Theoretische Informatik

Mitschrift zur Vorlesung von Prof. Dr. Raimund Seidel

Martin Grochulla martin@mgrochulla.de
Pascal Gwosdek pascal@planetk.de
Jan Hendrik Dithmar jh.dithmar@planetk.de

21. Februar 2006

Inhaltsverzeichnis

Kapitel 1

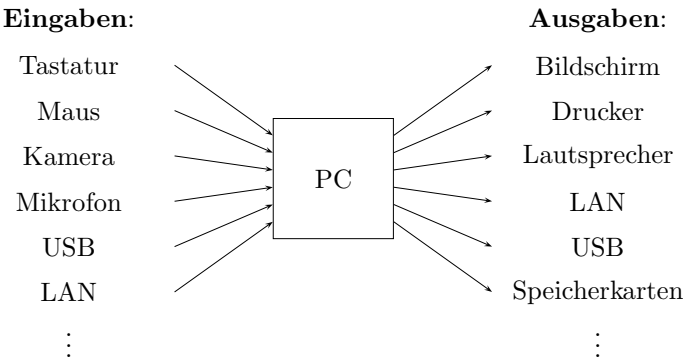
Rechnen „an sich“

1.1 Grundlegende Fragen

- Was ist Rechnen?
- Welche Rechenmechanismen gibt es?
- Kann man „alles“ berechnen?
- Wie „teuer“ ist das Rechnen?

1.2 Rechensituationen

i. PC



Auf bestimmte Eingaben sollen bestimmte Ausgaben erfolgen.
„Dauerschleife“.

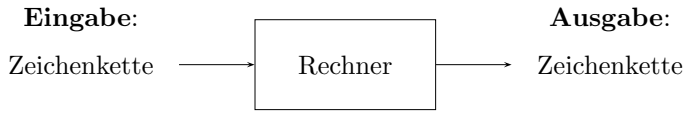
- schwer zu spezifizieren
- schwer zu analysieren
- schwierige Theoriebildung.

ii. Airbus A380

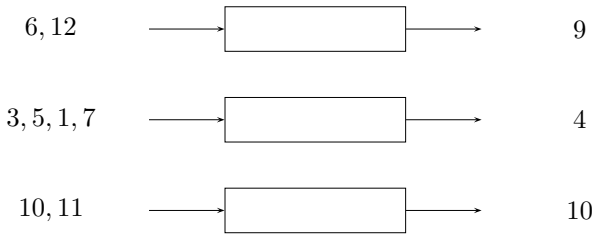
- Eingaben:
  - Sensoren
  - Steuereinheiten + Schalter
- Ausgaben:
  - Aktuatoren
  - Bildschirme
  - ...

Rechensystem: genau spezifiziertes Ein- und Ausgabeverhalten. „Dauerschleife“. $\Rightarrow$ nicht geeignet für Theoriebildung

iii. Einfache Situation



Beispiel:



Berechnung einer Funktion:

$$\text{Zeichenkette} \to \text{Zeichenkette}$$

Vereinfachung

Betrachte nur Funktionen

$$\text{Zeichenkette} \to \{ \underset{\text{Nein}}{0} , \underset{\text{Ja}}{1} \}.$$

- „Rechner“ „berechnet“ Funktion

$$F : \overset{\text{Zeichenketten}}{\overset{\|}{\Sigma^*}} \to \{0,1\}$$

- „Rechner“ erkennt (Sprache) Menge von Zeichenketten

$$\{w \in \Sigma^* | F(w) = 1\}$$

1.3 Kurzexkurs über „Zeichenketten“

Alphabet : Endliche Menge (von Symbolen)

$$\sum$$

Zeichenkette (über Σ, auch „string“ oder „Wort“): Endliche Folge von Symbolen aus Σ.

$$k \in \mathbb{N}. \quad \sum^k = \left\{ \text{strings } \underset{a_i \in \Sigma}{a_1 a_2 \dots a_k} \text{ über } \Sigma \text{ der Länge } k \right\}$$

$$\sum^0 = \{\varepsilon\} \quad \varepsilon : \text{„leeres Wort“, String der Länge 0}$$

$$\sum^* = \bigcup_{k \in \mathbb{N}} \sum^k \quad \text{alle Strings endlicher Länge}$$

Somit folgt:

$$|a_1 \dots a_k| = k$$

$$|\varepsilon| = 0$$

$$m \in \mathbb{N}. \quad a^m = \begin{cases} \varepsilon & m = 0 \\ a a^{m-1} & m > 0 \end{cases}$$

Konkatenation : „Verkettung“

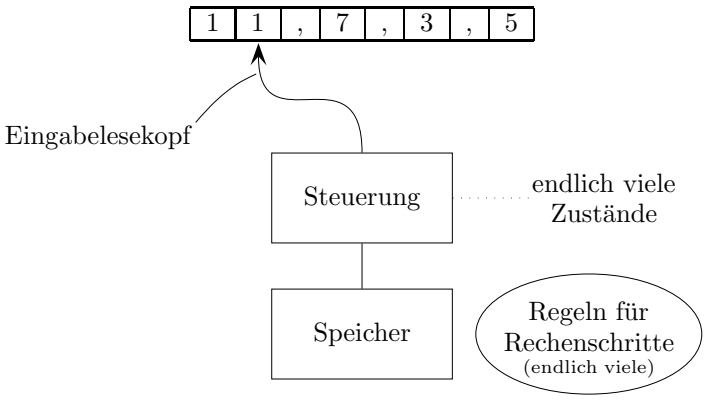
$$\begin{aligned} a &= a_1 a_2 ... a_k \in \Sigma^* \\ b &= b_1 b_2 ... b_l \in \Sigma^* \\ ab &= a_1 a_2 ... a_k b_1 ... b_l \\ |ab| &= |a| + |b| \\ a\varepsilon &= a = \varepsilon a \\ \left|\{\}^k\right| &= \begin{cases} 0 & k > 0 \\ 1 & k = 0 \end{cases} \\ \left|A^k\right| &= |A|^k \\ 0^0 &= 1 \end{aligned}$$

1.4 Automaten, Rechenmaschinen und Ideale Rechner

Wie sehen nun „Automaten“, auch „Rechenmaschinen“ oder „Ideale Rechner“, aus?

1.4.1 Übersicht

Eingabeband, unterteilt in **Zellen**, jede Zelle enthält ein Symbol der Eingabe:



Die Maschine macht eine Folge von „Rechenschritten“.
Ein **Schritt** ist abhängig vom

- gelesenen Symbol der Eingabe,
- Zustand der Steuerung und
- Inhalt des Speichers.

Abhängig davon kann man

- den Eingabekopf bewegen,
- in einen neuen Zustand gehen und
- den Inhalt des Speichers ändern.

Die Art der Operation ist durch die **Rechenschrittregeln** festgelegt.

1.4.2 Konfigurationen

Als **Konfiguration** bezeichnet man eine Momentaufnahme der Maschine, charakterisiert durch

- den Inhalt des Eingabebandes,
- die Position des Eingabekopfes,
- den Zustand der Steuerung und
- den Inhalt des Speichers.

1.4.3 Rechenschritt

Als **Rechenschritt** bezeichnet man einen Übergang von einer Konfiguration zu einer anderen. Dieser Übergang ist durch die Regeln vorgeschrieben.

1.4.4 Akzeptor

Sei durch start_w die Startkonfiguration für die Eingabe $w \in \Sigma^*$, ferner analog die Endkonfigurationen definiert. Der Automat „akzeptiert“ die Eingabe w, wenn er durch eine endliche Folge von Rechenschritten aus der Konfiguration start_w eine Endkonfiguration erreicht (**Determinismus**) bzw. erreichen kann (**Nichtdeterminismus**).

1.4.5 3 Arten von solchen Maschinen

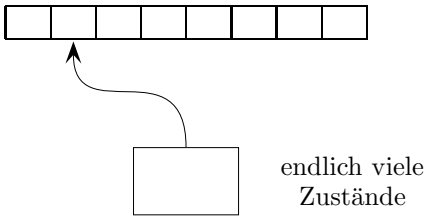
Wir betrachten drei Arten von Maschinen, charakterisiert durch die Art ihres Speichers:

i. kein Speicher (endlicher Automat)

ii. unbeschränktes Band von Zellen mit einem Schreiblesekopf (Turing Maschine)

iii. unbeschränktes Band von Zellen mit einem Schreiblesekopf, der immer am Ende des beschriebenen Teils agiert (Kellerautomat)

1.5 Endlicher Automat



Konfiguration = { Eingabe, Kopfposition, Zustand }

start_w:	Kopf ganz links in wohldefiniertem Startzustand
Endkonfigurationen:	Kopf ganz rechts in irgendeinem der definierten Endzustände

Pro Schritt wird der Kopf um eine Zelle weitergerückt.

1.5.1 Beispiel 1

Die Zustände des Automaten s_0, s_1, s_2 mit s_0 als Start- und s_2 als Endzustand.

Regeln

derzeitiger Zustand	gelesenes Symbol		neuer Zustand
s_0	a	$\rightarrow$	s_1
s_0	b	$\rightarrow$	s_0
s_1	a	$\rightarrow$	s_2
s_1	b	$\rightarrow$	s_0
s_2	a	$\rightarrow$	s_2
s_2	b	$\rightarrow$	s_0

Der Automat akzeptiert alle Strings über $\{\mathrm{a}, \mathrm{b}\}$ mit mindestens 2 a's am Ende.

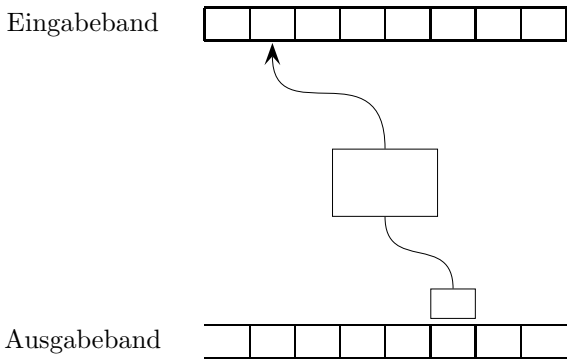
1.5.2 Beispiel 2

Wir möchten uns nun die nichtdeterministische Version ansehen.
Die Zustände des Automaten sind q_0, q_1, q_2 mit q_0 als Start- und q_2 als Endzustand.

Regeln

derzeitiger Zustand	gelesenes Symbol		neuer Zustand
q_0	a	$\rightarrow$	q_0
q_0	b	$\rightarrow$	q_0
q_0	a	$\rightarrow$	q_1
q_1	a	$\rightarrow$	q_2

1.6 Turing Maschine



Konfiguration = { Eingabe, Eingabekopfposition, Zustand, Arbeitsbandinhalt, Arbeitskopfposition }

start_w: Das Eingabeband enthält w.
Der Eingabekopf ganz ist „ganz links“ über x in einem wohldefinierten Startzustand.
Das Arbeitsband enthält nur leere Zellen.

Endkonfigurationen: Die Maschine ist in einem der „Endzustände“.

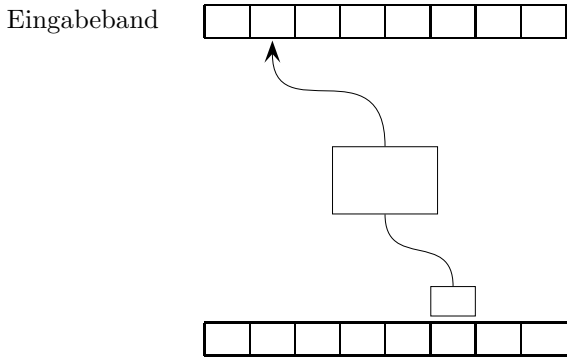
Pro Schritt bewegt sich der Eingabekopf um $\leqslant 1$ Zelle und der Arbeitskopf um $\leqslant 1$ Zelle. Die Zustände des Automaten sind s, l, r, f mit s als Start- und f als Endzustand. Das Zeichen ƀ stellt das Leerzeichen dar.

Regeln

derzeitiger Zustand	Symbol Eingabekopf	Symbol Ausgabekopf		neuer Zustand	Eingabekopfbewegung	Arbeitskopfsymbol	Arbeitskopfbewegung
s	a	ƀ	$\rightarrow$	s	$\rightarrow$	A	$\rightarrow$
s	\$	ƀ	$\rightarrow$	l	$\rightarrow$	ƀ	$\leftarrow$
l	b	A	$\rightarrow$	l	$\rightarrow$	A	$\leftarrow$
l	ƀ	ƀ	$\rightarrow$	f	$\downarrow$		$\downarrow$
l	b	ƀ	$\rightarrow$	r	$\downarrow$	ƀ	$\rightarrow$
r	b	A	$\rightarrow$	r	$\rightarrow$	A	$\rightarrow$
r	ƀ	ƀ	$\rightarrow$	f	$\downarrow$		$\downarrow$
r	b	ƀ	$\rightarrow$	l	$\downarrow$	ƀ	$\leftarrow$

Diese Turing Maschine akzeptiert alle Strings der Form $\mathrm{a}^k\$\mathrm{b}^{k \cdot l}$, $k \geqslant 1, l \geqslant 1$.

1.7 Keller-Automaten



Der Eingabekopf kann nicht nach links rücken. Immer, wenn der Arbeitskopf nach links rückt, hinterlässt er eine leere Zelle.

Konfiguration = $\begin{cases} \text{Eingabe} \\ \text{Eingabekopfposition} \\ \text{Zustand} \\ \text{Arbeitsbandinhalt} \end{cases}$

start_w: Das Eingabeband enthält w.
Der Eingabekopf ganz ist „ganz links“ über w in einem wohldefinierten Startzustand.
Das Arbeitsband enthält nur leere Zellen.

Endkonfigurationen: Der Eingabekopf ist zum Beispiel ganz rechts und das Arbeitsband leer.

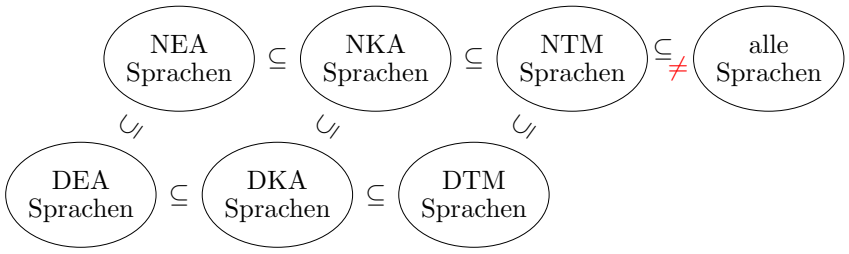
1.8 Definition (Zusammenhang zwischen Automaten und Sprachen)

Seien Σ ein Alphabet und $L \subset \Sigma^*$ eine Sprache. Wir sagen, L ist eine

- **DEA-Sprache**, wenn es einen Deterministischen Endlichen Automaten gibt, der L akzeptiert,
- **NEA-Sprache**, wenn es einen Nichtdeterministischen Endlichen Automaten gibt, der L akzeptiert,
- **DKA-Sprache**, wenn es einen Deterministischen Kellerautomaten gibt, der L akzeptiert,
- **NKA-Sprache**, wenn es einen Nichtdeterministischen Kellerautomaten gibt, der L akzeptiert,
- **DTM-Sprache**, wenn es eine Deterministische Turing Maschine gibt, die L akzeptiert und
- **NTM-Sprache**, wenn es eine Nicht Deterministische Turing Maschine gibt, die L akzeptiert.

Beachte

- Jede DEA-Sprache ist auch eine NEA-Sprache.
- Jede DEA-Sprache ist auch eine DKA-Sprache.
- Jede DKA-Sprache ist auch eine DTM-Sprache.
- ...



Frage

Welche dieser Sprachenklassen fallen zusammen, welche nicht?

1.9 Definitionen(„Baby-Mengenlehre“)

- Sei f eine Funktion $A \to B$. Die Funktion heißt
  - **injektiv**, wenn $\forall a, a' \in A, a \neq a' : f(a) \neq f(a')$,
  - **surjektiv**, wenn $\forall b \in B, \exists a \in A : f(a) = b$ und
  - **bijektiv**, wenn sie injektiv und surjektiv ist.

  Im Englischen sagt man auch **1-1** für injektiv, **onto** für surjektiv und **1-1 correspondence** für bijektiv.

- Zwei Mengen A, B heißen **gleichmächtig**, wenn es eine Bijektion zwischen A und B gibt.
  - **Beispiel**: Sei $\mathbb{G}$ die Menge der geraden, natürlichen Zahlen.
    Behauptung: $\mathbb{N}$ und $\mathbb{G}$ sind gleichmächtig, $\mathbb{G} \subsetneq \mathbb{N}$
    Beweis: $f(x) = 2 \cdot x$ ist eine Bijektion von $\mathbb{N} \to \mathbb{G}$.

- Eine Menge A heißt **endlich**, wenn A gleichmächtig mit $\{1, 2, ..., k\}$ für irgendein $k \in \mathbb{N}$ ist, sonst unendlich.

- Eine Menge A heißt **abzählbar unendlich**, wenn A gleichmächtig mit $\mathbb{N}$ ist.

- Eine Menge A heißt **abzählbar**, wenn A endlich oder abzählbar undendlich ist.

Beachte

A heißt abzählbar unendlich, wenn A als $A = \{a_0, a_1, a_2, ...\}$ „geschrieben“ werden kann. Es existiert eine Bijektion zwischen $\mathbb{N}$ und A, da $f(i) = a_i$.

1.9.1 Lemma 0

i. Eine Menge A ist genau dann abzählbar, wenn $\exists$ es eine Injektion $A \to \mathbb{N}$. Dazu äquivalent ist auch, dass $\exists$ eine Surjektion $\mathbb{N} \to A$.

Beweis

Übung

ii. Ist eine Menge B abzählbar und $A \subseteq B$, dann ist A abzählbar.

1.9.2 Lemma 1

$$
\begin{aligned}
&A_i \text{ ist endlich für } i = 1, \ldots, k \\
\Rightarrow &A_1 \cup A_2 \cup ... \cup A_k \text{ ist endlich und} \\
&A_1 \times A_2 \times ... \times A_k \text{ ist endlich.}
\end{aligned}
$$

Endliche Mengen sind also abgeschlossen unter endlicher Vereinigung und Produktbildung.

1.9.3 Lemma 2

A_i ist endlich für $i \in \mathbb{N} \Rightarrow \bigcup_{i\in\mathbb{N}} A_i$ ist abzählbar.
Ist also $\{A_i | i \in \mathbb{N}\}$ eine Familie von disjunkten endlichen Mengen, dann ist auch $\bigcup_{i\in\mathbb{N}} A_i$ abzählbar.

Beweis

Sei $k \in \mathbb{N} : s_k = \sum_{i<k} l_i$ mit $l_i = |A_i|$ und $A_i = \left\{a_{i_0}, a_{i_1}, a_{i_2}, \ldots, a_{i_{l_i-1}}\right\}$. Dann gilt:

$$f : \bigcup_{i\in\mathbb{N}} A_i \to \mathbb{N} : f(a_{i_j}) = s_i + j \quad \text{ist injektiv}$$

Aus Lemma 0 folgt: $\bigcup_{i\in\mathbb{N}} A_i$ ist abzählbar.

1.9.4 Korollar

Wenn Σ endlich ist, dann ist Σ^* abzählbar.

Beweis

$$\Sigma^* = \bigcup_{i\in\mathbb{N}} \Sigma^i, \text{ da } \Sigma^i \cap \Sigma^j = \emptyset \text{ für } i \neq j \text{ und } \Sigma^i \text{ endlich} \quad \forall i \in \mathbb{N}$$

Übung

Disjunktheitsannahme entfernen.

1.9.5 Lemma 3

Sind A und B abzählbar, dann ist auch $A \cup B$ abzählbar. Ferner ist dann $A \times B$ abzählbar.

Beweis

$$\begin{aligned} &A \text{ abzählbar } \Rightarrow \exists \text{ Surjektion } \alpha : \mathbb{N} \to A, \quad A = \{\alpha(0), \alpha(1), \alpha(2), \ldots\} \\ &B \text{ abzählbar } \Rightarrow \exists \text{ Surjektion } \beta : \mathbb{N} \to B, \quad B = \{\beta(0), \beta(1), \beta(2), \ldots\} \end{aligned}$$

i. Konstruiere Surjektion $f : \mathbb{N} \to A \cup B$:

$$\begin{aligned} f(2i) &= \alpha(i) \\ f(2i+1) &= \beta(i) \end{aligned}$$

f ist surjektiv, weil $x \in A \cup B$

$\Rightarrow x \in A \ldots x = \alpha(i)$ für irgendeine $\quad \Rightarrow x = f(2i)$

$\Rightarrow x \in B \ldots x = \beta(i)$ für irgendeine $\quad \Rightarrow x = f(2i+1)$

ii.

$$A = \{a_0, a_1, \ldots\} \qquad a_i = \alpha(i)$$
$$B = \{b_0, b_1, \ldots\} \qquad b_i = \beta(i)$$

$A \times B$	0	1	2	3	
0	(a_0, b_0)	(a_0, b_1)	(a_0, b_2)	(a_0, b_3)	...
1	(a_1, b_0)	(a_1, b_1)	(a_1, b_2)	(a_1, b_3)	...
2	(a_2, b_0)	(a_2, b_1)	(a_2, b_2)	(a_2, b_3)	...
3	(a_3, b_0)	(a_3, b_1)	(a_3, b_2)	(a_3, b_3)	...

Sei $k \in \mathbb{N}$. Dann gilt:

$$C_k = \{(a_i, b_j) \,|i + j = k\}$$

Zudem gilt dann:

$$|C_k| = k + 1$$
$$C_k \cap C_l = \emptyset \text{ für } k \neq l$$
$$A \times B = \bigcup_{k \in \mathbb{N}} C_k \overset{\text{L2}}{\Rightarrow} A \times B \text{ ist abzählbar.}$$

1.9.6 Korollar

$\mathbb{Q}$ ist abzählbar. $\approx \mathbb{N} \times \mathbb{N} \cup$ „ $- \mathbb{N} \times \mathbb{N}$“

1.9.7 Lemma 4

Sei $\{A_i | i \in \mathbb{N}\}$ die Familie von abzählbaren Mengen. $\Rightarrow \bigcup_{i \in \mathbb{N}} A_i$ ist abzählbar.

Beweis

Analog zu Beweis von Lemma 3 ii. .

1.9.8 Satz (Cantor, Russell)

Sei A eine Menge und $2^A = \{B | B \subset A\}$ die Potenzmenge von A. Dann gilt:

$$A \text{ und } 2^A \text{ sind nicht gleichmächtig,}$$

es gibt also keine Bijektion zwischen A und 2^A.

Beweis

Es genügt zu zeigen, dass es keine Surjektion $A \to 2^A$ gibt:

$$\forall f : A \to 2^A \quad f \text{nicht surjektiv}$$
$$\forall f : A \to 2^A \quad \exists \underbrace{\Delta}_{\subset A} \in 2^A : \forall a \in A : f(a) \neq \Delta$$
$$\forall f : A \to 2^A \quad \Delta = \Delta_f = \left\{ a \in A | a \notin \underbrace{f(a)}_{\subset A} \right\}$$

Behauptung

$\forall a \in A : f(a) \neq \Delta$ denn sie unterscheiden sich in a.

$$a \notin f(a) \Rightarrow a \in \Delta \qquad \Rightarrow f(a) \neq \Delta$$
$$a \in f(a) \Rightarrow a \notin \Delta \qquad \Rightarrow f(a) \neq \Delta$$

1.9.9 Korollar

i. $2^{\mathbb{N}}$ ist **nicht** abzählbar.

ii. 2^{Σ^*} ist **nicht** abzählbar $\left(2^{\Sigma^*} = \text{„Alle Sprachen“}\right)$.
Nebenbedingung: NTM − Sprachen sind abzählbar.
$\subsetneq 2^{\Sigma^*}$
Es gibt Sprachen, die von keiner NTM akzeptiert werden.

iii. Die Menge aller unbeschränkten Bitstrings

$$\{\alpha | \alpha : \mathbb{N} \to \{0,1\}\}$$

$\alpha(0)$	$\alpha(1)$	$\alpha(2)$	...
0	1	1	...

nicht abzählbar, denn jedes $A \subset \mathbb{N}$ identifizieren wir mit $\alpha_A : \mathbb{N} \to \{0,1\}$:

$$\alpha_A(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A \end{cases}$$

iv. $\mathbb{R}$ ist überabzählbar (selbst $[0,1] \cap \mathbb{R}$), denn jeder $x \in [0,1]$ ist als $0, xxxxx\ldots$ darstellbar und umgekehrt.
Binärdarstellung

1.10 Cantor'sches Diagonalisierungsverfahren

Hierbei handelt es sich um einen Spezialfall dieses Satzes für $A = \mathbb{N}$. Man geht wie folgt vor:

- Identifiziere $B \subset \mathbb{N}$ mit dem unbeschränkten Bitstring α_B
- Zeige, dass in jeder Aufzählung von unbeschränkten Bitstrings irgendein Bitstring fehlt.

	0	1	2	3	4	5	6
α_0	1^0	0	1	1	0	0	1
α_1	1	0^1	1	1	1	0	1
α_2	0	0	1^0	0	0	0	1
α_3	1	0	0	0^1	0	0	0
α_4	0	0	1	1	0^1	0	1

$\delta(j) = 1 - \alpha_j(j)$
$\delta \neq \alpha_i \quad \forall i \in \mathbb{N}$

Kapitel 2

Endliche Automaten und deren Sprachen

2.1 Definition

i. Ein **endlicher Automat** M spezifiziert sich durch

$\Sigma:$	Eingabealphabet (endlich)
$Q:$	Zustandsmenge (endlich)
$s \in Q:$	Startzustand
$F \subset Q:$	Endzustände
$\Delta \subset (Q \times \Sigma) \times Q:$	Übergangsrelation (Rechenschrittregeln)

M ist **deterministisch**, wenn

$$\forall (q,a) \in Q \times \Sigma : |\{q' \in Q | (q,a,q') \in \Delta\}| \leqslant 1$$

(es ist höchstens 1 Regel anwendbar)

Ansonsten heißt M **nicht-deterministisch**.

ii. Die Konfigurationen von M sind die Menge $K_M = Q \times \Sigma^*$.

- **Startkonfiguration:**

$$w \in \Sigma^* : start_w = (s, w)$$

- **Endkonfigurationen:**

$$\{(f, \varepsilon) | f \in F\}$$

- **Rechenschrittrelation für M:**
  Hierbei handelt es sich um eine Relation auf K_M

$$\forall q \in Q, a \in \Sigma, w \in \Sigma^* : (q, aw) \underset{M}{\vdash} (q'w) \Leftrightarrow ((q,a), q') \in \Delta$$

- **Rechenrelation für M:**
  $\underset{M}{\vdash}^*$ ist reflexive, transitive Hülle von $\underset{M}{\vdash}$

  $k \underset{M}{\vdash}^* k' \Leftrightarrow \exists m \in \mathbb{N}, \exists k_0, \ldots, k_m : k_0 = k, k' = k_m, k_{i-1} \underset{M}{\vdash} k_i$ für $0 < i \leqslant m$

iii. Die Maschine M akzeptiert die Eingabe $w \in \Sigma^*$ genau dann wenn

$$\exists \underset{\substack{\| \\ (f,\varepsilon), f \in F}}{k} \in Fin : start_w \underset{M}{\vdash}^* k.$$

Die von M akzeptierte Sprache $L(M) = \{w \in \Sigma^* | M \text{ akzeptiert } w\}$.

iv. $L \subset \Sigma^*$ heißt $\overset{\text{DEA}}{\text{NEA}}$-Sprache, wenn es einen $\overset{\text{DEA}}{\text{NEA}}$ M gibt mit $L(M) = L$.

2.1.1 Beispiele

Endlicher Automat M:

$$\begin{aligned}
\Sigma &= \{a, b\} \\
Q &= \{q_0, q_1, q_2\} \\
s &= q_0 \\
F &= \{q_1, q_2\} \\
\Delta &= \{(q_0, a, q_1), (q_1, a, q_2), (q_2, a, q_0), \\
&\qquad (q_0, b, q_0), (q_1, b, q_1), (q_2, b, q_2)\}
\end{aligned}$$

Daraus folgt: M ist deterministisch.

$$\begin{aligned}
K_M &= Q \times \Sigma^* \\
Fin &= \{(q, \varepsilon) | q \in F\} \\
&= \{(q_1, \varepsilon), (q_2, \varepsilon)\}
\end{aligned}$$

Verwenden wir nun einige Testeingaben:

- Mit der Eingabe $w = aab$ erhält man:

  $$start_w = (q_0, aab) \underset{M}{\vdash} (q_1, ab) \underset{M}{\vdash} (q_2, b) \underset{M}{\vdash} (q_2, \varepsilon) \in Fin$$

  Somit ist $aab \in L(M)$.

- Mit der Eingabe $w = baaba$ erhält man:

  $$\begin{aligned}
  start_w = (q_0, baaba) &\underset{M}{\vdash} (q_0, aaba) \\
  &\underset{M}{\vdash} (q_1, aba) \\
  &\underset{M}{\vdash} (q_2, ba) \\
  &\underset{M}{\vdash} (q_2, a) \\
  &\underset{M}{\vdash} (q_0, \varepsilon) \notin Fin
  \end{aligned}$$

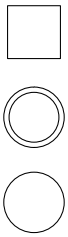
  Somit ist $baaba \notin L(M)$

2.2 Übergangsgraphen für endliche Automaten

Sei M, charakterisiert durch $(\Sigma, Q, s, F, \Delta)$ ein endlicher Automat. G_M ist ein gerichteter (Multi-)Graph mit Kantenbeschriftung aus Σ.

