Informatik Cheat Sheet

Formale Sprachen

(Noam) Chomsky-Hierarchie (50er Jahre):

Typ 0	rekursiv aufzählbare Sprachen: Turing-Maschine
Typ 1	kontextsensitive Grammatiken: ohne praktische Bedeutung
Typ 2	kontextfreie Grammatiken: Kellerautomat
Typ 3	reguläre Ausdrücke: endlicher Automat

Definitionen

Alphabet ist eine endliche, nicht leere Menge von Symbolen ($\Sigma = \{a, b, c, \dots\}$)

Zeichenreihe ist eine endliche Folgen von Symbolen eines Alphabets ($w|x|y|z$) Synonyme: Wort, String.

Leere Zeichenreihe enthält keine Symbole und wird als ϵ dargestellt.

Länge bezeichnet die Anzahl Symbole einer Zeichenreihe ($|w|$)

Σ^k Die Menge aller Zeichenreihen über dem Alphabet Σ mit der Längek k. $\Sigma^0 = \{\epsilon\}$

Σ^* Die Menge aller Zeichenreihen über einem Alphabet. $\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \dots$

Σ^+ Die Menge aller nichtleeren Zeichenreihen über einem Alphabet. $\Sigma^+ = \Sigma^* \setminus \Sigma^0$

Konkatenation bezeichnet die Verbindung zweier Zeichenreihen x und y zu xy. Es gilt:

- $x \neq y \Leftrightarrow xy \neq yx$
- $|xy| = |x| + |y|$
- $w\epsilon = \epsilon w = w$ (ϵ ist das neutrale Element)

Sprache ist eine Menge von Zeichenreihen aus Σ^* die als Sprache L über dem Alphabet Σ bezeichnet wird.
Es seien $s(L) := L$ ist eine Sprache, $c(w(L_1), w(L_2)) :=$ Die Menge aller Konkatenation von eines beliebigen $w \in L_1$ und einem beleibigen $w \in L_2$.

- Das Alphabet ist immer endlich, aber es kann unendliche viele Wörter geben.
- $\emptyset$ ist die leere Sprache für jedes Alphabet Σ
- Σ^* ist eine Sprache für jedes Alphabet Σ
- $L \subseteq \Sigma^*$
- $\Sigma_1 \subseteq \Sigma_2 \wedge L \subseteq \Sigma_1^* \Rightarrow L \subseteq \Sigma_2^*$
- $\{\epsilon\}$ ist die Sprache, die aus der leeren Zeichenkette ϵ besteht ($\emptyset \neq \epsilon$)
- Vereinigung: $s(L) \wedge s(M) \Rightarrow s(L \cup M)$
- Konkatenation: $s(L) \wedge s(M) \Rightarrow s(c(w(L), w(M)))$
- (Kleensche) Hülle, Stern: $s(L) \Rightarrow s(L^*)$ mit $L^* = \{c(w(L), w(L))\}$

Problem bezeichnet die Entscheidung, ob eine Zeichenreihe w aus Σ^* in einer Sprache L über dem Alphabet Σ enthalten ist. D. h. Entscheidung ob eine Zeichenreihe zu einer Sprache gehört oder nicht

Notationen

Folgenden Definitionen sind synonym:

$L = \{1^n 0^n | n \in \mathbb{N}\}$

$L = \{\epsilon, 10, 1100, 111000, \dots\}$

$L = \{w | w$ enthält $n \cdot 1$ gefolgt von $n \cdot 0$ für $n \in \mathbb{N}\}$

L ist die Menge aller Zeichenreihen über dem Alphabet$\{0, 1\}$, die aus n Einsen und n Nullen besteht

Reguläre Ausdrücke

1. ϵ und $\emptyset$ sind reguläre Ausdrücke und beschreiben die Sprache $L(\epsilon) = \epsilon$ bzw. $L(\emptyset) = \emptyset$
2. Wenn a ein Symbol ist, dann ist a auch ein regulärer Ausdruck und beschreibt die Sprache $L(a) = \{a\}$.
3. Wenn R ein regulärer Ausdruck ist, dann ist auch (R) ein regulärer Ausdruck, der die gleiche Sprache spezifiziert: $L((R)) = L(R)$ d. h. Klammern sind optional und dienen der Lesbarkeit.
4. Wenn R_1 und R_2 reguläre Ausdrücke sind, dann ist auch $R_1 + R_2$ ein regulärer Ausdruck und es gilt:
   a) $L(R_1 + R_2) = L(R_1) \cup L(R_2)$
   b) $L(R_1 R_2) = L(R_1) L(R_2)$
5. Wenn R ein regulärer Ausdruck ist, dann ist auch R^* ein regulärer Ausdruck und es gilt: $L(R^*) = (L(R))^*$

Reihenfolge Auswertung: $R^* > R_1 R_2 > R_1 + R_2$
$R? = R + \epsilon$ (Auftreten: ein oder keinmal)

Erweiterungen (Unix)

.	beliebiges Zeichen
$[a_1, a_2, \dots, a_n]$	Folgen
[...]	Bereichsangaben wie [0-9] oder Schlüsselwörter

Gesetze

Kommutativgesetz	$L + M = M + L$
Assoziativgesetz	$(L + M) + N = L + (M + N)$
Verkettung	$(LM)N = L(MN)$
Distributivgesetz	$L(M + N) = LM + LN$
Idempotenzgesetz	$L + L = L$

Rechenregeln

$\emptyset + L = L = L + \emptyset$	$\{\epsilon\}^* = \{\epsilon\}$
$\{\epsilon\}L = L = \{L\epsilon\}$	$LL^* = L^*L = L^+$
$\emptyset L = \emptyset = L\emptyset$	$(L^*)^* = L^*$
$\emptyset^* = \{\epsilon\}$	$L^* = L + \{\epsilon\}$

Anwendungen:

1. Mustersuche in Texten
2. Lexikalische Analyse (Compiler), Erkennung von Schlüsselwörtern („Token")
3. Darstellung von Symbolmengen
4. Nachweis der Gültigkeit von gültigen Gesetzen (Gleichheit regulärer Ausdrücke) wird auf die Frage der Gleichheit der Sprachen reduziert (via endliche Automaten, Minimierung und Vergleich)

Automatentheorie

(Stephan Kleene, Michael Rabin, Dana Scott, 40er/50er Jahre)

Endliche Automaten (EA)

5-Tupel: $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$

$Q = \{q_0, \dots, q_n\}$	endliche Menge von Zuständen
$\Sigma = \{a_0, \dots, a_m\}$	Eingabe-Alphabet
$\delta : Q \times \Sigma \to Q$	Übergangsfunktion
$q_0 \in Q$	Startzustand
$F = \{q_o, \dots, q_p\}$	Akzeptierende Endzustände

Deterministischer endlicher Automat (DEA) Geht auf Grund einer Eingabe von einem Zustand in genau einen Zustand über.

Nichtdeterministischer endlicher Automat (NEA) Geht auf Grund einer Eingabe in einen von mehreren Zuständen über.

$$\delta : Q \times \Sigma \to \mathcal{P}(Q)$$

ϵ-NEA wie NEA, erlaubt ϵ als Eingabe $\Rightarrow$ kann spontan wechseln.

