- Problem des längsten Wegs in einem gewichteten Graphen:
  Instanz: $\lfloor G, v_i, v_j, K \rfloor$, $G = (V, E, w)$ ist ein gewichteter (gerichteter oder ungerichteter) Graph mit der Knotenmenge $V = \{v_1, ..., v_n\}$, bestehend aus n Knoten, und der Kantenmenge $E \subseteq V \times V$; $v_i \in V; v_j \in V$; die Funktion $w: E \to \mathbf{R}_{\geq 0}$ gibt jeder Kante $e \in E$ ein Gewicht; $K \in \mathbf{R}_{\geq 0}$; $size([G, K]) = k = n \cdot B$ mit $B = \max\{\lceil \log(w(e)) \rceil \mid e \in E\}$
  Gesucht: ein einfacher Weg (d.h. ohne Knotenwiederholungen) von v_i nach v_j mit Gewicht $\geq K$.
  Beweis: $B_{[G, v_i, v_j, K]}$ ist eine Folge $i_1, i_2, ..., i_k$ von natürlichen Zahlen aus $\{1, ..., n\}$ (mit Länge in $O(n \cdot \log(n))$
  Arbeitsweise des Verifizierers: Überprüfung, ob $v_{i_1}, v_{i_2}, ..., v_{i_k}$ einen einfachen Weg (d.h. ohne Knotenwiederholungen) von v_{i_1} nach v_{i_k} mit Gewicht $\geq K$ beschreibt. In diesem Fall wird `ja` ausgegeben, ansonsten `nein`.

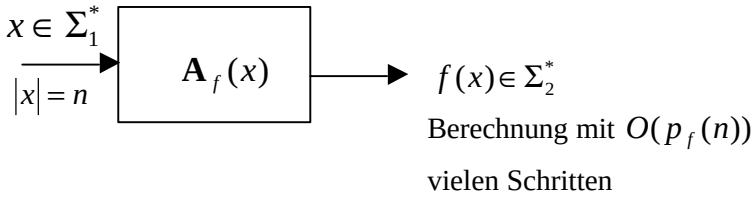
Man kennt heute mehrere tausend Probleme aus **NP**. In der angegebenen Literatur werden geordnet nach Anwendungsgebieten umfangreiche Listen von **NP**-Problemen aufgeführt.

5.4 NP-Vollständigkeit

Die bisher beschriebenen Probleme entstammen verschiedenen Anwendungsgebieten. Dementsprechend unterscheiden sich die Alphabete, mit denen man die jeweiligen Instanzen bildet. Boolesche Ausdrücke werden mit einem anderen Alphabet kodiert als gewichtete Graphen oder Instanzen für das Partitionenproblem. Selbstverständlich lassen sich letztlich alle Probleme mit Hilfe des Alphabets $\{0, 1\}$ kodieren, so daß man mit einem einzigen Alphabet auskommen könnte. Aber selbst, wenn die Eingabeinstanzen unterschiedlicher Probleme mit dem Alphabet $\{0, 1\}$ kodiert werden, weisen die dann so kodierten Bestandteile der betrachteten Objekte wie Graphen, Boolesche Variablen oder Zahlen immer noch grundlegend verschiedene Eigenschaften auf, so daß man weiterhin davon ausgehen kann, daß man unterschiedliche Anwendungen mit Hilfe unterschiedlicher Alphabete kodiert. Im folgenden wird eine Verbindung zwischen den unterschiedlichen Problemen und ihren zugrundeliegenden Alphabeten hergestellt.

Eine Funktion $f: \Sigma_1^* \to \Sigma_2^*$, die Wörter über dem endlichen Alphabet Σ_1 auf Wörter über dem endlichen Alphabet Σ_2 abbildet, heißt **durch einen deterministischen Algorithmus in**

polynomieller Zeit berechenbar, wenn gilt: Es gibt einen deterministischen Algorithmus $\mathbf{A}_f$ mit Eingabemenge Σ_1^* und Ausgabemenge Σ_2^* und ein Polynom p_f mit folgenden Eigenschaften:
bei Eingabe von $x \in \Sigma_1^*$ mit der Größe $|x| = n$ erzeugt der Algorithmus die Ausgabe $f(x) \in \Sigma_2^*$ und benötigt dazu höchstens $O(p_f(n))$ viele Schritte.



Es seien $L_1 \subseteq \Sigma_1^*$ und $L_2 \subseteq \Sigma_2^*$ zwei Mengen aus Zeichenketten (Wörtern) über jeweils zwei endlichen Alphabeten. L_1 heißt **polynomiell (many-one) reduzierbar** auf L_2, geschrieben
$L_1 \leq_m^p L_2$,
wenn gilt: Es gibt eine Funktion $f : \Sigma_1^* \to \Sigma_2^*$, die durch einen deterministischen Algorithmus in polynomieller Zeit berechenbar ist und für die gilt:
$x \in L_1 \Leftrightarrow f(x) \in L_2$.

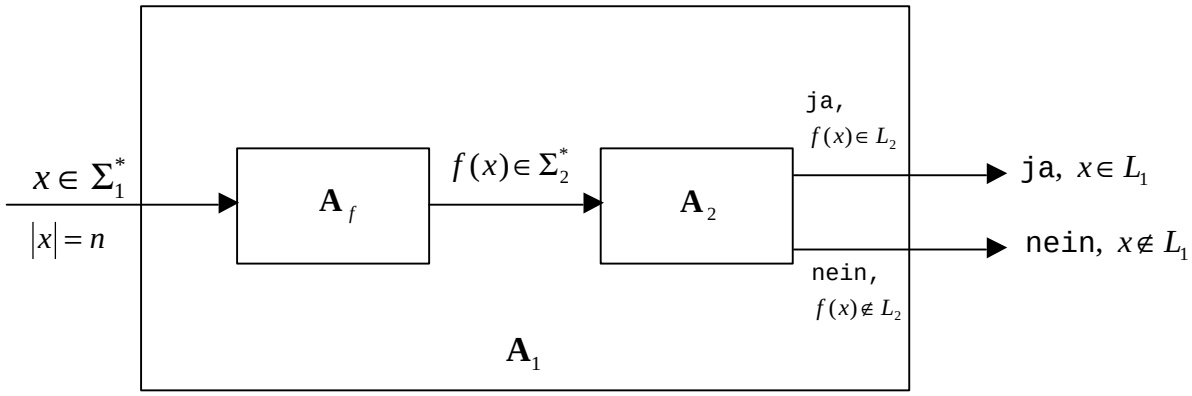
Diese Eigenschaft kann auch so formuliert werden:
$x \in L_1 \Rightarrow f(x) \in L_2$ und $x \notin L_1 \Rightarrow f(x) \notin L_2$.

Bemerkung: Es gibt noch andere Formen der Reduzierbarkeit zwischen Mengen, z.B. die allgemeinere Form der Turing-Reduzierbarkeit mit Hilfe von Orakel-Turingmaschinen.

Die **Bedeutung der Reduzierbarkeit** $\leq_m^p$ zeigt folgende Überlegung.

Für die Mengen $L_1 \subseteq \Sigma_1^*$ und $L_2 \subseteq \Sigma_2^*$ gelte $L_1 \leq_m^p L_2$ mittels der Funktion $f : \Sigma_1^* \to \Sigma_2^*$ bzw. des Algorithmus $\mathbf{A}_f$. Seine Zeitkomplexität sei das Polynom p_f. Für die Menge L_2 gebe es einen polynomiell zeitbeschränkten deterministischen Algorithmus $\mathbf{A}_2$, der L_2 erkennt; seine Zeitkomplexität sei das Polynom p_2 :

Mit Hilfe von $\mathbf{A}_f$ und $\mathbf{A}_2$ läßt sich durch Hintereinanderschaltung beider Algorithmen ein polynomiell zeitbeschränkter deterministischer Algorithmus $\mathbf{A}_1$ konstruieren, der L_1 erkennt:



Erkennung von L_1 mit Zeitaufwand der Größenordnung $O\left(p_f(n) + p_2\left(p_f(n)\right)\right)$, d.h. polynomiellem Zeitaufwand

Ein $x_2 \in \Sigma_2^*$ kann dazu dienen, für mehrere (*many*) $x \in \Sigma_1^*$ die Frage „$x \in L_1$?“ zu entscheiden (nämlich für alle diejenigen $x \in \Sigma_1^*$, für die $f(x) = x_2$ gilt). Allerdings darf man für jede Eingabe $f(x)$ den Algorithmus $\mathbf{A}_2$ nur einmal (*one*) verwenden, nämlich bei der Entscheidung von $f(x)$.

Gilt für ein Problem Π_0 zur Entscheidung der Menge $L_{\Pi_0} \subseteq \Sigma_{\Pi_0}^*$ und für ein Problem Π zur Entscheidung der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ die Relation $L_\Pi \leq_m^p L_{\Pi_0}$, so heißt das Problem Π auf das Problem Π_0 **polynomiell (many-one) reduzierbar**, geschrieben $\Pi \leq_m^p \Pi_0$.

Beispiel:

$\{0,1\}$-Lineare Programmierung:

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

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

Lösung: Entscheidung „ja“, falls gilt:
Es gibt eine Zuweisung der Werte 0 oder 1 an die Variablen in $\vec{z}$, so daß das lineare Ungleichungssystem $A \cdot \vec{z} \leq / \geq \vec{b}$ erfüllt ist. Die Bezeichnung $\leq / \geq$ steht hier für entweder $\leq$ oder $\geq$ in einer Ungleichung.

Es gilt CSAT $\leq_m^p$ $\{0,1\}$-Lineare Programmierung.

Einige leicht nachzuweisende Eigenschaften der polynomiellen many-one-Reduzierbarkeit faßt der nächste Satz zusammen.

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:

1. $L \leq_m^p L$, d.h. die Relation $\leq_m^p$ ist reflexiv.
2. Aus $L_1 \leq_m^p L_2$ und $L_2 \leq_m^p L_3$ folgt $L_1 \leq_m^p L_3$, d.h. die Relation $\leq_m^p$ ist transitiv.
3. Gilt $L_1 \leq_m^p L_2$ und $L_2 \in \mathbf{P}$, dann ist $L_1 \in \mathbf{P}$.
4. Gilt $L_1 \leq_m^p L_2$ und $L_2 \in \mathbf{NP}$, dann ist $L_1 \in \mathbf{NP}$.

Eine der zentralen Definitionen in der Angewandten Komplexitätstheorie ist die **NP**-Vollständigkeit:

Ein Entscheidungsproblem Π_0 zur Entscheidung (Akzeptanz, Erkennung) der Menge $L_{\Pi_0} \subseteq \Sigma_{\Pi_0}^*$ heißt **NP-vollständig**, wenn gilt:
$L_{\Pi_0} \in \mathbf{NP}$, und für jedes Probleme Π zur Entscheidung (Akzeptanz, Erkennung) der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ mit $L_\Pi \in \mathbf{NP}$ gilt $L_\Pi \leq_m^p L_{\Pi_0}$.

Mit obiger Definition der Reduzierbarkeit zwischen Problemen kann man auch sagen:

Das Entscheidungsproblem Π_0 ist **NP**-vollständig, wenn Π_0 in **NP** liegt und für jedes Entscheidungsprobleme Π in **NP** die Relation $\Pi \leq_m^p \Pi_0$ gilt.

Das erste Beispiel eines **NP**-vollständigen Problems wurde 1971 von Stephen Cook gefunden. Es gilt:

Satz 5.4-2:

Das Erfüllbarkeitsproblem für Boolesche Ausdrücke in allgemeiner Form (SAT) ist **NP**-vollständig.

Beweis:

Da es sich bei dieser Aussage um den zentralen Satz der angewandten Komplexitätstheorie handelt, soll die Beweisidee skizziert werden:

1. Die zu SAT gehörige Sprache ist $L_{\text{SAT}} = \{F \mid F \text{ ist ein erfüllbarer Boolescher Ausdruck }\}$. Bezeichnet Σ^*_{BOOLE} die Menge $\{F \mid F$ ist ein Boolescher Ausdruck$\}$, dann ist $L_{\text{SAT}} \subseteq \Sigma^*_{\text{BOOLE}}$. In Kapitel 5.3 wird gezeigt, daß SAT in **NP** liegt.
2. Für jedes Problem Π mit der Menge $L_\Pi \subseteq \Sigma^*_\Pi$ in **NP** ist die Relation $L_\Pi \leq^p_m L_{\text{SAT}}$ zu zeigen. Die einzige Eigenschaft, die bezüglich L_Π in einem Beweis genutzt werden kann, ist die Tatsache, daß es eine 1-NDTM TM_{L_Π} gibt, die bei Eingabe eines Wortes $x \in \Sigma^*_\Pi$ entscheidet, ob $x \in L_\Pi$ gilt oder nicht. Diese Entscheidung wird in $p_{L_\Pi}(|x|)$ vielen Schritten getroffen, wobei p_{L_Π} ein Polynom mit $p_{L_\Pi}(n) \geq n$ ist. Ist $x \in L_\Pi$, dann stoppt TM_{L_Π} im akzeptierenden Zustand q_{accept}. Ist $x \notin L_\Pi$, dann stoppt TM_{L_Π} in einem Zustand $q \neq q_{accept}$. TM_{L_Π} besucht dabei höchstens die Zellen mit den Nummern $-p_{L_\Pi}(|x|)$, ..., 0, 1, ..., $p_{L_\Pi}(|x|)+1$; hierbei wird das Nichtstandardmodell einer nichtdeterministischen Turingmaschine genommen (vgl. Kapitel 5.3).
   Zum Nachweis von $L_\Pi \leq^p_m L_{\text{SAT}}$ ist eine Funktion $f_\Pi : \Sigma^*_\Pi \to \Sigma^*_{\text{BOOLE}}$ anzugeben, die die Eigenschaft „$x \in L_\Pi \Leftrightarrow f_\Pi(x)$ ist erfüllbar“ besitzt ($f_\Pi(x)$ ist ein Boolescher Ausdruck); außerdem muß $f_\Pi(x)$ in $p_{f_\Pi}(|x|)$ vielen Schritten deterministisch berechenbar (konstruierbar) sein mit einem Polynom p_{f_Π}. Im folgenden wird beschrieben, wie $f_\Pi(x)$ aus x erzeugt werden kann. Der Boolesche Ausdruck $f_\Pi(x)$ beschreibt im wesentlichen, wie eine Berechnung von TM_{L_Π} bei Eingabe von $x \in \Sigma^*_\Pi$ abläuft.

   Die Zustandsmenge von TM_{L_Π} sei $Q = \{q_0, ..., q_k\}$, das Arbeitsalphabet von TM_{L_Π} sei $\Sigma_\Pi = \{a_0, ..., a_l\}$. Auf dem Eingabeband stehe das Wort $x \in \Sigma^*_\Pi$, $x = x_1 ... x_n$ mit $x_i \in \Sigma_\Pi$ für i = 1, ..., n. Es wird eine Reihe Boolescher Variablen erzeugt, deren Interpretation folgender Tabelle zu entnehmen ist:

Variable	Indizes	Interpretation
$zust_{t,q}$	$t = 0, ..., p_{L_\Pi}(n)$ $q \in Q$	$zust_{t,q} = \mathsf{TRUE}$ genau dann, wenn sich TM_{L_Π} im Schritt t im Zustand q befindet Anzahl an Variablen $zust_{t,q}$: $(k+1)\cdot(p_{L_\Pi}(n)+1)$
$pos_{t,i}$	$t = 0, ..., p_{L_\Pi}(n)$ $i = -p_{L_\Pi}(n), ..., p_{L_\Pi}(n)+1$	$pos_{t,i} = \mathsf{TRUE}$ genau dann, wenn sich der Schreib/Lesekopf von TM_{L_Π} im Schritt t über der Zelle mit Nummer i befindet Anzahl an Variablen $pos_{t,i}$: $2\cdot(p_{L_\Pi}(n)+1)^2$
$band_{t,i,a}$	$t = 0, ..., p_{L_\Pi}(n)$ $i = -p_{L_\Pi}(n), ..., p_{L_\Pi}(n)+1,$ $a \in \Sigma_\Pi$	$band_{t,i,a} = \mathsf{TRUE}$ genau dann, wenn sich im Schritt t in der Zelle mit Nummer i das Zeichen a befindet Anzahl an Variablen $band_{t,i,a}$: $2(l+1)(p_{L_\Pi}(n)+1)^2$

Der zu konstruierende Boolesche Ausdruck $f_\Pi(x)$ besteht aus mehreren Teilen und enthält insbesondere mehrmals eine Teilformeln *G*, die genau dann den Wahrheitswert TRUE erhält, wenn genau eine der in *G* vorkommenden Variablen den Wahrheitswert TRUE trägt. Sind $y_1, ..., y_m$ die in *G* vorkommenden Variablen, so lautet *G*:

$$G = G(y_1, ..., y_m) = \left(\bigvee_{i=1}^{m} y_i\right) \wedge \left(\bigwedge_{j=1}^{m-1} \bigwedge_{i=j+1}^{m} \neg(y_j \wedge y_i)\right)$$
$$= \left(\bigvee_{i=1}^{m} y_i\right) \wedge \left(\bigwedge_{j=1}^{m-1} \bigwedge_{i=j+1}^{m} (\neg y_j \vee \neg y_i)\right).$$

Die zweite Zeile zeigt, daß $G = G(y_1, ..., y_m)$ in konjunktiver Normalform formulierbar ist, ohne die Anzahl an Literalen zu ändern, und m^2 viele Literale enthält.

In $f_\Pi(x)$ kommen mehrere Teilformeln, die gemäß *G* aufgebaut sind, mit unterschiedlichen Variablen aus obiger Tabelle vor.

$f_\Pi(x)$ hat die Bauart $f_\Pi(x) = R \wedge A \wedge \ddot{U}_1 \wedge \ddot{U}_2 \wedge E$. Die Teilformel *R* beschreibt Randbedingungen, *A* Anfangsbedingungen, $\ddot{U}_1$ und $\ddot{U}_2$ beschreiben Übergangsbedingungen, und *E* beschreibt Endebedingungen.

In *R* wird ausgedrückt, daß TM_{L_Π} zu jedem Zeitpunkt *t* in genau einem Zustand *q* ist, daß der Schreib/Lesekopf über genau einer Zelle mit einer Nummer *i* aus dem Intervall von $-p_{L_\Pi}(n)$ bis $p_{L_\Pi}(n)+1$ steht, und daß jede Zelle genau ein Zeichen $a \in \Sigma_\Pi$ enthält:

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

Die Anzahl an Literalen in *R* beträgt

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

A beschreibt die Situation zum Zeitpunkt $t = 0$ (hierbei ist $b \in \Sigma_\Pi$ das Blankzeichen):

$$A = zust_{0,q_0} \wedge pos_{0,1} \wedge \left(\bigwedge_{i=1}^{n} band_{0,i,x_i} \right) \wedge \left(\bigwedge_{i=-p_{L_\Pi}(n)}^{0} band_{0,i,b} \right) \wedge \left(\bigwedge_{i=n+1}^{p_{L_\Pi}(n)+1} band_{0,i,b} \right).$$

A enthält $2\left(p_{L_\Pi}(n)+2\right) \in O\left(p_{L_\Pi}(n)\right)$ viele Literale.

$\ddot{U}_1$ beschreibt den Übergang von der zum Zeitpunkt *t* bestehenden Konfiguration zur Konfiguration zum Zeitpunkt $t + 1$ für $t = 0, ..., p_{L_\Pi}(n)$. Anstelle der Kopfbewegung *L*, *S* bzw. *R* werden hier die Werte $y = -1$, $y = 0$ bzw. $y = 1$ verwendet:

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

Die Indizes nehmen die Werte $t = 0, ..., p_{L_\Pi}(n)$, $q \in Q$, $i = -p_{L_\Pi}(n), ..., p_{L_\Pi}(n)+1$ und $a \in \Sigma_\Pi$ an.

Die geschweifte Klammer in $\ddot{U}_1$ enthält für feste Werte *t*, *q*, *a* und *i* höchstens $3(k+1)(l+1)$ viele Literale, in der eckigen Klammer sind es daher höchstens $3(k+1)(l+1)+3$. Insgesamt enthält $\ddot{U}_1$ höchstens

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

Um zu zeigen, wie sich auch $\ddot{U}_1$ in eine äquivalente Formel in konjunktiver Normalform umformen läßt, soll exemplarisch ein Ausdruck $zust_{t,q} \wedge pos_{t,i} \wedge band_{t,i,a} \Rightarrow \vee \left\{ zust_{t+1,q'} \wedge pos_{t+1,i+y} \wedge band_{t+1,i,a'} \mid (q',a',y) \in d(q,a) \right\}$ innerhalb der eckigen Klammer, der im rechten Teil zwei Alternativen $zust_{t+1,q'} \wedge pos_{t+1,i+y'} \wedge band_{t+1,i,a'}$ und $zust_{t+1,q''} \wedge pos_{t+1,i+y''} \wedge band_{t+1,i,a''}$ enthält, umgeformt werden. Um den Vorgang übersichtlich zu halten, wird dieser Ausdruck in der Form $X_1 \wedge X_2 \wedge X_3 \Rightarrow (Y_1 \wedge Y_2 \wedge Y_3) \vee (Z_1 \wedge Z_2 \wedge Z_3)$ geschrieben. Hier stehen X_i, Y_i und Z_i für jeweils drei Variablen. Es gilt

$$\begin{aligned} X_1 \wedge X_2 \wedge X_3 &\Rightarrow (Y_1 \wedge Y_2 \wedge Y_3) \vee (Z_1 \wedge Z_2 \wedge Z_3) \\ &= (\neg X_1 \vee \neg X_2 \vee \neg X_3) \vee (Y_1 \wedge Y_2 \wedge Y_3) \vee (Z_1 \wedge Z_2 \wedge Z_3) \\ &= (\neg X_1 \vee \neg X_2 \vee \neg X_3 \vee Y_1) \wedge (\neg X_1 \vee \neg X_2 \vee \neg X_3 \vee Y_2) \wedge (\neg X_1 \vee \neg X_2 \vee \neg X_3 \vee Y_3) \\ &\quad \wedge (\neg X_1 \vee \neg X_2 \vee \neg X_3 \vee Z_1) \wedge (\neg X_1 \vee \neg X_2 \vee \neg X_3 \vee Z_2) \wedge (\neg X_1 \vee \neg X_2 \vee \neg X_3 \vee Z_3). \end{aligned}$$

Wie man sieht, werden für jedes Y_i und Z_i vier Literale notiert, so daß sich für die Anzahl der Literale in der äquivalenten Formel innerhalb eines Ausdrucks in der eckigen Klammer im allgemeinen Fall eine Obergrenze von $12(k+1)(l+1)$ angeben läßt. Daher

bleibt die Anzahl der Literale in $\ddot{U}_1$, selbst in der konjunktiven Normalform, von der Ordnung $O\left(\left(p_{L_\Pi}(n)\right)^2\right)$.

$\ddot{U}_2$ besagt, daß sich der Inhalt von Zellen, über denen der Schreib/Lesekopf nicht steht, nicht ändert:
$\ddot{U}_2 = \underset{t,i,a}{\wedge}\left[\left(\neg pos_{t,i} \wedge band_{t,i,a}\right) \Rightarrow band_{t+1,i,a}\right]$.
Hierbei nehmen die Indizes die Werte $t = 0, ..., p_{L_\Pi}(n)$, $i = -p_{L_\Pi}(n), ..., p_{L_\Pi}(n)+1$ und $a \in \Sigma_\Pi$ an.
Die Anzahl an Literalen in $\ddot{U}_2$ beträgt $6\left(p_{L_\Pi}(n)+1\right)^2 \cdot (l+1) \in O\left(\left(p_{L_\Pi}(n)\right)^2\right)$.
$\ddot{U}_2$ läßt sich in eine äquivalente Formel in konfunktiver Normalform umformen, ohne die Anzahl an Literalen zu ändern:

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

E prüft nach, ob der Endzustand q_f zum Zeitpunkt $t = p_{L_\Pi}(n)$ erreicht ist:
$E = zust_{p_{L_\Pi}(n),q_f}$.

Insgesamt enthält $f_\Pi(x)$ eine Anzahl von Literalen der Ordnung $O\left(\left(p_{L_\Pi}(n)\right)^3\right)$, d.h. einer in der Länge des Eingabewortes polynomiellen Ordnung. Werden diese Literale durchnumeriert, so daß man $O\left(\left(p_{L_\Pi}(n)\right)^3\right)$ viele Literalpositionen bekommt, und dann binär kodiert (jede Zahl hat dann eine Länge der Ordnung $O\left(\log\left(\left(p_{L_\Pi}(n)\right)^3\right)\right) = O\left(\log\left(p_{L_\Pi}(n)\right)\right)$), so hat $f_\Pi(x)$ eine Länge der Ordnung $O\left(\left(p_{L_\Pi}(n)\right)^4\right)$. Daher ist $f_\Pi(x)$ bei Vorgabe von $x \in \Sigma_\Pi^*$ (für festes Π) deterministisch in polynomieller Zeit berechenbar.

Es läßt sich leicht nachweisen, daß die Eigenschaft „$x \in L_\Pi \Leftrightarrow f_\Pi(x)$ ist erfüllbar“ gilt. ///

Die Beweisskizze zeigt sogar:

Satz 5.4-3:
Das Erfüllbarkeitsproblem für Boolesche Ausdrücke in konjunktiver Normalform (CSAT) ist **NP**-vollständig.

NP-vollständige Probleme kann man innerhalb der Klasse **NP** als die am schwersten zu lösenden Probleme betrachten, denn sie entscheiden die **P**-**NP**-Frage:

Satz 5.4-4:
Gibt es mindestens ein **NP**-vollständiges Problem, das in **P** liegt , so ist **P** = **NP**.

Beweis:

Wegen $\mathbf{P} \subseteq \mathbf{NP}$ ist die umgekehrte Inklusion $\mathbf{NP} \subseteq \mathbf{P}$ zu zeigen. Es sei L_0 ein **NP**-vollständiges Problem, das in **P** liegt. Es sei $L \in \mathbf{NP}$. Zu zeigen ist $L \in \mathbf{P}$. Da L_0 **NP**-vollständig ist, gilt $L \leq_m^p L_0$. Mit Satz 5.4-1 folgt (wegen der Annahme $L_0 \in \mathbf{P}$) $L \in \mathbf{P}$. ///

Aus **NP**-vollständigen Problemen lassen sich aufgrund der Transitivität der $\leq_m$-Relation (Satz 5.4-1) weitere **NP**-vollständige Probleme ableiten:

Satz 5.4-5:
Ist das Problem zur Entscheidung einer Menge $L_0 \subseteq \Sigma_0^*$ **NP**-vollständig und ist $L_0 \leq_m^p L_1$ für eine Menge $L_1 \subseteq \Sigma_1^*$ so gilt:
Ist L_1 in **NP**, so ist L_1 ebenfalls **NP**-vollständig.