Knotenmenge :Q
Kantenmenge : $[q, q'\rangle$ mit Beschriftung $a \Leftrightarrow (q, a, q') \in \Delta$

Konventionen beim Zeichnen

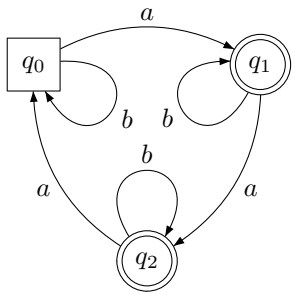


Startknoten (-zustand)

Endknoten (-zustand)

sonst

Vorheriges Beispiel:



$w = a_1 a_2 \dots a_m$ wird von M akzeptiert $\Leftrightarrow \exists$ Pfad in G_M vom Startzustand zu einem Endzustand mit Beschriftung w.
$\exists$ Knotenfolge $p_0, p_1, \dots, p_m$ mit $p_0 = s$, $p_m \in F$ für alle $0 < i \leqslant m$ $p_{i-1} \xrightarrow{a_i} p_i$

Fortsetzungssprachen

Seien $L \subset \Sigma^*$ ist eine Sprache, $w \in \Sigma^*$, $F_L(w) = \{w' \in \Sigma^* | ww' \in L\}$.

$$\mathcal{F}_L = \{F_L(w) | w \in \Sigma^*\}$$

Man nennt $\mathcal{F}_L$ die Menge der Fortsetzungssprachen von L. Es ist zu beachten, dass unter Umständen für $w \neq w' : F_L(w) = F_L(w')$!

2.2.1 Lemma

L DEA-Sprache $\Rightarrow \mathcal{F}_L$ endlich

Beweis

L DEA-Sprache $\Rightarrow \exists$ DEA-Sprache $M = (\Sigma, Q, s, F, \Delta)$ mit $L(M) = L$
Sei $q \in Q$. Man betrachte

$$L_q = \left\{ w \in \Sigma^* | (q, w) \underset{M}{\vdash}^* (f, \varepsilon) \text{ für irgendein } f \in F \right\}.$$

Sei $w \in \Sigma^*$: $q(w)$ ein eindeutiger Zustand (falls existent) mit $(s, w) \underset{M}{\vdash}^* (q(w), \varepsilon)$.
Dann gilt: $F_L(w) = L_{q(w)}$. Falls $q(w)$ nicht existiert, dann $F_L(w) = \emptyset$.
$\Rightarrow \mathcal{F}_L = \{L_q | q \in Q\}$ endlich.

Anwendungen

Um zu zeigen, dass L **keine** DEA-Sprache ist, reicht es zu zeigen, dass $\mathcal{F}_L$ nicht endlich ist.

Beispiel 1

$L = \{a^n b^n | n \in \mathbb{N}\}$ ist **nicht** DEA-Sprache.
Idee: Produziere $w_i \in \Sigma^*, i \in \mathbb{N}$, mit $F_L(w_i) \neq F_L(w_j)$ für $i \neq j$. Betrachte $w_i = a^i b; i \geqslant 1$.

$$F_L(w_i) = \left\{ b^{i-1} \right\}$$

Beispiel 2

$L = \{a^n | n \text{ ist Quadratzahl}\} = \left\{ a^{k^2} | k \in \mathbb{N} \right\}$
L ist keine DEA-Sprache, denn für $i < j$ unterscheiden sich $F_L(a^{i^2})$ und $F_L(a^{j^2})$, weil $a^{2i+1} \in F_L\left(a^{i^2}\right)$ und $a^{2i+1} \notin F_L\left(a^{j^2}\right)$.

$$F_L(a^4) = \left\{ \varepsilon, a^5, a^{12}, a^{21}, \ldots \right\}$$
$$F_L(a^9) = \left\{ \varepsilon, a^7, \ldots \right\}$$

Beispiel 3

$L = \{a^n | n \text{ ist Primzahl}\}$ ist keine DEA-Sprache.

Beweis

i. $\exists$ unendlich viele Primzahlen

ii. x, y teilerfremd $\Rightarrow \{x + iy | i \in \mathbb{N}\}$ enthält eine Primzahl (Satz von Dirichlet)

$p, q : p \neq q$ Primzahlen. Dann ist $F_L(a^p) \neq F_L(a^q)$. Daraus folgt: p, q teilerfremd $\Rightarrow \{p + iq | i \in \mathbb{N}\}$ enthält eine Primzahl, z.B. $p + Iq$
Beachte: $q + Iq$ ist keine Primzahl (durch q teilbar).
$a^{Iq} \in F_L(a^p), a^{Iq} \notin F_L(a^q)$

2.3 Satz von Myhill-Nerode

Sei L eine DEA-Sprache. Genau dann ist $\mathcal{F}_L$ endlich.

Beweis

„⇒“ Durch vorheriges Lemma gezeigt.

„⇐“ $L, \mathcal{F}_L$ endlich. Baue $M = (\Sigma, Q, s, F, \Delta)$ mit $L(M) = L$.

$$
\begin{aligned}
Q &= \mathcal{F}_L \\
s &= F_L(\varepsilon) = L \\
F &= \{G \in \mathcal{F}_L | \varepsilon \in G\} \\
\Delta &= \{(F_L(w), a, F_L(ww')) \,| w \in \Sigma^*, w' \in \Sigma\}
\end{aligned}
$$

Beachte:

i. Δ endlich, da Σ und $\mathcal{F}_L$ endlich.

ii. Sei $w, w' \in \Sigma^*$ und gilt: $F_L(w) = F_L(w') \Rightarrow F_L(wa) = F_L(w'a)$

$L(M) = L$, da:
$w, w' \in \Sigma^*, a \in \Sigma$. $start_{w'} = (F_L(\varepsilon), w')$, ferner gilt

$$(F_L(w), aw') \underset{M}{\vdash} (F_L(wa), w')$$

$$(F_L(\varepsilon), a_1 a_2 \dots a_n) \underset{M}{\vdash}^* F_L(a_1 \dots a_k), a_{k+1} \dots a_n) \quad \forall k$$

insbesondere mit $k = n, w_1 = a_1 \dots a_n$

$$(F_L(\varepsilon), w_1) \underset{M}{\vdash}^* (F_L(w_1), \varepsilon)$$

$F_L(w')$ ist hierbei Endzustand, wenn es ε enthält.

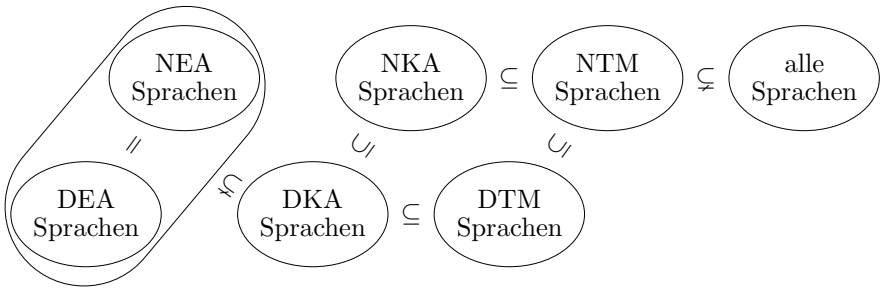
2.3.1 Anmerkungen

i. Diese Konstruktion liefert einen kanonischen DEA für L, KA_L

ii. DEA $M = (\Sigma, Q, s, F, \Delta)$ mit $L(M) = L \Rightarrow |\mathcal{F}_L| \leqslant |Q|; |\mathcal{F}_L| = |Q| \Rightarrow M \sim KA_L$

2.3.2 Konsequenzen

i. **Satz**
L ist NEA-Sprache $\Rightarrow$ L ist DEA-Sprache (d.h. nicht-Determinismus führt bei endlichen Automaten zu keinen neuen Sprachen)



Beweis

Werde L durch NEA $M = (\Sigma, Q, s, F, \Delta)$ akzeptiert. Es genügt zu zeigen, dass die Menge der Fortsetzungssprachen $\mathcal{F}_L$ endlich ist. Für $w \in \Sigma^*$ sei $Q(w) = \left\{p \in Q | (s, w) \underset{M}{\vdash}^* (p, \varepsilon)\right\}$. Für $q \in Q$ sei $L_q = \left\{w' \in \Sigma^* | (q, w') \underset{M}{\vdash}^* (f, \varepsilon) \text{ für irgendein } f \in F\right\}$.

$$F_L(w) = \bigcup_{q \in Q(w)} L_q$$

das heißt $F_L(w)$ wird durch eine Teilmenge von Q (nämlich $Q(w)$) eindeutig bestimmt. Da Q nur endlich viele Teilmengen hat, gilt $\mathcal{F}_L$ ist endlich.

ii. **Satz**
Wenn L von einem endlichen Automaten akzeptiert wird, „der den Lesekopf auch auf einer Zelle verweilen lassen kann“, dann ist L eine DEA-Sprache.

$$\Delta \subset (Q \times (\Sigma \cup \{\varepsilon\}) \times Q)$$

d.h. im Übergangsgraphen gibt es Kanten, die mit ε beschriftet sind.

iii. **Satz** Wird L von einem 2-Weg-Automat akzeptiert, dann ist L eine DEA-Sprache.
(Bei einem 2-Weg-Automat handelt es sich um einen DEA, bei dem der Lesekopf verweilen und auch nach links und rechts bewegen kann.)
Übergangsregeln:

$$(q, a, q', \mu) \in Q \times \Sigma \times Q \times \{L, B, R\}$$

Hierbei steht L für „links“, B für „bleiben“ und R für „rechts“.

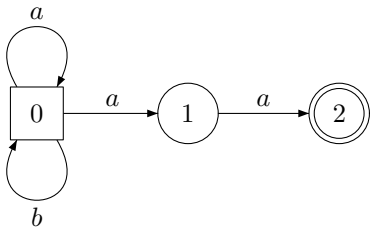
Unmittelbar aus Konsequenz 1 folgt:

2.4 Konstruktion eines DEAs aus einem NEA

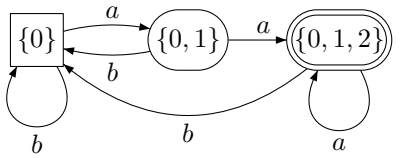
Ziel sei die direkte Konstruktion eines DEA $M' = (\Sigma, Q', s', F', \Delta')$ aus einem NEA $M = (\Sigma, Q, s, F, \Delta)$ mit $L(M') = L(M)$.
Man gehe folgendermaßen vor:

$$Q' = 2^Q$$
$$s' = \{s\} \in Q'$$
$$F' = \left\{P \underset{\subseteq Q}{\in} Q' \,|P \cap F \neq \emptyset\right\}$$
$$\Delta' = \left\{(P, a, R) \in \underbrace{Q'}_{=2^Q} \times \Sigma \times \underbrace{Q'}_{=2^Q} | P \subset Q, a \in \Sigma, R = \{q \in Q | (p, a, q) \in \Delta \text{ für irgendein } p \in P\}\right\}$$

2.4.1 Beispiel



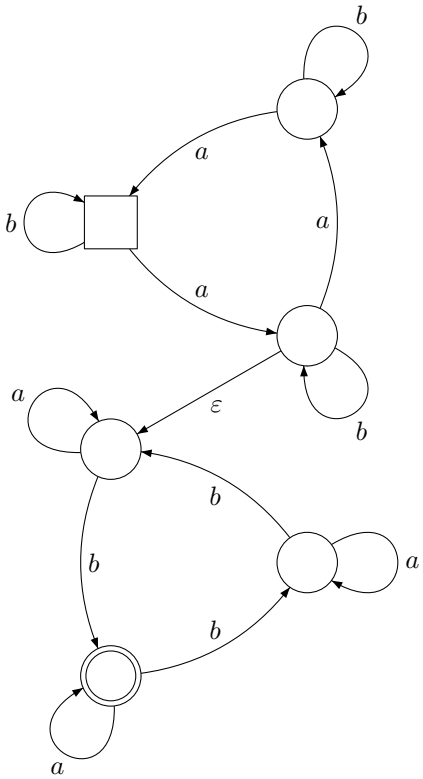
Konstruiere einen DEA. Vorgehensweise ist hierbei die Frage, „in welcher Menge von Zuständen könnte man sein?“.



Unmittelbar aus Konsequenz 2 folgt:

2.5 Verweilen des Lesekopfes

2.5.1 Beispiel



Akzeptiert alle Strings $w \in \Sigma^*$, die sich teilen lassen in $w = uv,\ u, v \in \Sigma^*$ mit $\#_a(u) \mod 3 = 1$ und $\#_b(v) \mod 3 = 1$.

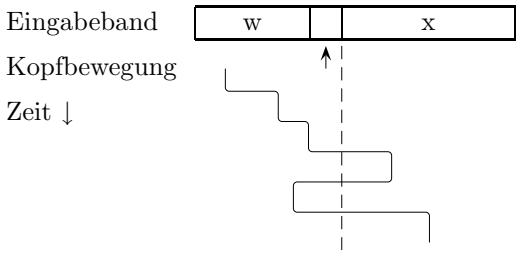
Beweis

Analog zu dem des vorhergehenden Satzes.
Unmittelbar aus Konsequenz 3 folgt:

2.6 2-Weg-Automat

Beweis

Wir wollen zeigen, dass $\mathcal{F}_L$ endlich ist.



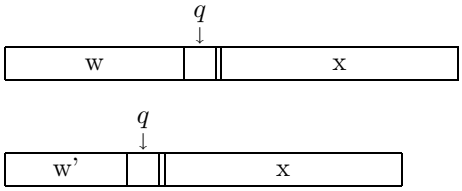
$w \in \Sigma^*, f_w : Q \to Q \times \{V, R\} \cup \{div\}$. Wir nehmen an, dass der Lesekopf auf dem rechtesten Zeichen von w steht und M im Zustand q ist.

1. Fall M rückt den Lesekopf im nächsten Schritt nach rechts und geht über in den Zustand p.
$$f_w(q) = (p, R)$$

2. Fall Der Fall 1 tritt nicht ein und das nächste Mal, wenn der Lesekopf wieder über der rechtesten Zelle von w steht, ist M im Zustand p.
$$f_w(q) = (p, V)$$

3. Fall Sonst $f_w(q) = div$

$e = \Sigma^* \to Q \cup \{div\}$
$e(w)$ ist gleich dem Zustand q, der erreicht wird, wenn M beginnend mit Lesekopf auf dem linken Ende von W und im Startzustand das rechte Ende von w zum ersten Mal erreicht. (falls das nie der Fall ist, liegt Divergenz vor).
$w, w' \in \Sigma^*, \quad e(w) = e(w')$ und $f_w = f_{w'}$. Daraus folgt:



$$\begin{aligned}\forall x \in \Sigma^* : & wx \text{ wird akzeptiert} \\ \Leftrightarrow & w'x \text{ wird akzeptiert} \\ \Rightarrow & F_L(w) = F_L(w')\end{aligned}$$

$F_L(w)$ ist durch $e(w) \in Q \cup \{div\}$ und $f_w \in (Q \times \{V, R\} \cup \{div\})^Q$ bestimmt. Daher gibt es höchstens

$$|Q| + 1 \cdot (2 \cdot |Q| + 1)^{|Q|}$$

viele Fortsetzungssprachen.

2.6.1 Konsequenz 4

Zustandsminimierung für DEAs

2.6.2 Satz

Für jeden DEA M kann man den **kanonischen** äquivalenten DEA KM berechnen.

i. $L(KM) = L(M)$

ii. Zustandsmenge von KM ist so klein wie möglich

iii. Berechnung läuft in Zeit polynomiell in der Beschreibungsgröße von M.

Beweis:

Sei $M = (\Sigma, Q, s, F, \Delta)$ ein DEA und $L = L(M)$ die Sprache, die von M akzeptiert wird. Wir wissen: $\mathcal{F}_L = \{L_q | q \in Q\}$ Wir möchten zeigen, dass $\forall \{p, q\}$ gilt

$\underbrace{L_p \neq L_q}_{\{p,q\} \text{ unterscheidbar}}$ Allgemein $L_p = L_q$ zusammenfassen nicht unterscheidbar

Plan

i. Bestimme, welche $\{p, q\}$ unterscheidbar sind und welche nicht.

ii. Fasse die nicht-unterscheidbaren Zustände zu Äquivalenzklassen zusammen und baue einen DEA, dessen Zustände diese Äquivalenzklassen sind.

Algorithmus für i.

0. Entferne alle Zustände, die vom Startzustand nicht erreichbar sind. „Vervollständige“ M'. $\forall$ Knoten q, $\forall a \in \Sigma$ muss es genau eine Kante geben, die in q beginnt und mit a beschriftet ist, füge also einen Knoten hinzu, auf den alle diese Kanten zeigen. Sei Maschine $M = (\Sigma, Q, s, F, \Delta)$.

1.
$$U = \{\{p, q\} | p \in F,\ q \in Q \setminus F\}$$
$$N = \binom{F}{2} \cup \binom{Q \setminus F}{2}, \quad \binom{X}{k} = \{A \subset X | |A| = k\}$$

2. while $\exists \{p, q\} \in N$: $\exists a \in \Sigma : \{p', q'\} \in U$ mit
$$\underset{(p,a,p') \in \Delta}{p \xrightarrow{a} p'}\ ,\ \underset{(q,a,q') \in \Delta}{q \xrightarrow{a} q'}$$
do

- entferne $\{p, q\}$ aus N
- füge $\{p, q\}$ in U ein

3. Alle Paare in U sind unterscheidbar, alle Paare in N sind nicht unterscheidbar.

2.6.3 Lemma

Wenn der Algorithmus terminiert, gilt

i. $\forall \{p, q\} \in U : L_p \neq L_q$

ii. $\forall \{p, q\} \in N : L_p = L_q$

iii. $U \cup N = \binom{Q}{2}$

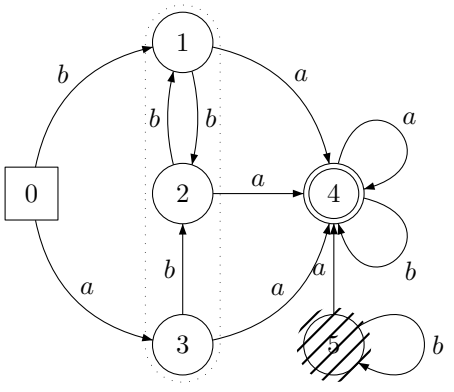
Beweis

iii. gilt als Schleifeninvariante.

i. $L_p \neq L_q$ gilt für alle $\{p, q\} \in U$ nach Schritt 1., da $\varepsilon \in L_p, \varepsilon \notin L_q$. In Schritt 2. werden nur Paare $\{p, q\}$ mit $L_p \neq L_q$ zu U hinzugefügt, da $L_p \neq L_{p'}$ (da $\{p', q'\} \in U$ „nach Induktionsvoraussetzung“), d.h. $\exists w \in \Sigma* : w \in L_{q'}, w \notin L_{p'} \Rightarrow \left.\begin{array}{l} aw \in L_q \\ aw \notin L_p \end{array}\right\} \Rightarrow L_p \neq L_q$

ii. Annahme: Bei Termination $\exists \{p, q\} \in N : \quad L_p \neq L_q$. So etwas wird durch „Zeugen“ $w \in \Sigma^*$ bezeugt: $\begin{array}{l} w \in L_p \\ w \notin L_q \end{array}$.
Sei $\{p, q\} \in N$ mit $L_p \neq L_q$ mit kürzesten Zeugen $w = w_1, \ldots, w_k$ $\{p_1, q_1\} \in U \quad \Rightarrow$ Schleife hat noch **nicht** terminiert $\lightning$. $\{p_1, q_1\} \in N$, Widerspruch zur Wahl eines Paares in N mit **kürzestem** Zeugen.

Beispiel

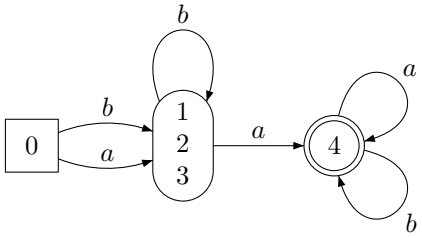


3	U			
2	U			
1	U			
0	U	U	U	U
	4	3	2	1

$U = \{\{p,q\}\,|p \in F, \quad q \in Q \setminus F\}$, jedes Paar $\{p,q\}$ ist also durch die Position p,q gekennzeichnet.

$$\begin{aligned}
\{0,3\} &\xrightarrow{a} \{3,4\} \in U \\
\{0,2\} &\xrightarrow{b} \{1,1\} \\
\{0,2\} &\xrightarrow{a} \{3,4\} \in U \\
\{0,1\} &\xrightarrow{a} \{3,4\} \in U \\
\{1,3\} &\xrightarrow{a} \{4,4\} \\
&\xrightarrow{b} \{2,2\} \\
\{1,2\} &\xrightarrow{a} \{4,4\} \\
&\xrightarrow{b} \{2,1\} \\
\{2,3\} &\xrightarrow{a} \{4,4\} \\
&\xrightarrow{b} \{1,2\}
\end{aligned}$$

$N = \{\{1,3\},\{1,2\},\{2,3\}\}, \quad L_1 = L_3, \quad L_1 = L_2, \quad L_2 = L_3, \quad L_1 = L_2 = L_3$



2.6.4 Korollar

Für 2 EA − Sprachen L, L' kann man testen, ob $L = L'$.
reguläre Sprachen

2.7 Eigenschaften von DEA-Sprachen (regulären Sprachen)

2.7.1 Satz

Seien L, L' reguläre Sprachen, $w, u \in \Sigma$ dann sind auch folgende Sprachen regulär:

i. $\overline{L}$

ii. $L \cup L', \quad L \cap L', \quad L \cdot L' = \{wu | w \in L,\ u \in L'\}$

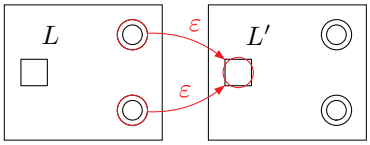
iii. $L^R = \{w | w^R \in L\}$

iv. $L^* = \bigcup_{k \in \mathbb{N}} L^k$
$L^k = \underbrace{L \cdot L \cdot \ldots \cdot L}_{k}$

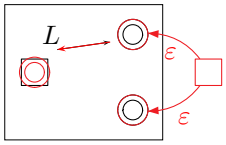
Beweis

Seien L, L' durch $\begin{matrix}\text{DEAs}\\ \text{NEAs}\end{matrix}$ gegeben mit den Übergangsgraphen:

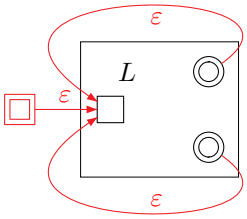
ii.



iii.



iv.



2.7.2 Satz

L, L' seien DEA-Sprachen (gegeben durch zwei DEAs). Man kann automatisch entscheiden, ob

i. $L = L'$

ii. $L = \emptyset$

iii. $L = \Sigma^*$

iv. $L \subset L'$

Beweis

ii. Sei $L = L(M)$, wobei M ein DEA ist. Betrachte den Übergangsgraph G_M.
$L = \emptyset \Leftrightarrow \exists$ kein Pfad von s zu einem Knoten in F.

iii. Teste, ob $L = \Sigma^*$.

i. Minimiere M und M' und teste, ob $G_M \approx G_{M'}$.

iv. $L \subset L' \Leftrightarrow L \cap \overline{L'} = \emptyset$

$$\left.\begin{matrix}L' \text{ regulär} \Rightarrow \overline{L'} \text{ regulär}\\ L\text{regulär}\end{matrix}\right\} \Rightarrow L \cap \overline{L'} \text{ regulär}$$

2.8 Pumping Lemma (für DEA-Sprachen)

Sei L eine DEA Sprache. Dann $\exists N \in \mathbb{N}\ \forall x \in L$ mit $|x| \geqslant N$

$$\begin{gathered}\exists \text{ Unterteilung } x = uvw \\ \text{mit } |uv| \leqslant N \\ \text{und } |v| \geqslant 1\end{gathered}$$

Somit gilt $\forall i \in \mathbb{N} : uv^i w \in L$.

2.8.1 Anmerkung

Das Pumping Lemma ist wegen seiner Umkehrung interessant:

$$\begin{aligned}&\neg \left(\exists N \in \mathbb{N} : \ \underset{|x|\geqslant N}{\forall x \in L} \exists \underset{|uv|\leqslant N, |v|\geqslant 1}{\text{ Unterteilung } x = uvw} \forall i \in \mathbb{N} : uv^i w \in L \right) \\ &\Rightarrow \neg \left(L \text{ ist DEA Sprache} \right)\end{aligned}$$

Das heißt man kann zeigen, dass L keine DEA-Sprache ist.

$$\forall N \in \mathbb{N} \underset{|x|\geqslant N}{\exists x \in L} : \ \forall \underset{|uv|\leqslant N, |v|\geq 1}{\text{ Unterteilungen } x = uvw} : \ \exists i \in \mathbb{N} : uv^i w \notin L$$

$\Rightarrow L$ keine DEA-Sprache

2.8.2 „Spiel um zu zeigen, dasss L nicht regulär“

i. Der Gegner gibt ein $N \in \mathbb{N}$ vor.

ii. Ich wähle ein $x \in L$ mit $|x| \geqslant N$.

iii. Der Gegner unterteilt $x = uvw$ mit $|uv| \leqslant N, |v| \geqslant 1$.

iv. Ich wähle ein $i \in \mathbb{N}$ sodass $uv^i w \notin L$.

2.8.3 Beispiel

Zeige, $L = \{a^n b^n | n \in \mathbb{N}\} \subset \{a, b\}^*$ ist nicht regulär.

i. Der Gegner gibt ein $N \in \mathbb{N}$ vor.

ii. Ich wähle ein $x = a^N b^N$.

iii. Der Gegner gibt die Unterteilung $a^N b^N = x$ vor.
$\underbrace{aa\ldots}_{u=a^\alpha}\underbrace{\ldots}_{v=a^\beta}\underbrace{ab\ldots b}_{w}$
Dabei gilt $\beta \geqslant 1$ und $\alpha + \beta \leqslant N$

iv. Ich wähle $i = 0$:

$$\begin{aligned}uv^0 w &= a^\alpha \cdot a^{N-\alpha-\beta} \cdot b^N \\ &= a^{N-\beta} b^N \notin L\end{aligned}$$

Beweis (Pumping Lemma)

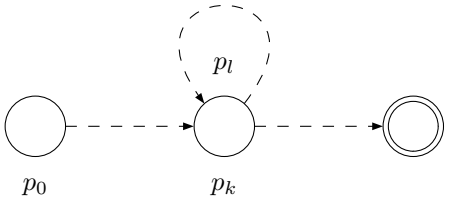
Sei L eine DEA Sprache. Somit $\exists$ DEA $M = (Q, \Sigma, s, F, \Delta)$ mit $L(M) = L$. Betrachte den Übergangsgraphen G_M.

- Sei nun $N = |Q|$.
- Betrachte $x \in L, |x| \geqslant N$
- Verfolge Pfad für x in G_M. Der Pfad hat die Länge $a \geqslant N$, also $a \geqslant N+1$ Knoten. Daher muss es Knotenwiederholungen geben:

  $$s = p_0 \overset{x_1}{\to} p_1 \overset{x_2}{\to} p_2 \overset{x_3}{\to} p_3 \to \ldots \to p_k \to \ldots \to p_l \to \ldots \to p_{|x|} \in F$$

  p_k, p_l sei dabei die erste Knotenwiederholung. Es gilt $l \leqslant N, k < l$.
  Setze

  $$u = x_1 \ldots x_k,$$
  $$v = x_{k+1} \ldots x_l,$$
  $$w = x_{l+1} \ldots x_{|x|},$$

  die Schleife kann also beliebig oft durchlaufen werden.

2.9 Reguläre Ausdrücke (Kleene)

Formalismus um Sprachen zu beschreiben (sei Σ das Alphabet):

	reguläre Ausdrücke	**beschriebene Sprachen**		
	$\emptyset$	$[\![\emptyset]\!]$	$=$	$\{\}$
$a \in \Sigma$	a	$[\![a]\!]$	$=$	$\{a\}$
	ε	$[\![\varepsilon]\!]$	$=$	$\{\varepsilon\}$
r_1, r_2 regulär	$(r_1 + r_2)$	$[\![(r_1 + r_2)]\!]$	$=$	$[\![r_1]\!] \cup [\![r_2]\!]$
	$r_1 r_2$	$[\![(r_1 r_2)]\!]$	$=$	$[\![r_1]\!] \cdot [\![r_2]\!]$
	$(r_1)^*$	$[\![(r_1)^*]\!]$	$=$	$[\![r_1]\!]^*$

Beispiel

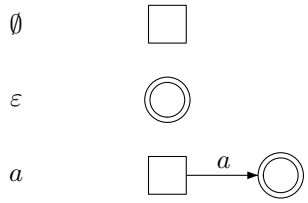
Sei $r = (aaa + bbb)\,((a + b))^*\,aaa$. Dann gilt:
$[\![r]\!]$ = Menge aller Strings über $\{a, b\}$ der Länge $\geqslant 6$, die in 3 a's enden und entweder mit 3 a's oder mit 3 b's beginnen.

2.9.1 Satz

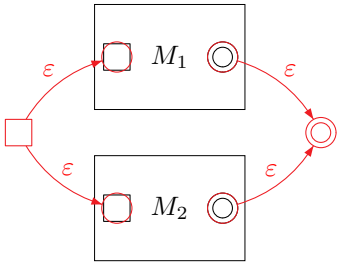
Sei $L = [\![r]\!]$ für irgendeinen regulären Ausdruck $r \Leftrightarrow L = L(M)$ für irgendeinen endlichen Automaten M.

Beweis

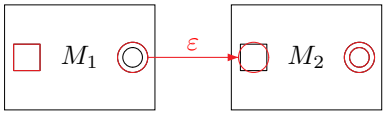
„⇒“
- Sei r ein regulärer Ausdruck.
- Konstruiere einen Endlichen Automaten M mit $L(M) = [\![r]\!]$.
- Strukturelle Induktion (sei M über $G(M)$ aufgeschrieben):
  Induktionsanfang:



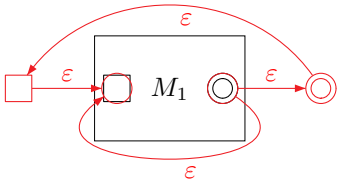
  Induktionsschritt:
  Seien $r1, r_2$ reguläre Ausdrücke, weiterhin existiere o.B.d.A. nur ein Endzustand. Dann sieht man:
  * $(r_1 + r_2)$:



  * r_1, r_2:



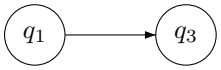
  * $(r_1)^*$:

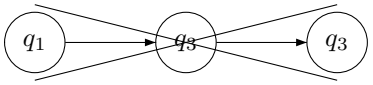


„⇐“ Gegeben sei ein DEA $M = (Q, \Sigma, s, F, \Delta)$ mit $Q = \{q_1, \ldots, q_n\}, s = q_1, F = \{q_i, \ldots, q_f\}$ mit $i \in \{1, 2\}, i \underset{s \in F}{=} 1$ oder $i \underset{s \notin F}{=} 2$. Konstruiere einen regulären Ausdruck r mit $[\![r]\!] = L(M)$, hierzu betrachte den Übergangsgraphen G_M:

Für $0 \leqslant k \leqslant n$ und $1 \leqslant i, j \leqslant n$ betrachte r_{ij}^k
$[\![r_{ij}^k]\!]$ = Menge aller Pfadbeschriftungen von den Pfaden, die in q_i beginnen, in q_j enden und **dazwischen** nur Knoten mit Index $\leqslant k$ haben.

