Theoretische Informatik - Zusammenfassung

Patrick Pletscher

1. Oktober 2004

1 Modelle von Berechnungen

Es gibt Modelle in welchen Berechnungen möglich sind und in anderen Modellen sind sie wiederum nicht möglich. Man spricht dann von *Berechnungsmodellen für spezielle Zwecke*. Kleine Änderungen im Berechnungsmodell können hier drastische Konsequenzen haben, so dass z.Bsp. etwas plötzlich möglich wird.
Es gibt aber auch *universelle Berechnungsmodelle*, Beispiele davon sind production systems, Turingmaschinen oder das Lambda-Kalkül. All diese unterschiedlichen Berechnungsmodelle sind äquivalent. Diese Modelle sind universell im Sinn dass sie alles berechnen können, was in anderen Modellen berechnet werden kann, wenn man annimmt, dass sie in Zeit und Speicher nicht eingeschränkt sind.

1.1 Sortiernetzwerke (sorting networks)

Netzwerke von Parallelen Drähten auf welchen Zahlen von links nach rechts reisen. An Stellen, die durch vertikale Linien bezeichnet sind, weist ein Vergleichgatter die kleinere Zahl zum oberen Ausgangsdraht, die grössere auf den unteren.

Theorem 1.1 (0-1 Prinzip). *Wenn ein Netzwerk S mit n Eingangslinien alle 2^n Vektoren von 0-en und 1-en in aufsteigender Reihenfolge sortiert, so sortiert S irgendeinen Vektor von n beliebigen Zahlen korrekt.*

Theorem 1.2. *(Testen beweist die Korrektheit!): Wenn ein Netzwerk S, welches nur Adjazenzvergleiche benutzt, den inversen Vektor $x_1 > x_2 > \ldots > x_n$, so sortiert er einen beliebigen Vektor.*

1.2 Threshold logic

Ein Thresholdgatter ist ein logisches Element mit mehreren Eingängen, welches durch einen Thresholdwert t charakterisiert wird. Es zählt die aktuellen Eingangswerte und produziert einen Ausgang von 1, falls die Summe grösser gleich t ist. Man benutzt auch noch errregende und hemmende Eingänge, eine Zelle feuert, falls mindestens t erregende Eingänge an sind und kein hemmender an ist.

1.3 Markovalgorithmen

Alphabet $A = \{0, 1\}$. Funktion $A^* \to A^*$. Markeralphabet $M = \{\alpha, \beta, \ldots\}$. Sequenz (geordnet) $P = P_1, P_2, \ldots$ von Umschreibungsregeln, welche von 2 Typen sind: $P_i = x \to y$ (weiterfahren) oder $P_i = x\neg y$ (terminiere), wobei $(A \cup M)^*$.
Ausführung: benutze die *erste Regel* welche auf den Datenstring angewendet werden kann, wende sie auf den linkesten Pattern match an. Eine Terminierungsregel stoppt den Prozess.

Beispiel 1.3.1 Hänge Suffix an
$P_1 : \alpha 0 \to 0\alpha, P_2 : \alpha 1 \to 1\alpha$
$P_3 : \alpha\neg 101$
$P_4 : \varepsilon \to \alpha$

Die erste Zeile könnte auch durch $\alpha B \to B\alpha$ ersetzt werden und wird Produktionsschema genannt.

Beispiel 1.3.2 Umkehren des Eingabestrings s

P_1	$\alpha\alpha\alpha \to \alpha\alpha$	zurückstauchen
P_2	$\alpha\alpha B \to B\alpha\alpha$	Auslöschen von einzelnen α
P_3	$\alpha\alpha\neg\varepsilon$	
P_4	$\alpha B'B'' \to B''\alpha B'$	umkehren
P_5	$\varepsilon \to \alpha$	repetiert ausgeführt, aber P_2 und P_3 stoppen P_5 vor dem Erstellen von 2 aufeinanderfolgenden α

Algorithmusdesign

1. Initialisierung. Durch einen Marker die Stelle markieren, wo etwas geschehen soll.

2. Das Durchführen der Operationen wird meist durch das Suchen nach einem bestimmten Marker bewerkstelligt.

3. Aufräumen der Marker

2 Endliche Automaten

2.1 Definitionen

Notation

Alphabet $A = \{a, b, \ldots\}$ oder $A = \{0, 1, \ldots\}$.
$A^* = \{w, w', \ldots\}$ wobei w Worte sind.
Nullstring ε.
Leere Menge $\emptyset$.
"Sprache" $L \subseteq A^*$.
Menge $S = \{s_0, s_1, \ldots\}$, Kardinalität $|S|$, Potenzmenge 2^S.

Deterministische endliche Automaten DFA

$M = (S, A, f, s_0, \ldots)$. Eine Menge von Zuständen S, Alphabet A, Übergangsfunktion $f : S \times A \to S$, Anfangszustand s_0. Andere Komponenten von M bezeichnet mit $\ldots$ können varieren entsprechend dem Zweck von M.

Beispiel 2.1.1 Mod 3 Teiler
Lese eine binär Zahl von links nach rechts, also MSB zuerst, und berechne seinen Rest mod 3.

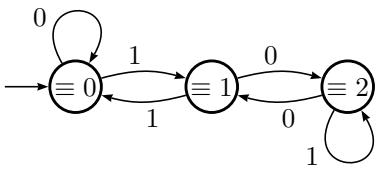


Abbildung 2.1: *mod 3 Teiler*

Acceptor

$M = (S, A, f, s_0, F)$, wobei $F \subseteq S$ eine Menge von Zuständen, die akzeptierende Zustände sind.

Um die Akzeptanz zu definieren, erweiteren wir f von $S \times A \to S$ zu $f : S \times A^* \to S$ wie folgt: $f(s, \varepsilon) = s$, $f(s, wa) = f(f(s, w), a)$ für $w \in A^*$.

Definition. M akzeptiert $w \in A^*$ wenn $f(s_0, w) \in F$. Setze $L \subseteq A^*$ akzeptiert von M: $L(M) = \{w | f(s_0, w) \in F\}$

Transducer

$M = \{S, A, f, g, s_0\}$, mit einer Funktion g welche einen Ausgabestring über einem Alphabet B produziert.

- $g : S \to B$ Moore Maschine (in Zustand)
- $h : S \times A \to B$ Mealy Maschine (bei Übergang)

Ein Acceptor ist ein Spezialfall eines Transducer.

Nicht-deterministische endliche Automaten (NFA) mit ε-Übergang: $f : S \times (A \cup \{\varepsilon\}) \to 2^S$

Eine nicht-deterministische Maschine bringt mehrere Kopien von sich hervor, wobei jede seinen eigenen root-to-leaf Pfad vom Baum aller möglichen Auswahlen verfolgt. Nicht-Determinismus führt also zu einem *exponentionellen Wachstum der Rechenleistung.*

Spezialfall: NFA ohne ε-Übergang: $f : S \times A \to 2^S$.

Erweitere $f : S \times A^* \to 2^S$: $f(s, \varepsilon) = \varepsilon$-Hülle von s = Alle Zustände erreichbar von s mit ε-Übergängen (s inklusive).

$f(s, wa) = \cup f(s', a)$ für $s' \in f(s, w)$.

Erweitere f weiter $f : 2^S \times A^* \to 2^S$ wie folgt: $f(s_1, \ldots, s_k, a) = \cup f(s_i, a)$ für $i = 1, \ldots, k$.

