University of Applied Sciences

Fachhochschule Nordostniedersachsen

Lüneburg
Buxtehude
Suderburg

Arbeitsberichte aus der Informatik

Incl. CD

FinAL

üneburg

rbeitsberichte

formatik

achhochschule Nordostniedersachsen

Ausgewählte Kapitel der Theoretischen Informatik

Ulrich Hoffmann

12. Jahrgang, Heft 3, August 2002, ISSN 0939-8821
Technical Reports and Working Papers
Hrsg: Hinrich E. G. Bonin
Volgershall 1, D-21339 Lüneburg
Phone: xx49.4131.677175 Fax: xx49.4131.677140

Inhaltsverzeichnis

1 Einleitung

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

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

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

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

Die Behandlung dieser Fragestellungen führt auf die Betrachtung von Modellen der Berechenbarkeit. Hierbei spielt die Turingmaschine eine herausragende Rolle. Ergänzend wird

wegen seiner inhaltlichen Nähe zu realen Computern das Modell der Random Access Maschine behandelt. Auf die Darstellung von Theorien wie die m-rekursiven Funktionen oder das l-Kalkül soll hier (im wesentlichen aus Platzgründen) verzichtet werden, obwohl sie eine hohe mathematische Eleganz und Relevanz aufweisen.

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

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

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

1.1 Einige grundlegende Bezeichnungen

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

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

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

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

Alternativ kann man auch folgendermaßen argumentieren:

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

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

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

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

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

Logisch äquivalent ist folgende Argumentation:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Formal lassen sich die Wörter über Σ wie folgt definieren:

(i) e ist ein Wort über Σ mit $|e| = 0$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

gezeigt ist.

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

Eine Menge M ist **endlich von der Mächtigkeit (Kardinalität)** n, wenn es eine bijektive Abbildung $f : M \to \{1, ..., n\}$ gibt, d.h. man kann die Elemente in M mit den natürlichen Zahlen 1, ..., n durchnumerieren: $M = \{m_1, ..., m_n\}$.

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

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

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

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

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

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

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

Satz 1.1-1:

Es sei Σ ein endliche Alphabet. Dann gilt:

1. Σ^* ist abzählbar unendlich.

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

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

Satz 1.1-2:

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

Beweis:

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

Ist *M* dagegen abzählbar unendlich, so gilt:

Satz 1.1-3:

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

Beweis:

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

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

Satz 1.1-4:

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

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

Einige Regeln:

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

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

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

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

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

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

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

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

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

Satz 1.1-5:

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

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

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

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

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

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

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

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

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

Insgesamt ergibt sich damit

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

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

darstellt[2]. Innerhalb eines Booleschen Ausdrucks kommen Variablen vor, die mit dem Zeichen *x* beginnen und von einer Folge von Zeichen aus {0, 1} ergänzt werden. Diese ergänzende 0-1-Folge, die an das Zeichen *x* „gebunden" ist, wird als Indizierung der Variablen interpretiert. Im folgenden werden daher Variablen als indizierte Variablen geschrieben, wobei die indizierende 0-1-Folge als Binärzahl in Dezimaldarstellung angegeben wird. Kommen die Zeichen 0 bzw. 1 innerhalb eines Booleschen Ausdrucks nicht an das Zeichen *x* gebunden vor, so werden sie als Konstanten FALSCH (FALSE) bzw. WAHR (TRUE) interpretiert.

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

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

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

x	$\neg x$
FALSE	TRUE
TRUE	FALSE

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

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

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

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

Ein Boolescher Ausdruck *F* ist in **konjunktiver Normalform**, wenn *F* die Form

---

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

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

und jedes F_i die Form

$F_i = (y_{i_1} \vee ... y_{i_k})$

hat, wobei y_{i_j} ein Literal oder eine Konstante 0 oder 1 ist, d.h. für eine Variable (d.h. $y_{i_j} = x_l$) oder für eine negierte Variable (d.h. $y_{i_j} = \neg x_l$) oder für eine Konstante 0 bzw. 1 steht.

