Themenübersicht

1. Notation
2. Was ist eine Sprache?
3. Endliche Automaten und reguläre Sprachen, reguläre Ausdrücke
4. Stackautomaten und kontextfreie Grammatiken, Parser
5. Turing-Maschinen
6. Entscheidbarkeit
7. Komplexität
8. Turing-Vollständigkeit und Programmiersprachen

Was will *AutoSpr*?

Fragen

- Was kann ein Computer?
- Was ist ein Computer?
- Was sind Sprachen?
- Was kann ein Computer grundsätzlich nicht?
- Was kann ein Computer grundsätzlich nicht effizient?

Erwartete Antworten

Allgemeine (d. h. immer wahre) Aussagen über jede Art von Computer, unabhängig von Hardwarelimitierungen oder Architektur.

Was ist *wahr*?

Definition: Wahrheit

Aussagen, die mit der objektiven Realität übereinstimmen, sind wahr.

Konsequenzen

- Ohne Beobachtung der Realität keine Wahrheit (methodologischer Naturalismus)
- Subjektive Erfahrungen können nicht als "wahr" bezeichnet werden.

⇒ Es braucht einen Prozess, mit dem wir wahre von falschen Aussagen unterscheiden können.

Die wissenschaftliche Methode

Annahme

Es gibt eine objektive Realität, über die sich wahre Gesetzmässigkeiten formulieren lassen.

Von Interesse sind nur Aussagen, die grundsätzlich auch falsch sein können:

Falsifizierbarkeit

Eine Aussage ist falsifizierbar, wenn es eine Beobachtung gibt, mit der die Aussage als falsch erkannt werden kann.

Vorgehen

1. Formuliere eine falsifizierbare Hypothese über ein Naturphänomen
2. Formuliere die Nullhypothese, die besagt, dass die Hypothese nicht zutrifft
3. Beobachtung durchführen ⇒ Nullhypothese falsifizieren ⇒ Hypothese gilt als bestätigt

Computer

These

Computer sind abstrakte mathematische Objekte

- Kommen Computer in der Natur vor?
- Wir interessieren uns für allgemeine Eigenschaften aller von Menschen geschaffenen Computer.
- Computer sind nicht Objekte der Naturwissenschaft!
- Computer sind physisches Realisierungen eines rein mathematischen Konzeptes
- Die Evolution hat auch in der Natur Computer gebaut (zelluläre Automaten, neuronale Netzwerke, Gehirn)
- Wir suchen nach allgemein wahren Aussagen über abstrakte, mathematisch definierte Computer.

⇒ Was heisst *wahr* in der Mathematik?

Beweis

Definition

Ein *Beweis* ist in der Mathematik die als fehlerfrei anerkannte Herleitung der Richtigkeit bzw. der Unrichtigkeit einer Aussage aus einer Menge von Axiomen, die als wahr vorausgesetzt werden, und anderen Aussagen, die bereits bewiesen sind.

- Ein Beweis *erklärt*, warum eine Aussage wahr ist.
- Ein als korrekt erkannter *Algorithmus* ist ebenfalls ein Beweis.

⇒ Beweise sind ein wesentlicher Bestandteil der Vorlesung *AutoSpr*

Alphabet und Wort

Alphabet

Eine nichtleere endliche Menge Σ heisst *Alphabet*. Die Elemente von Σ heissen *Zeichen*

Wort

Eine Zeichenkette der Länge n ist ein n-Tupel in $\Sigma^n = \Sigma \times \cdots \times \Sigma$. Ein Element von Σ^n heisst *Wort* der Länge n.

Abgekürzte Schreibweisen:

$$(\mathtt{A},\mathtt{u},\mathtt{t},\mathtt{o},\mathtt{S},\mathtt{p},\mathtt{r}) = \mathtt{AutoSpr}$$
$$\mathtt{a}^3\mathtt{b}^5 = \mathtt{aaabbbbb}$$
$$\mathtt{b}^5\mathtt{a}^3 = \mathtt{bbbbbaaa}$$

Leeres Wort

Die Zeichenkette $\varepsilon \in \Sigma^0 = \{\varepsilon\}$ der Länge 0 heisst das *leere Wort*.

Menge aller Wörter

Menge aller Wörter

Die Menge aller Wörter ist

$$\Sigma^* = \{\varepsilon\} \cup \Sigma \cup \Sigma^2 \cup \Sigma^3 \cup \cdots = \bigcup_{k=0}^{\infty} \Sigma^k$$

Insbesondere haben Wörter immer endliche Länge.

Wortlänge

Definition

- $w \in \Sigma^n \Rightarrow |w| = n$, *Länge* des Wortes w.
- Sei $w \in \Sigma^n$ und $\mathtt{a} \in \Sigma$, dann ist $|w|_\mathtt{a}$ die Anzahl Zeichen $\mathtt{a}$ im Wort w.

Beispiele

- $|\varepsilon| = 0$,
- $|01010|_0 = 3$
- $|01010|_1 = 2$
- $|01010|_2 = 0$
- $|\mathtt{a}^3\mathtt{b}^5| = 8$
- $|(1291)^7| = 7|1291| = 7 \cdot 4 = 28$, $|w^n| = n \cdot |w|$

Sprache

Sprache

Eine Teilmenge $L \subset \Sigma^*$ heisst *Sprache*

Beispiele

- $L = \Sigma^* \subset \Sigma^*$
- $L = \emptyset \subset \Sigma^*$, die *leere Sprache*
- $\Sigma = \{0, 1\}$, Sprache aller Binärstrings: $L = \Sigma^*$
- $\Sigma = \{0, 1\}$, $L = \{w \in \Sigma^* \mid |w|_0 = |w|_1\}$.
- $\Sigma =$ Unicode, $J = \{w \in \Sigma^* \mid w$ wird vom Java-Compiler akzeptiert$\}$.
- $\Sigma =$ ASCII, $C = \{w \in \Sigma^* \mid w$ wird vom GCC akzeptiert$\}$.
- M eine "Maschine": $L(M) = \{w \in \Sigma^* \mid w$ wird von M akzeptiert$\}$.

Beobachtung

Zu jeder Maschine M gibt es eine Sprache $L(M)$

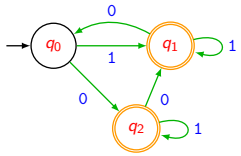
Deterministischer Endlicher Automat (DEA)

Definition

$A = (Q, \Sigma, \delta, q_0, F)$

- Zustände: $Q = \{q_0, q_1, \ldots, q_n\}$
- Alphabet: Σ
- Übergangsfunktion: $\delta \colon Q \times \Sigma \to Q$
- Startzustand: $q_0 \in Q$
- Akzeptierzustände: $F \subset Q$

Zustandsdiagramm von A



Tabellenform des DEA A

	0	1
q_0	q_2	q_1
q_1 /F	q_0	q_1
q_2 /F	q_1	q_2

(Q links, F links, Σ oben, δ rechts)

Reguläre Sprachen

Übergangsfunktion für Wörter

$$\delta \colon Q \times \Sigma^* \to Q : (q, w = a_1 \ldots a_n) \mapsto \delta(\ldots \delta(\delta(q, a_1), a_2), \ldots, a_n)$$

Übergänge ausgehend von q für Zeichen $a_1, a_2, \ldots, a_n$ nacheinander anwenden.

Wort akzeptieren

Der DEA $A = (\Sigma, Q, q_0, \delta, F)$ *akzeptiert* das Wort $w \in \Sigma^*$, wenn er A vom Startzustand in einen Akzeptierzustand $\delta(q_0, w) \in F$ überführt.

Akzeptierte Sprache, reguläre Sprache

Gegeben ein DEA $A = (\Sigma, Q, q_0, \delta, F)$. Die von A *akzeptierte Sprache* ist

$$L(A) = \{w \in \Sigma^* \mid A \text{ akzeptiert } w\} = \{w \in \Sigma^* \mid \delta(q_0, w) \in F\}.$$

Die Sprache $L \subset \Sigma^*$ heisst *regulär*, wenn es einen DEA A gibt mit $L(A) = L$.

Rekonstruiere DEA A einer regulären Sprache L

Definition

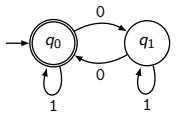
Für ein Wort $w \in \Sigma^*$ setze $L(w) = \{w' \mid ww' \in L\}$. Insbesondere $L(\varepsilon) = L$.

Myhill-Nerode Automat

Gegeben: reguläre Sprache L über Σ. Rekonstruiere A mit:

- $Q = \{L(w) \mid w \in \Sigma^*\}$
- $q_0 = L(\varepsilon) = L$
- $F = \{L(w) \in Q \mid \varepsilon \in L(w)\}$
- $\delta(L(w), a) = L(wa)$

w	$L(w)$	Q
ε	$L(\varepsilon) = L$	q_0
0	$L(0) = \{w \in \Sigma^* \mid \lvert w\rvert_0$ ungerade$\}$	q_1
1	$L(1) = \{w \in \Sigma^* \mid \lvert w\rvert_0$ gerade$\} = L$	q_0
$\vdots$	$\vdots$	$\vdots$



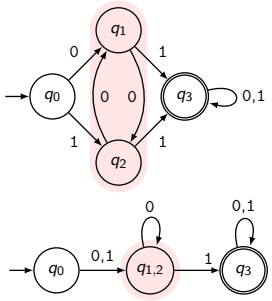
Beispiel

$\Sigma = \{0, 1\}$,
$L = \{w \in \Sigma^* \mid |w|_0$ gerade$\}$

Minimaler Automat

Ziel

Aus A einen DEA mit minimaler Zustandszahl konstruieren, z. B. für Vergleiche.



Markiere mit $\times$ Paare von Zuständen, die nicht äquivalent sein können:

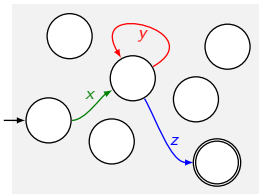
	q_0	q_1	q_2	q_3
q_0	$\equiv$	$\times$	$\times$	$\times$
q_1	$\times$	$\equiv$	$\equiv$	$\times$
q_2	$\times$	$\equiv$	$\equiv$	$\times$
q_3	$\times$	$\times$	$\times$	$\equiv$

$\times$: Akzeptier- und Nichtakzeptierzustände
$\times$: Nicht äquivalent nach Übergang
$\equiv$: Zustände q_1 und q_2 sind äquivalent und können zusammengelegt werden.

