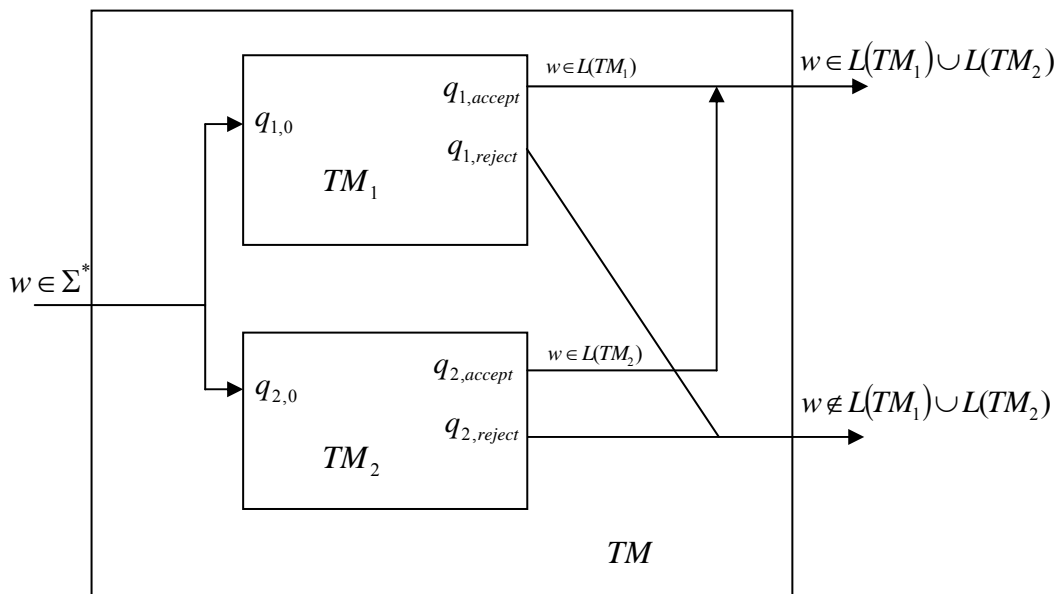
Zu L_1 bzw. L_2 gibt es nichtdeterministische Turingmaschinen TM_1 bzw. TM_2 mit $L_1 = L(TM_1)$ bzw. $L_2 = L(TM_2)$, die L_1 bzw. L_2 in polynomieller Zeit entscheiden; die dabei auftretenden Polynome seien p_1 bzw. p_2 .

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 Schrittzahl 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 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 Wortes $w \in \Sigma^*$ 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. $w = w_1 \dots w_i$ mit $w_k \in L$ für $k = 1, \dots, i$.

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 $i-1 \leq n$ Zahlen pos_1, \dots, pos_{i-1} mit $1 \leq pos_k \leq n$ für $k = 1, \dots, i-1$ in Binärformat nichtdeterministisch geraten. Die Zahlenfolge pos_1, \dots, pos_{i-1} zerlegt das Wort $w = a_1 \dots a_n$ in i Teilworte:

$$w = \underbrace{a_1 \dots a_{pos_1}}_{=w_0} \underbrace{a_{pos_1+1} \dots a_{pos_2}}_{=w_1} \dots \underbrace{a_{pos_{i-1}+1} \dots a_n}_{=w_{i-1}} = w_0 w_1 \dots w_{i-1}. \text{ Das Wort } w \text{ wird genau dann}$$

akzeptiert, wenn $w_k \in L(TM_0)$ für $k = 0, \dots, i-1$ gilt. Dazu wird i -mal eine akzeptierende Entscheidung von TM_0 gesucht.

Die Anzahl der Bits in der Zahlenfolge pos_1, \dots, pos_{i-1} ist wegen $1 \leq pos_k \leq n$ für $k = 1, \dots, i-1$ und $i-1 \leq n$ von der Ordnung $O(n \cdot \log(n))$. Alle Teilwörter w_k für $k = 0, \dots, i-1$ 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 \mathbf{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 \mathbf{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 $f : \mathbf{N} \rightarrow \mathbf{N}$ sei zeitkonstruierbar. Mit $\text{co-TIME}(f(n))$ werde die Klasse $\{L \mid L \subseteq \Sigma^* \text{ und } \Sigma^* \setminus L \in \text{TIME}(f(n))\}$ bezeichnet. Dann gilt $\text{TIME}(f(n)) = \text{co-TIME}(f(n))$.

Zu beachten ist, dass $\text{TIME}(f(n))$ 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 $A(F) = \text{ja}$ 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 $x_1 = \text{FALSE}$ und $x_2 = \text{FALSE}$), aber $F \notin \overline{L_{\text{SAT}}}$, denn F ist mit der Belegung $x_1 = \text{TRUE}$ und $x_2 = \text{FALSE}$ erfüllbar.

Es sei **NP** die Menge der **NP**-vollständigen Sprachen über einem endlichen Alphabet Σ . $\text{NPC} \subseteq \text{NP}$. Unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ gilt $\text{NPC} \cap \mathbf{P} = \emptyset$.

Es gilt folgender Satz²¹:

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 \mathbf{P} gehört, $A \leq_m^p B$, aber nicht $B \leq_m^p A$ gelten.

Satz 5.6-3 lässt sich folgendermaßen anwenden:

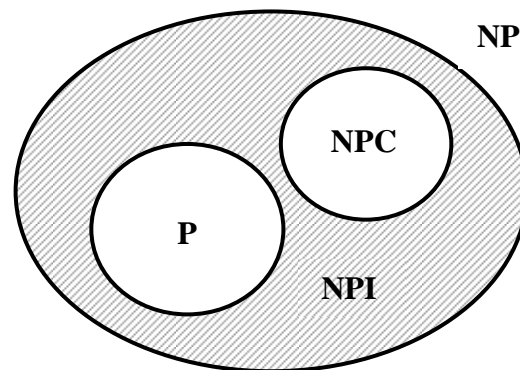
Es sei B eine **NP**-vollständige Sprache. Unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ gilt $B \notin \mathbf{P}$. Die Sprache $A = D \cap B$ gehört zu **NP**, da $D \in \mathbf{P}$ und $B \in \mathbf{NP}$ sind und die Klasse **NP** bezüglich Schnittbildung abgeschlossen ist. Nach dem obigen Satz gilt nicht $B \leq_m^p A$, also ist A nicht **NP**-vollständig. Folglich gilt:

Satz 5.6-4:

Unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ ist $\mathbf{NP} \setminus (\mathbf{P} \cup \text{NPC}) \neq \emptyset$.

Bezeichnet man $\mathbf{NP} \setminus (\mathbf{P} \cup \text{NPC})$ als die Menge **NPI** der **NP-unvollständigen Sprachen**, so zeigt **NP** (unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$) folgende Struktur:

²¹ Ladner, R.E.: On the structure of polynomial time reducibility, J.ACM, 22, 155-171, 1975.



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 $\text{ISO} \in \mathbf{P}$ noch der Nachweis $\text{ISO} \in \mathbf{NPC}$ gelungen ist.

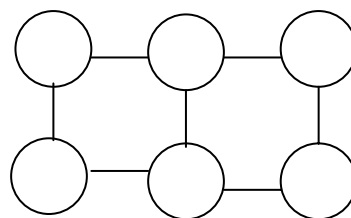
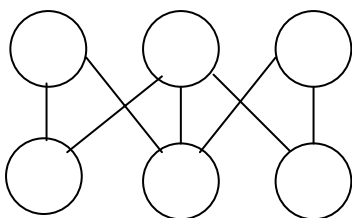
Graphenisomorphieproblem:

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 $f: V_1 \rightarrow V_2$ 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:

$$\mathbf{co-NP} = \{ \Sigma^* \setminus L \mid L \text{ ist eine Sprache über } \Sigma \text{ und } L \in \mathbf{NP} \}.$$

Aufgrund der Abgeschlossenheit der Klasse \mathbf{P} bezüglich Komplementbildung ist mit einer Sprache $L \in \mathbf{P}$, $L \subseteq \Sigma^*$, auch ihr Komplement $\Sigma^* \setminus L \in \mathbf{P}$. Daher gilt:

Satz 5.6-5:

$$\mathbf{P} \subseteq \mathbf{NP} \cap \mathbf{co-NP}.$$

Für $n \in \mathbf{N}$ sei $I_n = \{0, \dots, 2^n - 1\}$ die Menge natürlichen Zahlen, die mit n Bits darstellbar sind. Die Abbildung $f : I_n \rightarrow I_n$ sei bijektiv und besitze die Eigenschaften

- (i) Jeder Wert $f(x)$ ist mit polynomiellm 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 polynomiellm Zeitaufwand berechnet werden.

Dann liegt die Menge $L = \left\{ bin(x)\#bin(y) \mid x \in I_n, y \in I_n \text{ und } f^{-1}(y) < x \right\}$ in $\mathbf{NP} \cap \mathbf{co-NP} \setminus \mathbf{P}$.

Zum Beweis dieser Aussage sind drei Eigenschaften zu zeigen:

1. $L \in \mathbf{NP}$: 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 $\leq n$ 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. $L \in \mathbf{co-NP}$: 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 \mathbf{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 $1 \underbrace{0 \dots 0}_{(n-1)\text{-mal}} \# \text{bin}(y)$ in \mathbf{A}_L eingegeben; die Zeichenkette $1 \underbrace{0 \dots 0}_{(n-1)\text{-mal}}$ entspricht der Zahl 2^{n-1} . Antwortet \mathbf{A}_L mit ja (d.h. es ist $f^{-1}(y) < 2^{n-1}$), dann wird $1 \underbrace{0 \dots 0}_{(n-2)\text{-mal}} \# \text{bin}(y)$ 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 $11 \underbrace{0 \dots 0}_{(n-2)\text{-mal}} \# \text{bin}(y)$ 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 $\mathbf{NP} \neq \mathbf{co-NP}$ gilt. Aus der Gültigkeit von $\mathbf{NP} \neq \mathbf{co-NP}$ würde übrigens folgen, dass $\mathbf{P} \neq \mathbf{NP}$ ist, da wegen der Abgeschlossenheit der Klasse \mathbf{P} gegenüber Komplementbildung $\mathbf{P} = \mathbf{co-P}$ gilt. Falls also der Nachweis $\mathbf{NP} \neq \mathbf{co-NP}$ gelingt, wäre damit das **P-NP**-Problem gelöst. Es ist jedoch nicht auszuschließen, dass $\mathbf{NP} = \mathbf{co-NP}$ und $\mathbf{P} \neq \mathbf{NP}$ gelten.

Satz 5.6-6:

Gibt es eine Sprache $L \in \mathbf{NPC}$ mit $L \in \mathbf{co-NP}$, dann ist $\mathbf{NP} = \mathbf{co-NP}$.

Beweis:

Es sei $L \in \mathbf{NPC} \cap \mathbf{co-NP}$. Das der Sprache L zugrundeliegende Alphabet sei Σ , d.h. $L \subseteq \Sigma^*$.

Es gilt $\mathbf{NP} \subseteq \mathbf{co-NP}$: Dazu wird gezeigt, dass für jede Sprache $L_1 \in \mathbf{NP}$, $L_1 \subseteq \Sigma_1^*$, die Beziehung $L_1 \in \mathbf{co-NP}$ nachweisbar ist:

Wegen $L \in \mathbf{NPC}$ ist $L_1 \leq_m^p L$. Die dabei verwendete in polynomieller Zeit berechenbare totale Funktion sei f , d.h. es gilt $f: \Sigma_1^* \rightarrow \Sigma^*$ 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 $L \in \mathbf{co-NP}$ 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:

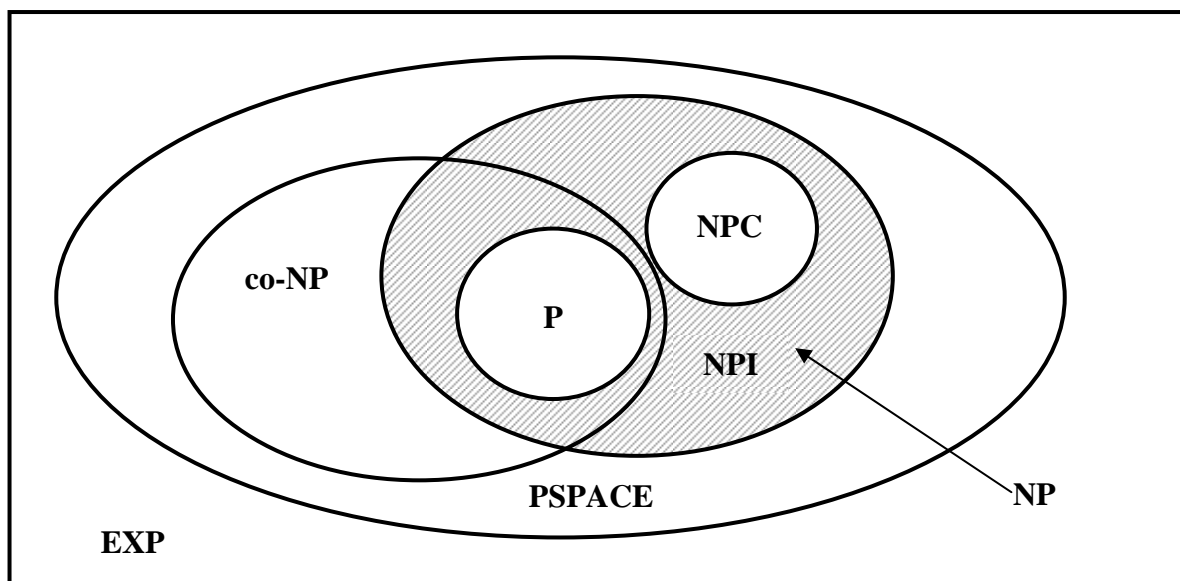
$$\begin{aligned} w \in \Sigma_1^* \setminus L_1 &\Leftrightarrow f(w) \in \Sigma^* \setminus L \\ &\Leftrightarrow \overline{NTM} \text{ stoppt im akzeptierenden Zustand} \\ &\Leftrightarrow \overline{NTM}_1 \text{ stoppt im akzeptierenden Zustand} \\ &\Leftrightarrow w \in L(\overline{NTM}_1). \end{aligned}$$

Daher ist $\Sigma_1^* \setminus L_1 \in \mathbf{NP}$ bzw. $L_1 \in \mathbf{co-NP}$.

Die umgekehrte Inklusion $\mathbf{co-NP} \subseteq \mathbf{NP}$ lässt sich mit ähnlichen Argumenten zeigen.

///

Unter den Annahmen $\mathbf{P} \neq \mathbf{NP}$ und $\mathbf{NP} \neq \mathbf{co-NP}$ 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 $y \in \text{SOL}_{\Pi}(x)$ 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: $y^* \in \text{SOL}_{\Pi}(x)$ mit $m_{\Pi}(x, y^*) = \min\{m_{\Pi}(x, y) \mid y \in \text{SOL}_{\Pi}(x)\}$ bei einem Minimierungsproblem (d.h. $goal_{\Pi} = \min$) bzw. $m_{\Pi}(x, y^*) = \max\{m_{\Pi}(x, y) \mid y \in \text{SOL}_{\Pi}(x)\}$ bei einem Maximierungsproblem (d.h. $goal_{\Pi} = \max$).

Der Wert $m_{\Pi}(x, y^*)$ einer optimalen Lösung wird auch mit $m_{\Pi}^*(x)$ bezeichnet.

Bemerkung: Der Subskript Π in m_{Π} bzw. m_{Π}^* wird in konkreten Beispielen häufig weggelassen.

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 $y \in \text{SOL}_{\Pi}(x)$ gilt $|y| \leq q(|x|)$, und für jedes y mit $|y| \leq q(|x|)$ ist in polynomieller Zeit entscheidbar, ob $y \in \text{SOL}_{\Pi}(x)$ 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 polynomieller Zeit erzeugt.

Der Verifizierer **V** für Π_{Ent} wird durch den folgenden Pseudocode gegeben:

```

FUNCTION V (  $[x, K]$  : ... ;
               $B$  : ... ) : ... ;
  {  $x \in \Sigma_{\Pi}^*$ ,  $K \in \mathbf{N}$ ,  $B \in \Sigma_0^*$  }

BEGIN { V }
  IF (  $|B| \leq q(|x|)$  ) AND (  $B \in \text{SOL}_{\Pi}(x)$  )           { Zeile 1 }
  THEN IF  $m_{\Pi}(x, B) \geq K$  THEN V := „ja“                 { Zeile 2 }
        ELSE V := „nein“
  ELSE V := „nein“;
END { V };

```

Zu zeigen ist

1. **V** arbeitet in polynomieller Zeit, gemessen in $\text{size}(x) = |x|$
2. $[x, K] \in L_{\Pi_{Ent}} \Leftrightarrow$ es gibt $B_x \in \Sigma_0^*$ mit $\mathbf{V}(x, B_x) = \text{ja}$.

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 $y^* \in \text{SOL}_{\Pi}(x)$ 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 $\mathbf{V}([x, K], y^*) = \text{ja}$.

Es gelte $[x, K] \notin L_{\Pi_{Ent}}$. Dann gilt für jedes $y \in \text{SOL}_{\Pi}(x)$: $m_{\Pi}(x, y) \leq m_{\Pi}^*(x) < K$. Es sei $B \in \Sigma_0^*$ mit $|B| \leq q(|x|)$. Ist $B \in \text{SOL}_{\Pi}(x)$, dann antwortet **V** wegen $m_{\Pi}(x, B) < K$ in Zeile 2 mit $\mathbf{V}([x, K], B) = \text{nein}$. Ist $B \notin \text{SOL}_{\Pi}(x)$, dann antwortet **V** wegen Zeile 1 mit $\mathbf{V}([x, K], B) = \text{nein}$.

///

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 $y^* \in \text{SOL}_{\Pi}(x)$ zusammen mit dem optimalen Wert $m_{\Pi}^*(x)$ der Zielfunktion in polynomieller Zeit (gemessen in $\text{size}(x) = |x|$) ermittelt.

Offensichtlich ist **PO** \subseteq **NPO**.

Satz 6-2:

Ist **P** \neq **NP**, dann ist **PO** \neq **NPO**.

Beweis:

Es sei **PO** = **NPO**. Dann folgt **P** = **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 **P** = **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 \mathbf{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 $y \in \text{SOL}_{\Pi}(x)$ bezeichnet

$$D(x, y) = \left| m_{\Pi}^*(x) - m_{\Pi}(x, y) \right|$$

den **absoluten Fehler von y bezüglich x** .

Ein Algorithmus \mathbf{A} ist ein **Approximationsalgorithmus (Näherungsalgorithmus)** für Π , wenn er bei Eingabe von $x \in \Sigma_{\Pi}^*$ eine zulässige Lösung liefert, d.h. wenn $\mathbf{A}(x) \in \text{SOL}_{\Pi}(x)$ gilt. \mathbf{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 \mathbf{NPO} wird mit **ABS** bezeichnet.

Das folgende Beispiel zeigt, dass $\mathbf{ABS} \neq \emptyset$ 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 $V = \{v_1, \dots, v_n\}$ und Kantenmenge $E \subseteq V \times V$,

2. eine Abbildung $c_V : V \rightarrow \mathbf{N}$ heißt **zulässige Knotenfärbung** von G , wenn für jede ungerichtete Kante $(v, w) \in E$ die Bedingung $c_V(v) \neq c_V(w)$ gilt²²,

$$\text{SOL}(I) = \left\{ c_V \mid c_V \text{ ist eine zulässige Knotenfärbung von } G \right\},$$

3. $m(I, c_V) = |c_V(V)|$ für $c_V \in \text{SOL}(I)$,

4. $goal = \min$.

²² Wie in Kapitel 1.1 bemerkt, wird eine Kante in der Form (v_i, v_j) anstelle von $\{v_i, v_j\}$ notiert.

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 $1, \dots, |c_V(V)|$ identifizieren. 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 in polynomieller Zeit (gemessen in der Größe des Eingabegraphen) eine zulässige Knotenfärbung:

Algorithmus COL zur Knotenfärbung eines ungerichteten Graphen:

Eingabe: $I = G = (V, E)$
 $G = (V, E)$ ist ein ungerichteter Graph mit Knotenmenge $V = \{v_1, \dots, v_n\}$ 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, \dots, v_i bereits mit den Farben $1, \dots, k$ gefärbt, so wird v_{i+1} mit einer der Farben $1, \dots, k$ gefärbt, falls dabei eine zulässige Knotenfärbung entsteht (gibt es mehrere Möglichkeiten einer Farbwahl aus $\{1, \dots, k\}$, dann wird die kleinste Farbnummer genommen). Ist eine zulässige Knotenfärbung mit Farben aus $\{1, \dots, k\}$ nicht möglich, d.h. es gibt mindestens 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: $\mathbf{COL}(I) = \{c_V(v_1), \dots, c_V(v_n)\} =$ 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 Graph mit der Knotenmenge $V = \{v_1, \dots, v_n\}$.

Für $i = 1, \dots, n$ sei k_i die Anzahl an Farben, die **COL** in seinem Ablauf zur Färbung der Knoten v_1, \dots, 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

$$k_i \leq 1 + \max\{d_i(v_j) \mid 1 \leq j \leq i\}$$

zeigen:

Es ist $k_1 = 1$ und $d_1(v_1) = 0$.

Es gelte $k_{i-1} \leq 1 + \max\{d_{i-1}(v_j) \mid 1 \leq j \leq i-1\}$. Mit G_{i-1} werde der durch $\{v_1, \dots, v_{i-1}\}$ induzierte Teilgraph von G bezeichnet. Alle Knoten in G_{i-1} sind (zulässig) gefärbt. Entsprechend bezeichne G_i den durch $\{v_1, \dots, v_i\}$ 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 $d_{i-1}(v_j) \leq d_i(v_j)$. Wird zur Färbung von v_i keine neue Farbe benötigt, so ist mit der Induktionsvoraussetzung

$k_i = k_{i-1} \leq 1 + \max\{d_{i-1}(v_j) \mid 1 \leq j \leq i-1\} \leq 1 + \max\{d_i(v_j) \mid 1 \leq j \leq i-1\}$. Da keine neue Farbe benötigt wird, ist $d_i(v_i) < k_{i-1}$ bzw. $d_i(v_i) + 1 \leq k_{i-1}$ und damit

$$k_i = k_{i-1} \leq 1 + \max\{d_i(v_j) \mid 1 \leq j \leq i\}.$$

Wird zur Färbung von v_i eine neue Farbe benötigt, so ist $k_i = k_{i-1} + 1$ und $d_i(v_i) \geq k_{i-1}$ bzw. $d_i(v_i) + 1 \geq k_{i-1} + 1 = k_i$. Damit ergibt sich ebenfalls

$$k_i \leq 1 + \max\{d_i(v_j) \mid 1 \leq j \leq i\}.$$

Für $i = n$ folgt $m(G, \mathbf{COL}(G)) = k_n \leq 1 + \max\{d_n(v_j) \mid 1 \leq j \leq n\} = 1 + \Delta(G)$.

Zu (ii): Da G mindestens eine Kante besitzt, ist $m^*(G) \geq 2$. Damit ergibt sich

$$|m^*(G) - m(G, \mathbf{COL}(G))| = m(G, \mathbf{COL}(G)) - m^*(G) \leq \Delta(G) + 1 - 2 = \Delta(G) - 1.$$

Die polynomielle Laufzeit des Verfahrens ergibt sich aus der Tatsache, dass jeder Knoten zu seiner Färbung nur einmal betrachtet wird und dabei jeweils nur die Knoten untersucht werden müssen, die mit ihm über eine Kante verbunden und bereits gefärbt sind. Insgesamt werden also während des Ablaufs des Verfahrens höchstens $O(n^3)$ viele Knoten betrachtet.

///

Bei Eingabe des folgenden Graphen G ermittelt der Algorithmus **COL** eine zulässige Kantenfärbung mit $\Delta(G)+1$ Farben:

$G = (V, E)$ mit der Knotenmenge $V = \{v_1, \dots, v_{2m}\}$ mit $m \geq 1$. 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_{2i} und allen Knoten mit ungeradem Index, außer mit dem Knoten v_{2i-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 $1, \dots, m$ färbt, ebenso die Knoten mit geradem Index, d.h. $m(G, \text{COL}(G)) = \Delta(G)+1$ und $|m^*(G) - m(G, \text{COL}(G))| = \Delta(G)-1$. 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 $|m^*(G) - m(G, \mathbf{A}(G))| \leq 3$ und $m(G, \mathbf{A}(G))/m^*(G) \leq 2$ 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 $m \geq 1$, 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 Induktionsvoraussetzung 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

$$g = \infty. \text{ Dann ist } m \leq \frac{g}{g-2} \cdot (n-2).$$

Beweis: Es bezeichne f_i die Anzahl der Flächen mit genau i begrenzenden Kanten

für $i = g, \dots, m$. Da jede Kante genau 2 Flächen begrenzt, ist $\sum_{i=g}^m i \cdot f_i = 2 \cdot m$.