Erweiterte Übergangsfunktion Ein Automat verarbeitet einem Reihe von Eingabesymbolen (das Eingabewort) in dem er mit Hilfe der Übergangsfunktion δ den jeweiligen Folgezustand ermittelt: $\delta(q_0, a_0) = q_1, \delta(q_1, a_1) = q_2, \dots$
Dies wird als erweiterte Übergangsfunktion $\hat{\delta} : Q \times \Sigma \to Q$ (d. h. $\hat{\delta}(q_0, w) = q_n$) definiert

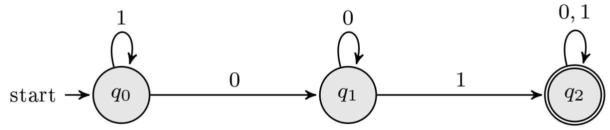
Akzeptieren Ein Eingabewort w wird dann akzeptiert, wenn $\hat{\delta}$ für w in einen akzeptierenden Endzustand $q_n \in F$ führt.

Sprache bzeichnet die Menge aller Zeichenreihen, die der Automat akzeptiert: $L(A) = \{w | \delta(\hat{q_0}, w) \to q_n \in F\}$

Beispiel

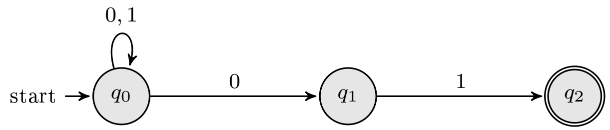
Gesucht wird ein endlicher Automat $\mathcal{A}$ der die Sprache $L = \{x01y \, | x, y \in \Sigma^*$ mit $\Sigma = \{0, 1\}\}$ akzeptiert.
Lösung (DEA):

$$\mathcal{A} = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$$
$$\delta = \{\delta(q_0, 1) = q_0, \delta(q_0, 0) = q_1, \delta(q_1, 0) = q_1, \delta(q_1, 1) = q_2, \delta(q_2, 0) = q_2, \delta(q_2, 1) = q_2\}$$



Lösung (NEA):

$$\mathcal{A} = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$$
$$\delta = \{\delta(q_0, 1) = \{q_0\}, \delta(q_0, 0) = \{q_0, q_1\}, \delta(q_1, 1) = \{q_2\}$$



Es gilt: RA $\Leftrightarrow$ DEA $\Leftrightarrow$ NEA $\Leftrightarrow$ ϵ-NEA

Eigenschaften regulärer Sprachen

Sind L_1 und L_2 reguläre Sprachen, dann gilt:

Vereinigung	$L_1 \cup L_2$ ist regulär
Durchschnitt	$L_1 \cap L_2$ ist regulär
Komplement	${L_1}^C$ ist regulär
Verkettung	$L_1 + L_2$ ist regulär
Differenz	$L_1 - L_2$ ist regulär
Hülle	L_1^* und L_1^+ sind regulär
Homomorphismus	$h(L_1)$ und $h^{-1}(L_1)$ sind regulär
Spiegelung	${L_1}^R$ ist regulär

Entscheidbarkeit regulärer Sprachen

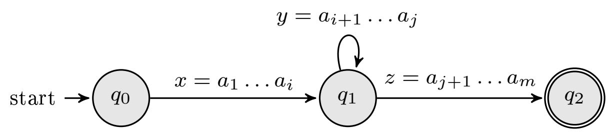
Gebeben sei eine reguläre Sprache L

Ist L leer?	entscheidbar
$w \in L$?	entscheidbar
$L_1 = L_2$?	entscheidbar

Pumping-Lemma

Sei L eine reguläre Sprache. Dann gibt es eine Konstante n, so dass jder Zeichenreihe $w \in L, |w| > n$, in drei Teilzeichenreihen $w = xyz$ wie folgt zerlegt werden kann:

$$y \neq \epsilon$$
$$|xy| \leq n$$
$$xy^k z \in L, \text{ für alle } k \geq 0$$



Kann für eine Sprache gezeigt werden, dass sie das Pumping-Lemma nicht erfüllt, ist sie nicht regulär.

Pro Memoria: Moore- & Mealy-Automat

Beide Typen sind logisch äquivalent und lassen sich ineinander überführen. Beide unterscheiden sich von den endlichen Automaten dadurch, dass sie neben finalen Zuständen auch eine Ausgabe erzeugen. Sie werden als 7-Tupel definiert:

$$\mathcal{A} = (Q, \Sigma, \Omega, \delta, \lambda, q_0, F)$$

- Q endliche Menge von Zuständen
- Σ endliches Eingabealphabet
- Ω endliches Ausgabealphabet
- δ Übergangsfunktion $\delta : Q \times \Sigma \to Q$
- λ Ausgabefunktion $\lambda : Q \to \Omega$
- q_0 Startzustand
- F endliche Menge finaler Zustände ($F \subset Q$)

Moore-Automat Die Ausgabe wird mit dem Zustand assoziiert und hängt somit nur vom Zustand ab.

Mealy-Automat Die Ausgabe wird mit der Transition assoziiert, dass heisst, sie hängt vom Zustand und der Eingabe ab. Beim Zustandswechsel wird zusätzlich etwas ausgegeben.

Kontextfreie Grammatik (kfG)

Eine kontextfreie Grammatik wird als 4-Tupel definiert:

$$G = (V, T, P, S)$$

- V Endliche Menge von Zeichenreihen $(A, B, C, \dots)$
- T Endliche Menge von Terminalsymbolen $(a, b, c, \dots)$
- P Endliche Menge von Produktionen/Regeln, definieren die Sprache rekursiv $(A \to AA)$
- S Das Startsymbol $(S \in V)$

Beispiel: Palindrome $(\Sigma = \{0, 1\})$

$$G_P = (\{A\}, \{0,1\}, P, A)$$
$$P : A \to \epsilon, A \to 0, A \to 1, A \to 0A0, A \to 1A1$$
$$\text{oder } P : A \to \epsilon|0|1|0A0|1A1$$

Ableitungsschritt Gegeben sei eine kfG G und eine Wort $\alpha A \beta$, mit α und β sind Wörter aus $V \cup T$ und $A \in V$ sowie einer Produktion $A \to \gamma \in P$. Dann ist die Relation $\alpha A \beta \Rightarrow \alpha\gamma\beta$ ein Ableitungsschritt in G.
Beispiel: $0A0 \Rightarrow 00A00$ ist ein Ableitungsschritt.

Ableitung Ein Ableitungsschritt kann zu einer Ableitung aus $0 - n$ Ableitungsschritten erweitert werden: $\alpha \underset{G}{\overset{*}{\Rightarrow}} \gamma$ ist eine Ableitung in G.
Beispiel:
$A \underset{G}{\overset{*}{\Rightarrow}} 1101011 : A \underset{G}{\Rightarrow} 1A1 \underset{G}{\Rightarrow} 11A11 \underset{G}{\Rightarrow} 110A011 \underset{G}{\Rightarrow} 1101011$

Um Ableitungsschritte zu definieren empfiehlt sich:

1. Ausdrucksproduktionen (A) bestimmen (z. B. $A \to A + A$)
2. Bezeichnerproduktionen (B) bestimmen (z. B. $B \to a|b|\dots|z$)
3. Ausdrücke ableiten (von links oder von rechts: $A \underset{G}{\Rightarrow} A + A$)
4. Bezeichner ableiten (von links oder rechts, aber gleich wie im vorherigen Schritt: $A + A \underset{G}{\Rightarrow} a + A \underset{G}{\Rightarrow} a + b$)

Linksseitige Ableitung in jedem Schritt wird die äusserste linke Variable durch eine Produktion ersetzt: $\alpha \underset{lm}{\overset{*}{\Rightarrow}} \gamma$.

Rechtsseitige Ableitung in jedem Schritt wird die äusserste rechte Variable durch eine Produktion ersetzt: $\alpha \underset{rm}{\overset{*}{\Rightarrow}} \gamma$.