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

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

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

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

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

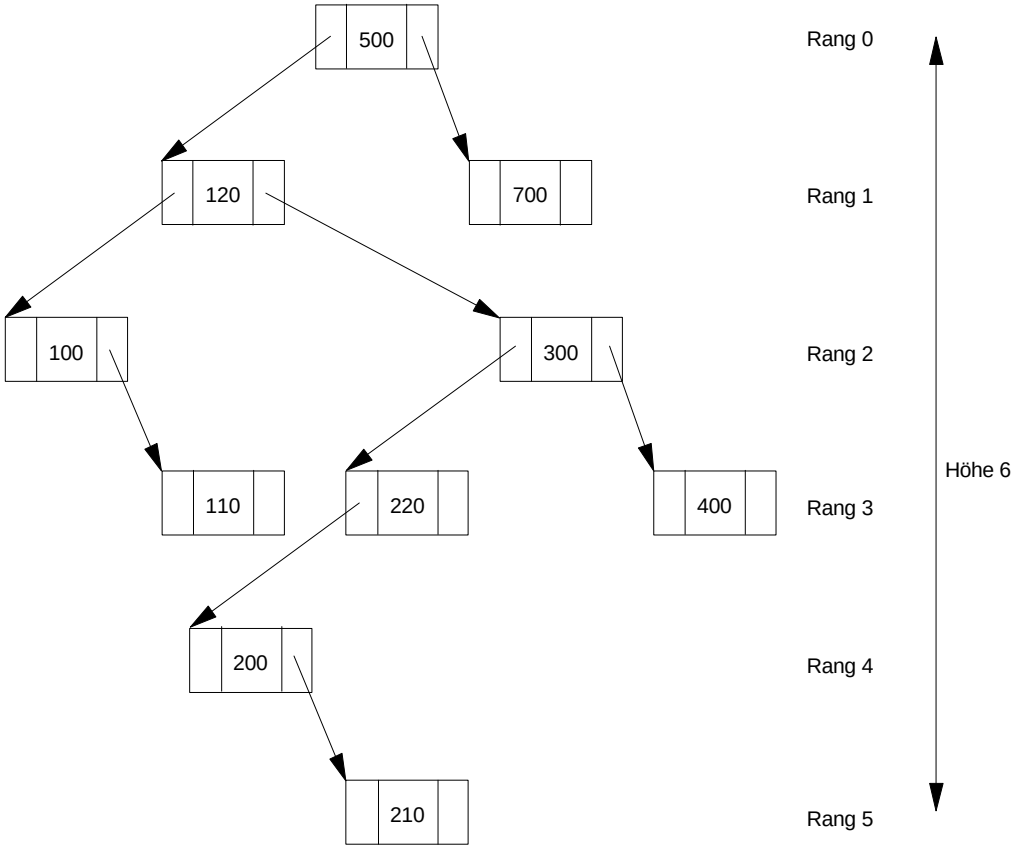
Die Kante $e = (v_i, v_j)$ läuft von v_i nach v_j (verbindet v_i mit v_j). Der Knoten v_i heißt **Anfangsknoten** der Kante $e = (v_i, v_j)$, der Knoten v_j **Endknoten** von $e = (v_i, v_j)$. Zu einem Knoten $v \in V$ heißt $pred(v) = \{v' | (v', v) \in E\}$ die **Menge der direkten Vorgänger** von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

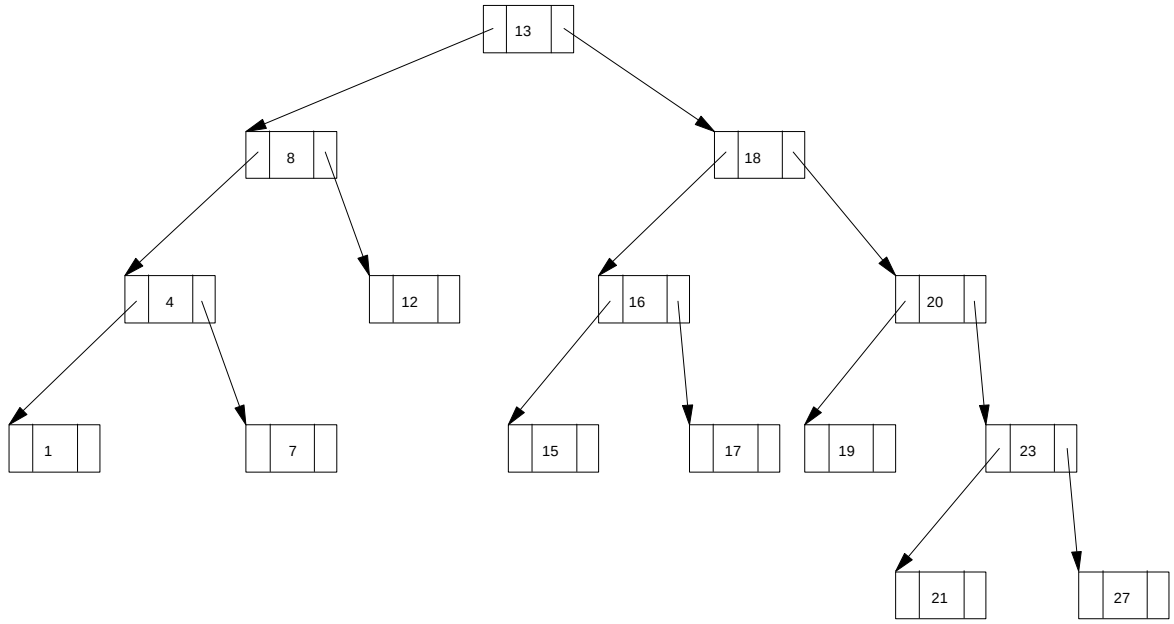
Ein **Binärbaum** B_n = (*V*, *E*) mit *n* Knoten wird durch folgende Eigenschaften charakterisiert:

1. Entweder ist $n \geq 1$ und $|V| = n \geq 1$ und $|E| = n - 1$,
   oder es ist $n = 0$ und $V = E = \varnothing$ (**leerer Baum**).

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

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





Der Rang eines Knotens läßt sich auch folgendermaßen definieren:

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

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

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

Satz 1.1-9:

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

1.2 Problemklassen

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

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

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

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

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

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

Problem des Handlungsreisenden als Optimierungsproblem
(minimum traveling salesperson problem)

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

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

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

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

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

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

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

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

Problem des Handlungsreisenden auf Graphen als Optimierungsproblem

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

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

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

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

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

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

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

Das Problem des Handlungsreisenden findet vielerlei Anwendungen in den Bereichen

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

Problem des Handlungsreisenden auf Graphen als Berechnungsproblem

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

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

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

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

Problem des Handlungsreisenden auf Graphen als Entscheidungsproblem

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

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

Lösung: Die Antwort auf die Frage:

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

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

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

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

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

Es werden folgende **Problemtypen** unterschieden:

Entscheidungsproblem Π:

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

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

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

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

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

Berechnungsproblem Π:

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

Lösung: Berechnung des Werts $f(x)$.

Optimierungsproblem Π:

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

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

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

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

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

4. $goal = \min$

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

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

1.3 Ein intuitiver Algorithmusbegriff

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

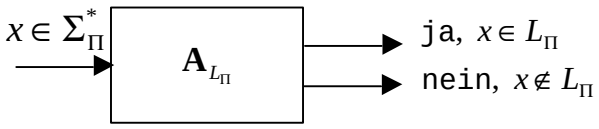
  Die Beantwortung dieser Frage liefert untere Komplexitätsschranken.

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

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

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

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



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

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

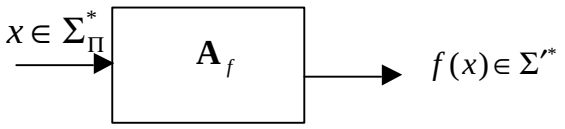
Wie bereits in Kapitel 1.2 bemerkt, erweist sich der definierte Begriff des Entscheidungsalgorithmus als zu eng gefaßt, wenn man fordert, daß der Algorithmus bei jeder Eingabe $x \in \Sigma_\Pi^*$ stoppt (entweder mit Antwort `ja` oder `nein`). Es werden daher auch Entscheidungsprobleme algorithmisch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

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

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

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

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



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

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

Satz 1.3-1:
Es gibt nichtberechenbare Funktionen der Form $g : \mathbf{N} \to \{0,1\}$.

Beweis:

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

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

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

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

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

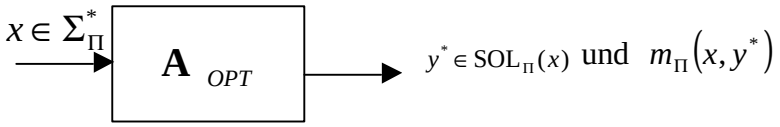
Daher ist $g_0 \notin B$ und damit eine nichtberechenbare Funktion der Form $g : \mathbf{N} \rightarrow \{0,1\}$. ///

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

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

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

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



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

2 Modelle der Berechenbarkeit

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

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

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

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

2.1 Deterministische Turingmaschinen

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

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

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

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

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

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

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

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