Definition. M akzeptiert $w \in A^*$ wenn $f(s_0, w) \cap F \neq \emptyset$. w wird akzeptiert wenn $\exists$ ein w-Pfad von s_0 zu F.
Setze $L \subseteq A^*$ akzeptiert von M: $L(M) = \{w | f(s_0, w) \cap F \neq \emptyset\}$

2.2 Äquivalenz von NFA und DFA

Definition. Zwei FAs sind *äquivalent* genau dann, wenn sie die gleiche Sprache akzeptieren.

Lemma 2.1 (ε-Übergänge). *Jede NFA mit ε-Übergängen kann in eine äquivalente NFA M ohne ε-Übergängen umgewandelt werden.*

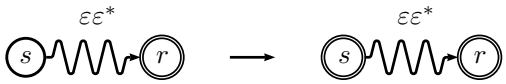
Man kann jede NFA durch eine DFA ersetzen dies kann aber zu einem *exponentionellen Wachstum der Anzahl der Zustände* führen.

Theorem 2.1 (Äquivalenz NFA-DFA). *Jede NFA N kann in eine äquivalente DFA M umgewandelt werden.*

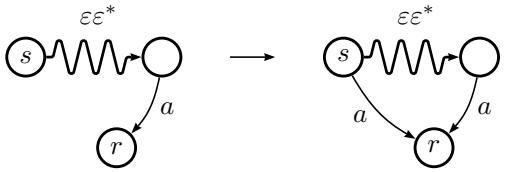
Entfernung von ε-Übergängen

Für alle Zustände s in der Zustandsmenge ...

- betrachte alle Pfade der Form $\varepsilon\varepsilon^*$, welche s mit einem akzeptierenden Zustand r verbinden und keine ε- Zyklen enthalten. Wenn ein solcher Pfad existiert, so wird s als akzeptierend gesetzt.



- betrachte für einen Buchstaben $a \in A$ jeden Pfad von der Form $\varepsilon\varepsilon^* a$ welcher s mit einem anderen Zustand r verbindet und keine ε-Zyklen enthält. Für jeden von diesen füge den Übergang $f(s, a) = r$ hinzu.



Am Ende kann man alle ε-Übergänge entfernen.

Umwandlung NFA in DFA - Der Subset Construction Algorithmus

1. Erstelle den Startzustand des DFA in dem man die ε-Hülle des Startzustands des NFA nimmt.
2. Führe folgendes für den neuen DFA Zustand aus:
   Für jedes mögliche Eingabesymbol:
   a) Suche alle Zustände welche vom neu-erstellten Zustand aus mit dem Eingabesymbol erreicht werden können; dies gibt eine Menge von Zuständen zurück.
   b) Finde die ε- Hülle dieser Menge von Zuständen, dies kann möglicherweise in einer neuen Menge enden.

Diese Menge von NFA Zuständen ist ein einziger Zustand in der DFA.

3. Jedes Mal wenn wir einen neuen DFA Zustand erstellen, müssen wir Schritt 2 auf ihn anwenden. Dieser Prozess ist komplett, wenn die Anwendung von Schritt 2 zu keinem neuen Zustand führt.

4. Die Endzustände des DFA sind diese, welche irgendeinen der Endzustände der NFA enthalten.

5. Führe zusätzlich einen Trap- Zustand ein, inden man kommt, sofern man in einem beliebigen DFA Zustand auf ein Zeichen trifft, für welches keine Übergangsrelation vorhanden ist. Für alle Zeichen bleibt man in diesem Trap- Zustand.

3 Reguläre Ausdrücke: Theorie

Alphabet A, reguläre Ausdrücke $\mathcal{R}(A)$. $\mathcal{R}(A)$ wird wie folgt erhalten.

3.1 Syntax

Primitive

$\varepsilon, \emptyset, a$ für jedes $a \in A$

Zusammengesetze

wenn E', E'' REs sind, so sind es auch $(E' \cup E'')$, $(E' \circ E'')$, $(E')^*$.

Priorität

Die Prioritäten der Operatoren sind wie folgt:

$$* \succ \circ \succ \cup$$

3.2 Semantik

jede RE definiert eine Sprache $L \subseteq A^*$.

Theorem 3.1. *Eine Menge L ist regulär wenn L durch einen regulären Ausdruck beschrieben wird.*

ε bezeichnet die Sprache die nur aus dem Nullstring besteht.
$\emptyset$ bezeichnet $\{\}$
0 bezeichnet $\{0\}$ usw. für andere Elemente von A.

Wenn E', E'' die Sprachen L', L'' bezeichnen, so gilt:

- $(E' \cup E'')$ bezeichnet $L' \cup L''$
- $(E' \circ E'')$ bezeichnet $\{w \in A^* | w = w'w'',\ w' \in L',\ w'' \in L''\}$
- $(E')^*$ bezeichnet $\{w \in A^* | w = w_1 w_2 \dots w_l,\ w_i \in L',\ l \geq 0\}$

3.3 Eigenschaften von REs

Definition. Eine Sprache (oder Menge) $L \subseteq A^*$ wird als regulär bezeichnet gdw. L durch eine FA akzeptiert wird.

Theorem 3.2. *Wenn $L, L' \subseteq A^*$ reguläre Mengen sind, so sind es auch $L \cup L'$, $L \circ L'$ und L^*.*

Theorem 3.3. *Wenn L regulär ist, so ist auch das Komplement $\neg L$ regulär.*

3.4 NFA $\geq$ RE

Theorem 3.4. *L definiert durch RE, so existiert eine NFA N für welche gilt $L = \mathcal{L}(N)$.*

3.5 DFA $\leq$ RE

Boolsche Matrix Multiplikation

Transitive Closure einer Adjazenzmatrix A.

$$Z_{ik} = \bigvee_{j=1\dots n} X_{ij} \wedge Y_{jk}$$

Hat Laufzeit $O(n^3 \log n)$, da man nur 2-er Potenzen von Adjazenzmatrix A betrachten kann.

Warshall Algorithmus

Verbesserung des Algorithmus für Transitive Closure

$$B_{ij}^k = B_{ij}^{k-1} \vee B_{ik}^{k-1} \wedge B_{kj}^{k-1}$$

Hat Laufzeit $O(n^3)$.

Anwendung von Warshall auf RE

R_{ij}^n: Regulärer Ausdruck definiert die Sprache von $\forall$ Strings welche $M = (Q, A, f, q_0, F)$ von **i** nach **j** überführt.

Es gilt also

$$R_{ij} = \begin{cases} a' \cup a'' \cup \dots a^{(n)} & \text{für alle } a^{(n)} \text{ mit } f(i, a^{(n)}) = j \\ \emptyset & \text{sonst} \end{cases}$$

Dann gilt folgende Formel

$$R_{ij}^k = R_{ij}^{k-1} \cup R_{ik}^{k-1} \circ (R_{kk}^{k-1})^* \circ R_{kj}^{k-1}$$

Für die Initialisierung:

$$\begin{aligned} R_{ij}^0 &= \{a | f(s_i, a) = s_j\} \text{ für } i \neq j \\ R_{ii}^0 &= \{a | f(s_i, a) = s_i\} \cup \{\varepsilon\} \end{aligned}$$

Für die Umwandlung einer FA in eine RE geht man wie folgt vor:

1. Schreibe in einer Tabelle alle möglichen direkten Übergänge (also $k = 0$) auf. Für Loop-Übergänge muss man auch ε als möglichen Übergang eintragen.

2. Erhöhe k um eins und benutze obige Formel um das Problem für das grössere k auf die vorherige Tabelle zurückzuführen.

3. Wiederhole Schritt 2 solange, bis $k = n = |Q|$. Wenn fertig, dann lese in Zeile von Startzustand all die REs für akzeptierende Zustände ab und verodere sie.

