Formale Grundlagen der Informatik 2
(Theoretische Informatik)

(Skriptum WS 2002/2003)

Franz Winkler

Institut für Symbolisches Rechnen
(RISC-Linz)
Johannes Kepler Universität
Linz

1. Algorithmen

1.1. Was ist ein Algorithmus?

Der Begriff des Algorithmus ist in der Mathematik von alters her gebräuchlich. Intuitiv verstehen wir darunter ein Verfahren, welches von einer Anzahl von Grunddaten (Eingabe des Algorithmus) ausgeht und nach einer endlichen Anzahl wohldefinierter, effektiver Schritte eine Anzahl von Resultaten (Ausgabe des Algorithmus) liefert. Als Urahn aller Algorithmen wird vielfach der *Euklid'sche Algorithmus* zur Berechnung des größten gemeinsamen Teilers ganzer Zahlen angesehen. Man findet ihn bereits im 7. Buch der *"Elemente"* von Euklid ($\sim$ 300 v.Chr.), viele Gelehrte vermuten jedoch, daß es sich dabei eigentlich um Euklid's Fassung eines auf Eudoxus ($\sim$ 375 v.Chr.) zurückgehenden Algorithmus handelt.

Euklid'scher Algorithmus (moderne Fassung):
Zu zwei gegebenen nichtnegativen ganzen Zahlen m und n findet der Algorithmus ihren größten gemeinsamen Teiler (ggt).

(1) $[n = 0?]$ Ist $n = 0$, so endet der Algorithmus mit m als Resultat.

(2) $[m \pmod n]$ Setze $l \leftarrow m \pmod n$, $m \leftarrow n$, $n \leftarrow l$ und fahre mit Schritt (1) fort. ■

Die Korrektheit des Algorithmus sieht man unmittelbar aus der Beziehung

$$\mathrm{ggt}(m, n) = \mathrm{ggt}(n, m - q \cdot n),$$

wobei q eine beliebige ganze Zahl ist.

Als Beispiel dafür berechnen wir etwa den größten gemeinsamen Teiler der beiden Zahlen 40902 und 24140:

$$\begin{aligned}
\mathrm{ggt}(40902, 24140) &= \mathrm{ggt}(24140, 16762) = \mathrm{ggt}(16762, 7378) \\
&= \mathrm{ggt}(7378, 2006) = \mathrm{ggt}(2006, 1360) \\
&= \mathrm{ggt}(1360, 646) = \mathrm{ggt}(646, 68) \\
&= \mathrm{ggt}(68, 34) = \mathrm{ggt}(34, 0) \\
&= 34.
\end{aligned}$$

Im Folgenden wollen wir einige Aspekte des *informalen* Begriffs *"Algorithmus"* und *"durch einen Algorithmus berechenbare Funktion"* zusammenstellen. Oft beschränken wir uns auf Algorithmen, welche als Eingabe ein k–Tupel natürlicher Zahlen nehmen (für ein festes k) und als Ausgabe eine natürliche Zahl liefern. Ein Algorithmus berechnet also eine zahlentheoretische Funktion. Praktisch ist das keine Einschränkung, da alle interessanten und vernünftigen Ein– und Ausgabemengen in den natürlichen Zahlen kodiert werden können.

Es ist wichtig zu unterscheiden zwischen dem Begriff des *Algorithmus*, d.h. der Rechenvorschrift, Prozedur, und dem Begriff der *durch einen Algorithmus berechenbaren Funktion*, d.h. der durch eine Rechenvorschrift gegebenen Abbildung. Ein und dieselbe Funktion kann durch viele verschiedene Algorithmen gegeben sein.

Einige Beispiele für Funktionen, für die ein Algorithmus bekannt ist:

a. $\lambda x, y.x + y$.

b. $\lambda x.(x-$te Primzahl). Ein Algorithmus ist zum Beispiel das *Sieb des Eratosthenes*.

c. $\lambda x, y.\mathrm{ggt}(x, y)$. Dazu kann man den *Euklidschen Algorithmus* benutzen.

d. $\lambda x.$(die natürliche Zahl ≤ 9, die als x–te Ziffer in der Dezimaldarstellung von $\pi = 3.14159\ldots$ vorkommt). Etwa durch Quadratur des Kreises mittels der *Simpson Regel*.

Def. 1.1.1: (informal) Wir führen einige Eigenschaften an, welche man üblicherweise mit dem informalen Begriff des *Algorithmus* verbindet. Davon sind die Eigenschaften (1) — (5) praktisch von allen Mathematikern und Computerwissenschaftern als notwendig anerkannt, über (6) — (10) gibt es gelegentlich verschiedene Auffassungen.

(1) Ein Algorithmus besteht aus einer endlichen Anzahl von Anweisungen oder Instruktionen.

(2) Es gibt einen *Rechner*, Mensch oder Maschine, der diese Instruktionen ausführen kann.

(3) Es besteht die Möglichkeit, Rechenschritte abzuspeichern und aus dem Speicher wieder abzurufen.

(4) Sei P eine Menge von Instruktionen wie in (1) und R ein Rechner wie in (2). Dann besteht die Abarbeitung von P durch R in einem diskreten, schrittweisen

Rechenvorgang (keine analogen Methoden).

(5) Die Abarbeitung von P durch R ist deterministisch, ohne Bezugnahme auf Zufallsmethoden, z.B. Würfeln.

(6) Die Länge der Eingabe ist nach oben hin nicht beschränkt. (Die Theorie der Algorithmen beschäftigt sich mit der Frage der prinzipiellen Berechenbarkeit, ohne Bezugnahme auf praktische Einschränkungen.)

(7) Die Anzahl der Instruktionen ist nach oben hin nicht beschränkt. (Siehe (6).)

(8) Der verfügbare Speicherplatz ist nach oben hin nicht beschränkt. (Siehe (6).)

(9) Der Rechner R kann nur eine endliche Anzahl von Typen von Instruktionen verarbeiten.

(10) Eine a priori Schranke für die Anzahl der benötigten Rechenschritte muß nicht vorhanden sein. ■

Dieses intuitive Konzept eines Algorithmus bezieht seine Legitimität aus der vielfach geteilten Überzeugung, daß es möglich ist zu erkennen, ob ein Verfahren effektiv ist oder nicht. Das mag noch angehen, wenn es sich darum handelt von einem vorgelegten Verfahren zu entscheiden, ob es ein Algorithmus ist oder nicht. Die Intuition ist aber von geringem Nutzen, wenn es darum geht von einem gewissen Problem nachzuweisen, daß es keinen wie immer gearteten Algorithmus zu dessen Lösung gibt. So gibt es z.B. keinen Algorithmus, der für zwei beliebig vorgegebene Pascal–Programme entscheiden könnte, ob sie das gleiche Input–Output Verhalten haben! Probleme dieser Art treten sehr häufig in der theoretischen Informatik auf, und haben in den letzten Jahrzehnten besondere Aufmerksamkeit erlangt. Eine wissenschaftlich einwandfreie Definition des Begriffs "Algorithmus" ist also in höchstem Maße wünschenswert.

Ein wesentlicher Anstoß zur exakten Fassung des Begriffs "Algorithmus" kam von David Hilberts Programm, das das Ziel verfolgte, einen Algorithmus zu finden, der es gestatten würde, von jeder logischen Formel über den ganzen Zahlen, etwa

$$\neg\Big(\exists n, x, y, z\Big)\Big(\quad x^n + y^n = z^n,\ x \cdot y \cdot z \neq 0,\ n > 2\Big)$$

(Fermats Letzter Satz [1]) zu entscheiden, ob sie gültig ist oder nicht. Daß dieses

[1] Die Gültigkeit von Fermats Letztem Satz wurde in jüngster Zeit von Andrew Wiles bewiesen. Zuvor war diese Frage mehr als 300 Jahre lang ein offenes Problem.

Vorhaben zum Scheitern verurteilt war, wurde 1931 von Kurt Gödel [2] in seinem Unvollständigkeitssatz nachgewiesen:

In jeder Axiomatisierung der Theorie der ganzen Zahlen gibt es Formeln, deren Gültigkeit sich im System weder beweisen noch widerlegen läßt.

Um 1930 arbeiteten viele Mathematiker an diesem Hilbertschen Projekt, unter ihnen A.M. Turing, E.L. Post, A. Church. Verschiedene Modelle für das intuitive Konzept des Algorithmus wurden entwickelt: Turing–Maschinen von A.M. Turing [3], ein ähnliches Modell von E.L. Post, der λ–Kalkül von A. Church. Es stellte sich jedoch heraus, daß alle diese Modelle dieselbe Berechnungspotenz besitzen.

Im folgenden wollen wir verschiedene Annäherungen an den Begriff der Berechenbarkeit etwas näher betrachten.

[2] K. Gödel: *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*, *Monatshefte f. Math. u. Physik*, 38, 171–198 (1931).

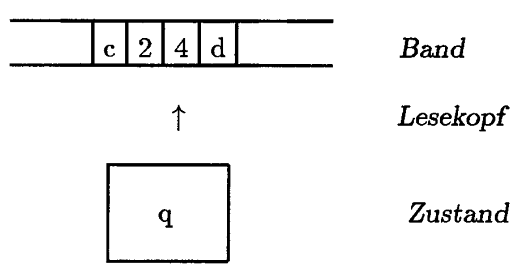
[3] A.M. Turing: *On computable numbers with an application to the Entscheidungsproblem*, *Proc. London Math. Soc.*, 2/42, 230–265 (1936). Korrektur, ibid., 43, 544–546.

1.2. Endliche Automaten und reguläre Sprachen

Deterministische endliche Automaten

Beispiel 1.2.1: Wir wollen einen Automaten konstruieren, welcher Identifiers in einer Programmiersprache erkennt. Ein Identifier ist dabei eine Folge von Buchstaben und Ziffern, welche mit einem Buchstaben beginnt.

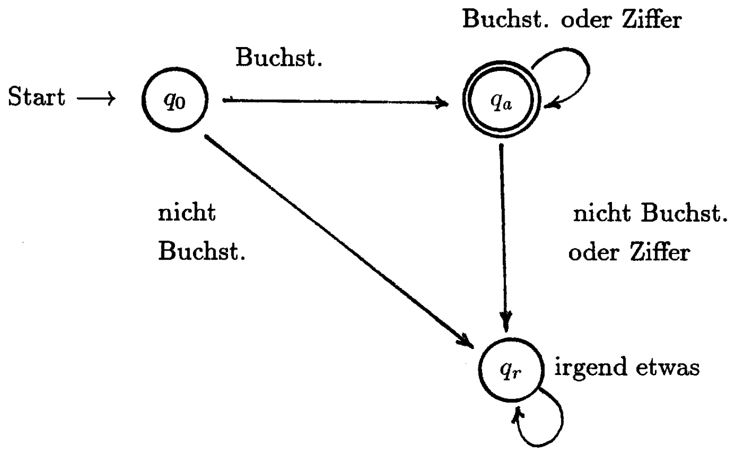
A:



Der Automat A beginnt im Zustand q_0 am Beginn der zu lesenden Folge von Symbolen. In jedem Verarbeitungsschritt liest A das Bandsymbol, auf welches der Lesekopf gerade zeigt, nimmt einen neuen Zustand an und bewegt den Lesekopf um eine Position nach rechts. Liest A im Zustand q_0 einen Buchstaben, so wird der Zustand q_a angenommen, ansonsten der Zustand q_r. Wird im Zustand q_a ein Buchstabe oder eine Ziffer gelesen, so verbleibt A im Zustand q_a. Wird im Zustand q_a etwas anderes als ein Buchstabe oder eine Ziffer gelesen, so wird der Zustand q_r angenommen. Befindet sich A im Zustand q_r, so wird dieser beibehalten, unabhängig vom gelesenen Symbol.

q_a is der sogenannte *akzeptierende* (oder *End*–) Zustand.

Der Überführungsgraph von A sieht wie folgt aus:

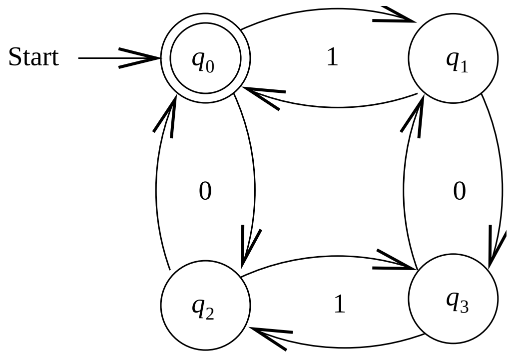


■

Beispiel 1.2.2: Ein Automat, der Folgen von 0 und 1 akzeptiert, in denen die Anzahl der 0en und 1en beide gerade sind.

Möglichkeiten:

#0	#1	
	gerade	ungerade
gerade	q_0	q_1
ungerade	q_2	q_3



■

Wir wollen nun eine formale Definition für einen endlichen Automaten geben.

Def. 1.2.1: Ein *(deterministischer) endlicher Automat (DEA)* ist ein 5–tupel $M = (Q, \Sigma, \delta, q_0, F)$. Dabei ist

Q eine endliche Menge (von *Zuständen*),

Σ eine endliche Menge (*Eingabealphabet*),

q_0 ein Element von Q (*Startzustand*),

$F \subseteq Q$ (Menge der *Endzustände*),

δ eine Funktion von $Q \times \Sigma$ nach Q (*Überführungsfunktion*).

Im folgenden sprechen wir meist einfach von einem "endlichen Automaten", wenn wir einen DEA meinen.

Die Überführungsfunktion wird folgendermaßen auf Strings beliebiger Länge über Σ (also Elemente von Σ^*) ausgedehnt. Wir bezeichnen die erweiterte Funktion ebenfalls mit δ.

$$\delta(q, \epsilon) = q, \quad (\epsilon \text{ ist der leere String}) \qquad \text{für beliebiges } q \in Q,$$
$$\delta(q, x\alpha) = \delta(\delta(q, x), \alpha), \qquad \text{für bel. } q \in Q, \alpha \in \Sigma, x \in \Sigma^*.$$

Der String $x \in \Sigma^*$ wird von M *akzeptiert* genau dann wenn $\delta(q_0, x) \in F$.

Die von M *akzeptierte Sprache* (Teilmenge von Σ^*) ist definiert als

$$L(M) := \{x \in \Sigma^* \mid x \text{ wird von } M \text{ akzeptiert }\}.$$

Wir sagen auch M *akzeptiert* die Sprache $L(M)$. ∎

Reguläre Sprachen

Im folgenden brauchen wir einige Notationen. Sei Σ ein Alphabet, also eine endliche Menge von Symbolen, und seien $L, L_1, L_2 \subseteq \Sigma^*$, also *Sprachen über* Σ.

$\overline{L} := \Sigma^* \setminus L$ (Komplement)
$L_1 \circ L_2 := \{xy \mid x \in L_1, y \in L_2\}$ (Verkettung)
$L^0 := \{\epsilon\}$
$L^i := L \circ L^{i-1}$ für $i \geq 1$
$L^* := \bigcup_{i=0}^{\infty} L^i$ (Kleene Abschluß)
$L^+ := \bigcup_{i=1}^{\infty} L^i$ (positiver Abschluß)

Def. 1.2.2: Sei Σ ein Alphabet. Ein *regulärer Ausdruck* über Σ ist entweder

$\emptyset, \epsilon, \alpha$ (für jedes Symbol $\alpha \in \Sigma$), oder

$(r + s)$, $(r \cdot s)$, (r^*), wobei r und s reguläre Ausdrücke sind.

Die Funktion L transformiert reguläre Ausdrücke über Σ in Untermengen von Σ^*:

$L(\emptyset) := \emptyset,\ L(\epsilon) := \{\epsilon\},\ L(\alpha) := \{\alpha\}$ (für jedes $\alpha \in \Sigma$),

$L(r + s) := L(r) \cup L(s)$,

$L(r \cdot s) := L(r) \circ L(s)$,

$L(r^*) := L(r)^*$.

Ist r ein regulärer Ausdruck, so heißt $L(r)$ die durch r bezeichnete *reguläre Menge* oder *reguläre Sprache*. ∎

Streng genommen müßten wir in obiger Definition natürlich $L((r + s))$ schreiben anstatt $L(r + s)$, ebenso für $r \cdot s$ und r^*. Wir gestatten uns aber die abkürzende

Schreibweise, wenn kein Mißverständnis möglich ist. Dabei vereinbaren wir folgende Präzedenzregel: * vor $\cdot$ vor $+$.

Beispiel 1.2.3: Sprache der Identifiers L_I: $\Sigma = \{a, \ldots, z, 0, \ldots, 9\}$,

$$\begin{aligned} L_I =& (\{a\} \cup \cdots \cup \{z\}) \ \circ \ \Sigma^* = \\ & L(\ (a + \cdots + z) \ \cdot \ (a + \cdots + z + 0 + \cdots + 9)^* \ ). \end{aligned}$$

■

Endliche Automaten und reguläre Ausdrücke erzeugen dieselben Sprachen.

Satz 1.2.1: *Sei Σ ein Alphabet, $L \subseteq \Sigma^*$.*

(i) L ist eine reguläre Sprache

genau dann wenn

(ii) L ist die von einem endlichen Automaten akzeptierte Sprache.

Beweis: (i) $\Longrightarrow$ (ii): Sie t ein regulärer Ausdruck über Σ, der die Sprache L bezeichnet.

(a) Zunächst konstruieren wir einen neuen regulären Ausdruck t' über einem neuen Alphabet Σ', so, daß jedes Vorkommen eines $x \in \Sigma$ in t' durch einen eigenen Index gekennzeichnet ist.

Zum Beispiel: $t = a \cdot a + b^* \cdot a \cdot b^*$,
$t' = a_1 \cdot a_2 + b_1^* \cdot a_3 \cdot b_2^*$.

Sei nun $L' := L(t')$.

(b) Ausgehend von t' konstruieren wir nun folgende Mengen:
B' .... mögliche Anfangssymbole von Strings in L',
E' .... mögliche Endsymbole von Strings in L',
S' .... mögliche Paare aufeinanderfolgender Symbole in Strings in L'.

Zum Beispiel: $B' = \{a_1, b_1, a_3\}$,
$E' = \{a_2, a_3, b_2\}$,
$S' = \{(a_1, a_2), (b_1, b_1), (b_1, a_3), (a_3, b_2), (b_2, b_2)\}$.

(c) Ausgehend von B', E', S' konstruieren wir nun einen endlichen Automaten M, sodaß $L = L(M)$, auf folgende Weise:

$Q_0 := \{q_0\}$; (für irgendein Zustandssymbol q_0)
$\delta(q_0, \alpha) := \{\alpha_i \mid \alpha_i \in B'\}$ für alle $\alpha \in \Sigma$;
$k := 0$;
$Q_1 := Q_0 \cup \{\delta(q_0, \alpha) \mid \alpha \in \Sigma\}$;
while $Q_{k+1} \neq Q_k$ **do**
  $\{k := k + 1$;
  $\delta(q, \alpha) := \{\alpha_i \mid$ es gibt ein $\beta_j \in q$ mit $(\beta_j, \alpha_i) \in S'\}$
    für alle $q \in Q_k \setminus \{q_0\}$ und $\alpha \in \Sigma$;
  $Q_{k+1} := Q_k \cup \{\delta(q, \alpha) \mid q \in Q_k, \alpha \in \Sigma\}$ };
$Q := Q_k$

Diese Prozedur terminiert offensichtlich, da es nur endlich viele Symbole in Σ' gibt.

Es ist klar, daß $x \in L$ genau dann wenn M, ausgehend von q_0 und dem Eingabewort x, einen Zustand erreicht, der ein Element von E' enthält. Wenn wir also setzen

$$q \in F :\Longleftrightarrow q \cap E' \neq \emptyset \quad \text{für } q \in Q \setminus \{q_0\}, \text{ und}$$
$$q_0 \in F :\Longleftrightarrow \epsilon \in L,$$

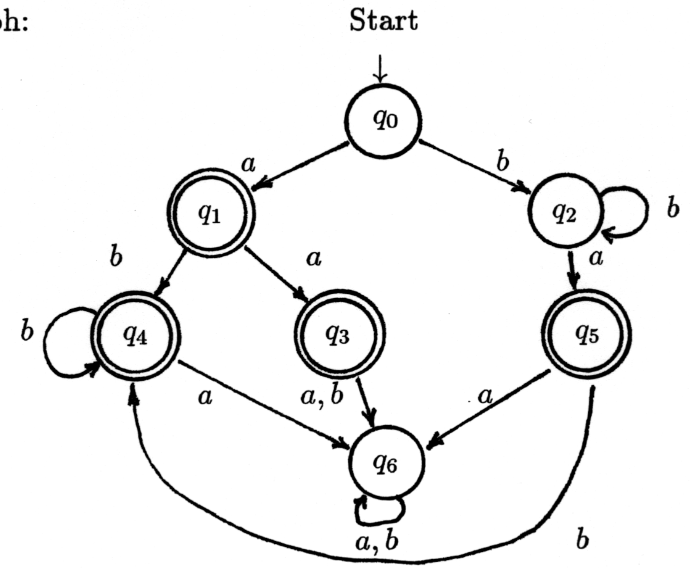
dann sind wir fertig.

Zum Beispiel: $Q_0 = \{q_0\}$,

$$\begin{aligned}
&\delta(q_0, a) = \{a_1, a_3\} =: q_1,\\
&\delta(q_0, b) = \{b_1\} =: q_2,\\
Q_1 &= \{q_0, q_1, q_2\},\\
&\delta(q_1, a) = \{a_2\} =: q_3,\\
&\delta(q_1, b) = \{b_2\} =: q_4,\\
&\delta(q_2, a) = \{a_3\} =: q_5,\\
&\delta(q_2, b) = \{b_1\} =: q_2,\\
Q_2 &= \{q_0, \ldots, q_5\},\\
&\delta(q_3, a) = \emptyset =: q_6,\\
&\delta(q_3, b) = \delta(q_4, a) = \delta(q_5, a) = \emptyset = q_6,\\
&\delta(q_4, b) = \delta(q_5, b) = \{b_2\} = q_4,\\
Q_3 &= \{q_0, \ldots, q_6\},\\
&\delta(q_6, a) = \delta(q_6, b) = \emptyset = q_6,\\
Q_4 &= Q_3.
\end{aligned}$$

Also $Q = Q_3 = \{q_0, \ldots, q_6\}$, und $F = \{q_1, a_3, q_4, q_5\}$.

Überführungsgraph:



(ii) $\Longrightarrow$ (i): Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein endlicher Automat mit $L = L(M)$. Für beliebige Zustände $q, p \in Q$ betrachten wir die Mengen

$$R_{q,p} := \{x \in \Sigma^* \mid \delta(q, x) = p\}.$$

Es gilt

$$L = \bigcup_{p \in F} R_{q_0,p}. \tag{1}$$

Nun sei $Q = \{q_0, q_1, \ldots, q_r\}$. Für beliebige $q, p \in Q$ und $1 \leq j \leq r+1$ definieren wir:

$$R^j_{q,p} := \{x \in R_{q,p} \mid \delta(q, y) \in \{q_0, \ldots, q_{j-1}\} \text{ für jeden Prefix } y \text{ von } x\} \tag{2}$$

und

$$R^0_{q,p} := \begin{cases} \{\alpha \in \Sigma \mid \delta(q, \alpha) = p\}, & \text{falls } q \neq p, \\ \{\alpha \in \Sigma \mid \delta(q, \alpha) = p\} \cup \{\epsilon\}, & \text{falls } q = p. \end{cases} \tag{3}$$

Dann gilt

$$R_{q,p} = R^{r+1}_{q,p}. \tag{4}$$

Für alle q, p ist $R^0_{q,p}$ eine reguläre Menge, da es die Vereinigung endlich vieler Mengen $\{a\}, a \in \Sigma$, oder $\{\epsilon\}$ ist. Weiters gilt

$$R^{j+1}_{q,p} = R^j_{q,p} \ \cup \ R^j_{q,q_j} \circ (R^j_{q_j,q_j})^* \circ R^j_{q_j,p}. \tag{5}$$

Aus (5) sehen wir, daß $R^j_{q,p}$ $(1 \leq j \leq r+1)$ reguläre Mengen sind. Somit ist wegen (4) und (1) auch L regulär. Tatsächlich haben wir sogar einen regulären Ausdruck konstruiert, der L bezeichnet. ■

Beispiel 1.2.4: Wir betrachten den im ersten Teil des Satzes konstruierten Automaten. Für diesen gilt

$$L = R_{q_0,q_1} \cup R_{q_0,q_3} \cup R_{q_0,q_4} \cup R_{q_0,q_5}.$$

$$\begin{aligned} R_{q_0,q_1} = & R^7_{q_0,q_1} \\ & R^0_{q_0,q_1} = \{a\}, \\ & R^1_{q_0,q_1} = \{a\} \ \cup \ \underbrace{R^0_{q_0,q_1}}_{\{a\}} \circ \underbrace{(R^0_{q_1,q_1})^*}_{\{\epsilon\}^*} \circ \underbrace{R^0_{q_1,q_1}}_{\{\epsilon\}} = \{a\}, \\ & R^2_{q_0,q_1} = \cdots = R^7_{q_0,q_1} = \{a\} \end{aligned}$$

etc. ■

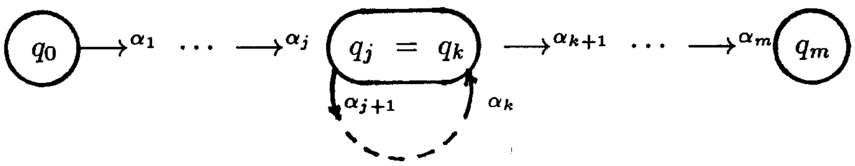
Der folgende Satz eignet sich oft sehr gut dazu zu entscheiden, ob eine vorliegende Sprache regulär ist oder nicht. Vereinfacht gesagt ist eine Sprache nur dann regulär, wenn genügend lange Wörter in ihr aufgebläht werden können ohne die Sprache zu verlassen. Daher kommt auch der Name für diesen Satz.

Satz 1.2.2: (Pumping Lemma) *Sei L eine reguläre Sprache. Dann gibt es eine Konstante $n (\in \mathbb{N})$, sodaß jedes Wort $z \in L$ der Länge mindestens n ($\mid z \mid \geq n$) auf solche Weise zerlegt werden kann in $z = uvw$, daß $\mid uv \mid \leq n$, $\mid v \mid \geq 1$, und für alle $i \geq 0$ auch $uv^iw \in L$. Die Konstante n ist dabei nicht größer als die Anzahl der Zustände des kleinsten Automaten, der L akzeptiert.*

Beweis: Sei $M = (Q, \Sigma, \delta, q_0, F)$ ein endlicher Automat, der L akzeptiert. Sei $n = \mid Q \mid$. Wir betrachten nun ein Wort z, sodaß

$$z = \alpha_1 \cdots \alpha_m \in L \quad \text{und} \quad \mid z \mid = m \geq n.$$

Wenn wir mit q_i denjenigen Zustand bezeichnen, welcher nach Abarbeitung von $\alpha_1 \cdots \alpha_i$ angenommen wird, so müssen offensichtlich zwei solche Zustände identisch sein. Also für gewisse j, k ($0 \leq j < k \leq n$),



z kann also zerlegt werden in $z = uvw$, wobei

$$u = \alpha_1 \cdots \alpha_j, \quad v = \alpha_{j+1} \cdots \alpha_k, \quad w = \alpha_{k+1} \cdots \alpha_m.$$ ■

Beispiel 1.2.5: $L = \{a^m b^m \mid m \in \mathbb{N}\}$ ist keine reguläre Sprache.

Angenommen L wäre regulär. Sei n die Konstante im Pumping Lemma. Wir betrachten das Wort

$$z = a^n b^n,$$

das natürlich in L enthalten ist. Wegen des Pumping Lemmas muß z zerlegbar sein in

$$a^n b^n = uvw,$$

wobei $|uv| \leq n$, $|v| \geq 1$, und $uv^i w \in L$ für jedes i. Somit müßte gelten

$$\underbrace{a^{n_1}}_{u}\ \underbrace{a^{n_2}}_{v}\ \underbrace{a^{n_3} b^n}_{w}, \qquad \text{wobei } n = n_1 + n_2 + n_3,\ n_2 \geq 1.$$

Aber $a^{n_1}(a^{n_2})^2 a^{n_3} b^n$ ist nicht in L. Somit kann L nicht regulär sein. ■

Beispiel 1.2.6: $L = \{0^{i^2} \mid i \in \mathbb{N}\}$ ist keine reguläre Sprache.

Angenommen L wäre regulär. Sei n die Konstante im Pumping Lemma. Wir betrachten das Wort

$$z = 0^{n^2},$$

das natürlich in L enthalten ist. Wegen des Pumping Lemmas muß z zerlegbar sein in

$$0^{n^2} = uvw,$$

wobei $|uv| \leq n$, $|v| \geq 1$, und $uv^i w \in L$ für jedes i. Das ist aber für $i = 2$, also $uv^2 w$, schon nicht der Fall. $n^2 < |uv^2 w| \leq n^2 + n < (n+1)^2$. Somit ist $uv^2 w \notin L$. ■

<u>Nichtdeterministische endliche Automaten</u>

Man kann sich nun aber auch endliche Automaten vorstellen, die bei gegebenem Zustand und Eingabesymbol nicht nun einen einzig möglichen Nachfolgezustand haben, sondern aus verschiedenen (oder auch gar keinen) Möglichkeiten für Nachfolgerzustände wählen können.

Wie soll die Semantik eines solchen "nichtdeterministischen endlichen Automaten" aussehen? Wenn im Zustand p der Automat das Symbol a liest und etwa $\{q_1, q_2\}$ für diesen Fall als mögliche Nachfolgezustände definiert sind, so kann der Automat entweder in den Zustand q_1 oder in den Zustand q_2 übergehen. Der Automat akzeptiert ein Wort $w \in \Sigma^*$, wenn es einen möglichen Berechnungspfad gibt, sodaß der Automat von einem Anfangszustand und Wort w auf dem Band zu einem Endzustand gelangen kann.

Def. 1.2.3: Ein *nichtdeterministischer endlicher Automat (NEA)* ist ein 5–tupel $M = (Q, \Sigma, \delta, S, F)$. Dabei ist

Q eine endliche Menge (von *Zuständen*),

Σ eine endliche Menge (*Eingabealphabet*),

$S \subseteq Q$ (Menge der *Anfangszustände*),

$F \subseteq Q$ (Menge der *Endzustände*),

δ eine Funktion von $Q \times \Sigma$ nach $\mathcal{P}(Q)$, der Potenzmenge von Q (*Überführungsfunktion*).

Die Überführungsfunktion wird folgendermaßen ausgedehnt zu einer Funktion von $Q \times \Sigma^*$ nach $\mathcal{P}(Q)$. Wir bezeichnen die erweiterte Funktion ebenfalls mit δ.

$\delta(q, \epsilon) = \{q\}$, für bel. $q \in Q$,

$\delta(q, x\alpha) = \{p \mid p \in \delta(r, \alpha) \text{ für ein } r \in \delta(q, x)\}$, für bel. $q \in Q, x \in \Sigma^*, \alpha \in \Sigma$.

Weiters dehnen wir δ aus zu einer Funktion von $\mathcal{P}(Q) \times \Sigma^*$ nach $\mathcal{P}(Q)$:

$\delta(P, x) = \bigcup_{p \in P} \delta(p, x)$, für bel. $P \subseteq Q$, $x \in \Sigma^*$.

Der String $x \in \Sigma^*$ wird von M *akzeptiert* genau dann wenn $\delta(S, x) \cap F \neq \emptyset$.

Die von M *akzeptierte Sprache* (Teilmenge von Σ^*) ist definiert als

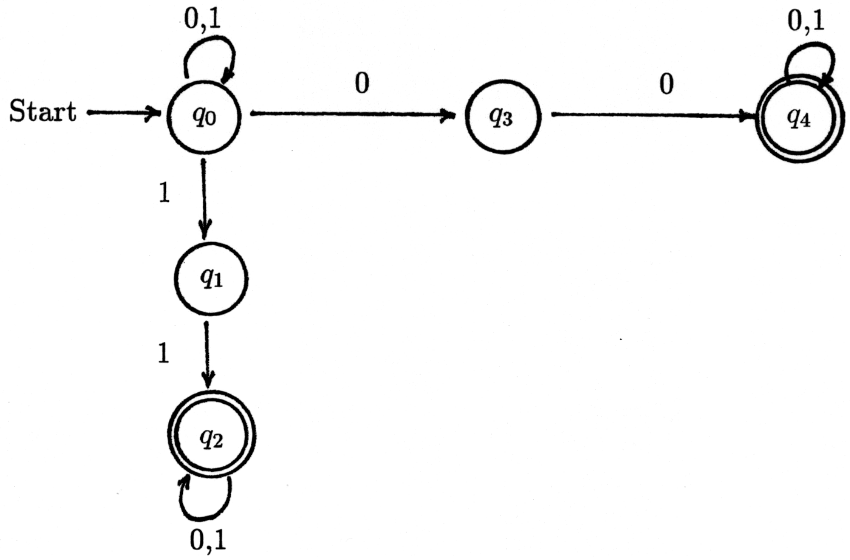
$$L(M) := \{x \in \Sigma^* \mid x \text{ wird von } M \text{ akzeptiert }\}.$$

Wir sagen auch M *akzeptiert* die Sprache $L(M)$. ■

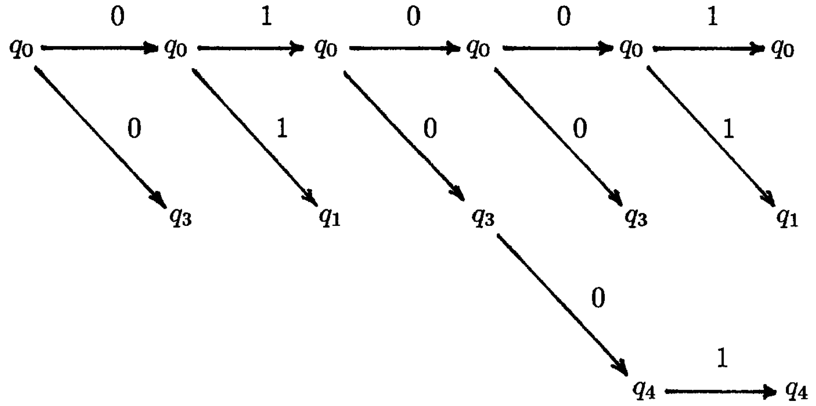
Beispiel 1.2.7: Wir betrachten folgenden NEA M, wobei $Q = \{q_0, q_1, q_2, q_3, q_4\}, \Sigma = \{0, 1\}, S = \{q_0\}, F = \{q_2, q_4\}$, und die Überführungsfunktion δ durch folgende Tabelle gegeben ist:

	0	1
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$

Der Überführungsgraph dieses NEA M sieht folgendermaßen aus:



Wenn der Automat die Möglichkeit hat, wie im Zustand q_0 bei Eingabe 0, verschiedene Nachfolgezustände anzunehmen, so können wir uns vorstellen, daß entsprechend viele Kopien des Automaten hergestellt werden, deren jede eine dieser Möglichkeiten realisiert. Diese Form der Abarbeitung eines Eingabestrings führt zu einem Berechnungsbaum. Wir stellen den Berechnungsbaum für den NEA M bei Eingabe 01001 dar:



Die von M akzeptierte Sprache $L(M)$ besteht aus all denjenigen Wörtern, welche 00 oder 11 enthalten. ∎

Es ist offensichtlich, daß jeder DEA aufgefaßt werden kann als ein NEA, bei dem eben die Menge der möglichen Nachfolgezustände jeweils eine einelementige Menge ist. Somit ist klar, daß jede Sprache L, welche von einem DEA akzeptiert wird, auch von einem NEA akzeptiert werden kann. Interessanterweise gilt auch die Umkehrung.