Sprache der kfG Menge der Zeichenreihen aus terminalen Symbolen, die sich vom Startsymbol ableiten lassen:

$$L(G) = \{w | S \underset{G}{\overset{*}{\Rightarrow}} w \wedge w \in T^*\}$$

Parsebäume

- stellen Ableitungen als Bäume dar
- verdeutlichen wie terminale Wörter aus Teilwörtern strukturiert sind
- werden von Compilern erzeugt: Datenstruktur, die Quellprogramme repräsentiert

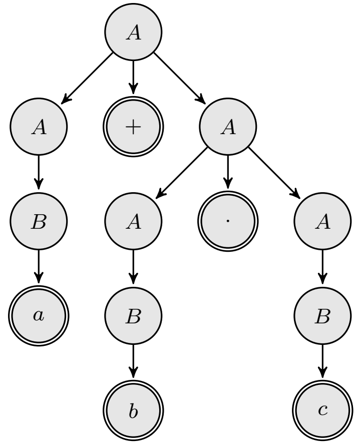
Gegeben sei eine kfG $G = (V, T, P, S)$. Ein Baum ist ein Parsebaum von G wenn:

1. Alle inneren Knoten mit einer in V enthaltenen Variable bezeichnet sind
2. Für jedes Blatt gilt, dass es entweder
   - mit einer Variablen aus V
   - einem terminalen Symbol aus T oder
   - dem leeren Wort ϵ bezeichnet ist, dann muss das Blatt aber das einzige des Vorgängerknotens sein
3. Für alle innere Knoten gilt (A = Bezeichnung des Knotens und $X_1, \dots, X_n$ Bezeichnung der Nachfolgerknoten von links nach rechts): $(A \to X_1, \dots, X_n) \in P$

Beispiel Programm-Ausdruck: $A \underset{G}{\overset{*}{\Rightarrow}} a + b \cdot c$:

(Linksseitige) Ableitung:

$$\begin{aligned} A &\underset{G}{\Rightarrow} A + A \\ &\underset{G}{\Rightarrow} A + A \cdot A \\ &\underset{G}{\Rightarrow} B + A \cdot A \\ &\underset{G}{\Rightarrow} B + B \cdot A \\ &\underset{G}{\Rightarrow} B + B \cdot B \\ &\underset{G}{\Rightarrow} a + B \cdot B \\ &\underset{G}{\Rightarrow} a + b \cdot B \\ &\underset{G}{\Rightarrow} a + b \cdot c \end{aligned}$$



Die Blätter eines Parsebaums ergeben von links nach rechts gelesen eine Zeichenreihe einer kfG $G = (V, T, P, S)$ wenn:

1. Die Wurzel mit dem Startsymbol bezeichnet ist
2. Alle Blätter mit terminalen Symbolen oder ϵ bezeichnet sind

Die Sprache $L(G)$ ist genau die Menge an Zeichenreihen, die sich aus der von links nach rechts gelesenen Zeichen der Blätter aller Parsebäume ergeben.

Mehrdeutigkeit

- Anwendungen (Programmiersprachen) erfordern, dass die mit einer kfG erzeugten Strukturen eindeutig sind
- Nicht jede kfG ist eindeutig
- Es gibt Sprachen, für die jede kfG immer mindestens einer Zeichenkette mehr als eine Struktur zuordnen. Diese Sprachen heissen „inhärent mehrdeutig“. Im Beispiel ($A \underset{G}{\overset{*}{\Rightarrow}} a + b \cdot c$) kann nicht erkannt werden ob $a + (b \cdot c)$ oder $(a + b) \cdot c$ zu rechnen ist (Aus der linksseitigen Ableitung ergibt sich $a + (b \cdot c)$, aus der rechtsseitigen ergäbe sich das andere).

Ein Sprache ist mehrdeutig, wenn für mindestens eine Zeichenreihe mehr als ein Parsebaum exisitiert.
Häufig (aber nicht immer) lässt sich eine kfG so anpassen, dass sie eindeutig wird.

Mehrdeutigkeit eliminieren

Das Beispiel $A \overset{*}{\underset{G}{\Rightarrow}} a + b \cdot c$ wird eindeutig durch:

- erzwungene Klammern oder
- angepasste Produktionen, so dass $\cdot > +$
- Linksableitung vorgeben $A \to A + A|A \cdot A|(A)$ (Addition wird vor der Multiplikation, die wiederum vor der Klammer aufgelöst)

Anwendungen

- Beschreibung natürlicher Sprachen (N. Chomsky): allerdings sind kfG nicht für natürliche Sprachen geeignet, weil Mehrdeutigkeiten häufig vom Kontext abhängen
- Compiler: Klammern, Blöcke, Bedingungen erfordern KfG, für andere Komponenten reichen reguläre Ausdrücke (z. B. $A \to \epsilon|if(A)else|if(A)|AA$)
- Parsergeneratoren (Quellcode $\to$ Parsebaum):
  - Bezeichner werden über lexikalische Analyse erkannt
  - Strukturen werden über kfG erkannt
- Markupsprachen (z. B. HTML): wie Parsegeneratoren
- XML: (kfG beschreibt die Semantik über die DTD(Dokumentendefinition))

Pushdown-Automaten (=Kellerautomaten)

Sind im Gegensatz zu endlichen Automaten unendliche Automaten.

Pushdown-Automat (PDA) Ein (nichtdeterministischer) Pushdown-Automat P wird als 7 Tupel definiert: $P = (Q, \Sigma, \Gamma, \delta, q_0, s_0, F)$

- Q endliche Menge von Zuständen
- Σ endliches Eingabealphabet
- Γ endliches Stackalphabet
- δ Übergangsfunktion $\delta\colon Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$
- q_0 Startzustand ($q_0 \in Q$)
- s_o Startsymbol des Stacks ($s_0 \in \Gamma$)
- F Menge der akzeptieren Zustände ($F \subset Q$)

Die Übergangsfunktion δ bildet das Tripel $\delta(q_v \in Q, e \in \Sigma, s_v \in \Gamma)$ auf eine endliche Menge von Paaren $(q_n \in Q, s_n \in \Gamma)$ ab. Die Indexe n und v stehen dabei für vorher (v) respektive nachher (n).

Stack Ein Stack ist ein Last In – First Out (LIFO) Speicher. Ein PDA liest das oberste Elemente $x \in \Gamma$ und ersetzt es durch $\gamma \in \Gamma$.

- $\gamma = \varepsilon$ („pop“, entfernt oberstes Element)
- $\gamma = x$ („void“, Stack unverändert)
- $\gamma \neq x \wedge \gamma \neq \varepsilon$ („pop/push“, ersetzt oberstes Element)
- Ist $|\gamma| > 1$ entspricht einem „pop“ und $|\gamma|$ „pushs“.

Konfiguration Das Tripel (q, w, γ) wird als Konfiguration K von P bezeichnet.