Beispiel 3.5.1 Umwandlung FA in RE

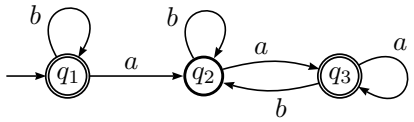


Abbildung 3.2: *Automat für Beispiel*

$k = 0$:

	1	2	3
1	$\varepsilon \mid b$	a	
2		$\varepsilon \mid b$	a
3		b	$\varepsilon \mid a$

$k = 1$:

	1	2	3
1	$(\varepsilon \mid b) \mid \underbrace{(\varepsilon \mid b)(\varepsilon \mid b)^*(\varepsilon \mid b)}_{=b^*}$	$(a) \mid \underbrace{(\varepsilon, b)(\varepsilon, b)^* a}_{=b^* a}$	
2		$\varepsilon \mid b$	a
3		b	$\varepsilon \mid a$

usw.

3.6 Zustandsminimierung

Definition. Zustände r und s von M sind äquivalent (nicht unterscheidbar) genau dann, wenn für alle $w \in A^*, f(r, w) \in F \Leftrightarrow f(s, w) \in F$.

Theorem 3.5. *Wenn r, s nicht unterscheidbar sind durch Worte w der Länge $|w| \leq n = |S|$, so sind r und s nicht unterscheidbar.*

Algorithmus für die Zustandsminimierung

In einer Tabelle von Zustandspaaren ...

1. Markiere alle Paare welche durch ε unterschieden werden können (einer akzeptierend, anderer Nicht-akzeptierend).
2. Für jedes unmarkierte Paar (r, s) $r, s \in S,\ r \neq s$ überprüfe für alle $a \in A$, ob das Paar $(f(r, a), f(s, a))$ als unterscheidbar markiert wurde. Wenn dies der Fall ist, markiere (r, s) unterscheidbar durch kleinsten Zeugen $w = aw'$, wobei w' vererbt ist von $(f(r, a), f(s, a))$
3. Wiederhole 2 $|Q|$ mal oder bis kein Paar in einem Durchgang markiert werden kann.

Beispiel 3.6.1 Zustandsminimierung

$|w| = 0$:

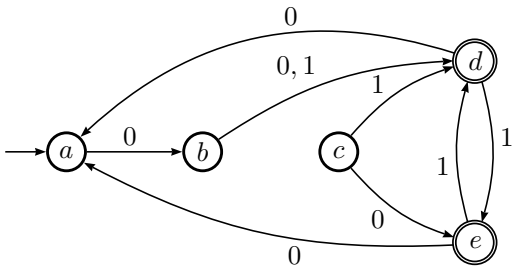


Abbildung 3.3: *Automat für Beispiel*

	b	c	d	e
a			ε	ε
b			ε	ε
c			ε	ε
d				

$|w| = 1$:

	b	c	d	e
a	0	0	ε	ε
b			ε	ε
c			ε	ε
d				

$|w| = 2$:

	b	c	d	e
a	0	0	ε	ε
b			ε	ε
c			ε	ε
d				

Es gilt also

$$b \equiv c, \quad d \equiv e$$

3.7 Pumping Lemma

Das Pumping Lemma ist eine präzise mathematische Aussage, dass ein FSM höchstens bis zu einer Konstante n zählen kann.

Lemma 3.1 (Pumping). *L regulär $\Rightarrow \exists n(L)$, so dass jedes $w \in L, |w| \geq n, w = xyz$.*

1. $|y| > 0$
2. $|xy| \leq n$
3. $xy^k z \in L$ *für* $k \geq 0$.

4 Endliche Automaten mit externem Speicher

Verschiedene Zugriffseinschränkungen führen zu verschieden Stärken der Automaten, es gilt:

$$\text{CA} < \text{DPDA} < \text{NPDA} < \text{LBA}$$

4.1 Konzepte, Konventionen und Notationen

FSM M kontrolliert Zugriff auf ein Eingabetape T (wenn es eines gibt) und ein Speichergerät S. Allgemeine Form eines Übergangs:
(momentaner Zustand von M, momentan untersuchtes Symbol von T, momentan untersuchtes Symbol von S) $\rightarrow$ (neuer Zustand von M, neu nach S geschriebenes Symbol, Bewegung des read/write Kopfs von S).

4.2 Counter Automat

Ein Counter Automat besteht aus einem fsm M, der durch ein Register (Counter) C erweitert wird, welches nur einen einzigen Integerwert von beliebiger Grösse speichern kann. C wird mit Null initialisiert. M kann C inkrementieren und dekrementieren und es auf 0 testen.

Durch die Einschränkung das man nur inkrementieren und dekrementieren kann, kann man nicht viel mehr als Zählen.

Beispiel 4.2.1 Counter Automat
Es soll ein Counter Automat für die Polnische Notation (suffix) die durch

$$G_s : S \rightarrow OSS|\neg S|x|y|z; O \rightarrow +|-|\cdot|/$$

definiert gefunden werden.

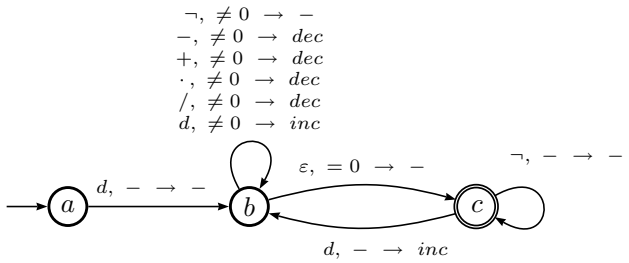


Abbildung 4.4: *Beispiel eines Counter Automat*

4.3 Deterministischer Pushdown Automat (DPDA)

PDA: FSM kontrolliert einen Stack. Unbeschränkter Speicherplatz, begrenzt auf last-in-first-out (LIFO) Zugriff.
Stackoperationen: push, pop, test empty.

Definition (DPA). $M = (Q, A, B, f, q_0, F)$ oder $M = (Q, A, B, f, q_0)$ in der Version "akzeptiere bei leerem Stack".
Dabei sind Q: Menge von Zuständen, q_0: Anfangszustand, $F \subseteq Q$: End oder akzeptierende Zustände.
A: Eingabealphabet, B: Stackalphabet (Konvention: enthält alle $a \in A$ und Boden des Stackssymbol $\not{\epsilon}$).
Übergangsfunktion $f : Q \times A_\varepsilon \times B_\varepsilon \rightarrow Q \times B^*$

Der LIFO Zugriff macht den PDA rechnerisch schwach. Nichtdeterminismus macht ihn stärker, aber entfernt den LIFO Flaschenhals nicht ganz.

Beispiel 4.3.1 Pushdown Automat
Es soll ein Pushdown Automat erstellt werden, welcher die Sprache $L(G)$ die durch die Produktionen

$$S \rightarrow SP|\varepsilon; P \rightarrow (S)|0$$

definiert ist, akzeptiert.

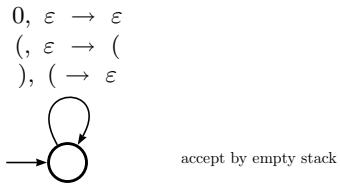


Abbildung 4.5: *Beispiel eines Pushdown Automat*

4.4 Nicht deterministischer Pushdown Automat (NPDA)

Definition (DPA). $M = (Q, A, B, f, q_0, F)$ oder $M = (Q, A, B, f, q_0)$ in der Version "akzeptiere bei leerem Stack".
Übergangsfunktion $f : Q \times A_\varepsilon \times B_\varepsilon \rightarrow 2^{Q \times B^*}$.

4.5 Linear beschränkter Automat (LBA)

Read/write Zugriff auf ein Tape T, von fixer Länge welche durch den Eingabestring w bestimmt ist.