Satz 5.4-6:
Bei $\mathbf{P} \neq \mathbf{NP}$ gilt:

Ist ein Entscheidungsproblem Π zur Entscheidung der Menge $L_\Pi \subseteq \Sigma_\Pi^*$ **NP**-vollständig, so ist das Problem Π schwer lösbar (intractable), d.h. es gibt keinen polynomiell zeitbeschränkten deterministischen Lösungsalgorithmus zur Entscheidung von L_Π. Insbesondere ist das zugehörige Optimierungsproblem, falls es ein solches gibt, erst recht schwer (d.h. nur mit mindestens exponentiellem Aufwand) lösbar.

NP-vollständige Probleme mit praktischer Relevanz sind heute aus vielen Gebieten bekannt, z.B. aus der Graphentheorie, dem Netzwerk-Design, der Theorie von Mengen und Partitionen, der Datenspeicherung, dem Scheduling, der Maschinenbelegung, der Personaleinsatzplanung, der mathematischen Programmierung und Optimierung, der Analysis und Zahlentheorie, der Kryptologie, der Logik, der Automatentheorie und der Theorie Formaler Sprachen, der Programm- und Codeoptimierung usw. (siehe Literatur). Die folgende Zusammenstellung listet einige wenige Beispiele auf, die z.T. bereits behandelt wurden.

Beispiele für NP-vollständige Entscheidungsprobleme

- **Erfüllbarkeitsproblem der Aussagenlogik (SAT):**

  Instanz: *F*,
  F ist ein Boolescher Ausdruck (Formel der Aussagenlogik)

  Lösung: Entscheidung „ja“, falls gilt:
  F ist erfüllbar, d.h. es gibt eine Belegung der Variablen in *F* mit Werten TRUE bzw. FALSE, wobei gleiche Variablen mit gleichen Werten belegt werden, so daß sich bei Auswertung der Formel *F* der Wert TRUE ergibt.

  Kodiert man Formeln der Aussagenlogik über dem Alphabet $A = \{\wedge, \vee, \neg, (,), x, 0, 1\}$, so ist eine Formel der Aussagenlogik ein Wort über dem Alphabet *A*. Nicht jedes Wort über dem Alphabet *A* ist eine Formel der Aussagenlogik (weil es eventuell syntaktisch nicht korrekt ist) und nicht jede Formel der Aussagenlogik als Wort über dem Alphabet *A* ist erfüllbar.
  $L_{\text{SAT}} = \{F \mid F \text{ ist eine } \textbf{erfüllbare} \text{ Formel der Aussagenlogik}\}$.

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

  Instanz: *F*,
  F ist eine Formel der Aussagenlogik in konjunktiver Normalform mit höchstens 3 Literalen pro Klausel, d.h. sind $x_1, ..., x_n$ die verschiedenen in *F* vorkommenden Booleschen Variablen, so hat *F* die Form
  $F = F_1 \wedge F_2 \wedge ... F_m$
  Hierbei hat jedes F_i die Form $F_i = (y_{i_1} \vee y_{i_2} \vee y_{i_3})$ oder $F_i = (y_{i_1} \vee y_{i_2})$ oder $F_i = (y_{i_1})$, und y_{i_j} steht für eine Boolesche Variable (d.h. $y_{i_j} = x_l$) oder für eine negierte Boolesche Variable (d.h. $y_{i_j} = \neg x_l$) oder für eine Konstante 0 (d.h. $y_{i_j} = 0$) bzw. 1 (d.h. $y_{i_j} = 1$).

  Lösung: Entscheidung „ja“, falls gilt:
  F ist erfüllbar.

  Kodiert man die Formeln wie bei SAT, so gilt $L_{\text{3-CSAT}} \subseteq L_{\text{SAT}}$.

- **Kliquenproblem (KLIQUE):**

  Instanz: $[G,k]$,
  $G=(V,E)$ ist ein ungerichteter Graph und *k* eine natürliche Zahl.

  Lösung: Entscheidung „ja“, falls gilt:
  G besitzt eine „Klique“ der Größe *k*. Dieses ist eine Teilmenge $V' \subseteq V$ der Knotenmenge mit $|V'| = k$, und für alle $u \in V'$ und alle $v \in V'$ mit $u \neq v$ gilt $(u,v) \in E$.

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

  Instanz: $[a_1, \ldots, a_n, b]$,
  $a_1, \ldots, a_n$ und *b* sind natürliche Zahlen.

  Lösung: Entscheidung „ja“, falls gilt:
  Es gibt eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} a_i = b$.

- **Partitionenproblem (PARTITION):**

  Instanz: $[a_1, \ldots, a_n]$,
  $a_1, \ldots, a_n$ sind natürliche Zahlen.

  Lösung: Entscheidung „ja“, falls gilt:
  Es gibt eine Teilmenge $I \subseteq \{1, \ldots, n\}$ mit $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$.

- **Packungsproblem (BINPACKING):**

  Instanz: $[a_1, \ldots, a_n, b, k]$,
  $a_1, \ldots, a_n$ („Objekte“) sind natürliche Zahlen mit $a_i \leq b$ für *i* = 1, ..., *n*, $b \in \mathbf{N}$ („Behältergröße“), $k \in \mathbf{N}$.

  Lösung: Entscheidung „ja“, falls gilt:
  Die Objekte können so auf *k* Behälter der Füllhöhe *b* verteilt werden, so daß kein Behälter überläuft, d.h. es gibt eine Abbildung $f: \{1, \ldots, n\} \rightarrow \{1, \ldots, k\}$,

so daß für alle $j \in \{1, ..., k\}$ gilt: $\sum_{f(i)=j} a_i \leq b$

- **Problem des Hamiltonschen Kreises in einem gerichteten (bzw. ungerichteten) Graphen (GERICHTETER bzw. UNGERICHTETER HAMILTONKREIS):**

  Instanz: *G*,
  $G = (V, E)$ ist ein gerichteter bzw. ungerichteter Graph mit $V = \{v_1, ..., v_n\}$.

  Lösung: Entscheidung „ja“, falls gilt:
  G besitzt einen Hamiltonschen Kreis. Dieses ist eine Anordnung $(v_{p(1)}, v_{p(2)}, ..., v_{p(n)})$ der Knoten mittels einer Permutation π der Knotenindizes, so daß für $i = 1, ..., n-1$ gilt: $(v_{p(i)}, v_{p(i+1)}) \in E$ und $(v_{p(n)}, v_{p(1)}) \in E$.

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

  Instanz: $[M, k]$,
  $M = (M_{i,j})$ ist eine $(n \times n)$-Matrix von „Entfernungen“ zwischen *n* „Städten“ und eine Zahl *k*.

  Lösung: Entscheidung „ja“, falls gilt:
  Es gibt eine Permutation π (eine Tour, „Rundreise“), so daß $\sum_{i=1}^{n-1} M_{p(i),p(i+1)} + M_{p(n),p(1)} \leq k$ gilt?

Zum Nachweis der **NP**-Vollständigkeit für eines dieser Probleme Π ist neben der Zugehörigkeit von Π zu **NP** jeweils die Relation $\Pi_0 \leq_m^p \Pi$ zu zeigen, wobei hier Π_0 ein Problem ist, für das bereits bekannt ist, daß es **NP**-vollständig ist, und das eine „ähnliche“ Struktur aufweist. Häufig beweist man

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

3-CSAT $\leq_m^p$ KLIQUE und

3-CSAT $\leq_m^p$ GERICHTETER HAMILTONKREIS

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

5.5 Bemerkungen zur Struktur von NP

Es sei **NPC** die Menge der **NP**-vollständigen Sprachen über einem endlichen Alphabet Σ. $\mathbf{NPC} \subseteq \mathbf{NP}$. Unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ gilt $\mathbf{NPC} \cap \mathbf{P} = \emptyset$.

Es gilt folgender Satz:

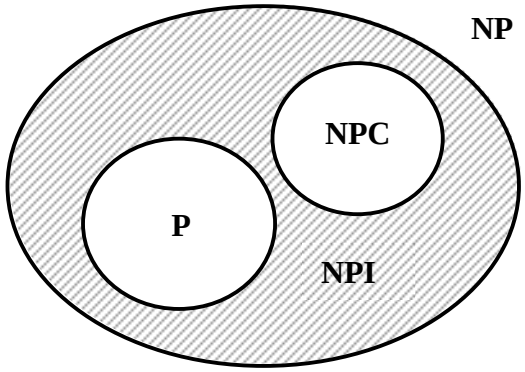
Satz 5.5-1:
Es sei B eine entscheidbare Menge mit $B \notin \mathbf{P}$. Dann gibt es eine Sprache $D \in \mathbf{P}$, so daß $A = D \cap B$ nicht zu **P** gehört, $A \leq_m B$, aber nicht $B \leq_m^p A$ gelten.

Satz 5.5-1 läßt sich folgendermaßen anwenden:

Es sei B eine **NP**-vollständige Sprache. Unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ gilt $B \notin \mathbf{P}$. Die Sprache $A = D \cap B$ gehört zu **NP**, da $D \in \mathbf{P}$ und $B \in \mathbf{NP}$ sind. Nach dem obigen Satz gilt nicht $B \leq_m^p A$, also ist A nicht **NP**-vollständig. Folglich gilt:

Satz 5.5-2:
Unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ ist $\mathbf{NP} \setminus (\mathbf{P} \cup \mathbf{NPC}) \neq \emptyset$.

Bezeichnet man $\mathbf{NP} \setminus (\mathbf{P} \cup \mathbf{NPC})$ als die Menge **NPI** der **NP-unvollständigen Sprachen**, so zeigt **NP** folgende Struktur:



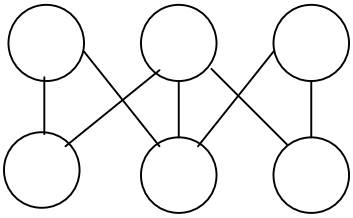
Die Sprachen in **NPI** liegen bezüglich ihrer Komplexität also zwischen den „leichten“ Sprachen in **P** und den „schweren“ Sprachen in **NPC**. Obwohl es (bei $\mathbf{P} \neq \mathbf{NP}$) unendlich viele Sprachen in **NPI** geben muß, ist die Angabe konkreter Beispiele schwierig. Bisher kennt man kein Problem aus **NPI**. Von dem folgenden Graphenisomorphieproblem ISO wird vermutet, daß es in **NPI** liegt, da bisher (trotz großer Anstrengung) weder der Nachweis für $\text{ISO} \in \mathbf{P}$ noch der Nachweis $\text{ISO} \in \mathbf{NPC}$ gelungen ist.

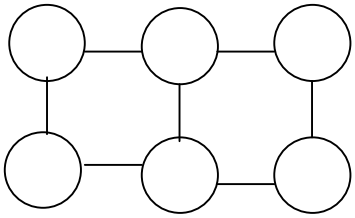
Graphenisomorphieproblem (ISO):

Instanz: Graphen $G_1 = (V_1, E_1)$ und $G_2 = (V_2, E_2)$

Lösung: Entscheidung „ja“, falls gilt:
G_1 und G_2 sind isomorph, d.h. es gibt eine Abbildung $f : V_1 \to V_2$ mit der Eigenschaft $(v,w) \in E_1 \Leftrightarrow (f(v), f(w)) \in E_2$.

Beispielsweise sind die folgenden beiden Graphen isomorph:





Die folgende Sprachklasse besteht aus Sprachen, deren Komplemente in **NP** liegen:

$$\textbf{co - NP} = \{ \Sigma^* \setminus L \mid L \text{ ist eine Sprache über } \Sigma \text{ und } L \in \textbf{NP} \}.$$

Für das Primzahlproblem PRIMES kann man zeigen, daß es in **NP** und in **co - NP** liegt (es wird vermutet, daß PRIMES in **P** liegt):

Primzahlproblem (PRIMES):

Instanz: $n \in \mathbf{N}$ in binärer Darstellung, $size(n) = \log(n)$.

Lösung: Entscheidung „ja“, falls *n* eine Primzahl ist.

Lösung: Entscheidung „ja“, falls *n* eine Primzahl ist.

Hierzu wird ein zahlentheoretischer Satz benötigt, der an dieser Stelle angeführt wird und dessen Beweis in der angegebenen Literatur nachgelesen werden kann:

Satz 5.5-3:

Die Zahl $n \in \mathbf{N}$ mit $n > 2$ ist genau dann eine Primzahl, wenn es eine Zahl $a \in \mathbf{N}$ mit $1 < a \leq n-1$ gibt, die folgende Eigenschaften besitzt:

(i) $a^{n-1} \equiv 1 (\mathrm{mod}\, n)$

(ii) $a^{(n-1)/p} \neq 1 (\mathrm{mod}\, n)$ für jeden Primteiler p von $n-1$.

Satz 5.5-4:

PRIMES $\in \mathbf{NP} \cap \mathbf{co}\text{-}\mathbf{NP}$.

Beweis:

Zu zeigen ist, daß es sowohl für $L_1 = \{ n \mid n \in \mathbf{N} \text{ und } n \text{ ist eine Primzahl} \}$ als auch für $L_2 = \{ n \mid n \in \mathbf{N} \text{ und } n \text{ ist keine Primzahl} \}$ jeweils einen polynomiell zeitbeschränkten Verifizierer gibt. Der Entwurf des Verifizierers für L_1 basiert auf Satz 5.5-3, ein Verifizierer für L_2 ergibt sich direkt aus den Eigenschaften von zusammengesetzten Zahlen. Beide Verifizierer werden hier in Form von Pseudocode beschrieben.

Ein Verifizierer $\mathbf{V}_{L_2}$ für $L_2 = \{ n \mid n \in \mathbf{N} \text{ und } n \text{ ist keine Primzahl} \}$ erhält als Eingabeinstanz eine Zahl $n \in \mathbf{N}$ (in Binärkodierung) und einen Beweis B, der in diesem Fall ebenfalls eine Zahl $B \in \mathbf{N}$ (in Binärkodierung) mit $size(B) \leq size(n)$ bzw. $1 < B \leq n-1$ ist. Die Arbeitsweise von $\mathbf{V}_{L_2}$ lautet:

```
FUNCTION V_L2 (n : INTEGER;
               B : INTEGER) : ...;

BEGIN { V_L2 }
  IF 1<B≤n-1                { Zeile 1 }
  THEN IF (n mod B)=0       { Zeile 2 }
       THEN V_L2 := ja
       ELSE V_L2 := nein
  ELSE V_L2 := nein;
END   { V_L2 };
```

Die Eingabe n für $\mathbf{V}_{L_2}$ belegt k Bits mit $k \in O(\log(n))$, $k = size(n)$. Die Überprüfung in Zeile 1 benötigt $O((\log(n))) = O(k)$ Operationen. Die Überprüfung in Zeile 2 (die Berechnung des

Wertes $(n \bmod B)$) kann mit $O\left((\log(n))^2\right) = O\left(k^2\right)$ vielen Bitoperationen durchgeführt werden; denn $(n \bmod B) = n - (n \operatorname{DIV} B) \cdot B$, und alle arithmetische Operationen benötigen nach Satz 2.1-3 höchstens quadratisch viele Bitoperationen. Daher arbeitet $\mathbf{V}_{L_2}$ in polynomieller Zeit, gemessen in der Größe der Eingabe *n*.

Ist *n* keine Primzahl, dann besitzt *n* einen Teiler $a \in \mathbf{N}$ mit $1 < a \leq n-1$. Gibt man diesen Wert (Beweis) in $\mathbf{V}_{L_2}$ für den Parameter *B* ein, so antwortet $\mathbf{V}_{L_2}$ mit $\mathbf{V}_{L_2}(n,a) = \texttt{ja}$.
Ist *n* eine Primzahl, dann teilt kein $a \in \mathbf{N}$ mit $1 < a \leq n-1$ die Zahl *n*, d.h. für alle $a \in \mathbf{N}$ mit $1 < a \leq n-1$ ist $(n \bmod a) \neq 0$. Daher wird $\mathbf{V}_{L_2}$ bei Eingabe von *n* und eines beliebigen Beweises *B* immer die Antwort $\mathbf{V}_{L_2}(n,B) = \texttt{nein}$ geben.

Insgesamt ist damit $L_2 \in \mathbf{NP}$ gezeigt, also PRIMES $\in$ **co - NP**.

Ein Verifizierer $\mathbf{V}_{L_1}$ für $L_1 = \{n \mid n \in \mathbf{N}$ und *n* ist eine Primzahl$\}$ ist komplizierter zu konstruieren und setzt die in Satz 5.5-3 beschriebene Charakterisierung von Primzahlen um.

```
FUNCTION V_L1 (n : INTEGER;
               B : INTEGER) : ...;

BEGIN { V_L1 }
  IF n = 2
  THEN V_L1 := ja
  ELSE IF (NOT (1 < B ≤ n-1)) OR ( B^(n-1) ≠ 1 (mod n) )          { Zeile 1 }
       THEN V_L1 := nein
       ELSE BEGIN
              erzeuge k ≤ log(n-1) Zahlen p_1, ..., p_k mit 3 ≤ p_i ≤ (n-1)/2
                     und k Zahlen B_1, ..., B_k mit 1 < B_i ≤ p_i - 1
                     für i = 1,...,k ;                             { Zeile 2 }
              IF NOT( (n-1 mod p_i) ≠ 0 für i = 1,...,k )          { Zeile 3 }
              THEN V_L1 := nein
              ELSE BEGIN
                     IF (( V_L1 ( p_i, B_i )= ja) für i = 1,...,k )   { Zeile 4 }
                     THEN IF ( B^((n-1)/p_i) ≠ 1 (mod n) ) für i = 1,...,k ){ Zeile 5 }
                          THEN V_L1 := ja
                          ELSE V_L1 := nein
                     ELSE V_L1 := nein;
                   END;
```

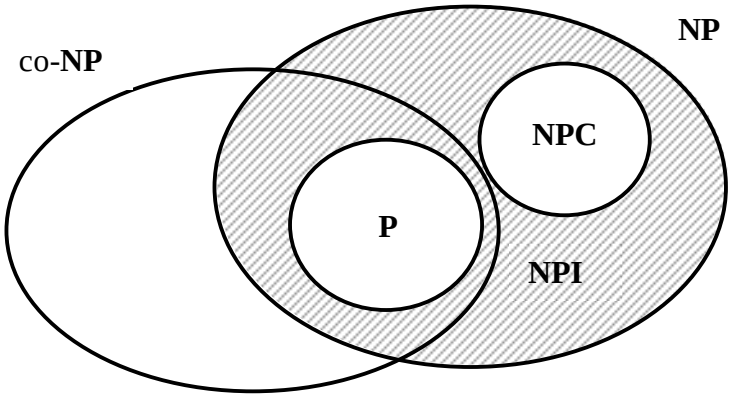
```
            END;
END   { V_L1 };
```

Für die Korrektkeit ist zu beachten, daß in Zeile 2 auf nichtdeterministische Weise die Menge der Primfaktoren von $n-1$ erzeugt wird. Die Werte B_i sind die zugehörigen Beweise. Die Anzahl der Primfaktoren von $n-1$ ist durch $\log(n-1)$ beschränkt. Außerdem gilt in Zeile 2 wegen $n \geq 3$, daß jeder Primfaktor von $n-1$ die Abschätzung $3 \leq p_i \leq (n-2)/2$ erfüllt. In Zeile 4 wird also $\mathbf{V}_{L_1}$ mit kleineren Werten rekursiv aufgerufen, deren Stellenzahl echt kleiner als die Stellenzahl von *n* ist. In Zeile 3 wird geprüft, ob die erzeugten Zahlen p_i Teiler von $n-1$ sind, und in Zeile 4 wird geprüft, ob die Zahlen p_i Primzahlen sind. Zeile 5 prüft Bedingung (ii) aus Satz 5.5-3. Es läßt sich zeigen (siehe angegebene Literatur), daß die Berechnung in den Zeilen 1 und 5 durch ein Laufzeitverhalten der Ordnung $O\left((\log(n))^3\right)$ abgeschätzt werden kann. Insgesamt läßt sich damit zeigen, daß die Laufzeit von $\mathbf{V}_{L_1}$ polynomiell in $size(n)$ ist. ///

Man hat für viele Probleme in **co - NP** jedoch nicht nachweisen können, daß sie in **NP** liegen (PRIMES gehört nicht dazu). Daher wird angenommen (obwohl es noch keinen Beweis dafür gibt), daß $\mathbf{NP} \neq \mathbf{co\text{-}NP}$ gilt. Aus der Gültigkeit von $\mathbf{NP} \neq \mathbf{co\text{-}NP}$ würde übrigens folgen, daß $\mathbf{P} \neq \mathbf{NP}$ ist, da wegen der Abgeschlossenheit der Klasse **P** gegenüber Komplementbildung $\mathbf{P} = \mathbf{co\text{-}P}$ gilt.

Satz 5.5-5:
Gibt es eine Sprache $L \in \mathbf{NPC}$ mit $\Sigma^* \setminus L \in \mathbf{NP}$, dann ist $\mathbf{NP} = \mathbf{co\text{-}NP}$.

Unter den Annahmen $\mathbf{P} \neq \mathbf{NP}$ und $\mathbf{NP} \neq \mathbf{co\text{-}NP}$ ergibt sich folgendes Gesamtbild der beschriebenen Klassen:



6 Approximation von Optimierungsaufgaben

Gegeben sei das Optimierungsproblem Π:

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

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

Im folgenden (und in den vorhergehenden Beispielen) werden Optimierungsproblemen untersucht, die auf der Grenze zwischen praktischer Lösbarkeit (tractability) und praktischer Unlösbarkeit (intractability) stehen. In Analogie zu Entscheidungsproblemen in **NP** bilden diese die Klasse **NPO**:

Das Optimierungsproblem Π gehört zur **Klasse NPO**, wenn gilt:

1. Die Menge der Instanzen $x \in \Sigma_\Pi^*$ ist in polynomieller Zeit entscheidbar
2. Es gibt ein Polynom q mit der Eigenschaft: für jedes $x \in \Sigma_\Pi^*$ und jede zulässige Lösung $y \in \mathrm{SOL}_\Pi(x)$ gilt $|y| \le q(|x|)$, und für jedes y mit $|y| \le q(|x|)$ ist in polynomieller Zeit entscheidbar, ob $y \in \mathrm{SOL}_\Pi(x)$ ist
3. Die Zielfunktion m_Π ist in polynomieller Zeit berechenbar.

Alle bisher behandelten Beispiele für Optimierungsprobleme liegen in der Klasse **NPO**. Dazu gehören insbesondere auch solche, deren zugehöriges Entscheidungsproblem **NP**-vollständig ist.

Den Zusammenhang zwischen den Klassen **NPO** und **NP** beschreibt der Satz

Satz 6-1:
Für jedes Optimierungsproblem in **NPO** ist das zugehörige Entscheidungsproblem in **NP**.

Beweis:

Der Beweis wird hier nur für ein Maximierungsproblem erbracht, für ein Minimierungsproblem kann man in ähnlicher Weise argumentieren.

Es sei Π ein Maximierungsproblem in **NPO** (die im Beweis verwendeten Funktionen werden in den obigen Definitionen benannt). Das zugehörige Entscheidungsproblem sei Π_{Ent}. Hierbei soll bei Vorlage einer Instanz $[x,K]$ mit $x \in \Sigma_\Pi^*$ und $K \in \mathbf{N}$ genau dann die Entscheidung $x \in L_{\Pi_{Ent}}$ getroffen werden, wenn $m_\Pi^*(x) \geq K$ ist. Um zu zeigen, daß Π_{Ent} in **NP** liegt, ist ein Verifizierer **V** für Π_{Ent} anzugeben, der bei Eingabe einer Instanz $[x,K]$ und eines Beweises B diesen in polynomieller Zeit verifiziert. Ein Beweis ist hier eine Zeichenkette, die über dem Alphabet Σ_0 gebildet wird, mit dem man auch die zulässigen Lösungen von *x* formuliert.

Der Verifizierer **V** für Π_{Ent} wird durch den folgenden Pseudocode gegeben:

```
FUNCTION V ([x,K] : ...;
            B     : ...) : ...;
          { x ∈ Σ*_Π , K ∈ N , B ∈ Σ*_0 }

BEGIN { V }
  IF (|B| ≤ q(|x|)) AND (B ∈ SOL_Π(x))          { Zeile 1 }
  THEN IF m_Π(x,B) ≥ K THEN V := „ja“           { Zeile 2 }
                       ELSE V := „nein“
  ELSE V := „nein“;
END   { V };
```

Zu zeigen ist

1. **V** arbeitet in polynomieller Zeit, gemessen in $|x|$
2. $[x,K] \in L_{\Pi_{Ent}} \Leftrightarrow$ es gibt $B_x \in \Sigma_0^*$ mit $\mathbf{V}(x,B_x) = \mathtt{ja}$.

Zu 1.: Da Π in **NPO** ist, lassen sich die Berechnungen in den Zeilen 1 und 2 in polynomieller Zeit ausführen; dieses wird durch die Punkte 2. und 3. in der Definition der Klasse **NPO** gesichert.

Zu 2.: Es sei $[x,K] \in L_{\Pi_{Ent}}$. Dann gibt es eine optimale Lösung $y^* \in \mathrm{SOL}_\Pi(x)$ mit $m^*_\Pi(x) = m_\Pi(x,y^*) \geq K$. Da Π in **NPO** ist, gilt $|y^*| \leq q(|x|)$. Bei Eingabe von $[x,K]$ und y^* in **V** ergibt sich $\mathbf{V}([x,K],y^*) = \texttt{ja}$.
Es gelte $[x,K] \notin L_{\Pi_{Ent}}$. Dann gilt für jedes $y \in \mathrm{SOL}_\Pi(x)$: $m_\Pi(x,y) \leq m^*_\Pi(x) < K$. Es sei $B \in \Sigma_0^*$ mit $|B| \leq q(|x|)$. Ist $B \in \mathrm{SOL}_\Pi(x)$, dann antwortet **V** wegen $m_\Pi(x,B) < K$ in Zeile 2 mit $\mathbf{V}([x,K],B) = \texttt{nein}$. Ist $B \notin \mathrm{SOL}_\Pi(x)$, dann antwortet **V** wegen Zeile 1 mit $\mathbf{V}([x,K],B) = \texttt{nein}$. ///

Analog zur Definition der Klasse **P** innerhalb **NP** läßt sich innerhalb **NPO** eine Klasse **PO** definieren:

Ein Optimierungsproblem Π aus **NPO** gehört zur **Klasse PO**, wenn es einen polynomiell zeitbeschränkten Algorithmus gibt, der für jede Instanz $x \in \Sigma_\Pi^*$ eine optimale Lösung $y^* \in \mathrm{SOL}_\Pi(x)$ zusammen mit dem optimalen Wert $m^*_\Pi(x)$ der Zielfunktion ermittelt.

Offensichtlich ist $\mathbf{PO} \subseteq \mathbf{NPO}$.

Es gilt:

Satz 6-2:
Ist $\mathbf{P} \neq \mathbf{NP}$, dann ist $\mathbf{PO} \neq \mathbf{NPO}$.