Pumping Lemma

Idee

Lange Wörter führen zu selbstkreuzenden Pfaden in einem endlichen Automaten. Was hat das für Folgen?



Lemma

Ist L eine reguläre Sprache, dann gibt es $N \in \mathbb{N}$, die pumping length so, dass jedes Wort $w \in L$ mit $|w| \geq N$ in drei Teile $w = xyz$ zerlegt werden kann mit

1. $|xy| \leq N$
2. $|y| > 0$
3. $xy^kz \in L \quad \forall k \in \mathbb{N}$

Oder: genügend lange Wörter ($|w| \geq N$) einer regulären Sprache ($w \in L$) können alle in einem Anfangsstück der Länge N ($|xy| \leq N$) aufgepumpt werden ($xy^kz \in L$).

Pumping-Lemma-Beweis

Behauptung

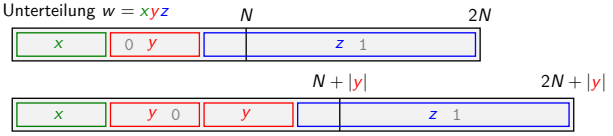
Die Sprache $L \subset \Sigma^*$ ist nicht regulär.

Beweis (Widerspruch)

1. Annahme: L ist regulär
2. Gemäss Pumping Lemma gibt es die Pumping Length N
   - Es darf keine Annahme über die konkrete Grösse von N gemacht werden!
3. Wähle ein Wort $w \in L$ mit $|w| \geq N$
   - Die Definition **muss** N verwenden!
4. Aufteilung des Wortes gemäss Pumping Lemma
   - $w = xyz$, $|xy| \leq N$, $|y| > 0$
5. Auswirkung des Pumpens
   - $xy^kz \notin L$ für mindestens ein $k \in \mathbb{N}$ (mit Begründung!)
6. Widerspruch und Schlussfolgerung, dass die Annahme nicht zutreffen kann

Beispiel: $L = \{0^n1^n \mid n \geq 0\}$ ist nicht regulär

1. Annahme: L ist regulär
2. $\exists N \in \mathbb{N}$, Pumping Length
3. $w = 0^N1^N$
4. Unterteilung $w = xyz$



5. Pumpen: nur die Anzahl der 0 wird erhöht, Anzahl 1 bleibt
6. $xy^kz \notin L$ für $k \neq 1$, im Widerspruch zum Pumping Lemma

Nichtdeterministische endliche Automaten

Definition NEA

$A = (Q, \Sigma, \delta, q_0, F)$

- Q endliche Menge von Zuständen
- Σ Alphabet
- $\delta : Q \times \Sigma \to P(Q)$
- Startzustand q_0
- Akzeptierzustände F

Definition Akzeptieren

Ein NEA A akzeptiert das Wort $w \in \Sigma^*$, wenn es eine Wahl von Übergängen gibt, derart, dass das Wort w den Automaten in einen Akzeptierzustand überführt.

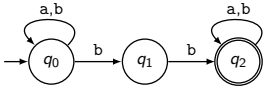
Faustregel

Nur genau diejenigen Pfeile einzeichnen, die man zum Akzeptieren braucht.

Beispiel

$\Sigma = \{a, b\}$,

$$L = \left\{ w \in \Sigma^* \;\middle|\; \begin{array}{l} w \text{ enthält zwei b} \\ \text{nacheinander} \end{array} \right\}$$



Transformation NEA → DEA

Übergangsfunktion für Mengen

Gegeben $\delta\colon Q \times \Sigma \to P(Q)$ eines NEA. Übergangsfunktion für Mengen $M \subset Q$

$$\delta'\colon P(Q) \times \Sigma \to P(Q) : (M, a) \mapsto \delta'(M, a) = \bigcup_{q \in M} \delta(q, a)$$

Satz

Ein NEA A kann in einen DEA A' umgewandelt werden mit

- $Q' = P(Q)$
- $\Sigma' = \Sigma$
- δ' *wie oben definiert*
- $q_0' = \{q_0\}$
- $F' = \{M \in P(Q) \mid F \cap M \neq \emptyset\}$

Implementation

Thompson-NEA:

- Zustand $M \subset Q$ realisiert durch Markierung der Zustände in M
- δ' realisiert durch Verschieben von Markierungen
- Akzeptieren: mindestens ein Akzeptierzustand markiert

ε-Übergänge: NEA$_\varepsilon$

ε-Übergang

Übergangsfunktion mit erweitertem Definitionsbereich

$$\delta\colon Q \times (\Sigma \cup \{\varepsilon\}) \to P(Q)$$

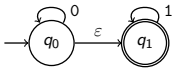
ε-Übergänge können ohne Verarbeitung eines Zeichens genommen werden.

NEA$_\varepsilon$

NEA mit ε-Übergängen

Beispiel

$L = \{0^k 1^l \mid k, l \geq 0\}$



Satz

Einen NEA$_\varepsilon$ kann man immer in einen NEA umwandeln.

Beweis.

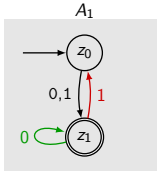
1. $E(q) =$ Menge der von q aus mit ε-Übergängen erreichbaren Zustände
2. $E(M) = \cup_{q \in M} E(q)$
3. δ ersetzen durch $\delta\colon Q \times (\Sigma \cap \{\varepsilon\}) \to P(Q) : (q, a) \mapsto E(\delta(q, a))$

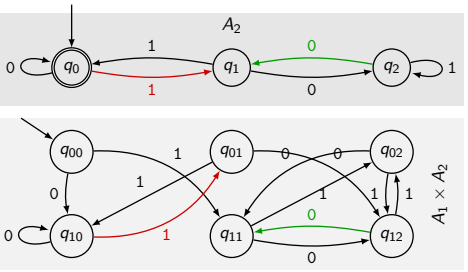
□

Mengenoperationen

Produktautomat

$L_i = L(A_i)$





Schnittmenge $L_1 \cap L_2$
$F = F_1 \times F_2$

Vereinigung $L_1 \cup L_2$
$F = F_1 \times Q_2 \cup Q_1 \times F_2$

Differenz $L_1 \setminus L_2$
$F = F_1 \times (Q_2 \setminus F_2)$

Reguläre Operationen

Alternative

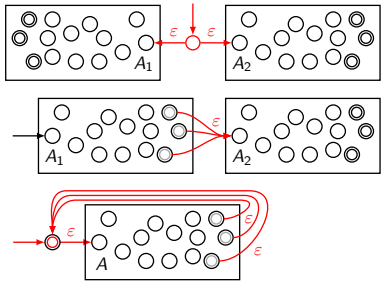
$$L = L_1 \cup L_2 = L(A_1) \cup L(A_2)$$

Verkettung

$$L = L_1 L_2 = L(A_1)L(A_2) = \{w_1 w_2 \mid w_i \in L_i\}$$

$*$-Operation

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup \ldots = \bigcup_{k=0}^{\infty} L^k$$



⇒ die Klasse der regulären Sprachen ist abgeschlossen unter regulären Operationen

Reguläre Ausdrücke und VNEA

Regulärer Ausdruck

Zeichenkette r zur Beschreibung einer regulären Sprache $L = L(r)$.

Primitive reguläre Ausdrücke

Reguläre Ausdrücke für Wörter mit Länge ≤ 1:

$L = L(r)$	r	NEA
$\emptyset$	$\emptyset$	
$\{\varepsilon\}$	ε	
$\{a\}$	a	
$\{h, s, r\}$	[hsr]	
$\{a, b, \ldots, s\}$	[a-s]	

Reguläre Operationen

$$L(r_1) \cup L(r_2) = L(r_1 | r_2)$$
$$L(r_1)L(r_2) = L(r_1 r_2)$$
$$L(r_1)^* = L(r_1 *)$$

Abkürzungen

$r+ = rr*$ $\qquad r? = \varepsilon | r = |r$

$r\{2\} = rr$ $\qquad r\{2,3\} = rr|rrr$

$r\{,3\} = |r|rr|rrr$ $\qquad r\{3,\} = rrrr*$

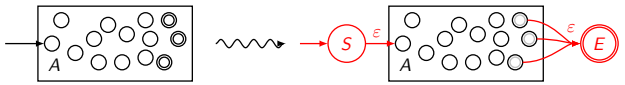
Verallgemeinerter NEA

NEA$_\varepsilon$, dessen Übergänge mit regulären Ausdrücken beschriftet sind.

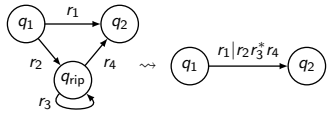
⇒ zu jedem regulären Ausdruck gibt es einen DEA

VNEA A umwandeln in Regex r

Keine Übergänge nach q_0 und nur ein Akzeptierzustand

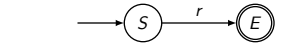


Zustand q_{rip} entfernen $\forall q_1, q_2 \in Q$



Regulärer Ausdruck

Nach Entfernen aller Zwischenzustände $q_{\text{rip}} \in Q$ bleibt ein regulärer Ausdruck r von A



⇒ jede reguläre Sprache lässt sich mit einem regulären Ausdruck beschreiben

$$\{L(A) \mid A \text{ ein DEA}\} = \{L(r) \mid r \text{ ein regulärer Ausdruck}\}$$

Kontextfreie Grammatik und Sprache

Kontextfreie Grammatik: $G = (V, \Sigma, R, S)$

- V: Variablen
- Σ: Terminalsymbole (Alphabet)
- R: Regeln der Form $A \to x_1 x_2 \ldots x_n$ mit $A \in V$, $x_i \in V \cup \Sigma$
- S: Startvariable

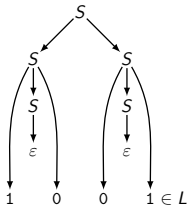
Ableitung und erzeugte Sprache

- Regel $A \to w$ erzeugt aus uAv das Wort uwv: $uAv \Rightarrow uwv$
- v aus u ableiten: $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_n \Rightarrow v$ oder $u \overset{*}{\Rightarrow} v$
- $L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$ von G erzeugte *kontextfreie Sprache*

Parse-Tree

$L = \{w \in \{0,1\}^* \mid |w|_0 = |w|_1\}$ mit der Grammatik

$$S \to 0S1 \mid 1S0 \mid SS \mid \varepsilon$$



Reguläre Operationen

Grammatik für reguläre Operationen

L_1 und L_2 kontextfreie Sprachen mit Grammatiken $G_i = (V_i, \Sigma, R_i, S_i)$.
Grammatik für reguläre Operationen:

- Neue Startvariable S_0
- $V = V_1 \cup V_2 \cup \{S_0\}$
- geeignet erweiterte Regeln R

⇒ $G = (V, \Sigma, R, S_0)$

Satz

Die Klasse der kontextfreien Sprachen ist abgeschlossen unter regulären Operationen.