Satz 1.2.3: *Sei M ein NEA und sei $L = L(M)$. Dann gibt es einen DEA M' mit $L(M') = L$.*

Beweis: Sei $M = (Q, \Sigma, \delta, S, F)$ der gegebene NEA. Wie konstruieren einen DEA M', der die gleiche Sprache akzeptiert, indem wir jede Teilmenge von Q als Zustand von M' auffassen. Alles übrige ergibt sich dann fast von selbst. Wir setzen also

$$M' = (Q', \Sigma, \delta', q_0', F'),$$

wobei

$$\begin{aligned} Q' &= \mathcal{P}(Q) \qquad \text{(Potenzmenge von } Q\text{)},\\ \delta'(q', \alpha) &= \bigcup_{p \in q'} \delta(p, \alpha) \ (= \delta(q', \alpha)\,), \quad \text{für } q' \in Q',\\ q_0' &= S,\\ F' &= \{q' \in Q' \mid q' \cap F \neq \emptyset\}. \end{aligned}$$

Nun sieht man leicht, daß für ein beliebiges Wort $x = \alpha_1 \ldots \alpha_n \in \Sigma^*$ gilt:

$x \in L(M)$

$\Longleftrightarrow$

$\delta(S, x) \cap F \neq \emptyset$

$\Longleftrightarrow$

es gibt eine Folge von Teilmengen $S = Q_0, \ldots, Q_n$ von Q, bzw. Zuständen $q_0' = Q_0, \ldots, q_n' = Q_n$ von M', sodaß $\delta'(q_0', \alpha_1) = q_1', \ldots, \delta'(q_{n-1}, \alpha_n) = q_n'$, und $q_n' \cap F \neq \emptyset$

$\Longleftrightarrow$

$\delta'(q_0', x) \in F'$

$\Longleftrightarrow$

$x \in L(M')$. ■

Die Erweiterung des Begriffs des endlichen Automaten zu einem nichtdeterministischen endlichen Automaten bringt also keine Potenzsteigerung der betrachteten Klasse von Automaten. Diese Beobachtung werden wir auch später wieder machen, wenn wir uns mit Turing–Maschinen befassen.

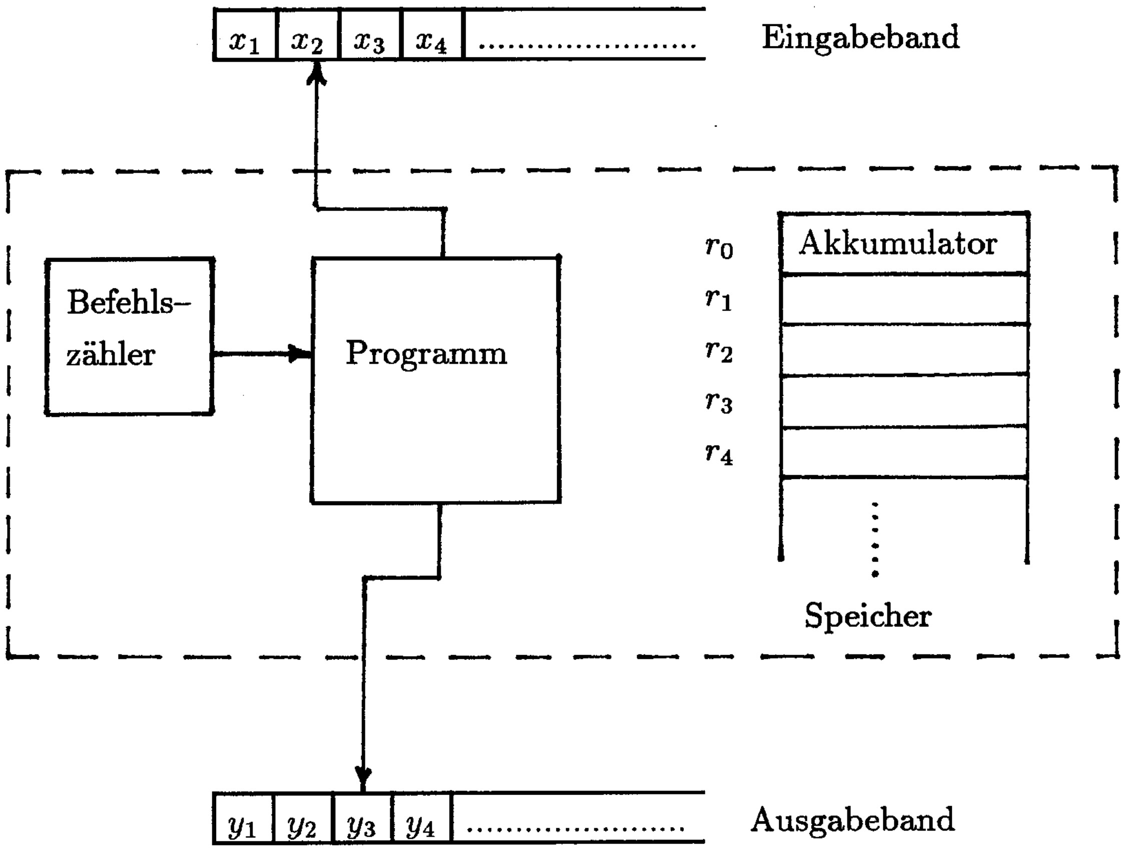
Neben den besprochenen Beschreibungsmöglichkeiten für reguläre Sprachen (von DEA akzeptiert, von NEA akzeptiert, durch einen regulären Ausdruck erzeugt) gibt es noch einige andere. So könnte man etwa auch reguläre Grammatiken betrachten. Diese führen wiederum zur selben Menge der regulären Sprachen.

Endliche Automaten kann man natürlich auch mit einer Ausgabefunktion versehen und auf diese Weise Funktionen berechnen lassen.

Wir wollen auf diese Variationen aber nicht mehr näher eingehen sondern verweisen auf die angefügte Literaturliste, insbesondere [HU] und [LP].

1.3. RAM und RASP

Definition 1.3.1: Eine *RAM (random access machine)* besteht aus einem *Eingabeband*, einem *Ausgabeband*, einem *Befehlszähler*, einem *Programm* und einem (unendlichen) *Speicher*.



Das Eingabeband ist eine unendliche Folge von *Feldern* und das Ausgabeband ist eine unendliche Folge von Feldern, von denen jedes eine ganze Zahl beinhalten kann. Der Speicher ist eine unendliche Folge von *Registern*, deren jedes eine ganze Zahl aufnehmen kann. Das Programm ist eine endliche Folge von (eventuell markierten) *Instruktionen.* Jede Instruktion besteht aus zwei Teilen, einem *Operationskode* und einer *Adresse* (die entweder ein *Operand* oder eine *Marke* ist).

Liste der RAM–Instruktionen:

Operationskode	Adresse	Operationskode	Adresse
1. `LOAD`	Operand	7. `READ`	Operand
2. `STORE`	Operand	8. `WRITE`	Operand
3. `ADD`	Operand	9. `JUMP`	Marke
4. `SUB`	Operand	10. `JGTZ`	Marke
5. `MULT`	Operand	11. `JZERO`	Marke
6. `DIV`	Operand	12. `HALT`	

Ein Operand kann wie folgt beschaffen sein:

1. `=i`, die Bedeutung ist die ganze Zahl i selbst,
2. `i`, wobei i eine nichtnegative ganze Zahl ist. Die Bedeutung ist der Inhalt des Registers i,
3. `*i`, die Bedeutung ist der Inhalt von Register j, wobei j die ganze Zahl in Register i ist (indirekte Adressierung). Ist $j < 0$, so bleibt die Maschine stehen.

Um die *Bedeutung* einer RAM R zu definieren, ziehen wir die *Speicherfunktion* $c : \mathbb{N}_0 \to \mathbb{Z}$ heran. $c(i)$ ist der Inhalt des Speicherregisters r_i.

Am Anfang der *Berechnung* ist $c(i) = 0$ für alle $i \geq 0$, der Befehlszähler ist auf 1 gesetzt, und jedes Feld des Ausgabebandes enthält Blank. Bis auf endlich viele Ausnahmen sind alle Felder des Eingabebandes leer. Der Lesekopf zeigt auf das erste Feld des Eingabebandes und der Schreibkopf zeigt auf das erste Feld des Ausgabebandes. Das Feld, auf welches der Lese– bzw. Schreibkopf gerade zeigt, heißt das *aktuelle* Eingabe– bzw. Ausgabefeld. In jedem Schritt der Berechnung wird diejenige Instruktion ausgeführt, auf die der Befehlszähler gerade zeigt, also die k–te Instruktion, wenn k der Inhalt des Befehlszählers ist. Nach Ausführung der k–ten Instruktion von R wird der Befehlszähler automatisch auf $k + 1$ gesetzt, wenn die k–te Instruktion nicht eine der Anweisungen `JUMP`, `HALT`, `JGTZ` oder `JZERO` war. Die Berechnung endet, wenn die Instruktion `HALT` abgearbeitet wird, oder zu einer Marke gesprungen wird, die im Programm nicht vorkommt.

Um die Bedeutung einer Instruktion anzugeben, definieren wir $v(a)$, den *Wert* des Operanden a, wie folgt:

$$v(=\mathtt{i}) = i, \quad v(\mathtt{i}) = c(i), \quad v(*\mathtt{i}) = c(c(i)).$$

Die folgende Tabelle gibt die Bedeutung der einzelnen Instruktionen an. Instruktionen, deren Bedeutung in der Tabelle nicht definiert ist, wie etwa `STORE =i`, haben dieselbe Bedeutung wie `HALT`. Ebenso beendet eine Division durch 0 die Berechnung.

Instruktion	Bedeutung
1. `LOAD a`	$c(0) \leftarrow v(a)$
2. `STORE i`	$c(i) \leftarrow c(0)$
`STORE *i`	$c(c(i)) \leftarrow c(0)$
3. `ADD a`	$c(0) \leftarrow c(0) + v(a)$
4. `SUB a`	$c(0) \leftarrow c(0) - v(a)$
5. `MULT a`	$c(0) \leftarrow c(0) \cdot v(a)$
6. `DIV a`	$c(0) \leftarrow \lfloor c(0)/v(a) \rfloor$
7. `READ i`	$c(i) \leftarrow$ Symbol im aktuellen Eingabefeld. Der Lesekopf wird um eine Position nach rechts gerückt.
`READ *i`	$c(c(i)) \leftarrow$ Symbol im aktuellen Eingabefeld. Der Lesekopf wird um eine Position nach rechts gerückt.

8. `WRITE a` — $v(a)$ wird in das aktuelle Ausgabefeld geschrieben. Der Schreibkopf wird um eine Position nach rechts gerückt.
9. `JUMP b` — Der Befehlszähler wird auf die Nummer der Instruktion mit der Marke b gesetzt.
10. `JGTZ b` — Falls $c(0) > 0$, dann wird der Befehlszähler auf die Nummer der Instruktion mit der Marke b gesetzt. Andernfalls wird der Befehlszähler um 1 erhöht.
11. `JZERO b` — Falls $c(0) = 0$, dann wird der Befehlszähler auf die Nummer der Instruktion mit der Marke b besetzt. Andernfalls wird der Befehlszähler um 1 erhöht.
12. `HALT` — Die Berechnung endet.

Die *Bedeutung* der RAM R ist die durch R definierte Abbildung von Eingabebändern zu Ausgabebändern. Da die Berechnung von R nicht auf allen Eingabebändern terminieren muß, ist diese Abbildung nur eine partielle Abbildung. Diese Abbildung kann nun in verschiedener Weise interpretiert werden, etwa als Funktion oder als Sprache.
Die RAM R lese immer genau n Zahlen $x_1, \ldots, x_n$ vom Eingabeband, und schreibe genau eine Zahl y auf das Ausgabeband oder terminiere nicht. Damit ist implizit die (partielle) Funktion

$$f(x_1, \ldots, x_n) = \begin{cases} y & \text{falls } R \text{ auf } x_1, \ldots, x_n \text{ terminiert und } y \text{ schreibt} \\ \perp & \text{falls } R \text{ auf } x_1, \ldots, x_n \text{ nicht terminiert} \end{cases}$$

erklärt. Wir sagen R *berechnet* die Funktion f.
Man kann R auch auffassen als Akzeptor einer Sprache *) über einem Alphabet $\{\alpha_1, \ldots, \alpha_k\}$. Das Symbol α_i kann kodiert werden als die ganze Zahl i. Ein Wort wird so auf das Eingabeband geschrieben, daß die Kodes für die einzelnen Symbole nacheinander in die Eingabefelder geschrieben werden und als Abschluß in das nächste Feld eine 0 geschrieben wird. Man sagt nun das Wort s wird von R *akzeptiert*, wenn R das Wort s und das Markierungssymbol 0 vom Eingabeband liest, eine 1 in das erste Feld des Ausgabebandes schreibt, und hält. Die *von R akzeptierte Sprache* ist die Menge aller Wörter, die von R akzeptiert werden. ■

Beispiel 1.3.1: Wir betrachten die Funktion $f(n)$, gegeben als

$$f(n) = \begin{cases} n^n & \text{falls } n \text{ positiv ist} \\ 0 & \text{sonst.} \end{cases}$$

---

*) Vgl. Sektion 1.2. Ein Alphabet Σ ist eine endliche Menge von Symbolen. Ein Wort w über Σ ist eine endliche Folge von Symbolen aus Σ. Eine Sprache L über Σ ist eine Teilmenge von Σ^*, der Menge aller Wörter über Σ.

Ein Programm zur Berechnung von $f(n)$ (in einer PASCAL–ähnlichen Sprache) könnte wie folgt aussehen:

```
begin
    read r1;
    if r1 ≤ 0 then write 0
    else
        begin
            r2 ← r1;
            r3 ← r1 − 1
            while r3 > 0 do
                begin
                    r2 ← r2 ∗ r1;
                    r3 ← r3 − 1
                end;
            write r2
        end
end
```

Ein entsprechendes RAM Programm ist

RAM Programm:			entsprechende PASCAL Statements:
	`READ`	`1`	**read** $r1$
	`LOAD`	`1`	
	`JGTZ`	`pos`	**if** $r1 \leq 0$ **then write** 0
	`WRITE`	`=0`	
	`JUMP`	`endif`	
`pos:`	`LOAD`	`1`	$r2 \leftarrow r1$
	`STORE`	`2`	
	`LOAD`	`1`	
	`SUB`	`=1`	$r3 \leftarrow r1 - 1$
	`STORE`	`3`	
`while:`	`LOAD`	`3`	
	`JGTZ`	`continue`	**while** $r3 > 0$ **do**
	`JUMP`	`endwhile`	

`continue:`	`LOAD`	`2`	
	`MULT`	`1`	$r2 \leftarrow r2 * r1$
	`STORE`	`2`	
	`LOAD`	`3`	
	`SUB`	`=1`	$r3 \leftarrow r3 - 1$
	`STORE`	`3`	
	`JUMP`	`while`	
`endwhile:`	`WRITE`	`2`	**write** $r2$
`endif:`	`HALT`		■

Beispiel 1.3.2: Wir erzeugen ein RAM Programm, welches die Sprache über dem Alphabet $\{1, 2\}$ akzeptiert, die aus allen Wörtern mit gleich vielen 1en wie 2en besteht. Das Programm liest jedes Eingabesymbol in Register 1 ein. In Register 2 wird jeweils die Differenz der Anzahl der bereits gelesenen 1en und 2en gespeichert. Sobald das Programm zum Endsymbol 0 kommt, prüft es, ob die Differenz 0 ist. Wenn ja, so wird 1 ausgedruckt und das Programm hält. Wir nehmen an, daß 0, 1 und 2 die einzigen möglichen Eingabesymbole sind.

Ein Pidgin–PASCAL Programm könnte wie folgt aussehen:

begin
  $d \leftarrow 0$;
  read x;
  while $x \neq 0$ **do**
    begin
      if $x \neq 1$ **then** $d \leftarrow d - 1$ **else** $d \leftarrow d + 1$;
      read x
    end;
  if $d = 0$ **then write** 1
end

Ein entsprechendes RAM Progamm ist

	`LOAD`	`=0`	$d \leftarrow 0$
	`STORE`	`2`	
	`READ`	`1`	**read** x
`while:`	`LOAD`	`1`	**while** $x \neq 0$ **do**
	`JZERO`	`endwhile`	
	`LOAD`	`1`	
	`SUB`	`=1`	**if** $x \neq 1$
	`JZERO`	`one`	
	`LOAD`	`2`	
	`SUB`	`=1`	**then** $d \leftarrow d - 1$
	`STORE`	`2`	
	`JUMP`	`endif`	
`one:`	`LOAD`	`2`	
	`ADD`	`=1`	**else** $d \leftarrow d + 1$

```
            STORE   2
endif:      READ    1             read x
            JUMP    while
endwhile:   LOAD    2
            JZERO   output        if d = 0 then write 1
            HALT
output:     WRITE   =1
            HALT
```

■

Das Programm einer RAM ist nicht im Speicher abgelegt, es kann sich also nicht selbst modifizieren. Wir wollen nun ein Maschinenmodell betrachten, das ganz gleich ist wie eine RAM, dessen Programm aber im Speicher abgelegt ist.

Definition 1.3.2: Die Instruktionen einer *RASP (random access stored program* machine) sind identisch mit den Instruktionen einer RAM, indirekte Adressierung ist jedoch nicht gestattet (eine RASP kann indirekte Adressierung simulieren durch Modifikation der Instruktionen während der Ausführung des Programms). Die Struktur einer RASP ist auch gleich wie die einer RAM, nur daß das Programm im Speicher abgelegt ist. Jede RASP Instruktion belegt zwei aufeinanderfolgende Register. Im ersten Register steht eine Kodierung des Operationskodes, im zweiten Register eine Adresse. Ist die Adresse von der Form `=i`, dann kodiert der Inhalt des ersten Registers auch den Umstand, daß der Operand ein Literal ist, und das zweite Register enthält i. Die Instruktionen werden als Zahlen kodiert, etwa auf folgende Weise:

Instruktion:	Kode:	Instruktion:	Kode:
`LOAD i`	1	`DIV i`	10
`LOAD =i`	2	`DIV =i`	11
`STORE i`	3	`READ i`	12
`ADD i`	4	`WRITE i`	13
`ADD =i`	5	`WRITE =i`	14
`SUB i`	6	`JUMP i`	15
`SUB =i`	7	`JGTZ i`	16
`MULT i`	8	`JZERO i`	17
`MULT =i`	9	`HALT`	18

So würde etwa die Instruktion `LOAD =32` abgespeichert als 2 in einem Register und 32 im nächsten Register.

Die Abarbeitung eines RASP Programms erfolgt im wesentlichen gleich wie die Abarbeitung eine RAM Programms, mit den folgenden geringfügigen Änderungen: zu Beginn der Berechnung ist das Programm in zusammenhängenden Speicherregistern abgelegt, und der Befehlszähler ist auf die erste Instruktion gesetzt. Die übrigen Register enthalten alle 0. Nach der Ausführung jeder Instruktion wird der Befehlszähler um 2 erhöht, außer nach einer Instruktion der Form `JUMP i`, `JGTZ i` (wenn der Inhalt des Akkumulators positiv ist) oder `JZERO i` (wenn der Inhalt des Akkumulators 0 ist). In diesen Fällen wird der Befehlszähler zu i gesetzt. ■

Jede Funktion (Sprache), welche von einem RAM Programm berechnet (akzeptiert) werden kann, kann auch von einem RASP Programm berechnet (akzeptiert) werden und umgekehrt; das wird im folgenden Satz ausgesprochen.

Satz 1.3.1: *Zu jedem RAM Programm gibt es ein äquivalentes RASP Programm und umgekehrt.*

Beweis: (a) Zunächst zeigen wir, wie ein RAM Programm P durch ein RASP Programm P' simuliert werden kann. Register 1 der RASP wird zur Zwischenspeicherung des RAM Akkumulators benutzt. Das Programm P' belegt die nächsten $r-1$ Register der RASP. Die Konstante r hängt ab vom RAM Programm P. Der Inhalt des RAM Registers i, $i \geq 1$, wird im RASP Register $r+i$ abgespeichert, sodaß alle Speicherzugriffe in P' um r gegenüber P verschoben sind.
Jede RAM Instruktion, die keine indirekte Adressierung benutzt, wird in die identische RASP Instruktion (mit entsprechend verschobenen Speicherzugriffen) übersetzt. Jede RAM Instruktion mit indirekter Adressierung wird übersetzt in eine Folge von sechs RASP Instruktionen; dabei wird die indirekte Adressierung durch Instruktionsmodifikation simuliert. Als Beispiel dafür betrachten wir die RAM Instruktion `SUB *i`, $i \geq 1$. Diese RAM Instruktion wird in eine Folge von RASP Instruktionen übersetzt, die

1. den Inhalt des Akkumulators in Register 1 zwischenspeichert,
2. den Inhalt von Register $r+i$ in den Akkumulator lädt (Register $r+i$ der RASP entspricht dem Register i der RAM),
3. r zum Akkumulator addiert,
4. den Inhalt des Akkumulators in das Adressenfeld einer `SUB` Instruktion abspeichert,
5. den Inhalt von Register 1 wieder in den Akkumulator zurücklädt, und schließlich
6. die in Schritt 4 erzeugte `SUB` Instruktion benutzt, um die Subtraktion auszuführen. So wird, unter der Annahme, daß die Folge der RASP Instruktionen in Register 100 beginnt, die RAM Instruktion `SUB *i` wie folgt simuliert:

Register	Inhalt	Bedeutung
100	3	`STORE 1`
101	1	
102	1	`LOAD` $r+i$
103	$r+i$	
104	5	`ADD =`r
105	r	
106	3	`STORE 111`
107	111	
108	1	`LOAD 1`
109	1	
110	6	`SUB` b, wobei b der Inhalt des RAM
111	–	Registers i ist.

Enthält P n Instruktionen mit m indirekten Adressierungen, so enthält P' $(n-m)+6m$ Instruktionen. Der Verschiebungsfaktor r ist also $2n+10m+1$.

(b) Zu einem gegebenen RASP Programm P' konstruieren wir ein äquivalentes RAM Programm P. Einige RAM Register werden für spezielle Zwecke benutzt:

Register 1:	für indirekte Adressierung,
Register 2:	der Befehlszähler der RASP,
Register 3:	der Akkumulator der RASP.

Register i, $i \geq 1$, der RASP wird in Register $i+3$ der RAM gespeichert.

Die RAM beginnt mit dem RASP Programm im Speicher ab Register 4. Register 2, der RASP Befehlszähler, enthält 4. Die Register 1 und 3 enthalten 0. Das RAM Programm ist eine Simulationsschleife, welche damit beginnt eine RASP Instruktion zu lesen (mit einer `LOAD *2` Instruktion), diese dekodiert und zu einer von 18 Instruktionsfolgen verzweigt, deren jede einen Typ von RASP Instruktionen simuliert. Als Beispiel betrachten wir die RAM Instruktionsfolge zur Simulation der RASP Instruktion 6, also `SUB i`:

`LOAD 2`	erhöhe den Befehlszähler um 1, sodaß er nun auf das Register
`ADD =1`	zeigt, welches den Operanden i der `SUB i` Instruktion enthält;
`STORE 2`	
`LOAD *2`	lade i in den Akkumulator, addiere 3 und speichere
`ADD =3`	das Ergebnis in Register 1;
`STORE 1`	
`LOAD 3`	lade den Inhalt des RASP Akkumulators aus Register 3, sub–
`SUB *1`	trahiere den Inhalt des Registers $i+3$ und lege das Ergebnis
`STORE 3`	in Register 3 ab;
`LOAD 2`	erhöhe den Befehlszähler um 1, sodaß er nun auf die nächste
`ADD =1`	RASP Instruktion zeigt;
`STORE 2`	
`JUMP a`	verzweige zum Anfang der Simulationsschleife (hier mit “`a`” bezeichnet).

Zu diesem Programmstück wird verzweigt, wenn $c(c(2)) = 6$, der Befehlszähler also auf ein Register mit Inhalt 6, dem Kode von `SUB`, zeigt. ■

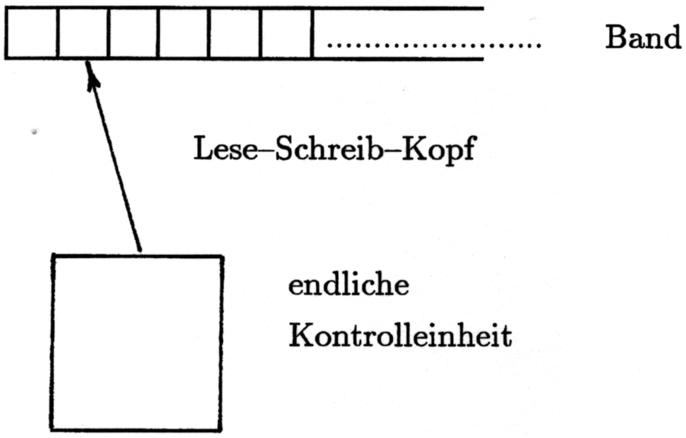
Beispiel 1.3.3: Es ist nicht schwer zu zeigen, daß die Sprachen aus den Beispielen 1.2.5 und 1.2.6 von einer RAM akzeptiert werden können. ■

Wie man auch unschwer sieht, kann jeder endliche Automat durch eine RAM simuliert werden. Wegen Beispiel 1.3.3 wissen wir also, daß das Konzept der RAM stärker ist als dasjenige endlicher Automaten, da mehr Sprachen akzeptiert werden können.

1.4. Turing–Maschinen und rekursiv aufzählbare Sprachen

Um zu beweisen, daß eine bestimmte Funktion "berechenbar" ist, oder ein gewisses Minimum an Zeit zur Berechnung benötigt (untere Komplexitätsschranke), brauchen wir ein Maschinenmodell, das zwar so allgemein aber einfacher als eine RAM oder RASP ist. Dieses Modell ist die Turing–Maschine.

Definition 1.4.1: Eine *Turing–Maschine* M können wir uns folgendermaßen vorstellen:



Sie besteht aus einer *endlichen Kontrolleinheit*, welche im Inneren eine endliche Anzahl von Zuständen aus einer *Zustandsmenge* $Q = \{q_0, \ldots, q_n\}$ annehmen kann, einem in eine Richtung unendlichen *Band*, welches in Zellen eingeteilt ist, deren jede ein Symbol aus einem endlichen *Alphabet von Bandsymbolen* $\Gamma = \{\gamma_0, \ldots, \gamma_m\}$ speichern kann — γ_0 wird dabei als das Blank Symbol $\sqcup$ aufgefaßt und eine Untermenge Σ von $\Gamma \setminus \{\sqcup\}$ ist das *Eingabealphabet* — , sowie aus einem *Lese–Schreib–Kopf*, welcher immer auf eine Bandzelle zeigt.

Wie funktioniert nun die Turing–Maschine M? Wenn die Maschine zu arbeiten beginnt, ist die endliche Kontrolleinheit im Anfangszustand q_0, und bis auf endlich viele Zellen am linken Ende des Bandes sind alle Bandzellen mit $\sqcup$'s belegt. In diskreten Zeitschritten, abhängig vom Zustand und vom Bandsymbol, auf welches der LS–Kopf gerade zeigt (Symbol in der *Arbeitszelle*), geschieht folgendes:

- M nimmt einen neuen Zustand an,
- schreibt ein Bandsymbol in die Arbeitszelle, und
- bewegt den LS–Kopf um eine Position nach rechts (R) oder nach links (L) oder läßt ihn stationär (S).

Erreicht dabei M einen Zustand aus einer Menge von *Endzuständen* F (Teilmenge der Zustandsmenge), so terminiert die Berechnung erfolgreich, M *akzeptiert* die Eingabe. Ansonsten fährt M in der Berechnung so lange fort, als zum jeweiligen Zustand und Symbol in der Arbeitszelle ein Nachfolgerzustand erklärt ist. Eine Berechnung, welche nicht erfolgreich terminiert, kann also auch unendliche Länge haben. ■

Beispiel 1.4.1: Als Beispiel betrachten wir die Turing–Maschine M_1, welche erkennt, ob eine endliche Folge x über $\{0, 1\}$ die Form 0^n1^n hat, für eine natürliche Zahl n. Bei Anfang der Berechnung steht die Folge x in den ersten Zellen des Bandes von M_1, gefolgt von einer unendlichen Folge von $\sqcup$'s. M_1 führt folgende Berechnungsschleife aus:

> M_1 ersetzt die am weitesten links stehende 0 durch X, bewegt den LS–Kopf nach rechts bis zur am weitesten links stehenden 1, ersetzt diese durch Y, bewegt den LS–Kopf nach links bis zum ersten X und anschließend um eine Position nach rechts zur am weitesten links stehenden 0 und wiederholt die Schleife.

Findet nun M_1 bei der Suche nach einer 1 anstatt dessen ein $\sqcup$, so stoppt M_1 die Berechnung und gibt "nein" als Antwort. Falls M_1, nachdem es eine 1 durch Y ersetzt hat, keine weitere 0 mehr findet, so prüft es, ob noch 1en vorhanden sind, und gibt in diesem Fall die Antwort "nein", andernfalls die Antwort "ja".

Etwas formaler könnten wir M_1 wie folgt angeben:

Zustandsmenge .... $Q = \{q_0, q_1, q_2, q_3, q_4\}$,
Eingabealphabet ... $\Sigma = \{0, 1\}$,
Bandalphabet ..... $\Gamma = \{\sqcup, 0, 1, X, Y\}$,
Endzustände ...... $F = \{q_4\}$,

Überführungsfunktion δ:

Zustand	Symbol				
	$\sqcup$	0	1	X	Y
q_0	–	(q_1, X, R)	–	–	(q_3, Y, R)
q_1	–	$(q_1, 0, R)$	(q_2, Y, L)	–	(q_1, Y, R)
q_2	–	$(q_2, 0, L)$	–	(q_0, X, R)	(q_2, Y, L)
q_3	$(q_4, \sqcup, R)$	–	–	–	(q_3, Y, R)
q_4	–	–	–	–	–

Jeden Zustand könnte man sich vorstellen als eine Anweisung oder eine Gruppe von Anweisungen in einem Programm.

– Der Zustand q_0 wird am Start angenommen und auch jeweils unmittelbar vor der Ersetzung der am weitesten links stehenden 0 durch X. Findet M_1 eine 0 vor, so ersetzt es diese durch X, geht über in q_1, und bewegt den LS–Kopf eine Position nach rechts. Findet q_0 ein Y vor, so nimmt M_1 den Zustand q_3 an.

– q_1 wird dazu benutzt, um von links nach rechts nach der ersten 1 zu suchen. Findet M_1 eine 1, so wird sie durch Y ersetzt, und M_1 nimmt den neuen Zustand q_2 an. Wird vor einer 1 ein $\sqcup$ oder ein X gefunden, so bricht die Berechnung ab, und die Eingabe wird zurückgewiesen.

– q_2 wird benutzt um von rechts nach links nach dem ersten X zu suchen. Wird ein X gefunden, so nimmt M_1 den Zustand q_0 an und bewegt den LS–Kopf eine Position nach rechts.

– Im Zustand q_3 wird der LS–Kopf über Y's hinweg nach rechts bewegt um zu prüfen, ob noch 1en auf dem Band vorkommen. Ist das erste Symbol nach den Y's ein $\sqcup$, so nimmt M_1 den Zustand q_4 an und die Eingabefolge wird damit akzeptiert. Andernfalls wird die Eingabefolge zurückgewiesen. ■

Definition 1.4.1: (Fortsetzung) Etwas formaler wird eine *Turing–Maschine* (kurz *TM*) so angegeben wie im obigen Beispiel, nämlich als ein 6–tupel $(Q, \Sigma, \Gamma, q_0, F, \delta)$, wobei Q die Zustandsmenge ist, Σ das Eingabealphabet, Γ das Bandalphabet, q_0 der Startzustand, F die Menge der Endzustände und δ die Überführungsfunktion. δ ist eine partielle Funktion von $Q \times \Gamma$ in $Q \times \Gamma \times \{L, R, S\}$. ■

Definition 1.4.2: Sei M ein TM. Eine *Konfiguration* von M legt den Zustand von M zu einem bestimmten Zeitpunkt fest, also eine Folge von Bandsymbolen $\alpha_1 \dots \alpha_k$, welche links vom LS–Kopf auf dem Band stehen, eine Folge von Bandsymbolen $\alpha_{k+1} \dots \alpha_l$, welche rechts vom LS–Kopf auf dem Band stehen einschließlich dem Symbol in der Arbeitszelle (die unendlich vielen $\sqcup$'s am rechten Bandende werden dabei vernachlässigt), sowie den Zustand q der endlichen Kontrolleinheit. Alle anderen Bandsymbole rechts von der Position l sind $\sqcup$. Wir schreiben dafür

$$\alpha_1 \dots \alpha_k q \alpha_{k+1} \dots \alpha_l.$$

Also $\alpha_1 \dots \alpha_k q \alpha_{k+1} \dots \alpha_l$ und $\alpha_1 \dots \alpha_k q \alpha_{k+1} \dots \alpha_l \sqcup$ sind nur zwei verschiedene Notationen für ein und dieselbe Konfiguration.

Eine Konfiguration $\beta_1 \dots \beta_k q \beta_{k+1} \dots \beta_m$ geht aus $\alpha_1 \dots \alpha_l p \alpha_{l+1} \dots \alpha_m$ hervor,

$$\alpha_1 \dots \alpha_l p \alpha_{l+1} \dots \alpha_m \quad \vdash \quad \beta_1 \dots \beta_k q \beta_{k+1} \dots \beta_m,$$

wenn $\alpha_i = \beta_i$ für $i \in \{1, \dots, l, l+2, \dots, m\}$ und

$$\delta(p, \alpha_{l+1}) = (q, \beta_{k+1}, S) \quad \text{und} \quad k = l,$$

oder

$$\delta(p, \alpha_{l+1}) = (q, \beta_k, R) \quad \text{und} \quad k = l+1,$$

oder

$$\delta(p, \alpha_{l+1}) = (q, \beta_{k+2}, L) \quad \text{und} \quad k = l-1.$$

Eine *Berechnung* der Turing–Maschine M ist eine Folge von Konfigurationen, wobei jeweils die $(i+1)$–te Konfiguration aus der i–ten Konfiguration hervorgeht.