Das ergibt $2 \cdot m = \sum_{i=g}^m i \cdot f_i \geq g \cdot \sum_{i=g}^m f_i = g \cdot f$. Mit 1. folgt

$$2 \cdot m \geq f \cdot g = 2 \cdot g - g \cdot n + g \cdot m, \text{ also}$$

$$(g-2) \cdot m \leq (n-2) \cdot g \text{ bzw. } m \leq \frac{g}{g-2} \cdot (n-2).$$

3. Es gilt $m \leq 3 \cdot n - 6$.

Beweis: Wegen $g \geq 3$ ist $g/(g-2) \leq 3$. 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 $\sum_{v \in V} d(v) = 2 \cdot m \geq 6 \cdot n$ bzw. $m \geq 3 \cdot n$ 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, \dots, v_n) der Knoten festgelegt: v_n sei ein Knoten mit minimalem

Grad in G . Sind die Knoten v_{i+1}, \dots, v_n bereits bestimmt, dann ist v_i ein Knoten mit minimalem Grad im Teilgraph, der durch $V \setminus \{v_{i+1}, \dots, v_n\}$ induziert wird. Es wird der Algorithmus **COL** mit der so definierten Reihenfolge der Färbung von $\{v_1, \dots, v_n\}$ angewendet. Der Knoten v_n hat wegen 4. einen Grad ≤ 5 . Der durch $V \setminus \{v_{i+1}, \dots, v_n\}$ 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 $V = \{v_1, \dots, v_n\}$ 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 $|m^*(G) - m(G, \mathbf{A}(G))| = 0$ und $m(G, \mathbf{A}(G))/m^*(G) = 1$. Im zweiten Fall ist $m^*(G) \geq 3$ und $m(G, \mathbf{A}(G)) = 6$, also $|m^*(G) - m(G, \mathbf{A}(G))| = m(G, \mathbf{A}(G)) - m^*(G) \leq 3$ und $m(G, \mathbf{A}(G))/m^*(G) \leq 2$.

///

Aus dem Beweis des bekannten Vier-Farben-Satz ergibt sich ein polynomiell zeitbeschränktes Verfahren, das für einen planaren Graphen eine zulässige Knotenfärbung mit 4 Farben ermittelt. Im Beweis des vorhergehenden Satzes wird ein Verfahren angegeben, das in polynomieller Zeit testet ob ein planarer Graph zulässig mit 2 Farben gefärbt werden kann. Das Problem zu entscheiden, ob ein gegebener planarer Graph mit 3 Farben gefärbt werden kann, ist bereits **NP-vollständig**²³.

²³ Siehe: Garey, M.R.; Johnson, D.: **Computers and Intractability, A Guide to the Theory of NP-Completeness**, Freeman, 1979.

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. $I = (A, M)$
 $A = \{a_1, \dots, a_n\}$ ist eine Menge von n Objekten und $M \in \mathbf{N}$ die „Rucksackkapazität“. Jedes Objekt a_i , $i = 1, \dots, n$, 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. $\text{SOL}(I) = \left\{ (x_1, \dots, x_n) \mid x_i = 0 \text{ oder } x_i = 1 \text{ für } i = 1, \dots, n \text{ und } \sum_{i=1}^n x_i \cdot w_i \leq M \right\}$
3. $m(I, (x_1, \dots, x_n)) = \sum_{i=1}^n x_i \cdot p_i$ für $(x_1, \dots, x_n) \in \text{SOL}(I)$
4. $goal = \max$

Lösung: Eine Folge x_1^*, \dots, x_n^* von Zahlen mit

- (1) $x_i^* = 0$ oder $x_i^* = 1$ für $i = 1, \dots, n$
- (2) $\sum_{i=1}^n x_i^* \cdot w_i \leq M$
- (3) $m(I, (x_1^*, \dots, x_n^*)) = \sum_{i=1}^n x_i^* \cdot p_i$ ist maximal unter allen möglichen Auswahlen x_1, \dots, 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 \mathbf{A} für das ganzzahlige 0/1-Rucksack-Maximierungsproblem, der bei Eingabe einer Instanz $I = (A, M)$ eine zulässige Lösung $\mathbf{A}(I) \in \text{SOL}(I)$ berechnet, für deren absoluter Fehler

$$D(I, \mathbf{A}(I)) = |m^*(I) - m(I, \mathbf{A}(I))| \leq k \quad \text{gilt.}$$

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 \mathbf{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 \mathbf{NP} -vollständig ist, folgt $\mathbf{P} = \mathbf{NP}$ (vgl. Satz 5.4-4).

Bei Annahme der Existenz von \mathbf{A} kann ein Algorithmus $\tilde{\mathbf{A}}$ definiert werden, der folgendermaßen arbeitet:

Aus einer Eingabe einer Instanz $I = (A, M)$ mit $A = \{(w_1, p_1), \dots, (w_n, p_n)\}$ für das ganzzahlige 0/1-Rucksack-Maximierungsproblem erzeugt $\tilde{\mathbf{A}}$ eine Instanz $\tilde{I} = (\tilde{A}, M)$ mit $\tilde{A} = \{(w_1, (k+1) \cdot p_1), \dots, (w_n, (k+1) \cdot p_n)\}$ (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 \mathbf{A} eingegeben. \mathbf{A} ermittelt eine

approximative Lösung $\mathbf{A}(\tilde{I}) = (x_1, \dots, x_n)$ mit $\sum_{i=1}^n x_i \cdot w_i \leq M$ und

$$\left| m^*(\tilde{I}) - m(\tilde{I}, \mathbf{A}(\tilde{I})) \right| = \left| m^*(\tilde{I}) - \sum_{i=1}^n x_i \cdot (k+1) \cdot p_i \right| \leq k \quad (\text{Ungleichung } (*)).$$

$\tilde{\mathbf{A}}$ gibt die von \mathbf{A} bei Eingabe von \tilde{I} ermittelte Lösung $\tilde{\mathbf{A}}(I) = \mathbf{A}(\tilde{I}) = (x_1, \dots, x_n)$ mit (dem Wert der Zielfunktion) $m(I, \tilde{\mathbf{A}}(I)) = \frac{m(\tilde{I}, \mathbf{A}(\tilde{I}))}{k+1}$ aus. Es ist $\tilde{\mathbf{A}}(I) \in \text{SOL}(I)$. Es gilt sogar $m(I, \tilde{\mathbf{A}}(I)) = m^*(I)$, d.h. $\tilde{\mathbf{A}}$ liefert in polynomieller Zeit (da \mathbf{A} in polynomieller Zeit arbeitet) eine optimale Lösung für die Eingabeinstanz I :

Da $m^*(\tilde{I})$ ein Vielfaches von $k+1$ ist, denn jeder Profit in \tilde{I} ist ein Vielfaches von $k+1$, kann man den Faktor $k+1$ auf der linken Seite des \leq -Zeichens in der Ungleichung (*) ausklammern und erhält

$$\left| m^*(\tilde{I}) - m(\tilde{I}, \mathbf{A}(\tilde{I})) \right| = \left| m^*(\tilde{I}) - \sum_{i=1}^n x_i \cdot (k+1) \cdot p_i \right| = (k+1) \cdot R \leq k$$

mit einer natürlichen Zahl R . Diese Ungleichung ist nur mit $R=0$ möglich, so dass

$$\left| m^*(\tilde{I}) - m(\tilde{I}, \mathbf{A}(\tilde{I})) \right| = 0 \text{ folgt, d.h. } \mathbf{A}(\tilde{I}) \text{ ist eine optimale Lösung für } \tilde{I}.$$

Jede zulässige Lösung für \tilde{I} ist auch eine zulässige Lösung für I , jedoch mit dem $(k+1)$ -fachen Profit. Umgekehrt ist jede zulässige Lösung für I eine zulässige Lösung für \tilde{I} . Damit folgt die Optimalität von $\tilde{\mathbf{A}}(I)$ (dazu ist $m(I, \tilde{\mathbf{A}}(I)) \geq m(I, y)$ für jedes $y \in \text{SOL}(I)$ zu zeigen):

$$m(I, \tilde{\mathbf{A}}(I)) = \frac{m(\tilde{I}, \mathbf{A}(\tilde{I}))}{k+1} = \frac{m^*(\tilde{I})}{k+1} \geq \frac{m(\tilde{I}, y)}{k+1} = m(I, y) \text{ für jedes } y \in \text{SOL}(I).$$

///

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 \mathbf{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 $m_{\Pi}(x, \mathbf{A}(x)) \leq m_{\Pi}^*(x)$ bei einem Maximierungsproblem bzw. $m_{\Pi}^*(x) \leq m_{\Pi}(x, \mathbf{A}(x))$ bei einem Minimierungsproblem.

Die **relative Approximationsgüte** $R_{\mathbf{A}}(x)$ von \mathbf{A} bei Eingabe einer Instanz $x \in \Sigma_{\Pi}^*$ wird definiert durch

$$R_{\mathbf{A}}(x) = \frac{m_{\Pi}^*(x)}{m_{\Pi}(x, \mathbf{A}(x))} \text{ bei einem Maximierungsproblem}$$

bzw.

$$R_{\mathbf{A}}(x) = \frac{m_{\Pi}(x, \mathbf{A}(x))}{m_{\Pi}^*(x)} \text{ bei einem Minimierungsproblem.}$$

Bemerkung: Um nicht zwischen Maximierungs- und Minimierungsproblem in der Definition unterscheiden zu müssen, kann man die relative Approximationsgüte von \mathbf{A} bei Eingabe einer Instanz $x \in \Sigma_{\Pi}^*$ auch durch

$$R_{\mathbf{A}}(x) = \max \left\{ \frac{m_{\Pi}^*(x)}{m_{\Pi}(x, \mathbf{A}(x))}, \frac{m_{\Pi}(x, \mathbf{A}(x))}{m_{\Pi}^*(x)} \right\}$$

definieren.

Es sei $r \geq 1$. Der Approximationsalgorithmus \mathbf{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 \mathbf{A} ein Approximationsalgorithmus für das Minimierungsproblem Π , und es gelte $m_{\Pi}(x, \mathbf{A}(x)) \leq r \cdot m_{\Pi}^*(x) + k$ für alle Instanzen $x \in \Sigma_{\Pi}^*$ (mit Konstanten r und k). Dann ist \mathbf{A} lediglich $(r+k)$ -approximativ und nicht etwa r -approximativ, jedoch **asymptotisch** r -approximativ (siehe unten).

Die **Klasse APX** besteht aus denjenigen Optimierungsproblemen aus **NPO**, für die es einen r -approximativen Algorithmus für ein $r \geq 1$ gibt.

Offensichtlich ist **APX** \subseteq **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 $R_A(x)$ eines Approximationsalgorithmus \mathbf{A} bei Eingabe einer Instanz $x \in \Sigma_{\Pi}^*$ gilt:

- (i) $1 \leq R_A(x)$. Je dichter $R_A(x)$ bei 1 liegt, um so besser ist die Approximation.
- (ii) Ist Π ein Maximierungsproblem, so bedeutet die Aussage „ $R_A(x) \leq r$ “ mit einer Konstanten $r \geq 1$ für alle Instanzen $x \in \Sigma_{\Pi}^*$:

$\frac{1}{r} \cdot m_{\Pi}^*(x) \leq m_{\Pi}(x, \mathbf{A}(x)) \leq m_{\Pi}^*(x)$ und $m_{\Pi}(x, \mathbf{A}(x)) \leq m_{\Pi}^*(x) \leq r \cdot m_{\Pi}(x, \mathbf{A}(x))$, insbesondere

$\frac{1}{R_A(x)} \cdot m_{\Pi}^*(x) \leq m_{\Pi}(x, \mathbf{A}(x)) \leq m_{\Pi}^*(x)$ und

$m_{\Pi}(x, \mathbf{A}(x)) \leq m_{\Pi}^*(x) \leq R_A(x) \cdot m_{\Pi}(x, \mathbf{A}(x))$.

- (iii) Ist Π ein Minimierungsproblem, so bedeutet die Aussage „ $R_A(x) \leq r$ “ mit einer Konstanten $r \geq 1$ für alle Instanzen $x \in \Sigma_{\Pi}^*$:

$m_{\Pi}^*(x) \leq m(x, \mathbf{A}(x)) \leq r \cdot m_{\Pi}^*(x)$ und $\frac{1}{r} \cdot m_{\Pi}(x, \mathbf{A}(x)) \leq m_{\Pi}^*(x) \leq m_{\Pi}(x, \mathbf{A}(x))$, insbesondere

$m_{\Pi}^*(x) \leq m_{\Pi}(x, \mathbf{A}(x)) \leq R_A(x) \cdot m_{\Pi}^*(x)$ und

$\frac{1}{R_A(x)} \cdot m_{\Pi}(x, \mathbf{A}(x)) \leq m_{\Pi}^*(x) \leq m_{\Pi}(x, \mathbf{A}(x))$.

Das folgende Optimierungsproblem liegt in **APX**:

Binpacking-Minimierungsproblem:

Instanz: 1. $I = [a_1, \dots, a_n]$

a_1, \dots, a_n („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, \dots, n$.

2. $\text{SOL}(I) = \left\{ [B_1, \dots, B_k] \left| \begin{array}{l} [B_1, \dots, B_k] \text{ ist eine Partition (disjunkte Zerlegung)} \\ \text{von } I \text{ mit } \sum_{a_i \in B_j} a_i \leq 1 \text{ für } j = 1, \dots, k \end{array} \right. \right\}$, d.h.

in einer zulässigen Lösung werden die Objekte so auf k „Behälter“ der Höhe 1 verteilt, dass kein Behälter „überläuft“.

3. Für $[B_1, \dots, B_k] \in \text{SOL}(I)$ ist $m(I, [B_1, \dots, B_k]) = k$, 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, \dots, B_{k^*} und (implizit) die Anzahl $k^* = m^*(I)$ 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 $i = 1, \dots, n$, so kann man das Binpacking-Problem durch Wahl der Behältergröße $b = \text{kgV}(q_1, \dots, q_n)$ 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: $I = [a_1, \dots, a_n]$,
 a_1, \dots, a_n („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, \dots, n$

Verfahren: $B_1 := a_1$;
 Sind a_1, \dots, a_i bereits in die Behälter B_1, \dots, 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 $[B_1, \dots, B_k]$ und $m(I, \mathbf{NF}(I)) = k$.

Satz 6.2-2:

Es sei $I = [a_1, \dots, a_n]$ eine Instanz des Binpacking-Minimierungsproblems. Mit

$a_{\max} = \max\{a_1, \dots, a_n\}$ gilt

$$m(I, \mathbf{NF}(I)) \leq \begin{cases} (1 + a_{\max}/(1 - a_{\max})) \cdot m^*(I) & \text{falls } 0 < a_{\max} \leq 1/2 \\ 2 \cdot m^*(I) & \text{falls } 1/2 < a_{\max} \leq 1 \end{cases}$$

$$R_{\mathbf{NF}}(I) \leq \begin{cases} (1 + a_{\max}/(1 - a_{\max})) & \text{falls } 0 < a_{\max} \leq 1/2 \\ 2 & \text{falls } 1/2 < a_{\max} \leq 1 \end{cases}, \text{ d.h. der Nextfit-Algorithmus ist 2-}$$

approximativ.

Das Binpacking -Minimierungsproblem liegt in **APX**.

Beweis:

Zur Herleitung der Aussage $m(I, \mathbf{NF}(I)) \leq 2 \cdot m^*(I)$ ist es nicht notwendig, $m^*(I)$ oder $m(I, \mathbf{NF}(I))$ zu kennen; es genügt eine Abschätzung von $m^*(I)$:

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 $j = 1, \dots, k$. 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 $1 \leq j < k$): Da b_{j+1} nicht mehr in den Behälter B_j passte, gilt $1 - u_j < b_j \leq u_{j+1}$ und damit $u_j + u_{j+1} > 1$. Summiert man diese $k - 1$ Ungleichungen auf, so erhält man

$$(u_1 + u_2) + (u_2 + u_3) + \dots + (u_{k-1} + u_k) = u_1 + 2u_2 + \dots + 2u_{k-1} + u_k > k - 1.$$

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

$$k < 2 \cdot \sum_{j=1}^k u_j + 1 - (u_1 + u_k) = 2 \cdot \sum_{i=1}^n a_i + 1 - (u_1 + u_k) \leq 2 \cdot \sum_{i=1}^n a_i + 1 \text{ und } k \leq 2 \cdot \sum_{i=1}^n a_i.$$

Trivialerweise gilt $m^*(I) \geq \sum_{i=1}^n a_i$ (hier gilt „ \leq “, wenn in einer optimalen Packung alle Behälter bis zur maximalen Füllhöhe 1 aufgefüllt werden). Damit ergibt sich $R_{\mathbf{NF}}(I) = k/m^*(I) \leq 2$.

Ist $0 < a_{\max} \leq 1/2$, dann gilt für $1 \leq j < k$: $1 - u_j < a_{\max}$ bzw. $1 - a_{\max} < u_j$. Summiert man diese Füllhöhen auf, so ergibt sich $(k-1) \cdot (1 - a_{\max}) < \sum_{j=1}^{k-1} u_j < \sum_{j=1}^k u_j = \sum_{i=1}^n a_i \leq m^*(I)$ und damit $(k-1) < 1/(1 - a_{\max}) \cdot m^*(I) = (1 + a_{\max}/(1 - a_{\max})) \cdot m^*(I)$ bzw. $k \leq (1 + a_{\max}/(1 - a_{\max})) \cdot m^*(I)$.

///

Die Grenze 2 im Nextfit-Algorithmus ist asymptotisch optimal: es gibt Instanzen I , für die $R_{\text{NF}}(I)$ beliebig dicht an 2 herankommt. Dazu betrachte man etwa eine Eingabeinstanz I mit $n = 2m$ vielen Objekten, wobei m eine gerade Zahl ist: $I = [a_1, \dots, a_{2m}]$. 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 $m(I, \text{NF}(I)) = m$

und $m^*(I) = m/2 + 1$, so dass $R_{\text{NF}}(I) = 2 \cdot \frac{m}{m+2}$ folgt, und dieser Wert kann für große m beliebig dicht an 2 herangehen.

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 ist linear in der Größe der Eingabeinstanz.

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

Instanz: 1. $I = (A, M)$

$A = \{a_1, \dots, a_n\}$ ist eine Menge von n Objekten und $M \in \mathbf{R}_{>0}$ die „Rucksackkapazität“. Jedes Objekt a_i , $i = 1, \dots, n$, 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. $\text{SOL}(I) = \left\{ (x_1, \dots, x_n) \mid x_i = 0 \text{ oder } x_i = 1 \text{ für } i = 1, \dots, n \text{ und } \sum_{i=1}^n x_i \cdot w_i \leq M \right\}$; man

beachte, dass hier nur Werte $x_i = 0$ oder $x_i = 1$ zulässig sind

3. $m(I, (x_1, \dots, x_n)) = \sum_{i=1}^n x_i \cdot p_i$ für $(x_1, \dots, x_n) \in \text{SOL}(I)$

4. $goal = \max$

Lösung: Eine Folge x_1^*, \dots, x_n^* von Zahlen mit

$$(1) \quad x_i^* = 0 \text{ oder } x_i^* = 1 \text{ für } i = 1, \dots, n$$

$$(2) \quad \sum_{i=1}^n x_i^* \cdot w_i \leq M$$

$$(3) \quad m(I, (x_1^*, \dots, x_n^*)) = \sum_{i=1}^n x_i^* \cdot p_i \text{ ist maximal unter allen möglichen Auswahlen}$$

x_1, \dots, x_n , die (1) und (2) erfüllen.

Zunächst wird eine **Verallgemeinerung des 0/1-Rucksackproblems** betrachtet, in dem die Bedingung (1) durch

$$(1)' \quad 0 \leq x_i^* \leq 1 \text{ für } i = 1, \dots, n$$

ersetzt, d.h. es können auch anteilige Profite und Gewichte der Objekte berücksichtigt werden. Eine Instanz $I = (A, M)$ dieses **Allgemeinen Rucksackproblems** wird in den folgenden Algorithmus **RUCK**, hier in informeller Beschreibung, eingegeben.

Die Objekte der Instanz werden nach absteigenden Werten p_i/w_i sortiert und in der durch diese Sortierung bestimmten Reihenfolge bearbeitet. Es wird daher angenommen, dass bereits $p_1/w_1 \geq p_2/w_2 \geq \dots \geq p_n/w_n$ gilt. Es wird $x_1 = 1$ gesetzt. Sind die Werte x_1, \dots, x_{i-1} bereits bestimmt und gilt $M - \sum_{l=1}^{i-1} w_l \geq w_i$, so wird $x_i = 1$ gesetzt und zum nächsten Objekt übergegangen.

Andernfalls ergibt sich x_i zu $x_i = \left(M - \sum_{l=1}^{i-1} w_l \right) / w_i$ und $x_l = 0$ für $l = j+1, \dots, n$, und

das Verfahren stoppt. Da $\sum_{i=1}^n w_i > M$ angenommen werden kann, gibt es einen Index j mit

$$x_1 = \dots = x_{j-1} = 1,$$

$$x_j = \left(M - \sum_{i=1}^{j-1} w_i \right) / w_j,$$

$$x_{j+1} = \dots = x_n = 0.$$

Der ermittelte Profit ist $m(I, \mathbf{RUCK}(I)) = \sum_{i=1}^{j-1} p_i + \left(\left(M - \sum_{i=1}^{j-1} w_i \right) / w_j \right) \cdot p_j$.

Satz 6.2-3:

Der Algorithmus **RUCK** ermittelt in polynomieller Zeit eine optimale Lösung für eine Instanz $I = (A, M)$ des Allgemeinen Rucksackproblems.

Beweis:

Für die Instanz I des Allgemeinen Rucksackproblems gelte $p_1/w_1 \geq p_2/w_2 \geq \dots \geq p_n/w_n$. Die von **RUCK** gelieferte Lösung sei x_1, \dots, x_n mit

$$x_1 = \dots = x_{j-1} = 1,$$

$$x_j = \left(M - \sum_{i=1}^{j-1} w_i \right) / w_j,$$

$$x_{j+1} = \dots = x_n = 0,$$

$$\sum_{i=1}^n x_i \cdot w_i = M \quad (\text{nach Konstruktion}).$$

Es sei y_1, \dots, y_n mit $0 \leq y_i \leq 1$ für $i = 1, \dots, n$ eine optimale Lösung, d.h. $\sum_{i=1}^n y_i \cdot p_i = m^*(I)$, für

die wegen der Optimalität $\sum_{i=1}^n y_i \cdot w_i = M$ gilt. Es werde angenommen, dass sich diese optimale Lösung von x_1, \dots, x_n unterscheidet (andernfalls ist die von **RUCK** ermittelte Lösung bereits als optimal gezeigt). Es sei k der kleinste Index, so dass $x_k \neq y_k$ gilt. Dann ist $k \leq j$ und $y_k < x_k$; denn:

1. Fall: $k < j$: $y_k \leq 1 = x_k$ und $x_k \neq y_k$ ergibt $y_k < x_k$.

2. Fall: $k = j$: Aus $M = \sum_{i=1}^n x_i \cdot w_i = \sum_{i=1}^n y_i \cdot w_i$, $x_i = y_i$ für $i = 1, \dots, k-1$ und $x_i = 0$ für

$$i = j+1, \dots, n \quad \text{folgt} \quad x_k \cdot w_k = y_k \cdot w_k + \sum_{i=k+1}^n y_i \cdot w_i \quad \text{und} \quad x_k = y_k + \sum_{i=k+1}^n y_i \cdot \frac{w_i}{w_k}.$$

Wegen $x_k \neq y_k$ sind nicht alle Werte $y_i = 0$ für $i = k+1, \dots, n$; daher ist $y_k < x_k$.

3. Fall: $k > j$: Mit $x_k \neq y_k$ und $x_k = 0$, also $y_k > 0$, ergibt sich der Widerspruch

$$\begin{aligned} M &= \sum_{i=1}^n y_i \cdot w_i = \sum_{i=1}^{j-1} y_i \cdot w_i + y_j \cdot w_j + \sum_{i=j+1}^{k-1} y_i \cdot w_i + y_k \cdot w_k + \sum_{i=k+1}^n y_i \cdot w_i \\ &= \sum_{i=1}^{j-1} x_i \cdot w_i + x_j \cdot w_j + \sum_{i=j+1}^{k-1} x_i \cdot w_i + y_k \cdot w_k + \sum_{i=k+1}^n y_i \cdot w_i \\ &= M + y_k \cdot w_k + \sum_{i=j+1}^n y_i \cdot w_i > M. \end{aligned}$$

Dieser Fall kann also nicht auftreten.