Im Laufe dieses Kapitels wird die Struktur der Klasse **NPO** genauer untersucht. Im folgenden liegen daher alle behandelten Optimierungsprobleme in **NPO**.

Bei Optimierungsaufgaben gibt man sich häufig mit Näherungen an die optimale Lösung zufrieden, insbesondere dann, wenn diese „leicht“ zu berechnen sind und vom Wert der optimalen Lösung nicht zu sehr abweichen. Unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ gibt es für ein Optimierungsproblem, dessen zugehöriges Entscheidungsproblem **NP**-vollständig ist, kein Verfahren, das eine optimale Lösung in polynomieller Laufzeit ermittelt. Gerade diese Probleme sind in der Praxis jedoch häufig von großem Interesse.

Für eine Instanz $x \in \Sigma_\Pi^*$ und für eine zulässige Lösung $y \in \mathrm{SOL}_\Pi(x)$ bezeichnet
$D(x,y) = \left| m_\Pi^*(x) - m_\Pi(x,y) \right|$
den **absoluten Fehler von** *y* **bezüglich** *x*.

Ein Algorithmus **A** ist ein **Approximationsalgorithmus (Näherungsalgorithmus)** für Π, wenn er bei Eingabe von $x \in \Sigma_\Pi^*$ eine zulässige Lösung liefert, d.h. wenn $\mathbf{A}(x) \in \mathrm{SOL}_\Pi(x)$ gilt. **A** heißt **absoluter Approximationsalgorithmus (absoluter Näherungsalgorithmus)**, wenn es eine Konstante *k* gibt mit $D(x, \mathbf{A}(x)) = \left| m_\Pi^*(x) - m_\Pi(x, \mathbf{A}(x)) \right| \le k$. Das Optimierungsproblem Π heißt in diesem Fall **absolut approximierbar**.

Ist Π ein Optimierungsproblem, dessen zugehöriges Entscheidungsproblem **NP**-vollständig ist, so sucht man natürlich nach absoluten Approximationsalgorithmen für Π, die polynomielle Laufzeit aufweisen und für die der Wert $D(x, \mathbf{A}(x))$ möglichst klein ist. Das folgende Beispiel zeigt jedoch, daß unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ nicht jedes derartige Problem absolut approximierbar ist. Dazu werde der folgende Spezialfall des 0/1-Rucksack-Maximierungsproblems betrachtet:

Das ganzzahlige 0/1-Rucksack-Maximierungsproblem

Instanz: 1. $I = (A, M)$
$A = \{a_1, ..., a_n\}$ ist eine Menge von *n* Objekten und $M \in \mathbf{N}$ die „Rucksackkapazität". Jedes Objekt a_i, $i = 1, ..., n$, hat die Form $a_i = (w_i, p_i)$; hierbei bezeichnet $w_i \in \mathbf{N}$ das Gewicht und $p_i \in \mathbf{N}$ den Wert (Profit) des Objekts a_i.
$size(I) = n$

2. $\mathrm{SOL}(I) = \left\{ (x_1, ..., x_n) \;\middle|\; x_i = 0 \text{ oder } x_i = 1 \text{ für } i = 1, ..., n \text{ und } \sum_{i=1}^{n} x_i \cdot w_i \le M \right\}$

3. $m(I, (x_1, ..., x_n)) = \sum_{i=1}^{n} x_i \cdot p_i$ für $(x_1, ..., x_n) \in \mathrm{SOL}(I)$

4. *goal* = max

Lösung: Eine Folge $x_1^*, ..., x_n^*$ von Zahlen mit

(1) $x_i^* = 0$ oder $x_i^* = 1$ für $i = 1, ..., n$

(2) $\sum_{i=1}^{n} x_i^* \cdot w_i \le M$

(3) $m(I, (x_1^*, ..., x_n^*)) = \sum_{i=1}^{n} x_i^* \cdot p_i$ ist maximal unter allen möglichen Auswahlen $x_1, ..., x_n$, die (1) und (2) erfüllen.

Das zugehörige Entscheidungsproblem ist **NP**-vollständig, so daß dieses Optimierungsproblem unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ keinen polynomiell zeitbeschränkten Lösungsalgorithmus (der eine *optimale* Lösung ermittelt) besitzt. Es gilt sogar:

Satz 6-3:

Es sei *k* eine vorgegebene Konstante. Unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ gibt es keinen polynomiell zeitbeschränkten Approximationsalgorithmus **A** für das ganzzahlige 0/1-Rucksack-Maximierungsproblem, der bei Eingabe einer Instanz $I = (A, M)$ eine zulässige Lösung $\mathbf{A}(I) \in \mathrm{SOL}(I)$ berechnet, für deren absoluter Fehler $D(I, \mathbf{A}(I)) = \left| m^*(I) - m(I, \mathbf{A}(I)) \right| \leq k$ gilt.

Beweis:

Die verwendete Beweistechnik ist auch auf andere Probleme übertragbar.

Es wird angenommen, daß es (bei Vorgabe der Konstanten *k*) einen derartigen polynomiell zeitbeschränkten Approximationsalgorithmus **A** gibt und zeigt dann, daß dieser (mit einer einfachen Modifikation) dazu verwendet werden kann, in polynomieller Zeit eine optimale Lösung für das ganzzahlige 0/1-Rucksack-Maximierungsproblem zu ermitteln. Damit kann man dann das zu diesem Problem gehörende Entscheidungsproblem in polynomieller Zeit lösen. Da dieses **NP**-vollständig ist, folgt $\mathbf{P} = \mathbf{NP}$ (vgl. Satz 5.1-1).

Bei Annahme der Existenz von **A** kann ein Algorithmus $\tilde{\mathbf{A}}$ definiert werden, der folgendermaßen arbeitet:
Aus einer Eingabe einer Instanz $I = (A, M)$ mit $A = \{(w_1, p_1), ..., (w_n, p_n)\}$ für das ganzzahlige 0/1-Rucksack-Maximierungsproblem erzeugt $\tilde{\mathbf{A}}$ eine Instanz $\tilde{I} = (\tilde{A}, M)$ mit $\tilde{A} = \{(w_1, (k+1) \cdot p_1), ..., (w_n, (k+1) \cdot p_n)\}$ (es werden dabei also lediglich alle Profite p_i durch $(k+1) \cdot p_i$ ersetzt). Diese neue Instanz $\tilde{I} = (\tilde{A}, M)$ wird in **A** eingegeben. **A** ermittelt eine approximative Lösung $\mathbf{A}(\tilde{I}) = (x_1, ..., x_n)$ mit $\sum_{i=1}^{n} x_i \cdot w_i \leq M$ und

$$\left| m^*(\tilde{I}) - m(\tilde{I}, \mathbf{A}(\tilde{I})) \right| = \left| m^*(\tilde{I}) - \sum_{i=1}^{n} x_i \cdot (k+1) \cdot p_i \right| \leq k \qquad \text{(Ungleichung (*)).}$$

$\tilde{\mathbf{A}}$ gibt die von **A** bei Eingabe von $\tilde{I}$ ermittelte Lösung $\tilde{\mathbf{A}}(I) = \mathbf{A}(\tilde{I}) = (x_1, ..., x_n)$ mit (dem Wert der Zielfunktion) $m(I, \tilde{\mathbf{A}}(I)) = \dfrac{m(\tilde{I}, \mathbf{A}(\tilde{I}))}{k+1}$ aus. Es ist $\tilde{\mathbf{A}}(I) \in \mathrm{SOL}(I)$. Es gilt sogar $m(I, \tilde{\mathbf{A}}(I)) = m^*(I)$, d.h. $\tilde{\mathbf{A}}$ liefert in polynomieller Zeit (da **A** in polynomieller Zeit arbeitet) eine optimale Lösung für die Eingabeinstanz *I*:

Da $m^*(\tilde{I})$ ein Vielfaches von $k+1$ ist, denn jeder Profit in $\tilde{I}$ ist ein Vielfaches von $k+1$, kann man den Faktor $k+1$ auf der linken Seite des $\leq$-Zeichens in der Ungleichung (*) ausklammern und erhält

$$\left|m^*(\tilde{I}) - m(\tilde{I}, \mathbf{A}(\tilde{I}))\right| = \left|m^*(\tilde{I}) - \sum_{i=1}^{n} x_i \cdot (k+1) \cdot p_i\right| = (k+1) \cdot R \leq k$$

mit einer natürlichen Zahl *R*. Diese Ungleichung ist nur mit $R = 0$ möglich, so daß $\left|m^*(\tilde{I}) - m(\tilde{I}, \mathbf{A}(\tilde{I}))\right| = 0$ folgt, d.h. $\mathbf{A}(\tilde{I})$ ist eine optimale Lösung für $\tilde{I}$.

Jede zulässige Lösung für $\tilde{I}$ ist auch eine zulässige Lösung für *I*, jedoch mit dem $(k+1)$-fachen Profit. Umgekehrt ist jede zulässige Lösung für *I* eine zulässige Lösung für $\tilde{I}$. Damit folgt die Optimalität von $\tilde{\mathbf{A}}(I)$ (dazu ist $m(I, \tilde{\mathbf{A}}(I)) \geq m(I, y)$ für jedes $y \in \mathrm{SOL}(I)$ zu zeigen):

$$m(I, \tilde{\mathbf{A}}(I)) = \frac{m(\tilde{I}, \mathbf{A}(\tilde{I}))}{k+1} = \frac{m^*(\tilde{I})}{k+1} \geq \frac{m(\tilde{I}, y)}{k+1} = m(I, y) \text{ für jedes } y \in \mathrm{SOL}(I). \text{ ///}$$

Die Forderung nach der Garantie der Einhaltung eines absoluten Fehlers ist also häufig zu stark. Es bietet sich daher an, einen Approximationsalgorithmus nach seiner relativen Approximationsgüte zu beurteilen.

Es sei ein Π wieder ein Optimierungsproblem und **A** ein Approximationsalgorithmus für Π (siehe oben), der eine zulässige Lösung $\mathbf{A}(x)$ ermittelt. Ist $x \in \Sigma_\Pi^*$ eine Instanz von Π, so gilt trivialerweise $m(x, \mathbf{A}(x)) \leq m_\Pi^*(x)$ bei einem Maximierungsproblem bzw. $m_\Pi^*(x) \leq m(x, \mathbf{A}(x))$ bei einem Minimierungsproblem.

Die **relative Approximationsgüte** $R_\mathbf{A}(x)$ von **A** bei Eingabe einer Instanz $x \in \Sigma_\Pi^*$ wird definiert durch

$$R_\mathbf{A}(x) = \frac{m_\Pi^*(x)}{m(x, \mathbf{A}(x))} \text{ bei einem Maximierungsproblem}$$

bzw.

$$R_\mathbf{A}(x) = \frac{m(x, \mathbf{A}(x))}{m_\Pi^*(x)} \text{ bei einem Minimierungsproblem.}$$

Bemerkung: Um nicht zwischen Maximierungs- und Minimierungsproblem in der Definition unterscheiden zu müssen, kann man die relative Approximationsgüte von **A** bei Eingabe einer Instanz $x \in \Sigma_\Pi^*$ auch durch

$$R_\mathbf{A}(x) = \max\left\{\frac{m_\Pi^*(x)}{m(x, \mathbf{A}(x))}, \frac{m(x, \mathbf{A}(x))}{m_\Pi^*(x)}\right\}$$

definieren.

Offensichtlich gilt immer $1 \le R_{\mathbf{A}}(x)$. Je dichter $R_{\mathbf{A}}(x)$ bei 1 liegt, um so besser ist die Approximation.

Die Aussage „$R_{\mathbf{A}}(x) \le c$“ mit einer Konstanten $c \ge 1$ für alle Instanzen $x \in \Sigma_{\Pi}^{*}$ bedeutet bei einem Maximierungsproblem $1 \le \frac{m_{\Pi}^{*}(x)}{m(x, \mathbf{A}(x))} \le c$, also $m(x, \mathbf{A}(x)) \ge 1/c \cdot m_{\Pi}^{*}(x)$, d.h. der Approximationsalgorithmus liefert eine Lösung, die sich mindestens um $1/c$ dem Optimum nähert. Gewünscht ist also ein möglichst großer Wert von $1/c$ bzw. ein Wert von *c*, der möglichst klein (d.h. dicht bei 1) ist.

Bei einem Minimierungsproblem impliziert die Aussage „$R_{\mathbf{A}}(x) \le c$“ die Beziehung $m(x, \mathbf{A}(x)) \le c \cdot m_{\Pi}^{*}(x)$, d.h. der Wert der Approximation überschreitet das Optimum um höchstens das *c*-fache. Auch hier ist also ein Wert von *c* gewünscht, der möglichst klein (d.h. dicht bei 1) ist.

6.1 Relativ approximierbare Probleme

Es sei $r \ge 1$. Der Approximationsalgorithmus **A** für das Optimierungsproblem Π aus **NPO** heißt ***r*-approximativer Algorithmus**, wenn $R_{\mathbf{A}}(x) \le r$ für jede Instanz $x \in \Sigma_{\Pi}^{*}$ gilt.

Bemerkung: Es sei **A** ein Approximationsalgorithmus für das Minimierungsproblem Π, und es gelte $m(x, \mathbf{A}(x)) \le r \cdot m_{\Pi}^{*}(x) + k$ für alle Instanzen $x \in \Sigma_{\Pi}^{*}$ (mit Konstanten *r* und *k*). Dann ist **A** lediglich $(r + k)$-approximativ und nicht etwa *r*-approximativ, jedoch **asymptotisch** *r*-approximativ (siehe Kapitel 6.2).

Die **Klasse APX** besteht aus denjenigen Optimierungsproblemen aus **NPO**, für die es einen *r*-approximativen Algorithmus für ein $r \ge 1$ gibt.

Offensichtlich ist $\mathbf{APX} \subseteq \mathbf{NPO}$.

Das folgende Optimierungsproblem liegt in **APX**:

Binpacking-Minimierungsproblem:

Instanz:
1. $I = [a_1, \ldots, a_n]$
   $a_1, \ldots, a_n$ („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, \ldots, n$
   $size(I) = n$
2. $\mathrm{SOL}(I) = \left\{ [B_1, \ldots, B_k] \;\middle|\; \begin{array}{l} [B_1, \ldots, B_k] \text{ ist eine Partition (disjunkte Zerlegung)} \\ \text{von } I \text{ mit } \sum_{a_i \in B_j} a_i \leq 1 \text{ für } j = 1, \ldots, k \end{array} \right\}$, d.h. in einer zulässigen Lösung werden die Objekte so auf k „Behälter“ der Höhe 1 verteilt, daß kein Behälter „überläuft“.
3. Für $[B_1, \ldots, B_k] \in \mathrm{SOL}(I)$ ist $m(I, [B_1, \ldots, B_k]) = k$, d.h. als Zielfunktion wird die Anzahl der benötigten Behälter definiert
4. *goal* = min

Lösung: Eine Partition der Objekte in möglichst wenige Teile $B_1, \ldots, B_{k^*}$ und (implizit) die Anzahl k^* der benötigten Teile.

Bemerkung: Da das Laufzeitverhalten der folgenden Approximationsalgorithmen nicht von den Größen der in den Instanzen vorkommenden Objekte abhängt, sondern nur von deren Anzahl, kann man für eine Eingabeinstanz $I = [a_1, \ldots, a_n]$ als Größe den Wert $size(I) = n$ wählen.

Das Binpacking-Minimierungsproblem ist eines der am besten untersuchten Minimierungsprobleme einschließlich der Verallgemeinerungen auf mehrdimensionale Objekte. Das zugehörige Entscheidungsproblem ist **NP**-vollständig, so daß unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ kein polynomiell zeitbeschränkter Optimierungsalgorithmus erwartet werden kann.

Der folgende in Pseudocode formulierte Algorithmus approximiert eine optimale Lösung:

Nextfit-Algorithmus zur Approximation von Binpacking:

Eingabe: $I = [a_1, \ldots, a_n]$,
$a_1, \ldots, a_n$ („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, \ldots, n$

Verfahren: $B_1 := a_1$;

Sind $a_1, ..., a_i$ bereits in die Behälter $B_1, ..., B_j$ gelegt worden, ohne daß einer dieser Behälter überläuft, so lege a_{i+1} nach B_j (in den Behälter mit dem höchsten Index), falls dadurch B_j nicht überläuft, d.h. $\sum_{a_l \in B_j} a_l \le 1$ gilt; andernfalls lege a_{i+1} in einen neuen (bisher leeren) Behälter B_{j+1}.

Ausgabe: $\mathbf{NF}(I) =$ die benötigten nichtleeren Behälter $[B_1, ..., B_k]$.

Satz 6.1-1:

Ist $I = [a_1, ..., a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $R_{\mathbf{NF}}(I) \le 2$, d.h. der Nextfit-Algorithmus ist 2-approximativ ($m(I, \mathbf{NF}(I)) \le 2 \cdot m^*(I)$). Das Binpacking -Minimierungsproblem liegt in **APX**.

Beweis:

Interessanterweise kann man die Aussage $m(I, \mathbf{NF}(I)) \le 2 \cdot m^*(I)$ machen, ohne $m^*(I)$ oder $m(I, \mathbf{NF}(I))$ zu kennen. Dazu wird $m^*(I)$ abgeschätzt:

Für eine Instanz *I* habe der Nextfit-Algorithmus *k* Behälter ermittelt, d.h. $m(I, \mathbf{NF}(I)) = k$. Die Füllhöhe im *j*-ten Behälter sei u_j für *j* = 1, ..., *k*. Das erste Element, das der Nextfit-Algorithmus in den Behälter B_j gelegt hat, sei b_j.

Man betrachte die beiden Behälter B_j und B_{j+1} (für $1 \le j < k$): Da b_{j+1} nicht mehr in den Behälter B_j paßte, gilt $1 - u_j < b_j \le u_{j+1}$ und damit $u_j + u_{j+1} > 1$. Summiert man diese $k-1$ Ungleichungen auf, so erhält man

$(u_1 + u_2) + (u_2 + u_3) + ... + (u_{k-1} + u_k) = u_1 + 2u_2 + ... + 2u_{k-1} + u_k > k - 1$.

Auf beide Seiten werden die Füllhöhen des ersten und letzten Behälters addiert, und man erhält $2 \cdot \sum_{j=1}^{k} u_j > k - 1 + u_1 + u_k$. Die Summe der Füllhöhen in den Behältern ist gleich der Summe der Objekte, die gepackt wurden. Damit folgt

$k < 2 \cdot \sum_{j=1}^{k} u_j + 1 - (u_1 + u_k) = 2 \cdot \sum_{i=1}^{n} a_i + 1 - (u_1 + u_k) \le 2 \cdot \sum_{i=1}^{n} a_i + 1$ und $k \le 2 \cdot \sum_{i=1}^{n} a_i$.

Trivialerweise gilt $m^*(I) \ge \sum_{i=1}^{n} a_i$ (hier gilt „=“, wenn in einer optimalen Packung alle Behälter bis zur maximalen Füllhöhe 1 aufgefüllt werden). Damit ergibt sich schließlich $R_{\mathbf{NF}}(I) = k / m^*(I) \le 2$. ///

Die Grenze 2 im Nextfit-Algorithmus ist asymptotisch optimal: es gibt Instanzen *I*, für die $R_{\mathbf{NF}}(I)$ beliebig dicht an 2 herankommt. Dazu betrachte man etwa eine Eingabeinstanz *I* mit $n = 2m$ vielen Objekten, wobei *m* eine gerade Zahl ist: $I = [a_1, \ldots, a_{2m}]$. Die Werte a_i seien definiert durch $a_i = \begin{cases} 1/2 - 1/3m & \text{für ungerades } i \\ 1/m & \text{für gerades } i \end{cases}$. Mit dieser Instanz gilt $m(I, \mathbf{NF}(I)) = m$ und $m^*(I) = m/2 + 1$, so daß $R_{\mathbf{NF}}(I) = 2 \cdot \frac{m}{m+2}$ folgt, und dieser Wert kann für große *m* beliebig dicht an 2 herangehen.

Es gilt sogar eine genauere Abschätzung der relativen Approximationsgüte, falls die Größe aller Objekte beschränkt ist: Mit $a_{\max} = \max\{a_i \mid i = 1, \ldots, n\}$ ist

$$m(I, \mathbf{NF}(I)) \le \begin{cases} (1 + a_{\max} / (1 - a_{\max})) \cdot m^*(I) & \text{falls } 0 < a_{\max} \le 1/2 \\ 2 \cdot m^*(I) & \text{falls } 1/2 < a_{\max} \le 1 \end{cases}$$

Zu beachten ist, daß der Nextfit-Algorithmus ein online-Algorithmus ist, d.h. die Objekte der Reihenfolge nach inspiziert, und sofort eine Entscheidung trifft, in welchen Behälter ein Objekt zu legen ist, ohne alle Objekte gesehen zu haben. Die Laufzeit des Nextfit-Algorithmus bei einer Eingabeinstanz der Größe *n* liegt in $O(n)$.

Eine asymptotische Verbesserung der Approximation liefert der Firstfit-Algorithmus, der ebenfalls ein online-Algorithmus ist und bei geeigneter Implementierung ein Laufzeitverhalten der Ordnung $O(n \cdot \log(n))$ hat:

Firstfit-Algorithmus zur Approximation von Binpacking:

Eingabe: $I = [a_1, \ldots, a_n]$,
$a_1, \ldots, a_n$ („Objekte“) sind rationale Zahlen mit $0 < a_i \le 1$ für *i* = 1, ..., *n*

Verfahren: $B_1 := a_1$;
Sind $a_1, \ldots, a_i$ bereits in die Behälter $B_1, \ldots, B_j$ gelegt worden, ohne daß einer dieser Behälter überläuft, so lege a_{i+1} in den Behälter unter $B_1, \ldots, B_j$ mit dem kleinsten Index, in den das Objekt a_{i+1} noch paßt, ohne daß er überläuft. Falls

es einen derartigen Behälter unter $B_1, ..., B_j$ nicht gibt, lege a_{i+1} in einen neuen (bisher leeren) Behälter B_{j+1}.

Ausgabe: $\mathbf{FF}(I) =$ die benötigten nichtleeren Behälter $[B_1, ..., B_k]$.

Satz 6.1-2:

Ist $I = [a_1, \ldots, a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $m(I, \mathbf{FF}(I)) \leq 1{,}7 \cdot m^*(I) + 2$.

Der Beweis verwendet eine „Gewichtung“ der Objekte der Eingabeinstanz. Wegen der Länge des Beweises muß auf die Literatur verwiesen werden.

Die Grenze 1,7 im Firstfit-Algorithmus ist ebenfalls asymptotisch optimal: es gibt Instanzen *I* mit beliebig großem Wert $m^*(I)$, für die $m(I, \mathbf{FF}(I)) \geq 1{,}7 \cdot (m^*(I) - 1)$ gilt. Daher kommt $R_{\mathbf{FF}}(I)$ asymptotisch beliebig dicht an 1,7 heran.

Zu beachten ist weiterhin, daß der Firstfit-Algorithmus kein 1,7-approximativer Algorithmus ist, sondern nur asymptotisch 1,7-approximativ (siehe Kapitel 6.2) ist.

Eine weitere asymptotische Verbesserung erhält man, indem man die Objekte vor der Aufteilung auf Behälter nach absteigender Größe sortiert. Die entstehenden Approximationsalgorithmen sind dann jedoch offline-Algorithmen, da zunächst alle Objekte vor der Aufteilung bekannt sein müssen.

FirstfitDecreasing-Algorithmus zur Approximation von Binpacking:

Eingabe: $I = [a_1, \ldots, a_n]$,
$a_1, \ldots, a_n$ („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für *i* = 1, ..., *n*

Verfahren: Sortiere die Objekte $a_1, \ldots, a_n$ nach absteigender Größe. Die Objekte erhalten im folgenden wieder die Benennung $a_1, \ldots, a_n$, d.h. es gilt $a_1 \geq \ldots \geq a_n$.
$B_1 := a_1$;

Sind $a_1, ..., a_i$ bereits in die Behälter $B_1, ..., B_j$ gelegt worden, ohne daß einer dieser Behälter überläuft, so lege a_{i+1} in den Behälter unter $B_1, ..., B_j$ mit dem kleinsten Index, in den das Objekt a_{i+1} noch paßt, ohne daß er überläuft. Falls es einen derartigen Behälter unter $B_1, ..., B_j$ nicht gibt, lege a_{i+1} in einen neuen (bisher leeren) Behälter B_{j+1}.

Ausgabe: **FFD**$(I) =$ die benötigten nichtleeren Behälter $[B_1, ..., B_k]$.

Satz 6.1-3:
Ist $I = [a_1, ..., a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $m(I, \mathbf{FFD}(I)) \leq 1{,}5 \cdot m^*(I) + 1$.

Beweis:

Die Objekte der Eingabeinstanz $I = [a_1, ..., a_n]$ seien nach absteigender Größe sortiert, d.h. $a_1 \geq ... \geq a_n$. Sie werden in vier Größenordnungen eingeteilt. Dazu wird
$A = \{a_i \mid a_i \in I \text{ und } a_i > 2/3\}$,
$B = \{a_i \mid a_i \in I \text{ und } 2/3 \geq a_i > 1/2\}$,
$C = \{a_i \mid a_i \in I \text{ und } 1/2 \geq a_i > 1/3\}$ und
$D = \{a_i \mid a_i \in I \text{ und } 1/3 \geq a_i > 0\}$ gesetzt.
Der FirstfitDecreasing-Algorithmus habe eine Packung mit *k* Behältern ermittelt, d.h. $m(I, \mathbf{FFD}(I)) = k$. Die Füllhöhe des *j*-ten Behälters B_j sei u_j.

1. Fall: Es gibt mindestens einen Behälter, der nur Objekte aus *D* enthält.
Dann sind alle anderen Behälter, bis eventuell auf den Behälter B_k zu mindestens $2/3$ gefüllt. Es gilt dann
$\sum_{i=1}^{n} a_i = \sum_{j=1}^{k-1} u_j + u_k \geq \sum_{j=1}^{k-1} 2/3 = 2/3 \cdot (k-1)$ und folglich
$m(I, \mathbf{FFD}(I)) = k \leq 3/2 \cdot \sum_{i=1}^{n} a_i + 1 \leq 3/2 \cdot m^*(I) + 1$.

2. Fall: Kein Behälter enthält nur Objekte aus *D*.
Es sei $\tilde{I} = I \setminus D$. Alle Objekte in $\tilde{I}$ haben eine Größe von mehr als $1/3$. Dann gilt $\mathbf{FFD}(\tilde{I}) = \mathbf{FFD}(I)$, da die Objekte in *D* noch auf die übrigen Behälter verteilt werden können und erst dann verteilt werden, wenn alle Objekte in *A*, *B* und *C* verteilt sind.