4.6 Turing Maschine und Automaten von äquivalenter Stärke

Eine Turing Maschine ist eine FSM, welche ein Tape von unbeschränkter Grösse als externes Speichermedium benutzt. Es sind Lese-/Schreiboperation erlaubt. Dieses Maschine ist *universell*.

FSM mit 2-Stacks

Ist auch universell.

FSM mit Queue

Als Speichermedium dient eine Queue, also FIFO. Auch diese Maschine ist universell.

5 Kontextfreie Grammatiken (CFG) und Sprachen (CFL)

5.1 Grammatiken

$$G = (V, A, \mathcal{P}, S)$$

V: Menge der Nicht-Terminalsymbole

A: Menge der Terminalsymbole, Symbole die nicht mehr weiter ersetzt werden können

S: Startsymbol

$\mathcal{P}$: Menge der Produktionen der Form $L \to R$; $L, R \in (V \cup A)^*$

Im Unterschied zu Markov-Algorithmen können beliebige Regeln angewandt werden.

5.2 Typen von Sprachen (nach Chomsky)

Typ 0: keine Restriktion

Typ 1: Kontext sensitiv

$$\forall P : w_1 \to w_2 : |w_1| \leq |w_2|$$

Ausnahme: $S \to \varepsilon$ erlaubt, wenn S nie auf recher Seite.

Typ 2: Kontextfrei

$$w_1 \in V$$

Typ 3: Regulär

$$w_2 \in A \cup AV$$

kontextfrei vs. kontext sensitiv

kontextfrei: $P \supset A \to x$
A kann bedingungslos ersetzt werden.

kontextsensitiv: Kann Regeln der Form $uAv \to uxv$ enthalten.
A kann nur abhängig vom Kontext ersetzt werden.

5.3 Kontextfreie Grammatiken und Sprachen

Umschreibschritt

Für $u, v, x, y, y', z \in (V \cup A)^* : u \to v$, wenn $u = xyz, v = xy'z$ und $y \to y' \in \mathcal{P}$.

$\to^*$ ist der transitive, reflexive Abschluss von $\to$: $u \to^* v$ wenn $\exists w_0, w_1, \ldots, w_k$ mit $k \geq 0$ und $u = w_0, w_{j-1} \to w_j, w_k = v$.

kontextfreie Sprachen

$L(G)$ ist die freie Sprache generiert durch G: $L(G) = \{w \in A^* | S \to^* w\}$

5.4 Äquivalenz von CFGs und NPDAs

Theorem 5.1 (CFG ≡ NPDA). *$L \subseteq A^*$ ist eine CF genau dann, wenn $\exists$ NPDA M, welche L akzeptiert.*

5.5 Normalformen

Jede CFG kann in eine Zahl von "Normalformen" verwandelt werden, welche (fast) äquivalent sind. Äquivalent meint hier, dass die beiden Grammatiken dieselbe Sprache definieren; fast ist nötig, da diese Normalformen den Nullstring nicht generieren können.

Chomsky Normalform (rechte Seiten sind kurz)

Alle Regeln sind von der Form $X \to YZ$ oder $X \to a$, für nicht Terminale $X, Y, Z \in V$ und Terminale $a \in A$.

Theorem 5.2. *Jede CFG G kann in eine Chomsky Normalform G' transformiert werden, so dass $L(G') = L(G) - \{\varepsilon\}$.*

Greibach Normalform

Produziere bei jedem Schritt ein Terminalsymbol ganz links - nützlich für Parsing.

Alle Regeln sind von der Form $X \to aw$, für ein Terminal $a \in A$ und ein $w \in V^*$.

Theorem 5.3. *Jede CFG G kann in eine Greibach Normalform G' transformiert werden, so dass $L(G') = L(G) - \{\varepsilon\}$.*

5.6 Das Pumping Lemma für CFLs

Theorem 5.4. *Für jede CFL L gibt es eine Konstante n, so dass jedes $z \in L$ von der Länge $|z| \geq n$ als $z = uvwxy$ geschrieben werden kann, so dass folgendes gilt:*

1. *$vx \neq \varepsilon$*
2. *$|vwx| \leq n$*
3. *$uv^k wx^k y \in L$ für alle $k \geq 0$*

Theorem 5.5. *$L = \{0^k 1^k 2^k | k \geq 0\}$ ist nicht kontextfrei.*

5.7 Abschlusseigenschaften der Klasse von CFLs

Theorem 5.6. *Die Klasse von CFLs über dem Alphabet A ist geschlossen unter der regulären Operationen Vereinigung, Katenation und Kleene Star.*

Theorem 5.7. *Die Klasse von CFLs über dem Alphabet A ist nicht geschlossen unter Durchschnitt und Komplement.*

5.8 Das "Wort Problem". CFL Parsing in $O(n^3)$ durch dynamische Programmierung

Das Wort Problem fragt: Gegeben G und $w \in A^*$, entscheide ob $w \in L(G)$ oder nicht. Oder genauer: Gibt es einen Algorithmus welcher anwendbar ist auf irgendeine Grammatik G in einer gegebenen Klasse von Grammatiken, und ein $w \in A^*$, welcher entscheidet ob $w \in L(G)$?

Für CFGs gibt es einen "bottom up" Algorithmus (Cocke, Younger, Kasami) welcher systematisch alle möglichen Parsebäume von zusammenhängenenden Substrings vom zu Parsenden String w berechnet und läuft in $O(|w|^3)$.

6 Universelle Berechnungsmodelle und Äquivalenz

6.1 Übersicht

Es gilt:

$$\text{TM} \geq \text{MA} \geq \text{QM} \geq \text{TM}$$

TM: Turing Maschine
MA: Markov Algorithmus
QM: Queue

Turing Maschine

Daten können entlang einem Band ausgelegt werden, Labels dazu benutzt werden um die verschiedenen Teile ausfindig zu machen. Dazu kommt ein endlicher Automat Kontroller mit dem das Band scannen kann bis man auf ein bestimmtes Label trifft.

Post oder Queue Maschine

Ist beschränkt durch eine Zugriffsbeschränkung: Daten können nur vom Kopf gelesen und gelöscht werden und am Ende angehängt werden.

Markov Algorithmen

Ist strikt sequentiell. Die erste Umschreibregel die zutrifft, wird beim ersten Pattern Match angewandt. Die Übergänge sind von der Form: $q_i, x \to q_j, y$ mit $x, y \in A^*$

7 Berechenbarkeit

7.1 Definition

Definition (Berechenbarkeit). Eine (evtl. partielle) Funktion $f : \mathbb{N}^k \to \mathbb{N}$ ist *berechenbar*, falls es ein Rechenverfahren gibt, das f berechnet.

Verfahren ist z.B. durch Java-Programm P gegeben; wobei Ausführung ohne Platzbeschränkung. P gestartet mit $(n_1, \ldots, n_k) \in \mathbb{N}^k$, terminiert nach endlich vielen Schritten mit Ausgabe $f(n_1, \ldots, n_k)$. Im Falle einer *partiellen* Funktion muss P nicht (immer) stoppen.