Daher ist $M = \sum_{i=1}^{k-1} x_i \cdot w_i + x_k \cdot w_k + \sum_{i=k+1}^n x_i \cdot w_i$ und $M = \sum_{i=1}^{k-1} y_i \cdot w_i + y_k \cdot w_k + \sum_{i=k+1}^n y_i \cdot w_i$. Durch

Differenzbildung und Multiplikation mit p_k/w_k folgt

$$p_k/w_k \cdot (x_k - y_k) \cdot w_k = \sum_{i=k+1}^n w_i \cdot (y_i - x_i) \cdot p_k/w_k \geq \sum_{i=k+1}^n w_i \cdot (y_i - x_i) \cdot p_i/w_i \text{ und}$$

$$p_k \cdot x_k \geq p_k \cdot y_k + \sum_{i=k+1}^n p_i \cdot y_i - \sum_{i=k+1}^n p_i \cdot x_i \quad \text{bzw.} \quad \sum_{i=k}^n p_i \cdot x_i \geq \sum_{i=k}^n p_i \cdot y_i. \text{ Wegen der Wahl von } k$$

ist damit $\sum_{i=1}^n p_i \cdot x_i \geq \sum_{i=1}^n p_i \cdot y_i = m^*(I)$, also $\sum_{i=1}^n p_i \cdot x_i = m^*(I)$.

Die Laufzeit des Verfahrens ist polynomiell in der Größe der Eingabe: Mit $z_{\max} = \max\{w_1, \dots, w_n, p_1, \dots, p_n, M\}$ ist $\text{size}(I) \in O(n \cdot \log(z_{\max}))$. Die Bildung der Quotienten p_i/w_i erfordert nach den Betrachtungen in Kapitel 2.3 einen Aufwand der Ordnung $O(n \cdot (\log(z_{\max}))^2)$; die anschließende Sortierung der Objekte nach absteigenden Größen p_i/w_i ist von der Ordnung $O(n \cdot \log(n))$; die Bestimmung der Werte x_i ist von der Ordnung $O(n \cdot \log(z_{\max}))$; insgesamt ist der Aufwand begrenzt durch einen Wert der Ordnung $O((\text{size}(I))^2)$, also polynomiell in $\text{size}(I)$.

///

Satz 6.2-3 legt nahe, ein ähnliches Verfahren für die Lösung des 0/1-Rucksackproblems zu entwickeln. Das Ergebnis ist der Approximationsalgorithmus **RUCK_APP**, der hier informell beschrieben wird:

- Die Objekte der Instanz $I = (A, M)$ werden nach absteigenden Werten p_i/w_i sortiert und in der durch diese Sortierung bestimmten Reihenfolge bearbeitet. Es wird daher angenommen, dass bereits $p_1/w_1 \geq p_2/w_2 \geq \dots \geq p_n/w_n$ gilt. Es wird $x_1 = 1$ gesetzt. Sind die Werte x_1, \dots, x_{i-1} bereits bestimmt und gilt $M - \sum_{l=1}^{i-1} w_l \geq w_i$, so wird $x_i = 1$, andernfalls $x_i = 0$ gesetzt und zum nächsten Objekt übergegangen.

- Es sei $p_{\max} = \max\{p_i \mid i = 1, \dots, n\}$. und k ein Index mit $x_k = p_{\max}$. Es wird

$$\mathbf{RUCK_APP}(I) = \begin{cases} x_1, \dots, x_n \text{ aus Teil 1} & \text{falls } \sum_{i=1}^n p_i \cdot x_i \geq p_{\max} \\ x_k = 1 \text{ und } x_i = 0 \text{ für } i \neq k & \text{falls } \sum_{i=1}^n p_i \cdot x_i < p_{\max} \end{cases} \quad \text{mit dem Wert}$$

$m(I, \mathbf{RUCK_APP}(I))$ der Zielfunktion ausgegeben.

Satz 6.2-4:

Für jede Instanz $I = (A, M)$ des 0/1-Rucksack-Maximierungsproblems gilt:
 $1 \leq m^*(I)/m(I, \mathbf{RUCK_APP}(I)) \leq 2$, d.h. das 0/1-Rucksack-Maximierungsproblem
 liegt in **APX**.

Beweis:

Es sei a_j das erste Objekt, für das im 1. Teil des Algorithmus **RUCK_APP** $x_i = 0$ gesetzt wird. Es gilt $\bar{w} = \sum_{i=1}^{j-1} w_i \leq M$, also $w_j > M - \bar{w}$. Der bisher entstandene Profit ist $\bar{p} = \sum_{i=1}^{j-1} p_i$.

Es werde die Verallgemeinerung des 0/1-Rucksack-Maximierungsproblems betrachtet, in dem die Forderung „ $x_i = 0$ oder $x_i = 1$ “ durch „ $0 \leq x_i \leq 1$ “ ersetzt ist. Für die so modifizierte Aufgabenstellung wird im 1. Teil des Algorithmus **RUCK_APP** $x_j = (M - \bar{w})/w_j$ und $x_{j+1} = \dots = x_n = 0$ gesetzt. Nach Satz 6.2-3 ist jetzt bereits eine optimale Lösung des modifizierten 0/1-Rucksack-Maximierungsproblems gefunden, und zwar mit einem Profit $\bar{p} + ((M - \bar{w})/w_j) \cdot p_j$. Da jede zulässige Lösung des (ursprünglichen) 0/1-Rucksack-Maximierungsproblems eine zulässige Lösung des modifizierten 0/1-Rucksack-Maximierungsproblems ist, folgt $m^*(I) \leq \bar{p} + ((M - \bar{w})/w_j) \cdot p_j < \bar{p} + p_j$ (die letzte Ungleichung ergibt sich aus $w_j > M - \bar{w}$).

Außerdem gilt $\bar{p} \leq \max\{\bar{p}, p_{\max}\} \leq m(I, \mathbf{RUCK_APP}(I)) \leq m^*(I)$.

Es werden zwei Fälle unterschieden:

1.Fall: $p_j \leq \bar{p}$

Dann gilt $m^*(I) < 2 \cdot \bar{p} \leq 2 \cdot m(I, \mathbf{RUCK_APP}(I))$.

2.Fall: $p_j > \bar{p}$

Dann gilt $p_{\max} \geq p_j > \bar{p}$ und $m^*(I) < \bar{p} + p_j < 2 \cdot p_{\max} \leq 2 \cdot m(I, \mathbf{RUCK_APP}(I))$.

In beiden Fällen folgt die Abschätzung $1 \leq m^*(I)/m(I, \mathbf{RUCK_APP}(I)) \leq 2$.

///

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

$m(I, \mathbf{RUCK_APP}(I)) = M/2 + 2$. Die optimale Lösung ist jedoch Ergebnis $x_1^* = 0$, $x_2^* = 1$ und $x_3^* = 1$ mit dem Profit $m^*(I) = M > M/2 + 2$. Damit ist

$$m^*(I) / m(I, \mathbf{RUCK_APP}(I)) = \frac{M}{M/2 + 2} = \frac{2M}{M + 4} = \frac{2 \cdot (M + 4) - 8}{M + 4} = 2 - \frac{8}{M + 4}.$$

Bei genügend großem M kommt dieser Wert beliebig dicht an 2 heran, so dass die Abschätzung in Satz 6.2-4 nicht verbessert werden kann.

Der folgende Satz zeigt, dass es unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ nicht für jedes Optimierungsproblem aus \mathbf{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 $V = \{v_1, \dots, v_n\}$, bestehend aus n Knoten und der Kantenmenge $E \subseteq V \times V$; die Funktion $w: E \rightarrow \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein nichtnegatives Gewicht

2. $\text{SOL}(G) = \{T \mid T = \langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \dots, (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle \text{ ist eine Tour durch } G\}$

3. für $T \in \text{SOL}(G)$, $T = \langle (v_{i_1}, v_{i_2}), (v_{i_2}, v_{i_3}), \dots, (v_{i_{n-1}}, v_{i_n}), (v_{i_n}, v_{i_1}) \rangle$, ist die Zielfunktion

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

4. $\text{goal} = \min$

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

Satz 6.2-5:

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 \mathbf{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 \mathbf{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 der Knotenmenge $V = \{v_1, \dots, v_n\}$.

Lösung: Entscheidung „ja“, falls gilt:

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

Es sei \mathbf{A} ein r -approximativen Algorithmus \mathbf{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 \mathbf{A} bereits eine optimale Lösung. Es wird ein Algorithmus $\tilde{\mathbf{A}}$ für UNGERICHTETER HAMILTONKREIS konstruiert, der folgendermaßen arbeitet:

Bei Eingabe eines ungerichteten Graphen $G = (V, E)$ mit $V = \{v_1, \dots, v_n\}$ und $n \geq 2$ konstruiert $\tilde{\mathbf{A}}$ einen vollständigen bewerteten ungerichteten Graphen $\tilde{G} = (V, \tilde{E})$ (die Knotenmenge wird beibehalten) mit $\tilde{E} = \{(v_i, v_j) \mid v_i \in V \text{ und } v_j \in V\}$ und der Kantenbewertung $w: \tilde{E} \rightarrow \mathbf{R}_{\geq 0}$, die durch

$w(v_i, v_j) = \begin{cases} 1 & \text{falls } (v_i, v_j) \in E \\ n \cdot r & \text{sonst} \end{cases}$ definiert ist. Diese Konstruktion erfolgt in polynomieller

Zeit, da höchsten $O(n^2)$ viele Kanten hinzuzufügen und $O(n^2)$ viele Kanten zu bewerten sind. Alle Bewertungen haben eine Länge der Ordnung $O(\log(n))$. Der Graph $\tilde{G} = (V, \tilde{E})$ wird in den Algorithmus \mathbf{A} eingegeben und das Ergebnis $m(\tilde{G}, \mathbf{A}(\tilde{G}))$ mit dem Wert $r \cdot n$ verglichen. $\tilde{\mathbf{A}}$ trifft die Entscheidung

$\tilde{\mathbf{A}}(G) = \begin{cases} \text{ja} & \text{für } m(\tilde{G}, \mathbf{A}(\tilde{G})) \leq r \cdot n \\ \text{nein} & \text{für } m(\tilde{G}, \mathbf{A}(\tilde{G})) > r \cdot n \end{cases}$.

Falls \mathbf{A} ein polynomiell zeitbeschränkter Algorithmus ist, dann auch $\tilde{\mathbf{A}}$.

Besitzt G einen Hamiltonkreis (mit Länge n), dann ist $m^*(\tilde{G}) = n$, und $\tilde{\mathbf{A}}(G) = \text{ja}$, da \mathbf{A} ein r -approximativen Algorithmus \mathbf{A} für das Handlungsreisenden-Minimierungsproblem ist und damit $m(\tilde{G}, \mathbf{A}(\tilde{G})) \leq r \cdot m^*(\tilde{G}) = r \cdot n$ ist.

Besitzt G keinen Hamiltonkreis, dann enthält jeder Hamiltonkreis in \tilde{G} mindestens eine Kante, die mit $r \cdot n$ bewertet ist (da \tilde{G} ein vollständiger Graph ist, enthält \tilde{G} einen Hamiltonkreis). Damit ergibt sich $m(\tilde{G}, \mathbf{A}(\tilde{G})) \geq m^*(\tilde{G}) \geq n - 1 + r \cdot n > r \cdot n$, d.h. $\tilde{\mathbf{A}}(G) = \text{nein}$.

In beiden Fällen trifft $\tilde{\mathbf{A}}$ die korrekte Entscheidung.

///

Das **metrische Handlungsreisenden-Minimierungsproblem** liegt jedoch in **APX**. Dieses stellt eine zusätzliche Bedingung an die Gewichtsfunktion einer Eingabeinstanz, nämlich die Gültigkeit der **Dreiecksungleichung**:

Für $v_i \in V$, $v_j \in V$ und $v_k \in V$ gilt $w(v_i, v_j) \leq w(v_i, v_k) + w(v_k, v_j)$.

Auch hierbei ist das zugehörige Entscheidungsproblem **NP**-vollständig. Es gibt (unabhängig von der Annahme $\mathbf{P} \neq \mathbf{NP}$) im Gegensatz zum (allgemeinen) Handlungsreisenden-Minimierungsproblem für dieses Problem einen 1,5-approximativen Algorithmus **CH**, der hier für den Fall eines vollständigen gewichteten ungerichteten Graphen mit Dreiecksungleichung informell beschrieben werden soll. Es ist nicht bekannt, ob es einen Approximationsalgorithmus mit einer kleineren relativen Approximationsgüte gibt oder ob aus der Existenz eines derartigen Algorithmus bereits $\mathbf{P} = \mathbf{NP}$ folgt.

Es sei $G = (V, E, w)$ ein vollständiger gewichteter ungerichteter Graph mit der Knotenmenge $V = \{v_1, \dots, v_n\}$ und der Kantenmenge $E = V \times V$. Für $v_i \in V$, $v_j \in V$ und $v_k \in V$ gelte die Dreiecksungleichung $w(v_i, v_j) \leq w(v_i, v_k) + w(v_k, v_j)$. Der Algorithmus, der als **Christofides' Algorithmus** bekannt ist, verläuft in mehreren Schritten und liefert eine Tour **CH**(G):

1. Es wird ein aufspannender Baum $T = (V, E_T)$ in G mit minimalem Gewicht $w(T) = \sum_{e \in E_T} w(e)$ bestimmt; ein aufspannender Baum enthält $|V| - 1$ vielen Kanten (entsprechende Algorithmen werden in der Literatur beschrieben²⁴). Der Aufwand für diesen Schritt ist von der Ordnung $O(n^2)$.

²⁴ Siehe: Hoffmann, U.: **Datenstrukturen und Algorithmen**, FINAL 15:2, 2005.

2. Es sei $C = \{v \mid v \in V \text{ und der Grad von } v \text{ in } T \text{ ist ungerade}\}$. Die Anzahl der Knoten in C ist gerade. Es wird eine Kantenmenge $H \subseteq E$ bestimmt, deren Kanten die Knoten aus C so verbinden, dass jeder Knoten den Grad 1 besitzt und dabei H eine Kantenmenge mit minimalem Gewicht $w(H) = \sum_{e \in H} w(e)$ ist. Die Menge H kann mit einem Aufwand der Ordnung $O(n^{2.5} \cdot (\log(n))^4)$ bestimmt werden²⁵.
3. Es wird $M = (V, E_T \cup H)$ gesetzt. Hierbei entsteht eventuell ein Multigraph, d.h. ein Graph, der zwischen zwei Knoten mehrere Kanten besitzt, die jeweils mit demselben Gewicht versehen sind. Jeder Knoten in V besitzt bezüglich M geraden Grad. Daher lässt sich in M eine geschlossene Kantenfolge K bestimmen, in der jede Kante aus M genau einmal vorkommt (die Existenz einer derartigen Eulerschen Linie sichern Sätze der Graphentheorie²⁶). O.B.d.A. ist der erste und letzte Knoten in dieser Kantenfolge der Knoten v_1 . Der Aufwand für diesen Schritt lässt sich durch eine Größe der Ordnung $O(n^2)$ begrenzen. Es sei K' die Folge der Anfangsknoten der Kanten in K gefolgt von v_1 .
4. Da alle Kanten aus $E_T \cup H$ in K jeweils genau einmal vorkommen und E_T als Endpunkte dieser Kanten die gesamte Knotenmenge V umfasst, gibt es eine Permutation π der Knotennummern $\{1, \dots, n\}$, so dass K' die Form $K' = (v_{\pi(1)}, \alpha_1, v_{\pi(2)}, \alpha_2, \dots, v_{\pi(n)}, \alpha_n, v_{\pi(1)})$ mit $v_{\pi(1)} = v_1$ besitzt (hierbei bezeichnen die Symbole α_i eventuell leere Folgen von Knotennummern). Die Permutation π wird dadurch bestimmt, dass man jeweils das erste Erscheinen eines weiteren Knotens betrachtet. Entfernt man aus K' die Teilfolgen $\alpha_1, \dots, \alpha_n$, so bildet $\mathbf{CH}(G) = ((v_{\pi(1)}, v_{\pi(2)}), (v_{\pi(2)}, v_{\pi(3)}), \dots, (v_{\pi(n)}, v_{\pi(1)}))$ eine Tour in G .

Satz 6.2-6:

Der Algorithmus **CH** ist ein polynomiell zeitbeschränkter 1,5-approximativer Algorithmus für das metrische Handlungsreisenden-Minimierungsproblem.

Beweis:

Die polynomielle Zeitschranke ergibt sich aus den Anmerkungen im Algorithmus.

²⁵ H wird ein perfektes Matching mit minimalem Gewicht genannt. Ein Algorithmus zur Bestimmung von H wird beschrieben in: Vaidya, P.: Geometry helps in matching, Proc. 20th ACM Symp. On Theory of Computing, 1988.

²⁶ Siehe: Harary, F.: **Graphentheorie**, Oldenbourg, 1974.

Die Anzahl der Knoten in C in Schritt 2. ist gerade: Bezeichnet $d_{G'}(v)$ den Grad des Knotens v in einem ungerichteten Graphen $G' = (V', E')$, so gilt $\sum_{v \in V'} d_{G'}(v) = 2 \cdot |E'|$; denn zählt man an jedem Knoten die von ihm ausgehenden Kanten (linke Seite der Gleichung), so wird jede Kante zweimal gezählt (rechte Seite der Gleichung), da jede Kante zwei Endpunkte hat. Damit ist $\sum_{\substack{v \in V \\ d_{G'}(v) \text{ ist ungerade}}} d_{G'}(v) = 2 \cdot |E'| - \sum_{\substack{v \in V \\ d_{G'}(v) \text{ ist gerade}}} d_{G'}(v)$ eine gerade Zahl und die Anzahl der (ungeraden) Summanden auf der linken Seite gerade.

Da $\mathbf{CH}(G) = ((v_{\pi(1)}, v_{\pi(2)}), (v_{\pi(2)}, v_{\pi(3)}), \dots, (v_{\pi(n)}, v_{\pi(1)}))$ nur Kanten aus $E_T \cup H$ enthält, gilt $m(G, \mathbf{CH}(G)) \leq w(T) + w(H)$.

Es gilt

1. $m^*(G) \geq 2 \cdot w(H)$; denn:

Es sei $(v_{i_1}, v_{i_2}, \dots, v_{i_k})$ mit $k = 2 \cdot |H|$ die Folge der Knoten aus T mit ungeradem Grad in der Reihenfolge, wie sie auf einer optimalen Tour durch G erscheinen. Es werden zwei Kantenmengen H_1 und H_2 wie folgt gebildet:

$$H_1 = \{(v_{i_1}, v_{i_2}), (v_{i_3}, v_{i_4}), \dots, (v_{i_{k-1}}, v_{i_k})\} \text{ und } H_2 = \{(v_{i_2}, v_{i_3}), (v_{i_4}, v_{i_5}), \dots, (v_{i_k}, v_{i_1})\}.$$

Mit $w(H_i)$ werde die Summe der Kantengewichte von H_i für $i = 1, 2$ bezeichnet. Nach Konstruktion von H ist dann für $i = 1, 2$: $w(H) \leq w(H_i)$. Da auf der optimalen Tour eventuell noch Knoten zwischen v_{i_j} und $v_{i_{j+1}}$ für $j = 1, \dots, k-1$ bzw. zwischen v_{i_k} und v_{i_1} liegen, ist wegen der Dreiecksungleichung $m^*(G) \geq w(H_1) + w(H_2) \geq 2 \cdot w(H)$.

2. $m(T) \leq m^*(G)$; denn:

Entfernt man eine Kante aus einer Tour durch den Graphen G , so erhält man einen aufspannenden Baum für G . Es sei e eine Kante in einer optimalen Tour. Da T ein minimaler aufspannender Baum ist, folgt $m(T) \leq m^*(G) - w(e) \leq m^*(G)$.

Damit ergibt sich $m(G, \mathbf{CH}(G)) \leq w(T) + w(H) \leq m^*(G) + m^*(G)/2 = 1,5 \cdot m^*(G)$.

///

Hat man für ein Optimierungsproblem aus **APX** einen r -approximativen Algorithmus gefunden, so stellt sich die Frage, ob dieser noch verbessert werden kann, d.h. ob es einen t -approximativen Algorithmus mit $1 \leq t < r$ gibt. Der folgende Satz 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 6.2-7:

Es sei Π' ein **NP**-vollständiges Entscheidungsproblem über Σ'^* und Π aus **NPO** ein Minimierungsproblem über Σ^* . Es gebe zwei in polynomieller Zeit berechenbare Funktionen $f: \Sigma'^* \rightarrow \Sigma^*$ und $c: \Sigma'^* \rightarrow \mathbf{N}$ und eine Konstante $\varepsilon > 0$ mit folgender Eigenschaft:

$$\text{Für jedes } x \in \Sigma'^* \text{ ist } m_{\Pi}^*(f(x)) = \begin{cases} c(x) & \text{für } x \in L_{\Pi'} \\ c(x) \cdot (1 + \varepsilon) & \text{für } x \notin L_{\Pi'} \end{cases}.$$

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 \mathbf{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 \mathbf{A} für Π eingegeben und der Näherungswert $m_{\Pi}(f(x), \mathbf{A}(f(x)))$ mit $c(x) \cdot (1 + \varepsilon)$ verglichen. \mathbf{A}' trifft die Entscheidung

$$\mathbf{A}'(x) = \begin{cases} \text{ja} & \text{für } m_{\Pi}(f(x), \mathbf{A}(f(x))) < c(x) \cdot (1 + \varepsilon) \\ \text{nein} & \text{für } m_{\Pi}(f(x), \mathbf{A}(f(x))) \geq c(x) \cdot (1 + \varepsilon) \end{cases}.$$

\mathbf{A}' ist ein polynomiell zeitbeschränkter Entscheidungsalgorithmus. Zu zeigen ist, dass die von \mathbf{A}' getroffene Entscheidung korrekt ist, d.h. dass $\mathbf{A}'(x) = \text{ja}$ genau dann gilt, wenn $x \in L_{\Pi'}$ ist.

Es sei $x \in L_{\Pi'}$. Da \mathbf{A} r -approximativ ist, gilt $\frac{m_{\Pi}(f(x), \mathbf{A}(f(x)))}{m_{\Pi}^*(f(x))} \leq r < 1 + \varepsilon$. Wegen $x \in L_{\Pi'}$ ist $m_{\Pi}^*(f(x)) = c(x)$. Also ist $m_{\Pi}(f(x), \mathbf{A}(f(x))) < m_{\Pi}^*(f(x)) \cdot (1 + \varepsilon) = c(x) \cdot (1 + \varepsilon)$, und $\mathbf{A}'(x) = \text{ja}$.

Es sei $x \notin L_{\Pi'}$. Dann folgt $m_{\Pi}^*(f(x)) = c(x) \cdot (1 + \varepsilon)$,
 $m_{\Pi}(f(x), \mathbf{A}(f(x))) \geq m_{\Pi}^*(f(x)) = c(x) \cdot (1 + \varepsilon)$ und $\mathbf{A}'(x) = \text{nein}$.

In beiden Fällen trifft \mathbf{A}' die korrekte Entscheidung.

Bemerkung: Der Beweis von Satz 6.2-7 zeigt, dass der Satz gültig bleibt, wenn die dort formulierte Voraussetzung über $m^*(f(x))$ ersetzt wird durch die Voraussetzung

$$\text{Für jedes } x \in \Sigma'^* \text{ ist } m_{\Pi}^*(f(x)) \begin{cases} = c(x) & \text{für } x \in L_{\Pi'} \\ \geq c(x) \cdot (1 + \varepsilon) & \text{für } x \notin L_{\Pi'} \end{cases}.$$

Ein entsprechender Satz gilt für Maximierungsprobleme:

Satz 6.2-8:

Es sei Π' ein **NP**-vollständiges Entscheidungsproblem über Σ'^* und Π aus **NPO** ein Maximierungsproblem über Σ^* . Es gebe zwei in polynomieller Zeit berechenbare Funktionen $f: \Sigma'^* \rightarrow \Sigma^*$ und $c: \Sigma'^* \rightarrow \mathbf{N}$ und eine Konstante $\varepsilon > 0$ mit folgender Eigenschaft:

$$\text{Für jedes } x \in \Sigma'^* \text{ ist } m_{\Pi}^*(f(x)) = \begin{cases} c(x) & \text{für } x \in L_{\Pi'} \\ c(x) \cdot (1 - \varepsilon) & \text{für } x \notin L_{\Pi'} \end{cases}.$$

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 beispielsweise zeigen, 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 6.2-9:

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 6.2-7 übernimmt das Binpacking-Minimierungsproblem die Rolle von Π ; für Π' wird das **NP**-vollständige Partitionenproblem (siehe Kapitel 5.4) genommen. Es wird definiert durch

Instanz: $I = [a_1, \dots, a_n]$,
 a_1, \dots, a_n sind natürliche Zahlen.