$[\![r_{13}^2]\!] \ni$ 

$[\![r_{13}^2]\!] \ni$ 

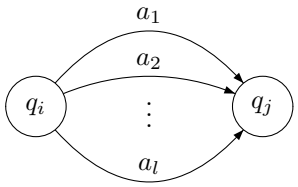


$$L(M) = \begin{cases} [\![r_{12}^n + \ldots + r_{1f}^n]\!] & \text{wenn } s \notin F \\ [\![r_{11}^n + \ldots + r_{1f}^n]\!] & \text{wenn } s \in F \end{cases}$$

r_{ij}^k sei induktiv definiert über k

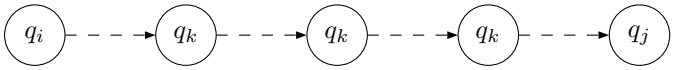
$$r_{ii}^0 = \varepsilon$$

$$i \neq j \quad r_{ij}^0 = \begin{cases} \emptyset & \text{wenn } \nexists \text{ Kante von } q_j \text{ nach } q_j \\ a_1 + a_2 + \ldots + a_l & \text{wenn } \exists \text{ genau } l \text{ Kanten von } q_i \text{ nach } q_j \\ & \text{mit Beschriftungen } a_1, a_2, \ldots a_l \end{cases}$$



Also gilt für $k > 0$:

$$r_{ij}^k = r_{ij}^{k-1} + r_{ik}^{k-1} \cdot \left(r_{kk}^{k-1}\right)^* \cdot r_{kj}^{k-1}$$

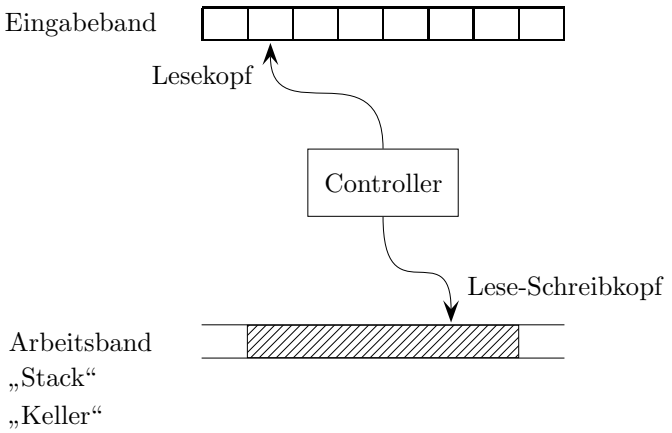


Kapitel 3

Kellerautomaten und ihre Sprachen

Kellerautomaten werden bedingt durch zumeist englischsprachige Literatur auch als „push down automata“ oder kurz PDA bezeichnet. Sie sind wie folgt Spezifiziert:

3.1 Spezifikation



Der Lese-/Schreibkopf bleibt immer am (rechten) Ende des beschriebenen Teils auf dem Arbeitsband, auch „Stack“ (Stapel) oder „Keller“ genannt.
Die Maschine M sei wie folgt spezifiziert:

Σ	Eingabealphabet
Γ	Kelleralphabet
$\in \in \Gamma$	Kellerbodensymbol
Q	endliche Zustandsmenge
$s \in Q$	Startzustand
$F \subset Q$	Endzustände
$\Delta \subset (Q \times \Gamma \times (\Sigma \cup \{\varepsilon\})) \times \Gamma^* \times Q$	Regeln, endlich!

Hierbei hat man die Definition von Δ mit Q als dem alten Zustand, Γ als dem obersten Kellersymbol, $\Sigma \cup \{\varepsilon\}$ als das nächste Eingabezeichen, Γ^* als der neue Kellergipfel und Q als der neue Zustand zu verstehen.

3.1.1 Konfigurationen

$$K_M = \underbrace{Q}_{\text{Derzeitiger Zustand}} \times \underbrace{\Gamma^*}_{\text{Kellerinhalt}} \times \underbrace{\Sigma^*}_{\text{Eingaberest}}$$

$$\text{Sei } w \in \Sigma^*.\ \text{start}_w = (s, \in, w)$$

$$\text{Fin} = \{(f, U, \varepsilon) \,|\, f \in F, U \in \Gamma^*\}$$

(Akzeptanz durch Endzustand).

3.1.2 Rechenschrittrelation

$$(q, AU, au) \underset{M}{\vdash} (q', VU, u) \Leftrightarrow (q, A, a, V, q') \in \Delta$$

mit $q \in Q$, $V, A \in \Gamma, U \in \Gamma^*, a \in \Sigma, u \in \Sigma^*$.

$$(q, AU, u) \underset{M}{\vdash} (q', VU, u) \Leftrightarrow (q, A, \varepsilon, V, q') \in \Delta$$

Sei weiterhin $\underset{M}{\vdash}^*$ die reflexive, transitive Hülle von $\underset{M}{\vdash}$.

$$L(M) = \left\{ w \in \Sigma^* | start_w \underset{M}{\vdash}^* \varphi \text{ für irgendein } \varphi \in \text{Fin} \right\}.$$

Eine alternative Möglichkeit ist:

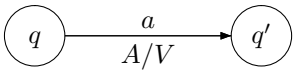
$$\text{Fin} = \{(q, \varepsilon, \varepsilon) \,|\, q \in Q\}$$

$$L_\varepsilon(M) = \left\{ w \in \Sigma^* | start_w \underset{M}{\vdash}^* \varphi \text{ für irgendein } \varphi \in \text{Fin} \right\}$$

(Akzeptanz durch leeren Keller).

Notation („Übergangsgraph“ für M)

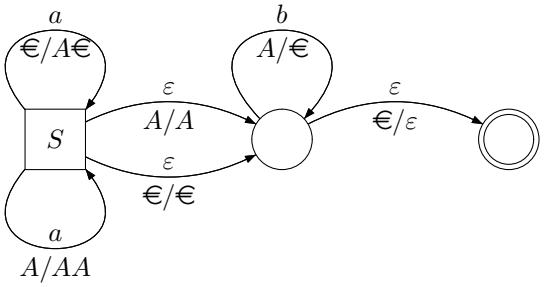
Man betrachte die Knotenmenge Q. Jede Regel (q, A, a, V, q') ergibt eine beschriftete Kante:



$a \in \Sigma \cup \{\varepsilon\}$

Beispiel

Sei $L = \{a^n b^n | n \in \mathbb{N}\}$. Man betrachte einen NKA M für L, wobei $\Sigma = \{a, b\}$ und $\Gamma = \{\in, A\}$.

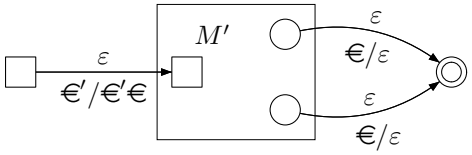


3.1.3 Satz

$L = L(M)$ für irgendeinen NKA $M \Leftrightarrow L = L_\varepsilon(M')$ für irgendeinen NKA M', also „Akzeptanz durch leeren Keller“ und „Akzeptanz durch Endzustand“ sind identisch.

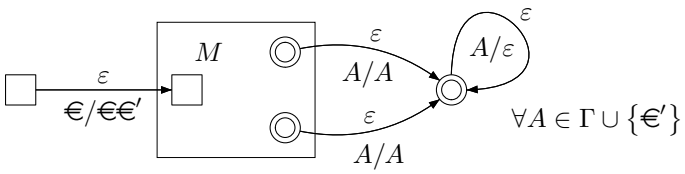
Beweis

„$\Leftarrow$“ Sei $M' = \left(\Sigma, \Gamma', €', Q', s', \Delta'\right)$ gegeben. Diese Maschine akzeptiert durch einen leeren Keller. Wir wollen die Maschine M, die genau die gleichen Strings aber durch einen Endzustand akzeptiert.
Dazu bauen wir eine Maschine M aus der Maschine M' und fügen einen Kellerunterboden hinzu, also $€ \notin \Gamma'$.



Wir führen einen neuen Startzustand s mit der zusätzlichen Regel $\left(s, €', \varepsilon, €'€, s'\right)$ sowie einen neuen Endzustand f und für alle $q \in Q'$ eine Regel der Form $(q, €, \varepsilon, \varepsilon, f)$ ein.

„$\Rightarrow$“ Sei $M = (\Sigma, \Gamma, €, Q, s, F, \Delta)$ gegeben. Diese Maschine akzeptiert durch einen Endzustand. Wir wollen eine Maschine M' bauen, die wie M agiert, aber am Ende ihren Keller leert. Die Idee hierbei ist, dass sobald ein Endzustand erreicht ist, die Leerung des Kellers erlaubt wird. Problematisch ist allerdings der Fall, wenn wir uns nicht in einem Endzustand befinden, der Keller aber leer ist. Die Lösung dieses Problems ist erneut ein Kellerunterboden.



3.1.4 Satz

Sei M ein NKA. Dann gibt es einen NKA $M' = \left(\Sigma, \Gamma', \in', Q', s', F', \Delta'\right)$ mit $L(M') = L(M)$, sodass für jede Regel $(q, A, a, V, q') \in \Delta'$ gilt $|V| \leqslant 2$. Der Keller kann somit immer nur um höchstens ein Symbol wachsen, gleich hoch bleiben oder um ein Symbol kleiner werden.

3.1.5 Satz (Pumping-Lemma für NKA-Sprachen (Kontextfreie Sprachen)

Sei L eine NKA-Sprache. Dann $\exists N \in \mathbb{N} : \forall z \in L, |z| \geqslant N : \exists$ eine Unterteilung $z = uvwxy$ mit $|vwx| \leqslant N$ und $|vx| > 0 : \forall i \in \mathbb{N} : uv^iwx^iy \in L$.
Der Beweis hierzu folgt später. Analog zum regulären Fall kann man dieses Lemma verwenden, um zu zeigen, dass die Sprache L **nicht** eine NKA-Sprache ist. Also muss gelten, dass $\forall N \in \mathbb{N} : \exists z \in L, |z| \geqslant N : \forall$ Unterteilungen $z = uvwxy$ mit $|vwx| \leqslant N$ und $|vx| > 0 \; \exists i \in \mathbb{N} : uv^iwx^iy \notin L$.
Somit ist L keine NKA-Sprache.

Verfahren zur Klassifikation einer Sprache nach der NEA-Eigenschaft

i. Der Gegner gibt $N \in \mathbb{N}$ vor.

ii. Ich wähle $z \in L$, $|z| \geqslant N$.

iii. Der Gegner gibt die Unterteilung $z = uvwxy$ vor mit $|vwx| \leqslant N$ und $|vx| > 0$.

iv. Ich wähle $i \in \mathbb{N}$, sodass $uv^iwx^iy \notin L$.

Beispiel

Betrachte die Sprache $L_{abc} = \{a^nb^nc^n | n \in \mathbb{N}\}$. Diese Sprache ist keine NKA-Sprache.

Beweis

Der Beweis erfolgt über das Pumping Lemma.

i. Gegner gibt N vor.

ii. Ich wähle $z = a^Nb^Nc^N \in L_{abc}$

iii. Gegner unterteilt $z = uvwxy$, $|vwx| \leqslant N, |vx| > 0$.

$$z = \underbrace{aaa\ldots a}_{u}\underbrace{aabbb\ldots}_{vwx}\underbrace{bbbccc\ldots ccc}_{y}$$

Da $|vwx| \leqslant N$, kann vwx nicht a's, b's und c's enthalten. Eines der 3 Symbole a, b, c kommt in vwx also nicht vor.

iv. Ich wähle $i = 0$
Betrachte $uv^0wx^0y = uwy$
$|uwy| = |uvwxy| - \underbrace{|vx|}_{>0} < 3N.$

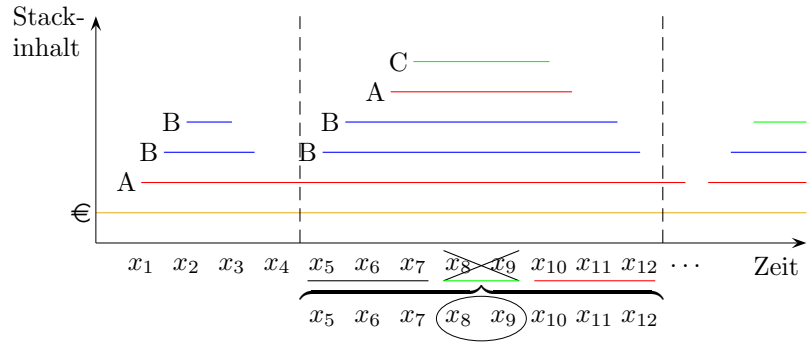
Aber das Symbol $(*)$ kommt in uwy N-mal vor. Also hat uwy nicht die Form $a^nb^nc^n$ für irgendein n. Also ist $uwy = uv^0wx^0y \notin L_{abc}$.

Übung

Zeige, dass $L = \{uu^Ru | u \in \{a, b\}^*\}$ keine NKA-Sprache ist.

3.1.6 Intuition für Pumping Lemma

$x = x_1x_2...x_n$

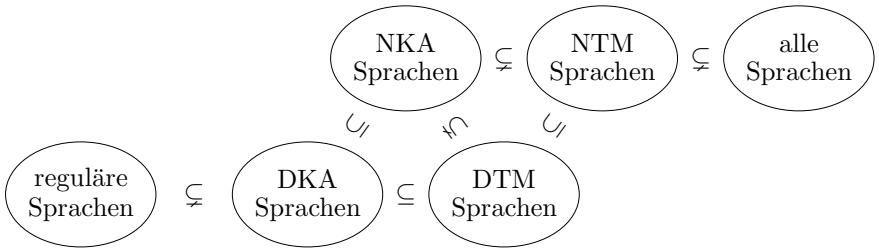


Behauptung

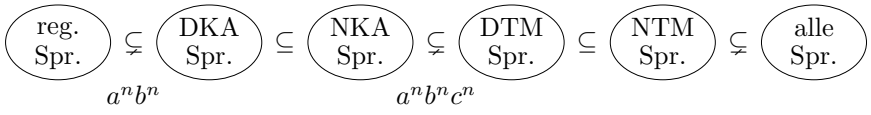
L_{abc} ist eine DTM-Sprache.

Korollar

NKA $\subsetneq$ NTM-Sprachen.



Damit lässt sich diese Anordnung aber wie folgt vereinfachen:



3.2 Eigenschaften von NKA-Sprachen (kontextfreien Sprachen)

3.2.1 Satz

Seien L, L' NKA-Sprachen und sei R eine reguläre Sprache.

i. $L \cap R$ ist eine NKA-Sprache.

ii. $L \cup L'$ ist eine NKA-Sprache.

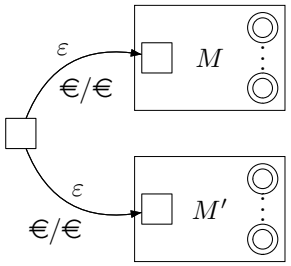
iii. $L \cap L'$ ist nicht unbedingt NKA-Sprache.

iv. $\overline{L}$ ist nicht unbedingt eine NKA-Sprache.

Beweis

i. Sei M ein NKA mit $L(M) = L$ und N ein NEA mit $L(n) = R$. Wir wollen zeigen, dass dann ein NKA M^* existiert, der x genau dann akzeptiert, wenn sowohl M wie auch N die Eingabe x akzeptieren. Die Idee hierbei ist, dass man M und N "parallel" auf die Eingabe x laufen lässt, die Zustandsmenge also durch $Q_M \times Q_N$ definiert.

ii. Seien M, M' NKA's für L und L'. Wir nehmen o.B.d.A. an, dass $\Gamma = \Gamma'$ und $€ = €'$ gilt. Wir wollen zeigen, dass dann ein NKA M^* existiert, der x akzeptiert, wenn M oder M' die Eingabe x akzeptiert.



iii. Man betrachte die Sprachen

$$L = \{a^n b^n c^m | n, m \in \mathbb{N}\} \text{ und}$$
$$L' = \{a^k b^n c^n | k, n \in \mathbb{N}\} \text{ als NKA} - \text{Sprachen.}$$
$$L \cap L' = L_{abc} \text{ ist aber } \textbf{keine} \text{ NKA} - \text{Sprache}$$

iv. Behauptung: $L = \overline{L_{abc}}$.
Es gilt: L ist NKA-Sprache. Aber $\overline{L} = \overline{\overline{L_{abc}}} = L_{abc}$ ist nicht NKA-Sprache.

$$\overline{L_{abc}} = \{a^k b^l c^m | \text{ mit } \neg(k = l = m)\}$$
$$\cup \{u \in \{a, b, c\}^* | \exists b \text{ vor einem } a \vee \exists c \text{ vor einem } a \vee \exists c \text{ vor einem } b\}$$

Man kann $\neg(k = l = m)$ auch schreiben als $k \neq l \vee l \neq m$. Dann ist $\{a^k b^l c^m | \text{ mit } \neg(k = l = m)\}$ aber gerade $\{a^k b^l c^m | k \neq l\} \cup \{a^k b^l c^m | l \neq m\}$, was eine NKA-Sprache ist.

3.3 Deterministische Kellerautomaten

Eine Maschine M sei wie folgt definiert:
$M = (\Sigma, \Gamma, \in, Q, s, F, \Delta)$ mit $\in \in \Gamma, s \in Q, F \subset Q, \Delta \subset (Q \times \Gamma \times (\Sigma \cup \{\varepsilon\})) \times \Gamma^* \times Q$.
Die Maschine M ist deterministisch, wenn in jeder Situation höchstens 1 Regel anwendbar ist, d.h.

i. $\forall (q, A, a) \in Q \times \Gamma \times (\Sigma \cup \{\varepsilon\}) \underset{\leqslant 1}{\exists} (U, q') \in \Gamma^* \times Q$ mit $(q, A, a, U, q') \in \Delta$.

ii. $(q, A, \varepsilon, U, q') \in \Delta \Rightarrow \forall a \in \Sigma$ gibt es keine Regel der Form $(q, A, a, U, q') \in \Delta, U \in \Gamma^*, q' \in Q$.

Wir wollen zeigen, dass $L = L(M)$ für einen DKA $M = (\Sigma, \Gamma, \in, Q, s, F, \Delta)$. Dann ist $\overline{L} = L(M')$ für irgendeinen DKA $M' = \big(\Sigma, \Gamma', \in', Q, s', F', \Delta\big)$.
Die Idee ist nun, dass wir $M' = M$ wählen, mit der Einschränkung, dass $F' = Q \setminus F$. Das kann aber problematisch sein, denn:

i. Bei der Eingabe x bleibt M wegen fehlenden Regel stecken,

ii. Bei der Eingabe x arbeitet M nicht die gesamte Eingabe ab, da sie in eine Schleife von ε-Regeln kommt.

iii. Nach vollkommenem Konsum der Eingabe x könnte M in einer Folge von ε-Regelanwendungen durch Zustände in F und in $Q \setminus F$ gehen.
(**Achtung:** Die Akzeptanzdefinition sagt, dass x akzeptiert wird, wenn es vollkommen konsumiert wird und irgendein in F erreicht werden kann.)

3.3.1 Lemma 1

Sei $M = (\Sigma, \Gamma, \in, Q, s, F, \Delta)$ ein DKA, dann existiert ein DKA M', der jede Eingabe x vollkommen konsumiert und $L(M') = L(M)$.

Beweis

Man führt einen Zustand $d \notin Q$ ein: $Q' = \{d\} \cup Q$. Dann:

i. $\forall (p, A, a) \in Q \times \Gamma \times \Sigma$, für die es keine Regel der Formen (p, A, aU, q) oder $(p, A, \varepsilon, U, q)$ gibt, füge eine Regel $(p, A, a, \in, d)$ zu Δ. Weiter füge Regeln $(d, \in, a, \in, d)$ zu Δ

ii. Jede Regel $(p, A, \varepsilon, \ldots, \ldots)$, die, wenn angewandt auf die Konfiguration (p, A, ε) zu einer unbeschränkten Rechenschrittfolge führt, in der keine Endzustände vorkommen, ersetze durch $(p, A, \varepsilon, \in, d)$. (**Achtung:** das ist nicht-konstruktiv spezifiziert, kann aber effektiv berechnet werden.)
Ansonsten, also wenn Endzustände vorkommen, ersetze es durch $(p, A, \varepsilon, \in, d')$, wobei $d' \in F$ ein neuer Zustand ist und führe $\forall a \in \Sigma$ die Regel $(d', \in, a, \in, d)$ ein.

3.3.2 Satz

Sei $L = L(M)$ für irgendeinen DKA M. Dann ist $\overline{L} = L(M^*)$ für irgendeinen DKA M^*.

Beweis

Verwende Lemma 1 und konstruiere M', mit $L(M') = L(M)$ und $M' = (\Sigma', \Gamma', \in, Q', s', F', \Delta')$ konsumiert jede Eingabe vollständig.
$Q^* = Q' \times \{0, 1\}$

$(q, 0) \rightarrow$ seitdem der letzte Buchstabe konsumiert wurde, wurde kein Endzustand erreicht.

$(q, 1) \rightarrow$ seitdem der letzte Buchstabe konsumiert wurde, wurde irgendein Endzustand erreicht.

$$
\begin{aligned}
(q, A, \varepsilon, U, q') \in \Delta'((q,0), A, \varepsilon, U, (q', 0)) \quad & \text{falls } q' \notin F' \\
((q,0), A, \varepsilon, U, (q', 1)) \quad & \text{falls } q' \in F' \\
((q,1), A, \varepsilon, U, (q', 1)) \quad & \\
(q, A, a, U, q') \in \Delta'((q,0), A, a, U, (q', 0)) \quad & \text{falls } q' \notin F' \\
((q,0), A, a, U, (q', 1)) \quad & \text{falls } q' \in F' \\
((q,1), A, a, U, (q', 0)) \quad & \text{falls } q' \notin F' \\
((q,1), A, a, U, (q', 1)) \quad & \text{falls } q' \in F'
\end{aligned}
$$

$$F^* = \left\{ (q,0) |\, q \notin F' \right\}_{q \in Q' \setminus F'}$$

3.3.3 Korollar

$\overline{L_{abc}}$ ist keine DKA-Sprache, aber eine NKA-Sprache. Wäre nämlich $\overline{L_{abc}}$ eine DKA-Sprache, dann wäre $\overline{\overline{L_{abc}}} = L_{abc}$ auch eine DKA-Sprache, was aber allein schon deswegen nicht sein kann, weil L_{abc} noch nicht mal eine NKA-Sprache ist. Daraus folgern wir:

3.3.4 Satz

DKA-Sprachen $\subsetneq$ NKA-Sprachen.

3.3.5 Satz

Wenn $L \subset \{a\}^*$ und L NKA-Sprache ist, dann ist L regulär.

3.3.6 Beweis des Pumping Lemmas für NKA-Sprachen

Sei L ein NKA, dann

$$\exists N \forall \underset{|z| \geqslant N}{z \in L} \exists \underset{|vx|>0, |vwx| \leqslant N, k \leqslant N}{z = uvwxy} \quad \forall i \in \mathbb{N} : uv^i wx^i y \in L$$

Bei $\Sigma = \{a\}$ bedeutet das:

$$\exists N \in \mathbb{N} \forall \underset{|z| \geqslant N, z = a^m}{z \in L} \exists k > 0 \forall i \in \mathbb{N} a^{m-k} a^{ik} \in L$$

Behauptung

$\exists N \forall \underset{|z| \geqslant N}{z \in L} : \forall i \in \mathbb{N} : za^{N!i} \in L$. Nach dem Pumping Lemma gilt, dass

$$\forall z \in L \; \exists 0 < k \leqslant N \; \forall j \in \mathbb{N} : za^{kj} \in L.$$

Betrachte $j = \frac{N!}{k} i \Rightarrow kj = N! \cdot i$. $\forall w \in L, |w| > N$ gilt:

$$m_w := \min \{r > N | r + iN! = |w|\}$$

für irgendein $i \in N$.
$B = \{m_w | w \in L, |w| > N\}$ ist endlich, enthält höchstens $N!$ viele Elemente.

$$L = \underbrace{\{w \in L | \; |w| \leqslant N\}}_{\text{endlich, regulär}} \cup \bigcup_{m \in B} \underbrace{\{a^m a^{iN!} | i \in \mathbb{N}\}}_{\text{regulär}}$$

3.3.7 Satz

Sei L eine NKA-Sprache. Dann

$\exists$ NKA M' mit $L_\varepsilon(M') = L$ und M' verwendet nur **einen** Zustand.

3.3.8 Anmerkung

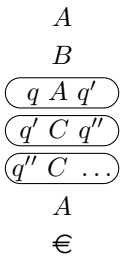
Sei $Q' = \{s'\}$. Alle Regeln haben die Form (s', A, a, U, s'). Der Zustand s' ist für die Regeln irrelevant. Eine Regel hat also die Form (A, a, U).

Beweis

Sei L eine NKA-Sprache. Dann ist $L = L_\varepsilon(M)$ für irgendeinen NKA $M = (\Sigma, \Gamma, Q, s, \Delta)$ (Da der Automat über einen leeren Stack akzeptiert, braucht hier keine Menge der Endzustände mit deklariert werden).

Idee

Wir konstruieren eine Maschine M' aus M, die ohne Zustände auskommen muss. Ohne Beschränkung der Allgemeinheit gilt, dass alle Regeln von M die Form (p, A, a, U, q) haben. Hierbei ist entweder $U = \varepsilon$, $U = B$ oder $U = BA$ mit $B \in \Gamma$. Der Fall $U = \varepsilon$ entspricht hierbei der Operation POP, der Fall $U = BA$ PUSH. M' soll M „simulieren“. Die Zustandsinformation von M muss im Keller von M' gespeichert werden. $\Gamma' = Q \times \Gamma \times Q \cup \{€'\}$. Hierbei bedeutet (q, A, q'): Wenn (diese Instanz von) A das nächste Mal oberstes Kellersymbol von M ist, dann ist M im Zustand q und nachdem A vom Keller entfernt worden sein wird, ist M im Zustand q'.



Die „Simulation“ des Automaten M durch M' hat die Form

$$(p_k, A_k, p_{k-1})\,(p_{k-1}, A_{k-1}, p_{k-2}) \ldots (p_2, A_2, p_1)\,(p_1, A_1, p_0)$$

Es gilt:

$$\Delta' \subset \cancel{Q' \times} \Gamma' \times (\Sigma \cup \{\varepsilon\}) \times \Gamma'^* \cancel{\times Q'},$$

das heißt $\left(€', \varepsilon, (s, €, p)\right)$ für jedes $p \in Q$ Ist

- $(q, A, a, B, q') \in \Delta$, dann $((q, A, p), a, (q', B, p))\ \forall p \in Q$, zu Δ' dazu.
- $(q, A, a, BA, q') \in \Delta$, dann $((q, A, p), a, (q', B, p')\,(p', A, p))\ \forall p, p' \in Q$, zu Δ' dazu.
- $(q, A, a, \varepsilon, q') \in \Delta$, dann $((q, A, q'), a, \varepsilon)$, zu Δ' dazu.

Die Maschine M' ist spezifiziert durch:

$$\begin{aligned}
&\Gamma' = \text{Kellersymbole}\\
&\Sigma' = \Sigma\\
&€' = \text{Kellerbodensymbol}\\
&\Delta'
\end{aligned}$$

3.3.9 Behauptung

$L_\varepsilon(M) = L_\varepsilon(M')$.

Beweis

Man muss zeigen, dass folgendes gilt:

$$(s, €, x) \underset{M}{\vdash}^* (q, \varepsilon, \varepsilon) \text{ für irgendein } q \in Q \Leftrightarrow \left(, €', x\right) \underset{M'}{\vdash}^* (, \varepsilon, \varepsilon).$$

Man zeigt über Induktion, dass gilt:

$$\begin{aligned}
&(s, €, x) \underset{M}{\vdash}^m (q, A_k A_{k-1} \ldots A_1, y)\\
\Leftrightarrow &\left(, €', x\right) \underset{M'}{\vdash}^{m+1} (, (q, A_k, p_{k-1})\,(p_{k-1}, A_{k-1}, p_{k-2})\,() \ldots (p_1, A_1, p_0), y)
\end{aligned}$$

$\forall p_0, \ldots, p_{k-1} \in Q$.

Kapitel 4

Grammatiken

Ein alternativer Mechanismus für die endliche Spezifizierung von Sprachen sind Grammatiken.

4.1 Definition

Eine Grammatik G ist ein Viertupel $G = (\Sigma, V, S, P)$, wobei hier

Σ	die endliche Menge der Terminale darstellt
V	die Variablen, die sogenannten Nichtterminale, sind
$S \in V$	das „Startsymbol“ ist und
$P \subset \left((\Sigma \cup V)^* V (\Sigma \cup V)^*\right) \times (\Sigma \cup V)^*$	die Produktionen sind.

$(\alpha, \beta) \in P$ wird geschrieben als $\alpha \to \beta$. Hierbei induziert G die Relation

- $\Rightarrow_G$ auf $(V \cup \Sigma)^*$ durch die Vorschrift $\alpha\beta\gamma \Rightarrow_G \alpha\beta'\gamma$, wenn $\beta \to \beta' \in P$.
- $\Rightarrow_G^*$ ist die reflexive transitive Hülle von $\Rightarrow_G$.

Die von G generierte Sprache ist $L(G) = \{w \in \Sigma^* | S \Rightarrow_G^* w\}$. Eine Folge

$$\alpha_1 \Rightarrow_G \alpha_2 \Rightarrow_G \alpha_3 \Rightarrow_G \ldots \Rightarrow_G \alpha_k$$

nennt man **Ableitung** (oder **Derivation**) von α_k aus α_1.