- q aktueller Zustand
- w noch nicht gelesener Teil der Eingabealphabet
- γ Inhalt des Stacks

Bewegung Sei (p, α) ein Element von $\delta(q, e, s)$, dann stellt ein Wechsel von Konfiguration $(q_v, ew, s\beta)$ nach $(q_n, w, \alpha\beta)$ eine Bewegung dar: $(q_v, ew, s\beta) \underset{P}{\vdash} (q_n, w, \alpha\beta)$

Berechnung Gegeben sei ein PDA und eine Folge von Konfigurationen $K_1, \ldots, K_n$ für die paarweise gilt: $K_i \vdash K_{i+1}$ ist eine Bewegung, dann stellt die Folge der Konfigurationen eine Berechnung vom PDA dar (analog $\hat{\delta}$): $K_1 \overset{*}{\underset{P}{\vdash}} K_n$

Sprache (Endzustand) Die Menge aller Zeichenreihen w, für die in P ausgehend von der Anfangskonfiguration K_0 eine Berechnung exisitiert, so dass P in einen akzeptierenden Endzustand wechselt wird mit $L(P)$ bezeichnet.

$$L(P) = \{w | (q_0, w, s_0) \overset{*}{\underset{P}{\vdash}} (f, \varepsilon, \beta) \wedge f \in F\}$$

P akzeptiert durch seinen Endzustand. Der Stack spielt keine Rolle.

Sprache (leerer Stack) Die Menge aller Zeichenreihen w für die in P_N ausgehend von einer Anfangskonfiguration K_0 eine Berechnung existiert, so dass P nach dem Einlesen von w einen leeren Stack aufweist wird mit $N(P)$ bezeichnet.

$$N(P) = \{w | (q_0, w, s_0) \overset{*}{\underset{P}{\vdash}} (f, \varepsilon, \varepsilon)\}$$

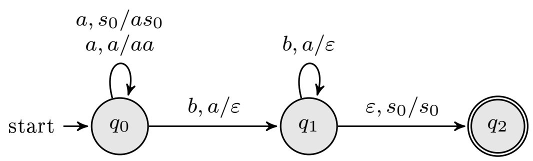
P akzeptiert durch leeren Stack. Der Zustand spielt keine Rolle d. h. es gibt keine Endzustände, d. h. $P = (Q, \Sigma, \Gamma, \delta, q_0, s_0)$ (6-Tupel!)

Der Automat schreibt also ein ε, wenn er s_0 im Keller hat und die Eingabe ε liest.

Beispiel (akzeptiert durch Endzustand)

Ein Automat, der die Sprache $L = \{a^n b^n | n > 0\}$ erkennt:

$$P = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, s_0\}, \{(q_0, w, s_0) \overset{*}{\vdash} (q_3, \varepsilon, \varepsilon)\}, q_0, s_0, q_2)$$



Der Automat führt beispielsweise die Berechnung $(q_0, aabb, s_0) \overset{*}{\vdash} (q_3, \varepsilon, s_0)$ durch. Die Bewegungen dabei sind:

$$(q_0, aabb, s_0) \vdash (q_0, abb, as_0) \vdash (q_0, bb, aas_0) \vdash (q_1, b, as_0)$$
$$\vdash (q_1, \varepsilon, s_0) \vdash (q_2, \varepsilon, s_0)$$

Sätze

Gegeben sei ein PDA $P = P = (Q, \Sigma, \Gamma, \delta, q_0, s_0, F)$:

- Wenn $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$ eine Berechnung in P ist, dann gilt für alle $w \in \Sigma^*$ und alle $\gamma \in \Gamma^*$, dass auch $(q, xw, \alpha\gamma) \overset{*}{\underset{P}{\vdash}} (p, yw, \beta\gamma)$ eine Berechnung in P ist.
- Wenn $(q, xw, \alpha) \overset{*}{\underset{P}{\vdash}} (p, yw, \beta)$ eine Berechnung in P ist, dann ist auch $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$ eine Berechnung in P (Umkehrung von 1)
- Aber aus $(q, x, \alpha\gamma) \overset{*}{\underset{P}{\vdash}} (p, \gamma, \beta\gamma)$ folgt nicht, dass auch $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (p, y, \beta)$ eine Berechnung ist, weil die Eingabe „verbraucht“ wird.
- Für jeden Automat P_N, der durch leeren Stack akzeptiert, gibt es auch einen Automat P_L, der durch den Endzustand akzeptiert: $L = N(P_N) = L(P_L)$
- Für jeden Automat P_L, der durch Endzustand akzeptiert, gibt es auch einen Automat P_N, der durch leeren Stack akzeptiert: $L = L(P_L) = N(P_N)$.
- $L(G) \Leftrightarrow N(P) \Leftrightarrow L(P)$ (mit G ist eine kfG).

Deterministischer PDA (DPDA)

Ein PDA ist deterministisch wenn:

1. $\delta(q, a, s)$ höchstens ein Element enthält
2. $\delta(q, a, \gamma) \neq \emptyset \Rightarrow \delta(q, \varepsilon, \gamma) = \emptyset$

Ein DPDA P kann in jeder Konfiguration höchstens eine Bewegung ausführen. Er kann reguläre Sprachen erkennen, aber nicht alles, was ein PDA erkennen kann.

$$L^*(\text{DEA}) = L^*(\text{NEA}) = L^*(\varepsilon\text{NEA}) \subsetneq L^*(\text{DPDA}_E) \subsetneq L^*(\text{PDA})$$

DPDA_E steht für einen Automaten, der durch Endzustand akzeptiert.
Für eine Sprache L und einen DPDA P, der über leeren Stack akzeptiert, gilt $N(P) = L$ genau dann wenn:

1. L präfixfrei (kein Wort ist der Anfang eines anderen Wortes) ist
2. Es einen DPDA P', der über leeren Stack akzeptiert, gibt mit $L(P) = L$

Ein DPDA der durch leeren Stack akzeptiert erkennt nicht mal alle reguläre Sprachen (z. B.: $L = \{0\}^*$), allerdings kann er auch nicht-reguläre Sprachen (z. B. $L = \{wcw^R | w \in \{0, 1\}\}$) erkennen.

DPDA und kfG

Gegeben sei eine Sprache L und ein DPDA P.

- $L = N(P) \Rightarrow L$ hat eine eindeutige kontextfreie Grammatik.
- $L = L(P) \Rightarrow L$ hat eine eindeutige kontextfreie Grammatik.
- L hat eine eindeutige kontextfreie Grammatik heisst aber nicht, dass es einen passenden DPDA gibt. (z. B. Palindrome)

Eigenschaften kontextfreier Sprachen

Seien L_1 und L_2 kontextfreie Sprachen über Σ. Dann gelten:

Vereinigung	$L_1 \cup L_2$ ist kontextfrei
Durchschnitt	$L_1 \cap L_2$ ist nicht kontextfrei
Komplement	${L_1}^C$ kann kontextfrei sein
Verkettung	$L_1 + L_2$ ist kontextfrei
Differenz	$L_1 - L_2$ kann kontextfrei sein
Hülle	L_1^* und L_1^+ sind kontextfrei
Homomorphismus	$h(L_1)$ und $h^{-1}(L_1)$ sind kontextfrei
Spiegelung	${L_1}^R$ ist eine kontextfrei

Entscheidbarkeit kontextfreier Sprachen

Gebeben sei ein kontextfreie Sprache L und eine kontextfreie Grammatik G

Ist L leer?	entscheidbar
$w \in L$?	entscheidbar
Ist L inhärent mehrdeutig?	nicht entscheidbar
Ist G mehrdeutig	nicht entscheibar
$L_1 \cap L_2 = \emptyset$?	nicht entscheidbar
$L_1 = L_2$?	nicht entscheidbar

Nicht kontextfreie Sprachen

Pumping-Lemma

Sei L eine kontextfreie Sprache. Dann gibt es eine eine Konstante n, so dass jede Zeichenreihe $w \in L$ und die Länge von $z \geq 0$ in fünf Teilzeichenreihen $w = uvxyz$ derart zerlegt werden kann, dass:

1. $|vxy| \leq n$
2. $vx \neq \varepsilon$
3. $uv^k xy^k z \in L$ für alle $k \geq 0$

Eine mittlere Teilzeichenreihe (vxy) wird beliebig oft wiederholt.
Beispiel: $L = \{0^m 1^m 2^m | m > 0\}$
Ist L kontextfrei, dann gibt es eine Konstante n, so dass $0^n 1^n 2^n$ ebenfalls in L ist: Nun muss w in $uvxyz$ so zerlegt werden, dass $vx \neq \varepsilon$ und $|vxy| \leq n$ gilt. vxy kann aber nicht zugleich 0 und 2 enthalten (sonst wäre der Abstand zwischen der letzen 0 und der ersten $2 \neq n + 1$)
Fall I: $2 \notin vxy \Rightarrow 0 \vee 1 \in uwz$: Widerspruch zur Annahme $uwz \in L$
Fall II: $0 \notin vxy \Rightarrow 1 \vee 2 \in uwz$: Widerspruch zur Annahme $uwz \in L$

Turing-Maschine (TM)

Hintergrund

Gibt es einen Algorithmus der die Wahrheit jeder mathematischen Aussage ermittelt?

Ja	für Aussgenlogkik
Nein	für die Prädikatenlogik (Gödel, 1930)

Definition

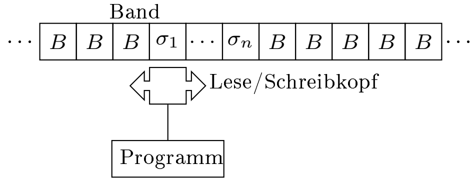
Eine Turing Maschine kann alle möglichen Berechnungen ausführen und wird als 7-Tupel definiert:

$$\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

Q, Σ, q_0, F wie gehabt (Zustände, Alphabet, ...)

- Γ endliches Bandalphabet $(\Sigma \subset \Gamma)$
- δ Übergangsfunktion: $Q \times \Gamma \to Q \times \Gamma \times D$
 d. h. $\delta(q_v, \sigma) = (q_n, \gamma, d)$. D steht für die Bewegung des Lese/Schreibkopfes (L = links, 0 = keine, R = rechts)
- B „Leerzeichen", $B \in \Gamma \wedge B \notin \Sigma$

- Das Band besteht aus einzelnen Zellen, die ein Element aus Γ enthalten können
- Zu Beginn enthält das Band die „Eingabe" – eine endliche Zeichreihe aus Σ. (Alle anderen enthalten das Leerzeichen B)
- Der Lese/Schreibkopf kann jeweils eine Zelle lesen und schreiben



Turing-Maschine mit der Anfangskonfiguration $w = \sigma_1 \sigma_2 \ldots \sigma_n$

Konfiguration/Instanzdeskripor Gegeben sei eine Turing-Maschine $\mathcal{M}$. Die Zeichenreihe $X_1 X_2 \ldots X_{i-1} q X_i X_{i+1} \ldots X_n$ heisst Konfiguration von $\mathcal{M}$. Dabei ist q der Zustand von $\mathcal{M}$, X_i die Position des Lese/Schreibkopfes. Alle anderen $X_a, a \neq i$ können leer sein.

Bewegung Gegeben sei eine Turing-Maschine $\mathcal{M}$. Mit der Konfiguration $X_1 X_2 \ldots X_{i-1} q X_i X_{i+1} \ldots X_n$

- Linksbewegung: $\delta(q, X_i) = (p, Y, L)$ d. h. $X_i \to Y$, Kopf neu bei X_{i-1}
  - $i = 1 \Rightarrow qX_1 \ldots X_n \underset{\mathcal{M}}{\vdash} pBYX_2 \ldots X_n$
  - $i = n \wedge Y = B \Rightarrow X_1 \ldots qX_n \underset{\mathcal{M}}{\vdash} X_1 \ldots X_{n-2} p X_{n-1}$
- Rechtsbewegung: $\delta(q, X_i) = (p, Y, R)$ d. h. $X_i \to Y$, Kopf neu bei X_{i+1}
  - $i = n \Rightarrow X_1 \ldots X_{n-1} q X_n \underset{\mathcal{M}}{\vdash} X_1 \ldots X_{n-2} Y p B$
  - $i = 1 \wedge Y = B \Rightarrow qX_1 \ldots X_n \underset{\mathcal{M}}{\vdash} pX_2 \ldots X_n$
- 0-Bewegung: $\delta(q, X_i) = (p, Y, 0)$ d. h. $X_i \to Y$ ohne dass sich der Kopf bewegt

Berechnung Gegeben sei eine Turing-Maschine $\mathcal{M}$ und ein Folge von Konfigurationen $K_1, \ldots, K_n$ für die paarweise gilt: $(K_i) \underset{\mathcal{M}}{\vdash} (K_{i+1}$ ist eine Bewegung in M, dann stellt die Folge eine Berechnung von M dar: $(K_1) \overset{*}{\underset{\mathcal{M}}{\vdash}} (K_n)$

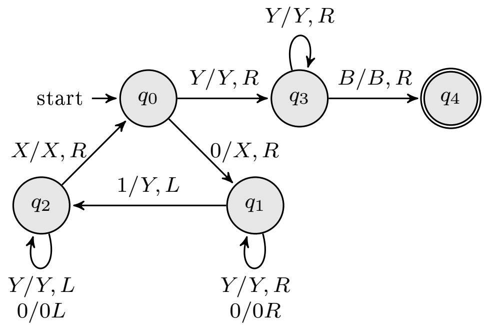
- Die Berechnung einer TM ist abgeschlossen wenn $\delta(q, x)$ nicht definiert ist. Sie hält an
- Zahlen werden als Blöcke von „0" dargestellt. Mehrere Parameter werden mit „1" getrennt
- Ist eine Funktion f für alle Parameter definiert, dann ist f eine total rekursive Funktion ($\widehat{=}$ rekursive Sprache)
- Wird f allgemein von TM berechnet, ist f eine partiell rekursive Funktion ($\widehat{=}$ rekursiv aufzählbare Sprache)
- Alle üblichen mathematischen Funktionen $(+, -, \cdot, \div, n!, \log, a^b, \ldots)$ sind total rekursive Funktionen

Rekursiv aufzählbare Sprache Die Menge aller Zeichenreihen $w \in \Sigma^*$ für die eine Turing-Maschine $\mathcal{M}$ in einen akzeptierenden Endzustand übergeht:

$$L(\mathcal{M}) = \{w | (q_0 w) \overset{*}{\vdash} (\alpha p \beta) \text{ für ein } p \in F \wedge \alpha, \beta \in \Gamma^*\}$$

w ist die Eingabe, $\mathcal{M}$ startet in q_0, der Lese/Schreibkopf beginnt beim ersten Zeichen von w. Solange die TM eine Eingabe bearbeitet und nicht anhält, kann nicht entschieden werden ob $w \in L(\mathcal{M})$.
Erkannt werden u. a. alle kontextfreien Sprachen. Es gilt: rekursive Sprache $\subset$ rekursiv aufzählbare Sprachen.

Beispiel Turing-Maschine für $L = \{0^n 1^n | n \in \mathbb{N}\}$



$$\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

$Q = \{q_0, q_1, \ldots, q_4\}$	$\Sigma = \{0, 1\}$
$\Gamma = \{0, 1, X, Y, B\}$	$q_0 = q_0$
$F = \{q_4\}$	
$\delta(q_0, 0) = (q_1, X, R)$	$\delta(q_0, Y) = (q_3, Y, R)$
$\delta(q_1, 0) = (q_1, 0, R)$	$\delta(q_1, 1) = (q_2, Y, L)$
$\delta(q_1, Y) = (q_2, Y, R)$	$\delta(q_2, 0) = (q_2, 0, L)$
...	