M akzeptiert das Eingabewort $x \in \Sigma^*$, wenn die Berechnung ausgehend von der Konfiguration $q_0 x$ nach endlich vielen Schritten zu einer Konfiguration führt, in welcher der Zustand q ein Element von F ist. ■

Beispiel 1.4.2: Auf der Eingabe 0011 etwa führt die Turing–Maschine M_1 folgende Berechnung aus:

$$\begin{aligned}
&q_0 0011 \vdash X q_1 011 \vdash X 0 q_1 11 \vdash X q_2 0 Y 1 \vdash \\
&q_2 X 0 Y 1 \vdash X q_0 0 Y 1 \vdash X X q_1 Y 1 \vdash X X Y q_1 1 \vdash \\
&X X q_2 Y Y \vdash X q_2 X Y Y \vdash X X q_0 Y Y \vdash X X Y q_3 Y \vdash \\
&X X Y Y q_3 \vdash X X Y Y \sqcup q_4.
\end{aligned}$$

Die Eingabe 0011 wird also von M_1 akzeptiert. ■

Def. 1.4.3: Sei M eine Turing–Maschine . $L(M)$, die *von M akzeptierte Sprache*, ist die Menge aller Eingabewörter x (Folgen von Symbolen aus dem Eingabealphabet Σ), welche von M akzeptiert werden. ■

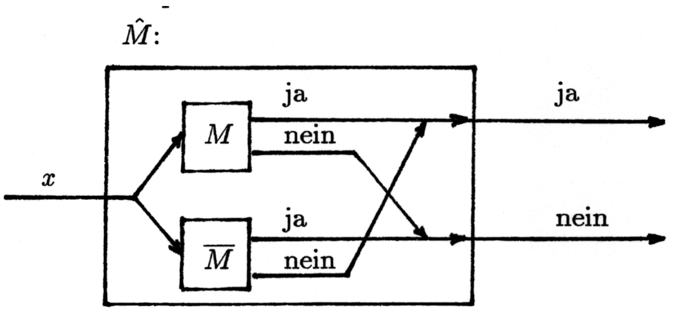
Also $L(M_1) = \{ \ 0^n 1^n \ \mid \ n \geq 1 \ \}$.

Def. 1.4.4: Eine Sprache L (Menge von Wörtern über einem Alphabet Σ) heißt *rekursiv aufzählbar (r.a.)*, wenn $L = L(M)$ für eine Turing–Maschine M. Eine r.a. Sprache L, welche von einer Turing–Maschine M akzeptiert wird, die auf jedem Eingabewort stoppt, heißt eine *rekursive* Sprache. In diesem Fall sagt man, M *erkennt* L. ■

Satz 1.4.1: *Die rekursiven Sprachen sind genau diejenigen Sprachen L, für die sowohl L als auch das Komplement $\overline{L}$ von L r.a. sind.*

Beweis: Sei L rekursiv. Sei M ein TM, welche immer terminiert und L akzeptiert. Dreht man die Antworten "ja/nein" von M einfach um, so erhält man eine TM $\overline{M}$, welche $\overline{L}$ akzeptiert.

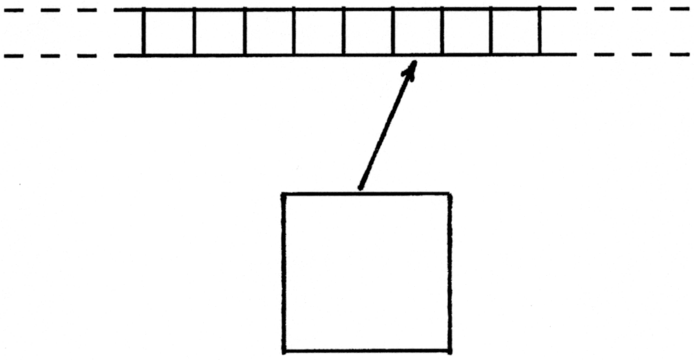
Umgekehrt seien sowohl L als auch $\overline{L}$ r.a. Seien $M, \overline{M}$ TMen, welche L bzw. $\overline{L}$ akzeptieren. Durch Parallelschaltung von M und $\overline{M}$ erhält man eine TM $\hat{M}$, welche immer terminiert und L akzeptiert.



Modifikationen von Turing–Maschinen

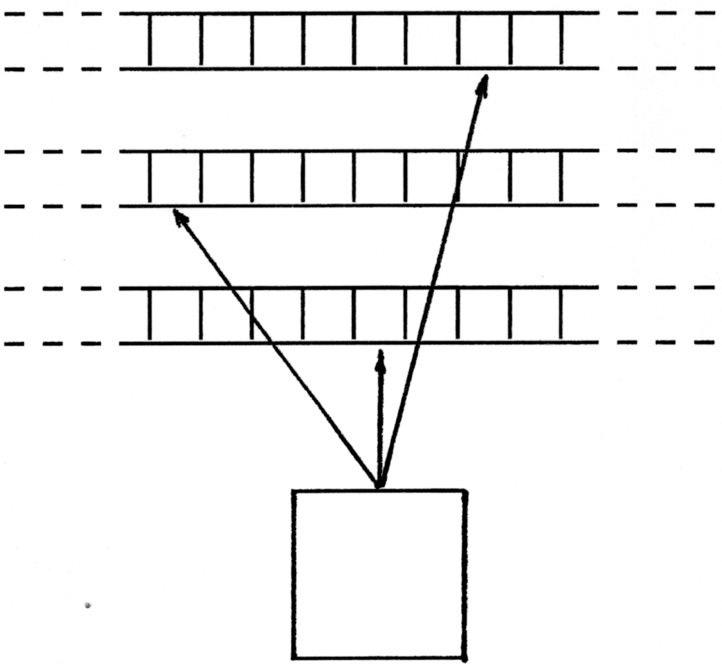
Das Konzept einer Turing–Maschine bietet sich an für eine ganze Reihe von Modifikationen, von denen wir einige kurz beschreiben wollen.

In beide Richtungen unendliches Band:



Das Band ist nicht nur nach rechts unendlich, sondern in beide Richtungen. Ansonsten ändert sich nichts an der Definition der Maschine. Bei Beginn der Berechnung steht das Eingabewort x auf dem Band, das ansonsten nur $\sqcup$'s enthält. Der LS–Kopf zeigt dabei auf das erste Symbol von x.

Turing–Maschine mit mehreren Bändern:



Anstatt eines einzigen Bandes hat die Maschine eine endliche Anzahl von Bändern zur Verfügung. Bei Beginn der Berechnung steht das Eingabewort x auf dem ersten Band, das ansonsten nur $\sqcup$'s enthält. Ebenso enthalten alle übrigen Bänder am Beginn der Berechnung nur $\sqcup$'s.

Abhängig vom Zustand der Kontrolleinheit und von den Symbolen, auf welche die LS–Köpfe zeigen, führt die Maschine M in einem Berechnungsschritt folgende Operationen aus:

– M nimmt einen neuen Zustand an,

– schreibt ein Bandsymbol in die Arbeitszelle eines jeden Bandes, und

– bewegt den LS–Kopf eines jeden Bandes unabhängig um eine Position nach rechts oder nach links, oder läßt ihn stationär.

Nichtdeterministische Turing–Maschine:

Abhängig vom Zustand und vom Symbol in der Arbeitszelle hat die Maschine endlich viele Möglichkeiten einen neuen Zustand anzunehmen, ein Symbol in die Arbeitszelle zu schreiben und den LS-Kopf zu bewegen.

Definition 1.4.5: Eine *nicht–deterministische Turing–Maschine* (kurz *nTM*) M ist definiert wie ein (deterministische) Turing–Maschine , nur liefert δ ausgehend von q

und α eine endliche Menge möglicher Antworten,

$$\delta(q, \alpha) = \{(p_1, \beta_1, X_1), \ldots, (p_m, \beta_m, X_m)\}.$$

Eine Konfiguration geht aus einer anderen hervor, wenn es eine mögliche Antwort der Überführungsfunktion gibt, mittels derer der Übergang hergestellt werden kann. Ausgehend von einer Konfiguration erhalten wir also nicht eine lineare Berechnung, sondern einen Berechnungsbaum.

M akzeptiert die Eingabe x, wenn der Berechnungsbaum ausgehend von der Konfiguration q_0x eine Konfiguration enthält, in welcher der Zustand der Kontrolleinheit q ein Element von F ist. ■

Beispiel 1.4.3: Sei L die Sprache

$$L = \{\, 0^n, (01)^n \;\mid\; n \geq 1 \,\}.$$

Die nichtdeterministische Turing–Maschine $M = (Q, \Sigma, \Gamma, q_0, F, \delta)$ akzeptiert L, wobei

$$\begin{aligned} Q &= \{q_0, q_1, q_2, q_3, q_4\}, \\ \Sigma &= \{0, 1\}, \\ \Gamma &= \{0, 1, \sqcup\}, \\ F &= \{q_4\}, \end{aligned}$$

und δ durch folgende Tabelle gegeben ist:

	$\sqcup$	0	1
q_0	–	$\{(q_1, 0, R), (q_2, 0, R)\}$	–
q_1	$\{(q_4, \sqcup, R)\}$	$\{(q_1, 0, R)\}$	–
q_2	–	–	$\{(q_3, 1, R)\}$
q_3	$\{(q_4, \sqcup, R)\}$	$\{(q_2, 0, R)\}$	–
q_4	–	–	–

■

Satz 1.4.2: *Sei M eine nichtdeterministische Turing–Maschine, welche die Sprache L akzeptiert, also $L = L(M)$. Dann gibt es eine deterministische Turing–Maschine M', welche ebenfalls L akzeptiert, also $L = L(M')$.*

Beweis: Für jedes Paar bestehend aus einem Zustand und einem Bandsymbol von M gibt es eine endliche Anzahl von Wahlmöglichkeiten in der Überführungsfunktion von M. Wir numerieren diese Möglichkeiten durch 1,2,... . Sei r die größte Anzahl von

Wahlmöglichkeiten für ein solches Paar. Dann kann jede endliche Folge von Wahlen dargestellt werden als eine Folge der Nummern 1 bis r. Natürlich entspricht nicht jede Folge solcher Nummern einer Folge von Wahlen, da es für ein gegebenes Paar eventuell weniger als r Wahlmöglichkeiten geben kann.

Die deterministische Turing–Maschine M' hat nun drei Bänder. Das erste Band ist das Eingabeband. Auf dem zweiten Band erzeugt M' systematisch endliche Folgen über $\{1, \ldots, r\}$. Dabei werden zunächst die Folgen der Länge 1 erzeugt, dann diejenigen der Länge 2, usw.

Für jede Folge, welche auf Band 2 erzeugt wird, kopiert M' die Eingabe auf Band 3 und simuliert M auf Band 3, wobei die Folge auf Band 2 dazu benutzt wird, um die Wahlen der entsprechenden Zweige des Berechnungsbaums durchzuführen. Gibt es im Berechnungsbaum von M einen Zweig, der zu einem akzeptierenden Zustand führt, so wird M' schließlich eine Kodierung dieses Zweiges auf Band 2 erzeugen, und bei der Simulierung dieses Zweiges wird M' die Eingabe akzeptieren. Gibt es im Berechnungsbaum von M keinen Zweig, der zu einem akzeptierenden Zustand führt, so wird auch M' nicht akzeptieren.

Also $L(M) = L = L(M')$. ■

Wir haben also nun einige Möglichkeiten zur Modifikation von Turing–Maschinen betrachtet. Keine dieser Möglichkeiten ist jedoch in der Lage, die Berechnungspotenz einer Turing–Maschine zu steigern. Eine Sprache L, welche von einer auf diese Weise modifizierten Turing–Maschine akzeptiert wird, kann auch von einer Turing–Maschine im ursprünglichen Sinn akzeptiert werden. Für eine genauere Betrachtung dieser Berechnungsäquivalenz verweisen wir auf [HU].

<u>Turing–Maschinen als Generatoren von Sprachen</u>

Definition 1.4.6: Sei M eine Turing–Maschine mit einem oder mehreren Bändern, welche das erste Band als Ausgabeband benützt. M hat ein ausgezeichnetes Bandsymbol #, und M bewegt den LS–Kopf des Ausgabebandes niemals nach links. In diesem Fall ist M geeignet eine Sprache zu erzeugen (generieren). Und zwar ist die *von M erzeugte (generierte) Sprache $G(M)$* die Menge aller Wörter, welche M ohne irgend eine Eingabe zwischen zwei #'s auf das Ausgabeband schreibt. ■

Satz 1.4.3: *Eine Sprache L über einem Alphabet Σ ist genau dann r.a., wenn L von einer Turing–Maschine generiert werden kann. (Daher auch die Bezeichnung "rekursiv* *aufzählbar".)*

Beweis: Wir wollen einen Beweis dieser Behauptung skizzieren. Angenommen L wird von der Turing–Maschine M generiert, also $G(M) = L$. Wir konstruieren eine Turing–Maschine M' mit einem Band mehr als M. M' simuliert M. Immer wenn M ein Wort x erzeugt, so vergleicht es M' mit dem Eingabewort. Stimmen sie überein, so akzeptiert M' das Eingabewort. Also $L(M') = L$.

Andererseits sei M eine Turing–Maschine, welche L akzeptiert, also $L(M) = L$. Wir konstruieren eine Turing–Maschine M', welche nacheinander alle Paare von positiven ganzen Zahlen (i, j) (in kodierter Form) erzeugt. Sobald das Paar (i, j) erzeugt worden ist, erzeugt M' das i–te Wort x_i über dem Alphabet Σ (in dieser Aufzählung sind die Wörter der Länge nach und Wörter gleicher Länge alphabetisch geordnet) und simuliert j Berechnungsschritte von M auf der Eingabe x_i. Wird x_i von M in nicht mehr als j Schritten akzeptiert, so erzeugt M' das Wort x_i. Somit ist $L = G(M')$. ■

Turing–Maschinen als Berechner von Funktionen

Turing–Maschinen können nicht nur zum Erkennen und Erzeugen von Sprachen eingesetzt werden, sondern sie sind auch in der Lage Funktionen zu berechnen.

Definition 1.4.7: Seien Σ_1 und Σ_2 zwei Alphabete und f eine (partielle) Funktion von $(\Sigma_1^*)^k$ in Σ_2^* (Σ_1^* ist die Menge aller Wörter über Σ_1). Eine Turing–Maschine M *berechnet* die Funktion f falls

- $\Sigma_1 \cup \Sigma_2$ eine Teilmenge des Bandalphabets von M ist, und
- für jedes $x = (x_1, \ldots, x_k) \in (\Sigma_1^*)^k$, wenn $f(x_1, \ldots, x_k) = y$ dann stoppt M auf dem Eingabewort $x_1 \sqcup \ldots \sqcup x_k$ mit Ausgabe (Bandinhalt nach Ende der Berechnung) y und wenn $f(x_1, \ldots, x_k)$ undefiniert ist, dann stoppt M auf dem Eingabewort $x_1 \sqcup \ldots \sqcup x_k$ nicht.

Eine (partielle) Funktion, welche von einer Turing–Maschine berechnet werden kann, heißt *(partiell) Turing–berechenbar*, abgekürzt *(p.)T.b.* ■

Beispiel 1.4.4: Ein Beispiel einer T.b. Funktion ist die modifizierte Differenz $\dot{-}$, welche für zwei natürliche Zahlen m, n den Wert $m - n$ liefert falls $m \geq n$ und den Wert 0 sonst. $\dot{-}$ wird von der Turing–Maschine M_2 berechnet mit

Zustandsmenge .... $Q = \{q_0, \ldots, q_6\}$,
Eingabealphabet ... $\Sigma = \{0\}$,

Bandalphabet ..... $\Gamma = \{\sqcup, 0, 1\}$,
Endzustände ...... $F = \{\}$.

Die Eingabeparameter m, n werden kodiert als

$$\underbrace{0\dots0}_{m \text{ mal}}\sqcup\underbrace{0\dots0}_{n \text{ mal}}.$$

Die Überführungsfunktion δ von M_2 arbeitet wie folgt:

(1) Start. Ersetze die führende 0 durch $\sqcup$. Ist keine 0 vorhanden, so lösche den Rest des Bandes.
$\delta(q_0, 0) = (q_1, \sqcup, R)$, $\delta(q_0, \sqcup) = (q_5, \sqcup, R)$.

(2) Bewege den LS–Kopf nach rechts und suche nach dem ersten $\sqcup$. Überschreibe dieses durch 1.
$\delta(q_1, 0) = (q_1, 0, R)$, $\delta(q_1, \sqcup) = (q_2, 1, R)$, $\delta(q_1, 1) = (q_2, 1, R)$.

(3) Bewege den LS–Kopf nach rechts über 1en hinweg bis zur ersten 0. Überschreibe diese durch 1.
$\delta(q_2, 1) = (q_2, 1, R)$, $\delta(q_2, 0) = (q_3, 1, L)$.

(4) Gehe nach links bis zum ersten $\sqcup$. Nimm Zustand q_0 an und wiederhole den Zyklus.
$\delta(q_3, 0) = (q_3, 0, L)$, $\delta(q_3, 1) = (q_3, 1, L)$, $\delta(q_3, \sqcup) = (q_0, \sqcup, R)$.

(5) Wird im Zustand q_2 ein $\sqcup$ vor einer 0 gefunden, so nimm Zustand q_4 an und gehe nach links, wobei alle 1en durch $\sqcup$'s überschrieben werden, bis zum ersten $\sqcup$. Ändere dieses $\sqcup$ wieder zu einer 0 ab, nimm Zustand q_6 an und stoppe.
$\delta(q_2, \sqcup) = (q_4, \sqcup, L)$, $\delta(q_4, 1) = (q_4, \sqcup, L)$, $\delta(q_4, 0) = (q_4, 0, L)$, $\delta(q_4, \sqcup) = (q_6, 0, R)$.

(6) Wird im Zustand q_0 eine 1 anstatt einer 0 gefunden, so ist offensichtlich der erste Block von 0en abgearbeitet. Nimm Zustand q_5 an und lösche den Rest des Bandes. Anschließend nimm Zustand q_6 an und stoppe.
$\delta(q_0, 1) = (q_5, \sqcup, R)$, $\delta(q_5, 0) = (q_5, \sqcup, R)$, $\delta(q_5, 1) = (q_5, \sqcup, R)$, $\delta(q_5, \sqcup) = (q_6, \sqcup, R)$.

Zustand	Symbol		
	0	1	$\sqcup$
q_0	$(q_1, \sqcup, R)$	$(q_5, \sqcup, R)$	$(q_5, \sqcup, R)$
q_1	$(q_1, 0, R)$	$(q_2, 1, R)$	$(q_2, 1, R)$
q_2	$(q_3, 1, L)$	$(q_2, 1, R)$	$(q_4, \sqcup, L)$

Zustand	Symbol 0	1	$\sqcup$
q_3	$(q_3, 0, L)$	$(q_3, 1, L)$	$(q_1, \sqcup, R)$
q_4	$(q_4, 0, L)$	$(q_4, \sqcup, L)$	$(q_6, 0, R)$
q_5	$(q_5, \sqcup, R)$	$(q_5, \sqcup, R)$	$(q_6, 0, R)$
q_6	$-$	$-$	$-$

Berechnung von M_2 auf $00 \sqcup 0 \quad (2 \dot{-} 1)$:

$$\begin{aligned}
&q_0 00 \sqcup 0 \vdash \sqcup q_1 0 \sqcup 0 \vdash \sqcup 0 q_1 \sqcup 0 \vdash \sqcup 01 q_2 0 \vdash \\
&\sqcup 0 q_3 11 \vdash \sqcup q_3 011 \vdash q_3 \sqcup 011 \vdash \sqcup q_0 011 \vdash \\
&\sqcup \sqcup q_1 11 \vdash \sqcup \sqcup 1 q_2 1 \vdash \sqcup \sqcup 11 q_2 \vdash \sqcup \sqcup 1 q_4 1 \vdash \\
&\sqcup \sqcup q_4 1 \vdash \sqcup q_4 \vdash \sqcup 0 q_6.
\end{aligned}$$

Berechnung von M_2 auf $0 \sqcup 00 \quad (1 \dot{-} 2)$:

$$\begin{aligned}
&q_0 0 \sqcup 00 \vdash \sqcup q_1 \sqcup 00 \vdash \sqcup 1 q_2 00 \vdash \sqcup q_3 110 \vdash \\
&q_3 \sqcup 110 \vdash \sqcup q_0 110 \vdash \sqcup \sqcup q_5 10 \vdash \sqcup \sqcup \sqcup q_5 0 \vdash \\
&\sqcup \sqcup \sqcup \sqcup q_5 \vdash \sqcup \sqcup \sqcup \sqcup \sqcup q_6.
\end{aligned}$$

■

Alle üblichen arithmetischen Funktionen wie $m + n$, $m \cdot n$, m^n, $n!$, $\lceil \log_2 n \rceil$, etc. sowie Kombinationen davon sind T.b.

Zwischen r.a. Sprachen und p.T.b. Funktionen besteht ein enger Zusammenhang.

Satz 1.4.4: *Ist f eine partielle Funktion von $(\Sigma_1^*)^k$ nach Σ_2^*, so ist f genau dann p.T.b., wenn $L_f = \{ (x_1, \ldots, x_k, y) \mid f(x_1, \ldots, x_k) = y \}$ r.a. ist.*

Beweis: Übung. ■

Im folgenden befassen wir uns mit der Äquivalenz zwischen dem RAM/RASP Modell und dem Turing–Maschinen Modell.

Satz 1.4.5: *Jede Turing–Maschine (mit mehreren Bändern) kann durch eine RAM simuliert werden. Also jede Sprache (Funktion), die von einer Turing–Maschine akzeptiert (berechnet) werden kann, kann auch von einer RAM akzeptiert (berechnet) werden.*

Beweis: (Skizze) Sei M eine k–Band Turing–Maschine . In der M simulierenden RAM R ist jede Bandzelle von M in einem Register abgespeichert, und zwar ist die i–te Zelle des j–ten Bandes im Register $ki + j + c$ abgespeichert. Dabei ist c so angelegt, daß R noch Arbeitsspeicher zur Verfügung hat, u.a. zur Abspeicherung der Positionen der k LS–Köpfe von M. Bandzellen von M können von R gelesen werden durch indirekte Adressierung mittels der Register, welche die Positionen der LS–Köpfe beinhalten. ■

Satz 1.4.6: *Jede RAM kann durch eine Turing–Maschine simuliert werden. Also jede Sprache (Funktion), die von einer RAM akzeptiert (berechnet) werden kann, kann auch von einer Turing–Maschine akzeptiert (berechnet) werden.*

Beweis: (Skizze) Die Turing–Maschine M, die die RAM R simuliert, besitzt neben den Ein– und Ausgabebändern noch drei zusätzliche Bänder B_3, B_4, B_5. B_3 dient zur Darstellung des Speichers von R und hat die Form

#	#	#	i_1	#	c_1	#	#	i_2#	c_2	#	#	$\cdots$	i_k	#	c_k	#	#	$\sqcup$	$\ldots$

Dabei sind $i_1, \ldots, i_k$ die Indizes derjenigen Register von R, welche nicht 0 enthalten und $c_j = c(i_j)$; i_j, c_j sind in Binärdarstellung auf B_3 wiedergegeben, für $1 \leq i \leq k$. Der Inhalt des Akkumulators von R ist in Binärdarstellung auf B_4 dargestellt, und B_5 wird als Arbeitsband verwendet.

Jeder Rechenschritt von R entspricht einer endlichen Folge von Zuständen der Turing–Maschine M. Als Beispiel betrachten wir die RAM Instruktionen `ADD *i` und `STORE i`. Die Instruktion `ADD *i` wird von M folgendermaßen simuliert:

1. Suche auf B_3 nach einer Eintragung für das RAM Register i, d.h. eine Folge $\#\#i'\#$, wobei i' die Binärdarstellung von i ist. Ist eine solche Eintragung vorhanden, so schreibe die nächste Zahl, welche der Inhalt von Register i ist, auf B_5; andernfalls halte (der Inhalt von Register i ist 0, die indirekte Adressierung kann also nicht durchgeführt werden).
2. Suche auf B_3 nach einer Eintragung für das RAM Register, dessen Index auf B_5 steht. Ist eine solche Eintragung vorhanden, so schreibe den Inhalt dieses Registers auf B_5; andernfalls schreibe 0 auf B_5.
3. Addiere die Zahl auf B_5 zum Inhalt des Akkumulators, der auf B_4 steht.

Die Instruktion `STORE i` wird von M folgendermaßen simuliert:

1. Suche auf B_3 nach einer Eintragung für das RAM Register i, also $\#\#i'\#$.

2. Ist eine solche Eintragung vorhanden, so kopiere den Bandinhalt von B_3 rechts von $\#\#i'\#$, außer der unmittelbar nächsten Zahl ($c(i)$), auf B_5. Kopiere den Inhalt von B_4 (Akkumulator) auf B_3 unmittelbar rechts von $\#\#i'\#$ und unmittelbar daran den Inhalt von B_5.
3. Ist eine Eintragung für Register i nicht vorhanden, so schreibe $i'\#$ auf B_3 unmittelbar nach dem letzten von $\sqcup$ verschiedenen Symbol, anschließend den Inhalt von B_4 (Akkumulator) und schließlich $\#\#$.

Auf ganz ähnliche Weise können auch die übrigen RAM Instruktionen simuliert werden.

■

Die Menge der (partiell) T.b. (zahlentheoretischen) Funktionen ist nicht nur identisch mit der Menge der (partiell) RAM berechenbaren Funktionen, sie ist auch identisch mit der Menge der (partiell) rekursiven Funktionen, oder mit der Menge der WHILE-berechenbaren Funktionen (siehe etwa [DW], [Sch]). Alle diese mathematischen Modelle liefern dieselbe Klasse algorithmischer Funktionen. Diese Klasse scheint also in gewisser Weise eine natürliche zu sein. Andererseits sind alle (bisher betrachteten) Funktionen, die man intuitiv als berechenbar bezeichnen würde, tatsächlich rekursiv. Die meisten Mathematiker und Computerwissenschafter sind daher der Ansicht, daß mit dem formalen Begriff der rekursiven bzw. Turing–berechenbaren Funktion der intuitive Begriff der "algorithmischen" Funktion charakterisiert ist. Diese Überzeugung wurde zum ersten Mal von A. Church ausgesprochen und heißt nach ihm die *Church'sche These*:

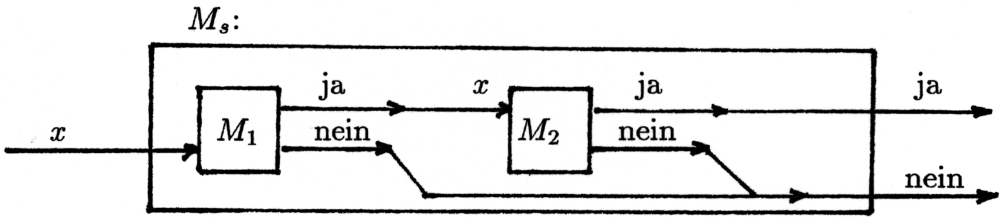
> *Eine Funktion ist berechenbar genau dann wenn sie Turing–berechenbar (rekursiv) ist.*

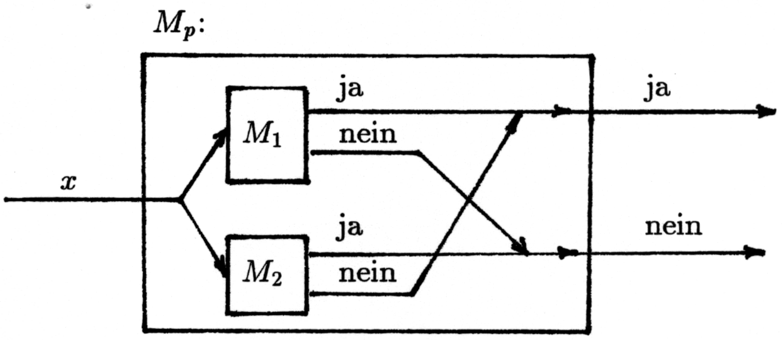
Im strengen Sinne beweisen läßt sich die Church'sche These natürlich nicht, da man auf einem intuitiven Begriff keinen Beweis aufbauen kann. Aufgrund der Church'schen These versteht man heutzutage unter einem Algorithmus eine Turing–Maschine . Wenn wir also sagen, daß sich ein Problem algorithmisch lösen läßt, so meinen wir streng genommen damit, daß es eine Turing–Maschine gibt, die das Problem löst.

Definition 1.4.8: Ein *Algorithmus* ist eine Turing–Maschine , die auf jeder Eingabe terminiert. ■

Bisher haben wir nur Algorithmen definiert, welche ganze Zahlen als Eingaben nehmen und wiederum ganze Zahlen produzieren. Im intuitiven Sinne kennt man natürlich auch Algorithmen mit ganz anderen Ein– und Ausgabebereichen. Zur Formalisierung solcher Algorithmen gibt es im Prinzip zwei Ansätze. Beim ersten Ansatz ordnet man jedem Element der Ein– und Ausgabemenge eine natürliche Zahl zu und kann dann das herkömmliche Konzept des zahlentheoretischen Algorithmus verwenden. Beim zweiten Ansatz verlangt man nur, das sich die Elemente der Ein– und Ausgabemenge als Wörter über einem Alphabet schreiben lassen. Solche Wörter können nun von einer Turing–Maschine gelesen bzw. geschrieben werden und man erhält auf diese Weise eine Verallgemeinerung des Begriffs "Algorithmus".

Übung: Seien M_1, M_2 Turing–Maschinen. Dann gibt es Turing–Maschinen M_s, M_p mit dem Verhalten





Übung: Das Komplement einer rekursiven Sprache ist rekursiv.

Übung: Die Vereinigung zweier rekursiver Sprachen ist rekursiv. Die Vereinigung zweier r.a. Sprachen ist r.a.

1.5. Rekursive Funktionen

Im vorangegangenen Abschnitt haben wir gesehen, wie Turingmaschinen zur Berechnung von Funktionen, insbesondere zahlentheoretischer Funktionen (von $\mathbb{N}^k$ nach $\mathbb{N}$) eingesetzt werden können. Wir haben auf diese Weise den Begriff der Turing-berechenbaren Funktion erhalten. Die Äquivalenz von Turingmaschinen und RAMs kann natürlich von der Akzeptierung von Sprachen auf die Berechnung von Funktion ausgedehnt werden, sodaß jede Turing-berechenbare Funktion auch von einer RAM berechnet werden kann. Wir sprechen also in Hinkunft nur mehr von berechenbaren Funktionen.

In diesem Abschnitt wollen wir einen anderen, rein mathematisch-logischen Zugang zum Begriff der berechenbaren Funktion behandeln, nämlich den Begriff der rekursiven Funktion. Wiederum kann man feststellen, daß der Begriff der berechenbaren Funktion mit demjenigen der rekursiven Funktion übereinstimmt. Somit haben wir eine weitere Bestätigung der Church'schen These vorliegen. Wir werden hier weitgehend auf Beweise der Äquivalenz von Berechenbarkeit und Rekursivität verzichten. Der Leser ist aufgefordert, sich mithilfe der Literaturliste jeweils Rechenschaft über diese Äquivalenz zu geben.

Immer, wenn wir in diesem Abschnitt von einer Funktion sprechen, so meinen wir eine zahlentheoretische Funktion im obigen Sinne.

Definition 1.5.1: Sei f eine Funktion von k Variablen, und seien $g_1, \ldots, g_k$ Funktionen von n Variablen. Sei

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n)).$$

Dann entsteht h aus f und $g_1, \ldots, g_k$ durch *Komposition.* ■

Natürlich müssen die Funktionen $f, g_1, \ldots, g_k$ nicht total sein, sondern können auch partiell sein. $h(x_1, \ldots, x_n)$ ist definiert, wenn alle $z_i = g_i(x_1, \ldots, x_n)$ definiert sind, und auch $f(z_1, \ldots, z_k)$ definiert ist.

Definition 1.5.2: Sei f eine Funktion von n Variablen und g eine Funktion von $n+2$ Variablen. Sei

$$h(m, x_1, \ldots, x_n) = \begin{cases} f(x_1, \ldots, x_n), & \text{falls } m = 0, \\ g(m-1, h(m-1, x_1, \ldots, x_n), x_1, \ldots, x_n), & \text{falls } m > 0. \end{cases}$$

Dann entsteht h aus f und g durch *Rekursion.* ■

Um die Klasse der rekursiven Funktionen einzuführen, werden wir Funktionen mittels Komposition und Rekursion (und der noch einzuführenden Minimalisierung) kombinieren. Wir brauchen aber auch eine Basis von Grundfunktionen, von denen wir ausgehen können.

Definition 1.5.3: Wir betrachten folgende Funktionen:

Nullfunktion:	$\mathrm{null}() = 0,$
Nachfolgerfunktion (successor):	$\mathrm{succ}(x) = x + 1,$
Projektionsfunktionen:	$\mathrm{proj}_i^n(x_1, \ldots, x_n) = x_i,\ 1 \leq i \leq n.$

Die Funktionen $\mathrm{null}, \mathrm{succ}, \mathrm{proj}_i^n$ heißen *Grundfunktionen*. ■

Damit haben wir nun die Voraussetzungen geschaffen, um zunächst den Begriff der primitiv rekursiven Funktion einzuführen.

Definition 1.5.4: Die Klasse $\mathcal{PR}$ der *primitiv rekursiven Funktionen* ist induktiv wie folgt definiert:

(i) die Grundfunktionen gehören zu $\mathcal{PR}$,

(ii) alle Funktionen, welche aus Funktionen in $\mathcal{PR}$ durch Komposition und Rekursion erzeugt werden, gehören zu $\mathcal{PR}$.

Keine andere Funktion gehört zu $\mathcal{PR}$. ■

Beispiel 1.5.1: Viele der bekannten zahlentheoretischen Funktionen sind primitiv rekursiv. Wir weisen das für einige Funktionen nach.

(a) "$x + y$" ist primitiv rekursiv: die Additionsfunktion $f(x, y) = x + y$ ist rekursiv definiert als

$$f(0, y) = y,$$
$$f(x + 1, y) = f(x, y) + 1.$$

Diese Definition kann geschrieben werden als

$$f(x, y) = \begin{cases} \mathrm{proj}_1^1(y), & \text{falls } x = 0, \\ \mathrm{succ}(\mathrm{proj}_2^3(x - 1, f(x - 1, y), y)), & \text{falls } x > 0. \end{cases}$$

"$x+y$" entsteht also aus den Grundfunktionen durch Rekursion und Komposition.

(b) "$x \cdot y$" ist primitiv rekursiv: die Multiplikationsfunktion $g(x, y) = x \cdot y$ ist rekursiv definiert als

$$g(0, y) = 0,$$
$$g(x + 1, y) = g(x, y) + y.$$

Diese Definition kann geschrieben werden als

$$g(x, y) = \begin{cases} \text{null}(y), & \text{falls } x = 0, \\ r(x-1, g(x-1, y), y), & \text{falls } x > 0. \end{cases}$$

Dabei sei $f(x, y) = x + y$ wie oben, und

$$r(y, v, w) = f(\text{proj}_2^3(u, v, w), \text{proj}_3^3(u, v, w)).$$

Die unäre Nullfunktion $\text{null}(y) = 0$ entsteht aus null() durch Komposition (dabei ist $k = 0$ in Def. 1.5.1). "$x \cdot y$" entsteht also aus den Grundfunktionen durch Rekursion und Komposition.