Alternative

Regeln für $L_1 \cup L_2$:

$$R = R_1 \cup R_2 \cup \{S_0 \to S_1, S_0 \to S_2\}$$

Verkettung

Regeln für $L_1 L_2$:

$$R = R_1 \cup R_2 \cup \{S_0 \to S_1 S_2\}$$

$*$-Operation

Regeln für L_1^*:

$$R = R_1 \cup \{S_0 \to S_0 S_1, S_0 \to \varepsilon\}$$

Chomsky-Normalform

Definition

Eine CFG ist in *Chomsky-Normalform* (CNF), wenn S auf der rechten Seite nicht vorkommt und jede Regel von der Form $A \to BC$ oder $A \to a$ ist, zusätzlich ist die Regel $S \to \varepsilon$ erlaubt.

Umwandlung in Chomsky-Normalform

1. Neue Startvariable $S_0 \to S$
2. ε-Regeln: $\left.\begin{matrix} A \to \varepsilon \\ B \to AC \end{matrix}\right\} \Rightarrow$ A kann weggelassen werden $\Rightarrow \begin{cases} B \to AC \\ \quad \to C \end{cases}$
3. Unit-Rules: $\left.\begin{matrix} A \to B \\ B \to CD \end{matrix}\right\} \Rightarrow$ aus A kann man wie aus B auch CD machen $\Rightarrow \begin{cases} A \to CD \\ B \to CD \end{cases}$
4. Verkettungen: $A \to u_1 u_2 \ldots u_n$ ersetzen durch $A \to u_1 A_1, A_1 \to u_2 A_2, \ldots, A_{n-2} \to u_{n-1} u_n$ und falls u_i ein Terminalsymbol ist: $A_{i-1} \to U_i A_i, U_i \to u_i$.

Deterministisches Parsen

Aufgabe: $S \overset{*}{\Rightarrow} w$

Kann das Wort $w \in \Sigma^*$ von der Grammatik $G = (V, \Sigma, R, S)$ erzeugt werden?

Verallgemeinerte Aufgabe: $A \overset{*}{\Rightarrow} w$

Kann das Wort $w \in \Sigma^*$ aus der Variablen A in der Grammatik $G = (V, \Sigma, R, S)$ abgeleitet werden?

Deterministische Antwort:

Gesucht ist ein Programm mit der Signatur

```
boolean  ableitbar(Variable A, String w);
```

welches entscheiden kann, ob ein Wort `w` aus der Variablen `V` ableitbar ist.

Lösung

Für eine Grammatik in Chomsky-Normalform machbar dank einfacher Regeln.

CYK-Ideen

Gegeben

1. Grammatik $G = (V, \Sigma, R, S)$
2. Variable $A \in V$
3. Wort $w \in \Sigma^*$

Frage

Ist w aus A ableitbar? in Zeichen: $A \overset{*}{\Rightarrow} w$

- Spezialfall $w = \varepsilon$: $A \overset{*}{\Rightarrow} \varepsilon \quad \Leftrightarrow \quad A \to \varepsilon \in R$
- Spezialfall $|w| = 1$: $A \overset{*}{\Rightarrow} w \quad \Leftrightarrow \quad A \to w \in R$
- Fall $|w| > 1$:

$$A \overset{*}{\Rightarrow} w \quad \Rightarrow \quad \exists \begin{cases} A \to BC & \in R \\ w = w_1 w_2 & w_i \in \Sigma^* \end{cases} \quad \text{mit} \quad \begin{cases} B \overset{*}{\Rightarrow} w_1 \\ C \overset{*}{\Rightarrow} w_2 \end{cases}$$

CYK-Algorithmus

```
boolean ableitbar(variable A, String w) {
    if (w.length() == 0) {
        return A → ε ∈ R;                                  Fall: w = ε
    }
    if (w.length() == 1) {
        return A → w ∈ R;                                  Fall: w ein Terminalsymbol
    }
    foreach Unterteilung w = w1w2 {                        Fall: |w| > 1
        foreach A → BC ∈ R {
            return ableitbar(B, w1) && ableitbar(C, w2);
        }
    }
    return false;                                          sonst
}
```

Stackautomat

Definition

Stackautomat $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$

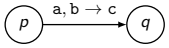
1. Q: Zustände
2. Σ: Eingabe-Alphabet
3. Γ: Stack-Alphabet
4. δ: $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to P(Q \times \Gamma_\varepsilon)$
5. $q_0 \in Q$: Startzustand
6. $F \subset Q$: Akzeptierzustände

$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$, $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$

Beachte

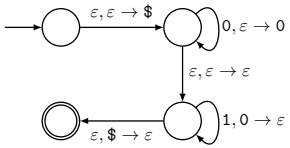
- immer nicht-deterministisch.
- $\Gamma \neq \Sigma$ möglich

Übergänge



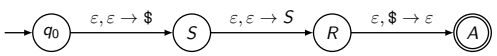
Verarbeite Input a und ersetze b auf dem Stack durch c.

Beispiel

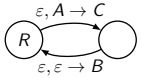


Grammatik → Stackautomat

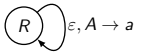
Grundgerüst



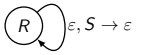
Regel $A \to BC$



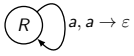
Regel $A \to a$



Regel $S \to \varepsilon$



$\forall a \in \Sigma$



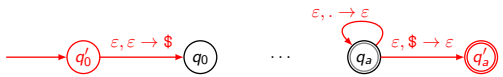
Satz

Ist L eine kontextfreie Sprache, dann gibt es einen Stackautomaten P, der L akzeptiert, $L = L(P)$.

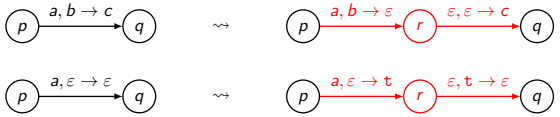
Stackautomat standardisieren

1. Nur ein Akzeptierzustand: neuer Akzeptierzustand q_a und Übergänge

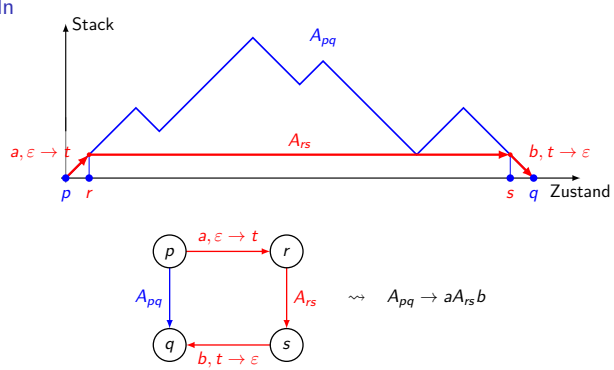
2. Stack leeren:



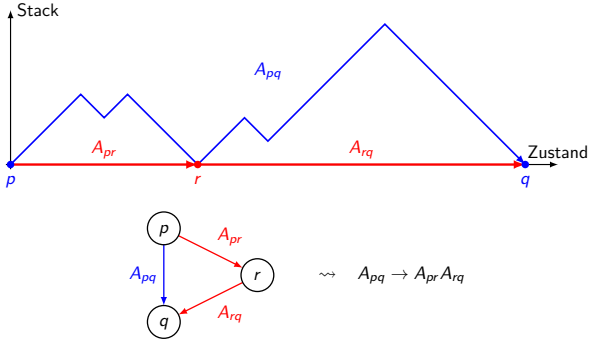
3. Jeder Übergang legt entweder ein Zeichen auf den Stack oder entfernt eines:



Regeln



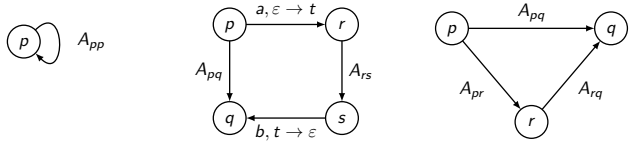
Regeln



Grammatik ablesen

Ausgangspunkt: standardisierte Grammatik mit Startzustand q_0 und $F = \{q_a\}$.

1. Startvariable: A_{q_0,q_a}
2. Regeln:



$A_{pp} \to \varepsilon$ $\quad A_{pq} \to aA_{rs}b$ $\quad A_{pq} \to A_{pr}A_{rq}$

Pumping Lemma (Herleitung)

Grammatik G in CNF

$$w \in L(G) \quad \Rightarrow \quad S \overset{*}{\Rightarrow} w$$

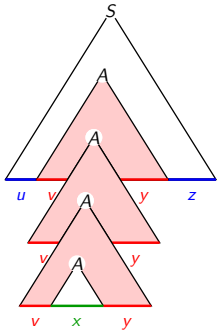
Wiederverwendete Variable

$|w| \geq N$ gross genug $\Rightarrow$ Variablen werden im Parse-Tree wiederverwendet
Die "unterste" wiederverwendete Variable A erzeugt zwei Wörter:

$$A \overset{*}{\Rightarrow} vxy$$
$$A \overset{*}{\Rightarrow} x$$

Pumpen

$A \overset{*}{\Rightarrow} vxy$ anstelle des "untersten" $A \overset{*}{\Rightarrow} x$ verwenden



Pumping Lemma

Pumping Lemma für CFL

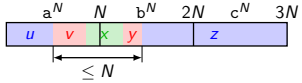
Ist L eine CFL, dann gibt es eine Zahl N, die Pumping Length, derart, dass jedes Wort $w \in L$ mit $|w| \geq N$ zerlegt werden kann in fünf Teile $w = uvxyz$ derart, dass

1. $|vy| > 0$
2. $|vxy| \leq N$
3. $uv^k xy^k z \in L \; \forall k \in \mathbb{N}$

Mit dem Pumping Lemma kann man beweisen, dass eine Sprache nicht kontextfrei ist.

Beispiel: $\{a^n b^n c^n \mid n \geq 0\}$

1. Annahme: L kontextfrei
2. Pumping length N
3. Wort: $w = a^N b^N c^N$
4. Zerlegung:



5. Beim Pumpen nimmt die Anzahl der a und b zu, nicht aber die Anzahl der c $\Rightarrow uv^k xy^k z \notin L \forall k \neq 1$
6. Widerspruch: L nicht kontextfrei

Abzählbar und überabzählbar unendlich

Definition

Mengen A und B heissen *gleich mächtig*, $A \simeq B$, wenn es eine Bijektion $A \to B$ gibt.