Lösung: Entscheidung „ja“ genau dann, wenn gilt:

Es gibt eine Aufteilung der Menge $\{a_1, \dots, a_n\}$ in zwei disjunkte Teile I_1 und

$$I_2 \text{ mit } \sum_{a_i \in I_1} a_i = \sum_{a_i \in I_2} a_i.$$

Es werden zwei Abbildungen f und c definiert, die den Bedingungen in Satz 6.2-7 genügen. Die Abbildung f ordnet jeder Instanz $I = [a_1, \dots, a_n]$ 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 $I = [a_1, \dots, a_n]$ des Partitionenproblems sei $B = \sum_{a_i \in I} a_i$. Es wird

$f(I) = [(2a_1)/B, \dots, (2a_n)/B]$ definiert, falls sich dadurch eine Instanz des Binpacking-Minimierungsproblems ergibt, d.h. falls $(2a_i)/B \leq 1$ für $i = 1, \dots, n$ 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 $I \in L_{\text{IT}}$, d.h. I ist eine „ja“-Instanz des Partitionenproblems bzw. es gibt eine Aufteilung der Menge $\{a_1, \dots, a_n\}$ 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

$a_i \leq B/2$ für $i = 1, \dots, n$, denn sonst könnte man I nicht in zwei gleich große Teile aufteilen. Daher ist $(2a_i)/B \leq 1$, $\sum_{a_i \in I_1} a_i = \sum_{a_i \in I_2} a_i = B/2$ und $\sum_{a_i \in I_1} (2a_i)/B = \sum_{a_i \in I_2} (2a_i)/B = 1$. Zur Packung der Instanz $f(I) = [(2a_1)/B, \dots, (2a_n)/B]$ des Binpacking-Minimierungsproblems sind genau 2 Behälter erforderlich, d.h. $m^*(f(I)) = 2 = c(I)$.

Ist $I \notin L_{\text{IT}}$ und $[(2a_1)/B, \dots, (2a_n)/B]$ keine Instanz des Binpacking-Minimierungsproblems, d.h. es gibt mindestens ein a_i mit $(2a_i)/B > 1$, dann ist $f(I) = [1, 1, 1]$ und $m^*(f(I)) = 3$.

Ist $I \notin L_{\text{IT}}$ und $[(2a_1)/B, \dots, (2a_n)/B]$ eine Instanz des Binpacking-Minimierungsproblems, dann ist $f(I) = [(2a_1)/B, \dots, (2a_n)/B]$ 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 6.2-7 zusammen mit der sich dort anschließenden Bemerkung folgt, dass es keinen polynomiell zeitbeschränkten r -approximativen Algorithmus für das Binpacking-

Minimierungsproblem mit $r < 1 + 1/2 = 1.5$ gibt, außer $\mathbf{P} = \mathbf{NP}$. Es gibt jedoch einen polynomiellen Approximationsalgorithmus \mathbf{SL} mit $m(I, \mathbf{SL}(I)) \leq 1.5 \cdot m^*(I)$ ²⁷.

///

Ein weiteres Beispiel für ein „interessantes“ Problem in \mathbf{APX} ist das

3-SAT-Maximierungsproblem

- Instanz:
1. $I = (K, V)$
 $K = \{F_1, F_2, \dots, F_m\}$ ist eine Menge von Klauseln, die aus Booleschen Variablen aus der Menge $V = \{x_1, \dots, x_n\}$ gebildet werden und jeweils die Form $F_i = (y_{i_1} \vee y_{i_2} \vee y_{i_3})$ für $i = 1, \dots, m$ 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. $\text{SOL}(I) =$ Belegung der Variablen in V mit Wahrheitswerten TRUE oder FALSE , d.h. $\text{SOL}(I)$ ist eine Abbildung $f : V \rightarrow \{\text{TRUE}, \text{FALSE}\}$
 3. Für $f \in \text{SOL}(I)$ ist $m(I, f) =$ Anzahl der Klauseln in K , die durch f erfüllt werden
 4. $goal = \max.$

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 \mathbf{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 \mathbf{APX} .

²⁷ Simchi-Levi, D.: New worst-case results for the bin-packing problem, Naval Research Logistics 41, 1994.

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 (ein etwas schwächeres Ergebnis wird in Satz 7.2-7 gezeigt).

Der $8/7$ -approximative polynomiell zeitbeschränkte Algorithmus soll hier informell beschrieben werden, weil er eine spezielle Technik, die Derandomisierung eines randomisierten Algorithmus (hier mit bedingten Erwartungswerten) einsetzt.

Eine Belegung der Variablen x_1, \dots, x_n mit Wahrheitswerten TRUE bzw. FALSE werde zufällig erzeugt, wobei $P(x_i = \text{TRUE}) = P(x_i = \text{FALSE}) = 1/2$ für $i = 1, \dots, n$ gelte. Diese Belegung induziert eine zufällige Belegung der drei verschiedenen Literale in der Klausel F_j für $j = 1, \dots, m$. 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 $P(F_j \text{ hat den Wahrheitswert TRUE}) = 7/8$.

Die Zufallsvariable X_j für $j = 1, \dots, m$ 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 $X = X_1 + \dots + X_m$.

Dann ist $\mathbf{E}[X_j] = P(X_j = \text{TRUE}) = 7/8$ und $\mathbf{E}[X] = 7/8 \cdot m$. Mit $m^*(I) \leq m$ ergibt sich für die relative Approximationsgüte einer zufälligen Belegung der Variablen mit Wahrheitswerten ein mittlerer Wert $\leq 8/7$.

Aus dieser Überlegung heraus lässt sich ein Algorithmus entwerfen, der nacheinander alle Variablen x_1, \dots, x_n in dieser aufsteigenden Reihenfolge mit Wahrheitswerten belegt.

Es gilt

$$\begin{aligned} \mathbf{E}[X] &= P(x_1 = \text{TRUE}) \cdot \mathbf{E}[X \mid x_1 = \text{TRUE}] + P(x_1 = \text{FALSE}) \cdot \mathbf{E}[X \mid x_1 = \text{FALSE}] \\ &= 1/2 \cdot (\mathbf{E}[X \mid x_1 = \text{TRUE}] + \mathbf{E}[X \mid x_1 = \text{FALSE}]) \\ &= 1/2 \cdot \left(\sum_{j=1}^m \mathbf{E}[X_j \mid x_1 = \text{TRUE}] + \sum_{j=1}^m \mathbf{E}[X_j \mid x_1 = \text{FALSE}] \right) . \end{aligned}$$

Die Werte $\mathbf{E}[X_j \mid x_1 = \text{TRUE}]$ und $\mathbf{E}[X_j \mid x_1 = \text{FALSE}]$ lassen sich folgendermaßen ermitteln:

$$\mathbf{E}[X_j \mid x_1 = \text{TRUE}] = \begin{cases} 1 & \text{falls das Literal } x_1 \text{ in } F_j \text{ vorkommt, also die Klausel erfüllt} \\ 3/4 & \text{falls das Literal } \neg x_1 \text{ in } F_j \text{ vorkommt} \\ 7/8 & \text{falls weder das Literal } x_1 \text{ noch das Literal } \neg x_1 \text{ in } F_j \text{ vorkommt,} \end{cases}$$

$$\mathbf{E}[X_j \mid x_1 = \text{FALSE}] = \begin{cases} 1 & \text{falls das Literal } \neg x_1 \text{ in } F_j \text{ vorkommt, also die Klausel erfüllt} \\ 3/4 & \text{falls das Literal } x_1 \text{ in } F_j \text{ vorkommt} \\ 7/8 & \text{falls weder das Literal } x_1 \text{ noch das Literal } \neg x_1 \text{ in } F_j \text{ vorkommt.} \end{cases}$$

Es wird x_1 genau dann mit TRUE belegt, wenn

$$\mathbf{E}[X \mid x_1 = \text{TRUE}] = \sum_{j=1}^m \mathbf{E}[X_j \mid x_1 = \text{TRUE}] \geq \sum_{j=1}^m \mathbf{E}[X_j \mid x_1 = \text{FALSE}] = \mathbf{E}[X \mid x_1 = \text{FALSE}] \text{ ist, andernfalls mit FALSE. Die so für } x_1 \text{ ermittelte Belegung sei } w_1 \in \{\text{TRUE}, \text{FALSE}\}. \text{ Dann ist}$$

$$7/8 \cdot m = \mathbf{E}[X] = 1/2 \cdot (\mathbf{E}[X \mid x_1 = \text{TRUE}] + \mathbf{E}[X \mid x_1 = \text{FALSE}]) \leq \mathbf{E}[X \mid x_1 = w_1].$$

Den Variablen x_1, \dots, x_{i-1} seien bereits Wahrheitswerte zugewiesen. Dann ist

$$\mathbf{E}[X_j \mid x_1, \dots, x_{i-1}, x_i = \text{TRUE}] = \begin{cases} 1 & \text{falls die Belegung von } x_1, \dots, x_i \text{ die Klausel } F_j \text{ erfüllt} \\ & \text{andernfalls:} \\ 7/8 & \left\{ \begin{array}{l} \text{falls kein Literal in } F_j \text{ bisher mit einem Wahrheitswert} \\ \text{belegt wurde} \end{array} \right. \\ 3/4 & \left\{ \begin{array}{l} \text{falls genau ein Literal in } F_j \text{ den Wahrheitswert FALSE hat} \\ \text{und den anderen beiden Literalen noch kein Wahrheitswert} \\ \text{zugewiesen wurde} \end{array} \right. \\ 1/2 & \left\{ \begin{array}{l} \text{falls genau zwei Literale in } F_j \text{ den Wahrheitswert FALSE} \\ \text{haben und dem dritten Literal noch kein Wahrheitswert} \\ \text{zugewiesen wurde} \end{array} \right. \\ 0 & \left\{ \begin{array}{l} \text{falls alle Literale in } F_j \text{ den Wahrheitswert} \\ \text{FALSE haben} \end{array} \right. \end{cases}$$

Entsprechende Werte gelten für $\mathbf{E}[X_j \mid x_1, \dots, x_{i-1}, x_i = \text{FALSE}]$. Damit lassen sich

$$\mathbf{E}[X \mid x_1, \dots, x_{i-1}, x_i = \text{TRUE}] = \sum_{j=1}^m \mathbf{E}[X_j \mid x_1, \dots, x_{i-1}, x_i = \text{TRUE}],$$

$$\mathbf{E}[X \mid x_1, \dots, x_{i-1}, x_i = \text{FALSE}] = \sum_{j=1}^m \mathbf{E}[X_j \mid x_1, \dots, x_{i-1}, x_i = \text{FALSE}].$$

Es wird x_i genau dann mit TRUE belegt, wenn

$\mathbf{E}[X \mid x_1, \dots, x_{i-1}, x_i = \text{TRUE}] \geq \mathbf{E}[X \mid x_1, \dots, x_{i-1}, x_i = \text{FALSE}]$ ist, andernfalls mit FALSE. Die so für x_i ermittelte Belegung sei $w_i \in \{\text{TRUE}, \text{FALSE}\}$. Diese erfüllt wieder

$$7/8 \cdot m = \mathbf{E}[X] = 1/2 \cdot (\mathbf{E}[X \mid x_1, \dots, x_{i-1}, x_i = \text{TRUE}] + \mathbf{E}[X \mid x_1, \dots, x_{i-1}, x_i = \text{FALSE}])$$

$$\leq \mathbf{E}[X \mid x_1, \dots, x_{i-1}, x_i = w_i]$$

Der folgende Algorithmus **A** (in Pseudocode) ermittelt eine Belegung der Variablen mit

$$\frac{m^*(I)}{m(I, \mathbf{A}(I))} \leq \frac{m}{(7/8 \cdot m)} = \frac{8}{7} \text{ und hat eine polynomielle Laufzeit der Ordnung } O(n \cdot m).$$

Approximationsalgorithmus für das 3-SAT-Maximierungsproblem

Eingabe: $I = (K, V)$

$K = \{F_1, F_2, \dots, F_m\}$ ist eine Menge von Klauseln, die aus Booleschen Variablen aus der Menge $V = \{x_1, \dots, x_n\}$ gebildet werden und jeweils die Form $F_i = (y_{i_1} \vee y_{i_2} \vee y_{i_3})$ für $i = 1, \dots, m$ besitzen. Dabei sind in jeder Klausel in den Literalen genau 3 verschiedene Variablen.

Verfahren: VAR W_0 : REAL;
 W_1 : REAL;
 i : INTEGER;

```
BEGIN
  FOR i := 1 TO n DO
    BEGIN
       $W_0 := E[X \mid x_1, \dots, x_{i-1}, x_i = \text{FALSE}]$ ;
       $W_1 := E[X \mid x_1, \dots, x_{i-1}, x_i = \text{TRUE}]$ ;
      IF  $W_0 \leq W_1$  THEN  $x_i := \text{TRUE}$ 
        ELSE  $x_i := \text{FALSE}$ ;
    END;
  END;
```

Ausgabe: x_1, \dots, x_n .

Der folgende Satz stellt einen Zusammenhang zwischen den Klassen **ABS** und **APX** her:

Satz 6.2-10:

Es sei Π ein Optimierungsproblem, für dessen Zielfunktion $m_\Pi(x, y) \geq 1$ mit $x \in \Sigma_\Pi^*$ und $y \in \text{SOL}_\Pi(x)$ ist. Dann gilt:
 Liegt Π in **ABS**, dann auch in **APX**.

Beweis:

Es sei Π in **ABS** und \mathbf{A} ein Approximationsalgorithmus mit $|m_{\Pi}^*(x) - m_{\Pi}(x, \mathbf{A}(x))| \leq k$ für ein $k \geq 0$ und jedes $x \in \Sigma_{\Pi}^*$ und jedes $y \in \text{SOL}_{\Pi}(x)$.

Ist Π ein Maximierungsproblem, dann folgt aus

$$m_{\Pi}^*(x) - m_{\Pi}(x, \mathbf{A}(x)) = |m_{\Pi}^*(x) - m_{\Pi}(x, \mathbf{A}(x))| \leq k :$$

$$R_{\mathbf{A}}(x) = \frac{m_{\Pi}^*(x)}{m_{\Pi}(x, \mathbf{A}(x))} \leq \frac{k}{m_{\Pi}(x, \mathbf{A}(x))} + 1 \leq k + 1.$$

Ist Π ein Minimierungsproblem, dann folgt aus

$$m_{\Pi}(x, \mathbf{A}(x)) - m_{\Pi}^*(x) = |m_{\Pi}^*(x) - m_{\Pi}(x, \mathbf{A}(x))| \leq k :$$

$$R_{\mathbf{A}}(x) = \frac{m(x, \mathbf{A}(x))}{m_{\Pi}^*(x)} \leq \frac{k}{m_{\Pi}^*(x)} + 1 \leq k + 1.$$

In beiden Fällen ist Π in **APX**.

///

Es sei \mathbf{A} ein Approximationsalgorithmus für das Minimierungsproblem Π . Dann ist die Forderung $m_{\Pi}(x, \mathbf{A}(x)) \leq r \cdot m_{\Pi}^*(x)$ mit Konstanten r für alle Instanzen $x \in \Sigma_{\Pi}^*$ eine eventuell zu starke Forderung, da wie beim Binpacking-Minimierungsproblem für $\mathbf{P} \neq \mathbf{NP}$ eine untere Grenze für r folgen kann. Die Idee, dass sich die relative Approximationsgüte $R_{\mathbf{A}}(x)$ eines Approximationsalgorithmus \mathbf{A} bei Eingabe einer Instanz $x \in \Sigma_{\Pi}^*$ bei wachsendem Wert einer optimalen Lösung $m_{\Pi}^*(x)$ eventuell verbessert, drückt sich in der folgenden Definition aus:

Es sei Π ein Optimierungsproblem aus **NPO**. Ein Approximationsalgorithmus \mathbf{A} heißt **asymptotisch r -approximativ** für Π , wenn es eine rationale Zahl $r > 1$ und eine Konstante k gibt, so dass gilt: für jede Eingabeinstanz x von Π liefert \mathbf{A} bei Eingabe von x eine zulässige Lösung, deren relative Approximationsgüte $R_{\mathbf{A}}(x)$ die Bedingung $R_{\mathbf{A}}(x) \leq r + k/m_{\Pi}^*(x)$ erfüllt.

Zur Erinnerung: die relative Approximationsgüte wurde definiert durch

$$R_{\mathbf{A}}(x) = \frac{m_{\Pi}^*(x)}{m_{\Pi}(x, \mathbf{A}(x))} \text{ bei einem Maximierungsproblem}$$

bzw.

$$R_{\mathbf{A}}(x) = \frac{m_{\Pi}(x, \mathbf{A}(x))}{m_{\Pi}^*(x)} \text{ bei einem Minimierungsproblem.}$$

Die Bedingung $R_{\mathbf{A}}(x) \leq r + k/m_{\Pi}^*(x)$ besagt also bei einem Maximierungsproblem:

$$m_{\Pi}(x, \mathbf{A}(x)) \geq \frac{1}{r} \cdot m_{\Pi}^*(x) - k' \quad \text{mit} \quad k' = \frac{k}{r \cdot (r + k/m_{\Pi}^*(x))} \leq k/r,$$

$$\text{daher } m_{\Pi}(x, \mathbf{A}(x)) \geq \frac{1}{r} \cdot m_{\Pi}^*(x) - k/r.$$

Bei einem Minimierungsproblem besagt sie $m_{\Pi}(x, \mathbf{A}(x)) \leq r \cdot m_{\Pi}^*(x) + k$.

Die minimal erreichbare relative Approximationsgüte ist beim Binpacking-Minimierungsproblem laut Satz 6.2-9 bei $\mathbf{P} \neq \mathbf{NP}$ gleich 1,5. 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: $I = [a_1, \dots, a_n]$,

a_1, \dots, a_n („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, \dots, n$

Verfahren: $B_1 := a_1$;

Sind a_1, \dots, a_i bereits in die Behälter B_1, \dots, B_j gelegt worden, ohne dass einer dieser Behälter überläuft, so lege a_{i+1} in den Behälter unter B_1, \dots, 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, \dots, 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 $[B_1, \dots, B_k]$.

Satz 6.2-11:

Ist $I = [a_1, \dots, a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $m(I, \mathbf{FF}(I)) \leq 1,7 \cdot m^*(I) + 2$.

Der Beweis verwendet eine „Gewichtung“ der Objekte der Eingabeinstanz. Wegen der Länge des Beweises muss auf die Literatur verwiesen werden²⁸.

²⁸ Johnson, D.S.: Fast Algorithms for Bin Packing, J. of Comp. and Syst. Sci., 8, S. 272 – 314, 1974.

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 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: $I = [a_1, \dots, a_n]$,
 a_1, \dots, a_n („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, \dots, n$

Verfahren: Sortiere die Objekte a_1, \dots, a_n nach absteigender Größe. Die Objekte erhalten im folgenden wieder die Benennung a_1, \dots, a_n , d.h. es gilt $a_1 \geq \dots \geq a_n$.

$B_1 := a_1$;

Sind a_1, \dots, a_i bereits in die Behälter B_1, \dots, B_j gelegt worden, ohne dass einer dieser Behälter überläuft, so lege a_{i+1} in den Behälter unter B_1, \dots, 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, \dots, 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 $[B_1, \dots, B_k]$.

Satz 6.2-12:

Ist $I = [a_1, \dots, a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $m(I, \mathbf{FFD}(I)) \leq 1,5 \cdot m^*(I) + 1$.

Beweis:

Die Objekte der Eingabeinstanz $I = [a_1, \dots, a_n]$ seien nach absteigender Größe sortiert, d.h. $a_1 \geq \dots \geq a_n$. Sie werden in vier Größenordnungen eingeteilt. Dazu wird

$$A = \{a_i \mid a_i \in I \text{ und } a_i > 2/3\},$$

$$B = \{a_i \mid a_i \in I \text{ und } 2/3 \geq a_i > 1/2\},$$

$$C = \{a_i \mid a_i \in I \text{ und } 1/2 \geq a_i > 1/3\} \text{ und}$$

$$D = \{a_i \mid a_i \in I \text{ und } 1/3 \geq a_i > 0\} \text{ gesetzt.}$$

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

$$\sum_{i=1}^n a_i = \sum_{j=1}^{k-1} u_j + u_k \geq \sum_{j=1}^{k-1} 2/3 = 2/3 \cdot (k-1) \text{ und folglich}$$

$$m(I, \mathbf{FFD}(I)) = k \leq 3/2 \cdot \sum_{i=1}^n a_i + 1 \leq 3/2 \cdot m^*(I) + 1.$$

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; $t_C \leq 1$.

$$m^*(\tilde{I}) = t_A + t_{BC} + t_{CC} + t_B + t_C.$$

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 . Insgesamt ergibt sich

$$m(\tilde{I}, \mathbf{FFD}(\tilde{I})) \leq t_A + t_{BC} + t_{CC} + t_B + t_C = m^*(\tilde{I}) \text{ und daher } m(\tilde{I}, \mathbf{FFD}(\tilde{I})) = m^*(\tilde{I}).$$

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.

Eine genauere Analyse des FirstfitDecreasing-Algorithmus zeigt, dass die in Satz 6.2-12 angegebene Schranke 1,5 verbessert werden kann. Es lässt sich zeigen, dass für jede Instanz $I = [a_1, \dots, a_n]$ des Binpacking-Minimierungsproblems die Abschätzung

$m(I, \mathbf{FFD}(I)) \leq 11/9 \cdot m^*(I) + 4$ gilt²⁹. 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: $I = [a_1, \dots, a_n]$,
 a_1, \dots, a_n („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, \dots, n$

Verfahren: Sortiere die Objekte a_1, \dots, a_n nach absteigender Größe. Die Objekte erhalten im folgenden wieder die Benennung a_1, \dots, a_n , d.h. es gilt $a_1 \geq \dots \geq a_n$.

$B_1 := a_1$;

²⁹ Siehe vorherige Fußnote.

Sind a_1, \dots, a_i bereits in die Behälter B_1, \dots, B_j gelegt worden, ohne dass einer dieser Behälter überläuft, so lege a_{i+1} in den Behälter unter B_1, \dots, B_j , der den kleinsten freien Platz aufweist. Falls a_{i+1} in keinen der Behälter B_1, \dots, B_j passt, lege a_{i+1} in einen neuen (bisher leeren) Behälter B_{j+1} .

Ausgabe: $\mathbf{BFD}(I) =$ die benötigten nichtleeren Behälter $[B_1, \dots, B_k]$.

Satz 6.2-13:

Ist $I = [a_1, \dots, a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $m(I, \mathbf{BFD}(I)) \leq \frac{11}{9} \cdot m^*(I) + 4$.

Auch der BestfitDecreasing-Algorithmus nur asymptotisch $11/9$ -approximativ. Das obige Beispiel zeigt, dass auch für diesen Algorithmus die $11/9$ -Grenze nicht verbessert werden kann.