(c) "$x!$" ist primitiv rekursiv: die Rekursionsgleichungen sind

$$\begin{aligned} 0! &= 1, \\ (x+1)! &= x! \cdot \text{succ}(x). \end{aligned}$$

Die Funktion $h(x) = x!$ kann also geschrieben werden als

$$h(x) = \begin{cases} \text{succ}(\text{null}()), & \text{falls } x = 0, \\ s(x-1, h(x-1)), & \text{falls } x > 0, \end{cases}$$

wobei

$$s(x_1, x_2) = \text{succ}(x_1) \cdot x_2.$$

Die Faktoriellenfunktion "$x!$" ist also primitiv rekursiv.

In den folgenden Beispielen primitiv rekursiver Funktionen überlassen wir es dem Leser zu überprüfen, daß die Rekursionsgleichungen in die exakte Gestalt gebracht werden können, wie sie von der Definition gefordert wird.

(d) "x^y": die Rekursionsgleichungen sind

$$\begin{aligned} x^0 &= 1, \\ x^{y+1} &= x^y \cdot x. \end{aligned}$$

(e) "pred(x)": die Vorgängerfunktion pred(x) ist folgendermaßen definiert:

$$\text{pred}(x) = \begin{cases} 0, & \text{falls } x = 0, \\ x - 1, & \text{sonst.} \end{cases}$$

(f) Die modifizierte Differenz "$x \dot{-} y$" ist definiert als

$$x \dot{-} y = \begin{cases} x - y, & \text{falls } x \geq y, \\ 0, & \text{falls } x < y. \end{cases}$$

Die Rekursionsgleichungen sind

$$x \dot{-} 0 = x,$$
$$x \dot{-} (t+1) = \mathrm{pred}(x \dot{-} t).$$

(g) Die Abstandsfunktion $| x - y |$ ist definiert als Absolutwert der Differenz zwischen x und y. Sie kann dargestellt werden als

$$| x - y | = (x \dot{-} y) + (y \dot{-} x). \qquad \blacksquare$$

Unter einem *Prädikat* verstehen wir einfach eine Funktion, welche nur die Werte 0 oder 1 annimmt. Interpretieren wir 1 als "wahr" und 0 als "falsch", so erhalten wir die übliche Vorstellung von einem Prädikat als boolescher Funktion. Da also Prädikate nichts anderes als spezielle zahlentheoretische Funktionen sind, können wir problemlos von primitiv rekursiven Prädikaten sprechen.

Beispiel 1.5.2: Die folgenden Prädikate sind primitiv rekursiv.

(a) Das Prädikat $\mathrm{istnull}(x)$ ist definiert als

$$\mathrm{istnull}(x) = \begin{cases} 1, & \text{falls } x = 0, \\ 0, & \text{falls } x \neq 0. \end{cases}$$

Wir können auch schreiben $\mathrm{istnull}(x) = 1 \dot{-} x$.

(b) Das Gleichheitsprädikat "$x = y$" entspricht der Funktion

$$d(x, y) = \begin{cases} 1, & \text{falls } x = y, \\ 0, & \text{falls } x \neq y. \end{cases}$$

Das Gleichheitsprädikat ist also primitiv rekursiv, da wir schreiben können

$$d(x, y) = \mathrm{istnull}(| x - y |).$$

(c) Das Prädikat "$x \leq y$" ist einfach die Funktion $\mathrm{istnull}(x \dot{-} y)$. ■

Satz 1.5.1: *Sind P, Q primitiv rekursive Prädikate, dann sind auch $\neg P, P \vee Q, P \wedge Q$ primitiv rekursiv.*

Beweis: $\neg P = \mathrm{istnull}(P)$, somit ist $\neg P$ primitiv rekursiv.

Die Konjunktion ist primitiv rekursiv wegen $P \wedge Q = P \cdot Q$.

Schließlich sieht man aus der Regel von De Morgan,

$$P \vee Q \iff \neg(\neg P \wedge \neg Q),$$

daß auch die Disjunktion primitiv rekursiv ist. ∎

Satz 1.5.2: *Seien g und h primitiv rekursive Funktionen und P ein primitiv rekursives Prädikat. Sei f die folgendermaßen definierte Funktion*

$$f(x_1,\ldots,x_n) = \begin{cases} g(x_1,\ldots,x_n) & \textit{falls } P(x_1,\ldots,x_n) \\ h(x_1,\ldots,x_n) & \textit{sonst.} \end{cases}$$

Dann ist auch f primitiv rekursiv.

Beweis: Das sieht man sofort aus der Beziehung

$$\begin{aligned} &f(x_1,\ldots,x_n) = \\ &g(x_1,\ldots,x_n)\cdot P(x_1,\ldots,x_n) + h(x_1,\ldots,x_n)\cdot(\neg P(x_1,\ldots,x_n)). \end{aligned}$$

∎

Satz 1.5.3: *Sei $f(t,x_1,\ldots,x_n)$ eine primitiv rekursive Funktion. Dann sind auch die folgendermaßen definierten Funktionen g und h primitiv rekursiv.*

$$\begin{aligned} g(y,x_1,\ldots,x_n) &= \sum_{t=0}^{y} f(t,x_1,\ldots,x_n), \\ h(y,x_1,\ldots,x_n) &= \prod_{t=0}^{y} f(t,x_1,\ldots,x_n). \end{aligned}$$

Beweis: Es gelten die Rekursionsgleichungen

$$\begin{aligned} g(0,x_1,\ldots,x_n) &= f(0,x_1,\ldots,x_n), \\ g(t+1,x_1,\ldots,x_n) &= g(t,x_1,\ldots,x_n) + f(t+1,x_1,\ldots,x_n). \end{aligned}$$

Da die Addition primitiv rekursiv ist, ist auch g primitiv rekursiv.

Analog verfährt man mit

$$\begin{aligned} h(0,x_1,\ldots,x_n) &= f(0,x_1,\ldots,x_n), \\ h(t+1,x_1,\ldots,x_n) &= h(t,x_1,\ldots,x_n)\cdot f(t+1,x_1,\ldots,x_n). \end{aligned}$$

(Warum ist es nicht zielführend, den Beweis durch Induktion über y zu versuchen?) ∎

Satz 1.5.4: *Sei $P(t,x_1,\ldots,x_n)$ ein primitiv rekursives Prädikat. Dann sind auch die folgenden Prädikate primitiv rekursiv,*

$$(\forall t)_{\le y} P(t,x_1,\ldots,x_n) \quad \text{und} \quad (\exists t)_{\le y} P(t,x_1,\ldots,x_n).$$

Beweis: Das folgt sofort aus der Beobachtung

$$(\forall t)_{\leq y} P(t, x_1, \ldots, x_n) \iff \Big[\prod_{t=0}^{y} P(t, x_1, \ldots, x_n)\Big] = 1$$

und

$$(\exists t)_{\leq y} P(t, x_1, \ldots, x_n) \iff \Big[\sum_{t=0}^{y} P(t, x_1, \ldots, x_n)\Big] \neq 0. \qquad \blacksquare$$

Beispiel 1.5.3: Die folgenden Prädikate sind primitiv rekursiv.

(a) "$y \mid x$", also das Prädikat "y teilt x".

$$y \mid x \iff (\exists t)_{\leq x}(y \cdot t = x).$$

(b) "Prime(x)", also das Prädikat "x ist eine Primzahl".

$$\mathrm{Prime}(x) \iff (x \geq 2) \wedge (\forall t)_{\leq x}\{t = 1 \vee t = x \vee \neg(t \mid x)\}. \qquad \blacksquare$$

Wie wir also gesehen haben, umfaßt die Klasse der primitiv rekursiven Funktionen eine große Anzahl der typischen zahlentheoretischen Funktionen und Prädikate. Alle diese primitiv rekursiven Funktionen sind (Turing-)berechenbar. Es gibt aber andererseits berechenbare Funktionen, welche nicht primitiv rekursiv sind. Ein klassisches Beispiel dafür ist die Ackermannfunktion

$$\begin{aligned}
\mathrm{ack}(0, y) &= y + 1,\\
\mathrm{ack}(x, 0) &= \mathrm{ack}(x-1, 1),\\
\mathrm{ack}(x, y) &= \mathrm{ack}(x-1, \mathrm{ack}(x, y-1)), \quad \text{sonst.}
\end{aligned}$$

Einen Beweis dieser Tatsache findet man etwa in [DW]. Um auch solche Funktionen behandeln zu können, muß man die Klasse $\mathcal{PR}$ erweitern um das Konstruktionsprinzip der Minimalisierung. Dadurch erhält man dann die Klasse der rekursiven Funktionen, welche dann tatsächlich mit den berechenbaren Funktionen übereinstimmt.

Definition 1.5.5: Sei $P(y, x_1, \ldots, x_n)$ ein Prädikat und sei $\min_y P(y, x_1, \ldots, x_n)$ die (partielle) Funktion, welche den kleinsten Wert y ergibt, sodaß das Prädikat $P(y, x_1, \ldots, x_n)$ wahr ist, falls ein solches y existiert. Andernfalls ist $\min_y P(y, x_1, \ldots, x_n)$ undefiniert. $\blacksquare$

Definition 1.5.6: Die Klasse $\mathcal{R}$ der *rekursiven Funktionen* ist induktiv wie folgt definiert:

(i) die Grundfunktionen gehören zu $\mathcal{R}$,

(ii) alle Funktionen, welche aus Funktionen (und Prädikaten) in $\mathcal{R}$ durch Komposition, Rekursion und Minimalisierung erzeugt werden, gehören zu $\mathcal{R}$.

Keine andere Funktion gehört zu $\mathcal{R}$. ∎

2. Komplexität von Algorithmen

2.1. Komplexitätsmaße

Wie wachsen die Kosten für Zeit und Raum für die Ausführung eines Algorithmus, wenn die Größe des Problems wächst? Das ist das Komplexitätsproblem für Algorithmen. Bevor man über das Komplexitätsverhalten eines Algorithmus sprechen kann, muß man ein Maß für die Problemgröße festlegen. So könnte ein Maß für die Größe eines Matrizenmultiplikationsproblems etwa die größte Dimension der Matrizen sein, oder ein Maß für die Größe eines Graphenproblems könnte die Zahl der Kanten sein.

Definition 2.1.1: (informal) Der Zeitaufwand, der für die Ausführung eines Algorithmus benötigt wird, ausgedrückt als Funktion der Problemgröße, heißt die *Zeitkomplexität* des Algorithmus. Das Grenzverhalten der Komplexität, wenn die Problemgröße gegen ∞ tendiert, heißt die *asymptotische Zeitkomplexität* des Algorithmus. Analog kann man die *Raumkomplexität* und die *asymptotische Raumkomplexität* eines Algorithmus definieren. Wird für eine gegebene Problemgröße die Komplexität als maximale Komplexität über alle Eingaben dieser Größe gemessen, so spricht man von der *worst–case Komplexität.* Wird die Komplexität als mittlere Komplexität über alle Eingaben einer festen Größe gemessen, so spricht man von der *erwarteten* oder *durchschnittlichen Komplexität.* ∎

Im allgemeinen werden wir die worst–case Komplexität eines Algorithmus messen, da sie mathematisch leichter handhabbar ist als die erwartete Komplexität. Man muß sich aber darüber im klaren sein, daß der Algorithmus mit der besten worst–case Komplexität nicht notwendigerweise auch die beste erwartete Komplexität besitzt. Bei der Bestimmung der Komplexität eines Algorithmus ist man nur an der Ordnung der Komplexitätsfunktion interessiert, d.h. additive und multiplikative Konstante werden vernachlässigt.

Definition 2.1.2: Eine Funktion $g(n)$ auf den natürlichen Zahlen ist $\mathcal{O}(f(n))$ (*von der Ordnung* $f(n)$), falls $g(n) \leq c \cdot f(n)$ für eine positive reelle Konstante c und alle genügend großen n ($n \geq N$ für ein $N \in \mathbb{N}$). ∎

Beispiel 2.1.1: a) $6n^2 + 7n$ ist $\mathcal{O}(n^2)$, denn $6n^2 + 7n \leq 7n^2$ für $n \geq 7$.
b) $\log_a n$ ist $\mathcal{O}(\log_b n)$ für $a, b \in \mathbf{R}^+$, denn $\log_a n = \log_a b \cdot \log_b n$. ∎

Nun könnte man annehmen, daß eine Steigerung der Geschwindigkeit von Rechenmaschinen die Entwicklung schneller Algorithmen überflüssig machen würde. Es tritt jedoch gerade der gegenteilige Effekt ein. Werden die Computer immer schneller und können wir dadurch immer größere Probleme behandeln, so ist es schließlich das Komplexitätsverhalten der Algorithmen welches die Umsetzbarkeit der schnelleren Hardware in größere Probleme bestimmt. Nehmen wir etwa an wir hätten fünf Algorithmen $A_1, \ldots, A_5$ mit der folgenden Zeitkomplexität:

Algorithmus	Zeitkomplexität
A_1	n
A_2	$n \log n$
A_3	n^2
A_4	n^3
A_5	2^n

Die Zeitkomplexität ist die Anzahl der Zeiteinheiten (etwa Millisekunden), die benötigt werden, um eine Eingabe der Größe n zu verarbeiten. Die folgende Tabelle gibt an, wie die Größe der verarbeitbaren Problem wächst mit der dafür aufgewendeten Zeit.

Algorithmus	Zeitkomplexität	Laufzeit		
		1 Sek.	1 Min.	1 Std.
A_1	n	1 000	6×10^4	3.6×10^6
A_2	$n \log n$	140	4 893	2.0×10^5
A_3	n^2	31	244	1 897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

Nehmen wir nun an, daß die nächste Generation von Computern zehn mal so schnell sei wie die derzeitige. Die folgende Tabelle stellt dar, wie stark sich die Zunahme der Geschwindigkeit auswirkt auf die Größe der Probleme, die behandelt werden können.

Algorithmus	Zeitkomplexität	maximale Problemgröße...	
		...vor speed-up	...nach speed-up
A_1	n	s_1	$10s_1$
A_2	$n \log n$	s_2	$\sim 10s_1$ für große s_2
A_3	n^2	s_3	$3.16s_3$
A_4	n^3	s_4	$2.15s_4$
A_5	2^n	s_5	$s_5 + 3.3$

Betrachten wir andererseits den Gewinn, wenn wir von einem langsamen Algorithmus zu einem schnelleren übergehen. Ersetzt man etwa A_4 durch A_3, so kann man in einer Minute ein sechs mal so großes Problem lösen; ersetzt man A_4 durch A_2, so kann man ein 125 mal so großes Problem lösen. Der Unterschied wird noch gravierender, wenn man die Rechenzeit erhöht.

2.2. Komplexität von RAM Programmen

Die Zeitkomplexität eines RAM Programms ergibt sich als Summe der benötigten Zeit für jede Instruktion des Programms. Analog ergibt sich die Raumkomplexität eines RAM Programms als Summe des benötigten "Raumes" für jedes angesprochene Register. Wir betrachten zwei verschiedene *Kostenkriterien* für RAM Programme. Unter dem *uniformen Kostenkriterium* benötigt jede RAM Instruktion eine Zeiteinheit und jedes Register eine Raumeinheit. Wenn nichts anderes ausdrücklich vereinbart ist, werden wir immer das uniforme Kostenkriterium zur Beurteilung der Komplexität von RAM Programmen heranziehen.

Das *logarithmische Kostenkriterium* zieht in Betracht, daß auf allen gegenwärtig verfügbaren Rechnern ein Speicherwort beschränkt ist, die Anzahl der benötigten Speicherworte für ein Register also abhängt von der Länge des Inhalts. Im allgemeinen braucht man $\lfloor \log n \rfloor + 1$ Bits, um eine Zahl n binär darzustellen. Sei $l(n)$ die folgende *logarithmische Funktion* auf den ganzen Zahlen:

$$l(n) = \begin{cases} \lfloor \log \mid n \mid \rfloor + 1, & \text{falls } n \neq 0 \\ 1, & \text{falls } n = 0. \end{cases}$$

Die logarithmischen Kosten $t(a)$ für die verschiedenen Formen eines Operanden a sind also

Operand a	Kosten $t(a)$
`=i`	$l(i)$
`i`	$l(i) + l(c(i))$
`*i`	$l(i) + l(c(i)) + l(c(c(i)))$.

Das logarithmische Kostenkriterium basiert auf der Annahme, daß die Kosten für die Ausführung einer Instruktion proportional sind zur Länge des Operanden. Die logarithmischen Kosten der einzelnen RAM Instruktionen sehen wie folgt aus (dabei sind $t(a)$ die Kosten des Operanden a):

Instruktion		Kosten
1.	`LOAD a`	$t(a)$
2.	`STORE i`	$l(c(0)) + l(i)$
	`STORE *i`	$l(c(0)) + l(i) + l(c(i))$
3.	`ADD a`	$l(c(0)) + t(a)$
4.	`SUB a`	$l(c(0)) + t(a)$
5.	`MULT a`	$l(c(0)) + t(a)$
6.	`DIV a`	$l(c(0)) + t(a)$
7.	`READ i`	$l(\text{input}) + l(i)$
	`READ *i`	$l(\text{input}) + l(i) + l(c(i))$
8.	`WRITE a`	$t(a)$

9.	`JUMP b`	1
10.	`JGTZ b`	$l(c(0))$
11.	`JZERO b`	$l(c(0))$
12.	`HALT`	1

Die *logarithmische Raumkomplexität* eines RAM Programms ist die Summe über alle verwendeten Register, Akkumulator eingeschlossen, von $l(x_i)$, wobei x_i die betragsmäßig größte in Register i gespeicherte Zahl ist.

Nach welchem Kostenkriterium man die Komplexität eines Algorithmus mißt hängt natürlich davon ab wie große Zahlen vom Algorithmus verarbeitet werden. Kann man annehmen, daß jede Zahl, die im Laufe einer Berechnung auftritt, in ein Computerwort paßt, so ist das uniforme Kostenkriterium angebracht. Andernfalls könnte das logarithmische Kostenkriterium eine realistischere Analyse ergeben.

Beispiel 2.2.1: Wir bestimmen die Zeit– und Raumkomplexität des RAM Programms aus Beispiel 1.3.1 zur Berechnung der Funktion

$$f(n) = \begin{cases} n^n & \text{für } n > 0, \\ 0 & \text{sonst.} \end{cases}$$

Die Zeitkomplexität des Programms ist beherrscht von der Schleife um die `MULT` Instruktion. Wenn die `MULT` Instruktion zum i–ten mal ausgeführt wird, enthält der Akkumulator n^i und Register 2 enthält n. $n-1$ solche `MULT` Intruktionen werden ausgeführt. Unter dem uniformen Kostenkriterium kostet jede `MULT` Intruktion eine Zeiteinheit, die Ausführung aller dieser `MULT` Instruktionen benötigt also $\mathcal{O}(n)$ Zeit. Unter dem logarithmischen Kostenkriterium kostet die i–te `MULT` Instruktion $l(n^i) + l(n) = (i+1)\log n$ Zeit, die Kosten für die Ausführung aller `MULT` Instruktionen betragen also

$$\textstyle\sum_{i=1}^{n-1}(i+1)\log n \ ,$$

das ist $\mathcal{O}(n^2 \log n)$. Alle anderen Operationen sind von nicht größerer Komplexität.

Die Raumkomplexität ist nur vom Inhalt der Register 0 bis 3 abhängig. Somit ist unter dem uniformen Kostenkriterium die Raumkomplexität einfach $\mathcal{O}(1)$. Unter dem logarithmischen Kostenkriterium ist die Raumkomplexität $\mathcal{O}(n \log n)$, da n^n die größte Zahl ist, die in einem dieser Register gespeichert wird, und $l(n^n) \simeq n \log n$.

	uniforme Kosten	logarithmische Kosten
Zeitkomplexität	$\mathcal{O}(n)$	$\mathcal{O}(n^2 \log n)$
Raumkomplexität	$\mathcal{O}(1)$	$\mathcal{O}(n \log n)$

Das uniforme Kostenkriterium ist nur für sehr kleine n realistisch. Paßt n^n nicht mehr in ein Computerwort, so ist auch das logarithmische Kostenkriterium unrealistisch. Es nimmt nämlich an, daß zwei Zahlen m und n in Zeit $\mathcal{O}(l(m) + l(n))$ miteinander multipliziert werden können. Es ist jedoch nicht bekannt, ob das möglich ist. ■

Beispiel 2.2.2: Für das RAM Programm in Beispiel 1.3.2 hängt die Komplexität in folgender Weise von der Länge n des Inputs ab:

	uniforme Kosten	logarithmische Kosten
Zeitkomplexität	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
Raumkomplexität	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$

Paßt n nicht mehr in ein Computerwort, so ist das logarithmische Kostenkriterium ziemlich realistisch. ■

Satz 2.2.1: *Sowohl unter dem uniformen als auch dem logarithmischen Kostenkriterium gibt es für jedes RAM Programm P mit Zeitkomplexität $T(n)$ eine Konstante k und ein äquivalentes RASP Programm P' mit Zeitkomplexität $kT(n)$.*

Beweis: Dazu betrachten wir den Teil (a) des Beweises von Satz 1.3.1. Im dort konstruierten RASP Programm P' kommen für jede RAM Instruktion höchstens 6 RASP Instruktionen vor. Unter dem uniformen Kostenkriterium ist also die Zeitkomplexität von P' höchstens $6T(n)$. Durch detailierte Untersuchung des Programms P' stellt man fest, daß die Behauptung auch für das logarithmische Kostenkriterium gilt mit $k = 6 + 11 \cdot l(r)$, wobei r der im Beweis von Satz 1.3.1 erstellte Verschiebungsfaktor ist. ■

Satz 2.2.2: *Sowohl unter dem uniformen als auch dem logarithmischen Kostenkriterium gibt es für jedes RASP Programm P' mit Zeitkomplexität $T(n)$ eine Konstante k und ein äquivalentes RAM Programm p mit Zeitkomplexität $kT(n)$.*

Beweis: Durch Analyse des Teils (b) des Beweises von Satz 1.3.1. ■

2.3. Komplexität von Turing–Maschinen

Definition 2.3.1: Die *(worst–case) Zeitkomplexität* $T(n)$ einer Turing–Maschine M ist die maximale Anzahl von Rechenschritten (Übergang von einer Konfiguration zur nächsten) zur Abarbeitung einer beliebigen Eingabe der Länge n. Gibt es ein Eingabewort der Länge n, für welches die Turing–Maschine nicht hält, so ist $T(n)$ undefiniert. Die *(worst–case) Raumkomplexität* $S(n)$ von M ist die maximale Distanz vom linken Bandende, welche einer der LS–Köpfe im Laufe der Abarbeitung eines Eingabewortes der Länge n zurücklegt. Gibt es eine Eingabe der Länge n für welche sich einer der LS–Köpfe unbegrenzt nach rechts bewegt, dann ist $S(n)$ undefiniert.

Eine nichtdeterministische Turing–Maschine M hat die *Zeitkomplexität* $T(n)$, falls bei einer Eingabe der Länge n keine Folge von nichtdeterministischen Wahlen existiert, welche eine Berechnung länger als $T(n)$ bedingt. Eine nichtdeterministische Turing–Maschine M hat die *Raumkomplexität* $S(n)$, falls bei einer Eingabe der Länge n keine Folge von nichtdeterministischen Wahlen existiert, welche M auf einem der Bänder mehr als $S(n)$ Zellen bearbeiten läßt. Vgl. [HU], p.288. ■

Beispiel 2.3.1: Wir bestimmen die Komplexität der Turing–Maschine M_1 aus Beispiel 1.4.1 zur Erkennung der Sprache $L = \{0^m 1^m | m \in \mathbb{N}\}$. Das Eingabewort w habe die Länge n. In der Hauptberechnungsschleife werden jeweils 2 Symbole von w ausgekreuzt (durch X bzw. Y ersetzt). Die Schleife kann also höchstens $(n+1)/2$ mal ausgeführt werden. Die Anzahl der Schritte für einen Schleifendurchlauf setzt sich zusammen aus den Schritten für die Suche nach rechts (höchstens $n+1$ Schritte), der Bewegung nach links (höchstens n Schritte) und der Bewegung zum nächsten Symbol (1 Schritt), sie kann also höchstens $2n+2$ betragen. Somit werden zur Abarbeitung aller Schleifendurchläufe höchstens $(2n+2)\cdot(n+1)/2 = (n+1)^2$ Schritte benötigt. Die Schlußbehandlung kann in n Schritten durchgeführt werden. Die Zeitkomplexität von M_1 ist also $\mathcal{O}(n^2)$.

Der LS–Kopf bewegt sich höchstens um eine Position über das Eingabewort hinaus, die Raumkomplexität von M_1 ist also $\mathcal{O}(n)$. ■

Definition 2.3.2: Zwei zahlentheoretische Funktionen $f_1(n)$ und $f_2(n)$ (also $f_1, f_2 : \mathbb{N} \to \mathbb{N}$) sind *in polynomialer Relation*, wenn es Polynome $p_1(x)$ und $p_2(x)$ mit ganzzahligen Koeffizienten gibt, sodaß für alle natürlichen Zahlen n gilt $f_1(n) \leq p_1(f_2(n))$ und $f_2(n) \leq p_2(f_1(n))$. ■

Beispiel 2.3.2: Die Funktionen $f_1(n) = 2n^2$ und $f_2(n) = n^5$ sind in polynomialer Relation; dazu können wir $p_1(x) = 2x$ und $p_2(x) = x^3$ wählen, da $2n^2 \leq 2n^5$ und $n^5 \leq (2n^2)^3$.

a^n und b^n sind in polynomialer Relation für $a, b > 1$. Für $p(x) = x^{\lceil \log_b a \rceil}$ gilt $a^n = (b^{\log_b a})^n = (b^n)^{\log_b a} \leq p(b^n)$.

Die Funktionen n^2 und 2^n sind jedoch nicht in polynomialer Relation, da es kein Polynom $p(x)$ gibt, für welches gelten würde $p(n^2) \geq 2^n$ für alle n. ■

Satz 2.3.1: Das RAM/RASP Modell unter dem logarithmischen Kostenkriterium und das Turing–Maschinen Modell sind in polynomialer Relation; d.h. die Komplexitätsfunktionen der beiden Modelle sind in polynomialer Relation. ■

Der Beweis des Satzes 2.3.1 ergibt sich aus einer Analyse der Simulation einer RAM durch eine Turing–Maschine und umgekehrt. Unter dem uniformen Kostenkriterium kann jede Turing–Maschine mit Komplexität $\mathcal{O}(T(n))$ durch ein RAM Programm mit Komplexität $\mathcal{O}(T^2(n))$ simuliert werden. Die Umkehrung gilt jedoch nicht. Unter dem uniformen Kostenkriterium kann nämlich eine RAM in $\mathcal{O}(n)$ Schritten die Zahl 2^{2^n} berechnen ($2^{2^n} = 2^{2^{n-1} \cdot 2} = (2^{2^{n-1}})^2$). Eine Turing–Maschine braucht bereits 2^n Zellen nur um diese Zahl zu speichern.

3. Effiziente Algorithmen und ihre Komplexität

3.1. Multiplikation ganzer Zahlen

Der "klassische" Algorithmus zur Multiplikation zweier ganzer Zahlen x und y hat die Komplexität $l(x) \cdot l(y)$, da im wesentlichen jede Ziffer von x mit jeder Ziffer von y multipliziert werden muß. Um zwei Zahlen der Länge n zu multiplizieren, braucht man also $\mathcal{O}(n^2)$ Zeit.

Im Jahre 1962 hatten A. Karatsuba und Yu. Ofman [1)] eine Idee, wie man einen Multiplikationsalgorithmus mit besserem Komplexitätsverhalten konstruieren könnte. Sei β der Radix der Zahldarstellung. Der Einfachheit halber nehmen wir an, die Länge n der beiden Multiplikanden x und y wäre eine Potenz von 2. Wir zerschlagen die Zahlen jeweils in zwei Teile

$$\begin{aligned} x &= a \cdot \beta^{n/2} + b, \\ y &= c \cdot \beta^{n/2} + d, \end{aligned} \tag{3.1.1}$$

wobei jeder dieser Teile die Länge $n/2$ hat. Das Produkt ergibt sich dann als

$$\begin{aligned} z = x \cdot y &= ac \cdot \beta^n + (ad + bc) \cdot \beta^{n/2} + bd \\ &= ac \cdot \beta^n + \big((a+b)(c+d) - ac - bd\big) \cdot \beta^{n/2} + bd. \end{aligned} \tag{3.1.2}$$

Zur Berechnung von xy genügen also drei Multiplikationen $(n/2)$–stelliger Zahlen und einige Additionen und Verschiebungen um Potenzen von β. Diese Operationen brauchen nur $\mathcal{O}(n)$ Zeit. Wir erhalten das Programm

$$\begin{array}{l} \textbf{begin} \\ \quad u \leftarrow (a+b) * (c+d); \\ \quad v \leftarrow a * c; \\ \quad w \leftarrow b * d; \\ \quad z \leftarrow v * \beta^n + (u - v - w) * \beta^{n/2} + w \\ \textbf{end} \end{array} \tag{3.1.3}$$

Zur Berechnung der Produkte u, v und w wird die Multiplikationsprozedur rekursiv angewandt.

Wir nehmen bis auf weiteres an, daß $a + b$ und $c + d$ jeweils $(n/2)$–stellige Zahlen wären und ignorieren somit den möglichen Übertrag, der bei der Addition von a und

---

1) A. Karatsuba und Yu. Ofman: *Dokl. Akad. Nauk SSSR* 145, 293–294 (1962) [englische Übersetzung: *Multiplication of Multidigit Numbers on Automata, Soviet Phys. Dokl.* 7, 595–596 (1963)].

b entstehen und zu einer $(n/2\ +1)$–stelligen Zahl führen kann. Die Zeit für die Multiplikation zweier n–stelliger Zahlen ist also nach oben beschränkt durch

$$T(n) = \begin{cases} k, & \text{für } n = 1 \\ 3T(n/2) + kn, & \text{für } n > 1, \end{cases} \tag{3.1.4}$$

wobei kn eine obere Schranke für die Additionen und Verschiebungen in (3.1.3) ist. Als Lösung von (3.1.4) erhalten wir

$$T(n) = 3k\,n^{\log 3} - 2kn, \tag{3.1.5}$$

was durch Induktion leicht zu beweisen ist [2]:

> die Basis, für $n = 1 = 2^0$, ist trivial. Erfüllt $T(\overline{n}) = 3k\,\overline{n}^{\log 3} - 2k\overline{n}$ die Beziehung (3.1.4), so gilt auch
>
> $$\begin{aligned} T(2\overline{n}) &= 3T(\overline{n}) + 2k\overline{n} \\ &= 3(3k\,\overline{n}^{\log 3} - 2k\overline{n}) + 2k\overline{n} \\ &= 3k \cdot 2^{\log 3} \cdot \overline{n}^{\log 3} - 6k\overline{n} + 2k\overline{n} \\ &= 3k(2\overline{n})^{\log 3} - 2k(2\overline{n}). \end{aligned}$$

Wir kehren zurück zum Problem, daß $a+b$ und $c+d$ eventuell die Länge $n/2\ +1$ haben könnten. In diesem Fall schreiben wir $a+b$ als $a_1\beta^{n/2}+b_1$, wobei a_1 das führende Bit von $a+b$ ist und b_1 die restlichen Bits. Ebenso schreiben wir $c+d$ als $c_1\beta^{n/2}+d_1$. Das Produkt $(a+b)(c+d)$ ergibt sich nun als

$$a_1c_1 \cdot \beta^n + (a_1d_1 + b_1c_1) \cdot \beta^{n/2} + b_1d_1. \tag{3.1.6}$$

Der Term b_1d_1 wird durch eine rekursive Anwendung der Multiplikationsprozedur auf Zahlen der Länge $n/2$ berechnet. Die übrigen Multiplikationen in (3.1.6) sind entweder Multiplikationen mit einem einzigen Bit oder Verschiebungen, haben also die Komplexität $\mathcal{O}(n)$.

Somit ist die Zeitkomplexität des Karatsuba Algorithmus $\mathcal{O}(n^{\log 3}) \simeq \mathcal{O}(n^{1.59})$. Bisher haben wir nur Potenzen von 2 als Längen n zugelassen. Ist n keine Potenz von 2, so erhöhen wir die Länge (durch Hinzufügen führender Nullen) zur nächsten Potenz von 2. Dabei wird die Länge höchstens verdoppelt. Der Faktor 2 kann aber in der asymptotischen Komplexität vernachlässigt werden, denn $\mathcal{O}((2n)^{\log 3}) = \mathcal{O}(n^{\log 3})$.

---

[2] log ist dabei der Logarithmus zur Basis 2.

Satz 3.1.1 (Karatsuba): *Zwei Zahlen der Länge n können in Zeit $\mathcal{O}(n^{\log 3})$ multipliziert werden.* ■

Beispiel 3.1.1: Wir betrachten das folgende Zahlenbeispiel für den Karatsuba Algorithmus. Der Radix β der Zahlendarstellung sei dabei 10.

$$\begin{array}{lll} x = 3141 & a = 31 & b = 41 \\ y = 5927 & c = 59 & d = 27 \end{array}$$

$$\begin{aligned} u &= (a+b)(c+d) = 72 \times 86 = 6192 \\ v &= ac = 31 \times 59 = 1829 \\ w &= bd = 41 \times 27 = 1107 \\ xy &= 1829 \times 10000 + (6192 - 1829 - 1107) \times 100 + 1107 \\ &= 18616707 \end{aligned}$$

■

Der Karatsuba Algorithmus ist ein Beispiel für einen "divide–and–conquer" Algorithmus. Dabei wird ein Problem dadurch gelöst, daß es in mehrere Teilprobleme geringerer Problemgröße zerlegt wird. Die Zeitkomplexität eines solchen Algorithmus ist bestimmt von der Anzahl und Größe der Teilprobleme und zu einem geringeren Ausmaß vom Arbeitsaufwand für die Zerlegung des Problems. Rekursionsrelationen der Art (3.1.4) treten bei der Analyse rekursiver "divide–and–conquer" Algorithmen häufig auf. Der folgende Satz gibt Auskunft über die Lösung solcher Rekursionsrelationen. Dabei betrachten wir den Fall, daß ein Problem der Größe n zerlegt wird in a Probleme der Größe n/c. Der Aufwand für die Zerlegung und anschließende Kombination der Teilergebnisse sei $b \cdot n$.