Definition

Eine Menge A heisst *unendlich*, wenn sie gleich mächtig wie eine echte Teilmenge ist.

Definition

A heisst *abzählbar unendlich*, wenn $A \simeq \mathbb{N}$

Voraussetzungen

A, B, A_k abzählbar unendlich, $k \in \mathbb{N}$

Resultate

1. $A \cup B$, $\bigcup_k A_k$ abzählbar unendlich
2. $A \times B$ abzählbar unendlich
3. Abzählbar unendlich: $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$
4. $P(A)$ überabzählbar unendlich
5. $\mathbb{R}$ überabzählbar unendlich

Anwendungen

- Abzählbar unendlich: Σ^*, Menge aller DEAs/NEAs/PDAs/CFGs
- Überabzählbar unendlich: Menge aller Sprachen $P(\Sigma^*)$

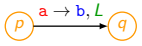
Turing Maschine (TM)

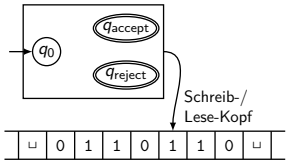
Definition: (deterministische) Turing Maschine

$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$

- Q Zustände
- Σ Alphabet
- Γ Bandalphabet, $\sqcup \in \Gamma \setminus \Sigma$
- $\delta\colon Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
- $q_0 \in Q$ Startzustand
- $q_{\text{accept}} \in Q$ Akzeptierzustand
- $q_{\text{reject}} \in Q$ Ablehnungszustand

Zustandsdiagramm

$\delta(p, a) = (q, b, L)$: 

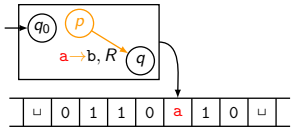


Übergang

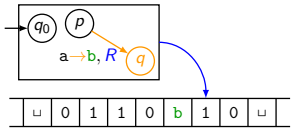
- Übergang möglich, wenn a unter dem Schreib-/Lese-Kopf
- Aktuelles Feld auf dem Band wird mit b überschrieben
- Kopfbewegung: L links, R rechts

Arbeitsweise einer Turing-Maschine

Vorher



Nachher



Programmablauf

- Input-Wort w auf Band
- Schreib-/Lesekopf positioniert auf 1. Zeichen
- Maschine starten, $t(w)$ Einzelschritte ausführen
- Maschine hält in q_{accept} oder q_{reject}.
  q_{accept}: Wort w akzeptiert
  q_{reject}: Wort w verworfen

Laufzeit: $t(w)$, $t(n) = \max\{t(w) \mid |w| \leq n\}$

Von einer TM erkannte Sprache

M eine Turing-Maschine

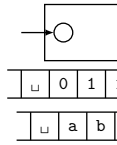
$$L(M) = \{w \in \Sigma^* \mid M \text{ akzeptiert } w\}$$

Varianten

Anderes Bandalphabet

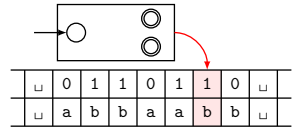
Bandalphabet Γ binär codieren, d. h. mit Zeichen aus $\Gamma_0 = \{0, 1\}$
Simulierbar in Zeit $O(t(n))$

Mehrere Bänder



Simulierbar in Zeit $O(t(n)^2)$

Mehrspurige Turing-Maschine



Simulierbar in Zeit $O(t(n))$

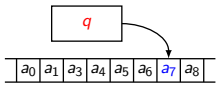
Nicht deterministische TM

$$\delta\colon Q \times \Gamma \to P(Q \times \Gamma \times \{L, R\})$$

Simulierbar in $2^{O(t(n))}$

Berechnungsgeschichte

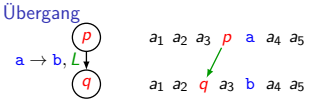
Notation für Maschinenzustand



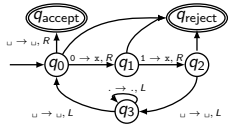
protokolliert als

$a_0 a_1 a_2 a_3 a_4 a_5 a_6 q a_7 a_8$

Übergang



Protokoll einer Berechnung



⊔ q_0 0 0 1 1 ⊔
⊔ x q_1 0 1 1 ⊔
⊔ x 0 q_1 1 1 ⊔
⊔ x 0 x q_2 1 ⊔
⊔ x 0 x 1 q_2 ⊔
⊔ x 0 x q_3 1 ⊔
⊔ x 0 q_3 x 1 ⊔
⊔ x q_3 0 x 1 ⊔
⊔ q_3 x 0 x 1 ⊔
...

Nichtdeterministische TM

Übergangsfunktion

Bei jedem Übergang maximal N verschiedene Möglichkeiten:

$$\delta\colon Q \times \Gamma \to P(Q \times \Gamma \times \{L, R\})$$

$\Rightarrow$ Mehrere mögliche Berechungswege

Wort akzeptieren

$w \in L(M)$ wenn es einen Berechnungsweg gibt, der zu q_{accept} führt.
(auch wenn es Wege gibt, die auf q_{reject} führen!)

Simulationsidee

Alle Berechnungswege durchführen, maximal $N^{t(n)}$

Simulation auf Standard-TM

Verwende 3 Bänder:

1. Arbeitsband
2. Kopie von w
3. Liste aller Folgen von Wahlmöglichkeiten

Simulation

1. Kopiere w von Band 2 auf Band 1
2. Führe TM aus auf Band 1 mit Wahlmöglichkeiten von Band 3
3. Inkrementiere zur nächsten Folge von Wahlmöglichkeiten auf Band 3

Entscheider und entscheidbare Sprachen

Deterministisch

Eine TM M heisst *Entscheider*, wenn sie auf jedem Input w anhält.

Nichtdeterministisch

Eine nichtdeterministische TM M heisst *Entscheider*, wenn jede Berechnungsgeschichte terminiert.

Turing-erkennbare Sprache

L heisst *Turing-erkennbar*, wenn es eine TM M gibt mit $L = L(M)$.

Turing-entscheidbare Sprache

L heisst *Turing-entscheibar*, wenn es einen Entscheider M gibt mit $L = L(M)$.

Ist $w \in L(M)$?

Wie unterscheidet man, ob M auf Input w *nicht* anhält oder einfach noch etwas länger braucht?

Ist $w \in L(M)$?

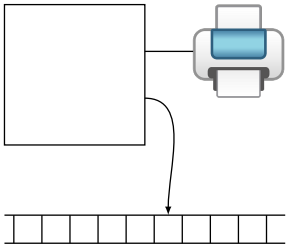
M wird auf Input w garantiert anhalten, Geduld...

Aufzähler

Definition
Ein *Aufzähler* ist eine TM mit einem Drucker, mit dem Wörter ausgedruckt werden können

Rekursiv aufzählbare Sprache
L ist rekursiv aufzählbar wenn es einen Aufzähler gibt, der sie aufzählt

Satz
Aufzählbare Sprache $\Leftrightarrow$ Turing-erkennbare Sprache



David Hilbert

- * 1862 Königsberg
- † 1943 Göttingen
- 1886 Habilitation in Königsberg
- 1895 Berufung nach Göttingen
- 1900 Vortrag am internationalen Kongress
- Beiträge zur Algebraischen Geometrie, Zahlentheorie, Geometrie, Mathematischen Physik
- 1915 allgemeine Relativitätstheorie (unabhängig von Einstein)
- Logik und Grundlagen der Mathematik: Ausweg aus der Grundlagenkrise

Hilberts Traum

Non ignorabimus
. . . denn in der Mathematik gibt es kein Ignorabimus

Programm

- Stelle die Mathematik auf eine lückenlose axiomatische Grundlage
- Beweise die Widerspruchsfreiheit der Axiome
- Von jeder Aussage kann man beweisen, ob sie war oder falsch ist

Epitaph
Wir müssen wissen, wir werden wissen

Kurt Gödel

- * 1906 Brünn
- † 1978 Princeton
- 1924 Studium der theoretischen Physik in Wien
- 1930 Dissertation *Über die Vollständigkeit des Logikkalküls*
- 1931 Unvollständigkeitssätze
- 1949 Allgemeine Relativitätstheorie: Gödel-Universum ermöglicht Zeitreisen
- 1956 P-NP-Problem

Gödels Unvollständigkeitssätze

Erster Unvollständigkeitssatz
Jedes hinreichend mächtige . . . formale System ist entweder widersprüchlich oder unvollständig

Zweiter Unvollständigkeitssatz
Jedes hinreichend mächtige konsistente formale System kann die eigene Konsistenz nicht beweisen

$\Rightarrow$ Hilberts Programm ist undurchführbar!

Alan Mathison Turing

- * 1912 London
- † 1954 Wilmslow
- 1928 Turing liest und versteht die Arbeiten Albert Einsteins
- 1931 Studium am King's College
- 1936 *On Computable Numbers, with an Application to the "Entscheidungsproblem"*
- 1953 Turing-Test, künstliche Intelligenz

Berechenbare Zahlen

Berechnung einer Zahl w
Eine Turingmaschine M berechnet die Zahl w, indem sie auf leerem Band startet und nach dem Anhalten die Zahl w auf dem Band zurücklässt

Definition
Eine Zahl w ist berechenbar, wenn es eine Turing-Maschine gibt, die sie berechnet

Beispiel
π, e, $\sqrt{2}$, γ, $\varphi = \frac{\sqrt{5}-1}{2}$

Wieviele berechenbare Zahlen?
Sind die reellen Zahlen berechenbar?

1. Es gibt höchstens so viele berechenbare Zahlen wie Turing-Maschinen
2. Die Menge der Turing-Maschinen ist abzählbar unendlich
3. Die Menge der reellen Zahlen $\mathbb{R}$ ist überabzählbar unendlich
4. $\Rightarrow$ die "meisten" reellen Zahlen sind nicht berechenbar

Es gibt nur abzählbar unendlich viele Spezifikationen von reellen Zahlen

Hilberts 10. Problem

Diophantische Gleichungen
Polynomgleichungen mit ganzzahligen Koeffizienten

$$x^2 - y = 0 \qquad x^4 + y^2 + z^{20} = -7$$
$$x^2 + y^2 = z^2 \qquad x^n + y^n = z^n$$

Gibt es ganzzahlige Lösungen?