Beispiel

Seien

$$
\begin{aligned}
&G = (V, \Sigma, S, P)\\
&\Sigma = \{a, b, c\}\\
&V = \{S, A, B, C\}\\
&P = \{ \qquad S \to \varepsilon \qquad (1)\\
&\qquad\quad S \to aDbc \qquad (2)\\
&\qquad\quad D \to aDbC \qquad (3)\\
&\qquad\quad D \to \varepsilon \qquad (4)\\
&\qquad\quad Cb \to bC \qquad (5)\\
&\qquad\quad Cc \to cc \qquad (6)\ \}
\end{aligned}
$$

Eine Ableitung ist z.B.:

$$
\begin{aligned}
&\underline{S} \underset{(2)}{\Rightarrow} a\underline{D}bc && \underset{(3)}{\Rightarrow} aa\underline{D}bCbc && \underset{(3)}{\Rightarrow} aaa\underline{D}bCbCbc\\
&\underset{(4)}{\Rightarrow} aaab\underline{Cb}Cbc && \underset{(5)}{\Rightarrow} aaabbC\underline{Cb}c && \underset{(5)}{\Rightarrow} aaabb\underline{Cb}Cc\\
&\underset{(5)}{\Rightarrow} aaabbbC\underline{Cc} && \underset{(6)}{\Rightarrow} aaabbb\underline{Cc}c && \underset{(6)}{\Rightarrow} \underbrace{aaabbbccc}_{\in L(G)}
\end{aligned}
$$

Behauptung

Sei $L(G) = \{a^n b^n c^n | n \in \mathbb{N}\} = L_{abc}$. Dann gilt $\forall n \in N : S \Rightarrow_G^* a^n b^n c^n$.

4.2 Klassifizierung von Grammatiken (Chomsky-Hierarchie)

Es gibt vier Typen:

- **Typ 0:** Keine Einschränkungen auf die Form der Produktionen.
- **Typ 1 (kontextsensitiv):** Die Produktionen müssen folgende Form haben: $\alpha \to \beta$ mit $|\alpha| \leqslant |\beta|$, in Ableitungen dürfen Strings also nicht kürzer werden.
- **Typ 2 (kontextfrei):** Die Produktionen müssen folgende Form haben: $A \to \alpha; A \in V$
- **Typ 3 (rechtslinear):** Die Produktionen müssen folgende Formen haben: $A \to aB,\ A \to a,\ A \to \varepsilon,\ a \in \Sigma, A, B \in V$.

Sprachen, die durch Typ i Grammatiken erzeugt werden können, heißen Typ i Sprachen.

Beispiele

- **Typ 1:** L_{abc}.
- **Typ 2:**

$$\begin{aligned} E &\to T|E+T \\ T &\to F|T*F \\ F &\to 0K|aW|(E) \\ K &\to 1K|0K|\varepsilon \\ W &\to aW|bW|\varepsilon \end{aligned}$$

  mit

$$\begin{aligned} V &= \{E,T,F,K,W\} \\ \Sigma &= \{+,*,(,),1,0,a,b\} \\ S &= E \end{aligned}$$

  Grammatik für geklammerte arithmetische Ausdrücke mit Operatoren $+,*$, Operanden Binärzahlen und Variablennamen $\in a\{a,b\}^*$.

- **Typ 3:** $W \to aW|bW|\varepsilon$ (erzeugt $\{a,b\}^*$).

4.2.1 Satz

L ist regulär genau dann, wenn $L = L(G)$ für eine rechtslineare Grammatik (Typ 3).

Beweis

„$\Leftarrow$“ Gegeben sei eine rechtslineare Grammatik $G = (\Sigma, V, S, P)$. Wir wollen zeigen, dass $L(G)$ nur endlich viele Fortsetzungssprachen hat. Es gilt aber, dass

$$\forall A \in V : L_A = \{v \in \Sigma^* | A \Rightarrow_G^* v\}$$

und

$$\forall u \in \Sigma^* : V_u = \{A \in V | S \Rightarrow_G^* uA\}$$

$F_L(u) = \bigcup \{L_A | A \in V_u\}$ ist durch $V_u \subset V$ vollständig bestimmt. Es gibt nur endliche viele V_U's (Teilmengen von V).

„$\Rightarrow$“ Ist L regulär, dann ist $L = L(M)$ für endliche Automaten. Gegeben ist also bereits ein Automat $M = (\Sigma, Q, s, F, \Delta)$ mit oBdA $\Delta = \{(q,a,q')\}$. Man konstruiert davon ausgehend eine Grammatik G mit

$$\begin{aligned} G &= (\Sigma, Q, s, P), \\ P &= \{q \to aq' | (q,a,q') \in \Delta\} \cup \{q \to \varepsilon | q \in F\} \end{aligned}$$

Behauptung

$$L(M) = L(G).$$

Es gilt:

$$(s, xy) \underset{M}{\vdash}^{*} (q, y) \Leftrightarrow s \Rightarrow_G^{*} xq \quad \forall x, y \in \Sigma^{*}$$

Induktion über Länge der Rechenschritte:

$i = 0$ $(s, y)\, s\checkmark$

$i > 0$

$$(s, xy) \underset{M}{\vdash}^{i-1} (q, y) \Leftrightarrow \qquad s \Rightarrow_y^{i-1} xq$$

$$(q, y) \vdash (q', y_2 \dots y_l) \text{ falls } (q, y_1, q') \in \Delta \quad xq \Rightarrow xy_1q' \text{ falls } q \to y_1q' \text{ in } P$$

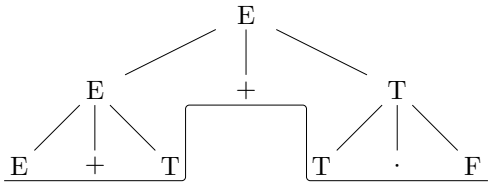
Zu zeigen bleibt immer noch der Sonderfall für Endzustände.

4.3 Kontextfreie Grammatiken und Sprachen

Fortsetzung von Beispiel:

$$E \Rightarrow \begin{matrix} \underline{E} + T \\ E + \underline{T} \end{matrix} \Rightarrow \begin{matrix} E + T + \underline{T} \\ \underline{E} + T * F \end{matrix} \Rightarrow E + T + T * F$$

Linksableitung
Im abgeleiteten String wird eine Produktion **immer** auf die linkeste Variable angewandt. Holistische Sichtweise aller Ableitungsvariationen eines Strings. Man erhält einen Ableitungsbaum:



4.3.1 Definition:

Ein Baum mit Knotenmarkierungen $\in \Sigma \cup V$, bei dem die inneren Knoten mit $\in V$ markiert sind, nennt man **Ableitungsbaum**. k (markiert mit X) ist ein innerer Knoten mit Kindern $k_1, \dots, k_i$ (markiert mit $X_1, \dots, X_i$), dann muss $X \to X_1 \dots X_i$ eine Produktion sein.

4.3.2 Lemma:

Sei $G = (\Sigma, V, S, P)$ kontextfrei, ferner $A \in V$ und $\alpha \in (V \cup \Sigma)^{*}$. $A \Rightarrow_G^{*} \alpha$ gilt genau dann, wenn $\exists$ eine Linksableitung von A nach α und damit genau dann wenn $\exists$ ein Ableitungsbaum mit Wurzelmarkierung A und Blattmarkierungen (in natürlicher links-rechts Reihenfolge) $\alpha_1, \alpha_2, \dots, \alpha_{|\alpha|}$

4.3.3 Satz

Eine Sprache L ist genau dann kontextfrei, wenn L eine NKA-Sprache ist. Analog kann man auch sagen, eine kontextfreie Grammatik G mit $L(G) = L$ existiert genau dann, wenn ein NKA M mit $L_\varepsilon(M) = L$ ist.

Beweis

„⇒“ Gegeben sei eine kontextfreie Grammatik $G = (\Sigma, V, S, P)$. Wir wollen einen NKA M konstruieren, dessen Rechenschrittfolgen genau zu Linksableitungen in G korrespondieren. $M = (\Sigma, \Sigma \cup V, \{s\}, s, S, \Delta)$ mit $\Delta = \{(s, A, \varepsilon, \alpha, s) \,|A \to \alpha \in P\} \cup \{(s, a, a, \varepsilon, s) \,|a \in \Sigma\}$

Behauptung

$$(s, S, xy) \underset{M}{\vdash}^* (s, \gamma, y) \Leftrightarrow S \underset{\text{links}}{\Rightarrow}^*_G x\gamma$$

Beweis

erfolgt über Induktion:

$i = 0$ (s, S, y)

$i > 0$

$$(s, S, xy) \underset{M}{\vdash}^{i-1} (s, \gamma, y) \Leftrightarrow S \underset{\text{links}}{\Rightarrow}^{i-1} x\gamma \Rightarrow x\alpha\overline{\gamma} \qquad \text{für } \gamma_1 \in V; \gamma_1 \to \alpha \in P$$

$$\vdash (s, \alpha\overline{\gamma}, y) \qquad \text{für } \gamma_1 \in V_i, \gamma_1 \to \alpha \in P, \text{ d.h. } (s, \gamma_1, \varepsilon, \alpha, s) \in P$$

$$(s, \gamma, y) \vdash (s, \overline{\gamma}, \overline{y}) \qquad \text{für } \gamma_1 \in \Sigma, \gamma_1 = y_1, \text{ d.h. } (s, \gamma_1, \gamma_1, \varepsilon, s) \in \Delta$$

$x\gamma \approx (x\gamma_1)\,\overline{\gamma}, \gamma_1 \in \Sigma$.
Hierbei ist $\gamma = \gamma_1\overline{\gamma}$ und $y = y_1\overline{y}$.

„⇐“ Sei L eine NKA-Sprache. Dann existiert ein NKA $M = (\Sigma, \Gamma, Q, s, €, \Delta)$ mit $Q = \{s\}$ und $L_\varepsilon(M) = L$. Alle Regeln in Δ haben die Form (s, A, a, X, s) mit $a \in \Sigma \cup \{\varepsilon\}$ und $X \in \Gamma^*$.
Wir wollen eine Grammatik G konstruieren mit $L(G) = L_\varepsilon(M)$.

$$G = (\Sigma, V, S, P)$$
$$V = \Gamma$$
$$S = €$$
$$P = \{A \to aX \,|\, (s, A, a, X, s) \in \Delta\}$$

Behauptung 1:

Sei $w \in \Sigma^*$.

$$w \in L(G) \Leftrightarrow w \in L_\varepsilon(M)$$
$$€ \Rightarrow^*_G w \Leftrightarrow (s, €, w) \underset{M}{\vdash}^* (s, \varepsilon, \varepsilon)$$

Behauptung 2:

Es gilt

$$\underbrace{\in \Rightarrow_G^i xU}_{\text{Linksableitung}} \quad \Leftrightarrow \forall y \in \Sigma^* : (s, \in, xy) \underset{M}{\vdash}^i (s, U, y)$$

Beweis:

Wir beweisen dies mit Induktion über i.
Schritt von i nach $i+1$:

$$\in \Rightarrow_G^i xU = x\underline{A}U' \qquad \Rightarrow_G xaXU'$$

also

$$\begin{array}{lll} & \in \Rightarrow_G^{i+1} & xaXU' \\ \Leftrightarrow & (s, \in, xy) \underset{M}{\vdash}^i (s, U, y) = (s, AU', ay') & \\ & \underset{M}{\vdash} (s, XU', y') & \end{array}$$

mit $U = AU'$, $A \in V$ und $y = ay'$, $a \in \Sigma \cup \{\varepsilon\}$. Gibt es in der Maschine eine Regel der Form $(s, A, a, X, s) \in \Delta$, so gibt es eine Produktion der Form $A \to aX \in P$.

4.4 Normalformen für kontextfreie Grammatiken

4.4.1 Definition

Eine Grammatik $G = (\Sigma, V, S, P)$ ist in Chomsky-Normalform, wenn alle Produktionen in P eine der folgenden Formen haben:

$$A \to a$$
$$A \to BC$$
$$S \to \varepsilon$$

mit $A, B, C \in V$ und $a \in \Sigma$.
In letzterem Fall darf aber S auf der rechten Seite keiner Produktion vorkommen.
Man beachte: Ableitungsbäume sind bei der Chomsky-Normalform binär (außer vor Blattlevel).

4.4.2 Definition

Eine Grammatik $G = (\Sigma, V, S, P)$ ist in Greibach-Normalform, wenn alle Produktionen P folgende Form haben:

$$A \to aU$$
$$S \to \varepsilon$$

mit $A \in V$, $a \in \Sigma$, $U \in V^*$. In letzterem Fall darf aber S auf der rechten Seite keiner Produktion vorkommen.

4.4.3 Satz

Folgende Aussagen sind äquivalent:

- L ist eine kontextfreie Sprache.
- Es gibt eine kontextfreie Grammatik G in Chomsky-Normalform mit $L(G) = L$.
- Es existiert eine kontextfreie Grammatik G in Greibach-Normalform mit $L(G) = L$.

4.4.4 Beweis des Pumping-Lemmas für kontextfreie Sprachen

Zu zeigen ist, dass, wenn L kontextfrei ist, dann $\exists N \in \mathbb{N} \forall z \in L, |z| \leqslant N, \exists$ Unterteilung $z = uvwxy \, \forall i \in \mathbb{N} : uv^i wx^i y \in L$.

Beweis

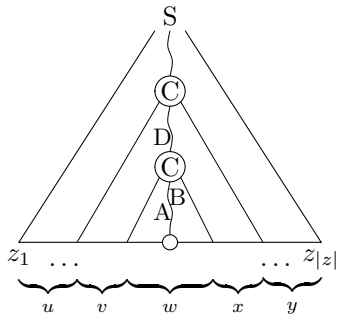
Sei L kontextfrei. Dann $\exists G = (\Sigma, V, S, P)$ in Chomsky-Normalform mit $L(G) = L$. Wir wählen nun $N = 2^{|v|+2}$ und betrachten $z \in L(G), |z| \geqslant N, \exists$ einen binären Ableitungsbaum mit z als Blattbeschriftung. Die Anzahl der Blätter beträgt hierbei

$$\#\text{ Blätter } = |z| \geqslant N = 2^{|v|+2}$$

Auf Grund der Tatsache, dass der Baum binär ist, folgt, dass

$$\text{Höhe } \geqslant |v| + 2,$$

es gibt also Blätter mit einem Abstand $\geqslant |v| + 2$ von der Wurzel. Das bedeutet also, dass der Pfad zur Wurzel $\geqslant |v| + 1$ innere Knoten hat, die mit Variablen beschriftet sind. Somit muss sich irgendein Nichtterminal wiederholen. Betrachte die „unterste“ Wiederholung.



4.5 Das „Wortproblem“ für kontextfreie Sprachen

Hierbei geht es um die automatische Beantwortung der Frage, ob $x \in L(G)$. $G = (\Sigma, V, S, P)$ sei ohne Beschränkung der Allgemeinheit in Chomsky-Normalform. Dann ist

$$x = x_1 \dots x_{n-1} \in \Sigma^*,$$

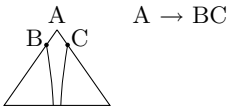
wobei $x[i:j] = x_i x_{i+1} \dots x_{j-1}$ mit $x = x[0:n]$. Dann ist

$$V[i;j] = \{A \in V | A \Rightarrow_G^* x[i:j]\}$$

Idee

Man berechne $V[0:n]$ und prüfe, ob es S enthält. Hierbei berechne man im Generellen zuerst $V[i:i+k]$ $\forall i$ nach steigendem k und stellt man fest, dass

$$A \in V[i:j] \Leftrightarrow \exists k : B \in V[i:k] \wedge C \in V[k:j] \wedge A \to BC \in P :$$




```
for i = 0 to n-1
    V[i:i+1] = { A ∈ V | A → x_i ∈ P }
for k = 2 to n
    for i = 0 to n-k
        V[i:i+k] = ∅
        for l = 1 to k-1
                                                  A → BC ∈ P
            V[i:i+k] = V[i:i+k] ∪{ A ∈ V |  B ∈ V[i:i+l]   }
                                                C ∈ V[i+l:i+k]
```

Wir erhalten eine Laufzeit von $\mathcal{O}(n^3)$. Diesen Algorithmus nennt man

4.5.1 CYK-Algorithmus

Er ist benannt nach Cocke, Younger und Kasami. Ein darauf basierender, aber verbesserter Algorithmus von Valient hat eine Laufzeit von $\mathcal{O}(n^{\log_2 7})$ Man nimmt an, dass Laufzeiten bis minimal $\mathcal{O}(n^2)$ möglich sind, konnte diese Annahme aber bisher nicht beweisen.

4.6 Mehrdeutigkeit

Heißt eine kontextfreie Grammatik G mehrdeutig, dann existiert $u \in L(G)$ mit 2 verschiedenen Ableitungsbäumen.
Eine kontextfreie Sprache L heißt inhärent mehrdeutig, wenn jede kontextfreie Grammatik G mit $L(G) = L$ mehrdeutig ist.

Beispiel

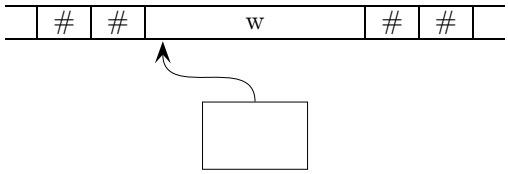
$L = \{a^n b^n c^m | n, m \in \mathbb{N}\} \cup \{a^k b^n c^n | k, n \in \mathbb{N}\}$.

Kapitel 5

Turing-Maschinen

5.1 Einfache Turing-Maschine

Sei $M = (\Sigma, \Gamma, \#, Q, s, F, \Delta)$:



Σ Eingabealphabet

$\Gamma \supset \Sigma$ Arbeitsalphabet

$\# \in \Gamma \setminus \Sigma$ Leerzeichen

Q endliche Zustandsmenge

$s \in Q$ Startzustand

$F \subset Q$ Endzustände

$\Delta \subset (Q \times \Gamma) \times (Q \times \Gamma \times \{L, B, R\})$ Übergangsrelation

5.1.1 Konfigurationen

Eine Konfiguration genügt der Form

$$\Gamma^* Q \Gamma^*,$$

besteht also aus den Elementen

- Bandinhalt,
- Zustand und
- Kopfposition.

So bedeutet also $A_1 A_2 \dots A_k q A_{k+1} \dots A_m$:

- Bandinhalt: $A_1 \dots A_m$.
- Zustand: q.
- Kopfposition: Am Zeichen A_{k+1}.

Anfangskonfiguration

für $x \in \Sigma^*$:

$$Init_x = sx, \quad Init_\varepsilon = s\#$$

Endkonfiguration

$$Fin = UfV \text{ mit } U, V \in \Gamma^*, f \in F.$$

5.1.2 Rechenschrittrelationen

Schritt nach links:

$$\begin{aligned} & A_1 \dots A_k q A_{k+1} \dots A_m \underset{M}{\vdash} A_1 \dots A_{k-1} q' A_k B A_{k+2} \dots A_m, \\ \Leftrightarrow \quad & (q, A_{k+1}, q', B, L) \in \Delta \end{aligned}$$

Keine Bewegung:

$$\begin{aligned} & A_1 \dots A_k q A_{k+1} \dots A_m \underset{M}{\vdash} A_1 \dots A_k q' B A_{k+2} \dots A_m \\ \Leftrightarrow \quad & (q, A_{k+1}, q', B, B) \in \Delta \end{aligned}$$

Schritt nach rechts:

$$\begin{aligned} & A_1 \dots A_k q A_{k+1} \dots A_m \underset{M}{\vdash} A_1 \dots A_k B q' A_{k+2} \dots A_m \\ \Leftrightarrow \quad & (q, A_{k+1}, q', B, R) \in \Delta \end{aligned}$$

Bandinhaltsenden

Schritt nach links:

$$\begin{aligned} & q A_1 \dots A_m \underset{M}{\vdash} q' \# B A_2 \dots A_m \\ \Leftrightarrow \quad & (q, A_1, q', B, L) \in \Delta \end{aligned}$$

Schritt nach rechts:

$$\begin{aligned} & A_1 \dots A_{m-1} q A_m \underset{M}{\vdash} A_1 \dots A_{m-1} B q' \# \\ \Leftrightarrow \quad & (q, A_m, q', B, R) \in \Delta \end{aligned}$$

$\underset{M}{\vdash}^*$ ist die reflexive transitive Hülle von $\underset{M}{\vdash}$. Ein Wort $x \in \Sigma^*$ wird von M genau dann akzeptiert, wenn

$$\mathrm{Init}_x \underset{M}{\vdash}^* q$$

für irgendein $q \in Fin$ gilt.
Sei

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

$L \subset \Sigma^*$ heißt genau dann einfache TM-Sprache, wenn es eine einfache Turing-Maschine M gibt mit $L(M) = L$.

5.1.3 Satz

Genau dann wenn L eine einfache TM-Sprache, ist L eine Typ-0 Sprache.

Beweis

„⇒“ Sei L eine einfache TM-Sprache. Dann $\exists$ eine einfache Turing-Maschine $M = (\Sigma, \Gamma, \#, Q, s, F, \Delta)$ mit $L(M) = L$. Wir suchen eine Grammatik G mit $L(G) = L$. Die Idee ist nun, dass die Ableitungen in G umgekehrte akzeptierte Rechenschrittfolgen von M darstellen sollen.

$$\begin{array}{rccccc} & s\beta^0 & \underset{M}{\vdash} \alpha^1 q^1 \beta^1 & \underset{M}{\vdash} \alpha^2 q^2 \beta^2 & \underset{M}{\vdash} \dots & \underset{M}{\vdash} a^N f \beta^N \\ \beta^0 = x \Leftarrow_G & s\beta^0 & \Leftarrow_G & \Leftarrow_G & \Leftarrow_G & \Leftarrow_G \end{array}$$

Idee

i. Die Grammatik erzeugt zuerst $q \in Fin$, also eine „geeignete“ Rechenfolge.

ii. Sie verfolgt die umgekehrte akzeptierte Rechenschrittfolge $Init_x \underset{M}{\vdash}^* q$.

Sei $G = (\Sigma, V, S, P)$, $V = \{S\} \cup Q \cup \Gamma \setminus \Sigma$.

i.

$$\begin{array}{rlcl} & \{ & S \to AS & | \quad A \in \Gamma \quad \} \\ \cup & \{ & S \to SA & | \quad A \in \Gamma \quad \} \\ \cup & \{ & S \to f & | \quad f \in F \quad \} \end{array}$$

ii.

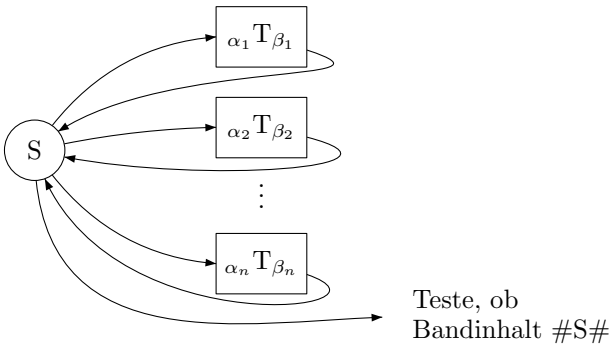
$$\begin{array}{rlcl} & \{ & q'AB \to AqA' & | \quad (q, A', q', B, L) \in \Delta \quad \} \\ \cup & \{ & q'B \to qA & | \quad (q, A, q', B, B) \in \Delta \quad \} \\ \cup & \{ & Bq' \to qA & | \quad (q, A, q', B, R) \in \Delta \quad \} \\ \cup & \{ & q'\#B \to qA & | \quad (q, A, q', B, L) \in \Delta \quad \} \\ \cup & \{ & Bq'\# \to qA & | \quad (q, A, q', B, R) \in \Delta \quad \} \end{array}$$

„⇐“ Sei L eine Typ-0 Sprache. Dann $\exists$ eine Grammatik $G = (\Sigma, V, S, P)$ mit $L(G) = L$. Gesucht ist eine einfache Turing-Maschine M mit $L(M) = L$. Die Idee ist nun, dass M die Ableitung $S \Rightarrow_G^* x$ in umgekehrter Reihenfolge verfolgt.

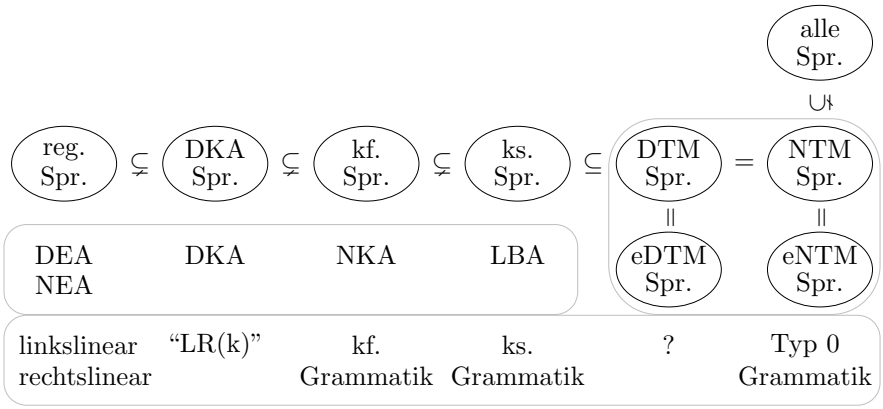
Sei ${}_\alpha T_\beta$ mit $\alpha, \beta \in \Gamma^*$ eine einfache Turing-Maschine mit folgender Funktionsweise:

- **Anfang:**
  Im Zustand $s_{\alpha\beta}$ mit Kopf ganz links
- **Ende:**
  Im Zustand $S_{\alpha\beta}$ mit Kopf ganz links und auf dem Band wurde ein Vorkommen von α durch β ersetzt. $x\alpha y \rightsquigarrow x\beta y$.

Die Gesamtmaschine M besteht aus ${}_\alpha T_\beta$ für jede Produktion $\beta \to \alpha \in P$.



Somit lässt sich unsere Sprachehierarchie wie folgt vereinfachen:



reg. Ausdrücke

Myhill-Nerode

5.1.4 Definition

Eine einfache Turing-Maschine heißt linear beschränkt, wenn sie keine Bandzelle verwendet, die nicht anfangs die Eingabe enthält.

5.1.5 Behauptung

L kontextsensitiv $\Leftrightarrow \exists$ linearbeschränkte eTM M mit $L(M) = L$

$A_1 \dots A_{i-1} (qA_i) \dots A_n$

5.1.6 Fakt 1

Sei L eine e$\begin{matrix}\mathrm{D}\\\mathrm{N}\end{matrix}$TM-Sprache, dann ist L eine $\begin{matrix}\mathrm{D}\\\mathrm{N}\end{matrix}$TM-Sprache.

Beweis

- Man kopiert die Eingabe vom Eingabeband auf das Arbeitsband.
- Man simuliert die einfache Turing-Maschine auf dem Arbeitsband.

5.2 k-Band Turing-Maschinen

5.2.1 Definition

Eine k-Band Turing-Maschine hat

- 1 Eingabeband (ein Lesekopf),
- k unabhängige Arbeitsbänder, die anfangs leer sind, mit je einem Lese-Schreibkopf und
- einer endlichen Kontrolle.

Sie hat folgende Rechenschrittregeln:

$$\left(Q \times \Sigma \times \Gamma^k\right) \times \left(Q \times \left(\Gamma \times \{L, B, R\}\right)^k \times \{L, B, R\}\right)$$

$\{L, B, R\}$ beschreibt hierbei die Bewegung des E-Lesekopfes.

5.2.2 Fakt 2

Wird L von einer $\begin{matrix}\mathrm{D}\\\mathrm{N}\end{matrix}$ Turing-Maschine akzeptiert, dann wird L von einer k-Band $\begin{matrix}\mathrm{D}\\\mathrm{N}\end{matrix}$ Turing-Maschine akzeptiert. Dabei sei $k \geqslant 1$.

5.2.3 Satz

Wird L von einer k-Band $\begin{matrix}\mathrm{D}\\\mathrm{N}\end{matrix}$ Turing-Maschine M akzeptiert, dann existiert eine einfache Turing-Maschine M', die L akzeptiert.

5.2.4 Korollar

eDTM, k-Band DTM und DTM generieren die gleichen Sprachklassen. Analoges gilt für den nichtdeterministischen Fall.

Beweis

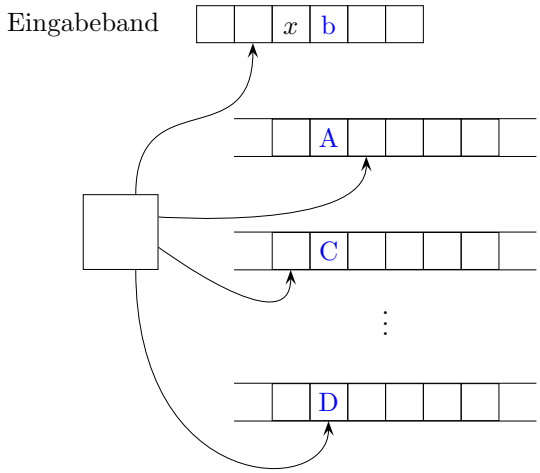
Sei $M = (\Sigma, \Gamma, \#, Q, s, F, \Delta)$ eine k-Band DTM mit

$$\left(Q \times \Sigma \times \Gamma^k\right) \times \left(Q \times (\Gamma \times \{L, B, R\})^k \times \{L, B, R\}\right).$$

Wir wollen zeigen, dass es eine einfache DTM $M' = (\Sigma, \Gamma', \#', Q', s', F', \Delta')$ mit $x \in L(M) \Leftrightarrow x \in L(M')$ gibt.

Idee

M' simuliert die Vorgehensweise von M. Die Maschine M' muss also „Konfigurationen“ von M darstellen können:



$\Gamma' = \Sigma \cup (\Gamma \times \{0,1\})^{k+1}$. Man muss sich vorstellen, dass das Band von M' $k+1$ Spuren hat, nämlich eine pro Band von M.

	$b/0$	Eingabeband
	A/0	Arbeitsband 1
	C/0	Arbeitsband 2
	D/1	Arbeitsband 3

Ein Eintrag $(a, 0)$ auf dem $i+1$-ten Band bedeutet, in entsprechender Zelle des i-ten A-Bandes von M steht das Symbol a und der Lese-Schreibkopf **ist nicht** über dieser Zelle. Ein Eintrag $(a, 1)$ auf dem $i+1$-ten Band bedeutet, in entsprechender Zelle des i-ten A-Bandes von M steht das Symbol a und der Lese-Schreibkopf **ist** über dieser Zelle.