Satz 3.1.2: *Seien a, b und c natürliche Zahlen. Die Lösung der Rekursionsrelation*

$$T(n) = \begin{cases} b, & \text{für } n = 1, \\ aT(n/c) + bn, & \text{für } n > 1 \end{cases}$$

für eine Potenz n von c ist

$$T(n) = \begin{cases} \mathcal{O}(n), & \text{falls } a < c, \\ \mathcal{O}(n \log n), & \text{falls } a = c, \\ \mathcal{O}(n^{\log_c a}), & \text{falls } a > c. \end{cases}$$

Beweis: Wir zeigen zunächst für alle Potenzen n von c

$$T(n) = bn \sum_{i=0}^{\log_c n} r^i, \qquad \text{wobei } r = a/c.$$

Die Basis, für $n = c^0 = 1$, ist klar. Erfüllt nun $T(\overline{n}) = b\overline{n} \sum_{i=0}^{\log_c \overline{n}} r^i$ die Rekursionsrelation, so gilt auch

$$\begin{aligned} T(c\overline{n}) &= aT(\overline{n}) + bc\overline{n} \\ &= a\big(b\overline{n} \textstyle\sum_{i=0}^{\log_c \overline{n}} r^i\big) + bc\overline{n} \\ &= b(c\overline{n})\big(\tfrac{a}{c} \cdot \textstyle\sum_{i=0}^{\log_c \overline{n}} r^i \ + 1\big) \\ &= b(c\overline{n}) \cdot \textstyle\sum_{i=0}^{\log_c (c\overline{n})} r^i. \end{aligned}$$

Ist $a < c$, dann konvergiert die geometrische Reihe $\sum_{i=0}^{\infty} r^i$, also $T(n)$ ist $\mathcal{O}(n)$.

Ist $a = c$, so ist jeder der $\mathcal{O}(\log n)$ Summanden in $T(n)$ 1, also $T(n)$ ist $\mathcal{O}(n \log n)$.

Ist $a > c$, dann ist die partielle Summe der geometrischen Reihe

$$T(n) \;=\; bn \cdot \frac{r^{1+\log_c n} - 1}{r-1} \;\leq\; bn \cdot \frac{r}{r-1} \cdot r^{\log_c n} \;\leq\; 2bn \cdot \frac{a^{\log_c n}}{\underbrace{c^{\log_c n}}_{=n^{\log_c c}=n}} \;=\; 2b \cdot a^{\log_c n},$$

also $\mathcal{O}(a^{\log_c n})$, oder anders ausgedrückt $\mathcal{O}(n^{\log_c a})$. Dabei haben wir das allgemeine Gesetz $u^{\log_s v} = v^{\log_s u}$ verwendet. ∎

Ist die Problemgröße n keine Potenz von c, so kann man gewöhnlich ein Problem der Größe n einbetten in ein Problem der Größe n', wobei n' die nächst größere Potenz von c ist. Die asymptotische Wachstumsrate aus Satz 2.1.3 gilt also für beliebiges n.

Aus Satz 3.1.2 sehen wir, daß man durch die Zerlegung eines Problems (bei linearem Aufwand für die Zerlegung) in zwei Unterprobleme der halben Größe einen $\mathcal{O}(n \log n)$ Algorithmus erhält. Ist die Anzahl der Unterprobleme 3, 4 oder 8, dann ist der Algorithmus von der Ordnung $n^{\log 3}$, n^2 oder n^3. Zerlegt man das Problem in 4, 9 oder 16 Unterprobleme der Größe $n/4$, so erhält man einen Algorithmus der Ordnung $n \log n$, $n^{\log 3}$ oder n^2. Man könnte also einen asymptotisch schnelleren Muliplikationsalgorithmus dadurch erhalten, daß man die Multiplikanden in 4 Teile zerlegt und das Produkt ausdrückt in Termini von 8 oder weniger Produkten der Teile. D.E. Knuth [3)] beweist

[3)] D.E. Knuth: *The Art of Computer Programming, Vol. 2,* Addison–Wesley, Reading, Massachusetts (1969).

Satz 3.1.3: *Für jede positive reelle Zahl ϵ gibt es einen Multiplikationsalgorithmus für natürliche Zahlen, welcher zur Multiplikation zweier Zahlen der Länge $\leq n$ Zeit $\mathcal{O}(n^{1+\epsilon})$ braucht.* ■

Den bisher bekannten Algorithmus mit der besten asymptotischen Komplexität erhält man durch Verwendung der schnellen Fourier Transformation. Es ist dies der Algorithmus von Schönhage und Strassen; seine Komplexität ist $\mathcal{O}(n \log n \log\log n)$. Wir verweisen dazu auf [AHU], [Lip] und [Sed].

Jede Verbesserung der Komplexität der Multiplikation zieht auch eine Komplexitätsverbesserung anderer Operationen auf den ganzen Zahlen nach sich. Sei $M(n)$ die Zeit für die Multiplikation zweier n–stelliger Zahlen, $D(n)$ die Zeit um eine $2n$–stellige Zahl durch eine n–stellige Zahl zu dividieren, so gilt der folgende Satz.

Satz 3.1.4: *Die Komplexitätsfunktionen $M(n)$ und $D(n)$ unterscheiden sich nur um einen konstanten Faktor.*

Beweis: [AHU] Theorem 8.5. ■

3.2. Multiplikation von Polynomen

Die “klassische Methode zur Multiplikation von Polynomen $p(x) = \sum_{i=0}^{m} p_i x^i$ und $q(x) = \sum_{j=0}^{n} q_j x^j$ über einem Ring R benutzt die Formel

$$r(x) = p(x) \cdot q(x) = \sum_{l=0}^{m+n} (\sum_{i+j=l} p_i \cdot q_j) x^l.$$

Sind p und q Polynome in ν Variablen in dichter Darstellungsweise, so ist der Zeitaufwand für die Multiplikation von p und q abschätzbar durch

$$M(p_0, q_o) \cdot \prod_{i=1}^{\nu} (1 + \deg_i p)(1 + \deg_i q),$$

wobei p_0 und q_0 die größten Koeffizienten in p und q sind und $M(r, s)$ eine Komplexitätsabschätzung für die Multiplikation von Elementen r, s in R ist. Ist etwa $M(r, s)$ von der Ordnung 1, so ist die Zeit für die Multiplikation zweier Polynome vom Grad $\leq n$ in jeder der ν Variablen $\mathcal{O}(n^{2\nu}) = \mathcal{O}((n^{\nu})^2)$.

Für den Fall dichter Polynome ist es auch sinnvoll, die Karatsuba Methode zur Multiplikation anzuwenden. Dazu zerlegen wir

$$p(x) = p_1(x) \cdot x^{\lceil n/2 \rceil} + p_0(x) \quad \text{und} \quad q(x) = q_1(x) \cdot x^{\lceil n/2 \rceil} + q_0(x).$$

Es gilt nun

$$r(x) = p(x) \cdot q(x) = p_1 q_1 x^{2\lceil n/2 \rceil} + \big((p_1 + p_0)(q_1 + q_0) - p_1 q_1 - p_0 q_0\big) x^{\lceil n/2 \rceil} + p_0 q_0.$$

Man kommt also mit drei Teilmultiplikationen aus, wobei die Multiplikanden jeweils vom Grad $\leq \lceil n/2 \rceil$ sind. Daraus ergibt sich (wiederum unter der Annahme $M(r, s) = \mathcal{O}(1)$) die Zeilkomplexität $\mathcal{O}(n^{\nu \cdot \log_2 3}) = \mathcal{O}((n^{\nu})^{\log_2 3})$ für die Multiplikation zweier Polynome mit Grad $\leq n$ in jeder der ν Variablen.

3.3. Evaluierung von Polynomen

Es sei das Polynom $p(x) = \sum_{i=0}^{n} p_i x^i$ in der Variablen x über dem Ring R und ein Element x_0 aus R gegeben. Gesucht ist der Wert von p an der Stelle x_0, also $p(x_0)$.

Man könnte nun sukzessive p_0, $p_1 x, \ldots, p_n x^n$ berechnen und bräuchte auf diese Weise $2n-1$ Multiplikationen und n Additionen im Ring R zur Berechnung von $p(x_0)$. Eine wesentliche Verbesserung der Komplexität läßt sich jedoch mit der *Hornerschen Regel* erreichen. Dabei wird $p(x_0)$ nach dem Schema

$$p(x_0) = (\ldots(p_n x_0 + p_{n-1}) \cdot x_0 + \ldots) \cdot x_0 + p_0$$

berechnet. Dazu sind n Multiplikationen und n Additionen in R notwendig. Die Hornersche Regel ergibt sich aus der Division von $p(x)$ durch $x - a$.

Zerlegt man das Polynom $p(x)$ in der Art

$$p(x) = \underbrace{\sum_{j=0}^{\lfloor n/2 \rfloor} a_j x^{2j}}_{p'} + \underbrace{\sum_{k=0}^{\lceil n/2 \rceil - 1} b_k x^{2k+1}}_{p''},$$

so ist

$$\begin{aligned} p'(x_0) &= (\ldots(a_{\lfloor n/2 \rfloor} x_0^2 + a_{\lfloor n/2 \rfloor - 1}) x_0^2 + \ldots) x_0^2 + a_0 \qquad \text{und} \\ p''(x_0) &= ((\ldots(b_{\lceil n/2 \rceil - 1} x_0^2 + b_{\lceil n/2 \rceil - 2}) x_0^2 + \ldots) x_0^2 + b_0) x_0. \end{aligned}$$

Diese Methode nennt man die *Hornersche Regel zweiter Ordnung*. Um nun $p(x_0)$ auf diese Weise zu berechnen, braucht man $1 + \lfloor n/2 \rfloor + \lceil n/2 \rceil = n+1$ Multiplikationen und $\lfloor n/2 \rfloor + (\lceil n/2 \rceil - 1) + 1 = n$ Additionen in R. Das ist zwar keine Verbesserung gegenüber der Hornerschen Regel erster Ordnung, hat man aber sowohl $p(x_0)$ als auch $p(-x_0)$ zu berechnen, so kann man das durch eine einzige zusätzliche Addition erreichen. Die Hornersche Regel zweiter Ordnung ergibt sich aus der Division von $p(x)$ durch $x^2 - a^2$.

4. Entscheidbarkeit — Unentscheidbarkeit

4.1. Akzeptierungsproblem und Halteproblem für Turing–Maschinen

Unter einem *(Entscheidungs–) Problem* verstehen wir eine Frage der Art *“Ist die natürliche Zahl n ein Quadrat?”* Für eine Festlegung des Parameters n ist die Antwort entweder *“ja”* oder *“nein”*. Eine *Instanz* eines Problems ist eine Liste von Argumenten, ein Argument für jeden Parameter des Problems. So ist etwa (15) eine Instanz des obigen Problems.

Einem Problem P mit Parametern $p_1, \ldots, p_k$ kann auf offensichtliche Weise eine Sprache L_P wie folgt zugeordnet werden:

$$L_P = \{(n_1, \ldots, n_k) \mid \text{die Antwort auf die Instanz } (n_1, \ldots, n_k) \text{ des Problems } P \text{ ist "ja"}\}.$$

Definition 4.1.1: Ein Problem P, dessen Sprache L_P rekursiv ist, heißt *entscheidbar*. Andernfalls heißt P *unentscheidbar*. Ist L_P r.a., so sagt man auch P wäre *semi–entscheidbar*. ■

Definition 4.1.2: Zunächst bringen wir jede Turing–Maschine in eine gewisse kanonische Form, man spricht dabei von der *Kodierung* einer Turing–Maschine. Sei M eine Turing–Maschine mit Zustandsmenge $\{q_1, \ldots, q_n\}$, Eingabealphabet $\{0, 1\}$, Bandalphabet $\{0, 1, \sqcup\}$, Anfangszustand q_1, Endzustandsmenge $F = \{q_2\}$ und Überführungsfunktion δ. Das stellt keine Einschränkung dar, da jedes Alphabet Σ in $\{0, 1\}$ kodiert werden kann. Wir bezeichnen die Bewegungsrichtungen “links”, “rechts” mit D_1, D_2 (der Einfachheit halber lassen wir nicht zu, daß der LS–Kopf stationär bleibt; diese Situation kann aber immer durch eine Folge von rechts–links Bewegungen simuliert werden), und die Symbole 0,1,$\sqcup$ mit X_1, X_2, X_3. Eine allgemeine Operation von M hat also die Gestalt

$$\delta(q_i, X_j) = (q_k, X_l, D_m).$$

Diese Operation wird kodiert als

$$0^i 1 0^j 1 0^k 1 0^l 1 0^m. \tag{4.1.1}$$

Die Turing–Maschine M wird kodiert als

$$111\ code_1\ 11\ code_2\ 11\ \ldots\ 11\ code_r\ 111, \tag{4.1.2}$$

wobei jeder $code_i$ die Form (4.1.1) hat und jede Operation von M von einem $code_i$ kodiert ist. Die (bis auf Umordnung der $code_i$'s eindeutig bestimmte) Kodierung von M wird mit $\langle M \rangle$ bezeichnet. ■

Beispiel 4.1.1: Als Beispiel einer solchen Kodierung betrachten wir die Turing–Maschine M mit $Q = \{q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, $\Gamma = \{0, 1, \sqcup\}$, $F = \{q_2\}$, und $\delta(q_1, 1) = (q_3, 0, R)$, $\delta(q_3, 0) = (q_1, 1, R)$, $\delta(q_3, 1) = (q_2, 0, R)$, $\delta(q_3, \sqcup) = (q_3, 1, L)$. Eine mögliche Kodierung von M (Version von $\langle M \rangle$) ist

111010010001010011000101010010011000100100101001100010001000100101111. ■

Es ist also klar, daß die (Kodes von) Turing–Maschinen in einer Reihe $M_1, M_2, M_3, \ldots$ angeordnet werden können (etwa der Länge nach und alphabetisch bei gleicher Länge). Ebenso können die Wörter in $\{0,1\}^*$ in einer Reihe $w_1, w_2, w_3, \ldots$ angeordnet werden. Für $i, j \in \mathbf{N}$ definieren wir nun

$$a_{ij} = \begin{cases} 0, & \text{falls } w_i \in L(M_j), \\ 1, & \text{sonst.} \end{cases}$$

Auf diese Weise erhalten wir eine nach unten und rechts unendliche Matrix

Wörter	Turing-Maschine			
	M_1	M_2	M_3	$\cdots$
w_1	$\mathbf{a_{11}}$	a_{12}	a_{13}	$\cdots$
w_2	a_{21}	$\mathbf{a_{22}}$	a_{23}	$\cdots$
w_3	a_{31}	a_{32}	$\mathbf{a_{33}}$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Definition 4.1.3: Wir benutzen die Diagonalelemente in dieser Matrix zur Definition der *Diagonalsprache*

$$L_d = \{\ w_j \ \mid\ a_{jj} = 1\ \}.$$

Also $w_j \in L_d \Longleftrightarrow w_j \notin L(M_j)$. ■

Satz 4.1.1: *Die Diagonalsprache L_d ist nicht r.a.*

Beweis: Angenommen es gäbe eine Turing–Maschine M_j, welche L_d akzeptiert. Daraus würde sich folgende Kontradiktion ergeben: falls $w_j \in L_d$, dann $a_{jj} = 1$, also $w_j \notin$

$L(M_j)$. Andererseits, falls $w_j \notin L_d$, dann $a_{jj} = 0$, also $w_j \in L(M_j)$. $L(M_j)$ wäre also verschieden von L_d. ∎

Definition 4.1.4: Das *Akzeptierungsproblem für Turing–Maschinen* ist das Problem, für eine gegebene Turing–Maschine M (mit Bandalphabet $\{0, 1, \sqcup\}$) und ein gegebenes Wort $w \in \{0, 1\}^*$ zu entscheiden, ob M das Wort w akzeptiert. Die zum Akzeptierungsproblem gehörige Sprache heißt die *universelle Sprache*. Sie ist definiert als

$$L_u = \{\ \langle M \rangle w \ \mid\ M \text{ akzeptiert } w\ \}.$$ ∎

L_u ist universell in dem Sinn, daß die Frage ob eine Folge w in $\{0, 1\}^*$ von einer Turing–Maschine M (o.B.d.A. mit Bandalphabet $\{0, 1, \sqcup\}$) akzeptiert wird äquivalent ist zur Frage ob $\langle M \rangle w \in L_u$.

Satz 4.1.2: L_u *ist r.a.*

Beweis: Wir konstruieren eine Turing–Maschine mit 3 Bändern M_u^3, welche L_u akzeptiert. Das erste Band von M_u^3 ist das Eingabeband und enthält also eine Instanz $\langle M \rangle w$ des Akzeptierungsproblems. Die Operationen der Turing–Maschine M sind zwischen den ersten zwei Blöcken von drei 1en auf dem ersten Band kodiert. Das zweite Band von M_u^3 simuliert das Band von M. Auf dem dritten Band von M_u^3 wird der Zustand q_i von M als 0^i abgespeichert. Die Turing–Maschine M_u^3 funktioniert folgendermaßen:

(1) Überprüfe den Inhalt von Band 1 darauf hin, ob er mit einem Wort der Form (4.1.2) beginnt und es keine zwei Operationskodes gibt, welche mit 0^i10^j1 für dasselbe i und j beginnen. Weiters prüfe für jeden Operationskode $0^i10^j10^k10^l10^m$ ob $1 \leq j \leq 3$, $1 \leq l \leq 3$ und $1 \leq m \leq 2$. Bei diesen Berechnungen kann Band 3 als Arbeitsband benutzt werden.

(2) Schreibe w, den Teil der Eingabe nach dem zweiten Block von drei 1en, auf Band 2. Schreibe 0 (die Kodierung für den Zustand q_1) auf Band 3. Alle LS–Köpfe werden am linken Bandende positioniert. Diese Symbole können markiert werden, sodaß sie leicht wieder aufgefunden werden können.

(3) Enthält Band 3 das Wort 00, den Kode für den Endzustand q_2, so halte und akzeptiere.

(4) Sei X_j das aktuelle Symbol (auf welches der LS–Kopf zeigt) von Band 2 und sei 0^i der Inhalt von Band 3. Suche auf Band 1 von links bis zum zweiten Block

von drei 1en nach einem Unterwort mit dem Präfix 110^i10^j1. Wird ein solches Wort nicht gefunden, so halte und weise die Eingabe zurück; M kann keinen nächsten Schritt ausführen und hat nicht akzeptiert. Wird jedoch ein Kode der Form $0^i10^j10^k10^l10^m$ gefunden, so schreibe 0^k auf Band 3, schreibe X_l auf die aktuelle Bandzelle in Band 2 und bewege den LS–Kopf 2 in die Richtung D_m. Fahre fort mit Schritt (3).

Offensichtlich akzeptiert M_u^3 das Eingabewort $\langle M\rangle w$ genau dann, wenn M das Wort w akzeptiert. Hält M nicht auf w oder hält es ohne zu akzeptieren, so tut M_u^3 dasselbe auf $\langle M\rangle w$. ∎

Definition 4.1.5: Da L_u r.a. ist, gibt es eine Turing–Maschine M_u (mit einem Band), welche L_u akzeptiert. Diese (allerdings nicht eindeutig bestimmte) Turing–Maschine M_u heißt die *universelle Turing–Maschine*, da sie die Arbeit jeder Turing–Maschine verrichtet. ∎

Satz 4.1.3: *L_u ist nicht rekursiv, das Akzeptierungsproblem für Turing–Maschinen ist also unentscheidbar.*

Beweis: Wir nehmen an, es gäbe eine Turing–Maschine M, welche L_u entscheidet (d.h. M stoppt auf jeder Eingabe und akzeptiert L_u). Dann könnten wir L_d wie folgt entscheiden: für eine gegebene Eingabe w bestimmen wir zunächst den Index i, für welchen gilt $w = w_i$. Aus i berechnen wir die Turing–Maschine M_i. Die Folge $\langle M_i\rangle w_i$ geben wir nun als Eingabe an M und akzeptieren w genau dann, wenn M die Folge $\langle M_i\rangle w_i$ nicht akzeptiert, also M_i das Wort w_i nicht akzeptiert. Da wir aber aus (1) wissen, daß L_d nicht rekursiv ist, kann es keinen solchen Entscheidungsalgorithmus geben, also auch keine Turing–Maschine M, die L_u entscheidet. ∎

Ein scheinbar einfacher zu behandelndes Problem ist das *Halteproblem "stoppt (hält) die Berechnung der Turing–Maschine M auf dem Eingabewort w?"* Man sieht aber leicht, daß auch das Halteproblem unentscheidbar sein muß, da man andernfalls sofort einen Algorithmus zur Entscheidung des Akzeptierungsproblems konstruieren könnte. Bei genauerer Betrachtung stellt sich sogar heraus, daß es nicht einmal einen Algorithmus gibt, der von einer Turing–Maschine entscheiden könnte, ob sie auf der leeren Eingabe stoppt.

Def. 4.1.6: Mit der selben Bezeichnung wie in Def. 4.1.4 seien

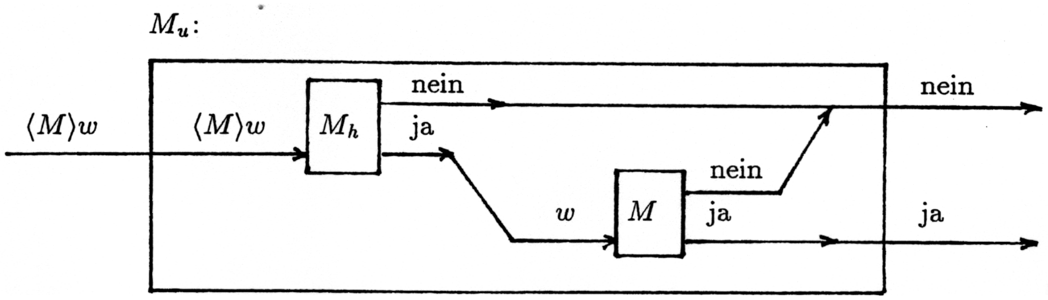
$$L_h = \{ \langle M \rangle w \mid M \text{ stoppt bei Eingabe } w \},$$

$$L_{h,\epsilon} = \{ \langle M \rangle \mid M \text{ stoppt bei Eingabe } \epsilon \}.$$

L_h ist die zum *Halteproblem* gehörige Sprache, $L_{h,\epsilon}$ die zum *eingeschränkten Halteproblem* gehörige Sprache. ■

Satz 4.1.4: *L_h ist nicht rekursiv, also das Halteproblem für Turing–Maschinen ist unentscheidbar.*

Beweis: Wäre L_h rekursiv, so wäre auch L_u rekursiv. Wäre nämlich M_h eine immer terminierende Turingmaschine, welche L_h akzeptiert, so wäre M_u eine immer terminierende universelle Turingmaschine:



■

Auch das eingeschränkte Halteproblem $L_{h,\epsilon}$ ist unentscheidbar, wir können das aber erst später beweisen.

Die üblichen Programmiersprachen, wie FORTRAN, Pascal, etc. sind universell, d.h. sie sind in der Lage, jede Turing–Maschine zu simulieren und umgekehrt. Es kann also z.B. auch kein Pascal–Programm geben, welches von einem beliebig vorgegebenen Pascal–Programm entscheiden könnte, ob dieses bei der Eingabe 0 stoppt.

4.2. Der Satz von Rice

Wir haben festgestellt, daß es unmöglich ist von einer Turing–Maschine M und einem Wort w zu entscheiden, ob M das Wort w akzeptiert, ja es ist nicht einmal möglich zu entscheiden, ob M auf der leeren Eingabe stoppt! Es erhebt sich also die Frage: Welche Eigenschaften von Turing–Maschinen sind denn nun entscheidbar? Oder mit anderen Worten: Welche Eigenschaften von r.a. Sprachen sind entscheidbar?

Zunächst definieren wir, was wir unter einer Eigenschaft r.a. Sprachen verstehen wollen.

Definition 4.2.1: Wir nennen jede Menge $\mathcal{S}$ rekursiv aufzählbarer Sprachen (über $\{0,1\}$) eine *Eigenschaft* r.a. Sprachen. $\mathcal{S}$ ist eine *triviale* Eigenschaft, wenn $\mathcal{S}$ leer ist oder alle r.a. Sprachen enthält. Die Eigenschaft $\mathcal{S}$ heißt *entscheidbar*, wenn die Sprache

$$L_{\mathcal{S}} = \{\ \langle M\rangle \ \mid \ L(M) \in \mathcal{S}\ \}$$

rekursiv ist, wenn es also möglich ist von jeder Turing–Maschine zu entscheiden, ob die von ihr akzeptierte Sprache in $\mathcal{S}$ ist. ∎

Satz 4.2.1 (Satz von Rice [1]): *Jede nichttriviale Eigenschaft r.a. Sprachen ist unentscheidbar.*

Beweis: Sei $\mathcal{S}$ eine nichttriviale Eigenschaft r.a. Sprachen. Wir können annehmen, daß $\emptyset \notin \mathcal{S}$, denn andernfalls könnten wir anstatt $\mathcal{S}$ sein Komplement $\overline{\mathcal{S}}$ betrachten. Sei nun L eine Sprache in $\mathcal{S}$ und M_L eine Turing–Maschine , die L akzeptiert. Wir konstruieren nun eine Turing–Maschine A, welche den Kode $\langle M\rangle$ einer Turing–Maschine und ein Wort w als Eingabe nimmt und den Kode $\langle M'\rangle$ einer anderen Turing–Maschine produziert, sodaß

$$L(M') \in \mathcal{S} \Longleftrightarrow \langle M\rangle w \in L_u \quad (M \text{ akzeptiert } w).$$

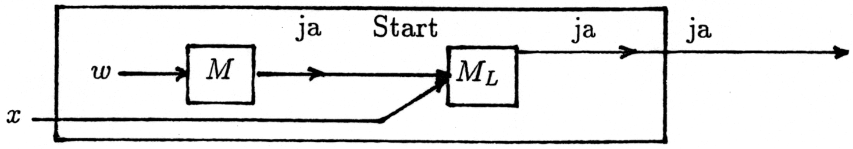
Konstruktion von A:

A produziert den Kode einer Turing–Maschine M' mit folgender Eigenschaft: zuerst ignoriert M' seine Eingabe und simuliert M auf w. Wird w von M nicht

[1] H.G. Rice: *Classes of recursively enumerable sets and their decision problems*, *Trans. Amer. Math. Soc.* 89, 25–59 (1953).

akzeptiert, so akzeptiert auch M' sein Eingabewort x nicht. Andernfalls simuliert M' das Verhalten von M_L auf x und akzeptiert x genau dann, wenn M_L x akzeptiert.

M':



Also

$$L(M') = \begin{cases} \emptyset, & \text{falls } w \notin L(M), \\ L, & \text{sonst.} \end{cases}$$

Wäre nun $\mathcal{S}$ entscheidbar, so gäbe es eine Turing–Maschine $M_{\mathcal{S}}$, welche $L_{\mathcal{S}}$ entscheidet. Das aber hätte zur Folge, daß wir einen Algorithmus zur Entscheidung von L_u konstruieren könnten:

1. Wende A auf $\langle M \rangle w$ an und erzeuge $\langle M' \rangle$.
2. Wende $M_{\mathcal{S}}$ auf $\langle M' \rangle$ an. Ist die Antwort "ja", so ist $\langle M \rangle w \in L_u$, andernfalls nicht.

L_u ist aber nicht rekursiv, also kann auch $\mathcal{S}$ nicht entscheidbar sein. ■

Nun sind wir in der Lage, relativ einfach einzusehen, daß auch das eingeschränkte Halteproblem unentscheidbar ist. Dazu betrachten wir auch noch das eingeschränkte Akzeptierungsproblem.

Definition 4.2.2: Mit der selben Bezeichnung wie in Def. 4.1.4 sei

$$L_{u,\epsilon} = \{ \langle M \rangle \mid M \text{ akzeptiert } \epsilon \}.$$

$L_{u,\epsilon}$ ist die zum *eingeschränkten Akzeptierungsproblem* gehörige Sprache. ■

Satz 4.2.2: *$L_{u,\epsilon}$ ist nicht rekursiv.*

Beweis: Sei $\mathcal{S}_a$ die Eigenschaft r.a. Sprachen, das leere Wort zu enthalten, also

$$L_{\mathcal{S}_a} = \{ \langle M \rangle \mid \epsilon \in L(M) \}.$$

$\mathcal{S}_a$ ist eine nichttriviale Eigenschaft r.a. Sprachen, und $L_{\mathcal{S}_a} = L_{u,\epsilon}$. Wegen des Satzes von Rice ist $\mathcal{S}_a$ unentscheidbar. ■

Satz 4.2.3: *$L_{h,\epsilon}$ ist nicht rekursiv.*

Beweis: Wäre das eingeschränkte Halteproblem entscheidbar, so wäre auch das eingeschränkte Akzeptierungsproblem entscheidbar. Für gegebenes $\langle M \rangle$ müßten wir einfach entscheiden, ob M auf ϵ hält, und gegebenenfalls ob M das leere Wort ϵ akzeptiert. Wegen Satz 4.2.2 kann das aber nicht möglich sein. ■

4.3. Die Unentscheidbarkeit eines Pflasterungsproblems

Wir geben uns eine endliche Menge von Typen von Pflastersteinen vor, von denen jeder die Form des Einheitsquadrats hat. Das Problem besteht darin, den ersten Quadranten der Ebene mit Pflastersteinen der gegebenen Typen zu pflastern. Die einzigen Einschränkungen bestehen darin, daß ein spezieller "Anfangs"Stein an der linken unteren Ecke plaziert werden muß, und daß nur gewisse Typen horizontal und vertikal benachbart sein dürfen. Es ist auch nicht gestattet, die Pflastersteine zu drehen.

Definition 4.3.1: Etwas formaler verstehen wir unter einer *Instanz des Pflasterungsproblems* ein Quadrupel $\mathcal{D} = (D, d_0, H, V)$, wobei D eine endliche Menge (von Pflastersteintypen) ist, $d_0 \in D$ (Anfangsstein), und $H, V \subseteq D \times D$ (erlaubte Paare benachbarter Typen). Eine *Pflasterung* durch $\mathcal{D}$ ist eine Funktion $f : \mathbf{N} \times \mathbf{N} \to D$ sodaß

$$f(0,0) = d_0,$$

$$(f(n,m), f(n+1,m)) \in H \quad \text{und} \quad (f(n,m), f(n,m+1)) \in V \quad \text{für alle } n, m \in \mathbf{N}.$$

Das *Pflasterungsproblem* lautet also wie folgt: *"Gibt es zum Pflasterungssystem $\mathcal{D}$ eine Pflasterung durch $\mathcal{D}$?"* ■

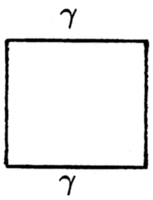
Satz 4.3.1: Das Pflasterungsproblem ist unentscheidbar.

Beweis: Daß das Pflasterungsproblem unentscheidbar ist, zeigen wir dadurch, daß wir das eingeschränkte Halteproblem darauf reduzieren. Wäre also das Pflasterungsproblem entscheidbar, so wäre auch das Halteproblem entscheidbar. Das ist aber laut Satz 4.2.3 nicht der Fall.

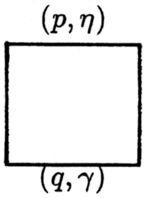
Dazu ist es notwendig für jede Turing–Maschine M ein Pflasterungssystem $\mathcal{D}_M$ zu konstruieren, sodaß es genau dann eine Pflasterung durch $\mathcal{D}_M$ gibt, wenn die Berechnung von M auf der leeren Eingabe nicht stoppt. Um die Betrachtung zu vereinfachen, schränken wir Turing–Maschinen dahingehend ein, daß sie nur entweder schreiben können, oder den LS–Kopf bewegen, aber nicht beides gleichzeitig. Man kann sich leicht überlegen, daß auf diese Weise die Berechnungspotenz einer Turing–Maschine nicht geschmälert wird.

Ist nun M eine Turing–Maschine mit Zustandsmenge Q, Anfangszustand q_0, Bandalphabet Γ und Überführungsfunktion δ, so ist $\mathcal{D}_M = (D, d_0, H, V)$, wobei die Paare in H, V an den benachbarten Seiten übereinstimmen müssen und D die folgenden Typen von Pflastersteinen enthält:

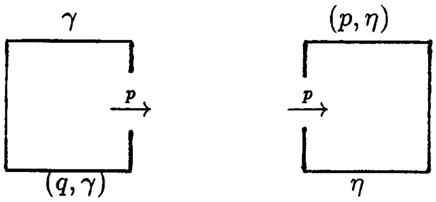
(a) Für jedes Symbol $\gamma \in \Gamma$ die Type



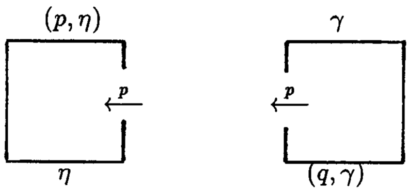
(b) Für jedes $\gamma \in \Gamma$ und $q \in Q$, sodaß $\delta(q,\gamma) = (p,\eta)$ für gewisse $p \in Q$, $\eta \in \Gamma$ die Type



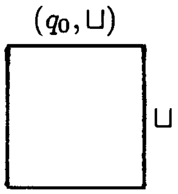
(c) Für jedes $\gamma \in \Gamma$ und $q \in Q$, sodaß $\delta(q,\gamma) = (p,R)$ für ein gewisses $p \in Q$, und für jedes $\eta \in \Gamma$ die Typen



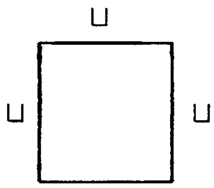
(d) Für jedes $\gamma \in \Gamma$ und $q \in Q$, sodaß $\delta(q,\gamma) = (p,L)$ für ein gewisses $p \in Q$, und für jedes $\eta \in \Gamma$ die Typen



(e) Darüberhinaus sei d_0 die Type

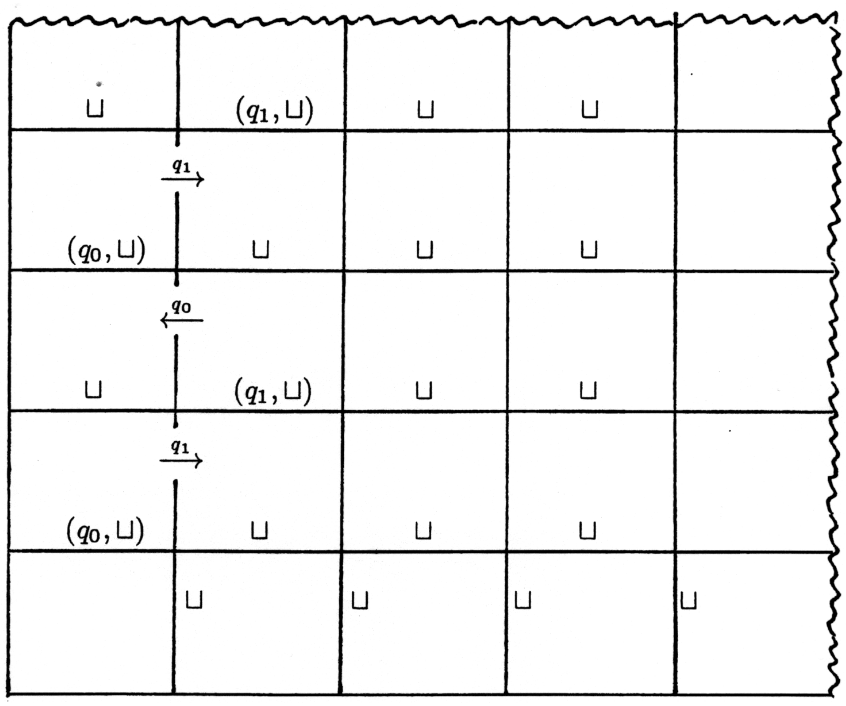


und wir haben auch die die folgende Type zur Verfügung.