Es gilt sogar $m(\tilde{I}, \mathbf{FFD}(\tilde{I})) = m^*(\tilde{I})$, d.h. der FirstfitDecreasing-Algorithmus liefert bei Eingabe von $\tilde{I}$ eine optimale Packung:

Dazu wird das Aussehen einer optimalen Packung von $\tilde{I}$ betrachtet:

- Es gibt t_A Behältern, die nur ein einziges Objekt aus *A* enthalten. Objekte aus *B* oder *C* passen dort nicht mehr hinein.
- Kein Behälter enthält 3 oder mehr Objekte.
- Es gibt *t* Behälter mit 2 Objekten, davon jeweils höchstens eines aus *B*, eventuell beide aus *C*. Die Anzahl der Behälter mit 2 Objekten, von denen eines aus *B* und eines aus *C* kommt, sei t_{BC}; die Anzahl der Behälter mit 2 Objekten, von denen beide aus *C* kommt, sei t_{CC}.
- Es gibt t_B Behälter, die nur ein Objekt aus *B* enthalten.
- Es gibt t_C Behälter, die nur ein Objekt aus *C* enthalten; $t_C \le 1$.

$m^*(\tilde{I}) = t_A + t_{BC} + t_{CC} + t_B + t_C$ und $|C| = t_{BC} + 2 \cdot t_{CC} + t_C$.

Bei der Abarbeitung von $\tilde{I}$ durch den FirstfitDecreasing-Algorithmus werden zuerst die Objekte aus $A \cup B$ gepackt; dazu werden $t_A + t_{BC} + t_B$ Behälter benötigt. Das erste Objekt $c \in C$ kommt in den Behälter, der ein Objekt aus *B* enthält, und zwar das größte Objekt $b \in B$ mit $b + c \le 1$. Ein Objekt aus *C* kommt erst dann in einen neuen Behälter oder in einen Behälter, der bereits ein Element aus *C* enthält, wenn es keinen Behälter gibt, der genau ein Element aus *B* enthält und zu dem man es legen könnte. Daher kommen auch t_{BC} Objekte aus *C* in Behälter mit einem Element aus *B*. Für die übrigen Objekte aus *C* benötigt der FirstfitDecreasing-Algorithmus noch $2 \cdot t_{CC} + t_C$ Behälter. Insgesamt ergibt sich

$m(\tilde{I}, \mathbf{FFD}(\tilde{I})) = t_A + t_{BC} + t_{CC} + t_B + t_C = m^*(\tilde{I})$.

Damit ergibt sich $m^*(I) \le m(I, \mathbf{FFD}(I)) = m(\tilde{I}, \mathbf{FFD}(\tilde{I})) = m^*(\tilde{I}) \le m^*(I)$; die letzte Ungleichung folgt aus der Inklusion $\tilde{I} \subseteq I$. Das bedeutet $m(I, \mathbf{FFD}(I)) = m^*(I)$. ///

Auch der FirstfitDecreasing-Algorithmus kein 1,5-approximativer Algorithmus, sondern nur asymptotisch 1,5-approximativ (siehe Kapitel 6.2).

Eine genauere Analyse des FirstfitDecreasing-Algorithmus zeigt, daß die in Satz 6.1-3 angegebene Schranke 1,5 verbessert werden kann. Es läßt sich zeigen, daß für jede Instanz $I = [a_1, \ldots, a_n]$ des Binpacking-Minimierungsproblems die Abschätzung $m(I, \mathbf{FFD}(I)) \le 11/9 \cdot m^*(I) + 4$ gilt. Das folgende Beispiel zeigt, daß die $11/9$-Grenze nicht verbessert werden kann: Man betrachte die Instanz *I*, die aus $n = 5m$ vielen Objekten besteht, und zwar *m* Objekte der Größe $1/2 + d$ mit $0 < d < 1/40$, *m* Objekte der Größe $1/4 + 2d$, *m* Objekte der Größe $1/4 + d$ und $2m$ Objekte der Größe $1/4 - 2d$. Die Objekte dieser Instanz

sind nach absteigender Größe sortiert. Es ist $m^*(I) = 3m/2$ und $m(I, \mathbf{FFD}(I)) = 11m/6$, also $m(I, \mathbf{FFD}(I)) = 11/9 \cdot m^*(I)$.

BestfitDecreasing-Algorithmus zur Approximation von Binpacking:

Eingabe: $I = [a_1, \ldots, a_n]$,
$a_1, \ldots, a_n$ („Objekte“) sind rationale Zahlen mit $0 < a_i \leq 1$ für $i = 1, \ldots, n$

Verfahren: Sortiere die Objekte $a_1, \ldots, a_n$ nach absteigender Größe. Die Objekte erhalten im folgenden wieder die Benennung $a_1, \ldots, a_n$, d.h. es gilt $a_1 \geq \ldots \geq a_n$.
$B_1 := a_1$;
Sind $a_1, \ldots, a_i$ bereits in die Behälter $B_1, \ldots, B_j$ gelegt worden, ohne daß einer dieser Behälter überläuft, so lege a_{i+1} in den Behälter unter $B_1, \ldots, B_j$, der den kleinsten freien Platz aufweist. Falls a_{i+1} in keinen der Behälter $B_1, \ldots, B_j$ paßt, lege a_{i+1} in einen neuen (bisher leeren) Behälter B_{j+1}.

Ausgabe: $\mathbf{BFD}(I) =$ die benötigten nichtleeren Behälter $[B_1, \ldots, B_k]$.

Satz 6.1-4:
Ist $I = [a_1, \ldots, a_n]$ eine Instanz des Binpacking-Minimierungsproblems, so gilt $m(I, \mathbf{BFD}(I)) \leq 11/9 \cdot m^*(I) + 4$.

Auch der BestfitDecreasing-Algorithmus nur asymptotisch $11/9$-approximativ (siehe Kapitel 6.2). Das obige Beispiel zeigt, daß auch für diesen Algorithmus die $11/9$-Grenze nicht verbessert werden kann.

Die folgende Zusammenstellung zeigt noch einmal die mit den verschiedenen Approximationsalgorithmen für das Binpacking-Problem zu erzielenden Approximationsgüten. In der letzten Zeile ist dabei ein Algorithmus erwähnt, der in einem gewissen Sinne (siehe Kapitel 6.2) unter allen approximativen Algorithmen mit polynomieller Laufzeit eine optimale relative Approximationsgüte erzielt. Zu beachten ist ferner, daß im Sinne der Definition die Algorithmen Firstfit, FirstfitDecreasing und BestfitDecreasing nicht *r*-approximativ (mit *r* = 1,7 bzw. *r* = 1,5 bzw. *r* = 1,22), sondern nur asymptotisch *r*-approximativ sind.

Approximationsalgorithmus	Approximationsgüte
Nextfit	$m(I,\mathbf{NF}(I)) \le \begin{cases} (1+a_{\max}/(1-a_{\max}))\cdot m^*(I) & \text{falls } 0 < a_{\max} \le 1/2 \\ 2\cdot m^*(I) & \text{falls } 1/2 < a_{\max} \le 1 \end{cases}$
Firstfit	$m(I,\mathbf{FF}(I)) \le 1{,}7\cdot m^*(I)+2$
FirstfitDecreasing	$m(I,\mathbf{FFD}(I)) \le 1{,}5\cdot m^*(I)+1$ (Satz 6.1-3) $m(I,\mathbf{FFD}(I)) \le 11/9\cdot m^*(I)+4$; $11/9 = 1{,}222$
BestFitDecreasing	$m(I,\mathbf{BFD}(I)) \le 11/9\cdot m^*(I)+4$
Simchi-Levi, 1994	$m(I,\mathbf{SL}(I)) \le 1{,}5\cdot m^*(I)$

In Kapitel 6 wurde gezeigt, daß das 0/1-Rucksack-Maximierungsproblem unter der Annahme $\mathbf{P} \neq \mathbf{NP}$ nicht absolut approximierbar ist. Es gibt jedoch für dieses Problem einen 2-approximativen Algorithmus:

Das 0/1-Rucksackproblem als Maximierungsproblem
(maximum 0/1 knapsack problem)

Instanz: 1. $I = (A, M)$

$A = \{a_1, ..., a_n\}$ ist eine Menge von n Objekten und $M \in \mathbf{R}_{>0}$ die „Rucksackkapazität". Jedes Objekt a_i, $i = 1, ..., n$, hat die Form $a_i = (w_i, p_i)$; hierbei bezeichnet $w_i \in \mathbf{R}_{>0}$ das Gewicht und $p_i \in \mathbf{R}_{>0}$ den Wert (Profit) des Objekts a_i.

$size(I) = n$

2. $\text{SOL}(I) = \left\{ (x_1, ..., x_n) \mid x_i = 0 \text{ oder } x_i = 1 \text{ für } i = 1, ..., n \text{ und } \sum_{i=1}^{n} x_i \cdot w_i \le M \right\}$; man beachte, daß hier nur Werte $x_i = 0$ oder $x_i = 1$ zulässig sind

3. $m(I,(x_1, ..., x_n)) = \sum_{i=1}^{n} x_i \cdot p_i$ für $(x_1, ..., x_n) \in \text{SOL}(I)$

4. *goal* = max

Lösung: Eine Folge $x_1^*, ..., x_n^*$ von Zahlen mit

(1) $x_i^* = 0$ oder $x_i^* = 1$ für $i = 1, ..., n$

(2) $\sum_{i=1}^{n} x_i^* \cdot w_i \le M$

(3) $m\left(I,\left(x_1^*,...,x_n^*\right)\right)=\sum_{i=1}^{n} x_i^* \cdot p_i$ ist maximal unter allen möglichen Auswahlen $x_1,...,x_n$, die (1) und (2) erfüllen.

Der Approximationsalgorithmus **RUCK_APP** wird hier informell beschrieben:

1. Bei Eingabe einer Instanz $I=(A,M)$ sortiere man die Objekte nach absteigenden Werten p_i/w_i. Die Objekte werden in der durch diese Sortierung bestimmten Reihenfolge bearbeitet. Ein Objekt a_i wird dabei in den Rucksack gelegt, wenn es noch paßt; in diesem Fall wird $x_i=1$ gesetzt. Paßt das Objekt a_i nicht, dann wird $x_i=0$ gesetzt und zum nächsten Objekt übergegangen.
2. Es sei $p_{\max}=\max\{p_i \mid i=1,...,n\}$. Man vergleiche das Ergebnis im Schritt 1 mit der Rucksackfüllung, die man erhält, wenn man nur das Objekt mit dem Gewinn $p_{\max}$ allein in den Rucksack legt. Man nehme diejenige Rucksackfüllung, die den größeren Wert der Zielfunktion liefert.

Das Ergebnis des Verfahrens ist eine 0-1-Folge **RUCK_APP**$(I)=\left(x_1,...,x_n\right)$ mit dem Wert $m\left(I,\textbf{RUCK\_APP}(I)\right)$ der Zielfunktion.

Satz 6.1-5:
Für jede Instanz $I=(A,M)$ des 0/1-Rucksack-Maximierungsproblems gilt: $1\le m^*(I)/m\left(I,\textbf{RUCK\_APP}(I)\right)\le 2$, d.h. das 0/1-Rucksack-Maximierungsproblem liegt in **APX**.

Beweis:

Es sei a_j das erste Objekt, das im 1. Teil des Algorithmus **RUCK_APP** nicht mehr in den Rucksack paßt. Zu diesem Zeitpunkt hat der Rucksack eine Füllung $\overline{w}=\sum_{i=1}^{j-1} w_i \le M$. Es gilt also $w_j > M-\overline{w}$. Der bisher entstandene Profit ist $\overline{p}=\sum_{i=1}^{j-1} p_i$.

Es werde eine modifizierte Aufgabenstellung des 0/1-Rucksack-Maximierungsproblems betrachtet, in dem die Forderung „$x_i=0$ oder $x_i=1$“ durch „$0\le x_i\le 1$“ ersetzt ist. Für die so modifizierte Aufgabenstellung kann im 1. Teil des Algorithmus **RUCK_APP** das Objekt a_j noch in den Rucksack gelegt werden, jedoch nur mit einem Anteil $x_j=\left(M-\overline{w}\right)/w_j$. Der Rucksack ist dann gefüllt, d.h. im 1. Teil von **RUCK_APP** wird für die modifizierte Aufga-

benstellung $x_{j+1} = ... = x_n = 0$ gesetzt. Es läßt sich zeigen, daß jetzt bereits eine optimale Lösung des modifizierten 0/1-Rucksack-Maximierungsproblems gefunden ist, und zwar mit einem Profit $m_{MOD} = \overline{p} + \left((M - \overline{w})/w_j\right) \cdot p_j$. Da jede zulässige Lösung des (ursprünglichen) 0/1-Rucksack-Maximierungsproblems eine zulässige Lösung des modifizierten 0/1-Rucksack-Maximierungsproblems ist, folgt $m^*(I) \le \overline{p} + \left((M - \overline{w})/w_j\right) \cdot p_j < \overline{p} + p_j$ (die letzte Ungleichung ergibt sich aus $w_j > M - \overline{w}$).

Außerdem gilt $\overline{p} \le \max\{\overline{p}, p_{\max}\} \le m(I, \mathbf{RUCK\_APP}(I)) \le m^*(I)$.

Es werden zwei Fälle unterschieden:

1.Fall: $p_j \le \overline{p}$

Dann gilt $m^*(I) < 2 \cdot \overline{p} \le 2 \cdot m(I, \mathbf{RUCK\_APP}(I))$.

2.Fall: $p_j > \overline{p}$

Dann gilt $p_{\max} \ge p_j > \overline{p}$ und $m^*(I) < \overline{p} + p_j < 2 \cdot p_{\max} \le 2 \cdot m(I, \mathbf{RUCK\_APP}(I))$.

In beiden Fällen folgt die Abschätzung $1 \le m^*(I)/m(I, \mathbf{RUCK\_APP}(I)) \le 2$. ///

Gibt man die Instanz $I = \left(\left(\frac{M}{2}+1, \frac{M}{2}+2\right), \left(\frac{M}{2}, \frac{M}{2}\right), \left(\frac{M}{2}, \frac{M}{2}\right), M\right)$ mit $M > 4$ in den Algorithmus **RUCK_APP** ein, so liefert er das Ergebnis $x_1 = 1$, $x_2 = 0$ und $x_3 = 0$ und den Profit $m(I, \mathbf{RUCK\_APP}(I)) = M/2 + 2$. Die optimale Lösung ist jedoch Ergebnis $x_1^* = 0$, $x_2^* = 1$ und $x_3^* = 1$ mit dem Profit $m^*(I) = M > M/2 + 2$. Damit ist

$$m^*(I) \Big/ m(I, \mathbf{RUCK\_APP}(I)) = \frac{M}{M/2+2} = \frac{2M}{M+4} = \frac{2 \cdot (M+4) - 8}{M+4} = 2 - \frac{8}{M+4}.$$

Bei genügend großem *M* kommt dieser Wert beliebig dicht an 2 heran, so daß die Abschätzung in Satz 6.1-5 nicht verbessert werden kann.

Der folgende Satz zeigt, daß es unter der Voraussetzung $\mathbf{P} \neq \mathbf{NP}$ nicht für jedes Optimierungsproblem aus **NPO** einen approximativen Algorithmus gibt. Dazu sei noch einmal das Handlungsreisenden-Minimierungsproblem mit ungerichteten Graphen angegeben:

Das Handlungsreisenden-Minimierungsproblem

Instanz: 1. $G=(V,E,w)$

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

2. $\mathrm{SOL}(G)=\{T \mid T=\langle (v_{i_1},v_{i_2}),(v_{i_2},v_{i_3}),...,(v_{i_{n-1}},v_{i_n}),(v_{i_n},v_{i_1})\rangle$ ist eine Tour durch $G\}$
3. für $T\in \mathrm{SOL}(G)$, $T=\langle (v_{i_1},v_{i_2}),(v_{i_2},v_{i_3}),...,(v_{i_{n-1}},v_{i_n}),(v_{i_n},v_{i_1})\rangle$, ist die Zielfunktion definiert durch $m(G,T)=\sum_{j=1}^{n-1} w(v_{i_j},v_{i_{j+1}})+w(v_{i_n},v_{i_1})$
4. $goal=\min$

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

Satz 6.1-6:

Ist $\mathbf{P}\neq\mathbf{NP}$, so gibt es keinen r-approximativen Algorithmus für das Handlungsreisenden-Minimierungsproblem (mit $r\in\mathbf{R}_{\geq 1}$).

Ist $\mathbf{P}\neq\mathbf{NP}$, so ist $\mathbf{APX}\subset\mathbf{NPO}$.

Beweis:

Die Argumentation folgt der Idee aus dem Beweis von Satz 6-3.

Es wird angenommen, daß es bei Vorgabe des Wertes $r\in\mathbf{R}_{\geq 1}$ einen r-approximativen Algorithmus **A** für das Handlungsreisenden-Minimierungsproblem gibt und zeigt dann, daß dieser (mit einer einfachen Modifikation) dazu verwendet werden kann, das Problem des Hamiltonschen Kreises in einem ungerichteten Graphen (UNGERICHTETER HAMILTONKREIS, vgl. Kapitel 5.4) in polynomieller Zeit zu entscheiden. Da dieses **NP**-vollständig ist, folgt $\mathbf{P}=\mathbf{NP}$ (vgl. Satz 5.1-1).

Das Problem des Hamiltonschen Kreises in einem ungerichteten Graphen (UNGERICHTETER HAMILTONKREIS) lautet wie folgt:

Instanz: G,
$G=(V,E)$ ist ein ungerichteter Graph mit $V=\{v_1,\ldots,v_n\}$.

Lösung: Entscheidung „ja“, falls gilt:
G besitzt einen Hamiltonschen Kreis. Dieses ist eine Anordnung $(v_{p(1)},v_{p(2)},\ldots,v_{p(n)})$ der Knoten mittels einer Permutation π der Knotenindizes, so daß für $i=1,\ldots,n-1$ gilt: $(v_{p(i)},v_{p(i+1)})\in E$ und $(v_{p(n)},v_{p(1)})\in E$.

Es sei **A** ein r-approximativen Algorithmus **A** für das Handlungsreisenden-Minimierungsproblem, d.h. für alle Instanzen I dieses Problems gilt $m(I,\mathbf{A}(I))\le r\cdot m^*(I)$. Man kann $r>1$ annehmen, denn für $r=1$ ermittelt **A** bereits eine optimale Lösung. Es wird ein Algorithmus $\tilde{\mathbf{A}}$ für UNGERICHTETER HAMILTONKREIS konstruiert, der folgendendermaßen arbeitet:

Bei Eingabe eines ungerichteten Graphen $G=(V,E)$ mit $V=\{v_1,\ldots,v_n\}$ konstruiert $\tilde{\mathbf{A}}$ einen vollständigen bewerteten ungerichteten Graphen $\tilde{G}=(V,\tilde{E})$ (die Knotenmenge wird beibehalten) mit $\tilde{E}=\{(v_i,v_j)\mid v_i\in V \text{ und } v_j\in V\}$ und der Kantenbewertung $w:\tilde{E}\to\mathbf{R}_{\ge 0}$, die durch

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

definiert ist. Diese Konstruktion erfolgt in polynomieller Zeit, da höchsten $O(n^2)$ viele Kanten hinzuzufügen und $O(n^2)$ viele Kanten zu bewerten sind. Alle Bewertungen haben eine Länge der Ordnung $O(\log(n))$. Der Graph $\tilde{G}=(V,\tilde{E})$ wird in den Algorithmus **A** eingegeben und das Ergebnis $m(\tilde{G},\mathbf{A}(\tilde{G}))$ mit dem Wert $r\cdot n$ verglichen. $\tilde{\mathbf{A}}$ trifft die Entscheidung

$$\tilde{\mathbf{A}}(G)=\begin{cases}\texttt{ja} & \text{für } m(\tilde{G},\mathbf{A}(\tilde{G}))\le r\cdot n\\ \texttt{nein} & \text{für } m(\tilde{G},\mathbf{A}(\tilde{G}))> r\cdot n\end{cases}.$$

Falls **A** ein polynomiell zeitbeschränkter Algorithmus ist, dann auch $\tilde{\mathbf{A}}$.

Besitzt G einen Hamiltonkreis (mit Länge n), dann ist $m^*(\tilde{G})=n$, und $\tilde{\mathbf{A}}(G)=\texttt{ja}$, da **A** ein r-approximativen Algorithmus **A** für das Handlungsreisenden-Minimierungsproblem ist und damit $m(\tilde{G},\mathbf{A}(\tilde{G}))\le r\cdot m^*(\tilde{G})=r\cdot n$ ist.

Besitzt G keinen Hamiltonkreis, dann enthält jeder Hamiltonkreis in $\tilde{G}$ mindestens eine Kante, die mit $r\cdot n$ bewertet ist (da $\tilde{G}$ ein vollständiger Graph ist, enthält $\tilde{G}$ einen Hamiltonkreis). Damit ergibt sich $m(\tilde{G},\mathbf{A}(\tilde{G}))\ge m^*(\tilde{G})\ge n-1+r\cdot n>r\cdot n$, d.h. $\tilde{\mathbf{A}}(G)=\texttt{nein}$.

In beiden Fällen trifft $\tilde{\mathbf{A}}$ die korrekte Entscheidung. ///

Das **metrische Handlungsreisenden-Minimierungsproblem** liegt jedoch in **APX**. Dieses stellt eine zusätzliche Bedingung an die Gewichtsfunktion einer Eingabeinstanz, nämlich die Gültigkeit der **Dreiecksungleichung**:

Für $v_i \in V$, $v_j \in V$ und $v_k \in V$ gilt $w(v_i, v_j) \leq w(v_i, v_k) + w(v_k, v_j)$.

Auch hierbei ist das zugehörige Entscheidungsproblem **NP**-vollständig. Es gibt (unabhängig von der Annahme $\mathbf{P} \neq \mathbf{NP}$) im Gegensatz zum (allgemeinen) Handlungsreisenden-Minimierungsproblem für dieses Problem einen 1,5-approximativen Algorithmus (siehe Literatur). Es ist nicht bekannt, ob es einen Approximationsalgorithmus mit einer kleineren relativen Approximatiosgüte gibt oder ob aus der Existenz eines derartigen Algorithmus bereits $\mathbf{P} = \mathbf{NP}$ folgt.

Hat man für ein Optimierungsproblem aus **APX** einen *r*-approximativen Algorithmus gefunden, so stellt sich die Frage, ob dieser noch verbessert werden kann, d.h. ob es einen *t*-approximativen Algorithmus mit $1 \leq t < r$ gibt. Der folgende Satz (gap theorem) besagt, daß man unter Umständen an Grenzen stößt, daß es nämlich Optimierungsprobleme in **APX** gibt, die in polynomieller Zeit nicht beliebig dicht approximiert werden können, außer es gilt $\mathbf{P} = \mathbf{NP}$.

Satz 6.1-7:

Es sei Π' ein **NP**-vollständiges Entscheidungsproblem über Σ'^* und Π aus **NPO** ein Minimierungsproblem über Σ^*. Es gebe zwei in polynomieller Zeit berechenbare Funktionen $f : \Sigma'^* \to \Sigma^*$ und $c : \Sigma'^* \to \mathbf{N}$ und eine Konstante $e > 0$ mit folgender Eigenschaft:

Für jedes $x \in \Sigma'^*$ ist $m^*(f(x)) = \begin{cases} c(x) & \text{für } x \in L_{\Pi'} \\ c(x) \cdot (1+e) & \text{für } x \notin L_{\Pi'} \end{cases}$.

Dann gibt es keinen polynomiell zeitbeschränkten *r*-approximativen Algorithmus für Π mit $r < 1+e$, außer $\mathbf{P} = \mathbf{NP}$.

Beweis:

Angenommen, es gibt einen polynomiell zeitbeschränkten *r*-approximativen Algorithmus **A** für Π mit $r < 1+e$. Dann kann man daraus einen polynomiell zeitbeschränkten Algorithmus $\mathbf{A}'$ für Π' konstruieren, der genau $L_{\Pi'}$ erkennt. Das bedeutet $\mathbf{P} = \mathbf{NP}$.

Die Arbeitsweise von $\mathbf{A}'$ wird informell beschrieben:

Bei Eingabe von $x \in \Sigma'^{*}$ in $\mathbf{A}'$ werden in polynomieller Zeit die Werte $f(x)$ und $c(x)$ berechnet. Der Wert $f(x)$ wird in den polynomiell zeitbeschränkten *r*-approximativen Algorithmus **A** für Π eingegeben und der Näherungswert $m(f(x), \mathbf{A}(f(x)))$ mit $c(x) \cdot (1+e)$ verglichen. $\mathbf{A}'$ trifft die Entscheidung

$$\mathbf{A}'(x) = \begin{cases} \texttt{ja} & \text{für } m(f(x), \mathbf{A}(f(x))) < c(x) \cdot (1+e) \\ \texttt{nein} & \text{für } m(f(x), \mathbf{A}(f(x))) \geq c(x) \cdot (1+e) \end{cases}.$$

$\mathbf{A}'$ ist ein polynomiell zeitbeschränkter Entscheidungsalgorithmus. Zu zeigen ist, daß die von $\mathbf{A}'$ getroffene Entscheidung korrekt ist, d.h. daß $\mathbf{A}'(x) = \texttt{ja}$ genau dann gilt, wenn $x \in L_{\Pi'}$ ist.

Es sei $x \in L_{\Pi'}$. Da **A** *r*-approximativ ist, gilt $\frac{m(f(x), \mathbf{A}(f(x)))}{m^{*}(f(x))} \leq r < 1+e$. Wegen $x \in L_{\Pi'}$ ist $m^{*}(f(x)) = c(x)$. Also ist $m(f(x), \mathbf{A}(f(x))) < m^{*}(f(x)) \cdot (1+e) = c(x) \cdot (1+e)$, und $\mathbf{A}'(x) = \texttt{ja}$.

Es sei $x \notin L_{\Pi'}$. Dann folgt $m^{*}(f(x)) = c(x) \cdot (1+e)$,
$m(f(x), \mathbf{A}(f(x))) \geq m^{*}(f(x)) = c(x) \cdot (1+e)$ und $\mathbf{A}'(x) = \texttt{nein}$.

In beiden Fällen trifft $\mathbf{A}'$ die korrekte Entscheidung. ///

Bemerkung: Der Beweis von Satz 6.1-7 zeigt, daß Satz 6.1-7 gültig bleibt, wenn die dort formulierte Voraussetzung über $m^{*}(f(x))$ ersetzt durch die Voraussetzung

$$\text{Für jedes } x \in \Sigma'^{*} \text{ ist } m^{*}(f(x)) \begin{cases} = c(x) & \text{für } x \in L_{\Pi'} \\ \geq c(x) \cdot (1+e) & \text{für } x \notin L_{\Pi'} \end{cases}.$$

Ein entsprechender Satz gilt für Maximierungsprobleme:

Satz 6.1-8:

Es sei Π' ein **NP**-vollständiges Entscheidungsproblem über Σ'^* und Π aus **NPO** ein Maximierungsproblem über Σ^*. Es gebe zwei in polynomieller Zeit berechenbare Funktionen $f:\Sigma'^* \to \Sigma^*$ und $c:\Sigma'^* \to \mathbf{N}$ und eine Konstante $e>0$ mit folgender Eigenschaft:

Für jedes $x \in \Sigma'^*$ ist $m^*(f(x)) = \begin{cases} c(x) & \text{für } x \in L_{\Pi'} \\ c(x)\cdot(1-e) & \text{für } x \notin L_{\Pi'} \end{cases}$.