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. 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. $I = [a_1, \dots, a_n]$
 a_1, \dots, a_n („Objekte“) sind natürliche Zahlen mit $a_i > 0$ für $i = 1, \dots, n$
 2. $\text{SOL}(I) = \{[Y_1, Y_2] \mid [Y_1, Y_2] \text{ ist eine Partition (disjunkte Zerlegung) von } I\}$
 3. Für $[Y_1, Y_2] \in \text{SOL}(I)$ 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: $I = [a_1, \dots, a_n]$, rationale Zahl $r > 1$,
 a_1, \dots, a_n („Objekte“) sind natürliche Zahlen mit $a_i > 0$ für $i = 1, \dots, n$

Verfahren:

```

VAR  $Y_1$  : SET OF INTEGER;
     $Y_2$  : SET OF INTEGER;
    k : REAL;
    j : INTEGER;

```



```

BEGIN
  IF  $r \geq 2$ 
  THEN BEGIN
     $Y_1 := \{a_1, \dots, a_n\};$ 
     $Y_2 := \{ \};$ 
  END
  ELSE BEGIN
    Sortiere die Objekte nach absteigender Größe; die dabei
    entstehende Folge sei  $[x_1, \dots, x_n];$ 
     $k := \lceil (2-r)/(r-1) \rceil;$ 
    { Phase 1: }
    finde eine optimale Partition  $[Y_1, Y_2]$  für  $[x_1, \dots, x_k];$ 
    { Phase 2: }
    FOR  $j := k+1$  TO  $n$  DO
      IF  $\sum_{x_i \in Y_1} x_i \leq \sum_{x_i \in Y_2} x_i$ 
      THEN  $Y_1 := Y_1 \cup \{x_j\}$ 
      ELSE  $Y_2 := Y_2 \cup \{x_j\}$ 
    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 $I = [a_1, \dots, a_n]$ 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 $I = [a_1, \dots, a_n]$ und r in den angegebenen Algorithmus eingegeben. Zur Vereinfachung werde angenommen, dass die Eingabewerte bereits nach absteigender Größe sortiert sind, d.h. $[x_1, \dots, x_n] = [a_1, \dots, a_n]$. 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 :

$$w(I) = \sum_{a_i \in I} a_i, \quad w(Y_2) \text{ die Summe aller Objekte in } Y_2: w(Y_2) = \sum_{a_i \in Y_2} a_i.$$

1. Fall: $r \geq 2$

Dann ist $m^*(I) \geq w(I)/2 = m(I, \mathbf{A}(I))/2$, 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.

Fall 2a: a_h wurde in Phase 1 des Algorithmus in Y_1 aufgenommen.

Es sei $[Z_1, Z_2]$ eine optimale Partition für $[a_1, \dots, a_k]$; o.B.d.A. sei $\sum_{a_i \in Z_1} a_i \geq \sum_{a_j \in Z_2} a_j$. Da in Phase 2 alle Objekte zu dem Teil mit kleinerer Summe, in diesem Fall zu Y_2 , kommen, ist $Y_1 = Z_1$. Weiterhin ist $\sum_{a_i \in Y_1} a_i = \sum_{a_i \in Z_1} a_i = m^*([a_1, \dots, a_k]) \leq m^*(I) \leq m(I, \mathbf{A}(I)) = \sum_{a_i \in Y_1} a_i$. Damit ist $m(I, \mathbf{A}(I)) = m^*(I) < r \cdot m^*(I)$.

Fall 2b: a_h wurde in Phase 2 des Algorithmus in Y_1 aufgenommen.

Es folgt nacheinander:

$$m(I, \mathbf{A}(I)) - a_h \leq w(Y_2),$$

$$2 \cdot m(I, \mathbf{A}(I)) - a_h \leq w(Y_2) + m(I, \mathbf{A}(I)) = w(I) \text{ und}$$

$$m(I, \mathbf{A}(I)) - w(I)/2 \leq a_h/2.$$

Außerdem gilt wegen der Sortierung der Objekte $a_h \leq a_j$ für $j = 1, \dots, k$. Diese Ungleichungen werden auf a_h aufsummiert zu dem Ergebnis:

$$(k+1) \cdot a_h = a_h + k \cdot a_h \leq \sum_{j=1}^k a_j + a_h \leq w(I), \text{ also } a_h \leq w(I)/(k+1).$$

Insgesamt ergibt sich mit $m^*(I) \geq w(I)/2$:

$$\frac{m(I, \mathbf{A}(I))}{m^*(I)} \leq \frac{m(I, \mathbf{A}(I))}{w(I)/2} \leq \frac{a_h/2 + w(I)/2}{w(I)/2} = 1 + \frac{a_h}{w(I)} \leq 1 + \frac{1}{k+1} \leq r;$$

die letzte Ungleichung folgt aus der Wahl von k , nämlich $k = \lceil (2-r)/(r-1) \rceil$:

$$k \geq (2-r)/(r-1), \text{ d.h. } k+1 \geq (2-r+r-1)/(r-1) = 1/(r-1).$$

Die Anzahl an Zeichen, um die Eingabeinstanz darzustellen, die Problemgröße $size(I)$, ist von der Ordnung $O(n \cdot \log(A))$ mit $A = \max\{a_i \mid i = 1, \dots, n\}$.

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 $k = k(r)$ 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 6.2-9 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 \mathbf{PTAS} Optimierungsprobleme, die ein polynomiell zeitbeschränktes Approximationsschema \mathbf{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 \mathbf{NPO} , die einen derartigen Algorithmus zulassen, bilden die Klasse \mathbf{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 \mathbf{FPTAS} zur Folge hätte.

Der folgende Satz zeigt, dass das ganzzahlige 0/1-Rucksackproblem als Maximierungsproblem in \mathbf{FPTAS} liegt. Daher ist die Klasse \mathbf{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 $r > 1$
 $A = \{a_1, \dots, a_n\}$ ist eine Menge von n Objekten und $M \in \mathbf{N}_{>0}$ die „Rucksackkapazität“. Jedes Objekt a_i , $i = 1, \dots, n$, 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 $p_{max} = \max\{p_1, \dots, p_n\} = p_{i_{max}}$.

Verfahren: VAR k : REAL;

$$k := (r-1) \cdot p_{\max} / n ;$$

FOR $i := 1$ TO n DO

$$p'_i := \left\lfloor \frac{p_i}{k} \right\rfloor ;$$

berechne eine optimale Rucksackfüllung für die Eingabeinstanz $I' = (A', M)$ mit $A' = \{a'_1, \dots, a'_n\}$ und $a'_i = (w_i, p'_i)$ für $i = 1, \dots, n$; die zugehörige Entscheidungsfolge sei (x_1^*, \dots, x_n^*) ;

IF $\sum_{i=1}^n x_i^* \cdot p_i \geq p_{\max}$ THEN Ausgabe (x_1^*, \dots, x_n^*) und $\sum_{i=1}^n x_i^* \cdot p_i$
 ELSE Ausgabe $(\underbrace{0, \dots, 0, 1, 0, \dots, 0}_{1 \text{ an Position } i_{\max}})$ 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

$$R_A(I) = \frac{m^*(I)}{m(I, \mathbf{A}(I))} \leq r \text{ gilt.}$$

Die Folge (x_1^*, \dots, x_n^*) ist eine optimale Entscheidungsfolge für $I' = (A', M)$ mit

$$A' = \{a'_1, \dots, a'_n\} \text{ und } a'_i = (w_i, p'_i) \text{ für } i = 1, \dots, n, \text{ d.h. } m^*(I') = \sum_{i=1}^n x_i^* \cdot \left\lfloor \frac{p_i}{k} \right\rfloor \text{ und}$$

$$\sum_{i=1}^n x_i^* \cdot w_i \leq M .$$

Es sei (x_1^*, \dots, x_n^*) eine optimale Entscheidungsfolge für die Ausgangsinstanz $I = (A, M)$, d.h.

$$m^*(I) = \sum_{i=1}^n x_i^* \cdot p_i \text{ und } \sum_{i=1}^n x_i^* \cdot w_i \leq M . \text{ Die letzte Ungleichung zeigt, dass } (x_1^*, \dots, x_n^*) \text{ eine zu-}$$

lässige Lösung auch für die Instanz $I' = (A', M)$ ist. Daher ist

$$m^*(I') = \sum_{i=1}^n x_i^* \cdot \left\lfloor \frac{p_i}{k} \right\rfloor \geq \sum_{i=1}^n x_i^* \cdot \left[\frac{p_i}{k} \right] .$$

Es ist

$$\begin{aligned}
m(I, \mathbf{A}(I)) &= \max \left\{ \sum_{i=1}^n x_i^* \cdot p_i, p_{\max} \right\} \\
&\geq \sum_{i=1}^n x_i^* \cdot p_i \geq \sum_{i=1}^n x_i^* \cdot k \cdot \left\lfloor \frac{p_i}{k} \right\rfloor = k \cdot \sum_{i=1}^n x_i^* \cdot \left\lfloor \frac{p_i}{k} \right\rfloor = k \cdot m^*(I') \\
&\geq k \cdot \sum_{i=1}^n x_i^* \cdot \left\lfloor \frac{p_i}{k} \right\rfloor \geq k \cdot \sum_{i=1}^n x_i^* \cdot \left(\frac{p_i}{k} - 1 \right) \\
&= \sum_{i=1}^n x_i^* \cdot (p_i - k) = m^*(I) - k \cdot \sum_{i=1}^n x_i^* \geq m^*(I) - k \cdot n,
\end{aligned}$$

also $m^*(I) \leq m(I, \mathbf{A}(I)) + k \cdot n$. Daraus folgt

$$R_{\mathbf{A}}(I) = \frac{m^*(I)}{m(I, \mathbf{A}(I))} \leq 1 + \frac{k \cdot n}{m(I, \mathbf{A}(I))} \leq 1 + \frac{k \cdot n}{p_{\max}} = 1 + \frac{((r-1) \cdot p_{\max}/n) \cdot n}{p_{\max}} = r.$$

Die Ermittlung einer optimalen Rucksackfüllung für die Eingabeinstanz $I' = (A', M)$ mit $A' = \{a'_1, \dots, a'_n\}$ und $a'_i = (w_i, p'_i)$ für $i = 1, \dots, n$ kann beispielsweise mit Hilfe eines Algorithmus erfolgen, der nach dem Prinzip der Dynamischen Programmierung arbeitet³⁰:

Bestimmung einer optimalen Lösung für das ganzzahlige 0/1-Rucksack-Maximierungsproblem

Eingabe: $I' = (A', M)$

$A' = \{a'_1, \dots, a'_n\}$ ist eine Menge von n Objekten und $M \in \mathbf{N}_{>0}$ die „Rucksackkapazität“. Jedes Objekt a_i , $i = 1, \dots, n$, hat die Form $a'_i = (w_i, p'_i)$; hierbei bezeichnet $w_i \in \mathbf{N}_{>0}$ das Gewicht und $p'_i \in \mathbf{N}$ den Wert (Profit) des Objekts a_i .

Verfahren: Für $j = 1, \dots, n$ und $y \leq M$ bezeichne $f_j(y)$ den Wert der Zielfunktion einer optimalen Lösung für die Eingabeinstanz (A'_j, y) mit $A'_j = \{a'_1, \dots, a'_j\}$; die zugehörige Entscheidungsfolge sei (x_1^*, \dots, x_j^*) . Gesucht werden $(x_1^*, \dots, x_n^*) \in \text{SOL}(I')$ der Wert $m^*(I') = f_n(M)$. Es gelten folgende Rekursionsgleichungen:

$$\begin{aligned}
f_0(y) &= 0 \text{ für } 0 \leq y \leq M \text{ und } f_0(y) = -\infty \text{ für } y < 0, \\
f_j(y) &= \max \{ f_{j-1}(y), f_{j-1}(y - w_j) + p'_j \} \text{ für } j = 1, \dots, n, 0 \leq y \leq M.
\end{aligned}$$

Die Sprungstellen dieser Treppenfunktionen bestimmen eindeutig den Funktionsverlauf und werden mittels Dynamischer Programmierung berechnet. Es sei S_j für $j = 0, \dots, n$ die Menge der Sprungstellen von f_j .

³⁰ Siehe: Hoffmann, U.: **Datenstrukturen und Algorithmen**, FINAL 15:2, 2005.

Zur Erzeugung der Funktionen f_0, \dots, f_n bzw. der Mengen ihrer Sprungstellen S_0, \dots, S_n und der Ermittlung einer optimalen Lösung werden im folgenden folgende Begriffe verwendet:

Ein Zahlenpaar (W, P) *dominiert* ein Zahlenpaar (W', P') , wenn $W \leq W'$ und $P \geq P'$ 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 $j = 1, \dots, n$: }

BEGIN

$S_0 := \{(0, 0)\};$

FOR $j := 1$ TO n DO

BEGIN

$\hat{S}_j := \{(W + w_j, P + p_j) \mid (W, P) \in S_{j-1}\};$

$S_j := S_{j-1} \otimes \hat{S}_j;$

END;

END;

{ Die Paare in S_j seien aufsteigend nach der ersten (und damit auch nach der zweiten) Komponente geordnet, d.h.

$S_j = \{(W_{0,j}, P_{0,j}), \dots, (W_{i_j,j}, P_{i_j,j})\}$ mit $W_{k,j} \leq W_{k+1,j}$ für $0 \leq k < i_j$.

Für $y \geq 0$ gelte $W_{k,j} \leq y < W_{k+1,j}$.

Dann ist $f_j(y) = P_{k,j}$. }

{ Berechnung einer optimierende Auswahl

x_1^*, \dots, x_n^* : }

Es sei (W, P) dasjenige Paar in S_n mit $f_n(M) = P$.

```

FOR  $j := n$  DOWNTO 1 DO
  BEGIN
    IF  $(W, P) \in S_{j-1}$ 
    THEN  $x_j^* := 0$ 
    ELSE BEGIN
       $x_j^* := 1$  ;
       $W := W - w_j$  ;
       $P := P - p_j'$  ;
    END ;
  END ;

```

Ausgabe: (x_1^*, \dots, x_n^*) und Wert $\sum_{i=1}^n x_i^* \cdot p_i'$ der Zielfunktion.

Die Laufzeit des Verfahrens bei Eingabe von $I' = (A', M)$ mit $A' = \{a'_1, \dots, a'_n\}$ und $a'_i = (w_i, p_i')$ für $i = 1, \dots, n$ ist von der Ordnung $O\left(n \cdot \sum_{i=1}^n p_i'\right)$. Es ist

$$n \cdot \sum_{i=1}^n p_i' = n \cdot \sum_{i=1}^n \left\lfloor \frac{p_i}{k} \right\rfloor \leq n \cdot \sum_{i=1}^n \frac{p_i}{k} \leq n^2 \cdot \frac{p_{\max}}{(r-1) \cdot p_{\max}/n} = n^3 \cdot \frac{1}{r-1},$$

also ein Ausdruck 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 $A = \max\{\{p_{\max}\} \cup \{w_1, \dots, w_n\}\}$. 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 $y \in \text{SOL}_{\Pi}(x)$ die Abschätzung $m_{\Pi}(x, y) \leq p(|x|)$ gilt.

Satz 6.3-4:

Ist $\mathbf{P} \neq \mathbf{NP}$, so liegt kein polynomiell beschränktes Optimierungsproblem aus \mathbf{NPO} , dessen Instanzen und zulässige Lösungen aus natürlichen Zahlen bestehen und dessen zugehöriges Entscheidungsproblem \mathbf{NP} -vollständig ist, in \mathbf{FPTAS} .

Beweis:

Es sei Π ein polynomiell beschränktes Maximierungsproblem aus \mathbf{NPO} , dessen Instanzen aus natürlichen Zahlen bestehen und dessen zugehöriges Entscheidungsproblem \mathbf{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 $y \in \text{SOL}_{\Pi}(x)$ 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 $R_{\mathbf{A}}(x) = \frac{m_{\Pi}^*(x)}{m_{\Pi}(x, \mathbf{A}(x, r))} \leq r$ in einer Zeit ermittelt, die polynomiell in $|x|$ und $1/(r-1)$

ist. Das zugehörige Entscheidungsproblem zu Π sei Π_{ENT} zur Entscheidung der Sprache L_{ENT} . Nach Voraussetzung ist L_{ENT} \mathbf{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 \mathbf{A} eingegeben. Dann ist

$$R_{\mathbf{A}}(x) = \frac{m_{\Pi}^*(x)}{m_{\Pi}(x, \mathbf{A}(x, r))} \leq r = 1 + \frac{1}{p(|x|)} = \frac{p(|x|)+1}{p(|x|)}. \text{ Daraus folgt}$$

$m_{\Pi}(x, \mathbf{A}(x, r)) \geq m_{\Pi}^*(x) \cdot \frac{p(|x|)}{p(|x|)+1} = m_{\Pi}^*(x) - \frac{m_{\Pi}^*(x)}{p(|x|)+1} > m_{\Pi}^*(x) - 1$. Die letzte Ungleichung ergibt sich aus der Tatsache, dass $m_{\Pi}^*(x) \leq p(|x|) < p(|x|)+1$ ist. Da \mathbf{A} eine zulässige Lösung ermittelt und diese eine natürliche Zahl ist, folgt $m(x, \mathbf{A}(x, r)) = m_{\Pi}^*(x)$. Wegen $\frac{1}{r-1} = p(|x|)$ 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 \mathbf{PTAS} angeführt, die die Voraussetzungen aus Satz 6.3-4 erfüllen³¹. Damit ergibt sich der folgende Satz.

³¹ Das Problem der Bestimmung einer maximalen unabhängigen Menge in einem planaren Graphen $G = (V, E)$, d.h. die Bestimmung einer möglichst großen Menge $V' \subseteq V$, so dass für jede Kante $(u, v) \in E$ entweder $u \notin V'$ oder $v \notin V'$ gilt, gehört zu \mathbf{PTAS} und erfüllt die Voraussetzungen von

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 wurde der Begriff eines asymptotisch r -approximativen Algorithmus für ein Problem Π eingeführt. Dieser wird auf ein Approximationsschema erweitert:

Es sei Π ein Optimierungsproblem aus \mathbf{NPO} . Ein Algorithmus \mathbf{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 \mathbf{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 \mathbf{A} für jedes feste r polynomiell in der Größe $|x|$ der Eingabeinstanz.

Die Klasse aller Optimierungsprobleme in \mathbf{NPO} , die ein asymptotisches Approximationsschema besitzen, wird mit \mathbf{PTAS}^{∞} bezeichnet.

Satz 6.3-6:

Das Binpacking-Minimierungsproblem liegt in \mathbf{PTAS}^{∞} , 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 werden³².

Eine Eingabeinstanz des Binpacking-Minimierungsproblems sei in der in Kapitel 5.3 beschriebenen Form $I = [a_1, \dots, a_n, b]$ gegeben. Hierbei sind a_1, \dots, a_n („Objekte“) natürliche

Satz 6.3-4, liegt also nicht in \mathbf{FPTAS} . Details findet man in: Ausiello, G.; Crescenzi, P.; Gambosi, G.; Kann, V.; Marchetti-Spaccamela, A.; Protasi, M.: **Complexity and Approximation**, Springer, 1999.

³² Ausiello, G.; Crescenzi, P.; Gambosi, G.; Kann, V.; Marchetti-Spaccamela, A.; Protasi, M.: **Complexity and Approximation**, Springer, 1999.

Zahlen mit $0 < a_i \leq b$ für $i = 1, \dots, n$ und $b \in \mathbf{N}$ („Behältergröße“). Gesucht ist eine Platzierung 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 $k = \lfloor (r-1)^2 \cdot n'/2 \rfloor$ und $m = \lfloor n'/k \rfloor$. Die Objekte in I' werden nach absteigender Größe sortiert; die entstehende Instanz sei $[a'_1, \dots, a'_{n'}]$, d.h. $a'_1 \geq \dots \geq a'_{n'}$. Diese Zahlenfolge wird in $m+1$ Gruppen G_i mit $G_i = \{a'_{(i-1)k+1}, \dots, a'_{ik}\}$ für $i = 1, \dots, m$ und $G_{m+1} = \{a'_{m \cdot k + 1}, \dots, a'_{n'}\}$ eingeteilt. Es wird eine Instanz I'' definiert, indem jedes Objekt in Gruppe G_i für $i = 2, \dots, m+1$ durch das größte Objekt in der Gruppe, d.h. durch $a'_{(i-1)k+1}$ 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 Platzierung der Objekte in I'' in eine möglichst kleine Anzahl Behälter der Größe b , bestimmt.
4. In der optimalen Lösung für I'' werden für $i = 2, \dots, m+1$ 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_i 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 $O(n^q \cdot p(n))$ ist, wobei q nur von r abhängt und p ein Polynom ist. Für festes r ist der Aufwand also polynomiell.

Weiterhin lässt sich zeigen, dass die Anzahl der ermittelten Behälter durch $r \cdot m^*(I) + 1$ beschränkt ist.

///

Das im Satz erwähnte asymptotische Approximationsschema, das polynomiell in der Problemgröße und in $1/(r-1)$ ist, verwendet fortgeschrittene Methoden der Angewandten Mathematik³³.

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 6.2-9 nicht in \mathbf{PTAS} ; es ist nach Satz 6.3-6 jedoch in \mathbf{PTAS}^∞ . 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}^∞ ordnet sich in die übrigen Approximationsklassen ein:

Satz 6.3-7:

Ist $\mathbf{P} \neq \mathbf{NP}$, so gilt $\mathbf{FPTAS} \subset \mathbf{PTAS} \subset \mathbf{PTAS}^\infty \subset \mathbf{APX} \subset \mathbf{NPO}$.

³³ Khachian, L.G.: A polynomial algorithm for linear programming, Doklady Akademiy Nauk SSSR 244, 1979.

7 Weiterführende Konzepte

Die Analyse des Laufzeitverhaltens eines Algorithmus \mathbf{A} im schlechtesten Fall $T_{\mathbf{A}}(n) = \max\{t_{\mathbf{A}}(x) \mid x \in \Sigma^* \text{ und } |x| \leq n\}$ liefert eine *Garantie* (obere Schranke) für die Zeit, die er bei einer Eingabe zur Lösung benötigt³⁴. Für jede Eingabe $x \in \Sigma^*$ mit $|x| = n$ gilt $t_{\mathbf{A}}(x) \leq T_{\mathbf{A}}(n)$. 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 \mathbf{A} untersuchen, wie es sich im Mittel darstellt. Die **Zeitkomplexität** von \mathbf{A} im **Mittel** wird definiert als

$$T_{\mathbf{A}}^{avg}(n) = \mathbf{E}[t_{\mathbf{A}}(x) \mid |x| = n] = \int_{|x|=n} t_{\mathbf{A}}(x) \cdot dP(x).$$

Die Untersuchung des mittleren Laufzeitverhaltens von Algorithmen ist in den meisten Fällen sehr viel schwieriger als die worst-case-Analyse. Trotzdem gibt es sehr ermutigende Ergebnisse, insbesondere bei der Lösung einiger „klassischer“ Probleme. Kapitel 7.1 liefert hierzu einige Beispiele. Der Einbau von Zufallsexperimenten in Algorithmen hat insbesondere auf dem Gebiet der Approximationsalgorithmen für Optimierungsprobleme aus **NPO** gute Ergebnisse geliefert. In neuerer Zeit hat der Einsatz von probabilistischen Modellen zu weiteren Erkenntnissen auf dem Weg zur Lösung der **P-NP**-Frage geführt.

7.1 Beispiele wahrscheinlichkeitstheoretischer Ansätze

Zu einigen in Kapitel 6.2 vorgestellten Approximationsalgorithmen für das Binpacking-Minimierungsproblem gibt umfangreiche Untersuchungen für den Fall, dass die Eingabewerte einer bekannten Verteilung unterliegen, insbesondere für den Fall, dass sie gleichverteilt sind. Diese Untersuchungen sind auch weiterhin Gegenstand der aktuellen Forschung. Die Analyse des Laufzeitverhaltens im Mittel erfordert in den meisten Fällen tiefgehende mathematische Hilfsmittel, so dass hier lediglich einige Resultate mit Hinblick auf die angegebene Literatur zitiert werden³⁵.

³⁴ $t_{\mathbf{A}}(x)$ = Anzahl der Anweisungen, die von \mathbf{A} zur Berechnung von $\mathbf{A}(x)$ durchlaufen werden.

³⁵ Siehe: Hofri, M.: **Probabilistic Analysis of Algorithms**, Springer 1987; Hoffmann, U.: Stochastische Packungsalgorithmen, Dissertation Universität Stuttgart, 1981; Ausiello, G.; Crescenzi, P.; Gambosi, G.; Kann, V.; Marchetti-Spaccamela, A.; Protasi, M.: **Complexity and Approximation**, Springer, 1999; Naaman, N.; Rom, R.: Average Case Analysis of Bounded Space Bin Packing Algorithms, *Algorithmica* 50:2008, 72 – 97.

Satz 7.1-1:³⁶

Es bezeichne $m^*(I_n)$ das Maß einer optimalen (minimalen) Lösung bei einer Eingabeinstanz $I_n = [a_1, \dots, a_n]$ für das Binpacking-Minimierungsproblem. Die Objekte in der Eingabeinstanz seien im Intervall $]0,1]$ gleichverteilt, d.h. es ist $P(a_i \leq x) = x$ für $i = 1, \dots, n$.

(1) Unter diesen Voraussetzungen gilt für den Nextfit-Algorithmus:

Bezeichnet B die Zufallsvariable, die die Füllhöhe eines Behälters angibt, so ist $P(B \leq x) = x^3$ für $0 \leq x \leq 1$, $\mathbf{E}[B] = 3/4$, $\text{Var}(B) = 3/80 = 0,0375$.

Für die Anzahl $m(I_n, \mathbf{NF}(I_n))$ der vom Nextfit-Algorithmus benötigten Behälter bei Eingabe von I_n ergibt sich:

$$\mathbf{E}[m(I_n, \mathbf{NF}(I_n))] = \begin{cases} 1 & \text{für } n = 1 \\ 2/3 \cdot n + 1/6 & \text{für } n \geq 2 \end{cases};$$

$$\text{Var}(m(I_n, \mathbf{NF}(I_n))) = \begin{cases} 0 & \text{für } n = 1 \\ 1/4 & \text{für } n = 2 \\ 17/36 & \text{für } n = 3 \\ (32 \cdot n - 13)/180 & \text{für } n \geq 4 \end{cases}.$$

(2) Für den FirstfitDecreasing-Algorithmus bzw. den BestfitDecreasing-Algorithmus gilt:

$$\mathbf{E}[m(I_n, \mathbf{FFD}(I_n)) - m^*(I_n)] \in O(\sqrt{n}). \text{ bzw.}$$

$$\mathbf{E}[m(I_n, \mathbf{BFD}(I_n)) - m^*(I_n)] \in O(\sqrt{n}).$$

Das Binpacking-Minimierungsproblem, das in \mathbf{PTAS}° liegt, ist asymptotisch beliebig dicht in polynomieller Zeit approximierbar. 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 absteigender 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 be-

³⁶ Bezüglich der verwendeten Notation wird auf Kapitel 6.2 verwiesen.

geschrieben, das voraussetzt, dass die Größen der Objekte einer Eingabeinstanz über dem Intervall $]0, 1]$ gleichverteilt sind:

**Probabilistisches online-Approximationsschema für das Binpacking-Minimierungsproblem
(online-Faltungsalgorithmus)**

Eingabe: $I = [a_1, \dots, a_n]$, $s \in \mathbf{N}$ mit $s \geq 4$ und $s \equiv 0 \pmod{2}$
 a_1, \dots, a_n („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, \dots, n$

```

Verfahren:  VAR idx : INTEGER;
            k      : INTEGER;
            I      : ARRAY [1..s] OF Intervall  $\subseteq ]0,1]$ ;