Durch diese Festlegung von $\mathcal{D}_M$ erzwingen wir, daß die "Zeilen" einer durch $\mathcal{D}_M$ erzeugten Pflasterung den Konfigurationen in der Berechnung der Turing–Maschine M auf der leeren Eingabe entsprechen. Der erste Quadrant kann also genau dann durch $\mathcal{D}_M$ gepflastert werden, wenn die Berechnung von M auf der leeren Eingabe nicht stoppt. Somit ist die Reduktion des Halteproblems auf das Pflasterungsproblem durchgeführt. ■

Beispiel 4.3.1: Als Beispiel betrachten wir die Turing–Maschine M_4 mit $Q = \{q_0, q_1\}$, $\Gamma = \{\sqcup\}$, $\delta(q_0, \sqcup) = (q_1, R)$, $\delta(q_1, \sqcup) = (q_0, L)$. M_4 bewegt einfach den LS–Kopf hin und her und kommt dabei niemals über die zweite Bandzelle hinaus. Die Pflasterung des ersten Quadranten, die der unendlichen Berechnung von M_4 auf der leeren Eingabe entspricht, sieht folgendermaßen aus



4.4. Das Korrespondenzproblem von Post

Ein anderes unentscheidbares Problem ist das Korrespondenzproblem von Post. Die Unentscheidbarkeit des Korrespondenzproblems von Post hat zur Folge, daß auch viele andere interessante Problem als unentscheidbar erwiesen werden können, so z.B. die Mehrdeutigkeit kontextfreier Grammatiken.

Definition 4.4.1: (a) Eine *Instanz des Korrespondenzproblems von Post (Post's correspondence problem, PCP)* besteht aus zwei Listen von Wörtern

$$A = w_1, \ldots, w_k \qquad \text{und} \qquad B = x_1, \ldots, x_k$$

über einem Alphabet Σ. Diese Instanz des PCP *hat eine Lösung* genau dann, wenn es eine Folge von ganzen Zahlen $i_1, \ldots, i_m \in \{1, \ldots, k\}$, mit $m \geq 1$, gibt, sodaß

$$w_{i_1} \cdots w_{i_m} = x_{i_1} \cdots x_{i_m}.$$

Die Folge $i_1, \ldots, i_m$ ist eine *Lösung* dieser Instanz des PCP.

(b) Ein *Instanz des modifizierten Korrespondenzproblems von Post (MPCP)* sieht aus wie eine Instanz des PCP. Für eine *Lösung* des MPCP muß aber zusätzlich gelten $i_1 = 1$, also der Lösungsstring muß jeweils mit dem ersten Element der Listen A, B beginnen. ■

Beispiel 4.4.1: Sei das zugrunde liegende Alphabet $\Sigma = \{0, 1\}$.

(a) Wir betrachten die Instanz des PCP, welche durch folgende Listen gegeben ist:

	Liste A	Liste B
i	w_i	x_i
1	1	111
2	10111	10
3	10	0

Diese Instanz des PCP hat eine Lösung, nämlich

$$m = 4, i_1 = 2, i_2 = 1, i_3 = 1, i_4 = 3,$$

also

$$w_2 w_1 w_1 w_3 = 101111110 = x_2 x_1 x_1 x_3.$$

(b) Wir betrachten die Instanz des PCP, welche durch folgende Listen gegeben ist:

	Liste A	Liste B
i	w_i	x_i
1	10	101
2	011	11
3	101	011

Angenommen $i_1, \ldots, i_m$ ist eine Lösung dieser Instanz des PCP. Um die Übereinstimmung am Stringanfang zu gewährleisten muß gelten $i_1 = 1$. Also wir haben bisher als Teillösung

$$w_1 = 10$$
$$x_1 = 101$$

Wir müssen nun aus A ein Wort wählen, das mit 1 beginnt, also $i_2 \in \{1, 3\}$. $i_2 = 1$ funktioniert aber nicht, da wir sonst in der Teillösung eine Diskrepanz an der vierten Stelle hätten. Somit bleibt nur $i_2 = 3$, also die Teillösung

$$w_1 = 10101$$
$$x_1 = 101011$$

Nun sind wir aber wieder in der gleichen Situation wie vorhin, nämlich daß die Teillösung aus B um eine 1 länger ist als die Teillösung aus A. Mit derselben Argumentation wie oben ergibt sich also $i_3 = i_4 = \ldots = 3$. Eine andere Wahlmöglichkeit gibt es nicht. Die beiden Strings können also nie dieselbe Länge haben. Somit ist diese Instanz des PCP unlösbar. ∎

Satz 4.4.1: *Wäre* PCP *entscheidbar, dann wäre auch* MPCP *entscheidbar.*

Beweis: Sei

$$A = w_1, \ldots, w_k, \qquad B = x_1, \ldots, x_k$$

eine Instanz von MPCP. Wir transformieren diese Instanz von MPCP in eine Instanz von PCP, welche genau dann lösbar ist, wenn die gegebene Instanz von MPCP lösbar ist. Damit ist der Satz dann bewiesen.

Sei Σ das kleinste Alphabet, welches alle Symbole in A und B enthält und zusätzlich die beiden Symbole $\emptyset$ und \$. Sei y_i das Wort, das aus w_i entsteht, indem man nach jedem Symbol in w_i das Symbol $\emptyset$ einfügt, und sei z_i das Wort, das aus x_i entsteht, indem man vor jedem Symbol in x_i das Symbol $\emptyset$ einfügt. Weiters betrachten wir neue Wörter

$$y_0 = \emptyset y_1, \qquad z_0 = z_1,$$

$$y_{k+1} = \$, \qquad z_{k+1} = \emptyset\$.$$

Seien nun

$$C = y_0, y_1, \ldots, y_{k+1}, \qquad D = z_0, z_1, \ldots, z_{k+1}.$$

Für die Listen aus Beispiel 4.4.1(a) ergibt sich etwa dadurch folgende Transformation:

MPCP:

	Liste A	Liste B
i	w_i	x_i
1	1	111
2	10111	10
3	10	0

PCP:

	Liste C	Liste D
i	y_i	z_i
0	∅1∅	∅1∅1∅1
1	1∅	∅1∅1∅1
2	1∅0∅1∅1∅1∅	∅1∅0
3	1∅0∅	∅0
4	\$	∅\$

Die Listen C und D spezifizieren eine Instanz des PCP, welche genau dann lösbar ist, wenn die gegebene Instanz des MPCP lösbar ist.

Denn ist etwa $1, i_1, \ldots, i_r$ eine Lösung des MPCP, dann ist $0, i_1, \ldots, i_r, k+1$ eine Lösung des PCP.

Andererseits, ist $i_1, \ldots, i_r$ eine Lösung des PCP, dann muß gelten $i_1 = 0$ und $i_r = k+1$. Sei j der kleinste Index sodaß $i_j = k+1$. Dann ist $i_1, \ldots, i_j$ auch eine Lösung, da das Symbol \$ nur als letztes Symbol von y_{k+1} und z_{k+1} vorkommt, und für kein l mit $1 \leq l < j$ gilt $i_l = k+1$. Somit ist $1, i_1, \ldots, i_{j-1}$ offenbar eine Lösung des MPCP.

Gibt es also einen Algorithmus um PCP zu entscheiden, so erhalten wir daraus auch einen Algorithmus um MPCP zu entscheiden. ■

Mit dieser Hilfsüberlegung sind wir nun in der Lage, die Unentscheidbarkeit von PCP zu beweisen.

Satz 4.4.2: PCP *ist unentscheidbar.*

Beweis: Wir zeigen, daß das Akzeptierungsproblem für Turing–Maschinen auf MPCP reduzierbar ist, d.h. wäre MPCP entscheidbar, so wäre auch das Akzeptierungsproblem entscheidbar. Das ist aber laut Satz 4.1.3 nicht der Fall. Somit ist MPCP nicht entscheidbar, und wegen Satz 4.4.1 auch PCP nicht entscheidbar.

Für jede TM M und jedes Wort w konstruieren wir ein MPCP, welches lösbar ist genau dann, wenn es eine Lösung der Form

$$\#q_0 w \# u_1 q_1 v_1 \# \cdots \# u_k q_k v_k \#$$

hat, wobei Strings zwischen aufeinanderfolgenden # aufeinanderfolgende Konfigurationen von M sind bei Eingabe w, und q_k ein akzeptierender Zustand von M ist.

Wir geben nun die Listen A und B des zugehörigen MPCP an. Wir nehmen an, daß die TM M stoppt, sobald ein akzeptierender Zustand erreicht wird. Außer für das erste Paar vergeben wir keine Nummern, da diese für die Existenz einer Lösung irrelevant sind.

Gruppe 0: erstes Paar

Liste A	Liste B
$\#$	$\#q_0 w\#$

Gruppe I: (Kopieren) für $\alpha \in \Gamma$:

Liste A	Liste B
α	α
$\#$	$\#$

Gruppe II: (Anwenden der Überführungsfunktion) für $q \in Q \setminus F$, $p \in Q$, $\alpha, \beta, \gamma \in \Gamma$:

Liste A	Liste B	
$q\alpha$	βp	falls $\delta(q, \alpha) = (p, \beta, R)$
$\gamma q \alpha$	$p \gamma \beta$	falls $\delta(q, \alpha) = (p, \beta, L)$
$q\#$	$\beta p \#$	falls $\delta(q, \sqcup) = (p, \beta, R)$
$\gamma q \#$	$p \gamma \beta \#$	falls $\delta(q, \sqcup) = (p, \beta, L)$

Gruppe III: (Endbehandlung) für $q \in F$, und $\alpha, \beta \in \Gamma$:

Liste A	Liste B
$\alpha q \beta$	q
αq	q
$q \beta$	q

Gruppe VI: (Liste A abschließen) für $q \in F$:

Liste A	Liste B
$q\#\#$	$\#$

Wir nennen (x, y) eine *Teillösung* von MPCP bzgl. der Listen A und B, wenn x ein Anfangsstück (Prefix) von y ist, und x bzw. y durch Verkettung korrespondierender Strings aus den Listen A bzw. B hervorgehen. Ist $xz = y$, so nennen wir z den *Rest* von (x, y).

Angenommen ausgehend von der Konfiguration q_0w gibt es eine gültige Folge von k weiteren Konfigurationen, also eine Berechnung dieser Länge (die aber nicht notwendigerweise hier zu Ende sein muß). Dann behaupten wir, daß es eine Teillösung der Form

$$\begin{aligned}(x, y) = (&\#q_0w\#u_1q_1v_1\# \cdots \#u_{k-1}q_{k-1}v_{k-1}\#,\\ &\#q_0w\#u_1q_1v_1\# \cdots \#u_{k-1}q_{k-1}v_{k-1}\#u_kq_kv_k\#)\end{aligned}$$

gibt. Zudem ist das die einzige Teillösung, deren längerer String so lang ist wie $|y|$.

Wir beweisen diese Behauptung durch Induktion über k. Für $k = 0$ ist die Behauptung trivial, da das Paar $(\#, \#q_0w\#)$ als erstes gewählt werden muß.

Angenommen die Behauptung gilt für ein k, und daß q_k nicht in F ist. Der Rest des Paares (x, y) ist $z = u_kq_kv_k\#$. Die nächsten Paare müssen nun so gewählt werden, daß ihre Anteile aus A den String z ergeben. Unabhängig davon welche Symbole rechts und links von q_k stehen, gibt es höchstens ein Paar in Gruppe II, welches es erlaubt, die Teillösung über q_k hinaus fortzusetzen. Dieses Paar entspricht in natürlicher Weise einem Berechnungsschritt von M ausgehend von der Konfiguration $u_kq_kv_k$. Die anderen Symbole von z führen zwangsläufig zu Paaren aus der Gruppe I. Keine andere Auswahl von Paaren erlaubt es, z darzustellen als Verkettung von Elementen der Liste A. Somit erhalten wir eine neue Teillösung $(y,\, yu_{k+1}q_{k+1}v_{v+1}\#)$. Offensichtlich ist $u_{k+1}q_{k+1}v_{k+1}$ die einzige Konfiguration, welche M in einem Schritt von $u_kq_kv_k$ erreichen kann. Es gibt auch keine andere Teillösung, deren zweiter String so lang ist wie $|yu_{k+1}q_{k+1}v_{k+1}|$. Somit ist die Behauptung auch für $k + 1$ nachgewiesen.

Ist nun $q_k \in F$, so findet man leicht Paare aus den Gruppen I und III, welche es erlauben die Teillösung (x, y) zu einer Lösung des MPCP mit Listen A und B zu vervollständigen, nachdem als letztes Paar dasjenige aus Gruppe IV gewählt wurde.

Somit ist klar, daß MPCP mit Listen A und B eine Lösung besitzt, wenn M ausgehend von der Konfiguration q_0w einen akzeptierenden Zustand erreicht. Erreicht M keinen akzeptierenden Zustand, so können die Paare aus den Gruppen III und IV nicht angewendet werden. Somit wird in jeder Teillösung der zweite String länger sein als der erste, und es ist daher keine vollständige Lösung möglich.

Abschließend stellen wir also fest, daß die Instanz des MPCP genau dann eine Lösung hat, wenn M bei Eingabe w in einem akzeptierenden Zustand hält. Die obige Konstruktion kann für jede TM M und Eingabewort w ausgeführt werden. Gäbe es also einen Algorithmus zur Entscheidung von MPCP, dann gäbe es auch einen Algorithmus zur Entscheidung des Akzeptierungsproblems für Turing–Maschinen, im Widerspruch zu Satz 4.1.3. ■

Übung: Sei M die TM

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, \sqcup\}, q_0, \{q_2\}, \delta),$$

wobei δ wie folgt definiert ist:

	0	1	$\sqcup$
q_0	$(q_1, 1, R)$	$(q_1, 0, L)$	$(q_1, 1, L)$
q_1	$(q_2, 0, L)$	$(q_0, 0, R)$	$(q_1, 0, R)$
q_2	–	–	–

Wie sieht die zugehörige Instanz des MPCP aus? M akzeptiert das Wort $w = 01$ in 4 Berechnungsschritten. Wie sieht die zugehörige Lösung der Instanz des MPCP aus?

4.5. Es gibt auch interessante entscheidbare Problem

Lösbarkeit algebraischer Gleichungssysteme

Zu gegebenen Polynomen $f_1(x_1,\ldots,x_\nu),\ldots,f_m(x_1,\ldots,x_\nu)$ im Polynomring $\mathbb{Q}[x_1,\ldots,x_\nu]$ in den Variablen $x_1,\ldots,x_\nu$ über dem Körper $\mathbb{Q}$ entscheide man, ob es eine Lösung des Gleichungssystems

$$\begin{aligned} f_1(x_1,\ldots,x_\nu)&=0\\ &\vdots\\ f_m(x_1,\ldots,x_\nu)&=0 \end{aligned} \tag{4.5.1}$$

in $\mathbb{C}^n$ gibt. Dazu berechnet man eine reduzierte Gröbner–Basis $g(x_1,\ldots,x_\nu),\ldots,$ $g_n(x_1,\ldots,x_\nu)$ für das von $f_1,\ldots,f_m$ aufgespannte Ideal [2]. Das Gleichungssystem (4.5.1) hat genau dann eine Lösung, wenn eines der Polynome g_i das Einspolynom ist.

Als Beispiel betrachten wir das algebraische Gleichungssystem

$$\begin{aligned} xz-xy^2-4x^2-\frac{1}{4}&=0\\ y^2z+2x+\frac{1}{2}&=0\\ x^2z+y^2+\frac{1}{2}x&=0 \end{aligned}$$

Durch den Übergang zur Gröbner–Basis erhalten wir das neue Gleichungsystem

$$\begin{aligned} z+\frac{64}{65}x^4-\frac{432}{65}x^3+\frac{168}{65}x^2-\frac{354}{65}x+\frac{8}{5}&=0\\ y^2-\frac{8}{13}x^4+\frac{54}{13}x^3-\frac{8}{13}x^2+\frac{17}{26}x&=0\\ x^5-\frac{27}{4}x^4+2x^3-\frac{21}{16}x^2+x+\frac{5}{32}&=0 \end{aligned}$$

Das Gleichungssystem ist also lösbar, und die Approximation $(-0.128475,\ 0.321145,$ $-2.356718)$ einer Lösung ist leicht ablesbar.

---

[2] B. Buchberger: *Gröbner bases: An algorithmic method in polynomial ideal theory*, in: *Mathematical Systems Theory*, N.K. Bose (ed.), D. Reidel Publ. Comp., Dordrecht/Boston/Lancaster (1985).

Integration transzendenter elementarer Funktionen

Für eine gegebene transzendente elementare Funktion $f(x)$ — aufgebaut aus den rationalen Funktionen $\mathbb{C}(x)$ über dem komplexen Zahlkörper durch Adjunktion von Logarithmen und Exponentialen — stelle man fest, ob $g(x) = \int f(x)dx$ ebenfalls eine transzendente elementare Funktion ist und berechne gegebenenfalls eine Darstellung von $g(x)$.

Dieses Problem ist mithilfe des Algorithmus von Risch [3)] lösbar, und wird in der Praxis auch von Programmsystemen der Computer Algebra gelöst. So erhält man etwa

$$\int 2x \cdot e^{x^2} \cdot \log(x) + \frac{e^{x^2}}{x} + \frac{\log(x) - 2}{(\log^2(x) + x)^2} + \frac{(2/x) \cdot \log(x) + 1/x + 1}{\log^2(x) + x}\, dx \quad =$$

$$e^{x^2} \cdot \log(x) - \frac{\log(x)}{\log^2(x) + x} + \log(\log^2(x) + x)$$

oder die Antwort, daß

$$\int e^{x^2} dx$$

nicht transzendent elementar ist.

Parametrisierung algebraischer Kurven

Eine (ebene) algebraische Kurve $\mathcal{C}$ besteht aus denjenigen Punkten $P = (x, y)$ in der Ebene, welche eine algebraische Gleichung der Form

$$f(x, y) = 0$$

lösen, wobei $f(x, y)$ ein Polynom in den Variablen x, y ist.

Ein Beispiel dafür ist die sogenannte "Tacnode"–Kurve $\mathcal{T}$, deren Punkte Lösungen der Gleichung

$$2x^4 - 3x^2y + y^4 - 2y^3 + y^2 = 0$$

sind. Ein Bild der Tacnode–Kurve befindet sich auf der nächsten Seite.

Diese Gleichungsdarstellung algebraischer Kurven ist manchmal vorteilhaft, etwa wenn man feststellen will, ob ein gegebener Punkt auf der Kurve liegt oder nicht. Sie eignet sich aber nicht besonders gut für andere Aufgabenstellungen, etwa wenn

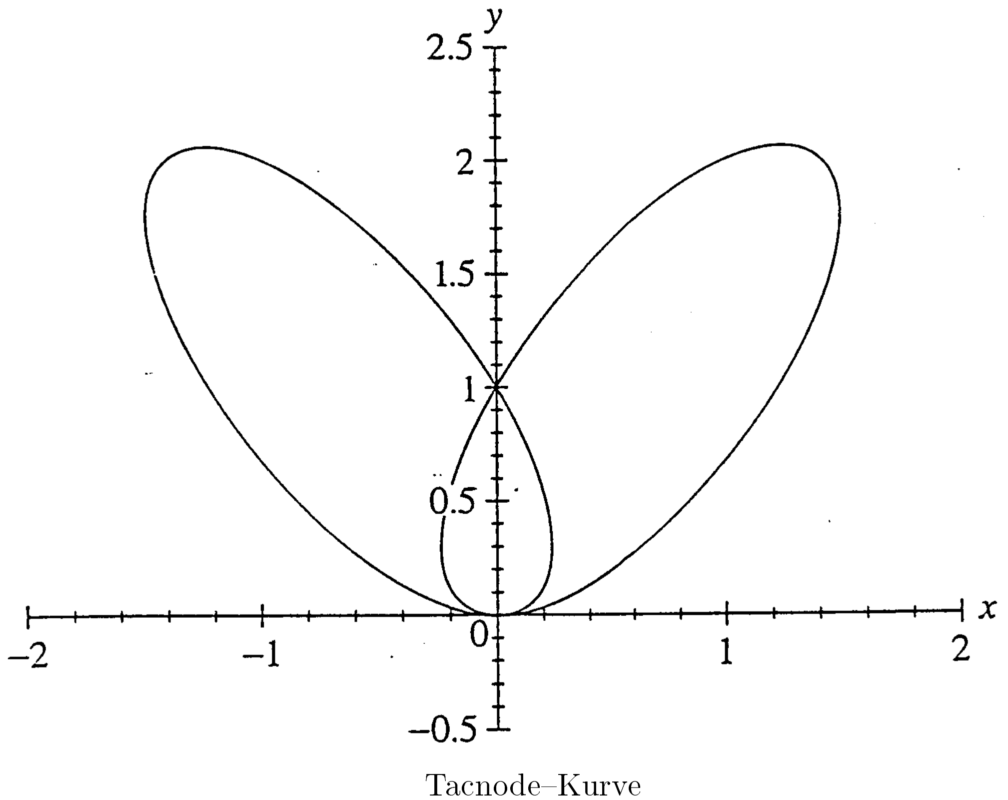
[3)] R.H. Risch: *The problem of integration in finite terms*, *Trans. Amer. Math. Soc.* 139, 167–189 (1969).

man Punkte auf der Kurve erzeugen will. Dazu wäre eine Parameterdarstellung, etwa mittels rationaler Funktionen in einem Parameter, vorteilhaft. Dabei drückt man die Koordinaten von Punkten auf der Kurve aus durch rationale Funktionen in einem unabhängigen Parameter:

$$x(t) = \frac{p_1(t)}{q_1(t)}, \quad y(t) = \frac{p_2(t)}{q_2(t)},$$

wobei p_1, p_2, q_1, q_2 Polynome in t sind. Allerdings ist nicht jede algebraische Kurve so darstellbar. Das Problem der rationalen Parametrisierung algebraischer Kurven besteht also darin, für eine gegebene Kurve in Gleichungsdarstellung zu entscheiden, ob sie parametrisierbar ist, und gegebenenfalls eine solche Parameterdarstellung zu berechnen. Eine symbolische algorithmische Lösung dieses Problems findet man etwa in den Arbeiten von Sendra und Winkler [4]. So kann die Tacnode–Kurve wie folgt rational parametrisiert werden:

$$x(t) = \frac{t^3 - 6t^2 + 9t - 2}{2t^4 - 16t^3 + 40t^2 - 32t + 9}, \quad y(t) = \frac{t^2 - 4t + 4}{2t^4 - 16t^3 + 40t^2 - 32t + 9}.$$



Tacnode–Kurve

[4] J.R. Sendra und F. Winkler: *Symbolic Parametrization of Curves*, *J. Symbolic Computation* 12, 607–631 (1991),
J.R. Sendra und F. Winkler: *Parametrization of Algebraic Curves over Optimal Field Extensions*, *J. Symbolic Computation* 23, 191–207 (1997).

4.6. Unentscheidbarkeitshierarchie

Das *Emptiness–Problem* besteht darin, für eine gegebene Turingmaschine M zu entscheiden, ob $L(M) = \emptyset$. Die zugehörige Sprache ist

$$L_e = \{\langle M\rangle | L(M) = \emptyset\}.$$

Satz 4.6.1: *Seien*

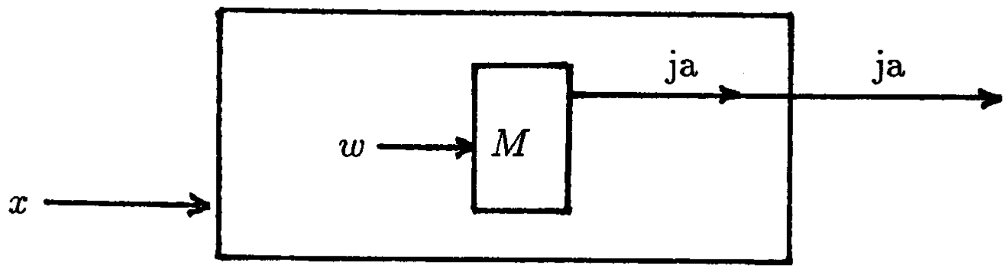
$$L_e = \{\langle M\rangle | L(M) = \emptyset\} \quad \textit{und} \quad L_{ne} = \{\langle M\rangle | L(M) \neq \emptyset\}.$$

L_{ne} ist r.a. aber nicht rekursiv, und L_e ist nicht r.a.

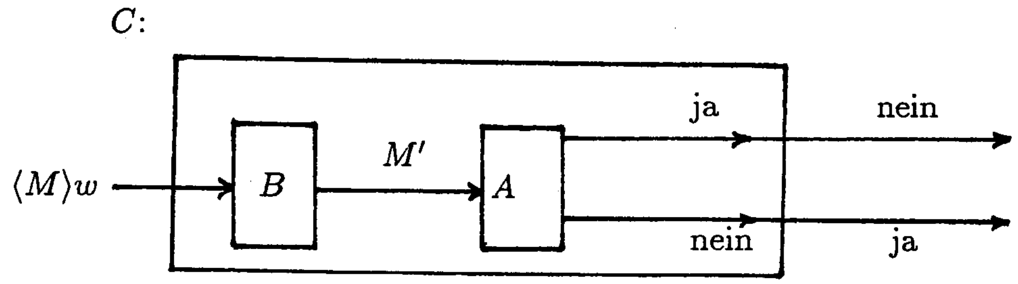
Beweis: Zunächst zeigen wir, daß L_{ne} r.a. ist, indem wir eine Turingmaschine M konstruieren, welche Kodes von Turingmaschinen erkennt, welche nichtleere Sprachen akzeptieren. M nimmt also als Eingabe einen Kode $\langle M_i\rangle$, und erzeugt systematisch Paare $(j,k) \in \mathbb{N}^2$. Für so ein Paar (j,k) simuliert M nun k Berechnungsschritte der Turingmaschine M_i auf dem j–ten Wort w_j (wie im Beweis zu Satz 1.4.1). Wird dabei w_j akzeptiert, so akzeptiert M den Kode $\langle M_i\rangle$.

Nun zeigen wir, daß L_e nicht rekursiv ist. Das folgt aus dem Satz von Rice. Wir wollen aber hier einen direkten Beweis geben, da wir die dabei notwendigen Konstruktionsschritte auch später noch brauchen werden. Es ist nicht schwer zu sehen, daß es einen Algorithmus (immer terminierende Turingmaschine) B gibt, welcher eine Eingabe der Form $\langle M\rangle w$ nimmt und eine Turingmaschine M' konstruiert, sodaß

$$L(M') = \begin{cases} \emptyset, & \text{falls } w \notin L(M), \\ (0+1)^*, & \text{falls } w \in L(M). \end{cases}$$



(Details siehe [HU], p. 185 ff). Angenommen L_e wäre rekursiv, und A wäre ein Algorithmus mit $L(A) = L_e$. Dann könnten wir einen Algorithmus C mit $L(C) = L_u$ (Akzeptierungsproblem, Def. 4.1.4) konstruieren wie in folgender Skizze:



So ein Algorithmus C kann aber nach Satz 4.1.3 nicht existieren. Also kann auch A nicht existieren. Somit ist L_e nicht rekursiv.

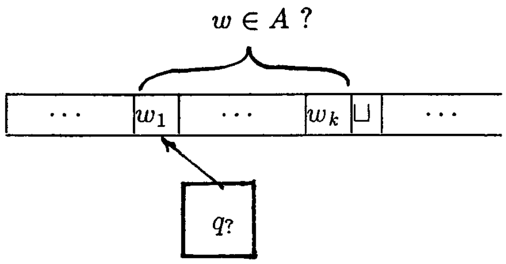
Wäre nun L_{ne} rekursiv, dann auch sein Komplement L_e. Somit ist L_{ne} nicht rekursiv.

Wäre L_e r.a., dann wären L_e und L_{ne} rekursiv (Bemerkung nach Def. 1.4.4). Also ist L_e nicht r.a. ■

Man könnte sich aber nun die Frage stellen, was wäre, wenn man das Emptiness–Problem (oder ein anderes unentscheidbares Problem) nun doch irgendwie entscheiden könnte. Wäre dann alles entscheidbar? Könnten wir dann alles berechnen?

Dabei muß man natürlich vorsichtig vorgehen. Wenn wir einfach die Entscheidbarkeit des Emptiness–Problems als zusätzliches Axiom annehmen würden, so wäre unsere Theorie sofort widersprüchlich, und wir könnten natürlich alles beweisen. Das vermeiden wir durch die Einführung von Orakel–Turingmaschinen.

Definition 4.6.1: Sei A eine Sprache über Σ, $A \subseteq \Sigma^*$. Eine *Orakel–Turingmaschine (OTM) mit Orakel* A ist eine (Ein–Band–) Turingmaschine M^A mit drei ausgezeichneten Zuständen, $q_?$, q_j und q_n. Wenn M^A den Zustand $q_?$ annimmt, so ersucht sie um eine Antwort auf die Frage: "ist das Wort rechts vom (und beginnend beim) LS–Kopf bis zum ersten Blank in A?" Je nachdem, ob die Antwort auf diese Frage "ja" oder "nein" ist, nimmt M im nächsten Schritt den Zustand q_j oder q_n an. ■



Eine so definierte Turingmaschine hat also ein Orakel zur Verfügung, an das sie Fragen betreffend die Sprache A stellen kann. Ist A eine rekursive Sprache, so kann dieses Orakel durch eine andere Turingmaschine simuliert werden, und die von der Orakel–Turingmaschine M^A akzeptierte Sprache ist daher r.a. Ist dagegen A nicht rekursiv, so akzeptiert M^A möglicherweise eine Sprache, welche nicht r.a. ist.

Definition 4.6.2: Eine Sprache B ist *rekursiv aufzählbar bezüglich* A, wenn es eine Orakel–Turingmaschine M^A gibt mit $B = L(M^A)$. Eine Sprache B ist *rekursiv bezüglich* A, wenn es eine Orakel–Turingmaschine M^A gibt, welche immer hält, mit $B = L(M^A)$. Zwei Sprachen A und B sind *äquivalent*, wenn jede rekursiv ist bezüglich der anderen. ■

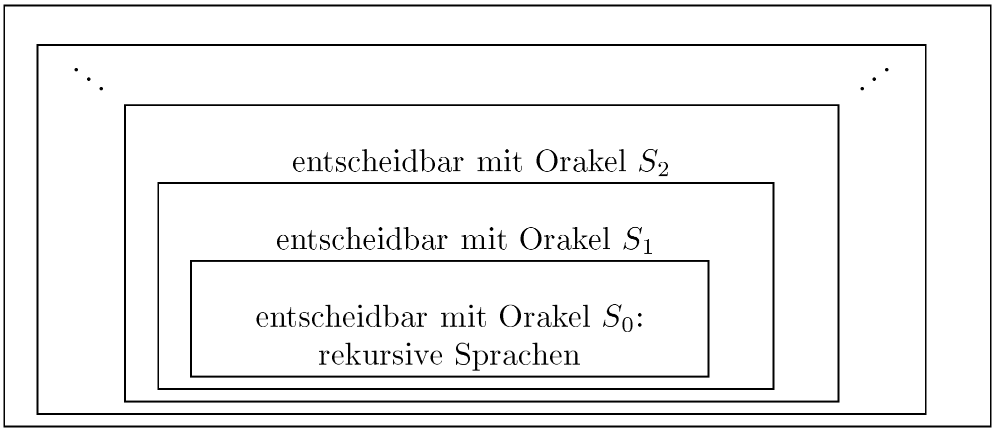
Natürlich können nicht alle Sprachen r.a. bzgl. L_e sein, da es überabzählbar viele Sprachen gibt, aber nur abzählbar viele Turingmaschinen. Es gibt also eine Sprache, welche nicht r.a. bzgl. L_e ist. Diese könnten wir wiederum als Orakel verwenden, und so fort. Auf diese Weise erhalten wir eine ganze Hierarchie unentscheidbarer Sprachen bzw. Probleme.

Definition 4.6.3: Wir definieren eine Folge kanonischer Orakelmengen:

$$\begin{aligned} S_0 &= \emptyset, \\ S_1 &= \{\langle M\rangle | L(M^{S_0}) = \emptyset\}, \\ &\vdots \\ S_{i+1} &= \{\langle M\rangle | L(M^{S_i}) = \emptyset\}, \\ &\vdots \end{aligned}$$

S_{i+1} ist also ein Orakel, um das Emptiness–Problem für Berechnungen bzgl. S_i zu lösen. ■

Eine TM mit Orakel S_0 ist also offenbar eine TM, welche kein Orakel benutzt. S_1 ist nichts anderes als die Menge L_e aus Satz 4.6.1. Wir erhalten folgende Einteilung der Klassen von Sprachen über $\{0, 1\}$:



Nun kann man einige (aber nicht alle) unentscheidbare Probleme nach ihrer Äquivalenz zu Elementen der Folge $(S_i)_{i\in\mathbb{N}}$ klassifizieren.

Satz 4.6.2: *Das Akzeptierungsproblem für Turingmaschinen ohne Orakel ist äquivalent zu* S_1.

Beweis: L_u rekursiv bzgl. S_1: Wir konstruieren eine OTM M^{S_1}, welche zunächst bei Eingabe $\langle M\rangle w$ den Kode der TM M' wie im Beweis zu Satz 4.6.1 konstruiert, also

$$L(M') = \begin{cases} \emptyset, & \text{falls } w \notin L(M), \\ (0+1)^*, & \text{falls } w \in L(M). \end{cases}$$

Hierauf nimmt M^{S_1} den Zustand $q_?$ an, mit dem Kode von M' rechts vom LS–Kopf und akzeptiert genau dann, wenn im nächsten Schritt der Zustand q_n angenommen wird. Das Akzeptierungsproblem für TM ohne Orakel ist also rekursiv bzgl. S_1.

S_1 rekursiv bzgl. L_u: Umgekehrt zeigen wir, daß es eine OTM $M_2^{L_u}$ mit Orakel L_u gibt, welche S_1 erkennt. Wir konstruieren eine TM M_2, welche bei Eingabe $\langle M\rangle$ eine neue TM M' erzeugt. M' operiert folgendermaßen: zunächst ignoriert M' seine Eingabe, und erzeugt systematisch alle Paare $(i,j)\in\mathbb{N}^2$. Für jedes (i,j) simuliert M' i Berechnungsschritte von M auf dem j–ten Wort. Akzeptiert M, so akzeptiert M' seine Eingabe.

$$L(M') = \begin{cases} \emptyset, & \text{falls } L(M) = \emptyset, \\ (0+1)^*, & \text{falls } L(M) \neq \emptyset. \end{cases}$$

$M_2^{L_u}$ befragt nun sein Orakel, ob M' das leere Wort ϵ akzeptiert, also ob $\langle M'\rangle\epsilon \in L_u$. Ist das der Fall, so weist $M_2^{L_u}$ die Eingabe M zurück, andernfalls akzeptiert es M. S_1 ist also rekursiv bzgl. L_u. ■

Nun wollen wir das Problem "$L(M) = \Sigma^*$?" betrachten. Dabei sei $\Sigma = \{0,1\}$ das Eingabealphabet der Turingmaschine M. Wir werden sehen, daß dieses Problem äquivalent ist zu S_2, also in gewisser Weise schwieriger ist als das Emptiness–Problem oder das Akzeptierungsproblem. In einem praktischen Sinn hat das natürlich keinerlei Bedeutung, da alle diese Probleme ohnedies unentscheidbar sind. Wenn wir aber diese Probleme einschränken, etwa auf reguläre Ausdrücke oder kontextfreie Grammatiken, so können wir tatsächlich feststellen, daß das "$= \Sigma^*$"–Problem schwieriger ist als das Akzeptierungsproblem.