Simulation

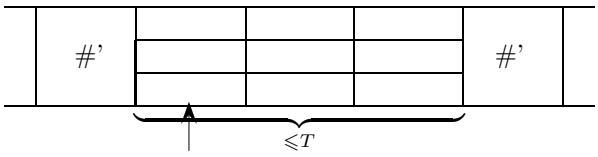
i. „Formatiere Band“, so dass Eingabe in 0-ter Spur und die anderen Spuren mit # gefüllt.

	#’	a	c	c	#’	#’	

wird somit zu

	#’	a,1	c,0	c,0	#’	#’	
		#,1	#,0	#,0			
		#,1	#,0	#,0			

ii. M' simuliert jeden Schritt von M auf folgende Art und Weise:



0. Bei Beginn der Schrittsimulation ist der Lese-Schreibkopf von M' ganz links. M' speichert den Zustand von M in einer endlichen Kontrolle.
1. Stelle fest, was von den $k+1$ Köpfen von M gelesen würde. Dazu müssen wir $k+1$ mal über den gesamten Bandinhalt laufen und in der jeweils i-ten Spur Kopfposition finden und uns das Symbol merken.
2. Jetzt sind M' der derzeitige Zustand von M und die $k+1$ gelesenen Symbole bekannt. Wähle nun eine anwendbare Regel von M und simuliere ihre Anwendung (durch $(k+1)$-faches Durchlaufen des Bandinhalts und Betrachten der i-ten Spur im i-ten Durchlauf).

5.2.5 Annahme

M braucht T Schritte zur Akzeptanz von x, $T \geqslant |x|$. Wie viele Schritte braucht M'?
Pro-Schrittsimulation benötigt M $\mathcal{O}(T)$ Schritte, also für T Schritte $T \cdot \mathcal{O}(T)$.

5.2.6 Definition

Eine Turing-Maschine ist deterministisch, wenn es in jeder „Situation“ höchstens 1 Rechenschrittregel gibt, die angewandt werden kann.

5.2.7 Satz

NTM-Sprachen = DTM-Sprachen.

Beweis

„$\supseteq$“ Sei L eine DTM-Sprache. Dann ist L eine NTM-Sprache.

„$\subseteq$“ L werde von einer NTM M akzeptiert. Wir suchen eine DTM M' mit $L(M') = L$.
Sei nun W die maximale Wahlvielfalt, also die Anzahl der Regeln, die in einer „Situation“ von der Maschine M angewandt werden können. Nun ist die Idee, dass man „Leitstrings“ $I \in \{1, \ldots, W\}^*$ betrachtet, z.B. sei bei $W = 4 : I = (2, 1, 4, 3)$. Der Leitstring $(i_1, \ldots, i_k)$ soll angeben, dass in der Simulation von M im j-ten Schritt die Wahlmöglichkeit i_j verfolgt werden soll. Man gehe dann wie folgt vor:

- Breche ab, falls es im j-ten Schritt weniger als i_j Wahl möglichkeiten gibt.
- Breche ab, falls nach k Schritten kein Endzustand erreicht wurde.

Die Idee hierbei ist, dass M' alle Strings $I \in \{1, \ldots, W\}^*$ aufzählt und jeden als Leitstring probiert, bis die Eingabe akzeptiert wird. M' hat $b+1$ Arbeitsbänder, davon b Stück für die Simulation von M und 1 für den Leitstring. Die Aktionen von M' sind wie folgt:

```
repeat
    generiere den nächsten Leitstring I auf Band b + 1
    repeat
       Simuliere M mit Leitstring I
    until Simulation zu Endzustand kommt
    mache A-Bänder 1 − b leer, alle Köpfe nach links
```

Kapitel 6

Berechenbarkeit

6.1 Church-Turing-These

Alles, was sich $\overbrace{\text{„berechnen“}}^{\text{intuitiv}}$ lässt, kann von einer $\overbrace{\text{Turing-Maschine berechnet}}^{\text{formal definiert}}$ werden. Was von einer Turing-Maschine nicht berechnet werden kann, ist überhaupt nicht berechenbar.

6.1.1 Von Turing-Maschinen berechnete Funktionen

Die Turing-Maschine bekommt die Eingabe x und produziert mit Erreichen des Endzustandes auf einem designierten Ausgabeband den Ausgabestring y. Eine solche Turing-Maschine definiert eine **partielle** Funktion von $\Sigma^* \to \Gamma^* : x \mapsto y$, falls die Maschine terminiert.

6.1.2 Definition

i. Eine (partielle) Funktion $f : \Sigma^* \to \Gamma^*$ heißt **berechenbar**, wenn es eine Turing-Maschine mit dem Ein-/Ausgabeverhalten $x \mapsto f(x)$ gibt.

ii. Eine partielle Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt **berechenbar**, wenn es eine Turing-Maschine mit dem Ein-/Ausgabeverhalten

$$(x_1)_2 \,\$\, (x_2)_2 \,\$\, \ldots \,\$\, (x_k)_2 \mapsto (f(x_1, \ldots, x_k))_2$$

(über Binärstrings) gibt.

6.1.3 Beispiel

$$f(n) = \begin{cases} 1 & n = \lfloor \pi \cdot 10^k \rfloor \text{ für irgendein } k \\ 0 & \text{sonst} \end{cases}$$

ist berechenbar.

$$g(n) = \begin{cases} 1 & \text{falls die } n\text{-te Stelle in der Dezimaldarstellung von } \pi \text{ eine 7} \\ 0 & \text{sonst} \end{cases}$$

ist berechenbar. Über die Funktion

$$h(n) = \begin{cases} 1 & n = \lfloor \pi \cdot 10^k \rfloor \mod 10^l \text{ für irgendwelche } l, k \\ 0 & \text{sonst} \end{cases}$$

(„kommt die Dezimaldarstellung von n als Teilstring von der Dezimaldarstellung von π vor?“) kann nicht gesagt werden, ob sie berechenbar ist. Hingegen gilt:

$$\varphi(n) = \begin{cases} 1 & \text{in Dezimaldarstellung von } \pi \text{ kommen } n \text{ aufeinanderfolgende 7 vor} \\ 0 & \text{sonst} \end{cases}$$

ist berechenbar, denn entweder ist

$$\varphi(n) = 1 \ \forall n \in \mathbb{N},$$

oder $\exists k \in \mathbb{N}$ für das gilt

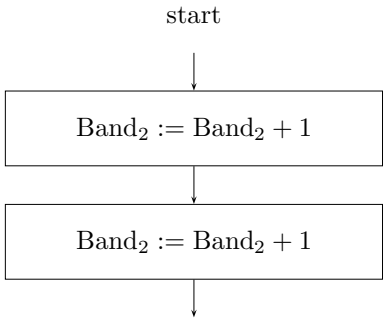
$$\varphi(n) = \begin{cases} 1 & n \leqslant k \\ 0 & \text{sonst} \end{cases}.$$

6.2 Turing-Maschinen-Konstruktion aus Teil-Turing-Maschinen

Der Begriff „Grundposition“ bedeutet hier, dass alle Köpfe „ganz links“ stehen. Die Teilmaschinen beginnen und enden in Grundposition. Wir stellen uns z.B. folgendes vor:

$$\text{Band}_i := \text{Band}_i + 1$$

Die Konkatenation, also Hintereinanderschaltung, von Teilmaschinen in der Form



Die Teilmaschinen haben verschiedene „Endzustände“ $s_{\text{ja}}, s_{\text{nein}}$.

$$\text{Band}_i \stackrel{?}{=} 0$$

6.3 Die Sprachen LOOP, WHILE und GOTO

Wir möchten an folgenden Beispielen die Church-Turing-These veranschaulichen:

6.3.1 Programmiersprache LOOP

Die Programmiersprache LOOP hat

- Variablen $\texttt{x}_0$, $\texttt{x}_1$, $\texttt{x}_2, \ldots$ als Metasymbole
- Konstanten `0`, `1`, `2`, ...
- Trennsymbole `;`, `:=`
- Operationszeichen `+`, $\dot{-}$
- Schlüsselwörter `LOOP`, `DO`, `END`.
- Syntax (induktiv):

  A) $\forall i, j$: $\texttt{x}_i$ `:=` $\texttt{x}_j$ `+ c;` und $\texttt{x}_i$ `:=` $\texttt{x}_j \dot{-}$ `c;` sind LOOP-Programme, wobei `c` eine Konstante ist

  B) $\forall$ LOOP-Programme $P_1, P_2, \forall i$
  - $P_1; P_2$ ist LOOP-Programm
  - `LOOP` $\texttt{x}_i$ `DO` P_1 `END;`

- Semantik (wie erwartet):
  - Ein LOOP-Programm berechnet die Funktion $f : \mathbb{N}^k \to \mathbb{N}$. Dabei sind $\texttt{x}_1, \ldots, \texttt{x}_k$ mit Eingaben $n_1, ..., n_k$ initialisiert. Die restlichen Variablen werden mit 0 initialisiert.
  - Bei Zuweisungen ist zu beachten, dass $\texttt{x}_i \dot{-}$ `c` nicht negativ werden kann, sondern in diesen Fällen 0 ist.
  - P_1`;` P_2 bedeutet, dass zuerst P_1 ausgeführt wird, dann P_2.
  - `LOOP` $\texttt{x}_i$ `DO` P_1 `END;` bedeutet, dass P_1 so oft ausgeführt wird, wie der Wert von $\texttt{x}_i$ vor der ersten Ausführung der Schleife angibt.
  - Das Ergebnis der Berechnung ist der Wert von $\texttt{x}_0$.

6.3.2 Definition (LOOP-Berechenbarkeit)

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt **LOOP-berechenbar**, falls es ein LOOP-Programm P gibt, das f berechnet.

6.3.3 „Syntaktischer Zucker“

	Anweisung	LOOP-Programm

•	`IF` $\texttt{x}_i$ `= 0 THEN` A `END;`	`y := 1;`
		`LOOP` $\texttt{x}_i$ `DO y := 0 END;`
		`LOOP y DO` A `END;`

•	Addition: $\texttt{x}_0$ `:=` $\texttt{x}_1$ `+` $\texttt{x}_2$	$\texttt{x}_0 := \texttt{x}_1$`;` `LOOP` $\texttt{x}_2$ `DO` $\texttt{x}_0$ `:=` $\texttt{x}_0$ `+ 1 END;`
•	Multiplikation, mod/div	analog

6.3.4 Programmiersprache WHILE

Die Syntax und Semantik dieser Sprache ist wie bei LOOP. Zusätzlich führt man die Anweisung `WHILE` $\mathtt{x}_i \neq \mathtt{0}$ `DO P END` ein, deren Semantik wie folgt ist:
Teste $\mathtt{x}_i$. Falls der Wert von $\mathtt{x}_i \neq \mathtt{0}$, führe P aus. Wiederhole solange, bis $\mathtt{x}_i$ bei einem Test `= 0`.
Auch eine `LOOP`-Schleife ist durch eine `WHILE`-Schleife simulierbar.

6.3.5 Definition (WHILE-Berechenbarkeit)

Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heißt **WHILE-berechenbar**, falls es ein WHILE-Programm P gibt, das f berechnet. Falls $f(x)$ undefiniert, dann hält P nicht auf x.

6.3.6 Satz

Turing-Maschinen können WHILE-Programme simulieren.

Gilt aber auch die Umkehrung, also dass WHILE-Programme Turing-Maschinen simulieren können?
Als Zwischenschritt nehmen wir ein Programm der

6.3.7 Programmiersprache GOTO

Sie sei spezifiziert durch Anweisungsfolgen folgender Form:

Marke	Anweisung
M_1	A_1
M_2	A_2
$\vdots$	$\vdots$
M_k	A_k

- Wertzuweisung: $\mathtt{x}_i$ `:=` $\mathtt{x}_j \mathbin{\dot{\pm}} \mathtt{c}$
- unbedingter Sprung: `GOTO` $\mathtt{M}_i$
- bedingter Sprung: `IF` $\mathtt{x}_i$ `= c THEN GOTO` $\mathtt{M}_i$
- Stop-Anweisung: `HALT`
  konkreter: Letzte Anweisung ist immer `HALT`

Man kann auf `WHILE`-Schleifen simulieren: Anweisungen der Form

`WHILE` $\mathtt{x}_i \neq$ `0 DO P END;`

schreibt man wie folgt:

```
M1:  IF xi = 0 THEN GOTO M2
     P;
     GOTO M1
M2:  ...
```

6.3.8 Satz

Jedes WHILE-Programm kann durch ein GOTO-Programm simuliert werden.

Kann man auch ein GOTO-Programm durch ein WHILE-Programm ersetzen? Offensichtlich kann man Fälle mit verschränkten Sprüngen nicht kanonisch durch Schleifen simulieren. Trotzdem ist diese Umkehrung aber möglich:

Annahme

Sei ohne Beschränkung der Allgemeinheit `count` eine beliebige Variable $\mathtt{x}_i$. Man habe eine Anweisungsfolge der Art

$$\mathtt{M}_1\mathtt{:}\ A_1\mathtt{;}\ \mathtt{M}_2\mathtt{:}\ A_2\mathtt{;}\ \ldots\mathtt{;}\ \mathtt{M}_k\mathtt{:}\ A_k.$$

Man kann diese wie folgt durch ein WHILE-Programm simulieren:

```
count := 1;
WHILE count ≠ 0 DO
IF count = 1 THEN A'_1 END
IF count = 2 THEN A'_2 END
...
IF count = k THEN A'_k END
END;
```

Jedes A'_i sei nun durch die folgende Befehlssequenz spezifiziert:

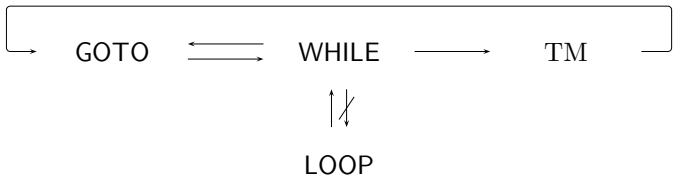
- Falls $A'_i =$ „$\mathtt{x}_j$:= $\mathtt{x}_l \mathbin{\dot{\pm}} \mathtt{c}$“:
  $\mathtt{x}_j$:= $\mathtt{x}_l \mathbin{\dot{\pm}} \mathtt{c}$
  `count := count + 1;`
- Falls $A'_i =$ „`GOTO` $\mathtt{M}_j$“:
  `count :=` j`;`
- Falls $A_i =$ „`IF` $\mathtt{x}_j$ `= c THEN GOTO` $\mathtt{M}_n$`; END;`“:
  `IF` $\mathtt{x}_j$ `= c THEN count = n; ELSE count := count + 1; END;`
- Falls $A_i =$ „`HALT;`“:
  `count := 0;`

6.3.9 Satz

Jedes GOTO-Programm kann durch ein WHILE-Programm simuliert werden.

6.3.10 Satz (Kleensche Normalform)

Jede WHILE-berechenbare Funktion f kann durch ein WHILE-Programm berechnet werden, das nur eine `WHILE`-Schleife hat plus `LOOP`-Schleifen. Diese Form bezeichnet man als **Kleensche Normalform**.



6.3.11 Simulation einer Turing-Maschine durch ein GOTO-Programm

Sei $M = (Q, \Sigma, \Gamma, \Delta, s, \#, F)$ eine Turing-Maschine. Dann kann man ein GOTO-Programm $\texttt{M}_1\texttt{:}\ P_1\texttt{;}\ \texttt{M}_2\texttt{:}\ P_2\texttt{;}\ \texttt{M}_3\texttt{:}\ P_3\texttt{;}\ \ldots$ angeben, dass die Turing-Maschine simuliert:

Seien

$$\begin{aligned} Q &= \{q_1, \ldots, q_l\} \\ \Gamma &= \{a_1, \ldots, a_m\} \\ b &> |\Gamma| \end{aligned}$$

Eine Turing-Maschinen-Konfiguration sieht wie folgt aus:

$$a_{i_1} \ldots a_{i_p} q_l a_{j_1} \ldots a_{j_q}$$

mit

$$\begin{aligned} x = (i_1, \ldots, i_p)_b &= \sum_{\mu=1}^{p} i_\mu b^{p-\mu} \\ y &= (j_q, \ldots, j_1)_b \\ x\ \texttt{MOD}\ b &= i_p \end{aligned}$$

Das Programm P_2 sieht wie folgt aus:

```
M2:  a := y MOD b      („j1“)
     IF (z = 1) AND (a = 1) THEN GOTO M11
     IF (z = 1) AND (a = 2) THEN GOTO M12
     ⋮
```

$((q_1, a_1), (q_{i'}, a_{j'}, L)) \in \Delta$:

```
M11:  z := i'
      y := y DIV b
      y := b * y + j'
      y := b * y + (x MOD b)
      x := x DIV b
      GOTO M2
```

Ist $((q_1, a_1), ?) \in \Delta$ undefiniert, dann `GOTO M3`.

6.3.12 Satz

Jede Turing-Maschine kann durch ein GOTO-Programm simuliert werden.

6.3.13 Konventionen

Konstanten werden mit $c \in \mathbb{N}$ bezeichnet, Eingaben mit $x_1, \ldots, x_r$ und Ausgaben mit x_0. Programme kann man also als Funktionen $\mathbb{N}^r \to \mathbb{N}$ auffassen.

6.3.14 Berechenbarkeit von WHILE, LOOP und GOTO-Programmen

Es liegt auf der Hand, dass man durch WHILE-Programme LOOP-Programme beschreiben kann. Auch GOTO-Programme kann man durch geeignete Umformungen in WHILE-Programme übersetzen. Es gilt aber auch, dass Turingberechenbar $\approx$ WHILE-berechenbar, analog also auch für die anderen beiden Sprachen.

6.4 Primitiv-rekursive und μ-rekursive Funktionen

Im Folgenden führen wir einen mathematischen Formalismus ein, um bestimmte Funktionen $\mathbb{N}^r \to \mathbb{N}$ zu berechnen.

6.4.1 Definition (Primitiv-rekursive Funktionen)

Primitiv-rekursive Funktionen umfassen

i. alle konstanten Funktionen (vom Typ $\mathbb{N}^r \to \mathbb{N}$),

ii. alle Projektionen $\pi_3^4 (x_1, x_2, x_3, x_4) = x_3$,

iii. Nachfolgefunktion $s(n) = n + 1$,

iv. Funktionen, die sich durch Komposition aus primitiv-rekursiven Funktionen ergeben und

v. jede Funktion, die sich durch Anwendung „primitiver Rekursion“ aus einer primitiv-rekursiven Funktion erzeugen lässt:
Seien g und h primitiv-rekursive Funktionen, gegeben durch

$$g: \mathbb{N}^{k-1} \to \mathbb{N}, \quad h: \mathbb{N}^l \to \mathbb{N}, \quad l \leqslant k.$$

Dann ist f wie folgt definiert:

$$f: \mathbb{N}^k \to \mathbb{N} \qquad \begin{aligned} f(0, x_2, \ldots, x_k) &= g(x_2, \ldots, x_k) \\ f(n+1, x_2, \ldots, x_k) &= h(f(n, x_2, \ldots, x_k), x_2, \ldots, x_l) \end{aligned}$$

Beispiele

i. $add: \mathbb{N}^2 \to \mathbb{N}$

$$\begin{aligned} add(0, x) &= x \\ add(n+1, x) &= s(add(n, x)) \end{aligned}$$

ii. $mult : \mathbb{N}^2 \to \mathbb{N}$

$$mult\,(0, x) = 0$$
$$mult\,(n+1, x) = add\,(mult\,(n, x), x)$$

iii. Vorgänger-Funktion: $u : \mathbb{N} \to \mathbb{N}, x \to x \dot{-} 1 := max\,(x - 1, 0)$:

$$u\,(0) = 0$$
$$u\,(n+1) = \pi_2^2(u(n), n)$$

iv. $sub : \mathbb{N}^2 \to \mathbb{N}$

$$sub\,(0, x) = x$$
$$sub\,(n+1, x) = u\,(sub\,(n, x))$$

v. $bin_2 : \mathbb{N} \to \mathbb{N}$, $bin_2\,(n) := \binom{n}{2}$. Dann sei $bin_2 = \binom{n+1}{2} = \binom{n}{2} + n$.

$$bin_2\,(0) = 0$$
$$bin_2\,(n+1) = add\,(bin_2\,(n), n)$$

6.4.2 Satz

F ist genau dann primitiv-rekursiv, wenn F LOOP-berechenbar ist.

Beweis

„⇒“ Übung

„⇐“ F sei LOOP-berechenbar, d.h. es existiert ein LOOP-Programm P, das F $(\mathbb{N}^r \to \mathbb{N})$ berechnet.

Idee

P verwendet Variablen $x_0, \ldots, x_k$. Kodiere $x_0, \ldots, x_k$ als eine einzelne natürliche Zahl $\langle x_0, \ldots, x_k \rangle \in \mathbb{N}$ und realisiere einzelne Programmschritte als $f : \mathbb{N} \to \mathbb{N}$. Dabei ist $\langle \quad \rangle : \mathbb{N}^{k+1} \to \mathbb{N}$ die Kodierung mit $0 \leqslant i \leqslant k$. $d_i : \mathbb{N} \to \mathbb{N}$ „dekodiert“ i-te Komponente.

$$d_i\,(\langle n_0, \ldots, n_k \rangle) = n_i$$

Sei nun P ein Programm, $a_0, \ldots, a_k$ der Variableninhalt vor der Ausführung von P und $b_0, \ldots, b_k$ der Variableninhalt danach. Wir wollen zeigen, dass $f_P : f_P\,(\langle a_0, \ldots, a_k \rangle) = \langle b_0, \ldots, b_k \rangle$. Dann ist:

$$F\,(x_1, \ldots, x_r) = d_0\,(f_P\,(\langle 0, x_1, \ldots, x_r, 0, \ldots, 0 \rangle))$$

Man spezifiziere f_P über strukturelle Induktion.

- Hat P die Form $\mathtt{x}_i$ `:=` $\mathtt{x}_j \dot{+} \mathtt{c}$, so sieht $f_P\,(n)$ wie folgt aus:

$$f_P\,(n) = \left\langle d_0\,(n), d_1\,(n), \ldots, d_{i-1}\,(n), \begin{matrix} add\,(d_j\,(n), c) \\ sub\,(c, d_j\,(n)) \end{matrix}, d_{i+1}\,(n), \ldots, d_k\,(n) \right\rangle$$

– Hat P die Form R; Q, dann:

$$f_P(n) = f_Q(f_R(n))$$

– Es bleibt die Möglichkeit, dass P in der Form `LOOP` x_i `DO` Q `END` vorliegt. Dann definiere man
$h_Q(n, x)$ als Funktion „f_Q wird n-mal auf x angewendet“, also:

$$\begin{aligned} h_Q(0, x) &= x \\ h_Q(n+1, x) &= f_Q(h_Q(n, x)) \end{aligned}$$

$$f_P(x) = h_Q(d_i(x), x)$$

6.4.3 Definition (μ-rekursive Funktionen)

μ-rekursive Funktionen sind alle primitiv-rekursiven Funktionen plus alle jene, die sich durch Anwendung des μ-Operators auf μ-rekursive Funktionen erzeugen lassen.

6.4.4 Definition (μ-Operator)

Gegeben sei eine Funktion

$$\begin{aligned} f &: \mathbb{N}^{k+1} \to \mathbb{N}, \\ \mu f &: \mathbb{N}^k \quad \to \mathbb{N} \end{aligned}$$

mit

$$\mu f(x_1, \ldots, x_k) = min\{n \in \mathbb{N} | f(n, x_1, \ldots, x_k) = 0 \text{ und für } 0 \leqslant m < n \text{ ist } f(m, x_1, \ldots, x_k) \text{ definiert}\}$$

Dann nennt man den Faktor μ auch **μ-Operator**. Nach obiger Definition ist f μ-rekursiv.

Beispiel

$$\begin{aligned} h(x, y) &= 7 \\ \mu h(y) &= \text{undefiniert } \forall y \in \mathbb{N} \end{aligned}$$

6.4.5 Satz

f ist genau dann μ-rekursiv, wenn f WHILE-berechenbar ist.

Beweis

„$\Rightarrow$“ -

„$\Leftarrow$“ Wir bauen auf dem vorherigen Beweis auf und erweitern diesen. P hat nun die Form `WHILE` $x_i \neq$`0 DO` Q `END`.
Man muss nun f_Q so oft anwenden, bis $x_i = 0$. Somit ist

$$\mu(d_i h_Q)(x) = min\{n | d_i(h_Q(n, x)) = 0\}.$$

Für $f_P(x)$ erhält man $f_P(x) = h_Q(\mu(d_i h_Q)(x), x)$.
Die Kodierung bzw. Dekodierung wird im Buch von Schöning, Seiten 111-112 behandelt.

6.4.6 Definition

Eine Funktion $f : \Sigma^* \to \Gamma^*$ heißt **(Turing-)berechenbar**, wenn es eine Turing-Maschine M gibt, die für jedes $x \in \Sigma^*$ als Eingabe mit Ausgabe $f(x)$ hält.

Sowohl Eingabe als auch Ausgabe müssen hier in bestimmten Codierungen $cod(\ldots)$ vorliegen, die an dieser Stelle aber ohne Beschränkung der Allgemeinheit nicht betrachtet werden sollen.

6.4.7 Definition

Sei $A \subset \Sigma^*$.

- A heißt **entscheidbar**, wenn die charakteristische Funktion χ_A berechenbar ist:
$$\chi_A(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A. \end{cases}$$
- A heißt **semi-entscheidbar**, wenn die positiv charakteristische Funktion χ_A^+ berechenbar ist:
$$\chi_A^+(x) = \begin{cases} 1 & x \in A \\ \text{undefiniert} & x \notin A, \end{cases}$$
es gibt also eine Turing-Maschine, die bei Eingabe über $x \in A$ hält und 1 ausgibt und dies bei Eingabe $x \notin A$ dies nicht tut (sie hält also nicht oder gibt nicht 1 aus).

6.4.8 Beobachtung

Genau dann, wenn A semi-entscheidbar, ist A Turing-akzeptierbar.

6.4.9 Satz

Eine Menge A ist genau dann entscheidbar, wenn sie selbst und auch ihr Komplement $\overline{A}$ jeweils semi-entscheidbar sind.

Beweis

„⇒“ Diese Richtung ist trivial.

„⇐“ Seien M^+ eine Turing-Maschine für χ_A^+ und M^- eine Turing-Maschine für $\chi_{\overline{A}}^+$. Wir suchen eine Turing-Maschine M für χ_A.
M lässt M^+ und M^- parallel für die Eingabe x laufen. Akzeptiert M^+ die Eingabe, dann hält die Maschine mit Ausgabe 1, analog hält sie mit 0, wenn M^- die Eingabe akzeptiert.

6.4.10 Korollar

Genau dann, wenn A nicht entscheidbar ist, ist A nicht Turing-akzeptierbar oder $\overline{A}$ ist nicht Turing-akzeptierbar.

6.4.11 Definition

$A \subset \Sigma^*$ heißt **rekursiv aufzählbar**, falls $A = \emptyset$ ist oder es eine überall definierte, berechenbare Funktion $F : \mathbb{N} \to \Sigma^*$mit

$$A = \{F(0), F(1), \ldots\} = \{F(n) \,|n \in \mathbb{N}\}$$

gibt. Hierbei sind Fälle der Art $F(i) = F(j)$ durchaus erlaubt. Man sagt, F *zählt* A *auf.*

6.4.12 Satz

A ist genau dann rekursiv aufzählbar, wenn A semi-entscheidbar ist.

Beweis

„⇒“ A werde durch F aufgezählt. Wir suchen eine Turing-Maschine M, die A akzeptiert. M verarbeite die Eingabe $x \in \Sigma^*$ und sei wie folgt aufgebaut:

```
n = 0
while F( n ) ≠ x do n = n + 1
halte mit Ausgabe 1
```

„⇐“ Wir nehmen an, dass eine Turing-Maschine M existiert, die A akzeptiert. Wir suchen eine Turing-Maschine N, die die Funktion F berechnet, die A aufzählt. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass $a \in A \neq \emptyset$. Sei G eine bijektive Funktion $\mathbb{N} \to \Sigma^* \times \mathbb{N}$. Diese Funktion existiert und ist auch berechenbar (vergleiche 1.9.5). N akzeptiere die Eingabe $n \in \mathbb{N}$ und sei wie folgt beschrieben:

```
Berechne G(n) = (x,t) ∈ Σ* × ℕ.
Lasse M mit Eingabe x für t Schritte laufen.
Akzeptiert M, halte mit Ausgabe x, sonst mit Ausgabe a.
```

6.4.13 Anmerkung

Die folgenden Ausagen sind äquivalent:

- A ist Turing-akzeptierbar.
- A ist semi-entscheidbar.
- A wird von einer Typ0-Grammatik generiert.
- χ_A^+ ist WHILE-berechenbar (μ-rekursiv).
- A ist rekursiv-aufzählbar.
- A ist Definitionsbereich einer berechenbaren Funktion (zum Beispiel von χ_A^+).
- A ist Wertebereich einer berechenbaren Funktion (zum Beispiel von χ_A^+).

6.5 Nichtentscheidbare Sprachen

An diesem Punkt stellt sich die Frage, ob es *interessante* Sprachen gibt, die nicht entscheidbar sind. Anders ausgedrückt: Gibt es Probleme, die mit dem Computer prinzipiell nicht lösbar sind?