Das 10. Problem
Hilberts Ansprache am 1. ICM 1900 in Paris: *Gibt es einen Algorithmus, mit dem man entscheiden kann, ob eine diophantische Gleichung eine Lösung hat?*

Antwort: Nein!
Yuri Matiyasevich 1970

Weitere nicht entscheidbare Probleme:

- Funktionen mehrerer Variablen aus arithmetischen Operationen und $\sin x$
- Gleichheit von algebraischen Ausdrücken
- Lösbarkeit allgemeiner Differentialgleichungssysteme

Informatik?
Relevante nicht entscheidbare Probleme der Informatik?

Entscheidbarkeit

Definition
Ein *Entscheider* ist eine Turing-Maschine, die auf jedem beliebigen Input anhält.

Definition
Eine Sprache L heisst *entscheidbar*, wenn es einen Entscheider M gibt mit $L = L(M)$. Man sagt, M *entscheidet* L.

Sprachproblem
Jedes Problem P kann in ein Sprachproblem übersetzt werden:

$$L_P = \left\{ w \in \Sigma^* \;\middle|\; \begin{array}{l} w \text{ ist Lösung des} \\ \text{Problems } P \end{array} \right\}$$

Beispiele
Leerheitsproblem:

$$E_{\text{DEA}} = \left\{ \langle A \rangle \;\middle|\; \begin{array}{l} A \text{ ein DEA und} \\ L(A) = \emptyset \end{array} \right\}$$

Gleichheitsproblem:

$$EQ_{\text{CFG}} = \left\{ \langle G_1, G_2 \rangle \;\middle|\; \begin{array}{l} G_i \text{ CFGs und} \\ L(G_1) = L(G_2) \end{array} \right\}$$

Akzeptanzproblem:

$$A_{\text{DEA}} = \left\{ \langle A, w \rangle \;\middle|\; \begin{array}{l} A \text{ ein DEA, der} \\ w \text{ akzeptiert} \end{array} \right\}$$

Halteproblem:

$$HALT_{\text{TM}} = \left\{ \langle M, w \rangle \;\middle|\; \begin{array}{l} M \text{ hält auf} \\ \text{Input } w \end{array} \right\}$$

Halteproblem

Theorem (Alan Turing)

A_{TM} ist nicht entscheidbar.

Beweis.

Konstruiere aus einem Entscheider H für A_{TM} eine Maschine D mit Input $\langle M \rangle$

1. Lasse H auf Input $\langle M, \langle M \rangle \rangle$
2. Falls H akzeptiert: q_{reject}
3. Falls H verwirft: q_{accept}

Wende jetzt D auf $\langle D \rangle$ an:

$D(\langle D \rangle)$ akzeptiert $\Leftrightarrow$ D verwirft $\langle D \rangle$
$D(\langle D \rangle)$ verwirft $\Leftrightarrow$ D akzeptiert $\langle D \rangle$

Widerspruch! □

Spezielles Halteproblem

Das spezielle Halteproblem

$$HALT\varepsilon_{\text{TM}} = \left\{ \langle M \rangle \,\middle|\, \begin{array}{l} M \text{ ist eine Turingmaschine und } M \text{ hält auf} \\ \text{leerem Band} \end{array} \right\}$$

ist nicht entscheidbar

Halteproblem

Das allgemeine Halteproblem

$$HALT_{\text{TM}} = \left\{ \langle M, w \rangle \,\middle|\, \begin{array}{l} M \text{ ist eine Turing-Maschine und } M \text{ hält} \\ \text{auf Input } w \end{array} \right\}$$

ist nicht entscheidbar.

Reduktion

Reduktionsabbildung

Berechenbare Abbildung $f : \Sigma^* \to \Sigma^*$ so, dass

$$w \in A \quad \Leftrightarrow \quad f(w) \in B$$

Notation: $f : A \leq B$, "A leichter entscheidbar als B"

Entscheidbarkeit

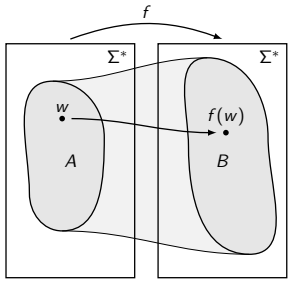
B entscheidbar, $f : A \leq B \Rightarrow A$ entscheidbar

Beweis.

H ein Entscheider für B, dann ist $H \circ f$ ein Entscheider für A. □

Folgerung

A nicht entscheidbar, $A \leq B \Rightarrow B$ nicht entscheidbar



Entscheider $H \circ f$ Entscheider H

Reduktion für das Halteproblem: $A_{\text{TM}} \leq HALT_{\text{TM}}$

A_{TM} ist nicht entscheidbar

Akzeptiert die Maschine M das Wort w?
Es gibt keinen Entscheider

$\langle M, w \rangle \quad \mapsto$

M akzeptiert $w \quad \Leftrightarrow$

Entscheider für A_{TM}

1. Konstruiere das Programm S
2. Wende H auf $\langle S, w \rangle$

Ist $HALT_{\text{TM}}$ entscheidbar?

Hält die Maschine M auf dem Wort w an?

Programm S mit Input w

1. Führe M auf Inputwort w aus
2. M hält in q_{accept}: akzeptiere
3. M hält in q_{reject}: Endlosschleife

S hält auf w

Entscheider für $HALT_{\text{TM}}$

Wenn H ein Entscheider für $HALT_{\text{TM}}$ wäre, könnte man daraus einen Entscheider für A_{TM} konstruieren

Vergleich der Entscheidbarkeit von Sprachen

Reduktion

Die Sprache A ist auf B *reduzierbar*, $A \leq B$, wenn es eine berechenbare Abbildung $f : \Sigma^* \to \Sigma^*$ gibt mit

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

Lesung: $A \leq B$ heisst A ist leichter entscheidbar als B.

Theorem

$A \leq B$, B entscheidbar, dann ist A entscheidbar.

Theorem

$A \leq B$, A nicht entscheidbar, dann ist B nicht entscheidbar.

Reduktion $A_{\text{TM}} \leq HALT_{\text{TM}}$

Aus $\langle M, w \rangle$ konstruiere eine neue Maschine S mit Input w:

1. Lasse M auf w laufen
2. Falls w akzeptiert wird: q_{accept}
3. Andernfalls: Endlosschleife

S hält auf w genau dann, wenn M das Wort w akzeptiert:

$$\langle M, w \rangle \mapsto \langle S, w \rangle$$

ist eine Reduktion

$$A_{\text{TM}} \leq HALT_{\text{TM}}$$

Reduktion für das Leerheitsproblem: $A_{\text{TM}} \leq E_{\text{TM}}$

A_{TM} ist nicht entscheidbar

Akzeptiert die Maschine M das Wort w?
Es gibt keinen Entscheider

$\langle M, w \rangle \quad \mapsto$

M akzeptiert $w \quad \Leftrightarrow$

Entscheider für A_{TM}

1. Konstruiere das Programm S
2. Wende H auf $\langle S \rangle$ an

Ist E_{TM} entscheidbar?

Ist die Sprache $L(M)$ leer? $\to \bar{E}_{\text{TM}}$ ist $L(M) \neq \emptyset$?

Programm S mit Input u

1. M auf w laufen lassen
2. M akzeptiert w: q_{accept}
3. q_{reject}

S akzeptiert $L(S) = \Sigma^* \neq \emptyset$

Entscheider für E_{TM}

Wenn H ein Entscheider für E_{TM} wäre, könnte man daraus einen Entscheider für A_{TM} konstruieren

Regularitätsproblem: $A_{\text{TM}} \leq REGULAR_{\text{TM}}$

A_{TM} ist nicht entscheidbar

Akzeptiert die Maschine M das Wort w?
Es gibt keinen Entscheider

$\langle M, w \rangle \quad \mapsto$

M akzeptiert $w \quad \Leftrightarrow$
M akzeptiert w nicht $\Leftrightarrow$

Entscheider für A_{TM}

1. Konstruiere das Programm S
2. Wende H auf $\langle S \rangle$ an

Ist $REGULAR_{\text{TM}}$ entscheidbar?

Ist die Sprache $L(M)$ regulär?

Programm S mit Input u

1. $u \notin \{0^n 1^n \mid n \geq 0\} \to q_{\text{reject}}$
2. M auf w laufen lassen
3. M akzeptiert w: q_{accept}
4. q_{reject}

S akz. $\{0^n 1^n \mid n \geq 0\}$, nicht regulär
S akz. $\emptyset$, regulär

Entscheider für $REGULAR_{\text{TM}}$

Wäre H ein Entscheider für $REGULAR_{\text{TM}}$, könnte man daraus einen Entscheider für A_{TM} konstruieren

Eigenschaften von Sprachen und Satz von Rice

Eigenschaften einer Sprache

Eigenschaften Turing-erkennbarer Sprachen

REGULAR	$L(M)$ ist regulär
E	$L(M)$ ist leer

Definition

Eine Eigenschaft P Turing-erkennbarer Sprachen heisst nichttrivial, wenn es zwei Turing-Maschinen M_1 und M_2 gibt mit

$L(M_1)$ hat Eigenschaft P
$L(M_2)$ hat Eigenschaft P nicht

Satz von Rice

Ist P eine nichttriviale Eigenschaft Turing-erkennbarer Sprachen, dann ist

$$P_{\text{TM}} = \{\langle M \rangle \mid L(M) \text{ hat Eigenschaft } P\}$$

nicht entscheidbar

Folgerung

Es ist nicht möglich, einem Programm anzusehen, ob die akzeptierte Sprache eine nichttriviale Eigenschaft hat

Reduktion für P_{TM}: $A_{\text{TM}} \leq P_{\text{TM}}$

A_{TM} ist nicht entscheidbar

Akzeptiert die Maschine M das Wort w?

$\langle M, w \rangle \quad \mapsto$

M akzeptiert $w \quad \Leftrightarrow$
M akzeptiert w nicht $\Leftrightarrow$

Ist P_{TM} entscheidbar?

Hat die Sprache $L(M)$ die Eigenschaft P?