7.2 Churchsche These

These 7.1 (Chursche). *Die durch die formale Definition der Turing-Berechenbarkeit (äquivalent: While, Goto, λ-Kalkül, ..-Berechnbarkeit) erfasste Klasse von Funktionen stimmt genau mit der Klasse der im intuitiven Sinne berechenbaren Funktionen überein.*

7.3 Turingmaschine

Definition (Turingmaschine). Eine Turingmaschine ist ein 7-Tupel

$$M = (Z, \Sigma, \Gamma, \delta, z_0, \Box, E)$$

Hierbei sind:

- Z die endliche *Zustandsmenge*
- Σ das *Eingabealphabet*
- $\Gamma \supset \Sigma$ das *Arbeitsalphabet*
- $\delta : Z \times \Gamma \to Z \times \Gamma \times \{L, R, N\}$ die *Überführungsfunktion*, im nichtdeterminstischen Fall:
  $\delta : Z \times \Gamma \to \mathcal{P}(Z \times \Gamma \times \{L, R, N\})$
- $z_0 \in Z$ der *Startzustand*
- $\Box \in \Gamma \setminus \Sigma$ das *Blank*
- $E \subseteq Z$ die Menge der *Endzustände*

Konfiguration

Eine *Konfiguration* formalisiert den globalen Zustand der Turingmaschine. Eine Konfiguration ist ein Wort $k \in \Gamma^* Z \Gamma^*$. $k = \alpha z \beta$ bedeutet, dass $\alpha\beta$ der nicht-leere Teil des Bandes ist. z ist der Zustand, in dem sich die Maschine gerade befindet und der Schreib-/Lesekopf steht auf dem ersten Zeichen von β.

Erreichbare Konfigurationen werden durch die binäre Relation $\vdash$ definiert. Es gilt

$a_1 \ldots a_m z b_1 \ldots b_n \vdash$

- $a_1 \ldots a_m z' c b_2 \ldots b_n$, falls $\delta(z, b_1) = (z', c, N), m \geq 0, n \geq 1$.
- $a_1 \ldots a_m c z' b_2 \ldots b_n$, falls $\delta(z, b_1) = (z', c, R), m \geq 0, n \geq 2$.
- $a_1 \ldots a_{m-1} z' a_m c b_2 \ldots b_n$, falls $\delta(z, b_1) = (z', c, L), m \geq 1, n \geq 1$.

Weiterhin gelten die zwei Sonderfälle

- $a_1 \ldots a_m z b_1 \vdash a_1 \ldots a_m c z' \, \Box$, falls $\delta(z, b_1) = (z', c, R)$
- $z b_1 \ldots b_n \vdash z' \, \Box \, c b_2 \ldots b_n$, falls $\delta(z, b_1) = (z', c, L)$

Eine TM kann sowohl Sprachen akzeptieren als auch Funktionen definieren.

Turing-Berechenbarkeit und Definierbarkeit

Definition (Sprachakzeptanz). Eine Turingmaschine *akzeptiert* die Sprache

$$T(M) = \{x \in \Sigma^* | z_0 x \vdash^* \alpha z \beta; \ \alpha, \beta \in \Gamma^*; \ z \in E\}$$

Definition (Turingberechenbarkeit). Eine Funktion $f : N^k \to N$ heisst *Turing-berechenbar*, falls es eine Turingmaschine M gibt, so dass für alle $n_1, \ldots, n_k, m \in N$ gilt

$$f(n_1, \ldots, n_k) = m \text{ gdw } z_0 \overline{n_1} \# \overline{n_2} \# \ldots \# \overline{n_k} \vdash^* \Box \ldots \Box \, z_e \overline{m} \, \Box \ldots \Box$$

wobei $z_0 \in E$ und $\overline{n}$ die (z.B. Binär-)Darstellung der Zahl n darstellt. # ist hierbei ein Trennzeichen.

Definition (Turingberechenbarkeit). Eine Funktion $f : \Sigma^* \to \Sigma^*$ heisst *Turing-berechenbar*, falls es eine Turingmaschine M gibt, so dass für alle $x, y \in \Sigma^*$ gilt:

$$f(x) = y \text{ gdw } z_0 x \vdash^* \Box \ldots \Box \, z_e y \, \Box \ldots \Box$$

Charakteristische Funktion

Man sagt, eine Sprache A ist von Typ 0, wenn sie von einer Turingmaschine M_A akzeptiert wird. Diese entspricht einer Turingmaschine, die die folgende Funktion $\chi_A : \Sigma^* \to \{0, 1\}$ berechnet

$$\chi_A(w) = \begin{cases} 1 & w \in A \\ \text{undefiniert} & w \notin A \end{cases}$$

M_A kann leicht umgebaut werden, um χ_A zu berechnen. Daher stimmen die Typ 0-Sprachen genau mit den *semi-entscheidbaren Sprachen überein.*

Mehrband-Turingmaschinen

Eine Mehrband-Turingmaschine kann auf $k \geq 1$ vielen Bänden unabhängig voneinander operieren. D.h. k Schreib-Leseköpfe, und δ ist eine Funktion von $Z \times \Gamma^k$ nach $Z \times \Gamma^k \times \{L, R, N\}$.

Satz 7.1 (Berechnungskraft von Mehrband- Turingmaschinen). *Zu jeder Mehrband- Turingmaschine M gibt es eine (Einband-)Turingmaschine M' mit $T(M) = T(M')$ bzw. so, dass M' dieselbe Funktion berechnet wie M.*

Ausdrucksmächtigkeit

Seien etwa $M_i = (Z_i, \Sigma, \Gamma_i, \delta_i, z_i, \ \square, E_i), i = 1, 2$. Wir bezeichnen *die sequentielle Komposition von M_1 und M_2* durch

$$\mathsf{start} \longrightarrow M_1 \longrightarrow M_2 \longrightarrow \mathsf{stop}$$

oder durch $M_1; M_2$.

Dieses Hintereinanderschalten wird definiert durch

$$M = (Z_1 \cup Z_2, \Sigma, \Gamma_1 \cup \Gamma_2, \delta, z_1, \ \square, E_2)$$

wobei (oBdA) $Z_1 \cap Z_2 = \emptyset$ und

$$\delta = \delta_1 \cup \delta_2 \cup \{(z_e, a, z_2, a, N) | z_e \in E_1, a \in \Gamma_1\}$$

Ausdrucksmächtigkeit: Test auf 0

$Z = \{z_0, z_1, ja, nein\}$; Startzustand z_0, Endzustände sind ja und nein.

$$\begin{aligned} \delta(z_0, a) &= (nein, a, N) \text{ für } a \neq 0 \\ \delta(z_0, 0) &= (z_1, 0, R) \\ \delta(z_1, a) &= (nein, a, L) \text{ für } a \neq \square \\ \delta(z_1, \square) &= (ja, \square, L) \end{aligned}$$

Ausdrucksmächtigkeit: if-then-else

$$\begin{array}{lllll} \mathsf{start} & \longrightarrow & M & \xrightarrow{z_{e1}} & M_1 \longrightarrow \mathsf{stop} \\ & & \downarrow z_{e_2} & & \\ & & M_2 & & \\ & & \downarrow & & \\ & & \mathsf{stop} & & \end{array}$$

Bezeichnet eine TM, wobei vom Endzustand z_{e1} von M aus nach M_1 übergangen wird, und von z_{e2} aus nach M_2. Alternative Schreibweise:

$$\mathsf{if}\ M\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2$$

7.4 LOOP-Programme

Semantik und Syntax

Für die Berechnung einer k-stelligen Funktion gehen wir davon aus, dass diese mit den Startwerten $n_1, \ldots, n_k \in N$ in den Variablen $x_1, \ldots, x_k$ gestartet wird und alle anderen Variablen den Anfangswert 0 haben.

$$\mathsf{LOOP}\ x_i\ \mathsf{DO}\ P\ \mathsf{END};$$

P wird genau (der Anfangswert von) x_i-mal ausgeführt. Das Ändern des Variablenwertes x_i im Innern von P hat keinen Einfluss auf die Anzahl der Wiederholungen.

Wertezuweisungen der Form $x_i := x_j + c$ und $x_i := x_j - c$, wobei aber hier "Proper Subtraction", also falls $c > x_j$ so wird Resultat zu 0.