Dann gibt es keinen polynomiell zeitbeschränkten *r*-approximativen Algorithmus für Π mit $r < 1/(1-e)$, außer $\mathbf{P} = \mathbf{NP}$.

Mit Hilfe dieser Sätze läßt sich zeigen beispielsweise, daß die Grenze *r* = 1,5 für einen *r*-approximativen (polynomiell zeitbeschränkten) Algorithmus unter der Annahme $\mathbf{P} \neq \mathbf{NP}$ für das Binpacking-Minimierungsproblem optimal ist:

Satz 6.1-9:

Ist $\mathbf{P} \neq \mathbf{NP}$, dann gibt es keinen *r*-approximativen Algorithmus für das Binpacking-Minimierungsproblem mit $r \leq 3/2 - e$ für $e > 0$.

Beweis:

In Satz 6.1-7 übernimmt das Binpacking-Minimierungsproblem die Rolle von Π; für Π' wird das **NP**-vollständige Partitionenproblem (siehe Kapitel 5.4) genommen. Es wird definiert durch

Instanz: $I = [a_1, \ldots, a_n]$,
$a_1, \ldots, a_n$ sind natürliche Zahlen.

Lösung: Entscheidung „ja“ genau dann, wenn gilt:
Es gibt eine Aufteilung der Menge $\{a_1, \ldots, a_n\}$ in zwei disjunkte Teile I_1 und I_2 mit $\sum_{a_i \in I_1} a_i = \sum_{a_i \in I_2} a_i$.

Es werden zwei Abbildungen *f* und *c* definiert, die den Bedingungen in Satz 6.1-7 genügen. Die Abbildung *f* ordnet jeder Instanz $I = [a_1, \ldots, a_n]$ des Partitionenproblems eine Instanz $f(I)$ des Binpacking-Minimierungsproblems zu.

Die Funktion c mit $c(I)=2$ für alle Instanzen I des Partitionenproblems ist trivialerweise in polynomieller Zeit berechenbar.

Für eine Instanz $I=[a_1,...,a_n]$ des Partitionenproblems sei $B=\sum_{a_i\in I} a_i$. Es wird $f(I)=[(2a_1)/B,...,(2a_n)/B]$ definiert, falls sich dadurch eine Instanz des Binpacking-Minimierungsproblems ergibt, d.h. falls $(2a_i)/B\le 1$ für $i=1,...,n$ gilt. Ansonsten wird $f(I)=[1,1,1]$ gesetzt. In jedem Fall ist $f(I)$ eine Instanz des Binpacking-Minimierungsproblems. Da dieses in **NPO** liegt und wegen Satz 2.1-3 ist $f(I)$ in polynomieller Zeit berechenbar.

Ist $I\in L_{\Pi'}$, d.h. I ist eine „ja"-Instanz des Partitionenproblems bzw. es gibt eine Aufteilung der Menge $\{a_1,...,a_n\}$ in zwei disjunkte Teile I_1 und I_2 mit $\sum_{a_i\in I_1} a_i=\sum_{a_i\in I_2} a_i$, dann gilt $a_i\le B/2$ für $i=1,...,n$, denn sonst könnte man I nicht in zwei gleichgroße Teile aufteilen. Daher ist $(2a_i)/B\le 1$, $\sum_{a_i\in I_1} a_i=\sum_{a_i\in I_2} a_i=B/2$ und $\sum_{a_i\in I_1}(2a_i)/B=\sum_{a_i\in I_2}(2a_i)/B=1$. Zur Packung der Instanz $f(I)=[(2a_1)/B,...,(2a_n)/B]$ des Binpacking-Minimierungsproblems sind genau 2 Behälter erforderlich, d.h. $m^*(f(I))=2=c(I)$..

Ist $I\notin L_{\Pi'}$ und $[(2a_1)/B,...,(2a_n)/B]$ keine Instanz des Binpacking-Minimierungsproblems, d.h. es gibt mindestens ein a_i mit $(2a_i)/B>1$, dann ist $f(I)=[1,1,1]$ und $m^*(f(I))=3$.
Ist $I\notin L_{\Pi'}$ und $[(2a_1)/B,...,(2a_n)/B]$ eine Instanz des Binpacking-Minimierungsproblems, dann ist $f(I)=[(2a_1)/B,...,(2a_n)/B]$ und $m^*(f(I))\ge 3$.
In beiden Fällen gilt $m^*(f(I))\ge 3=2\cdot\frac{3}{2}=c(I)\cdot\left(1+\frac{1}{2}\right)$.

Aus Satz 6.1-7 zusammen mit der sich dort anschließenden Bemerkung folgt, daß es keinen polynomiell zeitbeschränkten r-approximativen Algorithmus für das Binpacking-Minimierungsproblem mit $r<1+1/2=1.5$ gibt, außer $\mathbf{P}=\mathbf{NP}$. ///

6.2 Polynomiell zeitbeschränkte und asymptotische Approximationsschemata

In vielen praktischen Anwendungen möchte man die relative Approximationsgüte verbessern. Dabei ist man sogar bereit, längere Laufzeiten der Approximationsalgorithmen in Kauf zu nehmen, solange sie noch polynomielles Laufzeitverhalten bezüglich der Größe der Eingabeinstanzen haben. Bezüglich der relativen Approximationsgüte r akzeptiert man eventuell

sogar ein Laufzeitverhalten, das exponentiell von $1/(r-1)$ abhängt: Je besser die Approximation ist, um so größer ist die Laufzeit. In vielen Fällen kann man so eine optimale Lösung beliebig dicht approximieren, allerdings zum Preis eines dramatischen Anstiegs der Rechenzeit.

Es sei Π ein Optimierungsproblem aus **NPO**. Ein Algorithmus **A** heißt **polynomiell zeitbeschränktes Approximationsschema** (polynomial-time approximation scheme) für Π, wenn er für jede Eingabeinstanz *x* von Π und für jede rationale Zahl *r* > 1 bei Eingabe von (x,r) eine *r*-approximative Lösung für *x* in einer Laufzeit liefert, die polynomiell von *size*(*x*) abhängt.

Mit **PTAS** werde die Klasse der Optimierungsprobleme in **NPO** bezeichnet, für die es ein polynomiell zeitbeschränktes Approximationsschema gibt. Daß diese Klasse nicht leer ist, zeigt das folgende Berispiel.

Partitionen-Minimierungsproblem

Instanz:
1. $I = [a_1, \ldots, a_n]$
   $a_1, \ldots, a_n$ („Objekte“) sind natürliche Zahlen mit $a_i > 0$ für *i* = 1, ..., *n*
   size(*I*) = *n*
2. $\mathrm{SOL}(I) = \{[Y_1, Y_2] \mid [Y_1, Y_2] \text{ ist eine Partition (disjunkte Zerlegung) von } I\}$
3. Für $[Y_1, Y_2] \in \mathrm{SOL}(I)$ ist $m(I, [Y_1, Y_2]) = \max\left\{\sum_{a_i \in Y_1} a_i, \sum_{a_j \in Y_2} a_j\right\}$
4. *goal* = min

Lösung: Eine Partition der Objekte in zwei Teile $[Y_1, Y_2]$, so daß sich die Summen der Objekte in beiden Teilen möglichst wenig unterscheiden.

Der folgende in Pseudocode formulierte Algorithmus ist ein polynomiell zeitbeschränktes Approximationsschema für das Partitionen-Minimierungsproblem.

Polynomiell zeitbeschränktes Approximationsschema für das Partitionen-Minimierungsproblem

Eingabe: $I = [a_1, \ldots, a_n]$, rationale Zahl *r* > 1,
$a_1, \ldots, a_n$ („Objekte“) sind natürliche Zahlen mit $a_i > 0$ für *i* = 1, ..., *n*

Verfahren:
```
VAR Y1 : SET OF INTEGER;
    Y2 : SET OF INTEGER;
    k  : REAL;
```

```
      j : INTEGER;

BEGIN
  IF r >= 2
   THEN BEGIN
          Y1 := {a1,...,an};
          Y2 := { };
        END
   ELSE BEGIN
          Sortiere die Objekte nach absteigender Größe; die dabei
          entstehende Folge sei (x1,...,xn);
          k := ⌈(2−r)/(r−1)⌉;
          { Phase 1: }
          finde eine optimale Partition [Y1,Y2] für [x1,...,xk];
          { Phase 2: }
          FOR j := k+1 TO n DO
            IF ∑xi∈Y1 xi <= ∑xi∈Y2 xi
            THEN Y1 := Y1 ∪ {xj}
            ELSE Y2 := Y2 ∪ {xj}
        END;
```

Ausgabe: $[Y_1, Y_2]$.

Satz 6.2-1:
Das Partitionen-Minimierungsproblem liegt in **PTAS**.

Beweis:

Es ist zu zeigen, daß der angegebene Algorithmus, der mit **A** bezeichnet werde, bei Eingabe einer Instanz $I = [a_1, \ldots, a_n]$ für das Partitionen-Minimierungsproblem und einer rationalen Zahl $r > 1$ eine r-approximative Lösung für I in einer Laufzeit liefert, die polynomiell von n abhängt.

Es werden $I = [a_1, \ldots, a_n]$ und r in den angegebenen Algorithmus eingegeben. Die Ausgabe sei $[Y_1, Y_2]$. O.B.d.A. sei $\sum_{a_i \in Y_1} a_i \geq \sum_{a_j \in Y_2} a_j$. Dann ist $m(I, \mathbf{A}(I)) = \sum_{a_i \in Y_1} a_i$. Es sei $w(I)$ die

Summe aller Objekte der Eingabeinstanz I: $w(I)=\sum_{a_i\in I} a_i$, $w(Y_2)$ die Summe aller Objekte in Y_2: $w(Y_2)=\sum_{a_i\in Y_2} a_i$.

1. Fall: $r \geq 2$

Dann ist $m^*(I) \geq w(I)/2 \geq m(I,\mathbf{A}(I))/2$, also $m(I,\mathbf{A}(I)) \leq 2 \cdot m^*(I) \leq r \cdot m^*(I)$.

2. Fall: $1 \leq r < 2$

Es sei a_h das letzte zu Y_1 hinzugefügte Objekt.

Wurde a_h in Phase 1 des Algorithmus in Y_1 aufgenommen, so läßt sich $m(I,\mathbf{A}(I)) = m^*(I)$ zeigen. Es wird daher angenommen, daß a_h in Phase 2 des Algorithmus in Y_1 aufgenommen wurde. Es folgt nacheinander:

$m(I,\mathbf{A}(I)) - a_h \leq w(Y_2)$,

$2 \cdot m(I,\mathbf{A}(I)) - a_h \leq w(Y_2) + m(I,\mathbf{A}(I)) = w(I)$ und

$m(I,\mathbf{A}(I)) - w(I)/2 \leq a_h/2$.

Außerdem gilt wegen der Sortierung der Objekte $a_h \leq a_j$ für $j = 1,...,k$. Diese Ungleichungen werden auf a_h aufsummiert zu dem Ergebnis:

$(k+1) \cdot a_h = a_h + k \cdot a_h \leq \sum_{j=1}^{k} a_j + a_h \leq w(I)$, also $a_h \leq w(I)/(k+1)$.

Es gilt $m(I,\mathbf{A}(I)) \geq w(I)/2 \geq w(Y_2)/2$ und $m^*(I) \geq w(I)/2$.

Insgesamt ergibt sich:

$$\frac{m(I,\mathbf{A}(I))}{m^*(I)} \leq \frac{m(I,\mathbf{A}(I))}{w(I)/2} \leq \frac{a_h/2 + w(I)/2}{w(I)/2} = 1 + \frac{a_h}{w(I)} \leq 1 + \frac{1}{k+1} \leq r;$$

die letzte Ungleichung folgt aus der Wahl von *k*, nämlich $k = \lceil (2-r)/(r-1) \rceil$:

$k \geq (2-r)/(r-1)$, d.h. $k+1 \geq= (2-r+r-1)/(r-1) = 1/(r-1) > 1/r$.

Mit $k(r) = \lceil (2-r)/(r-1) \rceil$ ist das Laufzeitverhalten von der Ordnung $O(n \cdot \log(n) + n^{k(r)})$. Der erste Term gibt die Laufzeit zum Sortieren der Objekte der Eingabeinstanz an, der zweite Term die Laufzeit zur Ermittlung einer optimalen Lösung für die ersten *k* Elemente in Phase 1. Die Laufzeit für Phase 2 ist von der Ordnung $O(n)$. Da $k(r) \in O(1/(r-1))$ ist das Laufzeitverhalten bei festem *r* polynomiell in der Größe *n* der Eingabeinstanz, jedoch exponentiell in *n* und $1/(r-1)$. ///

Offensichtlich ist **PTAS** $\subseteq$ **APX**. In Kapitel 6.1 wurde erwähnt, daß es unter der Annahme **P** $\neq$ **NP** keinen *r*-approximativen Algorithmus für das Binpacking-Minimierungsproblem mit $r \leq 3/2 - e$ für $e > 0$ gibt. Daraus folgt:

Satz 6.2-2:
Ist $\mathbf{P} \neq \mathbf{NP}$, so gilt $\mathbf{PTAS} \subset \mathbf{APX} \subset \mathbf{NPO}$.

Es gibt in **PTAS** Optimierungsprobleme, die ein polynomiell zeitbeschränktes Approximationsschema **A** zulassen, das bei Eingabe einer Instanz x von Π und einer rationalen Zahl $r > 1$ eine r-approximative Lösung für x in einer Laufzeit liefert, die polynomiell von $|x|$ und $1/(r-1)$ abhängt.
Einen derartigen Algorithmus nennt man **voll polynomiell zeitbeschränktes Approximationsschema** (fully polynomial-time approximation scheme). Die Optimierungsprobleme, die einen derartigen Algorithmus zulassen, bilden die Klasse **FPTAS**.

In der Literatur werden Beispiele für Optimierungsprobleme genannt, die in **FPTAS** liegen.

Es läßt sich zeigen:

Satz 6.2-3:
Ist $\mathbf{P} \neq \mathbf{NP}$, so gilt $\mathbf{FPTAS} \subset \mathbf{PTAS} \subset \mathbf{APX} \subset \mathbf{NPO}$.

In Kapitel 6.1 wurden mehrere Approximationsalgorithmen für das Binpacking-Minimierungsproblem angegeben. Die Approximationsgüte $m(I, \mathbf{BFD}(I)) \leq 11/9 \cdot m^*(I) + 4$ von BestfitDecreasing zeigt, daß man (unabhängig von der Annahme $\mathbf{P} \neq \mathbf{NP}$) durchaus einen Approximationsalgorithmus entwerfen kann, dessen relative Approximationsgüte unterhalb der überhaupt für einen r-approximativen Algorithmus möglichen Untergrenze liegt. Eventuell gibt es sogar in einem erweiterten Sinne ein polynomiell zeitbeschränktes Approximationsschema für das Binpacking-Minimierungsproblem und andere Optimierungsprobleme. Diese Überlegung führt auf folgende Definition:

Es sei Π ein Optimierungsproblem aus **NPO**. Ein Algorithmus **A** heißt **asymptotisches Approximationsschema** für Π, wenn es eine Konstante k gibt, so daß gilt:
für jede Eingabeinstanz x von Π und für jede rationale Zahl $r > 1$ liefert **A** bei Eingabe von (x, r) eine (zulässige) Lösung, deren relative Approximationsgüte $R_{\mathbf{A}}(x)$ die Bedingung $R_{\mathbf{A}}(x) \leq r + k/m^*_{\Pi}(x)$ erfüllt. Außerdem ist die Laufzeit von **A** für jedes feste r polynomiell in der Größe *size*(x) der Eingabeinstanz.

Zur Erinnerung: die relative Approximationsgüte wurde definiert durch

$R_{\mathbf{A}}(x) = \dfrac{m^*_{\Pi}(x)}{m(x, \mathbf{A}(x))}$ bei einem Maximierungsproblem

bzw.

$R_{\mathbf{A}}(x) = \dfrac{m(x, \mathbf{A}(x))}{m^*_{\Pi}(x)}$ bei einem Minimierungsproblem.

Die Bedingung $R_{\mathbf{A}}(x) \le r + k/m^*_{\Pi}(x)$ besagt also
bei einem Maximierungsproblem:
$m(x, \mathbf{A}(x)) \ge 1/r \cdot m^*_{\Pi}(x) - k'$ mit $k' = k/\left(r + k/m^*_{\Pi}(x)\right) \le k/r$,
daher $m(x, \mathbf{A}(x)) \ge 1/r \cdot m^*_{\Pi}(x) - k/r$
bzw.
bei einem Minimierungsproblem: $m(x, \mathbf{A}(x)) \le r \cdot m^*_{\Pi}(x) + k$.

Die Bezeichnung „asymptotisches Approximationsschema“ ist aus der Tatsache zu erklären, daß für „große“ Eingabeinstanzen *x* der Wert $m^*_{\Pi}(x)$ der Zielfunktion einer optimalen Lösung ebenfalls groß ist. Daher gilt in diesem Fall $\lim_{size(x)\to\infty} R_{\mathbf{A}}(x) \le r$.

Die Klasse aller Optimierungsprobleme, die ein asymptotisches Approximationsschema zulassen, wird mit $\mathbf{PTAS}^{\infty}$ bezeichnet.

Satz 6.2-4:

Das Binpacking-Minimierungsproblem liegt in $\mathbf{PTAS}^{\infty}$, d.h. es kann asymptotisch beliebig dicht in polynomieller Zeit approximiert werden (auch wenn $\mathbf{P} \neq \mathbf{NP}$ gilt).

Es gilt sogar: Es gibt ein asymptotisches Approximationsschema, das polynomiell in der Problemgröße und in $1/(r-1)$ ist.

Die Klasse $\mathbf{PTAS}^{\infty}$ ordnet sich in die übrigen Approximationsklassen ein:

Satz 6.2-5:

Ist $\mathbf{P} \neq \mathbf{NP}$, so gilt $\mathbf{FPTAS} \subset \mathbf{PTAS} \subset \mathbf{PTAS}^{\infty} \subset \mathbf{APX} \subset \mathbf{NPO}$.

7 Weiterführende Konzepte

Die Analyse des Laufzeitverhaltens eines Algorithmus **A** im schlechtesten Fall $T_{\mathbf{A}}(n) = \max\{ t_{\mathbf{A}}(x) \mid x \in \Sigma^* \text{ und } |x| \le n \}$ liefert eine *Garantie* (obere Schranke) für die Zeit, die er bei einer Eingabe zur Lösung benötigt[4]. Für jede Eingabe $x \in \Sigma^*$ mit $|x| = n$ gilt $t_{\mathbf{A}}(x) \le T_{\mathbf{A}}(n)$. Dieses Verhalten ist oft jedoch nicht charakteristisch für das Verhalten bei „den meisten“ Eingaben. Ist eine Wahrscheinlichkeitsverteilung *P* der Eingabewerte bekannt oder werden alle Eingaben als gleichwahrscheinlich aufgefaßt, so kann man das Laufzeitverhalten von **A** untersuchen, wie es sich im Mittel darstellt. Die **Zeitkomplexität** von **A** im **Mittel** wird definiert als

$$T_{\mathbf{A}}^{avg}(n) = \mathbf{E}[t_{\mathbf{A}}(x) \mid |x| = n] = \int_{|x|=n} t_{\mathbf{A}}(x) \cdot dP(x).$$

Die Untersuchung des mittleren Laufzeitverhaltens von Algorithmen ist in den meisten Fällen sehr viel schwieriger als die worst-case-Analyse. Trotzdem gibt es sehr ermutigende Ergebnisse, insbesondere bei der Lösung einiger „klassischer“ Probleme. Der Einbau von Zufallsexperimenten in Algorithmen hat auf dem Gebiet der Approximationsalgorithmen für Optimierungsprobleme aus **NPO** gute Ergebnissen geliefert. In neuerer Zeit hat der Einsatz von probabilistischen Modellen zu neuen Erkenntnissen auf dem Weg zur Lösung der **P**-**NP**-Frage geführt.

7.1 Randomisierte Algorithmen

Im ansonsten deterministischen Algorithmus werden als zulässige Elementaroperationen Zufallsexperimente zugelassen. Ein derartiges Zufallsexperiment kann beispielsweise mit Hilfe eines Zufallszahlengenerators ausgeführt werden. So wird beispielsweise auf diese Weise entschieden, in welchem Teil einer Programmverzweigung der Algorithmus während seines Ablaufs fortgesetzt wird. Andere Möglichkeiten zum Einsatz eines Zufallsexperiments bestehen bei Entscheidungen zur Auswahl möglicher Elemente, die im weiteren Ablauf des Algorithmus als nächstes untersucht werden sollen.

Man nennt derartige Algorithmen **randomisierte Algorithmen**.

Grundsätzlich gibt es zwei Klassen randomisierter Algorithmen: **Las-Vegas-Verfahren**, die stets – wie von deterministischen Algorithmen gewohnt – ein korrektes Ergebnis berechnen.

---

[4] $t_{\mathbf{A}}(x) =$ Anzahl der Anweisungen, die von **A** zur Berechnung von $\mathbf{A}(x)$ durchlaufen werden.

Daneben gibt es **Monte-Carlo-Verfahren**, die ein korrektes Ergebnis nur mit einer gewissen Fehlerwahrscheinlichkeit, aber in jedem Fall effizient, bestimmen. Häufig findet man bei randomisierten Algorithmen einen Trade-off zwischen Korrektheit und Effizienz.

Beispiele für randomisierte Algorithmen vom Las-Vegas-Typ sind:

- Einfügen von Primärschlüsselwerten in einen binären Suchbaum: Statt die Schlüssel $S_1, ..., S_n$ in dieser Reihenfolge sequentiell in den binären Suchbaum einzufügen, wird jeweils der nächste einzufügende Schlüssel aus den restlichen, d.h. noch nicht eingefügten Schlüsseln zufällig ausgewählt und in den binären Suchbaum eingefügt. Das Ergebnis ist eine mittlere Baumhöhe der Ordnung $O(\log(n))$
- `Quicksort` zum Sortieren Elementen, auf denen eine lineare Ordnung definiert ist.

Sortieren von Elementen, auf denen eine Ordnungsrelation besteht

Instanz: $x = \langle a_1, ..., a_n \rangle$

x ist eine Folge von Elementen der Form $a_i = (key_i, info_i)$; die Elemente sind bezüglich ihrer *key*-Komponente vergleichbar, d.h. es gilt für $a_i = (key_i, info_i)$ und $a_j = (key_j, info_j)$ die Beziehung $a_i \le a_j$ genau dann, wenn $key_i \le key_j$ ist.

Lösung: Eine Umordnung $p : [1:n] \to [1:n]$ der Elemente in *x*, so daß $\langle a_{p(1)}, ..., a_{p(n)} \rangle$ nach aufsteigenden Werten der *key*-Komponente sortiert ist, d.h. es gilt: $a_{p(i)} \le a_{p(i+1)}$ für $i = 1, ..., n-1$.

Als Randbedingung gilt, daß die Sortierung innerhalb der Folge *x* erfolgen soll, also insbesondere nicht zusätzlicher Speicherplatz der Größenordnung $O(n)$ erforderlich wird (**internes Sortierverfahren**).