Annahmen

- Σ^* hat die Eigenschaft P
- Testprogramm T: $L(T)$ hat P nicht

Programm S mit Input u

1. Programm T akzeptiert u: q_{accept}
2. M auf w laufen lassen
3. M akzeptiert w: q_{accept}
4. q_{reject}

S akzeptiert Σ^*, hat P
S akzeptiert $L(T)$, hat P nicht

Entscheidbare Probleme

Problem	Wort	Bedingung		Entscheidungsalgorithmus/Grund
E_{DEA}	$\langle A \rangle$	$L(A) = \emptyset$	ja	Minimalautomat hat keinen Akzeptierzustand
E_{CFG}	$\langle G \rangle$	$L(G) = \emptyset$	ja	Chomsky-Normalform
E_{TM}	$\langle M \rangle$	$L(M) = \emptyset$	nein	
EQ_{DEA}	$\langle A_1, A_2 \rangle$	$L(A_1) = L(A_2)$	ja	Vergleich der minimalen Automaten
EQ_{CFG}	$\langle G_1, G_2 \rangle$	$L(G_1) = L(G_2)$	nein	
EQ_{TM}	$\langle M_1, M_2 \rangle$	$L(M_1) = L(M_2)$	nein	
A_{DEA}	$\langle A, w \rangle$	$w \in L(A)$	ja	Regex-Engines simulieren beliebige DEAs auf beliebigen Input-Wörtern w
A_{CFG}	$\langle G, w \rangle$	$w \in L(G)$	ja	Cocke-Younger-Kasami deterministischer Parse-Algorithmus
A_{TM}	$\langle M, w \rangle$	$w \in L(M)$	nein	Halteproblem

Entscheidbar → Effizient

Nicht entscheidbar
Eine Sprache L ist nicht entscheidbar, wenn es keinen Entscheider M gibt mit $L(M) = L$.

Beispiele
- A_{TM}
- E_{TM}
- $HALT_{\text{TM}}$
- $REGULAR_{\text{TM}}$
- P_{TM}

interessant, aber nur selten von Interesse für Anwendungen

Ineffizient
Entscheidbare Probleme mit exorbitant langer Laufzeit, sind praktisch auch nicht lösbar

Was heisst schnell?
- Laufzeit hängt von der Problemgrösse n ab: $t(n) = \max\{t(w) \,|\, |w| = n\}$
- Schnell: $t(n) = O(n), O(n^2), O(\log(n)) \ldots$
- Hardwareabhängig!

Hardwareeinfluss

Deterministische Hardware

Hardwareänderung	Simulationszeit
keine	$t(n)$
Alphabet	$O(t(n))$
Spuren	$O(t(n))$
Bänder	$O(t(n)^2)$

Gemeinsam
Verschiedene Hardware M_1 und M_2, gleicher Algorithmus, Laufzeiten $t_i(n)$:

$$O(t_1(n)) = O(t_2(n)^k), \quad k > 0$$

Nichtdeterministische Hardware
Kann in Zeit $2^{O(t(n))}$ simuliert werden ⇒ NTM ist exponentiell schneller

Unerreichbar
Die schnellste NTM ist *immer* schneller als jede TM, die L entscheiden kann

Definition
Eine Sprache L gehört zur Klasse NP, wenn L von einer nichtdeterministischen TM in polynomieller Zeit entschieden werden kann, $t(n) = O(n^k)$

Polynomielle Reduktion

Polynomieller Laufzeit-Vergleich
Seien A und B entscheidbare Sprachen. Eine berechenbare Abbildung

$$f\colon \Sigma^* \to \Sigma^* : w \mapsto f(w)$$

mit
1. $w \in A \Leftrightarrow f(w) \in B$ (Reduktion)
2. $f(w)$ ist berechenbar in polynomieller Zeit in $|w|$

heisst *polynomielle Reduktion* $A \leq_P B$. Lies: A ist polynomiell leichter entscheidbar als B.

Satz
Sei $A \leq_P B$. Dann gilt

$$B \in P \Rightarrow A \in P, \qquad A \notin P \Rightarrow B \notin P$$
$$B \in NP \Rightarrow A \in NP, \quad A \notin NP \Rightarrow B \notin NP$$

Beispiel
Stundenplan $\leq_P$ k-VERTEX-COLORING:

Zeitfenster ↦ Farbe
Anzahl Zeitfenster ↦ k
Fach ↦ Vertex
Anmeldung ↦ Kante

Anmeldung eines Stunden auf zwei Fächer ↦ Kante zwischen Vertizes

Stundenplan und k-VERTEX-COLORING

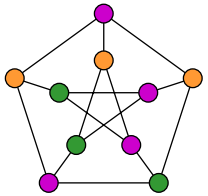
Stundenplan ↔ k-VERTEX-COLORING

Gegeben
- Fächer
- Anmeldungen von Studenten für Fächer
- k Zeitslots pro Woche

Gesucht
Stundenplan = Zuordnung von Zeitslots zu Fächern derart, dass jeder Student jedes angemeldet Fach besuchen kann

Gegeben
- Graph G
- k Farben

Gesucht
Kann man die Knoten so mit k Farben einfärben, dass benachbarte Knoten verschiedene Farbe haben?



Sudoku

9	5	4	1	3	7	6	8	2
2	7	3	6	8	4	1	9	5
1	6	8	2	9	5	7	3	4
4	9	5	7	2	8	3	6	1
6	8	1	4	5	3	2	7	9
3	2	7	9	6	1	5	4	8
7	4	9	3	1	2	8	5	6
5	1	6	8	7	9	4	2	3
8	3	2	5	4	6	9	1	7

Entscheidungs-Aufgabe
Gibt es Zeichen $1–n^2$ für jedes der $n^2 \times n^2$ Felder derart, dass die Zeichen pro Zeile, Spalte und Unterquadrat verschieden sind?

Lösung
Backtracking → exponentielle Laufzeit (in n)

Beobachtung
- Lösen ist schwierig
- Lösung verifizieren ist einfach

P, NP und polynomielle Verifizierer

Verifizierer
Ein *polynomieller Verifizierer* für die Sprache L ist eine Turingmaschine V, so dass es für jedes $w \in \Sigma^*$ ein Wort c (das *Lösungszertifikat*) gibt, für das gilt

$$w \in L \quad \Leftrightarrow \quad V \text{ akzeptiert } \langle w, c \rangle$$

Die Laufzeit von V ist polynomiell in $|w|$.

Komplexitätsklasse NP
Eine Sprache ist in NP, wenn sie von einer nichtdeterministischen Turing-Maschine in polynomieller Zeit entschieden werden kann.

Satz
Eine Sprache ist genau dann in NP, wenn sie in polynomieller Zeit verifiziert werden kann.

Verifizierer für: _____

Entscheidbar?

Zertifikat?

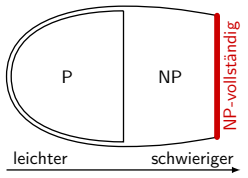
Verifikationsalgorithmus

Nr.	Was ist zu tun?	Aufwand
1		
2		
3		
4		
5		
	Total	

NP-vollständig

Definition
Eine entscheidbare Sprache B heisst *NP-vollständig*, wenn sich jede Sprache A in NP polynomiell auf B reduzieren lässt:

$$A \leq_P B \quad \forall A \in \text{NP}$$



Äquivalenz
NP-vollständige Probleme sind alle gleich schwierig:

$$A, B \text{ NP-vollständig} \Rightarrow \begin{cases} A \leq_P B \\ B \leq_P A \end{cases}$$

Satz
$$\left.\begin{array}{l} A \text{ NP-vollständig} \\ B \in NP \\ A \leq_P B \end{array}\right\} \Rightarrow B \text{ NP-vollständig}$$

P = NP?
Falls P ≠ NP, dann können NP-vollständige Probleme nicht in polynomieller Zeit gelöst werden.

Polynomielle Ausfüllrätsel

Definition
Ein *polynomielles Ausfüllrätsel* ist eine $n \times m$-Tabelle, in die Zeichen eines Alphabets Σ eingefüllt werden müssen, so dass als logische Formel ausdrückbare Regeln eingehalten werden, die in polynomieller Zeit ausgewertet werden können.

Satz
Jedes polynomielle Ausfüllrätsel A lässt sich polynomiell auf SAT *reduzieren.*

Beispiel
$n^2 \times n^2$-Sudoku, $\Sigma = \{1, \ldots, n^2\}$.

Variablen
$$x_{ijc} = \text{wahr} \Leftrightarrow \begin{cases} \text{Feld } (i,j) \text{ der Tabelle} \\ \text{enthält Zeichen } c \end{cases}$$

Genau ein Zeichen im Feld (i, j)
$$\varphi_{ij} = \bigvee_{c \in \Sigma} \Big(x_{ijc} \wedge \neg \bigvee_{d \neq c} x_{ijd} \Big)$$

c und kein anderes Zeichen in (i, j)

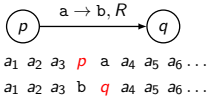
Regeln
Spielregeln als Formel φ in Variablen x_{ijc} ausdrücken, die wahr ist genau dann, wenn die Regeln erfüllt sind.

Berechnungsgeschichte

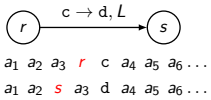
Beschreibung des Zustands der TM

- Bandinhalt als ein Zeichenkette von Bandalphabet-Zeichen
- Position und Zustand des Schreibe-/Lese-Kopfes: Zustand wird vor die aktuelle Position geschrieben:

Übergang R



Übergang L



Berechnungsgeschichte: Beschreibung der Berechnung einer TM

Folge von Zustands-Zeilen

Cook-Levin

Satz

SAT ist NP-vollständig

Was das bedeutet

A eine Sprache in NP.
$\Rightarrow$ Es gibt eine nichtdeterministische TM M, die A in polynomieller Zeit $O(t(n))$ entscheidet.
$\Rightarrow$ A kann polynomiell auf *SAT* reduziert werden: $A \leq_P SAT$

Plan

Beschreibe das Finden der Berechnungsgeschichte von M als ein polynomielles Ausfüllrätsel

Beweis

1. Zeichen der Berechnungsgeschichte stammen aus $Q \cup \Gamma$
2. Berechnungsgeschichte kann in eine $(n + t(n)) \times t(n)$ Tabelle gefüllt werden.
3. Erste Zeile: Blanks ($\sqcup$), Anfangszustand q_0 und Inputwort w
4. Letzte Zeile: enthält q_{accept}
5. Mögliche Übergänge durch logische Formeln über alle 2×3-Rechtecke in der Tabelle beschreiben (polynomiell)
6. Finden der Berechnungsgeschichte ist ein polynomielles Ausfüllrätsel.