Berechnung von $w = 01 : q_0 01 \overset{*}{\vdash} XYBq_4B$

$$q_0 01 \vdash Xq_1 1 \vdash q_2 XY \vdash Xq_0 Y \vdash XYq_3 B \vdash XYBq_4 B$$

Codierung einer TM

1. Jeder Zeichenreihe über Σ wird eine ganze Zahl in lexikographischer Ordnung zugewiesen: $1 = \varepsilon, n = \sigma_{n-1} \in \Sigma$
2. Die Zustände Q der TM $\mathcal{M}$ werden codiert als: q_1 =Start-, q_2 =akzeptierender Endzustand, $q_3 \ldots q_n$ übrige Zustände.
3. Die Bandsymbole Γ von $\mathcal{M}$ werden codiert als: $x_1 = 0, x_2 = 1, x_3$ =Blank, $x_4 \ldots x_n$ für alle weiteren Symbole
4. Die Richtung des Lese-Schreibkopfes wird codiert als: d_1 = links, d_2 = rechts
5. Die Übergangsfunktion δ von $\mathcal{M}$ wird codiert über die Zeichenreihe $0^i 1 0^j 1 0^k 1 0^l 1 0^m$, Übergangsfunktionen werden durch 11 voneinander getrennt

So lässt sich jede TM $\mathcal{M}$ als Zahl darstellen:

$$\mathcal{M} = (\{q_1, q_2, q_3\}, \{0, 1\}, \{0, 1, B\}, \delta, q_1, B, \{q_2\})$$
$$\delta = \{\delta(q_1, 1) = (q_3, 0, R\}, \delta(q_3, 0) = (q_1, 1, L),$$
$$\delta(q_3, 1) = (q_2, 0, R), \delta(q_3, B) = (q_3, 0, L)\}$$

Die Codierung $c(\mathcal{M})$ lautet dann:

$$\delta(q_1, 1) = (q_3, 0, R) \mapsto 0\,1\,00\,1\,000\,1\,0\,1\,00$$
$$\delta(q_3, 0) = (q_1, 1, L) \mapsto 000\,1\,0\,1\,0\,1\,00\,1\,0$$
$$\delta(q_3, 1) = (q_2, 0, R) \mapsto 000\,1\,00\,1\,00\,1\,0\,1\,00$$
$$\delta(q_3, B) = (q_3, 0, L) \mapsto 000\,1\,000\,1\,000\,1\,0\,1\,0$$

$$\Rightarrow c(\mathcal{M}) = 0100100010100\,11\,0001010100 10\,11\,00010010010100$$
$$11\,0001000100010 10_2 = 1\,480\,103\,890\,654\,955\,658_{10}$$

Sonstiges

- Es existieren Varianten von TM: Speicher, Mehrspur, Mehrband, Semiunendliches Band, Leerzeichenverbot. Alle Varianten sind gleich mächtig
- Eine TM ist gleich mächtig wie ein PDA mit 2 Stacks oder einer Zählermaschine mit zwei Zählern
- Eine TM kann Aufgaben an „Unterprogramme“, d. h. Unter-TMs delegieren
- Jede TM mit n akzeptierenden Zuständen kann in eine TM mit nur einem akzeptierenden Zustand überführt werden

Berechenbarkeit/Rekursionstheorie

(Kurt Gödel, Alonzo Church, Alan Turing, 30er Jahre)

$$\text{Computer} \Leftrightarrow \text{Turing-Maschine}$$

Berechenbarkeit Eine Funktion f ist berechenbar, wenn es eine TM $\mathcal{M}$ gibt, die f berechnet. D. h. es kann ein Algorithmus gefunden werden, der f berechnet

Jede TM kann *partiell rekursive Funktionen*/rekursive aufzählbare Sprachen berechnen/erkennen

Primitiv rekursive Funktion Eine Funktion, die über mindestens ein Abbruchkriterium verfügt und sich selbst aufruft. Für primitiv rekursive Funktionen gilt: Eine TM hält für alle Eingaben. Beispiele: Addition, Multiplikation, Potenz, Fibonacci-Zahlen

Partiell rekursive Funktion (μ-rekursive Funktion) Eine Funktion, welche die primitiv rekursiven Funktionen um den μ-Operator erweitert. Der μ bildet dabei eine $k+1$ stellige Funktion auf ein k stellige ab. (Beispiele: Nachfolgerfunktion $\sigma(n) = n+1$, Ackermannfunktion (s. u.)). Somit gilt:

$$\text{primitiv rekursive Funktionen} \subsetneq \mu - \text{rekursive Funktionen}$$

Rekursiv aufzählbar Es gibt einen Algorithmus, um allen Elementen einer Menge eine natürliche Zahl n zuzuordnen

Church'sche These Die Klasse der Turing-berechenbaren Funktionen ist genau die Klasse der „intuitiv“ berechenbaren Funktionen

Rekursive Sprache Eine Sprache L heisst rekursiv, wenn $L = L(\mathcal{M})$ für eine TM $\mathcal{M}$ ist und für jedes w gilt:

- $w \in L \Rightarrow \mathcal{M}$ hält und akzeptiert
- $w \notin L \Rightarrow \mathcal{M}$ hält und akzeptiert nicht

Programm-Typen

Als Beispiel ist jeweils die Addition von $x + y$ aufgeführt.

GOTO-Programme

Endliche Anzahl von Variablen $a_1, \dots a_n$ und Konstanten $K_1, \dots K_n$
Jede Anweisung (Zeile) ist nummeriert
Fünf verschiedene Anweisungen:

1. Zuweisung & „+“: $a = a + K$
2. Zuweisung & „−“: $a = a - K$
3. Sprunganw. (GOTO Zeile n)
4. Verzweigung (`IF ... ELSE`)
5. Stopp-Anweisung

```
1  k := x
2  l := y
3  k := k + 1
4  l := l - 1
5  if (l > 0) goto 3
6  end
```

GOTO-Programme stehen für unstrukturierten Code (Programmiersprache BASIC). Es gilt:

$$\text{GOTO-Programme} \Leftrightarrow \text{Turing-Maschine}$$

WHILE-Programme

Endliche Anzahl Variablen $a_1, \dots a_n$ und Konstanten $K_1, \dots K_n$
Drei verschiedene Anweisungen:

1. Zuweisung & „+“: $a = a + K$
2. Zuweisung & „−“: $a = a - K$
3. Bedingte Schleife (`WHILE ... DO ... END`)

```
k := x
l := y
while (l > 0) do
k := k + 1
l := l - 1
end
```

Jedes beliebige WHILE-Programm kann durch eine WHILE-Programm mit genau einer Schleife ersetzt werden!
WHILE-Programmen stehen für strukturierten Code (Programmiersprachen Pascal, C, Java, …). Es gilt:

$$\text{WHILE-Programme} \Leftrightarrow \text{GOTO-Programme}$$

LOOP-Programme

Endliche Anzahl Variablen $a_1, \dots a_n$ und Konstanten $K_1, \dots K_n$
Drei verschiedene Anweisungen:

1. Zuweisung & „+“: $a = a + K$
2. Zuweisung & „−“: $a = a - K$
3. Feste Schleife (`LOOP DO ... END`)
   Die Anzahl der Schleifen ist zu Beginn festgelegt (`FOR NEXT`)

```
k := x
l := y
loop l do
k := k + 1
end
```