Zur algorithmischen Behandlung wird eine Instanz *x* der Größe *n* in einem Feld `a` gespeichert, das definiert wird durch

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

TYPE  idx_bereich = 1..n;
      key_typ     = ...;     { Typ der key-Komponente          }
      info_typ    = ...;     { Typ der info-Komponente         }
      entry_typ   = RECORD
                      key  : key_typ;
                      info : info_typ
                    END;
```

```
feld_typ    = ARRAY [idx_bereich] OF entry_typ;
```

Die folgende Prozedur `interchange` vertauscht zwei Einträge miteinander:

```
PROCEDURE interchange (VAR x, y : entry_typ);
{ die Werte von x und y werden miteinander vertauscht            }

VAR z : entry_typ;

BEGIN { interchange }
  z := x;
  x := y;
  y := z;
END   { interchange };
```

Der Algorithmus `quicksort` beruht auf der Idee der Divide-and-Conquer-Methode und erzeugt die umsortierte Eingabe $\langle a_{p(1)}, ..., a_{p(n)} \rangle$:

Besteht *x* aus keinem oder nur aus einem einzigen Element, dann ist nichts zu tun. Besteht *x* aus mehr Elementen, dann wird aus *x* ein Element *e* (**Pivot-Element**) ausgewählt und das Problem der Sortierung von *x* in zwei kleinere Probleme aufgeteilt, nämlich der Sortierung aller Elemente, die kleiner als *e* sind (das sei das Problem der Sortierung der Folge *x(*lower*)*), und die Sortierung der Elemente die größer oder gleich *e* sind (das sei das Problem der Sortierung der Folge *x(*upper*)*); dabei wird als „Raum zur Sortierung" die Instanz *x* verwendet. Zuvor wird *e* an diejenige Position innerhalb von *x* geschoben, an der es in der endgültigen Sortierung stehen wird. Das Problem der Sortierung der erzeugten kleineren Probleme *x(*lower*)* und *x(*upper*)* wird (rekursiv) nach dem gleichen Prinzip gelöst. Die Position, an der *e* innerhalb von *x* in der endgültigen Sortierung stehen wird und damit die kleineren Probleme *x(*lower*)* und *x(*upper*)* bestimmt, sind dadurch gekennzeichnet, daß gilt:
$k < e$ für alle $k \in x(\text{lower})$ und $k \geq e$ für alle $k \in x(\text{upper})$.

Sortieralgorithmus mit `quicksort`:

Eingabe: $x = \langle a_1, ..., a_n \rangle$ ist eine Folge von Elementen der Form $a_i = (key_i, info_i)$; die Elemente sind bezüglich ihrer *key*-Komponente vergleichbar, d.h. es gilt für $a_i = (key_i, info_i)$ und $a_j = (key_j, info_j)$ die Beziehung $a_i \leq a_j$ genau dann, wenn $key_i \leq key_j$ ist

```
VAR a : feld_typ;        { Eingabefeld                    }
    i : idx_bereich;
```

```
FOR i := 1 TO n DO
 BEGIN
   a[i].key   := key_i ;
   a.[i].info := info_i
 END;
```

Verfahren: Aufruf der Prozedur `quicksort (a, 1, n);`

Ausgabe: `a` mit `a[`i`].key` $\leq$ `a[`i `+ 1].key` für $i = 1, ..., n-1$

```
PROCEDURE quicksort (VAR a     : feld_typ;
                         lower : idx_bereich;
                         upper : idx_bereich);

    VAR  t           : entry_typ;
         m           : idx_bereich;
         idx         : idx_bereich;
         zufalls_idx : idx_bereich;

BEGIN { quicksort }
  IF lower < upper
  THEN BEGIN  { ein Feldelement zufällig aussuchen und
                mit a[lower] austauschen                          }
              zufalls_idx := lower
                             + Trunc(Random * (upper - lower + 1));
              interchange (a[lower], a[zufalls_idx]);

              t := a[lower];
              m := lower;

              FOR idx := lower + 1 TO upper DO
                  { es gilt: a[lower+1] , ..., a[m] < t und
                             a[m+1], ... a[idx-1] >= t       }
                  IF a[idx].key < t.key
                  THEN BEGIN
                         m := m+1;
                         { vertauschen von a[m] und a[idx] }
                         interchange (a[m], a[idx])
                       END;

              { a[lower] und a[m] vertauschen }
              interchange (a[lower], a[m]);

              { es gilt: a[lower], ..., a[m-1] < a[m]  und
                             a[m] <= a[m+1], ..., a[upper]  }
```

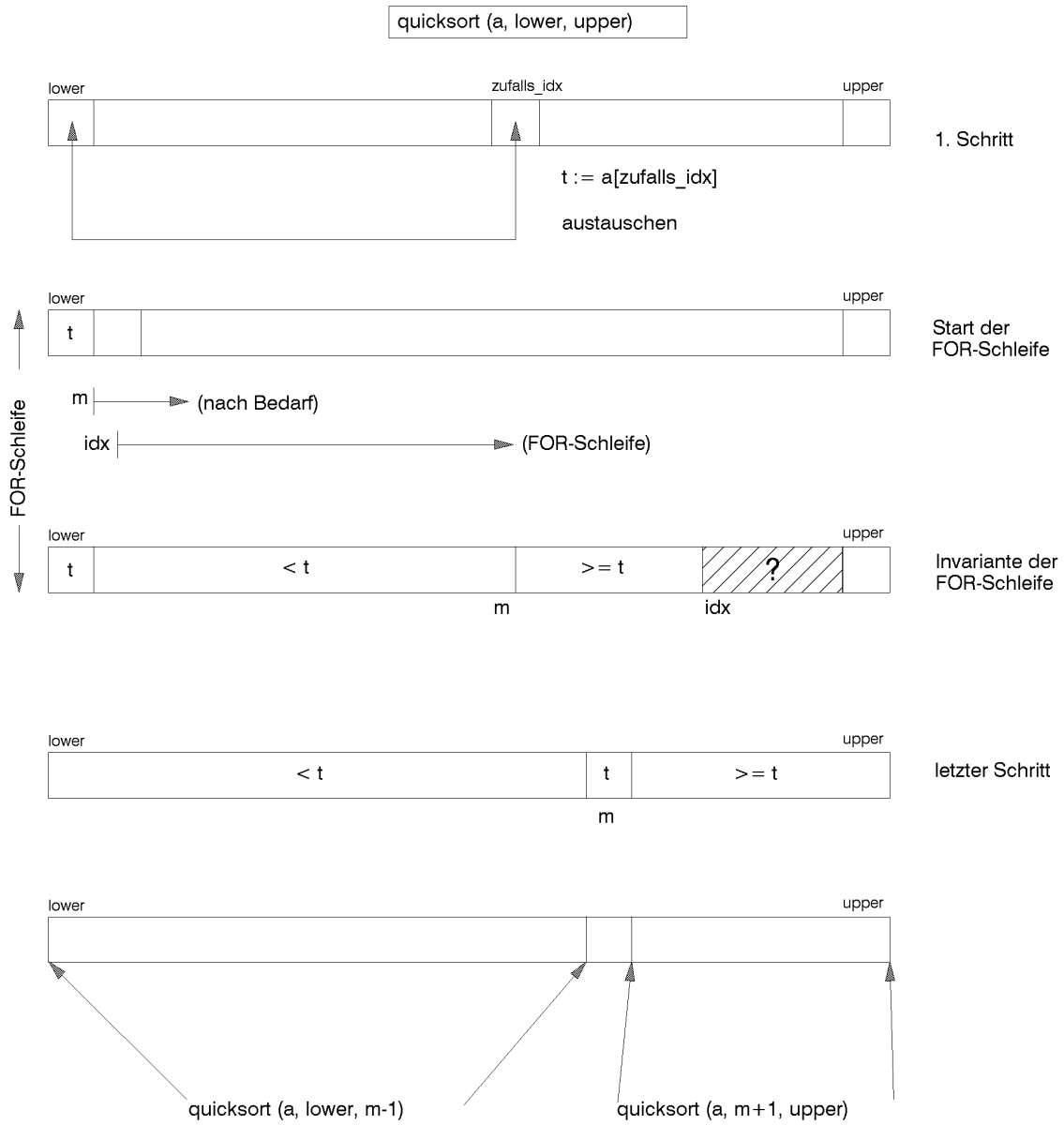
```
            quicksort (a, lower, m-1);
            quicksort (a, m+1, upper);

          END { IF}

    END { quicksort };
```

Der Algorithmus stoppt, da in jedem `quicksort`-Aufruf ein Element an die endgültige Position geschoben wird und die Anzahl der Elemente in den `quicksort`-Aufrufen in den beiden restlichen Teilfogen zusammen um 1 verringert ist. Die Korrektheit des Algorithmus folgt aus der Invariante der `FOR`-Schleife.

Satz 7.1-1:

Das beschriebene Verfahren mit der Prozedur `quicksort` sortiert eine Folge $x = \langle a_1, ..., a_n \rangle$ aus n Elementen mit einer (worst-case-) Zeitkomplexität der Ordnung $O(n^2)$. Diese Schranke wird erreicht, wenn zufällig in jedem `quicksort`-Aufruf das Pivot-Element so gewählt wird, daß die Teilfolgen *x(*lower*)* kein Element und *x(*upper*)* alle restlichen Elemente enthalten (bzw. umgekehrt). Im Mittel ist das Laufzeitverhalten jedoch wesentlich besser, nämlich von der Ordnung $O(n \cdot \log(n))$.

Es läßt sich zeigen, daß jedes Sortierverfahren, das auf dem Vergleich von Elementgrößen beruht, eine untere Zeitkomplexität mindestens der Ordnung $O(n \cdot \log(n))$ und eine mittlere Zeitkomplexität ebenfalls mindestens der Ordnung $O(n \cdot \log(n))$ hat, so daß das Verfahren mit der Prozedur `quicksort` optimales Laufzeitverhalten im Mittel aufweist. Daß das Verfahren mit der Prozedur `quicksort` in der Praxis ein Laufzeitverhalten zeigt, das fast alle anderen Sortierverfahren schlägt, liegt daran, daß die „ungünstigen“ Eingaben für `quicksort` mit gegen 0 gehender Wahrscheinlichkeit vorkommen und die Auswahl des Pivot-Elements zufällig erfolgt.



Beispiele für randomisierte Algorithmen vom Monte-Carlo-Typ sind die heute üblichen Primzahltests, die hier informell beschrieben werden sollen (Details findet man in der angegebenen Literatur).

Um eine natürliche Zahl $n > 2$ auf Primzahleigenschaft zu testen, könnte man alle Primzahlen von 2 bis $\lfloor \sqrt{n} \rfloor$ daraufhin untersuchen, ob es eine von ihnen gibt, die n teilt. Wenn die Zahl n

nämlich zusammengesetzt ist, d.h. keine Primzahl ist, hat sie einen Primteiler *p* mit $p \le \sqrt{n}$. Umgekehrt, falls alle Primzahlen *p* mit $p \le \sqrt{n}$ keine Teiler von *n* sind, dann ist *n* selbst eine Primzahl. Die Primzahlen könnte man etwa systematisch erzeugen (siehe Literatur unter dem Stichwort „Sieb des Eratosthenes“) oder man könnte sie einer Primzahltabelle (falls vorhanden) entnehmen. Allerdings ist dieser Ansatz für sehr große Werte von *n* nicht praktikabel und benötigt exponentiellen Rechenaufwand (in der Anzahl der Stellen von *n*): Die Anzahl der Stellen $b(n)$ einer Zahl $n \ge 1$ im Zahlensystem zur Basis *B* ist $b(n) = \lfloor \log_B(n) \rfloor + 1 = \lceil \log_B(n+1) \rceil$, d.h. $b(n) \in O(\log(n))$. Die Anzahl der Primzahlen unterhalb $\lfloor \sqrt{n} \rfloor$ beträgt für große *n* nach dem Primzahlsatz der Zahlentheorie $p(\sqrt{n}) \sim \frac{n^{1/2}}{\ln(n^{1/2})}$, also ein Wert der Ordnung $O\left(\frac{2^{b(n)/2}}{b(n)}\right)$. Jede in Frage kommende Primzahl muß daraufhin untersucht werden, ob sie *n* teilt. Dazu sind mindestens $O\left((b(n))^2\right)$ viele Bitoperationen erforderlich. Daher ist der Gesamtaufwand mindestens von der Ordnung $O\left(b(n) \cdot 2^{b(n)/2}\right)$.

Durch Anwendung zahlentheoretischer Erkenntnisse hat man versucht, effiziente Primzahltests zu entwickeln. Der bisher bekannte schnellste Algorithmus zur Überprüfung einer Zahl *n* auf Primzahleigenschaft, der APRCL-Test, hat eine Laufzeit der Ordnung $O\left((\log(n))^{c(\log(\log(\log(n))))}\right)$ mit einer Konstanten $c > 0$. Auch das ist keine polynomielle Laufzeit.

Es sind daher andere Ansätze für effiziente Primzahltests erforderlich. Als erfolgreich haben sich hierbei **probabilistische Algorithmen** erwiesen.

Um zu testen, ob eine Zahl *n* eine Primzahl ist oder nicht, versucht man, einen **Zeugen (witness) für die Primzahleigenschaft von** *n* zu finden. Ein Zeuge ist dabei eine Zahl *a* mit $1 \le a \le n-1$, der eine bestimmte Eigenschaft zukommt, aus der man *vermuten* kann, daß *n* Primzahl ist. Dabei muß diese Eigenschaft bei Vorgabe von *a* einfach zu überprüfen sein, und nach Auffinden einiger weniger Zeugen für die Primzahleigenschaft von *n* muß der Schluß gültig sein, daß *n* mit hoher Wahrscheinlichkeit eine Primzahl ist. Die Eigenschaft „die Primzahl *p* mit $2 \le p \le \lfloor \sqrt{n} \rfloor$ ist kein Teiler von *n*“ ist dabei nicht geeignet, da man im allgemeinen zu viele derartige Zeugen zwischen 2 und $\lfloor \sqrt{n} \rfloor$ bemühen müßte, um sicher auf die Primzahleigenschaft schließen zu können.

Es sei $E(a)$ eine (noch genauer zu definierende) Eigenschaft, die einer Zahl *a* mit $1 \le a \le n-1$ zukommen kann und für die gilt:

(i) $E(a)$ ist algorithmisch mit geringem Aufwand zu überprüfen

(ii) falls *n* Primzahl ist, dann trifft $E(a)$ für alle Zahlen *a* mit $1 \le a \le n-1$ zu

(iii) falls *n* keine Primzahl ist, dann trifft $E(a)$ für weniger als die Hälfte aller Zahlen *a* mit $1 \le a \le n-1$ zu.

Falls $E(a)$ gilt, dann heißt a **Zeuge (witness) für die Primzahleigenschaft von** n. Dann läßt sich folgender randomisierte Primzahltest definieren:

Eingabe: $n \in \mathbf{N}$, n ist ungerade, $m \in \mathbf{N}$

Verfahren: Aufruf der Funktion `random_is_prime (n : INTEGER; m : INTEGER): BOOLEAN;`

Ausgabe: n wird als Primzahl angesehen, wenn `random_is_prime (n, m) = TRUE` ist, ansonsten wird n nicht als Primzahl angesehen.

```
FUNCTION random_is_prime (n : INTEGER;
                          m : INTEGER): BOOLEAN;

   VAR idx      : INTEGER;
       a        : INTEGER;
       is_prime : BOOLEAN;


   BEGIN { random_is_prime }
     is_prime := TRUE;

     FOR idx := 1 TO m DO
       BEGIN
         wähle eine Zufallszahl a zwischen 1 und n-1;
         IF ( E(a) trifft nicht zu)
         THEN BEGIN
                is_prime := FALSE;
                Break;
              END;
       END;

     random_is_prim := is_prime;

   END   { random_is_prime };
```

Der Algorithmus versucht also, m Zeugen für die Primzahleigenschaft von n zu finden. Wird dabei zufällig eine Zahl a mit $1 \le a \le n-1$ erzeugt, für die $E(a)$ nicht zutrifft, dann wird wegen (ii) die korrekte Antwort gegeben. Ist n Primzahl, dann gibt der Algorithmus ebenfalls wegen (ii) die korrekte Antwort. Wurden m Zeugen für die Primzahleigenschaft von n festge-

stellt, kann es trotzdem sein, daß *n* keine Primzahl ist, obwohl der Algorithmus angibt, *n* sei Primzahl. Die Wahrscheinlichkeit, bei einer Zahl *n*, die nicht Primzahl ist, *m* Zeugen zu finden, ist wegen (iii) kleiner als $(1/2)^m$, d.h. die Wahrscheinlichkeit einer fehlerhaften Entscheidung ist in diesem Fall kleiner als $(1/2)^m$. Insgesamt ist die Wahrscheinlichkeit einer korrekten Antwort des Algorithmus größer als $1-(1/2)^m$. Da wegen (i) die Eigenschaft $E(a)$ leicht zu überprüfen ist, ist hiermit ein effizientes Verfahren beschrieben, das mit beliebig hoher Wahrscheinlichkeit eine korrekte Antwort liefert. Diese ist beispielsweise für $m = 20$ größer als 0,999999.

Die Frage stellt sich, ob es geeignete Zeugeneigenschaften $E(a)$ gibt. Einen ersten Versuch legt der Satz von Fermat nahe:

Satz 7.1-2:

Ist $n \in \mathbf{N}$ eine Primzahl und $a \in \mathbf{N}$ eine Zahl mit $ggT(a,n) = 1$, dann ist $a^{n-1} \equiv 1 \pmod{n}$.

Definiert man $E(a) =$ „$ggT(a,n) = 1$ und $a^{n-1} \equiv 1 \pmod{n}$“, dann sieht man, daß (i) und (ii) gelten. Leider gilt (iii) für diese Eigenschaft $E(a)$ nicht. Es gibt nämlich unendlich viele Zahlen *n*, die **Carmichael-Zahlen**, die folgende Eigenschaft besitzen: *n* ist *keine* Primzahl, und es ist $a^{n-1} \equiv 1 \pmod{n}$ für alle *a* mit $ggT(a,n) = 1$. Ist *n* also eine Carmichael-Zahl und *a* eine Zahl mit $1 \le a \le n-1$ und $ggT(a,n) = 1$, dann ist $a^{n-1} \equiv 1 \pmod{n}$. In diesem Fall gilt also für alle Zahlen *a* mit $1 \le a \le n-1$ und $ggT(a,n) = 1$ die Eigenschaft $E(a)$, und das sind mehr als die Hälfte aller Zahlen *a* mit $1 \le a \le n-1$.

Ein zweiter Versuch zur geeigneten Definition einer Zeugeneigenschaften $E(a)$ erweitert den ersten Versuch und wird durch die folgenden beiden Sätze begründet:

Satz 7.1-3:

Es sei *n* eine Primzahl. Dann gilt $x^2 \equiv 1 \pmod{n}$ genau dann, wenn $x \equiv 1 \pmod{n}$ oder $x \equiv -1 \equiv n-1 \pmod{n}$ ist.

Es sei *n* eine Primzahl mit $n > 2$. Dann ist *n* ungerade, d.h. $n = 1 + 2^j \cdot r$ mit ungeradem *r* und $j > 0$ bzw. $n - 1 = 2^j \cdot r$. Ist *a* eine Zahl mit $1 \le a \le n-1$, dann ist $ggT(a,n) = 1$ und folglich $a^{n-1} \equiv 1 \pmod{n}$. Wegen $a^{n-1} = a^{(2^{j-1} \cdot r)2} = \left(a^{2^{j-1} \cdot r}\right)^2 \equiv 1 \pmod{n}$ folgt mit Satz 7.1-3 (dort wird $x = a^{2^{j-1} \cdot r}$ gesetzt): $a^{2^{j-1} \cdot r} \equiv 1 \pmod{n}$ oder $a^{2^{j-1} \cdot r} \equiv -1 \pmod{n}$. Ist hierbei $j - 1 > 0$ und

$a^{2^{j-1}\cdot r} \equiv 1 (\mathrm{mod}\, n)$, dann kann man den Vorgang des Wurzelziehens wiederholen: In Satz 7.1-3 wird $x = a^{2^{j-2}\cdot r}$ gesetzt usw. Der Vorgang ist spätestens dann beendet, wenn $a^{2^{j-j}\cdot r} = a^r$ erreicht ist. Es gilt daher der folgende Satz:

Satz 7.1-4:

Es sei *n* eine ungerade Primzahl, $n = 1 + 2^j \cdot r$ mit ungeradem *r* und $j > 0$. Dann gilt für jedes *a* mit $1 \le a \le n-1$:

Die Folge $\left(a^r, a^{2r}, a^{4r}, ..., a^{2^{j-1}\cdot r}, a^{2^j \cdot r}\right)$ der Länge $j+1$, wobei alle Werte modulo *n* reduziert werden, hat eine der Formen

$(1, 1, 1, ..., 1, 1)$ oder

$(*, *, ..., *, -1, 1, ..., 1, 1)$.

Hierbei steht das Zeichen „*“ für eine Zahl, die verschieden von 1 oder -1 ist.

Wenn die in Satz 7.1-4 beschriebene Folge eine der drei Formen

$(*, *, ..., *, 1, 1, ..., 1, 1)$,

$(*, *, ..., *, -1)$ oder

$(*, *, ..., *, *, ..., *)$

aufweist, dann ist *n* mit Sicherheit keine Primzahl. Andererseits ist es nicht ausgeschlossen, daß für eine ungerade zusammengesetzte Zahl *n* und eine Zahl *a* mit $1 \le a \le n-1$ die Folge $\left(a^r, a^{2r}, a^{4r}, ..., a^{2^{j-1}\cdot r}, a^{2^j \cdot r}\right) \mathrm{mod}\, n$ eine der beiden Formen $(1, 1, 1, ..., 1, 1)$ oder $(*, *, ..., *, -1, 1, ..., 1, 1)$ hat. In diesem Fall heißt *n* **streng pseudoprim zu Basis** *a*. Es gilt folgender Satz (Beweis siehe Literatur):

Satz 7.1-5:

Es sei *n* eine ungerade zusammengesetzte Zahl. Dann ist *n* streng pseudoprim für höchtens ein Viertel aller Basen *a* mit $1 \le a \le n-1$.

Eine geeignete Zeugeneigenschaften $E(a)$ für die Funktion

```
random_is_prime (n : INTEGER;
                 m : INTEGER): BOOLEAN;
```

ist daher die folgende Bedingung:

$E(a) =$ „$ggT(a, n) = 1$ und

$a^{n-1} \equiv 1 (\mathrm{mod}\, n)$ und

die Folge $\left(a^r, a^{2r}, a^{4r}, ..., a^{2^{j-1}\cdot r}, a^{2^j \cdot r}\right) \mathrm{mod}\, n$ hat eine der Formen

$(1, 1, 1, ..., 1, 1)$ oder $(*, *, ..., *, -1, 1, ..., 1, 1)$“.

Die Wahrscheinlichkeit einer korrekten Antwort des Algorithmus mit dieser Zeugeneigenschaft ist wegen Satz 7.1-5 größer als $1-(1/4)^m$.

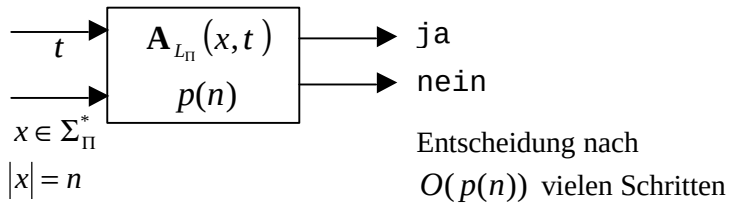
Es ist übrigens nicht notwendig, für eine große Anzahl von Zahlen a mit $1 \le a \le n-1$ die Zeugeneigenschaft zu überprüfen um sicher zu gehen, daß n eine Primzahl ist: Nur eine zusammengesetzte Zahl $n < 2{,}5 \cdot 10^{10}$, nämlich n = 3.215.031.751, ist streng pseudoprim zu den vier Basen a = 2, 3, 5 und 7. Für praktische Belange ist daher das Verfahren ein effizienter Primzahltest. Untersucht man große Zahlen, die spezielle Formen aufweisen, etwa Mersenne-Zahlen, auf Primzahleigenschaft bieten sich speziell angepaßte Testverfahren an. Schließlich gibt es eine Reihe von Testverfahren, die andere zahlentheoretische Eigenschaften nutzen.

7.2 Modelle randomisierter Algorithmen

In Zusammenführung der obigen Ansätze mit den Modellen aus der Theorie der Berechenbarkeit (Turingmaschinen, deterministische und nichtdeterministische Algorithmen) wurde eine Reihe weiterer Berechnungsmodelle entwickelt. Im folgenden werden wieder Entscheidungsprobleme betrachtet.

Ausgehend von der Klasse **P** der deterministisch polynomiell entscheidbaren Probleme erweitert man deren Algorithmen nicht um die Möglichkeit der Verwendung nichtdeterministisch erzeugter Zusatzinformationen (Beweise) wie beim Übergang zu den polynomiellen Verifizierern, sondern läßt zu, daß bei Verzweigungen während des Ablaufs der Algorithmen (Verzweigungen) ein Zufallsexperiment darüber entscheidet, welche Alternative für den weiteren Ablauf gewählt wird. Man gelangt so zu der Klasse **RP** (randomized polynomial time).

Es sei Π ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$. L_Π liegt in **RP** (bzw. „Π liegt in **RP**“), wenn es einen Algorithmus (**RP-Akzeptor**) $\mathbf{A}_{L_\Pi}$ gibt, der neben der Eingabe $x \in \Sigma_\Pi^*$ eine Folge $t \in \{0,1\}^*$ zufälliger Bits liest, wobei jedes Bit unabhängig von den vorher gelesenen Bits ist und $P(0) = P(1) = 1/2$ gilt. Nach polynomiell vielen Schritten (in Abgängigkeit von der Größe $|x|$ der Eingabe) kommt der Akzeptor auf die `ja`-Entscheidung bzw. auf die `nein`-Entscheidung. Eingaben für $\mathbf{A}_{L_\Pi}$ sind also die Wörter $x \in \Sigma_\Pi^*$ und eine Folge $t \in \{0,1\}^*$ zufälliger Bits:



Für jedes $x \in L_\Pi$ ist $P(\mathbf{A}_{L_\Pi}(x,t) = \texttt{ja}) \geq 1/2$,
für jedes $x \notin L_\Pi$ ist $P(\mathbf{A}_{L_\Pi}(x,t) = \texttt{nein}) = 1$

Man läßt also zur Akzeptanz von $x \in L_\Pi$ einen einseitigen Fehler zu. Jedoch muß bei $x \in L_\Pi$ der Algorithmus $\mathbf{A}_\Pi$ bei mindestens der Hälfte aller möglichen Zufallsfolgen τ auf die `ja`-Entscheidung kommen. Für die übrigen darf er auch auf die `nein`-Entscheidung führen. Für $x \notin L_\Pi$ muß er aber immer die `nein`-Entscheidung treffen. Der einseitige Fehler kann durch Einsatz geeigneter Replikations-Techniken beliebig klein gehalten werden.

Da dem Akzeptor nur polynomielle Zeit zur Verfügung steht, kann er auch nur polynomiell viele Bits der Zufallsfolge t lesen, so daß man annehmen kann, daß t aus polynomiell vielen Bits besteht.

Ein Beispiel für eine Sprache aus **RP** ist die Menge $L_2 = \{n \mid n \in \mathbf{N} \text{ und } n \text{ ist keine Primzahl}\}$ aus Kapitel 5.5. Dort wird $L_2 \in \mathbf{NP}$ gezeigt. Einen **RP**-Akzeptor $\mathbf{A}_{L_2}$ für L_2 erhält man aus der Funktion `random_is_prim` aus Kapitel 7.1. $\mathbf{A}_{L_2}$ liest zwei Eingaben, nämlich eine Zahl $n \in \mathbf{N}$ und eine Folge $t \in \{0,1\}^*$ zufälliger Bits der Länge $k = size(n)$. Diese Zufallsfolge wird als Binärkodierung einer Zufallszahl a mit $1 \leq a \leq n-1$ interpretiert (die Fälle $a = 0$ und $a = n$ sollen zur vereinfachten Darstellung hier ausgeschlossen sein). Dazu wird angenommen, daß es eine Funktion `make_INTEGER(`t`,` $size(n)$`)` gibt, die die Zufallsfolge t in eine natürliche Zahl a mit $1 \leq a \leq n-1$ umwandelt. Dann wird die Bedingung $E(a)$ (siehe Kapitel 7.1) überprüft und eine Entscheidung getroffen:

```
FUNCTION A_L2 (n : INTEGER;
               t : ...      ): ...;

   VAR a : INTEGER;