$SAT \leq_p 3SAT$

SAT ist NP-vollständig

$$\{\varphi \mid \varphi \text{ erfüllbare logische Formel}\}$$

3SAT ist NP-vollständig

$$\left\{ \varphi \;\middle|\; \begin{array}{l} \varphi \text{ erfüllbare logische Formel in} \\ \text{konjunktiver Normalform mit 3} \\ \text{Variablen pro Klausel (3CNF)} \end{array} \right\}$$

Polynomielle Reduktion

Wandle $\varphi \in SAT$ um in eine erfüllbare Formel in 3CNF um.

Äquivalenzumformung

Erzeugt exponentiell viele Terme
$\Rightarrow$ keine polynomielle Reduktion

Erfüllungsäquivalenz

Umformung mit Hilfe von Erfüllungsäquivalenz (Übung 1) ist polynomiell (Details: Skript 7.6.1)

$3SAT \leq_P k\text{-CLIQUE}$

Gegeben

Formel φ in konjunktiver Normalform mit 3 Variablen pro Klausel

Frage

Ist φ erfüllbar?

$$(x_1 \vee x_2 \vee x_3)$$

$$\varphi$$

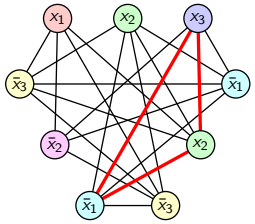
$$\wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (x_1 \vee x_2 \vee x_3)$$

Gegeben

Graph G

Gesucht

Gibt es k Knoten, die alle miteinander verbunden sind (k-Clique)

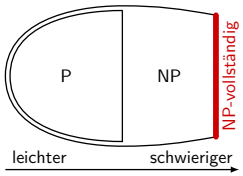


NP-Vollständigkeit

Wenn ein Problem NP-vollständig ist:

- Lösung braucht typischerweise exponentielle Zeit
- Korrektheit der Lösung ist leicht (in polynomieller Zeit) zu prüfen
- Skalierung ist schlecht (exponentiell)
- $\Rightarrow$ Problemstellung modifizieren (zusätzliche Daten, Approximation)

NP-Welt



Bekannt als NP-vollständig

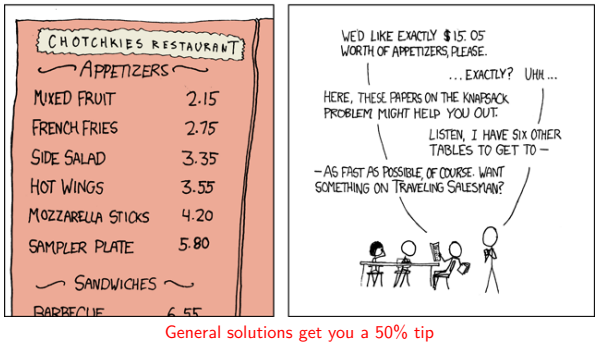
1. *SAT* / *3SAT*
2. *k-VERTEX-COLORING*
3. *k-CLIQUE*

$\Rightarrow$ wir brauchen mehr

Reduktionsprinzip

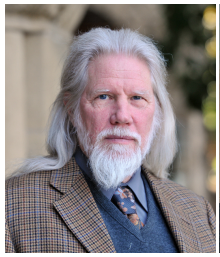
Wenn

1. A NP-vollständig



General solutions get you a 50% tip

Asymmetrische Kryptographie



Whitfield Diffie

Martin Hellman

Ralph C. Merkle

Öffentliches Schlüsselsystem (1978)

Privater Schlüssel

Stark wachsender Rucksack:

$$A = (1, 2, 4, 9, 20, 38)$$

Multiplikator $n = 31$, Modulus $q = 105$ (teilerfremd), $q > \sum_{a \in A} a$

Öffentlicher Schlüssel

$b_i = a_i \cdot n \mod q$:

$$B = (31, 62, 19, 69, 95, 23)$$

Verschlüsselung

Klartext: $d = 011010$

$$c = 0 \cdot 31 + 1 \cdot 62 + 1 \cdot 19 + 0 \cdot 69 + 1 \cdot 95 + 0 \cdot 23 = 176$$

Entschlüsselung

1. m Inverse von n mod q: $nm \equiv 1 \mod q$:
$$61 \cdot 31 = 1891 \equiv 1 \mod 105$$
2. Ciphertext mit m multiplizieren:
$$c \cdot m = 176 \cdot 61 = 26 \mod q$$
3. Lösung des stark wachsenden Rucksackproblems (einfach):
$$26 = 20 + 4 + 2 \quad \Rightarrow \quad d = 011010$$

Sicherheit

Allgemeines Rucksackproblem: NP-vollständig

$3SAT \leq_P SUBSET\text{-}SUM$

3SAT

$$\varphi = \overbrace{(x_1 \vee x_2 \vee x_3)}^{c_1} \wedge \overbrace{(\bar{x}_1 \vee x_2 \vee \bar{x}_3)}^{c_2} \wedge \overbrace{(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)}^{c_3}$$

SUBSET-SUM

$$\langle S = (y_i, z_i, g_k, h_k | i \leq l, k \leq n), t \rangle$$

Lösung $T \subset S$: $\sum_{s \in T} s = t$

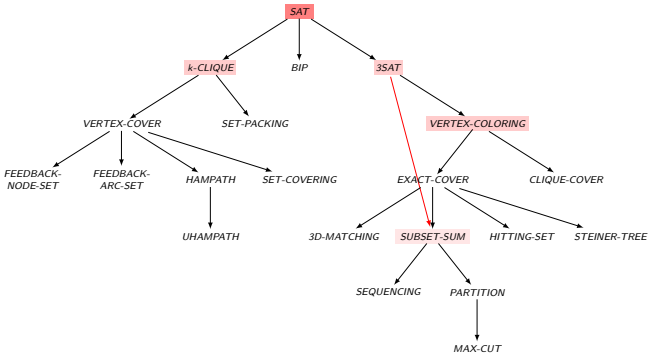
- $y_i \in S \Rightarrow x_i$ wahr
- $z_i \in S \Rightarrow x_i$ falsch
- g_i, h_i: Füller, maximal zwei pro Klausel-Spalte

Folgerung

φ erfüllbar $\Leftrightarrow \langle S, t \rangle$ lösbar

Zahl	x_1	x_2	x_3	c_1	c_2	c_3
y_1	1	0	0	1	0	0
z_1	1	0	0	0	1	1
y_2		1	0	1	1	0
z_2		1	0	0	0	1
y_3			1	1	0	0
z_3			1	0	1	1
g_1				1	0	0
h_1				1	0	0
g_2					1	0
h_2					1	0
g_3						1
h_3						1
t	1	1	1	3	3	3

Karp-Katalog



Fragen und Aufgaben

1. In welche Probleme geht eine ganze Zahl $k \in \mathbb{Z}$ ein?
2. In welchen Problemen geht es um eine Aufteilung in zwei Teilmengen? Wie unterscheiden sie sich?
3. Unterschied zwischen *HITTING-SET* und *EXACT-COVER*?
4. Unterschied zwischen *VERTEX-COVER*, *CLIQUE-COVER*, *EXACT-COVER* und *SET-COVERING*?
5. In welchen Problemen kommt die Zahl 3 vor?
6. Unterschied zwischen *SET-PACKING* und *SET-COVERING*?
7. Unterschied zwischen *FEEDBACK-NODE-SET* und *FEEDBACK-ARC-SET*?
8. Unterschied zwischen *HAMPATH* und *UHAMPATH*?
9. Entwerfen Sie für jedes Problem eine einprägsame Visualisierung.

TM und moderne Computer

Turing Maschine

1. Zustände Q
2. Band, d. h. ein unendlich grosser Speicher
3. Schreib-/Lesekopf
4. Anhalten, q_{accept} und q_{reject}
5. problemspezifisch

Moderner Computer

1. Zustände der CPU: Statusbits, Registerwerte
2. virtueller Speicher: praktisch unbegrenzter Speicher
3. Adress-Register, Programm-Zähler
4. `exit(EXIT_SUCCESS)`, `exit(EXIT_FAILURE)`
5. kann beliebige Programme ausführen

Die universelle Turing-Maschine

Vorläufer

Charles Babbage, Ada Lovelace: Analytical Engine

Alan Turing:

Es gibt eine Turing-Maschine, die jede beliebige andere Turing-Maschine simulieren kann.

Konstruktion

- Eigenes Band für eine Codierung der Übergangsfunktion $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$
- Eigenes Band für den aktuellen Zustand
- Arbeitsband
- Simulation dieser Maschine auf einer Standard-TM

Alan Turing (1936): *On Computable Numbers, With an Application to the Entscheidungsproblem*

Dinge, die einer TM fehlen

Ist eine Komponente wesentlich?

$\Leftrightarrow$ Komponente verändert die Fähigkeiten einer TM oder die Komplexität auf nicht-polynomielle Weise

Persistenter Speicher

- Files nicht unterscheidbar von Daten, die bereits irgendwo im Speicher liegen
- Memory Mapped Files

Interaktion

Lösung eines vollständig spezifizierten Problems braucht keine Interaktion

Input/Output

- Output: nicht wesentlich
- Input "existiert nicht":
  - Programm wird angehalten
  - Kontrolle geht an den Kernel
  - Kernel transferiert Daten in den Speicher
  - Kontrolle geht zurück an den Prozess

Resultat: Input-Daten stehen irgendwo auf dem Band

Vergleich von Maschinen

Beispiele

- Eine Mehrband-Turingmaschine kann auf einer Mehrspurmaschine *simuliert* werden
- Eine Mehrspur-Turingmaschine kann auf einer Maschine mit erweitertem Bandalphabet *simuliert* werden
- Eine Maschine mit grossem Alphabet kann auf einer Standard-Turingmaschine mit Alphabet $\Gamma = \{0, 1, \sqcup\}$ *simuliert* werden

Definition

Eine TM M_1 ist "leistungsfähiger" als eine TM M_2, wenn M_1 die Maschine M_2 simulieren kann

$$M_2 \leq_S M_1$$

lies: M_2 ist simulierbar auf M_1

Beispiele

- Zelluläre Automaten, Game of Life
- TM $\leq_S$ Minecraft, 8-bit CPU $\leq_S$ Minecraft $\leq_S$ TM

John Horton Conway

- * 26. Dezember 1937, Liverpool, England
- † 11. April 2020, New Brunswick, New Jersey, USA, an den Folgen von COVID-19
- 1968: Conway Gruppen
- 1970: Knotentheorie, Alexander-Conway-Polynom
- 1970: Game of Life
- 1985: *ATLAS of Finite Groups*
- 2004: Free will theorem: *if experimenters have free will, then so do elementary particles*

Game of Life

Zustände

Zellen sind tot oder lebend

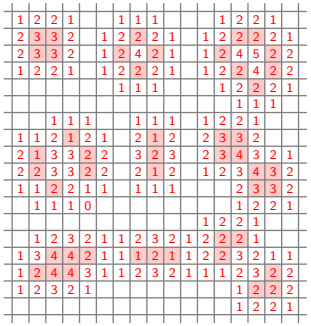
Regeln

n Anzahl lebender Nachbarn

1. Zelle tot, $n = 3$: Zelle neu geboren
2. Zelle lebend
   - $2 \leq n \leq 3$: überlebt
   - sonst: stirbt

Strukturen

- stabil, oszillatorisch
- "Raumschiffe", mobile Strukturen
- beliebige Gatter, **universelle TM**



Programmiersprachen und Turing-Vollständigkeit

Definition

Eine Sprache A heisst eine *Programmiersprache*, wenn es eine Abbildung

$$c : A \mapsto \Sigma^*$$

wobei $c(w)$ ein Programm für eine universelle Turing-Maschine ist.