Es gilt:

→ Jedes `LOOP`-Programm kann durch ein `WHILE`- oder `GOTO`-Programm ersetzt werden
→ Nicht jedes `WHILE-`/`GOTO`-Programm kann durch ein `LOOP`-Programm ersetzt werden
→ `LOOP`-Programme entsprechen den primitiv rekursiven Funktionen und terminieren immer
→ `GOTO-`/`WHILE`-Programme entsprechen den μ-rekursiven Funktionen und terminieren nicht immer

Partiell rekursive Funktionen

Mit Hilfe von struktureller Induktion und Widerspruch kann gezeigt werden, dass es Funktionen (z. B. Ackermannfunktion) gibt, die schneller wachsen als primitiv rekursive Funktionen. Die Ackermannfunktion ist rekursiv, aber nicht primitiv rekursiv.

$$a(0, m) = m + 1$$
$$a(n+1, 0) = a(n, 1)$$
$$a(n+1, m+1) = a(n, a(n+1, m))$$

Entscheidbarkeit

Die Entscheibarkeit eines Problems L lässt sich unterteilen:

Entscheibar sind alle Probleme/Sprachen, die rekursiv sind. D. h. eine TM hält für alle rekursiven Eingaben.

Semi-entscheidbar sind alle Probleme/Sprachen, die zwar rekursiv aufzählbar, aber nicht rekursiv sind. D. h. eine TM hält für alle rekursiv aufzählbaren Eingaben.

Unentscheidbar sind alle Probleme/Sprachen, die nicht rekursiv aufzählbar sind.

Unentscheidbare Probleme

Diagonalisierungssprache L_d (codiert eine TM)

Bildung:
Jede $\mathcal{M}_i$ lässt sich als Zeichenreihe w_i darstellen (Codierung TM). In einer Tabelle mit allen w_i als Zeilen und w_i als Spalten steht in Zelle z, s ob $\mathcal{M}_z \mathcal{M}_s$ als Eingabe akzeptiert (Wert: 1) oder nicht (Wert: 0) L_d ist die Sprache aus allen denen Wörtern, die Turing-Maschinen codieren, die sich selbst nicht als Eingabewort akzeptieren.

$$L_d = \{w_i | w_i \notin L(\mathcal{M}_i)\}$$

$\Rightarrow L_d$ ist nicht rekursiv aufzählbar $\Rightarrow$ nicht entscheidbar.

Semi-entscheidbare Probleme

Halteproblem

Satz. *Es ist unentscheidbar, ob eine TM für jedes codierte Paar $(c(\mathcal{M}), w)$ als Eingabe hält oder nicht.*

$\mathcal{M}$ hält nur für Paare bei denen $w \in L_U$ ist.

$\Rightarrow \{L_{\text{rekursiv}}\} \subsetneq \{L_{\text{rekursiv aufzählbar}}\} \not\subset \{L_{\text{nicht rekursiv aufzählbar}}\}$

Game-of-Life

Die Frage, ob eine gegebene Anfangskonfiguration (w) zu ein stabilen oder periodischen Muster, oder dem kompletten Aussterben führt, kann nicht immer beantwortet werden.

Collatz-Zahlen

Gegeben sei die Collatz-Funktion col:

$$\mathrm{col}(n) = \begin{cases} n/2 & \text{falls } n \equiv 0 \pmod 2 \\ 3n+1 & \text{sonst} \end{cases}$$

Wendet man die Funktion auf das Ergebnis einer ursprüngliche Eingabe $n \in \mathbb{N}$ wiederholt an, entsteht die Collatz-Folge. Sie endet bei $n = 1$. Es ist unbekannt, ob die Menge der Collatz-Zahlen rekursiv ist.

Fleissige Biber

Ein fleissiger Biber ist eine Turingmaschine mit dem Alphabet $\Sigma = \{0, 1\}$ und n Zuständen, die hält und zuvor auf ein leeres (aus Nullen bestehendes) Band die maximale Anzahl k_n von Einsen schreibt, verglichen mit allen anderen haltenden Turingmaschinen mit ebenfalls n Zuständen (Nur Nicht-haltende-Turingmaschinen schreiben mehr einsen). Die Fleissige-Biber-Funktion ist definiert als $\sum(n) = k_n$. Es ist nicht entscheidbar, ob eine gegebene TM tatsächlich eine Kette maximaler Länge schreibt.

Reduktion

Technik um zu zeigen, dass ein Problem P_2 mindestens so schwer ist, wie ein bekanntes Problem P_1. Dazu wird jede Instanz von P_1 wird auf eine Instanz von P_2 mit gleicher Antwort überführt. $\Rightarrow P_1 \subset P_2$

1. P_1 ist unentscheidbar $\Leftrightarrow$ P_2 ist unentscheidbar
2. P_1 ist nicht rekursiv aufzählbar $\Leftrightarrow$ P_2 ist nicht rekursiv aufzählbar

Unentscheidbarkeit

Satz. *Jede nicht triviale Eigenschaft der rekursiv aufzählbaren Sprachen ist unentscheidbar.*

Eigenschaft einer rekursiv aufzählbaren Sprache ist eine Menge von rekursiv aufzählbaren Sprachen

trivial Entweder leer (d. h. kann keiner Sprache zugeordnet werden) oder aus allen rekursiv aufzählbaren Sprachen bestehend

Unentscheidbar sind u. a. folgende Fragen:

- ist eine von $\mathcal{M}$ akzeptierte Sprache endlich?
- ist eine von $\mathcal{M}$ akzeptierte Sprache kontextfrei?
- ist eine von $\mathcal{M}$ akzeptierte Sprache regulär?
- berechnet eine $\mathcal{M}$ eine gegebene Funktion?
- sind zwei gegebene Programme äquivalent?
- ist eine berechnete Funktion injektiv, surjektiv oder monoton?
- ist eine kfG eindeutig?

Eine Eigenschaft .

Komplexitätstheorie

(Kleene, Rabin, Scott, 70er Jahre)

Effizienz Die Effizienz eines Algorithmus bemisst sich nach Laufzeit und Speicherplatzbedarf

Laufzeiteffizienz wird über eine Funktion $f(n)$ in Abhängigkeit der Eingabewerte n beschreiben und bestimmt die obere Schranke

Laufzeitvergleiche

- müssen unabhängig von Soft- und Hardware sein
- betrachten besonders das Verhalten für alle grosse Eingaben
- ausschlaggebend ist der schlechteste Fall – die obere Schranke
- erfolgen mit Hilfe der
  - Ω-Notation: untere Schranke (d. h. mindestens so lang)
  - O-Notation: obere Schranke (d. h. höchstens so lang)

Ω-Notation

Die untere Schranke lässt sich nur schwer bestimmen, weil gezeigt werden muss, dass alle Algorithmen mindestens diese Komplexität besitzen. Formell:

$$\Omega(f(n)) = \{g : \mathbb{N} \to \mathbb{N} | \exists\, n_0 > 0 \wedge \exists\, c > 0 : \forall\, n \geq n_0 : |g(n)| \geq c \cdot |f(n)|\}$$

O-Notation

(*asymptotische Laufzeitkomplexität*) ermöglicht eine Abschätzung der Laufzeit bei unendlich grossen Eingaben. Formell:

$$\mathcal{O}(f(n)) = \{g : \mathbb{N} \to \mathbb{N} | \exists\, n_0 > 0 \wedge \exists\, c > 0 : \forall\, n \geq n_0 : |g(n)| \leq c \cdot |f(n)|\}$$

$\mathcal{O}(f(n))$ ist eine Menge von Funktionen, für die gilt: Es gibt zwei Konstanten c und n_0, so dass für alle $n > n_0$ gilt, $g(n)$ und $f(n)$ unterscheiden sich im Wachstum um den Faktor c.