            BEGIN
            FOR idx := 1 TO s DO
                I[idx] :=  $\left] \frac{s-idx}{s}, \frac{s-idx+1}{s} \right]$ ;

            FOR k := 1 TO n DO
                BEGIN
                    idx :=  $s + 1 - \lceil s \cdot a_k \rceil$ ; { Bestimmung eines Intervalls
                                                            I[idx] mit  $a_k \in I[idx]$  }

                    IF (idx >= 2)
                        AND
                        (es gibt einen Behälter  $B$ , der ein Objekt  $b \in I[s-(idx-2)]$ 
                        enthält)
                    THEN BEGIN
                         $B := B \cup \{a_k\}$ ;
                        kennzeichne  $B$  als gefüllt;
                    END
                    ELSE BEGIN
                        plaziere  $a_k$  in einen neuen Behälter;
                        { für idx = 1 ist dieser Behälter gefüllt }
                    END;
                END;
            END;

```

Ausgabe: $\text{OFD}(I) =$ benötigte nichtleere Behälter B_1, \dots, 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, zeigen³⁷:

Satz 7.1-2:

Bezeichnet $m^*(I_n)$ das Maß einer optimalen (minimalen) Lösung bei einer Eingabeinstanz $I_n = [a_1, \dots, a_n]$ für das Binpacking-Minimierungsproblem und sind die Objekte in der Eingabeinstanz im Intervall $]0, 1]$ gleichverteilt, so gilt

$$1 - \frac{1}{s+1} \leq \lim_{n \rightarrow \infty} \mathbf{E} \left[m^*(I_n) / m(I_n, \mathbf{OFD}(I_n)) \right] \leq 1.$$

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 in der Praxis besonders interessant, da unter dieser Voraussetzung kleine Objekte häufiger als große Objekte auftreten.

In den vorhergehenden Beispielen werden wahrscheinlichkeitstheoretische Voraussetzungen an den Input der Algorithmen gestellt. Die Algorithmen selbst sind rein deterministisch, letztlich im Modell einer deterministischen Turingmaschine. In vielen Anwendungen kennt man die Verteilung der Eingabewerte jedoch nicht bzw. es können nur sehr allgemeine Eigenschaften der Eingabewerte genutzt werden. Dann hat folgender Ansatz gute Ergebnisse erzielt:

Im ansonsten deterministischen Algorithmus werden als zulässige Elementaroperationen Zufallsexperimente zugelassen. Ein derartiges Zufallsexperiment kann beispielsweise mit Hilfe eines Zufallszahlengenerators ausgeführt werden. So wird 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 Objekte eines aus mehreren Elementen zusammengesetzten Inputs, die im weiteren Ablauf des Algorithmus als nächstes untersucht werden sollen.

Man nennt derartige Algorithmen **randomisierte Algorithmen (probabilistische Algorithmen)**. Eine Einführung in die Entwurfsmethoden derartiger Verfahren findet man in der angegebenen Literatur.

³⁷ Hoffmann, U.: A class of simple stochastic bin packing algorithms, Computing 29, 227 – 239, 1982.

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**³⁸, 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, \dots, 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, das an die richtige Position geschoben werden soll, in dem jeweiligen Feldabschnitt 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 Primzahltafel (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

³⁸ 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 jeweils $O((\beta(n))^2)$ viele Bitoperationen erforderlich. Daher ist der Gesamtaufwand mindestens von der Ordnung $O(\beta(n) \cdot 2^{\beta(n)/2})$.

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 ist³⁹. Die Laufzeit dieses Verfahrens ist von der Ordnung $O((\log(n))^2)$ und damit trotz der polynomiellen Laufzeit für große n nicht praktikabel. Der bis dahin bekannte schnellste Algorithmus zur Überprüfung einer Zahl n auf Primzahleigenschaft, der APRCL-Test, hat eine Laufzeit der Ordnung $O((\log(n))^{c(\log(\log(\log(n))))})$ 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 $1 \leq a \leq n-1$, 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 $2 \leq p \leq \lfloor \sqrt{n} \rfloor$ 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 $1 \leq a \leq n-1$ 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 $1 \leq a \leq n-1$ zu
- (iii) falls n keine Primzahl ist, dann trifft $E(a)$ für weniger als die Hälfte aller Zahlen a mit $1 \leq a \leq n-1$ zu.

³⁹ Agrawal, M.; Kayal, N.; Saxena, N.: PRIMES is in P, preprint, <http://www.cse.iitk.ac.in/news/primalty.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;
    END;

    random_is_prim := is_prime;

  END { random_is_prime };

```

Der Algorithmus versucht also, m Zeugen für die Primzahleigenschaft von n zu finden. Ist n Primzahl, dann sind wegen (ii) alle m erzeugten Werte a Zeugen für die Primzahleigenschaft von n , und der Algorithmus gibt die korrekte Antwort. Ist n keine Primzahl und es wird zufällig eine Zahl a mit $1 \leq a \leq n-1$ erzeugt, für die $E(a)$ nicht zutrifft, dann wird ebenfalls die korrekte Antwort gegeben. Nur für den Fall, dass n keine Primzahl ist und doch m Werte a gefunden wurden, für die $E(a)$ zutrifft, liefert der Algorithmus eine fehlerhafte Entscheidung. Die Wahrscheinlichkeit, bei einer Zahl n , die nicht Primzahl ist, zufällig m Zeugen zu finden,

ist wegen (iii) kleiner als $(1/2)^m$, d.h. die Wahrscheinlichkeit einer fehlerhaften Entscheidung ist in diesem Fall kleiner als $(1/2)^m$. Insgesamt ist die Wahrscheinlichkeit einer korrekten Antwort des Algorithmus größer als $1 - (1/2)^m$. Da wegen (i) die Eigenschaft $E(a)$ leicht zu überprüfen ist, ist hiermit ein effizientes Verfahren beschrieben, das mit beliebig hoher Wahrscheinlichkeit eine korrekte Antwort liefert. Diese ist beispielsweise für $m = 20$ größer als 0,999999.

Die Frage stellt sich, ob es geeignete Zeugeneigenschaften $E(a)$ gibt. Einen ersten Versuch legt der Satz von Fermat nahe⁴⁰:

Satz 7.1-3:

Ist $n \in \mathbf{N}$ eine Primzahl und $a \in \mathbf{N}$ eine Zahl mit $\text{ggT}(a, n) = 1$, dann ist $a^{n-1} \equiv 1 \pmod{n}$.

Für die Primzahl $n = 13$ zeigt folgende Tabelle für alle a mit $1 \leq a \leq n-1$ die Werte $a^i \pmod{13}$ mit $i = 1, \dots, 12$:

⁴⁰ Auf die Beweise der folgenden Sätze aus der elementaren Zahlentheorie wird hier mit Verweis auf die angegebene Literatur, insbesondere auf Yan, S.Y.: **Number Theory für Computing**, Springer 2000, und Salomaa, A.: **Public Key Cryptography**, Springer 1996, verzichtet.

$a \bmod 13$	$a^i \bmod 13$ für $i = 1, \dots, 12$
1	1 für $i = 1, \dots, 12$
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 $1 \leq a \leq n-1$ und $ggT(a, n) = 1$, dann ist $a^{n-1} \equiv 1 \pmod{n}$. In diesem Fall gilt also für alle Zahlen a mit $1 \leq a \leq n-1$ und $ggT(a, n) = 1$ die Eigenschaft $E(a)$, und das sind mehr als die Hälfte aller Zahlen a mit $1 \leq a \leq n-1$.

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 $i = 1, \dots, 14$
1	1 für $i = 1, \dots, 14$
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, $16 \equiv 1$, 4, $16 \equiv 1$ <i>(falscher) Zeuge dafür, dass 15 Primzahl ist</i>
5	5, $25 \equiv 10$, $50 \equiv 5$, $25 \equiv 10$, $50 \equiv 5$, $25 \equiv 10$, $50 \equiv 5$, $25 \equiv 10$, $50 \equiv 5$, $25 \equiv 10$, $50 \equiv 5$, $25 \equiv 10$, $50 \equiv 5$, $25 \equiv 10$, $50 \equiv 5$, $25 \equiv 10$
6	6, $36 \equiv 6$, $36 \equiv 6$, $36 \equiv 6$, $36 \equiv 6$, $36 \equiv 6$, $36 \equiv 6$, $36 \equiv 6$, $36 \equiv 6$, $36 \equiv 6$, $36 \equiv 6$, $36 \equiv 6$, $36 \equiv 6$, $36 \equiv 6$, $36 \equiv 6$, $36 \equiv 6$
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, $49 \equiv 4$, $28 \equiv 13$, $91 \equiv 1$, 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, $64 \equiv 4$, $32 \equiv 2$, $16 \equiv 1$, 8, $64 \equiv 4$
9	9, $81 \equiv 6$, $54 \equiv 9$, $81 \equiv 6$, $54 \equiv 9$, $81 \equiv 6$, $54 \equiv 9$, $81 \equiv 6$, $54 \equiv 9$, $81 \equiv 6$, $54 \equiv 9$, $81 \equiv 6$, $54 \equiv 9$, $81 \equiv 6$, $54 \equiv 9$, $81 \equiv 6$
10	10, $100 \equiv 10$, $100 \equiv 10$, $100 \equiv 10$, $100 \equiv 10$, $100 \equiv 10$, $100 \equiv 10$, $100 \equiv 10$, $100 \equiv 10$, $100 \equiv 10$, $100 \equiv 10$, $100 \equiv 10$, $100 \equiv 10$, $100 \equiv 10$, $100 \equiv 10$, $100 \equiv 10$
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, $121 \equiv 1$, 11, $121 \equiv 1$ <i>(falscher) Zeuge dafür, dass 15 Primzahl ist</i>
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, $196 \equiv 1$, 14, $196 \equiv 1$, 14, $196 \equiv 1$ <i>(falscher) Zeuge dafür, dass 15 Primzahl ist</i>

Satz 7.1-4:

Entweder gilt für alle Zahlen a mit $1 \leq a \leq n-1$ und $\text{ggT}(a, n) = 1$ die Eigenschaft $a^{n-1} \equiv 1 \pmod{n}$ oder für höchstens die Hälfte.

Anders formuliert:

Entweder sind mit der Zeugeigenschaft $E(a) = \text{„ggT}(a, n) = 1 \text{ und } a^{n-1} \equiv 1 \pmod{n}\text{“}$ alle Zahlen a mit $1 \leq a \leq n-1$ 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 7.1-5:

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 $1 \leq a \leq n-1$, dann ist $\text{ggT}(a, n) = 1$ und folglich $a^{n-1} \equiv 1 \pmod{n}$. Wegen $a^{n-1} = a^{(2^{j-1} \cdot r) \cdot 2} = (a^{2^{j-1} \cdot r})^2 \equiv 1 \pmod{n}$ folgt mit Satz 7.1-5 (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 7.1-5 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 7.1-6:

Es sei n eine ungerade Primzahl, $n = 1 + 2^j \cdot r$ mit ungeradem r und $j > 0$. Dann gilt für jedes a mit $1 \leq a \leq n-1$:

Die Folge $(a^r, a^{2r}, a^{4r}, \dots, a^{2^{j-1} \cdot r}, a^{2^j \cdot r})$ der Länge $j+1$, wobei alle Werte modulo n reduziert werden, hat eine der Formen

$(1, 1, 1, \dots, 1, 1)$ oder

$(*, *, \dots, *, -1, 1, \dots, 1, 1)$.

Hierbei steht das Zeichen „*“ für eine Zahl, die verschieden von 1 oder -1 ist.

Wenn die in Satz 7.1-6 beschriebene Folge eine der drei Formen

$(*, *, \dots, *, 1, 1, \dots, 1, 1)$,

$(*, *, \dots, *, -1)$ oder

$(*, *, \dots, *, *, \dots, *)$

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 $1 \leq a \leq n-1$ die Folge $(a^r, a^{2r}, a^{4r}, \dots, a^{2^{j-1} \cdot r}, a^{2^j \cdot r}) \pmod{n}$ eine der beiden Formen $(1, 1, 1, \dots, 1, 1)$ oder $(*, *, \dots, *, -1, 1, \dots, 1, 1)$ hat. In diesem Fall heißt n **streng pseudoprim zur Basis a** . Es gilt folgender Satz:

Satz 7.1-7:

Es sei n eine ungerade zusammengesetzte Zahl. Dann ist n streng pseudoprim für höchstens ein Viertel aller Basen a mit $1 \leq a \leq n-1$.

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 $(a^r, a^{2r}, a^{4r}, \dots, a^{2^{j-1}r}, a^{2^j r}) \pmod{n}$ hat eine der Formen $(1, 1, 1, \dots, 1, 1)$ oder $(*, *, \dots, *, -1, 1, \dots, 1, 1)$ “.

Die Wahrscheinlichkeit einer korrekten Antwort des Algorithmus mit dieser Zeugeneigenschaft ist wegen Satz 7.1-7 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 $1 \leq a \leq n-1$ die Zeugeneigenschaft zu überprüfen um sicher zu gehen, dass n eine Primzahl ist: Nur eine zusammengesetzte Zahl $n < 2,5 \cdot 10^{10}$, nämlich $n = 3.215.031.751$, ist streng pseudoprim zu den vier Basen $a = 2, 3, 5$ und 7 . Für praktische Belange ist daher das Verfahren ein effizienter Primzahltest.

Untersucht man große Zahlen, die spezielle Formen aufweisen, etwa Mersenne-Zahlen, auf Primzahleigenschaft bieten sich speziell angepasste Testverfahren an. Schließlich gibt es eine Reihe von Testverfahren, die andere zahlentheoretische Eigenschaften nutzen.

7.2 Modelle randomisierter Algorithmen

Die Zusammenführung der obigen Ansätze mit den Modellen aus der Theorie der Berechenbarkeit (Turingmaschinen, deterministische und nichtdeterministische Algorithmen) erfordert die Entwicklung weiterer Berechnungsmodelle.

In einem randomisierten Algorithmus wird an mindestens einer Stelle der nächste Schritt von einer Zufallsentscheidung abhängig gemacht. Beispielsweise wird im `QUICKSORT` der Zugriff auf das Element, das anschließend an die richtige Position innerhalb der sortierten Folge geschoben wird, durch das Lesen einer Zufallszahl bestimmt. Im Primzahltest aus Kapitel 7.1 wird eine Zahl a zwischen 1 und $n-1$, ein potentieller Zeuge für die Primzahleigenschaft von n , zufällig ausgewählt. Der Algorithmus erhält also neben einer Eingabeinstanz weitere „zufällige Inputs“, die jeweils den weiteren Ablauf steuern. Um ein möglichst einfaches allgemeingültiges Modell der Eingabe für Zufallsentscheidungen zu erhalten, werden die in einem randomisierten Algorithmus durchgeführten Zufallsentscheidungen aus einer zusätzlichen Eingabefolge $\tau = (b_i)_{i \in \mathbb{N}}$ von stochastisch unabhängigen Bits b_i mit $P(b_i = 0) = P(b_i = 1) = 1/2$ generiert. Sobald der Algorithmus eine weitere Zufallszahl a benötigt, liest er die erforderlichen nächsten Bits aus τ , die dann als Binärdarstellung von a interpretiert werden. Findet der Algorithmus den Ausgang einer `IF ... THEN ... ELSE` – Entscheidung durch einen zufälligen Münzwurf, so kann diese Entscheidung durch das Lesen eines weiteren Bits aus τ modelliert werden. Die Eingabe eines randomisierten Algorithmus besteht daher aus einer Eingabeinstanz für das jeweilige Problem und einer Folge $\tau = (b_i)_{i \in \mathbb{N}}$ stochastisch unabhängiger Bits b_i mit $P(b_i = 0) = P(b_i = 1) = 1/2$.

Ein Modell zur Modellierung zufallsgesteuerter Algorithmen ist die **randomisierte Turingmaschine (probabilistische Turingmaschine)**. Einige Vorbemerkungen sollen das Verständnis der Arbeitsweise erleichtern.

Als Grundmodell wird die nichtdeterministische Turingmaschine gewählt; eine deterministische Turingmaschine ist dann eine nichtdeterministische Turingmaschine, deren Überföhrungsfunktion immer genau eine Alternative aufweist. Dasselbe gilt für den deterministischen Teil der Überföhrungsfunktion einer nichtdeterministischen Turingmaschine. Ein deterministischer Wert $\delta(q, a_1, \dots, a_k) = \{(q_1, (b_{11}, d_{11}), \dots, (b_{1k}, d_{1k}))\}$ wird ersetzt durch $\delta(q, a_1, \dots, a_k) = \{(q_1, (b_{11}, d_{11}), \dots, (b_{1k}, d_{1k})), (q_1, (b_{11}, d_{11}), \dots, (b_{1k}, d_{1k}))\}$, so dass dieser als nichtdeterministischer Wert mit genau zwei identischen Alternativen angesehen werden kann⁴¹. Hat die Überföhrungsfunktion eines nichtdeterministischen Algorithmus einen Wert der Form

⁴¹ Formal handelt es sich hierbei um eine Multimenge.

$\delta(q, a_1, \dots, a_k) = \{(q_1, (b_{11}, d_{11}), \dots, (b_{1k}, d_{1k})), \dots, (q_t, (b_{t1}, d_{t1}), \dots, (b_{tk}, d_{tk}))\}$ mit $t \geq 3$, dann kann dieser ersetzt werden durch die $t-1$ Werte

$$\delta(q, a_1, \dots, a_k) = \{(q_1, (b_{11}, d_{11}), \dots, (b_{1k}, d_{1k})), (q'_2, (a_1, S), \dots, (a_k, S))\},$$

$$\delta(q'_i, a_1, \dots, a_k) = \{(q_i, (b_{i1}, d_{i1}), \dots, (b_{ik}, d_{ik})), (q'_{i+1}, (a_1, S), \dots, (a_k, S))\} \text{ für } i = 2, \dots, t-2,$$

$$\delta(q'_{t-1}, a_1, \dots, a_k) = \{(q_{t-1}, (b_{t-1,1}, d_{t-1,1}), \dots, (b_{t-1,k}, d_{t-1,k})), (q_t, (b_{t1}, d_{t1}), \dots, (b_{tk}, d_{tk}))\}$$

mit neuen Zuständen q'_2, \dots, q'_{t-1} . Auf diese Weise wird der „Grad des Nichtdeterminismus“ gleich 2, d.h. im Berechnungsbaum (siehe Kapitel 2.5), dessen Knoten mit den während einer Berechnung erreichbaren Konfigurationen markiert sind, gibt es an jeder Stelle genau zwei Alternativen. Die Länge einer nichtdeterministischen Berechnung wird dadurch maximal um einen konstanten Faktor verlängert. Ist die Berechnung polynomiell zeitbeschränkt, so bleibt sie es auch nach dieser Änderung.

Im folgenden werden Entscheidungsalgorithmen für entscheidbare Probleme betrachtet, die zeitbeschränkt durch eine zeitkonstruierbare Funktion f sind. Daher ist für jede Eingabe x von Beginn an klar, nach wie vielen Schritten die ja/nein-Entscheidung spätestens getroffen wird, nämlich nach maximal $c \cdot f(|x|)$ vielen Schritten (mit einer Konstanten $c > 0$). Wird x akzeptiert, dann gibt es im Berechnungsbaum, dessen Wurzel mit der Anfangskonfiguration mit x markiert ist, einen Pfad der Länge $c \cdot f(|x|)$, der bei einem Blatt endet, das mit einer akzeptierenden Konfiguration ist, die also den Zustand q_{accept} enthält. Längere Pfade brauchen nicht betrachtet zu werden. Kürzere Pfade, die dadurch entstehen, dass sie in einer Konfiguration enden, für die es aufgrund der Überföhrungsfunktion keine Nachfolgekonfiguration gibt, können durch Hinzufügen einer geeigneten Anzahl von „Dummy“-Schritten bis auf die Länge $c \cdot f(|x|)$ gebracht werden und enden dann mit einer Konfiguration, die den verwerfenden Zustand q_{accept} enthält. Daher kann angenommen werden, dass in dem Berechnungsbaum bei Eingabe von x alle Pfade gleich lang sind, und dass dieser ein vollständiger Binärbaum der Höhe $c \cdot f(|x|)$ ist. Jedes Blatt repräsentiert genau einen der $2^{c \cdot f(|x|)}$ vielen möglichen Berechnungspfade. Für eine von der Turingmaschine akzeptierte Eingabe ist mindestens ein Blatt mit einer akzeptierenden Konfiguration markiert. Wird x nicht akzeptiert, enthält keine Konfiguration in den Blättern den akzeptierenden Zustand.

Für die nichtdeterministische Turingmaschine TM sollen Annahmen getroffen werden, die aufgrund dieser Vorbemerkungen nicht einschränkend sind:

1. Der Grad des Nichtdeterminismus von TM ist 2, d.h. jeder Wert in der Überföhrungsfunktion besteht aus genau 2 Alternativen.
2. Jeder Schritt von TM ist nichtdeterministisch, d.h. in einer Folge von Konfigurationen, die eine Berechnung beschreiben, wird in jedem Schritt eine nichtdeterministische Entscheidung (wegen 1. aus zwei Alternativen) getroffen.
3. Für eine Eingabe x sind alle möglichen Berechnungspfade gleichlang.

Die Turingmaschine TM ist eine **randomisierte Turingmaschine (probabilistische Turingmaschine)**, wenn sie bei einer Eingabe in jedem Schritt i auf dem Berechnungspfad eine der beiden möglichen Folgekonfigurationen $K_{i,0}$ bzw. $K_{i,1}$ zufällig auswählt. Dazu liest sie in jedem Schritt aus einer Folge $\tau = (b_i)_{i \in \mathbb{N}}$ stochastisch unabhängiger Bits b_i mit $P(b_i = 0) = P(b_i = 1) = 1/2$ das nächste Bit b_i und geht für $b_i = 0$ in die Konfiguration $K_{i,0}$ bzw. für $b_i = 1$ in die Konfiguration $K_{i,1}$. Der Nichtdeterminismus wird in jedem Schritt also durch einen zufallsgesteuerten „Münzwurf“ ($b_i = 0$ oder $b_i = 1$) ersetzt.

Da jede Berechnung als zeitlich beschränkt angesehen wird, werden aus der Zufallsfolge $\tau = (b_i)_{i \in \mathbb{N}}$ jeweils nur endlich viele Bits benötigt, so dass die Folge der Zufallsbits auch als Wort $\tau \in \{0, 1\}^*$ aufgefasst werden kann.

Für eine Eingabe x ist das Ergebnis der Berechnung („ x wird akzeptiert“ bzw. „ x wird nicht akzeptiert“) somit ein Zufallsereignis. Im Berechnungsbaum zu x , der $m = 2^{c \cdot f(|x|)}$ viele Blätter bzw. verschiedene Berechnungspfade besitzt, seinen k_{accept} viele Blätter mit akzeptierenden Konfigurationen und $m - k_{\text{accept}}$ viele Blätter mit nicht akzeptierenden (verwerfenden) Konfigurationen markiert. Dann ist die **Wahrscheinlichkeit, dass x akzeptiert wird (Akzeptanzwahrscheinlichkeit von x)**, gleich $P(x \text{ wird akzeptiert}) = k_{\text{accept}}/m$. Entsprechend ist die **Wahrscheinlichkeit, dass x nicht akzeptiert wird (Ablehnungswahrscheinlichkeit von x)**, gleich $P(x \text{ wird nicht akzeptiert}) = (m - k_{\text{accept}})/m = 1 - k_{\text{accept}}/m$.