Definition

Eine Programmiersprache heisst *Turing-vollständig*, wenn in ihr jede beliebige Turing-Maschine simuliert werden kann.

Frage

Gibt es einen in A geschriebenen Turing-Maschinen-Simulator?

Programmiersprache C

Frage

Gibt es einen in C geschriebenen Turing-Maschinen-Simulator?

Antwort

Programm `turing.c` in Github, Verzeichnis `code/turing`

(Demo)

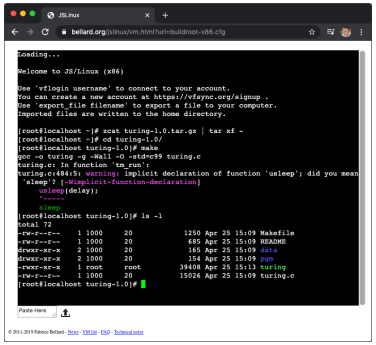
Javascript

Frage

Kann man jede beliebige Turing-Maschine in Javascript ausführen?

Antwort

1. Start den x86-Emulator von Fabrice Bellard: `http://bellard.org/jslinux/`
2. Boote Linux
3. Kompiliere den Turing-Maschinen-Simulator in C
4. ...



Sprachelemente

Grundelemente für Sprachen

- Konstanten c mit natürlichen Werten $0, 1, \ldots, 1291, \ldots$
- Variablen x_0, x_1,... mit Werten in $\mathbb{N}$
- Zuweisungsoperationen $x_i := c$
- Addition $x_i := x_j + c$
- Subtraktion $x_i := x_j - c$, ist $c > x_j$, ist 0 der neue Wert von x_i

Achtung:

- Keine Addition $x_i + x_j$
- Keine Subtraktion, Multiplikation, Division

LOOP

Kontrollstruktur

Führe ein Programm P genau x_i mal aus:

```
LOOP x_i DO
    P
END
```

Bedingte Anweisung

```
IF x_i THEN
    P
END
```

implementieren als

```
y := 0
LOOP x_i DO y := 1 END
LOOP y DO P END
```

Addition/Subtraktion

$x_i := x_j \pm x_k$

kann implementiert werden als

```
x_i := x_j
LOOP x_k DO
    x_i := x_i ± 1
END
```

Multiplikation

$x_i := x_j * x_k$

kann implementiert werden als

```
x_i := 0
LOOP x_k DO
    x_i := x_i + x_j
END
```

bzw.

```
x_i := 0
LOOP x_k DO
    LOOP x_j DO
        x_i := x_i + 1
    END
END
```

Lösung des Haltetheorems

Theorem

Jedes LOOP-Programm hält an. Insbesondere ist LOOP nicht Turing-vollständig.

Beweis.

Vollständige Induktion nach der Anzahl n der geschachtelten LOOP-Strukturen

- **Induktionsverankerung:** Programme ohne LOOP ($n = 0$) halten an
- **Induktionsannahme:** Programme mit bis zur Tiefe $n - 1$ geschachtelten LOOP-Strukturen halten an
- **Induktionsschritt:** Im Programm

  LOOP x_i DO P END

  mit höchstens n geschachtelten LOOPs ist P von der Art wie in der Annahme. Nach Induktionsannahme terminiert P. Das Programm führt P genau x_i-mal aus, P terminiert jedesmal. Also terminiert das Programm. □

FreeRadius `unlanguage`

- Konfigurationssprache für den FreeRadius Server
- Soll **nicht** Turing-vollständig sein
- `if ( ... ) {...} else {...}`
- `foreach &Attribute-Reference { ... }`
- Keine unlimitierten Schleifenkonstruktionen

⇒ Erweiterungsmodule im FreeRadius-Server halten immer an (wie LOOP)

⇒ `unlang` nicht Turing-vollständig

WHILE

Definition der Sprache WHILE

- Grundelemente wie in LOOP
- "Bedingte" Schleifenkonstruktion:

  WHILE $x_i > 0$ DO P END

  führt P so lange aus, bis $x_i = 0$

IF in WHILE

IF x_i THEN P END

kann in WHILE implementiert werden:

```
y := x_i
WHILE y > 0 DO
    P; y := 0;
END
```

LOOP in WHILE

Die Schleifenkonstruktion von LOOP

LOOP x_i DO P END

kann implementiert werden in WHILE

```
y := x
WHILE y > 0 DO
    P
    y := y - 1
END
```

Erweiterung

WHILE ist eine echte Erweiterung von LOOP

GOTO

Definition der Sprache GOTO

- Grundelemente wie in LOOP
- Markierte Folge von Anweisungen

$$M_1: A_1$$
$$M_2: A_2$$
$$\ldots: \ldots$$
$$M_k: A_k$$

Bedingte Anweisung

M_{k+1} : IF $x_i = c$ THEN GOTO M_{k+n}
$\ldots: \ldots$
$M_{k+n}: A_l$

Unbedingte Anweisung

$M_{k+1}: x_i := c$
M_{k+2} : IF $x_i = c$ THEN GOTO M_{k+n}
$\ldots: \ldots$
$M_{k+n}: A_l$

abgekürzt:

M_{k+1} :
M_{k+2} : GOTO M_{k+n}
$\ldots: \ldots$
$M_{k+n}: A_l$

Implementation von GOTO in WHILE

GOTO-Programm:

$M_1: A_1$
$M_2: A_2$
$\ldots: \ldots$
$M_k: A_k$

implementiert in WHILE:

```
z := 1
WHILE z > 0 DO
IF z = 1 THEN A'_1 END
IF z = 2 THEN A'_2 END
:
IF z = k THEN A'_k END
IF z = k + 1 THEN z := 0 END
END
```

Modifizierte Anweisung A'_i

- A_i eine bedingte Sprunganweisung

  M_l : IF $x_i = c$ THEN GOTO M_j

  wird übersetzt in A'_i

  IF $x_i = c$ THEN $z := j$ ELSE $z := z + 1$ END

- A_i eine gewöhnliche Anweisung

  $M_i : A_i$

  wird übersetzt in A'_i:

  A_i; $z = z + 1$;

Implementation von WHILE in GOTO

Übersetzung

Ein WHILE-Konstrukt

$$\texttt{WHILE } x_i > 0 \texttt{ DO } P \texttt{ END}$$

wird übersetzt in ein GOTO-Code-Segment

M_l : IF $x_i = 0$ THEN GOTO M_{l+3}
M_{l+1} : P
M_{l+2} : GOTO M_l
M_{l+3} :

Äquivalenz

Folgerung: Die Sprachen WHILE und GOTO sind äquivalent

Turing-Vollständigkeit

Ein Turing-Maschinen-Simulator kann in GOTO implementiert werden
⇒ GOTO ist Turing-vollständig
⇒ WHILE ist Turing-vollständig

Sind WHILE und GOTO Turing-vollständig?

Was ist zu tun?

1. Schreibe einen Compiler, der TM-Beschreibungen in GOTO-Programme übersetzt
2. Schreibe eine virtuelle Maschine, die GOTO-Programme ausführen kann

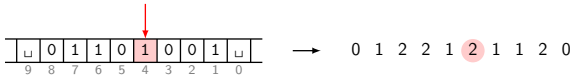
⇒ GOTO kann jede beliebige TM simulieren

⇒ GOTO ist Turing-vollständig

⇒ WHILE ist Turing-vollständig

Realisierung des Bandes

Idee



Band mit Alphabet $\Gamma = \{0, 1, ␣\}$ — Zahl im 4er-System

Variablen

Band: b

Kopfposition: $h = 4^k$, k die Feldnummer

Feldinhalt: z

Beispiel

$b = 122121120_4$

$k = 4$,
$h = 256 = 1000_4$

$z = b/h \mod 4 = 1$

Arithmetische (!) Operationen

- Feldinhalt lesen: $z = b/h \mod 4$
- Feld löschen: $b = b - z \cdot h$
- Feld beschreiben: $b = b + x \cdot h$
- Kopfbewegung links: $h = 4h$
- Kopfbewegung rechts: $h = h/4$

JSFuck

Javascript ist Turing-vollständig, aber viel zu durchschaubar:

```
alert("Hello world!");
```

Verbesserung: JS-Programme codieren mit nur 6 Zeichen `[]()!+`

1. Weak typing ⇒ implizite Typkonvertierung: `+[]` ist die Zahl 0, `+!+[]` ist die Zahl 1
2. `"a" = "false"[1] = (false+[])[1] = (![]+[])[1] = (![]+[])[+!+[]]`
3. Beliebige Strings durch Verkettung
4. `alert(1) = Function("alert(1)")() = []["filter"]["constructor"]("alert(1)")()`

⇒ Jedes beleibige JS-Programm lässt sich mit `[]()!+` codieren

BrainFuck

Brainfuck	C-Äquivalent
`>`	`++ptr;`
`<`	`--ptr;`
`+`	`++*ptr;`
`-`	`--*ptr;`
`.`	`putchar(*ptr);`
`,`	`*ptr = getchar();`
`[`	`while (*ptr) {`
`]`	`}`

1. ∞-grosser Speicher
2. bedingter Sprung
3. unbeschränkte Schleife

⇒ Turing-vollständig

Ook Ook

Ook		Brainfuck
Ook.	Ook?	>
Ook?	Ook.	<
Ook.	Ook.	+
Ook!	Ook!	-
Ook!	Ook.	.
Ook.	Ook!	,
Ook!	Ook?	[
Ook?	Ook!	]