Notation	Bedeutung	Beispiel
$g(n) \in \mathcal{O}(1)$	beschränkt	Array-Zugriff, Zuweisung
$g(n) \in \mathcal{O}(\log_x n)$	logarithmisch	binäre Suche
$g(n) \in \mathcal{O}(\sqrt[x]{n})$	Wurzelfunktion	„naiver“ Primzahlentest
$g(n) \in \mathcal{O}(n)$	linear	sequentielle Suche
$g(n) \in \mathcal{O}(n \cdot \log n)$	beschränkt	Mergesort
$g(n) \in \mathcal{O}(n^x)$	polynominell	Bubblesort
$g(n) \in \mathcal{O}(x^n)$	exponentiell	Erfüllbarkeit der Aussagenlogik
$g(n) \in \mathcal{O}(n!)$	faktoriell	Handlungsreisender

Algorithmen mit mindestens exponentiellen gelten als nicht praktikabel. Wenig praktikabel sind Algorithmen der Klasse $\mathcal{O}(n^x)$ für $x > 3$.

Rechenregeln

$\mathcal{O}(\mathcal{O}(f(n))) \in \mathcal{O}(f(n))$	Polynom vom Grad $x \in \mathcal{O}(n^x)$
$c + \mathcal{O}(f(n)) \in \mathcal{O}(f(n))$	$\mathcal{O}(f(n)) + \mathcal{O}(g(n)) \in \mathcal{O}(\max(f(n), g(n)))$
$c \cdot \mathcal{O}(f(n)) \in \mathcal{O}(f(n))$	$\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) \in \mathcal{O}((f(n) \cdot g(n)))$

Beispiel Bestimmung O-Notation

`function(int a, int b)`	$\Rightarrow 2 \cdot c_0 \Rightarrow \mathcal{O}(1)$
`  int x, y, z`	$\Rightarrow 3 \cdot c_1 \Rightarrow \mathcal{O}(1)$
`  for x = a to 0 do`	$\Rightarrow a \cdot (4 \cdot c_2) \Rightarrow \mathcal{O}(n)$
`    // vier Zuweisung/Berechnungen`	$\Rightarrow 4 \cdot c_2 \Rightarrow \mathcal{O}(1)$
`  end for`	
`  for x = b to a`	$\Rightarrow (b-a) \cdot (a \cdot (1 \cdot c_3) + 2 \cdot c_4 + 4 \cdot c_5) \Rightarrow \mathcal{O}(n^2)$
`    for y = a to 0`	$\Rightarrow a \cdot (1 \cdot c_3) \Rightarrow \mathcal{O}(n)$
`      z = z + 1`	$\Rightarrow 1 \cdot c_3 \Rightarrow \mathcal{O}(1)$
`    end for`	
`    // zwei Dateizugriffe`	$\Rightarrow 2 \cdot c_4 \Rightarrow \mathcal{O}(1)$
`    // vier Ausgabebefehle`	$\Rightarrow 4 \cdot c_5 \Rightarrow \mathcal{O}(1)$
`  end for`	
`end function`	

$\Rightarrow \mathcal{O}(1) + \mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(n^2) \in \mathcal{O}(n^2)$

Problemklassen

Einfaches Problem in polynominaler Zeit lösbar

Schwieriges Problem nur in mindestens exponentieller Zeit (d. h. nur für „kleine“ Eingabewerte) lösbar
Es gibt bei den schwierigen Problemen eine Unterklasse von Problemen, die mit einer nicht-deterministischen TM in polynominaler Zeit gelöst werden können (d. h. es ist schwer eine Lösung zu finden, aber einfach diese zu verifizieren)

P Eine Sprache L ist in der Klasse P enthalten, wenn $L = L(\mathcal{M})$ für eine deterministische TM $\mathcal{M}$ mit einer polynominalen Zeitkomplexität $T(n)$ gilt. D. h. $\mathcal{M}$ kann für $w \in \Sigma^*$ mit polynominalen Zeitaufwand prüfen, ob $w \in L$.

NP Eine Sprache L ist in der Klasse NP enthalten, wenn $L = L(\mathcal{M})$ für eine nichtdeterministische TM mit einer polynominalen Zeitkomplexität $T(n)$ gilt und für jede Eingabe der Länge n es keine Folge mit mehr als $T(n)$ Bewegungen in $\mathcal{M}$ gibt. D. h. $\mathcal{M}$ „rät“ die richtige Lösung und prüft diese mit polynominalen Zeitaufwand.
Offensichtlich gilt P $\subset$ NP. Die offene Frage ist:

$$\mathrm{P} \stackrel{?}{=} \mathrm{NP}$$

NP-schwierig Eine Sprache L heisst NP-schwierig, wenn für jede Sprache $L' \in$ NP eine Polynominalzeitreduktion von L' auf L existiert. D. h. L ist mindestens so schwierig, wie alle Probleme in NP (die übrigens alle gleich schwierig sind)

NP-vollständig (NPC) Eine Sprache L heisst NP-vollständig (NP-complete) wenn $L \in$ NP und für jede Sprache $L' \in$ NP eine Polynominalzeitreduktion von L' auf L existiert. D. h. im Gegensatz zu NP-schwierig existiert immerhin eine Lösung in NP.

Polynominalzeitreduktion Ein Algorithmus der ein Problem P_1 auf ein Problem P_2 reduziert darf höchstens polynominale Zeit verwenden. D. h. Der Algorithmus zum lösen von P_2 unterscheidet sich nur polynominal von dem von P_1. Darus folgt:

$$P_1 \in \mathrm{NP} \Rightarrow P_2 \in \mathrm{NP}$$

Beweistechnik $L \in$ NP

Via (transitiver) Polynominalzeitreduktion reduziert man ein Problem soweit, bis man bei einem bekannten NP-vollständigen Problem angelangt ist.

Bekannte NP-vollständige-Probleme

SAT Das „satisfiable“ Problem (Erfüllbarkeitsproblem): Gibt es eine Belegung der Variablen eines aussagelogischen Ausdrucks, so dass der Ausdruck wahr wird. (Satz von Cook, 1971)

TST Das Travelling Salesman Problem (kürzest mögliche Verbindung von verschiedenen Punkten)

„Färbungsproblem“ Wieviele Farben sind mindestens notwendig, um eine Landkarte so einzufärben, dass benachbarte Länder immer unterschiedliche Farben haben?

„Rucksack“-Problem Wie fülle ich einen Rucksack so, des Inhalts maximal, sein Gewicht aber minimal wird?

Lösungsalgorithmen sind für diese Probleme bekannt, doch sie erfordern exponentiellen Zeitaufwand. Es gelten:

1. jedes NP-vollständige Problem ist höchstens so schwierig wie das SAT-Problem
2. alle NP-vollständigen Probleme sind mindestens so schwierig wie das SAT-Problem

Lösungsstrategien für NP-vollständige Probleme

- Näherungsverfahren/Heuristiken, so dass
  - fast alle Lösungen optimal sind
  - alle Lösungen fast optimal sind
- Generierung von zufälligen Lösungen, bis man keine bessere mehr findet
- Optimierung des Basiswerts x, so dass $1 \leq x \leq 2$
- Exakte Algorithmen für kleine n
- „Lokale Suche“ (neu, noch ohne Relevanz)