Definition. Eine Funktion $f : \mathbb{N}^k \to \mathbb{N}$ heisst *LOOP-berechenbar*, falls es ein LOOP-Programm P gibt, das f in dem Sinne berechnet, dass P, gestartet mit $n_1, \ldots, n_k$ in den Variablen $x_1, \ldots, x_k$ (und 0 in den restlichen Variablen) stoppt mit dem Wert $f(n_1, \ldots, n_k)$ in der Variablen x_0.

Alle LOOP-berechenbaren Funktionen sind *totale* Funktionen. Aber nicht alle totalen Funktionen sind LOOP-Berechenbar.

Simulation von $\mathsf{IF}\ x = 0\ \mathsf{THEN}\ A\ \mathsf{END}$:
$y := 1$
$\mathsf{LOOP}\ x\ \mathsf{DO}\ y := 0\ \mathsf{END};$
$\mathsf{LOOP}\ y\ \mathsf{DO}\ A\ \mathsf{END};$

7.5 WHILE-Programme

Semantik und Syntax

$$\mathsf{WHILE}\ x_i \neq 0\ \mathsf{DO}\ P\ \mathsf{END}$$

P wird solange wiederholt ausgeführt, wie der Wert von x_i ungleich Null ist. Im Unterschied zu LOOP-Programmen wird hier P im Normalfall x_i verändern. WHILE-Programme können LOOP-Programme simulieren.

Satz 7.2. *Turingmaschinen können WHILE-Programme simulieren.*

Normalform für WHILE-Programme

Korollar 7.1. *Jede WHILE-berechenbare Funktion kann durch ein WHILE-Programm mit nur einer WHILE-Schleife berechnet werden.*

7.6 GOTO-Programme

Definition. GOTO-Programme bestehen aus Sequenzen von Anweisungen A_i, die jeweils durch eine Marke M_i eingeleitet werden:

$$M_1 : \ A_1; \ M_2 : \ A_2; \ldots M_k : \ A_K$$

Wobei Anweisungen A_i sein können:

- $x_i := x_j + c$ oder $x_i := x_j - c$
- $\mathsf{GOTO}\ M_i$
- $\mathsf{IF}\ x_i = c\ \mathsf{THEN\ GOTO}\ M_j$
- HALT

Satz 7.3. *Jedes WHILE-Programm kann durch ein GOTO-Programm simuliert werden und umgekehrt.*

7.7 TMs und WHILE/GOTO-Programme

Satz 7.4. *TMs können durch GOTO-Programme simuliert werden.*

Zusammenfassend gilt somit:

$$\mathsf{TM} \leq \mathsf{GOTO} \leq \mathsf{WHILE} \leq \mathsf{TM}$$

7.8 Funktionen: Grundlagen

Sei F_k die Menge aller Funktionen $f : \mathbb{N}^k \to \mathbb{N}$
Sei T_k die Menge aller *totalen* Funktionen $f : \mathbb{N}^k \to \mathbb{N}$
Sei

$$\begin{aligned} F &= \bigcup_{k\in\mathbb{N}} F_k \\ T &= \bigcup_{k\in\mathbb{N}} T_k \end{aligned}$$

7.9 Basis-Funktionen

Null-Funktion

Mit der Null-Funktion bezeichnen wir die Funktion $zero \in T_0$

$$zero = 0 \qquad (x \in \mathbb{N})$$

Successor-Funktion

Mit der Sucessor-Funktion bzeichnen wir die Funktion $s(x) \in T_1$

$$s(x) = x + 1 \qquad (x \in \mathbb{N})$$

Projektions-Funktion

Für alle $j, k \in \mathbb{N}^+, 1 \geq j \geq k$ bezeichnen wir die Projektions-Funktion $\pi^k_j \in T_k$

$$\pi^k_j(\mathbf{x}) = x_j \qquad (\mathbf{x} = (x_1, x_2, \ldots, x_k) \in \mathbb{N}^k)$$

7.10 Komposition/Primitive Rekursion

Komposition

Sei $k, l \in \mathbb{Z}^+$ und $\mathbf{f} = (f_1, \ldots, f_l)$ ein l-Tupel von Funktionen in F_k, und sei g eine Funktion in F_l. Sei $h = g \cdot \mathbf{f} = g \cdot (f_1(\mathbf{x}), \ldots, f_l(\mathbf{x}))$, die Funktion in F_k definiert wie folgt:

$$h(\mathbf{x}) = g(\mathbf{f}(\mathbf{x})) = g(f_1(\mathbf{x}), \ldots, f_l(\mathbf{x}))$$

Primitive Rekursion

Sei $k \in \mathbb{N}, f \in T_k$ und $h \in T_{k+2}$. Wir bauen eine neue Funktion g wie folgt:

$$\begin{aligned} g(0, \mathbf{x}) &= f(\mathbf{x}) && (\mathbf{x} \in \mathbb{N}^k) \\ g(y+1, \mathbf{x}) &= h(y, \mathbf{x}, g(y, \mathbf{x})) && (\mathbf{x} \in N^k, y \in N) \end{aligned}$$

Wobei $g \in T_{k+1}$.

7.11 μ-Minimierung

Sei $k \in \mathbb{N}$ und $f \in F_{k+1}$. Wir bauen eine neue Funktion $g = \mu f \in F_k$ wie folgt (für $\mathbf{x} \in \mathbb{N}^k$)

$$\begin{aligned} g(\mathbf{x}) = \min\{n | f(n, \mathbf{x}) = 0 \text{ und für alle } m < n \\ \text{ist } f(m, \mathbf{x}) \text{ definiert}\} \end{aligned}$$

Hierbei wird $\min \emptyset =$ undefiniert gesetzt. Durch Anwenden des μ-Operators können partielle Funktionen entstehen.

7.12 Definitionen

Definition. Die Klasse von *primitiv rekursiven (PR) Funktionen* ist die kleinste Teilmenge von F, die unter Basis-Funktionen, Komposition und Rekursion abgeschlossen ist.

Definition. Die Klasse von *(μ-)rekursiven Funktionen* ist die kleinste Teilmenge von F, die unter Basis-Funktionen, Komposition, Rekursion und *μ-Minimierung* abgeschlossen ist.

7.13 Beispiele

$$\begin{aligned} add(0, x) &= \pi^1_1(x) \\ add(y+1, x) &= h(y, x, add(y, x)) \end{aligned}$$

wobei h:

$$h(x, y, z) = s(\pi^3_3(x, y, z)) = s(z)$$

7.14 PR versus LOOP

Tupel kodieren

$$c(x, y) = \binom{x + y + 1}{2} + x$$

Stellt eine Funktion von $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ dar. Somit kann man mit c $k + 1$ Tupeln kodieren.

$$\langle n_0, n_1, \ldots, n_k \rangle = c(n_0, c(n_1, \ldots, c(n_k, 0)))$$

Für das Dekodieren benötigen wir e und f die Umkehrfunktionen von c

$$e(c(x, y)) = x, \qquad f(c(x, y)) = y$$

PR $\Leftrightarrow$ LOOP

Satz 7.5. *Die Klasse der PR-Funktionen stimmt genau mit der Klasse der LOOP-berechenbaren Funktionen überein.*

7.15 μ-Rekursion $\Leftrightarrow$ WHILE-Programme

μ ergibt eine echte Erweiterung der PR-Funktionen.