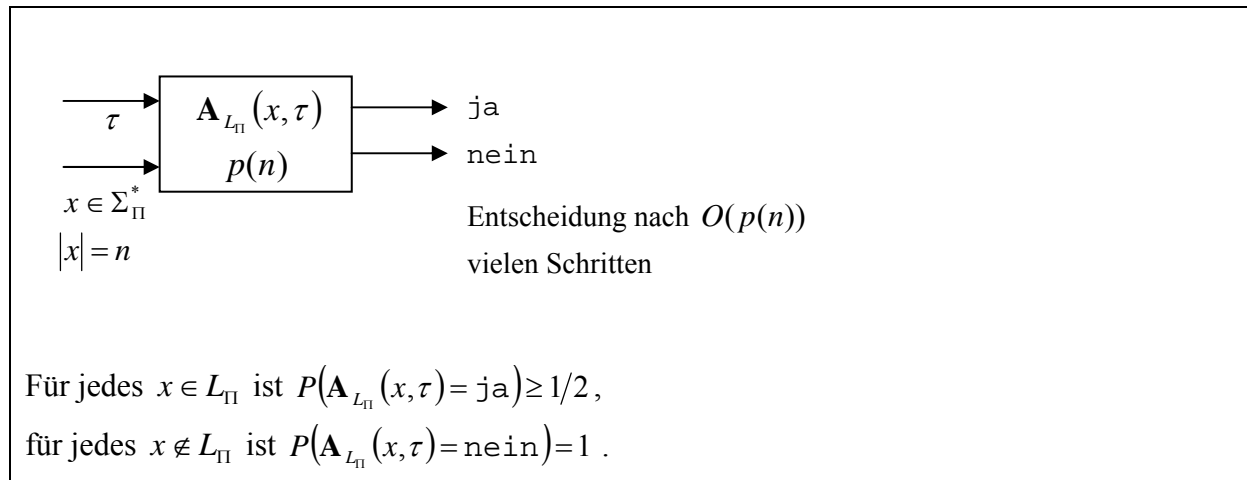
In den vorherigen Kapiteln wurden Turingmaschinen entworfen, um Sprachen zu entscheiden, genauer: Eine Turingmaschine TM (ohne Randomisierung) entscheidet die Sprache L , wenn gilt: TM akzeptiert die Eingabe x genau dann, wenn $x \in L$ ist. Für $x \in L$ wird x also akzeptiert; für $x \notin L$ wird x nicht akzeptiert. Eine randomisierte Turingmaschine kann bei einer Eingabe $x \in L$ diese auch nicht akzeptieren, nämlich dann, wenn für den Wert k_{accept}/m , d.h. für den Anteil akzeptierender Berechnungspfade, $k_{\text{accept}}/m < 1$ gilt. Umgekehrt kann eine Eingabe x mit $x \notin L$ durch die randomisierte Turingmaschine durchaus akzeptiert werden, nämlich dann, wenn $k_{\text{accept}}/m > 0$ ist. Es sind also verschiedene Typen von Fehlern möglich, die bei der Entscheidung durch eine randomisierte Turingmaschine auftreten können. Je nach Festlegung der erlaubten Fehlerarten bei der Berechnung einer randomisierten Turingmaschine werden unterschiedliche Untertypen randomisierter Turingmaschinen definiert (PP-Typ, ZPP-Typ, RP-Typ, BPP-Typ, jeweils mit weiteren Untertypen⁴²). Hier soll eine Beschränkung auf den RP-Typ erfolgen, da dieser eine neue Sicht auf die Klasse **NP** erlaubt.

⁴² Siehe: Bovet, D.P.; Crescenzi, P.: **Introduction to the Theory of Complexity**, Prentice Hall, 1994; Wegener, I.: **Komplexitätstheorie**, Springer, 2003.

Im folgenden werden Entscheidungsverfahren wieder als Algorithmen beschrieben, die auch als Programm in einer Programmiersprache formuliert sein können. Die obigen Annahmen 1. bis 3. lassen sich übertragen: Ein Algorithmus führt anstelle nichtdeterministischer Schritte durch Zufallsexperimente gesteuerte Entscheidungen aus, wobei diese zufällige Münzwürfe modelliert werden. Die benötigte Folge $\tau \in \{0,1\}^*$ stochastisch unabhängiger Bits b_i mit $P(b_i = 0) = P(b_i = 1) = 1/2$ wird als zusätzliche Eingabe bereitgestellt. Die Eingabe für \mathbf{A} sind also x und die Zufallsfolge $\tau \in \{0,1\}^*$. Wenn der Algorithmus \mathbf{A} bei einer Eingabe x eine durch $c \cdot f(|x|)$ beschränkte Anzahl von Schritten ausführt, bis er seine ja/nein-Entscheidung trifft und die jeweils von Zufallsbits gesteuert werden, gibt es für diese Eingabe $m = 2^{c \cdot f(|x|)}$ mögliche Berechnungen. Der Anteil der Berechnungen, die auf die ja-Entscheidung führen ($\mathbf{A}(x, \tau) = \text{ja}$), sei k_{ja} , der Anteil der Berechnungen, die auf die nein-Entscheidung führen ($\mathbf{A}(x, \tau) = \text{nein}$), entsprechend $(m - k_{\text{ja}})/m = 1 - k_{\text{ja}}/m$. Dann ist $P(\mathbf{A}(x, \tau) = \text{ja}) = k_{\text{ja}}/m$ und $P(\mathbf{A}(x, \tau) = \text{nein}) = 1 - k_{\text{ja}}/m$.

Es sei Π ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$. L_Π liegt in \mathbf{RP} (bzw. „ Π liegt in \mathbf{RP}^c “), wenn es einen Algorithmus (**RP-Akzeptor**) \mathbf{A}_{L_Π} 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 Abhängigkeit von der Größe $|x|$ der Eingabe) kommt der Akzeptor auf die ja-Entscheidung bzw. auf die nein-Entscheidung. Dabei gilt:

Für jedes $x \in L_\Pi$ ist $P(\mathbf{A}_{L_\Pi}(x, \tau) = \text{ja}) \geq 1/2$,
für jedes $x \notin L_\Pi$ ist $P(\mathbf{A}_{L_\Pi}(x, \tau) = \text{nein}) = 1$.



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}_{Π} bei mindestens der Hälfte aller möglichen durch eine Zufallsfolge τ gesteuerten Berechnungen 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, d.h. bei der Zufallsfolge, die nein-Entscheidung treffen.

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 $NONPRIMES$ 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 $k = \text{size}(n)$. Diese Zufallsfolge wird als Binärcodierung einer Zufallszahl a mit $1 \leq a \leq n-1$ interpretiert (die Fälle $a = 0$ und $a = n$ sollen zur vereinfachten Darstellung hier ausgeschlossen sein). Dazu wird angenommen, dass es eine Funktion `make_INTEGER(τ , $\text{size}(n)$)` gibt, die die Zufallsfolge τ in eine natürliche Zahl a mit $1 \leq a \leq n-1$ umwandelt. Dann wird die Bedingung $E(a)$ (siehe Kapitel 7.1) überprüft und eine Entscheidung getroffen:

```

FUNCTION  $\mathbf{A}_{NONPRIMES}$  ( $n$  : INTEGER;
                         $\tau$  : ... ) : ...;

VAR a : INTEGER;

BEGIN {  $\mathbf{A}_{NONPRIMES}$  }
  a := make_INTEGER( $\tau$ , size( $n$ ));
  IF ( $E(a)$  trifft nicht zu)
  THEN  $\mathbf{A}_{NONPRIMES}$  := ja
  ELSE  $\mathbf{A}_{NONPRIMES}$  := nein;
END {  $\mathbf{A}_{NONPRIMES}$  };

```

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 $1 \leq a \leq n-1$ zu. Daher trifft $E(a)$ für mehr als die Hälfte aller Zahlen a mit $1 \leq a \leq n-1$ nicht zu, und es gilt $P(\mathbf{A}_{NONPRIMES}(n, \tau) = \text{ja}) \geq 1/2$.

Ist $n \notin NONPRIMES$, d.h. n ist eine Primzahl, dann trifft nach Definition $E(a)$ für alle Zahlen a mit $1 \leq a \leq n-1$ zu. Daher antwortet $\mathbf{A}_{NONPRIMES}$ mit $\mathbf{A}_{NONPRIMES} := \text{nein}$.

Der einseitige Fehler kann durch Einsatz geeigneter Replikations-Techniken beliebig klein gehalten werden: Dazu wird der Algorithmus $\mathbf{A}_{L_{\Pi}}$ bei Eingabe von $x \in \Sigma_{\Pi}^*$ bis zu k -mal wiederholt, solange er zum Ergebnis $\mathbf{A}_{L_{\Pi}}(x, \tau) = \text{nein}$ kommt; bei jeder Wiederholung werden neue Zufallsbits verwendet. Die Wiederholungen werden abgebrochen, wenn der Algorithmus bei einer der Ausführungen den Ausgang $\mathbf{A}_{L_{\Pi}}(x, \tau) = \text{ja}$ erreicht. Ist $x \notin L_{\Pi}$, dann gibt $\mathbf{A}_{L_{\Pi}}$ wegen $P(\mathbf{A}_{L_{\Pi}}(x, \tau) = \text{nein}) = 1$ die richtige Antwort bei allen Durchläufen. Ist $x \in L_{\Pi}$, dann ist wegen $P(\mathbf{A}_{L_{\Pi}}(x, \tau) = \text{ja}) \geq 1/2$ die Wahrscheinlichkeit k -mal die falsche Antwort nein zu geben $\leq (1/2)^k$ bzw. $P(\mathbf{A}_{L_{\Pi}}(x, \tau) = \text{ja}) \geq 1 - (1/2)^k$.

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 **NP**-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 $\mathbf{A}_{L_{\Pi}}(x, \tau) = \text{ja}$ (da $x \in L_{\Pi}$ und in diesem Fall $P(\mathbf{A}_{L_{\Pi}}(x, \tau) = \text{ja}) \geq 1/2$ gelten, führen mindestens die Hälfte aller durch eine Zufallsfolge gesteuerten Berechnungen auf die ja-Entscheidung, d.h. es gibt mindestens eine als Beweis geeignete Zufallsfolge, die zu $\mathbf{A}_{L_{\Pi}}(x, \tau) = \text{ja}$ führt). Ist $x \notin L_{\Pi}$, dann führen wegen $P(\mathbf{A}_{L_{\Pi}}(x, \tau) = \text{nein}) = 1$ alle durch eine Zufallsfolge gesteuerten Berechnungen auf die nein-Entscheidung.

///

Ein weiteres Modell randomisierter Algorithmen erhält man durch Ersetzung der konstanten Fehlerwahrscheinlichkeit $1/2$ durch eine von der Eingabelänge abhängige Fehlerwahrscheinlichkeit:

Es sei Π ein Entscheidungsproblem Π mit der Menge $L_{\Pi} \subseteq \Sigma_{\Pi}^*$ und $\varepsilon : \mathbf{N} \rightarrow [0, 1[$ eine reellwertige Funktion. L_{Π} liegt in $\mathbf{RP}(\varepsilon(n))$ (bzw. „ Π liegt in $\mathbf{RP}(\varepsilon(n))$ “), wenn es einen Algorithmus (**RP**($\varepsilon(n)$)-**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 Abhängigkeit von der Größe $|x|$ der Eingabe) kommt der Akzeptor auf die ja-Entscheidung bzw. auf die nein-Entscheidung. Dabei gilt:

Für jedes $x \in L_{\Pi}$ ist $P(\mathbf{A}_{L_{\Pi}}(x, \tau) = \text{ja}) \geq 1 - \varepsilon(n)$ bzw. $P(\mathbf{A}_{L_{\Pi}}(x, \tau) = \text{nein}) \leq \varepsilon(n)$,
für jedes $x \notin L_{\Pi}$ ist $P(\mathbf{A}_{L_{\Pi}}(x, \tau) = \text{nein}) = 1$.

L_{Π} liegt in \mathbf{RP}^* (bzw. „ Π liegt in \mathbf{RP}^* “), wenn L_{Π} in $\mathbf{RP}(\varepsilon(n))$ liegt für eine Funktion $\varepsilon : \mathbf{N} \rightarrow [0, 1[$.

Für einige Typen von Fehlerwahrscheinlichkeitsfunktionen ε kann man wie im konstanten Fall durch wiederholte Durchläufe des jeweiligen Akzeptors eine Verbesserung im Fehlerverhalten erzielen:

Satz 7.2-2:

Es seien $p(n)$ und $q(n)$ Polynome. Dann gilt

$$\mathbf{RP}(1-1/p(n)) = \mathbf{RP}(2^{-q(n)}).$$

Beweis:

Der Lauf eines $\mathbf{RP}(1-1/p(n))$ -Akzeptors $A_{L_{\Pi}}$ wird $t(n)$ -mal mit $t(n) = \lceil \ln(2) \cdot p(n) \cdot q(n) \rceil$ wiederholt, solange er den nein-Ausgang erreicht. Dabei werden jeweils neue Zufallsbits verwendet. Wird bei jeder Wiederholung der nein-Ausgang erreicht, so gibt der Algorithmus die Antwort nein aus. Die Wiederholungen werden abgebrochen mit Antwort ja bei Erreichen des ja-Ausgangs.

Da $t(n)$ ein Polynom ist, bleibt die Laufzeit polynomiell beschränkt.

Ist $x \notin L_{\Pi}$, so wird in jeder Wiederholung die korrekte Antwort nein gegeben.

Ist $x \in L_{\Pi}$, so ist die Fehlerwahrscheinlichkeit, d.h. die Wahrscheinlichkeit, die Antwort nein zu geben, beschränkt durch $(1-1/p(n))^{\lceil \ln(2) \cdot p(n) \cdot q(n) \rceil}$. Es ist

$$\begin{aligned} (1-1/p(n))^{\lceil \ln(2) \cdot p(n) \cdot q(n) \rceil} &\leq (1-1/p(n))^{\ln(2) \cdot p(n) \cdot q(n)} && \text{(wegen } 1-1/p(n) < 1) \\ &\leq e^{-\ln(2) \cdot q(n)} && \text{(wegen } (1-1/m)^m < e^{-1}) \\ &= 2^{-q(n)}. \end{aligned}$$

///

Der folgende Satz beschreibt die Klasse \mathbf{NP} durch randomisierte Algorithmen und ersetzt damit den Nichtdeterminismus durch Zufallsexperimente:

Satz 7.2-3:

$$\mathbf{NP} = \mathbf{RP}^* .$$

Beweis:

Es sei $L \in \mathbf{NP}$, $L \subseteq \Sigma^*$. Dann gibt es eine nichtdeterministische Turingmaschine TM mit $L = L(TM)$, die polynomiell zeitbeschränkt mit einem Polynom $p(n)$ ist. Für TM kann man

die am Anfang des Kapitels getroffenen Annahmen voraussetzen. Es sei $x \in \Sigma^*$ mit $|x| = n$. Der Berechnungsbaum bei Eingabe von x in TM hat die Höhe $p(n)$ und $2^{p(n)}$ viele Blätter. Einen randomisierten Algorithmus \mathbf{A} erhält man, indem man jeden nichtdeterministischen Schritt in diesem Berechnungsbaum von einem Münzwurf abhängig macht. Der Algorithmus gibt ja aus, wenn dabei ein Blatt im Berechnungsbaum erreicht wird, das mit einer akzeptierenden Konfiguration markiert ist, ansonsten nein.

Ist $x \in L$, so gibt es mindestens ein Blatt, das mit einer akzeptierenden Konfiguration markiert ist. Daher ist $P(\mathbf{A}(x, \tau) = \text{ja}) \geq 1/2^{p(n)}$ bzw. $P(\mathbf{A}(x, \tau) = \text{nein}) \leq 1 - 1/2^{p(n)}$.

Ist $x \notin L$, so sind alle Blätter mit nicht-akzeptierenden Konfigurationen markiert. Daher ist $P(\mathbf{A}(x, \tau) = \text{nein}) = 1$.

Insgesamt ergibt sich $L \in \mathbf{RP}(1 - 1/2^{p(n)})$.

Ist umgekehrt $L \in \mathbf{RP}(\varepsilon(n))$, dann kann der \mathbf{RP} -Akzeptor für L als nichtdeterministischer Algorithmus aufgefasst werden, wobei die verwendete Zufallsfolge den Beweis darstellt (vgl. den Beweis zu Satz 7.2-2, d.h. $L \in \mathbf{NP}$).

///

Setzt man für eine Sprache $L \in \mathbf{NP}$ den nach Satz 7.2-3 existierenden \mathbf{RP} -Akzeptor ein, um L zu entscheiden, so kann dieser bei $x \in L$ eine fehlerhafte Antwort geben und zwar unter Umständen mit einer sehr hohen Fehlerwahrscheinlichkeit von $1 - 1/2^{p(n)}$, d.h. mit einer Fehlerwahrscheinlichkeit, die gegen 1 konvergiert. Eine Antwort nein des \mathbf{RP} -Akzeptors kann mit sehr hoher Wahrscheinlichkeit falsch sein. Für praktische Fälle ist der probabilistische Ansatz also wenig geeignet. Auch die Idee der Fehlerwahrscheinlichkeitsreduktion durch Wiederholungen wie im Beweis von Satz 7.2-2 ist in der Regel nicht möglich, da die dort angegebene Wiederholungsanzahl für den vorliegenden Fall gleich $t(n) = \lceil \ln(2) \cdot 2^{p(n)} \cdot q(n) \rceil$ lautet, also nicht polynomiell ist.

Die Zusammenführung der Konzepte des Zufalls und des Nichtdeterminismus bei polynomiell zeitbeschränktem Laufzeitverhalten führt auf die Klasse \mathbf{PCP} (probabilistically checkable proofs). Hierbei werden Verifizierer, wie sie bei der Definition der Klasse \mathbf{NP} eingeführt wurden, um die Möglichkeit erweitert, Zufallsexperimente auszuführen:

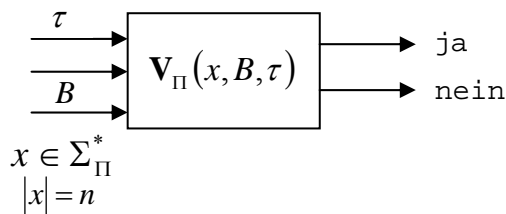
Es seien $r: \mathbf{N} \rightarrow \mathbf{N}$ und $q: \mathbf{N} \rightarrow \mathbf{N}$ Funktionen. Ein Verifizierer (nichtdeterministischer Algorithmus) \mathbf{V}_Π heißt $(r(n), q(n))$ -beschränkter Verifizierer für das Entscheidungsproblem Π über einem Alphabet Σ_Π , wenn er Zugriff auf eine Eingabe $x \in \Sigma_\Pi^*$ mit $|x| = n$, einen Beweis $B \in \Sigma_0^*$ und eine Zufallsfolge $\tau \in \{0, 1\}^*$ hat und sich dabei folgendermaßen verhält:

1. V_{Π} 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 V_{Π} $O(q(n))$ 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 V_{Π} in polynomieller Zeit auf die ja- bzw. nein-Entscheidung.

Die Entscheidung, die V_{Π} bei Eingabe von $x \in \Sigma_{\Pi}^*$, $B \in \Sigma_0^*$ und $\tau \in \{0,1\}^*$ liefert, wird mit $V_{\Pi}(x, B, \tau)$ 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 V_{Π} gibt, der L_{Π} in folgender Weise akzeptiert:

Für jedes $x \in L_{\Pi}$ gibt es einen Beweis B_x mit $P(V_{\Pi}(x, B_x, \tau) = \text{ja}) = 1$,
für jedes $x \notin L_{\Pi}$ und alle Beweise B gilt $P(V_{\Pi}(x, B, \tau) = \text{nein}) \geq 1/2$.



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

$$\mathbf{P} = \bigcup_{k=0}^{\infty} \mathbf{TIME}(n^k) = \mathbf{TIME}(poly(n)) \quad \text{und} \quad \mathbf{NP} = \bigcup_{k=0}^{\infty} \mathbf{NTIME}(n^k) = \mathbf{NTIME}(poly(n)).$$

Satz 7.2-4:

$$\mathbf{PCP}(r(n), q(n)) \subseteq \mathbf{NTIME}(q(n) \cdot 2^{O(r(n))})$$

Beweis:

Es sei $L_{\Pi} \in \mathbf{PCP}(r(n), q(n))$. Dann gibt es einen $(r(n), q(n))$ -beschränkten Verifizierer \mathbf{V}_{Π} für L_{Π} . Aus diesem kann man auf folgende Weise einen Verifizierer $\tilde{\mathbf{V}}_{\Pi}$ (im Sinne der Definition einer nichtdeterministischen Turingmaschine) für L_{Π} konstruieren:

Es sei $x \in \Sigma_{\Pi}^*$ mit $|x|=n$ und B_x ein Beweis. Für jede der $2^{O(r(n))}$ vielen Zufallsfolgen der Länge $O(r(n))$ simuliert $\tilde{\mathbf{V}}_{\Pi}$ in jeweils polynomiell vielen Schritten die Berechnung des Verifizierers \mathbf{V}_{Π} und akzeptiert x genau dann, wenn \mathbf{V}_{Π} die Eingabe x für jede Zufallsfolge akzeptiert. $\tilde{\mathbf{V}}_{\Pi}$ ist daher eine nichtdeterministische Turingmaschine, deren Laufzeit bei Eingabe von $x \in \Sigma_{\Pi}^*$ mit $|x|=n$ von der Ordnung $O(q(n) \cdot 2^{O(r(n))})$ ist. Außerdem gilt:

Ist $x \in L_{\Pi}$, dann gibt es einen Beweis B_x mit $P(\mathbf{V}_{\Pi}(x, B_x, \tau) = \text{ja}) = 1$, d.h. \mathbf{V}_{Π} kommt bei Eingabe von x , B_x und jeder Zufallsfolge τ auf den ja-Ausgang. Also akzeptiert $\tilde{\mathbf{V}}_{\Pi}$ die Eingabe x . Ist $x \notin L_{\Pi}$, so verwirft \mathbf{V}_{Π} für jeden Beweis B wegen $P(\mathbf{V}_{\Pi}(x, B, \tau) = \text{nein}) \geq 1/2$ die Eingabe für mindestens die Hälfte aller in Frage kommenden Zufallsfolgen, daher akzeptiert $\tilde{\mathbf{V}}_{\Pi}$ die Eingabe x nicht.

Daher ist $L \in \mathbf{NTIME}(q(n) \cdot 2^{O(r(n))})$.

///

Setzt man im speziellen für $r(n)$ und $q(n)$ Polynome ein, so gilt sogar:

Satz 7.2-5:

$$\mathbf{PCP}(\text{poly}(n), \text{poly}(n)) \subseteq \mathbf{NTIME}(2^{\text{poly}(n)})$$

Weiter gelten folgende Aussagen, die die Klassen **P** und **NP** im **PCP**-Modell beschreiben:

Satz 7.2-6:

(i) $\mathbf{PCP}(0, 0) = \mathbf{P}$

(ii) $\mathbf{PCP}(0, \text{poly}(n)) = \mathbf{NP}$

(iii) $\mathbf{PCP}(\log(n), \text{poly}(n)) = \mathbf{NP}$

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 $P(\mathbf{V}_{\Pi}(x, B, \tau) = \text{nein}) \geq 1/2$ nur $P(\mathbf{V}_{\Pi}(x, B, \tau) = \text{nein}) = 1$ möglich. Daher wird L_{Π} von \mathbf{V}_{Π} 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): $\mathbf{PCP}(0, \text{poly}(n))$ erlaubt beliebig lange Beweise, von denen aber nur polynomiell viele Bits gelesen werden. Da es keine Zufallsfolge gibt, sind dies 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 $P(\mathbf{V}_{\Pi}(x, B, \tau) = \text{nein}) \geq 1/2$ nur $P(\mathbf{V}_{\Pi}(x, B, \tau) = \text{nein}) = 1$ möglich. Daher ist $\mathbf{PCP}(0, \text{poly}(n)) \subseteq \mathbf{NP}$.

Die Umkehrung $\mathbf{NP} \subseteq \mathbf{PCP}(0, \text{poly}(n))$ ist wieder offensichtlich.

Zu (iii): Die Inklusion $\mathbf{PCP}(\log(n), \text{poly}(n)) \subseteq \mathbf{NP}$ folgt direkt aus Satz 7.2-4.

Die Beziehung $\mathbf{NP} \subseteq \mathbf{PCP}(\log(n), \text{poly}(n))$ folgt aus (ii).

///

Aus $\mathbf{PCP}(r(n), q(n)) \subseteq \mathbf{NTIME}(q(n) \cdot 2^{O(r(n))})$ folgt unmittelbar:

Satz 7.2-7:

$$\mathbf{PCP}(\log(n), 1) \subseteq \mathbf{NP}$$

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 \mathbf{NPO} geführt.

Satz 7.2-8:

$$\text{PCP}(\log(n), 1) = \text{NP}$$

„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 7.2-8 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 **P = NP**. In Kapitel 6.1 wird gezeigt, dass es in **APX** liegt.

3-SAT-Maximierungsproblem

- Instanz:
1. $I = (K, V)$
 $K = \{F_1, F_2, \dots, F_m\}$ ist eine Menge von Klauseln, die aus Booleschen Variablen aus der Menge $V = \{x_1, \dots, x_n\}$ gebildet werden und jeweils die Form $F_i = (y_{i_1} \vee y_{i_2} \vee y_{i_3})$ für $i = 1, \dots, m$ 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. $\text{SOL}(I) =$ Belegung der Variablen in V mit Wahrheitswerten **TRUE** oder **FALSE**, d.h. $\text{SOL}(I)$ ist eine Abbildung $f : V \rightarrow \{\text{TRUE}, \text{FALSE}\}$
 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 7.2-9:

Das 3-SAT-Maximierungsproblem liegt nicht in **PTAS**, d.h. es besitzt kein polynomiell zeitbeschränktes Approximationsschema, außer **P = NP**.

Beweis:

Es sei $L_{SAT} = \{F \mid F \text{ ist ein erfüllbarer Boolescher Ausdruck}\}$.

$L_{SAT} \subseteq \Sigma_{BOOLE}^* = \{F \mid F \text{ ist ein Boolescher Ausdruck}\}$, L_{SAT} ist **NP**-vollständig.

Es wird eine Abbildung h definiert, die jedem $F \in \Sigma_{BOOLE}^*$ 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 $F \in L_{SAT}$, dann sind alle Klauseln in K erfüllbar.