6.5.1 Codierung von Turing-Maschinen und Worten

Codieren

- Turing-Maschinen:
  Sei $M = (\Sigma, Q, \Gamma, s, F, \mathfrak{b}, \Delta)$, die oBdA nur ein Band besitzt und durch $Q = \{q_0, \ldots, q_m\}, \Sigma = \{a_1, \ldots, a_s\}, \Gamma = \{a_1, \ldots, a_g\} (g \geqslant s), s = q_0, F = \{q_1\}, \Delta = \{\delta_1, \ldots, \delta_D\}$ spezifiziert ist. Dann ist
  $$cod(M) = cod(\delta_1)\, cod(\delta_2) \ldots cod(\delta_D)$$
  und
  $$\delta = (q_i, a_j, a_k, \mu_h, q_l).$$
  Hierbei ist $\mu_1 = L, \mu_2 = B, \mu_3 = R$ für $\Delta \subset Q \times \Gamma \times \Gamma \times \{L, B, R\} \times Q$. Insbesondere gilt $cod(\delta) = 01^i01^j01^k01^h01^l0$.
  Man beachte, dass $cod(M)$ nicht eindeutig ist, bedingt durch die Reihenfolge der δ.
- Worte:
  Sei $\Sigma = \{a_1, \ldots, a_s\}$. Dann gilt für $x \in \Sigma^*,\ |x| = n$:
  $$x = a_{i_1} a_{i_2} a_{i_3} \ldots a_{i_n},$$
  womit für die Codierung
  $$cod(x) = 0^{i_1}1^{i_2}0^{i_3} \ldots,$$
  $$cod(\varepsilon) = \varepsilon$$
  gilt.

Decodieren

- Turing-Maschinen:
  Sei $w \in \{0,1\}^*$. Dann definiert man
  $$M_w = \begin{cases} M \text{ sodass } cod(M) = w & \text{falls } w \text{ eine legale Codierung ist} \\ \text{eine Turing-Maschine, die nie hält} & \text{sonst} \end{cases}$$
  M_w interpretiert w als Turing-Maschine, beziehungsweise als deren Programm.
- Worte:
  Man erhält als Dekodierung $decod(w) = a_{i_1} a_{i_2} a_{i_3} \ldots$ für sowohl $w = 0^{i_1}1^{i_2}0^{i_3} \ldots$, als auch für $w = 1^{i_1}0^{i_2}1^{i_3} \ldots$. Der Grund dafür liegt in der Tatsache, dass man beliebig viele ε bei der Codierung an den Anfang setzen kann, diese aber als $\varepsilon = 0^0 = 1^0$ in die Codierung eingehen.

Man beachte, dass gilt:

$$M_{cod(M)} = M, \text{ aber } decod(cod(x)) \approx x.$$

6.5.2 Die Sprache UNIV

Man betrachte folgende „Universalsprache“:

$$\mathsf{UNIV} = \left\{ w\$v \in \{0,1\}^* \,\$\, \{0,1\}^* \,|M_w \text{ akzeptiert } decod\,(v) \right\}$$

Die Frage, ob die Turing-Maschine M die Eingabe x akzeptiert, wird dadurch beantwortet, ob $cod\,(M)\,\$\; cod\,(x) \in \mathsf{UNIV}$.

6.5.3 Entscheidbarkeit von UNIV

Es wäre nun gut zu wissen, ob UNIV entscheidbar ist, denn in diesem Fall könnte man testen, ob eine gegebene Turing-Maschine (Programm) M auf einer gegebenen Eingabe x hält oder nicht. Die Antwort auf diese Frage lautet jedoch nein.

6.5.4 Satz

UNIV ist semi-entscheidbar.

Beweis

Idee

Man baue eine Turing-Maschine U (universelle Maschine), die als Eingabe ein „Programm“ $cod(M)$ und eine Eingabe für das Programm $cod(x)$ bekommt, und dann M mit Eingabe x emuliert.

6.5.5 Satz

$\overline{\mathsf{UNIV}}$ ist nicht semi-entscheidbar, also nicht Turing-akzeptierbar.

Für den Beweis müssen wir den Umweg über eine weitere Sprache, SAM gehen. Zuvor notieren wir aber noch ein

6.5.6 Korollar

UNIV ist nicht entscheidbar. Das folgt unmittelbar aus 6.5.4 und 6.5.5.

6.6 Die Sprache SAM

Die Sprache SAM beschreibt die selbst akzeptierenden Maschinen.

$$\mathsf{SAM} = \left\{ w \in \{0,1\}^* \middle|\, w \in L\,(M_w) \right\}$$

6.6.1 Behauptung

SAM ist Turing-akzeptierbar.

Beweis

$$w \in \mathsf{SAM} \Leftrightarrow w\$\; cod(w) \in \mathsf{UNIV}$$

6.6.2 Behauptung

$\overline{\mathsf{SAM}} = \{w \in \{0,1\}^* \,|w \notin L(M_w)\}$ ist nicht Turing-akzeptierbar.

Beweis

Wir müssen zeigen, dass es keine Turing-Maschine M mit $L(M) = \overline{\mathsf{SAM}}$ gibt. Anders ausgedrückt bedeutet dies, dass für alle Turing-Maschinen $M : L(M) \neq \overline{\mathsf{SAM}}$. Daher muss für alle $w \in \{0,1\}^* : L(M_w) \neq \overline{\mathsf{SAM}}$. Das ist in der Tat der Fall, denn in w unterscheiden sich $\overline{\mathsf{SAM}}$ und $L(M_w)$:

$$w \in L(M_w) \Rightarrow w \notin \overline{\mathsf{SAM}} \qquad \text{und} \qquad w \notin L(M_w) \Rightarrow w \in \overline{\mathsf{SAM}}.$$

6.6.3 Korollar

SAM ist nicht entscheidbar.

6.6.4 Korollar

UNIV ist nicht entscheidbar.

Beweis

Wir nehmen an, es gebe eine Turing-Maschine X, die UNIV entscheidet und wollen zeigen, dass es dann eine Turing-Maschine Y gibt, die SAM entscheidet. Y verarbeitet die Eingabe $w \in \{0,1\}^*$ und ist wie folgt aufgebaut:

`schreibe zunächst` $\$cod(w)$ `hinter die Eingabe und mache den Bandinhalt somit zu` $w\$cod(w)$
`lasse` X `mit der Eingabe` $w\$cod(w)$ `laufen`

$decod(cod(w))$ ist aber gerade w und so antwortet X mit JA, wenn $w \in L(M_w)$ und antwortet mit NEIN, wenn $w \notin L(M_w)$. Y antwortet aber genauso, d.h. Y entscheidet SAM. $\lightning$

Nun stellt sich die Frage, inwieweit man testen kann, ob eine Turing-Maschine den leeren String akzeptiert oder nicht. Anders ausgedrückt, ist

$$\mathsf{LEER\text{-}STRING} = \{w \in \{0,1\}^* \,|\varepsilon \in L(M_w)\}$$

entscheidbar?

6.6.5 Behauptung

LEER-STRING ist nicht entscheidbar.

Beweis

Wir nehmen an, dass LEER-STRING von einer Turing-Maschine X entscheidbar ist und wollen zeigen, dass es dann eine Turing-Maschine Y gibt, die UNIV entscheidet, was unmittelbar einen Widerspruch darstellt. Die Turing-Maschine Y, die die Eingabe $w\$\ x$ verarbeitet, sieht wie folgt aus:

```
baue eine Metamaschine M, die die Eingabe z verarbeitet und wie folgt
aufgebaut ist:
    ersetze z durch x
    lasse M_w auf der Eingabe decod(x) laufen
lasse X auf cod(M) laufen
```

Es gilt:

$$M \text{ akzeptiert } \varepsilon \Leftrightarrow M_w \text{ akzeptiert } decod\,(x)\,,$$

wobei die Akzeptanz von ε durch M_w von X getestet wird und die Akzeptanz von $decod\,(x)$ durch M_w bedeutet, dass $w\$\ x \in \mathsf{UNIV}$. Hieraus folgt also unmittelbar, dass UNIV von Y entschieden wird.

6.7 Reduktionen

6.7.1 Definition

Seien $A \subset \Sigma^*, B \subset \Gamma^*$. Man sagt, A ist auf B **reduzierbar** (oder A **ist leichter als** B), geschrieben $A \preccurlyeq B$, wenn es eine überall berechenbare Funktion $f : \Sigma^* \to \Gamma^*$ gibt, sodass

$$\forall a \in \Sigma^* : a \in A \Leftrightarrow f(a) \in B.$$

6.7.2 Lemma

- Sei $A \preccurlyeq B$ und B entscheidbar, dann ist auch A entscheidbar.
- Sei $A \preccurlyeq B$ und A nicht entscheidbar, dann ist auch B nicht entscheidbar.

Beweis

- $A \preccurlyeq B$, das heißt $\exists$ Turing-Maschine M, die die Funktion $f : \Sigma^* \to \Gamma^*$ berechnet, wobei $\forall x \in \Sigma^* : x \in A \Leftrightarrow f\,(x) \in B$.
- B ist entscheidbar, also $\exists$ Turing-Maschine X, die bei der Eingabe $y \in \Gamma^*$ mit ja oder nein antwortet, je nach dem, ob $y \in B$ oder nicht.

Die Turing-Maschine Y, die die Eingabe $x \in \Sigma^*$ verarbeitet, sieht dann wie folgt aus:

```
berechne y = f(x) mit Hilfe von M
teste durch Aufruf von X, ob y ∈ B, somit also ob f(x) ∈ B
```

Damit wird A von Y entschieden.

6.7.3 Beispiel

$\mathsf{SAM} \preccurlyeq \mathsf{UNIV}$. Das wird deutlich, wenn man die Reduktionsfunktion $f\,(x) = w\$\ cod\,(w)$ betrachtet. Es gilt nämlich

$$w \in L\,(M_w) \Leftrightarrow w\$\ cod\,(w) \in \mathsf{UNIV} \Leftrightarrow decod\,(cod\,(w)) \in L\,(M_w)\,.$$

6.7.4 Behauptung

$$\mathsf{SAM} \preccurlyeq \mathsf{UNIV}.$$

Hierzu betrachten wir $F : w \mapsto w\$cod\,(w)$. Weiterhin sieht man aber: Da $w \in \mathsf{SAM} \Leftrightarrow w\$cod\,(w) \in \mathsf{UNIV}$ gilt, wenn SAM nicht entscheidbar, dann ist auch UNIV **nicht** entscheidbar.

6.7.5 Behauptung

Für jedes u gilt, dass

$$\mathsf{ACC}_u = \left\{ w \in \{0,1\}^* \middle|\, u \in L\,(M_w) \right\}, \quad u \in \Sigma^*,$$

nicht entscheidbar ist.

Beweis

Es genügt zu zeigen, dass $\mathsf{UNIV} \preccurlyeq \mathsf{ACC}_u$. Wir brauchen die Instanz $F : w\$x$ für UNIV. Diese produziert einen String Z, welcher eine Beschreibung einer Turing-Maschine ist, sodass $w\$x \in \mathsf{UNIV} \Leftrightarrow Z$ akzeptiert u. Die durch Z vorgegebene Turing-Maschine ist die folgende:
Bei der Eingabe y arbeitet die Maschine wie folgt:

`lasse` M_w `mit der Eingabe` $decod\,(x)$ `laufen`
`if` y `=` u `then akzeptiere else divergiere`

$$F : w\$x \mapsto cod(Z)$$

Z verhält sich folgendermaßen:

- Wenn $x \notin L(M_w)$ dann $L(Z) = \emptyset, u \notin L(Z)$
- Wenn $x \in L(M_w)$ dann $u \in L(Z)$

Das bedeutet für die simulierte Maschine:

- Wenn $w\$x \notin \mathsf{UNIV}$ dann $L(Z) = \emptyset, cod(Z) \notin \mathsf{ACC}_u$
- Wenn $w\$x \in \mathsf{UNIV}$ dann $cod(Z) \in \mathsf{ACC}_u$

6.8 Satz von Rice

Seien $\mathcal{RE}$ die Familie aller rekursiv aufzählbaren Sprachen und $\emptyset \neq \mathcal{S} \subsetneq \mathcal{RE}$. Dann gilt für jedes dieser $\mathcal{S}$, dass die Sprache

$$C(\mathcal{S}) = \{w | L(M_w) \in \mathcal{S}\}$$

nicht entscheidbar ist. Intuitiver ausgedrückt:
Alle Eigenschaften von Turing-Maschinen (Programmen), die sich durch die **akzeptierte Sprache** ausdrücken lassen, sind nicht entscheidbar.

6.8.1 Beispiele

- $\{w | u \in L(M_w)\} = \mathsf{ACC}_u$
- $\{w | L(M_w)$ ist endlich$\}$
- $\{w | L(M_w)$ ist regulär$\}$
- ...

6.8.2 Ausnahmen

Von der durch den Satz von Rice begründeten Aussage sind explizit triviale Eigenschaften ausgenommen, die von allen (oder keiner) Sprache besessen werden.

6.8.3 Beispiele

- $\{w | L(M_w)$ ist rekursiv aufzählbar$\}$
- $\{w | L(M_w)$ ist nicht rekursiv aufzählbar$\} = \emptyset$

Im ersten Fall ist $\mathcal{S} = \mathcal{RE}$, im zweiten $\mathcal{S} = \emptyset$. Diese Sonderfälle sind in der formalen Spezifikation des Satzes aber gerade auch ausgeschlossen.

Beweis

Sei $\mathcal{S} \subseteq \mathcal{RE}$ und o.B.d.A. $\emptyset \notin \mathcal{S}$ (in diesem Fall betrachte $\overline{\mathcal{S}}$). Ferner sei $L \in \mathcal{S}$ und es existiere eine Turing-Maschine M_L, die L akzeptiert. Wir wollen zeigen, dass

$$\mathsf{UNIV} \preccurlyeq C(\mathcal{S}).$$

Wir brauchen dazu eine Funktion

$$F : w\$x \mapsto cod(ZZ),$$

die wie folgt arbeitet:

$$\begin{aligned} w\$x \in \mathsf{UNIV} &\Leftrightarrow decod(x) \in L(M_w) \\ &\Leftrightarrow cod(ZZ) \in C(\mathcal{S}) \\ &\Leftrightarrow L(ZZ) \in \mathcal{S} \end{aligned}$$

Hierbei verarbeitet ZZ die Eingabe y und ist wie folgt aufgebaut:

`lasse` M_w `mit der Eingabe` $decod(x)$ `laufen`
`lasse` M_L `mit der Eingabe` y `laufen`

Die Maschine ZZ verhält sich wie folgt:

- Ist $w\$x \in \mathsf{UNIV}$, dann akzeptiert M_w $decod(x)$, womit $L(ZZ) = L(M_L) = L \in \mathcal{S}$ (ZZ verhält sich also wie M_L).
- Ist $w\$x \notin \mathsf{UNIV}$, dann akzeptiert M_w $decod(x)$ nicht, womit $L(ZZ) = \emptyset \notin \mathcal{S}$ (ZZ akzeptiert keine Eingabe).

6.9 Post-sches Korrespondenzproblem

6.9.1 Post-Spiel

Man habe k Typen von Karten, $\boxed{\begin{array}{|c|}\hline x \\ \hline y \\ \hline \end{array}}$, $x, y \in \{0,1\}^+$. So ist z.B. für $k = 3$:

$$\underset{\text{I}}{\boxed{\begin{array}{|c|}\hline 1 \\ \hline 101 \\ \hline \end{array}}} \quad , \quad \underset{\text{II}}{\boxed{\begin{array}{|c|}\hline 10 \\ \hline 00 \\ \hline \end{array}}} \quad , \quad \underset{\text{III}}{\boxed{\begin{array}{|c|}\hline 011 \\ \hline 11 \\ \hline \end{array}}} \quad .$$

Von jedem Typ habe man beliebig viele Karten.

Vorgehensweise

Lege die Karten nebeneinander, sodass sich oben und unten der gleiche String ergibt. In unserem Beispiel sieht das wie folgt aus:

$$\underset{\text{I}}{\boxed{\begin{array}{|c|}\hline 1 \\ \hline 101 \\ \hline \end{array}}} \quad \underset{\text{III}}{\boxed{\begin{array}{|c|}\hline 011 \\ \hline 11 \\ \hline \end{array}}} \quad \underset{\text{II}}{\boxed{\begin{array}{|c|}\hline 10 \\ \hline 00 \\ \hline \end{array}}} \quad \underset{\text{III}}{\boxed{\begin{array}{|c|}\hline 011 \\ \hline 11 \\ \hline \end{array}}}$$

Für $k = 4$ kann man sich folgendes Beispiel ausdenken:

$$\underset{\text{I}}{\boxed{\begin{array}{|c|}\hline 001 \\ \hline 0 \\ \hline \end{array}}} \quad \underset{\text{II}}{\boxed{\begin{array}{|c|}\hline 01 \\ \hline 011 \\ \hline \end{array}}} \quad \underset{\text{III}}{\boxed{\begin{array}{|c|}\hline 01 \\ \hline 101 \\ \hline \end{array}}} \quad \underset{\text{IV}}{\boxed{\begin{array}{|c|}\hline 10 \\ \hline 001 \\ \hline \end{array}}}$$

Die kürzeste Lösung kann hier mit 66 Karten erreicht werden.

Ziel

Absicht dieses Modells ist die Überlegung, dass man entscheiden können möchte, ob es eine Lösung gibt oder nicht.

6.9.2 Post-sches Korrespondenzproblem (PCP_Σ)

Sei Σ das Alphabet. Gegeben ist $k \geqslant 1$ und $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$ mit $x_i, y_i \in \Sigma^+$

Problem

Es stellt sich die Frage, ob $\exists I \in \{1, \ldots, k\}^+ : X[I] = Y[I]$.

Ansatz für die Lösung

$I = (i_1, i_2, \ldots, i_n)$:

$$X[I] = x_{i_1} x_{i_2} \ldots x_{i_n}$$
$$Y[I] = y_{i_1} y_{i_2} \ldots y_{i_n}$$

Wir können PCP_Σ nun als Sprache wie folgt definieren:

$$\mathsf{PCP}_\Sigma = \Big\{ (k, (x_1, y_1), \ldots, (x_k, y_k)) \in \mathbb{N} \times \left(\left(\Sigma^+\right)^2 \right)^k \Big| \exists I \in \{1, \ldots, k\}^+ : X[I] = Y[I] \Big\}$$

6.9.3 Behauptung

PCP_Σ ist **nicht** entscheidbar, aber akzeptierbar.

Anmerkung

Um PCP_Σ zu akzeptieren, baut man eine NTM, die I errät und verifiziert.

6.9.4 Das modifizierte Post-sche Korrespondenzproblem (MPCP_Σ)

Hierbei handelt es sich um eine Modifikation zu PCP_Σ.

$$\mathsf{MPCP}_\Sigma = \Big\{ (k, ((x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k))) \Big| \exists I = (i_1, \ldots, i_n) \in \{1, \ldots, k\}^+ : X[I] = Y[I] \text{ und } i_1 = 1 \Big\}$$

6.9.5 Lemma

Seien $\$, \# \notin \Sigma$. Dann

$$\mathsf{MPCP}_\Sigma \preccurlyeq \mathsf{PCP}_{\Sigma \cup \{\$, \#\}}$$

Mit diesem Lemma ist es uns längerfristig möglich zu zeigen, dass $\mathsf{UNIV} \preccurlyeq \mathsf{PCP}$, dazu nutzen wir die Transitivität der $\preccurlyeq$-Relation aus (Übung).

Beweis

Seien

$$w = a_1 \ldots a_m \in \Sigma^+$$
$$\overline{w} = \#a_1\#a_2\# \ldots \#a_m\#$$
$$\overleftarrow{w} = \#a_1\#a_2\# \ldots \#a_m$$
$$\overrightarrow{w} = a_1\#a_2\# \ldots \#a_m\#$$

Wir benötigen eine Transformation F, die Instanzen von MPCP_Σ in Instanzen von $\mathsf{PCP}_{\Sigma \cup \{\$, \#\}}$ überführt, sodass lösbare Instanzen in lösbare Instanzen überführt werden, und unlösbare in unlösbare.

$$\underbrace{(k, ((x_1, y_1), \ldots, (x_k, y_k)))}_{A} \to \underbrace{(k+2, ((\overline{x_1}, \overleftarrow{y_1}), (\overrightarrow{x_1}, \overleftarrow{y_1}), (\overrightarrow{x_2}, \overleftarrow{y_2}), \ldots, (\overrightarrow{x_k}, \overleftarrow{y_k}), (\$, \#\$)))}_{B}$$

Hat A die Lösung I mit $i_1 = 1$, dann hat B die Lösung $I = (1, i_2, \ldots, i_n)$.

$$\begin{aligned} x_1 x_{i_2} \ldots x_{i_n} &= y_1 y_{i_2} \ldots y_{i_n} \\ \overline{x_1} \diagup x_{i_2} \ldots \diagup x_{i_n} &= \diagdown y_1 \diagdown y_{i_2} \ldots \diagdown y_{i_n} \#\$ \\ \#x_1 \#x_{i_2} \ldots \#x_{i_n} \#\$ &= \#y_1 \#y_{i_2} \ldots \#y_{i_n} \#\$ \end{aligned}$$

Hat B eine Lösung, dann hat auch A eine Lösung. Es gibt in B nur ein Paar, das als Anfang des Lösungswortes dienen kann:

$$\left(\overline{x_1}, \diagdown y_1\right) = \left(\underline{\#} x_1 \#, \underline{\#} y_1\right)$$

(ansonsten unterscheiden sich nämlich die Anfangsbuchstaben).

6.10 Das Wortproblem für allgemeine Grammatiken

Man bilde als Instanz die Kodierung einer Grammatik $G = (\Sigma, V, P, S)$ und nehme ein Wort $w \in \Sigma^*$. Es stellt sich die Frage, ob w von G generiert wird, also ob $w \in L(G)$.

6.10.1 Definition

Die Sprache WORT ist wie folgt definiert:

$$\mathsf{WORT} = \{cod(G)\,\$w | w \in L(G)\}$$

6.10.2 Lemma

$$\mathsf{UNIV} \preccurlyeq \mathsf{WORT}$$

Beweis

Wir haben bereits gezeigt, dass man für jede Turing-Maschine M automatisch ein Grammatik G_M konstruieren kann mit $L(G_M) = L(M)$.

$$w\$x \xrightarrow{F} cod(G_M)\,\$decod(x) \quad \square$$

Wir haben bereits

$$\mathsf{UNIV} \preccurlyeq \mathsf{WORT} \overset{?}{\preccurlyeq} \mathsf{MPCP} \preccurlyeq \mathsf{PCP}$$

gegeben, müssen also nur noch den Übergang von WORT zu MPCP zeigen.

6.10.3 Lemma

$$\mathsf{WORT} \preccurlyeq \mathsf{MPCP}$$

Beweis

Wir müssen aus einer Instanz $G\ w$ von WORT eine Instanz $F(G, w)$ von MPCP generieren, die eine Lösung genau dann hat, wenn $w \in L(G)$.

Idee

Man konstruiert $F(G, w)$ so, dass die Lösung dieses MPCPs einer Derivation von w in G entspricht.

$$\$w \Leftarrow U_m \Leftarrow U_{m-1} \Leftarrow \ldots \Leftarrow U_2 \Leftarrow U_1 \Leftarrow S€$$
$$\$w \Leftarrow U_m \Leftarrow U_{m-1} \Leftarrow \ldots \Leftarrow U_2 \Leftarrow U_1 \Leftarrow S€$$

6.10.3.1 Beispiel

$CAb \to acb$ ist eine Produktion in G.

$$\$abacbba \Leftarrow abCAbba \Leftarrow \ldots$$
$$\$abacbba$$

Seien $G = (\Sigma, V, S, P)$, $w \in \Sigma^*$. Weiterhin seien

$$P = \{\alpha_j \to \beta_j | 1 \leqslant j \leqslant t\}$$
$$V = \{A_i | 1 \leqslant i \leqslant |V|\}$$
$$\Sigma = \{a_i | 1 \leqslant i \leqslant |\Sigma|\}$$

$\$w \Leftarrow$	$\Leftarrow$	A_1	$\ldots$	$A_{\lvert V\rvert}$
$\$$	$\Leftarrow$	A_1		$A_{\lvert V\rvert}$

a_1	$\ldots$	$a_{\lvert\Sigma\rvert}$	α_1	$\ldots$	α_t	€
a_1		$a_{\lvert\Sigma\rvert}$	β_1		β_t	$\Leftarrow S$ €

Hierbei handelt es sich lediglich um die noch ungeordneten erzeugten Karten. Um zur Lösung zu gelangen, muss man diese umordnen.

6.10.4 Satz

PCP ist **nicht** entscheidbar.

Beweis

UNIV $\preccurlyeq$ WORT $\preccurlyeq$ MPCP $\preccurlyeq$ PCP, also UNIV $\preccurlyeq$ PCP. UNIV ist aber nicht entscheidbar.

6.10.5 Satz

Die folgenden Probleme sind **nicht** entscheidbar. Gegeben sind kontextfreie Grammatiken G_1 und G_2.

i. Gilt $L(G_1) \cap L(G_2) = \emptyset$?

ii. Gilt $L(G_1) \cap L(G_2)$ ist endlich?

iii. Gilt $L(G_1) \subseteq L(G_2)$?

iv. Gilt $L(G_1) = L(G_2)$?

v. Gilt $L(G_1) \cap L(G_2)$ ist kontextfrei?

6.10.6 Anmerkung

Für reguläre Grammatiken sind diese Probleme aber entscheidbar.

Beweis

Wir wollen zeigen, dass eine Lösung dieser Fragen die Lösung von PCP implizieren würde.

i. Gilt $L(G_1) \cap L(G_2) = \emptyset$?
Sei $(k, ((x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)))$ eine Instanz von PCP. Dabei sind $X = (x_1, \ldots, x_k)$ und $Y = (y_1, \ldots, y_k)$.
Wir betrachten die Sprachen

$$S_X = \left\{ I^R \$ X[I] | I \in \{1, \ldots, k\}^+ \right\}$$
$$S_Y = \left\{ J^R \$ Y[J] | J \in \{1, \ldots, k\}^+ \right\}$$

$S_X \cap S_Y \neq \emptyset \Leftrightarrow \exists I \in \{1, \ldots, k\}^+$ mit $I^R \$ X[I] = I^R \$ Y[I] \Leftrightarrow \exists I : X[I] = Y[I]$ also hat PCP eine Lösung.
Sei G_X eine kontextfreie Grammatik für S_X mit

$$G_X = (\Sigma \cup \{\$\} \cup \{1, \ldots, k\}, \{S\}, S, P)$$

und

$$P = \{S \to 1Sx_1, S \to 2Sx_2, \ldots, S \to kSx_k, S \to \$\}.$$

Eine kontextfreie Grammatik G_Y für die Sprache S_Y lässt sich analog angeben.
Man beachte hierbei, dass S_X und S_Y sogar deterministisch kontextfrei sind.

ii. Gilt $L(G_1) \cap L(G_2)$ ist endlich?
Es gibt nur folgende Möglichkeiten:

- $S_X \cap S_Y = \emptyset$ (also endlich) oder
- $S_X \cap S_Y$ ist unendlich, denn $\exists I$ mit: $I^R \$ X[I] = I^R \$ Y[I]

$$\Rightarrow \underbrace{I^R \ldots I^R}_{l} \$ X[\underbrace{I \ldots I}_{l}] = \underbrace{I^R \ldots I^R}_{l} \$ Y[\underbrace{I \ldots I}_{l}] \quad \forall l \geqslant 1$$

iii. Gilt $L(G_1) \subseteq L(G_2)$?
Es gilt $A \cap B = \emptyset \Leftrightarrow A \subseteq \overline{B}$. Gegeben sei eine Instanz k, X, Y von PCP. Man bilde eine kontextfreie Grammatik G_1 für S_X sowie eine kontextfreie Grammatik G_2 für $\overline{S_Y}$. Letzteres geht, da S_Y deterministisch kontextfrei ist. Daraus folgt:

$$\begin{aligned} L(G_1) \subseteq L(G_2) &\Leftrightarrow S_X \subseteq \overline{S_Y} \\ &\Leftrightarrow S_X \cap S_Y = \emptyset \\ &\Leftrightarrow \text{PCP (ist unlösbar)} \end{aligned}$$

Die Antwort auf diese Frage ist also gleichbedeutend mit der Lösung des PCP-Problems. Damit ist die Frage also nicht entscheidbar.

iv. Gilt $L(G_1) = L(G_2)$?
Es gilt, dass $A \subseteq B \Leftrightarrow A \cup B = B$. Es seien zwei kontextfreie Grammatiken G_1 und G_2 gegeben. Zur Entscheidung der Frage $L(G_1) \subseteq L(G_2)$ bilde man eine kontextfreie Grammatik $\widehat{G}$ für $L(G_1) \cup L(G_2)$. Damit erhalten wir die neue Frage, ob $L\left(\widehat{G}\right) = L(G_2)$. Das ist aber äquivalent zu $L\left(\widehat{G}\right) \subseteq L(G_2)$, was man auf iii. zurückführen kann.

v. Ist $L(G_1) \cap L(G_2)$ kontextfrei?
Man nehme eine Instanz k, X, Y von PCP.

$$L_X = \left\{ I^R \# K \$ K^R \# X[I] | I, K \in \{1, \ldots, k\}^+ \right\}$$
$$L_Y = \left\{ J^R \# J \$ H^R \# Y[H] | J, H \in \{1, \ldots, k\}^+ \right\}$$

a. L_X ist kontextfrei. Die Grammatik lautet:

$$\begin{aligned} S &\to i S x_i && 1 \leqslant i \leqslant k \\ S &\to i \# T \# x_i && 1 \leqslant i \leqslant k \\ T &\to i T i && 1 \leqslant i \leqslant k \\ T &\to i \$ i && 1 \leqslant i \leqslant k \end{aligned}$$

b. L_Y ist kontextfrei. Wir betrachten den Fall $L_X \cap L_Y \neq \emptyset$. Gilt $L_X \cap L_Y \neq \emptyset$, dann folgt daraus $I = J \wedge J = K \wedge K = H$. Daher ist $H = I$, womit $X[I] = Y[H] = Y[I]$ und eine Lösung für das PCP existiert. Das Wort in $L_X \cap L_Y$ hat die Form $I^R \# I \$ I^R \# X[I]$, womit dann aber $\forall I$ Lösungen von PCP folgt, dass auch I^n eine Lösung für das PCP darstellt.
Anders herum gesagt:

$$\begin{aligned} L_X \cap L_Y = \emptyset \Leftrightarrow\ & \text{Das PCP hat keine Lösung} \\ \Rightarrow\ & L_X \cap L_Y \text{ ist kontextfrei} \end{aligned}$$

Hat aber das PCP eine Lösung I, dann enthält $L_X \cap L_Y$ das Wort

$$I^{R^n} \# I^n \$ I^{R^n} \# X[I^n] \quad \forall n \geqslant 1.$$

Verwendet man das Pumping Lemma für kontextfreie Sprachen, so stellt man fest, dass $L_X \cap L_Y$ nicht kontextfrei ist.
Wir haben das Problem somit auf den Fall von i. zurückgeführt, weswegen es nicht entscheidbar ist.