Satz 7.6. *Die Klasse der μ-rekursiven Funktionen stimmt genau mit der Klasse der WHILE- (GOTO-, Turing-) berechenbaren Funktionen überein.*

Satz 7.7 (Kleene). *Für jede μ-rekursive Funktionen f gibt es zwei $(n+1)$-stellige PR-Funktionen p und q, so dass sich f darstellen lässt als*

$$f(\overline{\mathbf{x}}) = p(\mu q(\overline{\mathbf{x}}), \overline{\mathbf{x}})$$

Hierbei ist μq die durch Anwendung des μ-Operators auf q entstehende (n-stellige) Funktion.

8 Nicht-Berechenbarkeit

8.1 Reduktion

Technik um Nicht-Berechenbarkeit zu zeigen. Damit zeigt man, dass ein Problem mindestens so schwer ist wie ein anderes.

Notation: $A \leq B$

Bedeutung: Falls B berechenbar ist, dann ist es A auch. Falls A nicht berechenbar ist, dann ist es B auch nicht.

8.2 Entscheidbarkeit

Definition. Eine Menge $A \subseteq \Sigma^*$ heisst *entscheidbar*, falls die *charakteristische Funktion* von A, nämlich $\chi_A : \Sigma^* \to \{0,1\}$, berechenbar ist.

$$\chi_A(w) = \begin{cases} 1, & \text{falls } w \in A \\ 0, & \text{falls } w \notin A \end{cases}$$

Definition. Eine Menge $A \subseteq \Sigma^*$ heisst *semi-entscheidbar*, falls die "halbe" charakteristische Funktion von A, nämlich $\chi'_A : \Sigma^* \to \{0,1\}$, berechenbar ist.

$$\chi_A(w) = \begin{cases} 1, & \text{falls } w \in A \\ \text{undefiniert}, & \text{falls } w \notin A \end{cases}$$

Satz 8.1. *Eine Sprache A ist entscheidbar gdw sowohl A als auch $\overline{A}$ semi-entscheidbar sind.*

8.3 Rekursive Abzählbarkeit

Definition. Eine Sprache $A \subseteq \Sigma^*$ heisst *rekursiv aufzählbar*, falls $A = \emptyset$ oder falls es eine totale und berechenbare Funktion $f : \mathbb{N} \to \Sigma^*$ gibt, so dass

$$A = \{f(0), f(1), f(2), \ldots\}$$

Nebenbemerkung: $f(i)$ darf gleich $f(j)$ sein.

Satz 8.2. *Eine Sprache ist rekursiv aufzählbar, gdw sie semi-entscheidbar ist.*

Korollar 8.1. *A ist entscheidbar gdw A und $\overline{A}$ beide rekursiv aufzählbar sind.*

8.4 Äquivalente Begriffe

Folgende Aussagen sind äquivalent

- A ist rekursiv aufzählbar
- A ist semi-entscheidbar
- χ'_A ist (Turing-, WHILE-, GOTO-) berechenbar
- $A = T(M)$ für eine TM M
- A ist ein Definitionsbereich einer berechenbaren Funktion
- A ist Wertebereich einer berechenbaren Funktion

8.5 Aufzählbare versus abzählbare Mengen

Definition. Eine Menge A heisst *abzählbar*, falls $A = \emptyset$ oder falls es eine totale (nicht notwendigerweise berechenbare!) Funktion f gibt, so dass

$$A = \{f(0), f(1), \ldots\}$$

Satz 8.3. *Jede Teilmenge A' einer abzählbaren Menge $A = \{f(0), f(1), \ldots\}$ ist wieder abzählbar.*

Aber: Nicht jede Teilmenge einer rekursiv aufzählbaren Menge muss wieder rekursiv aufzählbar sein.

Nichtabzählbarkeit

Satz 8.4. *Nicht alle Mengen sind abzählbar.*

8.6 Nichtentscheidbarkeit

Kodierung von TMs

- Sei $\Gamma = \{a_0, \ldots, a_k\}$ und $Z = \{z_0, \ldots, z_n\}$ (oBdA) durchnummeriert.
- Jedem Tupel in δ der Form $(z_i, a_j, z_{i'}, a_{j'}, y)$ ordnen wir das Wort

  $$\#bin(i)\#bin(j)\#bin(i')\#bin(j')\#bin(m)$$

  zu, wobei

  $$m = \begin{cases} 0, & y = L \\ 1, & y = R \\ 2, & y = N \end{cases}$$

- Wir schreiben diese Wörter in (beliebiger) Reihenfolge hintereinander und erhalten - als Zwischenschritt - ein Wort über $\{0, 1, \#\}$.
- Wir fügen die binäre Kodierung der Start- und Endzustände hinzu.
- Wir ordnen ein Wort über $\{0,1\}$ durch Kodierung zu:

  $$\begin{aligned} 0 &\mapsto 00 \\ 1 &\mapsto 01 \\ \# &\mapsto 11 \end{aligned}$$

Jetzt entspricht eine TM M einem Wort über $\{0,1\}^*$. Jedoch nicht jedes Wort über $\{0,1\}^*$ kodiert eine TM. Sei $\hat{M}$ irgendeine beliebige feste TM (hier als "Default-Programm" benutzt), dann können wir *für jedes* $w \in \{0,1\}^*$ festlegen, dass M_w eine TM bezeichnet, nämlich

$$M_w = \begin{cases} M, & \text{falls } w \text{ Codewort von } M \text{ ist} \\ \hat{M}, & \text{sonst} \end{cases}$$

Spezielles Halteproblem

Definition. $M(w) \downarrow$ bedeutet, dass die TM M, angesetzt auf w, hält. $M(w) \uparrow$ bezeichnet Nicht-Terminierung.

Definition. Das *spezielle Halteproblem* oder *Selbstanwendungsproblem* ist die Sprache

$$K = \{w \in \{0,1\}^* | M_w(w) \downarrow\}$$

Satz 8.5. *Das spezielle Halteproblem ist nicht entscheidbar.*

Reduktionen

Definition. Seien $A \subseteq \Sigma^*$ und $B \subseteq \Gamma^*$ Sprachen. Dann heisst A auf B reduzierbar - symbolisch mit $A \leq B$ bezeichnet - falls es eine totale und berechenbare Funktion $f : \Sigma^* \to \Gamma^*$ gibt, so dass für alle $x \in \Sigma^*$ gilt:

$$x \in A \Leftrightarrow f(x) \in B$$

Satz 8.6. *Falls $A \leq B$ und B entscheidbar (bzw. semi-entscheidbar) ist, so ist auch A entscheidbar (bzw. semi-entscheidbar).*

Korollar 8.2. *Falls A unentscheidbar ist und $A \leq B$, dann ist auch B unentscheidbar.*

Allgemeines Halteproblem

Definition. Das (allgemeine) *Halteproblem* ist die Sprache (für $w, x \in \{0,1\}^*$)

$$H = \{w\#x | M_w(x) \downarrow\}$$

Satz 8.7. *Das Halteproblem H ist nicht entscheidbar.*

Halteproblem auf leerem Band

Definition. Das *Halteproblem auf leerem Band* ist die Sprache

$$H_0 = \{w | M_w(\varepsilon) \downarrow\}$$

d.h. M_w, angesetzt auf leerem Band, hält.

Satz 8.8. *H_0 ist nicht entscheidbar.*

Rice-Satz

Es ist hoffnungslos, irgendeinen Aspekt des funktionalen Verhaltens einer TM algorithmisch bestimmen zu wollen.

Satz 8.9 (Rice). *Sei R die Klasse aller Turing-berechenbaren Funktionen. Sei S eine beliebige Teilmenge hiervon (mit Ausnahme von $S = \emptyset$ und $S = R$). Dann ist die Sprache*

$$C(S) = \{w | \textit{die von } M_w \textit{ berechnete Funktion liegt in } S\}$$

unentscheidbar.