Ist $F \notin L_{SAT}$, 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 6.2-8 folgt, dass es keinen polynomiell zeitbeschränkten r -approximativen Algorithmus für das 3-SAT-Maximierungsproblem mit $r < 1/(1 - \varepsilon)$ gibt, außer **P = NP**. Daher besitzt das 3-SAT-Maximierungsproblem kein polynomiell zeitbeschränktes Approximationsschema.

Da L_{SAT} **NP**-vollständig ist, gibt es für L_{SAT} nach Satz 7.2-8 einen $(\log(n), 1)$ -beschränkten Verifizierer V_{SAT} . In 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 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. $V = \{x_1, x_2, x_3, \dots, x_k\}$ mit $k = q \cdot n^c$. V enthält polynomiell viele Variablen und ist in polynomieller Zeit aus F konstruierbar. Zu jedem Beweis $B = b_1 \dots b_k$ der Länge $k = q \cdot n^c$ lässt sich eine Belegung

$f_B : V \rightarrow \{\text{TRUE}, \text{FALSE}\}$ der Variablen in V durch

$$f_B(x_i) = \begin{cases} \text{TRUE} & \text{für } b_i = 1 \\ \text{FALSE} & \text{für } b_i = 0 \end{cases}$$

definieren.

Ist umgekehrt eine Belegung $f : V \rightarrow \{\text{TRUE}, \text{FALSE}\}$ der Variablen in V gegeben, so lässt sich ein Beweis $B_f = b_{f,1} \dots b_{f,k}$ angeben mit

$$b_{f,i} = \begin{cases} 1 & \text{für } f(x_i) = \text{TRUE} \\ 0 & \text{für } f(x_i) = \text{FALSE} \end{cases}$$

Für die Zufallsfolge τ seien die q Positionen, die in einem Beweis erfragt werden, die Positionen τ_1, \dots, τ_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 $|A_\tau| \leq |\Sigma_0|^q = 2^q$. Für jedes $(b_1, \dots, b_q) \in A_\tau$ wird eine Formel $(y_{\tau_1} \vee \dots \vee y_{\tau_q})$ 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 Formeln.

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 $(y_{\tau_1} \vee \dots \vee y_{\tau_q})$ bereits die gewünschte Form. Für $q > 3$ werden für jede Formel $G = (y_{\tau_1} \vee \dots \vee y_{\tau_q})$ $q - 3$ neue Variablen $z_{G,1}, \dots, z_{G,q-3}$ eingeführt und G durch $(y_{\tau_1} \vee y_{\tau_2} \vee z_{G,1})$, $(\neg z_{G,1} \vee y_{\tau_3} \vee z_{G,2})$, \dots , $(\neg z_{G,q-4} \vee y_{\tau_{q-2}} \vee z_{G,q-3})$, $(\neg z_{G,q-3} \vee y_{\tau_{q-1}} \vee y_{\tau_q})$ 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 $i = 1, \dots, q - 3$ mit FALSE belegt, und damit alle neuen Klauseln erfüllt. Ist G erfüllbar, wobei $y_{\tau_1}, \dots, y_{\tau_{j-1}}$ mit FALSE und y_{τ_j} für $j > 2$ mit TRUE belegt ist, dann werden $z_{G,1}, \dots, z_{G,j-2}$ mit TRUE und $z_{G,j-1}, \dots, z_{G,q-3}$ mit FALSE belegt; wieder sind alle neuen Klauseln erfüllt.

Sind umgekehrt alle neuen Klauseln erfüllt, dann können nicht alle y_{τ_i} für $i = 1, \dots, q$ mit FALSE belegt sein: Denn wären alle Klauseln $(y_{\tau_1} \vee y_{\tau_2} \vee z_{G,1})$, $(\neg z_{G,1} \vee y_{\tau_3} \vee z_{G,2})$, \dots , $(\neg z_{G,q-4} \vee y_{\tau_{q-2}} \vee z_{G,q-3})$, $(\neg z_{G,q-3} \vee y_{\tau_{q-1}} \vee y_{\tau_q})$ erfüllt und gleichzeitig alle y_{τ_i} gleich FALSE, so müssten $z_{G,1}, \dots, z_{G,q-3}$ 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 Klauselmengue bildet die Menge K . Es ist $|K| \leq (q-2) \cdot |K'|$, und auch V behält eine polynomielle Größe.

Die Berechnung von $h(F) = (K, V)$ erfolgt in polynomieller Zeit.

Ist $F \in L_{SAT}$, dann gibt es einen Beweis B_F , so dass der Verifizierer \mathbf{V}_{SAT} für jede Zufallsfolge τ die Entscheidung $\mathbf{V}_{SAT}(F, B_F, \tau) = \text{ja}$ trifft, denn $P(\mathbf{V}_{SAT}(F, B_F, \tau) = \text{ja}) = 1$. Ist für eine Zufallsfolge τ die definierte Menge $A_\tau = \emptyset$, so wurde keine Formel in K' bzw. K aufgenommen. Ist $A_\tau \neq \emptyset$ und sind τ_1, \dots, τ_q die Positionen in B_F , die von \mathbf{V}_{SAT} aufgrund der Zufallsfolge τ erfragt werden, etwa mit den Werten a_1, \dots, a_q , dann wird durch die Festlegung

$$x_{\tau_i} = \begin{cases} \text{TRUE} & \text{für } a_i = 1 \\ \text{FALSE} & \text{für } a_i = 0 \end{cases}$$

eine partielle Belegung der Variablen in V definiert.

Es sei $(b_1, \dots, b_q) \in A_\tau$ mit der dazu gebildeten Formel $(y_{\tau_1} \vee \dots \vee y_{\tau_q}) \in K'$. Dann ist $(a_1, \dots, a_q) \neq (b_1, \dots, b_q)$, denn (a_1, \dots, a_q) führt auf eine ja-Entscheidung, und alle $(b_1, \dots, b_q) \in A_\tau$ 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 $(y_{\tau_1} \vee \dots \vee y_{\tau_q})$ 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 $F \notin L_{SAT}$, 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 $\mathbf{V}_{SAT}(F, B, \tau) = \text{nein}$ trifft, denn für jeden Beweis B ist in diesem Fall $P(\mathbf{V}_{SAT}(F, B, \tau) = \text{nein}) \geq 1/2$. Es sei eine Belegung $f: V \rightarrow \{\text{TRUE}, \text{FALSE}\}$ der Variablen in V gegeben. Der zu f konstruierbare Beweis $B_f = b_{f,1} \dots b_{f,k}$ (siehe oben) ist bestimmt durch

$$b_{f,i} = \begin{cases} 1 & \text{für } f(x_i) = \text{TRUE} \\ 0 & \text{für } f(x_i) = \text{FALSE} \end{cases}$$

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, \dots, τ_q , die Werte an diesen Positionen in B_f seien a_1, \dots, a_q . Dann ist nach Definition $(a_1, \dots, a_q) \in A_\tau$. Die aus (a_1, \dots, a_q) gebildete Formel $(y_{\tau_1} \vee \dots \vee y_{\tau_q})$ 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 $(y_{\tau_1} \vee \dots \vee y_{\tau_q})$ 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 $z_{G,1}, \dots, z_{G,q-3}$ 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

$$2^{c \cdot \log(n)} / (2 \cdot |K|) \geq 2^{c \cdot \log(n)} / (2 \cdot (q-2) \cdot 2^{q+c \cdot \log(n)}) = 1 / ((q-2) \cdot 2^{q+1}).$$

Mit $\varepsilon = 1 / ((q-2) \cdot 2^{q+1})$ folgt die Behauptung.

///

Stichwortverzeichnis

- { 0, 1 }-LP 226, 229
- 0/1-Rucksack-Entscheidungsproblem 222
- 0/1-Rucksack-Maximierungsproblem
 - Bestimmung einer optimalen Lösung 309
 - FPTAS 307
 - nicht absolut approximierbar 274
 - relativ approximierbar 281
 - RUCK_APP 282, 284
 - Verallgemeinerung 282, 285
- 3-CSAT 239, 241, 247, 250, 251
- 3-SAT-Maximierungsproblem 340
 - Approximation 294
- abgeschlossen bzgl. einer Operation 54
- ABS** 268
- Algorithmus
 - Beweis 103
 - Effizienz 197
 - intuitiver Algorithmusbegriff 32
 - Korrektheit 33
 - Las-Vegas-Verfahren 320
 - leicht durchführbar 196
 - Monte-Carlo-Verfahren 320
 - nichtdeterministisch 104, 210
 - obere Komplexitätsschranke 33
 - partielle Korrektheit 33
 - polynomielle Laufzeit 196
 - probabilistisch 319
 - Problemgröße 198
 - randomisiert 319, 328
 - schwer durchführbar 196
 - Speicherplatzkomplexität 33
 - untere Komplexitätsschranke 33
 - Verifizierer 103
 - Zeitkomplexität 33
 - Zeitkomplexität im Mittel 33
 - Zeitkomplexität im schlechtesten Fall 33
 - Zertifikat 103
- Allgemeines Rucksackproblem *Siehe* 0/1-Rucksack-Maximierungsproblem, Verallgemeinerung
- Alphabet 12
- Approximation
 - ABS** 268, 297
 - absolut approximierbar 268
 - absoluter Approximationsalgorithmus 268
 - absoluter Fehler 268
 - Approximationsalgorithmus 268
 - APX 277, 297
 - asymptotisch r -approximativer Algorithmus 298
 - asymptotisches Approximationsschema 313
 - FPTAS** 307
 - polynomiell zeitbeschränktes Approximationsschema 303
 - PTAS** 303
 - PTAS**^o 313
 - r -approximativer Algorithmus 277
 - relative Approximationsgüte 276, 298
 - voll polynomiell zeitbeschränktes Approximationsschema 307
- Arithmetische Operationen
 - Addition 71, 73
 - Division 74
 - Exponentiation 77
 - Karatsuba-Methode 72
 - Modulo 76
 - Multiplikation 72
- Aussagenlogik
 - Antivalenz 9
 - Äquivalenz 9
 - Assoziativgesetz 10
 - de Morgan 11
 - Disjunktion 9
 - Distributivgesetz 10
 - Implikation 9
 - Junktor 8
 - Kommutativgesetz 10
 - Konjunktion 9
 - logischer Junktor 8
 - Negation 9
- Automatentheorie 134
- Binärbaum
 - Blatt 24
 - Höhe 26
 - leer 24
 - Niveau 26
 - Pfad 24
 - Pfadlänge 24
 - Rang eines Knotens 24
 - Wurzel 24
- binärer Suchbaum 320
- BINPACKING 223, 246

- Binpacking-Entscheidungsproblem 223
- Binpacking-Minimierungsproblem
 - asymptotisches **PTAS** 313
 - BestfitDecreasing-Algorithmus 302, 317
 - Firstfit-Algorithmus 299
 - FirstfitDecreasing-Algorithmus 300, 317
 - Grenze der relativen Approximation 292
 - Nextfit-Algorithmus 279, 317
 - probabilistisches
 - Approximationsschema 317
 - zulässige Lösung 278
- Boolesche Formel *Siehe* Boolescher Ausdruck
- Boolescher Ausdruck
 - Bindungsstärke der Junktoren 21
 - Disjunktion 21
 - erfüllbar 23
 - Exklusiv-Oder 23
 - Junktor 21
 - Klausel 23
 - Konjunktion 21
 - konjunktive Normalform 23
 - Konstante 21
 - Literal 22
 - Negation 21
 - Variable 21
 - Wahrheitswert 22
- Chomskyhierarchie 134
- Christofides' Algorithmus 288
- Churchsche These 35
- co-NP** 261
- $co-TIME(f(n))$ 258
- CSAT 229, 235
- Derandomisierung 295
- Diagonalisierung 17, 33, 113
- Endlicher Automat
 - Anfangszustand 178
 - deterministisch 178
 - Eingabealphabet 178
 - Endzustand 178
 - Konfiguration 178
 - nichtdeterministisch 178
 - Startzustand 178
 - Transitionsdiagramm 179
 - Überföhrungsfunktion 178
 - Zustandsmenge 178
- Erfüllbarkeitsproblem
 - 2-CSAT 217
 - 3-CSAT 220, 221
 - allgemeine Form 221
 - allgemeine Form 216
 - CSAT 215, 221
 - deterministischer Lösungsalgorithmus 106
 - erfüllende Wahrheitsbelegung 215
 - konjunktive Normalform 215
 - konjunktive Normalform 221
 - konjunktive Normalform mit genau 2 Literalen 217
 - konjunktive Normalform mit genau 3 Literalen 220, 221
 - nichtdeterministischer Lösungsalgorithmus 106
 - SAT 216, 221
- EXP** 209
- FÄRBBARKEIT 225, 247
- Färbbarkeits-Minimierungsproblem
 - COL 269
 - ungerichteter Graph 268
 - ungerichteter planarer Graph 271
 - zulässige Knotenfärbung 268
- formale Sprache
 - Ableitungsregel 134
 - Abschluss 13
 - Abschlusseigenschaften 191
 - Beschreibungsmittel 190
 - Chomsky-Normalform 156
 - deterministisch kontextfreie Sprache 172
 - erzeugte Sprache 135
 - Erzeugung 135
 - Erzeugungsregel 134
 - Grammatik 134
 - kontextfreie Grammatik 151
 - kontextfreie Sprache 151
 - kontextsensitive Sprache 139
 - kontextsensitive Grammatik 139
 - Leerheitsproblem 137, 195
 - Leerheitsproblem für Typ-0-Grammatiken 138
 - Leerheitsproblem für Typ-1-Grammatiken 150
 - Leerheitsproblem für Typ-2-Grammatiken 162
 - Leerheitsproblem für Typ-3-Grammatiken 188
 - $LR(k)$ -Sprache 174
 - nichtdeterministisch vs. deterministisch 191

- Nichtterminalsymbol 134
- Normalform 156
- positiver Abschluss 13
- Produkt 13
- Produktion 134
- pumping lemma 157
- Rechenregeln 14
- Rechtsableitung 174
- rechtslineare Sprache 174
- reguläre Sprache 174
- regulärer Ausdruck 175
- Startsymbolsymbol 134
- Terminalsymbol 134
- Typ-0-Grammatik 137
- Typ-0-Sprache 137
- Typ-1-Grammatik 139
- Typ-1-Sprache 139
- Typ-2-Grammatik 151
- Typ-2-Sprache 151
- Typ-3-Grammatik 174
- Typ-3-Sprache 174
- $uvwxy$ -Theorem 157
- Variable 134
- VERIFIZIERE_G* 136
- VERIFIZIERE_G_Typ_0* 138
- VERIFIZIERE_G_Typ_1* 150
- VERIFIZIERE_G_Typ_2* 161
- VERIFIZIERE_G_TYP_3* 187
- Wortproblem 136, 195
- Wortproblem für Typ-0-Grammatiken 138
- Wortproblem für Typ-1-Grammatiken 150
- Wortproblem für Typ-2-Grammatiken 161
- Wortproblem für Typ-3-Grammatiken 187
- FPTAS** 307
- Funktion
 - bijektiv 12
 - Definitionsbereich 12
 - injektiv 12
 - linkseindeutig 12
 - linkstotal 12
 - Ordnung 18
 - partiell 12
 - platzkonstrierbar 100
 - rechtseindeutig 12
 - rechtstotal 12
 - speicherplatzkonstrierbar 100
 - surjektiv 12
 - Umkehrfunktion 12
 - Wachstumsverhalten 21
 - Wertebereich 12
 - zeitkonstrierbar 94
- Graph
 - Anfangsknoten 24
 - Endknoten 24
 - gerichtet 24
 - Grad eines Graphen 269
 - Grad eines Knotens 269
 - Hamiltonkreis 287
 - Kanten 24
 - Knoten 24
 - Nachfolger 24
 - Pfad 24
 - planar 271
 - ungerichtet 24
 - Vier-Farben-Satz 273
 - Vorgänger 24
- Graphenisomorphieproblem 261
- Halteproblem
 - mit leerem Eingabeband 121
 - zugehörige Sprache 119
- HAMILTONKREIS
 - GERICHTET 224, 251, 254
 - UNGERICHTET 224, 254, 255
- HANDLUNGSREISENDER 224, 255
- Kellerautomat
 - ε -Überführung 164
 - akzeptierte Sprache 166, 167
 - Anfangskonfiguration 166
 - Anfangszustand 165
 - deterministisch 172
 - Eingabealphabet 165
 - Endkonfiguration 166
 - Endzustand 165
 - Kelleralphabet 165
 - Konfiguration 166
 - Nachfolgekonfiguration 166
 - nichtdeterministisch 164, 165
 - Startsymbol im Keller 165
 - Startzustand 165
 - Überföhrungsfunktion 165
 - Zustandsmenge 165
- KLIQUE 222, 250
- Kliquenproblem 222
- Komplexitätsklasse
 - co-NP** 261
 - deterministisch 208
 - EXP** 209
 - nichtdeterministisch 211

- NP** 211, 212, 214
- NPC** 260
- NPI** 260
- NPO** 265
- NPSpace** 212
- $NTIME(f(n))$ 210
- P** 208, 212, 214
- PCP** 336
- P-NP-Problem** 211
- PO** 267
- PSPACE** 209
- RP** 331
- RP**($\epsilon(n)$) 334
- RP*** 334
- $SPACE(f(n))$ 208
- Speicherplatzkomplexität 208
- $TIME(f(n))$ 208
- Zeitkomplexität 208
- Konkatinaton *Siehe* Wort
- LÄNGSTER WEG 226
- Las-Vegas-Verfahren *Siehe* Algorithmus
- LBA-Problem 149
- Leerheitsproblem *Siehe* formale Sprache
- Linear beschränkter Automat *Siehe*
 - Turingmaschine
- Menge *Siehe* Mengenlehre
- Mengenlehre
 - abzählbar 16
 - Assoziativgesetz 8
 - de Morgan 8
 - disjunkte Zerlegung 8
 - elementare 7
 - endlich 16
 - entscheidbar 115
 - gleichmächtig 15
 - Kommutativgesetz 8
 - reduzierbar 117
 - rekursiv *Siehe* entscheidbar
 - rekursiv aufzählbar 110
 - überabzählbar 16
 - unendlich 16
- Monte-Carlo-Verfahren *Siehe*
 - Algorithmus
- Näherungsalgorithmus *Siehe*
 - Approximation
- natürliche Zahlen 12
- NP**
 - Abschlusseigenschaften 256
- NPC** 260
- NPI** 260
- NPO** 265, 316
- NPSpace** 212
- NP-unvollständig** 260
- NP-vollständig** 214, 231, 239
 - CSAT 235
 - SAT 231
- Ordnung einer Funktion *Siehe* Funktion
- P** 208
 - Abschlusseigenschaften 256
- Packungsproblem 223
- PARTITION 223, 244, 246
- Partitionen-Minimierungsproblem
 - PTAS 304
- Partitionenproblem 223
 - Entscheidungsproblem 90
 - ganzzahlig 90, 104
- Pascal
 - Object Pascal 152
- PCP** 336
- P-NP-Problem** 211
- PO** 267
- polynomiell reduzierbar 228, 229
- Prädikatenlogik
 - Quantoren 11
- Primzahlen 161
 - linear beschränkter Automat 145
- Primzahltest 320, 328
 - Carmichael-Zahlen 324
 - randomisiert 322
 - streng pseudoprim 326
 - Zeuge 321
- Problem
 - Berechnungsproblem 31
 - Eingabeinstanz 27
 - Entscheidungsproblem 30
 - Instanz 27
 - optimale Lösung 265
 - Optimierungsproblem 31, 265
 - polynomiell beschränkt 311
 - Problemgröße 198
 - Problemparameter 27
 - Problemtypen 28, 30
 - Wert einer zulässigen Lösung 31
 - Zielfunktion 31
 - zugehöriges Entscheidungsproblem
 - 202, 266
 - zulässige Lösung 31
- Problem der $\{0, 1\}$ -Linearen
 - Programmierung 226
- Problem der Knotenfärbbarkeit 225
- Problem des Hamiltonschen Kreises 224
- Problem des Handlungsreisenden

- Berechnungsproblem 29
- Entscheidungsproblem 29, 224
- metrisch 288
- mit ganzzahligen Gewichten 204
- nicht relativ approximierbar 286
- Optimierungsproblem 28
- Tour 28
- Problem des längsten Wegs 226
- Programmiersprache
 - arithmetische Operationen 70
 - Größe der Eingabe 69
 - Konzept 67
 - Speicherplatzkomplexität 69, 70
 - Zeitkomplexität 69, 70
- Prozedur *Siehe* Algorithmus
- PSPACE** 209
- PTAS** 303, 340
- PTAS[∞]** 313
- Quadratzahlen
 - linear beschränkter Automat 147
 - Typ-1-Sprache 141
- Quicksort 320, 328
- Random Access Maschine
 - Akkumulator 57
 - Akzeptor 61
 - Anweisung 58
 - Ausgabeband 57
 - Befehl 58
 - Befehlsvorrat 58
 - Befehlszähler 58
 - Eingabeband 57
 - Komplexität 62
 - Konzept 57
 - Kosten einer Operation 63
 - Länge einer ganzen Zahl 63
 - logarithmisches Kostenkriterium 63
 - Operand 58
 - Operationscode 58
 - Programm 57
 - Register 57
 - Simulation einer Turingmaschine 66
 - Speicher 57
 - Speicherabbildungsfunktion 58
 - Speicherplatzkomplexität 62
 - symbolische Marken 58
 - uniformes Kostenkriterium 63
 - Zeitkomplexität 62
- reduzierbar 117
- RP** 331
- RP($\varepsilon(n)$)** 334
- RP*** 334
- RUCKSACK 222, 241, 244
- SAT 231, 239
- Sprache *Siehe* formale Sprache
- Traveling Salesman Problem *Siehe* Problem des Handlungsreisenden
- Turingmaschine
 - RP($\varepsilon(n)$)-Akzeptor** 335
 - Ablehnungswahrscheinlichkeit 330
 - Akzeptanzwahrscheinlichkeit 330
 - akzeptierende Konfiguration 39, 89
 - akzeptierender Zustand 36, 46, 88
 - akzeptiert 39
 - akzeptierte Sprache 39, 89
 - algorithmisch äquivalent 57
 - Anfangskonfiguration 38, 88
 - Anfangszustand 36, 88
 - Arbeitsalphabet 36, 88
 - Arbeitsweise 36
 - Ausgabeband 42
 - Band 36
 - Berechnung 89
 - Berechnungsbaum 89, 329
 - Berechnungsvorschrift 42
 - beschränkter Verifizierer 336
 - Beweis 99
 - Blankzeichen 36, 88
 - deterministisch 36
 - deterministischer linear beschränkter Automat 149
 - Eingabe wird akzeptiert 47
 - Eingabe wird verworfen 47
 - Eingabealphabet 36, 88
 - Eingabeband 42
 - Endkonfiguration 39, 89
 - Endlicher Automat *Siehe* Endlicher Automat
 - Endzustand 36, 88
 - Folgekonfiguration 38, 88
 - GENERIERE_TM** 83
 - Hintereinanderschaltung 48
 - i*-te Turingmaschine 83
 - Kellerautomat *Siehe* Kellerautomat
 - kodierte Turingmaschine 83
 - Kodierung 79
 - Kodierung der *i*-ten Turingmaschine 83
 - Konfiguration 37, 88
 - Konzept 36
 - leere Zelle 36
 - Leerzeichen 36, 88
 - Linear beschränkter Automat 143
 - nicht-akzeptierender Zustand 46

- nichtdeterministisch 88
- nichtdeterministischer linear
 - beschränkter Automat 149
- nichttriviales Entscheidungsproblem 127
- ORDNUNG_TM* 84
- Parallelschaltung 50
- Platzkomplexität 54, 94
- probabilistisch 328
- randomisiert 328
- raten (Nichtdeterminismus) 99
- RP**($\varepsilon(n)$)-Akzeptor 334
- RP**-Akzeptor 331
- RV-Modell 98
- Schreib/Lesekopf 36
- Speicherplatzkomplexität 54, 94
- Startzustand 36, 88
- Steuerwerk 36
- stoppt 37
- stoppt nicht 37
- Überföhrungsfunktion 36, 88
- universelle 79, 85
- Unterprogramm 53
- Varianten 56
- Verhalten bei einer Eingabe 47
- Verifikationsphase 100
- VERIFIZIERE_TM* 82
- verwerfender Zustand 46
- zeitbeschränkt 54, 93
- Zeitkomplexität 54, 93, 316
- Zeitkomplexität im Mittel 316
- Zelle 36
- Zertifikat 99
- Zusammensetzung 47
- Zustandsmenge 36, 88
- Universelle Turingmaschine *Siehe* Turingmaschine
- Wort
 - Buchstaben 12
 - formale Definition 13
 - Konkatination 13
 - Länge 13
 - leer 13
 - lexikalische Reihenfolge 16
 - lexikographische Reihenfolge 16
- Wortproblem *Siehe* formale Sprache
- Zeichenkette *Siehe* Wort
- Zweierpotenzen 161
 - linear beschränkter Automat 146
- Typ-1-Sprache 140