6.11 Spracheinteilung

[Diagramm wird nachgereicht]

Kapitel 7

Komplexitätstheorie

7.1 Motivation

Was, wenn L zwar entscheidbar ist, aber jede TM, die L entscheidet, bei einer Eingabe der Länge n immer mindestens

$$\underbrace{10^{10^{10^{10^{\cdots^{10}}}}}}_{n \text{ Schritte}}$$

benötigt?

Man stellt fest, dass sich die Probleme dieser Art in eine Hierarchie einordnen lassen, die man sich wie folgt vorstellen kann:

schwarz ... grau ... weiß
unentscheidbar **entscheidbar**

Was, wenn wir Turing-Maschinen mit Ressourcenbeschränkungen betrachten? Hierunter verstehen wir

Ressourcen: - Zeit: Anzahl der Schritte
- Platz: Anzahl der Bandzellen

Es wird deutlich, dass weitere Unterteilungen der oben angegebenen Struktur vonnöten sind.

7.2 Grundlegende Definitionen

7.2.1 Definition

Seien $f, g : \mathbb{N} \to \mathbb{R}$. Eine Relation $f \geqslant_{f\ddot{u}} g$ ist definiert durch

$f \geqslant_{f\ddot{u}} g : \forall n \in \mathbb{N} :$ bis auf endlich viele Ausnahmen gilt $f(n) \geqslant g(n)$.

Eine dazu äquivalente Definition ist

$$f \geqslant_{f\ddot{u}} g : \exists n_0 \in \mathbb{N} : \forall n \geqslant n_0 : f(n) \geqslant g(n).$$

Analog definiert man:

$$\leqslant_{f\ddot{u}}, <_{f\ddot{u}}, >_{f\ddot{u}}, =_{f\ddot{u}}$$

7.2.2 Definition

Die Menge $\mathsf{F\ddot{U}P}$ ist definiert durch

$$\mathsf{F\ddot{U}P} = \{f : \mathbb{N} \to \mathbb{R} | f >_{f\ddot{u}} 0\}.$$

7.2.3 Definition

Sei $g \in \mathsf{F\ddot{U}P}$. Dann ist:

$$\begin{aligned}
\mathcal{O}(g) &= \left\{f \in \mathsf{F\ddot{U}P} | \exists c > 0 : f \leqslant_{f\ddot{u}} c \cdot g\right\} \\
o(g) &= \left\{f \in \mathsf{F\ddot{U}P} | \forall c > 0 : f \leqslant_{f\ddot{u}} c \cdot g\right\} \\
\Omega(g) &= \left\{f \in \mathsf{F\ddot{U}P} | \exists c > 0 : f \geqslant_{f\ddot{u}} c \cdot g\right\} \\
\omega(g) &= \left\{f \in \mathsf{F\ddot{U}P} | \forall c > 0 : f \geqslant_{f\ddot{u}} c \cdot g\right\} \\
\Theta(g) &= O(g) \cap \Omega(g)
\end{aligned}$$

7.2.4 Definition

Sei $f \in \mathsf{F\ddot{U}P}$. Eine deterministische Turing-Maschine hat folgende Eigenschaften:

i. M hat eine **worst-case Laufzeit** von $f(n)$, wenn M für jedes Eingabewort w nach höchstens $f(|w|)$ Schritten hält.

ii. M hat einen **worst-case Platzverbrauch** von $f(n)$, wenn M für jede Eingabe w hält und dabei auf jedem Arbeitsband höchstens $f(|w|)$-viele Bandzellen verwendet.

Sei M nun eine nicht-deterministische Turing-Maschine. M hat folgende Eigenschaften:

iii. M hat eine **worst-case Laufzeit** von $f(n)$, wenn M für jedes $w \in L(M)$ eine akzeptierende Berechnung hat, die aus höchstens $f(|w|)$ vielen Rechenschritten besteht.

iv. M hat einen **worst-case Platzverbrauch** von $f(n)$, wenn M für jedes $w \in L(M)$ eine akzeptierende Berechnung hat, die auf jedem Arbeitsband höchstens $f(|w|)$-viele Bandzellen verwendet.

7.2.5 Definition

Sei $f \in \mathsf{F\ddot{U}P}$.

- $\mathsf{DTIME}(f)$ ist die Familie aller Sprachen L, für die es eine deterministische Turing-Maschine M gibt, die L mit einer worst-case Laufzeit in $\mathcal{O}(f)$ entscheidet.
- $\mathsf{DSPACE}(f)$ ist die Familie aller Sprachen L, für die es eine deterministische Turing-Maschine M gibt, die L mit einem worst-case Platzverbrauch in $\mathcal{O}(f)$ entscheidet.
- $\mathsf{NTIME}(f)$ ist die Familie aller Sprachen L, für die es eine nichtdeterministische Turing-Maschine M gibt, die L mit einer worst-case Laufzeit in $\mathcal{O}(f)$ akzeptiert.

- $\mathsf{NSPACE}\,(f)$ ist die Familie aller Sprachen L, für die es eine nichtdeterministische Turing-Maschine M gibt, die L mit einem worst-case Platzverbrauch in $\mathcal{O}\,(f)$ akzeptiert.

Als Referenzmaschine verwenden wir im Folgenden eine Turing-Maschine mit einem ausschließlich lesbaren Eingabeband und einer beliebigen, aber konstanten Anzahl von Arbeitsbändern.

7.2.6 Beispiel

Wir betrachten die Sprache $L = \left\{a^i b^i | i \in \mathbb{N}\right\}$.

Behauptung 1

$$L \in \mathsf{DTIME}\left(n^2\right)$$

Beweis

baue eine deterministische Turing-Maschine M. Diese funktioniert wie folgt:

```
repeat
   lösche a am linken Bandende
   rücke ganz nach rechts
   lösche b
   rücke ganz nach links
```

Laufzeit $\approx \sum_{j=i}^{1}(4j) \approx \mathcal{O}\left(n^2\right)$

Behauptung 2

$$L \in \mathsf{DTIME}\,(n)$$

Beweis

```
kopiere a's auf Arbeitsband
teste, ob gleich viele b's auf dem Eingabeband wie a's auf dem
Arbeitsband
```

7.3 Inklusionen

7.3.1 Lemma

$$\begin{array}{ccc} \mathsf{DTIME}(f) & \underset{\neq ?}{\subseteq} & \mathsf{DSPACE}(f) \\ \overset{?}{\not\in} \cap & & \overset{?}{\not\in} \cap \\ \mathsf{NTIME}(f) & \underset{\neq ?}{\subseteq} & \mathsf{NSPACE}(f) \end{array}$$

7.3.2 Lemma A

i. $L \in \mathsf{DTIME}(f(n)) \Rightarrow L \in \mathsf{DSPACE}(f(n))$

ii. $L \in \mathsf{DSPACE}(f(n)) \Rightarrow L \in \mathsf{DTIME}\left(a^{f(n)}\right)$, für irgendeine von L abhängige Konstante a

Voraussetzung ist hierbei, dass $f(n) \geqslant \log_2 n$.

Beweis

i. Trivial.

ii. $L \in \mathsf{DSPACE}(f(n)) \Rightarrow \exists$ deterministische Turing-Maschine M, die L entscheidet und auf Eingabe w höchstens $g(|w|)$ viele Bandzellen/Arbeitsband verwendet, $g \in (\mathcal{O}(f))$: $g(n) \leqslant C \cdot f(n) \quad \forall n \in \mathbb{N}$.
M habe Q viele Zustände, B sei die Größe des Arbeitsalphabets und k sei die Anzahl der Arbeitsbänder. M bekomme eine Eingabe w. Dann ist $n = |w|$ die Anzahl der möglichen Eingabekopfpositionen und $N = g(|w|)$ die der möglichen Arbeitskopfpositionen. Man muss sich nun überlegen, wie viele verschiedene Konfigurationen die Maschine M bei der Eingabe w erreichen kann. Man stellt fest, dass diese Zahl höchstens

$$X = Q \cdot n \cdot \left(B^N \cdot N\right)^k$$

annehmen kann. B^N stellt hierbei die Anzahl der möglichen Inhalte des Arbeitsbandes dar.
Wir bauen nun eine Maschine M' mit $k+1$ Arbeitsbändern. Sie arbeitet wie folgt:

```
berechne X und schreibe das Ergebnis auf das Arbeitsband
k+1
emuliere M und verringere mit jedem Schritt den Zähler auf
dem Arbeitsband k+1
lehne ab, wenn der Zähler 0 erreicht
```

Wenn der Zähler nämlich 0 erreicht, dann hat die Maschine definitiv eine Konfiguration mehrfach durchlaufen, man kann also mit Sicherheit sagen, das M divergiert. In diesem Fall entscheidet M aber nichts, insbesondere nicht L.
Man sieht, dass $Q \leqslant Q^N, n \leqslant 2^{f(n)}$ und $N \leqslant 2^N$. Daraus folgt mittels $N = g(n) \leqslant C \cdot f(n)$, dass

$$\begin{aligned} Q \cdot n \cdot \left(B^N \cdot N\right)^k &\leqslant 2^{f(n)} \cdot (Q \cdot B \cdot 2)^k \\ &\leqslant 2^{f(n)} \left((Q \cdot B \cdot 2)^{C \cdot k}\right)^{f(n)} \\ &= \left(2 \cdot (Q \cdot B \cdot 2)^{C \cdot k}\right)^{f(n)} \\ &:= a \end{aligned}$$

M' macht also höchstens $a^{f(n)}$ Schritte, womit $L \in \mathsf{DTIME}\left(a^{f(n)}\right)$.

7.3.3 Lemma B

i. $L \in \mathsf{DTIME}(f(n)) \Rightarrow L \in \mathsf{NTIME}(f(n))$

ii. $L \in \mathsf{NTIME}(f(n)) \Rightarrow L \in \mathsf{DTIME}\left(b^{f(n)}\right)$, für irgendeine von L abhängige Konstante b

Beweis

i. Trivial.

ii. Hier greifen wir auf den Beweis zu 5.2.7 zurück. Dort haben wir „Leitstrings“ betrachtet, weswegen auch jetzt die Idee ist, alle möglichen „Leitstrings“ zu betrachten:

$$L \in \mathsf{NTIME}(f(n)) \Rightarrow \exists \text{ nichtdeterministische Turing-Maschine } M,$$

wobei M die Sprache L in Laufzeit $\leqslant C \cdot f(n)$ akzeptiert. Die Maschine bekommt eine Eingabe w mit $|w| = n$ und $N = C \cdot f(n)$. β ist die Anzahl der nichtdeterministischen Wahlmöglichkeiten bzw. Schritte.
$\Rightarrow \exists$ weniger als β^N bzw. genauso viele Leitstrings, die den deterministischen Ablauf von M diktieren. Man baue eine deterministische Turing-Maschine M', die M auf allen Leitstrings ausprobiert. Die Anzahl der Schritte von M' auf der Eingabe w ist höchstens

$$X \leqslant \beta^N \cdot N \leqslant (2\beta)^N = (2\beta)^{C \cdot f(n)} = b^{f(n)}$$

Somit ist $L \in \mathsf{DTIME}\left(b^{f(n)}\right)$.

7.3.4 Lemma C

i. $L \in \mathsf{DSPACE}(f(n)) \Rightarrow L \in \mathsf{NSPACE}(f(n))$

ii. $L \in \mathsf{NSPACE}(f(n)) \Rightarrow L \in \mathsf{DSPACE}\left(f^2(n)\right)$

Voraussetzung ist hierbei, dass f mit $f\left(n^2\right)$ viel Platz berechnet werden kann. In den wenigsten Fällen ist diese Voraussetzung aber eine Einschränkung.

7.3.5 Satz von Savitch

Lemma C, ii heißt auch der **Satz von Savitch**. Er ist benannt nach Walter Savitch, der 1970 erstmals unten stehenden Beweis geführt hat.

Beweis

i. Trivial.

ii.

$$L \in \mathsf{NSPACE}(f(n)) \Rightarrow \exists \text{ nichtdeterministische Turing-Maschine } M,$$

wobei M die Eingabe $w \in L$ mit Platz $\leqslant C \cdot f(n)$ akzeptiert.

Idee

Man betrachte nun alle Konfigurationsgraphen G_M von M bei einer Eingabe w. Die Knoten sind hierbei alle möglichen Konfigurationen. Eine Kante $[K, K'\rangle$ existiert genau dann, wenn M in einem Rechenschritt von K nach K' gelangt, also $K \underset{M}{\vdash} K'$.
Die Anzahl der Konfigurationen in G_w ist kleiner oder genauso groß wie $a^{f(n)}$. Hierzu siehe den Beweis zu 7.2.8. Die Maschine M akzeptiert w genau dann, wenn es einen gerichteten Pfad in G_w von $Init_w$ (Anfangskonfiguration) zu irgendeiner Endkonfiguration φ gibt. Wir skizzieren nun die deterministische Maschine:

`teste jede Konfiguration` K `in` G_w
`if` K `eine Endkonfiguration und` $\exists$ `ein Pfad von` $Init_w$ `nach` K `in` G_w`,`
   `then akzeptiere` w
   `else lehne` w `ab`

Insbesondere ist der zweite Teil der Prämisse die zentrale Frage hiefür.

Verfeinerung

Man baue den gerichteten Graph $G = (V, E)$, der unter Umständen sehr groß werden kann.

$$s \in V \text{ Startknoten}$$
$$F \subset V \text{ Endknoten}$$

Es stellt sich die Frage, ob ein Pfad in G von s zu irgendeinem Knoten in F existiert. Durch folgendes Programm kann man diese aber beantworten:

`for` $v \in V$ `do if` $v \in F$ `and reachable(`s`,`v`) then return true`

Man beachte, dass der Graph $G = (V, E)$ implizit gegeben ist.
Folgende Rechenannahmen werden getroffen:

a. Man kann die Knoten in V in $\mathcal{O}(\log |V|)$ Platz aufzählen
b. für $v \in V$ kann man testen, ob $v \in F$
c. für $u, v \in V$ kann man testen, ob
   - $u = v$
   - $[u, v\rangle \in E$

Alle diese Operationen benötigen $\mathcal{O}(\log |V|)$ an Platz. `reachable(`s`,`v`,`l`)` bedeutet, ob v von s über Pfad der Länge $\leqslant l$ erreichbar ist, `reaachable(`s`,`v`)` $\approx$ `reachable(`s`,`v`,`$|V|$`)`. Man formuliert die Funktion wie folgt:

```
function reachable(s,v,l)
{
   if l = 0 then return (s = v)
```

```
   if l = 1 then return (s = v) or ([s,v⟩ ∈ E)
   h_1 = ⌊l/2⌋, h_2 = ⌈l/2⌉
   for each m ∈ V do
      if reachable(s,m,h_1) and reachable(m,v,h_2)
         then return true
}
```

Man definiere im Folgenden $S(N,l)$ als den Platzverbrauch von `reachable(`s`,`v`,`l`)` in einem Graphen mit N Knoten. Dann ist

$$S(N,0) = \mathcal{O}(\log N)$$
$$S(N,1) = \mathcal{O}(\log N)$$
$$S(N,l) = \mathcal{O}(\log N) + \mathcal{O}(\log l) + S\left(N, \frac{l}{2}\right) + \cancel{S\left(N, \frac{l}{2}\right)}$$

$$\Rightarrow S(N,l) = \mathcal{O}((\log N + \log l)\log l)$$

$l = \underbrace{N}_{|V|} \overset{\approx}{\Rightarrow}$ Platzverbrauch von `reachable(`s`,`v`,`$|V|$`)` ist $\underbrace{\mathcal{O}\left(\log^2 |V|\right)}_{=\mathcal{O}(f^2(n))}$
$|V| \leqslant A^{f(n)}$
$\log |V| = \mathcal{O}(f(n))$

7.3.6 Korollar

Sei $\gamma : \mathbb{N} \to \mathbb{R}$ eine Funktion, die nicht-fallend ist und es gelte $\lim_{n\to\infty} \gamma(n) = \infty$.

- $\mathsf{DSPACE}(f(n)) \subseteq \mathsf{DTIME}\left(2^{\gamma(n)f(n)}\right)$
- $\mathsf{NTIME}(f(n)) \subseteq \mathsf{DTIME}\left(2^{\gamma(n)f(n)}\right)$
- $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{DSPACE}\left(f^2(n)\right)$

7.3.7 Lemma D (Komplementbildung)

i. $L \in \mathsf{DTIME}(f(n)) \Rightarrow \overline{L} \in \mathsf{DTIME}(f(n))$

ii. $L \in \mathsf{DSPACE}(f(n)) \Rightarrow \overline{L} \in \mathsf{DSPACE}(f(n))$

iii. $L \in \mathsf{NSPACE}(f(n)) \Rightarrow \overline{L} \in \mathsf{NSPACE}(f(n))$

7.3.8 Satz von Immermann/Szelepcsényi

Lemma D, iii. heißt **Satz von Immermann/Szelepcsényi**.

Beweis

i. Trivial.

ii. Trivial.

iii. Sei $L \in \mathsf{NSPACE}(f(n))$. Dann existiert eine nichtdeterministische Turing-Maschine M mit Platzverbrauch $f(n)$ mit $L(M) = L$. Seien weiterhin $w \in \Sigma^*, |w| = n$. Wir brauchen eine nichtdeterministische Turing-Maschine mit $f(n)$ Platzverbrauch, der genau alle $w \notin L$ akzeptiert. Wir betrachten dazu $KG_M(w, N)$ mit $N = f(n)$. Wir müssen „beweisen", dass es keinen Pfad von $Init_w$ zu einer Endkonfiguration gibt.

Verfeinerung

Wir haben einen gerichteten Graphen $G = (V, E)$ mit Startknoten $s \in V$ und Endknoten $F \subset V$. Wir müssen beweisen, dass es keinen Pfad von s zu einem Knoten in F gibt:

$$\forall v \in F : \nexists \text{ Pfad von } s \text{ zu } v$$

$$R_s(l) = \{v \in V | v \text{ von } s \text{ über Pfad der Länge } \leqslant l \text{ erreichbar}\}$$
$$N_s(l) = |R_s(l)|$$

Wir suchen einen Test, um zu zeigen, dass $v \notin R_s(|V|)$.

Behauptung 0

Man kann mit $\mathcal{O}(\log |V|)$ Platz testen, dass $v \in R_s(l)$. Das Testen des Platzes erfolgt nichtdeterministisch.

Beweis

Errate Pfad und verifiziere.

Behauptung 1

Man nehme an, $N_s(l)$ sei bekannt. Damit kann man mit $\mathcal{O}(log |V|)$ Platz nichtdeterministisch testen, ob $v \notin R_s(l)$.

Beweis

```
z = 0
for each x ∈ V \ {V} do
   if x ∈ R_s(l)
      then
         beweise es wie bei Behauptung 0
         z = z + 1
if z = N_s(l − 1)
   then antworte Ja.
```

Behauptung 2

$N_s(l-1)$ sei bekannt. Man kann nichtdeterministisch mit $\mathcal{O}(\log |V|)$ Platz $N_s(l)$ berechnen.

Beweis

```
z = 0
for each x ∈ V do
   if x ∈ R_s(l) then
      verifiziere wie bei Behauptung 0
      z = z + 1
   if x ∉ R_s(l) then
      for each u ∈ V do
         zeige, dass keine Kante von u ∈ R_s(l − 1) zu x:
            teste, ob u ∈ R_s(l − 1) (Behauptung 0 und 1)
            falls ja, verifiziere, [u, x⟩ ∉ E
```

Führen wir nun den Beweis von Lemma D zuende:
$\nexists$ Pfad von s nach v: $N_s(0) = 1$,

```
for l = 1 to |V| do berechne N_s(l) aus N_s(l − 1)
   teste, ob v ∈ R_s(|V|) (Behauptung 1)
```

Platzverbrauch: $\mathcal{O}\left(\underbrace{\log|V|}_{=\mathcal{O}(f(n))}\right)$, $|V| \leqslant A^{f(n)}$

7.3.9 Lemma E (Hierarchiesatz 1)

wir nehmen an, dass $f \in \mathcal{O}(g)$ gilt. Dann ist klar, dass

$$\mathsf{DTIME}(f(n)) \subseteq \mathsf{DTIME}(g(n))$$
$$\mathsf{DSPACE}(f(n)) \subseteq \mathsf{DSPACE}(g(n))$$

7.3.10 Lemma F (Strikte Hierarchie)

i. Sei $f \in o(g)$ für $\mathcal{O}(f)$ Platz berechenbar. Dann gilt

$$\mathsf{DSPACE}(f(n)) \subsetneq \mathsf{DSPACE}(g(n))$$

ii. $f \in o\left(\frac{g(n)}{\log g(n)}\right)$ für $\mathcal{O}(f)$ Zeit berechenbar. Dann gilt

$$\mathsf{DTIME}(f(n)) \subsetneq \mathsf{DTIME}(g(n)).$$

7.3.11 Anmerkungen

i. Analoges gilt für nichtdeterministische Klassen.

ii. Berechenbarkeitsbedingungen für f sind notwendig.

7.3.12 Beispiel (Borodin's Gap Theorem)

Es gibt eine berechenbare Funktion $f : \mathbb{N} \to \mathbb{N}, \lim_{n\to\infty} f(n) = \infty$, sodass $\mathsf{DSPACE}(f(n)) = \mathsf{DSPACE}\left(2^{2^{2^{f(n)}}}\right)$.

7.4 Komplexitätsklassen

Wir haben die Komplexitätsklassen $\mathsf{DTIME}(f)$, $\mathsf{NTIME}(f)$, $\mathsf{DSPACE}(f)$ und $\mathsf{NSPACE}(f)$ gegeben. Somit stellt sich natürlich die Frage, für welche f sie interessant sind.
Ein Indiz geben bekannte Algorithmen:

- suchen (und löschen) des maximalen Elementes einer unsortierten Liste:

$$f(n) = n$$

- n-malige Maximumsuche und Aneinanderreihung liefert Sortierung:

$$f(n) = n^2$$

- Multiplikation zweier $n \times n$-Matrizen:

$$f(n) = n^3$$

Polynome tauchen häufig als Zeitschranken auf. Sie sind auch mathematisch „angenehm“: Polynome sind abgeschlossen unter

- Addition,
- Multiplikation und
- Komposition.

Wir betrachten gröbere Klassen:

- $\mathsf{P} = \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}\left(n^k\right)$:
  Probleme, die sich in polynomieller Zeit lösen lassen.
- $\mathsf{NP} = \bigcup_{k\in\mathbb{N}} \mathsf{NTIME}\left(n^k\right)$:
  Probleme, deren Lösungen in polynomieller Zeit verifizierbar sind.
- $\mathsf{PSPACE} = \bigcup_{k\in\mathbb{N}} \mathsf{DSPACE}\left(n^k\right)$
- $\mathsf{NPSPACE} = \bigcup_{k\in\mathbb{N}} \mathsf{NSPACE}\left(n^k\right)$
- $\mathsf{L} = \mathsf{DSPACE}(\log n)$
- $\mathsf{NL} = \mathsf{NSPACE}(\log n)$

7.4.1 Satz

Es gilt:

$$\mathsf{L} \underset{\checkmark}{\subseteq} \mathsf{NL} \underset{?}{\subseteq} \mathsf{P} \underset{\checkmark}{\subseteq} \mathsf{NP} \underset{?}{\subseteq} \mathsf{PSPACE} \underset{\checkmark}{\subseteq} \mathsf{NPSPACE} \quad (*)$$

Beweis

Die durch ✓ bezeichneten Inklusionen wurden bereits bewiesen oder ergeben sich unmittelbar aus bereits geführten Beweisen. Also genügt es, die zwei verbliebenen Inklusionen zu zeigen. Man betrachte zunächst die Relation zwischen NP und DSPACE:

$$\begin{aligned} & S \in \mathsf{NP}, \exists k : S \in \mathsf{NTIME}\left(n^k\right), \\ & S \in \mathsf{NSPACE}\left(n^k\right) \\ \Rightarrow\ & S \in \mathsf{DSPACE}\left(n^{2 \cdot k}\right) \\ \Rightarrow\ & S \in \mathsf{PSPACE} \end{aligned}$$

Die Relation zwischen L und P ist da schon schwieriger:

Behauptung 1

Es gilt: $\mathsf{PSPACE} = \mathsf{NPSPACE}$.

Beweis

- $\mathsf{PSPACE} \subseteq \mathsf{NPSPACE}$: Trivial.
- $\mathsf{NPSPACE} \subseteq \mathsf{PSPACE}$:

$$\begin{aligned} & \exists k : S \in \mathsf{NSPACE}\left(n^k\right) \\ \Rightarrow & \exists k : S \in \mathsf{DSPACE}\left(n^{2 \cdot k}\right) \\ \Rightarrow & \quad S \in \mathsf{PSPACE} \end{aligned}$$

Behauptung 2

Es gilt: $\mathsf{NL} \neq \mathsf{NPSPACE}$.

Beweis

$$\mathsf{NL} = \mathsf{NSPACE}\left(\log n\right) \subseteq \mathsf{DSPACE}\left(\log^2 n\right) \underset{\text{(HS)}}{\subsetneq} \mathsf{PSPACE}\left(n\right) \subseteq \mathsf{NPSPACE}\left(n\right)$$

Damit wird $(*)$ zu

$$\mathsf{L} \subseteq \mathsf{NL} \subseteq^* \mathsf{P} \subseteq^* \mathsf{NP} \subseteq^* \mathsf{PSPACE}\,(= \mathsf{NPSPACE})$$

wobei wenigstens eine der Inklusionen $\subseteq^*$ echt ist. HS steht hierbei für den Hierarchiesatz.

Vermutung

Alle 5 Klassen sind verschieden.
Bis heute ist es aber nicht gelungen, diese Vermutung zu bestätigen.

Im Folgenden wenden wir uns dem Bereich „$\ldots \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \ldots$“ zu.

7.4.2 Abgrenzung von P gegen NP

In P sind Probleme, die sich in polynomieller Zeit lösen lassen, die also effizient lösbar sind. Man kann sich informell merken, dass

$$\mathsf{P} \approx \text{ „gut“ und } \mathsf{NP} \approx \text{ „schlecht“.}$$

7.5 Das CLIQUE-Problem

Wir geben zunächst ein Beispiel an:

7.5.1 Definition

Es sei $G = (V, E)$ ein ungerichteter Graph. $W \subseteq V$ heißt **Clique**, wenn für alle $u, v \in W$ gilt: $[u, v] \in E$.

[Skizze wird nachgereicht]

Die Ecken □ bilden eine Clique.

7.5.2 Drei Varianten des CLIQUE-Problems

V1 **Entscheidungsproblem**
Seien $G = (V, E)$ ein ungerichteter Graph und $k \in \mathbb{N}$.
Frage: Gibt es in G eine Clique W mit $|W| = k$?

V2 **Optimierungsproblem**
Sei G wieder ein ungerichteter Graph.
Frage: Was ist das größte k, so dass G eine Clique der Größe k enthält?

V3 **Optimierungs-/Berechnungsproblem**
Sei G wieder ein ungerichteter Graph.
Aufgabe: Berechne die Clique maximaler Größe.

7.5.3 Korollar

Es liegt auf der Hand:

- Ist V3 in polynomieller Zeit lösbar, dann ist auch V2 in polynomieller Zeit lösbar.
- Ist V2 in polynomieller Zeit lösbar, dann ist auch V1 in polynomieller Zeit lösbar.

Zudem stellen wir aber fest:

7.5.4 Lemma

i. Ist V1 in polynomieller Zeit lösbar, dann ist auch V2 in polynomieller Zeit lösbar.

ii. Ist V2 in polynomieller Zeit lösbar, dann ist auch V3 in polynomieller Zeit lösbar.

Beweis

i. Probiere jedes k, $1 \leqslant k \leqslant |V|$, ob es eine Clique der Größe k gibt.

ii. Bestimme die Größe k der maximalen Clique. Entferne sukzessive Ecken und teste, ob der Graph weiterhin eine Clique der Größe k enthält.

Das bedeutet aber:

7.5.5 Korollar

Bei Polynomialzeitalgorithmen lassen sich die Lösungen der drei Varianten „mit polynomialem Mehraufwand“ ineinander umrechnen. Die verschiedenen Varianten fallen zusammen.

Damit stellt sich aber die Frage nach einem besseren Algorithmus für das CLIQUE-Problem.

Vermutung

Man vermutet, dass es keinen effizienteren (also deterministisch polynomiellen) Algorithmus für das CLIQUE-Problem gibt.
Das CLIQUE-Problem (Entscheidungsvariante) ist aber in NP, das heißt: Es existiert eine nichtdeterministische Methode, die in polynomieller Zeit verifiziert, dass G eine Clique der Größe k hat. Dies ist ein Beispiel von vielen wichtigen Problemen, für die eine Lösung effizient verifiziert werden kann, aber man nicht weiß, wie man eine Lösung findet.

7.6 Weitere Probleme (Entscheidungsvariante)

Im folgenden betrachten wir einige andere Probleme:

i. BIN-PACKING-PROBLEM
Gegeben sei eine Instanz $a_1, \ldots, a_n$ mit einer Schranke $b \in \mathbb{N}$. Alles soll binär kodiert sein.
Frage: Gibt es $f : \{1, \ldots, n\} \to \{1, \ldots, k\}$, sodass

$$\forall 0 \leqslant j \leqslant k : \sum_{i:f(i)=j} a_i \leqslant b?$$

Wir verteilen also auf k Behälter Gegenstände, wobei die Menge, die jedem Behältnis j zugeordnet wird, nicht dessen Größe b überschreiten darf.

ii. TRAVELING-SALESMAN-PROBLEM
Gegeben sei ein vollständiger, gerichteter Graph mit gewichteten Kanten. Es gibt eine Kostenfunktion $c : E \to \mathbb{N}$ und eine Konstante $B \in \mathbb{N}$.