Satz 4.6.3: *Das Problem "$L(M) = \Sigma^*?$" ist äquivalent zu S_2.*

Beweis: Zuerst zeigen wir, daß das "$= \Sigma^*$"–Problem rekursiv bzgl. S_2 ist. Wir konstruieren eine OTM $M_3^{S_2}$, welche genau jene Turingmaschinen M erkennt, die alle Eingabewörter akzeptieren. Zunächst erzeugt $M_3^{S_2}$ eine OTM $\hat{M}^{S_1}$ mit Orakel S_1, welche folgendermaßen operiert: $\hat{M}^{S_1}$ zählt Wörter $x \in \Sigma^*$ auf, und für jedes x wird das Orakel S_1 befragt, ob M das Wort x akzeptiert (siehe Satz 4.6.2). $\hat{M}^{S_1}$ akzeptiert seine Eingabe, wenn es ein x gibt, welches von M nicht akzeptiert wird, also

$$L(\hat{M}^{S_1}) = \begin{cases} \emptyset, & \text{falls } L(M) = \Sigma^*, \\ \Sigma^*, & \text{sonst.} \end{cases}$$

Nach der Konstruktion von $\hat{M}^{S_1}$ befragt nun $M_3^{S_2}$ sein Orakel S_2, ob $L(\hat{M}^{S_1}) = \emptyset$. Ist das der Fall, so akzeptiert $M_3^{S_2}$ seine Eingabe M. Andernfalls wird M zurückgewiesen. Also

$$L(M_3^{S_2}) = \{\langle M \rangle \mid L(M) = \Sigma^*\}.$$

Um zu zeigen, daß S_2 rekursiv ist bzgl. "$= \Sigma^*$", konstruiert man eine OTM $M_4^{L_*}$, welche S_2 akzeptiert, wobei

$$L_* = \{\langle M \rangle \mid L(M) = \Sigma^*\}.$$

Wir führen diesen Teil des Beweises aber nicht mehr aus, sondern verweisen auf [HU], Theorem 8.20. ■

5. Problemkomplexität

Im vorangegangenen Kapitel haben wir uns mit der Frage beschäftigt, ob ein Problem (eine Sprache) entscheidbar (erkennbar) ist. In diesem Kapitel betrachten wir nur entscheidbare Probleme (erkennbare Sprachen) und stellen die Frage nach der Effizienz dieser Entscheidung (Erkennung). Siehe etwa [Weg] für Zusatzinformation.

5.1. Komplexitätsklassen

Def. 5.1.1: Eine rekursive Sprache L hat *Zeitkomplexität* $T(n)$ (*Raumkomplexität* $S(n)$), wenn es einen Algorithmus (d.h. immer terminierende Turing–Maschine) mit Zeitkomplexität $T(n)$ (Raumkomplexität $S(n)$) gibt, welcher L erkennt. Die Sprache L hat *nichtdeterministische Zeitkomplexität* $T(n)$ (*nichtdeterministische Raumkomplexität* $S(n)$), wenn es eine nichtdeterministische Turing–Maschine mit Zeitkomplexität $T(n)$ (Raumkomplexität $S(n)$) gibt, welche L erkennt. ■

Beispiel 5.1.1: Sei L die Sprache $\{wcw^R \mid w \in \{0,1\}^*\}$. Dabei ist w^R die Umkehrung des Wortes w. Die Sprache L hat Zeitkomplexität $n+1$. Es gibt nämlich eine Turing–Maschine M_1 mit zwei Bändern, welche die Eingabe links von c auf das zweite Band kopiert, und anschließend synchron den LS–Kopf des ersten Bandes weiter nach rechts und den LS–Kopf des zweiten Bandes nach links bewegt. Dabei werden in jedem Schritt die beiden aktuellen Symbole verglichen. Sind diese jeweils gleich und ist überdies die Anzahl der Symbole links von c gleich der Anzahl der Symbole rechts von c, so akzeptiert M_1 die Eingabe. M_1 braucht nur $n+1$ Schritte, wenn die Eingabe von der Länge n ist.

Die Raumkomplexität von L ist offensichtlich $\mathcal{O}(n)$. ■

Def. 5.1.2: DTIME($T(n)$) ist die Familie der Sprachen mit Zeitkomplexität $T(n)$, DSPACE($S(n)$) ist die Familie der Sprachen mit Raumkomplexität $S(n)$.
NTIME($T(n)$) ist die Familie der Sprachen mit nichtdeterministischer Zeitkomplexität $T(n)$, NSPACE($S(n)$) ist die Familie der Sprachen mit nichtdeterministischer Raumkomplexität $S(n)$.
Alle diese Familien von Sprachen heißen Komplexitätsklassen. ■

Def. 5.1.3: Die Sprachen, welche in deterministischer polynomialer Zeit erkannt werden können, bilden eine natürliche und wichtige Klasse, welche wir mit $\mathcal{P}$ bezeichnen,

$$\mathcal{P} = \bigcup_{i \geq 1} \mathrm{DTIME}(n^i).$$

Die Klasse der Sprachen, welche in nichtdeterministischer polynomialer Zeit erkannt werden können, bezeichen wir mit $\mathcal{NP}$,

$$\mathcal{NP} = \bigcup_{i \geq 1} \mathrm{NTIME}(n^i).$$

Weiters definieren wir

$$\mathrm{PSPACE} = \bigcup_{i \geq 1} \mathrm{DSPACE}(n^i)$$

und

$$\mathrm{NSPACE} = \bigcup_{i \geq 1} \mathrm{NSPACE}(n^i). \qquad \blacksquare$$

Das Problem der Hamiltonschen Zyklen P_{hc} lautet folgendermaßen: *"Gibt es im Graphen G einen Zyklus, der jede Ecke genau einmal enthält?"* P_{hc} ist offensichtlich in $\mathcal{NP}$. Ein einfacher nichtdeterministischer Algorithmus ist etwa: errate die Ecken des Zyklus und verifiziere, daß sie tatsächlich einen Hamiltonschen Zyklus bilden. Es scheint jedoch keinen deterministischen Algorithmus mit polynomialer Zeitkomplexität zu geben, der P_{hc} entscheidet.

Der Unterschied zwischen $\mathcal{P}$ und $\mathcal{NP}$ ist vergleichbar mit dem Unterschied zwischen dem effizienten Auffinden eines Beweises für eine Behauptung (etwa "dieser Graph hat einen Hamiltonschen Zyklus") und dem effizienten Nachprüfen eines Beweises (etwa zu prüfen, ob eine gegebene Eckenfolge einen Hamiltonschen Zyklus darstellt). Intuitiv nimmt man wohl an, daß es einfacher ist einen gegebenen Beweis zu überprüfen als einen solchen aufzufinden. Mit Sicherheit wissen wir das aber nicht.

Man kann zeigen, daß PSPACE = NSPACE [1)]. Es gilt offensichtlich $\mathcal{P} \subseteq \mathcal{NP} \subseteq$ PSPACE. Von keiner dieser Inklusionen weiß man aber, ob sie echt ist.

In Kapitel 4 haben wir die Methode der Reduktion benutzt, um Probleme (Sprachen) als unentscheidbar zu erweisen. Dazu betrachteten wir eine Reduktionsfunktion g, sodaß $x \in L' \iff g(x) \in L$. Indem man die Reduktionsfunktion g einschränkt auf einfach zu berechnende Funktionen, ist es oft möglich nachzuweisen, daß

[1)] W.J. Savitch: *"Relations between nondeterministic and deterministic tape complexities", J. Computer and Systems Sciences* 4/2, 177–192 (1970).

eine Sprache L in einer der Klassen $\mathcal{P}$, $\mathcal{NP}$ oder PSPACE liegt. Wir werden uns speziell für zwei Arten von Reduktionen interessieren: Polynom–Zeit Reduktionen und log–Raum Reduktionen.

Def. 5.1.4: Seien L, L' Sprachen. L' ist *Polynom–Zeit reduzierbar auf* L, wenn es eine Turing–Maschine mit polynomialer Zeitkomplexität gibt, welche immer stoppt und für jede Eingabe x eine Ausgabe y erzeugt, sodaß $x \in L' \iff y \in L$. ∎

Satz 5.1.1: Sei L' Polynom–Zeit reduzierbar auf L. Dann
(a) $L \in \mathcal{P} \Longrightarrow L' \in \mathcal{P}$,
(b) $L \in \mathcal{NP} \Longrightarrow L' \in \mathcal{NP}$.

Beweis: (a) Sei $L \in \mathcal{P}$. Sei die Zeitkomplexität der Reduktion beschränkt durch das Polynom $p_1(n)$ und sei L erkennbar in Zeit $p_2(n)$, p_2 ein Polynom. Für ein gegebenes $x \in L'$ der Länge n berechnet man nun mittels der Polynom–Zeit Reduktion y. Da pro Rechenschritt in der Reduktion höchstens 1 Symbol ausgeschrieben werden kann, ist $\mid y \mid \leq p_1(n)$. Der Test "$y \in L$" kann in Zeit $p_2(p_1(n))$ ausgeführt werden. Um also "$x \in L'$" zu entscheiden, braucht man $p_1(n) + p_2(p_1(n))$ Zeit; das ist polynomial in n. Somit ist $L' \in \mathcal{P}$.

Der Beweis von (b) verläuft analog. ∎

Def. 5.1.5: Ein *log–Raum Übersetzer* ist eine (mehrbändige) immer haltende Turing–Maschine mit einem read–only Eingabeband, einem write–only Ausgabeband, auf dem der LS–Kopf niemals nach links bewegt wird, und $\log n$ Arbeitsspeicher (Bandzellen) für eine Eingabe der Länge n.
Eine Sprache L' ist *log–Raum reduzierbar auf* L, wenn es einen log–Raum Übersetzer gibt, welcher für jede Eingabe x eine Ausgabe y erzeugt, sodaß $x \in L' \iff y \in L$. ∎

Satz 5.1.2: Sei L' log–Raum reduzierbar auf L. Dann
(a) $L \in \mathcal{P} \Longrightarrow L' \in \mathcal{P}$,
(b) $L \in \text{DSPACE}(\log^k n) \Longrightarrow L' \in \text{DSPSACE}(\log^k n)$.

Beweis: (a) Wenn wir zeigen können, daß eine log–Raum Reduktion nicht mehr als polynomiale Zeit braucht, dann folgt die Behauptung aus Satz 5.1.1(a). Da das Ausgabeband die Berechnung nicht beeinflussen kann, ist das Produkt P der Anzahl der Zustände $\mid Q \mid$, Speicherbandinhalte und Positionen der Eingabe und Speicher LS–Köpfe eine obere Schranke für die Anzahl der Rechenschritte, die ausgeführt werden können, bevor der log–Raum Übersetzer in eine Schleife kommt, was per Definition ausgeschlossen ist. Hat nun das Speicherband die Länge $\log n$ und ist A das Bandalphabet, so ist

$$P = |\,Q\,| \cdot |\,A\,|^{\log n} \cdot n \cdot \log n \leq |\,Q\,| \cdot n^{\log|A|} \cdot n \cdot n = |\,Q\,| \cdot n^{2+\log|A|} \ .$$

Die Anzahl der Schritte in der Übersetzung ist also polynomial in der Länge n der Eingabe.

(b) Dazu verweisen wir auf [HU] Lemma 13.2(c). ∎

5.2. Vollständige Probleme

Wie bereits oben erwähnt, ist nicht bekannt, ob $\mathcal{P} = \mathcal{NP}$. Wollte man etwa versuchen $\mathcal{P} \neq \mathcal{NP}$ zu zeigen, so liegt es nahe nach einem "schwierigsten" Problem in $\mathcal{NP}$ zu suchen. Dieses wäre ein geeigneter Kandidat für $\mathcal{NP} \setminus \mathcal{P}$. Intuitiv könnte man etwa sagen, eine Sprache L_0 ist schwierig, wenn jede andere Sprache in $\mathcal{NP}$ auf L_0 reduzierbar ist bezüglich einer einfach zu berechnenden Reduktion. Ob eine Sprache "schwierig" ist, hängt natürlich von der Art der Reduktion ab.

Def. 5.2.1: Sei $\mathcal{C}$ eine Klasse von Sprachen. Eine Sprache L ist *schwierig für $\mathcal{C}$ bezüglich Polynom–Zeit (*bzw. *log–Raum) Reduktion*, wenn jede Sprache in $\mathcal{C}$ Polynom–Zeit (bzw. log–Raum) reduzierbar ist auf L. (L ist nicht notwendigerweise in $\mathcal{C}$.)

L ist *vollständig für $\mathcal{C}$ bezüglich Polynom–Zeit (*bzw. *log–Raum) Reduktion*, wenn L schwierig ist für $\mathcal{C}$ bzgl. Polynom–Zeit (bzw. log–Raum) Reduktion und $L \in \mathcal{C}$.

Für zwei häufig verwendete Begriffe führen wir Abkürzungen ein: L ist *NP–vollständig (NP–schwierig)* falls L vollständig (schwierig) ist für $\mathcal{NP}$ bzgl. log–Raum Reduktion [2]. L ist *PSPACE–vollständig (PSPACE–schwierig)*, wenn L vollständig (schwierig) ist für PSPACE bzgl. Polynom–Zeit Reduktion. ■

Viele NP–vollständige Probleme sind sehr natürlich und effiziente Lösungen sind seit langer Zeit gesucht worden. Für keines dieser Probleme ist aber eine polynomiale Lösung bekannt. Wäre eines dieser Probleme in $\mathcal{P}$, so wären sie alle in $\mathcal{P}$ (und überdies $\mathcal{P} = \mathcal{NP}$). Daß eines dieser Probleme eine polynomiale Lösung hat, ist also sehr unwahrscheinlich. Im folgenden gehen wir auf ein NP–vollständiges Problem, das Erfüllbarkeitsproblem, näher ein.

Def. 5.2.2: Ein *Boolescher Ausdruck E* ist auf die übliche Weise aufgebaut aus *Variablen* $x_1, x_2, \ldots$, den *Konstanten* `T` (wahr) und `F` (falsch), Klammern, und den *Operatoren* $\neg$ (nicht), $\wedge$ (und) und $\vee$ (oder). Anstatt $\neg x$ schreiben wir oft $\overline{x}$ für eine Variable x.

Eine *Klause* ist ein Boolescher Ausdruck der Form $\alpha_1 \vee \ldots \vee \alpha_k$, wobei jedes α_i, $1 \leq i \leq k$, ein *Literal* ist, also entweder eine Konstante oder x oder $\neg x$ für eine Variable x.

---

[2] Wir folgen dabei [HU].

Ein Boolescher Ausdruck E ist in *konjunktiver Normalform (CNF)*, wenn er von der Gestalt $E_1 \wedge \ldots \wedge E_l$ ist, wobei jeder Boolescher Ausdruck E_i, $1 \leq i \leq l$, eine Klause ist. E ist in *3–CNF*, wenn jede Klause genau 3 Literale enthält. ■

Def. 5.2.3: Ein Boolescher Ausdruck E heißt *erfüllbar*, wenn es eine Zuweisung von $\{\mathtt{T}, \mathtt{F}\}$ auf die Variablen in E gibt, sodaß E den Wert $\mathtt{T}$ erhält. Das *Erfüllbarkeitsproblem* besteht darin, für einen Boolescher Ausdruck festzustellen, ob er erfüllbar ist. ■

Beispiel 5.2.1: Der Boolesche Ausdruck $(x_1 \vee x_2) \wedge (\neg x_2 \vee x_1)$ ist erfüllbar; eine erfüllende Zuweisung ist etwa $\{\mathtt{T} \rightarrow x_1, \mathtt{F} \rightarrow x_2\}$. Dagegen ist der Boolesche Ausdruck $x_1 \wedge \neg x_2 \wedge (\neg x_1 \vee x_2)$ unerfüllbar. ■

Def. 5.2.4: Dem Erfüllbarkeitsproblem kann man auf folgende Weise eine Sprache L_{sat} (satisfiability) zuordnen. Als Alphabet wählt man $\{\mathtt{T}, \mathtt{F}, x, \neg, \wedge, \vee, 0, 1, (,)\}$. Die Funktion *code* bildet ab

$$\begin{array}{rcl} \mathtt{T}, \mathtt{F}, \neg, \wedge, \vee, (,) & \mapsto & \mathtt{T}, \mathtt{F}, \neg, \wedge, \vee, (,), \\ x_i & \mapsto & x b_1 \ldots b_r, \\ & & \text{wobei } b_1 \ldots b_r \text{ die Binärdarstellung von } i \text{ ist.} \end{array}$$

code wird auf natürliche Weise auf Boolesche Ausdrücke ausgedehnt.

$$L_{sat} = \{code(E) \mid E \text{ ist ein erfüllbarer Boolescher Ausdruck}\}.$$ ■

Beispiel 5.2.2: So ist etwa $(x1 \vee x10) \wedge (\neg x10 \vee x1) \in L_{sat}$ und $x1 \wedge \neg x10 \wedge (\neg x1 \vee x10) \notin L_{sat}$. ■

Ist E ein Boolescher Ausdruck der Länge n, so ist die Länge von $code(E) \leq \lceil n \log n \rceil$. Im folgenden werden wir nicht unterscheiden zwischen der Länge n von E und der Länge $n \log n$ von $code(E)$. Dadurch können aber keine Probleme auftreten, da wir log–Raum Reduktionen betrachten werden und $\log(n \log n) \leq 2 \cdot \log n$.

Satz 5.2.1 (Satz von Cook [3]): L_{sat} (also das Erfüllbarkeitsproblem) ist NP–vollständig.

[3] S.A. Cook: *"The complexity of theorem proving procedures", Proc. 3rd Annual ACM Symposium on the Theory of Computing,* 151–158 (1971).

Beweis: Zunächst ist natürlich klar, daß $L_{sat} \in \mathcal{NP}$. Um festzustellen, ob ein Ausdruck der Länge n erfüllbar ist, wählt man nichtdeterministisch Werte für alle Variablen und evaluiert den Ausdruck. Zur Evaluierung braucht man nur polynomiale Zeit.

Um zu zeigen, daß jede Sprache in $\mathcal{NP}$ auf L_{sat} reduzierbar ist, geben wir für jede nichtdeterministische Turing–Maschine M, deren Zeitkomplexität ein Polynom $p(n)$ ist, einen log–Raum Algorithmus $A_{p(n)}$ an, der für ein Eingabewort x einen Booleschen Ausdruck E_x berechnet, der genau dann erfüllbar ist, wenn M das Wort x akzeptiert. Wir geben nun an, wie dieser Ausdruck E_x erzeugt wird.

Sei $\#\beta_0\#\beta_1\#\cdots\#\beta_{p(n)}$ eine Berechnung von M. Dabei sei jedes β_i eine Konfiguration von M mit genau $p(n)$ Symbolen (wenn notwendig fügt man rechts zum letzten Bandsymbol noch einige Blanks hinzu). Wird die Eingabe bereits vor dem $p(n)$–ten Schritt akzeptiert, so gestatten wir eine Wiederholung der akzeptierenden Konfiguration, sodaß jede Berechnung aus genau $p(n) + 1$ Konfigurationen besteht. In jeder Konfiguration fassen wir den Zustand und das aktuelle Symbol zu einem einzigen Symbol zusammen. Dieses zusammengesetzte Symbol in der i–ten Konfiguration enthalte zusätzlich eine Nummer m, welche die nichtdeterministische Wahl anzeigt, durch welche die $(i + 1)$–te aus der i–ten Konfiguration entsteht. Dazu werden die nichtdeterministischen Wahlmöglichkeiten von M bei gegebenem Zustand und Bandsymbol durchnummeriert.

Für jedes Symbol X, das in der Berechnung vorkommen kann, und für jedes i, $0 \le i < (p(n) + 1)^2$, definieren wir eine Boolesche Variable c_{iX}, welche angibt, ob X das i–te Symbol in der Berechnung ist. (Das 0–te Symbol ist #.)

Wir konstruieren nun den Ausdruck E_x so, daß er genau dann für eine gegebene Zuweisung zu den Variablen c_{iX} wahr ist, wenn die c_{iX}'s mit Wert `T` einer akzeptierenden Berechnung entsprechen. Der Ausdruck E_x sagt folgendes aus:

(1) Die c_{iX}'s mit Wert `T` entsprechen einer Folge von Symbolen, für jedes i ist also genau ein c_{iX} wahr.
(2) Die Konfiguration β_0 ist eine initiale Konfiguration von M mit Eingabe x.
(3) Die letzte Konfiguration enthält einen Endzustand.
(4) Jede Konfiguration folgt aus der vorangegangenen durch den angegebenen Schritt von M.

Der Ausdruck E_x ist die Konjunktion von vier Ausdrücken, deren jeder eine dieser Bedingungen erzwingt. Der Ausdruck, der (1) erzwingt, ist

$$\bigwedge_i \Big[\bigvee_X c_{iX} \wedge \neg\Big(\bigvee_{X \ne Y} (c_{iX} \wedge c_{iY})\Big)\Big].$$

Sei $x = a_1 a_2 \cdots a_n$. Der Ausdruck, der (2) erzwingt, ist seinerseits die Konjunktion der folgenden Unterausdrücke (i) — (iv).

(i) $c_{0\#} \wedge c_{p(n)+1,\#}$. Die Symbole an der 0–ten und $(p(n)+1)$–ten Stelle sind #.

(ii) $c_{1Y_1} \vee c_{1Y_2} \vee \cdots \vee c_{1Y_k}$, wobei $Y_1, Y_2, \ldots, Y_k$ alle die zusammengesetzten Symbole sind, welche das Bandsymbol a_1, den Anfangszustand q_0, und die Nummer eines erlaubten Schrittes von M beim Zustand q_0 und aktuellen Symbol a_1 darstellen. Dieser Teilausdruck behauptet, daß das erste Symbol von β_0 korrekt ist.

(iii) $\bigwedge_{2 \leq i \leq n} c_{ia_i}$. Die Symbole von der zweiten bis zur n–ten Stelle von β_0 sind korrekt.

(iv) $\bigwedge_{n < i \leq p(n)} c_{i\sqcup}$. Die restlichen Symbole von β_0 sind Blank.

Die Formel, welche (3) erzwingt, ist

$$\bigvee_{p(n)(p(n)+1) < i < (p(n)+1)^2} \Big(\bigvee_{X \in F} c_{iX} \Big),$$

wobei F die Menge der zusammengesetzten Symbole ist, welche einen Endzustand enthalten.

Wir kommen nun zum Ausdruck, welcher (4) erzwingen soll. Jedes Symbol in β_i kann hergeleitet werden vom entsprechenden Symbol in β_{i-1} und den Symbolen links und rechts davon (wovon eines # sein kann). Enthält die vorangegangene Konfiguration einen Endzustand, so sind die gegenwärtige und die vorangegangene Konfiguration gleich. Es ist also nicht schwierig ein Prädikat $f(W, X, Y, Z)$ zu definieren, das genau dann erfüllt ist, wenn das Symbol Z in Position j einer Konfiguration vorkommen kann, vorausgesetzt daß W, X und Y die Symbole in den Positionen $j-1$, j und $j+1$ der vorangegangenen Konfiguration sind (W ist # falls $j = 1$ und Y ist # falls $j = p(n)$). Der Einfachheit halber sei auch $f(W, \#, X, \#)$ erfüllt; auf diese Weise können die Markierungen zwischen den Konfigurationen genau so behandelt werden wie die Symbole in den Konfigurationen. Wir erhalten somit für (4) den Ausdruck

$$\bigwedge_{p(n) < j < (p(n)+1)^2} \Big(\bigvee_{\substack{W,X,Y,Z \text{ sodass} \\ f(W,X,Y,Z)}} (c_{j-p(n)-2,W} \wedge c_{j-p(n)-1,X} \wedge c_{j-p(n),Y} \wedge c_{jZ}) \Big).$$

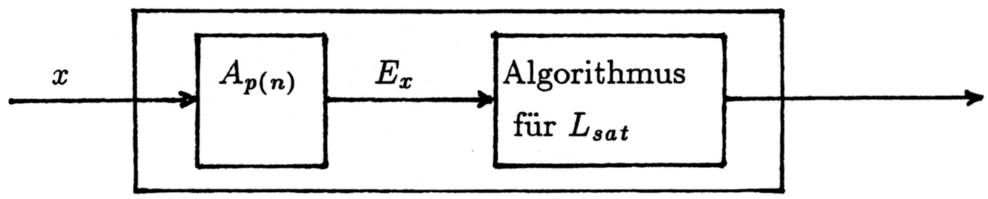
Ausgehend von einer akzeptierenden Berechnung von M auf x ist es leicht, Wahrheitswerte für die Variablen c_{iX} zu finden, welche den Ausdruck E_x wahr machen. Ist umgekehrt eine Zuordnung von Wahrheitswerten gegeben, welche E_x wahr macht, so garantieren die oben definierten vier Ausdrücke, daß eine akzeptierende Berechnung von M auf x existiert.

Der Ausdruck E_x hat die Länge $\mathcal{O}(p^2(n))$ und ist einfach genug, daß ihn eine log–Raum Turing–Maschine mit Eingabe x erzeugen kann. Diese Turing–Maschine braucht nur genügend Speicherplatz, um bis $(p(n)+1)^2$ zählen zu können. Wegen $\log(p(n)) = c \cdot \log n$ für eine Konstante c, genügen dazu $\mathcal{O}(\log n)$ Speicherzellen.

Somit haben wir gezeigt, daß jede Sprache in $\mathcal{NP}$ log–Raum reduzierbar ist auf L_{sat}, also L_{sat} NP–vollständig ist. ∎

Man kann zeigen, daß L_{csat}, das Erfüllbarkeitsproblem für CNFs, NP–vollständig ist, ja sogar L_{3-csat}, das Erfüllbarkeitsproblem für 3–CNFs.

Hätten wir also einen deterministischen Algorithmus mit polynomialer Zeitkomplexität zur Entscheidung von L_{sat}, so könnten wir ihn dazu benutzen, jede Sprache in $\mathcal{NP}$ in polynomialer Zeit zu erkennen. Sei L eine Sprache, die von einer nichtdeterministischen Turing–Maschine M mit Zeitkomplexität $p(n)$ erkannt wird und sei $A_{p(n)}$ der im Beweis zu Satz 5.2.1 definierte log–Raum (also Polynom–Zeit) Übersetzer. Ein deterministischer Polynom–Zeit Algorithmus zur Erkennung von L wäre dann



Aus der Existenz eines deterministischen Polynom–Zeit Algorithmus für L_{sat} würde also $\mathcal{P} = \mathcal{NP}$ folgen.

Einige weitere NP–vollständige Probleme sind

— das Problem der Hamiltonschen Zyklen;

— das Problem der *ganzzahligen linearen Programmierung*: für $A \in \mathbf{Z}^{m \times n}, b \in \mathbf{Z}^m$, entscheide ob es einen Vektor $x \in \mathbf{Z}^n$ gibt, sodaß $Ax \geq b$;

— das *chromatische Zahlenproblem*: für einen gegebenen Graphen G und eine ganze Zahl k, entscheide ob G mit k Farben gefärbt werden kann, sodaß keine zwei benachbarten Ecken dieselbe Farbe haben;

— das *Problem des Handlungsreisenden*: zu einem gegebenen Graphen mit gewichteten Kanten bestimme den Hamiltonschen Zyklus mit kleinstem Gewicht. Um das Problem als Sprache ausdrücken zu können, verlangt man, daß die Gewichte ganze Zahlen seien und fragt dann, ob es einen Hamiltonschen Zyklus mit Gewicht $\leq k$ gibt.

— Das *Partitionsproblem*: existiert zu einer gegebenen Liste von ganzen Zahlen $i_1, i_2, \ldots, i_k$ eine Unterliste, deren Summe $\frac{1}{2}(i_1 + i_2 + \cdots + i_k)$ ist. Man beachte, daß die Länge einer Instanz nicht $i_1 + i_2 + \cdots + i_k$ ist, sondern die Summe der Längen der i_j's.

Für eine intensive Untersuchung dieser und vieler anderer NP–vollständiger Probleme verweisen wir auf das Buch von Garey und Johnson [5].

Man kann sich dem $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ Problem auch auf folgende Weise nähern.

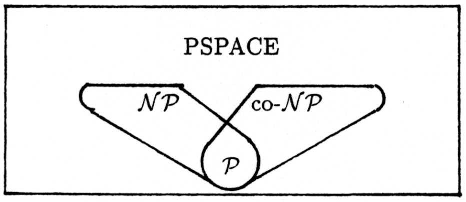
Def. 5.2.5 Die Komplexitätsklasse co-$\mathcal{NP}$ enthält alle diejenigen Sprachen, deren Komplement in $\mathcal{NP}$ ist, also co–$\mathcal{NP} = \{L | \overline{L} \in \mathcal{NP}\}$. ∎

Es ist nicht bekannt, ob $\mathcal{NP}$ abgeschlossen ist bezüglich der Bildung des Komplements, also ob co–$\mathcal{NP} = \mathcal{NP}$. Sollte das nicht der Fall sein, so wäre $\mathcal{P} \neq \mathcal{NP}$, denn $\mathcal{P}$ ist abgeschlossen bezüglich der Bildung des Komplements. Von keinem NP–vollständigen Problem weiß man, daß sein Komplement ebenfalls in $\mathcal{NP}$ ist. Um etwa die Nichterfüllbarkeit eines booleschen Ausdrucks mit n Variablen zu bestimmen, scheint es notwendig zu sein alle 2^n Zuweisungen zu testen, auch wenn der Algorithmus nicht–deterministisch ist. Sollte von einem NP–vollständigen Problem entdeckt werden, daß auch sein Komplement in $\mathcal{NP}$ gibt, so wäre $\mathcal{NP}$ abgeschlossen bezüglich der Bildung des Komplements.

Satz 5.2.2 $\mathcal{NP}$ ist abgeschlossen bezüglich Bildung des Komplements genau dann, wenn es ein NP–vollständiges Problem gibt, dessen Komplement in $\mathcal{NP}$ ist.

Beweis: Die Implikation "$\Longrightarrow$" ist offensichtlich. Für die Implikation "$\Longleftarrow$" sei S ein NP–vollständiges Problem. Wir nehmen an, $\overline{S}$ wäre in $\mathcal{NP}$. Da jedes L in $\mathcal{NP}$ log–Raum reduzierbar ist auf S, ist jedes $\overline{L}$ log–Raum reduzierbar auf $\overline{S}$. Also ist $\overline{L}$ in $\mathcal{NP}$. ∎

Die Komplexitätsklassen $\mathcal{P}$, $\mathcal{NP}$, co-$\mathcal{NP}$, PSPACE stehen also in folgender Relation



[5] M.R. Garey und D.S. Johnson: *"Computers and Intractability: A Guide to the Theory of NP–Completeness"*, H. Freeman, San Francisco (1978).

5.3. The P versus NP Problem (by S. Cook)

Der folgende Artikel von S. Cook ist eine in [Coo] veröffentlichte Beschreibung eines der wichtigsten offenen Probleme in Mathematik und Informatik.

The P versus NP Problem

Stephen Cook
University of Toronto

1. Statement of the Problem

The **P** versus **NP** problem is to determine whether every language accepted by some nondeterministic algorithm in polynomial time is also accepted by some (deterministic) algorithm in polynomial time. To define the problem precisely it is necessary to give a formal model of a computer. The standard computer model in computability theory is the Turing machine, introduced by Alan Turing in 1936 [Tur36]. Although the model was introduced before physical computers were built, it nevertheless continues to be accepted as the proper computer model for the purpose of defining the notion of *computable function*.

Informally the class **P** is the class of decision problems solvable by some algorithm within a number of steps bounded by some fixed polynomial in the length of the input. Turing was not concerned with the efficiency of his machines, but rather his concern was whether they can simulate arbitrary algorithms given sufficient time. However it turns out Turing machines can generally simulate more efficient computer models (for example machines equipped with many tapes or an unbounded random access memory) by at most squaring or cubing the computation time. Thus **P** is a robust class, and has equivalent definitions over a large class of computer models. Here we follow standard practice and define the class **P** in terms of Turing machines.

Formally the elements of the class **P** are languages. Let Σ be a finite alphabet (that is, a finite nonempty set) with at least two elements, and let Σ^* be the set of finite strings over Σ. Then a *language over* Σ is a subset L of Σ^*. Each Turing machine M has an associated *input alphabet* Σ. For each string w in Σ^* there is a computation associated with M with input w. (The notions of Turing machine and computation are defined formally in the appendix.) We say that M *accepts* w if this computation terminates in the accepting state. Note that M fails to accept w either if this computation ends in the rejecting state, or if the computation fails to terminate. The *language accepted by* M, denoted $L(M)$, has associated alphabet Σ and is defined by

$$L(M) = \{w \in \Sigma^* \mid M \text{ accepts } w\}$$

We denote by $t_M(w)$ the number of steps in the computation of M on input w (see the Appendix). If this computation never halts, then $t_M(w) = \infty$. For $n \in \mathbb{N}$ we denote by $T_M(n)$ the *worst case run time* of M; that is

$$T_M(n) = \max\{t_M(w) \mid w \in \Sigma^n\}$$

where Σ^n is the set of all strings over Σ of length n. We say that M *runs in polynomial time* if there exists k such that for all n, $T_M(n) \leq n^k + k$. Now we define the class $\mathbf{P}$ of languages by

$$\mathbf{P} = \{L \mid L = L(M) \text{ for some Turing machine } M \text{ which runs in polynomial time}\}$$

The notation $\mathbf{NP}$ stands for "nondeterministic polynomial time", since originally $\mathbf{NP}$ was defined in terms of nondeterministic machines (that is, machines that have more than one possible move from a given configuration). However now it is customary to give an equivalent definition using the notion of a *checking relation*, which is simply a binary relation $R \subseteq \Sigma^* \times \Sigma_1^*$ for some finite alphabets Σ and Σ_1. We associate with each such relation R a language L_R over $\Sigma \cup \Sigma_1 \cup \{\#\}$ defined by

$$L_R = \{w\#y \mid R(w, y)\}$$

where the symbol $\#$ is not in Σ. We say that R is *polynomial-time* iff $L_R \in \mathbf{P}$.

Now we define the class $\mathbf{NP}$ of languages by the condition that a language L over Σ is in $\mathbf{NP}$ iff there is $k \in \mathbb{N}$ and a polynomial-time checking relation R such that for all $w \in \Sigma^*$,

$$w \in L \iff \exists y(|y| \leq |w|^k \text{ and } R(w, y))$$

where $|w|$ and $|y|$ denote the lengths of w and y, respectively.

Problem Statement: Does $\mathbf{P} = \mathbf{NP}$?

It is easy to see that the answer is independent of the size of the alphabet Σ (we assume $|\Sigma| \geq 2$), since strings over an alphabet of any fixed size can be efficiently coded by strings over a binary alphabet. (For $|\Sigma| = 1$ the problem is still open, although it is possible that $\mathbf{P} = \mathbf{NP}$ in this case but not in the general case.)