9 Komplexitätstheorie

9.1 Komplexitätsklassen

Man wählt als Berechnungsmodell im Normalfall Mehrband-TMs, man könnte aber irgend ein TM-äquivalentes Modell nehmen.

Definition. $time_M : \Sigma^* \to \mathbb{N}$ ist die Anzahl der Rechenschritte einer (Mehrband-)TM M bei Eingabe von Σ^*.

Definition. Sei $f : \mathbb{N} \to \mathbb{N}$. Die Klasse $TIME(f(n))$ besteht aus allen Sprachen A, für die es eine deterministische (Mehrband-)TM M gibt mit $A = T(M)$ und $\forall x \in \Sigma^* : time_M(x) < f(|x|)$.

Polynomielle Zeit P

Definition. Ein *Polynom* ist eine Funktion $p : \mathbb{N} \to \mathbb{N}$ der Form

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$$

für $a_i \in \mathbb{N}$ und $k \in \mathbb{N}$

Definition. Die Komplexitätsklasse P ist:

$$\begin{aligned} P &= \{A | \text{ es gibt eine TM M und ein Polynom } p \text{ mit} \\ &\quad T(M) = A \text{ und } time_M(x) \leq p(|x|)\} \\ &= \bigcup_{p \ Polynom} TIME(p(n)) \end{aligned}$$

Um zu zeigen, dass ein Problem in P ist, genügt es zu zeigen, dass es eine Lösung (=Algorithmus) mit Komplexität $O(n^k)$ gibt für eine Konstante k.

Satz 9.1. *Die Klasse P und auch weitere komplexere Klassen, etwa $TIME(2^n)$, $TIME(2^{2^n})$ usw., sind PR/LOOP-berechenbar.*

NP

Mit *Nichtdeterminismus* ergibt sich die Möglichkeit für unterschiedliche Berechnungszeiten. Wir werden die kürzeste (akzeptierende) Berechnung als Basis benutzen.

Definition. Für M, eine nichtdeterministische TM, sei

$$ntime_M(x) = \begin{cases} min[\text{Länge einer akzeptierenden Berechnung von } M(x)] & x \in T(M) \\ 0 & x \notin T(M) \end{cases}$$

Sei $f : \mathbb{N} \to \mathbb{N}$. Die Klasse $NTIME(f(n))$ besteht aus der Sprache A für die es eine *nichtdeterministische* Mehrband-TM M gibt mit $A = T(M)$ und $ntime_M(x) \leq f(|x|)$. Ferner definieren wir:

$$NP = \bigcup_{p \ Polynom} NTIME(p(n))$$

Es folgt aus der Definition, dass $P \subseteq NP$ gilt.

9.2 Verifikation vs Existenz

Definition. Ein *Verifikationsverfahren* (VF) für eine Sprache A ist ein Algorithmus V, wobei

$$A = \{w | V \text{ akzeptiert } w\#c \text{ für ein } c \in \Gamma^*\}$$

Ein VF hat *polynomielle Laufzeit*, wenn die Laufzeit polynomiell in $|w|$ ist. c heisst ein Zertifikat. Es kann eine zusätzliche Information sein, z.B. ein Lösungsvorschlag.

Satz 9.2. *Eine Sprache A ist in NP gdw es ein polynomielles VF für A gibt.*

9.3 Polynomielle Reduzierbarkeit

Definition. Seien $A \subseteq \Sigma^*$ und $B \subseteq \Gamma^*$ Sprachen. Dann heisst *A auf B polynomiell redzuzierbar* - symbolisch $A \leq_p B$ - falls es eine totale und in polynomieller Zeit berechenbare Funktion $f : \Sigma^* \to \Gamma^*$ gibt, so dass für alle $x \in \Sigma^*$ gilt:

$$x \in A \Leftrightarrow f(x) \in B$$

Lemma 9.1. *Falls $A \leq_p B$ und $B \in P$ (bzw. $B \in NP$), so ist auch $A \in P$ (bzw. $A \in NP$).*

9.4 NP-hart/ vollständig

Lemma 9.2. *$\leq_p$ ist eine transitive Relation.*

Definition. Eine Sprache A heisst *NP-hart*, falls für alle Sprachen $L \in NP$ gilt: $L \leq_p A$.

Definition. Eine Sprache A heisst *NP-vollständig*, falls A NP-hart ist und $A \in NP$ gilt.

A ist NP-vollständig, falls A mindestens so schwierig ist wie jedes Problem in NP.

9.5 P vs NP

Satz 9.3. *Sei A NP-vollständig. Dann gilt:*

$$A \in P \Leftrightarrow P = NP$$

9.6 SAT

Definition. Das *Erfüllbarkeitsproblem der Aussagenlogik*, kurz SAT, ist:

$SAT = \{code(F) \in \Sigma^* | F \text{ ist eine erfüllbare Formel der Aussagenlogik}\}$

Also gegeben eine Formel F der Aussagenlogik, gefragt: ist F erfüllbar.

Satz 9.4 (Cook). *SAT ist NP-vollständig.*

9.7 3KNF-SAT

Eine Boolsche Formel F in konjunktiver Normalform mit höchstens 3 Literalen pro Klausel. Ist F erfüllbar?

Satz 9.5. *3KNF-SAT ist NP-vollständig.*

9.8 Mengenüberdeckung

Ein Mengensystem über einer endlichen Grundmenge M, also $T_1, \ldots, T_k \subseteq M$, sowie eine Zahl $n \leq k$. Gibt es eine Auswahl aus n Mengen $T_{i_1}, \ldots, T_{i_n}$, in der bereits alle Elemente aus M vorkommen?

Satz 9.6. *Mengenüberdeckung ist NP-vollständig.*

9.9 Clique

Ein ungerichteter Graph $G = (V, E)$ und eine Zahl $k \in N$. Besitzt G eine "Clique" der Grösse mindestens k? D.h. eine Teilmenge $V' \subseteq V$ mit $|V'| \geq k$ und für alle $u, v \in V'$ mit $u \neq v$ gilt: $\{u, v\} \in E$.

Satz 9.7. *Clique ist NP-vollständig.*

9.10 Knotenüberdeckung

Ein ungerichteter Graph $G = (V, E)$ und eine Zahl $k \in N$. Besitzt G eine "überdeckende Knotenmenge" der Grösse höchstens k? D.h. eine Teilmenge $V' \subseteq V$ $|V'| \leq k$, so dass für alle Kanten $\{u, v\} \in E$ gilt: $u \in V'$ oder $v \in V'$.

Satz 9.8. *Knotenüberdeckung ist NP-vollständig.*

9.11 Rucksack

Natürliche Zahlen $a_1, \ldots, a_k \in \mathbb{N}$ und $b \in \mathbb{N}$. Gibt es eine Teilmenge $I \subseteq \{1, 2, \ldots, k\}$ mit $\sum_{i \in I} a_i = b$?

Satz 9.9. *Rucksack ist NP-vollständig.*

9.12 Partition

Natürliche Zahlen $a_1, \ldots, a_k \in \mathbb{N}$. Gibt es eine Teilmenge $J \subseteq \{1, 2, \ldots, k\}$ mit $\sum_{i \in J} a_i = \sum_{i \notin J} a_i$?

Satz 9.10. *Partition ist NP-vollständig.*

9.13 Bin Packing

Eine "Behältergrösse" $b \in \mathbb{N}$, die Anzahl der Behälter $k \in \mathbb{N}$ und "Objekte" $a_1, \ldots, a_n$. Können die Objekte so auf die k Behälter verteilt werden, dass kein Behälter überläuft?

Satz 9.11. *Bin Packing ist NP-vollständig.*