Frage: Gibt es eine Permutation $\pi \in S_n$:

$$\begin{aligned} & c\left(\left[\pi\left(1\right), \pi\left(2\right)\right\rangle\right) \\ +& c\left(\left[\pi\left(2\right), \pi\left(3\right)\right\rangle\right) \\ +& \ldots \\ +& c\left(\left[\pi\left(n-1\right), \pi\left(n\right)\right\rangle\right) \\ +& c\left(\left[\pi\left(n\right), \pi\left(1\right)\right\rangle\right) \qquad\qquad \leqslant B? \end{aligned}$$

iii. POLYGONWACHMANN-PROBLEM
Gegeben sei ein einfaches Polygon P in der Ebene ($k \in \mathbb{N}$).
Frage: Kann man k Wächter so in P positionieren, dass jeder Punkt von P von einem Wächter gesehen wird?

iv. INTEGER-PROGRAMMING-PROBLEM
Gegeben sei $a_{ij} \in \mathbb{Z}, 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant n$ in Binärkodierung.
Frage: Gibt es ein $(x_1, \ldots, x_n) \in \mathbb{Z}^n$, sodass für $1 \leqslant i \leqslant m$:

$$a_{i1} \cdot x_1 + a_{i2} \cdot x_2 + \ldots + a_{in} \cdot x_n \leqslant a_{i0}?$$

7.7 Polynomzeitreduktionen

7.7.1 Definition

Seien $L, L' \subseteq \Sigma^*$. Wir nennen L **polynomiell leichter** als L' (oder L **polynomiell reduzierbar** auf L'), in Zeichen

$$L \preccurlyeq_P L',$$

wenn es eine Funktion $f : \Sigma^* \to \Sigma^*$ gibt mit

i. $\forall w \in \Sigma^* : w \in L \Leftrightarrow f(w) \in L'$,

ii. f ist in polynomieller Zeit berechenbar.

7.7.2 Lemma 1

Seien $L, L' \subseteq \Sigma^*$. Es gilt

- Wenn $L \preccurlyeq_P L'$ und $L' \in \ \mathsf{P}$, dann ist $L \in \mathsf{P}$.
- Wenn $L \preccurlyeq_P L'$ und $L \in \mathsf{NP}$, dann ist $L' \in \mathsf{NP}$.

Hierzu vergleiche man mit den Folien der 21. Vorlesung, Folie 2.

Beweis

Ist $L \preccurlyeq_P L'$, dann $\exists$ TM F, die f mit $w \in L$ berechnet $\Leftrightarrow f(w) \in L'$. Die Laufzeit ist polynomiell, etwa $\mathcal{O}\left(n^k\right), n = |w|$.

- $L' \in \mathsf{P} : \exists$ TM M', die L' entscheidet. Die Laufzeit sei etwa $\mathcal{O}\left(n^l\right)$.
  Wir bauen eine Turing-Maschine M für L: Wir haben die Eingabe w.

i. Berechne $f(w)$ mit F

ii. Teste ob $f(w) \in L'$ mit M'

Die Turing-Maschine entscheidet L.
Laufzeit:

1. Schritt: $\mathcal{O}\left(n^k\right), |f(w)| \in \mathcal{O}\left(n^k\right)$
2. Schritt: $\mathcal{O}\left(|f\left(w\right)|^l\right) \in \mathcal{O}\left(\left(n^k\right)^l\right) \in \mathcal{O}\left(n^{k \cdot l}\right)$

Damit gilt: $L \in \mathsf{P}$.

- $L \in \mathsf{NP}$:
  Dieser Beweis hierzu erfolgt analog zu $L' \in \mathsf{P}$.

7.7.3 Lemma 2

Seien $L, L', L'' \subseteq \Sigma^*$. Wenn $L \preccurlyeq_P L'$ und $L' \preccurlyeq_P L''$ gilt, dann gilt auch $L \preccurlyeq_P L''$.

Beweis

Übung.

7.7.4 Definition

i. Eine Sprache S heißt **NP-schwer**, wenn für alle $L \in \mathsf{NP}$ gilt: $L \preccurlyeq_P S$.

ii. S heißt **NP-vollständig**, wenn S NP-schwer und $S \in \mathsf{NP}$ ist.

7.7.5 Satz 1

Ist S NP-vollständig und $S \in \mathsf{P}$, dann $\mathsf{P} = \mathsf{NP}$.

Beweis

Es gilt: $\mathsf{P} \subseteq \mathsf{NP}$. ✓
Noch zu zeigen bleibt, dass $\mathsf{NP} \subseteq \mathsf{P}$:
Wir wissen bereits, dass S NP-schwer ist. Daraus folgt für alle $L \in \mathsf{NP} : L \preccurlyeq_P S$.
Außerdem ist $S \in P$, womit nach Lemma 1 $L \in P$ ist.

7.7.6 Satz 1 (Alternative Formulierung)

Ist $\mathsf{P} \neq \mathsf{NP}$ und S NP-vollständig, so ist $S \notin P$.

Zeigt man also für S, dass S NP-vollständig ist, dann ist dies aufgrund der Vermutung $\mathsf{P} \neq \mathsf{NP}$ ein starker Grund zu der Annahme, dass $S \notin \mathsf{P}$, dass also S nicht effizient lösbar ist.

7.7.7 Beweismöglichkeiten für NP-schwer

Wie zeigt man, dass ein Problem NP-schwer ist? Prinzipiell hat man zwei Möglichkeiten:

i. Direkt oder

ii. mit Lemma 3.

7.7.8 Lemma 3

Ist L NP-schwer und $L \preccurlyeq_P S$, dann ist S NP-schwer.

Beweis

Seien L NP-schwer und $L' \in$ NP. Dann folgt aus der Definition, dass $L' \preccurlyeq_P L$. Zusammen mit $L \preccurlyeq_P S$ folgt aus Lemma 2, dass $L' \preccurlyeq_P S$.

7.8 NP-vollständige Probleme

Eine Sprache S ist NP-vollständig, wenn

i. $S \in$ NP und

ii. wenn für eine beliebige NP-vollständige Sprache L gilt: $L \preccurlyeq_P S$.

Hierbei stellt sich natürlich unmittelbar die Frage, ob es NP-vollständige Probleme überhaupt gibt. Die Antwort ist „ja“.

7.8.1 Beispiel 1

$$\text{P-UNIV} = \{\, cod(M)\$w\$1^m \mid \text{ eine NTM } M \text{ akzeptiert } w \text{ in } \leqslant m \text{ Schritten}\}$$

7.8.2 Behauptung

P-UNIV ist NP-vollständig.

Beweis

i. P-UNIV$\in$ NP: Simuliere M für m richtig geratene Schritte.

ii. $L \in$ NP : $L \preccurlyeq_P$ P-UNIV
$L \in$ NP : $\exists$ TM M mit $L(M) = L$ mit Laufzeit etwa $c \cdot n^k$

Baue TM F. Diese Turing-Maschine berechnet die Funktion $f(w) = cod(M)\$w\$1^{c \cdot n^k}$.
Es gilt:

i. $|f(w)| \in \mathcal{O}(1) + \mathcal{O}(|w|) + \mathcal{O}(n^k) \in \mathcal{O}(n^k)$

ii. $w \in L \Leftrightarrow f(w) \in$ P-UNIV.

7.8.3 Beispiel 2

Wir betrachten im Folgenden die Sprache der Mehrwertigen Erfüllbarkeit (MERF).

Instanz

Gegeben seien

- Variablen $Y_1, \ldots, Y_n$ mit Wertebereichen $B_1, \ldots, B_n$
- eine Formel F aufgebaut aus $\vee$ und $\wedge$ von Termen der Form

$$Y_i = a, \quad Y_i \neq a, \quad Y_i = Y_j \text{ und } Y_i \neq Y_j.$$

Man möchte die Frage entscheiden, ob F erfüllbar ist.

Beispiel

Wir haben $Y_1, Y_2, Y_3; B_1 = B_2 = B_3 = \{1, 2, 3, 4\}$. F könnte dann wie folgt aussehen:

$$F = (Y_1 = 1 \vee Y_2 = 3) \wedge (Y_2 = 2 \vee Y_3 = 1) \wedge (Y_1 = 2 \vee Y_3 = 2) \wedge (Y_2 = Y_3)$$

F wird wahr für $Y_1 = 1, Y_2 = 2, Y_3 = 2$. Wir definieren MERF wie folgt:

7.8.4 Definition (Sprache MERF)

$$\mathsf{MERF} = \{cod(Y_1) \ldots cod(Y_n)\, \$ cod(B_1) \ldots cod(B_n)\, \$ cod(F) \mid F \text{ erfüllbar}\}$$

7.8.5 Behauptung

MERF ist NP-vollständig.

Beweis

i. $\mathsf{MERF} \in \mathsf{NP}$: Wir raten wie immer die Variablenbelegungen und verifizieren F.

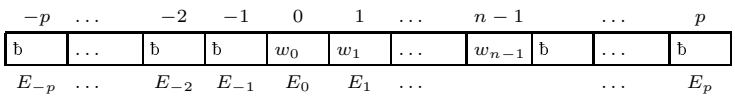
ii. $\forall L \in \mathsf{NP} : L \preccurlyeq_P \mathsf{MERF}$.
Die Idee ist die folgende: Wenn $L \in \mathsf{NP}$, dann existiert eine Turing-Maschine M mit $L(M) = L$. Wir beschreiben die Konfiguration von M und Übergänge durch Formeln.

Verfeinerung

MERF ist NP-vollständig. $L \in \mathsf{NP}$, wir müssen also zeigen, dass $L \preccurlyeq_P$ MERF. Daher brauchen wir eine Transformation

$$w \in \Sigma^* \mapsto \langle Y_1 \rangle \ldots \langle Y_n \rangle \,\$\, \langle B_1 \rangle \ldots \langle B_n \rangle \,\$ F$$

mit $\langle Z \rangle := cod(Z)$, die in polynomieller Zeit arbeitet. Wenn diese Transformation nämlich existiert, $w \in L$ und das neue Wort $\in$ MERF, so gilt $L \Leftrightarrow$ MERF. Ist $L \in \mathsf{NP}$, dann existiert eine nichtdeterministische Turing-Maschine M, die L in Zeit $c \cdot n^k$ akzeptiert. w wird von M in $c \cdot |w|^k$ Schritten akzeptiert. Die Idee ist nun, eine MERF-Formel zu bauen, die erfüllbar ist, genau dann wenn M w in $c \cdot |w|^k$ Schritten akzeptiert. Man kodiere nun Konfigurationen durch Variablenfolgen und Rechenschritte durch Relationen zwischen diesen Variablen. Es sei $p = g(|w|)$, wir erhalten ein Eingabeband wie folgt:



Eingabeband

Als Konfigurationen hat man:

n	z	n_E	n_A	Arbeitsbandinhalt
0	Z^0	I^0	J^0	-p -1 0 1 2 p b b b b b b b A^0_{-p} A^0_{-1} A^0_0 A^0_1 A^0_2 A^0_p
1	Z^1	I^1	J^1	-p -1 0 1 2 p b b c b b b A^1_{-p} A^1_{-1} A^1_0 A^1_1 A^1_2 A^1_p
				⋮
$t-1$	Z^{t-1}	I^{t-1}	J^{t-1}	-p t-1 p A^{t-1}_{-p} A^{t-1}_j A^{t-1}_p
t	Z^t	I^t	J^t	-p t p A^t_{-p} A^t_j A^t_p
				⋮
p	Z^p	I^p	J^p	-p p A^p_{-p} A^p_p

Hierbei sei:

- n der jeweilige Schritt, z der aktuelle Zustand, n_E die Eingabekopfposition, n_A die Arbeitskopfposition,
- $A_i^t \in \Gamma$ der Inhalt der Arbeitsbandzelle i nach dem Schritt t. Der Wertebereich Γ enthält $(p+1) \cdot (2 \cdot p - 1)$ Variablen.
- $Z^t \in Q$ der Zustand von M nach dem Schritt t.
- $I^t \in \{-p, \ldots, +p\}$ die Eingabekopfposition.
- $J^t \in \{-p, \ldots, +p\}$ die Arbeitskopfposition.
- $E_i \in \Sigma \cup \{t\}$ der Inhalt von Eingabebandzelle i.

Wir überlegen uns, dass

$$F_w = \text{ Anfang } \wedge \text{ Übergänge } \wedge \text{ Ende}$$

mit

$$\begin{aligned}
\text{Anfang} &= \big(Z^0 = s\big) \wedge \big(I^0 = 0\big) \wedge \big(J^0 = 0\big) \\
&\quad \wedge \big((E_{-p} = \text{ћ}) \wedge \ldots \wedge (E_{-1} = \text{ћ}) \wedge (E_0 = w_0) \wedge \ldots \wedge \\
&\qquad (E_{k-1} = w_{k-1}) \wedge (E_k = \text{ћ}) \wedge \ldots \wedge (E_p = \text{ћ})\big) \\
&\quad \wedge \big(\big(A^0_{-p} = \text{ћ}\big) \wedge \ldots \wedge \big(A^0_p = \text{ћ}\big)\big)\,, \\
\text{Übergänge} &= \bigwedge_{1 \leqslant t \leqslant p} \ddot{U}^t, \text{ wobei} \\
\ddot{U}^t &= \bigvee_{\substack{-p \leqslant i \leqslant p \\ -p \leqslant j \leqslant p}} \ddot{U}^t_{ij} \text{ und} \\
\ddot{U}^t_{ij} &= \big(I^{t-1} = i\big) \wedge \big(J^{t-1} = j\big) \wedge \bigwedge_{\substack{-p \leqslant l \leqslant p \\ l \neq j}} \big(A^t_l = A^{t-1}_l\big) \\
&\quad \wedge \big(\big(Z^{t-1} = q\big) \wedge (E_i = a) \wedge \big(A^{t-1}_j = B\big) \wedge \\
&\qquad \big(Z^t = q'\big) \wedge \big(I^t = i + \lambda\big) \wedge \big(A^t_j = B'\big) \wedge \big(J^t = j + \mu\big)\big)
\end{aligned}$$

(da $(q, a, B, q', \lambda, B', \mu) \in \Delta$) und schließlich

$$\text{Ende} \approx \bigvee_{f \in F} (Z^p = f)\,.$$

Ohne Beschränkung der Allgemeinheit gelte

$$\forall f \in F, \forall a \in \Sigma, \forall A \in \Gamma : (f, a, A, f, 0, A, 0) \in \Delta,$$

wir erlauben also explizit „leere“ Schritte der Maschine.

7.8.6 Beispiel 3

Wir betrachten jetzt die Sprache der simplen mehrwertigen Erfüllbarkeit (SMERF). SMERF ist wie MERF, in der Formel sind aber nur Terme der Form $Y_i = a$ erlaubt. Nicht erlaubt sind somit $Y_i \neq a, Y_i = Y_j, Y_i \neq Y_j$.

7.8.7 Behauptung

SMERF ist NP-vollständig.

Beweis

Übung.

7.8.8 Beispiel 4 und Definition

Wir betrachten nun das Problem der bool'schen Erfüllbarkeit, modelliert durch die Sprache (SAT). Gegeben sei eine Formel F in bool'schen Variablen, wir erlauben also hier nur noch 2 Werte: wahr oder falsch.

Beispiel

$$\left(X_1 \vee X_2 \vee \overline{X_3}\right) \wedge \left(X_3 \vee \overline{X_4}\right)$$

Es stellt sich die Frage, ob F erfüllbar ist.

7.8.9 Behauptung

SAT ist NP-vollständig.

Beweis

i. SAT $\in$ NP: Man rate und verifiziere.

ii. SMERF $\preccurlyeq_P$ SAT: Wir suchen eine Funktion, die in polynomieller Zeit aus einer SMERF-Instanz $cod\,(Y_1) \ldots cod\,(Y_n)\,\$cod\,(B_1) \ldots cod\,(B_n)\,\$cod\,(F)$ eine SAT-Formel F berechnet mit

$$F' \text{ erfüllbar } \Leftrightarrow F \text{ erfüllbar.}$$

Man verwende folgende Variablen:

$$X_{ib}, \text{ wobei } 1 \leqslant i \leqslant n, \forall b \in B \text{ und } X_{ib} \text{ kodiert } Y_i = b.$$

Beispiel

$$(Y_1 = a \vee Y_2 = b) \wedge (Y_1 = b \vee Y_2 = c) \mapsto (X_{1a} \vee X_{2b}) \wedge (X_{1b} \vee X_{2c})$$

Bilde aus F' eine neue Formel F_1, in der Terme $Y_i = b$ durch X_{ib} ersetzt werden.

Beispiel

$$(Y_1 = a) \wedge (Y_1 = b) \mapsto X_{1a} \wedge X_{1b}$$

Wir müssen sicherstellen, dass für jedes $1 \leqslant i \leqslant n$ genau eines der X_{ib} mit $b \in B_i$ wahr ist. Das bedeutet, dass Y_i genau einen Wert aus B_i annimmt. Dazu führen wir neue Variablen ein:

$$D_i = \bigvee_{b \in B_i} \left(X_{ib} \wedge \bigwedge_{\substack{b' \in B_i \\ b' \neq b}} \overline{X_{ib'}} \right) \qquad (1 \leqslant i \leqslant n)$$

$$F_2 = \bigwedge_{1 \leqslant i \leqslant n} D_i$$

Zusammen:

$$F = F_1 \wedge F_2$$

Es gilt:

$$F' \text{ erfüllbar} \Leftrightarrow F \text{ erfüllbar}$$

denn:
Die Laufzeit ist proportional zu $|F| = |F_1| + |F_2|$, wobei die Größe der SMERF-Instanz durch g gegeben sei. Somit

$$\begin{aligned} |F_1| &\in \mathcal{O}(|F'|) \subseteq \mathcal{O}(g) \\ |D_i| &\leqslant |B_i|^2 \in \mathcal{O}(g^2) \\ |F_2| &\in \mathcal{O}(g^3) \end{aligned}$$

Zusammen erhalten wir polynomielle Laufzeit $\mathcal{O}(g^k)$

7.8.10 Beispiel 5

Wir betrachten die Sprache 3-SAT. Gegeben sei eine bool'sche Formel F in konjunktiver Normalform (KNF):

$$\begin{aligned} F &= K_1 \wedge \ldots \wedge K_i \wedge \ldots \wedge K_m \\ \text{mit } K_{i_j} &= L_{i_1} \vee \ldots \vee L_{i_j} \vee \ldots \vee L_{i_t} \\ \text{und } L_{i_j} &= X_j \text{ oder } \overline{X_j} \end{aligned}$$

mit weniger als 3 Literalen L_{i_j} pro Klausel K_i.

7.8.11 Behauptung

3-SAT ist NP-vollständig.

Beweis

Wir müssen zeigen, dass

i. 3-SAT $\in$ NP und

ii. SAT $\preccurlyeq_P$ 3-SAT

Dabei gehen wir wie folgt vor:

i. Hier rate man die Belegung und verifiziere.

ii. Wir brauchen eine SAT-Formel F, die sich in einer Transformation polynomieller Zeit in eine 3-SAT-Formel F' übersetzen lässt mit

$$F' \text{ erfüllbar } \Leftrightarrow F \text{ erfüllbar.}$$

Die SAT-Formel F sei vollständig geklammert gegeben, also z.B.

$$F = (\neg (x_1 \vee \neg (x_2 \vee \neg x_3)) \wedge x_2) .$$

Wir stellen F durch einen Ausdrucksbaum dar:

[Skizze wird nachgereicht]

Durch Anwendung der de-Morgan'schen Regeln kann man alle Negationen zu den Blättern schieben:

[Skizze wird nachgereicht]

Wir führen für jeden inneren Knoten (Operatorenknoten) neue Variablen ein. Diese Variable soll Wert des Teilbaums (also Teilausdrucks) darstellen:

$$(Z_2 \Leftrightarrow x_2 \vee \neg x_3) \wedge (Z_1 \Leftrightarrow \neg x_1 \vee \neg Z_2) \wedge (Z_{root} \Leftrightarrow Z_1 \vee \neg x_2) \wedge Z_{root}$$

Hierbei sind die Regeln:

- Die $\wedge$-Knoten sind mit Z_i und Kindern markiert: W_j, W_l mit

  $$W_j, W_l \in \{Z_i\}_j \cup \{x_j\}_j \cup \{\overline{x_j}\}\,.$$

  Man erzeuge nun den Term:

  $$Z_i \Leftrightarrow W_j \wedge W_l$$

- Damit gilt für die $\vee$-Knoten:

  $$Z_i \Leftrightarrow W_j \vee W_l$$

  Verwende alle diese Terme mit Z_{root} und wandle die Terme in 3-KNF Form um:

  - $A \Leftrightarrow B$:

    $$\begin{aligned} & A \Leftrightarrow B \\ \equiv\ & (A \wedge\ \ B) \vee (\neg A \wedge \neg B) \\ \equiv\ & \underbrace{(A \vee \neg A)}_{T} \wedge (A \vee \neg B) \wedge (B \vee \neg A) \wedge \underbrace{(B \vee \neg B)}_{T} \\ \equiv\ & (A \vee \neg B) \wedge (B \vee \neg A) \end{aligned}$$

  - $Z_i \Leftrightarrow W_j \vee W_i$:

    $$\begin{aligned} & Z_i \Leftrightarrow W_j \vee W_l \\ \equiv\ & \Big(A \vee \neg\Big(\overline{W_j} \underset{\wedge}{\vee} \overline{W_l}\Big)\Big) \wedge (W_j \vee W_l \vee \neg Z_i) \\ \equiv\ & \big(A \vee \overline{W_j}\big) \wedge \big(A \vee \overline{W_l}\big) \wedge \big(W_j \vee W_l \vee \overline{Z_i}\big) \end{aligned}$$

  - Analog für $Z_i \Leftrightarrow W_j \wedge W_l$.

Wir kehren zurück zum CLIQUE-Problem 7.5.

7.8.12 Erinnerung: Das CLIQUE-Problem

Gegeben sei ein ungerichteter Graph $G = (V, E)$ mit $C \in \mathbb{N}$. Wir suchen noch immer die Antwort auf die Frage, ob in G eine Clique W der Größe $|W| = C$ existiert (vergleiche 7.5.2, V1 sowie 7.5.4).

7.8.13 Behauptung

CLIQUE ist NP-vollständig.

Beweis

i. CLIQUE $\in$ NP ✓
(Man errate einfach C Knoten und verifiziere, dass alle zueinander benachbart sind.)

ii. 3-SAT $\preccurlyeq_P$ CLIQUE.
Von 3-SAT wissen wir, dass das Problem NP-vollständig ist. Wir brauchen hierzu eine Transformation polynomieller Zeit von einer 3-SAT-Formel F in einen Graphen G sowie eine Konstante C mit

$$F \text{ erfüllbar} \Leftrightarrow G \text{ hat eine Clique der Größe } C.$$

F ist von der Form

$$F = (z_{11} \vee z_{12} \vee z_{13}) \wedge (z_{21} \vee z_{22} \vee z_{23}) \wedge \ldots \wedge (z_{m1} \vee z_{m2} \vee z_{m3}),$$

wobei

- m = Anzahl der Klauseln,
- z_{ij} = Literale der Form x_k oder $\overline{x_k}$ mit
- $x_1 \ldots x_n$ = Variablen

Wir erhalten für $G = (V, E)$:

- $V = \{z_{ij} | 1 \leqslant i \leqslant m, 1 \leqslant j \leqslant 3\}$, wir nehmen also ohne Beschränkung der Allgemeinheit an, dass jede Klausel genau 3 Literale hat, sowie
- $E = \{\{z_{ij}, z_{kl}\} \mid i \neq k,\ z_{ij}$ und z_{kl} können gleichzeitig wahr werden, d.h. $z_{ij} \neq \overline{z_{kl}}\}$.

Genau dann, wenn der Graph G eine Clique der Größe $C = m$ hat gilt, dass F erfüllbar ist. Da insbesondere die Erzeugung des Graphen in polynomieller Zeit erfolgen kann und der Graph polynomiell Speicherplatz belegt, handelt es sich hierbei um eine zulässige polynomielle Reduktion.

7.9 (Gerichteter) Hamiltonscher Kreis und HaKr-Problem

7.9.1 Definition

Es sei ein gerichteter Graph $G = (V, E)$ gegeben. Man möchte nun wissen, ob es einen Hamiltonschen Kreis gibt, ob also eine Aufzählung

$$v_0, v_1, \ldots, v_{n-1} \text{ von } V, \quad (n = |V|)$$

gibt, sodass

$$[v_i, v_{i+1}\rangle \in E,\ \forall i, 0 \leqslant i \leqslant n.$$

Alle i sind hierbei Indizes, in modulo n angegeben. Dieses Problem bezeichnet man als HaKr.

7.9.2 Behauptung

HaKr ist NP-vollständig.

Beweis

i. HaKr $\in$ NP:
Wir erraten dazu die Rehenfolge $v_0, \ldots, v_{n-1}$ und verifizieren die Kanten.

ii. 3-SAT $\preccurlyeq_P$ HaKr:
Wir müssen zeigen, dass sich aus jeder 3-KNF-Formel F ein Graph G_F in polynomieller Zeit berechnen lässt, sodass gilt: Genau dann, wenn F erfüllbar ist, hat G einen Hamilton'schen Kreis.

Verfeinerung

Gegeben sei beispielsweise ein Baustein („gadget"):

[Skizze wird nachgereicht]

Dieser Baustein kann (modulo der Rotationssymmetrie) von einem Hamilton'schen Kreis nur folgendermaßen durchquert werden:

[Skizzen werden nachgereicht]

Der Pfad verlässt also diesen Baustein stets „auf der selben Höhe" wie beim Eingang. Man baue nun G aus F (ohne Beschränkung der Allgemeinheit habe F **genau** 3 Literale pro Klausel und bestehe aus n Variablen und m Klauseln) wie folgt:
Für jede Klausel verwende eine Kopie des Bausteins, benenne also die Knoten durch Literale, z.B. $x_4 \vee \overline{x_1} \vee x_7$.

[Skizze wird nachgereicht]

Jede Variable sei durch einen Knoten repräsentiert. Wir benennen diese Knoten mit den jeweiligen Variablennamen:

[Skizze wird nachgereicht]

Ein Pfad $\overset{x_1=true}{\rightsquigarrow}$ führt durch alle Bausteine, in denen Knotennamen x_1 in beliebiger Reihenfolge vorkommen.

[Skizze wird nachgereicht]

Ein Pfad $\overset{\overline{x_1}=true}{\rightarrow}$ führt durch alle Bausteine mit $\overline{x_1}$.
Analog kann man dies für alle x_i konstruieren.

7.9.3 Behauptung

Der im vorangegangenen Beweis erzeugte Graph hat genau dann einen Hamilton'schen Kreis, wenn F erfüllbar ist.

Beweis

„⇐“ Sei F erfüllbar. Man wähle eine erfüllende Wahrheitsbelegung für Variablen. Wenn $x_i = true$, verwende $x_i = true$ Pfad für den Hamilton'schen Kreis und durchquere jeden Baustein je nach der Anzahl der wahren Literale.

„⇒“ G habe einen Hamiltonschen Kreis. Je nachdem, ob dieser Kreis den x_i-Knoten über den „$x_i = true$“-Pfad verlässt oder nicht, setze x_i auf wahr. Somit erhält man eine erfüllende Wahrheitsbelegung, damit ist F erfüllbar.

7.10 SubsetSum

7.10.1 Definition

Es seien n Zahlen $A_1, \ldots, A_n$ sowie eine Zahl S jeweils in Dezimaldarstellung gegeben. Man möchte wissen, ob $\exists I \subset \{1, \ldots, n\}$, sodass $\sum_{i \in I} A_i = S$.

7.10.2 Behauptung

SubsetSum ist NP-vollständig.

Beweis

i. SubsetSum $\in$ NP:
Man errate I und verifiziere $\sum_{i \in I} A_i = S$.

ii. 3-SAT $\preccurlyeq_P$ SubsetSum:
Wir müssen aus einer 3-KNF-Formel F die Zahlen $A_1, \ldots, A_N, S$ in polynomieller Zeit generieren, sodass gilt:

$$F \text{ erfüllbar } \Leftrightarrow \exists I \in \{1, \ldots, N\} : \sum_{i \in I} A_i = S$$

	$(x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee \overline{x_3} \vee x_4) \wedge (x_2 \vee \overline{x_4} \vee x_1)$			x_1	x_2	x_3	x_4
x_1	1	0	1	1	0	0	0
$\overline{x_1}$	0	0	0	1	0	0	0
x_2	0	1	1	0	1	0	0
$\overline{x_2}$	1	0	0	0	1	0	0
x_3	1	0	0	0	0	1	0
$\overline{x_3}$	0	1	0	0	0	1	0
x_4	0	1	0	1	0	0	0
$\overline{x_4}$	0	0	1	0	0	0	1
S	3	3	3	1	1	1	1
	1	0	0	0	0	0	0
	1	0	0	0	0	0	0
	0	1	0	0	0	0	0
	0	1	0	0	0	0	0
	0	0	1	0	0	0	0
	0	0	1	0	0	0	0

7.11 Weitere NP-vollständige Probleme

Abschließend möchten wir noch einige NP-vollständige Probleme nennen. Diese werden hier nur angesprochen und in der Vorlesung „Komplexitätstheorie“ ausführlich behandelt.

- Gegeben sei ein Polygon. Man kann es stets so in Dreiecke zerlegen, dass die Ecken der Dreiecke ausschließlich auf den Ecken des Polygons liegen. Interessanterweise ist der analoge Fall im Dreidimensionalen, also der Versuch, Polyeder in Tetraeder aufzuteilen, sodass deren Ecken ausschließlich mit den Ecken des Polyeders zusammenfallen, nicht immer möglich. Das Problem festzustellen, ob bei gegebenem Körper eine solche Unterteilung durchgeführt werden kann, ist NP-vollständig.

- Gleiches trifft für die Fragestellung zu, ob ein gegebener NEA eine bestimmte Sprache L akzeptiert. Zwar ist naheliegend, den NEA zuerst in einen DEA zu überführen und diesen zu verifizieren, jedoch benötigt eben diese Umwandlung exponentielle Rechenzeit im Bezug auf die Anzahl der Knoten, was die Klassifikation gerechtfertigt.

- Gegeben seien Punkte im Dreidimensionalen samt ihrer Koordinaten. Das Problem, ob es ein Dreieck aufgespannt zwischen je dreien dieser Punkte gibt, dessen Kantenlänge höchstens k ist, ist ebenfalls NP-vollständig.