It is trivial to show that $\mathbf{P} \subseteq \mathbf{NP}$, since for each language L over Σ, if $L \in \mathbf{P}$ then we can define the polynomial-time checking relation $R \subseteq \Sigma^* \cup \Sigma^*$ by

$$R(w, y) \iff w \in L$$

for all $w, y \in \Sigma^*$.

Here are two simple examples, using decimal notation to code natural numbers: The set of perfect squares is in $\mathbf{P}$ and the set of composite numbers is in $\mathbf{NP}$ (and not known to be in $\mathbf{P}$). For the latter, the associated polynomial time checking relation R is given by

$$R(\bar{a}, \bar{b}) \iff 1 < b < a \text{ and } b|a \tag{1}$$

In general the decimal notation for a natural number c is denoted by $\bar{c}$.

2. History and Importance

The importance of the $\mathbf{P}$ vs $\mathbf{NP}$ questions stems from the successful theories of $\mathbf{NP}$-completeness and complexity-based cryptography, as well as the potentially stunning practical consequences of a constructive proof of $\mathbf{P} = \mathbf{NP}$.

The theory of $\mathbf{NP}$-completeness has its roots in computability theory, which originated in the work of Turing, Church, Gödel, and others in the 1930's. The computability precursors of the classes $\mathbf{P}$ and $\mathbf{NP}$ are the classes of decidable and c.e. (computably enumerable) languages, respectively. We say that a language L is c.e. (or *semi-decidable*) iff $L = L(M)$ for some Turing machine M. We say that L is *decidable* iff $L = L(M)$ for some Turing machine M which satisfies the condition that M halts on all input strings w. There is an equivalent definition of c.e. which brings out its analogy with $\mathbf{NP}$, namely L is c.e. iff there is a computable "checking relation" $R(x, y)$ such that $L = \{x \mid \exists y R(x, y)\}$.

Using the notation $\langle M \rangle$ to denote a string describing a Turing machine M, we define the Halting Problem HP as follows:

$$HP = \{\langle M \rangle \mid M \text{ is a Turing machine which halts on input } \langle M \rangle\}$$

Turing used a simple diagonal argument to show that HP is not decidable. On the other hand, it is not hard to show that HP is c.e.

Of central importance in computability theory is the notion of reducibility, which Turing defined roughly as follows: A language L_1 is *Turing reducible* to a language L_2 iff there is an oracle Turing machine M which accepts L_1, where M is allowed to make membership queries of the form $x \in L_2$? which are correctly answered by an "oracle" for L_2. Later the more restricted notion of many-one reducibility ($\leq_m$) was introduced and defined as follows.

Definition 1: Suppose that L_i is a language over Σ_i, $i = 1, 2$. Then $L_1 \leq_m L_2$ iff there is a (total) computable function $f : \Sigma_1^* \rightarrow \Sigma_2^*$ such that $x \in L_1 \iff f(x) \in L_2$, for all $x \in \Sigma_1^*$.

It is easy to see that if $L_1 \leq_m L_2$ and L_2 is decidable, then L_1 is decidable. This fact provides an important tool for showing undecidability; for example if $HP \leq_m L$ then L is undecidable.

The notion of **NP**-complete is based on the following notion from computability theory:

Definition 2: A language L is c.e.-*complete* iff L is c.e., and $L' \leq_m L$ for every c.e. language L'.

It is easy to show that HP is c.e.-complete. It turns out that most "natural" undecidable c.e. languages are in fact c.e.-complete. Since $\leq_m$ is transitive, to show that a c.e. language L is c.e. complete is suffices to show that $HP \leq_m L$.

The notion of polynomial-time computation was introduced in the 1960's by Cobham [Cob64] and Edmonds [Edm65] as part of the early development of computational complexity theory (although earlier von Neumann [vN53] in 1953 distinguished between polynomial time and exponential-time algorithms). Edmonds called polynomial-time algorithms "good algorithms", and linked them to tractable algorithms.

It has now become standard in complexity theory to identify polynomial-time with feasible, and here we digress to discuss this point. It is of course not literally true that every polynomial-time algorithm can be feasibly executed on small inputs; for example a computer program requiring n^{100} steps could never be executed on an input even as small as $n = 10$. Here is a more defensible statement (see [Coo91]):

Feasibility Thesis: A natural problem has a feasible algorithm iff it has a polynomial-time algorithm.

Examples of natural problems that have both feasible and polynomial-time algorithms abound: Integer arithmetic, linear algebra, network flow, linear programming, many graph problems (connectivity, shortest path, minimum spanning tree, bipartite matching) etc. On the other hand the deep results of Robertson and Seymour [RS95] provide a potential source of counterexamples to the thesis: They prove that every minor-closed family of graphs can be recognized in polynomial time (in fact in time $O(n^3)$), but the algorithms supplied by their method have such huge constants that they are not feasible. However each potential counter example can be removed by finding a feasible algorithm for it. For example a feasible recognition algorithm is known for the class of planar graphs, but none is currently known for the class of graphs embeddable in $\mathbb{R}^3$ with no two cycles linked. (Both examples are minor-closed families.) Of course the words "natural" and "feasible"in the thesis above should be explained; generally we do not consider a class with a parameter as natural, such as the set of graphs embeddable on a surface of genus k, $k > 1$.

We mention two concerns related to the "only if" direction of the thesis. The first comes from randomized algorithms. We discuss at the end of section 3 the possibility that a source of random bits might be used to greatly reduce the recognition time required for some language. Note however that it is not clear whether a truly random source exists in nature. The second concern comes from Quantum computers. This computer model incorporates the idea of superposition of states from quantum mechanics, and allows a potential exponential speed-up of some computations over Turing machines. For example, Shor [Sho97] has shown that some quantum computer algorithm is able to factor integers in polynomial time, but no polynomial-time integer factoring algorithm is known for Turing machines. However physicists have so far been unable to build a quantum computer that can handle more than a half-dozen bits, so this threat to the feasibility thesis is hypothetical at present.

Returning to the historical treatment of complexity theory, in 1971 the present author [Coo71] introduced a notion of **NP**-completeness as a polynomial-time analog of c.e. completeness, except that the reduction used was a polynomial- time analog of Turing reducibility rather than of many-one reducibility. The main results in [Coo71] are that several natural problems, including Satisfiability and 3-SAT (defined below) and subgraph isomorphism are **NP**-complete. A year later Karp [Kar72] used these completeness results to show that 20 other natural problems are **NP**-complete, thus forcefully demonstrating the importance of the subject. Karp also introduced the now standard notation **P** and **NP** and redefined **NP**-completeness using the polynomial-time analog of many-one reducibility, a definition which has become standard. Meanwhile Levin [Lev73] independently of Cook and Karp defined the notion of "universal search problem", similar to **NP**-complete problem, and gave six examples, including Satisfiability.

The standard definitions concerning **NP**-completeness are close analogs of Definitions 1 and 2 above.

Definition 3: Suppose that L_i is a language over Σ_i, $i = 1, 2$. Then $L_1 \leq_p L_2$ (L_1 is p-reducible to L_2) iff there is a polynomial-time computable function $f : \Sigma_1^* \to \Sigma_2^*$ such that $x \in L_1 \iff f(x) \in L_2$, for all $x \in \Sigma_1^*$.

Definition 4: A language L is **NP**-*complete* iff L is in **NP**, and $L' \leq_p L$ for every language L' in **NP**.

The following proposition is easy to prove: Part (b) uses the transitivity of $\leq_p$, and part (c) follows from part (a).

Proposition 1: (a) If $L_1 \leq_p L_2$ and $L_2 \in$ **P** then $L_1 \in$ **P**.

(b) If L_1 is **NP**-complete, $L_2 \in$ **NP**, and $L_1 \leq_p L_2$ then L_2 is **NP**-complete.

(c) If L is **NP**-complete and $L \in$ **P**, then **P**=**NP**.

Notice that parts (a) and (b) have close analogs in computability theory. The analog of part (c) is simply that if L is c.e.-complete then L is undecidable. Part (b) is the basic method for showing new problems are **NP**-complete, and part (c) explains why it is probably a waste of time looking for a polynomial-time algorithm for an **NP**-complete problem.

In practice a member of **NP** is expressed as a decision problem, and the corresponding language is understood to mean the set of strings coding YES instances to the decision problem using standard coding methods. Thus the problem Satisfiability is: Given a propositional formula F, determine whether F is satisfiable. To show that this is in **NP** we define the polynomial-time checking relation $R(x, y)$, which holds iff x codes a propositional formula F and y codes a truth assignment to the variables of F which makes F true. This problem was shown to be **NP**-complete in [Coo71] essentially by showing that for each polynomial-time Turing machine M which recognizes a checking relation $R(x, y)$ for an **NP** language L, there is a polynomial-time algorithm which takes as input a string x and produces a propositional formula F_x (describing the computation of M on input (x, y), with variables representing the unknown string y) such that F_x is satisfiable iff M accepts the input (x, y) for some y with $|y| \leq |x|^{O(1)}$.

An important special case of Satisfiability is 3-SAT, which was also shown to be **NP**-complete in [Coo71]. Instances of 3-SAT are restricted to formulas in conjunctive normal form with three literals per clause. For example, the formula

$$(P \vee Q \vee R) \wedge (\bar{P} \vee Q \vee \bar{R}) \wedge (P \vee \bar{Q} \vee S) \wedge (\bar{P} \vee \bar{R} \vee \bar{S}) \tag{2}$$

is a YES instance to 3-SAT since the truth assignment τ satisfies the formula, where $\tau(P) = \tau(Q) = True$ and $\tau(R) = \tau(S) = False$.

Many hundreds of **NP**-complete problems have been identified, including SubsetSum (given a set of positive integers presented in decimal notation, and a target T, is there a subset summing to T?), many graph problems (given a graph G, does G have a Hamiltonian cycle? Does G have a clique consisting of half of the vertices? Can the vertices of G be colored with three colors with distinct colors for adjacent vertices?). These problems give rise to many scheduling and routing problems with industrial importance. The book [GJ79] provides an excellent reference to the subject, with 300 **NP**-complete problems listed in the appendix.

Associated with each decision problem in **NP** there is a search problem, which is, given a string x, find a string y satisfying the checking relation $R(x, y)$ for the problem (or determine that x is a NO instance to the problem). Such a y is said to be a *certificate* for x. In the case of an **NP**-complete problem it is easy to see that the search problem can be efficiently reduced to the corresponding decision problem. In fact if $\mathbf{P} = \mathbf{NP}$ then the associated search problem for every **NP** problem has a polynomial-time algorithm. For example, an algorithm for the decision problem Satisfiability can be used to find a truth assignment τ satisfying a given satisfiable formula F by, for each variable P in F in turn, setting P to True in F or False in F, which ever case keeps F satisfiable.

The set of complements of **NP** languages is denoted co**NP**. The complement of an **NP**-complete language is thought not to be in **NP**; otherwise $\mathbf{NP} = \mathrm{co}\mathbf{NP}$. The set TAUT of tautologies (propositional formulas true under all assignments) is the standard example of a co**NP**-complete language. The conjecture $\mathbf{NP} \neq \mathrm{co}\mathbf{NP}$ is equivalent to the assertion that no formal proof system (suitably defined) for tautologies has short (polynomial-bounded) proofs for all tautologies [CR79]. This fact has motivated the development of a rich theory of proportional proof complexity [Kra95], one of whose goals is to prove superpolynomial lower bounds on the lengths of proofs for standard propositional proof systems.

There are interesting examples of **NP** problems not known to be either in **P** or **NP**-complete. One example is the set of composite numbers, mentioned in the first section, with checking relation (1). Here it is conjectured that there is no polynomial-time Turing reduction from the search problem (integer factoring) to the decision problem. Specifically, Miller [Mil76] showed how to determine in polynomial time whether a given number is composite, assuming the Extended Riemann Hypothesis, but a polynomial-time integer factoring algorithm is thought unlikely to exist. In fact an efficient factoring algorithm would break the RSA public key encryption scheme [ARS78] commonly used to allow (presumably) secure financial transactions over the internet.

The complement of the set of composite numbers (essentially the set of primes) was proved to be in **NP** by an interesting argument due to Pratt [Pra75], and hence is unlikely to be **NP**-complete.

There is an **NP** decision problem with complexity equivalent to that of integer factoring, namely

$$L_{fact} = \{\langle a, b\rangle \mid \exists d(1 < d < a \text{ and } d|b)\}$$

Given an integer $b > 1$, the smallest prime divisor of b can be found with about $\log_2 b$ queries to L_{fact}, using binary search. Using Pratt's theorem, it is easy to see that the complement of L_{fact} is also in **NP**: a certificate showing $\langle a, b \rangle$ is a non-instance of L_{fact} could be the complete prime decomposition of b, together with Pratt certificates proving that each of the prime factors is indeed prime. Thus it is considered unlikely that L_{fact} is **NP**-complete.

Computational complexity theory plays an important role in modern cryptography [Gol00]. The security of the internet, including most financial transactions, depend on complexity-theoretic assumptions such as the difficulty of integer factoring or of breaking DES (the Data Encryption Standard). If **P**= **NP** these assumptions are all false. Specifically, an algorithm solving 3-SAT in n^2 steps could be used to factor 200-digit numbers in a few minutes.

Although a practical algorithm for solving an **NP**-complete problem (showing **P** = **NP**) would have devastating consequences for cryptography, it would also have stunning practical consequences of a more positive nature, and not just because of the efficient solutions to the many **NP**-hard problems important to industry. For example, it would transform mathematics by allowing a computer to find a formal proof of any theorem which has a proof of reasonable length, since formal proofs can easily be recognized in polynomial time. Example theorems may well include all of the CMI prize problems. Although the formal proofs may not be initially intelligible to humans, the problem of finding intelligible proofs would be reduced to that of finding a recognition algorithm for intelligible proofs. Similar remarks apply to diverse creative human endeavors, such as designing airplane wings, creating physical theories, or even composing music. The question in each case is to what extent an efficient algorithm for recognizing a good result can be found. This is a fundamental problem in artificial intelligence, and one whose solution itself would be aided by the **NP**-solver by allowing easy testing of recognition theories.

Even if **P** $\neq$ **NP** it may still happen that every **NP** problem is susceptible to a polynomial-time algorithm which works on "most" inputs. This could render cryptography impossible and bring about most of the benefits of a world in which **P** = **NP**. This also motivates Levin's theory [Lev86, Imp95] of average case completeness, in which the **P** = **NP** question is replaced by the question of whether every **NP** problem with any reasonable probability distribution on its inputs can be solved in time polynomial on average.

In [Sma98] Smale lists the **P** vs **NP** question as problem 3 of mathematical problems for the next century. However Smale is interested not just in the classical version of the question, but also a version expressed in terms of the field of complex numbers. Here Turing machines must be replaced by a machine model that is capable of doing exact arithmetic and zero tests on arbitrary complex numbers. The **P** vs **NP** question is replaced by a question related to Hilbert's Nullstellensatz: Is there a polynomial-time algorithm which, given a set of k multivariate polynomials over $\mathbb{C}$, determines whether they have a common zero? See [BCSS98] for a development of complexity theory in this setting, including the intriguing result that the Mandelbrot set is undecidable.

The books by Papadimitriou [Pap94] and Sipser [Sip97] provide good introductions to mainstream complexity theory.

3. The Conjecture and Attempts to Prove it

Most complexity theorists believe that $\mathbf{P} \neq \mathbf{NP}$. Perhaps this can be partly explained by the potentially stunning consequences of $\mathbf{P} = \mathbf{NP}$ mentioned above, but there are better reasons. We explain these by considering the two possibilities in turn: $\mathbf{P} = \mathbf{NP}$ and $\mathbf{P} \neq \mathbf{NP}$.

Suppose first that $\mathbf{P} = \mathbf{NP}$ and consider how one might prove it. The obvious way is to exhibit a polynomial-time algorithm for 3-SAT or one of the other 1000 or so known **NP**-complete problems, and indeed many false proofs have been presented in this form. There is a standard toolkit available [CLR90] for devising polynomial-time algorithms, including the greedy method, dynamic programming, reduction to linear programming, etc. These are the subjects of a course on algorithms, typical in undergraduate computer science curriculums. Because of their importance in industry, a vast number of programmers and engineers have attempted to find efficient algorithms for **NP**-complete problems over the past 30 years, without success. There is similar strong motivation for breaking the cryptographic schemes which assume $\mathbf{P} \neq \mathbf{NP}$ for their security.

Of course it is possible that some nonconstructive argument, such as the Roberton/Seymour proofs mentioned earlier in conjunction with the Feasibility Thesis, could show that $\mathbf{P} = \mathbf{NP}$ without yielding any feasible algorithm for the standard **NP**-complete problems. However at present the best proven upper bound on an algorithm for solving 3-SAT is approximately 1.5^n, where

n is the number of variables in the input formula.

Suppose on the other hand that $\mathbf{P} \neq \mathbf{NP}$, and consider how one might prove it. There are two general methods that have been tried: diagonalization/reduction and Boolean circuit lower bounds.

The method of diaonalization with reduction has been used very successfully in computability theory to prove a host of problems undecidable, beginning with the Halting Problem. It has also been used successfully in complexity theory to prove super-exponential lower bounds for very hard decidable problems. For example, Presburger arithmetic, the first-order theory of integers under addition, is a decidable theory for which Fischer and Rabin [FR74] proved that any Turing machine deciding the theory must use at least $2^{2^{cn}}$ steps in the worst case, for some $c > 0$. In general, lower bounds using diagonalization and reduction relativize; that is they continue to apply in a setting in which both the problem instance and the solving Turing machine can make membership queries to an arbitrary oracle set A. However in [BGS75] it was shown that there is an oracle set A relative to which $\mathbf{P} = \mathbf{NP}$, suggesting that diagonalization/reduction cannot be used to separate these two classes. (There are nonrelativizing results in complexity theory, as will be mentioned below.) It is interesting to note that relative to a *generic* oracle, $\mathbf{P} \neq \mathbf{NP}$ [BI87, SCY97].

A *Boolean circuit* is a finite acyclic graph in which each non-input node, or *gate*, is labelled with a Boolean connective; typically from $\{AND, OR, NOT\}$. The input nodes are labeled with variables $x_1, ..., x_n$, and for each assignment of 0 or 1 to each variable the circuit computes a bit value at each gate, including the output gate, in the obvious way. It is not hard to see that if L is a language over $\{0, 1\}$ that is in $\mathbf{P}$, then there is a polynomial-size family of Boolean circuits $\langle B_n \rangle$ such that B_n has n inputs, and for each bit string w of length n, when w is applied to the n input nodes of B_n, then the output bit of B_n is 1 iff $w \in L$. In this case we say that $\langle B_n \rangle$ *computes* L.

Thus to prove $\mathbf{P} \neq \mathbf{NP}$ it suffices to prove a super-polynomial lower bound on the size of any family of Boolean circuits solving some specific $\mathbf{NP}$-complete problem, such as 3-SAT. Back in 1949 Shannon [Sha49] proved that for almost all Boolean functions $f : \{0, 1\}^n \rightarrow \{0, 1\}$, any Boolean circuit computing f requires at least $2^n/n$ gates. Unfortunately his counting argument gives no clue as to how to prove lower bounds for problems in $\mathbf{NP}$. Exponential lower bounds for **NP** problems have been proved for restricted circuit models, including monotone circuits [Raz85, AB87] and bounded depth circuits with unbounded fan-in gates [Has89, Smo87] (see [BS90]). However all attempts to find even super-linear lower bounds for unrestricted Boolean circuits for "explicitly given" Boolean functions have met with total failure; the best such lower bound proved so far is about $4n$. Razborov and Rudich [RR97] explain this failure by pointing out that all methods used so far can be classified as "Natural Proofs", and Natural Proofs for general circuit lower bounds are doomed to failure, assuming a certain complexity-theoretic conjecture asserting that strong pseudo-random number generators exist. Since such generators have been constructed assuming the hardness of integer factorization, we can infer the surprising result that a Natural Proof for a general circuit lower bound would give rise to a more efficient factoring algorithm than is currently known.

The failure of complexity theory to prove interesting lower bounds on a general model of computation is much more pervasive than the failure to prove $\mathbf{P} \neq \mathbf{NP}$. It is consistent with present knowledge that not only could Satisfiability have a polynomial-time algorithm, it could have a linear time algorithm on a multitape Turing machine. The same applies for all 21 problems mentioned in Karp's original paper [Kar72]. There are complexity class separations which we know exist but cannnot prove. For example, consider the sequence of complexity class inclusions

$$\mathbf{LOGSPACE} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE}$$

A simple diagonal arugment shows that the first is a proper subset of the last, but we cannot prove any particular adjacent inclusion is proper.

As another example, let **LINEAR-SIZE** be the class of languages over $\{0, 1\}$ which can be computed by a family $\langle B_n \rangle$ of Boolean circuits of size $O(n)$. It is not known whether either **P** or **NP** is a subset of **LINEAR-SIZE**, although Kannan [Kan82] proved that there are languages in the polynomial hierarchy (a generalization of **NP**) not in **LINEAR-SIZE**. Since if $\mathbf{P} = \mathbf{NP}$ then the polynomial hierarchy collapses to **P**, we have

Proposition 2: If $\mathbf{P} \subseteq$ **LINEAR-SIZE**, then $\mathbf{P} \neq \mathbf{NP}$.

This proposition could be interpreted as a method of proving $\mathbf{P} \neq \mathbf{NP}$, but a more usual belief is that the hypothesis is false.

A fundamental question in complexity theory is whether a source of random bits can be used to substantially speed up the recognition of some languages, provided one is willing to accept a small probability of error. The class **BPP** consists of all languages L which can be recognized by a randomized polynomial-time algorithm, with at most an exponentially small error probability on every input. Of course $\mathbf{P} \subseteq \mathbf{BPP}$, but it is not known whether the inclusion is proper. The set of prime numbers is in **BPP** [SS77], but it is not known to be in **P**. A reason for thinking that $\mathbf{BPP} = \mathbf{P}$ is that randomized algorithms are often successfully executed using a deterministic pseudo-random number generator as a source of "random" bits.

There is an indirect but intriguing connection between the two questions $\mathbf{P} = \mathbf{BPP}$ and $\mathbf{P} = \mathbf{NP}$. Let **E** be the class of languages recognizable in exponential time; that is the class of languages L such that $L = L(M)$ for some Turing machine M with $T_M(n) = O(2^{cn})$ for some $c > 0$. Let **A** be the assertion that some language in **E** requires exponential circuit complexity. That is

Assertion A: There is $L \in \mathbf{E}$ and $\epsilon > 0$ such that for every circuit family $\langle B_n \rangle$ computing L and for all sufficiently large n, B_n has at least $2^{\epsilon n}$ gates.

Proposition 3: If **A** then $\mathbf{BPP} = \mathbf{P}$. If **not A** then $\mathbf{P} \neq \mathbf{NP}$.

The first implication is a lovely theorem by Impagliazzo and Wigderson [IW97] and the second is an intriguing observation by V. Kabanets which strengthens Proposition 2. In fact Kabanets concludes $\mathbf{P} \neq \mathbf{NP}$ from a weaker assumption than **not A**; namely that every language in **E** can be computed by a Boolean circuit family $\langle B_n \rangle$ such that for at least one n, B_n has fewer gates than the maximum needed to compute any Boolean function $f : \{0,1\}^n \to \{0,1\}$. But there is no consensus on whether this hypothesis is true.

We should point out that Proposition 3 relativizes, and in particular relative to any **PSPACE**-complete oracle Assertion A holds and $\mathbf{BPP} = \mathbf{P} = \mathbf{NP}$. Thus a nonrelativizing construction will be needed if one is to prove $\mathbf{P} \neq \mathbf{NP}$ by giving small circuits for languages in **E**. However nonrelativizing constructions have been used successfully before, for example in showing **IP** (Interactive Polynomial time) contains all of **PSPSACE** [Sha92]. In this and other such constructions a key technique is to represent Boolean functions by multivariate polynomials over finite fields.

Appendix: Definition of Turing machine

A Turing machine M consists of a finite state control (i.e. a finite program) attached to read/write head moving on an infinite tape. The tape is divided into squares, each capable of storing one symbol from a finite alphabet Γ which includes the blank symbol b. Each machine M has a specified input alphabet Σ, which is a subset of Γ, not including the blank symbol b. At each step in a computation M is in some state q in a specified finite set Q of possible states. Initially a finite input string over Σ is written on adjacent squares of the tape, all other squares are blank (contain b), the head scans the left-most symbol of the input string, and M is in the initial state q_0. At each step M is in some state q and the head is scanning a tape square containing some tape symbol s, and the action performed depends on the pair (q, s) and is specified by the machine's transition function (or program) δ. The action consists of printing a symbol on the scanned square, moving the head left or right one square, and assuming a new state.

Formally a Turing machine M is a tuple $\langle \Sigma, \Gamma, Q, \delta \rangle$ where Σ, Γ, Q are finite nonempty sets with $\Sigma \subseteq \Gamma$ and $b \in \Gamma - \Sigma$. The state set Q contains three special states $q_0, q_{accept}, q_{reject}$. The *transition function* δ satisfies

$$\delta : (Q - \{q_{accept}, q_{reject}\}) \times \Gamma \to Q \times \Gamma \times \{-1, 1\}$$

If $\delta(q, s) = (q', s', h)$ the interpretation is that if M is in state q scanning the symbol s then q' is the new state, s' is the symbol printed, and the tape head moves left or right one square depending on whether h is -1 or 1.

We assume that the sets Q and Γ are disjoint.

A *configuration* of M is a string xqy with $x, y \in \Gamma^*$, y not the empty string, and $q \in Q$.

The interpretation of the configuration xqy is that M is in state q with xy on its tape, with its head scanning the left-most symbol of y.

If C and C' are configurations, then $C \xrightarrow{M} C'$ if $C = xqsy$ and $\delta(q, s) = (q', s', h)$ and one of the following holds:

$C' = xs'q'y$ and $h = 1$ and y is nonempty.

$C' = xs'q'b$ and $h = 1$ and y is empty.
$C' = x'q'as'y$ and $h = -1$ and $x = x'a$ for some $a \in \Gamma$.
$C' = q'bs'y$ and $h = -1$ and x is empty.

A configuration xqy is *halting* if $q \in \{q_{accept}, q_{reject}\}$. Note that for each non-halting configuration C there is a unique configuration C' such that $C \xrightarrow{M} C'$.

The *computation* of M on input $w \in \Sigma^*$ is the unique sequence $C_0, C_1, ...$ of configurations such that $C_0 = q_0 w$ (or $C_0 = q_0 b$ if w is empty) and $C_i \xrightarrow{M} C_{i+1}$ for each i with C_{i+1} in the computation, and either the sequence is infinite or it ends in a halting configuration. If the computation is finite, then the number of steps is one less than the number of configurations; otherwise the number of steps is infinite. We say that M *accepts* w iff the computation is finite and the final configuration contains the state q_{accept}.

Acknowledgments

My thanks to Avi Wigderson and Hugh Woodin for many helpful suggestions for improving an earlier version of this paper.

References

[AB87] N. Alon and R. B. Boppana. The monotone circuit complexity of boolean functions. *Combinatorica*, 7(1):1–22, 1987.

[ARS78] L. Adleman, R. L. Rivest, and A. Shamir. A method for obtaining digital signature and public-key cryptosystems. *Communication of the ACM*, 21(2), 1978.

[BCSS98] L. Blum, F. Cucker, M. Shub, and S. Smale. *Complexity and Real Computation.* Springer-Verlag, New York, 1998.

[BGS75] T. Baker, J. Gill, and R. Solovay. Relativizations of the P =? NP question. *SICOMP: SIAM Journal on Computing*, 1975.

[BI87] M. Blum and R. Impagliazzo. Generic oracles and oracle classes. In Ashok K. Chandra, editor, *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pages 118–126, Los Angeles, CA, October 1987. IEEE Computer Society Press.

[BS90] R. B. Boppana and M. Sipser. The complexity of finite functions. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 759–804. Elsevier and The MIT Press, 1990.

[CLR90] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw Hill, New York, 1990.

[Cob64] A. Cobham. The intrinsic computational difficulty of functions. In Yehoshua Bar-Hillel, editor, *Proceedings of the 1964 International Congress for Logic, Methodology, and Philosophy of Science*, pages 24–30. Elsevier/North-Holland, 1964.

[Coo71] Stephen Cook. The complexity of theorem-proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

[Coo91] Stephen Cook. Computational complexity of higher type functions. In Ichiro Satake, editor, *Proceedings of the International Congress of Mathematicians, Kyoto, Japan*, pages 55–69. Springer–Verlag, 1991.

[CR79] S. Cook and R. Reckhow. The relative efficiency of propositional proof systems. *J. Symbolic Logic*, 44(1), 1979.

[Edm65] Jack Edmonds. Minimum partition of a matroid into independent subsets. *J. Res. Nat. Bur. Standards Sect. B*, 69:67–72, 1965.

[FR74] M. J. Fischer and M. O. Rabin. Super-exponential complexity of Presburger arithmetic. In R. M. Karp, editor, *Complexity of Computation*, volume 7, pages 27–41. American Mathematical Society, Providence, RI, 1974.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractibility, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Co., San Francisco, 1979.

[Gol00] Oded Goldreich. *The Foundations of Cryptography–Volume 1*. Cambridge University Press, 2000.

[Has89] J. Hastad. Almost optimal lower bounds for small depth circuits. In S. Micali, editor, *Randomness and Computation, Advances in Computing Research, Vol. 5*, pages 143–170. JAI Press Inc., 1989.

[Imp95] R. Impagliazzo. A personal view of average-case complexity. In *10th IEEE Annual Conference on Structure in Complexity Theory*, pages 134–147, 1995.

[IW97] R. Impagliazzo and A. Wigderson. P = BPP if E requires exponential circuits: Derandomizing the XOR lemma. In *STOC: ACM Symposium on Theory of Computing (STOC)*, 1997.

[Kan82] Ravi Kannan. Circuit-size lower bounds and non-reducibility to sparse sets. *Information and Control*, 55:40–56, 1982.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, New York, 1972. Plenum Press.

[Kra95] J. Krajicek. *Bounded Arithmetic, Propositional Logic, and Complexity Theory.* Cambridge, 1995.

[Lev73] L. Levin. Universal search problems (in russian)". *Problemy Peredachi Informatsii*, 9(3):265–266, 1973. English translation in Trakhtenbrot, B. A.: A survey of Russian approaches to Perebor (brute-force search) algorithms. Annals of the History of Computing, 6 (1984), pages. 384-400.

[Lev86] L. Levin. Average case complete problems. *SIAM J. Computing*, 15:285–286, 1986.

[Mil76] G. L. Miller. Reimann's hypothesis and tests for primality. *J. Comput. System Sci*, 13:300–317, 1976.

[Pap94] Christos Papadimitriou. *Computational Complexity.* Addison–Wesley, 1994.

[Pra75] V. R. Pratt. Every prime has a succinct certificate. *SIAM Journal on Computing*, 4(3):214–220, 1975.

[Raz85] Alexander A. Razborov. Lower bounds on the monotone complexity of some boolean functions. *Soviet Math. Dokl.*, 31:354–357, 1985.

[RR97] Alexander A. Razborov and Steven Rudich. Natural proofs. *Journal of Computer and System Sciences*, 55(1):24–35, August 1997.

[RS95] N. Robertson and P. D. Seymour. Graph minors i–xiii. *Journal of Combinatorial Theory B*, 1983–1995.

[SCY97] R. Impagliazzo S. Cook and T. Yamakami. A tight relationship between generic oracles and type-2 complexity theory. *Information and Computation*, 137(2):159–170, 15 September 1997.

[Sha49] C. Shannon. The synthesis of two–terminal switching circuits. *Bell System Technical Journal*, 28:59–98, 1949.

[Sha92] Adi Shamir. Ip = pspace. *J.A.C.M.*, 39(4):869–877, 1992.

[Sho97] Peter Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Computing*, 26:1484–1509, 1997.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation*. PWS, 1997.

[Sma98] Steve Smale. Mathematical problems for the next century. *MATH-INT: The Mathematical Intelligencer*, 20, 1998.

[Smo87] R. Smolensky. Algebraic methods in the theory of lower bounds for boolean circuit complexity. In *ACM Symposium on Theory of Computing (STOC)*, volume 19, pages 77–82, 1987.

[SS77] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal on Computing*, 6(1):84–85, March 1977.

[Tur36] Alan Turing. On computable numbers with an application to the entscheidnungsproblem. *Proc. London Math. Soc. ser. 2*, 42:230–265, 1936-7.

[vN53] J. von Neumann. A certain zero-sum two-person game equivalent to the optimal assignment problem. In H. W. Kahn and A. W. Tucker, editors, *Contributions to the Theory of Games II*. Princeton Univ. Press, 1953.

Literatur

[AHU] A.V. Aho, J.E. Hopcroft, J.D. Ullman: *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, Massachusetts (1974)

[Cap] C.H. Cap: *Theoretische Grundlagen der Informatik*, Springer–Verlag Wien New York (1993)

[Coo] S. Cook: *The P versus NP Problem*, `http://www.claymath.org/prize_problems/p_vs_np.pdf` (2000)

[Dav] M.D. Davis: *Computability & Unsolvability*, McGraw–Hill, New York–Toronto–London (1958)

[DW] M.D. Davis, E.J. Weyuker: *Computability, Complexity, and Languages*, Academic Press, Orlando, Florida (1983)

[GJ] M.R. Garey, D.S. Johnson: *Computers and Intractability — A Guide to the Theory of NP–Completeness*, Freeman and Company, New York (1979)

[Har] J. Hartmanis (ed.): *Computational Complexity Theory*, Amer.Math.Soc. (1989)

[Hen] M. Hennessy: *The Semantics of Programming Languages*, J. Wiley (1990)

[HU] J.E. Hopcroft, J.D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, Addison–Wesley, Reading, Massachusetts (1979)

[Koz] D.C. Kozen: *Automata and Computability*, Springer-Verlag New York (1997)

[LP] H.R. Lewis, C.H. Papadimitriou: *Elements of the Theory of Computation*, Prentice–Hall, Englewood Cliffs, New Jersey (1981)

[Lip] J.D. Lipson: *Elements of Algebra and Algebraic Computing*, Addison–Wesley, Reading, Massachusetts (1981)

[Man] Z. Manna: *Mathematical Theory of Computation*, McGraw–Hill (1974)

[McN] R. McNaughton: *Elementary Computability, Formal Languages, and Automata*, Prentice–Hall, Englewood Cliffs, New Jersey (1982)

[Pep] P. Pepper: *Grundlagen der Informatik*, Oldenburg, München (1992)

[Rog] H. Rogers: *Theory of Recursive Functions and Effective Computability*, McGraw–Hill, New York, New York (1967)

[Sch] U. Schöning: *Theoretische Informatik — kurzgefaßt*, 2. Aufl., Spektrum Akad. Verlag, Heidelberg Berlin Oxford (1995)

[Sed] R. Sedgewick: *Algorithms*, Addison–Wesley, Reading, Massachusetts (1983)

[Weg] I. Wegener: *Theoretische Informatik*, Teubner, Stuttgart (1993)