   BEGIN { A_L2 }
     a := make_INTEGER(t , size(n));
     IF ( E(a) trifft nicht zu)
     THEN A_L2 := ja
     ELSE A_L2 := nein;
END   { A_L2 };
```

Ist $n \in L_2$, d.h. n ist keine Primzahl, dann trifft nach Definition $E(a)$ für weniger als die Hälfte aller Zahlen a mit $1 \le a \le n-1$ zu. Daher trifft $E(a)$ für mehr als die Hälfte aller Zahlen a mit $1 \le a \le n-1$ nicht zu, und es gilt $P(\mathbf{A}_{L_2}(n,t) = \texttt{ja}) > 1/2$.

Ist $n \notin L_2$, d.h. n ist eine Primzahl, dann trifft nach Definition $E(a)$ für alle Zahlen a mit $1 \le a \le n-1$ zu. Daher antwortet $\mathbf{A}_{L_2}$ mit $\mathbf{A}_{L_2}$:= nein.

Satz 7.2-1:

Es gilt $\mathbf{P} \subseteq \mathbf{RP}$ und $\mathbf{RP} \subseteq \mathbf{NP}$.

Ob diese Inklusionen echt sind, ist nicht bekannt, vieles spricht jedoch dafür.

Beweis:

Es sei $L_\Pi \in \mathbf{P}$, $L_\Pi \subseteq \Sigma_\Pi^*$. Dann gibt es einen deterministischen polynomiell zeitbeschränkten Entscheidungsalgorithmus für L_Π. Dieser ist kann als **RP**-Akzeptor betrachtet werden, der ohne Zufallsfolge auskommt. Daher gilt $L_\Pi \in \mathbf{RP}$.

Es sei $L_\Pi \in \mathbf{RP}$, $L_\Pi \subseteq \Sigma_\Pi^*$. Dann gibt es einen **RP**-Akzeptor $\mathbf{A}_{L_\Pi}$ für L_Π. Zu zeigen ist, daß es auch einen Verifizierer, wie er für die Klasse **NP** erforderlich ist, für L_Π gibt („**NP**-Verifizierer"). Der **RP**-Akzeptor $\mathbf{A}_{L_\Pi}$ kann als Verifizierer angesehen werden. Bei Eingabe von $x \in \Sigma_\Pi^*$ bekommt dieser als Beweis B_x eine Zufallsfolge t. Ist $x \in L_\Pi$, dann gibt es eine Zufallsfolge t mit $\mathbf{A}_{L_\Pi}(x,t) = \texttt{ja}$ (da $x \in L_\Pi$ und in diesem Fall $P(\mathbf{A}_{L_\Pi}(x,t) = \texttt{ja}) \ge 1/2$ gelten, führen mindestens die Hälfte aller Zufallsfolgen auf die ja-Entscheidung). Ist $x \notin L_\Pi$, dann führen wegen $P(\mathbf{A}_{L_\Pi}(x,t) = \texttt{nein}) = 1$ alle Zufallsfolgen auf die nein-Entscheidung. ///

Die Zusammenführung der Konzepte des Zufalls und des Nichtdeterminismus bei polynomiell zeitbeschränktem Laufzeitverhalten führt auf die Klasse **PCP** (probabilistically checkable proofs). Hierbei werden Verifizierer, wie sie bei der Definition der Klasse **NP** eingeführt wurden, um die Möglichkeit erweitert, Zufallsexperimente auszuführen:

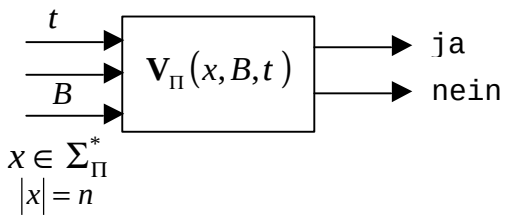
Es seien $r : \mathbf{N} \to \mathbf{N}$ und $q : \mathbf{N} \to \mathbf{N}$ Funktionen. Ein Verifizierer (nichtdeterministischer Algorithmus) $\mathbf{V}_\Pi$ heißt $(r(n), q(n))$-**beschränkter Verifizierer für das Entscheidungsproblem** Π über einem Alphabet Σ_Π, wenn er Zugriff auf eine Eingabe $x \in \Sigma_\Pi^*$ mit $|x| = n$, einen Beweis $B \in \Sigma_0^*$ und eine Zufallsfolge $t \in \{0,1\}^*$ hat und sich dabei folgendermaßen verhält:

1. $\mathbf{V}_\Pi$ liest zunächst die Eingabe $x \in \Sigma_\Pi^*$ (mit $|x| = n$) und $O(r(n))$ viele Bits aus der Zufallsfolge $t \in \{0,1\}^*$.
2. Aus diesen Informationen berechnet $\mathbf{V}_\Pi$ $O(q(n))$ viele Positionen der Zeichen von $B \in \Sigma_0^*$, die überhaupt gelesen (erfragt) werden sollen.
3. In Abhängigkeit von den gelesenen Zeichen in B (und der Eingabe x) kommt $\mathbf{V}_\Pi$ auf die `ja`- bzw- `nein`-Entscheidung.

Die Entscheidung, die $\mathbf{V}_\Pi$ bei Eingabe von $x \in \Sigma_\Pi^*$, $B \in \Sigma_0^*$ und $t \in \{0,1\}^*$ liefert, wird mit $\mathbf{V}_\Pi(x, B, t)$ bezeichnet.

Es sei Π ein Entscheidungsproblem Π mit der Menge $L_\Pi \subseteq \Sigma_\Pi^*$. $L_\Pi \in \mathbf{PCP}(r(n), q(n))$ („Π liegt in der Klasse $\mathbf{PCP}(r(n), q(n))$“), wenn es einen $(r(n), q(n))$-beschränkten Verifizierer $\mathbf{V}_\Pi$ gibt, der L_Π in folgender Weise akzeptiert:

Für jedes $x \in L_\Pi$ gibt es einen Beweis B_x mit $P(\mathbf{V}_\Pi(x, B_x, t) = \texttt{ja}) = 1$,
für jedes $x \notin L_\Pi$ und alle Beweise B gilt $P(\mathbf{V}_\Pi(x, B, t) = \texttt{nein}) \geq 1/2$.



Für Eingaben $x \in L_\Pi$ gibt es also einen Beweis, der immer akzeptiert wird. Der Versuch, eine Eingabe $x \notin L_\Pi$ zu akzeptieren, scheitert mindestens mit einer Wahrscheinlichkeit $\geq 1/2$.

Im folgenden bezeichnet $poly(n)$ die Klasse der Polynome. Dann gilt beispielsweise

$$\mathbf{P} = \bigcup_{k=0}^{\infty} TIME\left(n^k\right) = TIME\left(poly(n)\right) \text{ und } \mathbf{NP} = \bigcup_{k=0}^{\infty} NTIME\left(n^k\right) = NTIME\left(poly(n)\right).$$

Satz 7.2-2:

$$\mathbf{PCP}\left(r(n),q(n)\right) \subseteq NTIME\left(q(n) \cdot 2^{O(r(n))}\right)$$

Beweis:

Es sei $L_\Pi \in \mathbf{PCP}\left(r(n),q(n)\right)$. Dann gibt es einen $\left(r(n),q(n)\right)$-beschränkten Verifizierer $\mathbf{V}_\Pi$ für L_Π. Aus diesem kann man auf folgende Weise einen Verifizierer $\tilde{\mathbf{V}}_\Pi$ (im Sinne der Definition einer nichtdeterministischen Turingmaschine) für L_Π konstruieren:

Es sei $x \in \Sigma_\Pi^*$ mit $|x| = n$ und B_x ein Beweis. Für jede der $2^{O(r(n))}$ vielen Zufallsfolgen der Länge $O\left(r(n)\right)$ simuliert $\tilde{\mathbf{V}}_\Pi$ in jeweils polynomiell vielen Schritten die Berechnung des Verifizierers $\mathbf{V}_\Pi$ und akzeptiert *x* genau dann, wenn $\mathbf{V}_\Pi$ die Eingabe *x* für jede Zufallsfolge akzeptiert. $\tilde{\mathbf{V}}_\Pi$ ist daher eine nichtdeterministische Turingmaschine, die *x* genau dann akzeptiert, wenn $x \in L_\Pi$ gilt. ///

Setzt man im speziellen für $r(n)$ und $q(n)$ Polynome ein, so gilt sogar:

Satz 7.2-3:

$$\mathbf{PCP}\left(poly(n),poly(n)\right) = NTIME\left(2^{poly(n)}\right)$$

Weiter gelten folgende Aussagen:

Satz 7.2-4:

$$\mathbf{PCP}(0,0) = \mathbf{P},$$

$$\mathbf{PCP}\left(0,poly(n)\right) = \mathbf{NP},$$

$$\mathbf{PCP}\left(\log(n),poly(n)\right) = \mathbf{NP}$$

Aus $\mathbf{PCP}(r(n),q(n)) \subseteq NTIME\left(q(n)\cdot 2^{O(r(n))}\right)$ folgt unmittelbar:

Satz 7.2-5:
$\mathbf{PCP}(\log(n),1) \subseteq \mathbf{NP}$

Die Umkehrung dieser Aussage wird als das wichtigste Resultat der Theoretischen Informatik der letzten 10 Jahre angesehen (Arora, Lund, Motwani, Sudan, Szegedy, 1992). Es hat zu zahlreichen Konsequenzen für die Nicht-Approximierbarkeit von Optimierungsaufgaben aus **NPO** geführt.

Satz 7.2-6:
$\mathbf{PCP}(\log(n),1) = \mathbf{NP}$

„Wie man Beweise verifiziert, ohne sie zu lesen“

Das Ergebnis besagt, daß es zu jeder Menge L_Π für ein Entscheidungsproblem Π aus **NP** einen Verifizierer gibt, der bei jeder Eingabe nur konstant viele Stellen des Beweises liest, die er unter Zuhilfenahme von $O(\log(n))$ vielen zufälligen Bits auswählt, um mit hoher Wahrscheinlichkeit richtig zu entscheiden.

Ein Beispiel für die Anwendung von Satz 7.2-6 ist der Beweis des folgenden Satzes. In ihm wird gezeigt, daß das 3-SAT-Maximierungsproblem nicht in **PTAS** liegt (siehe Kapitel 6.2), d.h. kein polynomiell zeitbeschränktes Approximationsschema besitzt, außer $\mathbf{P} = \mathbf{NP}$.

3-SAT-Maximierungsproblem

Instanz: 1. $I = (K,V)$

$K = \{F_1, F_2, ..., F_m\}$ ist eine Menge von Klauseln, die aus Booleschen Variablen aus der Menge $V = \{x_1, ..., x_n\}$ gebildet werden und jeweils die Form $F_i = (y_{i_1} \vee y_{i_2} \vee y_{i_3})$ für $i = 1, ..., m$ besitzen. Dabei steht y_{i_j} für eine Boolesche Variable (d.h. $y_{i_j} = x_l$) oder für eine negierte Boolesche Variable (d.h. $y_{i_j} = \neg x_l$)

size(*I*) = Anzahl der Zeichen in I

2. SOL(I) = Belegung der Variablen in V mit Wahrheitswerten TRUE oder FALSE, d.h. SOL(I) ist eine Abbildung $f : V \to \{\text{TRUE}, \text{FALSE}\}$
3. Für $f \in \text{SOL}(I)$ ist $m(I, f)$ = Anzahl der Klauseln in K, die durch f erfüllt werden
4. *goal* = max

Es läßt sich ein 2-approximativer Algorithmus für das 3-SAT-Maximierungsproblem angeben (siehe Literatur), d.h. diese Problem liegt in **APX**.

Satz 7.2-7:
Das 3-SAT-Maximierungsproblem liegt nicht in **PTAS**, d.h. es besitzt kein polynomiell zeitbeschränktes Approximationsschema, außer $\mathbf{P} = \mathbf{NP}$.

Beweis:

Es sei L_{SAT} = {F | F ist ein erfüllbarer Boolescher Ausdruck }.
$L_{\text{SAT}} \subseteq \Sigma^*_{\text{BOOLE}}$ = {F | F ist ein Boolescher Ausdruck}, L_{SAT} ist **NP**-vollständig.

Es wird eine Abbildung f definiert, die jedem $F \in \Sigma^*_{\text{BOOLE}}$ eine Instanz $f(F) = (K,V)$ für das 3-SAT-Maximierungsproblem zuordnet, wobei $f(F)$ in polynomieller Zeit aus F berechnet werden kann, und die folgende Eigenschaft besitzt:
Ist $F \in L_{\text{SAT}}$, dann sind alle Klauseln in K erfüllbar.
Ist $F \notin L_{\text{SAT}}$, dann gibt es eine Konstante $e > 0$, so daß mindestens ein Anteil der Größe e aller Klauseln in K nicht erfüllbar ist. Das bedeutet mit $c(F) = |K|$ für $f(F) = (K,V)$:

$m^*(f(F)) = \begin{cases} = c(F) & \text{für } F \in L_{SAT} \\ \leq c(F) \cdot (1-e) & \text{für } F \notin L_{SAT} \end{cases}$. Mit Satz 6.1-8 folgt, daß es keinen polynomiell zeitbeschränkten r-approximativen Algorithmus für das 3-SAT-Maximierungsproblem mit $r < 1/(1-e)$ gibt, außer $\mathbf{P} = \mathbf{NP}$. Daher besitzt das 3-SAT-Maximierungsproblem kein polynomiell zeitbeschränktes Approximationsschema.

Da L_{SAT} **NP**-vollständig ist, gibt es für L_{SAT} nach Satz 7.2-6 einen $(\log(n),1)$-beschränkten Verifizierer $\mathbf{V}_{SAT}$. In $\mathbf{V}_{SAT}$ werden eine Formel F mit $size(F) = n$, ein Beweis B und eine Zufallsfolge t eingegeben. Man kann annehmen, daß das Alphabet, mit dem Beweise formuliert werden, das Alphabet $\Sigma_0 = \{0,1\}$ ist, d.h. jeder Beweis B ist eine 0-1-Folge. Man kann weiterhin annehmen, daß $\mathbf{V}_{SAT}$ genau $q > 2$ Zeichen von B erfragt (auch wenn nicht alle Zeichen des Beweises zur Verifikation benötigt werden). Da höchstens $c \cdot \log(n)$ Bits (mit einer Kon-

stanten *c*) aus t und dann *q* Positionen in *B* gelesen werden, brauchen nur Beweise betrachtet zu werden, die nicht länger als $q \cdot 2^{c \cdot \log(n)} = q \cdot n^c$ sind.

Es wird eine Menge *V* von Booleschen Variablen definiert: Für jede Position eines Beweises wird eine Boolesche Variable in *V* aufgenommen, d.h. $V = \{x_1, x_2, x_3, \ ..., \ x_k\}$ mit $k = q \cdot n^c$. *V* enthält polynomiell viele Variablen und ist in polynomieller Zeit aus *F* konstruierbar. Zwischen den Belegungen der Variablenmenge *V* und den Werten eines Beweises *B* besteht eine bijektive Abbildung *g*: Der Wert der *j*-ten Variablen x_j ist genau dann x_j = TRUE, wenn *B* an der *j*-ten Position den Wert $b_j = 1$hat. Mit $g(x_j)$ werde der Wert an der *j*-ten Position in *B* bezeichnet; entsprechend sei $g^{-1}(b_j)$ der Wert der *j*-ten Variablen in *V*. Die Abbildung *g* werde auf Literate $\neg x_j$ durch $g(\neg x_j) = g(x_j)$ erweitert.

Für die Zufallsfolge t seien die *q* Positionen, die in einem Beweis erfragt werden, die Positionen $t_1, \ ..., \ t_q$. Mit einigen der möglichen Wert-Kombinationen an diesen Positionen kommt $\mathbf{V}_{SAT}$ bei Auswertung zur `ja`-Entscheidung, mit anderen kommt $\mathbf{V}_{SAT}$ zur `nein`-Entscheidung. Mit A_t wird die Menge derjenigen 0-1-Kombinationen bezeichnet, für die $\mathbf{V}_{SAT}$ zur `nein`-Entscheidung kommt. Offensichtlich ist $|A_t| \le |\Sigma_0|^q = 2^q$. Für jedes $(b_1, ..., b_q) \in A_t$ wird eine Formel $(y_{t_1} \vee ... \vee y_{t_q})$ gebildet mit $y_{t_i} = \begin{cases} x_{t_i} & \text{für } b_i = 0 \\ \neg x_{t_i} & \text{für } b_i = 1 \end{cases}$.

Es sei eine Belegung der Variablen in *V* gegeben. Dann gilt:

(*) Unter dieser Belegung hat die Formel $(y_{t_1} \vee ... \vee y_{t_q})$, die aus $(b_1, ..., b_q) \in A_t$ gebildet wurde, genau dann den Wert TRUE, wenn $(g(y_{t_1}), ..., g(y_{t_q})) \neq (b_1, ..., b_q)$ ist.

Denn $(y_{t_1} \vee ... \vee y_{t_q})$ hat genau dann den Wert TRUE, wenn mindestens eines der Literale den Wert TRUE besitzt, etwa y_{t_j} = TRUE. Im Fall $y_{t_j} = x_{t_j}$ bedeutet dieses (nach Definition von y_{t_j}) $b_j = 0$ und $g(y_{t_j}) = 1$, im Fall $y_{t_j} = \neg x_{t_j}$ ist x_{t_j} = FALSE, d.h. $g(y_{t_j}) = 0$, und $b_j = 1$.

Alle so entstehenden Formeln (für alle Zufallsfolgen t) bilden die Formelmenge K'. Diese enthält höchstens $2^{c \cdot \log(n)} \cdot 2^q = 2^q n^c$ viele Formel. Alle Formeln der Menge K' lassen sich so in Klauseln umformen, daß jede neue Klausel genau 3 Literale enthält. Gilt $q = 3$, haben alle Formeln $(y_{t_1} \vee ... \vee y_{t_q})$ bereits die gewünschte Form. Für $q > 3$ werden für jede Formel $G = (y_{t_1} \vee ... \vee y_{t_q})$ $q-3$ neue Variablen $z_{G,1}, ..., z_{G,q-3}$ eingeführt und *G* durch $(y_{t_1} \vee y_{t_2} \vee z_{G,1})$, $(\neg z_{G,1} \vee y_{t_3} \vee z_{G,2})$, ..., $(\neg z_{G,q-4} \vee y_{t_{q-2}} \vee z_{G,q-3})$, $(\neg z_{G,q-3} \vee y_{t_{q-1}} \vee y_{t_q})$ er-

setzt. *G* ist genau dann erfüllbar, wenn alle so entstandenen Formeln erfüllbar sind. Die Variablenmenge *V* wird um die neu hinzugenommen Variablen erweitert; entsprechend wird unter einer Belegung $g(z_{G,j})=1$ genau dann gesetzt, wenn $z_{G,j}=\mathsf{TRUE}$ ist.

Diese eventuell erweiterte Klauselmenge bildet die Menge *K*. Es ist $|K|\le(q-2)\cdot|K'|$, und auch *V* behält eine polynomielle Größe.

Die Berechnung von $f(F)=(K,V)$ erfolgt in polynomieller Zeit.

Ist $F\in L_{\mathrm{SAT}}$, dann gibt es einen Beweis B_F, so daß der Verifizierer $\mathbf{V}_{SAT}$ für jede Zufallsfolgen t die Entscheidung $\mathbf{V}_{SAT}(F,B_F,t)=\mathtt{ja}$ trifft, denn $P(\mathbf{V}_{SAT}(F,B_F,t)=\mathtt{ja})=1$. Ist für eine Zufallsfolgen t die definierte Menge $A_t=\varnothing$, so wurde keine Formel in K' bzw. *K* aufgenommen. Ist $A_t\neq\varnothing$ und sind t_1, ..., t_q die Positionen in B_F, die von $\mathbf{V}_{SAT}$ aufgrund der Zufallsfolge t erfragt werden, etwa mit den Werten $a_1,...,a_q$, dann wird $x_{t_i}=\begin{cases}\mathsf{TRUE} & \text{für } a_i=1\\ \mathsf{FALSE} & \text{für } a_i=0\end{cases}$ gesetzt. Es sei $(b_1,...,b_q)\in A_t$ mit der dazu gebildeten Formel $(y_{t_1}\vee...\vee y_{t_q})\in K'$. Dann ist $(a_1,...,a_q)\neq(b_1,...,b_q)$, denn $(a_1,...,a_q)$ führt auf eine `ja`-Entscheidung, und alle $(b_1,...,b_q)\in A_t$ führen auf eine `nein`-Entscheidung. An mindestens einer Position, etwa an der Position *j* ist $a_j\neq b_j$. Ist $b_j=0$, dann ist $a_j=1$. Das Literal $y_{t_j}=x_{t_j}$ wurde auf `TRUE` gesetzt. Ist $b_j=1$, dann ist $a_j=0$. Das Literal $y_{t_j}=\neg x_{t_j}$ wurde auf `TRUE` gesetzt. In beiden Fällen ist $(y_{t_1}\vee...\vee y_{t_q})$ erfüllt. Das zeigt, daß alle Klauseln in K' und damit alle Klauseln in *K* erfüllbar sind und damit $m^*(f(F))=|K|=c(F)$ gilt.

Ist $F\notin L_{\mathrm{SAT}}$, dann gilt für jeden Beweis *B*, daß der Verifizierer $\mathbf{V}_{SAT}$ für mindestens die Hälfte aller $2^{c\cdot\log(n)}=n^c$ Zufallsfolgen die Entscheidung $\mathbf{V}_{SAT}(F,B,t)=\mathtt{nein}$ trifft, denn für jeden Beweis *B* ist in diesem Fall $P(\mathbf{V}_{SAT}(F,B,t)=\mathtt{nein})\ge 1/2$. Es sei eine Belegung von *V* gegeben, der mittels *g* zugehörige Beweis sei *B*. t sei eine der $2^{c\cdot\log(n)}/2$ Zufallsfolgen, die auf die `nein`-Entscheidung führen. Die Positionen in *B*, die von $\mathbf{V}_{SAT}$ aufgrund der Zufallsfolge t erfragt werden, seien t_1, ..., t_q, die Werte an diesen Positionen in *B* seien $a_1,...,a_q$. Dann ist nach Definition $(a_1,...,a_q)\in A_t$. Die aus $(a_1,...,a_q)$ gebildete Formel $(y_{t_1}\vee...\vee y_{t_q})$ hat wegen (*) den Wert `FALSE`. Für $q>3$ wurde durch obige Konstruktion die Formel $(y_{t_1}\vee...\vee y_{t_q})$ durch $q-3$ Formeln ersetzt. Es läßt sich zeigen, daß im vorliegenden Fall mindestens eine dieser Formeln den Wert `FALSE` trägt, unabhängig davon, wie die dort enthaltenen neuen Variablen $z_{G,j}$ belegt sind. Zu jeder der $2^{c\cdot\log(n)}/2$ Zufallsfolgen, die auf die

`nein`-Entscheidung führen, gibt es also mindestens eine Formel in *K*, die nicht erfüllt ist. Daher beträgt der Anteil nichterfüllbarer Klauseln mindestens
$2^{c\cdot\log(n)} / \left(2\cdot|K|\right) \geq 2^{c\cdot\log(n)} / \left(2\cdot(q-2)\cdot 2^{q+c\cdot\log(n)}\right) = 1/\left((q-2)\cdot 2^{q+1}\right)$.
Mit $e = 1/\left((q-2)\cdot 2^{q+1}\right)$ folgt die Behauptung. ///

8 Übungsaufgaben

Im folgenden werden Übungsaufgaben zu den einzelnen Kapiteln bereitsgestellt. Einige der Übungsaufgaben werden in der Veranstaltung behandelt. Aufgaben, die eventuell einiges Nachdenken und Nachschlagen in der Fachliteratur erfordern, sind mit dem Zeichen ¤ gekennzeichnet.

Aufgabe 1.1.1: Es sei $\Sigma = \{a, b, c\}$. Bestimmen Sie Σ^3 und Σ^*.

Aufgabe 1.1.2: Es sei a ein Buchstabe eines endlichen Alphabets. Bestimmen Sie $\{a^2\}^+$ und $\{a^7\}^* \cdot \{a^3\}$.

Aufgabe 1.1.3: Es seien A und B Mengen. Zeigen Sie: $(A^*)^* = A^*$, $(A \cup B)^* = (A^* \cdot B^*)^*$ und $\varnothing^* = \{e\}$.

Aufgabe 1.1.4: Können L^* bzw. L^+ jemals die leere Menge sein? Unter welchen Umständen sind L^* bzw. L^+ endliche Mengen?

Aufgabe 1.1.5: Es seien A und B Mengen. Beweisen oder widerlegen Sie durch ein entsprechendes Gegenbeispiel die Behauptungen
$(A \cup B)^* = A^* \cup B^*$, $(A \cdot B \cup A)^* \cdot A = A \cdot (B \cdot A \cup A)^*$

Aufgabe 1.1.6: ¤ Für ein endliches Alphabet $\Sigma = \{a_1, ..., a_n\}$ kann man die Wörter aus Σ^* in lexikographischer Reihenfolge durchnumerieren (siehe Kapitel 1.1). Welche Nummer enthält in dieser Numerierung das Wort $a_{i_1} ... a_{i_k}$ der Länge k? Überlegen Sie sich die Lösung der Aufgabe zunächst für eine zweielementige Menge $\Sigma = \{0, 1\}$.

Aufgabe 1.1.7: ¤ Zeigen Sie, daß die Menge der rationalen Zahlen gleichmächtig mit der Menge der natürlichen Zahlen ist. Zur Erinnerung: Man kann die rationalen Zahlen durch $\mathbf{Q} = \left\{ r/t \mid r \in \mathbf{Z} \text{ und } t \in \mathbf{N}_{>0} \right\} = \{(r, t) \mid r \in \mathbf{Z} \text{ und } t \in \mathbf{N}_{>0}\}$ definieren.

Aufgabe 1.1.8: Zeigen Sie: Die Mächtigkeit einer unendlich abzählbaren Menge ändert sich nicht, wenn man endlich viele Elemente entfernt.

Aufgabe 1.1.9: ¤ Zeigen Sie: Die Vereinigung abzählbar vieler abzählbarer Mengen ist wieder abzählbar.

Aufgabe 1.1.10: Es seien $f : \mathbf{N} \to \mathbf{R}$ und $g : \mathbf{N} \to \mathbf{R}$ zwei Funktionen. Zeigen Sie:
$O(f(n)) \cdot O(g(n)) \subseteq O(f(n) \cdot g(n))$,
$O(f(n) \cdot g(n)) \subseteq |f(n)| \cdot O(g(n))$ und
$O(f(n)) + O(g(n)) \subseteq O(|f(n)| + |g(n)|)$.

Aufgabe 2.1.1: Geben Sie eine Turingmaschine mit 2 Bändern und dem Eingabealphabet $I = \{0,1\}$ durch ihre Überführungsfunktion an, die zunächst das (neue) Zeichen # auf das 2. Band schreibt und dann das Eingabewort vom 1. Band auf das 2. Band kopiert.

Aufgabe 2.1.2: Geben Sie eine Turingmaschine mit dem Eingabealphabet $I = \{0,1\}$ durch ihre Überführungsfunktion an, die das Eingabewort auf dem 1. Band dupliziert, d.h. steht das Wort $w = a_1 \ldots a_n$ im Startzustand auf dem Eingabeband, so steht dort abschließend das Wort $ww = a_1 \ldots a_n a_1 \ldots a_n$.

Aufgabe 2.1.3: ¤ Geben Sie eine Turingmaschine *TM* mit $L(TM) = \{a^n b^n c^n \mid n \in \mathbf{N}\}$ an. Welche Speicherplatzkomplexität hat Ihre Turingmaschine?

Aufgabe 2.1.4: Beschreiben Sie die Arbeitsweise einer Turingmaschine, die die Funktion $f : \begin{cases} \mathbf{N} \times \mathbf{N} & \to & \mathbf{N} \\ (n,m) & \to & n+m \end{cases}$ berechnet. Die Eingabe dieser Turingmaschine auf dem 1. Band habe die Form $bin(n)\#bin(m)$, wobei # ein von 0 und 1 verschiedenes Zeichen ist. Am Ende der Berechnung soll auf dem Ausgabeband $bin(n+m)\#$ stehen.

Aufgabe 2.2.1: Zeigen Sie, daß das in der Vorlesung behandelte RAM-Programm **P** zur Akzeptanz von

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

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

Aufgabe 2.2.2: ¤ Die folgende Pascal-Prozedur 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;

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

Man könnte sich die Prozedur als in ein RAM-Programm umgesetzt denken. Dieses hätte dann nicht 6, sondern $c \cdot 6$ viele Anweisungen mit einer Konstanten $c > 0$. Anstatt die Kosten des entsprechenden RAM-Programms zu ermitteln, kann man dieses auch gleich für die Pascal-Prozedur machen, wenn man ohnehin nur an der Größenordnung der Komplexität interessiert ist. Beim uniformen Kostenkriterium zählt man dabei die Anzahl ausgeführter Pascal-Anweisungen; beim logarithmischen Kostenkriterium berücksichtigt man die Stellenzahl der in den beteiligten Variablen gespeicherten Werte. Enthält die Variable `idx` beispielsweise den Wert *m*, so verursacht die einmalige Ausführung der Anweisung `idx:=idx-1` Kosten der Ordnung $O(\log(m))$. Enthält die Variable `c` den Wert *k*, so verursacht die einmalige Ausführung der Anweisung `c:=c*c` Kosten der Ordnung $O(\log(k))$.

(a) Bestimmen Sie die Kosten der Ausführung der Prozedur bei Eingabe von *n* nach dem uniformen Kostenkriterium. Was fällt Ihnen auf?

(b) Bestimmen Sie die Kosten der Ausführung der Prozedur bei Eingabe von *n* nach dem logarithmischen Kostenkriterium.

(c) Falls die Laufzeit eines Verfahrens nicht nur von der Anzahl der Eingabeparameter, sondern auch von deren numerischen Werten abhängt, dann nimmt man als Meßgröße für die Laufzeitkomplexität die Anzahl der Binärstellen, die benötigt werden, um die Eingabeparameter darzustellen (oder einen Wert, der proportional hierzu ist). Man wendet dann das uniforme Kostenkriterium an. Im vorliegenden Fall kann man als Größe der Eingabe den Wert $k = \lceil \log_2(n+1) \rceil$. Welche Laufzeitkomplexität hat die Pascal-Prozedur unter diesen Voraussetzungen? Wie verbindet sich das Ergebnis mit den Resultaten aus (a) und (b)?

Aufgabe 2.2.3: ¤ Entwerfen Sie ein RAM-Programm mit einer Zeitkomplexität der Ordnung $O(\log(n))$ unter dem uniformen Kostenkriterium zur Berechnung von n^n.

Aufgabe 2.6.1: In der Vorlesung wird eine 3-NDTM zur Lösung des Partitionenproblems mit ganzzahligen Eingabewerten angegeben. Das Partitionenproblem ist folgendes Entscheidungsproblem:

Instanz: $I = \{a_1, ..., a_n\}$
I ist eine Menge von *n* natürlichen Zahlen.

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

Für eine Instanz $I = \{a_1, ..., a_n\}$ sei $B = \sum_{i=1}^{n} a_i$. Falls *B* ungerade ist, lautet die Entscheidung für *I* „nein“, also kann *B* im folgenden als gerade angenommen werden. Es wird ein Boolescher Ausdruck $T[i, j]$ definiert durch
$T[i, j] = \mathsf{TRUE}$ genau dann, wenn es eine Teilmenge von $\{a_1, ..., a_i\}$ gibt, deren Elemente sich auf genau *j* aufsummieren.

(a) Welche Wahrheitswerte haben $T[1,0]$, $T[1, a_1]$ und $T[1, j]$ für $j \neq 0$ und $j \neq a_1$?

(b) Welcher Zusammenhang besteht zwischen der ja/nein-Entscheidung für eine Instanz $I = \{a_1, ..., a_n\}$ und dem Wert $T[n, B/2]$?

(c) Zeigen Sie: $T[i, j] = \mathsf{TRUE}$ genau dann, wenn entweder $T[i-1, j] = \mathsf{TRUE}$ oder wenn $a_i \leq j$ und $T[i-1, j-a_i] = \mathsf{TRUE}$ ist.

(d) Es sei I = {1, 9, 5, 3, 8 }. Berechnen Sie $T[i, j]$ für $i = 1, ..., n$ und $j = 0, ..., B/2$.

(e) Entwickeln Sie einen deterministischen Algorithmus, der bei Eingabe einer Instanz $I = \{a_1, ..., a_n\}$ für das Partitionenproblem eine ja/nein-Entscheidung trifft und dabei eine Anzahl von Anweisungen der Größenordnung $O(n \cdot B)$ ausführt. Warum handelt es sich hierbei *nicht* um einen Algorithmus, dessen Laufzeit polynomiell in der Größe der Eingabe ist?

Aufgabe 2.6.2: Zeigen Sie, daß die folgenden Funktionen $S_i : \mathbf{N} \to \mathbf{N}$ platzkonstruierbar sind, indem Sie deterministische Turingmaschinen angeben, die bei Eingabe eines Wortes der Länge n ein spezielles Symbol in die $S_i(n)$-te Zelle eines ihrer Bänder schreibt, ohne jeweils mehr als $S_i(n)$ viele Zellen auf allen Bändern zu verwenden.

(a) $S_1(n) = n^2$

(b) $S_2(n) = n^3 - n^2 + 1$

(c) $S_3(n) = 2^n$

(d) $S_4(n) = n!$

Aufgabe 2.6.3: ¤ Zeigen Sie:

Wird L von einer k-NDTM $TM = (Q, \Sigma, I, d, b, q_0, q_{accept})$ mit Zeitkomplexität $T(n)$ akzeptiert, dann wird L von einer 1-NDTM TM' mit Zeitkomplexität der Ordnung $O(T^2(n))$ akzeptiert. Wie läßt sich das Ergebnis auf deterministische Turingmaschinen übertragen?

Aufgabe 3.1.2: Zeigen Sie, daß die folgenden Mengen entscheidbar sind. Dabei werden natürliche Zahlen in die entsprechenden Turimgmaschinen in Binärkodierung eingegeben.

(a) $\{n^2 \mid n \in \mathbf{N}\}$

(b) $\{2^n \mid n \in \mathbf{N}\}$

(c) $\{n \mid n \in \mathbf{N}, n \geq 2$ und $2n = p + q$ mit Primzahlen p und $q\}$; ist die Menge $\{n \mid n \in \mathbf{N}, n \geq 2$ und $2n = p - q$ mit Primzahlen p und $q\}$ entscheidbar?

Aufgabe 3.1.3: Zeigen Sie:

(a) Die Menge

$$L_{uni} = L(UTM) = \left\{ u\#w \;\middle|\; \begin{array}{l} u \in \{0,1\}^*, w \in \{0,1\}^*, \\ VERIFIZIERE\_TM(w) = \mathsf{TRUE} \text{ und } u \in L(K_w) \end{array} \right\} \subseteq \{0,1,\#\}^*$$ ist

rekursiv aufzählbar.

(b) Die Menge $\{0,1\}^* \setminus L_d$ ist nicht entscheidbar; hierbei ist L_d die in der Vorlesung definierte Menge $L_d = \{w \mid w = w_i \text{ und } w_i \notin L(K_{w_i})\}$.

(c) Die Menge L_{uni} aus Aufgabe (a) ist nicht entscheidbar.

Aufgabe 3.1.4: ¤ Zeigen Sie:

Die Menge

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

ist nicht rekursiv aufzählbar. Gehen Sie dabei vor wie im Beweis zu $L_e = \{w \mid L(K_w) \text{ ist entscheidbar}\}$. Zu gegebenen $u \in \{0,1\}^*$ und $w \in \{0,1\}^*$ mit *VERIFIZIERE_TM*(*w*) = TRUE entwerfen Sie eine Turingmaschine $TM_{u,w}$ mit

$$L(TM_{u,w}) = \begin{cases} \{0,1\}^* \# \{0,1\}^* & \text{falls } u \in L(K_w) \\ L_{uni} & \text{falls } u \notin L(K_w) \end{cases}.$$

Aufgabe 3.1.5: Es sei Σ ein endliches Alphabet und $L \subseteq \Sigma^*$ rekursiv aufzählbar. Zeigen Sie, daß dann $L \leq L_H$ gilt. Wie kann man diese Aussage interpretieren?

Aufgabe 3.1.6: Es sei Σ ein endliches Alphabet und $L \subseteq \Sigma^*$ entscheidbar. Für die Menge $M \subseteq \Sigma^*$ gelte $M \neq \emptyset$ und $M \neq \Sigma^*$. Zeigen Sie, daß dann $L \leq M$ gilt. Wie kann man diese Aussage interpretieren?

Aufgabe 3.1.7: Formalisieren Sie die folgenden umgangssprachlichen Feststellungen und beweisen Sie die entsprechenden Aussagen:

(a) Es ist nicht entscheidbar, ob ein beliebiges Programm (in einer realen Programmiersprache) für eine Eingaben in eine unendliche Schleife läuft.

(b) Es ist nicht entscheidbar, ob ein beliebiges Programm (in einer realen Programmiersprache) überhaupt eine Ausgabe erzeugt.

Aufgabe 4.1.1: Geben Sie eine Grammatik zur Erzeugung der Sprache $(0^*1^*)^*$ an.

Aufgabe 4.3.1: Zeigen Sie, daß die Sprache $L = \{0^{2^i} \mid i \geq 0\}$ kontextsensitiv ist.

Aufgabe 4.4.1: Zeigen Sie, daß die Sprache $L = \left\{ 0^{2^i} \mid i \geq 0 \right\}$ nicht kontextfrei ist.

Aufgabe 4.4.2: Es seien *a*, *b* und *c* paarweise verschiedene Symbole. Welche der folgenden Sprachen sind kontextfrei? Begründen Sie Ihre Antwort.

(a) $\left\{ a^k b^k c^i \mid k \in \mathbf{N}, i \in \mathbf{N} \right\}$

(b) $\left\{ a^i b^j c^k \mid i \in \mathbf{N},\ j \in \mathbf{N}, k \in \mathbf{N},\ i > j \text{ oder } j > k \right\}$

(c) $\left\{ a^i b^j a^i b^j \mid i \in \mathbf{N},\ j \in \mathbf{N} \right\}$

(d) $\left\{ ww \mid w \in \{a\}^* \right\}$

(e) $\left\{ ww \mid w \in \{a, b\}^* \right\}$

Aufgabe 4.4.3: Beschreiben Sie, wie ein nichtdeterministischer Kellerautomat mit zwei Kellern das Verhalten einer Turingmaschine simulieren kann.

Aufgabe 4.5.1: ¤ Beweisen Sie Satz 4.5-1.

Aufgabe 4.5.2: ¤ Zeigen Sie: Wird die Sprache *L* von einem nichtdeterministischen endlichen Automaten erkannt, dann gibt es einen deterministischen Automaten, der *L* erkennt.

Aufgabe 4.5.3: Es seien *a* und *b* verschiedene Buchstaben. Zeigen Sie, daß die folgenden Sprachen regulär sind:

(a) $\left\{ w \mid w \in \{a, b\}^*, \text{die Anzahl von } a\text{'s ist gerade, die Anzahl von } b\text{'s ist ungerade} \right\}$

(b) $\left\{ w \mid w \in \{a, b\}^*, w \text{ enthält höchstens ein Paar aufeinanderfolgende } a\text{'s oder } b\text{'s} \right\}$

(c) $\left\{ w \mid w \in \{a, b\}^*, w \text{ enthält nicht die Zeichenkette } aba \right\}$.

Aufgabe 4.5.4: Es seien *a* und *b* verschiedene Buchstaben. Zeigen Sie, daß die folgenden Sprachen nicht regulär sind:

(a) $\left\{ a^n b^n \mid n \in \mathbf{N} \right\}$

(b) $\left\{ a^n \mid n \in \mathbf{N}, n \text{ ist eine Quadratzahl} \right\}$

(c) $\left\{ a^n \mid n \in \mathbf{N}, n \text{ ist eine Primzahl} \right\}$.

Aufgabe 4.5.5: ¤ Es sei L eine kontextfreie Sprache und R eine reguläre Sprache. Zeigen Sie:

(a) $L \cap R$ ist kontextfrei.

(b) Es seien a und b verschiedene Buchstaben. Mit Hilfe des *uvwyx*-Theorems für kontextfreie Sprachen läßt sich zeigen, daß die Sprache $L_1 = \{ a^n b^m a^n b^m \mid n \in \mathbf{N}, m \in \mathbf{N} \}$ nicht kontextfrei ist. Warum ist auch $L_2 = \{ ww \mid w \in \{ a, b \}^+ \}$ nicht kontextfrei?

(c) Es seien c_1, ..., c_n neue paarweise verschiedene Buchstaben. Warum ist $L_3 = \{ x_1 w x_2 w x_3 \mid w \in \{ a, b \}^+, x_i \in \{c_1, ..., c_n\}^* \text{ für } i = 1, 2, 3 \}$ nicht kontextfrei?

(d) Es sei L_P die Menge aller korrekten Pascal-Programme (dasselbe kann man mit Java-, C++-, Cobolprogrammen usw. machen). Es wird eine Teilmenge L_4 korrekter Pascal-Programme definiert durch

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

Zusätzlich wird die reguläre Menge R definiert durch

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

Zeigen Sie mit Hilfe von L_4 und R, daß L_P nicht kontextfrei ist.

Aufgabe 4.6.1: Begründen Sie die Gültigkeit der in Kapitel 4.6 aufgeführten Eigenschaften der verschiedenen Sprachklassen.

Aufgabe 5.3.1: Zeigen Sie:

(a) Die Klasse **NP** ist abgeschlossen bezüglich Schnitt und Vereinigung.

(b) Ist L **NP**-vollständig und ist $L_0 \in \mathbf{NP}$ mit $L \leq_m L_0$, dann ist auch L_0 **NP**-vollständig.

(c) Mit $L \in \mathbf{NP}$ gilt auch $L^* \in \mathbf{NP}$.

Aufgabe 5.3.2: ¤ Zeigen Sie, daß die Sprache

$L = \{ (u, w, 0^t) \mid \text{die nichtdeterministische Turingmaschine } K_w \text{ akzeptiert } u \text{ in höchstens } t \text{ Schritten} \}$

NP-vollständig ist.

Aufgabe 5.3.3: Zeigen Sie: **P** = **NP** impliziert, daß $L = \{ 0, 1 \}$ **NP**-vollständig ist.

Aufgabe 5.4.1: ¤ Zeigen Sie (indem Sie beispielsweise die einschlägige Fachliteratur konsultieren) die Gültigkeit der nicht in der Vorlesung behandelten Reduktionen:

SAT $\leq_m^p$ 3-CSAT $\leq_m^p$ RUCKSACK $\leq_m^p$ PARTITION $\leq_m^p$ BIN PACKING und

3-CSAT $\leq_m^p$ KLIQUE und

3-CSAT $\leq_m^p$ GERICHTETER HAMILTONKREIS

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

Literaturauswahl

Die folgende Literaturauswahl wurde im vorliegenden Texte verwendet. Im Text sind die entsprechenden Stellen jedoch nicht explizit gekennzeichnet.

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.

Ausiello, G.; Crescenzi, P.; Gambosi, G.; Kann, V.; Marchetti-Spaccamela, A.; Protasi, M.: **Complexity and Approximation**, Springer, 1999.

Bartholomé, A.; Rung, J.; Kern, H.: **Zahlentheorie für Einsteiger**, Vieweg, 1995.

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

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

Dewdney, A.K.: **Der Turing Omnibus**, Springer, 1995.

Erk, K.; Priese, L.: **Theoretische Informatik**, 2. Aufl., Springer 2002.

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

Harel, D.: **Algorithmics**, 2nd Ed., Addison Wesley, 1992.

Hopcroft, J.E.; Ullman, J.D.: **Einführung in die Automatentheorie, formale Sprachen und Komplexitätstheorie**, Addison-Wesley, 1989.

Hromkovic, J.: **Algorithmische Konzepte der Informatik**, Teubner, 2001.

Paul, W.J.: **Einführung in die Komplexitätstheorie**, Teubner, 1990.

Salomaa, A.: **Public-Key Cryptography**, 2. Aufl., Springer, 1996.

Schöning, U.: **Theoretische Informatik kurzgefaßt**, 3. Aufl., Spektrum Akademischer Verlag, 1997.

Vossen, G.; Witt, K.-U.: **Grundlagen der Theoretischen Informatik mit Anwendungen**, Vieweg, 2000.

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

In der Reihe FINAL sind bisher erschienen:

1. Jahrgang 1991:

1. Hinrich E. G. Bonin; Softwaretechnik, Heft 1, 1991 (ersetzt durch Heft 2, 1992).
2. Hinrich E. G. Bonin (Herausgeber); Konturen der Verwaltungsinformatik, Heft 2, 1991 (überarbeitet und erschienen im Wissenschaftsverlag, Bibliographisches Institut & F. A. Brockhaus AG, Mannheim 1992, ISBN 3-411-15671-6).

2. Jahrgang 1992:

1. Hinrich E. G. Bonin; Produktionshilfen zur Softwaretechnik --- Computer-Aided Software Engineering --- CASE, Materialien zum Seminar 1992, Heft 1, 1992.
2. Hinrich E. G. Bonin; Arbeitstechniken für die Softwareentwicklung, Heft 2, 1992 (3. überarbeitete Auflage Februar 1994), PDF-Format (Passwort: arbeiten).
3. Hinrich E. G. Bonin; Object-Orientedness --- a New Boxologie, Heft 3, 1992.
4. Hinrich E. G. Bonin; Objekt-orientierte Analyse, Entwurf und Programmierung, Materialien zum Seminar 1992, Heft 4, 1992.
5. Hinrich E. G. Bonin; Kooperative Produktion von Dokumenten, Materialien zum Seminar 1992, Heft 5, 1992.

3. Jahrgang 1993:

1. Hinrich E. G. Bonin; Systems Engineering in Public Administration, Proceedings IFIP TC8/ WG8.5: Governmental and Municipal Information Systems, March 3--5, 1993, Lüneburg, Heft 1, 1993 (überarbeitet und erschienen bei North-Holland, IFIP Transactions A-36, ISSN 0926-5473).
2. Antje Binder, Ralf Linhart, Jürgen Schultz, Frank Sperschneider, Thomas True, Bernd Willenbockel; COTEXT --- ein Prototyp für die kooperative Produktion von Dokumenten, 19. März 1993, Heft 2, 1993.
3. Gareth Harries; An Introduction to Artificial Intelligence, April 1993, Heft 3, 1993.
4. Jens Benecke, Jürgen Grothmann, Mark Hilmer, Manfred Hölzen, Heiko Köster, Peter Mattfeld, Andre Peters, Harald Weiss; ConFusion --- Das Produkt des AWÖ-Projektes 1992/93, 1. August 1993, Heft 4, 1993.
5. Hinrich E. G. Bonin; The Joy of Computer Science --- Skript zur Vorlesung EDV ---, September 1993, Heft 5, 1993 (4. ergänzte Auflage März 1995).
6. Hans-Joachim Blanke; UNIX to UNIX Copy --- Interactive application for installation and configuration of UUCP ---, Oktober 1993, Heft 6, 1993.

4. Jahrgang 1994:

1. Andre Peters, Harald Weiss; COMO 1.0 --- Programmierumgebung für die Sprache COBOL --- Benutzerhandbuch, Februar 1994, Heft 1, 1994.

2. Manfred Hölzen; UNIX-Mail --- Schnelleinstieg und Handbuch ---, März 1994, Heft 2, 1994.
3. Norbert Kröger, Roland Seen; EBrain --- Documentation of the 1994 AWÖ-Project Prototype ---, June 11, 1994, Heft 3, 1994.
4. Dirk Mayer, Rainer Saalfeld; ADLATUS --- Documentation of the 1994 AWÖ-Project Prototype -- -, July 26, 1994, Heft 4, 1994.
5. Ulrich Hoffmann; Datenverarbeitungssystem 1, September 1994, Heft 5, 1994. (2. überarbeitete Auflage Dezember 1994)
6. Karl Goede; EDV-gestützte Kommunikation und Hochschulorganisation, Oktober 1994, Heft 6 (Teil 1), 1994.
7. Ulrich Hoffmann; Zur Situation der Informatik, Oktober 1994, Heft 6 (Teil 2), 1994.

5. Jahrgang 1995:

1. Horst Meyer-Wachsmuth; Systemprogrammierung 1, Januar 1995, Heft 1, 1995.
2. Ulrich Hoffmann; Datenverarbeitungssystem 2, Februar 1995, Heft 2, 1995.
3. Michael Guder / Kersten Kalischefski / Jörg Meier / Ralf Stöver / Cheikh Zeine; OFFICE-LINK --- Das Produkt des AWÖ-Projektes 1994/95, März 1995, Heft 3, 1995.
4. Dieter Riebesehl; Lineare Optimierung und Operations Research, März 1995, Heft 4, 1995.
5. Jürgen Mattern / Mark Hilmer; Sicherheitsrahmen einer UTM-Anwendung, April 1995, Heft 5, 1995.
6. Hinrich E. G. Bonin; Publizieren im World-Wide Web --- HyperText Markup Language und die Kunst der Programmierung ---, Mai 1995, Heft 6, 1995
7. Dieter Riebesehl; Einführung in Grundlagen der theoretischen Informatik, Juli 1995, Heft 7, 1995
8. Jürgen Jacobs; Anwendungsprogrammierung mit Embedded-SQL, August 1995, Heft 8, 1995
9. Ulrich Hoffmann; Systemnahe Programmierung, September 1995, Heft 9, 1995 (ersetzt durch Heft 1, 1999).
10. Klaus Lindner; Neuere statistische Ergebnisse, Dezember 1995, Heft 10, 1995

6. Jahrgang 1996:

1. Jürgen Jacobs / Dieter Riebesehl; Computergestütztes Repetitorium der Elementarmathematik, Februar 1996, Heft 1, 1996
2. Hinrich E. G. Bonin; "Schlanker Staat" & Informatik, März 1996, Heft 2, 1996
3. Jürgen Jacobs; Datenmodellierung mit dem Entity-Relationship-Ansatz, Mai 1996, Heft 3, 1996
4. Ulrich Hoffmann; Systemnahe Programmierung, (2. überarbeitete Auflage von Heft 9, 1995), September 1996, Heft 4, 1996 (ersetzt durch Heft 1, 1999).
5. Dieter Riebesehl; Prolog und relationale Datenbanken als Grundlagen zur Implementierung einer NF2-Datenbank (Sommer 1995), November 1996, Heft 5, 1996

7. Jahrgang 1997:

1. Jan Binge, Hinrich E. G. Bonin, Volker Neumann, Ingo Stadtsholte, Jürgen Utz; Intranet-/Internet- Technologie für die Öffentliche Verwaltung --- Das AÖW-Projekt im WS96/97 --- (Anwendungen in der Öffentlichen Verwaltung), Februar 1997, Heft 1, 1997
2. Hinrich E. G. Bonin; Auswirkungen des Java-Konzeptes für Verwaltungen, FTVI'97, Oktober 1997, Heft 2, 1997

8. Jahrgang 1998:

1. Hinrich E. G. Bonin; Der Java-Coach, Oktober 1998, Heft 1, 1998 (CD-ROM, PDF-Format; aktuelle Fassung)
2. Hinrich E. G. Bonin (Hrsg.); Anwendungsentwicklung WS 1997/98 --- Programmierbeispiele in COBOL & Java mit Oracle, Dokumentation in HTML und tcl/tk, September 1998, Heft 2, 1998 (CD-ROM)
3. Hinrich E. G. Bonin (Hrsg); Anwendungsentwicklung SS 1998 --- Innovator, SNiFF+, Java, Tools, Oktober 1998, Heft 3, 1998 (CD-ROM)
4. Hinrich E. G. Bonin (Hrsg); Anwendungsentwicklung WS 1998 --- Innovator, SNiFF+, Java, Mail und andere Tools, November 1998, Heft 4, 1998 (CD-ROM)
5. Hinrich E. G. Bonin; Persistente Objekte --- Der Elchtest für ein Java-Programm, Dezember 1998, Heft 5, 1998 (CD-ROM)

9. Jahrgang 1999:

1. Ulrich Hoffmann; Systemnahe Programmierung (3. überarbeitete Auflage von Heft 9, 1995), Juli 1999, Heft 1, 1999 (CD-ROM und Papierform), Postscript-Format, zip-Postscript-Format, PDF- Format und zip-PDF-Format.

10. Jahrgang 2000:

1. Hinrich E. G. Bonin; Citizen Relationship Management, September 2000, Heft 1, 2000 (CD-ROM und Papierform), PDF-Format --- Password: arbeiten
2. Hinrich E. G. Bonin; WI>DATA --- Eine Einführung in die Wirtschaftsinformatik auf der Basis der Web_Technologie, September 2000, Heft 2, 2000 (CD-ROM und Papierform), PDF-Format --- Password: arbeiten
3. Ulrich Hoffmann; Angewandte Komplexitätstheorie, November 2000, Heft 3, 2000 (CD-ROM und Papierform), PDF-Format
4. Hinrich E. G. Bonin; Der kleine XMLer, Dezember 2000, Heft 4, 2000 (CD-ROM und Papierform), PDF-Format, aktuelle Fassung --- Password: arbeiten

11. Jahrgang 2001:

1. Hinrich E. G. Bonin (Hrsg.): 4. SAP-Anwenderforum der FHNON, März 2001, (CD-ROM und Papierform), Downloads & Videos.
2. J. Jacobs / G. Weinrich; Bonitätsklassifikation kleiner Unternehmen mit multivariater linear Diskriminanzanalyse und Neuronalen Netzen; Mai 2001, Heft 2, 2001, (CD-ROM und Papierform), PDF-Format und MS Word DOC-Format --- Password: arbeiten
3. K. Lindner; Simultanttestprozedur für globale Nullhypothesen bei beliebiger Abhängigkeitsstruktur der Einzeltests, September 2001, Heft 3, 2001 (CD-ROM und Papierform).

12. Jahrgang 2002:

1. Hinrich E. G. Bonin: Aspect-Oriented Software Development. März 2002, Heft 1, 2002 (CD-ROM und Papierform), PDF-Format --- Password: arbeiten.
2. Hinrich E. G. Bonin: WAP & WML --- Das Projekt Jagdzeit ---. April 2002, Heft 2, 2002 (CD-ROM und Papierform), PDF-Format --- Password: arbeiten.

Herausgeber:

Prof. Dr. Dipl.-Ing. Dipl.-Wirtsch.-Ing. Hinrich E. G. Bonin Fachhochschule Nordostniedersachsen (FH NON), Volgershall 1, D-21339 Lüneburg, email: bonin@fhnon.de

Verlag:

Eigenverlag (Fotographische Vervielfältigung), FH NON

Erscheinungsweise:

ca. 4 Hefte pro Jahr Für unverlangt eingesendete Manuskripte wird nicht gehaftet. Sie sind aber willkommen.